Функциональное программирование

Лекция 1

Степан Львович Кузнецов

НИУ ВШЭ, факультет компьютерных наук

О курсе

- Курс посвящён функциональной парадигме программирования на примере одного из наиболее известных функциональных языков — Haskell.

О курсе

- Курс посвящён функциональной парадигме программирования на примере одного из наиболее известных функциональных языков — Haskell.
- На лекциях: теоретические основы функционального программирования (λ-исчисление, выведение типов, монады-лимонады и проч.).

О курсе

- Курс посвящён функциональной парадигме программирования на примере одного из наиболее известных функциональных языков — Haskell.
- На лекциях: теоретические основы функционального программирования (λ-исчисление, выведение типов, монады-лимонады и проч.).
- На практических занятиях (Даниил Рогозин и Юрий Сыровецкий): программирование на Haskell'е.

О курсе

- Курс посвящён функциональной парадигме программирования на примере одного из наиболее известных функциональных языков — Haskell.
- На лекциях: теоретические основы функционального программирования (λ-исчисление, выведение типов, монады-лимонады и проч.).
- На практических занятиях (Даниил Рогозин и Юрий Сыровецкий): программирование на Haskell'е.
- Теория и практика связаны между собой: как в шутку говорят, *Haskell — это язык, в котором нельзя напечатать "Hello, World" без знания теории категорий.*

Функции как объекты

- Функциональная парадигма программирования существенно отличается от обычной (императивной).

Функции как объекты

- Функциональная парадигма программирования существенно отличается от обычной (императивной).
- Мы постепенно будем обсуждать её особенности.

Функции как объекты

- Функциональная парадигма программирования существенно отличается от обычной (императивной).
- Мы постепенно будем обсуждать её особенности.
- **Первое свойство,** объясняющее термин *«функциональный»:* функции являются полноправными «гражданами» (объектами) языка. Functions are first-class citizens.

Функции как объекты

- Функциональная парадигма программирования существенно отличается от обычной (императивной).
- Мы постепенно будем обсуждать её особенности.
- **Первое свойство,** объясняющее термин *«функциональный»:* функции являются полноправными «гражданами» (объектами) языка. Functions are first-class citizens.
- В частности, функция может быть передана как аргумент другой функции. В этом случае последняя называется *функцией высшего порядка.*

Функция как объект в C

- Начнём с примеров функций высших порядков, которые встречаются в императивных языках.

Функция как объект в C

- Начнём с примеров функций высших порядков, которые встречаются в императивных языках.
- Так, в стандартной библиотеке C есть функция сортировки:

```c
void qsort(void *base, size_t nmemb, size_t size,
 int (*compar)(const void *, const void *));
```

Функция как объект в C

- Начнём с примеров функций высших порядков, которые встречаются в императивных языках.
- Так, в стандартной библиотеке C есть функция сортировки:

```
void qsort(void *base, size_t nmemb, size_t size,
 int (*compar)(const void *, const void *));
```

- Эта функция может сортировать массив *произвольных данных.*

Функция как объект в C

- Начнём с примеров функций высших порядков, которые встречаются в императивных языках.
- Так, в стандартной библиотеке C есть функция сортировки:

```c
void qsort(void *base, size_t nmemb, size_t size,
 int (*compar)(const void *, const void *));
```

- Эта функция может сортировать массив *произвольных данных*.
- Отсюда тип `void*` — указатель на произвольный объект. (При этом не производится проверка корректности типов данных, что плохо.)

Функция как объект в C

- Функция `compar` возвращает значение < 0, если первый аргумент меньше второго, $= 0$, если равны, и > 0, если второй аргумент меньше.

Функция как объект в C

- Функция `compar` возвращает значение < 0, если первый аргумент меньше второго, $= 0$, если равны, и > 0, если второй аргумент меньше.
- Например, для сравнения строк (`char*`) по алфавиту используется функция `strcmp` с соответствующим приведением типов:

```
int cmpstringp(const void *p1, const void *p2)
{
  return strcmp(*(const char **) p1, *(const char **) p2);
}
```

Функция как объект в C

- Функция compar возвращает значение < 0, если первый аргумент меньше второго, $= 0$, если равны, и > 0, если второй аргумент меньше.
- Например, для сравнения строк (`char*`) по алфавиту используется функция strcmp с соответствующим приведением типов:

```
int cmpstringp(const void *p1, const void *p2)
{
  return strcmp(*(const char **) p1, *(const char **) p2);
}
```

- Технически передача функции как аргумента реализуется в C как передача указателя на место в памяти, где находится код этой функции — так что сгенерировать новую функцию «на лету» не получится.

Функция как объект в Python'е

- Аналогично устроена сортировка в Python'е:

```
bigrams = {"AB": [10, 11, 12], "BC": [5, -5, 8],
  "CD": [105, 1, 0], "DE": [6, 6], "EF": [15, 20, 15],
  "FG": [22, 11, 32], "GH": [20, 20, 20]}
srtbg = sorted(bigrams, key=lambda key: sum(bigrams[key]),
  reverse=True)
```

Функция как объект в Python'е

- Аналогично устроена сортировка в Python'е:

```
bigrams = {"AB": [10, 11, 12], "BC": [5, -5, 8],
  "CD": [105, 1, 0], "DE": [6, 6], "EF": [15, 20, 15],
  "FG": [22, 11, 32], "GH": [20, 20, 20]}
srtbg = sorted(bigrams, key=lambda key: sum(bigrams[key]),
  reverse=True)
```

- Здесь ради эффективности используется не функция сравнения, а функция вычисления ключа (которые потом сравниваются как целые числа).

Функция как объект в Python'е

- Аналогично устроена сортировка в Python'е:

```
bigrams = {"AB": [10, 11, 12], "BC": [5, -5, 8],
  "CD": [105, 1, 0], "DE": [6, 6], "EF": [15, 20, 15],
  "FG": [22, 11, 32], "GH": [20, 20, 20]}
srtbg = sorted(bigrams, key=lambda key: sum(bigrams[key]),
  reverse=True)
```

- Здесь ради эффективности используется не функция сравнения, а функция вычисления ключа (которые потом сравниваются как целые числа).
- Интересно использование ключевого слова `lambda` для создания безымянной функции «на месте».

