POP 7g Rapport

Ida Helene Jensen (xdr622)
Julian Philippe Pedersen (rsk975)
Kjartan Martin Johannesen (vnb683)

November 2020

1 Indledning

Denne opgave satte sig for at genskabe det kendte kortspil Krig i Fsharp. Formålet er at få computeren til at spille kortspillet mod sig selv gennem en implementation i Fsharp. Her har vi skulle implementere spillet i Fsharp fra bunden. Dette betyder at der både har skulle laves en blanding-funktion samt en fordelings-funktion. Ydermere skulle selve spillet og dets regler oversættes til noget der kunne bruges i Fsharp. Til sidst er der blevet lavet statistik på computerens spil mod sig selv, som har til formål at vise win-ratioen mellem spiller 1 og 2, samt give et gennemsnit på udspil per spil.
Problemformulering: *Hvordan kan man rekursivt implementere kortspillet Krig i Fsharp, og hvordan ville win-ratioen se ud efter 10000 spil?*

2 Problemanalyse og Design

Vi startede vores kode ud med at definere nogle typer som vi kort vil redegøre for. Typerne havde til formål at gøre koden mere overskuelig. Først lavede vi en type "card"som er en integer. Dernæst lavede vi typen "deck"som er defineret som en card-list, altså en integer list. Senere oprettet vi en ny type "player"som er et dæk. Typen player er brugt i selve game-funktionen mens shuffle, deal og newdeck bruger typerne deck og card.
Vores første opgave gik ud på at lave tre forskellige funktioner. En funktion der dealer de to spillere deres kort, og lave en funktion der blander kortene før de gives ud. Til sidst skulle der laves en funktion der opretter et standard kortdæk.
Ved vores deal-funktion var kravene at den kunne dele et dæk op i to mindre dæk, hvor de øverste kort i det originale kortdæk blev de nederste i de nye.

Derudover skulle funktionen deale et kort af gange skiftevis til hver spiller. Til at lave denne funktion gjorde vi brug af stak-logikken, da denne stemte overens med grundideén bag deal-funktionen. Stakke kører efter princippet "last in, first out", dermed ville de øverste kort i vores dæk netop blive de nederste kort i vores to stakke. Vi brugte dermed logikken bag PUSH, som er et begreb der bruges indenfor stakke, til at skiftevis at give hver spiller et kort fra dækket. Vi valgte at gøre denne funktion rekursiv, da dette gjorde selve gennemløbning af dækket lettere og hurtigere.

Shuffle-funktion Krav til denne funktion var blot at denne skulle bruge den givne funktion rand. Rand er en funktion som tilfældigt vælger et tal mellem 0 og n, hvor n er input. Til at oprette shuffle-funktionen valgte vi at bruge elementer fra Fsharps indbyggede List-module. Dette gjorde vi da vi fandt frem til at List-modulet har nogle funktioner som kan gøre blanding lettere, som gøre koden mere kort og koncis. Vi overvejet også at bruge et matematisk formel for shuffle, og ikke lade den være random, da det kunne garantere en såkaldt perfekt shuffle. Dog ville dette betyde at dækket tit ville være ens og dermed ville spillet blive mere forudsigeligt, hvilket ikke er ønskeligt i denne opgave. I vores shuffle-funktion kunne man have brugt piping, for at gøre koden mere elegant og slippe for overflødige linjer kode. Vores shuffle-funktion er ikke universel, da vi i vores implementation valgte at sætte maks indeks til 52. Dette vil blive beskrevet nøjere i programbeskrivelsen. Dette har dog den effekt at selv shuffle-funktion er mindre generel end man kunne have ønsket.

