1 One-step $\beta$-Reduction (80%)

Because (as you saw in class) there may be many ways to $\beta$-reduce a single term, people have come up with various “strategies” which, given any term, specify what the next reduction step should be. Here are four (the names are historical).

- **Normal-Order Reduction**: reduce the application (of a function to an argument) whose $\lambda$ appears leftmost in (i.e., closest to the beginning of) the term.

  $$\left(\lambda x. (\lambda z. ((\lambda w. w)(\lambda v. v)))\right) (\lambda u. u)(\lambda y. y) \rightarrow_\beta \lambda z. ((\lambda w. w)(\lambda v. v)) \rightarrow_\beta \lambda z. (\lambda v. v)$$

  Normal-order reduction is interesting because it has the following guarantee: if there is any possible sequence of reductions to a term which has no occurrences of a function applied to an argument, then normal-order reduction will eventually produce this term.

- **Applicative-Order Reduction**: reduce the application whose $\lambda$ occurs leftmost if the argument of this application cannot be further reduced; otherwise, first reduce the argument by one step (using applicative-order reduction).

  $$\left(\lambda x. (\lambda z. ((\lambda w. w)(\lambda v. v)))\right) (\lambda u. u)(\lambda y. y) \rightarrow_\beta \left(\lambda x. (\lambda z. ((\lambda w. w)(\lambda v. v)))\right) (\lambda y. y) \rightarrow_\beta \lambda z. ((\lambda w. w)(\lambda v. v)) \rightarrow_\beta \lambda z. (\lambda v. v)$$

  Applicative reduction is does not have the same termination guarantee as normal-order reduction, but in many cases it is more efficient.

- **Call-by-Name Evaluation**: The same as normal-order reduction, except we never reduce applications inside the body of a function. (This strategy may stop before all reductions are gone.)

  $$\left(\lambda x. (\lambda z. ((\lambda w. w)(\lambda v. v)))\right) (\lambda u. u)(\lambda y. y) \rightarrow_\beta \lambda z. ((\lambda w. w)(\lambda v. v))$$
This is a less aggressive version of normal-order reduction. If the result is a function that is not being applied to anything, why bother working on the function?

- **Call-by-Value Evaluation**: The same as applicative-order reduction, except that we never reduce applications inside the body of a function. (This strategy may stop before all reductions are gone.)

\[
\left( \lambda x. (\lambda z. ((\lambda w. w)(\lambda v. v))) \right) \left( \lambda u. (\lambda y. y) \right) \\
\rightarrow \beta \left( \lambda z. ((\lambda w. w)(\lambda v. v)) \right) \left( \lambda y. y \right) \\
\rightarrow \beta \lambda z. ((\lambda w. w)(\lambda v. v))
\]

This is the corresponding less-aggressive version of applicative reduction.

Write four functions corresponding to the above four strategies:

- \text{normal} : \text{lam} \rightarrow \text{lam option}
- \text{applicative} : \text{lam} \rightarrow \text{lam option}
- \text{cbn} : \text{lam} \rightarrow \text{lam option}
- \text{cbv} : \text{lam} \rightarrow \text{lam option}

These functions should return \text{NONE} if the argument cannot be reduced according to the given strategy, and \text{SOME} \ e' if the argument can be reduced one step to get the new term \ e'.

You may find the following pattern to be useful for many cases within your functions:

\[
\begin{align*}
\text{(case } \ldots \text{ recursive call } \ldots \text{ of} \\\n\text{NONE }&\Rightarrow \ldots \text{ do something} \ldots \\
\text{ | SOME } e' &\Rightarrow \ldots \text{ do something else} \ldots \\
\text{)}
\end{align*}
\]

It is always wise to put parentheses around \text{case} statements.

## 2 Multistep $\beta$-Reduction (20%)

Write a function

\[
\text{reduce} : (\text{lam} \rightarrow \text{lam option}) \rightarrow \text{lam} \rightarrow \text{lam}
\]

which takes a reduction strategy (i.e., one of the four functions above) and a lambda expression as sequential arguments. The \text{reduce} function should then repeatedly do one-step reduction using the given strategy until no further reductions are possible; if this occurs, return the final \lambda-expression. Note that there are inputs such as

\[
(\lambda x. (x x))(\lambda y. (y y))
\]

for which \text{reduce} will not terminate for any reduction strategy, and there are other inputs such as

\[
(\lambda z. w)((\lambda x. (x x))(\lambda y. (y y)))
\]

or

\[
\lambda z. ((\lambda x. (x x))(\lambda y. (y y)))
\]

which terminate under some strategies but not others.