λ-оператор

- Знаком λ выделяется та переменная, которую мы будем считать аргументом функции.

λ-оператор

- Знаком λ выделяется та переменная, которую мы будем считать аргументом функции.
- В теоретическом материале мы будем использовать обозначение $\lambda x.u$, где u — выражение (*терм*), возможно содержащее x:

$$\lambda x.\underbrace{\boxed{\ \ldots x \ldots x \ldots x \ldots}}_{u}$$

λ-оператор

- Знаком λ выделяется та переменная, которую мы будем считать аргументом функции.
- В теоретическом материале мы будем использовать обозначение $\lambda x.u$, где u — выражение (*терм*), возможно содержащее x:

$$\lambda x.\underbrace{\boxed{\ldots x \ldots x \ldots x \ldots}}_{u}$$

- При вычислении значения функции $\lambda x.u$ на аргументе $x = a$ нужно подставить a вместо x вместо всех *свободных* (т.е. не связанных другими λ'ми) вхождений x в u.

λ-оператор

- Знаком λ выделяется та переменная, которую мы будем считать аргументом функции.
- В теоретическом материале мы будем использовать обозначение $\lambda x.u$, где u — выражение (*терм*), возможно содержащее x:

$$\lambda x.\underbrace{\boxed{\ldots x \ldots x \ldots x \ldots}}_{u}$$

- При вычислении значения функции $\lambda x.u$ на аргументе $x = a$ нужно подставить a вместо x вместо всех *свободных* (т.е. не связанных другими λ'ми) вхождений x в u.
- Это называется *β-преобразованием,* о нём мы поговорим позже.

λ-оператор

- Знаком λ выделяется та переменная, которую мы будем считать аргументом функции.
- В теоретическом материале мы будем использовать обозначение $\lambda x.u$, где u — выражение (*терм*), возможно содержащее x:

$$\lambda x.\underbrace{\boxed{\ldots x \ldots x \ldots x \ldots}}_{u}$$

- При вычислении значения функции $\lambda x.u$ на аргументе $x = a$ нужно подставить a вместо x вместо всех *свободных* (т.е. не связанных другими λ'ми) вхождений x в u.
- Это называется *β-преобразованием,* о нём мы поговорим позже.
- В математике вместо $\lambda x.u$ пишут $x \mapsto u$.

Функции и изменяемость

- Посмотрим на следующий код:

```
x=5
f=lambda y : y+x
print f(2)
x=7
print f(2)
```

Функции и изменяемость

- Посмотрим на следующий код:

```
x=5
f=lambda y : y+x
print f(2)
x=7
print f(2)
```

- Значение `f(2)` изменилось: действительно, `lambda` создаёт новую безымянную функцию, которая, помимо своего аргумента y имеет также неявный доступ к переменной x.

Функции и изменяемость

- Таким образом, в Python'е функции, введённые с помощью `lambda`, ведут себя всё же не совсем как обычные объекты.

Функции и изменяемость

- Таким образом, в Python'е функции, введённые с помощью `lambda`, ведут себя всё же не совсем как обычные объекты.
- Действительно, если бы вместо f была бы числовая переменная:

```
x=5
f=x+2
print f
x=7
print f
```

то значение бы не поменялось.

Функции и изменяемость

- Во многих функциональных языках (в частности, в Haskell'е) проблема с изменением значений решена радикально.

Функции и изменяемость

- Во многих функциональных языках (в частности, в Haskell'е) проблема с изменением значений решена радикально.
- В этом состоит **второе свойство** — immutability: значения переменных вообще запрещено изменять!

Функции и изменяемость

- Во многих функциональных языках (в частности, в Haskell'e) проблема с изменением значений решена радикально.
- В этом состоит **второе свойство** — immutability: значения переменных вообще запрещено изменять!
- Это свойство выглядит довольно дико, принуждая к созданию большого числа объектов вместо изменений одного. Однако этот негативный эффект компенсируется сборкой мусора и оптимизацией.

Неизменяемость

- За счёт неизменяемости функции получаются *чистыми* в математическом смысле: возвращаемое значение однозначно определяется значениями аргументов, и при этом функция не имеет *побочных эффектов.*

Неизменяемость

- За счёт неизменяемости функции получаются *чистыми* в математическом смысле: возвращаемое значение однозначно определяется значениями аргументов, и при этом функция не имеет *побочных эффектов.*
- В императивных языках, наоборот, функции взаимодействуют с неким *состоянием внешнего мира.*

Неизменяемость

- За счёт неизменяемости функции получаются *чистыми* в математическом смысле: возвращаемое значение однозначно определяется значениями аргументов, и при этом функция не имеет *побочных эффектов.*
- В императивных языках, наоборот, функции взаимодействуют с неким *состоянием внешнего мира.*
- Это делает осмысленными, в частности, функции, которые ничего не принимают и не возвращают: `void func();`

Неизменяемость

- За счёт неизменяемости функции получаются *чистыми* в математическом смысле: возвращаемое значение однозначно определяется значениями аргументов, и при этом функция не имеет *побочных эффектов.*
- В императивных языках, наоборот, функции взаимодействуют с неким *состоянием внешнего мира.*
- Это делает осмысленными, в частности, функции, которые ничего не принимают и не возвращают: `void func();`
- Конечно, связь с внешним миром нужна, но в «чистых» функциональных языках она прописывается явно.

Неизменяемость

- За счёт неизменяемости функции получаются *чистыми* в математическом смысле: возвращаемое значение однозначно определяется значениями аргументов, и при этом функция не имеет *побочных эффектов.*
- В императивных языках, наоборот, функции взаимодействуют с неким *состоянием внешнего мира.*
- Это делает осмысленными, в частности, функции, которые ничего не принимают и не возвращают: `void func();`
- Конечно, связь с внешним миром нужна, но в «чистых» функциональных языках она прописывается явно.
- В Haskell'е для этого (в частности — для ввода-вывода) используется механизм *монад,* основанный на теоретико-категорной конструкции.

Вычисление как преобразование

- Наконец, ещё более фундаментальное **третье свойство,** отличающее функциональные языки от императивных, заключается в самом понятии вычислительного процесса.

