Here is a first attempt at defining *ParseFct*.

```
functor ParseFct(
  structure SymTbl: SymTblSig
  structure Lex: LexSig) =
struct
  ...
   let val next = Lex.getsym()
   in SymTbl.update(table, next, "declared")
   end
end
```

However, the `let` expression in the body is not type correct! Since the type of *getsym*() is *Lex.Sym.sym*, *next* has type *Lex.Sym.sym*. However, by the specification of *update*, its second argument must be of type *SymTbl.Sym.sym*. The problem is that although we have specified that *SymTbl* depends on a *Sym* structure and *Lex* depends on a *Sym* structure, nowhere have we specified that they depend on the **same** *Sym* structure. The type checker will not make an attempt to identify these two types, for the idea is that the functor should be applicable to *any* arguments that satify the formal parameter specification (not just those that satisfy the specification and in addition have extra sharing). Therefore one is allowed to specify needed sharing as well as needed components by a so-called SHARING SPECIFICATION. Grammatically, a sharing specification can occur anywhere amongst structure, type, value and exception specifications.

```
functor ParseFct(
  structure SymTbl: SymTblSig
  structure Lex: LexSig
   sharing SymTbl.Sym = Lex.Sym) =
struct
  ...
   let val next = Lex.getsym()
   in SymTbl.update(table, next, "declared")
   end
end
```

One can specify sharing of structures and of types (but not of values or exceptions). In our example, we have to add yet a sharing specification, this time a type sharing specification.

```
functor ParseFct(
  structure SymTbl: SymTblSig
  structure Lex: LexSig
   sharing SymTbl.Sym = Lex.Sym
   and type SymTbl.Val.value = string) =
struct
  ...
   let val next = Lex.getsym()
   in SymTbl.update(table, next, "declared")
   end
end
```

2.7 Building the System

Notice that we have now written the code of the parser solely by declaring signatures and functors. We have not had to write a single top-level structure declaration. Having finished declaring the parser functor, we can return to the basics and declare functors that implement *Sym* and *Val*. These functors can be NULLARY, i.e. have an empty specification of formal parameters.

Exercise 13 Write nullary functors *ValFct*, *SymFct* and *IntMapFct* whose result match your signatures from the previous exercise.

We can now build the entire system by functor applications and top-level structure declarations.

```
structure Val = ValFct()

structure Sym = SymFct()

structure TTable =
  SymTblFct(structure IntMap=IntMapFct()
            structure Val = Val
            structure Sym = Sym

structure Lex = LexFct(Sym)

structure Parser =
  ParseFct(structure SymTbl = TTable
           structure Lex = Lex)
```

The compiler will check that the sharing specified in the declaration of *ParseFct* really is met by the actual argumets.

Exercise 14 What is wrong with the following attempt to build the parser?

```
structure Val = ValFct()

structure TTable =
  SymTblFct(structure IntMap=IntMapFct()
            structure Val = Val
            structure Sym = SymFct()

structure Lex = LexFct(SymFct())

structure Parser =
  ParseFct(structure SymTbl = TTable
           structure Lex = Lex)
```

2.8 Separate Compilation

Some ML implementations have facilities that allow you to compile declarations, for instance functor declarations, in such a way that the compiled code can persist between sessions. However, even without such a facility, using signatures and functors in the manner described above gives the valuable ability to separately compile modules consisting of signature and functor declarations, although the result of the compilation will not outlive the session.

Most ML systems have a *use* function which allows the ML source to be read from a file rather than from the terminal. One can then keep signatures in suitably named files and use these files at the beginning of each module.

```
use "symb.sig";
use "val.sig";
use "symtbl.sig";
use "lex.sig";
use "parse.sig";

functor ParseFct(
  structure SymTbl: SymTblSig
  structure Lex: LexSig
   sharing SymTbl.Sym = Lex.Sym
   and type SymTbl.Val.value = string): ParseSig =
struct
  ...
end
```

In this way one avoids repeating the same signature declaration in many files (and thus also the problem of updating all copies if the signature is changed).

2.9 Good Style

It is good practice to keep signatures as small as possible. If one programs using functors and signatures as described above then writing the body of a functor will reveal which components of its formal parameters that particular functor needs to know about.

