

Глава 3. Стили функционального программирования

3.1. Язык ЛИСП (John McCarthy, 1960)

Две формы представления данных:

□ атомы: A 123 B_12 + T NIL

□ списки: (A) (PLUS 12 25) (LAMBDA (X Y) (PLUS X Y)) ()

С помощью атомов представляются «атомарные» объекты – числа, символы и логические значения;

С помощью списков представляются составные объекты – структуры, выражения и функции;

Атомы могут обладать значением. Например, атом PLUS имеет значением функцию сложения, а атом 123 имеет значением самого себя. Списки тоже имеют значение – значение списка «вычисляется» по специальным правилам.

Вызов функции – применение функции к списку аргументов:

```
(PLUS 12 15)
```

```
(PLUS (MINUS 12 5) 15)
```

```
(COND ((LT X Y) (MINUS 0 1))
```

```
((EQ X Y) 0)
```

```
(T 1))
```

```
(QUOTE (LAMBDA (X) (PLUS 1 X))) ' (LAMBDA (X) (PLUS 1 X))
```

```
(LET (FACT 5)
```

```
(FACT '(LAMBDA (N) (COND ((EQ N 0) 1)
```

```
(T (MULT N (FACT (MINUS N 1))))))))))
```

Основные примитивные функции в ЛИСПе

Арифметические и логические функции: PLUS, MULT, LE, EQ, AND

Специальные функции: COND, QUOTE, LET

Функции обработки списков: CAR, CDR, CONS

(CONS 'A '(B C))	□	(A B C)
(CONS 'A (CONS 'B NIL))	□	(A B)
(CAR '(A B C))	□	A
(CDR '(A B C))	□	(B C)
(CDR '(A))	□	NIL
(CONS '(A B) '(C D))	□	((A B) C D)
(CAR (CDR '(A B C D)))	□	B
(CADDR '(A B C D))	□	C
(CONS 'LAMBDA (CONS '(X) '((PLUS X 1))))	□	(LAMBDA (X) (PLUS X 1))

Функции высших порядков в ЛИСПе:

```
(DEFINE MAP (LAMBDA (F L)
              (COND (L (CONS (F (CAR L)) (MAP F (CDR L))))
                    (T NIL)
              )))
```

(MAP '(LAMBDA (X) (PLUS X 1)) '(1 3 8)) □ (2 4 9)

Функции как значения

Функциональное значение – это либо встроенная функция, либо список, первым атомом которого является специальный атом LAMBDA. Например,

```
(LAMBDA (X) (MULT 2 X))
```

Если функция применяется к аргументу, то формальные параметры «связываются» со значением аргумента, поэтому, например, выражение

```
((LAMBDA (X) (MULT 2 X)) 3)
```

при вычислении выдает значение 6.

Как происходит вычисление, если в теле функции используются «глобальные» атомы?

```
(LAMBDA (X) (MULT Y X))
```

Два возможных подхода:

- статический: значения «глобальных» атомов фиксируются в момент определения функции (в момент «исполнения» функции LAMBDA).
- динамический: значения «глобальных» атомов определяются во время работы функции.

Разница может проявиться, если функция определена в одном контексте, а исполняется в другом.

В некоторых (старых) версиях ЛИСПа для «фиксации» контекста применялись специальные функции, например, если лямбда-выражение передавалось в качестве аргумента в другую функцию, то можно было использовать специальную функцию FUNCTION вместо QUOTE.

Подведение итогов: основные черты и особенности языка ЛИСП

- Простой синтаксис – скобочная запись данных и программ.
- Энергичная схема вычислений (кроме специальных функций).
- Нет образцов и конструкторов – определение функций построено на суперпозиции примитивных и ранее определенных функций; построение сложных объектов также использует функцию CONS .
- Возможность определения функций высших порядков, в которых аргументами и/или результатом работы являются функции.
- Возможность динамического построения фрагментов программы непосредственно в ходе ее выполнения.
- Определение «контекста переменных» с помощью блочной структуры (LET).

Язык ЛИСП послужил прообразом и идейной основой большой группы языков. В той или иной степени он лежит в основе всех современных функциональных языков программирования и многих других языков.

3.2. Система комбинаторного программирования FP (John Backus, 1978)

Теоретически любую функцию можно получить из базовых с помощью комбинаторных преобразований. FP – система, реализующая этот принцип.

