## 2. Основы лямбда-исчисления

### 2.1. Лямбда-исчисление: ключевые понятия

Лямбда-исчисление, разработанное А. Черчем для преодоления некоторых проблем в математической логике, фактически служит теоретическим базисом функционального программирования. Это исчисление формализует понятие вычислимости на базе математического понятия функции, причем мощность этого исчисления такова, что все функциональные программы можно преобразовать в эквивалентные вычислимые ***лямбда-выражения***. Кроме того, можно описать все базовые свойства функционального языка, опираясь на понятия лямбда-исчисления.

В то же время лямбда-исчисление элементарно с точки зрения синтаксиса его объектов (лямбда-выражений) и производимых над ними вычислительных действий.

Над лямбда-выражениями могут быть выполнены всего две операции: ***функциональная абстракция*** и ***функциональная аппликация***. Функциональная абстракция служит для превращения некоторого выражения в функцию, устанавливая ее аргумент (что соответствует понятию формального параметра в языках программирования). К примеру, выражение `x+y` еще не является функцией, однако запись `f(x)=x+y` уже фиксирует аргумент функции (и ее имя), тем самым определяя функцию.

Функциональная аппликация есть по сути применение функции к ее аргументу. Лямбда-исчисление оперирует только функциями от одной переменной, причем функции не имеют имени. Тем самым это исчисление анонимных (безымянных) функций, которое включает:

- специальную нотацию (лямбда-нотацию), устанавливающую синтаксические правила записи функций и выражений;
- правила преобразования (вычисления/переписывания/вывода) функциональных выражений.

Как и в любом формальном исчислении (в частности, логическом), преобразования функциональных выражений в лямбда-исчислении носят чисто синтаксический характер, являясь по сути текстовыми преобразованиями (переписыванием строк текста). Таким образом, лямбда-исчисление служит для чисто синтаксического описания свойств функций и манипулирования с ними.

### 2.2 Лямбда-выражения

Особенностью синтаксиса функциональных выражений в лямбда-исчислении является использование особой *префиксной записи*. В отличие от традиционной в математике префиксной записи функций, при которой имя функции стоит перед аргументами, а все аргументы группируются скобками и разделяются запятыми, например: *f(x,y)* , в лямбда-исчислении применяется префиксная запись, в которой аргументы не разделяются запятыми и не группируются скобками: *f x y*. По сути разделителем аргументов выступает пробел, а скобки используются при необходимости для группировки выражения, выступающего в роли аргумента функции. Инфиксная же запись операций вида `x + y` не применяется вовсе. Вот пример записи в лямбда-исчислении выражения с операциями-функциями сложения и умножения: `*(+ x y)2`

Приведем формальное определение понятия лямбда-выражения, или *лямбда-терма*, используя для этого БНФ, (в угловых скобках указаваются определяемые понятия):

< лямбда-выражение>::= λ<идентификатор> . <лямбда-выражение> |
<лямбда-выражение>< лямбда -выражение> |
(<лямбда-выражение>) |
<идентификатор> |
<константа>
<константа>::= 1|0|-1|2|-2|… |true|false|+| -| *| /| >| >=| …

В этом определении представлены пять случаев (видов) лямбда-выражения (терма).

Первый из них соответствует операции функциональной абстракции. Символ λ обозначает операцию абстракции, после него стоит идентификатор-переменная, которая рассматривается как аргумент функции. Абстракция по сути означает превращение переменной в аргумент функции. После точки записывается выражение для вычисления значения функции, им может быть любое допустимое лямбда-выражение, оно называется *телом абстракции*. Само же выражение называют *лямбда-абстрацией*, а переменную между λ и точкой – *связанной переменной абстракции*. Лямбда-абстракция интерпретируется следующим образом: функция от указанной переменной, которая возвращает значение – тело абстракции, например:
λx.+ x 1 – функция от x , возвращающая x+1 (увеличение аргумента на 1);
λx.λy.*(+ x y)2 – функция, вычисляющая удвоенную сумму двух своих аргументов x и y (абстракция применена дважды).

Второй случай (вид) лямбда-выражения соответствует функциональной аппликации, т.е. применению (вызову) функции и записывается в виде лямбда-выражения для этой функции (им может быть лямбда-абстракция), за которым следует выражение для её аргумента, например:

f(+ z 9)– применение функции f к аргументу (+ z 9);

(λx.+ x 1)7 – применение функции увеличения на 1 к аргументу-числу 7 ;

Последние примеры демонстрируют также третий случай лямбда-выражения - группировку скобками выражения для аргумента функции или групировку лямбда-абстракции.

Два последних случая лямбда-выражения – идентификатор и константа. Константы включают не только числа, но и знаки (имена) арифметических операций и операций сравнения.

Приведем дополнительные примеры лямбда-выражений:

6 + 6 1 λz.(λy.z) λx.x (последнее – тождественная функция)

Заметим, что лямбда-исчисление может не включать константы – в этом случае оно называется *чистым*. Для чистого исчисления можно сформулировать более короткую БНФ для лямбда-выражения (терма), в которой символ x обозначает переменную (идентификатор), а t - терм,:

t::= x | λx.t | t t| (t)

Таким образом, с помощью операции абстракции можно конструировать новые функции, а с помощью аппликации их применять с конкретными аргументами.

