**Московский государственный технический**

**университет им. Н.Э. Баумана**

Факультет «Радиотехнический»

Кафедра «Информатика и вычислительная техника»

Курс «Парадигмы и конструкции языков программирования»

Отчет по домашнему заданию

Тема «Язык программирования F#»

|  |  |  |
| --- | --- | --- |
| Выполнил: |  | Проверил: |
| студент группы РТ5-31Б: |  | преподаватель каф. ИУ5 |
| Ильина К.А. |  | Гапанюк Ю.Е. |
| Подпись и дата: |  | Подпись и дата: |

Москва, 2023 г.

**Содержание**

[**Введение** 3](#_Toc154549522)

[**Цель и задачи реферата** 3](#_Toc154549523)

[**Hello world!** 4](#_Toc154549524)

[**Функциональное программирование** 4](#_Toc154549525)

[**Программирование с помощью функций** 4](#_Toc154549526)

[**Неизменяемость** 5](#_Toc154549527)

[**Выражения** 6](#_Toc154549528)

[**Рекурсивные функции** 7](#_Toc154549529)

[**Функции высшего порядка** 7](#_Toc154549530)

[**Лямбды** 8](#_Toc154549531)

[**Замыкания** 8](#_Toc154549532)

[**Сопоставление с образцом (pattern matching).** 9](#_Toc154549533)

[**Реактивное программирование** 10](#_Toc154549534)

[**Работа с асинхронными операциями:** 10](#_Toc154549535)

[**Использование асинхронных вычислений и операций:** 11](#_Toc154549536)

[**Использование потоков данных и событий** 12](#_Toc154549537)

[**Заключение** 13](#_Toc154549538)

[**Источники** 13](#_Toc154549539)

# 

# **Введение**

F# впервые увидел свет в 2005 году в Microsoft Research. Это мультипарадигмальный язык для платформы Microsoft .NET, нацеленный преимущественно на функциональное программирование, имеющий, тем не менее, возможность создания объектно-ориентированного и императивного кода. F# является статически типизированным языком. Вследствие этогоинформация о типах доступна уже на этапе компиляции, что обеспечивает большую надежность программного кода

Несмотря на своё название, F# не является развитием C#, напротив будучи скорее его полной противоположностью. Из общего у них разве что разработчик, да возможность реализации всех трёх парадигм программирования, однако приоритеты у них принципиально разные.

**Цель и задачи реферата**

В реферате мы изучим основные концепции функционального программирования, такие как работа с функциями, неизменяемость данных, рекурсия, замыкания. Мы также рассмотрим особенности реактивного программирования.

# **Hello world!**

По традиции, чтобы начать, приведу пример программы, печатающий на экран “Hello world!” на языке F#:

*// HelloWorld.fs*

*printfn “Hello, World”*

# **Функциональное программирование**

Функциональное программирование — это стиль программирования, в котором особое значение придается использованию функций и неизменяемых данных. Типизированное функциональное программирование — это сочетание функционального программирования со статическими типами, как это характерно для F#. В целом в функциональном программировании применяются такие основные подходы:

* функции как основные используемые конструкции;
* выражения вместо инструкций;
* неизменяемые значения имеют приоритет перед переменными;
* декларативное программирование имеет приоритет перед императивным программированием.

## **Программирование с помощью функций**

В основе функционального программирования лежит представление

о коде в терминах математических функций. Рассмотрим две функции,

f и g:

f(x) = x^2 + x

g(x) = x + 1

Из этого следует, что:

f(2) = (2)^2 + (2)

g(2) = (2) + 1

И если теперь объединить эти две функции, получим следующее:

f g (2) = f(g(2))

= (g(2))^2 + (g(2))

= (2+1)^2 + (2+1)

= 12

Код эквивалентный этой математической записи на F# имеет следующий вид(делается это с помощью ключевого слова *let*):

let f x = x \*\* 2.0 + x

let g x = x + 1.0

## **Неизменяемость**

В языке F# неизменяемость данных является одним из ключевых принципов функционального программирования. Она означает, что данные, однажды созданные, не могут быть изменены. Вместо этого, любые операции, которые кажутся изменением данных, фактически создают новые данные, основываясь на существующих.

