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

Jon Sporring, Martin Elsman, Torben Mogensen, Christina Lioma

November 9, 2022

Contents

Chapter 1

LaTeX

1.1 LaTeX

1.1.1 Teacher's guide

1.1.2 Introduction

1.1.3 Exercise(s)

1.1.3.1: Make a new project in `https://overleaf.com`, and compile the default document. Download its pdf, and download the project. Verify that the pdf and the `.tex` file that you have downloaded, looks like what you have entered in overleaf.

1.1.3.2: Make a new project in `https://overleaf.com`, and update the default `.tex`-file to contain as little as possible, while still being able to compile. What is the shortest LaTeXprogram possible?

1.1.3.3: Make a new project in `https://overleaf.com`, and write a short document in LaTeX. The report should as minimum contain:

- A title produced using `\maketitle`,
- A section with a section title using `\section`,
- One or more figures of images from your program, using the `figure`-environment. All figures must include a caption text using `\caption`.
- A reference to the figure using the `\label`–`\ref` pair.
- The Danish letters 'æ', 'ø', and 'å'.

1.1.3.4: Make a new project in `https://overleaf.com`, replace the relevant file(s) with `opgave.tex` from Absalon. Compile it and check that the is correct.

Chapter 2

Scratch

2.1 Getting Started

2.1.1 Teacher's guide

This exercise focusses on getting the students up an running with Scratch, how to use the command line, and how to produce a simple report in LaTeX. It has no assumtions on the student's abilities.

Topics Imperativ programmering using Scratch, command line/terminal and the file structure, a text editor, report writing using LaTeX

Difficulty level Easy

2.1.2 Introduction

Scratch is a visual programming language using the imperative programming paradigm and where the programming elements are structured as blocks with connectors.

2.1.3 Exercise(s)

2.1.3.1: Start the command line (or terminal on MacOS). Use the `cd`-command to navigate to a suitable directory for your work. (e.g. the Documents folder). Use the

```
mkdir <name>
```

command to create a new directory from the command line. Replace `<name>` with the name of your new directory. Use `ls` or `dir` to verify that the new directory is empty. Locate the same directory with the Graphical User Interface.

2.1.3.2: Make a Scratch-program, which counts up every even number from 0 to 20.

2.1.3.3: Make a Scratch-program, which counts down from 10 to 1. You must use a variable and a repeat loop.

2.1.3.4: Make a Scratch-program, which counts down from 10 to 1. The countdown should start/begin when you click the mouse.

2.1.3.5: Design a game in Scratch:
You are to design a game of your own choosing. The game must

- include 2–5 sprites
- have a typical gameplay of about 1 minute
- must include at least 1 variable

You may use any existing block in Scratch, and the game may be similar to an existing game. The graphical appeal and the sound aspects of the games are of little importance.

A good approach is to:

(a) Start by brainstorming about a game, you would like to make and what the game mechanics should be.

(b) Sketch a design on paper about the gameplay.

(c) Implement your design as a sketch of a Scratch program, still on paper.

(d) Enter your prototype into Scratch and test it.

(e) Return to the top and update your game until you are satisfied with the result.

2.1.3.6: Make your own "hello world" Scratch-program. The program must make the default sprite say "Hello World" when you press the green flag.

2.1.3.7: Install Scratch on your machine.

2.1.3.8: Install Scratch on your machine.

2.1.3.9: Make a Scratch program with a sprite of your own choosing, which moves on the screen using the 'glide'-block and the 'forever' loop-block.

2.1.3.10:

2.1.3.11: Take one or more screenshots of one of your Scratch-programs while it runs.

2.2 Game

2.2.1 Teacher's guide

Emne lave et spil, lave den første Absalon aflevering

Sværhedsgrad Let

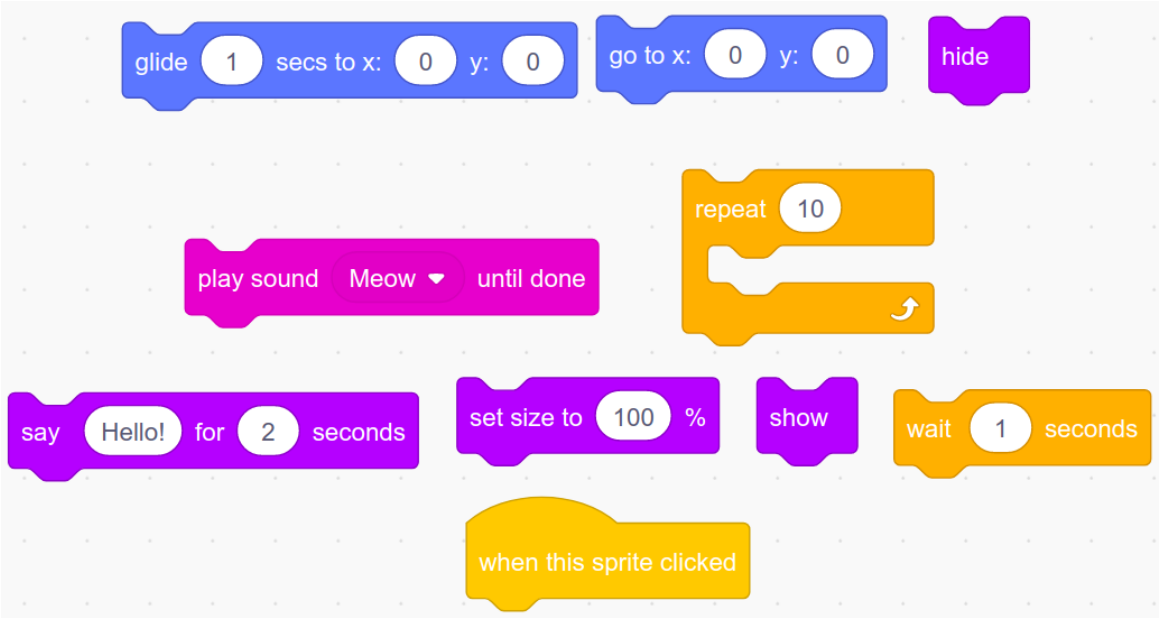


Figure 2.1: 10 Scratch-blokke

2.2.2 Introduction

2.2.3 Exercise(s)

2.2.3.1: Hvad kan I lave med 10 blokke?
I Figur 2.1 ser I 10 Scratch blokke. Jeres opgave er at lave et sjovt program kun ved brug af disse blokke. Hver blok må bruges 0, 1 eller flere gange. Prøv at sammensætte programmet ved at tegne blokkene på papir, og skriv ned, hvad I tror programmet vil gøre. Sæt jer dernæst til computeren, og indtast jeres program. Beskriv, i hvor høj grad programmet gør, som I forventede. Vend dernæst tilbage til designfasen og forbedre evt. programmet. Til slut uploades programmet til gruppens studio i Scratch.

2.2.3.2: Design et spil
I skal designe og implementere et spil efter eget valg. Spillet skal

- indeholde 2-5 sprites
- vare ca. 1 minut at spille
- benytte mindst 1 variabel

Det må benytte alle tilgængelige blokke i Scratch og må gerne minde om et spil I kender. Det er ikke vigtigt, at det er et grafisk eller lydmæssigt prangende spil.

Start med at tale om hvad I kunne tænke jer, spillet skal omhandle. Skitser på papir, hvordan game-playet, skal forløbe. Skitser derefter på papir hvordan det kunne implementeres i Scratch. Indtast programmet på computeren og afprøv, om spillet gør, som I forventer. Vend evt. tilbage til designfasen og forbedre spillet.

Chapter 3

Canvas

3.1 Canvas

3.1.1 Teacher's guide

Emne rekursion, grafik og winforms

Sværhedsgrad Middel

3.1.2 Introduction

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`.

3.1.3 Exercise(s)

3.1.3.1: Create an empty canvas of width 400 pixels and height 600 pixels (400×600). Draw a red, green, blue, and yellow line on the left, bottom, right, and top edge, and a black 200×300 rectangle in the middle.

3.1.3.2: Consider points on a circle is give by the coordinate functions

$$x(t) = x_0 + r\cos(t) \tag{3.1}$$
$$y(t) = y_0 + r\sin(t) \tag{3.2}$$

whose center is (x_0, y_0), r is its radius, and where $0 \leq t < 2\pi$. Use these equations to make a canvas of size 400×400 with an approximation of a circle centered at $(200, 200)$ and with a radius of 100. The approximation should consists of straight lines connecting $(x(t_i), y(t_i))$ with $(x(t_{i+1}), y(t_{i+1}))$ for $t = [0, 0.1, 0.2, \ldots, 2\pi)$.

3.1.3.3: Make a program, which draws a 20×20 square in the center of a canvas of size 400×400. When the user presses space, or an arrow key, the program should write to the terminal, which key has been pressed.

3.1.3.4: Extend the program in Exercise 3, such that when the arrow keys are pressed, then the square is moved in the direction of the arrow pressed.

3.1.3.5: 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.

3.1.3.6: 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`).

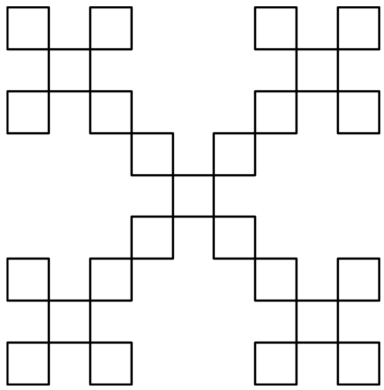
Funktionen `react`, som brugeren giver som 5. argument kaldes hver gang brugeren trykker på en tast. Funktionen tager som argument:

- 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`.

3.1.3.7: Med udgangspunkt i øvelsesopgave 1 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 2.

I det følgende skal vi benytte os af biblioteket `Canvas` til at lave et interaktiv applikation in F#.

3.2 The Box

3.2.1 Teacher's guide

Emne grafik og interaktion

Sværhedsgrad Middel

3.2.2 Introduction

I det følgende skal vi benytte os af biblioteket `Canvas` til at lave et interaktiv applikation in F#.

[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.

3.2.3 Exercise(s)

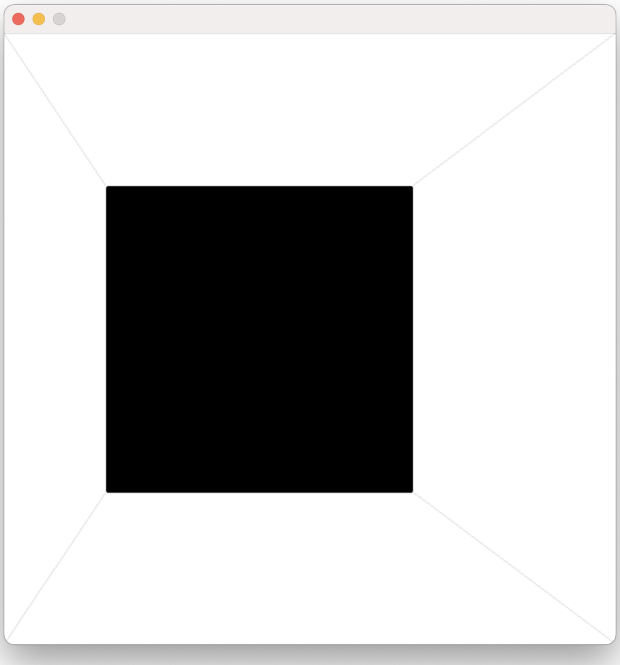
3.2.3.1: Make a simple, interactive program using the Canvas library, and which could be the start of a game. You are to modify the demo-program `color_boxes.fsx`, such that instead of colored boxes the resulting program:

(a) Creates an interactive canvas

(b) Draws a black box on a white background with grey lines from the corners of the box to the nearest corner of the canvas-

(c) When the user presses the right or left arror key, then the box with lines are moved to the right and left respectively.

An example of how this could look is:



The functions `runApp`, `setLine`, and `getKey` are most likely essential for this task.

Chapter 4

F# Functional Programming

4.1 Recursion

4.1.1 Teacher's guide

Emne Rekursion

Sværhedsgrad Middel

4.1.2 Introduction

Recursion is a central concept in functional programming, and is usually used instead of `for` and `while` loops. The following exercise thus train the use of recursion.

4.1.3 Exercise(s)

4.1.3.1: Use `List.fold` and `List.foldback` and your own implementations from Assignment 5 to calculate the sum of the elements in `[0 .. n]`, where `n` is a large number.

4.1.3.2: The Collatz conjecture is a famous unsolved problems in mathematics[1]. The conjecture states that for any integer larger than 0, recursively applying,

$$f(n) = \begin{cases} \frac{n}{2} & \text{if } n\%2 = 0 \\ 3n+1 & \text{if } n\%2 = 1 \end{cases}$$

will always lead the value 1. For example, starting with 19, we get the sequence, `[19; 58; 29; 88; 44; 22; 11; 34; 17; 52; 26; 13; 40; 20; 10; 5; 16; 8; 4; 2; 1]`.

(a) Implement a non-recursive function `collatzStep : n:int -> int` that gives the next number in the collatz sequence. Examples:

[1] `https://en.wikipedia.org/wiki/Collatz_conjecture`

```
collatzStep 1 = 1
collatzStep 2 = 1
collatzStep 3 = 10
collatzStep 9 = 28
```

The function should use pattern matching.

(b) Implement a recursive function that counts the number of steps in the collatz sequence starting at n:

`collatzStepsHelper : count:int -> n:int -> int:`

The function should use pattern matching, recursion and the previous function `collatzStep` to compute each intermediate step of a sequence.

(c) Implement a non-recursive function `collatzSteps : n:int -> int` that simply calls `collatzStepsHelper` with an initial `count` of 0.

4.1.3.3: Write a function, `fac : n:int -> int`, which calculates the factorial function $n! = \prod_{i=1}^{n} i$ using recursion.

4.1.3.4: Consider the factorial-function,

$$n! = \prod_{i=1}^{n} i = 1 \cdot 2 \cdot \ldots \cdot n \tag{4.1}$$

(a) Write a function

`fac : n:int -> int`

which uses recursion to calculate the factorial-function as (4.1).

(b) Write a program, which asks the user to enter the number `n` using the keyboard, and which writes the result of `fac n`.

(c) Make a new version,

`fac64 : n:int64 -> int64`

which uses `int64` instead of `int` to calculate the factorial-function. What are the largest values n, for which `fac` and `fac64` respectively can correctly calculate the factorial-function for?

4.1.3.5: Make your own implementation of `List.fold` and `List.foldback` using recursion.

4.1.3.6: The greatest common demoninator of 2 integers t and n is the greatest number c, which integer-divides both t and n with 0 remainder. Euclid's algorithm[2] finds this greatest common denominator via recursion:

$$\gcd(t,0) = t, \tag{4.2}$$
$$\gcd(t,n) = \gcd(n, t \ \% \ n), \tag{4.3}$$

where % is the remainder operator (as in F#).

(a) Implement Euclid's algorithm as the recursive function:

`gcd : int -> int -> int`

[2] `https://en.wikipedia.org/wiki/Greatest_common_divisor`

(b) Test your implementation.

(c) Make a tracing-by-hand on paper for `gcd 8 2` and `gcd 2 8`.

4.1.3.7: Write a function `lastFloat : float list -> float` that, using recursion, returns the last element of the argument list if the list is non-empty and returns the float value `NaN` if the argument list is empty.

For example, the call `lastFloat [2.1;4.2]` should return the float value `4.2` and the call `lastFloat []` should return the value `NaN`.

4.1.3.8: Write a function `length : 'a list -> int` that calculates the length of the argument list using recursion. The function should make use of pattern matching on lists.

4.1.3.9: Write a tail-recursive function `lengthAcc : acc:int -> xs:'a list -> int` that calculates the sum of `acc` plus the length of `xs` using tail-recursion. For instance, a call `lengthAcc 0 xs` should return the length of `xs`, and a call `lengthAcc n []` should return the value `n`.

4.1.3.10: Write a function `map : ('a -> 'b) -> 'a list -> 'b list` that takes a function and a list as arguments and returns a new list of the same length as the argument list with elements obtained by applying the supplied function on each of the elements of the argument list. The function should make use of recursion and pattern matching on lists.

As an example, a call `map (fun x -> x+2) [2;3]` should return the list `[4;5]`.

4.1.3.11: Write a function `pow2 : n:int -> int` that calculates the value 2^n using recursion. The function should return the value 0 when $n < 0$.

4.1.3.12: Using Steps 1, 3, 5, 7, and 8 from the 8-step guide

(a) write a recursive function which takes two integer arguments x and n and returns the value x^n.

(b) write another function wich takes one argument (x,n) and calls the former.

Document both functions using the `<summary>`, `<param>`, and `<returns>` XML tags. Consider what should happen, if $n < 0$, and whether there is any significant difference between the call of the two functions.

4.1.3.13: Write a function, `sum : n:int -> int`, which calculates the sum $\sum_{i=1}^{n} i$ using recursion. Make an implementation using a `while` loop, compute a table for the values $n = 1..10$ and compare the results. Discuss the difference between the two methods.

4.1.3.14: The following code calculates the sum $\sum_{i=0}^{n} i$ using recursion and 64-bit unsigned integeres.

```
let rec sum (n: uint64) : uint64 =
  match n with
    0UL -> 0UL
    | _ -> n+sum (n-1UL)

let n = uint64 1e4
printfn "%A" (sum n)
```

Depending on the computer, this program runs out of memory for large values of n, e.g., not many compupters can run this program when `n = uint64 1e10`. The problem is, that the algorithm is not using tail recursion.

(a) Explain why this is not tail recursion.

(b) Run the code on your computer and empirically find a small n for which the program runs out of memory on your computer.

(c) Rewrite the algorithm to become tail recursive by including the accumulated sum as the argument:

```
sum (acc: uint64) -> (n: uint64) -> uint64
```

and check that this does not run out of memory for the above identified n. Try also bigger.

(d) For which values of n would the tail recursive version break and why?

4.1.3.15: Write a function `sum : int list -> int` that takes a list of integers and returns their sum. The function must traverse the list using recursion and pattern matching.

4.1.3.16: Consider the following sum of integers,

$$\sum_{i=1}^{n} i \tag{4.4}$$

This assignment has the following sub-assignments:

(a) Write a function

```
sum : n:int -> int
```

which uses pattern-matching and recursion to compute the sum $1+2+\cdots+n$ also written in (5.2). If the function is called with any value smaller than 1, then it is to return the value 0.

(b) By induction one can show that

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}, n \geq 0 \tag{4.5}$$

Make a function

```
simpleSum : n:int -> int
```

which uses (5.3) to calculate $1+2+\cdots+n$ and which includes a comment explaining how the expression implemented is related to the mentioned sum.

(c) Write a program, which asks the user for the number n, reads the number from the keyboard, and write the result of `sum n` and `simpleSum n` to the screen.

(d) Make a program, which writes a table to the screen with 3 columns: `n`, `sum n` and `simpleSum n`. The table should have a row for each of $n = 1,2,3,..,10$, and each field must be 4 characters wide. Verify programmatically that the two functions calculate identical results.

(e) What is the largest value n that the two sum-functions can correctly calculate the value of? Can the functions be modified, such that they can correctly calculate the sum for larger values of n?

4.2 Continued Fractions

4.2.1 Teacher's guide

Emne Rekursion

Sværhedsgrad Middel

4.2.2 Introduction

In this assignment, you will work with simple continued fractions[3], henceforth just called continued fractions. Continued fractions are lists of integers which represent real numbers. The list is finite for rational numbers and infinite for irrational numbers.

Continued fractions to decimal numbers A continued fraction is written as $x = [q_0; q_1, q_2, \ldots]$ and the corresponding decimal number is found by the following recursive algorithm:

$$x = q_0 + \frac{1}{q_1 + \frac{1}{q_2 + \ldots}}. \tag{4.6}$$

The series of fractions continues as long as there are elements in the continued fraction.

For example, $[3;4,12,4] = 3.245$, since:

$$x = 3 + \frac{1}{4 + \frac{1}{12 + \frac{1}{4}}} \tag{4.7}$$

$$= 3 + \frac{1}{4 + \frac{1}{12.25}} \tag{4.8}$$

$$= 3 + \frac{1}{4.081632653} \tag{4.9}$$

$$= 3.245. \tag{4.10}$$

Note that all but the first digit must be larger than 0, e.g., $[1;0]$ is an illigal number, and that every rational number has exactly 2 representations $[q_0; q_1, \ldots, q_n] = [q_0; q_1, \ldots, (q_n - 1), 1]$ where the first is called the canonical representation. E.g., $[2;3] = [2;2,1]$, since

$$2 + \frac{1}{3} = 2 + \frac{1}{2 + \frac{1}{1}}. \tag{4.11}$$

Decimal numbers to continued fractions For a given number x on decimal form, its continued fraction $[q_0; q_1, q_2, \ldots]$ can be found using the following algorithm:

Let $x_0 = x$ and $i \geq 0$, and calculate

$$q_i = \lfloor x_i \rfloor \tag{4.12}$$

$$r_i = x_i - q_i \tag{4.13}$$

$$x_{i+1} = 1/r_i \tag{4.14}$$

$$\tag{4.15}$$

recursively until $r_i = 0$. The continued fraction is then the sequences of q_i.

For example, if $x = 3.245$ then

[3] `https://en.wikipedia.org/wiki/Continued_fraction`

i	x_i	$q_i = \lfloor x_i \rfloor$	$r_i = x_i - q_i$	$x_{i+1} = 1/r_i$
0	3.245	3	0.245	4.081632653...
1	4.081632653...	4	0.081632653	12.25
2	12.25	12	0.25	4
3	4	4	0	-

and hence, the continued fraction is in the third column as $3.245 = [3;4,12,4]$.

4.2.3 Exercise(s)

4.2.3.1: Write a recursive function

```
cfrac2float : lst:int list -> float
```

that takes a list of integers as a continued fraction and returns the corresponding real number.

4.2.3.2: Write a function

```
float2cfrac : x:float -> int list
```

that takes a real number and calculates its continued fraction. Recall that floating-point numbers are inaccurate, so you should check that r_i is reasonably close to 0 instead of comparing it for equality to 0.0. For example, `abs ri < 1e-10`.

4.2.3.3: Skriv en rekursiv funktion

```
frac2cfrac : t:int -> n:int -> int list
```

som tager tæller og nævner i brøken t/n og udregner dens repræsentation som kædebrøk udelukkende ved brug af heltalstyper.

4.2.3.4: Skriv en rekursiv funktion

```
cfrac2frac : lst:int list -> i:int -> int * int
```

som tager en kædebrøk og et index og returnerer t_i/n_i approximationen som tuplen `(ti, ni)`.

4.2.3.5: Collect the above functions in a library as the interface file `continuedFraction.fsi` and implementation file `continuedFraction.fs`. Make a white- and blackbox test of these functions as the application `continuedFractionTest.fsx`.

4.3 Sort

4.3.1 Teacher's guide

Emne Mønstergenkendelse og black-box test

Sværhedsgrad Let

4.3.2 Introduction

4.3.3 Exercise(s)

4.3.3.1: Betragt insertion sort funktionen `isort`. Omskriv funktionen `insert` således, at den benytter sig af pattern matching på lister.

4.3.3.2: Betragt bubble sort funktionen `bsort`. Omskriv den, at den benytter sig af pattern matching på lister. Funktionen kan passende benytte sig af "nested pattern matching" i den forstand at den kan implementeres med et match case der udtrækker de to første elementer af listen samt halen efter disse to elementer.

4.3.3.3: Opskriv black-box tests for de to sorteringsfunktioner og vær sikker på at grænsetilfældene dækkes (ingen elementer, et element, to elementer, samt flere elementer, sorteret, omvendt sorteret, etc.)

4.3.3.4: Omskriv funktionen `merge`, som benyttes i forbindelse med funktionen `msort` (mergesort) fra forelæsningen, således at den benytter sig af pattern matching på lister.

4.3.3.5: Opskriv black-box tests for sorteringsfunktionen `msort` og vær sikker på at grænsetilfældene dækkes (ingen elementer, et element, to elementer, samt flere elementer, sorteret, omvendt sorteret, etc.)

4.4 Random Text

4.4.1 Exercise(s)

4.4.1.1: Write a function,

```
histogram : src:string -> int list
```

which counts occurrences of each of the characters `['a'..'z']@[' ']` in a string and returns a list. The first element of the list should be the count of 'a's, second the count of 'b's etc. Use this function to replace the mockup function in your library.

4.4.1.2: The function `markovChain` is a random function, and you are to test its output by the cooccurrences of the characters in the string it produces.

Extend your library with the function,

```
diff2 : c1:int list list -> c2:int list list -> double
```

which compares two histograms as the average sum of squared differences,

$$\text{diff2}(c_1, c_2) = \frac{1}{N^2} \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} (c_1(i,j) - c_2(i,j))^2 \tag{4.16}$$

where c_1 and c_2 are two cooccurrence histograms of N elements such that $c_1(i,j)$ is the number of times character number i is found following character number j.

Extend your test file with a test of `markovChain` as follows:

(a) Convert The Story using `convertText` and calculate its cooccurrence histogram.

(b) Use this to generate a random text using `markovChain` with length N, where N is the length of the converted The Story.

(c) Calculate the distance between the cooccurrence histograms of The Story and the random texts using `diff2`.

(d) Set the test as passed if the difference is below some threshold.

You must choose a threshold in the above, and you must argue for your choice.

4.4.1.3: The function `randomString` is not easily tested using white- or blackbox testing since it is a random function. Instead you are to test its output by the histogram of the characters in the string it produces.

Extend your library with the function,

```
diff : h1:int list -> h2:int list -> double
```

which compares two histograms as the average sum of squared differences,

$$\text{diff}(h_1, h_2) = \frac{1}{N} \sum_{i=0}^{N-1} (h_1(i) - h_2(i))^2 \tag{4.17}$$

where h_1 and h_2 are two histograms of N elements.

Extend your test file with a test of `randomString` as follows:

(a) Convert The Story using `convertText` and calculate its histogram.

(b) Use this to generate a random text using `randomString` with length N, where N is the length of the converted The Story.

(c) Calculate the distance between the histograms of The Story and the random texts using `diff`.

(d) Set the test as passed if the difference is below some threshold.

You must choose a threshold in the above, and you must argue for your choice.

4.4.1.4: Add a function to your library which converts a string, such that all letters are converted to lower case, and removes all characters except a...z and space. It should have the following type:

```
convertText : src:string -> string
```

4.4.1.5: The script `mockup.fsx` contains a number of functions including

```
randomString : hist:int list -> len:int -> string
```

The function `randomString` generates an identically and independently distributed string of a given length, where the characters are distributed according to a given histogram. The script is complete in the sense that it compiles and runs without errors, but its `histogram` function is a mockup function and does not produce the correct histograms.

Create the library file `textAnalysis.fs` and add the functions from `mockup.fsx`

4.4.1.6: Extend your library with a function

```
markovChain : cooc:int list list -> len:int -> string
```

which generates a random string of length `len`, whose character pairs are distributed according to the cooccurrence histogram `cooc`.

4.4.1.7: Write a program that generates a random string of length `len`, whose character triples are distributed according to a user specified trioccurrence histogram `trioc`. The function must have the type:

```
sndOrderMarkovModel : trioc:int list list list -> len:int -> string
```

Use the function developed in Exercise 22, and test your function by generating a random string, whose character triples are distributed as the converted characters in H.C. Andersen's fairy tale, "Little Claus and Big Claus". Calculate the trioccurrence histogram for the random string, and compare this with the original trioccurrence histogram.

4.4.1.8: The function `randomString` may be considered a zero-order Markov Chain model, since each generated character is independent on any previously generated characters. The function `fstOrderMarkovModel` generates new characters dependent on the previous character and is therefore a first-order Markov Chain model. Consider a similar extension to an n'th-order Markov Chain where the occurrences of n-tupples of characters are stored in `n : int list list ... list`. What possible pit-falls are there with this representation?

4.4.1.9: (a) The script `readFile.fsx` reads the content of the text file `readFile.fsx`. Convert this script into a function which reads the content of any text file and has the following type:

```
readText : filename:string -> string
```

(b) Write a program that converts a string, such that all letters are converted to lower case, and removes all characters except a...z. It should have the following type:

```
convertText : src:string -> string
```

(c) Write a program that counts occurrences of each lower-case letter of the English alphabet in a string and returns a list. The first element of the list should be the count of 'a's, second the count of 'b's etc. The function must have the type:

```
histogram : src:string -> int list
```

(d) The script `sampleAssignment.fsx` contains the function

```
randomString : hist:int list -> len:int -> string
```

which generates a string of a given length, and contains random characters distributed according to a given histogram. Modify the code to use your histogram function. Further, write a program, which reads the text `littleClausAndBigClaus.txt` using `readText`, converts it using `convertText`, and calculates its histogram and generates a new random string using `histogram` and `randomString`. Test the quality of your code by comparing the histograms of the two texts.

