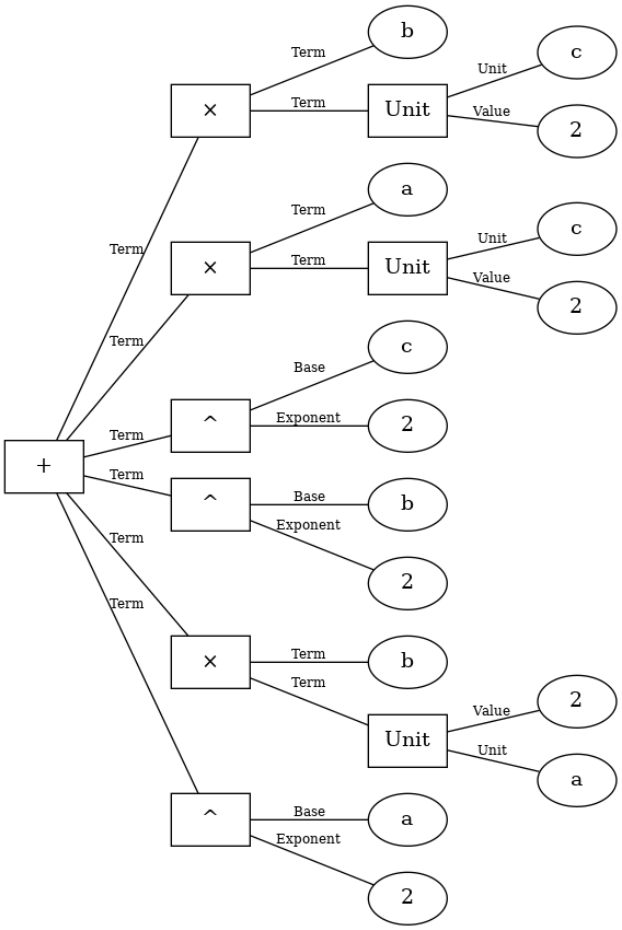
Tolkning och förenkling av matematiska uttryck i form av en symbolhanterare

Matstoms, Axel Janković, Luka Matstoms, Ivar

9 maj 2018



Handledare: Mikael Rydfalk
Medbedömare: Thomas Lejdå

Abstract

The purpose of this project is to look at some of the techniques used by sites such as WolframAlpha to simplify algebraic expressions and solve particular kinds of equations. To research this we wrote an application in Python with some of the features of such sites. We chose to work with Python due to its simplicity and due to the fact that everybody in the group knew it before the project even started. During the span of this project we were able to develop an application which is able to take input and construct an abstract syntax tree which is then simplified. The application is able to handle many of the goals that were set in the beginning of this project, including addition and multiplication of polynomials correctly, although not always fully simplified. The application is also able to solve polynomial equations up to and including the second degree. With time the application could easily be expanded to cover more aspects of what we set out to do such as polynomial division and solving polynomial equations of higher grades.

Keywords: Abstract syntax tree; Python; Algebraic Expressions

Innehåll

1 Inledning

Eftersom vi alla i gruppen delar en stor passion för programmering ville vi göra ett arbete som involverar både programmering och ett naturvetenskapligt ämne. Vi övervägde flera olika idéer, bland annat en realtids fysikmotor, implementationer av matematiska funktioner (såsom sinus, exponenter) på hårdvarunära nivå samt idén som valdes: att skriva en symbolhanterare vid namn GAMLP (**G**a-**A**rbete **M**atematikbib**L**iotek i **P**ython). En symbolhanterare, eller CAS, Computer Algebra System, är ett datorprogram som kan tolka och lösa ekvationer, exempelvis $5x + 10 = 20$. En annan funktion som en symbolhanterare har är förmågan att förenkla uttryck, exempelvis $(x + 2)(x - 3) \Leftrightarrow (x^2 - x - 6)$. Skillnaden mellan en vanlig miniräknare och en symbolhanterare är att symbolhanterare förstår algebra samt att en symbolhanterare kan hantera variabler, inte bara värden.

Syftet med arbetet är att undersöka hur verktyg såsom WolframAlpha fungerar, dvs. hur inmatad data tolkas, hanteras och förenklas och utifrån det skriva ett bibliotek som kan förenkla uttryck och lösa ekvationer. Arbetet har utgått från följande frågeställningar:

- Hur kan inmatad data representerad som trädstruktur tolkas och arbetas med av ett program?
- Hur kan uttryck förenklas m.h.a. dessa trädstrukturer?
- Hur kan algoritmer användas för att lösa ekvationer?

2 Teori

Detta teoriavsnitt beskriver hur uttryck representeras på ett sätt som är enklare för programmet att hantera samt själva strukturen på programmet. Multinomialsatsen kommer också förklaras då den används för att utveckla multinom.

2.1 Träd som datastruktur

Inom datavetenskap beskrivs träd som en form av graf som används för att representera hierarkisk data. Ett träd inom matematiken består av noder som är sammansatta på så vis att varje nod kan innefatta ett eller flera barn, som i sig också är noder. I det här dokumentet kallas barnen för subnoder. En subnods "förälder" kallas just så: förälder. Noder måste ha ett värde eller representera en operation, men det finns dock inget krav på att en nod måste

ha subnoder. Dessa noder kallas löv. Om en nod inte har en förälder betyder det att denna nod är högst upp i trädet. Denna nod kallas för rotnod eller bara rot.

