МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение
высшего образования
**«Национальный исследовательский
Нижегородский государственный университет им. Н.И. Лобачевского»
(ННГУ)**

Институт информационных технологий, математики и механики

Кафедра алгебры, геометрии и дискретной математики

Направление подготовки: Программная инженерия
Профиль подготовки: Разработка программно-информационных систем

ОТЧЕТ

по производственной технологической (проектно-технологической) практике
на тему:

«Изучение подстановок в языке со связанными переменными»

Выполнил: студент группы 381808-1
Оганян Роберт Владимирович
Подпись

Научный руководитель:
ст. преп., Ph.D. Макаров Е.М.
Подпись

Нижний Новгород
2021

Содержание

Введение

В наши дни информационные технологии занимают важное место не только в специализированных, но и в повседневных сферах жизни. Технологии прочно вошли в нашу повседневную жизнь, и уже довольно сложно представить современного человека без обработки какой-либо информации на компьютере. Программирование идёт бок о бок с техническим прогрессом и развивается столь же стремительно, как и остальные технологии. Разрабатываются новые языки программирования, дорабатываются старые. На данный момент существует больше сотни различных языков программирования. В чем же их различия? Существует несколько различных классификаций по отдельным признакам, например, разделение языков на императивные и декларативные.

Традиционные языки программирования, такие как Паскаль, Си, Java используют императивный стиль программирования. Программы на таких языках состоят из последовательности модификаций значений некоторого набора переменных, который называется состоянием. В противовес такому методу используются декларативные языки программирования.

Функциональные языки программирования, как подтип декларативных языков, представляют парадигму, в корне отличную от представленной выше модели. Функциональная программа представляет собой некоторое выражение (функцию в математическом смысле), а ее выполнение означает вычисление значения этого выражения.

Отметим некоторые свойства функциональных программ при сравнении функционального и императивного подходов:

- Переменные не изменяют свое значение.
- Не используется оператор присваивания.
- Ни одна команда не может повлиять на выполнение следующей, так что последовательное выполнение команд не несет смысла.
- Нет циклов. Вместо них используются рекурсивные функции.
- Функции используются гораздо более замысловато. Функции можно передавать в другие функции в качестве аргументов и возвращать в качестве результата, и даже в общем случае проводить вычисления, результатом которого будет функция.

В математическом смысле «функции» императивных языков, например, Си не являются функциями, потому что:

- Значение функций может зависеть не только от аргументов

- Два вызова одной и той же функции с одними и теми же аргументами могут привести к неоднозначным результатам
- Как результат возможны различные побочные эффекты (например, изменение значений глобальных переменных)

Функциональный подход имеет ряд преимуществ перед императивным. Функциональные программы соответствуют математическим объектам, что позволяет проводить строгие суждения. Узнать значение императивной функции может оказаться довольно затруднительным, в отличие от функциональной программы, которая может вывести значение практически непосредственно. Из факта отсутствия побочных эффектов у функциональных программ следует, что порядок вычисления ее подвыражений не оказывает влияния на результат. Таким образом, функциональные программы легко поддаются распараллеливанию.

Целью данной курсовой работы является изучение теоретической базы, необходимой для понимания принципов функционального программирования, и практической реализации интерпретатора бестипового лямбда-исчисления.

Основная часть

Теоретическое обоснование проблемы

Функциональное программирование сильно опирается на математическую систему, называющуюся лямбда-исчислением. Лямбда исчисление — это формальная система в математической логике для описания вычислений с помощью абстракции и аппликации функций, использующая связывание переменных и подстановку. Придумана в 1930-х годах Алонзо Черчем.

Для начала рассмотрим основы лямбда-исчисления.

Понятие лямбды-терма без типа

Лямбда-исчисление основано на понятии лямбда-терма, составляемого из переменных и некоторого фиксированного набора констант с использованием операции применения функции, и лямбда-абстрагирования. То есть все лямбда-выражения (лямбда-терма) можно разделить на следующие категории:

- Переменные: обозначаются произвольными строками, составленными из букв и цифр.
- Константы: также обозначаются строками; в отличие от переменных будем определять из контекста.
- Комбинации (аппликации): т.е. применения функции S к аргументу T; и S и T могут быть произвольными лямбда-термами. Комбинация записывается как ST
- Абстракции произвольного лямбда-терма S по переменной x, обозначаемые как $\lambda x.S$

Весь синтаксис лямбда-выражений описывается очень простой формулой БНФ (форма Бэкуса - Наура):

$$Exp ::= Var \mid Const \mid (Exp_1\ Exp_2)\ \mid \lambda\, Var\,.Exp\ ,$$

где *Exp* – выражение, *Var* – переменная, *Const* – константа.

Пример лямбда-терма: константная функция $(\lambda x.(\lambda y.x))$. Наш терм принимает переменную x и возвращает другой терм (функцию $(\lambda y.x)$). Эта функция принимает y, а возвращает x. В Haskell мы бы написали это так: $\backslash x -> (\backslash y -> x)$.

Вхождение переменной x называется **связанным**, если оно находится в теле t абстракции $\lambda x.t.$ (то есть если оно связано этой абстракцией). Вхождение x **свободно**, если оно не связано никакой вышележащей абстракцией переменной x. Переменная называется свободной в терме, если она имеет свободное вхождение в этом терме. Аналогично переменная называется связанной в терме, если она имеет связанное вхождение в этом терме. Множество переменных, свободных в t, обозначается $FV(t)$.

Например, вхождение x в xy – свободно; вхождение x в $\lambda y.xy$ – свободно; вхождение x в $\lambda x.x$ – связано; вхождение x в $\lambda z.\lambda x.\lambda y.x(yz)$ связаны.

Подстановки

Подстановка — это процесс замены всех свободных вхождений переменной в выражение другим выражением. Интуитивно ясно, что применение терма $\lambda x.S$ как функции к аргументу T дает в результате терм S, в котором все свободные вхождения переменной x заменены на T. Будем обозначать операцию подстановки терма S вместо переменной x в другом терме T как $T[x := S]$. Трудность подстановки состоит в том, что необходимо накладывать дополнительные ограничения, позволяющие избегать конфликта в именах переменных.

Определения подстановок:

- $x[x \coloneqq R] = \mathrm{R}$
- $y[x \coloneqq \mathrm{R}] = y$ при условии, что $x \neq y$
- $(\mathrm{TS})[x \coloneqq \mathrm{R}] = (T[x \coloneqq R])(\mathrm{S}[x \coloneqq \mathrm{R}])$
- $(\lambda x.S)[x := T] = \lambda x.S$
- $(\lambda y.S)[x := T] = \lambda y.(S[x := T])$, если $x \neq y$ и $y \notin FV(T)$

α – преобразование

При подстановке важно соблюсти критерий свежести**.** Переменная, которую мы заменяем, должна находиться в свободных переменных терма. Иначе мы изменим смысл функции.

Пример подобной ошибки: $(\lambda x.y)[y := x] = (\lambda x.(y[y := x])) = \lambda x.x$.

y была свободной переменной, но после данной подстановки терм стал замкнутым, тем самым изменилось поведение функции. Это может быть исправлено с помощью переименования (альфа-преобразованием).

Альфа-преобразование (альфа-конверсия) является вспомогательным механизмом для того, чтобы изменять имена связанных переменных. Лямбда-выражение (в данном случае это обязательно абстракция), к которому применяется альфа-конверсия, называется альфа-редексом. Правило α-преобразования всего лишь говорит о том, что в альфа-редексе связанные переменные могут быть переименован, если это не приводит к конфликту имен. То есть, например, $\lambda x.S \xrightarrow{a} \lambda y.S[x := y]$ при условии, что $y \notin FV(S)$.

В дальнейшем альфа-преобразования могут обозначаться как $\xrightarrow{a}$.

Исправим ошибку в рассмотренном выше пример с помощью альфа-преобразования:

$$(\lambda z.y)[y := x] = \lambda z.(y[y := x]) = \lambda z.x$$

Теперь в наших аргументах нет “коллизии имен”.

$\boldsymbol{\beta}$- редукция

Процесс подстановки аргументов в функции называется **β – редукцией (β - конверсией).** В качестве β – редекса может выступать только комбинации. Процедура β – конверсии аналогична вызову функции в языке программирования: тело *E* функции *λV.E* “выполняется” в ситуации, когда “формальному параметру” *V* присвоено значение “актуального параметра” *E*\`. То есть, формально говоря, бета-редукция позволяет нам избавиться от связанной переменной (а следовательно, если нет других переменных, то и от *λ*).

В дальнейшем бета-редукция может обозначаться как $\xrightarrow{B}$.

Что бы проиллюстрировать работу бета-редукции, рассмотрим следующий пример:

$$((\lambda x.\lambda y.x)y)z$$

Далее приведены шаги бета-редукции:

1. $((\lambda y.x)[x := y])z$ - заменяем x на y в теле $\lambda y.x$
2. $((\lambda y'.x)[x := y])z$ - после альфа-редукции
3. $(\lambda y'.y)z$ - первая бета-редукция завершена
4. $y[y' := z]$ - замена y' на z в y
5. y - вторая бета-редукция завершена

Существует ещё одна операция под названием бета-расширение, которую мы так же можем использовать. По определению, лямбда-выражение e_1 бета-расширяется в e_2, если e_2 бета-редуцируется до e_1.

Комбинаторы

Терм без свободных переменных называется **замкнутым**; замкнутые термы называют также **комбинаторами**. Рассмотрим некоторые из них:

1. **Тождественный комбинатор** $\boldsymbol{I = \lambda x. x}$ (в некоторых источниках носит название id); не выполняет никаких действий, а просто возвращает свой аргумент.
2. **Комбинатор – вычеркиватель** $\boldsymbol{K = \lambda x. \lambda y. x}$;
3. **Комбинатор – распределитель** $\boldsymbol{S = \lambda f. \lambda g. \lambda x. fx(gx)}$;
4. **Комбинатор омега** $\boldsymbol{\Omega = (\lambda x. xx)(\lambda x. xx)}$; используется для иллюстрации возможности бесконечных вычислений, так как у данного комбинатора отсутствует нормальная форма. **Комбинатор омега** можно обобщить до полезного терма, который называется **комбинатором неподвижной точки**.
5. **Комбинаторы неподвижной точки**. Например, комбинатор $\boldsymbol{Y = \lambda h. (\lambda x. h(xx))(\lambda x. h(xx))}$ (комбинатор Хаскелла Карри); с его помощью можно определять рекурсивные функции.
6. **Комбинатор – композитор** $\boldsymbol{B = \lambda f. \lambda g. \lambda x. f(gx)}$
7. **Комбинатор – перестановщик** $\boldsymbol{C = \lambda f. \lambda x. \lambda y. fyx}$
8. **Комбинатор – дубликатор** $\boldsymbol{W = \lambda x. \lambda y. xyy}$

Согласно определению выше, комбинаторы редуцируются следующим образом.

