Вступление

Целью данной статьи стоит популяризация функциональных языков, на примере работы с F#. Основной упор будет сделан на решение конкретной задачи с использованием парадигмы функционального языка. Автор не ставит своей целью обучению F#, но, по возможности будет пояснять код и принцип его написания таким образом, чтобы читатель, не знакомый или мало знакомый с функциональными языками, смог понять идею и ее реализацию в функциональном стиле.

Также стоит отметить, что как и везде в программировании, одну и ту же задачу можно решать разными способами. Данная статья не является исключением и в ней автор предлагает свое решение и не исключает того, что могут быть решения лучше.

Для начала кратко пробежимся по основам, чтобы читателю, который впервые столкнулся с F#, было понятно, о чем пойдет речь далее.

Итак, как уже говорилось ранее, F# - это функциональный язык, то есть в основе которого лежит парадигма функционального программирования. F# был разработан командой Microsoft на базе другого функционального языка Ocalm. Чем же отличается функциональный подход от императивного? Кратко, императивное программирование построено на основе переменных, а функциональное - на основе констант. Большинство современных высокоуровневых языков программирования являются императивными, в которых широко используются переменные. Примерами таких языков являются C/C++, C#, Java и так далее. Если вы программист, то скорее всего вы работаете с одним из таких языков и используете императивный стиль. Но в последнее время, по мере увеличения вычислительных мощностей компьютеров, повышается интерес и популярность функциональных языков. Одним из таких языков является F#, который и будет рассмотрен в данной статье. Особенностью F# является то, что он не является чистым функциональным, то есть допускает возможность работы в императивном стиле. Но авторы языка все таки делают основной упор на то, чтобы по возможности использовать именно функциональный стиль и для того, чтобы перейти на императивный стиль, нужно использовать специальный синтаксис. Это как лишнее напоминание программисту о том, что он работает все таки с функциональным языком. Еще одной особенностью F#, которая и делает его столь привлекательным, является то, то он интегрирован в общую платформу .NET. А следовательно, программист F# может пользоваться всей мощью готового функционала .NET, взаимодействовать с другими языками

.NET, и в то же время, пользоваться преимуществами функционального программирования.

Одним из первых вопросов, которые возникают у читателей, в чем же преимущество констант перед переменными? В функциональных языках константа является результатом выполнения некоторой функции, которая на входе принимает аргументы, а на выходе отдает результат вычисления. При этом функция работает таким образом, что получая одни и те же аргументы, она гарантированно будет возвращать одни и те же результаты. Это возможно благодаря тому, что функциональная парадигма не предусматривает наличие переменных. А это значит, что внутреннее состояние функции при стабильных аргументах всегда будет стабильно одинаковым и программист точно знает, что у него не возникнут неожиданности, которые он должен будет проверять и реагировать при императивном подходе.

Побочным эффектом функционального подхода является то, что компилятор может взять на себя большой кусок работы по автоматическому определению типов данных и программист может писать меньше кода. Конечно, программист может вручную указать компилятору о том, какой тип данных используется. Иногда это и придется делать, когда компилятор не может этого сделать автоматически и уме нужно больше данных, или если программист хочет сделать код более понятным, явно указывая тип аргументов. Но в большинстве случаев этого делать не нужно.

Так как автор статьи не ставит своей целью обучению F#, то он отсылает читателя к учебникам, список которых будет приведен в конце статьи. Дальше мы начнем решать практическую задачу и будем смотреть, как она может быть решена на F#.

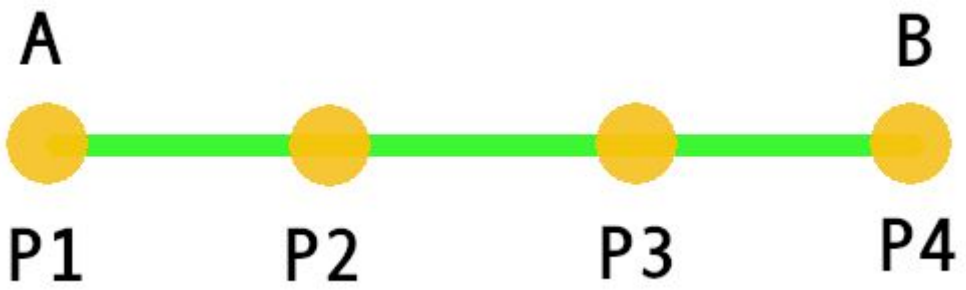
Постановка задачи

Функциональные языки работают хорошо не везде. Например, если нужно работать с UI и обеспечить взаимодействие с пользователем, то удобнее использовать уже привычные языки. Поэтому использование функциональных языков везде только ради того, чтобы их использовать - не самая хорошая идея. Поэтому и мы себе поставим задачу, более подходящую для функциональных языков.

Предположим, что у нас имеется некий массив объектов, которые мы назовем “дорога” (*Road*). Каждая дорога содержит в себе список точек, через которые

она проходит. Задача заключается в том, чтобы преобразовать данный список в граф, в котором узлами будут являться точки пересечения дорог, а ребрами - отрезки узлами. Будем считать, что две дороги пересекаются, если в их внутренних списках точек есть точки, имеющие одинаковые координаты. Под графом примем объект, содержащий в себе матрицу смежностей, которая описывает данный граф. Будем считать, что наш граф не имеет направления, то есть между узлами А и Б нет определенного направления и движение возможно как от узла А к узлу Б, так и наоборот. Ниже представлен поясняющий рисунок.

