Programmering og Problemløsning
Datalogisk Institut, Københavns Universitet
Arbejdsseddel 7 - gruppeopgave

Martin Elsman og Jon Sporring

19. oktober - 7. november.
Afleveringsfrist: lørdag d. 7. november kl. 22:00.

Denne arbejdsseddel strækker sig over to uger og indeholder således øvelsesopgaver dækkende matriale både for uge 7, omhandlende typer og mønstergenkendelse, og for uge 8, omhandlende rekursive datastrukturer.

Emnerne for denne arbejdsseddel er:

- rekursion, mønstergenkendelse (pattern matching),
- sum-typer,
- træstrukturer.

Opgaverne er opdelt i øve- og afleveringsopgaver. I denne periode skal I arbejde i grupper med jeres afleveringsopgaver. Regler for gruppe- og individuelle afleveringsopgaver er beskrevet i "'Noter, links, software m.m."'→"'Generel information om opgaver"'. Afleveringsopgaverne er designet således at I kan starte på dem allerede i uge 7.

Øveopgaver for uge 7

I det efterfølgende skal der arbejdes med sum-typen:

```
type weekday = Monday | Tuesday | Wednesday | Thursday
             | Friday | Saturday | Sunday
```

som repræsenterer ugens dage.

7ø1 Lav en funktion `dayToNumber : weekday -> int`, der givet en ugedag returnerer et tal, hvor mandag skal give tallet 1, tirsdag tallet 2 osv.

7ø2 Lav en funktion `nextDay : weekday -> weekday`, der givet en ugedag returnerer den næste dag, så mandag skal give tirsdag, tirsdag skal give onsdag, osv, og søndag skal give mandag.

7ø3 Lav en funktion `numberToDay : n : int -> weekday option`, sådan at `numberToDay n` returnerer `None`, hvis `n` ikke ligger i intervallet $1 \ldots 7$, og ellers returnerer ugedagen `Some d`. Det skal gælde, at `numberToDay (dayToNumber d)` $\rightsquigarrow$ `Some d` for alle ugedage `d`.

We will use the following three types to implement various functions relating to cards.

```
type suit = Hearts | Diamonds | Clubs | Spades // The suit of a card

type rank = Two | Three | Four | Five | Six    // The rank of a card
          | Seven | Eight | Nine | Ten
          | Jack | Queen | King | Ace

type card = rank * suit           // Combination of a rank and a suit
```

7ø4 Write a function `succSuit : suit -> suit option` that takes a suit as argument and returns the next suit as an optional value. The call `succSuit Hearts` should return the value `Some Diamonds`. The call `succSuit Spades` should return `None`.

7ø5 Write a function `succRank : rank -> rank option` that takes a rank as argument and returns the next rank as an optional value. The call `succRank Two` should return the value `Some Three`. The call `succRank Ace` should return the value `None`.

7ø6 Write a function `succCard : card -> card option` that takes a card as argument and returns the next card as an optional value. To implement the function, use a `match` construct and the two functions `succSuit` and `succRank`.

The call `succCard (Ace, Spades)` should return `None`. If `succRank` returns `None` and `succSuit` returns `Some` s, where s is a suit, `succCard` should return the value `Some (Two,`s`)`.

7ø7 Using recursion and pattern matching, write a function `initDeck : unit -> card list` that returns a full deck of cards. Check that the call `initDeck()` returns a list of length 52. **Hint:** Implement a recursive helper function that takes a card c as argument and uses pattern matching on the result of calling `succCard` on c. Use the card `(Two,Hearts)` in the initial call to your helper function.

7ø8 Write a function `sameRank : card -> card -> bool` that checks that the two argument cards have the same rank.

7ø9 Write a function `sameSuit : card -> card -> bool` that checks that the two argument cards are of the same suit.

7ø10 Write a function `highCard : card -> card -> card` that takes two cards as arguments and returns the card with the highest rank. In case the cards have the same rank, the function should return the first argument. **Hint:** You can use the >= operator to compare ranks.

The following exercises are about defining and using simple sum types.

7ø11 Implement a function `safeDivOption : int -> int -> int option` that takes two integers a and b as arguments and returns `None` if b is 0 and the value `Some`(a/b), otherwise.

