Modules

Structure Int.toString

Int ascribes to a signature called integer

Abstraction < abstract data / information hiding

↓

specified via a signature

Implementation (within a structure) < abstraction function / representation invariant

A signature specifies an interface

A structure provides an implementation.

eg. Queue signature

meaning: provides all the items specified in a

```
signature Queue =
sig
  type 'a q          (* abstract *)
  val empty : 'a q
  val enq : 'a q * 'a -> 'a q
  val null : 'a q -> bool
  exception Empty
  (* raise Empty if called on empty q *)
  val deq : 'a q -> 'a * 'a q
end
```

ascribe to

```
structure Queue : QUEUE =
struct
  type 'a q = 'a list
  val empty = []
  fun enq (q, x) = q @ [x]     O(n)
  val null = List.null
  exception Empty
  fun deq [] = raise Empty
    | deq (x::q) = (x, q)
end
```

transparent ascription

val q2 = Queue.enq (Queue.enq(Queue.empty, 1), 2)

↓

type: int Queue.q 'a is int => int q. but may be many implementation

=> use structure name: Queue

q2 = [1, 2] visible because of transparent ascription

val (a, b) = Queue.deq q2

1 [2] still [1, 2]

2nd implementation (front, back)

abstraction function: front @ (rev back)

```
structure Q :> QUEUE =
struct
    type 'a q = 'a list * 'a list
    val empty = ([], [])
    fun enq ((f,b), x) = (f, x::b)     O(1)
```

opaque ascription

```
fun null ([], []) = true
  | null _        = false
exception Empty
fun deq ([], []) = raise Empty
  | deq ([], b) = deq (rev b, [])      amortized O(1)
  | deq (x::f, b) = (x, (f, b))
```

```
val q2' = Q.enq (Q.enq (Q.empty, 1), 2)        val (a, b) = Q.deq q2
```

$q2'$ ↓ int Q.q; a ↓ 1; b ↓ [2]; $q2$ ↓ still [1, 2]

([], []) → ([], [1]) → ([], [2, 1]) → ([1, 2], []) → ([2], [])

Dictionary Signature

```
signature DICT =
sig
  type key = string             (* concrete *)
  type 'a entry = k * 'a        (* concrete *)
  type 'a dict                  (* abstract *)
  val empty : 'a dict
  val lookup : 'a dict -> key -> 'a option
  val insert : 'a dict * 'a entry -> 'a dict
end
```

Abstraction Function:
(key, value) items in the tree constitute the dictionary.

Representation Invariant:
tree must be sorted on key
all functions within the structure must assume & ensures R.I.

```
structure BST : DICT =
struct
  type key = string
  type 'a entry = key * 'a
  datatype 'a tree = Empty | Node of 'a tree * 'a entry * 'a tree
  type 'a dict = 'a tree
  val empty = Empty
  fun lookup --
  fun insert ---
end
```

```
(* insert : 'a dict * 'a entry -> 'a dict *)
fun insert (Empty, e) = Node (Empty, e, Empty)
  | insert (Node (lt, e' as (k', _), rt), e as (k, _)) =
```

layered pattern matching

```
    case String.compare (k, k') of
      EQUAL => Node (lt, e, rt)
    | LESS => insert (lt, e)
    | GREATER => insert (rt, e)
```

val d : int BST.dict   val look = BST.lookup d

String -> int option

(BST, key)

Functor

A functor expects a structure as argument, produce a structure

abstraction	signature	type
implementation	structure	value
mapping	functor	function

Type Class: a type with some operations for the type (not necessarily all oper)

```
signature ORDERED =
sig
   type t       -> not specify. type t will be some already existing types, so t is a parameter
                (* parameter *)
   val compare : t * t -> order
end
```

Concrete type: client & implementation know what the type is

Abstract: client doesn't know how the type is implemented

Parameter: client supplies the type

```
signature DICT =
sig
                   always :, no :>
   structure Key : ORDERED        (* parameter *)
   type 'a entry = Key.t * 'a     (* concrete *)
   type 'a dict                   (* abstract *)
   val empty : 'a dict
   val lookup : 'a dict -> Key.t -> 'a option
   val insert : 'a dict * 'a entry -> 'a dict
end
```

```
structure IntLtDict : DICT =
struct
   structure Key = IntLt
   ... rest use Key.compare instead of String.compare
end
```

May accidentally use IntLtDict.insert, then IntGtDict.lookup ?

(IntLtDict.Key.t & IntGtDict.Key.t are both int)

No ! IntLtDict.dict & IntGtDict.dict are different ! Type checker prevents us.

Each datatype 'a dict = ... declaration creates a new type

* If we use association list, mix will happen!

```
functor TreeDict (K: ORDERED) : DICT =
struct
  structure Key = K
  type 'a entry = Key.t * 'a
  datatype 'a dict = ...
  val empty : 'a dict
  val lookup : 'a dict -> Key.t -> 'a option
  val insert : 'a dict * 'a entry -> 'a dict
end
```

argument! if opaque, :>

```
structure IntLtDict = TreeDict (IntLt)
```

↳ hide the key type in DICT. need it to be known to be same as input key type

```
functor TreeDict (K : ORDERED) :> DICT where type Key.t = K.t
= struct ... end
```

expose types in a signature

Multiple where type are allowed

```
structure IntLtDict = TreeDict (IntLt) where type t = int
```

Syntactic Sugar

```
functor PairOrder (structure Ox : ORDERED
                   structure Oy : ORDERED) : ORDERED = ...
struct
   type t = Ox.t * Oy.t
   fun compare ((x1, y1), (x2, y2)) =
     (case Ox.compare (x1, x2) of EQUAL => Oy.compare (y1, y2)
                                | other => other)
end
```

no comma! (Ox, Oy)

⇓ desugars

```
functor PairOrder ( P: sig
                      structure Ox : ORDERED
                      structure Oy : ORDERED
                    end ) : ORDERED = ...
```

(P.Ox, P.Oy ...)

2D - Grid

```
functor PairOrder (structure Ox : ORDERED
                   structure Oy : ORDERED) : ORDERED = ...
```

```
structure GridOrder = PairOrder (structure Ox = StringLt
                                 structure Oy = IntLt )
```

Consistant! Define functor & call it with sugar

OR Define functor & call it without sugar

```
structure Board = TreeDict (GridOrder)
val b = Board.insert (Board.empty, ("A", 1), fn x => x+1 )
```

type: (int -> int) Board.dict

Key.t * 'a entry type

string * int

int -> int