Modules

To build large systems, need to split code into separate chunks.

- Separate name spaces to avoid name clashes.

- Separation of specification from implementation to support abstraction.

- Separate compilation of portions of the system to speed development.

Standard ML supports these with a **module** system comprised of **structures**, **signatures**, and **functors**.

Not specifically tied to functional programming.

Currently being revised for new version of SML.

Moscow ML supports only a simpler, file-based module system with no functors.

Referring to Structure Components

Referring to Structure Components To refer to the **members** (types and functions) of the structure from outside its definition, we use **dot notation**:

```
- val a = Set.insert Set.empty 3;
val a = SET [3] : int Set.Set
- Set.member a 2;
val it = false : bool
```

To make such references, `Set` must already have been defined.

If we're going to make lots of use of a particular structure, we can `open` it, allowing its members to be referenced without naming the structure.

```
- open Set;
open Set
val empty = SET [] : 'a Set
val insert = fn : 'a Set -> 'a -> 'a Set
val member = fn : ''a Set -> ''a -> bool
val remove = fn : ''a Set -> ''a -> ''a Set
- val b = insert empty 42;
val b = SET [42] : int Set
```

Of course, this removes the benefit of each structure having a separate namespace, so `open` should be used sparingly. Since `open` behaves like a declaration, it can be used in a `let` or `local`, and this is generally preferable to a top-level use.

Structures and Signatures

Basic notion of a module is a collection of types, functions, and values grouped together into a named **structure**.

```
- structure Set =
  struct
    datatype 'a Set = SET of 'a list
    val empty = SET nil
    fun insert (SET s) x = SET(x::s)
    fun remove (SET s) x = SET(filter (fn y => y <> x) s)
    fun member (SET s) x = exists (fn y => y = x) s
  end ;
structure Set :
  sig
    datatype 'a Set
      con SET : 'a list -> 'a Set
    val empty : 'a Set
    val insert : 'a Set -> 'a -> 'a Set
    val member : ''a Set -> ''a -> bool
    val remove : ''a Set -> ''a -> ''a Set
  end
```

Each structure has a **signature**; in this case, an (anonymous) default one reported back by the system. The signature is a list of type declarations and typed value declarations. More later.

Libraries

The SML standard library is organized as a set of tt structures some pre-opened at top level and others not:

Bool	*open*
Integer	*open*
Real	*open*
List	*open*
String	*open*
Ref	*open*
IO	*open*
General	*open*
Array	*not open*
Vector	*not open*
RealArray	*not open*
ByteArray	*not open*
Bits	*not open*

We can refer to individual elements of these structures (whether already open or not) using the dot notation. We can also `open` any of them; note that re-opening something like `Integer` will destroy overloading.

Similarly, the New Jersey library is a further set of structures. Note that the names of files containing structure definitions don't actually matter, although by convention we usually put one structure in one file with a similar name. (Moscow ML differs.)

More on Signatures

A **signature** is a description of the contents (types, typed values) of a structure, but says nothing about the **implementation** of the structure.

Signatures can exist independently of structures, and are useful in their own right as a way of describing interfaces or specifications, e.g., for abstract data types.

```
- signature Stack =
    sig
      type 'a Stack
      exception Empty
      val empty : 'a Stack
      val push : 'a Stack -> 'a -> 'a Stack
      val pop : 'a Stack -> 'a Stack
      val top : 'a Stack -> 'a
    end;
signature Stack =
  sig
    type 'a Stack
    exception Empty
    val empty : 'a Stack
    val push : 'a Stack -> 'a -> 'a Stack
    val pop : 'a Stack -> 'a Stack
    val top : 'a Stack -> 'a
  end
```

Note that signatures can be **named** or **anonymous**.

