ЛЯМБДА

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

Парадигма программирования – совокупность идей и понятий, которые определяют общий стиль написания компьютерных программ, построения их структуры и отдельных элементов программной системы.

Функциональное программирование – парадигма, согласно которой процесс исполнения программы представляется последовательностью вычислений значений для математических функций (основана на чистых функциях). Программы на функциональных языках точнее соответствуют математическим объектам, и их свойства легче доказывать. Поскольку вычисление выражений не имеет побочных эффектов для любых состояний, то отдельные подвыражения могут вычисляться в произвольном порядке, не влияя друг на друга. Это означает, что функциональные программы хорошо поддаются распараллеливанию.

Основные принципы ФП включают: Чистые функции, Иммутабельность (Данные не изменяются после их создания. Вместо изменения создаются новые объекты с новыми значениями), ФВП, Ленивые вычисления, Функциональная композиция (Объединение простых функций для создания более сложных).

+ Чистые функции и иммутабельность делают тестирование и отладку проще. Благодаря отсутствию изменяемого состояния, параллельные вычисления проще и безопаснее. Функции высшего порядка и композиция функций способствуют созданию модульного и повторно используемого кода. Чистые функции всегда ведут себя одинаково при одинаковых входных данных, что упрощает понимание и предсказание поведения программы.

- Из-за иммутабельности могут потребоваться дополнительные ресурсы для копирования данных. ФП может быть сложным для изучения. В реальных приложениях часто приходится работать с состоянием и побочными эффектами, что может усложнить использование чисто функциональных подходов.

**Функция высших порядков** — это функция, которая принимает другие функции в качестве аргументов или возвращает функцию в качестве результата. fun hOrderFunc(a:Int, f:(Int) -> Int):Int = f(a)

**Чистая функция** — это функция, которая не изменяет состояния программы вне себя. Это означает, что функция не изменяет значения глобальных переменных, не выполняет ввода/вывода, не изменяет аргументы, не работает с внешними состояниями и не взаимодействует с другими функциями или методами, которые изменяют состояние. Результат функции зависит только от ее входных параметров и не зависит от какого-либо внешнего состояния. fun add(a:Int, b: Int): Int{ return a + b}

**Каррирование** позволяет превращать функцию от нескольких аргументов в цепочку функций от одного аргумента. Пример: fun add(a: Int): (Int) -> Int { return { b -> a + b } }

**Ленивые вычисления** — вычисления откладываются до тех пор, пока результат не потребуется.

1. 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 лямбда выражении.

λ-исчисление сформировано Алонзо Черчем как теория исчисления

λ-исчисление является теоретическим базисом для функционального программирования.

Математические предположения:

Функции как объекты: функции рассматриваются как объекты, которые можно представить другим функциям в качестве аргументов.

Применение функции: функции могут применяться к аргументам

Анонимные функции: функции без имени.

Лямбда-термом называется:

• выражение, состоящее из одной переменной,

• константы, их количество определяется конкретной λ-нотацией, иногда их нет вовсе.

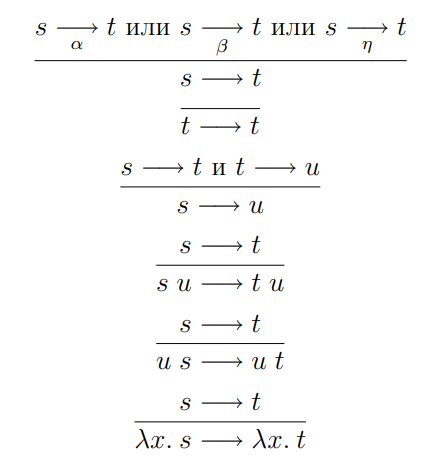
• выражение вида (MN), называемое аппликацией (применение), где M и N — λ-термы,

• выражение вида (λx.N), называемое λ-абстракцией, где N — λ-терм, x — переменная.

Для упрощения λ-выражения скобки можно опустить, если это не приводит к неоднозначности. λx1x2x3 . . . xn.M ≡ (λx1.(λx2.(λx3 . . . (λxn.M) . . .)))

1. Дайте определения редукции лямбда термов. Опишите стратегии редукции лямбда термов.

Редукция - процедура вычисления терма, в ходе которой последовательно вычисляются комбинации вида f(x), где f — λ-абстракция.



β-редукция: применение абстракций и замена связанных переменных фактическими значениями. Например: (λx.(\* x x) 2) → (\* 2 2) , (λx.x M) → M . Если на некотором этапе оказывается, что не могут быть применены никакие правила редукции, кроме α-преобразований, то говорят, что терм имеет нормальную форму.

Стратегии:

Аппликативный порядок редукций (АПР) предписывает всегда выбирать самый левый из внутренних редексов.

Нормальный порядок редукций (НПР) предписывает всегда выбирать самый левый из внешних редексов. Стратегия НПР откладывает вычисление аргумента до тех пор, пока он действительно не потребуется.

Теорема. Если справедливо s → t, где терм t имеет нормальную форму, то последовательность редукций, которая начинается с терма s и состоит в применении правил редукции к самому левому редексу, всегда завершается и приводит к терму в нормальной форме. (Лучше использовать НПР).

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

В терме вида (λx.N) терм N называется областью действия λ-абстрактора по переменной x.

Вхождение переменной в терм называется свободным в рассматриваемом терме, если оно не находится в области действия λ-абстрактора по этой переменной.

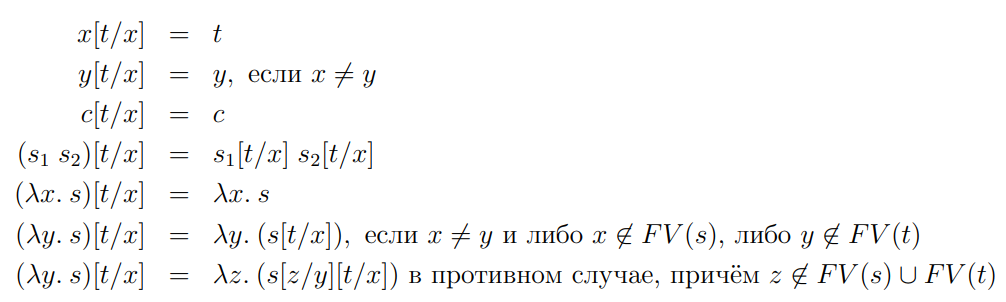
Множество свободных переменных в терме s обозначается F V (s), его рекурсивное определение: FV(x) = {x}; FV(c) = ∅; FV(s t) = FV(s) ∪ FV (t); FV(λx.s) = FV(s) − {x}.

Множество связанных переменных BV(s): BV(x) = ∅; BV(c) = ∅; BV(s t) = BV(s) ∪ BV(t); BV(λx.s) = BV(s) ∪ {x} Например, если s = (λx y.x) (λx.z x), то FV(s) = {z} и BV(s) = {x, y}.

Теорема. Для произвольного λ-терма s множество FV(s) конечно.

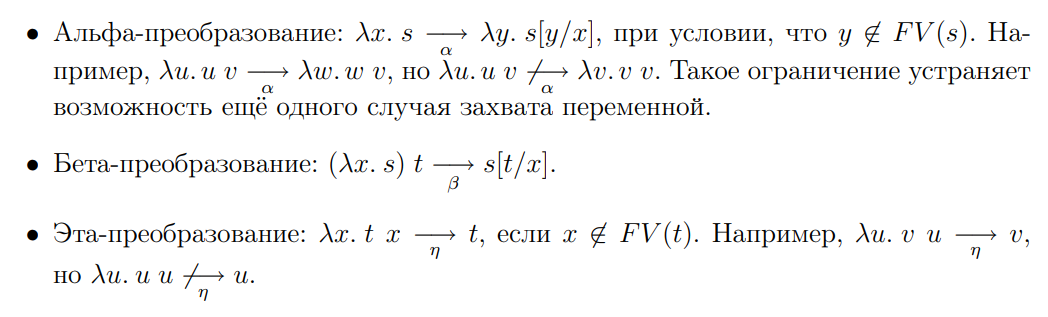
1. Дайте понятия подстановки и преобразования. Сформулируйте понятия эквивалентности.

**Подстановка** в лямбда-исчислении означает замену свободной переменной в лямбда-выражении другим лямбда-выражением. Операция подстановки терма s вместо переменной x в другой терм t обозначается t[s/x]. Рекуррентное определение понятия подстановки выглядит так:



Чтобы не терялся смысл исходного терма после подстановки необходимо предварительно переименовать связанную переменную. Пример: M=λx.(x y) N=z, тогда подстановка даст λx.(x z).

