Universiteit Utrecht

Programmeren in de Wiskunde

# Verslag Eindopdracht

*Begeleider:*
Emiel Broeders

*Auteurs:*

| | |
|---|---|
| Wessel van Eeghen | #4007557 |
| Mathijs de Lepper | #3987949 |
| Jurriaan Parie | #3938549 |


## Expressieboom

https://github.com/mdelepper/Eindopdracht

3 juli 2015

# 1 Inleiding

In dit project gaan we aan de slag met *Expressiebomen*. We zetten expressies, aangeleverd als strings met behulp van de Reverse Polish Notation (RPN) om in een heuse expressieboom. Vervolgens proberen we in deze expressiebomen zoveel mogelijk functies toe te passen, zodat we allerlei operaties en functionaliteiten makkelijk kunnen uitvoeren. Hierbij moet rekening worden gehouden met de verschillende manieren waarop je je informatie aangeleverd kan krijgen, de manier waarop een expressie uiteindelijk in een expressieboom wordt geplaatst en natuurlijk de structuur van de bomen zelf.

We hebben een basisbestand gekregen van waaruit we zijn begonnen met het uitbreiden van onze code. Bij het ontwikkelen van het de code hebben we gelet op functionaliteiten van de bomen. ¨Welke aanpassingen moet je kunnen maken om een gewenste functie overzichtelijk te kunnen implementeren?¨ Ook wilden we het programma zo gebruiksvriendelijk mogelijk maken. Verschillende invoermethoden moeten geaccepteerd worden en op de bedoelde wijze in de boom terecht komen. Ook moet je de uitdrukkingen kunnen versimpelen en kunnen berekenen.

De code is op een structurele manier opgebouwd. De klasse `Expression` is de meest fundamentele klasse (de superklasse). Deze bevat niet veel methoden, maar basisfuncties als `__str__` moeten hier wel in staan. Verder staan andere functies wel al gedefinieerd, maar die worden later in het bestand overschreven voor het gewenste resultaat. De reden om bepaalde definities al in `Expression` te definiëren, is dat alle subklassen en kind-objecten automatisch ook met de definities bekend zijn.
De klasse `BinaryNode` is een uitbreiding op (ofwel een subklasse van) de klasse `Expression`. Deze geeft een zinnigere implementatie van basisdefinities en geeft ook ruimte voor het invoeren van nieuwe functionaliteiten. Vervolgens bestaan weer uitbreidingen van `BinaryNode` (`AddNode`, `PowerNode`, ...). Deze objecten hebben standaard niks bijzonders, maar werken de functionaliteiten tot op het meest precieze niveau uit.

Verder bespreken we in dit verslag de theoretische achtergrond, functionaliteit en complexiteit van onze methoden.

# 2 De Opdracht

We hebben de aangeleverde code aanzienlijk uitgebreid. Naast de lijst met standaard functionaliteiten hebben we ook elementen toegevoegd uit de lijst met Extra mogelijkheden. Daarnaast hebben we nog een aantal (eigen) functies buiten de lijst om toegevoegd. Hieronder worden alle aanpassingen besproken en worden alle aanpassingen per stuk toegelicht.

## Standaard Aanpassingen

- **Aritmetische Operatoren**
  We hebben de aangeleverde code waarin het mogelijk is om de operator `+` uit te voeren dusdanig aangepast is zodat ook `-`, `*`, `/` en `**` uitgevoerd kunnen worden. Hiertoe hebben we in de klasse `Expression` voor elke operator een nieuwe functie gedefinieerd: `__sub__`, `__mul__`, `__truediv__`, en `__pow__`. Vervolgens worden deze functies doorverwezen naar de subklassen, respectievelijk: `SubstractNode`, `MulitplyNode`, `DivisionNode` en `PowerNode`. Hier wordt op de meest basale manier de actie bij het tegenkomen van een operator overschreven.

- **Het vertalen van een Expressie naar een string**
  Nu willen we ervoor zorgen dat overbodige haakje weggelaten worden. Hiervoor is het van belang dat we eerst de precedentie en associativiteit van operatoren invoeren. Deze functies definiëren we buiten alle klassen om, zodat ze altijd en overal gelden. Optellen en aftrekken krijgen precedentie één, vermenigvuldiging en deling precedentie twee, en machtsverheffing precedentie drie. Optelling en vermenigvuldiging krijgen associativiteit nul waar aftrekken, delen en machtsverheffen associativiteit één krijgen. Dit verschil wordt gemaakt omdat bijvoorbeeld wel geldt: `3+5 = 5+3`, maar niet `2/4 = 4/2`. Later in het project, wanneer we functies toe gaan voegen, voeren we de precedentie van vier in.

