Entwicklung einer DSL zum Rechnen mit mathematischen Formeln für Anwendungen im Maschinellem Lernen

Studienarbeit

für die Prüfung zum

Bachelor of Science

des Studienganges Angewandte Informatik

an der

Dualen Hochschule Baden-Württemberg Karlsruhe

von

Sebastian Bernauer

Abgabedatum 20.05.2019

Bearbeitungszeitraum	5 + 6 Semster
Matrikelnummer	7390071
Kurs	TINF16B5
Ausbildungsfirma	United Internet AG Karlsruhe
Betreuer	Oliver Rettig

Erklärung

Ich versichere hiermit, dass ich meine Studienarbeit mit dem Thema: „Entwicklung einer DSL zum Rechnen mit mathematischen Formeln für Anwendungen im Maschinellem Lernen“ selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Ich versichere zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

Ort Datum	Unterschrift

Zusammenfassung

In der Robotik werden oft Zeitreihen gemessen. Die können z.B. die Koordination von Markern auf einem Robotergreifarm sein. De gemessenen Daten werden oft für maschinelles Lernen aufbereitet, beispielsweise um Anomalien zu erkennen. Bisher erfolgte dies auf verschiedenen Wegen.
In dieser Arbeit wird eine Domain-specific language (DSL) entwickelt und implementiert, die speziell auf Zeitreihen zugeschnitten ist und eine einfache Syntax bereit stellt.
Für die einfachere Verwendung der neuen Sprache PrePro wird zusätzlich ein Plugin mit Syntaxhervorhebung für die NetBeans IDE implementiert. Die Unterstürzung eines Debuggers könnte in Zukunft noch umgesetzt werden.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Liste der Codefragmente

KAPITEL 1

Motivation

Immer öfter werden Probleme der realen Welt mittels neuronaler Netze gelöst. Allerdings kann man in den meisten Fällen nicht einfach das neuronale Netz auf die gemessenen Daten angewendet werden. Die Daten müssen vorher aufbereitet werden und gegebenenfalls unwichtige Daten entfernt werden. Dies geschieht mittels verschiedener Frameworks in verschiedenen Sprachen. Es soll eine DSL entwickelt werden, die genau auf diesen Anwendungsfall zugeschnitten ist. Durch die DSL soll einen einheitliche Sprache geschaffen werden, welche das Vorprozessieren der Daten vereinfacht. Die Sprache soll plattformübergreifend sein und Central processing unit (CPU)- und Graphics processing unit (GPU)-Berechnungen ermöglichen.

KAPITEL 2

Überblick über Technologien

Das Aufbereiten der Daten für das neuronale Netz kann in mehreren Frameworks in mehreren Sprachen erfolgen. Oft wird für das Vorverarbeiten die gleiche Sprache wie für das neuronale Netz verwendet. Die gängigsten Frameworks sind in Tabelle 2.1 gelistet.

Framework	Sprache
Tensorflow	Python
DL4J	Java

Tabelle 2.1: Die gängigsten Frameworks für maschinelles Lernen

2.1 Unterschied Compiler, Transpiler und Interpreter

Compiler

Übersetzt von einer höheren Sprache in eine niedrigere Sprache.

Transpiler

Übersetzt zwischen zwei Sprachen mit ungefähr gleichem Abstraktionsgrad.

Interpreter

Führt Code einer höheren Sprache direkt aus ohne den Code in eine andere Sprache zu übersetzen.

2.2 ND4J

ND4J ist eine Framework für die Sprache Java, in welchem effiziente Matrizenoperationen durchgeführt werden können. Es kann auf der CPU oder GPU ausgeführt werden. In ND4J ist größtenteils nur das Konstrukt einer Matrix bekannt, Vektoren oder Skalare sind nur ein Spezialfall einer Matrix.

KAPITEL 3

Aufgabenstellung

3.1 Zu Grunde liegendes Framework

In dem Umfeld der Arbeit hat sich das Framework ND4J in der Programmiersprache Java für den Praxiseinsatz durchgesetzt. Daher soll die in dieser Arbeit entwickelte DSL auf diesem Framework aufbauen.

3.2 Compiler und Interpreter

Die DSL ist für das Vorprozessieren von Daten für maschinelles Lernen. Für die DSL kommt ein Compiler oder Interpreter in Frage. Ein Transpiler ist nicht möglich, da es keine in der Praxis verwendete Sprache für das Vorprozessieren der Daten gibt, in die übersetzt werden kann. In der Tabelle 3.1 werden die Implementierungsmöglichkeiten mittels Compiler und Interpreter gegenüber gestellt.
Wegen der Vorteile (vornehmlich das Debugging) von Interpretern im Vergleich zu Compiler soll PrePro als Interpreter implementiert werden.

Implementierung	**Vorteile**	**Nachteile**
Compiler	Generierter Java-Code kann auf jeder Java virtual machine (JVM) ausgeführt werden, es wird kein Interpreter benötigt.	Debugging ist nur in dem generierten Java-Code möglich.
Interpreter	Debugging leichter möglich	Möglicherweise nicht so performant

Tabelle 3.1: Vor- und Nachteile einer Implementierung mittels Compiler oder Interpreter

3.3 Verwendete Technologien

Die DSL wird mittels einem Interpreter ausgeführt. Dieser ist in Java (genauer: Groovy) geschrieben und baut auf folgenden Technologien auf:

ND4J

Matrizen-Berechnungen werden mittels dem ND4J-Framework durchgeführt.

Java

Das ND4J-Framework ist in der Programmiersprache Java verfügbar. Damit der Interpreter es verwenden kann, wird in dieser Sprache geschrieben.

Groovy

Groovy ist eine Sprache, die auf Java aufbaut und kompatibel ist. Sie unterstützt zum Beispiel dynamic dispatching[1].

Antlr

Antlr in der Version 4 wird für das Parser der eingegebenen Programme verwendet. Für das Netbeans-Plugin wird Antlr in der Version 3 verwendet.

[1] `https://en.wikipedia.org/wiki/Dynamic_dispatch`

KAPITEL 4

Design der DSL

4.1 Funktionen

In Programmiersprachen werden meist manche Codezeilen häufig benötigt. Anstatt diese Zeilen mehrfach zu kopieren, kann man diese Zeilen in eine sogenannte Funktion packen. Diese Funktionen können an beliebiger Stelle im Code aufgerufen werden. Auf diese Weise kürzt man den entstandenen Code, erhöht die Lesbarkeit und verhindert Kopierfehler.
Daher soll die zu entwickelnde DSL auch Funktionen unterstützen.

Parameter

Funktionen können auch parametrisiert werden, was bedeutet, dass bei dem Aufruf der Funktion Werte mitgegeben werden können. Jede Funktion hat ihren eigenen Variablen-Gültigkeitsbereich, das bedeutet, dass Funktionen ihren Variablen den gleichen Namen geben können, aber unterschiedliche Variablen verwenden. Wenn eine Funktion Werte übergeben bekommen möchte, so muss sie diese mitsamt ihrem Typ angeben.

Rückgabetyp

Funktionen können eine Wert zurückgeben. Dieser muss einen bestimmten Typ haben. In der DSL ist ein Rückgabetyp möglich und muss mittels “returns <Typ>” gekennzeichnet sein. Ist keine Angabe gemacht ist keine Rückgabe vorhanden.

Überladen von Funktionen

Eine Funktion bezeichnet man als überladen, wenn es mehrere Funktionen mit gleichen Namen, aber unterschiedlicher Zahl oder Art von Parametern gibt. In der DSL ist ein Überladen von Funktionen nicht vorgesehen, könnte aber nachträglich noch implementiert werden.

Ein Beispiel von verschiedenen Funktionen befindet sich in Codefragment 4.1 auf der nächsten Seite.

4.1.1 Main-Funktion

Jedes prozedurale Programm benötigt einen Einstiegspunkt, wo das Programm gestartet wird. Da die DSL ein Framework verwendet, welches in Java geschrieben ist, ist es nicht unwahrscheinlich, dass die zukünftigen Nutzer vorher in Java programmiert haben. Daher wurde als Einstiegspunkt des Programms - wie in Java - eine Main-Funktion gewählt. In der DSL besitzt sie keine Parameter. Um Daten in sein Programm zu laden wurde der Ansatz eines DataSets gewählt, mehr dazu in Kapitel 4.4 auf Seite 8.

4.1.2 Weglassen der Main-Funktion

Es gibt mehrere Gründe, warum die Definition einer Main-Funktion unnötig ist:

- Es soll nur ein einziger arithmetischer Ausdruck ausgewertet werden.
- Kompatibilität mit bisher bestehenden anderen Tools, die keine Funktionen bieten, sondern nur eine Liste von Anweisungen entgegennehmen.

