Министерство науки и высшего образования Российской Федерации
федеральное государственное бюджетное образовательное учреждение
высшего образования
«Иркутский государственный университет»
(ФГБОУ ВО «ИГУ»)

Институт математики и информационных технологий
Кафедра информационных технологий

КУРСОВАЯ РАБОТА

На тему «Разработка приложения на языке функционального программирования»

Студент 3 курса очного отделения
группы 02321-ДБ
Бутаков Максим Павлович

Руководитель: К. т. н., доцент

_____ Черкашин Е. А.

Иркутск 2022г.

Оглавление

1. Введение

Функциональное программирование — парадигма программирования, в которой процесс вычисления трактуется как вычисление значений функций в их математическом понимании.

В функциональном программировании нет переменных, циклов – почти одни только функции. Но, несмотря на это, оно имеет ряд преимуществ, которые позволяют оставаться этой парадигме программирования актуальной и иметь множество приверженцев.

Целью моей работы является написание простой игры и изучение особенностей функциональной парадигмы программирования.

Цель определяет следующие **задачи**:

1. Изучение функционального подхода программирования на примере языка Haskell
2. Создание игры, подобной известной игре АйТи-95
    a. Создание архитектуры программы
    b. Реализация внутренней логики программы
    c. Реализация интерфейса для взаимодействия с пользователем

2. Теоретическая часть

Почти одновременно с первым языком программирования Fortran появился совершенно непохожий на него язык – Lisp. Для Lisp последовательность выполнения программ была несущественной. Развитие начатой им ветви программирования привело к образованию семейства своеобразных языков программирования, которые позволяли исполнять программу одновременно несколькими процессорами. Отдельные части этих программ могут выполняться независимо друг от друга. Это семейство языков сейчас и называется функциональным.

Программы написанные на функциональных языках программирования также обладают другими особенностями. Например, они зачастую пишутся гораздо короче своих аналогов на других языках.

Один из самых распространенных в настоящее время языков программирования – Haskell. Этот язык является чистым функциональным языком программирования, поддерживает ленивые вычисления, а также имеет множество модулей и библиотек.

Есть несколько библиотек для создания графических приложений. Из них я выбрал Gloss, потому что она самая простая.

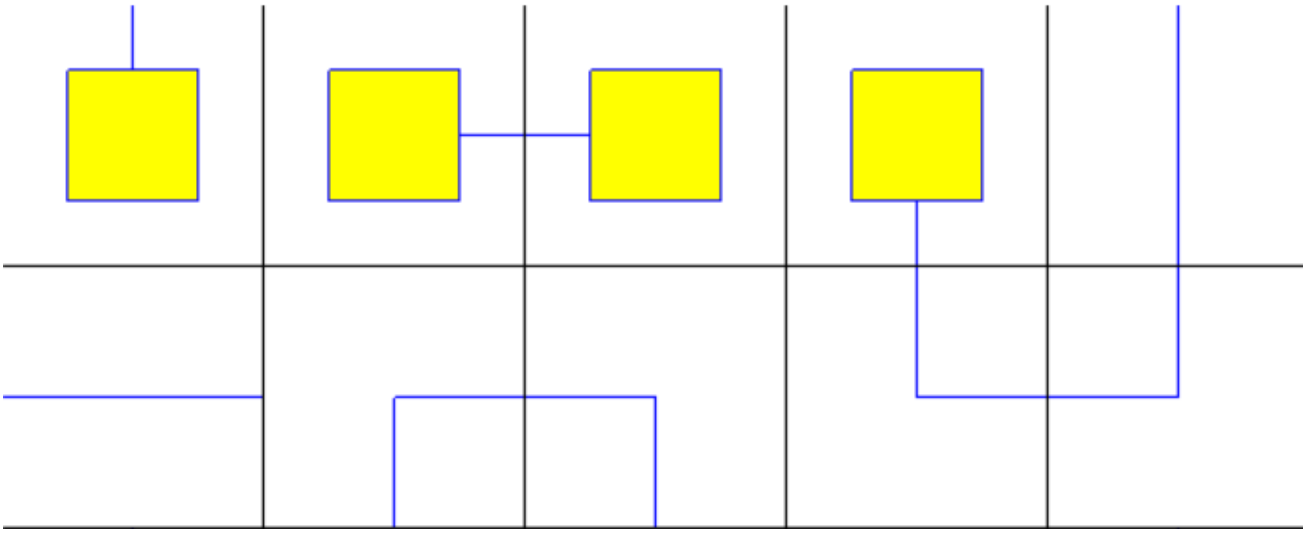
Конкретно в своей теме я использовал следующий ряд модулей:

1. `Graphics.Gloss` – модуль для работы с векторной 2D графикой, благодаря которой можно открывать окно заданного размера и рисовать в нём что хочешь
2. `Graphics.Gloss.Data.ViewPort` – модуль для создания глобального преобразования, применяемого к изображению
3. `Graphics.Gloss.Interface.Pure.Game` – модуль, позволяющая добавлять действие на нажатие большинства кнопок, доступных пользователю
4. `System.Random` – модуль для генерации псевдослучайных чисел

5. `System.IO` – модуль для работы с вводом/выводом

Правила моей игры, следующие:

- Есть поле, заполненное ячейками с блоками вида



- Цель игры – составить из всех блоков некоторое количество замкнутых цепей, то есть, чтобы при старте из любой клетки и проходу по синей линии, мы приходили в жёлтый квадрат.
- Взаимодействие осуществляется нажатием левой кнопки мыши на квадрат, после чего он вращается по часовой стрелке.

3. Практическая часть

Приступлю к описанию особенностей разработки и выполнения своей программы.

3.1. Архитектура

Своё приложение я делал, используя Cabal (расшифровывается как «общая архитектура сборки приложений и библиотек») – сборщик проекта, а так же компилятор ghc.

В файле CABAL можно указать особенности сборки, а также библиотеки которые необходимо подгрузить для работы проекта.

Запуск программы производится из файла Main.hs, точка входа – функция main :: IO ().

В своей программе я использовал некоторое количество функций для хранения информации о поле: его размеры, правильно решённое поле, а также размер одной ячейки поля.

По ходу программы происходят следующие действия:

- На основе правильно решенного поля создаётся случайно преобразованное поле;
- Запускается окно, с некоторой периодичностью выполняющее перерисовку поля;
- Обрабатываются нажатия мыши, при которых вызывается функция изменения текущего поля;
- После каждого нажатия проверяется, не находится ли наше поле в победном состоянии;
- В случае победы состояние поле меняется и выходит сообщение о том, что пользователь прошёл игру.

3.2. Реализация

Размеры поля возвращает функция `fieldSize`, которая также присваивает значения `fieldWidth` и `fieldHeight`.

Размер ячейки поля хранится в функции `cellSize`

Исходное поле представлено в виде одной строки, содержащей символы из набора ↑→←↓|—┌┐└┘, а также символ переноса строки. Она обёрнута в тип `type Field = String`. Так же в программе в комментариях указаны заготовки для других полей (для их работы соответственно нужно изменить размеры поля).

При запуске программы на основе генератора случайных чисел в функциях `createField` и `feelField` производится обработка исходного поля, в результате чего на выходе получаем наше начальное поле `Field`.

Тип данных `GameState` хранит информацию о текущем поле `Field` и состоянии игры `gameOver` – игра окончена или же продолжается.

Для изменения нашего поля служит функция `modification`, которая ищет нужную нам ячейку и, посредством вызова функции `change`, производит замену символа на данной позиции.

Для проверки текущего поля на выигрышное состояние имеется функция `checkField`. Она производит обход всего нашего поля и для каждой клеточки смотрит, существует ли замыкающая её путь соседняя клетка. Она вызывает функцию `get`, которая по заданному индексу возвращает хранящуюся там клетку, либо символ переноса строки в граничном случае.

3.3. Интерфейс

Прежде чем начать описывать интерфейс стоит упомянуть о несложных вспомогательных функциях: `both`, `screenToCell`, `cellToScreen`, `windowSize`. Функция `both` служит для применения функции к обоим элементам кортежа. Функция `screenToCell` выполняет преобразование координат мыши к ячейке нашего поля. Функция `cellToScreen` выполняет преобразование ячейки нашего поля к координатам квадрата в окне. Функция `windowSize` нужна для определения размеров нашего окна.

