Notes regarding my code response: thenDo.ml

1. Logging a trace of execution is an output type of side effect
   a. a function with logging would have a return type which includes the output and/or a function that handles the side effect/output/logging
   b. a function of this type is a standard output variation Monad, implemented in OCaml as a record (*I chose record over 2-tuple pair because I wanted to take advantage of the unordered nature of the record, and also wanted to be more descriptive in my code using record labels to document more explicitly):*

```
Type M a = (Output, a)
OCaml:  type ('a, 'b) logger = {out:'a; res:'b}
```

*because OCaml cannot accept the capital-L of your example as an identifier, I went to the more descriptive identifiers *out* (output), *res* (result) and ‘a ‘b

2. applying the logger monad, we can rework the cube and sine functions….

```
type M a = (Output, a)
cube'' :: Logger → Logger
```

   but where do we get a logger type to pass to cube'' ?

3. If we want to pass a native type value to one of these functions as a generic, we need to transform native types to type logger:

```
promoteType ’a :: ‘a → Logger(‘a) = (“”,’a)
OCaml:  let type promoteType x = {" ", x}
```

4. It would be clunky to require implementations of f(x) such as cube(x) to be rewritten to handle instead of x an argument of type logger, so we need a function to transform existing functions this way. Fortunately promoteType can already consume a function of type (‘a → ‘b)

```
liftFunc f :: (‘a  →  ‘b) → (Logger → Logger)
OCaml:  let liftFunc f x = promoteType (f(x))
```

5. Function composition of functions with output of type logger require extra work, especially since we want to use liftFunc to keep the composition more opaque to users

```
let compose f g = fun x -> f (g x)
```

   * produces an error for f :: *Int* → (Output, *Int*)

```
let composeWithLogging f g = fun x ->
     let gx=g(x) in
     let gs=output(x) in
     let gr=result(x) in
     let fy=g(gr) in
     let fs=output(fy) in
     let fr=result(fy) in
          {out=gs^fs; res=fr};;
```

6. composeWithLogging returns a prefix function. In the example, you apply an infix function thenDo'. OCaml support for infix functions is limited and requires tapping additional libraries; to keep this implementation more simple, I rework composeWithLogging as thenDo', which take *t*, a logger type that has already been returned from a function, and applies it to g (I also change some notation, *gs* becomes *go*, to be more consistent with my identifier scheme):

```
let thenDo' t g=
     let fo=output(t) in
     let fr=result(t) in
     let gx=g(fr) in
     let go=output(gx) in
     let gr=result(gx) in
          {out=fo^go; res=gr};;
```

   Then overload the >> infix operator OCaml-style to apply thenDo' :

```
     let (>>) = thenDo';;
```

7. An example of >> reads nicely and implies thenDo readability

```
Haskell: (cube'' 3) thenDo' sine'' thenDo' (liftFunc round)
OCaml:     cube''(3) >> sine'' >> round'' ;;
           promoteType(3) >> cube'' >> sine'' >> round''
```

8. A final note… in this implementation *Output* is of type *String*, and thenDo’ concatenates strings. Since this could theoretically nest any number of times, this implementation may not be memory safe. It might be preferable to implement *Output* as a list of strings for example, and handle that list when evaluation is required.