Lec 17 Functors

* Overview

interface	—	type	— Signature
implementation	—	value	— Structure
mapping	—	function	— functor

└ function that operate on structure

* Fixing dict from last time

```
Signature DICT =
  Sig
    type key = string      ← problem: it's indexed by string only.
                             WT generalise key type
    type 'a entry = key * 'a
    type 'a dict
    val empty : 'a dict
    val lookup : 'a dict → key → 'a option
    val insert : 'a dict → 'a entry → 'a dict
  end
```

Try 1 (not going to work) :C

```
Signature DICT =                          ↙ Valid signature, but can't compare 'a
  Sig
    type 'a key : 'a
    type ('a,'b) entry = 'a key * 'b
    type ('a,'b) dict
    val empty : ('a,'b) dict
    val lookup : ('a,'b) dict → 'a key → 'b option
    val insert : (('a,'b) dict * ('a,'b) entry) → ('a,'b) dict
  end          ↑ we could make them take cmp func ('a * 'a → order)
```

```
⋮
fun lookup cmp d k = ... use cmp
```

↑ More problem: can throw different order at it. Don't know what will happen. Interface shouldn't allow this to happen. No good.

→ Idea 1: put cmp inside dictionary ← will work, but not cause more generalisation if we have more complex interface.

→ Idea 2: signature commits to different type, but we spawn them rather than write each of them.

Try 2:

↓ type with operations, often used as input to functor

```
type class
  Signature ORDERED =
    Sig                    ↑ descriptive: saying that we can use this if they have compare
      type t     (* parameter *)
      val compare : t * t → order
    end
```

	Impl	Client
abstract	✓	
concrete	✓	✓
param		✓

```
Structure IntLt : ORDERED =
  Struct
    type t = int
    val compare = Int.compare
  end
```

```
Structure IntGt : ORDERED =
  Struct
    type t = int
    fun compare (x,y) = Int.compare (y,x)
  end

Structure StringLt : ORDERED =
  Struct
    type t = int
    val compare = String.compare
  end


Signature DICT =
  Sig
    Structure Key : ORDERED        (* param *)
    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 dict -> 'a dict
  end

Structure IntLtDict : DICT =
  Struc
    Structure Key = IntLt          <- only place to commit type.
    type 'a entry = Key.t * 'a
      :   use Key.compare to impl
      :
      :   type 'a dict = 'a entry tree
  end
```

↑ But notice we need to repeat code if we want IntGt or StringLt, etc. here

Issues — did we solve them?

1. restrict to same cmp function

... kind of, but we used transparent struct.
exposes type so they look the same
(but actually the are different but named the same
so it works by accident

If we did `type 'a dict = (int * 'a) list`, two different can have same dict type

What if ... we do `IntLtDict :> DICT` ... lol we get a different problem — it stops client from tinkering but then client has no idea what the key type is. Duh

↳ Solution: where type to tell client

↖ Translucent signature: leak some info

```
Structure IntLtDict : DICT where type Key.t = int = ...
```

↑ Fill in param. a step towards functor

2. Code duplication. Obviously not good :(

Try3: Take in struct, return struct. Functor! (Finally)

```
functor TreeDict (K: ORDERED) :> DICT where type Key.t = K.t =
  Struct
    Structure Key = K
    type 'a entry = Key.t * 'a
    type 'a dict = ...
      :
  end
```

Calling it

```
Structure IntLtDict = TreeDict (IntLt)
Structure IntGtDict = TreeDict (IntGt)
                  ⋮
```

Problem solved

Alt Syntax ← good for multiple arguments

```
(* functor to do pair order *)
functor PairOrder ( Structure Ox : ORDERED
                    Structure Oy : ORDERED )
  :> ORDERED where type t = Ox.t * Oy.t
  = Struct
      type t = Ox.t * Oy.t
      fun compare ((x1, y1), (x2, y2)) =
          (case Ox.compare (x1, x2) of
              EQUAL => Oy.compare (y1, y2) |
              ord => ord
          )
    end

Structure GridOrder = PairOrder ( Structure Ox = StringLt
                                  Structure Oy = IntLt    )
Structure Board = TreeDict ( GridOrder )

val b = Board.insert ( Board.empty, (("A", 1), (fn x => x+1)))
    b : (int -> int) Board.dict
```

need to put Structure for multiple