Набор констант не определен, но предполагается, что есть списки и логические значения. Набор базовых функций не определен, но предполагается, что он достаточно богат. Также предполагается, что принята энергичная схема вычислений (все функции – строгие).

Определим следующие комбинаторы:

- Композиция $f \circ g$
 $(f \circ g) x = f (g x)$
- Условие $f \rightarrow g; h$
 $(f \rightarrow g; h) x = \begin{cases} g x, & \text{если } f x - \text{ истинно} \\ h x, & \text{если } f x - \text{ ложно} \\ \text{не опр.}, & \text{если } f x - \text{ не определено} \end{cases}$
- Константа \underline{k}
 $\underline{k} x = \begin{cases} k, & \text{если } x - \text{ определено} \\ \text{не опр.}, & \text{если } x - \text{ не определено} \end{cases}$
- Конструкция $[f_1, f_2, \dots, f_n]$
 $[f_1, f_2, \dots, f_n] x = \langle f_1 x, f_2 x, \dots, f_n x \rangle$

Еще несколько важных комбинаторов

□ Отображение α f **(map)**

$$(\alpha f) : \langle x_1, x_2, \dots, x_n \rangle = \langle f x_1, f x_2, \dots, f x_n \rangle$$

□ Правая вставка $/$ f **(foldr1)**

$$(/ f) : \langle x \rangle = x$$

$$(/ f) : \langle x_1, x_2, \dots, x_n \rangle = f : \langle x_1, (/ f) : \langle x_2, \dots, x_n \rangle \rangle$$

□ Левая вставка \backslash f **(foldl1)**

$$(\backslash f) : \langle x \rangle = x$$

$$(\backslash f) : \langle x_1, x_2, \dots, x_n \rangle = f : \langle (\backslash f) : \langle x_1, x_2, \dots, x_{n-1} \rangle, x_n \rangle$$

□ Вариант правой вставки с базовой константой $/_k$ f **(foldr)**

$$(/_k f) : \langle \rangle = k$$

$$(/_k f) : \langle x_1, x_2, \dots, x_n \rangle = f : \langle x_1, (/_k f) : \langle x_2, \dots, x_n \rangle \rangle$$

□ Вариант левой вставки с базовой константой \backslash_k f **(foldl)**

$$(\backslash_k f) : \langle \rangle = k$$

$$(\backslash_k f) : \langle x_1, x_2, \dots, x_n \rangle = f : \langle (\backslash_k f) : \langle x_1, x_2, \dots, x_{n-1} \rangle, x_n \rangle$$

Определим типичный язык программирования на основе FP-системы

Введем набор констант и базовых функций

Константы:

- целые числа (0, 1, 2, ..., -1, -2, ...);
- логические значения (Т и F);
- символы ('s', '*', ...);
- гетерогенные списки (<>, <1, <1, 2>, Т>, ...)

*в программах в явном виде
не присутствуют!*

Функции:

Арифметические функции: +, -, *, add1, mul5, sub3:

+ : <2,5> = 7

add1 : 3 = 4

Логические функции: =, /=, <, >, ..., and, or, not, ... eq0, neq5, lt2:

= : <2,5> = F

< : <3,4> = T

lt2 : 5 = T

or : <T,F> = T

Функции-селекторы: 1, 2, ..., 1r, 2r, ...:

2 : <2,5,7> = 5

1r : <3,4,8> = 8

Тождественная функция: id:

id : <3,5> = <3,5>

Функции обработки списков

hd : <3,5,8> = 3

tl : <3,5,8> = <5,8>

hr : <3,5,8> = 8

tr : <3,5,8> = <3,5>

cons : <3, <5,4>> = <3,5,4>

consr : <<5,4>, 3> = <5,4,3>

null : <> = T

null : <3,5> = F

distl : <3, <1,2,5>> = <<3,1>, <3,2>, <3,5>>

distr : <<1,2,5>, 3> = <<1,3>, <2,3>, <5,3>>

l : 5 = <1,2,3,4,5>

l : 0 = <>

Программа в *FP* – это определение функций:

def sqr = * ◦ [id, id] sqr : 5 = * : <5, 5> = 25

def pow4 = * ◦ [sqr, sqr] pow4 : 3 = * : <sqr : 3, sqr : 3> = 81

Примеры программ в языке программирования на базе *FP*

Haskell: `fact n = if n = 0 then 1 else n * fact (n-1)`

FP: `def fact = eq0 → 1; (* ∘ [id, fact ∘ (- ∘ [id, 1])])`
`def fact = (/1 *) ∘ ↓`