(e) Write a program that counts occurrences of each pairs of lower-case letter of the English alphabet in a string and returns a list of lists (a table). The first list should be the count of 'a' followed by 'a's, 'b's, etc., second list should be the count of 'b' followed by 'a's, 'b's, etc. etc.

```
cooccurrence : src:string -> int list list
```

(f) Write a program that generates a random string of length `len`, whose character pairs are distributed according to a user specified cooccurrence histogram `cooc`. The function must have the type:

```
fstOrderMarkovModel : cooc:int list list -> len:int -> string
```

Test your function by generating a random string, whose character pairs are distributed as the converted characters in H.C. Andersen's fairy tale, "Little Claus and Big Claus", calculate the cooccurrence histogram for the random string, and compare this with the original cooccurrence histogram.

4.4.1.10: Extend your library with a function

```
randomWords : wHist:Tree list -> nWords:int -> string
```

which generates a string with `nWords` number of words randomly selected to match the word-histogram in `wHist`.

4.4.1.11: The function `randomWords` is a random function, and you are to test its output by the histogram of the words in the string it produces.

Extend your library with the function,

```
diffw : t1:Tree list -> t2:Tree list -> double
```

which compares two word-histograms as the average sum of squared differences,

$$\text{diffw}(t_1,t_2) = \frac{1}{M}\sum_{i=0}^{M-1}(t_1(i)-t_2(i))^2 \tag{4.18}$$

where t_1 and t_2 are two word histograms, and M is the total number of different words observed in the two texts.

Extend your test file with a test of `randomWords` as follows:

(a) Convert The Story using `convertText` and calculate its word histogram.

(b) Use this to generate a random text using `randomWords` with M words, where M is the number of words in the converted The Story.

(c) Calculate the distance between the word histograms of The Story and the random texts using `diffw`.

(d) Set the test as passed if the difference is below some threshold.

You must choose a threshold in the above, and you must argue for your choice.

4.4.1.12: The function `randomWords` is a random function, and you are to test its output by the histogram of the words in the string it produces.

Extend your library with the function,

```
diffw : w1: wordHistogram -> w2:wordHistogram -> double
```

which compares two word-histograms as the average sum of squared differences,

$$\text{diffw}(w_1,w_2) = \frac{1}{M}\sum_{i=0}^{M-1}(w_1(i)-w_2(i))^2 \tag{4.19}$$

where w_1 and w_2 are two word histograms, and M is the total number of different words observed in the two texts.

Extend your test file with a test of `randomWords` as follows:

(a) Convert The Story using `convertText` and calculate its word histogram.

(b) Use this to generate a random text using `randomWords` with M words, where M is the number of words in the converted The Story.

(c) Calculate the distance between the word histograms of The Story and the random texts using `diffw`.

(d) Set the test as passed if the difference is below some threshold.

You must choose a threshold in the above, and you must argue for your choice.

4.4.1.13: Extend your library with a function

```
randomWords : wHist:wordHistogram -> nWords:int -> string
```

which generates a string with `nWords` number of words randomly selected to match the word-histogram in `wHist`.

4.4.1.14: In terms of a Markov Chain, what order is `randomWords`? Suggest an extension of `type Tree` to include first order Markov Chains. Speculate on whether this principle can be used to extend to n'th order mordels, and, in case, speculate on how the storage requirements will grow as n grows.

4.4.1.15: The script `readFile.fsx` reads the content of the text file `readFile.fsx`. Convert this script into a function which can read the content of any text file and has the following type:

```
readText : filename:string -> string
```

Add this function to your library.

4.4.1.16: Write a short report, which

- is no larger than 5 pages;
- includes answers to questions posed;
- contains a brief discussion on how your implementation works, and if there are any possible alternative implementations, and in that case, why you chose the one, you did;
- includes output that demonstrates that your solutions work as intented.

4.4.1.17: Extend your test file with a white-box test of `cooccurrence`.

4.4.1.18: Write a white-box test of the library functions `readText`, `convertText`, and `histogram`, and add these to your test file.

4.4.1.19: Write a program which reads The Story discard all characters that are not in `['a'..'z', 'A'..'Z',' ']`, convert all the remaining characters to lower case and calculate the occurrence of the remaining words as a `Tree list` type.

4.4.1.20: Write a white-box test of `wordHistogram`, and add it to your test file.

4.4.1.21: Extend your test file with a white-box test of `cooccurrenceOfWords`.

4.4.1.22: Write a program that counts occurrences of each triple of lower-case letter of the English alphabet in a string and returns a list of lists of lists. The program must include the function

```
triOccurrence : src:string -> int list list list
```

to calculate the number of occurrences of triples.

4.4.1.23: Extend your library with the function

```
type wordCooccurrences = (string * wordHistogram) list
cooccurrenceOfWords : src:string -> wordCooccurrences
```

which counts occurrences of each pair of words in a string and returns a list of pairs. Each returned pair must be a word and the wordHistogram of counts of cooccurrences. E.g., for the string "a hat and a cat" the function must return,

```
[("a", [("hat", 1); ("cat", 1)]);
 ("and", [("a", 1)]);
 ("cat", []);
 ("hat", [("and", 1)])]
```

4.4.1.24: Extend your library with a function that counts occurrences of each word in a string and returns a list. The counts must be organized as a list of trees using the following `Tree` type:

```
type Tree = Node of char * int * Tree list
```

An illustration of a value of this type is shown in Figure 4.1 Words are to be represented as

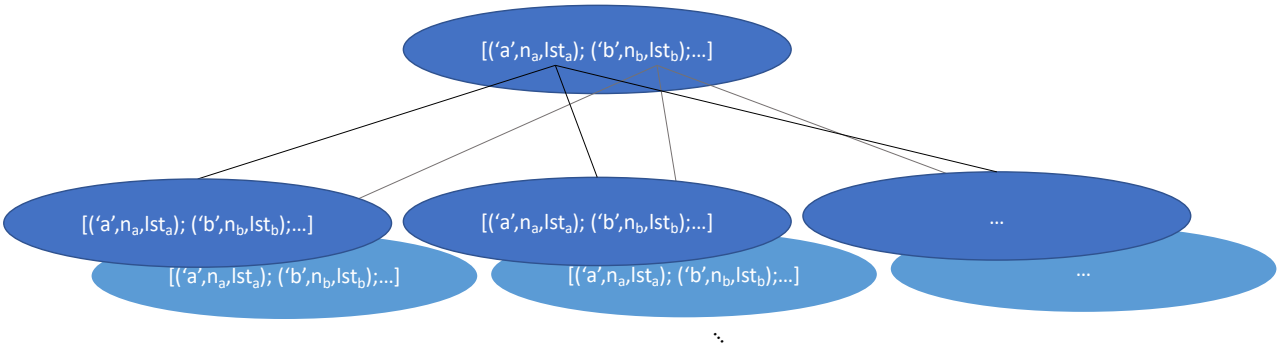


Figure 4.1: An illustration of a list of values of the type `Tree`.

the sequence of characters from the root til a node. The associated integer to each node counts the occurrence of a word ending in that node. Thus, if the count is 0, then no word with that endpoint has occurred. For example, a string with the words "a abc ba" should result in the following tree,

```
[Node ('a', 1, [Node ('b', 0, [Node ('c', 1, [])])]);
 Node ('b', 0, [Node ('a', 1, [])])]
```

Notice, the counts are zero for the combinations "ab" and "b", which are words not observed in the string. The function must have the type:

```
wordHistogram : src:string -> Tree list
```

4.4.1.25: The function `wordMarkovChain` is a random function, and you are to test its output by the cooccurrences of the words in the string it produces.

Extend your library with the function,

```
diffw2 : c1:wordCooccurrences -> c2:wordCooccurrences -> double
```

which compares two cooccurrence histograms as the average sum of squared differences,

$$\text{diffw2}(c_1, c_2) = \frac{1}{M^2} \sum_{i=0}^{M-1} \sum_{j=0}^{M-1} (c_1(i,j) - c_2(i,j))^2 \tag{4.20}$$

where c_1 and c_2 are two cooccurrence histograms of M elements such that $c_1(i,j)$ is the number of times word number i is found following word number j.

Extend your test file with a test of `wordMarkovChain` as follows:

(a) Convert The Story using `convertText` and calculate its cooccurrence histogram.

(b) Use this to generate a random text using `wordMarkovChain` with length M, where M is the length of the converted The Story.

(c) Calculate the distance between the cooccurrence histograms of The Story and the random texts using `diffw2`.

(d) Set the test as passed if the difference is below some threshold.

You must choose a threshold in the above, and you must argue for your choice.

4.4.1.26: Extend your library with a function that counts occurrences of each word in a string and returns a list. The list must be organized using the following type:

```
type wordHistogram = (string * int) list
```

For example, a string with the words "a abc ba ba" should result in the following list,

```
[("a", 1); ("abc", 1); ("ba",2)]
```

The function must have the type:

```
wordHistogram : src:string -> wordHistogram
```

4.4.1.27: Extend your library with a function

```
wordMarkovChain : wCooc: wordCooccurrences -> nWords:int -> string
```

which generates a random string with `nWords` words, whose word-pairs are distributed according to the cooccurrence histogram `wCooc`.

4.5 Imgutil

4.5.1 Teacher's guide

Emne rekursion, grafik og winforms

Sværhedsgrad Middel

4.5.2 Introduction

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`.

4.5.3 Exercise(s)

4.5.3.1: 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.

4.5.3.2: 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`).

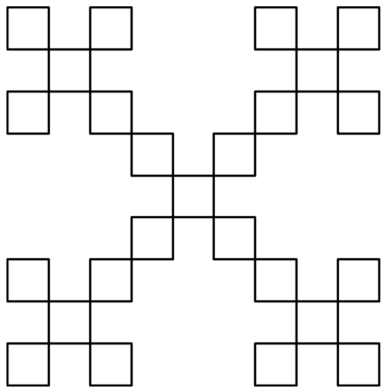
Funktionen `react`, som brugeren giver som 5. argument kaldes hver gang brugeren trykker på en tast. Funktionen tager som argument:

- en værdi svarende til den nuværende tilstand for applikationen, og
- et argument der kan benyttes til at afgøre hvilken tast der blev trykket på.[4]

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`.

4.5.3.3: Med udgangspunkt i øvelsesopgave 1 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 2.

4.6 Weekday

4.6.1 Teacher's guide

Emne Sumtyper og lister

Sværhedsgrad Let

4.6.2 Introduction

In the following, you are to work with the discriminated union `weekday`:

[4] 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.

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

which represents the days of the week.

4.6.3 Exercise(s)

4.6.3.1: Make a function `dayToNumber : weekday -> int` which given a `weekday` returns an integer, such that Monday is 1, Tuesday is 2, etc.

4.6.3.2: Make a function `nextDay : weekday -> weekday` which given a day returns the next day, i.e., Tuesday is the next day of Monday, and Monday is the next day of Sunday.

4.6.3.3: Make a function `numberToDay : n : int -> weekday option` which given an integer in the range $1 \ldots 7$ returns one of the weekdays Monday…Sunday as an option type. An integer not in the range, i.e. < 1 or > 7 should return `None`.
Examples:
The call `numberToDay 1` should return `Some Monday` and The call `numberToDay 42` should return `None`.

4.7 Sudoku

4.7.1 Teacher's guide

Emne Funktionsprogrammering

Sværhedsgrad Hård

4.7.2 Introduction

Temaet for ugernes opgaver er at programmere et Sudoku-spil. *Sudoku* er et puslespil, som er blevet opfundet uafhængigt flere gange; den tidligste "ægte" version af sudoku synes at kunne spores tilbage til det franske dagblad *Le Siècle* i 1892.

Vi betragter her kun den basale variant, som spilles på en matrix af 81 små felter, arrangeret i 9 rækker og 9 søjler. Matricen er desuden inddelt i 9 "bokse" eller "regioner", hver med 3 gange 3 felter.

Nogle af felterne er udfyldt på forhånd, og puslespillet går ud på at udfylde de resterende felter på en sådan måde, at hver af de 9 rækker, hver af de 9 søjler og hver af de 9 regioner kommer til at indeholde en permutation af symbolerne fra et forelagt alfabet af størrelse 9; vi vælger her (som man plejer at se det) alfabetet bestående af cifrene fra 1 til 9.

Her er en lovlig starttilstand for et spil sudoku:

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Og følgende er en vindende tilstand (en "løsning") af ovenstående:

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Nummerering og filformat

Lad os nummerere rækkerne $r = 0, 1, \ldots, 8$ ovenfra og ned og søjlerne $s = 0, 1, \ldots, 8$ fra venstre mod højre. Også regionerne vil vi nummerere $q = 0, 1, \ldots, 8$, i "normal læseretning" (for vestlige sprog).

Sammenhængen mellem rækkenummer r, søjlenummer s og regionsnummer q kunne så udtrykkes i følgende formel ved brug af heltalsoperationer:

- Feltet i række nummer r og søjle nummer s vil ligge i region nummer

$$q = r \texttt{ / 3 * 3 + } s \texttt{ / 3} \tag{4.21}$$

- Region nummer q består af felterne

$$(r,s) = (q \texttt{ / 3 * 3 + } m, q \texttt{ \% 3 * 3 + } n);\ m,n \in \{0,1,2\}. \tag{4.22}$$

En spiltilstand kan gemmes i en fil, ved at antage, at indholdet *altid* ser ud som følger:

- Der er mindst 90 tegn i filen (der kan være flere, men vi er kun interesseret i de 90 første)
- De første 90 tegn i filen er delt op i 9 grupper, som repræsenterer indholdet af række 0,..., 8 i nævnte rækkefølge. Hver gruppe består af 10 tegn: Først 9 tegn, som er et blandt `'1'`, ..., `'9'`, `'*'`, og til sidst strengen `"\n"`.

F.eks. er nedenstående indholdet i en fil, som indeholder starttilstanden for ovenstående sudoku:

```
53**7****\n6**195***\n*98****6*\n8***6***3\n4**8*3**1\n7***2***6\n*6****28*\n***419**5\n****8**79\n
```

Hvis vi fortolker strengen `"\n"` som "ny linje", bliver ovenstående lettere at læse:

```
53**7****
6**195***
*98****6*
8***6***3
4**8*3**1
7***2***6
*6****28*
***419**5
****8**79
```

4.7.3 Exercise(s)

4.7.3.1: I skal programmere et Sudoku spil og skrive en rapport. Afleveringen skal bestå af en pdf indeholdende rapporten, et katalog med et eller flere fsharp programmer som kan oversættes med Monos fsharpc kommando og derefter køres i mono, og en tekstfil der angiver sekvensen af oversættelseskommandoer nødvendigt for at oversætte jeres program(mer). Kataloget skal zippes og uploades som en enkelt fil. Kravene til programmeringsdelen er:

(a) Programmet skal kunne læse en (start-)tilstand fra en fil.

(b) Brugeren skal kunne indtaste filnavnet for (start-)tilstanden

(c) Brugeren skal kunne indtaste triplen (r,s,v), og hvis feltet er tomt og indtastningen overholder spillets regler, skal matrixen opdateres, og ellers skal der udskrives en fejlmeddelelse på skærmen

(d) Programmet skal kunne skrive matricens tilstand på skærmen (på en overskuelig måde)

(e) Programmet skal kunne foreslå lovlige tripler (r,s,v).

(f) Programmet skal kunne afgøre, om spillet er slut.

(g) Brugeren skal have mulighed for at afslutte spillet og gemme tilstanden i en fil.

(h) Programmet skal kommenteres ved brug af fsharp kommentarstandarden

(i) Programmet skal struktureres ved brug af et eller flere moduler, som I selv har skrevet

(j) Programmet skal unit-testes

Kravene til rapporten er:

(k) Rapporten skal skrives i LaTeX.

(l) I skal bruge `rapport.tex` skabelonen

(m) Rapporten skal som minimum i hoveddelen indeholde afsnittene Introduktion, Problemformulering, Problemanalyse og design, Programbeskrivelse, Afprøvning, og Diskussion og Konklusion. Som bilag skal I vedlægge afsnittene Brugervejledning og Programtekst.

(n) Alle gruppemedlemmer skal give feedback på et af hovedafsnittene i en anden gruppes rapport. Hvis og hvilke dele I gav feedback og hvem der gav feedback på jeres rapport skal skrives i Forordet i rapporten.

(o) Rapporten må maximalt være på 20 sider alt inklusivt.

Bemærk, at Sudoku eksemplerne i denne tekst er sat med LaTeX-pakken `sudoku`.

4.8 Mastermind

4.8.1 Teacher's guide

Emne Funktionsprogrammering

Sværhedsgrad Hård

4.8.2 Introduction

Spillet Mastermind er et spil for 2 deltagere, en opgavestiller og en opgaveløser. Opgavestilleren laver en skjult opgave bestående af en kombination af 4 farvede opgavestifter i ordnet rækkefølge, hvor farverne kan være rød, grøn, gul, lilla, hvid og sort. Opgaveløseren skal nu forsøge at gætte opgave. Dette gøres ved, at opgaveløseren foreslår en kombination af farver, og til hvert forslag svarer opgavestilleren med et antal hvide og sorte Svarstifter. Antallet af sorte Svarstifter svarer til hvor mange af opgavestifterne, som havde den rigtige farve og er på den rette plads, og antallet af hvide Svarstifter svarer til antallet af opgavestifter, som findes i opgaven men på den forkerte plads. F.eks. hvis opgaven er (rød, sort, grøn, gul), og gættet er (grøn, sort, hvid, hvid) er svaret 1 sort og 1 hvid.

4.8.3 Exercise(s)

4.8.3.1: I skal programmere spillet Mastermind. Minimumskrav til jeres aflevering er:

- Det skal være muligt at spille bruger mod bruger, program mod bruger i valgfrie roller og program mod sig selv.
- Programmet skal kommunikere med brugeren på engelsk.
- Programmet skal bruge følgende typer:

```
type codeColor =
  Red | Green | Yellow | Purple | White | Black
type code = codeColor list
type answer = int * int
type board = (code * answer) list
type player = Human | Computer
```

  hvor `codeColor` er farven på en opgavestift; `code` er en opgave bestående af 4 opgavestifter; `answer` er en tuple hvis første element er antallet af hvide og andet antallet af sorte stifter; og `board` er en tabel af sammenhørende gæt og svar.
- Programmet skal indeholde følgende funktioner:

```
    makeCode : player -> code
```

  som tage en spillertype og returnerer en opgave enten ved at få input fra brugeren eller ved at beregne en opgave.

```
    guess : player -> board -> code
```

  som tager en spillertype, et bræt bestående af et spils tidligere gæt og svar og returnerer et nyt gæt enten ved input fra brugeren eller ved at programmet beregner et gæt.

```
    validate : code -> code -> answer
```

  som tager den skjulte opgave og et gæt og returnerer antallet af hvid og sort svarstifter.
- Programmet skal kunne spilles i tekst-mode dvs. uden en grafisk brugergrænseflade.
- Programmet skal dokumenteres efter fsharp kodestandarden
- Programmet skal afprøves
- Opgaveløsningen skal dokumenteres som en rapport skrevet i LaTeX på maksimalt 20 oversatte sider ekslusiv bilag, der som minimum indeholder
  - En forside med en titel, dato for afleveringen og jeres navne
  - En forord som kort beskriver omstændighederne ved opgaven
  - En analyse af problemet
  - En beskrivelse af de valg, der er foretaget inklusiv en kort gennemgang af alternativerne
  - En beskrivelse af det overordnede design, f.eks. som pseudokode
  - En programbeskrivelse
  - En brugervejledning

  - En beskrivelse af afprøvningens opbygning
  - En konklusion
  - Afprøvningsresultatet som bilag
  - Programtekst som bilag

Gode råd til opgave er:

- Det er ikke noget krav til programmeringsparadigme, dvs. det står jer frit for om I bruger funktional eller imperativ programmeringsparadigme, og I må også gerne blande. Men overvej i hvert tilfælde hvorfor I vælger det ene fremfor det andet.
- For programmet som opgaveløser er det nyttigt at tænke over følgende: Der er ikke noget "intelligens"-krav, så start med at lave en opgaveløser, som trækker et tilfældigt gæt. Hvis I har mod på og tid til en mere advanceret strategi, så kan I overveje, at det totale antal farvekombinationer er $6^4 = 1296$, og hvert afgivne svar begrænser de tilbageværende muligheder.
- Summen af det hvide og sorte svarstifter kan beregnes ved at sammenlinge histogrammerne for de farvede stifter i hhv. opgaven og gættet: F.eks. hvis opgaven består af 2 røde og gættet har 1 rød, så er antallet af svarstifter for den røde farve 1. Ligeledes, hvis opaven består af 1 rød, og gættet består af 2 røde, så er antallet af svarstifter for den røde farve 1. Altså er antallet af svarstifter for en farve lig minimum af antallet af farven for opgaven og gættet for den givne farve, og antallet af svarstifter for et gæt er summen af minima over alle farver.
- Det er godt først at lave et programdesign på papir inden I implementerer en løsning. F.eks. kan papirløsningen bruges til i grove træk at lave en håndkøring af designet. Man behøver ikke at have programmeret noget for at afprøve designet for et antal konkrete situationer, såsom "hvad vil programmet gøre når brugeren er opgaveløser, opgaven er (rød, sort, grøn, gul) og brugeren indtaster (grøn, sort, hvid, hvid)?"
- Det er ofte sundt at programmere i cirkler, altså at man starter med at implementere en skald, hvor alle de væsentlige dele er tilstede, og programmet kan oversættes (kompileres) og køres, men uden at alle delelementer er færdigudviklet. Derefter går man tilbage og tilføjer bedre implementationer af delelementer som legoklodser.
- Det er nyttigt at skrive på rapporten under hele forløbet i modsætning til kun at skrive på den sidste dag.
- I skal overveje detaljeringsgraden i jeres rapport, da I ikke vil have plads til alle detaljer, og I er derfor nødt til at fokusere på de vigtige pointer.
- Husk at rapporten er et produkt i sig selv: hvis vi ikke kan læse og forstå jeres rapport, så er det svært at vurdere dens indhold. Kør stavekontrol, fordel skrive- og læseopgaverne, så en anden, end den der har skrevet et afsnit, læser korrektur på det.
- Det er bedre at aflevere noget end intet.
- Der er afsat 1 undervisningsfri og 3/2 alm. undervisningsuger til opgaven (7/11-30/11 fraregnet 14/11-20/11, som er mellemuge, og kursusaktiveter på parallelkurser). Det svarer til ca. 40 timers arbejde. Brug dem struktureret og målrettet. Lav f.eks. en tidsplan, så I ikke taber overblikket over projektforløbet.

4.9 Triangles

4.9.1 Teacher's guide

Emne Sumtyper

Sværhedsgrad Middel

4.9.2 Introduction

Til de følgende opgaver udvider vi typen `figure` med en mulighed for at repræsentere trekanter:

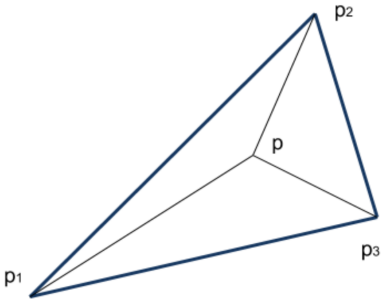
```
type figure =
  | Circle of point * int * colour
     // defined by center, radius, and colour
  | Rectangle of point * point * colour
     // defined by corners bottom-left, top-right, and colour
  | Mix of figure * figure
     // combine figures with mixed colour at overlap
  | Triangle of point * point * point * colour
     // defined by the three points and colour
```

Konstruktøren `Triangle` tager tre punkter og en farve som argument. Der er intet krav til hvordan de tre punkter er placeret i forhold til hinanden.

For at bestemme hvorvidt et punkt er placeret inde i en trekant benytter vi os af et trick der forudsætter at vi kan beregne arealet af en trekant ved at kende dens hjørnepunkter. Det viser sig at hvis en trekant er bestemt af punkterne $p_1 = (x_1, y_1)$, $p_2 = (x_2, y_2)$ og $p_3 = (x_3, y_3)$ vil følgende relativt simple formel kunne benyttes til at udregne arealet:

$$area = \left| \frac{x_1(y_2 - y_3) + x_2(y_3 - y_1) + x_3(y_1 - y_2)}{2} \right|$$

Tricket som vi nu skal benytte til at afgøre om et punkt p ligger inden i en trekant udspændt af hjørnerne p_1, p_2 og p_3 forklares lettest ved at iagttage figuren til højre. Såfremt arealet af de tre trekanter (p_1, p_2, p), (p_2, p_3, p), og (p_1, p_3, p) tilsammen er større end arealet af trekanten (p_1, p_2, p_3), da ligger punktet p udenfor trekanten (p_1, p_2, p_3); ellers ligger punktet indenfor trekanten.



4.9.3 Exercise(s)

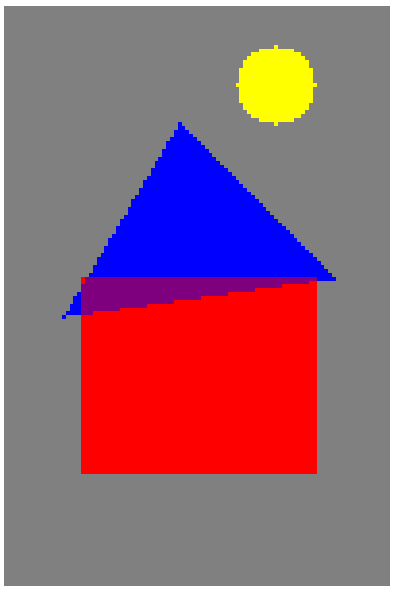
4.9.3.1: Lav en figur `figHouse : figure`, som består af en rød firkant udspændt af punkterne (20,70) og (80,120), en blå trekant udspændt af punkterne (15,80), (45,30) og (85,70), samt en gul cirkel med centrum (70,20) og radius 10.

4.9.3.2: Skriv en F# funktion `triarea2` der kan beregne den dobbelte værdi af arealet af en trekant udfra dens tre hjørnepunkter ved at benytte formlen ovenfor. Funktionen skal tage hjørnepunkterne som argumenter og have typen `point -> point -> point -> int`. Test funktionen på et par simple trekanter.[5]

4.9.3.3: Udvid funktionen `colourAt` til at håndtere trekantsudvidelsen ved at implementere tricket nævnt ovenfor samt ved at benytte den implementerede funktion `triarea2`.

4.9.3.4: Lav en fil `figHouse.png`, der viser figuren `figHouse` i et 100×150 bitmap.

Resultatet skulle gerne ligne figuren nedenfor.



4.9.3.5: Udvid funktionerne `checkFigure` og `boundingBox` fra øvelsesopgaverne til at håndtere udvidelsen.

`boundingBox houseFig` skulle gerne give `((15, 10), (85, 120))`.

4.10 Awari

4.10.1 Teacher's guide

Emne Typer, lister, mønstergenkendelse, funktionsprogrammering

Sværhedsgrad Hård

4.10.2 Introduction

I denne opgave skal I programmere spillet Awari, som er en variant af Kalaha. Awari er et gammelt spil fra Afrika, som spilles af 2 spillere, med 7 pinde og 36 bønner. Pindene lægges så der dannes 14

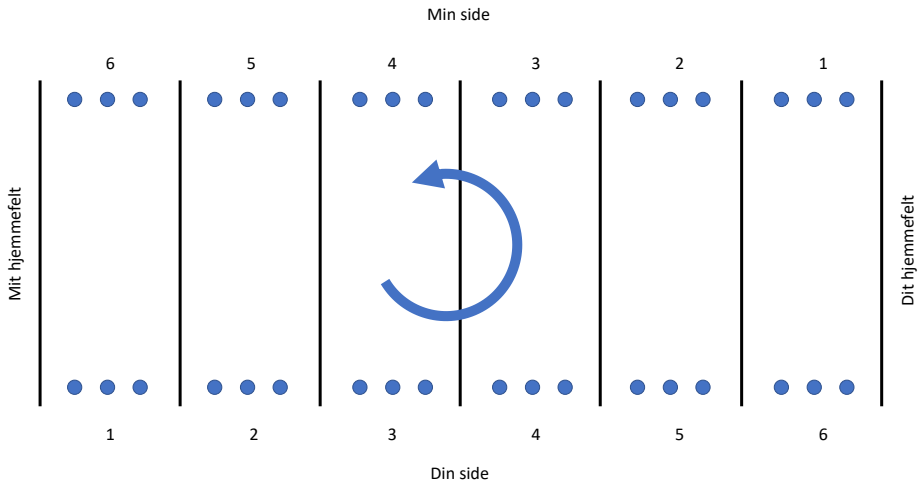


Figure 4.2: Udgangsopstillingen for spillet Awari.

felter ('pits' på engelsk), hvoraf 2 er hjemmefelter. Bønnerne fordeles ved spillet start med 3 i hvert felt pånær i hjemmefelterne. Startopstillingen er illustreret i Figur 4.2.

