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

TODO

Inhaltsverzeichnis

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 Domain-specific language (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 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 11.

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 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.
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.

4.5 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.

5.2 Abstrakter Syntaxbaum

Ein Abstract Sytax Tree (AST) ist ein Baum, der bei dem Parsen 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. In PrePro gibt es eine Vielzahl an verschiedener Knoten-Typen.
Die Klassenhierarchie ist in Abbildung 5.1 auf der nächsten Seite abgebildet. Als oberstes Element ist das Interface `PreProNode` definiert. Die `MainNode` ist der Wurzelknoten des AST.

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

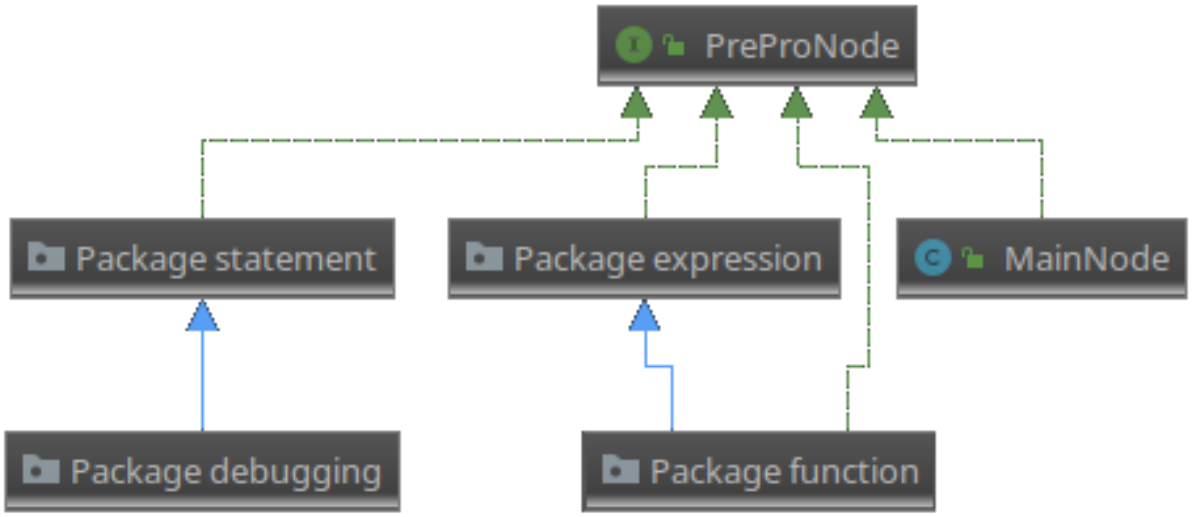

Abbildung 5.1: Die Knoten-Klassenhierarchie

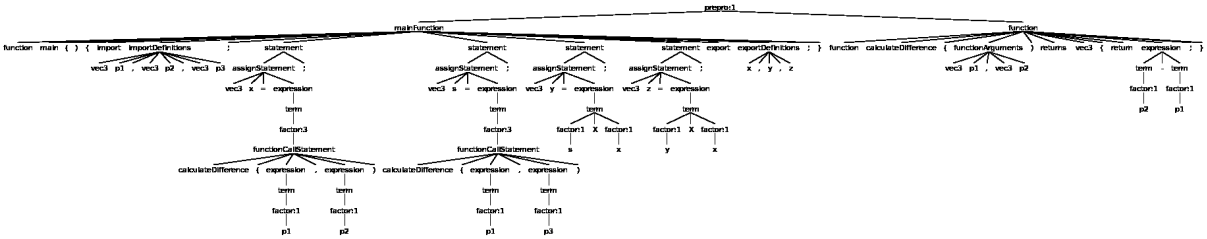

Abbildung 5.2: Beispielhafter AST

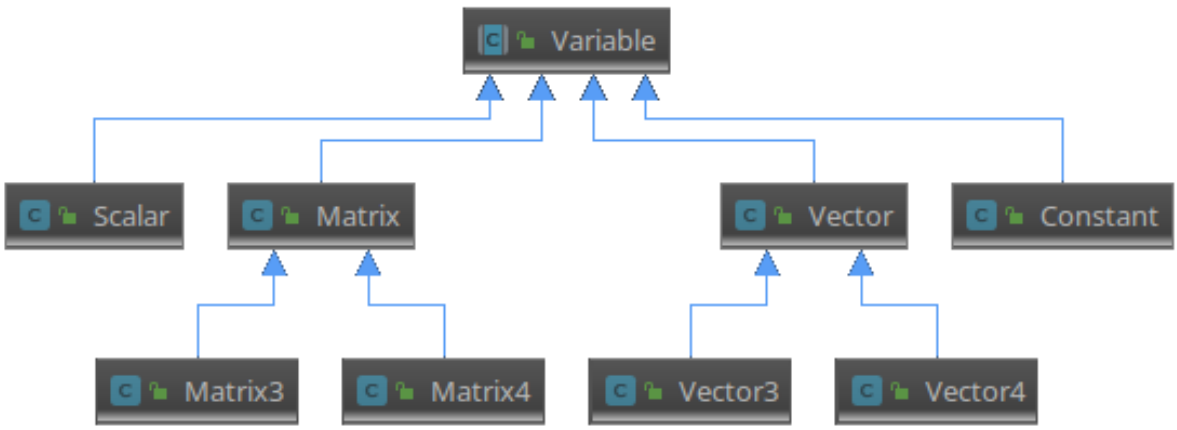

Abbildung 5.3: Typsystem von PrePro

abstrakten Klasse definierte - Funktion verwendet, und daraufhin eine Exception geworfen, dass die mathematische Operation nicht definiert sei.

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 FunctionTable

5.5 SymbolTable

5.6 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. Sie Operationen sind elementweise und werden direkt von ND4J zur Verfügung gestellt und können aufgerufen werden.

5.7 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

5.7.1 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) * drei 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 drei] konvertiert und durch eine Vector3-Wrapper-Klasse als Vector mit 3 Werten gekennzeichnet. Der Algorithmus ist in Codefragment 5.1 auf der nächsten Seite 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 5.1: Implementierung Kreuzprodukt mittels for-Schleife

