Лабораторная работа №2

Исследование методов обработки колекций и функций высших порядков в функциональном программировании

1. Цель работы

Исследовать способы реализации и обработки кортежей, массивов и коллекций. Реализовать и исследовать функции высших порядков для обработки колекций в языке Scala

2. Основные положения

2.1. Кортежи

Кортежи (tuples) – это упорядоченный набор фиксированной длины. Простейшим случаем кортежа является пара.

В Scala, как и во всех функциональных языках программирования имеются кортежи. Элементы кортежа могут иметь отличные друг от друга типы. Например:

```
val cortezh = (7, 2.4, “Scala”, true)
```

Значение cortezh имеет тип Tuple4[Int, Double, String, Boolean] или упрощённо (Int, Double, String, Boolean), причём скобки являются частью типа.

Обратиться к элементам кортежа можно следующим образом:

```
cortezh._1    // вернёт 7
cortezh._2    // вернёт 2.4
cortezh._3    // вернёт “Scala”
cortezh._4    // вернёт true
```

Нумерация элементов кортежа начинается с 1. Но обращаться с элементами кортежа выше обозначенным образом не удобно, поэтому можно применить механизм сопоставления с образцом (о том, что это такое будет рассказано в дальнейшей лабораторной работе):

```
val (chislo, drobnoe, stroka, bulevskoe) = cortezh
println(chislo)       // напечатает 7
println(drobnoe)      // напечатает 2.4
println(stroka)       // напечатает “Scala”
println(bulevskoe)    // напечатает true
```

Как видно из примера выше, можно извлечь значения из кортежа и присвоить их отдельным переменным. Если значение какого-то элемента кортежа не нужно извлекать, вместо него можно поставить прочерк:

```
val (chislo, _, _, bulevskoe) = cortezh
```

В результате такого вызова будут созданы только 2 переменные (со значениями 7 и true соответственно).

С помощью кортежей удобно возвращать несколько значений из функции.

Пример функции, которая возвращает сумму и разность своих аргументов в виде пары:

```
def sumDif(a: Int, b: Int): (Int, Int) = {
  (a+b, a-b)
}

// или еще короче
def sumDif(a: Int, b: Int) = (a+b, a-b)
```

2.2. Ассоциативные массивы

Ассоциативные массивы (Map) – это коллекция пар «ключ-значение». Создать экземпляр ассоциативного массива в языке Scala можно следующим образом:

```
val prices = new scala.collection.mutable.Map(
  (“laptop”, 30000.0),
  (“smartphone”, 20000.0)
)
```

ИЛИ

```
import scala.collection.mutable.Map

val prices = Map(
  (“laptop”, 30000.0),
  (“smartphone”, 20000.0)
)
```

Если необходимо создать пустой ассоциативный массив, необходимо указать типы ключей и значений:

```
val pricesEmpty = scala.collection.mutable.Map[String, Double]
```

Небольшое отступление по поводу mutable и immutable. В парадигме функционального программирования все значения являются неизменяемыми. Поэтому, например, когда необходимо изменить какое-то значение внутри списка, ассоциативного массива, или любой другой коллекции данных, меняется не одно это значение внутри коллекции, а создаётся новая коллекция с обновлённым соответствующим элементом. На первый взгляд, создавать каждый раз новую коллекцию при изменениях это крайней неэффективно, но новые и старые коллекции разделяют одинаковые значения (ссылаются на один и те же объекты в памяти). Это безопасно делать как раз из-за неизменяемости данных.

Но Scala так же поддерживает парадигму императивного программирования, поэтому у всех стандартных коллекций существует две реализации: изменяемые (mutable) и неизменяемые (immutable), которые содержаться в пакетах scala.collection.mutable и scala.collection.immutable соответственно.

Различие изменяемых и неизменяемых коллекций можно продемонстрировать следующий образом:

```
val immutableQueue = new Queue[Int]  // неизменяемая очередь целых чисел
val newQueue = immutableQueue.enqueue(2)  // добавит элемент в
// очередь и вернёт новую очередь.
// immutableQueue останется без изменений

val mutableQueue = new scala.collection.mutable.Queue[Int]
mutableQueue.enqueue(2) // добавит элемент в очередь, сам метод вернёт
// ничего (Unit). mutableQueue будет содержать изменённую очередь
```

По-умолчанию используются неизменяемые коллекции, поэтому, если необходимо использовать изменяемые коллекции – необходимо явно это указывать, как на примерах выше.