2.1.1 Abstrakt syntaxträd

Abstrakta syntaxträd är träd som representerar syntaxen av t.ex. ett språk. Operatorprioritet och parenteser representeras av trädets struktur, vilket gör det enklare för datorprogram att hantera (Ranta, 2011). Abstrakta syntaxträd används oftast inom datavetenskap för att strukturera programmeringskod så en kompilator kan omvandla källkoden till ett exekverbart program. Men på grund av likheterna mellan programmeringsspråkens syntax och matematiska ekvationer användes abstrakta syntaxträd i detta arbete.

2.2 Klasser och Objekt

En viktig del av struktureringen av noder handlar om att en nod kan vara en utvidgad (eng. extended) typ av en annan nod. Det vill säga, istället för att på nytt skapa varje nod kan mancd skapa en allmän nod för t.ex. operatörer, och sedan utvidga den för varje sorts operation. Detta kallas för Objektorienterad Programmering (**O**bject **O**riented **P**rogramming, OOP).

2.2.1 Objektorienterad programmering

Objektorienterad programmering fungerar genom att man för varje föremål, struktur, etc. man vill beskriva i ett program har en s.k. klass. En klass är en abstrakt modellering av ett objekt; dvs. den innehåller definitioner för egenskaper och funktioner. Sedan används klassen för att skapa objekt som kan fylla på med värden på attributerna och faktiskt utföra de beskrivna handlingarna. Ett exempel är att föreställa sig klassen Bil. I denna klass är det definierat att varje bil har t.ex en färg, maxhastighet, osv. Den har också vissa handlingar (alt. metoder) beskrivna; man kan köra bilen, tanka den, osv. Detta är beskrivningen av klassen Bil, och det är väldigt viktigt att komma ihåg att klassen, dvs. definitionen av bilen i sig ej är en bil, mer som en ritning för bilen. Istället skapas en instans av klassen bil; alltså kan man säga att objektet är skapat med klassen som ritning. Vi återvänder till Bil-exemplet och skapar ett objekt; en bil med färgen blå och maxhastighet 200 km/h. Denna teoretiska bil kan nu köras, tankas osv.

2.2.2 Subklasser och arv

Ett användbart redskap inom objektorienterad programmering är arv. För att fortsätta med bil-exemplet kan vi säga att klassen Bil är en subklass (dvs ärver från) klassen Fordon. Klassen Bil har då alla attributer och metoder som klassen Fordon har samt attributer och metoder som läggs till i definitionen av Bil. Klassen fordon kan exempelvis ha attributerna kan-flyga och bränsle-typ. Därmed kan en instans av klassen Bil anta värden för dessa attributer.

2.3 Multinomialsatsen

För att utveckla multinom (e.x $(a+b+c)^3$) används multinomialsatsen. Som man kan ana på namnet liknar multinomialsatsen binomialsatsen. Skillnaden är att multinomialsatsen är ett mer generaliserat sätt att beskriva utveckling av polynom, och är alltså ej begränsad till endast binom. Satsen lyder (ProofWiki, 2016):

$$(a_1 + a_2 + \ldots + a_m)^n = \sum_{k_1+\cdots+k_m=n} \binom{n}{k_1, k_2, \ldots, k_m} a_1^{k_1} \cdot \ldots \cdot a_m^{k_m}$$

Den behöver dock lite förklaring. Med $\sum_{k_1+\cdots+k_m=n}$ menas alla möjliga kombinationer med återläggning av exponenter i $a_1^{k_1} \cdot a_2^{k_2} \cdot \ldots \cdot a_m^{k_m}$ där $\sum_{i=0}^{m} k_i = n$ uppfylls. Vidare behöver $\binom{n}{k_1,k_2,\ldots,k_m}$ förklaras. Detta kallas för *multinomialtal* eller *multinomialkoefficient*, som i sig en generalisering av binomialkoefficienten $\binom{n}{k}$. Multinomialkoefficienten kan skrivas om på detta vis.

$$\binom{n}{k_1, k_2, \ldots, k_m} = \frac{n!}{k_1! \cdot k_2! \cdot \ldots \cdot k_m!}$$

Ett exempel på binomialsatsen lyder:

$$(a+b+c)^3 = a^3 + 3a^2b + 3a^2c + 3ab^2 + 6abc + 3ac^2 + b^3 + 3b^2c + 3bc^2 + c^3$$

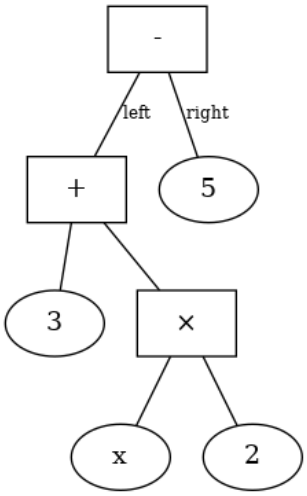
3 Metod

3.1 Implementation av syntaxträd

Trädet utgår från en rotnod. Om trädet representerar en ekvation är denna nod alltid ett likhetstecken. Likhetsnoden kan bara anta två subnoder, en för VL och en för HL. Därefter kan leden byggas upp av olika slags noder. Om trädet representerar ett uttryck kan rotnoden vara precis vilken slags nod som helst bortsett från likhetstecken, då skulle ju trädet representera en ekvation!

3.2 Noder

Varje nod måste anta ett värde. Med detta menas ett numeriskt värde (e.x. 5 eller 10), eller en operator (e.x. $+$, $-$, $\cdot$, $\div$, $\hat{}$). Alla nummernoder och okända blir löv i trädstrukturen. Alla andra sorters noder kräver subnoder, eftersom man givetvis inte kan bygga upp ett uttryck med enbart operatörer. Se Figur 1 för ett exempel på hur ett uttryck kan representeras med hjälp av en trädstruktur.