Deshalb wird die Main-Funktion in Pre-Processing: PrePro ist die Bezechnung der DSL, die aus dieser Arbeit heraus entstanden ist (PrePro) als optional gesehen und muss nicht deklariert werden. Es reicht aus die Befehle untereinander zu schreiben.
Trotzdem wird es als guter Stil erachtet, eine Main-Funktion zu deklarieren.

```
function main() {
  import vec3 p1, vec3 p2, vec3 p3;

  vec3 x = calculateDifference(p1, p2);
  vec3 s = calculateDifference(p1, p3);
  vec3 y = s X x;
  vec3 z = y X x;


  printResults(x, y, z);

  export x, y, z;
}
```

```
function calculateDifference(vec3 p1, vec3 p2) returns vec3 {
  return p2 - p1;
}

function printResults(vec3 x, vec3 y, vec3 z) {
  print x;
  print y;
  print z;
}
```

Codefragment 4.1: Beispiel Funktionen

4.2 Variablen

Die DSL ist für den Einsatz auf Zeitreihenberechnungen ausgelegt. Daher stellt in der DSL jede Variable eine Zeitreihe dar. Die Operationen der DSL sind immer auf Zeitreihen definiert.
Beispielhaft wird der Ausdruck `x = a - b;` angenommen. In diesem Fall sind `a` und `b` gemessene Zeitreihen von Sensordaten. Die entstehende Variable `x` ist wiederum eine Zeitreihe, welche durch elementweise Subtraktion jedes Zeitelements entstanden ist.
Der Vorteil liegt darin, dass der simple Ausdruck `x = a - b;` sehr leicht les- und wartbar ist. Wenn jede Variable keine Zeitreihe, sondern ein einzelner Messpunkt wäre, müsste man eine Schleife verwenden oder sich eigene Methoden definieren bzw. (falls in der Sprache möglich) die Operatoren überschreiben.
Variablen haben in der DSL immer einen Typ. Bei dem Anlegen einer Variablen muss dieser auch immer definiert werden. Ein Typ ist zum Beispiel ein Vector3 (vec3) oder eine Matrix (mat). Ein Vector3 ist eine Zeitreihe von Vektoren mit der Länge 3, eine Matrix eine Zeitreihe von Matrizen. Ein dem Programm zur Verfügung gestellter Vektor der Länge 3 kann nun als Vector3 oder auch als Matrix aufgefasst werden. Daher muss dem Interpreter beim Anlegen der Variablen immer der Typ mitgeteilt werden. Wenn die Variable schon existiert, muss der Typ nicht erneut angegeben werden.
Ein Beispiel befindet sich in Codefragment 4.2 auf der nächsten Seite.
Das Import-Statement wird in Kapitel 4.4 auf der nächsten Seite erläutert, relevant ist an dieser Stelle nur, dass mit dem Import Daten aus einem DataSet geladen werden.

```
import vec3 p1, vec3 p2, vec3 p3;

vec3 x = p2 - p1;
vec3 s = p3 - p1;
vec3 y = s X x;
vec3 z = y X x;
```

Codefragment 4.2: Beispiel Variablenzuweisung

4.3 DataSet

Ein DataSet ist in der DSL eine Sammlung von Variablen der DSL. In das DataSet können beliebig viele Variablen unter ihrem Namen gespeichert werden. Es ist nicht möglich zwei Variablen mit dem gleichen Namen abzulegen. Eine Variable kann bequem aus dem DataSet ausgelesen werden.

4.4 Import & Export

Ein Programm, das nur Berechnungen anstellt erscheint auf den ersten Blick sinnlos. Das Programm muss die Möglichkeit haben, Daten zu lesen und zu schreiben. Im Falle der DSL wird ein eigenes DataSet definiert. Das Programm erhält bei der Ausführung ein DataSet und gibt als Ergebnis wieder ein DataSet zurück. In das Eingabe-DataSet werden alle Variablen gespeichert, die für die Berechnungen benötigt werden. In dem Ausgabe-DataSet sind anschließend alle Variablen gespeichert, die berechnet wurden.
Die Verwendung eines DataSet hat gegenüber dem Hereingeben mittels Parametern in die Main-Funktion folgende Vorteile:

- Es gibt nur einen Rückgabetyp (DataSet). Andernfalls müsste ein Konstrukt ersonnen werden, mehrere Variablen von der Main-Funktion zurückgeben zu lassen.

- Einfacher Aufruf der Main-Funktionen (ab 4 Parametern wird der Funktionsaufruf unübersichtlich[2]). Statt 20 Parameter zu übergeben kann übersichtlich das DataSet zusammengebaut werden und als einziges Argument übergeben werden.

- Einfaches “Weiterschleifen” von DataSets zwischen mehreren PrePro-Programmen. Falls mehrere PrePro-Programme nacheinander ausgeführt werden kann bequem das Ausgabe-DataSet des ersten Programms als Eingabe-DataSet des zweiten Programms genommen werden.

4.4.1 Optionale Imports

Es soll möglich sein, Importe in die Main-Funktion als optional zu kennzeichnen. Diese werden mit dem Schlüsselwort `optional` gekennzeichnet. Eine Beispiel mit optionalen Imports ist in Codefragment 4.3 gegeben. Wenn die angegebene, zu importierende Variable nicht in dem DataSet existiert, wird keine Fehlermeldung erzeugt, sondern die Variable nicht importiert.
Achtung: Wenn ein Import optional ist und die Variable nicht in dem DataSet existierte, wurde die Variable nicht geladen. Wenn nun in dem Programm versucht wird auf die Variable zuzugreifen wird eine Fehlermeldung erzeugt, dass die Variable nicht definiert sei. Deshalb ist es wichtig, die Gültigkeit der Variablen mittels der `exists`-Funktion zu prüfen, die im nächsten Kapitel vorgestellt wird. Wenn bei dem konstruieren des Ausführungspfad nicht aufgepasst wird, kann auf eine nicht geladene Variable zugegriffen werden. Das ist vergleichbar mit Nullpointern, die in eine Funktion hereingegeben werden können und zunächst mal auf `!= null` geprüft werden müssen.
Deshal ist es definitiv nicht ratsam, alle Imports als optional zu kennzeichnen.

```
function main() {
  import vec3 p1, optional const quaternion_1, optional mat4
     notPresent;

  const isPresent = exists(p1);
  const isNotPresent = foo();

  export p1, quaternion_1, isPresent, isNotPresent;
}

function foo() returns const {
  return exists(notPresent);
}
```

Codefragment 4.3: PrePro-Code mit optionalen Imports

4.4.2 Die exists-Funktion

Mittels der Funktion `exists(<VariablenName>)` kann abgefragt werden, ob diese Variable geladen wurde. Wurde die Variable geladen, gibt die Funktion eine 1 zurück, andernfalls eine 0. Nach Abschluss der Berechnungen in Codefragment 4.3 auf der vorherigen Seite hat die Variable `isPresent` der Wert 1, die Variable `not Present` den Wert 0. Dies ist in folgendem Fall nützlich: In eine Programm werden zwei Koordinaten-Zeitreihen A und B gegeben. Im Idealfall möchte man auf dem Mittelpunkt zwischen diesen Werten arbeiten. Ist jedoch die Variable A nicht definiert, soll für die Berechnung B verwendet werden, und anders herum. Eine kleine Abweichung wird hier in Kauf genommen. Dieser Fall mag ungewöhnlich wirken, warum sollte man den Code nicht einfach an A oder B anpassen? Dies gibt mehr Sinn, wenn man ein Skript allgemeine verwenden möchte, und häufig genutzte Berechnungen auszulagern. In diesem Fall möchte man nicht jedes Mal die Berechnung aufs neue anpassen, sondern die Funktion einmal allgemein schreiben und die Ausführung abhängig von den gegebenen Daten machen.
Das oben genannte Beispiel mit A und B könnte in etwa so ausgedrückt werden. Der gegebene Code in Codefragment 4.4 ist nicht unbedingt mit dem PrePro-Interpreter ausführbar, soll aber das Prinzip verdeutlichen.

```
import optional vec3 a, optional vec3 b;

vec3 center = (exist(a) && exist(b)) * ( a + b ) / 2
|| a * exist(a)
|| b * exist(b);
```

Codefragment 4.4: PrePro-Code mit optionalen Imports

Falls die Variable a und b beide existieren ergibt der Ausdruck `(exist(a) && exist(b))` eine 1, andernfalls eine 0. Allerdings beinhaltet das Codefragment 4.4 ein großes Problem: Falls a existiert und b nicht, ergibt der Ausdruck `(exist(a) && exist(b))` eine 0, der Interpreter rechnet dann weiter, also `* ( a + b ) / 2`. Das Ergebnis einer Multiplikation ist immer null, soweit würde alles passen, da durch das `||` die anderen Fälle probiert

werden. Das Problem ist: Der Interpreter versucht `* ( a + b ) / 2.` zu berechnen und erzeugt - vollkommen zurecht - eine Fehlermeldung, dass b nicht definiert ist.
Zur Lösung dieses Problems gibt es 2 Möglichkeiten:

1. **Lazy Multiplikation**
   Es wird ein neuer Multiplikations-Operator eingeführt, der den rechten Teil nur berechnet, wenn der linke Teil != 0 ist. Dadurch würde nach dem Teil `(exist(a) && exist(b))` die Berechnung abgebrochen, da der linke Teil 0 ergibt. Durch das Abbrechen wird nicht versucht Variablen zu verwenden, die nicht definiert sind. Für diesen neuen Multiplikations-Operator muss ein neues Symol definiert werden, z.B. "**".

