m

```
{-
Klasa Monad jest w Haskellu deklarowana w następujący sposób:

class Monad m where
    >>=    :: m a -> (a -> m b) -> m b
    return :: a -> m a

i ewentualnie
    >>     :: m a -> m b -> m b
    ...

Ta ostatnia funkcja powinna byc zdefiniowana wzorem

    x >> e  =  x >>= \_ -> e

Kwestie typów: m to zmienna typowa, za którą można podstawic nazwę typu,
który wymaga jednego parametru, albo w innym języku, za m podstwiamy konstruktor typu
wymagającego jednego parametru (konstruktor typu w przeciwieństwie do konstruktora danych).

Operacje >>= i return powinny spełniac następujące równości:

    return a >>= f  = f a
    x >>= return    = x
    (x >>= f) >>= g = x >>= (\t -> f t >>= g)

Operatory >>= oraz >> łączą w lewo (tak, jak +) i mają pierwszeństwo 1, czyli większosc
operatorów wiąże silniej, z wyjątkiem m.in. operatora $, który ma pierwszeństwo 0.

-}
{-
Przykład: -}

type I a = a
instance Monad I where
   return x  = x
   x >>= f   = f x

Dla tego przykładu zachodzą wymagane prawa:

   return a >>= f = a >>= f = f a
   x >>= return   = return x = x
   x >>= (\t -> f t >>= g) = (\t -> f t >>= g) x = f x >>= g
   (x >>= f) >>= g = f x >>= g

Zauważmy też, że w tym przypadku

   x >> y = y
```

a więc w szczególności

```
   5 >> 7 = 7
-}

{-
Przykład: -}

data Id a = E a deriving Show
instance Monad Id where
   return x  = E x
   E x >>= f = f x

{- Dla tego przykładu zachodzą wymagane prawa:

   return a >>= f = E a >>= f = f a
   x >>= return   = E a >>= return  = return a = E a = x
   x >>= (\t -> f t >>= g) = E a >>= (\t -> f t >>= g) = (\t -> f t >>= g) a = f a >>= g
   (x >>= f) >>= g = (E a >>= f) >>= g = f a >>= g
```

Zauważmy też, że w tym przypadku

```
   x >> y = y
```

a więc w szczególności

```
   E 5 >> E 7 = E 7

-}
{-
```

Semantyka i składnia notacji do

Uproszczona składnia:

```
        do { stmts }
```

gdzie

```
        stmts -> exp | stmt; stmts
        stmt  -> exp | var <- exp | let decls
```