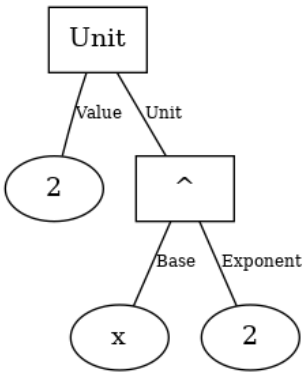


Figur 1: Visuell representation av $(2x + 3) - 5$ som ett abstrakt syntaxträd.

Operatörsnoder Operatörsnoder är noder som representerar operationer och kräver därför subnoder. Exempel på operatörsnoder är additionsnod, som representerar addition mellan två eller flera noder och divisionsnod, som representerar division mellan två noder.

Homogena och inhomogena operatörsnoder Homogena operatörsnoder representerar operationer där ordningen av värdena ej spelar någon roll, exempelvis $a + b + c = b + c + a$. Detta betyder att homogena noder inte behöver ha en strikt vänsternod och högernod, utan kan istället lagra subnoderna som en lista. Därmed kan en operatörsnod ha ett godtyckligt antal subnoder. Inhomogena operatörsnoder representerar operationer där ordningen spelar roll, som t.ex. minus ($a-b \neq b-a$). Detta betyder att inhomogena operatörsnoder endast kan ha två subnoder, eftersom måste ha en strikt defenition för vänster och höger nod.

Enhetsnoder Eftersom programmet ska kunna hantera ekvationslösning, måste okända variabler kunna representeras (e.x. $5x + 10 = 20$. Här är x, en okänd varaibel.) För att representera variabler används en speciell typ av nod: en så kallad enhetsnod. Denna nod antar ett värde för koefficienten och en representation av den okända variabeln, en så kallad enhet. Denna enhet består av själva variabeln och vilken grad som den representeras i. Se Figur 2 för en visuell representation av $2x^2$ som en enhetsnod.



Figur 2: $2x^2$ representerat som en enhetsnod med enheten x^2 och värdet 2. Med *Unit* menas enhet.

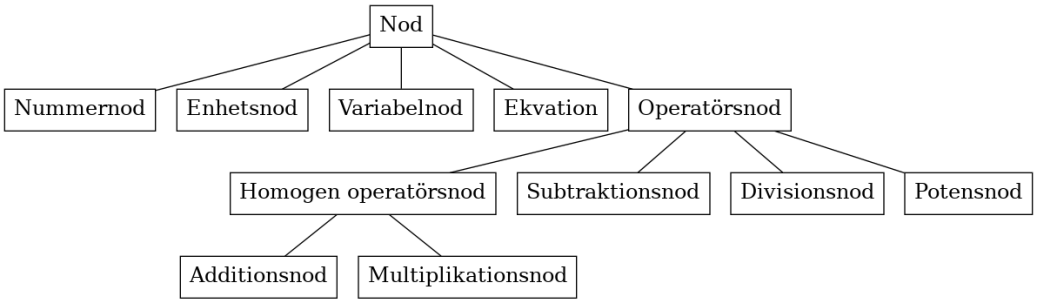
3.2.1 Noder representerade som klasser

Alla noder är subklasser av klassen Nod. Nodklassen innehåller definitioner för funktioner som används av och är gemensamma för alla typer av noder (se Tabell 1 för en lista). Alla operatörsnoder är subklasser av klassen operatörsnod. Operatörsnoden har dock ingen funktion i sig, utan används för att enklare urskilja vilka noder som är operatörsnoder och vilka som inte är. Därefter grupperas additionsnodens och multiplikationsnodens klass som var sin subklass av klassen honomgen operatörsnod. Detta gjordes eftersom både

multiplikationsnoden och additionsnoden är homogena noder.

Namn	Användning
`hash_node`	Returnerar ett unikt nummer för just denna nod som används för att identifiera den.
`get_children`	Returnerar subnoder.
`get_int_value`	Om noden går att räkna ut och resultatet är ett heltal returnera heltalet.
`latex`	Returnerar en latex representation.
`simplified`	Bygger upp en nod som är en förenkling av sig själv.
`eval`	Räknar ut värdet på en nod.
`eq`	Jämför noder (samma struktur, variabler etc.)
`flattened`	"plattar" ut ett träd
`eval`	räknar ut värdet av en nod
`formatted`	Returnerar en text representation av noden.

Tabell 1: Funktioner och dess användning i basklassen nod.



Figur 3: Diagram av nodklassen samt dess alla subklasser.

3.3 Tolkning av inmatade ekvationer

För att läsa in uttryck på normal form och för att enkelt omvandla de till abstrakta syntaxträd krävs en lexikalisk analys av den inmatade datan som sedan tolkas av en parser. Att hantera lexikalisk analys och att utveckla en parser ligger utanför projektet. Därför användes biblioteket purplex, som utför denna lexikaliska analys och har en inbyggd parser.