  Nu kunnen we haakjes netjes weg gaan werken. Dit gebeurt in de definitie `__str__` binnen de klasse `BinaryNode`. We hebben dan dus al te maken met expressiebomen. In de boom bepalen we in welke gevallen we wel en in welke gevallen we geen haakjes nodig hebben, afhankelijk van de precedentie en associativiteit van de operatoren. Dit doen we eerst voor de linkertak en dan voor de rechtertak (we scheiden

links en rechts omdat er verschillen zijn tussen beide in het geval dat de associativiteit van de operatoren verschilt). We maken de methode af door links en rechts te combineren en recursief toe te passen.

Met deze uitbreiding van de code kunnen we bijvoorbeeld de volgende berekening verwerken.

```
>>> a = Expression.fromString('(5 ** 3) + 33 / (11 + 4)')
>>> print(a)
5 ** 3 + 33 / (11 + 4)
```

- **Gelijke bomen**
  We willen twee bomen kunnen vergelijken. Dit doen we door in de verschillende klassen een overload van `def__eq__` te gebruiken. Op deze manier worden de twee expressiebomen vanaf een *top-down approach* bekeken en vergeleken.

```
a = Expression.fromString('3sin(x)'))
>>> print(a)
3 * sin(x)
b = Expression.fromString('sin(x)3')
>>> print(b)
sin(x) * 3
>>> a == b
False
```

- **De klasse Variable**
  Om ook variabelen te ondersteunen hebben we een klasse `Variable` geschreven, als subklasse van de`Constant` klasse. Hierin staat een constructor en een overload van `def __str__`, die zichzelf als waarde teruggeeft en een overload van de definitie `evaluate`. Als er een waarde is meegegeven voor de variabele, dan willen we deze waarde gebruiken in de berekening. Als dit niet het geval is, dan moet de variabele als

variabele worden teruggegeven.

```
a = Expression.fromString('3sin(x)'))
>>> a.evaluate()
'3.0 * sin(x)'
>>> a.evaluate('x':math.pi/2)
3.0
```

- **Evaluate**
  In de bovenstaande code is te zien hoe we de numerieke waarde van een expressie kunnen berekenen. Dit doen we door middel van de functie `evaluate`. In `Expressie`, de hoofd-/superklasse van de code, hoeft er nog niks te gebeuren. We gebruiken daarom een `pass` en overloaden de methode in de subklassen van `Expressie`. Zo willen we bijvoorbeeld bij de klasse `BinaryNode` dat (recursief) de linker- en rechterkant worden geëvalueerd. Afhankelijk van wat de zogeheten *child* teruggeeft, willen we een uiteindelijke waarde of expressie teruggeven aan de oproep. Dit houdt in dat bijvoorbeeld een constante zijn eigen waarde teruggeeft en een variabele `x` geeft óf `x` terug óf de waarde die voor `x` is meegegeven in de vorm van een `dictionary`.

## Extra Aanpassingen

- **Expression from String**
  We hebben een functie `post_tokenize` toegevoegd. Deze maakt het mogelijk om bij het gebruik van `Expression.fromString` een functie in te voeren zonder *-tekens te gebruiken tussen getallen en variabelen/functies. Om te voorkomen dat bijvoorbeeld `sin` wordt opgebroken in `s*i*n`, hebben we een methode `infunclist` gedefinïeerd. Hierdoor kunnen we met een `bool` checken of een deel van de string een bekende functie is of niet. Uiteindelijk kunnen we onder andere het volgende

doen:

```
>>> b = Expression.fromString('3xylog(ab) +5'))
>>> print(b)
3 * x * y * log(a * b) + 5
```

- **Gedeeltelijke evaluatie**
  Aangezien sommige functies meerdere variabelen gebruiken, willen we het mogelijk maken om een expressie gedeeltelijk te kunnen evalueren. Als we bijvoorbeeld de functie die hierboven staat gedeeltelijk willen evalueren, krijgen we bijv:

  ```
  >>> b.evaluate({'a':3, 'b':2})
  '3.0 * x * y * 1.791759469228055 + 5.0'
  ```

- **Nulpunten vinden**
  Het leek ons ook wel mooi om nulpunten te kunnen vinden van (simpele) expressies/functies. Dit hebben we gedaan door middel van het importeren van het programma `bisection`. Dit programma was een inleveropdracht en het leek ons wel mooi om hier iets mee te doen. Hier volgt een klein voorbeeldje.

  ```
  b = Expression.fromString('x*x - 4')
  >>> b.findRoots('x', -3, 4, 0.001)
  [-1.99981689453125, 2.00006103515625]
  ```

- **Versimpeling**
  Onder het kopje Versimpeling hebben wij twee functies toegevoegd: `abridge`, voor het versimpelen van een uitdrukking zelf en `simplify` voor het herschrijven van de structuur van een boom.

