

DANMARKS TEKNISKE UNIVERSITET



Bachelor Projekt

FUNKTIONEL MODELLERING AF MATEMATISKE SYSTEMER I F#

Jonas Dahl Larsen (s205829)

15. maj 2024

Indhold

1	Introduktion	3
2	Fundamentale koncepter	4
2.1	Introduktion til Funktions Programmering	4
2.2	Typer	5
2.3	Signatur filer og implementerings filer	5
2.4	Overskrivning af operatorer	6
2.5	Property Based Testing	6
3	Symbolske udtryk	7
3.1	Tal mængder	7
3.1.1	Rationelle tal modul	7
3.1.2	Komplekse tal-modul	8
3.1.3	Tal modulet	9
3.2	Matematiske udtryk	12
3.2.1	Polsk notation	12
3.2.2	Udtryk som træer	12
3.2.3	Udtryksmodulet	13
3.3	Evaluering af udtryk	15
3.4	Konvertering mellem udtryks notation	16
3.5	Simplifikation af udtryk	17
3.6	Differentiering af udtryk	20
3.7	Multivariable polynomier af første grad	20
4	Vektorer og Matricer	22
4.1	Matrix operationer	22
4.2	Matematiske operationer	23
4.2.1	Skalering af en matrix	23
4.2.2	Addition af matricer	23
4.2.3	Matrix produkt	24
4.2.4	Projektion af en vektor	26
4.3	Gram-Schmidt	27
4.4	Række-echelon form	28
4.5	Lineært ligningssystem	29
5	PBT af programmet	31
5.1	PBT af udtryk	31
5.1.1	Tal modulet	33
5.1.2	Homomorfisme af evaluering	35
5.1.3	Invers morphism mellem infix og prefix	35
5.1.4	Simplifikation af udtryk	35
5.1.5	Differentiering af udtryk	35
5.2	PBT af vektorer og matricer	35
5.2.1	PBT af matrix operationer	35
5.2.2	PBT af Gram-Schmidt	36

6	Diskussion	38
7	Konklusion	38
8	Appendiks	40
8.1	complex.fsi	40
8.2	TreeGenerator.fs	40
8.3	CommutativeAddSub.fs	42
8.4	CommutativeMulDiv.fs	44
8.5	Matrix.fs	46

1 Introduktion

Dette projekt fokuserer på funktional 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 fordele og ulemper ved funktional programmering i matematiske sammenhænge. I 2023 valgte Danmarks Tekniske Universitet (DTU) at anvende Python som et hjælpeværktøj i deres grundlæggende matematik kursuser „01001 Matematik 1a“ og „01002 Matematik 1b“. Imidlertid åbner funktional programmering op for et andet perspektiv og andre metoder, som kan berige og muligvis forbedre forståelsen af matematiske koncepter hos studerende.

Projektet har til formål at demonstrere, hvordan funktional 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 fra de grundlæggende matematik kurser på DTU.

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 funktional 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 funktional 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. Vi begynder derfor med at betragte funktionen for fakultet Ligning (1).

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

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# ved udregning af $(3!)! = 6! = 720$.

2.2 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 $Factorial : int \rightarrow int$. Det ses dermed tydeligt hvordan f# benytter notationen for afbildninger i matematik, da den matematiske funktion for faktuel er en afbildning $f : \mathbb{Z} \rightarrow \mathbb{Z}$. 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:

$$f : T_1 * T_2 * \dots * T_n \rightarrow T_{n+1}$$

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

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 . Som eksempel på dette kan vi definere en multivariable funktion $f(x, y) = \sqrt{x! + y!}$ som er en afbildning $f : \mathbb{Z}^2 \rightarrow \mathbb{R}$ samt $g(y) = f(3, y) = \sqrt{3! + y!}$ som afbilder $g : \mathbb{Z} \rightarrow \mathbb{R}$, de tilsvarende F# funktion kan defineres på følgende måder:

```
1 // f: int -> int -> float
2 let f x y = sqrt <| float(factorial x + factorial y)
3
4 // g: int -> float
5 let g = f 3
```

Listing 4: Eksempel typerne for en multivariable funktion i F#

2.3 Signatur filer og implementerings filer

En standard F# fil er lavet med .fs extension, denne fil indeholder alt den kode som er nødvendigt for at kunne køre programmet. En implementeringsfil kan have en signaturfil med .fsi extension, denne fil indeholder en beskrivelse af de typer og funktioner i implementeringsfilen som er tilgængelige for andre filer. En signaturfil kan derfor bruges som et blueprint for andre der ønsker at anvende eller replicere implementeringsfilen. I andre programmerings sprog vil man anse funktionerne i signaturfilen, som værende de funktioner der er tilgængelige ved åbning af modulet.

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

2.4 Overskrivning 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.

2.5 Property Based Testing

Property Based Test (PBT) er en teknik til at teste korrektheden 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 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 „01001 Matematik 1a“ 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 5.

```
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 5: 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 6: Output ved PBT af (2)

Check.Quick er en del af „FsCheck“-biblioteket. Den tager en funktion som argument og genererer en række tilfældige input til funktionen på baggrund af funktionens type. Hvis funktionen returnerer „true“ for alle input, vil testen lykkes. Hvis funktionen returnerer „false“ for et input, vil testen fejle og give et eksempel på et input, der fejlede. I Listing 5 er der anvendt „Check.Quick“ til at teste, om (2) er gyldig. Funktionen „Check.Quick“ returnerer „Ok, passed 100 tests,“ hvilket indikerer, at (2) er gyldig. Det er vigtigt her at forstå, at dette ikke er det samme som at bevise, at (2) 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 anvendes Test Driven Development (TDD)² til at teste, om ens egenskab forbliver overholdt under implementeringen.

²[15] *Test-Driven Development*.

3 Symbolske udtryk

Det ønskes at kunne repræsentere simple udtryk som en type i F#. Vi skal 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 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. Tankegangen 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 hvor real- og imaginærdelen er rationale.

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.

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

Listing 7: Typen for rationelle tal

Nedenfor er der givet en signaturfil for rational modulet 8. I implementeringsfilen overloads de matematiske operatoren ved hjælp af de klassiske regneregler for brøker³.

```
1 module rational
2
3
4 type rational = R of int * int
5 with
6     static member ( ~- ) : rational -> rational
7     static member ( + ) : rational * rational -> rational
8     static member ( + ) : int * rational -> rational
9     static member ( - ) : rational * rational -> rational
10    static member ( - ) : int * rational -> rational
11    static member ( * ) : int * rational -> rational
12    static member ( * ) : rational * int -> rational
13    static member ( * ) : rational * rational -> rational
14    static member ( / ) : rational * rational -> rational
15    static member ( / ) : int * rational -> rational
16    static member ( / ) : rational * int -> rational
17    static member ( / ) : int * int -> rational
18
19 val newRational      : int * int -> rational
20 val equal            : rational * rational -> bool
21 val posetive         : rational -> bool
22 val toString         : rational -> string
23 val isZero           : rational -> bool
24 val isOne            : rational -> bool
```

³[13] *Rational Number wikipedia.*


```

25 val isInt      : rational -> bool
26 val makeInt    : rational -> int
27 val greaterThan : rational * rational -> bool
28 val isNegative  : rational -> bool
29 val absRational : rational -> rational
    
```

Listing 8: 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)⁴. 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 koncepter 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 8.1. Først defineres en type for komplekse tal, som består af et rationelt tal par, henholdsvis for realdelen og imaginærdelen, se Listing 9.

```

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

Listing 9: 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.

Definition 1 (Regneregler for komplekse tal).

Lad $a, b, c, d \in \mathbb{Q}$ Så er følgende defineret omkring komplekse tal⁵:

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**

$$\overline{a + bi} = a - bi$$

6. **Division**

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

⁴[5] Greatest common divisor wikipedia.

⁵[9] Mathematics 1a, se. Definition 3.3 s. 54, Definition 3.5 s. 56, Definition 3.8 s. 57, ligning 3-2 s. 58.

Ved implementeringen, se Listing 10, 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.

```

1 // complexDivRational: complex -> rational -> complex
2 let complexDivRational c (n) =
3     match c with
4     | _ when isZero n -> raise (System.DivideByZeroException("Complex.
5         divRational: Cannot divide by zero!"))
6     | C (a, b) -> C (a / n, b / n)
7
8 // mulConjugate: complex -> rational
9 let mulConjugate (C(a, b)) = a*a + b*b
10
11 // conjugate: complex -> complex
12 let conjugate (C (a, b)) = C (a, -b)
13
14 // mulComplex: complex -> complex -> complex
15 let mulComplex (C (a, b)) (C (c, d)) = C(a*c-b*d, b*c+a*d)
16
17 // divComplex: complex -> complex -> complex
18 let divComplex z1 z2 =
19     complexDivRational (mulComplex z1 (conjugate z2)) <| mulConjugate z2
20
21 type complex with
22     static member (+) (C(a, b), C(c, d)) = C(a + c, b + d)
23     static member (-) (C(a, b), C(c, d)) = C(a - c, b - d)
24     static member (*) (n, C(a, b)) = C(n * a, n * b)
25     static member (*) (C(a, b), n) = C(n * a, n * b)
26     static member (*) (z1, z2) = mulComplex z1 z2
27     static member (/) (z, n) = complexDivRational z n
28     static member (/) (z1, z2) = divComplex z1 z2
29     static member (~-) (C(a, b)) = C(-a, -b)
    
```

Listing 10: 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 „Number“ kan vi også opskrive en række egenskaber i 1 som vi ønsker at de skal opfylde. Egenskaberne vil blive testet i sektion 5.1.1.

Egenskab 1 (Egenskaber for Number-typen).

Lad $a, b, c \in \text{Number}$, så gælder følgende egenskaber⁶:

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

Vi begynder med at definere typen for tal i Listing 11, 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 11: Typen for Number

Typen Number er designet med henblik på, at den kan udvides med flere taltyper. Ved udvidelse er det et krav til den nye talmængde, at der er definerede matematiske operationer i form af addition, subtraktion, negation, multiplikation og division. Den nye Number type skal derudover forsat opfylde egenskaberne 1. En udvidelse kunne være for reelle tal, som kan håndtere „floating point errors“⁷, men for at undgå at komplicere programmet yderligere vil vi i denne opgave ikke inkludere decimal tal.

Betragtes signatur filen for Number modulet i Listing 12, 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
```

⁶[9] *Mathematics 1a*, se. Side 124 Definition 6.1.

⁷[3] *Floating-point error*.