Неизменяемость данных в F# имеет несколько преимуществ:

1. Предсказуемость: Поскольку данные неизменяемы, их значения не могут быть случайно изменены в процессе выполнения программы. Это делает программы более предсказуемыми и позволяет легче понимать и отлаживать код.
2. Потокобезопасность: Неизменяемые данные безопасны для использования в многопоточных средах, так как они не могут быть изменены одновременно из разных потоков. Это упрощает разработку параллельных и распределенных систем.
3. Упрощение разработки: Использование неизменяемых данных позволяет избежать множества проблем, связанных с изменяемостью, таких как состояние гонки, побочные эффекты и сложности синхронизации. Это делает код более простым, понятным и легким для поддержки.
4. Функциональные преимущества: Неизменяемость данных является ключевым аспектом функционального программирования. Она позволяет использовать функции без побочных эффектов, что упрощает разработку, тестирование и повторное использование кода. Кроме того, неизменяемость способствует применению функций высшего порядка, рекурсии и других функциональных конструкций.

В F# неизменяемость достигается путем использования неизменяемых типов данных, таких как кортежи, списки, массивы и записи (records). При выполнении операций над такими данными создаются новые экземпляры с измененными значениями, в то время как исходные данные остаются неизменными.

*Пример 1. Использования неизменяемых данных в F#:*

let numbers = [1; 2; 3; 4] // Создание неизменяемого списка чисел

let sum = List.sum numbers // Вычисление суммы элементов списка

let doubledNumbers = List.map (fun x -> x \* 2) numbers // Создание нового списка, удвоив каждый элемент

printfn "Sum: %d" sum

printfn "Doubled numbers: %A" doubledNumbers

В этом примере список numbers является неизменяемым, и операции List.sum и List.map создают новые списки с измененными значениями, оставляя исходный список неизменным.

Неизменяемость данных в F# способствует созданию более надежных, модульных и эффективных программ, а также упрощает разработку и поддержку кода.

*Пример 2. Вычисление суммы квадратов списка чисел в императивном*

*и функциональном стилях [1]*

let square x = x \* x

let imperativeSum numbers =

let mutable total = 0

for i in numbers do

let x = square i

total <- total + x

total

let functionalSum numbers =

numbers

|> Seq.map square

|> Seq.sum

## **Выражения**

Выражения — это конструкции, возвращающие значения. В отличие от инструкций, выполняющих действия, выражения можно рассматривать как действия с возвратом значения. В функциональном программировании чаще всего используются выражения, а не инструкции [2].

// 'x + 1' is an expression!

let addOne x = x + 1

Это результат выражения, определяющий тип результата функции addOne. Например, выражение, образующее эту функцию, можно изменить на другой тип, такой как string:

let addOne x = x.ToString() + "1"

Выражения — это не просто тела функций. Можно использовать выражения, возвращающие значение, которое затем используется в другой области. Одно из распространенных выражений — это if:

// Checks if 'x' is odd by using the mod operator

let isOdd x = x % 2 <> 0

let addOneIfOdd input =

let result =

if isOdd input then

input + 1

else

input

result

## **Рекурсивные функции**

Рекурсия - это процесс, в котором функция вызывает саму себя с измененными аргументами. Рекурсивные функции широко применяются в функциональном программировании для решения сложных задач, которые требуют итеративного подхода. В F# рекурсивные функции могут быть определены с помощью ключевого слова *rec*.

*Пример рекурсивной функции - вычисление факториала числа:*

let rec factorial n =

if n <= 1 then 1

else n \* factorial (n - 1)

let result = factorial 5 // Вызов рекурсивной функции

printfn "Factorial: %d" result

В этом примере функция factorial вызывает саму себя с аргументом n - 1, пока не будет достигнуто базовое условие n <= 1. Рекурсивная функция позволяет элегантно и компактно вычислить факториал числа.

## **Функции высшего порядка**

- это функции, которые принимают другие функции в качестве аргументов или возвращают их в качестве результатов. Они позволяют создавать абстрактные функции, которые могут быть применены к различным типам данных и обеспечивают гибкость в программировании.