1. $Sabc \overset{B}{\rightarrow} ac(bc)$
2. $Kab \overset{B}{\rightarrow} a$
3. $Ia \overset{B}{\rightarrow} a$
4. $Babc \overset{B}{\rightarrow} a(bc)$
5. $Cabc \overset{B}{\rightarrow} acb$
6. $Wab \overset{B}{\rightarrow} abb$
7. Пример использования Y-комбинатора: Функция, считающая значение факториала:
   $\boldsymbol{fact = fix\ fact'}$, где
   $\boldsymbol{fix = \lambda f. \big(\lambda x. f(xx)\big)\big(\lambda x. f(xx)\big),}$
   $\boldsymbol{fact' = \lambda f\ .\ \lambda x\ .\ if\ (isZero\ x)\ \overline{1}\ (mult\ x\ (f\ (pred\ x)))}$

На самом деле, не обязательно определять все комбинаторы. В качестве базиса в комбинаторной логике можно использовать наборы комбинаторов:

- K, S
- I, B, C, S
- B, W, K.

Остальные комбинаторы можно выразить через другие. Например, $I = SKK$.

Нормальная форма

Несмотря на то, что термин «редукция» подразумевает уменьшение размера лямбда-терма, в действительности это может быть не так. Например: $(\lambda x.xx)(\lambda x.xx) \xrightarrow{B} (\lambda x.xx)(\lambda x.xx) \xrightarrow{B} (\lambda x.xx)(\lambda x.xx)$ ...

Тем не менее отношение редукции соответствует систематическим попыткам вычислить терм, последовательно вычисляя комбинации f(x), где f-некоторая лямбда-абстракция. Когда для терма невозможно сделать никакую редукцию, за исключением α-преобразования, будем говорить, что терм находится в **нормальной форме**. Также, лямбда-выражение имеет нормальную форму, если существует последовательность бета-редукций и/или бета-расширений, которые приводят к нормальной форме.

Отсюда получаем несколько утверждений о нормальной форме:

1. Не каждое лямбда-выражение имеет нормальную форму (например, $(\lambda z.zz)(\lambda z.zz)$).
2. Если лямбда-выражение имеет нормальную форму, мы можем перейти к ней используя только бета-редукцию (следствие из теоремы Чёрча-Россера).
3. Не любая последовательность редукций приведет выражение в нормальную форму.
4. Существует стратегия для выбора бета-редукций, которая точно позволит прийти к нормальной форме – редукция нормального порядка.

Примеры приведения терма к нормальной форме

1. $(\lambda x.xy)(\lambda u.vuu) \xrightarrow{B} xy[x \coloneqq (\lambda u.vuu)] \equiv (\lambda u.vuu)y \xrightarrow{B} vuu[u \coloneqq y] \equiv vyy$
2. $(\lambda xy.yx)uv \equiv (\lambda x.(\lambda y\,.yx)u)v \xrightarrow{B} \big((\lambda y\,.yx)[x \coloneqq u]\big)v \equiv (\lambda y\,.yu)v \xrightarrow{B} yu[y \coloneqq v] \equiv vu$
3. $\big(\lambda x.x\big(x(yz)\big)x\big)(\lambda u.uv) \xrightarrow{B} \big(x\big(x(yz)\big)x\big)[x \coloneqq \lambda u.uv] \equiv$ $(\lambda u.uv)\big((\lambda u.uv)(yz)\big)(\lambda u.uv) \xrightarrow{B} (\lambda u.uv)\big((uv)[u \coloneqq yz]\big)(\lambda u.uv) \equiv$ $(\lambda u.uv)\big((yz)v\big)(\lambda u.uv) \xrightarrow{B} \big((uv)[u \coloneqq yz]\big)v(\lambda u.uv) \equiv ((yz)v)v(\lambda u.uv)$
4. $(\lambda x.xxy)(\lambda y.yz) \xrightarrow{B} (xxy)[x \coloneqq \lambda y.yz] \equiv (\lambda y.yz)(\lambda y.yz)y \xrightarrow{B} (yz)[y \coloneqq \lambda y.yz]y \equiv$ $\big((\lambda y.yz)z\big)y) \xrightarrow{B} (yz)[y \coloneqq z]y \equiv (zz)y \equiv zzy$
5. $(\lambda xy.xyy)(\lambda u.uyx) \equiv \big(\lambda x.(\lambda y.xyy)\big)(\lambda u.uyx) \xrightarrow{B} (\lambda y.xyy)[x \coloneqq \lambda u.uyx] \xrightarrow{a}$ $\xrightarrow{a} (\lambda z.xzz)[x \coloneqq \lambda u.uyx] \equiv \lambda z.\Big(\big((\lambda u.uyx)z\big)z\Big) \xrightarrow{B} \lambda z.\big((uyx)[u \coloneqq z]z\big) \equiv \lambda z.zyxz$

6. $(\lambda xy.x)(\lambda x.x) \equiv \big(\lambda x.(\lambda y.x)\big)(\lambda x.x) \overset{B}{\to} (\lambda y.x)[x \coloneqq \lambda x.x] \equiv \lambda y.(\lambda x.x) \equiv \lambda yx.x$

7. $(\lambda x.xx)(\lambda yz.yz) \equiv (\lambda x.xx)\big(\lambda y.(\lambda z.yz)\big) \overset{B}{\to} (\lambda x.xx)[x \coloneqq \lambda y.(\lambda z.yz)] \equiv$

$\equiv \big(\lambda y.(\lambda z.yz)\big)\big(\lambda y.(\lambda z.yz)\big) \overset{B}{\to} (\lambda z.yz)[y \coloneqq \lambda y.(\lambda z.yz)] \equiv$

$\equiv \lambda z.(\lambda y.(\lambda z.yz)z) \overset{B}{\to} \lambda z.\big((\lambda z.yz)[y \coloneqq z]\big) \overset{a}{\to} \lambda z.\big((\lambda z'.yz')[y \coloneqq z]\big) \equiv$

$\equiv \lambda z.(\lambda z'.zz') \equiv \lambda zz'.zz'$

8. $(\lambda xyz.xzy)(\lambda xy.x) \equiv \Big(\lambda x.\big(\lambda y.(\lambda z.xzy)\big)\Big)(\lambda xy.x) \overset{a}{\to} \Big(\lambda x.\big(\lambda y.(\lambda z.xzy)\big)\Big)(\lambda uv.u) \overset{B}{\to}$

$\overset{B}{\to} \big(\lambda y.(\lambda z.xzy)\big)[x \coloneqq \lambda uv.u] \equiv \lambda y.(\lambda z.(\lambda uv.u)zy) \equiv$

$\equiv \lambda y.\big(\lambda z.\big((\lambda u.(\lambda v.u))z\big)y\big) \overset{B}{\to}$

$\overset{B}{\to} \lambda y.(\lambda z.((\lambda v.u)[u \coloneqq z]y) \equiv \lambda y.(\lambda z.((\lambda v.z)y) \overset{B}{\to} \lambda y.(\lambda z.(z[v \coloneqq y])) \equiv$

$\equiv \lambda y.(\lambda z.z) \equiv \lambda yz.z$

9. $\lambda xy.\big(\lambda z.(\lambda x.zx)(\lambda y.zy)\big)(xy) \overset{B}{\to} \lambda xy.\Big(\big((\lambda x.zx)(\lambda y.zy)\big)[z \coloneqq xy]\Big) \overset{a}{\to}$

$\overset{a}{\to} \lambda xy.\Big(\big((\lambda x'.zx')(\lambda y'.zy')\big)[z \coloneqq xy]\Big) \equiv \lambda xy.\big((\lambda x'.xyx')(\lambda y'.xyy')\big) \overset{B}{\to}$

$\overset{B}{\to} \lambda xy.\big((xyx')[x' \coloneqq \lambda y'.xyy']\big) \equiv \lambda xy.(xy\lambda y'.xyy') \equiv \lambda xy.(xy\lambda y'.xyy')$

10. $\big(\lambda xyz.xz(yz)\big)\big((\lambda xy.yx)u\big)\big((\lambda xy.yx)v\big)w \overset{a}{\to}$

$\overset{a}{\to} \big(\lambda xyz.xz(yz)\big)\big((\lambda x'y'.y'x')u\big)\big((\lambda x''y''.y''x'')v\big)w \overset{a,B}{\longrightarrow}$

$\overset{a,B}{\longrightarrow} (\lambda yz.((\lambda x.(\lambda y'.y'x))u)z(y\,z)))\ ((\lambda x'.(y''y''x'))v)w \overset{a,B}{\longrightarrow}$

$\overset{a,B}{\longrightarrow} \left(\lambda z.\Big(\big(\lambda x.(\lambda y.yx)\big)u\Big)\ z\ \Big(\Big(\big(\lambda x'.(\lambda y'.y'x')\big)v\Big)z\Big)\right)w \overset{B}{\to}$

$\overset{B}{\to} \Big(\big(\lambda x.(\lambda y.yx)\big)u\Big)\ w\ \Big(\Big(\big(\lambda x'.(\lambda y'.y'x')\big)v\Big)w\Big) \overset{a,B}{\longrightarrow}$

$\overset{a,B}{\longrightarrow} (\lambda y.yu)\ w\ \Big(\Big(\big(\lambda x.(\lambda y'.y'x)\big)v\Big)w\Big) \overset{a,B}{\longrightarrow} w\ u\ \Big(\Big(\big(\lambda x.(\lambda y.yx)\big)v\Big)w\Big) \overset{B}{\to}$

$\overset{B}{\to} w\ u\ \Big(\big((\lambda y.yv)\big)w\Big) \overset{B}{\to} w\ u\ (w\ v)$

Булевские константы и числа Чёрча

В чистом бестиповом лямбда-исчислении отсутствует всё, кроме функций. Таким образом, такие базовые вещи как числа или булевы значения нужно реализовывать самостоятельно. Вернее сказать, необходимо создать некие активные сущности, которые будут вести себя подобно необходимым нам объектам. Процесс кодирования, естественно, будет заключаться в написании соответствующих функций. Так выглядят термы, воплощающие поведения True и False:

- $\boldsymbol{tru = \lambda t.\lambda f.t}$ - функция двух аргументов, которая всегда возвращает первый аргумент.
- $\boldsymbol{fls = \lambda t.\lambda f.f}$ – функция двух аргументов, которая всегда возвращает второй аргумент.