**Редукция (или преобразование)** в лямбда-исчислении означает процесс упрощения или вычисления выражений.



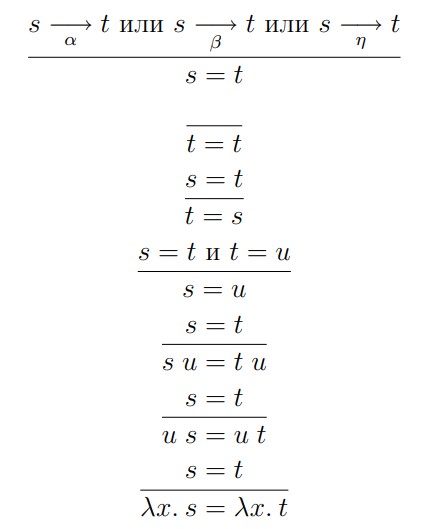
**α-преобразование** – переименование связанных переменных.

**β-преобразование** – вычисление функции для заданного аргумента.

**η-преобразование** – разновидность экстенсиональности (две функции f и g считаются равными, если для любого аргумента x, f(x) и g(x) дают один и тот же результат). Пусть f = λy.(y+1). Тогда η-преобразование утверждает, что: λx.((λy.(y+1)) x) ≡ λy.(y+1). Конгруэнтное замыкание этих преобразований означает, что мы рассматриваем не только сами преобразования, но и их применение в любом контексте. Это замыкание определяет отношение эквивалентности, называемое λ-эквивалентностью. Если два терма равны (α-эквивалентны, β-эквивалентны или η-эквивалентны), то они остаются равными в любом контексте.

Два терма **эквивалентны**, если один из них может быть получен из другого в ходе конечной последовательности α, β либо η-преобразований, которые применяются к произвольным подтермам как в прямом, так и в обратном направлении. Другими словами, отношение λ-эквивалентности представляет собой конгруэнтное замыкание трёх преобразований и обладает свойствами рефлексивности (M≡M), симметричности (M≡N⟹N≡M), транзитивности (M≡N⟹N≡P⟹M≡P) и заменяемости (C[M]≡C[N]). Для термов P и Q запись P = Q означает, что P →βQ или Q→βP.

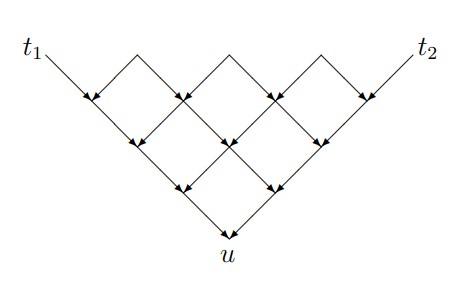
Индуктивное определение: если утверждение над горизонтальной чертой выполняется, то справедливо и утверждение под ней.



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

Теорема Черча-Россера (о ромбическом свойстве β-редукции). Если t→βs1 и t→βs2, то существует терм u такой, что s1→βu и s2→βu.

Следствие 1) Если t1 = t2 то найдётся терм u такой, что t1→βu и t2→βu. Доказательство: Легко показать (при помощи структурной индукции), что отношение λ-равенства = представляет собой симметричное транзитивное замыкание отношения редукции. Дальнейшее следует по индукции согласно свойствам симметричного транзитивного замыкания.

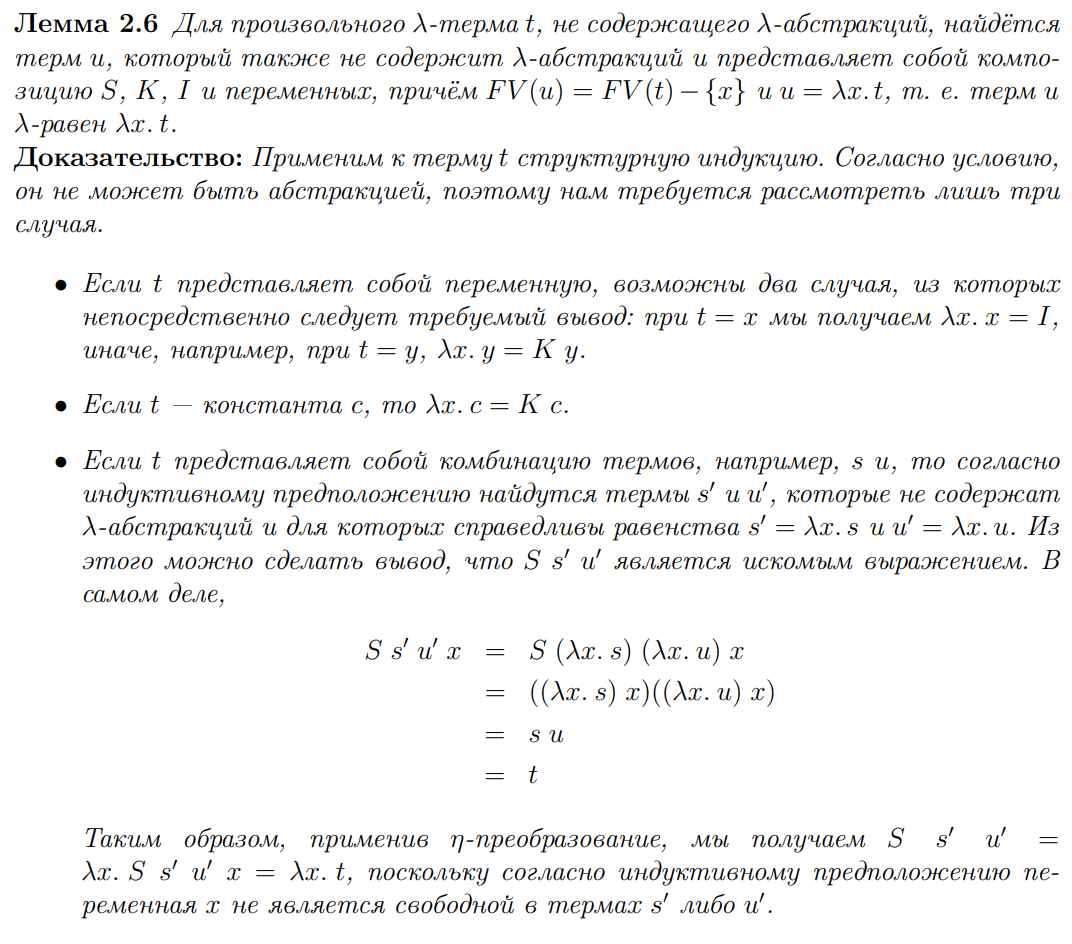


Мы полагаем, что t1 = t2, т. е. существует некоторая последовательность редукций в обеих направлениях (зигзагообразная линия в верхней части рисунка), которая их объединяет. Теорема Чёрча-Россера позволяет нам заполнить недостающие участки на краях диаграммы, после чего требуемый результат достигается композицией этих редукций.

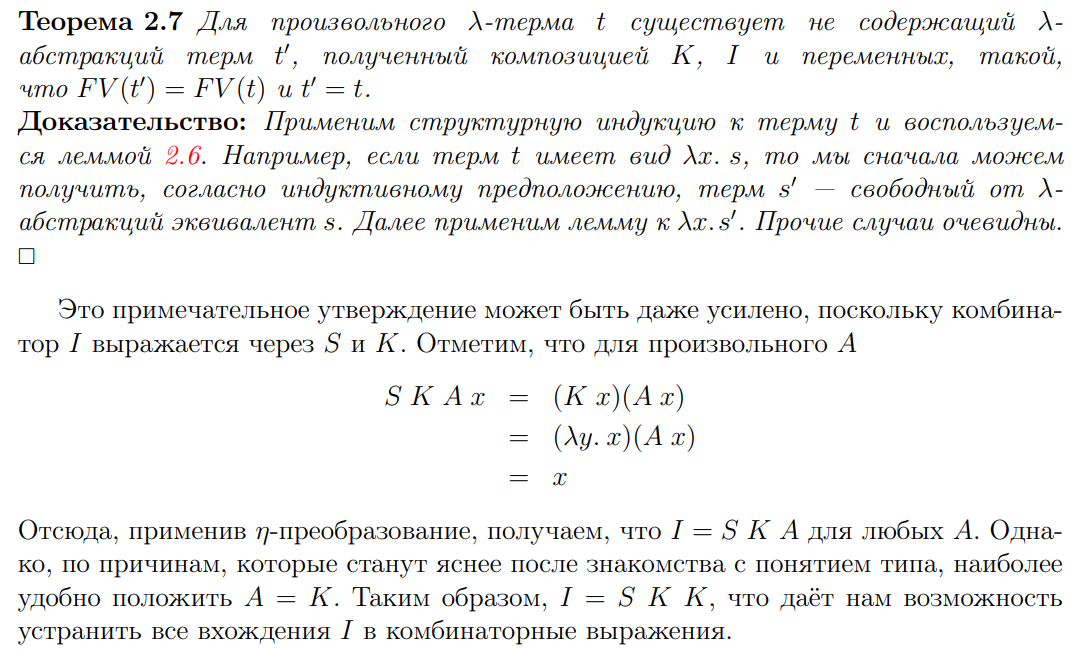
Следствие 2) Если t = t1 и t = t2, причём t1 и t2 имеют нормальную форму, то t1 ≡αt2, т. е. t1 и t2 равны с точностью до α-преобразований. Доказательство: Согласно изложенному выше, найдётся некоторый терм u такой, что t1 → u и t2 → u. Но так как t1 и t2 уже имеют нормальную форму, последовательность редукций, приводящая к терму u, может состоять лишь из α-преобразований.

