Оглавление

[**Prolog** 3](#_Toc168507951)

[1. Опишите понятие и структуру фактов в языке Пролог. Расскройте основные возможные типы, опишите понятие атом. Расскажите принцип работы терминальной машины Swi-Prolog, объясните каким образом происходит обработка вопросов. 3](#_Toc168507952)

[2. Опишите смысл термина унификация, приведите показательные примеры. Объясните, как задаются предикаты, что такое правила и каким образом происходит работа с ними. 5](#_Toc168507953)

[3. На примере числовых алгоритмов объясните смысл рекурсии вверх и рекурсии вниз в прологе. 7](#_Toc168507954)

[4. Раскройте на примерах понятие backtracking, оператор отсечения и смысл его применения. 8](#_Toc168507955)

[5. Опишите принцип работы со списками Черча в Swi-prolog. Покажите реализации встроенных предикатов работы со списками на основе механизма унификации(append, reverse nth0). 10](#_Toc168507956)

[6. Объясните принцип работы со строками. Покажите на примерах основные встроенные предикаты работы со строками. 12](#_Toc168507957)

[7. Покажите, каким образом происходит построение стандартных комбинаторных объектов средствами Swi-prolog. 14](#_Toc168507958)

[8. Покажите основные принципы реализации переборных алгоритмов на графах средствами пролога. 17](#_Toc168507959)

[9. Раскройте понятия статические и динамические факты. Поясните на примерах принцип работы с динамическими фактами. 18](#_Toc168507960)

[10. Объясните принцип работы предикатов var, nonvar, atom, atomic, name, functor, arg, repeat. 20](#_Toc168507961)

[**Котлин** 22](#_Toc168507962)

[1. Способы написания функций в котлине, разные виды тела функции. Функциональные аргументы и значения. Множество аргументов. Один аргумент. 22](#_Toc168507963)

[2. Структура описания класса в котлине. Переменные и значения. Разница и применение в функциональном подходе. Геттеры, сеттеры, свойства. Дата классы. 24](#_Toc168507964)

[3. Перечисления и конструкция when. Операторы проверки класса и преобразования к классу. 26](#_Toc168507965)

[4. Исключения в Котлин. Структура исключений. 28](#_Toc168507966)

[5. Работа с массивами в котлин. Работа со списками в котлин. 29](#_Toc168507967)

[6. Работа с множествами в котлин. Работа с ассоциативными массивами в котлин 32](#_Toc168507968)

[7. Общая структура классов коллекций в котлин. 34](#_Toc168507969)

[8. Наследование в котлин. Делегация в котлин. 35](#_Toc168507970)

[9. Понятие функции расширения в котлин. Понятие и особенности локальных функций в котлине. Функции расширения и локальные функции. 37](#_Toc168507971)

[10. Регулярные выражения в котлин. 38](#_Toc168507972)

[11. Интерфейсы и полиморфизм в котлин. 40](#_Toc168507973)

[12. Общий обзор наследования, абстрактных классов, интерфейсов и их реализации в котлин. Набор модификаторов. Сравнение с java 41](#_Toc168507974)

[13. Модификаторы доступа в котлин. Запечатанные классы. 43](#_Toc168507975)

[14. Способы задания конструкторов в котлине. 45](#_Toc168507976)

[15. Структура классов в котлин. Сравнение объектов в котлин. 46](#_Toc168507977)

[16. Совместное объявление класса и экзмепляра в котлин. Задание фабричных методов в котлин. 48](#_Toc168507978)

[17. Лямбда выражения. Передача лямбда выражений. Замыкание. Краткие способы записи лямбда выражений. 50](#_Toc168507979)

[18. Методы работы с коллекциями, принимающие функции, как аргумент. Filter, map. Count, sort, max 51](#_Toc168507980)

[19. Методы работы с коллекциями, принимающие функции, как аргумент. GroupBy, reduce, fold. 53](#_Toc168507981)

[20. Работа с последовательностями в котлин, в java 54](#_Toc168507982)

[21. Лямбда выражения с получателями. 55](#_Toc168507983)

[22. Поддержка значения null 57](#_Toc168507984)

[23. Базовые типы в котлин. 58](#_Toc168507985)

[24. Перегрузка арифметических операторов. 59](#_Toc168507986)

[**Лямбда** 60](#_Toc168507987)

[1. Опишите основные принципы функционального программирования и вытекающие из них преимущества и недостатки. Опишите понятия высшая функция, чистая 60](#_Toc168507988)

[2. Опишите математические предположения, которые привели к лямбда исчислению и объясните формат записи лямбда выражений. Дайте определение лямбда терма. Опишите соглашения о возможности опускать скобки, принятые в лябмда выражении. 64](#_Toc168507989)

[3. Дайте определения редукции лямбда термов. Опишите стратегии редукции лямбда термов. 65](#_Toc168507990)

[4. Дайте определения свободных и связанных переменных в лямбда термах. 68](#_Toc168507991)

[5. Дайте понятия подстановки и преобразования. Сформулируйте понятия эквивалентности. 71](#_Toc168507992)

[6. Сформулируйте теорему Черча-Россера и докажите два следствия из нее. 75](#_Toc168507993)

[7. Сформулируйте и докажите лемму о комбинаторах I, K, S 76](#_Toc168507994)

[8. Докажите, что любой терм представим в виде комбинаторов S K 78](#_Toc168507995)

[9. Числа Черча. Операция плюс 1 Операция (-1). 81](#_Toc168507996)

[10. Кортежи и Каррирование в лямбда исчилении. Объясните, в чем суть каррирования. 82](#_Toc168507997)

[11. Комбинатор неподвижной точки. Приведите два примера, покажите, что они комбинаторы неподвижной точки. 82](#_Toc168507998)

[12. Арифметические операции над числами Черча + \* ^ 84](#_Toc168507999)

[13. Let выражения 86](#_Toc168508000)

[14. Булевы константы и оператор if. Реализация булевых операций. 87](#_Toc168508001)

[15. Рекурсивные функции (на примере любой функции). 89](#_Toc168508002)

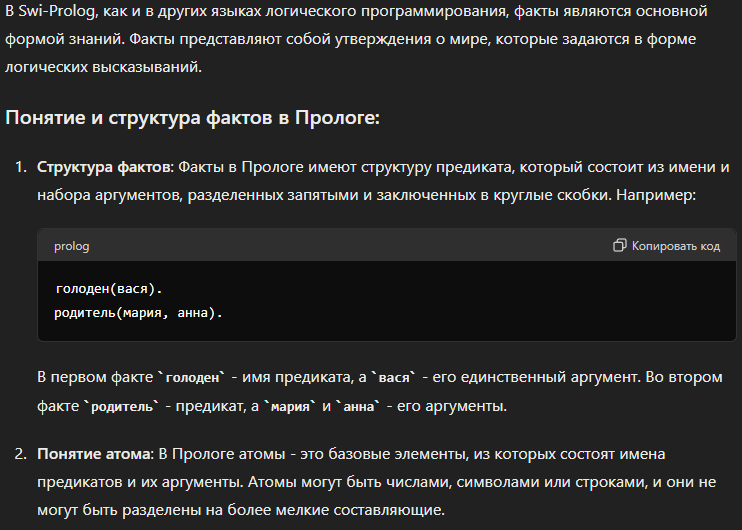
[16. Полнота лямбда исчисления по Тьюрингу 90](#_Toc168508003)

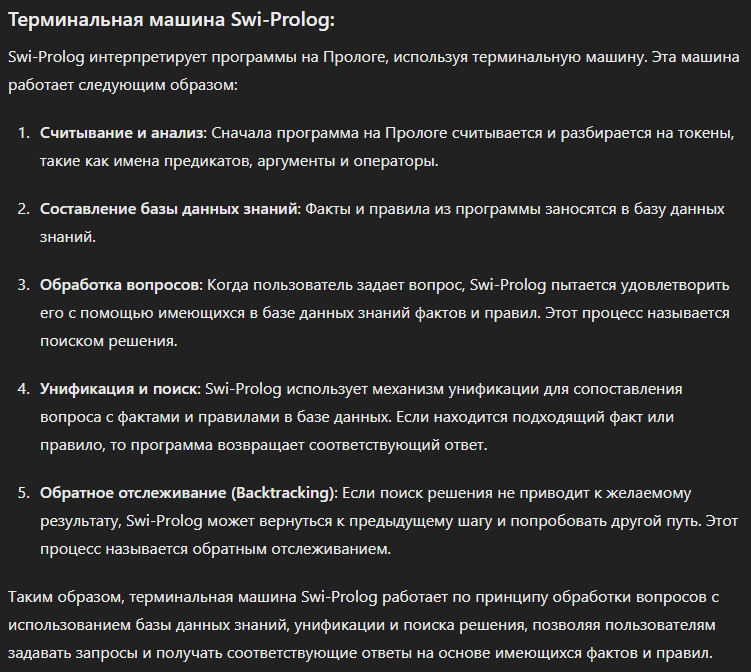
[17. Реализация списков Черча, head, tail. 91](#_Toc168508004)

[18. Покажите набор функций преобразования списков Чёрча. 92](#_Toc168508005)

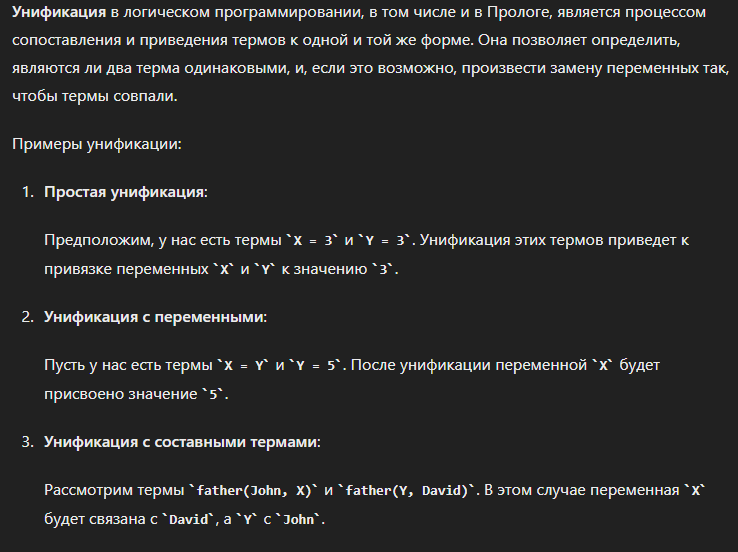
# **Prolog**

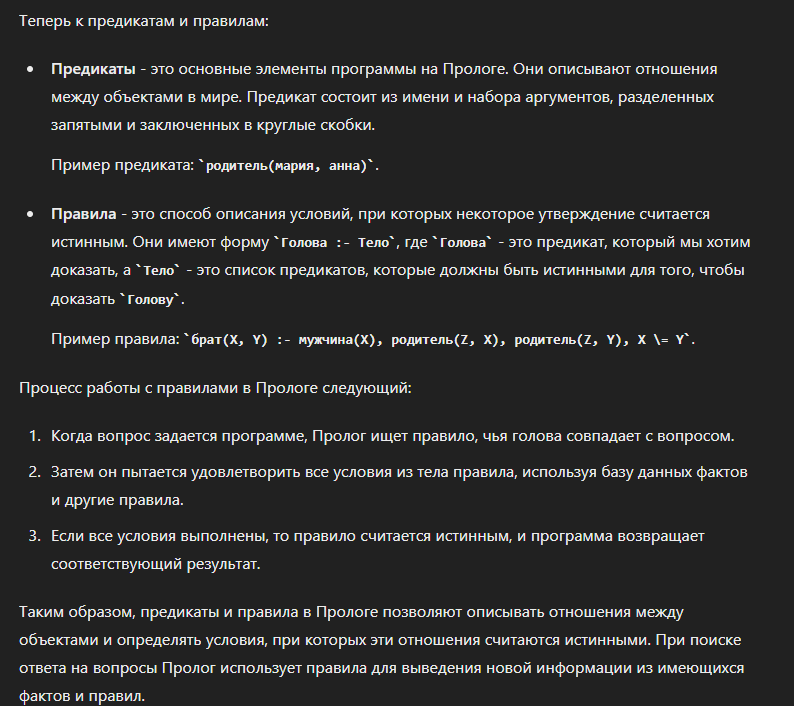
## 1. Опишите понятие и структуру фактов в языке Пролог. Расскройте основные возможные типы, опишите понятие атом. Расскажите принцип работы терминальной машины Swi-Prolog, объясните каким образом происходит обработка вопросов.



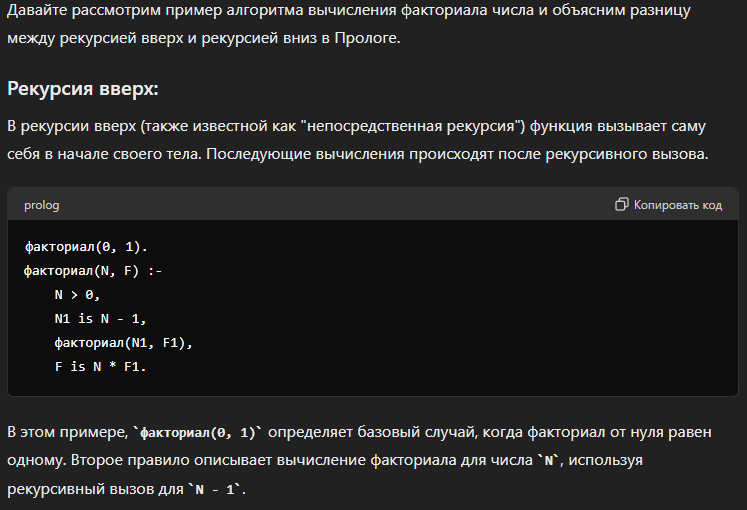


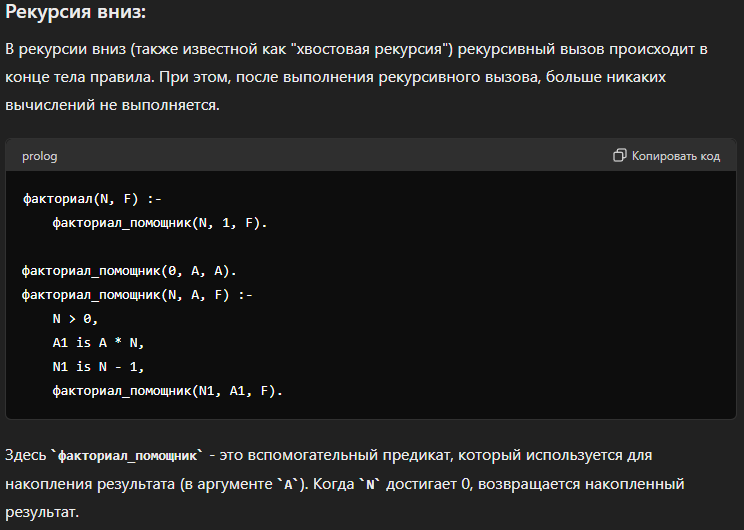
## 2. Опишите смысл термина унификация, приведите показательные примеры. Объясните, как задаются предикаты, что такое правила и каким образом происходит работа с ними.

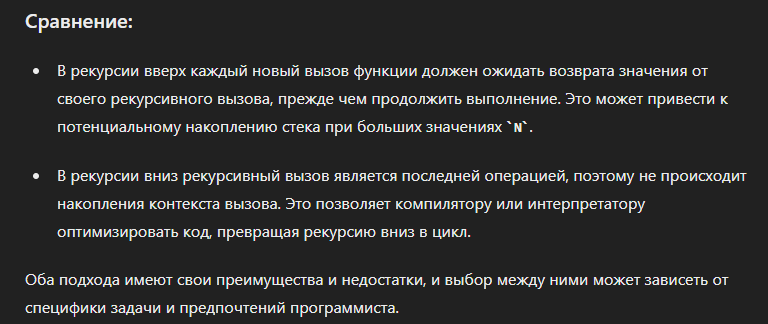




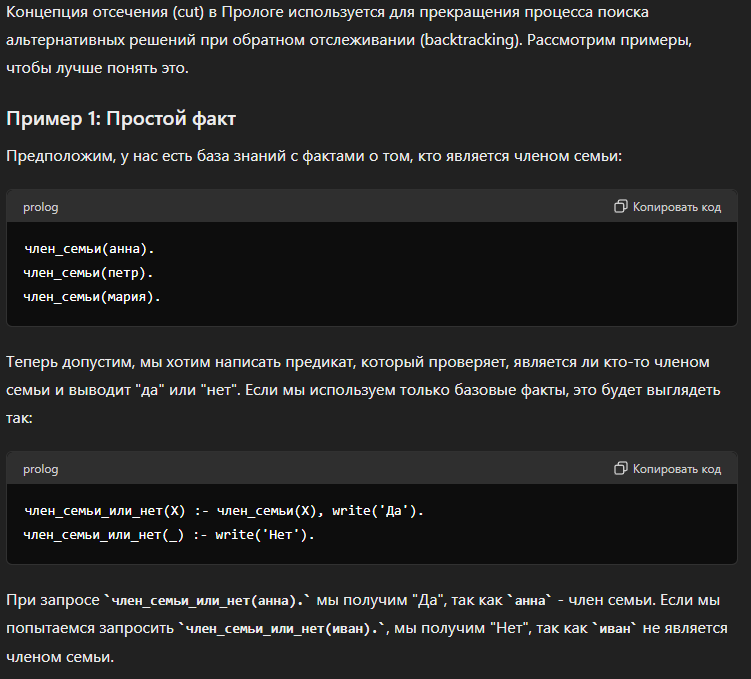
## 3. На примере числовых алгоритмов объясните смысл рекурсии вверх и рекурсии вниз в прологе.

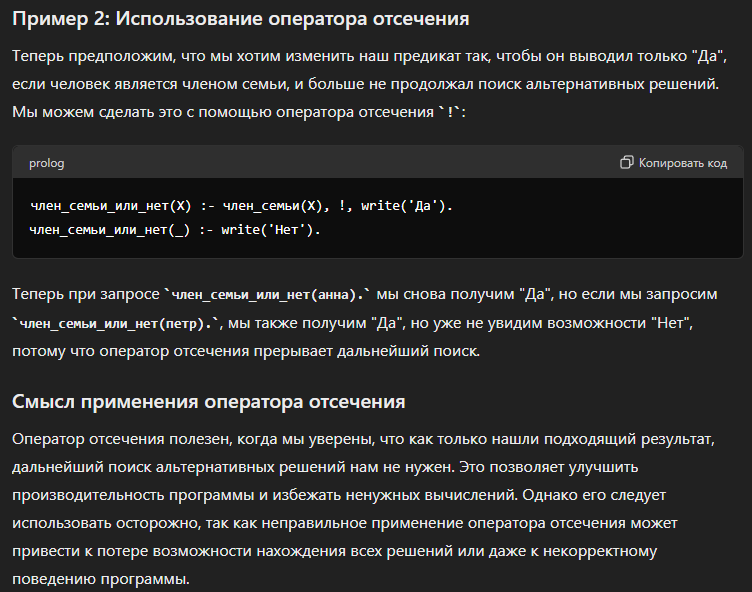






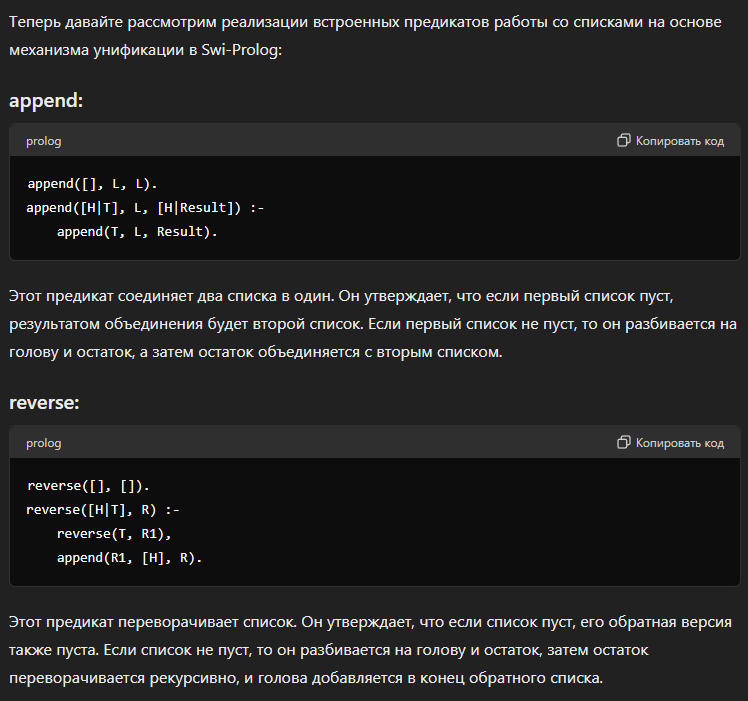
## 4. Раскройте на примерах понятие backtracking, оператор отсечения и смысл его применения.

