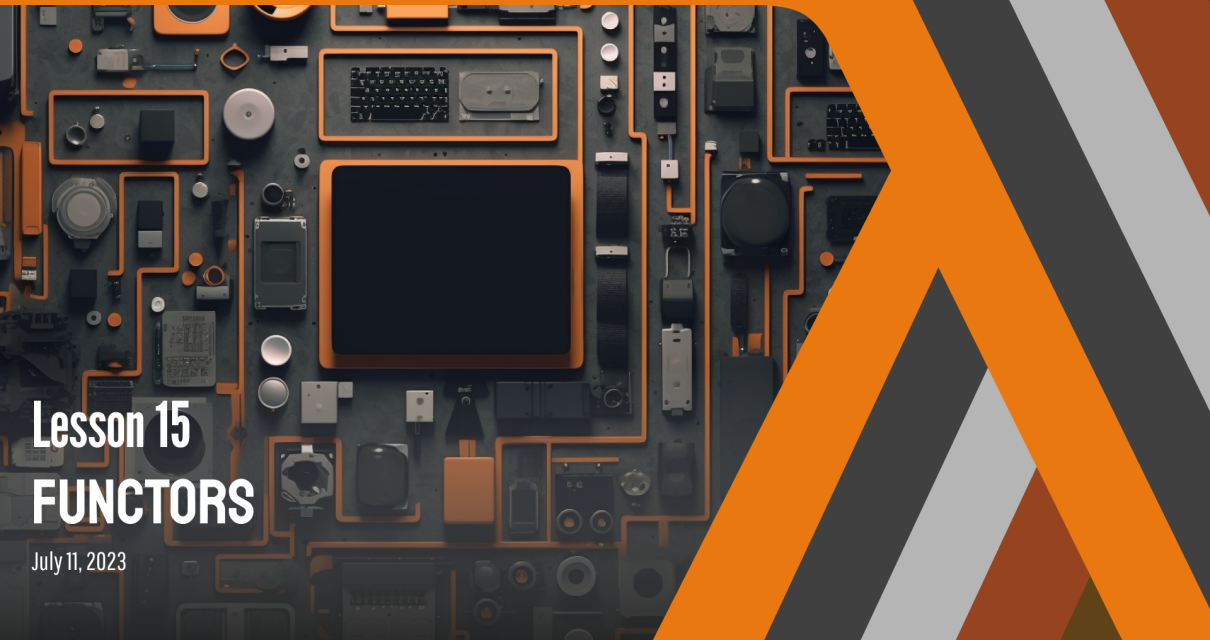


Lesson 15
FUNCTORS

July 11, 2023

1. Dictionaries
2. Polymorphic Dictionaries
3. Type Classes
4. Functors

Last lecture, we introduced the idea of **signatures**, which are first-class ideas of *interfaces*, or the information that we make available about a particular part of some software system.

We saw that we could use **modules**, or **structures**, which are groupings of declarations like values, functions, exceptions, and types. We used **transparent and opaque ascription** to **ascribe** these modules to signatures, which gives us the advantage of limiting the information available to users of a given module.

1 - Dictionaries

I want to play a game.

Let's write a library for dictionaries.

```
signature STR_DICT =
  sig
    type key = string
    type 'a t

    val empty : 'a t
    val insert : key * 'a -> 'a t -> 'a t
    val lookup : key -> 'a t -> 'a option
  end
```

```
structure Dict :> STR_DICT =
  struct
    type key = string
    type 'a t = (key * 'a) tree

    val empty = []
    fun insert (k, v) L = (k, v) :: remove (k, v) L
    fun lookup k [] = NONE
      | lookup k ((k', v')::xs) =
          if k = k' then SOME v'
  end
```

This implementation is pretty simple, but it is reasonably inefficient. Because we implement our dictionaries as lists, we might have to traverse to the end of the list to do a lookup! This gives us a lookup cost of $O(n)$, which is a little pricey.

What's another data structure that we know about for storing data in a more efficiently queryable way?

Binary search trees are one such data structure!

We will use a **comparison function** to store the elements of the tree in an ordered way, so that we only need to go on one side of the tree, when we search for any given element. In this case, since our dictionary has `string` keys, this will be `String.compare`.

This will give us $O(\log n)$ search time.[1]

Let's implement it.

