Marco Tran   
013281058  
CECS 424  
11/14/2018

open System.Diagnostics

open System

// An "enum"-type union for card suit.

type CardSuit =

| Spades

| Clubs

| Diamonds

| Hearts

// Kinds: 1 = Ace, 2 = Two, ..., 11 = Jack, 12 = Queen, 13 = King.

type Card = {suit : CardSuit; kind : int}

// The state of a single game of blackjack. Tracks the current deck, the player's hand, and the dealer's hand.

type GameState = {deck : Card list; playerHand : Card list; dealerHand : Card list}

// A log of results from many games of blackjack.

type GameLog = {playerWins : int; dealerWins : int; draws : int}

// Identifying who owns a given hand.

type HandOwner =

| Player

| Dealer

// UTILITY METHODS

// Returns a string describing a card.

let cardToString card =

let kind = match string card.kind with

| "1" -> "Ace"

| "11" -> "Jack"

| "12" -> "Queen"

| "13" -> "King"

| n -> string n

// "%A" can print any kind of object, and automatically converts a union (like CardSuit)

// into a simple string.

sprintf "%s of %A" kind card.suit

// Returns the "value" of a card in a poker hand, where all three "face" cards are worth 10

// and an Ace has a value of 11.

let cardValue card =

let value = match card.kind with

| 1 -> 11

| 11 | 12 | 13 -> 10 // This matches 11, 12, or 13.

| n -> n

value

let handTotal hand =

let sum = hand

|> List.map cardValue

|> List.sum

let numAces = hand

|> List.map(cardValue)

|> List.filter(fun x -> x = 1)

|> List.length

// Adjust the sum if it exceeds 21 and there are aces.

if sum <= 21 then

// No adjustment necessary.

sum

else

// Find the max number of aces to use as 1 point instead of 11.

let maxAces = (float sum - 21.0) / 10.0 |> ceil |> int

// Remove 10 points per ace, depending on how many are needed.

sum - (10 \* (min numAces maxAces))

// FUNCTIONS THAT CREATE OR UPDATE GAME STATES

// Creates a new, unshuffled deck of 52 cards.

// A function with no parameters is indicated by () in the parameter list. It is also invoked

// with () as the argument.

let makeDeck () =

// Make a deck by calling this anonymous function 52 times, each time incrementing

// the parameter 'i' by 1.

// The Suit of a card is found by dividing i by 13, so the first 13 cards are Spades.

// The Kind of a card is the modulo of (i+1) and 13.

List.init 52 (fun i -> let s = match i / 13 with

| 0 -> Spades

| 1 -> Clubs

| 2 -> Diamonds

| 3 -> Hearts

{suit = s; kind = i % 13 + 1})

// This global value can be used as a source of random integers by writing

// "rand.Next(i)", where i is the upper bound (exclusive) of the random range.

let rand = new System.Random()

// Creates a new game state by creating and shuffling a deck, and dealing 2 cards to

// each player.

// Call this function by writing "newGame ()".

let newGame () =

// Shuffles a list. Don't worry about this.

let shuffleList list =

let arr = List.toArray list

let swap (a: \_[]) x y =

let tmp = a.[x]

a.[x] <- a.[y]

a.[y] <- tmp

Array.iteri (fun i \_ -> swap arr i (rand.Next(i, Array.length arr))) arr

Array.toList arr

// Create the deck, and then shuffle it

let deck = makeDeck ()

|> shuffleList

// Construct the starting hands for player and dealer.

let player = [deck.Head ; deck.Tail.Tail.Head] // First and third cards.

let dealer = [deck.Tail.Head ; deck.Tail.Tail.Tail.Head] // Second and fourth.

// Return a fresh game state.

{

deck = List.skip 4 deck;

playerHand = player;

dealerHand = dealer;

}

// Given a current game state and an indication of which player is "hitting", deal one

// card from the deck and add it to the given person's hand. Return the new game state.

let hit (handOwner : HandOwner) (gameState : GameState) = // these type annotations are for your benefit, not the compiler

// Return the new game state, \*including\* new the deck with the top card removed.

match handOwner with

| Player -> {

deck = gameState.deck.Tail;

playerHand = gameState.deck.Head::gameState.playerHand;

dealerHand = gameState.dealerHand

}

| Dealer -> {

deck = gameState.deck.Tail;

playerHand = gameState.playerHand;

dealerHand = gameState.deck.Head::gameState.dealerHand

}

// Take the dealer's turn by repeatedly taking a single action, hit or stay, until

// the dealer busts or stays.

let rec dealerTurn gameState =

let dealer = gameState.dealerHand

let score = handTotal dealer

printfn "Dealer's hand: %A; %d points" (List.map cardToString dealer) score

// Dealer rules: must hit if score < 17.

if score > 21 then

printfn "Dealer busts!\n"

// The game state is unchanged because we did not hit.

