

David Šajina

Izveštaj Connect 4 igre

Projektni rad

Sadržaj

- 1. Detaljan opis arhitekture igre i korištenih funkcijskih koncepta
 - 1.1 Main.hs
 - 1.2. GameLogic.hs
 - 1.3. Board.hs
- 2. Izgled sučelja igre
- 3. Analiza funkcijskog programiranja pri upravljanju stanjima igre i korisničkim ulazima
 - 3.1 Nepromjenjivost (Immutability):
 - 3.2 Čiste funkcije:
 - 3.3 Rekurzija
 - 3.4 Upravljanje I/O
- programiranje 4. Diskusija o prednostima i izazovima razvoja igre koristeći funkcijsko
 - 4.1 Prednosti
 - 4.2 Izazovi
- 5. Pregled i usporedba s tradicionalnim pristupima u razvoju igara
 - 5.1 Tradicionalni pristupi:
 - 5.2 Funkcijsko programiranje:

Razvoj igre Connect 4 u Haskell-u koristeći funkcijsko programiranje

1. Detaljan opis arhitekture igre i korištenih funkcijskih koncepta

Igra Connect 4 implementirana je u Haskell-u koristeći tri glavna modula: `Main`, `GameLogic`, i `Board`. Svaki modul ima specifičnu ulogu u arhitekturi igre.

1.1 Main.hs

- Modul `Main` sadrži glavni ulaz u program i kontrolira tok igre.
- `main` funkcija postavlja način za standardni izlaz i započinje igru pozivajući `gameLoop` s inicijalnom pločom i igračem 1.

- `gameLoop` je rekurzivna funkcija koja prikazuje ploču, prima ulaz korisnika (stupac), provjerava valjanost poteza, ažurira ploču i provjerava je li igra završena (pobjeda ili neriješeno). Ako igra nije gotova, poziva se rekurzivno za sljedećeg igrača.

```
gameLoop :: Board -> Int -> IO ()
gameLoop board player = do
  printBoard board
  putStrLn $ "Player " ++ show player ++ "'s turn. Enter column (1-7):"
  col <- getLine
  let column = read col - 1
  if isValidMove board column
    then do
      let newBoard = makeMove board column player
      if checkWin newBoard player
        then printBoard newBoard >> putStrLn ("Player " ++ show player ++ " won, game over.")
        else
          if isBoardFull newBoard
            then printBoard newBoard >> putStrLn "The game is a draw!"
            else gameLoop newBoard (switchPlayer player)
    else putStrLn "Invalid move. Try again." >> gameLoop board player
```

1.2. GameLogic.hs

- Modul `GameLogic` sadrži funkcije vezane za logiku igre.
- `switchPlayer` mijenja trenutnog igrača.

```
switchPlayer 1 = 2
switchPlayer 2 = 1
```

- `checkWin` provjerava je li trenutni igrač pobijedio. Ova funkcija koristi različite smjerove za provjeru linija (horizontalne, vertikalne, dijagonalne) i poziva pomoćne funkcije da provjeri sve pozicije na ploči.

```
checkWin :: Board -> Int -> Bool
checkWin board player = any (checkDirection player) directions
  where

    -- smjerovi za provjeru pobjede
    directions = [horizontal, vertical, diagonal1, diagonal2]
    horizontal = [(0, 0), (0, 1), (0, 2), (0, 3)]
    vertical   = [(0, 0), (1, 0), (2, 0), (3, 0)]
    diagonal1  = [(0, 0), (1, 1), (2, 2), (3, 3)]
    diagonal2  = [(0, 0), (1, -1), (2, -2), (3, -3)]

    -- provjera svih smjerova
    checkDirection :: Int -> [(Int, Int)] -> Bool
    checkDirection player direction = any (hasLine player direction) allPositions
```

```

allPositions = [(row, col) | row <- [0 .. 5], col <- [0 .. 6]]

-- provjera 4 tocke u liniji (horizontalno, vertikalno, diagonalno lijevo, diagonalno
desno)
hasLine :: Int -> [(Int, Int)] -> (Int, Int) -> Bool
hasLine player direction (row, col) = all (isValidAndMatches player) (map
(applyDirection (row, col)) direction)

applyDirection :: (Int, Int) -> (Int, Int) -> (Int, Int)
applyDirection (row, col) (dRow, dCol) = (row + dRow, col + dCol)

isValidAndMatches :: Int -> (Int, Int) -> Bool
isValidAndMatches player (newRow, newCol) =
    isInBounds (newRow, newCol) && (board !! newRow !! newCol) == player

-- Unutar polja (1-7)
isInBounds :: (Int, Int) -> Bool
isInBounds (row, col) = row >= 0 && row < 6 && col >= 0 && col < 7

