

Facharbeit

Herstellen eines Funktionsplotters zum grafischen darstellen von Funktionen



Von Sebastian Doehring (Q1)  
Mathematik LK (Herr Röhr)

Inhaltsverzeichnis

[1. Einleitung - 2 -](#_Toc507626676)

[1.1 Begründung der Themenwahl - 2 -](#_Toc507626677)

[1.2 Ziele des Funktionsplotters - 3 -](#_Toc507626678)

[1.3 Grundlegende Problematik - 3 -](#_Toc507626679)

[2. Theoretische Grundlagen - 4 -](#_Toc507626680)

[2.1 Syntaxbaum - 5 -](#_Toc507626681)

[2.2 Parser - 6 -](#_Toc507626682)

[3. Praktische Umsetzung - 7 -](#_Toc507626683)

[3.1 Mathematischer Kontext (MathContext) - 7 -](#_Toc507626684)

[3.2 Implementation von Knotenpunkten (Node) - 8 -](#_Toc507626685)

[3.3 Implementation des Parsers - 8 -](#_Toc507626686)

[3.4 Implementation der Zeichnung des Graphen (GraphRenderer) - 11 -](#_Toc507626687)

[4. Quellenangaben - 13 -](#_Toc507626688)

# Einleitung

## Begründung der Themenwahl

Es ist ein schmaler grad der die Informatik von der Mathematik trennt. Schon vor langer Zeit habe ich angefangen mich intensiv mit der Informatik zu beschäftigen. Das Thema Compilerbau, welches dem des Funktionsplotters ähnelt, war da immer einer meiner Lieblingsthemen und eines mit dem ich mich auch am längsten Beschäftigt habe, weswegen ich beim Erarbeiten dieser Facharbeit auf ein nützliches Vorwissen zurückgreifen konnte. Ein Funktionsplotter ist dabei heruntergebrochen nichts anderes als ein sehr einfacher Compiler, der mathematische Terme in für den Computer verständliche Fragmente wandelt und daraus einen Graphen berechnet. Der Funktionsplotter verbindet damit theoretische & praktische Informatik mit den elementarsten Grundregeln der mathematischen Grammatik.

## Ziele des Funktionsplotters

Ziel ist es eine Anwendung zu entwickeln, welche verschachtelte mathematische Terme Analysiert und es möglich macht für beliebige Variablen Werte einzufügen, um aus dem Errechneten Term einen 2 Dimensionalen Graphen zu zeichnen.

Dabei sollen die Grundrechenarten der Arithmetik (Addition, Subtraktion, Multiplikation und Division), das Potenzieren, das Klammern und das Benutzen einiger Systemfunktionen (Wurzel, Sinus, Kosinus, Logarithmus) möglich sein.

## Grundlegende Problematik

Der mit Abstand komplizierteste Teil bei der Erstellung eines solchen Funktionsplotters, ist das Analysieren der mathematischen Terme. Diese können beliebig oft verschachtelt sein (Klammer in Klammer). Außerdem müssen die Grundlegende Gesetze der Mathematik für die Reihenfolge der Berechnungen (z. B. Punkt vor Strich, Assoziativgesetz) genau eingehalten werden. Der Fokus der Facharbeit liegt auf der Analysierung dieser Terme und der Herstellung des Parsers.  
Der Funktionsplotter wurde mit der Programmiersprache Java von Oracle und dem OpenJDK (Java Development Kit) 8u151 entwickelt. Diese Sprache eignet sich hervorragend da sie das nötige Informatische Paradigma der Objektorientieren und Funktionellen Programmierung einfach und leichtgewichtig implementiert hat, sowie eine GUI (Graphical User Interface) Bibliothek (AWT) im JDK enthalten ist. Als Entwicklungsumgebung wurde Jetbrains IntelliJ IDEA verwendet.

# Theoretische Grundlagen

Um die Funktionsweise hinter dem Funktionsplotter zu verstehen, benötigt man einige theoretische Grundlagen.

## Syntaxbaum

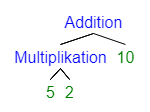
Ein Syntaxbaum bezeichnet eine baumfömige Darstellung der syntaktischen Struktur einer Zeichenkette, gemäß einer definierten Grammatik (z. B. die der Mathematik oder die von Programmiersprachen) [1] [2] [3]. In dem Fall des Funktionsplotters ist diese Zeichenkette die gegebene Funktion mit dem der Graphen berechnet werden soll.