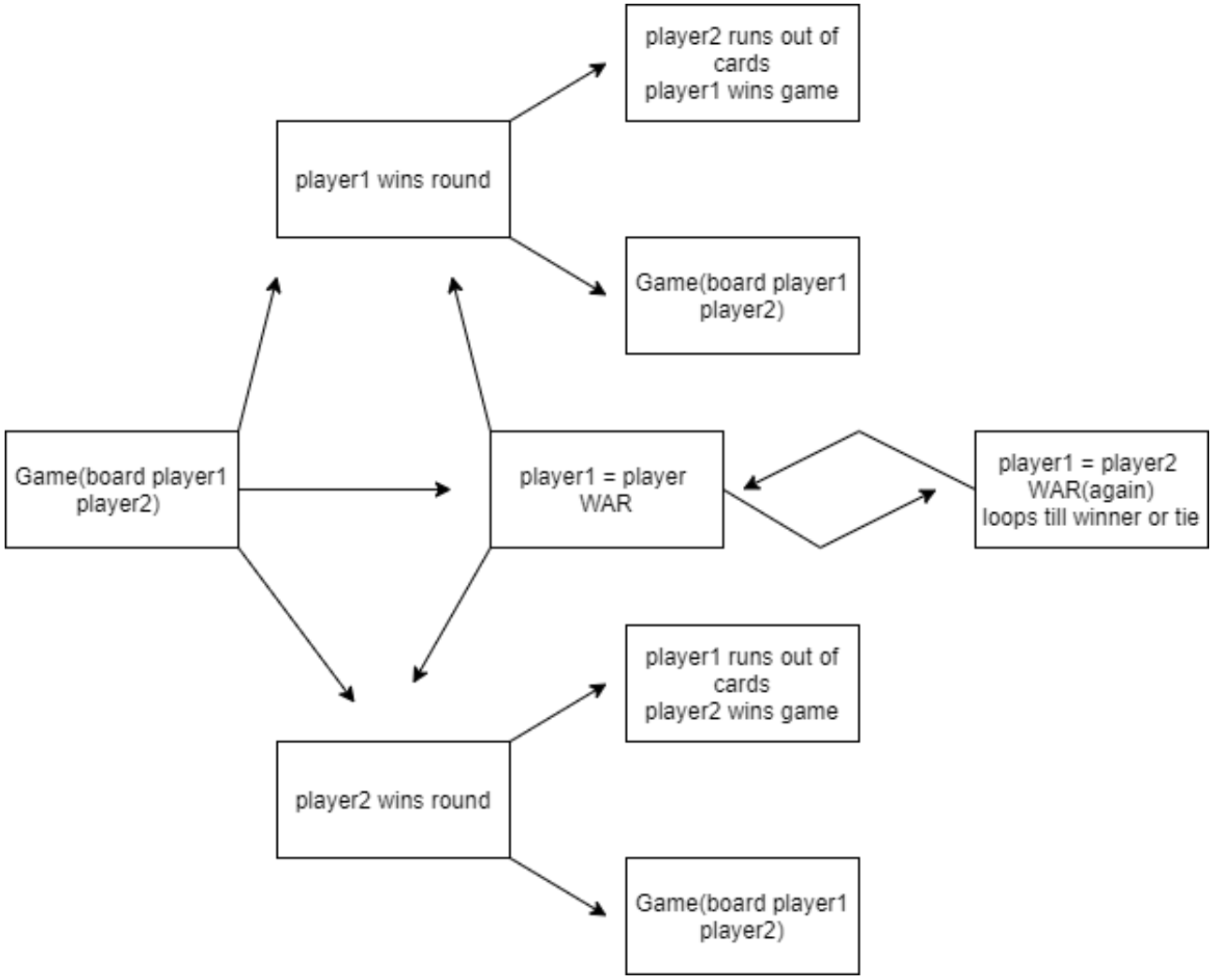
Vores sidste funktion til at opsætte selve kortdækket der skal spilles med er newdeck. Kravet til denne funktion var at den skulle oprettes et standard kortdæk hvor der findes fire serier af kort fra 2-14. For at opnå dette valgte vi at den letteste løsning ville være at lave en liste indeni funktionen der indeholdte tallene fra 2 til 14. Denne liste appendede vi med sig selv fire gange så vi netop endte med fire serier af tallene 2-14. Vi synes at denne metode var et fint udgangspunkt til at lave funktionen, da vi alligevel bruger shuffle-funktionen indeni newdeck. Dette vil sikre at selvom vores startdæk altid er det samme vil outputtet med lav sandsynlighed være ens. Her kunne man i stedet have lavet en mere optimeret løsning med pattern-matching samt gøre løsningen mere universel ved at kunne definere andre antal kort kortdækket med 52 kort. Der kunne altså havde været en smartere løsning ift. test af funktionen der gjorde at vi ikke altid skal teste newdeck med 52 kort. Dog valgte vi stadig at bruge den lette løsning, da vi indså at mængden af arbejde for at lave en anden implementation overgik fordelene ved at kunne lave mere generel kode, hvert fald i dette projekt. Den er hard-coded til vores ene case. Vores implementation gør dog funktionen let-læselig.

