```
(* 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
```