- de **achtergrond van het probleem**, de vraagstelling en de wiskunde/algoritmiek die aan bod komt.

- het **ontwerp van je algoritme**, denk ook aan datastructuren en complexiteit

- de**implementatie**en welke keuzes je hebt gemaakt. Geef ook een **korte handleiding** over hoe de code gebruikt kan worden.

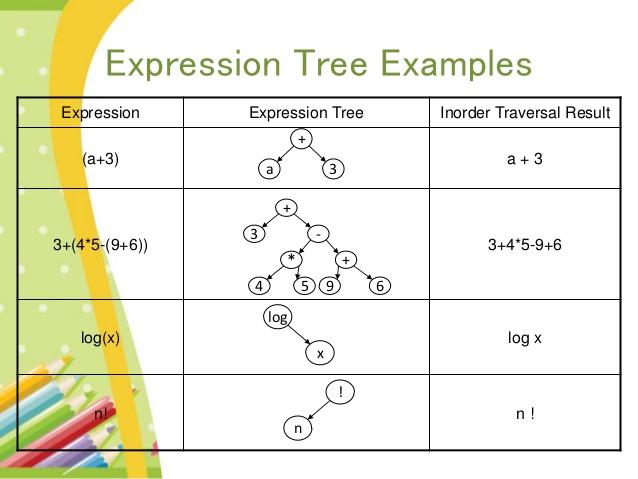
- de **resultaten**die je hebt verkregen met je code

- de **conclusies/discussie**. Bespreek bijvoorbeeld beperkingen van je code

- de **taakverdelingg**

# H1 Achtergrond van het probleem

## 1.1 Expressieboom

In deze opdracht zijn we aan de slag gegaan met een binaire expressieboom. Deze boom wordt gebruikt voor het representeren van wiskundige expressies. Dit is erg handig als je met wiskundige expressies wilt werken met python. Expressies die veel te zien zijn bij de binaire expressieboom zijn vaak van algebraïsche of booleaanse aard. Zo’n boom moet je visualiseren als een boom op de kop, dat wil zeggen de wortel is bovenaan en we gaan steeds levels omlaag en via binaire keuzes komen we steeds langs knopen waar een bepaalde wiskundige expressie in is opgeslagen. De wortel wordt ook wel de parent dit is de eerste node die we tegen komen in onze expressie en alles wat volgt via het doorlopen van de boom zijn children knopen van deze parent knoop.

## 1.2 Input van de boom

De input van de boom oftewel de expressies die gevormd gaan worden door de boom zijn expressie in postfix notatie. Met rekenkundige operaties zijn we gewend om te werken met infix notaties, bijvoorbeeld: a + b. Bij postfix notatie zouden a + b schrijven als a b +. Dit is handiger voor de computer voor het verwerken van de expressies in een boom. Zo zijn er geen haken meer nodig bij de postfix notatie en dat scheelt weer rekentijd voor de computer. De + operator zal de parent worden van deze drie knopige boom en de variabelen a en b zijn de children van de parent knoop +.

## 1.3 Shunting algortime

Zo’n postfix notatie krijg je niet zomaar, hiervoor heb je een algoritme nodig om dit te bewerkstelligen. Het algoritme wat we hier gebruikt hebben is het shunting algoritme. Dit algoritme is uitgevonden door een van de bekendste Nederlandstalige informatici/wiskundige Edsger Dijkstra. Het is een stacked-based algoritme, dit houdt in dat de keuze van datastructuur een stack is. Deze datastructuur is net iets anders, dan de standaard lijsten/arrays waar we mee werken in veel programmeertalen. In een stack wordt het lifo (last in first out) principe gehanteerd. Dat wil zeggen, het laatste element wat in de stack is toegevoegd gaat het als eerst uit. Stack wordt ook veel in recursie gebruikt. Bij recursieve problemen kun je vaak een bepaalde variabele niet meteen berekenen bij de eerste methode aanroep en moet je dieper gaan om uiteindelijk terug te komen bij de eerste aanroep. Heel leuk en aardig, maar nu weer terug naar het shunting algoritme. Bij dit algoritme worden elke stap tokens ingelezen en aan de hand daarvan stoppen we ze als het variabelen of constanten meteen op onze output array. De haakjes, rekenkundige operatoren en functies gaan op een stack aangezien deze op een bijzondere manier gesorteerd worden in postfix notatie, deze volgorde krijgen we het best voor elkaar om ze eerst in de stack te plaatsen. In het volgende hoofdstuk gaan we dieper op dit algritme en zullen de daadwerkelijke stappen van dit algoritme te zien zijn.

