Danmarks Tekniske Universitet

Bachelor Projekt

TITLE

Jonas Dahl Larsen (s205829)

17. marts 2024

Indhold

1 Introduktion

I 2023 valgte Danmarks Tekniske Universitet at anvende Python som et hjælpeværktøj i deres grundlæggende matematikkursus ”01001 Matematik 1a (Polyteknisk grundlag)”. Python er et af de mest anvendte programmeringssprog [1], kun overgået af to sprog, der primært bruges sammen til at udvikle hjemmesider. Derfor har Python, med en række matematiske programudvidelser som SymPy [2], været et oplagt valg som programmeringssprog til det grundlæggende matematikkursus tilbudt af DTU.

Projektet vil undersøge, hvordan et funktionsprogrammeringssprog, kan gavne de studerendes forståelse af de grundlæggende matematiske koncepter. Formålet er at guide læseren gennem opbygningen af en række funktionsprogrammer baseret på grundlæggende matematik [3] og dermed illustrere anvendelser. Projektet beskriver en generel struktur til opbygning og anvendelse af et funktions programmeringsprogram. Der tages udgangspunkt i F# [4], men beskrivelserne af programmerne vil også kunne anvendes i lignende funktionsprogrammeringssprog.

Rapporten begynder med at forklare nogle Fundamentale koncepter inden for funktionsprogrammering samt metoder til validering af programmerne.

2 Fundamentale koncepter

2.1 Introduktion til Funktions Programmering

Det er forventet af læseren har kendskab til programmering, der gives derfor kun en kort beskrivelse af syntaks og notationen, så læser ikke bekendt med F# kan forstå de eksempler der løbende vil forkomme i rappoten.