7ø12 Consider the parametric type `result`, defined as follows:

```
type ('a,'b) result = Ok of 'a | Err of 'b
```

Implement a function `safeDivResult : int -> int -> (int,string) result` that takes two integers a and b as arguments and returns `Err "Divide by zero"` if b is 0 and the value `Ok`(a/b), otherwise.

Øveopgaver for uge 8

The following exercises are about expanding and using the following recursive sumtype, which can be used for modelling expression terms:

```
type expr = Const of int | Add of expr * expr | Mul of expr * expr
```

7ø13 Implement a recursive function `eval : expr -> int` that takes an expression value as argument and returns the integer resulting from evaluating the expression term. The expression `eval (Add(Const 3,Mul(Const 2,Const 4)))` should return the integer value `11`.

7ø14 Extend the type `expr` with a case for subtraction, extend the evaluator with a proper `match`-case for subtraction, and evaluate that your implementation works in practice.

7ø15 Extend the type `expr` with a case for division and refine the evaluator function `eval` to have type `expr->(int,string)result`. Evaluate that your implementation will propagate "Divide by zero" errors to the toplevel.

In the following exercises, we shall investigate the following recursive type definition for trees:

```
type 'a tree = Leaf of 'a | Tree of 'a tree * 'a tree
```

The tree type is generic in the type of information that can be installed in `Leaf` nodes.

7ø16 Write a function `sum : int tree -> int` that returns the sum of the integer values appearing in the leafs of the tree. Evaluate that your function works as expected.

7ø17 Write a function `leafs : 'a tree -> int` that returns the number of leaf nodes appearing in a tree. Evaluate that your function works as expected.

7ø18 Write a function `find : ('a -> bool) -> 'a tree -> 'a option` that, using a preorder traversal, returns the first value that satisfies the provided predicate. If no such value appears in the tree, the function should return the value `None`. Evaluate that your function works as expected.

I det følgende skal vi benytte os af biblioteket `ImgUtil`, som beskrevet i forelæsningerne. Biblioteket `ImgUtil` gør det muligt at tegne punkter og linier på et canvas, at eksportere et canvas til en billedfil (en PNG-fil), samt at vise et canvas på skærmen i en simpel F# applikation. Biblioteket (nærmere bestemt F# modulet `ImgUtil`) er gjort tilgængeligt via en F# DLL kaldet `img_util.dll`. Koden for biblioteket og dokumentation for hvordan DLL'en bygges og benyttes er tilgængelig via github på `https://github.com/diku-dk/img-util-fs` samt i Absalon-folderen "Files → forelæsninger → Uge 06: Rekursion → kode → img-util-fs".

7ø19 Vi skal nu benytte biblioteket `ImgUtil` til at tegne Sierpinski-fraktalen, der kan tegnes ved at tegne små firkanter bestemt af et rekursivt mønster. Koden for Sierpinski-trekanten er givet som følger:

```
open ImgUtil

let rec triangle C len (x,y) =
  if len < 25 then setBox blue (x,y) (x+len,y+len) C
  else let half = len / 2
       do triangle C half (x+half/2,y)
       do triangle C half (x,y+half)
       do triangle C half (x+half,y+half)

do runSimpleApp "Sierpinski" 600 600
                (fun w h ->
                   let C = mk w h
                   in (triangle C 512 (30,30); C))
```

Tilpas funktionen således at trekanten tegnes med røde streger samt således at den kun tegnes 2 rekursionsniveauer ned. **Hint:** dette kan gøres ved at ændre betingelsen `len < 25`. Til at starte med kaldes funktionen `triangle` med `len=512`, på næste niveau kaldes `triangle` med `len=256`, og så fremdeles.