## 1.4 Boom doorlopen

Een algoritmische aanpak moet uitgevoerd worden om alle knopen te doorlopen van de boom. We willen graag de boom doorlopen, zodat we wiskundige operaties kunnen uitvoeren op de expressies, die we hebben opgeslagen in de boomstructuur. We kunnen bijvoorbeeld een polynoom als expressie in de boom opgeslagen en we willen deze expressie differentiëren, dit is heel goed mogelijk als we de boom gaan doorlopen ook wel tree traversal genoemd. We kunnen in principe twee bekende tree traversal algoritmes gebruiken, namelijk in-order traversal en post-order traversal. Die krijgen bij beide keuzes dezelfde expressie alleen de syntax van de notatie verschild wel. Zo krijg je een infix notatie waar we nomaliter meer rekenen bij in-order traversal en post-order traversal geeft ons de postfix notatie. Bij de postfix notatie staan de expressie net iets anders gesorteerd. Wij hebben zelf in-order gebruikt voor het doorlopen van de boom. In het volgende hoofdstuk is meer diepgang te vinden over dit algoritme.

# H2 Ontwerp algoritme en codering:

## 2.1 Algemene functies

Er zijn een aantal algemene functies in het programma. Ten eerste is er de functie ‘tokenize’. Deze functie heeft als input een normale expressie als string en geeft terug een lijst waarbij getallen en tokens gesplitst zijn. Ook houd het rekening met ‘\*\*’. Deze functie wordt gebruikt bij het Shunting-Yard algoritme.

Daarnaast zijn er nog vier kleine functies die informatie geven over een string. Ten eerste de functie ‘isexp’, die nagaat of een string een expressie is, ten opzichte van een los getal, niet-numerieke waarde of functie. De functie bekijkt allereerst of de string begint met ‘-‘. Als dit het het geval is, dan bekijkt het of de rest van de string een operatorsymbool ‘+’,’-‘,’\*’,’/’ bevat, en geeft aan de hand van dit True of False terug. Als de string niet begint met ‘-‘, dan doet de functie hetzelfde alleen dan voor de gehele string. Deze functie wordt vooral gebruikt voor de overload van ‘str’ in de BinaryNode subclass, om zo een versimpelde weergave te geven in een aantal gevallen.

Ten tweede is er de functie ‘isint’, die bekijkt of een object een integerwaarde representeert. De derde functie ‘isnumber’, die bekijkt of een string een numerieke waarde representeert. Beide functies worden gebruikt in het programma, om bijvoorbeeld te kijken of een constante een numerieke waarde representeert. Ten slotte is er de functie ‘ispos’. Deze bekijkt of een object “negatief” is, in de zin dat het object een negatieve numerieke waarde representeert, of een niet-numerieke waarde een negatie is. Deze functie is vooral bedoeld om minnetjes op te sporen, bij pogingen om expressies te versimpelen of te bepalen of haakjes geplaatst moeten worden.

## 2. 2 Expression klassse

Het centrale element uit de code is ‘Expression’subclass. Deze klasse, die de expressiebomen voorstellen (al bevat het ook de subclasses die slechts bladeren voorstellen), bevat de algemene overloads om aritmetica mee te bedrijven en het Shunting-Yard algoritme om een normale string expressie om te zetten naar een element van de Expression klasse. De overloads van optelling, subtractie, multiplicatie, deling, machtsverheffing en negatie van een expressie creëren allen een element van de subclass BinaryNode. Deze belangrijke subclass van Expression behandelen we hieronder. Naast subclasse BinaryNode, die de daadwerkelijke expressiebomen representeert, zijn er ook nog de subclasses: Constant, Variable en Function. We zullen al deze subclasses hieronder bespreken.

## 2.3 Shunting algoritme

Bij dit algoritme zijn we feitelijk steeds bewerkingen aan het doen binnen twee datastructuren. De eerste is een output array waar alle tokens geordend worden volgens het postfix principe en we hebben een stack waar o.a. rekenkundige operatoren in gaan, zodat we deze weer juist kunnen sorteren in de output array voor postfix notatie. Als eerst gaan we het stappenplan bekijken van het shunting algoritme. Er wordt een bepaalde string gelezen met expressie en elke losse expressie wordt gezien als een token.

