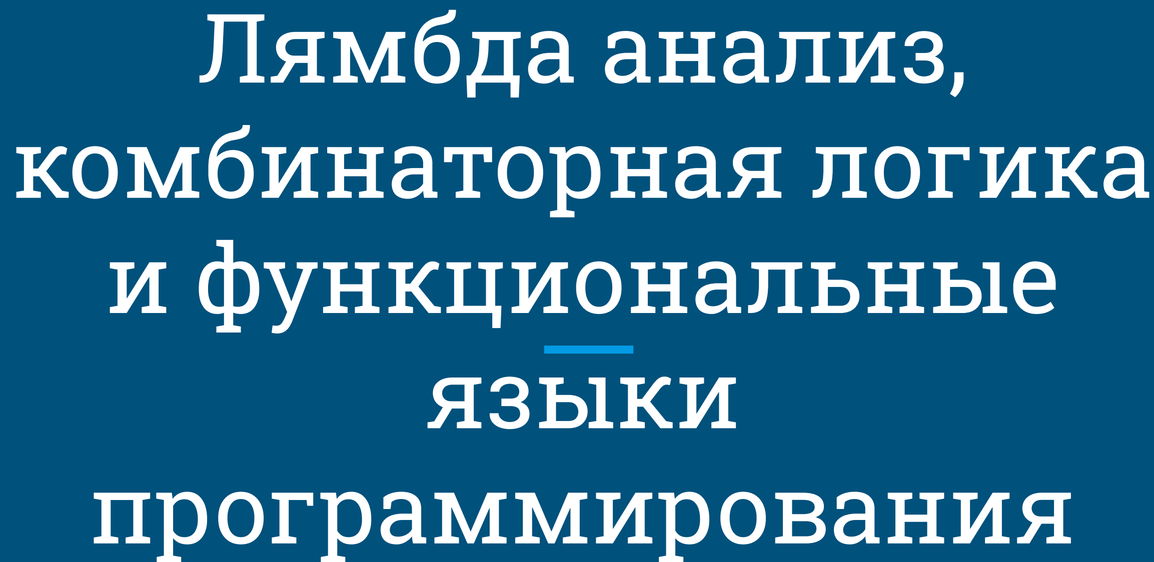

Лямбда анализ,
комбинаторная логика
и функциональные
языки
программирования

Руслан Голов

История

Λ-исчисление - это теоретическое понятие, которое требует изучения в случае, если вы планируете заниматься исследованиями в области лямбда анализа или программировать на функциональном языке.

- Проблема разрешения (Entscheidungsproblem) (1928 г., Давид Гильберт)
- Первый ответ дал Алонзо Чёрч, использовавший λ логику
- Второй ответ дал Алан Тьюринг изобретая машину Тьюринга.

Теорема Чёрча-Тьюринга:

Не существует алгоритма который бы принимал в качестве входных данных описание любого другого алгоритма на который были 2 возможных ответа: “да” или “нет” используя при этом формальную мат. логику.

История

Таким образом, λ-исчисление впервые привлекло внимание, но продолжало оставаться в области математической логики ещё несколько десятилетий. Однако в середине 1960-х годов Питер Ландин предложил новый взгляд на эту тему и обратил внимание на то, что сложные языки программирования можно более эффективно изучать, сформулировав их ядро в виде небольшой базовой системы, описывающей основные механизмы языка, и дополнив её удобными производными формами, которые можно было бы выразить через базовую систему. В качестве такой основы, Ландин выбрал λ-исчисление Чёрча.

Основы

Лямбда-анализ на поверхности представляет собой особую запись для функций. Оно чрезвычайно лаконично, но, на первый взгляд, сложно.

