Lessons:

• Parameterized Structures
• Type Classes
Lessons:
- Parameterized Structures
- Type Classes

A functor expects a structure as argument and produces a structure.
Lessons:

• Parameterized Structures
• Type Classes

A functor expects a structure as argument and produces a structure.

Simile: abstraction | signature | type
implementation | structure | value
mapping | functor | function
Before we get to functors, we need to explore some motivations.
Recall:

signature DICT =
sig
  type key = string (* concrete type *)
  type 'a entry = key * 'a (* concrete type *)

  type 'a dict (* abstract type *)

  val empty : 'a dict
  val lookup : 'a dict -> key -> 'a option
  val insert : 'a dict * 'a entry -> 'a dict
end

We had made the dictionary abstract, we allowed the entries to be arbitrary, but we fixed the keys to be strings.

What if we wanted the keys to be integers … or something else?
We could try to make the dictionaries doubly polymorphic:

signature DICT =
  sig
    type 'a key = 'a                       (* concrete type *)
    type ('a, 'b) entry = 'a key * 'b     (* concrete type *)

    type ('a, 'b) dict                    (* abstract type *)

    val empty : ('a, 'b) dict

    val lookup :
    val insert :

  end
We could try to make the dictionaries doubly polymorphic:

signature DICT =
.sig
  type 'a key = 'a
  type ('a, 'b) entry = 'a key * 'b
  type ('a, 'b) dict
  val empty : ('a, 'b) dict
  val lookup :
  val insert :
.end
We could try to make the dictionaries doubly polymorphic:

signature DICT =
  sig
    type 'a key = 'a (* concrete type *)
    type ('a, 'b) entry = 'a key * 'b (* concrete type *)

    type ('a, 'b) dict (* abstract type *)

    val empty : ('a, 'b) dict

    val lookup :
    val insert :
  end

What goes here?
We realize that we need to be able to compare values of our key type.

At the very least the key type needs some kind of equality comparison.

Ideally it should have some kind of order comparison so we can implement dictionaries using binary search trees.

How do we model that?
One possibility is to make the comparison function an argument to `insert` and `lookup`, so:

\[
\text{lookup} : (\text{'a'}\to \text{order}) \to \text{('a', 'b) dict} \to \text{'a} \to \text{'b option}
\]

\[
\text{insert} : (\text{'a'}\to \text{order}) \to \text{('a, 'b) dict} \times \text{('a, 'b) entry} \to \text{('a, 'b) dict}
\]
Then we could implement BST much as before:

structure BST : DICT =
  struct
    type 'a key = 'a
    type ('a, 'b) entry = 'a key * 'b

    datatype ('a, 'b) dict = Empty
      | Node of ('a, 'b) dict * ('a, 'b) entry * ('a, 'b) dict

    val empty = Empty

    fun lookup cmp d k =

    fun insert cmp (d, e) =

  end (* structure BST *)
Then we could implement BST much as before:

```plaintext
structure BST : DICT =
  struct
    type 'a key = 'a
    type ('a, 'b) entry = 'a key * 'b
    datatype ('a, 'b) dict = Empty
      | Node of ('a, 'b) dict * ('a, 'b) entry * ('a, 'b) dict

    val empty = Empty

    fun lookup cmp d k =
      (* implementation *)

    fun insert cmp (d, e) =
      (* implementation *)
  end  (* structure BST * )
```

Remember: These two types were specified concretely in the signature, so we need to implement them as specified.
Then we could implement BST much as before:

```
structure BST : DICT =
struct
  type 'a key = 'a
  type ('a, 'b) entry = 'a key * 'b

  datatype ('a, 'b) dict = Empty
    | Node of ('a, 'b) dict * ('a, 'b) entry * ('a, 'b) dict

  val empty = Empty

  fun lookup cmp d k = fun insert cmp (d, e) =
    end (* structure BST *)
```

The abstract dictionary type is again a tree, but now doubly polymorphic.

(And we wrote it without a separate hidden helper type, but that’s not significant.)
Then we could implement BST much as before:

```ml
structure BST : DICT =
struct
  type 'a key = 'a
  type ('a, 'b) entry = 'a key * 'b

  datatype ('a, 'b) dict = Empty
    | Node of ('a, 'b) dict * ('a, 'b) entry * ('a, 'b) dict

  val empty = Empty

  fun lookup cmp d k =

  fun insert cmp (d, e) =

end (* structure BST *)
```

Implement the empty dictionary as an Empty tree, as before.
Then we could implement BST much as before:

```ml
structure BST : DICT =
struct
  type 'a key = 'a
  type ('a, 'b) entry = 'a key * 'b

  datatype ('a, 'b) dict = Empty
    | Node of ('a, 'b) dict * ('a, 'b) entry * ('a, 'b) dict

  val empty = Empty

  fun lookup cmp d k =

  fun insert cmp (d, e) =

end (* structure BST *)
```

The bodies of `lookup` and `insert` are much as before, but they now use `cmp` in place of `String.compare`. 
Does this do the trick?