Под такие булевы константы оператор $\boldsymbol{if}$ имеет следующий вид:

$$\boldsymbol{if = \lambda b.\lambda x.\lambda y.bxy},$$

где *b* — *tru* или *fls*; x — ветка *then*; *y* — ветка *else*.

Рассмотрим пример работы с булевыми константами. Выражение $\boldsymbol{if\ fls\ t\ e}$ должно возвращать нам тело *else*-ветки, то есть *e*. Проверим это:

$$\boldsymbol{if\ fls\ t\ e} \equiv (\lambda b.\lambda x.\lambda y.bxy)fls\ t\ e \xrightarrow{B} \big((\lambda x.\lambda y.bxy)[b \coloneqq fls]\big)te \equiv$$

$$\equiv (\lambda x.\lambda y.flsxy)te \xrightarrow{B} \big((\lambda y.flsxy)[x \coloneqq t]\big)e \equiv (\lambda y.flsty)e \xrightarrow{B} flsty[y \coloneqq e] \equiv$$

$$\equiv fls\ t\ e \equiv \Big(\big((\lambda t.\lambda f.f)t\big)e\Big) \xrightarrow{B} \big((\lambda f.f)[t \coloneqq t]\big)e \equiv (\lambda f.f)e \xrightarrow{B} f[f \coloneqq e] \equiv \boldsymbol{e}$$

Перейдем к представлению натуральных чисел в виде лямбда-термов (к **числам Чёрча**). В основе реализации по-прежнему будут лежать функции, ведущие себя в заданном контексте подобно нулю, единице, двойке, тройке и т.п.

Итак, нам потребуется функция, принимающая два аргумента: фиксированное начальное значение и функцию следования (функцию для определения следующего элемента). Число будет закодировано в количестве применений функции следования к начальному значению:

- $\boldsymbol{\overline{0} = \lambda s.\lambda z.z}$ - функция *s* применяется к начальному значению *z* нуль раз.
- $\boldsymbol{\overline{1} = \lambda s.\lambda z.sz}$ функция *s* применяется к начальному значению *z* один раз.
- $\boldsymbol{\overline{2} = \lambda s.\lambda z.s(sz)}$ функция *s* применяется к начальному значению *z* два раза.
- $\boldsymbol{\overline{3} = \lambda s.\lambda z.s(s(sz))}$ функция *s* применяется к начальному значению *z* три раза.

И так далее. Отметим, что **константы и числа Чёрча** являются комбинаторами.

Функция следования прибавляет к заданному числу единицу. То есть в качестве аргумента функция принимает значение, которое требуется увеличить, и которое в свою очередь тоже является функцией двух аргументов. Таким образом, суммарно функция следования (+1) имеет три аргумента: предшествующее число Чёрча *n*; функция, которую надо применить *n+1* раз к начальному значению; само начальное значение:

$$\boldsymbol{scc = \lambda n.\lambda s.\lambda z.s(nsz)},$$

где (nsz) — n раз применённая к z функция s. Но так как нам нужно применить её $n+1$ раз, получаем $s(nsz)$.

Рассмотрим пример работы с числами Чёрча. Пусть нам требуется получить из единицы ($\boldsymbol{one = \lambda s.\lambda z.sz}$) двойку ($\boldsymbol{two = \lambda s.\lambda z.s(sz)}$). Распишем шаги:

$\boldsymbol{scc\ one\ sz} \xrightarrow{a} scc\ one\ s'z' \equiv \big(\lambda n.\lambda s.\lambda z.s(nsz)\big)one\ s'z' \xrightarrow{B}$

$\xrightarrow{B} \Big(\big(\lambda s.\lambda z.s(nsz)\big)[n \coloneqq one]\Big)s'z' \equiv \big(\lambda s.\lambda z.s(one\ sz)\big)s'z' \xrightarrow{B}$

$\xrightarrow{B} (\big(\lambda z.s(one\ sz)\big)[s \coloneqq s'])z' \equiv \big(\lambda z.s'(one\ s'z)\big)z' \xrightarrow{B} s'(one\ s'z)[\,z \coloneqq z'\,] \equiv$

$\equiv s'(one\ s'z') \equiv s'\big((\lambda s.\lambda z.sz)s'z'\big) \xrightarrow{B} s'\big(((\lambda z.sz)[s \coloneqq s'])z'\big) \equiv s'(\lambda z.s'z)z' \xrightarrow{B}$

$\xrightarrow{B} s'(s'z)[z \coloneqq z'] \equiv s'(s'z') \equiv \boldsymbol{two\ s'z'}$

Логические операции и пары

Аналогично true и false, можно вывести функции для других логических операторов:

- $\boldsymbol{true = \lambda t.\lambda f.t}$ - функция двух аргументов, которая всегда возвращает первый аргумент.
- $\boldsymbol{false = \lambda t.\lambda f.f}$ – функция двух аргументов, которая всегда возвращает второй аргумент.
- $\boldsymbol{if = \lambda b.\lambda x.\lambda y.bxy}$ – условный оператор, где b - true или false; x — ветка then; y — ветка else.
- $\boldsymbol{and = \lambda n.\lambda m.if\ n\ m\ false}$ – логический «И».
- $\boldsymbol{or = \lambda n.\lambda m.if\ n\ true\ m}$ – логическое «ИЛИ».
- $\boldsymbol{not = \lambda b.if\ b\ false\ true}$ – логическое отрицание.
- $\mathbf{isZero = \lambda n.n(\lambda c\,.false)\ true}$ – функция проверки, является ли n нулем.
- $\boldsymbol{pair = \lambda a.\lambda b.\lambda t.t\ a\ b}$ - функция pair, принимающая два значения и запаковывающая их в пару так, чтобы к ним можно было обращаться по first и second:

$$\boldsymbol{first = \lambda p.p\ true}$$

$$\boldsymbol{second = \lambda p.p\ false}$$

Применение: $\boldsymbol{first(pair\ a\ b)} = \ldots = \boldsymbol{a}$

Арифметические операции

Как уже было сказано раннее, в чистом бестиповом лямбда-исчислении отсутствует всё, кроме функций. Соответственно арифметические операции мы также должны реализовывать самостоятельно. Пример некоторых из них:

- **+1.**

  Эту функцию мы уже определили раннее:

  $$\boldsymbol{scc = \lambda n. \lambda s. \lambda z. s(nsz)}$$

- **Сложение.**

  Сложение двух чисел похоже на прибавление единицы. Только в этом случае надо прибавить не единицу, а второе число:

  $$\boldsymbol{plus = \lambda n. \lambda m. \lambda s. \lambda z. ns(msz)}$$

  Например, $\left(plus\ \overline{3\ 3}\right)(+1)0 \equiv 6$

- **Умножение.**

  $$\boldsymbol{mult = \lambda n. \lambda m. \lambda s. \lambda z. n(ms)z \xrightarrow{B} \lambda n. \lambda m. \lambda s. n(ms)}$$

  Например, $(mult\ \overline{3}\ \overline{3})\ (+1)\ 0 \equiv 9$

- **Возведение в степень**.

  $$\boldsymbol{power = \lambda n\ . \lambda m\ . \lambda s\ . \lambda z\ . m\ n\ s\ z}$$

  Например, $(power\ \overline{3}\ (succ\ \overline{3}))\ (+1)\ 0 \equiv 81$

Нотация де Брауна

До сих пор в качестве связанных переменных мы использовали именованные переменные. Такой подход имеет свои минусы. Например, при реализации исчисления программно, для каждого терма необходимо единое представление, однако мы не запрещаем одинаковые имена в разных абстракциях! Есть несколько способов решить эту и другие проблемы, один из таких – использование **нумерации де Брауна**.

Идея **де Брауна** состоит в том, чтобы представлять термы таким образом, чтобы обеспечить вхождения переменных прямыми указаниями на их связывающие определения вместо того, чтобы называть их по имени. Это достигается заменой именованных переменных натуральными числами (**индексами де Брауна**) так, чтобы число k означало «переменная, связанная *k-й* охватывающей *λ*». Например:

Стандартная форма	Нотация де Брауна
$\lambda x. x$	$\lambda. 0$

$\lambda x. \lambda y. x$	$\lambda. \lambda. 1$
$\lambda x. \lambda y. \lambda s. \lambda z. x\ s\ (y\ s\ z)$	$\lambda. \lambda. \lambda. \lambda. 3\ 1(2\ 1\ 0)$
$(\lambda x. x\ x)(\lambda x. x\ x)$	$(\lambda. 0\ 0)(\lambda. 0\ 0)$
$(\lambda x. \lambda x. x)(\lambda y. y)$	$(\lambda. \lambda. 0)(\lambda. 0)$

Таблица 1. Примеры нотации де Брауна

В **нотациях де Брауна** вместо имени переменной хранится натуральное число — количество абстракций в дереве разбора, на которое нужно подняться, чтобы найти лямбду, с которой связана текущая переменная.

Важная особенность **индексов де Брауна** - всякий замкнутый терм имеет ровно одно представление. Тогда два терма в обычном представлении эквивалентны с точностью до переименования связанных переменных тогда и только тогда, когда они идентичны в представлении де Брауна.

Со связанными переменными разобрались. Но что делать с термами, содержащими свободные переменные? Для них нам потребуется понятие **контекста именования**.

Контекст именования – биекция из множества имен переменных в множество натуральных чисел. Например, нам нужно представить $\lambda x. yx$ в виде безымянного терма. С x все понятно, это связанная переменная. А для y неизвестно, как «далеко» эта переменная будет определена, и какой ей сопоставить номер. Для решения этой ситуации надо раз и навсегда выбрать присвоение индексов **де Брауна** свободным переменным (определить **контекст именования**), и последовательно использовать это присвоение, когда требуется выбрать номер для свободной переменной.