```

15
16 val zero      : Number
17 val one       : Number
18 val two       : Number
19 val isZero    : Number -> bool
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 12: Signatur filen for Number modulet

Ved implementeringen af de matematiske operationer, hvis der eksisterer 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 13 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. Dermed bliver heltal og rationale tal konverteret til vores komplekse tal type, før operationen udføres.

```

1 // makeComplex: Number -> complex
2 let makeComplex n =
3     match n with
4     | Int x -> newComplex (newRational(x, 1), newRational(0, 1))
5     | Rational x -> newComplex (x, newRational(0, 1))
6     | Complex x -> x
7
8 // operation: Number -> Number -> (complex -> complex -> complex) -> Number
9 let operation a b f =
10     f (makeComplex a) (makeComplex b) |> Complex
    
```

Listing 13: operation funktionen til udførelse af matematiske operationer

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 operationers overladnings 14.

```

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 14: Overladnings funktionerne for Number

Ved at have implementeret de matematiske operationer på denne måde, er det blandt andet muligt at fortage heltales division, da det vil resultere i et rationalt tal.

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 matematisk udtryk.

3.2 Matematiske udtryk

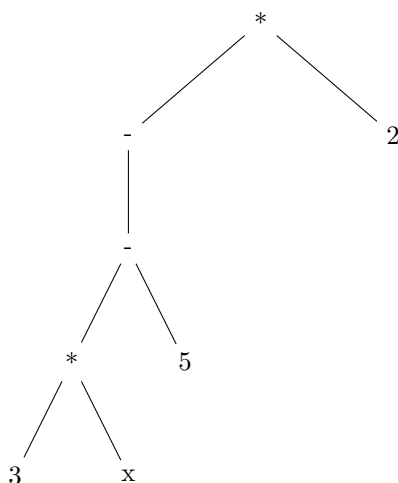
3.2.1 Polsk notation

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 regler⁸.

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

3.2.2 Udtryk som træer

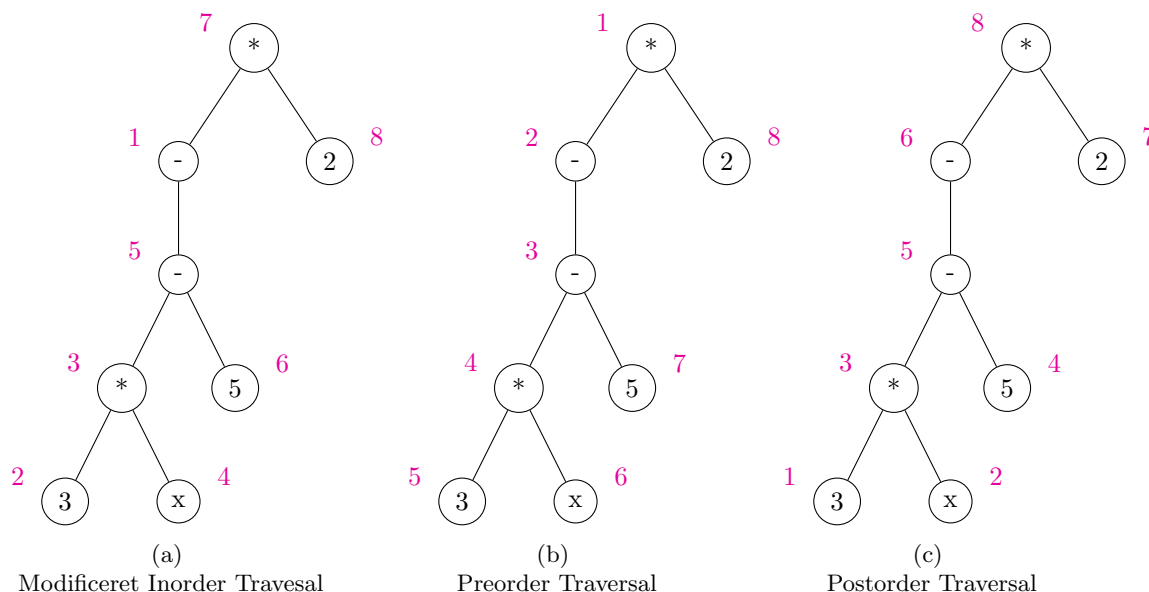
Et matematisk udtryk kan repræsenteres som et binært træ, hvor bladene er operandere i det anvendte matematiske rum og alle andre noder er operationer. Som eksempel kan udtrykket $-(3 \cdot x - 5) \cdot 2$ repræsenteres som følgende træ Figur 1.



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

⁸[11] *Polsk notation wikipedia*.

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 Traversal⁹, 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 bliver blandt andet 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 programmet med et modul for matematiske udtryk. Vi starter med at definere en polymorf type for udtryk, som beskrevet i Listing 15. Denne type omfatter flere konstruktører, hver

⁹[16] *Tree Traversal Techniques – Data Structure and Algorithm Tutorials*.

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 11. 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.

```
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 15: 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 eksemplar på en `Expr<Number>` er givet i Listing 16. 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 16: $-(3 \cdot x - 5) \cdot 2$ som et udtryks træ. Funktionen `tree` bliver beskrevet i 3.4.

Signatur filen indeholder overloadings på de matematiske operationer, så de kan anvendes på udtryk. Samt en funktion `eval` til at evaluere et udtryk beskrevet i 3.3.

```
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 isAdd : (Expr<Number> -> Expr<Number> -> Expr<Number>) -> bool
21 val isSub : (Expr<Number> -> Expr<Number> -> Expr<Number>) -> bool
22 val isMul : (Expr<Number> -> Expr<Number> -> Expr<Number>) -> bool
23 val isDiv : (Expr<Number> -> Expr<Number> -> Expr<Number>) -> bool
24 val isNeg : (Expr<Number> -> Expr<Number>) -> bool
25 val isZero : Expr<Number> -> bool
26 val containsX : Expr<Number> -> Expr<Number> -> bool
```

```

27 val getNumber : Expr<Number> -> Number
28 val getVariable : Expr<Number> -> char

```

Listing 17: Signatur filen for Expression modulet

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 18. Lignende funktioner er implementeret for de andre operationer.

```

1  // mul: Expr<Number> -> Expr<Number> -> Expr<Number>
2  let rec mul 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 isOne a           -> b
6      | N a, _ | _, N a when isZero a          -> N zero
7      | Div(a, b), c | c, Div(a, b)             -> Div (mul a c, b)
8      | Div(a, b), Div(c, d)                   -> Div ((mul a c), (mul b d))
9      | Neg a, Neg b                           -> mul a b
10     | _, _                                    -> Mul(e1, e2)
11
12 // add: Expr<Number> -> Expr<Number> -> Expr<Number>
13 let rec add 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 isZero a          -> b
17     | a, b when a = b                         -> Mul (N two, b)
18     | Neg a, Neg b                           -> neg (add a b)
19     | Neg a, b | b, Neg a                     -> Sub (b, a)
20     | Mul(a, X b), Mul(c, X d)                -> Mul(X b, a), Mul(c, X d)
21     | Mul(X b, a), Mul(c, X d)                -> Mul(a, X b), Mul(X d, c)
22     | Mul(a, X b), Mul(X d, c)                -> Mul(X b, a), Mul(X d, c)
23     | Mul(X b, a), Mul(X d, c) when b = d     -> Mul(add a c, X b)
24     | _, _                                    -> Add(e1, e2)

```

Listing 18: 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 2. Egenskaben vil blive testet i sektion 5.1.2.

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 19 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.

```

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 19: Evaluering af et udtryk

3.4 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 3. Egenskaben bliver testet i sektion 5.1.3.

Egenskab 3 (Invers morphism¹⁰ mellem infix og prefix).

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

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

Dermed gælder følgende egenskaber

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

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 20 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
    
```

¹⁰[7] Inverse function.

```

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
18 let infixExpression e = etf e false
    
```

Listing 20: konvertering 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 tokens beskriver, om en karakter i udtrykket er en operand, en operator, eller en konstant, hvor en operator også indeholder information om præcedens og associativitet¹². Typen for disse tokens kan ses i Listing 21. 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 Appendix 8.2.

```

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 21: Konvertering fra infix til udtrykstræ

3.5 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 5.1.4, 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.

¹¹[1] *Convert Infix expression to Postfix expression.*

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

Egenskab 4 (Simplifikation af udtryk).

Lad e være et udtryk, så gælder følgende:

$$\text{eval}(e) = \text{eval}(\text{simplifyExpr}(e))$$

Vi begynder med at betragte funktionen `simplifyExpr` i Listing 22, 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.

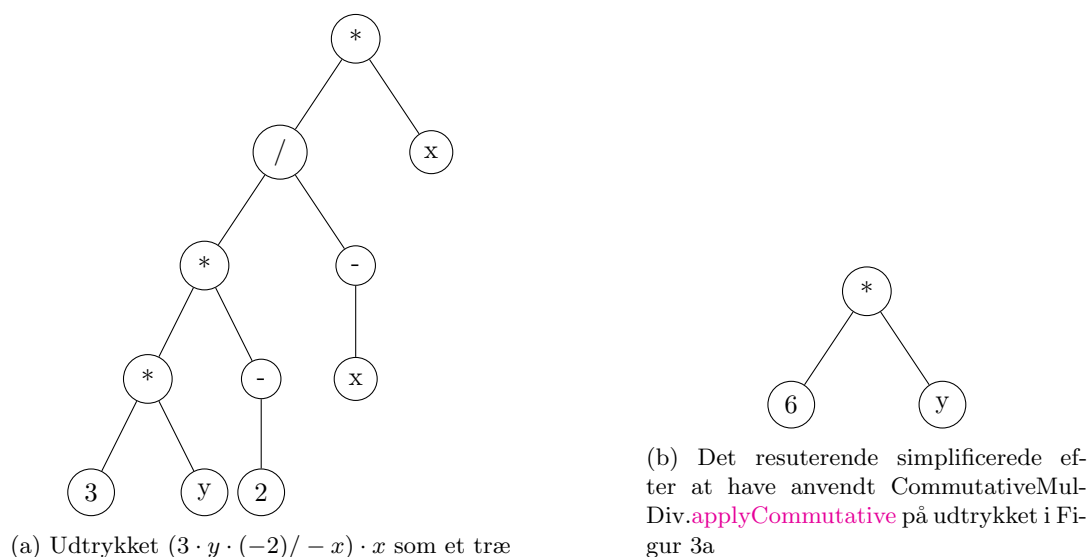
```

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

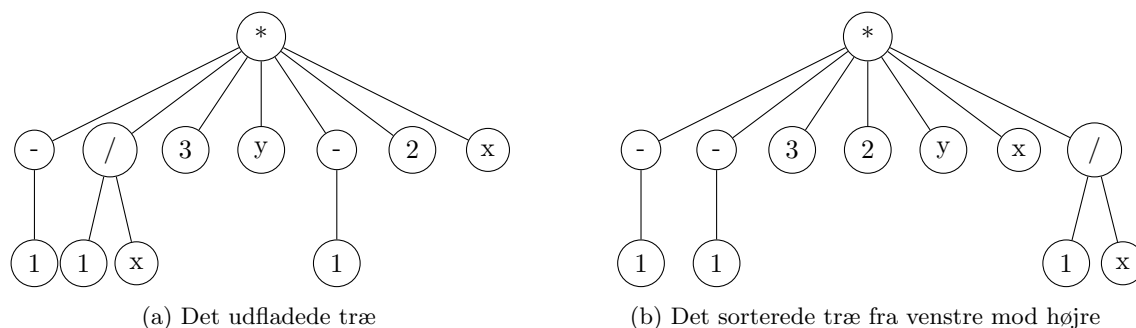
Listing 22: Simplifikation af et udtryk

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 8.3 og 8.4. 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.



