

Московский государственный технический университет
имени Н.Э. Баумана

Факультет «Информатика и системы управления»
Кафедра «Системы обработки информации и управления»

Г.И. Ревунков, Ю.Е. Гапанюк

ВВЕДЕНИЕ В ФУНКЦИОНАЛЬНОЕ И ЛОГИЧЕСКОЕ ПРОГРАММИРОВАНИЕ

Электронное учебное издание – рабочая версия

Москва

(С) 2017 МГТУ им. Н.Э. БАУМАНА

ОГЛАВЛЕНИЕ

1	КРАТКИЙ ОБЗОР ИСТОЧНИКОВ ПО ФУНКЦИОНАЛЬНОМУ И ЛОГИЧЕСКОМУ ПРОГРАММИРОВАНИЮ.....	5
1.1	ФУНКЦИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ.....	5
1.1.1	<i>Журнал на русском языке по ФП.....</i>	6
1.1.2	<i>Наиболее распространенные ФП-языки.....</i>	6
1.1.2.1	F#.....	6
1.1.2.2	OCaml.....	6
1.1.2.3	Scala.....	6
1.1.2.4	Erlang.....	7
1.1.2.5	Haskell.....	7
1.2	МУЛЬТИПАРАДИГМАЛЬНОЕ ПРОГРАММИРОВАНИЕ.....	7
1.3	ЛОГИЧЕСКОЕ ПРОГРАММИРОВАНИЕ.....	8
1.3.1	<i>Прямой логический вывод.....</i>	8
1.3.2	<i>Обратный логический вывод.....</i>	8
1.3.2.1	Язык Prolog.....	8
1.3.3	<i>Изоморфизм Карри-Ховарда.....</i>	9
2	МОТИВИРУЮЩИЙ ПРИМЕР ИСПОЛЬЗОВАНИЯ ФУНКЦИОНАЛЬНОГО ПОДХОДА К ПРОГРАММИРОВАНИЮ. АЛГЕБРАИЧЕСКИЕ ТИПЫ И СОПОСТАВЛЕНИЕ С ОБРАЗЦОМ	10
2.1	ПРИМЕР НА ЯЗЫКЕ C# С ИСПОЛЬЗОВАНИЕМ СПИСКА КОРНЕЙ.....	10
2.2	ПРИМЕР НА ЯЗЫКЕ C# С ИСПОЛЬЗОВАНИЕМ ПЕРЕЧИСЛЕНИЯ.....	11
2.3	ПРИМЕР НА ЯЗЫКЕ F# С ИСПОЛЬЗОВАНИЕМ АЛГЕБРАИЧЕСКОГО ТИПА.....	12
2.4	ПРИМЕР НА ЯЗЫКЕ C# С ИСПОЛЬЗОВАНИЕМ ИНТЕРФЕЙСА И НАСЛЕДУЕМЫХ КЛАССОВ.....	14
2.5	ПРИМЕР НА ЯЗЫКЕ F# С ИСПОЛЬЗОВАНИЕМ ИНТЕРФЕЙСА И НАСЛЕДУЕМЫХ КЛАССОВ.....	15
3	ОСНОВЫ ЯЗЫКА F#	16
3.1	СОЗДАНИЕ КОНСОЛЬНОГО ПРОЕКТА В F#.....	16
3.2	КОММЕНТАРИИ В F#.....	18
3.3	ОСНОВНЫЕ ТИПЫ ДАННЫХ.....	19
3.4	ОПЕРАТОРЫ ВВОДА-ВЫВОДА.....	20
3.5	ОБЪЯВЛЕНИЕ И НЕИЗМЕНЯЕМОСТЬ ПЕРЕМЕННЫХ.....	20
3.6	ИСПОЛЬЗОВАНИЕ КОРТЕЖЕЙ.....	21
3.7	СОЗДАНИЕ ФУНКЦИЙ.....	23
3.7.1	<i>Создание простых функций.....</i>	23
3.7.2	<i>Возврат кортежей из функций.....</i>	23
3.7.3	<i>Рекурсивные функции.....</i>	24
3.7.4	<i>Взаимно-рекурсивные функции и эмуляция конечного автомата.....</i>	24
3.7.5	<i>Хвостовая рекурсия.....</i>	28
3.7.6	<i>Каррирование.....</i>	30

3.7.7	<i>Лямбда-выражения</i>	35
3.7.8	<i>Аналог делегатного типа в F#</i>	36
3.7.9	<i>Обобщенные типы в функциях</i>	37
3.8	КОЛЛЕКЦИИ	41
3.8.1	<i>Списки</i>	41
3.8.2	<i>Массивы</i>	44
3.9	ФУНКЦИИ ВЫСШЕГО ПОРЯДКА	45
3.9.1	<i>length</i>	46
3.9.2	<i>exists</i>	46
3.9.3	<i>forall</i>	47
3.9.4	<i>find</i>	47
3.9.5	<i>filter</i>	47
3.9.6	<i>map</i>	48
3.9.7	<i>fold</i>	48
3.9.8	<i>append</i>	49
3.9.9	<i>zip</i>	50
3.9.10	<i>rev</i>	50
3.9.11	<i>sort</i>	50
3.9.12	<i>partition</i>	51
3.9.13	<i>Функции агрегирования</i>	51
3.10	АЛГОРИТМ БЫСТРОЙ СОРТИРОВКИ СПИСКОВ	51
3.11	ОПЕРАТОРЫ ПОТОКОВ И КОМПОЗИЦИИ ФУНКЦИЙ	54
3.11.1	<i>Операторы потоков</i>	54
3.11.2	<i>Операторы композиции функций</i>	55
4	ОСНОВНЫЕ ОПЕРАТОРЫ ЯЗЫКА F#	56
4.1	ОПЕРАТОРЫ ЦИКЛОВ	56
4.1.1	<i>Цикл For</i>	56
4.1.2	<i>Цикл While</i>	56
4.2	УСЛОВНЫЕ ОПЕРАТОРЫ	57
4.2.1	<i>Оператор if</i>	57
4.2.2	<i>Сопоставление с образцом. Оператор match</i>	57
4.3	ОБРАБОТКА ИСКЛЮЧЕНИЙ	59
5	РАБОТА С ТИПАМИ В ЯЗЫКЕ F#	60
5.1	ЗАПИСИ (RECORD TYPES)	61
5.2	АЛГЕБРАИЧЕСКИЕ ТИПЫ (DISCRIMINATED UNIONS)	61
5.3	КЛАССЫ (CLASSES)	62
5.4	АБСТРАКТНЫЕ КЛАССЫ	64
5.5	ИНТЕРФЕЙСЫ	65
5.6	АКТИВНЫЕ ШАБЛОНЫ	66

6	ФУНКТОРЫ, АППЛИКАТИВНЫЕ ФУНКТОРЫ И МОНАДЫ	67
6.1	ФУНКТОРЫ	69
6.2	АППЛИКАТИВНЫЕ ФУНКТОРЫ	75
6.3	ФУНКЦИИ ЛИФТИНГА.....	81
6.4	МОНАДЫ	81
6.5	ВЫВОДЫ ПО РАЗДЕЛУ.....	87
7	БИБЛИОТЕКА FPARSEC ДЛЯ СИНТАКСИЧЕСКОГО АНАЛИЗА ТЕКСТА	87
7.1	УСТАНОВКА FPARSEC	88
7.2	ОСНОВЫ РАБОТЫ С FPARSEC	88
7.3	ПРИМЕР РАЗБОРА ДАННЫХ С ПРИМЕНЕНИЕМ АЛГЕБРАИЧЕСКОГО ТИПА.....	91
8	ИСПОЛЬЗОВАНИЕ МУЛЬТИАГЕНТНОГО ПОДХОДА В F#.....	93
9	ЗАДАНИЯ ПО КУРСУ	96
9.1	ЛАБОРАТОРНЫЕ РАБОТЫ.....	96
9.1.1	<i>Лабораторная работа №1.....</i>	<i>96</i>
9.1.2	<i>Лабораторная работа №2.....</i>	<i>96</i>
9.1.3	<i>Лабораторная работа №3.....</i>	<i>97</i>
9.1.4	<i>Лабораторная работа №4.....</i>	<i>98</i>
9.1.5	<i>Лабораторная работа №5.....</i>	<i>99</i>
9.1.6	<i>Лабораторная работа №6.....</i>	<i>99</i>
9.1.7	<i>Лабораторная работа №7.....</i>	<i>100</i>
9.2	ДОМАШНЕЕ ЗАДАНИЕ	100

1 Краткий обзор источников по функциональному и логическому программированию

1.1 Функциональное программирование

Описание ФП —

https://ru.wikipedia.org/wiki/Функциональное_программирование

Языки ФП делятся на чистые (Haskell) и с побочными эффектами (большинство языков) —

https://ru.wikipedia.org/wiki/Чистота_языка_программирования

С точки зрения парадигмы программирования языки ФП можно разделить на:

1. Однопарадигмальные. Используют только приемы функционального программирования. Примеры – Haskell, Erlang.
2. Объектно-функциональные. Наряду с ФП позволяют использовать ООП. Примеры – F#, OCaml, Scala.
3. Функционально-логические. Сочетают концепции функционального и логического программирования. Пример – Mercury —
[https://ru.wikipedia.org/wiki/Mercury_\(язык_программирования\)](https://ru.wikipedia.org/wiki/Mercury_(язык_программирования))
4. Мультипарадигмальные с развитым метапрограммированием. Используя метапрограммирование (поддержку макросов) как ядро языка существует возможность реализовать разные парадигмы, в том числе функциональную. Пример – LISP. Как правило такие языки обладают свойством самоотображаемости (гомоиконичности) – <https://ru.wikipedia.org/wiki/Гомоиконичность>

1.1.1 Журнал на русском языке по ФП

«Практика функционального программирования» - <http://fprog.ru/>
 Обзор литературы о функциональном программировании –
<http://fprog.ru/2009/issue1/alex-ott-literature-overview/>

1.1.2 Наиболее распространенные ФП-языки

1.1.2.1 F#

https://ru.wikipedia.org/wiki/F_Sharp

Документация - <http://fsharp.org/about/index.html#documentation>

Книги:

1. Сошников Д.В. «Функциональное программирование на F#»
2. Don Syme, Adam Granicz, Antonio Cisternino «Expert F# 4.0»
3. Tao Liu «F# for C# Developers»

1.1.2.2 OCaml

<https://ru.wikipedia.org/wiki/OCaml>

Книга:

Ярон Мински, Анил Мадхавапедди и Джейсон Хикки
 «Программирование на языке OCaml».

На OCaml написана одна из самых известных систем автоматического
 доказательства теорем - <https://ru.wikipedia.org/wiki/Coq>

1.1.2.3 Scala

[https://ru.wikipedia.org/wiki/Scala_\(язык_программирования\)](https://ru.wikipedia.org/wiki/Scala_(язык_программирования))

Книга:

Кей Хостманн «Scala для нетерпеливых»

Дистанционный цикл из 5 курсов (специализация) на английском
 языке - <https://www.coursera.org/specializations/scala>

1.1.2.4 Erlang

<https://ru.wikipedia.org/wiki/Erlang>

Синтаксис основан на логическом языке Пролог, хотя Erlang – функциональный язык.

Используется динамическая типизация, поэтому очень многие техники ФП, связанные с типами, не работают в Erlang.

Erlang широко применяется на практике для разработки высоконагруженных систем.

Хорошая вводная статья по Erlang -

<https://rdsn.org/article/erlang/GettingStartedWithErlang.xml>

Книга:

Фред Хеберт «Изучай Erlang во имя добра!»

1.1.2.5 Haskell

<https://ru.wikipedia.org/wiki/Haskell>

Книги:

1. Миран Липовача «Изучай Haskell во имя добра!»

2. Книги Р.В. Душкина

Сайт «Русский учебник по Haskell» - <http://anton-k.github.io/ru-haskell-book/book/toc.html>

Дистанционный курс на русском языке -

<https://stepik.org/course/Функциональное-программирование-на-языке-Haskell-75>

1.2 Мультипарадигмальное программирование

Как правило такие языки обладают свойством самоотображаемости (гомоиконичности) – <https://ru.wikipedia.org/wiki/Гомоиконичность> (в статье приведены примеры языков).

Наиболее известный язык – LISP и его диалекты.

Наиболее известный диалект - Clojure

<https://ru.wikipedia.org/wiki/Clojure>

Документация - <https://clojure.org/reference/reader>

Книга:

Чаз Эмерик, Брайен Карпер, Кристоф Гранд «Программирование на Clojure»

1.3 Логическое программирование

1.3.1 Прямой логический вывод

От данным к цели (вывод, управляемый данными).

https://en.wikipedia.org/wiki/Forward_chaining

Программа записывается в виде продукционных правил «ЕСЛИ условие ТО действие». Порядок вызова правил определяется динамически машиной вывода. Результат выполнения предыдущих правил меняет состояние машины вывода (операционную память) и приводит к вызову следующих правил. Цель динамически выводится в результате выполнения правил.

Традиционно применялись в экспертных системах.

Пример – язык CLIPS - <https://ru.wikipedia.org/wiki/CLIPS>

В последнее время также применяются как средство программирования общего назначения.

Пример – система Drools - <https://en.wikipedia.org/wiki/Drools>

1.3.2 Обратный логический вывод

От цели к данным.

https://en.wikipedia.org/wiki/Backward_chaining

1.3.2.1 Язык Prolog

[https://ru.wikipedia.org/wiki/Пролог_\(язык_программирования\)](https://ru.wikipedia.org/wiki/Пролог_(язык_программирования))

База знаний задается в виде предикатов. Правила вывода определяют связь между предикатами. Необходимо указать цель поиска и машина вывода пытается перебрать все комбинации предикатов, чтобы доказать цель. Задача машины вывода – найти комбинацию исходных данных при которых цель выполняется или доказать что их нет.

Фактически этот подход является аналогом языка запросов к базе знаний, выводимая цель соответствует конкретному запросу.

Существует диалект Пролога - Datalog (<https://en.wikipedia.org/wiki/Datalog>) который применяется как язык запросов к базам данных, в том числе к реляционным.

Книга:

Иван Братко «Алгоритмы искусственного интеллекта на языке Prolog»

Реализации Prolog:

1. Простая учебная реализация SWI Prolog - <http://www.swi-prolog.org/>
2. Одна из наиболее развитых реализаций, включающая логику высших порядков XSB Prolog - <http://xsb.sourceforge.net/>

1.3.3 Изоморфизм Карри-Ховарда

Существует важный принцип, который устанавливает связь между функциональным и логическим программированием – https://ru.wikipedia.org/wiki/Соответствие_Карри_—_Ховарда

Книга:

Пирс Б. «Типы в языках программирования».

2 Мотивирующий пример использования функционального подхода к программированию. Алгебраические типы и сопоставление с образцом

Примеры проектов приведены в архиве на сайте. Используется Visual Studio 2015 Community Edition.

Рассмотрим пример вычисления корней квадратного уравнения.

Проект «SquareRoot».

2.1 Пример на языке C# с использованием списка корней

```
using System;
using System.Collections.Generic;

namespace SquareRoot
{
    /// <summary>
    /// Простое вычисление корней
    /// </summary>
    class SquareRoot_Simple
    {
        /// <summary>
        /// Вычисление корней
        /// </summary>
        public List<double> CalculateRoots(double a, double b, double c)
        {
            List<double> roots = new List<double>();
            double D = b * b - 4 * a * c;
            //Один корень
            if (D == 0)
            {
                double root = -b / (2 * a);
                roots.Add(root);
            }
            //Два корня
            else if (D > 0)
            {
                double sqrtD = Math.Sqrt(D);
                double root1 = (-b + sqrtD) / (2 * a);
                double root2 = (-b - sqrtD) / (2 * a);
                roots.Add(root1);
                roots.Add(root2);
            }
            return roots;
        }

        /// <summary>
        /// Вывод корней
        /// </summary>
        public void PrintRoots(double a, double b, double c)
        {
```

```

List<double> roots = this.CalculateRoots(a, b, c);
Console.WriteLine("Коэффициенты: a={0}, b={1}, c={2}. ", a, b, c);
if (roots.Count == 0)
{
    Console.WriteLine("Корней нет.");
}
else if (roots.Count == 1)
{
    Console.WriteLine("Один корень {0}", roots[0]);
}
else if (roots.Count == 2)
{
    Console.WriteLine("Два корня {0} и {1}", roots[0], roots[1]);
}
}
}
}

```

Не очень надежно. Вдруг произошел сбой и вернулась пустая коллекция.

Для увеличения надежности будем использовать перечисление (enum).

2.2 Пример на языке C# с использованием перечисления

```

using System;
using System.Collections.Generic;

namespace SquareRoot
{
    /// <summary>
    /// Перечисление для обозначения количества корней
    /// </summary>
    enum RootsEnum { NoRoots, OneRoot, TwoRoots }

    /// <summary>
    /// Вычисление корней с использованием перечисления
    /// </summary>
    class SquareRoot_Enum
    {
        /// <summary>
        /// Вычисление корней
        /// </summary>
        public void CalculateRoots(double a, double b, double c, out List<double>
roots, out RootsEnum rootFlag)
        {
            rootFlag = RootsEnum.NoRoots;
            roots = new List<double>();
            double D = b * b - 4 * a * c;
            //Один корень
            if (D == 0)
            {
                rootFlag = RootsEnum.OneRoot;
                double root = -b / (2 * a);
                roots.Add(root);
            }
            //Два корня

```

```

else if (D > 0)
{
    rootFlag = RootsEnum.TwoRoots;
    double sqrtD = Math.Sqrt(D);
    double root1 = (-b + sqrtD) / (2 * a);
    double root2 = (-b - sqrtD) / (2 * a);
    roots.Add(root1);
    roots.Add(root2);
}
}

/// <summary>
/// Вывод корней
/// </summary>
public void PrintRoots(double a, double b, double c)
{
    List<double> roots;
    RootsEnum rootFlag;
    this.CalculateRoots(a, b, c, out roots, out rootFlag);
    Console.WriteLine("Коэффициенты: a={0}, b={1}, c={2}. ", a, b, c);
    if (rootFlag == RootsEnum.NoRoots)
    {
        Console.WriteLine("Корней нет.");
    }
    else if (rootFlag == RootsEnum.OneRoot)
    {
        Console.WriteLine("Один корень {0}", roots[0]);
    }
    else if (rootFlag == RootsEnum.TwoRoots)
    {
        Console.WriteLine("Два корня {0} и {1}", roots[0], roots[1]);
    }
}
}
}

```

Не очень удобно. Список корней и значение перечисления возвращаются по отдельности. Можно ли их объединить? Теоретически их можно объединить в отдельный класс, но все равно список корней и значение перечисления можно изменить отдельно друг от друга.

В F# они объединены в единую структуру, которая называется алгебраическим типом.

2.3 Пример на языке F# с использованием алгебраического типа

```

//Для использования классов Math и Console
open System

// Алгебраический тип или "Discriminated Unions"
// Алгебраический тип - тип сумма из типов произведений
// | - означает "или" и задает тип-сумму
// * - означает "и" и задает произведение (кортеж, который соединяет все элементы)
// В абстрактных алгебрах наиболее близкой алгеброй является полукольцо

```

```

///Тип решения квадратного уравнения
type SquareRootResult =
    | NoRoots
    | OneRoot of double
    | TwoRoots of double * double //кортеж из двух double

///Функция вычисления корней уравнения
let CalculateRoots(a:double, b:double, c:double):SquareRootResult =
    let D = b*b - 4.0*a*c;
    if D < 0.0 then NoRoots
    else if D = 0.0 then
        let rt = -b / (2.0 * a)
        OneRoot rt
    else
        let sqrtD = Math.Sqrt(D)
        let rt1 = (-b + sqrtD) / (2.0 * a);
        let rt2 = (-b - sqrtD) / (2.0 * a);
        TwoRoots (rt1,rt2)

///Вывод корней (тип unit - аналог void)
let PrintRoots(a:double, b:double, c:double):unit =
    printf "Коэффициенты: a=%A, b=%A, c=%A. " a b c
    let root = CalculateRoots(a,b,c)
    //Оператор сопоставления с образцом
    let textResult =
        match root with
        | NoRoots -> "Корней нет"
        | OneRoot(rt) -> "Один корень " + rt.ToString()
        | TwoRoots(rt1,rt2) -> "Два корня " + rt1.ToString() + " и " + rt2.ToString()
    printfn "%s" textResult

[<EntryPoint>]
let main argv =
    //Тестовые данные
    //2 корня
    let a1 = 1.0;
    let b1 = 0.0;
    let c1 = -4.0;
    //1 корень
    let a2 = 1.0;
    let b2 = 0.0;
    let c2 = 0.0;
    //нет корней
    let a3 = 1.0;
    let b3 = 0.0;
    let c3 = 4.0;

    PrintRoots(a1,b1,c1)
    PrintRoots(a2,b2,c2)
    PrintRoots(a3,b3,c3)

    //|> ignore - перенаправление потока с игнорирование результата вычисления
    Console.ReadLine() |> ignore
    0 // возвращение целочисленного кода выхода

```

У алгебраического типа есть недостаток. Его нельзя расширять в процессе реализации. А в C# можно воспользоваться решением на основе интерфейсов.