Вычисление как преобразование

- Наконец, ещё более фундаментальное **третье свойство,** отличающее функциональные языки от императивных, заключается в самом понятии вычислительного процесса.
- В функциональной парадигме вычисление есть последовательное *преобразование* некоего выражения (*терма*), пока он не дойдёт до некоторой далее не преобразуемой формы. (Например, терм, в явном виде представляющий натуральное число.)

Вычисление как преобразование

- Наконец, ещё более фундаментальное **третье свойство,** отличающее функциональные языки от императивных, заключается в самом понятии вычислительного процесса.
- В функциональной парадигме вычисление есть последовательное *преобразование* некоего выражения (*терма*), пока он не дойдёт до некоторой далее не преобразуемой формы. (Например, терм, в явном виде представляющий натуральное число.)
- Преобразования призваны «упрощать» терм, и поэтому также называются *редукциями.*

Вычисление как преобразование

- Наконец, ещё более фундаментальное **третье свойство,** отличающее функциональные языки от императивных, заключается в самом понятии вычислительного процесса.
- В функциональной парадигме вычисление есть последовательное *преобразование* некоего выражения (*терма*), пока он не дойдёт до некоторой далее не преобразуемой формы. (Например, терм, в явном виде представляющий натуральное число.)
- Преобразования призваны «упрощать» терм, и поэтому также называются *редукциями.*
- В реальности редукции не всегда упрощают терм, и возможны бесконечные их последовательности (что соответствует неостанавливающейся программе на императивном языке).

Вычисление как преобразование

- В этом смысле исполнение функциональной программы напоминает вычисление арифметического выражения:

$$(1 + 2) \cdot (3 + 4) \to 3 \cdot (3 + 4) \to 3 \cdot 7 \to 21.$$

Вычисление как преобразование

- В этом смысле исполнение функциональной программы напоминает вычисление арифметического выражения:

$$(1+2)\cdot(3+4) \to 3\cdot(3+4) \to 3\cdot 7 \to 21.$$

- При этом, в отличие от императивной программы, порядок преобразований не задан жёстко:

$$(1+2)\cdot(3+4) \to (1+2)\cdot 7 \to 3\cdot 7 \to 21.$$

Вычисление как преобразование

- В этом смысле исполнение функциональной программы напоминает вычисление арифметического выражения:

$$(1+2)\cdot(3+4) \to 3\cdot(3+4) \to 3\cdot 7 \to 21.$$

- При этом, в отличие от императивной программы, порядок преобразований не задан жёстко:

$$(1+2)\cdot(3+4) \to (1+2)\cdot 7 \to 3\cdot 7 \to 21.$$

- Преобразование можно применить к любому подвыражению (подтерму), которое может быть упрощено. Такой подтерм называется *редексом.*

Конфлюэнтность

- Если синтаксис разработан неправильно, то разные последовательности редукций могут давать разные ответы. Например, так получится, если не использовать скобки:

$$
\begin{tikzcd}
1 + 2 \cdot 3 + 4 \arrow[r] \arrow[dd] & 3 \cdot 3 + 4 \arrow[r] \arrow[d] & 3 \cdot 7 \arrow[r] & 21 \\
 & 9 + 4 \arrow[r] & 13 & \\
1 + 6 + 4 \arrow[r] & 7 + 4 \arrow[r] & 11 &
\end{tikzcd}
$$

Конфлюэнтность

- Если синтаксис разработан неправильно, то разные последовательности редукций могут давать разные ответы. Например, так получится, если не использовать скобки:

$$
\begin{tikzcd}
1 + 2 \cdot 3 + 4 \arrow[r] \arrow[dd] & 3 \cdot 3 + 4 \arrow[r] \arrow[d] & 3 \cdot 7 \arrow[r] & 21 \\
 & 9 + 4 \arrow[r] & 13 & \\
1 + 6 + 4 \arrow[r] & 7 + 4 \arrow[r] & 11 &
\end{tikzcd}
$$

- В «хороших» системах этого не происходит за счёт *конфлюэнтности (свойства Чёрча – Россера):* если $u \twoheadrightarrow v_1$ и $u \twoheadrightarrow v_2$, то существует такой терм w, что $v_1 \twoheadrightarrow w$ и $v_2 \twoheadrightarrow w$.

Ленивость

- За счёт порядка преобразований какие-то подтермы могут оказаться вообще не вычисленными.

Ленивость

- За счёт порядка преобразований какие-то подтермы могут оказаться вообще не вычисленными.
- Например, $\mathrm{length}[u, v, w]$ можно сразу редуцировать к 3, не пытаясь вычислить значения u, v, w.

Ленивость

- За счёт порядка преобразований какие-то подтермы могут оказаться вообще не вычисленными.
- Например, $\mathrm{length}[u, v, w]$ можно сразу редуцировать к 3, не пытаясь вычислить значения u, v, w.
- Это свойство называется *ленивостью* вычислений.

- λ-исчисление — простейшая модель и основа функциональных языков программирования.

λ-исчисление

- λ-исчисление — простейшая модель и основа функциональных языков программирования.
- Термы λ-исчисления (*λ-термы*) строятся из переменных с помощью всего лишь двух операций:
    - *применение:* если u и v — термы, то (uv) — терм;
    - *λ-абстракция:* если u — терм, x — переменная, то $\lambda x.u$ — терм.

- λ-исчисление — простейшая модель и основа функциональных языков программирования.
- Термы λ-исчисления (*λ-термы*) строятся из переменных с помощью всего лишь двух операций:
    - *применение:* если u и v — термы, то (uv) — терм;
    - *λ-абстракция:* если u — терм, x — переменная, то $\lambda x.u$ — терм.
- Запись (uv) означает применение функции u к v.

- λ-исчисление — простейшая модель и основа функциональных языков программирования.
- Термы λ-исчисления (*λ-термы*) строятся из переменных с помощью всего лишь двух операций:
    - *применение:* если u и v — термы, то (uv) — терм;
    - *λ-абстракция:* если u — терм, x — переменная, то $\lambda x.u$ — терм.
- Запись (uv) означает применение функции u к v.
- Более привычное обозначение было бы $u(v)$, однако бесскобочное обозначение также применяется в математике — например, $\sin\alpha$.

