Project: Yatzy

Program Design and Data Structures (1DL201) 2020/2021

Group 29: Judit Hohenthal, Erdem Garip, Kasper Gillberg Kling

March 4 2021

Table of Contents

1. Introduction

The PKD-course gave a task to a group of three students to create a project of their choosing and we discussed some ideas briefly within our group and decided to implement a Yahtzee game in Haskell. Our initial idea was to divide the project into two components, a Yahtzee game and a graphical user interface. The first component, namely a Yahtzee game, could successfully be implemented but it was decided to skip the GUI-part due to lack of time and lack of knowledge of relevant Haskell libraries.

1.2 Summary

The program lets the user play a game of Yahtzee on their computer. The user will get to specify how many players will be participating and have the option to play against AI players. A game ledger will be drawn up for each player of the game.

The program uses a random number generator to simulate rolling dice. Users will get to decide how to reroll their dice in order to get the best possible result. They will then be able to choose what slot in their ledger to write in their dice in. The program will check the legality of each player's move, which prohibits any cheating from occurring.

The game is played in- and printed to the terminal. All user interaction with the game therefore takes place at the command line.

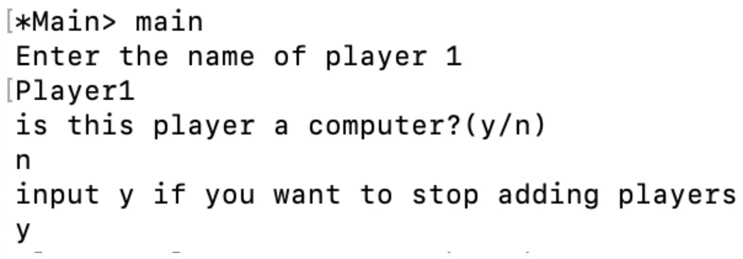
2. User Cases

2.1 Software Requirements

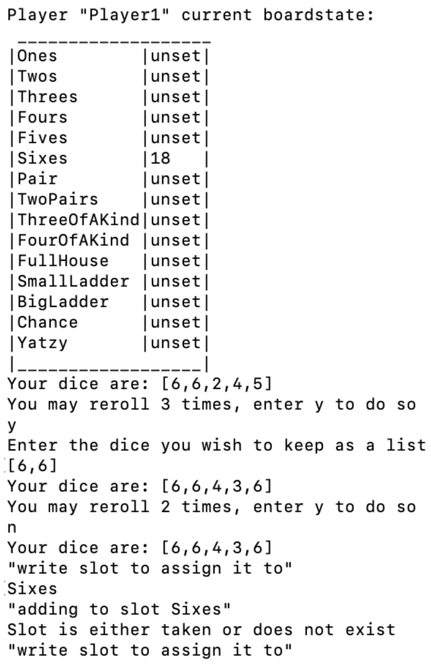
In order to use the program the user has to have a GHC-compiler for Haskell installed on their computer. The user also has to install the System.Random library and hUnit in order to get the program to work. These can be installed using the cabal system. The program comes with cabal files that makes it possible to run the game.

2.2 User Example

To play the game the user will have to navigate to the repository where the project files are stored, type in ‘ghci’ and then ‘TerminalGui.hs’ into their terminal. To start the game the user then writes in ‘main’ in the terminal. Or alternatively use cabal run to build and/or run the executable file. The program will ask you to write in players' names, and if the player is a computer or not. When all the desired players have been added the user should press the 'y' key and the game will begin.



