Лекция 1: Введение, основы ФП

19.02.2019

1. О чём этот курс

В этом семестре курс называется «Структуры и алгоритмы компьютерной обработки данных», но это довольно размытое название, а заниматься мы будем последней из популярных парадигм программирования, которая ещё не была покрыта на практиках — функциональным программированием. Речь пойдёт сначала про функциональное программирование вообще и функциональное программирование на примере языка F#, но потом мы затронем и другие темы, используя F# в качестве языка, такие как продвинутое многопоточное программирование, быть может, анализ данных и машинное обучение.

Про функциональное программирование будет рассказано немного теории (про нетипизированное лямбда-исчисление прежде всего, и про него будет всего одна пара, то есть меньше, чем про него обычно рассказывают в курсах про ФП, но у нас тут практика — очень рекомендую спецкурс про ФП на 4-м курсе, или, для любителей математики, курс от Дениса Николаевича Москвина https://www.lektorium.tv/course/22797, там про лямбда-исчисление более обстоятельно и курс реально очень годный; есть ещё годные курсы от Москвина https://stepik.org/course/75 и https://stepik.org/course/693, они про Хаскель, но какая разница?). Будет и много практики — как вообще на этом программировать (без состояний, следовательно, без переменных, циклов и прочих привычных штук), про функции высших порядков, каррирование и прочие подобные вещи.

Будет отдельная пара (и, может, не одна) про типы в функциональных языках и связанные с ними вещи: чем хорошо то, что нет состояния, для реализации таких структур данных, как списки, что даёт автоматический вывод типов — генерики, автоматическое обобщение типов функций и т.д. Будут и специфичные для функционального программирования паттерны и стили — point-free (стиль программирования, представляющий программу как композицию функций, с явным указанием аргументов только в самом конце), Continuation Passing Style (стиль записи последовательного вычисления как цепочки функций, каждая из которых делает свою работу и вызывает «продолжение»), монады (или, в F#, Computation Expressions или Workflows, способ управлять самим процессом вычисления, исполняя или не исполняя некоторые операторы в программе, выполняя какие-то действия между исполнением операторов, и т.д., при этом без дополнительного кода, if-ов и прочих вещей).

Будет и про технологические аспекты программирования на F#, и, на самом деле, на .NET вообще. В частности, F#, помимо того, что функциональный, ещё и объектно-ориентированный язык, со своими особенностями, о которых в этом курсе будет рассказано (кстати, совмещение объектно-ориентированного и функционального стиля даёт неко-

торые интересные эффекты, например, параметризация объектов функциями, что иногда может превратиться во что-то, очень похожее на полиморфизм, только гораздо гибче). Интересно то, что эти особенности постепенно переносятся в C# и иногда даже в Java (в Java 10 таки сделали локальный автовывод типов, например). Вообще, F#, по крайней мере, объектно-ориентированную его часть, можно рассматривать как C# из недалёкого будущего (даже primary constructors в C# уже вроде как реализованы и находятся в статусе experimental feature, в C# 8 сделали switch expressions, которые в F# назывались оператором match и были всегда). Так что знать F# полезно не только из любви к науке, но и чтобы лучше владеть C# (и немножко Java, Java слегка отстаёт от научно-технического прогресса, но есть Scala и Котлин — Scala в каком-то смысле можно считать “F# для Java-машины со странным синтаксисом”).

Будет немного продолжения про многопоточное программирование — не потому, что оно связано с функциональным, а просто потому, что про него обязательно надо знать, а программа обучения на матобесе устроена так, что можно счастливо избежать подробностей про него. Правда, функциональное и многопоточное программирование всё-таки хорошо уживаются вместе и дополняют друг друга: с одной стороны, чисто функциональные программы легко параллелятся, поскольку не имеют разделяемого состояния, с другой стороны, в функциональных языках есть методы управления процессом вычисления (те самые монады), которые позволяют писать многопоточные программы просто и красиво (что не отменяет возможности прострелить себе ногу, если не понимать, что происходит). В F#, в частности, есть монада *async*, которая работает так же, как ключевое слово *async* в C#, но не требует поддержки компилятора, это просто библиотечный класс.

2. Введение в функциональное программирование

Теперь, собственно, можно перейти к содержательной части пары. Сегодня речь пойдёт про функциональное программирование вообще, безотносительно F# (хотя про него, конечно, тоже немножко будет, потому что на F# уже надо будет сдавать домашку). Ну и начнём мы с главных отличий функционального программирования от того, как вы программировать привыкли.

