Приёмы функционального программирования, как мы их использовали при выполнении курсовой, и как будем использовать в дальнейшей жизни

Функциональное программирование — это парадигма программирования, главный упор в которой делается на работу с функциями. В отличии от классического императивного программирования, в котором программа рассматривается как четкий набор инструкций определяющих последовательное изменение ее состояний, в функциональном программировании используется декларативный подход, который заключается в описании конечного результата работы программы, а не шагов для его получения.

Такой подход имеет ряд преимуществ при написании кода.

Во-первых, вычислительный процесс, описанный в декларативном стиле, соответствует привычному мышлению человека, не имеющего опыта в программировании, так как в реальной жизни люди, выполняя какие-либо действия, чаще всего думают о результате.

Во-вторых, программы, написанные в функциональном стиле, получаются более компактными. При их создании больше внимания нужно уделять не механическому написанию кода, а обдумыванию решения. Это позволяет делать программы более читаемыми.

Далее рассмотрим основные приемы функционального программирования.

Неизменяемость данных. Идея данного приема заключается в том, чтобы использовать структуры данных: переменные, списки, массивы, - которые нельзя изменить. В преимущества этого приема можно отнести отсутствие побочных эффектов, что ведет за собой сокращение количества ошибок, а также простоту распараллеливания программы.

При создании интерпретатора для нашего языка мы использовали этот прием, так как в языке программирования F# все структуры

данных по умолчанию являются неизменяемыми, а для создания переменных, которые можно переприсваивать, нужно явно указывать это.

Нам этот подход показался достаточно удобным, и именно поэтому в созданным нами языке мы также реализовали неизменяемые структуры данных. Причиной этого стало то, что программист, пишущий на языке, в котором реализована такая возможность, будет полностью уверен в том, что лежит в этой переменной, и точно знать, что он ее нигде не мог изменить.

Чистые функции. Использование данного приема подразумевает создание функций, которые не оказывают никакого влияние на внешние ресурсы, такие как файлы или базы данных, не изменяет объекты и не вызывает внутри себя другие функции, не являющимися чистыми. Таким образом, чистые функции не производят никаких побочных эффектов и зависят только от переданных ей аргументов.

Так как объекты в языке программирования F# по умолчанию являются неизменяемыми, то и внутри функций мы также не можем изменить эти переменные.

Использование чистых функций в своей программе делает ее более читаемой и понятной для любого программиста, так как не нужно смотреть имплементацию этой функции, чтобы удостовериться в том, что в теле функции нет никаких изменений внешних переменных, нужно лишь знать, какое значение возвращает эта функция, что гораздо легче и приводит к более быстрому пониманию кода, а также позволяет иметь более предсказуемое поведение программы и упрощают её отладку.

В реализованном нами языке программировании все функции являются чистыми.

Рекурсия. Концепция рекурсии состоит в том, что функция может вызывать саму себя. Рекурсия является достаточным универсальным инструментом, так как с помощью неё можно реализовывать большое количество алгоритмов и структур данных. При рекурсивных вызовах функции при работе программы создается стек вызовов, в котором хранятся локальные переменные и адрес возврата для каждого вызова функции. При создании рекурсивных функций нужно быть аккуратным и обязательно указывать условие, при котором эта рекурсия должна закончиться, так как в противном случае возникнут бесконечная рекурсия и переполнение стека вызовов. Для того чтобы как-то улучшить опыт работы с рекурсиями, существует хвостовая рекурсия. Для создания такой рекурсии функция должна обладать следующими свойствами: линейность (единственный рекурсивный вызов в теле функции), рекурсивный вызов – последняя операция в теле функции, отсутствие локальных переменных. Благодаря этому компилятор языка сможет хорошо оптимизировать рекурсию так, чтобы не было необходимости хранить адреса вызов.

При разработке интерпретатора своего языка программирования рекурсия является необходимым приемом, так как существует необходимость обхода абстрактного синтаксического дерева для выполнения кода программы.

Для нашего языка программирования мы также реализовали возможность создания рекурсивных функций.

Лямбда-выражения. В современных языках программирования этот прием обычно представлен лямбда-функциями. Лямбда-функции, в отличие от обычных функций, являются анонимными и поэтому объявляются без имени, а при необходимости лямбда-функции можно присваивать переменным. Это делает такие функции очень удобными в случаях, когда необходимо передать функцию в качестве аргумента другой функции. Если функция, которую необходимо передать, достаточно компактна и используется в коде программы только один

раз, удобнее описать ее в списке аргументов при вызове функции, которая принимает другую функцию как аргумент.

