Opgavebesvarelse 6g

Shatin Nguyen (hlv332), Peter Asp Hansen (glt832) & Oliver Fontaine Raaschou (vns328)

23. februar 2023

I denne afleveringsopgave bygger vi et spil der minder om 2048[1]. Der er følgende krav til spillet:

1. Et Canvas som er kvadratisk med 3x3 placeringer.
2. Et felt skal have et af følgende farver rød (2), grøn (4), blå (8), gul (16) og sort (32).
3. Hver placering må maksimalt indeholde en brik.
4. Startbetingelsen skal være en rød felt og en blå felt placeret tilfældigt i 2 af de 9 felter.
5. Spillet skal kunne vippes i alle piletasternes retninger (højre,venstre,op og ned) når de indtastes.
6. Når spillet vippes i en retning, skal alle felter bevæge sig i samme retning, den længste vej.
7. Hvis to ens farvede brikker vippes ind i hinanden, skal der dannes en ny farvet brik. For eksempel, kombinationen af to røde felter danner et grønt felt.
8. To sorte felter kan kombineres til et sort felt.
9. Efter hvert træk, skal der placeres et nyt rødt felt på en tilfældig tom placering.
10. Spillet slutter når der ikke er flere ledige pladser og der ikke findes nogen træk som kombinerer felterne.

Til at løse opgaven, laver vi en dokumentations fil (`6g0Lib.fsi`) som indeholder alle funktionerne beskrevet fra opgavebeskrivelsen. Herefter laver vi en implementationen fil (`6g0Lib.fs`). Til sidst har vi også en `6g0App.fsx` hvilket er vores executable fil hvor vi starter spillet. Her i rapporten vil vi gennemgå de forskellige funktioner.

Først definere vi de forskellige `type` som anvendes i programmet taget fra opgavebeskrivelsen.

```
1 type pos = int*int //Position af to int
2 type value = Red|Green|Blue|Yellow|Black //Farverne anvendt
3 type piece = value*pos //En farve og position
4 type state = piece list //En liste med pieces
```

Figur 1: Type

Typen `pos` definere placeringen af et felt.
Typen `value` definere farverve et felt kan have.
Typen `piece` er et felt som er en samling af de to typer `value` og `pos` i en tuple. Første værdi i tuplen er farven og den anden er placeringen.
Typen `state` er en liste af pieces. Typen `state` skal derfor beskrive alle felternes placering og farve.

[1] https://2048.io/

Den første funktionen er `fromValue` som tager en `value` og konverterer det til en Canvas farve. Til at konvertere de forskellige farver, bruger vi `match-with`, som tager de forskellige farver og returnere den respektive Canvas farve.

```
let fromValue (v:value) : Canvas.color =
    match v with
        |Red -> Canvas.red
        |Green -> Canvas.green
        |Blue -> Canvas.blue
        |Yellow -> Canvas.yellow
        |Black -> Canvas.black
```

Figur 2: `fromValue`

Anden funktionen `nextColor` tager en `value` og returnere en ny `value`. Denne funktion skal bruges til at definere den næste farve når to felter af samme farve kombineres. Til at lave funktionen, bruger vi `match-with`, hvor betingelser er, hvis `c` = Red returnere Green, hvis `c` = Green, returnere funktionen Blue osv. Hvor c er den nuværende farve på feltet. Til sidst i koden har vi sørget for at opretholde spillets krav 8. Det vil altså sige at hvis farven er sort, forbliver den sort også selvom de slås sammen.

```
let nextColor (c:value) : value =
    match c with
        |Red -> Green
        |Green -> Blue
        |Blue -> Yellow
        |Yellow -> Black
        |Black -> Black

```

Figur 3: `nextColor`

Funktionen `filter` tager en integer `k` og en state `s` og returnerer de `pieces` fra `state`, som indeholdes i en række `k`. For at lave funktionen, laver vi en funktion `func` som bruger `match-with` som returnere `TRUE` når feltets y-koordinater er lig med `k`, og ellers `FALSE`. Herefter bruger vi `List.filter` til at returnere alle `pieces` fra `state` som returnere `TRUE` i funktionen `func`.

```
let filter (k:int) (s:state) : state =
    let func (k:int) (p:piece) : bool =
        match p with
            |(c,(i,j)) when j = k -> true
            |_-> false
    List.filter (fun i -> func k i) (s)

