

Трансформеры монад

В функциональном программировании преобразователь монад — это конструктор типов, который принимает монаду в качестве аргумента и возвращает монаду в результате.

Преобразователи монад можно использовать для составления функций, инкапсулированных в монады, таких как состояние, обработка исключений и ввод/вывод, модульным способом. Как правило, монадный трансформер создаётся путём обобщения существующей монады. Более того, в современном Haskell «классические» монады получаются применением монадных трансформеров к единичной монаде `Identity`.

Монада `Identity`

Иногда эту тривиальную монаду рассматривают как простое упражнение по теме монад. Но в настоящее время, видимо, это последняя «оставшаяся в живых» монада (наряду с монадой `IO`), т.е., которая работает не через соответствующие трансформеры. Правда, стоит отметить, что также существует тривиальный монадный трансформер `IdentityT`.

Данная монада обеспечивает простое применение функции. Стратегия связывания заключается в передаче аргумента к задействованной функции, можно даже сказать, что фактически нет особой вычислительной стратегии. Есть только применение связанной функции к своему входу без каких-либо изменений. С вычислительной точки зрения нет причин использовать монаду `Identity` вместо гораздо более простого применения функций к своим аргументам. Целью монады `Identity` является её фундаментальная роль в теории монадных трансформеров (преобразователей). Любой монадный трансформер, применённый к монаде `Identity`, даст простую (нетрансформерную) версию соответствующей монады.

```
newtype Identity a = Identity { runIdentity :: a }
```

```
instance Monad Identity where
  return x = Identity x      -- i.e. return = id
  (Identity x) >>= f = f x   -- i.e. x »= f = f x
```

Функция `fail` здесь организована наподобие обычной функции `error`:

```
fail :: String -> m a
```

Деструктор `runIdentity` (в других терминах — метка поля) используется в определении типа, потому что таков стиль определения монады, который явно представляет значения монады как вычисления. В этом стиле монадное вычисление создается с использованием монадических операторов, а затем значение вычисления извлекается с использованием функции `run***`. Поскольку монада `Identity` не выполняет никаких вычислений, её определение тривиально.

Типичное использование монады `Identity` — получение монад из трансформеров монад:

```
type Maybe a = MaybeT Identity a
type State s a = StateT s Identity a
type Writer w a = WriterT w Identity a
```

Полиморфный тип (конструктор типов) `Identity` находится в модуле [Data.Functor.Identity](#), поэтому реально, определения выше выглядят сложнее (результаты применения команды `:i State` и т.п.):

```
type Writer w = WriterT w Data.Functor.Identity.Identity :: * -> *
type State s = StateT s Data.Functor.Identity.Identity :: * -> *
```

[The Identity monad \(in All About Monads\)](#)

«Анатомия» трансформеров

Поддержку монад и монадных трансформеров осуществляют два базовых пакета: [mtl](#) и [transformers](#). Применение модулей из пакета мы увидим на практике далее.

Так как понимание и применение трансформеров довольно сложно, в данной лекции ограничимся примерами их необходимости и использования.

Вспомним задачу из 14-й лекции, когда нам был необходим пример комбинации монадных вычислений в монаде `Maybe`:

```
f :: Int -> Maybe Int
f x = if x `mod` 2 == 0
      then Nothing
      else Just (2*x)
```

```
g :: Int -> Maybe Int
g x = if x `mod` 3 == 0
      then Nothing
      else Just (3*x)
```

```
h :: Int -> Maybe Int
h x = if x `mod` 5 == 0
      then Nothing
      else Just (5*x)
```

```
k :: Int -> Maybe Int
k x = do y <- f x
        z <- g y
        h z
```

Для начала перепишем пример в различных стилях монадных записей:

```
import Control.Monad

f :: Int -> Maybe Int
f x = do
  if x `mod` 2 == 0
  then Nothing
  else return (2*x)
```

```

g :: Int -> Maybe Int
g x = do
    guard (x `mod` 3 /= 0)
    return (3*x)

h :: Int -> Maybe Int
h x = do
    when (x `mod` 5 == 0) Nothing
    return (5*x)

```

Функции **guard** и **when** в данном случае ведут себя практически одинаковым способом. Хотя можно считать основной запись для **f**.

Композицию **k** мы могли бы переписать и таким образом:

```
k = f >=> g >=> h
```

Но нам будет удобнее оставить её как есть для доступа к промежуточным переменным.

```

k :: Int -> Maybe Int
k x = do y <- f x
        z <- g y
        h z

```

Напомним, запуск дает примерно такие результаты:

```

*Main> k 5
Nothing
*Main> k 6
Nothing
*Main> k 7
Just 210

```

Теперь, допустим, мы хотим добавить функциональность из другого типа монадных вычислений, например, у нас есть функция

```

t :: Int -> [Int]
t x = [x, (x+1), (x+2)]

```

которая является неоднозначной, т.е. возвращает вместо одного значения — список значений, и мы хотели бы объединить такие разные функции в одном вычислении. Например, чтобы вместо трёх предыдущих отдельных запусков, мы в одном вычислении сразу бы запустили 3 ветви вычислений.

Все Haskell-пособия на тему монадных трансформеров рекомендуют включать необходимую функциональность с помощью функции **lift**:

```

k2 x = do
    u <- lift $ t x
    y <- f u

```

```
z <- g y
h z
```

Но, к сожалению, этот пример не пройдёт проверку типов. Лифтинг осуществляется из нижележащей монады (у нас это список `[Int]`), но не в монаду, а в монадный трансформер.

Таким образом, функции `f, g, h` должны иметь иной тип, возвращая не монадные значения как ранее, а типы монадных трансформеров. В этом примере это можно сделать «по месту», например так:

```
k2 :: Int -> MaybeT [] Int
k2 x = do
    u <- lift $ t x
    y <- MaybeT $ return $ f u
    z <- MaybeT $ return $ g y
    MaybeT $ return $ h z
```

```
rk2 :: Int -> [Maybe Int]
rk2 = runMaybeT . k2
```

Для преобразования функций `f, g, h` к типу, возвращающему монадный трансформер, я использовал статью:

[Haskell/Monad transformers](#)

Где как раз описано, как выглядит `return` в версии монадных трансформеров для `MaybeT`:

```
instance Monad m => Monad (MaybeT m) where
    return = MaybeT . return . Just
```

ну или даже

```
MaybeT . return . return
```

Наши функции `f, g, h` уже давали результат, обернутый тэгом `Just`, поэтому достаточно было добавить ещё две обёртки:

```
MaybeT $ return
```

где `return` оборачивал монадой, которую мы хотели бы добавить лифтингом (т.е. нижележащая монада), а конструктор `MaybeT` как раз бы сообщал, что итоговый результат принадлежит типу монадного трансформера `MaybeT`.