2.4 Пример на языке C# с использованием интерфейса и наследуемых классов

```

using System;
using System.Collections.Generic;

namespace SquareRoot
{
    interface RootsResult { }
    class NoRoots : RootsResult { }
    class OneRoot : RootsResult
    {
        public double root { get; set; }
    }
    class TwoRoots : RootsResult
    {
        public double root1 { get; set; }
        public double root2 { get; set; }
    }

    /// <summary>
    /// Возможные варианты решения расширяются за счет использования интерфейса
    /// </summary>
    class SquareRoot_WithInterface
    {
        /// <summary>
        /// Вычисление корней
        /// </summary>
        public RootsResult CalculateRoots(double a, double b, double c)
        {
            List<double> roots = new List<double>();
            double D = b * b - 4 * a * c;
            //Один корень
            if (D == 0)
            {
                double rt = -b / (2 * a);
                return new OneRoot()
                {
                    root = rt
                };
            }
            //Два корня
            else if (D > 0)
            {
                double sqrtD = Math.Sqrt(D);
                double rt1 = (-b + sqrtD) / (2 * a);
                double rt2 = (-b - sqrtD) / (2 * a);
                return new TwoRoots()
                {
                    root1 = rt1,
                    root2 = rt2
                };
            }
            //Нет корней
            else
            {
                return new NoRoots();
            }
        }
    }

    /// <summary>

```

```

/// Вывод корней
/// </summary>
public void PrintRoots(double a, double b, double c)
{
    RootsResult result = this.CalculateRoots(a, b, c);
    Console.WriteLine("Коэффициенты: a={0}, b={1}, c={2}. ", a, b, c);
    string resultType = result.GetType().Name;
    if (resultType == "NoRoots")
    {
        Console.WriteLine("Корней нет.");
    }
    else if (resultType == "OneRoot")
    {
        OneRoot rt1 = (OneRoot)result;
        Console.WriteLine("Один корень {0}", rt1.root);
    }
    else if (resultType == "TwoRoots")
    {
        TwoRoots rt2 = (TwoRoots)result;
        Console.WriteLine("Два корня {0} и {1}", rt2.root1, rt2.root2);
    }
}
}
}

```

Этот вариант лучше, потому что он позволяет расширять варианты решения, добавляя новые классы, наследуемые от интерфейса. Но в F# тоже так можно.

2.5 Пример на языке F# с использованием интерфейса и наследуемых классов

```

open System

///Интерфейс
type SquareRootEmpty = interface end
//Наследуемые классы с вариантами решения
type NoRoots()=
    interface SquareRootEmpty
//Клсс содержит параметры, которые присваиваются свойству
type OneRoot(p:double)=
    interface SquareRootEmpty
    // Объявление свойства
    member val root = p : double with get, set

type TwoRoots(p1:double,p2:double)=
    interface SquareRootEmpty
    // Объявление свойства
    member val root1 = p1 : double with get, set
    member val root2 = p2 : double with get, set

///Функция вычисления корней уравнения
let CalculateRoots(a:double, b:double, c:double):SquareRootEmpty =
    let D = b*b - 4.0*a*c;
    if D < 0.0 then (new NoRoots() :> SquareRootEmpty)
    else if D = 0.0 then
        let rt = -b / (2.0 * a)

```

```

    //Требуется явное приведение к интерфейсному типу
    (OneRoot(rt) :> SquareRootEmpty)
else
    let sqrtD = Math.Sqrt(D)
    let rt1 = (-b + sqrtD) / (2.0 * a);
    let rt2 = (-b - sqrtD) / (2.0 * a);
    (TwoRoots(rt1,rt2) :> SquareRootEmpty)

///Вывод корней (тип unit - аналог void)
let PrintRoots(a:double, b:double, c:double):unit =
    printf "Коэффициенты: a=%A, b=%A, c=%A. " a b c
    let root = CalculateRoots(a,b,c)
    //Оператор сопоставления с образцом по типу - :?
    let textResult =
        match root with
        | :? NoRoots -> "Корней нет"
        | :? OneRoot as r -> "Один корень " + r.root.ToString()
        | :? TwoRoots as r -> "Два корня " + r.root1.ToString() + " и " + r.root2.ToString()
        | _ -> "" // Если не выполняется ни один из предыдущих шаблонов
    printfn "%s" textResult

[<EntryPoint>]
let main argv =
    //Тестовые данные
    //2 корня
    let a1 = 1.0;
    let b1 = 0.0;
    let c1 = -4.0;
    //1 корень
    let a2 = 1.0;
    let b2 = 0.0;
    let c2 = 0.0;
    //нет корней
    let a3 = 1.0;
    let b3 = 0.0;
    let c3 = 4.0;

    PrintRoots(a1,b1,c1)
    PrintRoots(a2,b2,c2)
    PrintRoots(a3,b3,c3)

    //|> ignore - перенаправление потока с игнорирование результата вычисления
    Console.ReadLine() |> ignore
    0 // возвращение целочисленного кода выхода

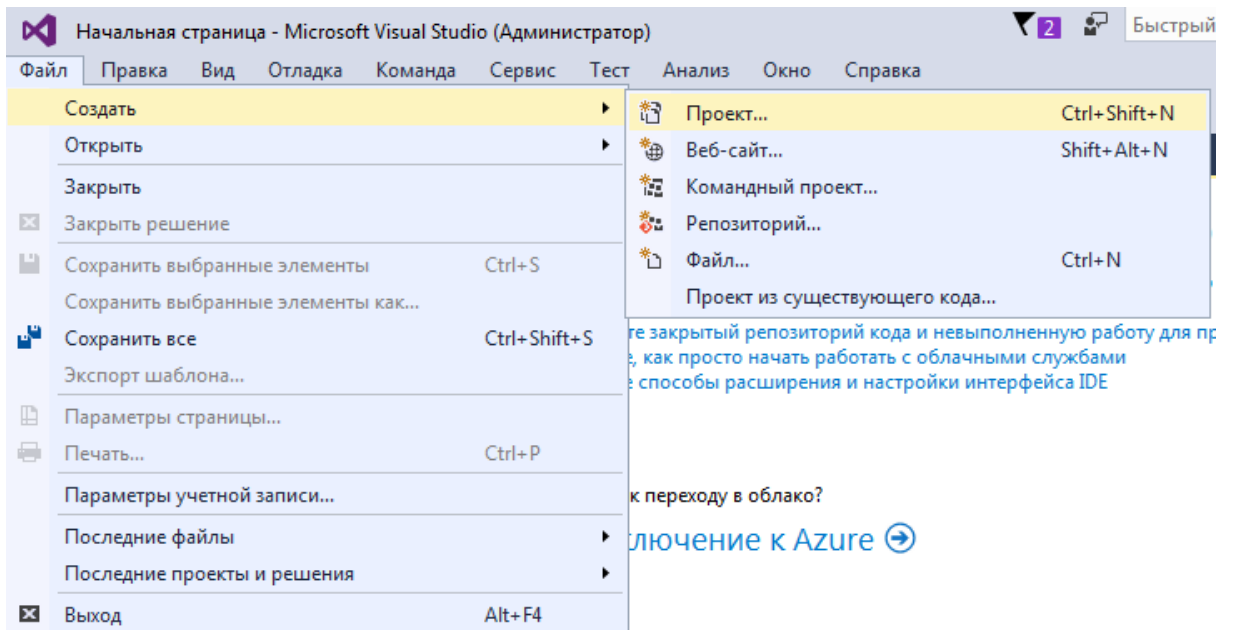
```

Таким образом в F# можно использовать как «закрытые» алгебраические типы так и «открытую» к расширению реализацию на основе интерфейса и наследуемых классов.

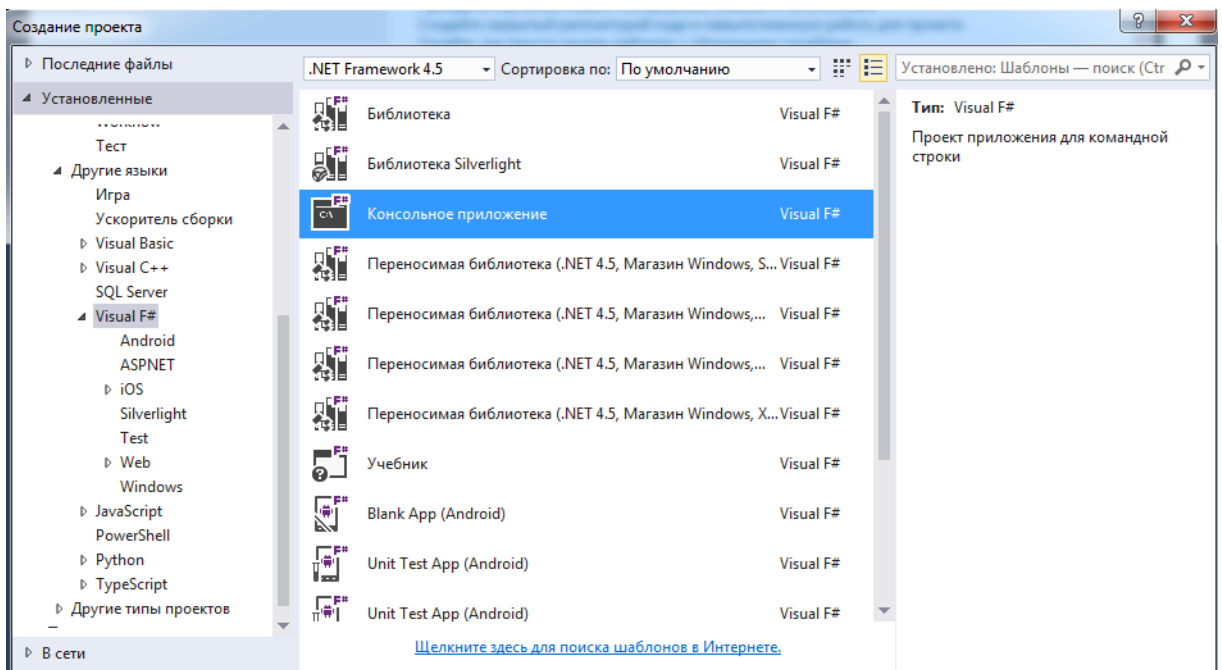
3 Основы языка F#

3.1 Создание консольного проекта в F#

В Visual Studio необходимо выбрать пункт меню «Создать проект»



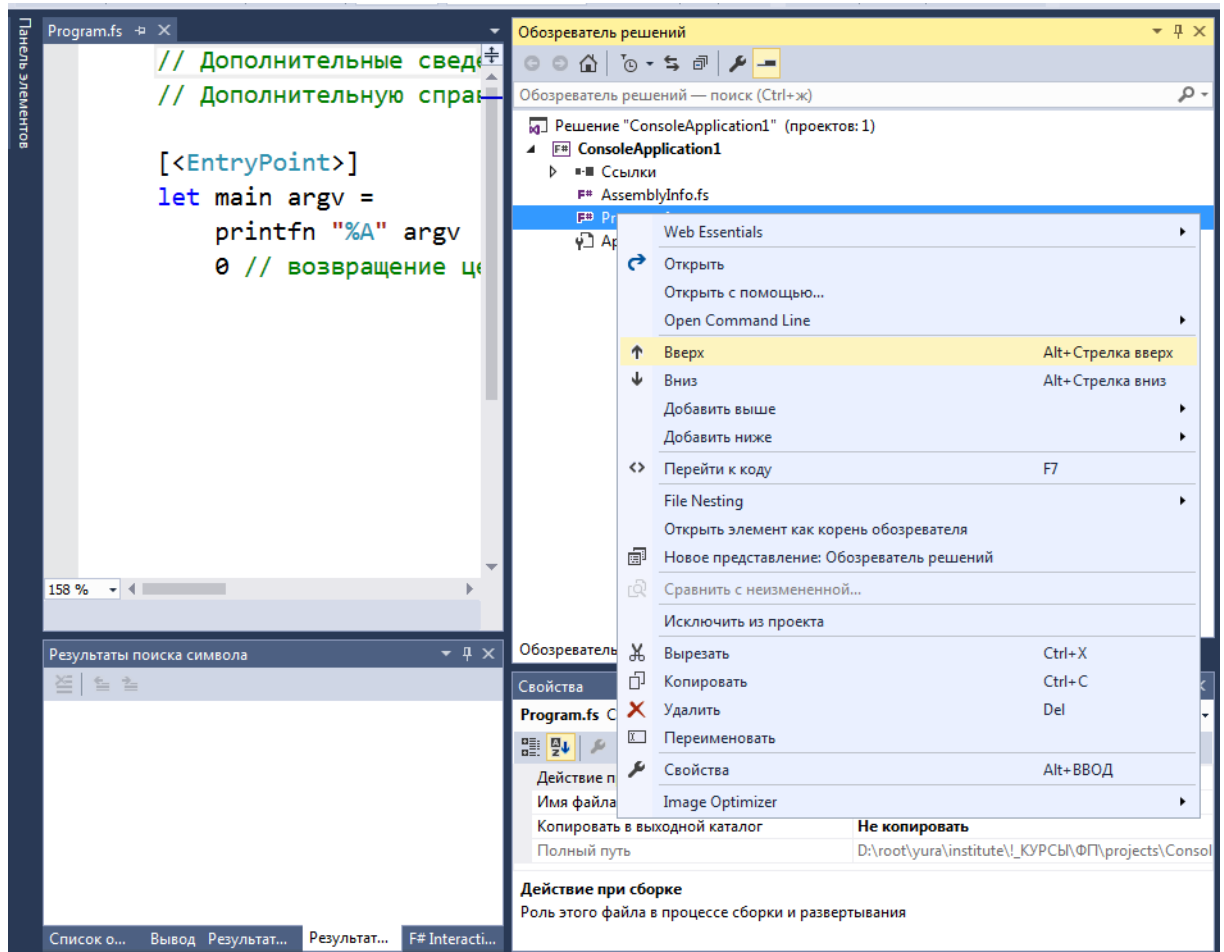
И выбрать консольное приложение F#.



Необходимо обратить внимание на особенность F# связанную с порядком компиляции файлов.

В F# файлы компилируются последовательно в том порядке, в котором они указаны в проекте. Элементы, расположенные в файле (типы, функции и т.д.) могут ссылаться только на элементы расположенные выше в списке компиляции.

Поэтому в дереве проекта есть специальные пункты меню, чтобы передвинуть файл вверх или вниз в списке компиляции как показано на следующем рисунке.



По умолчанию в консольном проекте создается один файл F# содержащий функцию main.

3.2 Комментарии в F#

`//` строчный комментарий

`(*`
 блочный комментарий
`*)`

`///` Аналог XML-комментария перед объявлениями типов, переменных, функций

3.3 Основные типы данных

При указании константы определенного типа необходимо использовать постфикс, соответствующий типу (для некоторых типов постфикс может быть пустым). Поэтому явное указание типа переменной не требуется, компилятор F# выводит тип переменной по постфиксу.

Основные типы данных представлены в таблице:

Наименование	Описание	Пример C#	Пример F#
Целочисленные			
short	Целое 16-битное число со знаком	<code>short short1 = 123;</code>	<code>let short1 = 123s let short2:int16 = 123s</code>
int	Целое 32-битное число со знаком	<code>int int1 = 333;</code>	<code>let int1 = 333 let int2:int = 333</code>
long	Целое 64-битное число со знаком	<code>long long1 = 333;</code>	<code>let long1 = 333L let long2:int64 = 333L</code>
byte	Целое 8-битное число без знака	<code>byte byte1 = 123;</code>	<code>let byte1 = 123uy let byte2:byte = 123uy //Двоичный код let byte3 = 0b1111000uy</code>
sbyte	Целое 8-битное число со знаком	<code>sbyte sbyte1 = -123;</code>	<code>let sbyte1 = -123y let sbyte2:sbyte = -123y</code>
С плавающей точкой			
float	Число с плавающей точкой 32-битное одинарной точности, около 7 значащих цифр после запятой	<code>float float1 = 123.123F; 123.123F;</code>	<code>let float1 = 123.123F let float2:float32 = 123.123F</code>
double	Число с плавающей точкой 64-битное двойной точности, около 15 значащих цифр после запятой	<code>double double1 = 123.123; 123.123;</code>	<code>let double1 = 123.123 let double2:double = 123.123</code>
decimal	Число с плавающей точкой 128-битное повышенной точности, около 28 значащих цифр после запятой	<code>decimal decimal1 = 123.123m; 123.123m;</code>	<code>let decimal1 = 123.123m let decimal2:decimal = 123.123m</code>
Логический тип			
bool	Может принимать значения true или false	<code>bool bool1 = true;</code>	<code>let bool1 = true let bool2:bool = true</code>
Символьный и строковый типы			

char	Символ Unicode	<code>char char1 = 'A';</code>	<code>let char1 = 'A'</code> <code>let char2:char = 'A'</code>
string	Строка символов Unicode	<code>string string1 =</code> <code>"ABC";</code>	<code>let string1 = "ABC"</code> <code>let string2:string = "ABC"</code>

3.4 Операторы ввода-вывода

В F# можно использовать консольный класс C#. Пример:

```
Console.WriteLine("{0:d}", 123);
let str = Console.ReadLine();
```

Для этого необходимо сослаться на пространство имен System с помощью команды

```
open System
```

которая является аналогом команды using в C#.

Также можно использовать оператор вывода F# printfn.

Формат команды:

printfn “строка формата” список аргументов через пробел

Строка формата может включать следующие элементы для форматирования данных различных типов:

%s – строка;

%b – логическое значение;

%i – целое число;

%f – число с плавающей точкой;

%A – форматирование сложных объектов, кортежей.

Пример:

```
printfn "%i - %f - %s" 333 123.123 "string1"
```

Результат вывода в консоль:

```
333 - 123.123000 - string1
```

3.5 Объявление и неизменяемость переменных

Переменные и функции в языке F# объявляются с помощью ключевого слова let.

По умолчанию переменные в F# являются неизменяемыми.

Переменная рассматривается как частный случай функции, которая не принимает параметров и возвращает константу заданного типа (тип переменной фактически соответствует типу возвращаемого значения функции).

```
let p1 = 1
//ошибка, невозможно изменить значение переменной
//p1 = p1 + 1
```

Возможно объявить изменяемую переменную:

```
//Объявление изменяемой переменной
//и присвоение ей начального значения
let mutable p2 = 1
//Изменение значения переменной
p2 <- p2 + 1
```

В этом случае « = » используется для присвоения начального значения, а « <- » для изменения значения (аналог оператора присваивания).

Необходимо отметить, что чистые функциональные языки вообще не предполагают изменяемых переменных, так как это нарушает чистоту языка и создает побочные эффекты:
https://ru.wikipedia.org/wiki/Чистота_языка_программирования

3.6 Использование кортежей

При изучении C# мы познакомились с типом Tuple (кортеж), который применяется достаточно редко.

В функциональном программировании и в частности в языке F# кортежи являются одним из основных типов данных.