```

Figur 4: `filter`

Funktionen `flipUD` tager en `state` og returnere en ny `state`, hvor alle felternes x-værdier er $(2 - x)$. Vi laver en funktion `func` som bruger `match-with` og tager en `piece` og returnere en ny `piece`, hvor dens x-koordinat er $(2 - x)$. Herefter bruger vi `List.map` til at tager elementerne fra `state` og udfører `func` på hvert element og returnere den nye `state`.
En overvejelse til løsningen er at bruge `List.map` og returnere en ny liste, hvor $(2-x)$ uden brug af `func`. Da vi allerede havde lavet `flipUD` med `func` funktionen, valgte vi at beholde den.

```
let flipUD (s:state) : state =
    let func (p:piece) : piece =
        match p with
            |(c,(i,j)) -> (c,(2-i,j))
    List.map func s

```

Figur 5: `flipUD`

Funktionen `transpose` tager en state og returnerer en state, hvor x og y koordinaterne er byttet således: `[i,j] -> [j,i]`. For at lave funktionen, laver vi en hjælpefunktion `func`. Hjælpefunktionen bruger `match-with` og tager koordinaterne og bytter dem rundt. Herefter bruger vi `List.map` til at lave en ny liste, hvor alle koordinaterne er byttet.
En overvejelse af løsning var at lavet en `List.map`, som tog hvert element og byttede x og y koordinaterne, uden brug af hjælpefunktionen. Grundet samme argument fra `flipUD`, valgte vi at beholde den.

```
let transpose (s:state) : state =
    let func (p:piece) : piece =
        match p with
            |(c,(i,j)) -> (c,(j,i))
    List.map func s

```

Figur 6: `transpose`

Funktionen `empty` tager en `state` og returnere en pos liste. Funktionen skal sørge for at holde overblik over alle tomme positioner på brættet. For at lave funktionen, definerer vi en liste `lst` som indeholder alle de mulige placeringer. Listen `lst` er lavet med `List.allPairs` for at undgå tastefejl, hvis vi skrev alle placeringerne manuelt. Dernæst laver vi endnu en ny liste `toPos` ved hjælp af `List.map` som tager `state` og returnere en liste med x og y koordinaterne i en `pos list`. Nu har vi to `pos list` og vi bruger `List.except` som tager de to lister og returnere en liste, hvor `toPos` er fjernet fra `lst`.

```
let empty (s:state) : pos list =
    let lst = [0..2] |> List.allPairs [0..2]
    let toPos = List.map (fun (c,(i,j)) -> (i,j)) s
    lst |> List.except toPos

```

Figur 7: `empty`

Funktionen `addRandom` tager en `value` og en `state`, og returnere en `state option`. For at lave funktionen, anvender vi en `match-with`, hvor længden af `empty s` matches således, hvis `(empty s).Length`=0 returnerers `None`. Ellers får vi `Some state` concatenated med en tilfældig placeret ved brug af `empty` funktionen. For at returnere en tilfældigt tom placering, tager vi `empty s` og returnere et tilfældigt indeks fra 0 til længden af `empty s`, i `empty s`.

```
let addRandom (c:value) (s:state) : state option =
    match (empty s).Length with
        |0 -> None
        |_-> Some (s@[c,(empty s)[rnd.Next((empty s).Length)]])

```

Figur 8: `addRandom`

Funktionen `shiftLeft` har som navnet hentyder til formål at rykke hvert felt til venstre. Yderligere bruger vi også denne funktionen til tillade kombinationen af to felter ved overlap. Dette gør vi vha funktionen merge som benytter halerekursion. I denne funktion matcher vi tre tilfælde. Det første tilfælde er hvor `state` er på formen `a::b::rst` og det første `piece`, `a`, og det andet `piece`, b, indeholder den samme farve. Dette er vist ved `fst a = fst b`. I dette tilfælde ændres `state` vha. cons operatoren som binder en ny `piece` som betegnes `nextColor (fst b),snd b)` til funktionskaldet `merge rst`. Bemærk at dette `piece` får en ny farve vha `nextColor` og beholder `pos` fra `b`. Det andet tilfælde vi matcher med er `a::rst` som er tilfældet hvor a ikke har samme farve som andet `piece` i `state`. Her returneres blot `a:: merge rst`. Til sidst matcher vi med `[]` som blot returnerer `[]`. Denne hjælpefunktion benytter vi i den anden hjælpe funktion `shift`. Det ses at vi i denne funktion piper, `|>`, på vores `state`, `s`. Vi piper først med `filter` som filtrerer alle de `pieces` som ikke er i rækken `i`. Derefter sorteret de tilbageværende `pieces` efter x-koordinat med `List.sortBy`. Bemærk at dette er vigtigt fordi ellers kan vi risikere at kombinere to ensfarvede felter, hvor et felt med anden farve er placeret mellem dem på en række. Efter at vi har sikret os, at dette ikke kan lade sig gøre bruger vi nu `merge` som allerede er blevet beskrevet. Tilsidst

