Деревья

```
data Tree a = Empty | Node a (Tree a) (Tree a)
```

```
list2tree :: [a] -> Tree a
```

list2tree

Написать функцию, строящую сбалансированное дерево в один проход

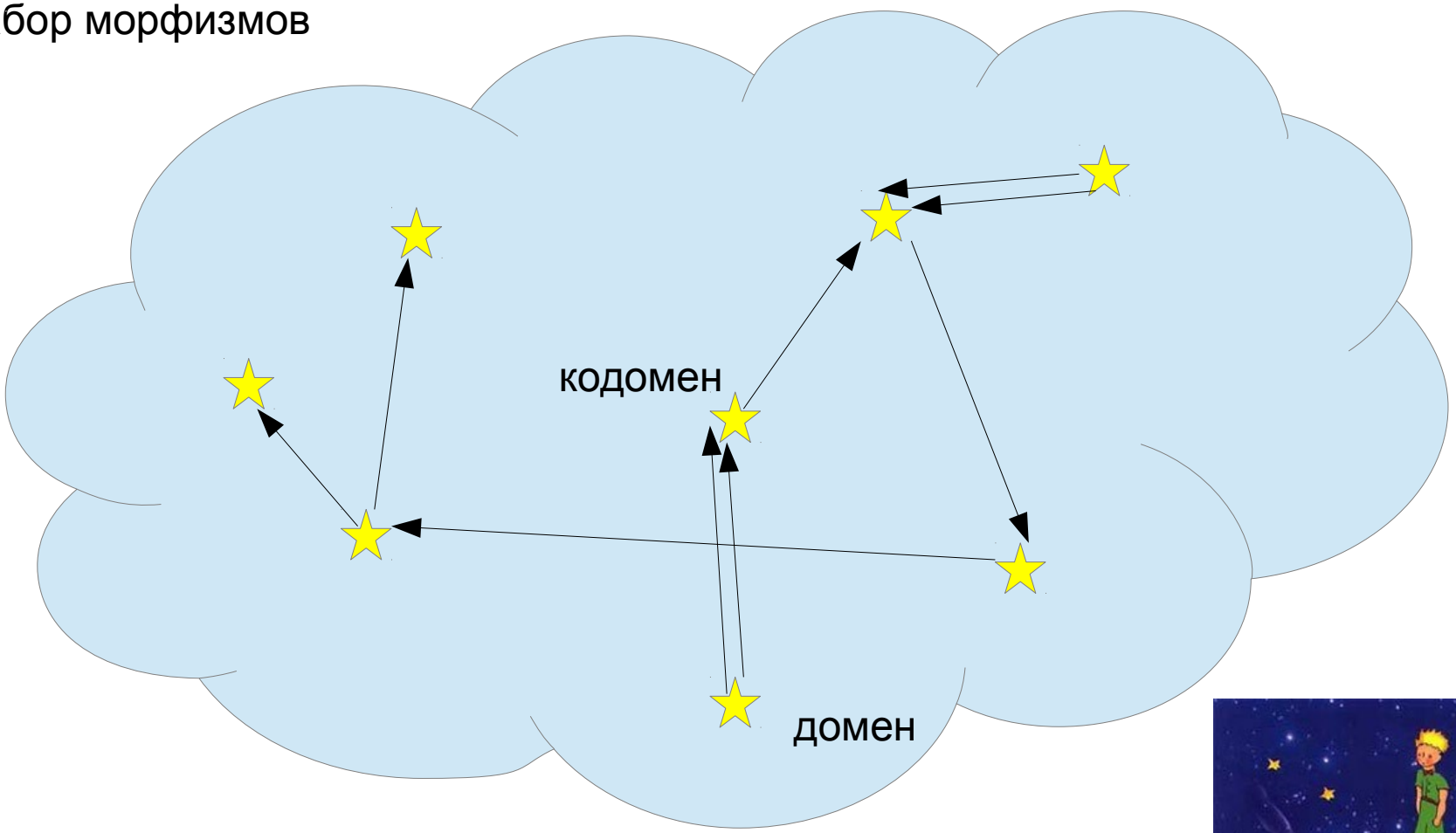
Деревья

```
list2tree xs = fst (build (length xs) xs)


build :: Int -> [a] -> (Tree a, [a])
build 0 xs = (Empty, xs)
build 1 (x:xs) = (Node x Empty Empty, xs)
build n (x:xs) = (Node x u v, xs'')
    where
        m = div (n-1) 2
        (u, xs') = build m xs
        (v, xs'') = build (n-1-m) xs'
```

