CS212. Парадигмы и технологии программирования: часть 1, функциональное программирование

Лекция 6. Монады

В. Н. Брагилевский

28 марта 2019 г.

Направление «Фундаментальная информатика и информационные технологии»
Институт математики, механики и компьютерных наук имени И. И. Воровича
Южный федеральный университет

Понятие вычислительного контекста

- `v :: a` — значение типа `a`
- `w :: C a` — значение типа `a` в контексте `C`
- `f :: a1 -> a2 -> ... -> C a` — функция, результатом которой является значение в контексте

Таким образом, контекст — это тип сорта `* -> *`.

Примеры вычислительных контекстов

- **`IO`** — вычисление с потенциально возможными побочными эффектами (пример: `getLine :: IO String`).
- **`Maybe`** — вычисление с возможной неудачей (пример: `Just 5` или **`Nothing`**).
- **`Either a`** — вычисление с возможной неудачей и сообщением об ошибке типа `a` (пример: **`Right 'x'`** или **`Left "Incorrect File Format"`**).
- **`[]`** — вычисление с недетерминированным результатом (пример: `[1,3,5]`).

Определение класса Functor

```haskell
class Functor (f :: * -> *) where
  fmap :: (a -> b) -> f a -> f b
```

- f — это вычислительный контекст.
- Функтор — преобразование значения с сохранением контекста.

Определение аппликативного функтора

```
class Functor f => Applicative f where
    pure :: a -> f a
    (<*>) :: f (a -> b) -> f a -> f b
```

- Две функции:
  - помещение значения в контекст;
  - преобразование значения в контексте с помощью функции, находящейся в контексте.

Проблема

```haskell
class Functor f => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

- Как ввести строку, а затем вывести её на консоль?
- Наблюдение: второе действие зависит от результата первого.
- Вывод: возможностей аппликативных функторов недостаточно.

Решение: класс типов Monad

```
class Applicative m => Monad (m :: * -> *) where
  (>>=)  :: m a -> (a -> m b) -> m b
  (>>)   :: m a -> m b -> m b
```

- Операция >>= называется монадическим связыванием (bind)
- (>>) игнорирует результат первого вычисления

Примеры

```
ghci> getLine >>= putStrLn
abc
abc
ghci> getLine >>= putStr >> putStrLn "!"
hello
hello!
```

Примеры

```
ghci> getLine >>= putStrLn
abc
abc
ghci> getLine >>= putStr >> putStrLn "!"
hello
hello!

ghci> Just 7 >>= (\x -> Just $ x+1)
Just 8
ghci> Nothing >>= (\x -> Just $ x+1)
Nothing
ghci> [1,2] >>= (\x -> [x-1, x+1])
[0,2,1,3]
```

Синтаксис монадических операций: do-блоки

Синтаксис монадических операций: do-блоки

```
do
  action1
  action2
```

```
action1 >> action2
```

Синтаксис монадических операций: do-блоки

```
do
  action1
  action2
```

```
action1 >> action2
```

```
do
  pat <- expr
  morelines
```

```
expr >>= (\pat -> do
                  morelines)
```

Синтаксис монадических операций: do-блоки

```
do
  action1
  action2
```

```
action1 >> action2
```

```
do
  pat <- expr
  morelines
```

```
expr >>= (\pat -> do
                  morelines)
```

- Например:

```
do
  x <- action
  process x
```

```
action >>= process
```

Синтаксис монадических операций: do-блоки

```
do
  action1
  action2
```

```
action1 >> action2
```

```
do
  pat <- expr
  morelines
```

```
expr >>= (\pat -> do
                  morelines)
```

- Например:

```
do
  x <- action
  process x
```

```
action >>= process
```

```
do
  x <- action
  pure x
```

```
action
```

Монада IO

- Область применения: ввод-вывод.
- Эффект: наличие побочных эффектов во время вычислений.

Монада IO

- Область применения: ввод-вывод.
- Эффект: наличие побочных эффектов во время вычислений.

Пример

Вычислить количество строк в файле, имя которого задано в параметрах командной строки.

Монада IO

- Область применения: ввод-вывод.
- Эффект: наличие побочных эффектов во время вычислений.

Пример

Вычислить количество строк в файле, имя которого задано в параметрах командной строки.

Решение