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.

3.6 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 linearitetssegenskaber, som differentieringen skal overholde. Disse egenskaber vil blive testet i sektion 5.1.5.

Egenskab 5 (Linearitetsbetingelserne¹³).

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 23, som tager et udtryk og en variabel som input og differentierer udtrykket med hensyn til variabelen. 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.

```
1 // diff: Expr<Number> -> char -> Expr<Number>
2 let rec diff e dx =
3     match e with
4     | X f when f = dx -> N (Int 1)
5     | X _ -> N (Int 0)
6     | N _ -> N (Int 0)
7     | Neg f -> diff (Mul (N (Int 1), f)) dx
8     | Add(f, g) -> Add(diff f dx, diff g dx)
9     | Sub(f, g) -> Sub(diff f dx, diff g dx)
10    | Mul(f, g) -> Add(Mul(diff f dx, g), Mul(f, diff g dx))
11    | Div(f, g) -> Div(Sub(Mul(diff f dx, g), Mul(f, diff g dx)), Mul(g, g))
```

Listing 23: Differentiering af et udtryk

3.7 Multivariable polynomier af første grad

Da vi senere i projektet skal betragte matricer, vil vi i den forbindelse også lave en løsning af lineære ligningssystemer i sektion 4.5. Det kræver derfor, at vi kan isolere variable i et multivari-

¹³[2] *Differentiation Rules*.

abelt polynomium af første grad. Vi betragter derfor nu to simple funktioner til at udføre denne isolation, se Listing 24. Funktionen `isolateX` undersøger først, om den variabel, som skal isoleres, befinder sig på højre eller venstre side, derefter kaldes funktionen `expressionOnX`, som fungerer ved at evaluere til en funktion, der giver den omvendte operation af den operation, som variabelen er involveret i. Dertil anvendes den omvendte funktion på begge sider af ligningen, hvor hvis variabelen er isoleret, gives et udtrykspar, hvor det første udtryk er den isolerede variabel. Hvis variabelen ikke er isoleret, kaldes funktionen rekursivt med de nye højre og venstre sider.

```

1 // expressionOnX: Expr<'a> -> Expr<'a> -> (Expr<'a> -> Expr<'a>)
2 let rec expressionOnX hs x =
3     match hs with
4     | N _ | X _ -> fun e -> e
5     | Neg(a) when a = x -> fun e -> Neg e
6     | Sub(a, b) when a = x -> fun e -> Add(e, b)
7     | Div(a, b) when a = x -> fun e -> Mul(e, b)
8     | Div(_, a) when a = x -> fun e -> Mul(e, a)
9     | Mul(a, b) | Mul(b, a) when a = x -> fun e -> Div(e, b)
10    | Add(a, b) | Add(b, a) | Sub(b, a) when a = x -> fun e -> Sub(e, b)
11    | Add(a, b) | Sub(a, b) | Mul(a, b) | Div(a, b) -> fun e -> expressionOnX a
12    | Neg(a) -> fun e -> expressionOnX a x e
13
14 // isolateX: Expr<Number> -> Expr<Number> -> Expr<Number> -> Expr<Number> *
15 // Expr<Number>
16 let rec isolateX lhs rhs x =
17     let operation =
18         if containsX lhs x
19         then expressionOnX lhs x
20         elif containsX rhs x
21         then expressionOnX rhs x
22         else
23             failwith "Variable not found in either side of the equation"
24     match operation lhs |> simplifyExpr, operation rhs |> simplifyExpr with
25     | a, b | b, a when a = x -> (a, b)
26     | a, b -> isolateX a b x

```

Listing 24: Isolering af variable i et udtryk

Da funktionen kun betragter operationen på den variable, der ønskes isoleret, eksisterer der mange tilfælde, hvor funktionen ikke vil kunne isolere variabelen. Men den fungerer til at løse ligninger af formen $a_1x_1 + a_2x_2 + \dots + a_nx_n = b$, hvilket er tilstrækkeligt for vores formål.

4 Vektorer og Matricer

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 Listing 25.

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

Listing 25: Typen for order

Typen Order, anvendes til at angive, om en matrix er i rækkefølge (row-major) eller kolonnefølge (column-major)¹⁴. En vektor, der er lagret i rækkefølge, kan betragtes som den transponeret kolonnefølge vektor. Vi kan derfor nu definere en type for matricer, ved hjælp af en type for vektore i Listing 26.

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

Listing 26: 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 27.

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

Listing 27: Typen for dimissionen

4.1 Matrix operationer

Der vil i denne sektion beskrives en række funktioner som er nødvendige før vi kan betragte nogle funktion for anvendelse af matricer. 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 28. Funktionen laver et kald til `matrixValidMajor` som genere en fejl hvis ikke alle vektorer og matrien har samme lagringsordning. `matrixVectorLength` finder længden på vektor listen en i matricen.

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

Listing 28: Funktion til at finde dimissionen af en matrix

¹⁴[14] Row- and column-major order.

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.

4.2 Matematiske operationer

I denne sektion bør det bemærkes, at flere listings ikke inkluderer fejlhåndtering; dette er udeladt for at forbedre læsbarheden. De funktioner, der anvendes i det implementerede modul, har passende fejltjek, herunder dimensionstjek på matricerne. Den fulde implementering med fejlhåndtering kan findes i appendiks 8.5.

Før vi implementere funktioner til at udføre de ønskede matematiske operationer, vil vi først definere nogle egenskaber matricerne skal opfylde i egenskab 6.

Egenskab 6 (Vektor Aksiomer).

Lad $c, d \in \mathbb{F}$ og $v_i \in \mathbb{F}^n$ for $i = 1 \dots m$ så gælder:¹⁵

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. $c \cdot (v_1 + \dots + v_m) = c \cdot v_1 + \dots + c \cdot v_m$

4.2.1 Skalering af en matrix

Vi begynder med at betragte en funktion til at skalere en matrix se Listing 29.

```
1 // scalarVector : Number -> Vector -> Vector
2 let scalarVector (n: Number) (V (v1, o)) =
3     V ((List.map (fun x -> x * n) v1), 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 29: 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 Python¹⁶.

4.2.2 Addition af matricer

```
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, _)) =
```

¹⁵[9] *Mathematics 1a*, Theorem 7.2 s. 155.

¹⁶[12] *Python - List Comprehension*.


```

7     M (List.map2 addVector v11 v12, 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 <| 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 30: Funktion til at addere matricer og subtraktion af vektorer

Addition af vektorer reduceres til at udføre additionen elementvis, som vist i Listing 30, ved brug af List-funktionen `map2`. Vi kan bruge `addVector` til at definere matrix addition og subtraktion af vektorer. Vektor addition bruges også til at summere rækkerne i en matrix (`sumRows`), hvilket vil blive anvendt i implementeringen af matrix produkt i næste sektion og Gram-Schmidt-processen i sektion 4.3. Funktionen bliver yderligere beskrevet i definition 2.

Definition 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

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

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

4.2.3 Matrix produkt

Definition 3 (Matrix-Vektor Produkt).

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 Produkt).

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 2 for `sumRows` og definition 3 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 `matrixVectorProduct` til matrix-vektor produkt se Listing 31. Denne funktion skalerer først søjlerne i matrixen med de tilsvarende elementer i vektoren og summer derefter rækkerne i matrixen.

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

Listing 31: Funktion til matrix-vektor produkt

Definition 4 (Matrix produkt).

Lad $A \in \mathbb{F}^{m \times n}$ og $B \in \mathbb{F}^{n \times \ell}$. Lad søjlerne i B er være givet ved $b_1, \dots, b_\ell \in \mathbb{F}^n$, dermed¹⁷

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

Så defineres matrixproduktet som

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

Ud fra definition 4 ses det, at funktionen `matrixProduct` i Listing 32 til at tage produktet af to matricer, ved at udfører matrix-vektor produkt på hver søjle i matricen. Da `matrixVectorProduct` evaluere til 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 -> matrixVectorProduct a bv |> matrixToVector ) vlb
5     M(product, C)
```

Listing 32: Funktion til at tage produktet af to matricer

4.2.4 Projektion af en vektor

Definition 5 (Projektion af en vektor).

Projektionen af en vektor på en linje defineres i som følgende, hvor $Y = \text{span}\{y\}$ ¹⁸.

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

hvor det standard indre produkt er defineret som:

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

For at kunne projekte en vektor, som defineret i destination 5 skal vi først kunne konjugere en vektor. Funktionen `conjugateVector` konjugerer elementerne i en vektor. Derudover defineres en funktion til at multiplicere to vektorer elementvis (`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
```

¹⁷[9] *Mathematics 1a*, Definition 7.12 s. 162.

¹⁸[8] *Mathematics 1b - Functions of several variables*, s. 40.

```

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 33: Funktioner til at projicere 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.

Til sidst kan funktionen `proj` skrives direkte som den er defineret i definition 5.

4.3 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 fra Listing 11, 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.

Der er også 2 egenskaber 7 som vi i sektion 5.2.2 vil teste for at sikre at vores implementering af Gram-Schmidt processen er korrekt.

Egenskab 7 (Gram-Schmidt).

Lad w_1, w_2, \dots, w_n være de nye vektore som er dannes ud fra gram-schmidt processen på v_1, v_2, \dots, v_n som er lineært uafhængige vektorer. Så gælder der:

1. w_1, w_2, \dots, w_n er ortogonale.
2. w_1, w_2, \dots, w_n udspænder det samme underrum som v_1, v_2, \dots, v_n . dvs.
 $\text{span}\{v_1, v_2, \dots, v_n\} = \text{span}\{w_1, w_2, \dots, w_n\}$.

```

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

Listing 34 viser implementeringen af Gram-Schmidt processen. Lavet ud fra beskrivelse side 45 i 'Mathematics 1b'¹⁹.

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 vektore 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.