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

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

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

Listing 1: Eksempel på Fakultet i F#

[1]https://www.statista.com/statistics/793628/worldwide-developer-survey-most-used-languages/
[2]https://www.sympy.org/en/index.html
[3]https://mat1a.compute.dtu.dk/intro.html
[4]https://fsharp.org/

```python
// Fakultet i Python
def factorial(n):
    if n == 0:
        return 1
    elif n > 0:
        return n * factorial(n - 1)
    else:
        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 input-argument 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 genkende på, separeret med '|'. Resultatet for funktionen vil blive koden som er eksekveret efter '→'.

2.1.1 Typer

Typer er tildelt ved kompilering, i modsætning til python som kør det under kørsel. Alle udtryk funktioner inkluderet har en type i F#, typen for Listing factorial1 er $int \rightarrow int$. Det er derfor ikke muligt at kalde funktionen med et argument der ikke er af typen int. Typen for funktionen skrives som $Factorial : int \rightarrow int$. Vi kan dermed formulere følgende omkring typer[5]:

$$f : T_1 \rightarrow T_2$$
$$fe : T_2 \iff e : T_1$$

Er en funktion kaldt med et argument der ikke passer til funktionens type, gives en fejlmeddelelse. Derudover kan en type også være bestående at en tuple af typer.

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

En tuple som kun består af to typer, kaldes et par.
I F# er det ikke nødvendigt at anvende parenteser som i andre programmeringssprog. De vil derfor kun anvendes hvor det er nødvendige, gennem rapporten. Når man laver en tuple er det nødvendigt. Givet en funktion $g : T_1 \rightarrow T_2 \rightarrow T_3$ betyder den tager et udtryk af typen T_1 som giver en funktion af typen $T_2 \rightarrow T_3$ hvor evalueringen af funktionen giver T_3

2.2 Signatur filer og implementerings filer

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

[5] s14 FPU F#

2.3 Overloading operatorer

I F# er det muligt at overskrive standard operatorer, så man kan anvende dem på egne typer. Det vil igennem rapporten blive anvendt til at definere matematiske operationer på de typer som vi kommer til at bygge.

2.4 Property Based Testing

Property Based Test (PBT) er en teknik til at teste korrekthed af egenskaber som man ved altid skal være opfyldt. En PBT test generer en række tilfældige input til en funktion og tester om en egenskab er opfyldt. Hvis en egenskab ikke er opfyldt, vil PBT give et eksempel på en fejl. De matematiske studerende på DTU, begynder med at lære om logik. I den forbindelse lærer man at en udsagnslogisk formel er gyldig (tautologi) hvis den altid er sand. Der eksistere mange teknikker til at vise at en udsagnslogisk formel er gyldig, i DTU's matematiske kursus lærer man at anvende sandhedstabellen. De viser hvordan 2 er gyldig.

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

Vi vil også kunne anvende PBT til at undersøge om (2) er gyldig, ved at udtrykke egenskaben som en funktionen af P, Q og R se Listing 3 3.

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

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

Listing 3: PBT af (2), for at undgår shortcircuting har begge sider af lighedstegnet omgivet af parenteser.

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

Listing 4: Output ved PBT af (2)

Check.Quick er en del af "FsCheck"biblioteket, den tager en funktion som argument, og generere en række tilfældige input til funktionen på baggrund af funktionens type. Hvis funktionen returnere "true"for alle input, vil testen lykkedes. Hvis funktionen returnere "false"for et input, vil testen fejle og give et eksempel på et input der fejlede. I Listing 3 er der anvendt "Check.Quick"til at teste om (2) er gyldig. Funktionen "Check.Quick"returnere "Ok, passed 100 tests."hvilket indikerer at (2) er gyldig. Det vigtigt her at forstå dette ikke er det samme som at bevise at den er gyldig, da ikke alle muligheder er blevet testet. I dette tilfælde kan vi regne sandsynligheden for hvor vidt alle kombinationer er testet. $P, Q, R \in \{True, False\}$, derfor er der $2^3 = 8$ mulige kombinationer for input til funktionen. Hvis vi antager at alle kombinationer er lige sandsynlige, er sandsynligheden for at alle kombinationer er blevet testet $1 - (\frac{7}{8})^{100} = 0.999998$. Derfor er det meget sandsynligt at (2) er gyldig.
Det vil generelt ikke være muligt at regne denne sandsynlighed, da vi senere vil anvende PBT til at teste funktioner der tager argumenter som ikke har et endeligt antal kombinationer. Dog vil

det stadig give en god indikation hvorvidt en egenskab er overholdt. I nogle tilfælde vil det være en fordel at opskrive en PBT før implementeringen af en funktion som man ved skal overholde en egenskab, på den måde anvende Test Driven Development (TDD) [6] til at teste om ens egenskab forbliver overholdt, under implementering.

3 Symbolske lignings udtryk

Det ønskes at kunne repræsentere simple ligninger som en type i F#. Vil derfor gennemgå en del teori og funktion som er nødvendige for at kunne dette. Det vil give os et grundlæggende fundament for at kunne udføre matematiske evalueringer som differentiering i F#. Som de fleste andre programmer har F# kun float og int som kan repræsentere tal. Derfor vil vi begynde med at definere et mondul som indeholder en type for tal. Tanke gangen her at gennemgå en opbygning af en måde at kunne repræsentere ligninger samt simplificere dem. Vi begrænset os selv til at kun have matematiske operationer som addition, subtraktion, negation, multiplikation og division.

3.1 Tal mængder

Vi begynder med opbygningen af et mondul som kan repræsentere tal mængder. Typen for tal, består af tre konstruktører, for henholdvis heltal, rationale tal og komplekse tal. Dog er mondulet lavet med henblik på at kunne udvides med flere typer af tal. Måden resten af programmet er lavet på, gør de eneste grav til tal er at der er definerede matematiske operationer i form af addition, subtraktion, negation, multiplikation og division. Samt at tallet inden for addition og multiplikation er associative. Dette gælder blandt andet ikke for en vektor, derfor vil vi senere betragte at udvide programmet med en type for vektorer. En udvidelse kunne være for reele tal, som kan håndtere "floating point errors"[7], men for ikke at komplicere programmet vil vi i denne opgave ikke betragte floats.

3.1.1 Rationelle tal Mondul

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

```
     [language={FSharp},
     label={type_rationel},
     caption={Typen for rationelle tal}]
type rational = R of int * int
```

Nedestående er der givet en signatur fil for rational mondulet 5. i Implementerings filen overloades de matematiske operatorer, ved hjælp af de klassiske regneregler for brøker[8].