*Пример функции высшего порядка - функция map, которая применяет заданную функцию к каждому элементу списка:*

let map f list =

List.map f list

let square x = x \* x

let numbers = [1; 2; 3; 4]

let squaredNumbers = map square numbers

printfn "Squared numbers: %A" squaredNumbers

В этом примере функция map принимает функцию f и список list, а затем применяет функцию f к каждому элементу списка с помощью функции List.map. Таким образом, мы можем использовать функцию map для применения различных функций к списку чисел.

Функции высшего порядка позволяют создавать абстрактные функции, которые могут быть переиспользованы для различных типов данных и операций. Они также способствуют улучшению модульности и гибкости кода.

## **Лямбды**

F# поддерживает анонимные функции или лямбды, которые используются, если нет необходимости присваивать функции имя, когда она передается в качестве параметра для другой функции. *Пример лямбды ниже:*

    List.map (fun x -> x\*\*2) [1..10]

Данная функция выдаст список, состоящий из квадратов всех чисел от одного до десяти.  
Кроме того, в F# существует и еще один способ определения лямбды с помощью ключевого слова *function*. Определенная таким образом лямбда может содержать внутри себя операцию сравнения с шаблоном (pattern matching), однако она принимает только один параметр. Но даже и в этом случае можно сохранить каррируемость функции:

    function x -> function y -> x + y

Лямбды в F# поддерживают *замыкание [3]*.

## **Замыкания**

В F# замыкания позволяют сохранять состояние и использовать его при каждом вызове функции. Замыкание - это функция, которая захватывает переменные из окружающего контекста, включая переменные, объявленные во внешней области видимости. Захваченные переменные сохраняются в замыкании и могут быть использованы при каждом вызове функции.

*Пример использования замыкания в F#:*

let createCounter () =

let mutable count = 0

fun () ->

count <- count + 1

count

let counter = createCounter()

printfn "Count: %d" (counter()) // Выводит: Count: 1

printfn "Count: %d" (counter()) // Выводит: Count: 2

printfn "Count: %d" (counter()) // Выводит: Count: 3

В этом примере функция createCounter создает замыкание, которое захватывает переменную count из своего окружающего контекста. Замыкание возвращает анонимную функцию, которая увеличивает значение count при каждом вызове и возвращает его. Переменная count сохраняется в замыкании и используется для подсчета количества вызовов.

При вызове createCounter() мы получаем замыкание, которое можно присвоить переменной counter. Затем мы можем вызывать counter() для увеличения значения count и получения текущего счетчика.

## **Сопоставление с образцом (pattern matching).**

Используется для фильтрования и сортировки данных.

Сопоставление с образцом напоминает инструкцию switch в языках C# и C++, но обладает более широкими возможностями.

По сути это набор правил, которые выполняются при совпадении входного значения с образцом. Затем выражение сопоставления с образцом возвращает результат правила, для которого было найдено соответствие. Вследствие этого всеправила должны возвращать значения одного и того же типа.

Применяются ключевые слова *match* и *with* с набором правил, за каждым из которых следует стрелка ->.

*Пример простого сопоставления с образцом:*

let isOdd x = (x % 2 = 1)

let describeNumber x =

match isOdd x with

| true -> printfn “x is odd”

| false -> printfn “x is even”;;

val isOdd : int -> bool

val describeNumber : int -> unit

94 Глава 3. Функциональное программирование

> describeNumber 4;;

x is even

val it : unit = ()

*Пример. Создание таблицы истинности с помощью сопоставления*

*с образцом:*

> // Таблица истинности для функции AND с помощью сопоставления с образцом

let testAnd x y =

match x, y with

| true, true -> true

| true, false -> false

| false, true -> false

| false, false -> false;;

val testAnd : bool -> bool -> bool

> testAnd true true;;

val it : bool = true

# **Реактивное программирование**

Реактивное программирование в F# позволяет эффективно работать с асинхронными операциями, потоками данных и событиями.

## **Работа с асинхронными операциями:**

F# предоставляет ключевое слово *async* для определения асинхронных вычислений. Это позволяет писать асинхронный код в более последовательном стиле, используя конструкции *let!* и *do!* для ожидания завершения асинхронных операций.

