PoP – ugeopgave 7g

Sofie Elisabeth Stone Havn <gdx888>
Stine Wittendorff Petersen <zcv967>
Sofus Ostrowska Bjørn <dxq257>

4. november 2021

Introduktion

Vi har lavet et program, som fremstiller et puzzle, kaldet Rotate. Programmet indeholder hverken for-, while-løkker, arrays, eller 'mutable' variable. Brugeren er i stand til at interagere med programmet i konsollen gennem input fra tasterturet. Vi har særligt arbejdet med pattern-matching, rekursion, og sum-typer.

Programanalyse og design

Vi vil nu gennemgå vores funktioner, og vi starter fra toppen med typerne og de funktioner `rnd` og `boardSizeFun`:

```
module rotate
type Position = int
type Board = char list
let rnd = System.Random ()
let boardSizeFun (b : Board) : int =
    b |> List.length |> double |> sqrt |> int
```

Listing 1: Typer og funktioner

Her defineres først, vores module: `rotate`. Vi definerer dernæst typerne `Position` og `Board`, hhv. som en integer og en character list. Derudover defineres `rnd` som `System.Random()` som vil optræde senere i vores `scramble` funktion. Til sidst har vi funktionen `boardSizeFun`, som tager et `Board`, og udregner kvadratroden af længden, som viser sig at være en meget gavnlig operation i de følgende funktioner.

```
let create (n: uint) : Board =
  let lst = ['a' .. 'y']
  match n with
  |n when n > 5u  -> []
  |_ -> lst.[0 ..(int(n*n)-1)]
```

Listing 2: Funktionen create

Create: Der er blevet benyttet pattern-matching på argumentet 'n'. Der er to muligheder, enten er det et ulovligt input, hvis $n > 5u$, som vil outputte en tom liste, og ellers laves et Board, vha. slicing, fra det første element i vores liste, `lst`, 'a', til $n^2 - 1$. Det er en relativ simpel og effektiv funktion.

```
let rec board2Str (b : Board) : string =
  let str (b : Board) : string = List.toArray b |> System.String
  match b with
  | x when (boardSizeFun (b) = 2) ->
    (str (b.[0 .. 1])) + "\n" + (str (b.[2 .. 3]))
        |> String.collect (sprintf " %c")
  | x2 when (boardSizeFun (b) = 3) ->
    (str (b.[0..2])) + "\n" + (str (b.[3..5])) + "\n" + (str (b.[6..9]))
        |> String.collect (sprintf " %c")
  | x3 when (boardSizeFun (b) = 4) ->
    (str (b.[0..3])) + "\n" + (str (b.[4..7])) + "\n" + (str (b.[8..11])) + "\n" +
        (str(b.[12..15])) |> String.collect (sprintf " %c")
  | x4 when (boardSizeFun (b) = 5) ->
    (str (b.[0..4])) + "\n" + (str (b.[5..9])) + "\n" +
        (str (b.[10..14])) + "\n" + (str (b.[15..19])) + "\n" +
            (str (b.[20..24])) |> String.collect (sprintf " %c")
  |_ -> "Illegal input"
```

Listing 3: Funktionen board2Str

Board2Str: Til denne funktion, er der også blevet benyttet pattern-matching, denne gang med argumentet 'b'. Her har vi 5 forskellige cases. Vi matcher på de forskellige mulige board størrelser, som kan være enten 2,3,4 eller 5. Det sidste case, er alt andet input end de 4 størrelser. Denne case er egentlig irrelevant, da vores game applikationsfil tager sig af ulovlige input. Samtidig tillader create ikke fremstillingen af ulovlige boards. Vi har dog valgt at tage casen med, for at få en komplet pattern matching på funktionen selv. En af ulemperne ved funktionen, er at den ikke er særlig elegant. På grund af længden på koden, kan det være svært for en udefrakommende, at læse koden.

```
let validRotation (b : Board) (p : Position) : bool =
   match p with
   | p when (1 + p = boardSizeFun (b)) -> false
   | p2 when (1 + p = 2 * (boardSizeFun (b))) -> false
   | p3 when (1 + p = 3 * (boardSizeFun (b))) -> false
   | p4 when (1 + p = 4 * (boardSizeFun (b))) -> false
   | p5 when (1 + p > b.Length - (boardSizeFun (b))) -> false
   | _ -> true
```

