Danmarks Tekniske Universitet

Bachelor Projekt

Funktionel Modellering af Matematiske Systemer i F#

Jonas Dahl Larsen (s205829)

10. maj 2024

Indhold

1 Introduktion

Dette projekt fokuserer på funktionel modellering af matematiske systemer ved brug af programmeringssproget F#. I en tid, hvor programmeringssprog som Python dominerer i tekniske og videnskabelige miljøer, undersøger dette projekt potentialet og fordelene ved funktionel programmering i matematiske sammenhænge. I 2023 valgte Danmarks Tekniske Universitet at anvende Python som et hjælpeværktøj i deres grundlæggende matematikkursus "01001 Matematik 1a (Polyteknisk grundlag)". Imidlertid åbner funktionel programmering op for et andet perspektiv og metoder, som kan berige og muligvis forbedre forståelsen af matematiske koncepter hos studerende.

Projektet har til formål at demonstrere, hvordan funktionel programmering, specifikt gennem F#, kan anvendes til at opbygge og manipulere matematiske udtryk og systemer. Ved at introducere læserne til grundlæggende såvel som avancerede funktioner og teknikker i F#, vil rapporten guide dem gennem opbygningen af funktionelle programmer, der kan løse matematiske problemer.

Rapporten vil først og fremmest dykke ned i konstruktionen af et specifikt modul for håndtering af symbolske matematiske udtryk og matrix manipulering, og deres anvendelser i forskellige matematiske kontekster. Projektets struktur og metodologi har til formål at give læseren en dybdegående forståelse af, hvordan funktionel programmering kan benyttes strategisk i matematiske discipliner, og hvordan det adskiller sig fra mere traditionelle imperative programmeringstilgange.

Gennem en systematisk tilgang til design og implementering af matematiske moduler vil rapporten udforske, hvordan matematiske og logiske principper kan integreres direkte i softwareudvikling gennem funktionel programmering. Dette vil ikke kun fremme en bedre forståelse af teoretiske koncepter gennem praktisk anvendelse, men også demonstrere F#'s kapacitet og effektivitet i behandlingen af matematiske egenskaber.

2 Fundamentale koncepter

2.1 Introduktion til Funktions Programmering

Det forventes, at læseren har kendskab til programmering. Der gives derfor kun en kort beskrivelse af syntaks og notation, så læsere, der ikke er bekendt med F#, kan forstå de eksempler, der løbende vil forekomme i rapporten.

$$f(n) = \begin{cases} 1 & n = 0 \\ n \cdot f(n-1) & n > 0 \\ \text{undefined} & n < 0 \end{cases} \tag{1}$$

Vi begynder derfor med at betragte funktionen for fakultet Ligning 1. Et eksempel på en implementering i F# er givet i Listing 1, som kan sammenlignes med Python-koden i Listing 2, da Python og pseudokode er næsten det samme.

```
1 // Fakultet i F#
2 let rec factorial n =
3     match n with
4     | 0             -> 1
5     | x when x > 0  -> x * factorial (x - 1)
6     | _             -> failwith "Negative argument"
```

Listing 1: Eksempel på Fakultet i F#

```
1 # Fakultet i Python
2 def factorial(n):
3     if n == 0:
4         return 1
5     elif n > 0:
6         return n * factorial(n - 1)
7     else:
8         raise ValueError("Negative argument")
```

Listing 2: Eksempel på Fakultet i Python

I F# anvendes let til at definere en ny variabel eller, i dette tilfælde, en funktion kaldet factorial. Næste nøgleord er rec, hvilket indikerer, at funktionen er rekursiv. Funktionen tager et inputargument n, og i linje 3 starter et match-udtryk. Her er n vores udtryk, og efter with begynder en række mønstre, som udtrykket forsøger at matche på, separeret med '|'. Resultatet af funktionen vil være den kode, der eksekveres efter '→', på den linje, hvor mønsteret er genkendt.

I F#, er det som udgangspunkt ikke nødvendigt at anvende parenteser som i andre programmeringssprog. Derfor vil de kun blive anvendt, hvor det er nødvendigt gennem rapporten, typisk i sammenhænge med kædning af funktioner. For at undgå brugen af parenteser kan man i F# benytte pipe-operatorerne, $|>$ og $<|$, som fører resultatet fra en udledning direkte ind i den næste funktion. Nedenstående eksempel viser tre ækvivalente udtryk, der demonstrerer anvendelsen af disse operatorer.

```
> factorial (factorial 3);;
val it: int = 720

> factorial <| factorial 3;;
val it: int = 720

> factorial 3 |> factorial;;
val it: int = 720
```

Listing 3: Eksempel på anvendelse af pipe-operatorer i F#

2.1.1 Typer

I F#, i modsætning til Python, er typer tildelt ved kompileringstidspunktet, ikke under kørsel. Alle udtryk, inklusiv funktioner, har en defineret type. Typen for funktionen i Listing 1 er $int \rightarrow int$. Det betyder, at det ikke er muligt at kalde funktionen med et argument, der ikke er af typen int. Typen for funktionen beskrives som $Factorial : int \rightarrow int$. Vi kan derfor formulere følgende omkring typer[1]:

$$f : T_1 \rightarrow T_2$$
$$f(e) : T_2 \iff e : T_1$$

Hvis en funktion kaldes med et argument, der ikke matcher funktionens type, genereres en fejlmeddelelse. Derudover kan en type også bestå af en tuple af typer:

$$f : T_1 * T_2 * .. * T_n \rightarrow T_{n+1}$$
$$f(e_1, e_2, .., e_n) : T_{n+1} \iff e_1 : T_1 \wedge e_2 : T_2 \wedge .. \wedge e_n : T_n$$

En tuple, der kun består af to typer, kaldes et par. Givet en funktion $g : T_1 \rightarrow T_2 \rightarrow T_3$, betyder dette, at den tager et udtryk af typen T_1, som giver en funktion af typen $T_2 \rightarrow T_3$, hvor evalueringen af funktionen resulterer i T_3.

2.2 Signatur filer og implementerings filer

En standard F# fil er lavet med .fs extension, denne fil indeholder alt den kode som er nødigt for at kunne køre programmet. En implementerings fil kan have en signatur fil med .fsi extension, denne fil indeholder en beskrivelse af de typer og funktioner i implementerings filen som er tilgængelige for andre filer. En signatur fil kan derfor bruges som et blueprint for andre der ønsker at anvende eller replicere implementerings filen. I andre programmerings sprog vil man anse funktionerne i signatur filen som værende "public"og de funktioner der ikke er i signatur filen, men er i implementerings filen som værende "private".

2.3 Overloading af operatorer

I F# er det muligt at overskrive standardoperatorer, så de kan anvendes på egne typer. Denne teknik vil blive benyttet igennem rapporten til at definere matematiske operationer for de typer, vi udvikler.

[1] *Functional Programming Using F#*, s. 14.

2.4 Property Based Testing

Property Based Test (PBT) er en teknik til at teste korrekthed af egenskaber som man ved altid skal være opfyldt. Ved PBT genereres en række tilfældige input til en funktion, hvorefter det kontrolleres, om en given egenskab holder.

På DTU lærer de matematiske studerende først om logik, hvor det introduceres, at en udsagnslogisk formel er gyldig (en tautologi), hvis den altid er sand. Der findes mange teknikker til at påvise gyldigheden af en udsagnslogisk formel. I de indledende matematiske kurser på DTU lærer man at anvende sandhedstabeller, som demonstrerer gyldigheden af en formel. Eksempelvis vises hvordan 2 er gyldig.

$$P \wedge (Q \wedge R) \iff (P \wedge Q) \wedge R \tag{2}$$

Vi kan også bruge PBT til at undersøge, om (2) holder, ved at definere egenskaben som en funktion af P, Q og R, som vist i Listing 3 (4).

```
#r "nuget: FsCheck"
open FsCheck

// proposition formula: bool -> bool -> bool -> bool
let propositional_formula P Q R =
    (P && ( Q && R)) = ((P && Q) && R)
```

Listing 4: PBT af ligning 2. Begge sider er omgivet af parenteser da = har en højere præcedens end &&

```
> let _ = Check.Quick propositional_formula;;
Ok, passed 100 tests.
```

Listing 5: Output ved PBT af (2)

Check.Quick er en del af "FsCheck"biblioteket, den tager en funktion som argument, og generere en række tilfældige input til funktionen på baggrund af funktionens type. Hvis funktionen returnere "true"for alle input, vil testen lykkedes. Hvis funktionen returnere "false"for et input, vil testen fejle og give et eksempel på et input der fejlede. I Listing 4 er der anvendt "Check.Quick"til at teste om (2) er gyldig. Funktionen "Check.Quick"returnere "Ok, passed 100 tests."hvilket indikerer at (2) er gyldig. Det vigtigt her at forstå dette ikke er det samme som at bevise at den er gyldig, da ikke alle muligheder er blevet testet. I nogle tilfælde vil det være en fordel at opskrive en PBT før implementeringen af en funktion som man ved skal overholde en egenskab, på den måde anvende Test Driven Development (TDD)[2] til at teste om ens egenskab forbliver overholdt, under implementering.

