CS 450

— The Last Lecture —

Alonzo Church, Lambda Calculus, and the Birth of Theoretical Computer Science

Carl D. Offner

Contents

1 The foundations of theoretical computer science in the 1930's **1**

2 Church's lambda calculus: arithmetic **2**
2.1 Lambda calculus and functional programming 2
2.2 Church numerals 3
2.3 Arithmetic on Church numerals 4
2.4 How Church wrote this 5
2.5 More arithmetic 6

3 Fixed point theorems **7**

4 Church's lambda calculus: recursion **10**
4.1 The problem of representing a recursive function 10
4.2 The Y operator 13
4.3 How to write a recursive function 14
4.4 Is this too good to be true? 15
4.5 One final word 16

A Appendix: An example of how the `Y` operator works **17**

1 The foundations of theoretical computer science in the 1930's

Alonzo Church (1903–1995) was a professor of mathematics at Princeton University, arriving there in 1929. In the 1930's he, together with his students John Barkley Rosser and Stephen Cole Kleene (both of whom became famous in their own right) developed the *lambda calculus*, which forms the basis for functional programming languages, including Scheme. It was a remarkable period: also at Princeton (or at the Institute for Advanced Study next door) were John von Neumann, Kurt Gödel, and Alan Turing.

Church and his students were, however not really interested in what we would now call programming languages. And they really couldn't have been, because the first widely known computer—the

ENIAC, containing 20,000 vacuum tubes—didn't even appear until 1946, and there was only one of them. Further, it was programmed in its own machine language. Programming languages like Fortran, Cobol, and Lisp did not appear till the mid- to late-1950's.

What these people were really interested in was the question of what could *in principle* be computed.

This may seem like a strange question, but it was actually very much in the air. Around the turn of the century mathematicians—led by David Hilbert, the greatest mathematician of his day, and one of the greatest ever—had for a time been convinced that all of mathematics could be put on a complete and purely logical axiomatic foundation. In 1913, Alfred North Whitehead and Bertrand Russell published their monumental work *Principia Mathematica*, which in over 1800 pages went a fair distance in that direction.

But then in 1931, Kurt Gödel proved that in any axiomatic system that was powerful enough to support ordinary arithmetic, there would of necessity exist statements expressible in that system (that is, in the language of the system) that could neither be proved nor disproved using only the axioms of the system.

And subsequently there have been quite straightforward mathematical questions, such as whether certain ordinary looking algebraic equations have solutions that are integers, that have been shown to fall into this category. That is, there is, even in principle, no algorithm that can take one of these equations as input and return true or false depending on whether or not that equation has a solution in integers[1].

While this last result was not known until many years later, the question in general was a hot topic, and a number of different people worked on it: *What in principle could be computed?* The techniques they developed to answer this question form the foundation of what is now regarded as theoretical computer science.

2 Church's lambda calculus: arithmetic

2.1 Lambda calculus and functional programming

Church's lambda calculus could be thought of as a programming language, although of course the term "programming language" didn't even exist at the time. As such, it amounts to a really stripped-down version of Scheme. Pretty much the only thing it contains is lambda expressions.

Another way of thinking about this is that Church was defining what was meant by "functional programming". Again, this term didn't exist either at the time. But looking back at things from today's perspective, this is not a bad way to think of what he was doing.

Church's lambda calculus is a functional language in the sense that there is nothing in it but functions.

Now this may seem very strange indeed. We are certainly used to computer programming languages that contain functions—in fact, anything above the level of assembly language pretty much has to contain functions. But those functions have to operate on something, right? And almost always, we understand that they are operating on some sort of objects of various types. And those types

[1] This is the famous Davis-Putnam-Robinson-Matiyasevich theorem.

are ultimately defined in terms of some primitive types of the language—integers, for instance, or characters, or maybe strings.

The problem with that, from our point of view, is that this makes the language hard to reason about. Suppose we want to have objects of integer type. Well then, we have to decide what an integer is, and how it is represented. We know, of course, that in any particular computer, an integer is represented as a sequence of 0's and 1's, and these in turn correspond to patterns of electric charge in the semiconductors in the processor chip. Further, that sequence has a certain length, so we have to worry about things like integer overflow and the like. And if we want to deal with more complicated objects like floating-point numbers, the representation becomes more complex, and there are many more things that can go wrong. In fact, modern specifications, like the IEEE floating-point standard, are quite complex for just this reason. This makes any real computer language difficult or even impossible to reason definitively about, and the whole point of Church's lambda calculus was to provide an extremely simple and bare-bones language in which we could reason both simply and effectively about what could be computed.

