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

Jon Sporring, Martin Elsman, Torben Mogensen, Christina Lioma

June 23, 2020

Contents

Chapter 1

Scratch

1.1 Getting Started

1.1.1: Installér Scratch, Emacs og LaTeX

1.1.2: Install Scratch on your machine.

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

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

1.1.5: Make a program, which counts down from 10 to 1. The countdown must first start, when you press the mouse.

1.1.6: Make a program, which counts up fra 0 to 20 but only even numbers.

1.1.7: Tag et eller flere skærmbilleder af jeres program, mens det kører.

1.1.8: Skriv en kort rapport i LaTeX vha. Emacs og oversæt den til pdf via kommandoterminalen. Rapporten skal som minimum indeholde:

- En titel ved brug af `\maketitle`,
- et afsnit vha. `\section`,
- en eller flere figurer med skærmbillederne fra jeres Scratch program med `\begin{figure}` og `\end{figure}`, og som har en figurtekst,
- en henvisning til figuren ved brug af `\label` og `\ref` strukturen.
- de danske bogstaver æ, ø, og å.

1.2 Game

1.2.1: Hvad kan I lave med 10 blokke?
I Figur 1.1 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

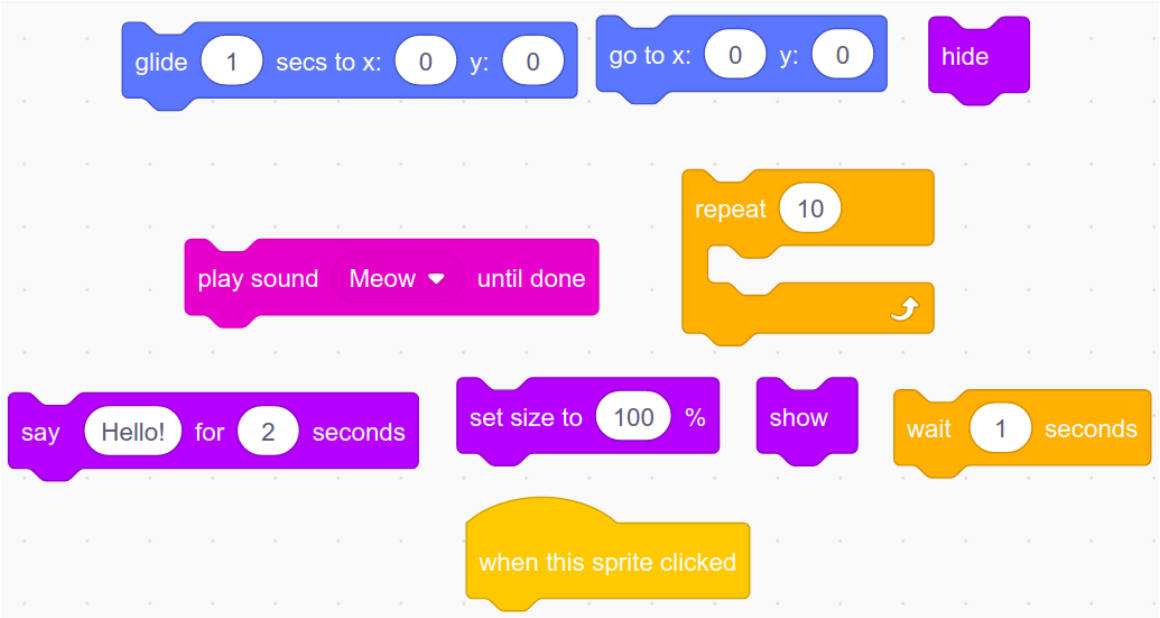


Figure 1.1: 10 Scratch-blokke

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.

1.2.2: 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 2

F# Imperative Programming

2.1 My first Fsharp program

2.1.1: Start en interaktiv F# session og indtast følgende (efterfulgt af ny linie):

Listing 1: My first F#.

```
3.14+2.78;;
```

Beskriv (for dig selv), hvad F# gjorde, og hvis der opstod en fejl, find fejlen og gentag.

2.1.2: Gentag øvelsen ovenfor, men denne gang indtast udtrykket i Emacs og gem det i en fil med suffix `.fsx`. Kør filen med `fsharpi` og `fsharpc+mono`. Overvej om resultatet er som forventet, og hvis ikke, forklar hvorfor.

2.1.3: Skriv et udtryk, som konkatenerer strengene `"hello"`, `" "`, `"world"` og afprøv det i F#.

2.1.4: Prøv følgende udtryk i F#,

Listing 2: Problematic F#.

```
3 + 1.0;;
```

og forklar resultatet. Forbedr evt. udtrykket.

2.1.5: Brug papir og blyant til at skriv heltallet 3_{10} op på binær form ved at bruge division-med-2 algoritmen. Skriv heltallet 1001_2 op i 10-talssystemet med gang-med-2 algoritmen. Skriv heltallet 47_{10} op på hexadecimal og på oktal form.

2.1.6: Indtast 47_{10} på decimal, hexadecimal, oktal, og floating-point form i fsharp, og verificer, at de alle repræsenterer den samme værdi.