   Vorteile:

   - Einfache Implementierung.
   - Wiederverwendung des Operators an anderer Stelle möglich.

   Nachteile:

   - Die Nutzer müssen einen zusätzlichen Operator kennen und verstehen.

2. **Dedizierte If-Statements**
   In die DSL PrePro wird noch das Konstrukt einer If-Verzweigung eingebaut. Der beispielhafte PrePro-Code würde somit wie in Codefragment 4.5 aussehen.

```
import optional vec3 a, optional vec3 b;

if(exists(a) && exists(b)) {
    vec3 center = ( a + b ) / 2;
} else if (exists(a)) {
    vec3 center = a;
} else if (exists(b)) {
    vec3 center = b;
}
<<OPTIONAL>>
else {
  throw "Neither a or b was defined";
}
```

Codefragment 4.5: PrePro-Code mit If-Statements und der exists-Funktion

Vorteile:

- Leichter lesbar und verständlich.
- Auch für andere Fälle als die Multiplikation ist das If-Statement verwendbar.

Nachteile:

- Erhöht die Anzahl der Codezeilen dramatisch.
- Höherer Aufwand bei der Implementierung

4.5 Kommentare

Kommentare sollen in der DSL möglich sein.
Einen Zeilen-Kommentar wird ein “//” vorangestellt.
Ein Block-Kommentar wird mir “/*” und “*/” umschlossen.

KAPITEL 5

Implementierung der DSL

5.1 Lexer & Parser

Lexer und Parser werden beide von einer ANTLR4 Grammatik erzeugt. Hierfür wird eine kombinierte Grammatik verwendet, die zugleich Lexer und Parser erzeugt.
Die Grammatik ist unter `src/main/java/de/sbernauer/prepro/parser` abgelegt. Mittels dem Befehl `./generateLexerAndParser.sh` im Root-Verzeichnis des Projekts werden die benötigten Klassen von ANTLR aus der Grammatik generiert. Dies ist notwendig, wenn Änderungen an der Grammatik durchgeführt wurden, damit diese in den Interpreter übernommen werden.

5.2 Abstrakter Syntaxbaum

Ein AST ist ein Baum, der bei dem Parsen der Grammatik der DSL aufgebaut wird. Dieser wird anschließend von dem Interpreter abgearbeitet / ausgeführt.
Der Baum besteht aus Knoten, die in einer Baumstruktur an einer Wurzel hängen. Jeder Knoten besitzt wiederum eine beliebige Anzahl an Kindknoten, diese kann auch 0 sein. Es gibt einen einzigen Wurzelknoten, an dem der gesamte AST aufgehängt ist.
In Abbildung 5.1 auf der nächsten Seite ist ein beispielhafter AST abgebildet. Es ist ersichtlich, dass ein AST schnell unübersichtlich wird. Der in Abbildung 5.1 auf der nächsten Seite abgebildete AST wurde aus dem PrePro-Code in Codefragment 5.1 auf der nächsten Seite erzeugt. Trotz, dass der ursprüngliche Sourcecode überschaubar ist, ist der entstandene AST fast nicht mehr lesbar. Dementsprechend groß werden die ASTs bei längeren Programmen. Da die ASTs allerdings von dem Parser aufgebaut und von dem Interpreter abgearbeitet werden bekommt der Nutzer diese nie zu Gesicht.

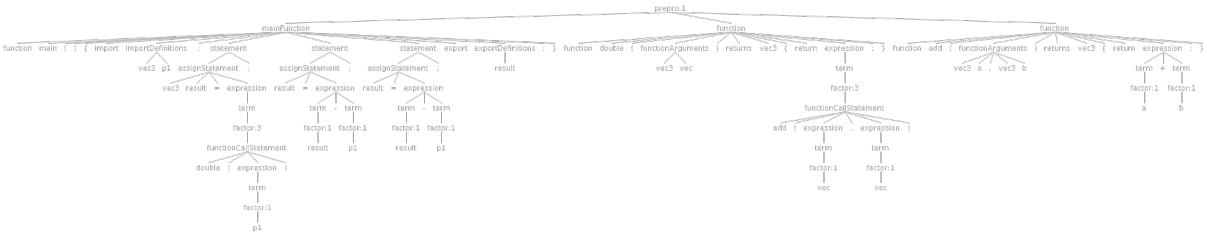

Abbildung 5.1: Beispielhafter AST

```
function main() {
    import vec3 p1;

    vec3 result = double(p1);
    result = result - p1;
    result = result - p1;

    export result;
}

function double(vec3 vec) returns vec3 {
    return add(vec, vec);
}

function add(vec3 a, vec3 b) returns vec3 {
    return a + b;
}
```

Codefragment 5.1: PrePro-Code für den beispielhaften AST

Ein AST-Baum besteht nicht aus lauter gleichen Knoten, sondern aus vielen verschiedenen Klassen. In PrePro gibt es eine Vielzahl an verschiedener Knoten-Typen, welche durch Java-Klassen repräsentiert werden. Die Klassenhierarchie ist in Abbildung 5.2 auf der nächsten Seite abgebildet.
Die dort aufgeführten Packages werden in den nachfolgenden Kapiteln im Detail erläutert.

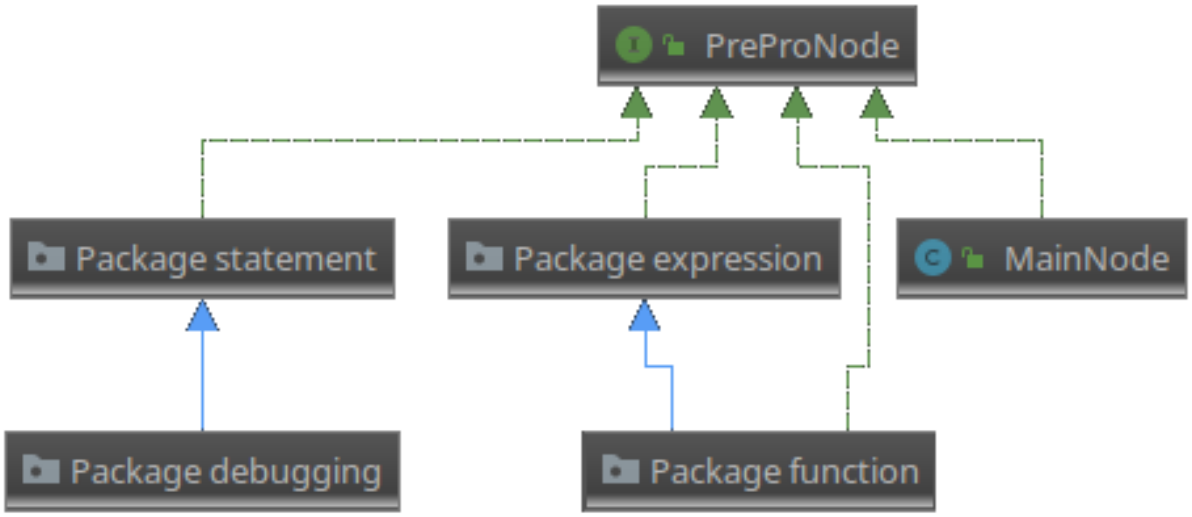

Abbildung 5.2: Die Knoten-Klassenhierarchie

5.2.1 Wurzelknoten

Die `MainNode` ist der Wurzelknoten des AST. Er beinhaltet als Kinder alle Funktionen, die in dem Programm definiert sind. In Abbildung Abbildung 5.1 auf der vorherigen Seite sind die 3 Teilbäume ersichtlich, die den 3 Funktionen aus Codefragment 5.1 auf der vorherigen Seite entsprechen.
Alle Knoten müssen das `PreProNode`-Interface implementieren. Dieses ist ein leeres Interface, besitzt somit keine Funktionen und dient nur dazu, alle Knoten als Knoten zu markieren.

