



Министерство науки и высшего образования Российской Федерации  
Калужский филиал  
федерального государственного бюджетного  
образовательного учреждения высшего образования  
«Московский государственный технический университет имени Н.Э. Баумана  
(национальный исследовательский университет)»  
(КФ МГТУ им. Н.Э. Баумана)

**ФАКУЛЬТЕТ** ИУК «Информатика и управление»

**КАФЕДРА** ИУК5 «Системы обработки информации»

## ЛАБОРАТОРНАЯ РАБОТА №2

«Типы данных Haskell»

**ДИСЦИПЛИНА:** «Перспективные технологии разработки программных средств»

Выполнил: студент гр. ИУК4-31М \_\_\_\_\_ ( Сафронов Н.С. )  
(подпись) (Ф.И.О.)

Проверил: \_\_\_\_\_ ( Кириллов В.Ю. )  
(подпись) (Ф.И.О.)

Дата сдачи (защиты):

Результаты сдачи (защиты):

- Балльная оценка:

- Оценка:

Калуга, 2025

## **Цель:**

Изучение идеологии типизации данных в Haskell. Получение навыков проектирования типов данных приложения.

## **Задачи:**

- Подготовка и настройка платформы для Haskell.
- Программирование и анализ результатов.
- Написание отчета о работе.

## **Результаты выполнения работы**

### **Вопрос 14.1**

Обратите внимание, что Enum не требует ни Eq, ни Ord, хотя отображает типы на Int (который реализует Eq и Ord). Игнорируя тот факт, что вы легко можете использовать deriving для Eq и Ord, примените сгенерированную реализацию Enum, чтобы упростить ручное определение экземпляров Eq и Ord.

## **Ответ:**

```
data Color = Red | Green | Blue
    deriving (Enum)

instance Eq Color where
    c1 == c2 = fromEnum c1 == fromEnum c2

instance Ord Color where
    compare c1 c2 = compare (fromEnum c1) (fromEnum c2)
```

### **Задание 14.2**

Определите пятигранную кость (тип FiveSidedDie). Затем определите класс типов Die и хотя бы один метод, который был бы полезен для игровой кости. Также включите суперклассы, которые, по вашему мнению, будут полезны для кости. Наконец, сделайте ваш FiveSidedDie его экземпляром.

## Ответ:

```
-- FiveSidedDie представляет собой пятигранную кость
data FiveSidedDie = Side1 | Side2 | Side3 | Side4 | Side5
    deriving (Show, Eq, Ord, Enum, Bounded)

class (Eq a, Ord a, Bounded a, Enum a) => Die a where
    rollValue :: a -> Int
    allSides :: [a]
    allSides = [minBound .. maxBound]

instance Die FiveSidedDie where
    rollValue side = fromEnum side + 1
import Data.Bits (xor)
import Data.Char (ord, chr)
```

## Задание 15.5

```
-- Cipher - класс шифра
class Cipher c where
    encode :: c -> String -> String
    decode :: c -> String -> String

-- prng генерирует значение
prng :: Int -> Int -> Int -> Int -> Int
prng a b maxNumber seed = (a * seed + b) `mod` maxNumber

-- prngStream генерирует бесконечный список значений
prngStream :: Int -> Int -> Int -> Int -> [Int]
prngStream a b maxNumber seed = next : prngStream a b maxNumber next
    where
        next = prng a b maxNumber seed

-- StreamCipher - потоковый шифр
data StreamCipher = StreamCipher
    { scA :: Int
    , scB :: Int
    , scMax :: Int
    , scSeed :: Int
    }

cipherWithStream :: [Int] -> String -> String
```

```

cipherWithStream keyStream text = zipWith go text keyStream
  where
    go c k = chr (ord c `xor` (k `mod` 256))

instance Cipher StreamCipher where
  decode = encode
  encode cipher = cipherWithStream (prngStream (scA cipher) (scB cipher) (scMax
cipher) (scSeed cipher))

myCipher :: StreamCipher
myCipher = StreamCipher
  { scA = 1337
  , scB = 7
  , scMax = 1000000
  , scSeed = 12345
  }

-- GHCi:
-- > let secret = encode myCipher "Hello, Nikita!"
-- > decode myCipher secret
-- "Hello, Nikita!"