Этот вариант работает, но только как все трансформеры, требует особую «функцию-запускалку» `runMaybeT`, которая в композиции с `k2` и даёт то, что делала ранее `k` самостоятельно:

```
*Main> rk2 5
[Nothing, Nothing, Just 210]
```

Вот мы и получили три ветви вычисления.

Первая попытка добавить функционал

Добавим ещё функционал. Например, нам хочется вести журнал (лог) вычислений. Надо добавить возможности монады `Write`. У нас получится луковица: нижняя монада `[]`, потом средняя монада `Write`, потом монада (трансформер) **Maybe**. Кстати, с точки зрения вычисления, всё будет в точности иначе: внутренняя монада будет **Maybe** (ведь и в прошлый раз все **Nothing** и **Just...** были внутри списка), потом будет слой, обеспечивающий журналирование, потом будет слой, всё обволакивающий списком.

Одно из правил гласит, что эти эффекты интуитивно помещаются в стек монад в порядке, обратном тому, в котором они появляются в луковице преобразователя. (Алехандро Мена, с.219)

К сожалению, опять проблема. Даже если мы быстро разберёмся, на какой слой помещать списки и журналирование, напр.:

```
k3 x = do
    lift $ tell $ "x:␣" ++ (show x) ++ ";␣"
    u <- lift $ lift $ t x
    lift $ tell $ "u:␣" ++ (show u) ++ ";␣"
    y <- MaybeT $ return $ f u
    ...
```

Данный код вновь не пройдёт проверку типов. И опять проблема в типе наших функций `f, g, h`. Опять надо усложнять упаковки по числу вложенных слоёв. Это утомительно и ведёт к трудным ошибкам.

Делаем трансформеры правильно

Поэтому, поступим в этот раз иначе. Давайте немного переделаем исходные функции `f, g, h` так, чтобы их возвращаемый результат был действительно как трансформер, не зависящий от промежуточных монад и не требующий дальнейших изменений.

Данную идею мне не удалось обнаружить нигде в мануалах и сайтах, фактически, это собственный способ адекватной работы с функциями-трансформерами. Сама идея возникла из рассуждений над инвариантностью монадных утилит, которые заведомо обладали этим свойством. Таким образом, нужно было либо изучить «шестерёнки и внутренности» таких утилит (вроде `get`, `tell` и т.п.), либо догадаться, как это делать. Мне кажется, я смог догадаться... :)

Теперь, [глянем ещё раз на переписанную форму](#) для функций `f, g, h` и сделаем из них трансформерные функции (несколько мистичным оказывается то, что определение сработало для функции `guard` без явного указания ветки для **Nothing**):

```
import Control.Monad
import Control.Monad.Trans.Maybe
import Control.Monad.Trans
import Control.Monad.Writer

f :: Monad m => Int -> MaybeT m Int
f x = do
    if x `mod` 2 == 0
```

```

    then MaybeT $ return Nothing
    else MaybeT $ return $ Just (2*x)

g :: Monad m => Int -> MaybeT m Int
g x = do
    guard (x `mod` 3 /= 0)
    MaybeT $ return $ return (3*x)

h :: Monad m => Int -> MaybeT m Int
h x = do
    when (x `mod` 5 == 0) (MaybeT $ return Nothing)
    MaybeT $ return $ Just (5*x)

```

Оставим специально эти разные определения для лучшего понимания нового механизма.

Теперь, перепишем определение для функции k, оно не изменилось (разве, лишь сигнатура):

```

k1 :: Monad m => Int -> MaybeT m Int
k1 x = do y <- f x
          z <- g y
          h z

```

сигнатура нам показывает, как и для функций f, g, h выше, что теперь это трансформные функции, готовые к сопряжению с другими монадами. В текущей сигнатуре эти «любые монады» обозначены переменной типа m.

А чтобы превратить эту функцию в нам знакомую k, необходимо сопрячь k1 с монадой Identity.

Но сначала, попрактикуемся на исходных относительно простых функциях f, g, h. Добавим в заголовок:

```
import Control.Monad.Identity
```

и попробуем так:

```

f' :: Int -> MaybeT Identity Int
f' x = do
    x' <- f x
    return x'

```

или даже так:

```

f'' :: Int -> MaybeT Identity Int
f'' = f >=> return

```

Мы только что сопрягли трансформенную функцию f с монадой Identity, о которой говорили выше. Но этого мало, так как функции выдают ещё не тот ответ, который мы желали бы:

```

*Main> f'' 1
MaybeT (Identity (Just 2))

```

```

*Main> f' 1
MaybeT (Identity (Just 2))

```

Для нормального функционирования необходимо скомбинировать две подряд «запускалки»:

```
rf' :: Int -> Maybe Int
rf' x = runIdentity $ runMaybeT $ f' x
```

или так

```
rf'' :: Int -> Maybe Int
rf'' = runIdentity . runMaybeT . f''
```

и эти функции уже возвращают нужный нам результат:

```
*Main> rf' 2
Nothing
*Main> rf' 3
Just 6
```

Теперь, так же превращаем k1 в аналог «старой знакомой», функции k:

```
rk1 :: Int -> Maybe Int
rk1 = runIdentity . runMaybeT . k1
```

с ожидаемым результатом:

```
*Main> rk1 7
Just 210
*Main> rk1 6
Nothing
```

Добавляем функционал

[Вернёмся к функции k2 из раздела выше](#), и перепишем теперь её правильно:

```
t :: Int -> [Int]
t x = [x, (x+1), (x+2)]

k22 :: Int -> MaybeT [] Int
k22 x = do
    u <- lift $ t x
    y <- f u
    z <- g y
    h z
```

Но чтобы превратить эту функцию в нам знакомую k, необходимо осуществить запуск функции-трансформера, как уже было:

```
rk22 :: Int -> [Maybe Int]
rk22 = runMaybeT . k22
```

