

DANMARKS TEKNISKE UNIVERSITET



Bachelor Projekt

FUNKTIONEL MODELLERING AF MATEMATISKE SYSTEMER I F#

Jonas Dahl Larsen (s205829)

9. maj 2024

Indhold

1	Introduktion	2
2	Fundamentale koncepter	3
2.1	Introduktion til Funktions Programmering	3
2.1.1	Typer	4
2.2	Signatur filer og implementerings filer	4
2.3	Overloading af operatorer	4
2.4	Property Based Testing	5
3	Symbolske udtryk	5
3.1	Tal mængder	6
3.1.1	Rationelle tal Mondul	6
3.1.2	Komplekse tal Mondul	7
3.1.3	Tal Mondulet	7
3.2	Matematiske udtryk	9
3.2.1	Polsk notation	9
3.2.2	Udtryk som træer	9
3.2.3	Udtryksmodulet	10
3.3	Evaluering af udtryk	12
3.3.1	Konvertering mellem udtryks notation	13
3.4	Simplifikation af udtryk	14
3.5	differentiering af udtryk	17
3.6	PBT af udtryk	17
3.6.1	Homomorfisme af evaluering	17
3.6.2	Invers morphism mellem infix og prefix	17
3.6.3	Simplifikation af udtryk	17
4	Vektorer og Matricer	17
4.1	Matrix operationer	17
4.1.1	Matematiske operationer	18
4.1.1.1	Skalering af en matrix	18
4.1.1.2	Addition af matricer	19
4.1.1.3	Matrix multiplikation	19
4.1.1.4	Projektion af en vektor	21
4.2	PBT af matrix operationer	22
4.3	Række-echelon form	23
4.4	Gram-Schmidt	23
4.5	PBT af Gram-Schmidt	24
5	Appendiks	26
5.1	TreeGenerator.fs	26
5.2	CommutativeAddSub.fs	26
5.3	CommutativeMulDiv.fs	26
5.4	Matrix.fs	26

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.

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

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 operatører.

```
> 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 typer¹:

$$f : T_1 \rightarrow T_2$$

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

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 nødt til 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.

¹Functional 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.

$$P \wedge (Q \wedge R) \iff (P \wedge Q) \wedge R \quad (2)$$

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).

```
1 #r "nuget: FsCheck"
2 open FsCheck
3
4 // proposition formula: bool -> bool -> bool -> bool
5 let propositional_formula P Q R =
6     (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

²Test-Driven Development.

andre programmer har F# kun float og int som kan repræsentere tal. Derfor vil vi begynde med at definere et mondul 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 mondul som kan repræsentere tal mængder. Typen for tal, består af tre konstruktører, for henholdsvis heltal, rationale tal og komplekse tal. Dog er mondulet lavet med henblik på at kunne udvides med flere typer af tal. Måden resten af programmet er lavet på, gør de eneste krav til tal er at der er definerede matematiske operationer i form af addition, subtraktion, negation, multiplikation og division. Samt at tallet inden for addition og multiplikation er associative. Dette gælder blandt andet ikke for en vektor, derfor vil vi senere betragte at udvide programmet med en type for vektorer. En udvidelse kunne være for reelle tal, som kan håndtere "floating point errors"³, men for ikke at komplicere programmet vil vi i denne opgave ikke betragte floats.

3.1.1 Rationelle tal Mondul

Repræsentationen af rationale tal kan laves ved hjælp af danne et par af integers, hvor den ene integer er tælleren og den anden er nævneren.

```
1 [language={FSharp},
2   label={type_rational},
3   caption={Typen for rationelle tal}]
4 type rational = R of int * int
```

Nedestående er der givet en signatur fil for rational mondulet 6. i Implementerings filen overloades de matematiske operatorer, ved hjælp af de klassiske regneregler for brøker⁴.

```
1 module rational
2
3 [

```

³Floating-point error.

⁴Rational Number wikipedia.

```

20 val positive      : rational -> bool
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 6: Signatur filen for rational mondulet

For at kunne sammenligne, men også for nemmere at undgå for store brøker, vil alle rationelle tal blive reduceret til deres simpleste form. Dette kan gøres ved at finde den største fælles divisor (GCD)⁵. Der udover er det vigtigt at være opmærksom på man ikke foretager nul division. Derfor vil implementerings filen kaste en "System.DivideByZeroException" hvis nævneren er eller bliver nul. Signatur filen indeholder en række funktioner som bliver anvendt af andre filer.

3.1.2 Komplekse tal Mondul

3.1.3 Tal Mondulet

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. Vi begynder med at definere en type for tal 7, som indeholder konstruktører for de tal typer vi har definerede samt en for heltal.