Кортеж примерно соответствует одной записи в реляционной таблице и может состоять из нескольких полей.

```
//Кортеж
```

```
let tuple1 = (123, 123.123, "string1")
//Кортеж с явным указанием типов данных
let tuple2:int*double*string = (123, 123.123, "string1")
```

Кортежи могут быть вложенными:

```
//Кортеж кортежей
let tuple3 = ((123, 123.123, "string1"),(124, 124.123,
"string2"))
//Кортеж кортежей с явным указанием типов данных
let tuple4:((int*double*string)*(int*double*string)) =
((123, 123.123, "string1"),(124, 124.123, "string2"))
```

```
//Получение значения кортежей
let (i1, d1, str1) = tuple1
printfn "%A" i1
printfn "%A" d1
printfn "%A" str1
```

Результат вывода:

```
123
123.123
"string1"
```

Если какое-то значение кортежа не нужно, то вместо него используют символ подчеркивания:

```
//Получение значения второго элемента кортежа
//первый и третий элементы игнорируются
let (_, d1, _) = tuple1
```

Примеры функций, которые возвращают значения первого и второго элементов кортежа (для кортежа из двух элементов).

```
let first(a,_) = a
let second(_,b) = b

//Передача аргументов с помощью скобок
let c = first((1,2))
//Передача аргументов с помощью пробела
let d = second (1,2)
```

3.7 Создание функций

3.7.1 Создание простых функций

Функции, как и переменные, создаются с помощью ключевого слова `let`. Вместо фигурных скобок используются отступы.

```
//Функция принимает 2 параметра int
//и возвращает int
//= отделяет объявление функции от реализации
let sum1(a:int, b:int):int =
    //создание временной переменной
    let result = a + b
    //возвращение результата
    //результат указывается в виде выражения
    //ключевое слово return не используется
    result
```

```
///Вариант функции, записанный в одну строку
let sum2(a,b) = a + b
//Пример вызова функции
let q = sum2(1,2)
```

Комментарий `///` выдается в подсказке IntelliSense:

```
///Вариант функции, записанный в одну строку
let sum2(a,b) = a + b
//Пример вызова функции
let q = sum2(1,2)
```

```
sum2(int, int) : int
Вариант функции, записанный в одну строку
```

3.7.2 Возврат кортежей из функций

Что делать в том случае, если функция должна вернуть несколько значений? В C# для этих целей использовались out-параметры. В F# нет необходимости их использовать, так как функция может вернуть составное значение в виде кортежа.

```
///Функция возвращает число, его квадрат и куб
let Powers(x) =
```

```
let x2 = x*x
let x3 = x*x*x
(x, x2, x3)
```

```
///Вариант функции Powers записанный в одну строку
let PowersInline(x) = (x, x*x, x*x*x)
```

3.7.3 Рекурсивные функции

Для объявления рекурсивных функций в F# необходимо использовать ключевое слово `rec`.

```
///Рекурсивная функция для вычисления факториала
//данная функция может быть записана в одну строку
let rec factorial(n:int):int =
    if n<=1 then 1
    else n*factorial(n-1)
//пример вызова
let q3 = factorial(5)
```

3.7.4 Взаимно-рекурсивные функции и эмуляция конечного автомата

Рассмотрим следующий пример на языке C#:

```
static void loop1()
{
    for(int i=1;i<=10;i++)
    {
        if(i<=3)
        {
            Console.WriteLine("{0:d} - (+1) {0:d}", i, i+1);
        }
        else if(i<=6)
        {
            Console.WriteLine("{0:d} - (^2) {0:d}", i, i * i);
        }
        else
        {
            Console.WriteLine("{0:d} - (^3) {0:d}", i, i * i * i);
        }
    }
}
```

Результат работы функции:

```
1 - (+1) 2
```


2 - (+1) 3

3 - (+1) 4

4 - (\wedge^2) 16

5 - (\wedge^2) 25

6 - (\wedge^2) 36

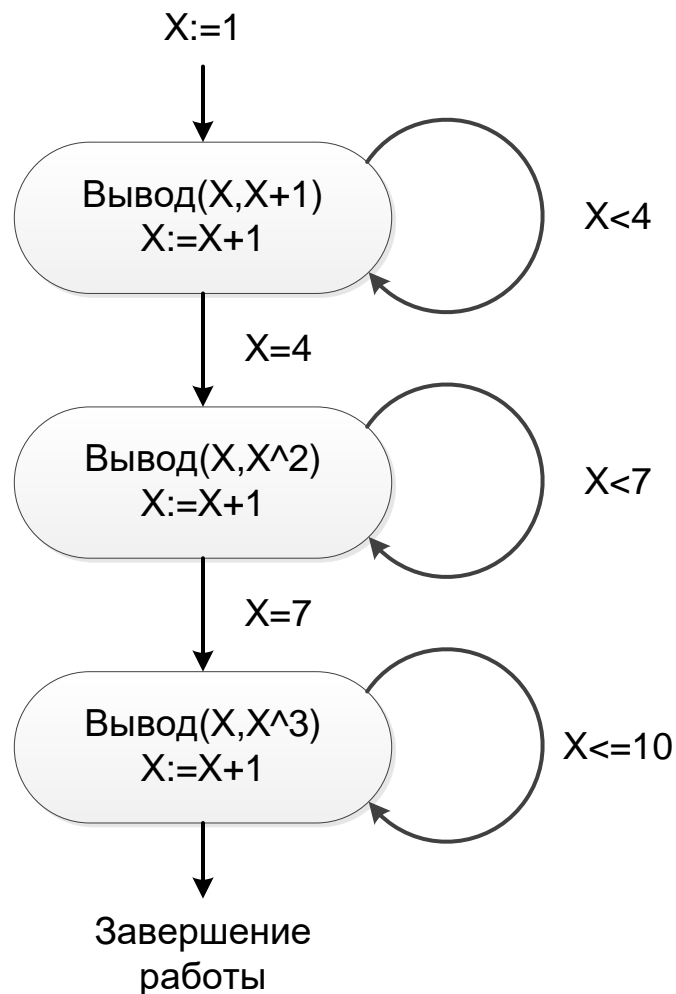
7 - (\wedge^3) 343

8 - (\wedge^3) 512

9 - (\wedge^3) 729

10 - (\wedge^3) 1000

Приведенный код на C# не является сложным. Но данный код в императивном стиле не совсем естественным образом эмулирует следующий конечный автомат (https://ru.wikipedia.org/wiki/Конечный_автомат).



Автоматное программирование часто выделяют в отдельную парадигму и применяют как в функциональных, так и в императивных языках – https://ru.wikipedia.org/wiki/Автоматное_программирование

На императивных языках (в том числе на C#) также можно писать программы в автоматном стиле.

Данный конечный автомат можно эмулировать с помощью трех функций State1, State2 и State3, которые соответствуют состояниям автомата. Переходы между состояниями могут быть эмулированы путем вызова функций друг другом, вызов, в том числе, может быть рекурсивным.

Запишем данную программу в автоматном стиле:

```
static void State1(int x)
{
    Console.WriteLine("{0:d} - (+1) {1:d}", x, x + 1);
    int x_next = x + 1;
    if (x_next > 3) State2(x_next);
    else State1(x_next);
}
static void State2(int x)
{
    Console.WriteLine("{0:d} - (^2) {1:d}", x, x * x);
    int x_next = x + 1;
    if (x_next > 6) State3(x_next);
    else State2(x_next);
}
static void State3(int x)
{
    Console.WriteLine("{0:d} - (^3) {1:d}", x, x * x * x);
    int x_next = x + 1;
    if (x_next <= 10) State3(x_next);
}

//Вызов с начальным условием
State1(1);
```

Результат работы программы таков же, как и в предыдущем случае:

1 - (+1) 2

2 - (+1) 3

3 - (+1) 4

4 - (^2) 16
 5 - (^2) 25
 6 - (^2) 36
 7 - (^3) 343
 8 - (^3) 512
 9 - (^3) 729
 10 - (^3) 1000

Необходимо отметить, что в императивных языках автоматная техника применяется достаточно редко, тогда как в функциональных языках она является классической.

Рассмотрим пример реализации данного конечного автомата на языке F#:

```
//Определение рекурсивной функции
let rec State1(x:int) =
    printfn "%i - (+1) %i" x (x+1)
    let x_next = x+1
    if x_next>3 then State2(x_next)
    else State1(x_next)
//ключевое слово and объявляет функцию
//как взаимно-рекурсивную к предыдущей
//это необходимо так как функция
//State1 ссылается на State2 и F#
//не допускает опережающего описания
and State2(x:int) =
    printfn "%i - (^2) %i" x (x*x)
    let x_next = x+1
    if x_next>6 then State3(x_next)
    else State2(x_next)
and State3(x:int) =
    printfn "%i - (^3) %i" x (x*x*x)
    let x_next = x+1
    if x_next<=10 then State3(x_next)
//Вызов с начальным условием
State1(1)
```

Результат работы программы:

1 - (+1) 2

2 - (+1) 3

3 - (+1) 4

4 - (^2) 16

5 - (^2) 25

6 - (^2) 36

7 - (^3) 343

8 - (^3) 512

9 - (^3) 729

10 - (^3) 1000

Возможно данный код покажется на первый взгляд более громоздким чем неавтоматный вариант на C#. Но необходимо отметить, что он естественным образом эмулирует конечный автомат, каждая функция соответствует состоянию автомата, а вызовы функций (в том числе рекурсивные) соответствуют переходам между состояниями.

В случае сложного автомата попытка свернуть его в цикл, скорее всего, будет менее интуитивно понятна.

Необходимо отметить, что чистые функциональные языки вообще не предполагают использования циклов, вместо циклов используются рекурсивные вызовы функций.

3.7.5 Хвостовая рекурсия

https://ru.wikipedia.org/wiki/Хвостовая_рекурсия (tail recursion) предполагает что рекурсивная функция накапливает вычисляемое значение в отдельном параметре, который часто называют аккумулятором.

Название «хвостовая рекурсия» связано с тем, что, как правило, рекурсивный вызов происходит в конце функции, то есть последняя строка («хвост») функции рекурсивно вызывает сама себя.

Преимущество хвостовой рекурсии состоит в том, что при компиляции аккумулятор можно заменить на переменную цикла, что

позволяет генерировать оптимизированный код. (Ведь генерируемый машинный код является императивным. На уровне машинного кода цикл будет выполняться намного быстрее, чем рекурсивный вызов.)

Но проблема состоит в том, что далеко не каждый рекурсивный вызов можно преобразовать в хвостовую рекурсию, формального алгоритма для этого не существует.

Рассмотрим пример традиционной рекурсии для вычисления факториала.

```
///Рекурсивная (не хвостовая) функция для вычисления факториала
//аккумулятор отсутствует
let rec factorial1 n = if n<=1 then 1 else n*factorial(n-1)
```

Пример вызова функции для n=3.

n	Возвращаемое значение	Накопленное вычисление
3	3*factorial(2)	3*factorial(2)
2	2*factorial(1)	3*2*factorial(1)
1	1	3*2*1

Для хранения накопленных вычислений, как правило, используется системный стек. После завершения цепочки рекурсивных вызовов значения 3*2*1 последовательно выбираются из стека и вычисляются.

Косвенным признаком отсутствия хвостовой рекурсии является то, что рекурсивно вычисляемая функция входит в выражение как операнд – n*factorial(n-1).

В хвостовой рекурсии вычисления производятся, как правило, в операндах рекурсивно вызываемой функции:

```
///Функция для вычисления факториала на основе
хвостовой рекурсии
let rec fact_tr(n:int, acc:int):int =
    if n=1 then acc
    else fact_tr(n-1, n*acc)
```

```

///Обертка для сокрытия хвостовой рекурсии
let rec factorial2 n = fact_tr(n,1)
//пример вызова
let q5 = factorial2(3)

```

Пример вызова функции `fact_tr` для $n=3$.

Входные параметры		Возвращаемое значение
n	acc	
3	1	fact_tr(3-1, 3*1) fact_tr(2, 3)
2	3	fact_tr(2-1, 3*2) fact_tr(1, 6)
1	6	6

Отметим, что характерным признаком хвостовой рекурсии является отсутствие накопленных вычислений, так как вычисления производятся в операндах вызываемой функции. Поэтому нет необходимости накапливать данные в стеке, и хвостовая рекурсия может быть преобразована в цикл, где роль переменной цикла выполняет аккумулятор.

Большинство современных компиляторов с функциональных языков программирования гарантируют генерацию оптимизированного кода в случае хвостовой рекурсии.

3.7.6 Каррирование

Каррирование является исключительно важной техникой ФП – <https://ru.wikipedia.org/wiki/Каррирование>

Каррирование позволяет преобразовать функцию, которая принимает n входных параметров к n функциям, которые принимают 1 параметр.

Рассмотрим пример функции, которая возвращает сумму трех целочисленных аргументов. Это «искусственный» пример каррирования на языке C# с использованием обобщенного делегата Func.

```
//Пример каррирования
//Исходная функция принимает 3 аргумента и возвращает их сумму
Func<int, int, int, int> step0 = (a, b, c) => a + b + c;
//Первый шаг каррирования
//функция принимает один аргумент и возвращает функцию
//которая принимает остальные аргументы и возвращает результат
//функцию (b, c) => a + b + c
Func<int, Func<int, int, int>> step1 = a => (b, c) => a + b + c;
//ОТМЕТИМ что функция (b, c) => a + b + c является выходным
результатом step1
//Второй шаг каррирования
Func<int, Func<int, Func<int, int>>> step2 = a => b => c => a + b +
c;
//Таким образом исходная функция (a, b, c) => a + b + c;
//в которой параметры передаются через кортеж преобразована в набор
вложенных функций
//a => b => c => a + b + c или a => (b => (c => a + b + c))

//При вызове полный набор аргументов можно передавать в виде набора
скобочек
int res1 = step2(1)(2)(3);
//и можно применять параметры частично
var res2 = step2(1);
```

(локальная переменная) Func<int, Func<int, int>> res2

```
int res2_1 = res2(2)(3);
var res3 = step2(1)(2);
```

(локальная переменная) Func<int, int> res3

```
int res3_1 = res3(3);
```

Данный пример показывает, что каррирование тесно связано с частичным применением. Если параметры функции каррированы, то применение одного из параметров порождает функцию с меньшим количеством параметров и тем же результатом.

Отметим, что эффекта частичного применения нельзя добиться, используя некаррированные параметры. Для одного аргумента можно использовать подстановку но это действие нельзя выполнять «в цепочку»:

```
Func<int, int, int> uncarry1 = (b, c) => step0(1, b, c);
int uncarry1_res1 = uncarry1(2,3);
//var uncarry1_res2 = uncarry1(2); - ОШИБКА
```

Рассмотрим аналогичный пример на языке F#. Данный пример приведен только для понимания, так как каррирование встроено в F# с помощью специальных механизмов. Также данный пример использует лямбда-выражения и аналог делегатного типа, о которых пойдет речь далее в этой главе. Символ «*» означает склеивание параметров в кортеж, а символ «->» каррирование параметров или возвращаемое значение. Для объявления лямбда-выражений используется ключевое слово `fun` и стрелка «->».

```
let step0:(int*int*int->int) = fun (a,b,c) -> a + b + c
let step1:(int->(int*int->int)) = fun a -> fun (b,c) -> a + b + c
let step2:(int->(int->(int->int))) = fun a -> fun b -> fun c -> a + b + c
let res1:int = step2(1)(2)(3)
let res2 = step2(1)
```

```
val res2 : (int -> int -> int)
```

```
let res2_1:int = res2(2)(3)
let res3 = step2(1)(2)
let res3_1:int = res3(3)
```

Главным преимуществом функциональных языков является то, что каррирование в них автоматически производится компилятором, не необходимости в «ручном» каррировании как в показанном примере.

В некоторых ФП-языках все параметры по умолчанию каррированы. В F# можно явно указывать каррированы параметры или нет.

В F# существует два способа передачи параметров:

- Параметры передаются традиционным образом в виде набора аргументов. В этом случае говорят, что передается кортеж параметров, так как в F# и других функциональных языках для передачи таких параметров используются кортежи.
- Параметры передаются в каррированном виде. F# автоматически генерирует все необходимые промежуточные функции. Прикладной программист может считать, что он

работает с одной внешней функцией, которая принимает «цепочку» параметров. Для такой функции работает механизм частичного применения параметров.

- Возникает вопрос – возможен ли «гибридный» способ, когда часть параметров каррирована, а часть нет. Теоретически такой способ возможен, если один из каррированных параметров является кортежем. Но необходимо отметить, что в функциональных языках каррированные параметры предпочтительны, так как обеспечивают частичное применение. Например, в языке Haskell все параметры по умолчанию каррированы.

Пример некаррированного объявления – набор параметров передается в виде кортежа:

```
///пример некаррированной функции
let uncarry1(a: int, b: int, c:int) = a + b + c
let q6 = uncarry1(1,2,3)
```

```
val uncarry1 : (int * int * int -> int)
```

```
пример некаррированной функции
```

Набор входных параметров считается кортежем, символ «*» соединяет элементы кортежа.

Пример каррированного объявления – набор параметров передается в виде отдельных элементов, символ «->» означает применение функции:

```
///пример каррированной функции 1
let carry1(a: int)(b: int)(c:int) = a + b + c
let q71 = carry1(1)(2)(3)
let q72 = carry1 1 2 3
```

```
val carry1 : (int -> int -> int -> int)
```

```
пример каррированной функции 1
```

При каррированном объявлении функции цепочка преобразований функции от n аргументов к n функциям от 1 аргумента выполняется автоматически. Отметим, что стрелки « \rightarrow » при объявлении каррированной функции интуитивно напоминают стрелки лямбда-выражений из приведенного выше примера.

При вызове функции для разделения аргументов можно использовать как скобки, так и пробелы.

При объявлении каррированной функции вместо скобок также можно использовать пробелы.

```
///пример каррированной функции 2
let carry2 a b c = a + b + c
let q81 = carry2(1)(2)(3)
let q82 = carry2 1 2 3
```

```
val carry2 : (int -> int -> int -> int)
```

```
пример каррированной функции 2
```

Для каррированных функций может быть реализовано частичное применение параметров:

```
let carry1withFixedParams(a: int) = carry1 1 2 a
let q9 = carry1withFixedParams(3)
```

```
val carry1withFixedParams : (int -> int)
```

В функции `carry1withFixedParams` производится вызов функции `carry1` с фиксированными параметрами 1 и 2, а третий параметр является параметром функции `carry1withFixedParams`.

Для параметров-кортежей попытка подобного объявления приведет к ошибке:

```
let uncarry1withFixedParams(a: int) = uncarry1 1 2 a
```

```
Данное значение не является функцией и не может быть применено
```

Интересной особенностью функциональных языков является возможность опускать формальные параметры функций, если они не используются. Например, предыдущий пример может быть переписан в следующем виде:

```
let carry1withFixedParams(a: int) = carry1 1 2 a
let q9 = carry1withFixedParams(3)
```

```
val carry1withFixedParams : (int -> int)
```

```
let carry1withFixedParams2 = carry1 1 2
```

```
val carry1withFixedParams2 : (int -> int)
```

Не смотря на отсутствие формального параметра (a:int) во втором случае, у обеих функций совпадают сигнатуры.

Данная особенность называется по названию греческой буквы η -эта (ита) η -редукцией (эта-редукцией или эта-конверсией).

В частности возможно следующее объявление:

```
let Sin1 = Math.Sin
let Sin1Res = Sin1(0.5)
```

Компилятор «понимает» что аргумент функции Sin1 совпадает с Math.Sin и добавляет параметр функции автоматически.

3.7.7 Лямбда-выражения

Объявление лямбда-выражений очень напоминает объявление в C#. Для объявления используется ключевое слово fun и стрелка «->».

Пример объявления лямбда-выражения с некаррированными параметрами:

```
let lambda1 = fun (a: int, b: int, c:int) -> a + b + c
let q10 = lambda1(1,2,3)
```

```
val lambda1 : (int * int * int -> int)
```

Пример объявления лямбда-выражения с каррированными параметрами:

```
let lambda2 = fun (a: int)(b: int)(c: int) -> a + b + c
let q11 = lambda2 1 2 3
```

```
val lambda2 : (int -> int -> int -> int)
```

Необходимо отметить, что лямбда-выражения в F# поддерживают замыкания (closure), то есть позволяют внутри лямбда-выражения использовать переменные из внешнего контекста:

```
let context1 = 300
let lambdaWithClosure = fun x -> context1 + x
let lambda3res = lambdaWithClosure 33
```

```
lambda3res 333
```

Также необходимо отметить, что замыкания лучше применять только в случае необходимости, все параметры лучше передавать в функцию явным образом.