Для создания пары (кортежа из двух значений) можно также использовать следующий синтаксис:

```
val pair = “notepad” -> 200.0
// pair имеет значение (“notepad”, 200.0) типа (String, Double)
```

Таким образом, инициализация ассоциативного массива может выглядеть более естественно:

```
val anotherPrices = Map(
  “Table” -> 2300.0,
  “Pencil” -> 12.0,
  “Bread” -> 17.0
)
```

Об ассоциативном массиве можно думать как о частично определённой функции (частично – значит не для любого входного значения). Таким образом, получить значение по ключу можно следующим образом:

```
anotherPrices(“Table”)    // вернёт 2300.0
```

Если попробовать получить значение по ключу, который отсутствует в ассоциативном массиве, то будет возбуждено исключение. Для проверки наличия ключа можно использовать метод contains, например:

```
anotherPrices.contains(“Pencil”)    // true
```

```
anotherPrices.contains(“Snickers”)  // false
```

Для получения значения по-умолчанию в случае отсутствия ключа можно использовать следующий метод:

```
anotherPrices.getOrElse(“Pencil”, 0.0)    // вернёт 12.0
anotherPrices.getOrElse(“Snickers”, 0.0)  // вернёт 0.0
```

2.3. Изменение ассоциативных массивов

2.3.1. Изменяемые (mutable) ассоциативные массивы

Добавить новые значения или изменить уже существующие можно следующим образом:

```
anotherPrices(“Table”) = 3000.0  // изменение существующего значения
anotherPrices(“Paper”) = 500.0   // добавление нового значения
```

Для добавления сразу нескольких пар ключ-значение можно исопльзовать оператор +=

```
anotherPrices += (“Pen” -> 50.0, “Cookie” -> 320.0)
```

Удалить ключ и значение можно с помощью оператора -=

```
anotherPrices -= “Pen”
```

2.3.2. Неизменяемые (immutable) ассоциативные массивы

Как уже упоминалось ранее, для изменение неизменяемой коллекции необходимо создавать новую коллекцию. Следовательно, произвести необходимые изменения с неизменяемым ассоциативным массивом можно следующим образом:

```
val newAnotherPrices = anotherPrices + (“Paper” -> 230.0, “Eggs” -> 50.0)
```

newAnotherPrices содержит все значения из anotherPrices, кроме значения по ключу Paper (измененное значение) + он содержит новое добавленное значение по ключу Eggs.

Аналогично, удаление по ключу происходит следующим образом:

```
val pricesWithoutEggs = newAnotherPrices – “Eggs”
```

2.4. Метод mkString

На замену метода toString из Java, коллекции в Scala содержат более удобный метод для преобразования в строку mkString. Его использование лучшего всего показать на примере:

```
val langs = List("Scala", "Java", "C#")
langs.mkString                  // вернёт "ScalaJavaC#"
langs.mkString(", ")            // вернёт "Scala, Java, C#"
langs.mkString("[", "|", "]")   // вернёт "[Scala|Java|C#]"
```

2.5. for-генератор

В Scala отсутствует привычный цикл for из языков с C-подобным синтаксисом. Взамен имеется оператор for с более богатыми возможностями.

В заголовке оператора for указываются генераторы в форме *переменная <– коллекция*. Например:

```
for (i <- 1 to 10) println(i)   // напечатает числа от 1 до 10
```

В данном виде оператор for напоминает foreach – для каждого значения из коллекции выполняет какую-то операцию. Но на этом его возможности не ограничиваются. К генераторам в заголовке можно применять ограничения. Например:

```
val lst = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
for (i <- lst if i % 2 != 0) println(i)  // печатает нечётные числа
```

Также в заголовке for может быть несколько генераторов:

```
// объявим двумерный массив (массив массивов)
val arr = Array(Array(1, 2, 3), Array(4, 5, 6), Array(7, 8, 9))

for (
  i <- arr;  // в переменной i – вложенные массивы из arr
  j <- i     // в переменной j – значения элементов массива i
) println(j) // в итоге напечатаются все числа из arr

// то же самое но с привычными индексами массива
for (
  i <- 1 to 3;
  j <- 1 to 3
) println( arr(i)(j) )

// получение элементов на главной диагонали
for (
  i <- 1 to 3;
  j <- 1 to 3
  if i == j
) println( arr(i)(j) )
```

Стоит отметить, что, когда в заголовке оператора for содержаться несколько генераторов, они исполняются начиная с конца. В последнем примере выше сначала i получит значение 1, потом j последовательно получит значения от 1 до 3, потом i получит значение 2, j снова будет получать значения от 1 до 3 и т.д.