```
module rational

[<Sealed>]
```

[6] https://en.wikipedia.org/wiki/Test-driven_development
[7] https://en.wikipedia.org/wiki/Floating-point_error_mitigation
[8] https://en.wikipedia.org/wiki/Rational_number

```
type rational =
    static member ( ~- ) : rational -> rational
    static member ( + )  : rational * rational -> rational
    static member ( + )  : int * rational -> rational
    static member ( - )  : rational * rational -> rational
    static member ( - )  : int * rational -> rational
    static member ( * )  : int * rational -> rational
    static member ( * )  : rational * int -> rational
    static member ( * )  : rational * rational -> rational
    static member ( / )  : rational * rational -> rational
    static member ( / )  : int * rational -> rational
    static member ( / )  : rational * int -> rational
    static member ( / )  : int * int -> rational

val make          : int * int -> rational
val equal         : rational * rational -> bool
val posetive      : rational -> bool
val toString      : rational -> string
val isZero        : rational -> bool
val isOne         : rational -> bool
val isInt         : rational -> bool
val makeRatInt    : rational -> int
val greaterThan   : rational * rational -> bool
val isNegative    : rational -> bool
val absRational   : rational -> rational
```

Listing 5: Signatur filen for rational mondulet

For at kunne sammenligne, men også for nemmere at undgå for store brøker, vil alle rationelle tal blive reduceret til deres simpleste from. Dette kan gøres ved at finde den største fælles divisor (GCD) [9]. Der udover er det vigtigt at være opmærksom på man ikke foretager nul division. Derfor vil implementerings filen kaste en "System.DivideByZeroException"hvis nævneren er eller bliver nul. Signatur filen indeholder en række funktioner som bliver anvendt af andre filer.

3.1.2 Komplekse tal Mondul

3.1.3 Tal Mondulet

Vi har nu beskrevet en måde at kunne repræsentere bruger definere tal på ved brug af typer i F#. Det vil derfor være oplagt at have en type som indeholder alle de typer tal vi ønsker at kunne anvende i de matematiske udtryk vi er ved at opbygge. Fordelen ved at samle dem til en type er at vi kan lave en række funktioner blandt andet matematiske operationer som kan anvendes på alle type tal. Vi begynder med at definere en type for tal 6, som indeholder konstruktører for de tal typer vi har definerede samt en for heltal.

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

Listing 6: Typen for Number

[9] https://en.wikipedia.org/wiki/Greatest_common_divisor

Betragtes signatur filen for Number mondulet 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.

```
module Number
// open rantionalAndComplex
open rational
open complex

type Number =
    | Int of int
    | Rational of rational
    | Complex of complex
    with
    static member ( + )  : Number * Number -> Number
    static member ( - )  : Number * Number -> Number
    static member ( * )  : Number * Number -> Number
    static member ( / )  : Number * Number -> Number
    static member ( ~- ) : Number -> Number




val zero         : Number
val one          : Number
val two          : Number
val isZero       : Number -> bool
val isOne        : Number -> bool
val isNegative   : Number -> bool
val absNumber    : Number -> Number
val greaterThan  : Number -> Number -> bool
val tryReduce   : Number -> Number
val toString     : Number -> string
val conjugate         : Number -> Number
```

Listing 7: Signatur filen for Number mondulet

Ved implementeringen af de matematiske operationer, hvis der eksistere en konstruktør i Number, der repræsentere en tal mængde hvor alle andre konstruktører er delmængder af denne mængde. Er det muligt at definere en enkelt funktion som kan udføre alle binærer operationer. Som et eksempel er funktionen 8 givet, som tager to tal og en funktion i form af den ønskede binærer operation som parameter. Funktionen vil derefter matche på de to tal og anvende den operation på de to tal.

