Текстова приключенска игра

проект към курс “Функционално програмиране”

2022 - 2023г.

—

Стефания Цветкова
Софтуерно инженерство
фак. №: 62573

Описание

Проектът представлява текстова приключенска игра, която се играе от 1 играч, който взаимодейства с играта чрез въвеждане на текстови команди и получава информация какво се случва на база неговите ходове чрез текстови съобщения.

Играта се развива на терен, на който има както опасности и препятствия, така и бонуси и рицари, с които играчът може да се бие. Ако играчът изгуби всичкия си живот или сила, умира. Целта на играта е играчът да остане жив максимално дълго.

Играчът притежава живот и сила, които се променят в хода на играта на база преживяванията на играча. Играчът може да се движи из игрова карта, като движението се осъществява във всяка посока на света: север, юг, изток и запад. На игровата карта се намират следните неща:

- Празна клетка, през която играчът може да премине свободно
- Клетка с бонус “размяна” - когато играчът посети тази клетка, той придобива възможността по време на битка да направи размяна на здраве за сила, ако пожелае. Този бонус е вечен, т.е. може да бъде използван неограничен брой пъти по време на играта.
- Клетка с бонус живот - когато играчът посети тази клетка, той получава 1 бонус живот.
- Клетка с бонус “бойно подкрепление” - когато играчът посети тази клетка, той получава 1 бонус “бойно подкрепление”. Той може да бъде използван еднократно в битка, като помага на играча да я спечели, независимо колко живот и сила има.
- Клетка “капан” - когато играчът попадне в тази клетка, той губи 1 живот.
- “Бойна” клетка - когато играчът посети тази клетка, той бива изправен пред рицар, който го предизвиква на дуел. Влизането в бой няма как да бъде пропуснато, като като преди началото е известна силата на рицаря. Играчът може да използва “бойно подкрепление” или “размяна”, ако притежава тези бонуси. Битката бива спечелена от играча, ако той има по-голяма сила от рицаря. При победа животът и силата на играча се увеличават със съответните живот и сила на противника му, а при загуба се намаляват.
- Непроходима клетка - това е клетка, в която играчът не може да отиде. Когато наррави опит да я посети, той получава грешка и не сменя местоположението си.

Движейки се из игровата карта, играчът посещава различни от описаните по-горе клетки, като след посещение на клетка, тя става празна.

При стартиране на играта по произволен начин се избира 1 от 2 игрални карти, като по този начин потребителят по-трудно би могъл да наизусти картата, което би обезсмислило играта.

Ако потребителят попадне в ситуация, когато не знае какви команди може да използва, той може да поиска помощ и/или да прегледа историята на събитията до момента. Ако прецени, може да спре играта и преди да е умрял.

Технологии

Проектът е изработен изцяло на Haskell, като са използвани следните помощни функции (които са част от публични библиотеки):

- **when** от **Control.Monad**
- **randomIO** от **System.Random**
- **unsafePerformIO** от **System.IO.Unsafe**

Стартиране на проекта

Проектът може да бъде стартиран чрез Glasgow Haskell Compiler чрез следните команди:

```
stack ghci --package random
:l main.hs
main
```

Командите трябва да бъдат изпълнени в терминал, в който предварително е отворена директорията, в която се намира сорс кода на проекта.

Имплементация

Разпределение на кода

Кодът е разпределен в отделни файлове, които са разпределени в отделни папки, което помага по-лесно и бързо човек да се ориентира къде може да открие дадена функция, както и да разбере какво се съдържа в даден файл само по неговото име и място в йерархията на файловете.

Файловете на проекта са разпределени в 5 директории - **constants**, **data**, **helpers**, **input** и **output**, като файлът **main.hs** не е част от никоя директория:

```
> constants
      commands.hs
      initialValues.hs
      map.hs
      stats.hs
> data
      coordinates.hs
      player.hs
      Stats.hs
> helpers
      endOfGame.hs
      fight.hs
      map.hs
      mapMovement.hs
> input
      command.hs
      name.hs
> output
      endOfGame.hs
      errors.hs
      fight.hs
      help.hs
      inputExplanation.hs
      introduction.hs
      moves.hs
      Player.hs
main.hs
```

В директорията **constants** се намират файлове, съдържащи единствено и само константи (т.е. функции, които нямат параметри и винаги имат един и същи резултат).

В директорията **data** се намират файлове, съдържащи нови типове данни и помощни методи към тях.

В директорията **helpers** се намират файлове, съдържащи помощни функции с логиката на играта.