Давайте рассмотрим функцию 3x:

○	f(x) = 3x	Мат. язык
○	function f(x) begin f := 3*x end;	Pascal
○	(lambda (x) (*3x))	Lisp
○	λ x . 3x	Лямбда запись

Основы - задания 1

Давайте вычислим следующие выражения:

- λ5. +3 5
- λ5. 3 5
- λ5. λ9. 3 5 9
- λ5 9. 3 5 9

Основы - Преобразование Карри-Шейнфинкеля

Как мы должны были заметить $\lambda y.\lambda x.* \; y \; x = \lambda yx.* \; y \; x$

А теперь увидим, что если функция может выдавать в качестве значения функцию и сама применяться к функциям, то функции многих аргументов не нужны. В самом деле, преобразуем вызов f(x,y,z) следующим образом. Применим f к x. Получится остаточная функция $f_1(y,z)$ уже от двух аргументов, которая, будучи применена к y, даёт опять остаточную функцию $f_2(z)$, которая наконец-то выдаст результат. Таким образом, можно было бы записать как f(x)(y)(z) или ((f x) y) z или (f x y z).

В теории данное преобразование называется преобразованием Карри-Шейнфинкеля (преобразованием Карри (рус.) или currying (анг.)), а в практике программирования — частичной параметризацией.

Основы - Определение комбинатора

В общем, функции в лямбда анализе могут быть высшего порядка.

Функции высшего порядка — это такие функции которые оперируют не над множествами чисел, матрицами и тп., а над функциями (higher-order functions анг.)

Оператор — математические операции, применяемые к функциям, если результат это тоже функция. Хотя сейчас они чаще всего называются функционалами высших типов.

Комбинатор — это такой оператор, который нельзя получить из других операторов.

Комбинаторы - SKI логика

Шейнфинкель выделил три главных таких комбинатора:

- (I x) = λx.x = x (единичный комбинатор)
- (K x y) = λxy.x = x (поглотитель)
- (S x y z) = λx.z λy.z = (x z)(y z) (заменитель)

Теперь выведем и определим некоторые другие комбинаторы:

- (F x y) = (S K x y) = K y (x y) = y
- (N x) = (F K x)

Комбинаторы - рекурсия

Далее рассмотрим рекурсивные и самовыражающиеся комбинаторы:

(D x) = (S I I x) = (I x)(I x) = x x   (удвоитель)

Ω = (D D) = S I I(S I I) = (S I I (I (S I I)) = (I (S I I)(S I I)) = (D D) = Ω

(H x y) = (S(K x)(S I I)) y = K x y (D y) = x (y y)

А теперь рекурсивные выражения:

(D H) = (S I I H) = (H H) = x (H H) = x (x (H H)) = x (x (x (H H))) = …

(Y x y) = (S (K S) K) D (S (S (K (S (K S) K)) S) (K K) (S (K S) K) D) =

= λy.(λx. y (D x)) (λx.y (D x)) (рекурсия)

Комбинаторы - отход от λ выражений

Давайте преобразуем λ выражение через комбинаторы.

Пример 1.

- $\lambda xy.yx$

  $\lambda xy.yx = \lambda x.\lambda y.yx = \lambda x.S(\lambda y.y)\lambda y.x = \lambda x.SI\ \lambda y.x = \lambda x.SIKx =$

  $= S(\lambda x.SI)(\lambda x.Kx) = (SKSI)(S\lambda x.K)\lambda x.x$

  $= (SKSI)(SKKI)$

Комбинаторы - отход от λ выражений

Давайте преобразуем λ выражение через комбинаторы.

Пример 2.

- λfx.fxx, где f - произвольная функция

  λfx.fxx = λf.λx.fxx =λf.S(λx.fx)(λx.x) = λf.SSλx.f(λx.x)I =

  = λf.SSKfII.S(λf.SSKfI)(λf.I) = λf.SSKfII.S(λf.SSKfI)(KI) =

  = SS(λf.S)(λf.SSfKI)(KI) =(SS(KS))(SSλf.S)(λf.f)(KI)(KI) =

  = (SS(KS))(SS(KS))(S(KK)I(KIKI)

Задания 2

1. Найдите такую последовательность K и S что можно выразить I комбинатор
2. Упростите выражение: K S(I(S K S I))
3. Упростите выражение: (((λx. λy.(x y))(λy.y)) w)