```
// makeRational: Number -> rational
let makeRational a =
    match a with
    | Int x       -> make(x, 1)
    | Rational x  -> x

```

```
// operation: Number -> Number -> (rational -> rational -> rational) -> Number
let operation a b f =
    f (makeRational a) (makeRational b) |> Rational
```

Listing 8: Number.operation funktionen

Det vil her til være oplagt på alle de matematiske operationer at anvende en funktionen til at forsøge at konvenere tal typen til den simpleste talmængde, som i vores tilfælde er heltal. Dette er gjort ved at anvende funktionen tryMakeInt på alle de matematiske operations overloadnings 9.

```
// tryMakeInt: Number -> Number
let tryMakeInt r =
    match r with
    | Rational a when isInt a -> Int (makeRatInt a)
    | _ -> r

type Number with
  static member (+) (a, b) = operation a b (+) |> tryMakeInt
  static member (-) (a, b) = operation a b (-) |> tryMakeInt
  static member (*) (a, b) = operation a b (*) |> tryMakeInt
  static member (/) (a, b) = operation a b (/) |> tryMakeInt
  static member (~-) (a)   = neg a |> tryMakeInt
```

Listing 9: Overloadnings funktionerne for Number

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

3.2 Matematiske ligninger

3.2.1 Polsk notation

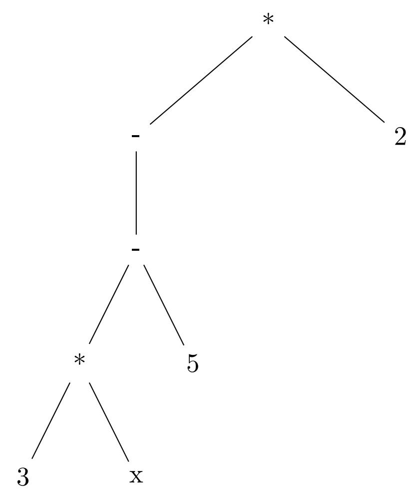
Matematiske ligningsudtryk som vi normalt kender dem er skrevet med infix notation I infix notation skrives en binær operator mellem to operandere, kende tegnet for sproget er at det indeholder parenteser samt præcedens regler. Dette gør det generalt kompliceret at evaluere og håndtere matematiske udtryk i et programmeringssprog. Derfor er det mere oplagt at kunne anvende polsk notation (prefix) istedet, hvor operatoren skrives før operandere eller omvendt polsk notation (postfix). Da de hverken indeholder parenteser eller præcedens regler [10].

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

[10] https://en.wikipedia.org/wiki/Polish_notation

3.2.2 Ligninger som træer

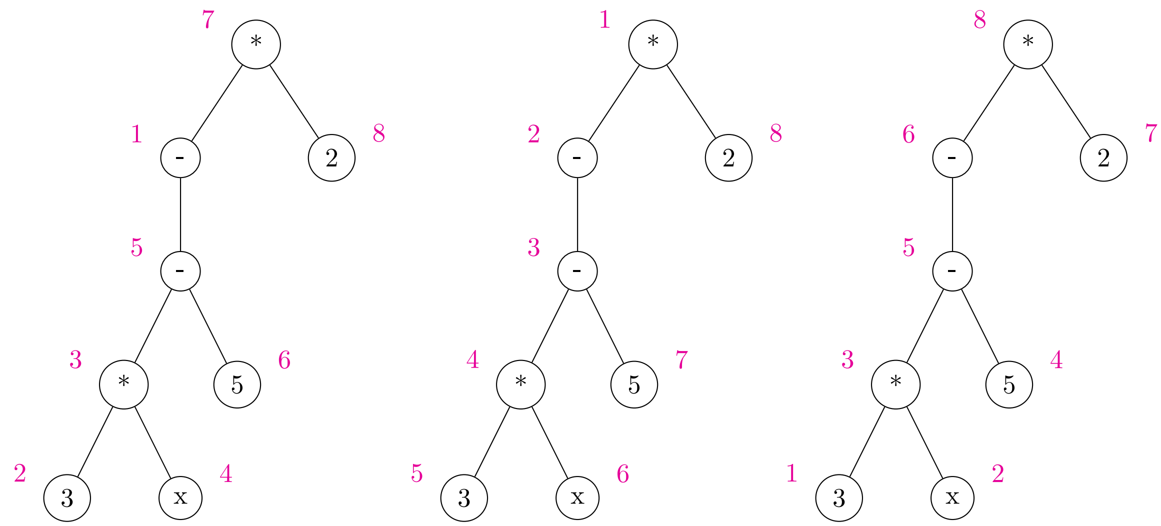
Et matematisk udtryk kan repræsenteres som et binært træ, hvor bladene er operander i det anvendte matematiske rum og alle andre noder er operationer. Som eksempel kan udtrykket $-(3 \cdot x - 5) \cdot 2$ repræsenteres som følgende træ 1.



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

Det skal bemærkes der er forskel på den unære og binære operator $-$ i træet, den unære betyder negation og den binære er subtraktion. Givet et binært træ for en matematisk ligning, vil det være muligt omdanne dem til infix, prefix eller postfix notation. Dette kan gøres ved at anvende modificeret Inorder, Preorder eller Postorder Traversal [11], algorithmerne er illustreret i figut 2.

[11] `https://www.geeksforgeeks.org/tree-traversals-inorder-preorder-and-postorder/`