2.1.7: Udfyld følgende tabel,

Decimal	Binær	Hexadecimal	Oktal
10			
	10101		
		3f	
			73

således at hver række repræsenterer den samme værdi men opskrevet på 4 forskellige former, og angiv mellemregningerne du brugte, for at udregne konverteringerne.

2.1.8: Opskriv sandhedstabellen (truth table) for udtrykket a **or** b **and** c, hvor a, b og c er boolske værdier.

2.1.9: Betragt F#-udtrykket `164uy+230uy`. Forklar hvad `"uy"` betyder, udregn udtrykket i F# og diskutér resultatet.

2.1.10: Opskriv et F#-udtryk for en streng, som indeholder "abc...æøå" udelukkende ved brug af unicode escapekoder.

2.1.11: Opskriv et F#-udtryk, som indicerer det 3. element og substrengen fra det 2. til 4. element i strengen "abcdef".

2.1.12: Opskriv to F#-udtryk, som ved brug af indiceringssyntaksen udtrækker 1. og 2. ord i strengen "hello world".

2.1.13: Betragt F#-udtrykket `"hello\nworld\n"`. Forklar hvad "\n" betyder, evaluér udtrykket i F# og diskutér resultatet.

2.1.14: Opskriv et F#-udtryk for en streng, som indeholder "\n" men hvor "\n" ikke opfattes som en escapekode. Hvor mange forskellige måder kan det gøres på?

2.2 Bindings

2.2.1: Indtast følgende program i en tekstfil, oversæt og kør programmet

Listing 3: Værdibindinger.

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

Forklar hvad parentesen i kaldet af `printfn` funktionen gør godt for. Tilføj en linje i programmet, som udregner udtrykket $ax+b$ og binder resultatet til `y`, og modificer kaldet til `printfn` så det benytter denne nye binding. Er det stadig nødvendigt at bruge parentes?

2.2.2: Listing 3 benytter F#'s letvægtssyntaks (Lightweight syntax). Omskriv programmet (enten med eller uden y bindingen), så det benytter regulær syntaks.

2.2.3: Følgende program,

Listing 4: Strenge.

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

skulle skrive "Hello Jon Sporring!" ud på skærmen, men det indeholder desværre fejl og vil ikke oversætte. Ret fejlen(e). Omskriv programmet til en linje (uden brug af semikolonner). Overvej hvor mange forskellige måder, dette program kan skrives på, hvor det stadig gør brug af bindingerne `firstName` `lastName` `name` og `printfn` funktionen.

2.2.4: Tilføj en funktion

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

til Listing 3, hvor `a`, `b` og `x` er argumenter til udtrykket $ax+b$, og modificer kaldet til `printfn` så det benytter funktionen istedet for udtrykket `(a * x + b)`.

2.2.5: Brug funktionen udviklet i Opgave 4, således at du udskriver værdien af funktionen for $a = 3$, $b = 4$ og $x = 0 \ldots 5$ ved brug af 6 `printfn` kommandoer. Modificer nu dette program vha. af en `for` løkke og kun en `printfn` kommando. Gentag omskrivningen men nu med en `while` løkke.

2.2.6: Lav et program, som udskriver 10-tabellen på skærmen, således at der er 10 søjler og 10 rækker formateret som

	1	2	…	10
1	1	2	…	10
2	2	4	…	20
⋮				
10	10	20	…	100

hvor venstre søjle og første række angiver de tal som er ganget sammen. Du skal benytte to `for` løkker, og feltbredden for alle tallene skal være den samme.

2.2.7: Som en variant af Opgave 6, skal der arbejdes med funktionen

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

som tager 1 argument og returnerer en streng indeholdende de første $1 \leq n \leq 10$ linjer i multiplikationstabellen inklusiv ny-linje tegn, således at hele tabellen kan udskrives med et enkelt `printf "%s"` statement. F.eks. skal kald til `mulTable 3` returnere

Listing 5: Eksempel på brug og output fra `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
```

hvor alle indgange i tabellen har samme bredde. Opgaven har følgende delafleveringer:

(a) Lav

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

så den som lokal værdibinding benytter en og kun en streng, der indholder tabellen for $n = 10$, og benyt streng-indicering til at udtrække dele af tabellen for $n < 10$. Afprøv `mulTable n` for $n = 1, 2, 3, 10$.

(b) Lav

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

så den benytter en lokal streng-variabel, som bliver opbygget dynamisk vha. 2 `for` løkker og `sprintf`. Afprøv `loopMulTable n` for $n = 1, 2, 3, 10$.

(c) Lav et program, som benytter sammenligningsoperatoren for strenge "=", og som skriver en tabel ud på skærmen med 2 kolonner: `n`, og resultatet af sammenligningen af `mulTable n` med `loopMulTable n` som `true` eller `false`.

(d) Forklar forskellen mellem at benytte `printf "%s"` og `printf "%A"` til at printe resultatet af `mulTable`.

2.2.8: Fakultetsfunktionen kan skrives som,

$$n! = \prod_{i=1}^{n} i = 1 \cdot 2 \cdot \ldots \cdot n \tag{2.1}$$

(a) Skriv en funktion

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

som benytter en `while` løkke, en tællevariabel og en lokal variabel til at beregne fakultetsfunktionen.

(b) Skriv et program, som beder brugeren indtaste et tal `n`, læser det fra tastaturet, og derefter udskriver resultatet af `fac n`.

(c) Hvad er det største n, som funktionen kan beregne fakultetsfunktionen for, og hvad er begrænsningen? Lav en ny version,

