Modules

`datatype` provides concrete user-defined data types,
but with representation exposed (e.g. for pattern matching)

Modules support **encapsulated** abstract data types,
hidden operations, values, and types,
and some kinds of polymorphism

Overview:

- `structure` defines module implementation
- `signature` defines module interface,
  hides other aspects of underlying `structure`
- `open` supports importing a module, for naming convenience
- `functor` supports parameterized module implementations

Structures

Package up any set of declarations into a structure:

```
- structure Queue1 = struct
    type 'a T = 'a list; (* T is conventional name *)
    (* constructors: *)
    val empty = nil;
    fun enq x q = q @ [x]; (* @ is append *)
    (* accessors: *)
    exception empty_queue;
    fun head (x::q) = x
      | head  nil   = raise empty_queue;
    fun deq (x::q) = (x,q)
      | deq  nil   = raise empty_queue
  end;
```

Access members of structure using dot notation:

```
- val q = Queue1.enq 3 Queue1.empty;
val q = [3] : int list
- val q2 = Queue1.enq 4 q;
val q2 = [3,4] : int list
- Queue1.head q2;
3 : int
```


Importing

To avoid long qualified names, use `open`

```
- open Queue1;
- val q = enq 4 (enq 3 empty);
val q = [3,4] : int list
- let val (x,q2) = deq(q) in head q2 end;
4 : int
```

Often, limit scope of `open` using `let` or `local`:

```
- structure QueueExtension = struct
    local open Queue1 in
      fun enq2 x y q = enq y (enq x q);
      ...
    end;
  end;

- QueueExtension.enq2 3 4 Queue1.empty;
[3,4] : int list
- QueueExtension.enq 3 QueueExtension.empty;
... error: enq not in QueueExtension ...
```


Signatures

Signatures define the “type” (interface) of a module

```
- signature QUEUE = sig
    type 'a T;
    val empty: 'a T;
    val enq: 'a -> 'a T -> 'a T;
    exception empty_queue;
    val head: 'a T -> 'a;
    val deq: 'a T -> 'a * 'a T;
  end;
```

Using signatures for hiding of values & operations

Apply signature to restrict interface to structure:

```
- structure Queue2:QUEUE = struct ... end;
```

If there were values or operations in `struct` that weren't in `sig`, then these would become inaccessible

But the representation of types is still visible to clients

```
- Queue2.empty = nil;
true : bool;
```

Not a true abstract data type,
just a collection of values & operations on some existing type

Using signatures for full encapsulation

Use `:>` instead of `:` when applying signature to structure:

```
- structure Queue3:>QUEUE = struct ... end;
```

[New with SML97]

Type `Queue3.T` is unknown

- clients can't assume it's a list
  ⇒ can be changed in impl w/o affecting clients

```
- Queue3.empty = nil;
... type error ...
```

- structures w/ different choices for `T` have same signature

True abstract data types

Partially exposed representations

Can reveal parts of implementation of types, if desired, by specifying in signature

```
- signature QUEUE4 = sig
    type 'a T = 'a list;
    ...
  end;
- structure Queue4:>QUEUE4 = struct ... end;
- Queue4.empty = nil;
true : bool;
```


Functors

Can parameterize structures by other structures

```
- functor QueueUser(Q:QUEUE) = struct
    ... Q.enq ... Q.deq ...
  end;
```

- only know aspects of `Q` that are defined by `QUEUE`

Instantiate functors to build regular structures:

```
- structure QU1 = QueueUser(Queue1);
- structure QU2 = QueueUser(Queue3);
```

Functors for bounded quantification

Define a signature representing the operations needed

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

Define quantified algorithms as elements of functors parameterized by required signature

```
- functor Sort(O:ORDERED) = struct
    fun min(x,y) =
      if O.lt(x,y) then x else y;
    fun sort(lst) =
      ... O.lt(x, y) ...;
  end;
```


An instantiation of `Sort`

Create specialized sorter by instantiating functor with appropriate operations

```
- structure IntOrder:ORDERED = struct
    type T = int;
    val lt = (op <);
    val eq = (op =);
  end;
- structure IntSort = Sort(IntOrder);
- IntSort.sort([3,5,~2,...]);
```

Use `IntOrder:ORDERED`, not `IntOrder:>ORDERED`

- `IntOrder` is a view/extension of an existing type, `int`; it isn't creating a new ADT w/ only 2 operations

Another instantiation of `Sort`

Create nested, multiply parameterized functors:

```
- functor PairOrder(
      structure First:ORDERED;
      structure Second:ORDERED):ORDERED =
  struct
    type T = First.T * Second.T;
    fun lt((x1,x2),(y1,y2)) =
      First.lt(x1,y1) andalso Second.lt(x2,y2);
    fun eq((x1,x2),(y1,y2)) = ...;
  end;

- structure IntStringSort = Sort(
    PairOrder(structure First = IntOrder;
              structure Second = StringOrder));
```


Signature "subtyping"

Signature specifies a particular interface

Any structure that satisfies that interface can be used where that interface is expected

- e.g. in functor application

Structure can:

- have more operations
- have more general/polymorphic operations
- have more details of implementation of types

than required by signature

Some limitations of ML modules

Structures are not first-class values

- must be named or be argument to functor application
- must be declared at top-level or
  nested inside another structure or signature

Cannot instantiate functors at run-time to create “objects”
  ⇒ cannot simulate classes and object-oriented programming

No type inference for functor arguments

Modules vs. ADTs

Abstract data types implicitly define a single type,
  with associated constructors, observers, and mutators

Modules can define 0, 1, or many types in same module,
  with associated operations over several types

- multiple types can share private data & operations

Functors similar to parameterized ADTs

Modules more general than ADTs,
  but clumsier for common case