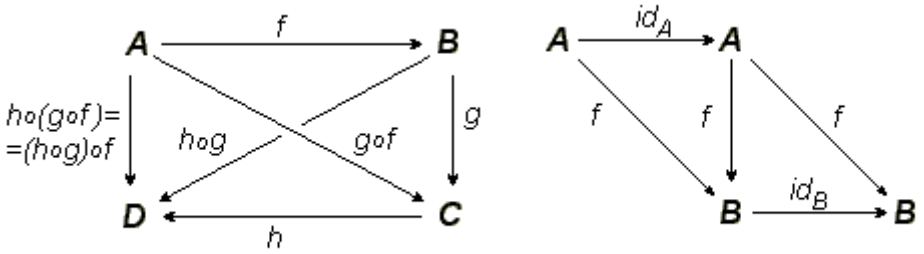
Категории

- набор объектов
- набор морфизмов



Категории

всякие аксиомы:
- композиция ассоциативна
- есть тождественный морфизм



(ц) википедия

Категории

когда домен = кодомен, это – эндоморфизм

классический пример: категория Set

объекты категории Set — это всевозможные множества, а морфизмы — это функции между этими множествами.

категория Hask.
объекты — это типы данных, возможные в языке Haskell: Int, [], Tree, a, b

а морфизмы — функции языка Haskell: Int → Int, (a → b) → a → b, Tree → Int

Категории

```
class  Category (~>)  where
  (.) :: (a ~> b) -> (b ~> c) -> (a ~> c)
  id  :: a ~> a
```

законы:

1. ∀ a, b, c : a . (b . c) = (a . b) . c
2. ∀ m : m . id = m = id . m

Функторы

функтор F — это отображение между категориями, такое, что структура категории сохраняется или, другими словами, это гомоморфизм между двумя категориями

При этом функтор отображает объекты первой категории в объекты второй, а морфизмы первой — в морфизмы второй категории. К тому же накладываются определённые ограничения — аксиомы.

Если функтор отображает категорию саму в себя, такой функтор называется эндофунктором.

Функторы

Любой конструктор выполняет преобразования
[]: a → [a]
Maybe: b → Maybe b: Just b | Nothing
data C c = …
C: c → C c
Если теперь такое отображение объектов (конструктор типов C с одним параметром) дополнить отображением морфизмов, то получим функтор, действующий из Hask в Hask, эндофунктор на категории Hask.

морфизм is a -> b

Согласно аксиомам функторов, морфизм между объектами a и b должен отображаться в морфизм между объектами (C a) и (C b). Таким образом, отображение морфизмов должно быть функцией следующего вида:

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

Функторы

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

1. ∀ f, g : fmap f . fmap g = fmap (f . g)
2. fmap id = id
   id x = x
   fmap id x = id x

Функторы

```
> map (+1) [1,2,3,4,5]
[2,3,4,5,6]
```

```
instance Functor [] where
  fmap = map
```

Функторы

```
fmap id = id
fmap (g . h) = fmap g . fmap h


(read . show $ 4)::Int
fmap (show . (+2)) [1..4]
(fmap show . fmap (+2)) [1..4]
```

Функторы

```
instance Functor Tree where
    fmap g EmptyTree = EmptyTree
    fmap g (Node a l r) = Node (g a) (fmap g l) (fmap g r)
```