Listing 4: Funktionen validRotation

ValidRotation: Der bliver benyttet pattern matching på argumentet 'p'. Der bliver matchet på hvert muligt board. Hvor der i hver case, bliver afgjort om det er et ulovligt input. Denne løsningen er ikke specielt elegant.

```
let rotate (b: Board) (p: Position) : Board =
   match p with
   | p when (not(validRotation (b) (p))) -> b
   | _ -> b.[0..(p-1)] @ (b.[p + boardSizeFun (b)] :: (b.[p] ::
    (b.[(p+2) .. (p+boardSizeFun(b)-1)] @ (b.[p+boardSizeFun(b)+1] ::
        (b.[p+1] :: (b.[(p+boardSizeFun(b)+2)..])))))))
```

Listing 5: Funktionen rotate

Rotate: Her benyttes der igen patteren-matching på argumentet 'p', hvor vi har 2 cases. Enten er det en validRotation, ellers er det ikke. Hvis det er en validRotation, vil der blive udført en rotation af den valgte 2x2 firkant på boardet. Dette er ikke en særlig elegant løsning. Koden kunne være lavet mere elegant ved brug af nogle af de indbygget list-funktioner. Det har vi dog ikke været i stand til, at finde frem til.

```
let rec scramble (b: Board) (m: uint) : Board =

  let rec rotationsAllowed (b : Board) (p : Position) : Position =
    if validRotation (b) (p) then p
    else rotationsAllowed (b) (rnd.Next (b.Length))

  match m with
  | 0u -> b
  | _ -> scramble (rotate (b) (rotationsAllowed (b) (rnd.Next (b.Length)))) (m-1u)
```

Listing 6: Funktionen scramble

Scramble: Vores rekursive scramble funktion, gør brug af en rekursiv hjælpefunktion og patternmatching. Hjælpefunktionen, rotationsAllowed, finder et 'lovlig' input til den valgte boardsize, ved at kalde `b.Length |> (System.Random()).Next`, som altså finder et tilfældigt tal under længden af listen. Hvis det tilfældige tal ikke er en 'valid rotation' kalder funktionen sig selv, og bliver ved med dette indtil den har fundet et tal der opfylder `validRotation`.Vores patternmatching har dernæst basecasen, `0u -> b`, som altså returnerer boardet hvis $m = 0u$. Ved alle andre inputs, kalder scramble, sig selv, med et board roteret med `rotate` med et tilfældigt tal fra `rotationsAllowed`, og `m-1u`, som altså vil scramble boardet med m, lovlige roteringer, og til sidst returnere det færdige bræt når $m = 0u$.

```
let solved (b: Board) : bool =
   match b with
   | b when b = create((uint(boardSizeFun (b)))) -> true
   | _ -> false
```

Listing 7: Funktionen solved

Solved: Her benyttes der pattern-matching på argumentet 'b', hvor den tjekker ved at matche, om boardet er blevet løst eller ikke løst. Denne funktion kan umildbart ikke simplificeres, da den bare tjekker om boardet er i den løste form eller ej. For at tjekke om boardet er i den løste form, sammenlignes det med boardets tilstand, når det bliver fremstillet.

Game applikationsfilen

```
open rotate

let rec theGame (b : Board): unit =
      System.Console.Clear ()
      printfn "\n%s" (board2Str (b))
      printfn "\nIf you give up, simply type: quit"
      printfn "Choose your desired position to rotate:"
      let userInput = System.Console.ReadLine()
      match userInput with
      | "quit" ->
            printfn "\nThanks for playing, better luck next time"
            ()
      |_ -> match System.Int32.TryParse userInput with
            | false,int ->
                  printfn "Illegal input"
                  (theGame (b))
            | _ -> match userInput with
                        | x when (int(userInput)) < 1 ->
                              printfn "Illegal input"
                              (theGame (b))
                        | _ -> match b with
                                    |x when (solved (rotate (b) (int(userInput) - 1))) ->
                                          printfn "\n%s" (rotate b ((int userInput) - 1) |> board2Str)
//board2Str(rotate (b) (int(userInput)))
                                          printfn "Solved!"
                                          ()
                                    |_ -> rotate b ((int userInput) - 1) |> theGame

let rec game (unit) : unit =
  printfn "\nPlease choose your desired boardsize (2, 3, 4, 5):"
  let userInput2 = System.Console.ReadLine()
  match System.Int32.TryParse userInput2 with
      | false,int ->
            printfn "Illegal input"
            (game ())
      | _ -> match userInput2 with
                  | x when (int(userInput2)) > 5 ->
                        printfn "Illegal input"
                        (game ())
                  | x2 when (int(userInput2)) < 2 ->
                        printfn "Illegal input"
                        (game ())
                  | _ -> (theGame (scramble (create (uint(userInput2))) (10u)))

game ()
```

