```
-- Signal Solution
-- | binusoidal of given frequency
sin5 :: Double -> Signal Double
sin5 freq = map7 (\freq'| freq' == freq
test1 = sin5 1
test2 = map6 (* sin5 0.1
average :: Fractional a => a -> a -> a
average x y = (x + y) / 2.0
average :: Fractional a => Signal a -> Signal a -> Signal a
averageIs xs ys = mapS average xs ys
-- | It can also be generalized to an arbitrary Applicative functor
averageA :: (Fractional a, Applicative f) => f a -> f a -> f a
averageA = liftA2 (\x y -> (x + y) / 2.0)
-- | or slightly shorter
averageA' :: (Fractional a, Applicative f) => f a -> f a -> f a
averageA' = liftA2 average
scale :: Num a => Signal a -> Signal a
scale = map5 (\|10*|, |1+|)
-- | Discrete Signal
discrete :: Signal Double -> Signal Int
discrete = map5 round
-- | convert to 'analog'
toBars :: Signal Int -> Signal String
toBars = map5 (\replicate '#')
displayLength = 100 :: Int
-- | display the signal as a number of points
displayIs :: Signal Double -> IO ()
displayIs ss = forM_ [0..displayLength] $ \k ->
    putStrLn (show ss ss)
-- | The display magic. Note how we take advantage of function composition,
-- types defined so far, etc.
image :: Signal Double -> IO()
image = display . toBars . discrete . scale
-- | Signal is an instance of functor
instance Functor Signal where fmap = mapS
instance Applicative Signal where pure = constS, (<*>) = ($$)
-- | Running a program is simply returning its semantics.
run :: Program a -> IOSem a
run = snd
-- | Program
type Input  = String
type Output = String
data Program a where
    Put    :: Char -> Program ()
    Get    :: Program (Maybe Char)
    Return :: a -> Program a
    Bind   :: Program b -> (b -> Program b) -> Program b
    Type IOSem a = (Input, Output) -> (a, Input, Output)
-- | run function translates its semantics
run :: Program a -> IOSem a
run (Put c)  = \(i, o) -> ((), i, c:o)
run Get      = \(c:i, o) -> (Nothing, (i, i))
run Get      = \(i, o) -> (Just c, (i, i))
run (Return x)  = \(i, o) -> (x, (i, o))
run (Bind p k)  = \(i, o) -> let (x, (i', o1) = run p
                                 (y, (i'', o2) = run (k x)
                                 in (y, (i, o1 ++ o2)
-- | Running a program is simply returning its semantics.
run :: Program a -> IOSem a
run = snd
-- | Program
type Input  = String
type Output = String
data Program a where
    Put    :: Char -> Program ()
    Get    :: Program (Maybe Char)
    Return :: a -> Program a
    Bind   :: Program b -> (b -> Program b) -> Program b
    Type IOSem a = (a, Input, Output)
-- | run function translates its semantics
run :: Program a -> IOSem a
run (Put c)  = \(_, i, o) -> ((), i, c:o)
run Get      = \(_, c:i, o) -> (Nothing, (i, i))
run Get      = \(_, i, o) -> (Just c, (i, i))
run (Return x)  = \(_, i, o) -> (x, (i, o))
run (Bind p k)  = \(_, i, o) -> let (x, (i', o1) = run p
                                    (y, (i'', o2) = run (k x)
                                    in (y, (i, o1 ++ o2)
-- | Parser
type Input = String
type Output = Int
data Program a where
    Put    :: Char -> Program ()
    Get    :: Program (Maybe Char)
    Return :: a -> Program a
    Bind   :: Program b -> (b -> Program b) -> Program b
    Type IOSem a = (a, Input, Output)
-- | run function translates its semantics
run :: Program a -> IOSem a
run (Put c)  = \(_, i, o) -> ((), i, c:o)
