1. **Опишите математические предположения, которые привели к лямбда исчислению и объясните формат записи лямбда выражений.**

**Базовые функции и переменные**: В лямбда-исчислении используются только функции и переменные. Функции могут принимать любое количество аргументов, но могут возвращать только одно значение.

**Абстракция**: В лямбда-исчислении функции определяются через абстракцию, то есть через выражение, которое описывает, как функция должна работать на своих аргументах, без конкретных значений этих аргументов.

**Подстановка:** В лямбда-исчислении подстановка — это процесс замены переменной на другое выражение в функции. Это ключевой механизм, позволяющий работать с абстракциями.

**Вычисление:** В лямбда-исчислении вычисления происходят по следующим правилам:

Если выражение является переменной, оно не изменяется.

Если выражение является функцией, применяется правило применения функции.

Если выражение является абстракцией, происходит подстановка.

λx.E, где λ (лямбда) — символ, обозначающий начало абстракции, x — переменная, на которую абстрагируется выражение, . — точка, разделяющая переменную и выражение, E — выражение, которое определяет функцию.

1. **Дайте определение лямбда терма.**

Лямбда-терм — это выражение в лямбда-исчислении, которое может быть функцией, переменной или абстракцией.

1. **Опишите соглашения о возможности опускать скобки, принятые в лябмда выражении.**

**Опускание скобок вокруг аргументов функций**: если функция применяется к одному аргументу, то скобки вокруг этого аргумента можно опустить. Например, (λx.x \* x) y можно записать как λx.x \* x y.

**Опускание скобок вокруг абстракций**: если абстракция применяется к одному аргументу, то скобки вокруг этого аргумента можно опустить. Например, (λx.x \* x) y можно записать как λx.x \* x y.

**Опускание скобок вокруг выражений внутри абстракций**: если выражение внутри абстракции состоит из одной функции, то скобки вокруг этой функции можно опустить. Например, λx.(x \* x) можно записать как λx.x \* x.

**Опускание скобок вокруг выражений внутри функций**: если выражение внутри функции состоит из одной функции, то скобки вокруг этой функции можно опустить. Например, (λx.x \* x) y можно записать как λx.x \* x y.

**Опускание скобок вокруг выражений внутри выражений**: если выражение внутри другого выражения состоит из одной функции, то скобки вокруг этой функции можно опустить. Например, λx. (λy.x \* y) z можно записать как λx.λy.x \* y z.

1. **На примере лямбда термов опишите принцип каррирования и его работы.**

Каррирование — это процесс преобразования функции с несколькими аргументами в последовательность функций, каждая из которых принимает один аргумент.

λx.(λy.x + y)5 -> (λy.5 + y)3 -> 5+3=8

1. **Дайте определения свободных и связанных переменных в лямбда термах.**

Свободная переменная — это переменная, которая определена вне области видимости функции или выражения, в котором она используется, и не связана с параметрами этой функции или выражения. Свободные переменные могут быть изменены или доступны для чтения в любой части программы, где они определены. В контексте лямбда-термов, свободные переменные могут быть использованы внутри лямбда-выражения, но они не являются частью его параметров.

Связанная переменная — это переменная, которая является параметром функции или выражения, в котором она используется. Связанные переменные привязаны к конкретному вызову функции и не могут быть изменены вне этой функции. В контексте лямбда-термов, связанные переменные являются параметрами лямбда-выражения.

1. **Дайте определения редукции лямбда термов.**

Редукция лямбда-термов — это процесс приведения лямбда-терма к более простому или каноническому виду с использованием определенных правил преобразования. В контексте лямбда-исчисления, это ключевой процесс, который позволяет вычислять значения функций и их свойства. Редукция может быть выполнена по различным правилам, в зависимости от используемой системы лямбда-исчисления.

1. **Опишите стратегии редукции лямбда термов.**

Стратегии редукции лямбда-термов определяют последовательность и правила, по которым выполняются преобразования лямбда-термов с целью приведения их к более простому или каноническому виду.