```

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

Listing 7: Typen for Number

Betragtes signatur filen for Number mondulet 8, 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 rationalAndComplex
3 open rational
4 open complex
5
6 type Number =
7     | Int of int
8     | Rational of rational
9     | Complex of complex
10 with
11     static member ( + ) : Number * Number -> Number
12     static member ( - ) : Number * Number -> Number
13     static member ( * ) : Number * Number -> Number
14     static member ( / ) : Number * Number -> Number
15     static member ( ~- ) : Number -> Number
16
    
```

⁵Greatest common divisor wikipedia.


```

17
18
19
20 val zero      : Number
21 val one       : Number
22 val two       : Number
23 val isZero    : Number -> bool
24 val isOne     : Number -> bool
25 val isNegative : Number -> bool
26 val absNumber : Number -> Number
27 val greaterThan : Number -> Number -> bool
28 val tryReduce  : Number -> Number
29 val toString   : Number -> string
30 val conjugate  : Number -> Number
31 val inv        : Number -> Number
32 val isInt      : Number -> bool
    
```

Listing 8: Signatur filen for Number mondulet

Ved implementeringen af de matematiske operationer, hvis der eksistere en konstruktør i Number, der repræsenterer en tal mængde hvor alle andre konstruktører er delmængder af denne mængde. Er det muligt at definere en enkelt funktion som kan udføre alle binære operationer. Som et eksempel er funktionen 9 givet, som tager to tal og en funktion i form af den ønskede binære 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 9: Number.operation funktionen

Det vil her til være oplagt på alle de matematiske operationer at anvende en funktionen til at forsøge at konvertere 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 10.

```

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 10: Overloadnings funktionerne for Number

Dermed har vi et monad 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

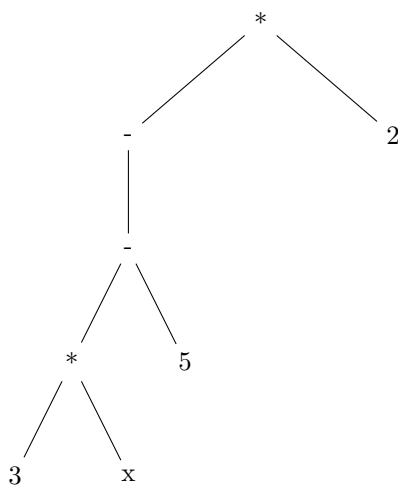
3.2.1 Polsk notation

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 generelt 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 regler⁶.

Infix Notation: $(A + B) \cdot C$
 Prefix Notation: $\cdot + ABC$
 Postfix Notation: $AB + C \cdot$

3.2.2 Udtryk som træer

Et matematisk udtryk kan repræsenteres som et binært træ, hvor bladene er operandere 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.

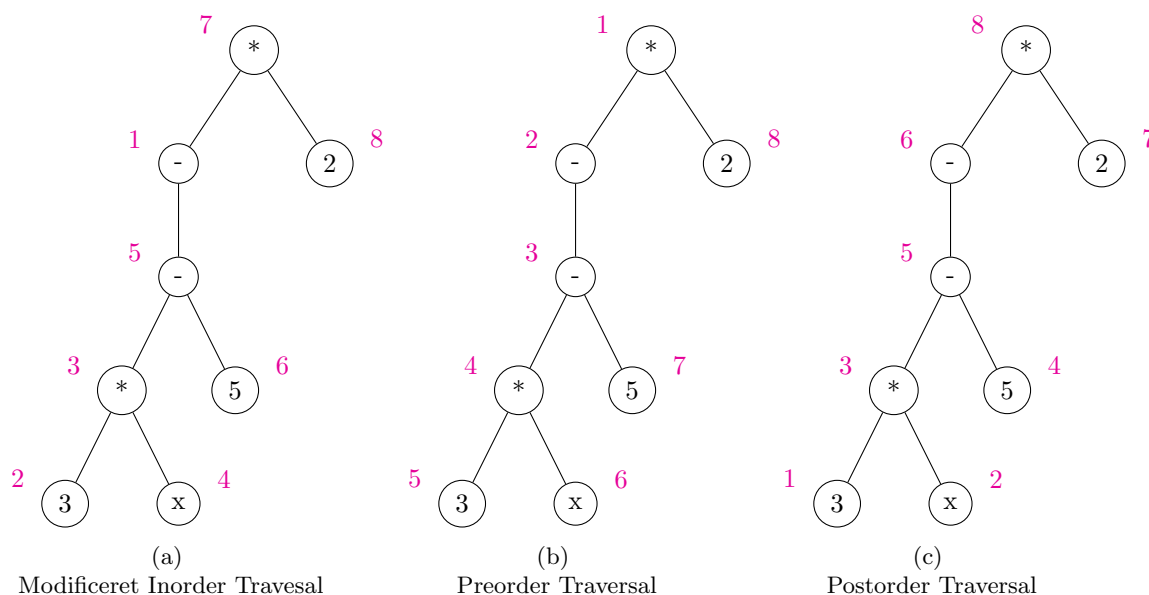


Figur 1: Et binært træ der repræsenterer udtrykket $-(3 \cdot x - 5) \cdot 2$

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

⁶Polsk notation wikipedia.