4.4 Række-echelon form

I forbindelse med udførelsen af PBT til Gram-Schmidt metoden vil vi blandt andet anvende række-echelon form til at sikre os, at et sæt af vektorer er lineært uafhængige. Derfor vil vi i denne sektion betragte implementeringen af række-echelon form, ud fra „Algorithm 9 for computing a row echelon form of a matrix“ angivet på side 142 af noterne til „01001 Mathematics 1a“²⁰. Efter at have udført PBT af Gram-Schmidt metoden, slog den i første omgang fejl, hvilket efter nærmere undersøgelse viste sig at være grundet en mindre fejl i Algorithm 9. Linje 14 burde være „ $b \leftarrow$ the j 'th entry of the i -th row of B “. Listing 35 viser implementeringen af række-echelon form, lavet ud fra psudokoden „Algorithm 9“.

```

1 // rowEchelonForm : Matrix -> Matrix
2 let rec rowEchelonForm A =

```

¹⁹[8] *Mathematics 1b - Functions of several variables*, s. 45.

²⁰[9] *Mathematics 1a*, s. 142.

```

3   if not <| correctOrderCheck A R then rowEchelonForm (correctOrder A R)
4   else
5   let (D(r, c)) = dimMatrix A
6   match A with
7   | M([], _) -> A
8   | _ when isZeroMatrix A -> A
9   | M(v::_, _) when r = 1 -> firstNonZero v |> inv |> scalarMatrix A
10  | M(_, o) ->
11      let (i, j) = firstNonZeroIndexMatrix A 0 (-1, -1)
12      let (M(B, _)) = swapFirstWith A i
13      let b = List.head B |> firstNonZero
14      let (M(B, _)) = scalarIthVector (inv b) 0 (M(B, o))
15      let R1 = List.head B
16      let R2m = List.tail B
17      let B = rowOps j 1 r R1 (M(R2m, o))
18      let (M(Cm, _)) = rowEchelonForm B
19      M(R1::Cm, o)
20
21  // rowOps: int -> int -> int -> Vector -> Matrix -> Matrix
22  and rowOps column i nrows R1 acc_m =
23      if i >= nrows then acc_m else
24      let Ri = getMatrixIthVector (i - 1) acc_m
25      let b = getVectorIthNumber column Ri
26      rowOps column (i + 1) nrows R1 <| replaceMatrixIthVector (i-1) acc_m (Ri
        - b * R1)
    
```

Listing 35: Funktion til at finde række-echelon form

4.5 Lineært ligningssystem

Som den sidste del i anvendelsen af matricer vil vi beskrive, hvordan vi kan løse et lineært ligningssystem af formen $Ax = b$. Ved at foretage række-echelon form på totalmatricen $[A | b]$ kan vi finde en løsning til ligningssystemet, hvis ranken af A er forskellig fra ranken af den totale matrix²¹.

Efter at have foretaget række-echelon form på totalmatricen, skal vi tage matrix-vektor-produktet af $\text{ref}(A)$ og x for at kunne have m ligninger med n ubekendte, hvis vi antager, at $A \in \mathbb{F}^{m \times n}$.

Den nemmeste tilgang vil være at introducere to nye typer for at repræsentere en vektor og matrix, som kan indeholde variable. Dette kan gøres ved hjælp af vores type for udtryk fra Listing 15. Vi definerer derfor følgende typer i Listing 36.

```

1 type ExprVector = list<Expr<Number>>
2 type ExprMatrix = list<ExprVector>
    
```

Listing 36: Typer for at repræsentere en vektor og matrix med variable

Det er her undladt at definere lagringsordningen for matricerne af udtryk, da vi ikke bruger dem til mere end en mellemregning i løsningen af ligningssystemet. Men alle funktioner, vi har lavet i Listing 26, ville man også kunne lave for udtryksmatricer, hvis man anvender en lagringsordning.

Ved at have udtryksmatricer defineret som en liste af udtryk, kan vi anvende den samme teknik til at foretage matrix-vektor-produktet mellem $\text{ref}(A)$ og x , som vi benyttede i Listing 31. Dette

²¹[9] *Mathematics 1a*, Corollary 6.27 s. 146.

vil resultere i en liste af udtryk, som vi ved hjælp af funktionen `isolateX` i Listing 24 kan bruge til at isolere x_i i den i 'te ligning ved at sætte den lig med b_i og indsætte det i ligningerne 1 til $i - 1$.

Den fulde implementering er at finde i appendiks 8.5. Det er værd at bemærke, at denne implementering vil resultere i en vektor af `Number`. Derfor vil den fejle, hvis der er flere ubekendte end ligninger, hvilket betyder, at koefficientmatricen A skal have fuld rank.

5 PBT af programmet

5.1 PBT af udtryk

Før vi begynder at udføre PBT på alle egenskaber vedrørende udtryk, skal vi først bygge en generator for vores talmængde og udtryk.

Vi begynder med at definere en række generatorer, som kan generere blade i vores udtrykstræer i Listing 37.

```

1 let max = 3
2 let min = -3
3
4 // noneZeroGen: Gen<int>
5 let noneZeroGen =
6     Gen.oneof [
7         Gen.choose(1, max) ;
8         Gen.choose(min, -1)]
9
10 // numberGen: Gen<Number>
11 let numberGen =
12     Gen.oneof [
13         Gen.map2 (fun x y -> newRational(x, y) |> Rational |> tryReduce ) (Gen
14             .choose(min, max)) noneZeroGen;
15         Gen.map (fun x -> Int x) (Gen.choose(min, max));
16         Gen.map4 (fun a b c d -> newComplex (newRational(a, b), newRational(c,
17             d)) |> Complex |> tryReduce ) (Gen.choose(min, max)) noneZeroGen (Gen.
18             choose(min, max)) noneZeroGen]
19
20 // numberInExprGen: Gen<Expr<Number>>
21 let numberInExprGen =
22     Gen.map (fun x -> N x) numberGen
23
24 // randomListElement: list<'a> -> Gen<'a>
25 let randomListElement xlist =
26     gen { let! i = Gen.choose(0, List.length xlist - 1)
27         return xlist.[i] }
28
29 // variableGen: list<char> -> Gen<Expr<'a>>
30 let variableGen xlist = Gen.map X (randomListElement xlist)
31
32 // leafGen: list<char> -> Gen<Expr<Number>>
33 let leafGen xlist =
34     if xlist <> [] then
35         Gen.oneof [numberInExprGen; variableGen xlist]
36     else
37         numberInExprGen
38
39 // onlyIntleafGen: list<char> -> Gen<Expr<Number>>
40 let onlyIntleafGen xlist : Gen<Expr<Number>> =
41     if xlist <> [] then
42         Gen.oneof [Gen.map (fun x -> N <| Int x) (Gen.choose(-10, 10));
43             variableGen xlist]
44     else
45         Gen.map (fun x -> N <| Int x) (Gen.choose(-10, 10))
46
47 // charsSeqGen: char -> char -> seq<Gen<char>>
48 let charsSeqGen c1 c2 = seq { for c in c1 .. c2 do

```



```

45         yield gen { return c } }
46
47 // charGen: Gen<char>
48 let charGen = gen { return! Gen.oneof (charsSeqGen 'A' 'Z')}
49
50 // smallEnvGen: Gen<Map<char, Number> * list<char>>
51 let smallEnvGen =
52     gen {
53         let! i = Gen.choose (0, 5)
54         let! xlist = Gen.listOfLength i charGen
55         let! ns = Gen.listOfLength i numberGen
56         return (Map.ofList (List.zip xlist ns), xlist) }
57
58 // exprGen: 'a -> int -> ('a -> Gen<Expr<'b>>) -> Gen<Expr<'b>>
59 let rec exprGen xlist n leafType =
60     if n = 0 then
61         leafType xlist
62     else
63         Gen.oneof [
64             // leaf occurs twice because leaf is X or N giving the same
65             // probability for each expression
66             leafType xlist;
67             leafType xlist;
68             Gen.map2 (fun x y -> Add (x, y)) (exprGen xlist (n/2) leafType) (
69                 exprGen xlist (n/2) leafType);
70             Gen.map2 (fun x y -> Mul (x, y)) (exprGen xlist (n/2) leafType) (
71                 exprGen xlist (n/2) leafType);
72             Gen.map2 (fun x y -> Div (x, y)) (exprGen xlist (n/2) leafType) (
73                 exprGen xlist (n/2) leafType);
74             Gen.map2 (fun x y -> Sub (x, y)) (exprGen xlist (n/2) leafType) (
75                 exprGen xlist (n/2) leafType);
76             Gen.map (fun x -> Neg x) (exprGen xlist (n/2) leafType)]
77
78 type SmallEnv = Map<char, Number> * char list
79 type SmallEnvGen =
80     static member SmallEnv() =
81         {new Arbitrary<SmallEnv>() with
82             override _.Generator = smallEnvGen
83             override _.Shrinker _ = Seq.empty}
84
85 type NumberGen =
86     static member Number() =
87         {new Arbitrary<Number>() with
88             override _.Generator = numberGen
89             override _.Shrinker _ = Seq.empty}

```

Listing 37: Generatorene anvendt til PBT af udtryk

Vi begynder med at definere to variable, som alle funktioner har til rådighed, `max` og `min`, som definerer intervallet for de heltal, der kan anvendes i genereringen af Numbers. Den første funktion, `noneZeroGen`, er en generator for et tilfældigt heltal fra sættet S_1 :

$$S_1 = \{x \mid x \in \mathbb{Z} \setminus \{0\}, \min \leq x \leq \max\}.$$

Vi kan dermed bruge `noneZeroGen`, da vi ikke ønsker at generere rationale tal med nævneren 0. For at definere en generator for Numbers i form af `numberGen`, som genererer et gyldigt Number

fra sættet S_5 :

$$\begin{aligned} S_2 &= \{x \mid x \in \mathbb{Z}, \quad \min \leq x \leq \max\} \\ S_3 &= \left\{ \frac{x}{y} \mid x \in S_1, y \in S_2 \right\} \\ S_4 &= \{x + yi \mid x, y \in S_3\} \\ S_5 &= S_2 \cup S_3 \cup S_4 \end{aligned}$$

Funktionen `numberInExprGen` er en generator, som ved brug af `numberGen` konverterer et `Number` til et `Expr<Number>`. Det er disse generatorer, der anvendes til at generere tal til vores talmængde.

Dernæst kommer nogle generatorer til generering af variable. Først har vi `randomListElement`, som tager en liste af en vilkårlig type og udvælger et tilfældigt element fra listen. Derudover har vi `variableGen`, som tager en liste af karakterer og bruger `randomListElement` til at udvælge en af karaktererne og konvertere den til et `Expr<Number>`.

Dermed er det nu muligt at lave en generator, som kan generere enten et tal eller en variabel fra konstruktørerne af `Expr<Number>`. Disse to konstruktører er også bladene i vores udtrykstræer. `leafGen` genererer et tilfældigt blad i vores udtrykstræ ud fra en liste af karakterer. `onlyIntleafGen` fungerer ud fra det samme princip, men genererer kun heltal fra intervallet -10 til 10.