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

Комбинатор — это терм без свободных переменных. Множество всех комбинаторов обозначим через λ0. Такие термы также принято называть замкнутыми, поскольку их значение не зависит от значений каких-либо переменных. Произвольный терм может быть выражен при помощи определённого множества комбинаторов и всевозможных переменных, операция λ-абстракции становится ненужной. В частности, замкнутый терм может быть представлен исключительно через комбинаторы: I=λx.x (тождественный); K=λxy.x (канцелятор); S=λfgx.(fx)(gx) (коннектор).



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



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

Для представления чисел 0, 1, 2, . . . , n . . . существуют комбинаторы, называемые нумералами Черча, которые обозначаются следующим образом:

0 ≡ λfx.x

1 ≡ λfx.fx

2 ≡ λfx.f(fx)

3 ≡ λfx.f(f(fx))

. . . . . . . . . . . . . . . . . .

n ≡ λfx.fnx

. . . . . . . . . . . . . . . . . .

Здесь f1x ≡ fx и fnx ≡ f(fn-1x)

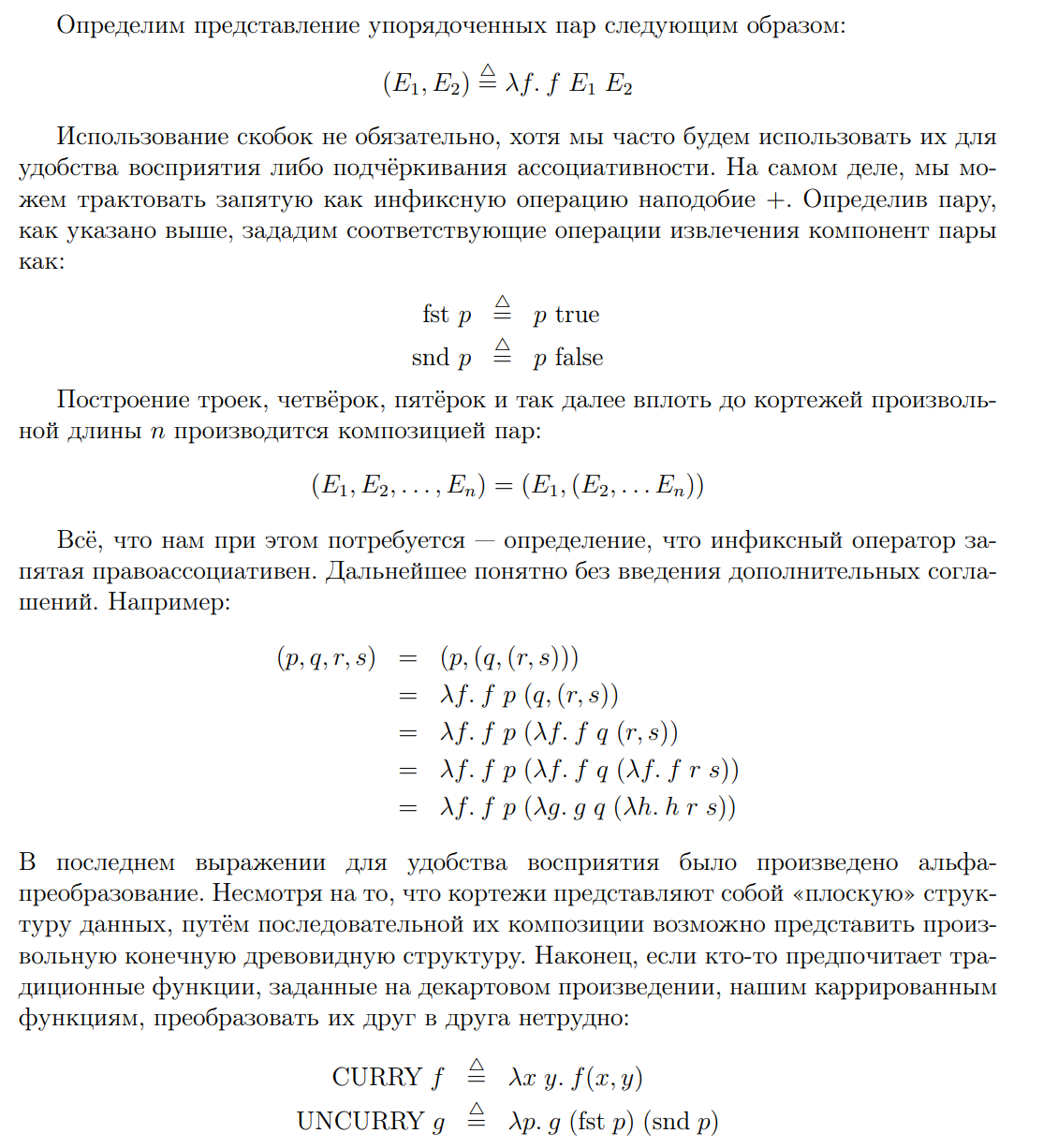
Нумерал n можно рассматривать как функцию, которая принимает на свой вход функцию f в качестве аргумента и возвращает n-кратную суперпозицию функции f.

Inc – комбинатор n+1: Inc ≡ λnfx.f(nfx).

Комбинатор, вычисляющий n-1 при n > 0: Dec ≡ λnfx.n(λgh.h(gf))(λu.x)(λu.u)

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

Каррирование – это способ представления функции с несколькими аргументами. Например, мы трактуем выражение λxy.x + y таким образом, что функция получает аргументы по очереди, по одному, а не все сразу. (λxy.x + y) 1 2 = (λy.1 + y) 2 = 1 + 2. Каррирование позволяет преобразовать функцию, которая принимает несколько аргументов одновременно, в цепочку функций, каждая из которых принимает один аргумент.



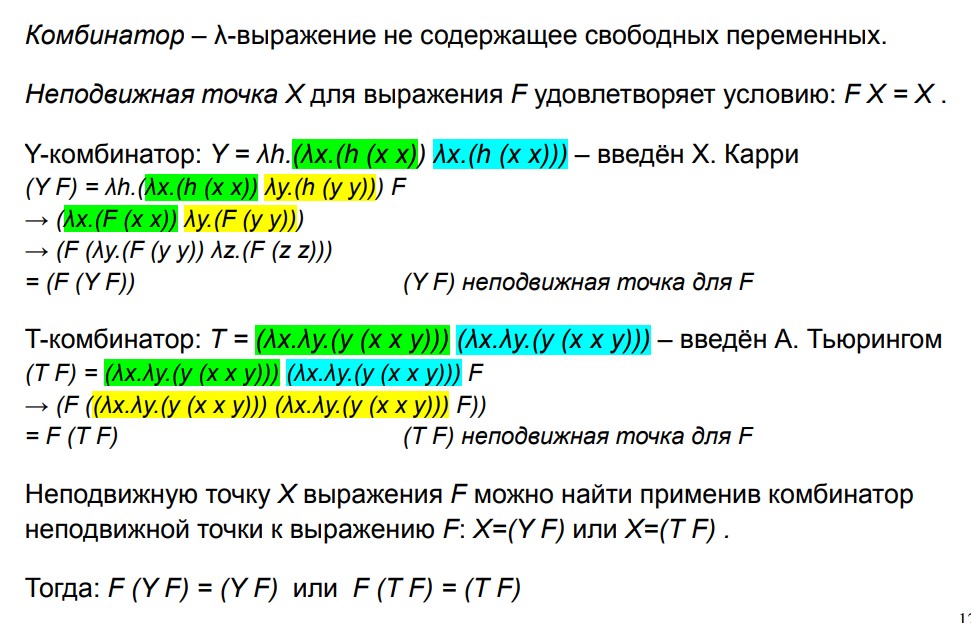
Рассмотрим функцию f(x,y)=x+y: Некаррированная версия: f(x,y)=x+y Каррированная версия: curried = λx.(λy.(x+y)).

Работа функции CURRY f=λx.(λy.(x+y)).

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

Теорема (о неподвижной точке в λ-исчислении).

(a) Для любого λ-терма F существует такой λ-терм X, что F X = X. Такой терм X называется неподвижной точкой F.

(б) Более того, существует такой комбинатор Y, что для любого λ-терма F терм (YF) является неподвижной точкой F, т.е., F(YF) = YF. (Такой комбинатор Y называют комбинатором неподвижной точки.)