So Church didn't have any special integer types, or any other types. All he had was functions.

And so you might ask: what in the world did these functions operate on?

Well, since all he had was functions, they could only operate on other functions.

2.2 Church numerals

And at this point, you might be thinking, "This is nonsense. At least we have to have the ability to operate on ordinary integers like 0, and 1, and 2, and so on."

Well, that's true. And Church certainly understood that. What he together with his students did was this: they gave a way that certain functions could be identified with what we regard as the natural numbers (i.e., the non-negative integers 0, 1, 2, ...).

Again, you may be thinking that this is just bizarre. After all, functions aren't numbers. But actually, sequences of electric charges in a processor chip aren't numbers, either. Yet they perform perfectly well as numbers in computers, and we normally don't even have to think of the underlying physics, unless we are chip designers. And for what we're doing now, we're not chip designers—we're software engineers. So we will think in terms of software.

The way Church got functions to represent integers was this:

The number n will correspond to the function f_n which takes as its argument any other function g and produces a function that amounts to applying g n times. In mathematical notation, we could write something like this:

$$f_n(g) = \underbrace{g \circ g \circ \cdots \circ g}_{n\ g\text{'s}}$$

or equivalently,

$$f_n(g)(x) = \underbrace{g \circ g \circ \cdots \circ g}_{n\ g\text{'s}}(x)$$

In fact, we won't use the notation f_n. To emphasize that this function is for all practical purposes the "same as" the number n, we will denote it by $\overline{n}$. Thus we have

$$\overline{n}(g) = \underbrace{g \circ g \circ \cdots \circ g}_{n\ g\text{'s}}$$

The functions $\overline{n}$ are called *Church numerals*.

2.3 Arithmetic on Church numerals

Well that's all very nice, but really numbers don't mean much if we can't do arithmetic with them. And in fact, we can. The first step is to create a *successor* function, which we will denote by S. The idea is that

$$S(\overline{n}) \text{ should be the function } \overline{n+1}$$

(You have to be careful when reading this. $\overline{n}$ and $\overline{n+1}$ are Church numerals. But n and $n+1$ are our ordinary numbers.)

How can we define the function S so this is true? Here's how: we define

$$\begin{aligned} S(\overline{n})(g)(x) &= \big(g \circ \overline{n}\big)(g)(x) \\ &= g \circ \overline{n}(g)(x) \qquad \text{(that's the way procedures combine)} \end{aligned}$$

In other words, $S(\overline{n})$ is a function that takes g and applies it n times to its argument, and then applies it once more. And that of course amounts to applying it $n+1$ times, which is exactly what the successor function should do.

Once we have the successor function, we define addition of Church numerals, like this:

$$\text{plus}(\overline{n}, \overline{m}) = \big(\overline{n}S\big)\overline{m}$$

This works because with this definition, we have

$$\begin{aligned} \text{plus}(\overline{n}, \overline{m})(g)(x) &= \big(\overline{n}S\big)\overline{m}(g)(x) \\ &= \Big(\underbrace{S \circ S \circ \cdots \circ S}_{n\ S\text{'s}} \circ \overline{m}\Big)(g)(x) \\ &= \Big(g \circ \underbrace{S \circ S \circ \cdots \circ S}_{n-1\ S\text{'s}} \circ \overline{m}\Big)(g)(x) \\ &= \cdots \\ &= \Big(\underbrace{g \circ g \circ \cdots g}_{n\ g\text{'s}} \circ \overline{m}\Big)(g)(x) \\ &= \underbrace{g \circ g \circ \cdots \circ g}_{n+m\ g\text{'s}}(x) \end{aligned}$$

and since this is the application of the function g $n + m$ times to its argument, it is exactly what we mean by $\overline{n+m}(g)(x)$.

This gets us on our way. We could also define multiplication, like this:

$$\text{mult}(\overline{n}, \overline{m}) = \overline{n}(\overline{m})$$

That is, with this definition, we have

$$\begin{aligned}
\text{mult}(\overline{n}, \overline{m})(g)(x) &= \overline{n}(\overline{m})(g)(x) \\
&= \overline{n}\big(\overline{m}(g)\big)(x) && \text{(that's the way procedures combine)} \\
&= \underbrace{\overline{m}g \circ \overline{m}g \circ \cdots \circ \overline{m}g}_{n\ \overline{m}g\text{'s}}(x)
\end{aligned}$$

which applies g nm times to its argument x, and that's exactly what we want.

