# Функциональное программирование
# Лекция 1. Лямбда-исчисление

Денис Николаевич Москвин

ИТМО, магистратура JB SE

07.09.2020

# План лекции

1. Функциональное vs императивное программирование
2. Неформальное введение в $\lambda$-исчисление
3. Чистое $\lambda$-исчисление: термы
4. Эквивалентность термов

# План лекции

# Императивное программирование

Императивное программирование: вычисление описывается в терминах инструкций, изменяющих состояние вычислителя.
В императивных программах:

- Инструкции исполняются последовательно `C1; C2; C3`;
- Состояние изменяется инструкциями присваивания значений изменяемым переменным: `v := value`;
- Есть механизм условного исполнения: инструкции `if`, `switch`;
- группы инструкций можно повторять с помощью циклов: инструкции `while`, `for`;
- типы данных описываются с оглядкой на их физическое представление в памяти.

Иногда говорят про стиль фон Неймана (термин Джона Бэкуса).

# Императивное программирование: пример

- Факториал числа для фон-неймановского языка

```
long factorial (int n) {
   long res = 1;
   for (int i = n; i > 0; i--)
      res = res * i;
   return res;
}
```

- Выполнение программы: переход вычислителя из начального состояния в конечное с помощью последовательных инструкций.
- Часть конечного состояния (память, доступная по адресу, на который ссылается локальная переменная `res`) интерпретируется как результат вычислений.

# Функциональное программирование

Функциональная программа — *выражение*, её выполнение — вычисление (*редукция*) этого выражения.

```
factorial n = if n == 0 then 1 else n * factorial (n-1)
```

Выполнение программы: редукция выражения с помощью *подстановки* определений функций в местах их «вызова» с заменой формальных параметров на фактические.

```
factorial 3
⟶ if 3 == 0 then 1 else 3 * factorial (3-1)
⟶ ...
⟶ 3 * factorial (3-1)
⟶ 3 * if (3-1) == 0 then 1 else (3-1) * factorial ((3-1)-1)
⟶ ...
⟶ 3 * 2 * 1 * 1 ⟶ 6
```

Реализация может быть эффективнее, но «подстановочная» семантика должна сохраняться.

# Функциональное программирование

Модель вычислений в ФП не использует понятия изменяемого состояния.

- Нет состояний — нет *изменяемых* переменных
- Нет переменных — нет присваивания
- Нет циклов, поскольку нет различий между итерациями
- Последовательность не важна, поскольку выражения независимы

Вместо этого есть:

- Рекурсия — замена циклов
- Функции высших порядков (HOF)
- Сопоставление с образцом

Все функции — чистые, то есть возвращаемое значение зависит только от значений параметров.

# Сильные стороны ФП

- Регулярный и лаконичный синтаксис.
- Мощная типизация, необременительная благодаря эффективным алгоритмам вывода типов.
- Эффективная доказуемость свойств программ алгебраическими методами.
- Возможность генерации программ по набору свойств.
- Высокоуровневые оптимизации на базе эквивалентных преобразований.

# План лекции

# λ-исчисление

- λ-исчисление — формальная система, лежащая в основе функционального программирования.
- Разработано Алонзо Чёрчем в 1930-х для формализации и анализа понятия вычислимости.
- Имеет бестиповую и множество типизированных версий.
- Дает возможность компактно описывать семантику вычислительных процессов.

В λ-исчислении имеются два способа строить выражения:

- *применение (аппликация, application)*;
- *лямбда-абстракция (abstraction)*.

# Неформальное введение в λ-исчисление: применение

Нотация *применения* (аппликации) F к X:

$$F\,X$$

- С точки зрения программиста: F (алгоритм) применяется к X (входные данные).
- Скобки вокруг X не используются. Можно интерпретировать пробел как инфиксный оператор применения. (Иногда используют явный @.)
- Различия между алгоритмами и данными нет. Например, допустимо самоприменение:

$$F\,F$$

# Неформальное введение в λ-исчисление: абстракция

- Пусть $M$ — выражение, возможно содержащее $x$. Тогда *лямбда-абстракция* или просто *абстракция*

$$\lambda x.\, M$$

обозначает функцию

$$x \mapsto M[x],$$

то есть каждому $x$ сопоставляется $M[x]$.