3.7.8 Аналог делегатного типа в F#

Для того чтобы передать функцию в качестве параметра в C# используются делегаты. В F# также возможно использование делегатов для взаимодействия с C#.

Но в функциональных языках программирования аналог делегатного типа (так называемые функциональные значения – function values) «встроены» в язык. Аналог делегатов может быть описан с использованием символов «*» и «->».

Пример функции, которая принимает следующие параметры:

- два целых числа
- функцию, которая принимает два целых числа в виде кортежа, производит с ними действия и возвращает результат в виде целого числа:

```
let del1(a:int, b:int, func1: int*int->int) = func1(a,b)
let del1call = del1(1, 2, fun(a,b)->a+b) // результат - 3
```

```
val del1 : (int * int * (int * int -> int) -> int)
```

Пример функции, которая принимает следующие параметры:

- два целых числа
- функцию, которая принимает два целых числа в каррированном виде, производит с ними действия и возвращает результат в виде целого числа:

```
let del2(a:int, b:int, func1: int->int->int) = func1 a b
let del2call = del2(5, 2, fun a b -> a-b) // результат - 3
```

```
val del2 : (int * int * (int -> int -> int) -> int)
```

3.7.9 Обобщенные типы в функциях

Если типы параметров в функции не указаны, то типы автоматически считаются обобщенными.

```
let generic1(a,b) = (a,b)
```

```
val generic1 : ('a * 'b -> 'a * 'b)
```

В данном примере «'a» и «'b» – обобщенные типы. Обобщенные типы в F# принято записывать с апострофом слева.

Компилятор F# пытается обобщить параметры функций насколько это возможно. Этот принцип называется автоматическим обобщением (automatic generalization).

Приведенную выше функцию можно вызывать для любых типов данных:

```
let generic1_1 = generic1(123,123.123)
let generic1_2 = generic1("123",123.123)
```

Можно явным образом указать обобщенный тип:

```
let generic2(a:'T,b:'T) = (a,b)
```

```
val generic2 : ('T * 'T -> 'T * 'T)
```

```
let generic2_1 = generic2(123,45)
```

```
let generic2_2 = generic2("123", "123.123")
```

В приведенном примере оба параметра должны быть одного типа, но тип является обобщенным. Следующие примеры вызывают ошибку, так как тип второго параметра не совпадает с типом первого параметра:

```
//Ошибка - типы параметров различаются
```

```
let generic2_3 = generic2(123,123.123)
```

```
В данном выражении требовалось наличие типа
int
, но получен тип
float
```

```
//Ошибка - типы параметров различаются
```

```
let generic2_3 = generic2("123",123.123)
```

```
В данном выражении требовалось наличие типа
string
, но получен тип
float
```

Если вычисления, производимые в функции, добавляют ограничение на тип, то данное ограничение компилятор автоматически учитывает при выводе типа. Например, в следующем примере ко второму параметру прибавляется константа типа float, поэтому компилятор автоматически определяет тип второго параметра как float. Ограничений на тип первого параметра не вводится, поэтому он остается обобщенным.

```
let generic3(a,b) = (a,b+100.0)
```

```
val generic3 : ('a * float -> 'a * float)
```

Подобные ограничения могут быть в F# неожиданными. Например, можно ожидать, что следующая функция будет обобщенной, но неожиданно компилятор определяет ее параметры как целочисленные:

```
let sum_int(a,b) = a + b
```

```
val sum_int : (int * int -> int)
```

Результат кажется странным, ведь очевидно оператор «+» перегружен для всех числовых типов. Но из-за ограничений компилятора при выводе типа данный оператор вносит ограничение – аргументы считаются целочисленными. Это ограничение выполняется для всех арифметических операторов: «+, -, *, /» (для целых чисел выполняется целочисленное деление).

Тип данных можно указать явно и перегруженный оператор будет работать также для других типов:

```
let sum_float(a:float,b) = a + b
```

```
val sum_float : (float * float -> float)
```

Интересно, что в данном примере достаточно указать тип данных только у первого аргумента, тип второго выводится автоматически, так как они складываются в теле функции.

Но как же бороться с ограничениями, вносимыми арифметическими операторами. Для этого рекомендуется создавать обобщенные функции в несколько этапов. На первом этапе создается функция, которая не вносит ограничений на типы.

```
let sum_generic(a,b,sum_func) = sum_func(a,b)
```

```
val sum_generic : ('a * 'b * ('a * 'b -> 'c) -> 'c)
```

Здесь обобщенная функция `sum_func` применяется к обобщенным аргументам `a` и `b`.

На втором этапе создаются функции, которые уточняют первую функцию с помощью ограничений для конкретных типов.

Оператор «+» определяет ее параметры как целочисленные:

```
let sum_int(a,b) = sum_generic(a,b,fun(a,b)->a+b)
```

```
val sum_int : (int * int -> int)
```

Добавление слагаемого 0.0 задает параметры типа float:

```
let sum_float(a,b) = sum_generic(a,b,fun(a,b)->a+b+0.0)
```

```
val sum_float : (float * float -> float)
```

Добавление слагаемого "" задает параметры типа string:

```
let sum_string(a,b) = sum_generic(a,b,fun(a,b)->a+b+"")
```

```
val sum_string : (string * string -> string)
```

Примеры вызовов функций:

```
let sum_int_res = sum_int(1,2)
let sum_float_res = sum_float(1.0, 2.0)
let sum_string_res = sum_string("str1", "str2")
```

Также необходимо учитывать, что в F# не работает классическая перегрузка для функций – нельзя определить функцию с одинаковым именем и различными сигнатурами.

```
//Первое определение функции sum_num
let sum_num(a:int,b) = a + b
//Второе определение не приводит к ошибке,
//но перекрывает первое
let sum_num(a:float,b) = a + b
//Нет ошибки - параметры типа float
let sum_num1 = sum_num(1.0,2.0)
//Ошибка - параметры типа int
let sum_num2 = sum_num(1,2)
```

```
В данном выражении требовалось наличие типа
float
, но получен тип
int
```


3.8 Коллекции

Так как F# является языком платформы .NET, то в нем возможно использовать стандартные изменяемые коллекции .NET, например списки и словари.

Пример использования списка .NET:

```
let lst1 = new System.Collections.Generic.List<int>()
lst1.Add(1)
lst1.Add(2)
lst1.Add(3)
for x in lst1 do
    printfn "%i" x
```

Пример использования словаря .NET::

```
let dict1 = new
System.Collections.Generic.Dictionary<string, int>()
dict1.Add("str1", 1)
dict1.Add("str2", 2)
dict1.Add("str3", 3)
for x in dict1 do
    printfn "%s - %i" x.Key x.Value
```

Но в F# также имеются неизменяемые коллекции, которые рекомендуется использовать в соответствии с практикой функционального программирования.

Для обработки таких коллекций, как правило, используют рекурсию.

3.8.1 Списки

Списки в F# являются неизменяемыми, в них нельзя динамически добавить элемент.

Примеры объявления списков:

```

//Перечисление элементов списка
let list1 = [1;2;3]
//Явное указание типа
let list2:int list = [1;2;3]
//Диапазон значений
let list3 = [1..10]
//Использование оператора :: (cons)
let list4 = 1::2::3::[]
//Использование генератора списка (list comprehension)
let list5 = [for x in [1..5] do yield x*x]

```

Index	Value
[0]	1
[1]	4
[2]	9
[3]	16
[4]	25

С помощью генератора списков можно генерировать не только линейные списки но и более сложные конструкции. В следующем примере генерируется список кортежей:

```

//Список кортежей
let list6 = [for x in [1..5] do yield (x, x*x)]

```

Index	Value
[0]	{(1, 1)}
[1]	{(2, 4)}
[2]	{(3, 9)}
[3]	{(4, 16)}
[4]	{(5, 25)}

Внутри генератора списков можно использовать условия (и даже сопоставление с образцом) что делает генератор универсальной конструкцией для генерации списков.

```
//Использование условий
let list6_1 = [for x in [1..10] do if x % 2 = 0 then yield (x, x*x) else yield (x, 0)]
```

Index	Value
[0]	{(1, 0)}
[1]	{(2, 4)}
[2]	{(3, 0)}
[3]	{(4, 16)}
[4]	{(5, 0)}
[5]	{(6, 36)}
[6]	{(7, 0)}
[7]	{(8, 64)}
[8]	{(9, 0)}
[9]	{(10, 100)}

Для конкатенации списков используется оператор « @ ».

```
//Конкатенация списков
let list12 = list1 @ list2
```

Index	Value
[0]	1
[1]	2
[2]	3
[3]	1
[4]	2
[5]	3

Оператор « :: » cons используется для конкатенации элемента к списку слева.

```
let list4_1 = 4::[1;2;3]
```

Отметим, что аналогичный оператор для правой конкатенации отсутствует, так как он не имеет смысла – рекурсивная обработка списков происходит слева. Но возможно использование оператора « @ » следующим образом:

```
let list4_2 = [1;2;3] @ [4]
```

Для удобства рекурсивной обработки списков у списка есть поля Head (первый элемент списка) и Tail (оставшаяся часть списка).

```

///Рекурсивная обработка списка
let rec PlusOne(lst:int list):int list =
    //Если входной список пуст, то возвращается пустой список
    if lst.IsEmpty then []
    //иначе к первому элементу прибавляется 1
    //и функция рекурсивно вызывается для
    //оставшейся части списка
    else (lst.Head+1)::PlusOne(lst.Tail)
//Вызов функции
let PlusOneRes = PlusOne([1..3])

```

PlusOneRes Length = 3	
[0]	2
[1]	3
[2]	4
▶ Базовое представление	

В F# как и в большинстве ФП-языков в функциях можно использовать сопоставление с образцом. Так предыдущий пример можно переписать без использования оператора if с использованием сопоставления с образцом на уровне функции:

```

///Рекурсивная обработка списка (сопоставление с образцом)
let rec PlusOne1 = function
    | [] -> []
    | x::xs -> x+1::PlusOne1(xs)
//Вызов функции
let PlusOne1Res = PlusOne1([1..3])

```

PlusOne1Res Length = 3	
[0]	2
[1]	3
[2]	4
▶ Базовое представление	

В сопоставлении с образцом список часто представляют в виде $x::xs$, где x соответствует голове списка (один элемент), а xs хвосту списка (оставшаяся часть списка).

3.8.2 Массивы

Массивы в F# являются изменяемыми, в них можно динамически добавить элементы. Как правило, используются реже списков.

Примеры объявления массивов:

```

//Перечисление элементов
let arr1 = [|1;2;3|]
//Явное указание типа
let arr2:int array = [|1;2;3|]
//Диапазон значений
let arr3 = [|1..10|]
//Использование генератора массива (array comprehension)
let arr4 = [|for x in [1..5] do yield x*x|]

```

Index	Value
[0]	1
[1]	4
[2]	9
[3]	16
[4]	25

```

//Массив кортежей
let arr5 = [|for x in [1..5] do yield (x, x*x)|]
//Обращение к элементам массива по индексу осуществляется через точку
let item0 = arr1.[0]
//Возможно использование срезов по индексам
let items1 = arr4.[1..]

```

Index	Value
[0]	4
[1]	9
[2]	16
[3]	25

```
let items2 = arr4[..3]
```

Index	Value
[0]	1
[1]	4
[2]	9
[3]	16

```
let items3 = arr4.[1..3]
```

Index	Value
[0]	4
[1]	9
[2]	16

3.9 Функции высшего порядка

Для обработки данных в ФП-языках используются функции высшего порядка. Как правило, в качестве параметров принимают коллекцию и лямбда-выражения для обработки коллекции.

Если функция принимают другие функции как параметры, то ее принято называть функцией высшего порядка. Теоретически такая функция может выполнять любые действия.

Но на практике большинство функций высшего порядка используются именно для обработки коллекций. Они выполняют задачи аналогичные LINQ to Objects в языке C#.

Мы будем изучать функции высшего порядка на примере списков.

Большинство функций высшего порядка каррированы, поэтому принимают параметры через пробел.

Далее рассмотрим основные функции высшего порядка.

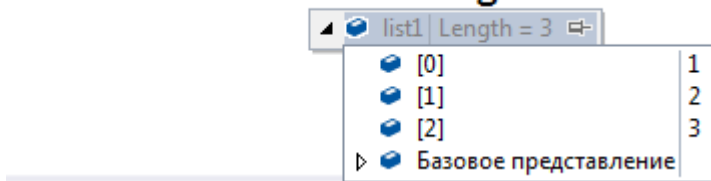
В качестве примера данных в большинстве примеров используется следующие списки:

```
let list1 = [1;2;3]
let list2:int list = [1;2;3]
let list3 = [1..10]
```

3.9.1 length

Возвращает длину коллекции.

```
let len1 = list1.Length
```



3.9.2 exists

Проверяет, что существует хотя бы один элемент списка, который удовлетворяет заданному предикату.

Предикатом в функциональном программировании принято называть функцию (как правило, лямбда-выражение) которая на выходе возвращает

логическое значение. (В логическом программировании термин предикат имеет другое значение).

Пример:

```
let res1 = List.exists(fun x-> x>2)(list1)
```

или

```
let res1 = List.exists(fun x-> x>2) list1
```

```
res1 | true ⇐
```

3.9.3 forall

Проверяет, что все элементы списка удовлетворяют заданному предикату.

```
let res2 = List.forall(fun x-> x<5) list1
```

```
res2 | true ⇐
```

3.9.4 find

Возвращает первый элемент списка, удовлетворяющий предикату.

В данном примере в x по очереди подставляются все элементы списка и возвращается первый четный элемент.

```
let res3 = List.find(fun x->x % 2 = 0) list1
```

```
res3 | 2 ⇐
```

3.9.5 filter

Возвращает все элементы списка, удовлетворяющие предикату.

В данном примере в x по очереди подставляются все элементы списка и возвращаются все четные элементы.

```
let res4 = List.filter(fun x->x % 2 = 0) list3
```

Index	Value
[0]	2
[1]	4
[2]	6
[3]	8
[4]	10

3.9.6 map

Применяет лямбда-функцию к каждому элементу списка, все результаты сохраняются в новый список.

Функция map – это функциональный аналог цикла.

```
let res5 = List.map(fun x->x*x) list1
```

Index	Value
[0]	1
[1]	4
[2]	9

Возможно формирование более сложной структуры данных, например списка кортежей.

```
let res51 = List.map(fun x->(x, x*x)) list1
```

Index	Value
[0]	{(1, 1)}
[1]	{(2, 4)}
[2]	{(3, 9)}

3.9.7 fold

Осуществляет свертку элементов коллекции в единое значение. В некоторых ФП-языках отдельно выделяются функции левой и правой свертки, которые начинают обход списка слева или справа соответственно.

```
let res6 = List.fold(fun acc x -> acc + x) 0 list1
```

Variable	Value
res6	6

В данном примере лямбда-функция принимает два параметра – аккумулятор асс и текущее значение х. Лямбда-функция выполняет необходимую операцию агрегирования (в данном примере суммирование) и возвращает результат агрегирования на текущей итерации. На следующей итерации данный результат попадает в параметр асс.

Функция fold возвращает результат выполнения лямбда-функции для последнего элемента списка. Параметр «0» - значение аккумулятора по умолчанию.

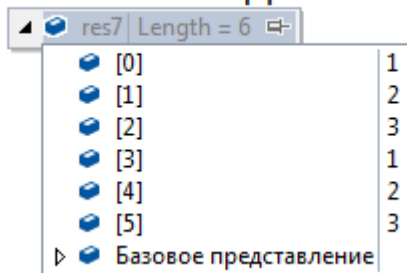
Пример работы функции показан в следующей таблице.

Итерация	Элемент списка	Значение аккумулятора	Результат вычисления лямбда-выражения
1	1	0 (получено как параметр функции)	1
2	2	1	3
3	3	3	6 (результатирующее значение функции)

3.9.8 append

Осуществляет конкатенацию коллекций.

```
let res7 = List.append list1 list2
```



3.9.9 zip

Соединяет элементы списков в список пар. Списки должны быть одинаковой длины.

```
let res5 = List.map(fun x->x*x) list1
let res8 = List.zip list1 res5
```

Index	Value
[0]	{(1, 1)}
[1]	{(2, 4)}
[2]	{(3, 9)}
Базовое представление	

3.9.10 rev

Меняет порядок следования элементов на обратный.

```
let res9 = List.rev list1
```

Index	Value
[0]	3
[1]	2
[2]	1
Базовое представление	

3.9.11 sort

Сортировка элементов списка по возрастанию.

```
let res10 = List.sort [1;3;5;4;2]
```

Index	Value
[0]	1
[1]	2
[2]	3
[3]	4
[4]	5
Базовое представление	

В сочетании с rev можно добиться сортировки по убыванию.

```
let res11 = List.rev (List.sort [1;3;5;4;2])
```

res11 Length = 5	
[0]	5
[1]	4
[2]	3
[3]	2
[4]	1
Базовое представление	

3.9.12 partition

Принимает два параметра – список и предикат. Делит исходный список на 2 части – элементы, для которых предикат вернул значение true, и элементы для которых предикат вернул значение false.

```
let listTrue, listFalse = List.partition (fun x -> x >= 3) [1;3;5;4;2]
```

listTrue Length = 3	
[0]	3
[1]	5
[2]	4
Базовое представление	

```
let listTrue, listFalse = List.partition (fun x -> x >= 3) [1;3;5;4;2]
```

listFalse Length = 2	
[0]	1
[1]	2
Базовое представление	

3.9.13 Функции агрегирования

Функции sum, min, max, average.

```
let res12 = List.max (List.sort [1;3;5;4;2])
```

res12	
5	

3.10 Алгоритм быстрой сортировки списков

Одним из традиционных «шедевров» ФП является алгоритм быстрой сортировки списков QuickSort – https://ru.wikipedia.org/wiki/Быстрая_сортировка

Идея алгоритма состоит в том, что выбирается некоторый средний элемент коллекции и предполагается, что он находится на своем месте. Далее перебираются все элементы коллекции слева и справа от элемента.

Если слева находится больший элемент, а справа меньший, то они меняются местами. В результате такого прохода выбранный элемент действительно находится на своем месте, слева от него находятся все меньшие, а справа все большие элементы, однако элементы справа и слева не упорядочены. Поэтому далее алгоритм вызывается отдельно для левой и правой части, где также выбираются средние элементы и работа алгоритма рекурсивно повторяется до тех пор, пока размер сортируемого подмассива не будет равен одному элементу.

Не смотря на «наивность» и относительную простоту реализации алгоритма, доказано, что он является одним из самых эффективных алгоритмов сортировки.

В некоторых функциональных языках программирования данный алгоритм реализуется почти в одну строку кода (не считая проверки на пустоту списка и т.д.).

Пример реализации на C#:

```

/// <summary>
/// Алгоритм быстрой сортировки
/// </summary>
private void Sort(int low, int high)
{
    int i = low;
    int j = high;
    T x = Get((low + high) / 2);
    do
    {
        while (Get(i).CompareTo(x) < 0) ++i;
        while (Get(j).CompareTo(x) > 0) --j;
        if (i <= j)
        {
            Swap(i, j);
            i++; j--;
        }
    } while (i <= j);

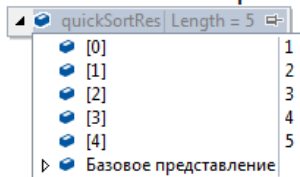
    if (low < j) Sort(low, j);
    if (i < high) Sort(i, high);
}

```

Пример реализации на F#:

```
let rec quickSort (lst: int list) =
    match lst with
    //Если список пустой или состоит из 1 элемента,
    //то возвращается список
    | [] | [_] -> lst
    //иначе список представляется в виде шаблона голова::хвост
    //x - первый элемент списка (один элемент)
    //xs - все оставшиеся элементы списка (список)
    | x::xs ->
        //с использованием функции partition получаем два списка
        //listLessX - элементы меньше x
        //listGreaterX - элементы большие или равные x
        let listLessX, listGreaterX = xs |> List.partition ((>=) x)
        //возвращаем список содержащий конкатенацию 3 списков
        //quickSort(listLessX) - рекурсивно сортируем элементы меньше x
        //[x] - список, содержащий сам элемент
        //quickSort(listLessX) - рекурсивно сортируем элементы большие x
        quickSort(listLessX) @ [x] @ quickSort(listGreaterX)

let quickSortRes = quickSort [1;3;5;4;2]
```



Отметим, что на других функциональных языках алгоритм реализуется похожим способом.