Haskell: `fact n = foldr (*) 1 [1..n]`

Haskell: `test elem [] = False`
`test elem (x:lst) | x == elem = True`
`| otherwise = test elem lst`

FP: `def test = null ∘ 2 → E; eq ∘ [1, hd ∘ 2] → I; test ∘ [1, tl ∘ 2]`
`def test = (/F or) ∘ (α eq) ∘ distl`

`test : <5, <1,7,5,2>>`
`(/F or) : ((α eq) : (distl : <5, <1,7,5,2>>))`
`(/F or) : ((α eq) : <<5,1>, <5,7>, <5,5>, <5,2>>)`
`(/F or) : <F,F,T,F>`
`T`

Haskell: `test elem = (foldr (||) False) . (map (== elem))`

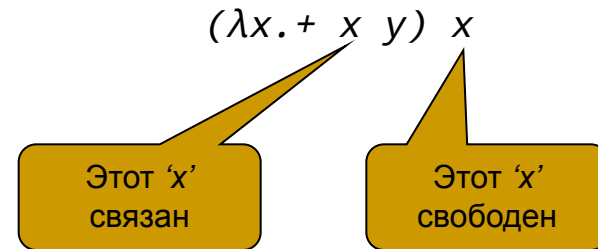
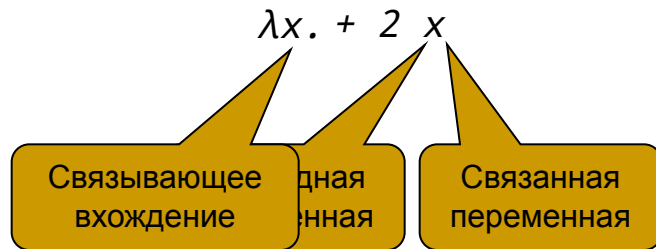
Глава 4. Основы лямбда-исчисления.

4.1. Основные понятия

Будем задавать функции с помощью «лямбда-выражений», которые будем преобразовывать по определенным правилам (правила «вычислений»).

Основные типы лямбда-выражений:

□ переменная	x			
□ константа	c	$+$	3	nil
□ применение функции	$(e1\ e2)$	$((+ 2) 3)$		
□ определение функции	$(\lambda x. e)$	$(\lambda x. ((+ 2) x))$		
		$(\lambda x. (x x))$	$(\lambda x. (x x))$	



Замкнутое выражение – выражение, не содержащее свободных переменных.

Правила преобразований выражений.

1. δ -редукция

$$f\ e_1\ e_2 \dots e_k \rightarrow result$$

$$+ 3\ 5 \rightarrow 8$$

$$or\ T\ F \rightarrow T$$

2. β -редукция

$$(\lambda x. e_1)\ e_2 \rightarrow e_1\{x/e_2\}$$

$$(\lambda x. +\ 1\ x)\ 5 \rightarrow +\ 1\ 5$$

$$(\lambda x. x\ x)(\lambda x. x\ x) \rightarrow (\lambda x. x\ x)(\lambda x. x\ x)$$

