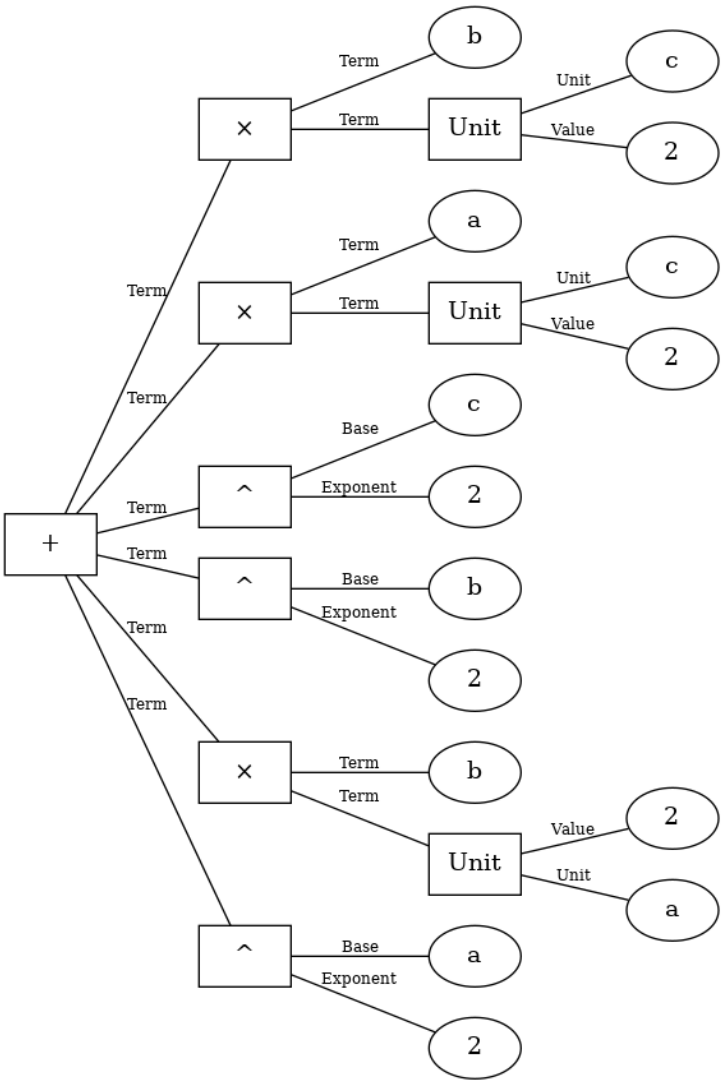
GAMLP: En symbolhanterare

Matstoms, Axel Janković, Luka Matstoms, Ivar

15 mars 2018



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 because of how quickly code can be written in Python and due to the fact everybody in the group knew Python 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 our goals set in the beginning, including addition and multiplication of polynomials correctly, although not always fully simplified. The application is also able to solve polynomials up to 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.ga

Keywords: Abstract syntax tree; Python; Algebraic Expressions

Innehåll

1 Inledning **4**

2 Teori **4**
2.1 Abstrakt syntaxträd 4
2.1.1 Allmänt om träd 4
2.1.2 Implementation av syntaxträd 5
2.1.3 Noder 5
2.2 Allmänt om förenkling 6
2.3 Homogena noder 6
2.3.1 Allmän förenkling av homogena noder 6
2.3.2 Additionsnod 7
2.3.3 Multiplikationsnod 7
2.4 Inhomogena noder 9
2.4.1 Potensnod 9
2.4.2 Subtraktionsnod 10
2.4.3 Divisionsnod 10
2.4.4 Ekvationsnod 10
2.5 Ekvationslösning 11
2.5.1 Ekvationsmetoder 11

3 Metod **12**
3.1 Klasser och Objekt 12
3.1.1 Objektorienterad programmering 12
3.1.2 Subklasser och arv 12
3.1.3 Noder 13
3.2 Tolkning av inmatade ekvationer 14
3.2.1 Lexikalisk analys 14