Рассмотрим примеры со следующим контекстом именования:

$$\Gamma = \{x \mapsto 4, y \mapsto 3, z \mapsto 2, a \mapsto 1, b \mapsto 0\}$$

Тогда $x\ (y\ z)$ будет представлен как $4\ (3\ 2)$; $\lambda w. yw$ будет представлен как $\lambda. 4\ 0$; $\lambda w. \lambda a. x$ — как $\lambda. \lambda. 6$

Сдвиг, подстановка, редукция.

Теперь необходимо определить операцию подстановки ($[k \rightarrow s]t$) на безымянных термах. Для этого нам потребуется вспомогательная операция, называемая **«сдвигом»**, перенумеровывающая индексы свободных переменных в терме.

Сдвиг терма t на d позиций с отсечкой c, обозначаемый $\uparrow_c^d (t)$ определяется так:

$$\uparrow_c^d (k) = \begin{cases} k, \text{если } k < c \\ k + d, \text{если } k \geq c \end{cases}$$

$$\uparrow_c^d (\lambda. t_1) = \lambda. \uparrow_{c+1}^d (t_1)$$

$$\uparrow_c^d (t_1\, t_2) = \uparrow_c^d (t_1)\ \uparrow_c^d (t_2)$$

Когда подстановка проникает внутрь лямбда-абстракции, например, *[1 ↦ s] (λ.2)* (или *[x ↦ s] (λy.x)*, предполагая, что 1 - индекс x во внешнем контексте), длина контекста, в котором происходит подстановка, становится больше длины исходного на одну переменную. Тогда получается, что нам требуется увеличить индексы свободных переменных в s, чтобы в новом контексте они ссылались на те же переменные, что и раньше. Однако нельзя просто увеличить на единицу все индексы переменных s, так как при этом сдвинулись бы и связанные переменные внутри s. Функция сдвига должна работать по описанным в определении выше правилам. Она принимает параметр «отсечки» c, управляющий тем, какие переменные сдвигаются. Изначально он равен 0 (говорит о том, что сдвигать нужно все переменные), и увеличивается каждый раз, когда функция сдвига пересекает границу абстракции. Таким образом, при вычислении $\uparrow_c^d (t)$ мы знаем, что терм t происходит изнутри c слоев абстракции по отношению к исходному аргументу $\uparrow^d$ (запись $\uparrow^d$ означает $\uparrow_0^d$). Получается, что все идентификаторы $k < c$ внутри t связаны в исходном аргументе и их сдвигать не нужно, а идентификаторы $k \geq c$ свободны, и подлежат сдвигу.

Теперь дадим определение оператора подстановки *[j ↦ s] t*. Когда мы используем подстановку, мы обычно подставляем последнюю переменную в контексте (т. е., j = 0), так как для того, чтобы определить бета-редукцию, нам нужен этот случай. Но для того, чтобы подставить значение переменной = 0 в терме, являющимся лямбда-абстракцией, нужна возможность подстановки значения переменной = 1 в теле этой абстракции. Таким образом, определение подстановки должно работать с произвольной переменной. Получаем следующее определение подстановки.

Подстановка терма s вместо переменной номер j в терме, записываемая в виде *[j → s] t*, определяется следующим образом:

$$[j \rightarrow s]k = \begin{cases} s, & \text{если } k = j \\ k, & \text{иначе} \end{cases}$$

$$[j \rightarrow s]\,(\lambda.t_1) = \lambda.[\mathrm{j}+1 \rightarrow \uparrow^1 s]t_1$$

$$[j \rightarrow s]\,(t_1\ t_2) = ([j \rightarrow s]t_1\ \ [j \rightarrow s]t_2)$$

Наконец, для вычислений на безымянных термах нам необходимо изменить правило **бета-редукции** так, чтобы оно использовало нашу новую операцию подстановки.

Во-первых, необходимо учесть, что редукция редекса «расходует» связанную переменную. То есть в результате подстановки необходимо переименовать переменные, чтобы отразить тот факт, что «расходуемая» переменная больше не является частью контекста. Например, $(\lambda.\,1\ 0\ 2)(\lambda.\,0) \xrightarrow{B} 1\ (\lambda.\,0)2$ – неверно. $(\lambda.\,1\ 0\ 2)(\lambda.\,0) \xrightarrow{B} 0\ (\lambda.\,0)1$ – верно, мы не забыли переименовать переменные после "расхода" переменной.

Во-вторых, для $((\lambda x.\,t_{12})v_2\,)$ требуется сдвинуть переменные в v_2 перед подстановкой в t_{12}, поскольку терм t_{12} определен в более крупном контексте, чем v_2 .

Собирая все эти соображения вместе, получаем такое правило **бета-редукции:**

$$\boldsymbol{(\lambda.\,t_{12})\,v_2 \ \rightarrow\ \uparrow^{-1}([0 \rightarrow \uparrow^{1}(v_2\,)]\ t_{12})}$$

Практическая часть

Для реализации интерпретатора лямбда-исчислений на безымянных термах был выбран язык программирования *C++14*. Была использована среда разработки *JetBrains CLion*.

Демонстрация работы

Запустим интерпретатор.

```
Welcome to beta-reduction calculator. Choose the option:
1. Type 'rules' to read rules for syntax
2. Type 'test' to run implemented tests
3. Type the lambda term to reduce it
4. Type 'exit' to quit the program
>
```

Рисунок 1. Стартовое окно

Запросим синтаксические правила.

```
>rules
 -+-+-+-+-+-+-+-+-+-+  Rules  -+-+-+-+-+-+-+-+-+-+-+-+-+
1. Capital letters are only used for Library Functions such as True, False, Not ...
2. Variables can be named only with cursive letters or words fully consisting of cursive letters
3. Do not mess up with brackets
4. Lambda is '\x'; Expression splits with spaces, for example '(\x x y) a -> a y'
5. There are some checks for correct syntax, but not for every possible mistake. Do your best to not make any mistakes
>
```

Рисунок 2. Вывод списка грамматических правил

Запросим тестирование для написанных разработчиком тестов, чтобы убедиться в корректности работы алгоритмов.

```
>test
 -+-+-+-+-+-+-+-+-+-+  Running tests  -+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
Correct! (\x \y x y y)(\u u y x) reduced correctly to \y y y' x y
Correct! (\x x x)(\y \z y z) reduced correctly to \z (\z' z z')
Correct! \x \y (\z (\x z x) (\y z y)) (x y) reduced correctly to \x (\y (x y) (\y' (x y) y'))
Correct! Omega a reduced correctly to Term can't be reduced
Correct! (\x x (x (y z)) x) (\u u v) reduced correctly to ((y z) v) v (\u u v)
Correct! First (Pair a b) reduced correctly to a
Correct! (\f \x f (f x)) (\x x) reduced correctly to \x x
Correct! (\x \y \z x z (y z))((\x \y y x) u) ((\x \y y x) v) w reduced correctly to w u (w v)
Correct! (\x \y \z x z y)(\x \y x) reduced correctly to \y (\z z)
Correct! (\b \x \y b x y) False t e reduced correctly to e
Tests passed successfully!
```

Рисунок 3. Результаты тестирования

Запросим редукцию для какого-нибудь тривиального терма. Например, для разобранного раннее в отчете терма: $(\lambda xy.x)(\lambda x.x) \equiv \big(\lambda x.(\lambda y.x)\big)(\lambda x.x) \xrightarrow{B} (\lambda y.x)[x \coloneqq \lambda x.x] \equiv \lambda y.(\lambda x.x) \equiv \lambda yx.x$. Проверим, так ли это:

```
>(\x \y x)(\x x)
  -> (\x (\y x)) (\x' x')
 -> \y (\x x)
```

Рисунок 4. Редукция терма (λxy.x)(λx.x)

Верно. Попробуем ввести нетривиальный терм, разобранный раннее в отчете: $\big(\lambda xyz.xz(yz)\big)\big((\lambda xy.yx)u\big)\big((\lambda xy.yx)v\big)w$. Должно получиться $w\ u\ (w\ v)$.

```
>(\x \y \z x z (y z))((\x \y y x) u) ((\x \y y x) v) w
  -> (\x (\y (\z x z (y z)))) ((\x' (\y' y' x')) u) ((\x'' (\y'' y'' x'')) v) w
 -> (\y (\z ((\x (\y' y' x)) u) z (y z))) ((\x' (\y'' y'' x')) v) w
 -> (\z ((\x (\y y x)) u) z (((\x' (\y' y' x')) v) z)) w
 -> ((\x (\y y x)) u) w (((\x' (\y' y' x')) v) w)
 -> (\y y u) w (((\x (\y' y' x)) v) w)
 -> w u (((\x (\y y x)) v) w)
 -> w u ((\y y v) w)
 -> w u (w v)
>
```

Рисунок 5. Редукция терма (λxyz.xz(yz))((λxy.yx)u)((λxy.yx)v)w

Верно. Теперь попробуем 1 умножить на 2 и прибавить к этому 0. Должны получить 2.

```
>(Plus Zero (Mult One Two))
  -> (\n (\m (\s (\z (n s (m s z)))))) (\s' (\z' z')) ((\n' (\m' (\s'' n' (m' s'')))) (\s''' (\z'' s''' z'')) (\s'''' (
\z''' s'''' (s'''' z'''))))
 -> (\m (\s (\z ((\s' (\z' z')) s (m s z))))) ((\n (\m' (\s'' n (m' s'')))) (\s''' (\z'' s''' z'')) (\s'''' (\z''' s''''
 (s'''' z'''))))
 -> \s (\z ((\s' (\z' z')) s (((\n (\m (\s'' n (m s'')))) (\s''' (\z'' s''' z'')) (\s'''' (\z''' s'''' (s'''' z''')))) s
 z)))
 -> \s (\z ((\z' z') (((\n (\m (\s' n (m s')))) (\s'' (\z'' s'' z'')) (\s''' (\z''' s''' (s''' z''')))) s z)))
 -> \s (\z (((\n (\m (\s' n (m s')))) (\s'' (\z' s'' z')) (\s''' (\z'' s''' (s''' z'')))) s z))
 -> \s (\z (((\m (\s' (\s'' (\z' s'' z')) (m s'))) (\s''' (\z'' s''' (s''' z'')))) s z))
 -> \s (\z ((\s' (\s'' (\z' s'' z')) ((\s''' (\z'' s''' (s''' z''))) s')) s z))
 -> \s (\z ((\s' (\z' s' z')) ((\s'' (\z'' s'' (s'' z''))) s) z))
 -> \s (\z ((\z' ((\s' (\z'' s' (s' z''))) s) z') z))
 -> \s (\z (((\s' (\z' s' (s' z'))) s) z))
 -> \s (\z ((\z' s (s z')) z))
 -> \s (\z (s (s z)))
>
```

Рисунок 6. Редукция терма (Plus Zero (Mult One Two))

Получаем $\boldsymbol{\lambda s.\lambda z.s(sz)} = \overline{\mathbf{2}}$**.** Редукция выполнена правильно. Выйдем из программы, введя “exit”.

```
>exit

Process finished with exit code 0
```

Рисунок 7. Выход из программы

Описание программной реализации

Исполняемый файл

- int main() - точка выполнения программы. Реализует собой работу интерпретатора. Введенный пользователем запрос обрабатывается функцией DoRequest.

