Кредо программиста (в приложении к стилю кодирования)

Однажды меня попросили подготовить рекомендации по программированию на Scala для сотрудников нашей компании Solar. В поисках лучших рецептов приходилось заглядывать в самые пыльные уголки Интернета, и в одном из них, в старом проржавелом сундуке я заметил старинный конверт. Просто из любопытства решил заглянуть туда, благо конверт был уже вскрыт. Ветхие страницы, поблекшие чернила, но содержание настолько меня заворожило, что я просто не мог не поделиться со всеми!

К сожалению, не удалось установить ни автора, ни адресата. Местами текст совершенно невозможно было прочитать, так что пришлось додумывать самому. Но даже несмотря на потери, письмо всё ещё достойно Вашего внимания.

> Приятного чтения!

Для удобства разбил письмо на озаглавленные части, а свои комментарии оставил в [таких скобочках].

Содержание

Кредо

Приветствую тебя, дорогой друг!

Приятно было получить твоё письмо и узнать, что у вас… [Фрагмент утрачен]

В письме ты интересуешься кредо, чем оно способно помочь в достижении наших целей. Я расскажу тебе всё, что знаю сам.

> **Ничто не истинно, всё дозволено.**

Это может показаться циничной доктриной. Но *кредо* — всего лишь отражение природных вещей. Тезис «Ничто не истинно» подразумевает, что основы, на которых держится современная разработка, зыбки, и мы сами должны строить свои программы. Говоря «Всё дозволено», мы подразумеваем, что сами решаем, что нам делать, и несём ответственность за последствия, какими бы они ни были.

Кредо — это то что необходимо держать в голове и вспоминать, когда необходимо принимать какое-либо решение. В былые времена во время обряда посвящения в программисты произносилась такая формула:

> — Когда остальные слепо следуют за популярными парадигмами, помни…
> — Ничто не истинно…
> — Когда остальные ограничены набором паттернов и «лучших практик», помни…
> — Всё дозволено…

В каждый момент мы полагаемся только на собственное чутье и то, что нам известно в данный момент. Завтра ты можешь внезапно осознать, что ошибся вчера, но это нормально! Такое открытие обязательно пригодится тебе в будущем. Принимая неудачные решения бессмысленно оправдываться чужим опытом, авторитетными мнениями, или просто привычкой. Ты несёшь персональную ответственность за все сделанные выборы и должен уметь обосновывать их каждому, а прежде всего, самому себе.

Ничто не истинно

В нашем цеху мы занимается автоматизацией различных процессов, что даёт людям возможность свободнее распоряжаться своим временем. Наше с тобой призвание — нести людям свободу.

Но нужно помнить, что сама наша работа отнимает время у всех, кто к ней причастен. Значит и здесь нам нужно обеспечивать максимальную свободу как товарищам по цеху, так и самим себе. В частности, не стоит вводить необоснованные ограничения на написание программного кода.

Одним и примеров таких ограничений можно считать паттерны программирования. Среди них, конечно, встречаются полезные примеры, с которыми стоит познакомится каждом неофиту, но вот их навязывание, постоянное упоминание в книгах, публичных обсуждениях, приводит к тому, что в общественном сознании закрепляется ошибочное мнение, будто бы знание паттернов необходимо для качественного программирования! Будь осторожен — это всё происки [Abstergo? Ubisoft? Невозможно разобрать].

На самом деле частые мысли о паттернах приводит к формированию шаблонного мышления, развитию эффекта «туннельного зрения». Вместо прямого решения поставленной задачи, программист будет стараться натянуть её на полюбившийся ему паттерн. Это явление может показаться даже забавным, когда сталкиваешься с ним впервые, но такая беда встречается повсеместно! И если у опытных программистов получаются просто многословные и хрупкие

решения, то начинающие способны сходу снести «правильными», но неуместными паттернами все заслоны тестирования.

Также в некоторых командах принято жёстко фиксировать правила форматирования кода — как расставлять скобки, отступы, табуляции внутри строки и т.п. Или хуже, настраивают в среде разработке механизмы автоматического форматирования. Оправдывается это зачастую тем, что строгие правила призваны сэкономить команде время на споры о «красоте кода», предотвратить возможные конфликты. Но так ли это ценно? Ведь, как известно, в споре рождается истина, а с «конфликтностью» стоит бороться путём повышения культуры общения! Встречал и такое, что опытные программисты теряют способность воспринимать чужой код, не укладывающийся в привычный формат.

То же самое касается и утверждения какого-либо конкретного стиля кодирования, что особенно критично для языка программирования Scala, допускающего множество разных стилей. Подобные ограничения не только приводят к невыразительному коду, что само по себе косвенно снижает качество программы, но, что гораздо хуже, ограничивают развитие навыков программирования.

Однако, не все «истины» навязываются извне. Нередко случается, что программист сам ограничивает себя выработанными привычками. И это не всегда плохо — полезные привычки бывают действительно… полезны! Вот только эту самую «полезность» стоит регулярно перепроверять, не потерялась ли она в связи с обретением новых знаний и навыков. А такая привычка, к сожалению, встречается не у каждого программиста.

Возможно ты уже слышал про «принцип наименьшего удивления». Обычно имеется ввиду прежде всего «неприятное» удивление, вызывающее вопросы типа «а зачем тут это вообще?». Но название принципа само по себе просто ужасно! Оно намекает, что удивление — это что-то плохое! Как раз именно из этого пытаются вывести ценность знаний о паттернах, строгих правилах форматирования кода и т.п…

Удивление — это позитивное понятие, это когда ты открываешь для себя что-то новое и хорошее. Когда же человек теряет способность удивляться, значит он стареет душой. Что же до программирования, то не нужно ставить себе целью удивить читателя, не и не нужно боятся чем-то удивить. Качество кода вообще никак не связано с чьим-то «удивлением».

Перечисленные мною ограничения не оставляют места творческой составляющей, отнимают необходимость самостоятельно видеть, оценивать и выбирать лучше решения. Лишенный свободы выбора человек деградирует, теряет себя. Мы не должны этого допускать.

