Learning to Program with F#
Exercises
Department of Computer Science
University of Copenhagen

Jon Sporring, Martin Elsman, Torben Mogensen, Christina Lioma

October 10, 2020

0.1 warGame

0.1.1 Lærervejledningn

Emne Classes, Objects, Methods Attributes

Sværhedsgrad Middel

0.1.2 Introduktion

Spillet krig (warGame) er et kortspil for to personer. Et almindeligt spil kort med 52 kort blandes og deles mellem de to spillere. Hver spiller holder sin bunke kort i hånden med billedsiden nedaf. Spillerne spiller nu hver et kort ud på bordet fra toppen af deres bunke. Spilleren der spiller det højeste kort vinder kortene på bordet og disse kort placeres i bunden af vinderens bunke. I tilfælde af at der spilles to ens kort (f.eks. to 7'ere) er der krig.

Hver spiller spiller nu et ekstra kort ud på bordet fra toppen af deres bunke men med billedsiden nedaf. Herefter fortsætter spillet indtil en af spillerne vinder bunken på bordet. I tilfælde af at en spiller løber tør for kort taber spilleren. I det specielle og meget sjældne tilfælde at de to spillere samtidig løber tør for kort er der tale om uafgjort.

I denne opgave skal der arbejdes mod at få computeren til at spille kortspillet krig mod sig selv. Vi er i sidste ende interesseret i at få viden om hvormange udspil en spiller i gennemsnit skal foretage før et spil er afgjort samt hvor ofte spillet ender uafgjort.

Opgaven er delt i tre dele. I den første delopgave skal der arbejdes mod at finde metoder til at blande og dele kort fra en bunke. I den anden delopgave skal der arbejdes mod at få computeren til at spille mod sig selv. Endelig skal der i den tredie delopgave arbejdes mod at lave statistik på et antal kørsler af spillet.

0.1.3 Opgave(r)

0.1.1: Kortspilsoperationer.

I denne første delopgave skal der implementeres en række generelle funktioner på kortspil. I det følgende skal vi antage at kuløren på et kort er uden betydning. Vi kan derfor repræsentere et kort som et heltal mellem 2 og 14 (et Es repræsenteres som værdien 14). Et spil kort repræsenteres som en liste af kort:

```
type card = int
type deck = card list
```

Implementér en funktion `deal` med type `deck->deck*deck`, som deler en kortstak i to lige store stakke ved skiftevis at give et kort til hver stak. I tilfælde af at der er et ulige antal kort i argumentstakken skal den første stak i resultatparret prioriteres. Kaldet `deal [4;9;2;5;3]` skal således returnere parret `([3;2;4],[5;9])`.

Følgende funktion kan benyttes til at generere tilfældige heltal i F#:

```
let rand : int -> int = let rnd = System.Random()
                        in fun n -> rnd.Next(0,n)
```

Funktionen `rand` tager et heltal $n > 0$ og returnerer et tal i intervallet $[0;n[$. Bemærk at funktionen ikke er funktionel i matematisk forstand (den returnerer ikke nødvendigvis den samme værdi hver gang den kaldes med det samme argument.)

Implementér nu, ved brug af `rand`, en funktion `shuffle` med type `deck->deck`, der givet en kortstak returnerer en blandet version af kortstakken. Her følger en mulig strategi for at blande en kortstak D af længde n:

(a) Konstruér en liste L af n tilfældige tal.

(b) Konstruér en liste af par bestående af parvis elementer fra D og L.

(c) Sortér listen af par ved brug af en ordningsrelation på de tilfældige tal i parrene.

(d) Udtræk kortene fra den sorterede liste.

Ovenstående strategi kan implementeres i F# ved brug af `List.map2`, `List.sortWith`, samt funktionen `compare` anvendt på heltal.[1]

Der findes andre (f.eks. rekursive) strategier hvormed en kortstak kan blandes. I er velkomne til at designe og skrive jeres egen blandingsfunktion.

Implementer også en funktion `newdeck` med type `unit->deck`, der returnerer en blandet kortstak bestående af 52 kort (4 serier af 13 kort med pålydende værdier fra 2 til 14).