Пример реализации QuickSort на языке Haskell (++ - оператор конкатенации списков):

```
quickSort [] = []
quickSort (x:xs) = (quickSort lesser) ++ [x] ++ (quickSort greater)
    where
        lesser = filter (< p) xs
        greater = filter (>= p) xs
```

Пример реализации QuickSort на языке Erlang:

```
quickSort ([]) -> [];
quickSort ([x|xs]) ->
    quickSort([i || i <- xs, i < x])
    ++ [x] ++
    quickSort([i || i <- xs, i >= x]).
```

Необходимо отметить, что примеры на F# и Haskell используют функции высшего порядка, а пример на Erlang использует list comprehension.

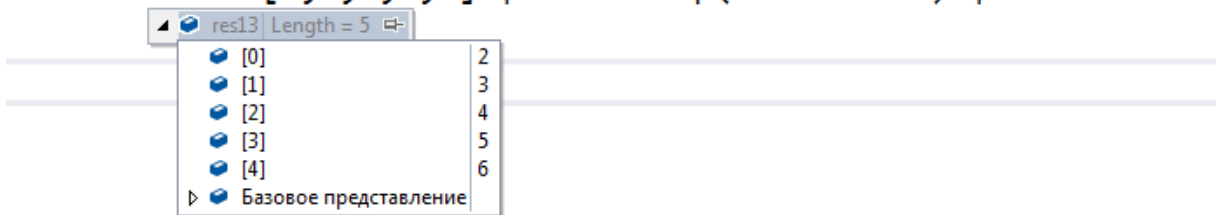
Как правило, эти два способа являются альтернативными. Большинство задач обработки списков можно решить как с помощью функций высшего порядка, так и с помощью list comprehension.

3.11 Операторы потоков и композиции функций

3.11.1 Операторы потоков

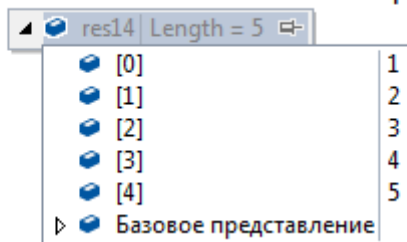
Оператор потока «`|>`» позволяет соединять результаты выполнения функций высшего порядка.

```
let res13 = [1;3;5;4;2] |> List.map(fun x->x+1) |> List.sort
```



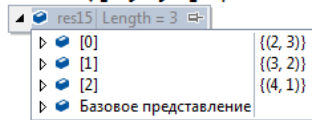
Оператор потока «`<|`» позволяет соединять результаты выполнения функций высшего порядка в обратном порядке. Это иногда помогает избежать простановки лишних скобок.

```
let res14 = List.sort <| [1;3;5;4;2]
```



Оператор «`||>`» позволяет преобразовать кортеж из двух элементов в каррированное представление и передать на вход функции, которая принимает два каррированных параметра.

```
let res15 = ([1;2;3] |> List.map(fun x->x+1), [1;2;3] |> List.rev) ||> List.zip
```



Разберем пример подробнее:

`[1;2;3] |> List.map(fun x->x+1)` – возвращает список `[2;3;4]`

`[1;2;3] |> List.rev` – возвращает список `[3;2;1]`

С помощью `(... , ...)` эти выражения соединены в кортеж.

Оператор « `||>` » преобразует кортеж в каррированное значение из двух списков, которое передается на вход функции `List.zip` (которая попарно соединяет элементы из двух списков).

3.11.2 Операторы композиции функций

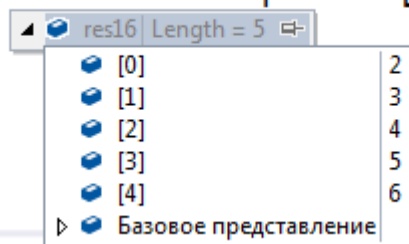
Операторы композиции функций очень похожи на операторы потоков, но они возвращают не результаты обработки данных, а функцию, которая осуществляет обработку.

```
let FuncMapSort = List.map(fun x->x+1) >> List.sort
```

```
val FuncMapSort : (int list -> int list)
```

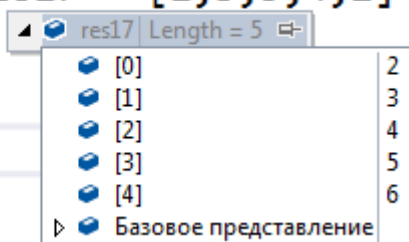
Результат вызова композиции функций:

```
let res16 = FuncMapSort [1;3;5;4;2]
```



ИЛИ

```
let res17 = [1;3;5;4;2] |> FuncMapSort
```



4 Основные операторы языка F#

Функциональные языки программирования отличаются относительно малым количеством операторов, потому что многие сложные элементы программ в них реализуются за счет использования лямбда-выражений и функциональных значений – function values. F# не является исключением и также содержит относительно малое количество операторов.

4.1 Операторы циклов

Отметим, что в чистых функциональных языках (Haskell, Erlang) циклы отсутствуют, вместо них используется рекурсия.

4.1.1 Цикл For

Цикл For может использоваться в режиме счетного цикла:

```
for i=1 to 5 do
    printfn "%A" i

for i=5 downto 1 do
    printfn "%A" i
```

Или в режиме перебора коллекции (аналог цикла foreach):

```
for i in [1..5] do
    printfn "%A" i

let list1 = [1..5]
for i in list1 do
    printfn "%A" i
```

4.1.2 Цикл While

Данная конструкция реализует цикл с предусловием. Необходимо отметить, что цикл с постусловием (например, do ... while() в C#) в F# отсутствует.

```
let mutable iw = 1
while iw <= 5 do
    printfn "%A" iw
    iw <- iw + 1
```


4.2 Условные операторы

4.2.1 Оператор if

Реализует проверку условия. Может быть единственной конструкцией в функции.

```
let x = 0
if x=0 then printfn "zero"
elif x>0 then printfn "pos"
else printfn "neg"
```

В F# может использоваться как аналог тернарного оператора.

Результат выполнения if может быть присвоен переменной:

```
let x_res = if x=0 then "zero" elif x>0 then "pos" else "neg"
printfn "%A" x_res
```

В F# используются логические операторы И (&&), ИЛИ (||), функция НЕ (not).

```
//проверка условий
if (x<0) && not(x<10) || (x>10) then printfn "result1"
else printfn "result2"
```

4.2.2 Сопоставление с образцом. Оператор match

Оператор match является аналогом оператора switch ... case в C++ и C#. Но если в C++ и C# предпочтение отдается оператору if, а switch ... case используется существенно реже, то в F# и других функциональных языках программирования основной конструкцией является именно сопоставление с образцом.

```
//Реализация с помощью if
let c1 =
    if x=0 then "zero"
    else "nonzero"

//Не совсем корректная реализация с помощью match
let c2 =
    match x=0 with
    | true -> "zero"
    | false -> "nonzero"

//Корректная реализация с помощью match
let c3 =
    match x with
    | 0 -> "zero"
```

```
| _ -> "nonzero"
```

Важным понятием является завершенность шаблонов (exhaustive match).

Пример незавершенного шаблона, который не покрывает все возможные случаи:

```
let c4 =
  match x with
  | 1 -> "1"
  | 2 -> "2"
```

В таком случае компилятор компилирует код, но выдает предупреждение: «Незавершенный шаблон соответствует данному выражению. К примеру, значение "0" может указывать на случай, не покрытый шаблоном(ами)».

Соответствующий завершенный шаблон:

```
let c4 =
  match x with
  | 1 -> "1"
  | 2 -> "2"
  | _ -> "unknown"
```

Оператор match может содержать дополнительные «охранные выражения» (guards), которые накладывают дополнительные условия на соответствующие ветви проверки:

```
let y1 = 1
let y2 = 2
let c5 =
  match y1 with
  | 1 when y2>0 -> "result1"
  | 1 when y2<=0 -> "result2"
  | _ -> "result3"
```

С помощью охранных выражений можно также проверять условия на неравенство. Обратите внимание, что в следующем примере охранный выражение использует основную переменную для проверки:

```
let y3 = 1
let c6 =
  match y3 with
  | _ when y3>0 -> "result1"
  | _ when y3<=0 -> "result2"
  | _ -> "result3"
```

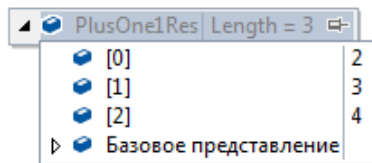
Для проверки типов в шаблоне используется символ «:?». Пример:

```
let checkType (x : obj) =
    match x with
    | :? string -> "строка"
    | :? int -> "целое число"
    | _ -> "другой тип"

printfn "%A" (checkType(1))
printfn "%A" (checkType("1"))
printfn "%A" (checkType(1.0))
```

Оператор `match` может быть реализован на уровне функции. Фактически, таким образом в функциональных языках реализуется перегрузка функций. Пример (рассмотренный ранее):

```
///Рекурсивная обработка списка (сопоставление с образцом)
let rec PlusOne1 = function
    | [] -> []
    | x::xs -> x+1::PlusOne1(xs)
//Вызов функции
let PlusOne1Res = PlusOne1([1..3])
```



Необходимо ответить, что `list comprehensions` (аналог цикла) и `match` (аналог условного оператора) являются основными конструкциями функциональных языков программирования.

4.3 Обработка исключений

Пример исключения при делении на 0.

```
try
    let a = 3 / 0
    printfn "%d" a
with
    | :? DivideByZeroException -> printfn "divided by zero"
```

В F# поддерживаются конструкции `try...with` или `try...finally`, но не `try...with...finally`. Следующий пример не компилируется:

```
try
    let a = 3 / 0
```

```

    printfn "%d" a
with
| :? DivideByZeroException -> printfn "divided by zero"
finally
    printfn "finally"

```

Но конструкции `try...with` и `try...finally` могут быть вложены друг в друга. Следующий пример компилируется:

```

try
  try
    let a = 3 / 0
    printfn "%d" a
  with
  | :? DivideByZeroException -> printfn "divided by zero"
finally
  printfn "finally"

```

Команда `failwith` генерирует исключение.

Можно получить доступ к переменной исключения с помощью следующей конструкции:

```

try
  failwithf "дата-время %A" DateTime.Now
with _ as e -> printfn "Информация об исключении: %s" e.Message

```

Возможно создание собственных исключений с помощью конструкции `exception` (объявляется не в функции, а на уровне модуля) и генерация исключений с помощью команды `raise`.

```

//на уровне модуля
exception Exception1 of int * int

```

```

//в функции
try
  raise (Exception1(1, 2))
with _ as e -> printfn "%A" e

```

5 Работа с типами в языке F#

В различных функциональных языках программирования работа с типами реализована различным образом.

В Erlang работа с типами реализована на базовом уровне, сложные типы фактически отсутствуют.

В Haskell реализована мощная система типов, но отсутствует понятие классов и наследования. Реализован механизм «классов типов» (typeclasses), который в значительной степени напоминает понятие интерфейсов C#.

Вероятно, наиболее мощной и гибкой системой типов обладают объектно-функциональные языки (Scala и F#), в которых реализованы как типы, используемые в функциональном программировании, так и обычные классы ООП. Необходимо отметить, что система типов в Scala более гибкая, чем в F#.

Рассмотрим основные механизмы работы с типами в F#.

5.1 Записи (record types)

Используются только для типов, содержащих данные.

```
type Person = {
    Name : string
    DateOfBirth : DateTime }

//объявление записи
let person1 = { Name = "Иван"; DateOfBirth = DateTime(1980, 09, 02) }
//объявление записи на основе существующей
let person2 = { person1 with Name = "Петр" }
```

5.2 Алгебраические типы (discriminated unions)

```
///Тип решения квадратного уравнения
type SquareRootResult =
    | NoRoots
    | OneRoot of double
    | TwoRoots of double * double //кортеж из двух double
```

Объявление нескольких типов одновременно (взаимно рекурсивные типы):

```
type Person = {
    Name : string
    DateOfBirth : DateTime }
and Persons =
    | Head of Person
    | Department of Person list
```

```
let head1 = Head({ Name = "Сергей"; DateOfBirth = DateTime(1980, 09, 02) })
```

5.3 Классы (classes)

Классы ООП, которые поддерживают наследование.

Особенности реализации классов:

- В класс можно передавать параметры, которые являются параметрами конструктора класса. Конструктор фактически «размазан» по всему классу, параметры можно присваивать членам класса.
- Можно объявлять дополнительные конструкторы класса, но они могут только вызывать первичный конструктор с дополнительными параметрами.
- Все члены класса необходимо объявлять с помощью ключевого слова «member».
- Класс может содержать привязки let, но они являются аналогами приватных полей и методов и не видны снаружи класса. Привязки let должны располагаться до member.
- Если у класса отсутствуют параметры конструктора, то при его объявлении необходимо использовать ключевое слово «class».
- Статические члены объявляются с помощью ключевого слова «static».

Пример объявления класса:

```

type Class1(ap: int, bp: string) =
  //свойство (property) поле данных
  let mutable PropertyValue = 0
  //приватные поле и метод
  let private_int = 123
  let private_func() = private_int * 2
  //поля инициализируются параметрами конструктора
  member this.a: int = ap
  member this.b: string = bp
  //методы
  member this.Print() = printfn "%i %s" this.a this.b
  member this.Multiply(mult : int) = printfn "%i" (this.a * mult)
  //статический метод
  static member Static1() = printfn "Static1"
  //метод использующий приватные поля
  member this.Print2() =
    let temp = private_func()
    printfn "%i" temp
  //свойство (property)
  member this.ReadWriteProperty
    with get () = PropertyValue
    and set (value) = PropertyValue <- value
  //альтернативные конструкторы
  new() = Class1(2, "str2")
  new(i:int) = Class1(i, "str3")
  //виртуальный метод
  abstract VirtPrint: unit -> unit
  default this.VirtPrint() = printfn "%i %s" this.a this.b

```

Создание объекта и работа с классом:

```

let c11 = Class1(1, "str")
c11.Print()
c11.VirtPrint()
c11.Print2()
c11.Multiply(333)
Class1.Static1()
c11.ReadWriteProperty <- 333
let ReadWritePropertyVal = c11.ReadWriteProperty
let c111 = Class1()
let c112 = Class1(1)

```

При объявлении класса можно явно указывать ключевое слово «class»:

```

type Class11() = class
  member this.a: int = 0
  member this.b: string = ""
end

```

Для наследования классов используется ключевое слово «inherit». Как и в C# наследоваться можно только от одного класса.

Пример наследуемого класса и создание объекта класса:

```
type Class2(ap: int, bp: string, cp: float) =
  //наследование от класса 1
  inherit Class1(ap, bp)
  member this.c: float = cp
  //перегрузка виртуального метода
  override this.VirtPrint() = printfn "%i %s %f" this.a this.b this.c
  //перегрузка метода ToString для класса Object
  override this.ToString() = "!!!"

let c12 = Class2(1, "str1", 3.14)
c12.VirtPrint()
```

Для объявления виртуального метода используется двойное объявление, сначала он объявляется как абстрактный (указывается только сигнатура метода), а затем дается его реализация по умолчанию:

```
//виртуальный метод
abstract VirtPrint: unit -> unit
default this.VirtPrint() = printfn "%i %s" this.a this.b
```

В данном примере сигнатура «unit -> unit» означает, что метод не принимает параметров и не имеет возвращаемого значения. Обратите внимание, что сигнатура задается в виде функционального выражения.

В наследуемом классе перегрузка реализуется с помощью ключевого слова «override».

```
//перегрузка виртуального метода
override this.VirtPrint() = printfn "%i %s %f" this.a this.b this.c
```

5.4 Абстрактные классы

Абстрактные классы содержат абстрактные методы «abstract» без реализации по умолчанию «default». При этом абстрактный класс должен быть помечен аннотацией «[<AbstractClass>]».

```
[<AbstractClass>]
type Abstract1() =
  //абстрактный метод
  abstract member Add: int -> int -> int
  //абстрактное неизменяемое свойство
  abstract member Pi : float
  //абстрактное изменяемое свойство
  abstract member Prop1 : float with get,set
```


Наследуемый конкретный класс должен реализовать все абстрактные методы:

```
type Concrete1() =
  inherit Abstract1()
  let mutable Prop1Value = 0.0
  override this.Add a b = a+b
  override this.Pi = 3.14
  override this.Prop1
    with get () = Prop1Value
    and set (value) = Prop1Value <- value
```

Абстрактный класс может содержать часть реализованных методов, например:

```
[<AbstractClass>]
type Abstract1() =
  //абстрактный метод
  abstract member Add: int -> int -> int
  //абстрактное неизменяемое свойство с реализацией по умолчанию
  abstract member Pi : float
  default this.Pi = 3.14
  //абстрактное изменяемое свойство
  abstract member Prop1 : float with get,set
```

5.5 Интерфейсы

F# поддерживает использование интерфейсов с помощью ключевого слова «interface».

```
type Interface1 = interface
  //абстрактный метод
  abstract member Add: int -> int -> int
  //абстрактное неизменяемое свойство
  abstract member Pi : float
end
```

Обратите внимание, что при объявлении интерфейса не указываются скобки, так как у интерфейса не может быть конструктора по умолчанию.

Наследование интерфейсов также поддерживается:

```
type Interface2 = interface
  inherit Interface1
  //абстрактное изменяемое свойство
  abstract member Prop1 : float with get,set
end
```

Класс, реализующий интерфейс:

```
type Interface2Implement1() =
    //изменяемая переменная для свойства Prop1
    let mutable Prop1Value = 0.0
    //реализация интерфейса
    interface Interface2 with
        member this.Add a b = a+b
        member this.Pi = 3.14
        member this.Prop1
            with get () = Prop1Value
            and set (value) = Prop1Value <- value
```

При вызове методов необходимо явно преобразовать переменную класса к интерфейсному типу:

```
let ii2 = Interface2Implement1()
let ii2Int = ii2 :> Interface2
let sum = ii2Int.Add 1 2
```

Интересной особенностью F# является возможность реализации интерфейсов прямо в теле функции с помощью «объектных выражений» (Object Expressions):

```
//Реализация интерфейса с использованием Object Expressions
let Interface1Implement =
    { new Interface1 with
        member this.Add a b = a+b
        member this.Pi = 3.14 }

let sum1 = Interface1Implement.Add 1 2
```

Таким образом, для реализации интерфейса можно не создавать класс.

5.6 Активные шаблоны

Активные шаблоны являются оригинальной особенностью F#. Этот механизм позволяет динамически «выделить» варианты алгебраического типа в типе данных, рассмотреть тип данных «под различными углами».

Пример объявления комплексного числа, с которым можно работать как в декартовых, так и в полярных координатах:

```
type Complex(r : float, i : float) =
    static member Polar(mag, phase) = Complex(mag * cos phase, mag * sin phase)
    member x.Magnitude = sqrt(r * r + i * i)
    member x.Phase = atan2 i r
    member x.RealPart = r
    member x.ImaginaryPart = i
```

Определение активных шаблонов как «конструкторов», которые выделяют отдельно представления в декартовых и полярных координатах:

```
let (|Rect|) (x : Complex) = (x.RealPart, x.ImaginaryPart)
let (|Polar|) (x : Complex) = (x.Magnitude, x.Phase)
```

Теперь активные шаблоны можно использовать как дискриминаторы в функциях. Пример функций, которые умножают числа в декартовом и полярном представлениях:

```
let mulViaRect a b =
  match a, b with
  | Rect (ar, ai), Rect (br, bi) -> Complex (ar * br - ai * bi, ai * br + bi * ar)

let mulViaPolar a b =
  match a, b with
  | Polar (m, p), Polar (n, q) -> Complex.Polar (m * n, p + q)
```

6 Функторы, аппликативные функторы и монады

В данном разделе мы разберем более сложные концепции, которые используются в функциональном программировании.

Функторы, аппликативные функторы и монады предназначены для работы со значениями, расположенными в контекстах.