Функторы

```
> fmap length (list2tree [“goodbye”,”cruel”,”world”])
```

Функторы

```
instance Functor Maybe where
   fmap f (Just x) = Just (f x)
   fmap _ Nothing  = Nothing

(fmap show . fmap (+2)) $ Just 4
(fmap show . fmap (+2)) $ Nothing
```

Функторы

```
instance Functor Maybe where
   fmap f (Just x) = Just (f x)
   fmap _ Nothing  = Nothing
```

Доказать, что

```
fmap (f . g) = fmap f . fmap g

fmap (f . g) F = fmap f (fmap g F)
```

Понимание функтора

```
:t fmap

fmap :: Functor f => (a -> b) -> f a -> f b

fmap :: Functor f => (a -> b) -> (f a -> f b)

:t fmap (show . (1+))
… :: (Functor f, Num b, Show b) => f b -> f String

:t fmap (replicate 3)
… :: Functor f => f a -> f [a]

fmap (replicate 3) [1,2,3]
fmap (replicate 3) $ Just 2

fmap (replicate 3) Just 2 ?
```

Аппликативные функторы

```
let a = fmap (*) [1,2,3,4]
:t a
a :: [Integer -> Integer]
fmap (\f -> f 9) a
[9,18,27,36]

import Control.Applicative

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

Аппликативные функторы (Maybe)

```
instance Applicative Maybe where
    pure = Just
    Nothing <*> _ = Nothing
    (Just f) <*> something = fmap f something

Just (+3) <*> Just 9
pure (+3) <*> Just 9
Just (++"hahah") <*> Nothing
Nothing <*> Just "woot"

pure (+) <*> Just 3 <*> Just 5
pure (\x y z -> x + y + z) <*> Just 3 <*> Just 5 <*> Just 2
```

Аппликативные функторы (Maybe)

```
(<$>) :: (Functor f) => (a -> b) -> f a -> f b
f <$> x = fmap f x

pure (\x y z -> x + y + z) <*> Just 3 <*> Just 5 <*> Just 2
(\x y z -> x + y + z) <$> Just 3 <*> Just 5 <*> Just 2
```

Аппликативные функторы (List)

```
instance Applicative [] where
    pure x = [x]
    fs <*> xs = [f x | f <- fs, x <- xs]

[(*0),(+100),(^2)] <*> [1,2,3]
[0,0,0,101,102,103,1,4,9]

[(+),(*)] <*> [1,2] <*> [3,4]
[4,5,5,6,3,4,6,8]

[ x*y | x <- [2,5,10], y <- [8,10,11]]
[16,20,22,40,50,55,80,100,110]

(*) <$> [2,5,10] <*> [8,10,11]
[16,20,22,40,50,55,80,100,110]
```

Аппликативные функторы (ZipList)

```
instance Applicative ZipList where
        pure x = ZipList (repeat x)
        ZipList fs <*> ZipList xs =
          ZipList (zipWith (\f x -> f x) fs xs)

getZipList $ (+) <$> ZipList [1,2,3] <*> ZipList [1,1,1]
[2,3,4]

getZipList $ (+) <$> ZipList [1,2,3] <*> ZipList [1,1..]
[2,3,4]

getZipList $ (,,) <$> ZipList "dog" <*> ZipList "cat" <*> ZipList "rat"
[('d','c','r'),('o','a','a'),('g','t','t')]
```

Аппликативные функторы

```
liftA2 :: (Applicative f) => (a -> b -> c) -> f a -> f b -> f c
liftA2 f a b = f <$> a <*> b

liftA2 (:) (Just 1) (Just [2])

liftA3 (,,) [1] [2] [3]

sequenceA :: (Applicative f) => [f a] -> f [a]
sequenceA [] = pure []
sequenceA (x:xs) = (:) <$> x <*> sequenceA xs

sequenceA [Just 3, Just 2, Just 1]
sequenceA [Just 3, Nothing, Just 1]
sequenceA [(+3),(+2),(+1)] 3
sequenceA [[1,2,3],[4,5,6]]
sequenceA [[1,2,3],[4,5,6],[3,4,4],[]]

sequenceA через свертку !
```

Аппликативные функторы

```
sequenceA :: (Applicative f) => [f a] -> f [a]
sequenceA = foldr (liftA2 (:)) (pure [])
```

