I realized why I was so confused in class. It turns out you can't do what I was trying to do, at least not in the way I was trying to do it. Recall the problem: we have a verb 'admires' with ST $N \rightarrow (N \rightarrow S)$. So its semantic type should be: $E \rightarrow (E \rightarrow T)$. We wanted to know which particular function 'admires' means.

We know that it's a function of one variable: its domain is the set of objects of type E and its codomain (range) is the set of objects of type $E \rightarrow T$. Since it's a function of one variable, we can represent it as:

$[[admires]] = f(x)$

Now we just need to say what f(x) is. We know that the values it returns are functions (they're type $E \rightarrow T$) but *which* function it returns depends on the input, x. So:

$f(Bill) = g_0(y)$
$f(Tina) = g_1(y)$
$f(Fred) = g_2(y)$
$f(Sally) = g_3(y)$

And so on. And now we can say what each of these functions are:

$g_0(y)$ = true, if y admires Bill; false otherwise
$g_1(y)$ = true, if y admires Tina; false otherwise
$g_2(y)$ = true, if y admires Fred; false otherwise
$g_3(y)$ = true if y admires Sally; false otherwise

And so on. Finally we're in a position to say what f(x) is for x in general:

$f(x) = g_0(y)$ if x = Bill; $g_1(y)$ if x = Tina; $g_2(y)$ if x = Fred; $g_3(y)$ if x = Sally;…

This means that f(x) will have a different clause for each object of type E. That's a lot of clauses!

What we'd need to get around this limitation is an expression that isn't a name for a function, like f, g_0, or g_1, but rather denotes different functions relative to different assignments to its free variables. Consider this expression:

The smallest function such that given an argument y, it outputs x + y.

This doesn't denote any function at all, because what is x? We haven't said. If I say x = 5, the expression denotes h0(y) = 5 + y; if I say x = 17, it denotes h1(y) = 17 + y; and so on. The expression denotes a function only relative to a determination of x.

To bring this closer to home now, consider:

> The smallest function such that given an argument y, it outputs true if y admires x and false otherwise.

This expression, unlike g_0 or g_1 or g_2, doesn't denote any function at all. It only denotes a function relative to a determination of x. Now, consider this:

> The smallest function such that given an argument x, it outputs the smallest function such that given an argument y, it outputs true if y admires x and false otherwise.

This does name a function, in particular it denotes [[admires]] = f(x). To see this, imagine giving the function some argument, say, Tina. Then it would output the smallest function such that given an argument y, it outputs true if y admires Tina and false otherwise. But this is just g_1, which is f(Tina). In fact, I think you'll find that whatever argument you give it, it outputs just what f(x) would output.

This, incidentally, is the lambda calculus that I mentioned in class. Here's a lambda expression:

> λx. x + 5

The way you're supposed to read this is "the smallest function such that given an input x, it returns as output x + 5" i.e. f(x) = x + 5. But not every lambda expression denotes a function. The expression:

> λy. x + y

Only denotes a function relative to a determination of x. It means "The smallest function such that given an argument y, it outputs x + y." To bring this closer to home, we can consider:

> λy. true, if x admires y; false otherwise.

This denotes different functions given different values for x. For example, given x = Bill, it denotes g0, and given x = Tina, it denotes g1. Now we're in a position to characterize f(x), namely:

f(x) = λx. λy. true, if x admires y; false otherwise.

This is to be read, just as before: “The smallest function such that given an argument x, it outputs the smallest function such that given an argument y, it outputs true if y admires x and false otherwise.

So our biggest problem (or *my* biggest problem) was that we wanted there to be a single function g(y) that was the output of f(x) given a value for x. But there is no single such function; we have different functions as outputs of f(x) for each distinct value of x. It takes a long time to write out the definition of f(x) if we’re limited to static representations of functions (that is, a language of functions that doesn’t have names that variably refer to distinct functions), but once we introduce the lambda calculus, we can easily write down all the functions our semantic typology requires.