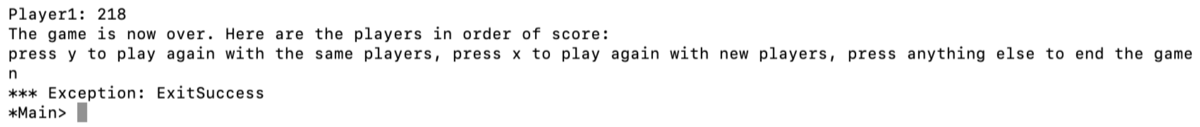
```
[*Main> main
 Enter the name of player 1
[Player1
 is this player a computer?(y/n)
 n
 input y if you want to stop adding players
 y
```



```
Player "Player1" current boardstate:
 __________________
|Ones        |unset|
|Twos        |unset|
|Threes      |unset|
|Fours       |unset|
|Fives       |unset|
|Sixes       |18   |
|Pair        |unset|
|TwoPairs    |unset|
|ThreeOfAKind|unset|
|FourOfAKind |unset|
|FullHouse   |unset|
|SmallLadder |unset|
|BigLadder   |unset|
|Chance      |unset|
|Yatzy       |unset|
|__________________|
Your dice are: [6,6,2,4,5]
You may reroll 3 times, enter y to do so
y
Enter the dice you wish to keep as a list
[6,6]
Your dice are: [6,6,4,3,6]
You may reroll 2 times, enter y to do so
n
Your dice are: [6,6,4,3,6]
"write slot to assign it to"
Sixes
"adding to slot Sixes"
Slot is either taken or does not exist
"write slot to assign it to"
```

Every round the player, whose turn it currently is, will be presented with their current board state, marked with their name, and five dice. They will be asked if they wish to re-roll their dice, which they may do three times. To re-roll the user has to press 'y' and then write in the dice they wish to keep as a list. Submitting this list will cause a re-roll of every dice that did not occur in the list. The user will then be presented with a new list of their re-rolled dice. Once a player is satisfied with their dice they get to choose a slot in their game table to add their dice to. If they try to put their dice in an illegal slot for the dice they have that slot will be marked with a 0. Otherwise the code will evaluate which dice of the player can be added to the slot and fill that in with the point the user would get for that move. A slot can only be selected once. If the player cannot place their dice in any other than a selected slot, the score in that slot will be replaced by a 0.

Once a player has chosen a slot to add their dice to, the next player's turn will begin. The game is finished once any player has filled in all the slots in their table. The program will then calculate all the players' scores. If a player gets more than 63 points in the first part of the game table, slots One through Sixes, they will get a bonus of 50 points. The program will present the scores of the players as a list, where the largest score is presented last and the smallest score is presented first. The program will then ask the user if they wish to play again with the same players, play again with different players or quit the game.

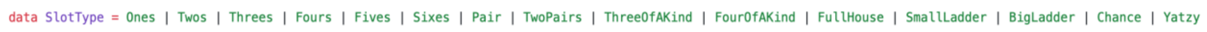


```
Player1: 218
The game is now over. Here are the players in order of score:
press y to play again with the same players, press x to play again with new players, press anything else to end the game
n
*** Exception: ExitSuccess
*Main>
```

3. Code Documentation

3.1 Data Structures

3.1.1 SlotType



```
data SlotType = Ones | Twos | Threes | Fours | Fives | Sixes | Pair | TwoPairs | ThreeOfAKind | FourOfAKind | FullHouse | SmallLadder | BigLadder | Chance | Yatzy
```

SlotType only serves the purpose of representing different slots in a game table.

3.1.2 Player

```
data Player = Player String Bool [(SlotType,Int)] deriving (Show, Eq)
```

The data type Player represents the board state for the players of the game. Player displays the player's name as a string. If the board belongs to a computer player the bool value will equate to True, otherwise it will be False. The data type represents the game table as a list of 15 tuples, containing a SlotType constructor and an Int (the score).