1 Inledning

Eftersom vi alla i gruppen delar en stor passion för programmering ville vi göra något som kombinerar programmering med ett naturvetenskapligt ämne. Vi övervägde flera olika projekt, bland annat en realtids fysikmotor, implementationer av matematiska funktioner (såsom sinus, exponenter) på hårdvarunära nivå samt idén som valdes: en symbolhanterare. En symbolhanterare, eller CAS, Computer Algebra System, är ett datorprogram som kan mata in, tolka, och lösa ekvationer, exempelvis $5x + 10 = 20$. En annan funktion som en symbolhanterare antar 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 och kan samt kan hantera variabler, inte bara värden.

2 Teori

2.1 Abstrakt syntaxträd

Det är besvärligt för ett program att tolka ekvationer som är skrivna på vanlig form (e.x. $5x + 10 = 15$). Därför är det viktigt att ekvationen först byggs upp på ett sätt som är mer hanterbart för ett datorprogram. Så kallade abstrakta syntaxträd används oftast inom datavetenskap för att strukturera programmeringskod så att 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.

2.1.1 Allmänt om träd

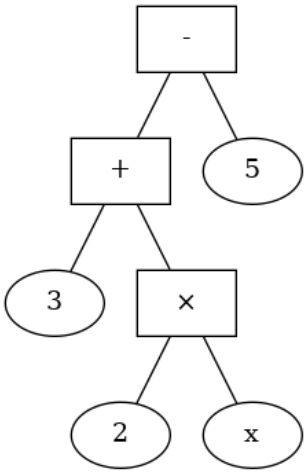
Inom datavetenskap beskrivs träd som en form av datastruktur för att representera hierarkisk data. Ett träd inom matematiken består av noder som är sammansatta på så vis att varje nod kan innefatta ett eller flera barn, som i sig också är noder. I det här dokumentet kallas barnen för barnnoder eller subnoder. Noder behöver dock ej innefatta några barn, och då kallas dessa noder för löv. Dock behöver varje nod anta ett värde. Detta kan vara t.ex. ett numeriskt värde (e.x. 5 eller 10), eller en operation (e.x. addition, multiplikation). Om en nod ej är ett barn till en annan nod betyder det att denna nod är högst upp i trädet. Denna typ av nod kallas för rotnod eller bara rot.

2.1.2 Implementation av syntaxträd

Trädet i sig utgår från en rotnod (eng. rootnode). I vårt fall är denna nod alltid ett likhetstecken (=), eftersom det är där ekvationen utgår ifrån. Precis som t.ex. en operatörsnod kan rotnoden anta två subnoder (I detta fall måste två noder antas, en för VL och en för HL). Därefter kan leden byggas upp av olika sorters noder.

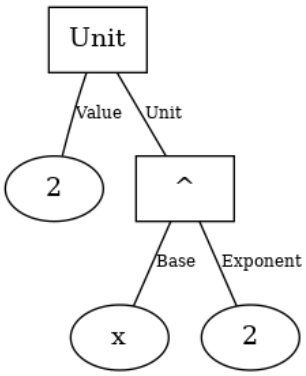
2.1.3 Noder

En nod måste anta ett värde. Detta kan uppnås genom att ge noden ett konkret värde (e.x. 5 eller 10), en konstant (e.x. π, e), funktion (e.x. log, sin) eller operator (e.x. $+$, $-$, $\times$, $\div$, $\hat{}$). Om noden är sådan att den kräver argument, dvs. kräver ett eller flera värden för att själv få ett värde ($+$ har ej ett värde utan två siffror på vardera sida av sig.) så kan noden själv anta noder (s.k. barnnoder, eng. childnodes) för att uppfylla sitt värde. Se Figur 1. Eftersom programmet ska kunna hantera ekvationslösning, måste



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

den okända variabeln kunna representeras (e.x. $5x + 10 = 20$. Här är x den okända variabeln). För att representera variabeln används en speciell typ av nod; 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.