```
main = do
  fname <- fmap head getArgs
  content <- readFile fname
  print $ length $ lines content
```

Решение без do-блока

```
main = head <$> getArgs >>= readFile
                        >>= print.length.lines
```

Решение без do-блока

```
main = head <$> getArgs >>= readFile
                        >>= print.length.lines
```

Приоритеты (`:info`)

```
infixl 4 <$>
infixl 1 >>=
infixr 9 .
```

Решение с явными приоритетами

```
main = ((head <$> getArgs) >>= readFile)
                   >>= (print.(length.lines))
```

Maybe: экземпляр класса типов Monad

```haskell
instance Monad Maybe where
    pure x = Just x

    Nothing >>= f = Nothing
    Just x  >>= f = f x
```

- Область применения: вычисления с возможной неудачей.
- Эффект: не нужно проверять предыдущую операцию на неудачу.

Maybe: экземпляр класса типов Monad

```haskell
instance Monad Maybe where
    pure x = Just x

    Nothing >>= f = Nothing
    Just x  >>= f = f x
```

- Область применения: вычисления с возможной неудачей.
- Эффект: не нужно проверять предыдущую операцию на неудачу.

```
ghci> Just 9 >>= \x -> pure (x*10)
Just 90
ghci> Nothing >>= \x -> pure (x*10)
Nothing
```

Задача о канатоходце

Условие