В отличие от математики и программирования лямбда-исчисление оперирует только функциями от одной переменной. Этой минимальной возможности (в смысле количества переменных) достаточно, чтобы выразить функции от нескольких аргументов. Действительно, известны два способа свести функции с несколькими параметрами только лишь к функциям от одного аргумента. В частности, для функции f от двух аргументов

I. можно сгруппировать ее аргументы в *кортеж* (пару) вида `(x,y)`, тогда применение функции имеет вид `f(x,y)` – что соответствует записи, принятой в математике.

II. можно использовать идею *частичного применения функции*, когда у нее фиксируется один аргумент – в результате получается функция, у которой на один аргумент меньше, чем у исходной. Например, фиксируя у функции вычисления максимума один аргумент, мы получаем функцию от другого ее аргумента: `max(3,y)=f(y)` . Аналогично, для лямбда-абстракции, в которой фиксирован второй аргумент операции сложения: `(λx.+ x 1)`. В обоих случаях функция двух переменных рассматривается как функция одной переменной, возвращающая функцию другой переменной.

Поскольку в лямбда-исчислении используется идея частичного применения, запись в лямбда-исчислении `λx.λy.*(+ x y)2` по сути означает функцию от `x`, возвращающую функцию от `y`, которая вычисляет `2*(x+y)`. Запись же `(+ 6)` означает одноместную функцию. Поскольку результатом частичного вычисления является другая функция, то функции от нескольких аргументов фактически являются функционалами. Напомним, что *функционалом*, или *функцией высшего порядка* называется функция, аргументом или результатом которой является другая функция.

Идею использования частичного вычисления для преобразования функции от нескольких переменных к одноместным (одноаргументным) функциям высказывали многие математики, но названо по имени одного из них – американского математика Хаскелла Карри (Haskel Curry).

***Карринг*** (***каррирование***) – преобразование функции от нескольких аргументов в эквивалентный набор (суперпозицию) одноаргументных функций (или иными словами, в функцию, берущую аргументы по одному).

Функция от двух аргументов `f: A×B→C` в результате каррирования преобразуется в функцию `f':A→(B→C)`, т.е. отображение декартова произведения множеств `A` и `B` (областей определения аргументов) во множество `C` преобразуется в отображение из `A` во множество функций из `B` в `C`.

В `λ`–исчислении каррирование фактически делается при записи функций. В некоторые функциональные языки (в частности, в Хаскель и ML) оно встроено, т.е. осуществляется автоматически.

Строго говоря, в лямбда-исчислении применения одноместных функций, образующих функцию от нескольких аргументов, должны заключаться в скобки (`B` – тело функции), тем не менее для упрощения записи скобки могут быть опущены:

$(\ldots((\lambda x_1.\ \lambda x_2\ldots.\ \lambda x_n.B)\ E_1)E_2)\ldots E_n) \equiv (\lambda x_1.\ \lambda x_2\ldots.\ \lambda x_n.B)\ E_1\ E_2\ \ldots E_n$

Такая возможность опирается на установленное соглашение об операции аппликации (применения функции): аппликация левоассоциативна. Это означает, что в записи вида $f\ E_1\ \ldots\ E_n$ функция всегда применяется к самому левому аргументу, и поэтому скобки возле каждого вложенного применения можно опустить без изменения смысла выражения. В частности, запись $f\ E_1\ E_2\ E_3$ эквивалентна следующей записи:

$f\ E_1\ E_2\ E_3 \equiv (((f\ E_1)\ E_2)\ E_3)$

`u v w = (u v) w`

При записи математических выражений скобки полезны, поскольку они фиксируют аргументы операций и тем самым явно указывают структуру выражений.

Тем не менее соглашения о приоритете и ассоциативности операций позволяют сократить количество (избавиться от излишних) скобок.

При вычислении алгебраических выражений обычно учитывается и применяется *левая ассоциативность* операций сложения и умножения: `a*b*c=(a*b)*c`
и *правая ассоциативность* операции возведения в степень: `a↑b↑c=a↑(b↑c)`

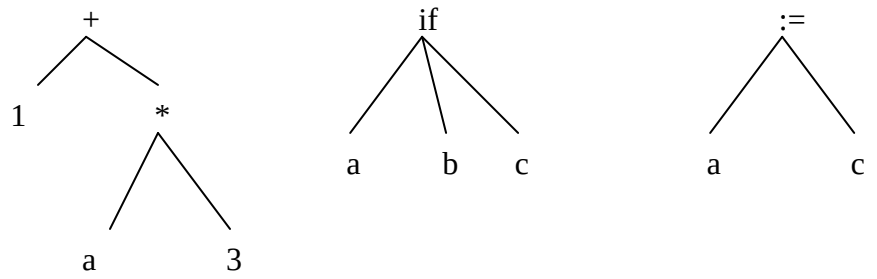
В лямбда-исчислении используется еще одно соглашение о записи выражений, которое позволяет опускать часть скобок при записи тел лямбда-абстракций. Считается, что тело абстракции простирается как можно дальше – либо до конца всего выражения, либо до первой непарной закрывающей скобки (иными словами: абстракция забирает себе все, до чего дотянется). К примеру, в записи `λx.λy.x y z` скобки должны быть расставлены следующим образом:
`λx.(λy.x y z) ≡ λx.(λy.((x y)z))`

Кроме этих соглашений, скобки играют привычную роль группировки операций в выражениях. Так, выражение `λx.λy.(λz.z)x(+ y z)` есть лямбда-абстракция, а с дополненными скобками `(λx.λy.(λz.z)x)(+ y z)` уже получаем лямбда-аппликацию.