// The dealer does not get to take another action.

gameState

elif score < 17 then

printfn "Dealer hits\n"

// The game state is changed; the result of "hit" is the new state.

// The dealer gets to take another action using the new state.

gameState

|> hit Dealer

|> dealerTurn

else

// The game state is unchanged because we did not hit.

// The dealer does not get to take another action.

printfn "Dealer must stay\n"

gameState

// Take the player's turn by repeatedly taking a single action until they bust or stay.

let rec playerTurn (playerStrategy : GameState->bool) (gameState : GameState) =

let player = gameState.playerHand

let score = handTotal player

printfn "Player's hand: %A; %d points" (List.map cardToString player) score

if score > 21 then

printfn "Player busts!\n"

gameState

elif (playerStrategy gameState) then

gameState

|> hit Player

|> playerTurn playerStrategy

else

printfn "Player must stay\n"

gameState

// Plays one game with the given player strategy. Returns a GameLog recording the winner of the game.

let oneGame playerStrategy gameState =

printfn "Dealer is showing: %s" (cardToString gameState.dealerHand.Head)

let oneTurnGame = playerTurn playerStrategy gameState

|> dealerTurn

let playerScore = handTotal oneTurnGame.playerHand

let dealerScore = handTotal oneTurnGame.dealerHand

if (playerScore <= 21) && (dealerScore > 21 || playerScore > dealerScore) then

printfn "++++ Player Wins ++++\n"

{playerWins = 1; dealerWins = 0; draws = 0}

elif(playerScore = dealerScore) then

printfn "==== Draw! ====\n"

{playerWins = 0; dealerWins = 0; draws = 1}

else

printfn "---- Dealer Wins ----\n"

{playerWins = 0; dealerWins = 1; draws = 0}

// Recursively plays n games using the given playerStrategy.

let manyGames n playerStrategy =

// This tail-recursive helper implements the manyGames logic.

let rec manyGamesTail n playerStrategy logSoFar =

if n = 1 then

logSoFar

else

let log = newGame()

|> oneGame playerStrategy

manyGamesTail (n-1) playerStrategy {

playerWins = logSoFar.playerWins + log.playerWins;

dealerWins = logSoFar.dealerWins + log.dealerWins;

draws = logSoFar.draws + log.draws

}

manyGamesTail n playerStrategy {playerWins = 0; dealerWins = 0; draws = 0}

// PLAYER STRATEGIES

let interactivePlayerStrategy gameState =

printfn "Hit? y/n"

let answer = System.Console.ReadLine()

// Return true if they entered "y", false otherwise.

answer = "y"

// Player never hits

let inactivePlayerStrategy gameState =

false

// Player hits only when less than 15

let cautiousPlayerStrategy gameState =

let player = gameState.playerHand

let score = handTotal player

// Hit if score is below 15

score < 15

// Player hits unless score is 21 or greater

let greedyPlayerStrategy gameState =

let player = gameState.playerHand

let score = handTotal player

// Hit if score is below 21

score < 21

let coinFlipPlayerStrategy gameState =

rand.Next(2) = 1

[<EntryPoint>]

let main argv =

let numGames = 1000

let results = manyGames numGames inactivePlayerStrategy

printfn "Inactive Player Strategy\n"

printfn "Player win: %.2f%%, %d/%d" ((float results.playerWins / float numGames) \* float 100) results.playerWins numGames

printfn "Dealer win: %.2f%%, %d/%d" ((float results.dealerWins / float numGames) \* float 100) results.dealerWins numGames

printfn "Draws: %.2f%%, %d/%d" ((float results.draws / float numGames) \* float 100) results.draws numGames

Console.ReadKey() |> ignore

0 // return an integer exit code

    
  
 