Nu når vi har implementeret et blandet kortdæk, vil vi nu gennemgå imple-

mentationen af selve gameplayet og funktioner rundt om. Til at starte med skulle vi lave en funktion der trækker det øverste kort fra hver stak, der returnere kortet samt resten af stakken. Denne funktion havde det benspænd at den skulle returnerer en option-type. Getcard bruger logikken bag POP, hvor vi tager hovedet fra listen og returnere dette sammen med halen af listen i form af to elementer i en tuple. Hvis spilleren løber tør for kort ved korttrækket vil funktionen returnere None. Her har vi valgt at bruge pattern-matching frem for if-statements, for at gøre koden mere elegant. Vores getCard bruger option-type, dette fandt vi smart fordi vi kun er interesseret i to tilfælde, tom eller ikke tom. Checket bliver derfor mere spilet, da vi kun skal tjekke om det er Some eller None. Hvis man ikke havde option-type skulle vi i game skrive de præcise resultater af hver kørsel af funktionen.

Addcards Denne funktion er endnu en hjælpefunktion som har til formål at tage de vundne kort og lægge dem nederst i den vunde spillers stak. Til at gøre dette har vi valgt at blande de vunde kort og derefter appende dem på decket. Her er princippet det samme som en kø da vi appender decket til boardet, så boardet kommer bagerest i listen og ikke forrest. Igen kunne man have brugt piping i vores implementation for at undgå overflødige variable.

Vi vil til sidst gennemgå game-funktionen, som er her hvor selve spillet er blevet implementeret. Understående billede viser vores programdesign ved vores funktion game. Flowchartet har til formål at vise hvad der sker i hvert tilfælde af udspil, altså hvad vores program skal være opmærksom på. Derudover viser diagrammet også hvordan vores program skal virke rekursivt. Ved hvert udspil er der tre muligheder, player1 kan vinde runden, player2 kan vinde runden eller der kan være krig. Hvis player1 vinder runden kan der være to udfald, enten er player2 løbet tør for kort ellers skal der laves et nyt udspil, altså spilles en ny runde. Det samme gælder hvis player2 vinder runden. Ved krig kan der være 5 udfald (dog er to af dem med vilje ikke med i figuren for overskuelighedens skyld). Det første tilfælde er at en af spillerne løber tør for kort når kortet med billedsiden ned af skal udspilles, dette vil gøre at den anden spiller vinder før krigen overhovedet af afgjort. Dernæst kan begge spillere have kort nok til at komme igennem tilsidelægningsfasen og her ville funktionen rekursivt kalde sig selv hvor et nyt udspil vil blive udført, dog med de endnu ikke vundne kort med som argument. Det sidste udfald i krig er at der igen kommer krig ved næste korttræk. Her vil det i meget usandsynlige tilfælde ende med et loop hvor begge spillere trækker kort til der enten er en der vinder eller spillet er uafgjort.