λ-исчисление

- λ-исчисление — простейшая модель и основа функциональных языков программирования.
- Термы λ-исчисления (*λ-термы*) строятся из переменных с помощью всего лишь двух операций:
    - *применение:* если u и v — термы, то (uv) — терм;
    - *λ-абстракция:* если u — терм, x — переменная, то $\lambda x.u$ — терм.
- Запись (uv) означает применение функции u к v.
- Более привычное обозначение было бы $u(v)$, однако бесскобочное обозначение также применяется в математике — например, $\sin\alpha$.
- В функциональных языках чаще используется бесскобочная запись.

Функции многих аргументов

- С помощью λ-абстракции можно задать функцию *одного* аргумента x. Как быть с функциями многих аргументов?

Функции многих аргументов

- С помощью λ-абстракции можно задать функцию *одного* аргумента x. Как быть с функциями многих аргументов?
- Для этого используется приём, называемый *каррированием* (в честь Х. Карри): $f = \lambda x.\lambda y.\lambda z.u$.

Функции многих аргументов

- С помощью λ-абстракции можно задать функцию *одного* аргумента x. Как быть с функциями многих аргументов?
- Для этого используется приём, называемый *каррированием* (в честь Х. Карри): $f = \lambda x.\lambda y.\lambda z.u$.
- В каррированном виде функция f является функцией одного аргумента (x), возвращающая, в свою очередь, опять же функцию одного аргумента (y) и т.д.

Функции многих аргументов

- С помощью λ-абстракции можно задать функцию *одного* аргумента x. Как быть с функциями многих аргументов?
- Для этого используется приём, называемый *каррированием* (в честь Х. Карри): $f = \lambda x.\lambda y.\lambda z.u$.
- В каррированном виде функция f является функцией одного аргумента (x), возвращающая, в свою очередь, опять же функцию одного аргумента (y) и т.д.
- Для каррированных функций многих аргументов используется сокращённое обозначение $\lambda xyz.u$.

Примеры и бестиповость

- Простейший пример λ-терма: $\mathbf{I} = \lambda x.x$. Этот терм реализует тождественную функцию:

```
def identity(x):
  return x
```

Примеры и бестиповость

- Простейший пример λ-терма: $\mathbf{I} = \lambda x.x$. Этот терм реализует тождественную функцию:

```
def identity(x):
  return x
```

- Отметим, что наше λ-исчисление (как и Python) *бестиповое:* любой терм можно применить, как функцию, к любому другому.

Примеры и бестиповость

- Простейший пример λ-терма: $\mathbf{I} = \lambda x.x$. Этот терм реализует тождественную функцию:

```
def identity(x):
  return x
```

- Отметим, что наше λ-исчисление (как и Python) *бестиповое:* любой терм можно применить, как функцию, к любому другому.
- Более содержательный пример — абстрактная программа для композиции функций

$$\mathbf{B} = \lambda fgx.f(gx).$$

Преобразования λ-термов

- Главное преобразование термов в λ-исчислении — *β-редукция:*

$$(\lambda x.u)v \to_\beta u[x := v].$$

Преобразования λ-термов

- Главное преобразование термов в λ-исчислении — *β-редукция:*

$$(\lambda x.u)v \to_\beta u[x := v].$$

  - Запись $u[x := v]$ означает подстановку v вместо каждого свободного вхождения x в u.

Преобразования λ-термов

- Главное преобразование термов в λ-исчислении — *β-редукция:*

$$(\lambda x.u)v \to_\beta u[x := v].$$

  - Запись $u[x := v]$ означает подстановку v вместо каждого свободного вхождения x в u.
  - Условие корректности подстановки: переменные, свободные в v, не должны оказаться связанными в u. (Например, $(\lambda x.\lambda y.x)y \not\to_\beta \lambda y.y$.)

Преобразования λ-термов

- Главное преобразование термов в λ-исчислении — *β-редукция:*

$$(\lambda x.u)v \to_\beta u[x := v].$$

  - Запись $u[x := v]$ означает подстановку v вместо каждого свободного вхождения x в u.
  - Условие корректности подстановки: переменные, свободные в v, не должны оказаться связанными в u. (Например, $(\lambda x.\lambda y.x)y \not\to_\beta \lambda y.y$.)

- β-редукция может применяться к произвольному редексу вида $(\lambda x.u)v$:

$$\boxed{\ldots \quad (\lambda x.u)v \quad \ldots} \to_\beta \boxed{\ldots \quad u[x := v] \quad \ldots}$$

Преобразования λ-термов

- Помимо β-редукции имеется вспомогательное преобразование — *α-конверсия:*

$$\lambda x.u \to_\alpha \lambda y.u[x := y],$$

  где y — новая переменная.

Преобразования λ-термов

- Помимо β-редукции имеется вспомогательное преобразование — *α-конверсия:*

$$\lambda x.u \to_\alpha \lambda y.u[x := y],$$

  где y — новая переменная.
- Термы, которые можно свести к одному и тому же α-конверсиями, называются *α-равными* и в дальнейшем считаются вариантами одного терма.

Преобразования λ-термов

- Помимо β-редукции имеется вспомогательное преобразование — *α-конверсия:*

$$\lambda x.u \to_\alpha \lambda y.u[x := y],$$

где y — новая переменная.

- Термы, которые можно свести к одному и тому же α-конверсиями, называются *α-равными* и в дальнейшем считаются вариантами одного терма.
- α-конверсия помогает решить проблему с недопустимой подстановкой при β-редукции:

$$(\lambda x.\lambda y.x)y =_\alpha (\lambda x.\lambda z.x)y \to_\beta \lambda z.y$$

Нормализация

- *Нормальная форма* — это терм, в котором нет β-редексов (т.е. который далее нельзя редуцировать).

Нормализация

- *Нормальная форма* — это терм, в котором нет β-редексов (т.е. который далее нельзя редуцировать).
- **Теорема.** β-редукция обладает свойством Чёрча – Россера:

$$\begin{tikzcd}
 & v_1 \arrow[dr, dashed, ->>] & \\
u \arrow[ur, ->>] \arrow[dr, ->>] & & w \\
 & v_2 \arrow[ur, dashed, ->>] &
\end{tikzcd}$$

Нормализация

- *Нормальная форма* — это терм, в котором нет β-редексов (т.е. который далее нельзя редуцировать).
- **Теорема.** β-редукция обладает свойством Чёрча – Россера:

$$
\begin{tikzcd}
& v_1 \arrow[dr, dashed, two heads] & \\
u \arrow[ur, two heads] \arrow[dr, two heads] & & w \\
& v_2 \arrow[ur, dashed, two heads] &
\end{tikzcd}
$$

  - Доказательство этой, как и некоторых других, теорем, будет опубликовано в виде конспекта.