[1] At least, in principle. In actuality, we have no guarantee that such BSTs are relatively balanced, meaning that search time may be linear still. We will leave this problem for another lecture.

```
structure Dict :> STR_DICT =
  struct
    type key = string
    type 'a t = (key * 'a) tree

    val empty = Empty

    (* ... *)
```

Now, we need to implement our `insert` and `lookup` functions. These will take the form of straightforward recursive functions on trees.

Note In this case, we choose to **opaquely ascribe** the `Dict` structure to the `STR_DICT` signature.

```
(* ... *)

  fun insert (k, v) Empty = Node (Empty, (k, v), Empty)
    | insert (k, v) (Node (L, (k', v'), R)) =
        case String.compare (k, k') of
          EQUAL   => Node (L, (k, v), R)
        | LESS    => Node (insert (k, v) L, (k', v'), R)
        | GREATER => Node (L, (k', v'), insert (k, v) R)

  fun lookup (k, v) Empty = NONE
    | lookup (k, v) (Node (L, (k', v'), R)) =
        case String.compare (k, k') of
          EQUAL   => SOME v'
        | LESS    => lookup (k, v) L
        | GREATER => lookup (k, v) R
end
```

Note that we chose to opaquely ascribe to the `DICT` signature.

This is because binary search trees have an important invariant, which is that they are sorted by their comparison function! Just like in the last lecture, we choose to do some **information hiding** so that users cannot see the type of the dictionary, and thus cannot break that invariant.

2 - Polymorphic Dictionaries

This is cool and all, but what if we don't want the keys to be strings?

This comes up all the time, actually. Often, we want an arbitrary map from values of type `t1` to values of type `t2`, where `t1` can be something arbitrarily complicated! It could be lists of strings, it could be a database records for students, it could be a set of names.

What we don't want is to have to write a function of type `t1 -> string` and use that as a preprocessing step every time we use a dictionary.

```
fun getStudentGrade (x : student) (records : int Dict.t) =
  let
    val student_string : string = studentToString x
  in
    Dict.lookup student_string records
  end
```

```
signature POLY_DICT =
  sig
    (* mapping keys of type 'a to values of 'b *)
    type ('a, 'b) t

    val empty : ('a, 'b) t
    val insert : 'a * 'b -> ('a, 'b) t -> ('a, 'b) t
    val lookup : 'a -> ('a, 'b) t -> 'b option
  end
```

A structure ascribing to `POLY_DICT` implements dictionaries of arbitrarily-typed keys. It is **doubly polymorphic**, in that it takes in two type variables, `'a` for the type of its keys, and `'b`, for the type of its values.

```
structure Dict :> POLY_DICT =
  struct
    type ('a, 'b) t = ('a * 'b) tree

    val empty = Empty

    fun insert (k, v) Empty = Node (Empty, (k, v), Empty)
      | insert (k, v) (Node (L, (k', v'), R)) =
          (* what do we do now? *)

    fun lookup (k, v) Empty = NONE
      | lookup (k, v) (Node (L, (k', v'), R)) =
          (* or here... *)
end
```

Uh oh. How do we compare our key, of an arbitrary type `'a`, to another key of an arbitrary type `'a`?[2]

Similarly to how we tried to implement `sort : 'a list -> 'a list`, we need a little bit of a helping hand here, because we don't know what our comparison function is!

[2] It is technically misleading to say "of type `'a`", because I really mean something more like, for some type `t`, two keys of type `t`. But this is close enough.

```
signature POLY_DICT =
  sig
    (* mapping keys of type 'a to values of 'b *)
    type ('a, 'b) t

    val empty : ('a, 'b) t
    val insert :
      ('a * 'a -> order) -> 'a * 'b -> ('a, 'b) t -> ('a, 'b) t
    val lookup :
      ('a * 'a -> order) -> 'a -> ('a, 'b) t -> 'b option
  end
```

This looks better. We use the same familiar of parameterizing our `insert` and `lookup` functions by a comparison function.

```
structure Dict :> STR_DICT =
  struct
    type ('a, 'b) t = ('a * 'b) tree

    val empty = Empty

    (* ... *)
```

This part stays the same.

```
(* ... *)

  fun insert cmp (k, v) Empty = Node (Empty, (k, v), Empty)
    | insert cmp (k, v) (Node (L, (k', v'), R)) =
        case cmp (k, k') of
          EQUAL   => Node (L, (k, v), R)
        | LESS    => Node (insert cmp (k, v) L, (k', v'), R)
        | GREATER => Node (L, (k', v'), insert cmp (k, v) R)

  fun lookup cmp (k, v) Empty = NONE
    | lookup cmp (k, v) (Node (L, (k', v'), R)) =
        case cmp (k, k') of
          EQUAL   => SOME v'
        | LESS    => lookup cmp (k, v) L
        | GREATER => lookup cmp (k, v) R
end
```

Alright, that's polymorphic dictionaries. End of the road. We're done here.

