```
structure Heap =
 struct
  type item = int;
  fun leq(p: item: q: item): bool = p <= q;
  fun max(p, q) = ...
  and min(p, q) = ...
  datatype tree = L of item
                | N of item * tree * tree;
  val t = ...
  fun top(L i) = ...
  fun depth(L _) = ...
  fun isHeap(L _):bool = ...
  val initial = 0
  exception InitHeap
  fun initHeap n = ...
  fun replace(i, h) = ...
  and insert(i, L _) = ...
 end; (* Heap *)

 val smallHeap = Heap.initHeap(1);
 Heap.replace(20, smallHeap);
```

Identifiers declared in a structure are accessed from outside the structure by a LONG IDENTIFIER, for instance *Heap*.*initHeap* (which can be read "the initHeap in Heap" or just "Heap dot initHeap"). In a large program containing many structures, long identifiers make it much easier for the reader to find the definition of an identifier.

1.6 Signatures

The "type" of a structure is called a SIGNATURE. A signature can specify types, without necessarily saying what the types are. Moreover, one can specify values (in particular functions) by specifying a type for each variable, without saying how such a specification can be met by an actual declaration.

```
signature HEAP =
 sig
  type item
  val leq: item * item -> bool
  val max: item * item -> item
  val min: item * item -> item
  datatype tree = L of item
                | N of item * tree * tree
  val t: item
  val top: tree -> item
  val depth: tree -> int
  val isHeap: tree -> bool
  val initial: item
  exception InitHeap
  val initHeap: int -> tree
  val replace: item * tree -> item * tree
  val insert: item * tree -> tree
 end; (* HEAP *)
```

As one can check, the structure *Heap* MATCHES signature *HEAP* in the following sense: for every type specified in *HEAP*, there is a corresponding type in *Heap*; for every exception specified in *HEAP*, there is a corresponding exception in *Heap*; and for every value specified in *HEAP* there is a corresponding value in *Heap* which has the specified type.

1.7 Coersive Signature Matching

However, the signature *HEAP* reflects details of the implementation in *Heap* which heap users should not have to worry about. (Obviously, the value t is completely unnecessary, and there is no reason why users should have access to the constructors L and N given that we have already given the user *initHeap* and *replace*.) By pruning the signature we obtain

the following shorter declaration of *HEAP*.

```
signature HEAP =
 sig
  type item
  val leq: item * item -> bool

  type tree
  val top: tree -> item
  exception InitHeap
  val initHeap: int -> tree
  val replace: item * tree -> item * tree
 end; (* HEAP *)
```

This is a much cleaner interface, so whenever we refer to *HEAP* in the following, we mean this version.

In practice, one should write down a signature *before* one attempts to write down a structure which matches it. In this way one can decide what types and operations are needed without having to think about algorithms at the same time. So let us assume that we started out by declaring the *HEAP* signature. We then imprint the view provided by *HEAP* on the declaration of the structure *Heap* by a SIGNATURE CONSTRAINT:

```
structure Heap: HEAP =
struct
  type item = int;
  ...
end; (* Heap *)

? Heap.t
Heap.replace(7, Heap.initHeap 3);
```

After this declaration of *Heap*, we cannot write *Heap*.*t*, since *t* is not mentioned in *HEAP*. However, we can write *Heap*.*replace* as *replace* is specified. Moreover, although *HEAP* does not specify that *item* should be *int*, the ML system discovers that *item* is in fact *int* in *Heap* and that is why `7` will be accepted as an *item* in the application *Heap*.*replace*(`7, `*Heap*.*initHeap* `3`). Thus a signature constraint may hide components of a structure, but it does not hide the true identity of the types declared in the structure, except that one can hide the constructors of a `datatype` by specifying it as a `type`.

1.8 Functor Declaration

Almost all of what we did for heaps containing integer items would work for a heap whose items are of a different type. More precisely, given any type *item*, any binary function *leq* on items and any *initial* item, the signature *HEAP* is satisfied by the declarations we have already written. Let us specify the general requirements of a *Heap* structure.

```
signature ITEM =
 sig
  type item
  val leq: item * item -> bool
  val initial: item
 end;
```

What we are after is a structure which is parameterised on any structure, *Item*, say, which matches *ITEM*. In ML, a parameterised structure is called a FUNCTOR. The following table contains the complete functor declaration; the new bits are in bold face.

```
functor Heap(Item: ITEM): HEAP =
struct
 type item = Item.item
 fun leq(p: item, q: item): bool =
      Item.leq(p,q)
 fun intmax(i: int, j) =
     if i <= j then i else j
 infix leq;
 fun max(p, q) = if p leq q then q else p
 and min(p, q) = if p leq q then p else q
 datatype tree = L of item
               | N of item * tree * tree;
 fun top(L i) = i
 |   top(N(i, _, _)) = i;
 fun depth(L _) = 1
 |   depth(N(i, l, r)) =
       1 + intmax(depth l, depth r);
 fun isHeap(L _):bool = true
 |   isHeap(N(i, l, r)) =
       i leq top l andalso
       i leq top r andalso
       isHeap l andalso
       isHeap r
 exception InitHeap
 fun initHeap n =
     if n<1 then raise InitHeap
     else if n = 1 then L(Item.initial)
     else let val t = initHeap(n - 1)
          in  N(Item.initial, t, t)
          end
 fun replace(i, h) = (top h, insert(i, h))
 and insert(i, L _) = L(i)
 |   insert(i, N(_, l, r))=
          if i leq min(top l, top r)
          then N(i, l, r)
          else if (top l) leq (top r) then
               N(top l, insert(i, l) , r)
          else (* top r < min(i, top l) *)
               N(top r, l, insert(i, r));
end; (* Heap *)
```

In the first line, *Item* is the PARAMETER structure of the functor and *HEAP* is the RESULT SIGNATURE of the functor. The BODY of the functor is everything after the = in the first line.