2.4 How Church wrote this

Church didn't actually use symbols like $\overline{n}$, and in fact even though we are careful to distinguish between $\overline{n}$ and n, this can get a bit confusing. He used pure lambda calculus notation. We won't do that, but we *can* write these definitions as Scheme functions, using lambda expressions, which is very close to what Church actually did.

So for instance, we can define the Church numeral zero like this:

```
(define zero
  (lambda (g)
    (lambda (x)
      x)
    )
  )
```

That is, zero is a function that takes g as an argument and yields a function that when applied to x returns x. In other words, zero applies g 0 times to its argument, thereby leaving x unchanged. (Equivalently, zero turns any function g into the identity function.)

Then we can define the successor function

```
(define S
  (lambda (f)
    (lambda (g)
      (lambda (x)
        ((g (f g)) x)
        )
      )
    )
  )
```

The way to understand this is that S is operating on the function f, which we assume is a Church numeral. It produces a function which takes g as an argument and applies g "f" times and then once more to x. So (S zero) is going to be what we have been writing as $\overline{1}$, and (S(S zero)) is going to be what we have been writing as $\overline{2}$, and so on. This gives us all the Church numerals.

We could then go on and define addition:

```
(define plus
  (lambda (n m)
    (lambda (g)
      (lambda (x)
        ((((n S) m) g) x)
        )
      )
    )
  )
```

Written this way, this may seem pretty opaque, but it should be clear that it is exactly the same as what we were writing above, just with slightly different notation.

2.5 More arithmetic

Well, we can go on in this way. It should be at least plausible that we could also define exponentiation of Church numerals—and indeed we can. I won't bother reproducing it here, although it's not very complicated.

The difficulty comes with subtraction. It's not so easy to see how to define it. And in fact, even though Church himself hoped that his lambda calculus would be sufficient to model any computation, no one, including any of his students, was initially convinced of this. Then Kleene discovered how to model subtraction, and at that point Church was convinced that he had what we would now call a universal programming language.

I won't go into the details of how subtraction was implemented, but I'll just describe the idea. First of all, just as the non-negative integers can be represented by lambda expressions, it's also possible to represent truth values (i.e., "true" and "false") as expressions, and also to create an if-then-else expression. And using them, Kleene was able to define a "predecessor" function P. P is similar to the successor function S except that it returns the "previous" Church numeral. (And there is one wrinkle—since we are only dealing with non-negative numbers, the convention is that P applied to zero is again zero. But this is a minor point.)

Well, once we have the predecessor function, we can define subtraction by repeated applications of P, just as we defined addition by repeated applications of S.

And now we're really in business. We can do arithmetic on the non-negative integers, and once we can do that we can extend our model to include negative integers, and rationals, and lots of other things as well.

Now the point is not that any of this is at all efficient—it isn't. But it is quite remarkable that we have built up this programming language, and the possibility of all sorts of complex data structures

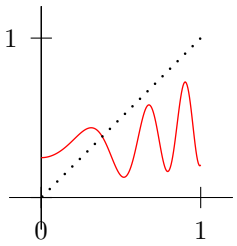

Figure 1: Graph of a function $f : [0, 1] \to [0, 1]$. The graph of the function is red. The dotted line is the graph of the function $g(x) = x$. The x-coordinate of the point at which the dotted line intersects the red curve is a fixed point of f.

by starting with the simplest notion—that of a lambda expression, and nothing else. And this makes this whole area possible to reason about with confidence.

3 Fixed point theorems

We now have to digress for a bit to discuss fixed point theorems. These are theorems that originally arose in other parts of mathematics, but then were seen to be important in theoretical computer science. So right now I want to give a very quick overview of the most famous fixed point theorem. I won't be proving anything—and in fact the proofs of these results are very sophisticated—but I *will* try to make the results understandable, even though they are quite surprising.

Suppose then we have a function $f : [0, 1] \to [0, 1]$. We could make a graph of this function, as in Figure 1. So long as the function f is continuous (and so in particular does not have any "jumps", or discontinuities), there will be some point $x \in [0, 1]$ such that $f(x) = x$. This particular point x is called a "fixed point" of the function f, because applying f doesn't move the point—the point remains fixed in its original position.

The fact that any such function f has a fixed point is actually not trivial to prove, but it should be intuitively plausible, particularly if you try to draw a few functions by hand. (And remember that we're talking about mathematical functions here—they don't have to have explicit formulas.) Some functions have more than one fixed point—for instance, *every* number $x \in [0, 1]$ is a fixed point of the function $f(x) = x$. But the important thing is that *every* continuous function $f : [0, 1] \to [0, 1]$ has at least one fixed point.

This fact, which everyone has to think about to some extent before they really believe it, is true in higher dimensions as well; and that fact is not obvious at all, I would say.