```
fac : n:int -> int64
```

som benytter `int64` istedet for `int` til at beregne fakultetsfunktionen. Hvad er nu det største n, som funktionen kan beregne fakultetsfunktionen for?

2.2.9: Betragt følgende sum af heltal,

$$\sum_{i=1}^{n} i. \tag{2.2}$$

Man kan ved induktion vise, at $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$, $n \geq 0$. Opgaven har følgende delafleveringer:

(a) Skriv en funktion

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

som ud over tælleværdien benytter en lokal variabel `s` og en `while` løkke til at udregne summen $1 + 2 + \cdots + n$.

(b) Lav en funktion

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

som i stedet benytter formlen $\frac{n(n+1)}{2}$.

(c) Skriv et program, som beder brugeren indtaste et tal `n`, læser det fra tastaturet, og derefter udskriver resultatet af `sum n` og `simpleSum n`.

(d) Lav et program, som skriver en tabel ud på skærmen med 3 kolonner: `n`, `sum n` og `simpleSum n`, og et passende antal rækker. Verificer ved hjælp af tabellen at de 2 funktioner beregner til samme resultat.

(e) Hvad er det største n de 2 versioner kan beregne `sum` funktionen korrekt for? Hvordan kan programmet modificeres, så funktionen kan beregnes for større værdier af n?

2.3 RGB

2.3.1: 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.

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

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

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

2.3.5: 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.

2.4 Vec

2.4.1: Skriv et bibliotek `vec2d.fs`, som implementerer signatur filen givet i Listing 6.

Listing 6 vec2d.fsi:
A signature file.

```
1  /// A 2 dimensional vector library.
2  /// Vectors are represented as pairs of floats
3  module vec2d
4  /// The length of a vector
5  val len : float * float -> float
6  /// The angle of a vector
7  val ang : float * float -> float
8  /// Multiplication of a float and a vector
9  val scale : float -> float * float -> float * float
10 /// Addition of two vectors
11 val add : float * float -> float * float -> float * float
12 /// Dot product of two vectors
13 val dot : float * float -> float * float -> float
```

2.4.2: Skriv en White-box afprøvning af biblioteket.

2.4.3: Punkter på en cirkel med radius 1 kan beregnes som $(\cos\theta, \sin\theta)$, $\theta \in [0, 2\pi)$. Betragt det lukkede polygon, som består af $n > 1$ punkter på en cirkel, hvor $\theta_i = \frac{2\pi i}{n}$, $i = 0..(n-1)$. Skriv et program med en funktion,

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

som benytter ovenstående bibliotek til at udregne længden af polygonet. Længden udregnes som summen af længden af vektorerne mellem nabopunkter. Programmet skal desuden udskrive en tabel af længder for et stigende antal værdier n, og resultaterne skal sammenlignes med omkredsen af cirklen med radius 1. Ud fra tabellen, hvad ser det ud til at der sker med længden af polygonet, når $n \to \infty$?

2.4.4: Biblioteket `vec2d` tager udgangspunkt i en repræsentation af vektorer som par (2-tupler). Lav et udkast til en signaturfil for en variant af biblioteket, som ungår tupler helt. Diskutér eventuelle udfordringer og større ændringer, som varianten ville kræve, både for implementationen og programmet.

2.5 Modules

2.5.1: 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.

2.5.2: Skriv en implementation af signaturfil fra Opgave 1 og kompiler begge filer til et bibliotek (`dll`-fil).

2.5.3: Skriv 2 programmer: Et som benytter biblioteket udviklet i Opgave 1 og 2 via `fsharpi` og et som benytter det via `fsharpc`.

2.5.4: Lav en White-box afprøvning af jeres bibliotek fra Opgave 2 og ved brug af `fsharpc`.

2.5.5: Udvid biblioteket (både signatur- og implementationsfilen fra Opgave 1 og 2) 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.

2.6 Lists

2.6.1: Skriv en funktion `oneToN : n:int -> int list`, som returnerer listen af heltal `[1; 2; ...; n]`.

2.6.2: Skriv en funktion `multiplicity: x:int -> xs:int list -> int`, som tæller antallet af gange tallet `x` optræder i listen `xs`.

2.6.3: Skriv funktionen `split: xs:int list -> (xs1: int list) * (xs2: int list)`, som deler listen `xs` i 2 og returnerer resultatet som en tuple, hvor alle elementer med lige index er i første element og resten i andet element. F.eks. `split [x0; x1; x2; x3; x4]` skal returnere `([x0; x2; x4], [x1; x3])`.

2.6.4: Definer en funktion `reverseApply : x:'a -> f:('a -> 'b) -> 'b`, sådan at kaldet `reverseApply x f` returnerer resultatet af funktionsanvendelsen `f x`.

2.6.5: Forklar forskellen mellem typerne `int -> (int -> int)` og `(int -> int) -> int`, og giv et eksempel på en funktion af hver type.

2.6.6: Brug `List.filter` til at lave en funktion `evens : lst:int list -> int list`, der returnerer de lige heltal i liste `lst`.

2.6.7: Brug `List.map` og `reverseApply` (fra Opgave 4) til at lave en funktion `applylist : lst:('a -> 'b) list -> x:'a -> 'b list`, der anvender en liste af funktioner `lst` på samme element `x` for at returnere en liste af resultater.

2.6.8: Opskriv typerne for funktionerne `List.filter` og `List.foldBack`.

2.6.9: 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.

2.6.10: Brug `Array.init` til at lave en funktion `squares: n:int -> int []`, sådan at kaldet `squares n` returnerer arrayet af de *n* første kvadrattal. For eksempel skal `squares 5` returnere arrayet `[|1; 4; 9; 16; 25|]`.

2.6.11: Skriv en funktion `reverseArray : arr:'a [] -> 'a []` ved brug af `Array.init` og `Array.length`, og som returnerer arrayet med elementerne i omvendt rækkefølge af `arr`. For eksempel skal kaldet `printfn "%A" (reverseArray [|1..5|])` udskrive `[|5; 4; 3; 2; 1|]`.

2.6.12: Brug en while-løkke og overskrivning af array-elementer til at skrive en funktion `reverseArrayD : arr:'a [] -> unit`, som overskriver værdierne i arrayet `arr`, så elementerne kommer i omvendt rækkefølge. Sekvensen

