PoP Opgave 7

Hold 1 - Gruppe 7

Aditya Fadhillah (hjg708)

Signe Dueholm Nielsen (rmp985)

Simon Krogh Anderson (cdp934)

30. november 2021

Forord

I denne rapport arbejder vi med at lave et spil. Rapporten er lavet af Aditya Fadhillah, Signe Dueholm Nielsen og Simon Krogh Anderson i forbindelse med 7'ende opgave i kurset: "Programmering og Problemløsning". Opgaven er stillet af Jon Sporring og tæller som den ene af de opgaver, der skal bestås før kurset kan bestås. Vi vil gerne sige tak til Casper og Phillipa vores højtærede TA's, som har hjulpet os igennem de sværre tider. Det havde ikke været muligt uden jer.

Introduktion og Problemformulering

I denne opgave programmeres et spil kaldet "Rotate", spillet går ud på at der printes et "Board"som består af en kvadratisk tabel af bogstaver, med enten 2, 3, 4 eller 5 som sidelængde. Dette printes som følgende:

2 x 2		**3 x 3**			**4 x 4**				**5 x 5**				
a	b	a	b	c	a	b	c	d	a	b	c	d	e
c	d	d	e	f	e	f	g	h	f	g	h	i	j
		g	h	i	i	j	k	l	k	l	m	n	o
					m	n	o	p	p	q	r	s	t
									u	v	w	x	y

Når man har valgt et board, bliver boardet scramblet med den sværhedsgrad man vælger via keybordet. Nu er opgaven at dreje et 2 x 2 matrix, så hele bordet kommer tilbage i sin "solvede"stand. Når boardet er tilbage til originalen printer programmet til skærmen, at man har vundet.

I programmet bruges blandt andet rekursive funktioner, til at styre vores loops, og der bruges lister til at holde styr på boardet. Programmet bruger ikke mutables, men laver derimod nye elementer/lister, hver gang der skal genereres et output.

Det færdige produkt er et færdigt spil, der kan kompileres og køres i terminalen. Dette med fuldt "gameplay"og introduktion til spilles regler osv.

Problemanalyse og design

De indledende funktioner og deres typer er givet på forhånd. Herunder er en kort beskrivelse af vores tanker til hver funktion, i flere af funktionerne har vi brugt n, som er defineret som kvadratroden af længden af vores board.

Funktionen create

Vores create-funktion bruger pattern-matching til at lave vores board. Den oprindelige tanke var at lave et "case"til hvert "n", og så prædefinere hvilken char-list, der skal outputtes. På denne måde kunne vi også sikre os, at hvis inputtet er mindre end 2 bliver boardet et 2 x 2 board. Hvis inputtet er mere end 5 bliver boardet et 5 x 5 board. Denne tanke kommer fra, at en potentiel spiller, der ikke forstår boardets grænser, stadig vil ønske at spille. Alternativt kunne programmet komme med en fejlmeddelelse, men vi foretrak den første løsning.

Vi endte dog med at bruge en enkelt linje som bruger indexering, til at hive de rigtige elementer ud.

```
['a'..'y'].[..(int n * int n)-1]
```

Funktionen board2Str

I denne funktion var tanken hele tiden af fjerne det første element i vores liste, og derefter køre funktionen rekursivt på halen. Vi bruger her n i et match-case til at vælge, hvornår vi skal printe et linjeskift, så vi får printet en pæn tabel. Vi bruger en hjælpefunktion for at sørge for, at vi beholder vores n.

Funktionen validRotation

I validRotation tager vi udgangspunkt i n når vi skal undersøge om vores rotation er valid. Vi laver nogle forskellige match-cases, hvor vi tjekker om inputtet går op i n, hvis det gør, bliver den selvfølgelig false da alle position der går op i n må ligge længst til hørjre. Desuden tjekker vi også når p er i den sidste række, som jo må være længden af listen minus n.