være muligt omdanne dem til infix, prefix eller postfix notation. Dette kan gøres ved at anvende modificeret Inorder, Preorder eller Postorder Traversal⁷, algorithmerne er illustreret i Figur 2.



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 traversal 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 11. 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 7. Til sidst

⁷Tree Traversal Techniques – Data Structure and Algorithm Tutorials.

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.

```

1 type Expr<'a> =
2   | X of char
3   | N of 'a
4   | Neg of Expr<'a>
5   | Add of Expr<'a> * Expr<'a>
6   | Sub of Expr<'a> * Expr<'a>
7   | Mul of Expr<'a> * Expr<'a>
8   | Div of Expr<'a> * Expr<'a>
    
```

Listing 11: 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 12. 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 12: $-(3 \cdot x - 5) \cdot 2$ 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.

```

1 module Expression
2 open Number
3
4 type Expr<'a> =
5   | X of char
6   | N of 'a
7   | Neg of Expr<'a>
8   | Add of Expr<'a> * Expr<'a>
9   | Sub of Expr<'a> * Expr<'a>
10  | Mul of Expr<'a> * Expr<'a>
11  | Div of Expr<'a> * Expr<'a>
12 with
13 static member ( ~- ) : Expr<Number> -> Expr<Number>
14 static member ( + ) : Expr<Number> * Expr<Number> -> Expr<Number>
15 static member ( - ) : Expr<Number> * Expr<Number> -> Expr<Number>
16 static member ( * ) : Expr<Number> * Expr<Number> -> Expr<Number>
17 static member ( / ) : Expr<Number> * Expr<Number> -> Expr<Number>
18
19 val eval : Expr<Number> -> Map<char,Number> -> Number
20 val containsX : Expr<Number> -> Expr<Number> -> bool
21 val getNumber : Expr<Number> -> Number
22 val getVariable : Expr<Number> -> char
23 val isAdd : (Expr<Number> -> Expr<Number> -> Expr<Number>) -> bool
24 val isSub : (Expr<Number> -> Expr<Number> -> Expr<Number>) -> bool
25 val isMul : (Expr<Number> -> Expr<Number> -> Expr<Number>) -> bool
26 val isDiv : (Expr<Number> -> Expr<Number> -> Expr<Number>) -> bool
27 val isNeg : (Expr<Number> -> Expr<Number>) -> bool
    
```

Listing 13: Signatur filen for Expression mondulet

De overloadede matematiske operatører i Expressions, laver overflade evalueringer samt simplifikationer på deres respektive argumenter. Overfalde evaluering vil sige at de individuelle funktioner kun betragter de to øverste niveauer på de udtryks træer de tager som input, mulige implementeringer af addition og multiplikation er givet i Listing 14.

```

1 // add: Expr<Number> -> Expr<Number> -> Expr<Number>
2 let rec add e1 e2:Expr<Number> =
3     match e1, e2 with
4     | N a, N b                                -> N (a + b)
5     | N a, b | b, N a when isZero a          -> b
6     | Mul(a, X b), Mul(c, X d)
7     | Mul(X b, a), Mul(c, X d)
8     | Mul(a, X b), Mul(X d, c)
9     | Mul(X b, a), Mul(X d, c) when b = d -> Mul(add a c, X b)
10    | _, _                                    -> Add(e1, e2)
11
12 // mul: Expr<Number> -> Expr<Number> -> Expr<Number>
13 let mul e1 e2:Expr<Number> =
14     match e1, e2 with
15     | N a, N b                                -> N (a * b)
16     | N a, b | b, N a when isOne a          -> b
17     | N a, _ | _, N a when isZero a        -> N zero
18     | _, _                                    -> Mul(e1, e2)
    
```

Listing 14: 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 homomorfe egenskaber 1. Egenskaben vil blive testet i sektion 3.6.1.