```

1.3. Board.hs

- Modul `Board` upravlja pločom igre.
- `initialBoard` postavlja početnu ploču kao listu lista s vrijednostima 0 (prazno).

```

initialBoard :: Board
initialBoard = replicate 7 (replicate 7 0)

```

- `printBoard` ispisuje ploču u čitljivom formatu.

```

printBoard board = do
    mapM_ printRow (transpose board)
    putStrLn " 1 2 3 4 5 6 7"
    where
        printRow row = putStrLn $ concatMap showCell row
        showCell 0 = " ."
        showCell 1 = " X"
        showCell 2 = " O"

```

- `makeMove` ažurira ploču za određenog igrača u danom stupcu.

```
makeMove :: Board -> Int -> Int -> Board
makeMove board col player =
    let (before, targetCol : after) = splitAt col board
        newCol = placeInColumn targetCol player
    in before ++ (newCol : after)
where
    placeInColumn :: [Int] -> Int -> [Int]
    placeInColumn col player = reverse (placeDisc (reverse col) player)

    placeDisc (0 : xs) p = p : xs
    placeDisc (x : xs) p = x : placeDisc xs p
    placeDisc [] _ = [] -- Slučaj da je pun stupac
```

- `isValidMove` provjerava je li potez valjan (tj. je li stupac unutar granica i ima li slobodnih mjesta).

```
isValidMove :: Board -> Int -> Bool
isValidMove board col
    | col < 0 || col >= length board = False -- Provjera da je u rangu.
    | otherwise = any (== 0) (board !! col)
```

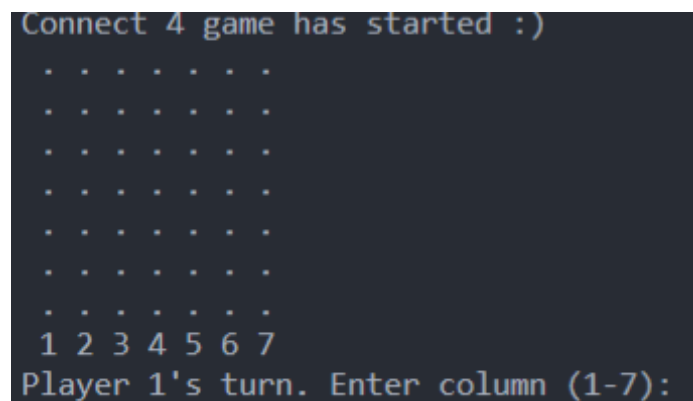
- `isBoardFull` provjerava je li ploča puna.

```
isBoardFull = all (all (/= 0))
```

2. Izgled sučelja igre

Sučelje je napravljeno u terminalu bez nikakvog dodatnog paketa za GUI

Možemo vidjeti kako izgleda inicijalna ploča koja se sastoji od 7x7 polja (Slika 1).



```
Connect 4 game has started :)
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .
1 2 3 4 5 6 7
Player 1's turn. Enter column (1-7):
```

Slika 1 : Prikaz inicijalne ploče

Nakon pokrenute igre, igrač unosi u koji stupac želi staviti svoju točku (Slika 2)

```

Player 1's turn. Enter column (1-7):
1
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .
X . . . . . .
1 2 3 4 5 6 7
Player 2's turn. Enter column (1-7):

```

Slika 2: Prikaz odigranog poteza 1 igrača 1

Nakon što je igrač 1 odigrao, dolazi na red igrač 2 (Slika 3)

```

Player 2's turn. Enter column (1-7):
5
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .
X . . . 0 . .
1 2 3 4 5 6 7
Player 1's turn. Enter column (1-7):

```

Slika 3: Prikaz odigranog poteza za igrača 2

Nakon što igrač postavi 4 točke u horizontalnu, vertikalnu ili dijagonalnu liniju pobjedi (Slika 4)

```

. . . . . . .
. . . . . . .
. . . . . . .
. . 0 X X . .
0 0 X X X . .
X 0 X 0 0 . .
X X X 0 0 0 .
1 2 3 4 5 6 7
Player 1 won, game over.