Как без лишних скобок, так и с ними, лямбда-выражения могут быть сложны для восприятия: `λy.((λy.y)(λx.(λy.y)x))(λz.y z)`

Для выявления структуры выражений может быть использован так называемый *абстрактный синтаксис*. В отличие от конкретного (поверхностного) синтаксиса, относящегося к строчной, текстовой записи выражения, абстрактный синтаксис задает внутреннее представление в виде размеченного дерева, делающего структуру выражений более наглядной и понятной. Эти деревья называются *абстрактными синтаксическими деревьями* (АСД), или деревьями абстрактного синтаксиса.

Абстрактный синтаксис выявляет структуру выражений различного рода - на Рис.1 показано несколько примеров АСД:




`5+a*(3- b)` `if a then b-8 else c` `a:=c/(6+b)` ближе к АСД

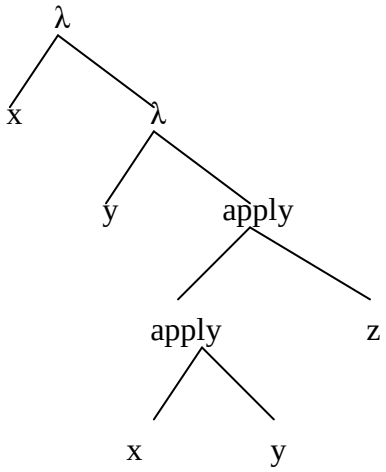
Рис. 1. Примеры АСД исправить деревья

Выражения в абстрактном синтаксисе упрощаются: исключаются лишние символы - разделители, скобки и т.п. В узлах дерева находятся операции и их операнды (последние - в листьях дерева).

Рассмотрим АСД для лямбда-выражений, в этих деревьях в качестве операций будут фигурировать только абстракция (обозначается символом `λ`) и аппликация (обозначается как `apply`). На Рис. 2 показано АСД для следующих двух выражений

λx.λy.x y z = λx.(λy.((x y)z)))

и λx.λy.(λz.z)x (+ y z) = λx.(λy.(((λz.z)x)(+ y z)))




λx.λy.x y z λx.λy.(λz.z)x (+ y z)

Рис.2 АСД двух лямбда-выражений

Можно заметить, что не все переменные в втором выражении связаны некоторой абстракцией. Переменная является *свободной* в некотором выражении *E*, если она не является связанной никакой объемлющей операцией абстракции. Таковой является Переменная z, участвующая в операции сложения рассматриваемого выражения, является свободной. Поскольку в это выражение переменная z входит дважды, точнее говорить о связанности/свободности каждого *вхождения переменной* в некоторое выражение.

Вхождение переменной в выражение *E свободно*, если в АСД выражения оно не связано никакой вышележащей абстракцией. Причем если по одной и той же переменной абстракция проводилась несколько раз, то эта переменная связана самой поздней (нижней в дереве) абстракцией. Так, в выражении λx.λy.λx.z переменная x связана с последней абстакцией по x. Рисунок 3 АСД-дерева

Рассмотрим примеры. В выражении (λy.x)(λx.x):

свободная↑ ↑связанная

т.е. в своем первом вхождении переменная x свободна, а во втором - связана. λx.λy.(λz.z)x(+ y z) первое вхождение z связано, а второе - свободно, вхождения же переменных x и y (они единственные) связаны. Еще пример

Заметим, что переменная может быть свободной в выражении *E*, но связанной в объемлющем выражении *E′*, например, переменная z связана в *E′*=λz.λy.λx.z, и свободна в *E*=λx.z.

Лямбда-выражение без свободных переменных называется *замкнутыми*. Замкнутые выражения называются также *комбинаторами*.

Простейший пример комбинатора - тождественная функция: id≡λx.x . Более сложный пример: ???

В чистом λ-исчислении всё является функциями (аргументы и результаты функций – функции ?)

## 2.3 Редукция в лямбда-исчислении

Редукция моделирует вычисление лямбда-выражений, она производится по нескольким правилам, представляющим по сути текстовые преобразования. Эти правила включают: α-преобразования, β- и δ-редукцию, а также правило вычисления констант и идентификаторов. Свое название редукция получила потому, что преобразования (вычисление) призваны редуцировать, т.е. упрощать исходное лямбда-выражение.

Рассмотрим подробнее правила редукции.

I. Вычисление константы даёт ту же самую константу, например: `27 → 27`

II. **δ-редукция** состоит в вычислении константных (встроенных) функций, в частности, сложение или умножение чисел, выполняемое по правилам их вычисления. Например:

$+\ 2\ 2 \rightarrow_{\delta} 4$

$*(+\ 2\ 2)(-\ 4\ 1) \rightarrow_{\delta} *(+\ 2\ 2)3 \rightarrow_{\delta} *\ 4\ 3 \rightarrow_{\delta} 12$

Первый пример показывает вычисление за один шаг δ-редукции. Результат (редуцированное выражение) записывается справа от стрелки (слева стоит редуцируемое выражение), а к самой стрелке приписывается символ ?? Во втором примере исходное выражение последовательно редуцируется уже за три шага, на каждом шаге выполняется соответствующее правило δ-редукции (?)

III. **β-редукция** представляет собой правило ?) применения лямбда-абстракций, т.е. уже описанных функций. Согласно этому правилу, происходит замена каждого вхождения в тело абстракции формального параметра функции на соответствующий фактический параметр, что в языках программирования соответствует вызову функции с параметрами.

