МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное
учреждение высшего образования
«СЕВАСТОПОЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

ПАРАДИГМЫ СОВРЕМЕННЫХ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ

Методические указания
к выполнению лабораторного практикума по дисциплине
для магистрантов направления подготовки
09.04.02 – «Информационные системы и технологии»
профиля «Интеллектуальные информационные системы»
всех форм обучения

Севастополь
СевГУ
2024

УДК 004.43:519.682(076.5)
ББК 32.973-018.1я73
П180

Р е ц е н з е н т ы :
В. С. Чернега – канд. техн. наук, доцент кафедры информационных систем
Т. В. Волкова – канд. техн. наук, доцент кафедры информационных технологий и компьютерных систем

Ответственный за выпуск:
И. П. Шумейко – канд. физ.-мат. наук, доцент заведующий кафедрой информационных систем

Составитель: В.А. Строганов

П180 Парадигмы современных языков программирования : методические указания к выполнению лабораторного практикума по дисциплине для магистрантов направления подготовки 09.04.02 – «Информационные системы и технологии» профиля «Интеллектуальные информационные системы» всех форм обучения / Севастопольский государственный университет ; сост. В. А. Строганов. – Севастополь : СевГУ, 2024. – 78 с. – Текст : электронный.

Цель методических указаний: обеспечить выполнение лабораторных работ, позволяющих на практике освоить способы реализации основных механизмов функционального программирования на примере языка Scala.

УДК 004.43:519.682(076.5)
ББК 32.973-018.1я73

Рассмотрены и утверждены на заседании кафедры информационных систем, протокол № 8 от 16 апреля 2024 г.

Рекомендованы к изданию на заседании Учёного совета Института информационных технологий, протокол № 5 от 18 апреля 2024 г. года.

Содержание

ВВЕДЕНИЕ

Современные языки программирования и подходы к созданию программных продуктов становятся всё более мультипарадигменными. После многолетнего господства объектно-ориентированной парадигмы на первый план выходят альтернативные подходы, в частности — функциональное программирование.

Парадигма программирования — это совокупность идей и понятий, определяющих стиль написания компьютерных программ (подход к программированию). Это способ концептуализации, определяющий организацию вычислений и структурирование работы, выполняемой компьютером. Другими словами, парадигмы программирования представляют собой основные концепции и подходы, которые определяют стиль написания программного кода. Парадигмы представляют собой набор принципов, методов и практик, которые определяют, каким образом программисты могут решать задачи с использованием компьютеров. Каждая парадигма программирования предлагает свой уникальный подход к решению проблем и организации кода. Понимание различий между этими парадигмами помогает программистам выбирать наиболее подходящий подход для конкретной задачи и создавать более эффективный и читаемый код. Основные группы парадигм различаются по тому, что именно описывает программа:

1) **декларативная парадигма** описывает, ЧТО должен вычислить компьютер;

2) императивная парадигма описывает, КАК он должен это вычислить.

Функциональная парадигма программирования относится к группе декларативных парадигм программирования. В функциональной парадигме программа представляет собой набор функций, которые принимают аргументы и возвращают результаты. Основное отличие **функциональной парадигмы** от процедурной и других императивных заключается в том, что в функциональной парадигме данные рассматриваются как неизменяемые (immutable), а изменения состояния программы минимизируются или отсутствуют вовсе. Это делает функциональное программирование более предсказуемым и устойчивым к ошибкам, так как отсутствие изменяемого состояния упрощает отладку и тестирование программ. Функции в функциональном программировании рассматриваются как математические функции, которые не имеют состояния и не изменяют свои аргументы. Это позволяет писать более декларативный и выразительный код, который легче читать и поддерживать.

Еще одним отличием функциональной парадигмы от процедурной является использование рекурсии вместо циклов. В функциональном

программировании рекурсия является основным механизмом повторения, что делает код более компактным и выразительным.

Предлагаемые лабораторные работы дают возможность на практике освоить основные концепции функционального программирования и получить опыт разработки на современном, активно развивающемся языке программирования Scala.

Курс включает шесть лабораторных работ. В первой работе изучаются основные особенности синтакстиса языка Scala и исследуются рекурсивные алгоритмы, в частности — сравниваются характеристики обычной и хвостовой рекурсии. Вторая работа посвящена исследованию функций высших порядков как наиболее часто используемого механизма функционального программирования. В третьей работе рассматриваются особенности использования классов и объектов в языке Scala. Четвёртая работа знакомит студента с алгоритмами сопоставления с образцом и возможностями их реализации на языке Scala. Пятая работа даёт возможность изучить и исследовать такие механизмы функционального программирования, как каррирование, частичное применения функций и замыкание. Шестая работа погружает студента в «высшую математику» функционального программирования: исследуются ковариативные, контравариативные и инвариативные функторы и возможности их реализации на языке Scala.

Для выполнения лабораторных работ рекомендуется использовать среду разработки IntelliJ IDEA. Порядок установки и настройки среды разработки приведён ниже.

По итогам выполнения каждой лабораторной работы оформляется отчет, включающий следующие разделы:

1. Цель работы;
2. Задание на работу;
3. Исходный код программы;
4. Результаты выполнения программы;
5. Выводы по результатам работы.

Оформленный отчет представляется к защите, которая проходит в форме устного опроса по материалам лабораторной работы.

ОПИСАНИЕ ЛАБОРАТОРНОЙ УСТАНОВКИ И МЕТОДИЧЕСКИЕ РЕКОМЕНДАЦИИ ПО РАБОТЕ СО СРЕДОЙ РАЗРАБОТКИ

Лабораторная установка состоит из персонального компьютера, на котором установлена и настроена среда разработки IntelliJ IDEA Community Edition. Ниже приведен порядок установки и настройки системы.

1. Установка Java Development Kit

Для выполнения лабораторных работ понадобится установленный Java Development Kit (JDK). Если он еще не установлен на компьютере, то скачать установщик можно на официальном сайте компании Oracle (http://www.oracle.com/technetwork/java/javase/downloads/index.html).
Актуальная версия JDK на момент написания данных методических указаний – Java SE Development Kit 8u102 (jdk1.8.0_102).

Внимание: Для выполнения лабораторных работ необходим именно **Java Development Kit (JDK)**, а не Java Runtime Enviroment (JRE)!

2. Установка среды разработки IntelliJ IDEA Community Edition

IntelliJ IDEA Community Edition является бесплатной средой разработки для языков семейства Java (и не только). Скачать ее можно бесплатно на официальном сайте компании IntelliJ (https://www.jetbrains.com/idea/#chooseYourEdition).

Рассмотрим подробно процесс установки IntelliJ IDEA. После запуска установочного файла, появится начальное окно установки (рисунок 1).

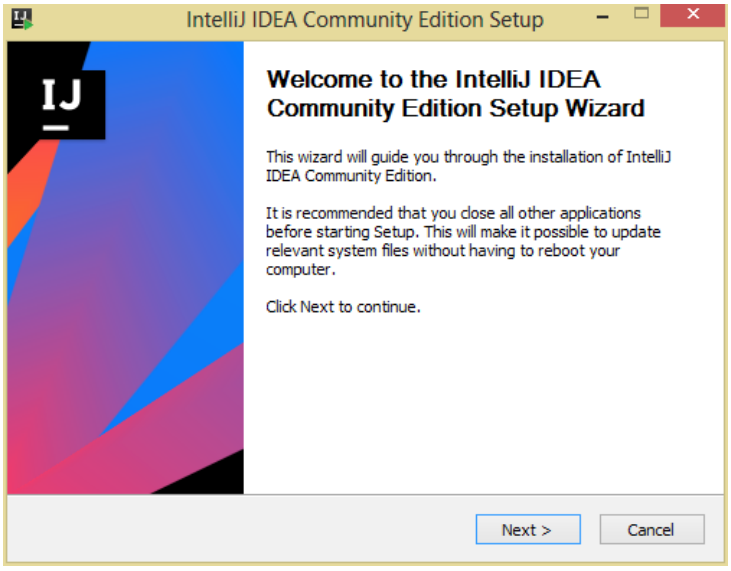

Рисунок 1 – Начальное окно установки IntelliJ IDEA

Нажимаем «Далее» для продолжения установки.

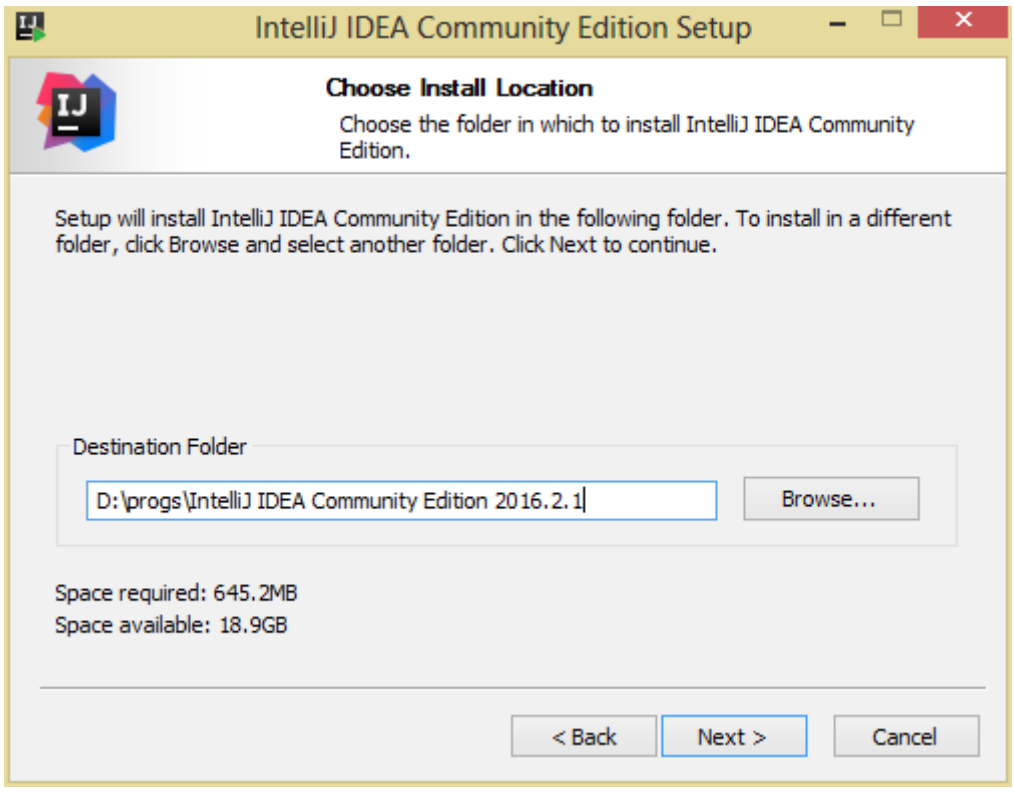

Рисунок 2 – Выбор пути установки

Выбираем путь установки (Рисунок 2) и нажимаем «Далее».

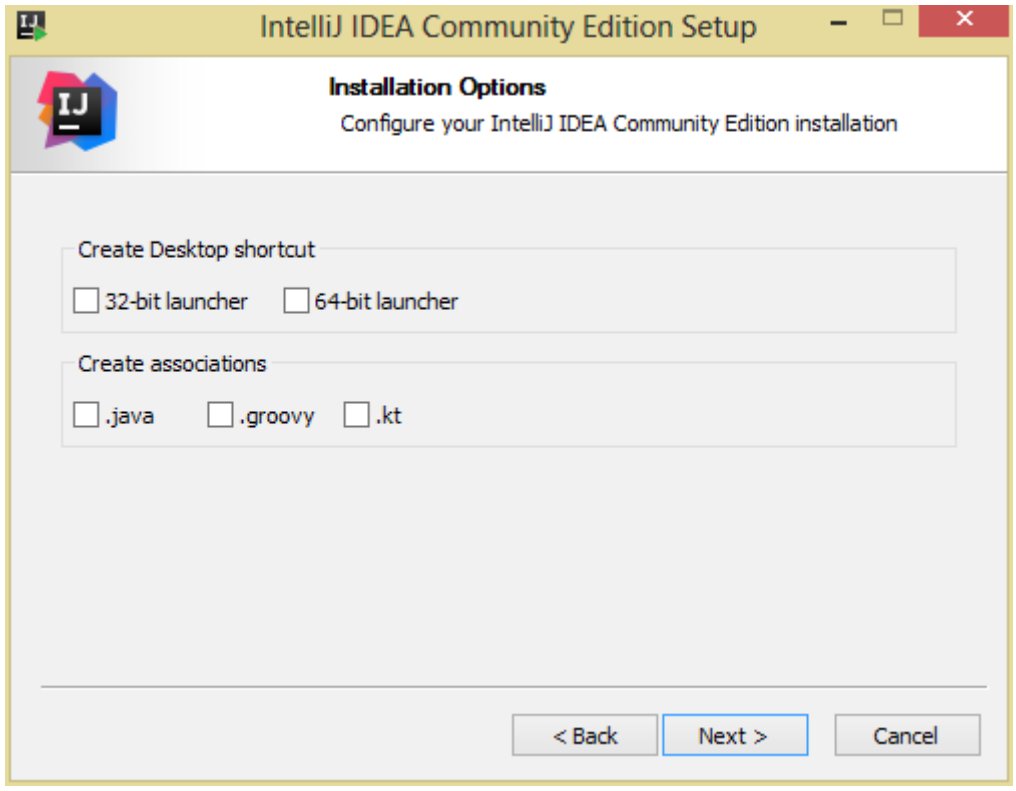

Рисунок 3 – Дополнительные настройки

Выбираем дополнительные настройки (рисунок 3). Если не знаете или не уверены, что надо выбирать, а что нет – поставьте галочки во всех чекбоксах.

Затем еще несколько раз нажимаем кнопку «Далее», пока не завершится установка.

2.3. Первый запуск и начальные настройки

При запуске IntelliJ IDEA впервые после установки, вы увидите следующее окно (рисунок 4).

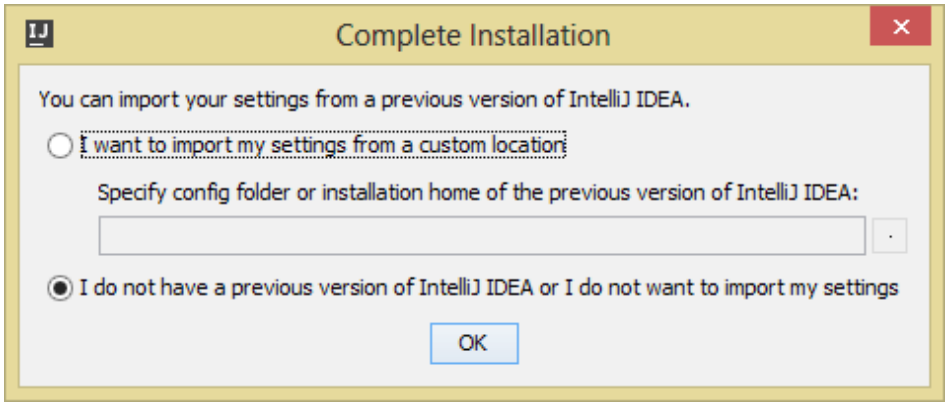

Рисунок 4 – Импорт настроек после установки

Выбираем нижний пункт и нажимаем «Ок». Далее соглашаемся с соглашением политики конфиденциальности (рисунок 2.5).

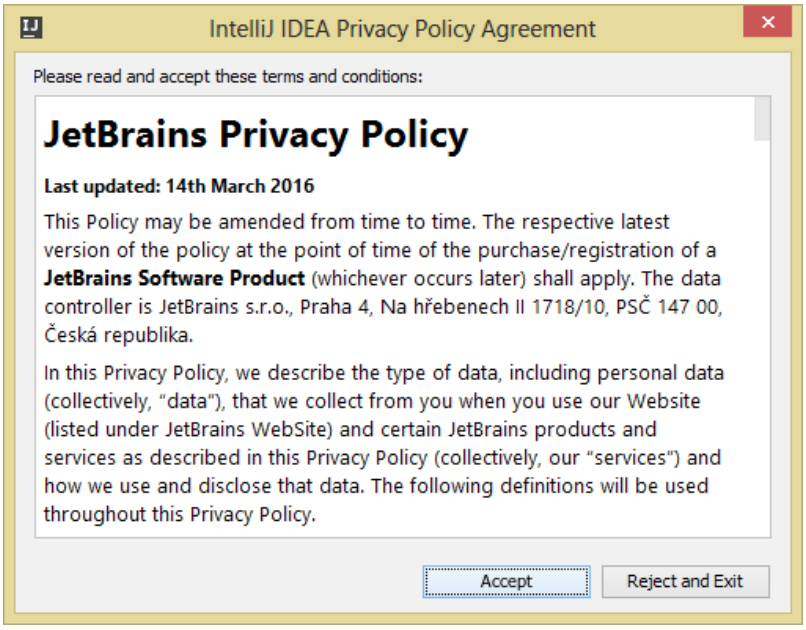

Рисунок 5 – Соглашение политики конфиденциальности.

