Приёмы функционального программирования

Функциональное программирование - не просто еще один стиль написания кода, это философия, которая коренным образом меняет подход к разработке программного обеспечения. Функциональное программирование предлагает инструменты для создания более предсказуемого и устойчивого кода, стремится к минимизации побочных эффектов и поддержанию высокого уровня абстракции, что делает код более модульным, тестируемым и легким в поддержке.

Говоря о различных парадигмах программирования, стоит сказать, что на сегодняшний день знание разных подходов для написания кода является необходимостью для программиста, обусловленной разнообразием задач и контекстов, в которых они возникают. Каждая парадигма предлагает уникальный набор принципов и практик, оптимально подходящих для решения определенного класса проблем. В то время как императивные и объектно-ориентированные подходы доминировали в индустрии на протяжении многих лет, функциональное программирование возможно только выходит на передний план, предлагая решения для современных задач. Поговорим в начале чуть более подробнее о данной парадигме.

Как уже было оговорено, функциональное программирование — парадигма программирования, в которой процесс вычисления трактуется как вычисление значений функций в математическом их понимании. Она предполагает вычисление результатов функций от исходных данных и результатов других функций, и не предполагает явного хранения состояния программы. Соответственно, не предполагает она и изменяемость этого состояния (в отличие от, например, императивной, где одной из базовых концепций является переменная, хранящая своё значение и позволяющая менять его по мере выполнения алгоритма).

В рамках курсового проекта по дисциплине “Функциональное программирование” нашей командой был написан свой собственный функциональный язык программирования “RADI”. На его примере хочется более подробно рассказать о функциональной парадигме программирования. Для этого кратко изложим, из каких частей состоит

наш проект; как его реализация отвечает функциональному стилю; и какие приёмы собственно были использованы.

RADI

Для написания своего языка программирования был выбран язык F# в соответствии с проходимым курсом. Разбивая на задачи написание всего языка, можно выделить две основных части: парсер и интерпретатор. Остановимся на каждой из них подробно.

Парсер

Как известно, парсер (он же - синтаксический анализатор) - код, преобразующий входные данные (текст) в какой-либо структурированный формат, с которым дальше происходит работа.

Парсеры существуют нескольких видов. В нашем же языке программирования было принято решение реализовывать так называемый комбинаторный парсер.

Комбинаторный парсер (combinator parser) - это функция высшего порядка, которая принимает на вход несколько парсеров и возвращает новый парсер на выходе. В данном случае парсер - это функция, принимающая какие-то строки на вход и возвращающая некоторую структуру на выходе.

Стоит подробнее остановиться на преимуществах использования комбинаторного парсера для функционального языка.

Как уже было сказано, комбинаторные парсеры являются функциональными конструкциями, представляющими собой композицию функций высшего порядка, что в первую очередь естественным образом вписывается в функциональную парадигму программирования. Благодаря тому, что комбинаторные парсеры легко комбинируются и составляются из более мелких частей, обеспечивается модульность и появляется возможность строить сложные грамматики для языка из небольших блоков, каждый из которых может быть протестирован и повторно использован. Кроме того, они предоставляют “элегантные” способы

обработки ошибок и отказоустойчивости, позволяют определять стратегии восстановления после ошибок, которые могут быть выражены функционально.

Многие функциональные языки, такие как Haskell или F#, имеют встроенную поддержку комбинаторных парсеров в виде библиотек или языковых конструкций. Так, например, F# имеет встроенную библиотеку FParsec, которая, к слову, и использовалась для написания парсера. Говоря об этой библиотеке, стоит сказать, что в ней используются те же классические концепции функционального программирования: использование функций в качестве аргументов и возвращаемых значений; неизменяемость данных после создания; рекурсия, комбинаторы и так далее.

Говоря о недостатках функционального подхода к написанию парсера, стоит сказать, что комбинаторные парсеры могут быть менее эффективными по сравнению с парсерами, написанными в императивном стиле, особенно когда речь идет о больших объемах данных. А из-за частого использования рекурсии могут возрасти расходы памяти. Что касается внесения правок и расширения парсера (добавление новых функций, изменение поведения), то здесь также могут возникнуть трудности в связи с высокой связностью компонентов.

Интерпретатор

Итак, парсер получил на вход какую-то строку, “распарсил” её. Что дальше? Дальше программа получает дерево выражений (абстрактное синтаксическое дерево). Это дерево представляет собой объект алгебраического типа данных (АТД).