В императивном программировании, как всем должно быть известно, программа представляет собой последовательность команд некоторому исполнителю (от непосредственно процессора, если программировать в машинных кодах, до некоторой абстрактной машины, если программировать на языках высокого уровня). Есть понятие «состояние», исполнитель может, выполняя команды, модифицировать это состояние, и вся программа представляет собой постепенную модификацию состояния из начального в конечное. Естественно, бывают бесконечно работающие программы, но сути дела это не меняет — есть некоторое начальное состояние и последовательность команд, которые в зависимости от текущего состояния могут оное состояние изменять.

Есть теорема Бёма-Якопини (https://en.wikipedia.org/wiki/Structured_program_theorem), доказанная аж в 1966 году, которая говорит о том, что любой алгоритм может быть представлен с помощью последовательного исполнения, оператора ветвления по булевому условию (обычного **if**) и цикла с булевым условием выхода (обычного **while**). Через два года после публикации работы с этой теоремой вышла знаменитая статья Э. Дейкстры «Go To Statement Considered Harmful», так появилось структурное программи-

рование. Состояние совершенно необходимо структурному программированию, потому что изменяющиеся значения переменных позволяют управлять процессом вычислений (те самые булевые условия в ветвлениях и циклах).

Но вообще, императивное программирование появилось гораздо раньше структурного, и само понятие «состояние» появилось благодаря машине Тьюринга (где, как вы, должно быть, помните, была бесконечная лента и исполнитель, который управлялся в том числе и символами, записанными на ленте, при этом он мог её модифицировать). Машина Тьюринга фактически оказалась реализованной в архитектуре фон Неймана — лента перестала быть бесконечной, схема работы с памятью стала посложнее, но принципы остались теми же. Есть ещё гарвардская архитектура, где программа и данные разделены, но она всё равно «предрасположена» к императивному стилю, разве что полиморфный код не позволяет писать, но его всё равно давно никто особо не пишет. Бывают и другие принципы, отличающиеся от машины Тьюринга, например, Dataflow architecture, где исполнение инструкций определяется не поведением вычислителя, а готовностью данных для инструкции, и, что интересно, современные процессоры исполняют инструкции именно по такой схеме, переупорядочивая инструкции в программе динамически, что доставляет такую боль при многопоточном программировании, но это внутреннее дело процессора, программы всё равно пишутся императивно. С компьютерами, работающими не по фон-неймановским принципам, так и не сложилось, и только относительно недавно стали популярны вычисления общего назначения на видеокартах, которые по сути представлят собой массив из сотен специализированных процессоров, имеющих каждый свою собственную и все вместе общую память, они уже достаточно сильно отличатся от классической фон-неймановской архитектуры. Долгое время ничего такого не было, поэтому императивные языки были столь популярны и, в силу большой инертности программистского и инженерного сообществ, остаются самым популярным подходом к программированию (объектно-ориентированное программирование, естественно, императивно по своей природе).

Функциональное программирование — это очередной радикальный поворот в представлениях о программировании, потому что оно базируется на λ-исчислении Чёрча и тем самым радикально отличается от императивного программирования и вообще от концепций фон Неймана и Тьюринга. По идее, это должно вызывать некоторые страдания, но, как показывает практика, основные концепции ФП усваиваются очень неплохо (видимо, потому, что оно проще и естественнее, чем ООП). Функциональная программа — это не более чем вычисление значения некоторой функции (в математическом смысле функции, то есть без побочных эффектов) на некоторых входных данных. В принципе, входные данные можно рассматривать как начальное состояние программы, результат — как конечное, но промежуточных состояний в функциональных программах нет. Вообще (ну, практически).

Если функциональные программы не оперируют состоянием вычислителя, из этого следуют забавные вещи: нет состояния — нет переменных, нет переменных — нет циклов, нет циклов и переменных — нет возможности задать поток управления или даже просто последовательность действий. Это могло бы быть очень печально, но раз состояния нет, то и последовательность вычислений не важна — результат зависит только от входных данных, в каком порядке вычисляются подвыражения в программе, не важно (на самом деле, не всё так просто, но про это на третьей паре). Забавный побочный эффект — у чисто функциональных программ сложно с взаимодействием с пользователем, ведь даже вывод на экран — это побочный эффект, а при условии отсутствия возможности явно управлять

потоком исполнения взаимодействие с пользователем может стать весьма хаотичным. К счастью, с применением некоторого количества высшей алгебры эта проблема решается, что доказывает, например, XMonad (оконный менеджер для Linux, написанный на Haskell, http://xmonad.org/).

