GeekBrains

Функциональное программирование (лекция 4)

Парадигмы программирования и языки парадигм

Оглавление

Вступление

Добрый день, коллеги! Меня зовут Александр Левин. В сегодняшней лекции мы изучим парадигму функционального программирования (ФП). Сначала мы будем рассматривать отдельные аспекты функционального программирования, отдельные определения, концепции, языки, историю, а затем, как всегда, соединим кусочки в единое целое на конкретных примерах. Таким образом, план на нашу сегодняшнюю лекцию:

- Что такое функциональное программирование
    - Контекст
    - Определения
    - Языки программирования
- Концепции ФП
- История вкратце
- Примеры кода
- Преимущества и недостатки подхода
- Подведение итогов

Если после окончания лекции у вас останутся вопросы, вы как всегда сможете написать их мне в Telegram по контакту, который я оставлю в конце лекции.

Термины, используемые в лекции

Функциональное программирование — это парадигма программирования, которая основывается на представлении программы в виде вычисления функций и их композиций

Функция (в программировании) — подпрограмма, которую можно вызвать по имени

Функция (в математике) — отображение из A в B - это правило, которое ставит в соответствие каждому элементу a из множества A ровно один элемент b из множества B

Композиция двух функций — применение результата первой функции ко второй

Побочные эффекты (англ. side-effects) — любые операции работающей программы, изменяющие состояние среды

Чистая функция — это подпрограмма, зависящая только от своих аргументов, возвращающая одинаковые значения при одинаковых аргументах и не имеющая побочных эффектов

Неизменяемый объект — это объект, который невозможно изменить после инициализации

Изменяемый объект — это объект, который возможно изменять после инициализации

Функции высшего порядка - функции, которые могут принимать и возвращать другие функции

Рекурсия - вызов функции из самой себя

Энергичные вычисления — стратегия вычисления, согласно которой результат всех выражений вычисляется сразу

Ленивые вычисления — стратегия вычисления, согласно которой вычисления откладываются до последнего момента, до тех пор пока их результат не требуется пользователем в явном виде

Генератор в Python — функция, которая возвращает по одному объекту за один вызов

Каррирование или **карринг** (англ. - currying) — процесс преобразования функции от многих аргументов в последовательность функций одного аргумента

Лямбда-исчисление — формальная система, которая используется для описания вычислений в терминах анонимных функций

Что такое функциональное программирование

Начинаем изучение функционального программирования.

Контекст

Сразу начнем с любимого вопроса: зачем это нужно? Короткий ответ: в некоторых случаях код становится более понятными производительным, а внесение изменений и поддержка - проще, ровно также как и в других парадигмах, которые мы рассмотрели на текущий момент. Но функциональный подход кардинально отличается от уже рассмотренных. Основное отличие функционального стиля от императивного - это оперирование черными ящиками - "**функциями**", плюс несколько очень важных концепций (их мы ещё подробно разберем), которые расширяют базовое представления о функциях в программировании.

Функциональное программирование является более узким направлением по сравнению с тем же объектно-ориентированным подходом, но крайне важным как минимум на уровне понимания, если вы занимайтесь разработкой чего-то из нижеперечисленного:

- Распределённые системы
- Препроцессинг и анализ данных
- Машинное обучение и искусственный интеллект
- Научные вычисления, например:
    - в биоинформатике для анализа ДНК
    - в квантовых вычислениях для создания абстракций кубитов и квантовых логических элементов

Вот некоторые реальные кейсы использования функционального программирования в индустрии:

- Meta (бывш. Facebook) - социальная сеть, использует Haskell для борьбы со спамом.
    - Из статьи "Fighting Spam with Haskell: "Haskell measures up quite well: It is a purely functional and strongly typed language, and it has a mature optimizing compiler and an interactive environment (GHCi). It also has all the abstraction facilities we would need, it has a rich set of libraries available, and it's backed by an active developer community."

- Twitter - социальная сеть, использует Scala для бекенда.
    - Из статьи “Effective Scala”: “Scala is one of the main application programming languages used at Twitter. Much of our infrastructure is written in Scala and we have several large libraries supporting our use”
- Jane Street - трейдинговая компания, использующая функциональное программирование для работы с финансовыми данными.
    - С сайта Jane Street Technology: “From systems automation to trading systems, from monitoring tools to research code, we write everything that we can in OCaml”
- NASA - государственная аэрокосмическая компания
    - Из статьи 1986 года: "Clearly, functional languages are radically different from traditional programming languages. In fact, the primary motivation behind the research on functional languages is a dissatisfaction with traditional programming languages”

С другой стороны, если ваша задача описать взаимодействие объектов (например, при написании игр или симуляций) - конечно ваш вариант это ООП; а функциональная парадигма вряд ли сильно поможет в этом случае, поскольку её преимущества лежат в работе с множествами и функциями, а не описании взаимодействий объектов. И конечно это понятия не взаимоисключающие для проекта в целом, но могут быть таковыми для задач низкого уровня. Мы рассмотрим как раз такой пример чуть позже в разделе с примерами. Тоже самое применимо для некоторых случаев во фронтенде: например, раскрасить кнопку скорее всего проще с помощью JS (например, фреймворка Vue.js), а не Haskell или Scala.

> 💡 Хотя для фронтенда тоже есть интересные решения, например функциональный язык **Elm**, который умеет взаимодействовать как с JS, так и с HTML и CSS.

Также, функциональная парадигма не упростит вам жизнь, если вы работаете с алгоритмами, которые плохо разбиваются на математические функции. В этом случае ваш вариант, скорее всего, - структурный и процедурный подходы.