Всё дозволено

Если ничто не истинно, чем же тогда руководствоваться в принятии решений? Конечно же, в любом случае нужно выполнить техническое задание, удовлетворить всем функциональным и нефункциональным требованиям. Поэтому я расскажу только об оформлении этих решений — о стиле, формате и прочих мелочах.

Программируя, необходимо по максимуму использовать все знания имеющиеся на данный момент. Но нужно пропускать эти знания через себя, каждый раз анализировать, на сколько они применимы для решения текущей задачи. Возможно уже завтра (после ревью, или здорового сна) ты убедишься, что можно было бы сделать лучше, но сейчас — это не важно.

Объективные критерии качества кода определяет многовековой опыт человечества. Но эти критерии не высечены на каменных скрижалях — они ежедневно меняются, подстраиваясь под всё новые знания. Причём в каждый момент времени сосуществуют несколько взаимоисключающих подходов. И всё же среди них можно найти общие черты.

Тексты наших программ имеют двойное назначение — их должны понимать как компьютер, так и люди. Значит мы должны, используя возможности языка программирования, создать свой язык так, чтобы описанные на нём модели данных и логика работы с ними были наиболее понятны всем людям, кто будет в дальнейшем работать с этим кодом, в том числе и самому себе. Код удовлетворяющий такому требованию иногда называют «самодокументирующимся».

Кроме того, нужно учесть, что наш код будет жить и развиваться, и нам, помимо прочего, необходимо упростить и обезопасить внесение любых изменений. Это очень важный аспект, специфичный именно для программирования. Но здесь также можно нащупать более или менее объективные критерии. А поможет нам в написании устойчивого к изменениям кода ~~тестирование~~ [кажется, автор письма сейчас не об этом] компилятор и/или инструменты подсветки синтаксиса, проверяющие код на соответствие заявленным типам.

Под разными аббревиатурами [DRY, KISS, YAGNI, SOLID и т.п.] в народе известны некоторые принципы, помогающий при написании качественного кода. Не все они одинаково полезны [видимо, автор имеет в виду элементы SOLID], по познакомиться с ними однозначно стоит.

Производительность кода ни в коем случае не должна ставится выше его качества, если только это не прописано явно в требованиях к программе. Качественный код, наиболее просто и лаконично выражающий решение поставленной задачи и так будет близок к оптимальному. Но только лишь повышение производительности не может оправдывать, например, предпочтение `if` вместо `Option.when`.

Чтобы писать понятный код, нужно [всего-то] уметь выражать свои мысли. Как говорил мой учитель,

> Если ты не можешь что-то объяснить другому человеку, значит ты просто это не понимаешь.

В достижении понятности кода нам помогает многовековой литературный опыт человечества.

Литературный стиль

Выразительные определения

Прежде всего, программистам полезно научиться точно формулировать понятия предметной области, которые он использует в своих алгоритмах. Идентификаторы типов, сущностей,

отношений должны легко читаться и однозначно указывать на свою семантику. Недопустимо предлагать читателям текст, составленный из безликих слов вида `Item`, `do`, или вообще из бессмысленных наборов букв — `i`, `foo`, и т.п…

Так уж повелось, что термины мы формируем из слов [какого-то «мёртвого языка»; латынь?], разделяяИхЗаглавнымиБуквами. Поэтому полезно хоть немного знать лексику и грамматику этого языка, чтобы не написать `styleForm` вместо `formStyle`, или `forSeal` вместо `forSale`. Ну и всегда можно воспользоваться услугами толмачей.

К идентификаторам для удобства можно дописывать необязательные окончания. Например, если в переменной храниться физическая величина, то хорошо бы подчеркнуть это в имени переменной. В итоге переменная для хранения количества секунд паузы между повторениями может быть объявлена так: `val repeatingDelaySecOpt: Option[Int]`.

Не всегда удаётся выразить смысл сущности в коротком идентификаторе, но это не страшно. Всё равно гораздо лучше читать какой-нибудь `selectedElephantsForAlpinCampaign`, нежели «ленивое» `yoprst`.

Для программ на нашем языке Scala характерно ещё одно очень логичное соглашение об именовании — идентификаторы из вселенной значений принято начинать с маленькой буквы, тогда как во вселенной типов всё начинается с заглавной: `value: Type`. Впрочем, из этого правила также есть исключения, например, объекты-компаньоны типов, которые суть значения, именуются в точности также, как и их типы, с заглавной буквы.

Секреты скорочтения

В современных инструментах редактирования и просмотра кода длина строк практически не ограничена. Всю программу можно написать хоть в одну строчку. Очевидно, что навигация по такому коду с помощью горизонтальной прокрутки экрана будет весьма утомительной. Обычно длину строк рекомендуют ограничивать восьмьюдесятью символами, потому что считается, что такие строки будут видны целиком на большинстве экранов, следовательно, при работе с кодом горизонтальная прокрутка не потребуется. Рекомендация полезная, но (ничто не истинно, всё дозволено) из любых правил бывают исключения, и жёсткое ограничение длины строки периодически часто будет приводить наоборот к некрасивому коду.

Впрочем, о длине строки есть и другие соображения. Во всём мире при верстке газет и журналов текст размещается в колонках, строки которых обычно не длиннее сорока символов. Уже давно было замечено, что такие строку читатель способен воспринимать целиком, а не как последовательность отдельных символов, или даже слов. В этом заключается секрет скорочтения — взгляд ходит не слева на право, а практически только сверху вниз от строчки к строчке.

Эту идею очень полезно иметь ввиду при написании программного кода. «Вертикальное чтение» могут облегчить такие факторы:

- короткие строки — жёстких ограничений нет, всегда нужно полагаться на свой глазомер;

- текст выровнен по левому краю;
- простой синтаксис в каждой строке;
- каждая строка по возможности имеет собственный законченный смысл;
- смыслы подряд идущих строк достаточно однородны, чтобы легко читалась суть всей последовательности.