Простой пример различия в стиле программирования, код на C++:

```
int factorial(int n) {
    int result = 1;
    for (int i = 1; i <= n; ++i) {
        result *= i;
    }
    return result;
}
```

И код на F#:

```
let rec factorial x =
    if x = 1 then 1 else x * factorial (x - 1)
```

Конечно, на C++ можно написать программу как на F#, но на функциональных языках можно писать программы только так, как в примере кода для F#-а. Никаких циклов, переменных, в которых накапливается значение, и прочих привычных штук. Однако видно, что if-ы есть (почему нет, это просто функция от трёх аргументов), рекурсия есть, возможность передавать параметры в функции есть. А это значит, что жить можно — переменные и вообще состояние можно эмулировать, передавая в функцию параметры и возвращая модифицированные параметры (на самом деле, в функциональных языках есть структуры, так что можно в качестве параметра передавать всякие сложные штуки, хоть всё состояние программы). Циклы, естественно, делаются с помощью рекурсии, в качестве счётчика цикла можно использовать один из параметров (как было продемонстрировано в факториале), последовательность вычислений эмулируется цепочками из рекурсивных вызовов функций.

Внимательный читатель, наверное, усомнится в достаточности рекурсии для любой программы — ведь размер стека вызовов конечен, так что рекурсивный факториал начнёт падать с переполнением стека уже при вычислении факториала от 10000, как же тогда, например, обходить списки из 10000 элементов и вообще делать то, что делалось в императивных языках обходом в цикле большой структуры данных. Интересно, что и программа из примера выше на самом деле упадёт с переполнением стека, но есть понятие «хвостовая рекурсия» — поддержка со стороны компилятора рекурсивного вызова функцией самой себя в самом конце, он такие штуки превращает в циклы. Зачем, как и как этим пользоваться, будет немного потом (на второй паре, кажется), сейчас важно знать, что есть простой приём программирования, который позволяет при рекурсивном вызове не использовать стек вообще, а вести себя так, будто это обычный цикл.

Вот ещё один небольшой пример, поясняющий суть дела:

```
let rec sumFirst3 ls acc i =
    if i = 3 then
         acc
    else
```

```
    sumFirst3
        (List.tail ls)
        (acc + ls.Head)
        (i + 1)
```

Это, как не трудно, наверное, догадаться, функция, складывающая первые три элемента в списке. Здесь, кстати, видно некоторые синтаксические особенности F#. Первое — это автоматический вывод типов: аннотаций типов в программе нет, тем не менее, тип каждой «переменной» (не переменной, а имени на самом деле) известен во время компиляции. Второе — рекурсивные функции объявляются с помощью ключевого слова rec, чтобы их имя было видно внутри их тела. Более по существу, здесь видно использование счётчика цикла (параметр *i*) и аккумулятора (параметр *acc*). Аккумулятор — это первый паттерн функционального программирования, с которым вам придётся познакомиться, он как раз и делает хвостовую рекурсию практически полезной.

Аккумулятор, по сути, это обычный параметр, который накапливает текущее посчитанное значение, чтобы по достижении базы рекурсии функция могла его просто вернуть. Можно было бы не использовать аккумулятор, а прибавлять ls.Head к результату рекурсивного вызова sumFirst3, но тогда **после** рекурсивного вызова пришлось бы делать операцию сложения, поэтому надо было бы хранить адрес инструкции, которую надо выполнять после рекурсивного вызова и значения параметров (рекурсивно вызвались от List.tail ls, но потом нам надо прибавить ls.Head, так что придётся хранить ls всё время), а для этого надо кадр стека вызовов. С аккумулятором дела обстоят иначе — сложение выполняется **до** рекурсивного вызова, так что рекурсивный вызов — это последняя операция в sumFirst3, так что при рекурсивном вызове достаточно просто присвоить новые значения параметрам и передать управление на начало функции, будто это цикл «while». Никакого кадра стека для хранения чего-либо не нужно. В этом и есть суть хвостовой рекурсии, и, как мы потом увидим, в байт-код она генерируется именно так. Плохая новость в том, что компилятор F# не умеет подсказывать, что рекурсия не хвостовая, хорошая новость в том, что хвостовая рекурсия довольно очевидна при некоторой привычке и будет получаться автоматически.