...

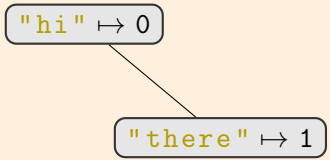
Or *are* we?

Consider the following trace of code.

```
val empty : (string, int) Dict.t = Dict.empty

val T1 = Dict.insert String.compare ("hi", 0) empty
val T2 = Dict.insert String.compare ("there", 1) T1
```

T1



T2

Consider the following comparison function, however:

```
fun to_pig_latin s =
  case String.explode s of
    [] => "ay"
  | c::cs => String.implode (cs @ [c]) ^ "ay"

fun compare_pig_latin (s1, s2) =
  String.compare (to_pig_latin s1, to_pig_latin s2)
```

This comparison function just compares two strings in Pig Latin[3] instead of in regular text.

[3] See this reference for more

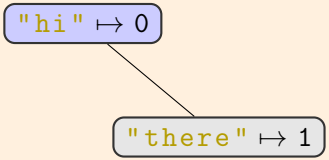
Notably, this is a valid comparison function of type `string * string -> order`!

That means we can use it in conjunction with our `(string, int) Dict.t` trees that we defined earlier.

```
val T3 = Dict.insert compare_pig_latin ("class", 2) T2
```

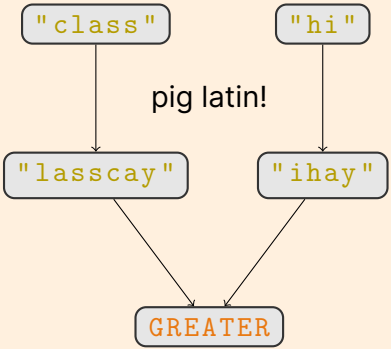
How does this insertion happen?

Inserting: "class" ↦ 2

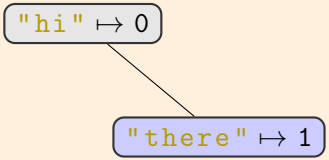


T2

Comparing:

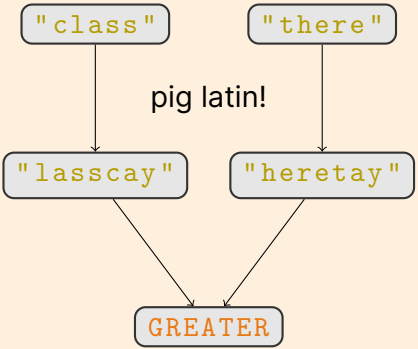


Inserting: "class" ↦ 2

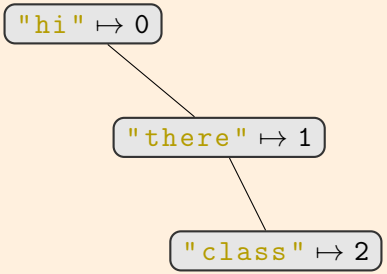


T2

Comparing:



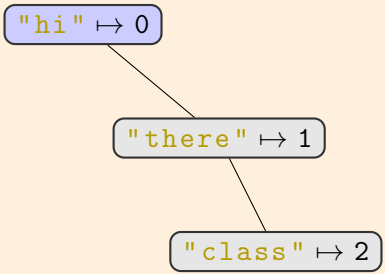
So finally, we end up with this tree:



T3

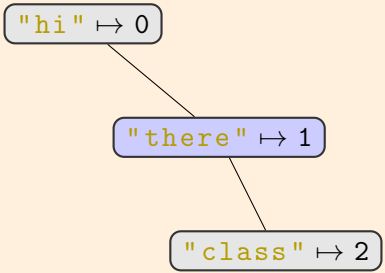
Who sees an issue?

What happens if we try to look up the key `class` using our original `String.compare` function?



T3

Well, `String.compare ("class", "hi")` ≅ `GREATER`, so:



And `String.compare ("class", "there")` ≅ `LESS`.