3. α -преобразование

$$\lambda x. e_1 \leftrightarrow \lambda z. e_1\{x_{cb.}/z\}$$

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

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

4. η -преобразование

$$\lambda x. E\ x \leftrightarrow E$$

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

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

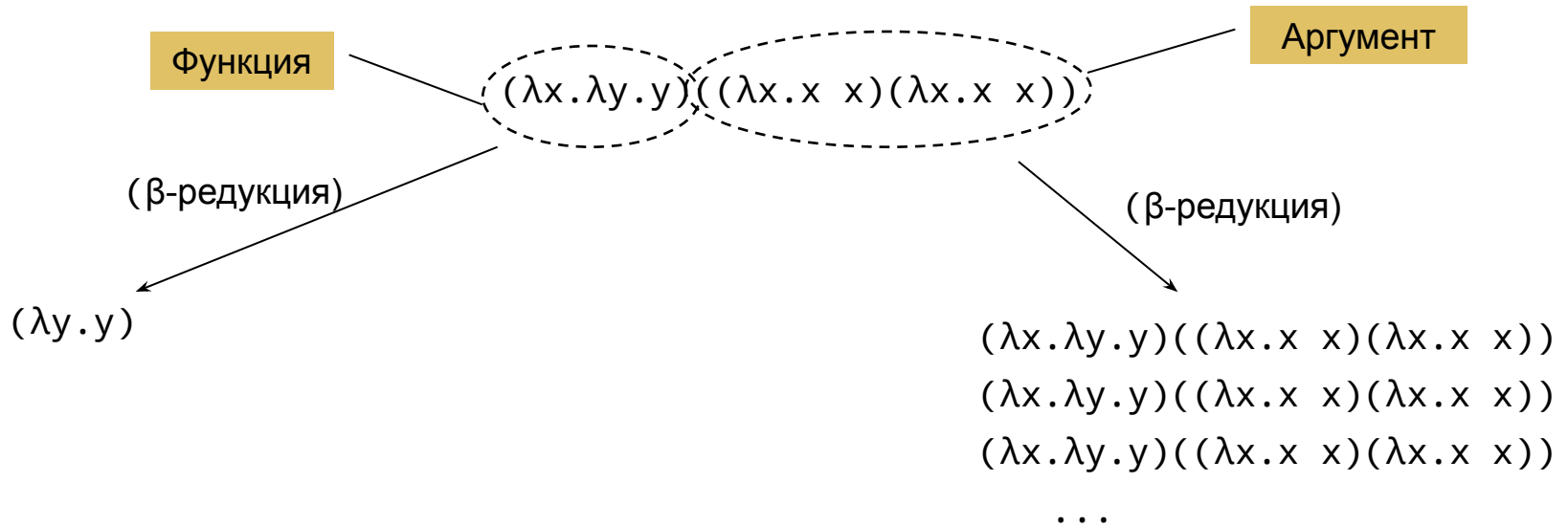
Примеры преобразования выражений

$(\lambda x. \lambda y. + x y) 2 5$

$(\lambda y. + 2 y) 5$ (β -редукция)

$+ 2 5$ (β -редукция)

7 (δ -редукция)



От порядка применения редукций может зависеть результат!

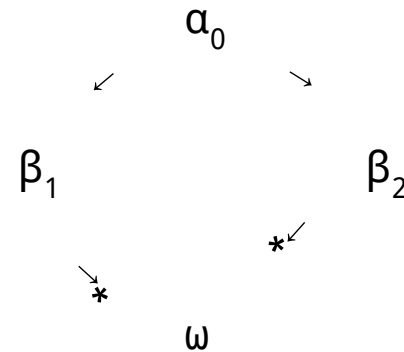
Ромбическое свойство системы редукций

Нормальная форма: лямбда-выражение, к которому не применимы β - и δ -редукции

$$\alpha_0 \rightarrow \alpha_1 \rightarrow \dots \rightarrow \alpha_k$$

$$\alpha_0 \rightarrow^* \alpha_k$$

Отношение \rightarrow^* это
рефлексивное*транзитивное
замыкание отношения \rightarrow



существует (?) форма ω

Теорема (Черча – Россела): система преобразований, основанная на применении β -редукций обладает ромбическим свойством.

Следствие: лямбда-выражение не может иметь более одной нормальной формы.

Замечание: в некоторых случаях применение редукций может, в зависимости от порядка, либо приводить к нормальной форме, либо не приводить к ней.