Канатоходец передвигается по канату с помощью длинного шеста. На левый и правый концы шеста могут садиться птицы. Если в какой-то момент разница в количестве птиц на концах шеста оказывается больше трёх, канатоходец падает.

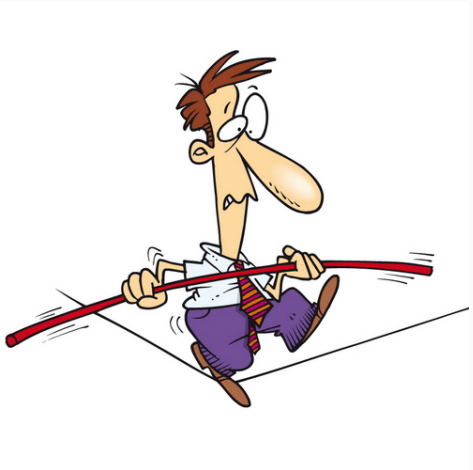
Задача о канатоходце

Условие

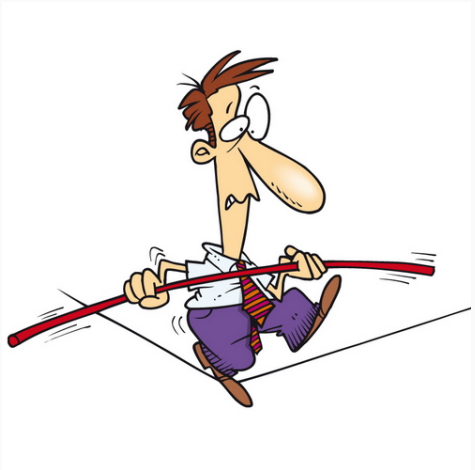
Канатоходец передвигается по канату с помощью длинного шеста. На левый и правый концы шеста могут садиться птицы. Если в какой-то момент разница в количестве птиц на концах шеста оказывается больше трёх, канатоходец падает.

- Вычисление — это учёт количества птиц на двух концах шеста.
- Ошибка возникает при отсутствии баланса в количестве птиц.

Пример

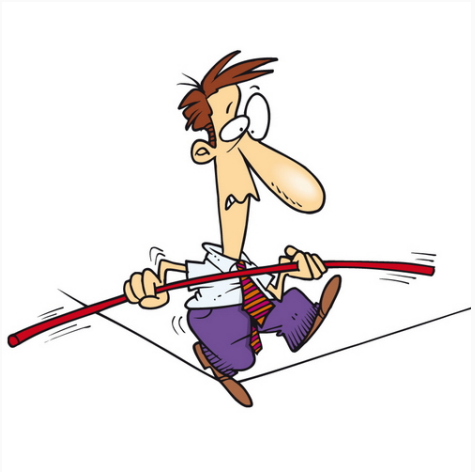


Пример

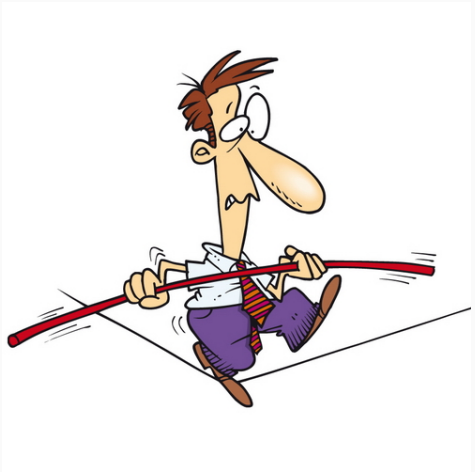


+1

Пример

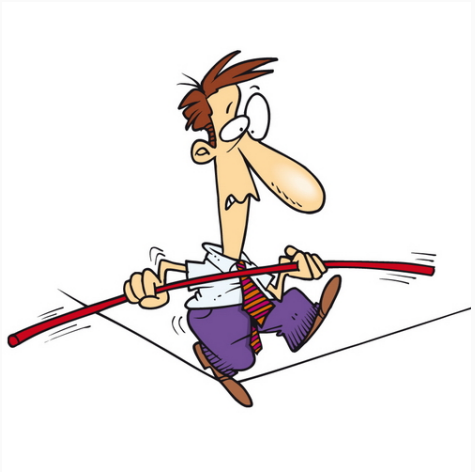


Пример

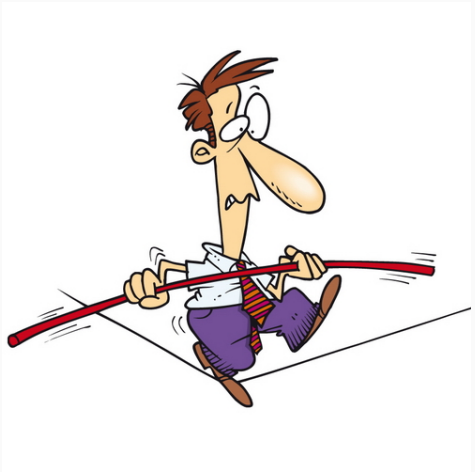


+4

Пример

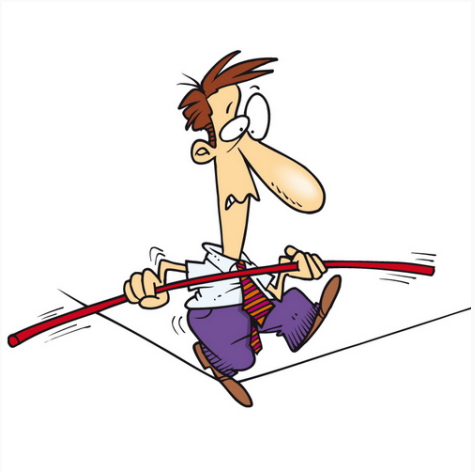


Пример

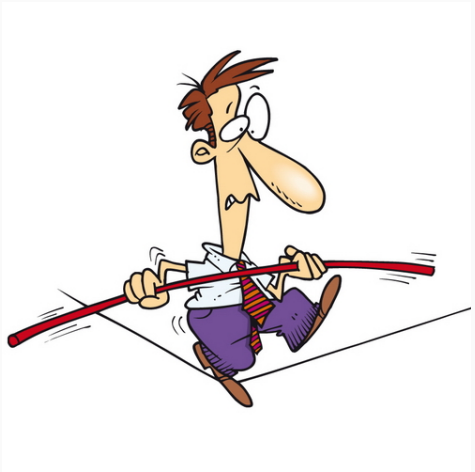


+2

Пример

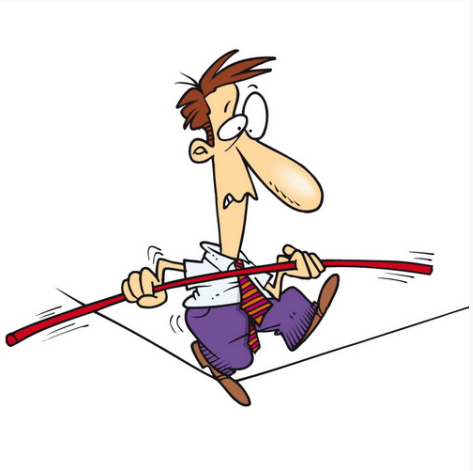


Пример



-3

Пример



Примеры вычислений

Вычисление 1

- Начальное значение: `(0,0)`
- +1 слева
- +4 справа
- +2 слева
- -3 справа
- Результат: `(3,1)`

Вычисление 2

- Начальное значение: `(0,0)`
- +1 слева
- +10 справа
- -8 справа
- Результат: падение

Типы данных и функции