И, наоборот, чтению мешают:

- длинные строки;
- навязываемые различной автоматизацией отступы, как, например, для списка аргументов методов в Scala,
- отступы для вложенных конструкций — «ёлочки» (лежащей на боку);
- множество знаков препинания, сложные синтаксические конструкции;
- даже короткие строки могут быть перенасыщены смыслом;
- «бессмысленные» строки, например, содержащие только скобки, или аргументы λ -выражений;
- частая смена смыслов, контекстов разных строчек, когда рвется нить повествования.

Опять же, для любых правил существуют исключения. Иногда (реже чем кажется!) действительно бывает полезно отвлечь внимание читателя строкой с отступом, или отдельной скобкой. В то же время, события для журналирования, или сообщения об ошибках могут не укладываться по длине ни в какие границы. Но такие строки мало кому интересны, и насильственно разрывать их на несколько строк, только чтобы уложиться в жёсткие рамки какого-то формата — это издевательство над большинством читателей!

Простые предложения

Как говорил один наш великий предшественник

> Всё должно быть изложено настолько просто, на сколько это возможно. Но не проще.

В классических художественных произведениях можно встретить громоздкие предложения, которые даже не всегда умещаются на одной странице. Не всякий читатель удержит в голове всё хитросплетение вложенных сложноподчинённых предложений и поймёт основную мысль, которую хотел передать автор. Это достаточно стандартный художественный приём, применяемый осознанно, и его использование совершенно уместно в некоторых художественных произведениях.

Но навряд ли кто-то найдёт оправдание этого приёма в продуктовой разработке. Наша задача состоит не в том, чтобы запутать читателя, но, наоборот, как можно короче и понятнее передать ему суть алгоритма. Тут нужны короткие, но ёмкие фразы из всего нескольких терминов. Сами же эти термины раскрываются уже в других «предложениях». Нужно стремиться избегать как перечисления большого количества операций, так и избыточной вложенности, которая обычно выглядит как «ёлочка».

Один из самых лучших способов достижения простоты является декомпозиция. Достаточно разбить сложный алгоритм на небольшие предложения, имеющие самостоятельный смысл, назначить им подходящие по смыслу идентификаторы, и тогда исходная программа представляется в виде простой комбинации вызовов всего нескольких подпрограмм! Конечно, такой приём требует некоторых усилий, но ведь это и есть наша работа — упрощать жизнь всем, кому достанутся результаты наших трудов.

А иногда встречается код, в котором вложенность удаётся устранить даже не прибегая к декомпозиции. Типичный пример — выполнение некоторых действий, только в случае, если выполняется некоторое условие. Частенько встречаются функции с телом вида

```
if (условие) {
  `Большое,
  очень большое
  описание
  счастливого пути`
} else
  `однострочный несчастливый вариант`
```

В этом случае вложенность совершенно не обоснована. Её можно устранить, применив технику раннего выхода:

```
if (!условие)
  return `однострочный несчастливый вариант`

`Большое,
очень большое
описание
счастливого пути`
```

В последнем фрагменте кода показан пример разбиения текста на абзацы — последовательности предложений с законченным смыслом отделяются друг от друга пустой строкой (изредка даже несколькими).

Также существует практика, когда в теле функции описываются вложенные функции. Инкапсуляция зависимостей в месте их использования обеспечивает «сильную связность» целостного фрагмента кода, делает его более «модульным». При переносе большой функции в другое место будет меньше шансов что-то упустить — такое решение защищает от ошибок рефакторинга. Но вот читать такой код значительно сложнее, так как основная суть метода теряется где-то за описаниями вложенных функций. А модульность лучше получать, например, сразу размещая сильно связанный код в отдельных файлах.

Конечно же, вложенные функции не всегда бывают неудобными. Например, если целевая и вложенная в неё функции достаточно маленькие, то основная суть скорее наоборот будет выделяться яснее.

Знание языка и диалектов

Сложность кода иногда может быть обусловлена несоответствием используемых инструментов актуальным версиям языка и библиотек. Очень важно не только регулярно поднимать версии зависимостей, но проверять, что же там изменилось. Возможно, там были закрыты некоторые уязвимости, или появились более простые инструменты, позволяющие упростить код и тем самым повысить его читаемость и устойчивость к правкам.

Твоя среда разработки может даже иногда посоветовать как заменить существующий код на более простой и, как правило, эффективный. Например, вместо `.filter(cond).headOption` удобнее писать `.find(cond)`. Обычно это действительно полезные советы, но, как и всегда, решения нужно принимать самостоятельно.

Но, например, при работе с библиотекой Cats среда разработки уже не подскажет, что вместо `.map(x => x -> x.id)` точнее будет `.fproduct(_.id)`, а вместо `.map(f).sequence.map(_.flatten)` лучше писать `.flatTraverse(f)`. Поэтому полезно самостоятельно изучать возможности используемых библиотек [диалектов, то бишь DSL] и каждый раз задумываться, какой инструмент будет сподручнее.

Чтобы находить наиболее удобные инструменты полезно понимать некоторые математические основы программирования. Различные Scala-библиотеки предоставляют свои контейнерные типы для работы с эффектами, но в них есть много общего, так как все они основаны на фундаментальных понятиях теории категорий. Полезно ознакомиться хотя бы элементами этой теории — что такое функторы, естественные преобразования и основанные на них монады, сопряжённые и аппликативные функторы, профункторы и стрелки т.п. Тогда, зная что искать, в любой библиотеке ты сможешь найти наиболее подходящие возможности контейнерных типов.

Избегаем повторов

Не могу припомнить книг, в которых мне встретились повторяющиеся фрагменты текста. Читатель, встречая повторяющиеся фрагменты текста задастся вопросами: «Зачем я потратил время на то, что уже прочёл ранее? Чего добивался автор?». Издательства такие сочинения стараются не тиражировать.