Spillerne skiftes til at spille en tur efter følgende regler:

- En tur spilles ved at spilleren tager alle bønnerne i et af spillerens felter 1-6 og placerer dem i de efterfølgende felter inkl. hjemmefelterne en ad gangen og mod uret. F.eks., kan første spiller vælge at tage bønnerne fra felt 4, hvorefter spilleren skal placere en bønne i hver af felterne 5, 6 og hjemmefeltet.
- Hvis sidste bønne lægges i spillerens hjemmefelt, får spilleren en tur til.
- Hvis sidste bønne lander i et tom felt som ikke er et hjemmefelt, og feltet overfor indeholder bønner, så flyttes sidste bønne til spillerens hjemmefelt, og alle bønnerne overfor fanges og flyttes ligså til hjemmefeltet.
- Spillet er slut når en af spillerne ingen bønner har i sine felter 1-6, og vinderen er den spiller, som har flest bønner i sit hjemmefelt.

4.10.3 Exercise(s)

4.10.3.1: (a) I skal implementere spillet Awari, som kan spilles af 2 spillere, og skrive en kort rapport. Kravene til jeres aflevering er:

- Koden skal organiseres som bibliotek, en applikation og en test-applikation.
- Biblioteket skal tage udgangspunkt i følgende signatur- og implementationsfiler:

[5]Det viser sig at være hensigtsmæssigt at undgå divisionen med 2, som kan forårsage uheldige afrundingsfejl.

Listing 1 awariLibIncompleteLowComments.fsi:
En ikke færdigskrevet signaturfil.

```
module Awari
type pit = // intentionally left empty
type board = // intentionally left empty
type player = Player1 | Player2

/// Print the board
val printBoard : b:board -> unit

/// Check whether a pit is the player's home
val isHome : b:board -> p:player -> i:pit -> bool

/// Check whether the game is over
val isGameOver : b:board -> bool

/// Get the pit of next move from the user
val getMove : b:board -> p:player -> q:string -> pit

/// Distributing beans counter clockwise,
/// capturing when relevant
val distribute :
    b:board -> p:player -> i:pit -> board * player *
   pit

/// Interact with the user through getMove to perform
/// a possibly repeated turn of a player
val turn : b:board -> p:player -> board

/// Play game until one side is empty
val play : b:board -> p:player -> board
```

Listing 2 awariLibIncomplete.fs:
En ikke færdigskrevet implementationsfil.

```
module Awari
type pit = // intentionally left empty
type board = // intentionally left empty
type player = Player1 | Player2

// intentionally many missing implementations and
   additions

let turn (b : board) (p : player) : board =
  let rec repeat (b: board) (p: player) (n: int) :
   board =
    printBoard b
    let str =
      if n = 0 then
        sprintf "Player %A's move? " p
      else
        "Again? "
    let i = getMove b p str
    let (newB, finalPitsPlayer, finalPit)= distribute
   b p i
    if not (isHome b finalPitsPlayer finalPit)
       || (isGameOver b) then
      newB
    else
      repeat newB p (n + 1)
  repeat b p 0

let rec play (b : board) (p : player) : board =
  if isGameOver b then
    b
  else
    let newB = turn b p
    let nextP =
      if p = Player1 then
        Player2
      else
        Player1
    play newB nextP
```

En version af signaturfilen med yderligere dokumentation og implementationsfilen findes i Absalon i opgaveområdet for denne opgave.

- Jeres løsning skal benytte funktionsparadigmet såvidt muligt.
- Koden skal dokumenteres vha. kommentarstandarden for F#
- Jeres aflevering skal indeholde en afprøvning efter white-box metoden.
- I skal skrive en kort rapport i LaTeX på maks. 10 sider og som indeholder:
  - en beskrivelse af jeres design og implementation
  - en gennemgang af jeres white-box afprøvning

– kildekoden som appendiks.

4.11 Peg Solitaire

4.11.1 Teacher's guide

Emne Classes, Objects, Methods Attributes

Sværhedsgrad Middel

4.11.2 Introduction

Peg Solitaire er et spil hvor en person skal forsøge at fjerne pinde i et bræt med huller ved at foretage en række træk indtil der kun er en pind tilbage på brættet. I hvert træk kan en pind fjernes ved at en nabopind flyttes over pinden til et ledigt hul. For hvert træk efterlades brættet med en pind færre.

I den klassiske engelske version af spillet Peg Solitaire består brættet af 33 huller, som til at starte med er fyldt med 32 pinde; det midterste hul i brættet er ikke udfyldt og opgaven består i at det netop er det midterste hul, der til slut skal indeholde en pind.

I denne opgave skal der arbejdes mod at få computeren til at finde en løsning til den engelske version af Peg Solitaire. En løsning vil bestå i at computeren udskriver de træk der skal flyttes. Opgaven er delt i tre dele. I den første delopgave arbejdes der mod at implementere et modul til at repræsentere en brætkonstellation samt operationer til at foretage flytninger og derved danne nye brætkonstellationer. I den anden delopgave skal der arbejdes mod at gøre det muligt for en spiller at spille spillet ved brug af `fsharpi` således at computeren tillader at der foretages træk hvorefter den nye brætkonstellation udskrives. I den tredie delopgave skal der skrives en algoritme, som returnerer en liste af træk, der efterlader en enkelt pind i midten af brættet.

4.11.3 Exercise(s)

4.11.3.1: Modulet `Board`.

Hvert hul i brættet er identificeret ved en position (r,c), hvor r er rækken for hullet (se billedet til højre) og c er kolonnen hullet optræder i. Således er positionen for den tomme plads i midten $(3,3)$.
I det følgende skal vi benytte os af 64-bit heltal til at indeholde en komplet brætkonstellation (vi gør kun brug af de 49 mindstbetydende bit).

	0	1	2	3	4	5	6
0			•	•	•		
1			•	•	•		
2	•	•	•	•	•	•	•
3	•	•	•		•	•	•
4	•	•	•	•	•	•	•
5			•	•	•		
6			•	•	•		

I F# kan et bræt således repræsenteres ved brug af typen `uint64`, der repræsenterer (unsigned) 64-bit heltal:

```
type b = uint64
```

I den første del af opgaven ønskes der implementeret en række funktioner til at operere på brætkonstellationer. Funktionerne ønskes implementeret i et modul `Board`, som vil kunne bruges både af en rigtig spiller til at spille spillet og af et modul der har til hensigt at finde en løsning til spillet.

Modulet `Board` skal indeholde følgende typer og funktioner:

```
type b                                  // board type
type pos = int * int                    // position type
type dir = Up | Down | Left | Right     // move direction
type mv = pos * dir                     // move

val init     : unit -> b                // initial board
val valid    : pos -> bool              // is the position valid?
val peg      : b -> pos -> bool         // true if pos valid and
                                        //  hole contains a peg
val mv       : b -> mv -> b option      // returns new board
val pegcount : b -> int                 // number of pegs
val print    : b -> string              // string representation
```

Her følger nogle gode råd til hvordan ovenstående modul implementeres:

- Start med at implementere to hjælpefunktioner `seti` og `geti` til henholdsvis at sætte en givet bit i en `uint64`-værdi samt at undersøge om en givet bit er sat (hertil skal I benytte et udvalg af bit-operationer, inklusiv `|||`, `&&&`, `~~~`, `>>>` og `<<<`).
- Implementér en hjælpefunktion `posi` til at omdanne en position (row-column pair) til et bit-index i brætrepræsentationen.
- Funktionen `valid` skal returnere `false` hvis positionen ikke repræsenterer en hul-position i et tomt bræt.
- Implementér en funktion `neighbor` af type `pos -> dir -> pos option`, som, givet en valid position og en retning, returnerer en valid naboposition, hvis en sådan findes i den specificerede retning, eller værdien `None`. Et kald `neighbor(1,4)Right` skal returnere værdien `None` og et kald `neighbor(2,4)Right` skal returnere værdien `Some(2,5)`.
- Funktionen `mv` kan nu implementeres ved brug af funktionerne `peg`, `neighbor`, `seti` og `posi`. Funktionen skal, givet en brætkonstellation `b` og et træk `(p,d)` returnere værdien `Some b'`, hvis (1) `p` er en position indeholdende en pind ifølge brætkonstellationen `b`, (3) `(p,d)` er et lovligt træk og (3) brættet `b'` er den konstellation, der fremkommer ved trækket `(p,d)`. Ellers skal funktionen returnere værdien `None`.
- For at implementere funktionen `print` kan der benyttes to nestede rekursive funktioner (eller to nestede for-løkker), som hver itererer over henholdsvis rækkerne og kolonnerne på brættet.

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

4.11.3.2: Modulet `Game`.

Implementér et modul `Game` med følgende funktionalitet:

```
val start : unit -> unit
val mv    : Board.mv -> unit
```

Modulet skal indeholde en reference til en "nuværende" brætkonstellation. Funktionen `start` skal sætte den nuværende brætkonstellation til det initielle bræt og udskrive brættet. Funktionen `mv` skal foretage en flytning (hvis det er lovligt) og udskrive den nye brætkonstellation.

I rapporten skal I demonstrere brugen af modulet `Game` i `fsharpi`.

4.11.3.3: Modulet `Solve`.

I denne delopgave skal der skrives en funktion `solve`, som foretager en udtømmende søgning efter en række træk som vil efterlade brættet i en konstellation med kun en pind, placeret i midten af brættet. Funktionen kan passende have følgende type:

```
type state = Board.b * Board.mv list
val solve : state -> Board.mv -> state option
```

Her består en tilstand af "den nuværende brætkonstellation" samt en liste af de træk der leder frem til denne konstellation (med det seneste træk forekommende først i listen). Funktionen tager yderligere et træk som argument. Ved at det gøres muligt at iterere gennem alle mulige træk (for hver konstellation), fra det første træk `((0,0),Right)` til det sidste træk `((6,6),Up)`, kan vi sikre at alle træk prøves. Funktionen skal benytte sig af rekursion til at foretage den (muligvis) udtømmende søgning. Givet en vilkårlig brætkonstellation samt en kandidat til et træk kan funktionen `solve` undersøge om trækket vil efterlade brættet i en ny konstellation eller om trækket ikke er gyldigt. Afhængigt af udfaldet kan det enten undersøges (ved eventuelt rekursivt at kalde `solve`) om den nye brætkonstellation har (eller er) en løsning eller om vi har bedre held med det næste træk i trækordningen (hvis et sådan træk findes).

For at implementere funktionen er det nyttigt først at implementere nogle hjælpefunktioner:

(a) Skriv en funktion `nextdir` af type `dir -> dir option`, som "roterer" en retningsværdi således at `Up` bliver til `Some Right`, `Right` bliver til `Some Down`, `Down` bliver til `Some Left` og `Left` bliver til `None`.

(b) Skriv en funktion `nextpos` af type `pos -> pos option`, som returnerer den næste position på et 7×7 hullers bræt (row-major). Et kald `nextpos(2,5)` skal returnere værdien `Some(2,6)` og et kald `nextpos(1,6)` skal returnere værdien `Some(2,0)`.

(c) Skriv en funktion `nextmv` af type `mv -> mv option`, som passende benytter sig af de to ovenfor specificerede funktioner. Funktionen skal give mulighed for at iterere gennem alle mulige flytninger, startende med flytningen `((0,0),Up)`. Bemærk at funktionen skal operere uden hensyn til en konkret brætkonstellation og at funktionen ikke skal tage højde for de præcise forekomster af huller i brættet (flytningerne kan senere filtreres blandt andet ved brug af funktionen `valid`). Således skal et kald `nextmv((1,2),Down)` returnere værdien `Some((1,2),Left)`, et kald `nextmv((1,6),Left)` skal returnere værdien `Some((2,0),Up)`. Endelig skal kaldet `nextmv((6,6),Left)` returnere værdien `None`.

I rapporten skal I vise koden for jeres implementation af den rekursive funktion `solve` og argumentere for at den finder en løsning til brætspillet, såfremt en sådan findes. Skriv også kode til at udskrive de fundne træk og vis i rapporten at jeres implementation finder en løsning til spillet i form af en liste af træk.

Rapporten skal også indeholde en beskrivelse af implementationens begrænsninger samt en reflektion over hvordan implementationen kan generaliseres til at finde løsninger til andre brætspecifikationer.

4.12 Levensthein

4.12.1 Teacher's guide

Emne Strenge, rekursion, tests, caching

Sværhedsgrad Svær

4.12.2 Introduction

Du skal i de følgende to opgaver arbejde med en funktion til at bestemme den såkaldte *Levenstheindistance* mellem to strenge a og b. Distancen er defineret som det mindste antal editeringer, på karakter-niveau, det er nødvendigt at foretage på strengen a før den resulterende streng er identisk med strengen b. Som editeringer forstås (1) sletninger af karakterer, (2) indsættelser af karakterer, og (3) substitution af karakterer.

Varianter af Levensthein-distancen mellem to strenge kan således benyttes til at identificere om studerende selv har løst deres indleverede opgaver eller om der potentielt set er tale om plagiatkode ;)

Matematisk set kan Levensthein-distancen $leven(a,b)$, mellem to karakterstrenge a og b, defineres som $lev_{a,b}(|a|,|b|)$, hvor $|a|$ og $|b|$ henviser til længderne af henholdsvis a og b, og hvor funktionen lev er defineret som følger:[6]

$$lev_{a,b}(i,j) = \begin{cases} \max(i,j) & \text{if } \min(i,j) = 0, \\ \min \begin{cases} lev_{a,b}(i-1,j)+1 \\ lev_{a,b}(i,j-1)+1 \\ lev_{a,b}(i-1,j-1)+1_{(a_i \neq b_j)} \end{cases} & \text{otherwise.} \end{cases}$$

hvor $1_{(a_i \neq b_j)}$ henviser til *indikatorfunktionen*, som er 1 når $a_i \neq b_j$ og 0 ellers.

4.12.3 Exercise(s)

4.12.3.1: Implementér funktionen *leven* direkte efter den matematiske definition (ved brug af rekursion) og test korrektheden af funktionen på nogle små strenge, såsom "house" og "horse"(distance 1) samt "hi" og "hej" (distance 2).

4.12.3.2: Den direkte implementerede rekursive funktion er temmelig ineffektiv når strengene a og b er store. F.eks. tager det en del millisekunder at udregne distancen mellem strengene "dangerous house" and "danger horse". Årsagen til denne ineffektivitet er at en løsning der bygger direkte på den rekursive definition resulterer i en stor mængde genberegninger af resultater der allerede er beregnet.

For at imødekomme dette problem skal du implementere en såkaldt "caching mekanisme" der har til formål at sørge for at en beregning højst foretages en gang. Løsningen kan passende gøre brug af gensidig rekursion og tage udgangspunkt i løsningen for den direkte rekursive definition (således skal løsningen nu implementeres med to gensidigt rekursive funktioner `leven`

[6] See `https://en.wikipedia.org/wiki/Levenshtein_distance`.

og `leven_cache` forbundet med `and`). Som cache skal der benyttes et 2-dimensionelt array af størrelse $|a| \times |b|$ indeholdende heltal (initielt sat til -1).

Funktionen `leven_cache`, der skal tage tilsvarende argumenter som `leven`, skal nu undersøge om der allerede findes en beregnet værdi i cachen, i hvilket tilfælde denne værdi returneres. Ellers skal funktionen `leven` kaldes og cachen opdateres med det beregnede resultat. Endelig er det nødvendigt at funktionen `leven` opdateres til nu at kalde funktionen `leven_cache` i hver af de rekursive kald.

Test funktionen på de små strenge og vis at funktionen nu virker korrekt også for store input.

Det skal til slut bemærkes at den implementerede løsning benytter sig af $O(|a| \times |b|)$ plads og at der findes effektive løsninger der benytter sig af mindre plads ($O(\max(|a|,|b|))$). Det er ikke et krav at din løsning implementerer en af disse mere pladsbesparende strategier.

4.13 Polynomials

4.13.1 Teacher's guide

Emne Højere-ordens funktioner, currying

Sværhedsgrad Middel

4.13.2 Introduction

I det følgene skal I arbejde med polynomier. Et polynomium af grad n skrives som

$$f(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n = \sum_{i=0}^{n} a_i x^i.$$

4.13.3 Exercise(s)

4.13.3.1: Skriv en funktion `poly: float list -> float -> float`, der tager som argumenter (1) en liste `a` af koefficienter med `a.[i]` $= a_i$ og (2) en x-værdi for derefter at returnere polynomiets værdi. Afprøv funktionen ved at lave tabeller for et lille antal polynomier af forskellig grad med forskellige koefficienter og forskellige værdier for x, og validér den beregnede værdi.

4.13.3.2: Definer en funktion `line : float -> float -> float -> float` ved brug af `poly`, således at `line a0 a1 x` beregner værdien for et 1. grads polynomium hvor `a0` $= a_0$, `a1` $= a_1$ og `x` $= x$. Afprøv funktionen ved at tabellere værdier for `line` med det samme sæt af koefficienter $a_0 \neq 0$ og $a_1 \neq 0$ og et passende antal værdier for x.

4.13.3.3: Benyt partiel anvendelse af `line` til at definere en funktion `theLine : float -> float`, hvor parametrene `a0` og `a1` er sat til det samme som brugt i Opgave 2. Afprøv `theLine` tilsvarende som `line` afprøves i Opgave 2.

4.13.3.4: Lav en funktion `lineA0 : float -> float` ved brug af `line`, men hvor `a1` og `x` holdes fast (funktionen `lineA0` tager således kun en a_0 værdi som argument). Diskutér om funktionen kan implementeres ved Currying uden brug af hjælpefunktioner? Hvis ikke, foreslå en hjælpefunktion, som vil gøre en definition vha. Currying mulig.

4.14 Integration

4.14.1 Teacher's guide

Emne Højere-ordens funktioner

Sværhedsgrad Middel

4.14.2 Introduction

De følgende opgaver omhandler integration. Integralet af næsten alle integrable funktioner kan approximeres som

$$\int_a^b f(x)\,dx \simeq \sum_{i=0}^{n-1} f(x_i)\Delta x,$$

hvor $x_i = a + i\Delta x$ og $\Delta x = \frac{b-a}{n}$.

4.14.3 Exercise(s)

4.14.3.1: Skriv en funktion `integrate : n:int -> a:float -> b:float -> (f : float -> float) -> float`, hvis argumenter `n`, `a`, `b`, er som i ligningerne, og `f: float -> float` er en F#-funktion, som repræsenterer en integrabel én-dimensionel funktion. Afprøv `integrate` på `theLine` fra Opgave 3 og på `cos` med $a = 0$ og $b = \pi$. Udregn integralerne analytisk og sammenlign med resultatet af `integrate`.

4.14.3.2: Funktionen `integrate` er en approximation, og præcisionen afhænger af n. Undersøg afhængigheden ved at udregne fejlen, dvs. forskellen mellem det analytiske resultat og approximationen for værdier af n. Dertil skal du lave to funktioner `integrateLine : n:int -> float` og `integrateCos : n:int -> float` vha. `integrate`, `theLine` og `cos`, hvor værdierne for a og b og f er fastlåste. Afprøv disse funktioner for $n = 1, 10, 100, 1000$. Overvej om der er en tendens i fejlen, og hvad den kan skyldes.

4.15 Exceptions

4.15.1 Teacher's guide

Emne Untagelser og option typen

Sværhedsgrad Let

4.15.2 Introduction

Denne opgave omhandler undtagelser (exceptions), option typer og Stirlings formel. Stirlings formel er en approximation til fakultetsfunktionen via

$$\ln n! \simeq n \ln n - n.$$

4.15.3 Exercise(s)

4.15.3.1: Implement the factorial function $n! = \prod_{i=1}^{n} i$, $n > 0$ as `fac : n:int -> int`. The function must cast a `System.ArgumentException` exception, if the function is called with $n < 1$. Call `fac` with the values $n = -4, 0, 1, 4$, and catch possible exceptions.

4.15.3.2: Add a new and selfdefined exception `ArgumentTooBig of string` to `fac` in Assignment 1, and cast it with the argument `"calculation would result in an overflow"`, when n is too large for the `int` type. Call the function with a small and a large value of n, catch the possible exception and handle it in case by writing the exception message to the screen.

4.15.3.3: Make a new factorial function `facFailwith : n:int -> int`, as `fac` in Assignment 2, but where the 2 exceptions are replaced with `failwith` with the arguments `"argument must be greater than 0"` and `"calculation would result in an overflow"` respectively. Call `facFailWith` with $n = -4, 0, 1, 4$, catch possible exceptions with the `Failure` pattern, and write the returned message from `failwith` to the screen.

4.15.3.4: Write a new factorial function as in Assignment 2 but with the name and type `facOption : n:int -> int option`, which returns `Some m`, when the result is computable and `None` otherwise. Call `fac` with the values $n = -4, 0, 1, 4$, and write the result to the screen.

4.15.3.5: Write a function `logIntOption : n:int -> float option`, which calculates the logarithm of `n`, if `n` > 0 and `None` otherwise. Test `logIntOption` for the values `-10`, `0`, `1`, `10`.

4.15.3.6: Write a function `logFac : int -> float option` which calculates $\log(n!)$ by combining `logIntOption` and `facOption` and using `Option.bind`. Compare the output of `logFac` with Stirlings approximation $\log(n!) \simeq n \log n - n \log e$, where $e = 2.718281\ldots$ is the natural exponential base and for $n = 1, 2, 4, 8$.

4.15.3.7: The function `logFac : int -> float option` can be defined in many ways as a combination of the functions `Some`, `Option.bind`, `logIntOption`, and `facOption`. Write 3 single-line statements, which uses `|>`, `>>` or none of them.

4.15.3.8: Make implementations of the following functions:

```
safeIndexIf : arr:'a [] -> i:int -> 'a
safeIndexTry : arr:'a [] -> i:int -> 'a
safeIndexOption : arr:'a [] -> i:int -> 'a option
```

Each of them must return the value of `arr` at index `i`, when `i` is a valid index, and otherwise handle the error-situation. The error-situations must be handled in different ways:

- `safeIndexIf` must not make use of `try-with` and must not cast an exception.

- `safeIndexTry` must use `try-with`, and it must call `failwith` when there is an error.
- `safeIndexOption` must return `None` in case of an error.

Make a short test of all 3 functions, by writing the content of an array to the screen (and not as an option type). The tests must also include examples of error situations and must be able to handle possible exceptions casted. In your opinion, is any of the above method superior or inferior in how they handle errors and why?

4.16 Tree structure

4.16.1 Teacher's guide

Emne Træstrukturer og grafik

Sværhedsgrad Middel

4.16.2 Introduction

I de følgende opgaver skal vi arbejde med en træstruktur til at beskrive geometriske figurer med farver. For at gøre det muligt at afprøve jeres opgaver skal I gøre brug af det udleverede bibliotek `img_util.dll`, der blandt andet kan omdanne såkaldte canvas-objekter til png-filer. Biblioteket er beskrevet i forelæsningerne (i kursusuge 7) og koden for biblioteket er tilgængeligt via github på `https://github.com/diku-dk/img-util-fs`.

Her bruger vi funktionerne til at tegne på et canvas samt til at gemme canvas-objektet som en png-fil:[7]

```
// colors
type color
val fromRgb : int * int * int -> color

// canvas
type canvas
val mk       : int -> int -> canvas
val setPixel : color -> int * int -> canvas -> unit

// save a canvas as a png file
val toPngFile : string -> canvas -> unit
```

Funktionen `toPngFile` tager som det første argument navnet på den ønskede png-fil (husk extension). Det andet argument er canvas-objektet som ønskes konverteret og gemt. Et canvas-objekt kan konstrueres med funktionen `ImgUtil.mk`, der tager som argumenter vidden og højden af billedet i antal pixels, samt funktionen `ImgUtil.setPixel`, der kan bruges til at opdatere canvas-objektet før det eksporteres til en png-fil. Funktionen `ImgUtil.setPixel` tager tre argumenter. Det første argument repræsenterer en farve og det andet argument repræsenterer et punkt i canvas-objektet (dvs. i

[7] Bemærk at interfacet ikke definerer de konkrete repræsentationstyper for typerne `color` og `canvas`. Disse typer er holdt *abstrakte*, hvilket vil sige at deres repræsentationer ikke kan ses af brugeren af modulet.

billedet). Det tredie argument repræsenterer det canvas-objekt, der skal opdateres. En farve kan nu konstrueres med funktionen `ImgUtil.fromRgb` der tager en triple af tre tal mellem 0 og 255 (begge inklusive), der beskriver hhv. den røde, grønne og blå del af farven.

Koordinatsystemet har nulpunkt $(0,0)$ i øverste venstre hjørne og, såfremt vidden og højden af koordinatsystemet er henholdsvis w og h, optræder punktet $(w-1,h-1)$ i nederste højre hjørne. Antag for eksempel at programfilen `testPNG.fsx` indeholder følgende F# kode:

```
let C = ImgUtil.mk 256 256
do ImgUtil.setPixel (ImgUtil.fromRgb (255,0,0)) (10,10) C
do ImgUtil.toPngFile "test.png" C
```

Det er nu muligt at generere en png-fil med navn `test.png` ved at køre følgende kommando:

```
fsharpi -r img_util.dll testPNG.fsx
```

Den genererede billedfil `test.png` vil indeholde et hvidt billede med et pixel af rød farve i punktet (10,10).

Bemærk, at alle programmer, der bruger `ImgUtil` skal køres eller oversættes med `-r img_util.dll` som en del af kommandoen.

Bonus information, hvis I på et tidpunkt skulle få brug for at inkludere png-filer, fx skabt vha `ImgUtil`, i et LaTeX dokument, så gøres det med LaTeX kommandoen `\includegraphics`.

I det følgende vil vi repræsentere geometriske figurer med følgende datastruktur:

```
type point = int * int // a point (x, y) in the plane
type color = ImgUtil.color

type figure =
  | Circle of point * int * color
     // defined by center, radius, and color
  | Rectangle of point * point * color
     // defined by corners top-left, bottom-right, and color
  | Mix of figure * figure
     // combine figures with mixed color at overlap
```

Man kan, for eksempel, lave følgende funktion til at finde farven af en figur i et punkt. Hvis punktet ikke ligger i figuren, returneres `None`, og hvis punktet ligger i figuren, returneres `Some` c, hvor c er farven.

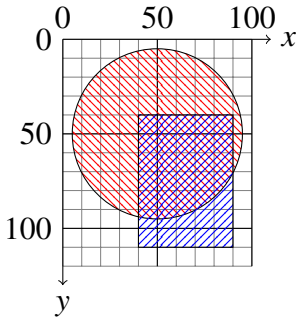
```
// finds color of figure at point
let rec colorAt (x,y) figure =
  match figure with
  | Circle ((cx,cy), r, col) ->
      if (x-cx)*(x-cx)+(y-cy)*(y-cy) <= r*r
        // uses Pythagoras' equation to determine
        // distance to center
      then Some col else None
  | Rectangle ((x0,y0), (x1,y1), col) ->
     if x0<=x && x <= x1 && y0 <= y && y <= y1
        // within corners
```

```
    then Some col else None
| Mix (f1, f2) ->
    match (colorAt (x,y) f1, colorAt (x,y) f2) with
    | (None, c) -> c  // no overlap
    | (c, None) -> c  // no overlap
    | (Some c1, Some c2) ->
    let (a1,r1,g1,b1) = ImgUtil.fromColor c1
    let (a2,r2,g2,b2) = ImgUtil.fromColor c2
    in Some(ImgUtil.fromArgb((a1+a2)/2, (r1+r2)/2,   // calculate
                             (g1+g2)/2, (b1+b2)/2))  // average
 color
```

Bemærk, at punkter på cirklens omkreds og rektanglens kanter er med i figuren. Farver blandes ved at lægge dem sammen og dele med to, altså finde gennemsnitsfarven.

4.16.3 Exercise(s)

4.16.3.1: Lav en figur `figTest : figure`, der består af en rød cirkel med centrum i (50,50) og radius 45, samt en blå rektangel med hjørnerne (40,40) og (90,110), som illustreret i tegningen nedenfor (hvor vi dog har brugt skravering i stedet for udfyldende farver.)