```
import Data.Traversable

and $ map (\f -> f 7) [(>4),(<10),odd]
True

and $ sequenceA [(>4),(<10),odd] 7
True

:t getLine
getLine :: IO String

sequenceA [getLine, getLine, getLine]
```

Аппликативные функторы

```
pure f <*> x = fmap f x
pure id <*> v = v
pure (.) <*> u <*> v <*> w = u <*> (v <*> w)
pure f <*> pure x = pure (f x)
```

Монада

монада – моноид в категории эндофункторов (ц) википедия

Пусть у нас есть какой-нибудь эндофунктор на категории Hask (т.е. тип f, являющийся экземпляром класса Functor). Дополним его структурой моноида.

Обобщённый моноид M в моноидальной категории C — это морфизм (++) в этой категории из M ⊗ M в M, а также морфизм mempty из I в M.

```
class  MonoidCat (~>) ( ⊗) I => Monoid (~>) ( ⊗) I M  where
  (++) :: M ⊗ M ~> M   (join :: Monad m => m (m a) -> m a)
  mempty :: I ~> M
```

Монады. Зачем?

return

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

> :t return
return :: Monad m => a -> m a
> return "Hello" == Just "Hello"
?
> return "Hello" == ["Hello"]
?
```

return

Согласованность
(p >>= return) = p
do {x <- p; return x} = do {p}

(return e >>= f) = f e
do {x <- return e; f x} = do {f e}

Ассоциативность (>>=) :
((p >>= f) >>= g) = p >>= (\x -> (f x >>= g))
Этот закон позволяет воспринимать последовательность a ; b ; c ; ... как монолитную и не заботиться о расстановке скобок в ней.

Композиция Клейсли
(>=>) :: Monad m => (a -> m b) -> (b -> m c) -> (a -> m c)
(f >=> g) x = f x >>= g

Монады. Зачем?

```
> let x = [1,2,3]
> (x >>= return) == x
?
```

Монады. Зачем?

```
> let x = [1,2,3]
> (x >>= return) == x
True
```

правая единица (x >>= return) = x

Можно переписать этот закон как
(x >>= (\a -> return a)) = x,

то есть, связывание монадного вычисления x с вычислением, зависящим от параметра и просто-напросто возвращающим этот параметр, есть тождественная функция

Стандартные монады

Монада Identity (тождественная монада), не меняет ни тип значений, ни стратегию связывания вычислений.

```
data Identity a = Identity a
return a = Identity a
(Identity a) >>= f = f a
```

Стандартные монады

Maybe (монада вычислений с обработкой отсутствующих значений)

```
data Maybe a = Nothing | Just a
```

Реализация тривиальна:

```
return a = Just a
Nothing >>= f = Nothing
(Just a) >>= f = f a
```

Стандартные монады

```
lookup :: a -> [(a, b)] -> Maybe b

case lookup 'f' [('f','u'),('n','n'),('y','?')] of
    Nothing -> Nothing
    Just y -> case lookup y [('o','l'),('u','c'),('k','y')] of
        Nothing -> Nothing
        Just z -> lookup z [('d','u'),('c','k')]

lookup x [('f','u'),('n','n'),('y','?')] >>=
    (\a -> lookup a [('o','l'),('u','c'),('k','y')]) >>=
        (\b → lookup b [('d','u'),('c','k')])

do
    y <- lookup x [('f','u'),('n','n'),('y','?')]
    z <- lookup y [('o','l'),('u','c'),('k','y')]
    return (lookup z [('d','u'),('c','k')])
```

Стандартные монады (Maybe)

```
maybeHalf :: Int -> Maybe Int
maybeHalf a
        | even a = Just (div a 2)
        | otherwise = Nothing

> Just 10 >>= maybeHalf
Just 5
> Just 10 >>= maybeHalf >>= maybeHalf
Nothing
> Just 10 >>= maybeHalf >>= maybeHalf >>= maybeHalf
Nothing
```

