Modules
and Abstract Data Types

COS 326
David Walker
Princeton University
Last Time: Modules

• Before the break, we introduced you to ML's module system:
  – *signatures*: interfaces that mention the names of abstract types and the names/types of operations over them
  – *structures*: implementations of abstract data types (they give concrete definitions to abstract types and code to implement the abstract operations)
  – *functors*: functions from modules to modules (they provide a way to allow us to parameterize our modules)
module type UNSIGNED_BIGNUM =

sig
  type ubignum
  val fromInt : int -> ubignum
  val toInt : ubignum -> int
  val plus : ubignum -> ubignum -> ubignum
  val minus : ubignum -> ubignum -> ubignum
  val times : ubignum -> ubignum -> ubignum
...
end
module My_UBignum_1000 : UNSIGNED_BIGNUM =
struct
    let base = 1000

    type ubignum = int list

    let toInt (b:ubignum):int = ...

    let plus (b1:ubignum) (b2:ubignum):ubignum = ...

    let minus (b1:ubignum) (b2:ubignum):ubignum = ...

    let times (b1:ubignum) (b2:ubignum):ubignum = ...

end
module BIGNUM_UNARY : UNSIGNED_BIGNUM =
struct
  type ubignum = Zero | Succ of ubignum

  let rec utoInt (u:ubignum) : int =
    match u with
    | Zero -> 0
    | Succ u' -> 1 + (toInt u')

  let rec plus (u1:ubignum) (u2:ubignum) : ubignum =
    match u1 with
    | Zero -> u2
    | Succ u1' -> uplus u1' (Succ u2)

end
The Abstraction Barrier

*Rule of thumb*: try to use the language mechanisms (e.g., modules, interfaces, etc.) to *enforce* the abstraction barrier.

- reveal as little information about *how* something is implemented as you can.
- provides maximum flexibility for change moving forward.
- pays off down the line

However, like all design rules, we must be able to recognize when the barrier is causing more trouble than it’s worth and abandon it.

- may want to reveal more information for debugging purposes
  - eg: conversion to string so you can print things out
module type QUEUE =

  sig
    type 'a queue
    val empty : unit -> 'a queue
    val enqueue : 'a -> 'a queue -> 'a queue
    val is_empty : 'a queue -> bool
    exception EmptyQueue
    val dequeue : 'a queue -> 'a queue
    val front : 'a queue -> 'a
  end
Another Example: Queues or Fifo’s

These queues are re-usable for different element types.

Here's an exception that client code might want to catch.
module AppendListQueue : QUEUE =

struct
    type 'a queue = 'a list
    let empty() = []
    let enqueue(x:'a)(q:'a queue) : 'a queue = q @ [x]
    let is_empty(q:'a queue) =
        match q with
        | [] -> true
        | _:::_ -> false

...

end
module AppendListQueue : QUEUE =

  struct
    type 'a queue = 'a list
    let empty() = []
    let enqueue(x:'a)(q:'a queue) : 'a queue = q @ [x]
    let is_empty(q:'a queue) = ...

  exception EmptyQueue

  let deq(q:'a queue) : ('a * 'a queue) =
      match q with
        | [] -> raise EmptyQueue
        | h::t -> (h,t)

  let dequeue(q:'a queue) : 'a queue = snd (deq q)
  let front(q:'a queue) : 'a = fst (deq q)

end
module AppendListQueue : QUEUE =

struct
    type 'a queue = 'a list
    let empty() = []
    let enqueue(x:'a)(q:'a queue) : 'a queue = q @ [x]
    let is_empty(q:'a queue) = ...

exception EmptyQueue

let deq(q:'a queue) : ('a * 'a queue) =
    match q with
    | [] -> raise EmptyQueue
    | h::t -> (h,t)

let dequeue(q:'a queue) : 'a queue = snd (deq q)
let front(q:'a queue) : 'a = fst (deq q)
end

Notice deq is a helper function that doesn’t show up in the signature.