3.3.1 Lexikalisk analys

En lexer separerar den inmatade datan till s.k. tokens baserat på fördefinierade mönster och bygger upp en array (lista) baserad på dessa tokens. Med

andra ord letar en lexer efter ett sorts mönster i indatan och fyller på array-en med motsvarande tokens. Tokens känns igen med hjälp av mönstermatchningsspråket regex. Nedan visas en tabell över vilka mönster som letas efter och vad de ersätts med i listan i den lexikaliska analysen. Exempel: $5x + 10 = 20 \Rightarrow$ ([värde, 5], [okänd, x], [addition], [värde, 10], [likhetstecken], [värde, 20])

Mönster	**Tokens**
Nummer	Nummernod
Bokstav	enhetsnod med värdet 1
Nummer och bokstav	Enhetsnod med värdet nummer
`+`	Additionsnod
`-`	Subtraktionsnod
`*`	Multiplikationsnod
`/`	Divisionsnod
`^`	Potensnod
$\surd$	Kvadratrotsnod
`=`	Ekvation

Tabell 2: Vilka mönster som letas efter i indatan och vad de ersätts med i den lexikaliska analysen.

3.3.2 Parser

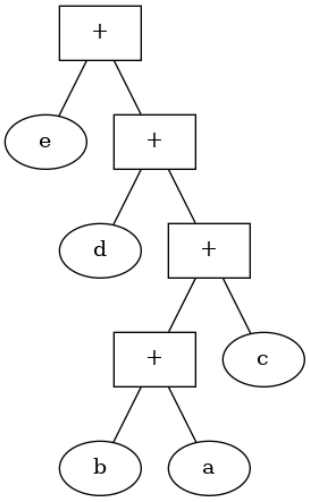
Parserns (svenska - tolk) uppgift i vårt projekt är att bygga upp ett syntaxträd av listan som skickades över från den lexikaliska analysen. Eftersom det är då strukturen på ekvationen byggs upp är det viktigt att räknesätten prioriteras på rätt sätt. Ordningen på räknesätten är förutbestämt i regler som parsern följer. När tokens definieras anges också vilken ordning de ska prioriteras.

3.4 Förenkling

Operatörsnoder vars subnoder endast består av numeriska värden räknas helt enkelt ihop med den operatorn som noden representerar. Då kommer denna operatörsnod ersättas med en nummernod som antar det ihopräknade värdet.

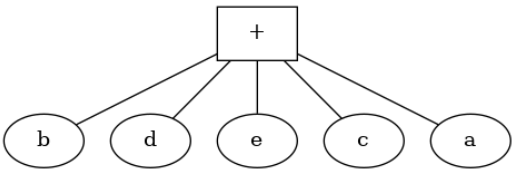
3.4.1 Homogena operatörsnoder

När man matar in ett uttryck i programmet skapas först ett träd där alla operatörsnoder har två subnoder, se Figur 4.



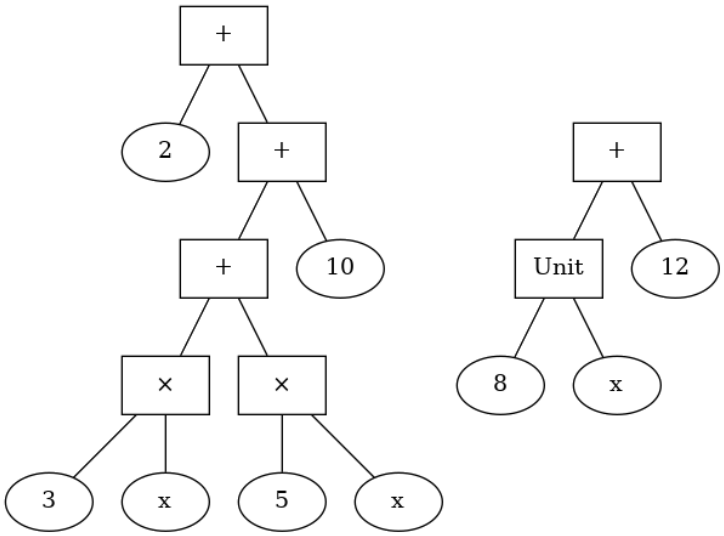
Figur 4: $(a + b + c + d + e)$ tolkat av programmet.

Första steget i förenklingen är att "platta ut" ett träd av homogena noder av samma typ till en enda nod, se Figur 5. Subnoder med ett numerisk värde separeras från övriga subnoder och räknas ihop. Övriga subnoder som sammanfogas i största möjliga utsträckning. Ett exempel på en sammanfogning är $x + x = 2x$. Exakt hur subnoder sammanfogas beror på nodens typ.



Figur 5: $(a + b + c + d + e)$ efter första steget av förenkling av homogen nod

Additionsnod Nummeriska värden summeras, $5 + 10 \Rightarrow 15$ och enhetsnoder kombineras genom att dess koeficient sätts baserat på antalet, exempelvis $x + x + x = 3x$, eller $5x + 3x + 10 + 2 = 8x + 12$, se Figur 6.



Figur 6: $5x + 3x + 10 + 2$ förenklas till $8x + 12$

Multiplikationsnod Om additionsnoder multipliceras kommer dessa lösas ut genom att kombinationer av additionsnodernas subnoder adderas. Därefter multipliceras resten av multiplikationsnodernas subnoder med summan av kombinationerna.