5.2.2 Funktionen

In PrePro gibt es Funktionen. Direkt unter dem Wurzelknoten sind die Funktion-Knoten aufgehängt. Sonst hängt nichts an dem Wurzelknoten. Jede Funktion wird durch einen eigenen Knoten mitsamt Kindern dargestellt. Zum Starten des Programms wird die Main-Funktion aus dem AST herausgesucht und gestartet. Eine Funktion hat Argumente, die ihr übergeben werden, diese müssen als Unterknoten in dem AST gespeichert werden, damit sie bei der späteren Ausführung mit in die Funktion rein gegeben werden können. Es wurde eine allgemeines `Function`-Interface definiert, welches die Funktionen `String getFunctionName()` und
`Variable execute(Arguments arguments, FunctionTable functionTable)` besitzt.

Das hat den Vorteil, dass das Interface unabhängig davon verwendet werden kann, ob die Funktion im Sourcecode definiert wurde oder von PrePro in Java-Code global zur Verfügung gestellt wurde. Im ersteren Fall erzeugt der Parser eine `FunctionNode`, im zweiten Fall wird die Funktion in reinem Java spezifiziert. Die `FuntionNode` führt die im Sourcecode angegebeben Statement-Nodes aus, die `CustomFunctionNode` ruft Java-Funktionen mittels Reflection auf.
Alle eben genannten Klassen sind im `nodes.function` Package abgelegt.

5.2.3 Statements

Eine Funktion besteht aus einer Reihe von Statements. Eine Statement ist eine Anweisung, die ausgeführt wird und keine Rückgabe besitzt. Dies grenzt sie von Expressions ab, die einen Wert zurückgeben. Die Zeile `a = 1 + 2;` ist in ihrer Gesamtheit ein Statement, da es keine Rückgabe.
Als allgemeines Statement wurde die abstrakte Klasse `StatementNode` definiert. Sie besitzt die Funktion
`void execute(SymbolTable symbolTable, FunctionTable functionTable)` Alle Klassen, die von dieser Klasse erben, müssen die `execute`-Funktion implementieren. Dazu zählt z.B. die `PrintStatementNode`, welche in der `execute`-Funktion einen Ausgabe auf dem Bildschirm erzeugen.

5.2.4 Expression

Die Teil `1 + 2` der Zeile `a = 1 + 2;` ist eine Expression. Eine Expression ist ein (mathematischer) Ausdruck, der ausgeführt werden kann und einen Rückgabewert besitzt. In einer Expression können neben mathematischen Operatoren, Konstanten, Variablen und Funktionsaufrufe beinhalten. Die abstrakte Superklasse `ExpressionNode` definiert die Funktion
`Variable execute(SymbolTable symbolTable, FunctionTable functionTable)`.
Wichtig ist an dieser Stelle, dass die Funktion im Gegensatz zu der Funktion bei der `StatementNode` einen Wert vom allgemeinen Typ `Variable` zurück liefert. Alle dafür nötigen Klassen sind in dem Package `nodes.expression` abgelegt.

5.2.5 Debugging

In PrePro ist zunächst eine rudimentäre Debugging-Möglichkeit eingebaut. Es ist möglich an einer beliebigen Zeile im Code ein `break;` zu schrieben. Bei der Ausführung stoppt der Interpreter in dieser Zeile und bietet eine kleine Kommandozeile um Variablen zu lesen und zu modifizieren. Dieses Break-Statement ist ein Statement, erbt also von der `StatementNode`. In der überschriebenen `execute`-Funktion wird auf die Eingabe und das Weiterspringen des Nutzers gewartet.

5.3 Typsystem

Das Typsystem von PrePro ist in Abbildung 5.3 auf der nächsten Seite dargestellt. Alle Variablen erben von er abstrakten Klasse `Variable`. Es gibt die Untertypen `Vector`, `Marix`, `Scalar` und `Constant`. Die Unterklassen `Vector` und `Matrix` haben wiederum Unterklassen für drei- und vierelementige Varianten. Diese Unterklassen sind wichtig, da z.B. eine `Matrix3` mit einem `Vector3` multipliziert werden kann, allerdings nicht mit einem `Vector4`.
In der abstrakten Klasse `Variable` sind die Funktionen `add`, `sub`, `mul` und `div` definiert. Werden die Funktionen auf der abstrakten Klasse aufgerufen, werfen sie eine Exception, dass die mathematische Operation nicht definiert sei.
Die Unterklassen haben nun die Möglichkeit, Operationen mit anderen Typen zu definieren. Mittels Polymorphie und dynamic dispatching wird bei einer arithmetischen Operation die passende Funktion gesucht. Falls keine passende Funktion wird die allgemeine - in der abstrakten Klasse definierte - Funktion verwendet, und daraufhin eine Exception geworfen, dass die mathematische Operation nicht definiert sei.

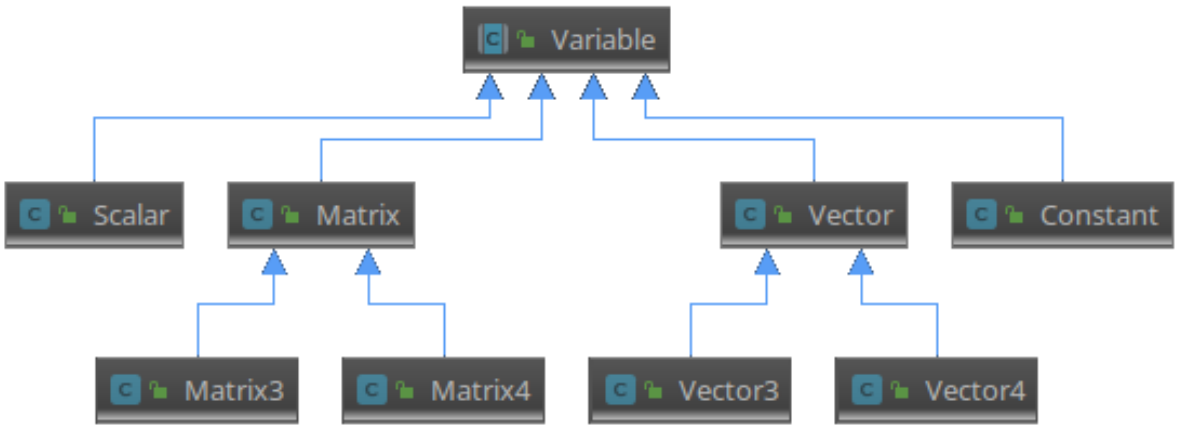

Abbildung 5.3: Typsystem von PrePro

5.3.1 Darstellung der Typen in Java

Alle Typen werden als `INDArray` von `ND4J` dargestellt.
Die Unterklassen besitzen einen Konstruktor, der ein `INDArray` übergeben bekommt. In den Konstruktoren wird jeweils geprüft, ob die Dimensionen dem entsprechenden Typ entsprechen (z.B eine Matrix4 muss eine 4x4-Matrix übergeben bekommen). Die einzige Ausnahme bildet der Typ "Constant", dieser bietet zusätzlich einen Konstruktur, dem ein `double` übergeben werden kann. Intern wird aus dem `double` ein `INDArray` erzeugt.

5.4 Implementierung der Operationen auf den Variablen-Typen

In dem vorherigen Abschnitt wurden verschiedene Variablen-Typen herausgearbeitet. Diese sind in Abbildung 5.3 abgebildet.
Zwischen manchen der Typen sind Operationen, wie z.B. das Kreuzprodukt definiert. Manche Typen können allerdings nicht sinnvoll kombiniert werden, z.B. eine Matrix3 und eine Matrix4. Diese Operationen müssen in Java implementiert werden. Ein erster Ansatz ist es, jedem Variablen-Typ die möglichen Operationen der Java-Klasse hinzuzufügen. Dieser Ansatz wurde in dem Codefragment 5.2 beispielsweise für die `Matrix4`-Klasse angewandt. Die Methoden sind jedoch lange nicht vollzählig.

```
public class Matrix4 {

        // same for sub, mul and div
        public Matrix4 add(Matrix4 right) {
                return new Matrix4(/* ... */);
        }

        // same for sub, mul and div
        public Matrix4 add(Constant right) {
                return new Matrix4(/* ... */);
        }

        // Cross-Product
        public Vector4 mul(Vector4 right) {
                return new Vector4(/* ... */);
        }
}
```

Codefragment 5.2: Implementierung der Operatoren in der Java-Klasse Matrix4

Es ist zu erkennen, dass sehr viele Funktionen angelegt werden müssen. Damit der Java-Compiler korrekt compilieren kann, müssen diese Funktionen in der Basisklasse `Variable` definiert werden und in den Unterklassen überschrieben werden. Dies hat den Vorteil, dass in der Basis-Funktion in der Basis-Klasse `Variable` als normales Verhalten eine Fehlerausgabe erzeugt, in der gesagt wird, dass diese Operation auf diesen Typen nicht möglich ist. In den Subklassen werden nun die Operatoren überschrieben und so das Standardverhalten der Fehlermeldung überschrieben.
Dieser Ansatz hätte zusätzlich den Vorteil, dass man Operatoren auch auf Klassen in der Mitte der Hierarchie - z.B. ein allgemeiner Vektor - anwenden kann. So kann die Vektoraddition an einer zentralen Stelle für `Vector3` und `Vector4` erfolgen.

