2 Compiler Construction (tgg22)

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

(*a*) Consider the following OCaml code of type

```
add_right : int list -> int
```

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

```
let rec add_right l =
  match l with
  | [] -> 0
  | h::tl -> h + (add_right tl);;
```

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

(*b*) Use the CPS transformation to rewrite `add_right` to a function that could be given the type

```
add_right_cps : int list -> (int -> int) -> int
```

[6 marks]

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

```
add_right_dfc : int list -> cnt -> int
```

[6 marks]

(*d*) The function `add_right` from Part (*a*) could be generalised to the following function.

```
let rec fold_right f l accu =
  match l with
  | [] -> accu
  | a::l -> f a (fold_right f l accu);;
```

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

```
fold_right : (int -> int -> int) -> int list -> int -> 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]