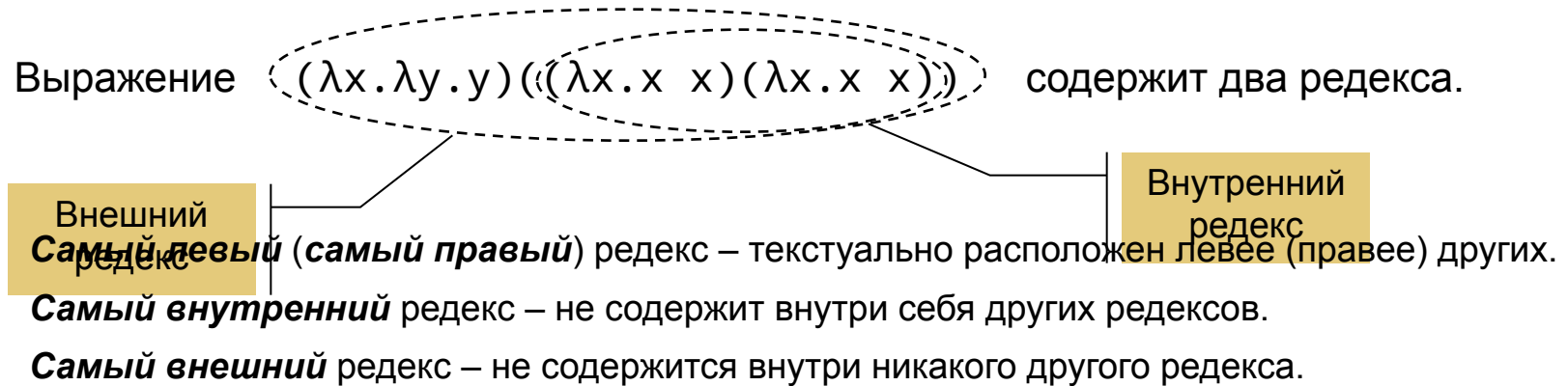
Стандартные порядки редукций.

Редекс (redex) – выражение, к которому применима одна из редукций.

Reducible Expression – редуцируемое выражение. β -редекс; δ -редекс...

Выражение может содержать (или не содержать) один или несколько редексов.

Выражение $(\lambda x. \lambda y. y) ((\lambda x. x \ x) (\lambda x. x \ x))$ содержит два редекса.

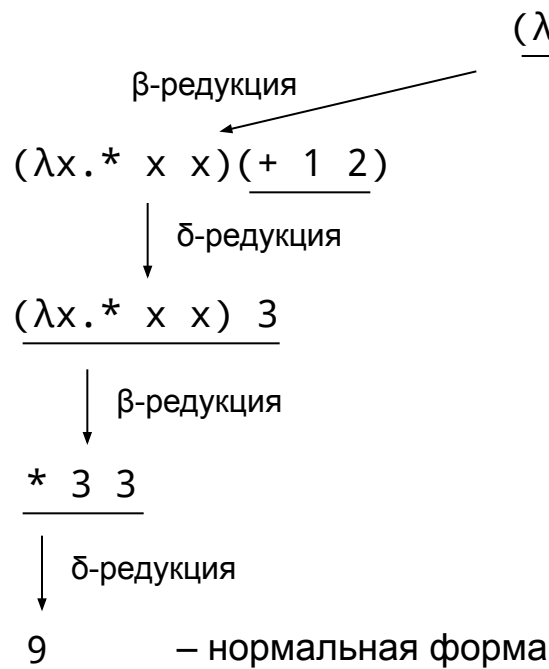


Аппликативный порядок редукций – редукция всегда применяется к самому левому из **самых внутренних** редексов

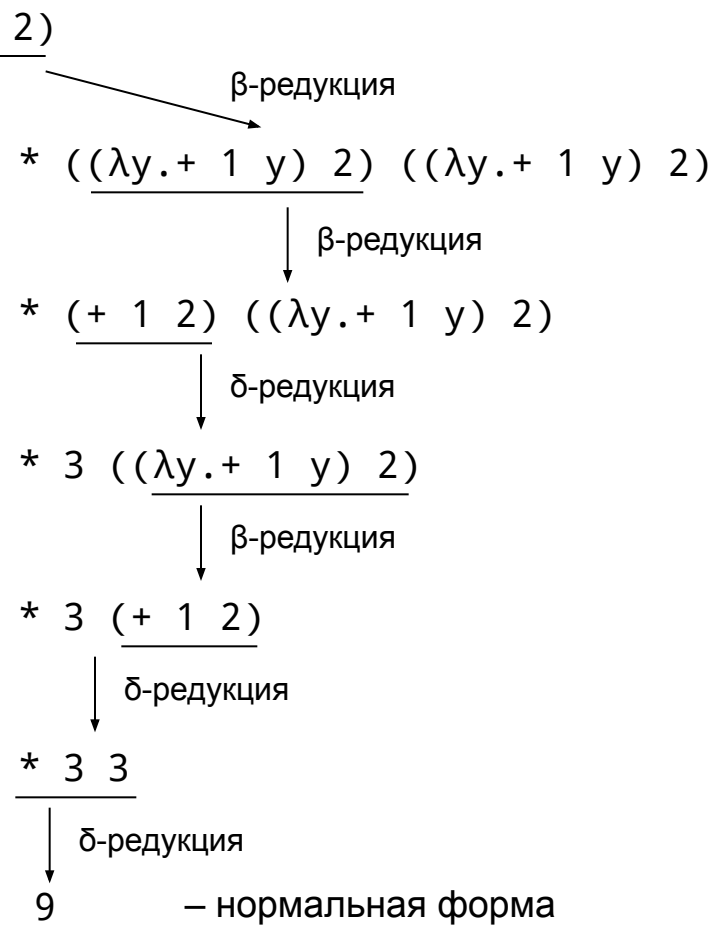
Нормальный порядок редукций – редукция всегда применяется к самому левому из **самых внешних** редексов