To understand what is going on, let us be as concrete as possible, and let us start out by restricting ourselves to two dimensions. For instance, suppose we denote by D the closed disk of unit radius in the 2-dimensional xy plane centered at the origin. (See Figure 2.) That is,

$$D = \{(x, y) : x^2 + y^2 \leq 1\}$$

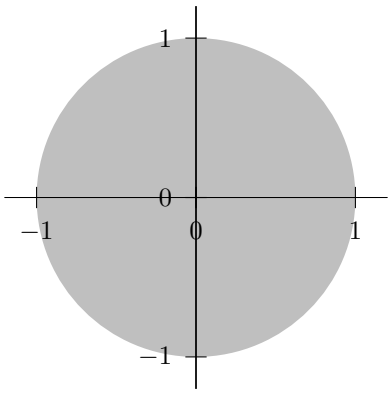

Figure 2: The unit disk D.

and suppose we have a continuous map $f : D \to D$. We can't graph the function f (it would take 4 dimensions to do that) but we can think of it simply as a transformation—that is, it takes each point inside D and moves it to some other point inside D, in such a way that nearby points wind up near each other. (That's pretty much what it means for f to be continuous.)

It's not necessary that boundary points of D wind up on the boundary, and in fact we have really a lot of leeway in how f acts, so long as it is continuous and takes points in D to other points in D.

Then it turns out that again f will have a fixed point.

Most people find this to be unbelievable, particularly after thinking about it a bit. Let me give a few simple examples.

First, suppose f is the map that shrinks D by a factor of 0.5. That is,

$$f(x, y) = (0.5x, 0.5y)$$

This is the map shown in Figure 3.

In Figure 3, the point a is mapped to the point $f(a)$. It's pretty obvious that $f(a)$ is closer to the origin than a is; in fact, that's how we got $f(a)$. So of course that point a is not a fixed point of f. On the other hand, this map f certainly does have a fixed point. It is the origin.

What about the map f that just rotates every point counter-clockwise about the origin by 30°? This map is certainly also a continuous map from D into D. And again the origin is a fixed point of this map.

Those are easy examples. But one could imagine making up harder ones. Suppose, for instance that we first shrunk the disk (as in Figure 3) and then moved it slightly to the right. The origin would no longer be a fixed point. *But there would be another one!* That's the theorem. You can see this example in Figure 4. Of course, this is just one example, not a proof that this always holds. But the theorem says that no matter how complicated we make the map—no matter how tricky we attempt to be—there will always be at least one fixed point. And this theorem can indeed be proved.

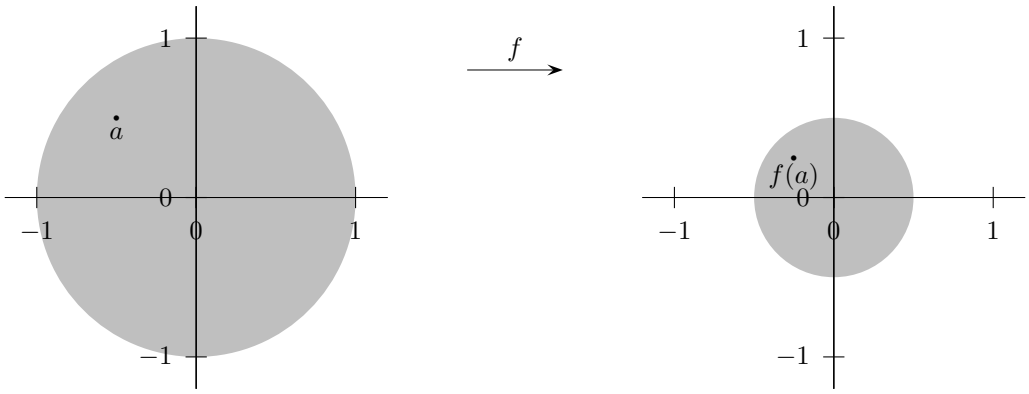

Figure 3: The map f that moves every point halfway toward the origin. The point a is mapped to the point $f(a)$.