Funktionerne `charGen` og `charsSeqGen` er generatorer, som sammen genererer en tilfældig karakter mellem 'A' og 'Z'. Endelig har vi en miljøgenerator, `smallEnvGen`, som genererer et miljø, der indeholder variable og deres tilsvarende værdier.

Den sidste generator, `exprGen`, vil i vores PBT tage en liste af variable, som den må generere blade ud fra, samt hvilken generator der skal anvendes til at generere bladene. Derudover vil den maksimale dybde af udtrykket være $\log_2(n)$.

Til sidst defineres tre typer, hvor `SmallEnvGen` og `NumberGen` gør det muligt inden kørsel af vores PBT at registrere generatorerne.

Vi har dermed nu lavet fundamentet, til at kunne teste vores egenskaber vedrørende udtryk.

5.1.1 Tal modulet

De 6 egenskaber fra 1 kan nu testes med PBT. Egenskaberne kan oversættes direkte til funktioner, grundet de overskrivninger vi har lavet på `number` typen i Listing 38.

```
1 #r "../bin/Release/net7.0/main.dll"
2 #r "nuget: FsCheck"
3 open FsCheck
4 open Number
5 open Generators
6
7 Arb.register<NumberGen>()
8
9 // Addition og multiplikation er associative
10 let associative (a:Number) (b:Number) (c:Number) =
11     a + (b + c) = (a + b) + c && a * (b * c) = (a * b) * c
12
```

```

13 // Addition og multiplikation er kommutative
14 let commutative (a:Number) (b:Number) =
15     a + b = b + a && a * b = b * a
16
17 // Distributivitet af multiplikation over addition
18 let distributive (a:Number) (b:Number) (c:Number) =
19     a * (b + c) = a * b + a * c
20
21 // Addition og multiplikation har et neutralt element
22 let neutralAdditive (a:Number) =
23     a + zero = a && a * one = a
24
25 // Omvendt funktion eksisterer til addition
26 let inverseAdditive (a:Number) =
27     a + (-a) = zero
28
29 // Omvendt funktion eksisterer til multiplikation
30 let inverseMultiplicative (a:Number) =
31     let res =
32         try
33             if a * inv a = one then 1 else 0
34         with
35             | :? System.DivideByZeroException -> 2
36     (res = 1 || res = 2)
37     |> Prop.classify (res = 1) "PropertyHolds"
38     |> Prop.classify (res = 2) "DivideByZeroExceptions"
39
40 let _ = Check.Quick associative
41 let _ = Check.Quick commutative
42 let _ = Check.Quick distributive
43 let _ = Check.Quick neutralAdditive
44 let _ = Check.Quick inverseAdditive
45 let _ = Check.Quick inverseMultiplicative
    
```

 Listing 38: *numberPBT.fsx* - funktioner til test af egenskaberne i 1

I `inverseMultiplicative` anvender vi „Prop.classify“ til at tillade, at egenskaben kan slå bestemte fejl. I dette tilfælde tillades det, at der opstår en „DivideByZeroException“. Dette sker, fordi generatoren godt kan generere 0, som der ikke kan tages en invers af. For at teste ovenstående funktioner, køres *.fsx* filen, og outputtet kan ses i Listing 39.

```

Ok, passed 100 tests.
Ok, passed 100 tests.
Ok, passed 100 tests.
Ok, passed 100 tests.
Ok, passed 100 tests.
Ok, passed 100 tests.
94% PropertyHolds.
6% DivideByZeroExceptions.
    
```

Listing 39: Outputtet fra PBT af Number typer, ved kørsel af Listing 38

Dermed viser testen at alle egenskaberne fra 1 holder.

5.1.2 Homomorfisme af evaluering

5.1.3 Invers morphism mellem infix og prefix

5.1.4 Simplifikation af udtryk

5.1.5 Differentiering af udtryk

5.2 PBT af vektorer og matricer

5.2.1 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 11.

```

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 40: Generatorene anvendt til PBT af matrix operationer i

Vi kan dermed nu lave definere egenskaberne fra 6 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 41: 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 42: Outputtet fra PBT af vektor Listing 41

5.2.2 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 // getIndependetBasisGen : Gen<Matrix>
30 let getIndependetBasisGen =
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 IndependetBasis = Matrix
38 type IndependetBasisGen =
    
```

```

39 static member IndependentBasis() =
40     {new Arbitrary<Matrix>() with
41         override _.Generator = getIndependentBasisGen
42         override _.Shrinker _ = Seq.empty}

```

Listing 43: Generatorene anvendt til PBT af Gram-Schmidt

Listing 43 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 44 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 44: 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 ortogonale 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:IndependentBasis) =
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 45: PBT af Gram-Schmidt processen

```

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

```

Listing 46: Output fra PBT af Gram-Schmidt processen

Outputtet fra PBT af Gram-Schmidt processen kan ses i Listing 46. Som sædvanligt indikere testen kun korrekthed, men ikke garanteret korrekthed.

6 Diskussion

7 Konklusion

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> (hentet 24.05.2024).
- [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).

8 Appendiks

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

8.1 complex.fsi

```

1 module complex
2 open rational
3
4 type complex = | C of rational * rational
5 with
6     static member ( + ) : complex * complex -> complex
7     static member ( - ) : complex * complex -> complex
8     static member ( * ) : rational * complex -> complex
9     static member ( * ) : complex * rational -> complex
10    static member ( * ) : complex * complex -> complex
11    static member ( / ) : complex * rational -> complex
12    static member ( / ) : complex * complex -> complex
13    static member ( ~- ) : complex -> complex
14
15 val newComplex : rational * rational -> complex
16 val isGreater : complex * complex -> bool
17 val realPart : complex -> rational
18 val isReal : complex -> bool
19 val isZero : complex -> bool
20 val toString : complex -> string
21 val isNegative : complex -> bool
22 val absComplex : complex -> complex
23 val conjugate : complex -> complex

```

Listing 47: complex.fsi

8.2 TreeGenerator.fs

```

1 module TreeGenerator
2 open Number
3 open Expression
4
5 type Associative = | Left | Right
6 type Precedence = int
7 type Operator = char * Precedence * Associative
8 type Token =
9     | Operand of char
10    | Operator of Operator
11    | Konstant of int
12 type OperatorList = Operator list
13
14 // Converts a infix string to a list of tokens
15 let rec infixToTokenList s =
16     mapToToken (Seq.toList s) true false
17
18 // maps a token to its corresponding token type
19 and mapToToken l allowUnary allowOperator=
20     match l with
21     | [] -> []

```

```

22 | x::tail when x = ' ' -> mapToToken tail allowUnary allowOperator
23 | x::tail when allowOperator && x = '/' || x = '*' -> Operator (x, 2, Left)
24 | x::tail when allowOperator && (x = '+' || (x = '-' && not allowUnary))
25 -> Operator (x, 1, Left)::mapToToken tail false false
26 | x::tail when (x = '-' && allowUnary) -> Operator ('~', 2, Right)::
27 mapToToken tail true false
28 | x::tail when x = '(' -> Operator (x, -1, Left)::mapToToken tail true
29 true
30 | x::tail when x = ')' -> Operator (x, -1, Left)::mapToToken tail false
31 true
32 | x::_ when System.Char.IsDigit(x) ->
33 let (k, tail) = foundInt 1 ""
34 Konstant (int k):: mapToToken tail false true
35 | x::tail when System.Char.IsLetter x -> Operand x::mapToToken tail false
36 true
37 | x::_ -> failwith ("Invalid syntax at: " + string x)
38
39 // Allowing more than one digit in a number
40 and foundInt 1 s =
41 match 1 with
42 | x::tail when System.Char.IsDigit(x) -> foundInt tail (s + string x)
43 | _ -> (s, 1)
44
45 // pops the last operator from the stack and adds it to the prefix list
46 let popprefixStack op prefix =
47 match op, prefix with
48 | x, e1::e2::tail when x = '+' -> Add(e2, e1)::tail
49 | x, e1::e2::tail when x = '-' -> Sub(e2, e1)::tail
50 | x, e1::e2::tail when x = '*' -> Mul(e2, e1)::tail
51 | x, e1::e2::tail when x = '/' -> Div(e2, e1)::tail
52 | x, e::tail when x = '~' -> Neg(e)::tail
53 | x, _ when x = '(' -> prefix
54 | _, _ -> failwith "match not found"
55
56 // Runs the algorithm to generate the expression tree
57 let rec generateExpression c (stack:OperatorList) prefix =
58 match c, stack with
59 | [], [] -> prefix
60 | [], (s,_,_):stack_tail -> generateExpression c stack_tail (
61 popprefixStack s prefix)
62 | Operand x :: tail, _ -> generateExpression tail stack (X x::prefix)
63 | Konstant x :: tail, _ -> generateExpression tail stack (N (Int x)::prefix)
64 )
65 | Operator (x, prec, lr)::tail, _
66 when x = '('
67 -> generateExpression tail ((x, prec, lr)::stack) prefix
68 | Operator (x, _, _):tail, _
69 when x = ')'
70 -> match stack with
71 | [] -> generateExpression tail stack prefix
72 | (s,_,_):stack_tail
73 when s = '('
74 -> generateExpression tail stack_tail prefix
75 | (s,_,_):stack_tail
76 -> generateExpression c stack_tail (popprefixStack s prefix)
77 | Operator e::tail, [] -> generateExpression tail (e::stack) prefix
78 | Operator e::tail, s::stack_tail

```

```

73     -> match e, s with
74     | (x, precX, lr), (y, precY, _)
75         when precX < precY || (precX = precY && lr = Left)
76         -> generateExpression c stack_tail (popprefixStack y prefix)
77     | _, _ -> generateExpression tail (e::stack) prefix
78
79 // Converts a infix string to a expression tree
80 let tree s =
81     match generateExpression (infixToTokenList s) [] [] with
82     | [] -> failwith "Tree is empty"
83     | tree::_ -> tree
84
85 // Adds parenthesis to a string if a boolean is true
86 let parenthesis b f =
87     if b then "(" + f + ")" else f
88
89 // Converts a expression tree to a infix string
90 let rec etf e p =
91     match e with
92     | N a when not <| isInt a -> parenthesis p <| toString a
93     | N a -> toString a
94     | X a -> string a
95     | Neg a -> parenthesis p <| "-" + etf a (not p)
96     | Add(a, b) -> parenthesis p <| etf a false + "+" + etf b false
97     | Sub(a, b) -> parenthesis p <| etf a false + "-" + etf b true
98     | Mul(a, b) -> parenthesis p <| etf a true + "*" + etf b true
99     | Div(a, b) -> parenthesis p <| etf a true + "/" + etf b true
100
101 // Converts a expression tree to a infix string
102 let infixExpression e = etf e false
    
```