Ответы 2

1. (S K K x) = (K x (K x)) = x
2. K S(I(S K S I)) = K S(I (K I (S I))) = K S (I (I)) = K S I = S
3. (((λx. λy.(x y))(λy.y)) w) = (((λx. λa.(x a))(λy.y)) w) = ((λa.((λy.y) a)) w) =

   = (λy.y) w = w

Комбинаторы - булева логика

Рассмотрим комбинаторы K, F, N поближе и увидим:

(K)N = K(F)(K) = F

(F)N = F(F)(K) = K   При данных условиях N ведет себя как логическое "Не"

(K)K(K) = K(K)(K) = K

(K)K(F) = K(K)(F) = K

(F)K(K) = F(K)(K) = K

(F)K(F) = F(K)(F) = F

(K)(K)F = K(K)(F) = K

(K)(F)F = K(F)(F) = F

(F)(K)F = F(K)(F) = F

(F)(F)F = F(F)(F) = F

K - комбинатор логического "Или", F - комбинатор логического "И"

Задания 3

Мы показали что λ выражения могут быть определять логические выражения, напрмер И, ИЛИ, НЕ. А можно ли условия преобразовывать запись с λ? Оказывается, можно.

ЕСЛИ = λxyf. λf(xyf)

А теперь попробуйте вычислить результат следующих выражений:

1. ЕСЛИ ЛОЖЬ ΩI

(вторая часть будет в задании 4)

Ответы 3

1. ЕСЛИ ЛОЖЬ ΩI = (λx. λy. λf. f x y f) (λx. λy. y) ((λx. x x) (λx. x x)) (λx. x) =
   = (λy. λf.(λx. λy. y) y f) ((λx. x x) (λx. x x)) (λx. x) =
   = (λf.(λx. λy. y) ((λx. x x) (λx. x x)) f) (λx. x) = (λx. λy. y)((λx. x x) (λx. x x)) (λx. x) =
   = (λy. y) (λx. x) = I I = I

Комбинаторы - числа как функции

Числа Чёрча - это представления натуральных чисел в кодировке Чёрча. функция высшего порядка, представляющая натуральное число n, - это функция, которая отображает любую функцию f в её n-кратную композицию. Проще говоря, "значение" числа эквивалентно количеству раз, инкапсулируя свой аргумент.

$f^{*n}(x) = f(f(f(\ldots f(x)))) = f * f * f * \ldots * f(x)$

$\mathbb{Z}^+$	Опр. через функции	Опр. через λ-функции
0	f(x) = x	λf.I
1	f(x) = f˜(x)	λf˜.Ix.Y
2	f(x) = f˜(f˜(x))	λf˜.Ix.Y Y
…	…	…
n	$f(x) = f^{*\sim n}(x)$	$\lambda \tilde{f}.Ix.Y^{*n}$

Комбинаторы - числа Чёрча

Мы определили числа Чёрча через функции, теперь определим операции над ними:

succ(n) = n + 1, succ(n) = λn.λf.λx.f (n f x) = λ succ n.succ(n) $f^{*(n+1)}(x) = f(f^{*n}(x))$

plus(m, n) = K m n succ(m) $f^{*(m+n)}(x) = f^{*m}(f^{*n}(x))$

mult(m, n) = K m n add(m, n) $f^{*(m*n)}(x) = (f^{*n})^{*m}(x)$

prev(n) = n - 1, prev(n) = λn.λf.λx. (n I(n, λsucc(n, -1).λn.x))

minus(m, n) = K m n prev(m)

Задания 4

1. Создайте комбинаторное выражение для экспоненты двух чисел Чёрча
2. Создайте комбинаторное выражение для деление двух чисел Чёрча
3. Из задания 3, создайте комбинаторное выражение которое проверяет на нечетность чисел Чёрча

Ответы 4

1. exp(m, n) = K m n mult(m, n)
2. divide(m, n) = S f I ((Y Y f m) n)
3. НЕЧЕТ = λn.ЕСЛИ 1 И ЕСЛИ НЕ НЕЧЕТ (n − 1) =

