Программирование в лямбда-исчислении

Дмитрий Халанский

7 сентября 2020 г.

(Питон умеет возвращать функции из функций)

print печатает свой аргумент. exit завершает исполнение программы.

```
def returningFunctions(n):
  if n % 2 == 0:
    return print
  else:
    return exit
```

В консоли:

```
>>> returningFunctions(6)("xyzzy")
xyzzy
>>> returningFunctions(7)(3)
$ echo $?
3
```

Трансляция

Лямбда-терм строится одним из трёх способов, каждому из которых мы сопоставим код на Python:

Лямбда-исчисление	**Python**
Переменная x	`x`
Аппликация $a\,b$	**a(b)**
Абстракция $\lambda x.\, M$	**def f(x): return M** **f** (или **lambda x: M**)

Примеры

- $\lambda x.\, x$

Примеры

- $\lambda x.x$

```
def i(x):
    return x
i
```

- $\lambda y.y$

Примеры

- $\lambda x.x$

```
def i(x):
    return x
i
```

- $\lambda y.y$

```
def i2(y):
    return y
i2
```

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

Примеры

- $\lambda x.x$

```
def i(x):
    return x
i
```

- $\lambda y.y$

```
def i2(y):
    return y
i2
```

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

```
def k(y):
    def helper(x):
        return y
    return helper
k
```

- $x\ x$

Примеры

- $\lambda x.x$

```
def i(x):
    return x
i
```

- $\lambda y.y$

```
def i2(y):
    return y
i2
```

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

```
def k(y):
    def helper(x):
        return y
    return helper
k
```

- $x\ x$

```
x(x)
```

(Для сравнения: `print(print)`)

Лямбда-абстракции — функции от одного аргумента

Рассмотрим

$$\mathsf{K}_* := \lambda x\ y.\, y$$

Не совсем правильная трактовка: “Функция, которая принимает два аргумента и возвращает второй”. Закодируем её в Питоне:

```
def kStarWrong(x, y):
  return y
```

Согласно бета-эквивалентности, $\mathsf{K}_*\ x =_\beta \lambda y.\, y$, но

```
>>> kStarWrong(3)
TypeError: kStarWrong() missing 1 required positional
  argument: 'y'
```

Правильная трактовка: “Функция, которая игнорирует свой аргумент и возвращает функцию, которая возвращает свой аргумент”.

Каррирование

$$
\begin{aligned}
\mathsf{I} &:= \lambda x.\, x \\
\mathsf{K} &:= \lambda x\ y.\, x \\
\mathsf{K}_* &:= \lambda x\ y.\, y
\end{aligned}
$$

```
def i(x):
  return x

def k(x):
  def tmp(y):
    return x
  return tmp

def kStar(x):
  return i

i(2) == 2
k(3)(4) == 3
kStar("x")(True) == True
```

Пример структуры данных: кошка

Python

```
class Cat:
  def __init__(self, name, numberOfLegs):
    self.name = name
    self.numberOfLegs = numberOfLegs
```

C++

```
struct Cat {
    string name;
    uint8_t numberOfLegs;
};
```

Java

```
public class Cat {
    public String name;
    public int numberOfLegs;
}
```

Что мы хотим от структуры данных?

Конструкторы (“создатели”): создание новых экземпляров.

```
barsik = Cat(name = "barsik", numberOfLegs = 4)
```

Элиминаторы (“сокращатели”): получение информации из экземпляров.

```
barsik.numberOfLegs
```

Мутаторы (“менятели”): изменение информации в экземплярах.

```
barsik.numberOfLegs = barsik.numberOfLegs - 1
```

Что мы хотим от структуры данных?

Конструкторы (“создатели”): создание новых экземпляров.

```
barsik = Cat(name = "barsik", numberOfLegs = 4)
```

Элиминаторы (“сокращатели”): получение информации из экземпляров.

```
barsik.numberOfLegs
```

Мутаторы (“менятели”): изменение информации в экземплярах.

```
barsik.numberOfLegs = barsik.numberOfLegs - 1
```

Не нужно: на этом курсе рассматриваем языки, в которых вместо мутатирования принято копировать:

```
newBarsik = Cat(name = barsik.name,
  numberOfLegs = barsik.numberOfLegs - 1)
barsik = newBarsik
```

Законы структуры данных

Хотим, чтобы элиминаторы возвращали то, что конструктор положил в экземпляр:

```
Cat(name = s, numberOfLegs = n).numberOfLegs == n
Cat(name = s, numberOfLegs = n).name == s
```

Это полностью описывает всё, что нужно знать про кошек как структуру данных.