Det eneste krav til game-funktionen var at den var rekursiv. Da vi startede vores implemebtation valgte vi at skrive den ved brug af if-statements, vi tog altså hver case set i figuren ovenfor og satte det op som if-else-statements. Sådan at efter hvert korttræk tjekkede vi om nogle af spillerne havde vundet eller om funktionen skulle kalde sig selv igen. Vi endte med en meget lang kodeblok med en masse if-statement inden i hinanden. Programmet virkede fint dog mente vi at vi godt kunne optimere koden, så den brugte pattern matching og var mere funktionel. Vi valgte at lave koden om til at gøre brug pattern matching, fordi pattern matching er mere pålidlig og overskuelig. Vores originale kode var meget lang og uoverskulig med de mange if-statement. Gennem pattern matching er vores kode blevet mere elegant og koncis, dette har skabt en mere overskuelig kode, hvor det er lettere at finde rundt i logikken.

3 Programbeskrivelse

I dette afsnit har vi valgt særligt at se på game-funktionen, da den er den mest spændende for opgavens besvarelse, dog vil vi starte med kort at gennemgå test af shuffle-funktionen, newdeck, deal-funktionen samt addcards og getcard. Som gennemgået i problemanalysen valgte vi at gå fra if-statement til pattern matching. Vi vil nu gennemgå hvordan vi har implementeret game-funktionen ved hjælp af pattern matching og hvordan vi konkret har gjort koden mere overskuelig og elegant.
Test af shuffle-funktionen og newdeck():

```
Shuffle and newdeck: [14; 4; 12; 3; 10; 3; 14; 9; 9; 10; 10; 5; 7; 11; 2; 13; 13; 4; 6; 2; 8; 9; 11;
 7; 5; 2; 8; 7; 4; 3; 5; 7; 12; 5; 6; 6; 12; 8; 11; 11; 14; 13; 13; 14; 3; 9; 12;
 6; 4; 10; 2; 8]
```

Som det kan ses på den ovenstående figur returnere vores funktion newdeck() et dæk med 52 der overholder reglerne for et almindeligt kortspil. Altså der er 4 af hver tal fra 2-14. Derudover ses det også at newdeck() selv blander tallene, da vi har kaldt shuffle-funktionen internt i newdeck()
ift. vores Shuffle-funktion valgte vi at bruge tre funktioner fra List-modulet i vores funktion `shuffle`. Først gjorde vi brug af `List.mapi` som gør indices eksplicitte som en ny liste. Dermed går funktionen fra at have et deck til at få typen deck*deck. Inden i den anonyme funktion i `List.mapi`, mapper vi indeks til et nyt random tal mellem 0 og 52, mens vi lader vores originale liste (kortdækket) være det samme i.f.t. værdier. 52 var valgt da vi ikke kommer til at blande med flere kort end 52, så der er ikke behov for højere indeks. Vi valgte at ændre indeks i blanding, da dette ville gøre at sikker på at selve talværdier i dækket blev ændret, blot skifter plads. Det gør at vi ikke skal være bekymret for at der kommer fejl, ved at et tal fx antager værdien 33. Dernæst bruger vi `List.sortby` til at sortere indekserne, så de efterfølgende står i sorteret rækkefølge. Dette er hvad der reelt blander kortene, da listen med kortene skifter plads som deres indeks gør. Til sidst ønsker vi kun at returnere dækket i tuplen vi lavede og ikke længere have indeks eksplicitte, til dette bruger vi `List.map`. **Test af deal-funktionen:** I denne test oprettet vi et kortdøk ved brug af newdeck() og derefter kørte vi deal på dækket. Ud af dette fik vi de to nedenstående stakke.

```
deal function: ([2; 4; 12; 3; 13; 14; 11; 12; 6; 12; 5; 4; 8; 5; 11; 8; 6; 13; 2; 7; 10; 9; 14;
  10; 12; 14],
 [8; 10; 6; 9; 14; 13; 11; 8; 6; 5; 7; 3; 7; 2; 7; 9; 2; 4; 13; 11; 5; 10; 9; 3;
  3; 4])
```