Затем выбираем визуальную тему по вкусу и нажимаем нижнюю правую кнопку (на рисунке 6 это кнопка «Next: Default plugins»).

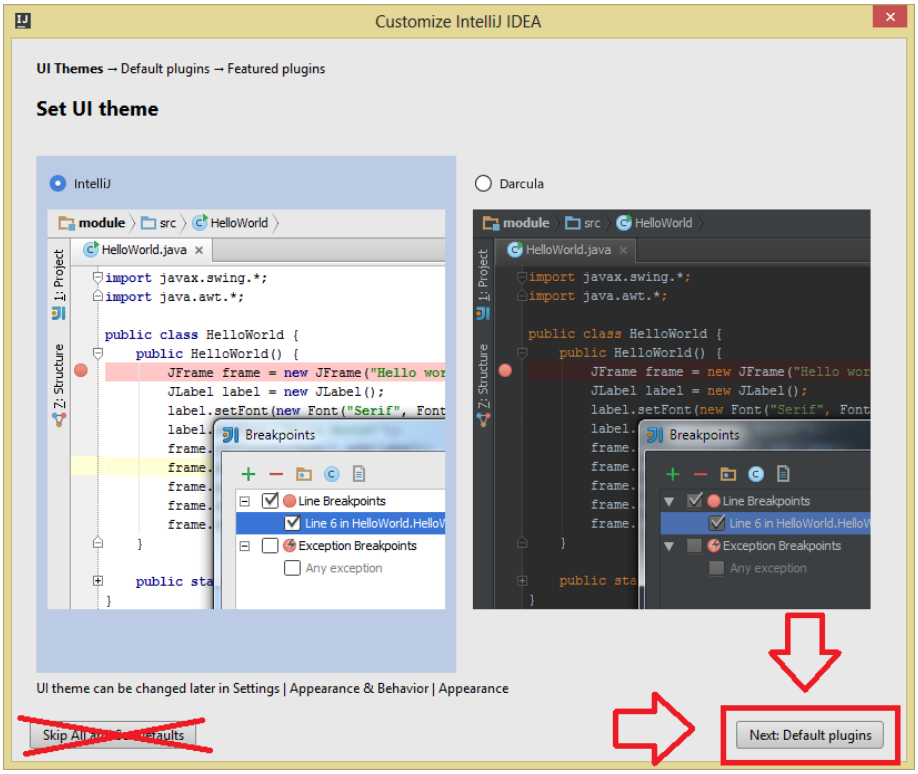

Рисунок 6 – Выбор визуальной темы

На следующем экране (рисунок 7) оставляем все как есть и опять нажимаем нижнюю правую кнопку.

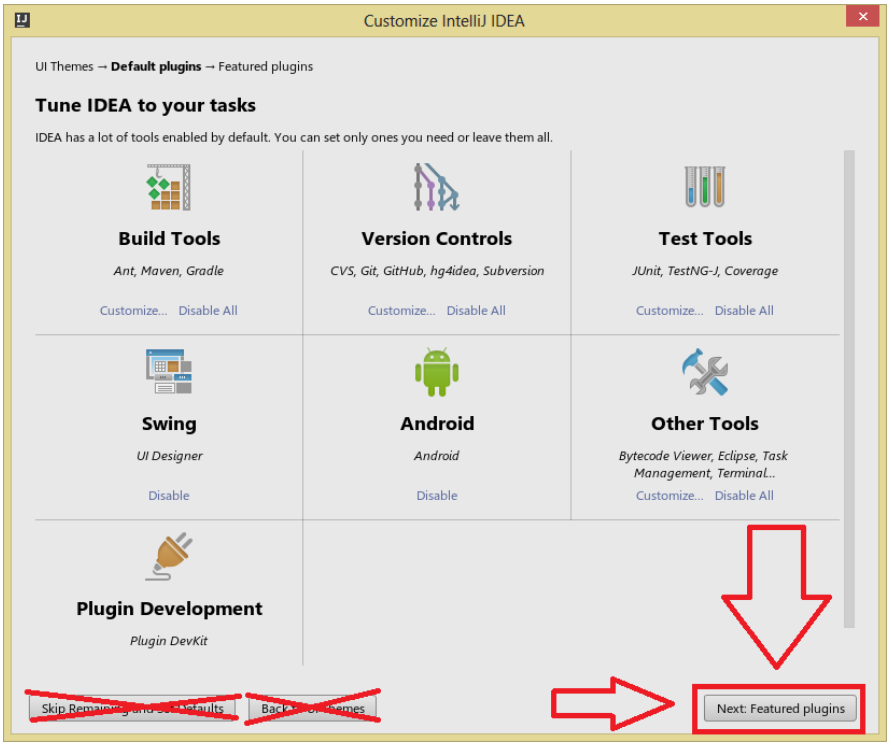

Рисунок 7 – Различные настройки

На следующем экране (рисунок 8) нажимаем кнопку для установки плагина для поддержки языка Scala.

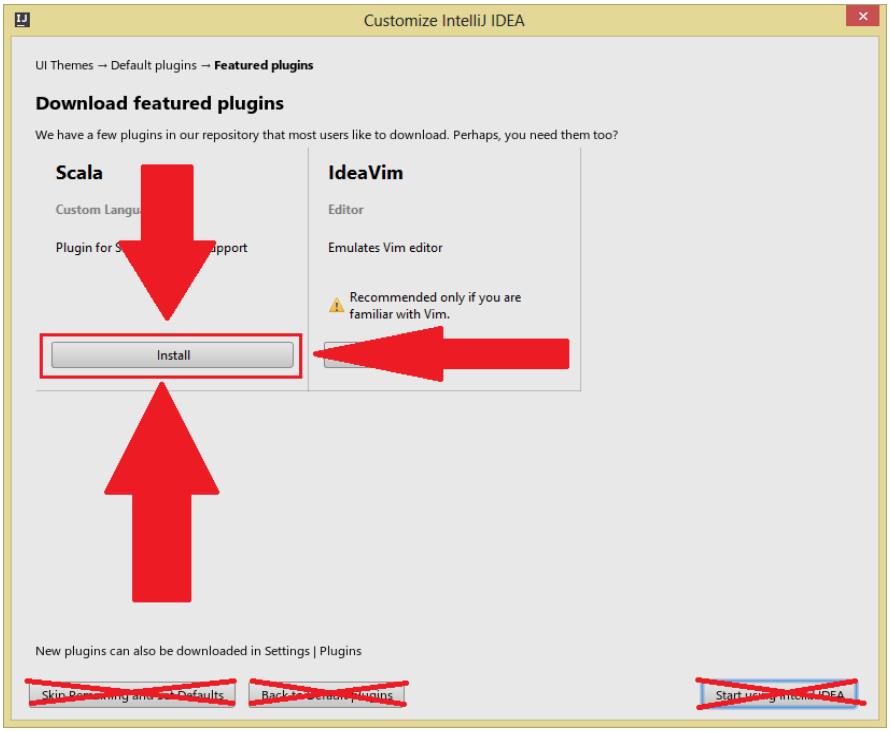

Рисунок 8 – Установка плагина для поддержки языка Scala

После завершение установки плагина нажимаем нижнюю правую кнопку (рисунок 9).

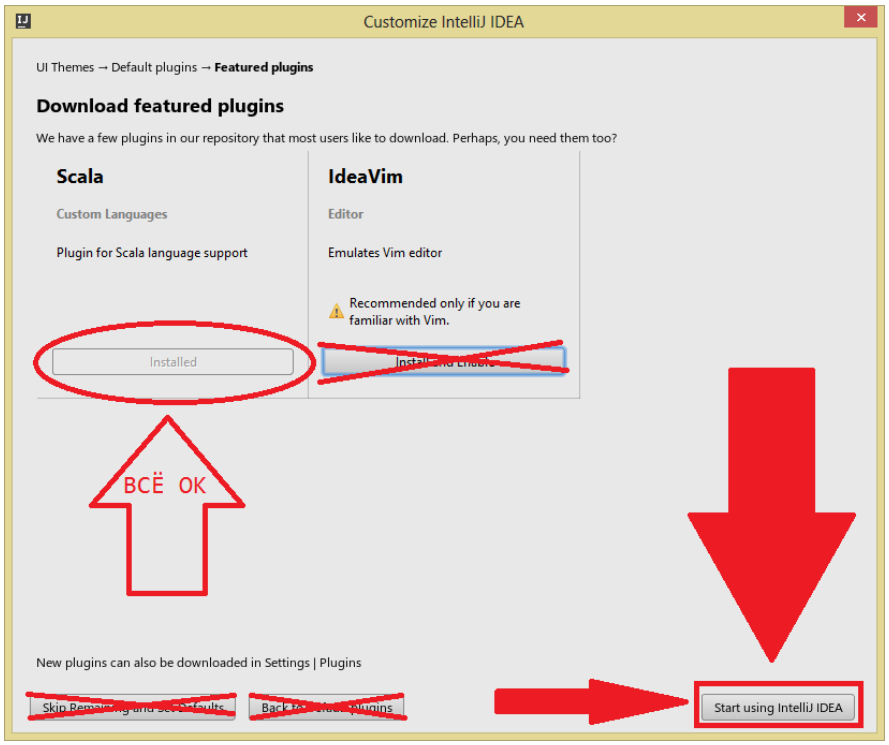

Рисунок 9 – Завершение первоначальной настройки

2.4. Создание Scala-проекта в IntelliJ IDEA

После запуска IntelliJ IDEA появится следующее окно (рисунок 10).

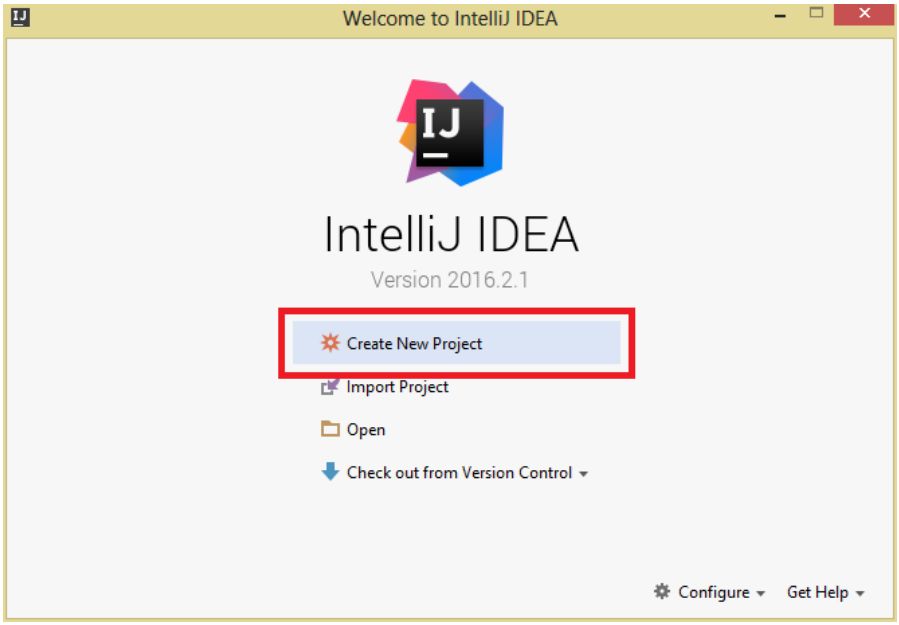

Рисунок 10 – Начальное окно IntelliJ IDEA

Выбираем создание нового проекта. В появившемся окне (рисунок 11) выбираем Scala.

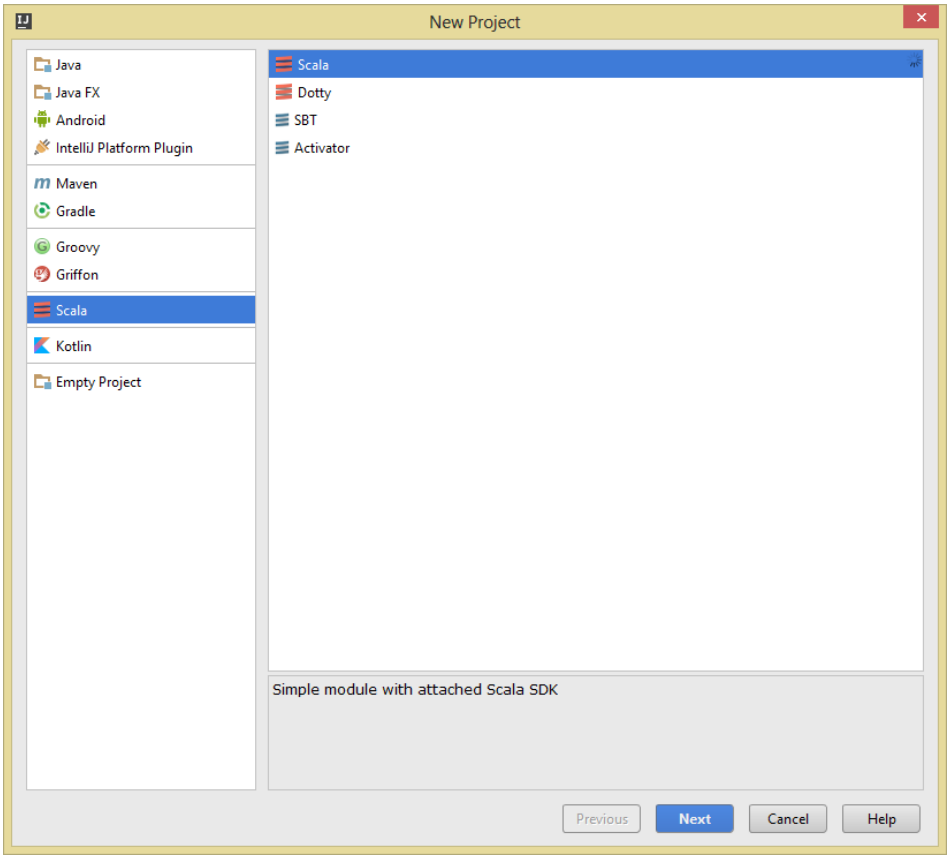

Рисунок 11 – Окно выбора типа нового проекта

Далее (рисунок 12) можно изменить имя проекта (1) и директорию проекта (2). Затем необходимо указать путь к JDK, для этого нажимаем кнопку «New» (3) и выбираем пункт «JDK» (4).