Нормализация

- *Нормальная форма* — это терм, в котором нет β-редексов (т.е. который далее нельзя редуцировать).
- **Теорема.** β-редукция обладает свойством Чёрча – Россера:

$$\begin{tikzcd} & v_1 \arrow[dr, dashed, two heads] & \\ u \arrow[ur, two heads] \arrow[dr, two heads] & & w \\ & v_2 \arrow[ur, dashed, two heads] & \end{tikzcd}$$

  - Доказательство этой, как и некоторых других, теорем, будет опубликовано в виде конспекта.
- **Следствие.** Данный терм не может редуцироваться к двум α-разным нормальным формам.

Нормализация

- Поскольку нормальная форма не зависит от пути, которым мы к ней пришли, её можно считать *результатом вычисления значения* данного λ-терма.

Нормализация

- Поскольку нормальная форма не зависит от пути, которым мы к ней пришли, её можно считать *результатом вычисления значения* данного λ-терма.
- Однако всё не так просто.

Нормализация

- Поскольку нормальная форма не зависит от пути, которым мы к ней пришли, её можно считать *результатом вычисления значения* данного λ-терма.
- Однако всё не так просто.
- Бывают термы, которые вообще не приводятся к нормальной форме (любое вычисление бесконечно).

Нормализация

- Поскольку нормальная форма не зависит от пути, которым мы к ней пришли, её можно считать *результатом вычисления значения* данного λ-терма.
- Однако всё не так просто.
- Бывают термы, которые вообще не приводятся к нормальной форме (любое вычисление бесконечно).
- Бывают и такие, для которых один путь приводит к нормальной форме (*слабая нормализуемость*), а другой бесконечен.

Нормализация

- Поскольку нормальная форма не зависит от пути, которым мы к ней пришли, её можно считать *результатом вычисления значения* данного λ-терма.
- Однако всё не так просто.
- Бывают термы, которые вообще не приводятся к нормальной форме (любое вычисление бесконечно).
- Бывают и такие, для которых один путь приводит к нормальной форме (*слабая нормализуемость*), а другой бесконечен.
- Наконец, если все пути приводят к нормальной форме, то такой терм *сильно нормализуем.*

Примеры

- Пусть $\omega = \lambda x.(xx)$, а $\Omega = \omega\omega$. Тогда Ω редуцируется только сам к себе:

$$\Omega = (\lambda x.(xx))(\lambda x.(xx)) \to_\beta (xx)[x := \omega] = \omega\omega = \Omega,$$

  значит, он не нормализуем.
- Можно построить и терм, который будет при «редукции» бесконечно разрастаться.
- Бывает и слабо нормализуемый терм, не являющийся сильно нормализуемым: например, $(\lambda x.y)\Omega$.

Нормализуемость

- Из-за существования слабо, но не сильно нормализуемых термов важен порядок, или *стратегия*, применения редукций.

Нормализуемость

- Из-за существования слабо, но не сильно нормализуемых термов важен порядок, или *стратегия*, применения редукций.
- О различных стратегиях редукций мы поговорим на следующей лекции.

Нормализуемость

- Из-за существования слабо, но не сильно нормализуемых термов важен порядок, или *стратегия*, применения редукций.
- О различных стратегиях редукций мы поговорим на следующей лекции.
- А пока что коротко обсудим вычислительные возможности бестипового λ-исчисления.

Натуральные числа по Чёрчу

- Натуральное число n можно представить следующим образом с помощью константы o (ноль) и функции s (взятие следующего):

$$\underbrace{s(s \ldots (s}_{n \text{ раз}}\, o) \ldots)$$

Натуральные числа по Чёрчу

- Натуральное число n можно представить следующим образом с помощью константы o (ноль) и функции s (взятие следующего):

$$\underbrace{s(s \ldots (s}_{n \text{ раз}}\, o) \ldots)$$

- В «чистом» λ-исчислении у нас нет констант, поэтому мы просто абстрагируем s и o как переменные, получив замкнутый (без свободных переменных) терм, называемый *нумералом Чёрча:*

$$\underline{n} = \lambda so.\, \underbrace{s(s \ldots (s}_{n \text{ раз}}\, o) \ldots)$$

Представление функций

- Заметим, что нумералы Чёрча не содержат β-редексов, т.е. являются нормальными формами.

Представление функций

- Заметим, что нумералы Чёрча не содержат β-редексов, т.е. являются нормальными формами.
- Таким образом, можно считать, что некий λ-терм F является программой, вычисляющей k-местную функцию f на натуральных числах, если

$$F\,\underline{n_1}\ldots\underline{n_k} \twoheadrightarrow_\beta \underline{f(n_1,\ldots,n_k)}$$

Представление функций

- Заметим, что нумералы Чёрча не содержат β-редексов, т.е. являются нормальными формами.
- Таким образом, можно считать, что некий λ-терм F является программой, вычисляющей k-местную функцию f на натуральных числах, если

$$F\,\underline{n_1}\ldots\underline{n_k} \twoheadrightarrow_\beta \underline{f(n_1,\ldots,n_k)}$$

- Здесь, в соответствии с функциональной парадигмой, процесс *вычисления* значения функции f представляется в виде *редукции* терма $F\,\underline{n_1}\ldots\underline{n_k}$.

Представление функций

- Заметим, что нумералы Чёрча не содержат β-редексов, т.е. являются нормальными формами.
- Таким образом, можно считать, что некий λ-терм F является программой, вычисляющей k-местную функцию f на натуральных числах, если

$$F\,\underline{n_1}\ldots\underline{n_k} \twoheadrightarrow_\beta \underline{f(n_1,\ldots,n_k)}$$

- Здесь, в соответствии с функциональной парадигмой, процесс *вычисления* значения функции f представляется в виде *редукции* терма $F\,\underline{n_1}\ldots\underline{n_k}$.
- В силу конфлюэнтности, результат вычисления определяется однозначно.