But, this means we need to go left, and the left node is `Empty`, so we don't find the mapping of `"class"` to `2` at all!

What went wrong?

The problem is that we mixed and matched our comparison functions.

We originally had a tree which was a BST according to `String.compare`, and then tried to insert a value into it as if it was a BST for our `compare_pig_latin` function!

Key Fact There is no *static guarantee* that we give `Dict.lookup` and `Dict.insert` the same comparison functions.

We must merely exercise caution and care to make sure that we don't mix up our comparison functions. This sounds like a precondition – but can we do better?

3 - Type Classes

Signatures can either specify their types as **concrete types**, or leave them as **abstract types**, with definitions which are left to the structures that implement them. For instance, in the signature of `POLY_DICT`, the dictionary type `('a, 'b) t` was left abstract, because the implementation of dictionaries could be anything.

We previously described opaque ascription as having the primary benefit of hiding the definitions of all involved abstract types, which helps in maintaining invariants and abstraction.

Sometimes, we might still want to transparently ascribe to a signature, though. This is most salient with **type classes**.

Def A **type class** is the signature which describes a type, and some operations which may be performed on that type.

We use type classes to implement structures which ascribe to that type class, which witnesses the fact that some type supports that type class's operations.

We call those structures which ascribe to a type class an **instance** of that type class.

For instance, consider the following signature, which is a type class:

```
signature ORD =
  sig
    type t

    val compare : t * t -> order
  end
```

This type class describes all types which admit a comparison function on them. Notably, this is a *signature*, meaning that it is only a description that a module *could* implement, but not an implementation itself!

Let's see some examples of implementations of the `ORD` signature:

```
structure StrOrd : ORD =
  struct
    type t = string
    val compare = String.compare
  end

structure IntOrd : ORD =
  struct
    type t = string
    val compare = Int.compare
  end
```

```
structure PigLatinOrd : ORD =
  struct
    type t = string
    val compare = compare_pig_latin
  end
```

Each of these structures are **instances** of the `ORD` type class, for a particular type!

Note We choose to transparently ascribe these structures, because the point of a typeclass is that you know what the type inside of it is. I mean, it's in the name of the structure.

Note also how there can be multiple type classes for a single type, because there is not necessarily a single "canonical" kind of way to order a type. It depends on your context.

```
signature POLY_DICT =
  sig
    structure Key : ORD

    (* mapping keys of type Key.t to values of 'a *)
    type 'a t

    val empty : 'a t
    val insert : Key.t * 'a -> 'a t -> 'a t
    val lookup : Key.t -> 'a t -> 'a option
  end
```

We make a third attempt at establishing a `POLY_DICT` signature. This time, instead of relying on doubly-parameterizing our dictionary type in the keys and values, we introduce an interior structure `Key`, which ascribes to the signature `ORD`.

In essence, we are specifying that implementations of `POLY_DICT` should come packaged with an implementation of the `ORD` type class. In essence, we are localizing our comparison function to the one provided by `Key`!

Here is one such implementation:

```
structure StrDict :> POLY_DICT =
  struct
    structure Key = StrOrd

    type 'a t = (Key.t * 'a) tree

    val empty = Empty

  (* ... *)
```

```
(* ... *)
  fun insert (k, v) Empty = Node (Empty, (k, v), Empty)
    | insert (k, v) (Node (L, (k', v'), R)) =
        case Key.compare (k, k') of
          EQUAL   => Node (L, (k, v), R)
        | LESS    => Node (insert (k, v) L, (k', v'), R)
        | GREATER => Node (L, (k', v'), insert (k, v) R)

  fun lookup (k, v) Empty = NONE
    | lookup (k, v) (Node (L, (k', v'), R)) =
        case Key.compare (k, k') of
          EQUAL   => SOME v'
        | LESS    => lookup (k, v) L
        | GREATER => lookup (k, v) R
end
```

(We also got rid of the `cmp` functions, but it's hard to highlight the absence of something)

What happens if we try to actually use this implementation, however?

```
structure StrDict : POLY_DICT
- val empty = StrDict.empty;
val empty = - : 'a StrDict.t
- val d1 = StrDict.insert ("hi", 1) empty;
stdIn:2.5-2.40 Error: operator and operand do not agree [tycon mismatch]
  operator domain: ?.StrDict.t * 'Z
  operand:         string * 'Y[INT]
  in expression:
    StrDict.insert ("hi",1)
```