   = λn.ЕСЛИ λf.Ix.Y F ЕСЛИ N НЕЧЕТ K m λf.Ix.Y λn.λf.λx. (m I(m, λsucc(m, -1).λm.x)) = …

   = H ((λx. H (x x)) (λx. H (x x))) = (λx. H (x x)) (λx. H (x x)) = (λf.(λx. f (x x)) (λx. f (x x))) H =

   = Y H

Комбинаторы - двоичная система

Мы узнали что через λ-выражения можно создавать числовую системы, булеву логику и пора определить двоичную систему. Это нужно, потому что компьютеры понимают только 0 и 1 и если определить λ-выражения через это же 0 и 1, то можно создавать вычислительную технику на лямбда анализе. Ученый De Bruijn сделал следующую рекурсию:

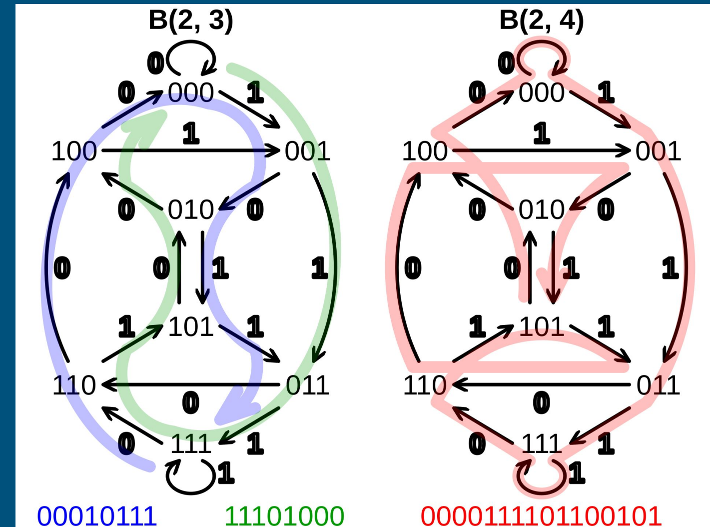
$\lambda x.00x$, $\lambda xy.01xy$, $i = 1^i0$ где i это числа Чёрча

Таким образом, каждый комбинатор можно представить как двоичное число.

Комбинаторы - двоичная система

Возьмем, например, поглотитель (K x y = λxy.x = λλ2 = 00001110)

Или λxyz.zxy = λλλ132 = 00001011 01110110



Комбинаторы - дополнительные

Давайте вернемся к комбинаторам и выведем те которые встречаются в функциональном программировании.

Комбинатор (в мат логике)	Комбинатор (в ЯП)	λ выражение	Комбинатор (в мат логике)	Комбинатор (в ЯП)	λ выражение
I	I	λx.x	Ψ	Ψ	λxyzw.x(yz)(yw)
K	K	λxy.x	S'	Φ	λxyzw.x(yw)(zw)
H	W	λxy.xyy	$Φ_1$	$Φ_1$	λxyzwu.x(ywu)(zwu)
S	S	λxyz.xz(yz)	B	B	λxyz.x(yz)
F	F	λxy.y	B_1	B_1	λxyzw.x(yzw)
D	D_0	λx.xx	S_2	D	λxyzw.xy(zw)
Y	Y/R	λx.λy.y(x x)(λy.y(x x))	S_3	D_2	λxyzwu.x(yw)(zu)