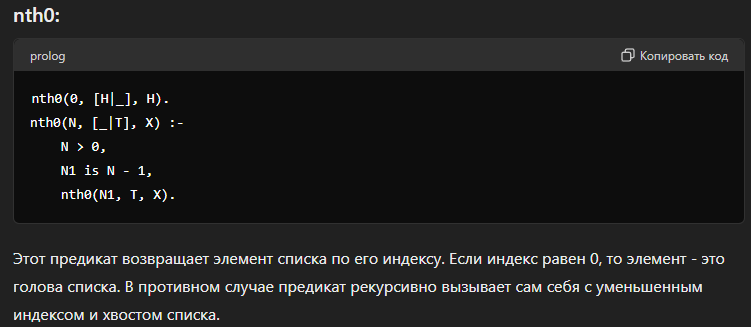




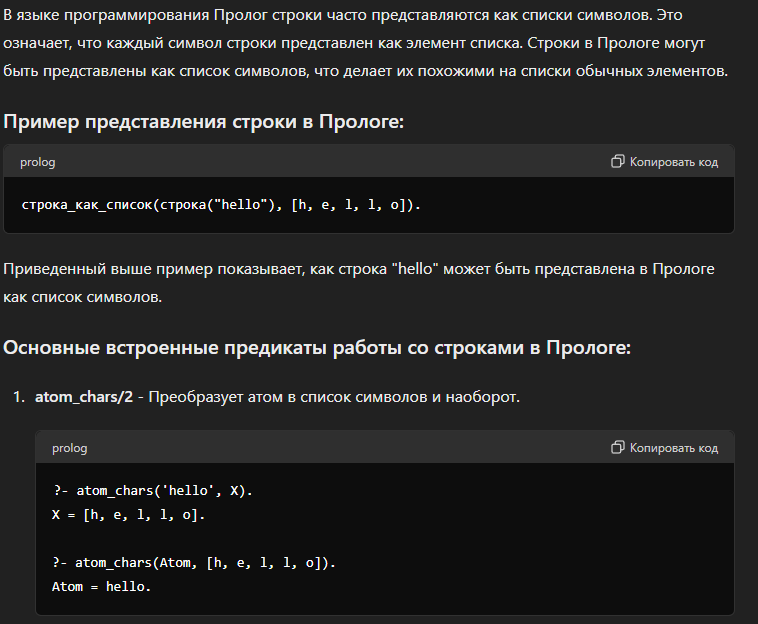
## 5. Опишите принцип работы со списками Черча в Swi-prolog. Покажите реализации встроенных предикатов работы со списками на основе механизма унификации(append, reverse nth0).

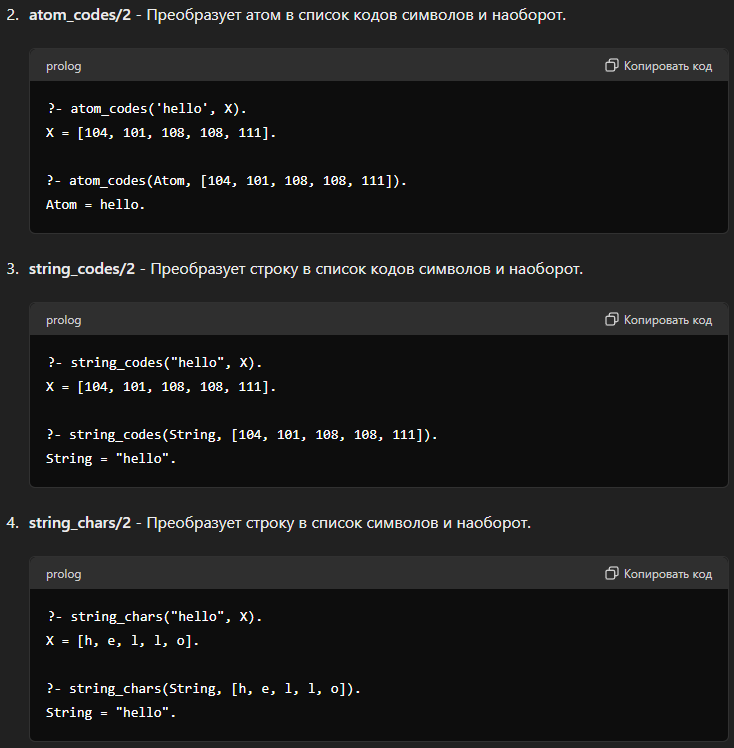


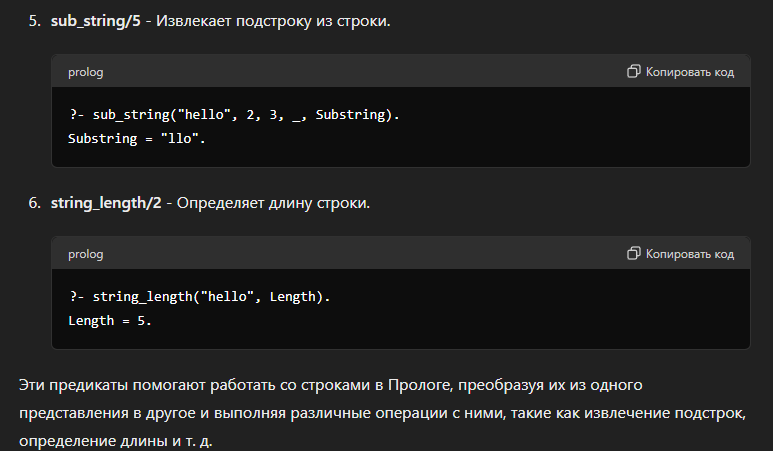




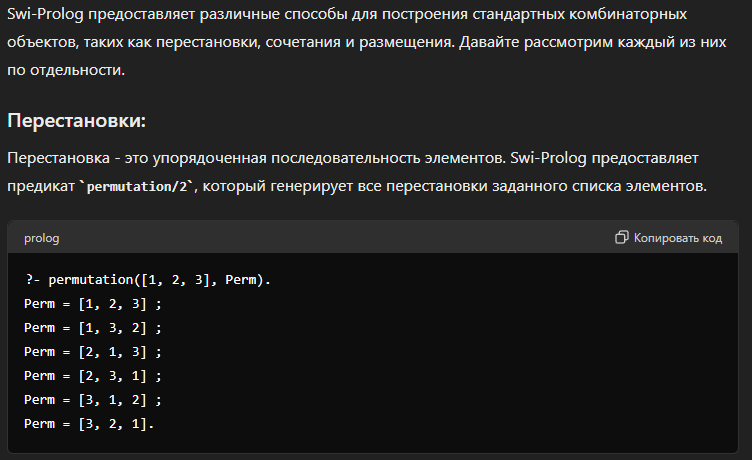
## 6. Объясните принцип работы со строками. Покажите на примерах основные встроенные предикаты работы со строками.

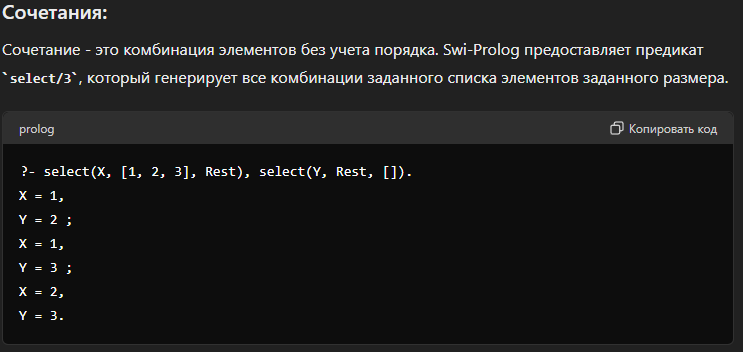


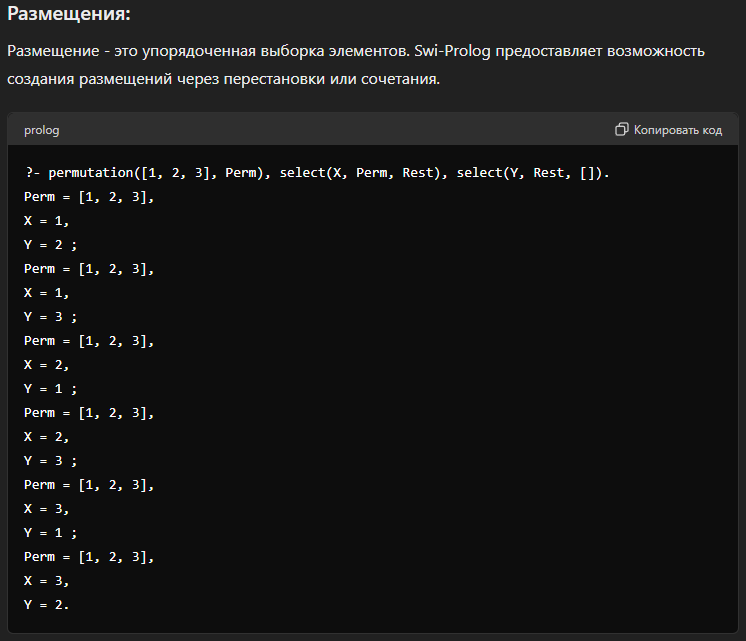


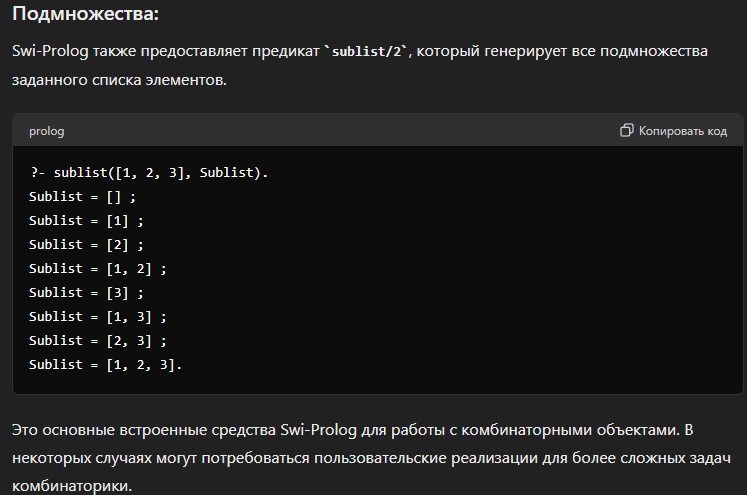


## 7. Покажите, каким образом происходит построение стандартных комбинаторных объектов средствами Swi-prolog.

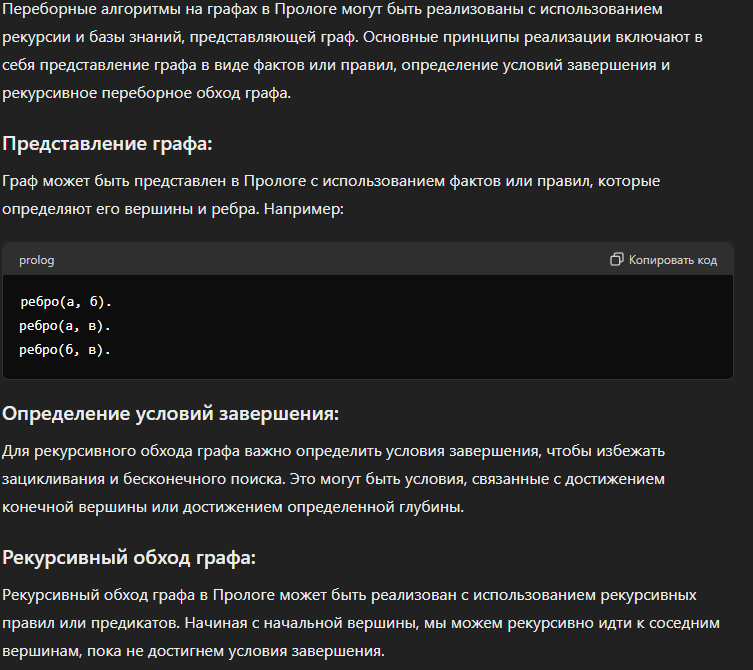


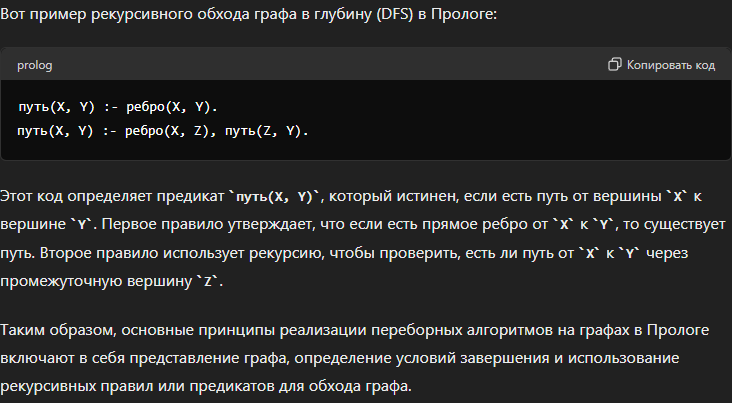




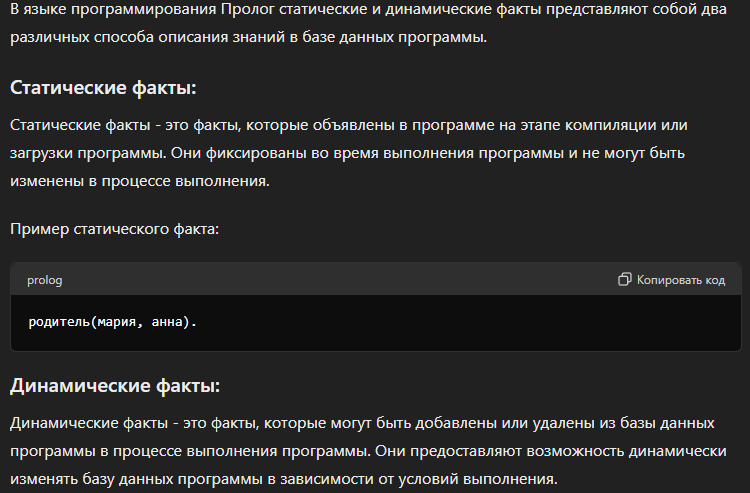


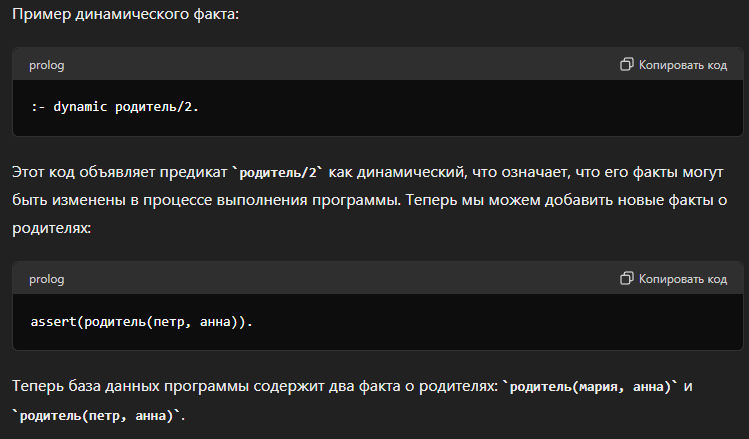
## 8. Покажите основные принципы реализации переборных алгоритмов на графах средствами пролога.

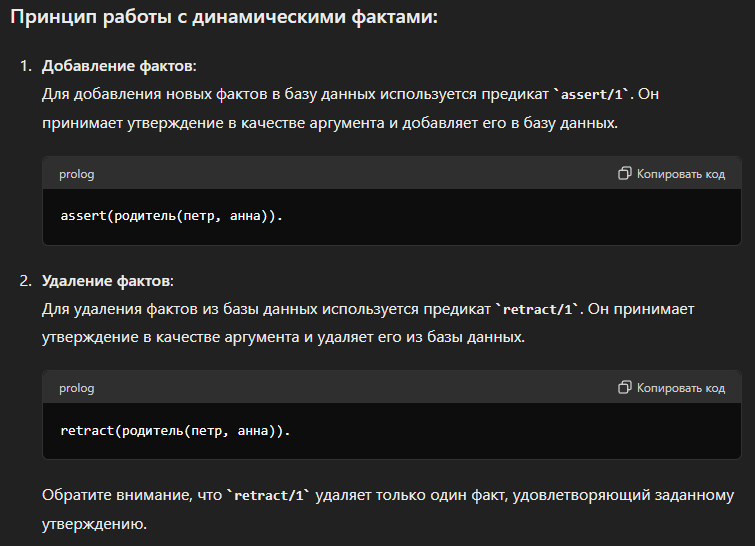


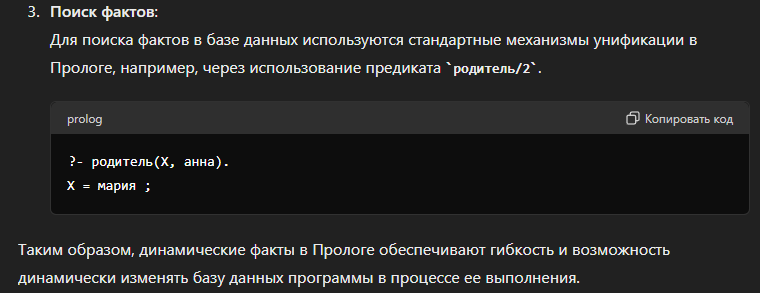


## 9. Раскройте понятия статические и динамические факты. Поясните на примерах принцип работы с динамическими фактами.

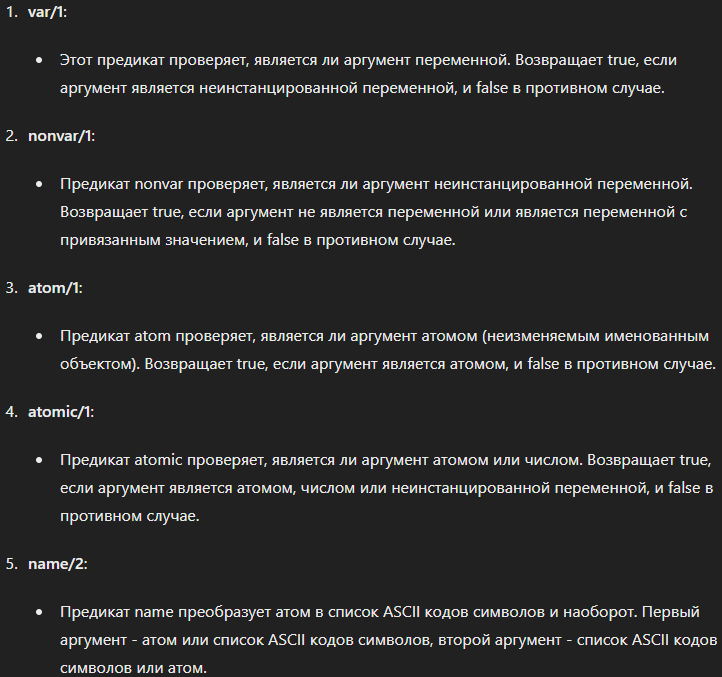


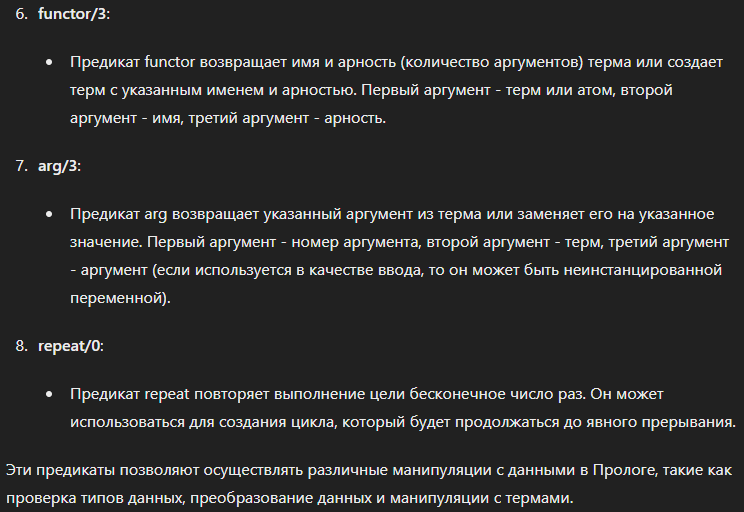






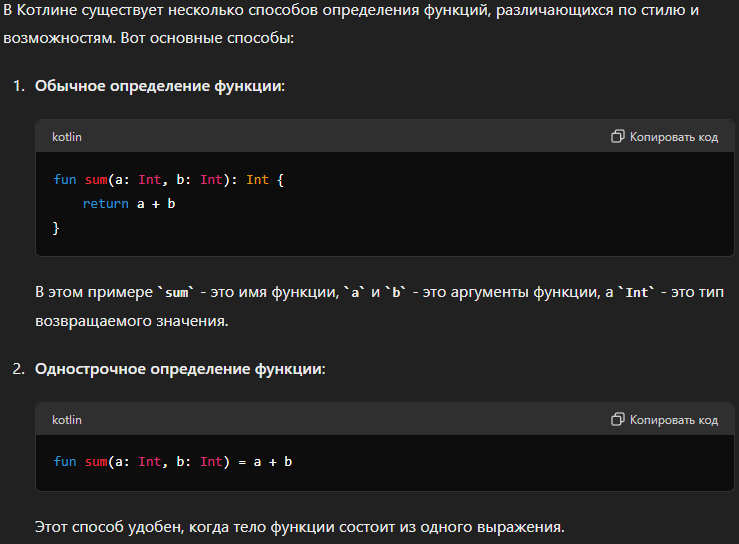
## 10. Объясните принцип работы предикатов var, nonvar, atom, atomic, name, functor, arg, repeat.