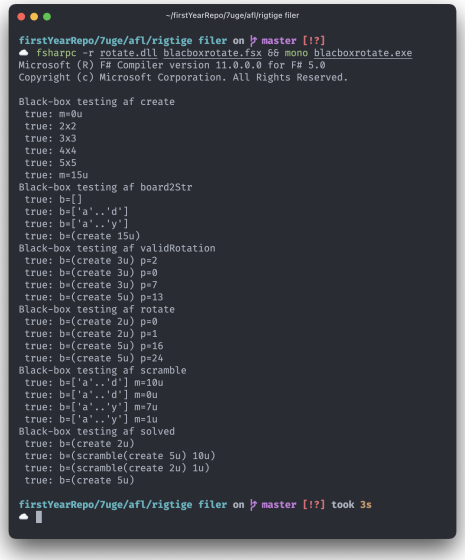
Figur 1: game programmet

Game.fsx: Vi har valgt at crash-sikre vores game-appliaktionsfil. Dette medfører at vi har en del match cases i starten af programmet. Vi har dog vurderet at det ekstra kodestykke var det værd. Da det sikrer at brugeren ikke kan crashe spillet, når der skrives et ulovligt input, såsom bogstaver, eller negative tal.

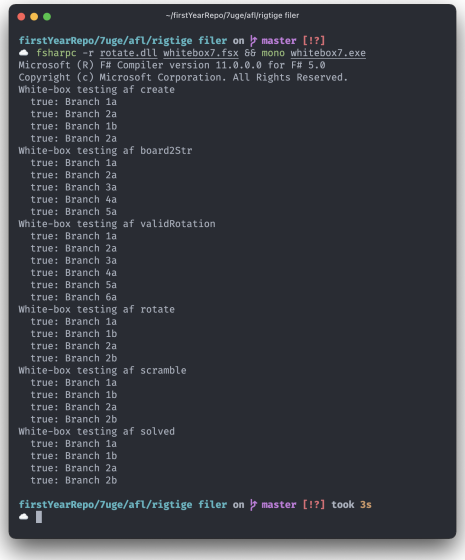
Afprøvning

Vi har udført en blackbox test og en whitebox test på vores program. De er begge uden fejl. Vi har dog en match case på vores funktion `board2Str` som aldrig anvendes, men som nævnt tidligere, har vi valgt at beholde denne

case, alligevel for at få et komplet pattern match på funktionen når den er alene og ikke benyttes i sammenhæng med de andre funktioner.



(a) Blackbox.test



(b) Whitebox-test

Figur 2: Vores lille test-suite

Besvarelse af 7g5

Problemet kunne skyldes at programmøren ikke har benyttet hale-rekursion, eller at der i rekursion bliver benyttet append, @, til at hægte listerne sammen, istedet for concatenate, ::. Når man appender to lister, kopierer man begge de lister, man vil sætte sammen, og sætter det ind i et nyt sted i hukommelsen med den samlede længde af de to lister. Hvis listen bliver lang, bliver det meget langsomt, og optager meget hukommelse, da der netop bliver oprettet en ny liste hver gang. Listerne skal desuden løbes igennem hver gang også, hvilket er med til at gøre det langsomt.

Konklusion

Vi har lavet et velfungerende rotate spil, der tager højde for samtlige brugerinput, og er derfor stort set 'crash safe'. Rotatefunktionen kunne klart gøres mere effektiv, ved blandt andet ikke at bruge @, dette er dog ikke muligt for os lige nu.