5.4.1 Dynamic Dispatching

In der Theorie würde der aufgeführte Ansatz perfekt funktionieren.
Warum in der Theorie? Dies allgemeine Problem wird in Codefragment 5.3 ersichtlich.

```
public class Main {

    public static void main(String[] args) {
        Dog dog1 = new Dog();
        test(dog1);
        Animal dog2 = new Dog();
        test(dog2);
    }

    private static void test(Animal a) {
        System.out.println("I'm an animal");
    }

    private static void test(Dog g) {
        System.out.println("I'm an dog");
    }
}

class Animal { }

class Dog extends Animal { }
```

Codefragment 5.3: Java-Code zur Illustrierung von Dynamic Dispatching

Das erwartete Verhalten wäre zwei Mal die Ausgabe “I’m a dog”.
Die tatsächliche Ausgabe ist aber “I’m a dog” gefolgt von “I’m a animal”.
Warum das? Java entscheidet schon zur Kompilierzeit, welche Funktion aufgerufen wird.
Da zur Kompilierzeit noch nicht klar sein kann, welcher genaue Typ sich in der Variable

`dog2` befindet, wird die die am besten passende Funktion für den deklarierten Typ der Variable verwendet. In diesem Fall ist die Variable `dog2` als `Animal` deklariert, deshalb wird die Funktion mit dem Parameter-Typ `Animal` ausgewählt.
Die Auswahl der Methode zur Laufzeit anhand des konkreten Typs statt zur Kompilierzeit nennt man `Dynamic Dispatching`.

5.4.2 Groovy

Eine Programmiersprache, die Dynamic Dispatching unterstützt ist Groovy. Groovy wurde 2007 offiziell herausgegeben und ist syntax-kompatibel zu Java. Der Code aus Codefragment 5.3 auf der vorherigen Seite kann 1:1 in Groovy ausgeführt werden und erzeugt dort die erwartete Ausgabe von zwei mal “I’m a dog”. Groovy sucht sich die aufzurufende Funktion zur Laufzeit anhand des konkreten Typs raus.
Da durch das Dynamic Dispatching die Implementierung deutlich lesbarer ist, soll für PrePro eine Sprache eingesetzt werden, welche Dynamic Dispatching einsetzt. Hierfür würden prinzipiell verschiedene Sprachen in Betracht kommen (z.B. C++, Go oder Rust). Allerdings müsste man für die drei genannten Sprachen das Projekt neu schreiben, auch steht die Verwendung dieser Sprachen in Konflikt mit den Anforderungen in Kapitel 3.2 auf Seite 3, da das verwendete Framework in Java geschrieben ist.
An dieser Stelle bietet sich Groovy an, da es syntax-kompatibel zu Java ist. Das bedeutet, das bestehende Projekt in Java kann mit nur sehr geringfügigen Änderungen in ein Groovy-Projekt transferiert werden. Groovy kann auch ohne Probleme Java-Code ausführen, kann also das verwendete Java-Framework auch ansprechen. Allerdings wurden aus Performancegründen nur die Klassen, die Dynamic Dispatching benötigen in Groovy transferiert, der Rest der Klassen wurden Java-Klassen gelassen.
Es wird allerdings darauf geachtet, dass der geschriebene Code Java-syntax-kompatibel ist (Es wäre auch möglich Sprachkonstukte in Groovy zu verwenden, die in Java nicht möglich sind, hierauf wurde verzichtet). Dadurch wird die Lesbarkeit erhöht, da keine ständiger Wechsel zwischen Java und Groovy-Syntax notwendig ist.

5.4.3 Bauen von Java und Groovy Projekten

Da die ND4J-Bibliothek in Java geschrieben ist, wird versucht - sofern möglich - alles in Java zu schreiben. Groovy wird nur in den Klassen verwendet, wo es benötigt wird. Hierfür ist zu beachten, dass dies nicht nur die Klassen mit den Variable-Typen selber sind, sondern alle Klassen, die die Funktionen mittels Dynamic Dispatching aufrufen wollen. Dadurch entsteht ein Projekt, welches Java und Groovy-Sourcecode beinhaltet. Das PrePro-Projekt wird mittels Maven verwaltet und kompiliert. Maven musste dafür das Plugin `org.codehaus.gmavenplus.gmavenplus-plugin` hinzugefügt werden, welches das Compilieren übernimmt.

5.5 FunctionTable

Ein PrePro-Programm besteht aus mehreren Funktionen. Die Funktionen werden - auf den Namen indiziert - in einer FunctionTable gespeichert. Aktuell ist diese als `HashMap<String, Function>` implementiert, weshalb Funktionen anhand ihrem Namen identifiziert werden. Damit ist aktuell kein Überladen von Funktionen möglich. Falls ein Überladen ermöglicht werden soll, muss statt dem Namen ein Tupel aus Namen und Signatur der Funktion verwendet werden. Auch muss bei dem Aufruf einer Funktion nicht nur nach dem Namen, sondern auch der Signatur geschaut werden.
Aufgrund dem Aufwand der Implementierung wurde dies in der ersten Version von PrePro nicht umgesetzt. Die Sprache wurde jedoch mit dem Gedanken im Hinterkopf entwickelt, sodass die oben genannten Maßnahmen umgesetzt werden können und ein Überladen möglich wird.

5.6 SymbolTable

In einem PrePro-Programm können Variablen als Zwischenspeicher verwendet werden. Die Symboltabelle ist bei dem PrePro-Interpreter ein Speicherplatz für diese Variablen. Die Symboltabelle ist als `HashMap<String, Variable>` implementiert. Aus ihr können die aktuellen Werte der Variablen gelesen und geschrieben werden. Durch Auslesen des Typs, der in der Symboltabelle gespeichert ist, kann der Typ der Variable ermittelt werden.
Jede Funktion erstellt zu Beginn der Ausführung der Funktion eine Symboltabelle und fügt die übergebenen Parameter hinzu. Anschließend wird auf der Symboltabelle die Funktion

abgearbeitet. Ist die Funktion fertig abgearbeitet wird der Rückgabewert aus der Symboltabelle berechnet, die Symboltabelle wird anschließend verworfen. Hat eine Funktion keinen Rückgabewert, entfällt der Berechnungsschritt für die Rückgae dementsprechend. Es ist möglich eine bereits gefüllte Symboltabelle bei dem Start eines PrePro-Programms mitzugeben. Diese steht dann in der Main-Funktion direkt zur Verfügung. So kann z.B. eine Symboltabelle mit Konstanten einmal aufgebaut und dann von mehreren PrePro-Programmen wiederverwendet werden.

5.7 Grundrechenarten

Bei der Darstellung der Zahlen im Speicher wird für jede Zahl ein Double verwendet. Das erhöht die Genauigkeit gegenüber einem float und erspart Konvertierungen zwischen Ganz- und Fließzahlen. Grundrechenarten sind die Addition, Subtraktion, Multiplikation und Division von Matrizen. Diese Operationen sind elementweise und werden direkt von ND4J zur Verfügung gestellt und können aufgerufen werden.