- Лямбда-абстракция — способ задать неименованную (анонимную) функцию.
- В лямбда-выражении часть слева от точки называют *абстрактором*, а справа — *телом*.
- Если $x$ в $M[x]$ отсутствует, то $\lambda x.\, M$ — константная функция со значением $M$.

# Неформальное введение в λ-исчисление: вычисления

- Применение и абстракция работают совместно:

$$\underbrace{(\lambda x.\, 5x + 2)}_{F}\ \underbrace{8}_{X} \;=\; 5 \cdot 8 + 2 \;\; (= 42).$$

- То есть выражение

$$(\lambda x.\, 5x + 2)\ 8$$

это применение функции $y(x) = 5x + 2$ к аргументу $8$, дающее в результате вычисления $5 \cdot 8 + 2$.

- Вычисление определяется как подстановка фактических аргументов вместо формальных.

# Неформальное введение в $\lambda$-исчисление: редукция

- Обобщая, вводят понятие *редукции*

$$(\lambda x.\, M)\, N \;\to\; [x \mapsto N]\, M$$

  где $[x \mapsto N]\, M$ обозначает *подстановку* $N$ вместо $x$ в $M$.
- Аппликацию вида $(\lambda x.\, M)\, N$, в которой левый аппликанд является абстракцией, называют $\beta$-*редексом*.
- Шаг вычисления по приведенному выше правилу называют *сокращением* редекса.
- В *чистом* $\lambda$-исчислении нет ничего кроме применения, абстракции и редукции.

# Функции нескольких переменных

- Если выражение содержит несколько переменных, то абстракция по одной из них вносит асимметрию:

$$\lambda n.\, 2m + 3n$$

  Переменную $n$ называют *связанной*, а $m$ — *свободной*.
- Продолжая абстракцию, получим *замкнутое* выражение

$$\lambda m.\, (\lambda n.\, 2m + 3n)$$

- Теперь вычисление требует двух редукций

$$((\lambda m\,.(\lambda n.\, 2m + 3n))\, 15)\, 4 \rightarrow$$
$$(\lambda n.\, 2 \cdot 15 + 3n)\, 4 \;\rightarrow\; 2 \cdot 15 + 3 \cdot 4$$

- Функции нескольких аргументов, в которые аргументы передают последовательно, называют *каррированными*.

# План лекции

# Термы чистого $\lambda$-исчисления

**Определение**

Множество $\lambda$-***термов*** $\Lambda$ индуктивно строится из переменных $V = \{x, y, z, \ldots\}$ с помощью применения и абстракции:

$$\begin{aligned} x \in V &\Rightarrow x \in \Lambda \\ M, N \in \Lambda &\Rightarrow (M\,N) \in \Lambda \\ M \in \Lambda, x \in V &\Rightarrow (\lambda x.\, M) \in \Lambda \end{aligned}$$

- В абстрактном синтаксисе

$$\Lambda \;::=\; V \mid (\Lambda\,\Lambda) \mid (\lambda V.\, \Lambda)$$

- **Соглашение.** Произвольные термы пишем заглавными буквами, переменные — строчными.

# Термы (примеры)

## Примеры $\lambda$-термов

$$x$$

$$(x\,z)$$

$$(\lambda x.\,(x\,z))$$

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

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

$$((\lambda y.\,((\lambda x.\,(x\,z))\,y))\,w)$$

$$(\lambda z.\,(\lambda w.\,((\lambda y.\,((\lambda x.\,(x\,z))\,y))\,w)))$$

В этом списке каждый следующий терм содержит предыдущие в качестве *подтерма*.