3 Symbolske udtryk

Det ønskes at kunne repræsentere simple udtryk som en type i F#. Vil derfor gennemgå en del teori og funktion som er nødvendige for at kunne dette. Det vil give os et grundlæggende fundament for at kunne udføre matematiske evalueringer som differentiering i F#. Som de fleste

[2] *Test-Driven Development*.

andre programmer har F# kun float og int som kan repræsentere tal. Derfor vil vi begynde med at definere et modul som indeholder en type for tal. Tanke gangen her at gennemgå en opbygning af en måde at kunne repræsentere udtryk samt simplificere dem. Vi begrænset os selv til at kun have matematiske operationer som addition, subtraktion, negation, multiplikation og division.

3.1 Tal mængder

Vi begynder med opbygningen af et modul, der kan repræsentere talgrupper. Typen for tal består af tre konstruktører, henholdsvis for heltal, rationale tal og komplekse tal. Dog er modulet designet med henblik på, at det kan udvides med flere taltyper. Ved udvidelse er det eneste krav til den nye talmængde, at der er definerede matematiske operationer i form af addition, subtraktion, negation, multiplikation og division. Desuden skal tallene inden for addition og multiplikation være associative. Dette gælder for eksempel ikke for en vektor, hvorfor vi senere vil overveje at udvide programmet med en type for vektorer. En udvidelse kunne være for reelle tal, som kan håndtere "floating point errors"[3], men for at undgå at komplicere programmet yderligere vil vi i denne opgave ikke inkludere decimal tal.

3.1.1 Rationelle tal modul

Repræsentationen af rationale tal kan laves ved hjælp af at danne et par af heltal, hvor det ene heltal er tælleren, og det andet er nævneren.

```
type rational = R of int * int
```

Listing 6: Typen for rationelle tal

Nedenfor er der givet en signaturfil for rational modulet 7. I implementeringsfilen overloades de matematiske operatorer ved hjælp af de klassiske regneregler for brøker[4].

```
module rational

[<Sealed>]
type rational =
    static member ( ~- ) : rational -> rational
    static member ( + )  : rational * rational -> rational
    static member ( + )  : int * rational -> rational
    static member ( - )  : rational * rational -> rational
    static member ( - )  : int * rational -> rational
    static member ( * )  : int * rational -> rational
    static member ( * )  : rational * int -> rational
    static member ( * )  : rational * rational -> rational
    static member ( / )  : rational * rational -> rational
    static member ( / )  : int * rational -> rational
    static member ( / )  : rational * int -> rational
    static member ( / )  : int * int -> rational

val make         : int * int -> rational
val equal        : rational * rational -> bool
val posetive     : rational -> bool
```

[3] *Floating-point error*.
[4] *Rational Number wikipedia*.

```
21 val toString      : rational -> string
22 val isZero        : rational -> bool
23 val isOne         : rational -> bool
24 val isInt         : rational -> bool
25 val makeRatInt    : rational -> int
26 val greaterThan   : rational * rational -> bool
27 val isNegative    : rational -> bool
28 val absRational   : rational -> rational
```

Listing 7: Signaturfilen for rational-modulet

For at kunne sammenligne og også for nemmere at undgå for store brøker, vil alle rationelle tal blive reduceret til deres simplest form. Dette gøres ved at finde den største fælles divisor (GCD)[5]. Derudover er det vigtigt at være opmærksom på ikke at foretage nul division. Derfor vil implementeringsfilen kaste en "System.DivideByZeroException", hvis nævneren er eller bliver nul. Signaturfilen indeholder en række funktioner, som anvendes af andre filer. Det vil desuden være nødvendigt at kunne håndtere overflow, idet heltallene, der repræsenterer de rationelle tal, under eller efter operationen kan blive for store til korrekt at blive repræsenteret af 32-bit. Da denne rapport fokuserer på implementeringen af matematiske koncepters og ikke numeriske algoritmer, vil modulet blot rapportere en fejl, hvis der opstår overflow.

3.1.2 Komplekse tal-modul

Vi skal nu dykke lidt mere ned i implementeringen af et modul for komplekse tal. Derfor er signaturfilen *complex.fsi* givet i Appendiks 5.1. Først defineres en type for komplekse tal, som består af et rationale tal par, henholdsvis for realdelen og imaginærdelen, se Listing 8.

```
1 type complex = C of rational * rational
```

Listing 8: Typen for komplekse tal

Vi begynder dermed med at opskrive en række regneregler i Definition 1 for operationer på komplekse tal, og betragter deres tilsvarende implementering i modulet.

Defination 1 (Regneregler for komplekse tal)**.**
Lad $a, b, c, d \in \mathbb{Q}$ Så er følgende defineret omkring komplekse tal[6]*:*

1. ***Addition***
   $(a + bi) + (c + di) = (a + c) + (b + d)i$
2. ***Subtraktion***
   $(a + bi) - (c + di) = (a - c) + (b - d)i$
3. ***Multiplikation***
   $(a + bi) \cdot (c + di) = (ac - bd) + (bc + ad)i$
4. ***Kvadratisk form***
   $(a + bi) \cdot (a - bi) = a^2 + b^2$
5. ***Konjugering***
   $\overline{a + bi} = a - bi$

[5] *Greatest common divisor wikipedia.*
[6] *Mathematics 1a*, se. Definition 3.3 s. 54, Definition 3.5 s. 56, Definition 3.8 s. 57, ligning 3-2 s. 58.

6. ***Division***
   $\frac{a+bi}{c+di} = \frac{(a+bi)\cdot(c-di)}{c^2+d^2}$

Ved implementeringen, se Listing 9, af addition, subtraktion, multiplikation samt skalering med et rationelt tal, som kan udledes fra multiplikation ved at lade den imaginære del være 0, er simple operationer, som ikke behøver at defineres i særskilte funktioner, men kan anvendes direkte på overskrivningen af deres respektive operatorer. Dog er det nødvendigt at definere multiplikation som en funktion, da den skal anvendes af divisionsfunktionen.

```
// complexDivRational: complex -> rational -> complex
let complexDivRational c (n) =
    match c with
    | _ when isZero n ->  raise (System.DivideByZeroException("Complex.
    divRational: Cannot divide by zero!"))
    | C (a, b) -> C (a / n, b / n)

// mulConjugate: complex -> rational
let mulConjugate (C(a, b)) = a*a + b*b

// conjugate: complex -> complex
let conjugate (C (a, b)) = C (a, -b)

// mulComplex: complex -> complex -> complex
let mulComplex (C (a, b)) (C (c, d)) = C(a*c-b*d, b*c+a*d)

// divComplex: complex -> complex -> complex
let divComplex z1 z2 =
    complexDivRational (mulComplex z1 (conjugate z2)) <| mulConjugate z2

type complex with
    static member (+)  (C(a, b), C(c, d)) =  C(a + c, b + d)
    static member (-)  (C(a, b), C(c, d)) =  C(a - c, b - d)
    static member (*)  (n, C(a, b))       =  C(n * a, n * b)
    static member (*)  (C(a, b), n)       =  C(n * a, n * b)
    static member (*)  (z1, z2)           =  mulComplex z1 z2
    static member (/)  (z, n)             =  complexDivRational z n
    static member (/)  (z1, z2)           =  divComplex z1 z2
    static member (~-) (C(a, b))          =  C(-a, -b)
```

Listing 9: Overskrivning af operationer på komplekse tal

Bortset fra division af to komplekse tal, ligner de resulterende overbelastninger på operationerne deres respektive matematiske definitioner. Men når vi nærmere studerer divisionen af to komplekse tal, ser vi, at der blot er brug for få funktioner til at kunne udføre divisionen. Først konjugeres nævneren, derefter multiplikeres resultatet med tælleren. Til sidst divideres resultatet af multiplikationen med kvadratet af nævneren. Da komplekse tal som en talmængde indeholder heltal og rationale tal, vil vi i det følgende afsnit omkring tal modulet anvende komplekse tal til udføre de matematiske operationer i modulet.

3.1.3 Tal modulet

Vi har nu beskrevet en måde at kunne repræsentere bruger definere tal på ved brug af typer i F#. Det vil derfor være oplagt at have en type som indeholder alle de typer tal vi ønsker at kunne anvende i de matematiske udtryk vi er ved at opbygge. Fordelen ved at samle dem til en type er at vi kan lave en række funktioner blandt andet matematiske operationer som kan

anvendes på alle type tal. Ved at samle dem til en type kan vi også opskrive en række egenskaber i 1 som vi ønsker at de skal opfylde. Egenskaberne vil blive testet i sektion 3.7.1.

