**Лекция 5** Рекурсивные функции.

**ЦЕЛЬ**: Ознакомить с понятиями: Рекурсивные функции.Примитивно-рекурсивные функции. Примитивно-рекурсивные предикаты. Частично-рекурсивные функции. Общерекурсивные функции. Тезис Черча. Лямбда-исчисление. Ком­би­на­тор­ная опре­де­ли­мость рекурсивных фун­кции

**ВОПРОСЫ**:

1. Рекурсивные функции.Примитивно-рекурсивные функции. Примитивно-рекурсивные предикаты. Частично-рекурсивные функции. Общерекурсивные функции.
2. Тезис Черча.
3. Лямбда-исчисление.
4. Ком­би­на­тор­ная опре­де­ли­мость рекурсивных фун­кции

**ОСНОВНЫЕ ПОНЯТИЯ:**

Еще одним подходом к проблеме формализации понятия алгоритма являются, так называемые, **рекурсивные функции.** Исторически этот подход возник первым, поэтому в математических исследованиях, посвященных алгоритмам, он имеет наибольшее распространение. «Частично определённые рекурсивные функции» и просто «везде определённые рекурсивные функции». Но длинные названия не прижились.

**Рекурсией** называется способ задания функции, при котором значение функции при определенном значении аргументов выражается через уже заданные значения функции при других значениях аргументов. Применение рекурсивных функций в теории алгоритмов основано на идее нумерации слов в произвольном алфавите последовательными натуральными числами. Таким образом, любой алгоритм можно свести к вычислению значений некоторой целочисленной функции при целочисленных значениях аргументов.

Введем несколько основных понятий. Пусть X, Y - два множества. Частичной функцией (или отображением) из Х в Y будем называть пару <D(f), f>, состоящую из подмножества D(f) X (называемого областью определения f) и отображения f: D(f) Y. Если D(f) пусто, то f нигде не определена. Будем считать, что существует единственная нигде не определенная частичная функция.

Через N будем обозначать множество натуральных чисел. Через (N)n (при п 1) будем обозначать n-кратное декартово произведение N на себя, т.е. множество упорядоченных n-ок (х1..., xn), хi N. Основным объектом дальнейших построений будут частичные функции из (N)m в (N)n для различных т и п.

*Частичная функция* f из (N)m в (N)n называется вычислимой, если можно указать такой алгоритм («программу»), который для входного набора х (N)m дает на выходе f(x), если х D(f) и нуль, если х D(f). В этом определении неформальное понятие алгоритма (программы) оказывается связанным (отождествленным) с понятием вычислимости функции. Вместо алгоритмов далее будут изучаться свойства вычислимых функций. Вместо вычислимых функций оказывается необходимым использовать более широкий класс функций (и более слабое определение) - полувычислимые функции. Частичная функция из (N)" в (N)" полувычислима, если можно указать такой алгоритм (программу), который для входного набора х ∈ (N)" дает на выходе х ∈D(f), или алгоритм работает неопределенно долго, если х ∈ D(f). Очевидно, что вычислимые функции полувычислимы, а всюду определенные полувычислимые функции вычислимы.

Частичная функция f называется *невычислимой*, если она не является ни вычислимой, ни полувычислимой. Из вновь введенных понятий основным является полувычислимость, так как вычислимость сводится к нему. Существуют как невычислимые функции, так и функции, являющиеся полувычислимыми, но не вычислимые. (Примеры функции см. литературу):

# Рекурсивная функция (теория вычислимости)

Термин **рекурсивные функции** в теории вычислимости используют для обозначения трёх множеств функций: примитивно рекурсивные функции; общерекурсивные функции; частично рекурсивные функции.

Множество частично рекурсивных функции включает в себя множество общерекурсивных функции, а общерекурсивные функции включают в себя примитивно рекурсивные функции. Частично рекурсивные функции иногда называют просто рекурсивными функциями.

Определение понятия примитивно рекурсивной функции является индуктивным. Оно состоит из указания класса базовых примитивно рекурсивных функций и двух операторов (*подстановки* и *примитивной рекурсии*), позволяющих строить новые примитивно рекурсивные функции на основе уже имеющихся.

К числу базовых примитивно рекурсивных функций относятся функции следующих трёх видов (описания, свойства и примеры см. литературу):

* *Нулевая функция* *O* — функция без аргументов, всегда возвращающая [0](http://ru.wikipedia.org/wiki/0_(ÑÐ¸ÑÐ).
* *Функция следования* *S* одного переменного, сопоставляющая любому натуральному числу *x* непосредственно следующее за ним натуральное число *x* + 1.
* Функции I_n^m, где 0<m\leqslant n, от n переменных, сопоставляющие любому упорядоченному набору x_1,\dots, x_nнатуральных чисел число *xm* из этого набора.

Определим вначале три оператора, позволяющих по одним функциям получать другие.

Сразу следует отметить, что они обладают тем свойством, что, применяя их к функциям, вычислимым в интуитивном смысле, получаем функции, также вычислимые в интуитивном смысле.

**Оператор суперпозиции.**

Суперпозиция является мощным средством получения новых функций из уже имеющихся. Напомним, что суперпозицией называется любая подстановка функций в функции.

Оператором суперпозиции называется подстановка в функцию от *m* переменных *m* функций каждая из которых зависит от *n* одних и тех же переменных. Суперпозиция дает новую функцию уже от *n* переменных.

Например, для функций *h(x*1*,x*2 *,… ,xm)*, *g1(x*1*,x*2 *,… ,xn), g2(x*1*,x*2 *,… ,xn),*

*gm(x1,x2,…,xn)* их суперпозиция дает новую функцию *f ( x ,x , ,xn )*:

*( h, g*1*,g*2 *,… ,gm) h( g 1(x*1*,x*2 *,… ,xn),g2 (x*1*,x*2 *,… ,xn), gm(x*1*,x*2 *,… ,xn))*

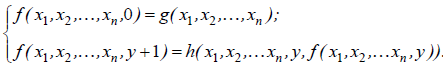
= *f (x*1*,x*2 *,… ,xn).* (1)

В этом случае говорят, что *n*-местная функция *f (x*1*,x*2 *,… ,xn)* получена с помощью оператора суперпозиции из *m*-местной функции *h(x*1*,x*2 *,… ,xm)* и *n*-местных функций *g1 (x*1*,x*2 *,… ,xn), g2 (x*1*,x*2 *,… ,xn), gm(x*1*,x*2 *,… ,xn)* ,

если *f (x*1*,x*2 *,… ,xn)* =*h( g1 (x*1*,x*2 *,… ,xn),g2 (x*1*,x*2 *,… ,xn), gm(x*1*,x*2 *,… ,xn))*  .

**Оператор примитивной рекурсии**.

Оператор примитивной рекурсии *Rn* определяет *( n* 1*)* -местную функцию *f* через *n*-местную функцию *g* и *( n* 2 *)* - местную функцию *h* следующим образом:

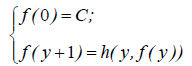
 (2)

Пара равенств (2) называется схемой примитивной рекурсии.

Тот факт, что функция *f* определена схемой (2) выражается равенством

*f (x*1*,x*2 *,… ,xn , y )* *Rn( g,h )*. Эта схема определяет *f* рекурсивно не только через другие функции, но и через значения *f* в предшествующих точках: значение *f* в точке *y+1* зависит от значения *f* в точке *y*. Для вычисления *f (x*1*,x*2 *,… ,xn,k )* понадобится *k+1* вычислений по схеме (2) для *y=*0,1,…,k. Существенным в операторе примитивной рекурсии является то, что независимо от числа переменных в *f* рекурсия ведется только по одной переменной *y*, а остальные *n* переменных *x*1*,x*2 *,… ,xn* на момент применения схемы (2) зафиксированы и играют роль параметров.