```
let aa = [|1..5|]
reverseArrayD aa
printfn "%A" aa
```

skal altså udskrive `[|5; 4; 3; 2; 1|]`.

2.6.13: Brug `Array2D.init`, `Array2D.length1` og `Array2D.length2` til at lave en funktion `transpose : 'a [,] -> 'a [,]` som returnerer det transponerede argument, dvs. spejler det over diagonalen.

2.6.14: En tabel kan repræsenteres som en ikke tom liste af lister, hvor alle listerne er lige lange. Listen `[[1; 2; 3]; [4; 5; 6]]` repræsenterer for eksempel tabellen

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

(a) Lav en funktion `isTable : llst:'a list list -> bool`, der givet en liste af lister afgør, om det er en lovlig ikke-tom tabel. For at det er en lovlig ikke-tom tabel, skal der gælde følgende:

- Der er mindst en liste med mindst et element.
- Alle lister i tabellen har ens længde.

(b) Lav en funktion `firstColumn : llst:'a list list -> 'a list`, der tager en liste af lister og returnerer listen af førsteelementer i de indre lister. F.eks. skal `firstColumn [[1; 2; 3]; [4; 5; 6]]` returnere listen `[1; 4]`. Hvis en eller flere af listerne er tomme, skal funktionen returnere den tomme liste af heltal `[] : int list`.

(c) Lav en funktion `dropFirstColumn : llst:'a list list -> 'a list list`, der tager en liste af lister og returnerer en liste af lister, hvor førsteelementerne i de indre lister er fjernet. F.eks. skal `dropFirstColumn [[1; 2; 3]; [4; 5; 6]]` returnere `[[2; 3]; [5; 6]]`.

(d) Lav en funktion `transpose : llst:'a list list -> 'a list list`, der spejler tabellens indgange over diagonalen, så den transponerede tabel til den herover viste tabel er

$$\begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}$$

Kaldet `transpose [[1; 2; 3]; [4; 5; 6]]` skal altså returnere `[[1; 4]; [2; 5]; [3; 6]]`. Bemærk, at `transpose (transpose t) = t`, hvis `t` er en tabel. Tip: Brug funktionerne `firstColumn` og `dropFirstColumn`.

2.6.15: Brug funktionerne opremset i [Kapitel 11, Sporring] til at definere en funktion `concat : 'a list list -> 'a list`, der sammensætter en liste af lister til en enkelt liste. F.eks. skal `concat [[2]; [6; 4]; [1]]` give resultatet `[2; 6; 4; 1]`.

2.6.16: Brug funktionerne fra [Kapitel 11, Sporring] til at definere en funktion `gennemsnit : float list -> float option`, der finder gennemsnittet af en liste af kommatal, såfremt dette er veldefineret, og `None`, hvis ikke.

Chapter 3

F# Functional Programming

3.1 Recursion

3.1.1: Skriv en funktion, `fac : n:int -> int`, som udregner fakultetsfunktionen $n! = \prod_{i=1}^{n} i$ vha. rekursion.

3.1.2: Skriv en funktion, `sum : n:int -> int`, som udregner summen $\sum_{i=1}^{n} i$ vha. rekursion. Lav en tabel som i Opgave 3i0 og sammenlign denne implementation af `sum` med `while`-implementation og `simpleSum`.

3.1.3: Skriv en funktion, `sum : int list -> int`, som tager en liste af heltal og returnerer summen af alle tallene. Funktionen skal traversere listen vha. rekursion.

3.1.4: Den største fællesnævner mellem 2 heltal, t og n, er det største heltal c, som går op i både t og n med 0 til rest. Euclids algoritme[1] finder den største fællesnævner vha. rekursion:

$$\gcd(t,0) = t, \tag{3.1}$$
$$\gcd(t,n) = \gcd(n, t \ \% \ n), \tag{3.2}$$

hvor % er rest operatoreren (som i F#).

(a) Implementer Euclids algoritm, som en rekursive funktion