Важно, что в лямбда-исчислении такая замена является чисто текстовой (т.е редукцией строк), при этом фактически происходит копирование тела абстракции с заменой всех вхождений связанной переменной абстракции на выражение аргумента функции. Например:

$(\lambda x.*\ x\ x)3 \rightarrow_{\beta} *\ 3\ 3 \rightarrow_{\delta} 9$

$\lambda x.\lambda y.(\lambda z.z)\ x\ (+\ y\ 1) \equiv \lambda x.(\lambda y.((\underline{\lambda z}.z)\ \underline{x})\ (+\ y\ 1))) \rightarrow_{\beta}$

$\lambda x.(\lambda y.(x\ (+\ y\ 1)))$

$(\lambda x.\lambda y.*(+\ x\ y)2)7\ 3 \rightarrow_{\beta} (\lambda y.*(+\ 7\ y)2)\ 3 \rightarrow_{\beta} *(+\ 7\ 3)2 \rightarrow_{\delta}$

$* 10\ 2 \rightarrow_{\delta} 20$

В первом и втором примерах β-редукция выполняется один раз, а в третьем сначала выполняется два шага β-редукции, а затем два шага δ-редукции.

Как можно заметить, при β-редукции происходит упрощение выражения за счет того, что исключается знак операции λ и связанная переменная, однако в целом упрощение происходит не всегда. В случаях, когда тело абстракции содержит много вхождений связанной переменной, а фактический параметр представляет собой довольно громоздкое выражение, при подстановке его получается более длинное выражение.

Лямбда-выражение, редуцируемое по правилам β- или δ-редукции, называется *редексом* (от англ. *redex* ≡ *reducible expression*).

IV. **α-преобразование** (**α-конверсия**) - его необходимость и условия применения рассмотрим на примере.

Пусть имеется функциональное выражение λx.(λx.x)(* 2 x)
Обе абстракции этой функции (внешняя и внутренняя) связывают одну и ту же переменную x, но в их телах эта переменная обозначает разные объекты. По этой причине вычисление функции с некоторым аргументом может быть ошибочным (β-редукция выполняется по ранее сформулированному правилу):

(λx.(λx.x)(* 2 x))1 →β (λx.1)(* 2 1) →β 1

Оба вхождения переменной были заменены на конкретный аргумент, хотя первое вхождение не подразумевало этого. Причина ошибки связана с конфликтом имен: переменная внутренней абстракции имеет такое же имя, что и заменяемая переменная, и при вычислении следует оставить внутреннюю абстракцию без изменений.

По аналогичной причине ошибка возникнет и при редукции выражения

λx.((λy.λx.* x y)x) →β λx.(λx.* x x)

Здесь функция от y (абстракция по y) применяется к конкретному аргументу-переменной x, которая является свободной (в теле самой внешней абстракции) и совпадает по имени с одной из связанных переменных тела применяемой абстракции.

Чтобы избежать подобных конфликтов, необходимо переименовать переменные в выражениях так, чтобы каждая переменная внутренней и внешней абстракции имела уникальное имя. Такое переименование связанных переменных абстракций называется *α-преобразованием*.

Интуитивно понятно, что переименование переменных функций не затрагивает значения этих функций, например, выражения (λy.y)z и (λx.x)z являются равнозначными. Будучи примененным к выше рассмотренным примерам, α-преобразование даст в итоге корректное вычисление:

(λx.(λx.x)(* 2 x))1 →α (λx.(λy.y)(* 2 x))1
→β (λy.y)(* 2 1) →β (* 2 1) →δ 2

λx.((λy.λx.* x y)x) →α λx.((λy.λz.* z y)x) →β λx.(λz.* z x)

Таким образом, α-преобразование необходимо для безопасного выполнения β-редукции. β-редукция *безопасна* для вычисления выражения $E_1$ $E_2$, если ни одно имя свободной в $E_2$ переменной не совпадает с именами связанных в $E_1$ переменных.

В свою очередь, переименование должно быть корректным: α-преобразование есть такое переименование λx.E →α λx'.E'
где E' получается из E заменой всех свободных вхождений переменной x на x' при условии, что x' сама не является свободной в E.

Вот пример правильного переименования: λx.f x y →α λz. f z y
и неправильного: λx.f x y → λy.f y y

Лямбда-выражения, эквивалентные с точностью до имён связанных переменных, называются *алфавитно-эквивалентными* (*α-эквивалентными*): $f \equiv_\alpha g$.
К примеру, выражения λx.λy.f x y, λc.λy.f c y и λc.λx.f c x являются алфавитно-эквивалентными, а λx.λy.f x y и λy.λx.f x y не являются.

Заметим, что подобное переименование переменных выполняется в функциональных языках программирования, однако это трудоемкая/затратная операция.

## 2.4 Нормальный и аппликативный порядок редукции

При вычислении лямбда-выражений встает вопрос об окончании его преобразований, а также о выборе редекса - выражения или подвыражения, подлежащего редукции на текущем шаге преобразования.

Говорят, что лямбда-выражение находится *в нормальной форме*, если к нему нельзя применить никакое правило редукции (т.е. оно не содержит редексов). Простейшим примером выражения в нормальной форме является выражение λx.x (тождественная функция)

Таким образом, нормальная форма соответствует концу вычислений. На каждом шаге вычисления происходит выбор редекса и применение одного из правил редукции – до тех пор, пока не достигнута нормальная форма.

Рассмотрим пример редукции, когда на каждом шаге есть только один редекс (переменная применяемой функции и ее аргумент подчеркиваются): только один?