1. Арифметические операции над числами Черча + \* ^

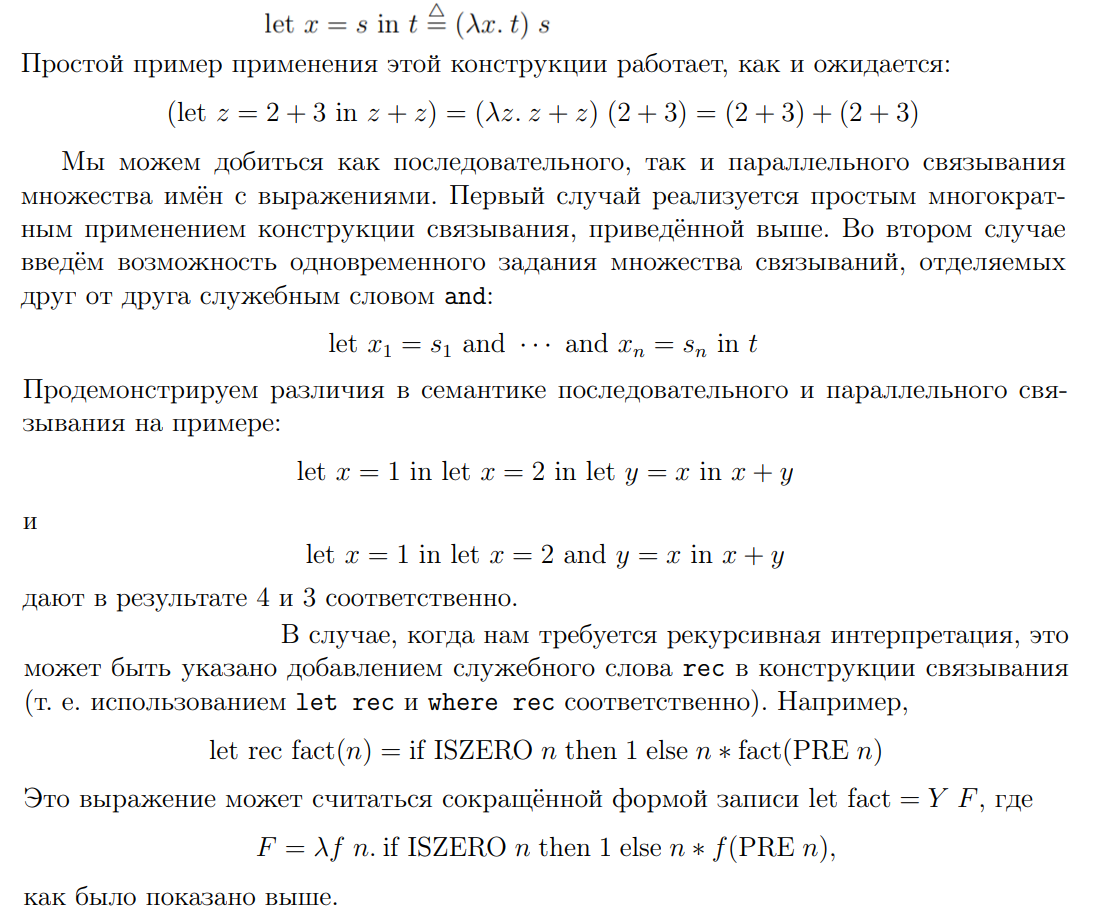
Add ≡ λmnfx.nf(mfx) +

Mult ≡ λmnf.m(nf) \*

Deg ≡ λmn.nm ^

1. Let выражения

**let-выражения** используются для связывания выражений с именами.



1. Булевы константы и оператор if. Реализация булевых операций

Для представления констант существуют комбинаторы:

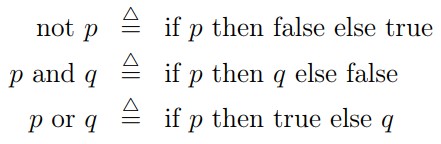
true ≡ λxy.x

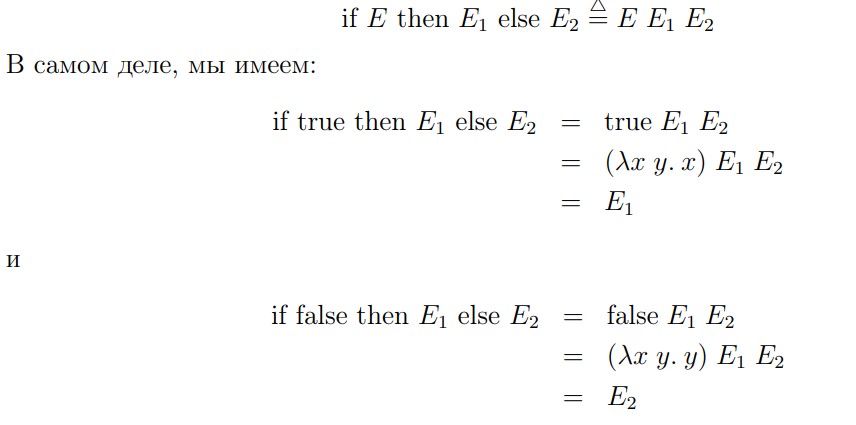
false ≡ λxy.y

Для представления конъюнкции, дизъюнкции и отрицания существуют соответственно комбинаторы:

And ≡ λ pq.pq false

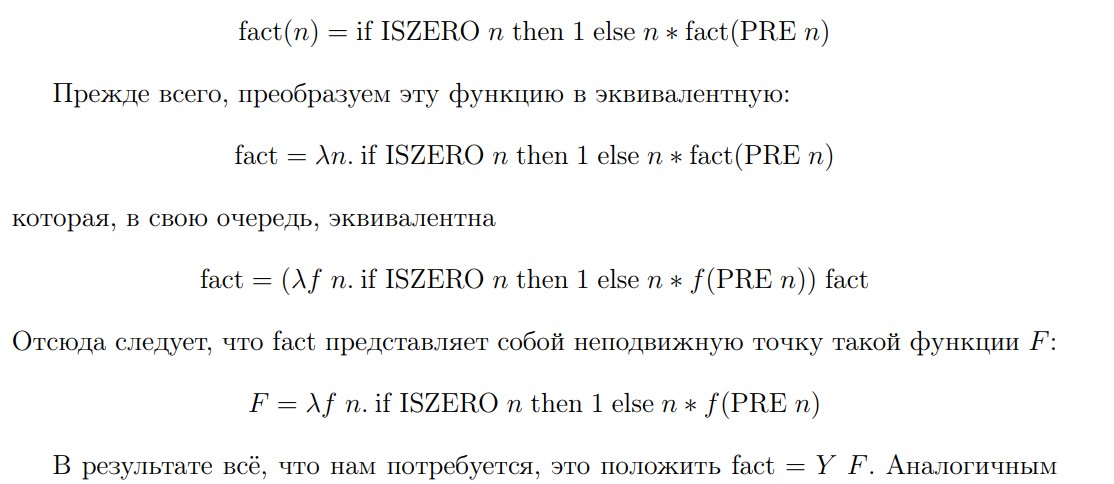
Or ≡ λ pq.p true q

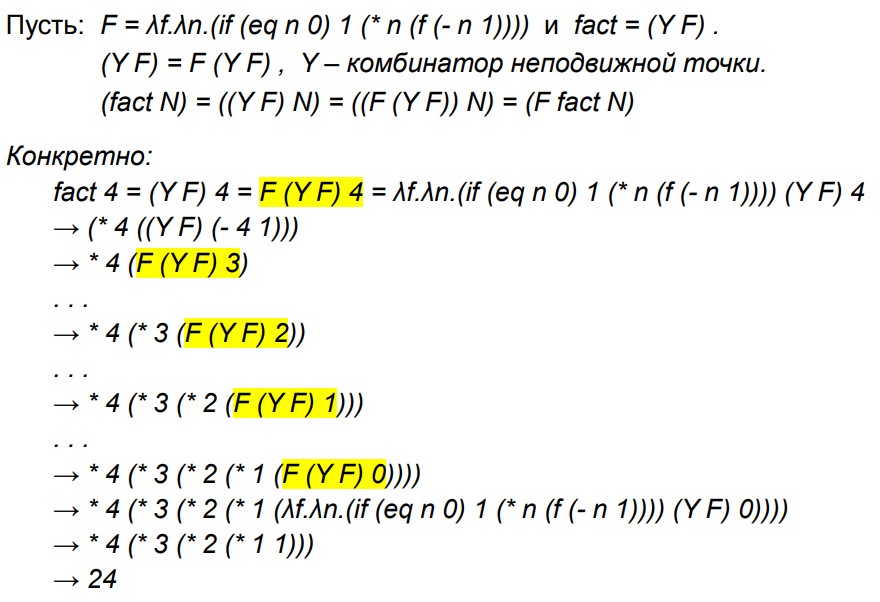
Not ≡ λ p.p false true



1. Рекурсивные функции (на примере любой функции)

Рекурсивные функции возможны благодаря комбинатору неподвижной точки. Рассмотрим, как комбинатор неподвижной точки (например, Y) может применяться для реализации рекурсии. Воспользуемся в качестве примера вычислением факториала. Мы хотим определить функцию fact следующим образом:





1. Полнота лямбда исчисления по Тьюрингу

**Тьюринг-полнота** — это свойство вычислительной системы, означающее, что эта система может вычислить любую функцию, которая может быть вычислена любой другой вычислительной системой, при наличии достаточного времени и памяти.