Задания 5

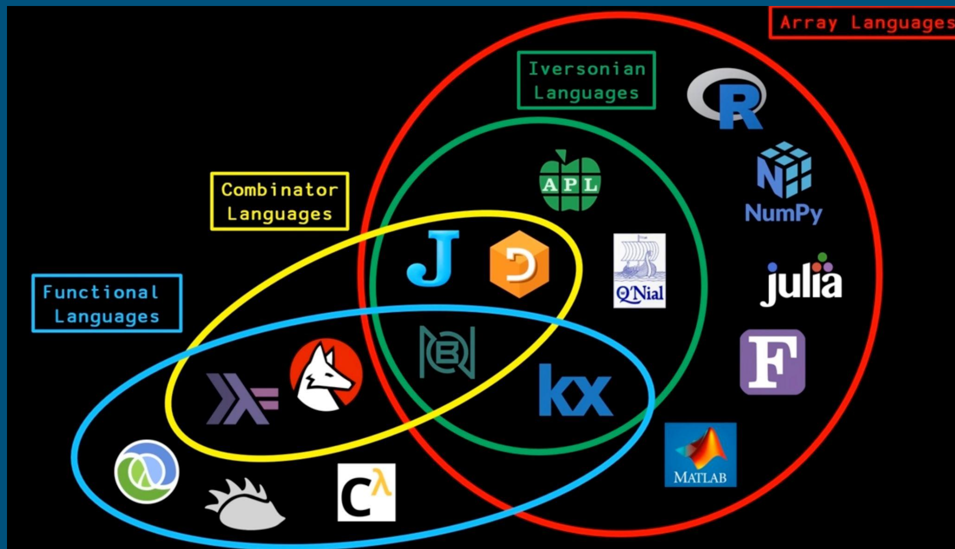
Преобразуйте следующее λ выражение через комбинаторы.

1. λxb(f(ax)), где a, b разные вводные данные и их нельзя упростить
2. Упростите комбинаторное выражение: Ф И(>a)(<b), где И - логическое И
3.
    3.1. Докажите что выражение: (λf.((λf.((λx.(f (x x)))(λx.(f (x x))))) f)) не сократимо
    3.2. Выразите это λ выражение через Y комбинатор

Ответы 5

1. λxb(f(ax)) = S(λx.b)(λx.f(ax)) = S(KB)(Sλx.f)(λx.ax) = S(Kb)(SKf)(Sλx.a)(λx.x) = S(Kb)(SKf)(SKa)I = S(Kb)(S(KK)I)(SKa)I
2. Φ И(>a)(<b) = Φ И(>a) НЕ(>b) = Ψ (НЕ И >) (a b) = ΨK(>)Ka(a b)Ka = WΦ(>)Ka(a b)Ka = WΦ(>)(a b)
3.
    3.1. (λf.((λf.((λx.(f (x x)))(λx.(f (x x))))) f)) = (λf.((λf.([(λx.(f (x x))λx.x](f (x x))))) f)) = = (λf.((λf.((f ((λx.(f (x x)) (λx.(f (x x))))))) f)) = (λf.((λf.((f ([(λx.(f (x x))λx.x](f (x x)))))) f)) = = (λf.((λf.((f (f ((λx.(f (x x)) (λx.(f (x x)))))))) f)) и так далее
    3.2. Y(K Ω)

ФП - парадигмы в программировании



ФП - синтаксис

Существует целый тип языков программирования (ЯП) которые построены на λ выражениях: функциональные и массивного типа. Самые известные функциональные: Haskell, F#, Kotlin. Масивного типа: APL, BQN, J, Uiua.

Table 7. I combinator.

Language	Name	Symbol
APL	Same	⊢
APL	Constant	⍨
BQN	Identity	⊢
BQN	Constant	˙
J	Same	]

Table 8. K combinator.

Language	Name	Symbol
APL	Right	⊢
BQN	Right	⊢
J	Right	]

Table 9. S combinator.

Language	Name	Symbol
APL	-	-
BQN	(Monadic) After	⟜
J	(Monadic) Hook	2-train

Table 10. B combinator.

Language	Name	Symbol
APL	(Monadic) Beside	∘
APL	(Monadic) Atop	⍤
APL	(Monadic) Over	⍥
APL	(Monadic) Atop	2-train
BQN	(Monadic) Atop	∘
BQN	(Monadic) Over	○
BQN	(Monadic) Atop	2-train
J	(Monadic) At	@:

Table 11. B_1 combinator.