5.8 Kreuzprodukt

Anders als die vier Grundarten aus dem vorherigen Abschnitt wird das Kreuzprodukt von ND4J nicht als Operation angeboten. Das Kreuzprodukt wird in der Praxis allerdings zu Beispiel für das Aufspannen von Vektoren im dreidimensionalen Raum benötigt. Daher wird an dieser Stelle das Kreuzprodukt zweier Vektoren selber implementiert. Falls ND4J in Zukunft den Operator Kreuzprodukt bereit stellt, kann in zukünftigen Varianten auf ihre Implementierung zugegriffen werden, da diese wahrscheinlich effizienter sein wird. Die eigene Implementierung des Kreuzprodukts kann auf folgende Arten geschehen:

1. Implementierung in Plain Java mittels einer for-Schleife.
2. Implementierung in Plain Java mittels Subarrays

5.8.1 Implementierung Kreuzprodukt mittels For-Schleife

Bei der Implementierung mittels der For-Schleife werden die eigentlichen Berechnungen in Java durchgeführt. Als erstes wird ein Double-Array als Zwischenspeicher für das Ergebnis angelegt. Es besitzt (Anzahl der Zeitelemente in den Eingabe-Vektoren) * 3 Elemente. Die

Anzahl der Elemente entspricht so der Anzahl der Elemente der Ergebnismatrix, die Daten können in dem Double-Array effizient gespeichert werden. Das Array wird im Anschluss in eine Matrix konvertiert. Für die Berechnung der Elemente wird mittels einer For-Schleife über alle Zeilen der Matrix iteriert. In jeder Zeile werden die 3 Werte des entstehenden Ergebnisvektors berechnet und in das Double-Array gespeichert. Abschließend wir das Double-Array in eine Matrix mit den Dimensionen [<Anzahl Zeitelemente> x 3] konvertiert und durch eine Vector3-Wrapper-Klasse als Vector mit 3 Werten gekennzeichnet. Der Algorithmus ist in Codefragment 5.4 dargestellt.

```
private Vector3 crossProduct(Vector3 left, Vector3 right) {
    INDArray a = left.getNdArray();
    INDArray b = right.getNdArray();

    int size = a.shape()[0];
    double[] result = new double[size * 3];

    for (int i = 0; i < size; i++) {
        result[i * 3 + 0] = a.getDouble(i, 1) * b.getDouble(i, 2) -
        ↪  a.getDouble(i, 2) * b.getDouble(i, 1);
        result[i * 3 + 1] = a.getDouble(i, 2) * b.getDouble(i, 0) -
        ↪  a.getDouble(i, 0) * b.getDouble(i, 2);
        result[i * 3 + 2] = a.getDouble(i, 0) * b.getDouble(i, 1) -
        ↪  a.getDouble(i, 1) * b.getDouble(i, 0);
    }
    return new Vector3(Nd4j.create(result, new int[]{size, 3}));
}
```

Codefragment 5.4: Implementierung Kreuzprodukt mittels for-Schleife

5.8.2 Implementierung Kreuzprodukt mittels Subarrays

Bei der Implementierung mittels Subarrays wird nur auf ND4J-Funktionen zurückgegriffen, da diese optimiert sind. Die Implementierung ist in Codefragment 5.5 auf der nächsten Seite zu finden.

```
private Vector3 crossProductSubArray(Vector3 left, Vector3 right) {
    INDArray a = left.getNdArray();
    INDArray b = right.getNdArray();

    INDArray a1 = a.getColumn(0);
    INDArray a2 = a.getColumn(1);
    INDArray a3 = a.getColumn(2);

    INDArray b1 = b.getColumn(0);
    INDArray b2 = b.getColumn(1);
    INDArray b3 = b.getColumn(2);

    INDArray c1 = (a2.mul(b3)).sub(a3.mul(b2));
    INDArray c2 = (a3.mul(b1)).sub(a1.mul(b3));
    INDArray c3 = (a1.mul(b2)).sub(a2.mul(b1));

    int size = a.shape()[0];
    INDArray result = Nd4j.create(size, 3);
    result.putColumn(0, c1);
    result.putColumn(1, c2);
    result.putColumn(2, c3);

    return new Vector3(result);
}
```

Codefragment 5.5: Implementierung Kreuzprodukt mittels for-Schleife

5.8.3 Vergleich der Implementierungen für Kreuzprodukt

Beide Impelemtierungsmöglichkeiten haben Vor- und Nachteile. Diese sind in Tabelle 5.1 auf der nächsten Seite aufgeführt.

Implementierungs-möglichkeit	**Vorteile**	**Nachteile**
Mittels For-Schleife	Leichter verständlich.	Wird direkt in Java ausgeführt. Mögliche Optimierungen von ND4J können nicht verwendet werden. Berechnungen finden nur auf der CPU statt!
Mittels Subarray	Durch die Verwendung von ND4J können die Optimierungen verwendet werden. Wenn ND4J so konfiguriert ist, dass es auf der GPU läuft, kann die eigentliche Berechnung weiterhin auf der GPU erfolgen.	Schwerer verständlich.

Tabelle 5.1: Vor- und Nachteile einer Implementierung mittels Compiler oder Interpreter

Für große Zeitreihen müsste sich die Implementierung mittels dem Subarray als effizienter erweisen, besonders wenn die Berechnungen auf der GPU durchgeführt werden. Als Nachweis und für das Effizienzverhalten bei verschiedenen Zeitreihen wurde ein Benchmark durchgeführt. Die Ergebnisse sind in Tabelle 5.2 festgehalten.

Anzahl Datensätze	For-Schleife	Subarray
100	6 (342)	4 (13)
1.000	20 (489)	7 (22)
10.000	53 (527)	5 (21)
100.000	333 (1159)	5 (15)
1.000.000	3205 (4129)	47 (64)
10.000.000	32358 (33594)	442 (453)

Tabelle 5.2: Benchmark-Ergebnisse der verschiedenen Implementierungsmöglichkeiten für das Kreuzprodukt. Die Messung ist nach 5 Durchläufen in Millisekunden gemessen, die Zahl in Klammern gibt die Zeit des ersten Durchlaufs an.

Es ist erkennbar, dass die For-Schleife stets langsamer ist als die Implementierung mittels Subarrays. Das Verhalten war aufgrund der Optimierung der ND4J-Funktionen erwartet. Interessant ist an dieser Stelle, dass die Laufzeitoptimierung von Java bei kleiner Anzahl Datensätze die Laufzeit nach 5 Schleifendurchläufen drastisch reduzieren konnte. Bei großer Anzahl Datensätze war dies allerdings nicht mehr möglich.
Da die Implementierung mittels Subarrays deutlich schneller ist, wurde sie für die DSL verwendet.

5.9 Tests

Für PrePro wurden keine Komponenten-Tests, sondern Integrationstests geschrieben. Hierbei wurde sich an Tests an einer Referenzimplementierung einer DSL von Oracle orientiert. Die Implementierung ist unter [4] zu finden.
Ein Test besteht aus zwei Dateien:

1. Einem Test-PrePro-Skript
2. Einer toString()-Ausgabe des erzeugten PreProDataSets.

Für jeden Test wird das gleiche PreProDataSet vorbereitet. Es besteht aus ein paar Variablen verschiedenen Typs, diese stehen in allen Tests zur Verfügung. Das gegebene PrePro–Skript, das getestet werden soll, wird mit diesem DataSet ausgeführt. Das Skript gibt alle benutzten und errechneten Variablen mittels dem Export-Statement zurück.
Die toString()-Funktion des PreProDataSets wurde so gestaltet, dass sie alle in dem

DataSet gespeicherten Variablen ausgibt. Wenn die toString()-Funktion von zwei PrePro-DataSets übereinstimmt, kann davon ausgegangen werden, dass die gleichen Variablen darin gespeichert sind.

5.9.1 Hinzufügen eines neuen Tests

Für einen neuen Test müssen folgende zwei Schritte durchgeführt werden:

1. Hinzufügen eines PrePro-Skriptes unter `src/test/tests`. Dieses Verzeichnis wird von dem TestExecutor nach Tests durchsucht, der Test muss nicht registriert werden. Das Skript kann unter einem beliebigen Subverzeichnis unter der Dateieinendung `*.prepro` gespeichert werden. Falls gewünscht, können die Tests ausgeführt werden und im Log geschaut werden, ob der Test korrekterweise ausgeführt wurde (`mvn test`).

2. Bereitstellen des erwarteten Ergebnisses mittels toString()-Funktion des entstandenen PreProDataSets. Der String wird in eine Datei mit der Dateiendung `*.out` gespeichert. Wichtig ist, dass der selbe Dateiname (ohne Endung) wie bei Schritt 1.) verwendet wird, damit der TestExecutor die Ausgabe dem PrePro-Skript zuordnen kann.

KAPITEL 6

Integration in NetBeans-IDE

Syntaxhervorhebung und das Anzeigen von Syntaxfehlern erleichtert die Entwicklung von Programmen. Deshalb soll dieser Komfort dem Nutzer zur Verfügung gestellt werden. Für die Implementierung gibt es zwei Möglichkeiten:

1. Implementierung eines eigenen Editors, der den Support für Syntaxhervorhebung und -fehler bietet.
   Vorteil: Man ist nicht auf bestehende Editoren angewiesen, und kann jedes Detail so gestalten, wie man es möchte.
   Nachteil: Alles muss von Grund auf implementiert werden.