Abstraction Properties

The module system provides three possible levels of abstraction for datatypes:

If signature itself specified a **datatype**, the structure must have an identical datatype definition, and constructors are fully visible outside.

```
- signature ExplicitSet = sig
    datatype 'a Set = SET of 'a list ... end
signature ExplicitSet =
  sig
    datatype 'a Set
      con SET : 'a list -> 'a Set
    ...
  end
- structure ESet : ExplicitSet = Set;
structure ESet : ExplicitSet
- ESet.SET [32];
val it = SET [32] : int Set.Set
```

Signature Constraints

Signatures (named or anonymous) can be used to control which parts of a structure definition are to be visible outside the structure body:

```
- structure Test : sig dotest : unit -> unit end =
struct
  type ...
  val ...
  fun ...
  fun doit () = ...
end
```

We can also use signature constraints to define multiple **views** of a single structure:

```
- signature GrowingSet =
  sig
    type ''a Set
    val empty : ''a Set
    val insert : ''a Set -> ''a -> ''a Set
    val member: ''a Set -> ''a -> bool
  end

- structure GSet : GrowingSet = Set;
```

Rule for constraints: structure must contain everything required by signature, but may contain more.

The default signature for a structure exports everything.

Abstraction Properties (Cont.)

If signature specifies a **type**, structure can implement this type in any way it likes, and constructors will not be visible outside, but name and properties of type will be (somewhat similar to a `local` declaration).

```
- signature HiddenSet = sig
  type 'a Set ... end
signature HiddenSet = sig type 'a Set ... end
- structure HSet : HiddenSet = Set;
structure HSet : HiddenSet
- HSet.empty;
val it = SET [] : 'a Set.Set
- HSet.SET [32];
std_in:27.1-27.8 Error: unbound variable
          or constructor: SET in path HSet.SET
```

To make properties of type completely invisible, use the **abstraction** keyword in place of `structure`:

```
- abstraction ASet : HiddenSet = Set;
structure ASet : HiddenSet
- ASet.empty;
val it = - : 'a ASet.Set
```

Generally, should use `abstraction` instead of `abstype`.

Functors

Often we'd like to write (and compile) code that uses a module specification (`signature`) even if no implementation (`structure`) of that specification yet exists. We can do this in ML using **functors**, which are just structures that are **parameterized** on other structures, values, or types.

For example, suppose we are writing a application involving sets, but don't want to commit to a particular implementation of the `Set` structure. We write:

```
functor MyApp (structure MySet : HiddenSet) =
struct
  fun s w = ... MySet.empty ...
  fun p z = ... MySet.insert s x ...  MySet.remove s y ...
end
```

This can be type-checked and compiled, even if there are no structures matching `HiddenSet` yet defined.

Later, we can write one or more implementations of HiddenSet:

```
structure BigFastSet : HiddenSet =
struct ... end

structure SmallSlowSet : HiddenSet =
struct .. end
```

Parameterized Set Example

```
- signature Set =
  sig
    type item
    type set
    val empty : set
    val insert : set -> item -> set
    ...
  end;
signature Set =
  sig
    ...
  end

- functor OrdSet (type item
                  val lt : item * item -> bool) : Set =
  struct
    datatype set = SET of item list
    type item = item
    val empty = SET nil
    fun insert (SET s) x = ... lt ...
    ...
  end;
functor OrdSet : <sig>
- structure IntSet : Set = OrdSet(type item = int
                                  val lt = Integer.<);
structure IntSet : Set;
- Intset.insert IntSet.empty 32;
val it = SET [32] : Iset.set
```

Functors (cont.)

Then we may choose to **apply** the functor to one (or both) of the `HiddenSet` implementations.

```
structure BigFastApp =
       MyApp(structure MySet = BigFastSet);

val answer1 = BigFastApp.s q
       (* invokes BigFastSet.empty *)

structure SmallSlowApp =
       MyApp(structure MySet = SmallSlowSet);

val answer2 = SmallSlowSet.s q
      (* invokes SmallSlowSet.empty *)
```

Functors are useful wherever abstraction at the module level is wanted.

Example: Recall the problem of implementing sets using ordered trees, where all set operations need to be parameterized by a "less-than" function.

One approach is to write the `Set` structure as a functor, parameterized by the underlying type and its `lt` predicate.

Core Language vs. Module Language

There is a close correspondence between the basic elements of core ML and the module language:

Values $\Longleftrightarrow$ Structures

Types $\Longleftrightarrow$ Signatures

Functions $\Longleftrightarrow$ Functors

Not too wrong to think of structures as **records** of values, etc. (although modules can also have **type** components).

But modules are **static** entities; all module-level calculations are performed **before** the program is "run."

Functors are similar to C++ templates, but in most implementations functor body is compiled just once; functor application can be thought of as a **link-time** step, and functors as an elaborate linking control language.