bruger vi `List.mapi` som evaluerer hver `piece` i en række og definerer `x` ud fra deres placering. Eks. det 0'te element får x værdi 0 osv.. Vi kan nu mappe over de tre rækker `[0..2]` med `filter`. Dermed har vi tre separate `states` som vi samler ved at pipe med `List.concat`. Dermed er funktionen `shiftLeft` nu færdig. Koden er vist i figur 9.

```
let shiftLeft (s:state): state =
    let rec merge (s:state) =
        match s with
            |a::b::rst when fst a = fst b -> (nextColor (fst b),snd b) :: merge rst
            |a::rst -> a :: merge rst
            |[] -> []
    let shift i =
        s
        |> filter i
        |> List.sortBy (fun (c,(x,y)) -> x)
        |> merge
        |> List.mapi (fun n (c,(x,y)) -> (c,(n,y)))
    List.map shift [0..2] |> List.concat

```

Figur 9: `shiftLeft`

Vi har nu defineret en masse funktioner som det nu er tid til at anvende. Hertil definerer vi to nye funktioner som gør brug af egenskaberne fra de forrige funktioner. Den ene af disse funktioner kalder vi for `draw`. Denne funktion tager et `state`, `s`, og to integer værdier, `w` og `h`, og returnerer et canvas.

```
let draw (w:int) (h:int) (s:state) : canvas =
    let C = Canvas.create w h
    let rec func (s:state) : unit =
        match s with
            [] -> ()
            |(c,(x,y)) :: rst ->
                let vh = (10+(w/3)*x,10+(h/3)*y)
                let hh = ((w/3)*(1+x)-10,(h/3)*(1+y)-10)
                do setFillBox C (fromValue c) vh hh
                func (rst)
    func s
    C
```

Figur 10: `draw`

I denne funktion benyttes en rekursiv funktion som gennemgår alle `piece` i `state` vha halerekursion og danner et firkantet felt i canvas med kommandoen `do setFillBox` som tager et canvas, som vi har betegnet `C`, en farve `fromValue`. Bemærk at denne farve skal være en canvas farve, og at vi derfor er nødsaget til at bruge funktionen `fromValue` på c. Yderligere definerer vi øvre venstre hjørne som `(10+(w/3)*x,10+(h/3)*y)` og nedre højre hjørne som `((w/3)*(1+x)-10,(h/3)*(1+y)-10)` i kommandoen. Således tegnes alle.
Note: Det er ikke beskrevet i opgaven, at felterne skal have et mellemrum. Under arbejds processen valgte vi at lave et lille mellemrum mellem felterne, da det var mere overskueligt og se farverne bevæge og kombinere sig. Derfor valgte vi at beholde mellemrummene i koden. Mellemrumme er tilføjet ved blot at lægge 10 til på x og y i første koordinat og trække 10 fra på x og y i andet koordinat.

Den anden funktion er `react`. Denne funktion tager et også en state og en key som er et tastatur tryk. Funktionen skal altså ændre opsætningen af vores pieces, på boardet, afhængig af hvilken piletast der er trykket.

```
let react (s:state) (k:key) : state option =
    match getKey k with
        |LeftArrow ->
            addRandom Red (shiftLeft s)
        | RightArrow ->
             addRandom Red (flipUD s |> shiftLeft |> flipUD)
        | UpArrow ->
            addRandom Red (transpose s |> shiftLeft |> transpose)
        | DownArrow ->
            addRandom Red (transpose s |> flipUD |> shiftLeft |> flipUD |> transpose)
```