Представление функций

- Однако возможна ситуация слабой нормализуемости, при которой мы можем пойти по «неправильному» пути и не достичь нормальной формы (которая при этом существует).

- Однако возможна ситуация слабой нормализуемости, при которой мы можем пойти по «неправильному» пути и не достичь нормальной формы (которая при этом существует).
- Бороться с этим нужно выбором правильной *стратегии нормализации*, о чём мы поговорим на следующей лекции.

Представление функций

- Однако возможна ситуация слабой нормализуемости, при которой мы можем пойти по «неправильному» пути и не достичь нормальной формы (которая при этом существует).
- Бороться с этим нужно выбором правильной *стратегии нормализации*, о чём мы поговорим на следующей лекции.
- Короткий ответ: если нормальная форма существует, то её можно достичь, всегда редуцируя *самый левый* (считая по начальной λ'е) β-редекс.

Представление функций

- На нумералах Чёрча легко определить операции сложения и умножения:

$$\underline{n} + \underline{m} = \lambda so.(\underline{n}s)(\underline{m}so);$$
$$\underline{n} \cdot \underline{m} = \lambda so.\underline{m}(\underline{n}s)o.$$

- Абстрагируя, получаем термы для (двуместных) функций сложения и умножения:

$$\mathbf{+} = \lambda xyso.(xs)(yso);$$
$$\cdot = \lambda xyso.x(ys)o.$$

- **Задача.** Задайте λ-термом функцию «предшественник»:

$$\mathrm{Prev}(n) = \begin{cases} 0, & \text{если } n = 0; \\ n - 1, & \text{если } n > 0. \end{cases}$$

Представление функций

- На самом деле, λ-термы умеют намного больше, чем сложение и умножение: с их помощью можно записать **любую алгоритмически вычислимую** функцию на натуральных числах.

Представление функций

- На самом деле, λ-термы умеют намного больше, чем сложение и умножение: с их помощью можно записать **любую алгоритмически вычислимую** функцию на натуральных числах.
- При этом функция может быть не всюду определённой — тогда на соответствующих значениях аргументов терм $F\,\underline{n_1}\ldots\underline{n_k}$ будет ненормализуемым.

Представление функций

- На самом деле, λ-термы умеют намного больше, чем сложение и умножение: с их помощью можно записать **любую алгоритмически вычислимую** функцию на натуральных числах.
- При этом функция может быть не всюду определённой — тогда на соответствующих значениях аргументов терм $F\,\underline{n_1}\ldots\underline{n_k}$ будет ненормализуемым.
- Мы обсуждаем такой «низкоуровневый» язык, как λ-исчисление, чтобы не перегружать изложение синтаксическими деталями.

Представление функций

- На самом деле, λ-термы умеют намного больше, чем сложение и умножение: с их помощью можно записать **любую алгоритмически вычислимую** функцию на натуральных числах.
- При этом функция может быть не всюду определённой — тогда на соответствующих значениях аргументов терм $F\,\underline{n_1}\ldots\underline{n_k}$ будет ненормализуемым.
- Мы обсуждаем такой «низкоуровневый» язык, как λ-исчисление, чтобы не перегружать изложение синтаксическими деталями.
- Можно сказать, что всё остальное в функциональных языках — надстройка для удобства, «синтаксический сахар».

Булевы операции

- В λ-исчислении можно ввести константы «истина» и «ложь» как функции выбора из двух аргументов:

$$\mathbf{T} = \lambda t.\lambda f.t; \qquad \mathbf{F} = \lambda t.\lambda f.f.$$

Булевы операции

- В λ-исчислении можно ввести константы «истина» и «ложь» как функции выбора из двух аргументов:

$$\mathbf{T} = \lambda t.\lambda f.t; \qquad \mathbf{F} = \lambda t.\lambda f.f.$$

- Условный оператор: $\left(\mathbf{if}\ b\ \mathbf{then}\ u\ \mathbf{else}\ v\right) = b\,u\,v.$

Булевы операции

- В λ-исчислении можно ввести константы «истина» и «ложь» как функции выбора из двух аргументов:

$$\mathbf{T} = \lambda t.\lambda f.t; \qquad \mathbf{F} = \lambda t.\lambda f.f.$$

- Условный оператор: $\big(\mathbf{if}\ b\ \mathbf{then}\ u\ \mathbf{else}\ v\big) = b\,u\,v$.
- Логические операции:

$$(b_1\ \mathbf{and}\ b_2) = \big(\mathbf{if}\ b_1\ \mathbf{then}\ \big(\mathbf{if}\ b_2\ \mathbf{then}\ \mathbf{T}\ \mathbf{else}\ \mathbf{F}\big)\ \mathbf{else}\ \mathbf{F}\big)$$

$$\ldots$$

Булевы операции

- В λ-исчислении можно ввести константы «истина» и «ложь» как функции выбора из двух аргументов:

$$\mathbf{T} = \lambda t.\lambda f.t; \qquad \mathbf{F} = \lambda t.\lambda f.f.$$

- Условный оператор: $\big(\textbf{if } b \textbf{ then } u \textbf{ else } v\big) = b\,u\,v$.
- Логические операции:

$$(b_1 \textbf{ and } b_2) = \big(\textbf{if } b_1 \textbf{ then } \big(\textbf{if } b_2 \textbf{ then T else F}\big) \textbf{ else F}\big)$$
$$\ldots$$

- Проверка на ноль: $\mathbf{Zero} = \lambda x.(x\,(\lambda z.\mathbf{F})\,\mathbf{T})$.

Рекурсия

- Чтобы достичь полноты по Тьюрингу, осталось реализовать **рекурсию** (которая в функциональных языках используется повсеместно, в т.ч. вместо циклов).

Рекурсия

- Чтобы достичь полноты по Тьюрингу, осталось реализовать **рекурсию** (которая в функциональных языках используется повсеместно, в т.ч. вместо циклов).
- Пример: факториал $f(n) = n! = 1 \cdot 2 \cdot \ldots \cdot n$.