Функции и функции

В чем же особенность конкретно этой парадигмы? Во-первых, оно построено на функциях приближенных к математическому смыслу. Через пару минут мы детально разберемся что здесь имеется в виду. Во-вторых, функциональное программирование является дочерней по отношению к декларативной парадигме. Как знаем из первой лекции, это означает, что совсем скоро мы будем использовать “черные ящики”, в которые у нас нету доступа, но которым мы можем “объявить” то,

что мы хотим получить. А также где-то мы увидим запрет на доступ к памяти, а значит и возможность изменить состояние программы и значения переменных. Про все это мы поговорим чуть позже. Начнем с определения функции.

Функция - в словосочетании “**функциональное программирование**” означает “**отображение**”, то есть “функцию” в математическом смысле. Функциональное программирование основано на функциях именно в это значении. Давайте подробнее обсудим что это значит. Но сначала несколько определений.

(определения взяты из: Верещигин, Шень - “Начало теории множеств”)

1. “Иногда вместо функций говорят об отображениях (резервируя термин «функция» для отображений с числовыми аргументами и значениями). Мы не будем строго придерживаться таких различий, употребляя слова «отображение» и «функция» как синонимы.”
2. “Любое подмножество R множества A × B называется **отношением** между множествами A и B.”
3. “Отношение F ⊂ A × B называется **функцией** из A в B, если оно не содержит пар с одинаковым первым членом и разными вторыми. Другими словами, это означает, что для каждого a ∈ A существует не более одного b ∈ B, при котором {a, b} ∈ F.”

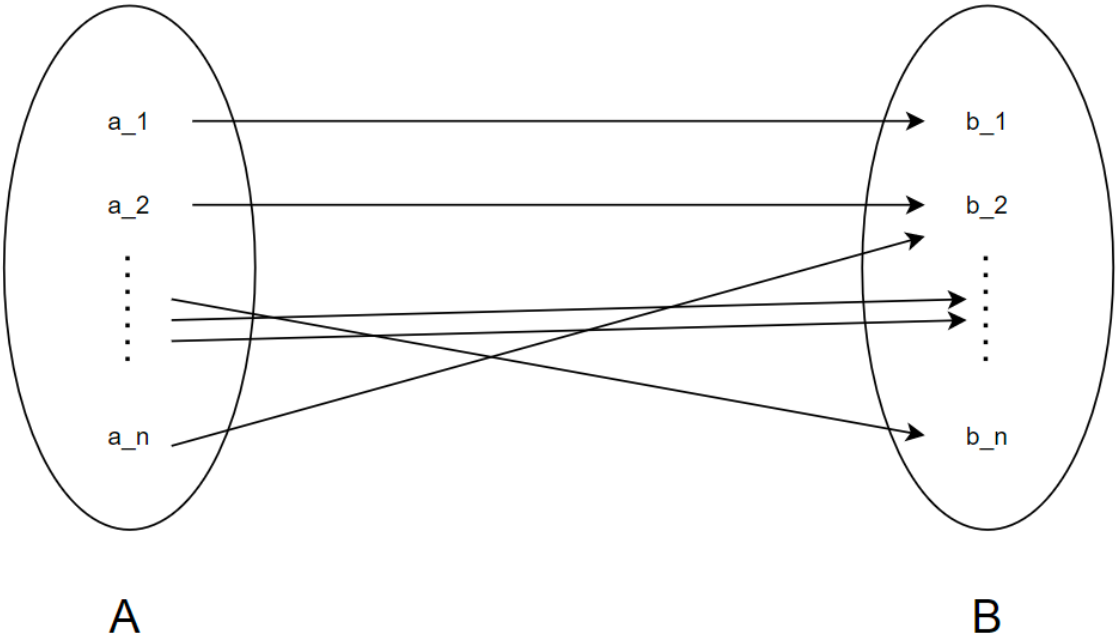
Теперь разберем определения пошагово прикладным языком. Есть два множества *A* и *B*. Будем создавать пары элементов, вытягивая один элемент из *A*, а второй элемент из *B*. Например, пара элементов {a_10, b_5}, где a_10 – элемент из множества A, b_5 – элемент из множества B:

$$\{ a\_10, b\_5 \}$$

Пример упорядоченной пары элементов

Давайте теперь создадим множество Y = A x B, элементы которого - это все возможные такие пары элементов. Такое множество называется “Декартовым произведением” (англ. - **Cartesian Product**). Теперь выберем любое подмножество R элементов из множества Y. Поскольку элементы подмножества Y всё ещё упорядоченные пары - то у нас получилось множество каких-то пар. Поскольку это множество является подмножеством декартова произведение, то оно называется **отношением**. Вообще говоря, отношения могут быть разными. Но существуют случаи, при которых мы выбрали подмножество таких элементов, для которых верно, что для каждого из элементов из A (слева) существует единственный элемент из B (справа). Такие отношения и называются **функциями**.

Таким образом, **отображение из A в B** (f: A -> B) - это правило, которое ставит в соответствие каждому элементу a из множества A ровно один элемент b из множества B. Рассмотрим на примере. Предположим, у нас есть абстрактные множества A и B. Мы знаем, что **функция** для каждого элемента a_i из A ставит в соответствие один единственный b_i из B. То есть запрещены ситуации, в которых одному элементу слева соответствует несколько элементов справа. Например, когда одному a_1 соответствует и b_3 и b_6 одновременно. При этом конечно же, не запрещены ситуации совпадения элементов справа (например, a_1 соответствует b_3 и a_2 соответствует b_3).



На картинке - каждая стрелка представляет собой упорядоченную пару, а множество всех стрелок - множество всех пар, то есть **отношение**. Если из каждого элемента из A (слева) стрелка выходит только один раз, значит мы смотрим на отношение, которое ещё и является **отображением**.

