Et eksempel på en rapport til ugeopgave 8g i PoP 2015/2016

Ugeopgave 8g - Sudoku spil

1 Forord

Denne rapport om implementeringen af et Sudoku spil kan bruges som et eksempel på en rapport til ugeopgaven 8g. I kan med fordel benytte jer af andre skabeloner på Absalon, som er blevet brugt under udarbejdelse af denne rapport.

2 Introduktion

Sudoku er et populært puslespil, som blev introduceret i danske aviser et årti tilbage. Og siden har enhver avis altid mindst én Sudoku som man kan løse i toget, bussen eller hvor man nu har brug for at slå lidt tid ihjel. Det geniale ved Sudoku er, at man ikke behøver at være god til matematik for at løse den, og man behøver ikke at kende en masse ord som f. eks i en kryds og tværs. Da Sudokuer kan laves med forskellige sværhedsgrader, er det alle aldre der kan være med, lige fra børn i børnehaven til voksne genier.
At lave en Sudoku som et computerspil er derfor nærliggende og findes da også allerede i mange versioner som app til smartphones og computere. Vi vil til gengæld implementere et Sudoku spil i det funktionelle programmeringssprog `F#`.

3 Problemformulering

Vi skal implementere et Sudoku spil i det funktionelle programmeringssprog `F#`. Spillet skal kunne udskrive en Sudoku på skærmen, og brugeren skal kunne indsætte værdier ved at taste række, kolonne og kvadrant. Disse værdier skal tjekkes om de er lovlige i forhold til spillets regler. Spillet skal også kunne afgøre, om der er nået en korrekt løsning. Hvis brugeren er i tvivl om hvad han skal gøre skal spillet også kunne foreslå lovlige trippler som kan indsættes, og som overholder spillets regler.
Programmet skal også kunne læse en Sudoku fra en tekstfil som brugeren kan løse, og hvis han ikke bliver færdig skal han også have mulighed for at gemme det.

4 Problemanalyse og design

Vi har valgt at implementere Sudoku spillet i tre moduler, som udgør vores løsningforslag. Fordelene ved at have funktionerne lagret i moduler er, at det forbedrer strukturen af programmet, da funktionerne er fordelt efter deres funktionaliteter. Herved fremmes læseligheden, samt fremtidig vedligholdelse af programmet. Således har vi defineret følgende moduler med deres tilhørende funktionaliteter:

- FileHandling - Håndteringen af Input/Output mht. filerne, herunder kommunikation med disse.
- GameBoard - Her foregår kommunikationen og visning af data mellem brugeren og programmet.
- Sudoku - Indeholder logikken af programmet, såsom beregningerne af Sudoku spillet.

I de følgende afsnit vil vi uddybe hvert modul, som tilsammen skal munde ud i et løsningforslag til vores Sudoku spil.

4.1 FileHandling

Til behandlingen af fil input og output, vil vi skrive modulet `FileHandling`, som vil gøre det muligt for brugeren at læse et spil fra en tekstfil, samt gemme en tilstand i en fil. Starttilstanden er en `string`, som lagres i en tekstfil. Som en central datastruktur har vi overvejet enten at arbejde med strenge, eller evt. at omdanne filens indhold til en liste med 9 lister, hvor hver har 9 elementer. En liste udgør således en række, og i strengen ville det svare til hvert afsnit mellem linebreak. Uanset hvilken datastruktur der vælges, skal det indeholde 81 elementer, som samlet set repræsenterer et Sudoku spil.
Vi anser en liste med 9 lister som den mest hensigtsmæssige datastruktur til at løse problemet på, da den videre behandling af spillet vil være den mest effektive, samt mindst besværlige. Dette begrundes med, at det vil være muligt for os at foretage listeoperationer, som vi allerede er bekendt med. Derudover undgår vi, at lave type konverteringer af hvert element, hver gang vi foretager ændringer i spillet. Dette ville medføre en langsommere køretid.
Før Sudoku spillet kan modificeres af `Sudoku` modulet, se 4.3, kræver det en omdannelse fra et input som en `string` data type til et output som en `int list list`. Det vil sige en liste med lister bestående af hele tal. Proceduren skal tage et filnavn som argument, læse filens tilstand, og returnere det som en `int list list`. Ligeledes skal programmet have en procedure, som tager et filnavn og en `int list list` som argumenter, konverterer sidst nævnte fra en `int list list` tilbage til en `string`, og slutteligt gemmer resultatet af brugerens Sudoku spil i en fil.
Begge procedurer skal gøres tilgængelige for andre filer i et bibliotek, hvor hver er pakket i en wrapper funktion, hvorfra de tilhørende hjælpefunktioner kaldes. Således skal der til læsningen af en fil defineres en `loadSudoku` funktion, og for at gemme tilstanden i filen, en `saveSudoku` funktion.

4.2 Gameboard

Brugeren som anvender programmet, skal have overblik og skal kunne navigere både i hovedmenuen og menuen i spillet. Kommandoerne skal være nemt anvendelige, hovedloopet skal sørge for at kommunikationen foregår optimalt imellem bruger og program. For at danne en nemmere tilgang til benyttelse af programmet har vi lavet `Gameboard` modulet. Modulet har 2 funktioner, `output` og `parseTripple`. Disse funktioner indgår ikke i opgavens krav, men er dog alligevel lavet, for at gøre brugerens kommunikation med spillet nemmere. Dette skyldes at vi gerne vil have en brugergrænseflade, der gør spillet overskueligt. Kommandoer skal også være nemme at benytte sig af, f. eks. skal spillet indeholde kommandoer med en tilhørende beskrivelse som spillet kan udføre. Dette skal f. eks. være `load`, `save`, `hint` osv. Herved tænker vi, at lave en funktion der kan forstå input med rækker, søjle og værdi, hvor man kan skrive inputtet på forskellige måder, fortrinsvis (1,2,3) eller 123.

4.3 Sudoku

Der skal være mulighed for at indsætte en tripple i et Sudoku spil. Dvs. 3 tal, ét for række, ét for kolonne og ét for værdi. Hertil skal vi definere en funktion `replaceAt` som kan erstatte en værdi i en liste. Vi skal også kunne tjekke om spillet har nået en korrekt løsning, og også om spillet har nået en tilstand hvorfra en løsning er umulig.
Spillet skal kunne foreslå lovlige trippler, dvs. en værdi mellem 1 og 9, og et felt angivet med en række og en kolonne. Dette skal gøres, så der ikke står en værdi mere end én gang i hver række, kolonne og kvadrant. Man kunne let returnere en tilfældig tripple, som kunne indsættes nu og her, men som ikke nødvendigvis ville føre spilleren nærmere en løsning, men snarere føre ham til en løsning som ikke giver mening. Vi vil derfor søge igennem Sudoku spillet, og rent faktisk returnere den næste logiske løsning, som et menneske ville have gjort det. Derfor kan vi ikke bruge `Donald Knuth's algorithm X`, da den finder en løsning på hele spillet og ikke nødvendigvis bare det næste træk.
En svær sudoku kan sjældent løses med simple regler. Der findes 12 regler for et Sudoku spil[1], men det vil være omfattende at lave kode til alle 12 regler, så vi vil implementere 2 simple regler. I fald en af reglerne fejler, vil vi anvende brute force til at søge igennem hele Sudoku spillet. På den måde vil vi få et korrekt hint. Dette hint er ikke nødvendigvis det næste logiske træk, men vi anser det for at være bedre at hintsystemet altid returnerer et forslag.
Når først vi har et hintsystem som altid giver et rigtigt forslag til løsning af en Sudoku kan vi også let lave en `solve` funktion som giver en korrekt udfyldt Sudoku.

[1] `https://www.stolaf.edu/people/hansonr/sudoku/12rules.htm`

5 Programbeskrivelse

5.1 Læs og gem Sudoku filer

Til programmets kommunikation med tekstfiler er der blevet defineret to funktioner, hhv. `saveSudoku`, som gemmer tilstanden, og `loadSudoku`, som læser en tilstand fra en tekstfil. Begge funktioner er tilgængelige når biblioteket er blevet genereret, og inkluderet i ens programfil. Til disse funktioner er der defineret en række hjælpe funktioner i implementerings filen `FileHandling.fs`.

5.1.1 LoadSudoku

Funktionen `loadSudoku` består af én linje af kode, hvormed den indkapsler de øvrige hjælpefunktioner. Den tager et filnavn som argument, og såfremt at det er et gyldigt Sudoku spil, returnerer den indholdet af filen som en liste af lister. Funktionen `loadSudoku` bruger funktion i figur 1 til at åbne en eksisterende tekstfil. Ligeledes oprettes og returneres inde i funktionen et `StreamReader` objekt.

```
let readFile filename =
  let streamreader = System.IO.File.OpenText filename
  readFileContent streamreader
```

Figur 1: Funktionen `readFile`

Herefter læses indholdet vha. det oprettede objekt af filen i funktionen `readFileContent`. Som det fremgår i figur 2 kører funktionen rekursivt indtil den har læst den sidste linje i en fil, hvormed den har opnået enden af strømmen. Dermed er alt indhold blevet læst og gemt i den lokale funktion `fileContent`.

```
let rec readFileContent (streamreader : System.IO.StreamReader) =
  let fileContent = (streamreader.ReadToEnd())
  if not (streamreader.EndOfStream) then
    readFileContent streamreader
  else
    streamreader.Close()
    textToList fileContent
```

Figur 2: Funktionen `readFileContent`

Når objektet er blevet lukket, bliver alt indhold omdannet i funktionen `textToList` til en liste med lister med hele tal. Definitionen af den omtalte funktion ses i figur 3. Funktionen er indviklet, og kræver således en dybere forklaring. Den tager en `string` som argument, som skal udgøre et sudoku spil. Herefter erstattes `"*"` med `"0"`, og \n med et blankt mellemrum. Efter udførelsen af begge operationer vil den midlertidige `string` se ud som følgende

”800000000003600000070090200050007000000045700000100030001000068008500010090000400”

Vi har således fået strengen til at indeholde udelukkende hele tal. Efterfølgende har vi defineret den lokale rekursive funktion `convert`. Funktionen tager imod 3 argumenter, et `string` input, som eksempelvis kunne være ovenstående Sudoku spil. Endvidere tager den imod en liste, samt et helt tal som skal stå for start positionen i en `string`. Når funktionen kaldes, tjekker den først om længden af `lister` er 9, fordi den derefter ikke bør tilføje flere lister til listen, og således vil returnere listen af lister.

```
let textToList (fileContent : string) =
  let newcont = fileContent.Replace("*", "0")
  let newcont2 = newcont.Replace("\n", "")
  let rec convert (strInput : string) lister (start : int) =
    if ((List.length lister) = 9) then
      lister
    else
      let result = (convertList (strInput.Substring(start,9),0))
      if (checkInput result) then
        (convert strInput (lister@[result]) (start+9))
      else
        raise WrongContent
  (convert newcont2 [] 0)
```