Der Syntaxbaum besteht aus Knotenpunkten, welche nach oben zur Wurzel verkettet sind. Ein Knotenpunkt kann bis zu 2 weitere Knotenpunkte unter sich geordnet haben und repräsentiert dann eine Rechenoperation (z. B. eine Multiplikation) oder er besitzt keine weiteren untergeordneten Knotenpunkte (Endknoten) und besitzt damit dann einen Eigenwert (z. B. einen atomaren Wert (Zahl) oder eine Variable).

Jeder Knotenpunkt besitzt einen Wert welcher er an den übergeordneten Wert weitergibt. Dieser ergibt sich aus der Berechnung mit dem knotenspezifischen Operator und den untergeordneten Knotenpunkt. Bei Endknoten ist dies ein konstanter Wert.

Der Wert des gesamten Terms ist gleichzeitig der äußere Wert der Wurzel, da diese alle darunterliegende Knoten zusammenrechnet.

Der Term wird im Folgenden in einem Syntaxbaum dargestellt. „Addition“ ist hierbei die Wurzel, „Multiplikation“ ein Knotenpunkt mit einem Operator und die grün markierten Knoten sind die atomaren Endknoten.



Der Wert des Multiplikationsknoten ist hierbei 10 (5 \* 2) und der Wert der Wurzel und damit des ganzen Terms beträgt damit 20 (10 + 10).

Diese Baumstruktur eignet sich hervorragend um beliebig komplexe und verschachtelte Terme logisch darzustellen und in einem Computerprogramm auszurechnen.

## Parser

Der Parser ist der Teil des Funktionsplotters, welche die Aufgabe hat, den Syntaxbaum anhand der eingegebenen Zeichenkette zu errechnen.

Der Parser geht dabei Zeichen für Zeichen von links nach rechts vor und errechnet mit einer Funktionskette die Knotenpunkte. Dabei prüft dann jede Funktion den Zustand des aktuellen Zeichens (ist das aktuelle Zeichen z. B. ein Pluszeichen handelt es sich um eine Addition). Sollte dieser Zustand nicht zutreffen, wird an die nächste Funktion weitergegeben.

Die Grammatikalischen Regeln müssen hierbei Rückwärts angewandt werden.  
Die Ausdrücke die am leichtesten Klammern (Addition und Subtraktion) werden dabei dann als erstes gesucht, da die Ausdrücke die als erstes gefunden werden im Syntaxbaum ganz oben stehen und damit als letztes Ausgerechnet werden. [4]

# Praktische Umsetzung

## Mathematischer Kontext (MathContext)

Der mathematische Kontext ist der Behälter, welcher die Werte von definierte Variablen sowie Funktionen beinhaltet. In der Informatik nennt man diese Bereiche einen Scope. Jede Funktion besitzt ihren eigenen Scope. [5] [6]

Ist z. B. die Funktion gegeben und sie wird mit aufgerufen, so wird ein neuer mathematischer Kontext erstellt, welcher die Variable „x“ mit 10 definiert und der Rückgabewert beim Auflösen des Terms von 150 stimmig ist.

Dazu hat der mathematische Kontext eine Map (Key:Value Pair), in dem mit dem Namen der Variable als String, beliebige Werte gespeichert werden können.

Im Konstruktor des mathematischen Kontextes werden einige Systemfunktionen (sin, cos, tan, sqrt, log, usw.) und Konstanten (PI, e) definiert.

Des Weiteren gibt es einige Funktionen die es möglich machen mit der Map zu interagieren (Werte setzen und auslesen):

* setFunction(name, parameter, term): Dies definiert eine Benutzerdefinierte Funktion mit dessen Namen, Parameter sowie dem Term als Zeichenkette.

Wird also z. B. versucht die Funktion zu definieren, muss dies folgendermaßen geschehen:

createFunction("h", "x", "x^3 + 2x^2 + 5");

* setNativeFunction(name, function): Dies definiert eine interne Java Methode als Funktion im Funktionsplotter. Dafür benötigt man den Namen der Funktion und die Methode als Methodenpointer.