Зачем все это нужно. Функциональное программирование часто используется для работы с массивами, которые могут быть реализованы очень по-разному, но с математической точки зрения являются множествами (иногда, упорядоченными). И если большую часть времени вы хотите, например, из списка элементов получать список элементов, может быть очень удобно рассуждать о них в терминах множеств и отображений: с точки зрения производительности, и понятности, лаконичности вашего кода. Приведу очень простой пример на языке Python. Задача: на вход подается целочисленный массив. Необходимо получить список, элементы которого

- это остаток от деления соответствующего элемента в исходном массиве на 5. Можно думать об этом как о последовательности действий: объявить новый список, взять элемент, получить остаток деления, положить в новый список и продолжать так до конца полученного списка.

```
DIVISION_NUM = 5
arr = [i for i in range(100)]
res = []
for el in arr:
    res.append(el % DIVISION_NUM)
```

Но с другой стороны, можно думать об этом как об отображении modulo из исходного списка arr в новый список res.

```
DIVISION_NUM = 5
arr = [i for i in range(100)]
modulo = lambda x: x % DIVISION_NUM
list(map(modulo, arr))
```

В этом примере исчез цикл, и, что самое главное, нам не интересен каждый элемент по отдельности. но куда интереснее это то отображение, которое мы используем для получения нового массива. В нашем случае отображение modulo реализовано в виде анонимной функции **lambda**. А вычисление результата происходит с помощью встроенного метода **map**.

Поэтому, очень важно, что **функциональное** и **процедурное программирование** - это РАЗНЫЕ парадигмы. Ровно как и слово **функция** здесь означает вовсе не подпрограмму, не последовательность действий.

Математическая функция предоставляет нам информацию о том как устроено какое-то правило. Функция позволяет анализировать правило единым целым, не акцентируя внимания на отдельных элементах (частных случаях). Поэтому, для анализа мы обычно совершаем манипуляции с самой функцией, вместо пошаговых вычислений её значений. Это очень эффективная и мощная методология, которой посвящен целый раздел математики “Математический анализ” (англ. Calculus).

С другой стороны, в программировании, как мы уже знаем, **процедуры** и **функции** - это последовательность команд которые пошагово изменяют состояние программы, после чего возвращается (или не возвращается) результат. Состояние программы - это некое множество переменных и их значения в отдельный момент времени. И это состояние меняется шаг за шагом, строка за строкой. А **процедура или функция**

- это **подпрограмма** (последовательность команд), которую можно вызвать по имени. То есть некая обертка для отдельных кусков кода.

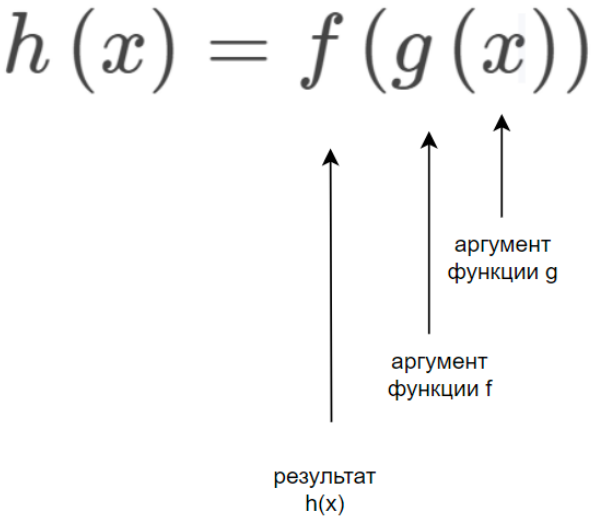
Термин “функциональное программирование”

Теперь разберемся с заголовком этой лекции. **Функциональное программирование** - это парадигма программирования, которая основывается на представлении программы в виде вычисления функций и их **композиций**. Причем, **функции** принимают входные данные и возвращают выходные данные без изменения состояния программы, то есть являются **чистыми**, (к этому мы ещё вернемся).

Композиция двух функций - это применение результата первой функции ко второй. Ещё можно сказать, что это “последовательное применение” функций. Простыми словами: есть две функции. Берем вход, например какое-нибудь число x, кладем в первую функцию g и получаем результат g(x). Потом берём этот результат и кладем во вторую функцию f. То, что получилась третья функция h(x) = f(g(x)) - это результат композиции.

$$h(x) = f(g(x))$$

Важно, что последовательность применения функций идёт “изнутри - наружу”. Сначала вычисляется внутренняя функция g, результат которой является аргументом внешней функции f, а затем вычисляется значение f(g(x)).



Таким образом, в определении имеется в виду, что мы можем вычислять результаты функций как по-отдельности, так и вместе, объединяя их в композиции.

Языки программирования

Рассмотрим некоторые из наиболее популярных функциональных языков.

Чистые функциональные языки	Мультипарадигменные функциональные языки
• Haskell • Miranda • PureScript • Elm • jq	• Python • Scala • F# • ML • Rust • Lisp • Erlang

И да, наш любимый Python тоже поддерживает функциональную парадигму. Мы, кстати, уже знакомились с ней в первой лекции, когда говорили о встроенных методах *sum()* и *reduce()*.

Концепции ФП

Чтобы **подпрограмма** стала **отображением**, необходимо обогатить её некоторыми свойствами. Эти концепции являются неотъемлемыми частями функциональной парадигмы. В нашей лекции всего их будет пять. Поехали!

Чистые функции

Начнём с так называемых “**чистых функций**” (англ. - pure functions). Для того чтобы разобраться, нам понадобится определение “побочного эффекта”. **Побочный эффект** (англ. side-effects) - любые операции работающей программы, изменяющие состояние среды. Например, если вы изменяете уже созданную переменную - это побочный эффект.