Zolang nog niet alle tokens van de string gelezen zijn, doe het volgende:

* Token is een constante of een variabele, deze gaan meteen naar de output array
* Als het eerste niet het geval is dan is de token een operator (functies inbegrepen). De tokens die we herkennen zitten in een array en elke operator heeft ook een dictionary waarin staat hoeveel voorrang de operator heeft.  
  Als de stack leeg is doet de volgorde van de operatoren er niet toe en kunnen we meteen de eerste operator op de stack zetten.
* Wanneer de stack niet leeg is gaan we kijken of het laatste element van de stack een lagere voorrang heeft dan de nieuwe inkomende operator die gelezen wordt. Als dit het geval is, oftewel de nieuwe operator is groter of heeft een hogere voorrang dan de laatste operator op de stack, dan wordt de operator toegevoegd op de stack anders gaat de operator meteen door naar de output.
* Als de nieuwe gelezen token een ‘(’ is gaan we iets anders te werk. Deze zetten we meteen op de stack aangezien deze een scheidingsfunctie heeft. Er zal in de toekomst ook het volgende teken gelezen moeten worden “)”. Als we deze lezen en het laatste element van de stack is niet “(“dan wordt deze operator gepoppet en op de output array gezet. Als we “(” tegenkomen, dan poppen we deze van de stack, maar beide scheiding operatoren gaan natuurlijk nooit naar de outputarrray. Aangezien we geen gebruik maken van haakjes in postfix notatie.
* Als alle token gelezen zijn gaan we pas gebruik maken van de stack datastructuur. We gaan items poppen van de stack. Nu zien we de handigheid van de stack voor postfix notatie. Bij een pop operatie gaat het laatst toegevoegde element weg en deze gaan we toevoegen aan de output array. Door de booleaanse logica te gebruiken met de voorrang van operators zullen de operatoren met een hogere voorrang eerder gepopet worden en komen ze eerder in de output array tevoorschijn, dan de operatoren met een lagere voorrang dit is wat we graag willen aangezien we geen haakjes kunnen gebruiken in postfix notatie.

Nu is eigenlijk het echte shunting algrimte afgelopen, maar de methode waarin deze algoritme is verwerkt loopt nog een klein stukje code door. In de laatste stap wordt deze postfix verwerkt in een structuur die gebruikt wordt voor de expressieboom.

Als we alleen naar het shunting algoritme kijken in deze methode, dan kunnen we dus concluderen dat er per iteratie steeds, maar één teken wordt gelezen en allesbehalve constanten en variabelen wordt alles één keer op de stack gezet en vervolgens gepushet op de output array. Het aantal instructies per token is dus redelijk constant en de running time zal dus hoogstwaarschijnlijk lineair zijn aan de grootte van de input, oftewel de complexiteit in big oh notitie is O(n) voor het shunting algoritme.

## 2. 4 BinaryNode subclass

Deze subclass representeert de daadwerkelijk structuur van een expressieboom. Ieder element van deze klasse bestaat uit 3 elementen. Het eerste element is een operatorsymbool (een string) uit de lijst +,-,\*,/,\*\*. De andere twee elementen zijn een linker- en rechterkind, beiden een expressie op zichzelf (mogelijk dus nieuwe BinaryNode). Het operatorsymbool weergeeft de operator van een interne knoop in de gehele expressieboom, en de kinderen waarop deze operator werkt.

De aritmetica van deze klasse is afkomstig van de overkoepelende klasse Expression. De overloads voor optelling, subtractie, multiplicatie, deling en machtsverheffing (van de klasse Expression) creëren ieder namelijk een subclass van de BinaryNode-klasse. Bijvoorbeeld het optellen van twee expressies, creëert een subclass AddNode. Deze AddNode is niets meer dan de BinaryNode met als operatorsymbool ‘+’ en als kinderen de expressies. Later zullen we zien dat we handig gebruik kunnen maken van het feit dat verschillende operators verschillende subclasses zijn. De overload van negatie geeft een SubtractNode met als linkerkind de constante 0 en als rechterkind de BinaryNode. Een andere mogelijkheid had geweest een MultiplicationNode met als linkerkind -1 en rechterkind de BinaryNode. Beiden keuzes verschillen weinig, aangezien beiden situaties vereenvoudigd zouden worden bij het printen van een string.