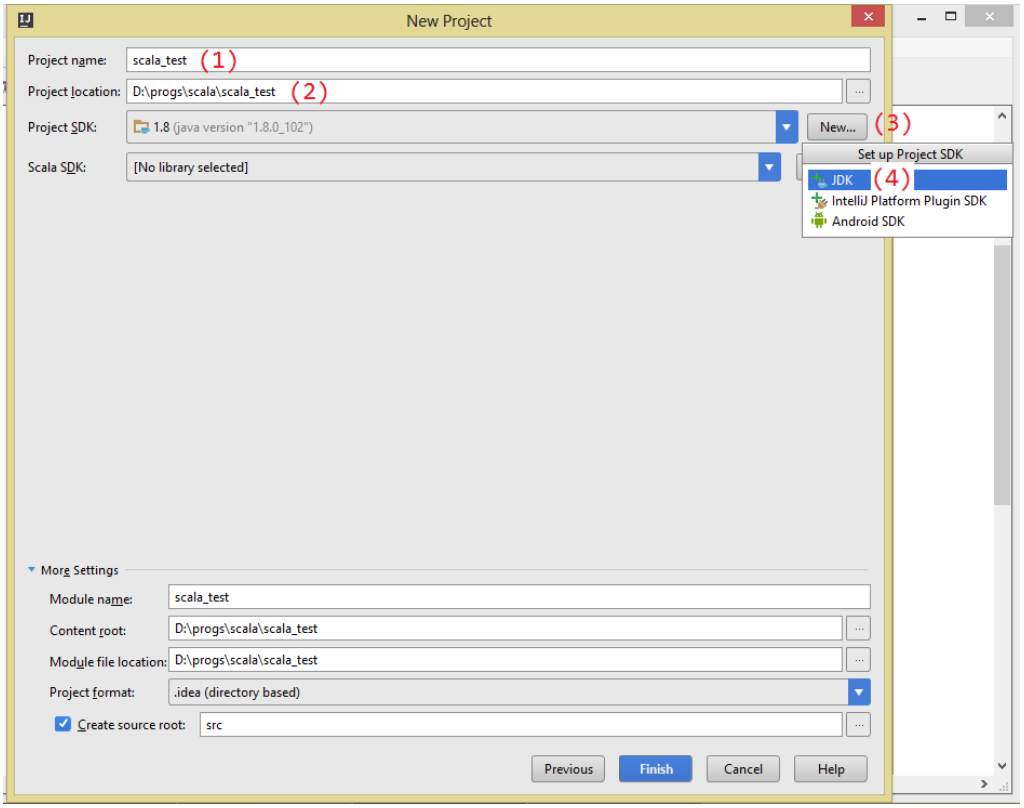

Рисунок 12 – Настройки проекта

В появившемся окне (рисунок 13) указываем путь к JDK, установленному в пункте 2.1 настоящих методических указаний. По-умолчанию это C:\Program Files\Java\jdk… или C:\Program Files (x86)\Java\jdk…

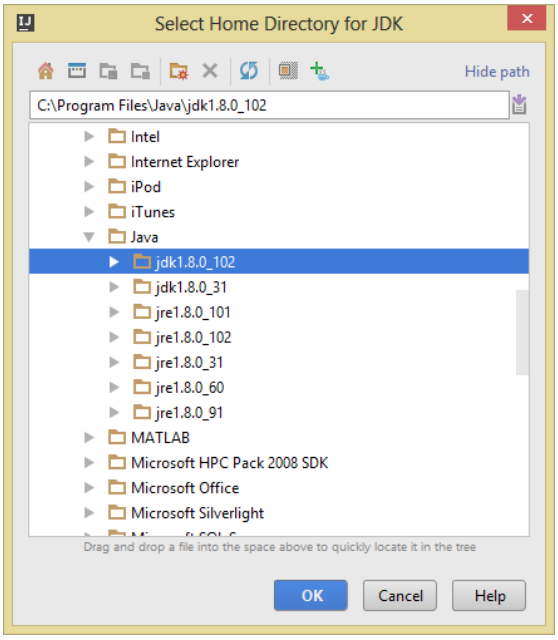

Рисунок 13 – Окно выбора пути к JDK

Затем надо указать Scala SDK. Для этого нажимаем кнопку «Create» напротив Scala SDK. В появившемся окне (рисунок 14) нажимаем кнопку «Download» и выбираем (рисунок 15) самую последнюю версию Scala (на момент написания настоящих методических указаний актуальной версией является 2.11.8).

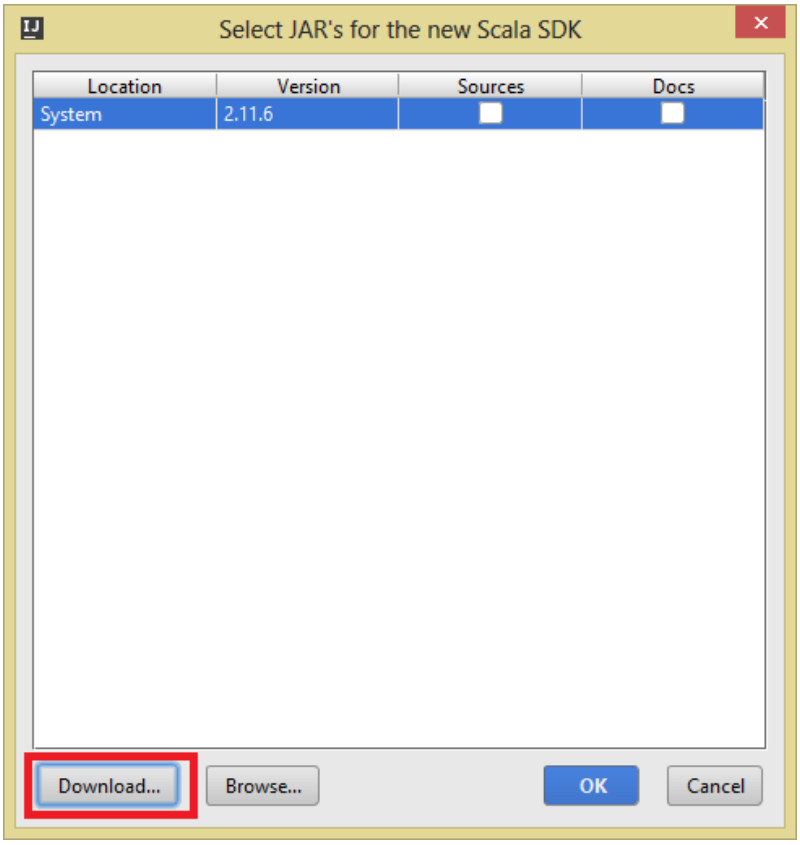

Рисунок 14 – Окно выбора Scala SDK

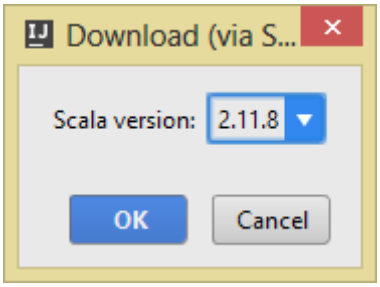

Рисунок 15 – Окно выбора версии Scala

После этого начнется процесс загрузки необходимых файлов (рисунок 16), который может занять времени больше, чем хотелось бы.

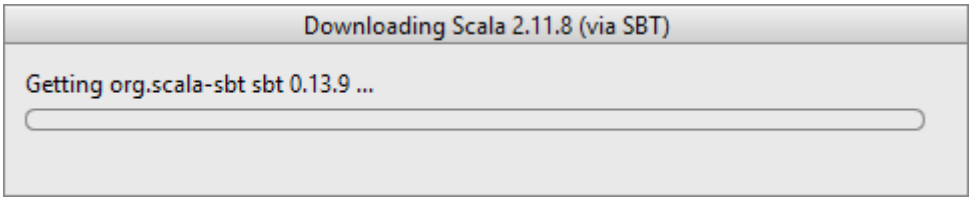

Рисунок 16 – Процесс установки необходимых файлов

После завершения установки нажимаем кнопку «Finish» (рисунок 17).

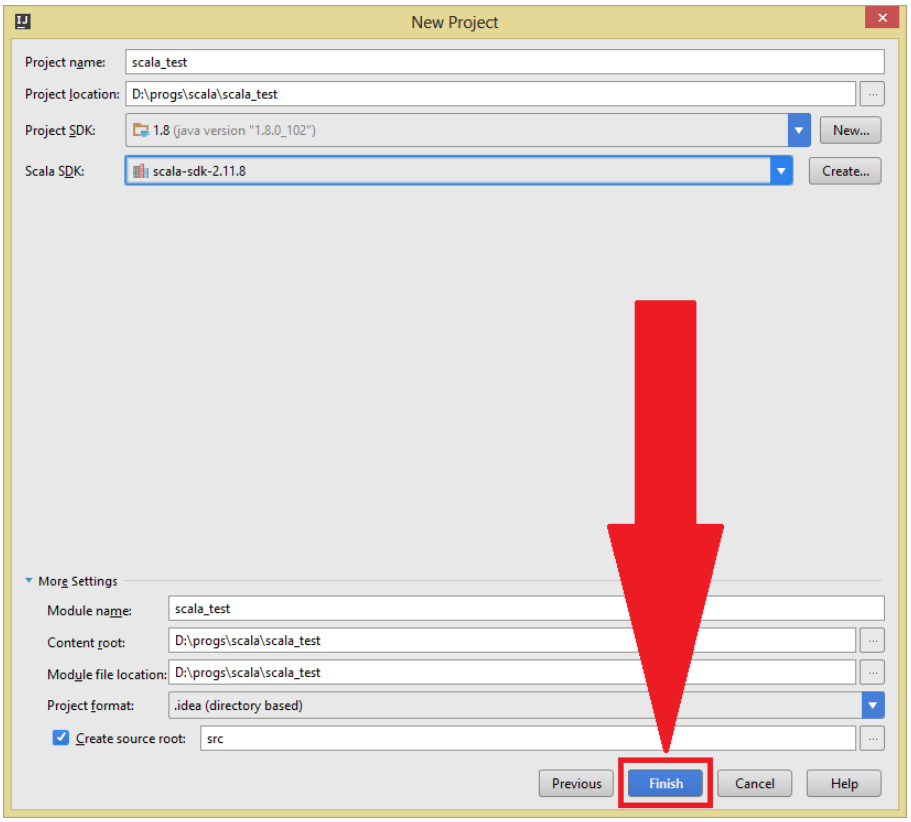

Рисунок 17 – Завершение настройки проекта

Откроется основное окно IntelliJ IDEA с созданным проектом (рисунок 18).

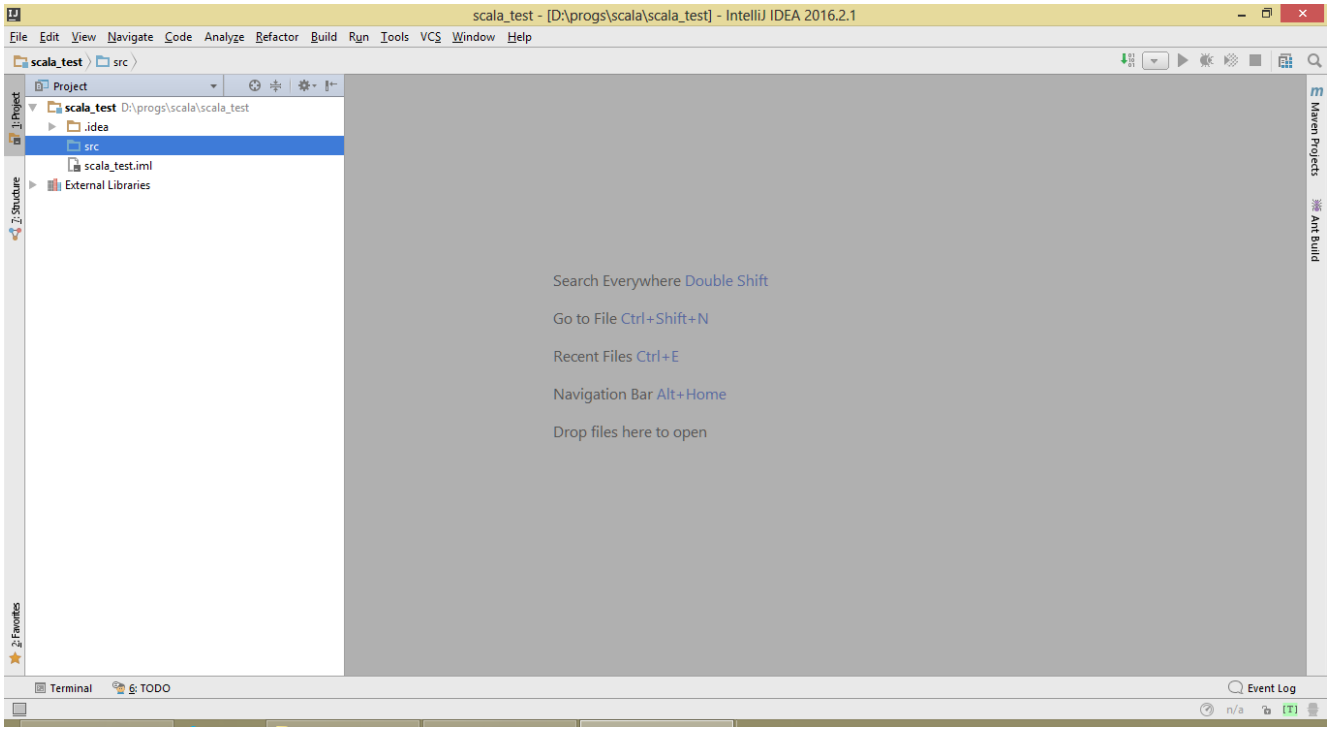

Рисунок 18 – Основное окно IntelliJ IDEA

В папке *src* создадим пакет *main*, внутри которого создадим пакет *scala* (рисунок 19).

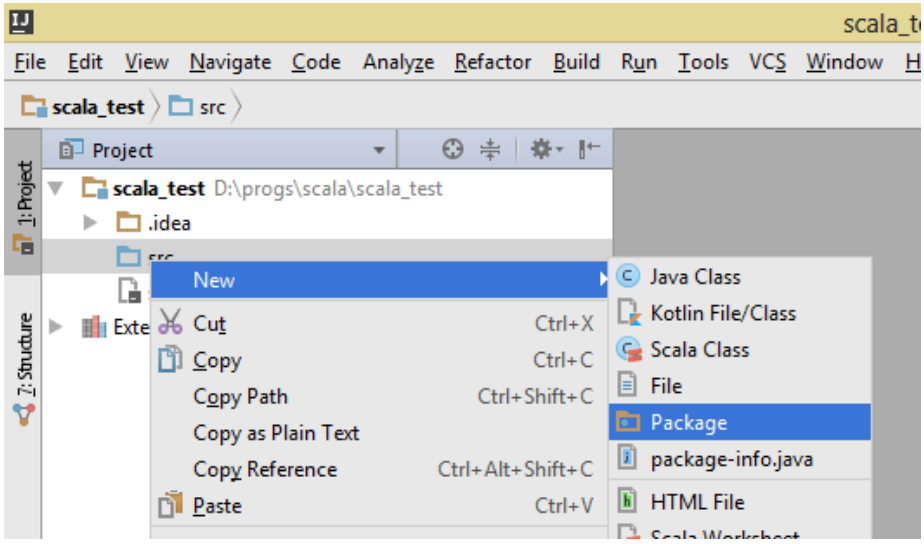

Рисунок 19 – Создание пакета

Затем в пакете *main.scala* создадим объект приложения (рисунки 20 и 21).

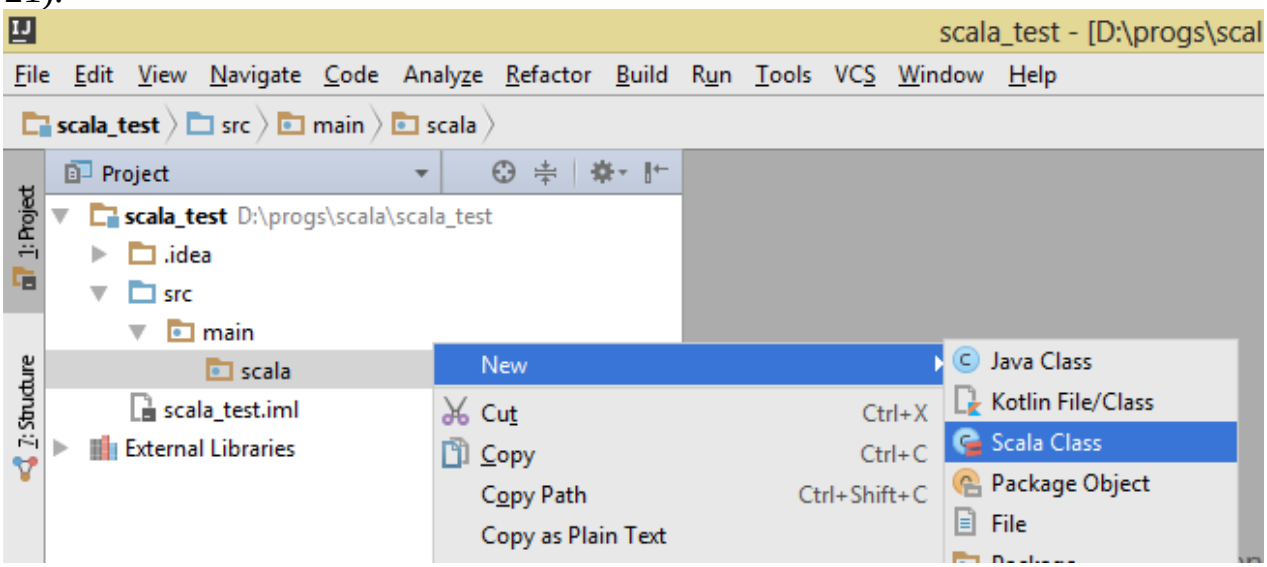

Рисунок 20 – Создание файла с исходным кодом Scala

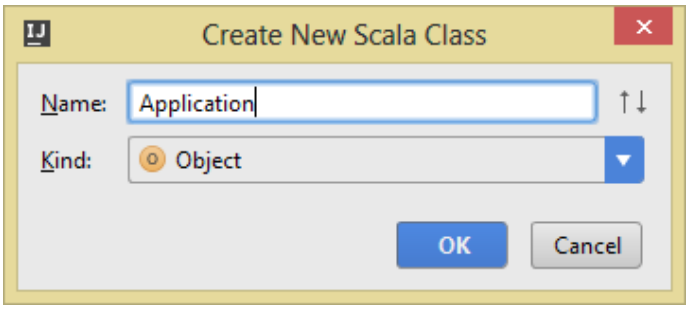

Рисунок 21 – Создание объекта приложения

В открывшемся файле введём код, представленный на рисунке 22.