Вычислительную модель, эквивалентную по своей полноте машине Тьюринга, принято называть полной по Тьюрингу либо Тьюринг-полной.

Лямбда-исчисление является Тьюринг-полным, что означает, что оно способно выразить любую вычислимую функцию. Тьюринг впоследствии доказал, что любая вычислимая функция, которая может быть выражена в виде программы на универсальной машине Тьюринга, может быть также выражена в λ-исчислении.

1. Реализация списков Черча, head, tail.

Числа Черча могут быть обобщены и представлены в виде списков. Представим списки с помощью операции образования пар. (LET pair = λxyf.f x y (пара); LET fst p = p true (первый элемент); LET snd p = p false (следующий элемент))

Кодирование списков:

LET nil = λz. z (пустой список)

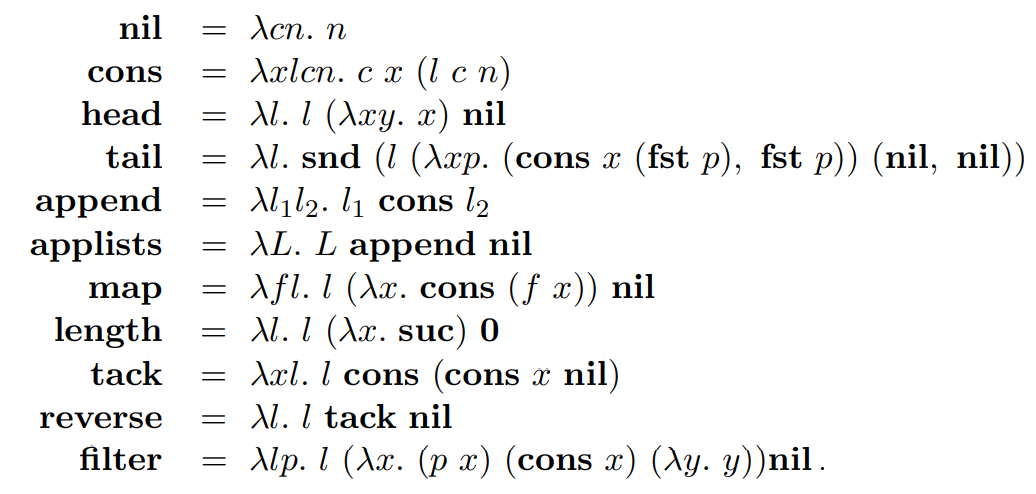
LET cons = λxy. pair false (pair xy) (создаёт новый список, добавляя элемент в голову существующего списка, где x голова, y – хвост)

LET null = fst (первый элемент пары)

LET hd = λz. fst (snd z) (получаем голову)

LET tl = λz. snd (snd z) (получаем хвост)

1. Покажите набор функций преобразования списков Чёрча.



**Определение списков, предложенное Маэрсоном. nil**: Представляет пустой список; **cons**: Создаёт новый список, добавляя элемент в голову существующего списка; **head**: Возвращает первый элемент списка (голову); **tail**: Возвращает все элементы списка, кроме первого (хвост); **append**: Соединяет два списка; **applists**: Объединяет список списков в один список; **map**: Применяет функцию ко всем элементам списка; **length**: Возвращает длину списка; **tack**: Добавляет элемент в голову списка; **reverse**: Разворачивает список; **filter**: Оставляет только те элементы списка, которые удовлетворяют предикату.

ПРОЛОГ

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

**Факт** определяет отношение (предикат) между объектами (аргументами), которых может быть любое количество. Факт всегда считается истинным утверждением.

Общая структура факта: имя\_факта(аргумент1… факт2(аргумент2)).

Пример: parent(svyatoslav,anna).

**Типы данных** (термы): переменные, константы: числа (вещественные и целые) и атомы; непеременные: списки, строки, структуры.

**Атом** – это простая, неделимая единица данных. Атомы могут представлять имена объектов, предикатов или любые другие константы. Например, anna, svyatoslav, parent.

Принцип работы Swi-Prolog:

Программа Prolog состоит из фактов и правил, которые описывают логические отношения между различными термами. Список фактов – называется базой данных. Пользователь задает вопросы системе, чтобы получить ответ или проверить истину утверждений. Унификация — это процесс сопоставления термов и подстановки переменных, чтобы сделать термы одинаковыми. При обработке запросов Prolog использует унификацию для сопоставления запросов с фактами и правилами в базе данных. Prolog применяет правила и пытается доказать запрос путем поиска совпадения с базой данных. Если текущий путь поиска не приводит к решению, Prolog использует механизм бектрекинга для поиска альтернативных решений. Также бектрекинг используется для поиска альтернативных решений по инициативе пользователя, с помощью ввода ;. Порядок найденных решений соответствует порядку расположения в программе подтверждающих фактов.

…

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

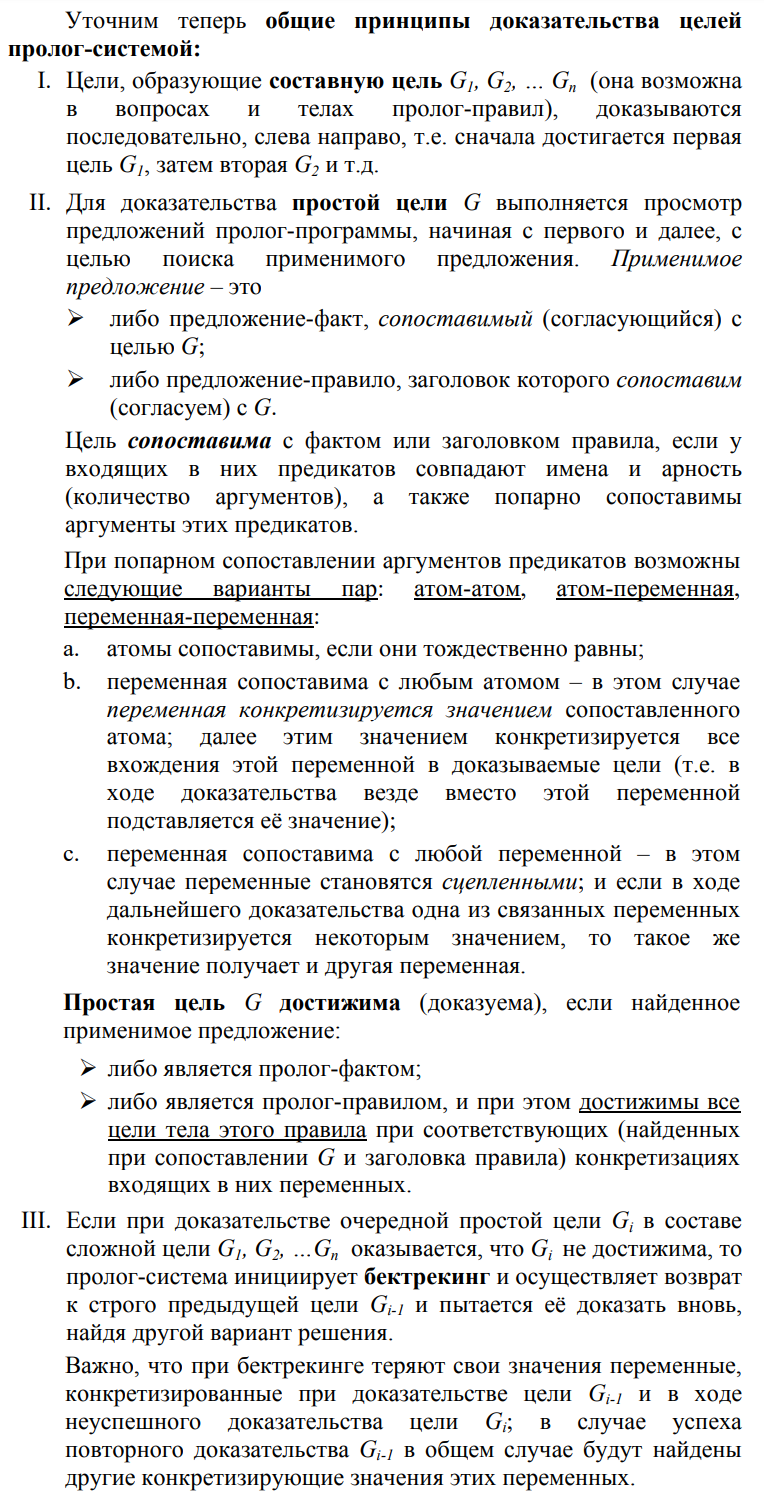
Унификация - это процесс нахождения значений для переменных таким образом, чтобы два терма стали одинаковыми (или совместимыми). Когда присваиваем значение переменной – конкретизируем ее. Примеры:  
Допустим, у нас есть два терма: X = 5 и Y = 5. В данном случае унификация приводит к тому, что переменные X и Y могут быть заменены на значение 5.  
Рассмотрим термы: father(john, mary) и father(john, X). Здесь унификация позволяет заменить переменную X на значение mary для того, чтобы оба терма стали эквивалентными.