De BinaryNode subclass bevat ook overloads van ‘str’, ‘==’ en de belangrijke functies om BinaryNodes te differentiëren en te evalueren in een punt. Al deze functies werken recursief. Omdat deze functies een hoofdbestanddeel zijn van gehele code, zullen deze ieder apart worden besproken.

## 2.5 Overload: string (str) voor BinaryNode

De overload van ‘str’ zorgt dat een expressieboom kan worden omgezet naar een expressie van “normale” vorm op een recursieve wijze. Over het algemeen geld dat de string van een BinaryNode er als volgt uitziet (waarbij LS de string van het linkerkind is, RS de string van het rechterkind is):

*LS operatorsymbool RS*,

met mogelijk haakjes rond LS of RS. In het bijzonder gaat deze overload er dus ook vanuit dat de string van de kinderen ook een leesbaar geheel is, ondanks dat een kind (Expression) in zijn algemeenheid geen overload van ‘str’ heeft. Echter, in deze code heeft iedere andere subclass van Expression ook een overload van ‘str’. Er zijn echter nog vele gevallen waarbij er haakjes om een string van een kind geplaatst worden. Ook kan in sommige gevallen de string worden vereenvoudigd. We zullen nu in volgorde het algoritme langsgaan.

Allereerst worden er vereenvoudigingen gemaakt in het geval dat LS en/of RS ‘0’ (of ‘0.0’) is. Als dit het geval is dan kunnen we in elk geval een versimpelde string terug te geven afhankelijk van de operator (bijvoorbeeld ‘0’ als de operator vermenigvuldiging is). Een bijzonder geval is als alleen LS=’0’ en het operatorsymbool ‘-‘ is. In dit geval bekijkt het algoritme, met de algemene functie ‘isexp’, RS een expressie is of slechts een enkele waarde. In het eerste geval wordt de string –(RS) teruggegeven. In het andere geval wordt -RS teruggegeven (waarbij meerdere minnetjes gecanceld worden). Als zowel LS als RS niet ‘0’ zijn, wordt er gekeken of een van beide ‘1’ is, om zo vereenvoudigingen toe te passen. Als dit niet het geval is, wordt er nog gekeken of RS of LS ‘-1’ is. Als dit het geval is, wordt simpelweg de string van de negatie van een van de kinderen teruggegeven, waarmee het algoritme (zie hierboven) mee om kan gaan. Overigens zorgen deze vereenvoudigen wel dat 0\*\*0=0 en 0/0=0, zonder een error te weergeven. We hebben helaas niet genoeg tijd gehad om deze uitzonderen te behandelen in het programma.

We zouden nog verder kunnen gaan in de vereenvoudiging, maar deze zijn niet toegepast. Daarmee kan gedacht worden aan de gevallen dat RS=LS en het operatorsymbool ‘-‘ of ‘/’ is. De samenvoeging LS en RS als deze beiden een numerieke waarde hebben is ook niet in ‘str’ toegevoegd, maar dit zal wel gebeuren als we de BinaryNode evalueren (zie hieronder). Het is ook de vraag in hoeverre we zouden willen versimpelen, omdat we ook de structuur van de expressiebomen willen laten zien. We hebben het daarom gelaten bij deze versimpelingen.

Als er geen vereenvoudigingen meer kunnen worden toegepast, bepaalt het algoritme of er haakjes rond LS of RS geprint worden. Dit wordt per kant apart bepaald. Voor LS is dit makkelijker dan voor RS. In het geval dat het linkerkind geen BinaryNode is of dat LS geen expressie is (wederom gebruik maken van de ‘isexp’ functie) worden er geen haakjes geplaatst. In de andere gevallen wordt er gekeken naar de precedentie van het operatorsymbool van het linkerkind (PL) ten opzicht van het operatorsymbool van de BinaryNode (PO) zelf. Als PL < PO, dan worden er haakjes geprint. Als PL > PO, dan worden er geen haakjes geprint. In het geval PL=PO worden er geen haakjes geprint, met uitzondering van het geval dat beide operatorsymbolen ‘\*\*’. Immers (a\*\*b)\*\*c=/=a\*\*b\*\*c=a\*\*(b\*\*c).