2.2 Allmänt om 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.

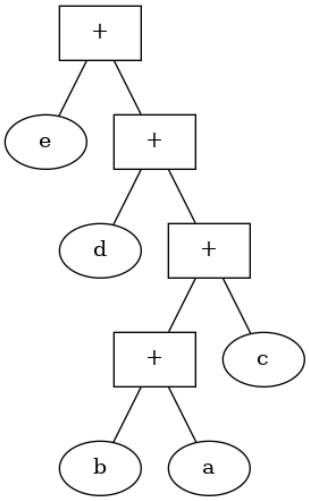
2.3 Homogena noder

Homogena noder är de noder där ordningen av subnoder ej spelar någon roll vilket gör att de kan lagras som en lista (array) och är inte limiterad i antal. Dock byggs homogena noder först upp av flera noder med bara två subnoder (Se 2.4 Inhomogena noder) eftersom det är så programmet tolkar inmatningen, se Figur 3.

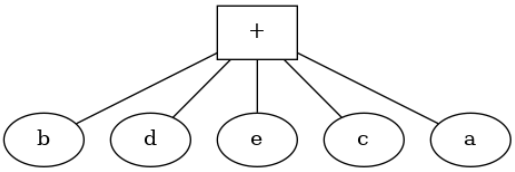
2.3.1 Allmän förenkling av homogena noder

Att förenkla noder som precis har tolkats av programmet (se Figur 3) är dock opraktiskt. Därför är det första steget i förenklingen att försumma staplade homogena noder av samma typ till en enda nod. Om noderna antar numeriska värden räknas de ihop till en subnod, medans variabler (x) helt enkelt lämnas som en subnod. Efter det läggs elementen in i noden en efter en och varje element kontrolleras om den kan slås ihop med en annan subnod. Detta gäller dock endast för noder med numeriska värden. Hur enhetsnoder (se 2.1.3 för förklaring av enhetsnod) hanteras är dock olika för varje typ av homogen nod. Därför förklaras detta i detalj i samband med varje typ av nod.

Två enhetsnoder med samma enhet kan slås ihop genom att addera värderna.



Figur 3: $(a + b + c + d + e)$ tolkat av programmet.



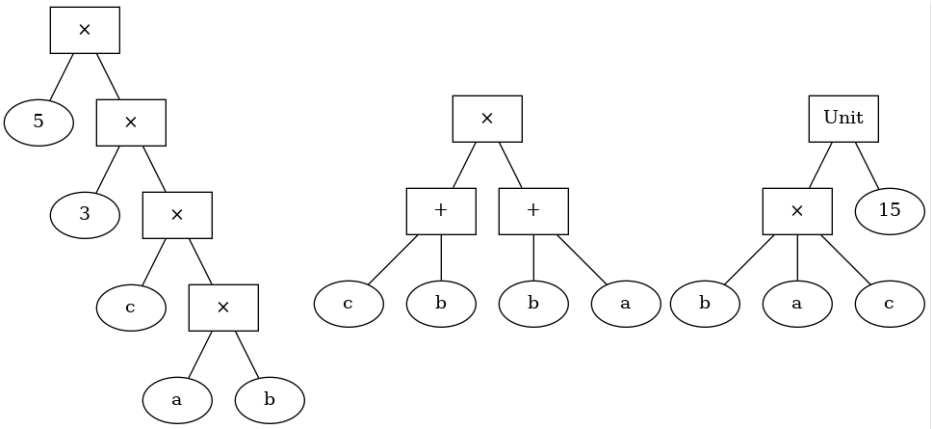
Figur 4: $(a + b + c + d + e)$ efter första steget av förenkling av homogen nod

