data Time = Seconds Int | Minutes Int

instance Eq Time where
  Seconds x == Seconds y = x == y
  Minutes x == Minutes y = x == y
  Seconds x == Minutes y = x == y*60
  Minutes x == Seconds y = x*60 == y

1. Define a function \( \text{secs} :: \text{Time} \rightarrow \text{Int} \)
2. Define an \( \text{Eq} \) instance for \text{Time} using \( \text{secs} \)

\[
\text{secs} :: \text{Time} \rightarrow \text{Int}
\]
\[
\text{secs} \ (\text{Minutes} \ m) = 60*m
\]
\[
\text{secs} \ (\text{Seconds} \ s) = s
\]

instance Eq Time where
  \( t == t' = \text{secs} \ t == \text{secs} \ t' \)
Exercises

4. Define a function `member :: Int -> IntSet -> Bool`

5. Define a function `subset :: IntSet -> IntSet -> Bool`

```haskell
data IntSet = ESet
            | Ins Int IntSet

member :: Int -> IntSet -> Bool
member i (Ins j s) = i==j || member i s
member _ ESet      = False

subset :: IntSet -> IntSet -> Bool
subset ESet      _ = True
subset (Ins i s) t = member i s && subset s t
```
6. Define an `Eq` instance for `IntSet` using `subset`

```
instance Eq IntSet where
  s == s' = subset s s' && subset s' s
```
Exercises

7. Define an `Ord` instance for `Time`.

8. Define an `Ord` instance for `IntSet`.

Bad idea!
Sets are only partially ordered

```haskell
instance Ord Time where
t < t' = secs t < secs t'
```

```haskell
instance Ord IntSet where
s < s' = subset s s' && not (s==s')
```
9. Define a `Show` instance for `Time` that produces output like `0:13` or `10:06`.

(Just minutes and seconds)
10. Define an `Eq` instance for `List a`.

```haskell
instance Eq a => Eq (List a) where
    Nil       == Nil       = True
    Cons x xs == Cons y ys = x==y && xs==ys
    _         == _         =   False
```

```haskell
instance Show a => Show (Maybe a) where
    show (Just a) = show a
    show Nothing  = "?"
```

```haskell
data List a = Nil | Cons a (List a)
```

(==) function defined on type `a`

(==) function defined on type `List a`
More Polymorphic Instances


```haskell
data List a = Nil | Cons a (List a)

instance Show a => Show (List a) where
  show xs = "["++showElems xs++"]"

showElems :: Show a => List a -> String
showElems Nil          = ""
showElems (Cons x Nil) = show x
showElems (Cons x xs)  = show x++","++showElems xs
```
Multiple Class Constraints

12. What is the type of `isqrt`?

\[
isqrt x = \text{head} \ [y \mid y \leftarrow [x, x-1 \ldots 1], y\cdot y \leq x]
\]

\[
isqrt :: (\text{Num } a, \text{Enum } a, \text{Ord } a) \Rightarrow a \rightarrow a
\]

\[
isqrt x = \text{head} \ [y \mid y \leftarrow [x, x-1 \ldots 1], y\cdot y \leq x]
\]
13. Define a type class `BiFunctor` for binary type constructors that has a member function `bmap`.

```haskell
class BiFunctor f where
    bmap :: (a -> c) -> (b -> d) -> f a b -> f c d
```

14. Define the pair type constructor `(,)` as an instance of `BiFunctor`.

```haskell
instance BiFunctor (,) where
    bmap f g (x,y) = (f x, g y)
```
class BiFunctor f where
    bmap :: (a -> c) -> (b -> d) -> f a b -> f c d

15. Define three functions `mapL`, `mapR`, and `mapLR` to map a function along the left, right, and both arguments of a type given by a binary type constructor.

```haskell
mapL :: BiFunctor f => (a -> c) -> f a b -> f c b
    mapL f = bmap f id

mapR :: BiFunctor f => (b -> c) -> f a b -> f a c
    mapR f = bmap id f

mapLR :: BiFunctor f => (a -> b) -> f a a -> f b b
    mapLR f = bmap f f
```
Using BiFunctors

16. Define `Graph` as an instance of `BiFunctor`.

```haskell
instance BiFunctor Graph where
  bmap f g (G ns es) = G (map f ns)
                          (map (\(v,w,l)->(v,w,g l)) es)
```

17. What meaning do the functions `mapL` and `mapR` have for elements of type `Graph a b`?

```haskell
mapNodeLabel :: (a -> c) -> Graph a b -> Graph c b
mapNodeLabel = mapL

mapEdgeLabel :: (b -> c) -> Graph a b -> Graph a c
mapEdgeLabel = mapR
```