ЛЕКЦИЯ 10
λ-Исчисление. Теоретические основы функционального программирования

Лямбда-исчисление

λ-исчисление – это набор формальных систем, основанных на нотации, которую придумал Алонзо Черч (A. Church) в 1930 г.

Исчисление анонимных функций

Лямбда-исчисление

$x-y$ можно рассматривать, как функцию

$f(x)$ и функцию $g(y)$

$f(x) = x-y$ $\qquad$ $g(y) = x-y$

$f: x \to x-y$ $\qquad$ $g: y \to x-y$

Scheme-аналог для f – (lambda (x) (- x y))

Scheme-аналог для g – (lambda (y) (- x y))

Нотация Черча:

$f \equiv \lambda x.\ x-y$ $\qquad$ $g \equiv \lambda y.\ x-y$

Где ≡ означает «записывается как …»

Настоящая нотация Чёрча

На самом деле Чёрч писал с «крышкой»: ^

$\hat{y}.\ x - y$

Есть версия, что при наборе его статьи в типографии не нашлось нужной литеры, поэтому напечатали так:

$\Lambda y.\ x - y$

Дальше при перепечатке заменили большую лямбду на маленькую:

$\lambda y.\ x - y$

Аппликация

Аппликация (apply, в Scheme -- композиция):

$f \equiv \lambda x.\ x{-}y$

$f(0)$ в нотации Чёрча: $(\lambda x.\ x{-}y)\ 0$ т. е. $0 - y$

на scheme: ((lambda (x) (− x y)) 0)

$g \equiv \lambda y.\ x{-}y$

$g(1)$ в нотации Чёрча: $(\lambda y.\ x{-}y)\ 1$ т. е. $x - 1$

на scheme: ((lambda (y) (- x y)) 1)

Всякая λ-функция – это функция от одного аргумента!

Заметим, что в записях вроде $\lambda x.\ x{-}y$ и $\lambda y.\ x{-}y$ мы не вполне придерживаемся нотации Чёрча. Чтобы применить функцию вычитания, следовало писать $\lambda x.\ -\ x\ y$ и $\lambda y.\ -\ x\ y$

Аппликация

Функции от двух и более аргументов в λ-нотации:

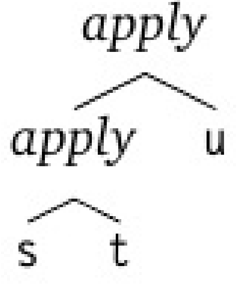
h(x, y) = x−y ==> h ≡ λx. λy. − x y

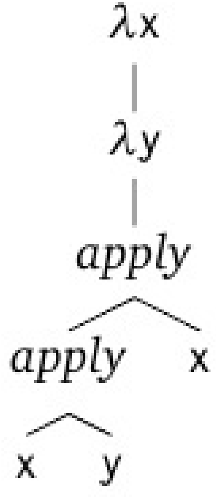
Аппликация левоассоциативная!

(λx. λy. − x y) 7 2 ==> ((λy. − 7 y) 2) ==> − 7 2

на Scheme (((lambda (x) (lambda (y) (- x y))) 7) 2)

s t u это ((s t) u)





Аппликация «заберёт всё, до чего дотянется»

λx. λy. x y x это λx. (λy. ((x y) x))

Каррирование / карринг

В Scheme/Racket мы использовали функции от многих переменных. Можно ли составлять программы с функциями от одной переменной? Или, точнее, можем ли получать результат составного вызова функций от одной переменной, такой же, как у вызова функции от n-переменных? Используем дополнительный модуль (require racket/function). В нём (curry proc) возвращает функцию, которая для n-арной функции proc вернёт функцию высшего порядка, которая возьмёт от 1 до n аргументов и вернёт снова функцию, добирающую остальное и делающую то же, что proc.

```
> (((curry list) 1 2 3) 4 5) => (1 2 3 4 5)
> (((curry ((curry list) 1)) 2) 3) => (1 2 3)
> (((curry list) 1 2 3)) => (1 2 3)
> (((curryr cons) 1) 2) => (2 . 1) ; curryr меняет порядок при склеивании
```

Формальное определение λ-термов

λ-терм (λ-выражение) – это:

- *переменная*; например, x
- *константа*; например, c1, *на самом деле их нет, но пусть будут*
- *комбинация*; или аппликация s t функции s к аргументу t, где s и t – λ-термы
- *абстракция* λx. s λ-терма s по переменной x; создание новой анонимной функции от переменной x с телом s.

