"modules" and in ML we use the word "structure".

Likewise, there is no standard terminology for "the type of a module", which has aquired names such as "package description", "interface" and the ML term "signature".

As we shall see, the real power of a modules system comes from the ability to parameterise modules. Ada has "generic packages". ML has "functors".

In ML, a structure is a collection of data types, types, values, exceptions and even other structures. A signature specifies types and data types and gives the types of values and exceptions. A functor is essentially a function from structures to structures. Functors cannot take functors as arguments, nor can they produce functors as results. The purpose of this lecture is to convince you that even this apparently simple notion of functor constitutes a powerful extension of the core language. As will be demonstrated, one can write an entire system using just signatures and functors and then build the system using functor applications.

Imagine we want to write a parser for a programming language. In order to build the parser top-down, we might start by sketching the parser itself. However, as programming normally is a complex process involving both the odd low-level implementation idea and more high-level considerations about overall structure, let us start at an intermediate level, the problem of writing a symbol table. (A symbol table is simply a facility which allows one to store and retrieve information about symbols.)

2.2 Signatures

The way one in ML sketches a structure is to write down a signature. Here is a first sketch of a symbol table signature, called *OTable* as it is opaque in the sense that it does not reveal many implementation details.

```
signature OTable =
sig
  type table
  exception Lookup
  val lookup: table * Sym.sym -> Val.value
  val update: table * Sym.sym * Val.value
                  -> table
end
```

At this early stage, we cannot know exactly what symbols are going to be; nor can we know what kind of values we are going to store with the symbols. Therefore we imagine structures *Sym* and *Val* which declare the types *sym* and *value*, respectively. *Sym.sym* is an example of a LONG IDENTIFIER, in this case a long type constructor. The two structure identifiers *Sym* and *Val* are FREE in *OTable*.

There are many different ways of implementing a symbol table which matches this signature. One possibility is to use an association list, i.e. a list of pairs of symbols and values. Since the symbol table is going to be used extensively, we will probably want something more efficient. We cannot use an array, for arrays map integers (rather than symbols) to values. (Actually, the ML language definition does not include arrays, but they are provided in most implementations). But we can implement the symbol table as a hash table: we can require that the *Sym* structure provides a hash function from symbols to integers and then assume the existence of another structure, *IntMap*, which implements maps on the integers. Since the hash function may map different symbols to the same integer, we take an *IntMap* which maps integers to lists of pairs of symbols and values:

```
signature TTable =
sig
 datatype table = TBL of
 (Sym.sym * Val.value) list IntMap.map
 exception Lookup
 val lookup: table * Sym.sym -> Val.value
 val update: table * Sym.sym * Val.value
                -> table
end
```

2.3 Structures

Here is a structure which implements a symbol table.

```
structure SymTbl =
struct
 datatype table = TBL of
 (Sym.sym * Val.value) list IntMap.map
 exception Lookup

 fun find(sym,[ ]) = raise Lookup
 |   find(sym,(sym',v)::rest) =
       if sym = sym' then v
       else find(sym,rest)

 fun lookup(TBL map, s) =
   let val n = Sym.hash(s)
       val l = IntMap.apply(map,n)
   in find(s,l)
   end handle IntMap.NotFound =>
       raise Lookup
 ...
end
```

When binding a structure to a structure identifier one can impose a SIGNATURE CONSTRAINT on the structure.

```
structure SymTbl : OTable =
struct
  ...
end
```

As a result, all identifiers of the structure that are not mentioned in the signature are hidden. In the above example we hide the constructor *TBL* and the function *find*. Besides *update*, the ... in *SymTbl* may declare extra values, exceptions and types, but as a result of the signature constraint, none of these extra components will be visible from outside the structure.

It is often the case that there is not a single signature which "best" constrains a given structure because different parts of the program should see different degrees of details of the structure. (The parser should be written using a opaque signature for the symbol table; by contrast, a structure which prints out the symbol table (for testing, for example) will need to know more details).

During the design of ML it was decided that it is vital to admit different views of the same structure. One way of achieving this is to bind the structure to more that one structure identifier, each time using a different signature constraint.

```
structure SymTbl : TTable =
struct
 datatype table = TBL of
 (Sym.sym * Val.value) list IntMap.map
 exception Lookup
 fun find(sym,[ ]) = raise Lookup
 |    find(sym,(sym',v)::rest) =
       if sym = sym' then v
       else find(sym,rest)

 fun lookup(TBL map, s) =
   let val n = Sym.hash(s)
        val l = IntMap.apply(map,n)
   in find(s,l)
   end handle IntMap.NotFound =>
        raise Lookup
 ...
end

structure SmallTbl: OTable = SymTbl
```

The dynamic evaluation of the `struct...end` yields an environment just as if we had typed the constituent declarations at top-level. Dynamically, there is just one *lookup* function, for example, and as a result of the above declarations, this function is shared between *SymTbl* and *SmallTbl*.

Statically, however, the elaboration of the above declarations yields two different views of this environment. Since there is just one *lookup* function, we should of course be free to refer to it as *SymTbl.lookup* or *SmallTbl.lookup*, whichever we prefer. This requires that these two long identifiers have the same type; so the static semantics must be such that the types *SymTbl.table* and *SmallTbl.table* are considered shared.