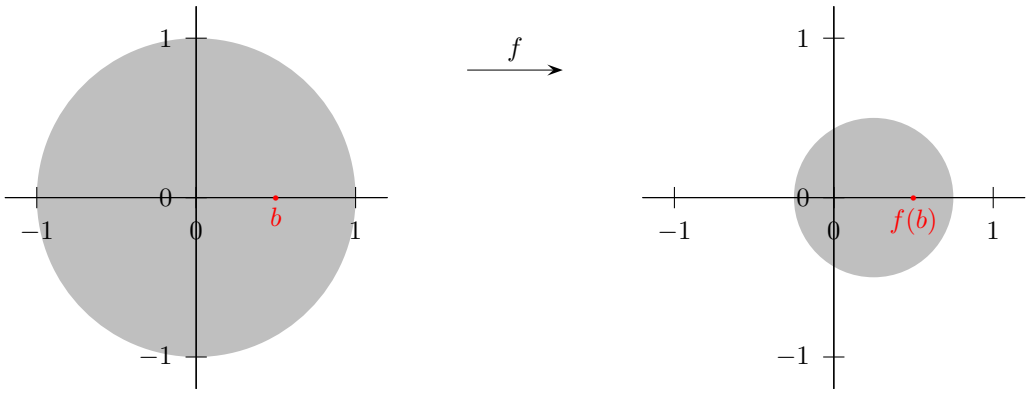

Figure 4: The map f that moves every point halfway toward the origin and then shifts every point a distance of 0.25 to the right. The point b is a fixed point of f, since $f(b) = b$.

You might think, for instance that if we took this last map and rotated everything somewhat we could get a map with no fixed point. But that idea just won't work. There really always will be a fixed point.

Here is another way you can think of this theorem: Suppose you start with two identical disks of paper, one on top of the other. Even better, let's say they are two identical flat rubber disks. You pick up the top disk and fold it and stretch it and compress it and crumple it up any way you want (the only thing you can't do is tear it), and then place it back down on the other disk. Then there will always be at least one point on the deformed disk that is exactly above where it started. Most people find that astonishing. I know I do.

And further, this continues to be true in higher dimensions. If you take a solid ball and map it continuously into itself, there will always be a fixed point. And it doesn't actually have to be a ball you start out with either, as long as it is something that could be continuously deformed into a ball. So if you started with a cup of coffee, and then you stir the coffee, there will always be one "point" in the coffee that winds up in its original position.

This theorem, which is true in any number of dimensions, is known as the *Brouwer fixed point theorem*. It is a deep and powerful result.

What is remarkable from our perspective here is that the notion of fixed points is also important in theoretical computer science. We'll see that next.

4 Church's lambda calculus: recursion

4.1 The problem of representing a recursive function

Recall that Church was trying to find a way to specify functions that were computable. He, as well as everyone else considering this problem, realized that a big issue in doing this was dealing with recursion.

Now at first—particularly if you are comfortable with computer programming, as we all are—this does not seem to be such a big deal. After all, we are all familiar with recursive functions, and we even know how to specify them. For instance, we have the function `fact` that generates factorials:

```
(define fact
  (lambda (n)
    (if (= n 1)
        1
        (* n
           (fact (- n 1))))))
```

This is all very nice, but there is a big problem here: In Church's lambda calculus, there was no special form `define`. And there was a reason for this, although Church probably wouldn't have expressed it exactly the way I'm going to here.

For our purposes, there are really two kinds of defines:

"safe" defines These are definitions that don't refer (either directly or indirectly) to themselves. So for instance, something like

```
(define a 3)
```

is a safe `define`. And it really is safe: if we see this definition in a program, we can actually do away with it completely, and just go and replace every mention of `a` in the program by 3. And if we had something like

```
(define x (- y 11))
```

we could replace `x` similarly. And—more to the point of what we're really talking about here, which is functions—this next definition

```
(define f
  (lambda (x)
    (* x x)))
```

is also safe, because whenever we see `(f x)`, we can replace it with `(* x x)` (and of course `x` can be any legal expression in our language).

So these defines are safe in the sense that they are just syntactic sugar and could be completely done away with by a trivial substitution[2].

"unsafe" defines These are definitions that refer (either directly or indirectly) to themselves. Certainly the definition of the factorial function above is unsafe in that sense.

Now, why do we say these are unsafe? Well, you can't just get rid of the name of the function you are defining by substituting. All that happens is that you wind up with an expression that contains another reference to the function which you likely will have to substitute for later, and so on. For instance, to evaluate 5!, we would have to do something like this: We start with

```
(fact 5)
```

Then, substituting in for `fact`, we get

```
((lambda (n)
   (if (= n 0)
       1
       (* n
          (fact (- n 1)))))
 5)
```

Substituting in again for `fact`, we get

[2]This is why the Abelson-Sussman text refers to the original model of Scheme interpretation in Chapters 1 and 2 as a *substitution model*.