(λx.λy.x(λz.y z))((λx.λy.y)8)(λx.(λy.y)x)) →β
λy.((λx.λy.y)8)(λx.(λy.y)x)) (λz.y z) →β
λy.((λy.y)(λx.(λy.y)x)) (λz.y z) →β
λy.((λy.y)(λx.x)) (λz.y z) →β
λy.(λx.x) (λz.y z) →β
λy.(λz.y z)

В следующем примере, при редукции выражения
(λx.x)((λx.x)(λz.(λx.x)z))
выбор редекса уже не однозначен. Запись этого выражения можно упростить, если использовать обозначение тождественной функции id≡ λx.x , тогда получаем:

id(id(λ.id z)) – в этом выражении три редекса (один их них - внешний, два других подчеркнуты), и оно вычисляется за три шага, при этом всегда редуцируется внешний редекс:

id(id(λ.id z)) →β id(λz.id z) →β λz.id z →β λz.z

Но другой порядок рассмотрения редексов приводит к тому же результату:

id(id(λ.id z)) →β id(id(λ.id z)) →β λz.id z →β λz.z

Аналогично, при вычислении выражения (λf.λx.f 4 x)(λy.λx.+ x y)3
при НПР:
(λf.λx.f 4 x)(λy.λx.+ x y)3 →β (λx.(λy.λx.+ x y)4 x)3 →β
(λy.λx.+ x y)4 3 →β (λx.+ x 4)3 →β + 4 3 →δ 7
и при АПР:
(λf.λx.f 4 x)(λy.λx.+ x y)3 →β (λx.(λy.λx.+ x y)4 x)3 →β
(λx.(λx.+ x 4)x)3 →β (λx.+ x 4)3 →β + 3 4 →δ 7
получаем одинаковый результат.

Рассмотрим еще один пример, когда редексов несколько и можно редуцировать выражение разными способами.

(λx.λy.y)((λz.z z)(λz.z z))

В этом выражении два редекса: всё выражение (внешний редекс), а также внутренний редекс (он подчеркнут). Если сначала вычислять внутренний редекс, получим:
(λz.z z)(λz.z z) →β (λz.z z)(λz.z z) →β …
т.е. вычисление будет бесконечным (зацикливается).

Если же сначала редуцировать внешний редекс, то применяется функция (λx.λy.y), которая игнорирует свой аргумент, то за один шаг вычислений получается нормальная форма: λy.y

Сформулируем несколько необходимых определений. Самым левым редексом является тот редекс, символ λ которого расположен левее всех.

Самым внешним является редекс, который не содержится внутри другого редекса. Самым внутренним редексом является редекс, который не содержит внутри себя других редексов.

В лямбда-исчислении известны два основных порядка редукции:

***Аппликативный порядок редукции*** (АПР) предписывает вычислять самый левый из самых внутренних редексов.

***Нормальный порядок редукции*** (НПР) предписывает вычислять самый левый из самых внешних редексов.

Если порядок не фиксирован, то говорят, что имеет место *полная редукция*.

Таким образом, в последнем из рассмотренных примеров НПР даёт нормальную форму, а АПР – зацикливание. Связано это с тем, что НПР откладывает вычисление аргумента (и оно может оказаться ненужным),а АПР в первую очередь вычисляет аргумент функции.

Кроме вопроса о зацикливаемости вычислений, ещё очень важен вопрос, одинаковые ли результаты получаются при разных порядках редукции.

Для лямбда-исчисления доказана **терема ЧерчаРоссера**:

Более известно **следствие теоремы Черча-Россера**:

если выражение может быть приведено разными способами к двум нормальным формам, то они либо совпадают, либо являются алфавитно-эквивалентными формами.

Это следствие означает, что если вычисление лямбда-выражения закончится, то мы получим идентичные результаты при любом порядке редукции (единственность нормальной формы с точностью до алфавитной эквивалентности).

Еще одно доказанное утверждение – **теорема стандартизации**:

НПР всегда преобразует выражение к нормальной форме, если нормальная форма существует (с точностью до алфавитной эквивалентности).

В контексте функциональных языков программирования НПР и АПР соответствуют понятиям ***энергичного вычисления*** ("делай всё, что можешь") и ***ленивого вычисления*** ("не делай ничего, пока это не потребуется").

Энергичное вычисление приблизительно соответствует вызову (передаче) аргумента по значению, а ленивое вычисление – передаче аргумента по имени (вычисление по необходимости?).

Сравнивая АПР и НПР применительно к реализации в функциональных языках программирования, следует отметить, что АПР значительно более эффективно при реализации на компьютерах несмотря на избыточное (и порою ненужное) вычисление аргументов функции. Неээффективность НПР связано с необходимостью сохранять все невычисленное выражение аргумента, в то же время НПР позволяет вычислить те выражения, что невычислимы при АПР (из-за возникающих бесконечных вычислений), в частности, дает возможность программировать обработку бесконечных структур данных.

В большинстве функциональных языков программирования обычно применяются как энергичные, так ленивые вычисления, но один из этих видов является основным. Так, в языке Лисп в основном энергичные вычисления (АПР), и большинство функций является строгими. ***Строгой*** называют функцию, которая предварительно вычисляет свои аргументы. В пику этому, в языке Хаскель в основном ленивые вычисления (НПР) и нестрогие функции.

## 2.5 Чистое лямбда-исчисление: моделирование величин и операций

Чистое лямбда-исчисление есть исчисление без констант, его выражения (термы) построены исключительно из переменных применением аппликации и абстракции.