Um also die Methode Math.sin(double) im Funktionsplotter als sin(x) zu definieren, benötigt man:

setNativeFunction("sin", Math::sin);

* setVariable(name, value): Dies definiert eine Variable. Der Name kann dabei beliebig Lang sein (Man ist also nicht auf xyz beschränkt, sondern kann jedes beliebige Wort benutzen). Als value muss hierbei der Typ double angegebenen werden.
* getVariable(name): Dies gibt den Wert einer im Kontext definierten Variable zurück. Der Typ dieser Variable muss dabei einen Wert (double) haben und keine Funktion sein.
* callFunction(name, parameterValue): Dies ruft eine Funktion mit dem entsprechenden Parameter auf.

So wird aus : callFunction("g", 30);

## Implementation von Knotenpunkten (Node)

Knotenpunkte sind mit dem Interface Node implementiert. Sie haben im Funktionsplotter nur eine definierte Funktion, welche den Wert des Knotenpunktes mit dem mathematischen Kontext ausrechnet. Da es nur eine Funktion gibt, lassen sich Knotenpunkte im Parser mit einem Funktionellen Lambda [7]   
„(context) -> …“ darstellen.

Also z.B. „return (context) -> context.getVariable(„x“)“ ist ein Endknoten mit dem Wert „x“.

## Implementation des Parsers

Die Klasse Parser besitzt 2 wichtige Felder:

* „characters“ ist der Term der Analysiert werden soll als Zeichenkette (char[])
* „cursor“ ist die Position des Zeichens das gerade Untersucht wird (beginnt bei 0).

Außerdem besitzt der Parser neben der zentral parsenden Funktionskette weitere 4 Funktionen welche sich wiederholenden Code reduzieren:

* „trimInput()“ überspringt alle Leerzeichen und beginnt bei dem nächsten Zeichen.
* „hasMoreCharacters()“ gibt zurück ob der Cursor das Ende der Zeichenkette erreicht hat und ob weitere Zeichen folgen könnten.
* „checkNext(char)“ gibt zurück ob das derzeitig geprüfte Zeichen das im Parameter angegebenen Zeichen übereinstimmt.
* „demand(char)“ fordert, dass das aktuell geprüfte Zeichen mit dem im Parameter angegebene Zeichen übereinstimmt. Dann inkrementiert es

den Cursor (rückt also ein Zeichen weiter). Stimmt das aktuelle Zeichen nicht überein, wirft diese Funktion einen Fehler zurück.

*(Idee aus [8])*

Der zentrale Teil ist die parsende Funktionskette, welche mit der Funktion „parse()“ seine Einsprungsfunktion hat. Die Einsprungsfunktion gibt die Wurzel des Syntaxbaumes als Knotenpunkt (Klasse Node) zurück.

*(Idee aus [9], Zeile 96)*

Wie im theoretischen Teil schon beschrieben werden hier die Gesetze der gegebenen Grammatik mit einer Funktionskette rückwärts angewendet.

Die erste Funktion „parseAddition()“ der Funktionskette hat die Aufgabe Additionen und Subtraktionen (welche am leichtesten Klammern) zu erkennen und dementsprechende Knotenpunkte zurückzugeben. Dies geschieht indem als erstes eine mögliche linke Seite berechnet wird, indem die 2. Funktion der Funktionskette („parseFactor()“) aufgerufen wird. Dann wird geschaut ob überhaupt eine Addition oder eine Subtraktion vorliegt, indem geschaut wird ob das nächste Zeichen ein Pluszeichen oder ein Minuszeichen ist.

Ist dies der Fall, so wird eine Rechte Seite für die Addition oder Subtraktion errechnet, in dem die Methode sich selbst aufruft (Rekursion). Dort wird dann genau der gleiche Vorgang ausgeführt. So können selbst beliebig lange Additions- oder Subtraktionsketten am Ende (wieder bei dem ersten Funktionsaufruf angelangt) in einem Knotenpunkt enden. Ist dies nicht der Fall, so liegt keine Addition oder Subtraktion vor und die die Funktion gibt die errechnete mögliche linke Seite zurück.

*(siehe dazu Programmablaufplan für Funktion 1)*

*(Idee aus [9], Zeile 321 sumExpression())*

Die 2. Funktion der Funktionskette („parseFactor“) funktioniert genauso wie die erste Funktion, nur das statt nach Additionen und Subtraktionen, nach Multiplikationen und Divisionen gesucht wird.