В директорията **input** се намират файлове, съдържащи функции, които вход от потребителя.

В директорията **output** се намират файлове, съдържащи функции, които печатат съобщения на конзолата.

Файлът **main.hs** е малко по-специален, затова не се намира в никоя директория - той съдържа само функцията **main**, от която започва играта.

Начало на играта

Играта започва с извикването на функцията **main**:

```
main :: IO ()
main = do
    name <- getName

    let gameMap =
            if unsafePerformIO randomIO then
                map1
            else
                map2

    let playerPosition = getPlayerPosition gameMap

    let player = Player name initialStats playerPosition initialCanExchange
initialFightReinforcementsCount []

    printIntroduction name

    movePlayer player gameMap
```

В нея първо се взима името на потребителя чрез помощната функция **getName**, след което се избира коя от двете игрови карти да се използва за текущата игра, печата се въведение в играта и се пристъпва към същинската част на играта, чрез извикване на функцията **movePlayer**.

Виждаме, че за намиране на произволна булева стойност функцията използва **randomIO** - функция, която връща произволна стойност от желан от нас тип - в случая **Bool**, тъй като **if-statement**-а очаква стойност от тип **Bool**. Функцията **unsafePerformIO** е използвана, за да можем да разглеждаме резултата на **randomIO** от тип **IO Bool** все едно е от тип **Bool**.

Комуникация с потребителя

Комуникацията с потребителя се осъществява чрез конзолата, като се печатат текстови съобщения и при нужда от вход от потребителя се прочита ред от конзолата, който се обработва.

Функциите, отговарящи за "изхода" на програмата (т.е. тези, които печатат на конзолата) се намират в директорията **output**. Една примерна такава функция е **printWaitingForName**, която индикира на потребителя, че от него се очаква да въведе своето име:

```
printWaitingForName :: IO ()
printWaitingForName =
    putStrLn "What is your name?"
```

Функциите, отговарящи за "входа" на програмата (т.е. тези, които четат от конзолата) се намират в директорията **input**. Една примерна такава функция е **getName**, която отговаря за вземането на потребителското име на потребителя (и неговата валидация). Тази функция използва функцията **printWaitingForName** от споменатите в горния параграф функции за изход, за може да бъде по-ясно за потребителя какво точно се очаква от него.

```
getName :: IO String
getName = do
    printWaitingForName
    name <- getLine
    putStrLn ""

    if name == "" then do
        printEmptyNameError
        getName
    else
        return name
```

Движение из игровата карта

Играчът може да се движи из игровата карта в 4те посоки на света чрез въвеждане на текстова команда. Логиката за това движение се намира във файла **mapMovement.hs**, като започва от функцията **movePlayer**:

```
movePlayer :: Player -> [String] -> IO ()
movePlayer player gameMap =
    let
        updatedMap = removeElement gameMap (position player)
        correctInputs =
            [
                moveLeftCommand,
                moveUpCommand,
                moveRightCommand,
                moveDownCommand,
                helpCommand,
                exitCommand,
                reviewGameCommand
            ]
    in do
        if isAlive player then do
            printPlayerStats player

            input <- getCommand

            if input `notElem` correctInputs then do
                printIncorrectMovementInput
                movePlayer player updatedMap
            else
                handleInput input player updatedMap
        else
            handleEndOfGame player True updatedMap
```

Първото нещо, което се случва във функцията, е да се “премахне” стойността на последната посетена клетка от картата - т.е. да се създаде нов списък от **String**-ове, който е аналогичен на първия с единствената разлика, че на позицията на играча (съответните ред и стълб от разглежданата матрица) има задължително празен символ. Това се случва в помощната функция **removeElement**, като получената матрица се именува **updatedMap**.

Всички възможни команди за хода за движение са записани в списък от **String**-ове и са именувани **correctInputs**.

Има две възможни ситуации, в които можем да се намираме - играчът да е “жив” или да не е. Ако е, то на конзолата се извежда неговата “статистика” (наличен към момента живот и сила), след което той съсежда команда. Ако въведената команда е грешна, то се отпечатва съобщение за грешка и отново се извиква функцията movePlayer, а в противен случай въведената команда се обработва. Ако играчът не е “жив”, играта приключва с извикване на функцията за край на играта.

