Эссе

Введение

Программы написанные на функциональном языке программирования заметно отличаются от программ на традиционных языках, выполняющие схожие действия. Это отличие заключается в идеологии программирования.

Традиционные языки являются продуктом развития идеи машины Тьюринга, где программа представляет собой набор команд, изменяющий состояние переменной(а затем и объекта) при помощи операции присваивания.

Функциональные языки же, можно сказать, из простых функций создают сложные, которые преобразуют входные данные в выходные без изменения состояния.

Написанный нами язык является “ленивым” и не будет ничего считать, пока мы не вызовем подходящую функцию. Идея сделать его ленивым пришла не случайно, как нам кажется ленивые языки имеют значительные преимущества. Итоговая конфигурация языка позволяет удобно работать с бесконечными структурами данных, а также просто распараллеливать вычисления. Сам язык работает следующим образом: весь файл с кодом на нашем языке парсится и превращается в дерево. Основная структурная единица дерева - OtherContext. Можно сказать, что он определяет область видимости кода. Соответственно, каждый раз, когда появляется новый контекст в общий контекст вводится новая сущность. Но как только этот контекст заканчивается, map возвращается к последнему значению. Переменные в нашем языке неизменные, перезаписать их нельзя. Если Вы попытаетесь создать переменную с таким же именем, то у Вас получится это сделать, однако пока контекст не завершится, вы будете обращаться к последней версии этой переменной.

Приемы в F#

В данной части работы мы обращаем внимание на ключевые приемы и концепции функционального программирования, которые мы успешно применили в разработке нашего языка программирования.

Основная часть курсовой работы состоит из парсера и интерпретатора. Начнем с парсера.

Для парсинга кода, который вводит пользователь мы воспользовались инструментами fparsec. Парсер фактически является скоплением более узких парсеров, которые предназначены для считывания информации своего типа. Для создания всех парсеров мы пользуемся функцией.

```
let DifferParse, OperRef = createParserForwardedToRef()
```

Фактически мы создаем основной парсер, объединяющий все другие парсеры и ссылку на него. Ссылка нам понадобится позже, теперь перейдем к другим парсерам.

Все парсеры после DifferParse можно разделить на два вида: простые и составные. Простые парсеры используют только примитивы fparse, составные же используют как примитивы, так и уже сделанные парсеры. Среди примитивов есть крайне удобные вещи, позволяющие оперируя малым количеством парсеров, парсить все что угодно.

Первое удобство - цепи парсеров. В парсере курсовой работы встречаются указания для парсеров: .>>, >>., |>> и другие. Эти средства позволяют отбрасывать ненужные для интерпретатора символы. Внутри себя они фактически используют такие приемы функционального программирования, как каррирование и композиция. Под каррированием я имею в виду технику, которая позволяет преобразовывать функцию с несколькими аргументами в последовательность функций с одним аргументом. В данном случае каррирование превращает составной парсер в последовательность парсеров с одним аргументом (вводимыми данными или их обработанными версиями).

Второе удобство - альтернативы. Мы пользовались следующими примитивами альтернативы - <|> , >>?, attempt. Примитив альтернативы <|> позволяет комбинировать два парсера так, чтобы если первый парсер не смог разобрать входные данные на токены, то те же данные попробовал обработать второй парсер. Примитив >>? используется в коде ниже. Это код является чистой функцией (зависит только от входных данных) и является композицией парсеров в аргументах в лучших традициях функционального программирования.

```
let private manyCharsBetween popen pclose pchar = popen >>? manyCharsTill pchar pclose
```

Альтернатива представляет собой такой прием функционального программирования как функция высших порядков. Действительно, она принимает в себя два парсера и применяет их к входным данным.

many1satisfy2 принимает функцию предиката в качестве аргумента, которая определяет, удовлетворяет ли каждый элемент последовательности определенному условию. Этот подход к парсингу позволяет легко изменять условия фильтрации без изменения основной логики парсера.

Для своих операций мы были вынуждены ввести свой синтаксис базовых вещей, чтобы их было можно считать. В результате для математических операций мы пользовались приемом частичного применения функции:

```
 let private Equal = pstring "eq" >>. spaces >>. skipChar '(' >>. spaces >>. DifferParse .>> skipChar ',' .>> spaces .>>. DifferParse .>> skipChar ')' .>> spaces |>> (fun (left, right) -> BasicOperation ("==", left, right))
```

Для того, чтобы язык программирования работал, нужно текстовый файл передать парсеру для создания дерева контекстов, а затем дерево передать интерпретатору для исполнения команд. Соответственно нужно обеспечить возможность считывать файлы с написанными на нашем языке программирования командами. Для этого используется parsercore из не упоминавшихся ранее приемах функционального программирования можно отметить следующие:

- Анонимные функции: Анонимные функции не используются напрямую в коде parsercore, но внутри конструкции match применяются лямбда-выражения для обработки различных вариантов результата парсинга.
- Неизменяемые состояния: Переменная text создается как неизменяемая и содержит текст из файла, который затем используется для парсинга.

Перейдем к интерпретатору. В нем мы интерпретируем полученное дерево команд.