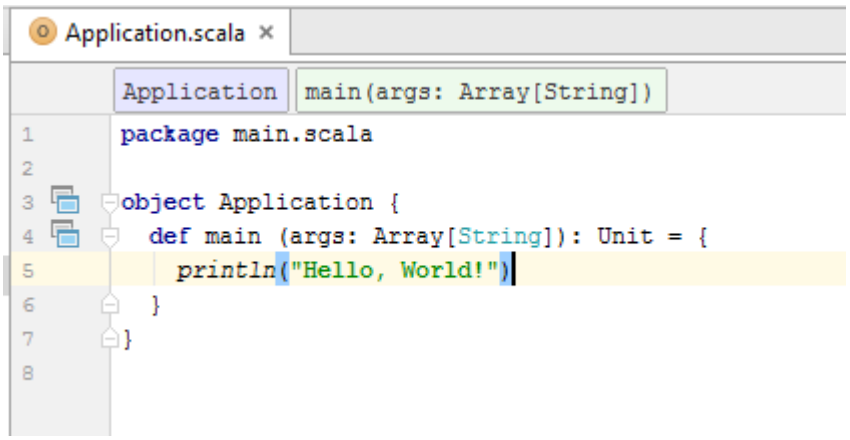

Рисунок 22 – Тестовый код для проверки работоспособности окружения

Сохраним файл и запустим его. Для этого нажмём правой кнопкой на вкладке с файлом и выберем «Run 'Application'» (рисунок 23).

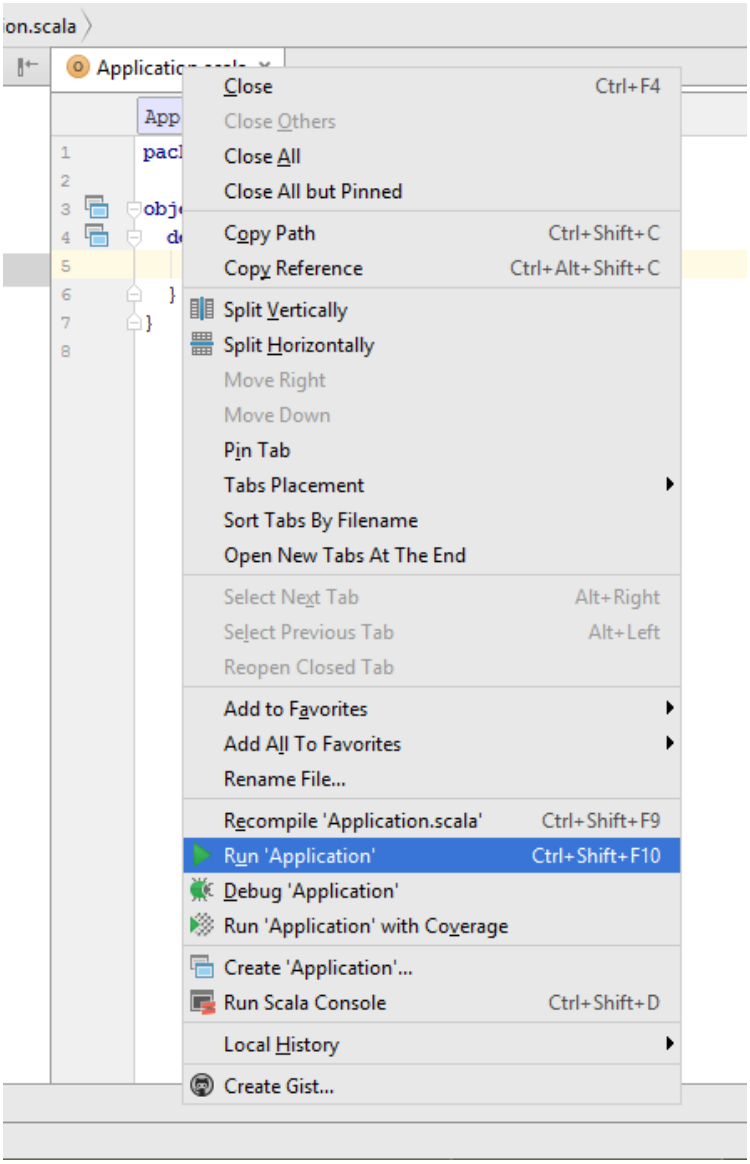

Рисунок 23 – Запуск программы

В результате в нижней части окна откроется консоль с результатами работы программы (рисунок 24).

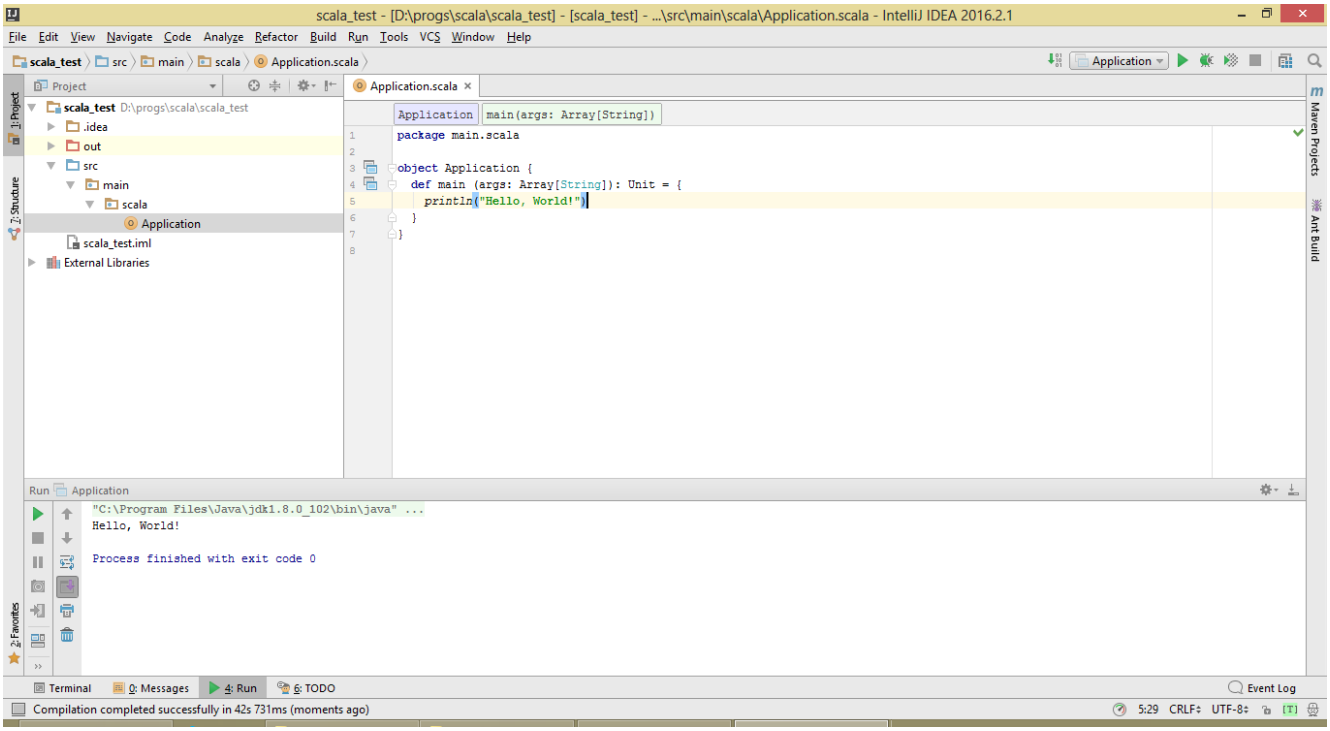

Рисунок 24 – Результаты работы программы

Таким образом, мы получили настроенную среду разработки и готовы к выполнению лабораторных работ.

ЛАБОРАТОРНАЯ РАБОТА № 1
ИССЛЕДОВАНИЕ СПОСОБОВ РЕАЛИЗАЦИИ РЕКУРСИВНЫХ АЛГОРИТМОВ В ФУНКЦИОНАЛЬНОМ ПРОГРАММИРОВАНИИ

1. ЦЕЛЬ РАБОТЫ

Исследовать различные способы реализации рекурсивных алгоритмов в функциональном программировании. Приобрести практические навыки реализации рекурсивных функций обработки списков на языке Scala.

2. ОСНОВНЫЕ ПОЛОЖЕНИЯ

2.1. Объявление переменных в языке Scala

В языке Scala существует два способа объявления переменных: с помощью ключевого слова val и ключевого слова var. Значения, объявленные с помощью val являются константами, они принимают своё значение один единственный раз – при объявлении. Значения, объявленные с помощью var являются обычными переменными, привычными по императивным языкам. Пример:

```
val i = 0
var j = 0
j = j + 1    // OK
i = i + 1    // Ошибка Reassignment to val
```

На примере выше видно, что:

1) Однострочный комментарий в языке Scala начинается с // (многострочный комментарий начинается с /* и заканчивается */),
2) В конце инструкции не обязательно ставить точку с запятой (;). Однако, если на одной строке идут подряд несколько инструкций, между ними необходимо ставить точку с запятой,
3) Объявлять тип переменной при её определении не обязательно (несмотря на то, что Scala – язык со статической типизацией). Компилятор в силах сам вывести тип переменной, стоящей слева от знака присваивания, основываясь на выражении, стоящем справа от знака присваивания.

Однако, если очень хочется, можно явно указать тип следующим образом:

```
val i: Double = 1    // аналог на Java – double i = 1;
```

2.2. Основные типы в языке Scala

Как и в языке Java, в Scala семь числовых типов – Byte, Char, Short, Int, Long, Float и Double. Также есть логический тип Boolean. В отличии от Java, все эти типы являются классами. Соответственно, числа являются объектами классов, что дает возможность вызывать их методы следующим образом:

```
1.toString()    // вернёт строку “1”
1.to(5)         // вернёт объект класса Range(1, 2, 3, 4, 5)
```

2.3. Упрощённый вызов методов в языке Scala

Язык Scala поддерживает упрощённый вызов методов, например:

```
a.method(b)
a method b
```

Описанные выше две строки идентичны. Такая возможность вызова методов позволяет в некоторых случаях писать более понятный код и реализовывать DSL (Domain-specific language).

Ещё одной особенностью языка Scala в плане вызовов методов является возможность не указывать скобки при вызове метода, не имеющего параметры, например:

```
1.toString    // идентично вызову 1.toString()
// или даже так:
1 toString
```

Среди программистов на языке Scala существует соглашение, функции/методы без параметров, которые не имеют побочных эффектов, вызываются без скобок, а функции/методы без параметров, которые имеют побочные эффекты, вызываются со скобками.

2.4. Метод Apply

В Scala используется синтаксис, напоминающий вызовы функций. Например, если *str* – это строка, то выражение *str(i)* вернёт *i*-ый символ строки.

```
“Hello”(4)    // вернёт ‘o’
```

Это можно считать перегруженной формой оператора (). Но на самом деле это метод с именем *apply*. И поэтому вызов *“Hello”(4)* является краткой формой записи *“Hello.apply(4)”*.

2.5. Условные выражения в языке Scala

В Scala имеется условное выражение if/else, как и в языке Java, но в Scala это не просто оператор, а именно выражение (со своим значением). Например, выражение

if (x > 0) 1 else -1

имеет значение 1 или -1, в зависимости от значения x. И всё это выражение имеет тип Int. По сути, это аналог тернарного оператора из Java/C++/C# (*x > 0 ? 1 : -1*).

Пример инициализации значения переменной в зависимости от условия на языке Java:

```
int y = 0;
if (x > 0) y = 1 else y = -1;
```

То же самое на языке Scala

```
val y = if (x > 0) 1 else -1    // или на Java: int y = x > 0 ? 1 : -1;
```

Если вместо одного выражения идёт блок выражений, то конечным значением всего блока будет значение последнего выражения. Например:

```
val i = {1; 2}    // i имеет значение 2

val test = if (x > 0)
{    // начало первого блока
  2
  println(2+2)
  “String”
}    // значение первого блока – “String”
else
{    // начало второго блока
  3
  2+2
}    // значение второго блока - 4
```

В данном примере ветви выражения if/else имеют разный тип. В таком случае типом переменной temp будет являться общий предок типов ветвей выражения. В данном случае тип ветви if – *String*, а тип ветви else – *Int*. Эти два типа не имеют другого общего предка, кроме класса *Any* (аналог класса *Object* в языке Java. Класс, от которого автоматически наследуются все остальные классы). Таким образом, переменная temp имеет тип *Any*.

2.6. Циклы в языке Scala

В Scala имеются циклы while и do-while, как в языках Java/C++/C#:

```
var n = -10
while (n < 0) {
  println(n)
```

```
  n = n + 1
}

var m = 0
do {
  println(m)
  m += 1    // в Scala нет оператора ++, но есть оператор +=
} while (m < 10)
```

В Scala нет прямого аналога оператора *for(<инициализация> ; <условие выхода> ; <изменение>)*, но есть своё собственное выражение for, которое будет рассмотрено в следующих лабораторных работах.

2.7. Функции и процедуры в языке Scala

Функции в языке Scala определяются следующим образом:

```
def <имя функции>(<список параметров>): <тип возвращаемого значения> = {
  <тело функции>
}
```

Например, функция, которая перемножает свои аргументы, на языке Scala определяется следующим образом:

```
def mult(a: Int, b: Int): Int = {
  a*b
}
```

Обратите внимание на отсутствие оператора return. Как уже упоминалось выше (пункт 2.5) значением блока является последнее выражение. В данном случае значением, которое возвращает функция, является значение последнего вычисленного в ней выражения. Так же, как и для переменных, для функций не обязательно задавать тип возвращаемого значения. Компилятор в силах сам вычислить тип (за исключением рекурсивных функций, в которых всегда необходимо явно указывать тип). Таким образом, нижеизложенные определения функций полностью идентичны и законны в рамках синтаксиса языка Scala:

```
def fun1(a: Int, b: Int): Int = { a+b }
def fun2(a: Int, b: Int) = { a+b }
def fun3(a: Int, b: Int) = a+b
```

Процедуры на языке Scala определяются так же, как и функции, с одной лишь особенностью: между сигнатурой функции и открывающей скобкой блока кода отсутствует знак ‘=’. Пример:

```
def funkciya() = { 2+3 } // это функция, которая всегда возвращает 5

def procedura() { 2+3 } // это процедура, которая всегда возвращает ничего
```

Для обозначения «ничего» в языке Scala существует класс Unit (аналог void в Java/C++/C#). Таким образом, все процедуры имеют тип Unit.

Если тело функции состоит всего из одного выражения, то указывать фигурные скобки не обязательно. Пример:

```
def sum(a: Int, b: Int): Int = a + b
```

2.8. Рекурсивные функции

Рекурсивная функция – это функция, которая в своём теле содержит вызов самой себя. Продемонстрируем рекурсивную функцию на примере вычисления факториала n-го числа:

```
def fact(n: Int): Int = {
  if (n == 0) 1
  else n * fact(n-1)
}
```

Первое правило рекурсии – прежде чем входить в рекурсию, подумай, как из неё выйти. В данном случае мы сразу определили граничное условие завершения рекурсии (n == 0). А далее следует тривиальное определение факториала ($fact_n = n*fact_{n-1}$).