*Пример работы с асинхронными операциями в F#:*

let asyncOperation () =

async {

printfn "Starting async operation..."

do! Async.Sleep 1000 // Асинхронная задержка на 1 секунду

printfn "Async operation completed"

return 42

}

let main () =

async {

printfn "Main started"

let! result = asyncOperation ()

printfn "Result: %d" result

}

## **Использование асинхронных вычислений и операций:**

F# предоставляет библиотеку *Async*, которая содержит функции и операции для работы с асинхронными вычислениями. Например, *Async.Parallel* позволяет выполнять асинхронные операции параллельно, а *Async.AwaitTask* позволяет ожидать выполнение задачи .NET Framework.

*Пример использования асинхронных операций в F#:*

let asyncOperation1 () =

async {

printfn "Async operation 1 started"

do! Async.Sleep 1000

printfn "Async operation 1 completed"

return 1

}

let asyncOperation2 () =

async {

printfn "Async operation 2 started"

do! Async.Sleep 2000

printfn "Async operation 2 completed"

return 2

}

let main () =

async {

printfn "Main started"

let! result1 = asyncOperation1 ()

let! result2 = asyncOperation2 ()

printfn "Result: %d, %d" result1 result2

}

## **Использование потоков данных и событий**

Основным инструментом для работы **с потоками данных** в F# является библиотека *System.Reactive*, которая предоставляет набор функций и операций для работы с потоками данных.

Основным типом данных для работы с потоками данных в F# является *Observable<'T>*, который представляет наблюдаемый поток данных элементов типа 'T. *Observable<'T>* предоставляет операции, такие как map, filter, scan и другие, которые позволяют преобразовывать, фильтровать и агрегировать элементы потока данных.

*Пример использования потоков данных в F#:*

open System

open System.Reactive.Linq

let numbers = Observable.Range(1, 10) // Создание потока данных с числами от 1 до 10

let evenNumbers = numbers

|> Observable.filter (fun x -> x % 2 = 0) // Фильтрация только четных чисел

|> Observable.map (fun x -> x \* 2) // Умножение каждого числа на 2

let subscription = evenNumbers.Subscribe(printfn "Result: %d") // Подписка на поток данных и вывод результатов

// Вывод:

// Result: 4

// Result: 8

// Result: 12

// Result: 16

// Result: 20

subscription.Dispose() // Отмена подписки

Как видно из примера, можно создавать цепочки операций над потоками данных, применяя различные преобразования и фильтры к элементам потока.

Кроме работы с потоками данных, F# также предоставляет инструменты для работы с **событиями**. В F# события представлены типами *Event* и *Event<'T>*. События могут быть использованы для обработки асинхронных сигналов и уведомлений.

*Пример использования событий в F#:*

open System

let event = new Event<\_>()

let subscription = event.Subscribe(printfn "Event occurred: %A") // Подписка на событие и вывод уведомлений

event.Trigger(42) // Генерация события с аргументом 42

event.Trigger("Hello") // Генерация события со строковым аргументом

// Вывод:

// Event occurred: 42

// Event occurred: "Hello"

subscription.Dispose() // Отмена подписки

В приведенном примере мы создаем событие *event* и подписываемся на него с помощью функции *Subscribe*. Затем мы генерируем событие с помощью метода *Trigger*, и все подписчики получают уведомление о событии.

Использование потоков данных и событий в F# позволяет создавать *реактивные системы,* которые эффективно обрабатывают потоки данных и реагируют на изменения данных и асинхронные сигналы. Это особенно полезно при разработке асинхронных приложений.

# **Заключение**

В ходе написания реферата были рассмотрены основные особенности таких парадигм программирования, как функциональное и реактивное программирование на языке F#. Применение этих парадигм позволяет разработчикам создавать гибкий, модульный и отзывчивый код c высокой производительностью.

# **Источники**

**1)** Смит К. Программирование на F#. – Пер. с англ. – СПб.: Символ-Плюс, 2011. – 448 с.

**2)** Руководство по языку F#. URL: https://learn.microsoft.com/ru-ru/dotnet/fsharp/tutorials/functional-programming-concepts

**3)** Три парадигмы F#. URL: https://habr.com/ru/articles/55272/