(1) The Church numerals are a way of encoding natural numbers as λ-terms, thus:

- 0 is \( \lambda f. \lambda x. x \)
- 1 is \( \lambda f. \lambda x. f(x) \)
- 2 is \( \lambda f. \lambda x. f(f(x)) \)
- and so on

Can you write a λ-term which encodes the successor function: that is, given a term representing \( n \), it returns a term representing \( n + 1 \)? (The answer is of course available everywhere, so don’t look up ‘Church numeral’ until you’ve tried!)

(2) One way to code up list structures in \( \lambda \)-calculus is this. The list \( (x, y, z) \) is represented as a function that takes two arguments \( c \) and \( n \), and gives back \( cx(cy(czn)) \); in other words, \( (x, y, z) \overset{\text{def}}{=} \lambda c. \lambda n. cx(cy(czn)) \). Similarly for lists of other lengths.

Explain this construction. (\( \text{Hint:} \) the choice of ‘\( c \)’ and ‘\( n \)’ as letters is not random.)

Give definitions in this encoding of λ-terms representing the \textit{nil} list, the \textit{cons} function, and the \textit{head} (or \textit{car} for LISPers) function.

What happens if you call your \textit{head} function on the \textit{nil} list?

(3) The recursion combinator we used was

\[
Y \overset{\text{def}}{=} \lambda F. (\lambda X. F(XX))(\lambda X. F(XX))
\]

What happens if you try to use \( Y \) in a call-by-value evaluation strategy?

Here is the version of \( Y \) that works for call-by-value:

\[
Y' \overset{\text{def}}{=} \lambda F. (\lambda X. F(\lambda Z.XXZ))(\lambda X. F(\lambda Z.XXZ))
\]

This is very similar – study it, and describe what technique, mentioned in the slides, is being used to make \( Y' \) from \( Y \). (\( \text{Hint:} \) a Greek letter is involved.)

(4) If \( t \) is a well-typed \( \lambda \)-term \( t : \tau \), then it evaluates into a well-typed term \( t' : \tau \). Is it true that for terms \( s \) and \( s' \), if \( s' : \tau \) and \( s \overset{\beta}{\rightarrow} s' \), then \( s : \tau \)?