Обобщённый элиминатор

Для такой структуры данных можно ввести один элиминатор, через который выражаются остальные. Для этого заведём функции

```
def eliminateCat(cat):
  def tmp(f):
    return f(cat.name)(cat.numberOfLegs)
  return tmp
```

Тогда можно произвести везде такие замены:

$$\begin{aligned} \texttt{cat.numberOfLegs} &\to \texttt{eliminate(cat)(kStar)} \\ \texttt{cat.name} &\to \texttt{eliminate(cat)(k)} \end{aligned}$$

Законы в терминах обобщённого элиминатора

```
def eliminateCat(cat):
  return lambda f: f(cat.name)(cat.numberOfLegs)
```

Перепишем старые законы:

```
eliminateCat(Cat(name = s, numberOfLegs = n))(kStar) == n
eliminateCat(Cat(name = s, numberOfLegs = n))(k) == s
```

Конструкцию eliminateCat(Cat(s, n)) можно упростить:

```
def eliminateCat2(s, n):
  return lambda f: f(s)(n)
```

Получается, нам не нужен сам Cat? Законы выполняются и так:

```
eliminateCat2(s, n)(kStar) == n
eliminateCat2(s, n)(k) == s
```

Пример структуры данных: логическая величина

Python	True и False
C++	true и false
Java	**true** и **false**

Конструкторы и элиминаторы логических величин

Конструктора два: значение истины и значение лжи.
Логическое значение несёт только информацию о том, истинное оно или ложное. Значит, элиминатор может проверять только это.
Что является элиминатором логической величины?

Конструкторы и элиминаторы логических величин

Конструктора два: значение истины и значение лжи.
Логическое значение несёт только информацию о том, истинное оно или ложное. Значит, элиминатор может проверять только это.
Что является элиминатором логической величины? if.

Законы для логических величин

```
if True: A() else: B() == A()
if False: A() else: B() == B()
```

Обобщённый элиминатор для логических величин

```
def eliminateBool(b):
  def tmp(fTrue, fFalse):
    if b:
      return fTrue()
    else:
      return fFalse()
  return tmp
```

Везде конструкцию `if b then: A() else: B()` можно заменить на eliminateBool(b)(A, B).

```
eliminateBool(True)(A, B) == A()
eliminateBool(False)(A, B) == B()
```

1. Трансляция лямбда-исчисления в Python

2. Конструкторы и элиминаторы
   - Типы-записи
   - Логические величины

3. Кодирование структур данных в лямбда-исчислении
   - Типы-записи
   - Логические величины
   - Натуральные числа

1. Трансляция лямбда-исчисления в Python

2. Конструкторы и элиминаторы
   - Типы-записи
   - Логические величины

3. Кодирование структур данных в лямбда-исчислении
   - Типы-записи
   - Логические величины
   - Натуральные числа

Кошки в лямбда-исчислении

$$\begin{aligned}
\textbf{newCat} &:= \lambda s\ n\ f.\ f\ s\ n \\
\textbf{name} &:= \lambda\ \text{cat}.\ \text{cat}\ \mathsf{K} \\
\textbf{numberOfLegs} &:= \lambda\ \text{cat}.\ \text{cat}\ \mathsf{K}_*
\end{aligned}$$

$$\begin{aligned}
\textbf{name}\ (\textbf{newCat}\ \text{“}barsik''\ 4) &=_\beta \text{“}barsik'' \\
\textbf{numberOfLegs}\ (\textbf{newCat}\ \text{“}barsik''\ 4) &=_\beta 4
\end{aligned}$$

Для сравнения:

```
def eliminateCat2(s, n):
  return lambda f: f(s)(n)

eliminateCat(Cat(name = s, numberOfLegs = n))(kStar) == n
eliminateCat(Cat(name = s, numberOfLegs = n))(k) == s
```

Пары в лямбда-исчислении

$$\begin{aligned}\textbf{pair} &:= \lambda a\ b\ f.\ f\ \ a\ b\\ \textbf{fst} &:= \lambda p.\ p\ \mathsf{K}\\ \textbf{snd} &:= \lambda p.\ p\ \mathsf{K}_*\end{aligned}$$

Логические величины в лямбда-исчислении

Для сравнения:

```
def eliminateBool(b):
  def tmp(fTrue, fFalse):
    if b:
      return fTrue()
    else:
      return fFalse()
  return tmp

eliminateBool(True)(T, E) == T()
eliminateBool(False)(T, E) == E()
```

$$\begin{aligned} \textbf{true} &:= \\ \textbf{false} &:= \\ \textbf{if} &:= \end{aligned}$$

$$\begin{aligned} \textbf{if true } t\ e &=_\beta t \\ \textbf{if false } t\ e &=_\beta e \end{aligned}$$

Логические величины в лямбда-исчислении

Для сравнения:

```
def eliminateBool(b):
  def tmp(fTrue, fFalse):
    if b:
      return fTrue()
    else:
      return fFalse()
  return tmp

eliminateBool(True)(T, E) == T()
eliminateBool(False)(T, E) == E()
```

$$\begin{aligned} \textbf{true} &:= \lambda t\ e.\ t \\ \textbf{false} &:= \lambda t\ e.\ e \\ \textbf{if} &:= \lambda b\ t\ e.\ b\ t\ e \end{aligned}$$

$$\begin{aligned} \textbf{if true}\ t\ e &=_\beta t \\ \textbf{if false}\ t\ e &=_\beta e \end{aligned}$$

Операции над логическими величинами (1)

Отрицание

x	$\neg x$
true	false
false	true

На Python:

Операции над логическими величинами (1)