Hier muss ebenfalls als erstes mit der unterliegenden Funktion („parseSign“) eine mögliche linke Seite errechnet werden, dann nach einem Multiplikationzeichen oder einem Divisionszeichen geprüft werden. Ist dies der Fall, ruft sich die Funktion selbst auf, um eine rechte Seite für den Operator zu errechnen und einen Multiplikations- oder Divisionsknoten zurückgegeben.

*(siehe dazu Programmablaufplan für Funktion 2)*

*(Idee aus [9], Zeile 337 term())*

Die 3. Funktion der Funktionskette („parseSign“) hat die Aufgabe Negativierungen festzustellen.

Minuszeichen bei z. B. bedeutet schließlich nicht, dass es sich um -x handelt, sondern, dass der gesamte Term x² negativ wird.

Die Funktion ist anders aufgebaut, als die ersten beiden Funktionen, da hier keine linke und rechte Seite benötigt wird, sondern lediglich der unterliegende Knotenpunkt negativiert wird.

So wird als erstes geprüft ob ein Minuszeichen Zeichen vorhanden ist. Ist dies der Fall wird ein unterliegender Knoten errechnet in dem sich die Funktion selbst aufruft. Daraus wird dann ein Negativknoten gebildet und zurückgegeben.  
Ist dies nicht der Fall, so wird die unterliegende Funktion („parsePower“) aufgerufen und zurückgegeben.

*(siehe dazu Programmablaufplan für Funktion 3)*

*(Idee aus [9], Zeile 370 signExpression())*

Die 4. Funktion der Funktionskette („parsePower“) hat die Aufgabe Potenzen festzustellen. Diese Funktion funktioniert wieder wie die 1. und 2. Funktion.

Es wird als erstes wieder eine mögliche linke Seite errechnet, in dem die weiter unterliegende Funktion („parseValue“) aufgerufen wird. Dann wird geprüft ob ein Potenzzeichen („^“) vorhanden ist. Ist dies der Fall, wird eine rechte Seite errechnet, in dem die Funktion sich wieder erneut aufruft und mit der linken und rechten Seite einen Potenzknotenpunkt bildet und diesen zurückgibt. Ist dies nicht der Fall wird der vorher errechnete mögliche linke Knotenpunkt zurückgegeben.

*(siehe dazu Programmablaufplan für Funktion 4)*

*(Idee aus [9], Zeile 360 factor())*

Die 5. Funktion („parseValue“) hat die Aufgabe atomische (Zahlen), Variablen (Wörter) sowie Funkionsaufrufe zu erkennen und als Endknoten zurückzugeben. Dazu wird als erstes geprüft ob das aktuelle Zeichen eine Ziffer ist. Ist dies der Fall, wird ein Speicher (StringBuffer) erstellt, in dem die Zahl als Wort zwischen gespeichert wird. Fortlaufend werden alle folgenden Zeichen auf Ziffern geprüft, und sollange an den Speicher gehangen, bis das aktuelle Zeichen keine Ziffer mehr ist. Dann wird geprüft ob ein Punkt vorliegt (für Dezimalzahlen). Ist dies der Fall, werden erneut alle folgenden Zeichen auf Ziffern geprüft und sollange als Nachkommastelle an den Speicher angehangen, bis keine Ziffer mehr das aktuelle Zeichen ist.

Aus dem Speicher wird dann per Double.parseDouble(speicher) ein Java Dezimal Double errechnet und damit ein Endknoten gebildet. Dieser wird dann zurückgegeben.

Sollte sich jedoch das erste Zeichen nicht um eine Ziffer handeln, so wird auf einen Buchstaben geprüft. Ist dieser vorhanden wird auch ein Speicher erstellt in welchem fortlaufend alle Buchstaben gespeichert werden. Ist das nächste aktuelle Zeichen dann kein Buchstabe mehr, wird geprüft ob eine Klammer (für Funktionsaufrufe) vorhanden ist. Ist dies der Fall wird die 1. Funktion aufgerufen um einen Funktionsparameter zu errechnen. Dann wird ein Endknoten als Funktionsaufruf zurückgegeben. Ist jedoch keine Klammer vorhanden gewesen, wird lediglich ein Endknoten mit der Referenz zur Variable zurückgegeben.