Egenskab 1 (Egenskaber for tal)**.**
Lad $a, b, c \in$ Number, så gælder følgende egenskaber[7]*:*

1. *Addition og multiplikation er* ***associativitet***
   $a + (b + c) = (a + b) + c \wedge a \cdot (b \cdot c) = (a \cdot b) \cdot c$
2. *Addition og multiplikation er* ***Kommutativitet***
   $a + b = b + a \wedge a \cdot b = b \cdot a$
3. ***Distributivitet*** *af multiplikation over addition*
   $a \cdot (b + c) = a \cdot b + a \cdot c$
4. *Addition og multiplikation har et* ***neutral element***
   $a + 0 = a \wedge a \cdot 1 = a$
5. ***Omvendt funktion*** *eksistere til addition*
   $a + (-a) = 0$
6. ***Omvendt funktion*** *eksistere til multiplikation for $a \in$ Number $\setminus \{0\}$*
   $a \cdot a^{-1} = 1$

Vi begynder med at definere en type for tal 10, som indeholder konstruktører for de tal typer vi har definerede samt en for heltal.

```
1     type Number = | Int of int | Rational of rational | Complex of complex
```

Listing 10: Typen for Number

Betragtes signatur filen for Number modulet 11, ses det at der igen er defineret overloading af de anvendte matematiske operationer. Derudover er der defineret en række funktioner som kan anvendes på Number typen.

```
1  module Number
2  open rational
3  open complex
4
5  type Number =
6      | Int of int
7      | Rational of rational
8      | Complex of complex
9  with
10     static member ( + )  : Number * Number -> Number
11     static member ( - )  : Number * Number -> Number
12     static member ( * )  : Number * Number -> Number
13     static member ( / )  : Number * Number -> Number
14     static member ( ~- ) : Number -> Number
15
16 val zero         : Number
17 val one          : Number
18 val two          : Number
19 val isZero       : Number -> bool
```

[7] *Mathematics 1a*, se. Side 60 Theorem 3.10 og Theorem 3.11.

```
20 val isOne         : Number -> bool
21 val isNegative    : Number -> bool
22 val absNumber     : Number -> Number
23 val greaterThan   : Number -> Number -> bool
24 val tryReduce     : Number -> Number
25 val toString      : Number -> string
26 val conjugate     : Number -> Number
27 val inv           : Number -> Number
28 val isInt         : Number -> bool
```

Listing 11: Signatur filen for Number modulet

Ved implementeringen af de matematiske operationer, hvis der eksistere en konstruktør i Number, der repræsentere en tal mængde hvor alle andre konstruktører er delmængder af denne mængde. Er det muligt at definere en enkelt funktion som kan udføre alle binærer operationer. Som et eksempel er funktionen 12 givet, som tager to tal og en funktion i form af den ønskede binærer operation som parameter. Funktionen vil derefter matche på de to tal og anvende den operation på de to tal.

```
1 // makeRational: Number -> rational
2 let makeRational a =
3     match a with
4     | Int x       -> make(x, 1)
5     | Rational x  -> x
6
7 // operation: Number -> Number -> (rational -> rational -> rational) -> Number
8 let operation a b f =
9     f (makeRational a) (makeRational b) |> Rational
```

Listing 12: Number.operation funktionen

Det vil her til være oplagt på alle de matematiske operationer at anvende en funktionen til at forsøge at konvenere tal typen til den simpleste talmængde, som i vores tilfælde er heltal. Dette er gjort ved at anvende funktionen tryMakeInt på alle de matematiske operations overloadnings 13.

```
1 // tryMakeInt: Number -> Number
2 let tryMakeInt r =
3     match r with
4     | Rational a when isInt a -> Int (makeRatInt a)
5     | _ -> r
6
7 type Number with
8   static member (+) (a, b) = operation a b (+) |> tryMakeInt
9   static member (-) (a, b) = operation a b (-) |> tryMakeInt
10  static member (*) (a, b) = operation a b (*) |> tryMakeInt
11  static member (/) (a, b) = operation a b (/) |> tryMakeInt
12  static member (~-) (a)   = neg a |> tryMakeInt
```

Listing 13: Overloadnings funktionerne for Number

Dermed har vi et modul som kan repræsentere tal, samt udføre matematiske operationer på dem. Vi vil nu begynde at betragte hvordan vi kan anvende den i et lignings udtryk.

3.2 Matematiske udtryk

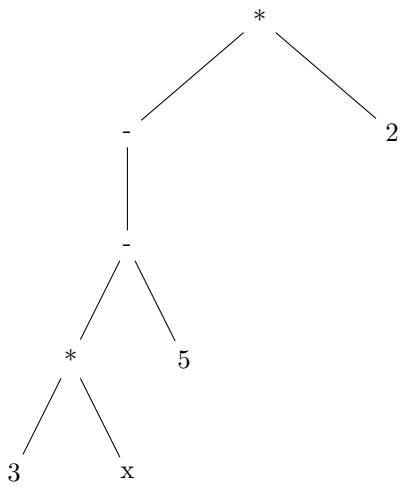
3.2.1 Polsk notation

Matematiske udtryk som vi normalt kender dem er skrevet med infix notation. I infix notation skrives en binær operator mellem to operandere, kende tegnet for sproget er at det indeholder parenteser samt præcedens regler. Dette gør det generalt kompliceret at evaluere og håndtere matematiske udtryk i et programmeringssprog. Derfor er det mere oplagt at kunne anvende polsk notation (prefix) istedet, hvor operatoren skrives før operandere eller omvendt polsk notation (postfix). Da de hverken indeholder parenteser eller præcedens regler[8].

$$\begin{aligned} \text{Infix Notation:} &\quad (A+B)\cdot C \\ \text{Prefix Notation:} &\quad \cdot +ABC \\ \text{Postfix Notation:} &\quad A\,B\,+\,C\,\cdot \end{aligned}$$

3.2.2 Udtryk som træer

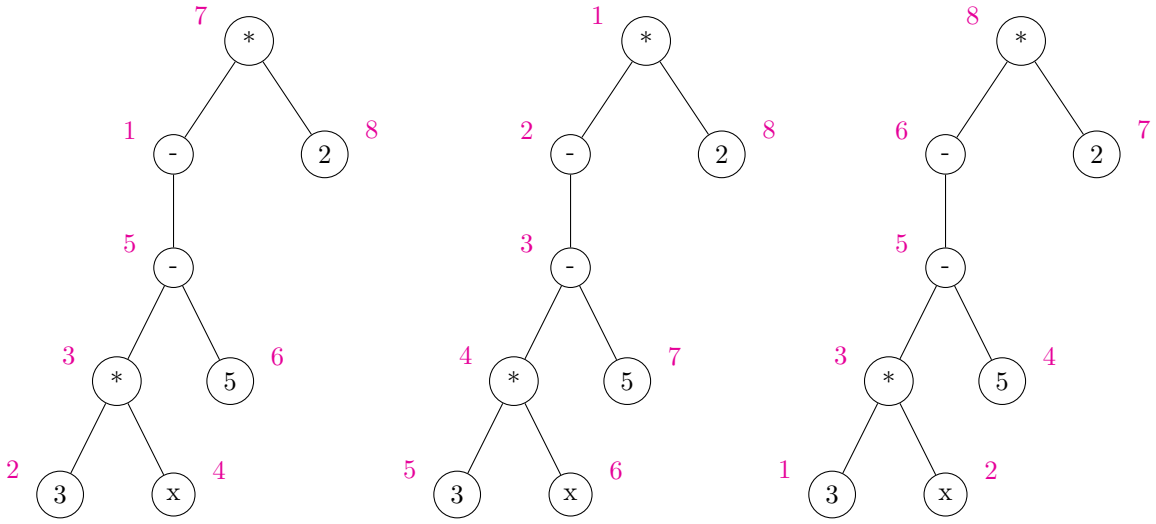
Et matematisk udtryk kan repræsenteres som et binært træ, hvor bladene er operander i det anvendte matematiske rum og alle andre noder er operationer. Som eksempel kan udtrykket $-(3 \cdot x - 5) \cdot 2$ repræsenteres som følgende træ Figur 1.



Figur 1: Et binært træ der repræsenterer udtrykket $-(3 \cdot x - 5) \cdot 2$

Det skal bemærkes der er forskel på den unære og binære operator '$-$' i træet, den unære betyder negation og den binære er subtraktion. Givet et binært træ for et matematisk udtryk, vil det være muligt omdanne dem til infix, prefix eller postfix notation. Dette kan gøres ved at anvende modificeret Inorder, Preorder eller Postorder Traversal[9], algorithmerne er illustreret i Figur 2.

[8] *Polsk notation wikipedia.*
[9] *Tree Traversal Techniques – Data Structure and Algorithm Tutorials.*



Modificeret Inorder Traversal: $-(3 \cdot x - 5) \cdot 2$

Preorder Traversal: $\cdot - - \cdot 3\, x\, 5\, 2$

Postorder Traversal: $3\, x\, \cdot 5\, - -2 \cdot$

Figur 2: Træet fra Figur 1 med forskellige travesal metoder

Vi vil i 3.2.3 betragte hvordan vi kan implementere et modul som kan repræsentere udtryk ved brug af prefix notation. Postorder Traversal blive anvendt til at kunne rekursivt simplificere og evaluere udtryk.

Grundet præcedens regler i infix notation, er det nødvendigt at modificere Inorder Traversal, da unære noder altid skal håndteres før dens børn. Desuden vil det også være nødvendigt at implementere regler for at håndtere parenteser, hvis der ønskes et symbolsk udtryk. Den modificeret Inorder Traversal anvendes til at kunne visualisere udtrykket i infix notation.