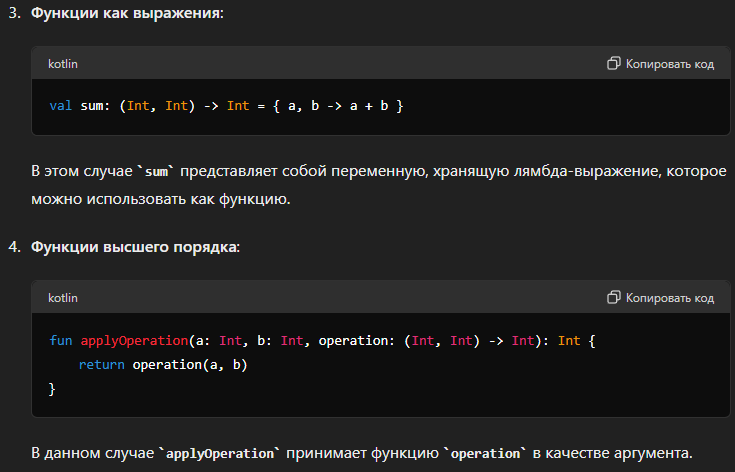


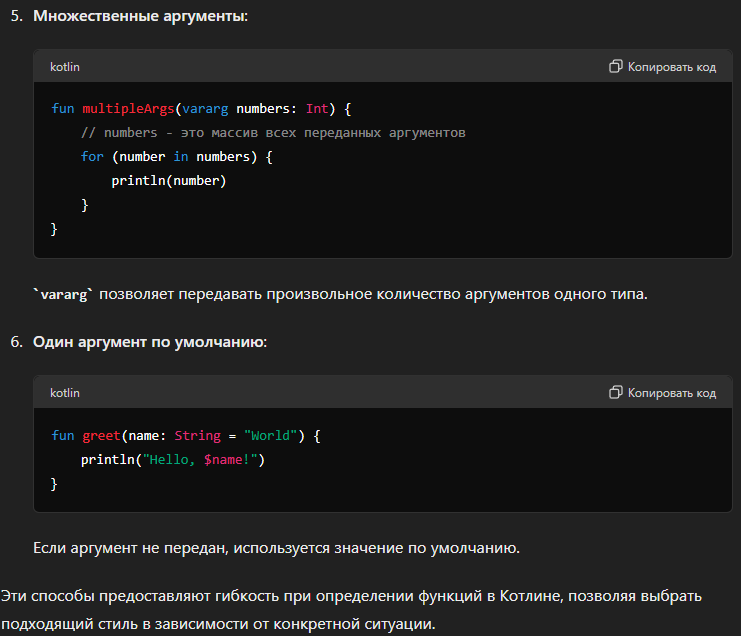


# **Котлин**

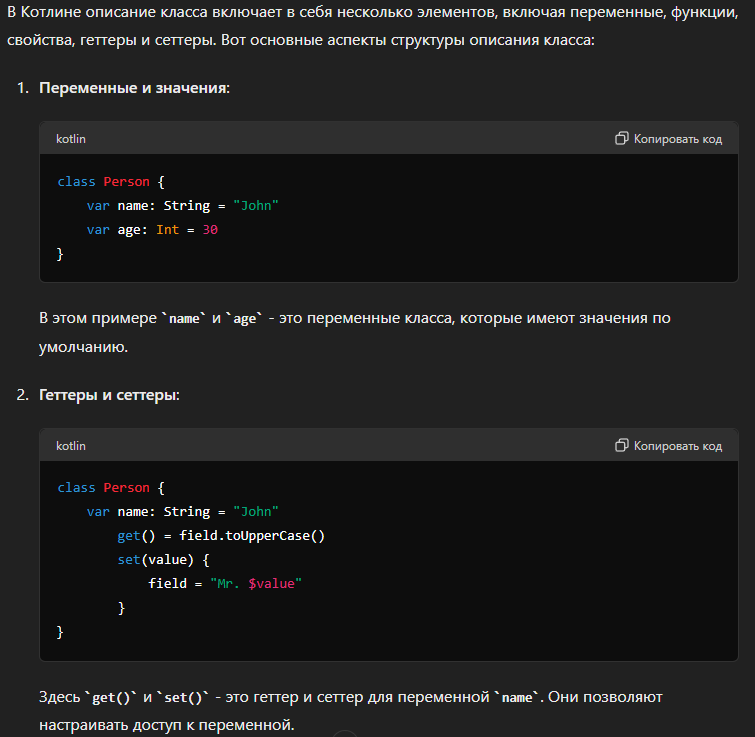
## 1. Способы написания функций в котлине, разные виды тела функции. Функциональные аргументы и значения. Множество аргументов. Один аргумент.

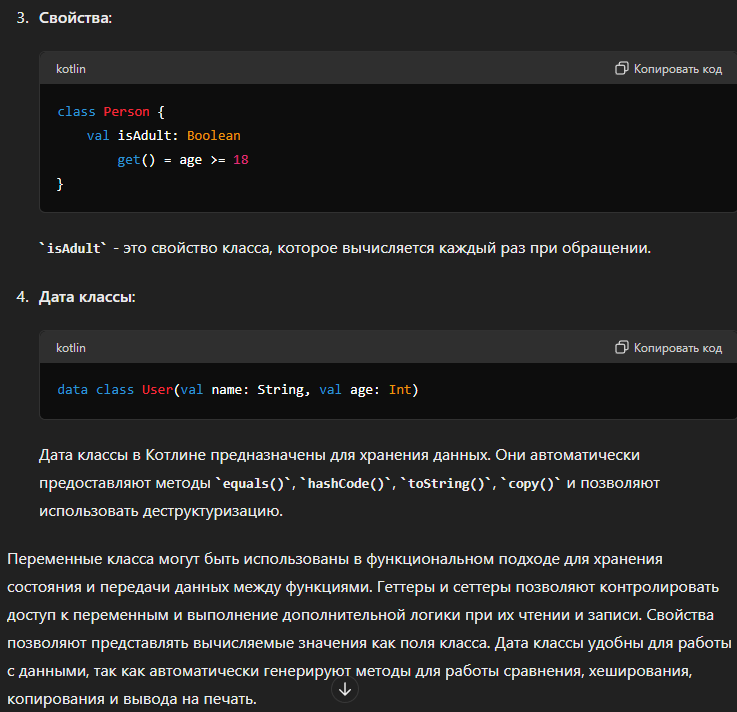




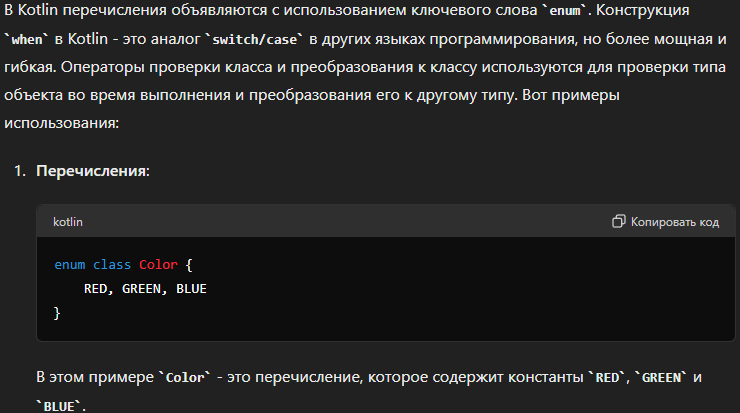


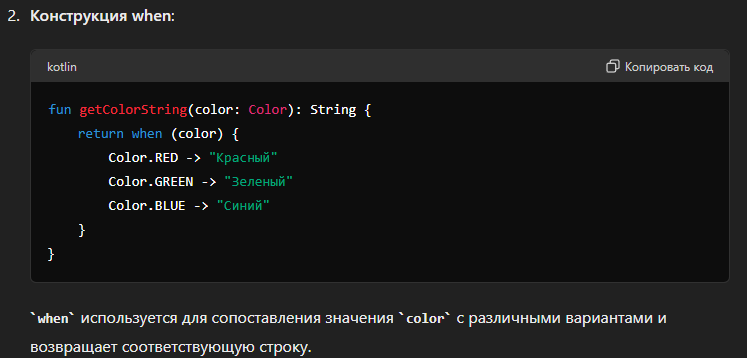
## 2. Структура описания класса в котлине. Переменные и значения. Разница и применение в функциональном подходе. Геттеры, сеттеры, свойства. Дата классы.

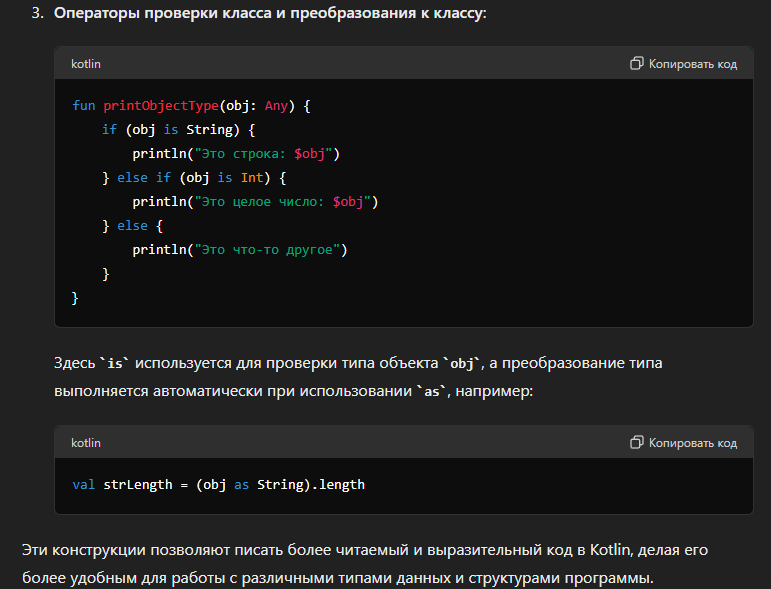




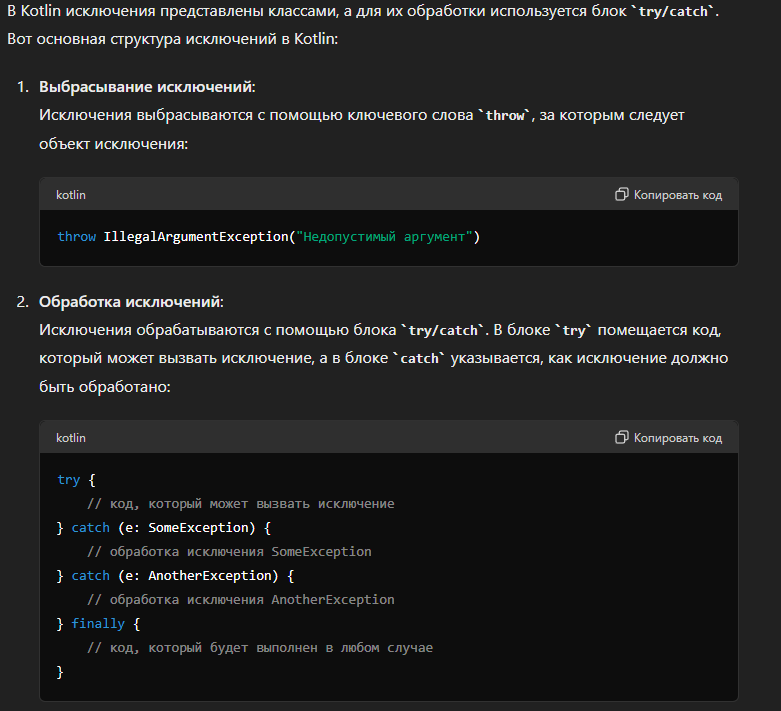
## 3. Перечисления и конструкция when. Операторы проверки класса и преобразования к классу.

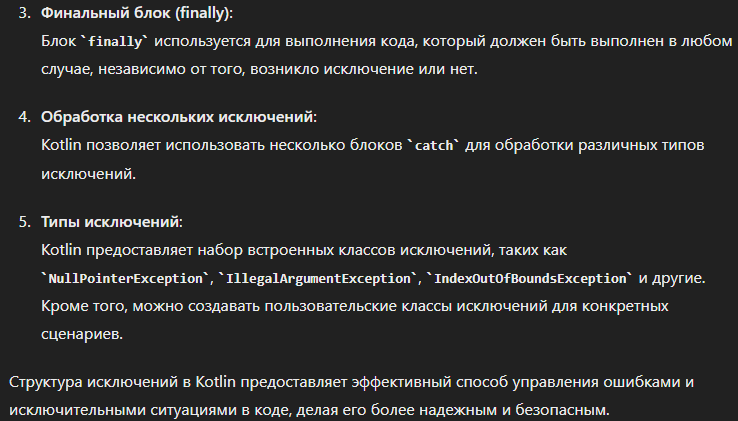




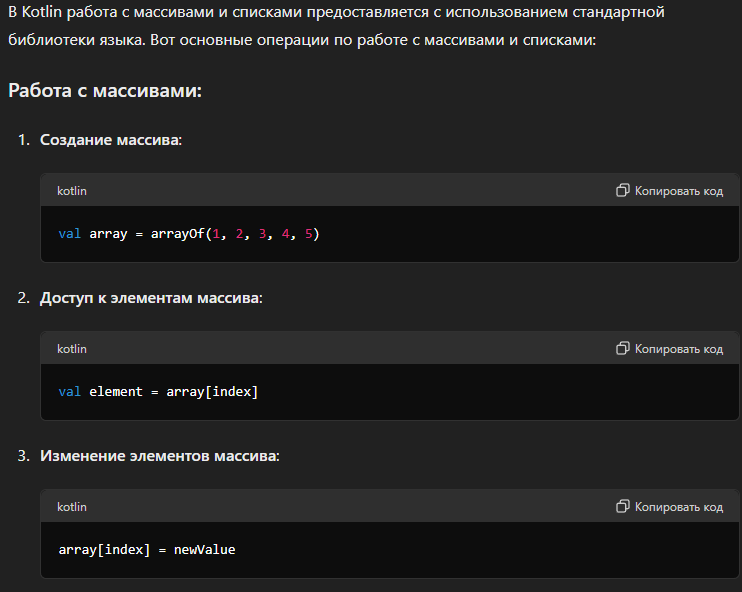


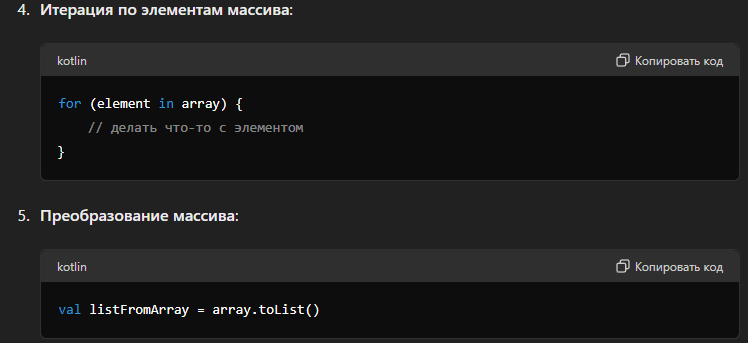
## 4. Исключения в Котлин. Структура исключений.