Listing 48: TreeGenerator.fs

8.3 CommutativeAddSub.fs

```

1 module commutativeAddSub
2 open Expression
3 open Number
4
5
6 // rank when sorting a commutative expression
7 let expressionSortRank e1 =
8     match e1 with
9     | N _ -> 1
10    | Neg (N _) -> 2
11    | X _ -> failwith "Variables should not be in the list" // all variables
    are multiplied with 1
12    | Add _ -> failwith "Addition should not be in the list" // all addition
    should be reduced
13    | Sub _ -> failwith "Subtraction should not be in the list" // all
    subtraction should be reduced
14    | Mul _ -> 6
15    | Div _ -> 7
16    | Neg _ -> 8
17
18 // Flattens a expression tree with respect to addition and subtraction
19 let rec flatTree e =
20     match e with
    
```

```

21 | Add (a, b) -> flatTree a @ flatTree b
22 | Sub (a, b) -> flatTree a @ flatTree (Neg b)
23 | Neg (Add(a, b)) -> flatTree (Neg a) @ flatTree (Neg b)
24 | Neg (Sub(a, b)) -> flatTree (Neg a) @ flatTree b
25 | X a -> [Mul(N one, X a)]
26 | Div (a, b) -> [a / b]
27 | Neg (Neg a) -> flatTree a
28 | N _ | Div _ | Mul _ | Neg _ -> [e]
29
30
31
32 let rec sort l = List.sortBy (fun e -> expressionSortRank e) l
33
34 // Reduces a sorted commutative list for addition
35 let rec reduceNumbers l =
36     // printfn "rn %A" l
37     match l with
38     | [] -> []
39     | N a :: N b :: tail -> reduceNumbers (N a + N b :: tail)
40     | N a :: Neg (N b) :: tail -> reduceNumbers (N a - N b :: tail)
41     | Neg (N a) :: Neg (N b) :: tail -> reduceNumbers (Neg (N a + N b) :: tail)
42     | Neg (N a) :: tail -> Neg (N a) :: tail
43     | N a :: tail -> N a :: tail
44     | _ -> l
45
46 // given a list of commutative expressions, and a variable, sums the
47 // coefficients of the variable
48 let rec sumInstancesOfVariable x l =
49     match x, l with
50     | _, [] -> ([], x)
51     | Mul(N n1, X x1), Mul(N n2, X x2) :: tail
52     | Mul(N n1, X x1), Mul(X x2, N n2) :: tail
53     | Mul(X x1, N n1), Mul(N n2, X x2) :: tail
54     | Mul(X x1, N n1), Mul(X x2, N n2) :: tail
55     when x1 = x2 -> sumInstancesOfVariable (Mul(N (n1 + n2), X x1)) tail
56     | _, head :: tail ->
57         let (l_new, x_new) = sumInstancesOfVariable x tail
58         (head :: l_new, x_new)
59
60
61
62 let rec reduceVariables l =
63     match l with
64     | [] -> []
65     | Mul(N a, X b) :: tail
66     | Mul(X b, N a) :: tail
67     ->
68         let (l_new, x_new) = sumInstancesOfVariable (Mul(N a, X b)) tail
69         x_new :: reduceVariables l_new
70     | head :: tail -> head :: reduceVariables tail
71
72 let rec rebuildTree l =
73     // printfn "rt %A" l
74     match l with
75     | [] -> N zero
76     | Neg x :: tail -> (rebuildTree tail) - x
77     | Mul (a, b) :: tail -> (rebuildTree tail) + (a * b)

```

```

78 | Div (a, b)::tail -> (rebuildTree tail) + (a / b)
79 | x::tail -> rebuildTree tail + x
80
81 let applyCommutative e =
82     match e with
83     | Sub _ | Add _ -> flatTree e |> sort |> reduceNumbers |> reduceVariables
84     |> rebuildTree
85     | _ -> e

```

Listing 49: CommutativeAddSub.fs

8.4 CommutativeMulDiv.fs

```

1 module commutativeMulDiv
2 open Expression
3 open Number
4
5
6 ///////////////////////////////////////////////////
7 /// Commutative RULES For multiplication and division ///
8 ///////////////////////////////////////////////////
9
10 // rank when sorting a commutativ expression
11 let expressionSortRank e1 =
12     match e1 with
13     | Neg _ -> 1
14     | N _ -> 2
15     | X _ -> 3
16     | Add _ -> 4
17     | Sub _ -> 5
18     | Mul _ -> 6
19     | Div _ -> 7
20
21 // sorts the commutativ list
22 let rec sort l = List.sortBy (fun e -> expressionSortRank e) l
23
24
25 // determines the sign of a commutativ list
26 let rec signList l s =
27     match l with
28     | [] -> s
29     | Neg _::tail -> signList tail (-1*s)
30     | _::tail -> signList tail s
31
32
33 // flattens the tree to a commutativ list
34 let rec flatTree e =
35     match e with
36     | N _ | X _ | Add _ | Sub _ -> [e]
37     | Neg a -> Neg (N one) :: flatTree a
38     | Mul (a, b) -> flatTree a @ flatTree b
39     | Div (N a, N b) -> [N (a / b)]
40     | Div (Neg a, b) | Div (a, Neg b) -> Neg (N one) :: flatTree (Div (a, b))
41     | Div (a, N b) -> N (one / b) :: flatTree a
42     | Div (a, b) -> Div (N one, b) :: flatTree a
43
44
45 // multiplies all Div elements in a commutativ list

```

```

46 let rec mulDivElements l =
47     // division will always be in the end of a sorted list, and numarator will
    be 1
48     match l with
49     | [] -> []
50     | Div (_, b) :: Div (_, c) :: tail -> mulDivElements (Div(N one, Mul(b, c)
    ) :: tail)
51     | x::tail -> x :: mulDivElements tail
52
53 // removes a element from a list
54 let rec removeElem e l =
55     match l with
56     | [] -> []
57     | x::tail when x = e -> tail
58     | x::tail -> x :: removeElem e tail
59
60 // reduces a commutativ list
61 let rec reduce l =
62     let sorted = divCancelling (sort l)
63     if signList sorted 1 > 0 then rebuildTree sorted else Neg (rebuildTree
    sorted)
64
65 // initiates the division cancelling
66 and divCancelling l =
67     // printfn "DivCancelling: %A" l
68     match List.rev l with
69     | [] -> 1
70     | Div(_, b)::tail ->
71         let (numerator, denominator) = cancelEquality (List.
    rev tail) (sort (flatTree b))
72         sort (flatTree (reduce numerator / reduce denominator)
    )
73     | _ -> 1
74
75 // cancels out equal elements in the numerator and denominator
76 and cancelEquality nu de =
77     // printfn "cancelEquality: %A / %A" nu de
78     match nu with
79     | [] -> ([], de)
80     | n::ntail when List.contains n de -> cancelEquality ntail (removeElem n
    de)
81     | n::ntail -> let (numerator, denominator) = cancelEquality ntail de
    (n::numerator, denominator)
82
83
84
85
86 // rebuilds a commutativ list to a tree
87 and rebuildTree l =
88     match l with
89     | [] -> N one
90     | Neg (N a)::tail when Number.isOne a -> rebuildTree tail
91     | Neg a::tail -> rebuildTree (a::tail)
92     | N a::N b::tail -> rebuildTree (N (a * b) :: tail)
93     | N a :: Add(b, c) :: tail -> rebuildTree (Add(reduce (flatTree (Mul(N a,
    b))), reduce (flatTree (Mul(N a, c)))) :: tail)
94     | N a :: Sub(b, c) :: tail -> rebuildTree (Sub(reduce (flatTree (Mul(N a,
    b))), reduce (flatTree (Mul(N a, c)))) :: tail)
95     | a::tail -> a * rebuildTree tail
96

```

```

97
98 // applies the commutativ rules to a tree
99 let applyCommutative e:Expr<Number> =
100     match e with
101     | Mul _ | Div _ -> reduce (flatTree e)
102     | _ -> e

```

Listing 50: CommutativeMulDiv.fs

8.5 Matrix.fs

```

1 module Matrix
2 open Number
3 open Expression
4 open SymbolicManipulation
5
6 // row or coloumn major ordere: default of a vector and matric is Column major
  // order.
7 type Order = | R | C
8
9 // Rows x Cols
10 type Dimension = D of int * int
11
12 // a "Normal" is C major and a Transposed one is R major
13 type Vector = V of list<Number> * Order
14 type Matrix = M of list<Vector> * Order
15
16 // Construct a vector
17 let vector nl = V (nl, C)
18
19 // The dimension of a vector
20 let dimVector (V(v, o)) =
21     let len = List.length v
22     match o with
23     | R -> D (1, len)
24     | C -> D (len, 1)
25
26 // Makes sure that a matrix has the same major order as its vectors
27 let matrixValidMajor (M(m, o)) =
28     match List.head m, o with
29     | V (_, R), R -> true
30     | V (_, C), C -> true
31     | _, _ -> failwith "Matrix's vector's dont have same major order"
32
33 // The list length of a vector
34 let vectorLength (V(v, _)) = List.length v
35
36 // The list length of a matrix's vectors
37 let matrixVectorLength (M(m, _)) =
38     match m with
39     | [] -> 0
40     | x::_ -> vectorLength x
41
42 // The dimension of a matrix
43 let dimMatrix (M(vl, o)) =
44     if vl = [] then D (0, 0)
45     else
46         let _ = matrixValidMajor (M(vl, o))

```

```

47 let d1 = List.length v1
48 let d2 = matrixVectorLength (M(v1, o))
49 match o with
50 | R -> D (d1, d2)
51 | C -> D (d2, d1)
52
53 // Multiplication of a scalar and a vector
54 let scalarVector (n:Number) (V (v1, o)) =
55     V ((List.map (fun x -> x * n) v1), o)
56
57 // Multiplication of a scalar and a matrix
58 let scalarMatrix (M (v1, o)) n =
59     M ((List.map (fun x -> scalarVector n x) v1), o)
60
61 // Addition two vectors
62 let addVector x y =
63     let (V (v1, o1)) = x
64     let (V (v2, o2)) = y
65     if o1 <> o2
66     then failwith "Vectors must have the same major order"
67     elif dimVector x <> dimVector y
68     then failwith "Vectors must have the same dimension"
69     else
70         V ((List.map2 (+) v1 v2), o1)
71
72 // Subtraction of two vectors
73 let subVector x y =
74     scalarVector (-one) y |> addVector x
75
76
77 // Construct a vector of n with length len
78 let vectorOf n len = V ((List.init len (fun _ -> n)), C)
79
80 // Construct a matrix of n with dimension d
81 let matrixOf n (D (r, c)) = M ((List.init c (fun _ -> vectorOf n r)), C)
82
83 // Transposes a vector
84 let transposeVector (V(v, o)) =
85     match o with
86     | R -> V(v, C)
87     | C -> V(v, R)
88
89 // adds a dimension to a vector
90 let extendVector (V(v, o)) n =
91     V(v @ [n], o)
92
93 // Extend a matrix with a number list
94 let extendMatrix (M(m, o)) nl =
95     if m <> [] && nl <> [] && List.length nl <> matrixVectorLength (M(m, o))
96     then failwith "The list must have the same length as the matrix's vectors"
97     elif nl = [] then M(m, o)
98     else
99         M (m @ [(V (nl, o))], o)
100
101 // extend a matrix with a matrix
102 let extendMatrixWithMatrix (M(m1, o1)) (M(m2, o2)) =
103     if o1 <> o2 then failwith "Matrices must have the same major order"
104     elif matrixVectorLength (M(m1, o1)) <> matrixVectorLength (M(m2, o2)) then
105         failwith "Matrices must have the same dimension"