```
gcd : t:int -> n:int -> int
```

(b) lav en white- og black-box test af den implementerede algoritme,

(c) Lav en håndkøring af algoritmen, gerne på papir, for `gcd 8 2` og `gcd 2 8`.

3.1.5: Lav dine egne implementationer af `List.fold` og `List.foldback` ved brug af rekursion.

3.1.6: Benyt `List.fold` og `List.foldback` og dine egne implementeringer fra Opgave 5 til at udregne summen af listen `[0 .. n]`, hvor `n` er et meget stort tal, og sammenlign tiden, som de fire programmer tager. Diskutér forskellene.

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

3.2 Continued Fractions

3.2.1: Skriv en rekursiv funktion

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

som tager en liste af heltal som kædebrøk og udregner det tilsvarende reelle tal.

3.2.2: Skriv en rekursiv funktion

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

som tager et reelt tal og udregner dens repræsentation som kædebrøk.

3.2.3: 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.

3.2.4: 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)`.

3.2.5: Saml alle ovenstående funktioner i et bibliotek bestående af dets interface og implementationsfil (`continuedFraction.fsi` `continuedFraction.fs`), og lav en applikationsfil, der udfører en white- og black-box test af funktionerne.

3.3 Sort

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

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.

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.)

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.

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.)

3.4 Random Text

3.4.1: 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.

3.4.2: 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
```

3.4.3: 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.

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

3.4.5: 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`

3.4.6: 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,

$$\text{diff}(h_1, h_2) = \frac{1}{N} \sum_{i=0}^{N-1} (h_1(i) - h_2(i))^2 \tag{3.3}$$

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.

3.4.7: Extend your library with the function

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

which counts occurrences of each pairs of lower-case letter of the English alphabet including space in a string and returns a list of lists (a table). In the returned list, the first element should be a list of the counts of 'a' being the initial character, i.e., how many times "aa", "ab", "ac",...,"az", "a " was observed. The second list should containt the counts of combinations starting with 'b', i.e., how many times "ba", "bb", "bc",... was observed and so on. The function should include overlapping pairs, for example, the input string "abcd" has the pairs "ab", "bc", and "cd".

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

3.4.9: 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`.

3.4.10: 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,

$$\text{diff2}(c_1, c_2) = \frac{1}{N^2} \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} \left(c_1(i,j) - c_2(i,j)\right)^2 \tag{3.4}$$

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.

3.4.11: 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?

3.4.12: 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 3.1 Words are to be represented as

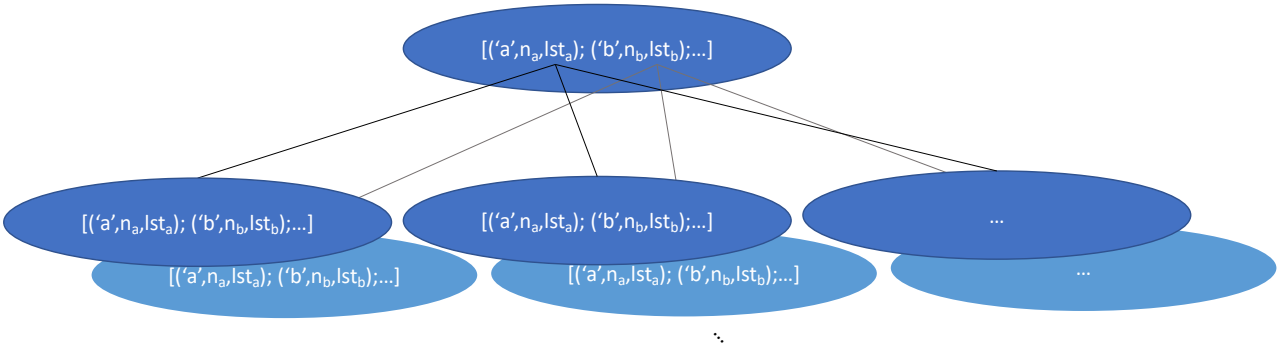


Figure 3.1: An illustration of a list of values of the type `Tree`.

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
```

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

3.4.14: 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`.

3.4.15: 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,

$$\text{diffw}(t_1,t_2) = \frac{1}{M}\sum_{i=0}^{M-1}\left(t_1(i)-t_2(i)\right)^2 \tag{3.5}$$

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.

3.4.16: 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.

3.4.17: 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.

3.5 Imgutil

3.5.1: Ved at benytte biblioteket `ImgUtil`, som beskrevet i forelæsningen, er det muligt at tegne simpel liniegrafik samt fraktaler, som f.eks. 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 bmp len (x,y) =
  if len < 25 then setBox blue (x,y) (x+len,y+len) bmp
  else let half = len / 2
       do triangle bmp half (x+half/2,y)
       do triangle bmp half (x,y+half)
       do triangle bmp half (x+half,y+half)

do runSimpleApp "Sierpinski" 600 600 (fun bmp -> triangle bmp
   512 (30,30) |> ignore)
```

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`.