В случае, когда *n*=0, т.е. определяемая функция *f* является одноместной, схема (2) принимает более простой вид:

, (3)

где *C* – константа.

Функция называется ***примитивно-рекурсивной***, если она может быть получена из нуль-функции *0(х),* функции следования *S(x)* и функции проекции с помощью конечного числа применений операторов суперпозиции и примитивной рекурсии.

Этому определению можно придать более формальный индуктивный вид.

1. Функции *0(х)* , *S(x)* и *x*1*,x*2 *,… ,xn* для всех натуральных *n*, *m*, где *m* *n*, являются примитивно рекурсивными.

2. Если *g1(x*1*,x*2 *,… ,xn),gm(x*1*,x*2 *,… ,xn), h(x*1*,x*2 *,… ,xn)* примитивно- рекурсивные функции, то  *( h, g*1*,g*2 *,… ,gm) -*  примитивно-рекурсивные функции для любых натуральных *n, m*.

3. Если *g(x*1*,x*2 *,… ,xn)* и *h(x*1*,x*2 *,… ,xn, y, z)* – примитивно рекурсивные функции, то *Rn( g,h )* – примитивно-рекурсивная функция.

4. Других примитивно-рекурсивных функций нет.

Из такого индуктивного описания нетрудно извлечь процедуру, порождающую все примитивно-рекурсивные функции.

**Пример 1** Доказать, что сложение, умножение, возведение в степень примитивно-рекурсивно:

1. Сложение *f**( x,y )* *x* *y* примитивно-рекурсивно.

Решение.

*f+ ( x,* 0*)* *x*  *( x*);

*f**( x,y* 1*)* *f**( x,y )*1*S( f**( x, y ))*.\_\_

Таким образом, 𝑓+(𝑥, 𝑦) = ℎ(𝑥, 𝑦 − 1, 𝑓(𝑥, 𝑦 − 1)) = 𝑅1((𝑥), ℎ(𝑥, 𝑦, 𝑧)),

где ℎ(𝑥, 𝑦, 𝑧) = 𝑧 + 1.

Действительно, имеем:

*f**( x,*0*)* *x;*

*f* *( x,*1*)* *h( x,*0*, f* *( x,*0*))* *x* 1*;*

*f* *( x,*2*)* *h( x,*1*, f* *( x,*1*))* *x* 11*x* 2*;*

*f* *( x,*3*)* *h( x,*2*, f* *( x,*2*))* *x* 2 1*x* 3*;*



*f* *( x,y* 1*)* *h( x,y* 2*, f* *( x,y* 2*))* *x* *( y* 2*)*1*x* *y* 1*;*

*f* *( x,y )* *h( x,y* 1*, f* *( x,y* 1*))* *x* *( y* 1*)*1*x* *y.*

2. Умножение *f**( x,y )* *x* *y* примитивно-рекурсивно.

Решение.

*f**( x,*0*)* 0*;*

*f**( x,y* 1*)* *x* *( y* 1*)* *x* *y* *x* *f**( x,y )**x* *f**( x, f**( x,y ));*

Или *f**( x,y* 1*)* *h( x,y, f**( x,y ))* *x* *z* , где *z=f**(x,y).*

Выполняя действия по полученной схеме, получаем:

*f**( x,*0*)* 0*;*

*f**( x,*1*)* *x* 0 *x;*

*f**( x,*2*)* *x* *x* 2*x;*

*f**( x,*3*)* *x* 2*x* 3*x;*



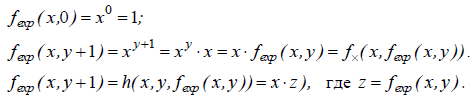
*f**( x,y* 1*)* *x* *( y* 2*)x* *x* *y* *x;*

*f**( x,y )* *x* *x* *y* *x* *x* *y* .

3. Возведение в степень *y*

*fexp ( x, y )* *x* примитивно-рекурсивно.

Решение.

**

*fexp ( x, y* 1*)* *h( x,y, fexp ( x, y ))* *x* *z ),* где *z* *fexp ( x, y )*.

Действительно, выполняя вычисления по полученной схеме, получаем;

*fexp ( x,*0*)* *x* 0 1*;*

*fexp ( x,*1*)* *x* 1*x;*

*fexp ( x,* 2*)* *x* *x* *х* 2*;*



*fexp ( x,* 3 *)* *x*2*x* *х*3*;*



*  

*f exp ( x, y*1*)* *x* *x y* 2*x y -*1*;*

*fexp ( x, y )* *x* *x y*1 *x y* .

**Оператор минимизации**

Пусть задана некоторая функция *f ( x, y )* . Зафиксируем значение *х* и выясним, при каком *y* функция *f(x,y)=*0.

Более сложной задачей является отыскание для данной функции *f(x,y)* и фиксированного *х* наименьшего из тех значений *у*, при которых функция *f(x,y)=*0. Так как результат решения задачи зависит от *х*, то наименьшее значение *у*, при котором функция *f(x,y)=*0 есть функция *х*. Принято обозначение

*( x )* *y**f ( x,y )* 0, (4)

которое читается как: «наименьшее у такое, что *f(x,y)=0*».

Аналогично определяется функция многих переменных:

 *(x*1*,x*2 *,… ,xn)* *y**f (x*1*,x*2 *,… ,xn,y )* 0. (5)

Переход от функции *f (x*1*,x*2 *,… ,xn, y )* к функции  *(x*1*,x*2 *,… ,xn)* принято называть применением **-*оператора***.

Для вычисления функции можно использовать следующий алгоритм:

1.Вычислим *f ( x*1*,x*2 *,… ,xn ,*0*)*. Если это значение *f ( x*1*,x*2 *,… ,xn ,*0*)* равно нулю, то пролагаем *( x*1*,x*2 *,… ,xn)* 0*.* Если *f (x*1*,x*2 *,… ,xn ,*0*)* 0, то переходим к следующему шагу.

2. Вычислим *f( x*1*,x*2 *,… ,xn,*1*)* . Если *f (x*1*,x*2 *,… ,xn,*1*)* 0*,* то полагаем *( x*1*,x*2 *,… ,xn)* 1*.* Если же *f (x*1*,x*2 *,… ,xn,*1*)* 0, то переходим к следующему шагу. И т.д.

Если окажется, что для всех *у* функция *f (x*1*,x*2 *,… ,xn, y )* 0*,* то функцию  *(x*1*,x*2 *,… ,xn)* в этом случае считают неопределенной. Но возможно, что существует такое *у0*, что *f (x*1*,x*2 *,… ,xn, y*0 *)* 0 и, значит, есть и наименьшее *у*, при котором *f (x*1*,x*2 *,… ,xn, y )* 0, и в то же время может случиться, что при некотором *z* (*0<z<y0*) значение функции *f (x*1*,x*2 *,… ,xn,z )* не определено. Очевидно, что в этом случае процесс вычисления наименьшего *у*, при котором *f (x*1*,x*2 *,… ,xn, y )* 0не дойдет до *y0*. И здесь функцию *(x*1*,x*2 *,… ,xn,)* считают неопределенной.

**Пример 2.**

С помощью оператора минимизации реализовать функцию

*f ( x,y )* *x* *y* .