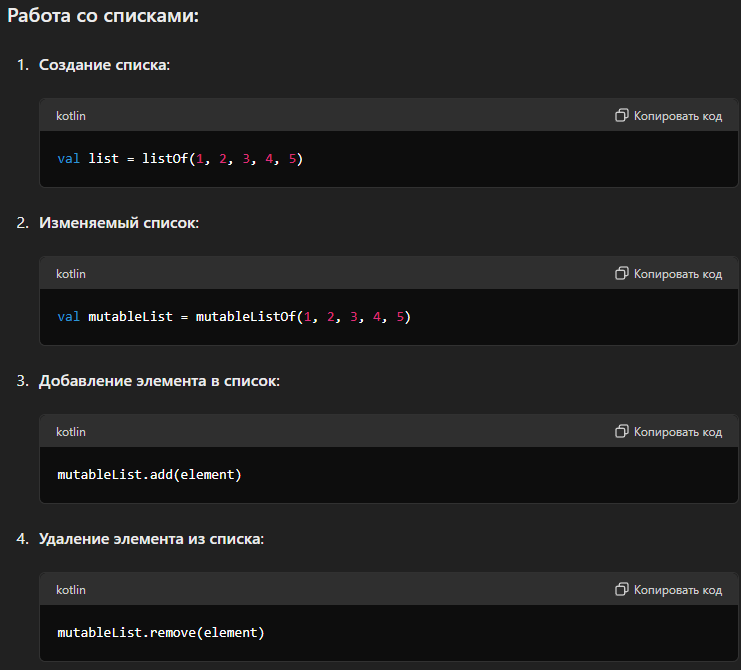


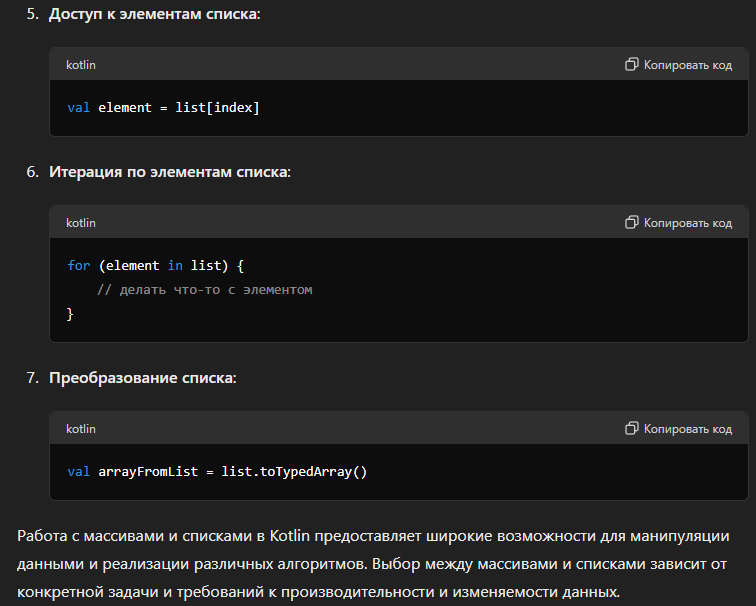


## 5. Работа с массивами в котлин. Работа со списками в котлин.

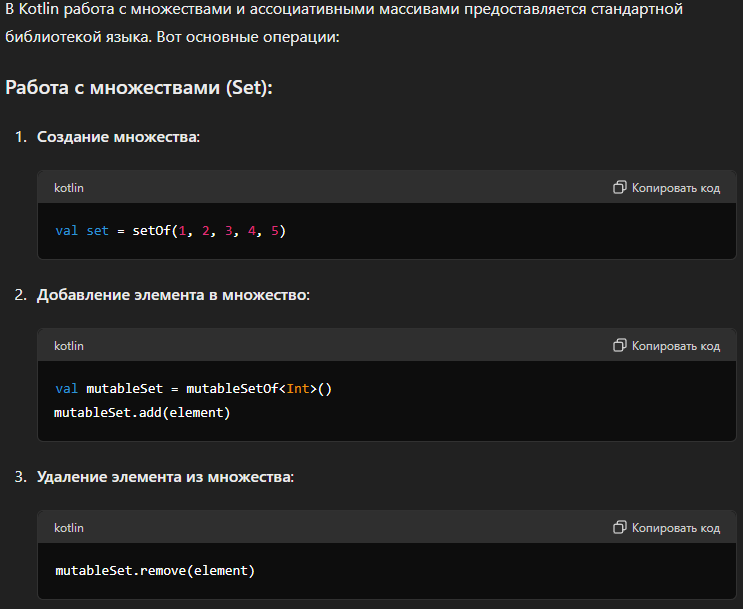


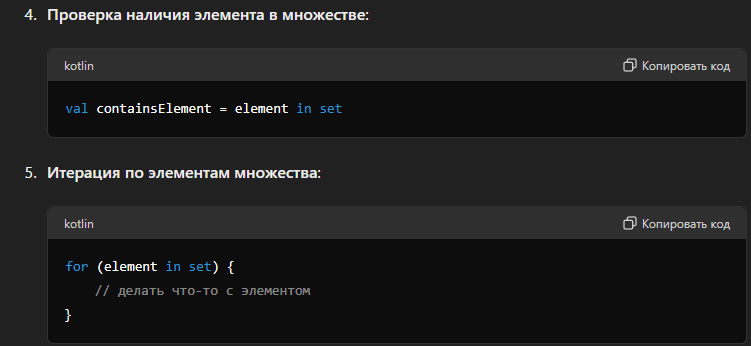


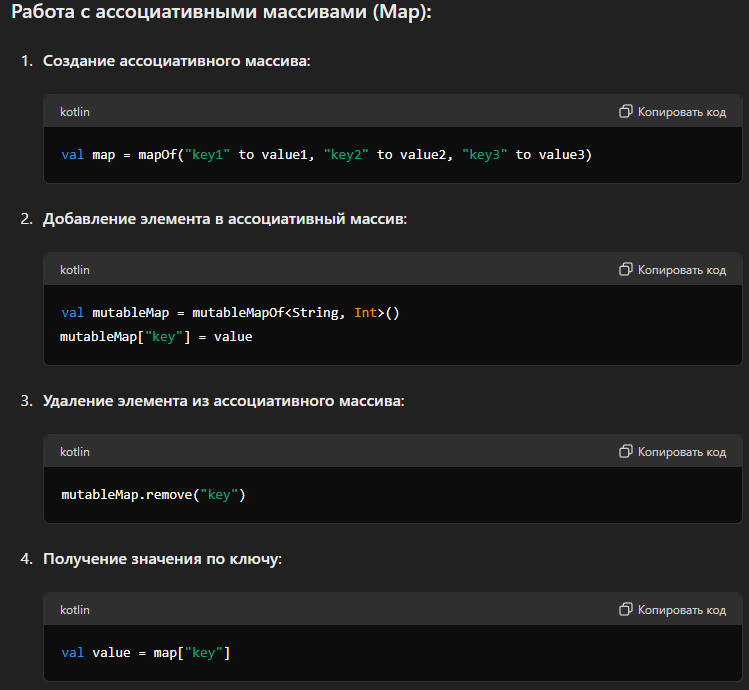


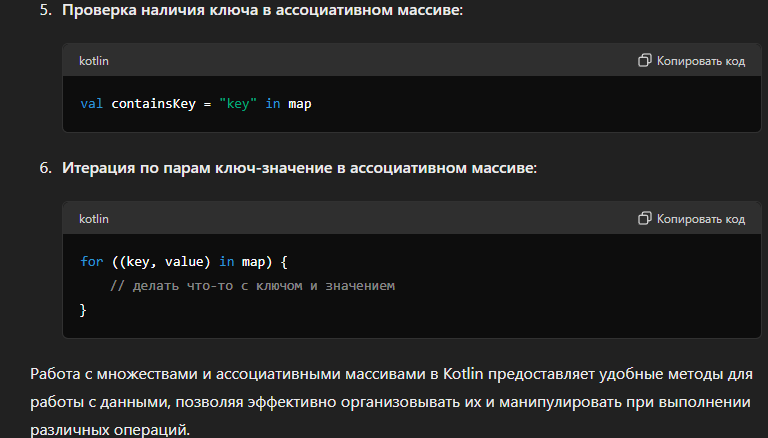


## 6. Работа с множествами в котлин. Работа с ассоциативными массивами в котлин

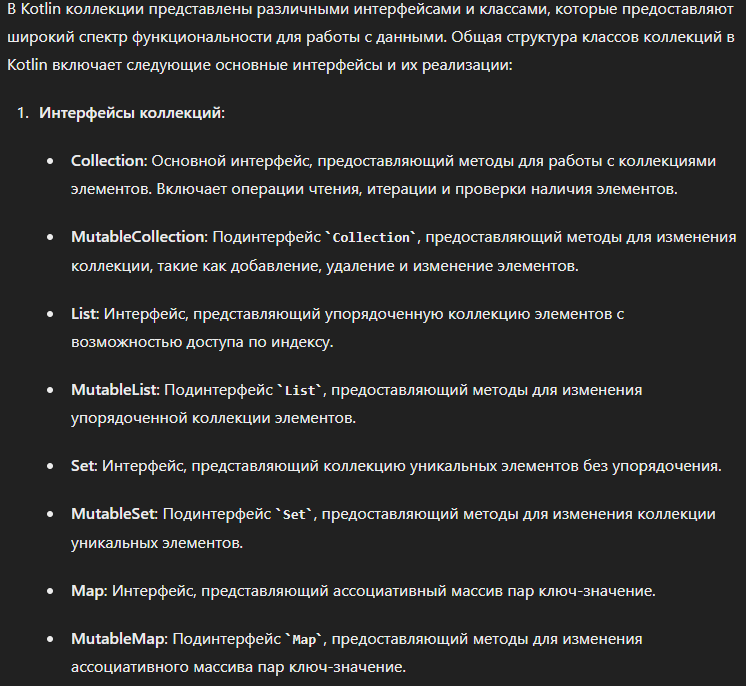


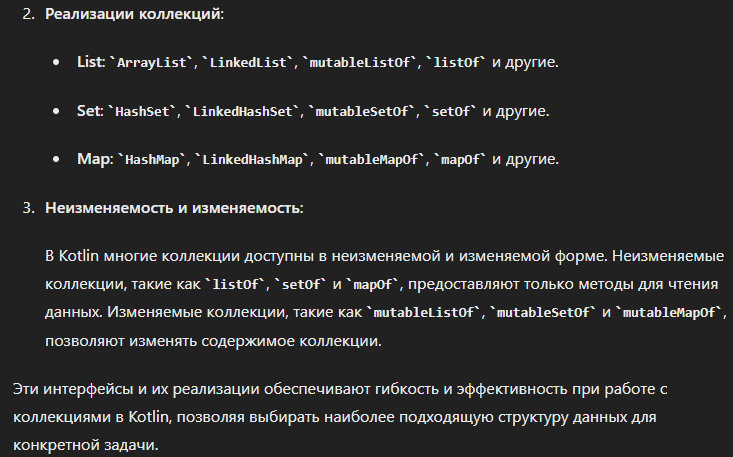




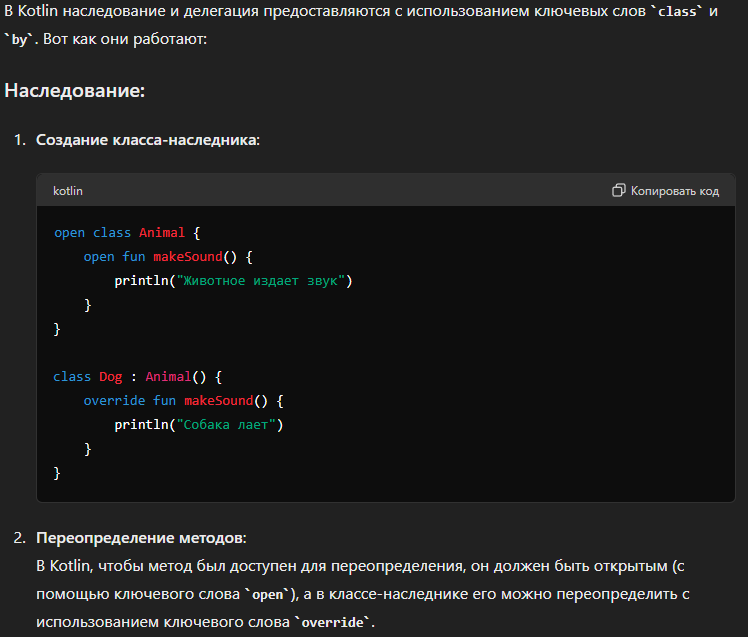


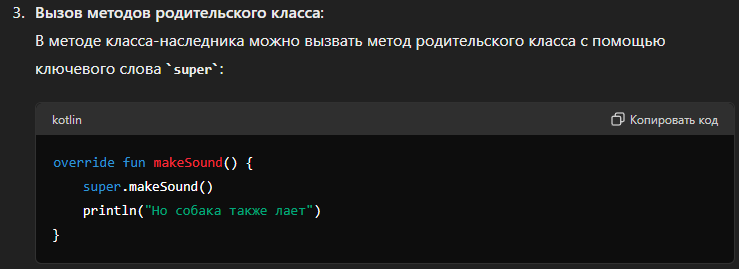
## 7. Общая структура классов коллекций в котлин.

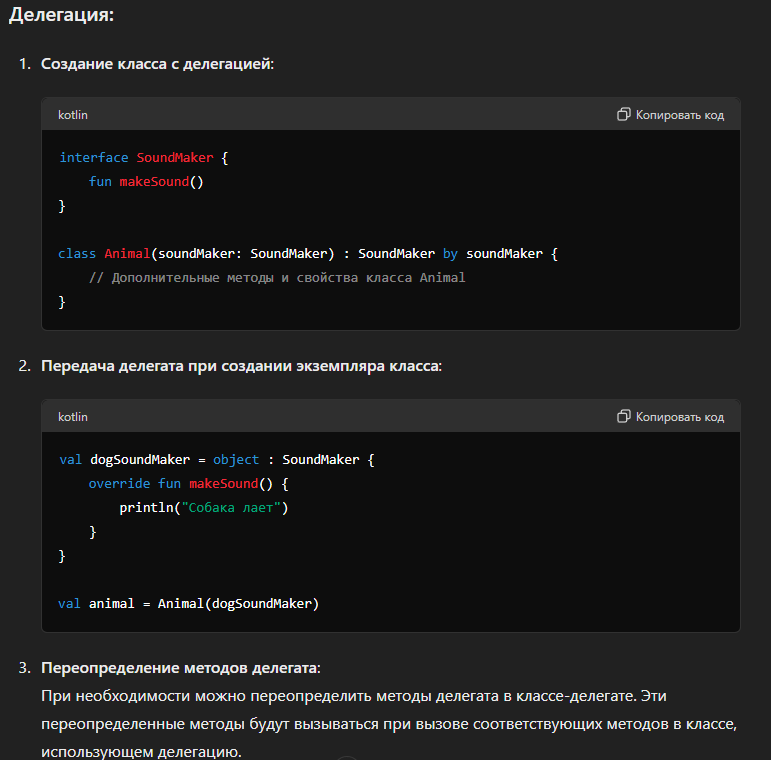


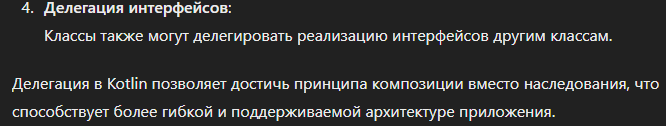


## 8. Наследование в котлин. Делегация в котлин.

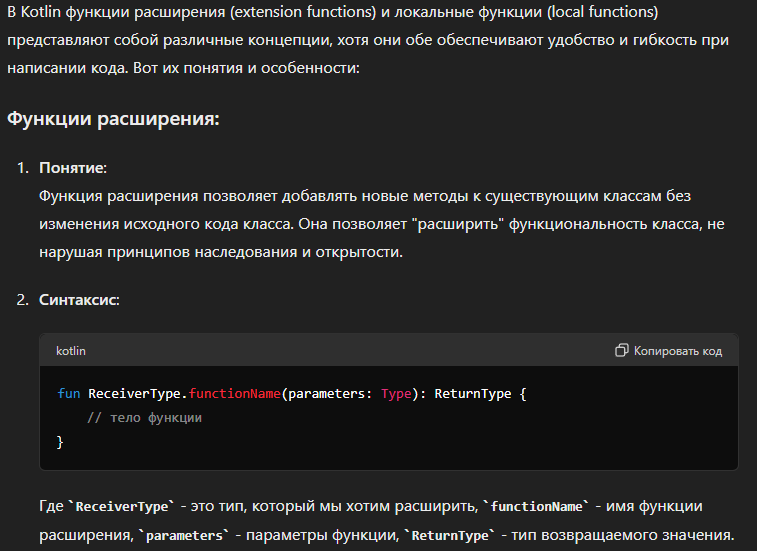


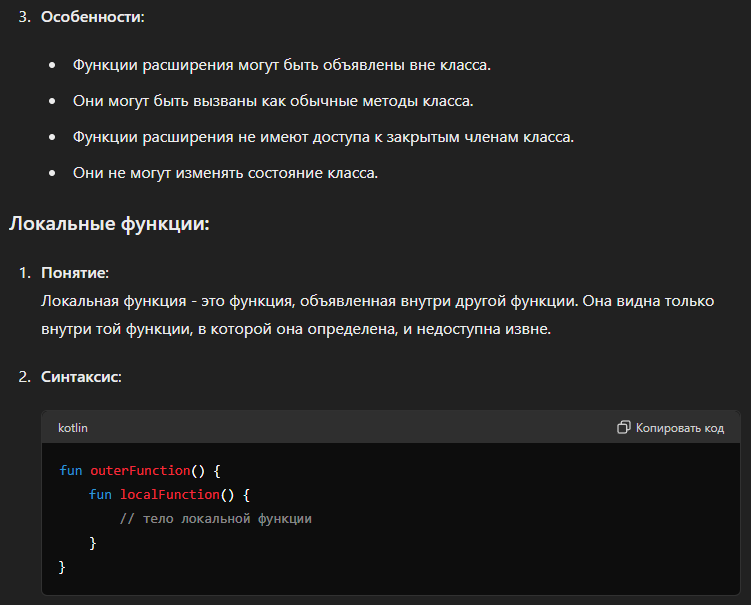


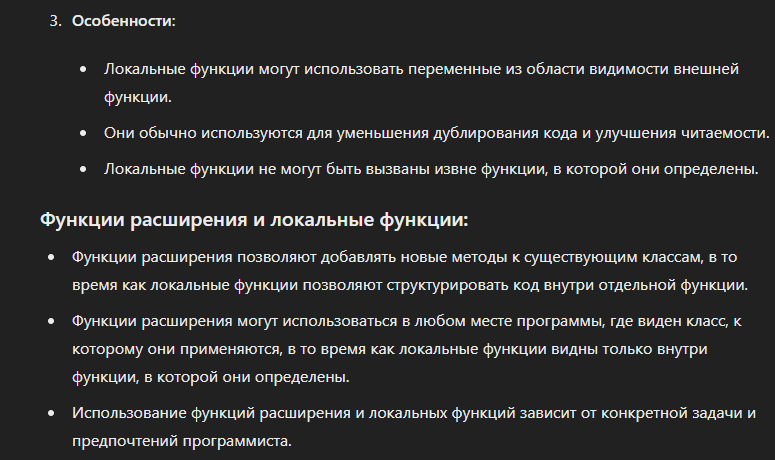




## 9. Понятие функции расширения в котлин. Понятие и особенности локальных функций в котлине. Функции расширения и локальные функции.







## 10. Регулярные выражения в котлин.

В Kotlin работа с регулярными выражениями осуществляется с использованием класса Regex из стандартной библиотеки языка. Вот основные операции:

**Создание регулярного выражения**:

kotlin

Копировать код

val regex = Regex(pattern)

Где pattern - это строка с регулярным выражением.

**Поиск совпадений**:

kotlin

Копировать код

val matches: Sequence<MatchResult> = regex.findAll(input)

Метод findAll ищет все совпадения с регулярным выражением в строке input и возвращает последовательность объектов MatchResult, содержащих информацию о каждом совпадении.

**Проверка наличия совпадений**:

kotlin

Копировать код

val hasMatch: Boolean = regex.containsMatchIn(input)

Метод containsMatchIn возвращает true, если входная строка содержит хотя бы одно совпадение с регулярным выражением, и false в противном случае.

**Замена совпадений**:

kotlin

Копировать код

val result: String = regex.replace(input, replacement)

Метод replace заменяет все совпадения с регулярным выражением в строке input на строку replacement и возвращает полученный результат.

**Извлечение групп**:

kotlin

Копировать код

for (match in matches) {

val groupValue: String = match.groups[1]?.value ?: ""

}

Для каждого совпадения можно получить доступ к значениям групп с помощью свойства groups, которое содержит список групп совпадений. Группы индексируются с 1, где 0 - это весь текст совпадения.

**Использование строкового шаблона**:

kotlin

Копировать код

val regex = """\d{3}-\d{2}-\d{4}""".toRegex()

Kotlin поддерживает использование тройных кавычек для создания строковых шаблонов без необходимости экранирования символов.

Регулярные выражения в Kotlin предоставляют мощные инструменты для работы с текстовыми данными, позволяя осуществлять поиск, замену и анализ текста с помощью гибких шаблонов.

## 11. Интерфейсы и полиморфизм в котлин.

ChatGPT

В Kotlin интерфейсы и полиморфизм играют ключевые роли в создании гибких и расширяемых программ. Давайте рассмотрим их основные концепции:

**Интерфейсы:**

**Определение интерфейса**:

kotlin

Копировать код

interface Drawable {

fun draw()

}

**Реализация интерфейса**:

kotlin

Копировать код

class Circle : Drawable {

override fun draw() {

println("Рисуем круг")

}

}

**Множественная реализация интерфейсов**:

kotlin

Копировать код

interface Shape {

fun calculateArea()

}

class Circle : Drawable, Shape {

override fun draw() {

println("Рисуем круг")

}

override fun calculateArea() {

println("Вычисляем площадь круга")

}

}

**Полиморфизм:**

**Использование полиморфизма**: Полиморфизм позволяет использовать объекты различных классов через их общий интерфейс или родительский класс. Например:

kotlin

Копировать код

fun drawShape(shape: Drawable) {

shape.draw()

}

**Пример использования**:

kotlin

Копировать код

val circle = Circle()

val square = Square()

drawShape(circle) // Рисуем круг

drawShape(square) // Рисуем квадрат

**Переопределение методов в дочерних классах**: В дочерних классах можно переопределять методы, определенные в родительском классе или интерфейсе, чтобы предоставить свою собственную реализацию.

Полиморфизм позволяет обрабатывать различные типы объектов с помощью общего интерфейса или родительского класса, что делает код более гибким и расширяемым. Интерфейсы, в свою очередь, позволяют определить контракты, которые должны быть реализованы в классах, обеспечивая более чистый и модульный код.