You can't use it outside the module.
module AppendListQueue : QUEUE =

struct

  type 'a queue = 'a list
  let empty() = []
  let enqueue(x:'a)(q:'a queue) : 'a queue = q @ [x]
  let is_empty(q:'a queue) = ...

exception EmptyQueue

  let deq(q:'a queue) : ('a * 'a queue) =
    match q with
    | [] -> raise EmptyQueue
    | h::t -> (h,t)

  let dequeue(q:'a queue) : 'a queue = snd (deq q)
  let front(q:'a queue) : 'a = fst (deq q)
end

Notice enqueue takes time proportional to the length of the queue

Dequeue runs in constant time.
module DoubleListQueue : QUEUE =

  struct
    type 'a queue = {front:'a list; rear:'a list}

  ...

end
let q0 = empty;;  \{front=\[]; rear=\[]\}

let q1 = enqueue 3 q0;;  \{front=\[]; rear=\[3\]\}

let q2 = enqueue 4 q1 ;;  \{front=\[]; rear=\[4;3\]\}

let q3 = enqueue 5 q2 ;;  \{front=\[]; rear=\[5;4;3\]\}

let q4 = dequeue q3 ;;  \{front=\[4;5\]; rear=\[]\}

let q5 = dequeue q4 ;;  \{front=\[5\]; rear=\[]\}

let q6 = enqueue 6 q5 ;;  \{front=\[5\]; rear=\[6\]\}

let q7 = enqueue 7 q6 ;;  \{front=\[5\]; rear=\[7;6\]\}
module DoubleListQueue : QUEUE =

  struct
      type 'a queue = {front:'a list; rear:'a list}

      let empty() = {front=[]; rear=[]}

      let enqueue x q = {front=q.front; rear=x::q.rear}

      let is_empty q =
          match q.front, q.rear with
          | [], []       -> true
          | _, _          -> false

  end
module DoubleListQueue : QUEUE =

  struct
      type 'a queue = {front:'a list; rear:'a list}
      ...
  exception EmptyQueue

  let deq (q:'a queue) : 'a * 'a queue =
      match q.front with
      | h::t -> (h, {front=t; rear=q.rear})
      | [] -> match List.rev q.rear with
              | h::t -> (h, {front=t; rear=[]})
              | [] -> raise EmptyQueue

  let dequeue (q:'a queue) : 'a queue = snd(deq q)
  let front (q:'a queue) : 'a = fst(deq q)
end
How would we design an abstraction?

• Write some test cases:
  – what operations might you want?
  – what *abstract* types might you want?

• From this, we can derive a signature
  – list the types
  – list the operations with their types
  – don’t forget to provide enough operations that you can debug!

• Then we can build an implementation
  – when prototyping, build the simplest thing you can.
  – later, we can swap in a more efficient implementation.
  – (assuming we respect the abstraction barrier.)
Common Interfaces

- The stack and queue interfaces are quite similar:

```ocaml
module type STACK =
  sig
    type 'a stack
    val empty : unit -> 'a stack
    val push : int -> 'a stack -> 'a stack
    val is_empty : 'a stack -> bool
    exception EmptyStack
    val pop : 'a stack
    val top : 'a stack
  end

module type QUEUE =
  sig
    type 'a queue
    val empty : unit -> 'a queue
    val enqueue : 'a -> 'a queue -> 'a queue
    val is_empty : 'a queue -> bool
    exception EmptyQueue
    val dequeue : 'a queue
    val front : 'a queue -> 'a
  end
```
module type CONTAINER =

  sig
  type 'a t
  val empty : unit -> 'a t
  val insert : 'a -> 'a t -> 'a t
  val is_empty : 'a t -> bool
  exception Empty
  val remove : 'a t -> 'a t
  val first : 'a t -> 'a
  end

Slap the same interface on both the Queue module and the Stack module -- interfaces are reuseable in ML!