```

## Задание 16.2

Создайте тип Shape, включающий в себя следующие формы: Circle, Square и Rectangle. Затем напишите функции вычисления периметра и площади Shape.

### Ответ:

```

-- Circle - круг, который задаётся радиусом
newtype Circle = Circle Double
  deriving (Show)

-- Square - квадрат, который задаётся длиной стороны
newtype Square = Square Double
  deriving (Show)

-- Rectangle - прямоугольник, который задаётся шириной и высотой
data Rectangle = Rectangle Double Double
  deriving (Show)

```

```

data Shape =
  ShapeCircle Circle
  | ShapeSquare Square
  | ShapeRectangle Rectangle
  deriving (Show)

area :: Shape -> Double
area (ShapeCircle (Circle r))      = pi * r * r
area (ShapeSquare (Square s))      = s * s
area (ShapeRectangle (Rectangle w h)) = w * h

perimeter :: Shape -> Double
perimeter (ShapeCircle (Circle r))  = 2 * pi * r
perimeter (ShapeSquare (Square s))  = 4 * s
perimeter (ShapeRectangle (Rectangle w h)) = 2 * (w + h)

```

### Задание 17.1

Ваша текущая реализация `Color` не содержит нейтрального элемента. Перепишите свой код так, чтобы у `Color` был нейтральный элемент, а затем сделайте `Color` представителем класса `Monoid`.

### Ответ:

```

import Data.Semigroup (Semigroup)

-- Color - перечисление цветов
data Color = Clear | Red | Yellow | Blue | Green | Purple | Orange
  deriving (Enum)

instance Eq Color where
  c1 == c2 = fromEnum c1 == fromEnum c2

instance Ord Color where
  compare c1 c2 = compare (fromEnum c1) (fromEnum c2)

combineColors :: Color -> Color -> Color
combineColors Clear c = c
combineColors c Clear = c
combineColors Red Blue = Purple
combineColors Blue Red = Purple
combineColors Red Yellow = Orange

```

```

combineColors Yellow Red = Orange
combineColors Yellow Blue = Green
combineColors Blue Yellow = Green
combineColors c1 c2
  | c1 == c2 = c1
  | otherwise = error "Unknown color combination"

```

```

instance Semigroup Color where
  (<)> = combineColors

```

```

instance Monoid Color where
  mempty = Clear

```

## Задание 20

```

import Data.List
import qualified Data.Map as Map
import Data.Semigroup
import Data.Maybe

data TS a = TS [Int] [Maybe a]

createTS :: [Int] -> [a] -> TS a
createTS times values = TS completeTimes extendedValues
  where completeTimes = [minimum times .. maximum times]
        timeValueMap = Map.fromList (zip times values)
        extendedValues = map (`Map.lookup` timeValueMap) completeTimes

fileToTS :: [(Int, a)] -> TS a
fileToTS tvPairs = createTS times values
  where (times, values) = unzip tvPairs

showTVPair :: Show a => Int -> Maybe a -> String
showTVPair time (Just value) = mconcat [show time, "|", show value, "\n"]
showTVPair time Nothing = mconcat [show time, "|NA\n"]

instance Show a => Show (TS a) where
  show (TS times values) = mconcat rows
    where rows = zipWith showTVPair times values

insertMaybePair :: Ord k => Map.Map k v -> (k, Maybe v) -> Map.Map k v