- void DoRequest(const string &request) – в зависимости от полученного запроса выполняет различные действия (вывод правил, запуск тестирования, бета-редукция введенного терма). Также функция ответственна за вывод шагов бета-редукции и вывод результатов тестирования.

- bool RunTests() – Запускает различные тесты, созданные с помощью CreateTest. Если хотя бы один тест показал неправильный результат, возвращается false, иначе – true.

- bool CreateTest(const string &test, const string &ans) – принимает две строки test и ans. Редуцирует первую строку и полученный результат сравнивает со второй строкой. Выводит результат теста и возвращает соответственно true или false.

Файл для описания известных термов (библиотека термов)

Данный файл служит для описания структуры данных LibFuncs.

У данного класса всего один член - map<string, string> functions. С помощью этого сортируемого ассоциативного контейнера хранятся известные выведенные лямбда-термы.

Например:

- Булевская константа Чёрча true: *functions["True"] = "(\\t (\\f t))".*
- Функция сложения: *functions["Plus"] = "(\\n (\\m (\\s (\\z (n s (m s z))))))".*
- Комбинатор-распределитель: *functions["S"] = "(\\f (\\g (\\x f x (g x))))".*

В дальнейшей реализации этот контейнер служит для того, чтобы во введенном терме заменить ключевые слова (типа *True, One, Plus, K…*) на их представление в виде терма.

Что касается методов данного класса:

- bool exist(const string &str) const – проверяет находится ли запрашиваемый терм в библиотеке.

- string **operator[]**(const string &index) – перегруженный оператор [] для доступа к значению по ключу.

Файл со вспомогательными функциями

- bool ChangeLibFuncsToTerms(vector<string> &term_vec) – принимает лямбда-терм в виде строки, ищет в нем имена библиотечных термов и подставляет их представление в виде терма. Функция возвращает true, если не было найдено синтаксических ошибок; false – иначе. В случае нахождения ошибки также выводится тип ошибки.

- vector<string> ParseToVec(string term) – принимает лямбда-терм в виде строки и «парсит» его в вектор строк. Скобки, лямбда с переменной под ней, переменные в теле лямбды, подставляемые переменные являются различными элементами вектора. Пробелы игнорируются. Например, $(\lambda x. x)\ a$ запишется как

Index	0	1	2	3	4
Vector[Index]	(	λx	x	)	a

Таблица 2. Пример парсинга терма-строки

- bool IsSyntaxCorrect(const string &term) – принимает лямбда-терм в виде строки и проверяет его на синтаксическую корректность. Проверяет, правильно ли расставлены скобки, проверяет наличие лямбда-абстракций и их корректное использование. Возвращает false в случае нахождения ошибки и выводит тип ошибки.

- bool IsSyntaxCorrect(const vector<string> &term_vec) - принимает лямбда-терм в виде вектора строк и проверяет его на корректное использование лямбда-абстракций. Возвращает false в случае нахождения ошибки и выводит тип ошибки.

Стоит отметить, что функции проверки синтаксиса не покрывают 100% возможных ошибок.

Файл с основными функциями

Для реализации исчислений на безымянных термах потребовалось ввести дополнительную структуру данных - class Term. Этот класс имеет три поля:

- int index – непосредственно индекс де Брауна;
- string term – имя терма;
- int alpha_conversion_count – переменная, нужная для корректной альфа-конверсии;

Несколько замечаний по этому классу:

- При альфа-конверсии переменная *x* переименовывается не в какую-то случайную другую переменную (к примеру, в *z*, если она не занята), а в *x*` (если она не занята). Тогда формально *alpha_conversion_count* означает сколько символов «`» у этой переменной.
- В качестве экземпляров данного выступают не только термы, но и скобки. Сформированный раннее вектор строк в последствии преобразовывается в вектор из Term по следующим правилам:
    - Для открывающей скобки index = -2.
    - Для закрывающей скобки index = -3;
    - Для лямбда-абстракции index = -1;

- Для переменных index равен индексу Де Брауна в своем контексте именования.

Теперь рассмотрим основные функции:

- vector<string> BetaReduction(const string &term) – принимает на вход терм, введенный пользователем. Затем внутри идет вызов функций проверок корректности синтаксиса, функции преобразования терма в правильную форму (см. ниже), функции парсера из строки в вектор строк, функции подстановки термов вместо библиотечных функций, функции представления терма с помощью индексов де Брауна и наконец функции бета-редукции. На выходе получаем вектор строк, где каждая из строк – терм на каждом шаге редукции.

- string MakeCorrectForm(const string &term) – принимает на вход терм в виде строки и каждую вложенную лямбда-абстракцию заключает в скобки. Это нужно для корректных дальнейших вычислений. Возвращает измененную строку.

- vector<Term> RemoveUnnecessaryBrackets(const vector<Term> &term_vec) – функция удаляет лишние скобки из терма в таких случаях, как *(), (λx), (x), ((…))*. Это так же нужно для корректных дальнейших вычислений. Возвращает измененный терм.

- vector<Term> ConvertToDeBruijnNotation(const vector<string> &term_vec) – функция принимает терм в виде вектора строк. Затем из этих строк формируются экземпляры класса Term. По каким правилам это происходит уже было описано выше. Отметим, что сначала обрабатывается все кроме свободных переменных, а затем только даются номера для свободных переменных. Нумерация де Брауна соответствует правилам, описанным в теоретической части отчета. Функция возвращает вектор из Term.

- vector<Term> RecalculateDeBrujnNotation(const vector<Term> &term_vec) – функция, которая делает практически то же самое, что и предыдущая. Однако на вход она принимает не вектор строк, а вектор из Term. Эта функция вызывается на каждом шаге бета-редукции для перенумерации индексов де Брауна. Принципиальное отличие данной функции от предыдущей состоит в том, что эта функция учитывает альфа-конверсию у переменных (учитывает текущее количество «`»).

- vector<string> ReduceNormalStrat(const vector<Term> &term_vec_original) – функция, выполняющая редукцию терма до момента, пока терм не будет приведен к своей нормальной форме. В начале каждого шага редукции вызывается функция GetOutPutString (см. ниже), и её результат добавляется в вектор, который в конце выполнения всех редукций возвращает эта функция. То есть функция ReduceNormalStrat выполнят бета-редукцию и возвращает состояния термов на каждом шаге редукции. Теперь о том, как работает бета-редукция:

    - Сначала определяется самая внешняя левая абстракция, заключенная в скобки (реализовано с помощью прохода по элементам вектора и подсчета скобок).
    - Затем по похожему принципу мы ищем выражение, которое будем подставлять.
    - Если такое выражение не найдено, то возвращаемся к первому шагу и ищем другую самую левую внешнюю абстракцию.
    - Если такое выражение найдено, то выполняется подстановка.
    - Для полученного выражения необходимо пересчитать индексы де Брауна и переименовать некоторые переменные, в связи с «расходом» связанной переменной (вызов RecalculateDeBrujnNotation).
    - Если же в ходе всей итерации не была произведена подстановка, мы получили нормальную форму.
    - Если в ходе последних двух итераций бета-редукции мы получили идентичные термы, это означает, что терм редуцируется в самого себя, а значит он не имеет нормальной формы (например, комбинатор Омега).

- string GetOutPutString(const vector<Term> &term_vec_original) – принимает вектор из Term. Сначала удаляет ненужные внешние скобки (которые образовались в результате MakeCorrectForm), затем формирует строку на вывод с корректными именами переменных (корректное количество «`») и возвращает эту самую строку.

Выводы

В результате данного отчета мы:

- описали и изучили основы лямбда-исчисления, которые являются фундаментом функционального программирования;
- изучили представление лямбда-термов без имен переменных;
- написали интерпретатор для бестипового лямбда-исчисления.

Список литературы

1. Пирс, Б. Типы в языках программирования / Б. Пирс. – М.: Лямбда пресс & Добросвет, 2011. – 655 с.
2. Башкин, В. А. Лямбда-исчисление / В. А. Башкин. – Ярославль: ЯрГУ, 2018. – 51 с.
3. Довек, Ж. Введение в теорию языков программирования / Ж. Довек, Ж. Леви. – М.: ДМК-Пресс, 2015. – 134 с.
4. Вики-конспекты // Режим доступа: https://neerc.ifmo.ru/wiki/index.php?title=Лямбда-исчисление. – Загл. С экрана.

Приложение

В данном разделе предоставлен листинг кода.