Начнем с сущности Construction. Для реализации языка мы пользовались алгебраическим типом данных. Алгебраические типы данных позволяют объединить несколько типов данных в один тип, который может принимать различные формы. В данном случае, тип `Construction` представляет собой алгебраический тип данных, который может принимать различные формы, такие как `Float`, `Bool`, `OtherContext`, `Function` и так далее.

Каждый вариант `Construction` представляет различный тип конструкции или выражения в вашем языке программирования. Это позволяет явно определить все возможные варианты конструкций и обрабатывать их соответствующим образом в коде интерпретатора.

В функции eval мы использовали много приемов, например рекурсивные вызовы LibraryOperation и CatalogOperaion. Сама функция является функцией сопоставления с образцом для правильного определения типа конструкции и выполнения подходящих операций. В самом интерпретаторе в основном использованы те же принципы что и раньше указанные: рекурсия, функции первого класса, функции высшего порядка и другие.

Как это использовать в дальнейшие жизни.

В данной главе мы обсудим ключевые приемы и концепции, представленные в нашей курсовой работе, которые, как нам кажется, окажутся наиболее значимыми и крайне полезными в дальнейшем развитии нас как программистов. Кроме того, мы также обсудим приемы, которые мы считаем наиболее перспективными для использования в будущем, но которые не были представленные в рамках данной работы.

Функциональное программирование было придумано достаточно давно, но не сыскало популярности, так как в это время память была дорогой и использование ее для разработки на функциональных языках было неоправданно. Сейчас память стала доступной, нельзя сделать транзисторы значительно меньше, для быстрого выполнения задач используется больше одного ядра и операционные сети утратили тотальный контроль за потоками выполнения задач. В результате возникает проблема - для ускорения вычислений мы можем, пока что, только использовать больше ядер, но из-за принципов работы традиционных языков происходит конкуренция за память. Тут вспоминают про функциональные программирование, позволяющее избегать сложности с конкаренси, блокировкой и параллелизмом. Я считаю, что это главное преимущество функционального подхода. И на самом деле не только я, потому что функциональные принципы пытаются внедрить в новые языки программирования для обеспечения удобного распараллеливания программ.

Так, например, в языке Go заимствуют приемы и философию функционального метода в реализации своей многопоточности. Не вдаваясь в подробности, Go поощряет безопасное изменение данных путем обеспечения гарантированного доступа к данным из одного goroutine в определенное время.

Кроме потоков есть ещё одна проблема решаемая функциональными подходами - побочные эффекты. Когда пишется приложение полагающие взаимодействие множества пользователей возникает множество проблем. В многопоточных программах тестирование и понимание процесса становится ещё сложнее. Так, наиболее частый случай (и пример плохой практики программирования),

это изменение состояния переменной вне области видимости функции, например глобальной. И если программа имеет побочные эффекты, то модульными тестами не докажешь корректность программы. Пользователи являются непредсказуемыми агентами и могут сделать то, что тестировщики не могут предвидеть. Соответственно с такими технологиями программирования пользователи могут случайно (или специально) вызвать непредсказуемое поведение и принести большие убытки компании. Чтобы таких ситуаций не возникало существует такой шаблон программирования как монады. Побочные действия внутри функций, которые могут привести к неопределенному или непредсказуемому поведению, монады предлагают способ описать эти эффекты в виде чистых функций.

Одним из ключевых преимуществ функционального программирования и монад является то, что чистые функции не имеют побочных эффектов. Это означает, что функция всегда возвращает одинаковый результат для одинаковых входных данных и не зависит от состояния программы или внешних условий.

Монады предоставляют способ комбинировать функции, которые работают с побочными эффектами, в чистом функциональном стиле. Они позволяют инкапсулировать эффекты и обеспечивают безопасный способ их выполнения, гарантируя при этом, что программы остаются предсказуемыми и легко тестируемыми.

Например, монада Option в F# используется для обработки возможного отсутствия значения (null). Вместо того чтобы проверять значение на null в каждом месте программы блоками try/catch, можно использовать монадические операции, такие как map, flatMap или bind, чтобы работать с данными внутри монады Option. К слову об асинхронном программировании, есть монада Async, которая представляет асинхронную операцию, выполняющуюся в фоновом режиме. Это позволяет эффективно управлять асинхронными операциями, такими как ввод-вывод или сетевые запросы, и не вызывать побочные эффекты.

Говоря о более простых, но не менее полезных приемах функционального программирования, хочется упомянуть мемоизацию и предподсчет. Как было сказано ранее, память стала доступнее, а значит эти подходы могут значительно ускорить работу приложений. И это делается уже сейчас: мемоизация используется в научных вычислениях для оптимизации расчетов, в веб разработке для оптимизации времени ответа на повторяющиеся запросы.

В заключении хочу сказать, что хоть, вероятно, на функциональных языках мы не будем программировать все время своей работы разработчиками, принципы и функциональные подходы отложатся в голове. Функциональный подход обеспечивает предсказуемое поведение программ, позволяет писать чистый и понятный человеку код, а главное писать его быстро, безопасно, с возможность легкого тестирования и сопровождения. И знание этого подхода позволяет проще адаптировать к изменяющимся требованиям и ситуациям, что особенно важно в быстро развивающемся мире информационных технологий, в котором мы живем.