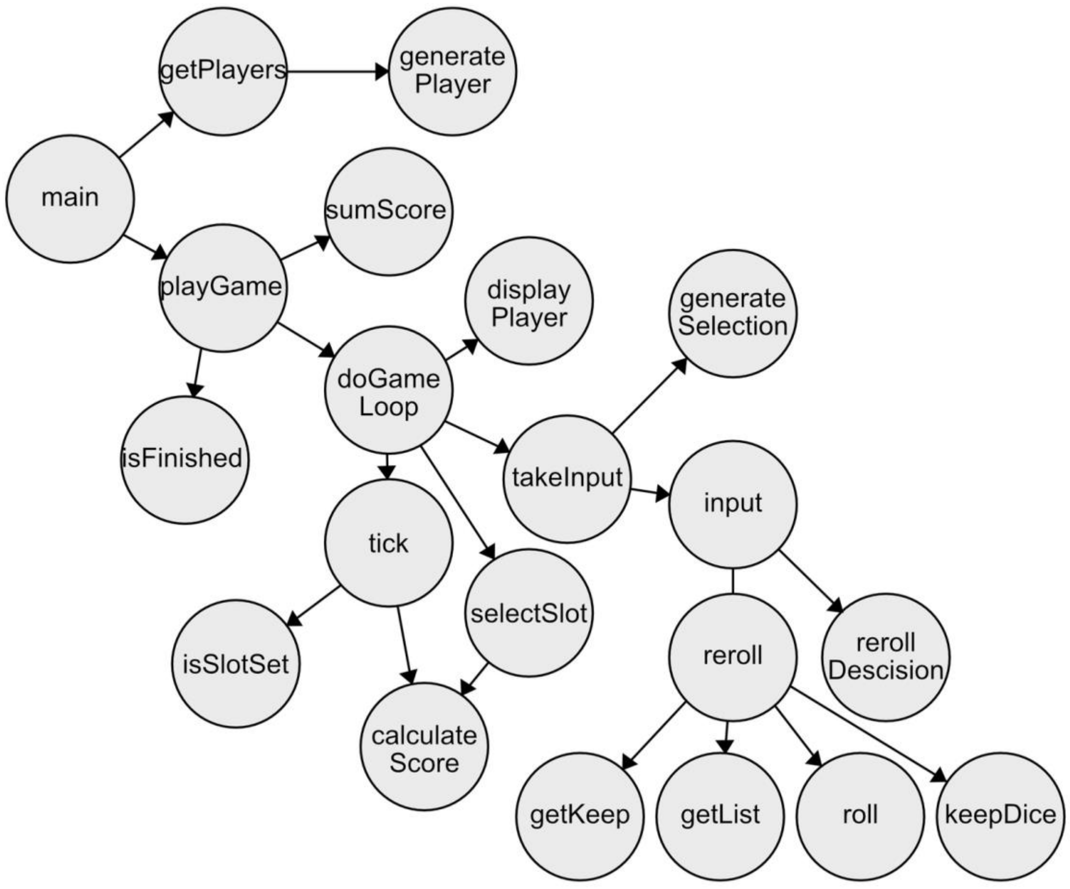
3.1.3 Selection

```
type Selection = [(Int,Bool)]
```

Selection is used to represent the dice in the game. It is made up of a list of five tuples, each containing an Int and a Bool. The Int is the number-value of the dice. The Bool element decides if the dice should be able to be rerolled or not. If the boolean is True the dice can be rerolled, if it’s False the dice cannot be rerolled.

3.2 Algorithms and Functions

Our program consists of different components that are all tied together by one main function. This can be illustrated as such:



Where main goes out to two different branches: getPlayers and playGame. The function playGame's help functions can be categorized into two components: Dice-Functions and Game-State-Functions. The function getPlayers can be categorized as Player-Functions. This leaves us with the three main components that make up the program: Players, Dice and the Game Board.

3.2.1 Player Functions

The player functions deal with adding players to the game, keeping track of the players that have been added and identifying whether the player is computer controlled or not. This section of the program has the main component 'getPlayers', which gets help by the functions getComInput and generatePlayer.