Funktionen retunere en tuple med to deck-typer i sig, som senere i selve spillet bliver player1 og player2. Deal sørger for at bruge "last in, first out"konceptet,

hvilket kan ses ved at det orignale newdeck() startede med 14,4 og player1's dæk slutter på 14 mens player2's dæk slutter på 4.
Test af getCard og addCards: Når getCard bliver kaldt bliver først kort i dækket trukket, derfor returnere getCard en option-type bestående af en int og en liste. Dette ses på nedenstående test hvor det første kort i dækket er tager ud of returneret som int.

```
getcard: Some
  (2,
   [4; 12; 3; 13; 14; 11; 12; 6; 12; 5; 4; 8; 5; 11; 8; 6; 13; 2; 7; 10; 9; 14;
    10; 12; 14])
```

Ved addCards testede vi ved at tilføje kortene [1;5;7], ved addCards skal de tilføjede kort blandes og placeres tilsidst i dækket - netop for at sørge for det ikke altid er de samme kort der er i spil. Testen af addCards ses her:

```
add cards: [2; 4; 12; 3; 13; 14; 11; 12; 6; 12; 5; 4; 8; 5; 11; 8; 6; 13; 2; 7; 10; 9; 14;
 10; 12; 14; 1; 5; 7]
```

Gennemgang af game-funktionen

Vores rekursive game-funktioner tager fire argumenter, nemlig board, som er kortene der kan vindes efter hver runde, player1, den første spiller, player2, den anden spiller og acc, en form for counter der tæller hver udspil. game-funktionen returnerer efter udvidelsen ift. statistik en tuple af int*int. Det første element i tuplen repræsenterer vinderen, og kan derfor kun returnere -1,0,1,2. 1 betyder player1 har vundet spillet, 2 betyder player2 har vundet spillet, 0 betyder spillet er uafgjort og -1 returneres kun hvis noget er gået galt i spillet.
Vores implementation starter altid ud med at trække et kort fra stakkene idet game er kaldt. Dernæst bliver der matchet på de forskellige outcomes ved at trække et kort. Det første koden gør er at tjekke om nogle af spillerne er løbet tør for kort. Dette tjekkes før vi overhovedet går videre til hvem der vundet, dette gøres for at undgå fejl hvis det sidste kort hos den ene spiller nu skulle gå i krig. Dette ville ikke være muligt for en spiller med 0 kort i deres dæk at gå i krig, derfor bliver vi nød til at tjekke om der er en af spillerne der har vundet.

```
let rec game (board:deck)(player1:player)(player2:player)(acc:int):int*int =
    match getCard player1, getCard player2 with
        | None, None -> 0, acc
        | _, None -> 1, acc
        | None, _ -> 2, acc
```

Som set i kodeblokken returnerer koden 0, hvis begge løber tør for kort samtidig. Hvis player1 har et kort, men player2 løber tør returnerer den 1. Hvis player2 har et kort, men player1 er løbet tør returnerer den 2. Det samme gør sig gældende i vores tredje virkefelt, hvor vi igen er nød til at tjekke om der er en vinder når et kort er kastet med billedesiden nedaf. Ellers ville vi få en fejl. Derudover definerer vi den lokale variable cardPile før vi går ned i næste pattern matching. cardPile er en variable der appender de to trukne kort til board, som nu er det der kan vindes efter rundes afslutning. Efter vi har tjekket om begge spillerne har kort til at fortsætte, matcher vi på vores tre cases, som også ses i programdesignet. Nemlig om player1 har vundet runden, om player2 har vundet runden eller om der er krig.

```
...INSERT...
match p1Card, p2Card with
    | p1Card, p2card when p1Card > p2Card ->
        game [] (addCards cardPile p1Deck) p2Deck
```

I det ovenstående kodestykke ses at hvis player1 vinder runden vil vi eksekvere funktionen addCards som gennemgået tidligere, samt kalde game rekursivt.