# Термы (соглашения)

Общепринятые следующие соглашения:

- Внешние скобки опускаются.
- Применение ассоциативно *влево*:

$$\mathrm{F\,X\,Y\,Z} \quad \text{обозначает} \quad (((\mathrm{F\,X})\,\mathrm{Y})\,\mathrm{Z})$$

- Абстракция ассоциативна *вправо*:

$$\lambda x\,y\,z.\,\mathrm{M} \quad \text{обозначает} \quad (\lambda x.\,(\lambda y.\,(\lambda z.\,(\mathrm{M}))))$$

- Тело абстракции простирается вправо насколько это возможно:

$$\lambda x.\,\mathrm{M\,N\,K} \quad \text{обозначает} \quad \lambda x.\,(\mathrm{M\,N\,K})$$

# Термы (примеры)

## Те же примеры, что и выше, но с использованием соглашений

$$
\begin{aligned}
x &= x \\
(x\,z) &= x\,z \\
(\lambda x.\,(x\,z)) &= \lambda x.\,x\,z \\
((\lambda x.\,(x\,z))\,y) &= (\lambda x.\,x\,z)\,y \\
(\lambda y.\,((\lambda x.\,(x\,z))\,y)) &= \lambda y.\,(\lambda x\,.\,x\,z)\,y \\
((\lambda y.\,((\lambda x.\,(x\,z))\,y))\,w) &= (\lambda y.\,(\lambda x\,.\,x\,z)\,y)\,w \\
(\lambda z.\,(\lambda w.\,((\lambda y.\,((\lambda x.\,(x\,z))\,y))\,w))) &= \lambda z\,w.\,(\lambda y.\,(\lambda x\,.\,x\,z)\,y)\,w
\end{aligned}
$$

# Свободные и связанные переменные

Абстракция $\lambda x\,.\,M[x]$ *связывает* дотоле *свободную* переменную $x$ в терме $M$.

$$(\lambda y.\,(\lambda w.\,w\,z)\,y)\,x$$

Переменные $y$ и $w$ — связанные, а $z$ и $x$ — свободные.

Связывание переменной ограничивает ее область видимости телом лямбды. Поэтому вне тела лямбды то же самое имя может связываться повторно.

$$(\lambda x.\,(\lambda x.\,x\,z)\,x)\,x$$

Переменная $x$ — связанная (дважды!) и свободная, а $z$ — свободная.

# Свободные и связанные переменные (2)

**Определение**

Множество $\mathrm{FV}(T)$ ***свободных (free) переменных*** в $\lambda$-терме $T$:

$$\begin{aligned} \mathrm{FV}(x) &= \{x\}; \\ \mathrm{FV}(M\,N) &= \mathrm{FV}(M) \cup \mathrm{FV}(N); \\ \mathrm{FV}(\lambda x.\, M) &= \mathrm{FV}(M) \setminus \{x\}. \end{aligned}$$

**Определение**

Множество $\mathrm{BV}(T)$ ***связанных (bound) переменных*** в терме $T$:

$$\begin{aligned} \mathrm{BV}(x) &= \varnothing; \\ \mathrm{BV}(M\,N) &= \mathrm{BV}(M) \cup \mathrm{BV}(N); \\ \mathrm{BV}(\lambda x.\, M) &= \mathrm{BV}(M) \cup \{x\}. \end{aligned}$$

# Комбинаторы

## Определение

$M$ — ***замкнутый*** $\lambda$***-терм*** (или ***комбинатор***), если $FV(M) = \varnothing$. Множество замкнутых $\lambda$-термов обозначается $\Lambda^0$.

Многим комбинаторам даны общепринятые имена:

## Классические комбинаторы

$$
\begin{aligned}
\mathbf{I} &= \lambda x.\, x \\
\boldsymbol{\omega} &= \lambda x.\, x\, x \\
\boldsymbol{\Omega} &= \boldsymbol{\omega}\, \boldsymbol{\omega} = (\lambda x.\, x\, x)(\lambda x.\, x\, x) \\
\mathbf{K} &= \lambda x\, y.\, x \\
\mathbf{K}_* &= \lambda x\, y.\, y \\
\mathbf{C} &= \lambda f\, x\, y.\, f\, y\, x \\
\mathbf{B} &= \lambda f\, g\, x.\, f\, (g\, x) \\
\mathbf{S} &= \lambda f\, g\, x.\, f\, x\, (g\, x)
\end{aligned}
$$