А вот в программном коде повторяющиеся фрагменты текста, к сожалению, часто проходят заслоны ревью. Причина их появления в том, что бездумное копирование кода немного проще, чем переиспользование, или декомпозиция (например, вынос этого фрагмента в отдельный метод). Да и распознать такое мошенничество не так-то просто. Ведь в отличие от книг, текст программы, разбитый на блоки (методы, классы и т.п.), не подразумевает, что его будут читать «подряд», наталкиваясь на повторяющиеся фрагменты текста. Наоборот, расследуя сбои, или внося правки зачастую сложно заметить такие повторы.

Как раз в этом и прячется наибольшая опасность — ты может обмануться, посчитав важным лишь один фрагмент кода, но не заметив его копии. Правки такой логики могут привести к одновременному существованию разных её версий, дающих непредсказуемое поведение в зависимости от контекста обращения к ней. Дублирование кода принесло слишком много боли человечеству… Но даже если ты потратишь время на поиски всех копии, придётся либо

вносить правки в каждой, либо (правильный выбор!) самому устранить дублирование, с которым почему-то не справился предыдущий автор.

Устраняется дублирование кода знакомым способом — вводится новый идентификатор, за которым скрывается повторяющийся фрагмент кода, а все копии заменяются на обращение к этому идентификатору.

> Но получается, что теперь сам идентификатор постоянно повторяется. Можно ли избежать и этого? Один из способов защиты от такого дублирования дают **линейные типы**. Идентификатор значения такого типа может встретиться в тексте программы лишь дважды — в момент определения (присвоения) и в момент единственного использования. Линейные типы могут принести много пользы, но работа с ними требует определённых навыков, ещё несвойственных современной культуре программирования. Да и реализованы они пока лишь в нескольких не самых популярных языках программирования.

И всё же, иногда повторение кода может быть оправдано. Например, это могут быть части совершенно независимой логики, и лишь на данном этапе эти части случайно оказались одинаковыми, но известно, что в дальнейшем их пути развития разойдутся, они перестанут быть похожими. Такие рассуждения применимы не только к логике, но и моделям данных — описание интерфейсов разных приложений может частично совпадать, но нет причин устранять это дублирование.

При сопоставлении с шаблонами какого либо типа-суммы часто встречается шаблонный код вида

```
case MyAlgebraicType.Constructor1(_, _) => ???
case MyAlgebraicType.Constructor2(_)    => ???
case MyAlgebraicType.Constructor3       => ???
...
```

В большинстве подобных случаях будет удобнее заранее импортировать все члены типа-суммы:

```
import MyAlgebraicType.*
```

Аналогично, при работе с большими типами-произведений (например, при конвертировании DTO), чтобы не повторять перед каждым полем `entityDto.` можно импортировать все члены сущности:

```
def convertToDomain(entityDto: MyEntityDTO) = {
  import entityDto.*
  MyDomainEntity(
    dtoField2,
    dtoField3,
    dtoField1,
    ...
```

```
  )
}
```

Ещё одна разновидность повторений связана с частой привычкой расставлять везде аннотации типов. Следующий код прямо-таки тяжело читать:

```
val elephant: Elephant = Elephant()
```

Не нужно подражать Капитану О[чего-то там]. Ошибочно считать, что повсеместная расстановка типов способствует лучшему пониманию кода. Типы нужны компилятору для проверки корректности кода, но при чтении кода гораздо важнее точные названия переменных и функций. Проставлять типы полезно лишь для тех возможностей, которые действительно предназначены для стороннего использования:

```
trait MyService[F[_]]:
  def method: (SomeData, SomeAnotherData) => F[SomeResult] // только типы

object MyService:
  def apply = MyService[IO]:
    val method = (someData, someAnotherData) => ???        // только значения
```

Подобных аннотаций должно быть достаточно компилятору, чтобы проверить всю программу.

У автоматических средств проверки программы иногда возникает каприз, когда они требует либо явно проставлять типы у публичных членов, либо приписывать к ним модификатор `private`. Оба варианта приводят к необоснованному повторению кода. Про аннотирование интерфейсов я только что рассказал. В свою очередь, мания «прятать изменчивые потроха классов» берёт своё начало из объектно-ориентированного программирования и практически неактуальна в Scala. Чтобы избавить код необходимости повторять аннотации типов, или ООП-шный `private`, я рекомендую тебе найти способ отключить такую проверку насовсем.

Помимо ситуаций полного дублирования кода, может встречаться также множество фрагментов кода, организованных по определённому шаблону. Обычно такое встречается при обработке громоздких структур данных в пределах одного метода, поэтому сразу бросается в глаза при чтении. Шаблонный код чреват теми же проблемами, что и повторяющийся, поэтому когда сталкиваешься с таким кодом можно попробовать следующее:

- поискать возможность устранить повторяемость;
- собрать шаблонный код в одном месте, строчка за строчкой;
- по возможности, отформатировать шаблонный код, расставив пробелы, чтобы получилось некое подобие таблицы.

Вот пример табулированного шаблонного кода:

```
ifaceName             = ifname,
area                  = area,
```

```
authentication        = fields("authentication")     .headOption.flatten,
simpleKey             = fields("authentication-key") .headOption.flatten,
helloIntervalSec      = fields("hello-interval")
.headOption.flatten.flatMap(_.toIntOption),
deadIntervalSec       = fields("dead-interval")
.headOption.flatten.flatMap(_.toIntOption),
retransmitIntervalSec = fields("retransmit-
interval").headOption.flatten.flatMap(_.toIntOption),
cost                  = fields("cost")
.headOption.flatten.flatMap(_.toIntOption),
priority              = fields("priority")
.headOption.flatten.flatMap(_.toIntOption),
passiveAddress        = fields("passive")
.headOption.pipe(ThreeState.fromOptOpt),
md5Keys               = fields("message-digest-key") .flatten.pipe(parseMd5Keys),
```

Выгода от такого форматирования в том, что элементы шаблона, будучи единожды прочитаны, больше не отнимают внимания читателя, а вот любое нарушение шаблонности сразу заметно. Табулирование даёт некую предсказуемость при чтении шаблонного кода, и в этом смысле его можно сравнить с рифмой в поэзии.