Для добавления интерфейса программы я использовал библиотек `Graphics.Gloss`. В ней запуск приложения происходит посредством запуска

функции play (она вызывается в моей функции `startGame`). Она принимает на вход 7 параметров: в чём запустить наше приложение, какого цвета будет фон приложения, сколько раз необходимо обновлять экран каждую секунду, наша структура для хранения состояния игры `world`, а также три функции – `renderer`, `handler`, `updater`.

Здесь `renderer` – это наша функция обновления экрана, она имеет вид (world -> Picture), то есть преобразует наше состояние поля в картинку на экране. У меня эта функция запускается путём сопоставления по образцу. Если игра окончена, то она просто выводит на пустом экране надпись `Victory!` Иначе она прорисовывает поле: она объединяет несколько картинок в одну: рисует сначала каждую клеточку и изображение на ней, а потом прорисовывает решётку поля. Состояние данной клетки получается путём вызова вышеописанной функции `get` с некоторыми преобразованиями. Функции `uncurry` и `translate` отвечают за перевод клеточек из начала координат в соответствующие им места на поле. Функция `applyViewPortToPicture` с параметром – функцией `viewPort` в аргументе выполняет преобразования рисования из центрованной системы координат к рисованию из левого нижнего угла.

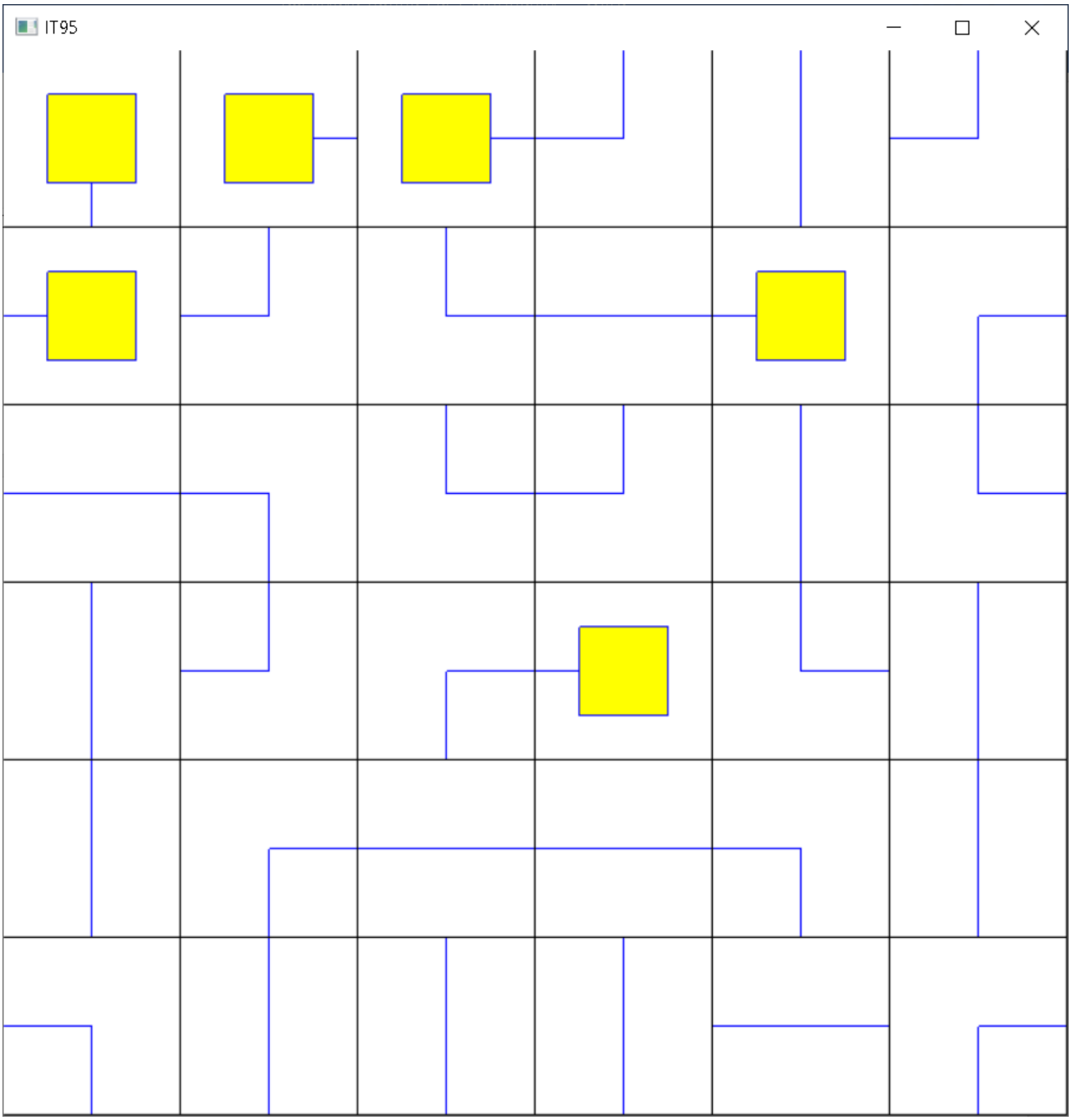
Функция `handler` – это функция для обработки нажатий мыши. При вызове от состояния неоконченной игры она получает соответствующую нажатую клетку и выполняет вызов функции `modification`. После этого осуществляется проверка того, не окончена ли игра. После этого состояние нашей игры обновляется. При вызове от оконченной игры она ничего не делает.