Теперь попробуем разобраться с чистыми функциями. **Чистая функция** - это такая функция (в программировании), которая:

1. Всегда возвращает значение, зависящее только от ее аргументов
2. Всегда возвращает одинаковое значение при одинаковых аргументах
3. Не имеет побочных эффектов

Если говорить простыми словами, то функция чистая, если во-первых все операции у неё происходят внутри, без влияния на внешнюю среду, а во-вторых результат вычислений всегда один и тот же для одинаковых входов (ещё говорят, что функция

детерминирована). Пример чистой функции на языке Python - функция возведения в степень:

```
1 def pow(x, y):
2     return x ** y
```

Предположим, вы передали x = 5, y = 6. Вот сколько раз вы не передадите эти значения, а результат все равно останется тем же. Таким образом, функция является детерминированной. А есть ли у неё побочные эффекты? Ответ: нету, поскольку эта функция возвращает новое значение “x в степени y” без изменения исходных. Таким образом, наша функция *pow* является **чистой**.

💡 Кстати, в Python тип int (целое число) является **неизменяемым** (**immutable**): когда вы думаете, что изменяете int в Python-е, интерпретатор создает новую переменную, которую просто называет прежним именем, и вам кажется, что эта та же самая переменная. Но на самом деле, это уже новый объект, поэтому строго говоря, int (как и некоторые другие типы) в Python изменить невозможно. Можете проверить данную особенность языка с помощью встроенной функции id().

💡 К **неизменяемым** типам в Python помимо int относятся например: float, tuple, complex, string, frozenset.

Конечно, эту функцию можно сделать “нечистой”, если нарушится одно из условий. Например, если функция будет зависеть от чего-то чего нет в её сигнатуре:

```
1 x = 5
2 def pow(y):
3     return x ** y
```

В данном примере мы всегда будем возводить в степень пятерку, поэтому функция здесь вроде бы как детерминирована, но поскольку пятерка находится за пределами функции, то функция *pow* перестаёт быть чистой.

Неизменяемость

Возможно вы уже слышали, что в языках программирования типы данных бывают **изменяемые** (mutable) и **неизменяемые** (immutable). Для нас это важно, поскольку в функциональном программировании все структуры данных - **неизменяемые**. Что это означает: **неизменяемый объект** - это такой, который невозможно изменить после инициализации (под объектом как правило подразумевается переменная или функции). Следовательно, принять решение о состоянии объекта можно только во время инициализации, но никак не после! С другой стороны, **изменяемые** объекты

можно без ограничений изменять после инициализации. Получается, что если мы имеем **неизменяемый** объект и хотим его изменить, то единственное что нам остается это создать новый объект, который будет основан на старом, но уже с учетом желаемых изменений. А если наш объект **изменяемый**, то мы можем просто изменить его прямо в той же ячейке памяти, в которой он лежит.

Неизменяемость типов данных - это необходимое условия для функционального программирования: оно делает его более предсказуемым, безопасным и упрощает отладку. Также, оно предоставляет дополнительные возможности для оптимизации: например возможность кешировать значения функций в точках, поскольку все функции являются **чистыми**, а переменные гарантированно не будут изменены во время исполнения (являются **неизменяемыми**).

Функции высшего порядка

Функциональное программирование позволяет писать так называемые **функции высшего порядка** - то есть такие, которые могут принимать и возвращать другие функции.

💡 В функциональном анализе для функций ставящих в соответствие другую функцию зарезервировано слово “**оператор**”.

```
1 def function_getter(func):
2     print("I got the function: ", func.__name__)
3     return func
```

Ещё говорят, что язык программирования обладает функциями высшего порядка, если функция в этом языке является таким же объектом, как и обычная переменная. То есть, опять же, её можно

- передать в функцию в качестве аргумента
- получить из другой функции в качестве возвращаемого значения
- сохранить в переменную и позже использовать

Ранее, когда речь шла об отображениях, мы использовали встроенный Python метод *map()*. Этот метод как раз является **функцией высшего порядка**, поскольку он принимает функцию на вход.

💡 Для функций передаваемых в другие функции как обычный объект существует отдельный термин - **callback**. Конечно же, если вы используете **callback**, это вовсе не означает что вы используете **функциональное программирование**, но точно означает, что вы используете как минимум один из её базовых принципов - **функции высшего порядка**.

💡 **Callback**-и довольно активно используются в различных областях. Например, в машинном обучении, в ситуациях, когда вам нужно вызывать функцию с какой-то периодичностью, например, чтобы каждую эпоху обучения обновлять какой-нибудь гиперпараметр.

Это позволяет делать функции более универсальными абстрактными. Например, создавать сложные композиции функций, при этом не вычисляя в моменте всю цепочку для конкретной переменной, а просто сохраняя эту композицию на будущее для дальнейшего использования.

Рекурсия

Возможно, вы уже знакомы с рекурсией. **Рекурсия** - это вызов функции из самой себя в процессе исполнения. Существуют алгоритмы, реализация которых получается гораздо проще и лаконичнее через рекурсию. Канонический пример, когда с рекурсией гораздо удобнее чем без неё - числа Фибоначчи. Числа Фибоначчи - это последовательность чисел, в которой первые два числа равны 0 и 1 соответственно, а каждое следующее является суммой двух предыдущих. А мы, как пользователи, можем захотеть получить из этой последовательности число с индексом n. Рассмотрим реализацию чисел Фибоначчи на Python:

```python
def fibonacci(n):
    if 0 ⩽ n < 2:
        return n
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

Что тут написано: есть два тривиальных кейса, когда n = 0 и n = 1. В этом случае все понятно. А во всех остальных случаях, мы возвращаем сумму результата этой же функции для n-1 и n-2 - это и есть **рекурсия**. То есть интерпретатор будет падать в else до тех пор, пока не дойдёт до тривиальных кейсов, где n = 0 и n = 1.

Ленивые вычисления

В контексте функционального подхода нас интересуют два типа стратегии вычислений: “**энергичные**”(англ. - eager) и “**ленивые**” (англ. - lazy). Чем же они отличаются? При стратегии энергичных вычислений, результат всех выражений вычисляется сразу. А “**Ленивое**” или “**отложенное**” вычисление (англ. - Lazy Evaluation) - наоборот, это стратегия вычисления, согласно которой вычисления следует откладывать до последнего момента, до тех пор пока их результат не требуется пользователем в явном виде.

Зачем это нужно? Во-первых, оптимизация ресурсов, поскольку мы не загружаем в память все результаты всех вычислений, а только те, которые нам точно нужны. К тому же не тратим время на их вычисление. Таким образом, мы получаем возможность работать с очень большими данными и объектами. Во-вторых, это позволяет нам инициализировать бесконечные объекты и работать с ними, не требуя при этом бесконечное количество памяти.

Кстати, в Python есть функционал, позволяющий работать с ленивыми вычислениями: создавать собственные ленивые функции и вызывать их пошагово. Этот функционал реализован с помощью **генераторов** - функций, которые возвращают по одному объекту за один вызов. Вот пример реализации и вызова простого генератора, перебирающего числа от 0 до бесконечности:

```
1 def lazy_function():
2     i = 0
3     while True:
4         yield i
5         i += 1
6 
7 f = lazy_function()
8 print(next(f)) # 0
9 print(next(f)) # 1
10 print(next(f)) # 2
```

Важным условием для генератора является наличие цикла внутри и ключевого yield, которое означает вернуть “значение текущего шага цикла”. Здесь мы объявляем i равным нулю. Далее, с помощью *while True* объявляем бесконечный цикл, на каждом шаге которого i сначала возвращается, а затем - увеличивается. Далее в строках с 7 по 10 мы инициализируем и используем этот генератор с помощью встроенного метода *next()*, который делает очередной шаг цикла *while* и возвращает то, что написано после *yield*, в нашем случае - это элемент *i*.

Каррирование

Эта концепция, как и язык Haskell, была названа в честь математика Хаскелла Карри. **Каррирование** или **карринг** (англ. - currying) - это процесс преобразования функции от многих аргументов в последовательность функций одного аргумента. Например, если у нас была функция с сигнатурой *f(a, b, c)*, тогда её карриванная версия - *f(a)(b)(c)*. Рассмотрим на примере функции сложения.

Каррирование дробит функцию на более мелкие функции, позволяя рассматривать шаги вычисления независимо. Оно помогает получать “промежуточный” результат в случаях, когда не все из аргументов доступны прямо сейчас, а значит появляется

пространство для оптимизации ресурсов. Также, карринг помогает сделать ленивые вычисления ещё более ленивыми. Благодаря ему мы можем вычислять функцию не полностью. Предположим есть цикл, который исполняется лениво и на каждом его шаге вычисляется 3 функции. Следовательно, теперь мы можем не только лениво вычислять один шаг этого цикла, а например только одну или две функции из трёх этого шага.

Рассмотрим пример кода. Вот у нас есть функция двух аргументов, возвращающая сумму за один вызов add.

```
1 def add(x, y):
2     return x + y
3 add(2, 5) # 7
```

Применим каррирование к этой функции.