И результаты для rk22 как и для функции k и rk2:

```
*Main> rk22 7
[Just 210,Nothing,Nothing]
*Main> rk22 8
```

```
[Nothing,Nothing,Nothing]
*Main> rk22 9
[Nothing,Nothing,Just 330]
*Main> rk22 11
[Just 330,Nothing,Just 390]
```

Теперь мы будем в состоянии добавлять ещё больше функционала! Вернёмся к добавлению логов:

```
k3 :: Int -> MaybeT (WriterT String []) Int
k3 x = do
    lift $ tell $ "x:␣" ++ (show x) ++ ";␣"
    u <- lift $ lift $ t x
    lift $ tell $ "u:␣" ++ (show u) ++ ";␣"
    y <- f u
    lift $ tell $ "y:␣" ++ (show y) ++ ";␣"
    z <- g y
    lift $ tell $ "z:␣" ++ (show z) ++ ";␣"
    j <- h z
    lift $ tell $ "j:␣" ++ (show j) ++ ";␣"
    return j
```

```
rk3 :: Int -> [(Maybe Int, String)]
rk3 = runWriterT . runMaybeT . k3
```

С соответствующим выводом в GHCi:

```
*Main> rk3 5
[(Nothing, "x:␣5;␣u:␣5;␣y:␣10;␣z:␣30;␣"),
 (Nothing, "x:␣5;␣u:␣6;␣"),
 (Just 210, "x:␣5;␣u:␣7;␣y:␣14;␣z:␣42;␣j:␣210;␣")]
```

Хорошо видно, какие ветви вычисления остановились и на каком шаге.

Возвращаемый тип `MaybeT (WriterT String []) Int` очень сложен, фактически здесь компилятор нам плохо помогает с подсказками. Так, например, я не смог сделать вариант, где `Writer` будет внизу, а список посередине. Компилятор сам просил подсказку, но я не смог ни экспериментально, ни теоретически понять требуемый тип.

Нужно запомнить одно очень важное обстоятельство: монадические преобразователи не обладают коммутативностью. То есть вычислительный эффект результирующей монады зависит от всего остального, что есть в преобразованиях. Например, монада `StateT s []` может представлять такие недетерминированные вычисления, где каждый из путей имеет разный результат и разное внутреннее состояние. Однако монада `ListT (State s)` реализует такие вычисления, которые могут возвращать несколько результатов, но состояние используется всеми ветвями совместно. (Алехандро Мена, с.219)

Смысл типа `MaybeT (WriterT String []) Int` в том, что сначала мы применяем трансформер `WriterT String []` с типом журналирования в строку `String` к списочной монаде `[]` частично (без указания типа, он будет позже), и получаем монаду `Writer` над списком. Потом мы применяем трансформер `MaybeT` к этой монаде и типу `Int`, который у нас был базовым, собственно это и приводит к получению типа


```
rk3 :: Int -> [(Maybe Int, String)]
rk3 = runWriterT . runMaybeT . k3
```

Без распаковки с помощью runWriterT тип бы выглядел менее красиво:

```
*Main> :t (runMaybeT . k3)
(runMaybeT . k3) :: Int ->
  WriterT String [] (Maybe Int)
```

И давал такой бы возврат:

```
*Main> (runMaybeT . k3) 5
WriterT [(Nothing,"x:_5;_u:_5;_y:_10;_z:_30;_"),
  (Nothing,"x:_5;_u:_6;_"),
  (Just 210,"x:_5;_u:_7;_y:_14;_z:_42;_j:_210;_")]
```

Данная цепочка из 3-х вложенных слоёв монад показывает, что написание и поддержка таких цепочек уже не очень лёгкое дело. На практике, чаще всего обходимся цепочками из двух слоёв, и это за счёт готовых утилит происходит вполне естественно.

В [11-й лекции](#), когда мы говорили об изменяемых переменных, был уже представлен такой код:

```
import Control.Monad.State
import System.IO

code :: StateT Int IO ()
code = do
  x <- get
  liftIO $ print x
  liftIO $ putStr "Input_number:_ "
  y <- liftIO $ (readLn :: IO Int)
  let z = x + y
  put z
  liftIO $ print z
  return ()

main :: IO ()
main = do
  hSetBuffering stdout NoBuffering
  runStateT code 1
  return ()
```

И его аналог на Паскале:

```
var
  x,y,z,datastorage: integer;
begin
  datastorage := 1;
  x := datastorage;
  writeln(x);
  write('Input_number:_ ');
  readln(y);
  z := x+y;
```

```

    datastorage := z;
    writeln(z);
end.

```

Здесь в монаду State с помощью особого вида лифтинга `liftIO` был втянут функционал монады ввода-вывода **IO**. Само вычисление имеет тип **IO**

```

*Main> :t runStateT code 1
runStateT code 1 :: IO ((), Int)
*Main> :t runStateT code
runStateT code :: Int -> IO ((), Int)

```

и поэтому помещено в главную «запускалку»

```

main = do
    ...

```

Класс MonadTrans

Базовая поддержка трансформеров формально заключена в классе типов `MonadTrans` (обычно используется библиотека [mtl](#)) и описывает только один метод:

```

class MonadTrans t where
    lift :: (Monad m) => m a -> t m a

```

который должен подчиняться следующим законам:

```

lift . return = return
lift (m >>= f) = lift m >>= (lift . f)

```

где `m` — это монада и результат применения трансформера (`t m`) к монаде также должен быть монадой.

Таким образом, например, воплощение для трансформера `MaybeT` будет задано так:

```

instance MonadTrans MaybeT where
    lift = MaybeT . liftM Just

```

О методе `liftM` мы поговорим в соответствующем разделе, а пока заметим, что он является монадической версией `fmap` (или даже `map`).

Для монад `m`, в которые можно встроить вычисления ввода-вывода (**IO**), предлагается метод `liftIO` из класса [Control.Monad.IO.Class](#):

```

class Monad m => MonadIO m where
    liftIO :: IO a -> m a

```

который (кстати, метод [тоже называют иногда трансформером](#)) должен удовлетворять законам

```

liftIO . return = return
liftIO . return = return

```

Где воплощения определяются, например, следующим образом

```
instance MonadIO IO where
  liftIO = id
```

```
instance (MonadIO m) => MonadIO (MaybeT m) where
  liftIO = lift . liftIO
```

Так как при вычислениях с **IO** эта монада будет самой внутренней в цепочке монадных трансформетов, то метод `liftIO` рекурсивным образом подымает **IO**-значения на самую вершину.