Пример обхода значений ассоциативного массива с помощью оператора for:

```
val assoc = Map(
  “first” -> 1
  “second” -> 2
)

for ((k, v) <- assoc) println(“по ключу ” + k + “ значение “ + v)

// в Scala есть интерполяция строк, поэтому можно переписать следующим образом
for ((k, v) <- assoc) println(s“по ключу $k значение $v”)
```

Рассмотренные ранее операторы for использовались как циклы в понимании императивного программирования, значение оператора было Unit. Однако оператор for способен возвращать значение, для этого нужно использовать дополнительный оператор yield. Например:

```
for (i <- 1 to 5) yield i*2 // вернёт Vector(2, 4, 6, 8, 10)
```

Такого рода операторы for называют for-генераторами.

Еще пример использования for-генератора:

```
val langs = List(“Scala”, “Java”, “C#”)

for (lang <- langs) yield {
  val howManyA = lang.count(_ == ‘a’) // считает количество букв a
  s”$lang have $howManyA a”
}
```

В результате получим следующий список строк:

```
List(
  “Scala have 2 a”,
  “Java have 2 a”,
  “C# have 0 a”
)
```

2.6. Анонимные функции (лямбда-функции)

В функциональных языках программирования функции являются такими же элементами языка как переменные и значения. Функции можно сохранять в переменных, передавать в виде параметров в другие функции и возвращать

функцию из функции. Для упрощения понимания этой концепции можно функцию представлять как ассоциативный массив, который по ключу (параметру функцию) выдает значение (возвращаемое значение из функции).

Лямбда-функция – функция, которая объявлена в месте своего использования (анонимная функция).

В языке Scala задать лямбда-функцию можно следующим образом:

```
val f = (x: Int) => x * 5
println( f(4) )    // напечатает 20
```

Справа от оператора = записано объявление анонимной функции (лямбда-выражение), которая сохраняется в значение f. При этом значение f имеет тип (Int) => Int, то есть функция, которая принимает один параметр типа Int и возвращает Int. Как видно из примера выше, если в значение сохранена анонимная функция, можно вызывать её по имени значения как обычную функцию.

Опишем лямбда-функцию, которая принимает два параметра и возвращает их сумму:

```
val s = (a: Int, b: Int) => a+b
println( s(1, 3) )    // напечатает 4
```

Значение s имеет тип (Int, Int) => Int. Рассмотрим анонимную функцию, которая принимает пару, состоящую из значений типа Int:

```
val p = (pair: (Int, Int)) => pair._1 + pair._2
println( p((5, 7)) )    // напечатает 12
```

Значение p имеет тип ((Int, Int)) => Int.

2.7. Функции высшего порядка

Функциями высшего порядка являются функции, которые в качестве одного из аргументов принимают функцию высшего порядка, либо в качестве результата возвращают функцию высшего порядка. Рассмотрим пример функции, которая принимает функцию и значение, к которому нужно применить данную функцию и возвращает полученный результат:

```
def highOrder(func: (Int) => Int, n: Int): Int = {
  func(n)
}

// Аналогично можно было бы объявить эту функцию через лямбда-выражение:
val ha = (func: (Int) => Int, n: Int) => func(n)
```

Рассмотрим вызов объявленной функции, для этого нужно в качестве параметров передать функцию, которая принимает Int и возвращает Int, и само значение типа Int:

```
println( highOrder((x: Int) => x * 2, 5) )  // напечатает 10
// то же самое со второй функцией
println( ha((x: Int) => x - 3, 7) )  // напечатает 4
```

Т.к. при объявлении функции highOrder мы указали, что первый параметр является функцией, отображающей значение типа Int в значение типа Int, можно опустить указание типа для значения x:

```
// до
println( highOrder((x: Int) => x * 2, 5) )  // напечатает 10

// после
println( highOrder((x) => x * 2, 5) )  // напечатает 10
```

Т.к. функция принимает только один параметр, можно опустить скобки вокруг этого параметра:

```
println( highOrder(x => x * 2, 5) )  // напечатает 10
```

В особо простых лямбда-выражениях можно опустить указание имени переменной и использовать символ _ . Например:

```
println( highOrder(_ * 2, 5) )  // напечатает 10
```

Для примера напишем функцию, похожую на предыдущую, но теперь будет применять функцию не к одному значению, а к списку:

```
def processList(lst: List[Int], func: (Int) => Int): List[Int] = {
  if (lst == Nil) Nil
  else func(lst.head) :: processList(lst.tail, func)
}

// примеры использования:
println(processList(1::2::3::Nil, _ * 5)) // напечатает List(5, 10, 15)

val myList = List(21, 3, 7, 87)
println(processList(myList, _ - 3)) // напечатает List(18, 0, 4, 84)
```

2.8. Встроенные функции высшего порядка для обработки коллекций.