4.16.3.2: Brug `ImgUtil`-funktionerne og `colorAt` til at lave en funktion

`makePicture : string -> figure -> int -> int -> unit`

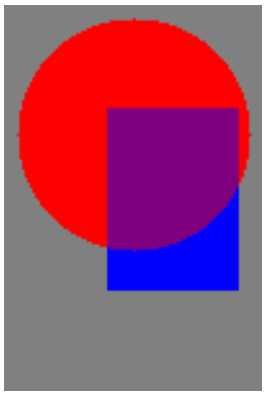
sådan at kaldet `makePicture` *`filnavn figur b h`* laver en billedfil ved navn *`filnavn`*`.png` med et billede af *`figur`* med bredde *`b`* og højde *`h`*.

På punkter, der ingen farve har (jvf. `colorAt`), skal farven være grå (som defineres med RGB-værdien (128,128,128)).

Du kan bruge denne funktion til at afprøve dine opgaver.

4.16.3.3: Brug funktionen `makePicture` til at konstruere en billedfil med navnet `figTest.png` og størrelse 100×150 (bredde 100, højde 150), der viser figuren `figTest` fra Opgave 1.

Resultatet skulle gerne ligne figuren nedenfor.



4.16.3.4: Lav en funktion `checkFigure : figure -> bool`, der undersøger, om en figur er korrekt: At radiusen i cirkler er ikke-negativ og at øverste venstre hjørne i en rektangel faktisk er ovenover og til venstre for det nederste højre hjørne (bredde og højde kan dog godt være 0).

4.16.3.5: Lav en funktion `move : figure -> int * int -> figure`, der givet en figur og en vektor flytter figuren langs vektoren.

Ved at foretage kaldet

```
makePicture "moveTest" (move figTest (-20,20)) 100 150
```

skulle der gerne laves en billedfil `moveTest.png` med indholdet vist nedenfor.

4.16.3.6: Lav en funktion `boundingBox : figure -> point * point`, der givet en figur finder hjørnerne (top-venstre og bund-højre) for den mindste akserette rektangel, der indeholder hele figuren.

Funktionskaldet `boundingBox figTest` skulle gerne give resultatet `((5, 5), (95, 110))`.

4.17 Cat

4.17.1 Teacher's guide

Emne working with files

Sværhedsgrad Easy

4.17.2 Introduction

The program `cat` is a UNIX-program, which concatenates (i.e. joins) files. The program exists on both Linux and macOS. When passing two text files to `cat`, e.g. `a.txt` and `b.txt`, then the program prints the contents of file `a.txt` followed by the contents of `b.txt` to the screen. Consider an inverse version of `cat`, called `tac`, which prints the files in reverse order and prints each file from the last to the first character. For example, if the file `a.txt` contains the characters `abc\ndef\n` and the file `b.txt` contains the characters `123\n456\n` with `\n` being the newline character, then

```
cat a.txt b.txt
```

will output `abc\ndef\n123\n456\n` to the screen. In contrast,

```
tac a.txt b.txt
```

will output `654\n321\nfed\ncba\n` to the screen.

In the following assignments you are to write a (functional) implementation of `cat` and `tac` in F#.

4.17.3 Exercise(s)

4.17.3.1: Make the library `readNWrite.fs` with the function,

```
readFile : filename:string -> string option
```

which takes a filename and returns the contents of the text file as a `string option`. If the file does not exist, the function should return `None`.

4.17.3.2: Make a function,

```
printFile : filenname:string -> bool
```

which prints the content of the file with the name `filename` to the screen. If no error occurs, then the function must return `true`, and otherwise `false`. The function should be placed in the implementation-file `readNWrite.fs`.

4.17.3.3: First extend the library `readNWrite.fs` with a function,

```
cat : filenames:string list -> string option
```

which takes a list of filenames. The function should use `readFile` (Exercise 1) to read the contents of the files. The contents of the files should be merged into a single `string option`, which the function returns. If any of the files do not exist, then the function should return `None`.

Then write an application, `cat`, which takes a list of filenames as command-line arguments, calls the `cat` function with this list and prints the resulting string to the screen. The program must return 1 in case of an error and 0 otherwise.

4.17.3.4: First extend the library `readNWrite.fs` with a function,

```
tac : filenames:string list -> string option
```

which takes a list of files, reads their content with `readFile` (Exercise 1), reverses the order of each file in a line-by-line manner and reverses each line (i.e. the opposite of `cat`) and concatenates the result. If any of the files do not exist, then the function should return `None`.

Then write an application, `tac`, which takes a list of filenames as command-line arguments, calls the `tac` function with this list and prints the resulting string to the screen. The program must return 0 or 1 depending on whether the operation was successful or not.

4.18 Sum types

4.18.1 Teacher's guide

4.18.2 Introduction

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

4.18.3 Exercise(s)

4.18.3.1: 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.

4.18.3.2: 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.

4.19 Rotate

4.19.1 Teacher's guide

Emne Rekursion og lister

Sværhedsgrad Middel

4.19.2 Introduction

In this assignment, you will be working with a puzzle called Rotate. The puzzle consists of a square chess-like board with $n \times n, n \in \{2,3,4,5\}$ fields. Each field has a unique id-number, which we will

call the field's position, and, for a particular configuration of the board, each field is associated with a unique letter from the alphabet 'a', 'b', ... For example, with $n = 4$, here is a possible board configuration:

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

Moreover, the positions of the fields are laid out as follows:

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

The puzzle is solved by rotating the letters in small 2×2 subsquares clockwise until the board reaches the *solved* configuration:

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

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 $p_1, p_2, p_3, p_4 \rightarrow q_1, q_2, q_3, q_4$ denote a rotation from p_* to q_*, where p_1 is the top-left corner. Then, a rotation of subsquare 1 results in $1, 2, 5, 6 \rightarrow 5, 1, 6, 2$, or equivalently,

h	o	l	k		b	h	l	k
b	i	g	e	$\rightarrow$	i	o	g	e
f	m	c	a		f	m	c	a
j	n	d	p		j	n	d	p

The overall task of this assignment is to build a program that generates rotate-puzzles and that allows you iteratively to enter a sequence of positions until the puzzle is solved.

Here is a list of detailed requirements:

- If your program includes loops, the loops must be programmed using recursion.
- Your program must use lists and not arrays.
- Your program is not allowed to use mutable values (or variables).
- Your solution must be parameterized by n, the size of the board.

- You must represent your board as a list of letters. Thus, for $n = 4$, the board for a solved puzzle must be identical to the list `['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` and `rotate.fs` must constitute the interface and the implementation of a library with your main types, functions, and values; the file `game.fsx` must be a maximally 10-line program that defines the value n and starts the game; and `whiteboxtest.fsx` and `blackboxtest.fsx` must contain your tests for the library.

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

Notice that calls to `System.Random ()` returns a random number generator object. This object has a method `Next : n:int -> int`, which draws a random non-negative integer less than `n`. For example, the code

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

prints 3 random non-negative integers less than 10.

4.19.3 Exercise(s)

4.19.3.1: Write the interface file for the library `rotate`. The interface should specify two user-defined types, named `Board` and `Position`, which are defined to be a list of characters and an integer, respectively. The interface should also specify the following functions:

```
create : n:uint -> Board
board2Str : b:Board -> string
validRotation : 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 as argument and return an $n \times n$ board in its solved state.

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

The function `validRotation` must take a board and a rotation position as arguments 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 rotation position as arguments and return a board identical to the original but where a local 2×2 rotation has been performed at the indicated position. If an invalid position is given, the function must return the board that was passed as the first argument.

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

The function `solved` must take a board as argument and return true or false depending on whether the board is in the solved configuration.

The interface must include documentation following the documentation standard.

4.19.3.2: Write a program `blackboxtest.fsx` that performs a blackbox test of the yet to be implemented `rotate` library.

4.19.3.3: Write the implementation file of the `rotate` library.

4.19.3.4: Write a program `whiteboxtest.fsx` that performs a whitebox test of the `rotate` library.

4.19.3.5: Write a short program `game` that defines the size of the board n, starts the game, and implements the interaction with the user in a game-loop using recursion.

4.19.3.6: A fellow programmer has made the function

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

which takes as argument a non-negative integer m and returns either the empty list or a list of rotation positions (of length m or less) that will leave the board in the solved configuration (if one such list exists). 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?

4.20 Route finding

4.20.1 Teacher's guide

Emne rekursion, grafik og winforms

Sværhedsgrad Middel

4.20.2 Introduction

In the following you are to work with the movement of a small robot in two dimensions. The robot can be placed on integer positions `type pos = int*int`, and in each step, it can move one position up, down, left, or right.

4.20.3 Exercise(s)

4.20.3.1: (a) Given a source and target grid point, write the function

```
dist: p1: pos -> p2: pos -> int
```

which calculates the squared distance between positions p_1 and p_2. I.e., if $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$ then $\text{dist}(p_1, p_2) = (x_2 - x_1)^2 + (y_2 - y_1)^2$.

(b) Given a source and a target and `dist`, write the function

```
candidates: src: pos -> tg: pos -> pos list
```

which returns the list of candidate next positions, which brings the robot closer to its target. I.e., if $\text{src} = (x, y)$, then the function must consider all the neighbouring positions, $\{(x+1, y), (x-1, y), (x, y+1), (x, y-1)\}$, and return those whose distance is equal to or smaller than $\text{dist}(\text{src}, \text{tg})$. This can be done with `List.filter`.

(c) Given a source and a target and by use of `candidates` the above functions, write a recursive function

```
routes: src: pos -> tg: pos -> pos list list
```

which calculates the list of all the shortest routes from `src` to `tg`. For example, the list of shortest routes from $(3,3)$ to $(1,1)$ are

```
[[(3, 3); (2, 3); (1, 3); (1, 2); (1, 1)];
 [(3, 3); (2, 3); (2, 2); (1, 2); (1, 1)];
 [(3, 3); (2, 3); (2, 2); (2, 1); (1, 1)];
 [(3, 3); (3, 2); (2, 2); (1, 2); (1, 1)];
 [(3, 3); (3, 2); (2, 2); (2, 1); (1, 1)];
 [(3, 3); (3, 2); (3, 1); (2, 1); (1, 1)]]
```

Beware, this list grows fast, the further the source and target is from each other, so you will be wise to only work with short distances. This can be done with a recursive function and a `List.map` of a `List.map`.

(d) Consider now a robot, which also can move diagonally. Extend `candidate` to also consider the diagonal positions $\{(x+1, y+1), (x+1, y-1), (x-1, y+1), (x-1, y-1)\}$, and update `routes` to return the list of shortest routes only. For example, the shortest routes from $(3,4)$ to $(1,1)$ should be

```
[[(3, 4); (2, 3); (1, 2); (1, 1)];
 [(3, 4); (2, 3); (2, 2); (1, 1)];
 [(3, 4); (3, 3); (2, 2); (1, 1)]]
```

but not necessarily in that order.

(e) Optional: Make a Canvas program, which draws routes, and apply it to the routes found above.

4.21 Vector v.2

4.21.1 Teacher's guide

4.21.2 Introduction

This assignment is about 2-dimensional vectors. A 2-dimensional vector or just a vector is a geometric object consisting of a direction and a length. Typically, vectors are represented as a coordinate pair

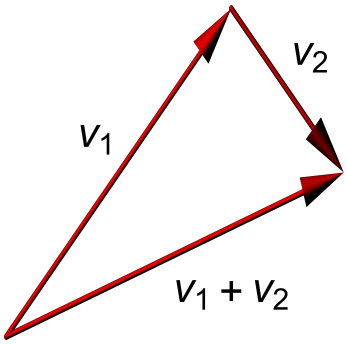


Figure 4.3: Illustration of vector addition in two dimensions.

$\vec{v} = (x, y)$. The vector's ends are called its tail and tip, and when the tail is placed in $(0,0)$, then its tip will be in the (x, y). Vectors have a number of standard operations on them:

$$\vec{v}_1 = (x_1, y_1) \tag{4.23}$$
$$\vec{v}_2 = (x_2, y_2) \tag{4.24}$$
$$\vec{v}_1 + \vec{v}_2 = (x_1 + x_2, y_1 + y_2) \tag{4.25}$$
$$a\vec{v}_1 = (ax_1, ay_1) \tag{4.26}$$

Addition can be drawn as shown in Figure 5.1. Rotation of a vector counter-clockwise around its tail by a can be done as,

$$R_a\vec{v}_1 = (x\cos(a) - y\sin(a), x\sin(a) + y\cos(a)) \tag{4.27}$$

In F#, the trigonometric functions are found in `cos` and `sin`, and they both take an angle in radians as the argument. The constant π is found in `System.Math.PI`. In the following, we will use the type abbreviation:

```
type vec = float * float
```

4.21.3 Exercise(s)

4.21.3.1: Using Canvas, you are to draw vectors. For this,

(a) Make a function

```
toInt: vec -> int * int
```

which takes a vector of floats and returns a vector of ints.

(b) Using `add` and `toInt`, make a function

```
setVector: canvas -> color -> vec -> vec -> unit
```

which takes a canvas, a color, a vector `v`, and a position `p` and draws a line from `p` to `p+v` using `setLine`. Demonstrate that this works by creating a horizontal vector with its tail at the center of the canvas, and show it on screen using `show`.

(c) Using `rot` and `setVector` make a function

```
draw: int -> int -> canvas
```

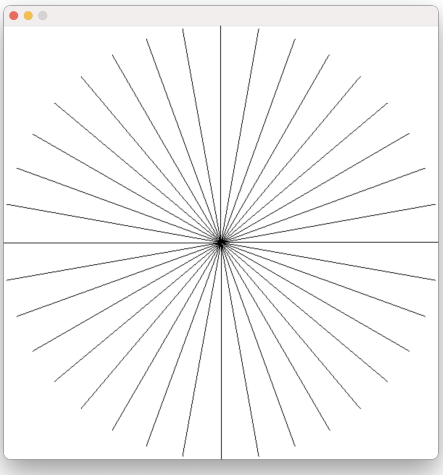


Figure 4.4: 36 radial lines from the center of a canvas.

which creates a canvas with a given width and height, adds 36 spokes as illustrated in Figure 4.4, and returns the canvas. Demonstrate that this works by showing the canvas on screen with `show`.

(d) Optional: Use these in `runApp` to make an interactively rotating set of spokes as follows: Extend `draw` with a float state parameter `s`, which draws the spokes with the angular offset `s`. Add a reaction function `react` which changes the offset by ± 0.01 when the right and left arrow key are pressed respectively.

The functions are to be documented using the `<summary>`, `<param>`, and `<returns>` XML tags.

4.21.3.2: In the following, you are to work with tuples.

(a) Make a type abbreviation called `vec3`, which is a 3-tuple of `floats`.

(b) Make a value of `vec3`.

(c) Make a function, which takes a `vec3` as argument and returns the squared sum of its elements, and test it on the value, you created. Consider the different ways, the function's type could be written, and what their qualitative differences would be.

4.21.3.3: In an earlier assignment, you implemented a small set of functions for vector operations in F#:

(a) addition of vectors

```
add: vec -> vec -> vec
```

(b) multiplication of a vector and a floating-point number

```
mul: vec -> float -> vec
```

(c) rotation of a vector by radians

```
rot: vec -> float -> vec
```

and wrote a small program to test these functions. Convert your earlier programs into a library called `Vec`, consisting of a signature file, an implementation file, and an application file. Create a `.fsproj` project file, and run the code using first `dotnet fsi` and then `dotnet run`.

4.21.3.4: Building upon the exercises concerning vectors and their representation as tuples, we will construct a small library of functions for 2-dimensional vector operations, with vectors represented as tuples of floats.

(a) Write a function:

```
scaleFloatVec : float * float -> scalar:float -> float * float
```

that given a vector of floats and a scalar, scales the vector and returns the resulting vector.

(b) Write a function:

```
addFloatVecs : float * float -> float * float -> float * float
```

that given two vectors of floats, adds them and returns the resulting vector.

(c) Write a function:

```
subFloatVecs : float * float -> float * float -> float * float
```

that given two vectors of floats, subtracts them with vector subtraction and returns the resulting vector.

(d) Write a function:

```
lengthFloatVec : float * float -> float
```

that given a vectors of floats, computes the length of the vector and returns the result as a float.

(e) Write a function:

```
dotFloatVecs : float * float -> float * float -> float
```

that given two vectors of floats, computes the dot product and returns the result as a float.

4.21.3.5: Using Steps 1, 3, 5, 7, and 8 from the 8-step guide to write a small set of functions in F#:

(a) addition of vectors (4.25)

```
add: vec -> vec -> vec
```

(b) multiplication of a vector and a constant (4.26)

```
mul: vec -> float -> vec
```

(c) rotation of a vector (4.27)

```
rot: vec -> float -> vec
```

The functions are to be documented using the `<summary>`, `<param>`, and `<returns>` XML tags.

In the following, you are to work with the abstract datatype known as a *queue*. A queue is a a sequence of elements that supports the following operations: checking whether the sequence is empty; removing an element from the front ("left"); adding an element at the end ("right"). Queues appear often in real life: The line[8] waiting for service at a shop counter, orders to be filled in a warehouse, students waiting to be examined at an oral examination.

The abstract datatype of *purely functional queues* consists of:

[8] In American English. Called indeed *queue* in British English.

- the types named `element` and `queue`, where `element` is the type of elements and `queue` the type of queues with such elements;
- the following value and two functions

```
// the empty queue
emptyQueue: queue
// add an element at the end of a queue
enqueue: element -> queue -> queue
// remove and return the element at the front of a queue
// precondition: input queue is not empty
dequeue: queue -> element * queue
// check if a queue is empty
isEmpty: queue -> bool
```

- the properties these operations must satisfy and that another programmer can rely on, such as queuing an element on an empty queue and then dequeuing from it yields the element added at first and leaves an empty queue behind; or, more generally, first repeatedly queuing elements and then dequeuing until the queue is empty yields the same elements.[9]

These queues are called *(purely) functional* because the enqueue and dequeue operations return a *new* queue whenever they are called, without destroying the old queue. For example, adding an element e_1 to a queue q_0 of length 15 results in a queue q_1 of length 16; then adding another element e_2 to q_0 also results in a queue q_2 of length 16, but one that is different from q_1 in its last element. At this point we have 3 separate queues, each of which we can used in future operations: q_0, q_1 and q_2.[10]

In this exercise, you will work with functional queues in F#. We'll omit writing "functional" below.

4.22 Queue

4.22.1 Teacher's guide

Emne rekursion, grafik og winforms

Sværhedsgrad Middel

4.22.2 Introduction

In the following, you are to work with the abstract datatype known as a *queue*. A queue is a a sequence of elements that supports the following operations: checking whether the sequence is empty; removing an element from the front ("left"); adding an element at the end ("right"). Queues appear

[9] This can be expressed as a precise mathematical property, which in turn can be used to systematically test one's implementation to find errors. We'll do this only informally here.

[10] There are also *ephemeral* (also called *imperative*) queues, where enqueue and replace the original queue with the new queue such that there is always just one "current" queue that changes over time. Ephemeral queues have more limited functionality and are easier to implement efficiently using imperative data structures, which we will encounter later in the course.

often in real life: The line[11] waiting for service at a shop counter, orders to be filled in a warehouse, students waiting to be examined at an oral examination.

The abstract datatype of *purely functional queues* consists of:

- the types named `element` and `queue`, where `element` is the type of elements and `queue` the type of queues with such elements;
- the following value and two functions

```
// the empty queue
emptyQueue: queue
// add an element at the end of a queue
enqueue: element -> queue -> queue
// remove and return the element at the front of a queue
// precondition: input queue is not empty
dequeue: queue -> element * queue
// check if a queue is empty
isEmpty: queue -> bool
```

- the properties these operations must satisfy and that another programmer can rely on, such as queuing an element on an empty queue and then dequeuing from it yields the element added at first and leaves an empty queue behind; or, more generally, first repeatedly queuing elements and then dequeuing until the queue is empty yields the same elements.[12]

These queues are called *(purely) functional* because the enqueue and dequeue operations return a *new* queue whenever they are called, without destroying the old queue. For example, adding an element e_1 to a queue q_0 of length 15 results in a queue q_1 of length 16; then adding another element e_2 to q_0 also results in a queue q_2 of length 16, but one that is different from q_1 in its last element. At this point we have 3 separate queues, each of which we can used in future operations: q_0, q_1 and q_2.[13]

In this exercise, you will work with functional queues in F#. We'll omit writing "functional" below.

4.22.3 Exercise(s)

4.22.3.1: In the following you are to create a dotnet project for the assignment.

(a) Using the `dotnet` command line tool, create a new `F#` console application named `5i`.
`dotnet new console -lang "F#" -o 5i`

(b) Rename the file `Program.fs` to `testQueues.fs`.

(c) Update `5i.fsproj` to compile `testQueues.fs` instead of `Program.fs`.

(d) Ensure the project works by running `dotnet build` and `dotnet run`.

[11] In American English. Called indeed *queue* in British English.

[12] This can be expressed as a precise mathematical property, which in turn can be used to systematically test one's implementation to find errors. We'll do this only informally here.

[13] There are also *ephemeral* (also called *imperative*) queues, where enqueue and replace the original queue with the new queue such that there is always just one "current" queue that changes over time. Ephemeral queues have more limited functionality and are easier to implement efficiently using imperative data structures, which we will encounter later in the course.

4.22.3.2: In the following, you are to implement your own queue module using lists in F# to represent the (sequence of elements in) a queue. The module is to be called `IntQueue`.

(a) Given the description of the abstract datatype Queue above, write a signature file `intQueue.fsi` for the functional queues.

(b) Write an implementation file `intQueue.fs`, implementing the signature file above using lists in F# where the elements are F# integers.

(c) Add `intQueue.fsi` and `intQueue.fs` to `5i.fsproj`, so they are compiled *before* `testQueues.fs`.

(d) In `testQueues.fs`, show your implementation works by using your queue. As a minimum, you should add the following series of tests of your `IntQueue` module:
changed from v1: negate the result of `IntQueue.isEmpty q1` so `nonEmptyTestResult` is `true` when the implementation of `IntQueue.isEmpty` is correct.

```
let intQueueTests () =
    let q0 = IntQueue.emptyQueue
    let emptyTestResult = IntQueue.isEmpty q0
    emptyTestResult
    |> printfn "An empty queue is empty: %A"

    let e1,e2,e3 = 1,2,3
    let q1 = q0 |> IntQueue.enqueue e1
                |> IntQueue.enqueue e2
                |> IntQueue.enqueue e3
    let nonEmptyTestResult = not (IntQueue.isEmpty q1)

    nonEmptyTestResult
    |> printfn "A queue with elements is not empty: %A"

    let (e,q2) = IntQueue.dequeue q1
    let dequeueTestResult = e = e1
    dequeueTestResult
    |> printfn "First in is first out: %A"

    let allTestResults =
        emptyTestResult &&
        nonEmptyTestResult &&
        dequeueTestResult

    allTestResults
    |> printfn "All IntQueue tests passed: %A"
    // Return the test results as a boolean
    allTestResults

// Run the IntQueue tests
let intQueueTestResults = intQueueTests ()
```

4.22.3.3: A problem with the queue specification above is that there is a precondition on the dequeue operation: A programmer must always ensure that the argument to dequeue is nonempty before calling dequeue. In other words, even though the F# type system does not flag it as an error, it *is* an error (by the programmer) to call dequeue with the empty queue.

In the following, you are to implement another version of your queue module using lists in F# to represent the (sequence of elements in) a queue, with error handling. The module is to be called `SafeIntQueue`, the signature file `safeIntQueue.fsi` and the implementation file `safeIntQueue.fs`.

Change the queue specification such that `SafeIntQueue.dequeue` returns an `(element option) * queue` value and remove the precondition. Add the new module to `5i.fsproj` and in `testQueues.fs`, add a corresponding test suite that shows your implementation works; that is, its operations perform queuing and dequeuing, and, additionally `SafeIntQueue.dequeue(emptyQueue)` returns `None`.

4.22.3.4: A queue is an abstract datatype, and as such the operations of a queue should not depend on the type of elements. In the following you are to implement a *generic queue*, so it is possible to create queues of any type, e.g. queues of `int`, queues of `float`, queues of `string` or even queues of `queue`.

The module is to be called `Queue`, the signature file `queue.fsi` and the implementation file `queue.fs`. Add the new module to `5i.fsproj` and in `testQueues.fs`, add a corresponding test suite that shows your implementation works; that is, its operations perform queuing and dequeuing, and additionally that you can build queues of different types with the same generic library. As a minimum, demonstrate queues of `int`, `float` and `string`.

4.23 Leaf trees

4.23.1 Teacher's guide

4.23.2 Introduction

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

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

A term of the form `Leaf $x$` is called a *leaf* (or *leaf node*); a term of the form `Tree $(t1, t2)$` is called a *tree node*. The tree type is generic in the type of data that in the leaf nodes.

4.23.3 Exercise(s)

4.23.3.1: 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.

4.23.3.2: Write a function

```
find : pred: ('a -> bool) -> 'a tree -> 'a option
```

that returns the first value in preorder that satisfies the provided predicate `pred`. If no such value appears in the tree, the function should return the value `None`.

Test whether your function works as specified.

4.23.3.3: 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.

4.24 Higher-order Functions

4.24.1 Teacher's guide

4.24.2 Introduction

De følgende opgaver omhandler højereordens funktioner. Specifikt handler opgaverne om brugen af pipe-funktionerne (`|>` og `<|`) samt af de højereordens funktioner, der kan benyttes til funktionssammensætning (>> og <<).

4.24.3 Exercise(s)

4.24.3.1: Skriv en funktion `sumsq : float list -> float`, som benytter sig af en `let`-binding til at definere en variabel `squares`. Variablen `squares` skal bindes til resultatet af at konstruere en liste indeholdende kvadratet på hvert element i argumentlisten (benyt `List.map`). Funktionen `sumsq` kan nu returnere resultatet af at udregne udtrykket `List.fold (+) 0.0 squares`. Test funktionen på en række forskellige float-lister. For eksempel skal kaldet `sumsq [3.0;4.0]` resultere i værdien `25.0`.

4.24.3.2: Opskriv typerne for de fire funktioner `|>`, `<|`, << og >>:

```
val |> : ____________________________________   // right-pipe

val <| : ____________________________________   // left-pipe

val << : ____________________________________   // left-compose

val >> : ____________________________________   // right-compose
```

4.24.3.3: Omdefinér funktionen `sumsq : float list -> float` således at den benytter sig af F#'s **right-pipe funktion** `|>` til at pipe resultatet af `List.map` udtrykket ind i `List.fold` udtrykket. Test funktionen på en række forskellige float-lister.

4.24.3.4: Omdefinér funktionen `sumsq : float list -> float` således at den benytter sig af F#'s **left-pipe funktion** `<|` til at pipe resultatet af `List.map` udtrykket ind i `List.fold` udtrykket, men fra højre mod venstre. Test funktionen på en række forskellige float-lister.

4.24.3.5: Omdefinér funktionen `sumsq : float list -> float` således at den benytter sig af F#'s **right-compose funktion** >> til at sammensætte (1) en funktion defineret ved at kalde `List.map` med en funktion samt (2) en funktion defineret ved at kalde `List.fold` med to argumenter. Definitionen skal følge formen:

```
let sumsq : float list -> float =
  ... >> ...
```

Test funktionen på en række forskellige float-lister.

4.24.3.6: Omdefinér funktionen `sumsq : float list -> float` således at den benytter sig af F#'s **left-compose funktion** `<<` til at sammensætte (1) en funktion defineret ved at kalde `List.map` med en funktion samt (2) en funktion defineret ved at kalde `List.fold` med to argumenter. Definitionen skal følge formen:

```
let sumsq : float list -> float =
  ... << ...
```

Bemærk at F#'s left-compose funktion svarer til matematisk funktionssammensætning givet ved $(f \circ g)(x) = f(g(x))$. Test funktionen på en række forskellige float-lister.

4.24.3.7: Betragt funktionen

```
let sumadd2sq (xs : float list) : float =
  let ys = List.map (fun x -> x*x) xs
  let zs = List.map (fun y -> y+2.0) ys
  in List.fold (+) 0.0 zs
```

Omskriv funktionen ved at benytte ligningen `map f << map g = map (f << g)` til at undgå at listen `ys` konstrueres. Omskriv funktionen yderligere ved enten at benytte en pipe-funktion eller funktionssammensætning til at undgå `let`-bindingen af variablen `zs`. Test funktionen på en række forskellige float-lister. For eksempel skal kaldet `sumadd2sq [2.0;3.0]` resultere i værdien `17.0`.

4.25 Forest

4.25.1 Teacher's guide

Emne rekursion, grafik og winforms

Sværhedsgrad Middel

4.25.2 Introduction

In the following we are going to work with lists and Canvas. The module Canvas has the ability to perform simple turtle graphics. To draw in turtle graphics, we command a little invisible turtle, which moves on the canvas with a pen. The function `turtleDraw` is given a list of `turtleCmds`, such as `PenUp` and `PenDown` to raise and lower the pen, `Turn 250` and `Move 100` to turn 250 degrees and move 100 pixels, and `SetColor red` to pick a red pen. For example in Figure 4.5, the function `tree sz` creates the set of turtle commands for drawing a fractal tree of size `sz` and returns the turtle to the starting point. The command `turtleDraw` executes the list of turtle commands, which in this case draws the tree on a canvas and displays it. In this exercise, you are to work with turtle commands.