Данная функция может быть построена с помощью оператора

минимизации следующим образом:

Вычислим, например, *f*(8,3), т.е. значение функции при *у*=3, *х*=8.

Для этого положим *у*=3 и будем придавать *х* последовательно значения:

z=0, 3+0=38;

z=1, 3+1=48;

z=2, 3+2=58;

z=3, 3+3=68;

z=4, 3+4=78;

z=5, 3+5=8=8;

Таким образом, *f*(8,3)=5.

Итак, из простейших функций: константы нуля (0(*х*)=0), функции следования (*S(x)=x+1)* и функции проекции ( *(x*1*,x*2 *,… ,xn)* *x*) с помощью операторов суперпозиции и примитивной рекурсии может быть получено огромное разнообразие функций. Тем самым выяснено, что эти функции имеют примитивно-рекурсивное описание, которое однозначно определяет процедуру их вычисления. Следовательно, они естественно относятся к классу вычислимых функций.

Также следует отметить, что, во-первых, все примитивно-рекурсивные функции всюду определены. Это следует из того, что простейшие функции всюду определены и операторы и *Rn* это свойство сохраняют. Во-вторых, строго говоря, мы имеем дело не с функциями, а с их примитивно-рекурсивными описаниями. Эти описания также можно разбить на классы эквивалентности, отнеся в один класс все описания, задающие одну и ту же функцию. Однако задача распознавания эквивалентности примитивно-рекурсивных описаний, как будет далее показано, алгоритмически неразрешима.

Операторы подстановки и примитивной рекурсии определяются следующим образом:

* **Оператор подстановки**. Пусть *f* — функция от m переменных, а g_1, \dots, g_m — упорядоченный набор функций от *n* переменных каждая. Тогда результатом подстановки функций *gk* в функцию *f* называется функция *h* от *n* переменных, сопоставляющая любому упорядоченному набору x_1, \dots, x_nнатуральных чисел число

h(x_1,\ldots,x_n)=f(g_1(x_1,\ldots,x_n),\ldots,g_m(x_1,\ldots,x_n)).

* **Оператор примитивной рекурсии**. Пусть *f* — функция от n переменных, а *g* — функция от *n* + 2 переменных. Тогда результатом применения оператора примитивной рекурсии к паре функций *f* и *g* называется функция *h* от *n* + 1 переменной вида

h(x_1,\ldots,x_n,0)=f(x_1,\ldots,x_n);

h(x_1,\ldots,x_n,y+1)=g(x_1,\ldots,x_n,y,h(x_1,\ldots, x_n,y)).

Множество **примитивно рекурсивных функций** — это минимальное множество, содержащее все базовые функции и замкнутое относительно указанных операторов подстановки и примитивной рекурсии.

## Частично рекурсивные функции определяются аналогичным образом, только к двум операторам подстановки и примитивной рекурсии добавляется ещё оператор минимизации аргумента.

* **Оператор минимизации аргумента**. Пусть *f* — функция от n  натуральных переменных. Тогда результатом применения оператора минимума аргумента к функции *f* называется функция *h* от *n* − 1 переменной, задаваемая следующим определением:

h(x_1,\ldots, x_{n-1}) = min (y), при условии f(x_1,\ldots,x_{n-1},\,y)=0

То есть функция *h* возвращает минимальное значение последнего аргумента функции *f*, при котором её значение равно 0.

Аналогия с императивными языками программирования: Примитивные функции соответствуют программным функциям, в которых используется только арифметические операции, а также условный оператор и оператор арифметического цикла (оператор цикла, в котором число итераций известно на момент начала цикла).

Если же программист начинает использовать оператор цикла while, в котором число итераций заранее неизвестно, и в принципе, может быть бесконечным, то он переходит в класс частично рекурсивных функций. Важным моментом является то, что частично рекурсивные функции для некоторых значений аргумента могут быть не определены, так как оператор минимизации аргумента не всегда корректно определён, именно, функция f может быть не равной нулю ни при каких значениях аргументов. Собственно, оттого, что частично рекурсивные функции могут иметь корректно определённое значение лишь на части аргументов, и пошло их название. С точки зрения программиста результатом частично рекурсивной функции может быть не только число, но и исключение (Exception) или «зависание», соответствующее неопределённому значению.

**Общерекурсивные функции** — это подмножество частично рекурсивных функций, определённых для всех значений аргументов. Задача определения того, является ли частично рекурсивная функция с данным описанием общерекурсивной или нет, алгоритмически неразрешима.

Фундаментальным открытием теории вычислимости явился, так называемый, **тезис Черча**, который в слабейшей форме имеет следующий вид: можно явно указать а) семейство простейших полувычислимых функций; б) семейство элементарных операций, которые позволяют строить по одним полувычислимым функциям другие полувычислимые функции с тем свойством, что любая полувычислимая функция получается за конечное число шагов, каждый из которых состоит в применении одной из элементарных операций к ранее построенным или к простейшим функциям.

**Простейшие функции:**

suc: *N N;* suc*(x)* = *x*+1 - определение следующего за *х* числа;

*l(n)*: *(N)n* *N; l(n) (x1,..., хn)* = 1, *п* 0 - определение «размерности» области определения функции;

*рr: (N)n* *N; pr(x1,..., хn)* = *хi, х* 1 - «проекция» области определения на одну из переменных.

**Элементарные операции над частичными функциями:**

а) *композиция* (или подстановка) ставит в соответствие паре функций *f* из *(N)m* в *(N)n* и *g* из *(N)n* в *(N)p* функцию *h = gof* из *(N)m* в *(N)p,* которая определяется как

*б) соединение* ставит в соответствие частичным функциям *fi* из *(N)ni, i =* 1,..., *k* функцию (*f*i,..., *fk)* из *(N)m* в *(N)n1* х... х *(N)nk,* которая определяется как

в) *рекурсия* ставит в соответствие паре функций *f* из *(N)n* в *N* и *g* из *(N)n+2* в *N* функцию *h* из *(N)n+2 в N,* которая определяется рекурсией по последнему аргументу

*h(x1,*..., *хn*, 1) = *f* (*x1*,*...,xп)* (начальное условие),

*h (x1,...,хn, k+1) = g(x1,...,xn, k, h(x1,...,хn, k))* при *k*1 (рекурсивный шаг).

Область определения *D(h)* описывается также рекурсивно:

г) *операция т,* которая ставит в соответствие частичной функции *f* из *(N)n+1* в *N* частичную функцию *h* из *(N)n* в *N,* которая определяется как Операция *т* позволяет вводить в вычисления перебор объектов для отыскания нужного в бесконечном семействе.

Теперь, когда введены простейшие функции и элементарные операции, можно дать следующие основные определения:

а) последовательность частичных функций *f*i,...,*fN* называют **частично рекурсивным** (соответственно примитивно рекурсивным) описанием функции *fN = f*, если *f*i - одна из простейших функций; *fi* для всех *i* 2 либо является простейшей функцией, либо получается применением одной из элементарных операций к некоторым из функций *f*i,..., *fi-1* (соответственно одной из элементарных операций, кроме *т);*

б) функция *f* называется частично рекурсивной (соответственно примитивно рекурсивной), если она допускает частично рекурсивное (соответственно примитивно рекурсивное) описание.

Теперь можно привести **тезис Черча в обычной форме**:

а) функция *f* полувычислима, если и только если она частично рекурсивна;

б) функция *f* вычислима, если и только если рекурсивны *f* и характеристическая функция *XD(f).*

Характеристическая функция подмножества *Х* в *Y(X Y)* есть такая функция, что Тезис Черча может использоваться как определение алгоритмической неразрешимости.