```haskell
type Birds = Int
type Pole = (Birds, Birds)

balanceLimit = 3

checkBalance :: Pole -> Bool
checkBalance (l, r) = abs (l - r) <= balanceLimit
```

Типы данных и функции: учёт падений

Типы данных и функции: учёт падений

```haskell
setPole :: Pole -> Maybe Pole
setPole p
    | checkBalance p = Just p
    | otherwise = Nothing

landLeft :: Birds -> Pole -> Maybe Pole
landLeft n (left, right) =
   setPole (left + n, right)

landRight :: Birds -> Pole -> Maybe Pole
landRight n (left, right) =
    setPole (left, right + n)
```

Примеры использования

```
ghci> landLeft 2 (0, 0)
Just (2,0)
ghci> landLeft 10 (0, 3)
Nothing
```

Примеры использования

```
ghci> landLeft 2 (0, 0)
Just (2,0)
ghci> landLeft 10 (0, 3)
Nothing

ghci> landRight 1 (0, 0) >>= landLeft 2
Just (2,1)
ghci> Nothing >>= landLeft 2
Nothing
ghci> setPole (0, 0) >>= landRight 2 >>= landLeft 3
Just (3,2)
ghci> setPole (0, 0) >>= landLeft 10 >>= landRight 4
Nothing
```

Примеры вычислений

Вычисление 1

- Начальное значение: `(0,0)`
- +1 слева
- +4 справа
- +2 слева
- -3 справа
- Результат: `(3,1)`

Примеры вычислений

Вычисление 1

- Начальное значение: `(0,0)`
- +1 слева
- +4 справа
- +2 слева
- -3 справа
- Результат: `(3,1)`

```
ex1 = setPole (0,0)
      >>= landLeft 1
      >>= landRight 4
      >>= landLeft 2
      >>= landRight (-3)

ghci> ex1
Just (3,1)
```

Примеры вычислений

Вычисление 2

- Начальное значение: `(0,0)`
- +1 слева
- +10 справа
- -8 справа
- Результат: падение

Примеры вычислений

Вычисление 2

- Начальное значение: `(0,0)`
- +1 слева
- +10 справа
- -8 справа
- Результат: падение

```
ex2 = setPole (0,0)
      >>= landLeft 1
      >>= landRight 10
      >>= landRight (-8)

ghci> ex2
Nothing
```

Как это всё работает

```
(>>=)  :: m a -> (a -> m b) -> m b
(>>)   :: m a -> m b -> m b
pure :: a -> m a

setPole :: Pole -> Maybe Pole
landLeft :: Birds -> Pole -> Maybe Pole
landRight :: Birds -> Pole -> Maybe Pole
```

Как это всё работает