Еще пример редукции выражения

Аппликативный порядок редукций



Нормальный порядок редукций



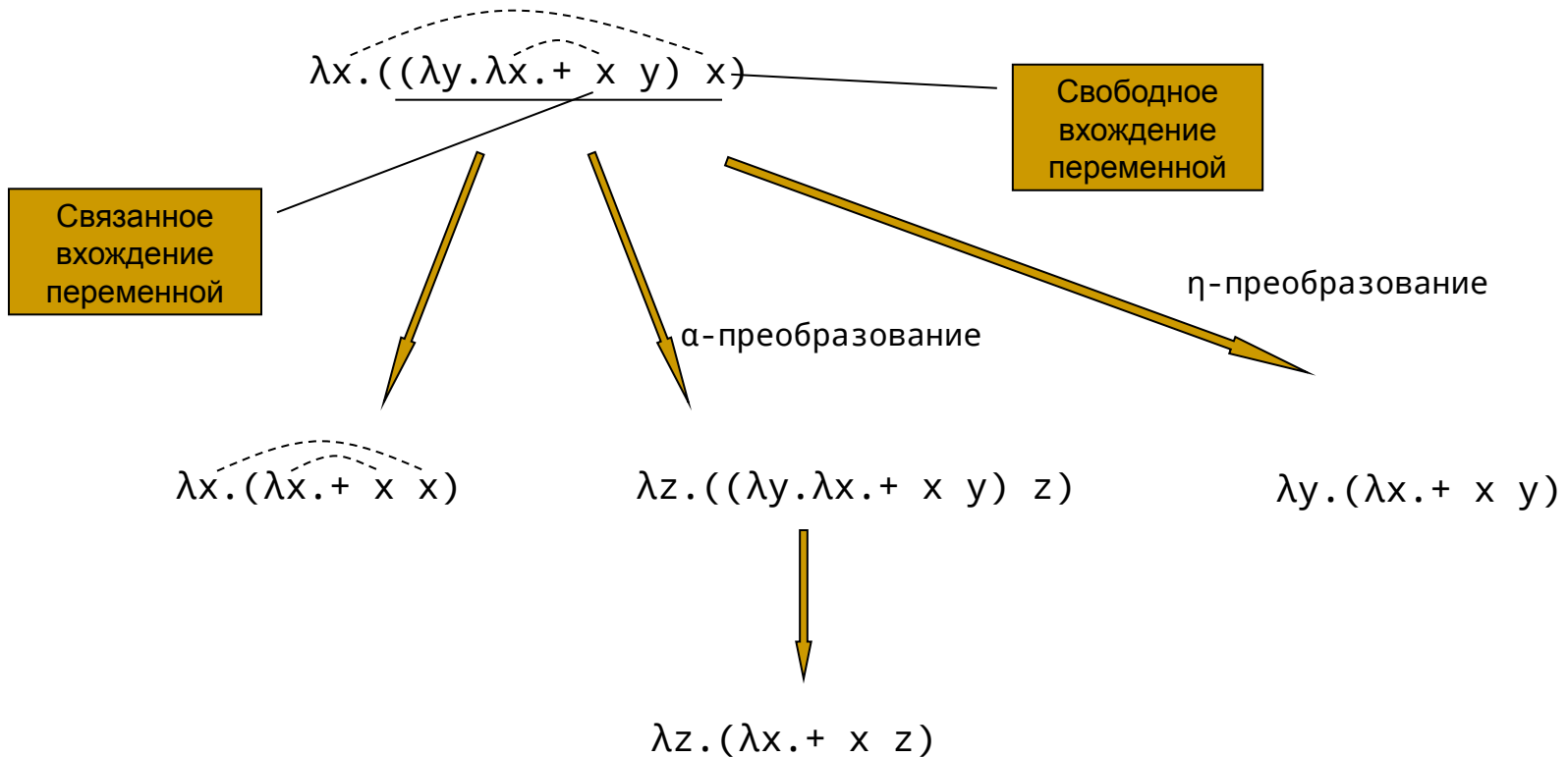
Сравнение различных порядков редукций.