Sollte es sich jedoch weder um eine Ziffer noch um einen Buchstaben handeln, wird die 6. und letzte Funktion aufgerufen.

*(siehe dazu Programmablaufplan für Funktion 5)*

Die 6. Funktion hat die Aufgabe die Klammern zu parsen. Klammern haben in der Grammatik die Fähigkeit, alle anderen Gesetze aufzuheben, weswegen sie als letztes aufgerufen werden (damit am stärksten Klammern).

In der Funktion wird als erstes geprüft, ob es sich bei dem derzeitigen Zeichen um eine Klammer handelt. Ist dies der Fall, werden die inneren Zeichen der Klammer mit der 1. Funktion geparsed und als Knoten zurückgegeben. Klammern stellen keinen direkten Knotenpunkt dar, sondern beeinflussen lediglich in welcher Reihenfolge die Zeichen geparsed werden.

Sollte es sich bei dem ersten Zeichen jedoch nicht um eine Klammer handeln, so ist das Zeichen an dieser Stelle nicht definiert und es wird ein Fehler zurückgegeben und der gesamte Parsevorgang abgebrochen.

*(siehe dazu Programmablaufplan für Funktion 6)*

## 3.4 Implementation der Zeichnung des Graphen (GraphRenderer)

Der Hauptfokus der Facharbeit liegt auf der Bearbeitung und Implementation des Parsers und Knotenpunkte. Deshalb wird dieser Teil nur grob erklärt.