Выводы о работе с трансформерами монад

При работе с вычислениями «комбинированной стратегии», когда нам необходим функционал нескольких монад, следует иметь в виду следующее:

1. Определиться с порядком вложенности монад. В простейшем случае, это две монады, и к основному функционалу одной монады мы собираемся добавить ещё что-то. В рассматриваемых выше случаях это были вычисления внутри монады **Maybe** (т.е. вычисления с возможной неудачей), к которым мы хотим добавить например, возможность журналирования, или возможность работы с состоянием, или возможность работы с вводом-выводом (с терминалом, файлами, глобальным генератором случайных чисел и т.п.)
2. При простом комбинировании функций внутри монады **Maybe** нам требовались монадические функции с сигнатурой **Int -> Maybe Int**. Для комбинированных вычислений наши функции должны приобрести новую сигнатуру

```
Monad m => Int -> MaybeT m Int
```

и при возвращении результата мы используем не простой **Just (5*x)**, а комбинацию:

```
MaybeT $ return $ Just (5*x)
```

Таким образом, теперь мы обобщили наши функции до работы в трансформере **MaybeT**, который готов работать с другой монадой **m**.

3. Дополнительный функционал, как правило, какие-то утилиты или монадические функции, мы «подтягиваем», используя `lift`, например:

```
lift $ tell ...
lift $ t x
lift $ get
```

4. В итоге, наш трансформер действует на нижележащую монаду и у нас получается некоторая новая монада более сложной структуры с комбинированными свойствами. Причём, акцент, смысл слоённого пирога трансформера и монады, как бы меняется местами (мы хотели ведь просто добавить функционал, а в итоге стратегия вычисления стала перевёрнутой):

```
MaybeT [] Int
```

— монада, обеспечивающая множественные вычисления, каждое из которых в любом месте может закончиться неудачей

MaybeT (Writer String) Int

— монада, обеспечивающая вычисления с журналированием, каждое из которых в любом месте может закончиться неудачей

5. Несколько особое место занимает монада **IO**. Она в матрёшке может быть только самой последней, в нашем случае двух слоёв:

MaybeT IO Int

— монада, обеспечивающая вычисления с побочными эффектами, каждое из которых в любом месте может закончиться неудачей. И подтягивание утилит этой монады рекомендуется делать с помощью `liftIO` (хотя в нашем последнем случае этом может быть и `lift`)

```
liftIO $ print "Let's go!"
```

И смотреть ещё многослойный пример (от 2023-го года) ниже.

6. Запуск «слоённого пирога» трансформеров монад делается в обратном порядке к тому, как мы подтягивали функционал, в случае двух слоёв, например при использовании журналирования:

```
runWriter . runMaybeT . k2 $ 3
```

А запуск для **IO** мы делаем в `main`:

```
main = do
    runMaybeT . k2 $ 3
```

Но следует иметь в виду, что если тут тип `main`

```
main :: IO (Maybe Int)
```

и в `ghci` результат будет:

```
*Main> main
"Let's go!"
Nothing
```

то при запуске реальной программы (через компиляцию или `runghc`) возвращаемые значения связаны с кодами ошибок или успешного завершения, и в нашем случае будет просто:

```
C:\code\MaybeT>runghc test-IO-2023.hs
"Let's go!"
```

Дополнительный код примеров

В тексте вывода выше использовались такие примеры комбинирования с `Writer` и `IO` (определения функций `f`, `g`, `h` используем в трансформенном виде):

```
import Control.Monad
import Control.Monad.Trans.Maybe
import Control.Monad.Identity
```

```

import Control.Monad.Trans
import Control.Monad.Writer

-- f, g, h are the same

k2 :: Int -> MaybeT (Writer String) Int
k2 x = do
    lift $ tell "Let's go!"
    y <- f x
    z <- g y
    h z

rk2 = runWriter . runMaybeT . k2 $ 3

```

и

```

k2 :: Int -> MaybeT IO Int
k2 x = do
    lift $ print "Let's go!"
    y <- f x
    z <- g y
    h z

main = do
    runMaybeT . k2 $ 3

```

А следующий пример (2023) удачного сочетания 4 слоёв монад и трансформеров.

К сожалению, модуль `Control.Monad.List` из-за ошибок в базовой структуре более не поддерживается. Рекомендуется либо скачать [его старую версию](#) и под другим именем импортировать в свой файл. Либо использовать более надёжные реализации в пакетах: [list-t](#) или [List](#) — но их документированность оставляет желать лучшего. Либо использовать идеи статьи [ListT done right](#) и сделать собственную реализацию.

```

{-# OPTIONS_GHC -fno-warn-deprecations #-}
{-# LANGUAGE FlexibleContexts #-}

```

```

import Control.Monad
import Control.Monad.Trans.Maybe
import Control.Monad.List
import Control.Monad.Writer
import Control.Monad.Identity
import System.IO
import System.Random(randomRIO)

```

```

{-
    2023: добавлена функциональность работы с монадой IO:
    - в функцию k4 просто как лог в терминал
    - в новой функции k5 помимо этого, аргумент получается глобальным генератором случайных ч
-}