$$(a + b) \cdot (c + d) \cdot ef$$

$$\Leftrightarrow$$

$$(ac + ad + bc + bd) \cdot ef$$

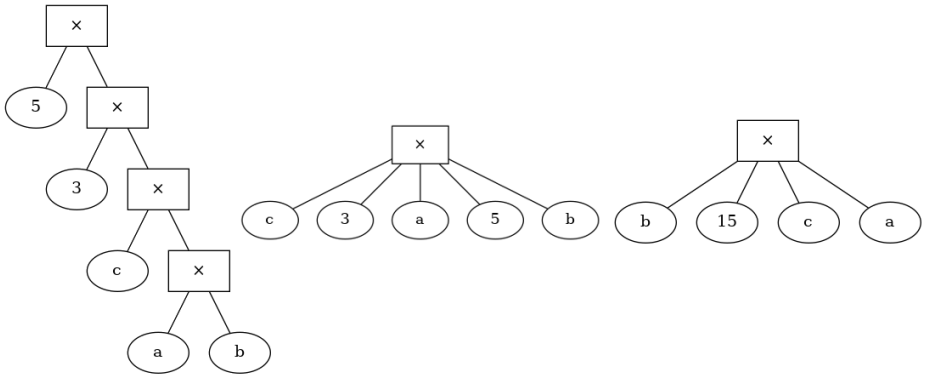
$$\Leftrightarrow$$

$$acef + adef + bcef + bdef$$

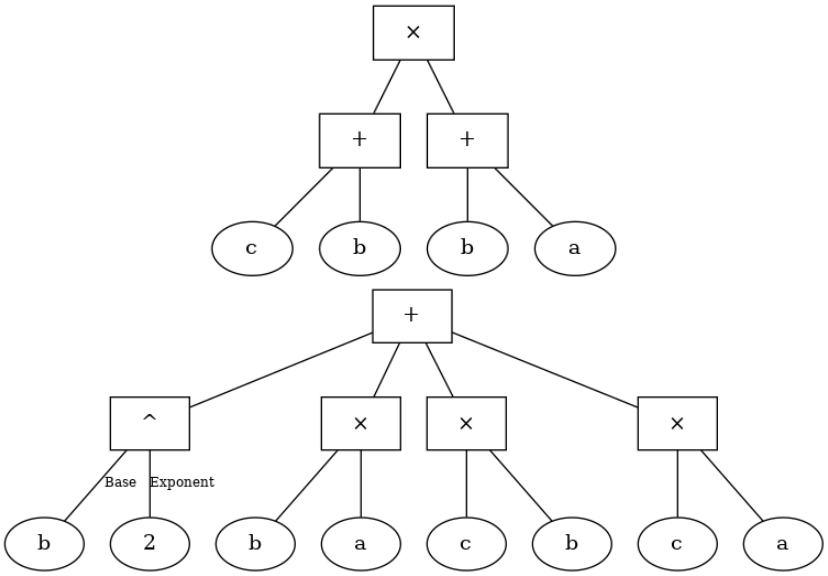
Enhetsnoder med samma okänd kombineras genom att skapa en ny enhetsnod där enheten blir produkten av de enheterna för noderna och koefficienten produken av koefficienterna (e.x. $5x \cdot 10x = 50x^2$). Enhetsnoder kan också kombineras med övriga noder genom att multiplicera in i enhetsnodens koefficient.

Om en potensnod multipliceras med en nod vars värde motsvarar potensnodens basvärde kan dessa slås samman genom att öka potensnodens exponentvärde med ett. Exempelvis $a^3 * a = a^4$.

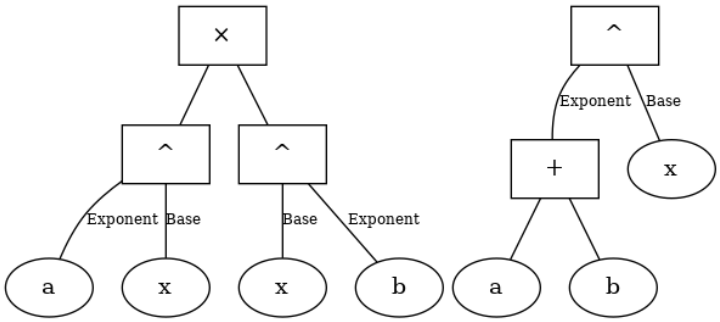
Två eller flera potensnoder kan multipliceras genom att exponenterna adderas. Detta gäller också för enhetsnoder om basen är samma. Exempelvis $5^3 * 5^2 = 5^5$ och $a^3 * a^2 = a^5$



Figur 7: $(a \cdot b \cdot c \cdot 3 \cdot 5)$ förenklas i tre steg, a,b,c lyfts upp till samma nod och talen separeras till sin egen nod, tillslut räknas talen ihop.



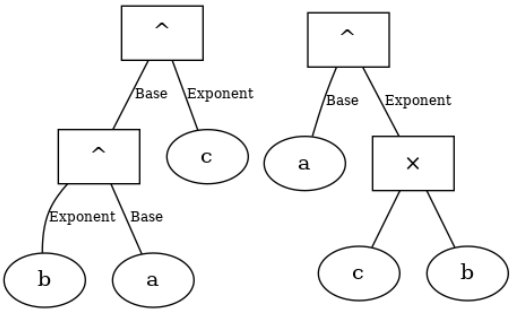
Figur 8: $(a + b) \cdot (c + b)$ förenklas till $(a \cdot b) + (b \cdot c) + (b^2) + (a \cdot c)$



Figur 9: $x^a \cdot x^b$ blir förenklat till x^{a+b}

3.4.2 Inhomogena noder

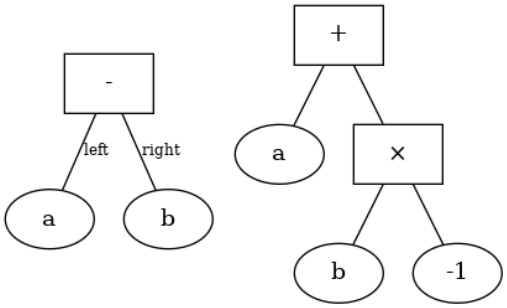
Potensnod Potensnoder innehåller två subnoder, basen och exponenten. En potensnod kan ha vilken sorts nod som helst som basnod (bortsett från ekvationsnod), exempelvis $(a + b + c)^3$, där basnoden är en additionsnod. Om basnoden är en additionsnod kommer hela potensnoden förenklas m.h.a. multinomialsatsen, annars kommer basnoden förenklas separat, innan potensnoden förenklas. En potensnod med potens noll förenklas till en nummernod med värdet ett och en potensnod med potensen ett förenklas till bara basnoden. Om potensnoder är staplade, dvs. basen av en potensnod består av en annan potensnod förenklas dessa till en enda potensnod vars exponentvärde består av produkten av de ursprungliga potensnodernas exponentvärden.