```
#r "nuget:diku.canvas, 1.0.1"
open Canvas

let rec tree (sz: int) : Canvas.turtleCmd list =
   if sz < 5 then
     [Move sz; PenUp; Move (-sz); PenDown]
   else
     [Move (sz/3); Turn -30]
     @ tree (sz*2/3)
     @ [Turn 30; Move (sz/6); Turn 25]
     @ tree (sz/2)
     @ [Turn -25; Move (sz/3); Turn 25]
     @ tree (sz/2)
     @ [Turn -25; Move (sz/6)]
     @ [PenUp; Move (-sz/3); Move (-sz/6); Move (-sz/3)]
     @ [Move (-sz/6); PenDown]

let w = 600
let h = w
let sz = 100
turtleDraw (w,h) "Tree" (tree sz)
```

Figure 4.5: A turtle graphics program for drawing a fractal tree in Canvas.

4.25.3 Exercise(s)

4.25.3.1: Consider a list of pairs of integers `(dir,dist)` which are to be translated into a list of turtle commands, such that `[(10, 30); (-5, 127); (20, 90)]` is translated into `[Turn 10; Move 30; Turn -5; Move 127; Turn 20; Move 90]`. You are to

(a) Write the following functions:

```
type move = int*int // a pair of turn and move
fromMoveRec: lst: move list -> Canvas.turtleCmd list
fromMoveMap: lst: move list -> Canvas.turtleCmd list
fromMoveFold: lst: move list -> Canvas.turtleCmd list
fromMoveFoldBack: lst: move list -> Canvas.turtleCmd list
```

where

i. `fromMoveRec` is a recursive function, that does not use the List module

ii. `fromMoveMap` is non-recursive function, which uses `List.map`

iii. `fromMoveFold` is non-recursive function, which uses `List.fold`

iv. `fromMoveFoldBack` is non-recursive function, which uses `List.foldBack`

In some of the above functions, you may also find it useful to use `List.concat` and `List.rev`.

(b) Demonstrate that these 4 functions produce the same result given identical input.

4.25.3.2: (a) The following program

```
let rnd = System.Random()
let v = rnd.Next 10
```

makes a random integer between the integer $0 \leq v < 10$. Use this to make a function

```
randomTree: sz: int -> Canvas.turtleCmd list
```

which calls `tree sz` and concatenates further turtle commands to place a tree randomly on a canvas, and return the turtle to the origin. Test your function by calling `turtleDraw` with such a list.

(b) Write a recursive function

```
forest: sz: int -> n: int -> Canvas.turtleCmd list
```

which makes n random trees on the canvas by calling `randomTree` n times. Test your function by calling `turtleDraw` with such a list.

4.26 Expression trees

4.26.1 Teacher's guide

4.26.2 Introduction

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

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

For eksempel, the expression $(5+8)*9$ is represented by

```
Mul (Add (Const 5, Const 8), Const 9)
```

4.26.3 Exercise(s)

4.26.3.1: 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 (Mul (Add (Const 5, Const 8), Const 9))` should return the integer value `117`.

4.26.3.2: Extend the type `expr` with cases for subtraction and division, think about the type of the evaluator extended expression language: What should be the result when dividing by zero? Modify and extend your evaluator to include subtraction and division, and test whether your implementation works in practice.

4.26.3.3: 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.

4.27 Cards

4.27.1 Teacher's guide

4.27.2 Introduction

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
```

4.27.3 Exercise(s)

4.27.3.1: 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.

4.27.3.2: 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.

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

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

4.27.3.5: 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)`.

4.27.3.6: 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`.

4.27.3.7: 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`.

4.28 2048

4.28.1 Teacher's guide

Emne rekursion, grafik og winforms

Sværhedsgrad Middel

4.28.2 Introduction

2048 is a popular solitaire board game available, e.g., online on `https://2048.io/`. In this exercise, you are to implement a version of this game in F# and using `Canvas`. The rules, you are to implement are as follows:

4.28.2.1: The board is a square board with 3×3 field.

4.28.2.2: The pieces are colored squares: red, green, blue, yellow, black corresponding to the values 2, 4, 8, 16, 32.

4.28.2.3: There can at most be one piece per field on the board.

4.28.2.4: The initial conditions is a red and a blue piece placed on the board.

4.28.2.5: The game can be tilted left, right, up, down by pressing the corresponding arrow keys.

4.28.2.6: When the game is tilted, then all the pieces are to be moved to the corresponding side of the board.

4.28.2.7: If two pieces of the same color are pushed into each other in the process of tilting, then they are replaced by a single piece of double the value. For example, the board in Figure 4.6(a) is tilted to the right, and the two red pieces are replaced with green piece. However, replacement is not performed in a cascading fashion, e.g., the board in Figure 4.6(c) is tilted to the right combining the two green to a blue, but the resulting two blues are not combined.

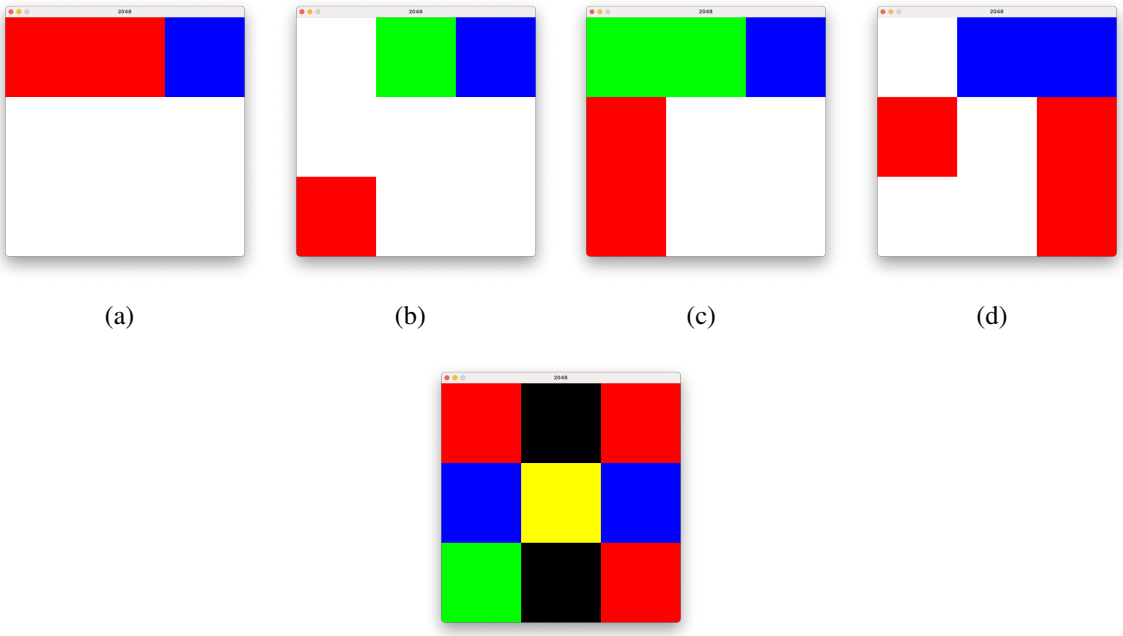
4.28.2.8: Two black pieces are combined into one black piece.

4.28.2.9: After each turn, a new red piece is to be placed randomly on an available field on the board.

4.28.2.10: The game ends when there are no possible moves and no empty locations for a new piece to spawn.

In your solution, you are to represented a board with its pieces as a list of pieces, where each piece has a color and a position. This is captured by the following type abbreviations:

```
type pos = int*int // A 2-dimensional vector in board-coordinats (not
   pixels)
type value = Red | Green | Blue | Yellow | Black // piece values
type piece = value*pos //
type state = piece list // the board is a set of randomly organized
   pieces
```



(a) (b) (c) (d)

(e)

Figure 4.6: Some examples

In the following, the first coordinate in pos will thought of as a up-down axis also called the row, and the second as an left-right axis also called the column with $(0,0)$ being the top-left.

4.28.3 Exercise(s)

4.28.3.1: Make a library consisting of a signature and an implementation file. The library must contain the following functions

```
// convert a 2048-value v to a canvas color E.g.,
// > fromValue Green;;
// val it: color = { r = 0uy
//   g = 255uy
//   b = 0uy
//   a = 255uy }
fromValue: v: value -> Canvas.color

// give the 2048-value which is the next in order from c, e.g.,
// > nextColor Blue;;
// val it: value = Yellow
// > nextColor Black;;
// val it: value = Black
nextColor: c: value -> value

// return the list of pieces on a column k on board s, e.g.,
// > filter 0 [(Blue, (1, 0)); (Red, (0, 0))];;
// val it: state = [(Blue, (1, 0)); (Red, (0, 0))]
// > filter 1 [(Blue, (1, 0)); (Red, (0, 0))];;
// val it: state = []
```

```
filter: k: int -> s: state -> state

// tilt all pieces on the board s to the left (towards zero on
// the first coordinate), e.g.,
// > shiftUp [(Blue, (1, 0)); (Red, (2, 0)); (Black, (1,1))];;
// val it: state = [(Blue, (0, 0)); (Red, (1, 0)); (Black, (0,
   1))]
shiftUp: s: state -> state

// flip the board s such that all pieces position change as
// (i,j) -> (2-i,j), e.g.
// > flipUD [(Blue, (1, 0)); (Red, (2, 0))];;
// val it: state = [(Blue, (1, 0)); (Red, (0, 0))]
flipUD: s: state -> state

// transpose the pieces on the board s such all piece positiosn
// change as (i,j) -> (j,i), e.g.,
// > transpose [(Blue, (1, 0)); (Red, (2, 0))];;
// val it: state = [(Blue, (0, 1)); (Red, (0, 2))]
transpose: s: state -> state

// find the list of empty positions on the board s, e.g.,
// > empty [(Blue, (1, 0)); (Red, (2, 0))];;
// val it: pos list = [(0, 0); (0, 1); (0, 2); (1, 1); (1, 2);
   (2, 1); (2, 2)]
empty: s: state -> pos list

// randomly place a new piece of color c on an empty position on
// the board s, e.g.,
// > addRandom Red [(Blue, (1, 0)); (Red, (2, 0))];;
// val it: state option = Some [(Red, (0, 2)); (Blue, (1, 0));
   (Red, (2, 0))]
addRandom: c: value -> s: state -> state option
```

With these functions and Canvas it is possible to program the game in a few lines. Add the following to your library:

(a) Write a canvas draw function

```
draw: w: int -> h: int -> s: state -> canvas
```

which makes a new canvas and draws the board in s.

(b) Write a canvas react function

```
react: s: state -> k: key -> state option
```

which titles the board base according to the arrow-key, the user presses. Note that tilt left is given by the `shiftLeft` function. Tilt right can be accomplished by `fliplr >> shiftLeft >> fliplr`, and tilt up and down can likewise be accomplished with the additional use of `transpose`.

Finally, make an application program, which calls `runApp "2048" 600 600 draw react board`.

All above mentioned functions are to be documented using the XML-standard, and simple test examples are to be made for each function showing that it likely works.

4.29 Abstract list

4.29.1 Teacher's guide

4.29.2 Introduction

4.29.3 Exercise(s)

4.29.3.1: A linked list is an abstract datatype that is built into `F#`.

In this exercise you will create your own implementations of a linked list, one only supporting `int` and a generic linked list that can be used with any datatype, just like the built-in `F#` lists.

(a) Copy the following signature into `intLinkedList.fsi` and write a corresponding implementation in `intLinkedList.fs`

```
module IntLinkedList
type intLinkedList = Nil | Cons of int * intLinkedList
val head : intLinkedList -> int
val tail : intLinkedList -> intLinkedList
val isEmpty : intLinkedList -> bool
val length : intLinkedList -> int
val add : int -> intLinkedList -> intLinkedList
```

Write a small test program to ensure your implementation works as expected. You can take inspiration from the following listing:

```
open IntLinkedList
let emptyList = Nil
let l1 = Cons (1, Nil)
let l2 = Cons (1, Cons (2, Nil))
let l3 = add 3 l2
isEmpty emptyList |> printfn "Empty list is empty: %A"
isEmpty l1 |> not |> printfn "Non-empty list is not empty: %A"
head l1 = 1 |> printfn "head gives the first element: %A"
tail l1 = Nil |> printfn "tail gives the rest of the list: %A"
length l3 = 3 |> printfn "l3 has length 3: %A"
```

(b) In the previous sub-exercise the linked list is restricted to the type `int`. In this sub-exercise you should construct a generic linked list module.

Copy and finish the following signature in `linkedList.fsi` and write a corresponding implementation in `linkedList.fs`.

```
module LinkedList
type LinkedList<'a> = Nil | Cons of 'a * intLinkedList<'a>
val head : LinkedList<'a> -> 'a
val tail : ?? // Fill in yourself
val isEmpty : ?? // Fill in yourself
val length :  ?? // Fill in yourself
val add :  ?? // Fill in yourself
```

Write a small test-program showing you can construct linked lists of all types. You should be able to reuse `all` of your test from the previous sub-exercise and thus create linked lists of `LinkedList<int>`, as well as the following:

```
open LinkedList
let emptyList = Nil
let l1Float = add 2.0 emptyList |> add 3.14 // A float list
let l1String = add "Linked lists are cool!" emptyList
let l2String = add "What is cool?" l1String
// A list of int lists
let intLstLst = add l1 emptyList |> add l2 |> add emptyList
// a list of string lists
let strLstLst = add l1String emptyList |> add l2String
```

(c) Implement `fold` for your linked list module, similar to `List.fold`.

(d) Implement `foldBack` for your linked list module, similar to `List.foldBack`.

(e) Implement `map` for your linked list module, similar to `List.map`.

4.30 Random text

4.30.1 Teacher's guide

Emne Functional programming, histograms, random values, tree types

Sværhedsgrad Hard

4.30.2 Introduction

H.C. Andersen (1805-1875) is a Danish author who wrote plays, travelogues, novels, poems, but perhaps is best known for his fairy tales. An example is Little Claus and Big Claus (Danish: Lille Claus og store Claus), which is a tale about a poor farmer, who outsmarts a rich farmer. A translation can be found here: `http://andersen.sdu.dk/vaerk/hersholt/LittleClausAndBigClaus_e.html`. It starts like this:

> "LITTLE CLAUS AND BIG CLAUS a translation of Hans Christian Andersen's 'Lille Claus og Store Claus' by Jean Hersholt. In a village there lived two men who had the selfsame name. Both were named Claus. But one of them owned four horses, and the other owned only one horse; so to distinguish between them people called the man who had four horses Big Claus, and the man who had only one horse Little Claus. Now I'll tell you what happened to these two, for this is a true story".

The assignment is to design a spell checker that is based on the text by H.C. Andersen. As part of this handout please find `spellCheck.fsx`, which you should complete by solving the exercises of this assignment.

4.30.3 Exercise(s)

4.30.3.1: (a) Please read and understand the data structure Trie that has been handed out in `spellCheck.fsx` and understand how a word tree is created in this Trie and how the operations are performed.

(b) We need to create an `autoComplete` functionality based on a lookup of a specific word in the trie. This function `lookup` should have the signature `let lookup (prefix: string) (trie: Trie<char>) : Trie<char> Option`. It should return an option of the (sub)trie found by the lookup. `autoComplete` will use `lookup` to check if the word beginning with the `prefix` exists, after which it should return a sequence of strings of words which `prefix` in the trie. The signature of `autoComplete` is `let autoComplete (prefix: string) (trie: Trie<char>) : string seq`.

(c) Implement `spellCheck` with the signature `let spellCheck (word: string) (trie: Trie<char>) : bool` with the functionality to check if a given word is contained in the trie.

(d) Implement `genText` with the signature `let genText (len: int) (trie: Trie<char>) : string` which generates a string text of length `len` of random words that are generated by the function `randWord` with the signature `let randWord (trie: Trie<char>) : char list`. `randWord` should retrieve a random word from the given trie.

(e) Test your implementation with the given tests and, potentially, add extra tests.

Chapter 5

F# Imperative Programming

5.1 My first Fsharp program

5.1.1 Teacher's guide

Emne De studerende skal

- lave deres første F# program
- komme til at kende forskellen mellem decimal, binær, hexadecimal, og oktal repræsentation af heltal, samt at kunne konvertere imellem dem.
- komme til at kunne beskrive simple typer i F#: `int`, `float`, `char`, `string`, `bool`, samt konvertering imellem dem.
- komme til at kunne bruge F# som en lommeregner.

Sværhedsgrad Let

5.1.2 Introduction

F# is a functional programming first language, i.e., functional programming is very well supported by the language constructions. It uses types, but they do not always need to be specified. It has two modes: Interactive and Compile mode. The following assignments are designed to get you accustommed with F#.

5.1.3 Exercise(s)

5.1.3.1: Start an interactive F# session and type the following ending with a newline:

Listing 3: My first F#.

```
3.14+2.78;;

```

Describe what F# did and if there was an error, find it and repeat.

5.1.3.2: Consider the F#-expression `"hello\nworld\n"`. Explain what the "\n" means, compute the expression using F# and discuss the result.

5.1.3.3: Repeat Exercise 1, but this time, type the code in a text editor and save the result in a file with the suffix `.fsx`. Run through `fsharpi` from the console, and by first compiling it with `fsharpc` and executing the compiled file using `mono`. Consider whether the result was as expected and why.

5.1.3.4: Write an expression which concatenates the strings `"hello"`, `" "`, `"world"` and run it in F#.

5.1.3.5: Use a recursive-loop and pattern recognition to write a program, which prints the numbers $1 \ldots 10$ on the screen.

5.1.3.6: Use a `while`-loop and a mutable value to write a program, which prints the numbers $1 \ldots 10$ on the screen.

5.1.3.7: Using pen and paper:

a) Write the integer 3_{10} on binary form by using the divide-by-2 algorithm.

b) Write the integer 1001_2 on decimal form using the multiply-by-2 algorithm.

c) Write the integer 47_{10} on hexadecimal and octal form.

5.1.3.8: Modify the following program (`quickStartRecursiveInput.fsx`):

```
let rec readNonZeroValue () =
  let a = int (System.Console.ReadLine ())
  match a with
    0 ->
      printfn "Error: zero value entered. Try again"
      readNonZeroValue ()
    | _ ->
      a
printfn "Please enter a non-zero value"
let b = readNonZeroValue ()
printfn "You typed: %A" b
```

such that instead of asking the user for a non-zero value, repatedly asks the user for the name of a programming language. If the user enters `"quit"`, then the program should stop. If the user enters `"fsharp"`, then the program should write `"fsharp is cool"`. In all other cases, program should write `"I don't known <name>"`, where `"<name>"` is the string, the user entered. Example of a user dialogue is:

```
% dotnet fsi fsharpIsCool.fsx
Please enter the name of a programming language:
c
I don't know "c"
Please enter the name of a programming language:
fsharp
Fsharp is cool
Please enter the name of a programming language:
quit
```

5.1.3.9: Start `dotnet fsi` and interactive mode, write a function `subInt a b`, where the arguments are integers, and which returns $a - b$. Write a similar function `subFloat a b` where the arguments are floats. In the type-signature returned by F#, how can you see that these functions have 2 arguments and what their types are?

5.1.3.10: Write a program in an editor, which

(a) Writes `"Hello, what is your name:"` to the screen

(b) Reads the users name from the keyboard

(c) Prints `"Hello <name>"` to the screen where `<name>` is replaced by what the user enters.

Save the file, and run the program using `"dotnet fsi <filename>"` and verify that it does as you expect.

5.1.3.11: Start an editor and write a program which prints "hello world" on the screen. Save the file as a `.fsx` file. Execute the program from the command line using

```
dotnet fsi <filename>
```

where `<filename>` is the name of the file you chose, and verify that it prints as expected.

5.1.3.12: Describe the 3 ways an F# program can be run from the command line (terminal), and discuss the advantages and disadvantages of each method.

5.1.3.13: Enter the integer 47_{10} on hexadecimal, octal, and floating-point form in F# and verify that all represents the same value.

5.1.3.14: Make a program, which opens a canvas and draws a simple shape of your own choosing. Save the image to a `.png` file and verify that the result is as expected.

5.1.3.15: Write an F#-expression which extracts the 3. element and the substring from the 2. to the 4. element in the string "abcdef".

5.1.3.16: Write the F#-expression, which extracts the first and second word from the string "hello world" using slicing.

5.1.3.17: Use pen and paper to complete the following table

Decimal	Binary	Hexadecimal	Octal
10			
	10101		
		2f	
			73

such that every row represents the same value written on 4 different forms. Include a demonstration of how you converted binary to decimal, decimal to binary, binary to hexadecimal, hexadecimal to binary, binary to octal, and octal to binary.

5.1.3.18: Write the truth table for the boolean expression a **or** b **and** c, where a, b, and c are boolean values.

5.1.3.19: Type the following expression in F# interactive mode,

Listing 4: Problematic F#.

```
3 + 1.0;;
```

and explain the result. Consider whether you can improve the expression.

5.1.3.20: Write an F#-expression for a string that contains the characters "edb" solely by using unicode escape codes.

5.1.3.21: Consider the F#-expression `164uy+230uy`. Explain what `"uy"` means, compute the expression with `fsharpi`, and discuss the result.

5.1.3.22: Write an F#-expression for a string which contains the character sequence "\n", but where "\n" is not converted to a newline. How many different ways can this be done?

5.2 Bindings

5.2.1 Teacher's guide

Emne bindinger af værdier, funktioner, mutérbare variable, og løkker

Sværhedsgrad Let

5.2.2 Introduction

Being a functional-first programming language, many of its structures are designed to support functional programming style. However, imperative programming constructs are also available. In the following exercises, you will work with lightweight and verbose syntax and simple imperative programming assignments.

5.2.3 Exercise(s)

5.2.3.1: Type the following program in a text file, compile, and execute it:

Listing 5: Value bindings.

```
let a = 3
let b = 4
let x = 5
printfn "%A * %A + %A = %A" a x b (a * x + b)
```

Explain why there must be a paranthesis in the `printfn` statement. Add a binding of the expression $ax + b$ to the name `y`, og modify the `printfn` call to use `y` instead. Are parantheses still necessary?

5.2.3.2: Listing 5 uses Lightweight syntax. Rewrite the program such that Verbose syntax is used instead.

5.2.3.3: The following program:

Listing 6: Strings.

```
let firstName = "Jon"
let lastName = "Sporring" in let name = firstName + " " +
    lastName;;
printfn "Hello %A!" name;;
```

was supposed to write "Hello Jon Sporring!" to the screen. Unfortunately, it contains an error and will not compile. Find and correct the error(s). Rewrite the program into a single line without the use of semicolons.

5.2.3.4: Add a function:

```
f : a:int -> b:int -> x:int -> int
```

to Listing 5, which returns the value of $ax+b$, and modify the call in `printfn` to use the function rather than the expression `(a * x + b)`.

5.2.3.5: Use the function developed in Assignment 4, such that the values for the arguments $a = 3$, $b = 4$, and $x = 0 \ldots 5$ are written using 6 `printfn` statements. Modify this program to use a `for` loop and a single `printfn` statement. Repeat the modification, but this time by using a `while` loop instead. Which modification is simplest and which is most elegant?

5.2.3.6: Make a program, which writes the multiplication table for the number 10 to the screen formatted as follows:

	1	2	...	10
1	1	2	...	10
2	2	4	...	20
⋮				
10	10	20	...	100

I.e., left row and top columns are headers showing which numbers have been multiplied for an element in the temple. You must use `for` loops for the repeated operations, and the field width of all the positions in the table must be identical.

5.2.3.7: Consider multiplication tables of the form,

	1	2	3	4	5	6	7	8	9	10
1	1	2	3	4	5	6	7	8	9	10
2	2	4	6	8	10	12	14	16	19	20
3	3	6	9	12	15	18	21	24	27	30
...										

where the elements of the top row and left column are multiplied and the result is written at their intersection.

In this assignment, you are to work with a function

```
mulTable : n:int -> string
```

which takes 1 argument and returns a string containing the first $1 \leq n \leq 10$ lines in the multiplication table including <newline> characters. Each field must be 4 characters wide. The resulting string must be printable with a single`printf "%s"`statement. For example, the call `mulTable 3` must return.

Listing 7: An example of the output from `mulTable`.

```
printf "%s" (mulTable 3);;
        1   2   3   4   5   6   7   8   9  10
    1   1   2   3   4   5   6   7   8   9  10
    2   2   4   6   8  10  12  14  16  18  20
    3   3   6   9  12  15  18  21  24  27  30
```

All entries must be padded with spaces such that the rows and columns are right-aligned. Consider the following sub-assignments:

(a) Create a function with type

```
mulTable : n:int -> string
```

such that it has one and only one value binding to a string, which is the resulting string for $n = 10$, and use indexing to return the relevant tabel for $n \leq 10$. Test `mulTable n` for $n = 1,2,3,10$. The function should return the empty string for values $n < 1$ and $n > 10$.

(b) Create a function with type

```
loopMulTable : n:int -> string
```

such that it uses a local string variable, which is built dynamically using 2 nested `for`-loops and the `sprintf`-function. Test `loopMulTable n` for $n = 1,2,3,10$.

(c) Make a program, which uses the comparison operator for strings, "=", and write a table to the screen with 2 columns: `n`, and the result of comparing the output of `mulTable n` with `loopMulTable n` as `true` or `false`, depending on whether the output is identical or not.

(d) Use `printf "%s"`and `printf "%A"`to print the result of `mulTable`, and explain the difference.

5.2.3.8: Consider the factorial-function,

$$n! = \prod_{i=1}^{n} i = 1 \cdot 2 \cdot \ldots \cdot n \tag{5.1}$$

(a) Write a function

```
fac : n:int -> int
```

which uses a `while`-loop, a counter variable, and a local variable to calculate the factorial-function as (4.1).

(b) Write a program, which asks the user to enter the number `n` using the keyboard, and which writes the result of `fac n`.

(c) Make a new version,

```
fac64 : n:int -> int64
```

which uses `int64` instead of `int` to calculate the factorial-function. What are the largest values n, for which `fac` and `fac64` respectively can calculate the factorial-function for?

5.2.3.9: Consider the following sum of integers,

$$\sum_{i=1}^{n} i. \tag{5.2}$$

This assignment has the following sub-assignments:

(a) Write a function

```
sum : n:int -> int
```

which uses the counter value, a local variable (mutable value) `s`, and a `while`-loop to compute the sum $1+2+\cdots+n$ also written in (5.2). If the function is called with any value smaller than 1, then it is to return the value 0.

(b) By induction one can show that

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}, n \geq 0 \tag{5.3}$$

Make a function

```
simpleSum : n:int -> int
```

which uses (5.3) to calculate $1+2+\cdots+n$ and which includes a comment explaining how the expression implemented is related to the mentioned sum.

(c) Write a program, which asks the user for the number n, reads the number from the keyboard, and write the result of `sum n` and `simpleSum n` to the screen.

(d) Make a program, which writes a table to the screen with 3 columns: `n`, `sum n` and `simpleSum n`. The table should have a row for each of $n = 1,2,3,..,10$, and each field must be 4 characters wide. Verify that the two functions calculate identical results.

(e) What is the largest value n that the two sum-functions can correctly calculate the value of? Can the functions be modified, such that they can correctly calculate the sum for larger values of n?

5.2.3.10: Perform a trace-by-hand of the following program

```
let a = 3.0
let b = 4.0
let f x = a * x + b

let x = 2.0
let y = f x
printfn "%A * %A + %A = %A" a 2.0 b y
```

5.3 RGB

5.3.1 Teacher's guide

Emne Højere-ordens funktioner, currying

Sværhedsgrad Middel

5.3.2 Introduction

En farve repræsenteres ofte som en triple (rød, grøn, blå), hvor hver indgang kaldes en farvekanal eller blot en kanal, og hver kanal er typisk et heltal mellem 0 og 255:

$$c = (r, g, b) \tag{5.4}$$

Farver kan lægges sammen ved at addere deres kanaler,

$$c_1 + c_2 = \big(\text{trunc}(r_1 + r_2), \text{trunc}(g_1 + g_2), \text{trunc}(b_1 + b_2)\big), \tag{5.5}$$

$$c_i = (r_i, g_i, b_i) \tag{5.6}$$

$$\text{trunc}(v) = \begin{cases} 0, & v < 0 \\ 255, & v > 255 \\ v, & \text{ellers} \end{cases} \tag{5.7}$$

og farver kan skaleres ved at gange hver kanal med samme konstant.

$$ac = \big(\text{trunc}(ar), \text{trunc}(ag), \text{trunc}(ab)\big) \tag{5.8}$$

Farver, hvor alle kanaler har samme værdi, $v = r = g = b$, kaldes gråtoner, og man kan konvertere en farve til gråtone ved at udregne gråtoneværdien som gennemsnittet af de 3 kanaler,

$$v = \text{gray}(c) = \frac{r + g + b}{3} \tag{5.9}$$

5.3.3 Exercise(s)

5.3.3.1: Skriv en signaturfil for et modul, som indeholder funktionerne `trunc`, `add`, `scale`, og `gray` ud fra ovenstående matematiske definitioner og ved brug af tupler, hvor muligt.

