```
(* CS 251: ML Modules and Abstract Data Types *)

signature MATHLIB =
sig
    val fact : int -> int
    val half_pi : real
    (* val doubler : int -> int *) (* can hide bindings from clients *)
end

structure MyMathLib :> MATHLIB =
struct
    fun fact 0 = 1
      | fact x = x * fact (x - 1)

    val half_pi = Math.pi / 2.0

    fun doubler y = y + y
end

val pi = MyMathLib.half_pi + MyMathLib.half_pi

(* val twenty_eight = MyMathLib.doubler 14 *)


(* This signature hides gcd and reduce.  Clients cannot assume they
   exist or call them with unexpected inputs.  But clients can still
   build rational values directly with the constructors Whole and
   Frac.  This makes it impossible to maintain invariants about
   rationals, so we might have negative denominators, which some
   functions do not handle, and toString may print a non-reduced
   fraction. *)
signature RATIONAL_CONCRETE =
sig
    datatype rational = Frac of int * int | Whole of int
    exception BadFrac
    val make_frac : int * int -> rational
    val add : rational * rational -> rational
    val toString : rational -> string
end

(* This signature abstracts the rational type.  Clients can acquire
   values of type rational using make_frac and manipulate them using
   add and toString, but they have know way to inspect the
   representation of these values or create them on their own.  They
   are tightly sealed black boxes. This ensures that any invariants
   established and assumed inside an implementation of this signature
   cannot be violated by external code.

   This is a true Abstract Data Type. *)
signature RATIONAL =
sig
    type rational (* type now abstract *)
    exception BadFrac
    val make_frac : int * int -> rational
    val add : rational * rational -> rational
    val toString : rational -> string
end

(* As a cute trick, it is actually okay to expose the Whole
   function since no value breaks our invariants, and different
   implementations can still implement Whole differently.
   Clients know only that Whole is a function.
   Cannot use as pattern. *)
signature RATIONAL_WHOLE =
sig
    type rational (* type still abstract *)
    exception BadFrac
    val Whole : int -> rational
    val make_frac : int * int -> rational
    val add : rational * rational -> rational
    val toString : rational -> string
end

(* Can ascribe any of the 3 signatures above.  We choose to use the
   Abstract Data Type. *)
structure Rational :> RATIONAL =
struct

  (* Invariant 1: all denominators > 0
     Invariant 2: rationals kept in reduced form *)

  datatype rational = Whole of int | Frac of int*int
  exception BadFrac

  (* gcd and reduce help keep fractions reduced,
     but clients need not know about them *)
  (* they _assume_ their inputs are not negative *)
  fun gcd (x,y) =
      if x=y
      then x
      else if x < y
      then gcd (x,y-x)
      else gcd (y,x)

  fun reduce r =
      case r of
          Whole _ => r
        | Frac (x,y) =>
          if x=0
          then Whole 0
          else let val d = gcd (abs x,y) in (* using invariant 1 *)
                   if d=y
                   then Whole (x div d)
                   else Frac (x div d, y div d)
               end

  (* When making a frac, ban zero denominators and put valid fractions
     in reduce form. *)
  fun make_frac (x,0) = raise BadFrac
    | make_frac (x,y) =
      if y < 0
      then reduce (Frac (~x,~y))
      else reduce (Frac (x,y))

  (* Using math properties, both invariants hold for the result
     assuming they hold for the arguments. *)
  fun add (Whole (i), Whole (j))   = Whole (i+j)
    | add (Whole (i), Frac (j,k))  = Frac (j+k*i,k)
    | add (Frac (j,k), Whole (i))  = Frac (j+k*i,k)
    | add (Frac (a,b), Frac (c,d)) = reduce (Frac (a*d + b*c, b*d))
```

```
  (* Assuming r is in reduced form, print r in reduced form *)
  fun toString (Whole i) = Int.toString i
    | toString (Frac (a,b)) = (Int.toString a) ^ "/" ^ (Int.toString b)

end


(* This structure can have all three signatures we gave
   Rational, and/but it is *equivalent* under signatures
   RATIONAL and RATIONAL_WHOLE.

   This structure does not reduce fractions until printing.