3. Зачем нужно функциональное программирование

Понятно, что большая часть интересных вещей в мире делается ради лулзов, но функциональные языки давно уже не относятся к эзотерическим. Хотя они пока больше распространены среди академического сообщества, многие из них активно используются и в индустрии, кроме того, идеи из функциональных языков «просачиваются» в привычные объектно-ориентированные. Например, лямбда-функции, одна из основ функциональных языков, появились в C#, C++, последней сдалась Java. В том числе поэтому четвёртый семестр проги посвящён функциональному программированию — может, вы никогда не будете программировать на F# (что вряд ли, он всё больше популярен), но на человека, который не в курсе, что такое лямбда-функции, замыкание, таплы, pattern matching и т.д., даже C#-программисты (да что там, даже Java-программисты, несмотря на отчаянную консервативность этого сообщества) будут смотреть как на идиота. Это всё можно было бы учить и на примере C#, но надо понимать, откуда это всё взялось, на чём основано и для чего на самом деле нужно. Интересно, кстати, что C# перенимает языковые механизмы из F#

настолько активно и бессовестно (благо их поддерживает одна компания), что F# можно считать чем-то вроде C# версии $n+3$, так что к программированию на F# можно относиться как к программированию на секретной превью-бета-версии C# из будущего. Поэтому, кстати, для этого курса выбран F#, а не Haskell — можно будет не только познакомиться с ФП, но и получить ряд практически полезных в индустриальном программировании навыков.

Так вот, преимущества функционального программирования таковы.

- Строгая математическая основа — функциональные языки базируются на λ-исчислении, которое само по себе строгая формальная система, при этом достаточно простая. Императивные языки тоже формализуются с помощью машин Тьюринга, но работать с ними мало удовольствия, в чём будет возможность убедиться в курсе матлогики и потом ещё в курсе про формальные языки.

- Семантика программ более естественна — для описания семантики, то есть строгого смысла императивных программ, используются, например, такие продвинутые математические механизмы, как абстрактные машины состояний, и то получается плохо. Например, семантика языка C (самого популярного по сей день в мире) так полностью и не описана, из-за чего случаются совершенно жгучие вещи (http://tmpaconf.org/images/pdf/2015/Nikolay-Shilov-A-Need-To-Specify-and-Verify-Standard-Functions.pdf). Семантика функциональных программ более очевидна.

  - Применима математическая интуиция — может быть, попрограммировав немного на F#, вы поймёте, зачем была нужна АТЧ. Это может отпугнуть, но вообще математикой человечество занимается несколько тысячелетий, так что возможность переиспользовать хоть в какой-то степени накопленные результаты при программировании и получать от этого какие-то преимущества кажется привлекательной.

- Программы проще для анализа в силу более простой семантики и меньшего количества зависимостей между их частями.

  - Автоматический вывод типов.
  - Широкие возможности для оптимизации — чем лучше компилятор понимает программу, тем проще ему оптимально транслировать её в машинный код. Например, возможные нетривиальные зависимости по данным сильно мешают при оптимизации императивных программ, в функциональных программах таких проблем нет.

- Оно более декларативно — мы пишем в большой степени не как сделать, а что сделать. Это несколько упрощает жизнь программиста, позволяя не думать о деталях реализации, и жизнь компилятора, предоставляя ему пространство для манёвра.

  - Ленивость — если значение выражения так и не понадобится, его лучше не считать, поэтому давайте вообще не считать значение выражения, пока нас явно не попросили. Некоторые параметры функции, например, могут так и не посчитаться, даже когда функция уже отработала (это так в Haskell, в F# из коробки

всё считается как обычно, но можно попросить язык в конкретном случае считать лениво).

  - Распараллеливание — про это уже не раз говорилось, но ещё раз обращаю внимание — есть программа без побочных эффектов и зависимостей по данным, состоящая большей частью из большого количества вычислений не зависящих друг от друга функций, есть видеокарты, на которых несколько сотен довольно быстрых процессоров и несколько гигов оперативки. Не так сложно сопоставить эти факты. Из коробки ничего интересного не происходит, но есть кое-что (например, https://github.com/gsvgit/Brahma.FSharp), что может помочь.