Yes and No.

If we are careful to use the same comparison function `cmp` in `insert` as in `lookup`, and do that consistently for all operations with a given dictionary, then everything is fine.
However, it is easy to make a mistake. (A malicious user might do so intentionally.)

For example, perhaps we have created the following tree using Int.compare:

![Tree diagram]

If we now binary search for 1, using cmp below, we won’t find it:

```ml
fun cmp (x,y) = Int.compare (y,x)
```
Let’s take advantage of the type system to ensure that all operations on a given dictionary use the same comparison function.
A type class is a type along with some collection of operations for that type (not necessarily all operations).

Example:

```
signature ORDERED =
  sig
    type t (* parameter *)
    val compare : t * t -> order
  end
```

Signature ORDERED specifies an “ordered type class” to consist of a type $t$ along with a comparison function `compare` for $t$. 
**A type class** is a type along with some collection of operations for that type (not necessarily all operations).

Example:

```plaintext
signature ORDERED = 
sig
  type t (* parameter *)
  val compare : t * t -> order
end
```

Signature **ORDERED** specifies an “ordered type class” to consist of a type \( t \) along with a comparison function `compare` for \( t \).

Comment: The signature does not specify \( t \) concretely, but \( t \) need not be abstract. In a given setting, type \( t \) will be some already existing type, so \( t \) is a “parameter”. The signature is said to be “descriptive” of what we mean by an “ordered type class”. This is in contrast to our signature for dictionaries, which was “prescriptive”, defining a brand new abstract type along with operations for it.
Three structures implementing different ORDEREDs:

```ocaml
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 = string
  val compare = String.compare
end
```
Three structures implementing different ORDEREDs:

```plaintext
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 = string
    val compare = String.compare
  end
```
Three structures implementing different ORDEREDs:

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 = string
  val compare = String.compare
end
Three structures implementing different ORDEREDs:

```ml
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 = string
    val compare = String.compare
  end
```

We may want different comparison functions for a given type. Package each up in its own structure.
Three structures implementing different ORDEREDs:

signature ORDERED =
sig
  type t (* parameter *)
  val compare : t * t -> order
end

structure IntLt : ORDERED =
struct
  type t = intval
  val compare = Int.compare
end

structure IntGt : ORDERED =
struct
  type t = int
  val compare = Int.compare
end

structure StringLt : ORDERED =
struct
  type t = string
  val compare = String.compare
end
Let us now redefine the dictionary signature:

```ml
signature DICT =
  sig
  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
```

Instead of a polymorphic key we have an “ordered” key.
We now implement dictionaries with different keys:

```haskell
structure IntLtDict : DICT =
struct
  structure Key = IntLt
  (* rest of code much as in original BST but now
     using Key.compare instead of String.compare. *)
end

structure IntLtDict : DICT =
struct
  structure Key = IntGt
  (* ... uses Key.compare instead of String.compare ... *)
end

structure StringLtDict : DICT =
struct
  structure Key = StringLt
  (* ... uses Key.compare instead of String.compare ... *)
end
```
We now implement dictionaries with different keys:

structure IntLtDict : DICT =
  struct
    structure Key = IntLt
    (* rest of code much as in original BST but now
       using Key.compare instead of String.compare. *)
  end

structure IntGtDict : DICT =
  struct
    structure Key = IntGt
    (* ... uses Key.compare instead of String.compare ... *)
  end

structure StringLtDict : DICT =
  struct
    structure Key = StringLt
    (* ... uses Key.compare instead of String.compare ... *)
  end
A couple points to consider:

(1) Have we solved the problem of inserting with one comparison function but looking up elements with a different one?

(2) Can we avoid rewriting the same code over and over when implementing dictionaries that use different keys?
(1) Have we solved the problem of inserting with one comparison function but looking up elements with a different one?

For instance, could we accidentally insert into a dictionary using `IntLtDict.insert` but then lookup using `IntGtDict.lookup`?

After all, `IntLtDict.Key.t` and `IntGtDict.Key.t` are both `int`. 
(1) Have we solved the problem of inserting with one comparison function but looking up elements with a different one?

Yes!

