1 Давайте поговорим о функциях

1.1 Как объявлять функции

У хаскеля есть комилятор *ghc*. Он компилит вашу программу в бинарь
У нас етсь топ левел модуль. Там обычно есть функция main

```
1 module Main where
2 main = putStrLn "Hello, World!"
```

Вот эта штука и будет запускаться после компиляции файла. Порядок определений значения не имеет. Можно даже сначала сделать определение функции, а лишь потом объявление.

1.1.1 Объявление функций

Никаких лишних скобочек для аргументов функций нет. Например вот функция для прибавления к крестику единицы

```
1 f x = x + 1
```

а вот для сложения двух чисел

```
1 f x y = x + y
```

Функцию можно определить в 2 этапа

```
1 f :: Int -> Int
2 f x = x + 1
```

А можно и не определять, так как в хаскеле нормально все с type inference. В таких простых случаях он выводит типы сам. (это 2 примера выше)
Если нужно в мейне вывести больше, чем 1 действие, можно делать так:

```
1 main = do
2     print (g 2 4)
3     print (f 5)
```

1.1.2 Лямбда функции

Вообще говоря в хаскеле есть лямбдочки в таком виде

```
1 b = \x -> x + 1
```

И все вычисления строится именно на лямбдах. Любая функция в хаскеле будет преобразована в такую вот лямбду
В хаскеле есть инфиксные функции. Например

```
1 (+++) :: Int -> Int -> Int
2 x +++ y = x + y + x + y
```

1.1.3 Функция $

Есть интересный оператор $. У него объявление такое:

```
1 ($) :: (a -> b) -> a -> b
2 f $ x = f x
```

Т.е. это своего рода инфиксные приоритетопонижатель...
Зачем? У него самый низкий приоритет, а у функций (с именем) наоборот самый высокий приоритет.
Это позволяет эквивалентно переписывать конструкции... Например вот так:

```
1 main = do
2     print (g 2 4)
3     print $ g 2 4
```

Получается "возьми функцию слева и примени к ней все что получится из функции (функций) справа когда та досчитается

1.1.4 undefined вместо определения

Можно функцию объявить, но не определить.

```
1 funk :: Int -> Int
2 funk x = undefined
```

Оно скомпилится, будет тайп-чекаться везде, но при вызове выкинет жуку.

1.1.5 Ленивость в хаскеле

Хаскель ничего не делает, пока ему не нужно. По сему, вы можете определять функции, которые возвращают бесконечные списки... И даже найти их длину!

```
ones = 1 : ones
main = do
    print $ leng (take 100000 ones)
```

Эта штука работает, так как хаскель не пытается вытащить все единички из вызова *ones*, а берет только необходимое количество, чтобы выполнился *take*. К слову, *leng* тут тоже считает длину сразу же с получением нового элемента из *take*.

1.2 Стандартные конструкции языка

1.2.1 Комментарии

Однострочечные комментарии - это два минуса. Многострочечные - это фигурная скобка и минус.

```
-- funk :: Int -> Int one line comment

{- multiple line comment
funk x = undefined
 blalbla
-}
```

1.2.2 If'чики

```
sign x =
    if x < 0
    then -1
    else
        if x > 0
        then 1
        else 0
```

ифчики это хорошо, но чаще используют pattern mathing.

1.2.3 pattern mathing

Как пример: функция, которая возвращает строкое представление

```
data Foo = A | B

fromFoo :: Foo -> String
fromFoo A = "A"
fromFoo B = "B"
```

эта штука на самом деле тоже сахар. Она превращается в

```
data Foo = A | B

fromFoo' :: Foo -> String
fromFoo' x =
    case x of
      A -> "A"
      B -> "B"
```

На самом деле там есть и еще более упоротый синтаксис с фигурными скобками, но так редко пишут... разве что некоторые деды из haskell сообщества.
Паттерн матчинг умный и сам определит, что он объявлен избыточно или недостаточно. При чем во втором случае это может быть и варнинг, и ошибка, в зависимости от параметров компиляции.

```
goodNumber 1 = True
goodNumber 7 = True
goodNumber x
    | x < 100 = True
    | x >= 100 = False
```

Тут мы вообще начали объявлять функцию сразу через паттерн матчинг.

1.2.4 Паттер матчинг, рекурсия и работа со списками

Циклов в хаскеле нет. Но у нас же есть рекурсия! Вот так, например, можно найти длину списка

```
1 leng [] = 0
2 leng (_:xs) = 1 + leng xs
```

Лист - это на самом деле голова листа + его хвост. В данном случае хвост - это xs, а голова - это нечто, но мы его не используем, поэтому вместо головы мы пишем _.
Последний элемент можно взять, например, вот так:

```
1 lastElem [] = error "Oops!"
2 lastElem (x:[]) = x
3 lastElem (_:xs) = lastElem xs
```

Тут мы список проматчим на 3 значения:

1. Пустой список. Для него мы кидаем ошибку (к слову, вот так можно кинуть ошибку)
2. Список из головы, но без хвоста, т.е. один единственный элементм
3. Список из головы + хвоста. Здесь голову мы отбрасываемся и опускаемся дальше в хвост

Хаскель компилятор очень хорошо умеет оптимизировать и хвостовую рекурсию, и прочие хвостовые вызовы.
ДЗ: Реализовать функцию хода для задачи по ссылке: https://gist.github.com/astynax/1eb88e195c4bab2b8d31d04921b18dd0
Целевое решение можно посмотреть в папке с этим файлом move4x4.hs