This lets us write an algorithm, like a tree traversal, using a generic container interface. To get depth-first traversal, use the stack; to get breadth-first traversal, use the queue; to get prioritized traversal, use a priority queue; etc.
FUNCTORS
• Suppose I ask you to write a generic package for matrices.
  – e.g., matrix addition, matrix multiplication
• The package should be parameterized by the element type.
  – We may want to use ints or floats or complex numbers or binary values or ... for the elements.
• What we'll see:
  – **RING**: a signature to describe the type (and necessary operations) for matrix elements
  – **MATRIX**: a signature to describe the available operations on matrices
  – **DenseMatrix**: a functor that will generate a MATRIX with a specific RING as an element type
module type RING =

    sig
        type t
        val zero : t
        val one  : t
        val add : t -> t -> t
        val mul : t -> t -> t
    end
module IntRing =
  struct
    type t = int
    let zero = 0
    let one = 1
    let add x y = x + y
    let mul x y = x * y
  end

module FloatRing =
  struct
    type t = float
    let zero = 0.0
    let one = 1.0
    let add = (+.)
    let mul = ( *. )
  end

module BoolRing =
  struct
    type t = bool
    let zero = false
    let one = true
    let add x y = x || y
    let mul x y = x && y
  end
module type MATRIX =
  sig
    type elt
    type matrix
    val matrix_of_list : elt list list list -> matrix
    val add : matrix -> matrix -> matrix
    val mul : matrix -> matrix -> matrix
  end
module DenseMatrix (R:RING) : (MATRIX with elt = R.t) =
struct
...
end
The DenseMatrix Functor

```plaintext
module DenseMatrix (R:RING) : (MATRIX with elt = R.t) =
struct
  ...
end
```

- Argument R must be a RING
- Specify Result.elt = R.t
- Result must be a MATRIX
module DenseMatrix (R:RING) : (MATRIX with elt = R.t) =
struct
...
end

module IntMatrix = DenseMatrix(IntRing)
module FloatMatrix = DenseMatrix(FloatRing)
module BoolMatrix = DenseMatrix(BoolRing)

Use DenseMatrix like it is a function from modules to modules
module DenseMatrix (R:RING) : (MATRIX with elt = R.t) =
struct
  ...
end

module type MATRIX =
sig
  type elt
  type matrix

  val matrix_of_list : elt list list -> matrix

  val add : matrix -> matrix -> matrix
  val mul : matrix -> matrix -> matrix
end

module IntMatrix = DenseMatrix(IntRing)
module FloatMatrix = DenseMatrix(FloatRing)
module BoolMatrix = DenseMatrix(BoolRing)
The DenseMatrix Functor

module DenseMatrix (R:RING) : (MATRIX with elt = R.t) =
struct

If the "with" clause is redacted then IntMatrix.elt is abstract -- we could never build a matrix because we could never generate an elt

end

module type MATRIX =
sig
    type elt
    type matrix

    val matrix_of_list : elt list list list -> matrix

    val add : matrix -> matrix -> matrix
    val mul : matrix -> matrix -> matrix
end

module IntMatrix = DenseMatrix(IntRing)
module FloatMatrix = DenseMatrix(FloatRing)
module BoolMatrix = DenseMatrix(BoolRing)
The DenseMatrix Functor

module DenseMatrix (R:RING) : (MATRIX with elt = R.t) =

struct
...
end

module type MATRIX =

sig

type elt = int

type matrix

val matrix_of_list : elt list list -> matrix

val add : matrix -> matrix -> matrix

val mul : matrix -> matrix -> matrix

end

module IntMatrix = DenseMatrix(IntRing)
module FloatMatrix = DenseMatrix(FloatRing)
module BoolMatrix = DenseMatrix(BoolRing)
The DenseMatrix Functor

```ocaml
module DenseMatrix (R:RING) : (MATRIX with elt = R.t) =
struct

    The "with" clause makes IntMatrix.elt equal to int -- we can build a matrix from any int list list

    module type MATRIX =
    sig
        type elt = int
        type matrix

        val matrix_of_list : elt list list -> matrix
        val add : matrix -> matrix -> matrix
        val mul : matrix -> matrix -> matrix
    end

    list of list of ints

end

module IntMatrix = DenseMatrix(IntRing)
module FloatMatrix = DenseMatrix(FloatRing)
module BoolMatrix = DenseMatrix(BoolRing)
```