## 12. Общий обзор наследования, абстрактных классов, интерфейсов и их реализации в котлин. Набор модификаторов. Сравнение с java

В Kotlin, как и в Java, наследование, абстрактные классы и интерфейсы предоставляют средства для создания гибких и масштабируемых программ. Давайте рассмотрим каждый из них, а также набор модификаторов доступа:

**Наследование:**

**Определение:** Наследование позволяет классу наследовать свойства и методы другого класса, называемого родительским классом.

**Пример:**

kotlin

Копировать код

open class Animal {

open fun makeSound() {

println("Animal makes a sound")

}

}

class Dog : Animal() {

override fun makeSound() {

println("Dog barks")

}

}

**Абстрактные классы:**

**Определение:** Абстрактный класс - это класс, который содержит абстрактные (несколько реализованных) методы, которые должны быть реализованы в его подклассах.

**Пример:**

kotlin

Копировать код

abstract class Shape {

abstract fun draw()

}

class Circle : Shape() {

override fun draw() {

println("Drawing circle")

}

}

**Интерфейсы:**

**Определение:** Интерфейс - это контракт, который определяет набор методов, которые должны быть реализованы классом.

**Пример:**

kotlin

Копировать код

interface Drawable {

fun draw()

}

class Circle : Drawable {

override fun draw() {

println("Drawing circle")

}

}

**Набор модификаторов:**

public: Элемент доступен везде

internal: Элемент доступен в пределах модуля

protected: Элемент доступен в подклассах

private: Элемент доступен только в текущем классе или файле

**Сравнение с Java:**

В Kotlin классы по умолчанию являются финальными (нельзя наследовать без явного указания open), в отличие от Java, где классы открыты для наследования.

В Kotlin нет явного ключевого слова abstract при объявлении абстрактных методов и классов. Используется непосредственно модификатор abstract.

В Kotlin интерфейсы могут содержать реализацию методов по умолчанию, что не возможно в Java до версии 8.

Кotlin стремится предоставить более компактный и выразительный синтаксис, сохраняя при этом семантику Java. Однако, в Kotlin есть некоторые особенности, которые делают его более безопасным и удобным в использовании, такие как отсутствие неявного нулевого указателя и возможность использования расширений и функций высшего порядка.

## 13. Модификаторы доступа в котлин. Запечатанные классы.

ChatGPT

В Kotlin модификаторы доступа определяют уровень доступа к классам, свойствам, методам и другим элементам программы. Запечатанные классы представляют особый вид классов, которые ограничивают наследование от них. Давайте рассмотрим их подробнее:

**Модификаторы доступа:**

**public**:

Элемент доступен везде, как внутри текущего модуля, так и за его пределами.

Это модификатор доступа по умолчанию в Kotlin, если не указан другой модификатор.

**internal**:

Элемент доступен только в пределах текущего модуля.

Это позволяет создавать абстрактные модули и ограничивать доступ к их содержимому извне.

**protected**:

Элемент доступен внутри класса и его подклассов (наследников).

В Kotlin доступ к защищенным элементам в пределах одного модуля аналогичен доступу к элементам с модификатором internal.

**private**:

Элемент доступен только внутри текущего класса или файла.

Данный модификатор наиболее ограничивающий и обеспечивает инкапсуляцию данных.

**Запечатанные классы (Sealed classes):**

Запечатанные классы - это классы, которые могут иметь только ограниченное количество подклассов, определенных в том же файле, где и основной класс.

Они полезны, когда тип может быть одним из ограниченного набора вариантов, и необходимо предотвратить создание новых подклассов извне.

Запечатанные классы часто используются вместе с оператором when для обработки всех возможных вариантов типа.

Пример использования запечатанных классов:

kotlin

Копировать код

sealed class Result

class Success(val data: String) : Result()

class Error(val message: String) : Result()

fun handleResult(result: Result) {

when (result) {

is Success -> println("Success: ${result.data}")

is Error -> println("Error: ${result.message}")

}

}

Вот общий обзор модификаторов доступа и запечатанных классов в Kotlin. Эти механизмы обеспечивают гибкий и безопасный способ управления доступом и организации иерархии классов.

## 14. Способы задания конструкторов в котлине.

В Kotlin есть несколько способов определения конструкторов в классах, включая основные конструкторы, вспомогательные конструкторы и инициализацию свойств при объявлении. Давайте рассмотрим каждый из них:

1. Основной конструктор (Primary constructor):

Основной конструктор объявляется в заголовке класса и может содержать параметры. Он может иметь вид сразу после имени класса, либо находиться в заголовке после имени класса, а его параметры могут быть использованы внутри класса для инициализации свойств.

Пример:

kotlin

Копировать код

class Person(firstName: String, lastName: String) {

// Основной конструктор

init {

println("Имя: $firstName, Фамилия: $lastName")

}

}

2. Вспомогательные конструкторы (Secondary constructors):

Вспомогательные конструкторы могут быть определены с использованием ключевого слова constructor. Они должны вызывать другой конструктор того же класса (основной или другой вспомогательный) с помощью this().

Пример:

kotlin

Копировать код

class Person {

constructor(firstName: String, lastName: String) {

println("Имя: $firstName, Фамилия: $lastName")

}

}

3. Инициализация свойств при объявлении (Initializing properties in class declaration):

Свойства класса могут быть инициализированы прямо при объявлении, что удобно и позволяет избежать объявления вторичных конструкторов.

Пример:

kotlin

Копировать код

class Person(val firstName: String, val lastName: String)

4. Применение модификаторов видимости к конструкторам:

Конструкторы также могут иметь модификаторы доступа, такие как private, protected, internal. Эти модификаторы позволяют ограничивать доступ к конструкторам.

Пример:

kotlin

Копировать код

class Person private constructor(firstName: String, lastName: String)

5. Конструкторы с аннотациями (Constructors with annotations):

Конструкторы также могут иметь аннотации, что позволяет использовать их в различных сценариях, таких как внедрение зависимостей и анализ аргументов.

Пример:

kotlin

Копировать код

class Person @Inject constructor(firstName: String, lastName: String)

6. Вызов конструктора родительского класса (Calling superclass constructor):

При наследовании от другого класса необходимо вызвать конструктор родительского класса с помощью ключевого слова super().

Пример:

kotlin

Копировать код

class Student(firstName: String, lastName: String, studentId: Int) : Person(firstName, lastName) {

// ...

}

В Kotlin есть множество способов определения и использования конструкторов, что обеспечивает гибкость и удобство при создании классов и их экземпляров.

## 15. Структура классов в котлин. Сравнение объектов в котлин.

В Kotlin структура классов похожа на Java, но с некоторыми отличиями и дополнениями. Давайте рассмотрим основные элементы структуры классов и сравнение объектов:

**Структура классов в Kotlin:**

**Объявление класса**:

kotlin

Копировать код

class MyClass {

// тело класса

}

**Свойства класса**:

kotlin

Копировать код

class Person {

var name: String = ""

var age: Int = 0

}

**Методы класса**:

kotlin

Копировать код

class Calculator {

fun add(a: Int, b: Int): Int {

return a + b

}

}

**Конструкторы**:

kotlin

Копировать код

class Car(make: String, model: String) {

// тело класса

}

**Инициализация блока**:

kotlin

Копировать код

class Person(val name: String, val age: Int) {

init {

println("Имя: $name, Возраст: $age")

}

}

**Сравнение объектов:**

В Kotlin для сравнения объектов используются операторы == и ===:

Оператор == сравнивает содержимое объектов.

Оператор === сравнивает ссылки на объекты (проверяет, указывают ли две переменные на один и тот же объект в памяти).

Примеры:

kotlin

Копировать код

val x = MyClass()

val y = MyClass()

println(x == y) // Сравнение содержимого (true или false)

println(x === y) // Сравнение ссылок (false, потому что это разные объекты)

Если вам нужно сравнить содержимое двух объектов класса, вы можете переопределить метод equals():

kotlin

Копировать код

class Person(val name: String, val age: Int) {

override fun equals(other: Any?): Boolean {

if (this === other) return true

if (other !is Person) return false

return name == other.name && age == other.age

}

}

Это общий обзор структуры классов в Kotlin и способов сравнения объектов. Kotlin предоставляет гибкие средства для создания классов и операций с объектами, что делает разработку приложений более эффективной и удобной.

## 16. Совместное объявление класса и экзмепляра в котлин. Задание фабричных методов в котлин.

В Kotlin нет точного эквивалента совместного объявления класса и экземпляра, как, например, в Java с использованием анонимных классов. Однако, Kotlin предоставляет средства для объявления классов с инициализацией, что делает код более компактным. Рассмотрим несколько способов:

1. Объявление класса с инициализацией:

kotlin

Копировать код

class Person {

val name: String

val age: Int

init {

name = "John"

age = 30

}

}

2. Использование инициализации в конструкторе:

kotlin

Копировать код

class Person(val name: String = "John", val age: Int = 30)

Задание фабричных методов:

В Kotlin фабричные методы могут быть объявлены как статические методы в companion object. Они могут использоваться для создания экземпляров класса с дополнительной логикой или инициализацией.

kotlin

Копировать код

class Person(val name: String, val age: Int) {

companion object {

fun createJohn(): Person {

return Person("John", 30)

}

fun createJane(age: Int): Person {

return Person("Jane", age)

}

}

}

Эти фабричные методы могут быть вызваны без создания экземпляра класса:

kotlin

Копировать код

val john = Person.createJohn()

val jane = Person.createJane(25)

Таким образом, хотя в Kotlin нет точного совместного объявления класса и экземпляра, можно использовать инициализацию в конструкторе или в блоке init, а также определять фабричные методы в companion object для создания экземпляров класса с дополнительной логикой.

## 17. Лямбда выражения. Передача лямбда выражений. Замыкание. Краткие способы записи лямбда выражений.

Лямбда-выражения в Kotlin позволяют создавать анонимные функции, которые могут быть переданы в качестве параметров или сохранены как переменные. Они часто используются для определения кратких блоков кода, которые могут быть переданы в функции высшего порядка. Давайте рассмотрим основные аспекты лямбда-выражений:

Лямбда-выражения:

**Синтаксис**: Лямбда-выражение начинается с фигурных скобок {} и может иметь список параметров, разделенных запятыми, после которых идет стрелка ->, за которой следует тело функции.

kotlin

Копировать код

val sum: (Int, Int) -> Int = { a, b -> a + b }

**Передача лямбда-выражений**: Лямбда-выражения могут быть переданы в качестве аргументов функций, которые принимают функции высшего порядка.

kotlin

Копировать код

fun calculate(x: Int, y: Int, operation: (Int, Int) -> Int): Int {

return operation(x, y)

}

val result = calculate(10, 5) { a, b -> a + b }

**Замыкания (Closures)**: Лямбда-выражения в Kotlin являются замыканиями, что означает, что они могут иметь доступ к переменным из окружающего контекста, включая переменные за пределами лямбда-выражения.

kotlin

Копировать код

fun closureExample(): () -> Unit {

var counter = 0

return {

counter++

println("Counter: $counter")

}

}

val incrementCounter = closureExample()

incrementCounter() // Output: Counter: 1

incrementCounter() // Output: Counter: 2

Краткие способы записи лямбда-выражений:

**Сокращенный синтаксис**: Если лямбда-выражение имеет только один параметр, его можно опустить и использовать ключевое слово it для обращения к этому параметру.

kotlin

Копировать код

val square: (Int) -> Int = { it \* it }

**Если тип параметра известен**: Если тип параметра известен, его можно опустить.

kotlin

Копировать код

val square: (Int) -> Int = { x -> x \* x }

**Ссылка на методы (Method reference)**: Можно использовать ссылки на методы с помощью оператора двойного двоеточия ::.

kotlin

Копировать код

val isEven: (Int) -> Boolean = Int::isEven

Лямбда-выражения являются мощным инструментом в Kotlin для написания компактного и выразительного кода, особенно в сочетании с функциями высшего порядка и замыканиями. Вместе с краткими способами записи они делают код более читаемым и понятным.

## 18. Методы работы с коллекциями, принимающие функции, как аргумент. Filter, map. Count, sort, max

Методы работы с коллекциями, такие как filter, map, count, sort, max, предоставляют функциональные возможности для манипулирования элементами коллекций. Они могут принимать функции в качестве аргументов, что делает код более гибким и выразительным. Давайте рассмотрим каждый из них:

1. filter:

Метод filter фильтрует элементы коллекции, оставляя только те, которые соответствуют заданному условию.

Пример:

kotlin

Копировать код

val numbers = listOf(1, 2, 3, 4, 5, 6)

val evenNumbers = numbers.filter { it % 2 == 0 }

// evenNumbers содержит [2, 4, 6]

2. map:

Метод map преобразует каждый элемент коллекции с помощью функции, переданной в качестве аргумента.

Пример:

kotlin

Копировать код

val numbers = listOf(1, 2, 3, 4, 5)

val squaredNumbers = numbers.map { it \* it }

// squaredNumbers содержит [1, 4, 9, 16, 25]

3. count:

Метод count возвращает количество элементов в коллекции, соответствующих заданному условию.

Пример:

kotlin

Копировать код

val numbers = listOf(1, 2, 3, 4, 5, 6)

val evenCount = numbers.count { it % 2 == 0 }

// evenCount содержит 3

4. sort:

Метод sort сортирует элементы коллекции в соответствии с заданным компаратором.

Пример:

kotlin

Копировать код

val numbers = mutableListOf(3, 1, 4, 1, 5, 9, 2, 6)

numbers.sort()

// numbers содержит [1, 1, 2, 3, 4, 5, 6, 9]

5. max:

Метод max возвращает максимальный элемент в коллекции, используя заданный компаратор.

Пример:

kotlin

Копировать код

val numbers = listOf(3, 1, 4, 1, 5, 9, 2, 6)

val maxNumber = numbers.max()

// maxNumber содержит 9

Эти методы представляют лишь некоторые из возможностей работы с коллекциями в Kotlin. Они позволяют более компактно и выразительно выполнять операции над коллекциями, делая код более читаемым и эффективным.

## 19. Методы работы с коллекциями, принимающие функции, как аргумент. GroupBy, reduce, fold.

Методы работы с коллекциями в Kotlin, такие как groupBy, reduce и fold, позволяют выполнять различные операции над элементами коллекций с использованием функций, передаваемых в качестве аргументов. Давайте рассмотрим каждый из них:

1. groupBy:

Метод groupBy группирует элементы коллекции по результатам применения функции к каждому элементу. Результатом является Map, в котором ключами являются значения, возвращаемые функцией, а значениями - списки элементов, для которых функция возвращает то же самое значение.

Пример:

kotlin

Копировать код

val words = listOf("apple", "banana", "cherry", "avocado", "blueberry")

val groupedByFirstLetter = words.groupBy { it.first() }

// Результат: {a=[apple, avocado], b=[banana, blueberry], c=[cherry]}

2. reduce:

Метод reduce объединяет элементы коллекции, используя заданную функцию и возвращая единственное значение. Этот метод последовательно применяет функцию к каждой паре элементов, начиная с первых двух.

Пример:

kotlin

Копировать код

val numbers = listOf(1, 2, 3, 4, 5)

val sum = numbers.reduce { acc, num -> acc + num }

// Результат: 15 (1 + 2 + 3 + 4 + 5)

3. fold:

Метод fold похож на reduce, но имеет начальное значение, которое можно задать явно. Это начальное значение будет использоваться в качестве начального аккумулятора.

Пример:

kotlin

Копировать код

val numbers = listOf(1, 2, 3, 4, 5)

val sum = numbers.fold(0) { acc, num -> acc + num }

// Результат: 15 (0 + 1 + 2 + 3 + 4 + 5)

В обоих случаях reduce и fold возвращают единственное значение, но в fold можно указать начальное значение, что делает его более гибким в некоторых сценариях.

Эти методы представляют мощные инструменты для работы с коллекциями в Kotlin. Они позволяют выполнять различные операции с элементами коллекций, используя функции, что делает код более читаемым и компактным.

## 20. Работа с последовательностями в котлин, в java

Работа с последовательностями в Kotlin и Java имеет некоторые сходства, но в Kotlin предоставляются более высокоуровневые абстракции для работы с последовательностями, что делает код более компактным и эффективным. Давайте рассмотрим основные различия и сходства:

**В Kotlin:**

**Sequences (Последовательности)**:

В Kotlin есть тип данных Sequence, который представляет собой ленивую коллекцию элементов, обрабатываемых последовательно.

Последовательности обладают ленивой природой, что означает, что операции выполняются только при необходимости, и это может улучшить производительность.

Операции над последовательностями, такие как map, filter, flatMap и другие, не выполняются немедленно, а формируют цепочку операций, которая будет выполнена только при вызове терминальной операции (например, toList, first, count и т. д.).

Пример работы с последовательностями в Kotlin:

kotlin

Копировать код

val list = listOf(1, 2, 3, 4, 5)

val result = list.asSequence()

.filter { it % 2 == 0 }

.map { it \* it }

.toList()

**В Java:**

**Stream API**:

В Java для работы с последовательностями элементов используется Stream API, введенный в Java 8.

Stream представляет собой поток элементов, который можно преобразовывать, фильтровать и агрегировать.

Операции над потоками могут быть промежуточными (например, map, filter, flatMap) или терминальными (например, collect, count, findFirst).

Пример работы с потоками в Java:

java

Копировать код

List<Integer> list = Arrays.asList(1, 2, 3, 4, 5);

List<Integer> result = list.stream()

.filter(n -> n % 2 == 0)

.map(n -> n \* n)

.collect(Collectors.toList());

**Сходства:**

**Функциональные операции**:

Оба подхода позволяют использовать функциональные операции для работы с элементами коллекции/последовательности.

Оба поддерживают операции, такие как map, filter, reduce, collect и другие.

**Ленивые операции**:

В обоих случаях могут использоваться ленивые операции для улучшения производительности, когда не все элементы коллекции/последовательности должны быть обработаны сразу.

**Параллельные операции**:

Как в Kotlin, так и в Java можно выполнять параллельные операции над потоками/последовательностями.

Использование последовательностей в Kotlin или потоков в Java зависит от требований и предпочтений разработчика. В Kotlin Sequence является более высокоуровневым и удобным в использовании, но Stream API в Java предоставляет мощные инструменты для работы с последовательностями.

## 21. Лямбда выражения с получателями.

Лямбда-выражения с получателями, также известные как лямбда с получателем (receiver lambda), являются мощным инструментом в Kotlin, который позволяет передавать объект-получатель в лямбда-выражение и вызывать методы этого объекта без явного указания имени переменной. Это делает код более читаемым и уменьшает его объем. Давайте рассмотрим их синтаксис и использование:

**Синтаксис:**

kotlin

Копировать код

val lambdaWithReceiver: ReceiverType.() -> ReturnType = {

// Тело лямбда-выражения

}

ReceiverType - тип объекта-получателя.

ReturnType - тип возвращаемого значения лямбда-выражения.

**Пример использования:**

kotlin

Копировать код

data class Person(val name: String)

fun main() {

val printName: Person.() -> Unit = {

println(this.name)

}

val person = Person("John")

person.printName() // Вывод: John

}

В этом примере printName - это лямбда с получателем, которая принимает объект типа Person в качестве получателя. Внутри тела лямбда-выражения this ссылается на объект-получатель, поэтому мы можем обращаться к его членам без явного указания имени переменной. В данном случае мы вызываем метод name, который принадлежит объекту Person.

**Применение лямбда с получателем:**

**Работа с DSL (Domain-Specific Language)**: Лямбда с получателем часто используется в DSL для создания выразительного и читаемого синтаксиса.

**Улучшение API**: Они могут быть использованы для улучшения читаемости и уменьшения объема кода в API.

**Расширение функциональности классов**: Позволяют добавлять дополнительную функциональность к классам без наследования или изменения оригинального класса.

Лямбда-выражения с получателями являются мощным инструментом, который делает код более читаемым и компактным. Они особенно полезны при создании DSL и улучшении API.

## 22. Поддержка значения null

Поддержка значения null в Kotlin осуществляется с помощью системы типов, которая предоставляет безопасные средства работы с нулевыми значениями. Вот основные инструменты поддержки значения null:

Типы с отметкой о нуле (Nullable Types):

В Kotlin, по умолчанию, типы данных не допускают значение null. Однако, если типу необходима поддержка null, его можно объявить с отметкой о нуле, добавив ? к типу:

kotlin

Копировать код

var nullableString: String? = null

Теперь переменная nullableString может содержать значение null.