```

```

f :: Monad m => Int -> MaybeT m Int
f x = do
    if x `mod` 2 == 0
    then MaybeT $ return Nothing
    else MaybeT $ return $ Just (2*x)

g :: Monad m => Int -> MaybeT m Int
g x = do
    guard (x `mod` 3 /= 0)
    MaybeT $ return $ return (3*x)

h :: Monad m => Int -> MaybeT m Int
h x = do
    when (x `mod` 5 == 0) (MaybeT $ return Nothing)
    MaybeT $ return $ Just (5*x)

tL :: Monad m => Int -> ListT m Int
tL x = ListT $ return [x, (x+1), (x+2)]

someX :: IO Int
someX = randomRIO (1,9)

k4 :: Int -> MaybeT (WriterT String (ListT IO)) Int
k4 x = do
    lift $ tell $ "x:␣" ++ (show x) ++ ";␣"
    liftIO $ putStrLn $ "x:␣" ++ (show x) ++ ";␣"
    u <- lift $ lift $ tL x
    lift $ tell $ "u:␣" ++ (show u) ++ ";␣"
    y <- f u
    lift $ tell $ "y:␣" ++ (show y) ++ ";␣"
    z <- g y
    lift $ tell $ "z:␣" ++ (show z) ++ ";␣"
    j <- h z
    lift $ tell $ "j:␣" ++ (show j) ++ ";␣"
    return j

k5 :: MaybeT (WriterT String (ListT IO)) Int
k5 = do
    x <- liftIO $ someX
    lift $ tell $ "x:␣" ++ (show x) ++ ";␣"
    liftIO $ putStrLn $ "x:␣" ++ (show x) ++ ";␣"
    u <- lift $ lift $ tL x
    lift $ tell $ "u:␣" ++ (show u) ++ ";␣"
    y <- f u
    lift $ tell $ "y:␣" ++ (show y) ++ ";␣"
    z <- g y
    lift $ tell $ "z:␣" ++ (show z) ++ ";␣"
    j <- h z
    lift $ tell $ "j:␣" ++ (show j) ++ ";␣"
    return j

```

```

main = do
  putStrLn "x=3"
  answer <- runListT . runWriterT . runMaybeT $ k4 $ 3
  print answer
  putStrLn ""

  putStrLn "x=7"
  answer <- runListT . runWriterT . runMaybeT . k4 $ 7
  print answer
  putStrLn ""

  putStrLn "x=random="
  answer <- (runListT . runWriterT . runMaybeT $ k5)
  print answer

  return ()

```

и таким вариантом выданного результата:

```

x = 3
x: 3;
[(Nothing,"x:3;u:3;y:6;"),(Nothing,"x:3;u:4;"),(Nothing,"x:3;u:5;y:
x = 7
x: 7;
[(Just 210,"x:7;u:7;y:14;z:42;j:210;"),(Nothing,"x:7;u:8;"),(Nothi
x - random =)
x: 5;
[(Nothing,"x:5;u:5;y:10;z:30;"),(Nothing,"x:5;u:6;"),(Just 210,"x:5

```

И ещё развитие одного простого примера (также основанно на устаревшем модуле `Control.Monad.List`) из лекции-13, который изначально был дан в книге Липовача:

```

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

```

с таким выводом:

```

[(1,'a'),(1,'b'),(2,'a'),(2,'b')]

```

Изменим код, чтобы вместо чисел использовались булевы значения:

```

listOfTuples :: [(Bool,Char)]
listOfTuples = do
  n <- [False,True]
  ch <- ['a','b']
  return (n,ch)

[(False,'a'),(False,'b'),(True,'a'),(True,'b')]

```



Рис. 1: лавина букв и цифр

Теперь, сделаем возможность внешнего состояния, чтобы можно было перенумеровать все листовые вершины при их последовательном вычислении. Для этого «подтянем» утилиты монады State:

```
import Control.Monad
import TransList -- local old version
import Control.Monad.State
import Control.Monad.Identity

f1 :: Monad m => ListT m Bool
f1 = ListT $ return $ [False, True]

f2 :: Monad m => ListT m Char
f2 = ListT $ return $ ['a', 'b']

listTRtuples :: ListT (State Int) (Int, Bool, Char)
listTRtuples = do
  b <- f1
  ch <- f2
  n <- lift $ get
  lift $ put (n+1)
  return (n, b, ch)

test = (evalState . runListT) listTRtuples 1
```

С таким вот результатом:

```
[(1, False, 'a'), (2, False, 'b'), (3, True, 'a'), (4, True, 'b')]
```

Данный обзор-введение в трансформеры монад был весьма поверхностен, хотя и практичен по возможности. Для дальнейшего изучения рекомендуются следующие материалы:

[Haskell/Monad_transformers](#) (перевод)

[Haskell/Monad_transformers](#)
[All_About_Monads/Monad_transformers](#)
[FPComplete: Monad Transformers](#)
[en.wikipedia: Monad transformer](#)
[hackage: Control.Monad.Trans.Class](#)
[mtl, transformers, monads-fd, monadLib, and the paradox of choice](#)
[Real World Haskell. Chapter 18. Monad transformers](#)
[Monday Morning Haskell: Monad Transformers](#)
[Трансформеры монад](#)
[Haskell. Монады. Монадные трансформеры. Игра в типы](#)
[How do you use the StateT Monad Transformers in Haskell?](#)
[ListT done right](#)
[stackoverflow: list monad transformer](#)

Полезные общие утилиты для работы с монадами

Мы уже рассматривали на прошлой лекции удобные функции для работы с монадами из класса **MonadPlus** (напр., **guard**). В этот раз рассмотрим ещё ряд функций-утилит, более общих.

Монадические вычисления с условием

Для выполнения монадических вычислений с условием предусмотрены две функции. Функция **when** принимает логический аргумент и монадическое вычисление с типом **()** и выполняет вычисление только тогда, когда логический аргумент имеет значение **True**. Если условие не выполняется, то реализуются следующие действия. Функция **unless** делает то же самое, за исключением того, что выполняет вычисления, если только логический аргумент не равен **True**.

Фактически, это аналог усечённого **if** из мира императивного программирования.

```
when :: (Monad m) => Bool -> m () -> m ()
when p s = if p then s else return ()

unless :: (Monad m) => Bool -> m () -> m ()
unless p s = when (not p) s
```

Пример работы:

```
> when (1 == 1) (print "OK")
"OK"
```

Как видим, в последовательности монадических вычислений вполне можно использовать и полную версию **if..then..else..** в том числе и без присваивания.

Замечание 1.

Напомним, что действие **return(x)** хотя и обладает намеренной похожестью на поведение **return(x)** в императивном программировании, но всё-таки таковым не является, например, не осуществляется выход из «процедуры», да и самих **return** может быть несколько. Вот как может быть переписан исходный код в [начале лекции](#):

```
k :: Int -> Maybe Int
k x = do y <- f x
        return(y)
        z <- g y
        return(z)
        h z
```

здесь **return(y)**, **return(z)** не производят выход из тела функции, а всего лишь обворачивают значения **y** и **z** в монадный контекст (в данном случае с помощью тэга **Just**).