Стандартные монады (Maybe)

```
maybeHalfN :: Int -> Int -> Maybe Int
?
```

Стандартные монады (Maybe)

```
maybeHalfN :: Int -> Int -> Maybe Int
maybeHalfN 0 a = Just a
maybeHalfN n a = maybeHalf a >>= (\x -> maybeHalfN (n-1) x)

maybeHalfN n a = do
  result <- maybeHalf a
  maybeHalfN (n-1) a
```

Стандартные монады

Монада List (вычисления, которые могут возвращать 0 или более результатов)

В этой монаде значения представляют собой списки, которые можно интерпретировать как несколько возможных результатов одного вычисления. Если одно вычисление зависит от другого, то второе вычисление производится для каждого результата первого, и полученные результаты (второго вычисления) собираются в список.

```
return a = [a]
params >>= f = concat [f x | x <- params]
```

Стандартные монады

Таким образом, params >>= f :: [b], как и следовало ожидать – это список возможных результатов применения функции к каждому из вариантов входного аргумента.

Например,

```
let x = [1 .. 5] >>= (\x -> [x .. x+3 ])
```

```
["~/music", "~/work"] >>= getDirectoryContents =
  [
    "~/music/Bach",
    "~/music/Beethoven",
    "~/music/Rammstein",
    "~/work/projects",
    "~/work/documents"
  ]
```

pipe, F#

```
[1,2,3,4] |> filter (fun x -> odd x)
```

Стандартные монады

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

[(x, y) | x <- [1, 2, 3], y <- [1, 2, 3], x /= y]

do x <- [1, 2, 3]
   y <- [1, 2, 3]
   True <- return (x /= y)
   return (x, y)

[1, 2, 3] >>= (\x -> [1, 2, 3] >>= (\y -> return (x /= y) >>=
  (\r -> case r of
    True -> return (x, y)
    _    -> fail "")))


>=>
```

Стандартные монады

Монада IO

определяет операцию (>>=) как последовательное выполнение двух её операндов, а результат выполнения первого операнда последовательно передаётся во второй

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

  m >> k = m >>= \_ -> k
```

return

```
numIO :: IO Int
numIO = return 42
```

Do notation

do {p} = p

do {p;stmts} = p >> do {stmts}

do {x <- p;stmts} = p >>= \x -> do {stmts}

IO

```
putChar :: Char -> IO ()
putChar 'x'

(>>) :: IO () -> IO () -> IO ()
putChar 'x' >> putChar '\n'

return :: a -> IO a

psLn :: String -> IO ()
psLn xs = foldr (>>) (return ()) (map putChar xs) >> putChar '\n'
```

IO

```
getChar :: IO Char


(>>=) :: IO a -> (a -> IO b) -> IO b


gl :: IO [Char]
gl = getChar >>= \x ->
        if x == '\n' then return []
        else getLine >>= \xs -> return (x:xs)
```

do-notation

```
gl = do
        x <- getChar
        if x == '\n' then return []
        else do
                xs <- gl
                return (x:xs)
```

IO – функтор

```
instance Functor IO where
    fmap f action = do
        result <- action
        return (f result)

main = do
    line <- getLine
    let line' = (++ “!”) line
    putStrLn line

main = do
    line <- fmap (++ “!”) getLine -- line <- fmap ((++ “!”) . reverse) getLine
    putStrLn line
```

IO – аппликативный функтор

```
instance Applicative IO where
    pure = return
    a <*> b = do
        f <- a
        x <- b
        return (f x)

concatLines :: IO String
concatLines = do
    a <- getLine
    b <- getLine
    return $ a ++ b

concatLines :: IO String
concatLines = (++) <$> getLine <*> getLine
```

IO

```
import Control.Monad
import System.Random

rollDiceIO :: IO (Int, Int)
rollDiceIO = liftM2 (,) (randomRIO (1,6)) (randomRIO (1,6))

rollNDiceIO ?
```