Пусть имеется счетная последовательность «задач» *P1*, *P2,...,* которые имеют ответ «да» или «нет». Такая последовательность носит название «массовой проблемы». Свяжем с ней функцию *f* из *N* в *N:*

**Массовая проблема *Р* называется алгоритмически разрешимой, если функции *f* и *XD(f)* частично рекурсивны. В противном случае *Р* называется алгоритмически неразрешимой.**

**Ля́мбда-исчисле́ние** (*λ-исчисление*, *лямбда-исчисление*) — формальная система, разработанная американским математиком [Алонзо Чёрчем](http://ru.wikipedia.org/wiki/%D0%A7%D1%91%D1%80%D1%87,_%D0%90%D0%BB%D0%BE%D0%BD%D0%B7%D0%BE" \o "Чёрч, Алонзо), для [формализации](http://ru.wikipedia.org/wiki/%D0%A4%D0%BE%D1%80%D0%BC%D0%B0%D0%BB%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D1%8F) и анализа понятия[вычислимости](http://ru.wikipedia.org/wiki/%D0%92%D1%8B%D1%87%D0%B8%D1%81%D0%BB%D0%B8%D0%BC%D0%BE%D1%81%D1%82%D1%8C).λ-исчисление может рассматриваться как семейство [прототипных языков программирования](http://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%82%D0%BE%D1%82%D0%B8%D0%BF%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5" \o "Прототипное программирование). Их основная особенность состоит в том, что они являются языками *высших порядков*. Тем самым обеспечивается систематический подход к исследованию операторов, *аргументами* которых могут быть другие операторы, а [значением](http://ru.wikipedia.org/w/index.php?title=%D0%97%D0%BD%D0%B0%D1%87%D0%B5%D0%BD%D0%B8%D0%B5%D0%BC&action=edit&redlink=1) также может быть оператор. Языки в этом семействе являются функциональными, поскольку они основаны на представлении о [функции](http://ru.wikipedia.org/wiki/%D0%A4%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D1%8F_(%D0%BC%D0%B0%D1%82%D0%B5%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)) или [операторе](http://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%82%D0%BE%D1%80_(%D0%BC%D0%B0%D1%82%D0%B5%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)), включая функциональную аппликацию и функциональную абстракцию. λ-исчисление реализовано [Джоном Маккарти](http://ru.wikipedia.org/wiki/%D0%94%D0%B6%D0%BE%D0%BD_%D0%9C%D0%B0%D0%BA%D0%BA%D0%B0%D1%80%D1%82%D0%B8) в языке [Лисп](http://ru.wikipedia.org/wiki/%D0%9B%D0%B8%D1%81%D0%BF). В основу **λ-исчисления** положены две фундаментальные операции: *Аппликация*  и *Абстракция* или *λ-абстракция.*  ( описания см. литературу)

**Комбина́торная ло́гика** — раздел дискретной математики, который тесно связан с λ-исчислением, т. к. описывает вычислительные процессы. С момента своего возникновения комбинаторная логика и лямбда-исчисление были отнесены к *неклассическим логикам*. **Комбинаторная логика и лямбда-исчисление** — это такие формальные системы, в которых центральной разрабатываемой сущностью является представление об [объекте](http://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82). В первой из них — комбинаторной логике, — механизм связывания переменных в явном виде отсутствует, а во второй он имеется. Наличие явного механизма связывания предполагает и наличие связанных переменных, но тогда есть и свободные переменные, а также механизмы замещения формальных параметров — связанных переменных, — на фактические параметры, то есть*подстановка*.Изначальным назначением комбинаторной логики был именно анализ *процесса подстановки*. В качестве ее сущностей планировалось использовать объекты в виде *комбинаций констант*. Лямбда-исчислению отводилась роль средства уточнения представлений об *алгоритме* и *вычислимости*. Как следствие, комбинаторная логика дает в руки инструмент для анализа процесса подстановки. Через короткий промежуток времени оказалось, что обе эти системы можно рассматривать как языки программирования (см. также комбинаторное программирование).

В обеих системах исчисляются объекты, они являются исчислениями или языками *высших порядков*, то есть имеются средства описания отображений или операторов, которые определяются на отображениях или операторах, а в качестве результата вырабатывают также отображения или операторы. Самое существенное, что именно *отображение* считается объектом. В этом их принципиальное отличие от всего многообразия других систем, в которых первичной сущностью обычно считают представление о *множестве* и его элементах.К настоящему времени оба эти языка не только стали основой для всей массы исследований в области компьютерных наук и компьютинга, но и широко используются в теории программирования.

**Ком­би­на­тор­ная опре­де­ли­мость рекурсивных фун­кции.**(описания и примеры р*е­ше­ния* см. литературу). (пример:Уста­но­вление ком­би­на­тор­ной опре­де­ли­мости фун­кции f(x)=x )

**ВЫВОДЫ:** Применение рекурсивных функций в теории алгоритмов основано на идее нумерации слов в произвольном алфавите последовательными натуральными числами. Таким образом, любой алгоритм можно свести к вычислению значений некоторой целочисленной функции при целочисленных значениях аргументов.Множество частично рекурсивных функции включает в себя множество общерекурсивных функции, а общерекурсивные функции включают в себя примитивно рекурсивные функции..λ-исчисление может рассматриваться как семейство прототипных языков программирования,они являются функциональными языками, языками  *высших порядков*. Фундаментальным открытием теории вычислимости явился тезис Черча.

[**ЛЯМБДА-ИСЧИСЛЕНИЕ**](https://anton-k.github.io/ru-haskell-book/book/14.html#%D0%BB%D1%8F%D0%BC%D0%B1%D0%B4%D0%B0-%D0%B8%D1%81%D1%87%D0%B8%D1%81%D0%BB%D0%B5%D0%BD%D0%B8%D0%B5)

Лямбда-исчисление описывает понятие алгоритма. Ещё до появления компьютеров в 30-е годы двадцатого века математиков интересовал вопрос о возможности создания алгоритма, который мог бы на основе заданных аксиом дать ответ о том верно или нет некоторое логическое высказывание. Например у нас есть базовые утверждения и логические связки такие как “и”, “или”, “для любого из”, “существует один из”, с помощью которых мы можем строить из базовых высказываний составные. Некоторые из них окажутся ложными, а другие истинными. Нам интересно узнать какие. Но для решения этой задачи прежде всего необходимо было понять а что же такое алгоритм?

Ответ на этот вопрос дали Алонсо Чёрч (Alonso Church) и Алан Тьюринг (Alan Turing). Чёрч разработал лямбда-исчисление, а Тьюринг теорию машин Тьюринга. Оказалось, что задача автоматического определения истинности формул в общем случае не имеет решения.

В основе лямбда-исчисление лежит понятие функции. Мы можем составлять сложные функции из простейших, а также подставлять в функции аргументы, которые могут быть как константами так и другими функциями. Как только мы составили выражение мы можем передать его вычислителю. Он подставляет аргументы в функции и возвращает такое выражение, в котором невозможно далее проводить подстановки аргументов. Этот процесс проведения подстановок считается вычислением алгоритма.

В рамках теории машин Тьюринга алгоритм описывается по-другому. Машина Тьюринга имеет внутреннее состояние, Состояние содержит некоторое значение, которое изменяется по ходу работы машины. Машина живёт не сама по себе, она читает ленту символов. Лента символов – это большая цепочка букв. На каждую букву машина реагирует серией действий. Она может изменить значение состояния, обновить букву в ленте или перейти к следующему или предыдущему символу. Есть состояния, которые обозначают конец работы, они называются терминальными. Как только машина дойдёт до терминального состояния мы считаем, что вычисление алгоритма закончилось. После этого мы можем считать результат из состояний машины.