Figur 3: Funktionen `textToList`

Hvis dette ikke er tilfældet, går den videre til at omdanne en `substring` af `strInput`, som udgør en række, dvs. de første 9 elementer af strengen. Selve omdannelsen foregår i funktionen `convertList`, se figur 4. Funktionen er defineret til at omdanne hvert tegn i strengen til et helt tal, og tilføjer det til en liste. Funktionen er defineret rekursivt, har to patterns og tager imod en tuple af to parametre. Her er `x` strengen, og `y` er indekset af den. Eksempelvis vil den ved et input som $(80000000, 0)$, begynde med at omdanne 8 til en `int`, og trække 48 fra. Dette skyldes at tegnet er en ASCII værdi. For at omdanne det til et decimal tal, skal 48 (30 hexadecimalt) trækkes fra. Elementet tilføjes til listen, og funktionen kalder sig selv igen og lægger én til indekset. Dette fortsættes indtil `y` er større end længden af strengen, og da det er 0 indekseret, trækkes 1 fra længden. På den måde bliver 9 elementer tilføjet, hvorefter funktionen afsluttes. For eksempel vil første række af inputtet, som blev brugt som eksempel, returneres som $[8; 0; 0; 0; 0; 0; 0; 0; 0]$.

```
let rec convertList = function
  | (x,y) when (y > (String.length x)-1) -> []
  | (x,y) -> (int (x.[y])-48)::(convertList (x,y+1))
```

Figur 4: Funktionen `convertList`

For hver gang der returneres et resultat af `convertList`, tjekkes dette i funktionen `checkInput`. Funktionen tjekker om alle elementerne i listen er mellem 0 og 9. Dette skyldes, at vi vil sikre os gyldigheden af værdierne ift. Sudokus regler. Normalt ville 0 ikke være hensigtsmæssig, men vi har i vores tilfælde defineret 0 til at være ikke løste felter. Denne procedure kører rekursivt, og hvis alle tallene ligger mellem 0 og 9, returneres der `true`.

```
let checkInput result = (List.forall (fun x -> x <= 9 && x >= 0) result)
```

Figur 5: Funktionen `checkInput`

I figur 3 fortsætter `convert` funktionen i rekursiv iteration, hvor der ved hvert kald tilføjes listen til `lister`, som indeholder alle listerne. Dette fortsættes indtil længden af listen af listerne er nået til 9, som så returnerer den samlede liste af lister.

5.1.2 SaveSudoku

Funktionen `saveSudoku` kan kaldes i de tilfælde brugeren ønsker at gemme sin spiltilstand i en fil. Funktionen er en indkapslet funktion, som kalder `saveFile` funktionen med to argumenter. Disse er hhv. `filename`, som er en streng, og `lister`, som er en liste af lister med hele tal. Dermed sagt kræver et kald af `saveSudoku` de to nævnte argumenter. I figur 6 ses implementeringen af `saveFile`.

```
let saveFile filename userInput =
  let toSave = listToText userInput
  let outputStream = System.IO.File.CreateText filename
  outputStream.Write toSave
  outputStream.Close()
```

Figur 6: Funktionen `saveFile`, som kræver to argumenter

Funktionen `userInput`, som er listen af lister, bliver vha. `listToText` funktionen omdannet fra en liste af lister til en streng, og til sidst returneret. Se figur 7 for implementeringen. Omdannelsen sker i fem funktioner, hvor hver funktion behandler det der blev returneret af den foregående. I første funktion omdannes hvert element i listen af listerne til `string` typer. Dernæst tilføjes der `\n` til hver liste. I `listEdit3` sættes alle listerne i listen sammen. I næste funktion bliver der vha. `foldBack` funktionen akkumuleret hvert element `x` i listen `acc` på en tom `string`. Resultatet af denne operation er en streng med et Sudoku spil. Til sidst bliver alle felter med 0 konverteret til *, således at der bliver returneret den oprindelige struktur af `string` indholdet.

```
let listToText lister =
  let listEdit = List.map (fun x -> (List.map (fun y -> string y) x)) liste
  let listEdit2 = List.map (fun x -> (List.append x ["\\n"])) listEdit
  let listEdit3 = List.concat listEdit2
  let listEdit4 = List.foldBack (fun x acc -> x+acc) listEdit3 ""
  let listEdit5 = listEdit4.Replace("0","*")
  listEdit5
```

Figur 7: Funktionen `listToText`

Når `listToText` er kørt og har returneret sit output, vil en `StreamWriter` til filnavnet blive oprettet. Herefter bliver strengen skrevet til strømmen, som overskriver det til filen. I fald filen ikke eksisterer, sørger `CreateText` objektet for at filen bliver oprettet. Slutteligt bliver den tilknyttede strøm lukket. Den returnerede værdi af `saveSudoku` funktionen er af typen `unit`, hvilket skyldes at den udfører en opgave, som bliver udført i en fil.

5.1.3 Øvrige funktioner

Ud over at have defineret de to forudgående funktioner, som danner fundamentet for fil kommunikationen, har vi defineret yderligere funktioner. Funktionen `checkFile` skal således tage imod et filnavn, tjekke om filen findes i samme mappe som kørefilen, hvorefter den returnerer en `bool` type. Funktionen kan med fordel drages nytte af i tilfælde af at man vil sikre sig at filen, som brugeren har tastet ind som input, eksisterer, inden man fortsætter med Sudoku spillet.
Herudover har vi defineret funktionen `checkFileName` som sørger for at brugeren kun bruger lovlige tegn i filnavnet, hvis brugeren vælger at gemme et Sudoku spil i en tekstfil. Disse regler er både defineret af os selv, samt operativ systemet. Eksempelvis er der specifikke tegn som operativ systemet Windows ikke tillader i filnavnet. Ydermere har vi ikke tilladt `.`, da vi vil sikre os at filtypen altid bliver en tekstfil, hvilket vi ligeledes har defineret i koden.
For at tjekke for disse tegn, har vi gjort brug af den indbyggede funktion `exists`, der tjekker hvert tegn af `filename` argumentet for en af de nedenstående tegn, se figur 8. Endvidere tjekkes der for at længden af filnavnet er mellem 3 og 255 tegn, med alle mellemrum trukket fra. Således har vi vha. af funktionen sikret os at inputtet fra brugeren er indenfor disse ovennævnte regler. I så fald returneres `true`.

```
let checkFileName filename =
  (not(String.exists (fun c -> (c='/') || (c='\\') || (c='*') || (c='?')
  || (c='<') || (c='>') || (c='|') || (c=':') || (c='.')) filename)
  && (String.length (filename.Trim()) >= 3)
  && (String.length (filename.Trim()) < 255))
```

Figur 8: Funktionen `checkFileName`

De to sidste funktioner `showAllTxtFiles` og `getCurrentPath` bidrager til at informere brugeren om hvilke tekstfiler der opbevares i samme mappe, som kørefilen befinder sig i. Dette giver brugeren en oversigt over de mulige filer, der kan læses.

5.2 Sudoku spillet

I hjertet af Sudoku spillet, finder vi funktioner som skal indsætte værdierne i Sudokuen og tjekke at inputtet er lovligt i forhold til spillets regler. Vi har også et hint system som kan hjælpe spilleren i retning af en rigtig løsning. Disse funktioner er samlet i modulet `Sudoku.dll`, som indeholder funktionerne `insertRsv`, `isFinished`, `isCorrect`, `getHint` og `solve`. Vi definerer også en `type Sudoku` som kan benyttes i stedet for at skrive `int list list`.

5.2.1 InsertRsv

Funktionen `insertRsv` sørger for at indsætte et tal i Sudokuen. Den tager en tripple som argument med række, kolonne og værdi og returnerer en Sudoku som int list list. `InsertRsv` bruger funktionen `getRow` til at returnere en række med et givent index, og herefter bruger den `replaceAt` til at erstatte det givne index med en ny værdi. Funktionen `replaceAt` er gjort generel, så den både kan bruges på en liste af lister og en liste af integers. Se figur 9.

```
let replaceAt index e list =
  let rec replace index e list acc =
    match list with
    | [] -> []
    | x0::xs -> if (index = acc) then
                  e::replace index e xs (acc+1)
                else
                  x0::replace index e xs (acc+1)

  replace index e list 0
```

Figur 9: Funktionen `replaceAt` som erstatter et element i en liste.

`InserRsv` skal sikre at den får en lovlig tripple, dvs. at hvis tallet 7 bliver indsat i et felt, så må tallet 7 ikke stå andetsteds i den samme række, kolonne og kvadrant som feltet. Dette gøres med `checkRsv` som ses i figur 10. Den tjekker først at række, kolonne og værdi er mellem 1 og 9. Dernæst tjekkes om feltet rent faktisk er tomt, dvs at feltet indeholder værdien 0. Nu bruges `getRow`, `getColumn` og `getQuadrant` til at få 3 lister med henholdsvis rækken, kolonnen og kvadranten for vores felt. `List.exists` bruges, til at tjekke at den ønskede værdi, som skal indsættes ikke er i de 3 lister.

```
let checkRsv (r:int, c:int, v:int) (s:Sudoku) =
  if r>=1 && r<=9 && c>=1 && c<=9 &&
    v>=1 && v<=9 && s.[(r-1)].[(c-1)] = 0 then
      let row = getRow (r-1) s
      let column = getColumn (c-1) s
      let q = (r-1)/3*3 + (c-1)/3
      let quad = getQuadrant q s
      not (List.exists (fun e -> e = v) row) &&
        not (List.exists (fun e -> e = v) column) &&
        not (List.exists (fun e -> e = v) quad)
  else
    false
```

Figur 10: Funktionen `checkRsv` som kalder `getRow`, `getColumn` og `getQuadrant` og tjekker om værdien brugeren har indtastet eksisterer i en af listerne.

Hvis alle tjek går godt returneres true ellers false.

5.2.2 IsFinished

Funktionen `isFinished` returnerer en bool med true, hvis vi har nået en korrekt løsning for Sudokuen. Hjælpefunktionen `checkList` som ses i figur 11 bruges til at tjekke at alle tal fra 1 til 9 er i den liste som gives som input.

```
let checkList list =
  let s = Set list
  Set.foldBack (fun x x0 -> x+x0) s 0 = (1+2+3+4+5+6+7+8+9)
```

Figur 11: Funktionen `checkList` som tjekker at alle tal fra 1 til 9 er repræsenteret i listen.

Dette gøres ved at konvertere listen til et Set og så bruge biblioteksfunktionen `Set.foldBack` til at lægge alle tal i sættet sammen. Hvis denne sum giver 45 $(1+2+3+4+5+6+7+8+9)$, så ved vi at alle tal fra 1 til 9 er i listen, og der returneres true. Grunden til at vi konverterer til et Set er, at hvis listen indeholdt værdierne 2,2,2,4,5,6,7,8,9 ville summen også give 45, men er ikke en tilladt liste. Ved at konvertere til et Set bliver listen til 2,4,5,6,7,8,9 og summen vil ikke give 45, fordi 1 og 3 mangler og der kun er ét 2-tal. På den måde sikrer vi os at alle tal er repræsenteret i listen.
Nu bruges `checkRows`, `checkColumns` og `checkQuadrants` til at udføre `checkList` på samtlige rækker, kolonner og kvadranter og hvis de alle returnerer true, så er vi nået en rigtig løsning.