sharing constraint

known to be int when R.t = int like when R = IntRing

list of list of ints
module DenseMatrix (R:RING) : (MATRIX with elt = R.t) =
struct
  type elt = R.t
  type matrix = (elt list) list
let matrix_of_list rows = rows
let add m1 m2 =
  List.map (fun (r1,r2) ->
    List.map (fun (e1,e2) -> R.add e1 e2))
    (List.combine r1 r2))
    (List.combine m1 m2)
let mul m1 m2 = (* good exercise *)
end

module IntMatrix = DenseMatrix(IntRing)
module FloatMatrix = DenseMatrix(FloatRing)
module BoolMatrix = DenseMatrix(BoolRing)
module **type** BASE =
  sig
    val base : int
  end

module UbignumGenerator(Base:BASE) : UNSIGNED_BIGNUM =
struct
  type ubignum = int list
  let toInt(b:ubignum):int =
    List.fold_left (fun a c -> c*Base.base + a) 0 b ...  
end

module Ubignum_10 =
  UbignumGenerator(struct let base = 10 end) ;;;

module Ubignum_2 =
  UbignumGenerator(struct let base = 2 end) ;;;
A module matches any interface as long as it provides \textit{at least} the definitions (of the right type) specified in the interface.

But as we saw earlier, the module can have more stuff.  
– e.g., the \texttt{deq} function in the \texttt{Queue} modules

Basic principle of subtyping for modules:  
– wherever you are expecting a module with signature \( S \), you can use a module with signature \( S' \), as long as all of the stuff in \( S \) appears in \( S' \).  
– That is, \( S' \) is a bigger interface.
module type GROUP =
    sig
      type t
      val zero : t
      val add : t -> t -> t
    end
module type RING =
    sig
      type t
      val zero : t
      val one : t
      val add : t -> t -> t
      val mul : t -> t -> t
    end
module IntGroup : GROUP = IntRing
module FloatGroup : GROUP = FloatRing
module BoolGroup : GROUP = BoolRing
module type GROUP =
  sig
    type t
    val zero : t
    val add : t -> t -> t
  end
module type RING =
  sig
    type t
    val zero : t
    val one : t
    val add : t -> t -> t
    val mul : t -> t -> t
  end
module IntGroup : GROUP = IntRing
module FloatGroup : GROUP = FloatRing
module BoolGroup : GROUP = BoolRing

RING is a sub-type of GROUP.
module type GROUP =
  sig
    type t
    val zero : t
    val add : t -> t -> t
  end
module type RING =
  sig
    type t
    val zero : t
    val one : t
    val add : t -> t -> t
    val mul : t -> t -> t
  end
module IntGroup : GROUP = IntRing
module FloatGroup : GROUP = FloatRing
module BoolGroup : GROUP = BoolRing

Groups versus Rings

There are more modules matching the GROUP interface than the RING one.
Any module expecting a GROUP can be passed a RING.
module type GROUP =
  sig
    type t
    val zero : t
    val add : t -> t -> t
  end
module type RING =
  sig
    include GROUP
    val one : t
    val mul : t -> t -> t
  end
module IntGroup : GROUP = IntRing
module FloatGroup : GROUP = FloatRing
module BoolGroup : GROUP = BoolRing

The include primitive is like cutting-and-pasting the signature’s content here.
module type GROUP =
	sig
	  type t
	  val zero : t
	  val add : t -> t -> t
	end
module type RING =
	sig
	  include GROUP
	  val one : t
	  val mul : t -> t -> t
	end
module IntGroup : GROUP = IntRing
module FloatGroup : GROUP = FloatRing
module BoolGroup : GROUP = BoolRing