Язык Scala содержит множество полезных функций (методов) высшего порядка для обработки коллекций. Рассмотрим некоторые из них.

2.8.1. map

Функция (а точнее метод) map применяет заданную функцию к каждому элементу коллекции и возвращает новую коллекцию с изменёнными элементами.

```
// Возведём каждый элемент коллекции в квадрат
(1 to 10).map(x => x * x)  // вернёт Vector(1, 4, 9, 16, 25, ..., 100)

// Обработка списка строк
val strings = List("is cool", "is amazing", "- just perfect")

strings.map("Scala " + _)  // вернёт список строк:
// List("Scala is cool", "Scala is amazing", "Scala - just perfect")
```

Если переданная функция вместо одного значения возвращает коллекцию, то получается следующий результат:

```
val numbers = (2 to 10 by 3).toList  // numbers = List(2, 5, 8)

numbers.map(x => (x-1 to x+1).toList)
// вернёт List(List(1, 2, 3), List(4, 5, 6), List(7, 8, 9))
// а хотелось бы List(1, 2, 3, 4, 5, 6, 7, 8, 9)
```

Чтобы не было вложенной коллекции, нужно использовать flatMap:

```
numbers.flatMap(x => (x-1 to x+1).toList)
// вернёт List(1, 2, 3, 4, 5, 6, 7, 8, 9)
```

2.8.2. filter

Метод filter принимает функцию, возвращающую булевское значение (предикат) и возвращает новую коллекцию, в которой содержатся элементы исходной коллекции удовлетворяющие предикату. Пример:

```
(1 to 10).filter(_ % 2 == 0) // вернёт коллекцию чётных элементов

strings.filter(_.length < 11) // вернёт List("is cool", "is amazing")
```

2.8.3. reduceLeft и reduceRight

Методы reduceLeft и reduceRight последовательно применяют указанную двуместную функцию (функцию с двумя параметрами) к элементами коллекции. Пример:

```
(1 to 5).reduceLeft(_ + _)
// то же что и ((((1 + 2) + 3) + 4) + 5)

(1 to 5).reduceRight(_ + _)
```

```
// то же что и (1 + (2 + (3 + (4 + 5))))

// на самом деле запись
// _ + _
// расшифровывается как
// (a, b) => a + b
```

2.8.4. foldLeft и foldRight

Методы foldLeft и foldRight работают так же, как и reduceLeft/reduceRight, за исключением того, что вычисление начинается с заданного значения. Пример:

```
(1 to 5).foldLeft(0)(_ + _)  // вернёт 15
// то же что и (((((0 + 1) + 2) + 3) + 4) + 5)

(1 to 5).foldLeft(10)(_ + _)  // вернёт 25
// то же что и (((((10 + 1) + 2) + 3) + 4) + 5)

(1 to 5).foldRight(-5)(_ + _)
// то же что и (1 + (2 + (3 + (4 + (5 + (-5))))))
```

Операция fold может служить заменой циклу. Например необходимо составить ассоциативный массив, где ключём является символ, а значением – количество вхождений символа в строку. Для начала решим это с помощью цикла:

```
val freq = scala.collection.mutable.Map[Char, Int]()
for (c <- “aggregate”) {
  freq(c) = freq.getOrElse(c, 0) + 1
}
// теперь freq содержит следующее:
// Map(
//   ‘e’ -> 2,
//   ‘t’ -> 1,
//   ‘g’ -> 3,
//   ‘a’ -> 2,
//   ‘r’ -> 1
// )
```

А теперь добьёмся того же результата с помощью foldLeft:

```
val freq2 = scala.collection.mutable.Map[Char, Int]()
“aggregate”.foldLeft(freq2) {
  (mapFreq, ch) => mapFreq + (ch -> (mapFreq.getOrElse(ch, 0) + 1))
}
// ВЕРНЁТ результат Map(
//   ‘e’ -> 2,
//   ‘t’ -> 1,
//   ‘g’ -> 3,
//   ‘a’ -> 2,
```

```
//   'r' -> 1
// )
// исходный freq2 останется без изменений
```