Рассмотрим подробнее как происходит вычисление факториала. Для примера вычислим факториал от числа 3 (рисунок 1):

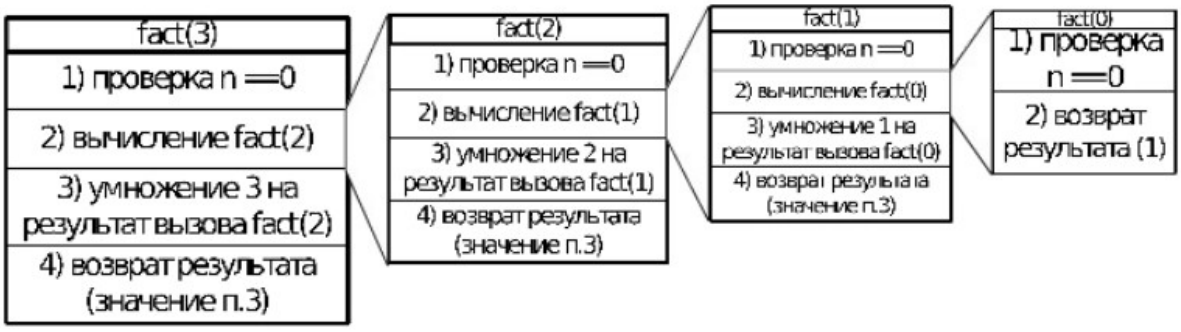

Рисунок 1 – Рекурсивные вызовы функции вычисления факториала

На рисунке 1 видно, что между рекурсивным вызовом (п. 2) и возвращением результата из функции (п. 4) есть еще действия, которые необходимо выполнить (п. 3). Таким образом, во время каждого нового рекурсивного вызова необходимо сохранять в стеке контекст предыдущего вызова, чтобы к нему можно было вернуться и продолжить вычисления. Это ведёт к накладным расходам и, в критическом случае, переполнению стека.

Выходом из данной ситуации является хвостовая рекурсия.

2.9. Хвостовая рекурсия

Хвостовая рекурсия – это особый вид рекурсии. Отличие заключается в том, что в теле функции рекурсивный вызов находится в самом конце, перед возвратом значения из функции.

Это позволяет на уровне компиляции оптимизировать рекурсивные вычисления и преобразовать их в циклы. При таком подходе значение, которое необходимо вернуть из функции не вычисляется напрямую, (как в п.3 рисунка 1). Значением, возвращаемым из функции, является значение следующего рекурсивного вызова. Таким образом, нет нужды сохранять контекст выполнения вызывающей функции при рекурсивном вычислении и можно использовать одну и ту же ячейку стека для последующих вызовов.

Чтобы преобразовать обычную рекурсию в хвостовую, необходимо перенести вычисление результата (п.3 на рисунке 1) из тела функции в её параметр (т.н. аккумулятор). Преобразуем нашу функцию факториала к рекурсивному виду.

```
def fact(n: Int, acc: Int): Int = {
  if (n == 0) acc
  else fact(n-1, acc * n)
}
// Начальное значение аккумулятора = 1
fact(7, 1) // Запуск рекурсии
```

Как видно, при преобразовании функции к рекурсивному виду изменился ее интерфейс (был один входной параметр, а стало два). В случаях, когда надо сохранить интерфейс хвостовую рекурсию оборачивают в функцию с желаемым интерфейсом:

```
def fact(n: Int): Int = {
  // внутри функции определим вспомогательную функцию,
  // которая реализует хвостовую рекурсию
  // (да, в Scala можно объявлять функции внутри функций)
  def factAcc(_n: Int, acc: Int): Int = {
    if (_n == 0) acc
    else factAcc(_n-1, acc * _n)
  }

  // и в качестве результата внешней функции вернём результат
  // работы функции с хвостовой рекурсией
  factAcc(n, 1)
}
```

Таким образом, мы добились двух вещей:

1) Преобразовали обычную рекурсию в хвостовую.
2) Сохранили интерфейс функции.

2.10. Списки в языке Scala

Список (List) – это коллекция данных, которая имеет определённую структуру. Список можно представить в виде цепочки, где каждое звено состоит из двух ячеек. В первой ячейке n-го звена содержится значение n-го элемента списка, а во второй ячейке звена содержится указатель на продолжение списка (на звено n+1). Структура списка «12, 3, 7, 4, 21, 9» представлена на рисунке 2.

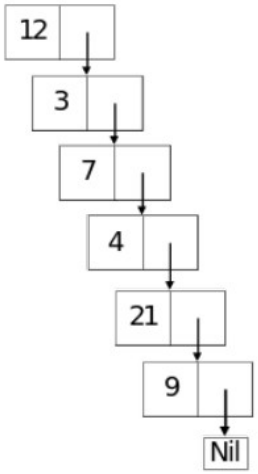

Рисунок 2 – Структура списка

Последним элементом списка обязательно должен является элемент, обозначающий конец списка (Nil).

В Scala списки можно создать следующим образом:

```
val spisok = 12 :: 3 :: 7 :: 4 :: 21 :: 9 :: Nil // spisok имеет тип List[Int]
// или
val spisok_dva = List(12, 3, 7, 4, 21, 9) // в данном случае добавлять
// Nil в конец списка не надо (но он там есть)
```

При объявлении переменной spisok_dva мы не указали тип содержимого списка, но компилятор в силах сам вывести тип, таким образом типом значения spisok_dva является List[Int].

Для работы со списком есть два основных свойства – head (для получения первого значения в списке) и tail (для получения продолжения списка).

```
val nums = 3 :: 7 :: 4
nums.head // вернёт 3
nums.tail // вернёт List(7, 4)
nums.tail.tail // вернёт List(4) – не значение, а список из одного значения
nums.tail.tail.tail // вернёт Nil
```

Стандартные списки в Scala неизменяемые (как и большинство значений в функциональном программировании). Это означает, что для

изменения значения списка необходимо создать новый список с обновленным значением.

Этот факт налагает различную стоимость на операции со списком. Операции с головой списка производятся быстро. Можно сразу получить значение головы списка, также можно сразу удалить голову списка (вызвать tail) или заменить голову (присоединить новое значение к хвосту списка).

```
val lst = List(4, 9 ,6)
nums.tail // удаление головы (вернёт список без его головы)
2 :: nums.tail // изменение головы списка
```

Добавление в начало списка также проводится быстро:

```
val another_list = 375 :: 88 :: 42
// добавление числа 99 к голове списка
99 :: another_list // вернёт новый список List(99, 375, 88, 42)
```

Чтобы получить значение n-го элемента списка, необходимо перебрать n-1 элемент списка, которые предшествуют искомому значению, следовательно, чем дальше элемент, который мы хотим получить, тем дольше будет выполняться операция получения значения.

С добавлением элемент в конец списка тоже не всё так просто. При добавлении элемента в начало списка создается новое звено из двух ячеек. В первую ячейку помещается значение, которое мы хотим добавить к списку, а во вторую – указатель на имеющийся список. По аналогии, чтобы добавить значение к концу списка, необходимо создать новое звено, в первую ячейку помещается добавляемое значение, а во вторую – Nil. Чтобы присоединить это звено к списку необходимо изменить указатель последнего звена имеющегося списка, чтобы вместо Nil он указывал на только что созданное новое звено. Но, т.к. объекты и значение неизменяемые, то для того, чтобы изменить последнее звено списка, нужно создать его заново (со старым значением и новым указателем на продолжение списка). Это влечёт за собой пересоздание предпоследнего звена списка, затем предпредпоследнего и т.д. Таким образом, при добавлении элемента в конец списка, происходит перестройка всего списка.

2.11. Обработка списков

Для обработки списков удобно применять рекурсивные функции шаблона «пока не Nil – обрабатываем голову – делаем рекурсивный вызов от хвоста списка». Например, продемонстрируем функцию, которая печатает каждый элемент списка:

```
def printList(lst: List[Int]): Unit = {
  if (lst == Nil) println(“Конец списка”)
  else {
    println(lst.head)
    printList(lst.tail)
```

```
  }
}
```

Ниже изображены еще два примера рекурсивных функций для обработки списков:

```
// Функция возвращает сумму ввсех его элементов
def sumList(lst: List[Int]): Int = {
  if (lst == Nil) 0
  else lst.head + sumList(lst.tail)
}

// Функция возвращает новый список, где каждый элемента
// в два раза больше соответствующего элемента исходного списка
def doubleList(lst: List[Int]): List[Int] = {
  if (lst == Nil) Nil
  else lst.head * 2 :: doubleList(lst.tail)
}
```

Перепишем эти функции используя хвостовую рекурсию:

```
// Функция возвращает сумму ввсех его элементов
def sumList(lst: List[Int]): Int = {
  def sumListAcc(__list: List[Int], acc: Int): Int = {
    if (__list == Nil) acc
    else sumListAcc(__list.tail, acc + __list.head)
  }

  sumListAcc(lst, 0)
}

// Функция возвращает новый список, где каждый элемента
// в два раза больше соответствующего элемента исходного списка
def doubleList(lst: List[Int]): List[Int] = {
  def doubleListAcc(__list: List[Int], acc: List[Int]): List[Int] ={
    if (__list == Nil) acc
    else doubleListAcc(__list.tail, __list.head * 2 :: acc)
  }

  doubleListAcc(lst, Nil).reverse
}
```

3. ЗАДАНИЕ НА ЛАБОРАТОРНУЮ РАБОТУ

Написать и исследовать функции, которые преобразуют список, в соответствии с вариантом задания:

1. Для своего варианта задания написать два вида функции – обычную рекурсию и хвостовую рекурсию.
2. Убедиться, что на достаточно большом количестве элементов в списке обычная рекурсия выдаёт ошибку переполнения стека (stack overflow), а хвостовая рекурсия продолжает работать.
3. Зафиксировать длину списка (количество элементов), для которого обычная рекурсия перестаёт работать.

3.1. Варианты заданий

1. Функция возвращает новый список, в котором каждый элемент является суммой предыдущего элемента нового списка и текущего элемента входного списка. Пример: List(1, 4, 2) преобразуется в List(1, 5, 7);
2. Функция возвращает новый список, в котором каждый элемент входного списка повторяется дважды. Пример: List(33, 9, 321) преобразуется в List(33, 33, 9, 9, 321, 321);
3. Функция возвращает новый список, в котором каждый элемент является список из чисел от 0 до модуля соответствующего элемента входного списка. Пример: List(2, -6, 10) преобразуется в List(List(0, 1, 2), List(0, 1, 2, 3, 4, 5, 6), List(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10)).

4. КОНТРОЛЬНЫЕ ВОПРОСЫ

4.1. Какие существуют виды переменных в языке Scala?
4.2. Поясните отличие переменных var от значений val.
4.3. Приведите пример упрощенного методов в Scala.
4.4. Приведите пример использования метода apply() в Scala.
4.5. Поясните, чем отличается условная конструкция if-else в Scala и в Java?
4.6. В чем особенность использования оператора return в функциях Scala?
4.7. Дайте определение рекурсивной функции.
4.8. В чем причина возможного переполнения стека при использовании рекурсивных функций?
4.9. Дайте определение хвостовой рекурсии.
4.10. Опишите структуру списка в Scala и назовите основные методы класса List.
4.11. Объясните алгоритм добавления элемента в начало и в конец списка в Scala.

ЛАБОРАТОРНАЯ РАБОТА №2
ИССЛЕДОВАНИЕ ФУНКЦИЙ ВЫСШИХ ПОРЯДКОВ И МЕТОДОВ ОБРАБОТКИ КОЛЕКЦИЙВ ФУНКЦИОНАЛЬНОМ ПРОГРАММИРОВАНИИ

1. ЦЕЛЬ РАБОТЫ

Исследовать способы программной реализации и обработки кортежей, массивов и коллекций. Приобрести практические навыки разработки и применения функций 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 изменения 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 С-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. Метод 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. Метод 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. Методы 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. Методы 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. Методы 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. Методы 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. Методы 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. Методы 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 2.7 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 и исследовать 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 написать пример использования метода для своего факультета (в зависимости от варианта задания).

4. КОНТРОЛЬНЫЕ ВОПРОСЫ

4.1. Дайте определение кортежа.

4.2. Приведите пример создания кортежа в Scala.

4.3. Каким образом можно обратиться получить значение элементов кортежа?

4.4. Дайте определение ассоциативного массива.

4.5. Приведите пример создания ассоциативного массива в Scala.

4.6. Поясните особенности изменения элементов колекций в функциональном программировании.

4.7. Приведите пример использования изменяемых (mutable) и неизменяемых (immutable) коллекций в Scala.

4.8. В чем сходство ассоциативного массива и частично определенной функции?

4.9. Приведите пример использования for-генератора в Scala.

4.10. Дайте определение лямбда-функции.

4.11. Приведите пример описания лямбда-функции.

4.12. Дайте определение функции высшего порядка.

4.13. Перечислите встроенные функции высшего порядка для обработки коллекций в Scala и приведите пример их использования.

ЛАБОРАТОРНАЯ РАБОТА №3
ИССЛЕДОВАНИЕ СПОСОБОВ РЕАЛИЗАЦИИ КЛАССОВ И ОБЪЕКТОВ В ЯЗЫКЕ SCALA

1. ЦЕЛЬ РАБОТЫ

Исследовать особенности реализации классов, объектов и трейтов в языке Scala. Приобрести практические навыки реализации программных модулей для обработки сложных структур данных, используя объектно-ориентированный и функциональный подходы.

2. ОСНОВНЫЕ ПОЛОЖЕНИЯ

2.1. Классы в языке Scala

2.1.1. Базовое определение и использование классов

В общем виде классы на языке Scala выглядят очень похоже на классы на языке Java:

```
class Counter {
  private var value: Int = 0    // поля должны инициализироваться

  def increment() {
    value += 1
  }

  def current() = value
}
```

Создание и использование класса полностью аналогично привычным языкам программирования:

```
val myCounter = new Counter    // или new Counter()
myCounter.increment()
println(myCounter.current)    // или println(myCounter.current())
```

В языке Scala существует соглашение, согласно которому методы-мутаторы (методы, которые изменяют что-либо, имеют побочный эффект) вызываются с пустыми скобками (в примере это метод increment()), а методы-акцессоры (методы, которые возвращают значение и не имеют побочных эффектов) вызываются без скобок (в примере это метод current).

Можно насильно заставить использовать синтаксис вызова метода без скобок, если опустить скобки в объявлении метода:

```
...
def current = value    // объявляем без скобок
...
println(myCounter.current())    // вызов со скобками, ошибка компиляции
```

2.1.2. Свойства и методы доступа к ним

В языке Java не принято использовать публичные поля. Взамен этого создаётся приватное поле value и пара публичных методов getValue, setValue:

```
// это Java:
public class Person
{
   private int age;

   public int getAge()
   {
      return age;
   }

   public void setAge(int newAge)
   {
      this.age = newAge;
   }
}
```

Такой подход позволяет при необходимости добавлять логику в работу со свойством. Например, ограничить изменение возраста в меньшую сторону (нельзя омолодить человека):

```
// это Java:
public class Person
{
   private int age;

   public int getAge()
   {
      return age;
   }

   public void setAge(int newAge)
   {
      if (newAge > this.age) {  // изменяем возраст только на бОльший
         this.age = newAge;
      }
   }
}
```

В Scala методы чтения и записи создаются автоматически для каждого поля класса. Например:

```
class Person {
  var age = 0
}
```

Данный код скомпилируется в класс, содержащий приватное поле age и методы чтения и записи. Эти методы будут публичными (т.к. поле age не было объявлено приватным, для приватного поля будут созданы приватные методы чтения и записи).

В Scala метод чтения получит имя age, а метод записи получит имя age_=. Пример:

```
val john = new Person()
println(john.age)    // вызовет метод john.age()
john.age = 32        // вызовет метод john.age_=(32)
```

Методы чтения и записи можно свободно переопределять на своё усмотрение:

```
class Person {
  private var privateAge = 0

  def age = this.privateAge

  def age_=(newAge: Int): Unit = {
    if (newAge > privateAge) {
      this.privateAge = newAge
    }
  }
}
```

Итого в Scala на выбор существует четыре варианта реализации свойства класса:

1) var attribute – Scala создаёт методы для чтения и для записи.
2) val attribute – Scala создаёт только метод чтения.
3) Самому определить методы attribute и attribute_=.
4) Самому определить метод attribute.