# Переименование связанных переменных

Имена связанных переменных не важны; их можно (почти) безболезненно переименовывать:

$$\begin{aligned} \mathbf{I} &= \lambda x.\, x = \lambda y.\, y = \lambda z.\, z \\ \mathbf{B} &= \lambda f\, g\, x.\, f\,(g\, x) = \lambda u\, v\, z.\, u\,(v\, z) \end{aligned}$$

Это верно в том смысле, что термы с переименованиями при вычислениях дают один и тот же результат:

$$\begin{aligned} (\lambda x.\, x)\, N &\to N \\ (\lambda y.\, y)\, N &\to N \\ (\lambda z.\, z)\, N &\to N \end{aligned}$$

Термы, отличающиеся только именами связанных переменных, называют $\alpha$-эквивалентными.

# Комбинáторная логика

- Комбинаторы можно определить как примитивы, задав их вычислительное поведение на аргументах-переменных:

**Классические комбинаторы**

$$\begin{aligned} \mathbf{I}\,x &= x \\ \boldsymbol{\omega}\,x &= x\,x \\ \mathbf{K}\,x\,y &= x \\ \mathbf{S}\,f\,g\,x &= f\,x\,(g\,x) \end{aligned}$$

- Вычисление опять задается подстановкой определения с заменой формальных аргументов на фактические:

$$\boldsymbol{\omega}\,\mathbf{I} \to \mathbf{I}\,\mathbf{I} \to \mathbf{I}.$$

- Выбрав базис (например $\mathbf{S}, \mathbf{K}$), получим исчисление, эквивалентное $\lambda$-исчислению.

# План лекции

# Подстановка: нюансы и проблемы

Подстановка выполняется только вместо свободных вхождений:

$$[x \mapsto \lambda z.\, z](x\,(\lambda x.\, x\,y)\,x) = (\lambda z.\, z)\,(\lambda x.\, x\,y)\,(\lambda z.\, z)$$

Проблема *захвата переменной* (variable capture):

$$[x \mapsto y](\lambda y.\, x\,y) = \lambda y.\, y\,y$$

**Соглашение Барендрегта**

Имена связанных переменных всегда будем выбирать так, чтобы они отличались от имён свободных переменных.

Тогда коллизий, связанных с захватом, можно избежать:

$$[x \mapsto y](\lambda y'.\, x\,y') = \lambda y'.\, y\,y'$$

Однако удобнее встроить переименование в определение подстановки.

# Подстановка терма

## Определение подстановки $[x \mapsto N]\, M$

***Подстановка*** терма $N$ вместо свободных вхождений переменной $x$ в терм $M$ задается индукцией по $M$:

$$
\begin{array}{lcll}
[x \mapsto N]\ x & = & N, & \\
[x \mapsto N]\ y & = & y, & \\
 & & & \\
[x \mapsto N]\ (P\,Q) & = & ([x \mapsto N]\ P)\,([x \mapsto N]\ Q), & \\
 & & & \\
[x \mapsto N]\ (\lambda x.\, P) & = & \lambda x.\, P, & \\
[x \mapsto N]\ (\lambda y.\, P) & = & \lambda y.\, [x \mapsto N]\ P, & \text{если } y \notin \mathrm{FV}(N), \\
[x \mapsto N]\ (\lambda y.\, P) & = & \lambda z.\, [x \mapsto N]\ ([y \mapsto z]\ P), & \text{если } y \in \mathrm{FV}(N).
\end{array}
$$

Подразумевается, что $x \not\equiv y$, а $z$ — свежая, то есть $z \notin \mathrm{FV}(N) \cup \mathrm{FV}(P)$.