Рис. 1 (объект Road)



Здесь представлен объект *Road*, который содержит в себе список из четырех точек P1, P2, P3, P4. При этом только точки P1 и P4 (A и B) в полученном графе будут представлены в виде вершин.

Подобный тип в F# можно описать при помощи “записей”. Создадим новую запись *Road*.

```
type Road = { name: string; points: Point list }

// name - строковое имя road
// points - список точек, из которых road состоит
```

К этому типу мы еще вернемся позже, а пока подробнее распишем новый тип *Point*. Как уже было написано выше, этот тип инкапсулирует в себе знания о точке.

```
type Point = { x: float; y: float }

// x - координата по оси x (горизонтальная)
// y - координата по оси y (вертикальная)
```

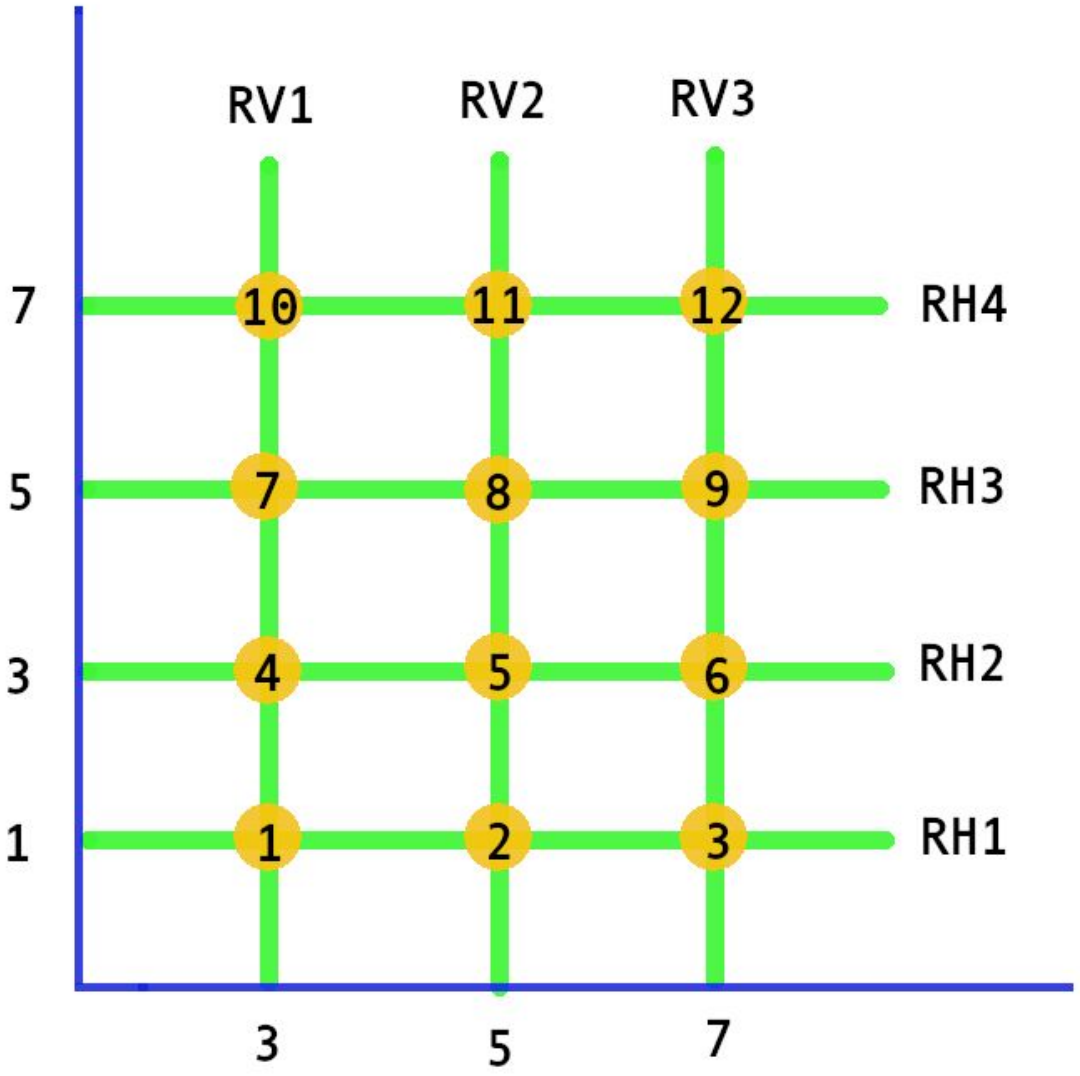
F# позволяет расширять функционал записей при помощи методов и операторов, поэтому добавим пару полезных операторов, которые могут быть использованы в дальнейшем для сравнения двух точек

```
type Point = { x: float; y: float }
  with
    static member (==) (p1, p2) = p1.x = p2.x && p1.y = p2.y
    static member (!=) (p1, p2) = p1.x <> p2.x || p1.y <> p2.y
  end
```

Пока что ничего сложного нет, поэтому продолжим.