Безопасные операции с нулевым значением:

Для безопасной работы с нулевыми значениями в Kotlin используются операторы безопасного вызова ?., оператор нулевого объединения ?:, а также функция проверки на наличие ?.let { }.

kotlin

Копировать код

val length = nullableString?.length // Если nullableString равно null, length также будет равен null

val result = nullableString ?: "Default" // Если nullableString равно null, result будет равен "Default"

nullableString?.let { println(it) } // Блок выполнится только если nullableString не равно null

Проверка на нулевое значение:

Для явной проверки на наличие нулевого значения используется оператор if или оператор ?.:

kotlin

Копировать код

if (nullableString != null) {

// Блок выполнится, если nullableString не равно null

}

nullableString?.toUpperCase() // Будет вызван метод toUpperCase только если nullableString не равно null

Функции с отметкой о нуле:

Функции также могут принимать аргументы с отметкой о нуле и возвращать значения с отметкой о нуле:

kotlin

Копировать код

fun getLength(nullableString: String?): Int? {

return nullableString?.length

}

Поддержка значения null в Kotlin позволяет писать более безопасный код, предотвращая многие типичные ошибки, связанные с нулевыми значениями.

## 23. Базовые типы в котлин.

ChatGPT

В Kotlin, как и во многих других языках программирования, есть базовые типы данных, которые представляют основные значения, такие как целые числа, числа с плавающей точкой, символы, логические значения и строки. Вот основные базовые типы данных в Kotlin:

**1. Целочисленные типы (Integer Types):**

**Byte**: 8-битное целое число. Диапазон значений от -128 до 127.

**Short**: 16-битное целое число. Диапазон значений от -32768 до 32767.

**Int**: 32-битное целое число. Диапазон значений от -2^31 до 2^31-1.

**Long**: 64-битное целое число. Диапазон значений от -2^63 до 2^63-1.

**2. Числа с плавающей точкой (Floating-Point Types):**

**Float**: 32-битное число с плавающей точкой одинарной точности.

**Double**: 64-битное число с плавающей точкой двойной точности.

**3. Символьный тип (Character Type):**

**Char**: 16-битный символ Unicode. В Kotlin символы представлены в одинарных кавычках, например, 'a', 'b', 'c'.

**4. Логический тип (Boolean Type):**

**Boolean**: логическое значение, может быть true или false.

**5. Строковый тип (String Type):**

**String**: последовательность символов Unicode. Строки представлены в двойных кавычках, например, "Hello, World!".

**Примеры использования базовых типов:**

kotlin

Копировать код

val number: Int = 42

val pi: Double = 3.14159

val char: Char = 'A'

val isTrue: Boolean = true

val greeting: String = "Hello, Kotlin!"

**Примечания:**

В Kotlin нет неявных преобразований между базовыми типами данных, поэтому преобразования должны быть явно указаны.

Все базовые типы данных в Kotlin являются объектами, что позволяет вызывать методы и использовать операторы для них, как для любых других объектов.

Kotlin также поддерживает типы данных с отметкой о нуле (nullable types), что позволяет значениям базовых типов быть равными null.

Кроме того, Kotlin предоставляет функции для выполнения преобразований между различными базовыми типами данных.

## 24. Перегрузка арифметических операторов.

ChatGPT

В Kotlin, так же как и во многих других языках программирования, можно перегружать арифметические операторы для пользовательских классов. Перегрузка операторов позволяет определить своеобразное поведение для операторов, таких как +, -, \*, / и других, когда они используются с объектами пользовательских классов. Вот как это делается:

Перегрузка бинарных арифметических операторов:

kotlin

Копировать код

class Point(val x: Int, val y: Int) {

operator fun plus(other: Point): Point {

return Point(x + other.x, y + other.y)

}

}

fun main() {

val point1 = Point(3, 4)

val point2 = Point(1, 2)

val sum = point1 + point2 // Используется перегруженный оператор +

println("Sum: (${sum.x}, ${sum.y})") // Вывод: Sum: (4, 6)

}

В приведенном примере plus - это перегруженная функция, определенная в классе Point, которая определяет поведение оператора + для объектов класса Point. Теперь мы можем использовать оператор + для сложения двух объектов типа Point.

Перегрузка унарных арифметических операторов:

kotlin

Копировать код

data class Point(val x: Int, val y: Int) {

operator fun unaryMinus(): Point {

return Point(-x, -y)

}

}

fun main() {

val point = Point(3, 4)

val negated = -point // Используется перегруженный оператор унарного минуса

println("Negated: (${negated.x}, ${negated.y})") // Вывод: Negated: (-3, -4)

}

В этом примере unaryMinus - это перегруженная функция, определенная в классе Point, которая определяет поведение унарного оператора -. Теперь мы можем использовать унарный оператор - для получения отрицательного значения объекта типа Point.

Другие арифметические операторы:

Таким же образом можно перегружать и другие арифметические операторы, такие как minus (-), times (\*), div (/) и другие.

kotlin

Копировать код

operator fun minus(other: Point): Point { ... }

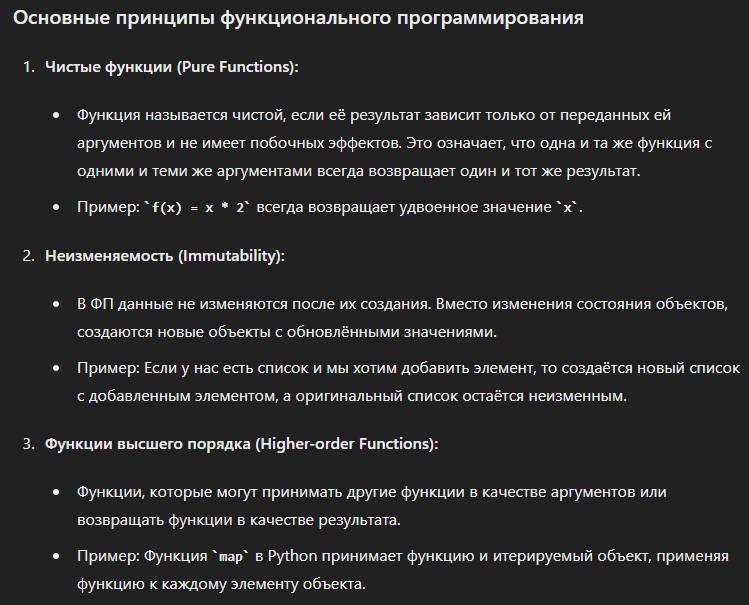
operator fun times(other: Point): Point { ... }

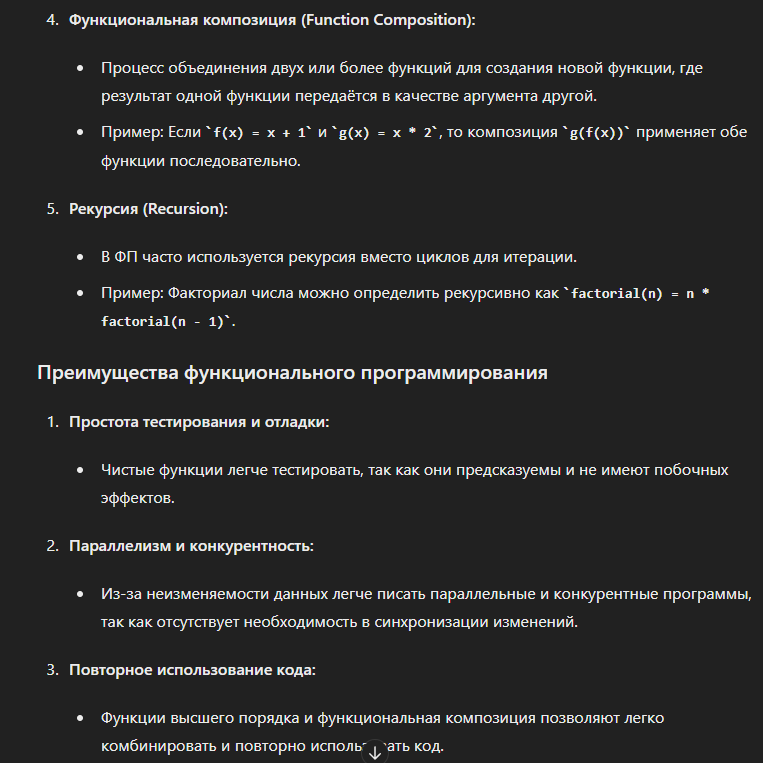
operator fun div(other: Point): Point { ... }

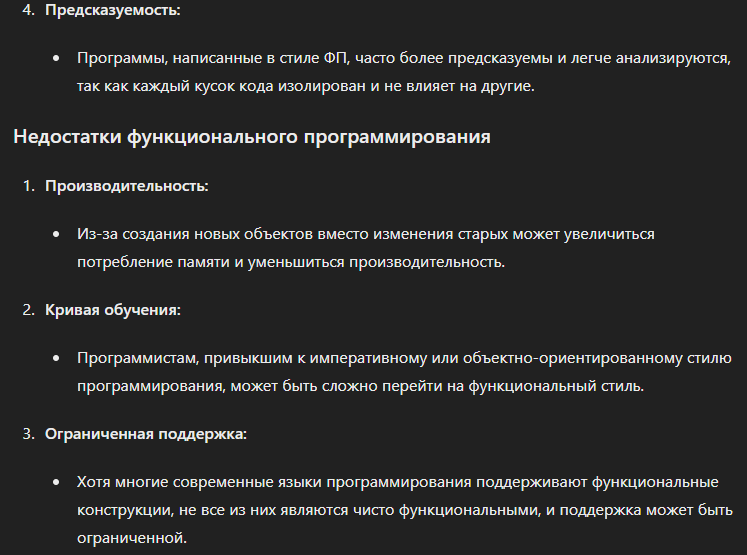
Перегрузка арифметических операторов в Kotlin позволяет создавать более выразительный и интуитивно понятный код при работе с пользовательскими типами данных.

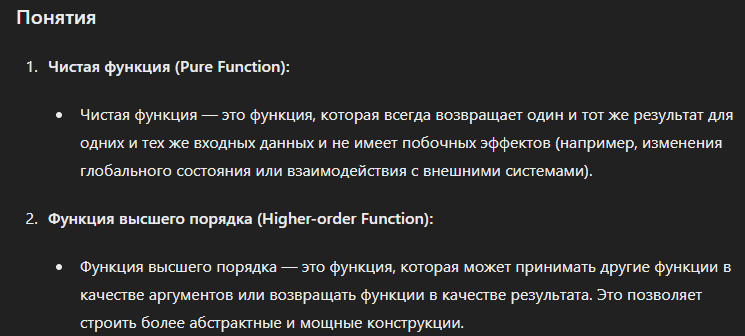
# **Лямбда**

## 1. Опишите основные принципы функционального программирования и вытекающие из них преимущества и недостатки. Опишите понятия высшая функция, чистая

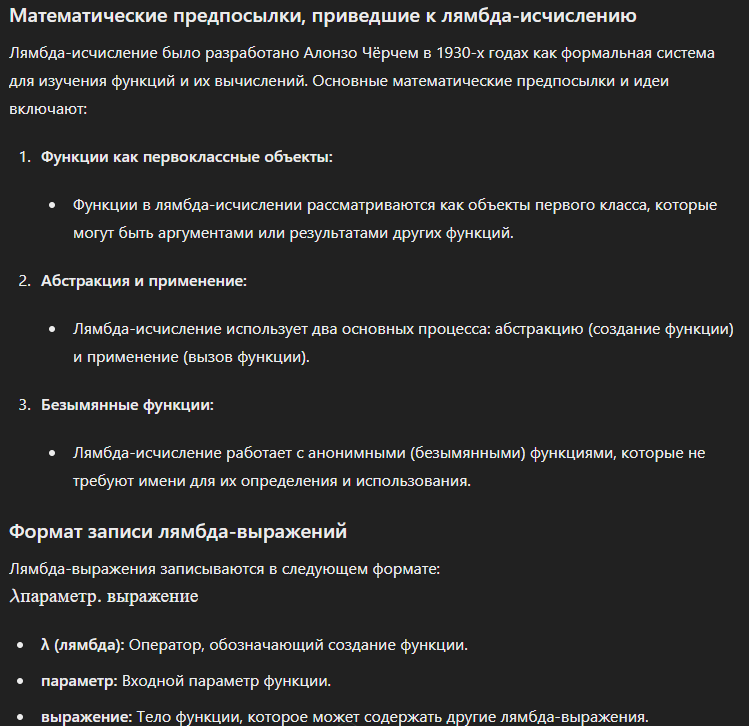


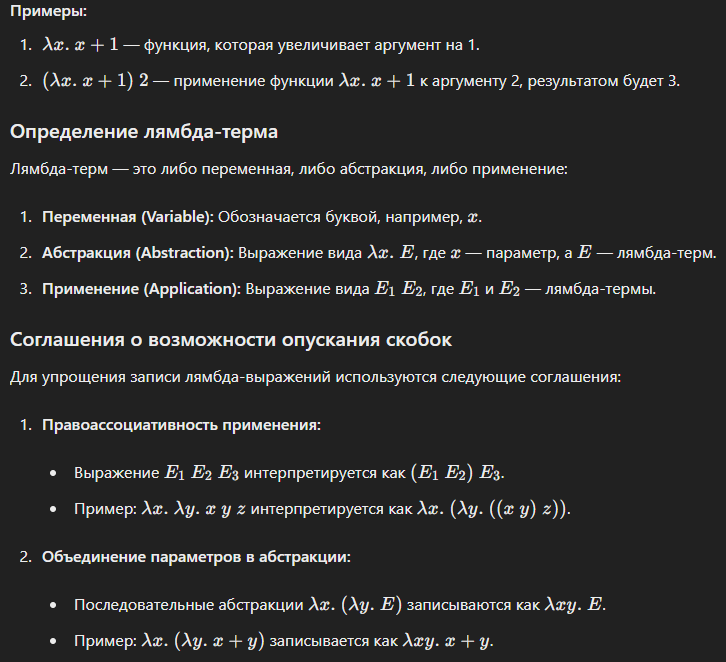






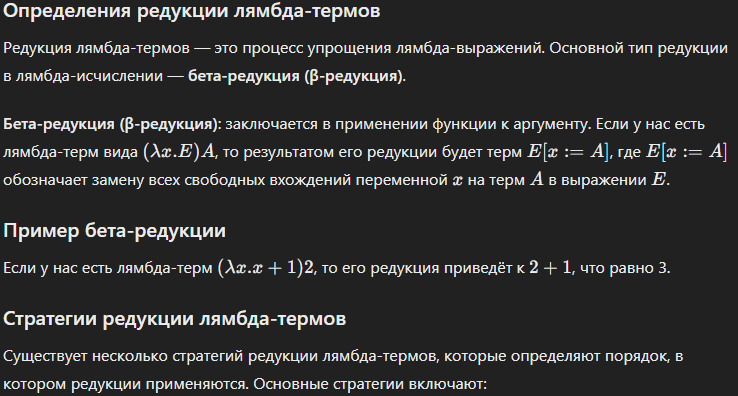
## 2. Опишите математические предположения, которые привели к лямбда исчислению и объясните формат записи лямбда выражений. Дайте определение лямбда терма. Опишите соглашения о возможности опускать скобки, принятые в лябмда выражении.

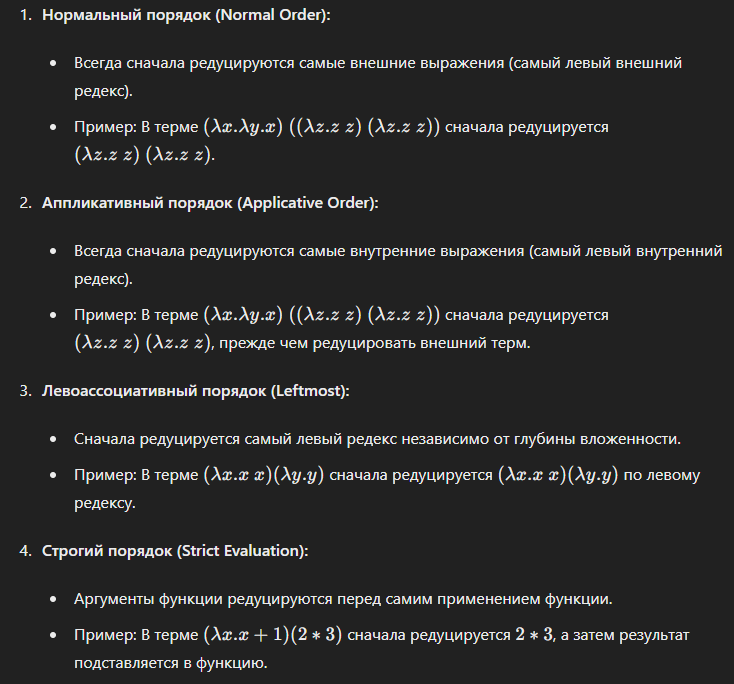


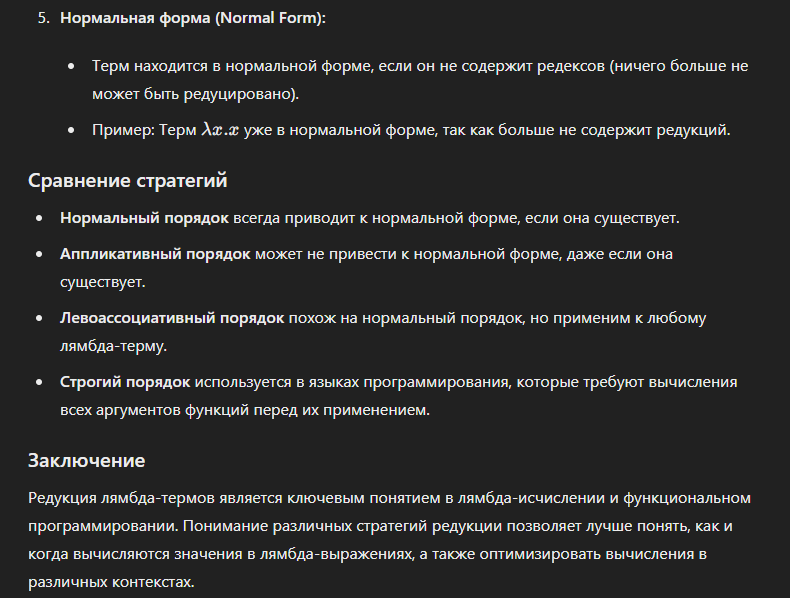




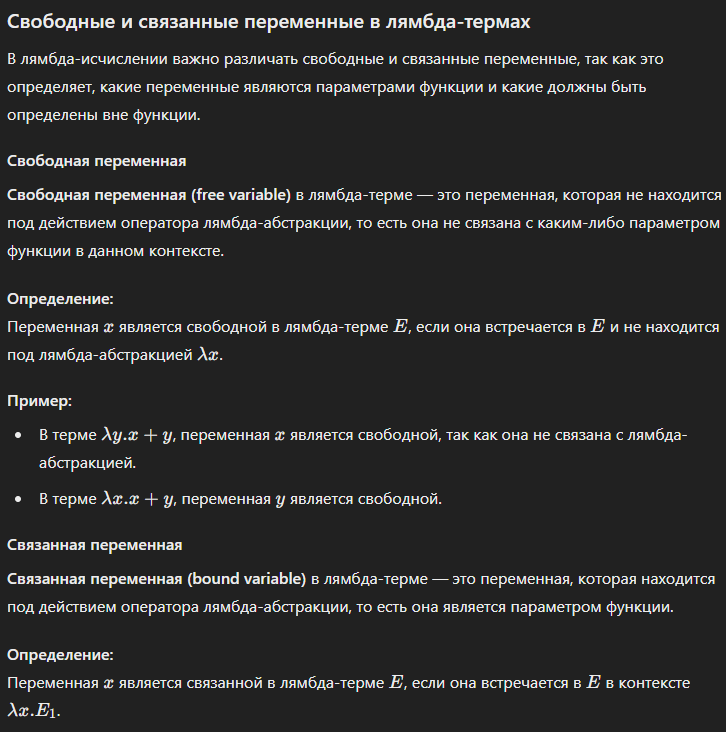
## 3. Дайте определения редукции лямбда термов. Опишите стратегии редукции лямбда термов.