**Предикаты** в Prolog представляют собой утверждения, которые считаются истинными. Например: отец(джон, питер). Отец в данном случае – это предикат, который отображает отношение, всегда записывается с маленькой буквы.

Правила используются в случае, если надо сказать, что некоторый факт зависит от группы других фактов.

Например: отношение “быть внуком или внучкой” можно записать в виде правила grandchild(X, Z) :- parent(Y, X), parent(Z, Y). Где X – внук или внучка, а Z – бабушка или дедушка. Часть после логической импликации (:-) называется телом правила. Если данные условия выполняются значит заголовок правила является верным.

Факт является всегда истинным, а правило только в том случае, если выполняются все цели в его правой части.



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

В рекурсии вверх результаты вычислений накапливаются после возврата из рекурсивного вызова. Это классический подход, где сначала происходит рекурсивный вызов, а затем, по мере возвращения, происходит вычисление.

fact(0, 1). fact(N, X) :- N > 0, N1 is N - 1, fact(N1, X1), X is X1 \* N.

В данном примере:

1. Если N = 0, то результат X = 1 (базовый случай).
2. Для N > 0:
   * Рекурсивный вызов fact(N1, X1) вычисляет факториал для N-1.
   * После возврата из рекурсивного вызова результат X1 умножается на N для получения конечного результата X.

Этот подход можно представить как подъем по стеку вызовов, накопление промежуточных результатов и их использование при возврате.

В рекурсии вниз промежуточные результаты накапливаются во время рекурсивных вызовов. Результаты передаются от одного вызова к другому до достижения базового случая. Это может быть более эффективным, поскольку промежуточные результаты передаются явно.

fact2(0, X, X). fact2(N, A, X) :- N > 0, N1 is N - 1, A1 is A \* N, fact2(N1, A1, X). call\_fact2(N, X) :- fact2(N, 1, X).

Аккумулятор A используется для накопления промежуточного результата, и в конечном итоге, когда N становится 0, аккумулятор содержит результат вычисления.

В данном примере:

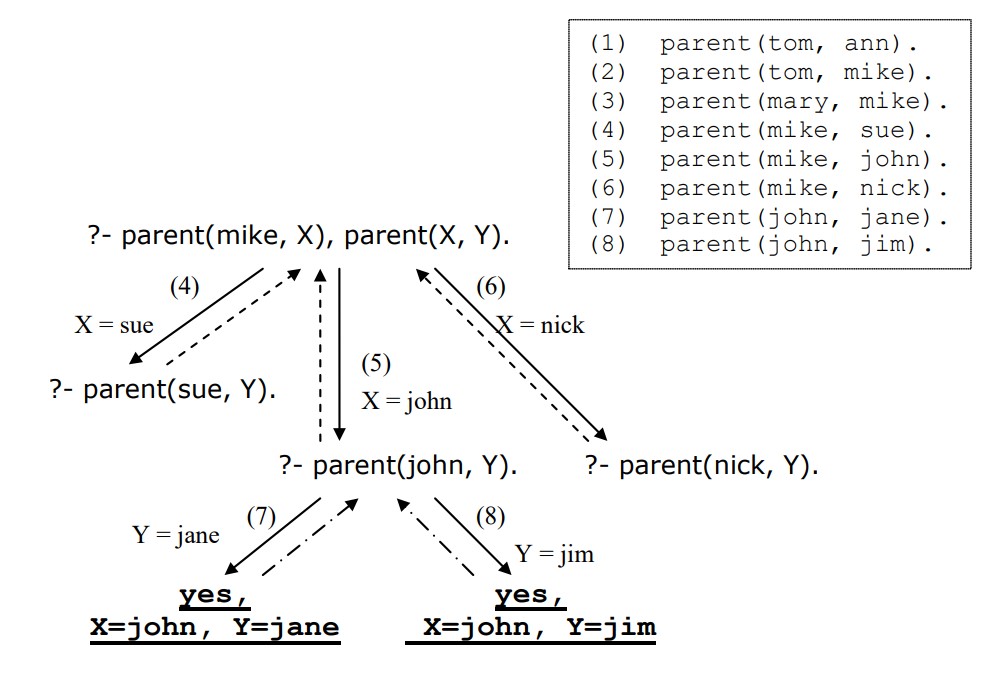
1. Базовый случай: когда N = 0, промежуточный результат A становится окончательным результатом X.
2. Для N > 0:
   * Рекурсивный вызов fact2(N1, A1, X) вычисляет факториал для N-1 с учетом накопленного результата A1.
   * Промежуточный результат A обновляется на каждом шаге (умножается на N) и передается дальше.

Этот подход можно представить как нисхождение по стеку вызовов с накоплением промежуточных результатов и их передачей вниз.

**Рекурсия вверх**: Промежуточные результаты накапливаются после возврата из рекурсивного вызова.

**Рекурсия вниз**: Промежуточные результаты накапливаются и передаются явно на каждом шаге рекурсивного вызова.

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



Бэктрекинг (backtracking) — это механизм поиска решений в Прологе. Когда Пролог сталкивается с точкой выбора (несколько возможных правил для одного и того же предиката), он выбирает первое правило. Если этот выбор приводит к неудаче, Пролог возвращается (откатывается) к точке выбора и пытается использовать следующее правило.

Сначала делаются неопределенными все переменные, которые были конкретизированы в ходе доказательства данного целевого утверждения. Затем возобновляется поиск в базе данных, начиная с того места, где был оставлен маркер. Если будет найдено другое утверждение, соответствующее целевому, Пролог помечает это место, и дальше продолжает поиск...

Пример:

parent(john, mary). parent(mary, susan). parent(john, george).

ancestor(X, Y) :- parent(X, Y).

ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).

?- ancestor(john, X).

Этот запрос будет решаться следующим образом:

1. Пролог сначала пытается найти правило, соответствующее ancestor(john, X).
2. Первое правило: ancestor(X, Y) :- parent(X, Y).
   * Подстановка: X = john, Y = mary (успех, X = mary).
3. Пролог запоминает точку выбора (второе правило) и продолжает поиск других решений.
4. Следующее правило: ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).
   * Подстановка: X = john, Z = mary, ancestor(mary, Y).
   * Пролог делает рекурсивный вызов: ancestor(mary, Y).
5. Пролог снова применяет первое правило: ancestor(mary, Y) :- parent(mary, Y).
   * Подстановка: Y = susan (успех, X = susan).
6. Пролог возвращается к точке выбора и ищет другие возможные значения для X (например, X = george), если такие имеются.

Оператор отсечения (!) используется для управления бэктрекингом. Он указывает Прологу прекратить бэктрекинг на текущем этапе, отбрасывая все альтернативные правила и подстановки, которые могли бы быть проверены позже.

Рассмотрим пример с предикатом, который определяет максимальное из двух чисел:

max(X, Y, X) :- X >= Y, !.

max(\_, Y, Y).

?- max(4, 5, Max).

Этот запрос будет решаться следующим образом:

1. Пролог начинает с первого правила: max(X, Y, X) :- X >= Y, !.
   * Подстановка: X = 4, Y = 5.
   * Условие 4 >= 5 ложно, поэтому это правило не применяется.
2. Пролог переходит ко второму правилу: max(\_, Y, Y).
   * Подстановка: Y = 4 (успех, Max = 5).

?- max(5, 4, Max).

Этот запрос будет решаться следующим образом:

1. Пролог начинает с первого правила: max(X, Y, X) :- X >= Y, !.
   * Подстановка: X = 5, Y = 4.
   * Условие 5 >= 4 истинно.
   * Пролог выполняет оператор !, который отсекает все остальные альтернативные правила.
   * Результат: Max = 5.