  - `abridge`
    We wilden expressies korter kunnen opschrijven zodat overbodige operaties worden weggelaten. Hiervoor hebben we een aparte functie gedefinieerd. Deze hebben we gedefinieerd in de klasse `Expression`, waarna we hem volledig overschrijven in de subklassen `AddNode`, `SubNode`, `MultiplyNode`, `DivisionNode` en `PowerNode`.

Het gaat erom dat we bijvoorbeeld een expressie als `5*3+0` gewoon terugkrijgen als `5*3`. Hetzelfde geldt voor bijvoorbeeld `x/1`, dat wordt `x`. We plaatsten de definitie van abridge in de klassen `Expression`, zodat alle kind-objecten de functie automatisch ook kennen. Omdat de regels vervolgens per operator verschillend zijn, overschrijven we de de functie in deze `Node`-klassen. De versimpelingen die we hebben ingevoerd zijn: `+0` en `-0` worden weggelaten, `*1` en `*0` worden versimpeld, `0/`, en `/1` worden versimpeld en `/0` geeft voortaan een foutmelding. Verder wordt `**0` en `**1` ook versimpeld opgeschreven.

– `simplify`
Met de `simplify`-functie veranderen we de structuur van een boom. De bedoeling is om iedere expressie in een boom te kunnen krijgen met een vergelijkbare structuur. Zo kun je makkelijker vergelijking tussen expressies maken en kun je snel en overzichtelijk zien wat het verschil is tussen twee uitdrukkingen.

Ook hier betrekken we de precedentie van operatoren in ons systeem. Omdat we te maken hebben met bomen definiëren we de `simplify`-functie in klasse `BinaryNode`. Het idee is om operatoren met lage precedentie zo hoog mogelijk in de boom te zetten en om de operatoren met hogere precedentie naar beneden te duwen. Zo wordt de uitdrukking `(2+3)*4` omgeschreven naar `(2*4 + 3*4)`.

We gaan te werk door eerst de linkerkant van een `Node` te versimpelen en vervolgens de rechterkant. Zo hoef je niet alle combinaties van operatoren boven, links en rechts apart te behandelen. We stellen functies `simplify_left` en `simplify_right` op. Vervolgens maken we gebruik van recursie en combineren we de beide functies tot een eindresultaat.

- **Bekende en Onbekende Functies - de Unaire Boom**
  Om niet alle elementen uit een berekening in een binaire boom te hoeven plaatsen, hebben we de unaire boom geintroduceerd. Dit biedt de mogelijkheid om ook functies in een berekening op te nemen. Deze unaire boom heeft slecht één kind, hetgeen de functie evalueert. Dit kind kan zowel een binaire- danwel een unaire boom zijn. Zo is het mogelijk dat berekeningen met functies in functies geëvalueerd kunnen worden. Functies hebben allen hun eigen klasse gekregen, als subklasse van de `UnaryNode` klasse.

  ```
  >>> Expression.fromString('sin x ** sinh log x + 3')
  sin(x) ** sinh(log(x)) + 3
  ```

- **Negatie**
  Het − teken kent een dubbele betekenis. In het geval van *negatie* vervangen we het − teken in onze berekening door een ∼ om het vervolgens als een functie te beschouwen.

  ```
  >>> expr = print(Expression.fromString('-x / y + -sin -x'))
  -x / y + -sin(-x)
  ```

# 3 Discussie

Code-inhoudelijk zijn er een aantal opmerkingen en gebreken die boven water komen drijven. We bespreken ze hieronder stuksgewijs.

- **Het vertalen van Expressie naar String**
  Een opmerking die we over deze functie kunnen maken is dat we in de definitie van `__str__` weer twee nieuwe definities voor links en rechts hebben gebruikt. Iets wat niet preferabel is. We hebben bij het ordenen en mooier maken van de code geprobeerd dit uit elkaar te halen, zodat we geen definities in definities hebben, maar dit gaf ineens een onverwachte foutmelding, die we zo tegen het eind van het project niet snel genoeg meer konden verhelpen. We hebben de oude methode dus toch maar aangehouden met het oog op pragmatisme. Zo werkt de code in ieder geval. Voor de functie simplify geldt exact hetzelfde.