Language	Name	Symbol
APL	(Dyadic) Atop	⍤
APL	(Dyadic) Atop	2-train
BQN	(Dyadic) Atop	∘
BQN	(Dyadic) Atop	2-train
J	(Dyadic) At	@:

Table 12. C combinator.

Language	Name	Symbol
APL	Commute	⍨
BQN	Swap	˜
J	Passive	~

Table 13. W combinator.

Language	Name	Symbol
APL	Commute	⍨
BQN	Self	˜
J	Reflex	~

ФП - комбинаторы

Table 14. Ψ combinator.

Language	Name	Symbol
APL	Over	⍥
BQN	Over	○
J	Appose	&:

Table 17. D combinator.

Language	Name	Symbol
APL	(Dyadic) Beside	∘
BQN	(Dyadic) After	⟜

Table 18. D_2 combinator.

Language	D_2
BQN	a⊸b⟜c
Extended Dyalog APL	a⍛b∘c

Vs

Table 19. Combinators in Haskell.

	Name	Library
I	id	Prelude
K	const	Prelude
W	join	Control.Monad
C	flip	Prelude
B	(.)	Prelude
S	(<*>) / ap	Control.Applicative
B_1	(.:)	Data.Composition
Ψ	on	Data.Function
Φ	liftA2	Control.Applicative
A	($)	Prelude

ФП - в практике

Давайте решим вместе задачу:

Дан массив чисел nums, упорядоченных по неубыванию. Требуется вернуть максимум между количеством положительных и количеством отрицательных чисел.

Другими словами, если количество положительных чисел в nums равно pos, а количество отрицательных чисел равно neg, то следует вернуть максимум между pos и neg.

Число 0 не считается ни положительным, ни отрицательным.

ФП - алгоритмы решения

1. Целочисленный двоичный поиск (просто линейный поиск тут же)
2. Одна редукция/рекурсия
   2.1. Цикл Для
   2.2. Редукционный алгоритм
3. Две редукции
   3.1. Фильтр/если и длина
   3.2. Генератор и длина
   3.3. Генератор счетчика

ФП - алгоритмы решения

Решение 2.1 C

```
int maximumCount(int* nums, int numsSize){
    int pos = 0, neg = 0;
    for (int i = 0; i < numsSize; ++i) {
        if (nums[i] > 0) ++pos;
        else if (nums[i] < 0) ++neg;
    }
    return pos > neg ? pos : neg;
}
```

ФП - алгоритмы решения

Решение 2.2

C++

```
auto maximum_count(std::vector<int> nums) -> int {
    auto const [pos, neg] = std::accumulate(
        nums.cbegin(), nums.cend(), std::pair{0, 0},
        [](auto p, auto e) {
            auto const [pos, neg] = p;
            return std::pair{pos + (e > 0), neg + (e < 0)};
        });
    return std::max(pos, neg);
}
```

ФП - алгоритмы решения

Решение 1

C++

```
int maximumCount(std:: vector<int> nums) {
    auto [a, b] = std::equal_range(nums.begin(), nums.end(), 0);
    return std::max(
        std::distance(nums.begin(), a),
        std::distance(b, nums.end()));
}
```

ФП - алгоритмы решения

Решение 3.1

C#

```
public int MaximumCount(int[] nums) {
    return Math.Max(
        nums.Count(x => x < 0),
        nums.Count(x => x > 0));
}
```

ФП - алгоритмы решения

Решение 3.2

Python

```python
def maximumCount(nums: List[int]) -> int:
    return max(
        len([x for x in nums if x < 0]),
        len([x for x in nums if x > 0]))
```

ФП - BQN

Давайте построим решение в BQN шаг за шагом.

Попытка 1:

```
MaximumCount ← {(+´0<𝕩)⌈+´0>𝕩}
```

X - наш массив, +’ - редукция сложения, ⌈ - максимум

Попытка 2:

```
MaximumCount ← (+´0<⊢)⌈(+´0>⊢)
```

Теперь вместо переменной у нас ⊢ (I комбинатор)

Попытка 3:

```
MaximumCount ← (0<⊢)⌈○(+´)(0>⊢)
```

○ - Ψ комбинатор

Попытка 4:

```
MaximumCount ← 0⊸(<⌈○(+´)>)
```

⊸ - левый B комбинатор

ФП - в практике

Еще одна задачка:

Дан массив чисел nums, и нужно вернуть массив из нечетных чисел.