```



```

104     else M (m1 @ m2, o1)
105
106
107 // Alternates the Major order
108 let alternateOrder o =
109     match o with
110     | R -> C
111     | C -> R
112
113 // Alternates the Major order of a matrix
114 let alternateOrderMatrix (M(m, o)) =
115     M (m, alternateOrder o)
116
117 // The i'th number of a vector
118 let getVectorIthNumber i (V(v, _)) = v.[i]
119
120 // The i'th vector of a matrix
121 let getMatrixIthVector i (M(m, _)) = m.[i]
122
123 // replaces the i'th vector of a matrix
124 let replaceMatrixIthVector i (M(m, o)) v =
125     M (m.[0..i-1] @ [v] @ m.[i+1..], o)
126
127 // pops a vector
128 let seperateFistNumberFromVector (V(v, o)) =
129     (v.Head, V(v.Tail, o))
130
131 // first element of a vector
132 let headVector (V(v, _)) = List.head v
133
134 // Gives a vector of all the first elements of the vectors in a matrix
135 let rec firstElemetsVectors (M(m, o)) v_acc m_acc =
136     match m with
137     | [] -> (v_acc, M(m_acc, o))
138     | x::xs ->
139         let (n, tail) = seperateFistNumberFromVector x
140         firstElemetsVectors (M(xs, o)) (v_acc @ [n]) (m_acc @ [tail])
141
142 // Helper function for changeOrderMatrix
143 let rec chaingingOrderMatrix (M(m, o)) m_acc =
144     match matrixVectorLength (M(m, o)) with
145     | 0 -> m_acc
146     | _ ->
147         let (v, m_new) = firstElemetsVectors (M(m, o)) [] []
148         extendMatrix m_acc v |> chaingingOrderMatrix m_new
149
150 // Changes the order of a matrix
151 let changeOrderMatrix (M(m, o)) =
152     alternateOrderMatrix (M([], o)) |> chaingingOrderMatrix (M(m, o))
153
154 // Makes sure that two matrices have the same major order, if diffrent then
    changes the order of the second matrix
155 let giveMatrixHaveSameOrder (M(_, o1)) (M(m2, o2)) =
156     if o1 <> o2 then changeOrderMatrix (M(m2, o2)) else M(m2, o2)
157
158 // Transposes a matrix
159 let transposeMatrix (M(vl, o)) =
160     M(List.map (fun x -> transposeVector x) vl, alternateOrder o)
161

```

```

162 // Adds two matrices elemwise
163 let addMatrix m1 m2 =
164     let m3 = giveMatrixHaveSameOrder m1 m2
165     if dimMatrix m1 <> dimMatrix m3
166     then failwith "addMatrix: Matrices must have the same dimension"
167     else
168         let (M(vl1, o)) = m1
169         let (M(vl3, _)) = m3
170         M (List.map2 addVector vl1 vl3, o)
171
172 // Construct a matrix
173 let matrix vl =
174     let rec mc vl m =
175         match vl with
176         | [] -> m
177         | (V(x, _))::xs -> mc xs (extendMatrix m x)
178     mc vl (M([], C))
179
180 // Changes the order of af matrix to x
181 let correctOrder (M(m, o)) x =
182     if o = x then M(m, o) else changeOrderMatrix (M(m, o))
183
184 // Boolean value of if a matrix has the correct order
185 let corectOrderCheck (M(_, o)) x =
186     o = x
187
188 // Sum the rows of a matrix
189 let rec sumRows m =
190     if not <| corectOrderCheck m C
191     then sumRows <| correctOrder m C
192     else
193         let zeroVector = vectorOf zero <| matrixVectorLength m
194         let (M(vl, _)) = m
195         matrix [List.fold (addVector) zeroVector vl]
196
197
198
199 // matrix vector product  $A \cdot v = b$  - Definition 7.10
200 let rec matrixVectorProduct m v =
201     let (D(rv, _)) = dimVector v
202     let (D(_, cm)) = dimMatrix m
203     if rv <> cm
204     then failwith "matrixVector: the number of columns of the matrix has to be
205         the same as the number of entries in the vector."
206     elif not <| corectOrderCheck m C then matrixVectorProduct (correctOrder m
207         C) v
208     else
209         let (M(vl, _)) = m
210         let (V(nl, _)) = v
211         M (List.map2 (fun mc n -> scalarVector n mc) vl nl, C)
212         |> sumRows
213
214 // Converts a matrix to a vector if possible
215 let matrixToVector m =
216     match dimMatrix m, m with
217     | D(r, c), M(v::_, _) when r = 1 || c = 1 -> v
218     | _, _ -> failwith "mactrixToVector: Matrix is not a vector"
219
220 // Multiplies two matrices - Definition 7.12

```

```

219 let rec matrixProduct a b =
220     let (D(_, ca)) = dimMatrix a
221     let (D(rb, _)) = dimMatrix b
222     if ca <> rb
223     then failwith "matrixProduct: matrix product A .* B is defined only if the
224         number of columns of A is the same as the number of rows of B"
225     elif not <| correctOrderCheck b C then matrixProduct a (correctOrder b C)
226     else
227         let (M(vlb, _)) = b
228         let product = List.map (
229             fun bv -> matrixVectorProduct a bv |> matrixToVector ) vlb
230         M(product, C)
231
232 type Vector with
233     static member (+) (v1, v2) = addVector v1 v2
234     static member (-) (v1, v2) = subVector v1 v2
235     static member (*) (n, v) = scalarVector n v
236     static member (*) (v, n) = scalarVector n v
237     static member (~-) (v) = scalarVector (-one) v
238
239 type Matrix with
240     static member (+) (m1, m2) = addMatrix m1 m2
241     static member (+) (m, n) = dimMatrix m |> matrixOf n |> addMatrix m
242     static member (+) (n, m) = dimMatrix m |> matrixOf n |> addMatrix m
243     static member (*) (n, m) = scalarMatrix m n
244     static member (*) (m, n) = scalarMatrix m n
245     static member (*) (m, v) = matrixVectorProduct m v
246     static member (*) (m1, m2) = matrixProduct m1 m2
247     static member (/) (m, n) = scalarMatrix m (inv n)
248
249 // Horizontal flip of a matrix
250 let flip m =
251     let (M(m_new, _)) = correctOrder m C
252     List.rev m_new |> matrix
253
254 // extracts the first vector of a matrix
255 let extractfirstVector (M(m, o)) =
256     match m with
257     | [] -> failwith "Matrix is empty"
258     | x::xs -> x, M(xs, o)
259
260 // Last vector of a matrix
261 let rec extractlastVector m =
262     let (v, mf) = flip m |> extractfirstVector
263     (v, flip mf)
264
265 // Multiplies two vectors element wise
266 let vectorMulElementWise (V(u, o1)) (V(v, o2)) =
267     if o1 <> o2
268     then failwith "Vectors must have the same major order"
269     elif List.length u <> List.length v
270     then failwith "Vectors must have the same dimension"
271     else
272         V (List.map2 (*) u v, o1)

```

```

277
278 // Conjugates a vector
279 let conjugateVector (V(v, o)) =
280     V (List.map conjugate v, o)
281
282 // Inner product of two vectors
283 let innerProduct u v =
284     let (V(w, _)) = conjugateVector v |> vectorMulElementWise u
285     List.fold (+) zero w
286
287 // Projection of a vector on another vector
288 let proj y x =
289     scalarVector (innerProduct x y / innerProduct y y) y
290
291 // Dot product of two vectors
292 let dotProduct (V(u, ou)) (V(v, ov)) =
293     if List.length u <> List.length v
294     then failwith "dotProduct: Vectors must have same length."
295     else
296
297         match ou, ov with
298         | R, C -> innerProduct (V(u, C)) (V(v, C))
299         | _ -> failwith "Missing implementation for this case."
300
301
302 // Orthogonal basis using the Gram-Schmidt process
303 let rec orthogonalBasis m =
304     if not <| correctOrderCheck m C
305     then orthogonalBasis (correctOrder m C)
306     else
307
308         // Gram-Schmidt process but without the normalization
309         let rec Gram_Schmidt vm acc_wm =
310             match acc_wm [], vm with
311             | x, M([], _) -> x
312             | M([], _), M(v1::vrest, o) ->
313                 Gram_Schmidt (M(vrest,o))
314                 <| fun x -> extendMatrix (M([v1], C)) x
315             | M(w, _), M(vk::vrest, o) ->
316                 let (V(wk, _)) = vk - sumProj w vk
317                 Gram_Schmidt (M(vrest,o))
318                 <| fun x -> extendMatrix (acc_wm wk) x
319
320             // Sum all the projections of vk on w1 to wk-1
321             and sumProj w vk =
322                 List.map (fun x -> proj x vk) w
323                 |> matrix
324                 |> sumRows
325                 |> matrixToVector
326
327             Gram_Schmidt m (fun _ -> M([], C))
328
329
330
331 // Checks if a vector is a zero vector
332 let isZeroVector (V(v, _)) =
333     List.forall (fun x -> Number.isZero x) v
334
335 // Checks if a matrix is a zero matrix