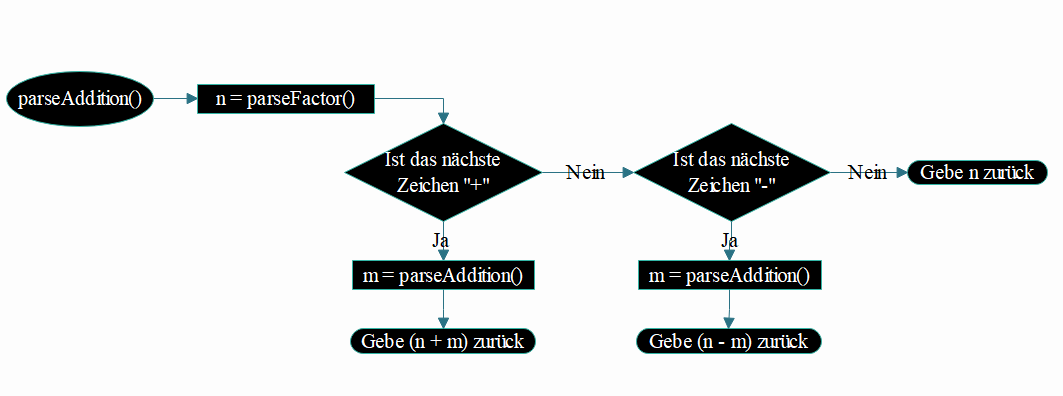
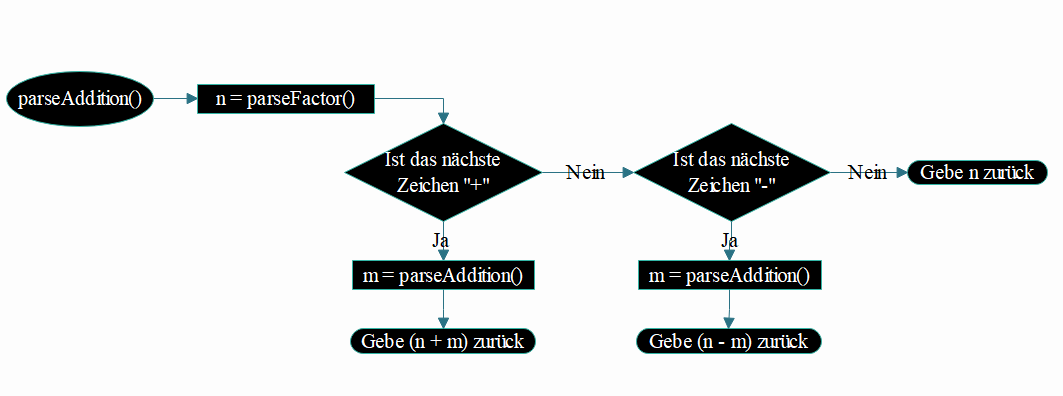
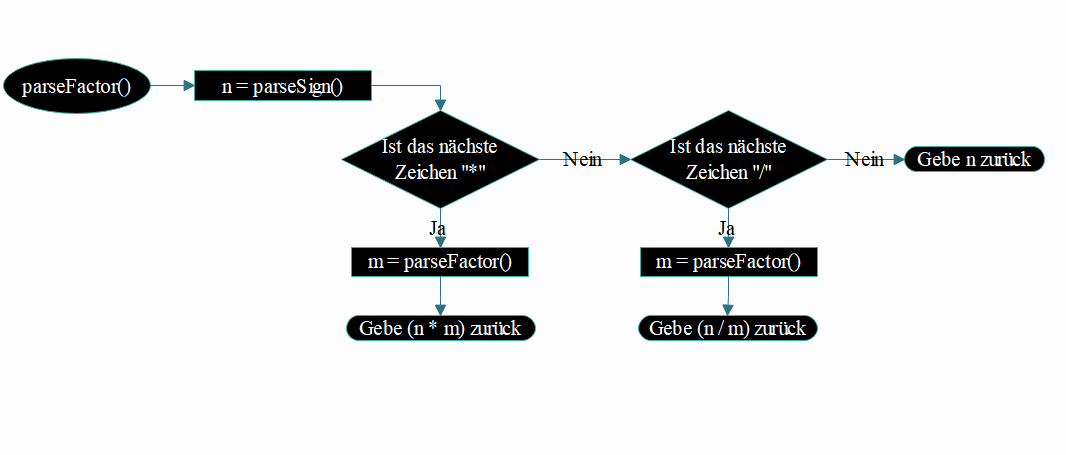
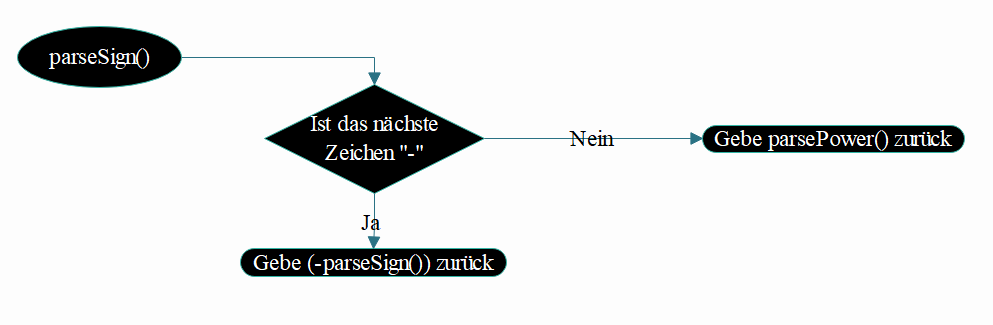
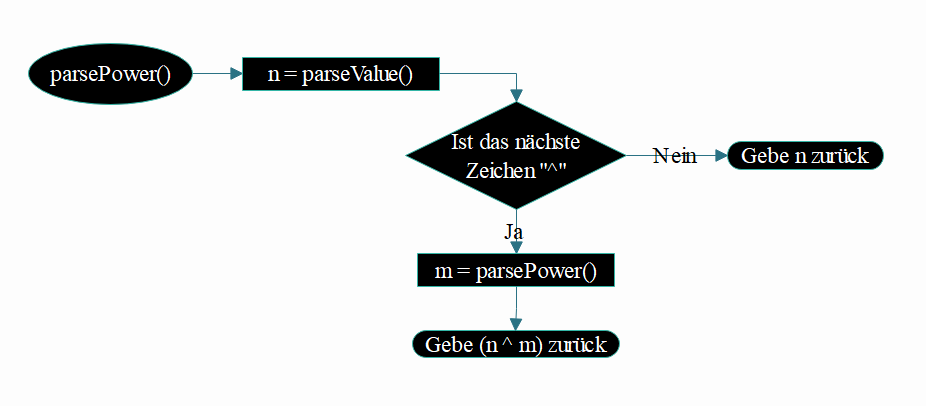
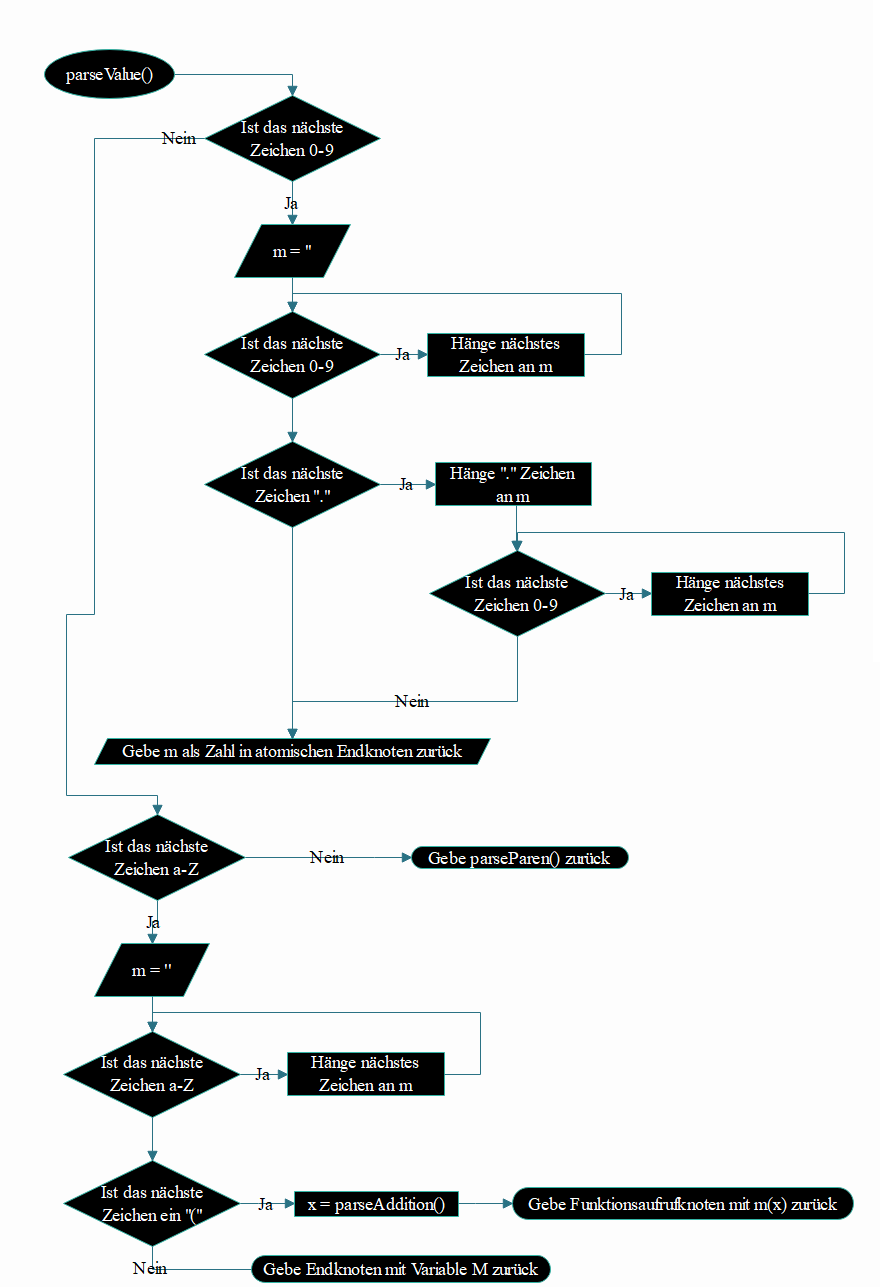
Die Klasse GraphRenderer besitzt lediglich eine Funktion („drawGraph“), welche verschieden Parameter zur Zeichnung des Graphens akzeptiert und eine Zeichnung in Form eines BufferedImage zurückgibt.