Отрицание

x	$\neg x$
true	false
false	true

На Python:

```
def neg(b):
  if b:
    return False
  else:
    return True
```

В лямбда-исчислении:

Операции над логическими величинами (1)

Отрицание

x	$\neg x$
true	false
false	true

На Python:

```
def neg(b):
  if b:
    return False
  else:
    return True
```

В лямбда-исчислении:

$$\textbf{neg} := \lambda b.\, \textbf{if}\ b\ \textbf{false}\ \textbf{true} = \lambda b.\, b\ \textbf{false}\ \textbf{true}$$

$$\textbf{neg} := \lambda b\ t\ e.\, b\ e\ t = \textbf{C}$$

Операции над логическими величинами (2)

Конъюнкция

x	y	$x \wedge y$
true	true	true
true	false	false
false	true	false
false	false	false

На Python:

Операции над логическими величинами (2)

Конъюнкция

x	y	$x \wedge y$
true	true	true
true	false	false
false	true	false
false	false	false

На Python:

```
def andb(x, y):
  if x:
    return y
  else:
    return False
```

В лямбда-исчислении:

Операции над логическими величинами (2)

Конъюнкция

x	y	$x \wedge y$
true	true	true
true	false	false
false	true	false
false	false	false

На Python:

```
def andb(x, y):
  if x:
    return y
  else:
    return False
```

В лямбда-исчислении:

$$\textbf{and} := \lambda x\ y.\, \textbf{if}\ x\ y\ \textbf{false} = \lambda x\ y.\, x\ y\ \textbf{false}$$

$$\textbf{and} := \lambda x\ y\ t\ e.\, x\ (y\ t\ e)\ e$$

Конструкторы и элиминаторы натуральных чисел

$$\mathbb{N} := \{0, 1, 2, 3, \cdots\}$$

Конструкторы **Первая аксиома Пеано**: ноль является натуральным числом;

Вторая аксиома Пеано: число, следующее за натуральным, тоже является натуральным.

Элиминаторы Фундаментальное свойство числа n — наша способность досчитать до n.

Обобщённый элиминатор натуральных чисел

Простая версия:

```
def eliminateNat(n, f):
  i = 0
  while (i < n):
    f()
    i = i + 1
```

Обобщённый элиминатор натуральных чисел

Простая версия:

```
def eliminateNat(n, f):
  i = 0
  while (i < n):
    f()
    i = i + 1
```

Проблема: хочется, чтобы элиминатор что-то возвращал. Заводим какое-то дополнительное состояние, которое и меняет функция.

```
def eliminateNat(n):
  def tmp(incrementFunction, initialValue):
    i = 0
    currentValue = initialValue
    while (i < n):
      currentValue = incrementFunction(currentValue)
      i = i + 1
    return currentValue
  return tmp
```

Числа Чёрча

$$
\begin{aligned}
\mathbf{0} &:= \lambda s\ z.\, z \\
\mathbf{1} &:= \lambda s\ z.\, s\ z \\
\mathbf{2} &:= \lambda s\ z.\, s\ (s\ z) \\
&\vdots \\
\forall k, \mathbf{k} &:= \lambda s\ z.\, \underbrace{s\ (s\ (\cdots s\ (}_{k}\ z\ \underbrace{))\cdots)}_{k}
\end{aligned}
$$

Операции над числами Чёрча (1)

Увеличить на единицу Хотим получить **n+1**.

$$\begin{aligned}
\mathbf{n{+}1} &= \lambda s\ z.\ \underbrace{s\ (s\ (\cdots (s}_{n+1}\ (z)\ \underbrace{\cdots))}_{n+1} \\
&= \lambda s\ z.\ s\ (\underbrace{s\ (\cdots (s}_{n}\ (z)\ \underbrace{\cdots)}_{n}) = \lambda s\ z.\ s\ (\mathbf{n}\ s\ z) \\
\mathbf{n{+}1} &= \lambda s\ z.\ \underbrace{s\ (s\ (\cdots (s}_{n+1}\ (z)\ \underbrace{))\cdots))}_{n+1} \\
&= \lambda s\ z.\ \underbrace{s\ (\cdots\ (s}_{n}\ (s\ z)\ \underbrace{\cdots))}_{n} = \lambda s\ z.\ \mathbf{n}\ s\ (s\ z)
\end{aligned}$$

$$\mathbf{suc} := \lambda\ n\ s\ z.\ s\ (n\ s\ z)$$

$$\mathbf{suc} := \lambda\ n\ s\ z.\ n\ s\ (s\ z)$$

Операции над числами Чёрча (2)

Сложение Хотим получить **n**+**m**.

- Это всё равно что n раз увеличить m на единицу:

$$\textbf{plus} := \lambda n\ m.\ n\ \textbf{suc}\ m$$

- Это всё равно что сначала отсчитать m, а потом от этого отсчитать n:

$$\textbf{plus} := \lambda n\ m\ s\ z.\ n\ s\ (m\ s\ z)$$

Можно реализовать что угодно: умножение, возведение в степень, вычитание, взятие остатка от деления и всё такое; это пойдёт в домашнее задание.