We’ve discussed how to represent Boolean values in the untyped λ-calculus, and integers, and lists of integers, and binary trees, etc. The approach was: look at the constructors for the data type, and λ-abstract over them. The same approach has to work for representing the λ-calculus inside the calculus itself.

Here’s what I mean. For a λ-term \( t \), let’s write \( \lceil t \rceil \) for its representation. Let’s give the name \( \text{LC} \) the data type of untyped λ-terms. The informal declaration of this data type is:

\[
\text{type LC=} \\
\text{Var: Int -> LC} \\
\text{App: LC -> LC -> LC} \\
\text{Abs: LC -> LC}
\]

We represent variable occurrences by lexical addresses that say how many λs to “skip over” to get to the right λ-binder—for example, we’d code \( \lambda x. x \) as \( \text{abs (var \ 0)} \), \( \lambda x. \lambda y. xy \) as \( \text{abs (abs (app (var \ 1) (var \ 0)))} \), coding binding addresses as numerals of type \( \text{Int} \), and \( \lambda x. x (\lambda y. xy (\lambda z. xyz (\lambda w. xyzw))) \) as

\[
\text{abs (app (var \ 0))} \\
\text{(abs (app (app (var \ 1) (var \ 0))) (var \ 0))} \\
\text{(abs (app (app (var \ 3) (var \ 2)) (var \ 1)) (var \ 0))})
\]

To code this representation in the λ-calculus itself, we λ-abstract over the constructors \( \text{app, abs, and var} \), so that (for example) the above small example \( \lambda x. x \) is coded as \( \lambda \text{app.} \lambda \text{abs.} \lambda \text{var.} \text{abs (var \ 0)} \), and \( \lambda x. \lambda y. xy \) as \( \lambda \text{app.} \lambda \text{abs.} \lambda \text{var.} \text{abs (abs (app (var \ 1) (var \ 0)))} \). Numerals are now Church numerals, also abstracting over its constructors.

Dispensing with our informal data type \( \text{LC} \), write \( \lceil t \rceil \) for the representation of \( t \) in the untyped λ-calculus. For example, for the term \( t \equiv \lambda w.w \), we have (expanding the Church numerals \( \overline{k} \) appropriately):

\[
\begin{align*}
\lceil t \rceil & \equiv \lambda \text{app.} \lambda \text{abs.} \lambda \text{var.} \text{abs (var (\lambda s.\lambda z.s))} \\
\lceil \lceil t \rceil \rceil & \equiv \lambda \text{app.} \lambda \text{abs.} \lambda \text{var.} \\
& \quad \text{abs (abs (app (var (\lambda s.\lambda z.s))) (app (var (\lambda s.\lambda z.z))(abs (abs (var (\lambda s.\lambda z.z)))))))} \\
\lceil t \lceil t \rceil \rceil & \equiv \lambda \text{app.} \lambda \text{abs.} \lambda \text{var.} \\
& \quad \text{app (abs (var (\lambda s.\lambda z.z)))} \\
& \quad \quad \text{abs (abs (app (var (\lambda s.\lambda z.s))) (app (var (\lambda s.\lambda z.z))(abs (abs (var (\lambda s.\lambda z.z)))))})}
\end{align*}
\]
**Exercise 0.** Define an appending or juxtaposing operator $\star$ for coding applications, so that $\star [t] [\![t]\!] = [t \! [t\!]$—or, more generally, $\star [t] [u] = [t u]$.

**Exercise 1.** Define a quotation operator that works only on representations of $\lambda$-terms, so that $\text{quoteit} [t] = [\![t]\!]$. (This is a little tricky, but worth the thinking.) A hint: confusing the $\lambda$-abstracted variables in $[t]$ and $[\![t]\!]$ is easy, so clear your head by renaming them, for example

\[
[t] \equiv \lambda app'. \lambda abs'. \lambda var'. \lambda s'. \lambda z'. (\lambda s. \lambda z. z')
\]

\[
[\![t]\!] \equiv \lambda app. \lambda abs. \lambda var. \lambda s. \lambda z. (\lambda s. \lambda z. z) (\lambda s. \lambda z. z) (\lambda s. \lambda z. z) (\lambda s. \lambda z. z) (\lambda s. \lambda z. z)
\]

Or pick names that are even more disjoint: $\lambda Fred. \lambda Wilma. \lambda Barney. \ldots$ A further hint: try and see precisely what it is in $[\![t]\!]$ that is simulating what structure in $[t]$.

For the above exercises, you might find it useful to do some programming for experimental purposes in your favorite functional programming language, as a testbed for your ideas. Neither of the solutions to Exercises 0 and 1 should use recursion, or any feature from the programming language other than function application, as in many codings shown in class.

**Exercise 2.** System F lets us code Booleans as terms of type $\textbf{Bool} \equiv \forall P. P \to P \to P$, Church numerals of type $\textbf{Int} \equiv \forall P. (P \to P) \to P \to P$, and lists of Church numerals as $\forall P. (\textbf{Int} \to P \to P) \to P \to P$. What is the System F type for the representation of $\lambda$-terms described above?

**Exercise 3.** Take the untyped $\lambda$-terms that are your solutions to Exercises 0 and 1, and type them as System F terms.

**Exercise 4.** Show that $\lambda w. ww$ can be given the type $\sigma \equiv (\forall P. P \to P) \to (\forall P. P \to P)$. Explain why the same is true of $\lambda w. www$ and $\lambda w. wwwww(wwwww)ww(wwww)www(wwwww)www(wwwww)$.