IO

```
rollNDiceIO :: Int -> IO [Int]
rollNDiceIO 0 = return []
rollNDiceIO n = do
  randomRIO (1, 6) >>=
    (\num -> rollNDiceIO (n-1) >>=
        (\list -> return (num:list)))
```

IO

```
rollNDiceIO :: Int -> IO [Int]
rollNDiceIO 0 = return [0]
rollNDiceIO n = do
  num <- randomRIO (1, 6)
  list <- rollNDiceIO (n-1)
  return (num:list)
```

IO

коммутативность

```
num <- randomRIO (1, 6)
list <- rollNDiceIO (n-1)
return (num:list)
```

```
list <- rollNDiceIO (n-1)
num <- randomRIO (1, 6)
return (num:list)
```

IO

```
rollNDiceIO :: Int -> IO [Int]
rollNDiceIO n = replicateM n (randomRIO (1, 6))
```

```
rollNDiceIO n = sequence $ map (\a -> randomRIO(1,6)) [1..n]
```

Control.Monad

```
sequence :: Monad m => [m a] -> m [a]

mapM, mapM_  :: Monad m => (a -> m b) -> [a] -> m [b]

putStr  :: String -> IO ()

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

liftM :: Monad m => (a1 -> r) -> m a1 -> m r

ap :: Monad m => m (a -> b) -> m a -> m b

return f `ap` x1 `ap` ... `ap` xn

liftMn f x1 x2 ... xn

fail :: String -> m a
```

State

```
type State s a = s -> (a,s)


put :: s -> State s ()
get :: State s s
state :: (s -> (a,s)) -> State s a
runState State s a -> ( s -> (a, s))
evalState State s a -> s -> a
```

State

```
newtype State s a = State { runState :: s -> (a, s) }
instance Monad (State s) where
return :: a -> State s a
return x = State ( \ st -> (x, st) )
(>>=) :: State s a -> (a -> State s b) -> State s b
processor >>= processorGenerator = State $ \ st ->
                                   let (x, st') = runState processor st
                                   in runState (processorGenerator x) st'
```

State

```
runState (return 'X') 1
runState get 1
runState (put 5) 1
runState (do { put 5; return 'X' }) 1
runState (put 5 >>= \_ -> return 'X') 1
postincrement = do { x <- get; put (x+1); return x }

runState postincrement 1
predecrement = do { x <- get; put (x-1); get }

runState predecrement 1
runState (modify (+1)) 1

runState (gets (+1)) 1
evalState (gets (+1)) 1
execState (gets (+1)) 1
```

State

```
return :: a -> State s a
return x s = (x,s)

get :: State s s
get s = (s,s)

put :: s -> State s ()
put x s = ((),x)

modify :: (s -> s) -> State s ()
modify f = do { x <- get; put (f x) }

gets :: (s -> a) -> State s a
gets f = do { x <- get; return (f x) }
```

State

```
import Control.Monad.State
list2tree xs = evalState (build (length xs)) xs
build :: Int -> State [a] (Tree a)
build 0 = return Empty
build 1 = do
        x:xs <- get
        put xs
        return (Node x Empty Empty)
build n = do
        x:xs <- get
        put xs
        let m = div (n-1) 2
        u <- build m
        v <- build (n-1-m)
        return (Node x u v)
```

State

data Tree a = Leaf a | Node (Tree a) (Tree a) ?

Random

```
type GeneratorState = State StdGen
type Generator = (GeneratorState Int, GeneratorState Int)

randomChar :: GeneratorState Char
randomChar = do
  gen <- get
  let (charInt, newGenerator) = randomR (97, 122) gen
  put newGenerator
  return (chr charInt)
```

Random

```
randomString :: GeneratorState String
randomString = do
  gen <- get
  let (v10, newGen) = randomR (1, 9) gen :: (Int, StdGen)
  case v10 of
    1 -> put newGen >> return []
    _ -> do
      let (x, newGen') = runState randomChar newGen
      let (xs, newGen'') = runState randomString newGen'
      put newGen''
      return (x:xs)
```