Входът се обработва от функцията **handleInput**:

```
handleInput :: String -> Player -> [String] -> IO ()
handleInput input player gameMap = do
    case () of
        ()
            | input == helpCommand -> do
                printMovementHelp
                movePlayer player gameMap
            | input == exitCommand ->
                handleEndOfGame player False gameMap
            | input == reviewGameCommand -> do
                reviewGame (history player)
                movePlayer player gameMap
            | otherwise -> do
                handleMapMovement input player gameMap
```

Ако той е команда за помощ, или печатане на история на събитията до момента, нужната информация се печата и се преминава към правене на нов ход. Ако входът е за край на играта, то се извиква функцията за край на играта. В останалите случаи се преминава към обработка на команда за движение из игровата карта.

Самото движение из картата се дирижира от функцията **handleMapMovement**:

```
handleMapMovement :: String -> Player -> [String] -> IO ()
handleMapMovement input player gameMap =
    let
        cellCoordinates = getMovementCoordinates input (position player)
    in
        if isPositionInMap cellCoordinates gameMap then
            goToCell player gameMap cellCoordinates
        else do
            printOutsideOfMapMove
            movePlayer player gameMap
```

В нея с помощта на функцията **getMovementCoordinates** се намират координатите на клетката, в която играчът трябва да се премести (съответните координати са +-1 на реда или +-1 на колоната на текущите координати на потребителя в зависимост от посоката на движение). След намиране на координатите на клетката се пристъпва към “отиване” в клетката чрез функцията **goToCell**. В нея се намира каква е стойността на клетката, в която отива играчът, също както и се създава нов играч, който има същите параметри като досегашния, но и добавен текущия ход към историята му на ходове. В зависимост от това в каква клетка е попаднал играча се преминава към помощна функция, която обработва събитието е тази клетка (тези функции са разгледани но-долу в настоящата документация), след което преминава

отново към следващия ход на играча. Ако пък случайно играчът се опита да посети клетка, в която не може да се отиде, той получава съобщение за този проблем и има възможност да играе нов ход.

Празен ход

Функцията **playerInEmptyCell** обработва случая, когато играчът попадне в празна клетка:

```
playerInEmptyCell :: Player -> [String] -> IO ()
playerInEmptyCell player gameMap = do
    printEmptyMove
    movePlayer player gameMap
```

Тогава се печата информиращо съобщение, че не се е случило нищо, и играта продължава с досегашните параметри на играча.

Бонус “размяна”

Функцията **playerInExchangeCell** обработва случая, когато играчът попадне в клетка с бонус “размяна”:

```
playerInExchangeCell :: Player -> [String] -> IO ()
playerInExchangeCell player gameMap = do
    printPlayerEarnedExchange
    movePlayer (addExchange player) gameMap
```

Тогава се печата информиращо съобщение, че играчът е спечелил бонус “размяна”, и играта продължава, като параметрите на играча се “променят”, т.е. функцията **movePlayer** се извиква с нов играч, който има опция да прави размени (параметърът **canExchange** има стойност True), а всички останали параметри са същите каквито са били до момента. Новият играч се създава чрез функцията **addExchange**:

```
addExchange :: Player -> Player
addExchange player =
    Player
        (name player)
        (stats player)
        (position player)
        True
        (fightReinforcementsCount player)
        (history player)
```

Бонус "бойно подкрепление"

Функцията **playerInFightReinforcementCell** обработва случая, когато играчът попадне в клетка с бонус "бойно подкрепление":

```
playerInFightReinforcementCell :: Player -> [String] -> IO ()
playerInFightReinforcementCell player gameMap = do
    printPlayerEarnedFightReinforcement
    movePlayer (addFightReinforcement player) gameMap
```

Тогава се печата информиращо съобщение, че играчът е спечелил бонус "бойно подкрепление", и играта продължава, като параметрите на играча се "променят", т.е. функцията **movePlayer** се извиква с нов играч, чийто параметър fightReinforcementsCount има стойност с 1 по-голяма от тази, която е била до момента, а всички останали параметри са същите каквито са били до момента. Новият играч се създава чрез функцията **addFightReinforcement**, която използва помощната функция **updateFightReinforcements**, която увеличава fightReinforcementsCount с подадено като параметър число:

```
addFightReinforcement :: Player -> Player
addFightReinforcement player =
    updateFightReinforcements player 1
```

```
updateFightReinforcements :: Player -> Int -> Player
updateFightReinforcements player fightReinforcementsToAdd =
    Player
        (name player)
        (stats player)
        (position player)
        (canExchange player)
        (playerFightReinforcementsCount + fightReinforcementsToAdd)
        (history player)
        where
            playerFightReinforcementsCount = fightReinforcementsCount player
```