3.2.3 Udtryksmodulet

Efter udviklingen af et modul til repræsentation af talmængder er vi nu klar til at udvide med et modul for matematiske udtryk. Vi starter med at definere en polymorf type for udtryk, som beskrevet i Listing 14. Denne type omfatter flere konstruktører, hver tilknyttet specifikke matematiske operationer vi ønsker at implementere. Desuden introducerer vi konstruktøren `N` til at repræsentere numeriske værdier ved at anvende talmængder defineret i Listing 10. Til sidst tilføjer vi konstruktøren `X` for variable. Således lagres matematiske udtryk i en træstruktur, se 3.2.2, eftersom hver konstruktør for en operation indeholder et eller to underudtryk af samme type.

```
type Expr<'a> =
    | X of char
    | N of 'a
    | Neg of Expr<'a>
    | Add of Expr<'a> * Expr<'a>
    | Sub of Expr<'a> * Expr<'a>
    | Mul of Expr<'a> * Expr<'a>
    | Div of Expr<'a> * Expr<'a>
```

Listing 14: Typen for Expr

Expr<'a> typen er dermed en polymorfisk type, hvor 'a er typen for den tal mængde hvor vi kan lave brugerdefinerede matematiske operationer. Et exemplar på en Expr<Number> er givet i Listing 15. Her ses det at når udtrykket $-(3 \cdot x - 5) \cdot 2$ visualiseres er det i prefix notation.

```
> tree "-(3*x-5)*2";;
val it: Expr<Number> =
  Mul (Neg (Sub (Mul (N (Int 3), X 'x'), N (Int 5))), N (Int 2))
```

Listing 15: $-(3 \cdot x - 5) \cdot 2$ som et udtryks træ. Funktionen tree bliver beskrevet i 3.3.1.

Signatur filen indeholder overloadings på de matematiske operationer, så de kan anvendes på udtryk. Samt en funktion eval til at evaluere et udtryk.

```
module Expression
open Number

type Expr<'a> =
  | X of char
  | N of 'a
  | Neg of Expr<'a>
  | Add of Expr<'a> * Expr<'a>
  | Sub of Expr<'a> * Expr<'a>
  | Mul of Expr<'a> * Expr<'a>
  | Div of Expr<'a> * Expr<'a>
  with
  static member ( ~- ) : Expr<Number> -> Expr<Number>
  static member ( + ) : Expr<Number> * Expr<Number> -> Expr<Number>
  static member ( - ) : Expr<Number> * Expr<Number> -> Expr<Number>
  static member ( * ) : Expr<Number> * Expr<Number> -> Expr<Number>
  static member ( / ) : Expr<Number> * Expr<Number> -> Expr<Number>

val eval : Expr<Number> -> Map<char,Number> -> Number
val containsX : Expr<Number> -> Expr<Number> -> bool
val getNumber : Expr<Number> -> Number
val getVariable : Expr<Number> -> char
val isAdd : (Expr<Number> -> Expr<Number> -> Expr<Number>) -> bool
val isSub : (Expr<Number> -> Expr<Number> -> Expr<Number>) -> bool
val isMul : (Expr<Number> -> Expr<Number> -> Expr<Number>) -> bool
val isDiv : (Expr<Number> -> Expr<Number> -> Expr<Number>) -> bool
val isNeg : (Expr<Number> -> Expr<Number>) -> bool
```

Listing 16: Signatur filen for Expression modulet

De overloadede matematiske operatorer i Expressions, laver overflade evalueringer samt simplifikationer på deres respektive argumenter. Overfalde evalueing vil sige at de individuellen funktioner kun betragter de to øverste niveauer på de udtryks træer de tager som input, mullige implementeringer af addition og multiplikation er givet i Listing 17.

```
// add: Expr<Number> -> Expr<Number> -> Expr<Number>
let rec add e1 e2:Expr<Number>  =
  match e1, e2 with
  | N a, N b                                -> N (a + b)
  | N a, b | b, N a when isZero a           -> b
  | Mul(a, X b), Mul(c, X d)
  | Mul(X b, a), Mul(c, X d)
  | Mul(a, X b), Mul(X d, c)
  | Mul(X b, a), Mul(X d, c) when b = d -> Mul(add a c, X b)
  | _, _                                    -> Add(e1, e2)

// mul: Expr<Number> -> Expr<Number> -> Expr<Number>
let mul e1 e2:Expr<Number> =
  match e1, e2 with
  |N a, N b                           -> N (a * b)
  |N a, b | b, N a when isOne a       -> b
  |N a, _ | _, N a when isZero a      -> N zero
  | _, _                              -> Mul(e1, e2)
```

Listing 17: Addition og multiplikation af to udtryk

3.3 Evaluering af udtryk

Vi vil nu betragte hvordan vi kan evaluere et udtryk, ved hjælp af et miljø som indeholder værdier for variable som er indeholdt i udtrykket. Evalueringen af udtryk skal kunne opfylde følgende homomorfiske egenskaber 2. Egenskaben vil blive testet i sektion 3.7.2.

Egenskab 2 (Homomorfisme af evaluering)**.**
Lad $\oplus \in \{+, -, \times, /\}$ *sættet af binære operationer,* $e1$ *og* $e2$ *være udtryk, så gælder følgende:*

$$eval(e1 \oplus e2) = eval(e1) \oplus eval(e2)$$

Derudover skal det om negation også gælde at:

$$eval(-e) = -eval(e)$$

Funktionen eval i Listing 18 tager et udtryk og et miljø som input og evaluere udtrykket til en numerisk værdi. Funktionen kører en Postorder Traversal på udtrykket og evaluerer dermed udtrykket nedefra og op, ved at foretage matematiske operationer defineret i Number modulet.

```
// eval: Expr<Number> -> Map<char, Number> -> Number
let rec eval (e:Expr<Number>) (env) =
  match e with
  | X x -> Map.find x env
  | N n -> n
  | Neg a -> - eval a env
  | Add (a, b) -> eval a env + eval b env
  | Sub (a, b) -> eval a env - eval b env
  | Mul (a, b) -> eval a env * eval b env
  | Div (a, b) -> eval a env / eval b env
```

Listing 18: Evaluering af et udtryk

3.3.1 Konventering mellem udtryks notation

Det er ønkes at kunne konvertere udtryk frem og tilbage mellem prefix notation, repræsenteret af Expression-typen, og den standard infix notation. Dette ønske skyldes, at infix notation er lettere for os at læse og skrive. Derfor er det essentielt, at de to konverteringsfunktioner fungerer som hinandens inverser. Dette krav er yderligere uddybet i egenskab 3. Egenskaben bliver test i sektion 3.7.3.

Egenskab 3 (Invers morphism[10] mellem infix og prefix)**.**
Lad Q^n være mængden af rationelle infix udtryk repræsenteret som en string, med n variable, så defineres følgende:

$$tree : Q^n \to Expr$$
$$tree^{-1} : Expr \to Q^n$$

Dermed gælder følgende egenskaber

$$tree^{-1} \circ tree = id_{Q^n}$$
$$tree \circ tree^{-1} = id_{Expr}$$

Hvor id_x er identitetsfunktionen på mængden x.

Vi begynder med at betragte den inverse funktion, som konverterer fra en expression til infix notation. Funktionen etf se Listing 19 fortager denne konventering ved at lave en modificeret Inorder Traversal på udtrykket, som beskrevet i 3.2.2. Den modificeret del er at håndtere parenteser samt håndtere negation som var det en binær node i træet hvor det venstre barn er et tomt udtryk.

```
// parenthesis: bool -> string -> string
let parenthesis b f = if b then "(" + f + ")" else f

// etf: Expr<Number> -> bool -> string
let rec etf e p =
    match e with
    | N a when not <| isInt a -> parenthesis p <| toString a
    | N a   -> toString a
    | X a   -> string a
    | Neg a -> parenthesis p <| "-" + etf a (not p)
    | Add(a, b) -> parenthesis p <| etf a false + "+" + etf b false
    | Sub(a, b) -> parenthesis p <| etf a false + "-" + etf b true
    | Mul(a, b) -> parenthesis p <| etf a true  + "*" + etf b true
    | Div(a, b) -> parenthesis p <| etf a true  + "/" + etf b true


// infixExpression: Expr<Number> -> string
let infixExpression e = etf e false
```

Listing 19: konventering fra expression til infix notation

Funktionen `tree`, som foretager konverteringen fra infix notation til et udtrykstræ, er baseret på algoritmen beskrevet i [1][11]. Først konverteres en udtryksstreng til en liste af tokens. Disse

[10] *Inverse function.*
[11] *Convert Infix expression to Postfix expression.*

tokens beskriver, om en karakter i udtrykket er en operand, en operator, eller en konstant, hvor en operator også indeholder information om præcedens og associativitet[12]. Typen for disse tokens kan ses i Listing 20. Herefter anvendes to stacks: én for operatorer og én for udtryk. Der anvendes en række regler, som beskrevet i [1], for hvornår der skal udføres pop og push på disse to stacks. Det skal bemærkes, at når en operator pushes til udtryksstacken, da navnet på operatorkonstruktøren på udtrykket står skrevet fra venstre mod højre, vil udtryksstacken være i prefix notation og ikke postfix notation som beskrevet i kilden. Funktionen `tree` er at finde i Appendiks 5.2.

```
type Associative = | Left | Right
type Precedence = int
type Operator = char * Precedence * Associative
type Token =
    | Operand of char
    | Operator of Operator
    | Constant of int
type OperatorList = Operator list
```

Listing 20: Konvertering fra infix til udtrykstræ

3.4 Simplifikation af udtryk

Vi skal nu betragte en sytematisk metode til at kunne simplificere matematiske udtryk, ved hjælp af simple algebraiske regler. Dette er en nødvendig at kunne for at bruge udtrykkene i en matematisk sammenhæng, da det vil kunne medføre både en reduktion i kompleksitet og en forbedring i læsbarhed når udtrykene visualiseres. Før vi betragter metoden, kan vi opskrive en egenskab som simplification skal overholde. Egenskaben vil blive testet i sektion 3.7.4, det er en nødvendighed at evaluere udtrykket før og efter simplifikationen da det ikke er kompleks opgave at skulle sammenligne og evaluere to udtryk er ækvivalente.