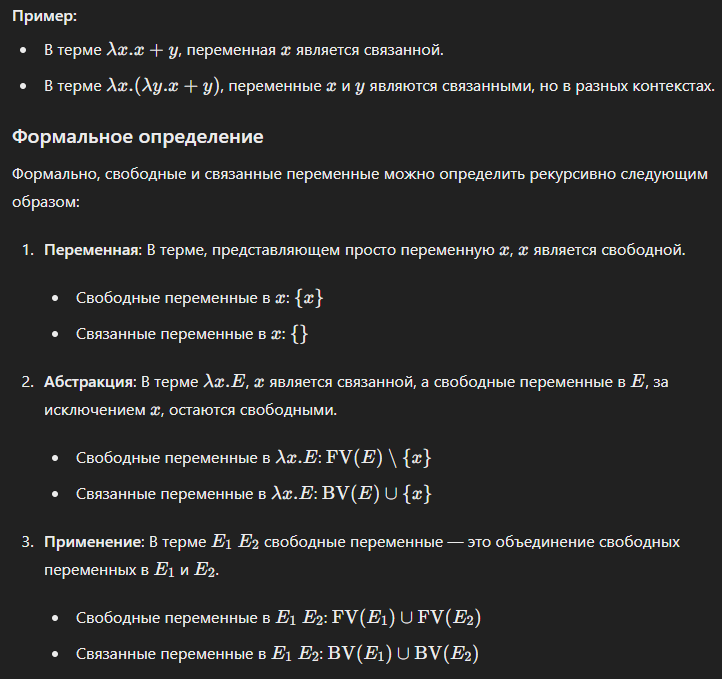


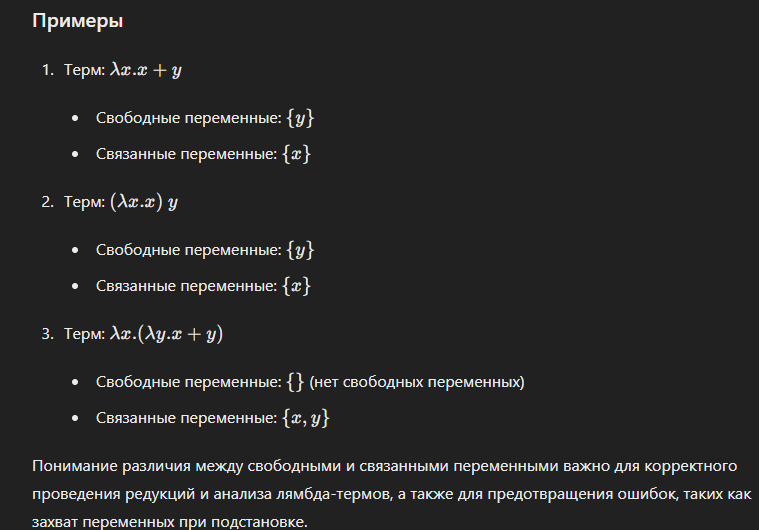




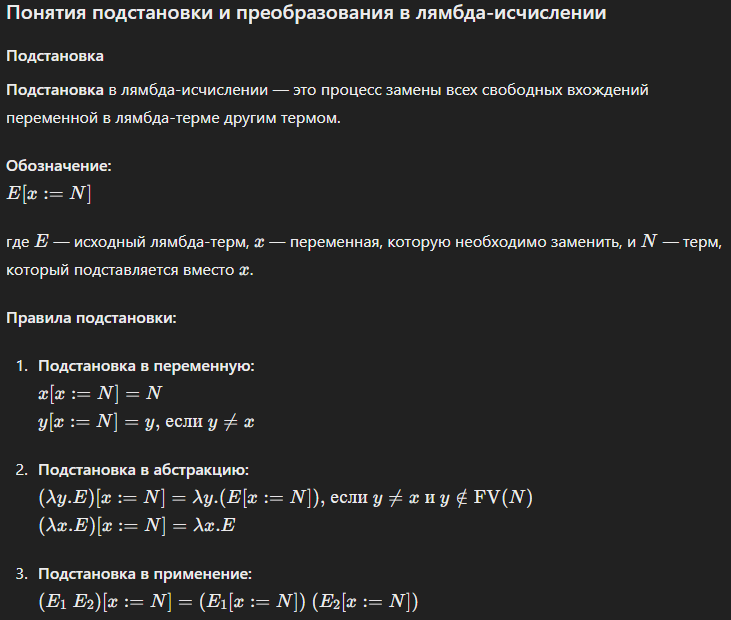
## 4. Дайте определения свободных и связанных переменных в лямбда термах.

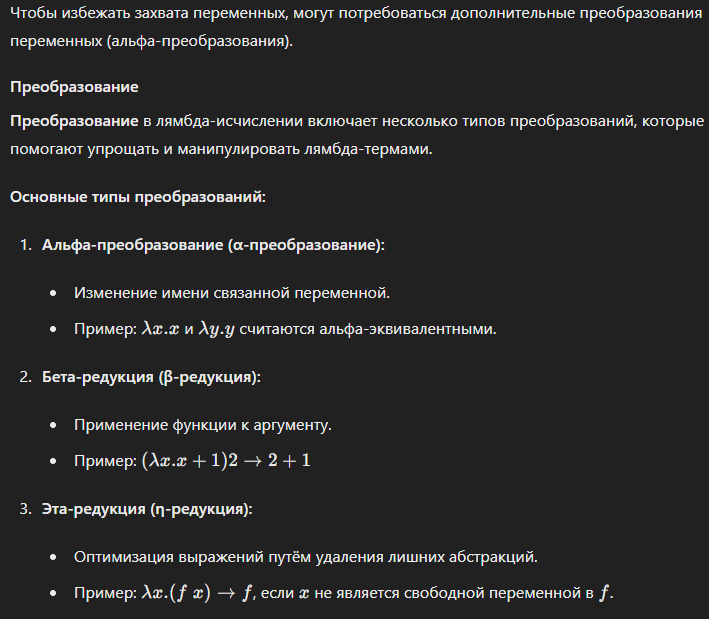


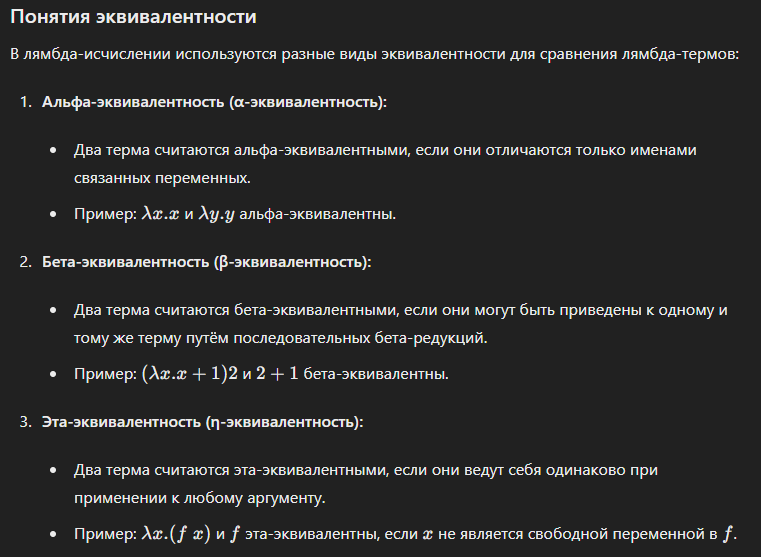


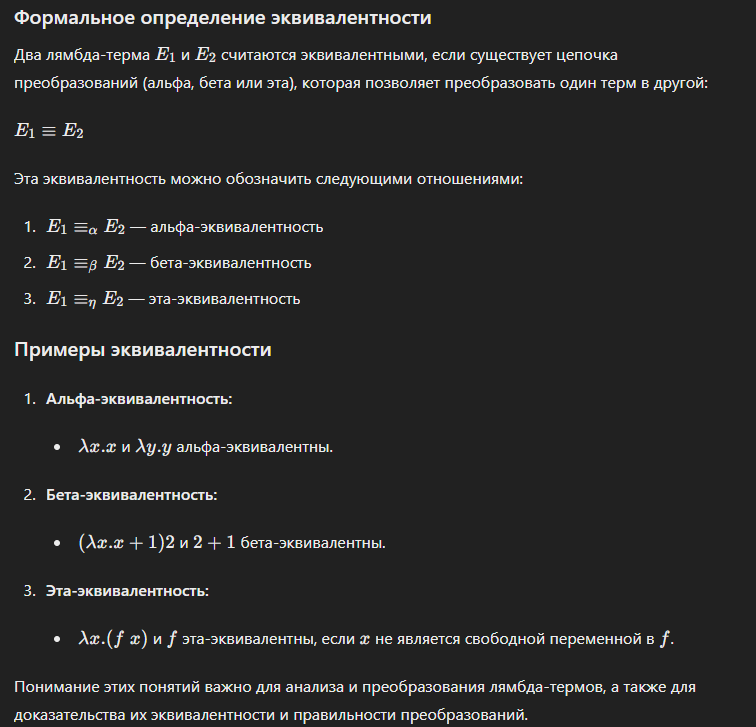


## 5. Дайте понятия подстановки и преобразования. Сформулируйте понятия эквивалентности.

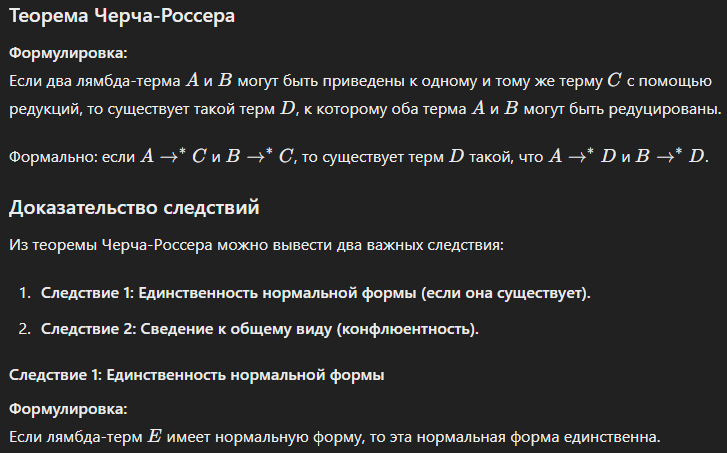


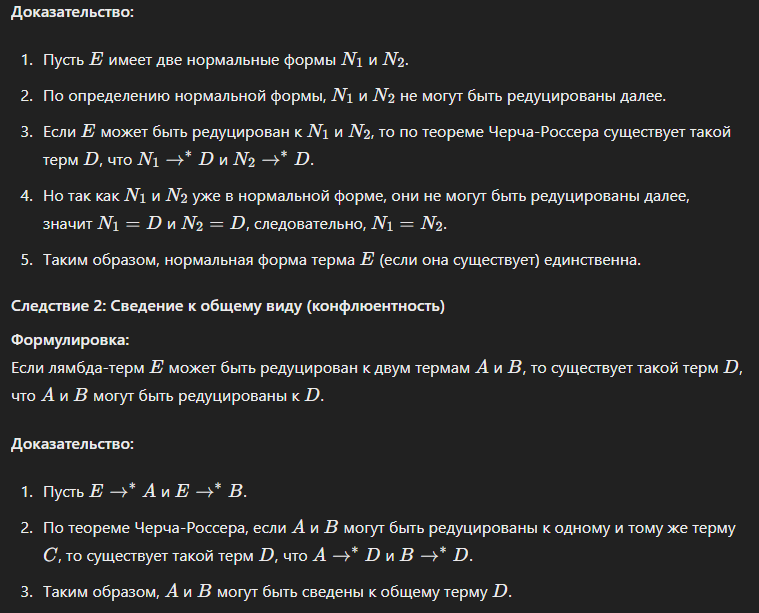




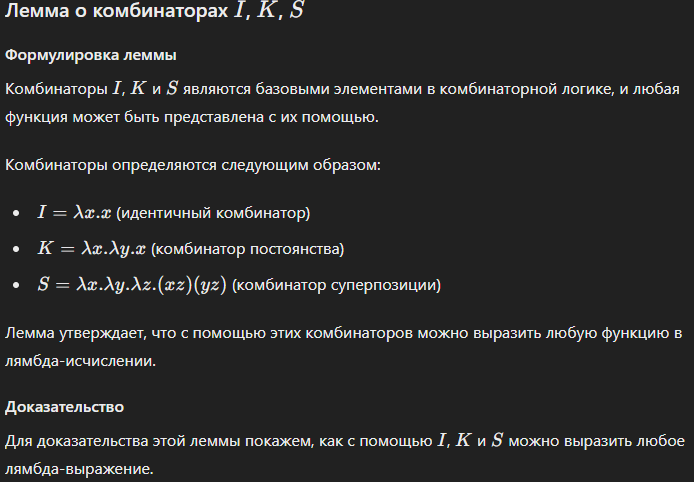


## 6. Сформулируйте теорему Черча-Россера и докажите два следствия из нее.

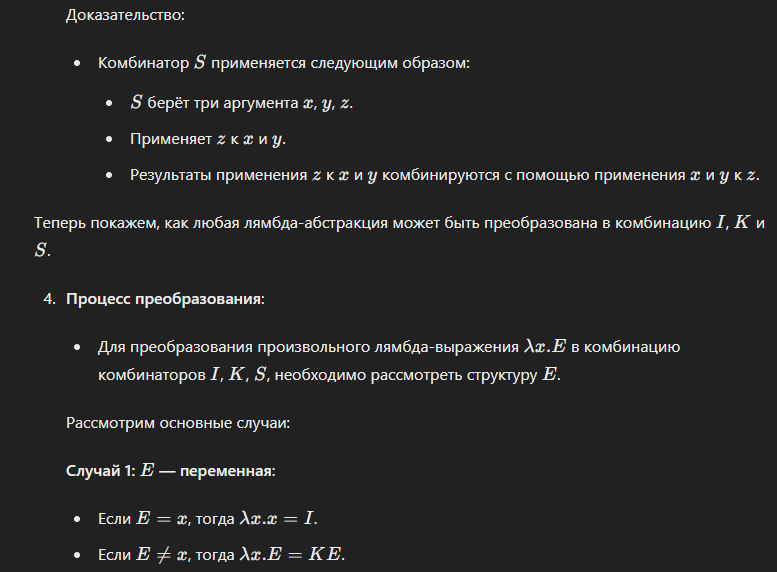


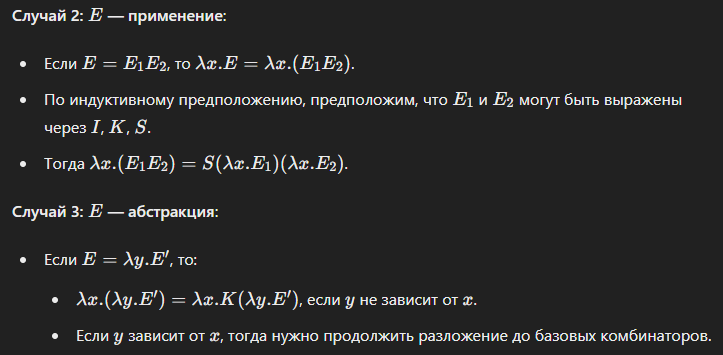


## 7. Сформулируйте и докажите лемму о комбинаторах I, K, S

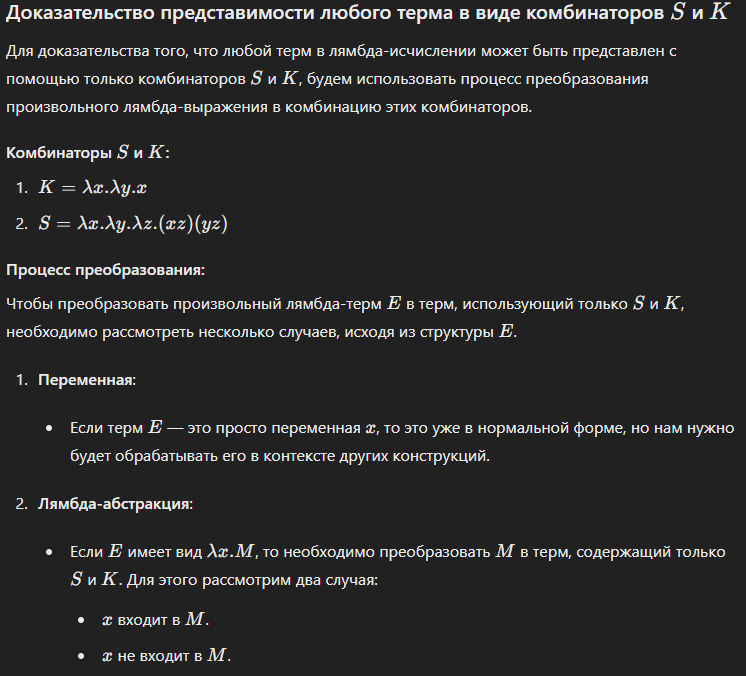


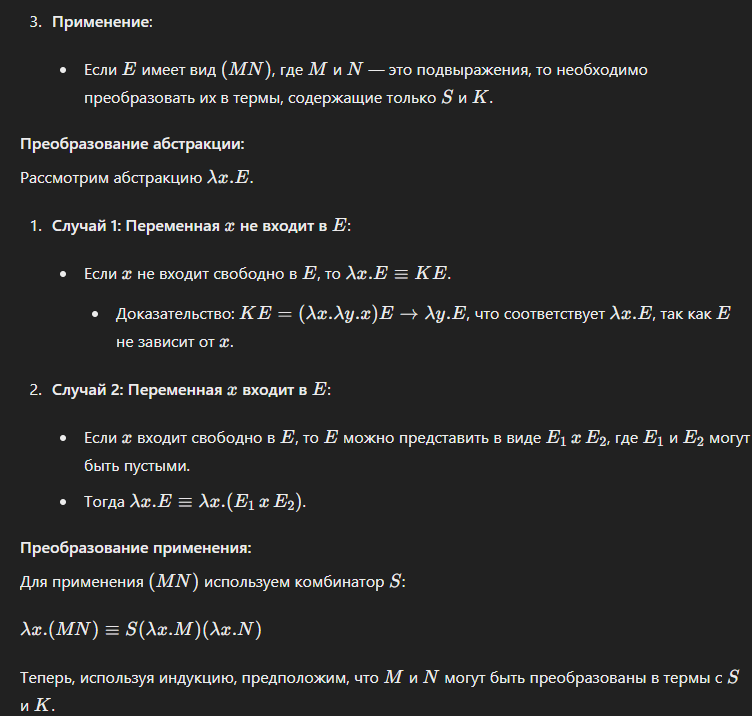


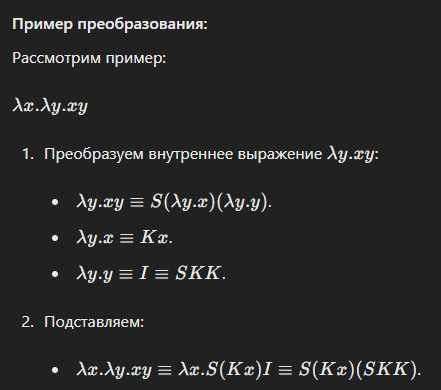




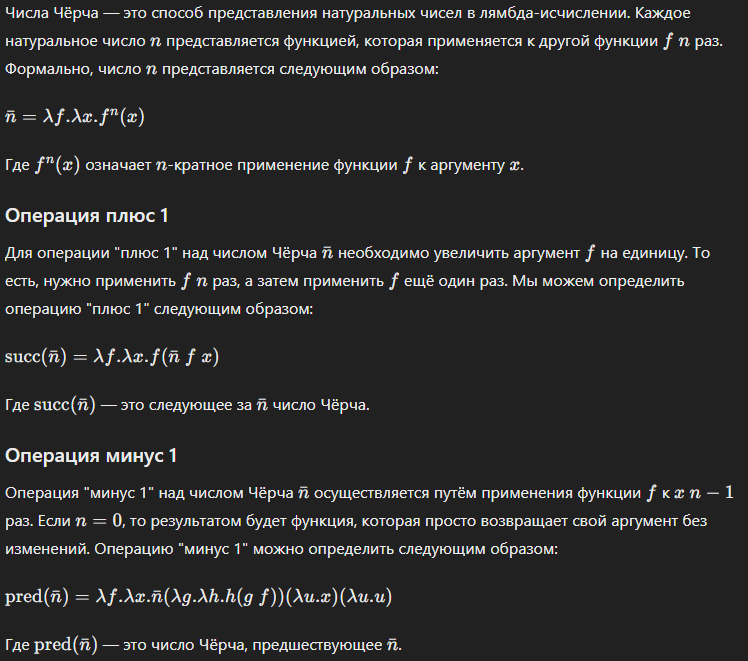
## 8. Докажите, что любой терм представим в виде комбинаторов S K