2. Wiederverwendung eines bestehenden Editors (z.B. Netbeans), und Hinzufügen des Supports.
   Vorteil: Der Nutzer kann so einfacher zwischen verschiedenen Programmiersprachen wechseln, ohne den Editor wechseln zu müssen.
   Nachteil: Der Editor muss eine Möglichkeit bieten den Support hinzuzufügen. Auch findet ein geringer Overhead statt, da man den Editor erst entsprechend konfigurieren und ist gezwungen die Schnittstellen zu nutzen - egal ob diese möglicherweise umständlich zu verwenden oder veraltet sind.

Aufgrund der einfacheren Benutzbarkeit und dem Verzicht einer Implementierung von Grund auf wurde die zweite Variante verwendet. Hierfür wurde ein Editor gesucht, der Support für DSLs bietet. Die Wahl fiel auf Netbeans, da für die Plattform eine große Community mit verschiedenen Tutorials und Beispielen existiert.[1]

6.1 Syntaxhervorhebung

Für die Syntaxhervorhebung in Netbeans gibt es mehrere Tutorials. Die älteren der Tutorials nutzen `javacc` als Lexer und Parser. Die neueren Tutorials verwendeten hingegen `Antlr`. Als Grundlage wurde das Tutorial unter [5] sowie Anmerkungen dazu unter [1]

[1] Die Community ist zu finden unter `https://netbeans.org/community/index.html`

aus dem Jahr 2009 verwendet. Da das Tutorial und die Anmerkungen dazu aus dem Jahr 2009 stammen, sind sie nicht mehr aktuell. Dadurch war die Umsetzung nicht so einfach wie anfangs angenommen.
Das Tutorial verwendet `Antlr` in der Version 3, zum Zeitpunkt des Schreibens ist die aktuelle Version 4.7.2, welche nicht abwärtskompatibel ist. Es wurde zwar versucht das Tutorial mit der neueren Version von `Antlr` umzusetzen, was jedoch gescheitert ist, da sonst Internas hätten umgeschrieben werden müssen, da einige Funktionen aus der Version 3 nicht mehr in der Version 4 verfügbar sind.

6.1.1 Neue Antlr-Grammatik für Syntax

Da der PrePro-Interpreter `Antlr` in der Version 4 verwendet, war es aufgrund der fehlenden Abwärtskompatibilität nicht möglich, die gleiche Grammatik für Interpreter und die Syntax-Features zu verwenden (welche Version 3 verwenden).
Das ist nicht so schlimm, wie es auf den ersten Blick wirkt, da die Grammatiken andere Anwendungsfälle abbilden:
Interpreter-Grammatik: Die Grammatik prüft nicht nur, ob die Syntax korrekt ist, sondern baut auch den AST auf. Dies geschieht in Java-Code und stellt einen Großteil der Grammatik dar.
Syntax-Grammatik: Diese Grammatik prüft nur die Syntax, es wird nichts aufgebaut, somit ist auch kein Java-Code vorhanden. Allerdings müssen alle Symbole als Token mit Namen exportiert werden. Zum Beispiel darf das Symbol “,” nicht als unbenanntes Token exportiert werden, sondern es muss einen Namen (in diesem Fall: “COMMA” besitzen.) Das hat zur Folge, dass in den einzelnen Grammatik-Regeln nicht einfach das Symbol “,” verwendet werden darf, sondern immer “COMMA” geschrieben werden muss.
Damit die Unterschiede klarer werden sind in Codefragment 6.1 auf der nächsten Seite und Codefragment 6.1 auf der nächsten Seite beispielhafte Grammatik-Regeln abgebildet. Erkennbar sind folgende Unterschiede:

- Die Regeln der Syntax-Grammatik sind deutlich kürzer. Die Interpreter-Grammatik hat 225 Zeilen, wobei hiervon 170 Zeilen die Grammatik-Regeln beinhalten.
  Die Syntax-Grammatik hat 105 Zeilen, wobei hiervon 36 Zeilen die Grammatik-Regeln beinhalten.

Die Regeln sind im Schnitt also fast 5-mal so groß in der Interpreter-Grammatik als in der Sytax-Grammatik.

- Die Regeln der Sytax-Grammatik fokussieren sich auf das Wesentliche.
- Jedes Literal in der Syntax-Grammatik hat sein eigenes, benanntes Token.

```
>>> INTERPRETER:
exportDefinitions returns [List<String> result]:
                                { $result = new ArrayList<>(); }
IDENTIFIER                      { $result.add($IDENTIFIER.text); }
(
    ','
    IDENTIFIER                  { $result.add($IDENTIFIER.text); }
)*
;

>>> SYNTAX:
exportDefinitions: IDENTIFIER (COMMA IDENTIFIER)*;
```

Codefragment 6.1: Grammatik-Regel “exportDefinitions” Interpreter vs Syntax

```
>>> INTERPRETER:
functionCallStatement returns [FunctionCallNode result]:
IDENTIFIER
'('                         { List<ExpressionNode>
  ↪ expressionList = new ArrayList<>(); }
(
    expression              { expressionList.add($expression.
      ↪ result); }
    (
        ',' expression      { expressionList.add($expression.
          ↪ result); }
    )*
)?
')'                         { $result = new FunctionCallNode(
  ↪ $IDENTIFIER.text, expressionList); }
```

```
;

>>> SYNTAX:
functionCallStatement: IDENTIFIER LEFTPAREN (expression (
   ↪ COMMA expression)*)? RIGHTPAREN;
```

Codefragment 6.2: Grammatik-Regel “functionCallStatement” Interpreter vs Syntax

6.1.2 Umsetzung

Das NetBeans-Modul teilt sich in zwei Submodule auf: `syntaxhighlighting` und `syntaxerror`. Der Sourcecode ist in 4 Packages unterteilt:

syntaxhighlighting
: Hier sind alle Klassen, die für das Hervorheben der Tokens in dem Sourcecode zuständig sind.

syntaxhighlighting.parser
: Die Komponenten Parser und Lexer mitsamt ihrer Antlr3-Grammatik sind hier abgelegt. Diese werden sowohl von dem syntaxhighlighting als auch dem syntaxerror Package verwendet.

syntaxhighlighting.syntaxerror
: Diese Komponenten erzeugen mehrere `ErrorDescription`-Objekte, die NetBeans hinweisen, wo sich welche Sytaxfehler befinden. NetBeans färbt die Syntaxfehler daraufhin rot ein.

syntaxhighlighting.utils
: In dem utils-Package sind zwei Helferklassen. Eine Klasse stellt welche eine Bridge zu dem Antlr-Framework dar. Die zweite Klasse liest bei dem Starten automatisch alle Token aus dem generierten Token-File von Antlr ein. Dadurch müssen nicht alle Token in dem Sourcecode hart codiert werden.

6.1.3 Pflege der Grammatiken

Aufgrund der aktuellen Implementierung mittels zwei separaten Grammatiken müssen beide Grammatiken gepflegt werden, es findet also ein geringer Overhead statt.

Empfehlenswert ist es, seine Änderung in der Interpreter-Grammatik durchzuführen, und erst ganz am Schluss die Änderungen in die Syntax-Grammatik zu übernehmen. Die ist wichtig, damit die Änderungen auch syntaktisch in dem Plugin unterstützt werden.
Dieses Vorgehen wurde schon mehrfach selbst angewandt und stellte keine Problem dar, da meist nur eine Zeile in der Syntax-Grammatik geändert werden musste. Dies ist möglich durch die kompakte Schreibweise in der Syntax-Grammatik.

6.1.4 Mögliche Verbesserungen

Das Plugin verwendet aktuell Antlr in der Version 3 für den Syntax-Support und sollte auf die neuere Version Antlr 4 aktualisiert werden.
Dies ist allerdings nicht ohne Probleme möglich, da Funktionen, die in dem Tutorial [5] verwendet werden, in der neueren Version von Antlr nicht mehr verfügbar sind.
Wenn dieser Schritt abgeschlossen ist, können die unterschiedlichen Grammatiken - für den Interpreter und die Syntax - zusammengeführt werden. Das erschwert zwar in geringem Maß die Lesbarkeit - z.B. muss jedes Token einen Namen bekommen - erleichtert aber die Pflege, da nur noch eine statt zwei Grammatiken gepflegt werden müssen.