Figur 10: $\left(a^b\right)^c$ blir förenklat till a^{bc}

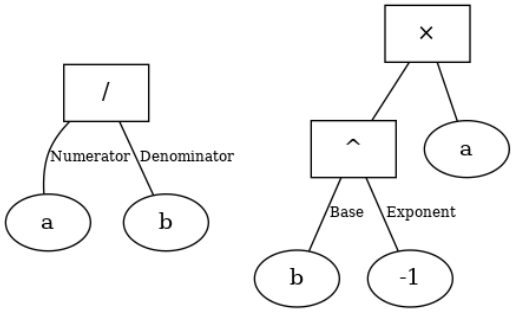
Subtraktionsnod Subtraktionsnoden är ej en homogen nod då ordningen på termerna spelar roll, $a - b \neq b - a, a \neq b$. Därmed måste subtraktionsnoden anta endast två subnoder; en för vänster term och en för höger term.

För att förenkla subtraktionsnoden används additionsnoden och multiplikationsnoden för att uppnå en minusoperation. Istället för att direkt räkna ut $a - b$ så skrivs uttrycket om till $a + ((-1) \cdot b)$. Därmed kan samma resultat uppnås med bara addition och multiplikation.



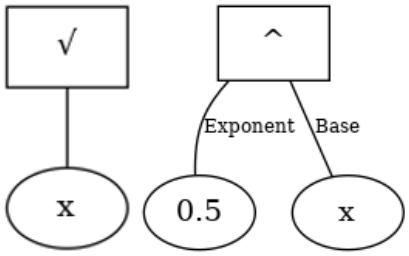
Figur 11: $a - b$ före och efter förenkling. Minus noden transformeras till en additionsnod med b ersatt med $b \cdot -1$

Divisionsnod På samma sätt som subtraktionsnoden skiver om uttrycket till en additionsnod så skriver divisionsnoden om uttrycket så att förenklingen istället kan hanteras av multiplikationsnoden och potensnoden. $\frac{a}{b} \Rightarrow a \cdot (b^{-1})$



Figur 12: $\frac{a}{b}$ förenklas till $a \cdot (b^{-1})$

Kvadratrotnoder För att förenkla kvadratrotsnoder skrivs de om till potensnoder. $\sqrt{x} = x^{\frac{1}{2}}$



Figur 13: $\sqrt{x}$ förenklas till $x^{\frac{1}{2}}$

3.5 Ekvationslösning

En ekvation löses genom att först subtrahera högerledet från båda led så att högerledet blir 0. Mer specifikt skapas en subtraktionsnod med båda leden som subnoder, som därefter förenklas.

Detta leder till att ekvationen kan lösas med en allmän metod som väljs beroende på vilken grad ekvationen är av. Vilken grad ekvationen är av bestäms enhetsnoden med högst potens.

3.5.1 Ekvationsmetoder

Dessa metoder används för att bestämma värdet på en enhetsnod från en ekvation.

Förstagradsekvationer $kx + m = 0 \Leftrightarrow x = \frac{-m}{k}$

Andragradsekvationer $ax^2 + bx + c = 0 \Leftrightarrow x = -\frac{b}{2a} \pm \sqrt{\frac{b^2}{(2a)^2} - \frac{c}{a}}$

Enkla exponensialekvationer $a \cdot x^b + k = 0 \Leftrightarrow x = (\frac{-k}{a})^{\frac{1}{b}}$

3.6 Bilder på träd

Bilderna på träden som används relativt ofta i denna rapport är också del av GAMLP, som har möjligheten att generera bilder av träd. Dessa genereras med hjälp av biblioteket *graphviz*. För varje träd går programmet igenom trädet uppifrån och ner och skapar listor med alla noder och kanter. Programmet skriver sedan kod med *dotsyntax*, som sedan tolkas av programmet *dot* för att generera en bild på trädet.

3.7 Källor

3.7.1 Struktur

Strukturen av projektet baserades till viss del på ett examensarbete vid namn *Constructing a Computer Algebra System Capable of Generating Pedagogical Step-by-Step Solutions* skriven av Dimitrij Lioubartsev (KTH, 2016). Syftet med detta examensarbete är att modifiera ett existerande bibliotek för symbolhantering, SymPy, så att det kan generera stegen som krävs för att kunna lösa en ekvation. Detta var givetvis ej syftet med vårt gymnasiearbete, men examensarbetet gick in i detalj hur en symbolhanterare kan fungera (kapitel 3). Denna text användes som källa då den kändes relevant till vårt arbete. Vi kunde använda källan för inspiration för hur arbetet skulle läggas samt som exempel på hur själva rapporten kunde se ut.

