Modules II

15-150

Lecture 17: October 30, 2025

Stephanie Balzer
Carnegie Mellon University

Announcement: midterm II

Be on time; next lecture starts at 12:30pm!

When and where:

- Thursday, **November 6**, **11:00am—12:20pm**.
- **PH 100**.

Scope:

- Lectures: 1—15.
- Labs: 1—8 and midterm review section of Lab 10.
- Assignments: up to including Exceptions/Regex.

What you may have on your desk:

- Writing utensils, something to drink/eat, tissues.
- 8.5” x 11” cheatsheet (back and front), handwritten or typeset.
- No cell phones, laptops, or any other smart devices.

Recap

Abstraction through separating specification from implementation:

- Specification: externally visible promise deliver.
- Implementation: internal choice of how to deliver promise.
- Allows us to hide implementation details from the client.
  - Representation independence: the client becomes independent of the choice of internal representation.
  - Any two implementations that satisfy specifications are indistinguishable to the client and thus equal.
- Facilitates modular reasoning (component-wise reasoning).

Recap

SML modules facilitate abstraction:

- Specification: **signature**.
- Implementation: **structure**.

SML modules allow us to control the “flow of information”:

- Structures can **hide** auxiliary, implementation-specific components, not specified by signature.
- **Transparent ascription**: for undefined type specified in signature, **representation type** chosen by structure is **revealed**.
- **Opaque ascription**: for undefined type specified in signature, **representation type** chosen by structure is **hidden**.

Today

- Functors (aka functions on structures).
- Type classes (aka descriptive signatures).
- Deeper exploration of transparent and opaque ascription.

Correspondence:

specification	signature	type
implementation	structure	value
mapping	functor	function

loosely

- Let's resume our dictionary example!

Example: dictionary

A dictionary is a collection of pairs of the form

```
(key, value)
```

where all keys must be unique within a dictionary.

Let's use strings as keys for now

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


end
```

Permit value type to be polymorphic

Example: dictionary

A dictionary is a collection of pairs of the form

```
(key, value)
```

where all keys must be unique within a dictionary.

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



end
```

Example: dictionary

A dictionary is a collection of pairs of the form

```
(key, value)
```

where all keys must be unique within a dictionary.

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

Replace entry, if key already exists

Search tree representation of dictionary

Implementation: **represent** dictionary as a binary search tree, where

`(key, value)`

are stored in nodes.

- Representation **invariant**:
  - Tree must be sorted.
  - Keys must be unique.
- All functions declared by structure
  - may **assume** that received tree is sorted,
  - and must **assert** that returned tree is sorted.
  - (Similarly for key uniqueness.)

Search tree representation of dictionary

```
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
```

Explore :>

```
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
```

Transparent ascription can be useful for debugging purposes.

```
end
```

Search tree representation of dictionary

```
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
```

```
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


end
```

forced by signature

Search tree representation of dictionary

```
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
```

```
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
end
```

Because datatype is not declared in signature, constructors (and thus pattern matching) are not available outside signature.

But bindings externally visible due to transparent ascription.

Search tree representation of dictionary

```
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
```

```
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



end
```

Search tree representation of dictionary

```
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
```

```
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 insert ...
  fun lookup ...
end
```

explore next!

Search tree representation of dictionary

```
(* ins : '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

Search tree representation of dictionary

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

Search tree representation of dictionary

```
(* ins : 'a dict * 'a entry -> 'a dict *)
fun insert (Empty, e) = Node(Empty, e, Empty)
  | insert (Node(lt, e’ as (k’,_), rt),
            e as (k,_)) =
      (case String.compare(k,k’) of
```

Search tree representation of dictionary

```
(* ins : 'a dict * 'a entry -> 'a dict *)
fun insert (Empty, e) = Node(Empty, e, Empty)
  | insert (Node(lt, e’ as (k’,_), rt),
            e as (k,_)) =
      (case String.compare(k,k’) of
        EQUAL => Node(lt, e, rt)
```

Replace existing entry with new one

Search tree representation of dictionary

```
(* ins : 'a dict * 'a entry -> 'a dict *)
fun insert (Empty, e) = Node(Empty, e, Empty)
  | insert (Node(lt, e’ as (k’,_), rt),
             e as (k,_)) =
      (case String.compare(k,k’) of
        EQUAL => Node(lt, e, rt)
```

Search tree representation of dictionary