getPlayers

```haskell
getPlayers :: Int -> IO [Player]
getPlayers i = do
    putStrLn ("Enter the name of player " ++ show i)
    name <- getLine
    cpu <- getComInput
    putStrLn "input y if you want to stop adding players"
    stop <- getLine
    rs <- do
        if stop == "y" || stop == "Y"
            then return []
            else getPlayers (i+1)
    return (generatePlayer name cpu : rs)
    where
        {-getComInput
            gets user input on wether the player is a computer or not
            RETURNS: True if the player is a computer. False if they're not
            SIDE: writes to the terminal and takes input
        -}
        getComInput :: IO Bool
        getComInput = do
            putStrLn "is this player a computer?(y/n)"
            cpu <- getLine
            if cpu == "y" || cpu == "Y"
                then return True
                else if cpu == "n" || cpu == "N"
                    then return False
                    else getComInput
```

getPlayers lets the user add players to the game by constructing a list with each added player's identifiers: a string, an int and a bool value. The string represents the player's name. The int tells the program which player was added to the game in what order. The bool identifies if the player is controlled by the computer, where True means it is and False means that it is not.

1) The function takes a string input from the user and stores it in the name variable. It then calls the getComInput function. This function asks whether the player is a computer or not. If they are the function returns True, if they are not the function returns False. The bool gets stored in the cpu variable.
2) The function asks the user if they wish to stop adding players to the game. The user can input ‘y’ for yes or anything else for no.

    2.1) If the user input is y or Y the variable ‘rs’ will store an empty list

    2.2) Otherwise the function will call itself, but with 1 added to its argument ‘i’, and the variable ‘rs’ will store a list of the new players added.

3) generatePlayer will get called with the name and cpu variables as its argument, and its result will get added to the ‘rs’ list.

3.2.2 Dice Functions

Yatzy is partly a game of chance, dependent on the randomness of a dice throw. And partly a game of logic, where the players have to choose which dice to keep in order to get the best score possible. For this section of the program we had to create one component for generating random dice and one component for the user to select and change their dice.

generateSelection

For the random dice generator we used a random number generator from the System.Random library. We got an understanding of the library from a stack overflow answer and took its randomList function [1].

```
generateSelection ::Int -> IO Selection
generateSelection 0 = return []
generateSelection i = do
    r <- randomRIO (1,6)
    rs <- generateSelection (i-1)
    return ((r,False):rs)
```

This function takes an integer as its argument and will generate that amount of random numbers between 1 and 6. Every randomly generated number gets placed in a list, and is paired with a False value in order to match the data type Selection.

roll and takeInput

We had to create some function that could re-roll selected dice and some function that let the player interact with the game and select dice, in order to let players pick which dice to keep and which to change each round of the game.

```
roll :: Selection -> IO Selection
roll [] = return []
roll ((x,False):xs) = do
    r <- randomRIO (1,6)
    rs <- roll xs
    return ((r, False) : rs)
roll ((x,True):xs) = do
    rs <- roll xs
    return ((x,True):rs)
```

We implemented the roll function, which takes a selection and checks the bool value of each dice. If the bool equates to True that dice will be rerolled. If the bool is still False however, the dice will be left as it was.

```
takeInput :: Bool -> IO Selection
takeInput cpu = do
  dice <- generateSelection 5
  input dice 3 cpu
```

takeInput is the main function for handling dice in the game. The function stores a selection in the dice variable in order to call the function 'input' with the newly generated dice as an argument. Input also takes the integer 3 and the players cpu.

```haskell
input :: Selection -> Int -> Bool -> IO Selection
input d s cpu = do
  if (s /= 0)
    then do
       putStrLn $ "Your dice are: " ++ show (map fst d)
       putStrLn $ "You may reroll " ++ (show s) ++ " times, enter y to do so"
       fin <- if cpu
           then do
            let r =  rerollDescision d
            putStrLn r
            return r
           else getLine
       if (fin == "y" || fin == "Y")
         then do
           dice <- reroll d cpu
           return dice
           input dice (s-1) cpu
         else do
           putStrLn $ "Your dice are: " ++ show (map fst d)
           return d
     else do
        putStrLn $ "Your dice are: " ++ show (map fst d)
        return d
```

The input function presents the user with the dice they have been given that round and asks if they wish to reroll, which they may do three times. If they do not wish to reroll the function simply returns the dice that the player was given at the start. If the player is controlled by the computer the rerollDecision will get called. If the player wishes to reroll they will have to input which of their given dice they wish to keep, and the rest will be rerolled. The function will then ask them if they wish to reroll again, stating that they may only reroll two more times now. This cycle will go on until either the player is satisfied with their dice or if they rerolled three times.