Egenskab 1 (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 15 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 mondulet.

```

1 // eval: Expr<Number> -> Map<char, Number> -> Number
2 let rec eval (e:Expr<Number>) (env) =
3     match e with
4     | X x -> Map.find x env
5     | N n -> n
6     | Neg a -> - eval a env
7     | Add (a, b) -> eval a env + eval b env
8     | Sub (a, b) -> eval a env - eval b env
    
```

```

9 | Mul (a, b) -> eval a env * eval b env
10 | Div (a, b) -> eval a env / eval b env

```

Listing 15: Evaluering af et udtryk

3.3.1 Konvertering mellem udtryks notation

Det er ønsket 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 2. Egenskaben bliver test i sektion 3.6.2.

Egenskab 2 (Invers morphism⁸ mellem infix og prefix).

Lad Q^n være mængden af rationelle infix udtryk repræsenteret som en string, med n variable, så defineres følgende:

$$\begin{aligned} \text{tree} : Q^n &\rightarrow \text{Expr} \\ \text{tree}^{-1} : \text{Expr} &\rightarrow Q^n \end{aligned}$$

Dermed gælder følgende egenskaber

$$\begin{aligned} \text{tree}^{-1} \circ \text{tree} &= \text{id}_{Q^n} \\ \text{tree} \circ \text{tree}^{-1} &= \text{id}_{\text{Expr}} \end{aligned}$$

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 16 fortager denne konvertering 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.

```

1 // parenthesis: bool -> string -> string
2 let parenthesis b f = if b then "(" + f + ")" else f
3
4 // etf: Expr<Number> -> bool -> string
5 let rec etf e p =
6     match e with
7     | N a when not <| isInt a -> parenthesis p <| toString a
8     | N a -> toString a
9     | X a -> string a
10    | Neg a -> parenthesis p <| "-" + etf a (not p)
11    | Add(a, b) -> parenthesis p <| etf a false + "+" + etf b false
12    | Sub(a, b) -> parenthesis p <| etf a false + "-" + etf b true
13    | Mul(a, b) -> parenthesis p <| etf a true + "*" + etf b true
14    | Div(a, b) -> parenthesis p <| etf a true + "/" + etf b true
15
16
17 // infixExpression: Expr<Number> -> string

```

⁸Inverse function.

```
18 let infixExpression e = etf e false
```

Listing 16: konvertering fra expression til infix notation

Funktionen `tree`, som foretager konverteringen fra infix notation til et udtrykstræ, er baseret på algoritmen beskrevet i [1]⁹. Først konverteres en udtryksstreng til en liste af tokens. Disse tokens beskriver, om en karakter i udtrykket er en operand, en operator, eller en konstant, hvor en operator også indeholder information om præcedens og associativitet¹⁰. Typen for disse tokens kan ses i Listing 17. 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 5.1.

```
1 type Associative = | Left | Right
2 type Precedence = int
3 type Operator = char * Precedence * Associative
4 type Token =
5     | Operand of char
6     | Operator of Operator
7     | Constant of int
8 type OperatorList = Operator list
```

Listing 17: Konvertering fra infix til udtrykstræ

3.4 Simplifikation af udtryk

Vi skal nu betragte en systematisk 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 udtrykkene visualiseres. Før vi betragter metoden, kan vi opskrive en egenskab som simplification skal overholde. Egenskaben vil blive testet i sektion 3.6.3, 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 3 (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 18, 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.

```
1 //simplifyOperation: Expr<Number> -> Expr<Number> -> (Expr<Number> -> Expr<
    Number> -> Expr<Number>) -> Expr<Number>
2 let rec simplifyOperation e1 e2 f =
```

⁹ Convert Infix expression to Postfix expression.

¹⁰ Operator Precedence and Associativity in C.