Рекурсия

- Чтобы достичь полноты по Тьюрингу, осталось реализовать **рекурсию** (которая в функциональных языках используется повсеместно, в т.ч. вместо циклов).
- Пример: факториал $f(n) = n! = 1 \cdot 2 \cdot \ldots \cdot n$.
- Рекурсивная реализация:

$$\mathrm{Fact} = \lambda x.\big(\textbf{if Zero } x \textbf{ then } \underline{1} \textbf{ else } (\mathrm{Fact}\,(\textbf{Prev } x) \cdot x)\big)$$

Рекурсия

- Чтобы достичь полноты по Тьюрингу, осталось реализовать **рекурсию** (которая в функциональных языках используется повсеместно, в т.ч. вместо циклов).
- Пример: факториал $f(n) = n! = 1 \cdot 2 \cdot \ldots \cdot n$.
- Рекурсивная реализация:

$$\mathrm{Fact} = \lambda x.\big(\mathbf{if}\ \mathbf{Zero}\ x\ \mathbf{then}\ \underline{1}\ \mathbf{else}\ (\mathrm{Fact}\,(\mathbf{Prev}\ x) \cdot x)\big)$$

- Проблема: Fact определяется через самоё себя.

Рекурсия

- Чтобы достичь полноты по Тьюрингу, осталось реализовать **рекурсию** (которая в функциональных языках используется повсеместно, в т.ч. вместо циклов).
- Пример: факториал $f(n) = n! = 1 \cdot 2 \cdot \ldots \cdot n$.
- Рекурсивная реализация:

$$\mathrm{Fact} = \lambda x.\big(\textbf{if Zero } x \textbf{ then } \underline{1} \textbf{ else } (\mathrm{Fact}\,(\textbf{Prev } x) \cdot x)\big)$$

- Проблема: Fact определяется через самоё себя.
- С помощью λ-абстракции можно сделать зависимость в правой части явной (функциональной):

$$\mathrm{Fact} = \underbrace{\big(\lambda g.\lambda x.\big(\textbf{if Zero } x \textbf{ then } \underline{1} \textbf{ else } (g\,(\textbf{Prev } x) \cdot x)\big)\big)}_{F}\ \mathrm{Fact}$$

Рекурсия

- Чтобы реализовать рекурсивно определённую функцию, используется *комбинатор неподвижной точки* ($\mathbf{Y}$-комбинатор, или комбинатор Карри) со следующим свойством: $\mathbf{Y}\,F =_\beta F(\mathbf{Y}\,F)$.

Рекурсия

- Чтобы реализовать рекурсивно определённую функцию, используется *комбинатор неподвижной точки* (**Y**-комбинатор, или комбинатор Карри) со следующим свойством: $\mathbf{Y}\,F =_\beta F(\mathbf{Y}\,F)$.
- $\mathbf{Y} = \lambda f.\big((\lambda x.f(xx))(\lambda x.f(xx))\big)$

Рекурсия

- Чтобы реализовать рекурсивно определённую функцию, используется *комбинатор неподвижной точки* (**Y**-комбинатор, или комбинатор Карри) со следующим свойством: $\mathbf{Y}\,F =_\beta F(\mathbf{Y}\,F)$.
- $\mathbf{Y} = \lambda f.\big((\lambda x.f(xx))(\lambda x.f(xx))\big)$
- Имеем $\mathbf{Y}\,F \to_\beta Y_F = (\lambda x.F(xx))(\lambda x.F(xx))$, при этом Y_F — неподвижная точка для F: $Y_F \to_\beta F(Y_F)$.

Рекурсия

- Чтобы реализовать рекурсивно определённую функцию, используется *комбинатор неподвижной точки* ($\mathbf{Y}$-комбинатор, или комбинатор Карри) со следующим свойством: $\mathbf{Y}\,F =_\beta F(\mathbf{Y}\,F)$.
- $\mathbf{Y} = \lambda f.\big((\lambda x.f(xx))(\lambda x.f(xx))\big)$
- Имеем $\mathbf{Y}\,F \to_\beta Y_F = (\lambda x.F(xx))(\lambda x.F(xx))$, при этом Y_F — неподвижная точка для F: $Y_F \to_\beta F(Y_F)$.
- Терм, использующий $\mathbf{Y}$-комбинатор, никогда не будет сильно нормализуемым: $Y_F \to_\beta F(Y_F) \to_\beta F(F(Y_F)) \to_\beta \ldots$

Рекурсия

- Чтобы реализовать рекурсивно определённую функцию, используется *комбинатор неподвижной точки* ($\mathbf{Y}$-комбинатор, или комбинатор Карри) со следующим свойством: $\mathbf{Y}\,F =_\beta F(\mathbf{Y}\,F)$.
- $\mathbf{Y} = \lambda f.\big((\lambda x.f(xx))(\lambda x.f(xx))\big)$
- Имеем $\mathbf{Y}\,F \to_\beta Y_F = (\lambda x.F(xx))(\lambda x.F(xx))$, при этом Y_F — неподвижная точка для F: $Y_F \to_\beta F(Y_F)$.
- Терм, использующий $\mathbf{Y}$-комбинатор, никогда не будет сильно нормализуемым: $Y_F \to_\beta F(Y_F) \to_\beta F(F(Y_F)) \to_\beta \dots$
- Однако если разбирать F, то процесс может сойтись: например, $\mathrm{Fact}\,\underline{n} \twoheadrightarrow_\beta \underline{n!}$

Рекурсия

$$\mathrm{Fact}\ \underline{0} \to_\beta Y_F\,\underline{0} \to_\beta F\,Y_F\,\underline{0} \twoheadrightarrow_\beta \left(\mathbf{if\ Zero}\ \underline{0}\ \mathbf{then}\ \underline{1}\ \mathbf{else}\ (Y_F\,(\mathbf{Prev}\ \underline{0}))\cdot\underline{0}\right) \twoheadrightarrow_\beta \underline{1}$$

Рекурсия

$$\mathrm{Fact}\,\underline{0} \to_\beta Y_F\,\underline{0} \to_\beta F\,Y_F\,\underline{0} \twoheadrightarrow_\beta \left(\mathbf{if\ Zero}\,\underline{0}\ \mathbf{then}\ \underline{1}\ \mathbf{else}\ (Y_F\,(\mathbf{Prev}\ \underline{0}))\cdot\underline{0}\right) \twoheadrightarrow_\beta \underline{1}$$

$$\begin{aligned}\mathrm{Fact}\,\underline{n+1} \to_\beta Y_F\,\underline{n+1} \to_\beta F\,Y_F\,\underline{n+1} \twoheadrightarrow_\beta& \\ \twoheadrightarrow_\beta \left(\mathbf{if\ Zero}\,\underline{n+1}\ \mathbf{then}\ \underline{1}\ \mathbf{else}\ (Y_F\,(\mathbf{Prev}\ \underline{n+1}))\cdot\underline{n+1}\right) \twoheadrightarrow_\beta& \\ \twoheadrightarrow_\beta (Y_F\,\underline{n})\cdot\underline{n+1}&\end{aligned}$$

Рекурсия

- Если рекурсивное определение «плохое» (например, забыто **Prev**), то терм будет ненормализуемым: **любая** последовательность редукций бесконечна.

