This question involves the derivation of “stack machines” using the CPS transformation.

(a) Consider the following OCaml code of type

\[
\text{add_right} : \text{int list} \rightarrow \text{int}
\]

that returns the sum of the integers in its argument list.

\[
\text{let rec add_right } l =
\begin{array}{l}
\text{match } l \text{ with } \\
\text{ | } [] \rightarrow 0 \\
\text{ | } h::tl \rightarrow h + (\text{add_right} \ tl)\\
\end{array}
\]

Explain why this code, as presented, is not tail recursive. \[2 \text{ marks}\]

(b) Use the CPS transformation to rewrite \text{add_right} to a function that could be given the type

\[
\text{add_right_cps} : \text{int list} \rightarrow (\text{int} \rightarrow \text{int}) \rightarrow \text{int}
\]

\[6 \text{ marks}\]

(c) Apply defunctionalisation to your code for \text{add_right_cps}. That is, define a (non-functional) data type \text{cnt} and a transformed function \text{add_right_dfc} of type

\[
\text{add_right_dfc} : \text{int list} \rightarrow \text{cnt} \rightarrow \text{int}
\]

\[6 \text{ marks}\]

(d) The function \text{add_right} from Part (a) could be generalised to the following function.

\[
\text{let rec fold_right } f \ l \ \text{accu} =
\begin{array}{l}
\text{match } l \text{ with } \\
\text{ | } [] \rightarrow \text{accu} \\
\text{ | } a::l \rightarrow f\ a\ (\text{fold_right} \ f \ l \ \text{accu})\\
\end{array}
\]

For simplicity, we will treat this code as if it had the type

\[
\text{fold_right} : (\text{int} \rightarrow \text{int} \rightarrow \text{int}) \rightarrow \text{int list} \rightarrow \text{int} \rightarrow \text{int}
\]
and not worry about polymorphism. Rewrite this program using the CPS transformation. Justify your treatment of the variable \( f \). What problems might you encounter in attempting to defunctionalise your CPS version? [6 marks]