```
(>>=)  :: m a -> (a -> m b) -> m b
(>>)   :: m a -> m b -> m b
pure :: a -> m a

setPole :: Pole -> Maybe Pole
landLeft :: Birds -> Pole -> Maybe Pole
landRight :: Birds -> Pole -> Maybe Pole
```

Как учитывается возможность падения?

Как это всё работает

```
(>>=)  :: m a -> (a -> m b) -> m b
(>>)   :: m a -> m b -> m b
pure :: a -> m a
```

```
setPole :: Pole -> Maybe Pole
landLeft :: Birds -> Pole -> Maybe Pole
landRight :: Birds -> Pole -> Maybe Pole
```

Как учитывается возможность падения?

- `setPole` при нарушении баланса помещает в монаду **`Nothing`**
- Определение `(>>=)` для **`Maybe`** гарантирует, что **`Nothing`** останется до конца вычислений.

Недостатки do-блоков

Решение с монадическим связыванием

```
ex1 = setPole (0,0)
      >>= landLeft 1
      >>= landRight 4
      >>= landLeft 2
      >>= landRight (-3)
```

Недостатки do-блоков

Решение с монадическим связыванием

```
ex1 = setPole (0,0)
      >>= landLeft 1
      >>= landRight 4
      >>= landLeft 2
      >>= landRight (-3)
```

Решение с использованием do-блока

```
ex1' = do
  p1 <- landLeft 1 (0,0)
  p2 <- landRight 4 p1
  p3 <- landLeft 2 p2
  landRight (-3) p3
```

Решение без использования монадических операций

```
ex1'' =
  case landLeft 1 (0,0) of
     Nothing -> Nothing
     Just p1 -> case landRight 4 p1 of
           Nothing -> Nothing
           Just p2 -> case landLeft 2 p2 of
                 Nothing -> Nothing
                 Just p3 -> landRight (-3) p3
```

```
ghci> ex1
Just (3,1)
ghci> ex1'
Just (3,1)
ghci> ex1''
Just (3,1)
```

Задача о канатоходце: случай неизвестного числа ходов

Задача о канатоходце: случай неизвестного числа ходов

Команда и её обработка

```
data Side = SLeft | SRight
data Command = Cmd Side Birds


processCmd :: Pole -> Command -> Maybe Pole
processCmd p (Cmd SLeft n) = landLeft n p
processCmd p (Cmd SRight n) = landRight n p
```

Задача о канатоходце: случай неизвестного числа ходов

Команда и её обработка

```
data Side = SLeft | SRight
data Command = Cmd Side Birds

processCmd :: Pole -> Command -> Maybe Pole
processCmd p (Cmd SLeft n) = landLeft n p
processCmd p (Cmd SRight n) = landRight n p
```

```
ex1''' = foldM processCmd (0,0)
                  [Cmd SLeft 1, Cmd SRight 4,
                   Cmd SLeft 2, Cmd SRight (-3)]
```

Задача о канатоходце: случай неизвестного числа ходов

Команда и её обработка

```haskell
data Side = SLeft | SRight
data Command = Cmd Side Birds

processCmd :: Pole -> Command -> Maybe Pole
processCmd p (Cmd SLeft n) = landLeft n p
processCmd p (Cmd SRight n) = landRight n p
```

```haskell
ex1''' = foldM processCmd (0,0)
                [Cmd SLeft 1, Cmd SRight 4,
                 Cmd SLeft 2, Cmd SRight (-3)]

foldM :: (Foldable t, Monad m) =>
         (b -> a -> m b) -> b -> t a -> m b
```

Список как монада

```haskell
instance Monad [] where
    (>>=) :: [a] -> (a -> [b]) -> [b]
    xs >>= f = concat (map f xs)
```

- Недетерминированные вычисления — функция (возвращающая список) применяется к каждому элементу исходного списка.

Список как монада

```haskell
instance Monad [] where
    (>>=) :: [a] -> (a -> [b]) -> [b]
    xs >>= f = concat (map f xs)
```

- Недетерминированные вычисления — функция (возвращающая список) применяется к каждому элементу исходного списка.