Функциональные языки программирования основаны на лямбда-исчислении. Поэтому мы будем говорить именно об этом описании алгоритма.

[**Лямбда исчисление без типов**](https://anton-k.github.io/ru-haskell-book/book/14.html#%D0%BB%D1%8F%D0%BC%D0%B1%D0%B4%D0%B0-%D0%B8%D1%81%D1%87%D0%B8%D1%81%D0%BB%D0%B5%D0%BD%D0%B8%D0%B5-%D0%B1%D0%B5%D0%B7-%D1%82%D0%B8%D0%BF%D0%BE%D0%B2)

[**Составление термов**](https://anton-k.github.io/ru-haskell-book/book/14.html#%D1%81%D0%BE%D1%81%D1%82%D0%B0%D0%B2%D0%BB%D0%B5%D0%BD%D0%B8%D0%B5-%D1%82%D0%B5%D1%80%D0%BC%D0%BE%D0%B2)

Можно считать, что лямбда исчисление это такой маленький язык программирования. В нём есть множество символов, которые считаются переменными, они что-то обозначают и неделимы. В лямбда-исчислении программный код называется термом. Для написания программного кода у нас есть всего три правила:

* Переменные x , y , z … являются термами.
* Если M и N – термы, то (MN) – терм.
* Если x – переменная, а M – терм, то (λx.M) – терм

В формальном описании добавляют ещё одно правило, оно говорит о том, что других термов нет. Первое правило, говорит о том, что у нас есть алфавит символов, который что-то обозначает, эти символы являются базовыми строительными блоками программы. Второе и третье правила говорят о том как из базовых элементов получаются составные. Второе правило – это правило применения функции к аргументу. В нём M обозначает функцию, а N обозначает аргумент. Все функции являются функциями одного аргумента, но они могут принимать и возвращать функции. Поэтому применение трёх аргументов к функции Fun будет выглядеть так:

(((Fun Arg1) Arg2) Arg3)

Третье правило говорит о том как создавать функции. Специальный символ лямбда ( λ ) в выражении (λx.M) говорит о том, что мы собираемся определить функцию с аргументом x и телом функции M . С такими функциями мы уже сталкивались. Это безымянные функции. Приведём несколько примеров функций. Начнём с самого простого, определим тождественную функцию:

(λx.x)

Функция принимает аргумент x и тут же возвращает его в теле. Теперь посмотрим на константную функцию:

(λx.(λy.x))

Константная функция является функцией двух аргументов, поэтому наш терм принимает переменную x и возвращает другой терм функцию (λy.x) . Эта функция принимает y , а возвращает x. В Haskell мы бы написали это так:

\x -> (\y -> x)

Точка сменилась на стрелку, а лямбда потеряла одну ножку. Теперь определим композицию. Композиция принимает две функции одного аргумента и направляет выход второй функции на вход первой:

(λf.(λg.(λx.(f(gx)))))

Переменные f и g – это функции, которые участвуют в композиции, а x это вход результирующей функции. Уже в таком простом выражении у нас пять скобок на конце. Давайте введём несколько соглашений, которые облегчат написание термов:

|  |  |  |
| --- | --- | --- |
|  | **Пишем** | **Подразумеваем** |
| Опустим внешние скобки: | λx.x | (λx.x) |
| В применении группируем скобки влево: | fghx | ((fg)h)x |
| В функциях группируем скобки вправо: | λx.λy.x | (λx.(λy.x)) |
| Пишем функции нескольких аргументов с одной лямбдой: | λxy.x | (λx.(λy.x)) |

С этими соглашениями мы можем переписать терм для композиции так:

λfgx.f(gx)

Сравните с выражением на языке Haskell:

\f g x -> f (g x)

Выражения очень похожи. Haskell иногда называют засахаренной версией лямбда исчисления. В лямбда-исчислении мы не будем ставить пробелы для применения аргументов к функции. Мы будем считать, что все имена однобуквенные. При этом переменные мы будем писать с маленькой буквы, а составные термы с большой.

Определим ещё несколько функций. Например так выглядит функция flip:

λfxy.fyx

Или можно записать в более явном виде, выделим функцию двух аргументов:

λf.λxy.fyx

Определим функцию on, она принимает функцию двух аргументов \* и функцию одного аргумента f , а возвращает функцию двух аргументов, в которой к аргументам сначала применяется функция f , а затем они передаются в функцию \* :

λ\*f.λx.\*(fx)(fx)

В лямбда-исчислении есть только префиксное применение поэтому мы написали \*(fx)(fx) вместо привычного (fx)\*(fx) . Здесь операция \* это не только умножение, а любая бинарная функция.

[**Абстракция**](https://anton-k.github.io/ru-haskell-book/book/14.html#%D0%B0%D0%B1%D1%81%D1%82%D1%80%D0%B0%D0%BA%D1%86%D0%B8%D1%8F)

Функции в лямбда-исчислении называют абстракциями. Мы берём терм M и параметризуем его по переменной x в выражении λx.M . При этом если в терме M встречается переменная x , то она становится связанной. Например в терме λx.λy.x $ Переменная x является *связанной*, но в терме λy.x , она уже не связана. Такие переменные называют *свободными*. Множество связанных переменных терма M мы будем обозначать BV(M) $ от англ. bound variables, а множество свободных переменных мы будем обозначать FV(M) от англ. free variables.

На интуитивном уровне процесс абстракции заключается в том, что мы смотрим на несколько частных случаев и видим в них что-то общее. Это общее мы выделяем в функцию, которая параметризована частностями. Например мы видим выражения:

λx.+xx, λx.\*xx

И в том и в другом у нас есть функция двух аргументов + или \* и мы делаем из неё функцию одного аргумента. Мы можем абстрагировать (параметризовать) это поведение в такую функцию:

λb.λx.bxx

На Haskell мы бы записали это так:

\b -> \x -> b x x

[**Редукция. Вычисление термов**](https://anton-k.github.io/ru-haskell-book/book/14.html#%D1%80%D0%B5%D0%B4%D1%83%D0%BA%D1%86%D0%B8%D1%8F.-%D0%B2%D1%8B%D1%87%D0%B8%D1%81%D0%BB%D0%B5%D0%BD%D0%B8%D0%B5-%D1%82%D0%B5%D1%80%D0%BC%D0%BE%D0%B2)

Процесс вычисления термов заключается в подстановке аргументов во все функции. Выражения вида:

(λx.M) N

Заменяются на

M[x=N]

Эта запись означает, что в терме M все вхождения x заменяются на терм N . Этот процесс называется *редукцией* терма. А выражения вида (λx.M) N называются *редексами*. Проведём к примеру редукцию терма:

(λb.λx.bxx)\*

Для этого нам нужно в терме (λx.bxx) заменить все вхождения переменной b на переменную \* . После этого мы получим терм:

λx.\*xx

В этом терме нет редексов. Это означает, что он вычислен или находится в *нормальной форме*.