Замечание 2.

Также отметим, что несмотря на схожесть утилит **guard** и **when** (или **unless**) они работают в разных «средах» (**guard** требует **MonadPlus**, и, значит, не будет работать с монадами **Either**, **Writer**, **State**) и работают несколькими способами. Сравните:

```
import Control.Monad

main = do
  let x = 0
  putStrLn "x=0"
  if x == 0 then putStrLn "if: x=0" else putStrLn "if: x/=0"
  when (x == 0) $ putStrLn "wh: x=0"
  putStrLn "and guard now!"
  guard (x == 0)
  putStrLn "aha"
  putStrLn "\nnext step, now x=1\n"
  let z = 1
  if z == 0 then putStrLn "if: z=0" else putStrLn "if: z/=0"
  when (z == 0) $ putStrLn "wh: z=0"
  putStrLn "and guard now!"
  guard (z == 0)
  putStrLn "aha"
```

вот с таким результатом:

```
x=0
if: x=0
wh: x=0
and guard now!
aha

next step, now x=1
```

```
if: z/=0
and guard now!
print.hs: user error (mzero)
```

т.е. на этом примере видно, что если **when** ведёт себя как усечённый **if..then**, то **guard** несёт дополнительную семантику, которую надо иметь в виду. Но на практике, чаще всего удобнее пользоваться знакомой конструкцией **if..then..else**, тем более она не требует дополнительного импорта.

MonadPlus definition for Haskell IO

<https://stackoverflow.com/questions/57447800/monadplus-io-isnt-a-monoid>

Лифтинг

Функция **liftM** *f m* позволяет немонадической функции *f* оперировать на контексте монады (с монадическим значением) *m*. Является аналогом функции **liftA** для аппликативных функторов, функции **fmap** для функторов и даже аналогом **map** для списков.

Вот как реализована функция **liftM**:

```
liftM :: (Monad m) => (a -> b) -> m a -> m b
liftM f m = m >>= (\x -> return (f x))
```

Или с использованием нотации **do**:

```
liftM :: (Monad m) => (a -> b) -> m a -> m b
liftM f m = do
  x <- m
  return (f x)
```

(буквы *m* в самой первой и в следующих строках несут разный смысл: в первой — это конструктор типа (вроде **Maybe**), в последующих — монадическое значение)

Примеры:

```
ghci> liftM sin (Just 0)
Just 0.0
ghci> liftM (replicate 10) ['a']
["aaaaaaaaaa"]

ghci> liftM (*3) (Just 8)
Just 24
ghci> fmap (*3) (Just 8)
Just 24
ghci> runWriter $ liftM not $ writer (True,"hello")
(False,"hello")
ghci> runWriter $ fmap not $ writer (True,"hello")
(False,"hello")
ghci> runState (liftM (+100) pop) [1,2,3,4]
(101,[2,3,4])
ghci> runState (fmap (+100) pop) [1,2,3,4]
(101,[2,3,4])
```

где функция `por` была определена, когда мы [тренировались со стеком](#) в предыдущей лекции.

Есть ещё варианты **liftM2** для функций двух переменных:

```
liftM2 :: (Monad m) =>
  (a -> b -> c) -> (m a -> m b -> m c)
liftM2 f =
  \a b -> do { a' <- a; b' <- b; return (f a' b') }
```

И пример её работы:

```
liftM2 (+) [0,1] [0,2] = [0,2,1,3]
liftM2 (+) (Just 1) Nothing = Nothing
```

ap

И есть ещё монадическая функция **ap**, аналогичная функции `<*>` для аппликативных функторов:

```
ap :: (Monad m) => m (a -> b) -> m a -> m b
ap = liftM2 ($)
```

т.е. `liftM2 f x y` эквивалентна

```
return f `ap` x `ap` y
```

Функция считается иногда более удобной, чем использование лифтинга:

```
Prelude Control.Monad> return (+) `ap` [0,1] `ap` [0,2]
[0,2,1,3]
Prelude Control.Monad> return (+) `ap` (Just 1) `ap` Nothing
Nothing
Prelude Control.Monad> return (*) `ap` (Just 2) `ap` (Just 3)
Just 6
```

Напомним тут разнообразные примеры из [лекции-12](#), применённые к нашему случаю списков, аппликативный вариант:

```
> (+) <$> [0,1] <*> [0,2]
[0,2,1,3]
```

Монадический вариант (и к сожалению, нужны скобки, так как для операторов `<$>` и `<*>` установлено **infixl 4**, а для функции **ap** это не так):

```
Prelude Control.Monad> ((+) <$> [0,1]) `ap` [0,2]
[0,2,1,3]
```

Или аппликативный вариант:

```
> :m Control.Applicative
> liftA (+) [0,1] <*> [0,2]
```

аналогично, монадический вариант:

```
> :m Control.Monad
> liftM (+) [0,1] <*> [0,2]
```

Или аппликативный вариант:

```
> :m Control.Applicative
> liftA2 (+) [0,1] [0,2]
```

аналогично, монадический вариант:

```
> :m Control.Monad
> liftM2 (+) [0,1] [0,2]
```

И аппликативный вариант:

```
pure (+) <*> [0,1] <*> [0,2]
```

аналогично, монадический вариант:

```
> :m Control.Monad
  return (+) `ap` [0,1] `ap` [0,3]
```

Все примеры дают одинаковые ответы.

Также можно рассмотреть `liftM3`, `liftM4`,... с многими аргументами, подобные таковым для аппликативных функторов.

[wiki.haskell: All_About_Monads/ ap and the lifting functions](#)

[The Haskell 98 Report/ 20 Monad Utilities](#)

[Описание Haskell 98/ 20. Утилиты работы с монадами](#)

lift

Для нас более полезной оказалась функция `lift`, которая объявлена в пакете `transformers` в модуле `Control.Monad.Trans.Class` в классе `MonadTrans`:

```
class MonadTrans t where
  lift :: (Monad m) => m a -> t m a
```

Реализация метода `lift` для каждого монадного трансформера своя, например, для `MaybeT`:

```
instance MonadTrans MaybeT where
  lift m = MaybeT (m >=> return . Just)
```

С `liftM` мы увидели, что сущность поднятия — перефразируя документацию — в продвижении чего-то в монаду. Функция `lift`, доступная для всех монадных трансформеров, выполняет разный тип поднятия: она продвигает вычисление из внутренней монады в комбинированную монаду.