Функция `updater` нужна для обработки мира через определённые промежутки времени. В данной задаче она нам ни к чему, поэтому вставляем заглушку.

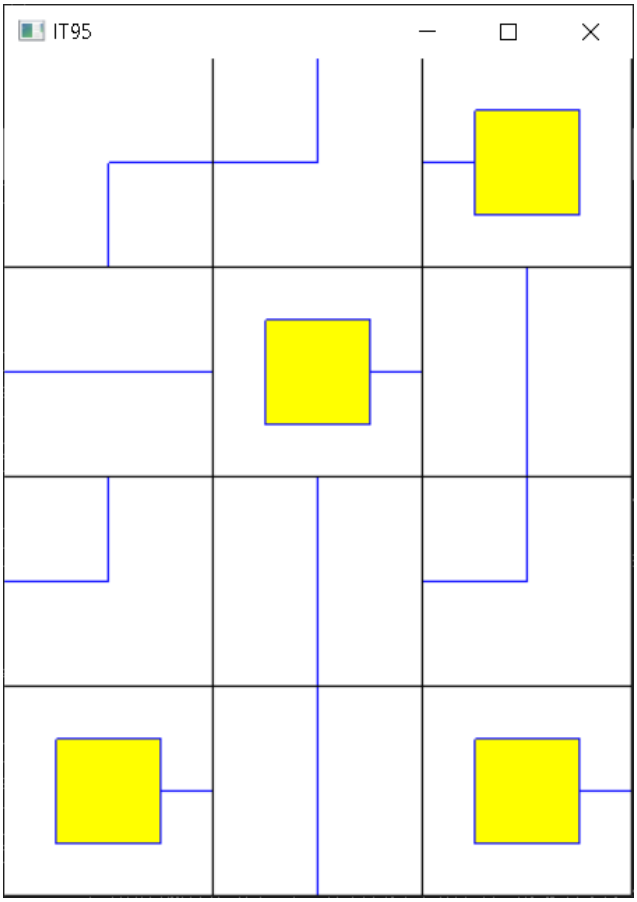
3.4. Демонстрация

Для демонстрации программы прикладываю скриншот игрового поля 6 на

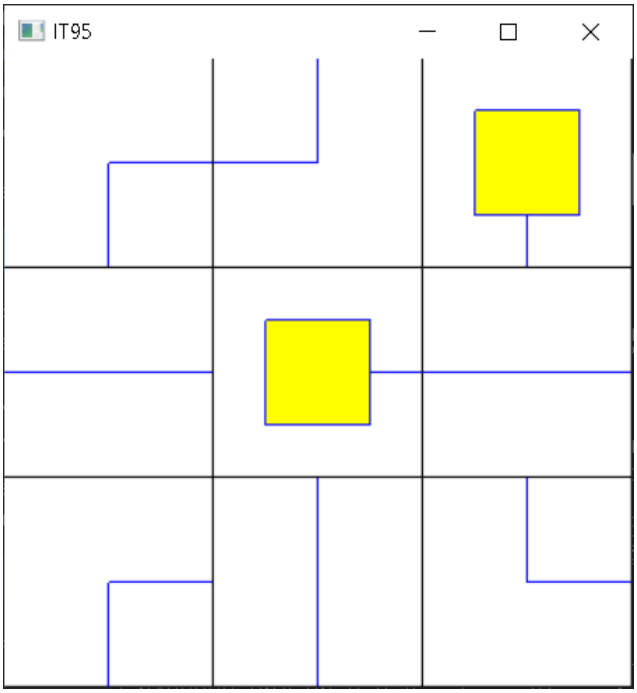
6:

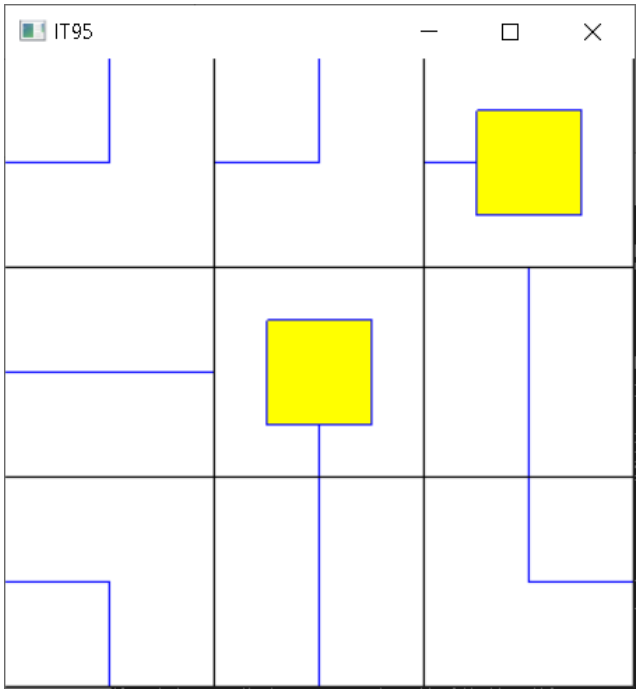


Также можно взять размер поля 3 на 4 (так же прикладываю скриншот поля, извещающего о победе):



Также можно убедиться, что при двух разных запусах поле генрируется по разному(пример 3 на 3):





4. Заключение

Изучение функционального языка в современных реалиях становится актуально, так как требуется производить параллельные вычисления. Это развивающееся направление программирования.

Опыт написания приложения на функциональном языке программирования лично для меня был очень интересен. Функциональное парадигма отличается от других и выглядит свежо.

Также мне удалось выполнить задачи, которые я себе поставил и я достиг цели работы, а именно написал простую игры и изучил особенности функциональной парадигмы программирования.

5. Список литературы

1. Кубенский, А. А. Функциональное программирование : учебник и практикум для вузов / А. А. Кубенский. — Москва : Издательство Юрайт, 2021. — 348 с. — (Высшее образование). — ISBN 978-5-9916-9242-7. — Текст : электронный // Образовательная платформа Юрайт [сайт]. — URL: https://urait.ru/bcode/469863 (дата обращения: 18.01.2022).
2. Сайт https://hackage.haskell.org
3. Сайт https://hoogle.haskell.org
4. Сайт https://ru.wikipedia.org

6. Приложение с исходниками

Файл CABAL

```
cabal-version:      2.4
name:               my-app-name
version:            0.1.0.0

-- A short (one-line) description of the package.
-- synopsis:

-- A longer description of the package.
-- description:

-- A URL where users can report bugs.
-- bug-reports:

-- The license under which the package is released.
-- license:

-- The package author(s).
-- author:

-- An email address to which users can send suggestions, bug reports,
and patches.
-- maintainer:

-- A copyright notice.
-- copyright:
-- category:
extra-source-files: CHANGELOG.md

executable my-app-name
    main-is:          Main.hs

    -- Modules included in this executable, other than Main.
    -- other-modules:
```

```
    -- LANGUAGE extensions used by modules in this package.
    -- other-extensions:
    build-depends:    base ^>=4.14.3.0
                    , random
                    , gloss
    hs-source-dirs:   app
    default-language: Haskell2010
```