- Модульность и переиспользуемость — в объектно-ориентированных программах переиспользуемость достигается обычно на уровне классов, в функциональных — на уровне функций и объединений функций —- модулей. Причём, поскольку нет состояния, нет и зависимостей по данным, так что функция может быть вызвана кем угодно и когда угодно, при этом заведомо ничего не сломается. В ООП может быть невозможно использовать класс в юнит-тестах, потому что в конструкторе он запускает ядерную ракету, в чистом ФП функция не может иметь побочных эффектов, поэтому можно её использовать хоть где и как. F#, правда, не чисто функциональный язык, так что там скорее как в ООП, но всё равно, более «мелкозернистое» переиспользование в совокупности с функциями высших порядков и разными функциональными паттернами позволяют писать кода меньше, а функциональности получать больше.

- Программы более выразительны. Прежде всего благодаря более развитой системе типов, чем в объектно-ориентированных языках и функциям высших порядков, и вообще отношению к функциям как к обычным данным. Можно писать очень красивый, короткий и переиспользуемый код.

Утверждение насчёт выразительности можно подтвердить примером. Берём наш старый пример со сложением первых трёх элементов списка:

```
let rec sumFirst3 ls acc i =
    if i = 3 then
         acc
    else
        sumFirst3
            (List.tail ls)
            (acc + ls.Head)
            (i + 1)
```

Переписываем его с использованием функции fold (на C# мы такое писали, если кто не помнит, освежите знания по википедии):

```
let sumFirst3 ls =
    Seq.fold
        (fun x acc -> acc + x)
        0
        (Seq.take 3 ls)
```

Записываем это же самое с помощью forward pipeline operator:

```
let sumFirst3 ls = ls |> Seq.take 3 |> Seq.fold (+) 0
```

Оный forward pipeline operator на самом деле не более чем применяет функцию к аргументу, позволяя записывать программу в виде этакого конвейера:

```
let (|>) x f = f x
```

А потом вспоминаем, что мы же на матмехе, и переписываем всю программу как композицию функций:

```
let sumFirst3 = Seq.take 3 >> Seq.fold (+) 0
```

Теперь у нас есть функция *sumFirst3*, которую можно применить к списку, при этом в её определении сам список вообще не упоминается. А зачем, ведь >> — это просто функция, принимающая две функции и возвращающая третью, ей не нужно ничего знать про аргументы. Такой стиль программирования, без явного указания аргументов функций, называется, кстати, point-free (point — это точка, в которой считается значение функции, терминология в ФП в основном пришла из математики, point-free — это «без явного указания аргумента»). Про point-free и вообще про то, как работает кусок кода выше (например, что такое «каррирование»), у нас будет дальше в курсе, но вот так на F# можно писать (это не значит, что нужно, кстати, программы в point-free-стиле быстро становятся нечитаемыми).

Вот ещё небольшой пример, возвести в квадрат и сложить все чётные числа в списке:

```
let calculate =
    Seq.filter (fun x -> x % 2 = 0)
    >> Seq.map (fun x -> x * x)
    >> Seq.reduce (+)
```

Принцип действия такой же, point-free и всё такое, но посмотрите, как естественно выглядит программа — применили фильтр, оставив только чётные числа, возвели всё, что получилось, в квадрат, выполнили свёртку с операцией «+», сложив всё, что получилось. Это не какие-то убогие циклы, как, например, в Java до появления stream-ов.

Встаёт вопрос, если всё так хорошо, то почему ещё не все пишут на функциональных языках. Причины довольно просты. Во-первых, функциональные программы не очень подходят для таких ненужных вещей, как взаимодействие с внешним миром. Это, конечно, обходится, но требует некоторых усилий и понимания, что неудобно. Во-вторых, функциональные программы, в силу большей декларативности, менее предсказуемы в плане скорости работы и потребляемой памяти. Пример проявления такой проблемы вы уже видели — берём функцию с хвостовой рекурсией, дописываем, скажем, «+1» в конце, программа внезапно начинает падать или потреблять неадекватное количество ресурсов. В-третьих, что очевидно связано с «во-вторых», функциональное программирование требует от программиста некоторой математической подготовки и желания думать, что хорошо в теории (программисты вообще склонны к честолюбию), но когда завтра релиз и ты осознаёшь, что всю ночь потратил, пытаясь наиболее изощрённо записать какой-нибудь кусок кода в point-free стиле так, чтобы компилятор не ругался, это печально. К тому же, у большинства программистов с математической подготовкой как раз дела не очень, не все же на матобесе учатся — у ПИшек вот даже матфизики не будет, например.

На этом пока всё, дальше будет более конкретно про F# и как на нём программировать (хотя общие принципы применимы ко всем функциональным языкам), но домашку уже надо делать на F#-е.