Now we can specify the most general types of functions like insert and insertSort:

```
> insert :: Ord a => a -> [a] -> [a]
> insert x [] = [x]
> insert x (y:ys)
>   | x <= y    = x:y:ys
>   | otherwise = y : insert x ys

> insertSort ::  Ord a => [a] -> [a]
> insertSort = foldr insert []
```

Bool is an instance of Ord as well as of Eq:

```
> instance Ord Bool where
>     False <= _      = True
>     True  <= x      = x
```

Int, Integer, Float, and Double belong to Eq and Ord:

```
> instance Eq  Int where (==)    = primEqInt
> instance Ord Int where compare = primCmpInt
> instance Ord Integer
>   where compare = primCmpInteger
> instance Eq  Float where (==) = primEqFloat
> instance Ord Float where compare = primCmpFloat
```

These prim… functions are Prelude names for functions which are provided by the platform (MacOS, Windows, AIX, Solaris, etc.).

```
> primitive primEqInt   :: Int -> Int -> Bool
> primitive primCmpInt :: Int -> Int -> Ordering
etc.
```


Lists belong not only to Eq but also to Ord —provided their elements' type belongs to Ord:

```
> instance Ord a => Ord [a] where
>   compare []    (_:_)  = LT
>   compare []    []     = EQ
>   compare (_:_) []     = GT
>   compare (x:xs) (y:ys)
>         = primCompAux x y (compare xs ys)
>  where
>  primCompAux
>     :: Ord a => a -> a -> Ordering -> Ordering
>     primCompAux x y o
>        = case compare x y of
>            LT -> LT; EQ -> o; GT -> GT
```

Numbers need more than Eq guarantees—

```
> class (Eq a, Show a) => Num a where
>     (+), (-), (*)  :: a -> a -> a
>     negate         :: a -> a
>     abs, signum    :: a -> a
>     fromInteger    :: Integer -> a
>     fromInt        :: Int -> a
>     x - y           = x + negate y
>     fromInt         = fromIntegral
>     negate x        = 0 - x
```

(Types in class Show have functions that convert their values to String.)

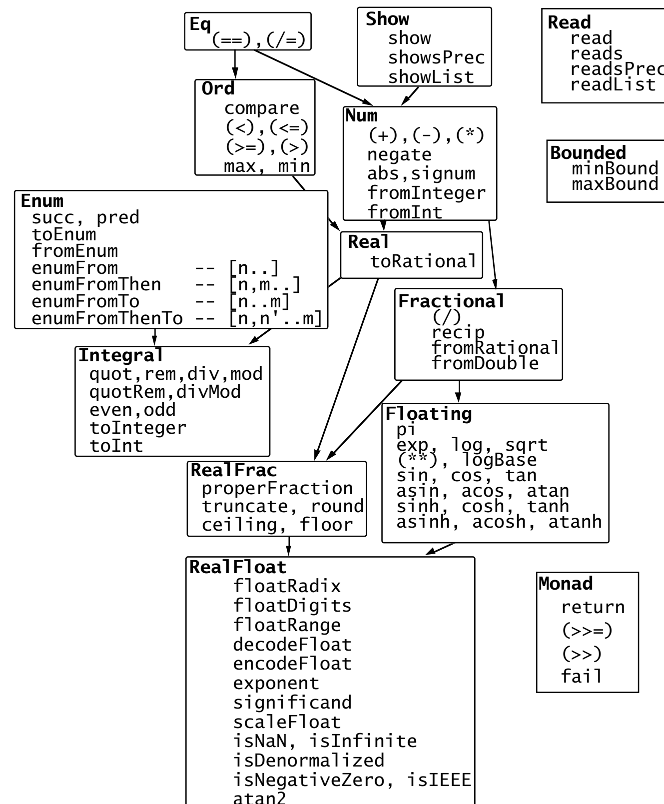
Num is derived from Eq, but not from Ord, because not all "numeric" types support (<=); for example—complex numbers and matrices.