2.4 Functors

Unfortunately, neither the declaration of the signature *OTable* nor the declaration of the structure *SymTbl* makes sense on its own. The reason is that they both contain free identifiers. *OTable* relies on structures *Val* and *Sym* and *SymTbl* in addition relies on *IntMap*. As a consequence, we can compile neither *OTable* nor *SymTbl* in the initial top-level environment.

What we need to achieve this is clearly the ability to abstract both *OTable* and *SymTbl* on their free identifiers. Such an abstraction is called a FUNCTOR in ML:

```
functor SymTblFct(
 structure IntMap: IntMapSig
 structure Val: ValSig
 structure Sym: SymSig):

sig
 type table
 exception Lookup
 val lookup: table * Sym.sym-> Val.value
 val update: table * Sym.sym * Val.value
                -> table
end =

struct
 datatype table = TBL of
 (Sym.sym * Val.value) list IntMap.map
 exception Lookup

 fun find(sym,[ ]) = raise Lookup
 |    find(sym,(sym',v)::rest) =
       if sym = sym' then v
       else find(sym,rest)

 fun lookup(TBL map, s) =
   let val n = Sym.hash(s)
        val l = IntMap.apply(map,n)
   in find(s,l)
   end handle IntMap.NotFound =>
        raise Lookup
 ...
end
```

Now the *Sym*, *Val* and *IntMap* that occur in the result signature and in the functor body are bound as formal parameters of the functor. Of course for this functor declaration to make sense, we must first declare the three signatures *IntMapSig*, *ValSig* and *SymSig*, but that can be done without worrying about how we get the corresponding structures. Indeed, declaring these signatures is healthy exercise, as it makes us summarise what a symbol table needs to know about symbols, values and intmaps.

Exercise 12 Declare the signatures *IntMapSig*, *ValSig* and *SymSig*. Also complete the functor body by declaring *update*, extending your signatures, if needed.

When, in due course, we have defined actual structures *FastIntMap*, *Data* and *Identifier*, say, corresponding to the formal structures *IntMap*, *Val* and *Sym*, respectively, we can obtain a particular symbol table by applying the symbol table functor.

```
structure MyTbl =
 SymTblFct(structure IntMap = FastIntMap
              structure Val = Data
              structure Sym = Identifier)
```

Dynamically, the functor body is not evaluated when the functor is declared, but once for each time the functor is applied. (In this respect, functors behave a functions in the core language.)

As part of the functor application, the compiler will check to see whether the actual argument structures match the specified signatures. If that is not the case (for instance, if *Identifier* does not contain a type *sym* or *FastIntMap .apply* takes three instead of two arguments) then an error will be reported and hence we are prevented from putting together the inconsistent structures.

What is the signature of *MyTbl*? It is not simply the result signature of *SymTblFct*, for that signature refers to the formal functor parameters *Val* and *Sym*. Clearly, if *Identifier.sym* is *string* and *Data.value* is *real*, then we should be able to write for instance

```
 sqrt(MyTbl.lookup(t, "pi"))
```

The signature of *MyTbl*, therefore, is obtained by substituting the types of the actual arguments for the types in the formal result signature of *SymTblFct*.

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

2.5 Substructures

As we saw above, the signature of the result of a functor application depends on the actual arguments to the functor. So apparently there is no single signature which describes all the symbol tables which can be created by applying *SymTblFct*. But how, then, are we going to declare a functor *ParseFct*, say, which we can apply to *any* symbol table created by *SymTblFct*?

The solution to this problem is to make explicit in the symbol table signature that any symbol table depends on a *Val* and a *Sym* structure.

```
signature SymTblSig =
sig
 structure Val: ValSig
 structure Sym: SymSig
 type table
 val lookup: table * Sym.sym-> Val.value
 val update: table * Sym.sym * Val.value
               -> table
end
```

The specifications of *Val* and *Sym* no longer refer to particular structures outside the signature, i.e. *Val* and *Sym* are now considered bound in the signature. (Of course the signature identifiers *SymSig* and *ValSig* are still free, but those you have already declared in the exercise.)

The idea is that a structures can contain not just values, exceptions and types as components, but even other structures. These are called the SUBSTRUCTURES of the structure. To make the result of *SymTblFct* match *SymTblSig*, we have to declare structures *Val* and *Sym* in the body. But that is easily done; we simply bind them to the formal parameters.

```
functor SymTblFct(
 structure IntMap: IntMapSig
 structure Val: ValSig
 structure Sym: SymSig) : SymTblSig =
struct
 structure Val = Val
 structure Sym = Sym
datatype table = TBL of
(Sym.sym * Val.value) list IntMap.map
exception Lookup

fun find(sym,[ ]) = raise Lookup
|    find(sym,(sym',v)::rest) =
      if sym = sym' then v
      else find(sym,rest)

fun lookup(TBL map, s) =
  let val n = Sym.hash(s)
       val l = IntMap.apply(map,n)
  in find(s,l)
  end handle IntMap.NotFound =>
       raise Lookup
...
end
```

2.6 Sharing

A signature for lexical analysers might be as follows (a lexical analyser reads individual characters from an input file and assembles them into symbols — in the case of a programming language typically reserved words and identifiers):

```
signature LexSig =
sig
 structure Sym : SymSig
 val getsym : unit -> Sym.sym
end
```

We have included the specification of a substructure *Sym* because a lexical analyser needs to know about symbols. (Indeed, if we want to declare *LexSig* before defining any particular *Sym* structure, we are *forced* to include the substructure specification.