```
1 def add(x):
2     def add_x(y):
3         return x + y
4     return add_x
5 add(2)(5)
```

Получилась новая функция: принимающая на вход один аргумент и возвращающая вторую функцию. Естественно, для того, чтобы получить сумму, её (уже вторую функцию) необходимо вызвать ещё раз и тоже передать один аргумент. Теперь мы получили такой же результат, как и раньше, но с помощью каррированной функции. Здесь, каждый вызов будет содержит только один аргумент, а в случае, если вычисление всё ещё не окончательно - возвращается не вычисленное значение, а просто функция, которая просит следующий аргумент.

На этом мы заканчиваем обсуждение концепций функционального программирования.

История вкратце

Предлагаю вкратце разобрать его историю (функционального подхода).

Лямбда-исчисление

Начнем с математики. Теоретическая основа функционального программирования - это так называемое “**Лямбда-исчисление**” (λ-исчисление, англ. Lambda Calculus). С него и начинается история **функционального программирования**.

Рассмотрим вкратце. **Лямбда-исчисление** - это формальная система, которая используется для описания вычислений в терминах функций. Важно отметить, что это не язык программирования, а только математический формализм, “диалект” для работы с **анонимными функциями** (функциями без имени). Эта система была разработана Алонзо Чёрчем в 1930-х годах. Лямбда-исчисление основано на использовании лямбда-выражений для определения функций и их применения. **Лямбда-выражение** - это формальное правило, которое определяет функцию в терминах ее аргументов. Оно состоит из переменных, функций и операторов применения. Вот например как на языке лямбда-исчисления выглядит функция возведения в квадрат одного аргумента.

$$\lambda x.\, x^2$$

Говоря прикладным языком, с помощью этой формальной системы, у вас есть возможность:

1. Определять анонимные функции
2. Применять (вызывать) функции к конкретным аргументам

Если вам интересно узнать больше, рекомендую, например, почитать [статью в энциклопедии Стэнфорда]().

> 💡 Такие функциональные языки программирования как например Haskell, Lisp, ML, основаны как раз на принципах из **Lambda Calculus**.

Lisp и Miranda

В 1950-х годах известнейший ученый Джон Маккарти, который по совместительству считается автором термина “искусственный интеллект”, создал один из первых функциональных языков программирования - **Lisp**. Lisp расшифровывается как “List Processing”, т.к. Lisp был предназначен для работы со списками как основной структуры данных. Это было инновацией. Также, в **Lisp** появились **функции первого класса** (возможность передавать функции в качестве аргумента в другие функции) и использовании **рекурсии** (вызов функции из самой себя). Также, Lisp обладал динамической типизацией, то есть возможностью изменить тип объекта после его инициализации (в Haskell, кстати, такой возможности нет), а также собственным интерпретатором.

В 1985 году Дэвид Тёрнер представил язык **Miranda** - первый язык программирования **ленивыми вычислениями** (lazy evaluation). Miranda к слову являлась проприетарным ПО, что возможно помешало её активному распространению.

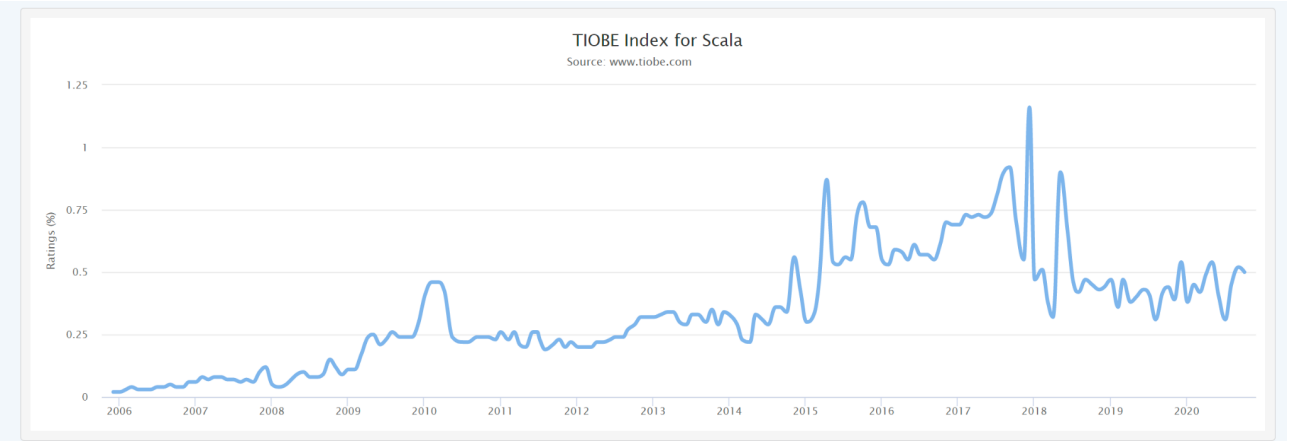
Haskell

Но у реализаций языков в функциональной парадигме оставались проблемы, мешающие широко применять её. В 1987 в городе Портленд, штата Орегон состоялась конференция по “Функциональному программированию и компьютерной архитектуре” (FPCA ‘87), на котором было предложено создать функциональный язык, который учитывал бы следующие условия с целью получить широкое признание (перевод с английского):

- Он должен быть пригоден для преподавания, исследований и приложений включая разработку больших систем
- Он должен быть полностью описан с помощью формального синтаксиса и семантики, которые должны быть в свободном доступе.
- Сам язык тоже должен быть в свободном доступе. Любому человеку должно быть разрешено внедрять этот язык и распространять его, кому заблагорассудится.
- Он должен основываться на идеях, пользующихся широким признанием.
- Он должен уменьшить ненужное разнообразие в функциональных языках программирования

По итогу этой конференции был создан специальный комитет разработчиков, который представил первые результаты в 1990 году (так называемый, Haskell Report). В нём они описывали новый язык программирования - **Haskell**, который был назван в честь известного математика Хаскелла Карри. **Haskell** был вдохновлен языком Миранда, вышедшей за 2 года до конференции FPCA, поэтому на самом деле он очень на неё похож. В итоге **Haskell** стал (и до сих пор является) одним из наиболее популярных функциональных языков.

За последние 30 лет, что функциональное программирование неспешно набирало (и продолжает набирать) популярность. Haskell на сегодняшний день активно поддерживается и развивается, хотя и не используется даже и близко так широко как какой-нибудь Python или Java. Например, на день разработки этого курса последний апдейт компилятора GHC - версия 9.6.1 от 10 марта 2023 года. Также, набирает известность и **Scala**, созданная в 2003 году. Она активно используется при работе с большими данными и машинным обучением. Вот например график индекса **TIOBE** для языка **Scala**.



Ну а дальнейшее будущее индустрии и науки, конечно же, зависит от вас с нами!

Примеры кода

Разберем примеры, чтобы получить практическое представление о функциональном программировании. В данном разделе мы, в духе этого курса, будем рассматривать примеры на языке Python, который, как мы уже знаем, поддерживает функциональную парадигму; а также на чисто функциональном языке - Haskell, поскольку этот язык является очень важным и известным в функциональной парадигме. И конечно же, если используете функциональную парадигму, Haskell для вас будет на порядок быстрее Python-а. Поэтому, если для ускорения ООП кода, который вы написали на Python, вы скорее всего будете переписывать его на какой-нибудь C++, то в случае с функциональным кодом либо на Си, либо на Haskell. Конечно, это в общем случае. И функциональный код на Haskell будет выглядеть лаконичнее. Но и на изучение данного языка придется выделить время.

Вообще, я намеренно не буду приводить сильно сложных примеров в данной лекции, поскольку считается, что парадигма функционального программирования имеет ещё более высокий порог вхождения чем ООП. Но в отличие от ООП применяется она на практике гораздо реже. Тем не менее, данный подход способен расширять сознание. Поэтому переходим к первому примеру.

Сумма квадратов

Рассмотрим простую задачу. На вход подается список с целыми числами. Необходимо возвести все числа в квадрат и вернуть их сумму (сумму квадратов).

Python

Начнем с реализации на Python.