А теперь то же самое, но с foldRight:

```
val freq3 = scala.collection.mutable.Map[Char, Int]()
"aggregate".foldRight(freq3) {
  (ch, mapFreq) => mapFreq + (ch -> (mapFreq.getOrElse(ch, 0) + 1))
}
```

2.8.5. scanLeft и scanRight

Методы scanLeft и scanRight очень похожи на foldLeft/foldRight, но возвращают не конечный результат, а коллекцию промежуточных результатов операции (вместе с конечным). Пример:

```
(1 to 3).foldLeft(10)(_-_)  // вернёт 4
(1 to 3).scanLeft(10)(_-_)  // вернёт Vector(10, 9, 7, 4)

(1 to 3).foldRight(10)(_-_)  // вернёт -8
(1 to 3).scanRight(10)(_-_)  // вернёт Vector(-8, 9, -7, 10)
```

2.8.6. take и drop

Методы take и drop позволяют отбрасывать лишние элементы коллекции.

```
List(1, 2, 3, 4, 5).take(3)    // вернёт List(1, 2, 3)
List(1, 2, 3, 4, 5).drop(3)    // вернёт List(4, 5)
```

2.8.7. takeWhile, dropWhile, span

Методы takeWhile и dropWhile работают так же как и обычные методы take и drop, но в качестве параметра принимают не количество элементов, а предикат (условие). Пример:

```
val list = List(-3, -2, -1, 0, 1, 2, 3)
list.takeWhile(_ < 0)    // вернёт List(-3, -2, -1)
list.takeWhile(_ > 0)    // вернёт List() (пустой список)
list.dropWhile(_ < 0)    // вернёт List(0, 1, 2, 3)
list.dropWhile(_ > 0)    // вернёт List(-3, -2, -1, 0, 1, 2, 3)
```

Метод span совмещает в себе методы takeWhile и dropWhile и возвращает пару, содержащую две подколлекции исходной коллекции. Пример:

```
list.span(_ < 0)    // вернёт (List(-3, -2, -1), List(0, 1, 2, 3))
list.span(_ > 0)    // вернёт (List(), List(-3, -2, -1, 0, 1, 2, 3))
// span(predicate) = (takeWhile(predicate), dropWhile(predicate))
```

2.8.8. zip, zipAll, zipWithIndex

С помощью метода zip можно объединить две коллекции в одну коллекцию, состоящую из пару. Пример:

```
val names = List(“John”, “Sam”, “Mike”)
val phones = List(458965, 286349)

val zipped = names zip phones  // или names.zip(phones)
// zipped = List((“John”, 458965), (“Sam”, 286349))
```

Как видно из примера, длинная объединённой коллекции равна длине самой короткой коллекции. Если необходимо, чтобы объединённая коллекция содержала все значения, необходимо использовать метод zipAll и указать значения по-умолчанию для обоих коллекций:

```
val zipped2 = names.zipAll(phones, “NoName”, 0)
// zipped2 = List((“John”, 458965), (“Sam”, 286349) , (“Mike”, 0))

// ИЛИ

val zipped3 = List[String]().zipAll(phones, “NoName”, 0)
// zipped3 = List((“NoName ”, 458965), (“NoName ”, 286349))
```

Если для обработки элементов коллекции необходимо иметь дело не только с элементом, но и с индексом элемента, в алгоритме можно использовать метод zipWithIndex. Для примера получим из коллекции элементы, которые стоят на чётных индексах:

```
val data = List(“One”, “Two”, “Three”, “Four”, “Five”)
val withIndex = data.zipWithIndex
// withIndex = List(
//   (“One”, 0),
//   (“Two”, 1),
//   (“Three”, 2),
//   (“Four”, 3),
//   (“Five”, 4)
// )

val evenElemsWithIndex = withIndex.filter({case (s, n) => n % 2 == 0})
// ИЛИ val evenElemsWithIndex = withIndex.filter(p => p._2 % 2 == 0)
// evenElemsWithIndex = List(
//   (“One”, 0),
//   (“Three”, 2),
//   (“Five”, 4)
```

```
// )
```

Обратите внимание на два варианта записи анонимной функции, которую мы передаём в метод filter. В первом случае мы с помощью механизма сопоставления с образцом (подробнее о нём будет рассказано в следующих лабораторных работах) «раскрываем» пару, содержащуюся в списке и присваиваем её элементам имена s (для строки) и n (для индекса) и проверяем условие делимости индекса на 2 без остатка. Во втором случае мы не используем сопоставление с образцом и работаем с парой как с целым значением, извлекая индекс с помощью метода _2 и так же проверяем условие делимости на 2 без остатка.