Файл Main

```haskell
import Graphics.Gloss
import Graphics.Gloss.Data.ViewPort
import Graphics.Gloss.Interface.Pure.Game
import System.Random
import System.IO

main :: IO ()
main = do

    gen <- getStdGen
    startGame gen

fieldSize@(fieldWidth, fieldHeight) = (6, 6) :: (Int, Int)
cellSize = 120

change :: Char -> Char
change '↑' = '→'
change '→' = '↓'
change '↓' = '←'
change '←' = '↑'
change '—' = '|'
change '|' = '—'
change '┌' = '┐'
change '┐' = '┘'
change '┘' = 'L'
change 'L' = '┌'
change a = a

modification :: Field -> Int -> Field
modification [] _ = []
modification (x:xs) i | i == 0    = change x : xs
                      | otherwise = x : modification xs (i-1)

get :: Field -> Int -> Char
get [] _     = '\n'
get (x:xs) 0 = x
get (x:xs) i | i > (fieldHeight * (fieldWidth + 1) - 2) || i < 0 = '\n'
```

```
            | otherwise                       = Main.get xs (i - 1)

checkField :: Field -> Int -> Field  -> Bool
checkField [] _ _ = True
checkField (x:xs) i field | x == '↑' && Main.get field (i - fieldWidth - 1) `elem`
"┌↓┐|" = checkField xs (i+1) field
                          | x == '→' && Main.get field (i + 1) `elem` "┘←┐─" =
checkField xs (i+1) field
                          | x == '←' && Main.get field (i - 1) `elem` "┌→└─" =
checkField xs (i+1) field
                          | x == '↓' && Main.get field (i + fieldWidth + 1) `elem`
"└↑┘|" = checkField xs (i+1) field
                          | x == '|' && Main.get field (i - fieldWidth - 1) `elem`
"┌↓┐|" && Main.get field (i + fieldWidth + 1) `elem` "└↑┘|" = checkField xs (i+1)
field
                          | x == '─' && Main.get field (i + 1) `elem` "┘←┐─" &&
Main.get field (i - 1) `elem` "┌→└─" = checkField xs (i+1) field
                          | x == '┌' && Main.get field (i + 1) `elem` "┘←┐─" &&
Main.get field (i + fieldWidth + 1) `elem` "└↑┘|" = checkField xs (i+1) field
                          | x == '┐' && Main.get field (i - 1) `elem` "┌→└─" &&
Main.get field (i + fieldWidth + 1) `elem` "└↑┘|" = checkField xs (i+1) field
                          | x == 'L' && Main.get field (i + 1) `elem` "┘←┐─" &&
Main.get field (i - fieldWidth - 1) `elem` "┌↓┐|" = checkField xs (i+1) field
                          | x == '┘' && Main.get field (i - 1) `elem` "┌→└─" &&
Main.get field (i - fieldWidth - 1) `elem` "┌↓┐|" = checkField xs (i+1) field
                          | x == '\n' = checkField xs (i+1) field
                          | otherwise = False

type Field = String

 --все символы "↑→←↓|─┌┐└┘"
 --поле 3 на 3 "┌┐↓\n|↑|\n└─┘"
 --поле 3 на 4 "┌┐↓\n|↑|\n└─┘\n→─←"
 --поле 4 на 4 "┌┐↓↓\n|↑||\n└─┘|\n→──┘"
 --поле 5 на 5 "→───┐\n┌─←┌┘\n└─←└┐┌──┘\n└───←"
 --поле 6 на 6 "→←↓┌─┐\n↓┌┘|→┘\n|└┐└─┐\n|┌┘→┐|\n|└──┘|\n└────┘"

createField :: StdGen -> Field
createField gen = feelField gen "→←↓┌─┐\n↓┌┘|→┘\n|└┐└─┐\n|┌┘→┐|\n|└──┘|\n└────┘"

feelField :: StdGen -> Field -> Field
feelField gen [] = []
feelField gen ('\n': xs) = '\n' : feelField gen xs
feelField gen (x:xs) = case fst getR of
    0 -> x : feelField (snd getR) xs
    1 -> change x : feelField (snd getR) xs
    2 -> change (change x) : feelField (snd getR) xs
    3 -> change (change (change x)) : feelField (snd getR) xs
```

```
 where getR = randomR (0 :: Integer, 3) gen

data GameState = GS
    { field    :: Field
    , gameOver :: Bool
    }

initState gen = GS (createField gen) False

startGame :: StdGen -> IO ()
startGame gen = play (InWindow "IT95" Main.windowSize (100, 100)) white 30
(initState gen) renderer handler updater

windowSize = both (* (round cellSize)) fieldSize

both :: (a -> b) -> (a, a) -> (b, b)
both f (a, b) = (f a, f b)

updater _ = id

handler (EventKey (MouseButton LeftButton) Down _ mouse) gs@GS
    { field = field
    , gameOver = False
    } = gs
    { field = newField
    , gameOver = newState
    } where
    newField = click cell field
    cell@(cx, cy) = screenToCell mouse
    click :: (Int, Int) -> Field -> Field