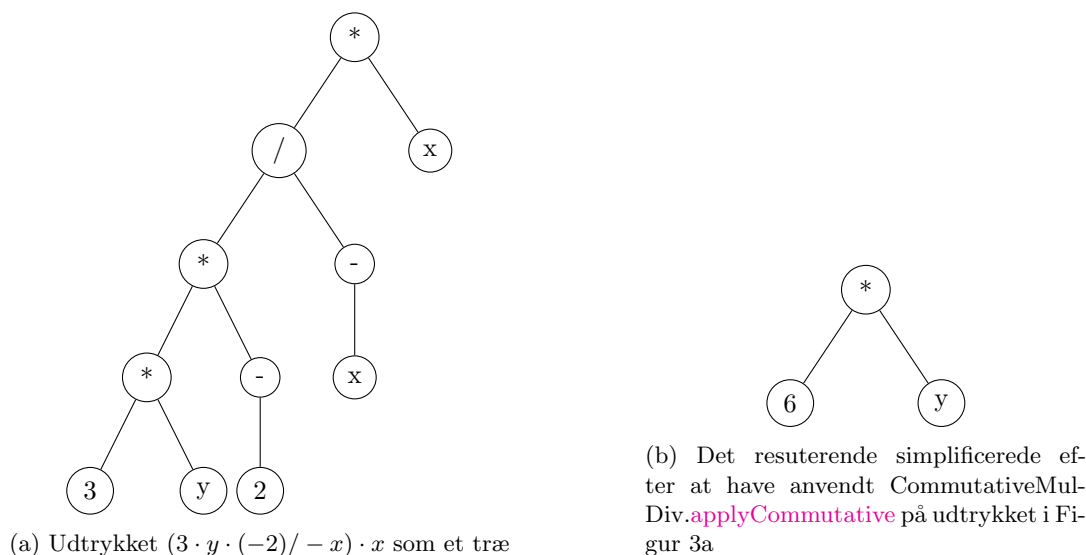
```

3  match f e1 e2 with
4  | Neg a ->
5      commutativeMulDiv.applyCommutative (Neg a) |> commutativeAddSub.
        applyCommutative
6  | Add(a, b) when isAdd f -> commutativeAddSub.applyCommutative (Add(a, b))
7  | Sub(a, b) when isSub f -> commutativeAddSub.applyCommutative (Sub(a, b))
8  | Mul(a, b) when isMul f -> commutativeMulDiv.applyCommutative (Mul(a, b))
9  | Div(a, b) when isDiv f -> commutativeMulDiv.applyCommutative (Div(a, b))
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         (/)
24     | Neg a -> - (simplifyExpr a)
25     | Add(a, b) -> simplifyOperation (simplifyExpr a) (simplifyExpr b) (+)
26     | Sub(a, b) -> simplifyOperation (simplifyExpr a) (simplifyExpr b) (-)
27     | Mul(a, b) -> simplifyOperation (simplifyExpr a) (simplifyExpr b) (*)
28     | Div(a, b) -> simplifyOperation (simplifyExpr a) (simplifyExpr b) (/)
29     | _ -> e

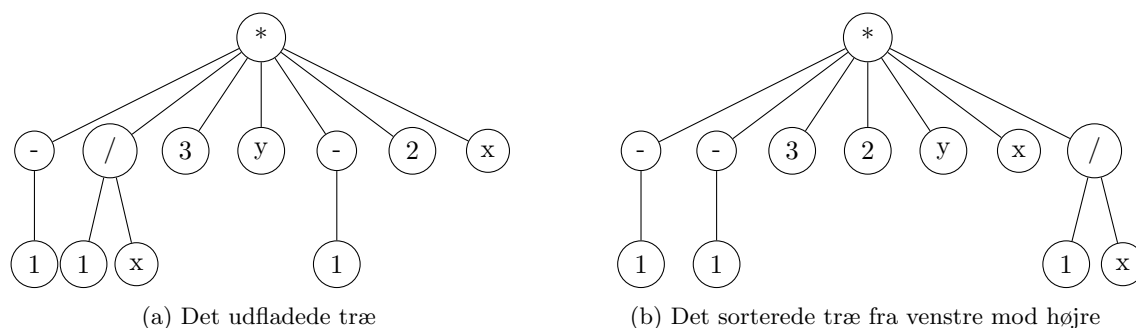
```

Listing 18: 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 Appendix 5.2 og 5.3. 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 udfledede træ, som fremtræder i nævneren af en division og i det udfledede 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 udfledede træ og sorteringen af det fladede træ.



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



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

3.6 PBT af udtryk

3.6.1 Homomorfisme af evaluering

3.6.2 Invers morphism mellem infix og prefix

3.6.3 Simplifikation 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 omtales begge kun referere til en matrix. For at kunne håndtere matricer på en systematisk måde begynder vi med at definere 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 19: Typen for order

Typen Order (se Listing 19), anvendes til at angive, om en matrix er i rækkefølge (row-major) eller kolonnefølge (column-major)¹¹. 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 20).

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

Listing 20: Typen for Matricer

Derudover er det en fordel at kunne kende dimensionen af en matrix. Derfor er der også defineret en type for dimensionen (se Listing 21).

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

Listing 21: Typen for dimensionen

.-

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.

¹¹Row- and column-major order.