Бонус живот

Функцията **playerInBonusLifeCell** обработва случая, когато играчът попадне в клетка с бонус живот:

```
playerInBonusLifeCell :: Player -> [String] -> IO ()
playerInBonusLifeCell player gameMap = do
    printPlayerEarnedBonusLife
    movePlayer (addHealth player 1) gameMap
```

Тогава се печата информиращо съобщение, че играчът е спечелил бонус живот, и играта продължава, като параметрите на играча се "променят", т.е. функцията **movePlayer** се извиква с нов играч, който има 1 живот повече от до момента, а всички останали параметри са същите каквито са били до момента. Новият играч се създава чрез функцията **addHealth**, като за брой животи се подава 1. Функцията **addHealth** използва помощната функция **addStats**, която добавя "статистиките", подадени като параметър, към тези на играча, а останалите му параметри оставя непроменени.

```
addHealth :: Player -> Int -> Player
addHealth player healthToAdd =
    addStats player (Stats healthToAdd 0)
```

```
addStats :: Player -> Stats -> Player
addStats player statsToAdd =
    Player
        (name player)
        (Stats newHealth newStamina)
        (position player)
        (canExchange player)
        (fightReinforcementsCount player)
        (history player)
        where
            newHealth = health (stats player) + health statsToAdd
            newStamina = stamina (stats player) + stamina statsToAdd
```

Капан

Функцията **playerInTrapCell** обработва случая, когато играчът попадне в клетка капан:

```
playerInTrapCell :: Player -> [String] -> IO ()
playerInTrapCell player gameMap = do
    printTrapMove
    movePlayer (removeHealth player 1) gameMap
```

Тогава се печата информиращо съобщение, че играчът е попаднал в клетка капан, от което губи 1 живот, и играта продължава, като параметрите на играча се "променят", т.е. функцията **movePlayer** се извиква с нов играч, който има 1 живот по-малко от до момента, а всички останали параметри са същите каквито са били до момента. Новият играч се създава чрез функцията **removeHealth**, като за брой животи се подава 1. Функцията **removeHealth** използва помощната функция **addStats** (описана

в предхосната точка), като броя животи за добавяне в addStats е равно на отрицателната версия на броя животи за премахване.

```
removeHealth :: Player -> Int -> Player
removeHealth player healthToRemove =
    addHealth player (negate healthToRemove)
```

Непроходима клетка

Функцията **playerOutsideOfMap** обработва случая, когато играчът попадне в непроходина клетка:

```
playerOutsideOfMap :: Player -> [String] -> IO ()
playerOutsideOfMap player gameMap = do
    printOutsideOfMapMove
    movePlayer player gameMap
```

Тогава се печата информиращо съобщение, че не може да се отиде в желаната клетка, и играта продължава с досегашните параметри на играча.

Битка

Функцията **playerInFightCell** обработва случая, когато играчът попадне в бойна клетка:

```
playerInFightCell :: Player -> [String] -> IO ()
playerInFightCell player gameMap = do
    updatedPlayer <- startFight player
    movePlayer updatedPlayer gameMap
```

Тогава играта продължава с играча, който е върнат от функцията **startFight**:

```
startFight :: Player -> IO Player
startFight player =
    let
        playerHasReinforcement = fightReinforcementsCount player > 0
        playerCanExchange = canExchange player
    in do
        printFight playerHasReinforcement playerCanExchange

        fight player
```

Функцията **startFight** проверява дали играчът може да използва размяна и бойно подкрепление, извиква помощната функция **printFight**, която печата на конзолата информация относно предстоящата битка, противника и възможните команди, които потребителя може да използва в зависимост от това какви бонуси притежава. След това се извиква функцията **fight**, която обработва самата битка:

```
fight :: Player -> IO Player
fight player =
    let
        playerHasReinforcement = fightReinforcementsCount player > 0
        playerCanExchange = canExchange player
    in do
        input <- getCommand

        case () of
            ()
                | input == fightCommand ->
                    handleFight player
                | playerHasReinforcement && input == fightReinforcementCommand
->
                    handleFightReinforcement player
                | playerCanExchange && (input == exchangeCommand) ->
                    handleExchange player
                | otherwise -> do
                    printIncorrectFightInput playerHasReinforcement
playerCanExchange
                    fight player
```

Функцията **fight** взима вход от потребителя и го обработва, като ако е коректна команда, за която играчът има права (т.е. Команда за използване на бонус размяна би била некоректна, ако играчът не е спечелил този бонус, но пък коректна, ако го е). Ако командата е коректна, то има 3 възможни случая - командата е за директно влизане в битка, за използване на бонус размяна или за използване на бонус бойно подкрепление.