4 Afprøvning og Statistik

TEST-funktion introduceres
For at lave den statistik vi gennemgår om lidt skulle vi først oprette en test-funktion som målte på de relevante parametre. Vores test-funktion tager et argument, n, som er et heltal. Dette tal afgør hvor mange gange game-funktionen skal køres. For hver kørsel lægges udspil per spiller sammen for hver game. Denne total deles med n til sidst i koden for at finde gennemsnitlig udspil per spil. Derudover opdateres counteren hver gang player1 eller player2 vinder eller i tilfælde af uafgjort. Test-funktionen returnerer en string hvor resultaterne er beskrevet. Vi har også valgt at implemtere en timer, der viser hvor lang tid det tager at spille 10000 spil af krig. Dette er mere et nice-to-know end et need-to-know, men gav os et ide ift. hvilken game-implementation var hurtigest da vi skulle vælge mellem if-statements og pattern-matching.
Afprøvning Nedenstående er vores resultat efter en tilfældig kørsel af game med 10000 gentagelser.

```
Wins player1: 4661, wins player2: 5339, ties: 0, average: 526
```

Som vores statistik viser er chancen for at vinde næsten 50/50 efter 10000 spil. Mere præcist er den i det her tilfælde 46.61% og 53.39%. Ydermere har vi fundet frem til det gennemsnit udspil per spiller ved 10000 spil er 526 udspil per spiller.
Det ses på vores kørsel at der ikke var en eneste uafgjort. Det virker også sandsynligt, da chancen for at få en uafgjort er ekstremt lille. For at få et uafgjort spil ville spillerne skulle få krig 13 gange i træk. Det vil sige et hver andet udspil skulle give krig. (Ikke hvert udspil da man smider et kort med billedesiden ned hver gang krig forkommer).

5 Konklusion

Vores problemstilling søgte at besvare hvordan kortspillet krig kunne implemteres rekursivt i Fsharp, samt hvordan win-ratio ville se ud efter 10000 spil. Til den første del af problemstilling fandt vi frem til at der skulle implementeres forskellige funktion der havde til formål at oprette kortdæk, blande og dele dem. Vi fik lavet en relativ hardcorded implementation af dækket, da shuffle-funktioen ikke kan håndtere at shuffle mere end 52 tal og newdeck() ikke kan indholde andet end 52 tal, 4 serier af 2-14. Vores deal-funktion er på den anden side ret generel, da dens eneste 'restriktion' er at den trækker på stak-logikken. Dernæst skulle vi implementere hjælpe-funktionerne getCard og addCards. Vores getCard bruger option-type, dette fandt vi smart fordi vi kun er interesseret i to tilfælde, tom eller ikke tom. Vi har game-funktionen som tager fire argumenter, nemlig board, som er kortene der kan vindes efter hver runde, player1, den første spiller, player2, den anden spiller og acc, en form for counter der tæller hver udspil. game-funktionen returnerer efter udvidelsen ift. statistik en tuple af int*int. Det første element i tuplen repræsenterer vinderen, og kan derfor kun returnere -1,0,1,2. 1 betyder player1 har vundet spillet, 2 betyder player2 har vundet spillet, 0 betyder spillet er uafgjort og -1 returneres kun hvis noget er gået galt i spillet. Det andet element i tuplen er antal runder spillet i et spil krig. Vi valgte at bruge pattern-matching frem for if-statements i vores game-implementation, da det er mere elegant.
Vores anden del af problemstillingen søgte at se på statestikken på spillet krig. Har vi en win-ratio på omkring 47/53. Hvilket er en meget pæn ratio. Vi fik ngen uafgjort, dog er dette også meget usandsynligt, hvilket vi gennemgår hvorfor.