Bij RS is de situatie hetzelfde, met uitzondering van het geval dat het operatorsymbool van het rechterkind ‘-‘ of ‘/’, en PR=PO. Immers kunnen we wel haakjes weghalen bij (a+b)-c, maar niet bij a-(b+c). Deze situatie wordt apart behandeld. Waarbij haakjes niet worden weggehaald in de situatie a-(-b), maar uiteraard is dit geen probleem voor ‘/’. Uiteraard konden we ervoor kiezen om a-(-b) te printen als a+b.

## 2.6 Overload: equality (==) voor BinaryNode

De overload van ‘==’ zorgt ervoor dat twee BinaryNodes vergeleken kunnen worden. Ook hier is spraken van recursie op de kinderen, dus kinderen moeten ook vergelijkbaar zijn (ondanks dat de hoofdklasse Expression hiervoor geen overload heeft).

De meeste eenvoudige vergelijking zou de vergelijking die True doorgeeft als zowel de linker- als rechterkinderen gelijk zijn. Echter optelling en vermenigvuldiging hebben beide de commutatieve eigenschap (a+b=b+a) als de associatieve eigenschap (a+b)+c=a+(b+c). Het algoritme zal deze eigenschappen, en combinaties hiervan, testen als beiden operatorsymbolen gelijk zijn aan ‘+’ of ‘\*’. De manier waarop het algoritme dit doet is simpelweg alle gevallen langsgaan. Dit is voor operatiesymbolen ‘+’ en ‘\*’ gelijk, dus we beschrijven het geval ‘+’. Het geval waar beide kinderen van beide BinaryNodes AddNodes zijn. Dit geval kan worden gezien als (a+b)+(c+d)==(x+y)+(z+p). Er wordt dan direct de zes gevallen gecontroleerd, waarbij de volgorde binnen de haakjes niet uitmaakt, omdat commutativiteit recursief wordt gecontroleerd. Daarna worden de vier gevallen waar voor beide BinaryNode precies een kind een AddNode is behandeld. Tenslotte wordt “gewone” commutativiteit behandeld. Met andere woorden als de bovenstaande gevallen niet gelden (maar wel beiden operatorsymbolen gelijk zijn aan ‘+’) worden simpelweg de twee commutatieve gevallen vergeleken (links==links, recht==rechts en links==rechts, rechts==links). In alle andere gevallen geld een gelijkheid alleen als zowel de linker- als rechterkinderen gelijk zijn.

## 2.7 Differentiatiefunctie voor BinaryNode

Zoals de voorgaande twee overloads, is deze functie ook recursief op de kinderen van de BinaryNodes. De differentiatie kan dus alleen gebeuren als het ook de kinderen (bijvoorbeeld variabelen of functies) kan differentiëren. Hierbij worden de standaard differentiatieregels toegepast. Als we bijvoorbeeld een DivideNode differentiëren maken we gebruik van de quotientregel om een object terug te geven dat afhangt van de differentiatie van beide kinderen. Een belangrijke opmerking is dat de differentiatiefunctie niet alle expressies aankan die wel in deze code beschreven kunnen worden. Een PowerNode kan namelijk alleen gedifferentieerd worden, als de macht een constante is. Expressies als 2\*\*x of sin(x)\*\*cos(x), hoewel deze wel gemaakt kunnen worden in de code, kunnen helaas niet gedifferentieerd worden. Het programma geeft dit helaas niet aan.

## 2.8 Evaluatiefunctie voor BinaryNode

Ook deze functie werkt door middel van recursie op de kinderen. Ook kan het partiele evaluatie aan (waar sommige variabelen onbepaald blijven). Het algoritme onderscheid hiervoor twee gevallen. Het eerste geval is als de evaluatiefuncties op beide kinderen beiden een numerieke waarden teruggeven. In dit geval zal de evaluatiefunctie simpelweg een aritmetisch resultaat van beide uitkomsten teruggeven (in de vorm van de subclass Constant), afhankelijk van het operatiesymbool. Het kan echter ook voorkomen dat een of beide evaluatiefuncties van de kinderen een niet-numerieke waarde teruggeven. Dit kan bijvoorbeeld als er een niet-bepaalde variabele, of niet numerieke constante zich ergens bevinden in de expressiesubboom onder het kind. In dit geval zal het algoritme simpelweg een BinaryNode creëren van beide uitkomsten van de evaluatiefuncties op de kinderen.