# Отношение $\beta$-эквивалентности

**Основная схема аксиом: для любых $M, N \in \Lambda$**

$$(\lambda x.\, M)N \;=_\beta\; [x \mapsto N]\, M \qquad \text{(правило } \beta\text{)}$$

- Логические аксиомы и правила

$$\begin{array}{lll} M \;=_\beta\; M; & & \\ M =_\beta N & \Rightarrow & N =_\beta M; \\ M =_\beta N, N =_\beta L & \Rightarrow & M =_\beta L. \end{array}$$

- Правила совместимости

$$\begin{array}{lll} M =_\beta M' & \Rightarrow & M\,Z =_\beta M'\,Z; \\ M =_\beta M' & \Rightarrow & Z\,M =_\beta Z\,M'; \\ M =_\beta M' & \Rightarrow & \lambda x.\, M =_\beta \lambda x.\, M' \quad \text{(правило } \xi\text{)}. \end{array}$$

- Если $M =_\beta N$ доказуемо в $\lambda$-исчислении, пишут $\lambda \vdash M =_\beta N$.

# Отношение $\alpha$-эквивалентности

Иногда вводят схему аксиом $\alpha$-преобразования:

$$\lambda x.\, M \;=_\alpha\; \lambda y.\, [x \mapsto y]\, M, \text{ если } y \notin \mathrm{FV}(M) \quad (\text{правило } \alpha).$$

Для рассуждений достаточно соглашения Барендрегта, но для компьютерной реализации $\alpha$-преобразование полезно:

Пусть $\omega = \lambda x.\, x\, x$ и $1 = \lambda y\, z.\, y\, z$. Тогда

$$\begin{aligned}
\omega\, 1 &= (\lambda x.\, x\, x)\, (\lambda y\, z.\, y\, z) \\
&=_\beta (\lambda y\, z.\, y\, z)\, (\lambda y\, z.\, y\, z) \\
&=_\beta \lambda z.\, (\lambda y\, z.\, y\, z)\, z \\
&=_\alpha \lambda z.\, (\lambda y\, z'.\, y\, z')\, z \\
&=_\beta \lambda z\, z'.\, z\, z' \\
&=_\alpha \lambda y\, z'.\, y\, z' \\
&=_\alpha \lambda y\, z.\, y\, z \;=\; 1
\end{aligned}$$

# Индексы Де Брауна (De Bruijn)

- ***Индексы Де Брауна (De Bruijn)*** представляют альтернативный способ представления термов.
- Связанные переменные не именуются, а индексируются, индекс показывает, сколько лямбд «назад» переменная была связана:

$$\begin{aligned} \lambda x.\,(\lambda y.\,x\,y) &\leftrightarrow \lambda\,(\lambda\,1\,0) \\ \lambda x.\,x\,(\lambda y.\,x\,y\,y) &\leftrightarrow \lambda\,0\,(\lambda\,1\,0\,0) \end{aligned}$$

- Свободные переменные при этом получают индексы, превышающие число лямбд слева:

$$\lambda x.\,z\,x\,y \leftrightarrow \lambda\,2\,0\,1$$

- При таком представлении все $\alpha$-эквивалентные термы кодируются одинаково, и коллизий захвата переменной не возникает.

# Отношение $\eta$-эквивалентности

- Схема аксиом $\eta$-преобразования:

$$\lambda x.\, M\, x \;=_\eta\; M \qquad \text{(правило } \eta\text{)}$$

  в предположении, что $x \notin \mathrm{FV}(M)$.
- Для определения отношения эквивалентности надо, конечно, добавить общелогические аксиомы и правила. Разумно также обеспечить совместимость.
- Смысл $\eta$- эквивалентности в том, что аппликативное поведение термов слева и справа от знака равенства одинаково; для произвольного $N$ верно

$$(\lambda x.\, M\, x)\, N =_\beta M\, N$$

# Функциональная экстенсиональность

- $\eta$-преобразование обеспечивает принцип ***экстенсиональности***: две функции считаются экстенсионально эквивалентными, если они дают одинаковый результат при любом одинаковом вводе:

$$\forall N \ : \ F\,N = G\,N.$$

- Выбирая $y \notin FV(F) \cup FV(G)$, получаем ($\xi$, затем $\eta$)

$$\begin{aligned} F\,y &= G\,y \\ \lambda y.\,F\,y &= \lambda y.\,G\,y \\ F &= G \end{aligned}$$

- В Хаскелле так называемый «бесточечный» стиль записи основан на $\eta$-преобразовании.