Контексты задаются с помощью типов или функциональных значений (function values).

Для простоты понимания будем использовать иллюстрации из замечательной статьи — http://adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures.html (существует русский перевод — <https://habrahabr.ru/post/183150/>).

Также в этом разделе используется статья с пояснениями функциональных возможностей F# — <https://fsharpforfunandprofit.com/posts/elevated-world>

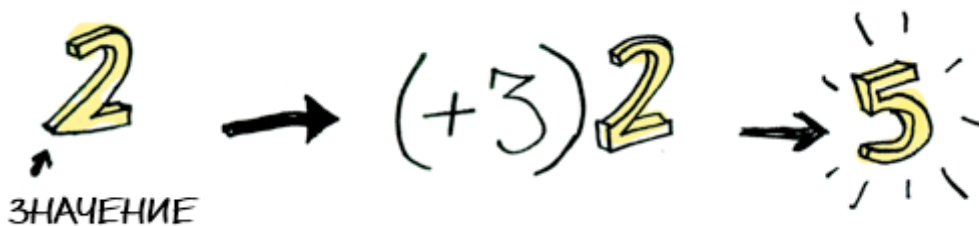
В качестве примера создадим в F# класс, который используется в Haskell как аналог типа Null (на этот класс ориентирована большая часть иллюстраций).

```
type Maybe<'a> =
  | Just of 'a
  | Nothing
```

Допустим есть простое число:



К нему можно применить функцию:

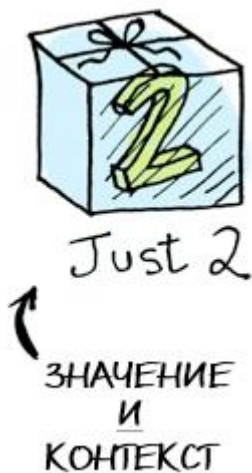


В F# используется немного другой синтаксис:

```
let plus a b = a + b
let temp1 = (plus 3) (2)
```

С вычислениями для чисел сложностей не возникает.

Предположим, что значение расположено в контексте (в данном примере находится внутри типа).

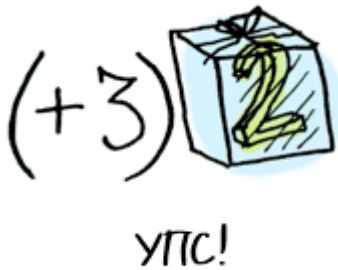


Из-за того, что значение «завернуто» в контекст, к нему невозможно напрямую прибавить число.

Но при этом понятно, что внутри Just2 хранится число 2 и должен быть какой-то способ получить в результате сложения, например, Just 5.

```
let Just2 = Just 2
let temp2 = plus Just2 3
```

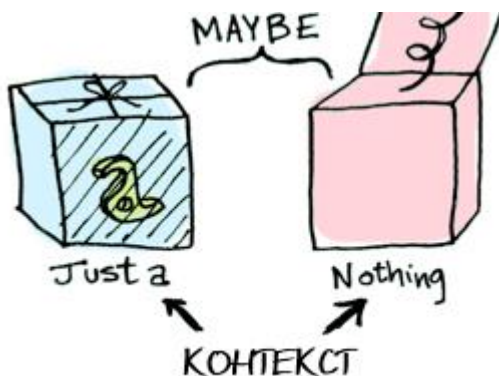
В данном выражении требовалось наличие типа int, но получен тип Maybe<int>



При этом в случае Nothing вычисления должны быть организованы другим способом.

То есть когда функция применяется к контексту, результаты получаются различными в зависимости от контекста. Это основная идея, на которой базируются функторы, аппликативные функторы, монады, и более сложные конструкции.

Тип данных Maybe определяет два связанных контекста:

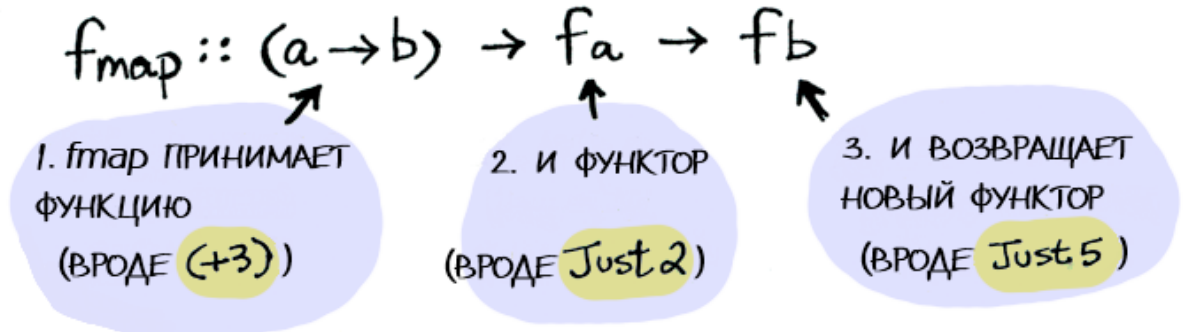


6.1 Функторы

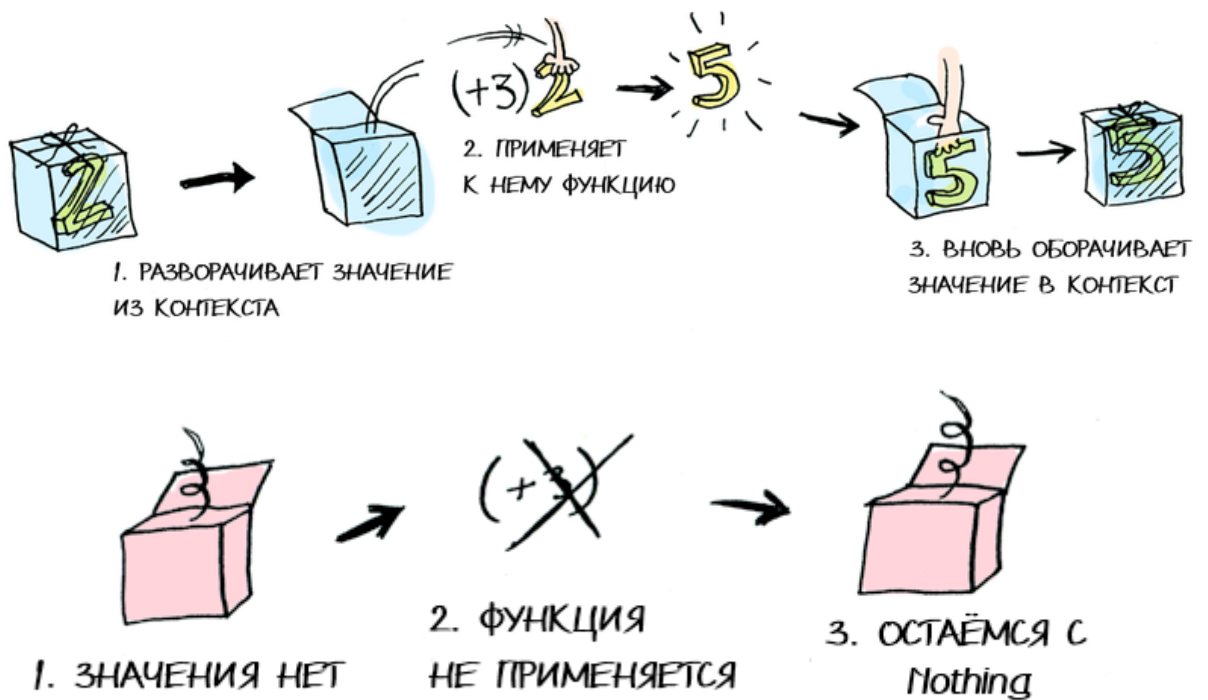
Функторы позволяют примерять функции к элементам внутри контекста. В Haskell существует функция fmap которая позволяет делать

это в общем виде, в F# в зависимости от контекста необходимо разрабатывать различные варианты функции.

Описание работы функции fmap в языке Haskell:



Примеры применения функции для класса Maybe:



Пример реализации на F#:

```
//реализация fmap для Maybe
let fmapMaybe f p =
    match p with
    | Just a -> Just (f a)
    | Nothing -> Nothing
```

Графическая интерпретация для списков:



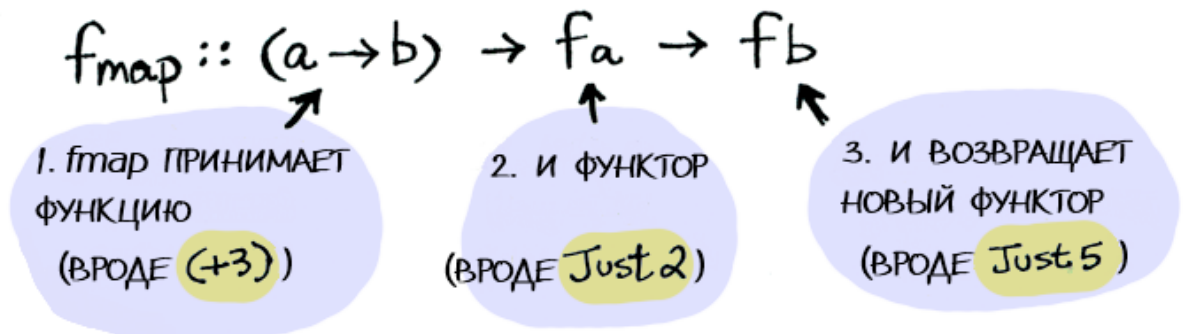
//реализация fMap для списка

```
let rec fmapList f list =
  match list with
  | [] -> []
  | x::xs -> f x :: fmapList f xs
```

```
let list1 = [1;2;3]
let lst1 = fmapList (fun x->x+1) list1
let lst2 = List.map (fun x->x+1) list1
```

Отметим, что для списков функция List.map работает аналогично fmap.

Убедимся, что сигнатуры функций соответствуют ожидаемой:



//реализация fMap для Maybe

```
let fmapMaybe f p =
```

```
val fmapMaybe: (('a -> 'b) -> Maybe<'a> -> Maybe<'b>)
```

```
match p with
| Just a -> Just (f a)
| Nothing -> Nothing
```

```
//реализация fmap для списка
```

```
let rec fmapList f list =
```

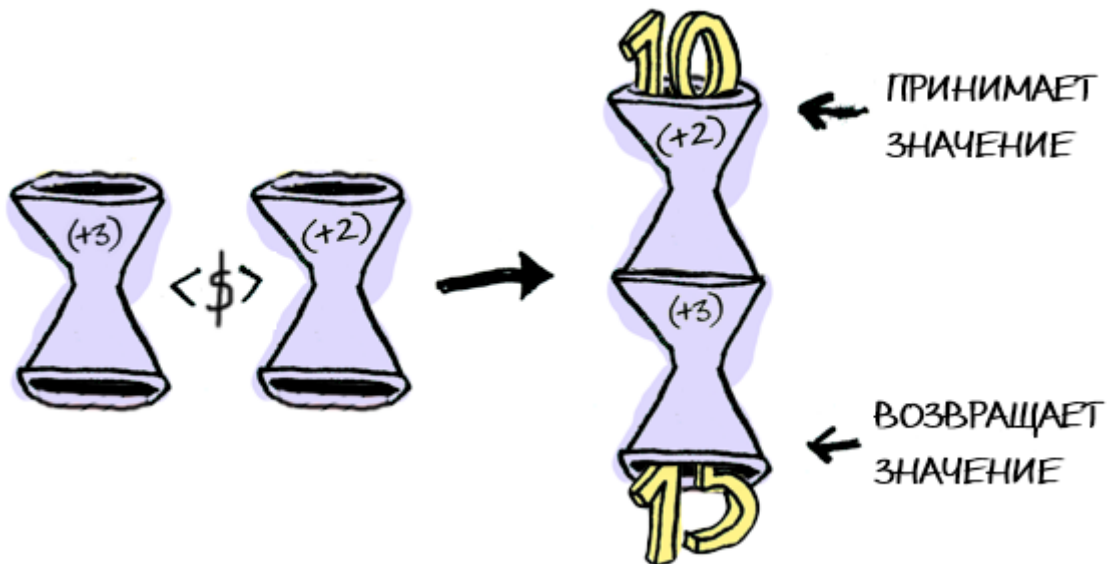
```
  val fmapList : ('a -> 'b) -> 'a list -> 'b list
```

```
  match list with
```

```
  | [] -> []
```

```
  | x::xs -> f x :: fmapList f xs
```

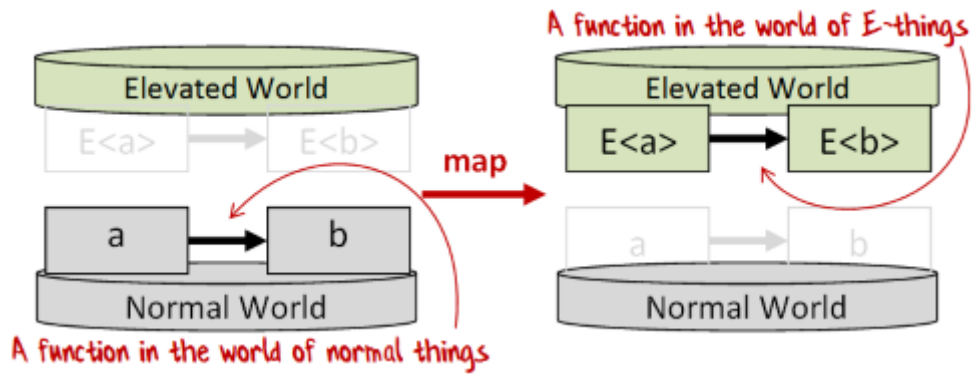
Отметим (без примера реализации), что функцию аналогичную `fmap` можно применить к функции (функция является функтором). В этом случае получается композиция функций:



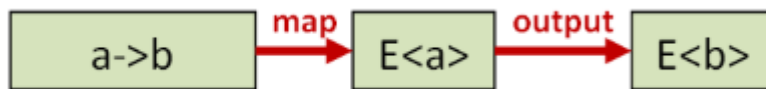
Пример на языке Haskell:

```
> import Control.Applicative
> let foo = fmap (+3) (+2)
> foo 10
15
```

Другая графическая интерпретация:



Или



Внесение вычисления (функции) в контекст называется **«поднятием (или подъемом) в контекст»**. Когда вычисление вносят внутрь контекста, то говорят, что его **«поднимают»**.

В английской версии используется термин «lift» или «lifting». Поэтому на рисунке вычисление внутри контекста называется поднятым «elevated».

Если для типа данных реализована функция, аналогичная `fmap`, то тип данных называется **функтором**.

Но при этом должны выполняться два закона функторов:

Закон 1.

Пусть `id` – функция, которая возвращает неизменным значение аргумента.

```
let id x = x
```

Тогда подъем этой функции в контекст не влияет на вычисление:

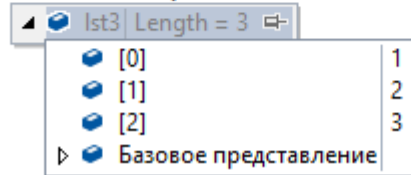


Проверим этот закон для `fmapList`:

```
let list1 = [1;2;3]
let lst1 = fmapList (fun x->x+1) list1
let lst2 = List.map (fun x->x+1) list1
```

```
let id x = x
```

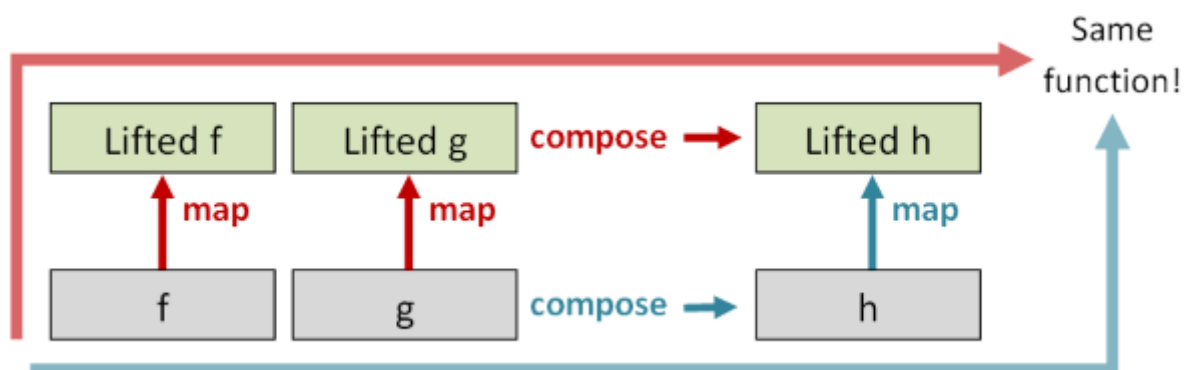
```
//Проверка для списка
let lst3 = fmapList id list1
```



Таким образом fmapList с функцией id вернул список, тождественный исходному.

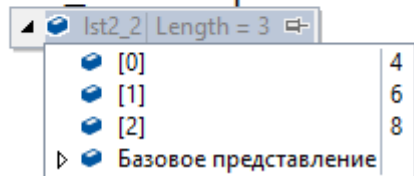
Закон 2.

Для двух функций f и g композиция их подъемов эквивалентна подъему композиции.



Проверим этот закон для fmapList:

```
//Проверка 2 закона функторов для списка
let func_f x = x + 1
let func_g x = x * 2
let lst2_1 = fmapList func_f list1
let lst2_2 = fmapList func_g lst2_1
```

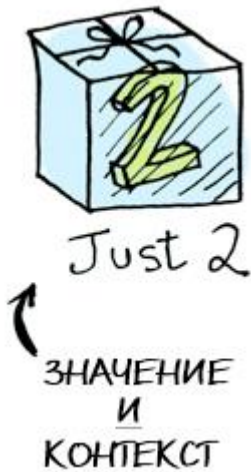


```
let lst2_3 = fmapList (func_f >> func_g) list1
```

Index	Value
[0]	4
[1]	6
[2]	8

6.2 Аппликативные функторы

В этом случае значение упаковано в контекст:



Но в контекст также упакована и сама функция:



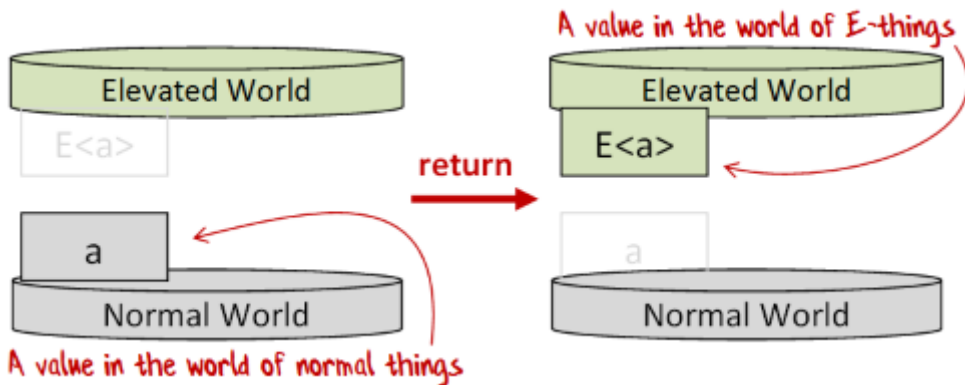
В языке Haskell аппликативный функтор обозначается оператором «`<*>`».

Принцип работы аппликативного функтора:



Аппликативный функтор содержит две функции:

1. $\text{return} :: a \rightarrow E(a)$



Функция `return` поднимает значение в контекст.