Two of the instance declarations for Num types:

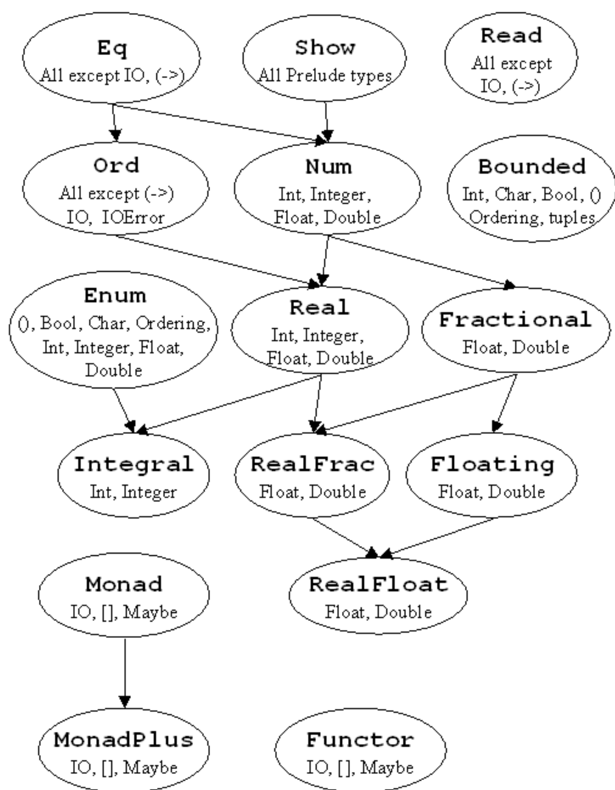
```
> instance Num Int where
>     (+)          = primPlusInt
>     (-)          = primMinusInt
>     negate       = primNegInt
>     (*)          = primMulInt
>     abs          = absReal
>     signum       = signumReal
>     fromInteger  = primIntegerToInt
>     fromInt x    = x

> instance Num Float where
>     (+)          = primPlusFloat
>     (-)          = primMinusFloat
>     negate       = primNegFloat
>     (*)          = primMulFloat
>     abs          = absReal
>     signum       = signumReal
>     fromInteger  = primIntegerToFloat
>     fromInt      = primIntToFloat
```


The hierarchy of Haskell classes defined in the Prelude, and the classes' signatures:



The hierarchy of Haskell classes defined in the Prelude, and the Prelude types that are instances of these classes:



Type classes and Algebraic types

For some classes, the Haskell compiler can derive algebraic-type instance declarations automatically:

For example,

```
> data Day = Sun | Mon | Tue | Wed | Thu | Fri | Sat
>    deriving (Eq, Ord, Enum, Show, Read, Bounded)
```

(Enum instances valid only for enumerated types).

```
Main> Tue == Tue
True
Main> Mon < Wed
True
Main> Mon > Wed
False
Main> max Tue Thu
Thu
Main> sort [Thu,Tue,Sun,Wed]
[Sun,Tue,Wed,Thu]
Main> [Mon .. Fri]
[Mon,Tue,Wed,Thu,Fri]
Main> read "Sat" :: Day
Sat
Main> reads "Sat.." :: [(Day,String)]
[(Sat,"..")]
Main> maxBound :: Day
Sat
Main> minBound :: Day
Sun
```


Derived instances aren't always useful.

```
> data Temp  = Celsius Float | Fahrenheit Float
>      deriving (Eq, Ord, Show, Read)
Main> read "Celsius 12" :: Temp
Celsius 12.0
Main> show (Fahrenheit 451)
"Fahrenheit 451.0"
Main> Celsius 35 < Celsius 40
True
Main> Celsius 35 < Fahrenheit 0
True                                   -- oops
Main> Celsius 0 == Fahrenheit 32
False                                  -- oops
```

Better:

```
> data Temp  = Celsius Float | Fahrenheit Float
>      deriving (Show, Read)

> instance Eq Temp where
>   t1 == t2 = degreesC t1 == degreesC t2

> instance Ord Temp where
>   t1 <= t2   =   degreesC t1 <= degreesC t2

> degreesC :: Temp -> Float
> degreesC (Celsius d)    = d
> degreesC (Fahrenheit d) = 5/9 * (d - 32)
```


Procedures and Functions in Imperative Languages: Definition & Invocation

Terminology:

functions *return values* —
their invocations can be used as expressions
(can also cause state changes)

procedures cause ***state changes only*** —
their invocations cannot be used as expressions

C nomenclature uses “function” for both.

Modula–2 nomenclature uses “PROCEDURE” for both (*function* procedures have return types)

Sethi's nomenclature:

function procedure (or “function”)
proper procedure (or “procedure”)

A procedure (either kind) is an **abstraction** (i.e., a generalization)

body: a program text that is defined once, used many times with different bindings for its free names

parameters: names that are free in the procedure's body, bound to caller's arguments (variables or expressions) at each call