Det muligt at definere en funktion til at finde dimensionen af en matrix (se Listing 22). 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.

```

1 // dimMatrix : Matrix -> Dimension
2 let dimMatrix (M(v1, o)) =
3     if v1 = [] then D (0, 0)
4     else
5         let _ = matrixValidMajor (M(v1, o))
6         let d1 = List.length v1
7         let d2 = matrixVectorLength (M(v1, o))
8         match o with
9         | R -> D (d1, d2)
10        | C -> D (d2, d1)
    
```

Listing 22: Funktion til at finde dimensionen 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 5.4, 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 23).

```

1 // scalarVector : Number -> Vector -> Vector
2 let scalarVector (n:Number) (V (n1, o)) =
3     V ((List.map (fun x -> x * n) n1), o)
4
5 // scalarMatrix : Matrix -> Number -> Matrix
6 let scalarMatrix (M (v1, o)) n =
7     M ((List.map (fun x -> scalarVector n x) v1), o)
    
```

Listing 23: 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 Python¹².

```

1 // addVector : Vector -> Vector -> Vector
2 let addVector (V (v1, o1)) (V (v2, _)) =
3     V ((List.map2 (+) v1 v2), o1)
4
5 // addMatrix : Matrix -> Matrix -> Matrix
6 let addMatrix (M(v11, o)) (M(v12, _)) =
7     M (List.map2 addVector v11 v12, o)
8
9 // subVector : Vector -> Vector -> Vector
    
```

¹²Python - List Comprehension.

```

10 let subVector x y =
11     scalarVector (-one) y |> addVector x
12
13 // sumRows : Matrix -> Matrix
14 let rec sumRows m =
15     if not <| correctOrderCheck m C
16     then sumRows <| correctOrder m C
17     else
18         let zeroVector = vectorOf zero <| matrixVectorLength m
19         let (M(v1, _)) = m
20         matrix [List.fold (addVector) zeroVector v1]
    
```

Listing 24: Funktion til at addere matricer og subtraktion af vektorer

4.1.1.2 Addition af matricer

Addition af vektorer kommer ned til at fortage additionen elementvis i Listing 24, 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 definition 1.

Definition 1 (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

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

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

4.1.1.3 Matrix multiplikation

Definition 2 (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 $\mathbf{v} = (v_1, v_2, \dots, v_n)^T$ være en vektor med n elementer. Så er matrix-vektor $A\mathbf{v}$ 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

$$A\mathbf{v} = \text{sumRows} \left[\begin{array}{c|c|c|c} & & & \\ \hline a_1v_1 & a_2v_2 & \cdots & a_nv_n \\ \hline & & & \end{array} \right]$$

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

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

Ved brug af definition 1 for `sumRows` og definition 2 ses det at

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

□

Vi kan dermed anvende Sætning 1 til at definere en funktion `matrixMulVector` for matrix-vektor multiplikation (se Listing 25). Som først skalere søjlerne i matrixen med de tilsvarende element i vektoren, og derefter summe rækkerne i matrixen.

```
1 // matrixMulVector : Matrix -> Vector -> Matrix
2 let rec matrixMulVector (M(v1, _)) (V(n1, _)) =
3     M (List.map2 (fun mc n -> scalarVector n mc) v1 n1, C)
4     |> sumRows
```

Listing 25: Funktion til matrix-vektor multiplikation

Definition 3 (Matrix multiplikation¹³).

Lad $A \in \mathbb{F}^{m \times n}$ og $B \in \mathbb{F}^{n \times \ell}$. Antag at søjlerne i B er givet ved $b_1, \dots, b_\ell \in \mathbb{F}^n$, dermed

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

Så defineres matrixe produktet som

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

Udfra definition 3, ses det at funktion `matrixProduct` i Listing 26 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.

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

Listing 26: 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'¹⁴, kan projektionen af en vektor defineres som følgende, hvor $Y = \text{span}\{y\}$.

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

Med det standard indreprodukt

$$\langle x, y \rangle = y^* x = \sum_{k=1}^n x_k \bar{y}_k \quad (4)$$

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 multiplicere to vektorer element vis `vectorMulElementWise`.

```
1 // vectorMulElementWise : Vector -> Vector -> Vector
2 let vectorMulElementWise (V(u, o1)) (V(v, o2)) =
3     V (List.map2 (*) u v, o1)
4
5 // conjugateVector : Vector -> Vector
```

¹³Mathematics 1a.

¹⁴Mathematics 1b - Functions of several variables.

```

6 let conjugateVector (V(v, o)) =
7     V (List.map conjugate v, o)
8
9 // innerProduct : Vector -> Vector -> Number
10 let innerProduct u v =
11     let (V(w, _)) = conjugateVector v |> vectorMulElementWise u
12     List.fold (+) zero w
13
14 // proj : Vector -> Vector -> Vector
15 let proj y x =
16     scalarVector (innerProduct x y / innerProduct y y) y