Знаки запинания

Язык Scala по синтаксису схож со многими популярных языков программирования: аргументы функций перечисляются через запятую в круглых скобках, блоки кода заключаются в фигурные скобки, вызовы метода отделяется от объекта точкой и т.п. В отличие от легко читаемых литературных текстов, программный код зачастую бывает перегружен целым ворохом подобных знаков препинания, что, конечно же, затрудняет чтение. К счастью, во многих случаях Scala позволяет облегчить код, смахнув ненужные значки.

Если реализация функции укладывается в единственное выражение то его вовсе не обязательно обрамлять фигурными скобками. Если же у тебя возникает желание отметить закрывающейся скобкой тело функции со слишком большим выражением, то практически наверняка тебе стоит задуматься о декомпозиции. Также в третьей версии Scala поддерживается новый синтаксис, когда лишние фигурные скобки не нужно писать в `if`, `for`, сопоставлениях с шаблонами и даже сложных блоках кода, состоящих из нескольких операций. А если всё-таки сочтёшь нужным, то ты всегда можешь преднамеренно «закрыть скобку», с помощью нового ключевого слова `end`.

Кортежи в Scala записываются как перечисленные через запятую значения в квадратных скобках `(a, b, c, d)`. Но в то же время для наиболее часто встречающихся кортежей-пар есть и другая форма записи: `key -> value`. Если действительно подразумевается сопоставление «ключ-значение», то лучше использовать именно такую форму записи. Она не только наглядна сама по себе, но и позволяет убрать лишнюю пару скобок, что особенно важно в ситуациях, когда скобки идут подряд.

Для ООП привычен такой синтаксис вызова метода: `object.do(something)`. Тут просматривается стандартная форма предложения: «подлежащее сказуемое дополнение», но мешают лишние знаки препинания. С точки зрения естественного языка гораздо удобнее читается `object do something`. И Scala поддерживает этот синтаксис, но только если метод принимает только один аргумент. Такое ограничение нацелено на то, чтобы было удобнее объявлять инфиксные операторы, вроде `zip`, `andThen` (примеры: `list1 zip list2`, `funct1 andThen funct2`). Чтобы ты мог так же вызывать собственные методы в последней версии Scala, нужно либо при вызове заключить имя метода в апострофы `object `do` something` (что всё равно читается лучше), либо явно указать такую возможность в объявлении метода `infix def do(...)`.

Снова замечу, что приведённые мной примеры не навязывают строгие правила, но раскрывают возможности, которые нужно учитывать, когда мы стремимся сделать свой код лучше.

Функциональное программирование

Что такое ФП

В наше время [не то, что сейчас!] всё ещё в моде императивный стиль программирования, в котором пишутся инструкции для исполнителя, последовательность приказов (императивов), таких как «пойди сюда», «возьми то», «отправь туда», «получи оттуда», «положи тут», «вернись обратно» и т.п. Так сложилось испокон веков, когда в ранних языках программирования были только такие команды, ориентированные на общепринятую архитектуру вычислительной машины. Языки развивались, но большинство из них по прежнему предлагали в основном императивные конструкции.

В то же время ещё до появления ЭВМ программисты знали, что чтобы написать хорошую программу достаточно всего двух основных языковых конструкций — объявление функции и её применение к значению [см. λ-исчисление]. Синтаксис такого языка максимально прост и корректность алгоритмов на нём удобно проверять автоматическими средствами ещё до исполнения!

Функции сами могут выступать в роли значений [функции, как объекты первого класса] — их можно передавать как аргументы в другие функции, и получать их в результате вычислений. Ввиду ориентированности на функции, такой подход называют функциональным программированием [ФП].

Программа строится не как последовательность приказов недотёпе-компьютеру, а как декларация ожидаемого решения: «нужен результат выполнения `функции1`, которая принимает на вход результат `функции2`, получающей на вход сумму первоначальных аргументов, передаваемых нашей программе». Поэтому функциональное программирование часто называют декларативным. Оно подразумевает, что компьютер сам поймёт, что от него ожидают, и составит последовательность низкоуровневых команд, приводящих к нужному ответу.

В функциональном программировании нет необходимости в императивных конструкциях, вроде `if`, `match`, `return`, `for`, ~~`goto`~~ и проч. В этом смысле все они являются усложнением синтаксиса, причём, зачастую, весьма существенным [не смотря на то, что к нему все привыкли]. Мы же стремимся избегать излишних сложностей в коде, делать код максимально простым.

В языке Scala во многих ситуациях вместо императивных операторов можно подобрать функциональные альтернативы:

- вместо `if` можно использовать использовать встроенные `Option.when`, `Either.cond`, или всякие `Applicative.whenA`, `FlatMap.ifM` и проч. из библиотеки Cats;
- сопоставление с шаблонами часто применяется для контейнерных типов, у которых есть встроенные возможности, вроде `fold`, `map` и т.п;
- ранний выход из функции [взамен `if (!cond) return`] можно организовать разными способами, например с использованием `Option.traverse`;
- вместо `for`-выражений лучше [см. далее] использовать цепочки вызовов всяких `.map`, `.flatMap` и прочих возможностей контейнерных типов.

Перечислю некоторые характерные черты функционального стиля, влияющие на качество. Лаконичность и выразительность кода (следовательно, и устойчивость к правкам) обеспечивают

- декларативные выражения, вместо императивных утверждений;
- функции, как значения, функции высокого рода.
  Надёжность кода (и, опять же, устойчивость) дают
- неизменяемые типы данных;
- чистые функции без побочных эффектов (по возможности);
- чистое обслуживание эффектов.

[Также см. What is Functional Programming?]

Но вспомним наше кредо! Во-первых, возможностей стандартных типов может не хватить, чтобы красиво реализовать все наши идеи и императивный код внезапно может оказаться проще. Во-вторых, императивные конструкции преподаются со школы, и не смотря на то, что они объективно добавляют сложность языку, для многих программистов они совершенно привычны и не доставляют проблем (хотя такое впечатление бывает обманчивым). Так что, как обычно, в каждом случае самостоятельно оценивай, нравится ли тебе твоё решение, и примут ли его товарищи.