```
ghci> [3,4,5] >>= \x -> [x,-x]
[3,-3,4,-4,5,-5]
ghci> [1,2] >>= \n -> ['a','b']
            >>= \ch -> pure (n,ch)
[(1,'a'),(1,'b'),(2,'a'),(2,'b')]
```

Три нотации для списков

```
ghci> [1,2] >>= \n -> ['a','b']
              >>= \ch -> pure (n,ch)
[(1,'a'),(1,'b'),(2,'a'),(2,'b')]
```

Три нотации для списков

```
ghci> [1,2] >>= \n -> ['a','b']
              >>= \ch -> pure (n,ch)
[(1,'a'),(1,'b'),(2,'a'),(2,'b')]
```

```
listOfTuples :: [(Int,Char)]
listOfTuples = do
    n <- [1,2]
    ch <- ['a','b']
    pure (n,ch)
```

```
ghci> listOfTuples
[(1,'a'),(1,'b'),(2,'a'),(2,'b')]
```

Три нотации для списков

```
ghci> [1,2] >>= \n -> ['a','b']
              >>= \ch -> pure (n,ch)
[(1,'a'),(1,'b'),(2,'a'),(2,'b')]
```

```
listOfTuples :: [(Int,Char)]
listOfTuples = do
    n <- [1,2]
    ch <- ['a','b']
    pure (n,ch)
```

```
ghci> listOfTuples
[(1,'a'),(1,'b'),(2,'a'),(2,'b')]
```

```
ghci> [(n,ch) | n <- [1,2], ch <- ['a','b']]
[(1,'a'),(1,'b'),(2,'a'),(2,'b')]
```

Монады и моноиды: классы Alternative и MonadPlus

```haskell
class Applicative f => Alternative f where
  empty :: f a
  (<|>) :: f a -> f a -> f a

class Alternative m, Monad m => MonadPlus m where
  mzero :: m a
  mplus :: m a -> m a -> m a
```

Монады и моноиды: классы Alternative и MonadPlus

```haskell
class Applicative f => Alternative f where
  empty :: f a
  (<|>) :: f a -> f a -> f a

class Alternative m, Monad m => MonadPlus m where
  mzero :: m a
  mplus :: m a -> m a -> m a
```

Функция guard (Control.Monad)

```haskell
guard :: Alternative f => Bool -> f ()
guard True  = pure ()
guard False = empty
```

Чем guard отличается от when?

```haskell
guard :: Alternative f => Bool -> f ()
guard True  = pure ()
guard False = empty

when :: Applicative f => Bool -> f () -> f ()
when True  s  = s
when False s  = pure ()
```

Примеры использования функции guard

```
ghci> guard True :: Maybe ()
Just ()
ghci> guard False :: Maybe ()
Nothing
ghci> guard True :: [()]
[()]
ghci> guard False :: [()]
[]
ghci> guard True >> pure "good" :: [String]
["good"]
ghci> guard False >> pure "good" :: [String]
[]
```

Реализация Alternative для Maybe и списков

```haskell
instance Alternative Maybe where
  empty = Nothing

  Nothing <|> r = r
  Just x  <|> _ = Just x
```

Реализация Alternative для Maybe и списков

```haskell
instance Alternative Maybe where
  empty = Nothing

  Nothing <|> r = r
  Just x  <|> _ = Just x

instance Alternative [] where
  empty = []
  (<|>) = (++)
```

Три нотации для списков: guard

```
ghci> [1..50] >>= (\x ->
        guard ('7' `elem` show x) >> pure x)
[7,17,27,37,47]
```

Три нотации для списков: guard

```
ghci> [1..50] >>= (\x ->
        guard ('7' `elem` show x) >> pure x)
[7,17,27,37,47]
```

```haskell
withSevensOnly :: [Int]
withSevensOnly = do
    x <- [1..50]
    guard ('7' `elem` show x)
    pure x
```

Три нотации для списков: guard

```
ghci> [1..50] >>= (\x ->
        guard ('7' `elem` show x) >> pure x)
[7,17,27,37,47]
```

```haskell
withSevensOnly :: [Int]
withSevensOnly = do
    x <- [1..50]
    guard ('7' `elem` show x)
    pure x
```

```
ghci> [x | x <- [1..50], '7' `elem` show x]
[7,17,27,37,47]
```

Задача о перемещении коня

Условие

Определить, может ли конь перейти из одной позиции в другую за заданное число ходов?

Задача о перемещении коня

Условие

Определить, может ли конь перейти из одной позиции в другую за заданное число ходов?