2.1.3. Конструктор и дополнительные конструкторы

В Scala класс может иметь произвольное количество конструкторов. Однако есть один конструктор, который важнее других – главный конструктор. Кроме этого, класс может иметь любое количество дополнительных конструкторов.

Для начала рассмотрим дополнительные конструкторы. Они имеют два отличия от конструкторов в других языках:

1) Дополнительные конструкторы называются this (в Java или C++ конструкторы имеют имя класса)

2) Каждый дополнительный конструктор должен начинаться вызовом одного из дополнительных конструкторов, объявленных выше.

Следующий класс имеет два дополнительных конструктора:

```
class Person {
  private var name: String = “init”
  private var age: Int = 0

  def this(name: String) {    // дополнительный конструктор
    this()    // вызов главного конструктора
    this.name = name
  }

  def this(name: String, age: Int) {// другой дополнительный конструктор
    this(name)  // вызов предыдущего дополнительного конструктора
  }
}
```

О главном конструкторе будет рассказано чуть позже. Пока что достаточно знать, что класс, не определяющий главный конструктор явно, получает главный конструктор без аргументов.

Создать экземпляр вышеописанного объекта можно тремя способами:

```
/*
 * главный конструктор
 * name = “init”
 * age = 0
 */
val p1 = new Person

/*
 * первый дополнительный конструктор
 * name = “Hamilton”
 * age = 0
 */
val p2 = new Person(“Hamilton”)

/*
 * второй дополнительный конструктор
 * name = “Jefferson”
 * age = 42
 */
val p3 = new Person(“Jefferson”, 42)
```

Главный конструктор на определяется, как метод this, а является частью определения класса.

Параметры главного конструктора перечисляются сразу после имени класса:

```
class Person(name: String, age: Int) {
  // ...
}
```

Параметры главного конструктора автоматически превращаются в поля класса, которые инициализируются аргументами конструктора. Вышенаписанное определение класса Person равносильна следующему коду на Java:

```
// это Java:
public class Person
{
   private String name;
   private int age;

   public Person(String name, int age)
   {
      this.name = name;
      this.age = age;
   }

   public String name()
   {
      return this.name;
   }

   public int age()
   {
      return this.age;
   }
}
```

Главный конструктор выполняет все инструкции в определении класса. Например:

```
class Person(val name: String, val age: Int) {
  println(“Создан экземпляр класса Person”)
  def description = name + “ is ” + age + “ years old”
}
```

Инструкция println является частью главного конструктора и будет вызвана при каждом создании экземпляра класса Person. Это удобно, если нужно настроить поле объекта в процессе создания:

```
class MyClass {
  private val someResource = new Resource
  someResource.load(...)
  // ...
}
```

Если после имени класса отсутствуют параметры, то класс получит главный конструктор без параметров, который при создании просто выполнит все инструкции внутри класса.

Так же можно задавать значение параметров класса по-умолчанию (и тем самым избавиться от необходимости задавать дополнительные конструкторы):

class Person(name: String = “John Doe”, age: Int = 42)

2.2. Объекты в языке Scala

2.2.1. Объекты-одиночки

В Scala отсутствуют статические методы и поля. Вместо этого используются встроенные в язык singleton объекты. Например:

```
object Account {
  private var count: Int = 0

  def getUniqueID(): Int = {
    this.count += 1
    this.count
  }
}

// пример использования
val nextID = Account.getUniqueID()
```

Данный код аналогичен следующему коду на Java:

```
// это Java:
public class Account
{
    private static int count = 0;

    public static int getUniqueID()
    {
        this.count += 1;
        return this.count;
    }
}
```

Конструктор объекта вызывается при первом обращении к нему в коде. Если к объекту нет обращений в коде – конструктор не вызывается и объект не используется.

Объект обладает всеми свойствами класса, он может наследователь другие классы и трейты (аналоги интерфейсов в Scala). Единственное ограничение – объект не может иметь конструктор с параметрами.

2.2.2. Объекты-компаньоны. Общая характеристика

В Java или C++ часто имеется класс, который определяет не только поля методы экземпляра класса, но и статические поля и методы. В Scala добиться такого же эффекта можно с помощью объекта компаньона – объекта, который имеет такое же имя, как и класс.

```
class Account {
  val id = Account.newUniqueID()
  private var balance = 0.0

  def deposit(amount: Double) {
    balance += amount
  }
  ...
}

object Account {
  private var count: Int = 0

  def getUniqueID(): Int = {
    this.count += 1
    this.count
  }
}
```

По сравнению с обычными объектами, объекты-компаньоны имеют два отличия:

1) Их имя совпадает с именем класса;

2) Объект-компаньон имеет доступ к приватными полям и методам класса и наоборот: класс имеет доступ к приватным полям и методам объекта компаньона.

2.2.3. Объекты и метод apply

Зачастую в объектах определяется метод apply. Он используется для упрощения синтаксиса создания нового экземпляра класса. Например:

```
// обычное создание экземпляра класса
class MyClass {
  var count: Int = 0

  def this(initCount: Int) = {
    this()
    this.count = initCount
  }
}
```

```
val foo = new MyClass(13)  // создаст экземпляр класса, где count = 13

// создание экземпляра класса с помощью метода apply объекта
object MyClass {
  def apply(initCount: Int): MyClass = {
    new MyClass(initCount)
  }
}

val bar = MyClass(42)      // отличие в отсутствии new перед именем класса
```

Что мы сделали во фрагменте кода выше? По сути мы сделали обёртку для обычного создания экземпляра класса (с помощью ключевого слова new). Но из-за особенности работы языка Scala с методом apply (см. лабораторную работу №1) нам удалось избавиться от необходимости использовать ключевое слово new каждый раз при создании нового экземпляра объекта. Это может облегчить читаемость кода в сложных выражениях. Например, сравните два варианта создания вложенных массивов:

```
// Классический вариант с использованием ключевого слова new:
val matrix = new Array(new Array(1, 2, 3), new Array(4, 5, 6))

// Вариант с использованием метода apply объекта-компаньона
val matrix2 = Array(Array(1, 2, 3), Array(4, 5, 6))
```

Стоит отметить, что для использования данной возможности метода apply объекта, имя объекта не обязательно должно совпадать с именем класса. Пример:

```
object DifferentName {
  def apply(initCount: Int): MyClass = {
    new MyClass(initCount)
  }
}

val foobar = DifferentName(7)    // вернёт экземпляр класса MyClass
```

2.3. Трейты в языке Scala

Трейт в языке Scala можно использовать точно так же, как и интерфейсы в языке Java:

```
trait Logger {
  def log(msg: String)    // абстрактный метод
}
```

В данном случае не требуется явно указывать, что метод является абстрактным. Нереализованные в трейтах методы автоматически являются абстрактными.

Реализация интерфейса:

```
class ConsoleLogger extends Logger {  // extends, не implements
  def log(msg: String) {
    println(msg)
  }
}
```

При переопределении абстрактных методов трейта не требуется указывать ключевое слово override.

Если требуется унаследовать более одного трейта, дополнительные трейты добавляются через ключевое слово with:

```
class MyClass extends TraitOne with TraitTwo with TraitThree
```

Как и в Java, в Scala класс может быть унаследован от одного класса, но от многих трейтов.

Помимо описанного выше, трейты обладают более обширными возможностями, по сравнению с обычными интерфейсами (например, трейты могут имеют реализацию методов по-умолчанию). Эти особенности, при желании, можно изучить самостоятельно.

3. ЗАДАНИЕ НА ЛАБОРАТОРНУЮ РАБОТУ

3.1. Реализовать структуру бинарного дерева поиска (элементы в левом поддереве меньше, чем элемент в корне, а элементы правого поддерева больше, чем элемент в корне). Для реализации:

3.1.1. Написать трейт Tree, определяющий абстрактные методы getLeftSubtree: Tree, getRightSubtree: Tree и getNodeData: Int;

3.1.2. Написать класс Node, который реализует трейт Tree и представляет собой узел дерева;

3.1.3. Написать класс Leaf, который реализует трейт Tree и представляет собой лист дерева.

3.2. Для реализованной структуры дерева написать следующие функции:

3.2.1. Функцию printTree(Tree): Unit, которая выводит на экран дерево;

3.2.2. Функцию insert(Int, Tree): Tree, которая принимает элемент для вставки и корень дерева, возвращает корень нового дерева со вставленным элементом (при этом изначальное дерево изменяться не должно);

3.2.3. Функцию contains(Int, Tree): Boolean, которая возвращает true или false в зависимости от того, содержится ли заданное число в дереве, или нет;

3.2.4. Функцию sum(Tree): Int, которая возвращает сумму всех элементов в дереве.

3.2.5. Исследовать свойства реализованной структуры бинарного дерева, сделать выводы.

4. КОНТРОЛЬНЫЕ ВОПРОСЫ

4.1. Приведите пример объявления класса в Scala.

4.2. В чем отличие вызова методов-мутаторов и методов-акцессоров в Scala?

4.3. Какие методы создаются автоматически для каждого поля класса в Scala?

4.4. Перечислите четыре варианта реализации свойства класса в Scala.

4.5. Назовите особенности описания дополнительных конструкторов класса в Scala.

4.6. Что произойдет, если не описать главный конструктор класса?

4.7. Приведите пример описания главного конструктора класса в Scala.

4.8. Что поднимается под объектами-одиночками в Scala?

4.9. Что такое объект-компаньон, в чем их отличие от обычных объектов?

4.10. Для каких задач используется метод apply объектов-компаньонов?

4.11. Что такое трейты в Scala?

4.12. В чем отличие наследования трейтов от наследования классов?

ЛАБОРАТОРНАЯ РАБОТА №4
ИССЛЕДОВАНИЕ СПОСОБОВ РЕАЛИЗАЦИИ АЛГОРИТМОВ СОПОСТАВЛЕНИЯ С ОБРАЗЦОМ В ЯЗЫКЕ SCALA

1. ЦЕЛЬ РАБОТЫ

Исследовать особенности реализации алгоритмов сопоставления с образцом в языке Scala. Приобрести практические навыки использования case-классов и класса Option в функциональном программировании.

2. ОСНОВНЫЕ ПОЛОЖЕНИЯ

2.1. Оператор match в языке Scala

В языке Scala имеется аналог оператора switch из C-подобных языков, но он использует сопоставление с образцом и, благодаря этому, обладает более широкими возможностями.

Для начала рассмотрим применение match в стиле C-подобных языков:

```
val ch: Char =  // ...
var sign: Int = // ...

ch match {
  case ‘+’ => sign = 1
  case ‘-‘ => sign = -1
  case _   => sign = 0
}
```

Вместо default используется образец «_», с которым сопоставляется всё что угодно. Если его не включить в match, то, при отсутствии должного образца, будет возбуждено исключение MatchError.

Сопоставление с образцами идёт сверху вниз (как и в switch), однако в match отсутствует оператор break.

Так же, как и if, match это не оператор, а выражение, следовательно, он имеет своё значение, которое можно присваивать. Рассмотрим концептуально правильное написание предыдущего примера:

```
val ch: Char =  // ...

val sign = ch match {
  case ‘+’ => 1
  case ‘-‘ => -1
  case _   => 0
}
```

Здесь выражение match возвращает одно из значений, в зависимости от значения ch, и происходит присваивание возвращенного значение в sign.

2.2. Ограничители в языке Scala

К образцам можно добавлять ограничители – логические условия, при которых происходит сопоставление. Допустим, в предыдущем примере нам, помимо символов «+» и «-» необходимо еще распознавать все цифровые знаки. В C-подобном switch нам бы пришлось перечислять все возможные случаи (case ‘1’: case ‘2’: case ‘3’ и т.д.). В Scala же можно решить эту задачу более элегантным способом – добавив ограничитель на общий образец:

```
val ch: Char =  // ...

val value = ch match {
  case ‘+’ => 1
  case ‘-‘ => -1
  case _ if Character.isDigit(ch) => Character.digit(ch, 10)
  case _   => 0
}
```

Обратите внимание на отсутствие скобок вокруг условия ограничителя (в отличии от обычного if). Третий образец стоит интерпретировать следующим образом: «**ch это всё что угодно** *(за это отвечает символ _)* **но только если ch является цифрой** *(за это отвечает условие Character.isDigit(ch))*».

2.3. Переменные в образцах

При сопоставлении с образцом можно использовать переменные, например:

```
val somePair = (42, “I am a String”)

somePair match {
  case (i, s) =>
    println(“I found this:”)
    println(i)
    println(“and this:”)
    println(s)
}
```

Пример выше показывает:

1) можно использовать сопоставление с образцом для работы со значениями в кортежах;

2) при сопоставлении можно указать несколько требуемых действий.

Если нам нужно только первое значение из пары, то на место второго значения можно поставить символ _.

```
somePair match {
  case (i, _) => println(“First element of pair is: ” + i)
}
```

В данном случае ко второму значению пары обратиться никак нельзя.

2.4. Сопоставление с типом

Предыдущий пример разбора пары значений будет сопоставляться с парой любых значений. Допустим, надо сделать так, чтобы он сопоставлялся только с парой типа (Int, String). Для этого можно указать тип в образце для сопоставления:

```
val somePair = (42, “I am a String”)

somePair match {
  case (i: Int, s: String) => {
    println(“I found this:”)
    println(i)
    println(“and this:”)
    println(s)
  }
}
```

Теперь наш пример будет сопоставляться только с парами типа (Int, String), а на все остальные возбуждать исключение MatchError (т.к. не указан общий образец _).

Рассмотрим более практичный пример указания типа в сопоставлении с образцом. Допустим, есть следующая структура классов:

```
abstract class Parent

class FirstChild extends Parent {
  def methodOfFirstChild() = println("I'm The First Child!")
}

class SecondChild extends Parent {
  def methodOfSecondChild() = println("I'm The Second Child!")
}
```

От класса Parent наследуются два класса FirstChild и SecondChild, каждый из которых определяет свой собственный метод.

Предположим, что у нас список типа List[Parent] и необходимо обработать каждый элемент этого списка (в нашем случае вызвать метод класса), при этом необходимо знать с каким именно классом мы работаем

(FirstChild или SecondChild). В Java пришлось бы делать проверки isInstance и приведение типов или оборачивать необходимое действие в интерфейс, общий для обоих классов. В Scala можно «извлечь» тип при сопоставлении с образцом.

```
val collection: List[Parent] = List(
  new SecondChild(),
  new SecondChild(),
  new FirstChild()
)

for (obj <- collection) {
     obj match {
       case o: FirstChild  => o.methodOfFirstChild()
       case o: SecondChild => o.methodOfSecondChild()
      }
   }
```

2.5. Сопоставление со списками, массивами

Ниже представлен пример сопоставления списка с различными образцами:

```
val lst: List[Int] = ...

lst match {
  case Nil          => println(“пустой список”)
  case head :: Nil  =>
    println(“Список содержит один элемент: ” + head)
  case head :: tail =>
    println(“Список содержит голову ” + head)
    println(“и хвост ” + tail)
  case head :: second :: Nil =>
    println("Список содержит ВСЕГО 2 элемента")
  case head :: second :: rest =>
    println("Список содержит КАК МИНИМУМ 2 элемента")
}
```

В данном примере Nil – явный конец списка, а tail – имя значения, в котором содержится хвост списка (который может быть пустой, а может быть и не пустой). Наиболее явно это различие демонстрируют последние два образца и соответствующие им сообщения в println.

Аналогичным образом можно проводить сопоставление с элементами массива:

```
val arr = Array(4, 5, 7, 8)
```

```
arr match {
  case Array(x, y, _*) => println(x); println(y)
}
```

Значение _* означает «сколько угодно (в том числе и 0) элементов в конце» и может применяться только в конце образца. Образец Array(x, _*, z) вызовет ошибку. К сожалению, невозможно с помощью образца получить «хвост» массива, как это было со списками.

Также стоит отметить, что данный список не совпадёт с образцом Array(x, y, z), т.к. образец подразумевает, что массив имеет только 3 элемента.