Тем не менее оно достаточно мощное – в нем можно смоделировать все необходимые величины, структуры данных, операции над ними.

Рассмотрим, как можно ввести в чистое лямбда-исчисление булевские величины `true` и `false` и логические операции с ними. Один из возможных способов:

`true ≡ λx.λy.x`

`false ≡ λx.λy.y`

Поскольку основное назначение этих логических констант – работа в рамках условных операторов и выражений, необходимо дать соответствующее определение в лямбда-исчислении условного выражения, т.е. функции `cond(p,q,r)` от трех аргументов, эквивалентной конструкции `if p then q else r`. Для `cond` необходимо выполнение следующих равенств: `cond(true,q,r)= q`

`cond(false,q,r)= r`

Вот определение этой функции: `cond ≡ λp.λq.λr.p q r`

Проверим правильность вычисления условной конструкции для `p=true`:

`cond(true,a,b)=`

`(λp.λq.λr.p q r)(λx.λy.x)a b →β (λq.λr.(λx.λy.x) q r)a b →β (λr.(λx.λy.x) a r)b →β (λx.λy.x) a b →β (λy.a)b →β a`

Подобным образом можно убедиться, что и для `p=false` условное выражение работает правильно:

`cond(false,a,b)= (λp.λq.λr.p q r)(λx.λy.x)a b →β … →β b`

Заметим, что по сути булевские значения как бы "переключатели" (условные выражения): они принимают два аргумента и выбирают либо первый из них, либо второй. Рассмотренное представление булевских величин не единственно возможное: они сделаны именно так, чтобы упростить условное выражение `cond`.

Для указанных булевских констант логические операции конъюнкции и дизъюнкции определяются следующим образом:

`and ≡ λx.λy.x y false`

`or ≡ λx.λy.(x true)y`

Убедимся в правильности работы конъюнкции на одном из возможных вариантов значений ее аргументов, вычислив `and true false` (в редексах будем подчеркивать выполняемые подстановки):

`(λx.λy.x y (λx.λy.y))(λx.λy.x)(λx.λy.y) →β`

`(λy.(λx.λy.x) y (λx.λy.y))(λx.λy.y) →β`

`(λx.λy.x)(λx.λy.y)(λx.λy.y) →β`

`(λy.(λx.λy.y))(λx.λy.y) →β`

`(λx.λy.y) ≡ false`

Аналогично доказывается ее правильность и на других наборах, а также правильность дизъюнкции, например, при вычислении `or true false`:

`(λx.λy.(x true) y)true false →β`

`(λy.(true true) y)false →β`

`(true true) false →β`

`((λx.λy.x) true)false →β`

`(λy. true) false → true`

Рассмотрим теперь аналог натуральных чисел, основанный на идее, что натуральное число - это либо некоторая начальная величина (число 0, точка отсчёта `z`) либо величина, которая следует за предыдущей (на 1 больше)

понятие натурального числа N: нужно ввести -, каждое следующее число – прибавление к предыдущему единицы.

Следующие лямбда-выражения (комбинаторы, так называемые нумералы Чёрча) моделируют натуральные числа:

$C_0 \equiv \lambda s.\lambda z.z$ представление нуля
$C_1 \equiv \lambda s.\lambda z.s\ z$ единица
$C_2 \equiv \lambda s.\lambda z.s\ (s\ z)$ двойка
$C_3 \equiv \lambda s.\lambda z.s\ (s\ (s\ z))$ тройка
…

Каждое число есть функция двух аргументов: начального значения z (нуль) и s – функции следования (перехода от предыдущего числа). Для $C_n$ – функция, которая n раз применяет s к z ("активное" число).

Функция succ прибавления единицы к такому числу должна иметь своим аргументом число, которое реализовано как функция от двух аргументов. Таким образом, получаем функцию от трех аргументов:

$succ \equiv \lambda n.\lambda s.\lambda z.s(n\ s\ z)$ $C_n \rightarrow C_{n+1}$

здесь n s z - n раз примененная к z функция s (число n) и эта функция применяется еще раз, для перехода к следующему числу (т.е. прибавление 1).

Заметим, что нуль и логическое значение false имеют одинаковое представление, такая ситуация встречается и в языках программирования, когда в роли логического отрицания выступает 0. Важна функция проверки на ноль, которая реализуется следующим лямбда-выражением:

$is\_zero \equiv \lambda m.m(\lambda x.false)true$

Если на вход этой функции подается число $C_0$, то согласно его реализации выражение $(\lambda x.false)$ игнорируется, и в итоге получается true , в ином случае выражение вычисляется с результатом false.

Сложение двух чисел похоже на прибавление единицы, но только необходимо к добавить не единицу, а второе число:

$plus \equiv \lambda m.\lambda n.\lambda s.\lambda z.m\ s(n\ s\ z)$

Функция plus применяет s n раз к z , а потом к результату применяет s еще m раз.

Функция mult умножения чисел m и n строится на основе сложения, реализуя сложение m копий числа n:

$mult\backslash times \equiv \lambda m.\lambda n.m(plus\ n)C_0$

что можно упростить до $mult\backslash times \equiv \lambda m.\lambda n.m(plus\ n)C_0$

Возведение числа n в степень m реализует функция

$power \equiv \lambda m.\lambda n.n\ m$

проверить для power 2 3

Также как и для булевских величин, числа можно представить в лямбда-исчислении не единственным образом. Однако различные представления одной структуры поведенчески эквивалентны, т.е их ключевые свойства одинаковы.

