Хотим расширить чистые функции (`a -> b`) до вычислений с «эффектом», которые

- иногда могут завершиться неудачей: `a -> Maybe b`
- могут возвращать много результатов: `a -> [b]`
- иногда могут завершиться ошибкой: `a -> (Either s) b`
- могут делать записи в лог: `a -> (s,b)`
- могут читать из внешнего окружения: `a -> ((->) e) b`
- работают с мутабельным состоянием: `a -> (State s) b`
- делают ввод/вывод (файлы, консоль): `a -> IO b`
- не делают ничего: `a -> Identity b`

Обобщая, получим *стрелку Клейсли*: `a -> m b`
Стрелка Клейсли обеспечивает зависимость *эффекта* от *значения*. Например, `\n -> replicate n 'A'`.

Какими должны быть требования к оператору над типами `m` в стрелке Клейсли `a -> m b`?

- Должен иметься универсальный интерфейс для упаковки значения в контейнер `m`.
- Должен иметься универсальный интерфейс для композиции вычислений с эффектом (стрелок Клейсли):

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

Используя только интерфейс функтора не реализовать:

```
k1 <=< k2 = \x -> k1 <$> k2 x -- увы :: m (m c)
```

- Нет универсального интерфейса для извлечения значения из контейнера `m`.
  (Эффект в общем случае нельзя отбросить.)

Если бы миром правили теоретики, ...

... то класс типов Monad был бы определён так

```
class Pointed m => Monad m where
  join :: m (m a) -> m a
```

Но вычислительно удобнее так

```
infixl 1 >>, >>=
class Applicative m => Monad m where
  (>>=) :: m a -> (a -> m b) -> m b -- произносят bind

  (>>) :: m a -> m b -> m b
  m1 >> m2 = m1 >>= \_ -> m2

  return :: a -> m a
  return = pure
```

`return :: a -> m a` определяет тривиальную стрелку Клейсли. Функция `pure :: a -> f a` из `Applicative` — полный её аналог.
Позволяет превратить `f :: a -> b` в стрелку Клейсли:

```
toKleisli :: Monad m => (a -> b) -> (a -> m b)
toKleisli f = return . f
```

```
GHCi> :type toKleisli cos
toKleisli cos :: (Monad m, Floating b) => b -> m b
GHCi> (toKleisli cos 0) :: Maybe Double
Just 1.0
GHCi> (toKleisli cos 0) :: [Double]
[1.0]
GHCi> (toKleisli cos 0) :: IO Double
1.0
```

Имеется обратный «связыватель» `(=<<) = flip (>>=)`, похожий на знакомые операции

```
($)     ::                        (a -> b) ->   a ->   b
(<$>)   :: Functor f =>           (a -> b) -> f a -> f b
(<*>)   :: Applicative f =>  f (a -> b) -> f a -> f b
(=<<)   :: Monad m =>           (a -> m b) -> m a -> m b
```

Прямой «связыватель» `(>>=) :: m a -> (a -> m b) -> m b` похож на их «флипы» (NB: для `<*>` не «флип»!)

```
(&)     ::                          a ->   (a -> b) ->   b
(<&>)   :: Functor f =>          f a ->   (a -> b) -> f b
(<**>)  :: Applicative f =>  f a -> f (a -> b) -> f b
(>>=)   :: Monad m =>           m a -> (a -> m b) -> m b
```

Для любого представителя Monad должны выполняться законы

```
return a >>= k     ≡  k a

m >>= return       ≡  m

(m >>= k) >>= k'   ≡  m >>= (\x -> k x >>= k')
```

Первые два закона выражают тривиальную природу return

```
GHCi> runIdentity $ wrap'n'succ 3
4
GHCi> runIdentity $ return 3 >>= wrap'n'succ
4
GHCi> runIdentity $ wrap'n'succ 3 >>= return
4
```

Третий закон класса типов Monad

```
(m >>= k) >>= k'   ≡  m >>= (\x -> k x >>= k')

 m >>= k >>= k'    ≡  m >>= \x -> k x >>= k'
```

задаёт некоторое подобие ассоциативности

```
GHCi> wrap'n'succ 3 >>= wrap'n'succ >>= wrap'n'succ
Identity 6
GHCi> wrap'n'succ 3 >>= \x -> wrap'n'succ x >>= wrap'n'succ
Identity 6
```

Прицепим `return` (можно в силу второго закона), и применим третий закон ко всем связываниям `(>>=)`

```
goWrap0 = wrap'n'succ 3 >>=
          wrap'n'succ >>=
          wrap'n'succ >>=
          return
goWrap1 = wrap'n'succ 3 >>= (\x ->
          wrap'n'succ x >>= (\y ->
          wrap'n'succ y >>= (\z ->
          return z)))
```

```
GHCi> runIdentity goWrap0
6
GHCi> runIdentity goWrap1
6
```

```
goWrap1 = wrap'n'succ 3 >>= (\x ->
          wrap'n'succ x >>= (\y ->
          wrap'n'succ y >>= (\z ->
          return z)))

goWrap2 = wrap'n'succ 3 >>= (\x ->
          wrap'n'succ x >>= (\y ->
          wrap'n'succ y >>= (\z ->
          return (x,y,z))))
```

```
GHCi> runIdentity goWrap1
6
GHCi> runIdentity goWrap2
(4,5,6)
```

Ой, мы изобрели императивное программирование!

Можем использовать `let`-связывание для обычных выражений:

```
goWrap3 = let i = 3 in
          wrap'n'succ i >>= (\x ->
          wrap'n'succ x >>= (\y ->
          wrap'n'succ y >>=  \z ->
          return (i,x,y,z)))
```

```
GHCi> runIdentity goWrap3
(3,4,5,6)
```

Если результат не интересен, можно его игнорировать, используя усеченный связыватель >>:

```
goWrap4 = let i = 3 in
          wrap'n'succ i >>= (\x ->
          wrap'n'succ x >>= (\y ->
          wrap'n'succ y >>
          return (i,x,y)))
```

```
GHCi> runIdentity goWrap4
(3,4,5)
```

Правила трансляции в Haskell Kernel для do-нотации

`do {e}`	≡	`e`
`do {e; stmts}`	≡	`e >> do {stmts}`
`do {p <- e; stmts}`	≡	`e >>= \p -> do {stmts}`
`do {let v = exp; stmts}`	≡	`let v = exp in do {stmts}`

Здесь все `e :: m a`.

```haskell
goWrap4 =    let i = 3 in -- выравнивание для красоты
             wrap'n'succ i >>= (\x ->
             wrap'n'succ x >>= (\y ->
             wrap'n'succ y >>
             return (i,x,y)))
goWrap5 = do let i = 3    -- выравнивание обязательно
             x <- wrap'n'succ i
             y <- wrap'n'succ x
             wrap'n'succ y
             return (i,x,y)
```