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

Jon Sporring

15. oktober - 13. november.
Afleveringsfrist: lørdag d. 13. november kl. 22:00.

Denne arbejdsseddel strækker sig over flere 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...7, og ellers returnerer ugedagen `Some d`. Det skal gælde, at `numberToDay (dayToNumber d)` ⇝ `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`).

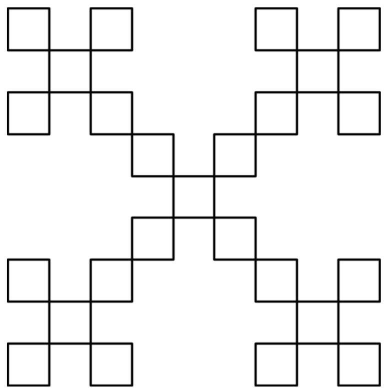
Funktionen `react`, som brugeren giver som 5. argument kaldes hver gang brugeren trykker på en tast. Funktionen tager som argument:

- 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

In this assignment, you are to work with a puzzle called Rotate. The puzzle consists of a square board with $n \times n, n \in \{2,3,4,5\}$ fields similarly to a chess-board. Each field has a unique id-number, which we will call the field's position, and on each field is one unique letter from the alphabet 'a', 'b', .... For example, when $n = 4$, then the board could look like,

| | | | |
|---|---|---|---|
| h | o | l | k |
| b | i | g | e |
| f | m | c | a |
| j | n | d | p |

and the position of the respective fields are

[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.

$$\begin{array}{cccc} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{array}$$

The puzzle is solved by rotating the letters in small $2 \times 2$ subsquares clockwise until the board reaches the state

$$\begin{array}{cccc} \mathrm{a} & \mathrm{b} & \mathrm{c} & \mathrm{d} \\ \mathrm{e} & \mathrm{f} & \mathrm{g} & \mathrm{h} \\ \mathrm{i} & \mathrm{j} & \mathrm{k} & \mathrm{l} \\ \mathrm{m} & \mathrm{n} & \mathrm{o} & \mathrm{p} \end{array}$$

A rotation is specified by the position of its top-left corner, and all but the right-most column and the bottom-most row are valid inputs to the rotation operation. Let $p_1, p_2, p_3, p_4 \to q_1, q_2, q_3, q_4$ denote a rotation from $p_*$ to $q_*$, where $p_1$ is the top-left corner, then for example, specifying a rotation of subsquare 1 results in $1,2,5,6 \to 5,1,6,2$, or equivalently,

$$\begin{array}{cccc} \mathrm{h} & \mathrm{o} & \mathrm{l} & \mathrm{k} \\ \mathrm{b} & \mathrm{i} & \mathrm{g} & \mathrm{e} \\ \mathrm{f} & \mathrm{m} & \mathrm{c} & \mathrm{a} \\ \mathrm{j} & \mathrm{n} & \mathrm{d} & \mathrm{p} \end{array} \to \begin{array}{cccc} \mathrm{b} & \mathrm{h} & \mathrm{l} & \mathrm{k} \\ \mathrm{i} & \mathrm{o} & \mathrm{g} & \mathrm{e} \\ \mathrm{f} & \mathrm{m} & \mathrm{c} & \mathrm{a} \\ \mathrm{j} & \mathrm{n} & \mathrm{d} & \mathrm{p} \end{array}$$

The overall task of this assignment is to build a program, that will generate rotate-puzzles and allow you to iteratively enter a sequence of positions untill the puzzle is solved. Detailed requirements are:

- If your program includes loops, then the loops must be programmed using recursion
- Your program must use lists and not arrays.
- Your program must not use mutable values (variables).
- Your solution must be parameterized by $n$, the size of the board.
- you must represent your board as a list of letters, e.g., for $n = 4$ the board for a solved puzzle must be `['a' .. 'p']`
- Your program must consist of the following files

  `game.fsx`, `rotate.fsi`, `rotate.fs`, `whiteboxtest.fsx`, and `blackboxtest.fsx`.

  The files `rotate.fsi`, `rotate.fs` must be the interface and implementation of a library with your main types, functions, and values; the file `game.fsx` must be a maximally 10 line program, which defines the value $n$ and starts the game; and `whiteboxtest.fsx` and `blackboxtest.fsx` must contain your test for the library.

As part of this assignment, you are to write a maximally 10-page report following `rapport.tex` template.

Note that calls to `System.Random ()` returns a random number generator which has a function `Next : n:int -> int` which draws a random non-negative integer less than `n`. For example,

```
let rnd = System.Random ()
for i = 1 to 3 do
  printfn "%d" (rnd.Next 10)
```

prints 3 random non-negative integers less than 10.

7g0 Write the interface file for the library `rotate` with user defined types `Board` which is a list of characters and `Position` which is an integer and with the following functions,

```
create : n:uint -> Board
board2Str : b:Board -> string
validPosition : b:Board -> p:Position -> bool
rotate : b:Board -> p:Position -> Board
scramble : b:Board -> m:uint -> Board
solved : b:Board -> bool
```

The function `create` must take an integer $n$ and return a $n \times n$ board in its solved state.

The function `board2Str` must take a board and return a string, containing the board formatted such that it can be printed with the `printfn "%s"` command and formatting string.

The function `validPosition` must take a board and rotation position and return true or false depending on whether the position is a valid rotation position or not.

The function `rotate` must take a board and a position and return another board, which is identical to the original but where a local $2 \times 2$ rotation has been performed at the indicated position. If an invalid position is given, then the function must return an empty board.

The function `scramble` must take a board and an unsigned int `m` and return another board, where all the elements of the original board have been rotated by `m` random set of legal rotations using `rotate`.

The function `solved` must take a board and return true or false depending on whether the puzzle has been solved or not.

The interface must include documentation following the documentation standard.

7g1 Write a program `blackboxtest.fsx` which performs a blackbox test of the yet to be implemented `rotate` library.

7g2 Write the implementation file of the `rotate` library.

7g3 Write a program `whiteboxtest.fsx` which performs a whitebox test of the `rotate` library.

7g4 Write a short program, `game`, which defines the size of the board $n$ and starts the game, and which has all the interaction with the user in a game-loop using recursion.

7g5 A fellow programmer has made the function

```
solutions : b:Board -> m:int -> int list
```

which takes an non-negative integer $m$ and seeks a possible solutions to a given board within maximally $m$ rotations. The program compiles and runs without error, but for some combinations of board-size $n$ and maximum rotations $m$, the program is very slow and the computer eventually runs out of memory. Why do you think this may be?

# 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: `rotate.fsi`, `rotate.fs`, `blackboxtest.fsx`, `whiteboxtest.fsx` og `game.fsx` 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.

God fornøjelse.