## 2.6 Лямбда-исчисление: рекурсивные функции

Одним из ключевых вопросов вычислений является представление рекурсивных функций. Обычно рекурсивной функцией называют функцию, которая в своем теле содержит вызов самой себя, причем этот вызов осуществляется по имени функции. Классическим примером является функция, вычисляющая факториал числа:

f(n) = if n=0 then 1 else n*f(n-1)

Однако в лямбда-исчислении все функции безымянны, так что необходим способ, позволяющий функциям вызывать себя не по имени, а как-то иначе. Этот менее очевидный способ - передача тела функции через ее аргумент, он и применяется в лямбда-исчислении. Тем самым, рекурсивная функция в лямбда-исчислении – это функция, использующая сама себя в качестве аргумента.

Рассмотрим этот способ на примере определения функции вычисления факториала, записав ее определение в лямбда-нотации, с использованием функций `cond`, `=`, `*`, `-` и числовых констант `0` и `1`, которые либо считаются встроенными (т.е. константными), либо выражены в чистом лямбда-исчислении рассмотренным выше способом:

```
λn.cond(= n 0)1(* n(f(- n 1)))
```

Поскольку функцию `f` необходимо сделать параметром, необходима абстракция по `f`:

```
λf.λn.cond(= n 0)1(* n(f(- n 1)))
```

Все это выражение мы обозначим как `g`:

```
g ≡ λf.λn.cond(= n 0)1(* n(f(- n 1)))
```

Для связи переменной `f` со значением вычисляемой функции будем использовать специальное лямбда-выражение (функцию) `Y`, удовлетворяющее свойству

```
Y g=g(Y g)
```

для любого выражения `g` (т.е. выражение в левой части редуцируется в выражение правой части. Такое выражение получило название *Y-комбинатора*, конкретное представление Y-комбинатора рассмотрим чуть ниже.

Проведем редукцию для нашей конкретной функции `g`, с использованием указанного свойства Y-комбинатора:

```
Y g= Y(λf.λn.cond(= n 0)1(* n(f(- n 1))))→ по свойству
(λf.λn.cond(= n 0)1(* n(f(- n 1))))(Y g)→β
(λn.cond(= n 0)1(* n((Y g)(- n 1))))→
```

Заметим, что подчеркнутое выражение совпадает с исходным, и поэтому его можно преобразовать подобным образом, получив в итоге выражение

```
... → (λn.cond(= n 0)1
        (* n(λn.cond(= n 0)1(* n((Y g)(- n 1))))
            (- n 1))))
```

$\rightarrow^{*}_{\beta}$ несколько шагов редукции

Таким несколько шагов редукции образом, внутреннее `Y g` конструирует копию функции `g`, в которой вместо `f` взято опять `Y g`, выражение "разворачивается", и `Y g` служет для дальнейших подобных "разворачиваний", по сути – рекурсивных вызовов функции `g`. За счет применения `Y g` получается самокопирующееся выражение, на очередном шаге вычисления которого происходит разворачивание очередной копии тела функции.

Рассмотренный способ используется в лямбда-исчислении для записи рекурсивных функций. В общем случае рекурсивная функция `f` с телом E (им служит лямбда-выражение, содержащее ссылки на `f`) записывается как `Y(λf.E)` – фактически, мы просто приписываем слева комбинатор-выражение `Y`.

В нашем примере для вычисления факториала соответствующим выражением будет:

```
Y(λf.λn.cond(= n 0)1(* n(f(- n 1))))
Factorial = Y g
```

и для вычисления факториала от числа `n` следует записать выражение `(Y g) n`

Приведем пример вычисления этой функции:

`Factorial 3 = (Y g)3` расписать

Один из возможных Y-комбинаторов ввёл Х. Карри:

`Y = λh.(λx.h(x x))(λx.h(x x))`

Он является некоторым обобщением ранее встречавшегося выражения w:

`w = (λx.x x)(λx.x x)`

которое дает бесконечное вычисление (шаги редукции): $w \to_\beta w \to_\beta w \to_\beta w \ldots$

Подобные выражения называются расходящимися.

Докажем, что `Y g = g(Y g)`:

`Y g = (λh.(λx.h(x x))(λx.h(x x)))g` $\to_\beta$

`(λx.g(x x))(λx.g(x x))` $\to_\beta$

`g((λx.g(x x))(λx.g(x x))) = g(Y g)`

Y-комбинатор называется также *комбинатором неподвижной точки*.

*Неподвижной точкой* некоторой функции `f` называется значение `a` из области определения этой функции такое, что `f(a)=a`. Многие математические функции имеют неподвижные точки, например, `0` и `1` являются неподвижными точками функции $f(x)=x^2$; а у тождественной функции `id` – бесконечно много неподвижных точек.

Для функционалов (функций высших порядков), каковыми являются все функции лямбда-исчисления, неподвижными точками являются другие функции `f(g)=g` (`g` – неподвижная точка, сама по себе функция)

Нетрудно заметить, что выражение `Y g` даёт неподвижную точку функции `g`. Тем самым любое определение рекурсивной функции может быть представлено как неподвижная точка соответствующей функции.

Поскольку в лямбда-исчислении каждое выражение может рассматриваться как функция высшего порядка, то существование комбинатора неподвижной точки означает, что у каждой функции есть хотя бы одна неподвижная точка, но некоторые функции могут иметь и несколько различных неподвижных точек.

