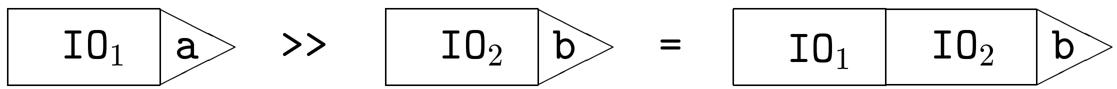
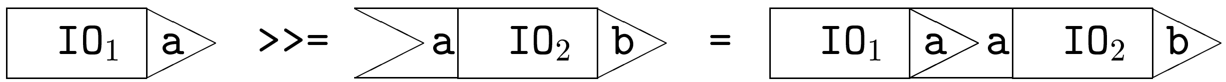
Functions Operating on the `IO` Monad

```
putChar :: Char -> IO ()                getChar :: IO Char

(>>) :: IO a -> IO b -> IO b            return :: a -> IO a
```




```
(>>=) :: IO a -> (a -> IO b) -> IO b
```



`getChar >> return ()`	reads a character and ignores it
`getChar >>= putChar`	reads a character and prints it on the screen

do-Notation

```
gets :: Int -> IO String
gets 0       = return []
gets (n+1) = getChar >>= \x  ->
                         gets n  >>= \xs ->
                                     return (x:xs)
```

```
p >>= \x ->
      q >>= \y ->
            r
```

can be written as

```
do x <- p
   y <- q
   r
```

```
gets :: Int -> IO String
gets 0       = return []
gets (n+1) = do x  <- getChar
                xs <- gets n
                return (x:xs)
```

Simple Evaluation Without Monads

```
data Value a = Result a


instance Show a => Show (Value a) where
         show (Result x) = "Result: " ++ show x


eval1 :: Term -> Value Float
eval1 (Con x)    = Result x
eval1 (Div t u) = Result (x/y)
                  where Result x = eval1 t
                        Result y = eval1 u
```

Evaluation and Exceptions Without Monads

```
data Maybe a = Nothing | Just a

instance Show a => Show (Maybe a) where
         show Nothing    = "Nothing"
         show (Just x)   = "Just " ++ show x

eval2 :: Term -> Maybe Float
eval2 (Con x)    = Just x
eval2 (Div t u) = case eval2 t of
                    Nothing -> Nothing
                    Just x  -> case eval2 u of
                                 Nothing -> Nothing
                                 Just y  -> if y == 0
                                              then Nothing
                                              else Just (x/y)
```

Evaluation and Counting Without Monads

```
data ST a = MakeST (Int -> (a, Int))

apply :: ST a -> Int -> (a, Int)
apply (MakeST f) s = f s

instance Show a => Show (ST a) where
  show tr = "Result: " ++ show x ++ ", State: " ++ show s
            where (x, s) = apply tr 0
```

```
eval3 :: Term -> ST Float
eval3 (Con x)   = MakeST (\s -> (x, s))
eval3 (Div t u) = MakeST (\s -> let (x, s')  = apply (eval3 t) s
                                    (y, s'') = apply (eval3 u) s'
                                in  (x/y, s''+1))
```

Simple Evaluation With Monads

```
data Value a = Result a

instance Show a => Show (Value a) where
         show (Result x) = "Result: " ++ show x

instance Monad Value where
                  return          = Result
                  Result x >>= q = q x

eval1 :: Term -> Value Float
eval1 (Con x)    = return x
eval1 (Div t u) = do x <- eval1 t
                     y <- eval1 u
                     return (x/y)
```

Evaluation and Exceptions With Monads

```
data Maybe a = Nothing | Just a

instance Show a => Show (Maybe a) where
         show Nothing    = "Nothing"
         show (Just x)   = "Just " ++ show x

instance Monad Maybe where
         return          = Just
         Nothing >>= q = Nothing
         Just x  >>= q = q x

eval2 :: Term -> Maybe Float
eval2 (Con x)    = return x
eval2 (Div t u) = do x <- eval2 t
                     y <- eval2 u
                     if y /= 0 then return (x/y) else Nothing
```

Evaluation and Counting With Monads

```
data ST a = MakeST (Int -> (a, Int))    apply :: ST a -> Int -> (a, Int)
                                        apply (MakeST f) s = f s


instance Show a => Show (ST a) where
  show tr = "Result: " ++ show x ++ ", State: " ++ show s
            where (x, s) = apply tr 0

instance Monad ST where
         return x = MakeST (\s -> (x, s))
         tr >>= q = MakeST (\s -> let (x, s')  = apply tr s
                                  in  apply (q x) s')


eval3 :: Term -> ST Float                increase :: ST ()
eval3 (Con x)   = return x               increase = MakeST (\s -> ((), s+1))
eval3 (Div t u) = do x <- eval3 t
                     y <- eval3 u
                     increase
                     return (x/y)
```