3.7.2 Multinomialsatsen

Förenkling av uttryck på formen $(x_1 + x_2 + ... + x_k)^m$ görs med multinomialsatsen. Då vi endast jobbat med ett specialfall av den här formen på uttryck, binomial, behövde vi hitta ett sätt att förenkla uttryck på den allmänna formen. Sidan om Multinomial Theorem på Proof Wiki var till stor hjälp och kunde även verifieras mot Encyclopædia Britannicas sida om samma ämne. Proof Wikis sida innehöll mer information och var till större nytta.

4 Resultat

Programmet har dessa funktioner:

- Representera inmatad data som syntaxträd
- Rekursivt förenkla operationer samt korrekt kunna hantera okända variabler för att förenkla uttryck.
- Lösa ut okända variabler från ekvationer genom att använda olika metoder för olika ekvationsgrad.

4.1 Vad kan biblioteket hantera

4.1.1 Förenkling av uttryck

Programmet strävar efter att förenkla ekvationen så den blir omskriven på polynomform, vilket kan vara fördelaktigt i vissa situationer, dock inte alla. Exempelvis för ekvationen $(x-3)\cdot(x-5) = 0$ är det uppenbart att rötterna

är $x = 3, x = 5$ men i polynomform $x^2 - 8x + 15 = 0$ blir ekvationen lite svårare att lösa. I det här fallet skulle programmet fortfarande kunna lösa ekvationen i polynomform, men om ekvationen var av tredje graden eller högre skulle detta inte vara fallet.

Uttryck	**Förenklas till**
$\frac{ac+bc}{c}$	$a + b$
$a + a$	$2a$
$a \cdot a$	a^2
$a - a$	0
$(a + b)^2$	$a^2 + 2ab + b^2$
$\frac{a}{a}$	1
$(a + b + \ldots)^k$	Utvecklas m.h.a. multinomialsatsen
$ak + am$	$a(k + m)$
$\frac{a+b}{c+d}$	Programmet kraschar

Tabell 3: Uttryck och vad de förenklas till (om möjligt)

4.1.2 Ekvationslösning

Programmet kan lösa relativt avancerade ekvationer. Programmet är i första hand utvecklat för att lösa polynomekvationer.

Allmänt uttryck	**Löser programmet ekvationen?**
$ax + b = 0$	Ja
$\frac{x}{a} = b$	Ja
$ax^2 + bx + c = 0$	Ja
$\frac{a \cdot x^2}{b} + \frac{x \cdot c}{d} + \frac{e}{f} = g$	Ja
$(x + a)(x + b) = c$	Ja
$\frac{(x+a)\cdot(x+b)}{c} + d = e$	Ja
$x^a + b = c$	Endast om $a > 0$
$(x + a) \cdot (x + b) \cdot (x + c) = d$	Nej

Tabell 4: Ekvationer och om de kan lösas

4.2 Arbetsprocess

4.2.1 Python

Vi valde att skriva programmet i programmeringsspråket Python då det är allmänt enkelt att förstå och att arbeta med. Dessutom hade vi alla kunskap inom språket innan vi började med projektet.

4.2.2 Github

Programmet sparades på GitHub, som är en service för versionshanteringsprogrammet git. Git underlättar programmeringsprocessen för projekt som innefattar flera deltagare. GitHub har också inbyggda funktioner för t.ex. problemhantering, då s.k. issues kan öppnas för att flagga upp utvecklarna att man har funnit ett eller flera problem i koden.

5 Diskussion

5.1 Komplexitet

En kods komplexitet definieras som gränsvärdet av tiden det tar för programmet att utföra sin uppgift när storleken på indatan går mot oändligheten. Att komplexiteten är ett gränsvärde har fördelen att den endast blir beroende av det största värdet, dvs. indatan. Till exempel: om det tar linjär tid att ta indata och kvadratisk tid att behandla indatan blir komplexiteten kvadratisk tid då den linjära termen inte spelar någon roll när storleken på indatan blir hög/stor. Detta skrivs som $O(n^2)$. Med O menas juste gränsvärdet när n går mot oändligheten.

Även om projektet blev ett bibliotek som kan användas inuti andra program så fokuserade vi inte på att göra programmet så snabbt som möjligt. Det var viktigare för oss att få så många kärnfunktioner som möjligt att fungera bra så att biblioteket, m.h.a. ett program som är inkluderat i projektet, kan användas som en avancerad miniräknare. Eftersom sannolikheten att programmet kommer behöva hantera stora mängder indata är låg, så ligger vikten inte på att hålla komplexiteten låg.

5.2 Kodstruktur

Kodstruktur är essentiellt för större projekt för att underlätta vidareutveckling. Därför delade vi upp projektet exempelvis i klasser. Under arbetets gång hjälpte det oss att förstå varandras kod.

Strukturen var dock inte perfekt. Klasserna delades upp i varsin fil, och eftersom klasserna ofta behöver ha tillgång till andra klasser och dess metoder behöver dessa då importeras. Detta ledde till att om två klasser behövde tillgång till varandra (vilket hände i vårt fall) så skulle de importera varandra. Inom programmering kallas detta för circular import, och är allmänt dålig praxis, eftersom det ökar risken till att programmet helt enkelt inte fungerar, eller att det t.ex. blir svårare att reorganisera projektet om det skulle behövas.

5.3 Möjliga förbättringar

Just nu kan biblioteket endast lösa ekvationer som är skrivna på polynomform, och endast upp till den andra graden. Dessvärre hanteras division på ett sätt som gör att vissa ekvationer (framförallt de som inkluderar division med en okänd variabel) ej går att lösas. På grund av detta kan programmet ej heller utföra polynomdivision.