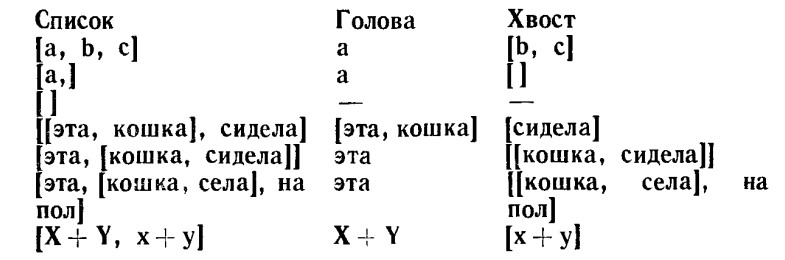
Без оператора отсечения, если бы условие X >= Y было истинно, Пролог все равно проверял бы следующее правило, что не всегда может быть желаемым поведением.

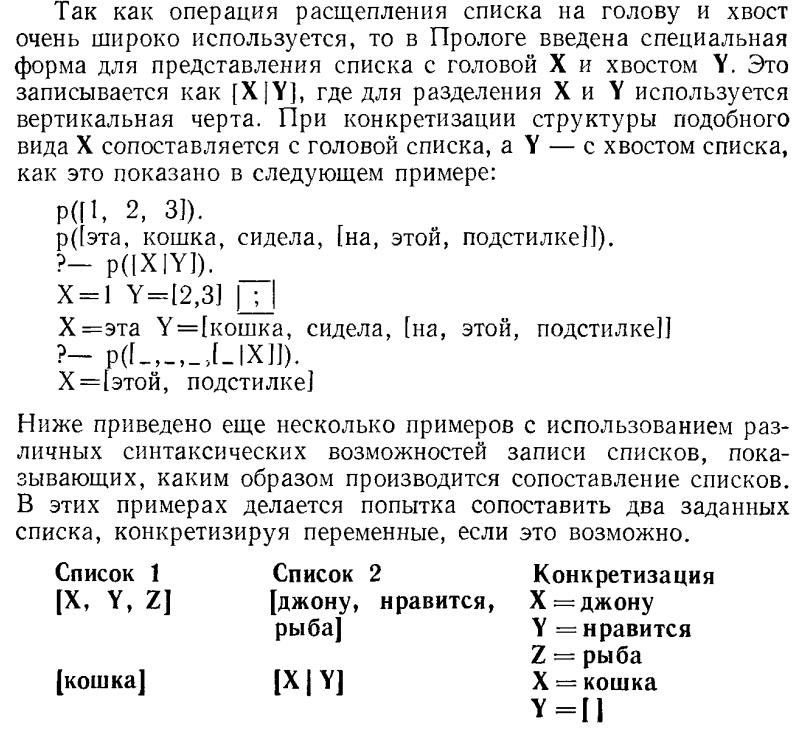
**Смысл применения оператора отсечения**

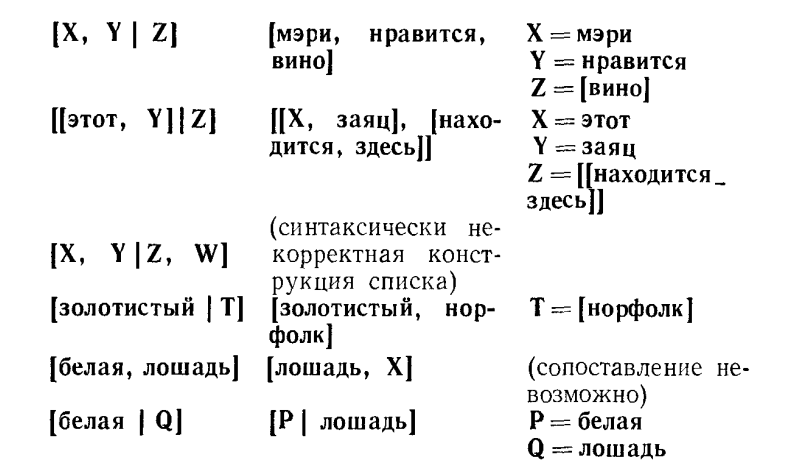
1. **Оптимизация производительности**: Оператор отсечения помогает избежать ненужных вычислений, что может значительно ускорить выполнение программы.
2. **Контроль логики программы**: В некоторых случаях мы знаем, что при выполнении определенного условия дальнейшие проверки не нужны. Отсечение позволяет явно это указать.
3. **Предотвращение неправильных решений**: В сложных программах отсечение помогает избежать неправильных решений, которые могли бы возникнуть при дальнейших проверках.
4. Опишите принцип работы со списками Черча в Swi-prolog. Покажите реализации встроенных предикатов работы со списками на основе механизма унификации (append, reverse, nth0).

Список – это упорядоченная последовательность элементов, которая может иметь произвольную длину. Элементами списка могут быть любые термы – константы, переменные, структуры, которые включают и другие списки.

Структура списка определяется рекурсивно: список является либо пустым, и обозначается [], либо состоит из двух компонент: первого элемента, называемого головой списка и остатка-хвоста, являющегося списком.







Рассмотрим реализацию некоторых встроенных предикатов на основе механизма унификации для обычных списков Пролога. Мы будем использовать стандартные списки Пролога, так как они удобнее для иллюстрации.

append([], L, L).

append([H|T], L, [H|R]) :- append(T, L, R).

?- append([1, 2], [3, 4], Result). % Result = [1, 2, 3, 4]

Предикат reverse/2 переворачивает список:

reverse(L, R) :- reverse(L, [], R).

reverse([], Acc, Acc).

reverse([H|T], Acc, R) :- reverse(T, [H|Acc], R).

?- reverse([1, 2, 3, 4], Result). % Result = [4, 3, 2, 1]

Предикат nth0/3 находит элемент по индексу (индексация с нуля):

nth0(0, [H|\_], H).

nth0(N, [\_|T], E) :- N > 0, N1 is N - 1,

nth0(N1, T, E).

?- nth0(2, [a, b, c, d], Result). % Result = c

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

В Прологе строки представляются как последовательности символов, заключенные в одинарные кавычки (') или двойные кавычки ("). Основные встроенные предикаты для работы со строками в Прологе включают:

1. atom\_chars/2: Преобразует атом в список символов и наоборот. Например:

prolog

Копировать код

?- atom\_chars('hello', L).

L = [h, e, l, l, o].

?- atom\_chars(Atom, [h, e, l, l, o]).

Atom = hello.

1. atom\_codes/2: Преобразует атом в список ASCII кодов символов и наоборот. Например:

prolog

Копировать код

?- atom\_codes('hello', L).

L = [104, 101, 108, 108, 111].

?- atom\_codes(Atom, [104, 101, 108, 108, 111]).

Atom = hello.

1. char\_code/2: Преобразует символ в его ASCII код и наоборот. Например:

prolog

Копировать код

?- char\_code('a', X).

X = 97.

?- char\_code(C, 97).

C = a.

1. string\_concat/3: Соединяет две строки. Например:

prolog

Копировать код

?- string\_concat('hello', 'world', Result).

Result = helloworld.

1. sub\_string/5: Извлекает подстроку из строки. Например:

prolog

Копировать код

?- sub\_string('hello world', 1, 5, \_, Substring).

Substring = hello.

1. string\_length/2: Возвращает длину строки. Например:

prolog

Копировать код

?- string\_length('hello', Length).

Length = 5.

1. string\_upper/2, string\_lower/2: Преобразуют строку к верхнему или нижнему регистру соответственно. Например:

prolog

Копировать код

?- string\_upper('hello', Upper).

Upper = 'HELLO'.

?- string\_lower('Hello', Lower).

Lower = 'hello'.

1. atomic\_list\_concat/3: Соединяет атомы из списка в строку, разделяя их указанным разделителем. Например:

prolog

Копировать код

?- atomic\_list\_concat([hello, world], ', ', Result).

Result = 'hello, world'.

Эти предикаты позволяют работать со строками в Прологе, выполняя операции, такие как разделение строк, конкатенация, изменение регистра и другие.

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

Пролог предоставляет мощные средства для работы с комбинаторными объектами, такими как перестановки, сочетания и размещения. Вот несколько примеров использования SWI-Prolog для работы с этими объектами:

1. **Перестановки**:

prolog

Копировать код

permutation([], []).

permutation(List, [X|Perm]) :-

select(X, List, Rest),

permutation(Rest, Perm).

Этот код генерирует все перестановки элементов списка. Например:

prolog

Копировать код

?- permutation([1,2,3], Perm).

Perm = [1, 2, 3] ;

Perm = [1, 3, 2] ;

Perm = [2, 1, 3] ;

Perm = [2, 3, 1] ;

Perm = [3, 1, 2] ;

Perm = [3, 2, 1] ;

false.

1. **Сочетания**:

prolog

Копировать код

combination(0, \_, []).

combination(N, [X|T], [X|Comb]) :-

N > 0,

N1 is N - 1,

combination(N1, T, Comb).

combination(N, [\_|T], Comb) :-

N > 0,

combination(N, T, Comb).

Этот код генерирует все сочетания из N элементов списка. Например:

prolog

Копировать код

?- combination(2, [1,2,3], Comb).