 *)
structure UnreducedRational :> RATIONAL (* or the others *) =
struct
    datatype rational = Whole of int | Frac of int*int
    exception BadFrac

    fun make_frac (x,0) = raise BadFrac
      | make_frac (x,y) =
        if y < 0
        then Frac (~x,~y)
        else Frac (x,y)

    fun add (Whole (i), Whole (j))   = Whole (i+j)
      | add (Whole (i), Frac (j,k))  = Frac (j+k*i,k)
      | add (Frac (j,k), Whole (i))  = Frac (j+k*i,k)
      | add (Frac (a,b), Frac (c,d)) = Frac (a*d + b*c, b*d)

    fun toString r =
        let fun gcd (x,y) =
                if x=y
                then x
                else if x < y
                then gcd (x,y-x)
                else gcd (y,x)

            fun reduce r =
                case r of
                    Whole _ => r
                  | Frac (x,y) =>
                    if x=0
                    then Whole 0
                    else
                        let val d = gcd (abs x,y) in
                            if d=y
                            then Whole (x div d)
                            else Frac (x div d, y div d)
                        end
        in
            case reduce r of
                Whole i   => Int.toString i
              | Frac (a,b) => (Int.toString a) ^ "/" ^ (Int.toString b)
        end
end


(* This structure uses a different concrete representation of the
   abstract type.  We cannot ascribe signature RATIONAL_CONCRETE to
   it.  To ascribe RATIONAL_WHOLE, we must add a Whole function.  It
   is indistinguishable from Rational under these two signatures. *)
structure PairRational :> RATIONAL (* or RATIONAL_WHOLE *)= struct
    type rational = int * int
    exception BadFrac

    fun make_frac (x,0) = raise BadFrac
      | make_frac (x,y) =
        if y < 0
        then (~x,~y)
        else (x,y)

    fun Whole i = (i,1)

    fun add ((a,b),(c,d)) = (a*d + c*b, b*d)

    fun toString (0,y) = "0"
      | toString (x,y) =
        let fun gcd (x,y) =
                  if x=y
                  then x
                  else if x < y
                  then gcd(x,y-x)
                  else gcd(y,x)
                val d = gcd (abs x,y)
                val num = x div d
                val denom = y div d
                val numString = Int.toString num
        in
           if denom=1
           then numString
           else numString ^ "/" ^ (Int.toString denom)
        end
end
```

```
(* ADT exercises.

   Complete two implementations of a set ADT with the SET signature:
   a list-based representation and a function-based representation.

   You may ignore "Warning: calling polyEqual" in this exercise. *)

(* Placeholder during development. *)
exception Unimplemented

(* SET describes operations over set values of type ''a t, where set
   elements are of type ''a.  Recall that the double-quote type
   variable ''a means that values of the type ''a can be compared
   using the = operation.

   Naming the type of the ADT t is a common idiom for signatures
   defining an ADT.  This means that for particular implementations
   (e.g., ListSet or FunSet), ADT values have type ListSet.t or
   FunSet.t, rather than the more verbose ListSet.set or FunSet.set.
   If a signature defines multiple types (especially if there's not
   one main type and other supporting types), this idiom is less
   commonly used. *)

signature SET =
sig
    (* The type of sets *)
    type ''a t

    (* An empty set *)
    val empty : ''a t

    (* Construct a single-element set from that element. *)
    val singleton : ''a -> ''a t

    (* Construct a set from a list of elements.
       Do not assume the list elements are unique. *)
    val fromList : ''a list -> ''a t

    (* Convert a set to a list. *)
    val toList : ''a t -> ''a list

    (* Construct a set from a predicate function:
       the resulting set should contain all elements for which
       this predicate function returns true.

       This acts like math notation for sets.  For example:
         { x | x mod 3 = 0 }
       would be written:
         fromPred (fn x => x mod 3 = 0)
    *)
    val fromPred : (''a -> bool) -> ''a t

    (* Convert a set to a predicate function. *)
    val toPred : ''a t -> ''a -> bool

    (* Convert a set to a string representation, given a function
       that converts a set element into a string representation. *)
    val toString : (''a -> string) -> ''a t -> string

    (* Check if a set is empty. *)
    val isEmpty : ''a t -> bool

    (* Check if a given element is a member of the given set. *)
    val member : ''a -> ''a t -> bool

    (* Construct a set containing the given element and all elements
       of the given set. *)
    val insert : ''a -> ''a t -> ''a t

    (* Construct a set containing all elements of the given set
       except for the given element. *)
    val delete : ''a -> ''a t -> ''a t

    (* Construct the union of two sets. *)
    val union : ''a t -> ''a t -> ''a t

    (* Construct the intersection of two sets. *)
    val intersection : ''a t -> ''a t -> ''a t

    (* Construct the symmetric difference of two sets. *)
    val difference : ''a t -> ''a t -> ''a t
end


(* Implement a SET ADT using lists to represent sets. *)
structure ListSet :> SET =
struct
    (* Sets are represented by lists. *)
    type ''a t = ''a list

    (* The empty set is the empty list. *)
    val empty = []

    fun fromPred _ = raise Unimplemented (* impossible *)

    (* complete this structure with implementations of all of the
       bindings given in the SET signature *)
end


(* Implement a SET ADT representing sets by predicate functions. *)
structure FunSet :> SET =
struct
    (* Sets are represented by functions that return true if the
       given element is in the set and false if it is not. *)
    type ''a t = ''a -> bool

    (* The empty set is a function that returns false on
       all arguments. *)
    fun empty _ = false

    (* The singleton set is a function that checks to see if
       its argument is the one element of the set. *)
    fun singleton x = fn y => y=x

    (* complete this structure with implementations of all of the
       bindings given in the SET signature.  Not all are possible.
       Which are not? *)
end
```