**1. Нормальная форма**

Нормальная форма — это состояние, в котором лямбда-терм не может быть дальше упрощен с помощью β-редукции. Это означает, что все возможные β-редукции были применены. В нормальной форме лямбда-терм не содержит свободных переменных, кроме тех, которые являются параметрами лямбда-выражения.

**2. Стратегия η-редукции**

Стратегия η-редукции фокусируется на упрощении функций, которые просто возвращают свои аргументы, применяя η-редукцию. Это позволяет упростить лямбда-термы, которые могут быть представлены более простыми формами.

**3. Стратегия β-редукции**

Стратегия β-редукции фокусируется на применении β-редукции для замены параметров функций на соответствующие аргументы. Это позволяет вычислить значения функций, заменяя параметры на конкретные значения.

**4. Стратегия альфа-редукции**

Стратегия альфа-редукции используется для переименования переменных в лямбда-терме без изменения его значения. Это позволяет избежать конфликтов имен при вложенных функциях и упрощает анализ лямбда-термов.

**5. Стратегия смешанная**

Стратегия смешанная комбинирует различные стратегии редукции, включая β-редукцию, альфа-редукцию и η-редукцию, в зависимости от конкретного лямбда-терма и целей редукции. Это может включать в себя применение β-редукции для вычисления значений, альфа-редукцию для устранения конфликтов имен и η-редукцию для упрощения функций, которые просто возвращают свои аргументы.

1. **Дайте понятия подстановки и преобразования.**

**Подстановка** — это процесс замены переменной на другое выражение внутри лямбда-терма. Это ключевой механизм, позволяющий применять β-редукцию (β-редукция) — процесс применения функции к её аргументу. Подстановка позволяет заменить параметры функции на конкретные значения, что является основой для вычисления значений функций в лямбда-исчислении.

**Преобразования**— это более широкий термин, который включает в себя различные процессы преобразования лямбда-термов, такие как подстановка, альфа-редукция (переименование переменных), β-редукция (применение функции к её аргументу) и η-редукция (упрощение функций, которые просто возвращают свои аргументы). Преобразования позволяют упрощать и анализировать лямбда-термы, приводя их к более простым или каноническим формам.

1. **Сформулируйте понятия эквивалентности.**

Понятие эквивалентности относится к отношению между двумя лямбда-термами, которые могут быть преобразованы друг в друга через серию преобразований, не изменяя их семантического значения. Это означает, что два лямбда-терма считаются эквивалентными, если они могут быть приведены к одному и тому же виду с использованием процессов подстановки, альфа-редукции, β-редукции и η-редукции.

1. **Сформулируйте теорему Черча-Россера и два следствия из нее.**

**Формулировка теоремы Черча-Россера:**

Для любого алгоритма, который может быть реализован на машине Тьюринга, существует соответствующий лямбда-терм, который вычисляет то же самое выражение.

**Следствия из теоремы Черча-Россера:**

Следствие о полноте: поскольку любой алгоритм, который может быть реализован на машине Тьюринга, может быть выражен в виде лямбда-терма, лямбда-исчисление является полным в смысле вычислений, то есть оно может выразить любую вычислимую функцию.

**Следствие о неразрешимости**: если существует лямбда-терм, который вычисляет функцию, то существует алгоритм, который может быть реализован на машине Тьюринга для вычисления этой функции. Это следуствие из того, что если лямбда-терм может выразить функцию, то существует способ его вычисления, который может быть адаптирован для работы на машине Тьюринга.

1. **Сформулируйте и докажите следствия из теоремы Черча-Россера**

**Следствие 1: Полнота лямбда-исчисления**

Полнота означает, что если выражение может быть вычислено, то оно может быть выражено в лямбда-исчислении.