2.3.2 Additionsnod

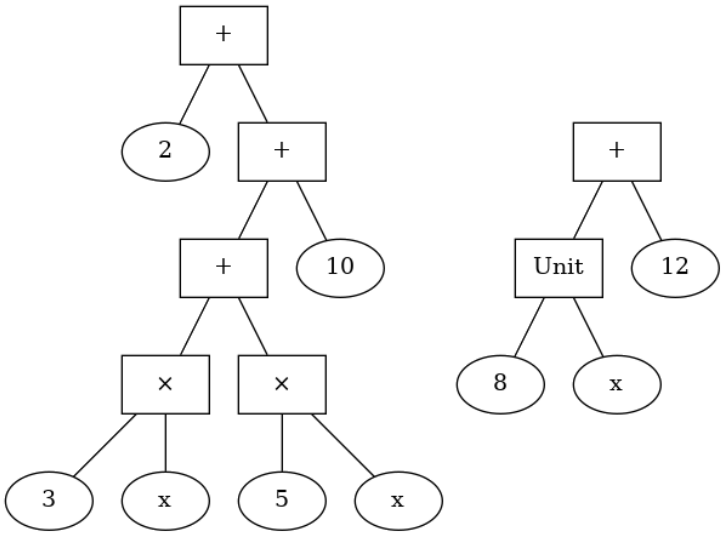
Förenklingen av en additionsnod funkar genom att fylla på en tom lista med värden från en additionsnods barnnoder och kollar om en nod i barnnodslistan kan läggas ihop med en nod i den nya listan. T.ex. om en additionsnod består av bara siffernoder så kommer dessa adderas. Om flera enhetsnoder med samma enhet förekommer så kommer deras koefficienter summeras (se Figur 6). En potensnod kan slås ihop med en annan nod om basen på potensnoden är samma som den andra noden, isåfall skapas en ny potensnoden med exponenten ökad med ett.

2.3.3 Multiplikationsnod

Efter de generella förenklingarna av homogena noder (Se 2.3.1 Allmän förenkling av homogena noder) separeras additionsnoder från övriga noder. Alla kombinationer av dessa separerade additionsnoders subnoder multipliceras med övriga noder och adderas ihop. Enhetsnoder kombineras genom att skapa en ny enhetsnod vars koefficient består av produkten av de ori-

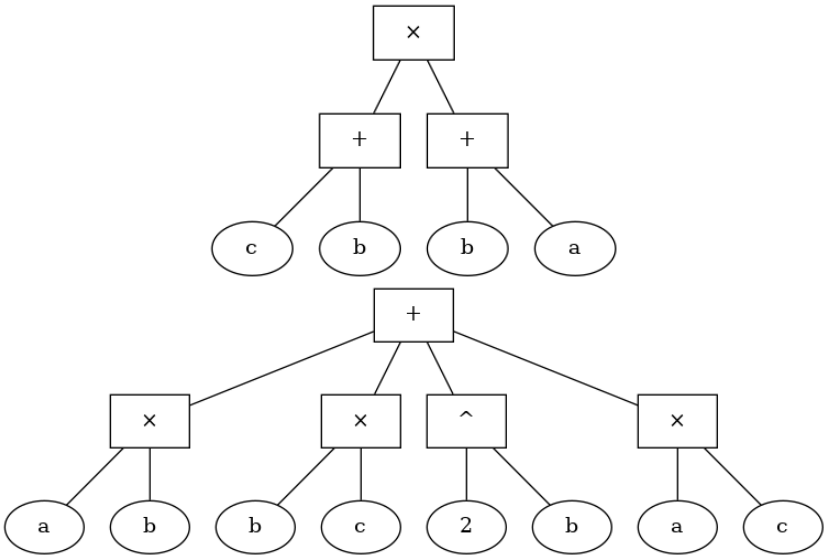


Figur 5: $(a \times b \times c \times 3 \times 5)$ förenklas i tre steg, a,b,c lyfts upp till samma nod och talen separeras till sin egen nod, tillslut räknas talen ihop.



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

ginala enhetsnodernas koeficienter och vilken grad den nya enhetsnoden representeras i bestäms av hur många enhetsnoder det från början var. (e.x. $5x \times 10x = 50x^2$). Enhetsnoder kan också kombineras med övriga noder genom att de övriga värdena multipliceras in i enhetsnodens värde (Se 2.1.3 Noder).