Comb = [1, 2] ;

Comb = [1, 3] ;

Comb = [2, 3] ;

false.

1. **Размещения**:

prolog

Копировать код

arrangement(0, \_, []).

arrangement(N, List, [X|Arr]) :-

N > 0,

select(X, List, Rest),

N1 is N - 1,

arrangement(N1, Rest, Arr).

Этот код генерирует все размещения из N элементов списка. Например:

prolog

Копировать код

?- arrangement(2, [1,2,3], Arr).

Arr = [1, 2] ;

Arr = [1, 3] ;

Arr = [2, 1] ;

Arr = [2, 3] ;

Arr = [3, 1] ;

Arr = [3, 2] ;

false.

Эти примеры демонстрируют, как в SWI-Prolog можно создавать и манипулировать комбинаторными объектами, что делает его мощным инструментом для решения комбинаторных задач.

1. Покажите основные принципы реализации переборных алгоритмов на графах средствами пролога.

Переборные алгоритмы на графах в Прологе основаны на использовании рекурсии и обходе графа в глубину (DFS) или в ширину (BFS). Вот основные принципы реализации переборных алгоритмов на графах в Прологе:

1. **Определение предикатов**:
   * Определите предикаты для работы с графами, например, предикат для проверки смежности вершин, предикаты для поиска путей и т. д.
2. **DFS (Обход в глубину)**:
   * Используйте рекурсивный подход для обхода графа в глубину.
   * Начните с одной вершины и рекурсивно переходите к смежным вершинам.
   * Используйте механизм отслеживания уже посещенных вершин, чтобы избежать зацикливания.
3. **BFS (Обход в ширину)**:
   * Используйте очередь для хранения вершин, которые еще предстоит посетить.
   * Начните с одной вершины и поочередно посещайте все смежные вершины, добавляя их в очередь.
   * Продолжайте обход, пока очередь не пуста.
4. **Поиск путей**:
   * Реализуйте предикат для поиска путей между двумя вершинами.
   * Используйте DFS или BFS для поиска путей в графе.
   * При рекурсивном обходе отслеживайте текущий путь и вершины, которые уже были посещены.
5. **Проверка связности**:
   * Реализуйте предикаты для проверки связности графа.
   * Проверьте, существует ли путь между каждой парой вершин в графе, используя DFS или BFS.

Пример простого рекурсивного предиката для обхода графа в глубину:

prolog

Копировать код

dfs(Node, Graph, Visited) :-

dfs(Node, Graph, [], Visited).

dfs(Node, Graph, Path, [Node|Visited]) :-

\+ member(Node, Path), % Проверяем, что вершина не посещена ранее

adjacent(Node, Next, Graph),

dfs(Next, Graph, [Node|Path], Visited).

dfs(\_, \_, Visited, Visited). % Базовый случай: достигнут конец пути

Этот предикат dfs использует рекурсивный подход для обхода графа в глубину, отслеживая уже посещенные вершины в списке Visited.

% Определение предикатов

edge(a, b).

edge(a, e).

edge(b, a).

edge(b, c).

edge(c, b).

edge(c, d).

edge(d, c).

edge(e, a).

edge(e, f).

edge(f, e).

edge(f, b).

edge(f, g).

edge(g, f).

% Проверка смежности вершин

adjacent(X, Y) :- edge(X, Y).

adjacent(X, Y) :- edge(Y, X).

% Поиск путей между вершинами

path(X, Y, Path) :- path(X, Y, [], Path).

path(X, Y, Visited, [Y|Visited]) :- adjacent(X, Y).

path(X, Y, Visited, Path) :-

adjacent(X, Z),

Z \== Y,

\+ member(Z, Visited),

path(Z, Y, [X|Visited], Path).

% Обход графа в глубину (DFS)

dfs(Node, Visited) :- dfs(Node, [], Visited).

dfs(Node, Visited, Visited) :- \+ member(Node, Visited).

dfs(Node, Visited, Result) :-

\+ member(Node, Visited),

dfs\_neighbours(Node, NewVisited, Result),

dfs(NewVisited, Result).

dfs\_neighbours(Node, Visited, Result) :-

adjacent(Node, Next),

\+ member(Next, Visited),

dfs(Next, [Next|Visited], Result).

% Проверка связности графа

connected(Graph) :- \+ (member(Node, Graph), \+ path(Node, \_, \_)).

% Пример использования

:- initialization(main).

main :-

writeln("Пути между вершинами:"),

path(a, d, PathAD),

writeln(PathAD),

path(a, g, PathAG),

writeln(PathAG),

nl,

writeln("Обход графа в глубину:"),

dfs(a, VisitedDFS),

writeln(VisitedDFS),

nl,

writeln("Связность графа:"),

(connected([a,b,c,d,e,f,g]) -> writeln("Граф связный") ; writeln("Граф несвязный")).

Этот код реализует обход графа в глубину (DFS), поиск путей между вершинами и проверку связности графа.

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

В Прологе статические и динамические факты отличаются по своему способу объявления и обновления.

1. **Статические факты:**
   * Определяются в коде программы и остаются неизменными во время выполнения программы.
   * Объявляются в разделе Clauses программы.
   * Пролог-система анализирует статические факты перед выполнением программы и использует их для вывода результатов запросов.

Пример статического факта:

prolog

Копировать код

human(socrates).

human(plato).

1. **Динамические факты:**
   * Могут быть добавлены, изменены или удалены во время выполнения программы.
   * Объявляются с помощью директив assert и retract.
   * Позволяют программе динамически адаптироваться к изменяющимся данным или среде.

Пример использования динамических фактов:

prolog

Копировать код

% Декларация предиката как динамического

:- dynamic human/1.

% Добавление фактов в базу знаний во время выполнения

add\_human(john).

add\_human(mary).

% Удаление фактов из базы знаний во время выполнения

remove\_human(john).

% Пример запроса после добавления и удаления фактов

?- human(X).

В этом примере факт human/1 объявлен как динамический с помощью :- dynamic human/1.. Затем во время выполнения программы факты добавляются и удаляются с помощью assert и retract. Таким образом, мы можем динамически обновлять базу знаний и использовать ее для вывода результатов запросов.

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

Предикаты var, nonvar, atom, atomic, name, functor, arg, repeat в Prolog используются для различных типов проверки и обработки данных. Ниже приводится объяснение принципа работы каждого из этих предикатов с примерами.

1. **var/1:**
   * Проверяет, является ли аргумент переменной.
   * Возвращает истину (true), если аргумент - переменная, иначе ложь (false).

Пример:

prolog

Копировать код

?- var(X).

true.

?- var(5).

false.

1. **nonvar/1:**
   * Проверяет, является ли аргумент не переменной (т.е. связан с некоторым значением).
   * Возвращает истину, если аргумент - не переменная, иначе ложь.

Пример:

prolog

Копировать код

?- nonvar(X).

false.

?- nonvar(5).

true.

1. **atom/1:**
   * Проверяет, является ли аргумент атомом (т.е. именем).
   * Возвращает истину, если аргумент - атом, иначе ложь.

Пример:

prolog

Копировать код

?- atom(hello).

true.

?- atom(X).

false.

?- atom(5).

false.

1. **atomic/1:**
   * Проверяет, является ли аргумент атомом или числом.
   * Возвращает истину, если аргумент - атом или число, иначе ложь.

Пример:

prolog

Копировать код

?- atomic(hello).

true.

?- atomic(5).

true.

?- atomic(X).

false.

1. **name/2:**
   * Преобразует атом в список его символов и наоборот.
   * Формат: name(Atom, List).

Пример:

prolog

Копировать код

?- name(hello, L).

L = [104, 101, 108, 108, 111].

?- name(A, [104, 101, 108, 108, 111]).

A = hello.

1. **functor/3:**
   * Возвращает функтор и арность терма или создает терм из функтора и арности.
   * Формат: functor(Term, Functor, Arity).

Пример:

prolog

Копировать код

?- functor(foo(a, b, c), F, A).

F = foo,

A = 3.

?- functor(T, foo, 3).

T = foo(\_G123, \_G124, \_G125).

1. **arg/3:**
   * Возвращает i-тый аргумент терма.
   * Формат: arg(Index, Term, Arg).

Пример:

prolog

Копировать код

?- arg(2, foo(a, b, c), A).

A = b.

?- arg(1, foo(X, b, c), A).

X = A.

1. **repeat/0:**
   * Предикат, который всегда успешно выполняется и вызывает backtracking.
   * Обычно используется в комбинации с другими предикатами для создания циклов.

Пример:

prolog

Копировать код

?- repeat, write('Hello'), nl, fail.

Hello

false.

% Прекращение цикла:

?- repeat, write('Hello'), nl, !.

Hello

true.

Эти предикаты обеспечивают мощные возможности для обработки данных в Prolog и являются важной частью его функциональности.