5.2.3 IsCorrect

Vi har også brug for en funktion `isCorrect` til at tjekke at Sudokuen har nået en tilstand, hvorfra den ikke kan løses. Dette kunne være hvis brugeren har lavet en fejl og indsat en værdi, som ikke er korrekt. Funktionen løber alle indeks i Sudokuen igennem, først igennem kolonnerne og bagefter rækkerne. Hvis der i et felt er værdien 0, dvs. et tomt felt, bruges funktionen `findPossible` til at finde alle værdier der mangler i dette felt. Funktionen findPossible som ses i figur 12 bruger `getRow`, `getColumn` og `getQuadrant` til at finde de værdier, der allerede står i rækken, kolonnen og kvadranten. Dette giver 3 lister som sættes sammen og laves om til et Set. Herved får vi de unikke værdier som allerede er i Sudokuen. Dette sæt trækkes fra et Set indeholdende alle tal fra 1 til 9 og vi får nu de mulige værdier for dette felt.

```
let findPossible (r,c) (s:Sudoku) =
  if (s.[(r-1)].[(c-1)] = 0) then
    let q = (r-1)/3*3 + (c-1)/3
    let row = getRow (r-1) s
    let column = getColumn (c-1) s
    let quad = getQuadrant q s
    let uSet = Set.ofList (row @ column @ quad)
    if ((Set.count uSet) < 10) then
      Set.difference (Set [1;2;3;4;5;6;7;8;9]) uSet
    else
      Set [-1]
  else
    Set []
```

Figur 12: Funktionen `findPossible` som finder de mulige værdier i et felt

Hvis der ingen mulige værdier er returneres et sæt indeholdende -1 for at indikere en fejltilstand og hvis der allerede er en værdi i feltet returneres et tomt sæt. Hvis `isCorrect` ser at det Set som `findPossible` returnerer indeholder -1, så returneres false med det samme. Hvis hele Sudokuen er løbet igennem og `findPossible` altid finder 0,1 eller flere muligheder for et felt, så returneres true.

5.2.4 GetHint

Funktionen `getHint` returnerer en tripple, som kan hjælpe spilleren til at nå en løsning. Dette opnås ved at gennemsøge 2 regler og fejler det, laves en brute force søgning. Først bruges funktionen `buildSets`,

som tager en Sudoku som int list list og bruger `findPossible` til at returnere en Set<int> list list hvor hvert Set<int> indeholder de mulige værdier for dette felt. Dette er vores `SuggestionSet`. Den lokale funktion `gh` vil gennemsøge `SuggestionSet` og se, om der er et sæt, hvor der kun kan stå én værdi. Findes det, returneres denne værdi som et hint. Dette er en regel, som man benytter til at løse lette Sudokuer med.
En anden regel er, at se hvad der mangler i hvert felt i en række, kolonne eller kvadrant og hvis en værdi kun står i et felt i f. eks en række, så er vi sikre på at dette er en løsning, og vi kan returnere det som et hint. De 3 funktioner `hintFromRow`, `hintFromColumn` og `hintFromQuadrant` udfører denne anden regel. De fungerer alle på samme måde.
Hvis vi kigger på `hintFromRow`, vil den først tage en række af Set<int>'s fra suggestionSet med `getRow` og bruge `countSetList` til at tælle hvor mange af hvert tal fra 1 til 9, der er i de 9 Sets. Hjælpefunktionen `findUnique` bruges, til at se hvilken værdi der kun er én af og denne værdi kan returneres som et hint.
Hvis en af disse hint funktioner ikke kan finde et hint, returneres (-1,-1,-1). Disse funktioner kaldes én af gangen nederst i `getHint` funktionen og fejler de allesammen kaldes `searchHint` som er vist i figur 13.

```
let searchHint () =
  let rec sh = function
  | 10 -> (-1,-1,-1)
  | n -> let (s0, (r, c)) = findPair n suggestionSet
         if ((r,c) = (-1,-1)) then
           sh (n+1)
         else
           let sl = Set.toList s0
           let rec loopSuggestions = function
           | [] -> (-1,-1,-1)
           | [x] -> (r,c,x)
           | x0::xs -> let news = insertRsv (r,c, x0) s
                       if (testSudoku news) then
                         (r,c, x0)
                       else loopSuggestions xs

           let su = loopSuggestions sl
           if (su = (-1,-1,-1)) then
             sh (n+1)
           else su
  sh 2
```

Figur 13: Funktionen `searchHint` som er en lokal funktion i `getHint`

Det giver færrest søgninger, hvis vi starter med et felt hvor der kun er 2 mulige værdier, og `findPair` benyttes til at finde et Set med netop 2 elementer. I tilfælde af at der ikke er et felt med kun 2 værdier fortsættes til 3 værdier osv. Funktionen `findPair` returnerer række og kolonne samt sættet med de 2 værdier.
Dette sæt konverteres til en liste og det første element skilles ud og indsættes med `insertRsv`. Nu kaldes `testSudoku` som ses i figur 14, som bliver ved med at kalde `getHint` og `insertRsv` for at nå en vindende tilstand eller returnere en exception `NotLegalInput`.

```
and testSudoku (s:Sudoku) =
  let rec ts (su:Sudoku) =
    let (r,c,v) = try (getHint su)
                  with
                  | NoHint -> (-1,-1,-1)

    let news = try (insertRsv (r,c,v) su)
               with
               | NotLegalInput -> [[]]

    if (isFinished news) then
      true
    else if (v = -1) then
      false
    else ts news
  ts s
```

Figur 14: Funktionen `testSudoku` som er gensidigt rekursiv med `getHint`

Der returneres enten true eller false og `searchHint` kan returnere værdien som et hint, hvis den fik true. Hvis der retuneres false og det næste element i listen er det sidste, så returneres det som et hint. Det sparer os for at teste den anden værdi i parret. Hvis begge værdier ikke giver en løsning, så vil det senere hen vise sig da `insertRsv` vil fejle. Derfor kan vi godt acceptere denne løsning for at spare søgetid.
`getHint` og `testSudoku` er gjort gensidigt rekursive (mutual recursive), fordi de skal kalde hinanden. `searchHint` må kalde `testSudoku`, som igen kalder `getHint` for at komme til en løsning på Sudokuen. Dette gøres med `let rec getHint = ... and testSudoku = ...`
Hvis ikke det er muligt at løse sudokuen fra den tilstand som den har nået, og getHint kaldes, kan vi risikere at `getHint` returnerer en tripple med værdien -1. Dette kommer fra funktionen `findPossible` som returnerer et sæt med -1 hvis ikke der er mulige værdier for et felt. Dette er dog ikke noget problem, fordi `isCorrect` bliver kaldt i hovedloopet, og fanger en fejltilstand inden getHint kaldes.

5.2.5 Solve

`Solve` bruger samme princip som `testSudoku`, men returnerer istedet den færdige udfyldte Sudoku som int list list, som kan vises til spilleren hvis han har givet op.

5.3 Gameboard og Brugerinput

Da programmet skal kunne benyttes af en bruger, har vi defineret et modul, der skal gøre det muligt for brugeren at kommunikere med programmet. Modulet vi har lavet til dette er navngivet `Gameboard`, og der er lavet filen `Run` som oversættes til en `.exe` fil. Denne fil fungerer som det primære program, der kobler de definerede moduler sammen, hvormed Sudoku spillet spilles. I modulet Gameboard findes der følgende funktioner: `output` og `parseTripple`. I `Run.fsx` findes der: `playGame`, `showMainMenu` og `printSudoku`.

5.3.1 ParseTripple

Funktionen `parseTripple` er vist i figur 15. Funktionen gør det nemmere at skrive inputtet som skal benyttes af insertRsv. Dette gør den ved at konvertere en `string` til 3 `int` typer, som kan bruges som række, søjle og værdi. Med vores funktion er det ligemeget om man skriver (1,2,3), 123 eller (123) osv. Denne funktion benyttes når brugeren skal skrive sit input, når han vil indsætte en værdi i sudokuen. Efter at inputtet er kørt igennem `parseTripple`, vil den blive sendt til funktionen `insertRsv`, hvis kode er defineret i del 5.2.1.

```
let parseTripple (tripple: string) =
  let rec pt n =
    if (n = (String.length tripple)) then []
    else
         let c = tripple.Chars(n)
         if c = '(' then (pt (n+1))
         else if c >= '0' && c<='9' then (((int c)-48)::pt (n+1))
         else if c = ',' then (pt (n+1))
         else if c = ')' then (pt (n+1))
         else failwith "Not a triple"

  let numberList = pt 0
  (numberList.[0], numberList.[1], numberList.[2])
```

Figur 15: parseTripple funktionen, tager en `string` og filtrerer 3 int som laves til en triple

Ved hjælp af denne funktion opnår vi bedre kommunikation mellem brugeren og programmet, hvilket var vores ønske til Gameboard modulet.

5.3.2 Output

I figur 16 ses koden til `output` funktionen. For at gøre det nemmere for brugeren at interagere med spillet, tager vi vores Sudoku, som er en liste af lister og udskriver sudokuen på skærmen på en pæn måde således at den appellerer bedre til brugeren. Funktionen kræver en `int list list` ved hvert kald. Alle indekserne bliver kørt rekursivt igennem og ved hver 3. kolonne udskrives et `|`. Dette gøres vha. modulo operatoren. Modulo returnerer resten af en division, og hvis den er 0 ved operationen $n\%3$, ved vi at vi er er nået til 3. indeks. Dette gøres ligeledes for vores rækker og der udskrives passende skillelinier for hver kvadrant.

```
let output (s:int list list) =
  printfn "******Sudoku Spillet*****"
  let rec out = function
  | (8, 9) -> printfn "|\n+-------+-------+-------+"
  | (m, 9) -> printfn "|"
             out (m+1,0)
  | (m, n) -> if (m % 3 = 0 && n=0) then
                printfn "+-------+-------+-------+"
              if (n % 3 = 0) then
                printf "| "
              if (s.[m].[n] = 0) then
                printf "  "
              else
                printf "%d " (s.[m].[n])
              out (m, n+1)
  out (0,0)
```

Figur 16: Output funktionen, tager en `Sudoku` og danner en brugergrænseflade

Efter at en sudoku er sendt til output funktionen, vil spillefladen vises således:

```
+-------+-------+-------+
| 2     |       | 6     |
|       |       |       |
|       | 3     |       |
+-------+-------+-------+
|   4   |       |       |
|       |       |       |
| 1     |       |   7   |
+-------+-------+-------+
|       |   8   |       |
|       |       |       |
|       |       |       |
+-------+-------+-------+
```

5.3.3 PlayGame

Funktionen `playGame` i `Run.fsx` er den der holder styr på kommandoerne, og opdaterer Sudoku spillet. Funktionen bliver sat i gang efter `printSudoku`, hvorved den tager den indlæste Sudoku, og danner nye kommandoer. Ydermere printer den selve Sudokuen i terminalen. Til hvert kald af disse kommentarer er der tilknyttet funktioner. De nye kommandoer samt deres tilknyttede funktioner er:

- `save`: Kalder `saveSudoku`, beskrevet i del 5.1.2.
- `hint`: Kalder `getHint`, beskrevet i del 5.2.4.
- `solve`: Kalder `solve`, beskrevet i del 5.2.5.
- `menu`: Kalder `showMainMenu`, beskrevet i del 5.3.5.
- `exit`: Indbygget funktion, `exit 1` terminerer det kørende program.

Funktionen kan ses i appendix B, del 7.

5.3.4 PrintSudoku

Funktionen `printSudoku` er stadiet hvor brugeren indlæser den ønskede fil, den tager en `System.Console.ReadLine()` og tilføjer `.txt` til det indtastede. Derved kan `FileHandling` modulet indlæse filen, og Sudokuen kan udskrives vha. `Gameboard.output`. Herefter kaldes funktionen `playGame` på den indlæste Sudoku. Hvis filnavnet er ugyldigt, vil funktionen kalde sig selv, og spørge brugeren om at indtaste et andet filnavn.