**Доказательство**: предположим, что у нас есть алгоритм, который может быть реализован на машине Тьюринга. Этот алгоритм может быть представлен в виде последовательности команд, которые машина Тьюринга выполняет для вычисления выражения. Мы можем представить каждую команду машины Тьюринга как лямбда-терм, который выполняет соответствующую операцию. Затем мы можем комбинировать эти лямбда-термы в один большой лямбда-терм, который представляет весь алгоритм. Этот лямбда-терм будет вычислять то же самое выражение, что и алгоритм на машине Тьюринга. Таким образом, если выражение может быть вычислено, оно может быть выражено в лямбда-исчислении.

**Следствие 2: Неразрешимость проблемы остановки**

Неразрешимость означает, что не существует алгоритма, который может определить, остановится ли машина Тьюринга на любом входном наборе данных или будет выполняться бесконечно.

**Доказательство**: предположим, что существует алгоритм, который может определить, остановится ли машина Тьюринга на любом входном наборе данных. Этот алгоритм может быть представлен в виде лямбда-терма. Однако, по теореме Черча-Россера, если лямбда-терм может выразить функцию, то существует алгоритм, который может быть реализован на машине Тьюринга для вычисления этой функции. Таким образом, если существует алгоритм, который может определить, остановится ли машина Тьюринга, то существует лямбда-терм, который вычисляет эту функцию. Это противоречит неразрешимости проблемы остановки, поскольку это означало бы, что проблема остановки может быть решена, что является известным неверным утверждением.

1. **Сформулируйте и докажите лемму о комбинаторах I, K, S**

Для любого лямбда-терма M, который не содержит свободных переменных, существует лямбда-терм, который использует только комбинаторы I, K, S, и который вычисляет M.

**Доказательство леммы о комбинаторах I, K, S:**

**Комбинатор I**: I = \x.x. Комбинатор I является идентичностью, который просто возвращает свой аргумент.

**Комбинатор K**: K = \x.\y.x. Комбинатор K возвращает свой первый аргумент, игнорируя второй.

**Комбинатор S**: S = \x.\y.\z.x z (y z). Комбинатор S применяет свой первый аргумент к результату применения своего второго аргумента к третьему.

**Доказательство:**

Для доказательства леммы о комбинаторах I, K, S, рассмотрим структуру лямбда-терма M. Мы можем разбить M на компоненты, которые могут быть выражены с использованием комбинаторов I, K, S.

**Базовый случай**: Если M является простым выражением, например, переменной или константой, то M может быть выражен напрямую с использованием комбинаторов I, K, S.

**Рекурсивный случай**: Если M является более сложным выражением, например, применением функции к аргументу, то мы можем разложить M на компоненты и рекурсивно применить лемму к каждому из этих компонентов.

**Для** любого лямбда-терма M, который не содержит свободных переменных, мы можем построить соответствующий лямбда-терм, используя комбинаторы I, K, S, и который вычисляет M. Это доказывает лемму о комбинаторах I, K, S.

1. **Докажите, что любой терм представим в виде комбинаторов S K**

Для любого лямбда-терма M, который не содержит свободных переменных, существует лямбда-терм, который использует только комбинаторы S и K, и который вычисляет M.

**Доказательство:**

**Базовый случай**: Если M является простым выражением, например, переменной или константой, то M может быть выражен напрямую с использованием комбинаторов S и K. Например, если M является константой c, то M может быть выражено как K c, где K возвращает свой первый аргумент.

**Рекурсивный случай**: Если M является более сложным выражением, например, применением функции к аргументу, то мы можем разложить M на компоненты и рекурсивно применить доказательство к каждому из этих компонентов.

1. **Понятие и виды комбинаторов**

**Комбинатор I** (идентичность): I = \x.x. Комбинатор I возвращает свой аргумент без изменений. Он является самым простым комбинатором и служит базовым строительным блоком для создания других комбинаторов.

**Комбинатор K** (константа): K = \x.\y.x. Комбинатор K игнорирует свой второй аргумент и возвращает свой первый аргумент. Он используется для создания функций, которые возвращают константу.