7ø20 I stedet for at benytte funktionen `ImgUtil.runSimpleApp` er det nu meningen at du skal benytte funktionen `ImgUtil.runApp`, som giver mulighed for at din løsning kan styres ved brug af tastaturet. Funktionen `ImgUtil` har følgende type:

```
val runApp : string -> int -> int
          -> (int -> int -> 's -> canvas)
          -> ('s -> Key -> 's option)
          -> 's -> unit
```

De tre første argumenter til `runApp` er vinduets titel (en streng) samt vinduets initielle vidde og højde. Funktionen `runApp` er parametrisk over en brugerdefineret type af tilstande (`'s`). Antag at funktionen kaldes som følger:

```
do runApp title width height draw react init
```

Dette kald vil starte en GUI applikation med titlen `title`, vidden `width` og højden `height`. Funktionen `draw`, som brugeren giver som 4. argument kaldes initielt når applikationen starter og hver gang vinduets størrelse justeres eller ved at funktionen `react` er blevet kaldt efter en tast er trykket ned på tastaturet. Funktionen `draw` modtager også (udover værdier for den aktuelle vidde og højde) en værdi for den brugerdefinerede tilstand, som initielt er sat til værdien

`init`. Funktionen skal returnere et canvas, som for eksempel kan konstrueres med funktionen `ImgUtil.mk` og ændres med andre funktioner i `ImgUtil` (f.eks. `setPixel`).

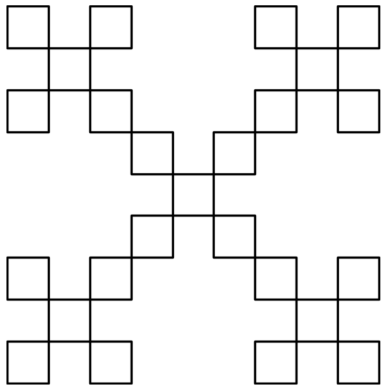
Funktionen `react`, som brugeren giver som 5. argument kaldes hver gang brugeren trykker på en tast. Funktionen tager som argument:

- en værdi svarende til den nuværende tilstand for applikationen, og
- et argument der kan benyttes til at afgøre hvilken tast der blev trykket på.[1]

Funktionen kan nu (eventuelt) ændre på dens tilstand ved at returnere en ændret værdi for tilstanden.

Tilpas applikationen således at dybden af fraktalen kan styres ved brug af piletasterne, repræsenteret ved værdierne `Gdk.Key.u` og `Gdk.Key.d`.

7ø21 Med udgangspunkt i øvelsesopgave 7ø19 skal du i denne opgave implementere en GUI-applikation der kan tegne en version af X-fraktalen som illustreret nedenfor (eventuelt i en dybde større end 2).



Bemærk at det ikke er et krav, at dybden på fraktalen skal kunne styres med piletasterne, som det er tilfældet med Sierpinski-fraktalen i øvelsesopgave 7ø20.

Afleveringsopgaver

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

[1] Hvis `k` har typen `Gdk.Key` kan betingelsen `k = Gdk.Key.d` benyttes til at afgøre om det var tasten "d" der blev trykket på. Desværre er det ikke muligt med den nuværende version af `ImgUtil` at reagere på tryk på piletasterne.

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.

7g0 **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, hvert mellem 0 og k hvor $k >> n$.

(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.[2]

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

[2] 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.

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).

7g1 **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
```

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).

7g2 **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 $N = 10000$.

Krav til afleveringen

Afleveringen skal bestå af

- en zip-fil, der hedder `7g_<navn>.zip` (f.eks. `7g_jon.zip`)
- en pdf-fil , der hedder `7g_<navn>.pdf` (f.eks. `7g_jon.pdf`)

Zip-filen `7g_<navn>.zip` skal indeholde en og kun en mappe `7g_<navn>`. I den mappe skal der ligge en `src` mappe og filen `README.txt`. I `src` skal der ligge følgende og kun følgende filer: `wargame-shuffle.fs`, `wargame-game.fs` og `wargame-game-stats.fs` svarende til hver af delopgaverne. De skal kunne oversættes med `fsharpc`, og de oversatte filer skal kunne køres med mono. Funktioner skal dokumenteres ifølge dokumentationsstandarden som minimum ved brug af `<summary>`, `<param>` og `<returns>` XML-tagsne. Filen `README.txt` skal ganske kort beskrive, hvordan koden oversættes og køres. Pdf-filen skal indeholde jeres rapport oversat fra LaTeX. Husk at pdf-filen skal uploades ved siden af zip-filen på Absalon.

Addendum: Yderligere skal I aflevere testfiler for de forskellige delopgaver. Testfiler kan afleveres enten som `fs`-filer eller som `fsx`-filer. I rapporten skal I dokumentere resultatet af at køre de forskellige testfiler.

God fornøjelse.