Egenskab 4 (Simplifikation af udtryk)**.**
Lad e være et udtryk, så gælder følgende:

$$eval(e) = eval(simplifyExpr(e))$$

Vi begynder med at betragte funktionen `simplifyExpr` i Listing 21, som simplificerer et udtryk ved at foretage en Postorder Traversal på udtrykket. På den måde sikre sig at når en node i udtrykstræet bliver simplificeret, vil dens børn allerede være simplificeret.

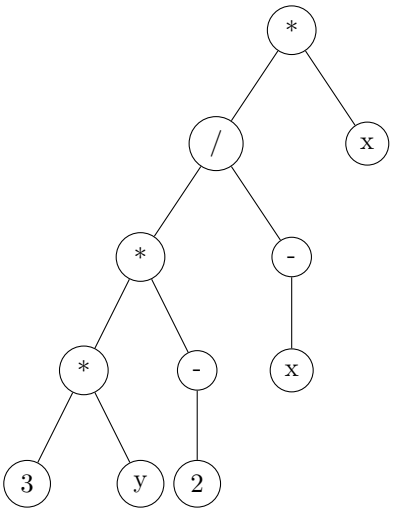
```
//simplifyOperation: Expr<Number> -> Expr<Number> -> (Expr<Number> -> Expr<
  Number> -> Expr<Number>) -> Expr<Number>
let rec simplifyOperation e1 e2 f =
match f e1 e2 with
| Neg a ->
  commutativeMulDiv.applyCommutative (Neg a) |> commutativeAddSub.
  applyCommutative
| Add(a, b) when isAdd f -> commutativeAddSub.applyCommutative (Add(a, b))
| Sub(a, b) when isSub f -> commutativeAddSub.applyCommutative (Sub(a, b))
| Mul(a, b) when isMul f -> commutativeMulDiv.applyCommutative (Mul(a, b))
| Div(a, b) when isDiv f -> commutativeMulDiv.applyCommutative (Div(a, b))
```

[12] *Operator Precedence and Associativity in C*.

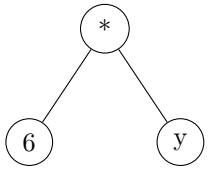
```
10    | Add(a, b) -> simplifyOperation a b (+)
11    | Sub(a, b) -> simplifyOperation a b (-)
12    | Mul(a, b) -> simplifyOperation a b (*)
13    | Div(a, b) -> simplifyOperation a b (/)
14    | a -> a
15
16
17  //simplifyExpr: Expr<Number> -> Expr<Number>
18  let rec simplifyExpr e =
19    match e with
20    | N a when Number.isNegative a -> Neg (N (Number.absNumber a))
21    | N (Rational(R(a, b))) ->
22      simplifyOperation (simplifyExpr (N (Int a))) (simplifyExpr (N (Int b)))
        (/)
23    | Neg a      -> - (simplifyExpr a)
24    | Add(a, b) -> simplifyOperation (simplifyExpr a) (simplifyExpr b) (+)
25    | Sub(a, b) -> simplifyOperation (simplifyExpr a) (simplifyExpr b) (-)
26    | Mul(a, b) -> simplifyOperation (simplifyExpr a) (simplifyExpr b) (*)
27    | Div(a, b) -> simplifyOperation (simplifyExpr a) (simplifyExpr b) (/)
28    | _ -> e
```

Listing 21: Simplifikation af et udtryk

Det er simplifyOperation, som foretager selve simplificeringen af et givet udtryk. Funktionen tager som input to udtryk samt den binære operation, der skal anvendes på disse. Funktionen anvendes på udtrykkene, hvorefter overfladisk simplifikation, som blev beskrevet i 3.2.3, udføres. Hvis overfladisk simplifikation resulterer i en ændring af den anvendte operation, kalder funktionen sig selv rekursivt med de to udtryk og den nye operation. Hvis overfladisk simplifikation ikke resulterer i ændringer i operationen, vil funktionen foretage en dybere simplifikation af de to udtryk. Denne dybere simplifikation udføres af funktionerne applyCommutative fra filerne *CommutativeAddSub.fs* og *CommutativeMulDiv.fs*, som findes i Appendiks 5.3 og 5.4. Disse funktioner arbejder efter samme princip, hvor de starter med at flade udtrykstræerne ud ifølge de kommutative regler for henholdsvis addition og multiplikation. Derefter sorterer de udtrykstræet, hvilket muliggør at foretage overfladisk når ved at gendanne træet. For multiplikation anvendes samme metode i nævneren for division, og der undersøges, om der er fælles udtryk i det udfladede træ, som fremtræder i nævneren af en division og i det udfladede træ, der indeholder divisionen. Et eksempel på anvendelse af den kommutative multiplikationssimplifikation på et udtryk er givet i Figur 3, som viser det visuelle input og det resulterende svar fra funktionen, samt Figur 4, der viser det udfladede træ og sorteringen af det fladede træ.

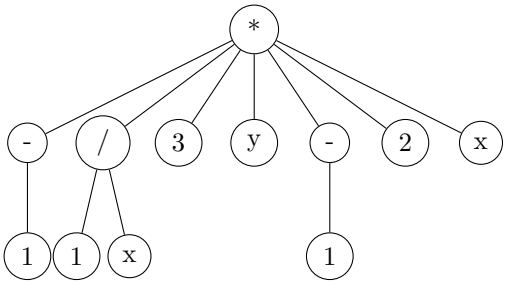


(a) Udtrykket $(3 \cdot y \cdot (-2)/-x) \cdot x$ som et træ

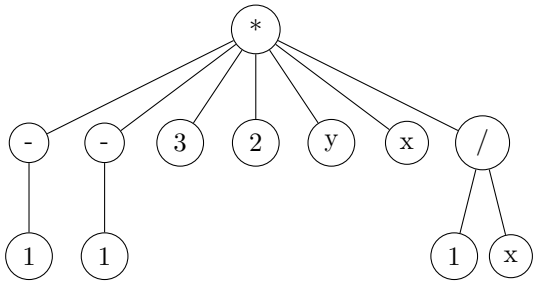


(b) Det resuterende simplificerede efter at have anvendt CommutativeMulDiv.applyCommutative på udtrykket i Figur 3a

Figur 3: Før og efter simplifikation af et udtryk ved brug af CommutativeMulDiv.applyCommutative



(a) Det udfladede træ



(b) Det sorterede træ fra venstre mod højre

Figur 4: Det udfladede og sorterede af udtrykket $(3 \cdot y \cdot (-2)/-x) \cdot x$ i processen af kommutativ simplifikation

Generelt, når det gælder simplificering af udtryk, skal man være opmærksom på ikke at ende i et uendeligt loop. Derfor er det vigtigt ikke at definere nogle overfladiske simplificeringer, som er hinandens inverse funktioner. Desuden forsøger funktionerne i dette program altid at skubbe negation af udtryk så langt ud som muligt i håb om, at de kan ophæve hinanden. Dette illustreres blandt andet i figur 4, hvor ved udfladning af træet, hvis et af de kommutative udtryk for multiplikation er negativt, fjernes negationen, og der tilføjes i stedet en negation af tallet 1, som ved sortering skubbes til venstre.

3.5 Differentiering af udtryk

Vi kan nu betragte den første implementering, der benytter vores udtryksmodul, som samtidig vil understrege vigtigheden af at kunne simplificere udtryk. Vi begynder igen med at opskrive nogle algebraiske linearitetsegenskaber, som differentieringen skal overholde. Disse egenskaber vil blive testet i sektion 3.7.5.