1) If s is larger than 0 the function will: Print a list of the players current dice. It will then tell the user that they may reroll ‘s’ amount of times, and to input ‘y’ to do so.

> 1.1) If the user input is ‘y’ the function will call the ‘reroll’ function. reroll lets the user input a list of ints and reroll the dice that were not mentioned in the list. The rerolled dice will get stored in the ‘dice’ variable. The function will then call itself with 1 subtracted from s and the new, rerolled dice as its arguments
>
> 1.2) Otherwise the function will print out the users current dice to the terminal.

2) Otherwise the function will print out the users current dice.

3.2.3 Game Table and Score Functions

As for the game table we had to create ways for a players to add dice to a slot, but also to make sure that a player could not cheat. For this we reacted the game-table function tick, and the score function calculateScore.

tick

```
tick :: Player -> Selection -> SlotType -> Player
tick (Player name cpu l) s target
   |isSlotSet l target = error "Slot is already set"
   |otherwise = Player name cpu (updateLedger l s target)
   where
   updateLedger :: [(SlotType, Int)] -> Selection -> SlotType -> [(SlotType, Int)]
   updateLedger ((slot,i):xs) s target
   --Variant: length ledger
     |slot == target = (slot, calculateScore target s) : xs
     |otherwise = (slot, i): updateLedger xs s target
```

The function takes the players' target slot (which slot they wish to add their dice to) as an argument and either updates the players ledger with the score they got from their dice, or prints out an error message if the move they were trying to do was illegal.

1) The function calls isSlotSet with the player's current ledger and their target slot. isSlotSet checks if the target slot has already been filled or not.
   1.1) If the move is not legal the function will create an error message
   1.2) Otherwise the function will update the players ledger, by calling the local function updateLedger, which finds the target slot in the players ledger and writes in their score, using the calculateScore function with the target slot and the players dice as its argument.

calculateScore

```
calculateScore :: SlotType -> Selection -> Int
calculateScore Ones [] = 0
calculateScore Ones ((x, _) : xs)
--Variant: length dice
  | x == 1 = 1 + calculateScore Ones xs
  | otherwise = calculateScore Ones xs
calculateScore Twos [] = 0
calculateScore Twos ((x, _) : xs)
--Variant: length dice
  | x == 2 = 2 + calculateScore Twos xs
  | otherwise = calculateScore Twos xs
```

The function calculateScore takes a SlotType and a selection and calculates what score the dice would get in that slot.

(The function is actually a lot longer than pictured.)

sumScore

```
sumScore :: Player -> Int
sumScore (Player _ _ tabel) = sum [x | x <- map snd tabel, x /= (unset)] + if oneThroughSix > 63 then 50 else 0
  where
    oneThroughSix = sum  [x | x <- take 6 $ map snd tabel, x /= (unset)]
```

sumScore calculates the total score for a player, by adding all the values in that players' ledger together. If the player scored higher than 63 points in the slots 1-6 they get an added bonus of 50 points.

3.2.4 Game AI

For our computer controlled players we chose to have them make their decisions based on a naive algorithm[2]. The algorithm will always try to go for a Yatzy while rolling. It will write in its result in whichever slot would yield them the highest points. The AI consists of three different functions.

rerollDecision

```
rerollDescision :: Selection -> String
rerollDescision s = if length (group $ map fst s) /= 1 then "y" else "n"
```

This function takes the current dice and checks if it would qualify for a Yatzy. If it does the function will return a 'n', since the computer does not have to reroll their dice anymore. If the dice is not a yatzy the computer will return a 'y', since it should reroll if possible. rerollDecision is used in input to determine whether

getKeep

```
getKeep :: Selection -> [Int]
getKeep s = maximumBy (compare `on` length) (group (sort (map fst s)))
```

This function decides which dice the computer should reroll, basing its decision on the highest numbers and which dice are most likely to get the player a Yatzy. It does that by keeping the numbers that the Selection has the most off of picking a higher number if there are multiple numbers with the same amount.