Även om fokuset på projektet ej låg på hastigheten (se 5.1) betyder det inte att det inte finns fördelar med att göra projektet snabbare, mer effektivt. Användningsområdet av biblioteket skulle utökas drastiskt, då det skulle kunna användas i applikationer som kräver förenkling av t.ex. polynom av någon hundratals grad.

Biblioteket har också problem med att utföra konjugatregeln. För det mesta gör biblioteket rätt men det har problem med att förenkla $ab+(-1)\cdot ab$.

5.4 Lärdomar

Vi har lärt oss om av syntaxträd och hur man kan strukturera sådana, inklusive hur man kan specialisera olika noder så att de har olika funktioner (Se 3.2). Vi lärde oss också hur man med hjälp av programmeringsspråkets implementation av klasser och arv kan implementera det i ett bibliotek.

Algoritmen för potensnoden krävde att vi skulle känna till multinomialsatsen; något som ingen av oss kände till innan projektet.

6 Källkritik

Det ligger i naturen hos mycket av den informationen vi använt oss av att den är väldigt lätt att verifiera. Antingen fungerar en algoritm eller inte, det går inte att vinkla en algoritm.

6.0.1 KTH

Examensarbetet är skrivet år 2016 av Dimitrij Lioubartsev, student vid KTH. Examensarbetet kontrollerades av handledare och därför kan vi vara säkra på att källan är pålitlig. Dessutom är KTH ett väldigt välrespekterat universitet.

6.0.2 Chalmers

Informationen hämtades från anteckningar av en föreläsning på Chalmers, 2011 för information om abstrakta syntaxträd. Föreläsningen hölls av professorn Aarne Ranta, som också är avdelningschef för funktionell programmering på Chalmers.

6.0.3 Encyclopædia Britannica

Encyclopædia Britannica är ett av om inte det mest respekterade uppslagsverken som finns. Informationen är skriven av experter och verifierad, det gör det till en bra källa.

6.0.4 Proofwiki

Proofwiki är en wiki för matematiska bevis. För att skriva och ändra krävs godkännande, medlemskap söks inte skapas. Den är fortfarande overifierad med eftersom informationen överensstämmer med britannica.

6.0.5 Python.org

Python.org är skrivit av Pythons utvecklare, och det gör informationen extremt trovärdig. Dessutom är även den här typen av information antingen fungerande eller inte.

Referenser

[1] http://www.diva-portal.se/smash/get/diva2:945222/FULLTEXT01.pdf Dmitrij Lioubartsev, *Constructing a Computer Algebra System Capable of Generating Pedagogical Step-by-Step Solutions*, KTH, 2016, [2017-10-21]

[2] https://www.britannica.com/science/multinomial-theorem William L. Hosch, *Multinomial theorem*, Britannica, [2018-02-18]

[3] https://proofwiki.org/wiki/Multinomial_Theorem *Multinomial Theorem*, ProofWiki, [2018-03-11]

[4] http://www.graphviz.org/pdf/dotguide.pdf Emden R. Gansner, Eleftherios Koutsofios och Stephen North *Graphviz dot dokumentation*, [2015-01-05]

[5] http://www.cse.chalmers.se/edu/year/2011/course/TIN321/lectures/proglang-02.html Aarne Ranta, Chalmers, 2011 [2018-05-06]

[6] https://docs.python.org/3/ *Python dokumentation*, Python Software Foundation , [2017-2018]

Bilagor

A

Begrepp

Uttryck	Förklaring	Se
Träd	En riktad graf utan cykler där noder har endast en förälder	2.1
Nod	En del av trädet	2.1
Subnod	En nod under en annan nod	2.1
Rotnod	Noden högs upp i trädet	2.1
Homogena noder	Operatörsnoder där ordningen inte spelar någon roll(+,·)	3.4.1
Inhomogena noder	Operatörsnoder där ordningen spelar roll(-,/)	3.4.1
Lexikalisk analys	Separerar en text rad tex 'a+b' blir a,+,b	3.3.1
Token	sv tecken, i detta fallet delar av ett utryck tex ett tal	3.3.1
Parser	Gör om en lista av tokens till ett träd med avsende till prioritet	3.3.1
Object	Har metoder och egenskaper	2.2.1
Klass	Ritning för ett objekt	2.2.1
Dotsyntax	Programmeringsspråk som används av biblioteket graphviz för att generera bilder på träd	3.6

B

Hur man använder biblioteket/programmet

B.1

Installationskrav

- GNU+Linux / BSD / macOS / termux(Android) / cygwin(Windows)
- Python 3
- Purplex
- PIL

B.2 Valfria krav

- Graphviz
- Feh

B.3 Installationsinstruktioner

1. Ladda ner GAMLPs källkod från github

   ```
   $ git clone https://github.com/ALI-Team/GAMLP.git
   ```

2. Kör installationsskriptet setup.py (administratörsrättigheter kan behövas)

   ```
   # python(3) setup.py install
   ```

B.4 Anxvänding av programmet

Biblioteket kommer med ett program för att enkelt kunna testa/använda det. Kör "gamlp" i terminalen för att starta det.
Utryck kan sedan direkt matas in i programmet som kommer bearbeta den inmatade datan utifrån olika regler, som i GAMLP kallas för *flaggor*. Vilka flaggor som är tillgängliga fås av kommandot "flags" (exempel på flaggor kan vara t.ex. generera en bild på trädet.) För att ändra på en flagga används kommandot "set [namn] [on/off]".