(a) Modificeret Inorder Travesal

(b) Preorder Traversal

(c) Postorder Traversal

$$
\begin{aligned}
\text{Modificeret Inorder Traversal:} \quad & -(3 \cdot x - 5) \cdot 2 \\
\text{Preorder Traversal:} \quad & \cdot - - \cdot 3\, x\, 5\, 2 \\
\text{Postorder Traversal:} \quad & 3\, x\ \cdot 5\ - -2\, \cdot
\end{aligned}
$$

Figur 2: Træet fra 1 med forskellige travesal metoder

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

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 en symbolsk ligning. Den modificeret Inorder Traversal anvendes til at kunne visualisere ligningsudtryk i infix notation.

3.2.3 Ligningsudtryk mondulet

Efter at have lavet et mondul til at repræsentere tal mængder, er det nu muligt at kunne implementere et mondul som kan repræsentere ligningsudtryk. Vi begynder ved at definere en polymorf type for et ligningsudtryk 10. Typen består af en række konstruktører, som repræsentere de matematiske operationer vi ønsker at kunne anvende, samt en konstruktør N for at repræsentere matematiske strukturer og X for at repræsentere variable. Vi begynder med at betragte matematiske strukturer, i form af de tal mængder vi har definerede som Number 6. Vi vil senere betragte hvordan det vil være muligt at udvide programmet til at kunne repræsentere flere matematiske strukturer så som vektorer.

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

Listing 10: Typen for Expr

Expr<’a> typen er dermed en polymorf type, hvor ’a er typen for den matematiske struktur hvor vi kan lave brugerdefinerede matematiske operationer. Et exemplar på en Expr<Number> er givet i 11.

```
> 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 11: $-(3 \cdot x - 5) \cdot 2$ som et udtryks træ. Funktionen tree bliver beskrevet i 3.2.4.

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

```
module Expression
open Number

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

val eval : Expr<Number> -> Map<char,Number> -> Number
```

Listing 12: Signatur filen for Expression mondulet

De overloadede matematiske operatorer i Expressions, laver overflade evalueringer samt simplifikationer på deres respektive argumenter. Overfalde evalueing vil sige at de individuellen funktioner kun betragter de to øverste niveauer på de lignings udtryk træer de tager som input, mullige implementeringer af addition og multiplikation er givet i 13.

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

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

Listing 13: Addition og multiplikation af to ligningsudtryk

3.2.4 Generering af Ligningsudtryk

3.3 Evaluering af ligningsudtryk

3.3.1 PBT af evalueringen

3.4 Simplifikation af Ligningsudtryk

3.4.1 PBT af simplifikationen

3.5 differentiering af Ligningsudtryk