Modules

(Version of 27 September 2004)

A common software engineering principle is to *hide representation*, for reasons of:

- *Safety*: Stopping data objects being put in a bad state, i.e., with broken representation invariants.
- *Ease of changing the design*: If the rest of the program does not depend on the representation, then it is easy to change from, say, binary search trees to AVL trees, without affecting the rest of the program.

In addition to local declarations and abstract datatypes (ADTs), the module system in SML provides mechanisms for doing this.

Structures

Structures on their own do not provide data-hiding,
but they do provide a useful grouping and naming mechanism:

```
structure Set =
struct
  type 'a set = 'a list
  val empty = []
  fun singleton a = [a]
  fun member (x,ys) = List.exists (fn y => x = y) ys
  fun insert (x,ys) = if member (x,ys) then ys else x::ys
end
- Set.empty ;
> val 'a it = [] : 'a list
```

```
- val s1 = Set.insert (1,Set.singleton 2) ;
> val s1 = [1, 2] : int list
```

Notice the polymorphic type for the empty set,
and the fact that integer *lists* get constructed.

We can *open* or *import* a structure:

```
- open Set ;
> type 'a set = 'a list
  val 'a empty : 'a list
  val 'a singleton : 'a -> 'a list
  val ''a member : ''a * ''a list -> bool
  val ''a insert : ''a * ''a list -> ''a list
- empty ;
> val 'a it = [] : 'a list
```

Running Example: Stacks

A *stack* is a data structure where the last thing we add (to its top) is the first thing that comes off.

A stack has the following operations (or operations like them):

empty is the empty stack

push puts an item on top of the stack

pop removes an item from the top of the stack

top returns the top of the stack

Signatures

Signatures are the client-code view of structures.

We can think of them as empty structures waiting to be filled.

```
signature AbsStack =
sig
  exception EmptyStack
  type 'item stack
  val empty : 'item stack
  val pop : 'item stack -> 'item stack
  val push : 'item * 'item stack -> 'item stack
  val top : 'item stack -> 'item
end
```

Module = Signature + Structure

Via *transparent signature matching*:

```
structure Stack : AbsStack =
struct
  exception EmptyStack
  datatype 'item stack = Empty
                       | Node of 'item * 'item stack
  val empty = Empty
  fun pop (Empty) = raise EmptyStack
    | pop (Node(x,S)) = S
  fun push (x,S) = Node(x,S)
  fun top (Empty) = raise EmptyStack
    | top (Node(x,S)) = x
end
```

Usage:

```
- val s = Stack.empty ;
> val 'a s = ?{Empty} : 'a stack
- Stack.push (32,s) ;
> val it = ?{Node} (32, ?{Empty}) : int stack
```

Notice the polymorphism of `s`.

There can be help functions within a module
that are not mentioned in the signature:
their composite identifiers are then *not* bound to anything.

Hiding the Details

However, we can still get access to the details!
To hide the details, we make the declaration
via *opaque signature matching*:

```
structure Stack1 :> AbsStack =
...
```

the rest being the same as before.

Now, we get:

```
- Stack1.push (32,Stack1.empty) ;
> val it = <stack> : int stack
- Stack1.top it ;
> val it = 32 : int
```

Another Implementation: Stacks as Lists

An implementation via lists can enforce the representation convention that the head of the list is the top of the stack, etc.

```
structure Stack2 : AbsStack =
struct
  exception EmptyStack
  type 'item stack = 'item list
  val empty = []
  fun pop [] = raise EmptyStack
    | pop (x::S) = S
  fun push (x,S) = x::S
  fun top [] = raise EmptyStack
    | top (x::S)  = x
end
```

Usage:

```
- Stack2.empty ;
> val 'a it = [] : 'a list
- Stack2.push (34,it) ;
> val it = [34] : int list
- Stack2.push (35,it) ;
> val it = [35, 34] : int list
- Stack2.push (99,it) ;
> val it = [99, 35, 34] : int list
- Stack2.top it ;
> val it = 99 : int
```

Functors

Functors, or parametric modules, build structures from structures.
The general form of a functor definition goes as follows:

```
functor FunctorName (ParamName : SigName)
  = struct
      ...
    end;
```

To make a structure, we do as follows:

```
structure ResultStructure
     = FunctorName (ArgumentStructure)
```

This builds a new structure with the argument structure.
This works provided the types and signatures match up.

Functors: An Example

Let us return to the `removeSmaller` function from an earlier lecture.

First, we rewrite it as a structure:

```
structure Remove =
struct
  fun smaller a [] = []
    | smaller a (x::xs) =
      if x < a then (smaller a xs)
      else x::(smaller a xs)
end
```

But this structure is limited to integer lists...

What we need now is a thing that we can do comparisons with.

Functors build new structures out of old structures.

So we make a structure that allows us to compare things:

```
signature Ordered =
sig
  type T
  val eq  : T * T -> bool   (* equals *)
  val lt  : T * T -> bool   (* is less than *)
  val leq : T * T -> bool   (* is less than or equal to *)
end
```

```
structure Integer : Ordered =
struct
  type T = int
  fun eq  (x,y) = (x = y)
  fun lt  (x,y) = (x < y)
  fun leq (x,y) = (x <= y)
end

functor RemoveGen (s : Ordered) =
struct
  fun smaller a [] = []
    | smaller a (x::xs) =
      if (s.lt (x,a)) then (smaller a xs)
      else x::(smaller a xs)
end
```

To build a new structure, we instantiate the functor:

```
structure IntRemove = RemoveGen (Integer)
```

Usage:

```
- IntRemove.smaller 12 [12,31,14] ;
> val it = [12, 31, 14] : int list
- IntRemove.smaller 15 [12,31,14] ;
> val it = [31] : int list
```

Read Section 10.4.2 and then Section 11.6.2 of the Hansen & Rischel textbook for an example about sets with an equality predicate!

Read Section 8.2 and then Example 11.1 of the Hansen & Rischel textbook for an example about expression trees!

Why Use Functors and Structures?

- Functors are a very powerful way to organise software.
- In a type-safe way, we can build things from smaller components.
- An abstract datatype allows only one implementation.
- An abstract datatype cannot be generated by functors.