5.3.5 ShowMainMenu

Denne funktion er det første der møder brugeren når programmet åbnes. Den giver et valg imellem at spille et Sudoku spil eller forlade programmet. Menuen bliver først printet så brugeren får en ide om hvordan han kan navigere. Brugeren kan navigere i spillet ved at indtaste et input i terminalen, som bliver tildelt `choice` funktionen. Funktionen matcher med et af valgmulighederne, og svarer på det tilsvarende input. Inputtet 1 fra brugeren vil få koden til at printe stien til mappen, samt printe mappens indeholdte `.txt` filer til brugeren. Inputtet 2 fra brugeren, er en simpel `exit` funktion som rydder konsollen.

```
let rec showMainMenu () =
  System.Console.Clear()
  printfn "----- Velkommen til Sudoku -----\n"
  printfn "1 - Spil et Sudoku spil"
  printfn "2 - Forlad programmet\n"
  printfn "Venligst vaelg en valgmulighederne"
  let rec mainMenuResponse () =
    System.Console.Write("\nVaelg en mulighed: ")
    let choice = (System.Console.ReadLine())
    match choice with
      | "1" -> printfn "\nDen aabnede mappe indeholder foelgende
                .txt filer i \n%s: \n\n%A"
                  FileHandling.getCurrentPath()
                    FileHandling.showAllTxtFiles ()
               printSudoku ()
      | "2" -> System.Console.Clear()
               exit 1
      | _ -> printfn "Not a valid choice"
             mainMenuResponse ()
  mainMenuResponse()
  showMainMenu()
```

Figur 17: showMainMenu, den første menu der bliver præsenteret for brugeren

Først printer koden information til brugeren, og venter derefter på et svar vha. `System.Console.ReadLine()`. Dette valg bliver sammenlignet i en pattern matching hvorved den ved valg 1, kalder på `FileHandling` modulet. Herefter kalder den så på `printSudoku()` funktionen. Valget 2 vil resultere i en `System.Console.Clear()` som rydder skærmen, og derefter terminerer programmet vha. `exit 1`. Menuen vil printe `"Not a valid choice"`, hvis inputtet ikke passer ind i en af de to valgmuligheder. Herefter vil funktionen kalde sig selv hvorved brugeren kan prøve igen.

6 Afprøvning

Vi vil afprøve vores moduler ved at bruge unit-tests. Vi giver vores funktioner nogle testinputs og tjekker at vi får det rigtige resultat. Dette kunne f.eks. være at funktionen kaster en `exception` for et input, som ikke er lovligt. Således skal det bidrage til at bekræfte korrektheden af vores program, ved at teste små enheder.

6.1 Sudoku modulet

Unittests for Sudoku modulet er implementeret i filen `SudoUnitTest.fsx`

6.1.1 InsertRsv

Vi starter med test1 som skal sikre at en rigtig værdi bliver indsat i Sudokuen. Og vi prøver at indsætte en ulovlig tripple i test4, se figur 18. Derfor returnerer vi `true` i `with`-blokken, da et korrekt resultat kaster en exception.

```
let test4 = try
              let x = (Sudo.insertRsv (3,4,1) s)
              false
            with
              | _ -> true
```

Figur 18: Test af `insertRsv`. Den ulovlige tripple (3,4,1) indsættes i Sudokuen s

6.1.2 IsFinished

Funktionen bliver testet med en korrekt udfyldt Sudoku, herefter med en der ikke er udfyldt og til sidst med en sudoku med fejl.

6.1.3 IsCorrect

IsCorrect bliver testet med 2 sudokuer som er forkerte. En med fejl i inputtet og en med en umulig løsning.

6.1.4 GetHint

For at teste `getHint` må vi bruge `insertRsv`. Hvis vi kalder getHint på en Sudoku kan vi være lidt i tvivl om hvilket hint den returnerer, da der godt kan være flere felter som kan give et hint. Derfor bruger vi `insertRsv` for at teste om vi har modtaget en lovlig tripple. Dette gøres i `test2`. True returneres hvis ikke insertRsv kaster en exception. `Test2` ses i figur 19

```
let test2 = try
              let (r,c,v) = Sudo.getHint wh
              try
                let x = (Sudo.insertRsv (r,c,v) wh)
                true
              with
                | _ -> false

            with
              | _ -> false
```

Figur 19: Test af `insertRsv`. Den ulovlige tripple (3,4,1) indsættes i Sudokuen s

Man skal være opmærksom på at `getHint` sagtens kan returnere et rigtigt hint på en sudoku der ikke kan løses, dvs. isCorrect returnerer false. Grunden til dette er at `getHint` returnerer det næste hint mens `isCorrect` gennemsøger hele sudokuspillet.

6.1.5 Solve

Vi benytter funktionen `isFinished` for at teste om solve kommer til en rigtig løsning. Der testes også om Solve kaster en exception hvis der ikke er en løsning og til sidst en direkte sammenligning af en løst Sudoku med resultatet for Solve.

6.1.6 Sammenfatning

Alle testene består. Testene er simple, men sikrer dels at funktionerne virker, men også at de udfører det stykke arbejde som opgaven stiller. Og de er med til at sikre, at koden stadig virker hvis man en gang i fremtiden skal opdatere sine funktioner. Det er desværre meget let at komme til at introducere nye fejl, når man har rettet en gammel.

6.2 FileHandling modulet

Vi har i filen `UnitTestFH` defineret unit tests til `FileHandling` modulet. Der er defineret i alt 4 tests, som hver tester outputtet af de kaldte funktioner. Fordi vi har anskuet `loadSudoku` og `saveSudoku` til at være søjlerne i modulet, har vi designet 3 ud af 4 tests, som skal vise deres korrekthed.

6.2.1 ListToText

Eksempelvis har vi defineret `test2` funktionen i figur 20, som tjekker de to nævnte funktioner. Således er `wh` et forud defineret Sudoku spil, i form af en liste med lister. Ikke desto mindre gemmer funktionen `test2` Sudoku spillet i en fil, og definerer herefter funktionen `loadGame` til at have indholdet af filen. Til

sidst tjekkes der om det læste indhold stemmer over ens med `wh` funktionen, hvis indhold blev gemt i samme fil.

```
let test2 =
  try
    let saveGame = (FileHandling.saveSudoku "SudokuLoadTest2.txt" wh)
    let loadGame = (FileHandling.loadSudoku "SudokuLoadTest2.txt")
    (wh = loadGame)
  with
    | _ -> false
```

Figur 20: Test af `listToText`

Den ovenstående test, vi har foretaget figur 20, har returneret `true`. Dette har bekræftet funktionenernes korrekthed.

6.2.2 textToList

Funktionen `textToList` blev testet i funktionen `test1`. Funktionen tjekker om der er 9 lister i den returnerede liste fra `loadSudoku`, samt om hver liste har 9 elementer. I de tilfælde hvor vi kaldte funktionen med et gyldigt Sudoku spil, blev der returneret `true`.
Desuden har vi tjekket udfaldet for de tilfælde, hvor der er et minus tal i spillet. Funktionen `test3` skal således returnere `true` hvis dette er tilfældet. Dette har vi testet, og blev opfyldte vores forventning.

Ved hjælp af de foretagne tests, har vi fået bekræftet korrektheden af både `loadSudoku` og `saveSudoku` funktionerne, herunder sammenspillet mellem hjælpefunktionerne.

6.2.3 checkFile

I funktionen `test4` har vi tjekket om `checkFile` returnerer `false`, hvis filen ikke findes i mappen som kørefilen. Dette har vi gjort med et tilfældigt input, som bekræftede at funktionen returnerer det korrekte output.

7 Diskussion og konklusion

Vi har fået lavet de tre moduler som blev beskrevet i afsnit 4, og modulerne virker som beskrevet i afsnittet. Vores Sudoku spil virker stabilt og gennemgået. Ved hjælp af funktionen `insertRsv` har vi fået implementeret en funktionalitet, som gør det muligt at indtaste en værdi i spillet. Dette gøres efter mønstret række, kolonne og kvadrant.
Herudover har vi defineret funktioner der både gemmer og læser spiltilstandene til og af tekstfiler. Ved hjælp af funktionen `getHint` kan brugeren få et hint til spillet. Slutteligt afgør `isCorrect` funktionen om et input er gyldigt.
Som højdepunkt af programmet anser vi `solve` funktionaliteten, der sågar har formået at løse verdens sværeste Sudoku spil.
Vi anskuer processen til at være lærerig og indsigtfuld i funktionel programmering. Vi har så vidt muligt forsøgt at anvende de tilegnede evner fra de forudgående uger. Sideløbende har vi også forsøgt at udvide vores kendskab til programmeringen. Således vurderer vi resultatet af vores arbejde som en succes.

Appendix A: Brugervejledning

Når spillet først bliver kørt, vil brugeren først blive mødt af en menu. Brugeren skal vælge et menupunkt, ved at taste enten 1 eller 2, og trykke enter. Denne første menu er en start-menu inden spillet, som giver brugeren to forskellige valgmuligheder.
Den ene er hvor du går videre til næste menu, som hjælper dig med at indlæse et spil fra en fil, ved at taste 1 ind og trykke enter. Den anden er at forlade programmet ved at svare 2.
Hvis valgmuligheden 1 bliver valgt, vil brugeren komme til en menu hvor de kan se både den nuværende mappens indhold og stien til mappen hvor programmet er kørt fra. Brugeren kan nu indtaste et filnavn hvorefter programmet vil åbne filen, og komme ind i selve spillet. Herefter kan brugeren spille spillet ved at indtaste kommandoer.
Spillet vil blive præsenteret i venstre hjørne, hvor man ville kunne se selve sudoku værdierne som den indlæste fil indeholder. En af brugerens muligheder er, at skrive input til spillet, dvs. hvilken værdi der skal sættes i en specifik række og søjle. Dette gøres ved at skrive rækkefølgen række, søjle, værdi som følgende eksempel illustrerer: `1,2,3` eller `123` eller `(1,2,3)`. Brugeren kan selv vælge hvilket input han ønsker. Dernæst vil Sudoku spillet blive opdateret med det nye input, og man kan herefter spille videre på spillet så meget man lyster.

Valgmulighederne i menuen, som findes når spillet går igang hedder følgende:

- `save`: Denne valgmulighed gør at brugeren kan gemme spillet som en fil på computeren, efter man har skrevet "save"skal man indtaste det ønskede navn for filen.
  Filnavnet må ikke indeholde et af følgende tegn: / * ? < > | : . '
- `hint`: Du kan med denne valgmulighed få spillet til at give dig et gyldigt hint til næste træk. Dette hint vil blive præsenteret lige under selve spillefladen når kommandoen er indtastet.
- `solve`: Denne valgmulighed kan brugeren anvende hvis det ønskes at få Sudokuen til at løse sig af sig selv.
- `menu`: Dette input returnerer brugeren til den foregående menu. De vil få præsenteret den menu de mødte, da de først åbnede programmet.
- `exit`: Dette input forlader spillet og lukker programmet i terminalen.