2. $\text{apply} :: E(a \rightarrow b) \rightarrow E(a) \rightarrow E(b)$



Функция `apply` применяет поднятую функцию к поднятым аргументам.

```
//реализация apply для Maybe
let applyMaybe lf p =
  match lf, p with
  | Just f, Just a -> Just (f a)
  | _ -> Nothing

let am1 = applyMaybe (Just (fun x->x+1)) (Just 2)
am1 | Just 3
```

Убедимся, что сигнатура функции соответствует аппликативному функтору:

```
//реализация apply для Maybe
let applyMaybe lf p =
  val applyMaybe : (Maybe<'a -> 'b> -> Maybe<'a> -> Maybe<'b>)

  match lf, p with
  | Just f, Just a -> Just (f a)
  | _ -> Nothing
```

```
//реализация apply для списка
let rec applyList (lf: ('a->'b) list) (list: 'a list) =
  [ for f in lf do
    for x in list do
      yield f x]
```

```
let al1 = applyList [(fun x->x+1);(fun x->x*10)] [1;2;3]
```

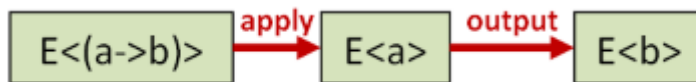
Index	Value
[0]	2
[1]	3
[2]	4
[3]	10
[4]	20
[5]	30

Убедимся, что сигнатура функции соответствует аппликативному функтору:

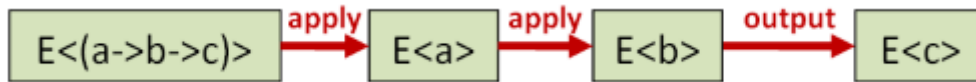
```
//реализация apply для списка
let rec applyList (lf: ('a->'b) list) (list: 'a list) =
    val applyList : (('a -> 'b) list -> 'a list -> 'b list)

    [ for f in lf do
      for x in list do
        yield f x]
```

Поднятая функция может быть как от одного аргумента:

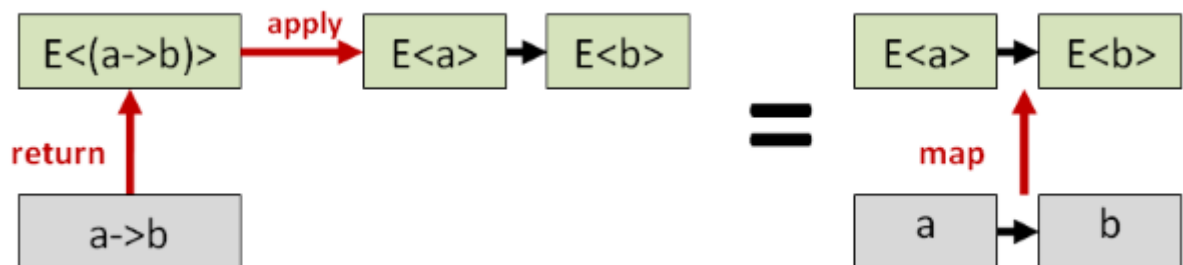


Так и от большего числа аргументов (пример с двумя аргументами):



Сравнение функтора и аппликативного функтора.

Аппликативный функтор считается более гибким средством чем обычный функтор, потому что с помощью комбинаций функций `apply` и `return` можно получить `map` (`fmap`). Но обратное невозможно.

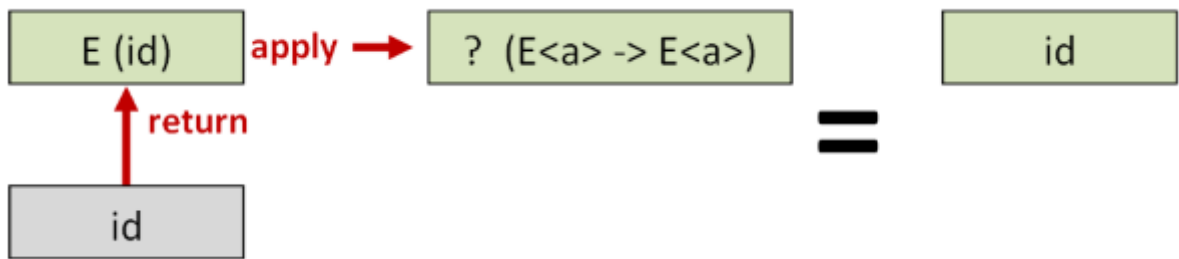


Если для типа данных реализованы функции `apply` и `return`, в соответствии с приведенными выше сигнатурами, то тип данных называется аппликативным функтором.

Но при этом должны выполняться 4 закона аппликативных функторов:

Описание на языке Haskell:

1. `pure id <*> v = v` -- Identity
2. `pure f <*> pure x = pure (f x)` -- Homomorphism
3. `u <*> pure y = pure ($ y) <*> u` -- Interchange
4. `pure (.) <*> u <*> v <*> w = u <*> (v <*> w)` -- Composition

Закон 1.

Применение поднятой функции `id` к поднятому значению эквивалентно применению неподнятой функции `id` к неподнятому значению.

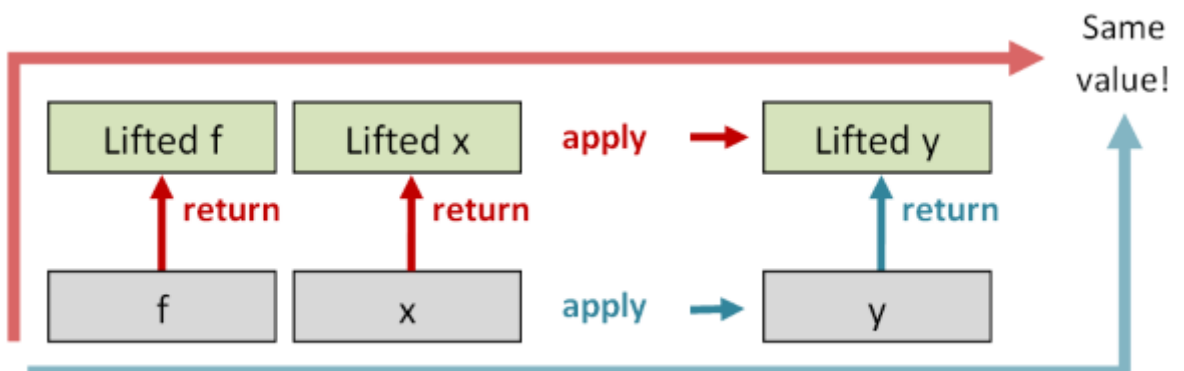
Проверим для `applyList`:

```
let at11 = id 1
```

```
let at11 = id 1
```

```
let at12 = applyList [id] [1;2;3]
```

Функция `id` переданная в списке `[id]` применяется также как и функция `id` вне списка.

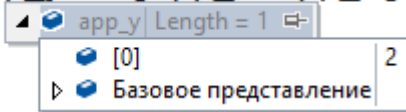
Закон 2.

Если $y=f(x)$, то подъем функции `f` и значения `x` и применение к ним функции `apply` приведет к такому же результату, что и подъем `y`.

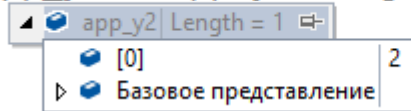
Проверим для `applyList`:

В качестве функции return используется конструктор списка – [].

```
let app_f = fun x -> x+1
let app_x = 1
let app_y = [app_f app_x]
```



```
let app_y2 = applyList [app_f] [app_x]
```



Законы 3 и 4 более сложны, но мы разберем их в базовом варианте.

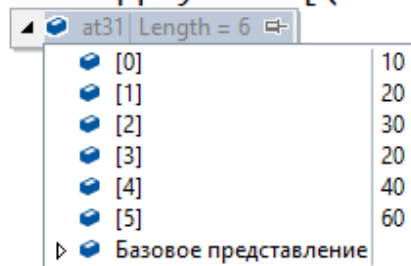
Закон 3.

Аргументы apply можно менять местами.

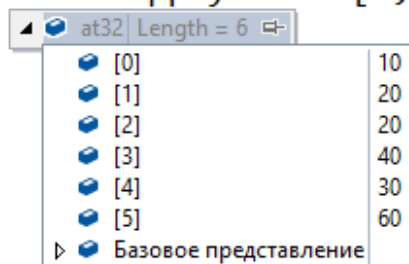
Проверим для applyList:

```
//Проверка 3 закона
```

```
let at31 = applyList [(fun x->x*10);(fun x->x*20)] [1;2;3]
```



```
let at32 = applyList2 [1;2;3] [(fun x->x*10);(fun x->x*20)]
```



Поскольку в F# нет встроенной функции <*> (apply), то необходимо дописать функцию:

```
let rec applyList2 (list: 'a list) (lf: ('a->'b) list) =
    [ for x in list do
      for f in lf do
        yield f x]
```


Закон 4.

Композиция функций `apply` ассоциативна. Из-за отсутствия встроенной функции `<*>` (`apply`) продемонстрировать проверку данного правила невозможно.

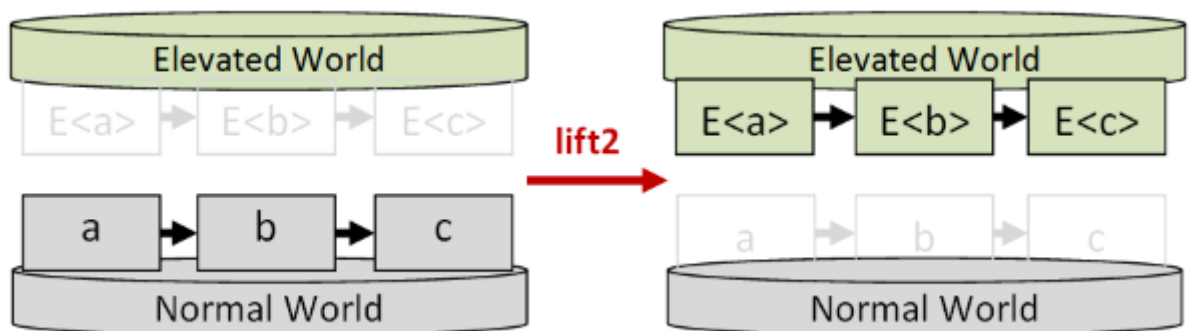
6.3 Функции лифтинга

Существуют функции серии `liftN`, которые на основе неподнятой функции формируют поднятую функцию. Для реализации используются функции `apply` и `return`.

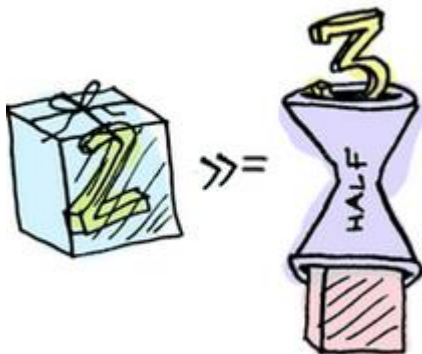
```
lift2: (a->b->c) -> E<a> -> E<b> -> E<c>
```

```
lift3: (a->b->c->d) -> E<a> -> E<b> -> E<c> -> E<d>
```

Пример реализации функции `lift2`.

**6.4 Монады**

Монада применяет к поднятому значению функцию от обычного аргумента, которая возвращает поднятое значение.



В Haskell основным оператором для монады является оператор «>>=» (оператор bind).



Монада определяется тремя параметрами:

1. монадический тип m

2. $\text{return} :: a \rightarrow m(a)$

3. $\text{bind} ::$

$(a \rightarrow m\langle b \rangle) \rightarrow m\langle a \rangle \rightarrow m\langle b \rangle$

Или:

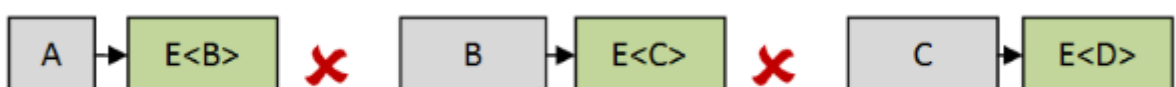
$m\langle a \rangle \rightarrow (a \rightarrow m\langle b \rangle) \rightarrow m\langle b \rangle$

Иногда для обозначения монадического типа применяется обозначение E как и в случае функторов и аппликативных функторов.

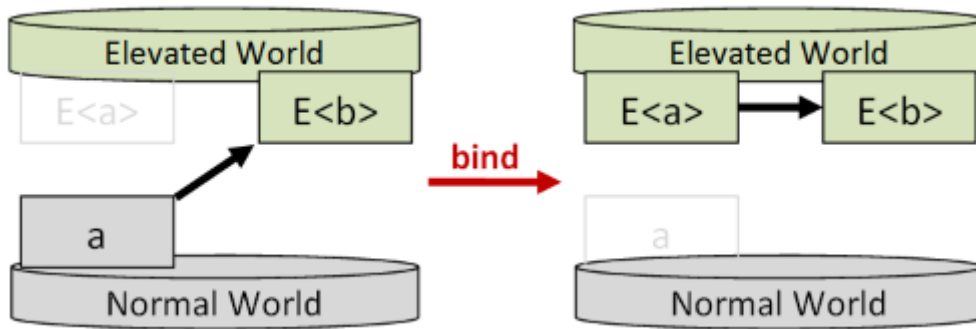
Существуют различные интерпретации монад.

Интерпретация 1.

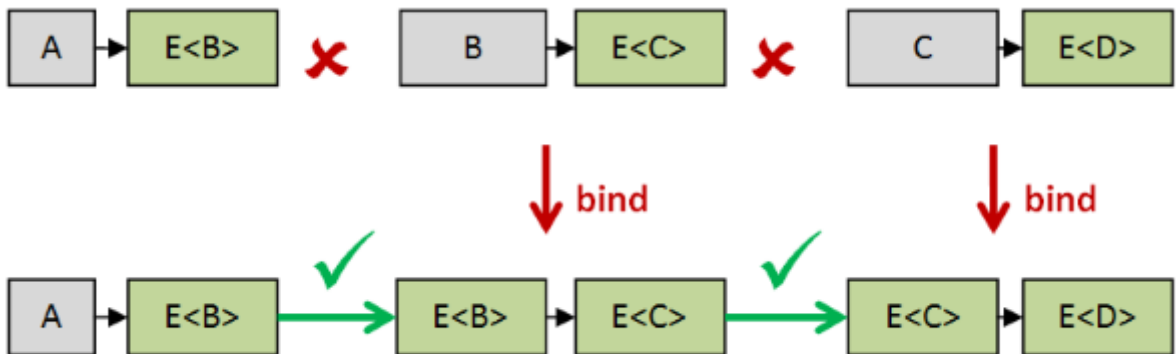
Существуют функции с неподнятым аргументом и поднятым значением. Их невозможно соединить в композицию.



Функция `bind` применяет функцию и возвращает поднятый результат:



В результате этого поднятые функции можно соединить в композицию:



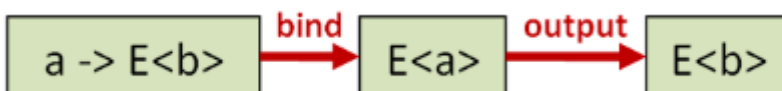
Основное преимущество монад – возможность организации цепочек вычислений внутри контекстов.

Интерпретация 2. (чаще встречается в литературе, но вероятно менее понятна).

Функция `bind` – это функция от 2 аргументов:

1. Функция которая неподнятое значение `a` преобразует в поднятое значение `b`.
2. Поднятое значение `a`

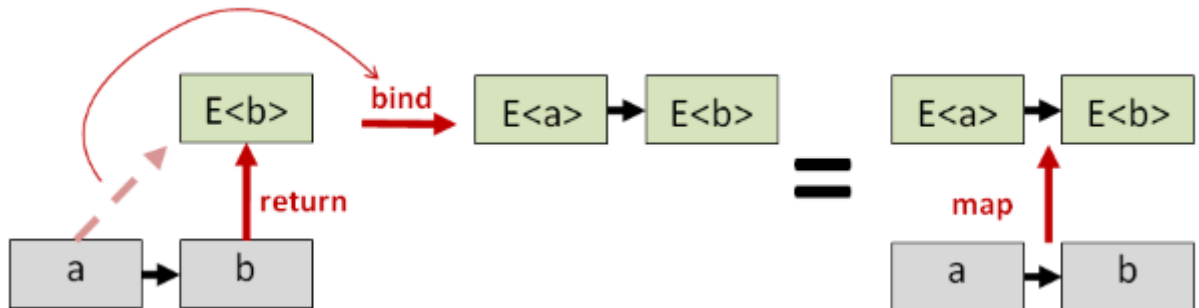
На выходе возвращается поднятое значение `b`.



Сравнение монады, функтора и аппликативного функтора.

Монада считается более гибким средством, чем аппликативный функтор и функтор, потому что с помощью функций `bind` и `return` можно построить функции `map` (функтор) и `aply` (аппликативный функтор).

Пример эмуляции `map`:



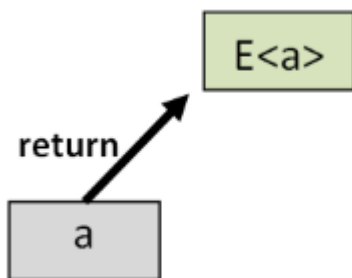
Если для типа данных реализованы функции `bind` и `return`, в соответствии с приведенными выше сигнатурами, то тип данных называется монадой.

Но при этом должны выполняться 3 закона монад:

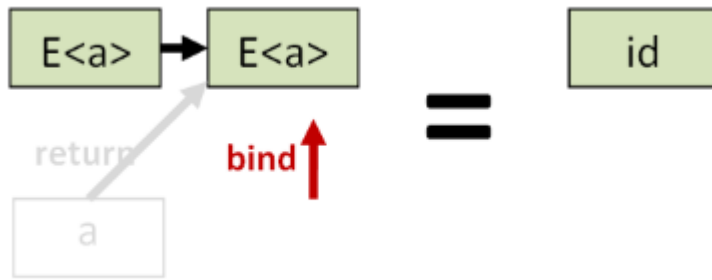
Описание на языке Haskell:

```
m >>= return      = m                -- right unit
return x >>= f    = f x              -- left unit
(m >>= f) >>= g   = m >>= (\x -> f x >>= g) -- associativity
```

Закон 1.



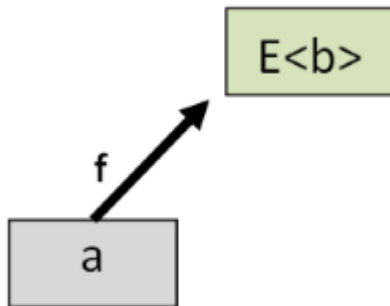
Функция `return` может быть поднята с помощью функции `bind`. Если это сделать, то она должна быть эквивалентна функции `id`.



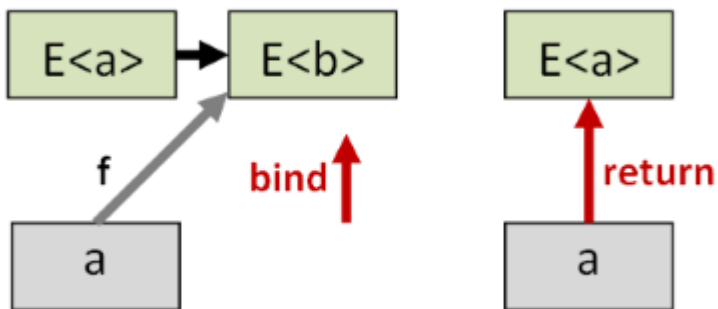
Закон 2.

Похож на закон 1, но в нем изменен на обратный порядок вызовов функций `bind` и `return`.

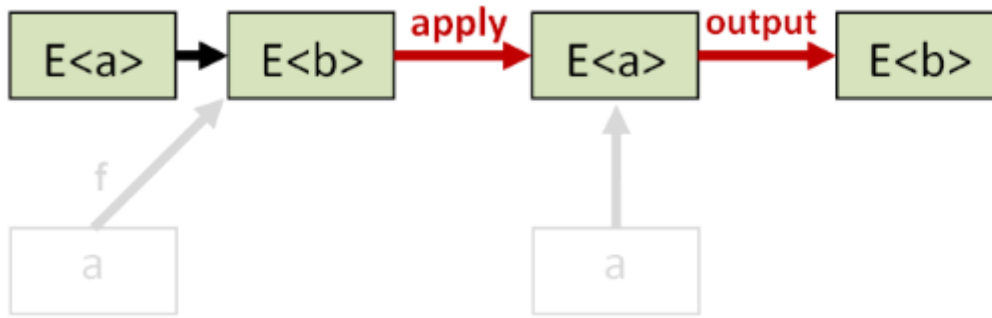
Пусть есть функция `f`:



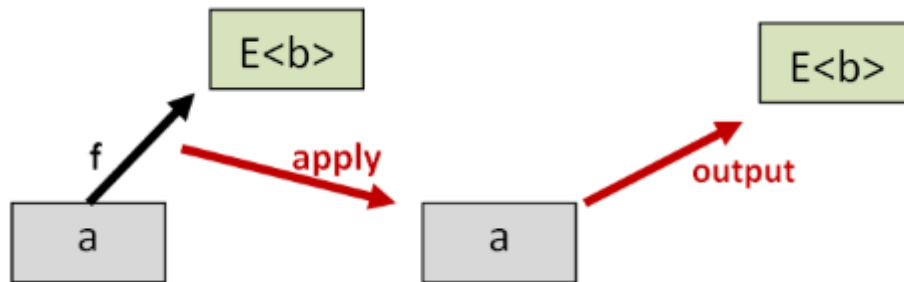
Поднимем `f` с помощью `bind`, и `a` с помощью `return`:



Если применить поднятую версию `f` к поднятой версии `a`, то получится поднятое значение `b`:



Но в соответствии с определением функции f , если применить f к a , то должно получиться то же самое значение:



То есть функции `bind` и `return` не должны нарушать целостность данных.