Ако командата е за влизане в битка, то се извиква функцията **handleFight**:

```
handleFight :: Player -> IO Player
handleFight player =
    let
        playerStamina = stamina (stats player)
        opponentStamina = stamina fightWeight
    in do
        printStartFight

        if playerStamina > opponentStamina then do
            printWonFight
            return (addStats player fightWeight)
        else do
            printLostFight
            return (removeStats player fightWeight)
```

В нея се извършва същинската битка, като първоначално на конзолата се печата информация относно стартиращата битка. Битката има 2 възможни изхода - победа за играча, когато силата ми е по-голяма от силата на противника му, и загиба - в противен случай. Когато изграчът победи, той печели живота и силата от битката, а в противен случай ги губи. “Промяната” на статистиките на играча се извършва чрез функциите **addStats** и **removeStats**, които споменахме по-рано. Функцията връща самият играч с нови характеристики

Ако командата е за за използване на бойна помощ, то се извиква функцията **handleFightReinforcement**:

```
handleFightReinforcement :: Player -> IO Player
handleFightReinforcement player = do
    printWonFight
    return (removeFightReinforcement (addStats player fightWeight))
```

В нея първо се печата съобщение на екрана, че потребителят е спечелил битката, След което, аналогично на **handleFight** метода, **handleFightReinforcement** връща играч, които има 1 по-малко **playerFightReinforcementsCount** и добавя спечелените здраве и сила (тежестта на битката).

Ако командата е за за използване на бонус размяна, то се извиква функцията **handleExchange**:

```
handleExchange :: Player -> IO Player
handleExchange player =
    let
        playerHealth = health (stats player)
    in do
        if playerHealth > health exchangeRate then do
            printExchange
            handleFight (exchangeStats player exchangeRate)
        else do
            printErrorExchange
            handleFight player
```

В нея ако играчът има достатъчно здраве, за да направи исканата размяна, здраве се конвертира в сила по определения курс и се започва битка, а в противен случай се печата съобщение за грешка и битката започва с непроменени живот и сила на играча. Функцията **exchangeStats** променя статистиката на играча, като намалява здравето и увеличава силата му:

```
exchangeStats :: Player -> Stats -> Player
exchangeStats player statsToExchange =
    Player
        (name player)
        (Stats newHealth newStamina)
        (position player)
        (canExchange player)
        (fightReinforcementsCount player)
        (history player)
        where
            newHealth = health (stats player) - health statsToExchange
            newStamina = stamina (stats player) + stamina statsToExchange
```

История на събитията

Историята на събитията се печата чрез функцията **reviewGame**, която печата на конзолата всички действия, които са извършени от потребителя, като за целта се използва функцията **printHistory**, която печата всеки елемент от подадения списък със String-ове на отделен ред. Историята на събитията на играча се подава в обратен ред на **printHistory**, така че на конзолата да излезнат хронологично

```
reviewGame :: [String] -> IO ()
reviewGame history = do
    putStrLn "Here is this game's timeline:"
    printHistory (reverse history)
    putStrLn ""
```

Край на играта

Играта може да приключи по 2 начина - когато играчът е умрял и когато е жив, но е пожелал да прекрати играта. Когато се стигне до този момент се печата съобщение (различно в зависимост от причината за край на играта), след което се взима последен вход от потребителя. Ако той е команда да преглеждане на историята на събитията, то тя се печата на конзолата. Печата се съобщение за край на играта и програмата приключва. Това се извършва от функцията **handleEndOfGame** :

```
handleEndOfGame :: Player -> Bool -> [String] -> IO ()
handleEndOfGame player isDead gameMap = do
    if isDead then
        printPlayerDied
    else
        printExitGame

    input <- getLine
    when (input == reviewGameCommand) (reviewGame (history player))

    printGoodbye
```

Възможности за развитие

Текущият проект представя първа базова версия на текстовата игра, като в бъдеще тя може да бъде подобрена по много начини, в зависимост от нуждите на играчите и проблемите, които биха били открити. Добро подобрение би било в картата да се появяват нови елементи, така че тя да се променя динамично с течение на хода на играта, също както и наличните карти да бъдат разширени както по размер, така и н брой. Друго добро подобрение би било да се измисли концепция за “победа” на играча, тъй като текущата версия на играта няма такава.