[**α -преобразование**](https://anton-k.github.io/ru-haskell-book/book/14.html#alpha-%D0%BF%D1%80%D0%B5%D0%BE%D0%B1%D1%80%D0%B0%D0%B7%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)

При подстановке необходимо следить за тем, чтобы у нас не появлялись лишние связывания переменных. Например рассмотрим такой редекс:

(λxy.x) y

После подстановки за счёт совпадения имён переменных мы получим тождественную функцию:

λy.y

Переменная y была свободной, но после подстановки стала связанной. Необходимо исключить такие случаи. Поскольку с ними получается, что имена связанных переменных в определении функции влияют на её смысл. Например смысл такого выражения

(λxz.x) y

После подстановки будет совсем другим. Но мы всего лишь изменили обозначение локальной переменной y на z . И смысл изменился, для того чтобы исключить такие случаи пользуются переименованием переменных или *α -преобразованием*. Для корректной работы функций необходимо следить за тем, чтобы все переменные, которые были свободными в аргументе, остались свободными и после подстановки.

[**β -редукция**](https://anton-k.github.io/ru-haskell-book/book/14.html#beta-%D1%80%D0%B5%D0%B4%D1%83%D0%BA%D1%86%D0%B8%D1%8F)

Процесс подстановки аргументов в функции называется *β -редукцией*. В редексе (λx.M)N вместо свободных вхождений x в M мы подставляем N . Посмотрим на правила подстановки:

x[x=N] ⇒ N

y[x=N] ⇒ y

(PQ)[x=N] ⇒ (P[x=N] Q[x=N])

(λy.P)[x=N] ⇒ (λy.P[x=N]), y∉FV(N)

(λx.P)[x=N] ⇒ (λx.P)

Первые два правила определяют подстановку вместо переменных. Если переменная совпадает с той, на место которой мы подставляем терм N , то мы возвращаем терм N , иначе мы возвращаем переменную:

x[x=N] ⇒ N

y[x=N] ⇒ y

Подстановка применения термов равна применению термов, в которых произведена подстановка:

(PQ)[x=N]⇒(P[x=N] Q[x=N])

При подстановке в лямбда-функции необходимо учитывать связность переменных. Если переменная аргумента отличается от той переменной на место которой происходит подстановка, то мы заменяем в теле функции все вхождения этой переменной на N :

(λy.P)[x=N]⇒(λy.P[x=N]), y∉FV(N)

Условие y∉FV(N) означает, что необходимо следить за тем, чтобы в N не оказалось свободной переменной с именем y , иначе после подстановки она окажется связанной. Если такая переменная в N всё-таки окажется мы проведём α -преобразование в терме λy.M и заменим y на какую-нибудь другую переменную.

В последнем правиле мы ничего не меняем, поскольку переменная x оказывается связанной. А мы проводим подстановку только вместо свободных переменных:

(λx.P)[x=N]⇒(λx.P)

Отметим, что не любой терм можно вычислить, например у такого терма нет нормальной формы:

(λx.xx)(λx.xx)

На каждом шаге редукции мы будем вновь и вновь возвращаться к исходному терму.

[**Стратегии редукции**](https://anton-k.github.io/ru-haskell-book/book/14.html#%D1%81%D1%82%D1%80%D0%B0%D1%82%D0%B5%D0%B3%D0%B8%D0%B8-%D1%80%D0%B5%D0%B4%D1%83%D0%BA%D1%86%D0%B8%D0%B8)

В главе о ленивых вычислениях нам встретились две стратегии вычисления выражений. Это вычисление по имени и вычисление по значению. Также там мы узнали о том, что ленивые вычисления это улучшенная версия вычисления по имени, в которой аргументы функций вычисляются не более одного раза.

Эти стратегии вычисления пришли из лямбда-исчисления. Если нам нужно избавиться от всех редексов в выражении, то с какого редекса лучше начать? В вычислении по значению (*аппликативная стратегия*) мы начинаем с самого левого редекса, который не содержит других редексов, то есть с самого маленького подвыражения. А в вычислении по имени (*нормальная стратегия*) мы начинаем с самого левого внешнего редекса. Левый редекс означает, что в записи выражения он находится ближе всех к началу выражения.

**Теорема** (Карри)

Если у терма есть нормальная форма, то последовательное сокращение самого левого внешнего редекса приводит к ней.

Эта теорема говорит о том, что стратегия вычисления по имени может вычислить все термы, которые имеют нормальную форму. В том, что вычисление по значению может не справиться с некоторыми такими термами мы можем на следующем примере:

(λxy.x) z ((λx.xx)(λx.xx))

Этот терм имеет нормальную форму z несмотря на то, что мы передаём вторым аргументом в константную функцию терм, у которого нет нормальной формы. Алгоритм вычисления по значению зависнет при вычислении второго аргумента. В то время как алгоритм вычисления по имени начнёт с самого внешнего терма и там определит, что второй аргумент не нужен.

Ещё один важный результат в лямбда-исчислении был сформулирован в следующей теореме:

**Теорема** (Чёрча-Россера)

Если терм X редуцируется к термам Y1 и Y2 , то существует терм L , к которому редуцируются и терм Y1 и терм Y2 .

Эта теорема говорит о том, что у терма может быть только одна нормальная форма. Поскольку если бы их было две, то существовал третий терм, к которому можно было бы редуцировать эти нормальные формы. Но по определению нормальной формы, мы не можем её редуцировать. Из этого следует, что нормальные формы должны совпадать.

Теорема Чёрча-Россера указывает на способ сравнения термов. Для того чтобы понять равны термы или нет, необходимо привести их к нормальной форме и сравнить. Если термы совпадают в нормальной форме, значит они равны.

[**Рекурсия. Комбинатор неподвижной точки**](https://anton-k.github.io/ru-haskell-book/book/14.html#%D1%80%D0%B5%D0%BA%D1%83%D1%80%D1%81%D0%B8%D1%8F.-%D0%BA%D0%BE%D0%BC%D0%B1%D0%B8%D0%BD%D0%B0%D1%82%D0%BE%D1%80-%D0%BD%D0%B5%D0%BF%D0%BE%D0%B4%D0%B2%D0%B8%D0%B6%D0%BD%D0%BE%D0%B9-%D1%82%D0%BE%D1%87%D0%BA%D0%B8)

В лямбда-исчислении все функции являются безымянными. Это означает, что мы не можем в теле функции вызвать саму функции, ведь мы не можем на неё сослаться, кажется, что у нас нет возможности строить рекурсивные функции. Однако это не так. Нам на помощь придёт комбинатор неподвижной точки. По определению комбинатор неподвижной точки решает задачу: для терма F найти терм X такой, что

FX=X

Существует много комбинаторов неподвижной точки. Рассмотрим Y -комбинатор:

Y = λf.(λx.f(xx))(λx.f(xx))

Убедимся в том, что для любого терма F , выполнено тождество: F(YF)=YF :

YF=(λx.F(xx))(λx.F(xx))=F(λx.F(xx))(λx.F(xx))=F(YF)

Так с помощью Y -комбинатора можно составлять рекурсивные функции.

[**Кодирование структур данных**](https://anton-k.github.io/ru-haskell-book/book/14.html#%D0%BA%D0%BE%D0%B4%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5-%D1%81%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80-%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85)

Вы наверное заметили, что пока мы составляли лишь обобщённые функции. Эти функции комбинируют другие функции, они не выполняют никаких действий над элементами. Что если нам захочется вычислять логические значения или воспользоваться числами?

Оказывается, что логические значения, числа, пары, списки и другие конструкции могут быть закодированы с помощью термов лямбда-исчисления. Тезис Чёрча утверждает, что с помощью лямбда-терма можно представить любую вычислимую числовую функцию. В 1936 году Чёрч с помощью лямбда-исчисления доказал существование неразрешимых проблем в теории чисел. Из этого следовала неразрешимость арифметики и неразрешимость исчисления логики предикатов первого порядка. Система аксиом называется разрешимой в том случае, если существует такой алгоритм, который позволяет по виду формулы определить следует ли она из заданных аксиом или нет.