Примитивные типы

Говоря о функциональном программировании не устану напоминать о ключевой роли типов в обеспечении качества кода. Удачные названия локальных переменных, полей и методов повышают читаемость кода, но за надёжность отвечают именно типы, идентификаторы которых хоть и встречаются в гораздо реже, но неявно присутствуют всегда. Типы нужны

компилятору, чтобы гарантировать корректность программы. Поэтому, чем точнее мы будем определять типы, тем качественнее получится программный продукт. И я говорю не только о правильном наименовании.

При моделировании предметной области и даже API нужно стараться не использовать примитивные типы, вроде `String`, `Bool`, `Int`. Когда в большом проекте приходится жонглировать десятками сущностей с такими типами, и перепутать их проще простого. Эти ошибки будет весьма непросто найти.

А вот защититься от этого достаточно легко — самые базовые понятия предметной области нужно замоделировать отдельными уникальными типами. Например: `AdultAge`, `WeightInTones`, `IpAddress`, `RegistrationNumber`, `Postal` и т.п. Перепутать значения таких типов уже не получится, так как компилятор сразу же подсветит несоответствие.

Более того, в отдельных типах можно реализовать валидацию при создании экземпляра. Тогда можно быть уверенным, что у тебя не возникнет корректных состояний программы (отрицательный вес, нешестизначный почтовый индекс и т.п.), что положительно повлияет на надёжность программы.

Самый простой способ определения новых типов взамен примитивных — это классы-образцы:

```
case class WeightInTones(value: double)
```

За простоту придётся платить отсутствием валидации (здесь — на положительность), что не очень хорошо. Лучше закрыть конструктор, а валидируемое создание экземпляра реализовать в объекте компаньоне.

Единожды определённые примитивы предметной области могут явно использоваться, как в логике приложения, так и в моделях передачи данных каких-либо API. Значит, с ними нужно предоставлять средства сериализации [формирования схемы OpenApi и проч.]. По этому лучше использовать уже готовые специализированные решения, вроде библиотеки iron (refined для Scala 2), для которые уже реализованы необходимые интеграции для автоматической сериализации и т.п.:

```
import io.github.iltotore.iron.*
import io.github.iltotore.iron.constraint.all.*

type NameTag = MaxLength[200] & Match["^\\S+(.*\\S+)?$"]

opaque type UserName <: String = String :| NameTag
object UserName extends RefinedTypeOps[String, NameTag, UserName]
```

Ещё сильнее повысить надёжность программы поможет более широкое использование зависимых типов. Но это уже экспериментальная область. Прежде всего, в плане объяснения этой концепции коллегам.

Композиция вычислений

Язык Scala ориентирован на безопасную работу с эффектами, прежде всего, с асинхронностью. Эффекты обрабатываются по возможности «чисто», посредством цепочек монадических вычислений в типах-контейнерах. Синтаксис языка позволяет реализовать композицию таких вычислений различными способами. Вот самые типичные:

- простая «стековая» композиция;
- `for`-выражения;
- потоковые вычисления посредством комбинаторов вроде `.flatMap`;
- операторная композиция;
- бесточечный стиль посредством буквенных (`andThenK`) или символьных (`>=>`) комбинаторов.

Давай посмотрим их на примере таких функций:

```
import cats.syntax.all.*
import cats.effect.IO

val funct1:  Int       => IO[Int] = ???
val funct2:  Int       => IO[Int] = ???
val funct3:  Int       =>    Int  = ???
val funct4: (Int, Int) => IO[Int] = ???
```

Общие выводы будут справедливы для любой системы эффектов.

Первый способ композиции, к счастью, практически нигде не используется:

```
val prog1 = (i: Int) =>
  val res1 = funct1(i)
  val res2 = res1.flatMap(funct2)
  val res3 = res2.map(funct3)
  val res4 = res1.product(res3) // res1 выполнится дважды!
  res4.flatMap(funct4.tupled)
```

Такая наивная программа имеет множество недостатков:

- она многословна,
- перенасыщена знаками препинания,
- множество идентификаторов, большинство которых нужны лишь в следующей строчке, но по ошибке могут использованы и позднее,
- а, пожалуй, главное — в случае «ленивого» контейнера `IO` эффект в `res1` будет выполнен дважды, что навряд ли является ожидаемым результатом.

Последняя проблема исчезает, если программу переписать с помощью `for`-выражения:

```
val prog2 = (i: Int) => for
  res1 <- funct1(i)
  res2 <- funct2(res1)
  res3 =  funct3(res2)
  res4 <- funct4(res3, res1)
yield res4
```

Заодно и код стал немного опрятнее, но большинство проблем осталось. Хоть и знаков препинания теперь меньше, но взамен появилась новая сложная синтаксическая конструкция. И по-прежнему сохранились типичные опасности стекового стиля:

- множество идентификаторов с неоправданно долгим временем жизни, которые не только перегружают код малополезной информацией, но и могут быть неосторожно использованы не по месту, причём даже компилятор не обнаружит ошибку;
- последовательность вычислений зачастую недостаточно фиксируется, из-за чего не только страдает производительность, но приводит к плохо предсказуемому срабатыванию побочных эффектов.

Последний пункт усугубляет то, что `for`-выражение обслуживает лишь последовательную композицию вычислений. Для независимых [«аппликативных»] вычислений придётся как и прежде явно вызывать подходящие методы у контейнеров эффектов. К сожалению, этот аспект частенько ускользает от внимания некоторых программистов, и все вычисления компонуются последовательно, посредством `for` [вспоминается анекдот про слабительное, как лекарство от всех болезней…]. На практике же очень часто приходится пользоваться и другими возможностями, что в сочетании `for` не то, чтобы сильно упрощает код, но чаще, наоборот, усложняет.

`for`-выражения позиционируются в Scala как [что-то на латыни… «killer feature»?]. «Переходите на наш язык, где вы сможете работать в функциональной парадигме почти также привычно, как и ранее на [Java?]!». Вот только с популяризацией этого паттерна связана очень большая проблема. Многие программисты используют `for`-выражения повсеместно, даже не задумываясь. Этот инструмент действительно неплохо подходит для адаптации неофитов, но он не способствует развитию навыков функционального программирования, скорее наоборот.