## 2.9 Constant subclass

De Constant subclass representeert alle constante waarden. In de expressieboom neemt hij alleen plaats op de plek van een blad. Een constante heeft slechts een element, een waarde (‘self.value’). Dit is in veel gevallen een numerieke waarde (integer of float), maar kan ook een string zijn. In het laatste geval representeert de constante een onbepaalde constante. De Constant subclass heeft overloads voor ‘str’ en ‘==’, een aparte overload voor negatie en functies voor differentiatie en evaluatie. De string overload geeft simpelweg de string van zijn waarde terug. De overload voor gelijkheid geeft simpelweg de gelijkheid van de waarden. Differentiatie van constante geeft altijd de constante Constant(0). De evaluatiefunctie geeft altijd zichzelf terug, ongeacht de dictionary. Negatie van een constante werkt echte iets minder eenvoudig. Als de waarde een numerieke waarde is, geeft negatie gewoon de constante van de negatie van de waarde. Als de waarde een string is, is deze methode uiteraard onmogelijk. Daarom geeft het simpelweg de constante met waarde de oorspronkelijke string met een ‘-‘ teken ervoor terug. Om ervoor te zorgen dat de negatie van een negatie de oorspronkelijke waarde teruggeeft, hebben we ervoor gekozen om bij negatie het ‘-‘-teken juist weg te halen als dit het eerste symbool van de waardestring is. Problematisch zou uiteraard zijn als je een constante zou definiëren met waarde een string die begint met een of meerdere ‘-‘-tekens, terwijl je daar niet een negatie mee bedoelt. Echter, zouden wij niet weten waarom iemand dit zou doen, en dus verwachten wij dat het voor deze opdracht niet problematisch is.

## 2.10 Variable subclass

Naast de constante is er ook een subclass voor variabelen. De Variable subclass representeert de variabelen. Ook deze neemt in de expressieboom alleen plaats op bladeren. Een variabele heeft slechts een element, het karakterteken (‘self.char’). Dit is een string die de variabele representeert. De overloads van ‘str’, ‘==’ en negatie zijn identiek aan de overloads van de Constant subclass in het geval dat de waarde van een constante een string is (onbepaald is). In het bijzonder is negatie het toevoegen of weghalen van het ‘-‘ teken voor de string. De differentiatiefunctie geeft terug de constante 1 als het karakter in de bijgeleverde dictionary zit, -1 als de karakter van de negatie in de dictionary zit en constante 0 als beiden niet in de dictionary zitten. De evaluatiefunctie geeft terug de constante met de dictionarywaarde van het karakter als deze in de dictionary zit, de constante met negatieve dictionarywaarde van het karakater van de negatie als het karakter van de negatie in de dictionary zit, en constante 0 in de andere gevallen.

## 2.11 Function subclass en Basic subclass

We hebben ook een start gemaakt met een subclass voor functies. Deze subclass Function representeert functies met slechts een variabele en de functies kunnen alleen plaatsnemen op bladeren. Het programma heeft dus geen moglijkheden om compositie van functies te behandelen, behalve als de compositie expliciet gedefinieerd is. Dit heeft gevolgen voor de differentiatiefunctie die gedefinieerd was in de BinaryNode subclass (zie kopje *Differentiatiefunctie voor BinaryNode*). Deze subclasse is voor de rest ook weinig nuttig. De enige elementen van een functie zijn een karakter die de functie representeert (self.funchar) en een karakter van de variabele van de functie (self.varchar). Het enige wat op dit moment mogelijk is, is het printen van een algemene functie en het vergelijken van twee functies. Een gelijkheid geldt alleen als zowel de functiekarakters als variabelekarakters gelijk zijn.

En redelijk nuttige subclass van Function is de subclass Basic. Deze klasse representeert drie standaardfuncties (sin,cos,log) en de negaties van deze functies. Een overload van de negatie en differentiatie is hierdoor mogelijk. Ook is evaluatie mogelijk, waardoor differentiatie en evaluatie van een expressieboom met standaardfuncties in de bladeren mogelijk is.

**Ontbrekende elementen**

Er zijn ook een aantal zaken die nog ontbreken in de code. Sommige zijn al besproken (zoals functiecompositie, functies met meerdere variabelen of ). Intergratie moet je mogelijk bespreken hier. Ook hebben we geen tijd gehad om onbekende functies toe te voegen en de functie subclass echt goed uit te breiden.