5.3.3.2: Skriv en implementation af ovenstående signaturfil og kompiler begge filer til et bibliotek (`dll`-fil).

5.3.3.3: Skriv 2 programmer: Et som benytter ovenstående bibliotek via `fsharpi` og et som benytter det via `fsharpc`.

5.3.3.4: Lav en White-box afprøvning af jeres bibliotek og ved brug af `fsharpc`.

5.3.3.5: Udvid biblioteket (både signatur og implementationsfilen) med en funktion, som konverterer en farvetriplet til en gråtonetriplet. Udvid afprøvningen med en passende afprøvning af den nye funktion. Diskutér om bibliotek, program, og afprøvning er struktureret på en måde, så denne udvidelse har været let, eller om der er uhensigtmæssige afhængigheder, som gør rettelse, vedligeholdelse og udvidelse besværlig og med risiko for fejl.

5.4 Vec

5.4.1 Teacher's guide

Emne Moduler, namespaces og afprøvning

Sværhedsgrad Middel

5.4.2 Introduction

This assignment is about 2-dimensional vectors. A 2-dimensional vector (henceforth just called a vector) is a geometrical object consisting of a length and a direction. Typically, a vector is represented as a pair of numbers, $\vec{v} = (x, y)$, where its length and direction are found as,

$$\text{len}(\vec{v}) = \sqrt{x^2 + y^2} \tag{5.10}$$

$$\text{ang}(\vec{v}) = \text{atan2}(y, x) \tag{5.11}$$

Vectors are often drawn as arrows with a head and a tail. In the Cartesian coordinate system, if the tail is placed at $(0,0)$, then the head will be at (x, y). Addition of vectors is performed elementwise:

$$\vec{v}_1 = (x_1, y_1) \tag{5.12}$$

$$\vec{v}_2 = (x_2, y_2) \tag{5.13}$$

$$\vec{v}_1 + \vec{v}_2 = (x_1 + x_2, y_1 + y_2) \tag{5.14}$$

Addition can also be drawn, as shown in Figure 5.1.

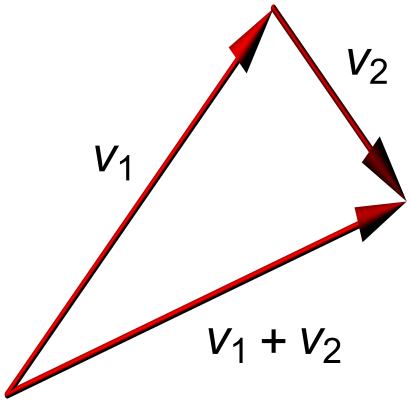


Figure 5.1: An illustation of vector addition.

5.4.3 Exercise(s)

5.4.3.1: Consider the signature file given in Listing 8 which contains some of the standard operations for vectors.

Listing 8 vec2dsmall.fsi:
A signature file for vector operations.

```
/// A 2 dimensional vector library.
/// Vectors are represented as pairs of floats
module vec2d
/// The length of a vector
val len : float * float -> float
/// The angle of a vector
val ang : float * float -> float
/// Addition of two vectors
val add : float * float -> float * float -> float * float
```

Solve the following sub-tasks:

a) Extend the signature file with documentation using the documentation standard.

b) Write a library `vec2dsmall.fs` implementing the signatures.

c) Compile the signature and the implementation into `vec2dsmall.dll` demonstrating that there are no syntax errors.

5.4.3.2: Write a White-box test of the library.

5.4.3.3: Points on a circle of radius 1 can be calculated as $(\cos\theta, \sin\theta)$, $\theta \in [0, 2\pi)$. Consider the closed polygon consisting of $n > 1$ points on a circle, where $\theta_i = \frac{2\pi i}{n}$, $i = 0..(n-1)$, and where neighbouring points are conected with straight lines.

Write a program with the function,

```
polyLen : n:int -> float
```

which uses the above library to calculate the length of the polygon. The length is calculated as the sum of line pieces connecting neighbouring points. The program should further write a table of lengths for increasing number of points n, and the results should be compared with the circumference of a circle with radius 1. What appears to be the limit, when $n \to \infty$?

5.4.3.4: The library `vec2d` is based on the representations of vectors as pairs (2-tuples). Make a sketch of a signaturfil for a variant of the library, which avoids tuples. Discuss possible challenges and major changes, which the variant will require both for the implementation of the library and the application program.

5.4.3.5: Consider the following application

Listing 9: Simple usage of the Color library.

```
let v = (1.3, -2.5)
printfn "Vector %A: (%f, %f)" v (vec2d.len v) (vec2d.ang v)
let w = (-0.1, 0.5)
printfn "Vector %A: (%f, %f)" w (vec2d.len w) (vec2d.ang w)
let s = vec2d.add v w
printfn "Vector %A: (%f, %f)" s (vec2d.len s) (vec2d.ang s)

```

First run the code with `fsharpc`. Then perform a tracing by hand of the above code and the implementation of your library. Did you discover any errors? Do you get the same output?

5.4.3.6: Write a White-box test of the library.

5.4.3.7: Write a Black-box test of the library.

5.5 Modules

5.5.1 Teacher's guide

Emne Moduler, namespaces og afprøvning

Sværhedsgrad Middel

5.5.2 Introduction

A color is often represented as a triple (red, green, blue), where each entry is called a color-channel, and each channel is typically an integer between and including 0 and 255:

$$c = (r, g, b). \tag{5.15}$$

Colors can be added, by adding their channels,

$$c_1 + c_2 = \big(\text{trunc}(r_1 + r_2), \text{trunc}(g_1 + g_2), \text{trunc}(b_1 + b_2)\big), \tag{5.16}$$

$$c_i = (r_i, g_i, b_i), \tag{5.17}$$

$$\text{trunc}(v) = \begin{cases} 0, & v < 0 \\ 255, & v > 255 \\ v, & \text{ellers} \end{cases} \tag{5.18}$$

and colors can be scaled by a factor by multiplying each channel with that same factor,

$$ac = \big(\text{trunc}(ar), \text{trunc}(ag), \text{trunc}(ab)\big). \tag{5.19}$$

Colors where the channels have identical values, $v = r = g = b$, are grays, and colors are converted to grays as the average,

$$v = \text{gray}(c) = \frac{r + g + b}{3}. \tag{5.20}$$

5.5.3 Exercise(s)

5.5.3.1: Write a signature file for a module which contains the functions `trunc`, `add`, `scale`, and `gray` from the mathematical definitions above and use tuples where possible.

5.5.3.2: Write an implementation of the signature file from Assignment 1 and compile both files into a library (`dll`-file).

5.5.3.3: Write two programs: One which uses the library developed in Assignment 1 and 2 using `fsharpi` and one which uses `fsharpc`.

5.5.3.4: Make a Black-box test of your library from Assignment 2.

5.5.3.5: Make a White-box test of your library from Assignment 2.

5.5.3.6: Consider the library from Assignment 2. Assuming that your module is called `Color`, consider the following application

Listing 10: Application of a Color library.

```
let red = (255,0,0)
let green = (0,255,0)
let avg = Color.add red green
let factor = 1.25
let bright = Color.scale factor avg
printfn "Bright gray is: %A" bright

```

If your functions `add` and `scale` have a different interface, then adjust accordingly. Perform a tracing by hand of the above code including the implementation of your library. Run the (adjusted) code with `fsharpc`. Did you discover any errors? Do you get the same output?

5.5.3.7: Extend the library (both the signature and the implementation file) from Assignment 1 and 2) with a function that converts a color tripple into a tripple of identical gray values. Extend your test with a suitable set of tests of this new function. Discuss whether the library, application, and test are structured in a way such that the extension has been easy, or whether there are dependencies that makes correcting, maintaining, extending the code difficult and with a high risk of introducing new errors.

5.6 Lists

5.6.1 Teacher's guide

Emne lister og arrays

Sværhedsgrad Middel

5.6.2 Introduction

A list is a very important programming data structure, and functional programming is particularly well suited for processing lists. Hence, F# has constructs that support list operations, and some of these will be worked with in the following assignments.

5.6.3 Exercise(s)

5.6.3.1: Use `Array.init` to make a function `squares: n:int -> int []`, such that the call `squares n` returns the array of the first n square numbers. For example, `squares 5` should return the array `[|1; 4; 9; 16; 25|]`.

5.6.3.2: Arrays are an alternative data structure for tables.

(a) Use `Array2D.init`, `Array2D.length1` and `Array2D.length2` to make the function `transposeArr : 'a [,] -> 'a [,]` which transposes the elements in input.

(b) Make a whitebox test of `transposeArr`.

(c) Comparing this implementation with Assignment 26d, what are the advantages and disadvantages of each of these implementations?

(d) For the application of tables, which of lists and arrays are better programmed using the imperative paradigm and using the functional paradigm and why?

5.6.3.3: The function `List.allPairs: 'a list -> 'b list -> ('a * 'b) list`, takes two lists and produces a list of all possible pairs. For example,

```
> List.allPairs [1..3] ['a'..'d'];;
val it: (int * char) list =
  [(1, 'a'); (1, 'b'); (1, 'c'); (1, 'd'); (2, 'a'); (2, 'b');
   (2, 'c'); (2, 'd'); (3, 'a'); (3, 'b'); (3, 'c'); (3, 'd')]
```

Make your own implementation using two `List.map` and `List.concat`.

5.6.3.4: Project Euler.net 1: Multiples of 3 and 5

If we list all the natural numbers below 10 that are multiples of 3 or 5, we get 3, 5, 6 and 9. The sum of these multiples is 23. Find the sum of all the multiples of 3 or 5 below 1000.

5.6.3.5: Project Euler.net 2: Even Fibonacci numbers

Each new term in the Fibonacci sequence is generated by adding the previous two terms. By starting with 1 and 2, the first 10 terms will be: 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ... By considering the terms in the Fibonacci sequence whose values do not exceed four million, find the sum of the even-valued terms.

5.6.3.6: Project Euler.net 3: Largest prime factor

The prime factors of 13195 are 5, 7, 13 and 29. What is the largest prime factor of the number 600851475143 ?

5.6.3.7: Make a function `even: int -> bool` which returns `true` if the input is even and false otherwise. Use `List.filter` and `even` to make another function `filterEven : int list -> int list`, which returns all the even numbers of a given list.

5.6.3.8: Write the types for the functions `List.filter` and `List.foldBack`.

5.6.3.9: Write a function `printLstAlt: 'a list -> ()`, which uses `for-in`, to print every element of a given list to the screen. Ensure that your function works for lists of various types, e.g., int list and string list.

5.6.3.10: Make a function `avg: (lst: float list) -> float` using `List.fold` and `lst.Length` which calculates the average value of the elements of `lst`.

5.6.3.11: Use `Array2D.init`, `Array2D.length1` and `Array2D.length2` to make the function `transpose : 'a [,] -> 'a [,]` which transposes the elements in input. Comparing this implementation with Assignment 26d, what are the advantages and disadvantages of each of these implementations?

5.6.3.12: In the following, you are to work with different ways to create a list:

(a) Make an empty list, and bind it with the name `lst`.

(b) Create a second list `lst2`, which prepends the string `"F#"` to `lst` using the cons operator `::`. Consider whether the types of the old and new list are the same.

(c) Create a third list `lst3` which consists of 3 identical elements `"Hello"`, and which is created with `List.init` and the anonymous function `fun i -> "Hello"`.

(d) Create a fourth list `lst4` which is a concatenation of `lst2` and `lst3` using "`@`".

(e) Create a fifth list `lst5` as `[1; 2; 3]` using `List.init`

(f) Write a recursive function `oneToN : n:int -> int list` which uses the concatenation operator, "`@`", and returns the list of integers `[1; 2; ...; n]`. Consider whether it would be easy to create this list using the "`::`" operator.

(g) Write a recursive function `oneToNRev : n:int -> int list` which uses the cons operator, "`::`", and returns the list of integers `[n; ...; 2; 1]`. Consider whether it would be easy to create this list using the "`@`" operator.

5.6.3.13: Write a function `printLst: 'a list -> ()`, which uses `List.iter`, an anymous function, and `printfn "%A"` to print every element of a given list to the screen. Ensure that your function works for lists of various types, e.g., int list and string list.

5.6.3.14: Sometimes tables have to be reshaped into single lists and back again.

(a) Define a function `flatten : 'a list list -> 'a list`, which concatenates a list of lists to a single list. For example, `flatten [[1; 2; 3]; [4; 5; 61]]` should return `[1; 2; 3; 4; 5; 6]`.

(b) Define the inverse function `reshape : m : int -> 'a list -> 'a list list`, which takes a number of rows and a list and returns the corresponding table. For example, `reshape 2 [1; 2; 3; 4; 5; 6]` should return `[[1; 2; 3]; [4; 5; 61]]`.

(c) Make a whitebox test of the above functions.

5.6.3.15: Use `List.map` write a function, which takes a list of integers and returns the list of floats where each element has been divided by `2.0`. For example, if the function is given the input `[1; 2; 3]`, then it should return `[0.5; 1.0; 1.5]`.

5.6.3.16: Use `List.map` to make a function `applylist : ('a -> 'b) list -> 'a -> 'b list`, which applies a list of functions to the same element and returns a list of results. For example `applylist [cos; sin; log; exp] 3.5` should return approximately `[-0.94; -0.35; 1.25; 33.11]`.

5.6.3.17: Write a function `multiplicity: x:int -> xs:int list -> int`, which counts the number of occurrences of the number `x` in the list `xs` using `List.filter`, an anonymous function, and the `Length` property.

5.6.3.18: Write a recursive function `oneToN : n:int -> int list` which uses the cons operator, `::`, and returns the list of integers `[1; 2; ...; n]`.

5.6.3.19: Write a recursive function `rev: 'a list -> 'a list`, which uses the concatenation operator `"@"` to reverse the elements in a list.

5.6.3.20: Define a function `reverseApply : 'a -> ('a -> 'b) -> 'b`, such that `reverseApply x f` returns the result of `f x`.

5.6.3.21: Write a function `reverseArray : arr:'a [] -> 'a []` using `Array.init` and `Array.length` which returns an array with the elements in the opposite order of `arr`. For eksample, `printfn "%A" (reverseArray [|1..5|])` should write `[|5; 4; 3; 2; 1|]` to the screen.

5.6.3.22: Write the function `reverseArrayD : arr:'a [] -> unit`, which reverses the order of the values in `arr` using a while-loop to overwrite its elements. For example, the program

```
let aa = [|1..5|]
reverseArrayD aa
printfn "%A" aa
```

should output `[|5; 4; 3; 2; 1|]`.

5.6.3.23: Write a function `rev: 'a list -> 'a list`, which uses `List.fold`, an anymous function, and the `"::"` operator to reverse the elements in a list. Ensure that your function works for lists of various types, e.g., int list and string list.

5.6.3.24: En snedig programmør definerer en sorteringsfunktion med definitionen `ssort xs = Set.toList (Set.ofList xs)`. For eksempel giver `ssort [4; 3; 7; 2]` resultatet `[2; 3; 4; 7]`. Diskutér, om programmøren faktisk er så snedig, som han tror.

5.6.3.25: Write a function `split: xs:int list -> (xs1: int list) * (xs2: int list)` which separates the list `xs` into two and returns the result as a tuple where all the elements with even index is in the first element and the rest in the second. For example, `split [x0; x1; x2; x3; x4]` should return `([x0; x2; x4], [x1; x3])`.

5.6.3.26: A table can be represented as a non-empty list of equally long lists, for example, the list `[[1; 2; 3]; [4; 5; 6]]` represents the table:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

(a) Make a function `isTable : llst:'a list list -> bool`, which determines whether `llst` is a legal non-empty list, i.e., that

- there is at least one element, and
- all lists in the outer list has equal length.

(b) Make a function `firstColumn : llst:'a list list -> 'a list` which takes a list of lists and returns the list of first elements in the inner lists. For example, `firstColumn [[1; 2; 3]; [4; 5; 6]]` should return `[1; 4]`. If any of the lists are empty, then the function must return the empty list of integers `[] : int list`.

(c) Make a function `dropFirstColumn : llst:'a list list -> 'a list list` which takes a list of lists and returns the list of lists where the first element in each inner list is removed. For example, `dropFirstColumn [[1; 2; 3]; [4; 5; 6]]` should return `[[2; 3]; [5; 6]]`. Ensure that your function fails gracefully, if there is no first elements to be removed.

(d) Make a function `transposeLstLst : llst:'a list list -> 'a list list` which transposes a table implemented as a list of lists, that is, an element that previously was at `a.[i,j]` should afterwards be at `a.[j,i]`. For example, `transposeLstLst [[1; 2; 3]; [4; 5; 6]]` should return `[[1; 4]; [2; 5]; [3; 6]]`. Ensure that your function fails gracefully. Note that `transposeLstLst (transposeLstLst t) = t` when `t`

is a table as list of lists. Hint: the functions `firstColumn` and `dropFirstColumn` may be useful.

(e) Make a whitebox test of the above functions.

5.6.3.27: Explain the difference between the types `int -> (int -> int)` and `(int -> int) -> int`, and give an example of a function of each type.

Chapter 6

F# Object-oriented Programming

6.1 Classes

6.1.1 Teacher's guide

Emne Classes, Objects, Methods Attributes

Sværhedsgrad Let

6.1.2 Introduction

6.1.3 Exercise(s)

6.1.3.1: Implement a class `student`, which has 1 property `name` and an empty constructor. When objects of the `student` type are created (instantiated), then the individual name of that student must be given as an argument to the default constructor. Make a program that creates 2 `student` objects and prints the name stored in each object using the "."-notation.

6.1.3.2: Change the class in Exercise 1 such that the value given to the default constructor is stored in a mutable field called `name`. Make 2 methods `getValue` and `setValue`. `getValue` must return the present value of an object's mutable field, and `setValue` must take a name as an argument and set the object's mutable field to this new value. Make a program that creates 2 `student` objects and prints the name stored in each object using `getValue`. Use `setValue` to change the value of one of the object's mutable fields, and print the object's new field value using `getValue`.

6.1.3.3: Implement a class `Counter`. The class must have two methods:

- `get` which returns the present value of the counter field, and
- `incr` which increases the counter field by 1.

The constructor must make a counter field whose value is initially 0. Write a white-box test class that tests `Counter`.

6.1.3.4: Implement a class `Car` with the following properties:

(a) a specific fuel economy measured in km/liter

(b) a variable amount of fuel in liters in its tank

The fuel economy for a particular `Car` object must be specified as an argument to the constructor, and the initial amount of fuel in the tank should be set to 0.

`Car` objects must have the following methods:

- `addGas`: Add a specific amount of fuel to the car.
- `gasLeft`: Return the present amount of fuel in the car.
- `drive`: Let the car drive a specific length in km, reducing the amount of fuel in the car. If there is too little fuel then cast an exception.

Make a white-box test class `CarTest` to test `Car` and run it.

6.1.3.5: Implement a class `Moth`, which represents a moth that is attracted to light. The moth and the light live in a 2-dimensional coordinate system with axes (x, y), and the light is placed at $(0, 0)$. The moth must have a field for its position in a 2-dimensional coordinate system of floats. Objects of the `Moth` class must have the following methods:

- `moveToLight` which moves the moth in a straight line from its position halfway to the position of the light.
- `getPosition` which returns the moth's current position.

The constructor must accept the initial coordinates of the moth. Make a white-box test class and test the `Moth` class.

6.1.3.6: Write a class `Car` that has the following properties:

- `yearOfModel`: The car's year model.
- `make`: The make of the car.
- `speed`: The car's current speed.

The `Car` class should have a constructor that accepts the car's year model and make as arguments. Set the car's initial speed to 0. The `Car` class should have the following methods:

- `accelerate`: The `accelerate` method should add 5 to the `speed` attribute each time it is called.
- `brake`: The `brake` method should subtract 5 from the `speed` attribute each time it is called.
- `getSpeed`: The `getSpeed` method should return the current speed.

Design a program that instantiates a `Car` object, and then calls the `accelerate` method five times. After each call to the `accelerate` method, get the current speed of the car and display it. Then call the `brake` method five times. After each call to the `brake` method, get the current speed of the car and display it.

Extend class `Car` with the attributes `addGas, gasLeft` from exercise 4, and modify the methods `accelerate` and `brake` so that the amount of gas left is reduced when the car accelerates

or brakes. Call `accelerate` and `brake` five times, as above, and after each call display both the current speed and the current amount of gas left.

Test all methods. Create an object instance that you know will not run out of gas, and another object instance that you know will run out of gas and test that your methods `accelerate` and `brake` work properly.

6.1.3.7: In a not-so-distant future drones will be used for delivery of groceries. Imagine that the drone traffic has become intense in your area and that you have been asked to decide if drones collide.

We will make the simplifying assumptions that all drones fly at the same altitude, that drones fly with different speeds measured in centimetre/second and in different directions, and that drones fly with constant speed (no acceleration). If two drones are less than 5 meters from each other, then they collide. When a drone reaches its destination, then it lands and can no longer collide with any other drone. Create an implementation file `simulate.fs`, and add to it a `Drone` class with the following properties and methods:

- `Position` (property): returns the drone's current position in (x,y) coordinates.
- `Speed` (property): returns the drone's present speed in centimetre/second.
- `Destination` (property): returns the drone's present destination in (x,y) coordinates. If the drone is not flying, its present position and its destination are the same.
- `Fly` (method): Set the drone's new position after one second flight.
- `AtDestination` (method): Returns `true` or `false` depending on whether the drone has reached its destination or not.

The constructor must take the start position, the destination, and the speed as arguments. All positions and speeds are integers.

Extend your implementation file with a class `Airspace` that contains the drones and as a minimum has the following properties and methods:

- `Drones` (property): The collection of drones instances.
- `DroneDist` (method): The distance between two given drones.
- `FlyDrones` (method): Advance the position of all flying drones in the collection by one second.
- `AddDrone` (method): Add a new drone to the collection of drones.
- `WillCollide` (method): Given a time interval (number of minutes), determine which drones will collide. After two (or more) drones collide they are assumed to fall to the ground and are no longer considered. The method should return a list of pairs of drones that collided in the time interval.

  In the unfortunate event that three drones A, B and C are destined to collide at the same time, the list should contain the pairs (A,B), (A,C) and (B,C). In this case, you may choose between two interpretations: either the three drones collide simultaneously or one of them gets a lucky break and dodges the crash (in which case it shouldn't be in the list of collisions). Clearly document the choice you take.

Write a black-box test class `testSimulate.fsx` that tests both the `Drone` class and the `Airspace` class.

Notice that the required methods and properties are *minimum requirements*; feel free to add methods and properties if you need them.

6.1.3.8: Implement a class `account`, which is a model of a bank account. Each account must have the following properties

- `name`: the owner's name
- `account`: the account number
- `transactions`: the list of transactions

The list of transactions is a list of pairs `(description, balance)`, such that the head is the last transaction made and the present balance. If the list is empty, then the balance is zero. The transaction amount is the difference between the two last transaction balances. To ensure that there are no reoccurring numbers, the bank account class must have a single static field, `lastAccountNumber`, which is shared among all objects, and which contains the number of the last account number. When a new account is created, i.e., when an object of the `account` class is instantiated, the class' `lastAccountNumber` is incremented by one and the new account is given that number. The class must have a class method:

- `lastAccount` which returns the value of the last account created.

Further, each account object must also have the following methods:

- `add` which takes a text description and a transaction amount, and prepends a new transaction pair with the updated balance.
- `balance` which returns the present balance of the account

Make a program, which instantiates 2 objects of the `account` class and which has a set of transactions that demonstrates that the class works as intended.

6.2 Simple Jackl

6.2.1 Teacher's guide

Emne Classes, Objects, Methods Attributes

Sværhedsgrad Middel

6.2.2 Introduction

Simple Jack er en forsimplet udgave af kortspillet Blackjack. I Simple Jack spiller man ikke om penge/jetoner men blot om sejr/tab mellem en spiller og dealer. Reglerne for Simple Jack er som følger:

Spillet består af en dealer, 1-5 spillere samt et normalt kortspil (uden jokere). Ved spillets start får dealer og hver spiller tildelt 2 tilfældige kort fra bunken som placeres med billedsiden opad foran spilleren, så alle kan se dem. I Simple Jack spilles der med åbne kort dvs. alle trukne kort til hver en tid er synlige for alle spillere. Kortene har værdi som følger:

6.2.2.1: Billedkort (knægt, dame og konge) har værdien 10

6.2.2.2: Es kan antage enten værdien 1 eller 11

6.2.2.3: Resten af kortene har den påtrykte værdi

For hver spiller gælder spillet om at ende med en korthånd hvis sum af værdier er højere en dealers sum af værdier, uden at summen overstiger 21, i hvilket tilfælde spilleren er "bust". Spillerne får nu en tur hver, hvor de skal udføre en af følgende handlinger:

6.2.2.1: "Stand": Spilleren/dealeren vælger ikke at modtage kort og turen går videre.

6.2.2.2: "Hit": Spilleren/dealeren vælger at modtage kort fra bunken et ad gangen indtil han/hun vælger at stoppe og turen går videre.

Det er dealers tur til sidst efter alle andre spillere har haft deres tur. Når dealer har haft sin tur afsluttes spillet. Ved spillets afslutning afgøres udfaldet på følgende måde: En spiller vinder hvis ingen af følgende tilfælde gør sig gældende:

6.2.2.1: Spilleren er "bust"

6.2.2.2: Summen af spillerens kort-værdier er lavere end, eller lig med dealers sum af kort-værdier

6.2.2.3: Både spilleren og dealer har SimpleJack (SimpleJack er et Es og et billedkort)

Bemærk at flere spillere altså godt kan vinde på en gang. Et spil Simple Jack er mellem en spiller og dealer, så med 5 spillere ved bordet, er det altså 5 separate spil som spilles.

6.2.3 Exercise(s)

6.2.3.1: Design og implementér et program som kan simulere Simple Jack ved brug af klasser. Start med grundigt at overveje hvilke aspekter af spillet som giver mening at opdele i klasser. Spillet skal implementere således, at en spiller enten kan være en bruger af Simple Jack programmet, som foretager sine valg og ser kortene på bordet via terminalen, eller en spiller kan være en AI som skal følge en af følgende strategier:

(a) Vælg altid "Hit", medmindre summen af egne kort kan være 17 eller over, ellers vælg "Stand"

(b) Vælg tilfældigt mellem "Hit" og "Stand". Hvis "Hit" vælges trækkes et kort og der vælges igen tilfældigt mellem "Hit" og "Stand" osv.

Dealer skal følge strategi nummer 1. Der skal også laves:

- En rapport (maks 2 sider)
- Unit-tests
- Implementation skal kommenteres jævnfør kommentarstandarden for F#

Hint: Man kan generere tilfældige tal indenfor et interval (f.eks. fra og med 1 til og med 100) ved brug af følgende kode:

```
let gen = System.Random()
let ran_int = gen.Next(1, 101)
```

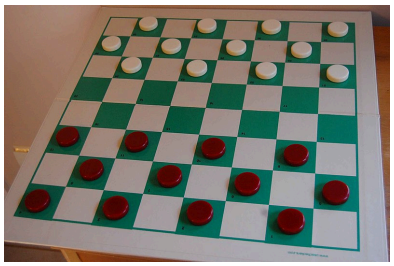


Figure 6.1: The starting position in Checkers [https://commons.wikimedia.org/wiki/File:CheckersStandard.jpg]

6.3 Object-oriented design

6.3.1 Teacher's guide

Emne Object oriented design

Sværhedsgrad Medium

6.3.2 Introduction

6.3.3 Exercise(s)

6.3.3.1: A calendar is a system for organizing meetings and events in time. A description of a calendar is as follows:

> The gregorian calendar consists of dates (day/month/year), with 12 months per year, and with months consisting of 28, 29, 30 or 31 days. The years are counted numerically with Jesus Christus' first year being called 1 AD, followed by 2 AD, etc., and the year prior is called 1 BC, preceded by 2 BC, etc. Thus, this calendar has no year 0, and the traditional time line is ..., 2 BC, 1 BC, 1 AD, 2 AD, ...
>
> A user can enter items such as a meeting or an event into a calendar. An item consists of a start date and time, end date and time, and a text-piece. Items can also be whole-day items.