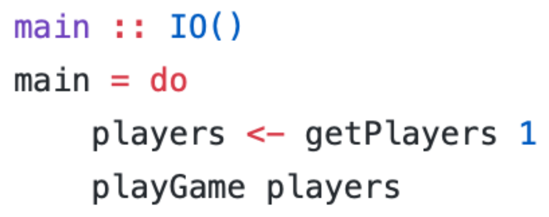
selectSlot

```
selectSlot :: Player -> Selection -> SlotType
selectSlot (Player _ _ table) s = fst $ head scores
    where
        scores = sortBy (flip compare `on` snd) (map (\(a,b) -> (a,calculateScore a s)) (filter (\(a,b) -> b ==unset) table))
```

This function decides which slot the computer should add their dice result to. It uses calculateScore and the player's current table state to figure out the best slot. The AI is bound by game rules and cannot cheat in order to win.

3.2.5 Game Functions

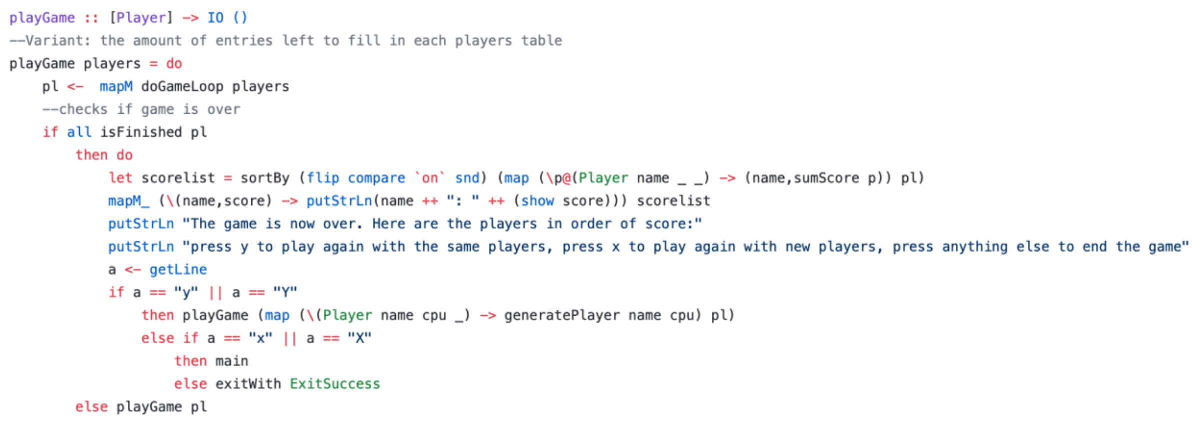
main



```haskell
main :: IO()
main = do
     players <- getPlayers 1
     playGame players
```

The function main is what actually pieces together all the different components of the program in order to make the game work. It calls the function getPlayers, which provides a list of all the players of the game, and stores it in the variable ‘players’. The function then calls ‘playGame’ with the newly constructed list of all the players of the game.

playGame



```haskell
playGame :: [Player] -> IO ()
--Variant: the amount of entries left to fill in each players table
playGame players = do
    pl <-  mapM doGameLoop players
    --checks if game is over
    if all isFinished pl
        then do
            let scorelist = sortBy (flip compare `on` snd) (map (\p@(Player name _ _) -> (name,sumScore p)) pl)
            mapM_ (\(name,score) -> putStrLn(name ++ ": " ++ (show score))) scorelist
            putStrLn "The game is now over. Here are the players in order of score:"
            putStrLn "press y to play again with the same players, press x to play again with new players, press anything else to end the game"
            a <- getLine
            if a == "y" || a == "Y"
                then playGame (map (\(Player name cpu _) -> generatePlayer name cpu) pl)
                else if a == "x" || a == "X"
                    then main
                    else exitWith ExitSuccess
        else playGame pl
```

The playGame function runs the game by calling doGameLoop repeatedly. When the game is finished it compares all the players' scores and lists the scores of each player from lowest to highest. The function then gives the user three different choices: to play again with the same players they are to input ‘y’. To play again with different players they are to input ‘x’ and to quit playing they can press anything else.