There can of course be scoping issues that come up in doing this. But Church and his students understood scoping perfectly well (and this was even before modern programming languages!), and again there is no real problem in getting rid of safe defines even in the presence of scoping.

```
(
 (lambda (n)
   (if (= n 0)
       1
       (* n
          (
           (lambda (n-minus-1)
             (if (= n-minus-1 0)
                 1
                 (* n-minus-1
                    (fact (- n-minus-1 1)))))
           (- n 1)
          ) ))
 )
 5
)
```

Here we have named the inner formal function argument `n-minus-1`, because we can see that it is being given the value `(- n 1)`. Of course it could have any name. We just name it this way to make the code easier to read.

Substituting in yet again for `fact`, we get

```
(
 (lambda (n)
   (if (= n 0)
       1
       (* n
          (
           (lambda (n-minus-1)
             (if (= n-minus-1 0)
                 1
                 (* n-minus-1
                    (
                     (lambda (n-minus-2)
                       (if (= n-minus-2 0)
                           1
                           (* n-minus-2
                              (fact (- n-minus-2 1)))))
                     (- n-minus-1 1)
                    ) ))
           )
           (- n 1)
          ) ))
 )
 5
)
```

and you can see that eventually, we'll end up stopping with the correct value for 5!, but along the way, we have had to remember the definition of `fact` so that we could substitute it in our expression. And we had to remember this quite a few times.

The way that Scheme really deals with the "remembering" that is needed to support these unsafe defines is by means of an environment: An entry for `fact` is put into the environment. It is bound to a procedure object whose body contains a reference to the symbol `fact`. When the function is executed, the body is evaluated. In the course of that evaluation, the symbol `fact` will be seen, and it will be looked up in the environment. The procedure object that it is bound to will be retrieved, and will be executed, and this will happen recursively as many times as is needed.

And so unsafe defines only work in Scheme because we have an environment. And an environment is a separate data structure. This is not at all what Church had in mind. He didn't have any built-in data structures other than lambda expressions. (In most programming languages, including Scheme, it's even worse than that—the environment is typically modified in the course of running a program—and Church certainly never had mutable data. That would have made things much too complex from the point of reasoning about program behavior[3].)

So what's a poor programmer to do? Well, Church and his students figured out a really clever way to implement recursive functions without using unsafe defines. It involves fixed points. And that's what we'll look at next.

4.2 The Y operator

There are some strange things that can happen with lambda expressions. Here is one of the strangest:

Suppose we try to evaluate this expression:

```
((lambda (x) (x x))
 (lambda (x) (x x)))
```

What probably immediately strikes you as strange about this is that we are applying x (which presumably is a function) to itself. Well, let's pretend this doesn't make us uncomfortable and just go on. The expression as a whole is also a procedure application. The procedure is `(lambda (x) (x x))` and it also is being applied to itself. So if we take the body (i.e., `(x x)`) of the function and substitute in the argument (i.e., `(lambda (x) (x x))`), we wind up with

```
((lambda (x) (x x))
 (lambda (x) (x x)))
```

That is, this expression evaluates to itself. It's very curious.

It turns out that this curious expression can be turned to good use. Suppose we modify that expression slightly as follows:

[3] In fact, reasoning about mutable data is arguably the most difficult part of analyzing programs—the kind of thing that compilers are supposed to be able to do.

```
((lambda (x) (f (x x)))
 (lambda (x) (f (x x))))
```

When we evaluate this, we get

```
(f ((lambda (x) (f (x x)))
    (lambda (x) (f (x x)))))
```

Thus, if we define `Y` as follows (and please note that this is a *safe* define):

```
(define Y
  (lambda (f)
    ((lambda (x) (f (x x)))
     (lambda (x) (f (x x))))))
```

then for any function `f`, we see that `(Y f)` evaluates to `(f (Y f))`. Or more simply,

$$\texttt{(f (Y f))} \equiv \texttt{(Y f)} \tag{1}$$

This is a definition of what is usually called the `Y` operator[4].

Well that's nice, but here is the really powerful way to think of this: Suppose we have a function `f`. And suppose we are looking for a fixed point of `f`. This just means an argument `x` such that `f(x) = x`.

Now before, when we were talking somewhat geometrically, `x` was a number on the line (in the interval $[0, 1]$), or a geometrical point in the unit disk D, or a point in a ball in 3-dimensional space, or something like that. But now we are in the realm of lambda calculus, in which there is nothing but functions. And so `x` will of necessity be a function.

And what we have just shown is this: if we start with a function `f` and form the expression `(Y f)`, then this expression `(Y f)` is a fixed point of the original function `f`. In other words, `Y` is a remarkable function: it takes as input any function at all (of one argument) and produces a fixed point for that function. That's just magic.