Use the nouns-and-verbs method to identify possible classes and their interactions and write 1-2 lines of description for each.

6.3.3.2: Checkers also known as draughts is a ancient board game. A simplified version can be described as follows:

> Checkers is a turn-based strategy game for two players. The game is (typically) played on an 8×8 checkerboard of alternating dark- and light-colored squares. Each player starts with 12 pieces, where player one's pieces are light, and player two's pieces are dark in color, and the initial position of the pieces is shown in Figure 6.1. Players take turns moving one of their pieces. A player must move a piece if possible, and when one player has no more pieces, then that player has lost the game.

A piece may only move diagonally into an unoccupied adjacent square. If the adjacent square contains an opponent's piece and the square immediately beyond is vacant, then the piece jumps over the opponent's piece and the opponent's piece is removed from the board.

Use the nouns-and-verbs method to identify possible classes and their interactions and write 1-2 lines of description for each.

6.3.3.3: War is a card game for two players. A simplified version can be described as follows:

War is a card game for two players using the so-called French-suited deck of cards. The deck is initially divided equally between the two players, which is organized as a stack of cards. A turn is played by each player showing the top of their stack. The player with the highest card wins the hand. Aces are the highest. The won cards are placed at the bottom of the winner's stack. When one player has all the cards, then that player wins the game.

Use the nouns-and-verbs method to identify possible classes and their interactions and write 1-2 lines of description for each.

6.4 Inheritance

6.4.1 Teacher's guide

Emne Classes and inheritance

Sværhedsgrad Let

6.4.2 Introduction

6.4.3 Exercise(s)

6.4.3.1: Write a `Person` class with data properties for a person's name, address, and telephone number. Next, write a class named `Customer` that is a subclass of the `Person` class. The `Customer` class should have a positive integer data property for a unique customer number and a boolean data property indicating whether the customer wishes to be on a mailing list. Write a small program, which makes an instance of the `Customer` class.

6.4.3.2: Extend `Customer` in Exercise 1 to implement the `comparison` interface. The comparison method should compare customers based on their customer number. Create a list of several Customers and use List.sort to sort them.

6.4.3.3: (a) Write an `Employee` class that keeps data properties for the following pieces of information:

- Employee name
- Employee number

Next, write a class named `ProductionWorker` that is a subclass of the `Employee` class. The `ProductionWorker` class should keep data properties for the following information:

- Shift number (an integer, such as 1 or 2)
- Hourly pay rate

The workday is divided into two shifts: day and night. The shift property will hold an integer value representing the shift that the employee works. The day shift is shift 1 and the night shift is shift 2. Write the appropriate methods for each class.

When you have written these classes, write a program that prompts the user to enter the relevant data for a production worker, and create an object of the `ProductionWorker` class. Then use the object's methods to retrieve the relevant data and display it on the screen.

(b) Extend the previous exercise as follows: Let a shift supervisor be a salaried employee who supervises a shift. In addition to salary, the shift supervisor earns a yearly bonus when his or her shift meets production goals. Write a `ShiftSupervisor` class that is a subclass of the `Employee` class you created in the previous exercise. The `ShiftSupervisor` class should keep a data property for the annual salary and a data property for the annual production bonus that a shift supervisor has earned. Demonstrate the class by writing a program that uses a `ShiftSupervisor` object.

(c) **(Extra difficult).** Considering that production during night shifts is reduced by 5% compared to production during day shifts, and that the hourly pay rate during night shifts is double the hourly pay rate during day shifts, compute the best possible worker & shift allocation over the period of 12 months. You need to think how to measure productivity and salary cost, and then find their best tradeoff in the period of 12 months.

6.4.3.4: Cheetahs, antelopes and wildebeests are among the world's fastest mammals. This exercise asks you to simulate a race between them. You are not asked to simulate their movement on some plane, but only some of the conditions that affect their speed when running a certain distance.

Your base class is called `Animal` and has these properties:

- The amount of food needed daily (measured in kilograms)
- The weight of the animal (measured in kilograms)
- The maximum speed of the animal (measured in kilometres per hour)
- The current speed of the animal (measured in kilometres per hour)

The `Animal` class should have a primary constructor that takes two arguments: the animal's weight and the animal's maximum speed. The `Animal` class should also have an additional constructor that takes as input only the animal's maximum speed and generates the animal's weight randomly within the range of 70 - 300 kg. The `Animal` class should have two methods:

- The first method should set the current speed of the animal proportionately to its food intake and maximum speed as follows: if the animal eats 100% of the amount of food it needs daily, the animal's current speed should be its maximum speed; if the animal eats 50% of the amount of food it needs daily, the animal's current speed should be 50% of its maximum speed, and so on.
- The second method should set the amount of food needed daily proportionately to the animal's weight as follows: the animal should eat half its own weight in food every day (if the animal weighs 50 kg, it should eat 25kg of food daily).

Create a subclass `Carnivore` that inherits everything from class `Animal`, and modifies the second method as follows: the animal should eat 8% of its own weight in food every day.

Create a subclass `Herbivore` that inherits everything from class `Animal`, and modifies the second method as follows: the animal should eat 40% of its own weight in food every day.

Create an instance of `Carnivore` called `cheetah` and two instances of `Herbivore` called `antelope, wildebeest`. Set their weight and maximum speed to:

- cheetah: 50kg, 114km/hour
- antelope: 50kg, 95km/hour
- wildebeest: 200kg, 80km/hour

Generate a random percentage between 1 - 100% (inclusive) separately for each instance. This random percentage represents the amount of food the animal eats with respect to the amount of food it needs daily. E.g., if you generate the random percentage 50% for the antelope, this means that the antelope will eat 50% of the amount it should have eaten (as decided by the second method).

For each instance, display the random percentage you generated, how much food each animal consumed, how much food it should have consumed, and how long it took for the animal to cover 10km. Repeat this 3 times (generating different random percentages each time), and declare winner the animal that was fastest on average all three times. If there is a draw, repeat and recompute until there is a clear winner.

Write a white-box test of your classes.

Optional extra: repeat the race without passing as input argument the weight of each animal (i.e. letting the additional constructor generate a different random weight for each instance).

6.5 Predator-Prey

6.5.1 Teacher's guide

Emne Inheritance

Sværhedsgrad Middel

6.5.2 Introduction

Consider a simulation of a natural habitat as two groups of animals interact. One group is the prey, a population of animals that are the food source for the other population of animals, the predators. Both groups have a fixed birthrate. The prey usually procreate faster than the predators, allowing for a growing prey population. But as the population of prey increases, the habitat can support a higher number of predators. This leads to an increasing predator population, and, after some time, a decreasing prey population. Around that time, the predator population grows so large as to reach a critical point, where the number of prey can no longer support the present predator population, and the predator population begins to wane. As the predator population declines, the prey population recovers, and the two populations continue this interaction of growth and decay.

An actual example of studying predator-prey relationships is the one between wolves and moose on Isle Royale in Lake Superior (http://www.isleroyalewolf.org/). Its population of wolves and moose are isolated on the island.

6.5.3 Exercise(s)

6.5.3.1: In the following, we will build a simulator of a predator-prey relationship in a closed environment using the following rules:

(a) The habitat updates itself in units of time called clock ticks. During one clock tick, every animal in the island gets an opportunity to do something.

(b) All animals are given an opportunity to move into an adjacent space, if an empty adjacent space is found. One move per clock tick is allowed.

(c) Both the predators and prey can reproduce. Each animal is assigned a fixed breed time. If the animal is still alive after breed time ticks of the clock, it will reproduce. The animal does so by finding an unoccupied adjacent space and fills that space with the new animal – its offspring. The animal's breed time is then reset to zero. An animal can breed at most once in a clock tick.

(d) The predators must eat. They have a fixed starve time. If they cannot find a prey to eat before starve time ticks of the clock, they die.

(e) When a predator eats, it moves into an adjacent space that is occupied by prey (its meal). The prey is removed and the predator's starve time is reset to zero. Eating counts as the predator's move during that clock tick.

(f) At the end of every clock tick, each animal's local event clock is updated. All animals' breed times are decremented and all predators' starve times are decremented.

Lav et program, som kan simulere rov- og byttedyrene som beskrevet ovenfor og skrive en lille rapport. Kravene til programmeringsdelen er:

(a) Man skal kunne angive antal af tiks (clock ticks), som simuleringen skal køre, formeringstid (breeding time) for begge racer og udsultningstid for rovdyrene ved programstart.

(b) Antallet af dyr per tik skal gemmes i en fil.

(c) Programmet skal benytte klasser og objekter

(d) Der skal være mindst en (fornuftig) nedarvning

(e) Programmets klasser skal bla. beskrives ved brug af et UML-klassediagram

(f) Programmet skal kommenteres ved brug af fsharp kommentarstandarden

(g) Programmet skal unit-testes

Kravene til rapporten er:

(h) Rapporten skal skrives i LaTeX.

(i) I skal bruge `rapport.tex` skabelonen

(j) Rapporten skal som minimum i hoveddelen indeholde afsnittene Introduktion, Problemanalyse og design, Programbeskrivelse, Afprøvning, og Diskussion og Konklusion. Som bilag skal I vedlægge afsnittene Brugervejledning og Programtekst.

(k) Rapporten må maximalt være på 10 sider alt inklusivt.

6.6 Wolves and mooses

6.6.1 Teacher's guide

Emne Inheritance

Sværhedsgrad Middel

6.6.2 Introduction

I Lake Superior på grænsen mellem USA og Canada ligger en øde ø kaldet Isle Royale. Her har man over en lang årrække fulgt populationen af ulve og elge (http://www.isleroyalewolf.org/). Bestanden af de 2 dyrarter er tæt knyttet til hinanden som rov- og byttedyr.

Denne opgave omhandler simulering af populationen af ulve og elge i et lukket miljø. Elge spiser planter og i denne opgave vil vi antage at ulve kun spiser elge. Både ulve og elge formerer sig, hvilket medfører at populationstørrelserne svinger. Typiske mønstre er, at hvis elgbestanden bliver stor, så vokser ulvebestanden efterfølgende, da der nu kan brødfødes flere ulve. Når ulvebestanden er stor, så falder elgbestanden efterfølgende, da ulvene nedlægger mange elge. Når der er få elge, så falder ulvebestanden pga. manglende føde, hvorefter elgbestanden igen vokser.

6.6.3 Exercise(s)

6.6.3.1: I det følgende skal der simuleres et lukket miljø med ulve og elge. Simuleringen skal benytte følgende regler:

(a) Et miljø består af $n \times n$ felter.

(b) Alle levende dyr har en coordinat i miljøet, og der kan højst være et dyr per felt. Når et dyr dør, fjernes det fra miljøet. Hvis et dyr fødes, tilføjes det i et tomt felt. Ved simuleringens begyndelse skal der være u ulve og e elge som placeret tilfældigt i tomme felter.

(c) Miljøet opdateres i tidsenheder, som kaldes tiks, og simuleringen udføres T tiks. Indenfor et tik kan dyrene gøre et af følgende: Flytte sig, formere sig, og for ulvenes vedkommende spise en elg. Kun et dyr handler ad gangen og rækkefølgen er tilfældig.

(d) Dyr kan flytte sig et felt per tik til et af de 8 nabofelter, som er tomme.

(e) Alle dyr har en artsspecifik formeringstid f angivet i antal tiks, og som tæller ned. Når formeringstiden når nul (for et levende dyr), og der er et tomt nabofelt, så fødes der et nyt dyr af samme type ved at det nye dyr tilføres i et tomt nabofelt. Moderdyrets formeringstid sættes til startværdien, hhv. f_{elg} og f_{ulv}.

(f) Ulve har en sulttid s angivet i antal tiks, og som tæller ned. Hvis sulttiden når nul, så dør ulven, og den fjernes fra miljøet.

(g) Ulve kan spiser elge. Hvis der er en elg i et nabofelt vil ulven spise elgen, elgen fjernes fra miljøet, ulven flytter til elgens felt, og ulvens sulttid sættes til startværdien, s.

(h) I hvert tik reduceres alle formerings- og sulttællere for levende dyr med 1.

Lav et program, som kan simulere dyrene som beskrevet ovenfor og skrive en rapport. Til opgaven udleveres følgende kildefiler:

`animalsSmall.fsi`, `animalsSmall.fs`, og `testAnimalsSmall.fs`.

Opgaven er at tage udgangspunkt i disse filer og programmere følgende regler:

(a) Der skal laves et bibliotek som implementerer klasser for miljø, ulve og elge. Det er ikke et krav at der bruges nedarvning.

(b) Man skal kunne starte simuleringen med forskellige værdier af T, n, u, e, f_{elg}, f_{ulv} og s

(c) Der skal laves en white-box test af biblioteket.

(d) Der skal laves en applikation, som kører en simulering, og tidsserien over antallet af dyr per tik skal gemmes i en fil. Filnavn og parametrene T, n, e, f_{elg}, u, f_{ulv} og s skal angives som argumenter til det oversatte program fra komandolinjen. Eksempelvis kunne:

```
mono experimentWAnimals.exe 40 test.txt 10 30 10 2 10 4
```

starte et eksperiment med $T = 40$, $n = 10$, $e = 30$, $f_{\text{elg}} = 10$, $u = 2$, $f_{\text{ulv}} = 10$ og $s = 4$ og hvor tidsserien skrives til filen `test.txt`.

(e) Der skal laves et antal eksperimenter, hvor simuleringen køres med forskellige værdier af simuleringens parametre. For hvert eksperiment skal der laves en graf (ikke nødvendigvis i F#), der viser antallet af ulve og elge over tid.

(f) Koden skal kommenteres ved brug af F# kommentarstandarden.

Kravene til rapporten er:

(g) Rapporten skal skrives i LaTeX og tage udgangspunkt i `rapport.tex` skabelonen

(h) Rapporten skal som minimum indeholde afsnittene Introduktion, Problemanalyse og design, Programbeskrivelse, Afprøvning, Eksperiment og Konklusion. Som bilag skal I vedlægge afsnittene Brugervejledning og Programtekst.

(i) Eksperimentafsnittet skal kort diskutere hvert eksperiments udfald.

(j) Rapporten minus bilag må maximalt være på 10 A4 sider alt inklusivt.

6.7 Owls and mice

6.7.1 Teacher's guide

Emne classes and objects

Sværhedsgrad Hard

6.7.2 Introduction

This assignment is about simulating a preditor-prey relationship in a simplified setting.

6.7.3 Exercise(s)

6.7.3.1: You are to simulate owls and mice in a closed environment. The owls are immortal and hunt mice to eat, and the mice run around randomly and multiply (propagate). The overall rules for the simulation are:

(a) The environment must consist of $n \times n$ fields organized as a checkerboard.

(b) Alive animals have a coordinate in the environment, and there can only be one animal per coordinate.

(c) The simulation updates in ticks, and after each tick, all animals perform an action.

(d) The simulation runs for T ticks.

(e) There must initially be O owls and M mice.

The possible actions are:

(f) A mouse can move to a neighbouring empty field.

(g) A mouse must have a counter, such that after p ticks, the mouse will not move but multiply. The effect is that an offspring is created in an empty neighbouring field. If there is no empty neighbouring field, then the mouse waits a turn.

(h) An owl can move to all neighbouring fields not occopied by another owl. If an owl moves to a field with a mouse, then the mouse is eaten and the mouse is removed from the board.

You are to:

(a) Use the object-oriented programming paradigm and include inheritance in your solution.

(b) Create a program `simulate.fsx`, which runs the simulation and prints the tick number and the total number of mice after each tick to the textfile `simulation.txt`. The program must accept the parameters n, T, p, M, O, at the command-line when the simulation starts.

(c) Collect the main classes in an implementation file called `preditorPrey.fs`, which `simulate.fsx` links to.

(d) Make a white-box test of the implementation file, `testPreditorPrey.fsx`.

(e) Find parameters, where the mice population diminishes to zero quickly, explodes, and is seemingly in balance, and demonstrate this by copying and renaming the textfile `simulation.txt` to the three corresponding files

- `simulationExtinction.txt`,
- `simulationOverpopulation.txt`, and
- `simulationBalance.txt` respectively.

You are also to write a report:

(f) The report must as a minimum include the sections: Introduction, Problem analysis and design, Program description, Testing, Experiments, and Conclusion. Include a User guide and your source code as appendices.

(g) The report must be no longer than 10 pages excluding the appendices.

6.8 Chess

6.8.1 Teacher's guide

Emne Classes, inheritance and UML class diagrams

Sværhedsgrad Svær

6.8.2 Introduction

Sporring, "Learning to program with F#", 2017, Chapter 21.4 describes a simplified version of Chess with only Kings and Rooks, and which we here will call Simplechess, and which is implemented in 3 files: `chess.fs`, `pieces.fs`, and `chessApp.fsx`. In this assignment you are to work with this implementation.

6.8.3 Exercise(s)

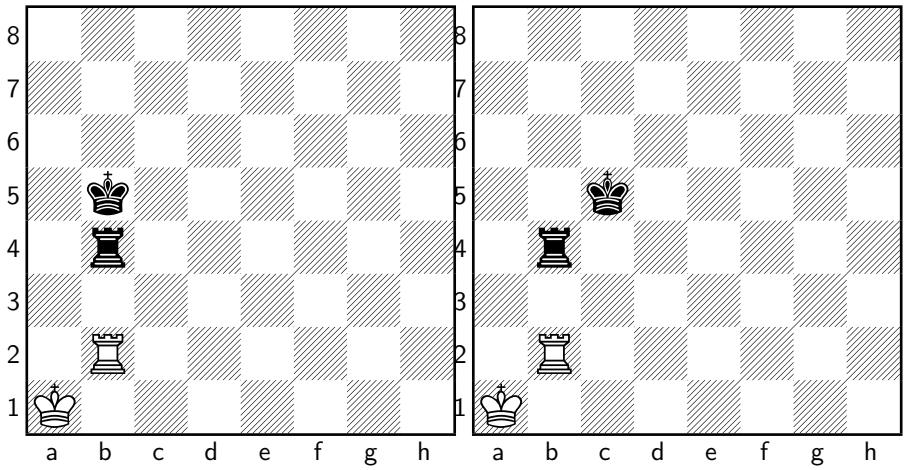
6.8.3.1: Produce a UML class diagram describing the design presented of Simple Chess in the book.

6.8.3.2: The implementation of `availableMoves` for the King is flawed, since the method will list a square as available, even though it can be hit by an opponents piece at next turn. Correct `availableMoves`, such that threatened squares no longer are part of the list of vacant squares.

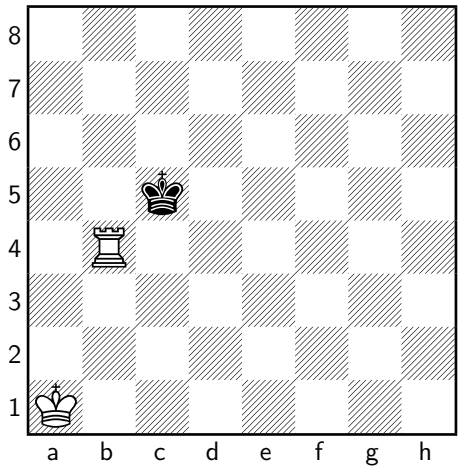
6.8.3.3: Extend the implementation with a class `Player` and two derived classes `Human` and `Computer`. The derived classes must have a method `nextMove`, which returns a legal movement as a codestring or the string "quit". A codestring is a string of the name of two squares separated by a space. E.g., if the white king is placed at a4, and a5 is an available move for the king, then a legal codestring for moving the king to a5 is "a4 a5". The player can be either a human or the computer. If the player is human, then the codestring is obtained by a text dialogue with the user. If the player is the computer, then the codestring must be constructed from a random selection of available move of one of its pieces.

6.8.3.4: Extend the implementation with a class `Game`, which includes a method `run`, and which allows two players to play a game. The class must be instantiated with two player objects either human or computer, and `run` must loop through each turn and ask each player object for their next move, until one of the players quits by typing "quit".

6.8.3.5: Extend `Player` with an artificial intelligence (AI), which simulate all possible series of moves at least $n \geq 0$ turns ahead or until a King is strikken. Each series should be given a fitness, and the AI should pick the move, which is the beginning of a series with the largest fitness. If there are several moves which have series with same fitness, then the AI should pick randomly among them. The fitness number must be calculated as the sum of the fitness of each move. A move, which does not strike any pieces gets value 0, if an opponent's rook is strikken, then the move has value 3. If the opponent strikes the player's rook, then the value of the move is -3. The king has in the same manner value ± 100. As an example, consider the series of 2 moves starting from Figure 6.2(a), and it is black's turn to move. The illustrated series is `["b5 b6"; "b2 b4"]`, the fitness of the corresponding moves are `[0;-3]`, and the fitness of the series is



(a) Initial position (b) b5 c5



(c) b2 b4

Figure 6.2: Starting at the left and moving white rook to b4.

−3. Another series among all possible is `["b5 b6"; "b2 c2"]`, which has fitness `0`. Thus, of the moves considered, `"b5 b6"` has the maximum fitness of `0` and is the top candidate for a move by the AI. Note that a rook has at maximum 14 possible squares to move to, and a king 8, so for a game where each player has a rook and a king each, then the number of series looking n turns ahead is $\mathcal{O}(22^n)$.

6.8.3.6: Draw an extended UML class diagram showing the final design including all the extending classes.

6.9 UML

6.9.1 Teacher's guide

Emne UML class diagrams

Sværhedsgrad Let

6.9.2 Introduction

6.9.3 Exercise(s)

6.9.3.1: Draw the UML diagram for the following programming structure: A `Person` class has data property for a person's name, address, and telephone number. A `Customer` has data property for a customer number and a Boolean data property indicating whether the customer wishes to be on a mailing list.

6.9.3.2: Draw an UML class diagram for the following structure:

A `Employee` class that keeps data properties for the following pieces of information:

- Employee name
- Employee number

A subclass `ProductionWorker` that is a subclass of the `Employee` class. The `ProductionWorker` class should keep data properties for the following information:

- Shift number (an integer, such as 1 or 2)
- Hourly pay rate

A class `Factory` which has one or more instances of `ProductionWorker` objects.

6.9.3.3: Write a UML class diagram for the following:

A class called `Animal` and has the following properties (choose names yourself):

- The amount of food needed daily (measured in kilograms)
- The weight of the animal (measured in kilograms)
- The maximum speed of the animal (measured in kilometres per hour)
- The current speed of the animal (measured in kilometres per hour)

The `Animal` class should have two methods (choose appropriate names):

- The first method should set the current speed of the animal proportionately to its food intake and maximum speed as follows: if the animal eats 100% of the amount of food it needs daily, the animal's current speed should be its maximum speed; if the animal eats 50% of the amount of food it needs daily, the animal's current speed should be 50% of its maximum speed, and so on.
- The second method should set the amount of food needed daily proportionately to the animal's weight as follows: the animal should eat half its own weight in food every day (if the animal weighs 50 kg, it should eat 25kg of food daily).

A subclass `Carnivore` that inherits everything from class `Animal`.

A subclass `Herbivore` that inherits everything from class `Animal`, and modifies the second method as follows: the animal should eat 40% of its own weight in food every day.

A class called `Game` consisting of one or more instances of `Carnivore` and `Herbivore`.

6.10 roguelike

6.10.1 Teacher's guide

Emne Classes, Objects, Methods Attributes

Sværhedsgrad Middel

6.10.2 Introduction

Denne opgave går ud på at lave et såkaldt retro-style *roguelike* spil. Et roguelike går ud på at spilleren skal udforske en verden og løse nogle opgaver, ofte er denne verden et underjordisk fantasy *dungeon* befolket af monstre som skal nedkæmpes, og gåder der skal løses.

I denne opgave skal der arbejdes med at lave et objekt-orienteret design, som gør det nemt at udvide spillet med nye skabninger og spil-mekanismer.

Opgaven er delt i fire dele. I den første delopgave skal der arbejdes med at implementere en *canvas* i terminalen til at vise vores verden. Anden delopgave går ud på at lave et klasse-hierarki til at repræsentere skabninger og genstande i verden. Endelig skal der i den tredie delopgave arbejdes mod at sætte de forskellige dele sammen til et samlet spil. Fjerde del indeholde en række forslag til udvidelser, hvoraf I skal implementere mindst to.

I det følgende er der kun givet minimums-krav til hvilke metoder og properties I skal implementere på jeres klasser. I må gerne lave ekstra metoder eller hjælpe-funktioner, hvis I synes det kan hjælpe til at skrive et mere elegant og forståeligt program.

Rapport

Ud over jeres programkode skal I også aflevere en rapport (skrevet i LaTeX). I rapporten skal I beskrive implementeringen af jeres klasser, det vil sige hvilken skjult tilstand (interne variable og lignende), som jeres metoder arbejder på.

Ligeledes skal rapporten indeholde et UML-klassediagram over klasserne i jeres løsning.

6.10.3 Exercise(s)

6.10.3.1: Brugergrænseflade i Terminal

For at vise spillets verden implementerer vi en klasse `Canvas`, som er et gitter af felter. Hvor feltet i øverste ventre hjørne har position $(0,0)$, og x-koordinatet tælles op mod højre, og y-koordinatet tælles op når man bevæger sig fra top mod bund.

Hvert felt har en `char`, og så *kan* feltet have en forgrundsfarve, og det *kan* have en baggrundsfarve.

Implementér klassen `Canvas` som har følgende signatur:

```
type Color = System.ConsoleColor
type Canvas =
```

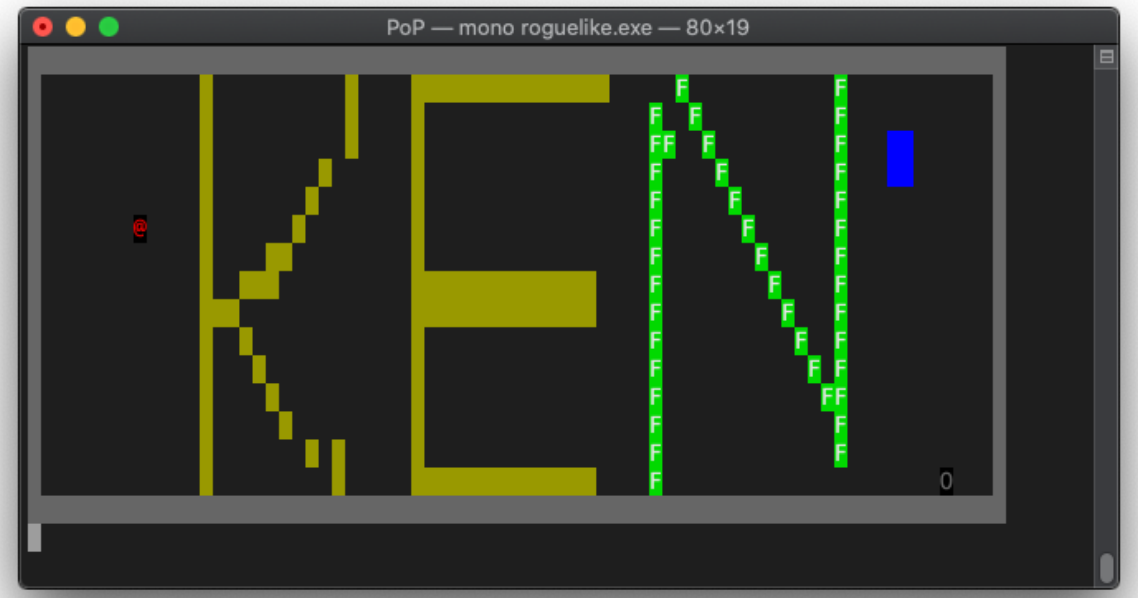


Figure 6.3: Eksempel på hvordan spillet kunne se ud i terminalen. Den røde `@` er spilleren, resten er genstande og skabninger som spilleren kan interagere med (måske med fatale følger).

```
class
  new : rows:int * cols:int -> Canvas
  member Set: x:int * y:int * c:char * fg:Color * bg:Color ->
 unit
  member Show: unit -> unit
end
```

Det vil sige:

- En konstruktør der tager antal rækker og koloner som argumenter.
- en metode `Set` til at sætte indhold og farver på et felt.
- en metode `Show` til at vise en canvas i terminalen. Figur 6.3 viser et eksempel på hvordan det kunne se ud.