Далее на рисунке показана структура дорог, которую мы будем использовать для создания графа. Горизонтальная ось координат - это ось X, а вертикальная, соответственно - Y.

Рис. 2 (список Road в системе координат x, y)



Здесь “RH1”-”RH4” - имена дорог, которые параллельны оси X, а “RV1”-”RV3” - которые параллельны оси Y (road horizontal/vertical).

Попробуем создать объект, который будет описывать нужную нам структуру дорог.

```
let rh1 = { name = "rh1"; points = [
                                        { x = 1.; y = 7. };
                                        { x = 3.; y = 7. };
                                        { x = 5.; y = 7. };
                                        { x = 6.; y = 7. };
                                        { x = 8.; y = 7. }
                                    ]
          }
```

```
// По аналогии создаем дороги rh2, rh3, rh4

let rv1 = { name = "rv1"; points = [
                                        { x = 3.; y = 8. };
                                        { x = 3.; y = 7. };
                                        { x = 3.; y = 5. };
                                        { x = 3.; y = 3. };
                                        { x = 3.; y = 1. };
                                        { x = 3.; y = 0. }
                                    ]
          }
// По аналогии создаем дороги rv2, rv3
```

Как видим, получилось достаточно громоздко и трудно читаемо. Попробуем немного модифицировать код, чтобы облегчить процесс создания нужного нам объекта.

```
let point (x, y) = { x = x; y = y }
let road n pl = { name = n; points = pl }
```

Здесь мы пишем две функции, первая создает запись типа Point, а вторая - запись типа Road. Компилятор автоматически вычисляет, какие типы записей вы хотите использовать без их явного указания, так как мы перечисляем имена параметров и присваиваем им значения правильного типа. Этой информации компилятору достаточно, чтобы избавить программиста от явного указания типа.

Теперь процесс инициализации структуры дорог будет выглядеть немного проще. Визуально уже можно понять, что к чему относится.

```
let rh1 =
        road "rh1" [
                        point (1., 7.);
                        point (3., 7.);
                        point (5., 7.);
                        point (7., 7.);
                        point (8., 7.);
                    ]


// По аналогии создаем дороги rh2, rh3, rh4
```

```
let rv1 =
        road "rv1" [
                        point (3., 8.);
                        point (3., 7.);
                        point (3., 5.);
                        point (3., 3.);
                        point (3., 1.);
                        point (3., 0.);
                    ]
// По аналогии создаем дороги rv2, rv3


// Создает список дорог, который в дальнейшем будет
преобразовываться в граф
let roads = [rh1; rh2; rh3; rh4; rv1; rv2; rv3]
```

Мы получили список дорог *roads* и теперь нам нужно его преобразовать в граф.

Теория говорит о том, что граф - это объект, состоящий из двух множеств

```
type Graph = { vertexs: Set<Point>; edges: Set<Edge>; }
```

vertexs - это множество, содержащее в себе вершины графа
edges - это множество, содержащее в себе ребра графа

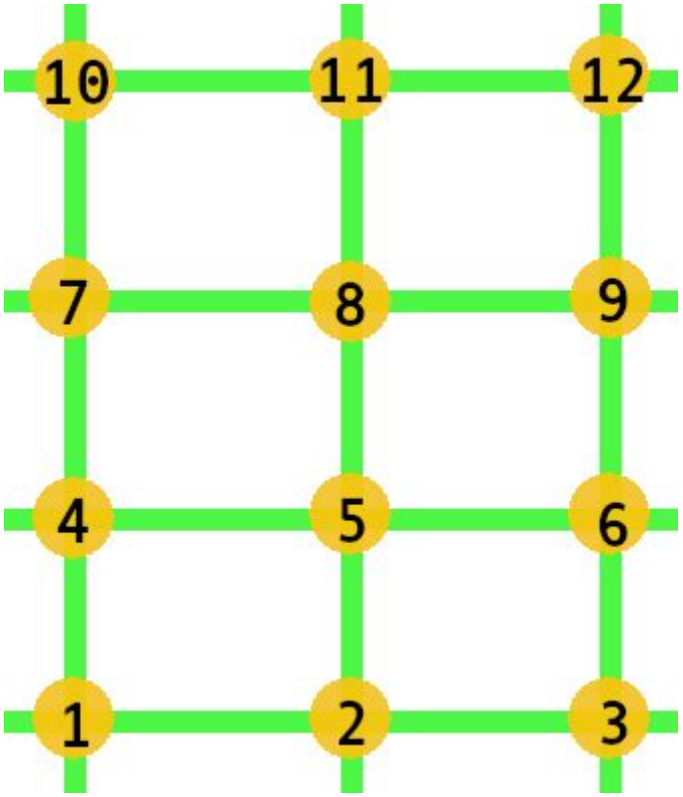
Ребро графа мы может обозначить при помощи следующей записи.

```
type Edge = { u: Point; v: Point }
```

В программировании данное определение графа не является удобным и оптимальным, поэтому мы будем пользоваться понятием “матрица смежности”.

