Signatures & Structures

A signature specifies an interface.

A structure provides an implementation.

Example:

A queue is a first-in first-out datastructure.

We can describe a queue abstractly by specifying a (new) queue type, along with operations on that type.

That’s a signature.
Then we implement it in a structure.
Queue Signature

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

  (* will raise Empty if called on empty q *)

  val deq : 'a q -> 'a * 'a q

end
First QUEUE implementation

Use a single list.

Need to say how the list represents the abstract queue:

(called “abstraction function”)

The list represents the queue elements in arrival order.
First QUEUE implementation

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
  val deq : 'a q -> 'a * 'a q
end

structure Queue : QUEUE =
struct

Pronounced “ascribes” or “ascribes to” or “ascribes transparently”.
It means: The structure provides all the items specified in the signature. (The structure may contain additional items, e.g., helper functions, but those will not be visible outside the structure.)

end
First QUEUE implementation

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
    val deq : 'a q -> 'a * 'a q
  end

structure Queue : QUEUE =
  struct
    type 'a q = 'a list
    val empty = []
    fun enq (q, x) = q @ [x]
    val null = List.null
    exception Empty
    fun deq [] = raise Empty
      | deq (x::q) = (x, q)
  end
val q2 = Queue.enq(Queue.enq(Queue.empty,1),2)

Q: What is the type of \texttt{q2}?

(\textit{ignore that you know it is int list})
val q2 = Queue.enq(Queue.enq(Queue.empty,1),2)

Q: What is the type of \texttt{q2}?
A: \texttt{int Queue.q}

Why? Because:

First, the signature specifies that queues have type \texttt{\textquoteleft a q}, with \texttt{'a} representing the value type. That is \texttt{int} here.

Second, we have implemented queues using a structure called \texttt{Queue}. The type is defined inside the structure, so the type has the qualified name \texttt{\textquoteleft a Queue.q}, here with \texttt{'a} instantiated to \texttt{int}. 
Interacting with the Queue

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

Q: What is the type of q2?
A: int Queue.q

Also:
ML will print the list \([1,2]\). We can see the list because of transparent ascription (more on how to hide that later).

Next, consider:

val (a, b) = Queue.deq q2
val (c, _) = Queue.deq q2
val (d, _) = Queue.deq b

Q: What are the bindings for a, c, d?
A: [1/a, 1/c, 2/d]

(We also have the binding \([2/b]\).)
Second QUEUE implementation

Use a pair of lists:

\[(\text{front}, \text{back})\].

Abstraction Function:

\[\text{front @ (rev back)}\]

represents the queue elements in arrival order.
Second QUEUE implementation

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
  val deq : 'a q -> 'a * 'a q
end

structure Q :> QUEUE =
struct

  “opaque ascription”

This means the representation details are hidden from any user external to the structure. Only items specified by the signature are visible.

With transparent ascription, a user can see and sometimes mess with a representation (earlier, ML would print out lists for queues).

With opaque ascription, ML will only print a dash. An external user cannot see or mess with the internal representation.
Second QUEUE implementation

```ml
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
    val deq : 'a q -> 'a * 'a q
  end
```

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

Satisfies requirement that  \( f @ (\text{rev}(x::b)) \) constitute the queue elements in arrival order.
Second QUEUE implementation

```ml
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
    val deq : 'a q -> 'a * 'a q
  end

structure Q :> QUEUE =
  struct
    type 'a q = 'a list * 'a list
    val empty = ([],[])
    fun enq ((f,b), x) = (f, x::b)
    fun null ([],[]) = true
        | null _ = false
    exception Empty
    fun deq ([],[]) = raise Empty
        | deq ([], b) = deq (rev b, [])
        | deq (x::f, b) = (x, (f, b))
  end
```
The Two Implementations

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

structure Q :> QUEUE =
struct
  type 'a q  = 'a list * 'a list
  val empty = ([],[])
  fun enq ((f,b), x) = (f, x:::b)
  fun null ([],[]) = true
    | null _      = false
  exception Empty
  fun deq ([],[]) = raise Empty
    | deq ([], b) = deq (rev b, [])
    | deq (x:::f, b) = (x, (f, b))
end
Dictionary Signature

A dictionary is a collection of pairs of the form \((key, value)\).

We require all the keys to be unique in a given dictionary.

\[
\text{signature DICT = } \\
\text{sig}
\]

end
Dictionary Signature

A dictionary is a collection of pairs of the form \((key, value)\).

We require all the keys to be unique in a given dictionary.

```
signature DICT =  (for the time being, we’ll fix the key type)
sig
  type key = string  (* concrete *)
end
```
Dictionary Signature

A dictionary is a collection of pairs of the form \((\text{key}, \text{value})\).

We require all the keys to be unique in a given dictionary.

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

(for the time being, we’ll fix the key type) we’ll allow the value type to be polymorphic
Dictionary Signature

A dictionary is a collection of pairs of the form \((\text{key}, \text{value})\).

We require all the keys to be unique in a given dictionary.

```plaintext
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

(replace entry if key already appears in the dictionary)
```
Dictionary Implementation

We will use a tree implementation.

Abstraction Function: The \((\text{key}, \text{value})\) items in the tree constitute the dictionary.

We further impose a Representation Invariant:

The tree must be sorted on \text{key}.

This means:

All functions within the structure may \textit{assume} that any trees they receive are sorted

\textit{and}

\textit{must ensure} that any trees returned are sorted.
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

structure BST : DICT =
struct
  type key = string
  type 'a entry = key * 'a

  datatype 'a tree =
    Empty
  | Node of 'a tree * 'a entry * 'a tree

Observe: Because the datatype is not declared in the signature, a user external to the structure cannot pattern match on or otherwise use the constructors.

They will be visible because we will declare type 'a dict = 'a tree and because we are using transparent ascription.

So, a user can see the internals of our representation, but cannot mess with them.
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

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
BST Implementation of Dictionaries

(* insert : 'a dict * 'a entry -> 'a dict *)
BST Implementation of Dictionaries

(* insert : 'a dict * 'a entry -> 'a dict *)

fun insert (Empty, e) = Node(Empty, e, Empty)
| insert (Node(lt, e' as (k',_), rt),...) =

Layered Pattern Matching

Here, this creates bindings
of the full (key, value) entry to e',
of just the key part to k', and
the wildcard _ matches the value part,
without producing a binding.
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 entry on same key
BST Implementation of Dictionaries

(* 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, _)) =
      (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)))
fun lookup tree key =
  let
    fun lk (Empty) = NONE
    | lk (Node(lt, (k,v), r)) =
      (case String.compare(key,k) of
        EQUAL => SOME(v)
        | LESS => lk left
        | GREATER => lk right)
  in
    lk tree
  end