run Get      = \(_, c:i, o) -> (Nothing, (i, i))
run Get      = \(_, i, o) -> (Just c, (i, i))
run (Return x)  = \(_, i, o) -> (x, (i, o))
run (Bind p k)  = \(_, i, o) -> let (x, (i', o1) = run p
                                    (y, (i'', o2) = run (k x)
                                    in (y, (i, o1 ++ o2)
-- | Parser
type Input = Int
type Output = Int
data Program a where
    Put    :: Char -> Program ()
    Get    :: Program (Maybe Char)
    Return :: a -> Program a
    Bind   :: Program b -> (b -> Program b) -> Program b
    Type IOSem a = (a, Input, Output)
-- | run function translates its semantics
run :: Program a -> IOSem a
run (Put c)  = \(_, i, o) -> ((), i, c:o)
run Get      = \(_, c:i, o) -> (Nothing, (i, i))
run Get      = \(_, i, o) -> (Just c, (i, i))
run (Return x)  = \(_, i, o) -> (x, (i, o))
run (Bind p k)  = \(_, i, o) -> let (x, (i', o1) = run p
                                    (y, (i'', o2) = run (k x)
                                    in (y, (i, o1 ++ o2)
import Control.Monad(>>=>)
-- >[>>=]:: Monad m => (a -> m b) -> (b -> m c) -> (a -> m c)
-- >[>>=]:: (a -> b) -> (b -> c) -> (a -> c)
-- >[>>=]:: (a -> b) -> (b -> c) -> (a -> c)
type Output = String
data Program a = PutBind Char (Program a)
               | GetBind (Maybe Char) (Program a)
               | Return a
-- | It turns out that bind can still be defined!
instance Monad Program where
    return = Return
    (PutBind c p) >>= k = PutBind c (p >>= k)
    (GetBind f p) >>= k = GetBind f (p >>= k)
    Return x  >>= k = k x
-- | pulled the above code off of my hat, but...
-- We can "calculate" the correct definition of (>>=) using
-- the following instances of PutBind and GetBind
    -- @PutBind c p >>= putC c >>= p@
    -- @GetBind f p >>= k@
    -- @Return x >>= k@
    -- Line 1: return x >>= f  == f x
    -- Line 2: m >>= return    == m
    -- Line 3: (m >>= f) >>= g == m >>= (\x -> f x >>= g)
    -- For instance,
    -- GetBind f >>= k         { meaning of GetBind }
    -- (GetBind f >>= k) >>= k { meaning of GetBind }
    -- (GetBind f >>= k) >>= k { meaning of GetBind }
    -- GetBind (\c -> f c >>= k)
    -- GetBind f >>= k
    -- PutBind c >>= Return
    -- PutBind c >>= PutBind c >>= k
    -- PutBind c >>= GetBind f >>= k
    -- Program (Maybe Char)
    -- Program (Maybe Char)
type IOSem a = Program a
run :: Program a -> IOSem a
run (PutBind c p)  = \(x, i, o)
    where (x, i', o) = run p
run (GetBind f)    = \(x, i, o)
    where (x, i', o) = run (f Nothing)
run (GetBind f)    = \(x, c:i, o) -> run (f $ Just c) i
run (Return x)     = \(x, i, o) -> (x, i, o)
-- | Parser
type Input = Int
type Output = Int
data Program a where
    Put    :: Char -> Program ()
    Get    :: Program (Maybe Char)
    Return :: a -> Program a
    Bind   :: Program b -> (b -> Program b) -> Program b
    Type IOSem a = (a, Input, Output)
-- | run function translates its semantics
run :: Program a -> IOSem a
run (Put c)  = \(_, i, o) -> ((), i, c:o)
run Get      = \(_, c:i, o) -> (Nothing, (i, i))
run Get      = \(_, i, o) -> (Just c, (i, i))
run (Return x)  = \(_, i, o) -> (x, (i, o))
run (Bind p k)  = \(_, i, o) -> let (x, (i', o1) = run p
                                    (y, (i'', o2) = run (k x)
                                    in (y, (i, o1 ++ o2
```