```
(* ins : 'a dict * 'a entry -> 'a dict *)
fun insert (Empty, e) = Node(Empty, e, Empty)
  | insert (Node(lt, e’ as (k’,_), rt),
             e as (k,_)) =
      (case String.compare(k,k’) of
        EQUAL => Node(lt, e, rt)
        | LESS => Node(insert(lt,e), e’, rt)
```

Search tree representation of dictionary

```
(* ins : 'a dict * 'a entry -> 'a dict *)
fun insert (Empty, e) = Node(Empty, e, Empty)
  | insert (Node(lt, e’ as (k’,_), rt),
            e as (k,_)) =
      (case String.compare(k,k’) of
        EQUAL => Node(lt, e, rt)
        | LESS => Node(insert(lt,e), e’, rt)
        | GREATER => Node(lt, e’, insert(rt,e)))
```

Search tree representation of dictionary

```
(* lookup : 'a dict -> key -> 'a option *)
fun lookup tree key =
 let
    fun lk (Empty) = NONE
    | lk (Node(left, (k,v), right)) =
      (case String.compare(key,k) of
        EQUAL => SOME(v)
        | LESS => lk left
        | GREATER => lk right)
  in
    lk tree
  end
```

Search tree representation of dictionary

Let's interact with **BST**:

```
val d = BST.insert(BST.insert(BST.insert(
          BST.empty,(“a",1)),("b",2)),("c",3))
```

What is the type of **d**?

```
int BST.dict
```

The binding for **d** will be revealed because of opaque ascription. However, because the tree datatype is not declared in the signature, a client cannot pattern match on its constructors.

Now consider: `val look = BST.lookup d`

What is the type of **look**?

```
BST.key -> int option
```

Search tree representation of dictionary

Let's interact with **BST**:

```
val d = BST.insert(BST.insert(BST.insert(
          BST.empty,(“a",1)),("b",2)),("c",3))
```

What is the type of **d**?

```
int BST.dict
```

Now consider: `val look = BST.lookup d`

What is the type of **look**?

```
BST.key -> int option
```

Now consider:

```
val x = look “e”
val y = look “a”
```

Bindings: `[NONE/x, (SOME 1)/y]`

Let's reconsider our DICT signature

```
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 :
  val insert :
end
```

What if we needed keys other than strings?

We could try to make key polymorphic too.

Let's reconsider our DICT signature

```
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 if we needed keys other than strings?
- We could try to make key polymorphic too.

Let's reconsider our DICT signature

```
signature DICT =
sig
  type 'a key = 'a                       (* concrete type *)
  type ('a,'b) entry = 'a key * 'b (* concrete type *)
  type ('a,'b) dict                         ype *)
  val empty : ('a,'b) dict
  val lookup :
  val insert :
end
```

How to implement now?

Used `String.compare`

- What if we needed keys other than strings?
- We could try to make key polymorphic too.
- Keys should become comparable!

Let's reconsider our DICT signature

```
lookup:

insert:
```

Keys should become comparable!

Let’s reconsider our DICT signature

```
lookup: ('a *'a -> order) -> ('a,'b) dict -> ‘a ->'b option
insert: ('a *'a -> order) -> (('a,'b) dict * ('a,'b) entry)
                          -> ('a,'b) dict
```

- Keys should become comparable!
- Require a comparison function as an argument.
- Restricts polymorphism of keys!

Let's update our BST structure accordingly

As specified by signature

```
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 insert cmp d k =

  fun lookup cmp (d, k) =
end
```

Let’s update our BST structure accordingly

```
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 insert cmp d k =

  fun lookup cmp (d, k) =
end
```

Again, binary search tree as representation type.

This time, with polymorphic key.

Let’s update our BST structure accordingly

```
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 insert cmp d k =

  fun lookup cmp (d, k) =
end
```

As before.

Let's update our BST structure accordingly

```
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 insert cmp d k =

  fun lookup cmp (d, k) =
end
```

Bodies of `insert` and `lookup` now use `cmp` instead of `String.compare`.

Let’s update our BST structure accordingly

```
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 insert cmp d k =

  fun lookup cmp (d, k) =
end
```

Does this do the trick?

Let’s update our BST structure accordingly

```
fun insert cmp d k =
fun lookup cmp (d, k) =
```

Does this do the trick?

