3 Compiler Construction (tgg22)

(*a*) Suppose we are writing a compiler for an ML-like language and we want to employ the equation

```
(map f l1) @ (map f l2) = map f (l1 @ l2)
```

as a left-to-right rewrite rule for optimisation. The symbol `@` represents list append.

Discuss the merits of this idea. Is it always correct? If so, state clearly what assumptions you are making about `@` and `map`. [5 marks]

(*b*) A compiler's front-end will often expand some syntactic constructs into lower-level representations. Consider the following fragment for the abstract syntax of a SLANG-like language.

```
type var = string

type exp =
  (* abstract syntax *)      (* concrete syntax *)
  | Var of var               (* x *)
  | Project of int * exp     (* proj i e *)
  | Tuple of exp list        (* (e1, e2, ..., en) *)
  | Let of var * exp * exp   (* let x = e1 in e2 *)
  | Apply of exp * exp       (* e1 e2 *)
  | Function of var * arg_pattern * exp   (* fun f p = e *)

and arg_pattern =
  | APvar of var                    (* x *)
  | APtuple of arg_pattern list     (* (p1, p2, ... pn) *)
```

This language has general projections for n-tuples so

$$\text{proj } i \ (e_1,\ e_2, \cdots, e_k)$$

will evaluate to v_i, the value of e_i. If i is not in the range between 1 and k there will be a run-time error.

In this language we can write functions with simple (possibly nested) patterns for function arguments:

```
fun f (a, b, (c, (d, e)) = b a
```

[continued ...]

Now suppose we want our front-end to eliminate such patterns. That is, we want to write a function of type

```
eliminate_tuple_patterns : exp -> exp
```

so that the resulting expression contains functions with patterns only of the form `APvar x` for some (new) variable `x`. For example, the code for `f` above should be translated to a semantically equivalent expression of the form

```
fun f x =  ...
```

that contains only simple variable arguments (that is, only `APvar` patterns in the abstract syntax).

Your task is to write this function in OCaml. You can assume that you have a function for generating fresh variable strings.

```
new_var : unit -> string
```

[15 marks]