3.5.2: I stedet for at benytte `ImgUtil.runSimpleApp` funktionen skal du nu benytte `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 -> System.Drawing.Bitmap)
          -> ('s -> System.Windows.Forms.KeyEventArgs
                 -> '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:

```
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 bitmap, som for eksempel kan konstrueres med funktionen `ImgUtil.mk` og ændres med andre funktioner i `ImgUtil` (f.eks. `setPixel`).

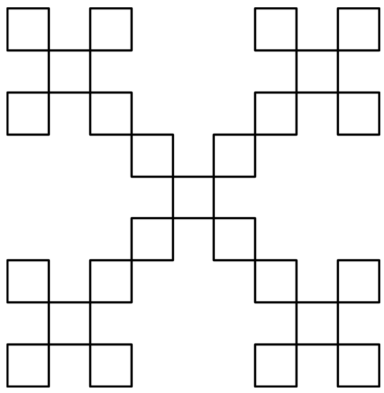
Funktionen `react`, som brugeren giver som 5. argument kaldes hver gang brugeren trykker på en tast. Funktionen tager som argument:

- en værdi svarende til den nuværende tilstand for applikationen, og
- et argument der kan benyttes til at afgøre hvilken tast der blev trykket på.[2]

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

Tilpas applikationen således at dybden af fraktalen kan styres ved brug af piletasterne, repræsenteret ved værdierne `System.Windows.Forms.Keys.Up` og `System.Windows.Forms.Keys.Down`.

3.5.3: 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).



[2] Hvis `e` har typen `System.Windows.Forms.KeyEventArgs` kan betingelsen `e.KeyCode = System.Windows.Forms.Keys.Up` benyttes til at afgøre om det var tasten "Up" der blev trykket på.

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.

3.6 Weekday

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

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

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

3.7 Sudoku

3.7.1: 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`.

3.8 Mastermind

3.8.1: 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.

3.9 Triangles

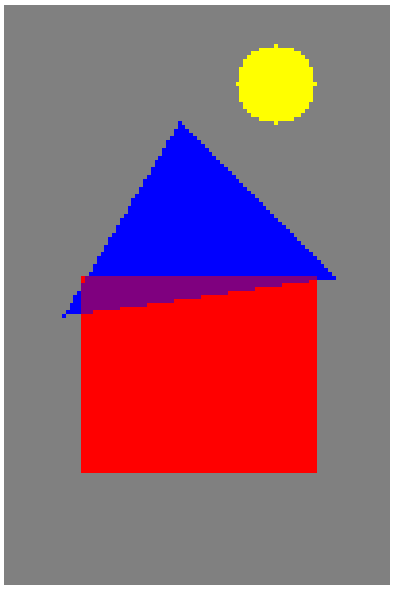
3.9.1: Lav en figur `figHouse : figure`, som består af en rød firkant udspændt af punkterne (20,70) og (80,120), en blå trekant udspændt af punkterne (15,80), (45,30) og (85,70), samt en gul cirkel med centrum (70,20) og radius 10.

3.9.2: 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.[3]

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

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

Resultatet skulle gerne ligne figuren nedenfor.



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

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

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

3.10 Awari

3.10.1: (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:

Listing 7 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 8 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.

3.11 Levensthein

3.11.1: 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).

3.11.2: 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` 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.

3.12 Polynomials

3.12.1: Skriv en funktion `poly: a:float list -> x:float -> float`, som tager en liste af coefficienter med `a.[i]` $= a_i$ og en x-værdi og returnerer 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.

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

3.12.3: Benyt Currying af `line` til at lave en funktion `theLine : x:float -> float`, hvor parametrene `a0` og `a1` er sat til det samme som brugt i Opgave 2. Afprøv `theLine` som Opgave 2.

3.12.4: Lav en funktion `lineA0 : a0:float -> float` ved brug af `line`, men hvor `a1` og `x` holdes fast. Diskutér om dette kan laves ved Currying uden brug af hjælpefunktioner? Hvis ikke, foreslå en hjælpefunktion, som vil gøre en definition vha. Currying muligt.

3.13 Integration

3.13.1: 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` er en integrabel 1 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`.

3.13.2: 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.

3.14 Exceptions

3.14.1: Implementer fakultetsfunktionen $n! = \prod_{i=1}^{n} i,\ n > 0$ som `fac : n:int -> int` og kast en `System.ArgumentException` undtagelse, hvis funktionen bliver kaldt med $n < 1$. Kald `fac` med værdierne $n = -4, 0, 1, 4$, og fang evt. untagelser.

3.14.2: Tilføj en ny og selvdefineret undtagelse `ArgumentTooBig of string` til `fac`, og kast den med argumentet `"calculation would result in an overflow"`, når n er for stor til `int` typen. Fang undtagelsen og udskriv beskeden sendt med undtagelsen på skærmen.

3.14.3: Lav en ny fakultetsfunktion `facFailwith : n:int -> int`, som `fac`, men hvor de 2 undtagelser bliver erstattet med `failwith` med hhv. argument `"argument must be greater than 0"` og `"calculation would result in an overflow"`. Kald `facFailWith` med $n = -4, 0, 1, 4$, fang evt. undtagelser vha. `Failure` mønsteret, og udskriv beskeden sendt med `failwith` undtagelsen.

3.14.4: Omskriv fakultetsfunktionen i Opgave 2, som `facOption : n:int -> int option`, således at den returnerer `Some m`, hvis resultatet kan beregnes og `None` ellers. Kald `fac` med værdierne $n = -4, 0, 1, 4$, og skriv resultatet ud vha. en af `printf` funktionerne.

3.14.5: Skriv en funktion `logIntOption : n:int -> float option`, som udregner logaritmen af `n`, hvis `n` > 0 og `None` ellers. Afprøv `logIntOption` for værdierne `-10`, `0`, `1`, `10`.

3.14.6: Skriv en ny funktion `logFac : int -> float option` vha. `Option.bind` 1 eller flere gange til at sammensætte `logIntOption` og `facOption`, og sammenlign `logFac` med Stirlings approximation `n * (log n) - n` for værdierne $n = 1, 2, 4, 8$.

3.14.7: Funktionen `logFac : int -> float option` kan defineres som en enkelt sammensætning af funktionerne `Some` og `Option.bind` en eller flere gange og med `logIntOption` og `facOption` som argument til `Option.bind`. Opskriv 3 udtryk, der bruger hhv. `|>` eller `>>` operatorerne eller ingen af dem.

3.14.8: 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
```