I rapporten skal I beskrive jeres designovervejelser, samt redegøre for hvilke data en canvas har.

Hints: Til `Show` skal I bruge følgende funktionalitet fra standard-biblioteket:

- `System.Console.ForegroundColor <- System.ConsoleColor.White` til at sætte forgrundsfarven til hvid (kan også bruges til andre farver).
- `System.Console.BackgroundColor <- System.ConsoleColor.Blue` til at sætte baggrundsfarven til blå (kan også bruges til andre farver).
- `System.Console.ResetColor()` til at sætte farverne i terminalen tilbage til normal.

6.10.3.2: Genstande og Skabninger

Vi bruger klassen `Entity` til at repræsentere genstande og skabninger i vores verden. Disse skal kunne renderes på en canvas. Tag udgangspunkt i følgende erklæring:

```
type Entity() =
  abstract member RenderOn : Canvas -> unit
  default this.RenderOn canvas = ()
```

Hvis I får behov for det må I gerne tilføje tilstand (data og properties), metoder og en anden default implementering af `RenderOn` til `Entity`.

Til at repræsentere spilleren bruger vi klassen `Player`:

```
type Player =
  class
    inherit Entity
    new : ...
    member Damage : dmg:int -> unit
    member Heal : h:int -> unit
    member MoveTo : x:int * y:int -> unit
    member HitPoints : int
    member IsDead : bool
  end
```

En spiller starter med ti hit points. En spiller er død hvis de har mindre end nul hit points. En spiller har et maksimum hit points de kan helbredes op til (I betemmer hvor mange, husk at dokumentere det i rapporten).

Metoderne `Damage` og `Heal` bruge til at gøre skade på og, henholdsvis, helbrede spilleren med et antal hit points.

Til at repræsentere genstande og skabninger, som spilleren kan interagere med, bruger vi den abstrakte klasse `Item`:

```
type Item =
  class
    inherit Entity
    abstract member InteractWith : Player -> bool
    member FullyOccupy : bool
  end
```

Den måde en spiller interagerer med et `Item` på, er ved at gå ind i `Item` (det kommer vi tilbage til i næste delopgave). Til dette skal vi bruge `FullyOccupy` til at sige om `Item` fylder feltet helt ud eller om spilleren kan stå i samme felt som genstanden. Metoden `InteractWith` bruges dels til at genstanden kan have effekter på spilleren, og dels så siger retur-værdien om genstanden stadigvæk skal være i verden (`true`) efter interaktionen, eller om den skal fjernes (`false`) fra verden.

Implementér følgende fem konkrete klasser der nedarver fra `Item`:

- `Wall` der fylder et helt felt, men ellers ikke har effekter på spilleren.
- `Water` der ikke fylder feltet helt ud, og helbreder med to hit points.
- `Fire` der ikke fylder feltet helt ud, og giver ét hit point i skade ved hver interaktion med spilleren. Når spilleren har interageret fem gange med ilden går den ud.
- `FleshEatingPlant` der fylder feltet helt ud, og giver fem hit point i skade ved hver interaktion med spilleren.
- `Exit` vejen ud af dungeon!

6.10.3.3: Verden

Implementer klassen `World`:

```
type World =
  class
    new : ...
    member AddItem : item:Item -> unit
    member Play : unit -> unit
  end
```

Metoden `AddItem` bruges til at befolke verden med ting som spilleren kan interagere med. Typisk inden spillet går i gang.

Metoden `Play` bruges til at starte spillet, og tager sig af interaktionen med brugeren via terminalen. Spillet er tur-baseret og foregår på følgende vis:

(a) Vis hvordan verden ser ud, samt om der er eventuelt er sket noget for spilleren

(b) Hent brugerens træk som gives ved brug af pile-tasterne.

(c) Afgør hvilke `Items` som brugeren eventuelt interagerer med, samt hvad det betyder for hvad spillerens position og helbred er.

(d) Hvis spilleren er død eller hvis spilleren har fundet `Exit` vis et afslutningsskærmbillede og stop spillet, ellers start forfra.

Klassen `World` samt de andre klasser fra de andre delopgaver skal være i filen `roguelike.fs`. Lav derudover en fil `roguelike-game.fsx`, der som minimum laver en ny verden og kalder `Play`.

Basal Storyline

Den mest basale udgave af spillet: Spilleren starter et sted i et underjordisk dungeon, og skal finde udgangen. Når spilleren finder udgangen skal de have mindst fem *hit points* for at kunne tvinge døren op og undslippe dungeon.

Det er op til jer hvordan dungeon skal se ud, hvor spilleren starter, samt hvor mange genstande og skabninger der er i verden.

Hints:

- Det er en vigtig pointe at `World` ikke tager sig af at rendere spilleren og `Items` i verden, men blot skaber en canvas, som de forskellige `Entry` kan rendere sig selv på.
- Brug `System.Console.Clear()` at fjerne alt fra terminalen inden verden vises.
- Brug `Console.ReadKey(true)` til at hente et træk fra brugeren
- Hvis `key` er resultatet fra `Console.ReadKey` så er `key.Key` lig med `System.ConsoleKey.UpArrow`, hvis brugeren trykkede på op-pilen.

6.10.3.4: Udvidelser

Lav mindst 2 udvidelser til spillet og beskriv dem i jeres rapport. Følgende er nogle forslag til udvidelser, men I må gerne selv lade fantasien råde.

I er ligeledes velkommen til at udvide storyline.

- Teleport, lav en teleport der flytter spilleren fra et sted i verden til et (evt tilfældigt) andet sted i verdenen.

- Udvid `Item` så de kan påvirke verdenen. Fx, så kunne `FleshEatingPlant` sætte en stikling (en ny `FleshEatingPlant`) i et ledigt felt ved siden af den, hver tredje tur den ikke interagerer med spilleren.
- Monstre der kan bevæge sig rundt i verden, fx tilfældigt hvis de er langt fra spilleren, men går mod spilleren hvis de er tæt på.
- Udvid `Player` med et *inventory*, så man kan samle ting op i verden og flytte rundt på dem. Det kan fx bruges til at spilleren skal finde en nøgle for at komme gennem en dør.
- Udvid `Canvas` til at kunne vise emoji. Det kan gøres ved at hvert felt kan indeholde en `string` frem for kun en `char`, og så skal I være opmærksomme på at emoji ofte fylder det samme som to almindelige tegn.
- Skriv en level-generator (stor udvidelse!)
- Gør det muligt at indlæse et level fra en tekstfil
- Udvid Player-klassen med hhv. `Hunger` og `Thirst`.
  Tilføj, fx, `Food` og `WaterBottle` som `Items`. For hvert træk bliver `Player` mere sulten og tørstig. `Player` dør, hvis `Player` løber tør for enten mad eller vand.
- Giv `Player` en bue/magi/et sværd og gør det muligt at slås med monstre.
- Tilføj krukker der kan ødelægges. Krukkerne indeholder måske guld, som `Player` kan samle op.

6.11 ricochet-robots

6.11.1 Teacher's guide

Emne Classes, Objects, Methods Attributes

Sværhedsgrad Middel

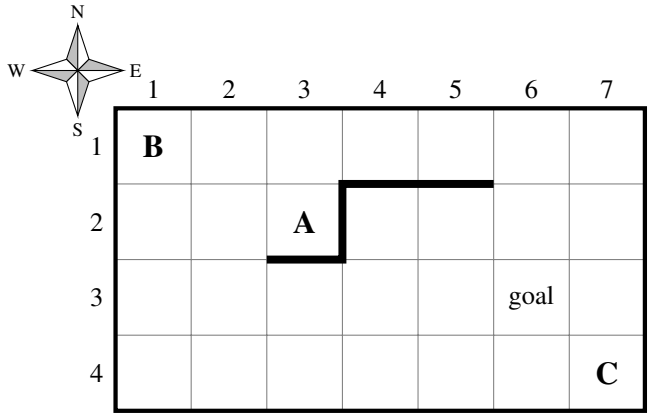
6.11.2 Introduction

Denne opgave går ud på at implementere en variant af brætspillet *Ricochet Robots* (først udgivet i Tyskland under navnet *Rasende Roboter* `http://en.wikipedia.org/wiki/Ricochet_Robot`).

I denne opgave skal der arbejdes med at lave et objekt-orienteret design, som gør det nemt at udvide spillet med nye regler og elementer.

Opgaven er delt i fire dele. Den første delopgave går ud på at vise en splilleplade i terminalen. Anden delopgave går ud på at lave et klasse-hierarki til at repræsentere forskellige spileelementer bl.a. robotter. Endelig skal der i den tredie delopgave arbejdes mod at sætte de forskellige dele sammen til et samlet spil. Fjerde del indeholde en række forslag til udvidelser, hvoraf I skal implementere mindst to.

I det følgende er der kun givet minimums-krav til hvilke metoder og properties I skal implementere på jeres klasser. I må gerne lave ekstra metoder eller hjælpe-funktioner, hvis I synes det kan hjælpe jer med at skrive et mere elegant og forståeligt program.



Spilleplade med 4×7 felter.

Robotter: **A** i felt (2,3), **B** i felt (1,1) og **C** i felt (3,6).

Indre vægge, tre stk:
vertikal, 1 felt, øst, startfelt (2,3);
horisontal, 1 felt, syd, startfelt (2,3);
horisontal, 2 felter, syd, startfelt (1,4).

Målfelt: (3,6)

Figure 6.4: Eksempel startposition

Rapport

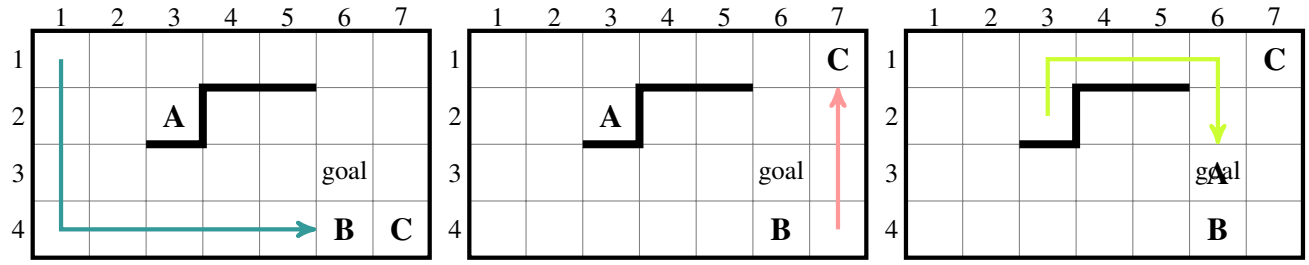
Ud over jeres programkode skal I også aflevere en rapport (skrevet i LaTeX). I rapporten skal I beskrive implementeringen af jeres klasser, det vil sige hvilken skjult tilstand (interne variable og lignende), som jeres metoder arbejder på.

Ligeledes skal rapporten indeholde et UML-klassediagram over klasserne i jeres løsning.

Spillets basisregler

Spillet foregår på en *plade* med $r \times c$ felter, det vil sige r rækker og c kolonner, hvor et antal *robotter* kan flyttes rundt. Hver robot starter i et separat felt på pladen, og kan derefter flyttes ved at *glide* i en af de fire retninger *nord*, *syd*, *øst* eller *vest*. Målet med spillet er at få flyttet en af robotterne hen til et *målfelt* i det laveste antal træk. Udfordringen er at når en robot starter med at glide i en retning så fortsætter den med at glide i den retning indtil, at den rammer enten en væg eller en anden robot. For at gennemføre et spil, skal en robot stoppe i et målfelt. Det tæller ikke som en løsning, hvis en robot blot glider gennem målfeltet.

Figur 6.4 viser et eksempel på en startposition for et spil. Dette spil kan fx løses i seks træk: flyt **B** syd, flyt **B** øst, flyt **C** nord, flyt **A** nord, flyt **A** øst, flyt **A** syd. Der er 24 forskellige løsninger på 6 træk til spillet i Figur 6.4, og ingen løsninger der er kortere end 6 træk.



6.11.3 Exercise(s)

6.11.3.1: Visning af spilleplade

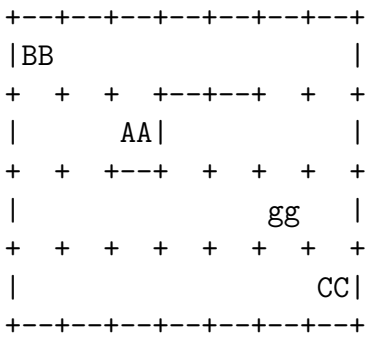


Figure 6.5: Eksempel på visning af spilleplade.

For at kunne vise en spilleplade implementerer vi en klasse `BoardDisplay`, som er et gitter af felter. Hvor feltet i øverste ventre hjørne har position $(1,1)$, og første koordinat tælles op når man bevæger sig fra nord til syd (top mod bund) og anden koordinat tælles op fra vest mod øst.

En plade har altid alle ydre vægge. For at håndtere indre vægge, så kan hver felt have en nedre væg og en højre væg, men ikke andre vægge. Hvert felt kan vise en tekststreng på maksimalt to tegn. Se Figur 6.5 for at se en visning af spillet fra Figur 6.4.

Implementér klassen `BoardDisplay` med følgende signatur:

```
type BoardDisplay =
  class
    new : rows:int * cols:int -> BoardDisplay
    member Set : row:int * col:int * cont:string -> unit
    member SetBottomWall : row:int * col:int -> unit
    member SetRightWall : row:int * col:int -> unit
    member Show : unit -> unit
  end
```

Det vil sige:

- En konstruktør der tager antal rækker og koloner som argumenter.
- en metode `Set` til at sætte indhold i et felt.
- to metoder `SetBottomWall` og `SetRightWall` til at sætte indre vægge for et felt.
- en metode `Show` til at vise en canvas i terminalen.

I rapporten skal I beskrive jeres designovervejelser, samt redegøre for hvilke data klassen `BoardDisplay` har.

6.11.3.2: Spilelementer

Vi bruger den abstrakte klassen `BoardElement` til at repræsentere spilelementer og klassen `Robot` til repræsentere robotter. Tag udgangspunkt i følgende erklæringer:

```
type Direction = North | South | East | West
type Action =
  | Stop of Position
  | Continue of Direction * Position
  | Ignore

[<AbstractClass>]
type BoardElement() =
```

```
    abstract member RenderOn : BoardDisplay -> unit
    abstract member Interact : Robot -> Direction -> Action
    default __.Interact _ _ = Ignore
    abstract member GameOver : Robot list -> bool
    default __.GameOver _ = false

and Robot(row:int, col:int, name:string) =
    inherit BoardElement()
    member this.Position = ...
    override this.Interact other dir = ...
    override this.RenderOn display = ...
    member val Name = ...
    member robot.Step dir = ...
```

Hvor typen `Direction` bruges til at angive en retning i forhold til spillepladen. Typen `BoardElement` bruges til at repræsentere spilelementer som ikke er robotter, denne klasse har tre abstrakte metoder: `RenderOn` til at rendere et element på et `BoardDisplay`; `GameOver` som bruges til at afgøre om et spil er gennemført, baseret på robotternes positioner; `Interact` som bruges at afgøre et spilelements indflydelse på en robot, inden den flyttes. Typen `Action` bruges til at specificere hvilken indflydelse et spilelement har på en robots flytning.

Klassen `Robot` bruges til at repræsentere robotter. Robotter er også spilelementer, da de kan påvirke andre robotter. Det vil sige, at klassen skal implementere metoderne fra `BoardElement`. Derudover har en robot:

- To properties: `Position` et par, (r,c), der angiver hvor på pladen robotten er og `Name` der angiver robottens navn, som blandt andet kan bruges til at rendere en robot på et `BoardDisplay`.
- Metoden `Step` som bruges at flytte robotten et felt i retningen `dir`. Dette er blot en hjælpemetode til at ændre `Position`, da `Step` *ikke* skal tage hensyn til andre spilelementer.
- Metoden `Interact` bruges til at stoppe en anden robot der forsøges at flytte ind i robotten. Fx, hvis `other` (argumenter til `Interact`, den anden robot) står i felt $(1,4)$ og er på vej vest, og robotten (`this`) står i felt $(1,3)$, så skal `Interact` returnere `Stop(1,4)`.

Implementér følgende fire konkrete klasser, der alle nedarver fra `BoardElement`, til at repræsentere indre og ydre vægge samt målfelt:

- `Goal` der skal have en konstruktor der tager to heltals argumenter, r og c, som angiver et målfelt. Metoden `GameOver` skal returnere `true`, hvis en robot er stoppet ovenpå målfeltet.
- `BoardFrame` bruges til at repræsentere rammen for en spilleplade (de ydre vægge). Klassen skal have en konstruktor der tager to heltals argumenter, r og c, som angiver størrelsen på pladen.
- `VerticalWall` bruges til at repræsentere en indre vertikal væg. Klassen skal have en konstruktor der tager tre heltals argumenter, r, c og n, som angiver startfelter for væggen, (r,c), samt længden af væggen, n. Hvis n er positiv løber væggen fra nord til syd, ellers løber den fra syd mod nord. Væggen er på østsiden af startfeltet.
- `HorizontalWall` bruges til at repræsentere en indre horisontal væg. Klassen skal have en konstruktor der tager tre heltals argumenter, r, c og n, som angiver startfelter for væggen,

(r, c), samt længden af væggen, n. Hvis n er positiv løber væggen fra vest til øst, ellers løber den fra øst mod vest. Væggen er på sydsiden af startfeltet.

Hvis I får behov for det må I gerne tilføje tilstand (data og properties) samt flere metoder til alle klasser. Men de skal kunne bruges med de angivende minimums specifikationer.

6.11.3.3: Interaktion

Implementer klassen `Board`:

```
type Board =
  class
    new : ...
    member AddRobot : robot:Robot -> unit
    member AddElement : element:BoardElement -> unit
    member Elements : BoardElement list
    member Robots : Robot list
    member Move: Robot -> Direction -> unit
  end
```

Metoden `AddElement` bruges til at sætte en spilleplade op. Typisk inden spillet går i gang. Property `Elements` bruges til at få en liste af alle spilelementer (inklusiv robotter), og `Robots` bruges til at få en liste af alle robotter. Et `Board` har altid et `BoardFrame` spilelement.

Metoden `Move` bruges til at flytte en robot. En robot flyttes ved at der fortages et antal skridt med robotten i en givet retning. Inden hvert skridt løbes gennem alle spilelementer (undtagen robotten selv), og metoden `Interact` kaldes for hvert element. Hvis alle spilelementer returnere `Ignore` kan robotten flyttes eet felt i den givne retning, og robotten forsøges at flyttes endnu et felt. Hvis et spilelement returnerer `Stop pos`, stoppes robottens flytning i felt `pos` (som ikke nødvendigvis er robottens nuværende position). Hvis et spilelement returnerer `Continue dir pos` fortsætter robotten fra felt `pos` med retning `dir` (bemærk at ingen af de obligatoriske spilelementer bruger `Continue`).

Implementer klassen `Game`:

```
type Game =
  class
    new : Board -> Game
    member Play: unit -> int
  end
```

Metoden `Play` bruges til at starte spillet, og tager sig af interaktionen med brugeren via terminalen, returværdien er hvor mange træk der blev brugt. Spillet foregår på følgende vis:

(a) Vis hvordan pladen ser ud, hvor mange træk der er brugt indtilvidere, navnene på robotterne, samt evt anden information som I finder relevant.

(b) Lad brugeren vælge en robot (fx ved at skrive navnet på robotten), herefter kan robotten flyttes rundt ved brug af pile-tasterne indtil at der tastes enter.

(c) Når en robots træk er slut, får alle spilelementer mulighed for at afgøre om et spil er slut. Hvis spillet er slut, vis et afslutningsskærmbillede og stop spillet.

Klasserne `Board` og `Game`, samt de andre klasser fra de andre delopgaver, skal være i filen `robots.fs` i modulet `Robots`. Lav derudover en fil `robots-game.fsx`, der som minimum laver et spil og kalder `Play`, så vi kan prøve dit spil.

Hints:

- Det er en vigtig pointe at `Board` *ikke* holder styr på hvor de forskellige spilelementer er, det skal de selv holde styr på. Ligeledes skal `Board` *ikke* tage sig af at rendere spilelementer, men skal blot skabe et `BoardDisplay`, og bagefter bede de forskellige spilelementer om at rendere sig selv på det.
- Brug `System.Console.Clear()` at fjerne alt fra terminalen inden brugergrænseflade vises.
- Brug `Console.ReadKey(true)` til at hente tryk på piletasterne fra brugeren
- Hvis `key` er resultatet fra `Console.ReadKey` så er `key.Key` lig med `System.ConsoleKey.UpArrow`, hvis brugeren trykkede på op-pilen.

6.11.3.4: Udvidelser

Lav mindst to udvidelser til spillet og beskriv dem i jeres rapport. Følgende er nogle forslag til udvidelser, men I må gerne selv lade fantasien råde. Hvis I laver udvidelser som kræver ændringer til de eksisterende typer og klasser, så skal I diskutere hvorfor denne slags ændringer kan være problematiske i jeres rapport. Hvis I laver nye spilelementer, så skal jeres rapport indeholde en regelbeskrivelse for de nye elementer som hvis man spillede en fysisk udgave af spillet, fx hvad der sker hvis to robotter ender i samme felt (kan ikke fysisk lade sig gøre).

- Teleport, lav en teleport der flytter en robot fra et sted i verden til et andet sted i verdenen.
- Vægge der kan bevæge sig.
- Bomber som der eksploderer ved sammenstød med en robot, og fjerner nærtliggende vægge.
- Udvid `BoardDisplay` og `Game` til at kunne vise farver og emoji. Vær opmærksom på at emoji ofte fylder det samme som to almindelige tegn. Brug

  ```
  System.Console.BackgroundColor <- System.ConsoleColor.Blue
  ```

  til, fx, at sætte baggrundsfarven til blå.
- Mulighed for at loade forskellige spil/startposition fra en fil.

Chapter 7

F# Event-driven Programming

7.1 IO

7.1.1 Teacher's guide

Emne Input/output

Sværhedsgrad Let

7.1.2 Introduction

7.1.3 Exercise(s)

7.1.3.1: Write a program `myFirstCommandLineArg` which takes an arbitrary number of arguments from the command line and writes each argument as, e.g.,

```
$ mono myFirstCommandLineArg.exe a sequence of args
4 arguments received:
0: "a"
1: "sequence"
2: "of"
3: "args"
```

The program must exit with the status value 0.

7.1.3.2: Make a program `myFirstReadKey` which continuously reads from the keyboard using the `System.Console.ReadKey()` function. The following key-presses must result in the following:

'a' writes "left" to the screen

's' writes "right" to the screen

'w' writes "up" to the screen

'z' writes "down" to the screen
shift+'q' quits the program

All other key-presses must be ignored. When the program exits, the exit status must be 0.

7.1.3.3: Make a program `myFirstReadFile` which

(a) opens the text file "myFirstReadFile.fsx" as a stream using the `System.IO.File.OpenText` function,

(b) reads each individual character using the `System.IO.StreamReader.Read` function,

(c) writes each character to the screen using the `printf` function, and

(d) closes the stream using `System.IO.FileStream.Close`.

The *program's* exit status must be 1 in case of error and 0 otherwise.

7.1.3.4: Make a program `myFirstWriteFile` which

(a) opens a new text file "newFile.txt" as a stream using the `System.IO.File.CreateText` function,

(b) writes the characters 'a' ... 'z' one at a time to the file using `System.IO.StreamReader.Write`, and

(c) closes the stream using the `System.IO.FileStream.Close` function.

The program's exit status must be 1 or 0 depending on whether there was an error or not when running the program.

7.1.3.5: Write a function,

```
filenameDialogue : question:string -> string
```

which initiates a dialog with the user using the `question`. The function should return the filename the user inputs as a string. If the user wishes to abort dialogue, then the user should input an empty string.

7.1.3.6: Make a program with the function,

```
printFile : unit -> unit
```

which initiates a dialogue with the user using `filenameDilaogue` from Exercise 5. The function must ask the user for the name of a file, and if it exists, then the content is to be printed to the screen. The program must return 0 or 1 depending on whether the specified file exists or not.

7.1.3.7: Make a program with the function,

```
printWebPage : url:string -> string
```

which reads the content of the internetpage `url` and returns its content as a string option.

7.1.3.8: Make a calculator program

```
simpleCalc : unit -> unit
```

which starts an infinite dialogue with the user. The user must be able to enter simple expressions of positive numbers. Each expression must consist of a value, one of the binary operators `+`, `-`, `*`, `/`, and a value. When the user presses `<enter>`, the the expression is evaluated and the result is written as `ans = <result>` with the correct result entered. The input-values can either be a positive integer or the string "ans", and the string "ans" should be the result of the previous evaluated expression or 0, in case this is the first expression typed. As an example, a dialogue could be as follows:

```
$ simpleCalc
>3+5
ans=8
>ans/2
ans=4
```

Here we used the character > to indicate, that the program is ready to accept input.

If the input is invalid or the evaluation results in an error, then the program should give an error message, and the input should be ignored.

7.1.3.9: Make a program with a function,

```
fileReplace :
  filename:string -> needle:string -> replace:string -> unit
```

which replaces all occurrences of the string `needle` with the string `replace` in the file `filename`. Your solution must use the `System.IO.File.OpenText`, `ReadLine`, and `WriteLine` functions.

7.2 Web

7.2.1 Teacher's guide

Emne Input/output from the web

Sværhedsgrad medium

7.2.2 Introduction

The internet is a great source of information, and many files are published as html-files. In the following assignment(s), you are to work with html-files on the internet.

Note that most internet pages requires a valid certificate before they will allow your program to access it. By default, Mono has no certificates installed. One way to install useful certificates is to use `mozroots`, which is a part of the Mono package. On Linux/MacOS you do the following from the console:

```
mozroots --import --sync
```

On Windows you type the following (on one line)

```
mono "C:\Program Files (x86)\Mono\lib\mono\4.5\mozroots.exe" --import
--sync
```

Note that your installation of `mozroots` may be in a different path, and you may have to adapt the above path to your installation. After running the above, your program should be able to read most pages without being rejected.

7.2.3 Exercise(s)

7.2.3.1: Make a program with the function,

```
printWebPage : url:string -> string option
```

which reads the content of the internetpage `url` and returns its content as a string option.

7.2.3.2: In the html-standard, links are given by the `<a></a>` tags. For example, a link to Google's homepage is written as `<a href="http://google.com">Press to go to Google</a>`.

Make a program `countLinks` which includes the function

```
countLinks : url:string -> int
```

The function should read the page given in `url` and count how many links that page has to other pages. You should count by counting the number of `<a` substrings. The program should take a url, pass it to the function and print the resulting count on the screen. In case of an error, then the program should handle it appropriately.

7.3 WinForms

7.3.1 Teacher's guide

Emne WinForms

Sværhedsgrad Middel

7.3.2 Introduction

7.3.3 Exercise(s)

7.3.3.1: Lav et program, som åbner et vindue og skriver teksten "Hello World" i vinduet vha. en `Label`.

7.3.3.2: Lav et program, som åbner et vindue og vha. `TextBox` beder brugeren indtaste sin vægt m i kilogram og højde h i meter, udregner body-mass-index efter formlen $bmi = h/m^2$, og skriver resultatet i vinduet vha. `Label`.

7.3.3.3: Lav et program, som beder brugeren om navnet på en input-tekstfil og en output-tekstfil vha. `OpenFileDialog` og `SaveFileDialog`, indlæser inputfilen og gemmer den i omvendt rækkefølge, så sidste bogstav bliver det første og første bliver det sidste.

7.3.3.4: (a) Lav et program, der tegner en streg mellem 2 punkter i et vindue.

(b) Opdater 4a, således at efter kort tid så slettes den gamle streg, og en ny tegnes tæt på den forrige. Hvert endepunkt skal parametriseres som en vektor (x,y), og det skal følge en ret linje parametriseret ved (dx,dy) og

$$(x_{i+1},y_{i+1}) = (x_i,y_i) + \alpha(dx,dy) \tag{7.1}$$

hvor α er en lille konstant. Hvis et endepunkt (x_{i+1},y_{i+1}) er udenfor vinduets tegnbare areal, skal punktet ignoreres og istedet skal der vælges en ny vektor (dx,dy) tilfældigt og et nyt endepunkt skal udregnes.

7.4 Clock

7.4.1 Teacher's guide

Emne WinForms

Sværhedsgrad Middel

7.4.2 Introduction

7.4.3 Exercise(s)

7.4.3.1: Der skal laves en grafisk repræsentation af et analogt ur i WinForms. Uret skal have en urskive, visere for timer, minutter og sekunder og det skal opdateres minimum 1 gang per sekund. Desuden skal uret vise dato og tid på digital form.