6.2 Anzeigen von Syntaxfehler

Neben dem Hervorheben der verschiedenen Schlüsselwörter und -typen ist es für den Programmierer auch nützlich auf Fehler in seiner Syntax unmittelbar hingewiesen zu werden. Die verhindert den Effekt, dass eine große Menge Code geschrieben wird, und anschließend festgestellt wird, dass in der ersten Zeile ein Syntaxfehler ist. Nun kann es durch den langen Code sein, dass die Erinnerung an die erste Zeile verblasst und, und eine erneute Einarbeitung in die erste Zeile nötig ist, um den Syntaxfehler zu finden.
Wird allerdings der Fehler sofort hervorgehoben, kann man ihn beheben, solange man die Zeile noch im Arbeitsgedächtnis hat, eine Einarbeitung entfällt.
Die Umsetzung wurde auch an das Tutorial aus [5] angelehnt.

6.3 Debugging

Debugging bezeichnet den Prozess ein Programm schrittweise auszuführen, um Fehler in dem Programmablauf zu finden. Um dem Nutzer eine einfachere Benutzung von PrePro

step	Springe über die aktuell ausgewählt Zeile zur nächsten Zeile
vars	Liste alle aktuell existierenden Vriablen mitsamt ihrem Inhalt (String-Repräsentation) auf
exit	Beende den Debugger
eval	Führe den Code aus, der als Parameter dem Befehl mitgegeben wird

Abbildung 6.1: Mögliche Befehle bei dem Kommandozeilen-Debugger

zu bieten sollte ihm eine Debugging-Möglichkeit geboten werden.
Es gibt verschiedene Möglichkeiten das Debugging grafisch zu gestalten:

1. Debugging in einer Kommandozeile, mit der Möglichkeit Befehle einzugeben
2. Debugging in einem grafisch aufwändigen Editor.

6.3.1 Debugging mittels Kommandozeile

Bei dem Debugging mittels der Kommandozeile stehen verschiedene Befehle bereit. Diese sind in Tabelle 6.1 aufgeführt.
Die Implementierung ist dafür relativ einfach, da keine Oberfläche entwickelt werden muss, sondern nur Tastatureingaben eingelesen werden müssen. Die Ausgabe erfolgt textuell.
Der Nachteil für den Nutzer besteht darin, dass die aktuelle Zeile nicht hervorgehoben wird. Dafür wird jederzeit die aktuelle Zeile angezeigt.

6.3.2 Debugging im grafischen Editor

In dem Editor wird die aktuelle Zeile hervorgehoben und zusätzlich weitere Informationen zum Programmablauf angezeigt. Der Nutzer kann so viel bequemer debuggen. Der grafische Editor hat den Nachteil einer deutlich aufwändigeren Implementierung. Hier gilt allerdings das Gleiche, wie bei dem Syntax-Support: Die Implementierung reduziert sich deutlich, wenn auf bestehende Editoren mit dem Support für benutzerdefiniertes Debugging aufgebaut werden kann. Da der Support für die Syntax schon in NetBeans umgesetzt wurde, ist es an dieser Stelle sinnvoll auch NetBeans zu verwenden, da somit alle Funktionalität für den Nutzer in einem Editor gebündelt ist.
Eine Anleitung für das Implementieren eines eigenen Debuggers ist unter [3] zu finden. Die konkrete Implementierung sei zukünftigen Arbeiten überlassen.

KAPITEL 7

Ausblick

7.1 Verbesserungen an PrePro

An dieser Stelle werden ein paar Punkte genannt, in denen das Projekt PrePro verbessert und weiter entwickelt werden kann.

7.1.1 Überladen von Funktionen

Überladen ist das Definieren von Funktionen mit dem gleichen Namen, die sich nur in der Zahl und Art ihrer Parameter unterscheiden. So ist es z.B. möglich eine `add(vec3 a, vec3 b)`-Funktion und eine `add(mat3 a, mat3 b)`-Funktion zu definieren. Abhängig davon, ob ein Vektor oder eine Matrix als Parameter übergeben wird, wird die passende Funktion aufgerufen. Dies ermöglicht es, den Code lesbarer zu gestalten.
Aktuell sind die Funktionen in PrePro nicht überladbar, dies könnte aber in einer zukünftigen Arbeit hinzugefügt werden.
Für die Umsetzung müsste die `FunctionTable` angepasst werden. Diese speichert zur Zeit die Funktionen in einer `HashMap<String, Function>`. Diese HashMap muss durch eine eigene Implementierung ersetzt werden, welche die Argumenttypen mit einbezieht.
Die Methode
`public Function getFunction(String functionName)` in der `FunctionTable` müsste um die Argumenttypen als Parameter erweitert werden. Sie könnte dann z.B. so aussehen:
`public Function getFunction(String functionName, Class... parameterTypes)`

7.1.2 Sämtliche Operatoren für die Datentypen implementieren

Während der Erstellung der Arbeit war es leider zeitlich nicht möglich, alle Operationen, die mit den Datentypen durchgeführt werden können zu implementieren. Hier muss noch nachgearbeitet werden, damit zukünftigen Nutzern sämtliche Operationen zur Verfügung stehen.
Die Operationen sind in den Groovy-Klassen der entsprechenden Datentypen (z.B.

`Vector3`) zu implementieren. An dieser Stelle sei auf die bisherigen Operatoren als Referenz verwiesen. Wichtig ist es an dieser Stelle, auch Tests für den Operator zu schreiben, damit seine korrekte Funktionalität dauerhaft sichergestellt werden kann.

7.1.3 Performance der Operationen verbessern

Das Thema Performance ist keine einfaches Gebiet, man kann sich sehr lange damit beschäftigen. Im Allgemeinen sollte man versuchen, möglichst viele Operationen mit ND4J-Operationen auszudrücken. Diese wurden von ND4J bereits optimiert und laufen - je nach Umgebung - sogar auf der GPU. Daher sollte es vermieden werden, in einer Operation eine Java-Schleife zu verwenden, da diese zwangsläufig auf der CPU ausgeführt wird, und somit den Vorteil der schnelleren GPU zunichte macht.
Trotzdem gibt in manchen Operatoren Java-Schleifen. Diese sollten entfernt werden und durch ND4J-Befehle ersetzt werden.

7.1.4 Portieren in die Truffle/Graal Architektur

7.1.5 Debugger implementieren

Wie bereits in Kapitel 6.3.1 auf Seite 34 angedeutet wurde, wäre eine Debugger für PrePro-Skripte in der NetBeans-IDE wünschenswert. Die Gründe und eine Anleitung ist in dem genannten Kapitel verlinkt.

7.2 Verbesserungen an der umgebenden Infrastruktur

Für den Fall, das viele Nutzer PrePro nutzen, kann die umgebende Infrastruktur angenehmer für den Nutzer gestaltet werden. So senkt sich die Einstiegshürde für neue Benutzer.

7.2.1 PrePro-Artefakt ins Maven Central laden

Alle benötigten Abhängigkeiten werden von Maven standardmäßig aus dem Maven Central geladen. Es ist eine große Sammlung von Artefakten, welche bequem per Maven-Project Object Model (POM) heruntergeladen und eingebunden werden können. Ein Hochladen

in das Maven Central ermöglicht es, dass Nutzer ganz einfach PrePro in ihren Projekten verwenden können.

7.2.2 NetBeans-Plugin in öffentliche Plguin-Sammlung laden

NetBeans bietet online eine Sammlung von Plugins, welche seht bequem in der NetBeans IDE heruntergeladen und installiert werden können. Somit entfällt das aufwendige Verteilen einer Datei und das Importieren dieser in NetBeans.

Abkürzungsverzeichnis

AST Abstract Sytax Tree v

CPU Central processing unit 1

DSL Domain-specific language i

GPU Graphics processing unit 1

JVM Java virtual machine 3

POM Project Object Model 36

PrePro Pre-Processing: PrePro ist die Bezechnung der DSL, die aus dieser Arbeit heraus entstanden ist 6

Literatur

[1] Joe Areeda. *Antlr Notes*. 2009. URL: http://wiki.netbeans.org/Antlr_Notes [besucht am 07. 03. 2019] [siehe S. 29].

[2] Rober C. Martin. *Robert C. Martin's Clean Code Tip of the Week #10: Avoid Too Many Arguments*. 2009. URL: http://www.informit.com/articles/article.aspx?p=1375308 [besucht am 21. 02. 2019] [siehe S. 8].

[3] Andreas Stefik. *New Language Support Tutorial Antlr*. 2010. URL: https://dzone.com/articles/how-reuse-netbeans-debugger [besucht am 13. 03. 2019] [siehe S. 34].

[4] Graal Team. *A simple example language built using the Truffle API*. 2019. URL: https://github.com/graalvm/simplelanguage [besucht am 16. 05. 2019] [siehe S. 27].

[5] unbekannt. *New Language Support Tutorial Antlr*. 2009. URL: http://wiki.netbeans.org/New_Language_Support_Tutorial_Antlr [besucht am 07. 03. 2019] [siehe S. 29, 33].