```haskell
type KnightPos = (Int, Int)

moveKnight :: KnightPos -> [KnightPos]
moveKnight (c,r) = do
    (c',r') <- [(c+2,r-1),(c+2,r+1),
                (c-2,r-1),(c-2,r+1),
                (c+1,r-2),(c+1,r+2),
                (c-1,r-2),(c-1,r+2)]
    guard (c' `elem` [1..8] && r' `elem` [1..8])
    pure (c',r')
```

Пример вычисления moveKnight

```
ghci> moveKnight (1,1)

[(3,0), (3,2),  (-1,0),  (-1,2),  (2,-1),  (2,3),  (0,-1)...

F       T       F        F        F        T       F...
[]      [()]    []       []       []       [()]    []...

[(3,0)] [(3,2)] [(-1,0)] [(-1,2)] [(2,-1)] [(2,3)] [(0,-1)]...

[]      [(3,2)] []       []       []       [(2,3)] []...
```

Упрощённая задача: три хода

Упрощённая задача: три хода

```haskell
in3 start = do
   first <- moveKnight start
   second <- moveKnight first
   moveKnight second

in3 start = moveKnight start
            >>= moveKnight
            >>= moveKnight
```

Упрощённая задача: три хода

```
in3 start = do
   first <- moveKnight start
   second <- moveKnight first
   moveKnight second

in3 start = moveKnight start
            >>= moveKnight
            >>= moveKnight
```

- Здесь гнездо из трёх циклов!

Упрощённая задача: три хода

```haskell
in3 start = do
    first <- moveKnight start
    second <- moveKnight first
    moveKnight second

in3 start = moveKnight start
            >>= moveKnight
            >>= moveKnight
```

- Здесь гнездо из трёх циклов!

Решение

```haskell
canReachIn3 :: KnightPos -> KnightPos -> Bool
canReachIn3 start end = end `elem` in3 start
```

Решение задачи о коне

Решение задачи о коне

```haskell
inMany :: Int -> KnightPos -> [KnightPos]
inMany n st = foldr (<=<)
                    pure
                    (replicate n moveKnight) st

canReachIn :: Int -> KnightPos -> KnightPos -> Bool
canReachIn n start end = end `elem` inMany n start
```

- Здесь гнездо из *n* циклов!

Решение задачи о коне

```
inMany :: Int -> KnightPos -> [KnightPos]
inMany n st = foldr (<=<)
                    pure
                    (replicate n moveKnight) st

canReachIn :: Int -> KnightPos -> KnightPos -> Bool
canReachIn n start end = end `elem` inMany n start
```

- Здесь гнездо из *n* циклов!

```
foldr :: Foldable t => (a -> b -> b) -> b -> t a -> b
(<=<) :: Monad m => (b -> m c) -> (a -> m b) -> a -> m c
pure :: a -> m a
replicate :: Int -> a -> [a]
moveKnight :: KnightPos -> [KnightPos]
```

Монады: выхода нет

```
fmap :: (a -> b) -> f a -> f b

pure :: a -> f a
(<*>) :: f (a -> b) -> f a -> f b

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

- Поместить в монаду значение можно, а извлечь нельзя.
- Для конкретных монад можно написать соответствующую функцию (`fromJust`).

- **`Functor`** — изменение значения в контексте без изменения контекста.
- **`Applicative`** — применение функции в контексте к значению в контексте.
- **`Monad`** — зависимость вычисления в контексте от результата предыдущего вычисления в контексте.