Figur 7: $(a + b) * (c + b)$ förenklas till $(a * b) + (b * c) + (b^2) + (a * c)$

2.4 Inhomogena noder

Inhomogena noder genomför inhomogena operationer; dvs. operationer som där ordningen av subnoderna spelar roll. (e.x. $a - b \neq b - a$). Därmed kan inhomogena noder ej kombineras rekursivt på samma sätt som homogena noder kan. (Se 2.3.1 Allmän förenkling av homogena noder) Detta medger också att det inte finns ett allmänt sätt att förenkla inhomogena noder.

2.4.1 Potensnod

Potensnoder innehåller två barnnoder, basen och exponenten. En potensnod med en additionsnod i basen, exempelvis $(a + b + c)^3$, expanderas med hjälp av 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

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

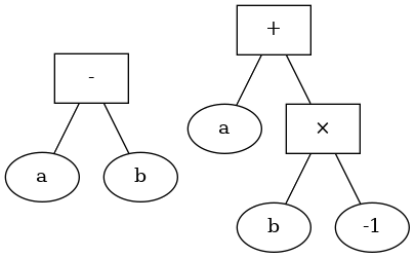
Den behöver dock lite förklaring. Med $\sum_{k_1+\cdots+k_m=n}$ menas alla möjliga kombinationer med återläggning av exponenter i $a_1^{k_1} \cdot a_2^{k_2} \cdot \ldots \cdot a_m^{k_m}$ där $\sum_{i=0}^{m} k_i = n$

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.

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

2.4.2 Subtraktionsnod

Subtraktionsnoden är ej en homogen nod då ordningen på ekvationen spelar roll $a-b \neq b-a$ och därmed måste subtraktionsnoden anta endast två noder; en för vänster sida och en för höger sida. För att simplifiera subtraktions noden så implementeras inget nytt. Den använder sig istället av additions noden och multiplikations noden för att uppnå en minus operation. Istället för att direkt räkna ut $a-b$ så skrivs ekvationen om till $a+((-1)\cdot b)$. Därmed kan samma resultat uppnås med bara addition och multiplikation.



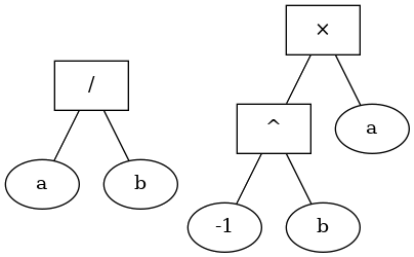
Figur 8: $a - b$ före och efter förenkling. Minus noden transformeras till en additionsnod med b ersatt med $b * -1$

2.4.3 Divisionsnod

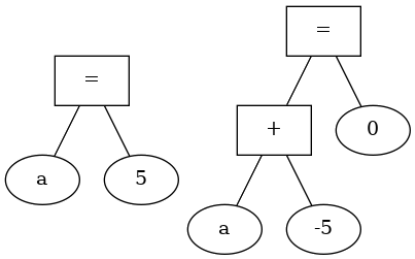
På samma sätt som subtraktionsnoden skiver om ekvationen $a - b \Rightarrow a + ((-1) * b)$ så skriver divisionsnoden om ekvationen så att förenklingen istället kan hanteras av multiplikationsnoden. $a/b \Rightarrow a * (b^{-1})$

2.4.4 Ekvationsnod

En ekvationsnod antar en nod för varje led (I vårt projekt kan en ekvationsnod endast anta två subnoder; en för VL och en för HL.) Målet med ekvationsnoden är att representera en ekvation som kan lösas ut och förenklas. Detta innebär att subtrahera ett led från båda leden, som på så sätt lämnar ett led tomt.



Figur 9: a/b förenklas till $a * (b^{-1})$



Figur 10: $(x + 4) \cdot (2 + x) = 10$ förenklas till $-2 - 10 = 0$

2.5 Ekvationslösning

Att lösa ekvationer med en okänd (dvs. lösa ut x) innebär att först förenkla ekvationen så att en sida antar värdet 0. Därefter kommer olika regler appliceras på ekvationen beroende på vilken grad den är i. Vilken grad den är i bestäms av den högsta potensen.