```

Slika 4: Prikaz pobjede igrača 1

3. Analiza funkcijskog programiranja pri upravljanju stanjima igre i korisničkim ulazima

Funkcijsko programiranje koristi nepromjenjive podatke i čiste funkcije, što olakšava upravljanje stanjima igre. Evo kako se to manifestira u ovoj igri:

3.1 Nepromjenjivost (Immutability):

- Stanja igre (ploča) su nepromjenjiva. Svaka promjena stanja rezultira novom pločom, dok originalna ostaje nepromijenjena. To pomaže u izbjegavanju grešaka povezanih s nepredvidivim promjenama stanja.

3.2 Čiste funkcije:

- Funkcije kao što su `makeMove`, `checkWin` i `switchPlayer` su čiste, što znači da uvijek daju isti izlaz za isti ulaz bez nuspojava. Ovo čini testiranje i razumijevanje koda lakšim.

3.3 Rekurzija

- Rekurzivne funkcije, poput `gameLoop`, omogućuju elegantno upravljanje ponavljajućim procesima (na primjer, izmjenom poteza između igrača) bez potrebe za eksplicitnim petljama ili mutabilnim varijablama.

3.4 Upravljanje I/O

- Ulazi korisnika se čitaju i obrađuju na način koji minimizira mogućnost grešaka. Na primjer, unos stupca se validira prije nego što se potez napravi.

4. Diskusija o prednostima i izazovima razvoja igre koristeći funkcijsko programiranje

4.1 Prednosti

1. Jednostavnost i jasnoća:
 - Kod je često kraći i jednostavniji za razumijevanje zbog deklarativnog pristupa i fokusiranja na "što" treba učiniti umjesto "kako".
2. Manje grešaka:
 - Nepromjenjivost i čiste funkcije smanjuju mogućnost grešaka povezanih s promjenama stanja i nuspojavama.
3. Lakše testiranje:
 - Zbog čistoće funkcija, testiranje je jednostavnije jer funkcije nemaju nuspojave i uvijek daju isti rezultat za isti ulaz.

4.2 Izazovi:

1. Učenje i prilagodba:
 - Teža prilagodba na funkcijski stil nakon navike imperativnog programiranja zbog drugačijeg načina razmišljanja i rješavanja problema.

2. Debugging:

- Debugging funkcijskog koda je puno izazovnije zbog rekurzivnih poziva i manjka eksplicitnih stanja. Teže je diagnosticirati gdje je nešto pošlo po krivom. Nije jednostavno kao što je to za imperativne jezike.

3. Kompleksnost u složenijim projektima:

- Kada projekti počnu skalirati, počinje biti jako teško pratiti sve funkcije, stanja i interakcije.

5. Pregled i usporedba s tradicionalnim pristupima u razvoju igara

5.1 Tradicionalni pristupi:

- Imperativno programiranje:
- Koristi eksplicitne naredbe za promjenu stanja.
- Mutabilne varijable omogućuju direktne promjene stanja, što može biti efikasnije, ali sklonog greškama.
- Objektno orijentirano programiranje:
- Stanja i ponašanja se grupiraju unutar objekata.
- Koristi klase i objekte za modeliranje igre, što omohućava bolju i čistiju organizaciju koda, ali može dovesti do kompleksne hijerarhije i teškog praćenja stanja.

5.2 Funkcijsko programiranje:

- Deklarativan pristup:
- Fokusira se na opisivanje što treba učiniti, a ne kako.
- Koristi nepromjenjive podatke i čiste funkcije, što smanjuje greške povezane s promjenama stanja.
- Jednostavnost i jasnoća:
- Kod je često lakši za čitanje i razumijevanje zbog deklarativnog pristupa i jasnog razdvajanja logike i stanja.

Funkcijsko programiranje nam daje veću pouzdanost i održivost koda kada je ispravno napisan. Zahtjeva shvaćanje svoje logike programiranja i drugačije pristupe programiranju od tradicionalnih.