Файл main.cpp:

```
// Copyright 2021 Oganyan Robert

#include <iostream>
#include <string>
#include "./beta-reduction.h"

//#define Show_Steps_At_Tests
#define Show_Steps_At_Interpreter
using std::cout;
using std::string;
using std::cin;

bool CreateTest(const string &test, const string &ans) {
    bool result;

    auto res = BetaReduction(test);
    if (*(--res.end()) == ans) {
        cout << "Correct! " << test << " reduced correctly to " << ans;
        result = true;
    } else {
        cout << " Oopsie. " << test << " reduced wrong to " << *(--res.end());
        result = false;
    }
    cout << "\n";

#ifdef Show_Steps_At_Tests
    cout << "Showing steps: " << "\n";
    for (auto &element : res) {
        cout << " -> " << element << "\n";
    }
#endif

    return result;
}

bool RunTests() {
    bool is_passed = true;

    is_passed &= CreateTest("(\\x \\y x y y)(\\u u y x)", "\\y y y' x y");
    is_passed &= CreateTest("(\\x x x)(\\y \\z y z)", "\\z (\\z' z z')");
    is_passed &= CreateTest("\\x \\y (\\z (\\x z x) (\\y z y)) (x y)", "\\x (\\y
(x y) (\\y' (x y) y'))");
    is_passed &= CreateTest("Omega a", "Term can't be reduced");
    is_passed &= CreateTest("(\\x x (x (y z)) x) (\\u u v)", "((y z) v) v (\\u u
v)");
    is_passed &= CreateTest("First (Pair a b)", "a");
    is_passed &= CreateTest("(\\f \\x f (f x)) (\\x x)", "\\x x");
    is_passed &= CreateTest("(\\x \\y \\z x z (y z))((\\x \\y y x) u) ((\\x \\y
y x) v) w", "w u (w v)");
    is_passed &= CreateTest("(\\x \\y \\z x z y)(\\x \\y x)", "\\y (\\z z)");
    is_passed &= CreateTest("(\\b \\x \\y b x y) False t e", "e");

    return is_passed;
}

void DoRequest(const string &request) {
    if (request == "rules") {
```

```
        cout << "-+-+-+-+-+-+-+-+-+-+  Rules  -+-+-+-+-+-+-+-+-+-+-+-+-+-+" << "\n";
        cout << "1. Capital letters are only used for Library Functions such as True, False, Not ..." << "\n";
        cout << "2. Variables can be named only with cursive letters or words fully consisting of cursive letters" << "\n";
        cout << "3. Do not mess up with brackets" << "\n";
        cout << "4. Lambda is '\\x'; Expression splits with spaces, for example '(\\x x y) a -> a y' " << "\n";
        cout << "5. There are some checks for correct syntax, but not for every possible mistake. Do your best to not make any mistakes" << "\n";

        return;
    }

    if (request == "test") {
        cout << "-+-+-+-+-+-+-+-+-+-+  Running tests  -+-+-+-+-+-+-+-+-+-+-+-+-+-+" << "\n";
        if (RunTests()) {
            cout << "Tests passed successfully!" << "\n";
        } else {
            cout << "Some of tests did not pass" << "\n";
        }
        return;
    }

    if (!IsSyntaxCorrect(request)) {
        return;
    }

    auto output = BetaReduction(request);

#ifdef Show_Steps_At_Interpreter
    for (auto &element : output) {
        cout << " -> " << element << "\n";
    }

#endif

#ifndef Show_Steps_At_Interpreter
    cout << " -> " << *(--output.end()) << "\n";
#endif


}

int main() {
    cout << "Welcome to beta-reduction calculator. Choose the option:" << "\n"
         << "1. Type 'rules' to read rules for syntax " << "\n"
         << "2. Type 'test' to run implemented tests" << "\n"
         << "3. Type the lambda term to reduce it" << "\n"
         << "4. Type 'exit' to quit the program" << "\n";
    while (true) {
        cout << ">" << " ";
        string input;
        getline(cin, input);

        if (input == "exit") {
            return 0;
        }

        DoRequest(input);
    }
```

```
    return 0;
}
```

Файл terms_lib.h:

```
// Copyright 2021 Oganyan Robert

#ifndef OGANYAN_LAMBDA_CALC_TERMS_LIB_H
#define OGANYAN_LAMBDA_CALC_TERMS_LIB_H

#include <iostream>
#include <string>
#include <map>

using std::map;
using std::string;

class LibFuncs {
private:
    map<string, string> functions;
public:
    LibFuncs() {

        // Logical functions

        functions["True"] = "(\\t (\\f t))";
        functions["False"] = "(\\t (\\f f))";
        functions["If"] = "(\\b (\\x (\\y b x y)))";
        functions["And"] = "(\\b (\\c b c False))";
        functions["Or"] = "(\\b (\\c b True c))";
        functions["Not"] = "(\\b b False True)";
        functions["IsZero"] = "(\\n n (\\c False) True)";

        // Nums

        functions["Zero"] = "(\\s (\\z z))";
        functions["One"] = "(\\s (\\z s z))";
        functions["Two"] = "(\\s (\\z s (s z)))";

        // Arithmetic functions

        functions["Scc"] = "(\\n (\\s (\\z s (n s z))))";
        functions["Plus"] = "(\\n (\\m (\\s (\\z (n s (m s z))))))";
        functions["Mult"] = "(\\n (\\m (\\s n (m s))))";
        functions["Pow"] = "(\\n (\\m (\\s (\\ z (n s z)))))";

        //  Combinators

        functions["I"] = "(\\x x)";
        functions["S"] = "(\\f (\\g (\\x f x (g x))))";
        functions["K"] = "(\\x (\\y x))";
        functions["Omega"] = "((\\x x x)(\\x x x))";
        functions["B"] = "(\\f (\\g (\\x f (g x))))";
        functions["C"] = "(\\f (\\x (\\y f y x)))";
        functions["V"] = "(\\x (\\y x y y))";


        //  Pairs
        functions["Pair"] = "(\\f (\\s (\\b b f s)))";
        functions["First"] = "(\\p p True)";
        functions["Second"] = "(\\p p False)";
    }
```

```
    bool exist(const string &str) const {
        return (functions.find(str) != functions.end());
    }

    //  Not string& to avoid changing data
    string operator[](const string &index) {
        if (exist(index)) {
            return functions[index];
        }
        else {
            return "";
        }
    }
};

#endif //OGANYAN_LAMBDA_CALC_TERMS_LIB_H
```

Файл correct_syntax_check.h:

```
// Copyright 2021 Oganyan Robert

#ifndef OGANYAN_LAMBDA_CALC_CORRECT_SYNTAX_CHECK_H
#define OGANYAN_LAMBDA_CALC_CORRECT_SYNTAX_CHECK_H

#include <iostream>
#include <string>
#include <vector>
#include <cctype>
#include "terms_lib.h"

using std::cout;
using std::string;
using std::vector;
using std::to_string;

//  Check if term is correctly written. If not, returns false and prints the
mistake.

bool IsSyntaxCorrect(const string &term) {

    // Check for correct brackets

    int brackets_count = 0;
    for (const char &elem : term) {
        if (elem == '(') {
            brackets_count++;
        } else if (elem == ')') {
            brackets_count--;
        }
        if (brackets_count < 0) {
            cout << "Invalid syntax: some ending brackets dont belong to opening
brackets" << "\n";
            return false;
        }
    }

    if (brackets_count < 0) {
        cout << "Invalid syntax: invalid syntax: too much of ending brackets" <<
"\n";
        return false;
    }

    if (brackets_count > 0) {
```

```
        cout << "Invalid syntax: invalid syntax: too much of opening brackets"
<< "\n";
        return false;
    }

    //  Check for existing lambdas

    bool exist_lib_fun = false;
    for (char num : term) {
        if ((num >= 'A' && num <= 'Z')) {
            exist_lib_fun = true;
            break;
        }
    }
    if (!exist_lib_fun) {
        for (size_t num = 0; num < term.size(); ++num) {
            if (term[num] == '\\') {
                break;
            } else {
                if (num == term.size() - 1) {
                    cout << "There are no lambdas: nothing to reduce" << "\n";
                    return false;
                }
            }

        }
    }


    //  Check for correct lambda-usages

    for (size_t num = 0; num < term.size() - 1; ++num) {
        if (term[num] == '\\' && !isalpha(term[num + 1])) {
            cout << "Invalid syntax: there is no variable after lambda" << "\n";
            return false;
        }
    }

    if (term[term.size() - 1] == '\\') {
        cout << "Invalid syntax: there is no variable after lambda" << "\n";
        return false;
    }

    return true;
}

bool IsSyntaxCorrect(const vector<string> &term_vec) {
    for (size_t elem = 0; elem < term_vec.size() - 1; ++elem) {
        if (term_vec[elem][0] == '\\') {
            bool is_correct = false;
            for (size_t j = elem + 1; j < term_vec.size(); ++j) {
                if (term_vec[j] != ")") {
                    is_correct = true;
                }
            }
            if (!is_correct) {
                cout << "Invalid syntax: there is no variable after lambda" <<
"\n";
                return false;
            }
        }
    }
    return true;
```

```
}

// Basically just transforms string to vector

vector<string> ParseToVec(string term) {
    vector<string> term_vector;
    LibFuncs library;
    for (size_t num = 0; num < term.size(); ++num) {
        if (term[num] == ' ') {
            continue;
        }
        if ((term[num] == ')') || (term[num] == '(')) {
            string element;
            element += term[num];
            term_vector.push_back(element);
            continue;
        }

        string element;
        element.push_back(term[num]);
        while ((num + 1 < term.size()) &&
               term[num + 1] != ' ' &&
               term[num + 1] != '(' &&
               term[num + 1] != ')') {
            element.push_back(term[num + 1]);
            ++num;
        }
        if ((element[0] >= 'A') && ((element[0] <= 'Z'))) {
            if (!library.exist("There is no such library function!"));
        }

        term_vector.push_back(element);
    }
    return term_vector;
}

//  Finds library functions in term (such as True, False, etc) and decompose
them
bool ChangeLibFuncsToTerms(vector<string> &term_vec) {
    LibFuncs library;
    while (true) {
        bool need_to_parse = false;
        string new_term;
        for (size_t elem = 0; elem < term_vec.size(); elem++) {
            bool found_capital = false;
            string cur_term = term_vec[elem];
            for (char letter : cur_term) {
                if ((letter >= 'A') && (letter <= 'Z')) {
                    found_capital = true;
                    break;
                }
            }
            if (!found_capital) {
                new_term += cur_term;
                if (elem != term_vec.size() - 1) {
                    new_term += " ";
                }
                continue;
            }
            if (cur_term[0] == '\\') {
                if (!library.exist(cur_term.substr(1))) {
                    cout << "Invalid syntax: capital characters can be only used
for library functions" << "\n";
```

```
                    return false;
                }
            } else {
                if (!library.exist(cur_term)) {
                    cout << "Invalid syntax: capital characters can be only used
for library functions" << "\n";
                    return false;
                }
            }
            new_term += library[cur_term];
            if (elem != term_vec.size() - 1) {
                new_term += " ";
            }
            need_to_parse = true;
        }
        term_vec = ParseToVec(new_term);
        if (!need_to_parse) break;
    }
    return true;
}


#endif //OGANYAN_LAMBDA_CALC_CORRECT_SYNTAX_CHECK_H
```