Посмотрим как с помощью термов кодируются структуры данных. Далее для сокращения записи мы будем считать, что в лямбда исчислении можно определять синонимы с помощью знака равно. Запись N=M говорит о том, что мы дали обозначение N терму M . Этой операции нет в лямбда-исчислении, но мы будем пользоваться ею для удобства.

[**Логические значения**](https://anton-k.github.io/ru-haskell-book/book/14.html#%D0%BB%D0%BE%D0%B3%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B5-%D0%B7%D0%BD%D0%B0%D1%87%D0%B5%D0%BD%D0%B8%D1%8F)

Суть логических значений заключается в операторе If , с помощью которого мы можем организовывать ветвление алгоритма. Есть два терма True и False , которые для любых термов a и b , обладают свойствами:

If True a b = a

If False a b = b

Термы True , False и If , удовлетворяющие таким свойствам выглядят так:

True = λt f.t

False = λt f.f

If = λb x y.bxy

Проверим выполнение свойств:

If True a b⇒(λb x y.bxy)(λt f.t) a b⇒(λt f.t) a b⇒a

If False a b⇒(λb x y.bxy)(λt f.f) a b⇒(λt f.f) a b⇒b

Свойства выполнены. Логические константы кодируются постоянными функциями двух аргументов. Функция True возвращает первый аргумент, игнорируя второй. А функция False делает то же самое, но наоборот. В такой интерпретации логическое отрицание можно закодировать с помощью функции flip. Также мы можем выразить и другие логические операции:

And = λa b.a b False

Or = λa b.a True b

Мы определили логические значения не конкретными значениями, а свойствами функций. Мы построили функции, которые ведут себя как логические значения. Этот способ определения напоминает, определение класса типов. Мы объявили три метода True , False и If и сказали, что экземпляр класса должен удовлетворять определённым свойствам, которые накладывают взаимные ограничения на методы класса. Ни один из методов не имеет смысла по отдельности, важно то как они взаимодействуют.

[**Натуральные числа**](https://anton-k.github.io/ru-haskell-book/book/14.html#%D0%BD%D0%B0%D1%82%D1%83%D1%80%D0%B0%D0%BB%D1%8C%D0%BD%D1%8B%D0%B5-%D1%87%D0%B8%D1%81%D0%BB%D0%B0)

Оказывается, что с помощью термов лямбда исчисления можно закодировать и натуральные числа с арифметическими операциями. Мы будем кодировать числа Пеано. Для этого нам понадобится нулевой элемент и функция определения следующего элемента. Их можно закодировать так:

Zero = λsz.z

Succ = λnsz.s(nsz)

Как и в случае логических значений числа кодируются функциями двух аргументов. Число определяется по терму, подсчётом цепочки первых аргументов s . Например так выглядит число два:

Succ (Succ Zero)⇒(λnsz.s(nsz))(Succ Zero)⇒λsz.s((Succ Zero)sz)⇒

λsz.s(((λnsʹzʹ.sʹ(nsʹzʹ)) Zero)sz)⇒λsz.s((λsʹzʹ.sʹ(Zero sʹzʹ)) sz)⇒

λsz.s((λsʹzʹ.sʹzʹ) sz)⇒λsz.s(sz)

И мы получили два вхождения первого аргумента в теле функции. Определим сложение и умножение. Сложение принимает две функции двух аргументов и возвращает функцию двух аргументов.

Add=λ m n s z.m s (n s z)

В этой функции мы применяем m раз аргумент s к значению, в котором аргумент s применён n раз, так мы и получаем m+n применений аргумента s . Сложим 3 и 2:

Add 3 2⇒λs z.3 s (2 s z)⇒λs z.3 s (s (s z))⇒λs z.s ( s (s (s (s z))))⇒5

В умножении чисел m и n мы будем m раз складывать число n :

Mul=λm n s z.m (Add n) Zero

[**Конструктивная математика**](https://anton-k.github.io/ru-haskell-book/book/14.html#%D0%BA%D0%BE%D0%BD%D1%81%D1%82%D1%80%D1%83%D0%BA%D1%82%D0%B8%D0%B2%D0%BD%D0%B0%D1%8F-%D0%BC%D0%B0%D1%82%D0%B5%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)

В конструктивной математике существование объекта может быть доказано только описанием алгоритма, с помощью которого можно построить объект. Например доказательство методом “от противного” отвергается.

Лямбда исчисление строит конструктивное описание функции. По лямбда-терму мы можем не только вычислять значения функции, но и понять как она была построена. В классической теории, функция это множество пар (x, f(x)) аргумент-значение, которое обладает свойством:

x=y ⇒ f(x)=f(y)

По этому определению мы ничего не можем сказать о внутренней структуре функции. Мы можем собирать из одних функций другие с помощью подстановки значений, но мы никак не сможем понять, что находится внутри функции. Лямбда исчисление решает эту проблему.

[**Расширение лямбда исчисления**](https://anton-k.github.io/ru-haskell-book/book/14.html#%D1%80%D0%B0%D1%81%D1%88%D0%B8%D1%80%D0%B5%D0%BD%D0%B8%D0%B5-%D0%BB%D1%8F%D0%BC%D0%B1%D0%B4%D0%B0-%D0%B8%D1%81%D1%87%D0%B8%D1%81%D0%BB%D0%B5%D0%BD%D0%B8%D1%8F)

Предположим, что мы решили написать язык программирования на основе лямбда-исчисления. Было бы очень неэффективно представлять числа с помощью чисел Пеано. Ведь у нас есть процессор и мы можем спросить у него чему равно значение и получить ответ очень быстро.

В этом случае пользуются расширенным лямбда исчислением. В нём два типа примитивов это переменные и константы. Для констант мы можем определять специальные правила редукции. Например мы можем дополнить исчисление константами:

+, \*, 0, 1, 2, ...

И ввести для них правила редукции, которые запрашивают ответ у процессора:

a+b = AddWithCPU(a,b)

a\*b = MulWithCPU(a,b)

Так же мы можем определить и константы для логических значений:

True, False, If, Not, And, Or

И определить правила редукции:

If True a b = a

If False a b = b

Not True = False

Not False = True

Add False a = False

Add True b = b

…

Такие правила называют δ -редукцией (дельта-редукция).

[**Комбинаторная логика**](https://anton-k.github.io/ru-haskell-book/book/14.html#%D0%BA%D0%BE%D0%BC%D0%B1%D0%B8%D0%BD%D0%B0%D1%82%D0%BE%D1%80%D0%BD%D0%B0%D1%8F-%D0%BB%D0%BE%D0%B3%D0%B8%D0%BA%D0%B0)

Одновременно с лямбда-исчислением развивалась комбинаторная логика. Она отличается более компактным представлением. Есть всего лишь одно правило, это применение функции к аргументу. А функции строятся не из произвольных термов, а из набора основных функций. Набор основных функций называют *базисом*.

Рассмотрим лямбда-термы:

λx.x, λy.y, λz.z

Все эти термы несут один и тот же смысл. Они представляют тождественную функцию. Они равны, но с точностью до обозначений. Эта навязчивая проблема с переобозначением аргументов была решена в комбинаторной логике. Посмотрим как строятся термы:

* Есть набор переменных x , y , z , …. Переменная – это терм.
* Есть две константы K и S , они являются термами.
* Если M и N – термы, то (MN) – терм.
* Других термов нет.