Мы получили список нужных элементов, но это список пар, в котором, помимо самих элементов, остались индексы. У методов zip/zipAll/zipWithIndex есть обратный метод unzip. Он принимает список пар и возвращает пару списков. Избавимся от индексов с помощью метода unzip:

```
val pairOfLists = evenElemsWithIndex.unzip
// pairOfLists = (List( “One”, “Three”, “Five”), List(0, 2, 4))
// Для получения списка элементов достаточно взять первый элемент пары
val result = pairOfLists._1
// result = List( “One”, “Three”, “Five”)
```

3. Задание на лабораторную работу

3.1. Переписать функцию processList из пункта 2.6. с использованием хвостовой рекурсии.
3.2. Написать функцию типа (List[Int]) => List[String], которая преобразует число в строку «Элемент под номером **индекс_элемента** равен **значение_элемента**».
3.3. На основе своего варианта, выданного преподавателем, и используя данные из Приложения А написать функции:

a. Вариант 1 (филология):
    i. Написать функцию, которая возвращает список, содержащий имя и курс всех студентов факультета филологии, старше 93 года.
    ii. Написать функцию, которая возвращает список, содержащий имя ID и номер комнаты студентов факультета филологии, проживающих в одной комнате.
b. Вариант 2 (АВТ)
    i. Написать функцию, которая возвращает список, содержащий год рождения всех студентов факультета АВТ, проживающих в общежитии.

ii. Написать функцию, которая возвращает список, содержащий имя курс и номер комнаты всех студентов факультета АВТ, проживающих в соседних комнатах в общежитии.

c. Вариант 3 (МТС)

i. Написать функцию, которая возвращает список, содержащий имя и статус проживания в общежитии всех студентов факультета МТС женского пола.

ii. Написать функцию, которая возвращает список, содержащий ID, курс и номер комнаты общежития всех студентов факультета МТС, проживающих в двуместной комнате.

3.4. Самостоятельно изучить метод groupBy и для своего факультета (в зависимости от варианта)

Приложение А

Вспомогательный код к выполнению лабораторной работы

```
object Application {
  def main (args: Array[String]): Unit = {
    type Student = (
      Int,     // ID
      String,  // Имя
      Int,     // Год рождения
      String,  // Факультет
      Char,    // Пол
      Int,     // Курс
      Boolean  // Проживает ли в общежитии
    )

    val sudents: List[Student] = List(
      (0, "Алёна", 1995, "FIL", 'F', 1, true),
      (1, "Гриша", 1994, "AVT", 'M', 2, true),
      (2, "Настя", 1993, "MTS", 'F', 3, false),
      (3, "Коля", 1997, "MTS", 'M', 1, false),
      (4, "Миша", 1997, "AVT", 'M', 3, true),
      (5, "Оля", 1992, "FIL", 'F', 3, false),
      (6, "Маша", 1991, "AVT", 'F', 5, true),
      (7, "Таня", 1993, "FIL", 'M', 4, true),
      (8, "Женя", 1992, "FIL", 'F', 4, true),
      (9, "Света", 1989, "AVT", 'F', 3, true),
      (10, "Аня", 1996, "MTS", 'F', 4, false),
      (11, "Лена", 1996, "AVT", 'F', 2, true),
      (12, "Сергей", 1994, "FIL", 'M', 3, false),
      (13, "Влад", 1993, "FIL", 'M', 5, false),
      (14, "Гена", 1996, "MTS", 'M', 1, true),
      (15, "Дима", 1995, "AVT", 'M', 5, false),
      (16, "Катя", 1991, "FIL", 'F', 4, false),
      (17, "Артём", 1994, "MTS", 'M', 3, true),
      (18, "Диана", 1995, "FIL", 'M', 4, false)
    )

    type Room = (
      Int,       // Номер комнаты
      Int,       // Вместимость комнаты
      List[Int]  // ID студентов, проживающих в комнате
    )

    val rooms: List[Room] = List(
      (37, 3, List(0, 7, 8)),
      (42, 2, List(1, 4)),
      (43, 3, List(6, 9, 11)),
      (54, 2, List(14, 17))
    )
  }
}
```