Матрицу смежности проще понять, если визуально посмотреть на граф и на соответствующую ему матрицу смежности.

Рис. 3 (граф и соответствующая ему матрица смежности)



	1	2	3	4	5	6	7	8	9	10	11	12
1		x		x								
2	x		x		x							
3		x				x						
4	x				x		x					
5		x		x		x		x				
6			x		x				x			
7				x				x		x		
8					x		x		x		x	
9						x		x				x
10							x				x	
11								x		x		x
12									x		x	

Внимательно рассмотрим полученную матрицу. Ее размерность равна квадрату вершин графа, в нашем случае 12x12 ячеек, так как в графе 12 вершин. Недостатком матрицы смежностей является высокое расходование оперативной памяти и излишество, если граф разреженный и имеет небольшое количество ребер. Достоинством - быстрый поиск зависимостей

между вершинами, а также - быстрая вставка или удаление ребра. Однако, высокое потребление памяти будет в том случае, если для хранения мы будем использовать типы, занимающие много памяти, например Int32. Для построение графа достаточно матрицы не целых чисел, а битовая матрица, где элементом ячейки является один бит или булевое значение. Также, матрицу можно обработать алгоритмом архивации, что еще позволит сэкономить память. Поэтому, мы не будем обращать внимания на данные недостатки и для упрощения будем использовать матрицу Int, где значение 1 будет соответствовать X в таблице выше.

Итак, если сравнить между собой матрицу смежностей и граф на рисунке 3, то можно легко найти зависимость матрицы от графа. По горизонтали будут расположены вершины, из которых исходят ребра, а по вертикали вершины, в которые ребра входят. Таким образом, если мы хотим найти, с какими вершинами графа связана вершина 1, то мы просто смотрим горизонтальную строку, обозначенную как 1 и ищем X в ячейках этой строки. если X установлено, то это значит, что вершина 1 соединена с соответствующей вершиной из горизонтального столбика. Согласно таблице из рисунка 3, вершина 1 связана с вершинами 2 и 4 исходящими ребрами.

Таким образом, на основе матрицы смежностей можно также моделировать граф, у которого можно задавать направление ребер между вершинами.

Таким образом, наша задача сводится к тому, чтобы из списка, содержащего дороги, построить матрицу смежностей. Далее, на основе этой матрицы можно будет построить два множества (множество вершин и множество ребер), если это будет нужно.

Пишем алгоритм

Безусловно, задачу по построению графа через матрицу смежностей можно решить по разному, несколькими способами. Какой то из них будет более эффективен, какой то менее. Автор данной статьи не ставил своей целью найти максимально быстрый и экономный алгоритм. Данная задача была выбрана для того, чтобы опробовать функциональный стиль F# на практике и посмотреть, что из этого получится.

Итак, повторим поставленную задачу. У нас на входе имеется список *Road*, каждая из которых в свою очередь содержит список *Point*. А на выходе нужно получить матрицу смежностей, на основе которой можно будет построить граф. Вершинами графа будут считаться *Points*, которые являются точками

пересечения *Roads*. А ребрами - отрезки между соседними вершинами. Пересечением двух *Roads* будем считать *Points*, у которых координаты X и Y одинаковые. В статье ранее мы уже обозначили подобные типы через записи, которые инкапсулируют в себе знания о самом типе и расширенные полезными методами. Сразу отмечу, что методов будет больше и мы их будем добавлять по мере прохождения статьи.

Кажется очевидным, что алгоритм должен уметь находить вершины (точки пересечения) дорог. А далее, на основе этих вершин можно строить матрицу смежностей. Безусловно, можно найти и другие способы решения алгоритма, но в данной статье мы остановимся на следующем:

1. Вычисляем все вершины графа, то есть находим все точки пересечения всех дорог.
2. Строим матрицу смежностей на основе знаний о вершинах и списка дорог.

Нам потребуется функция, которая сможет определить, находится ли точка в каком нибудь *Road*, а точнее - в списке *Point*, из которых *Road* состоит. Оформим данную функцию в виде активного шаблона, так как в дальнейшем она будет использована в конструкциях match/with

```
let (|Contains|DoesntContains|) (point, points) =
    let isExists = List.exists (fun p ->
                                    if point == p then true
                                    else false) points
    if isExists then Contains point
    else DoesntContains
```

Данный активный шаблон на входе принимает два аргумента
point - запись типа *Point*
points - список, в котором нужно найти *Point*

Обратите внимание, что здесь можно не указывать тип аргументов, так как из контекста активного шаблона компилятор сам выводит эти типы. Для наглядности можно было бы записать так

```
let (|Contains|DoesntContains|) (point: Point, points: Point
list) =
    // здесь остальная часть активного шаблона
```

Но мы будем исходить из того, чтобы по возможности перекладывать на компилятор груз знаний, какие типы у аргументов, а сами будем максимально лениться и явное указание типов опускать.

Так как это активный шаблон, то у него нету явного имени и он может быть использован в конструкциях match/with. Компилятор сам будет определять, какой активный шаблон нужно использовать, исходя из контекста.

Также мы можем заметить, что этот активный шаблон возвращает два типа значений

Contains - означает, что точка находится на линии, плюс возвращает и саму точку. Это кажется лишним, но такой подход удобно использовать в некоторых случаях
DoesntContains - означает, что точка не лежит на линии.