    click c@(cx, cy) f = modification field ((fieldHeight - 1 - cy) * (fieldWidth +
1) + cx)
    newState = checkField newField 0 newField

handler _ gs = gs

renderer GS {gameOver = True} = translate (-cellSize * fromIntegral fieldWidth / 6)
0 .
                                scale (cellSize * fromIntegral fieldWidth / 1000)
((cellSize * fromIntegral fieldHeight) / 1000)
                                . color red $ text "Victory!"

renderer GS { field = field, gameOver = False } = applyViewPortToPicture viewPort $
pictures $ cells ++ grid where
    grid = [uncurry translate (cellToScreen (x, y)) $ color black $ rectangleWire
cellSize cellSize | x <- [0 .. fieldWidth - 1], y <- [0 .. fieldHeight - 1]]
    cells = [uncurry translate (cellToScreen (x, y)) $ drawCell x y | x <- [0 ..
fieldWidth - 1], y <- [0 .. fieldHeight - 1]]
```

```
    drawCell x y = case get field ((fieldHeight - 1 - y) * (fieldWidth + 1) + x) of
        '|'       -> pictures [ color white $ rectangleSolid cellSize cellSize
                                , color blue $ line [(0, -cellSize/2), (0,
cellSize/2)]
                                ]
        '—'       -> pictures [ color white $ rectangleSolid cellSize cellSize
                                , color blue $ line [(-cellSize/2, 0),
(cellSize/2, 0)]
                                ]
        'Γ'       -> pictures [ color white $ rectangleSolid cellSize cellSize
                                , color blue $ line [(cellSize/2, 0), (0,0), (0,
-cellSize/2)]
                                ]
        '7'       -> pictures [ color white $ rectangleSolid cellSize cellSize
                                , color blue $ line [(-cellSize/2, 0), (0,0),
(0, -cellSize/2)]
                                ]
        'L'       -> pictures [ color white $ rectangleSolid cellSize cellSize
                                , color blue $ line [(cellSize/2, 0), (0,0), (0,
cellSize/2)]
                                ]
        'J'       -> pictures [ color white $ rectangleSolid cellSize cellSize
                                , color blue $ line [(-cellSize/2, 0), (0,0),
(0, cellSize/2)]
                                ]
        '↑'       -> pictures [ color white $ rectangleSolid cellSize cellSize
                                , color blue $ line [(0, cellSize/2), (0,
cellSize/4)]
                                , color yellow $ rectangleSolid (cellSize/2)
(cellSize/2)
                                , color blue $ rectangleWire (cellSize/2)
(cellSize/2)
                                ]
        '→'       -> pictures [ color white $ rectangleSolid cellSize cellSize
                                , color blue $ line [(cellSize/2, 0),
(cellSize/4, 0)]
                                , color yellow $ rectangleSolid (cellSize/2)
(cellSize/2)
                                , color blue $ rectangleWire (cellSize/2)
(cellSize/2)
                                ]
        '←'       -> pictures [ color white $ rectangleSolid cellSize cellSize
                                , color blue $ line [(-cellSize/2, 0), (-
cellSize/4, 0)]
                                , color yellow $ rectangleSolid (cellSize/2)
(cellSize/2)
                                , color blue $ rectangleWire (cellSize/2)
(cellSize/2)
```

```
                                    ]
        '↓'       -> pictures [ color white $ rectangleSolid cellSize cellSize
                                    , color blue $ line [(0, -cellSize/2), (0, -
cellSize/4)]
                                    , color yellow $ rectangleSolid (cellSize/2)
(cellSize/2)
                                    , color blue $ rectangleWire (cellSize/2)
(cellSize/2)
                                    ]

viewPort = ViewPort (both (negate . (/ 2) . (subtract cellSize)) $ cellToScreen
fieldSize) 0 1

screenToCell = both (round . (/ cellSize)) . invertViewPort viewPort

cellToScreen = both ((* cellSize) . fromIntegral)
```