5.7.2 Implementierung Kreuzprodukt mittels Subarrays

```
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 5.2: Implementierung Kreuzprodukt mittels for-Schleife

5.8 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 kleinen 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 in ms der verschiedenen Implementierungsmöglichkeiten für das Kreuzprodukt. Die Messung ist nach 5 Durchläufen gemessen, die Zahl in Klammern gibt die Zeit des ersten Durchlaufs an.

3 Varianten

Benchmarks! Klassifizierung CPU oder GPU-Workload.

Möglicherweise lohnt sich die eher GPU-betonte Variante erst ab gewisser Größe. => Dann mit konstantem Aufwand entscheiden, welches Verfahren. Vom Nutzer (während Laufzeit) auswählbar?

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 gab 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 [3] 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 stammt, ist es 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

TODO

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 [3] 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 [3] angelehnt.

6.3 Debugging

Glossar

Inhalt

Daten, welche auf den Portal-Homepages[2] angezeigt werden, z.B. Lottodaten, Wetterdaten, Bundesliga-Liveticker und das Horoskop.

Portal-Homepage

Die Startseite einer der Portale web.de, gmx.net, gmx.ch, gmx.at und home.1und1.de.

[2]Die Startseite einer der Portale web.de, gmx.net, gmx.ch, gmx.at und home.1und1.de

Abkürzungsverzeichnis

AST Abstract Sytax Tree 13

CPU Central processing unit 4

DSL Domain-specific language 4

GPU Graphics processing unit 4

JVM Java virtual machine 6

Abbildungsverzeichnis

Literatur

[1] Joe Areeda. *Antlr Notes*. 2009. URL: http://wiki.netbeans.org/Antlr_Notes [besucht am 07. 03. 2019] [siehe S. 21].

[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. 11].

[3] unbekannt. *New Language Support Tutorial Antlr*. 2009. URL: http://wiki.netbeans.org/New_Language_Support_Tutorial_Antlr [besucht am 07. 03. 2019] [siehe S. 21, 24, 25].