That ensures we will be a sub-type of the included signature.
module type SET =

sig
  type elt
  type set
  val empty : set
  val is_empty : set -> bool
  val insert : elt -> set -> set
  val singleton : elt -> set
  val union : set -> set -> set
  val intersect : set -> set -> set
  val remove : elt -> set -> set
  val member : elt -> set -> bool
  val choose : set -> (elt * set) option
  val fold : (elt -> 'a -> 'a) -> 'a -> set -> 'a
end
Our Set Implementation is a Functor:

```ocaml
module ListSet (Elt : sig type t end) : (SET with elt = Elt.t) =

struct
  type elt = Elt.t
  type set = elt list
  let empty : set = []
  let is_empty (s:set) =
    match xs with
    | [] -> true
    | _::_ -> false
  let singleton (x:elt) : set = [x]
...
end

module IntListSet = ListSet(struct type t = int end)
module StringListSet = ListSet(struct type t = string end)
```
Our Set Implementation is a Functor:

```ocaml
module ListSet (Elt : sig type t end) : (SET with elt = Elt.t) =
struct
  type elt = Elt.t
  type set = elt list
  let empty : set = []
  let is_empty (s:set) =
    match xs with
    | [] -> true
    | _::_ -> false
  let singleton (x:elt) : set = [x]
...
end
```

ListSet is a parameterized module – given a module argument for Elt, it generates a new module.

```ocaml
module IntListSet = ListSet(struct type t = int end)
module StringListSet = ListSet(struct type t = string end)
```
Our Set Implementation is a Functor:

```ocaml
module ListSet (Elt : sig type t end) : (SET with elt = Elt.t) =
struct
  type elt = Elt.t
  type set = elt list
  let empty : set = []
  let is_empty (s:set) =
    match xs with
    | [] -> true
    | _::_ -> false
  let singleton (x:elt) : set = [x]
...
end
```

This is a very simple, anonymous signature (it just specifies there’s some type t) for the argument to ListSet.

```ocaml
module IntListSet = ListSet(struct type t = int end)
module StringListSet = ListSet(struct type t = string end)
```
Our Set Implementation is a Functor:

```ocaml
module ListSet (Elt : sig type t end) : (SET with elt = Elt.t) =
  struct
    type elt = Elt.t
    type set = elt list
    let empty : set = []
    let is_empty (s:set) =
      match xs with
      | [] -> true
      | _::_ -> false
    let singleton (x:elt) : set = [x]
  ...
end

module IntListSet = ListSet(struct type t = int end)
module StringListSet = ListSet(struct type t = string end)
```

This is the signature of the resulting module – we have a set plus the knowledge that the Set’s elt type is equal to Elt.t
Our Set Implementation is a Functor:

module ListSet (Elt : sig type t end) :
  (SET with elt = Elt.t) =

struct
  type elt = Elt.t
  type set = elt list
let empty : set = []
let is_empty (s:set) =
  match xs with
    | [] -> true
    | _::_ -> false
let singleton (x:elt) : set = [x]
...
end

module IntListSet = ListSet(struct type t = int end)
module StringListSet = ListSet(struct type t = string end)

These are two SET modules that I created with the ListSet functor.
module ListSet (Elt : sig type t end) :
  (SET with elt = Elt.t) =

struct
  type elt = Elt.t
  type set = elt list
let empty : set = []
let is_empty (s:set) =
  match xs with
  | [] -> true
  | _::_ -> false
let singleton (x:elt) : set = ...
end

module IntListSet = ListSet(struct type t = int end)
module StringListSet = ListSet(struct type t = string end)

Our Set Implementation is a Functor:

In this case, I’m passing in an anonymous module for Elt that defines t to be int.
module ListSet (Elt : sig type t end) : (SET with elt = Elt.t) =

struct
type elt = Elt.t
type set = elt list
let empty : set = []
let is_empty (s: set) =
  match xs with
  | [] -> true
  | _::_ -> false
let singleton (x:elt) : set = [x]
...
end

module IntListSet = ListSet(struct type t = int end)
module StringListSet = ListSet(struct type t = string end)