Egenskab 5 (Linearitetsbetingelserne[13])**.**
Lad f og g være udtryk, og a og b være tal fra talmodulet, så gælder følgende:

1. ***Skaleringsreglen***
   $\frac{d}{dx}(af) = a\frac{df}{dx}$
2. ***Sumreglen***
   $\frac{d}{dx}(f+g) = \frac{df}{dx} + \frac{dg}{dx}$
3. ***Subtraktionsreglen***
   $\frac{d}{dx}(f-g) = \frac{df}{dx} - \frac{dg}{dx}$
4. ***Produktreglen***
   $\frac{d}{dx}(fg) = \frac{df}{dx}g + f\frac{dg}{dx}$
5. ***Kvotientreglen***
   $\frac{d}{dx}\left(\frac{f}{g}\right) = \frac{\frac{df}{dx}g - f\frac{dg}{dx}}{g^2}$

Funktionen for differentiering diff i Listing 22, som tager et udtryk og en variabel som input og differentierer udtrykket med hensyn til variablen. Dette er en af de store fordele ved at anvende et funktionelt programmeringssprog, da det ses, hvordan fire af reglerne fra egenskab 5 er implementeret direkte, som de er beskrevet.

```
// diff: Expr<Number> -> char -> Expr<Number>
let rec diff e dx =
    match e with
    | X f when f = dx -> N (Int 1)
    | X _ -> N (Int 0)
    | N _ -> N (Int 0)
    | Neg f -> diff (Mul (N (Int 1), f)) dx
    | Add(f, g) -> Add(diff f dx, diff g dx)
    | Sub(f, g) -> Sub(diff f dx, diff g dx)
    | Mul(f, g) -> Add(Mul(diff f dx, g), Mul(f, diff g dx))
    | Div(f, g) -> Div(Sub(Mul(diff f dx, g), Mul(f, diff g dx)), Mul(g, g))
```

Listing 22: Differentiering af et udtryk

[13] *Differentiation Rules*.

3.6 Multivariable polynomialer af første grad

3.7 PBT af udtryk

3.7.1 Tal modulet

3.7.2 Homomorfisme af evaluering

3.7.3 Invers morphism mellem infix og prefix

3.7.4 Simplifikation af udtryk

3.7.5 Differentiering af udtryk

4 Vektorer og Matricer

Vi vil nu betragte et modul for vektorer og matricer. Da en Vector også kan betragtes som en matrix, vil vi herfra når der omtalles begge kun referare til en matrix. For at kunne håndtere matricer på en systematik måde begynder vi med at defienre en type for major order.

Vi vil nu betragte opbyggelsen af et modul for vektorer og matricer. Eftersom en vektor også kan opfattes som en matrix, vil vi i det følgende, når begge dele omtales, udelukkende referere til matricer. For systematik at kunne håndtere matricer, starter vi med at definere en type for lagringsordning.

```
1 type Order = | R | C
```

Listing 23: Typen for order

Typen Order (se Listing 23), anvendes til at angive, om en matrix er i rækkefølge (row-major) eller kolonnefølge (column-major)[14]. En vektor, der er lagret i kolonnefølge, kan betragtes som den transponeret rækkefølge vektor. Vi kan derfor nu definere en type for matricer, ved hjælp af en type for vektore (Listing 24).

```
1 type Vector = V of list<Number> * Order
2 type Matrix = M of list<Vector> * Order
```

Listing 24: Typen for Matricer

Derudover er det en fordel at kunne kende dimissionen af en matrix. Derfor er der også defineret en type for dimissionen(se Listing 25).

```
1 // Rows x Cols
2 type Dimension = D of int * int
```

Listing 25: Typen for dimissionen

.-

[14] *Row- and column-major order.*

4.1 Matrix operationer

Der vil i denne sektion beskrives en række funktioner som er nødvendige før vi kan betragte nogle rekursive algoritmer som kan anvendes på en matrice. Da modulet indeholder mange hjælpe funktioner, vil der fokuseres på de funktioner med matematisk relevans.

Det muligt at definere en funktion til at finde dimissionen af en matrix (se Listing 26). Funktionen laver et kald til matrixValidMajor genere en fejl hvis ikke alle vektorer og matrien har samme lagringsordning. matrixVectorLength finder længden af en vektor i matricen.

```
    // dimMatrix : Matrix -> Dimension
let dimMatrix (M(vl, o)) =
    if vl = [] then D (0, 0)
    else
    let _ = matrixValidMajor (M(vl, o))
    let d1 = List.length vl
    let d2 = matrixVectorLength (M(vl, o))
    match o with
    | R -> D (d1, d2)
    | C -> D (d2, d1)
```

Listing 26: Funktion til at finde dimissionen af en matrix

Hvis en matrix er gemt som rækkefølge, vil antallet af rækker være længden af en vektor og antallet af kolonner være længden af vektor listen, og omvendt for kolonnefølge.

4.1.1 Matematiske operationer

I denne section bør bemærkes flere Listings ikke anvender fejlhåndtering, dette er udelukket for læsbarhedens skyld. De anvendte funktioner i mondulet har passende dimensions tjek på matricerne se evt. appendiks 5.5, der sikre at operationerne altid er lovlige.

4.1.1.1 Skalering af en matrix

Vi begynder med at betragte en funktion til at skalere en matrix (se Listing 27).

```
// scalarVector : Number -> Vector -> Vector
let scalarVector (n:Number) (V (nl, o)) =
    V ((List.map (fun x -> x * n) nl), o)

// scalarMatrix : Matrix -> Number -> Matrix
let scalarMatrix (M (vl, o)) n =
    M ((List.map (fun x -> scalarVector n x) vl), o)
```

Listing 27: Funktion til at skalere en matrix

Det at skalere en matrice er svare til at skalere hvert element i matricen. Derfor ved at have en funktion scalarVector, der skalere hvert element i en givet vektor bliver scalarMatrix at skalere hver vektor i en givet matrice. List.map svare til at lave en list comprehension i Python[15].

```
// addVector : Vector -> Vector -> Vector
let addVector (V (v1, o1)) (V (v2, _)) =
```

[15] *Python - List Comprehension.*

```
3       V ((List.map2 (+) v1 v2), o1)
4
5   // addMatrix : Matrix -> Matrix -> Matrix
6   let addMatrix (M(vl1, o)) (M(vl2, _)) =
7       M (List.map2 addVector vl1 vl2, o)
8
9   // subVector : Vector -> Vector -> Vector
10  let subVector x y =
11      scalarVector (-one) y |> addVector x
12
13  // sumRows : Matrix -> Matrix
14  let rec sumRows m =
15      if not <| corectOrderCheck m C
16      then sumRows <| correctOrder m C
17      else
18      let zeroVector = vectorOf zero <| matrixVectorLength m
19      let (M(vl, _)) = m
20      matrix [List.fold (addVector) zeroVector vl]
```

Listing 28: Funktion til at addere matricer og substraktion af vektorer

4.1.1.2 Addition af matricer

Addition af vektorer kommer ned til at fortage additionen elementvis i Listing 28, ved brug af List funktionen map2. Vi kan bruge addVector til at definere, matrix addition og subtraktion af vektorer sidst bruges den også til at summere rækkerne i en matrice (sumRows) som vil blive brugt i implementeringen af Gram-Schmidt processen i sektion 4.4, funktionen bliver yderligere beskrevet i defination 2.

Defination 2 (Summering af rækker i en matrix)**.**
Lad A være en matrix med m rækker og n søjler, hvor

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$$

Så gælder om sumRows at

$$sumRows(A) = v = \begin{bmatrix} \sum_{j=1}^{n} a_{1j} \\ \sum_{j=1}^{n} a_{2j} \\ \vdots \\ \sum_{j=1}^{n} a_{mj} \end{bmatrix}$$

Dermed er $v_i = \sum_{j=1}^{n} a_{ij}$ for $i = 1, 2, \ldots, m$.

4.1.1.3 Matrix multiplikation

Defination 3 (Matrix-Vektor Multiplikation)**.**
Lad A være en matrix med m rækker og n søjler, hvor

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$$

og $\mathbf{v} = (v_1, v_2, \ldots, v_n)^T$ være en vektor med n elementer. Så er matrix-vektor Av produktet defineret som

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix} = \begin{bmatrix} a_{11}v_1 + a_{12}v_2 + \cdots + a_{1n}v_n \\ a_{21}v_1 + a_{22}v_2 + \cdots + a_{2n}v_n \\ \vdots \\ a_{m1}v_1 + a_{m2}v_2 + \cdots + a_{mn}v_n \end{bmatrix} = \begin{bmatrix} \sum_{j=1}^{n} a_{1j}v_j \\ \sum_{j=1}^{n} a_{2j}v_j \\ \vdots \\ \sum_{j=1}^{n} a_{mj}v_j \end{bmatrix}$$

Sætning 1 (Matrix-Vektor Multiplikation)**.**
Lad A være en matrix med m rækker og n søjler, og lad $\mathbf{v}$ være en vektor med n elementer. Så gælder der

$$Av = \textit{sumRows} \begin{bmatrix} | & | & & | \\ a_1v_1 & a_2v_2 & \cdots & a_nv_n \\ | & | & & | \end{bmatrix}$$