БНФ для λ-термов:

<λ-терм> ::= <переменная> | <константа> |
<λ-терм> <λ-терм> | λ <переменная>. <λ-терм>

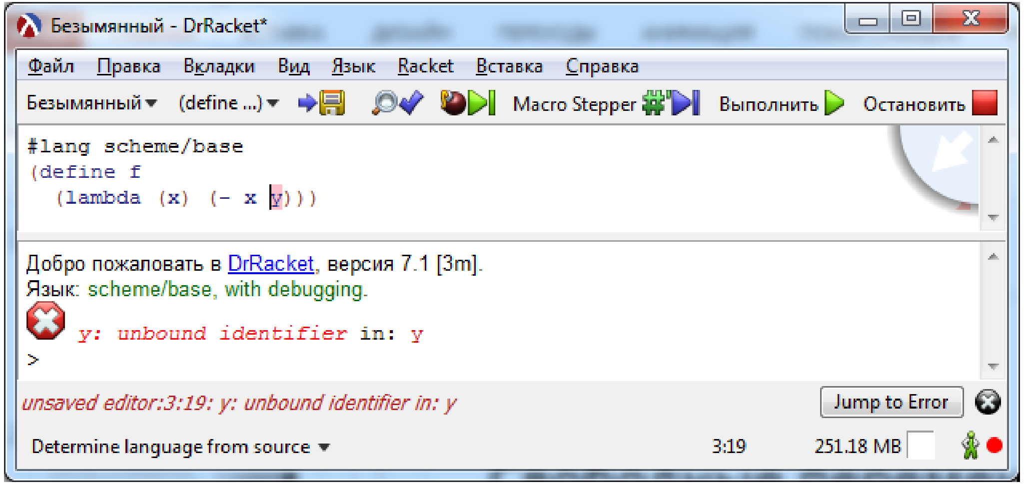
λ-нотация

Свободные переменные и связанные переменные

λx. – x y

x – связанная переменная

y – свободная переменная



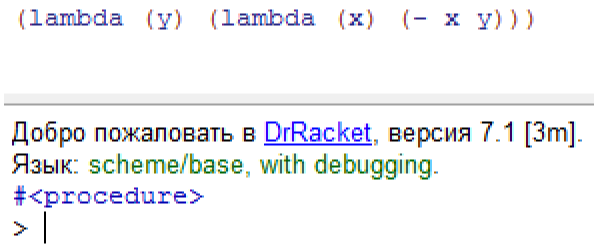
Абстракция терма s

по свободной переменной y

делает y связанной переменной

в новом терме:

λy. λx. – x y



абстракция может быть

по переменной, которой нет в s,

она тоже станет связанной λw. λx. – x y

Свободные и связанные вхождения переменных

Вхождение переменной x в λ-терм t является свободным, если оно лежит вне области действия соответствующей абстракции.

Формальное определение:

FV(t) – множество свободных переменных терма t:

$FV(x) = \{x\}$

$FV(c1) = \emptyset$

$FV(s\ t) = FV(s) \cup FV(t)$

$FV(\lambda x.\ s) = FV(s) - \{x\}$

BV(t) – множество связанных переменных терма t:

$BV(x) = \emptyset$

$BV(c1) = \emptyset$

$BV(s\ t) = BV(s) \cup BV(t)$

$BV(\lambda x.\ s) = BV(s) \cup \{x\}$

Пример: $s \equiv s1\ s2 \equiv (\lambda z.\ z\ x)\ (\lambda x.\ z\ x)$ где $s1 \equiv (\lambda z.\ z\ x)$; $s2 \equiv (\lambda x.\ z\ x)$

$FV(s1) = \{x\}$ $FV(s2) = \{z\}$

$FV(s) = FV(s1) \cup FV(s2) = \{x , z\}$

$BV(s1) = \{z\}$ $BV(s2) = \{x\}$

$BV(s) = BV(s1) \cup BV(s2) = \{x , z\}$

это *разные вхождения* z и x; перепишем $s' \equiv (\lambda w.\ w\ x)\ (\lambda y.\ z\ y)$ всё ОК

Подстановка

Подстановка $t[s/x]$ – это терм, получаемый из терма t заменой переменной x на терм s.