Notice that we included declarations of *item* and *leq* in the body of the functor; since the result signature specifies them, they must be provided. If you read the body carefully, you will see that it makes sense for *any* structure which matches *ITEM*.

Exercise 7 Declare a functor *Pair* which takes as a parameter a structure matching the simple signature

```
sig type coord end
```

and has the following result signature:

```
sig
 type point
 val mkPoint: coord * coord -> point
 val x_coord: point -> coord
 val y_coord: point -> coord
end
```

You do not have to name these signatures (by the use of signature declarations); they can be written down directly where you need them, if you prefer.

Exercise 8 When the author first tried to write the *Heap* functor, he simply copied the original depth function which used *max*, not **intmax.** However, the type checker did not let him get away with that. Why?

1.9 Functor Application

We can now get various heaps (indeed heaps of heaps) by applying the *Heap* functor to different argument structures. Of course, we can only apply it to structures that match *ITEM*; this will be checked by the compiler.

Here is how one can get a string heap:

```
structure StringItem =
struct
 type item = string
 fun leq(i:item, j) =
   ord(i) <= ord(j)
 val initial = " "
end;

structure StringHeap = Heap(StringItem)

val (out1, t1) =
    StringHeap.replace("abe",
     StringHeap.initHeap(1));
val (out2, t2) =
    StringHeap.replace("man", t1);
```

The pervasive *ord* function applied to a string s returns the ASCII ordinal value of the first character in s, and raises exception `Ord` when s is empty.

Exercise 9 Declare a structure *IntItem* using the declarations we originally used for integer heaps. Then obtain a structure *IntHeap* by functor application.

Exercise 10 How does one get an integer heap whose top is always *maximal*?

Exercise 11 Declare a structure *IntHeapHeap* whose items themselves are integer heaps. (You can use the *top* function to define a *leq* function on integer heaps.)

1.10 Summary

ML consists of a CORE LANGUAGE and a MODULES LANGUAGE. The core language has values (functions are values), data types, type abbreviations and exceptions. The modules language has structures, signatures and functors. There is no actual language construct called a module, but ML programmers often refer to a module meaning "a structure or a functor".

2 Programming with ML Modules

2.1 Introduction

This lecture gives a more thorough introduction to the modules part of ML and describes a methodology for programming with its main constructs: structures, signatures and functors.

The core language is interactive: you type a declaration, get a reply, type another declaration and so on, thus gradually adding more and more bindings to the top-level environment. If we could think strictly bottom-up, declaring one value or type in terms of the preceding values and types, without ever making unfortunate implementation decisions or losing the perspective of the entire project, then this gradual expansion of the top-level environment would be quite sufficient. Unfortunately, we cannot, indeed a program which is written as one long list of core language declarations can easily end up looking rather like a long shopping list where items have been added in the order they came to mind.

Regardless of whether a programming language is interactive or not, one needs the ability to divide large programs into relatively independent units which can be written, read, compiled and changed in relative isolation from each other.

One approach, taken by some, is to provide more or less language independent software packages that help programmers organise collections of programs typically by allowing (or forcing) them to document their programs in specific ways. The crucial problem with this approach is of course to ensure consistency between the documentation and the programs, in particular to ensure that the information held by the tool really is sufficient to ensure that the constituent units can be put together in a consistent manner.

Another approach, taken in several programming languages (e.g. Ada and ML), is to provide module facilities in the programming language itself. Many of the operations one needs when programming with modules are similar to operations one needs when programming in the small, so many ideas from usual programming languages apply to programming in the large as well. For instance, just as it is a type error (in the small) to add *true* and 7, say, so it is a type error (in the large) to write a module M2, say, assuming the existence of a module M1 which provides a function f, and then combine M2 with an actual module M1 which either does not provide any f or provides an f of the wrong type. The idea is that such mistakes should be detected by a type checker at the modules level.

This leads to the exciting idea of having just one language with constructs that work uniformly for "small" as well as for "large" programs. One such language is Pebble by Burstall and Lampson. In Pebble records can contain types, so a module consisting of a collection of types and values is now itself a value, which for example can be passed as an argument to a function. There are some trade-offs, however. The ML type checker is based on a strict separation of run-time and compile-time. In designing the modules language it has been necessary to restrict the operations on types in comparison with the operations on values in order to maintain the static type checking. This has led to a stratified language, in which the modules language contains phrases from the core language, but not the other way around.

I shall use the term "module" rather vaguely to mean "a relatively independent program unit". In particular languages they have been called "packages", "clusters",