Let's update our BST structure accordingly

```
fun insert cmp d k =
fun lookup cmp (d, k) =
```

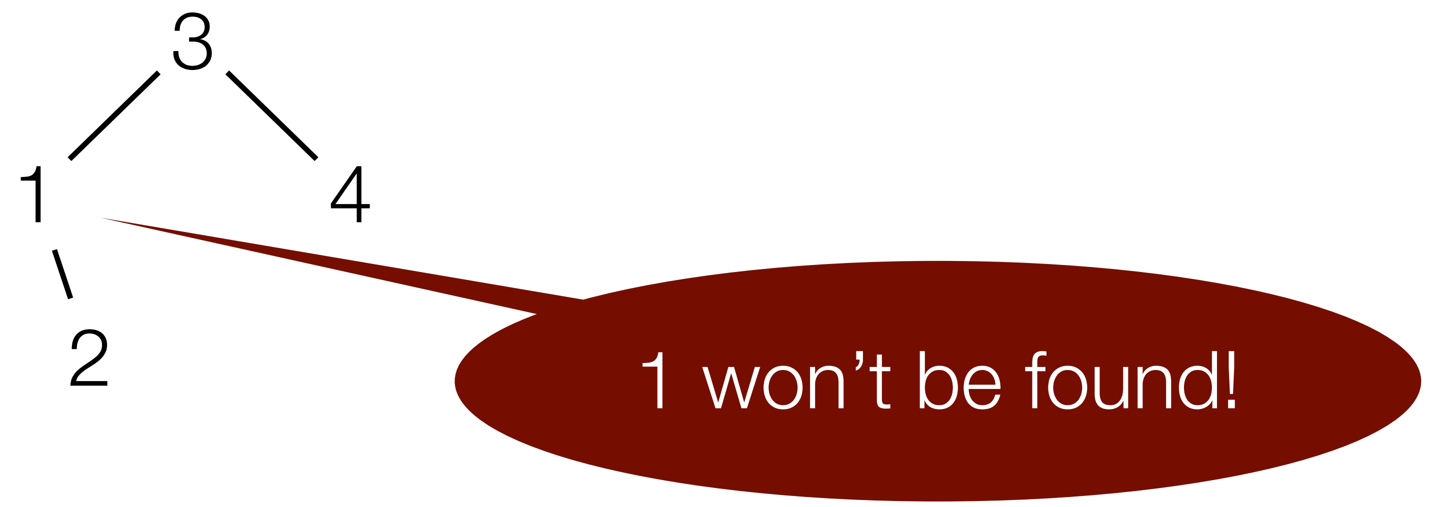
Does this do the trick? Well, not quite.

What if a client provides different `cmp` functions to `insert` than to `lookup`, for example?

Let's update our BST structure accordingly

- Does this do the trick? Well, not quite.
- What if a client provides different `cmp` functions to `insert` than to `lookup`, for example?

For example, a client creates the following tree using `insert` and `Int.compare`:



For `lookup` of `1`, the client now uses:

```
fun cmp (x,y) = Int.compare (y,x)
```

Let’s update our BST structure accordingly

Does this do the trick? Well, not quite.

What if a client provides different `cmp` functions to `insert` than to `lookup`, for example?

Can we enforce the invariant, that all operations use the same comparison function by typing?

Yes, but we need type classes for this!

Type classes

Type class

- A signature specifying a type and associated operations.
  - No expectation that specification is exhaustive.

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`**.

Type classes

Example:

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

- Even though `t` is not concrete, it is not abstract.
- We expect `t` to be some already existing type, hence use the comment **parameter**.
- Signature `ORDERED` is said to be **descriptive**.
- Signature `DICT` is in contrast **prescriptive**, defining an abstract type with all its operations, exhaustively.

Type classes

- Even though `t` is not concrete, it does not have to be abstract.
- We expect `t` to be some already existing type, hence use the comment **parameter**.
- Signature `ORDERED` is said to be **descriptive**.
- Signature `DICT` is in contrast **prescriptive**, defining an abstract type with all its operations, exhaustively.
- We tend to use transparent ascription for descriptive signatures (aka type classes), and opaque ascription for prescriptive signatures.

Perspective of types in signatures

Concrete:

Signature dictates representation type, which is thus visible to client.

Abstract:

Signature hides representation type. Client code must work regardless of the representation type chosen by structure.

Parameter:

Client supplies the type, implementation must work with whatever the clients supplies.

Different ways of implementing ORDERED

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

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

Different ways of implementing ORDERED

```
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
```

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

Different ways of implementing ORDERED

```
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
```

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

Redefine DICT using type class ORDERED

Use type class as a parameter

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

Redefine DICT using type class ORDERED

```
signature DICT =
sig
  structure Key : ORDERED                 (* parameter *)
  type 'a entry = key * '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
```

Any structure implementing `DICT` will comprise a sub-structure implementing `ORDERED`.