2.6. Объявление значений через сопоставление с образцом

Механизм сопоставления с образцом можно использовать вне match для раскрытия значений.

```
// данная строка создаст две раздельные переменные d: Int и s: String
// и присвоит им соответствующие значения
val (d, s) = (42, “String”)


// извлекает из массива первые два значения и присваивает их переменным
val Array(d, s) = Array(42, 13)

val Array(d, s) = Array(42, 13, 14)      // ошибка
val Array(d, s, _*) = Array(42, 13, 14)  // OK
```

2.7. Case-классы в языке Scala

Case-классы это обычные классы, которые обладают рядом особенностей:

1) при объявлении case-класса автоматически создается объект-компаньон с методом apply, который позволяет создавать экземпляры объекта без помощи new;

2) case-класс и его значения можно использовать в сопоставлении с образцом;

3) генерируются методы toString, hashCode, equals и copy, если они не были заданы явно.

Приведём сравнение методов toString обычного класса и case-класса:

```
class First(d: Int)
case class Second(d: Int)
```

```
// ...

val v1 = new First(1)
val v2 = Second(2)

println(v1.toString)  // выводит main.scala.First@7907ec20
println(v2.toString)  // выводит Second(2)
```

Сгенерированные методы equals и hashCode применяются для сравнения двух экземпляров case-класса по значениям их аргументов (а не по значению ссылки, как это обычно происходит).

Таким образом, применение case-классов упрощает жизнь.

Рассмотрим пример, демонстрирующий применение case-классов и сопоставления с образцом на примере структуры бинарного дерева:

```
abstract class Tree
case class Leaf(data: Int) extends Tree // лист дерева
case class Node(data: Int, left: Tree, right: Tree) extends Tree // узел дерева

val tree = Node(
  5,
  Node(
    3,
    Leaf(2),
    Leaf(4)
  ),
  Node(
    7,
    Leaf(6),
    Leaf(10)
  )
)

def printTree(tree: Tree, lvl: Int): Unit = {
  tree match {
    case Node(d, l, r) =>
      println("-" * lvl + d)
      printTree(l, lvl+1)
      printTree(r, lvl+1)
    case Leaf(d) => println("-" * lvl + d)
  }
}

printTree(tree, 0)
```

Небольшие пояснения к этому фрагменту кода:

println("-" * lvl + d)

В Scala можно умножать строку s на целочисленное число n. В результате получается повторение исходной строки s n раз.

Приведённый код выводит в консоль следующее:

```
5
-3
--2
--4
-7
--6
--10
```

2.8. Класс Option в языке Scala

В Scala есть класс Option, который является обёрткой для любых других значений/классов. Его задача – представлять значения, которые могут быть, а могут и не быть. Класс Option имеет два дочерних класса Some и None. Some содержит внутри себя искомое значение, а None представляет несуществующее значение (почти как null).

Например, чтобы получить значение по ключу ассоциативного массива, можно использовать метод get, который возвращает Option:

```
val dictionary: Map[Char, String] = Map(
  'a' -> "alphabet",
  'b' -> "behemoth",
  's' -> "Scala"
)

dictionary.get(‘a’)  // вернёт Some(“alphabet”)
dictionary.get(‘g’)  // вернёт None
```

Также Option можно использовать в сопоставлении с образцом:

```
dictionary.get(‘s’) match {
  case Some(str) => println(str)
  case None => println(“ничего не найдено”)
}
```

3. ЗАДАНИЕ НА ЛАБОРАТОРНУЮ РАБОТУ

Все задания необходимо выполнять с помощью сопоставления с образцом (match) и не использовать условные выражения (if).

3.1. Написать функцию типа (List[(Int, Int)]) => List[Option[Double]], которая принимает на вход список из пар двух целочисленных значений и возвращает список результатов деления первого числа пары на второе в виде Option (Some[Double], если второй элемент пары не равен нулю, или None, если второй элемент пары равен нулю).

3.2. Написать функцию типа (List[Option[Double]]) => List[String], которая принимает на вход список Option'ов типа Double (результаты работы функции из п.3.1) и преобразует его в список строк по следующему правилу: значения Some преобразуются в строку «Результат деления = __число_из_Some__», а значения None преобразуются в строку «Деление на ноль невозможно».

3.3. Реализовать задание из п.3.1, но вместо пар использовать case-класс. Исследовать такую реализацию задания, сделать выводы.

3.4. На основе case-классов реализовать структуру дерева вычислений. Написать функцию, которая это дерево преобразует в строку обратной польской нотации.

4. КОНТРОЛЬНЫЕ ВОПРОСЫ

4.1. Опишите особенности оператора match в Scala.

4.2. Приведите пример использования ограничителей в Scala.

4.3. Каким образом используются переменные при сопоставлении с образцом?

4.4. Приведите пример сопоставления с типом в Scala.

4.5. Приведите пример сопоставления со списками в Scala.

4.6. Приведите пример сопоставления с массивами в Scala.

4.7. Каким образом используется сопоставление с образцом для объявления значений в Scala?

4.8. Что такое Case-класс в Scala? Перечислите отличия Case-классов от обычных классов в Scala.

4.9. Опишите назначение и основные особенности класса Option.

4.10. Приведите пример использования класса Option.

ЛАБОРАТОРНАЯ РАБОТА № 5
ИССЛЕДОВАНИЕ ОСОБЕННОСТЕЙ РЕАЛИЗАЦИИ ОПЕРАЦИЙ КАРРИРОВАНИЯ, ЧАСТИЧНОГО ПРИМЕНЕНИЯ ФУНКЦИЙ И ЗАМЫКАНИЙ В ЯЗЫКЕ SCALA

1. ЦЕЛЬ РАБОТЫ

Исследовать понятия каррирования, частичного применения и замыкания в функциональном программировании. Приобрести практические навыки использования механизмов каррирования, частичного применения и замыкания в языке Scala.

2. ОБЩИЕ ПОЛОЖЕНИЯ

2.1. Частичное применение функций

В функциональном программировании при вызове функции можно задать не все аргументы, а только их часть, т. е. несколько первых аргументов мы можем указать, а остальные опустить. Таким образом, результатом вызова такой функции будет являться функция, в которую необходимо передать остальные аргументы. Такая функция, результат вызова которой будет представлять собой функцию, в которую можно передать все оставшиеся аргументы называется частично применённой.

2.1.1 Частичное применение функции в Scala

Язык Scala также допускает частичное применение функций. Рассмотрим пример, представленный ниже:

```
def price(product : String) : Double =
  product match {
    case "apples" => 140
    case "oranges" => 223
}

def withTax(cost: Double, state: String) : Double =
  state match {
    case "NY" => cost * 2
    case "FL" => cost * 3
}

val locallyTaxed = withTax(_: Double, "NY")
```

```
val costOfApples = locallyTaxed(price("apples"))

// выдаст ошибку java.lang.AssertionError: assertion failed
// если условие не выполнится
assert(Math.round(costOfApples) == 280)
```

В примере сначала создаётся функция price, которая возвращает отображение между product (товар) и price (цена). Затем объявляется функция withTax(), которая принимает аргументы cost и state. Пусть для определенного исходного файла известно, что вычисления будут производиться с налогами (taxes) только одного штата (state). Вместо того чтобы "каррировать" лишний аргумент при каждом вызове, можно "частично применить" аргумент state и возвратить версию функции, в которой значение state зафиксировано. Функция locallyTaxed принимает единственный аргумент cost.

2.2. Каррирование

2.2.1. Каррирование в λ-исчислении

Если функция f имеет тип A1 → (A2 → (. . .(An → B). . .)), то, чтобы полностью вычислить значение f(a1, a2, . . . , an), необходимо последовательно провести вычисление (. . .(f(a1)a2). . .)an. И результатом вычисления будет объект типа B. Соответственно, выражение, в котором все функции рассматриваются как функции одного аргумента, а единственной операцией является аппликация (применение), называются выражениями в форме «оператор — операнд». Такие функции получили название «каррированные», а сам процесс сведения типа функции к виду, приведенному выше — каррирование в честь Хаскелля Карри, который переоткрыл, развил и популяризовал данное понятие (хотя первоотрывателем является Моисей Эльевич Шейнфинкель).

Если вспомнить λ-исчисление то обнаружится, что в нем уже есть математическая абстракция для аппликативных форм записей. Например:

$f(x) = x^2 + 5 \Rightarrow \lambda x.(x^2 + 5)$,

f(x, y) = x + y ⇒ λy.λx.(x + y),

$f(x, y, z) = x^2 + y^2 + z^2 \Rightarrow \lambda z.\lambda y.\lambda x.(x^2 + y^2 + z^2)$.

Таким образом, каррирование по сути образует цепочку функций с одним аргументом.

2.2.2. Каррирование в Scala

В Scala функции позволяют задавать несколько списков аргументов виде наборов круглых скобок. Если вы вызываете какую-либо функцию с

меньшим количеством аргументов, чем задано для нее, эта функция берет списки опущенных аргументов в качестве своих аргументов. Рассмотрим пример из документации Scala:

```
def filter(xs: List[Int], p: Int => Boolean): List[Int] =
  if (xs.isEmpty) xs
  else if (p(xs.head)) xs.head :: filter(xs.tail, p)
  else filter(xs.tail, p)

def modN(n: Int)(x: Int) = ((x % n) == 0)

val nums = List(1, 2, 3, 4, 5, 6, 7, 8)
println(filter(nums, modN(2)))
println(filter(nums, modN(3)))
```

В примере функция filter() рекурсивно применяет переданные критерии фильтрации. Функция modN() определена с двумя списками аргументов. При вызове функции modN с использованием функции filter(), передаётся единственный аргумент. Функция filter() принимает в качестве своего второго аргумента функцию с аргументом Int и возвращает значение Boolean соответствующее сигнатуре каррированной функции, которая и передаётся. Таким образом,функция modN берёт недостающие аргументы уже будучи вызванной внутри функции filter, но уже под другим именем — «p».

2.3. Замыкания

2.3.1. Замыкания в λ-выражениях

Замыкания это некая функция с какими-то переменными, но эти переменные не являются локальными для данной функции, а определены извне, т. е. в той же области видимости, где определена функция. Такие переменные принято называть контекстом.

Замыкание — (англ closures) это лямбда-выражение, которое содержит помимо связанной переменной и тела функции еще и контекст.

Замыкание тесно связано с понятием слабой заголовочной нормальной формы.

Выражение, не имеющее свободных переменных (замкнутое) находится в слабой заголовочной нормальной форме (СЗНФ), если оно имеет один из следующих видов:

- константа: c;
- определение функции: λx.E;
- частичное применение функции: P E_1 E_2 … E_k

Например, выражение λx.((λy. λx.+ x y) x) находится в СЗНФ.

Вычисления, происходящие при исполнении программы в «ленивых» вычислениях, соответствуют редукции выражения в нормальном порядке до приведения к СЗНФ, дополненной эффектом «разделения» значений переменных при подстановке аргумента, еще не находящегося в СЗНФ.

2.3.2. Замыкания в Scala

Пример простого замыкания представлен ниже:

```
val outer = 10
val myFuncLiteral = (y: Int) => y * outer
val result = myFuncLiteral(2)
```

После выполнения данного фрагмента кода функция myFuncLiteral «захватит» из области видимости своего определения значение переменной outer и в результате result будет равен 20.

3. ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

3.1. Реализовать на языке Scala функцию, возвращающую количество собственных вызовов.

3.2. Реализовать функцию на Scala, используя механизм каррирования:

Вариант 1: Функция возведения в степень. Пример вызова функции на псевдокоде:

возведениеВСтепень(3)(2)(1)(0) → 1.

Вариант 2: Функция перевода десятичного числа в n-ичное представление. Пример вызова:

десятичноеЧислоВNичное(число, порядок_числа).

Используя каррирование, напишите функцию преобразования десятичного числа в двоичное.

3.3. Реализовать функцию, которая принимает один параметр, начальное число Z, и возвращает функцию, которая принимает один параметр, длину списка, и возвращает список заданной длины, содержащий случайные числа. Модуль разности случайных чисел и начального числа Z не должен превышать 5.

3.4. Составить отчет, содержащий результаты выполнения программы.

4. КОНТРОЛЬНЫЕ ВОПРОСЫ

4.1. Что понимается под частичным применением функций?

4.2. Объясните механизм частичного применения функций в Scala.

4.3. Что такое каррирование?
4.4. Чем каррирование отличается от частичного применения?
4.5. Приведите пример каррирования в Scala.
4.6. Что такое замыкание в лямбда-исчислении?
4.7. Приведите пример замыкания.
4.8. Дайте определение слабой заголовочной нормальной формы
4.9. Приведите пример реализации замыкания в Scala

ЛАБОРАТОРНАЯ РАБОТА № 6
ИССЛЕДОВАНИЕ ОСОБЕННОСТЕЙ РЕАЛИЗАЦИИ ПАРАМЕТРИЗОВАННЫХ КЛАССОВ И ФУНКТОРОВ В ЯЗЫКЕ SCALA

1. ЦЕЛЬ РАБОТЫ

Исследовать основные свойства и способы построения функторов в функциональном программировании. Получить практические навыки программной реализации параметризованных классов и функторов в языке Scala.

2. ОБЩИЕ ПОЛОЖЕНИЯ

2.1. Параметризованные классы в языке Scala

Параметризованные классы – классы, которые могут оперировать над значениями различных типов.

В качестве примера реализуем класс, который получает в конструкторе значение и имеет метод для печати полученного значения. Особенность класса заключается в том, что он создан не для работы с каким-то конкретным типом, а может работать со всеми типами.

```
case class MyPrinter[T](attribute: T) {
  def print() = println(attribute)
}

// Пример 1
MyPrinter(1).print()         // 1
// Пример 2
MyPrinter("test").print()    // test
// Пример 3
MyPrinter[Double](1).print()  // 1.0
```

При создании класса не обязательно указывать конкретный тип параметра (примеры 1 и 2), но, если очень хочется, то можно (пример 3).

Добавим в класс MyPrinter метод, который будет печатать два значения - атрибут класса и аргумент метода:

```
case class MyPrinter[T](attribute: T) {
  def print()       = println(attribute)
  def print2(arg: T) = println(attribute + " and " + arg)
```

```
}

MyPrinter("test").print2("testtest")  // test and testtest
```

Метод print2() может принимать аргумент только того типа, которому принадлежит атрибут класса (и там и там значения имеют один и тот же тип T). Попытка распечатать вместе String и Double с помощью MyPrinter(“test”).print2(3.0) является ошибкой.

Помимо классов, можно также параметризовать и функции, так что вышеуказанную ошибку можно решить следующим способом:

```
case class MyPrinter[T](attribute: T) {
  def print()          = println(attribute)
  def print2(arg: T)   = println(attribute + " and " + arg)
  def print3[R](arg: R) = println(attribute + " and " + arg)
}

// Пример 1
MyPrinter("test").print3(3)              // test and 3
// Пример 2
MyPrinter("test").print3[Double](3)      // test and 3.0
// Пример 3
MyPrinter("test").print3("another string")  // test and another string
```

Как видно из примера 3, типы T и R не обязательно должны быть различны.

2.2. Ковариативный функтор

С точки зрения синтаксиса, ковариативный функтор это такой тип X, который параметризирован типом T и имеет метод (назовём его map) с особой сигнатурой:

```
trait X[T] {
  def map[R](f: T => R): X[R]
}
```

Этот тип можно представлять себе как источник, откуда можно **что-то** взять, и это **что-то** имеет тип T. Но, если надо получить не тип T, а тип R, то мы на выходе этого источника ставим преобразующую функцию из T в R и получаем новый источник, который даёт тип R.

Визуально ковариативный функтор представлен на рисунке 1.

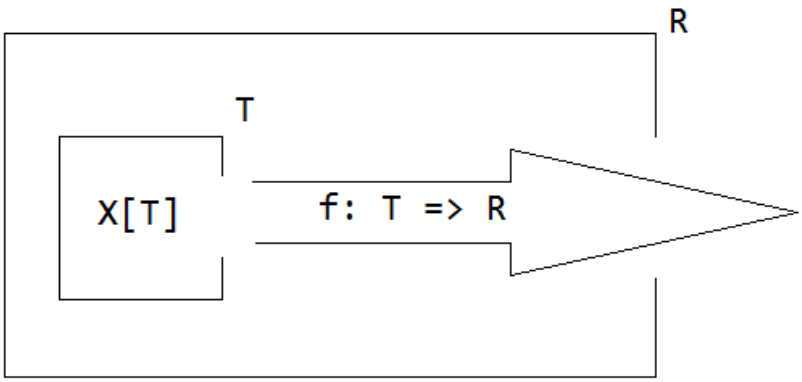

Рисунок 1 – Ковариативный функтор

Примерами ковариативных функторов являются Scala классы List, Option, Try.