Реализация lift

Отдельно стоит упомянуть про вариант `lift`, специфичный для **IO** и называемый `liftIO`, который является единственным методом класса `MonadIO`:

```
class (Monad m) => MonadIO m where
  liftIO :: IO a -> m a
```

и определен в [Control.Monad.IO.Class](#), выполняя следующие правила:

```
liftIO . return = return
liftIO (m >>= f) = liftIO m >>= (liftIO . f)
```

`liftIO` может быть удобен, когда у нас множеств трансформеров помещены друг за другом (как в стек) в одну комбинированную монаду. В подобных случаях, **IO** будет всегда самой внутренней монадой, и таким образом обычно нужен более чем один `lift`, чтобы поднять **IO**-значения на вершину стека. `liftIO`, однако, определен для воплощений таким образом, что позволяет нам поднять **IO**-значение из произвольной глубины, написав функцию лишь единожды.

Haskell: lift vs liftIO

join

Для списков нам как-то приходилось сталкиваться с функцией-утилитой `concat`:

```
> concat [[1,2],[3,4,5]]
[1,2,3,4,5]
> concat [[1,2],[3,4,5],[6..9]]
[1,2,3,4,5,6,7,8,9]
> concat []
[]
```

Интересно, если считать списки монадой, тогда есть ли обобщенный вариант `concat`? Оказывает, да, такая функция есть, зовут `join`, обитает в модуле `Control.Monad`.

Для списков получаем ровно то же самое:

```
Prelude> :m Control.Monad
Prelude Control.Monad join [[1,2],[3,4,5],[6..9]]
[1,2,3,4,5,6,7,8,9]
> join []
[]
```

Для монады `Maybe`:

```
> join (Just (Just 9))
Just 9
> join (Just Nothing)
Nothing
> join Nothing
Nothing
```

Применение **join** к **Nothing** равносильно применению к пустому списку: `[]`.

Кстати:

```
> concat [1]
<interactive>:30:1: error:
  * Non type-variable argument in the...
```

Такая же ошибка будет при `join [1]`, `join $ Just 1`.

С двойным вложенным значением `Writer` ситуация будет немного сложнее. Чтобы разглядеть значение монады `Writer`, результат которого сам является значением монады `Writer`, используем:

```
> runWriter $ join (writer(writer(1,"aaa"),"bbb"))
(1,"bbbbaaa")
```

Внешнее значение `"bbb"` идёт первым, затем к нему конкатенируется строка `"aaa"`. На интуитивном уровне, когда вы хотите проверить результат значения типа `Writer`, сначала вам нужно записать его моноидное значение в журнал, и только потом вы можете посмотреть, что находится внутри него.

Примеры с такими монадами как `State` можно найти в книге Липовача (с.447-448).

Собственно определение **join** таково:

```
join :: (Monad m) => m (m a) -> m a
join mm = do
  m <- mm
  m
```

Поскольку результат `mm` является монадическим значением, мы берём этот результат, а затем просто помещаем его на его собственную строку, потому что это и есть монадическое значение.

Роль функции **join**, вообще говоря, выходит за рамки просто утилиты. Она, за счёт своих свойств может быть использована для определения монады вместо привычных функций **return** и **>>=**:

```
fmap    :: (a -> b) -> m a -> m b
return  :: a -> m a
join    :: m (m a) -> m a
```

где `fmap` применяет данную функцию к каждому элементу в контейнере (т.е. из обычной функции делает отображение между монадными значениями, это функтор); **return** упаковывает элемент в монадный контейнер; **join** берет контейнер с контейнерами и сплюсчивает его в один контейнер.

При этом верны такие определения и свойства ([en.wikibooks: Understanding_monads](http://en.wikibooks.org/Understanding_monads)):

```
m >>= g = join (fmap g m)
fmap f x = x >>= (return . f)
join m   = m >>= id
```

И сами монадные законы могут быть выражены в терминах этих функций, см. напр.

Monad laws expressed in terms of join instead of bind?

Monad_(functional_programming). Definition

У функции **join**, правда, есть ещё и ряд моментов, похожих на откровенные трюки:

```
> join (,) 1
(1,1)
> join (+) 1
2
> join (-) 1
0
```

Их объяснение связано с монадой $((\rightarrow) r)$

The “magic” of “join (,)”

The $((\rightarrow) r)$ monad

«Функции в качестве аппликативных функторов» (Липовача (с.325))

foldM

Разбор этой функции с примером уже был в [Лекции-14](#).

Ранее мы изучали свёртки на списках, и у нас была функция левой свёртки **foldl**:

```
foldl :: (a -> b -> a) -> a -> [b] -> a
```

функция **foldl** принимает бинарную функцию, исходный аккумулятор и сворачиваемый список, а затем сворачивает его слева в одно значение, используя бинарную функцию.

Функция **foldM** делает то же самое, только она принимает бинарную функцию, производящую монадическое значение, и сворачивает список с её использованием.

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

Если **t** понимать как список, то будет такой тип:

```
foldM :: (Monad m) =>
  (a -> b -> m a) -> a -> [b] -> m a
```

Функция **foldM** аналогична **foldl**, за исключением того, что её результат инкапсулируется в монаде. **foldM** работает над перечисленными аргументами слева направо. (При этом могла бы возникнуть проблема там, где (\gg) и «сворачивающая функция» не являются коммутативными)

```
foldM f a1 [x1, x2, ..., xm]
```

```
==
```

```
do
```

```
  a2 <- f a1 x1
  a3 <- f a2 x2
  ...
  f am xm
```


Если требуется вычисление справа налево, входной список следует обратить (поменять порядок элементов на обратный).

Control-Monad: foldM

Давайте сложим список чисел с использованием свёртки:

```
ghci> foldl (\acc x -> acc + x) 0 [2,8,3,1]
14
```

Теперь, нужно сложить список чисел, но с дополнительным условием: если какое-то число в списке больше 9, всё должно закончиться неудачей?! Иметь бы смысл использовать бинарную функцию, которая проверяет, больше ли текущее число, чем 9. Если больше, то функция оканчивается неудачей; если не больше — продолжает свой «радостный путь». Из-за этой добавленной возможности неудачи давайте заставим нашу бинарную функцию возвращать аккумулятор **Maybe** вместо обычного.

