The Expression Problem

Exercise 1. Implement the `string_of_` functions or methods, covering all data cases, corresponding to the `eval_` functions in at least two examples from the lecture, including both an object-based example and a variant-based example (either standard, or polymorphic, or extensible variants).

Exercise 2. Split at least one of the examples from the previous exercise into multiple files and demonstrate separate compilation.

Exercise 3. Can we drop the tags `Lambda_t`, `Expr_t` and `LExpr_t` used in the examples based on standard variants (file `FP_ADT.ml`)? When using polymorphic variants, such tags are not needed.

Exercise 4. Factor-out the sub-language consisting only of variables, thus eliminating the duplication of tags `VarL`, `VarE` in the examples based on standard variants (file `FP_ADT.ml`).

Exercise 5. Come up with a scenario where the extensible variant types-based solution leads to a non-obvious or hard to locate bug.

Exercise 6. * Re-implement the direct object-based solution to the expression problem (file `Objects.ml`) to make it more satisfying. For example, eliminate the need for some of the `rename`, `apply`, `compute` methods.

Exercise 7. Re-implement the visitor pattern-based solution to the expression problem (file `Visitor.ml`) in a functional way, i.e. replace the mutable fields `subst` and `beta_redex` in the `eval_lambda` class with a different solution to the problem of treating `abs` and non-`abs` expressions differently.

* See if you can replace the reference cells `result` in `eval`*N* and `freevars`*N* functions (for *N* = `1,2,3`) with a different solution to the problem of polymorphism wrt. the type of the computed values.

Exercise 8. Extend the sub-language `expr_visit` with variables, and add to arguments of the evaluation constructor `eval_expr` the substitution. Handle the problem of potentially duplicate fields `subst`. (One approach might be to use ideas from exercise 6.)

Exercise 9. Impement the following modifications to the example from the file `PolyV.ml`:

1. Factor-out the sub-language of variables, around the already present `var` type.
2. Open the types of functions `eval3`, `freevars3` and other functions as required, so that explicit subtyping, e.g. in `eval3 [] (test2 :> lexpr_t)`, is not necessary.
3. Remove the double-dispatch currently in `eval_lexpr` and `freevars_lexpr`, by implementing a cascading design rather than a "divide-and-conquer" design.

Exercise 10. Streamline the solution `PolyRecM.ml` by extending the language of λ-expressions with arithmetic expressions, rather than defining the sub-languages separately and then merging them. See slide on page 15 of Jacques Garrigue *Structural Types, Recursive Modules, and the Expression Problem*.

Exercise 11. Transform a parser monad, or rewrite the parser monad transformer, by adding state for the line and column numbers.

* How to implement a monad transformer transformer in OCaml?

Exercise 12. Implement `_of_string` functions as parser combinators on top of the example `PolyRecM.ml`. Sections 4.3 and 6.2 of *Monadic Parser Combinators* by Graham Hutton and Erik Meijer might be helpful. Split the result into multiple files as in Exercise 2 and demonstrate dynamic loading of code.

Exercise 13. What are the benefits and drawbacks of our lazy-monad-plus (built on top of *odd lazy lists*) approach, as compared to regular monad-plus built on top of *even lazy lists*? To additionally illustrate your answer:

1. Rewrite the parser combinators example to use regular monad-plus and even lazy lists.
2. Select one example from Lecture 8 and rewrite it using lazy-monad-plus and odd lazy lists.