В своей работе мы использовали лямбда-функции при обработке списков и словарей с помощью свертки, отображения и других методов, реализованных в языке программирования F#. Так как над элементами этих структур данных не требовалось совершать каких-то сложных действий, было удобнее описывать эти действия не в отдельных функциях, а при вызове методов этих структур.

При разработке нашего языка Flopper мы также попытались реализовать лямбда-функции. Это позволило упростить разработку интерпретатора, так как все функции в нашем языке имеют такое же представление в абстрактном синтаксическом дереве, как и обычные переменные. Единственное отличие функциональной переменной от численной или строковой — это ее значение, которое представляет собой лямбда функцию, а не число или строку.

Функции высших порядков. Этот прием заключается в использовании функций в качестве аргументов и возвращаемых значений других функций. Он позволяет создавать универсальные функции, которые можно использовать в программе повторно без дублирования кода. В языке F# реализовано большое количество таких функций, использующихся для обработки различных структур данных. Примером могут служить функции свертки и отображения списков, принимающие как аргумент функцию, которую необходимо применить ко всем элементам списка.

Эти функции часто использовались при написании интерпретатора, так как код программы преобразуется парсером в список утверждений. Для сохранения и передачи информации об объявленных переменных в текущей области видимости при обработке утверждений удобно использовать функцию свертки со словарем, который хранит пары «имя-значение», в качестве аккумулятора.

Такая универсальность функций показалась нам очень удобной, поэтому наш язык программирования также реализует функции высших порядков.

Замыкание. Замыканием называется комбинация самой функции и ее лексического окружения, которое определяется текущей областью видимости. Часто замыкание используется в функциях высших порядков, которые возвращают функции, способные сохранять свое состояние между вызовами. Это состояние определяется в момент вызова функции высшего порядка.

С его помощью замыкания можно удобно описывать вспомогательные функции внутри других функций. Такая вспомогательная функция может использовать все аргументы, переданные внешней функции, и при этом она не засоряет внешнюю область видимости. Таким образом с помощью замыкания можно эмулировать приватные методы классов из объектно-ориентированного программирования.

Также замыкание может накладывать свои ограничения. Например, при декларации функции в нашем языке программирования Flopper, несмотря на то что все функции анонимные, требуется явно указать имя переменной, которой присвоена эта функция. Если этого не сделать, то эта переменная не войдет в лексическое окружение функции, и будет невозможно выполнить рекурсивный вызов функции.

Такой прием отлично сочетается с функциями высших порядков и значительно расширяет область их применения, поэтому мы также используем его в языке Flopper.

Применение описанных подходов в будущем.

Функциональное программирование предоставляет мощный набор идей, которые становятся все более популярными в разработке ПО. Мы обнаружили, что такие подходы, как неизменяемость данных и чистые функции, значительно снижают вероятность побочных эффектов, делая программы более предсказуемыми и надежными. В будущем, эти принципы могут быть широко применены при разработке безопасных и масштабируемых систем, особенно в областях, таких как финтех и облачные вычисления, где требуется высокий уровень контроля над состоянием и безопасностью данных.

Рекурсия и использование лямбда-выражений облегчают написание кода, делая программы более читаемыми и снижая количество ошибок. Функции высшего порядка и замыкания позволяют разработчикам создавать более абстрактные и мощные интерфейсы, способствуя модульности кода. С увеличением сложности систем и требований к ним, эти принципы функционального программирования могут стать необходимыми инструментами для каждого программиста.

Однако, функциональное программирование не является панацеей. В своей книге "Чистый код", Роберт Мартин подчеркивает, что функциональное программирование наиболее применимо для структур, которые не имеют собственного поведения, поскольку это облегчает добавление новой функциональности. В то же время, это может быть неэффективно для классов с полиморфными сущностями, поскольку для добавления нового класса придется изменять уже существующие функции.

Заключение.

Функциональное программирование, начавшееся в академической среде, сейчас играет важную роль в промышленной разработке ПО. Многие из подходов, которые мы использовали в нашем курсе, создают основу для написания эффективного и безопасного кода. По мере развития мира программирования, навыки и знания в области

функционального программирования становятся незаменимыми для решения будущих технологических задач. Наш опыт показывает, что функциональное программирование будет занимать все более важное место в сфере разработки ПО, требуя от специалистов постоянного обучения и развития в этом направлении.