C++

```cpp
auto filter_odds(std::vector<int> nums) {
  return nums
       | std::views::filter(
           [](auto e) { return e % 2 == 1; });
}
```

Python

```python
filter_odds = list(filter(lambda x: x % 2 == 1, nums))
```

ФП - BQN

А теперь в BQN:

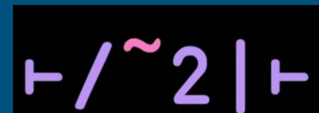
Попытка 1:

X - наш массив, 2| - маска единиц и нулей на 2x, / - вычитание маски x, получая маску нечетных чисел

Попытка 2:

⊢ (I комбинатор), ⊸ - левый B комбинатор, все вместе ⊸|/ это Φ комбинатор

Попытка 3:



Композиция Φ комбинатора и W комбинатора (~)

Попытка 4:

⊸|⊸/ -Все вместе это D комбинатор

Задания 6

Ссылка на редактор BQN: BQNPAD

1. Напишите программу в BQN которая принимает массив nums и возвращает сумму квадратов элементов
2. Напишите программу в BQN которая высчитывает среднеквадратичное отклонение массива
3. Напишите программу в BQN которая высчитывает сумму чисел главной диагонали квадратной матрицы (Подсказка: ⍉ значит транспозиция)
4. Напишите программу в BQN которая умножает две квадратные матрицы (Подсказка: ⎉ значит ранг матрицы)

Ответы 6

1.

```
SoSQ ← {+´×˜𝕩×0=(1+↕)⊸|≠𝕩}
```

```
SoSQ ← 2⋆˜⊢×0=·(1+↕)⊸|≠
```

2.

```
StDev ← (+´÷≠)⌾(√⁼)-⟜(+´÷≠)
```

3.

```
Trace ← +´0‿0⍉⊢
```

4.

```
Mult ← +˝∘×⎉1‿∞
```

Дополнительные задачи

1. Упростите следующие выражения:
   1.1. (λx.xyx)λz.z
   1.2. λx.λx.x
2. Запишите λ выражения в самой краткой комбинаторной форме:
   2.1. (λx.(λy.yx)(λz.xz))(λy.yy)
   2.2. (λx.xy)(λu.vuu)
   2.3. (λxyz.xz(yz))((λxy.yx)u)((λxy.yx)v)w
3. Какие из следующих записей эквивалентно этой (λxy.y(λx.xy)z)?
   3.1. (λxy. a(λx. xa)a)
   3.2. (λzy. y(λx. xy)z)
   3.3. (λxy. y(λz. zy)z)
   3.4. (λxy. y(λz. zy)a)
   3.5. (λxa. a(λa. aa)z)
   3.6. (λxa. a(λx. xa)a)
   3.7. (λxa. a(λz. za)z)
   3.8. (λza. a(λz. za)z)
4. Напишите комбинаторное выражение для формулы производной n степени, используя арифметику чисел Чёрча.

Ресурсы

Combinatory Logic and Combinators in Array Languages - Conor Hoekstra - conorhoekstra@gmail.com

Lambda Calculus - CSE 340 – Principles of Programming Languages

Lambda Calculus - Mattox Beckman - https://uiuc-cs421-sp22.netlify.app/handouts/lambda-calculus.pdf

Lambda Calculus - Steven Syrek - https://github.com/sjsyrek/presentations/blob/master/lambda-calculus/slides.md

Combinators - Pol Dellaiera - https://github.com/loophp/combinator/blob/master/README.md

Русская литература:

Ламбда-исчисления, его синтаксис и семантика - Х. Барендрегх

Комбинаторная логика - В. Э. Вольфенгаген - перевод

Спасибо за внимание!