Appendix B: Programtekst

Listing 1: Sudoku.fs indhold

```
///int -> 'a list list -> int list //returner raekken med index r fra en
    liste af
///lister f. eks en Sudoku
///<param name="r">Index fra 0-8</param>
///<param name="s">En 'a list list f. eks en Sudoku</param>
///<returns>Liste med 9 elementer svarende til raekken</returns>
let getRow (r:int) (s:'a list list) = s.[r]

///int -> Sudoku -> int list //returner kolonnen med index c fra en
   liste af
///lister f. eks en Sudoku
///<param name="c">Index fra 0-8</param>
///<param name="s">En 'a list list f. eks en Sudoku</param>
///<returns>Liste med 9 elementer svarende til kolonnen</returns>
let rec getColumn (c:int) (s:'a list list) =
  match s with
  | [] -> []
  | l0::ls -> l0.[c] :: getColumn c ls
```

```
///int -> Sudoku -> int list //returner kvadranten med index q fra en
    liste af
///lister f. eks en Sudoku
///<param name="q">Index fra 0-8</param>
///<param name="s">En 'a list list f. eks en Sudoku</param>
///<returns>Liste med 9 elementer svarende til kvadranten</returns>
let getQuadrant (q:int) (s:'a list list) =
  let r = (q/3)*3       //the upper row in the quadrant
  let c = (q%3)*3       //the left column of the quadrant

  s.[r].[c] :: s.[r].[c+1] :: s.[r].[c+2] ::
  s.[r+1].[c] :: s.[r+1].[c+1] :: s.[r+1].[c+2] ::
  s.[r+2].[c] :: s.[r+2].[c+1] :: s.[r+2].[c+2] :: []

///int*int*int -> Sudoku -> bool //tjekker at (r,c,v) er et lovligt
    input for denne Sudoku
///<param name="(r,c,v)">Inputtet med (raekke, kolonne og vaerdi) i
    sudokuen</param>
///<param name="s">Sudokuen som int list list</param>
///<returns>True hvis triplen er lovlig ellers false</returns>
let checkRsv (r:int, c:int, v:int) (s:Sudoku) =
  if r>=1 && r<=9 && c>=1 && c<=9 && v>=1 && v<=9 && s.[(r-1)].[(c-1)] =
      0 then
    let row = getRow (r-1) s
    let column = getColumn (c-1) s
    let q = (r-1)/3*3 + (c-1)/3
    let quad = getQuadrant q s
    not (List.exists (fun e -> e = v) row) &&
      not (List.exists (fun e -> e = v) column) &&
      not (List.exists (fun e -> e = v) quad)
  else
    false

/// int -> 'a -> 'a list -> 'a list //replace the element at index
/// Erstat et index i en liste list med vaerdien e. Listen er en liste
    af generiske typer
/// saa den kan baade bruges til at erstatte en int, en liste i en liste
     eller en hvilken som
/// helst anden type
/// <param name="index">Indeks i listen der skal erstattes</param>
/// <param name="e">Elementer der skal saettes ind paa indeksets plads</
    param>
/// <param name="list">Listen der skal have erstattet et element</param>
let replaceAt index e list =
  let rec replace index e list acc =
    match list with
    | [] -> []
    | x0::xs -> if (index = acc) then
                  e::replace index e xs (acc+1)
                else
                  x0::replace index e xs (acc+1)

  replace index e list 0

/// Indsaetter vaerdien v i Sudokuen og returnerer en liste af lister
    med en ny Sudoku.
/// Funktionen tjekker ogsaa om det er en lovlig triple og kaster en
    exception
```

```
/// NotLegalInput hvis ikke vaerdien kan saettes ind.
/// Der bliver tjekket om vaerdien v er mellem 1 og 9 og at vaerdien
    ikke allerede
///staar i hverken raekker, kolonner eller kvadranter
/// c er en int med soejlenummeret (column) mellem 1 og 9 og v er
    vaerdien som skal
/// saettes ind i Sudokuen.
/// <param name="(r,c,v)">tripplen med (raekke, kolonne, vaerdi). r er
    en int med
/// raekken mellem 1 og 9.</param>
/// <param name="s">Sudokuen hvor vi vil indsaette vaerdien</param>
/// <returns>En ny Sudoku med den indsatte vaerdi</returns>
/// <exception cref="NotLegalInput">Thrown naar inputtet ikke er mellem
    1 og 9 eller
/// raekker, soejler eller kvadranter allerede indeholder vaerdien.</
    exception>
let insertRsv (r:int, c:int, v:int) (s:Sudoku) =
  if (checkRsv (r, c, v) s) then
    let row = getRow (r-1) s
    let newRow = replaceAt (c-1) v row
    replaceAt (r-1) newRow s

  else raise NotLegalInput

///int list -> bool
///Tjekker at en liste indeholder alle tal fra 1 til 9. Dette goeres ved
    at konvertere
///til et Set for kun at faa unikke vaerdier og herefter laegge alle
///tal sammen i saettet og tjekke at det giver 1+2+3+4+5+6+7+8+9 = 45
///Funktionen virker fordi insertRsv har tjekket at der ikke er ens tal
    i listen
///<param name="list">En liste med 9 elementer fra enten raekke, kolonne
    eller kvadrant</param>
///<returns>True hvis alle tal er fra 1-9 er i listen, ellers false</
    return>
let checkList list =
  let s = Set list
  Set.foldBack (fun x x0 -> x+x0) s 0 = (1+2+3+4+5+6+7+8+9)

//Bruger checkList til at tjekke alle raekker i en Sudoku
//<param name="s">Sudoku hvis raekker skal tjekkes</param>
//<returns>True hvis alle raekker indeholder 1 til 9</returns>
let checkRows (s:Sudoku) =
  List.foldBack (fun l0 b -> b && (checkList l0) ) s true  //check all
     rows

//Bruger checkList til at tjekke alle kolonner i en Sudoku
//<param name="s">Sudoku hvis kolonner skal tjekkes</param>
//<returns>True hvis alle kolonner indeholder 1 til 9</returns>
let checkColumns (s:Sudoku) =
  let rec cColumns = function
    | 0 -> (checkList (getColumn 0 s))
    | n -> (checkList (getColumn n s)) && (cColumns (n-1))
  cColumns 8

//Bruger checkList til at tjekke alle kvadranter i en Sudoku
//<param name="s">Sudoku hvis kvadranter skal tjekkes</param>
//<returns>True hvis alle kvadranter indeholder 1 til 9</returns>
```

```
let checkQuadrants (s:Sudoku) =
  let rec cQuadrants = function
    | 0 -> checkList (getQuadrant 0 s)
    | n -> checkList (getQuadrant n s) && cQuadrants (n-1)
  cQuadrants 8

/// Tjekker om Sudokuen har naaet en loesning. Dvs. at alle raekker,
    kolonner og kvadranter indeholder
/// alle tal fra 1 til 9
/// <param name="s">Sudokuen som skal tjekkes</param>
/// <returns>True hvis spillet er faerdigt ellers false</returns>
let isFinished s =
  checkRows s && checkColumns s && checkQuadrants s

//funktioner til hint systemet

///Set<int> -> int list -> int list
///Taeller hvor mange tal af hvert fra 1-9 som er i et saet. Returnerer
    en liste
///hvor index 0 er hvor mange 1'ere der er. Index 1 er hvor mange 2'ere
    der er osv.
///cnt er en start liste saa tidligere sets kan akkumuleres
///<param name="s">Et set som indeholder nogle af tallene mellem 1-9 (
    eller alle)
///<param name="cnt">Liste af int's med 9 elementer som er startvaerdien
    .
///<returns>Ny liste hvor hvert element er en taeller for hvert tal fra
    1 til 9</returns>
let countSet s cnt =
  Set.foldBack (fun x acc -> replaceAt (x-1) (acc.[x-1]+1) acc) s cnt

///int list -> int
///Koerer igennem listen returneret a countSetList og returnerer indexet
    hvor der staar vaerdien 1
///<param name="list">En liste med de 9 ints med de talte vaerdier fra
    countSet
///<returns>Index for den unikke vaerdi, dvs tallet er 1. (Dette index
    er 1-indekseret
///saa indekset svarer til et tal mellem 1 og 9)</returns>
let findUnique list =
  let rec fu i = function
  | [] -> -1
  | x0::xs -> if (x0=1) then i
              else fu (i+1) xs

  fu 1 list

///Set<int> list -> int list
///Taeller alle Sets i en liste af Set<int> som returneret af getRow,
    getColumn eller getQuadrant
///<param name="list">En liste af Sets. </param>
///<returns>Ny liste hvor hvert element er en taeller for hvert tal fra
    1 til 9</returns>
let countSetList list =
  //count all numbers in all sets and save it in a list
  let cnt = [0;0;0;0;0;0;0;0;0]

  let rec csl cnt = function
```

```
    | [] -> cnt
    | s0::ss -> csl (countSet s0 cnt) ss

    csl cnt list

  ///int -> Set<int> list -> int
  ///Hvilket index har den unikke vaerdi. Soeger igennem en liste af Sets
     og finder det
  ///index hvor det unikke tal fra 1-9 er
  ///<param name="v">Vaerdien fra 1-9 der soeges efter</param>
  ///<param name="list">En liste af sets f. eks returnet af getRow,
     getColumn eller getQuadrant
  ///<returns>Indekset for vaerdien v. F. eks hvis saettet er alle raekker
      faar i indekset for raekken</returns>
  let findIndexForValue (v:int) (list:Set<int> list) =
    let rec fifv i = function
    | [] -> -1
    | s0::ss -> if (Set.exists (fun x -> x = v) s0) then
                  i
                else fifv (i+1) ss

    fifv 1 list

  ///int*int -> Sudoku -> Set<int>
  ///Find mulige tal fra 1-9 for parret (raekke, kolonne) i Sodukoen s
  ///<param name="(r,c)">Raekke og kolonne for et felt i Sudokuen</param>
  ///<param name="s">Sudokuen som int list list
  ///<returns>Set<int> med tallene som er acceptable i dette felt eller et
      Set [-1]
  ///hvis det er umuligt at loese sudokuen<returns>
  let findPossible (r,c) (s:Sudoku) =
    if (s.[(r-1)].[(c-1)] = 0) then
      let q = (r-1)/3*3 + (c-1)/3
      let row = getRow (r-1) s
      let column = getColumn (c-1) s
      let quad = getQuadrant q s
      let uSet = Set.ofList (row @ column @ quad)
      if ((Set.count uSet) < 10) then   //10 for vaerdien 0 er ogsaa er
         med i saettet
        Set.difference (Set [1;2;3;4;5;6;7;8;9]) uSet
      else
        Set [-1]
    else
      Set []

  /// Tjekker om Sudokuen har naaet en absurditet. Dvs. at der er et sted
     hvor der ikke kan
  /// saettes et tal ind fordi der er en fejl i Sudokuen. Denne funktion
     tjekker kun
  /// om der er muligheder i de tomme felter. Hvis alle tal er udfyldt vil
      denne funktion
  /// returnere true
  /// <param name="s">Sudokuen som skal tjekkes</param>
  /// <returns>True sudokuen er korrekt og der stadig findes muligheder,
     og false hvis der er fejl
  /// og Sudokuen ikke kan loeses herfra<returns>
  let isCorrect (s:Sudoku) =
    let rec iw = function
```

```
    | (10,1) -> true
    | (m,10) -> iw (m+1,1)
    | (m,n) -> if ((s.[m-1].[n-1]) = 0) then
                 let ps = findPossible (m,n) s
                 let wrong = Set.contains -1 ps
                 if (wrong) then false
                 else iw (m,n+1)
               else iw (m,n+1)

  iw (1,1)

///Set<int> list list -> Set<int>* (int* int)
///Funktionen vil soege igennem alle saettene for hvert felt og finde
    det felt
///hvor der kun er 2 mulige tal. Dette giver det laveste antal
    soegninger da
///vi kun skal soege 2 loesninger igennem. Funktionen kan ogsaa finde 3
    mulige osv.
///<param name="cnt">Funktionen kan ogsaa soege efter tripler eller mere
     hvis det er noedvendigt
///til meget svaere sudokuer</param>
///<param name="list">Set<int> list list. Sudokuen men med et saet af
    mulige vaerdier
///for hvert felt</param>
///<returns>Et pair med et saet og endnu et par med raekke og kolonne nr
    .</returns>
let findPair cnt (list:Set<int> list list) =
  let rec fp cnt = function
  | (10, 1) -> ((Set []), (-1,-1))
  | (n, 9) -> let s0 = list.[n-1].[9-1]
              if (Set.count s0 = cnt) then (s0, (n,9))
              else fp cnt (n+1, 1)

  | (n, m) -> let s0 = list.[n-1].[m-1]
              if (Set.count s0 = cnt) then (s0, (n,m))
              else fp cnt (n, m+1)

  fp cnt (1,1)

///Sudoku -> Set<int> list list
///Funktion der bygger hele Set<int> list list af vaerdier der mangler
///<param name="s">Sudoku af int list list</param>
///<returns>Set<int> list list hvor hvert Set er de tal der kan staa i
    dette felt<returns>
let buildSets s =
  let rec bs row list = function
  | (9,9) -> let ps = findPossible (9,9) s
             list @ [row @ [ps]]

  | (n,9) -> let ps = findPossible (n,9) s
             bs [] (list @ [row @ [ps]]) (n+1,1)

  | (n,m) -> let ps = findPossible (n,m) s
             bs (row @ [ps]) list (n,m+1)

  bs [] [] (1,1)

///Looper igennem alle felter og ser hvad der mangler i hvert felt.
```

```
///Hvis der kun mangler et tal kan vi give det som hint. Funktionen
    samler alle saettene op i en liste af
///lister hvis der ikke nogen loesninger er, startes en alternativ hint
    funktion. Hvis denne alternative
///funktion ogsaa fejler kaldes en rekursiv soegefunktion som tester
    alle muligheder igennem for et felt
///der kun har 2 muligheder. Den loeser Sudokuen saa langt som den kan
    og hvis en rigtig loesning dukker op
///returnes et hint. Dette hint er desvaerre ikke noedvendigvis det
    naeste logiske skridt for en person der
///loeser Sudokuen men det ville vaere meget svaert at programme de ca.
    12 regler der findes for at loese
///de svaereste Sudokuer.
///<param name="s">En sudoku som int list list</param>
///<returns>En tripple med (raekke, kolonne, vaerdi)</returns>
///<exception cref="NoHint">Bliver kastet naar der ikke kan findes det
    naeste logiske
///korrekte traek</exception>
let rec getHint s =
  let suggestionSet = buildSets s

  let rec gh = function
  | (9,9) -> let ps = suggestionSet.[9-1].[9-1]
             if ((Set.count ps) = 1) then
               (9,9, (Set.toList ps).[0] )
             else
               (-1,-1,-1)

  | (n,9) -> let ps = suggestionSet.[n-1].[9-1]
             if ((Set.count ps) = 1) then
               (n,9, (Set.toList ps).[0] )
             else gh (n+1,1)

  | (n,m) -> let ps = suggestionSet.[n-1].[m-1]
             if ((Set.count ps) = 1) then
               (n,m, (Set.toList ps).[0] )
             else gh (n,m+1)

  let rec hintFromRow = function
  | 10 -> (-1,-1,-1)   //just return (-1,-,1-,1) if there is no hint
  | n -> let s0 = getRow (n-1) suggestionSet
         let cnt = countSetList s0
         let v = findUnique cnt
         if (v > 0) then
           (n, (findIndexForValue v s0), v)
         else
           hintFromRow (n+1)

  let rec hintFromColumn = function
  | 10 -> (-1,-1,-1)
  | n ->  let s0 = getColumn (n-1) suggestionSet
          let cnt = countSetList s0
          let v = findUnique cnt
          if (v > 0) then
            ((findIndexForValue v s0), n, v)
          else
            hintFromColumn (n+1)
```

```
    let rec hintFromQuad = function
    | 10 -> (-1,-1,-1)
    | n -> let s0 = getQuadrant (n-1) suggestionSet
           let cnt = countSetList s0
           let v = findUnique cnt
           if (v > 0) then
             let index = findIndexForValue v s0  //index i kvadranten
             let qm = (index-1)/3
             let qn = (index-1)%3
             let r = (n-1) / 3 * 3 + qm + 1
             let c = (n-1) % 3 * 3 + qn + 1
             (r, c, v)
           else
             hintFromQuad (n+1)

    let searchHint () =
      let rec sh = function
      | 10 -> (-1,-1,-1) //if we reach this all is doomed
      | n -> let (s0, (r, c)) = findPair n suggestionSet
             if ((r,c) = (-1,-1)) then
               sh (n+1)
             else
               let sl = Set.toList s0
               let rec loopSuggestions = function
               | [] -> (-1,-1,-1)
               | [x] -> (r,c,x)
               | x0::xs -> let news = insertRsv (r,c, x0) s
                           if (testSudoku news) then
                             (r,c, x0)
                           else loopSuggestions xs

               let su = loopSuggestions sl
               if (su = (-1,-1,-1)) then
                 sh (n+1)
               else su
      sh 2

    //evaluer hvert hint system et af gangen og returner hvis et hint er
       fundet
    let h = gh (1,1)
    if h = (-1,-1,-1) then
      let hr = hintFromRow 1
      if hr = (-1,-1,-1) then
        let hc = hintFromColumn 1
        if (hc = (-1,-1,-1)) then
          let hq = hintFromQuad 1
          if (hq = (-1,-1,-1)) then
            searchHint ()
          else hq
        else hc
      else hr
    else h

  ///Tager en Sudoku hvor der evt. er indsat en vaerdi som vi ikke er
     sikre paa.
  ///Der soeges nu igennem og ser om denne vaerdi giver en loesning eller
     en absurditet
```

```
///Denne funktion er gjort mutual recursive med getHint funktionen da de
    to funktioner
///har brug for at kalde hinanden. Dette goeres ved at skrive 'and'
    mellem funktionerne
///f. eks let funktion1 ... = ... and funktion2 ... = ...
///<param name="s">En Sudoku af int list list</param>
///<returns>True hvis der er en loesning, ellers false</returns>
and testSudoku (s:Sudoku) =
  let rec ts (su:Sudoku) =
    let (r,c,v) = try (getHint su)
                  with
                  | NoHint -> (-1,-1,-1)

    let news = try (insertRsv (r,c,v) su)
               with
               | NotLegalInput -> [[]]

    if (isFinished news) then
      true
    else if (v = -1) then
      false
    else ts news
  ts s

///Tager en Sudoku som input og returnerer en Sudoku som er udfyldt.
    Hvis ikke der
///findes en loesning kastes en NoSolution exception. Vaer opmaerksom
    paa at den returnerer
///een loesning. Hvis Sudokuen har flere loesninger vil denne funktion
    ikke returnere
///alle loesninger.
///<param name="s">En Sudoku som ikke er loest helt</param>
///<returns>En Sudoku som er udfyldt</returns>
///<exception cref="NoSolution">Bliver kastet hvis ikke der er en
    loesning til denne Sudoku
let solve (s:Sudoku) =
  let rec spil ss =
    let (r,c,v) = try (getHint ss)
                  with
                  | NoHint -> (-1,-1,-1)

    let newss = try (insertRsv (r,c,v) ss)
                with
                | NotLegalInput -> [[]]

    if (isFinished newss) then
      newss
    else if (v = -1) then
      raise NoSolution
    else spil newss

  spil s
```