4.3 How to write a recursive function

We can now use the `Y` operator to create recursive functions. Here's how.

Suppose we want to compute factorials. We can't use the ordinary recursive definition, because it is unsafe. But we can do this: We write a function—let's call it `F`—which would have the factorial function we are looking for as a fixed point. Here is the definition of `F`:

[4]The term "operator" in this context is just a fancy synonym for "function".

```
(define F
  (lambda (f)
    (lambda (n)
      (if (= n 0)
          1
          (* n (f (- n 1)))))))
```

Suppose there is a function `f` which is a fixed point of `F`. (As you can see, the argument to `F` is a function which itself takes one argument, `n`, so `f` must be a function of one argument.) Then we must have

```
(F f) = f
```

and this just means that `f = (F f)`, which in turn equals

```
(lambda (n)
   (if (= n 0)
       1
       (* n f (- n 1))))
```

That is, the fixed point `f` of `F` is precisely the function we are looking for. And we didn't need to write it using an unsafe define, or equivalently, an explicitly recursive definition[5]. In fact, it is just `(Y F)`.

4.4 Is this too good to be true?

Something may be bothering you at this point: The `Y` operator takes as input a function F and produces a fixed point f for that function. But it's easy to see that there are functions that have no fixed points. For example, suppose we consider the ordinary function F defined on the integers (or say, on the Church numerals) as

$$F(n) = n + 1$$

or a bit more formally,

$$F(n) = S(n)$$

or still more formally,

```
(define F
  (lambda (n)
    (S n)))
```

[5]There is one problem to beware of here: what we have written implicitly uses normal-order evaluation. Scheme uses applicative-order evaluation, and as a result if you tried to write this in Scheme, you would actually get an infinite recursion. However, there is a modified—and unfortunately, somewhat more complex—version of the `Y` operator that works with applicative-order evaluation. I'm not presenting it here, because I don't really think it's all that interesting or important for the points I'm making in this short writeup.

where S is the successor function we talked about earlier. It's certainly clear that F has no fixed point. And yet the `Y` operator when applied to F does yield a fixed point for F. What could this possibly mean?

Here's what it means: We have been playing a little fast and loose with our terminology. We have been referring to *functions* when we probably more accurately should have been talking about *lambda expressions*. These expressions take the form of functions, but it's easy to see that the functions they code for might not be everywhere defined. For instance—given the fact that we can, as mentioned above, encode conditional "if" constructs in the lambda calculus—we can write the following (using the `Y` operator to process recursive constructions):

```
(define F
  (lambda (n)
    (if (= n 0)
        1
        (F n))))
```

This amounts to a function defined only for $n = 0$. And it would be just as easy to write down a lambda expression that coded for a function defined nowhere. In theoretical computer science, we refer to functions that may or may not be defined everywhere—and might even be defined nowhere—as *partial functions*. And in fact the `Y` operator applied to the successor function above yields an expression that corresponds to a function defined nowhere. In a trivial sense, this really *is* a fixed point of the successor function.

OK, so now you may be thinking that perhaps the whole thing is a fraud. How do we know that the `Y` operator ever returns anything meaningful? Maybe it always produces an expression amounting to a function that is nowhere defined. That would be pretty annoying.

Fortunately, that's not the case. In fact, looking at the definition of F at the top of section 4.3, it is obvious that if f is *any* fixed point of F, then it must be the case that $f(0) = 1$. And subsequently, we can see that $f(1)$ must also be 1. And then $f(2)$ must be 2, $f(3)$ must be 6, and so on. So there are some values of the function f that are simply forced on us. In this particular case, these are the values of the factorial function, and we see that we really do get the factorial as a fixed point.

It's possible to write a recursive definition of a function[6] which only forces the values to be defined at certain points but not at others. It might be, for instance that the function would be well-defined at the even numbers, but the values at the odd numbers might be somewhat arbitrary—they might not be well-defined. In such a case, we would say that there was a *minimal* fixed-point solution of the recursion, defined only at the even numbers.

And this turns out (and can be proved) to be true in general: The `Y` operator produces a real fixed point which is in fact the minimal fixed point; that is, it is well-defined exactly where it has to be. And this allows us to write recursive formulas and get meaningful results from them.

4.5 One final word

Now all this is undeniably cute, but you still may be thinking that perhaps we are using a sledgehammer to kill a gnat. So let me just end by remarking that this `Y` operator construction is pretty

[6] Of course, it would be a different function.

much the same as a very powerful and well-known construction in the theory of computation known as the Recursion Theorem[7]. So while from one point of view this might seem to be just a very clever hack, virtually the same technique also appears as a basic tool used in proving deep results in the theory of computation. And those results are generally expressed not in terms of the lambda calculus—although they could be—but in terms of objects such as Turing Machines or other abstract models.