I rapporten skal I beskrive jeres designovervejelser og demonstrere at jeres funktioner fungerer som forventet (skriv tests for de implementerede funktioner).

0.1.2: Spillet

Til brug for implementation af spillet skal vi benytte følgende type til at repræsentere en spillers situation under et spil:

```
type player = deck
```

En spillers situation i et spil er således repræsenteret ved en stak af kort.

I første omgang ønskes implementet to funktioner til henholdsvis at trække et kort fra en spillers hånd (såfremt hånden ikke er tom) samt indsætte en stak kort i bunden af en spillers kortstak:

```
val getCard  : player -> (card * player) option
val addCards : player -> deck -> player
```

Når der tilføjes en kortstak i bunden af en spillers hånd skal kortstakken først blandes. Bemærk at `getCard` trækker kort fra toppen af kortstakken og `addCards` tilføjer kort i bunden af kortstakken.

Der ønskes nu implementeret en rekursiv funktion `game` der simulerer at to spillere spiller spillet krig mod hinanden:

```
val game : card list -> player -> player -> int
```

[1] Funktionen `compare:int->int->int` tager to argumenter `a` og `b` og returnerer 1 hvis `a>b`, -1 hvis `a<b` og 0 hvis `a=b`. Funktionen `List.sortWith` har type `('a->'a->int)->'a list->'a list` og tillader således at tage en ordningsrelation som argument.

Funktionen `game` tager some det første argument en liste af kort, der repræsenterer de kort der ligger på bordet (til at starte med er denne liste tom). Derudover tager funktionen repræsentationerne af de to spillere som argumenter. Kroppen af funktionen kan nu passende trække kort fra spillernes hænder og undersøge om den ene spiller vinder runden eller om der er krig. Funktionen skal enten foretage et rekursivt kald eller umiddelbart returnere et heltal, som har til hensigt at identificere hvilken af spillerne der har vundet (der skal returneres 1 hvis spiller 1 har vundet, 2 hvis spiller 2 har vundet og 0 hvis det er uafgjort.)

Hint: Kroppen af funktionen skal også indeholde kode der tager sig af at håndtere det tilfælde at der trækkes to ens kort (der er krig). I det tilfælde forsøges der igen med træk af kort fra spillernes hænder, hvorefter der fortsættes rekursivt med de trukne kort lagt på bordet.

I rapporten skal I beskrive jeres designovervejelser og demonstrere at jeres funktioner fungerer som forventet (skriv tests for de implementerede funktioner). Husk at vise jeres implementation af funktionen `game` i rapporten. Bemærk at det kan være vanskeligt at teste kode der benytter sig af funktionen `rand`. For at adressere dette problem kan I eventuelt (til brug for tests) erstatte funktionen `shuffle` med en mere deterministisk (dvs. funktionel) funktion. Til testformål vil det også være oplagt at køre koden på kortstakke med væsentlig færre end de sædvanlige 52 kort (f.eks. 4 og 6 kort).

0.1.3: Statistik på spillet.

Denne delopgave går ud på at besvare to interessante spørgsmål omkring spillet krig:

(a) Hvor lang tid tager det i gennemsnit (talt i antal udspil pr spiller) at spille et spil krig?

(b) Hvor tit sker det at spillet ender uafgjort.

I første omgang skal der implementeres en udviddelse til funktionen `game` der gør at funktionen også returnerer antallet af udspil en af de to spillere har foretaget.

Til at svare på de to spørgsmål har I nu brug for at skrive en funktion der kan køre N spil, hvor N er en parameter givet til funktionen. Funktionen, som passende kan kaldes `runGames`, skal udskrive en et-linies rapport om hvormange gange de to spillere hver har vundet, hvor mange gange spillet er endt uafgjort, samt spillenes gennemsnitlige antal udspil.

I rapporten skal I beskrive hvordan I har udviddet funktionen `game` til at returnere antal udspil samt vise en udskrift af kørsel med jeres kode hvor $N = 10000$.