```

Listing 27: Funktioner til projekte 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. Hvertil 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 4. Først defineres en generator for matricer, som generere matricer med tilfældige tal fra vores talmængde 7.

```

1 // vectorGen : int -> Gen<Vector>
2 let vectorGen n =
3     Gen.listOfLength n numberGen |> Gen.map (fun x -> vector x)
4
5 // matrixGen : Gen<Matrix>
6 let matrixGen =
7     gen {
8         let! row = Gen.choose(1, 6)
9         let! col = Gen.choose(1, 6)
10        let! vectors = Gen.listOfLength col (vectorGen row)
11        return matrix vectors
12    }
13
14 type MaxtrixGen =
15     static member Matrix() =
16         {new Arbitrary<Matrix>() with
17             override _.Generator = matrixGen
18             override _.Shrinker _ = Seq.empty}
19
20 type NumberGen =
21     static member Number() =
22         {new Arbitrary<Number>() with
23             override _.Generator = numberGen
24             override _.Shrinker _ = Seq.empty}

```

Listing 28: Generatorene anvendt til PBT af matrix operationer i

Egenskab 4 (Vektor Aksiomer¹⁵).

Lad $c, d \in \mathbb{F}$ og $v_i \in \mathbb{F}^n$ for $i = 1 \dots m$ så gælder:

¹⁵theorem 7.2 i mat 1 noterne

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

Vi kan dermed nu lave definere egenskaberne fra sætning 4 som nogle funktioner, og teste dem med PBT.

```

1 vectorCom : Matrix -> bool
2 let vectorCom m =
3     sumRows m = sumRows (flip m)
4
5 vectorScalarAss : Matrix -> Number -> Number -> bool
6 let vectorScalarAss (m:Matrix) (n1:Number) (n2:Number) =
7     n1 * (n2 * m) = (n1 * n2) * m
8
9 vectorAssCom : Matrix -> Number -> bool
10 let vectorAssCom m (c:Number) =
11     c * (sumRows m) = sumRows (c * m)

```

Listing 29: Egenskaberne fra sætning 4 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 30: Outputtet fra PBT af vektor Listing 29

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 7, idet $x \in \{\text{Number}\} \not\Rightarrow \sqrt{x} \in \{\text{Number}\}$. Derfor vil vi ikke normalisere vektorerne, hvilket medfører, at vi kun vil finde en ortogonal basis, fremfor en ortonormal basis.

```

1 // orthogonalBasis : Matrix -> Matrix
2 let orthogonalBasis m =
3     if not <| correctOrderCheck m C
4     then orthogonalBasis (correctOrder m C)
5     else

```



```

6
7 // Gram_Schmidt : Matrix -> (Vector list -> Matrix) -> Matrix
8 let rec Gram_Schmidt vm acc_wm =
9     match acc_wm [], vm with
10    | x, M([], _) -> x
11    | M([], _), M(v1::vrest, o) ->
12        Gram_Schmidt (M(vrest,o))
13        <| fun x -> extendMatrix (M([v1], C)) x
14    | M(w, _), M(vk::vrest, o) ->
15        let (V(wk, _)) = vk - sumProj w vk
16        Gram_Schmidt (M(vrest,o))
17        <| fun x -> extendMatrix (acc_wm wk) x
18
19 // sumProj : Vector list -> Vector -> Vector
20 and sumProj w vk =
21     List.map (fun x -> proj x vk) w
22     |> matrix
23     |> sumRows
24     |> matrixToVector
25
26 Gram_Schmidt m (fun _ -> M([], C))
    
```

Listing 31: 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, \dots, 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 vektorens 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 matrixen, gives den akkumulerede funktion. Hvis der ikke er nogle vektorer i akkumulatoren, tages den første vektor fra matrixen og tilføjes til akkumulatoren. Hvis der er vektorer i både akkumulatoren og matrixen, kaldes `sumProj` på den akkumulerede liste og den første vektor i matrixen. Resultatet trækkes fra den første vektor i matrixen, og dette bliver den nye vektor som tilføjes til akkumulatoren.

Funktionen `orthogonalBasis` tager en matrix og tjekker om matrixen er i kolonnefølge, hvis ikke kalder funktionen sig selv, med den korrekte lagringsordning. Ellers kaldes `Gram.Schmidt` med matrixen 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