We know that IntListSet.elt = int.
module ListSet (Elt : sig type t end) : (SET with elt = Elt.t) =

struct
    type elt = Elt.t
    type set = elt list
    let empty : set = []
    let is_empty (s:set) = match xs with
        | [] -> true
        | _::_ -> false
    let singleton (x:elt) : set = [x]
...
end

module type SET =
    sig
        type elt
        type set
        val empty : set
        val is_empty : set -> bool
        val insert : elt -> set -> set
        ...
    end

module IntListSet = ListSet(struct type t = int end)
module StringListSet = ListSet(struct type t = string end)

Our Set Implementation is a Functor:

We know that IntListSet.elt = int.
so we can actually build a set using insertions!
module ListSet (Elt : sig type t end) : (SET with elt = Elt.t) =

struct
    type elt = Elt.t
    type set = elt list
    let empty : set = []
    let is_empty (s:set) =
        match xs with
        | [] -> true
        | _::_ -> false
    let singleton (x:elt) : set = [x]
    let insert (x:elt) (s:set) : set =
        if List.mem x s then s else x::s
    ...
end
module ListSet (Elt : sig type t end)
   :(SET with elt = Elt.t) =

struct
  type elt = Elt.t
  type set = elt list
...
  let insert (x:elt) (s:set) : set =
      if List.mem x s then s else x::s
  let union (s1:set) (s2:set) : set = ???
end
Let’s Write the Rest of the Functor

module ListSet (Elt : sig type t end) :
  (SET with elt = Elt.t) =
struct
  type elt = Elt.t
  type set = elt list
...
let insert (x:elt) (s:set) : set =
  if List.mem x s then s else x::s
let union (s1:set) (s2:set) : set =
  s1 @ s2
...
end

Ugh. Wastes space if s1 and s2 have duplicates. (Also, makes remove harder...)
Let’s Write the Rest of the Functor

module ListSet (Elt : sig type t end) : (SET with elt = Elt.t) =

struct
  type elt = Elt.t
  type set = elt list
  ...
  let insert (x:elt) (s:set) : set =
    if List.mem x s then s else x:::s
  let union (s1:set) (s2:set) : set =
    List.fold_right insert s1 s2
  ...
end

Gets rid of the duplicates. Now remove can stop once it finds the element.
module ListSet (Elt : sig type t end) : (SET with elt = Elt.t) =

struct
  type elt = Elt.t
  type set = elt list
...
  let insert (x:elt) (s:set) : set =
    if List.mem x s then s else x::s
  let union (s1:set) (s2:set) : set =
    List.fold_right insert s1 s2
...
end

Gets rid of the duplicates. Now remove can stop once it finds the element.
module ListSet (Elt : sig type t end) : (SET with elt = Elt.t) =

struct
  type elt = Elt.t
  type set = elt list
...

let insert (x:elt) (s:set) : set =
  if List.mem x s then s else x::s

let union (s1:set) (s2:set) : set =
  List.fold_right insert s1 s2
...

end

But List.mem and List.fold_right take time proportional to the length of the list. So union is quadratic.

Gets rid of the duplicates. Now remove can stop once it finds the element.
Let’s Write the Rest of the Functor

```ocaml
module ListSet (Elt : sig type t end) : (SET with elt = Elt.t) =

struct
  type elt = Elt.t
  type set = elt list

  ...

  let insert (x:elt) (s:set) : set =
    if List.mem x s then s else x::s

  let union (s1:set) (s2:set) : set =
    List.fold_right insert s1 s2

  ...

end
```

If we knew that s1 and s2 were sorted we could use the merge from mergesort to compute the sorted union in linear time.
module type COMPARATOR = sig
  type t
  val compare : t -> t -> Order.order
end

module SortedListSet (Elt : COMPARATOR)
  : (SET with elt = Elt.t) =
struct ...
  let rec insert (x:elt) (s:set) : set =
    match s with
    | [] -> [x]
    | h::t -> (match Elt.compare x h with
               | Less -> x::s
               | Eq -> s
               | Greater -> h::(insert x t)) ...
end
module type COMPARATOR = sig
  type t
  val compare : t -> t -> Order.order
end

module SortedListSet (Elt : COMPARATOR) : (SET with elt = Elt.t) =
  struct ...
    let rec insert (x:elt) (s:set) =
      match s with
      | [] -> [x]
      | h::t -> (match Elt.compare x h with
                  | Less -> x::s
                  | Eq -> s
                  | Greater -> h::(insert x t)) ...
  end