Listing 2: Sudoku.fsi indhold

```
///Sudoku module
///Author: Andreas Broch
///Date: 3.-8. nov 2015
```

```
/// Compile with
/// <code>fsharpc -a Sudoku.fsi Sudoku.fs --doc:Sudoku.xml</code>

module Sudo
/// Typen Sudoku er en int list list. En Sudoku defineres ved en liste
    af lister
/// hvor et 0 indikerer et tomt felt. Derfor er den svaereste Sudoku
    nogensinde defineret
/// saaledes:
///let wh =   [[8;0;0;0;0;0;0;0;0];
///            [0;0;3;6;0;0;0;0;0];
///            [0;7;0;0;9;0;2;0;0];
///            [0;5;0;0;0;7;0;0;0];
///            [0;0;0;0;4;5;7;0;0];
///            [0;0;0;1;0;0;0;3;0];
///            [0;0;1;0;0;0;0;6;8];
///            [0;0;8;5;0;0;0;1;0];
///            [0;9;0;0;0;0;4;0;0]]
/// Ved gentagne gange at kalde getHint kan denne Sudoku loeses paa
    under et sekund
type Sudoku = int list list

/// Indsaetter vaerdien v i Sudokuen og returnerer en liste af lister
    med en ny Sudoku.
/// Funktionen tjekker ogsaa om det er en lovlig triple og kaster en
    exception
/// NotLegalInput hvis ikke vaerdien kan saettes ind.
/// Der bliver tjekket om vaerdien v er mellem 1 og 9 og at vaerdien
    ikke allerede staar i
/// hverken raekker, kolonner eller kvadranter
/// insertRsv kan kaldes med:
///<code>
///let newss = try (Sudo.insertRsv (r,c,v) ss)
///            with
///            | Sudo.NotLegalInput -> [[]]
///</code>
/// <param name="(r,c,v)">tripplen med (raekke, kolonne, vaerdi). r er
    en int med raekken mellem 1 og 9.</param>
/// <param name="s">Sudokuen hvor vi vil indsaette vaerdien</param>
/// <returns>En ny Sudoku med den indsatte vaerdi</returns>
/// <exception cref="NotLegalInput">Thrown naar inputtet ikke er mellem
    1 og 9 eller raekker, soejler eller kvadranter allerede indeholder
    vaerdien.</exception>
val insertRsv   : int*int*int -> Sudoku -> Sudoku

/// Tjekker om Sudokuen har naaet en loesning. Dvs. at alle raekker,
    kolonner og kvadranter indeholder
/// alle tal fra 1 til 9
/// <param name="s">Sudokuen som skal tjekkes</param>
/// <returns>True hvis spillet er faerdigt ellers false</returns>
val isFinished  : Sudoku -> bool

/// Tjekker om Sudokuen har naaet en absurditet. Dvs. at der er et sted
    hvor der ikke kan
/// saettes et tal ind fordi der er en fejl i Sudokuen
/// <param name="s">Sudokuen som skal tjekkes</param>
/// <returns>True hvis sudokuen er korrekt og der stadig findes
    muligheder, og
```

```
/// false hvis der er en fejl og Sudokuen ikke kan loeses herfra<returns
    >
val isCorrect  : Sudoku -> bool

///Looper igennem alle felter og ser hvad der mangler i hvert felt.
///Hvis der kun mangler et tal kan vi give det som hint. Hvis der ikke
    nogen loesninger er,
///startes en alternativ hint funktion. Hvis denne alternative
///funktion ogsaa fejler kaldes en rekursiv soegefunktion som tester
    alle muligheder igennem for et felt
///der kun har 2 muligheder, dernaest fortsaettes med felter der har 3
    muligheder om noedvendigt osv. Den
///loeser Sudokuen saa langt som den kan og hvis en rigtig loesning
    dukker op
///returnes et hint. Dette hint er desvaerre ikke noedvendigvis det
    naeste logiske skridt for en person der
///loeser Sudokuen men det ville vaere meget svaert at programmere de ca
    . 12 regler der findes for at loese
///de svaereste Sudokuer. Hvis denne funktion kaldes gentagne gange vil
    selv verdens svaereste Sudoku
///blive loest.
///getHint kan kaldes saadan:
///<code>
///let (r,c,v) = try (Sudo.getHint ss)
///              with
///              | Sudo.NoHint -> (-1,-1,-1)
///</code>
///<param name="s">En sudoku som int list list</param>
///<returns>En tripple med (raekke, kolonne, vaerdi)</returns>
///<exception cref="NoHint">Bliver kastet naar det naeste logiske
    korrekte traek ikke kan findes </exception>
val getHint     : Sudoku -> int*int*int

///Tager en Sudoku som input og returnerer en Sudoku som er udfyldt.
    Hvis ikke der
///findes en loesning kastes en NoSolution exception. Vaer opmaerksom
    paa at den returnerer
///een loesning. Hvis Sudokuen har flere loesninger vil denne funktion
    ikke returnere
///alle loesninger.
///<param name="s">En Sudoku som ikke er loest helt</param>
///<returns>En Sudoku som er udfyldt</returns>
///<exception cref="NoSolution">Bliver kastet hvis ikke der er en
    loesning til denne Sudoku
val solve       : Sudoku -> Sudoku

///<exception cref="NotLegalInput">Bliver kastet hvis en ulovlig triple
    bliver indtastet,
///som ikke kan indsaettes paa denne plads i Sudokuen</exception>
exception NotLegalInput

///<exception cref="NoHint">Bliver kastet naar getHint ikke kan finde en
     korrekt
///loesning paa naeste traek</exception>
exception NoHint

///<exception cref="NoSolution">Bliver kastet naar solve ikke kan finde
    en korrekt
```

```
///loesning paa hele Sudokuen</exception>
exception NoSolution
```