Uproszczona semantyka:

```
        do exp               = exp
        do exp;stmts         = exp >> do stmts
        do var <- exp;stmts = let f var = do stmts in exp >>= f
                             = exp >>= \var -> do stmts
```

```
                                                        m
        do let decls;stmts  = let decls in do stmts

W szczególności:

        do v1 <- e1; v2 <- e2; exp
                        = e1 >>= \v1 -> do v2 <- e2; exp
                        = e1 >>= \v1 -> e2 >>= \v2 -> exp
                        = e1 >>= (\v1 -> e2) >>= (\v2 -> exp)

W szczególności, w powyższym przykładzie:

        do v1 <- e1; v2 <- e2; exp
                        = exp[v2 <- e2[v1 <- e1]] (prawie, gdyż >>= to nie aplikacja
                                                   tylko aplikacja po zdjęciu E)


s,i :: Id Integer -> Id Integer -> Id Integer

s (E x) (E y) = E (x + y)
i (E x) (E y) = E (x * y)

ex1 = do a <- s (E 1) (E 3)
         b <- i (E a) (E 2)
         i (E 2) (s (return b) (i (return a) (E 7)))


ex2 = do x <- E 5
         (E.(+) 1) x {- = E 6 -}

ex3 = do E 5
         E 7          {- = E 7 -}

-}
{-
Typ wyrażeń z do.

W wyrażeniach do występują inne wyrażenia. O wyrażeniu exp mówimy, że występuje w
wyrażeniu do {stmts}, jeżeli exp jest jednym ze stwierdzeń w ciągu stmt lub
w ciągu stmts jest stwierdzenie postaci var <- exp.

W wyrażeniach postaci do exp wyrażenie exp może byc dowolne, wyrażenia
do exp i exp są tego samego typu (gdyż mają tę samą wartośc).

W poprawnie zbudowanych wyrażeniach do {stmts}, w których występuje wiele wyrażeń,
wszystkie wyrażenia muszą byc tego samego typu monadycznego, choc mogą byc to typy
z różnymi parametrami.
```

```
Typem wyrażenia do {stmts} jest typ ostatniego wyrażenia w ciągu stmts.

Własności do

do {e1;stmts} = do {_ <- e1; stmts}

Każde wyrażeniu do da się zapisac w postaci

do {v1 <- e1; v2 <- e2; ...; vn <- en; exp},

gdzie v1, ..., vn są zmiennymi lub _

do {v1 <- e1; v2 <- e2; ...; vn <- en; exp} =
   = e1 >>= (\v1 -> e2) >>= ... >>= (\vn -> exp).


-}


{-
Listy jako monady:

instance Monad [] where
    l >>= f  = concat (map f l)
    return x = [x]

    do x <- []; exp              = [] >>= \x -> exp  = concat (map (\x -> exp) []) = []
    do y <- e1; x <- []; exp     = e1 >>= \y -> []   = concat (map (\y -> []) e1)  =
                                 = concat [[], [], ...] = [] (dla e1 różnych od [])
    do x <- [a]; exp             = [a] >>= \x -> exp = concat (map (\x -> exp) [a])=
                                 = concat [exp[x <- a]] = exp[x <- a]
                                          (exp[x <- a) powinno byc listą)
    do y <- [b,c]; x <- [a]; exp = concat (map (\y -> exp[x<-a]) [b,c])=
                                 = concat [exp[x<-a][y<-b], exp[x<-a][y<-c]]=
                                 = exp[x<-a][y<-b] ++ exp[x<-a][y<-c]


-}


newtype StateComput s a = SC (s -> (a,s))


{-
```

```
return :: a -> StateComput s a
>>= :: (StateComput s a) -> (a -> StateComput s b) -> StateCompute s b


instance Monad (StateComput s) where
   return w     = SC (\s -> (w,s))
   (SC f) >>= g = SC (\s -> let (w1, s1) = f s
                                SC h = g w1
                                in h s1 )


   return w >>= f = SC (\s -> (w,s)) >>= f
                  = Sc (\t -> let (w1,s1) = (\s -> (w s)) t
                                  SC h = f w1
                                  in h s1)
                  = SC (\t -> let (w1, s1) = (w t)
                                  SC h = f w1
                                  in h s1)
                  = SC (\t -> let SC h = f w in h t)
                  = let SC h = f w in SC (\t -> h t)
                  = let SC h = f w in SC h
                  = f w

SC f >>= return = SC (\s -> let (w1, s1) = f s
                                SC h = return w1
                                in h s1 )
                = SC (\s -> let (w1, s1) = f s
                                SC h = SC (\s -> (w1,s))
                                in h s1 )
                = SC (\s -> let (w1, s1) = f s
                                h = \s -> (w1,s)
                                in h s1 )
                = SC (\s -> let (w1, s1) = f s in (w1, s1))
                = SC (\s -> f s)
                = SC f



SC h >>= (\t -> f t >>= g) = SC (\s -> let (w1, s1) = h s
                                       SC k = (\t -> f t >>= g) w1
                                       in k s1 )
                           = SC (\s -> let (w1, s1) = h s
                                       SC m = f w1
                                       SC k = (m >>= g)
                                       in k s1 )
                           = SC (\s -> let (w1, s1) = h s
```

```
                                    m
                          SC m = f w1
                          SC k = SC (\t -> let (w2, s2) = m t
                                              SC h = g w2
                                              in h s2 )
                          in k s1 )
                 = SC (\s -> let (w1, s1) = h s
                          SC m = f w1
                          k = (\t -> let (w2, s2) = m t
                                         SC h = g w2
                                         in h s2 )
                          in k s1 )
                 = SC (\s -> let (w1, s1) = h s
                          SC m = f w1
                          (w2, s2) = m s1
                          SC h = g w2
                          in h s2 )


((SC h) >>= f) >>= g = SC (\s -> let (w1, s1) = h s
                          SC k = f w1
                          in k s1 ) >>= g
                 = SC (\s -> let (w2, s2) = let (w1, s1) = h s
                                            SC k = f w1
                                            in k s1 )
                                 SC h = g w2
                                 in h s2 )
                 = SC (\s -> let (w1, s1) = h s
                                 SC k = f w1
                                 (w2, s2) = k s1
                                 SC h = g w2
                                 in h s2 )


   do e1; stmts = e1 >> do stmts
                = e1 >>= \_ -> do stmts
                = SC f1 >>= \_ -> do stmts
                = SC (\s -> let (w1, s1) = f1 s
                                SC h = do stmts
                                in h s1 )
                = let SC h = do stmts in SC (\s -> let (w1, s1) = f1 s in h s1 )
                = let SC h = do stmts; k = \s -> snd (f1 s) in SC (\s -> h.k s)
                = let SC h = do stmts; k = \s -> snd (f1 s) in SC (\s -> h.k s)
                = let SC h = do stmts; k = \s -> snd (f1 s) in SC h.k

-}
```