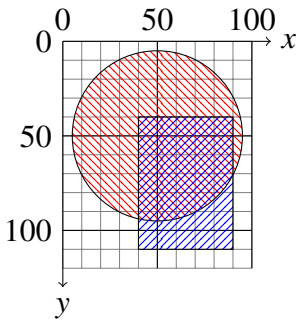
Each of them must return the value of `arr` at index `i`, when `i` is a valid index, and otherwise handle the error-situation. The error-situations must be handled in different ways:

- `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?

3.15 Tree structure

3.15.1: 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.)



3.15.2: Brug `ImgUtil`-funktionerne og `colourAt` til at lave en funktion

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

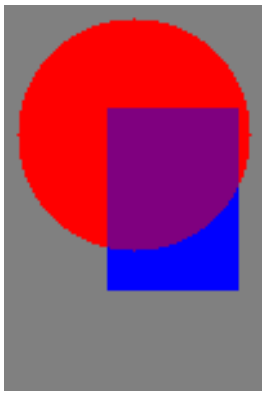
sådan at kaldet `makePicture` *`filnavn figur b h`* laver en billedfil ved navn *`filnavn`*`.png` med et billede af *`figur`* med bredde *`b`* og højde *`h`*.

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

Du kan bruge denne funktion til at afprøve dine opgaver.

3.15.3: Lav med `makePicture` 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.



3.15.4: Lav en funktion `checkFigure : figure -> bool`, der undersøger, om en figur er korrekt: At radiusen i cirkler er ikke-negativ, 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), og at farvekompenterne ligger mellem 0 og 255.

Vink: Lav en hjælpefunktion `checkColour : colour -> bool`.

3.15.5: 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
```

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

3.15.6: 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.

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

3.16 Cat

3.16.1: Make a 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`. The function should be placed in the implementation-file `readNWrite.fs`.

3.16.2: 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`.

3.16.3: First extend the implementation-file `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 a program, `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 0 or 1 depending on whether the operation was successful or not.

3.16.4: First extend the implementation-file `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 (i.e. the opposite of `cat` on a line-by-line basis) and concatenates the result. If any of the files do not exist, then the function should return `None`.

Then write a program, `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.

Chapter 4

F# Object-oriented Programming

4.1 Classes

4.1.1: 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, which creates 2 `student` objects and prints the name stored in each object using the "`.`"-notation.

4.1.2: 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, which 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`.

4.1.3: Implement a class `Counter`. The class must have 3 methods:

- The constructor must make a counter field whose value initially is 0,
- `get` which returns the present value of the counter field, and
- `incr` which increases the counter field by 1.

Write a white-box test class that tests `Counter`.

4.1.4: Implement a class `Car` with the following properties: A car has

(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.

4.1.5: 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:

- The constructor must accept the initial coordinates of the moth.
- `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 initial position.

Make a white-box test class and test the `Moth` class.

4.1.6: 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 methods `accelerate, break` so that the amount of gas left is reduced when the car accelerates or breaks. Call `accelerate, 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 `accelerate, brake` methods work properly.

4.1.7: 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. Assume that all drones fly at the same altitude and that drones fly with different speeds measured in meters/minute and in different directions. If 2 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 drone. Create an implementation file `simulate.fs`, and add to it a `Drone` class with properties and methods:

- The constructor must take start-position, -destination, and -speed.
- `position` (property): returns the drone's position in (x, y) coordinates.
- `speed` (property): returns the drone's present speed in meters/minute.
- `destination` (property): returns the drone's present destination in (x, y) coordinates. If the drone is not flying, then its present position and its destination are the same.
- `fly` (method): Set the drone's new position after 1 minutes flight.
- `isFinished` (method): Returns true or false depending on whether the drone has reached its destination or not.

Extend your implementaiton file with a class `Airspace`, which contains the drones and as a minimum has the 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 1 minute.
- `addDrone` (method): Add a new drone to the collection of drones.
- `willCollide` (method): Given a time interval, determine which drones will collide.

Write a white-box test class `testSimulate.fsx` that tests both the above classes.

4.1.8: 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.

4.2 Simple Jack

4.2.1: 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)
```

4.3 Object-oriented design

4.3.1: 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 BC, followed by 2 BC, etc., and the year prior is called 1 AD, preceded by 2 AD, etc. Thus, this calendar has no year 0.
>
> 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.

