Dealing with lexical scope

- We need lexical scope, of course, so we need functions to be represented as closures which carry an environment of their free variables around.
- We can't do lambda lifting, which a compiler would do, because it would alter what the program prints as, and we need to preserve that.
- We need to do this closure-building at parse time, by keeping an environment whilst converting from the OCaml tree, and inserting bindings into the Fun and Function nodes, otherwise too much work would be required for each step.
- When we evaluate, we then have to use these environments on entry to these function bodies, in place of the current environment.
- Are we sure nothing can get computed twice when we make this environments? e.g code which is executed at module-load time?

Explicit Let bindings are too clumsy

At the moment, we are having trouble getting a clean pattern match, because we might want to evaluate, for example:

```
(let x = 1 in let y = 2 in (fun p -> p + x + y)) 3
```

This is not of the form `App (f, x)` but `App (Let (Let (_, _), f), x)` so is hard to pattern-match. We don't want to do substitution, because there may be much bigger values than '1' and '2' and this would make the visualization awkward. So instead of the data type

```
type t =
  Int of int
| Let of (bool * binding list * t)
| ...
```

Instead we give every expression zero or more let-bindings, which can be taken to surround it:

```
type te =
  Int of int
| ...

and t =
  {te : te;
   lets : binding list}
```

So to evaluate something of type `t` under this new scheme, we first evaluate any of the lets which do not yet have a value for a RHS (since this is strict evaluation), then we can evaluate the expression of type `te` with a simple pattern match – no lets getting in the way, just using the lets as a way to build up the environment.

Now that we have environments and lets unified in this way, we can consider ways to improve the efficiency of the step-by-step interpreter so that its asymptotic complexity might be no worse that a normal interpreter.