```

```

336 let isZeroMatrix (M(m, _)) =
337     List.forall (fun x -> isZeroVector x) m
338
339 // first non zero element of a vector
340 let firstNonZero (V(v, _)) =
341     if isZeroVector (V(v, C)) then failwith "firstNonZero: Vector does not
342     have a non zero element"
343     else
344         List.find (fun x -> not (Number.isZero x)) v
345
346 // Index of the first non zero element of a vector
347 let firstNonZeroIndex (V(v, _)) =
348     if isZeroVector (V(v, C)) then -1
349     else
350         List.findIndex (fun x -> not (Number.isZero x)) v
351
352 // string a vector
353 let stringVector (V(v, o)) =
354     let space = if o = C then "\n" else " "
355     List.map (fun x -> toString x) v |> String.concat space
356
357 // string a matrix
358 let rec stringMatrix m =
359     if not <| correctOrderCheck m R then stringMatrix (correctOrder m R)
360     else
361         let (M(vl, _)) = m
362         List.map (fun x -> stringVector x) vl |> String.concat "\n"
363
364 // Index of the first non zero element of a matrix
365 let rec firstNonZeroIndexMatrix (M(m, o)) idx (r, c) =
366     if m = [] then (r, c) else
367         let fnxi = List.head m |> firstNonZeroIndex
368         match m with
369         | [] -> (r, c)
370         | _::vt when fnxi >= 0 && (fnxi < c || c = -1) -> firstNonZeroIndexMatrix
371         (M(vt, o)) (idx + 1) (idx, fnxi)
372         | _::vt -> firstNonZeroIndexMatrix (M(vt, o)) (idx + 1) (r, c)
373
374 // Switches two vectors in a matrix
375 let swapFirstWith (M(m, o)) i =
376     if i = 0
377     then M(m, o)
378     else
379         let rec swapper h m i idx acc_m =
380             match m with
381             | [] -> failwith "swapFirstWith: Index out of range"
382             | x::xs when idx = i -> x :: acc_m @ (h::xs)
383             | x::xs -> swapper h xs i (idx + 1) (acc_m @ [x])
384
385         match m with
386         | [] -> failwith "swapFirstWith: Matrix is empty"
387         | h::tail -> M(swapper h tail i 1 [], o)
388
389 // Multiplies the i'th vector with a scalar
390 let scalarIthVector c i (M(m, o)) =
391     let rec sIV c m i idx acc_m =
392         match m with

```

```

393         | [] -> failwith "scalarIthVector: Index out of range"
394         | x::xs when idx = i -> acc_m @ c * x :: xs
395         | x::xs -> sIV c xs i (idx + 1) (acc_m @ [x])
396
397     M(sIV c m i 0 [], o)
398
399 // Alters row j with Rj <- Rj - c * Ri
400 let rec rowOperation i j c m =
401     if i = j then failwith "rowOperation: Row i and j must be different j <> i"
402     elif not <| correctOrderCheck m R then rowOperation i j c <| correctOrder m
403         R
404     else
405         replaceMatrixIthVector (j-1) m <| getMatrixIthVector (j-1) m - c *
406         getMatrixIthVector (i-1) m
407
408 // row echelon form of a matrix
409 let rec rowEchelonForm A =
410     if not <| correctOrderCheck A R then rowEchelonForm (correctOrder A R)
411     else
412         let (D(r, c)) = dimMatrix A
413         match A with
414         | M([], _) -> A
415         | _ when isZeroMatrix A -> A
416         | M(v::_, _) when r = 1 -> firstNonZero v |> inv |> scalarMatrix A
417         | M(_, o) ->
418             let (i, j) = firstNonZeroIndexMatrix A 0 (-1, -1)
419             let (M(B, _)) = swapFirstWith A i
420             let b = List.head B |> firstNonZero
421             let (M(B, _)) = scalarIthVector (inv b) 0 (M(B, o))
422             let R1 = List.head B
423             let R2m = List.tail B
424             let B = rowOps j 1 r R1 (M(R2m, o))
425             let (M(Cm, _)) = rowEchelonForm B
426             M(R1::Cm, o)
427
428 // Ri <- Ri - b * R1 - possible error i mat 1 notes Algorithm 1 (should be b <-
429 // the jth entry og the ith row if B)
430 and rowOps coloumn i n rows R1 acc_m =
431     if i >= n rows then acc_m else
432     let Ri = getMatrixIthVector (i - 1) acc_m
433     let b = getVectorIthNumber coloumn Ri
434     rowOps coloumn (i + 1) n rows R1 <| replaceMatrixIthVector (i-1) acc_m (Ri
435     - b * R1)
436
437 // A standard bacis vector of length n with 1 at i
438 let standardBacisVector n i =
439     let rec sbv idx =
440         match idx with
441         | _ when idx < 1 -> []
442         | 1 when i <> 1 -> [zero]
443         | x when x = i -> one :: sbv (x - 1)
444         | _ -> zero :: sbv (idx - 1)
445     vector <| sbv n
446
447 // A standard bacis matrix of F^n
448 let standardBacis n =
449     let rec sbv idx =

```

```

448         match idx with
449         | _ when idx < 1 -> []
450         | 1 -> [standardBasisVector n 1]
451         | _ -> standardBasisVector n idx :: sb (idx - 1)
452     matrix <| sb n
453
454 // fullranked diagonal matrix
455 let fullrankedDiagonalMatrix n m =
456     if n > m then failwith "fullrankedDiagonalMatrix: number of rows must be
457     less than or equal to the number of columns"
458     else
459         let rec frdm i frm =
460             match i with
461             | _ when i < 0 -> failwith "fullrankedDiagonalMatrix: i must be
462             greater than or equal to 0"
463             | 0 -> frm
464             | _ ->
465                 let (V(zeroV, _)) = vectorOf zero n
466                 frdm (i - 1) <| extendMatrix frm zeroV
467         frdm (m - n) <| standardBasis n
468
469 // Checks if a matrix is upper triangular
470 let rec isUpperTriangular (M(vl, o)) =
471     if not <| correctOrderCheck (M(vl, o)) R then isUpperTriangular (
472     correctOrder (M(vl, o)) R)
473     else
474         let (D(_, m)) = dimMatrix (M(vl, o))
475         let rec iut vl idx =
476             match vl with
477             | [] -> true
478             | x::xs when idx >= m -> isZeroVector x && iut xs (idx + 1)
479             | x::xs -> firstNonZeroIndex x = idx && iut xs (idx + 1)
480         iut vl 0
481
482 // determines if a matrix has full rank
483 let hasFullRank m =
484     rowEchelonForm m |> isUpperTriangular
485
486 // checks is if every vector has inner product of zero with the next vector
487 let rec isOrthogonalBasis m =
488     let isob (M(vl, o)) =
489         match vl with
490         | [] -> true
491         | _::[] -> true
492         | v::vnext::vrest -> innerProduct v vnext = zero && isOrthogonalBasis
493         (M(vnext::vrest, o))
494     isob <| correctOrder m C && hasFullRank m
495
496 // split a matrix into two matrices
497 let rec splitMatrix o i m =
498     if not <| correctOrderCheck m o then splitMatrix o i (correctOrder m o)
499     else
500         let (M(vl, _)) = m
501
502         let rec sm vl idx acc =

```

```

503         match vl with
504         | [] -> failwith "splitMatrix: Index out of range"
505         | x::xs when idx = 0 -> matrix <| acc @ [x], matrix xs
506         | x::xs -> sm xs (idx - 1) (acc @ [x])
507     sm vl i []
508
509 // Determines if a matrix has a zero column
510 let rec dontHaveZeroCols m =
511     if not <| correctOrderCheck m C then dontHaveZeroCols (correctOrder m C)
512     else
513         let (M(vl, _)) = m
514         List.forall (fun x -> not <| isZeroVector x) vl
515
516
517 // Determines if a matrix has the same span as another matrix
518 let hasSameSpan m1 m2 =
519     let (D(r, c)) = dimMatrix m1
520     if (D(r, c)) <> dimMatrix m2 then failwith "Matrices must have the same
521     dimension"
522     else
523         let hSS m1 m2 =
524             let s1, s2 = extendMatrixWithMatrix m1 m2 |> rowEchelonForm |>
525             splitMatrix C (c-1)
526             hasFullRank s1 && dontHaveZeroCols s2 && hasFullRank s2
527         hSS m1 m2 && hSS m2 m1
528
529
530
531
532
533 ///////////////////////////////////////////////////////////////////
534 /// Functions to solve Ax = b ///
535 ///////////////////////////////////////////////////////////////////
536
537 type ExprVector = list<Expr<Number>>
538 type ExprMatrix = list<ExprVector>
539
540
541
542 // multiplies a expression with a vector and returns a Expr list
543 let scalarWithExpr (V(nl, _)) e=
544     List.map (fun x -> N x * e) nl
545
546 // matrix product with expression vector
547 let matrixProductExprList vl el =
548     let znl = List.head vl |> vectorLength |> vectorOf zero
549     let zeroExprList = scalarWithExpr znl (N zero) // zero expr vector
550     List.map2 (fun mc n -> scalarWithExpr n mc) el vl
551     |> List.fold (fun a b -> (List.map2 (+) a b)) zeroExprList
552
553 // Creates a vector of n variables
554 let rec charVector n =
555     match n with
556     | _ when n <= 0 -> []
557     | _ -> X (char (n - 1)) :: charVector (n - 1)
558
559 // inserts an enviroment into a vector

```



```

560 let rec vectorEnv n env =
561     match n with
562     | _ when n <= 0 -> V([], C)
563     | _ ->
564         let (V(nl, _)) = vectorEnv (n - 1) env
565         Map.find (char (n - 1)) env :: nl |> vector
566
567 // solves the system of lineary equations
568 let rec solveEquations el bl cl =
569     match el, bl, cl with
570     | [], [], [] -> Map.empty
571     | e::es, b::bs, _ when isZero e && isZero b ->
572         solveEquations es bs cl
573     | e::es, b::bs, c::cs ->
574         let env = solveEquations es bs cs
575         let (lhs, rhs) = isolateX (insertEnv e env) b c
576         Map.add (getVariable lhs) (getNumber rhs) env
577     | _, _, _ -> failwith "solveEquations: The number of equations and
578         variables must be the same"
579
580 // Solves the equation Ax = b
581 let matrixEquation A (V(nlb, ob)) =
582     let (D(r, c)) = dimMatrix A
583     if r <> List.length nlb || ob = R then
584         failwith "matrixEquation: b must be a column vector or have same
585         length as rows of A"
586
587 // Perform row echelon form on the total matrix
588 let ef_t = correctOrder (rowEchelonForm <| extendMatrix A nlb) C
589
590 // Extract the last vector of the matrix
591 let (V(ef_b, _), M(ef_vl, o)) = extractlastVector ef_t
592
593 // Check if the system of equations has a solution
594 if hasFullRank ef_t then
595     failwith "matrixEquation: The system of equations has no solution"
596 elif not <| hasFullRank (M(ef_vl, o)) then
597     failwith "matrixEquation: The system of equations has infinite
598     solutions"
599
600 let varlist = charVector c
601 let b = scalarWithExpr (vector ef_b) (N one)
602 solveEquations (matrixProductExprList ef_vl varlist) b varlist |>
603     vectorEnv c
    
```

Listing 51: Matrix.fs