Funktionen rotate

Funktionen rotate tager et board og et integer, der angiver placering af et element. Vi bruger n for at hjælpe med at specificere placering af elementerne. Desuden laver vi placeringen om fra et et-indekseret integer, til et nul-indekseret integer. Vi laver to if betingelser i funktionen. Den første; hvis placeringen er mindre end 0, udlevereres et uændret board. Hvis validRotation b p evaluerer false, printer den også et uændret board. Ellers roterer en 2x2 firkant på boardet, så den roterer med uret en plads.
Dvs. hvis $b = [\prime a\prime;\prime b\prime;\prime c\prime;\prime d\prime]$ vil *rotate* b 1 blive til $[\prime c\prime;\prime a\prime;\prime d\prime;\prime b\prime]$

Funktionen scramble

Vores Scramble tager et board og et heltal m. Den genererer herefter et tilfældigt tal x, som skal bruges som posisition. Herefter bruger vi et match-expression til enten at køre rotate på det valgte punkt, eller til at vælge et nyt punkt gennem et rekursivt kald. Den trækker kun 1 fra m, hvis punktet er validt. Derved sikre vi, at funktionen laver præcis m tilfældige rotationer.

Funktionen solved

I funktionen "solved"bruger vi pattern matching til at vurdere om boardet er "solved"eller "unsolved". Vi tager udgangspunkt i boardet og sammenligner med hjælp fra vores "create"funktion til at se, om boardet er lig "create"med et n defineret ud fra længden af boardet. Er Boardet det, da må det være løst og funktionen returnere true. Ellers er boardet ikke "solved"og den vil returnere false.

Program beskrivelse

Programmet køres via kommandolinjen, og starter med at printe nogle regler og grundprincipper for spillet.

For at implementere hele opgaven har vi brugt bibliotek-funktionen "."som man bruger til at hente funktioner fra et bibliotek. Herefter venter programmet på, at spilleren vil indtaste et tal der bestemmer størelsen på boardet. I denne linje bruger vi funktionen på denne måde "rotate.create"Der henter de implementerede funktioner inde fra biblioteket "rotate.dll"Derefter skal spillets sværhedsgrad defineres, også af brugeren. Jo flere tilfældige rotationer, jo sværre niveau. Dette gøres også på samme måde med brug af ".-notation. Hvis der ønskes mere information om implementation af opgaven henvises til "problemanalyse og design"samt kommentare i selve koden.

Nu begynder spillet, og der bruges System.Console.ReadLine, til at lade brugeren vælge hvilke 2 x 2 felter der skal roteres. Når/hvis boardet bliver roteret tilbage til det originale board vil programmet printe til skærmen, at man har løst boardet og vundet spillet.

Programmet game er bygget op omkring en rekursiv funktion med match-cases.

```
let rec gamePlay (b: Board) : Board =
    match b with
        | b when solved b -> b
        | _ -> gamePlay (rotate b (int(System.Console.ReadLine())))
```

Hvis boardet er i sin "solvede"stand, vil funktionen returnere dette board, ellers vil brugeren få muligheden for at rotere boardet med et kald til funktionen Rotate.

Afprøvning og eksperimenter

For at teste at vores spil virker efter hensigten, har vi udført en række af tests. Disse er blevet udført i henhold til standarten for hhv. black- og white-box test. Derudover er der udført test-kørsler på 3. parter, for at kontrollere brugervenligheden.

Demo-version afprøvet på forsøgspersoner

Her gav vi en 3. part muligheden for at prøvekøre spillet et par gange. Test-personen havde ingen kendskab til kommandolinjen, men formåede dog at gennemgå spillet med minimal interaktion fra en udvikler.

Under forsøget blev der gjort en række observationer, heriblandt er den mest afgørende at der skulle finpusses på formuleringerne i spillets regler og muligheder. Dernæst var det tydeligt, at der på daværende tidspunkt ikke var gjort så meget for at "fejl-sikre"programmet, man kunne derfor få det til at crashe, hvis man kom med ugyldige input. Disse kunne f.eks være hvis man taster noget andet end et heltal som input, eller trykker enter inden man har tastet noget.

White-Box

Her laver vi tests for hver mulig gren og forgreningsvej i vores kode. Der laves et program, hvor vi tester alle vores funktioner op mod et forventet input. Da vi kun køre ned af de grene vi kender, tester vi her ikke for ukorekte input, dette gøres derimod i vores black-box test. Ydderligere kommentarer til de enkelte tests kan findes i selve white-box-filen.