What gives? We defined the inner `Key` structure as `StrOrd`. Shouldn't this type-check?

```
signature POLY_DICT =
  sig
    structure Key : ORD

    (* mapping keys of type Key.t to values of 'a *)
    type 'a t

    val empty : 'a t
    val insert : Key.t * 'a -> 'a t -> 'a t
    val lookup : Key.t -> 'a t -> 'a option
  end
```

Recall that `StrDict` opaquely ascribes to this signature, `POLY_DICT`. This hides the type of `'a t`, as we wanted, but it also hides the type of `Key.t`! There's nothing here which tells us that `Key.t` is `string`.

Fortunately, there is a mechanism in SML that lets us change a signature to one which is slightly more transparent, by explicitly defining some types, and making them no longer opaque.

Def We can modify a signature with *selective transparency* by the syntax `SIG where type <name> = <type>`, which defines that the type named `<name>` is explicitly defined as the input `<type>`.

So, for instance, instead of writing `POLY_DICT`, we can write `POLY_DICT where type Key.t = string` to achieve the same signature, but where `Key.t` is explicitly `string`.

Our new implementation looks very similar to the last, we just change one line:

```
structure StrDict :> POLY_DICT where type Key.t = string =
  struct
    structure Key = StrOrd

    type 'a t = (Key.t * 'a) tree

    val empty = Empty

    (* ... *)

  end
```

That's it!

Now we can use the `StrDict` module, at the correct type.

```
structure StrDict : POLY_DICT?
- val empty = StrDict.empty;
val empty = - : 'a StrDict.t
- val d1 = StrDict.insert ("hi", 1) empty;
val d1 = - : int StrDict.t
```

But wait, all of this implementation was ultimately just specific to the `StrOrd` structure. This amounts to being exactly equivalent to our original dictionary, when we did dictionaries of only strings! What gives?

We haven't really achieved generality, because we would have to write this text again for every single `Key` structure we are interested in using, like `IntOrd`, or `PigLatinOrd`, etc.

If only there was some way we could parameterize our code on other modules.

4 - Functors

We have seen an analogy where we have **structures**, which are groupings of values and other declarations, and which have a kind of *"type"*, or interfaces, namely in the form of **signatures**.

We can think of this as a correspondence between values and structures, which are described by types and signatures, respectively. We also have functions, which act as maps between values to values.

Functors are the module analogue of functions, as maps from structures to structures.

Values	Types	Functions
Structures	Signatures	Functors

The syntax of functors is as follows:

```
functor Name (Arg : SIG) =
  struct
    (* ... *)
  end
```

It looks similar to that of a `structure`, except a functor can take in a module. In this case, this declares a functor named `Name`, which takes in a module which it names `Arg`, so long as the input module ascribes to the signature `SIG`.

This is analogous to a function declaration `fun name (arg : int) = (* ... *)`, which is named `name`, and takes in a value it names `arg`, so long as the input value has type `int`.

One application of functors is in creating a functor which can create instances of the `ORD` type class out of other instances of the `ORD` type class.

For instance, suppose we would like to order arbitrary tuples of two types, `t1` and `t2`. It would be annoying to have to write out, for every permutation of types,

```
structure IntStrOrd : ORD =
  struct
    type t = int * string

    fun compare ((i1, s1), (i2, s2)) =
      case Int.compare (i1, i2) of
        EQUAL   => String.compare (s1, s2)
      | LESS    => LESS
      | GREATER => GREATER
  end
```

```
functor PairOrd (structure A : ORD
                 structure B : ORD) =
  struct
    type t = A.t * B.t

    fun compare ((a1, b1), (a2, b2)) =
      case A.compare (a1, a2) of
        EQUAL   => B.compare (b1, b2)
      | LESS    => LESS
      | GREATER => GREATER
  end
```

This functor takes in two structures, `A` and `B`, both ascribing to `ORD`, and then does the natural left-to-right comparison of a tuple containing both, by leveraging the provided `A.compare` and `B.compare` functions.

```
functor PairOrd (structure A : ORD
                 structure B : ORD) =
  struct
    type t = A.t * B.t

    fun compare ((a1, b1), (a2, b2)) =
      case A.compare (a1, a2) of
        EQUAL   => B.compare (b1, b2)
      | LESS    => LESS
      | GREATER => GREATER
  end
```

But wait, what is going on in the highlighted region?