Файл beta-reduction.h:

```
// Copyright 2021 Oganyan Robert

#ifndef OGANYAN_LAMBDA_CALC_BETA_REDUCTION_H
#define OGANYAN_LAMBDA_CALC_BETA_REDUCTION_H

#include <iostream>
#include <string>
#include <vector>
#include <map>
#include <set>
#include <stack>
#include "correct_syntax_check.h"

//#define DEBUG

using std::cout;
using std::set;
using std::map;
using std::vector;
using std::string;
using std::stack;

const int lambda = -1;
const int opening_bracket = -2;
const int closing_bracket = -3;
const int unprocessed = -4;

//  class for de_bruijn. index - distance, term - name, alpha_conversion_count -
count of "'"
class Term {
public:
    int index;
    string term;
    int alpha_conversion_count;

    Term() {
        index = unprocessed;
```

```
        term = "";
        alpha_conversion_count = 0;
    }

    Term(int index_, string term_, int alpha_conversion_count_) :
            index(index_), term(std::move(term_)),
alpha_conversion_count(alpha_conversion_count_) {}

    Term(int index_, string term_) :
            index(index_), term(std::move(term_)), alpha_conversion_count(0) {}

};

//  Making appropriate form of term for output in future

string GetOutPutString(const vector<Term> &term_vec_original) {
    auto term_vec = term_vec_original;
    string result_string_term;
    // Remove the most left and the most right brackets if they are useless
    while (true) {
        bool cycle_is_done = true;
        if ((term_vec[0].term == "(") && (term_vec[term_vec.size() - 1].term ==
")")) {
            int brackets_count = 0;
            bool need_to_remove_brackets = true;
            for (size_t elem = 1; elem < term_vec.size() - 1; ++elem) {
                if (term_vec[elem].term == "(") {
                    brackets_count++;
                } else if (term_vec[elem].term == ")") {
                    brackets_count--;
                }
                if (brackets_count == -1) {
                    need_to_remove_brackets = false;
                    break;
                }
            }
            if (need_to_remove_brackets) {
                term_vec.pop_back();
                term_vec.erase(term_vec.begin());
                cycle_is_done = false;
            }
        }
        if (cycle_is_done) {
            break;
        }
    }

    // Need to rename back not necessary renamed variables
    map<string, set<int>> alpha_conversion_map;

    alpha_conversion_map["("].insert(0);
    alpha_conversion_map[")"].insert(0);

    // pre-processing
    for (const auto &term : term_vec) {
        if (term.term == "(" || term.term == ")") {
            continue;
        }
        if (term.term[0] == '\\') {

alpha_conversion_map[term.term.substr(1)].insert(term.alpha_conversion_count);
        } else {
            alpha_conversion_map[term.term].insert(term.alpha_conversion_count);
```

```
        }
    }

    // making the result string considering correct amount of "'"
    for (size_t elem = 0; elem < term_vec.size(); ++elem) {
        auto cur_term = term_vec[elem];
        result_string_term += cur_term.term;
        string name_of_term;
        auto alpha_conversion_count = alpha_conversion_map["("].begin();
        if (cur_term.term[0] == '\\') {
            name_of_term = cur_term.term.substr(1);
            alpha_conversion_count =
alpha_conversion_map[name_of_term].find(cur_term.alpha_conversion_count);
        } else {
            name_of_term = cur_term.term;
            alpha_conversion_count =
alpha_conversion_map[name_of_term].find(cur_term.alpha_conversion_count);
        }

        while (true) {
            if (alpha_conversion_count ==
alpha_conversion_map[name_of_term].begin()) {
                break;
            }
            result_string_term += "'";
            --alpha_conversion_count;
        }

        if (elem + 1 < term_vec.size()) {
            auto next_term = term_vec[elem + 1];
            if ((cur_term.term != "(") && (next_term.term != ")")) {
                result_string_term += " ";
            }
        }

    }
    return result_string_term;
}

//  Basically inserts brackets before every lambda and at the end of lambda's
scope

string MakeCorrectForm(const string &term) {
    string return_term = term;
    while (true) {
        string new_term;
        bool is_correct = true;
        for (size_t pos_for_opening_bracket = 0;
             pos_for_opening_bracket < return_term.size();
++pos_for_opening_bracket) {
            if (return_term[pos_for_opening_bracket] == '\\' &&
return_term[pos_for_opening_bracket - 1] != '(') {
                int brackets_count = 0;
                int pos_for_ending_bracket = return_term.size();
                for (size_t j = pos_for_opening_bracket + 1; j <
return_term.size(); ++j) {
                    if (return_term[j] == '(') {
                        brackets_count++;
                    } else if (return_term[j] == ')') {
                        brackets_count--;
                    }
                    if (brackets_count < 0) {
                        pos_for_ending_bracket = j;
```

```
                    break;
                }
            }
            new_term = return_term.substr(0, pos_for_opening_bracket) + '('
                       +
                       return_term.substr(pos_for_opening_bracket,
pos_for_ending_bracket - pos_for_opening_bracket)
                       + ')' +
                       return_term.substr(pos_for_ending_bracket,
return_term.size() - pos_for_ending_bracket);
            is_correct = false;
        }
    }
    if (is_correct) {
        break;
    }
    return_term = new_term;
    }
    return return_term;
}

//  Calculates term with De Bruin Notation
vector<Term> ConvertToDeBruijnNotation(const vector<string> &term_vec) {

    vector<Term> term_de_bruijn(term_vec.size(), Term());
    map<string, int> usage_counter;

    // Main part of processing indexes and making alpha-conversion for NOT FREE
(bounded) terms
    for (size_t cur_elem = 0; cur_elem < term_vec.size(); ++cur_elem) {
        if (term_de_bruijn[cur_elem].index != unprocessed) {
            continue;
        }
        string cur_term = term_vec[cur_elem];
        if (cur_term == "(") {
            term_de_bruijn[cur_elem] = Term(opening_bracket, cur_term);
            continue;
        }
        if (cur_term == ")") {
            term_de_bruijn[cur_elem] = Term(closing_bracket, cur_term);
            continue;
        }
        if (cur_term[0] == '\\') {
            term_de_bruijn[cur_elem] = Term(lambda, cur_term,
usage_counter[cur_term.substr(1)]++);
            int brackets_in_lambda_count = 0;
            int end_lambda = -1;
            // true if opening bracket is for the following lambda ((\n n));
false if opening bracket is for application ((a b))
            stack<bool> brackets_type;
            for (size_t j = cur_elem + 1; j < term_vec.size(); ++j) {
                string cur_sub_term = term_vec[j];
                if (cur_sub_term == "(") {
                    if (term_vec[j + 1][0] == '\\') {
                        brackets_type.push(true);
                        brackets_in_lambda_count++;
                    } else {
                        brackets_type.push(false);
                    }
                    continue;
                } else if (cur_sub_term == ")") {
                    if (brackets_type.empty()) {
                        break;
```

```
                        }
                        if (brackets_type.top()) {
                            brackets_in_lambda_count--;
                        }
                        brackets_type.pop();
                        if (brackets_in_lambda_count == end_lambda - 1) {
                            end_lambda = -1;
                        }
                        continue;
                    }
                    if (end_lambda == -1 && cur_sub_term == cur_term.substr(1)) {
                        term_de_bruijn[j] = Term(brackets_in_lambda_count,
cur_sub_term,

term_de_bruijn[cur_elem].alpha_conversion_count);
                        continue;
                    }
                    if (cur_sub_term == cur_term) {
                        end_lambda = brackets_in_lambda_count;
                    }
                }
            }
        }

    //  Pre-Processing for FREE terms.
    map<string, int> free_terms;
    for (int elem = static_cast<int>(term_vec.size()) - 1; elem >= 0; --elem) {
        if (term_de_bruijn[elem].index != unprocessed) {
            continue;
        }
        free_terms[term_vec[elem]] = free_terms.size() - 1;
    }

    int count_of_bound_terms = 0;
    stack<bool> brackets_type;
    //  Processing for FREE terms
    for (size_t elem = 0; elem < (int) term_vec.size(); ++elem) {
        if (term_vec[elem] == "(") {
            if (term_vec[elem + 1][0] == '\\') {
                count_of_bound_terms++;
                brackets_type.push(true);
            } else {
                brackets_type.push(false);
            }
        }
        if (term_vec[elem] == ")") {
            if (brackets_type.top()) {
                count_of_bound_terms--;
            }
            brackets_type.pop();
        }
        if (term_de_bruijn[elem].index == unprocessed) {
            term_de_bruijn[elem] = Term(free_terms[term_vec[elem]] +
count_of_bound_terms, term_vec[elem],
                                        usage_counter[term_vec[elem]]);
        }
    }

    return term_de_bruijn;
}

//  Almost the same as previous one, but it includes alpha-conversion while
processing
```

```
vector<Term> RecalculateDeBrujnNotation(const vector<Term> &term_vec) {
    vector<Term> term_de_bruijn(term_vec.size(), Term());
    map<string, int> usage_counter;

    // Main part of processing indexes and making alpha-conversion for NOT FREE
(bounded) terms
    for (size_t cur_elem = 0; cur_elem < term_vec.size(); ++cur_elem) {
        if (term_de_bruijn[cur_elem].index != unprocessed) {
            continue;
        }
        string cur_term = term_vec[cur_elem].term;
        if (cur_term == "(") {
            term_de_bruijn[cur_elem] = Term(opening_bracket, cur_term);
            continue;
        }
        if (cur_term == ")") {
            term_de_bruijn[cur_elem] = Term(closing_bracket, cur_term);
            continue;
        }
        if (cur_term[0] == '\\') {
            term_de_bruijn[cur_elem] = Term(lambda, cur_term,
usage_counter[cur_term.substr(1)]++);
            int brackets_in_lambda_count = 0;
            int end_lambda = -1;
            // true if opening bracket is for the following lambda ((\n n));
false if opening bracket is for application ((a b))
            stack<bool> brackets_type;
            for (size_t j = cur_elem + 1; j < term_vec.size(); ++j) {
                string cur_sub_term = term_vec[j].term;
                if (cur_sub_term == "(") {
                    if (term_vec[j + 1].term[0] == '\\') {
                        brackets_type.push(true);
                        brackets_in_lambda_count++;
                    } else {
                        brackets_type.push(false);
                    }
                    continue;
                } else if (cur_sub_term == ")") {
                    if (brackets_type.empty()) {
                        break;
                    }
                    if (brackets_type.top()) {
                        brackets_in_lambda_count--;
                    }
                    brackets_type.pop();
                    if (brackets_in_lambda_count == end_lambda - 1) {
                        end_lambda = -1;
                    }
                    continue;
                }