Пример: $(\lambda y.\ +\ x\ y)[5/x]$ даёт $\lambda y.\ +\ 5\ y$

Ещё пример: $(\lambda y.\ +\ x\ y)[y/x]$ даст $\lambda y.\ +\ y\ y$??? Нет! Даст $\lambda w.\ +\ y\ w$!!!

При подстановке следует учитывать свободные/связанные переменные.

Правила этого учёта:

$x[t/x] = t$ $\qquad$ $y[t/x] = y$, если $x \neq y$ $\qquad$ $c[t/x] = c$

$(s\ u)[t/x] = s[t/x]\ u[t/x]$

$(\lambda x.\ s)[t/x] = \lambda x.\ s$

$(\lambda y.\ s)[t/x] = \lambda y.\ (s[t/x])$, если $x \neq y$, и либо $x \notin FV(s)$, либо $y \notin FV(t)$

$(\lambda y.\ s)[t/x] = \lambda z.\ (s[z/y][t/x])$, $x \neq y$ и $x \in FV(s)$ или $y \in FV(t)$,

причём $z \notin FV(s) \cup FV(t)$

Преобразования λ-выражений

- α-редукция «переименование связанной переменной»
  - будем обозначать $\xrightarrow{\alpha}$
  - если v и w – переменные, а t – λ-терм, то

  $\lambda v.\ t \xrightarrow{\alpha} \lambda w.\ t[w/v]$, если $w \notin FV(t)$
- Пример: $\lambda x.\ \lambda y.\ x - y \xrightarrow{\alpha} \lambda x.\ \lambda v.\ x - v$ можно

  но нельзя $\lambda u.\ u\ v \cancel{\xrightarrow{\alpha}} \lambda v.\ v\ v$

Преобразования λ-выражений

- β-редукция «вычисление функции для заданного аргумента»
  - Будем обозначать $\xrightarrow{\beta}$
  - $(\lambda x.\ s)\ t \xrightarrow{\beta} s[t/x]$
- Пример:

$(\lambda x.\ (\lambda y.\ -\ x\ y))\ 7\ 2 \xrightarrow{\beta} (\lambda y.\ -\ 7\ y)\ 2 \xrightarrow{\beta} -\ 7\ 2$

```
(((lambda (x) (lambda (y) (- x y))) 7) 2)
```

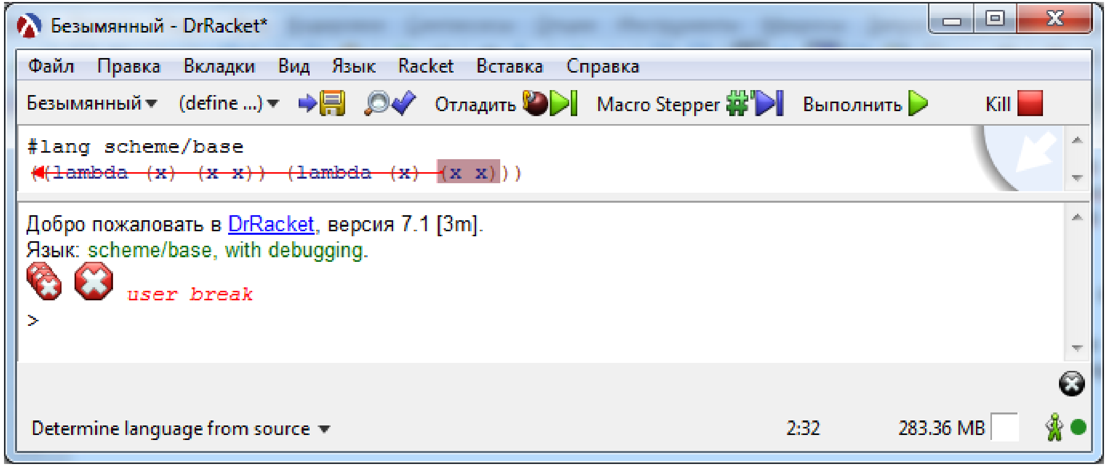
```
Добро пожаловать в DrRacket, версия 7.1 [3m].
Язык: scheme/base, with debugging.
5
>
```

- Бывает ещё η-редукция, но мы её не рассматриваем.