A Appendix: An example of how the `Y` operator works

No one in their right mind actually expands out expressions involving the `Y` operator. But you might be suspicious that these expressions do involve multiple and continued references to `Y` or to the function `F` for which we are trying to find a fixed point, and so we would need to have a way of "remembering" these definitions. And in fact, a number of worked-out examples of this (e.g., on Wikipedia, at least when I looked at it) do make it seem like that actually has to happen.

So let me just take our running example and show how it really works, without any multiple uses of `Y` or `F`.

As before we'll start with the definitions of `Y` and `F`:

```
(define Y
  (lambda (f)
    ((lambda (x) (f (x x)))
     (lambda (x) (f (x x))))))

(define F
 (lambda (f)
   (lambda (n)
     (if (= n 0)
         1
         (* n (f (- n 1)))))))
```

As we've already remarked, these are "safe" defines.

Now we want to compute the fixed point of `F`. We know that it will be the expression

```
(Y F)
```

So first we substitute for `Y` in this expression. Since `Y` is a "safe" define, this will only happen once. We get this:

```
((lambda (x) (F (x x)))
 (lambda (x) (F (x x))))
```

[7] I'm being a bit sloppy here. There are actually several such theorems. But they all accomplish similar ends. And they all are due to Kleene. And quite likely he was motivated in large part by constructions in the lambda calculus such as the `Y` operator.

Next, we substitute for F. And since F is a "safe" define, this will also only happen once. We get this:

```
((lambda(x) (lambda (n)
              (if (= n 0)
                  1
                  (* n
                     ((x x) (- n 1))))))
 (lambda(x) (lambda (n)
              (if (= n 0)
                  1
                  (* n
                     ((x x) (- n 1))))))  )
```

So now we have an expression that no longer involves Y or F. This expression has the form of a function application in which a big lambda expression is being applied to itself. We perform that application, and get this:

```
(lambda (n)
  (if (= n 0)
      1
      (* n
         (
          ((lambda(x) (lambda (n-minus-1)
                        (if (= n-minus-1 0)
                            1
                            (* n-minus-1
                               ((x x) (- n-minus-1 1))))))
           (lambda(x) (lambda (n-minus-1)
                        (if (= n-minus-1 0)
                            1
                            (* n-minus-1
                               ((x x) (- n-minus-1 1)))))))
          (- n 1)
         )
      )))
```

As we did earlier, here we have named the inner formal function argument `n-minus-1`, because we can see that it is being given the value `(- n 1)`. Just as before, it could have any name. We just name it this way to make the code easier to read.

What is different now though from what we did before is this: We do not have to "remember" or "look up" any variables or functions—in particular, we don't have to worry about the function `fact`, which doesn't even appear here. So we have no need for an environment.

Now if this function is applied to $n = 0$, we're immediately done: the answer is 1. Otherwise, we go on and evaluate the red expression in the middle of this:

```
(lambda (n)
  (if (= n 0)
      1
      (* n
         (

          (lambda (n-minus-1)
            (if (= n-minus-1 0)
                1
                (* n-minus-1
                   (
                    ((lambda(x) (lambda (n-minus-2)
                                  (if (= n-minus-2 0)
                                      1
                                      (* n-minus-2
                                         ((x x) (- n-minus-2 1))))))
                     (lambda(x) (lambda (n-minus-2)
                                  (if (= n-minus-2 0)
                                      1
                                      (* n-minus-2
                                         ((x x) (- n-minus-2 1)))))))
                    (- n-minus-1 1)
                   )
                )))
          (- n 1)

         )
      )))
```

If this is being applied to $n = 1$, we would be done: the answer would be $1 \cdot 1 = 1$. If $n > 1$, we can go on. At the next step we would see that if $n = 2$, the answer would be $2 \cdot 1 \cdot 1 = 2$. If $n = 3$, we would have to go one step farther, and the answer would be $3 \cdot 2 \cdot 1 \cdot 1 = 6$, and so on.

Clearly we can continue in this way as far as needed. Because of the way that the **if** special form works we stop as soon as we can. And it should now be clear that this function—which is the fixed point of `F`—when applied to any non-negative integer k, yields $k!$.

One final note about what we have done here: in pure lambda calculus, there is no **if** special form. In fact, there are no "special forms" at all. Nor are there primitive procedures such as subtraction and multiplication, for that matter. But as we have already seen, the lambda calculus—simple and bare-bones as it is—is rich enough to support all of this.