Рассмотрим, как этот активный шаблон работает. Сначала он пытается найти точку в списке путем сравнения

```
let isExists = List.exists (fun p ->
                                if point == p then true
                                else false) points
```

Для поиска мы использует метод *exists* из стандартной библиотеки .Net у класса типа *Set<T>*. В круглых скобках объявляется функция, которая должна вернуть true/false. Обратите внимание, что для сравнения двух точек мы используем оператор ==, которым была расширена запись *Point* выше.

```
type Point = { x: float; y: float; z: float }
    with
        // returns true if two points are equal
        static member (==) (p1, p2) =
            p1.x = p2.x && p1.y = p2.y && p1.z = p2.z
        static member (!=) (p1, p2) =
            p1.x <> p2.x || p1.y <> p2.y || p1.z <> p2.z
    end
```

По умолчанию F# не поддерживает операторы сравнения == или !=. Вместо них используются операторы = или <>, но так как у нас записи *Point* сложные и операторы сравнения могут иметь сложную логику, то мы их просто переопределим и вручную пропишем, как они должны работать.

Далее, активный шаблон проверяет результат работы функции *List.exists*, и если он не равен нулю, то возвращает *Contains*, содержащий в себе точку. Иначе возвращает *DoesntContains*

```
if isExists then Contains point
else DoesntContains
```

Как видим, в написании активных шаблонов нет ничего сложного и они очень похожи на обычные функции.

Теперь мы готовы к написанию функции, которая будет искать все вершины в исходном списке *Road*. Перед тем, как начинать писать функцию, нужно представить, как мы это будем делать? Кажется очевидным, что для поиска всех общих *Point* (вершин), нужно перебрать все возможные комбинации *Road*. К примеру, у нас есть список из 10 *Road*. Это значит. что из этого списка мы должны взять *Road1*, получить из него список *Point* и этот список применить ко всем оставшимся *Road*, чтобы найти, содержать ли они эту *Point*. Если содержат, то этот *Point* будет считаться вершиной и добавлена в конечный список, который вернет функция. Далее мы берем *Road2*, получаем из него список его *Point* и снова применяем его ко всем оставшимся *Road*. И так делаем до тех пор, пока не пройдемся по всему списку *Road*.

Исходя из алгоритма выше, кажется вполне очевидным следующая последовательность.

```
for r1 in roads do
    for r2 in roads do
        if r1.name <> r2.name then
            // делаем нужную проверку и ищем результат
```

Как вы уже заметили, присутствует проверка

```
if r1.name <> r2.name then
```

Это нужно сделать для того, чтобы алгоритм не допускал, чтобы дорога сканировала саму себя, так как в таком случае результат поиска всегда будет положительным и всегда будет неправильным. То есть, мы получим неверные результаты, что скажется на дальнейшей работе алгоритма.
Однако, двойной цикл нам не годится и вот почему. Цикл - это просто перебор вариантов, и этот перебор не возвращает никакого значения. В F# цикл *for* эквивалентен следующему коду

```
List.iter (fun r -> (*здесь какое то действие*)) roads
```

Функция, которая вызывается из *List.iter*, возвращает тип *unit* (то есть ничего не возвращает). Нам же нужно, чтобы перебор что то возвращал. В нашем конкретном случае мы должны получить список вершин. Это возможно сделать при помощи функций-сверток, например следующей

```
let res = List.fold (fun acc elem -> (*здесь какое то действие*)) [] source
```

Функция-свертка *List.fold* в качестве аргументов получает функцию обработчик, начальное значение аккумулятора и список с исходными данными. Текущее значение аккумулятора будет передаваться в функцию-обработчик при каждом вызове и задача функции-обработчика добавлять к аккумулятору результат текущей итерации. После окончания работы свертки возвращается итоговое накопленное значение аккумулятора. То есть функция *List.fold* по аналогии *List.iter* обрабатывает каждый элемент исходного списка, но возвращает некоторое итоговое значение (в нашем случае это будет массив вершин).

Мы расширим запись *Road* новой статической функцией, которая будет искать все вершины. Ниже будет приведен исходный код функции и мы ее рассмотрим более детально.

```
type Road = { name: string; points: Point list }
  with
    static member Vertexs roads =

      let res =
        List.fold
          (fun acc elem ->
             let filtered = List.filter (fun f -> f.name <> elem.name) roads
             List.fold (fun acc2 elem2 ->
                          let points =
                            List.filter
                              (fun p ->
                                 match (p, elem2.points) with
                                 | Contains(x) -> not (List.exists ((==) x) acc)
                                 | _ -> false
                                 ) elem.points
                           acc2 @ points
             ) acc filtered
          ) [] roads

      res |> List.iteri (fun i f -> printfn "vertex %d x: %f, y: %f" (i+1) f.x
f.y)
      res
  end
```

С функцией *List.fold* мы уже познакомились и поняли, что она работает как *for*, но дополнительно еще возвращает результат. Так как алгоритм предполагает двойной цикл, то мы также имеем двойной *List.fold* (вернее, один вложен в другой). Основную работу по поиску вершин делает внутренний *List.fold* (также он добавляет найденные результаты в аккумулятор). Задача внешнего *List.fold* - перебрать все возможные варианты из списка *Road*. Также мы в него добавили проверку, чтобы *Road* не сканировал сам себя. Это делается при помощи фильтра

```
let filtered = List.filter (fun f -> f.name <> elem.name) roads
```

И уже отфильтрованные данные передаются во внутренний *List.fold*. Это несколько отличается от исходных размышлений, но результат точно такой же. Можно легко сделать как в исходном алгоритме, просто добавив вовнутрь второго *List.fold* проверку имен. Но в таком случае мы получим больше переборов, поэтому оставим как есть.