Примеры β-редукции

- (λx. (λy. y x) z) v -β-> (λy. y v) z -β-> z v
- (λx. x x) (λx. x x) -β-> (λx. x x) (λx. x x) -β-> (λx. x x) (λx. x x) -β-> ...

  -β-> (λx. x x) (λx. x x) -β-> ...



- (λx. x x y) (λx. x x y) -β-> (λx. x x y) (λx. x x y) y -β-> (λx. x x y) (λx. x x y) y y

  -β-> ...

Эквивалентность

- Термы t и v эквивалентны (записывается t = v), если есть цепочка термов s, r, ..., такая что t ---> s ---> r ---> ... ---> v, где каждая ---> является либо $\xrightarrow{\alpha}$, либо $\xrightarrow{\beta}$, либо «обратной β-редукцией» $\xleftarrow{\beta}$
- Справедливо, что t = t

если s = t, то t = s	есть симметричность
если s = t и t = v, то s = v	есть транзитивность
если s = t, то s u = t u	сохраняется при аппликации
если s = t, то u s = u t	сохраняется при аппликации
если s = t, то λx. s = λx. t	сохраняется при абстракции

- Эквивалентность – не тождественность:

  λx. x = λy. y но они не тождественны (написание разное)

λ-Редукция

- λ-редукция, это «эквивалентность в одну сторону» (без «обратной β-редукции», т. е. без $\xleftarrow{\beta}$)

  $t \xrightarrow{\lambda} t$

  если $s \xrightarrow{\lambda} t$ и $t \xrightarrow{\lambda} v$, то $s \xrightarrow{\lambda} v$

  если $s \xrightarrow{\lambda} t$, то $s\ u \xrightarrow{\lambda} t\ u$

  если $s \xrightarrow{\lambda} t$, то $u\ s \xrightarrow{\lambda} u\ t$

  если $s \xrightarrow{\lambda} t$, то $\lambda x.\ s \xrightarrow{\lambda} \lambda x.\ t$

- термин λ-редукция соотносится с понятием «вычисление функциональной программы»

Нормальная форма

λ-выражение находится в нормальной форме, если ни одна β-редукция не может быть применена.

- Для выражения λx. ((λy. y x) z) v нормальная форма z v
- Для выражения (λx. x x) (λx. x x) нормальной формы нет, т. к. β-редукцию можно применить, а выражение не поменяется.
- Для выражения (λx. x x y) (λx. x x y) нормальной формы нет, т. к. β-редукцию можно применить, а в результате всегда будет подвыражение к которому применима β-редукция.

Нормальная форма

- В каком порядке делать редукции? Влияет ли порядок на результат?
- Например, обозначим L ≡ (λx. x x y) (λx. x x y)
- Найдем н. ф. выражения (λu. v) L
  - (λu. v) L = v, если начинаем слева
  - но можно всё время делать справа и циклиться:

  (λu. v) L $\xrightarrow{\beta}$ (λu. v) (L y) $\xrightarrow{\beta}$ (λu. v) (L y y) $\xrightarrow{\beta}$ (λu. v) (L y y y) $\xrightarrow{\beta}$ …
- Выражение (λx. x x) (λx. x x) нормальной формы не имеет (выбор стратегии не спасает).
- Мы ранее видели зацикливание при запуске кода

  ((lambda (x) (x x)) (lambda (x) (x x)))

Стратегии редукции

- Теорема: Если s $\xrightarrow{\lambda}$ t, где t имеет нормальную форму, то последовательность редукций, в которой всегда выбирается самое левое редуцируемое выражение, приводит к терму в нормальной форме.
- Это нормальный порядок редукции.
- При нормальном порядке редукции (левая) функция применяется до вычисления её аргументов.

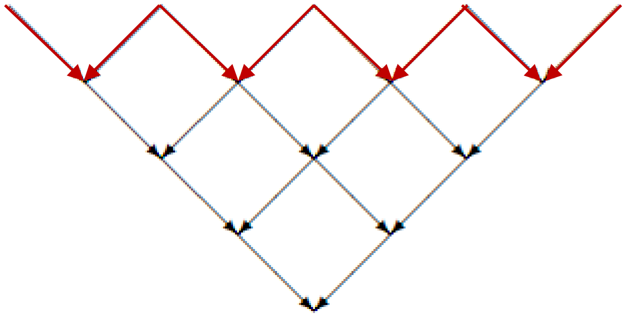
Теорема Черча-Россера

Если t $\xrightarrow{\lambda}$ u и t $\xrightarrow{\lambda}$ w, то существует терм v: u $\xrightarrow{\lambda}$ v и w $\xrightarrow{\lambda}$ v.