När en ekvation löses så förenklas den först så högerled är lika med 0. Sen går den igenom alla direkta subnoder nod för nod och kolla efter okända och potens. GAMLP kan bara lösa ekvationer med en okänd. Den högsta exponenten på den okända är graden på ekvationen. Programmet har en samling av olika metoder för att lösa ekvationer och hittar den lämpligaste metoden och utför den. Vilka metoder som används beskrivs i 2.5.1.

2.5.1 Ekvationsmetoder

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 Metod

3.1 Klasser och Objekt

En viktig del av struktureringen av noder (se 2.1.3, Noder) handlar om att en nod kan vara en utökad typ av en annan nod. Det vill säga, istället för att på nytt skapa varenda nod kan man istället skapa en generell nod för t.ex. operatörer, och sedan utöka den för varje sorts operation. Detta kallas för Objektorienterad Programmering (Object Oriented Programming, OOP).

3.1.1 Objektorienterad programmering

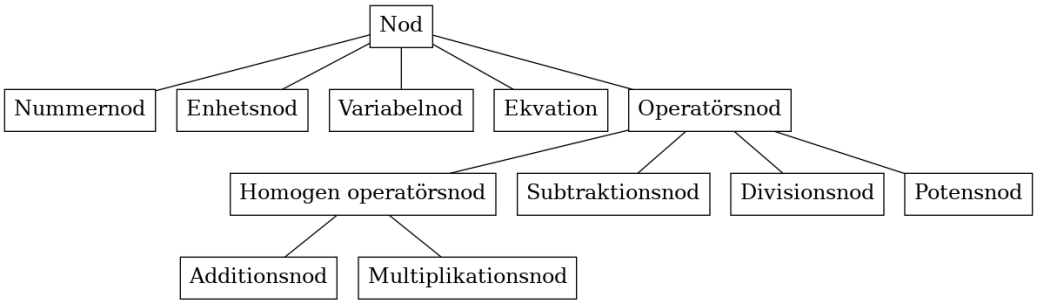
I princip funkar Objektorienterad programmering genom att man för varje föremål, struktur, etc. man vill beskriva i ett program har en s.k. klass. En klass är en abstrakt modellering av ett objekt; dvs. den innehåller definitioner för attributer och handlingar. (Kan liknas t.ex. en ritning; klassen innehåller information om strukturen, men är i sig inte en färdig produkt.). Sedan används klassen för att skapa objekt som kan fylla på med värden på attributerna och faktiskt utföra de beskrivna handlingarna. Ett exempel som är enkelt att förstå 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. 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.

3.1.2 Subklasser och arv

Ett användbart redskap inom objektorienterad programmering är arv. För att fortsätta med bil-exemplet (se 3.1 Klasser och Objekt) 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.

3.1.3 Noder

Alla noder är subklasser av klassen Nod. Nodklassen innehåller definitioner för metoder som används av och är gemensamma för alla subklasser. T.ex. så hanteras operationer av noder (e.x. addition, subtraktion) av denna klass (se tabell 1). 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 (se 2.3 Homogena noder är de noder där ordningen av subnoder ej spelar någon roll vilket gör att de kan lagras som en lista (array) och är inte limiterad i antal. Dock byggs homogena noder först upp av flera noder med bara två subnoder (Se 2.4 Inhomogena noder) eftersom det är så programmet tolkar inmatningen, se Figur 3.



Figur 11: Diagram av nodklassen samt dess alla subklasser.

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.)
`formatted`	Returnerar en text representation av noden.

Tabell 1: Funktioner och dess användning i basklassen nod.

3.2 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.

3.2.1 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
$\cdot$	Multiplikationsnod
$\div$	Divisionsnod
$\hat{}$	Potensnod
$=$	Ekvation

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