Bevis. Lad B være resultatet af at skalere søjlerne i matrix A med de tilsvarende elementer i vektoren $\mathbf{v}$. Vi har

$$\begin{aligned} B &= \begin{bmatrix} | & | & & | \\ a_1v_1 & a_2v_2 & \cdots & a_nv_n \\ | & | & & | \end{bmatrix} \\ &= \begin{bmatrix} a_{11}v_1 & a_{12}v_2 & \cdots & a_{1n}v_n \\ a_{21}v_1 & a_{22}v_2 & \cdots & a_{2n}v_n \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}v_1 & a_{m2}v_2 & \cdots & a_{mn}v_n \end{bmatrix} \end{aligned}$$

Ved brug af definition 2 for *sumRows* og definition 3 ses det at

$$\textit{sumRows}(B) = \begin{bmatrix} \sum_{j=1}^{n} a_{1j}v_j \\ \sum_{j=1}^{n} a_{2j}v_j \\ \vdots \\ \sum_{j=1}^{n} a_{mj}v_j \end{bmatrix} = A\mathbf{v}$$

□

Vi kan dermed anvende Sætning 1 til at definere en funktion matrixMulVector for matrix-vektor multiplikation (se Listing 29). Som først skalere søjlerne i matricen med de tilsvarende element i vektoren, og derefter summe rækkerne i matricen.

```
// matrixMulVector : Matrix -> Vector -> Matrix
let rec matrixMulVector (M(vl, _)) (V(nl, _)) =
    M (List.map2 (fun mc n -> scalarVector n mc) vl nl, C)
    |> sumRows
```

Listing 29: Funktion til matrix-vektor multiplikation

Defination 4 (Matrix multiplikation[16])**.**
Lad $A \in \mathbb{F}^{m\times n}$ og $B \in \mathbb{F}^{n\times \ell}$. Antag at søjlerne i B er givet ved $b_1, \ldots, b_\ell \in \mathbb{F}^n$, dermed

$$B = \begin{bmatrix} | & & | \\ b_1 & \cdots & b_\ell \\ | & & | \end{bmatrix}.$$

Så defineres matrice produktet som

$$A \cdot B = \begin{bmatrix} | & & | \\ A \cdot b_1 & \cdots & A \cdot b_\ell \\ | & & | \end{bmatrix}.$$

Udfra defination 4, ses det at funktion matrixProduct i Listing 30 til at multiplicere to matricer, bliver derfor at lave matrix-vektor multiplikation på hver søjle i matricen. Da matrixMulVector returnere en matrix, skal der bruges en funktion til at konvertere matricen til en vektor, hvilket matrixToVector gør.

```
// matrixProduct : Matrix -> Matrix -> Matrix
let rec matrixProduct a (M(vlb, _)) =
    let product = List.map (
        fun bv -> matrixMulVector a bv |> matrixToVector ) vlb
    M(product, C)
```

Listing 30: Funktion til at multiplicere matricer

4.1.1.4 Projektion af en vektor

Som beskrevet i afsnit 2.3 'Projections onto a line' i 'Mathematics 1b'[17], kan projektionen af en vektor defineres som følgende, hvor $Y = \text{span}\{y\}$.

$$\text{proj}_Y : V \to V, \quad \text{proj}_Y(x) = \frac{\langle x, y\rangle}{\langle y, y\rangle} y \tag{3}$$

Med det standard indreprodukt

$$\langle x, y\rangle = y^* x = \sum_{k=1}^{n} x_k \overline{y}_k \tag{4}$$

Den første funktion vi skal bruge er derfor en funktionen til at konjugere en vektor conjugateVector, det gøres ved at konjugere elementerne i vektoren. Udover dette defineres en funktion til at multiplikere to vektorer element vis vectorMulElementWise.

[16] *Mathematics 1a.*
[17] *Mathematics 1b - Functions of several variables.*

```fsharp
// vectorMulElementWise : Vector -> Vector -> Vector
let vectorMulElementWise (V(u, o1)) (V(v, o2)) =
    V (List.map2 (*) u v, o1)

// conjugateVector : Vector -> Vector
let conjugateVector (V(v, o)) =
    V (List.map conjugate v, o)

// innerProduct : Vector -> Vector -> Number
let innerProduct u v =
    let (V(w, _)) = conjugateVector v |> vectorMulElementWise u
    List.fold (+) zero w

// proj : Vector -> Vector -> Vector
let proj y x =
    scalarVector (innerProduct x y / innerProduct y y) y
```

Listing 31: Funktioner til projektere en vektor på en anden

Evalueringen af det standard indre produkt innerProduct mellem to vektore, bliver derfor at konjugere den ene vektor og derefter multiplicere elementvis med den anden vektor. Hvortil summen af elementerne i den resulterende vektor er det indre produkt.

Sidst kan funktionen proj skrives direkte som den er defineres i ligning 3.

4.2 PBT af matrix operationer

Det er nu muligt at opstille nogle PBT af der sikre at matricerne overholder matematiske egenskaber i sætning 6. Først defineres en generator for matricer, som generere matricer med tilfældige tal fra vores talmængde 10.

```fsharp
// vectorGen : int -> Gen<Vector>
let vectorGen n =
    Gen.listOfLength n numberGen |> Gen.map (fun x -> vector x)

// matrixGen : Gen<Matrix>
let matrixGen =
    gen {
        let! row = Gen.choose(1, 6)
        let! col = Gen.choose(1, 6)
        let! vectors = Gen.listOfLength col (vectorGen row)
        return matrix vectors
    }

type MaxtrixGen =
    static member Matrix() =
        {new Arbitrary<Matrix>() with
            override _.Generator = matrixGen
            override _.Shrinker _ = Seq.empty}

type NumberGen =
    static member Number() =
        {new Arbitrary<Number>() with
            override _.Generator = numberGen
            override _.Shrinker _ = Seq.empty}
```

Listing 32: Generatorene anvendt til PBT af matrix operationer i

Egenskab 6 (Vektor Aksiomer[18])**.**
Lad $c, d \in \mathbb{F}$ og $v_i \in \mathbb{F}^n$ for $i = 1 \dots m$ så gælder:

1. $(v_1 + \cdots + v_{m-1}) + v_m = v_1 + (v_2 + \cdots + v_m)$
2. $c \cdot \left(d \cdot \begin{bmatrix} | & & | \\ v_1 & \cdots & v_m \\ | & & | \end{bmatrix} \right) = (c \cdot d) \cdot \begin{bmatrix} | & & | \\ v_1 & \cdots & v_m \\ | & & | \end{bmatrix}$
3. $c \cdot (v_1 + \cdots + v_m) = c \cdot v_1 + \cdots + c \cdot v_m$

Vi kan dermed nu lave definere egenskaberne fra sætning 6 som nogle funktioner, og teste dem med PBT.

```
vectorCom : Matrix -> bool
let vectorCom m =
    sumRows m = sumRows (flip m)

vectorScalarAss : Matrix -> Number -> Number -> bool
let vectorScalarAss (m:Matrix) (n1:Number) (n2:Number) =
    n1 * (n2 * m) = (n1 * n2) * m

vectorAssCom : Matrix -> Number -> bool
let vectorAssCom m (c:Number) =
    c * (sumRows m) = sumRows (c * m)
```

Listing 33: Egenskaberne fra sætning 6 som funktioner

```
- Arb.register<MaxtrixGen>()
- Arb.register<NumberGen>()
- let _ = Check.Quick vectorCom
- let _ = Check.Quick vectorScalarAss
- let _ = Check.Quick vectorAssCom;;
Ok, passed 100 tests.
Ok, passed 100 tests.
Ok, passed 100 tests.
```

Listing 34: Outputtet fra PBT af vektor Listing 33

4.3 Række-echelon form

4.4 Gram-Schmidt

Vi kan nu betragte implementeringen af Gram-Schmidt processen. Denne proces kan anvendes rekursivt til at finde en ortonormal basis for et underrum udspændt af en liste af vektorer $v_1, v_2, \dots, v_n$. Processen kan implementeres rekursivt idet de nye vektorer w_k for $k = 2, 3, \dots, n$ konstrueres baseret på alle de tidligere vektorer $w_1, \dots, w_{k-1}$.

Før vi implementerer Gram-Schmidt processen, er vi dog begrænset af vores Number type 10, idet $x \in \{\text{Number}\} \implies \sqrt{x} \in \{\text{Number}\}$. Derfor vil vi ikke normalisere vektorerne, hvilket medfører, at vi kun vil finde en ortogonal basis, fremfor en ortonormal basis.

[18] theorem 7.2 i mat 1 noterne

```fsharp
// orthogonalBacis : Matrix -> Matrix
let orthogonalBacis m =
    if not <| corectOrderCheck m C
    then orthogonalBacis (correctOrder m C)
    else

    // Gram_Schmidt : Matrix -> (Vector list -> Matrix) -> Matrix
    let rec Gram_Schmidt vm acc_wm =
        match acc_wm [], vm with
        | x, M([], _) -> x
        | M([], _), M(v1::vrest, o) ->
            Gram_Schmidt (M(vrest,o))
            <| fun x -> extendMatrix (M([v1], C)) x
        | M(w, _), M(vk::vrest, o) ->
            let (V(wk, _)) = vk - sumProj w vk
            Gram_Schmidt (M(vrest,o))
            <| fun x -> extendMatrix (acc_wm wk) x

    // sumProj : Vector list -> Vector -> Vector
    and sumProj w vk =
        List.map (fun x -> proj x vk) w
        |> matrix
        |> sumRows
        |> matrixToVector

    Gram_Schmidt m (fun _ -> M([], C))
```

Listing 35: Dannelsen af en ortogonal basis, ved hjælp af Gram-Schmidt processen

Funktionen sumProj tager en liste med vektorer w, som i Gram-Schmidt-processen er de tidligere behandlede vektorer $w_1, \ldots, w_{k-1}$, og en vektor v_k som er den k'te vektor. v_k projiceres på alle vektorerne i w, hvorefter der tages summen af disse projektioner.