Listing 3: FileHandling.fs indhold

```
/// Denne fil indeholder implementationen for FileHandling modulet, som
    er ansvarlig for at haandtere filernes indhold. Saaledes er der
    implementeret en raekke funktioner, der som udgangspunkt skal
    repraesentere et sammenspil mellem brugeren og indholdet af Sudoku
    spillet, der lagres i tekstfiler paa computeren.
///
/// Author: Matthias Brix
/// Date: 11-11-2015
module FileHandling
exception WrongContent

/// filename -> bool
/// <summary>Tjekker om et filnavn som brugeren har tastet ind er
    gyldigt ved at tjekke om det eksisterer i den aktuelle mappe.</
    summary>
/// <param name="filename">Et filnavn, der skal vaere en .txt filtype</
    param>
/// <returns>Returnerer en bool. True hvis filnavnet er gyldigt og
    findes i mappen, og hvis ikke, returneres false</returns>
let checkFile filename =
  if not (System.IO.File.Exists filename) then
    false
  else
    true

/// string -> bool
/// <summary>Kaldes for at tjekke om et af filnavnene der gemmes med,
    ikke indeholder en af den naevnte tegn og at det ikke overgaar 255
    tegn</summary>
/// <param name="filename">String, filnavn</param>
/// <returns>Returnerer true, hvis navnet indeholder en af tegnene eller
     antal af tegn over 255, eller hvis antal af tegn er under 3, ingen
    mellemrum tilladt </returns>
let checkFileName filename =
  (not(String.exists (fun c -> (c='/') || (c='\\') || (c='*') || (c='?')
       || (c='<') || (c='>') || (c='|') || (c=':') || (c='.')) filename)
     && (String.length (filename.Trim()) >= 3) && (String.length (
      filename.Trim()) < 255))

/// unit -> string
/// <summary>Nedenstaaende funktion printer alle .txt filer ud der
    ligger i den aabnede mappe. Objektet .GetFiles tager path,
    searchPattern som argumenter, som saa tager alle filer der ender med
    .txt og gemmer disse i et array i funktionen filterFiles. Herefter
    gemmes en string, som bliver concatenated vha et komma.</summary>
/// <param name="">unit</param>
/// <returns>Returnerer en streng, som viser alle .txt filer i en mappe
    </returns>
let showAllTxtFiles () =
  let filterFiles = System.IO.Directory.GetFiles(".", "*txt")
  let allTxtFiles = String.concat "␣,␣" (Array.toList filterFiles)
  allTxtFiles
```

```
/// unit -> string
/// <summary>Funktionen printer den aabenede mappe ud i som en string</
    summary>
/// <param name="">unit</param>
/// <returns>Returnerer en streng, som er den aktuelle mappe</returns>
let getCurrentPath () = System.IO.Directory.GetCurrentDirectory()

/// string * int -> int list
/// <summary>Laver en string om til en liste. Dette goeres ved at den
    laegger hvert enkelte element af strengen paa en liste, som og kalder
     derefter funktionen rekursivt, indtil y, som er indeks parametren
    for stringen, er naaet til at vaere 9, som anses til at vaere
    maksimum for enkelt liste til et sudoku spil. En liste skal saaledes
    repraesentere en region i spillet</summary>
/// <param name="(x,y)">x : string * y : int</param>
/// <returns>Returnerer en int list</returns>
let rec convertList = function
  | (x,y) when (y > (String.length x)-1) -> []
  | (x,y) -> (int (x.[y])-48)::(convertList (x,y+1))

/// Tjek om tallene er mellem 0 og 9
let checkInput result = (List.forall (fun x -> x <= 9 && x >= 0) result)

/// fileContent : string -> int list list
/// <summary>Laver en string om til en liste af lister. Den begynder med
     at erstatte * med 0, \n med "", saaledes at strengen er klar til at
    blive gemt i listerne med hele tal. Herefter koeres lokal funktionen
    Convert rekursivt, og ved hvert kald haenges resultatet af
    ConvertList paa, som returnerer en liste med hele tal, som bestaar af
     9 elementer. Hvis dette saa er koert igennem hele strengen,
    returneres et helt Sudoku spil i form af en liste med 9 lister, hvor
    hver har 9 elementer.</summary>
/// <param name="fileContent">Indholdet af en fil i en string</param>
/// <returns>Liste af lister med hele tal, som svarer til et samlet
    Sudoku spil, med i alt 81 elementer</returns>
let textToList (fileContent : string) =
  let newcont = fileContent.Replace("*", "0")
  let newcont2 = newcont.Replace("\\n", "") // Works too, prints the
     input as one long coherent string without \n alternative: .Trim()
  let rec convert (strInput : string) lister (start : int) =
    if ((List.length lister) = 9) then
      lister
    else
      let result = (convertList (strInput.Substring(start,9),0))
      if (checkInput result) then
        (convert strInput (lister@[result]) (start+9))
      else
        raise WrongContent
  (convert newcont2 [] 0)

/// streamreader : System.IO.StreamReader -> int list list
/// <summary>Laeser vha. en Streamreader indholdet af en .txt fil, og
    gemmer dens indhold som en streng. Hvis dette er gjort, lukkes
    Streamreader, og funktionen TextToList bliver kaldt med indholdet af
    filen</summary>
/// <param name="streamreader">Indholdet af en fil i en string</param>
/// <returns>Liste af lister med hele tal, som svarer til et samlet
    Sudoku spil, med i alt 81 elementer</returns>
```

```
let rec readFileContent (streamreader : System.IO.StreamReader) =
  let fileContent = (streamreader.ReadToEnd())
  if not (streamreader.EndOfStream) then
    readFileContent streamreader
  else
    streamreader.Close()
    textToList fileContent

/// filename : string -> int list list
/// <summary>aabner en eksisterende tekstfile og opretter et
    Streamreader objekt. Kalder derefter ReadFileContent funktionen med
    objektet, som goer det muligt at laese indholdet af filen</summary>
/// <param name="filename">Navnet af en .txt fil</param>
/// <returns>Returnerer en int list list, som bliver fundet frem til i
    de forrige funktioner</returns>
let readFile filename =
  let streamreader = System.IO.File.OpenText filename
  readFileContent streamreader

/// filename : string -> int list list
/// <summary>Tager som argument et filnavn som string, og kalder
    derefter de noedvendige funktioner. Derefter returneres listen af
    listerne</summary>
/// <param name="filename">Navnet af en .txt fil</param>
/// <returns>Et Sudoku spil</returns>
let loadSudoku (filename : string) = (readFile filename)

/// lister : int list list -> string
/// <summary>Omdanner listens indhold om til en streng, som bliver en
    tekstfil</summary>
/// <param name="lister">Et Sudoku spil i en liste af lister med hele
    tal</param>
/// <returns>Et Sudoku spil i en streng</returns>
let listToText lister =
  let listEdit = List.map (fun x -> (List.map (fun y -> string y) x))
      lister // Denne funktion laver hvert element i listerne i listen om
       til strings
  let listEdit2 = List.map (fun x -> (List.append x ["\\n"])) listEdit
      // Appender \n til hver del liste
  let listEdit3 = List.concat listEdit2 // Saetter alle listerne i
      listen sammen og returnerer blot en liste
  let listEdit4 = List.foldBack (fun x acc -> x+acc) listEdit3 "" //
      Akkumulerer hvert element (x) paa startvaerdien "" (acc) og koerer
      dette paa listEdit3
  let listEdit5 = listEdit4.Replace("0","*") // laver alle 0 om til *
      saaledes at man returnerer til den oprindelige struktur af string
      inholdet
  listEdit5

/// filename : string -> userInput : int list list -> unit
/// <summary>Take the input from ListToText and print it out on the file
    . check that it saves the filename of the first file that has been
    read, and the passes that name into that function</summary>
/// <param name="filename">Navnet af en .txt fil</param>
/// <param name="userInput">Det aendrede Sudoku spil i en liste af
    lister</param>
/// <returns>unit ()</returns>
let saveFile filename userInput =
```

```
    let toSave = listToText userInput // Indholdet der skal gemmes i filen
        gemmes i funktionen
    let outputStream = System.IO.File.CreateText filename // Opretter et
       Streamwriter objekt til filnavnet
    outputStream.Write toSave // Indholdet overskriver outputtet til filen
        , siden dens standard indstilling er at overskrive og ikke tilfoeje
    outputStream.Close() // Lukker stroemmen

/// filename : string -> lister : int list list -> unit
/// <summary>Gemmer brugerens sidste aendring af Sudoku spillet i en fil
    </summary>
/// <param name="filename">Navnet af en .txt fil, som skal kommunikeres
    med</param>
/// <param name="lister">Et Sudoku spil i en liste af lister med hele
    tal</param>
/// <returns>unit ()</returns>
let saveSudoku (filename : string) (lister : int list list) = (saveFile
    filename lister)
```