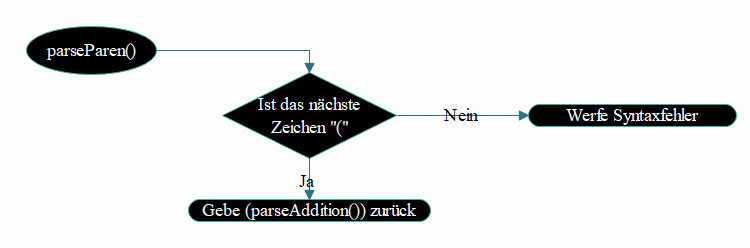
Dazu benötigt die Funktion die Größe der Zeichenfläche („canvasWidth“ und „canvasHeight“) sowie einen mathematischen Kontext („context“) in dem die Funktion die gezeichnet werden soll („function“ z. B. „f“) gezeichnet werden soll. Des Weiteren muss definiert werden, in welchem rahmen die Zeichenfläche dargestellt werden sollen. Es muss ein minimaler Punkt („startX“ und „startY“) sowie ein maximaler Punkt („endX“ und „endY“) definiert sein, nach welchem sich die Funktion richtet. Ist z. B. der minimalste Punkt (-200, -100) und der maximalste Punkt (200, 100), so ist der Bereich den die Zeichnung auf der X-Achse abdeckt 400 Einheiten und auf der Y-Achse 200 Einheiten. Die Funktion wird mit Werten von -200 bis 200 befüllt und die Werte werden per Dreisatz auf die Höhe angepasst (ein Y-Wert von 100 würde in dem Beispiel den obersten Punkt der Zeichnung bedeuten, ein Y-Wert von -100 den untersten Punkt in der Zeichnung bedeuten und ein Y-Wert von 150 würde bedeuten, dass der Punkt nicht mehr in der Zeichnung vorhanden ist.