```python
def sum_squares(lst: list) -> int:
    return sum(map(lambda x: x**2, lst))
```

Что мы видим: мы видим процедуру sum_squares, которая принимает переменную lst типа список и возвращает число int. Но во второй строке происходит самое интересное. Напомню, что lambda в Python - это анонимная функция, то есть одноразовая функция без имени. В нашем случае lambda - это функция которая принимает один аргумент (здесь он называется x) и возвращает x в степени 2. Ясно, что мы смотрим на функцию возведения одного любого числа в квадрат. Но мы хотим возвести в квадрат целый список числе. Для этого, мы берем эту функцию и передаем её в map, а рядом передаем список, к элементам которого мы хотим её применить. Получается, что map возвращает новый список, элементами которого являются квадраты элементов первого списка. Кстати, **map** как раз переводится как отображение, поэтому в общем-то логично, что он берёт массив данных и возвращает такой же массив, ставя каждом элементу в соответствие Какой-то, исходя из правила, которое нам интересно (в нашем случае - это возведение числа в квадрат). Ну а в конце все совсем просто, мы применяем уже хорошо нам знакомую функцию суммы, которая принимает на вход итерируемый объект (на самом деле не обязательно тип list), а возвращает число, а именно - сумму элементов этого массива. Что здесь важно отметить:

1. Мы не перебирали элементы массива в цикле самостоятельно. За нас это сделал Python, и мы не знаем ничего про этот цикл.
2. Мы не определяли промежуточные переменные для элементов массива, кроме конечного результата - самой суммы квадратов. За нас, опять же, это сделал Python, и мы не знаем какие переменные в какой момент времени возникали под капотом. Мы вообще ничего не знаем про элементы этого массива, и мы к ним не прикасаемся даже на уровне абстракции.

Добавлю также, что “Сумма квадратов” реализована в виде **композиции** трех функций:

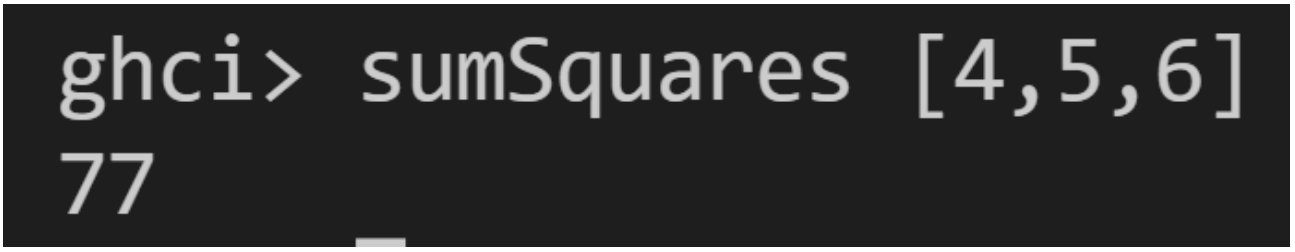
1. lambda (в нашем случае - возведения числа в квадрат)
2. map (в нашем случае - отображения из списка чисел в в список квадратов этих чисел)
3. sum

Haskell

Посмотрим то же самое на языке haskell

```haskell
sumSquares = sum . map (^2)
```

Если привыкнуть, то здесь синтаксис даже проще. Читаем также справа-налево. В скобках располагается оператор возведения в квадрат. Затем идёт функция **map**, которая кстати означает ровно то же самое что и в Python. Затем функция **sum**, которая, опять же совпадает с тёзкой в Python. Точка означает последовательное применение, то есть, как мы уже знаем - **композицию**. Здесь мы нигде не указываем что может быть подано на вход (хотя в Haskell такая возможность конечно же есть), и Haskell “догадался” за нас, что на вход будет подан массив (Array). Это избавляет нас от избыточного кода, который был в Python, когда мы объявляли процедуру. А вот так кстати будет выглядеть вызов в Haskell и ответ, который мы получим. Слово ghci в данном случае к коду не имеет отношения - это просто префикс интерпретатора, который используется в Haskell.



Структурно-процедурная реализация

А что если мы не умеем работать с функциональным программированием, и решили реализовать то же самое, но в структурно-процедурном стиле? Ну наверное получилось бы что-то вот такое:

```python
def sum_squares(lst: list) -> int:
    current_sum = 0
    for el in lst:
        current_sum += (el ** 2)
    return current_sum
```

Конечно, могут быть и другие реализации, но эту можно назвать “хорошей” с точки зрения алгоритмической сложности, поскольку проход по списку – только один. Читаем код: сигнатура нашей функции **sum_squares** точно такая же как была в функциональной реализации. Но дальше.. Объявляем переменную current_sum равной нулю, в ней будем накапливать результат. Далее перебираем элементы массива lst, и прибавляем к счётчику каждый элемент возведенный в квадрат. Затем возвращаем результат такого “накопления”. Что важнее всего в данном примере:

1. Мы создали переменную current_sum и знаем её значение на каждом шаге цикла
2. Мы явным образом используем цикл для перебора элементов

3. Мы “руками” прибавляем квадрат каждого элемента к общему результату, после чего “руками” её же возвращаем пользователю.

Поэтому, этот пример написан в чисто императивном стиле.

ООП реализация

Конечно, это немного странно применять ООП в рамках данного примера, но именно поэтому я продемонстрирую как бы это могло в теории выглядеть.