1) The function calls doGameLoop on every player in the game, and store the result for every player in the ‘pl’ variable
2) call isFinished on every player in the ‘pl’ list.
    2.1) The function calls isFinished on all the results stored in the variable pl, if someone has finished the game:
        2.1.1) Create a sorted list of tuples containing every players name and their score
        2.1.2) print every tuple of the list on their own row, starting with the first (lowest score) tuple, and announce that those are the result of the game.
        2.1.3) The function then presents the user with the choices to:
            - input ‘y’ to play the game again with the same players. In which case the function will reset all the players scores and ledgers and call on itself.
            - To input ‘x’ to play again with new players. In which case the function will call the main function.

- Or to press anything else to quit playing the game. In which case the function will exit.

2.2) Otherwise the function will call itself, which will start the next round of the game with the players current ledgers.

doGameLoop

```haskell
doGameLoop :: Player -> IO Player
doGameLoop p@(Player _ cpu _) = do
    displayPlayer p
    selection <- takeInput cpu
    getSlotType selection
    where
        {-getSlotType sel
            gets input for which slot to add the result to
            RETURNS: a player with sel applied to the slot
            SIDE: prints to the terminal and takes input
        -}
        getSlotType :: Selection -> IO Player
        getSlotType sel  = do
            print "write slot to assign it to"
            slot <- if cpu
                then return $ show (selectSlot p sel)
                else getLine
            catch (do
                target <- evaluate (read slot)
                print ("adding to slot " ++ (show target))
                evaluate (tick p sel target)
                ) ((\_ -> do
                    putStrLn "Slot is either taken or does not exist"
                    getSlotType sel)::SomeException -> IO Player)
```

The doGameLoop function plays a single round of yatzy for one player of the game.

1) The function calls the displayPlayer function, which prints out a players current game table to the terminal.
2) The function calls the takeInput function with the players cpu. This call will return what dice the player chose to use for this round. The result gets stored in the selection variable.
3) The function calls the local function getSlotType with the selection variable as its argument. getSlotType will ask the player to assign their dice to a slot.

3.1) If the player is computer-controlled getSlotType will call selectSlot and return the result,

3.2) If the player is not controlled by a computer the function will let the user input the slot they wish to add their result to.

3.2.1) If the move is legal the tick function will get called and the players ledger will get updated with the new result.
3.2.2) If the move is not legal the function will print: ‘’Slot is either taken or does not exist”, and getSlotType will get called again in order to let the player add their dice to a legal slot.

4. Known Shortcomings

Computer players are quite clumsy and greedy. They won’t play smartly (trying to get bonuses or rolling for ladders) but are instead fixated on a singular big goal even trying to obtain it while having it be unobtainable.

There are also some shortcomings in the implemented code. The program needs to use random generation to achieve goals, causing a lot of the data to be impure. It also repeatedly asks the user if they wish to reroll which feels slightly clumsy.

There is also no way to exit or save the game in the middle of a game. The user has to finish the game in order to exit. The user cannot either save player profiles or track victories which would be a neat feature for a group that plays yatzy frequently.

5. Sources

[1]: Carsten, List with random numbers in Haskell, Stack Overflow, 9 June 2015, Accessed: 24 February 2021 [Online], Available: https://stackoverflow.com/a/30741139

[2]: T. Beeksma, P. Kenny and M. Renner, Yahtzee Algorithms, University of North Carolina Wilmington, 25 November 2018, Accessed On: 1 March 2021, [Online], Available: http://people.uncw.edu/tagliarinig/Courses/380/F2018%20papers%20and%20presentations/High%20Rollers/Paper-CSC380.pdf