Black-Box

Her kaster vi alt, hvad vi har af gode(og dårlige) bud på input mod vores funktioner. Dette samles i et program, der køres efter at være blevet compiled med vores bibliotek rotate med vores funktioner.

Det vil sige, at vi tager de "normale"input, og derefter både store og små input. Der hvor det giver mening, prøver vi også med input, der er ugyldige. Det der menes her, at funktioner som validRotation og solved ikke kan tilgås direkte af brugeren, derfor giver det ikke mening at teste for "forkerte"input. Ved at teste de funktioner som brugeren har adgang til, sikre vi os at elementer der kommer igennem de funktioner, naturligvis også må være gyldige i de underlæggende funktioner. I selve blackbox-filen, er der givet uddybende kommentarer på de enkelte tests.

Diskussion og Konklusion

Som afslutning gennemgås herunder nogle af de udfordringer, overordnede tanker og løsninger der er dukket op under arbejdet med Rotate.

Diskussion

Opgaven med at lave spillet rotate kunne løse på mange forskellige måder, selvom der var givet faste retningslinjer for både typerne til vores forskellige funktioner, og antallet af funktioner. Dette viser godt, hvordan arbejdet med programmering helt generelt udfolder sig. I denne opgave, er der ikke lagt vægt på køretidsanalyse. Det er ellers i erhvervslivet tit et af de mest presserende punkter, da alt handler om optimering af tid. Derfor er der i denne opgave, lagt mere vægt på at funktionerne skal være overskuelige, og nemme at gennemskue - det skal altså være let at se, hvilke tanker der ligger bag de enkelte funktioner. Dette ses blandt andet i tidligere beskrevede create-funktion, hvor den kan laves med enten 6 eller 1 match-statement. Her er det meget lettere at gennemskue, hvad funktionen gør, men man kan forestille sig at køretiden må være længere i den funktion, hvor vi skal sammenligne med alle linjerne for at finde den rigtige linje. Det afhænger dog også af, hvor efektiv F-sharp er til at slice lister.

På samme måde har vi også i vores funktion ValidRatation haft flere forskellige tanker om implementeringen. Først ville vi bruge operatoren "mod", til at tjekke hvornår p var indeholdt i n-tabellen. Men vi har også her valgt at skrive det ud som flere linjer for overskueligheden. Dette valg bunder også i, at vores spil er 1-indekseret, men listerne er derimod 0-indekseret. Derfor skal vi trække 1 fra, hvilket ville gøre funktionen ugennemskuelig. Vi har lagt meget vægt på at brugervenligheden skal være høj. Derfor har vi også brugt lang tid på at vores funktioner så vidt muligt "failer gracefully". Det vil sige at hvis brugeren f.eks indtaster 0 eller negative tal vil programmet ikke chrashe, men derimod bare bede om et nyt tal. Denne funktion har vi dog ikke fået implementeret hvis brugeren enten indtaster bogstaver, eller trykker enter uden at taste noget. Dette er dog et mindre problem i praksis da det er beskrevet i introduktionen til spillet at man skal indtaste heltal, så taster man andet, er man selv udenom at det chrasher.

Konklusion

Konkluderende kan man sige at vores funktioner løser opgaven, vores spil er et færdigt produkt som kan køres i kommandolinjen som beskrevet i problemformuleringen. De udfordringer vi stødte på har vi fundet løsninger på. Spillet er gennemtestet både med black- og white-box og brugertests, og det må derfor menes at være et fungerende spil.

Opgave 5

For at funktionen finder en løsning til boardet, vil den være nødt til i første omgang at forsøge, at finde én løsning. Det er dog ikke sikkert at denne løsning overholder m. Derfor må den gemme listen, og forsøge at finde en bedre løsning, som forhåbentlig overholder m. Hvis der ikke findes en løsning, der er kort nok vil den blive ved med at tilføje til stakken i hver iteration, og vil derfor til sidst løbe tør for hukommelse. Dette sker, da man har bedt den om at finde en løsning, der har mindre end eller lig med m træk.

Derudover er det ikke sikkert, at computeren har fundet den bedste løsning, da den blot skal være kortere end m. Der kan derfor potientielt være en løsning som er bedre.