- **Versimpeling `abridge`**
  De reden dat ook de machten en de vermenigvuldiging en deling versimpeld worden, is niet alleen dat het mooier staat in een expressie, maar ook omdat je, als je een differentiatiefunctie wilt toepassen, het lelijk is als je een uitkomst met `x**0` of `x**1` krijgt. Op deze manier kun je makkelijker onderscheiden wanneer de afgeleide van een functie bijvoorbeeld een constante of zelfs nul geeft. Helaas zijn we niet meer toegekomen aan het definiëren van een differentiatiefunctie, maar alle benodigdheden liggen klaar om de uitbreiding te voltooien.

- **Versimpeling `simplify`**
  Helaas kunnen we de boom niet helemaal goed geordend krijgen, ingericht naar precedentie. `(2+3)**4` is immers niet gelijk aan `2**4 + 3**4`. Toch proberen we de bomen zover mogelijk naar één vaste structuur te brengen.

# 4 Reflectie

De samenwerking is ontzettend goed verlopen. *Github* was de enige die af en toe niet helemaal mee wilde werken, maar dat lag deels aan het feit dat we er nog niet zo heel lang mee bekend waren. Een grove verdeling van het werk is als volgt:

| Dag | Jurriaan | Mathijs | Wessel |
|---|---|---|---|
| 11 juni | Carrièremiddag | Carrièremiddag | Code begrijpen |
| 16 juni | Presentatie + Code begrijpen | Presentatie + EvaluateFunction voor alles behalve variabelen | Nieuwe operatoren ($-, *, /, **$ etc.) |
| 18 juni | Begin variabelen in de expressieboom | EvaluateFunction afmaken | Reduceren van haakjes |
| 23 juni | Standaard en onbepaalde functies | Versimpeling en GitHub | Extra omschrijving + GitHub |
| 25 juni | Functies afmaken, begin negatie | `4log(sin(3xy))` is nu `4*log(sin(3*x*y))` | Versimpeling van de boom en versimpeling van de uitdrukkingen |
| 30 juni | Negatie is af en functies zonder haakjes | Bisection geïmporteerd (werkt voor simpele functies) en PEP8 | Versimpelen is nu af |
| 3 juli | Verslag | Verslag | Verslag |

Het is echter niet zo dat iedereen alleen zijn eigen stukje heeft gezeten. We hebben regelmatig samen naar stukken code gekeken als we er niet uitkwamen en ook gewoon met elkaar overlegd tijdens het schrijven van de code. Ook hebben we overlegd over welke extra´s we wilden programmeren. Zo hebben we gekozen voor een aantal functionele extra's die op het lijstje stonden, zoals bijvoorbeeld *Versimpeling* en *Gedeeltelijke Evaluatie* en hebben we `Expression.fromString` aangepast en een functie voor nulpunten toegevoegd.

Gedurende de periode waarin we aan de opdracht hebben gezeten, kregen we steeds meer begrip voor hoe het programma in elkaar zat en in elkaar moest zitten. In eerste instantie moesten we uitvinden hoe *de template* in elkaar zat en waar we op voort moesten bouwen. Tijdens het eerste werkcollege dat we aan `Expressie` werkten, heeft Wessel bijvoorbeeld een speedcursus *Shunting-yardalgoritme* gegeven. Hierna was de code al een stuk duidelijker. Naarmate we de code meer en meer eigen hadden gemaakt, werden we ook steeds handiger in het debuggen van het programma. Af en toe kwamen we er niet helemaal uit en hebben we wat hulp gehad van de werkcollegebegeleiders, voornamelijk van degene die ons heeft begeleid: Emile.

Uiteindelijk hebben we naar ons idee een mooi resultaat geleverd en kunnen we terugkijken op een goede samenwerking. Het zou voor ons waarschijnlijk handig zijn als we volgende keer iets beter overweg zouden kunnen met *Github*. Het heeft ons af en toe wel wat gedoe opgeleverd als het *mergen* niet helemaal wilde lukken. Soms werkte de code dan ¨opeens¨ niet meer zo goed en moesten we op zoek gaan naar waar de fout zat. Uiteindelijk is het allemaal goed gekomen en zijn we blij met het huidige resultaat.

# Referenties

[1] https://nl.wikipedia.org/wiki/Shunting-yardalgoritme