Вот бинарная функция:

```
binSmalls :: Int -> Int -> Maybe Int
binSmalls acc x
  | x > 9      = Nothing
  | otherwise = Just (acc + x)
```

Поскольку наша бинарная функция теперь является монадической, мы не можем использовать её с обычной функцией **foldl**, следует использовать функцию **foldM**.

```
ghci> foldM binSmalls 0 [2,8,3,1]
Just 14
ghci> foldM binSmalls 0 [2,11,3,1]
Nothing
```

(Липовача, с.453)

filterM

Аналогично только что рассмотренной функции **foldM** данная монадическая функция является «кузиной» списочной функции **filter**:

```
filter :: (a -> Bool) -> [a] -> [a]
```

Только в этот раз мы сделаем следующие изменения:

```
filterM :: (Monad m) =>
  (a -> m Bool) -> [a] -> m [a]
```

Здесь мы видим, что к булевому значению и к списку-результату приложен некоторый общий контекст, монадная обёртка.

Рассмотрим на примере, как это можно было бы использовать.

Давайте возьмём список и оставим только те значения, которые меньше 4. Для начала мы используем обычную функцию **filter**:

```
ghci> filter (\x -> x < 4) [9,1,5,2,10,3]
[1,2,3]
```

Теперь давайте создадим предикат, который помимо представления результата **True** или **False** также предоставляет журнал своих действий.

```
keepSmall :: Int -> Writer [String] Bool
keepSmall x
  | x < 4 = do
    tell ["Saving:␣" ++ show x]
    return True
  | otherwise = do
    tell [show x ++ "␣too␣big"]
    return False
```

Теперь давайте передадим его функции **filterM** вместе со списком. Поскольку предикат возвращает значение типа **Writer**, результирующий список также будет значением типа **Writer**.

```
ghci> fst $ runWriter $ filterM keepSmall [9,1,5,2,10,3]
[1,2,3]
```

и журнал:

```
> snd $ runWriter $ filterM keepSmall [9,1,5,2,10,3]
["9␣too␣big","Saving:␣1","5␣too␣big",
 "Saving:␣2","10␣too␣big","Saving:␣3"]
```

и полный выход:

```
> runWriter $ filterM keepSmall [9,1,5,2,10,3]
([1,2,3],["9␣too␣big","Saving:␣1","5␣too␣big","Saving:␣2","10␣too␣big","Saving:␣3"])
```

И можно сделать красивый вывод (используя другую монадическую утилиту **mapM_**):

```
> mapM_ putStrLn $ snd $ runWriter $ filterM keepSmall [9,1,5,2,10,3]
9 too big
Saving: 1
5 too big
Saving: 2
10 too big
Saving: 3
```

Липовача (с.451)

И ещё:

Очень крутой трюк в языке Haskell — использование функции **filterM** для получения множества-степени списка (если мы сейчас будем думать о нём как о множестве). Липовача (с.451-452)

```
powerset :: [a] -> [[a]]
powerset xs = filterM (\x -> [True, False]) xs
```

И это работает!!

```
ghci> powerset [1,2,3]
[[1,2,3],[1,2],[1,3],[1],[2,3],[2],[3],[]]
```

sequence, sequence_, mapM, mapM_, forM, forM_

Функция **sequence** берет список монадических вычислений, выполняет каждое из них по очереди и возвращает список результатов. Если какое-либо из вычислений завершается ошибкой, вся функция завершается ошибкой:

```
sequence :: Monad m => [m a] -> m [a]
sequence = foldr mcons (return [])
    where mcons p q =
        p >>= \x -> q >>=
            \y -> return (x:y)
```

```
> sequence [print 1, print 2, print 3]
1
2
3
```

```
> sequence [Just 1, Just 2, Just 3]
Just [1,2,3]
```

Функция **sequence_** (обратите внимание на подчеркивание) ведет себя так же, как **sequence**, но не возвращает список результатов. Это полезно, когда важны только побочные эффекты монадических вычислений.

```
sequence_ :: Monad m => [m a] -> m ()
sequence_ = foldr (>>) (return ())
```

```
> sequence_ [print 1, print 2, print 3]
1
2
3
```

```
> sequence_ [Just 1, Just 2, Just 3]
Just ()
```

[All About Monads/ In the standard prelude](#)

[A tour of the Haskell Monad functions/ sequence](#)

Функция **mapM** отображает монадическое вычисление по списку значений и возвращает список результатов. Она определена в терминах списочной функции **map** и функции **sequence** выше:

```
mapM :: Monad m => (a -> m b) -> [a] -> m [b]
mapM f as = sequence (map f as)
```

Примеры:

```
> mapM Just [0, 1, 2]
Just [0,1,2]
```

```
> mapM print [0, 1, 2]
0
1
2
```

```
> mapM (\x -> [x]) [0, 1, 2]
[[0,1,2]]
```

Функция `mapM_` является аналогом функции `mapM`. Она используется, если результат функции не важен, а важны те действия, которые происходят при преобразовании списка. В нашем случае это накопление результата. Посмотрим на определение этой функции:

```
mapM_ :: Monad m => (a -> m b) -> [a] -> m ()
mapM_ f as = sequence_ (map f as)
```

Основное отличие состоит в функции `sequence_`. Раньше мы собирали значения в список, а теперь отбрасываем их с помощью константной функции `>>`. В конце мы возвращаем значение единичного типа `()`.

```
> mapM_ Just [0,1,2]
Just ()
```

```
> mapM_ print [0,1,2]
0
1
2
```

```
> mapM_ (\x -> [x]) [0,1,2]
[()]
```

Функция `forM` (определена в модуле `Control.Monad`) похожа на функцию `mapM`, но её параметры поменяны местами. Первый параметр — это список, второй — это функция, которую надо применить к списку и затем свести действия из списка в одно действие.

Трюк с её использованием можно найти у Липовача (с.221).

С функцией `forM_` — аналогично. У нас в лекции-10 был пример, когда она работала как императивный цикл:

```
forM_ [1..k] $ \i -> do
  putStrLn $ "i_=" ++ show i
```