```

```

insertMaybePair myMap (_, Nothing) = myMap
insertMaybePair myMap (key, Just value) = Map.insert key value myMap

combineTS :: TS a -> TS a -> TS a
combineTS (TS [] []) ts2 = ts2
combineTS ts1 (TS [] []) = ts1
combineTS (TS t1 v1) (TS t2 v2) = TS completeTimes combinedValues
    where bothTimes = mconcat [t1, t2]
          completeTimes = [minimum bothTimes .. maximum bothTimes]
          tvMap = foldl insertMaybePair Map.empty (zip t1 v1)
          updatedMap = foldl insertMaybePair tvMap (zip t2 v2)
          combinedValues = map (`Map.lookup` updatedMap) completeTimes

instance Semigroup (TS a) where
    (<>) = combineTS

instance Monoid (TS a) where
    mempty = TS [] []
    mappend = (<>)

mean :: (Real a) => [a] -> Double
mean xs = total/count
    where total = (realToFrac . sum) xs
          count = (realToFrac . length) xs

meanTS :: (Real a) => TS a -> Maybe Double
meanTS (TS _ []) = Nothing
meanTS (TS times values) = if all (== Nothing) values
    then Nothing
    else Just avg
    where justVals = filter isJust values
          cleanVals = map fromJust justVals
          avg = mean cleanVals

type CompareFunc a = a -> a -> a
type TSCompareFunc a = (Int, Maybe a) -> (Int, Maybe a) -> (Int, Maybe a)

makeTSCompare :: Eq a => CompareFunc a -> TSCompareFunc a
makeTSCompare func = newFunc
    where newFunc (i1, Nothing) (i2, Nothing) = (i1, Nothing)

```

```

newFunc (_, Nothing) (i, val) = (i, val)
newFunc (i, val) (_, Nothing) = (i, val)
newFunc (i1, Just val1) (i2, Just val2) = if func val1 val2 == val1
                                         then (i1, Just val1)
                                         else (i2, Just val2)

compareTS :: Eq a => (a -> a -> a) -> TS a -> Maybe (Int, Maybe a)
compareTS func (TS [] []) = Nothing
compareTS func (TS times values) = if all (== Nothing) values
                                   then Nothing
                                   else Just best

    where pairs = zip times values
          best = foldl (makeTSCompare func) (0, Nothing) pairs

minTS :: Ord a => TS a -> Maybe (Int, Maybe a)
minTS = compareTS min

maxTS :: Ord a => TS a -> Maybe (Int, Maybe a)
maxTS = compareTS max

diffPair :: Num a => Maybe a -> Maybe a -> Maybe a
diffPair Nothing _ = Nothing
diffPair _ Nothing = Nothing
diffPair (Just x) (Just y) = Just (x - y)

diffTS :: Num a => TS a -> TS a
diffTS (TS [] []) = TS [] []
diffTS (TS times values) = TS times (Nothing : diffValues)
    where shiftValues = tail values
          diffValues = zipWith diffPair shiftValues values

meanMaybe :: (Real a) => [Maybe a] -> Maybe Double
meanMaybe vals = if Nothing `elem` vals
                  then Nothing
                  else Just avg
    where avg = mean (map fromJust vals)

movingAvg :: (Real a) => [Maybe a] -> Int -> [Maybe Double]
movingAvg [] n = []

```

```

movingAvg vals n = if length nextVals == n
                    then meanMaybe nextVals : movingAvg restVals n
                    else []
  where nextVals = take n vals
        restVals = tail vals

movingAverageTS :: (Real a) => TS a -> Int -> TS Double
movingAverageTS (TS [] []) n = TS [] []
movingAverageTS (TS times values) n = TS times smoothedValues
  where ma = movingAvg values n
        nothings = replicate (n `div` 2) Nothing
        smoothedValues = mconcat [nothings, ma, nothings]

-- median находит медиану в TS
median :: (Real a, Ord a) => [a] -> Double
median xs
  | even len  = (realToFrac a + realToFrac b) / 2
  | otherwise = realToFrac (sorted !! mid)
  where
    xs' = map realToFrac xs
    sorted = sort xs'
    len = length xs
    mid = len `div` 2
    a = sorted !! (mid - 1)
    b = sorted !! mid

medianMaybe :: (Real a, Ord a) => [Maybe a] -> Maybe Double
medianMaybe vals =
  if Nothing `elem` vals
  then Nothing
  else Just (median (map fromJust vals))

movingMedian :: (Real a, Ord a) => [Maybe a] -> Int -> [Maybe Double]
movingMedian [] _ = []
movingMedian vals n =
  if length nextVals == n
  then medianMaybe nextVals : movingMedian restVals n
  else []
  where
    nextVals = take n vals

