Example interface with declarations

```
signature ILIST = sig
 type 'a ilist  (* nonempty; one element indicated *)
 val singletonOf    : 'a -> 'a ilist
 val indicated      : 'a ilist -> 'a
 val indicatorLeft  : 'a ilist -> 'a ilist
 val indicatorRight : 'a ilist -> 'a ilist
 val deleteLeft  : 'a ilist -> 'a ilist
 val deleteRight : 'a ilist -> 'a ilist
 val insertLeft  : 'a * 'a ilist -> 'a ilist
 val insertRight : 'a * 'a ilist -> 'a ilist
 val ifoldl : ('a * 'b -> 'b) -> 'b -> 'a ilist -> 'b
 val ifoldr : ('a * 'b -> 'b) -> 'b -> 'a ilist -> 'b
end
```

Design choice: placing interfaces

Interface "projected" from implementation:

- **No separate interface**
- **Compiler extracts from implementation (CLU, Java *class*, Haskell)**
- **When code changes, must extract again**
- **Few cognitive benefits**

Full interfaces:

- **Distinct file, separately compiled (Caml, Java *interface*, Modula, Ada)**
- **Implementations can change independently**
- **Full cognitive benefits**

ML module terminology

Interface is a signature

Implementation is a structure

Generic module is a functor

- A compile-time function over structures
- The point: reuse *without* violating abstraction

Structures and functors match signature

Analogy: Signatures are the "types" of structures.

Signature says what's in a structure

Specify types (w/kind), values (w/type), exceptions

Ordinary type examples:

```
type t          // abstract type, kind *
eqtype t
type t = ...    // 'manifest' type
datatype t = ...
```

Type constructors work too

```
type 'a t       // abstract, kind * => *
eqtype 'a t
type 'a t = ...
datatype 'a t = ...
```

Signature example: Ordering

```
signature ORDERED = sig
  type t
  val lt : t * t -> bool
  val eq : t * t -> bool
end
```

Signature example: Integers

```
signature INTEGER = sig
  eqtype int                (* <-- ABSTRACT type *)
  val ~    : int -> int
  val +    : int * int -> int
  val -    : int * int -> int
  val *    : int * int -> int
  val div  : int * int -> int
  val mod  : int * int -> int
  val >    : int * int -> bool
  val >=   : int * int -> bool
  val <    : int * int -> bool
  val <=   : int * int -> bool
  val compare : int * int -> order
  val toString   : int    -> string
  val fromString : string -> int option
end
```

Implementations of integers

A selection...

```
structure Int    :> INTEGER
structure Int31  :> INTEGER  (* optional *)
structure Int32  :> INTEGER  (* optional *)
structure Int64  :> INTEGER  (* optional *)
structure IntInf :> INTEGER  (* optional *)
```

Homework: natural numbers

```
signature NATURAL = sig
   type nat   (* abstract, NOT 'eqtype' *)
   exception Negative
   exception BadDivisor

   val ofInt   : int -> nat
   val /+/     : nat * nat -> nat
   val /-/     : nat * nat -> nat
   val /*/     : nat * nat -> nat
   val sdiv    : nat * int ->
                 { quotient : nat, remainder : int }
   val compare : nat * nat -> order
   val decimal : nat -> int list
end
```

Homework: integers

```
signature BIGNUM = sig
   type bigint
   exception BadDivision

   val ofInt     : int -> bigint
   val negated   : bigint -> bigint
   val <+>       : bigint * bigint -> bigint
   val <->       : bigint * bigint -> bigint
   val <*>       : bigint * bigint -> bigint
   val sdiv : bigint * int ->
              { quotient : bigint, remainder : int }
   val compare   : bigint * bigint -> order
   val toString : bigint -> string
end
```