Давайте рассмотрим внутреннюю свертку подробнее.

```
let filtered = List.filter (fun f -> f.name <> elem.name) roads
List.fold (fun acc2 elem2 ->
             let points =
               List.filter
                 (fun p ->
                    match (p, elem2.points) with
                    | Contains(x) -> not (List.exists ((==) x) acc)
                    | _ -> false
                  ) elem.points
              acc2 @ points
           ) acc filtered
```

На вход внутренней свертки *List.fold* подаются следующие параметры

acc - уже накопленный аккумулятор внешней свертки
filtered - предварительно отфильтрованный список *Road*
fun acc2 elem2 -> внутренняя свертка получает текущий аккумулятор и текущий элемент из отфильтрованного списка

Далее все просто. При помощи фильтра *List.filter* мы получаем список совпадающих точек (вершин). Для поиска точки внутри списка мы используем активный шаблон, который рассматривали ранее. Полученные вершины прикантовываем к аккумулятору. Остается только пояснить следующий код

```
| Contains(x) -> not (List.exists ((==) x) acc)
```

Здесь, в случае если точка была определена как вершина, мы ищем ее в итоговом аккумуляторе. Если ее там нет, тогда конструкция вернет *true*, в противном случае *false*. Если мы не будем делать такой проверки, то в итоговом списке вершины вернутся дважды и вот почему.

Допустим, что у нас есть список из двух *Road*. Когда мы получаем список точек из *Road1* и ищем совпадения в *Road2*, то все хорошо. По далее, когда мы получаем список точек из *Road2* и начинаем искать совпадения в *Road1*, то фактически мы найдем те же самые точки, которые были найдены в предыдущей итерации. То есть результат будет продублирован дважды, то есть неправильно. Данная проверка исключает дублирование и мы получим верный результатов.

Как вы помните, в итоговом коде функции также была такая строчка

```
res |> List.iteri (fun i f -> printfn "vertex %d x: %f, y: %f" (i+1) f.x f.y)
```

Она нужна всего лишь для визуальной отладки, чтобы во время тестов мы видели результат. Давайте запустим функцию и посмотрим, что получилось.

Исходный список *Road* у нас уже есть (мы его сформировали выше по статье), теперь вызовем функцию для поиска вершин

```
Road.Vertexs roads
```

Вот результат

```
vertex 1 x: 3.000000, y: 7.000000
vertex 2 x: 5.000000, y: 7.000000
vertex 3 x: 7.000000, y: 7.000000
vertex 4 x: 3.000000, y: 5.000000
vertex 5 x: 5.000000, y: 5.000000
vertex 6 x: 7.000000, y: 5.000000
vertex 7 x: 3.000000, y: 3.000000
vertex 8 x: 5.000000, y: 3.000000
vertex 9 x: 7.000000, y: 3.000000
vertex 10 x: 3.000000, y: 1.000000
vertex 11 x: 5.000000, y: 1.000000
vertex 12 x: 7.000000, y: 1.000000
```

Убеждаемся, что он верный и продолжаем. Нам остается построить матрицу смежностей на основе полученного списка вершин и списка дорог.

Построение матрицы смежностей

Здесь рассмотрим функцию, которая строит матрицу смежностей на основе готового списка вершин и списка дорог

```
let makeMatrix (vertexs: Point list) (roads: Road list) =

    let matrix = Array2D.init vertexs.Length vertexs.Length (fun _ _ -> (0))

    let inMatrix row col road =
        let vertexFrom = vertexs.[row]
        let vertexTo = vertexs.[col]

        let u = match (vertexFrom, road.points) with
                | Contains(x) -> Some(x)
                | _ -> None
        let v = match (vertexTo, road.points) with
                | Contains(x) -> Some(x)
                | _ -> None

        match (u, v) with
        | (Some(x), Some(y)) ->
            let iu = List.findIndex ((==) x) road.points
            let iv = List.findIndex ((==) y) road.points

            let isAnother = [(System.Math.Min (iu, iv) + 1)..(System.Math.Max
(iu, iv) - 1)]
                            |> List.exists (fun i ->
                                            (List.exists ((==) road.points.[i])
vertexs))

            if not isAnother then
                matrix.[row, col] <- 1
            | _ -> ()

            matrix
            |> Array2D.iteri (fun col row i ->
                              roads
                              |> List.iter (fun road ->
                                   if row <> col then
                                      inMatrix row col road)
                             )

            matrix
```

Как видно, функция *makeMatrix* принимает два параметра

vertexs - список *Point*. То есть здесь мы передаем уже ранее сформированный список найденных вершин графа
roads - список *Road*. Он нам все еще нужен будет для построения матрицы.