Redefine DICT using type class ORDERED

Use type class' type `t`

```
signature DICT =
sig
  structure Key : ORDERED              (* parameter *)
  type 'a entry = key * '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
```

Any structure implementing `DICT` will comprise a sub-structure implementing `ORDERED`.

Redefine DICT using type class ORDERED

```
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 -> 'a option
  val insert : 'a dict * 'a entry -> 'a dict
end
```

Any structure implementing `DICT` will comprise a sub-structure implementing `ORDERED`.

Redefine DICT using type class ORDERED

Use type class' type `t`

```
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 -> 'a option
  val insert : 'a dict * 'a entry -> 'a dict
end
```

Any structure implementing `DICT` will comprise a sub-structure implementing `ORDERED`.

Redefine DICT using type class ORDERED

```
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
```

Any structure implementing `DICT` will comprise a sub-structure implementing `ORDERED`.

Let’s define dictionaries with different keys!

Using our structures defined earlier implementing type class ORDERED, we can define dictionary structures with different keys:

```
structure IntLtDict : DICT =
struct




end
```

Let’s define dictionaries with different keys!

Using our structures defined earlier implementing type class ORDERED, we can define dictionary structures with different keys:

```
structure IntLtDict : DICT =
struct
  structure Key = IntLt


end
```

Let's define dictionaries with different keys!

Using our structures defined earlier implementing type class ORDERED, we can define dictionary structures with different keys:

```
structure IntLtDict : DICT =
struct
  structure Key = IntLt
  (* code as before but now using Key.t instead of key
     and Key.compare instead of String.compare *)
end

structure IntGtDict : DICT =
struct




end
```

Let's define dictionaries with different keys!

Using our structures defined earlier implementing type class ORDERED, we can define dictionary structures with different keys:

```
structure IntLtDict : DICT =
struct
  structure Key = IntLt
  (* code as before but now using Key.t instead of key
     and Key.compare instead of String.compare *)
end

structure IntGtDict : DICT =
struct
  structure Key = IntGt


end
```

Let’s define dictionaries with different keys!

Using our structures defined earlier implementing type class ORDERED, we can define dictionary structures with different keys:

```
structure IntLtDict : DICT =
struct
  structure Key = IntLt
  (* code as before but now using Key.t instead of key
     and Key.compare instead of String.compare *)
end

structure IntGtDict : DICT =
struct
  structure Key = IntGt
  (* code as before but now using Key.t instead of key
     and Key.compare instead of String.compare *)
end
```

Let’s define dictionaries with different keys!

```
structure IntLtDict : DICT =
struct
  structure Key = IntLt
  (* code as before but now using Key.t instead of key
     and Key.compare instead of String.compare *)
end

structure IntGtDict : DICT =
struct
  structure Key = IntGt
  (* code as before but now using Key.t instead of key
     and Key.compare instead of String.compare *)
end

structure StringLtDict : DICT =
struct
  structure Key = StringLt
  (* code as before but now using Key.t instead of key
     and Key.compare instead of String.compare *)
end
```

Only differ in **Key**!

Is that it?

- Have we solved the problem of inserting with one comparison function but looking up elements with a different one?
- Can we avoid rewriting (copying & pasting) the same code over and over when implementing dictionaries with different keys?

Is that it?

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

For example, 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`.

No, this is not possible! `IntGtDict.dict` and `IntLtDict.dict` are different types.

ML type checker will thus prevent intermingling of dictionaries.

Is that it?

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

YES!

Can we avoid rewriting (copying & pasting) the same code over and over when implementing dictionaries with different keys?

YES, but we need to use a functor for this!

A **functor** creates a structure, given a structure as an argument.

Let's write a functor that creates a structure ascribing to `DICT`, given a structure ascribing to `ORDERED` as an argument.

Avoid the bloat with a functor!

```
functor TreeDict (K : ORDERED) : DICT =
struct




end
```

Argument structure, of type `ORDERED`

Note: ":" denotes typing, not ascription mode.

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

Avoid the bloat with a functor!

```
functor TreeDict (K : ORDERED) : DICT =
struct



end
```

Denotes ascription mode, as usual.