```

```

restVals = tail vals

movingMedianTS :: (Real a, Ord a) => TS a -> Int -> TS Double
movingMedianTS (TS [] []) _ = TS [] []
movingMedianTS (TS times values) n = TS times smoothedValues
  where
    mm = movingMedian values n
    pad = replicate (n `div` 2) Nothing
    smoothedValues = mconcat [pad, mm, pad]

-- ratioPair считает отношение между парой в TS
ratioPair :: (Fractional a, Eq a) => Maybe a -> Maybe a -> Maybe a
ratioPair Nothing _ = Nothing
ratioPair _ Nothing = Nothing
ratioPair (Just x) (Just y) =
  if y == 0 then Nothing else Just (x / y)

-- ratioTS аналог diffTS, но вычисляется как отношение, а не разность
ratioTS :: (Fractional a, Eq a) => TS a -> TS a
ratioTS (TS [] []) = TS [] []
ratioTS (TS times values) = TS times (Nothing : ratioValues)
  where
    shiftValues = tail values
    ratioValues = zipWith ratioPair shiftValues values

-- stdDev вычисляет стандартное отклонение
stdDev :: (Real a) => [a] -> Double
stdDev xs = sqrt (mean squaredDeviations)
  where
    avg = mean xs
    deviations = map (\x -> realToFrac x - avg) xs
    squaredDeviations = map (**2) deviations

-- stdDevTS вычисляет стандартное отклонение для TS
stdDevTS :: (Real a) => TS a -> Maybe Double
stdDevTS (TS _ []) = Nothing
stdDevTS (TS _ values) =
  if all (== Nothing) values
  then Nothing
  else Just (stdDev cleanVals)

```

```

where
    cleanVals = catMaybes values

-- timeRange получает полную временную линию, включающую две последовательности
timeRange :: TS a -> TS a -> [Int]
timeRange (TS t1 _) (TS t2 _) =
    if null t1 && null t2
    then []
    else [minT .. maxT]
where
    minT = minimum (filter (const True) (t1 ++ t2))
    maxT = maximum (t1 ++ t2)

-- resample преобразует TS в заданную временную линию
resample :: [Int] -> TS a -> [Maybe a]
resample timeline (TS times values) = map lookupTime timeline
    where
        m = Map.fromList (zip times values)
        lookupTime t = Map.findWithDefault Nothing t m

-- zipTSWith соединяет два временных ряда, используя заданную функцию
zipTSWith :: (a -> a -> a) -> TS a -> TS a -> TS a
zipTSWith f ts1 ts2 = TS fullTimes resultVals
    where
        fullTimes = timeRange ts1 ts2
        vals1 = resample fullTimes ts1
        vals2 = resample fullTimes ts2
        resultVals = zipWith combine vals1 vals2
        combine (Just x) (Just y) = Just (f x y)
        combine _ _ = Nothing

addTS = zipTSWith (+)
subTS = zipTSWith (-)

```

## Оценка времени, затраченного на выполнение

На выполнение заданий ушло около 6 часов, из которых примерно 3 часа было потрачено на освоение специфики языка, а оставшиеся 3 — непосредственно на написание и отладку кода в заданиях.

## **Оценка времени, затраченного на выполнение**

Материал оказался понятным, сложность вызвал этап перехода от императивного мышления к чисто функциональному.

## **Выводы о наиболее сложных на текущей стадии аспектах языка**

- Система классов типов и их законы — особенно при реализации собственных экземпляров (например, `SemiGroup`, `Monoid`). Требуется не только написать код, но и соблюсти математические законы (ассоциативность, нейтральный элемент и т.д.).
- Работа с полиморфизмом через классы типов (в отличие от интерфейсов в Go или `duck typing` в Python).

## **Аналогии с другими знакомыми средствами разработки программ**

- `Maybe` монада подобна указателю на пустой элемент в других языках программирования, но куда более выразительна и безопасна.
- `Enum`'ы подобным перечислениям в других языках программирования.
- Классы типов (`typeclass`) — это нечто среднее между интерфейсами в Go (но с возможностью реализации «извне») и протоколами/абстрактными базовыми классами в Python (но с компиляторной проверкой).

**Вывод:** в процессе выполнения лабораторной работы изучена идеология типизации данных в Haskell, получены навыки проектирования типов данных приложения.