# Literaturverzeichnis

|  |  |
| --- | --- |
| [1] | D. Wagner, „Karlsruher Institut für Technogolie,“ 1 März 2018. [Online]. Available: https://i11www.iti.kit.edu/\_media/teaching/winter2011/tgi/tgi1112\_t14.pdf. |
| [2] | „Wiki der Fakultät Informatik,“ 28 September 2006. [Online]. Available: http://www.iwiki.de/wiki/index.php/Syntaxbaum. [Zugriff am 1 März 2018]. |
| [3] | „Wikipedia,“ 21 Februrar 2018. [Online]. Available: https://en.wikipedia.org/wiki/Parse\_tree. [Zugriff am 1 März 2018]. |
| [4] | „Wikipedia,“ 28 Februar 2018. [Online]. Available: https://en.wikipedia.org/wiki/Parsing. [Zugriff am 1 März 2018]. |
| [5] | „Wikipedia,“ 9 Februar 2018. [Online]. Available: https://en.wikipedia.org/wiki/Scope\_(computer\_science). [Zugriff am 1 März 2018]. |
| [6] | D. Marshall, „users.cs.cf.ac.uk,“ [Online]. Available: https://users.cs.cf.ac.uk/Dave.Marshall/PERL/node52.html. [Zugriff am 3 März 2018]. |
| [7] | L. Rückemann, „codecentric,“ 21 Oktober 2013. [Online]. Available: https://blog.codecentric.de/2013/10/java-8-erste-schritte-mit-lambdas-und-streams/. [Zugriff am 1 März 2018]. |
| [8] | C. Zemek, „github,“ 20 August 2008. [Online]. Available: https://github.com/grom358/zemscript/blob/master/src/net/zeminvaders/lang/TokenBuffer.java. [Zugriff am 1 März 2018]. |
| [9] | C. Zemek, „github,“ 13 Mai 2010. [Online]. Available: https://github.com/grom358/zemscript/blob/master/src/net/zeminvaders/lang/Parser.java. [Zugriff am 1 März 2018]. |

# Anhang

1. Funktionsablaufplan zu Funktion 1 („parseAddition“):
2. Funktionsablaufplan zu Funktion 2 („parseFactor“):
3. Funktionsablaufplan zu Funktion 3 („parseSign“)
4. Funktionsablaufplan zu Funktion 4 („parsePower“)
5. Funktionsablaufplan zu Funktion 5 („parseValue“)
6. Funktionsablaufplan zu Funktion 6 („parseParen“)



# Selbständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst und keine anderen Hilfsmittel als angegeben verwendet habe. Insbesondere versichere ich, dass ich alle wörtlichen und sinngemäßen Übernahmen aus anderen Werken als solche kenntlich gemacht habe.

Unterschrift: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_