Пример с Option:

```
import java.lang.Integer.toHexString

object Demo extends App {
  val k: Option[Int] = Option(100500)
  val s: Option[String] = k map toHexString
}
```

Или так:

```
import java.lang.Integer.toHexString

object Demo extends App  {
  val k: Option[Int] = Map("A" -> 0, "B" -> 1).get("C")
  val s: Option[String] = s map toHexString
}
```

Пример с List:

```
import java.lang.Integer.toHexString

object Demo extends App {
  val k: List[Int] = List(0, 42, 100500)
  val s: List[String] = k map toHexString
}
```

Класс Try похож на класс Option. Но он представляет собой не значение, которое может быть или не быть, а действие (вычисление), которое

может завершиться успешно, или не успешно, например, выбросив исключение.

Если в Option класс Some отвечает за существующее значение, а None за отсутствующее, то в Try класс Success отвечает за успешное завершение операции, а Failure за ошибку во время выполнения операции.

Рассмотрим пример применения Try для чтения файла. Если файл существует и операция завершилась успешно, то match совпадёт с образцом Success(lines), в lines находится содержимое файла – результат выполнения Source.fromFile(filename).getLines.toList. Если файла не существует или произошло другое исключение во время операции чтения, то match совпадёт с образцом Failure(f), в f находится сообщение об ошибке:

```
import scala.io.Source
import scala.util.{Try,Success,Failure}

object Test extends App {

  def readTextFile(filename: String): Try[List[String]] = {
    Try(Source.fromFile(filename).getLines.toList)
  }

  val filename = "/etc/passwd"
  readTextFile(filename) match {
    case Success(lines) => lines.foreach(println)
    case Failure(f) => println(f)
  }

}
```

Пример с Try как с ковариативным функтором:

```
import java.lang.Integer.toHexString
import scala.util.Try

object Demo App {
  val k: Try[Int] = Try(100500)
  val s: Try[String] = k map toHexString
}
```

Или так:

```
import java.lang.Integer.toHexString
import scala.util.Try
```

```
object Demo extends App {
  def f(x: Int, y: Int): Try[Int] = Try(x / y)
  val s: Try[String] = f(1, 0) map toHexString
}
```

Для того, чтобы тип X считался ковариативным функтором, также должны соблюдаться два правила:

Правило №1: Закон идентичности (Identity Law)

Пусть fun[T] является ковариативным функтором, тогда

fun.map(identity) ≡ fun

где identity - функция, которая возвращает свой аргумент (def identity[A](x: A) = x).

Другими словами: fun.map(identity) не должно ничего менять внутри функтора.

Например, следующий класс не является ковариативным функтором, т.к. после применения fun.map(identity) изменится значение переменной inc.

```
class Example[T](value: T, inc: Int = 0) {
  def map[R](f: T => R): Example[R] = new Example[R](f(value), inc + 1)
}
```

Правило №2: Закон композиции (Composition Law)

Пусть fun[T] является ковариативным функтором, f - функция типа T => R, g - функция типа R => Q, тогда:

fun.map(f).map(g) ≡ fun.map(f andThen g)

где andThen – стандартный оператор Scala. Выражение f andThen g возвращает функцию, которая аналогична последовательному применений функций f и g. То есть (f andThen g)(x) то же самое что и g(f(x)).

Иными словами, функтор, который отображают функцией f, а затем g, эквивалентен функтору, который отображают композицией функций f и g.

Рассмотрим оба правила на примере бинарного дерева:

```
trait Tree[T] {
  def map[R](f: T => R): Tree[R]
}
case class Node[T](value: T, fst: Tree[T], snd: Tree[T]) extends Tree[T] {
  def map[R](f: T => R) = Node(f(value), fst.map(f), snd.map(f))
}
case class Leaf[T](value: T) extends Tree[T] {
  def map[R](f: T => R) = Leaf(f(value))
```

```
}
```

В данной реализации при отображении дерева с помощью произвольной функции сохраняется его структура и обновляется значения в узлах и листьях. Изменим метод map следующим образом:

```
case class Node[T](value: T, fst: Tree[T], snd: Tree[T]) extends Tree[T] {
  def map[R](f: T => R) = Node(f(value), snd.map(f), fst.map(f))
}
```

С данной реализацией метода map бинарное дерево не является ковариативным функтором, т.к. с каждым отображением правое и левое поддерево узлов меняется местами.

Следовательно, tree.map(identity) вернёт дерево с тем же значением элементов, но поменяными поддеревьями (не соблюдается первое правило). В свою очередь, tree.map(f).map(g) изменит структуру дерева дважды (по одному разу на каждом вызове map), а tree.map(f andThen g) - всего один раз (не соблюдается второе правило).

Чтобы увидеть пользу от применения ковариативных функторов, рассмотрим следующий код:

```
val f = (x: Double) => x*2 + 3
val g = (x: Double) => x/5 - 7

// Список из Double
val list = (1 to 1000000 toList) map (_.toDouble)

// Пример 1
list.map(f).map(g)
// Пример 2
list.map(f andThen g)
```

В первом примере мы 2 раза проходим по всему списку чисел и 2 раза применяем функцию. Во втором примере мы 1 раз проходим по всему списку чисел и 1 раз применяем функцию. Зная то, что List является ковариативным функтором и зная правила, которым должен удовлетворять ковариативный функтор, мы можем оптимизировать первый пример и быть уверенными, что результат останется прежним.

2.3. Контравариативный функтор

С точки зрения синтаксиса, контравариативный функтор это такой тип X, который параметризирован типом T и имеет метод (назовём его contramap) с особой сигнатурой:

```
trait X[T] {
  def contramap[R](f: R => T): X[R]
}
```

Если ковариативный функтор можно представить как источник, то контравариативный функтор можно представить как приёмник, который принимать тип T. Если на входе (не на выходе, как это было с ковариативным функтором) поставить преобразующую функцию из R в T, то получится новый приёмник, который принимает тип R.

Визуально контравариативный функтор представлен на рисунке 2.

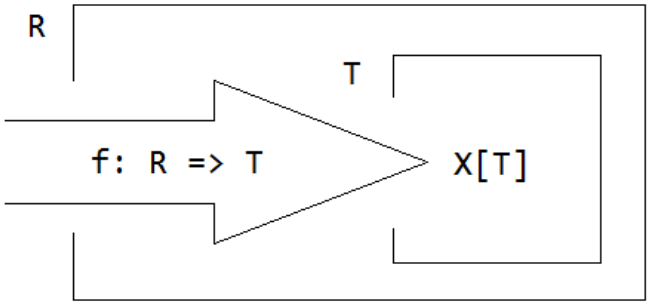

Рисунок 2 – Контравариативный функтор

Контравариативный функтор можно продемонстрировать на примере стандартного в scala класса Ordering. В нём метод Ordering.by является реализацией contramap для Ordering:

```
def by[T, S](f: T => S)(implicit ord: Ordering[S]): Ordering[T]
```

Смысл в следующем: если мы можем сравнивать (упорядочивать) элементы типа S, и существует функция, которая позволяется преобразовать тип T в тип S, тогда мы можем с помощью этого сравнивать тип T.

Здесь существуют два концептуальных преобразования. На высшем уровне мы хотим преобразовать Ordering[S] в Ordering[T], но для того, чтобы сделать это, нам нужен способ преобразовать T в S. Отсюда и «контр» в названии – из-за разности в направлениях преобразований.

Говоря общими словами, если у нас есть какая-то абстракция F[A] над типом A и функция, отображающая тип B в тип A, то мы можем построить абстракцию F[B] над типом B.

Пример из реального мира: есть числа, которые можно сравнивать между собой, и есть деньги, которые можно привести к численному виду: например, 1 рубль можно интерпретировать числом 100 (100 копеек). При этом:

1) Есть возможность сравнивать числа,

2) Есть возможность приводить деньги к численному виду.

Исходя из (1) и (2) можно реализовать сравнение денег. Это и делает contramap.

Исходя из этих положений можно реализовать программный пример. Сначала напишем класс Money:

```
case class Money(amount: Int)
```

Scala изначально знает, как сравнивать Int. На основе этого мы хотим указать способ, которым будут сравнивать экземпляры Money. То есть получить Ordering[Money] на основе Ordering[Int]. Для этого нужно реализовать преобразование Money в Int. В нашем случае это достаточно легко:

```
val funcForContramap: Money => Int = (money) => money.amount
```

Ordering[Int] в неявном виде уже находится в контексте нашей программы. Всё что осталось сделать – вызвать функцию, выполняющую contramap в классе Ordering, то есть Ordering.by.

```
implicit val moneyOrd: Ordering[Money] = Ordering.by(funcForContramap)
```

Теперь мы можем сравнивать экземпляры класса Money с помощью функции меньше «<»:

```
scala> import scala.math.Ordered._
import scala.math.Ordered._

scala> Money(13) < Money(20)
res0: Boolean = true

scala> Money(23) < Money(20)
res1: Boolean = false
```

Один из вариантов метода Ordering.by может быть следующим (реальная реализация сложнее из-за оптимизаций):

```
def by[T, S](f: T => S)(implicit ord: Ordering[S]): Ordering[T] =
  new Ordering[T] {
    def compare(x:T, y:T) = ord.compare(f(x), f(y))
  }
```

Как видно из кода, метод by создаёт экземпляр Ordering и реализует в нём метод compare, который делегирует сравнение исходному экземпляру класса Ordering, но перед этим производит преобразование с помощью функции f.

Для того, чтобы тип X считался контравариативным функтором, также должны соблюдаться два правила:

Правило №1: Закон идентичности (Identity Law)

Пусть fun[T] является контравариативным функтором, тогда

fun.contramap(identity) ≡ fun

где identity - функция, которая возвращает свой аргумент (def identity[A](x: A) = x).

Другими словами: fun.contramap(identity) не должно ничего менять внутри функтора.

Правило №2: Закон композиции (Composition Law)

Пусть fun[T] является контравариативным функтором, f - функция типа Q => R, g - функция типа R => T, тогда:

fun.contramap(g).contramap(f) ≡ fun.contramap(f andThen g)

где f andThen g возвращает функцию, которая аналогична последовательному применений функций f и g. То есть (f andThen g)(x) то же самое что и g(f(x)).

Иными словами, последовательное отображение контравариантного функтора парой функций эквивалентно отображению инвертированной композицией функций.

2.4. Инвариативный функтор

Инвариативный функтор — это объединение ковариантивного и контравариативного функтора (и источник и приёмник одновременно). С точки зрения синтаксиса он представляет собой следующее:

```
trait X[T] {
  def xmap[R](f: T => R, g: R => T): X[R]
}
```

Для получение нового инвариативного функтора с типом R необходимо предоставить функции преобразования на входе (g: R => T) и на выходе (f: T => R). Схематично инвариативный функтор изображён на рисунке 3.

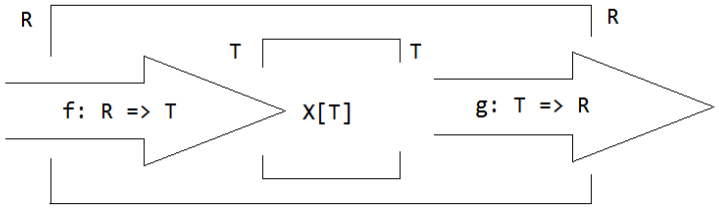

Рисунок 3 – Инвариативный функтор

То есть функтор F[R] сводится к исходному функтору F[T] с помощью пары преобразований из R в T и из T в R.

В качестве примера можно реализовать контейнер для значений:

```
trait Holder[T] { self =>
  def put(arg: T): Unit
  def get: T
  def xmap[R](f: T => R, g: R => T): Holder[R] = new Holder[R] {
    override def put(arg: R): Unit = self.put(g(arg))
    override def get: R = f(self.get)
  }
}
```

Пример контейнера для строк:

```
class StrHolder(var value: String = null) extends Holder[String] {
  override def put(arg: String): Unit = {value = arg}
  override def get: String = value
}
```

Демонстрация

```
object Demo extends App {
  val f: String => Int = Integer.parseInt(_, 16)
  val g: Int => String = Integer.toHexString

  val s: Holder[String] = new StrHolder
  val k: Holder[Int] = s xmap (f, g)

  k put 100500
  println(k.get)
}
```

Получился контейнер Int, который хранит их в строковом шестнадцатеричном формате.

Инвариативный функтор также должен соблюдать два правила:

Правило №1: Закон идентичности (Identity Law)

Пусть fun[T] является инвариативным функтором, тогда

fun.xmap(identity, identity) ≡ fun

где identity - функция, которая возвращает свой аргумент (def identity[A](x: A) = x).

Другими словами: fun.xmap(identity) не должно ничего менять внутри функтора.

Правило №2: Закон композиции (Composition Law)

Пусть fun[T] является инвариативным функтором, тогда для данного функтора и функций f1: T => R, g1: R => T, f2: R => Q, g2: Q => R должно выполняться следующее соотношение:

fun.xmap(f1, g1).xmap(f2, g2) ≡ fun.xmap(f2 compose f1, g1 compose g2),

где f compose g возвращает функцию, которая аналогична последовательному применений функций g и f. То есть (f compose g)(x) то же самое что и f(g(x)).

3. ЗАДАНИЕ НА ЛАБОРАТОРНУЮ РАБОТУ

3.1. Написать типизированный класс Box[T] который позволяет сохранять в себя значение типа T (метод save) и получать сохранённое значение (метод get).

3.2. Преобразовать класс Box из п.1 в ковариативный функтор. Исследовать свойства функтора, доказать, что он является ковариативным функтором.

3.3. Преобразовать класс Box в контравариативный функтор. Исследовать свойства функтора, доказать, что он является контравариативным функтором.

3.4. Преобразовать класс Box в инвариативный функтор. Исследовать свойства функтора, доказать, что он является инвариативным функтором.

5. КОНТРОЛЬНЫЕ ВОПРОСЫ

4.1. Дайте определение параметризованных классов в Scala.

4.2. Приведите пример параметризованного классов.

4.3. Что такое ковариативный функтор?

4.4. Каким правилам должен отвечать ковариативный функтор?

4.5. Приведите примеры классов Scala, которые являются ковариативными функторами.

4.6. Что такое контравариативный функтор?

4.7. Каким правилам должен отвечать контравариативный функтор?

4.8. Что такое инвариативный функтор?

4.9. Каким правилам должен отвечать инвариативный функтор?

ПРИЛОЖЕНИЕ А

Вспомогательный код к выполнению лабораторной работы №2

```
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", 2006, "FIL", 'F', 1, true),
      (1, "Гриша", 2005, "AVT", 'M', 2, true),
      (2, "Настя", 2004, "MTS", 'F', 3, false),
      (3, "Коля", 2002, "MTS", 'M', 1, false),
      (4, "Миша", 2002, "AVT", 'M', 3, true),
      (5, "Оля", 2000, "FIL", 'F', 3, false),
      (6, "Маша", 2001, "AVT", 'F', 5, true),
      (7, "Таня", 2004, "FIL", 'M', 4, true),
      (8, "Женя", 2000, "FIL", 'F', 4, true),
      (9, "Света", 2007, "AVT", 'F', 3, true),
      (10, "Аня", 2003, "MTS", 'F', 4, false),
      (11, "Лена", 2003, "AVT", 'F', 2, true),
      (12, "Сергей", 2005, "FIL", 'M', 3, false),
      (13, "Влад", 2004, "FIL", 'M', 5, false),
      (14, "Гена", 2003, "MTS", 'M', 1, true),
      (15, "Дима", 2006, "AVT", 'M', 5, false),
      (16, "Катя", 2001, "FIL", 'F', 4, false),
      (17, "Артём", 2005, "MTS", 'M', 3, true),
      (18, "Диана", 2006, "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))
    )
  }
}
```

ПАРАДИГМЫ СОВРЕМЕННЫХ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ

Методические указания
к выполнению лабораторного практикума по дисциплине

Составитель : **Строганов** Виктор Александрович

В авторской редакции

Изд. № 58/2024. Объем 5 п.л. Усл. печ. л. 4,65. Уч.-изд. л. 4,9.
Издательский центр ФГАОУ ВО «Севастопольский государственный университет»