Теорема об эквивалентности

- Если p = q, то существует выражение t, такое что p $\xrightarrow{\lambda}$ t и q $\xrightarrow{\lambda}$ t
- Схема доказательства:

p q



t

Верхний «зигзаг» существует, так как p = q. Теорема Черча-Россера позволяет достроить низ картинки.

Т. эквивалентности и т. Черча-Россера – это разные теоремы!

Единственность нормальной формы

Следствие теоремы об эквивалентности:
Если t = v и t = w, причём v и w находятся в нормальной форме, то v = w, причём цепочка редукций состоит только из α-редукций.

Значит, если нормальная форма существует, то она единственна с точностью до α-редукций (переименований связанных переменных)!

Для выражения λx. ((λy. y x) z) нормальная форма λx. (z x) или λw. (z w) или λy. (z y) …
Переименовывать связанные переменные в нормальной форме можно по-разному. Переименовывать свободные переменные нельзя. Нельзя переименовывать связанные так, чтобы их новые имена совпадали с именами свободных переменных.

Комбинаторы

Комбинатор – λ-терм без свободных переменных.

Примеры:

- I ≡ λx. x (тождественность)

```
> (define I (lambda (x) x))
> (I 1) => 1
> (I 2) => 2
> (identity 1) => 1 ; identity -- это аналог I из racket/function
```

- K ≡ λx. λy. x (константа a: K a => λy. a)

```
> (define K (lambda (x) (lambda (y) x)))
> ((K 10) 2) => 10
> ((K 10) 3) => 10
> ((const 10) 4) => 10 ; const – это аналог K из racket/function
> ((const 10) 5 6 7) => 10
```

результат вызова const забирает любое количество аргументов

Ещё комбинаторы

- S ≡ λf. λg. λx. (f x) (g x) (выделение)

> (define S (lambda (f) (lambda (g) (lambda (x) ((f x) (g x))))))

для примера введём доп. функции:

> (define (square x) (* x x)) ; возводит в квадрат аргумент

> (define (f-x x) (lambda (y) (- y x))) ; даёт функцию, вычитающую x из своего аргумента

> (((S f-x) square) 10) =>

90

т. е. (- (*10 10) 10)

Другая запись примера

> (define ((S1 f) g) (lambda (x) ((f x) (g x))))

> (((S1 (curryr -)) square) 5) =>

20

SKI-исчисление

Для любого λ-терма существует его эквивалент без λ-абстракций, являющийся композицией I, S, K и переменных. Комбинаторы S, K, I можно рассматривать как аналог машинных команд для λ-выражений.

Очевидно, что I = S K K, поэтому можно говорить о SK-исчислении

```
> ((S K) K) 10) => 10 ; (I 10)
> ((S K) K) 1) => 1     ; (I 1)
```

действительно ((S K) K) x) = ((K x) (K x)) = x = (I x)

j – комбинатор Криса Баркера (йота-комбинатор) -- минимальная база

```
> (define (j f) ((f S) K)) ; (define (j f) ((f S) const))
> ((j j) 10) => 10 ; это I
> (((j (j (j j))) 5) 10) => 5 ; это K
> (define S2 (j (j (j (j j))))) ; это S
> (((S2 (curryr -)) square) 5) => 20 ; работает
```

Снова комбинаторы

- B-комбинатор: B ≡ λf. λg. λx. (f (g x)) композиция

> (define (((B f) g) x) (f (g x)))
> (((B ((curryr -) 5)) square) 6) => 31 т. е. (- (* 6 6) 5)

Известно, что B = S (K S) K

Scheme даёт (((((S (K S)) K) ((curryr -) 5)) square) 6) => 31

- C-комбинатор: C ≡ λf. λy. λx. ((f x) y) пермутация, что-то вроде curryr

> (define (((C f) y) x) ((f x) y))
((((C (curryr -)) 10) 6) => 4 т. е. (- 10 6), не (- 6 10)

Известно, что C = S (B B S) (K K)

Scheme даёт (((((S ((B B) S)) (K K)) (curryr -)) 10) 6) => 4

- W-комбинатор: W ≡ λx. λy. ((x y) y) дублирование

(define ((W x) y) ((x y) y))))
((W (curry +)) 10) => 20 т. е. (+ 10 10) -- удвоение