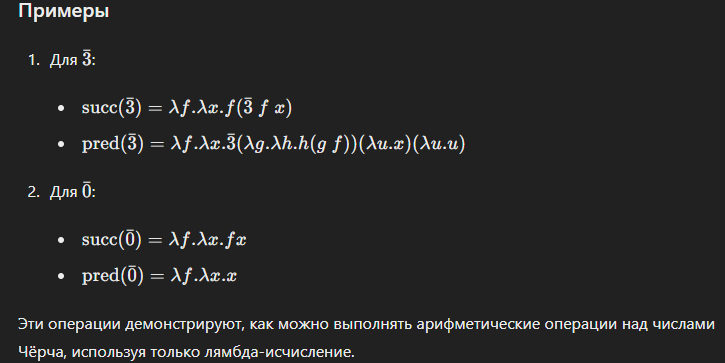




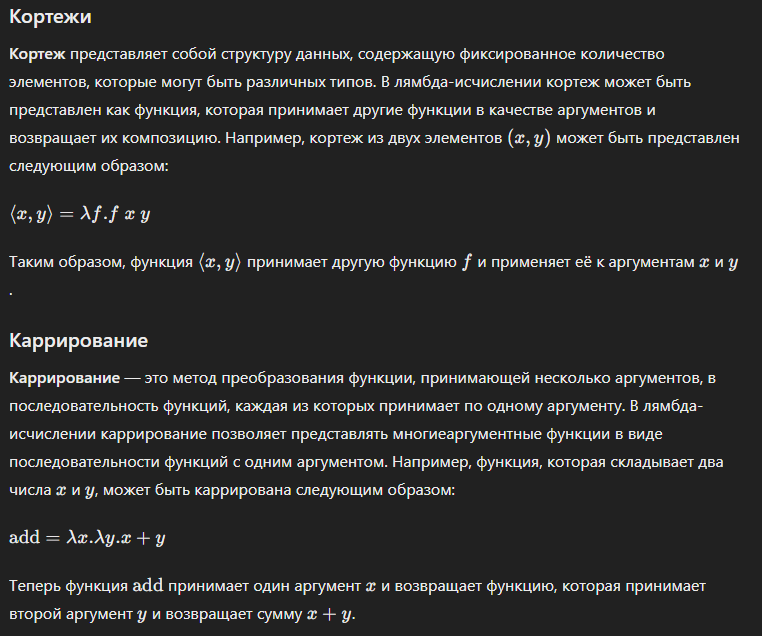


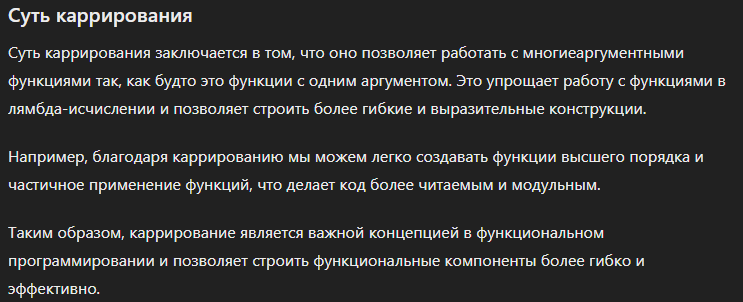
## 9. Числа Черча. Операция плюс 1 Операция (-1).



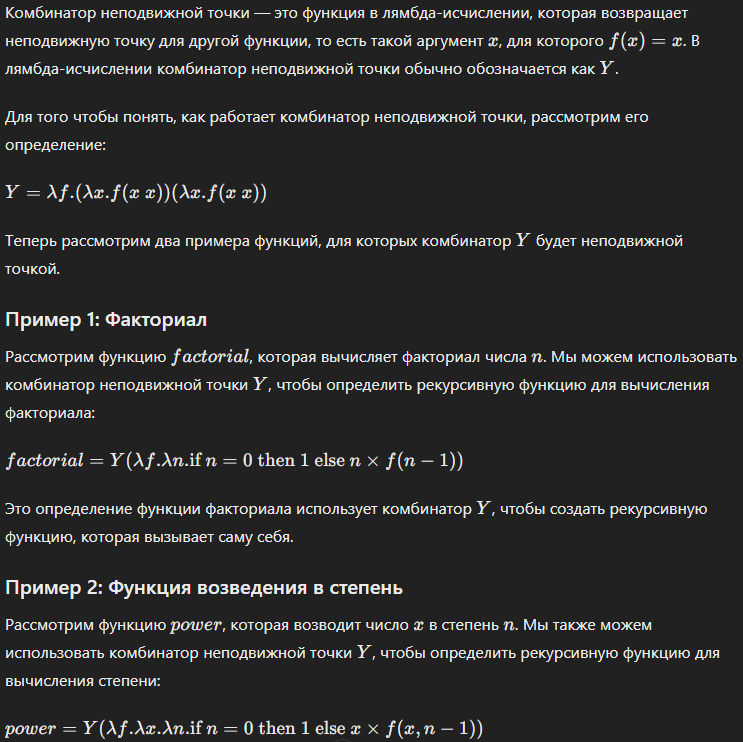


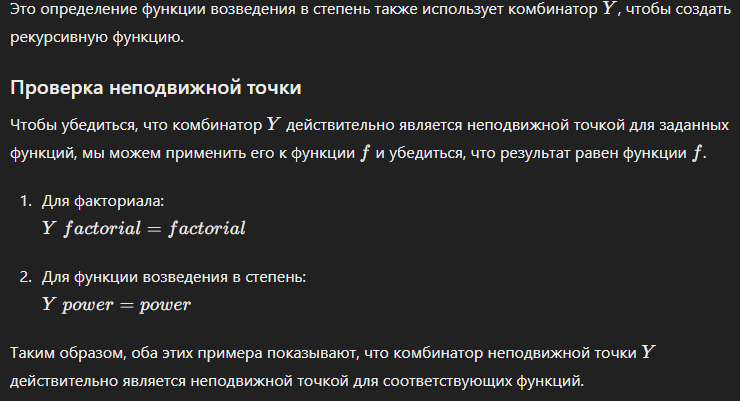
## 10. Кортежи и Каррирование в лямбда исчилении. Объясните, в чем суть каррирования.



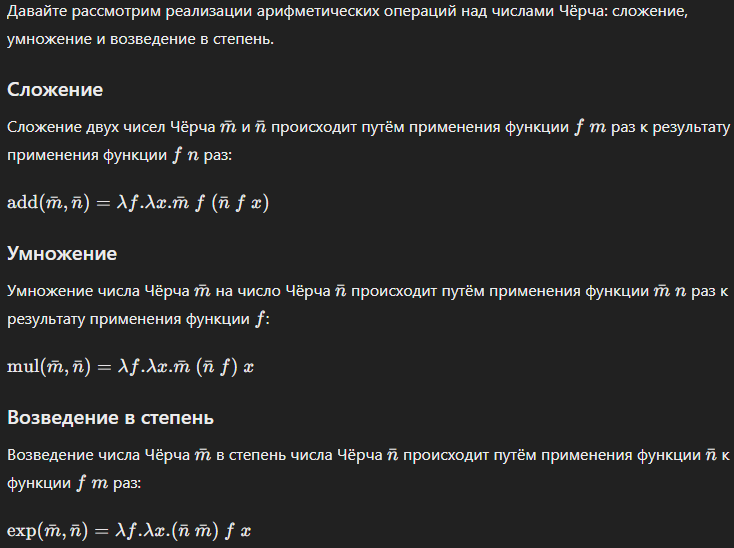


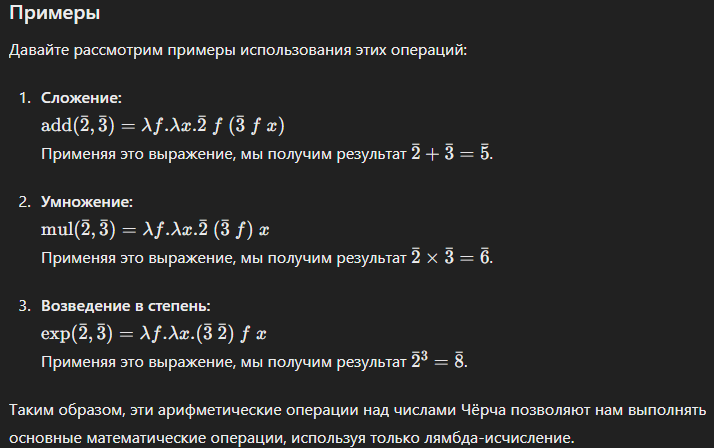
## 11. Комбинатор неподвижной точки. Приведите два примера, покажите, что они комбинаторы неподвижной точки.



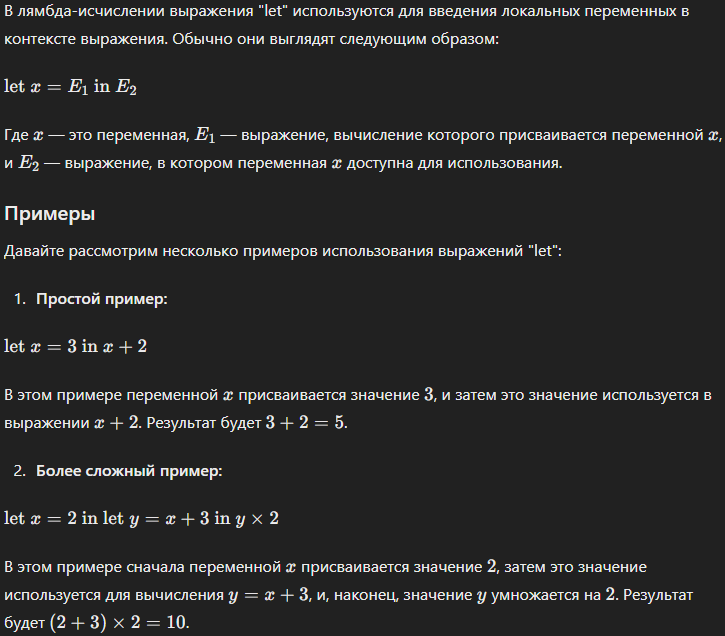


## 12. Арифметические операции над числами Черча + \* ^



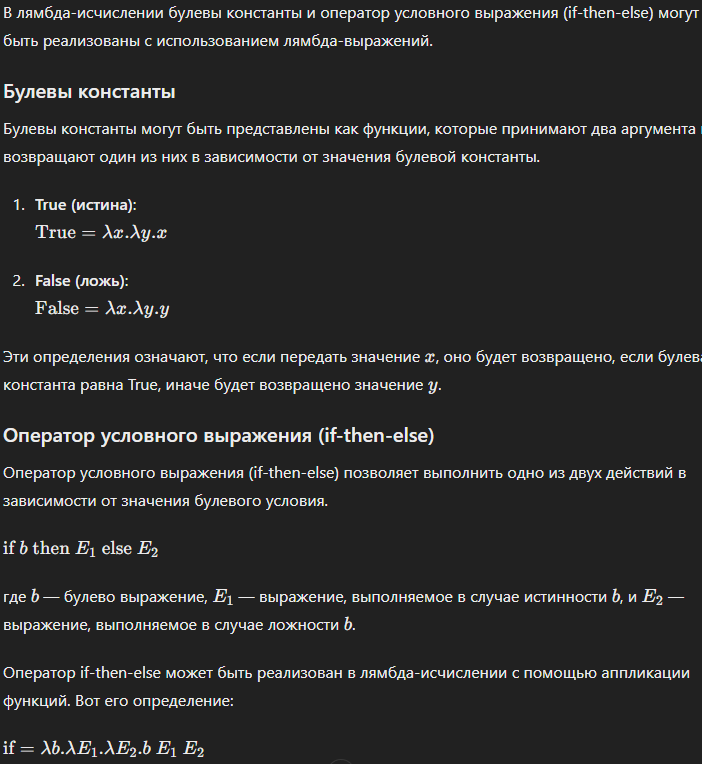


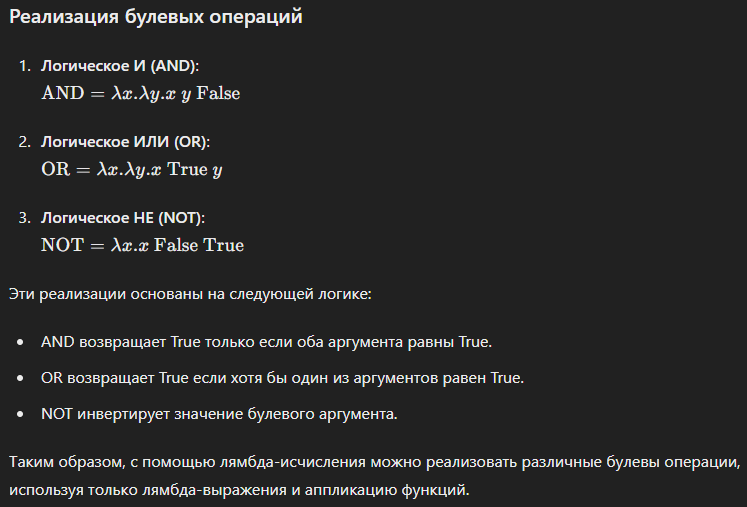
## 13. Let выражения



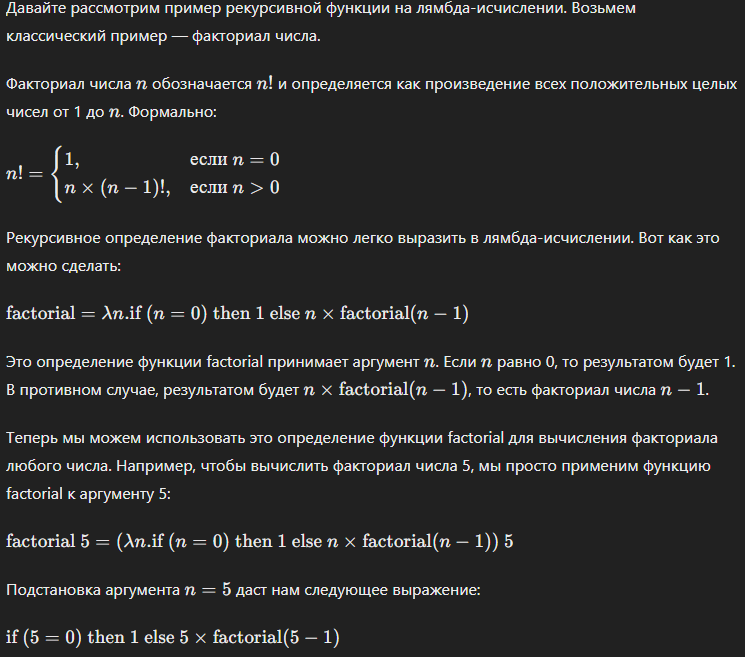


## 14. Булевы константы и оператор if. Реализация булевых операций.



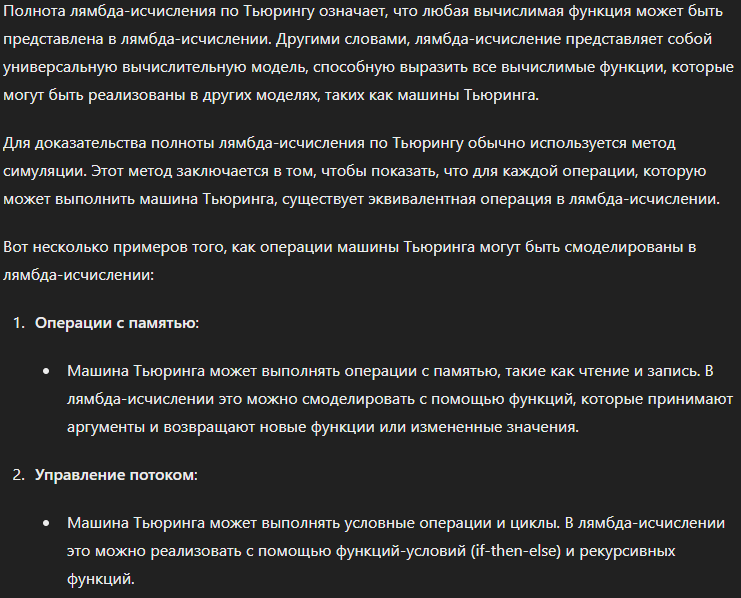


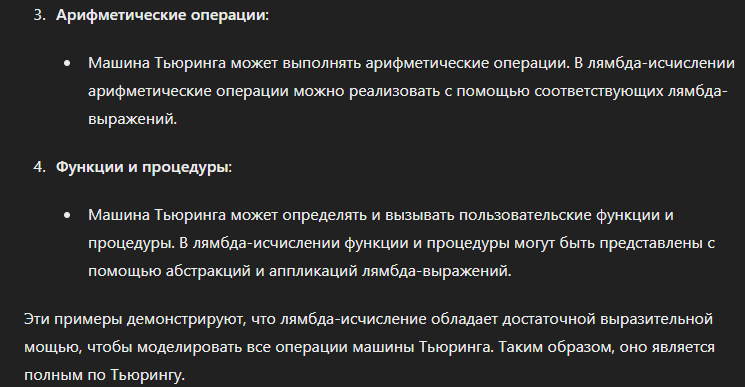
## 15. Рекурсивные функции (на примере любой функции).



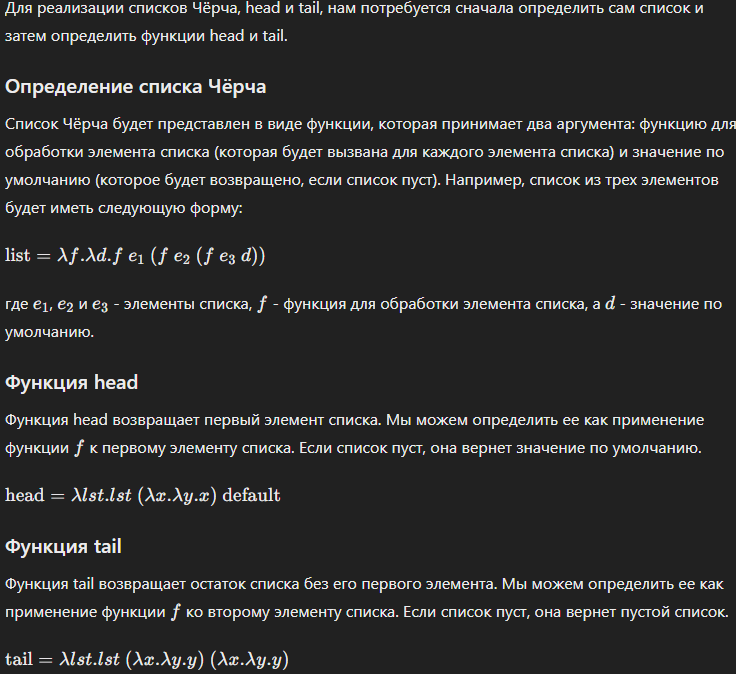


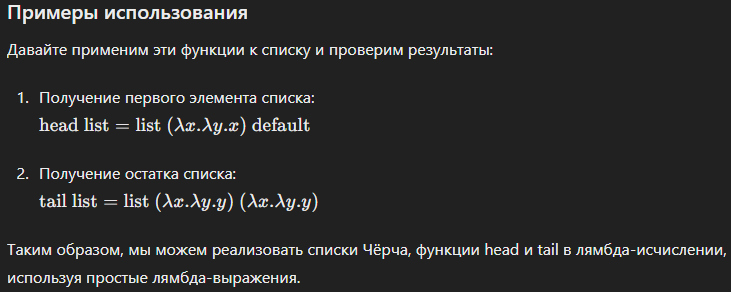
## 16. Полнота лямбда исчисления по Тьюрингу





## 17. Реализация списков Черча, head, tail.





## 18. Покажите набор функций преобразования списков Чёрча.