Рассмотренный Y-комбинатор – не единственный комбинатор неподвижной точки, существует несколько известных определений комбинаторов (а вообще их бесконечно много). Комбинаторы неподвижной точки позволяют определять аннонимные рекурсивные функцииY-комбинатор Карри один из самых известных и самых простых. Он работает только со стратегией НПР, а при АПР даёт бесконечные вычисления (поскольку при этой стратегии выражение `(Y g)` является расходящимся).

Для стратегии вычислений АПР используется чуть более сложный комбинатор неподвижной точки:

`λf.(λx.f(λy.x x y))(λx.f(λy.x x y))`

Возможность реализовать рекурсивные функции в лямбда-вычислении, а также записывать условные выражения означает возможность записи и вычисления любого алгоритма. В теории лямбда-исчисления было доказано, что лямбда-исчисление **алгоритмически полно**. В частности, была доказана полнота по Тьюрингу, означающая, что любой алгоритм, реализуемый МТ, может быть записан в лямбда-исчислении.

Таким образом, лямбда-исчисление может быть рассмотрено как простейший язык программирования, в котором можно представить (выразить) различные структуры и объекты (например, числа, списки, кортежи), причём не единственным образом. Это даёт представление о выразительной силе и вычислительной мощности

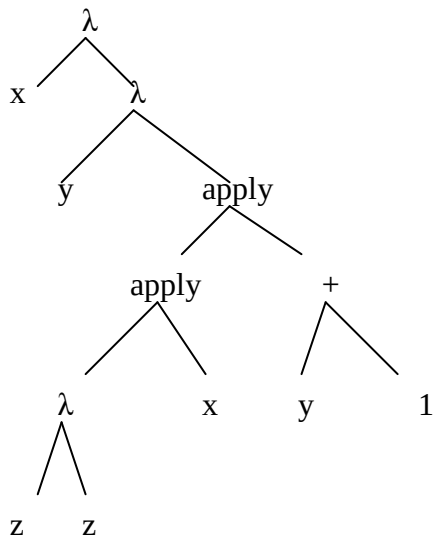
лямбда-исчисления. Тем не менее вычисления в этом простейшем языке программирования явно неэффективны: так, выражение функции `Factorial` в чистом лямбда-исчислении содержит 38 операций абстракции, а вычисление факториала числа 5 потребует более 65 тыс. шагов β-редукции.

**Упражнения к разделу**

1. В выражении `(λx.(λy.*(+ x y)2)) 5 7` расставить скобки и нарисовать АСД
2. Нарисуйте АСД для следующих выражений и укажите, какие вхождения переменных являются свободными, а какие - связанными.
   - а) `(λx.(λz.z)x)(λy.x y)`
   - б) `λx.λy.(λz.z) x (+ y 1)`
   - в) `(λf.λx.λy. f x y)(λz.(λy.z))`
3. Вычислить при разных стратегиях (НПР, АПР) выражения
   - а) `(λx.λy.x(λz.y z))(((λx.λy.y)8)(λx.(λy.y)x))`
   - б) `(λh.(λx.h(x x))(λx.h(x x)))((λx.x)(+1 5))`
4. Проверить работу в лямбда-исчислении условного выражения и дизъюнкции при следующих значениях их аргументов:
   ```
   cond false a b
   or false true
   ```
5. Выразить в чистом лямбда-исчислении логические функции `not` и `xor` (исключающее или), проверить их работу на нескольких наборах возможных значений их аргументов. ()
6. Вычислить для нумералов Черча:
   - а) `succ` $C_2$
   - б) `is_zero` $C_1$

**Решения задач:**

**2б)**




**3а)** `(λx.λy.x(λz.y z))(((λx.λy.y)8)(λx.(λy.y)x))`

3 редекса ?

НПР: `(λy.(((λx.λy.y)8)(λx.(λy.y)x))(λz.y z))`

→ λy.((λy.y)(λx.(λy.y)x))(λz.y z))
→ λy.(λx.(λy.y)x)(λz.y z)) → λy.(λy.y)(λz.y z)
→ λy.(λz.y z)

**3б)** (λh.(λx.h(x x))(λx.h(x x)))((λx.X)(+ 1 5))

АПР: →β (λx.(id(+ 1 5))(x x))(λx.(id(+ 1 5))(x x))

Зацикливание ??

При НПР: подчеркнуть аргументы

(λh.(λx.h(x x))(λx.h(x x)))(λa.λb.a)(+ 1 5)) →β
(λx.((λa.λb.a)(+ 1 5))(x x))) (λx.((λa.λb.a)(+ 1 5))(x x))) →β
(((λa.λb.a)(+ 1 5))(λx.((λa.λb.a)(+ 1 5))(x x)))
(λx.((λa.λb.a)(+ 1 5))(x x))) →β
((λb.(+ 1 5))(λx.((λa.λb.a)(+ 1 5))(x x)))
(λa.((λa.λb.a)(+ 1 5))(x x))) →β
(+ 1 5)(λx.((λa.λb.a)(+ 1 5))(x x)))
(λx.((λa.λb.a)(+ 1 5))(x x))) →β
(+ 1 5)) →δ 6

**5)** not ≡ λx.x false true
xor ≡ λx.λy.x (not y)y

**6а)** (λn.λs.λz.s(n s z))(λs.λz.s(s z)) →
λs.λz.s((λs.λz.s(s z))s z) →
λs.λz.s(λz.s(s z) z) → λs.λz.s(s(s z))