Закон 3.

Закон ассоциативности.

В неподнятом состоянии композиция функций ассоциативна:

```
let groupFromTheLeft = (a |> f) |> g
let groupFromTheRight = a |> (f >> g)
```

Закон 3 говорит о том, что после поднятия ассоциативность сохраняется:

```
let groupFromTheLeft = (a >>= f) >>= g
let groupFromTheRight = a >>= (fun x -> f x >>= g)
```

6.5 Выводы по разделу



Функтор. Применяется обычная функция к поднятому значению.

Апplikативный функтор. Применяется поднятая функция к поднятому значению.

Монада. К поднятому значению применяется функция от обычного аргумента, возвращающая поднятое значение.

Для того, чтобы проверить является ли тип функтором, аппликативным функтором или монадой, необходимо реализовать соответствующие функции для данных типов и доказать справедливость соответствующих законов.

Некоторые типы являются функтором, аппликативным функтором и монадой одновременно.

7 Библиотека FParsec для синтаксического анализа текста

Библиотека FParsec представляет собой библиотеку, построенную по принципу `Parser` `Combinator` (https://en.wikipedia.org/wiki/Parser_combinator), то есть функцию высшего порядка, которая принимает на вход простые парсеры (разборщики элементов текста) и возвращает сложный парсер, который распознает комбинацию простых элементов.

Библиотеки типа Parser Combinator достаточно просто реализуются на функциональных языках. Их преимуществом является относительная простота для использования прикладным программистом.

Аналогичным подходом является генерация парсеров на основе грамматик, в частности это известные библиотеки Lex и Yacc – <https://ru.wikipedia.org/wiki/Lex> и <https://ru.wikipedia.org/wiki/Yacc>. Существует реализация этих библиотек для F# – <http://fsprojects.github.io/FsLexYacc/> Использование этих библиотек как правило сложнее чем применение комбинаторов парсеров.

Документация по библиотеке FParsec – <http://www.quanttec.com/fparsec/>

7.1 Установка FParsec

Для работы с FParsec необходимо создать проект F# и установить Nuget-пакет FParsec – <https://www.nuget.org/packages/FParsec>

7.2 Основы работы с FParsec

FParsec поставляется с большим количеством готовых парсеров для разбора элементов базовых типов.

Для работы с вещественными числами используется парсер pfloat.

Для запуска парсера используется метод run.

Пример работы парсера:

```
let test p str =
    match run p str with
    | Success(result, _, _) -> printfn "Success: %A" result
    | Failure(msg, _, _) -> printfn "Failure: %s" msg

//разбор вещественного числа
test pfloat "333.333"
printfn "-----"
printfn "%A" (run pfloat "333.333")
test pfloat "qwerty333.333"
```



```
Success: 333.333
-----
Success: 333.333
Failure: Error in Ln: 1 Col: 1
qwerty333.333
^
Expecting: floating-point number
```

От простых парсеров (которые разбирают один элемент, например, вещественное число) перейдем к комбинаторам парсеров, которые позволяют объединять парсеры.

Комбинатор `A >>. B` разбирает последовательность элементов `A` и `B` после чего возвращает `B`. Нам гарантируется что `A` и `B` идут последовательно, но при этом результат `A` не важен.

Комбинатор `A .>> B` разбирает последовательность элементов `A` и `B` после чего возвращает `A`. Нам гарантируется что `A` и `B` идут последовательно, но при этом результат `B` не важен.

Пример:

```
//Разбор числа в скобках
let parseFloatBrackets =
  pstring "(" >>. pfloat .>> pstring ")"
printfn "%A" (run parseFloatBrackets "(123)")
```

```
Success: 123.0
```

Таким образом, конструкция `pstring "(" >>. pfloat .>> pstring ")"` гарантирует нам что число действительно находится в скобках, но при этом скобки не возвращаются, в результате разбора возвращается само число.

Пример с использованием потоков:

```
//С использованием потоков
let betweenStrings s1 s2 p = pstring s1 >>. p .>> pstring s2
let parseFloatBrackets2 = pfloat |> betweenStrings "(" ")"
printfn "%A" (run parseFloatBrackets2 "(123)")
```

```
Success: 123.0
```

Необходимо отметить, что существует встроенный комбинатор `between`.

Для разбора списка значений используется комбинатор `many`.

```
printfn "%A" (run (many parseFloatBrackets) "(123)(345)(333)")
```

```
Success: [123.0; 345.0; 333.0]
```

Комбинатор выбора `<|>` позволяет создавать альтернативы при разборе.

```
//Произвольная комбинация букв а и b
```

```
let a_or_b = many (pstring "a" <|> pstring "b")
```

```
printfn "%A" (run a_or_b "abbbabba")
```

```
printfn "%A" (run a_or_b "bbbbbaaaaaabbbbabba")
```

```
Success: [123.0; 345.0; 333.0]
```

```
Success: ["a"; "b"; "b"; "b"; "a"; "b"; "b"; "a"]
```

```
Success: ["b"; "b"; "b"; "b"; "b"; "b"; "a"; "a"; "a"; "a"; "a"; "a"; "b"; "b"; "b"; "b"; "a"; "b"; "b"; "a"]
```

Для игнорирования пробелов, переводов строк используется комбинатор `spaces`.

```
//Игнорирование пробелов
```

```
let pstring_ws s = spaces >>. pstring s .>> spaces
```

```
let float_ws = spaces >>. pfloat .>> spaces
```

```
let parseFloatBracketswithSpaces =
```

```
  pstring_ws "(" >>. float_ws .>> pstring_ws ")")
```

```
printfn "%A" (run parseFloatBracketswithSpaces " ( 123 ) ")
```

```
Success: 123.0
```

Если нужно получить несколько значений при последовательном разборе, то можно использовать комбинаторы от `pipe2` (2 значения) до `pipe5` (5 значений).

```
//Пример вычисления суммы
```

```
let a_plus_b = pipe5
```

```
  (pstring_ws "(")
```

```
  (float_ws)
```

```
  (pstring_ws "+")
```

```
  (float_ws)
```

```
  (pstring_ws ")")
```

```
  (fun _ x _ y _ -> x + y)
```

```
printfn "%A" (run a_plus_b " ( 2 + 3 ) ")
```

(В этом примере конструкция `let` должна быть записана в одну строку.)

```
Success: 5.0
```

Для того, чтобы игнорировать лишние параметры можно использовать комбинаторы `.>>` и `>>`. внутри `pipe`.

```
//Пример вычисления суммы 2
let a_plus_b2 = pipe2
  (pstring_ws "(" >>. float_ws .>> pstring_ws "+")
  (float_ws .>> pstring_ws ")")
  (fun x y -> x + y)
printfn "%A" (run a_plus_b2 " ( 3 + 4 ) ")
```

(В этом примере конструкция `let` должна быть записана в одну строку.)

```
Success: 7.0
```

Если мы хотим просто получить кортеж значений и ничего с ними не делать, то мы можем использовать комбинаторы от `tuple2` (2 значения) до `tuple5` (5 значений).

```
//Выделение данных
let a_plus_b2_tuple = tuple2
  (pstring_ws "(" >>. float_ws .>> pstring_ws "+")
  (float_ws .>> pstring_ws ")")
printfn "%A" (run a_plus_b2_tuple " ( 3 + 4 ) ")
```

```
Success: (3.0, 4.0)
```

7.3 Пример разбора данных с применением алгебраического типа

В данном примере используется оператор `«:=»`, который позволяет изменять данные по ссылке.

```
let refVar = ref 10
refVar := 50
printfn "%d" !refVar
```

В качестве основы будем использовать следующий алгебраический тип:

```

type Expr =
  | Num of float
  | Plus of Expr * Expr
  | Minus of Expr * Expr

```

Для разбора используется следующий код:

```

//Разбор выражения
//Создаем опережающее описание для парсера
let parseExpression, implementation =
createParserForwardedToRef<Expr, unit>()
  let parsePlus = tuple2 (parseExpression .>> pstring_ws "+")
parseExpression |>> Plus
  let parseMinus = tuple2 (parseExpression .>> pstring_ws "-")
parseExpression |>> Minus
  let parseOp = between (pstring_ws "(") (pstring_ws ")") (attempt
parsePlus <|> parseMinus)
  let parseNum = float_ws |>> Num
  implementation := parseNum <|> parseOp

//Пример не будет поддерживать больше двух операций из-за
проблемы с attempt

```

Комбинатор `|>>` сохраняет результат в дискриминатор (вариант алгебраического типа).

Комбинатор `attempt` позволяет продолжить разбор, если первый вариант разбора оказался неуспешным. К сожалению, его нельзя применять более одного раза. Поэтому в нашем примере мы не можем разобрать больше двух операций. Для решения этой проблемы вместо `createParserForwardedToRef` используется `OperatorPrecedenceParser`, но пример с его использованием получается более громоздкий.

```

//Функция упрощения выражения
let rec EvalExpr(e:Expr):float =
  match e with
  | Num(num) -> num
  | Plus(a,b) ->
    let left =
      match a with
      | Num(num) -> num
      | _ -> EvalExpr(a)
    let right =
      match b with
      | Num(num) -> num

```

```

    | _ -> EvalExpr(b)
  let res1 = left + right
  printfn "%f + %f = %f" left right res1
  res1
| Minus(a,b) ->
  let left =
    match a with
    | Num(num) -> num
    | _ -> EvalExpr(a)
  let right =
    match b with
    | Num(num) -> num
    | _ -> EvalExpr(b)
  let res2 = left - right
  printfn "%f - %f = %f" left right res2
  res2

```

Пример вызова:

```

let expr1Parser = run parseExpression "(((3+4)+(1-2))+10)"
printfn "%A" expr1Parser

```

```

match expr1Parser with
| Success(result, _, _) ->
  let eval1 = EvalExpr(result)
  printfn "Результат вычислений: %f" eval1
| Failure(msg, _, _) -> printfn "Failure: %s" msg
Success: Plus (Plus (Plus (Num 3.0,Num 4.0),Minus (Num 1.0,Num 2.0)),Num 10.0)
3.000000 + 4.000000 = 7.000000
1.000000 - 2.000000 = -1.000000
7.000000 + -1.000000 = 6.000000
6.000000 + 10.000000 = 16.000000
Результат вычислений: 16.000000

```

8 Использование мультиагентного подхода в F#

Мультиагентный (многоагентный) подход является одним из понятий искусственного интеллекта —

https://ru.wikipedia.org/wiki/Многоагентная_система

В программной инженерии употребляется термин «акторный подход» - https://ru.wikipedia.org/wiki/Модель_акторов

Термины «актор» и «агент» во многом являются синонимами. И тот, и другой представляет собой программную единицу, которая выполняется в отдельном потоке и может выполнять самостоятельные действия. Но если термин «актор» в большей степени означает программный модуль, то

термин «агент» связан с более сложными функциями, такими как поведение, взаимодействие с другими агентами и т.д.

Для .NET существует очень мощная акторная библиотека АККА.NET (порт библиотеки АККА, которая написана на языке Scala). Данную библиотеку рекомендуется использовать в реальных проектах. Но данная библиотека требует некоторого времени для изучения.

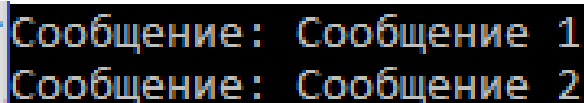
Поэтому мы познакомимся с акторным подходом в F# на примере встроенного класса MailboxProcessor.

Простейший пример актора, который печатает посланные ему сообщения:

```
let printerAgent = MailboxProcessor.Start(fun inbox->
    // обработка сообщений
    let rec messageLoop() = async{
        // чтение сообщения
        let! msg = inbox.Receive()
        // печать сообщения
        printfn "Сообщение: %s" msg
        return! messageLoop()
    }
    // запуск обработки сообщений
    messageLoop()
)
```

Операторы `async`, `let!` и `return!` используются для асинхронного программирования, которое мы к сожалению не рассматривали в курсе.

```
//Пример тестирования актора (агента)
printerAgent.Post "Сообщение 1"
printerAgent.Post "Сообщение 2"
```



```
Сообщение: Сообщение 1
Сообщение: Сообщение 2
```

Пример агента, который отгадывает числа:

```
let number = "5"
let answerAgent = MailboxProcessor.Start(fun inbox->
    // обработка сообщений
    let rec messageLoop() = async{
        // чтение сообщения
```

```
let! msg = inbox.Receive()
if msg = number then printfn "%s - ДА" msg
else printfn "%s - НЕТ" msg
return! messageLoop()
}
// запуск обработки сообщений
messageLoop()
)
```

Вызов агента:

```
for i in [1..10] do
    answerAgent.Post(i.ToString())
```

```
1 - НЕТ
2 - НЕТ
3 - НЕТ
4 - НЕТ
5 - ДА
6 - НЕТ
7 - НЕТ
8 - НЕТ
9 - НЕТ
10 - НЕТ
```

9 Задания по курсу

9.1 Лабораторные работы

9.1.1 Лабораторная работа №1

На основе рассмотренного примера составить программу на функциональном языке программирования для решения биквадратного уравнения с использованием алгоритма рассмотренного в разделе «Биквадратное уравнение» статьи https://ru.wikipedia.org/wiki/Уравнение_четвёртой_степени. Программа должна использовать алгебраические типы и механизм сопоставления с образцом.

В случае комплексных корней их вычисление не обязательно, можно выводить информацию о том, что корни комплексные.

9.1.2 Лабораторная работа №2

1. Создайте два варианта функции, которая возвращает кортеж значений. Первый вариант принимает на вход параметры в виде кортежа, второй вариант параметры в каррированном виде.
2. Выберите простой алгоритм, который может быть реализован в виде рекурсивной функции и реализуйте его в F#. Пример – вычисление суммы целых чисел в заданном диапазоне.
3. Преобразуйте разработанную рекурсивную функцию в форму хвостовой рекурсии.
4. По аналогии с пунктом 3.7.4 разработайте конечный автомат из трех состояний и реализуйте его в виде взаимно-рекурсивных функций.
5. На основе пунктов 3.7.7 и 3.7.8 разработайте функцию, которая принимает 3 целых числа и лямбда-выражение для их суммирования в виде кортежа и в каррированном виде.

9.1.3 Лабораторная работа №3

1. Разработайте функцию, которая принимает три параметра обобщенных типов и возвращает их в виде кортежа. Модифицируйте функцию: не указывая явно типы параметров, задавая выражения в теле функции, сделайте так, чтобы параметры были типов `int`, `float`, `string`.
2. С использованием двухэтапного создания обобщенных функций реализуйте функции, которые осуществляют сложение:
 - трех аргументов типа `int`;
 - трех аргументов типа `float`;
 - трех аргументов типа `string`.
3. С использованием `list comprehension` для четных элементов списка `[1..10]` верните список кортежей. Каждый кортеж содержит элемент списка, его квадрат и куб.
4. На основе пункта 3.8.1 напишите два варианта функции, которая принимает на вход список и возвращает квадраты его значений. Необходимо использовать свойства списка `Head` и `Tail`. Первый вариант функции использует оператор `if`, второй вариант использует сопоставление с образцом на уровне функции.
5. Последовательно примените к списку функции `map`, `sort`, `filter`, `fold`, `zip`, функции агрегирования. Функции применяются в любом порядке и произвольно используются в трех комбинациях.
 - Первая комбинация заканчивается функцией агрегирования (например, сумма элементов списка). Список предварительно может быть отсортирован, отфильтрован и т.д.
 - Вторая комбинация заканчивается функцией `fold`, которая осуществляет свертку списка. Вторая комбинация выполняет те же действия, что и первая комбинация и должна возвращать такой же результат.

- Третья комбинация заканчивается функцией zip, которая соединяет два списка.
6. Реализуйте предыдущий пункт с использованием оператора потока «|>».
 7. Реализуйте предыдущий пункт с использованием оператора композиции функций «>>».

9.1.4 Лабораторная работа №4

Часть 1. Использование классов, интерфейсов и наследования.

Разработать программу, реализующую работу с классами.

1. Программа должна быть разработана в виде консольного приложения на языке F#.
2. Абстрактный класс «Геометрическая фигура» содержит виртуальный метод для вычисления площади фигуры.
3. Класс «Прямоугольник» наследуется от класса «Геометрическая фигура». Ширина и высота объявляются как свойства (property). Класс должен содержать конструктор по параметрам «ширина» и «высота».
4. Класс «Квадрат» наследуется от класса «Прямоугольник». Класс должен содержать конструктор по длине стороны.
5. Класс «Круг» наследуется от класса «Геометрическая фигура». Радиус объявляется как свойство (property). Класс должен содержать конструктор по параметру «радиус».
6. Для классов «Прямоугольник», «Квадрат», «Круг» переопределить виртуальный метод Object.ToString(), который возвращает в виде строки основные параметры фигуры и ее площадь.
7. Разработать интерфейс IPrint. Интерфейс содержит метод Print(), который не принимает параметров и возвращает void. Для классов «Прямоугольник», «Квадрат», «Круг» реализовать наследование от

интерфейса IPrint. Переопределяемый метод Print() выводит на консоль информацию, возвращаемую переопределенным методом ToString().

Часть 2. Использование алгебраического типа и сопоставления с образцом.

1. Реализуйте класс геометрическая фигура в виде алгебраического типа (discriminated union), который содержит варианты (дискриминаторы) «Прямоугольник», «Квадрат», «Круг» с необходимыми параметрами.
2. Разработайте для данного класса функцию вычисления площади. Функция должна принимать параметр типа «геометрическая фигура» и вычислять различные варианты площади в зависимости от дискриминатора. Необходимо использовать механизм сопоставления с образцом. (Используйте пример вычисления корней квадратного уравнения).

9.1.5 Лабораторная работа №5

Для произвольно выбранного типа данных (например, Maybe) реализуйте функции функтора, аппликативного функтора, монады.

Проверьте для Вашей реализации справедливость соответствующих законов для функтора и аппликативного функтора (тех законов, которые можно проверить с использованием F#). Некоторые законы могут не выполняться. Это означает что данный тип не является в полной мере функтором, аппликативным функтором, монадой.

9.1.6 Лабораторная работа №6

Разработайте программу, которая осуществляет разбор текста с использованием библиотеки FParsec. Результатом разбора должны быть значения алгебраического типа.

9.1.7 Лабораторная работа №7

С использованием класса MailboxProcessor реализуйте агента, который реагирует на внешние события и выполняет различные действия (например, выдает результаты в консоль).

9.2 Домашнее задание

Домашнее задание – это реферат или статья по тематике курса.

Примерный список тем ДЗ:

1. Рассмотрение любой парадигмы программирования или конкретного языка на основе статьи из Википедии – https://ru.wikipedia.org/wiki/Парадигма_программирования
2. Углубленные разделы языка Пролог.
3. Рассмотрение теоретических основ лямбда-исчисления, лежащего в основе функционального программирования – <https://ru.wikipedia.org/wiki/Лямбда-исчисление>
4. Рассмотрение основных парадигм логического вывода:
https://en.wikipedia.org/wiki/Forward_chaining
https://en.wikipedia.org/wiki/Backward_chaining
5. Рассмотрение систем программирования на основе правил (<https://ru.wikipedia.org/wiki/BRMS>) в частности <https://en.wikipedia.org/wiki/Drools>