                if ((end_lambda == -1 && cur_sub_term == cur_term.substr(1))
                    && (term_vec[j].alpha_conversion_count ==
term_vec[cur_elem].alpha_conversion_count)) {
                    term_de_bruijn[j] = Term(brackets_in_lambda_count,
cur_sub_term,

term_de_bruijn[cur_elem].alpha_conversion_count);
                    continue;
                }
                if ((cur_sub_term == cur_term) &&
                    (term_vec[j].alpha_conversion_count ==
term_vec[cur_elem].alpha_conversion_count)) {
```

```
                    end_lambda = brackets_in_lambda_count;
                }
            }
        }
    }

    //  Pre-Processing for FREE terms
    map<string, int> free_terms;
    for (int elem = static_cast<int>(term_vec.size()) - 1; elem >= 0; --elem) {
        if (term_de_bruijn[elem].index != unprocessed) {
            continue;
        }
        free_terms[term_vec[elem].term] = free_terms.size() - 1;
    }

    int count_of_bound_terms = 0;
    stack<bool> brackets_type;

    //  Processing for FREE terms
    for (size_t elem = 0; elem < (int) term_vec.size(); ++elem) {
        if (term_vec[elem].term == "(") {
            if (term_vec[elem + 1].term[0] == '\\') {
                count_of_bound_terms++;
                brackets_type.push(true);
            } else {
                brackets_type.push(false);
            }
        }
        if (term_vec[elem].term == ")") {
            if (brackets_type.top()) {
                count_of_bound_terms--;
            }
            brackets_type.pop();
        }
        if (term_de_bruijn[elem].index == unprocessed) {
            term_de_bruijn[elem] = Term(free_terms[term_vec[elem].term] +
count_of_bound_terms, term_vec[elem].term,
                                        usage_counter[term_vec[elem].term]);
        }
    }

    return term_de_bruijn;
}

//  Removes no need brackets for correct working of beta-reduction
vector<Term> RemoveUnnecessaryBrackets(const vector<Term> &term_vec) {
    auto return_vec = term_vec;

    //  Remove (), (\x) (x)
    while (true) {
        bool found_useless_brackets = false;
        for (size_t elem = 0; elem < return_vec.size(); ++elem) {
            if (return_vec[elem].term != "(") {
                continue;
            }
            if (return_vec[elem + 1].term == ")") {
                found_useless_brackets = true;
                return_vec.erase(return_vec.begin() + elem, return_vec.begin() +
elem + 2);
                break;
            }
            if (return_vec[elem + 2].term == ")") {
                found_useless_brackets = true;
```

```
                return_vec.erase(return_vec.begin() + elem + 2);
                return_vec.erase(return_vec.begin() + elem);
                break;
            }
        }
        if (!found_useless_brackets) {
            break;
        }
    }


    // Remove ((*))
    while (true) {
        bool found_useless_brackets = false;
        for (size_t elem = 1; elem < return_vec.size(); ++elem) {
            if (return_vec[elem].term != "(" ||
                return_vec[elem - 1].term != "(") {
                continue;
            }
            int brackets_count = 0;
            for (size_t j = elem + 1; j < return_vec.size(); ++j) {
                if (return_vec[j].term == "(") {
                    brackets_count++;
                } else if (return_vec[j].term == ")") {
                    brackets_count--;
                    if (brackets_count == -1) {
                        if (return_vec[j + 1].term == ")") {
                            return_vec.erase(return_vec.begin() + j + 1);
                            return_vec.erase(return_vec.begin() + elem);
                            found_useless_brackets = true;
                        }
                        break;
                    }
                }
            }
        }
        if (!found_useless_brackets) {
            break;
        }
    }
    return return_vec;
}

// Reduce term in De Brujn's form with normal strategy (application for the
outermost left redex)
vector<string> ReduceNormalStrat(const vector<Term> &term_vec_original) {
    auto term_vec = term_vec_original;
    vector<string> result_reduce;
    while (true) {
        vector<Term> current_reduce_step;

        term_vec = RemoveUnnecessaryBrackets(term_vec);
        result_reduce.push_back(GetOutPutString(term_vec));

        //  For some kind of terms (f.e. (\x x x)...) we need to recalculate
DeBrujn indexes
        term_vec = RecalculateDeBrujnNotation(term_vec);

        //  If Beta-reduction is infinite and does not have normal form ((f.e
Combinator Omega))
        if (result_reduce.size() >= 2) {
            if (result_reduce[result_reduce.size() - 1] ==
result_reduce[result_reduce.size() - 2]) {
```

```
                result_reduce.clear();
                result_reduce.emplace_back("Term can't be reduced");
                return result_reduce;
            }
        }


        bool term_changed = false;

        for (size_t elem = 0; elem < term_vec.size(); ++elem) {
            if (term_vec[elem].index != lambda) {
                continue;
            }
            int brackets_count = 0;
            //  Getting the expression into which we will perform the
substitution
            int lambda_end_bracket = static_cast<int>(term_vec.size()) - 1;
            for (size_t j = elem + 1; j < term_vec.size(); ++j) {
                if (term_vec[j].index == opening_bracket) {
                    brackets_count++;
                } else if (term_vec[j].index == closing_bracket) {
                    brackets_count--;
                    if (brackets_count == -1) {
                        lambda_end_bracket = j;
                        break;
                    }
                }
            }

            //  Found no substituted expression
            if (lambda_end_bracket == term_vec.size() - 1) {
                continue;
            }

            // Found one
            vector<Term> substituted_term;
            if (term_vec[lambda_end_bracket + 1].index >= 0) {
                substituted_term.push_back(term_vec[lambda_end_bracket + 1]);
            } else if (term_vec[lambda_end_bracket + 1].index ==
opening_bracket) {
                brackets_count = 0;
                substituted_term.push_back(term_vec[lambda_end_bracket + 1]);
                for (int j = lambda_end_bracket + 2; j < (int) term_vec.size();
j++) {
                    if (term_vec[j].index == opening_bracket) {
                        brackets_count++;
                    } else if (term_vec[j].index == closing_bracket) {
                        brackets_count--;
                    }
                    substituted_term.push_back(term_vec[j]);
                    if (brackets_count == lambda) {
                        break;
                    }
                }
            }

            if (substituted_term.empty()) {
                continue;
            }

            for (size_t elem_before_application = 0; elem_before_application <
elem - 1; ++elem_before_application) {
```

```
current_reduce_step.push_back(term_vec[elem_before_application]);
            }

            //  True if lambda stays after opening bracket
            stack<bool> bracket_type;
            brackets_count = 0; //  Count of lambdas in current scope

            //  The main part. The one step of beta-reduction is processing
here.
            //  I don't know language that well to describe everything whats
going here
            for (size_t j = elem + 1; j < lambda_end_bracket; j++) {
                if (term_vec[j].index == opening_bracket) {
                    if (term_vec[j + 1].index == lambda) {
                        bracket_type.push(true);
                        brackets_count++;
                    } else {
                        bracket_type.push(false);
                    }
                }
                if (term_vec[j].index == closing_bracket) {
                    if (bracket_type.top()) {
                        brackets_count--;
                    }
                    bracket_type.pop();
                }
                if (term_vec[j].index < brackets_count) {
                    current_reduce_step.push_back(term_vec[j]);
                } else if (term_vec[j].index > brackets_count) {
                    current_reduce_step.emplace_back(term_vec[j].index - 1,
term_vec[j].term,

term_vec[j].alpha_conversion_count);
                } else {
                    stack<bool> substituted_term_bracket_type;
                    int item_cnt = 0;
                    for (size_t k = 0; k < substituted_term.size(); ++k) {
                        if (substituted_term[k].index == opening_bracket) {
                            if (substituted_term[k + 1].index == lambda) {
                                substituted_term_bracket_type.push(true);
                                item_cnt++;
                            } else {
                                substituted_term_bracket_type.push(false);
                            }
                        }
                        if (substituted_term[k].index == closing_bracket) {
                            if (substituted_term_bracket_type.top()) {
                                item_cnt--;
                            }
                            substituted_term_bracket_type.pop();
                        }
                        if (item_cnt <= substituted_term[k].index) {

current_reduce_step.emplace_back(substituted_term[k].index + brackets_count,

substituted_term[k].term,

substituted_term[k].alpha_conversion_count);
                        } else {
                            current_reduce_step.push_back(substituted_term[k]);
                        }
                    }
```

```
                }
            }
            for (size_t elem_after_application = lambda_end_bracket +
substituted_term.size() + 1;
                 elem_after_application < term_vec.size();
elem_after_application++) {
                current_reduce_step.push_back(term_vec[elem_after_application]);
            }
            term_changed = true;
            break;
        }
        if (!term_changed) {
            break;
        }
        term_vec = current_reduce_step;
    }
    return result_reduce;
}


vector<string> BetaReduction(const string &term) {
    //  Check for correct input
    if (!IsSyntaxCorrect(term)) {
        return vector<string>();
    }

    //  Make appropriate form of term
    auto good_form_term = MakeCorrectForm(term);
    if (!IsSyntaxCorrect(good_form_term)) {
        return vector<string>();
    }

    //  Convert string to vector
    vector<string> term_vec = ParseToVec(good_form_term);

    //  Check for correct syntax again

    if (!IsSyntaxCorrect(term_vec)) {
        return vector<string>();
    }

    //  Find Functions from Library and decompose them

    if (!ChangeLibFuncsToTerms(term_vec)) {
        return vector<string>();
    }

    //  Calculate De Brujn Notation
    vector<Term> term_de_bruijn = ConvertToDeBruijnNotation(term_vec);

#ifdef DEBUG

    cout << "\n";

    for (const auto& elem: term_de_bruijn) {
        cout << elem.term << " ";
    }
    cout << "\n";

    for (const auto& elem: term_de_bruijn) {
        cout << elem.index << " ";
    }
    cout << "\n";
```

```
    for (const auto& elem: term_de_bruijn) {
        cout << elem.alpha_conversion_count << " ";
    }
    cout << "\n";
#endif

    //  Beta-reduction

    vector<string> reduction_result = ReduceNormalStrat(term_de_bruijn);
    return reduction_result;
}

#endif //OGANYAN_LAMBDA_CALC_BETA_REDUCTION_H
```