Figur 11: `react`

Som vi kan se har vi defineret bevægelserne af vores `piece` ved brug af vores tidligere defineret funktioner. Første bevægelse `LeftArrow`, har vi allerede allerede funktionen `shiftLeft` som rykker alle `piece` mod venstre og den bruges herfra. Til bevægelsen mod højre `RightArrow`, bruger vi `flipUD -> shiftLeft -> flipUD` beskrevet i opgavebeskrivelsen. For at lave en op, `UpArrow`, bevægelse, bruger vi rækkefølgen: `transpose -> shiftLeft -> transpose`. Sidste bevægelse ned, `DownArrow`, bruger vi `transpose -> flipUd -> shiftLeft -> flipUD -> transpose`.

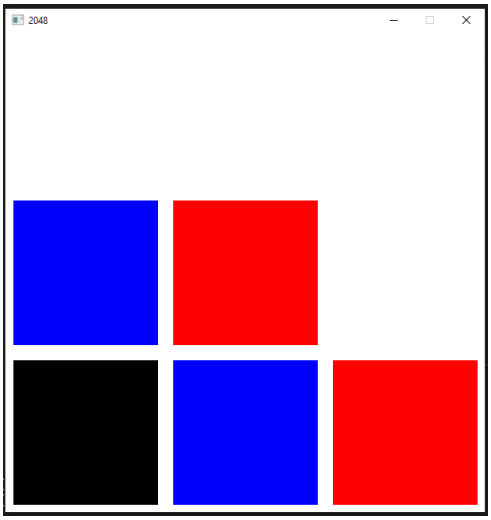
Hvergang at `react` bliver brugt, skal vi simulere en tur og et tilfældig rødt felt skal tilføjes. Det gør vi ved at bruge `addRandom Red state`, hvor `state` er fra bevægelserne.

XML tags

Funktionerne er dokumenteret med dokumentationsstandarden ved brug af $< summary >, < param >$ og $< returns >$ i 6g0Lib.fsi.

1 Test

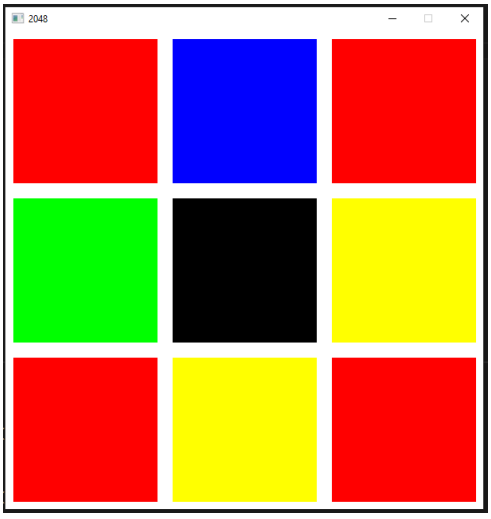
I de første testkørsler af vores koder har vi printet de lister vi får ud, for at sikre os at programmet kører korrekt. Herunder er et eksempel af et board og den tilsvarende `state`.




```
[(Green, (0, 1)); (Green, (1, 1)); (Black, (0, 2)); (Blue, (1, 2))]
[(Blue, (0, 1)); (Red, (1, 1)); (Black, (0, 2)); (Blue, (1, 2)); (Red, (2, 2))]
```

Figur 12: Caption

Vi kan se at ingen af koordinaterne er de samme hvilket må betyde at programmet er funktionelt.
Spillet slutter når brættet er fyldt og ingen felter kan kombineres. Man kunne sagtens have lavet en afsluttende condition der gik ud af spillet eller afsluttede på andenvis. Vi har valgt ikke at inkludere det i vores implementation da man som bruger nemt kan trykke på krydset eller alt +f4. Herunder er et eksempel på et fyldt bræt og den tilsvarende `state`.




```
[(Red, (0, 0)); (Blue, (1, 0)); (Red, (2, 0)); (Yellow, (0, 1)); (Black, (1, 1));
(Red, (0, 0)); (Green, (1, 0)); (Red, (2, 0)); (Yellow, (0, 1));
(Black, (1, 1)); (Blue, (2, 1)); (Red, (0, 2)); (Yellow, (1, 2)); (Red, (2, 2))]
```

Figur 13: Caption