```
class List:
    def __init__(self, lst):
        self.lst = lst

    def calculate_sum_squares(self):
        current_sum = 0
        for el in self.lst:
            current_sum += (el ** 2)
        self.sum_squares = current_sum
```

В предыдущей лекции мы убедились в том, что ООП основан на объектах и объектном мышлении, поэтому первый и главный вопрос, который скорее всего возникнет у вас при написании подобного решения - это “что в моем случае будет являться объектом”. Я например решил, что объект - это и есть список, а в самом классе можно будет посчитать сумму квадратов, после чего положить в отдельный атрибут. Для расчета суммы квадратов используется специальный метод calculate_sum_squares, и после его вызова, сумма сохраняется в переменную sum_squares нашего объекта типа List. Как вам такая реализация? На мой взгляд, конечно выглядит громоздко, как и структурно-процедурный вариант. Зачем писать все эти классы и циклы, если можно просто сказать что нам нужна сумма от квадратов элементов списка и на этом все? Но у вас может быть альтернативное мнение, и это совершенно нормально! Решающим становится ваше мнение, привычки и вкусы плюс текущие условия и контекст вашей работы. Напоследок, хочу отметить, что если прямо совсем нужно писать класс (использовать ООП), то можно использовать и функциональный подход и ООП вместе, например вот так:

```
class List:
    def __init__(self, lst):
        self.lst = lst

    def calculate_sum_squares(self):
        self.sum_squares = sum(map(lambda x: x**2, self.lst))
```

То есть, парадигмы не являются взаимоисключающими. Главное понять что для вас важно в той или иной ситуации.

Функция композиции

Давайте, чтобы стало совсем интересно, разберем следующий пример про функцию композиции. Как мы знаем, композиция - это процесс объединения функций. А что если мы хотим получить функцию композиции? То есть получить такую функцию, которая принимает на вход две функции, а возвращает функцию, являющуюся композицией двух полученных?

Python

В рамках функциональной парадигмы это сделать очень просто! Попробуйте сами написать такую функцию на Python. Если не получится, ничего страшного, результат вы в любом случае увидите на следующем слайде.

– пауза –. Давайте посмотрим на него.

```
1 def compose(f, g):
2     return lambda x: f(g(x))
```

И так, мы видим, что здесь все ровно как в определении: объявляется функция compose, у неё есть два аргумента: *f* и *g*. Функция compose возвращает объект lambda (следовательно, объект типа func), который на вход принимает *x*, а возвращает значение f (g (x)). ВАЖНО! Функция compose не возвращает значение, она возвращает функцию, которая может принимать x на вход и возвращать значение функции композиции от этого икса. Поэтому, когда вы напишите что-то вроде h = compose(f, g), у вас в руках будет именно функция, после чего которой можно будет воспользоваться, передавая x.

Haskell

Теперь посмотрим как то же самое делается на языке Haskell.

```
1 compose f g x = f (g x)
```

Читается это следующим образом: объявляется функция с именем compose, которая принимает три аргумента: функцию *f*, функцию *g* и *Икс*, и compose вычисляется применением *Икса* к *g* и того что получилось к *f*. Как же это использовать? Предположим, что функции f и g - определены. Тогда можно объявить, что h = compose от f и g следующим образом:

```haskell
f :: Int → Int
f x = x * 2

g :: Int → Int
g x = x + 3

h :: Int → Int
h = compose f g

```

Преимущества и недостатки ФП

Рассмотрим преимущества и недостатки функциональной парадигмы.

Преимущества

Независящие друг от друга функции без побочных эффектов предоставляют некоторые преимущества вашему коду:

- Легко поддерживать и отлаживать
- Легко распараллеливать
- Исполнение программ - безопасно и предсказуемо

Функции высших порядков позволяют реализовывать:

- Высокий уровень абстракции

Распараллеливание:

- Высокая скорость исполнения программ (при правильном использовании сопоставимая с Си или даже быстрее)

А ещё - это красиво! Особенно, если вы любите впечатлять коллег чем-то умным и загадочным.

Недостатки

Основные недостатки функционального программирования:

- Ограничения на изменяемость данных. Бывает, что вам необходимо применение операций, которые требуют изменения состояния программы, но функциональный подход их запрещает и вместо изменений, объекты создаются заново с учетом изменений.

- Ограничения на производительность. Узкое место функционального программирования в объеме используемой памяти. Поскольку объекты не изменяются in-place, а создаются новые, то не исключены проблемы с производительностью при работе с большими объемами данных.

- Ограниченная поддержка инструментов. Большая часть ФП языков имеют меньшие сообщества, а значит и меньший объем инструментов и библиотек, по сравнению с языками императивного стиля программирования. Конечно, это в среднем по больнице, но на деле разница ощутимая.

- Сложность в понимании. Высокий порог входа для программистов не имеющих опыта в декларативном подходе. Это одновременно и недостаток и возможность научиться новым подходам. Но скорее недостаток, когда вам необходимо сделать что-то прямо здесь и сейчас при отсутствии такого опыта.

Итоги лекции

В сегодняшней лекции мы разобрали функциональное программирование, а именно:

- Контекст
- Что такое ФП
- Концепции
- История
- Примеры
- Преимущества и недостатки
- Итоги

Если у вас остались вопросы, вы можете написать их мне в Telegram: @alexlevinML

Спасибо что прослушали это лекцию и увидимся в следующей!

Что можно почитать еще?

1. Официальный сайт языка Haskell
2. Книга: Душкин - “14 занимательных эссе о языке Haskell и функциональном программировании”
3. Книга: Simon Thompson - Haskell: the Craft of Functional Programming
4. Видео-лекции по Лямбда-Исчислению и функциональному программированию
5. Статья про Lambda Calculus в энциклопедии Стэнфорда
6. Twitter - Effective Scala
7. Facebook - Fighting Spam with Haskell
8. Jane Street Technology
9. NASA, 1986 - A Survey of Functional Programming Language Rinciples
10. Сайт языка Miranda
11. Статья на Хабре “Почему вы должны думать о функциональном программировании”
12. Статья на Хабре “Почему разработчики влюбляются в функциональное программирование?”
13. Статья на Medium “Функциональное программирование — будущее компьютерной науки?”
14. Статья на GeekBrains “13 языков программирования, которые изменят будущее”
15. Статья на GeekBrains “Функциональное программирование”

Используемая литература

1. Haskell Preface
2. Статья на Хабре “λ-исчисление. Часть первая: история и теория”
3. Верещигин, Шень - “Начало теории множеств”