**Комбинатор S** (сумма): S = \x.\y.\z.x z (y z). Комбинатор S применяет свой первый аргумент к результату применения своего второго аргумента к третьему. Он используется для создания функций, которые могут выполнять сложные операции, такие как применение функции к результату другой функции.

**Комбинатор B** (условие): B = \x.\y.\z.x z y. Комбинатор B применяет свой первый аргумент к результату применения своего второго аргумента к третьему, если третий аргумент истинен, иначе применяет второй аргумент к третьему. Он используется для создания условных выражений.

**Комбинатор C** (составление): C = \x.\y.\z.x (y z). Комбинатор C применяет свой первый аргумент к результату применения своего второго аргумента к третьему. Он используется для создания функций, которые могут выполнять последовательное применение функций.

1. **Числа Черча. Операция плюс 1**

Число n в числах Черча представляется как функция, которая принимает два аргумента f и x и применяет f к x n раз. Формально, число n можно записать как n = \f.\x.f^n x, где ^ обозначает операцию возведения в степень. Однако, поскольку в лямбда-исчислении нет встроенной операции возведения в степень, мы используем композицию функций для достижения того же эффекта. ОПЕРАЦИЯ ПЛЮС 1 – это ЗАПИСЬ ЧИСЛА 1, 2, 3 в лямбда виде

1. **Арифметические операции над числами Черча + \* ^**

Операция сложения для чисел Черча позволяет сложить два числа. Функция сложения Add принимает два числа m и n и возвращает число, равное их сумме.

Операция умножения для чисел Черча позволяет умножить два числа. Функция умножения Mult принимает два числа m и n и возвращает число, равное их произведению.

Операция возведения в степень для чисел Черча позволяет возвести число в степень. Функция возведения в степень Deg принимает число m и число степеней n и возвращает число, равное m в степени n.

1. **Разность чисел Черча**

Для вычитания числа m из числа n в числах Черча, мы можем использовать концепцию префиксного инкремента. Префиксный инкремент числа n на m равен числу n + m. Таким образом, чтобы вычесть m из n, мы можем преобразовать задачу вычитания в задачу инкремента, вычитая m из n и затем применяя префиксный инкремент к результату.

1. **Булевы константы и оператор if**

В лямбда-исчислении булевы константы обычно представляются как функции, которые возвращают себя или другую функцию в зависимости от их аргументов. Например, булева константа true может быть представлена как функция, которая всегда возвращает себя, а false может быть представлена как функция, которая всегда возвращает другой аргумент.

Оператор if в лямбда-исчислении позволяет выполнять одно из двух выражений в зависимости от булевого значения. Он может быть представлен как функция, которая принимает три аргумента: условие, выражение, выполняемое при истинном условии, и выражение, выполняемое при ложном условии.

1. **Реализация булевых операций.**

True ≡ λ xy.x

False ≡ λ xy.y

And ≡ λ pq.pq False

Or ≡ λ pq.p True q

Not ≡ λ p.p False True

IfThenElse ≡ λ pqr. Pqr

1. **Кортежи. Каррирование.**

Кортежи в лямбда-исчислении могут быть представлены как функции, которые принимают два аргумента: один для хранения первого элемента кортежа, а другой для хранения второго элемента. Например, кортеж из двух элементов может быть представлен как функция, которая принимает два аргумента и возвращает функцию, которая принимает два аргумента и возвращает первый из них.

Каррирование функции преобразует функцию с несколькими аргументами в последовательность функций, каждая из которых принимает один аргумент. Это позволяет легко создавать частично примененные функции.

1. **Комбинатор неподвижной точки**

Комбинатор неподвижной точки является одним из ключевых комбинаторов в лямбда-исчислении. Он позволяет определить рекурсивные функции и используется для создания функций, которые могут вызывать сами себя.

Комбинатор T работает, создавая функцию, которая принимает функцию f и возвращает результат применения f к самой себе. Это достигается путем использования лямбда-выражения, которое принимает два аргумента x и y, и возвращает результат применения x к y. Это позволяет функции f вызывать саму себя, создавая рекурсивный процесс.