Рекурсия

- Если рекурсивное определение «плохое» (например, забыто **Prev**), то терм будет ненормализуемым: **любая** последовательность редукций бесконечна.
- Статически проверить это невозможно, поскольку задача останова алгоритмически неразрешима.

- Если рекурсивное определение «плохое» (например, забыто **Prev**), то терм будет ненормализуемым: **любая** последовательность редукций бесконечна.
- Статически проверить это невозможно, поскольку задача останова алгоритмически неразрешима.
- **Y** — не единственный комбинатор неподвижной точки. Таков, например, также *комбинатор Тьюринга* $\mathbf{\Theta} = (\lambda xy.y(xxy))(\lambda xy.y(xxy))$

Рекурсия

- Если рекурсивное определение «плохое» (например, забыто **Prev**), то терм будет ненормализуемым: **любая** последовательность редукций бесконечна.
- Статически проверить это невозможно, поскольку задача останова алгоритмически неразрешима.
- **Y** — не единственный комбинатор неподвижной точки. Таков, например, также *комбинатор Тьюринга* $\mathbf{\Theta} = (\lambda xy.y(xxy))(\lambda xy.y(xxy))$
- … и даже ??????????????????????????, где $? = \lambda abcdefghijklmnopqstuvwxyzr.r(thisisafixedpointcombinator)$

Y-комбинатор в Python'e

- Терм, содержащий **Y**-комбинатор, не будет корректным, если следить за типами данных. Действительно, он содержит xx, значит, переменная x должна одновременно быть некоторого типа A и типа функции $A \to B$.

Y-комбинатор в Python'е

- Терм, содержащий **Y**-комбинатор, не будет корректным, если следить за типами данных. Действительно, он содержит xx, значит, переменная x должна одновременно быть некоторого типа A и типа функции $A \to B$.
- Тем не менее, в бестиповом языке, таком как Python, **Y**-комбинатор можно реализовать.

Y-комбинатор в Python'е

- Терм, содержащий **Y**-комбинатор, не будет корректным, если следить за типами данных. Действительно, он содержит xx, значит, переменная x должна одновременно быть некоторого типа A и типа функции $A \to B$.
- Тем не менее, в бестиповом языке, таком как Python, **Y**-комбинатор можно реализовать.
- Наивная попытка:

```
Y = lambda f : ((lambda x : f(x(x))) (lambda x : f(x(x))))
fact = Y (lambda g : lambda n : (n and n * g(n-1)) or 1)
```

- Терм, содержащий **Y**-комбинатор, не будет корректным, если следить за типами данных. Действительно, он содержит xx, значит, переменная x должна одновременно быть некоторого типа A и типа функции $A \to B$.
- Тем не менее, в бестиповом языке, таком как Python, **Y**-комбинатор можно реализовать.
- Наивная попытка:

```
Y = lambda f : ((lambda x : f(x(x))) (lambda x : f(x(x))))
fact = Y (lambda g : lambda n : (n and n * g(n-1)) or 1)
```

- Не работает: "maximum recursion depth exceeded". Python использует не тот порядок вычислений и уходит в бесконечное вычисление.

Y-комбинатор в Python'е

- Положение можно исправить, заменив xx на $\lambda z.xxz$:

```
Y = lambda f : ((lambda x : f(x(x))) (lambda x : f(lambda z : x(x)(z))))
```

Y-комбинатор в Python'е

- Положение можно исправить, заменив xx на $\lambda z.xxz$:

```
Y = lambda f : ((lambda x : f(x(x))) (lambda x : f(lambda z : x(x)(z))))
```

- Математически h и $\lambda z.hz$ эквивалентны (если h не зависит от z), однако β-редукцией друг к другу не сводятся — это *η-эквивалентность.*

Y-комбинатор в Python'е

- Положение можно исправить, заменив xx на $\lambda z.xxz$:

```
Y = lambda f : ((lambda x : f(x(x))) (lambda x : f(lambda z : x(x)(z))))
```

- Математически h и $\lambda z.hz$ эквивалентны (если h не зависит от z), однако β-редукцией друг к другу не сводятся — это *η-эквивалентность.*
- Вычисление откладывается до тех пор, пока `lambda z : x(x)(z)` окажется к чему-то применено.

- Положение можно исправить, заменив xx на $\lambda z.xxz$:

```
Y = lambda f : ((lambda x : f(x(x))) (lambda x : f(lambda z : x(x)(z))))
```

- Математически h и $\lambda z.hz$ эквивалентны (если h не зависит от z), однако β-редукцией друг к другу не сводятся — это *η-эквивалентность.*
- Вычисление откладывается до тех пор, пока `lambda z : x(x)(z)` окажется к чему-то применено.
- Теперь всё работает: например, `fact(6)` даёт 720.

Y-комбинатор в Python'е

- **Упражнение.** Реализуйте на Python'е комбинатор Тьюринга $\boldsymbol{\Theta} = (\lambda xy.y(xxy))(\lambda xy.y(xxy))$ (с соответствующим η-преобразованием).

- **Упражнение.** Реализуйте на Python'е комбинатор Тьюринга $\mathbf{\Theta} = (\lambda xy.y(xxy))(\lambda xy.y(xxy))$ (с соответствующим η-преобразованием).
- **Упражнение.** Верно ли, что для комбинатора $\widetilde{\mathbf{Y}} = \lambda f.(\lambda x.f(xx))(\lambda x.f(\lambda z.xxz))$ и терма F из определения факториала терм $\widetilde{\mathbf{Y}}F$ сильно нормализуем?