Structured returned, transparently ascribing to **DICT**

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

```
signature DICT =
sig
  structure Key = ORDERED               (* parameter *)
  type 'a entry = Key.t * 'a            (* concrete  *)
  type 'a dict                          (* abstract  *)
  ...
end
```

Avoid the bloat with a functor!

```
functor TreeDict (K : ORDERED) : DICT =
struct




end
```

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

```
signature DICT =
sig
  structure Key = ORDERED           (* parameter *)
  type 'a entry = Key.t * 'a        (* concrete  *)
  type 'a dict                      (* abstract  *)
  ...
end
```

Avoid the bloat with a functor!

```
functor TreeDict (K : ORDERED) : DICT =
struct
  structure Key = K




end
```

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

```
signature DICT =
sig
  structure Key = ORDERED                (* parameter *)
  type 'a entry = Key.t * 'a             (* concrete  *)
  type 'a dict                           (* abstract  *)
  ...
end
```

Avoid the bloat with a functor!

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


end
```

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

```
signature DICT =
sig
  structure Key = ORDERED               (* parameter *)
  type 'a entry = Key.t * 'a            (* concrete  *)
  type 'a dict                          (* abstract  *)
  ...
end
```

Avoid the bloat with a functor!

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

end
```

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

```
signature DICT =
sig
  structure Key = ORDERED                (* parameter *)
  type 'a entry = Key.t * 'a             (* concrete  *)
  type 'a dict                           (* abstract  *)
  ...
end
```

Avoid the bloat with a functor!

```
functor TreeDict (K : ORDERED) : DICT =
struct
  structure Key = K
  type 'a entry = Key.t * 'a
  datatype 'a dict = ...
  (* code as before, but using Key.t and Key.compare *)
end
```

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

```
signature DICT =
sig
  structure Key = ORDERED            (* parameter *)
  type 'a entry = Key.t * 'a         (* concrete  *)
  type 'a dict                       (* abstract  *)
  ...
end
```

Avoid the bloat with a functor!

```
functor TreeDict (K : ORDERED) : DICT =
struct
  structure Key = K
  type 'a entry = Key.t * 'a
  datatype 'a dict = ...
  (* code as before, but using Key.t and Key.compare *)
end
```

Now, we can define our earlier dictionaries as:

```
structure IntLtDict = TreeDict(IntLt)
structure IntGtDict = TreeDict(IntGt)
structure StringLtDict = TreeDict(StringLt)
```

Avoid the bloat with a functor!

```
functor TreeDict (K : ORDERED) : DICT =
struct
  structure Key = K
  type 'a entry = Key.t * 'a
  datatype 'a dict = ...
  (* code as before, but using Key.t and Key.compare *)
end
```

Let's use opaque ascription instead!

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

```
signature DICT =
sig
  structure Key = ORDERED                    (* parameter *)
  type 'a entry = Key.t * 'a                 (* concrete  *)
  type 'a dict                               (* abstract  *)
  ...
end
```

Avoid the bloat with a functor!

```
functor TreeDict (K : ORDERED) :> DICT =
struct
  structure Key = K
  type 'a entry = Key.t * 'a
  datatype 'a dict = ...
  (* code as before, but using Key.t and Key.compare *)
end
```

But now we hide the representation type for **`Key.t`**.

Instead, reveal that **`Key.t`** is the same as input key **`K.t`**!

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

```
signature DICT =
sig
  structure Key = ORDERED             (* parameter *)
  type 'a entry = Key.t * 'a          (* concrete  *)
  type 'a dict                        (* abstract  *)
  ...
end
```

Avoid the bloat with a functor!

```
functor TreeDict (K : ORDERED) :> DICT
                         where type Key.t = K.t =
struct
  structure Key = K
  type 'a entry = Key.t * 'a
  datatype 'a dict = ...
  (* code as before, but using Key.t and Key.compare *)
end
```

- But now we hide the representation type for `Key.t`.
- Instead, reveal that `Key.t` is the same as input key `K.t`!
- To rectify this, we need to add a `where` clause.

Summary

- **Prescriptive** signatures exhaustively specify a type's operations, typically using **opaque** ascription.
- **Descriptive** signatures (aka type classes) expose a type parameter's operations, typically using **transparent** ascription.
- A **functor** creates a structure, given a structure as an argument.
  - Functor arguments are typically type classes to prevent code redundancy.

A word on syntax:

Summary

A word on syntax:

- Functors only take a single structure as an argument.
- Multiple argument structures can be passed using nested structures or using specialized syntax.
  - More on this in labs and homework.
- Similarly, multiple where clauses are supported, using different syntactic forms.
  - More on this in labs and homework.

That's all for today. Happy 🎃👻🎃!