известно, что W = S S (K (S K K))

Комбинатор неподвижной точки

- комбинатор неподвижной точки – это комбинатор, применив который к функции f, мы получим неподвижную точку x для f , то есть такое x, что f (x) = x.
- $Y \equiv \lambda f.\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))$ – комбинатор Карри

  $Y\ g = (\lambda f.\ (\lambda x.\ f\ (x\ x))(\lambda x.\ f\ (x\ x)))\ g = (\lambda x.\ g\ (x\ x))(\lambda x.\ g\ (x\ x)) =$

  $= g\ ((\lambda x.\ g\ (x\ x))\ (\lambda x.\ g\ (x\ x))) = g\ (Y\ g) = g\ (g\ (Y\ g)) = g(\ldots g(Y\ g)\ \ldots)$
- Y-комбинатор – основа для рекурсивных функций там, где обычной рекурсии нет. Scheme-описание:

```
> (define (Y f) ((lambda (x) (x x)) (lambda (g) (f (lambda args (apply (g g)
    args))))))
> (define n! (Y (lambda (f) (lambda (n) (if (< n 2) 1 (* n (f (- n 1))))))))
> (n! 6) =>
  720
```

Y комбинатор. Продолжение

- В n! имеются остаточные вычисления.

```
(define n! (Y (lambda (f) (lambda (n) (if (< n 2) 1 (* n (f (- n 1))))))))
```

- Перепишем без них, заведя result.

```
(define n!v2 (lambda (n)
  ((Y (lambda (f) (lambda (i result) (if (< i 2) result  (f (- i 1) (* i result)))))) n 1)))
```

- «Любимые» наши числа Фибоначчи

```
(define fib (lambda (n)
   ((Y (lambda (f)
          (lambda (i fib-n-1 fib-n-2)
             (if (= i 0) fib-n-2  (f (- i 1) (+ fib-n-1 fib-n-2) fin-b-1)))))
        n 1 0)))
```

Арифметика в λ-исчислении

- $0 \equiv \lambda f.\ \lambda x.\ x$
- $1 \equiv \lambda f.\ \lambda x.\ f\ x$
- $2 \equiv \lambda f.\ \lambda x.\ f\ (f\ x)$
- $3 \equiv \lambda f.\ \lambda x.\ f\ (f\ (f\ x))$
- …

Арифметика в λ-исчислении

- SUCC $\equiv \lambda n.\ \lambda f.\ \lambda x.\ f\ (n\ f\ x)$
- PLUS $\equiv \lambda m.\ \lambda n.\ \lambda f.\ \lambda x.\ m\ f\ (n\ f\ x)$

т. е. PLUS $\equiv \lambda n.\ \lambda m.\ m$ SUCC n

- MULT $\equiv \lambda m.\ \lambda f.\ m\ (n\ f)$

т. е. MULT $\equiv \lambda m.\ \lambda n.\ m$ (PLUS n) 0

Пример

PLUS 2 3 ≡ (λ*m*. λ*n*. λ*f*. λ*x*. *m f* (*n f x*)) (λ*f*. λ*x*. *f* (*f x*)) (λ*f*. λ*x*. *f* (*f* (*f x*))) $\xrightarrow{\beta}$

(λ*n*. λ*f*. λ*x*. (λ*f*. λ*x*. *f* (*f x*)) *f* (*n f x*)) (λ*f*. λ*x*. *f* (*f* (*f x*))) $\xrightarrow{\alpha}$

(λ*n*. λ*f*. λ*x*. (λ*a*. λ*b*. *a* (*a b*)) *f* (*n f x*)) (λ*f*. λ*x*. *f* (*f* (*f x*))) $\xrightarrow{\beta}$

(λ*n*. λ*f*. λ*x*. (λ*b*. *f* (*f b*)) (*n f x*)) (λ*f*. λ*x*. *f* (*f* (*f x*))) $\xrightarrow{\beta}$

(λ*n*. λ*f*. λ*x*. *f* (*f* (*n f x*))) (λ*f*. λ*x*. *f* (*f* (*f x*))) $\xrightarrow{\beta}$

(λ*f*. λ*x*. *f* (*f* ((λ*f*. λ*x*. *f* (*f* (*f x*))) *f x*))) $\xrightarrow{\alpha}$