Не смотря на недостатки, for-выражения действительно бывают удобны и относительно безопасны в некоторых случаях. Но даже там, где они уместны, чисто в образовательных целях, я бы рекомендовал отказаться от них в пользу каких-либо альтернатив.

Например, в пользу такой:

```
val prog3 = (i: Int) =>
  funct1(i)
    .mproduct(funct2)
    .map(_.map(funct3))
    .flatMap(funct4.tupled)
```

Да, тут снова вернулись `map`, `flatMap`, которых не было с `for`, и всё же код стал гораздо компактнее. Ведь, помимо лишней синтаксической конструкции, не осталось и «промежуточных точек» — идентификаторов, откровенно засорявших код ранее. В данном же варианте от строчки к строчке передаётся только то состояние, которое требуется на следующем шаге. Нет смысла именовать его каждый раз затем, чтобы использовать его лишь в следующей строчке и сразу же забыть. Кроме того, этот код защищён от множества неочевидных ошибок, которые можно было бы допустить при внесении правок.

Потоковый стиль могу рекомендовать в качестве универсального. Он даёт единообразный синтаксис вызова метода для обращения ко всем возможностям контейнерных типов. Вот только сам по себе этот синтаксис перенасыщен знаками препинания, что особенно заметно при наличии вложенных вызовов. Впрочем, в таких ситуациях всегда выручает декомпозиция.

От лишних знаков препинания можно избавиться, перейдя к инфиксной форме вызова методов:

```
val mapFunct3 = (_: (Int, Int)) map funct3 // декомпозиция
val prog4 = (i: Int) =>
  funct1(i)     mproduct
  funct2        map
  mapFunct3     flatMap
  funct4.tupled
```

Выглядит почти идеально! Инфиксная запись акцентирует внимание именно на шаги вычислений, а комбинаторы уходят на второй план. Однако, и тут есть ограничение — операторная форма допустима только для методов одного аргумента. Ещё аргумент `i` был нужен только в первой строчке, но он доступен во всём теле функции, что оставляет возможность его ошибочного использования.

Многие библиотеки предлагают символьные операторы, вроде `>>`, или `>>=` взамен `flatMap`. С ними запись может получиться более идиоматичной. К сожалению, единого стандарта этих операторов в Scala нет, и в разных библиотеках (Cats Effect, ZIO) для тех же самых комбинаторов зачастую используются разные наборы символов. Но даже если эти операторы кому-то незнакомы, обычно не возникает проблем с пониманием таких выражений. Поэтому считаю вполне допустимым, когда тело функции состоит из простой комбинации вида `f(a) >> g(b) >> h(c)`.

Наиболее выразительным является бесточечный стиль, когда новые функции мы буквально собираем как конструктор из существующих:

```
import cats.Functor
val funct23 = // Int => IO[Int]
  funct2 andThen Functor[IO].lift(funct3)

val funct23Splitted = // Int => IO[(Int, Int)]
  identity[Int] merge funct23 andThen {_.sequence}
```

```
val prog5 =
  funct1          andThenF
  funct23Splitted andThenF
  funct4.tupled
```

На данном примере видно, что существующих [в библиотеке Cats, но не только] комбинаторов явно недостаточно. Определённо не помешали бы дополнительные `andFMap`, `mergeF`, а также другие комбинаторы, типичные для профункторов [см. Arrow]. Но для чуть менее сложных ситуаций бесточечный стиль определённо лучше остальных и по читаемости, и по стабильности. Кстати, в Cats есть и символьные аналоги таких комбинаторов, заимствованные из Haskell:

```
val anotherProg = // Int => IO[Int]
  funct1 >=> funct2 >=> funct4.curried(42)
```

Эклектика

Весьма неприятным фактором, нарушающим погружению читателя в материал, является смешение различных идей и стилей — эклектика. Очень странно читать, когда суровые викинги высокопарно обсуждают удобность ношения платьев с кринолином, или в якобы письме эпохи возрождения автор рассказывает о современных техниках программирования…

Эклектика столь же неприятна и в программном коде. Не вдаваясь в детали, просто приведу искусственный гротескный пример, которой вряд ли кто-нибудь сочтёт удачным стилем:

```
val eclecticism =
  funct1(42)
        .flatMap(r1 =>
          for{
                r3 <- (funct2 andThen {_.map(funct3)})(r1)
                r4 <- funct4(r1, r3) <* IO.println("log")
                res = if r4 % 2 == 0
                  then Some(r4)
                  else None
          } yield res match
                case Some(x) => x
                case None    => 42
        )
```

Но не стоит боятся использовать разный стиль в разных функциях! Например, при внесении правок в чужой код вполне допустимо написать новые методы в другом [лучшем!] стиле, нежели старый код в этом же файле.

[В дополнение порекомендую статью Functional Programming Anti-Patterns in Scala]

Логическое и метапрограммирование

В языке Scala есть замечательный механизм вывод терма по типу [см. Contextual Abstractions]. Например, написав `summon[MyType]` ты можешь «призвать» ~~могущественного помощника~~ значение типа `MyType`, если, конечно, в контексте призыва есть ~~такой пойманный монстр~~ такое значение. То есть, ранее оно было создано и размещено либо прямо в текущем контексте, либо в другом, который каким-то образом импортирован в текущий.

Этот не значительный с виду механизм обладает колоссальной мощью. Дело в том, что лучшим кодом является тот, который не написан. Контекстные абстракции Scala предоставляют те же возможности, что и в логическом программировании — достаточно добавить в контекст только базовые правила получения значений одних типов из значений других, и компилятор сам за тебя построит [в теории] их композицию для вычисления значения целевого типа из начальных условий!