АТД или Алгебраические типы данных — это ключевая концепция в функциональных языках программирования. АТД позволяют определять типы данных, которые могут принимать различные, но фиксированные формы. Алгебраические типы данных в свою очередь делятся на две категории:

Произведение типов - типы данных, которые могут содержать несколько значений разных типов одновременно. Например, записи, структуры или кортежи.

Сумма типов - типы данных, которые могут быть одним из нескольких предопределенных вариантов.

После того, как мы получили дерево выражения с помощью парсера (а вся программа у нас представляет собой одно выражение), мы вычисляем это выражение. За вычисление выражения отвечает функция "Evaluate"
В ней используется функциональный прием - сопоставление с образцом. Это всем известный и довольно простой метод анализа и обработки структур данных, основанный на выполнении определённых инструкций в зависимости от совпадения исследуемого значения с тем или иным образцом, в качестве которого может использоваться константа, предикат, тип данных или иная поддерживаемая языком конструкция.

Для вычисления дерева выражений был выбран аппликативный (строгий) порядок вычислений. Такой порядок предполагает, что аргументы функции вычисляются перед тем, как функция будет применена. То есть каждый аргумент вычисляется до того, как функция начнёт выполняться.

Помимо аппликативного порядка есть еще нормальный (ленивый) порядок. Его суть заключается в противоположном: аргументы функции не вычисляются до тех пор, пока это действительно не потребуется. Это означает, что функции получают свои аргументы в виде ещё не вычисленных выражений, а само вычисление происходит только в том случае, если результат аргумента необходим для продолжения выполнения программы. Преимущество этого подхода заключается в том, что он может предотвратить ненужные вычисления, особенно если аргументы функции в конечном итоге не используются.

Учитывая наши цели и задачи, мы выбрали аппликативный порядок вычисления, поскольку чаще всего все аргументы действительно были необходимы для вычисления функций.

Для обработки ошибок и описания последовательности действий был использован монадический тип Result. Монада - особый тип данных в функциональных языках программирования, для которого возможно задать императивную последовательность выполнения некоторых операций над хранимыми значениями. Монады позволяют задавать последовательность выполнения операций, производить операции с побочными эффектами и другие действия, которые сложно или вовсе невозможно реализовать в функциональной парадигме программирования другими способами.

Вот, пожалуй, и все приёмы функциональной парадигмы, которые мы использовали при выполнения курсовой работы. В конце хочется сказать несколько слов вывода о ней.

Заключение

Подводя итог проделанной работе, в первую очередь стоит отметить, что было очень интересно и крайне полезно познакомиться с новой парадигмой. Функциональное программирование действительно оказалось довольно красивым и лаконичным благодаря высокому уровню математических абстракций. Кроме того, мы познакомились с новыми приемами или взглянули на старые по-другому. К примеру, тогда как в императивной парадигме программисты зачастую стараются избегать рекурсии, в функциональной наоборот - данный приём является одним из ключевых и даже поощряется в силу более грамотного подхода и некоторых оптимизаций (по типу преобразования хвостовой рекурсии). Или, например, более часто стали использоваться композиции функций, что в свою очередь уменьшает длину программ, поскольку повторно не пишется один и тот же код.

На данный момент можно сказать, что функциональный подход не настолько “популярен” среди программистов (то ли дело ООП). Во многом это связано с тем, что он сложнее и может быть применен не ко всем задачам, тем не менее ряд преимуществ, как уже говорилось, (чистота кода, надежность программ и оптимизация) позволяют его популярности расти. И на наш взгляд, это хорошо: программисту не нужно замыкаться на одной парадигме, писать только в одном стиле. Комбинирование разных парадигм и приёмов будет намного эффективнее.

Выходит, недостатки у функциональной парадигмы не настолько существенные. Да, разработчику с опытом программирования в ООП, может быть, сложно перестроить мышление и подход. После императивной парадигмы было непросто обходиться без переменных. С другой стороны новичку освоить принципы ФП может быть немного проще с нуля. Что касается того, что для ФП не подходит для решения каких-то задач, ответ прост - использовать его в тех задачах, где оно подходит. Чаще всего на практике нет необходимости писать весь проект в функциональном стиле, поэтому можно комбинировать разные приемы, как говорилось выше.

Так, мы довольно близко познакомились с функциональной парадигмой программирования; открыли глаза на совершенно новый подход к написанию программ; убедились на практике (!) в её реальных пользе и удобстве. При дальнейшей работе над проектами будем использовать полученный опыт и знания в функциональном подходе.