Funktionen Gram_Schmidt, tager en matrix hvor søjlerne er de vektores som ønskes at finde en ortogonal basis for. Der udover tager den en akkumulerende funktion som indeholder de behandlede vektorer. Hvis der ikke er flere vektorer i matricen, gives den akkumulerede funktion. Hvis der ikke er nogle vektorer i akkumulatoren, tages den første vektor fra matricen og tilføjes til akkumulatoren. Hvis der er vektorer i både akkumulatoren og matricen, kaldes sumProj på den akkumulerede liste og den første vektor i matricen. Resultatet trækkes fra den første vektor i matricen, og dette bliver den nye vektor som tilføjes til akkumulatoren.

Funktionen orthogonalBacis tager en matrix og tjekker om matricen er i kolonnefølge, hvis ikke kalder funktionen sig selv, med den korrekte lagringsordning. Ellers kaldes Gram_Schmidt med matricen og en tom akkumulator. Resultatet bliver derfor en matrix med en ortogonal basis for underrummet udspændt af de givne vektorer, givet at vektorerne er lineært uafhængige.

4.5 PBT af Gram-Schmidt

Udfrodringen ved at lave en PBT af Gram-Schmidt er at vektorsættet skal være lineært uafhængige. Derfor laves der en generator som ved at udføre tilfældige række operationer på en diagonal matrix, kan generere en matrix med lineært uafhængige vektorer. #TODO : Lav et bevis for det beholder enskaben for lineært uafhængighed.

```fsharp
// getBacismatrixGen : int -> Gen<Matrix>
```

```
let getBacismatrixGen n =
    Gen.map (fun x -> standardBacis x) (Gen.choose (2, n))

// performRowOperationGen : Matrix -> Gen<Matrix>
let performRowOperationGen m =
    let (D(n, _)) = dimMatrix m
    gen {
        let! i = Gen.choose(1, n)
        let! j = match i with
            | 1 -> Gen.choose(2, n)
            | _ when i = n -> Gen.choose(1, n-1)
            | _ -> Gen.oneof [
                    Gen.choose(1, i-1);
                    Gen.choose(i+1, n)]
        let! a = numberGen
        return rowOperation i j a m }


// multipleRowOperationsGen : Matrix -> int -> Gen<Matrix>
let rec multipleRowOperationsGen m count =
    if count <= 0 then Gen.constant m
    else
        gen {
            let! newMatrix = performRowOperationGen m
            return! multipleRowOperationsGen newMatrix (count - 1)
        }

// getIndependetBacisGen : Gen<Matrix>
let getIndependetBacisGen =
    gen {
        let! m = getBacismatrixGen 5
        let! numberOfOperations = Gen.choose(1, 10)
        let! span = multipleRowOperationsGen m numberOfOperations
        return span }

type IndependetBacis = Matrix
type IndependetBacisGen =
    static member IndependetBacis() =
        {new Arbitrary<Matrix>() with
            override _.Generator = getIndependetBacisGen
            override _.Shrinker _ = Seq.empty}
```

Listing 36: Generatorene anvendt til PBT af Gram-Schmidt

Listing 36 viser de forskellige generatorer, som anvendes til PBT (Property-Based Testing) af Gram-Schmidt-processen. Først genereres en tilfældig basis matrix. Dernæst udvælges to tilfældige rækker, i og j, hvorefter der udføres en rækkeoperation på R_j, således at $R_j \leftarrow R_j - aR_i$, hvor a er et tilfældigt Number. Denne proces gentages et tilfældigt antal gange.

Dernæst skal vi bruge en funktion til at tjekke om en matrix er en ortogonal basis. isOrthogonalBacis i Listing 37 tjekker om alle vektorerne i en matrix er ortogonale, ved at tjekke om søjle v_i er ortogonal med v_{i+1}, for alle $i \in [1, n-1]$ hvor n er længden på søjlerne. To søjler er ortogonale hvis deres indreprodukt er 0.

```
// isOrthogonalBacis : Matrix -> bool
let rec isOrthogonalBacis (M(vl, o)) =
    if not <| corectOrderCheck (M(vl, o)) C
```

```
4       then isOrthogonalBacis <| correctOrder (M(vl, o)) C
5       else
6       match vl with
7       | [] -> true
8       | _::[] -> true
9       | v::vnext::vrest -> innerProduct v vnext = zero && isOrthogonalBacis (M(
        vnext::vrest, o))
```

Listing 37: Funktion til at tjekke om søjlerne i en matrix er en ortogonal basis

PB testen gramSchmidtIsOrthogonal bliver derfor blot at tjekke om en matrix bestående af lineært uafhængige vektorer, der udspænder et underrum, er orthogonale efter Gram-Schmidt processen er blevet anvendt. Grundet tilfældige matematiske operationer, opstår der en støre mængde opstå overflow fejl, derfor godtages disse men klassificeres som overflow.

```
1 let gramSchmidtIsOrthogonal (m:IndependetBacis) =
2     let res =
3         try
4             if orthogonalBacis m |> isOrthogonalBacis then 1 else 0
5         with
6             | :? System.OverflowException -> 2
7     (res = 1 || res = 2)
8     |> Prop.classify (res = 1) "PropertyHolds"
9     |> Prop.classify (res = 2) "OverflowException"
```

Listing 38: PBT af Gram-Schmidt processen

```
- Arb.register<IndependetBacisGen>()
- let _ = Check.Quick gramSchmidtIsOrthogonal;;
Ok, passed 100 tests.
69% PropertyHolds.
31% OverflowException.
```

Listing 39: Output fra PBT af Gram-Schmidt processen

Outputtet fra PBT af Gram-Schmidt processen kan ses i Listing 39. Som sædvanligt indikere testen kun korrekthed, men ikke garanteret korrekthed.

5 Appendiks

Hele projektet kan findes på Github, på følgende reference [4]. Udvalgte filer som omtaltes fra projektet er vedlagt i følgende sektioner.

5.1 complex.fsi

5.2 TreeGenerator.fs

5.3 CommutativeAddSub.fs

5.4 CommutativeMulDiv.fs

5.5 Matrix.fs

Litteratur

[1] *Convert Infix expression to Postfix expression.* URL: https://www.geeksforgeeks.org/%20convert-infix-expression-to-postfix-expression/ (hentet 17.02.2024).

[2] *Differentiation Rules.* URL: https://en.wikipedia.org/wiki/Differentiation_rules (hentet 10.05.2024).

[3] *Floating-point error.* URL: https://en.wikipedia.org/wiki/Floating-point_error_mitigation (hentet 30.03.2024).

[4] *Github repository for the project.* URL: https://github.com/Larsen00/funktionsprogrammeringForIndledend

[5] *Greatest common divisor wikipedia.* URL: https://en.wikipedia.org/wiki/Greatest_common_divisor (hentet 30.03.2024).

[6] Michael R. Hansen og Hans Rischel. *Functional Programming Using F#*. Cambridge University Press, 2013. ISBN: 9781107019027, 1107019028.

[7] *Inverse function.* URL: https://en.wikipedia.org/wiki/Inverse_function (hentet 06.05.2024).

[8] Ole Christensen og Jakob Lemvig. *Mathematics 1b - Functions of several variables.* URL: https://01002.compute.dtu.dk/_assets/notesvol2.pdf (hentet 09.02.2024).

[9] *Mathematics 1a.* URL: https://01001.compute.dtu.dk/_assets/enotesvol1.pdf (hentet 09.02.2024).

[10] *Operator Precedence and Associativity in C.* URL: https://www.geeksforgeeks.org/operator-precedence-and-associativity-in-c/ (hentet 17.02.2024).

[11] *Polsk notation wikipedia.* URL: https://en.wikipedia.org/wiki/Polish_notation.

[12] *Python - List Comprehension.* URL: https://www.w3schools.com/python/python_lists_comprehension.asp (hentet 31.03.2024).

[13] *Rational Number wikipedia.* URL: https://en.wikipedia.org/wiki/Rational_number (hentet 30.03.2024).

[14] *Row- and column-major order.* URL: https://en.wikipedia.org/wiki/Row-_and_column-major_order (hentet 31.03.2024).

[15] *Test-Driven Development.* URL: https://en.wikipedia.org/wiki/Test-driven_development (hentet 31.03.2024).

[16] *Tree Traversal Techniques – Data Structure and Algorithm Tutorials.* URL: https://www.geeksforgeeks.org/tree-traversals-inorder-preorder-and-postorder/ (hentet 30.03.2024).