Sometimes we are interested in a little more versatility in what our functor takes in. We might, for instance, want to take in a type and a value instead of just a module, or more than one structure.

To that end, we can write the following syntax for a functor which takes in a type `t` and a value `x` of type `int`:

```
functor Name (type t
              val x : int) =
  struct
    (* ... *)
  end
```

Note There is no semicolon or comma between the two declarations of `type t` and `val x : int`. There's just space.

It is worth noting that this is just syntactic sugar, though!

Note It is important to realize that a functor **can only take in one structure.**

This is no big issue, though, because structures themselves can contain types and values. The above syntax is really defining a functor `Name`, which takes in an **unnamed structure**, which ascribes to the signature

```
sig
  type t
  val x : int
end
```

We can, however, use the same syntactic sugar when calling the functor.

```
structure Result = Name (type t = string
                         val x = 3)
```

It is important to realize this, because this means that our previous definition of `PairOrd` is really taking in a single structure, which contains two structures `A` and `B`!

So in order to use it, we would write

```
structure IntStrOrd = PairOrd (structure A = IntOrd
                               structure B = StrOrd)
```

which will package both `IntOrd` and `StrOrd` into an unnamed module, which contains a module named `A` and `B`, which are just `IntOrd` and `StrOrd`.

This can mess you up. The important thing to remember: **do not mix** syntactic sugar and not syntactic sugar! If you define a functor with syntactic sugar, you must call it using syntactic sugar.

So now let's implement our polymorphic dictionaries, but now using this functor idea, so we can parameterize over all the possible key type classes.

```
functor MkDict (Key : ORD) :> POLY_DICT =
  struct
    structure Key = Key

    type 'a t = (Key.t * 'a) tree

    val empty = Empty

  (* ... *)
```

```
(* ... *)
  fun insert (k, v) Empty = Node (Empty, (k, v), Empty)
    | insert (k, v) (Node (L, (k', v'), R)) =
        case Key.compare (k, k') of
          EQUAL   => Node (L, (k, v), R)
        | LESS    => Node (insert (k, v) L, (k', v'), R)
        | GREATER => Node (L, (k', v'), insert (k, v) R)

  fun lookup (k, v) Empty = NONE
    | lookup (k, v) (Node (L, (k', v'), R)) =
        case Key.compare (k, k') of
          EQUAL   => SOME v'
        | LESS    => lookup (k, v) L
        | GREATER => lookup (k, v) R
end
```

Now, we can easily define dictionary structures by using our functor `MkDict`, like so:

```
structure IntDict       = MkDict (IntOrd)
structure StrDict       = MkDict (StrOrd)
structure PigLatinDict  = MkDict (PigLatinOrd)
```

Much, much nicer.

Now, we have the advantage that statically, `IntDict.t`, `StrDict.t`, and `PigLatinDict.t` are all recognized as *separate types*.

Even though `StrDict.t` and `PigLatinDict.t` have the same representation on the inside, they cannot be mixed, and they use their own comparison functions independently of each other.

Most critically, we avoid having to write intense amounts of boilerplate, and can (in a type-safe way) reuse our implementation of dictionaries, even though it was predicated on different types.

That's pretty cool!

```
signature POLY_DICT =
  sig
    structure Key : ORD

    (* mapping keys of type Key.t to values of 'a *)
    type 'a t

    val empty : 'a t
    val insert : Key.t * 'a -> 'a t -> 'a t
    val lookup : Key.t -> 'a t -> 'a option
  end
```

```
functor MkDict (Key : ORD) :> POLY_DICT =
  struct
    structure Key = Key

    type 'a t = (Key.t * 'a) tree

    val empty = Empty

    fun insert (k, v) Empty = Node (Empty, (k, v), Empty)
      | insert (k, v) (Node (L, (k', v'), R)) =
          case Key.compare (k, k') of
            EQUAL   => Node (L, (k, v), R)
          | LESS    => Node (insert (k, v) L, (k', v'), R)
          | GREATER => Node (L, (k', v'), insert (k, v) R)

    fun lookup (k, v) Empty = NONE
      | lookup (k, v) (Node (L, (k', v'), R)) =
          case Key.compare (k, k') of
            EQUAL   => SOME v'
          | LESS    => lookup (k, v) L
          | GREATER => lookup (k, v) R
  end
```

Thank you!