4.3.2: 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 4.1.

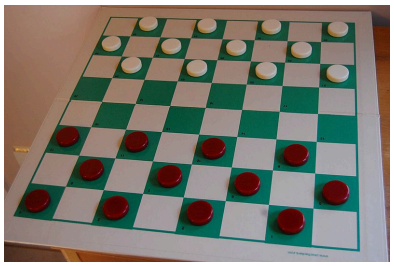


Figure 4.1: The starting position in Checkers [https://commons.wikimedia.org/wiki/File:CheckersStandard.jpg]

> 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.

4.3.3: 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.

4.4 Inheritance

4.4.1: 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 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.

4.4.2: 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.

4.4.3: (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.

Once you have written the classes, write a program that creates an object of the `ProductionWorker` class and prompts the user to enter data for each of the object's data properties. Store the data in the object and then use the object's methods to retrieve it 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.

4.4.4: 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).

4.5 Predator-Prey

4.5.1: 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 diagram

(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.

4.6 Wolves and mooses

4.6.1: 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.

4.7 Owls and mice

4.7.1: 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.

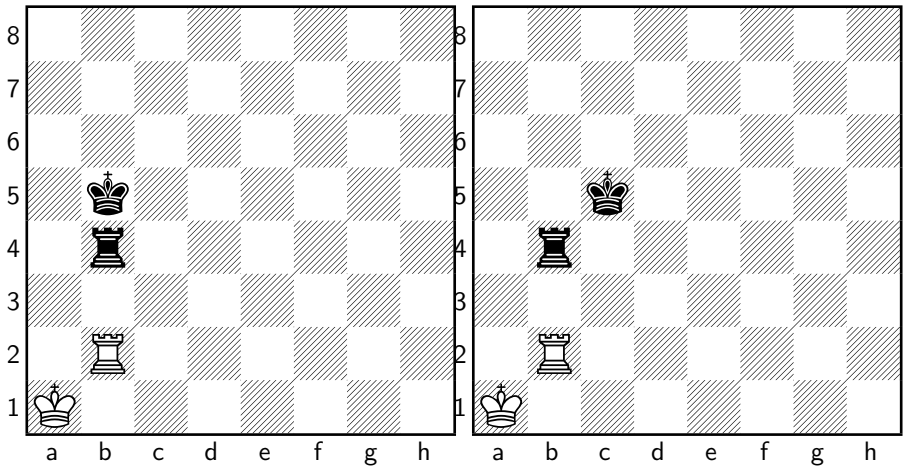
(g) The report must be no longer than 10 pages excluding the appendices.

4.8 Chess

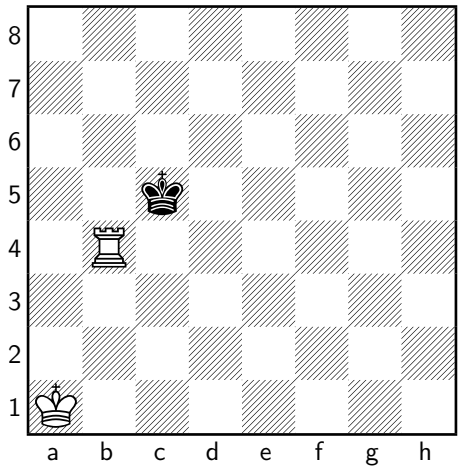
4.8.1: Produce a UML diagram describing the design presented of Simple Chess in the book.

4.8.2: 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.

4.8.3: 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.



(a) Initial position (b) b5 c5



(c) b2 b4

Figure 4.2: Starting at the left and moving white rook to b4.

4.8.4: 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".

4.8.5: 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 4.2(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 `-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 $\mathcal{O}(22^n)$.

4.8.6: Make an extended UML diagram showing the final design including all the extending classes.

4.9 UML

4.9.1: 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.

4.9.2: Make an UML 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.

4.9.3: Write a UML 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`.

Chapter 5

F# Event-driven Programming

5.1 IO

5.1.1: 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.

5.1.2: 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.

5.1.3: 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 or 0 depending on whether there was an error or not when running the program.

5.1.4: 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.

5.1.5: 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.

5.1.6: 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.

5.1.7: 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.

5.1.8: 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.

5.1.9: 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.

5.2 Web

5.2.1: 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.

5.2.2: 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="https://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.

5.3 WinForms

5.3.1: Lav et program, som åbner et vindue og skriver teksten "Hello World" i vinduet vha. en `Label`.

5.3.2: 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`.

5.3.3: 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.

5.3.4: (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

$$(x_{i+1},y_{i+1}) = (x_i,y_i) + \alpha(dx,dy) \tag{5.1}$$

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.

5.4 Clock

5.4.1: 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.