(λ*f*. λ*x*. *f* (*f* ((λ*a*. λ*b*. *a*(*a* (*a b*))) *f x*))) $\xrightarrow{\beta}$

(λ*f*. λ*x*. *f* (*f* (*f*(*f* (*f x*))))) – это 5

Логика в λ-исчислении

- TRUE ≡ $\lambda x.\ \lambda y.\ x$ *функция от x и y, возвращающая x*
- FALSE ≡ $\lambda x\ .\ \lambda y.\ y$ *функция от x и y, возвращающая y*
- NOT ≡ $\lambda p.\ \lambda a.\ \lambda b.\ p\ b\ a$

NOT TRUE ≡ $(\lambda p.\ \lambda a.\ \lambda b.\ p\ b\ a)\ (\lambda x.\ \lambda y.\ x) \dashrightarrow (\lambda a.\ \lambda b.\ (\lambda x.\ \lambda y.\ x)\ b\ a) \dashrightarrow$ $(\lambda a.\ \lambda b.\ (\lambda y.\ b)\ a) \dashrightarrow (\lambda a.\ \lambda b.\ b) \dashrightarrow (\lambda x.\ \lambda y.\ y)$ ≡ FALSE

аналогично NOT FALSE ---> TRUE

- AND ≡ $\lambda p.\ \lambda q.\ p\ q\ p$

AND FALSE w ≡ $(\lambda p.\ \lambda q.\ p\ q\ p)\ (\lambda x\ .\ \lambda y.\ y)\ w \dashrightarrow (\lambda q.\ (\lambda x\ .\ \lambda y.\ y)\ q\ (\lambda x\ .\ \lambda y.\ y))\ w$ $\dashrightarrow (\lambda q.\ \lambda y.\ y\ (\lambda x\ .\ \lambda y.\ y))\ w \dashrightarrow \lambda y.\ y\ (\lambda x\ .\ \lambda y.\ y) \dashrightarrow \lambda x\ .\ \lambda y.\ y$ ≡ FALSE

аналогично AND w FALSE ---> FALSE

AND TRUE TRUE ≡ $(\lambda p.\ \lambda q.\ p\ q\ p)\ (\lambda x\ .\ \lambda y.\ x)\ (\lambda x\ .\ \lambda y.\ x) \dashrightarrow$ $(\lambda a\ .\ \lambda b.\ a)\ (\lambda x\ .\ \lambda y.\ x)\ (\lambda x\ .\ \lambda y.\ x) \dashrightarrow (\lambda b.\ (\lambda x\ .\ \lambda y.\ x))\ (\lambda x\ .\ \lambda y.\ x) \dashrightarrow (\lambda x\ .\ \lambda y.\ x)$

Логика в λ-исчислении

- TRUE ≡ $\lambda x.\ \lambda y.\ x$ *функция от x и y, возвращающая x*
- FALSE ≡ $\lambda x\ .\ \lambda y.\ y$ *функция от x и y, возвращающая y*
- OR ≡ $\lambda p.\ \lambda q.\ p\ p\ q$

OR TRUE w ≡ $(\lambda p.\ \lambda q.\ p\ p\ q)\ (\lambda x.\ \lambda y.\ x)$ w ---> $(\lambda x.\ \lambda y.\ x)\ (\lambda x.\ \lambda y.\ x)$ w --->
$(\lambda a.\ \lambda b.\ a)\ (\lambda x.\ \lambda y.\ x)$ w ---> $(\lambda b.\ (\lambda x.\ \lambda y.\ x))$ w ---> $(\lambda x.\ \lambda y.\ x)$ ≡ TRUE

аналогично OR w TRUE ---> TRUE

OR FALSE FALSE ≡ $(\lambda p.\ \lambda q.\ p\ p\ q)\ (\lambda x\ .\ \lambda y.\ y)\ (\lambda x\ .\ \lambda y.\ y)$ --->
$(\lambda x\ .\ \lambda y.\ y)\ (\lambda x\ .\ \lambda y.\ y)\ (\lambda x\ .\ \lambda y.\ y)$ ---> $(\lambda x\ .\ \lambda y.\ y)$ ≡ FALSE

- IFTHENELSE ≡ $\lambda p.\ \lambda a.\ \lambda b.\ p\ a\ b$

проверяется аналогично