To support the sorting, I’m passing in a comparison operation to go with the element type.
module SortedListSet (Elt : COMPARATOR)
  : (SET with elt = Elt.t) =

struct ...

  let rec union (s1:set) (s2:set) : set =
    match s1, s2 with
    | [], _ -> s2
    | _, [] -> s1
    | h1::t1, h2::t2 ->
      (match Elt.compare h1 h2 with
       | Less -> h1::(union t1 s2)
       | Eq  -> h1::(union t1 t2)
       | _   -> h2::(union s1 t2))

  ...

end
module SortedListSet (Elt : COMPARATOR) : (SET with elt = Elt.t) =

struct ...

  let rec union (s1:set) (s2:set) : set = ...

  let insert (x:elt) (s:set) : set = union [x] s ;;

end
module BitVectorSet (Elt : sig type t
    val index : t -> int
    val max : int
  end)
  : (SET with elt = Elt.t) =

struct
  type set = bool array
  let empty = Array.create Elt.max false
  let member x s = s.(Elt.index x)
  let union s1 s2 =
    Array.init Elt.max
    (fun i -> s1.(i) || s2.(i))
  let intersect s1 s2 =
    Array.init Elt.max
    (fun i -> s1.(i) && s2.(i))
  ...

module BSTreeSet(Elt : sig
    type t
    val compare : t -> t -> Order.order
end) : (SET with elt = Elt.t) =

struct
    type set = Leaf | Node of set * elt * set

let empty() = Leaf

let rec insert (x:elt) (s:set) : set =
    match s with
    | Leaf -> Node(Leaf,x,Leaf)
    | Node(left,e,right) ->
        (match Elt.compare x e with
        | Eq -> s
        | Less -> Node(insert x left, e, right)
        | Greater -> Node(left, e, insert x right))

let rec member (x:elt) (s:set) : bool =
    match s with
    | Leaf -> false
    | Node(left,e,right) ->
        (match Elt.compare x e with
        | Eq -> true
        | Less -> member x left
        | Greater -> member x right)

... end
• It is often tempting to break the abstraction barrier.
  – e.g., during development, you want to print out a set, so you just call a convenient function you have lying around for iterating over lists and printing them out.

• But the whole point of the barrier is to support future change in implementation.
  – e.g., moving from unsorted invariant to sorted invariant.
  – or from lists to balanced trees.

• Many languages provide ways to leak information through the abstraction barrier.
  – “good” clients should not take advantage of this.
  – but they always end up doing it.
  – so you end up having to support these leaks when you upgrade, else you’ll break the clients.
• It is often tempting to break the abstraction barrier.
  – e.g., during development, you want to print out a set, so you just call a convenient function you have lying around for iterating over lists and printing them out.
• But the whole point of the barrier is to support future change in implementation.
  – e.g., moving from unsorted invariant to sorted invariant.
  – or from lists to balanced trees.
• Many languages provide ways to leak information through the abstraction barrier.
  – e.g., clients should not take advantage of this.
  – but they always end up doing it.
  – so you end up having to support these leaks when you upgrade, else you’ll break the clients.
Key Points

• Design in terms of *abstract* types and algorithms.
  – think “sets” not “lists” or “arrays” or “trees”
  – think “document” not “strings”

• Use linguistic mechanisms to insulate clients from implementations of mechanisms.
  – makes it easy to swap in new implementations
  – the *less* you reveal in an interface, the easier it is to replace the implementation
  – on the other hand, you need to reveal enough in the interface to make it useful for clients.

• In Ocaml, we can use the module system
  – provides support for *name-spaces*
  – *hiding information* (types, local value definitions)
  – *code reuse* (via functors, reusable interfaces, reusable modules)