Different functors will need different details. Rather than gradually extending a single signature till it gets very large, one can use the `include` specification to enrich an existing signature.

```
signature SmallTbl = sig ... end

signature BigTbl =
sig
 include SmallTbl
 datatype DebugInfo = ...
 val printInfo : unit->unit
end
```

2.10 Bad Style

Signature declarations can contain free structure and type identifiers.

Structure declarations can contain free identifiers of any kind.

This allows you to write for example

```
structure Parser: ParseSig= ParseFct(...)
```

Unfortunately it also allows you to write things like

```
structure Parser =
struct
 structure Lex = Lex
 structure MyPervasives = MyPervasives
 structure ErrorReports = ErrorReports
 structure PrintFcns = PrintFcns
 structure Table = Table
 structure BigTable =BigTable
 structure Aux = Aux

 fun f(...) = ... Table.lookup ...
end
```

Here, the programmer has apparently made some effort to show that the parser depends on the structures listed at the beginning. However, if he has missed out a couple of structures from his list, it will have no effect on the declarations that follow, and so one does not as a reader feel confident that the list is exhaustive.

Moreover, when the reader wants to find out what the type of the *lookup* function is, he has to look in the declaration of the *Table* structure. In case *Table* is constrained by a signature, the search continues in the declaration of the signature. Otherwise, one will have to look at the code for *lookup*.

Most serious of all, when encountering the call of *lookup* one has no idea whether *lookup* has side effects that are important to other structures. In that case, the value of structuring code into structures and substructures is purely cosmetic. The only reliable help it gives you is a pointer to the structure in which the identifier is declared.

One particular horror is the misuse of `open`. Avaiable both in the core language and in the modules language, `open` S is a declaration which has the effect of adding all the bindings

of the structure S to the current environment. This is helpful, if one has a single structure *MyPervasives*, say, which is used everywhere in the project. But look at this:

```
structure Parser =
struct
 structure Lex = Lex
 open MyPervasives ErrorReports PrintFcns
      Table BigTable Aux

 fun f(...) = ... lookup ...
end
```

Now finding *lookup* is reduced to pure guesswork!

Exercise 15 For each of the above points of criticism, consider to what extent it applies if one programs with signatures and functors only.

3 The Static Semantics of Modules

The purpose of this lecture is to explain the static semantics of modules. In particular, we shall look into the details of the crucial concepts SIGNATURE MATCHING and SHARING.

3.1 Elaboration

Consider the two following signatures, the first of which stem from the *MyTbl* example of Lecture 1.

```
sig
 type table
 exception Lookup
 val lookup: table * Identifier.sym
             -> Data.value
 val update: table * Identifier.sym
             * Data.value -> table
end

sig
 type table
 exception Lookup
 val lookup: table * string -> real
 val update: table * string * real -> table
end
```

In one sense, these signatures are very different; the meaning of the first one depends on the free structures *Data* and *Identifier*, whereas the second depends on the pervasives only. However, if *Identifier*.*sym* happens to be *string* and *Data*.*value* happens to be *real* then the two expression are just different ways of expressing the same meaning. In that sense, the two signatures turn out to be equal.

To avoid such confusion concerning equality, it is often helpful to distinguish between a SIGNATURE EXPRESSION (the syntactic object) and a SIGNATURE (its meaning). The transition from signature expressions to signatures is called ELABORATION. We use the word elaboration instead of evaluation, because, unlike evaluation, all elaboration can be done statically, by a compiler. The result of elaborating a signature expression depends on the meaning of the identifiers occurring free in the expression. In any given context, there are infinitely many signature expressions that elaborate to the same signature. It is even the case that in every context, every signature expression elaborates to infinitely many signatures, if it elaborates to any at all. However, among these there will always be some that are PRINCIPAL which means that, in a certain technical sense, all the others are instances of them, and one always takes a principal signature as the meaning of a signature declared at top-level.

Elaboration applies to STRUCTURE EXPRESSIONS and FUNCTOR DECLARATIONS as well, yielding STRUCTURES and FUNCTOR SIGNATURES, respectively.

Essentially, the modules part of ML is a language for computing these (abstract) signatures, structures and functor signatures. The purpose of this lecture is to explain the principles that govern elaboration.

We shall not introduce a separate notation for structures, signatures and functor signatures. In many cases these are very similar to the expressions from which they were obtained, so we make do with the device of "decorating" expressions with so-called names. Names are semantic objects, completely distinct from identifiers; in the above examples, *string* and *Identifier*.*sym* are both identifiers which elaborate to the same type name.