В самом начале мы объявляем двумерную матрицу смежностей и инициализируем ее нулями. Размерность матрицы по вертикали и по горизонтали равна количеству вершин графа.

```
let matrix = Array2D.init vertexs.Length vertexs.Length (fun _ _ -> (0))
```

В случае нахождения соответствия, мы ячейку будем инициализировать 1. Как уже говорилось ранее, использование подобной матрицы только лишь для того, чтобы хранить в ней 0 и 1 является нерациональным с точки зрения расходования памяти. Задача более оптимального хранения матрицы в памяти не входит в задачу данной статьи и рассматриваться здесь не будет.

Внутри функции *makeMatrix* мы добавляем внутреннюю функцию

```
let inMatrix row col road =
```

Данная функция будет выполнять основную работу по заполнению матрицы смежностей и принимает на вход следующие параметры

row - номер строки, то есть номер исходящей вершины
col - номер колонки, то есть входная вершина

Далее, по номер строки и колонки мы получаем исходящую и входную вершины из готового списка

```
let vertexFrom = vertexs.[row]
let vertexTo = vertexs.[col]
```

Следующим шагом мы определяем, находятся ли эти две вершины на одной линии, при помощи активного шаблона, который мы уже использовали раньше

```
let u = match (vertexFrom, road.points) with
        | Contains(x) -> Some(x)
        | _ -> None
let v = match (vertexTo, road.points) with
        | Contains(x) -> Some(x)
        | _ -> None
```

u и *v* имеют тип *Point option*, то есть это опциональный тип, который может либо содержать значение, либо *null*. Нас интересует только тот случай, когда оба значения чему то равны. Это будет означать, что обе вершины лежат на одной прямой и их можно анализировать, чтобы решить, есть между ними ребро или нету? Мы будем это делать при помощи матчинга

```
match (u, v) with
| (Some(x), Some(y)) -> // здесь делаем нужный нам анализ
```

```
| _ -> () // здесь ничего не делаем
```

Из фрагмента выше видно, что мы запускаем процедуру анализа только в том случае, если опциональные *u* и *v* чему то равны и здесь с ними ассоциированы уже не опциональные *x* и *y* (они имеют тип *Point*). Это удобно, так как не нужно будет делать дополнительные проверки в дальнейшем или принудительно извлекать значения из опционов.

Анализ и инициализация выглядит так

```
let iu = List.findIndex ((==) x) road.points
let iv = List.findIndex ((==) y) road.points

let isAnother = [(System.Math.Min (iu, iv) + 1)..(System.Math.Max (iu, iv) - 1)]
                |> List.exists (fun i ->
                                    (List.exists ((==) road.points.[i]) vertexs))

if not isAnother then
   matrix.[row, col] <- 1
```

Сначала мы получаем индексы вершин из соответствующих списков

```
let iu = List.findIndex ((==) x) road.points
let iv = List.findIndex ((==) y) road.points
```

Например, в списке *road.points* вершина *x* находится в самом начале. Значит, ее индекс будет равен 0. Если в конце списка, то индекс будет равно *Length-1*.

После того, как мы получили индексы обеих вершин, нужно убедиться, что они являются смежными, то есть соседями, и что между ними нет никаких других вершин. И только в случае, если они соседи, мы можем инициализировать соответствующую ячейку матрицы смежностей 1. Посмотрим, как это делается.

```
let isAnother = [(System.Math.Min (iu, iv) + 1)..(System.Math.Max (iu, iv) - 1)]
                |> List.exists (fun i ->
                                (List.exists ((==) road.points.[i]) vertexs))
```

Следующая конструкция является генератором списков

```
[(System.Math.Min (iu, iv) + 1)..(System.Math.Max (iu, iv) - 1)]
```

Он формирует диапазон индексов между найденными минимальным и максимальным индексами (не включая исходные индексы). Стоит отметить следующее, что в случае такого генератор списков

```
[(iu + 1)..(iv - 1)]
```

мы получим матрицу смежностей однонаправленного графа. То есть такого, где ребра есть только между вершинами с меньшим индексом к большему. То есть матрица была бы заполнена только наполовину. Поэтому при построении генератора списков нужно предварительно искать минимальные и максимальные значения индексов, чтобы ребра считались также и в обратную сторону.

Осталось совсем чуть чуть

```
|> List.exists (fun i -> (List.exists ((==) road.points.[i]) vertexs))
```

Для того, чтобы понять, являются ли две вершины соседями, нужно всего лишь посмотреть, являются ли промежуточные точки вершинами. Для этого по индексу получаем *Point* и ищем ее в списке *vertexs* (*List.exists*).

В итоге работы всей конструкции *isAnother* устанавливается в *false*, если промежуточная точка не является вершиной. И в этом случае мы инициализируем ячейку матрицы в 1

```
if not isAnother then
    matrix.[row, col] <- 1
```

С работой функции *inMatrix* мы разобрались, теперь осталось только ее вызвать и вернуть полученную матрицу.

```
matrix
|> Array2D.iteri
    (fun col row i ->
       roads
       |> List.iter (fun road ->
                        if row <> col then
                           inMatrix row col road)
    )

// возвращаем полученную матрицу
matrix
```

Рассмотрим подробнее.