Udfordringen 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.

```

1 // getBacismatrixGen : int -> Gen<Matrix>
2 let getBacismatrixGen n =
3     Gen.map (fun x -> standardBasis x) (Gen.choose (2, n))
4
5 // performRowOperationGen : Matrix -> Gen<Matrix>
6 let performRowOperationGen m =
    
```

```

7   let (D(n, _)) = dimMatrix m
8   gen {
9       let! i = Gen.choose(1, n)
10      let! j = match i with
11          | 1 -> Gen.choose(2, n)
12          | _ when i = n -> Gen.choose(1, n-1)
13          | _ -> Gen.oneof [
14              Gen.choose(1, i-1);
15              Gen.choose(i+1, n)]
16      let! a = numberGen
17      return rowOperation i j a m }
18
19
20 // multipleRowOperationsGen : Matrix -> int -> Gen<Matrix>
21 let rec multipleRowOperationsGen m count =
22     if count <= 0 then Gen.constant m
23     else
24         gen {
25             let! newMatrix = performRowOperationGen m
26             return! multipleRowOperationsGen newMatrix (count - 1)
27         }
28
29 // getIndependetBacisGen : Gen<Matrix>
30 let getIndependetBacisGen =
31     gen {
32         let! m = getBacismatrixGen 5
33         let! numberOfOperations = Gen.choose(1, 10)
34         let! span = multipleRowOperationsGen m numberOfOperations
35         return span }
36
37 type IndependetBacis = Matrix
38 type IndependetBacisGen =
39     static member IndependetBacis() =
40         {new Arbitrary<Matrix>() with
41             override _.Generator = getIndependetBacisGen
42             override _.Shrinker _ = Seq.empty}

```

Listing 32: Generatorene anvendt til PBT af Gram-Schmidt

Listing 32 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. `isOrthogonalBasis` i Listing 33 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.

```

1 // isOrthogonalBasis : Matrix -> bool
2 let rec isOrthogonalBasis (M(vl, o)) =
3     if not <| correctOrderCheck (M(vl, o)) C
4     then isOrthogonalBasis <| correctOrder (M(vl, o)) C
5     else
6         match vl with
7         | [] -> true
8         | _::[] -> true

```

```

9 | v::vnext::vrest -> innerProduct v vnext = zero && isOrthogonalBasis (M(
  vnext::vrest, o))

```

Listing 33: 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 større mængde opstå overflow fejl, derfor godtages disse men klassificeres som overflow.

```

1 let gramSchmidtIsOrthogonal (m:IndependetBasis) =
2   let res =
3     try
4       if orthogonalBasis m |> isOrthogonalBasis 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 34: PBT af Gram-Schmidt processen

```

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

```

Listing 35: Output fra PBT af Gram-Schmidt processen

Outputtet fra PBT af Gram-Schmidt processen kan ses i Listing 35. Som sædvanligt indikere testen kun korrekthed, men ikke garanteret korrekthed.

5 Appendiks

Hele projektet kan findes på Github, på følgende reference [3]. Udvalgte filer som omtales fra projektet er vedlagt i følgende sektioner.

5.1 TreeGenerator.fs

5.2 CommutativeAddSub.fs

5.3 CommutativeMulDiv.fs

5.4 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] *Floating-point error*. URL: https://en.wikipedia.org/wiki/Floating-point_error_mitigation (hentet 30.03.2024).
- [3] *Github repository for the project*. URL: <https://github.com/Larsen00/funktionsprogrammeringForIndledend>
- [4] *Greatest common divisor wikipedia*. URL: https://en.wikipedia.org/wiki/Greatest_common_divisor (hentet 30.03.2024).
- [5] Michael R. Hansen og Hans Rischel. *Functional Programming Using F#*. Cambridge University Press, 2013. ISBN: 9781107019027, 1107019028.
- [6] *Inverse function*. URL: https://en.wikipedia.org/wiki/Inverse_function (hentet 06.05.2024).
- [7] Ole Christensen og Jakob Lemvig. *Mathematics 1b - Functions of several variables*. URL: https://01002.compute.dtu.dk/_assets/notesvol2.pdf (hentet 09.02.2024).
- [8] *Mathematics 1a*. URL: https://01001.compute.dtu.dk/_assets/enotesvol1.pdf (hentet 09.02.2024).
- [9] *Operator Precedence and Associativity in C*. URL: <https://www.geeksforgeeks.org/operator-precedence-and-associativity-in-c/> (hentet 17.02.2024).
- [10] *Polsk notation wikipedia*. URL: https://en.wikipedia.org/wiki/Polish_notation.
- [11] *Python - List Comprehension*. URL: https://www.w3schools.com/python/python_lists_comprehension.asp (hentet 31.03.2024).
- [12] *Rational Number wikipedia*. URL: https://en.wikipedia.org/wiki/Rational_number (hentet 30.03.2024).
- [13] *Row- and column-major order*. URL: https://en.wikipedia.org/wiki/Row-_and_column-major_order (hentet 31.03.2024).
- [14] *Test-Driven Development*. URL: https://en.wikipedia.org/wiki/Test-driven_development (hentet 31.03.2024).
- [15] *Tree Traversal Techniques – Data Structure and Algorithm Tutorials*. URL: <https://www.geeksforgeeks.org/tree-traversals-inorder-preorder-and-postorder/> (hentet 30.03.2024).