Listing 4: FileHandling.fsi indhold

```
/// Denne fil indeholder signaturen for FileHandling modulet, som er
    ansvarlig for at haandtere filernes indhold. I forbindelse med det er
     der impementeret funktioner til baade inputtet af brugeren og
    outputtet til filerne.
///
/// Author: Matthias Brix
/// Date: 11-11-2015
module FileHandling

/// <summary>Tjekker om en et filnavn som brugeren har tastet ind er
    gyldigt ved at tjekke om det eksisterer i den aktuelle mappe.</
    summary>
/// <param name="filename">Et filnavn, der skal vaere en .txt filtype</
    param>
/// <returns>Returnerer en bool. True hvis filnavnet er gyldigt og
    findes i mappen, og hvis ikke, returneres false</returns>
val checkFile : string -> bool

/// Test eksempel, som boer returnere true: printfn "%A" (FileHandling.
    checkFileName "tes:")
/// <summary>Kaldes for at tjekke om et af filnavnene der gemmes med,
    ikke indeholder et tegn som Windows, Mac (ingen . til start) og Linux
     (ingen / og \0 - men det er daekket) ikke tillader:
/// / \ * ? < > | : .
/// Derudover maa laengden af strengen heller ikke vaere stoerre end 255
     tegn, og mindst 3 tegn, men ingen mellemrum
/// </summary>
/// <param name="filename">String, filnavn</param>
/// <returns>Returnerer true, hvis navnet indeholder en af tegenene</
    returns>
val checkFileName : string -> bool

/// <summary>Nedenstaaende funktion printer alle .txt filer ud der
    ligger i den aabnede mappe</summary>
/// <param name="">unit</param>
/// <returns>Returnerer en streng, som viser alle .txt filer i en mappe
    </returns>
```

```
val showAllTxtFiles : unit -> string

/// <summary>Funktionen printer den aabenede mappe ud i som en string</
    summary>
/// <param name="">unit</param>
/// <returns>Returnerer en streng, som er den aktuelle mappe</returns>
/// Samlet test af showAllTxtFiles() og getCurrentPath kunne eksempelvis
     vaere: printfn "\nDen aabnede mappe indeholder foelgende .txt filer
    i mappen\n%s: \n\n%A" (getCurrentPath ()) (showAllTxtFiles ())
val getCurrentPath : unit -> string

/// <summary>Funktionen tager imod et filnavn som argument, hvorefter
    den laeser filens indhold. Dernaest laves indholdet, som er en string
    , om til int list list, hvorefter Sudoku spillet er klar til at blive
     modificeret af brugeren</summary>
/// <param name="filename">Et filnavn, der skal vaere en .txt filtype</
    param>
/// <returns>Et Sudoku spil i form af lister i en list, hvor alle
    elementer er hele tal. Listen bestaar af 9 lister, med hver 9
    elementer, som giver i alt 81 elementer</returns>
val loadSudoku : string -> int list list

/// <summary>Funktionen gemmer et Sudoku spil som string i en tekstfil.
    Dette sker ved at den tager i mod en liste af lister med hele tal,
    som den laves om til en streng. Denne streng gemmes saa tekstfil,
    hvor dataene er kommet fra</summary>
/// <param name="filename">Et filnavn, der skal vaere en .txt filtype</
    param>
/// <param name="lists">En liste af lister med hele tal</param>
/// <returns>Gemmer "lists" som en string type i "filename", som lagres
    i den aabnede mappe</returns>
val saveSudoku : string -> int list list -> unit
```

Listing 5: Run.fsx indhold

```
#r "FileHandling.dll" //Kaldes med FileHandling.
#r "Sudoku.dll"       //Kaldes med Sudo.
#r "Gameboard.dll"    //Kaldes med Gameboard.

let rec playGame sudoku hint =
  if (Sudo.isCorrect sudoku) && not (Sudo.isFinished sudoku) then
    System.Console.Clear()
    Gameboard.output (sudoku)
    if hint then
      printfn "Hint, proev: %A\n" (Sudo.getHint sudoku)
    System.Console.Write("Skriv en triple med raekke, soejle, vaerdi for
        at input\n f.eks: (1,3,1) eller 131\n \nKommandoer:\n
    save - Gem spillet i en .txt fil\n
    hint - Faa et raad til at loese spillet\n
    solve - Loes spillet\n
    menu - Tilbage til menu\n
    exit - Forlad programmet\n")
    // Command in placeHolder patternmatching
    printf "\nSkriv input :> "
    let placeHolder = System.Console.ReadLine()
    match placeHolder with
    // Listen af strings der passer ind i patternmatching, hvor cmd er
        kommando indputtet, RSV
```

```
      // save kalder saveSudoku, hunt kalder spillet med en true bool
         vaerdi som genererer vores hint til loesning af spillet
      // solve kalder Sudo.solve paa spillet som udfoerer funktionen.
      // menu udfoerer en simpel printf som goer at spillet terminerer og
         dermed returnerer til menuen
      // exit kalder den indbyggede funktion exit 1, som terminerer
         programmet
        | "save" -> let rec saveFn () =
                      printf "Indtast␣filnavn:␣"
                      let SaveName = System.Console.ReadLine()
                      if (FileHandling.checkFileName SaveName) then
                        FileHandling.saveSudoku (SaveName + ".txt") (
                            sudoku)
                        playGame sudoku false
                      else
                        printfn "\nUgyldigt␣filnavn!␣Navnet␣skal␣vaere␣
                            mellem␣3␣og␣255␣tegn,␣ingen␣mellemrum␣medregnet
                            ,␣eller␣indeholde␣foelgende␣tegn:\n
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣/␣\␣*␣?␣<␣>␣|␣:␣.␣'\n"
                        saveFn ()
                    saveFn ()
        | "hint" -> playGame sudoku true
        | "solve" -> try
                       let s = (Sudo.solve (sudoku))
                       playGame s false
                     with
                       | _ -> printfn "Der␣findes␣ikke␣en␣loesning␣til␣
                           spillet!"
                              let a = System.Console.ReadLine()
                              printfn ""
        | "menu" -> printfn ""
        | "exit" -> exit 1
        | cmd ->  let newSudoku = try
                                    let PT = (Gameboard.parseTripple (cmd)
                                        )
                                    Sudo.insertRsv PT sudoku
                                  with
                                    | _ -> [[-1]]

                  if newSudoku = [[-1]] then
                    playGame sudoku false
                  else
                    playGame newSudoku false

    else if (Sudo.isFinished sudoku) then
        System.Console.Clear()
        Gameboard.output sudoku
        printfn "Tillykke,␣du␣har␣loest␣Sudokuen␣perfekt!␣\nTryk␣enter␣for
            ␣at␣komme␣tilbage␣til␣menuen"
        let waitforuser = System.Console.ReadLine()
        printfn ""
    else
      printfn "Desvaerre,␣sudokuen␣kan␣ikke␣loeses␣herfra!␣\nProev␣igen!␣
          Tryk␣enter␣for␣at␣komme␣tilbage␣til␣menuen"
      let waitforuser = System.Console.ReadLine()
      printfn ""

  //Menu funktioner til brugeren inden spillet gaar igang
```

```
let rec initiateGame () =
  printf "\n----- Indlaes et Sudoku spil fra en .txt fil -----\n\nVenligst
     indtast et .txt filnavn (men uden .txt): "
  let filename = (System.Console.ReadLine() + ".txt")
  if (FileHandling.checkFile filename) then
    // Store gameboard in gamepH (gamePlaceHolder)
    let gamepH = FileHandling.loadSudoku filename
    // Kalder playGame som tager filnavnet og saetter igang med spillet,
        med en false vaerdi
    // Som er hvis spillet skal benytte sig af hints
    playGame gamepH false
  else
      // Ellers returner til denne menu
      initiateGame ()

let getFiles () =
  if (FileHandling.showAllTxtFiles () = "") then
    printfn "\nIngen tekstfiler i mappen. Tilfoej en fil med et Sudoku
       spil og proev igen"
  else
    printfn "\nDen aabnede mappe indeholder foelgende .txt filer i
       mappen\n%s: \n\n%A" (FileHandling.getCurrentPath ()) (
       FileHandling.showAllTxtFiles ())

//Den foerste menu der bliver printet til brugeren
let rec showMainMenu () =
  System.Console.Clear()
  printfn "----- Velkommen til Sudoku ----\n"
  printfn "1 - Spil et Sudoku spil"
  printfn "2 - Forlad programmet\n"
  printfn "Venligst vaelg en valgmulighederne"
  let rec mainMenuResponse () =
    System.Console.Write("\nVaelg en mulighed: ")
    let choice = (System.Console.ReadLine())
    match choice with
      | "1" -> getFiles ()
               initiateGame ()
      | "2" -> System.Console.Clear()
               exit 1
      | _ -> printfn "Not a valid choice"
             mainMenuResponse ()
  mainMenuResponse()
  showMainMenu()

//Kalder menuen foerste gang naar man aabner den kompilede fil
showMainMenu ()
// "\nDen aabnede mappe indeholder foelgende .txt filer i mappen\n: \n\n
   "
```

Listing 6: Gameboard.fs indhold

```
/// Author: Jannick Fritz Drews
/// Date: 19-11-2015
/// Compile with
/// <code>fsharpc -a Gameboard.fsi Gameboard.fs --doc:Gameboard.xml</
    code>
module Gameboard

/// Denne funktion skal hjaelpe med at faa indputtet fra brugeren som
    vil indsaette en vaerdi, til at goere det
/// mere overskueligt og har en nemmere tilgang.
/// <summary> Funktionen tager et indput string fra brugeren som bliver
    konverteret til en triple,
/// funktionen er skrevet saaledes at den baade kan tage indput 123
    eller (1,2,3). </summary>
/// <param name="string">
/// Indeholder en string som bliver konverteret til et triple med 3 ints
/// </param>
/// <returns>
/// En triple med det indtastede input
/// </returns>
let parseTripple (tripple: string) =
  let rec pt n =
    if (n = (String.length tripple)) then []
    else
         let c = tripple.Chars(n)
         if c = '(' then (pt (n+1))
         else if c >= '0' && c<='9' then (((int c)-48)::pt (n+1))
         else if c = ',' then (pt (n+1))
         else if c = ')' then (pt (n+1))
         else failwith "Not a triple"

  let numberList = pt 0
  (numberList.[0], numberList.[1], numberList.[2])

/// Funktionen kan visualiseres paa den maade at inden sudokuen bliver
    printet til brugergraensefladen gaar
/// den igennem output funktionen som danner spillefladen saa den
    appelerer til brugeren og saa spillet
/// bliver mere medgoerligt visuelt.
/// <summary> Tager sodukoen og laver et "gitter" eller en spilleflade
    saa sudokuen bliver mere overskuelig
/// </summary>
/// <param name="s">
/// Indeholder sudokuen som skal printes
/// </param>
/// <returns>
/// int list list bliver konverteret til en unit som printes
/// </returns>
let output (s:int list list) =
  printfn "******Sudoku Spillet*****"
  let rec out = function
  | (8, 9) -> printfn "|\n+-------+-------+-------+"
  | (m, 9) -> printfn "|"
             out (m+1,0)
  | (m, n) -> if (m % 3 = 0 && n=0) then
                printfn "+-------+-------+-------+"
             if (n % 3 = 0) then
```

```
                printf "|␣"
              if (s.[m].[n] = 0) then
                printf "␣␣"
              else
                printf "%d␣" (s.[m].[n])
              out (m, n+1)
  out (0,0)
```

Listing 7: Gameboard.fsi indhold

```
/// View module
/// Signaturfilen til Gameboard modulet
/// Author: Jannick Fritz Drews
/// Date: 19-11-2015
/// Compile with
/// <code>fsharpc -a Gameboard.fsi Gameboard.fs --doc:Gameboard.xml</
    code>
module Gameboard

/// <summary> Funktionen tager et indput string fra brugeren som bliver
    konverteret til en triple,
/// funktionen er skrevet saaledes at den baade kan tage indput 123
    eller (1,2,3). </summary>
/// <code>
/// let z = System.Console.ReadLine()
/// match z with
///   | indput -> let s = (Gameboard.parseTripple (indput))
///               printfn "%A" s
/// </code>
/// <param name="string">
/// Indeholder en string som bliver konverteret til et triple med 3 ints
/// </param>
/// <returns>
/// En triple med det indtastede input
/// </returns>
val parseTripple : string -> int * int * int

/// <summary> Tager sodukoen og laver et "gitter" eller en spilleflade
    saa sudokuen bliver mere overskuelig
/// </summary>
/// <param name="s">
/// Indeholder sudokuen som skal printes
/// </param>
/// <returns>
/// int list list bliver konverteret til en unit som printes
/// </returns>
val output : int list list -> unit
```