Следующая конструкция перебирает все элементы двумерной матрицы и вызывает функцию-обработчик, куда передает колонку, строку и значение ячейки соответственно.

```
 matrix
|> Array2D.iteri function
```

Функция обработчик в свою очередь перебирает все элементы списка дорог (*roads*) и в случае, когда номер колонки и строки не совпадают (в противном случае мы всегда будет в матрице иметь заполненную единицами главную диагональ), вызываем функцию *inMatrix*

```
(fun col row i ->
       roads
       |> List.iter (fun road ->
                        if row <> col then
                           inMatrix row col road)
)
```

На этом все, остается только вернуть готовую матрицу.

Теперь расширим запись *Graph* статическим методом, который будет формировать матрицу смежностей. Код целиком можно посмотреть ниже. Его работа должна быть понятной, так как мы очень подробно его разобрали выше

```
type Graph = { vertexs: Set<Point>; edges: Set<Edge>; }
  with

    static member ofRoads roads =

      let makeMatrix (vertexs: Point list) (roads: Road list) =

        let matrix = Array2D.init vertexs.Length vertexs.Length (fun _ _ -> (0))

        let inMatrix row col road =

          let vertexFrom = vertexs.[row]
          let vertexTo = vertexs.[col]

          let u = match (vertexFrom, road.points) with
                  | Contains(x) -> Some(x)
                  | _ -> None
          let v = match (vertexTo, road.points) with
                  | Contains(x) -> Some(x)
                  | _ -> None

          match (u, v) with
          | (Some(x), Some(y)) ->
               let iu = List.findIndex ((==) x) road.points
               let iv = List.findIndex ((==) y) road.points

               let isAnother = [(System.Math.Min (iu, iv) + 1)..(System.Math.Max
(iu, iv) - 1)]
                               |> List.exists
                                   (fun i ->
                                     (List.exists ((==) road.points.[i])
vertexs))

               if not isAnother then
                  matrix.[row, col] <- 1
                  | _ -> ()

               matrix
               |> Array2D.iteri (fun col row i ->
                                   roads
                                   |> List.iter (fun road ->
                                                   if row <> col then
                                                     inMatrix row col road)
                                )

               matrix

               // get vertexs
               let vertexs = Road.Vertexs roads
               // get matrix
               let res = makeMatrix vertexs roads

               printfn "%A" res

  end
```

Теперь вызовем этот метод и посмотрим на полученный результат

```
[<EntryPoint>]
let main argv =
    printfn "Hello World from F#!"

    let rh1 =
        road "rh1" [
                          point (0., 7.);
                          point (1., 7.);
                          point (2., 7.);
                          point (3., 7.);
                          point (4., 7.);
                          point (5., 7.);
                          point (6., 7.);
                          point (7., 7.);
                          point (8., 7.);
                      ]
    // … make rh2-rh4

    let rv1 =
        road "rv1" [
                          point (3., 8.);
                          point (3., 7.);
                          point (3., 6.);
                          point (3., 5.);
                          point (3., 4.);
                          point (3., 3.);
                          point (3., 2.);
                          point (3., 1.);
                          point (3., 0.);
                      ]

    // … make rv2-rv3

    let roads = [rh1; rh2; rh3; rh4; rv1; rv2; rv3]
    let g = Graph.ofRoads roads

    0 // return an integer exit code
```

```
[[0; 1; 0; 1; 0; 0; 0; 0; 0; 0; 0; 0]
 [1; 0; 1; 0; 1; 0; 0; 0; 0; 0; 0; 0]
 [0; 1; 0; 0; 0; 1; 0; 0; 0; 0; 0; 0]
 [1; 0; 0; 0; 1; 0; 1; 0; 0; 0; 0; 0]
 [0; 1; 0; 1; 0; 1; 0; 1; 0; 0; 0; 0]
 [0; 0; 1; 0; 1; 0; 0; 0; 1; 0; 0; 0]
 [0; 0; 0; 1; 0; 0; 0; 1; 0; 1; 0; 0]
 [0; 0; 0; 0; 1; 0; 1; 0; 1; 0; 1; 0]
 [0; 0; 0; 0; 0; 1; 0; 1; 0; 0; 0; 1]
 [0; 0; 0; 0; 0; 0; 1; 0; 0; 0; 1; 0]
 [0; 0; 0; 0; 0; 0; 0; 1; 0; 1; 0; 1]
 [0; 0; 0; 0; 0; 0; 0; 0; 1; 0; 1; 0]]
```

На основе уже полученной матрицы смежностей можно получить граф в виде двух множеств, как того требует теория, но это уже не входит в задачу данной статьи. Читатель может самостоятельно попробовать решить эту задачу, так как автор надеется, что он разбудил у читателя интерес к функциональному программированию и к F# в частности.

Список литературы

“Программирование на F#”, автор Крис Смит
“Функциональное программирование на F#”, автор Сошников Д. В.
“Обзор F#”, https://docs.microsoft.com/ru-ru/dotnet/fsharp/tour