Аппликативный порядок (АПР)	Нормальный порядок (НПР)
Не приводит к копированию одних и тех же выражений, если они не находятся в нормальной форме.	Одни и те же выражения аргументов могут многократно копироваться при подстановке в тело лямбда-выражения.
Выполняет редукцию даже тех выражений, которые в дальнейшем могут быть отброшены.	Никогда не редуцирует выражение, если это может оказаться впоследствии ненужным.
Может не приводить к нормальной форме, даже если она существует.	Всегда приводит выражение к нормальной форме, если только она вообще существует.

Преобразование выражения к нормальной форме в АПР соответствует энергичному порядку вычислений выражений в языках программирования.

Преобразование выражения к нормальной форме в НПР соответствует вычислениям выражений с подстановкой аргументов «по наименованию».

Проблема конфликта имен.



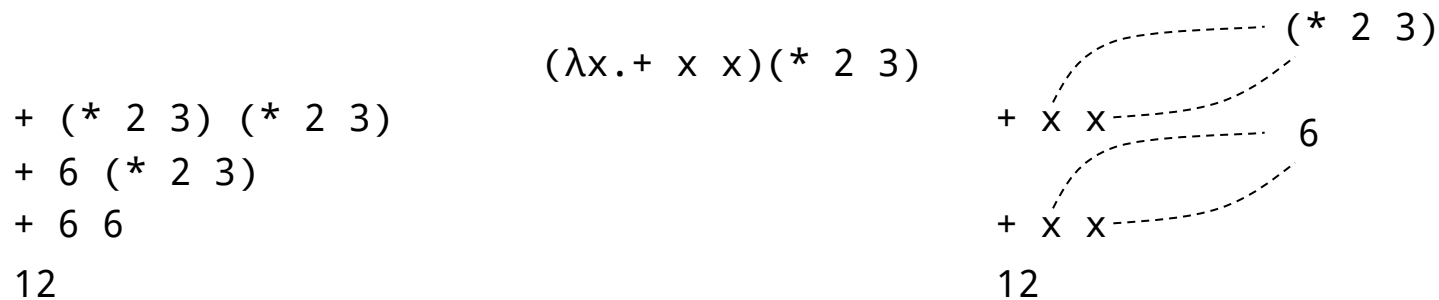
Слабая заголовочная нормальная форма (СЗНФ)

Выражение, не имеющее свободных переменных (замкнутое) находится в СЗНФ, если оно имеет один из следующих видов:

- Константа c
- Определение функции $\lambda x. E$
- Частичное применение функции $P E_1 E_2 \dots E_k$

Выражение $\lambda x. ((\lambda y. \lambda x. + x y) x)$ находится в СЗНФ.

Вычисления, происходящие при исполнении программы в «ленивых» вычислениях, соответствуют редукции выражения в НПР до приведения к СЗНФ, дополненной эффектом «разделения» значений переменных при подстановке аргумента, еще не находящегося в СЗНФ.



Итоги: механизмы редукций в лямбда-исчислении

1. От способа применения редукций может зависеть окончательный вид выражения, но не его функциональный смысл.
2. Если выражение может быть приведено к нормальной форме, то это может быть сделано с помощью редукций в НПР, возможно, с переименованиями переменных.
3. Аппликативный порядок редукций, приводящий выражение к СЗНФ соответствует энергичной схеме вычислений, принятой в некоторых функциональных языках.
4. «Ленивая» схема вычислений может быть смоделирована приведением выражений к СЗНФ при применении НПР + разделение переменных.