Определены правила редукции для базисных термов:

Kxy = x

Sxyz = xz(yz)

В этих правилах мы пользуемся соглашением о расстановки скобок. Также как и в лямбда исчислении в применении скобки группируются влево. Когда мы пишем Kxy , мы подразумеваем ((Kx)y) . Термы в комбинаторной логике принято называть комбинаторами. Редукция происходит до тех пор пока мы можем заменять вхождения базисных комбинаторов. Так если мы видим связку KXY или SXYZ , где X , Y , Z произвольные термы, то мы можем их заменить согласно правилам редукции. Такие связки называют редексами. Если в терме нет ни одного редекса, то он находится в нормальной форме. Замену редекса принято называть *свёрткой*

Интересно, что комбинаторы K и S совпадают с определением класса Applicative для функций:

instance Applicative (r->) where

pure a r = a

(<\*>) a b r = a r (b r)

В этом определении у функций есть общее окружение r , из которого они могут читать значения, так же как и в случае типа Reader. В методе pure (комбинатор K ) мы игнорируем окружение (это константная функция), а в методе <\*> (комбинатор S ) передаём окружение в функцию и аргумент и составляем применение функции в контексте окружения r к значению, которое было получено в контексте того же окружения.

Вернёмся к проблеме различного представления тождественной функции в лямбда-исчислении. В комбинаторной логике тождественная функция выражается так:

I=SKK

Проверим, определяет ли этот комбинатор тождественную функцию:

Ix=SKKx=Kx(Kx)=x

Сначала мы заменили I на его определение, затем свернули по комбинатору S , затем по левому комбинатору K . В итоге получилось, что

Ix=x

[**Связь с лямбда-исчислением**](https://anton-k.github.io/ru-haskell-book/book/14.html#%D1%81%D0%B2%D1%8F%D0%B7%D1%8C-%D1%81-%D0%BB%D1%8F%D0%BC%D0%B1%D0%B4%D0%B0-%D0%B8%D1%81%D1%87%D0%B8%D1%81%D0%BB%D0%B5%D0%BD%D0%B8%D0%B5%D0%BC)

Комбинаторная логика и лямбда-исчисление тесно связаны между собой. Можно определить функцию φ , которая переводит термы комбинаторной логики в термы лямбда-исчисления:

φ(x) = x

φ(K) = λxy.x

φ(S) = λxyz.xz(yz)

φ(XY) = φ(X) φ(Y)

В первом уравнении x – переменная. Также можно определить функцию ψ , которая переводит термы лямбда-исчисления в термы комбинаторной логики.

ψ(x) = x

ψ(XY) = ψ(X) ψ(Y)

ψ(λx.Y) = [x].ψ(Y)

Запись [x].T , где x – переменная, T – терм, обозначает такой терм D , из которого можно получить терм T подстановкой переменной x , выполнено свойство:

([x].T) x=T

Эта запись означает параметризацию терма T по переменной x . Терм [x].T можно получить с помощью следующего алгоритма:

[x].x = SKK

[x].X = KX, x∉V(X)

[x].XY = S([x].X)([x].Y)

В первом уравнении мы заменяем переменную на тождественную функцию, поскольку переменные совпадают. Запись V(X) во втором уравнении обозначает множество всех переменных в терме X . Поскольку переменная по которой мы хотим параметризовать терм (или абстрагировать) не участвует в самом терме, мы можем проигнорировать её с помощью постоянной функции K . В последнем уравнении мы параметризуем применение.

С помощью этого алгоритма можно для любого терма T , все переменные которого содержатся в {x1,...xn} составить такой комбинатор D , что Dx1...xn=T . Для этого мы последовательно парметризуем терм T по всем переменным:

[x1,..., xn].T=[x1].([x2,..., xn].T)

Так постепенно мы придём к выражению, считаем что скобки группируются вправо:

[x1].[x2]....[xn].T

[**Немного истории**](https://anton-k.github.io/ru-haskell-book/book/14.html#%D0%BD%D0%B5%D0%BC%D0%BD%D0%BE%D0%B3%D0%BE-%D0%B8%D1%81%D1%82%D0%BE%D1%80%D0%B8%D0%B8)

Комбинаторную логику открыл Моисей Шейнфинкель. В 1920 году на докладе в Гёттингене он рассказал основные положения этой теории. Комбинаторная логика направлена на выделение простейших строительных блоков математической логики. В этом докладе появилось понятие частичного применения. Шейнфинкель показал как функции многих переменных могут быть сведены к функциям одного переменного. Далее в докладе описываются пять основных функций, называемых комбинаторами:

Ix = x – функция тождества

Cxy = x – константная функция

Txyz = xzy – функция перестановки

Zxyz = x(yz) – функция группировки

Sxyz = xz(yz) – функция слияния

С помощью этих функций можно избавиться в формулах от переменных, так например свойство коммутативности функции A можно представить так: TA=A . Эти комбинаторы зависят друг от друга. Можно убедиться в том, что:

I = SCC

Z = S(CS)S

T = S(ZZS)(CC)

Все комбинаторы выражаются через комбинаторы C и S . Ранее мы пользовались другими обозначениями для этих комбинаторов. Обозначения K и S ввёл Хаскель Карри (Haskell Curry). Независимо от Шейнфинкеля он переоткрыл комбинаторную логику и существенно развил её. В современной комбинаторной логике для обозначения комбинаторов I , C , T , Z и S (по Шейнфинкелю) принято использовать имена I , K , C , B , S (по Карри).

[**Лямбда-исчисление с типами**](https://anton-k.github.io/ru-haskell-book/book/14.html#%D0%BB%D1%8F%D0%BC%D0%B1%D0%B4%D0%B0-%D0%B8%D1%81%D1%87%D0%B8%D1%81%D0%BB%D0%B5%D0%BD%D0%B8%D0%B5-%D1%81-%D1%82%D0%B8%D0%BF%D0%B0%D0%BC%D0%B8)

Мы можем добавить в лямбда-исчисление типы. Предположим, что у нас есть множество V базовых типов. Тогда тип это:

T=V ?∣ T→T

Тип может быть либо одним элементом из множества базовых типов. Либо стрелочным (функциональным) типом. Выражение “терм M имеет тип α ” принято писать так: Mα . Стрелочный тип α→β как и в Haskell говорит о том, что если у нас есть значение типа α , то с помощью операции применения мы можем из терма с этим стрелочным типом получить терм типа β .

Опишем правила построения термов в лямбда-исчислении с типами:

* Переменные xα , yβ , zγ , … являются термами.
* Если Mα→β и Nα – термы, то (Mα→βNα)β – терм.
* Если xα – переменная и Mβ – терм, то (λxα.Mβ)α→β – терм
* Других термов нет.

Типизация накладывает ограничение на то, какие выражения мы можем комбинировать. В этом есть плюсы и минусы. Теперь наша система является *строго нормализуемой*, это означает, что любой терм имеет нормальную форму. Но теперь мы не можем выразить все функции на числах. Например мы не можем составить Y -комбинатор, поскольку теперь самоприменение (ee) невозможно.

Мы ввели типы, но лишились рекурсии. Как нам быть? Эта проблема решается с помощью введения специальной константы Yτ(τ→τ)→τ , которая обозначает комбинатор неподвижной точки. Правило редукции для Y :

(Yτfτ→τ)τ=(fτ→τ(Yτfτ→τ))τ

Можно убедиться в том, что это правило проходит проверку типов. Типизированное лямбда-исчисление дополненное комбинатором неподвижной точки способно выразить все числовые функции.