The types `IntLtDict.dict` and `IntGtDict.dict` are different. Each `datatype 'a dict = ...` declaration creates a brand new type (Datatype Generativity).

Typechecker will prevent intermingling of dictionaries.
(1) Have we solved the problem of inserting with one comparison function but looking up elements with a different one?

Yes!

The types `IntLtDict.dict` and `IntGtDict.dict` are different.

Each `datatype 'a dict = ...` declaration creates a brand new type (*Datatype Generativity*).

(Printed representation is the same, but types are not.)

Typechecker will prevent intermingling of dictionaries.
Can we avoid rewriting the same code over and over when implementing dictionaries that use different keys?

Yes!

That’s where functors come into the picture.

A functor expects a structure and creates a structure.

Let’s write a functor that expects a structure ascribing to ORDERED and creates a structure ascribing to DICT.
functor TreeDict (K : ORDERED) : DICT =
struct
  structure Key = K
  type 'a entry = Key.t * 'a

  datatype 'a dict = ...

  (* code as before but now using
     Key.t and Key.compare *)
end
functor TreeDict (K : ORDERED) : DICT =

struct

  structure Key = K

  type 'a entry = Key.t * 'a

  datatype 'a dict = ...

(* code as before but now using Key.t and Key.compare *)

end
functor TreeDict (K : ORDERED) : DICT =
struct
  structure Key = K
  type 'a entry = Key.t * 'a

  datatype 'a dict = ...

  (* code as before but now using 
     Key.t and Key.compare       *)
end

And now can define our earlier dictionaries as:

structure IntLtDict = TreeDict(IntLt)
structure IntGtDict = TreeDict(IntGt)
structure StringLtDict = TreeDict(StringLt)
If we want to hide the tree implementation of dictionaries, we could use opaque ascription:

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

However, that also hides the key type in `DICT`. We need that to be known to be the same as the input key type. We therefore use a `where type` clause to expose the key type in `DICT`:

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

- `where type` clauses expose types in a signature. So we could also have defined the following (for instance):

  ```ml
  structure T = TreeDict(IntLt) :> 
  DICT where type Key.t = int
  ```

- Multiple `where type` clauses are permitted in SML/NJ.
Syntactic Sugar

One can pass multiple structures or even value declarations to a functor using a more verbose format. ML will wrap an implicit signature around these arguments. For instance, the following verbose format:

```ml
functor PairOrder (structure Ox : ORDERED
                        structure Oy : ORDERED) : ORDERED
= ... (* code that refers to Ox and Oy *)
```

desugars as:

```ml
functor PairOrder (P : sig
                       structure Ox : ORDERED
                       structure Oy : ORDERED
                     end)
       : ORDERED
= ... (* code that refers to P.Ox and P.Oy *)
```
Syntactic Sugar

One can pass multiple structures or even value declarations to a functor using a more verbose format. ML will wrap an implicit signature around these arguments. For instance, the following verbose format:

```ml
functor PairOrder (structure Ox : ORDERED
                     structure Oy : ORDERED) : ORDERED
= ... (* code that refers to Ox and Oy *)
```

desugars as:

```ml
functor PairOrder (P : sig
                     structure Ox : ORDERED
                     structure Oy : ORDERED
                 end)
     : ORDERED
= ... (* code that refers to P.Ox and P.Oy *)
```

no comma!
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)
      | otherwise => otherwise)
end
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)
        | otherwise => otherwise)
end
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)
            | otherwise => otherwise)
end
Now let’s put the pieces together to create a 2D grid, with integers indexing one coordinate and strings the other:

```plaintext
structure GridOrder = 
  PairOrder (structure Ox = StringLt
              structure Oy = IntLt)
```
Now let’s put the pieces together to create a 2D grid, with integers indexing one coordinate and strings the other:

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

Notice how we pass arguments in the verbose format: As if we were defining a structure that contains Ox and Oy as substructures.
Now let’s put the pieces together to create a 2D grid, with integers indexing one coordinate and strings the other:

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

Create a board structure indexed by the grid coordinates:

structure Board = TreeDict(GridOrder)

Create a board value with something on it:

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

Question: What is the type of `b`?
Now let’s put the pieces together to create a 2D grid, with integers indexing one coordinate and strings the other:

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

Create a board structure indexed by the grid coordinates:

```ml
structure Board = TreeDict(GridOrder)
```

Create a board value with something on it:

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

Question: What is the type of `b`?
Answer: `(int -> int) Board.dict`
That is all.

• Please have a good Wednesday.

• See you Thursday, when we will discuss an approach for maintaining hard-to-satisfy representation invariants in the context of Red Black Trees.