~ Lecture VI ~

Data abstraction and modularity

Reference:

- **Chapter 7** of *ML for the working programmer* (2ND EDITION) by L. C. Paulson. CUP, 1996.

ML module system

Signatures and structures

- An *abstract data type* is a type equipped with a set of operations, which are the only operations applicable to that type.

  Its representation can be changed without affecting the rest of the program.

- *Structures* let us *package* up declarations of related types, values, and functions.

- *Signatures* let us *specify* what components a structure must contain.

Example: Polymorphic functional stacks.

```
signature STACK =
sig
  exception E
  type 'a reptype   (* <-- INTERNAL REPRESENTATION *)
  val new: 'a reptype
  val push: 'a -> 'a reptype -> 'a reptype
  val pop: 'a reptype -> 'a reptype
  val top: 'a reptype -> 'a
end ;
```



```
structure MyStack: STACK =
struct
  exception E ;
  type 'a reptype = 'a list ;
  val new = [] ;
  fun push x s = x::s ;
  fun split( h::t ) = ( h , t )
    | split _ = raise E ;
  fun pop s = #2( split s ) ;
  fun top s = #1( split s ) ;
end ;
```

```
val MyEmptyStack = MyStack.new ;
val MyStack0 = MyStack.push 0 MyEmptyStack ;
val MyStack01 = MyStack.push 1 MyStack0 ;
val MyStack0' = MyStack.pop MyStack01 ;
MyStack.top MyStack0' ;
```

```
val MyEmptyStack = [] : 'a MyStack.reptype
val MyStack0 = [0] : int MyStack.reptype
val MyStack01 = [1,0] : int MyStack.reptype
val MyStack0' = [0] : int MyStack.reptype
val it = 0 : int
```


ML module system

Information hiding

In ML, we can limit outside access to the components of a structure by *constraining* its signature in *transparent* or *opaque* manners.

Further, we can *hide* the representation of a type by means of an `abstype` declaration.

The combination of these methods yields abstract structures.

Opaque signature constraints

```
structure MyOpaqueStack :> STACK = MyStack ;
```

```
val MyEmptyOpaqueStack = MyOpaqueStack.new ;
val MyOpaqueStack0 = MyOpaqueStack.push 0 MyEmptyOpaqueStack ;
val MyOpaqueStack01 = MyOpaqueStack.push 1 MyOpaqueStack0 ;
val MyOpaqueStack0' = MyOpaqueStack.pop MyOpaqueStack01 ;
MyOpaqueStack.top MyOpaqueStack0' ;
```

```
val MyEmptyOpaqueStack = - : 'a MyOpaqueStack.reptype
val MyOpaqueStack0 = - : int MyOpaqueStack.reptype
val MyOpaqueStack01 = - : int MyOpaqueStack.reptype
val MyOpaqueStack0' = - : int MyOpaqueStack.reptype
val it = 0 : int
```


abstypes

```
structure MyHiddenStack: STACK =
struct
  exception E ;
  abstype 'a reptype = S of 'a list (* <-- HIDDEN         *)
  with                              (*     REPRESENTATION *)
    val new = S [] ;
    fun push x (S s) = S( x::s ) ;
    fun pop( S [] ) = raise E
      | pop( S(_::t) ) = S( t ) ;
    fun top( S [] ) = raise E
      | top( S(h::_) ) = h ;
  end ;
end ;
```

```
val MyHiddenEmptyStack = MyHiddenStack.new ;
val MyHiddenStack0 = MyHiddenStack.push 0 MyHiddenEmptyStack ;
val MyHiddenStack01 = MyHiddenStack.push 1 MyHiddenStack0  ;
val MyHiddenStack0' = MyHiddenStack.pop MyHiddenStack01  ;
MyHiddenStack.top MyHiddenStack0' ;
```

```
val MyHiddenEmptyStack = - : 'a MyHiddenStack.reptype
val MyHiddenStack0 = - : int MyHiddenStack.reptype
val MyHiddenStack01 = - : int MyHiddenStack.reptype
val MyHiddenStack0' = - : int MyHiddenStack.reptype
val it = 0 : int
```


ML module system

Functors

- An ML *functor* is a structure that takes other structures as parameters.
- Functors let us write program units that can be combined in different ways. Functors can also express generic algorithms.

Example: Generic imperative stacks.

```
signature STACK =
 sig
  type itemtype
  val push: itemtype -> unit
  val pop: unit -> unit
  val top: unit -> itemtype
end ;
```



```
exception E ;
functor Stack( T: sig type atype end ) : STACK =
struct
  type itemtype = T.atype
  val stack = ref( []: itemtype list )
  fun push x
    = ( stack := x :: !stack )
  fun pop()
    = case !stack of [] => raise E
      | _::s => ( stack := s )
  fun top()
    = case !stack of [] => raise E
      | t::_ => t
end ;
```

```
structure intStack
  = Stack(struct type atype = int end) ;
```

```
structure intStack : STACK
```

```
intStack.push(0) ;
intStack.top() ;
intStack.pop() ;
intStack.push(4) ;
```

```
val it = () : unit
val it = 0 : intStack.itemtype
val it = () : unit
val it = () : unit
```



```
map ( intStack.push )  [3,2,1] ;
map ( fn _ => let val top = intStack.top()
              in  intStack.pop(); top  end )
    [(),(),(),()] ;
```

```
val it = [(),(),()] : unit list
val it = [1,2,3,4] : intStack.itemtype list
```