Functions Operating on the \( \textbf{I\!\!O} \) Monad

\[
\text{putChar} :: \text{Char} \rightarrow \text{I\!\!O} () \\
\text{getChar} :: \text{I\!\!O} \text{Char}
\]

\[
(\gg) :: \text{I\!\!O} \text{ a} \rightarrow \text{I\!\!O} \text{ b} \rightarrow \text{I\!\!O} \text{ b}
\]

\[
\text{return} :: \text{a} \rightarrow \text{I\!\!O} \text{ a}
\]

\[
\begin{array}{c}
\text{IO}_1 \text{ a} \gg \text{IO}_2 \text{ b} = \text{IO}_1 \text{ I}_2 \text{ b}
\end{array}
\]

\[
(\gg=) :: \text{I\!\!O} \text{ a} \rightarrow (\text{a} \rightarrow \text{I\!\!O} \text{ b}) \rightarrow \text{I\!\!O} \text{ b}
\]

\[
\begin{array}{c}
\text{IO}_1 \text{ a} \gg= \text{a} \text{ IO}_2 \text{ b} = \text{IO}_1 \text{ a} \text{ a} \text{I}_2 \text{ b}
\end{array}
\]

getChar \( \gg \) return ( ) reads a character and ignores it
getChar \( \gg= \) 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)

\[ p >>= \x -> \quad \text{do } x \leftarrow p \]
\[ q >>= \y -> \quad \text{can be written as } \]
\[ r \]
\[ y \leftarrow q \]
\[ r \]

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
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)
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))
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)
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

```haskell
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
eval3 (Con x) = return x
eval3 (Div t u) = do x <- eval3 t
                      y <- eval3 u
                      increase
                      return (x/y)
```