К сожалению современные инструменты разработки не обладают достаточно хорошей поддержкой контекстных абстракций, да и многие программисты ещё не готовы к такому счастью. При данных обстоятельствах неограниченное использование этого механизма может привести хоть и к лаконичному, полностью типобезопасному коду, но крайне неустойчивому к правкам, так как будет очень сложно понять, как он на самом деле работает. Поэтому авторы языка регулярно добавляют всё новые ограничения на контекстные абстракции, чтобы ими было ещё проще пользоваться в наиболее востребованных [сточки зрения этих авторов] сценариях, но при этом защититься от небезопасных решений. [В наше-то время, наоборот, убирают такие ограничения в языках, но развивают средства разработки и обучают логическому программированию. Так ведь? Да?]

Пожалуй, наиболее часто контекстные абстракции используются для обслуживания классов типов. Обычно речь идёт об экземплярах обобщённых типов вроде `Codec[_]`, `Schema[_]`, `Eq[_]`, `Show[_]`, предоставляемых для конкретных пользовательских типов. Библиотека Cats предлагает оперировать классами контейнерных типов: `Monad[_[_]]`, `Applicative[_[_]]` и т.п. Аналогичные, но уже пользовательские классы типов используются в финальной безтеговой технике [Tagless Final].

Есть и другие сценарии, где вывод термов не только полезен, но и проверен нашими предшественниками: инъекция зависимостей, неявные преобразования [например, между слоями предметной модели и каким-нибудь API], и т.п. И всё же рекомендую принимать решение о внедрении этих, или даже менее проверенных техник, лишь получив одобрение от своей команды.

Аналогичные соображения приходят на ум и при рассмотрении механизмов метапрограммирования. Я имею ввиду прежде всего автоматическое разворачивание макросов на этапе предкомпиляции. Сам по себе этот механизм обладает не самым приятным запахом, так как он призван решать задачи, с которыми не справляется основной язык программирования, или же его компилятор. [Имеет смысл уделять большее внимания развитию самого языка, нежели «нечестным» техникам на основе метапрограммирования.]

Макросы часто используются в самых разных библиотеках, например, для автоматического вывода тех же экземпляров классов пользовательских типов. Но и, помимо макросов, даже в

ядре языка нередко используются различные «мета-встраивания» кода [см. Inline].

Я не стал бы рекомендовать использование макросов в коде программного продукта. Но категорически запрещать тоже не стоит — а вдруг твой коллега найдёт действительно стоящее решение? Надеюсь только, что он предварительно обсудит его с товарищами, а не попробует протащить контрабандой через ревью…

[Порекомендую такую статью об относительно безопасном метапрограммировании: Inline your boilerplate – harnessing Scala 3 metaprogramming without macros]

Общие советы

Рефакторинг

Найдя и реализовав решение поставленной задачи всегда полезно посмотреть на его составляющие и подумать, «нужно ли мне это?», «нет ли тут чего-то лишнего?», «можно ли сделать проще?». Причём это касается не только твоих собственных правок, но и смежного кода. Например, если возникли затруднения с пониманием существующей логики, которую тебе нужно поправить, то переписать её — может быть лучшем решением. Конечно же, всегда нужно оценивать такой рефакторинг, и если затраты на него окажутся не целесообразными на данный момент, то не игнорировать проблему, а запланировать задачу на будущее [как минимум, `TODO` оставьте!].

Мне встречалось немало программистов, которые старательно избегали делать что-либо сверх того, что нужно по их задаче, и вносить лишь самые минимальные правки, стараясь вообще не касаться существующего кода. Иногда даже собственные правки намеренно не проверялись на предмет того, можно ли их сделать более качественно. Чаще всего это оправдывается спешкой, горящими сроками, но, подозреваю, что многим просто было [неразборчиво]…

Мой тебе совет — не пренебрегай рефакторингом! Нужно развивать этот навык, чтобы поверить в свои силы и не боятся каждый раз тратить немного времени на повышение качества кода. Навряд ли из-за этого могут сдвинуться какие-либо сроки, за то такой подход может существенно продлить время жизни всей кодовой базы.

Общение с товарищами

Приходилось слышать мнение, что на ревью не нужно спорить про стиль и формат. Но это ошибочное мнение! Проверять корректность решения задачи глазами — дело весьма неблагодарное. Да, можно заметить какие-нибудь архитектурные неточности, но не всегда у проверяющего есть весь необходимый контекст. А вот что точно бросается в глаза на ревью, так это именно качество кода — читаемость, выразительность, надёжность и безопасность дальнейших правок.

И это действительно стоит обсуждать — задавать вопросы, высказывать мнения, делиться опытом и учиться самому. Конструктивный диалог о качестве кода помогает развитию не

только способностей грамотного программирования, но и коммуникативных навыков, что тоже очень важно для программиста [для всех!]. Кстати, это ещё одна важная причина не фиксировать жёстко «истинность» каких-либо правил кодирования, а «дозволять» больше творческой свободы коллегам и пользоваться ею самому.

Если же у тебя есть сомнения в удачности твоего решения и в том, примут ли его коллеги, то лучше самостоятельно инициировать дискуссию, согласовать это решение со всеми заинтересованными, убедить их, или переубедиться самому. Иначе сомнительные правки могут случайно попасть в общую кодовую базу, что вызовет проблемы в дальнейшем.

Ввести ограничения для себя и коллег всегда проще, чем развить в коллективе культуру свободы — чувство прекрасного, ответственность за собственные и чужие решения, свободу обсуждения. Не нужно стремиться навязать стремление к свободе, это всё равно на получится. Но нужно общаться, убеждать и при этом всегда быть готовым к тому, что какая-то чужая истина в итоге окажется для тебя «более истинной».

Postscriptum

Когда тебя убеждают, что одна парадигма программирования лучше всех других, помни:
Ничто не истинно!

Когда тебе не рекомендуют пользоваться строгими ограничениями, помни
Всё дозволено!

~~Да направит тебя Отец Понимания!~~
~~Да прибудет с тобой сила!~~
[Точно не уверен, какое там напутствие]

Не привыкай доверять готовым ответам. Окончательное решение принимать тебе и отвечать за него будешь ты.