Universität Hamburg
Fakultät für Mathematik,
Informatik und Naturwissenschaften
Fachbereich Informatik

Bachelorarbeit

Anschauliche Visualisierung von Ausdrucksauswertung in BlueJ

2. Dezember 2013

Manuela Beckert

9beckert@informatik.uni-hamburg.de
Studiengang Computing in Science
Matr.-Nr. 6140959
Fachsemester 8

Erstgutachter Universität Hamburg:	Dr. Axel Schmolitzky
Zweitgutachter Universität Hamburg:	Prof. Dr.-Ing. Wolfgang Menzel

Inhaltsverzeichnis

1 Einleitung

Das Ziel dieser Bachelorarbeit ist es, die Auswertung von Java-Ausdrücken in der Entwicklungsumgebung BlueJ anschaulich zu visualisieren. Der Fokus liegt dabei auf Ausdrücken, die während der ersten vier Wochen im Modul „Softwareentwicklung 1“ an der Universität Hamburg eingeführt werden. Die Auswertungsreihenfolge soll verdeutlicht und gegebenenfalls anhand zusätzlicher Hinweise verständlich gemacht werden. Hierfür wird der Ausdruck mithilfe eines Parsers, der mit dem ANTLR-Parsergenerator erstellt wird, zunächst in eine Baumdarstellung überführt. Auf dieser Baumdarstellung wird die eigentliche Auswertung durchgeführt und das Ergebnis mit dem JUNG-Graph-Framework visualisiert. Das Tool ist in die Direkteingabe der Entwicklungsumgebung BlueJ eingebunden.

1.1 Motivation

In dem Modul „Softwareentwicklung 1“ wird die Programmiersprache Java nach dem Objects-First-Ansatz [KB13] gelehrt. In den ersten vier Wochen müssen von den Studenten viele Grundlagen schnell verstanden werden. Die Entwicklungsumgebung BlueJ bietet bereits viel Funktionalität zur Visualisierung von Klassen und Objekten und Interaktion mit diesen. In Bezug auf die Auswertung von Java-Ausdrücken besteht weniger Unterstützung. Es existiert eine Art Java-Konsole, in der interaktiv Ausdrücke eingegeben und ausgewertet werden können. Doch zum Verständnis, warum das berechnete Ergebnis herauskommt, gehört oft mehr Hintergrundwissen: dass zu Beginn eine statische Typprüfung stattfindet, welche Operatoren stärker binden, warum oft nicht alle Teile eines boolschen Ausdrucks ausgewertet werden müssen oder welche Auswirkungen die Ungenauigkeit von Gleitkommazahlen hat. All diese Zusammenhänge werden bisher wenig veranschaulicht.

1.2 Vorgehen

Zunächst wird auf den **Kontext** eingegangen, in dem das Tool später verwenden werden soll und aus diesem die **Problemstellung und Anforderungen** an das Tool abgeleitet. Nach Klärung der **Grundbegriffe**, die für das Verständnis des Entwurfs notwendig sind, werden verschiedene **Entwürfe** und ihre Vor- und Nachteile diskutiert. Anschließend

wird die **Implementierung** des favorisierten Entwurfs erläutert, sowohl in Bezug auf das entstandene Tool als auch dessen Integration in BlueJ. Zum Abschluss wird im Kapitel **Zusammenassung und Ausblick** kurz über Umsetzung reflektiert und Möglichkeiten der Erweiterung diskutiert.

1.3 CD-Beilage

In der beiliegenden CD befindet sich der Quelltext des modifierten BlueJ in zwei Varianten:

- reiner Quelltext
- Eclipse-Projekt mit Anweisungen zur Einbindung

2 Kontext

Den Hintergrund für das Thema dieser Bachelorarbeit bilden die Lehrveranstaltung „Softwareentwicklung 1“ und die darin verwendete Entwicklungsumgebung BlueJ.

2.1 Softwareentwicklung 1

„Softwareentwicklung 1“ (SE1) ist ein Modul, das am Fachbereich Informatik an der Universität Hamburg angeboten wird. Es ist ein Grundlagenmodul, das im ersten Semester aller Informatik-Studiengänge an der Universität Hamburg stattfindet, und besteht aus Vorlesung und Präsenzübung am Rechner. In dem Modul werden die Grundlagen objektorientierter Programmierung anhand der Programmiersprache Java gelehrt. Dabei wird nach dem Objects-First-Ansatz[KB13] vorgegangen und, wie der Dozent Dr. Axel Schmolitzky es ausdrückt, „Konsumieren vor Produzieren“ [SS07].

Dies ist eines der Leitmotive von SE1. Konsumieren bedeutet Lernen durch Benutzen. Analog ist Produzieren Lernen durch Erschaffen. Es einfach, etwas über die Funktion eines Fernsehers zu lernen, indem man ihn benutzt. Einen Fernseher zu bauen ist weit schwieriger, schafft jedoch ein tiefergehendes Verständnis. Analog verhält es sich mit Programmierkonzepten. Die Studenten erstellen Objekte von Klassen, lange bevor sie eine Zeile Quelltext gesehen haben, und benutzen Sammlungen (Listen, Mengen, Maps) ohne etwas von Arrays zu wissen. Sie haben damit Klassen und Sammlungen durch Benutzung kennengelernt, bevor sie selbst Klassen und Sammlungen erstellen. Dieser Ansatz ist in den Übungen umsetzbar durch die Unterstützung, die die Entwicklungsumgebung BlueJ bietet.

2.2 BlueJ

BlueJ [Köl13] ist eine Java-Entwicklungsumgebung, die speziell für die Lehre von Objektorientierung entworfen wurde. Sie ermöglicht die Lehre nach dem Objects-First-Ansatz. Dadurch, dass Klassen und Objekte interaktiv erzeugt werden können und mit ihnen interagiert werden kann, ist es möglich, Objekte ohne jedwedes Hintergrundwissen über zugrundeliegende Konzepte zu benutzen.

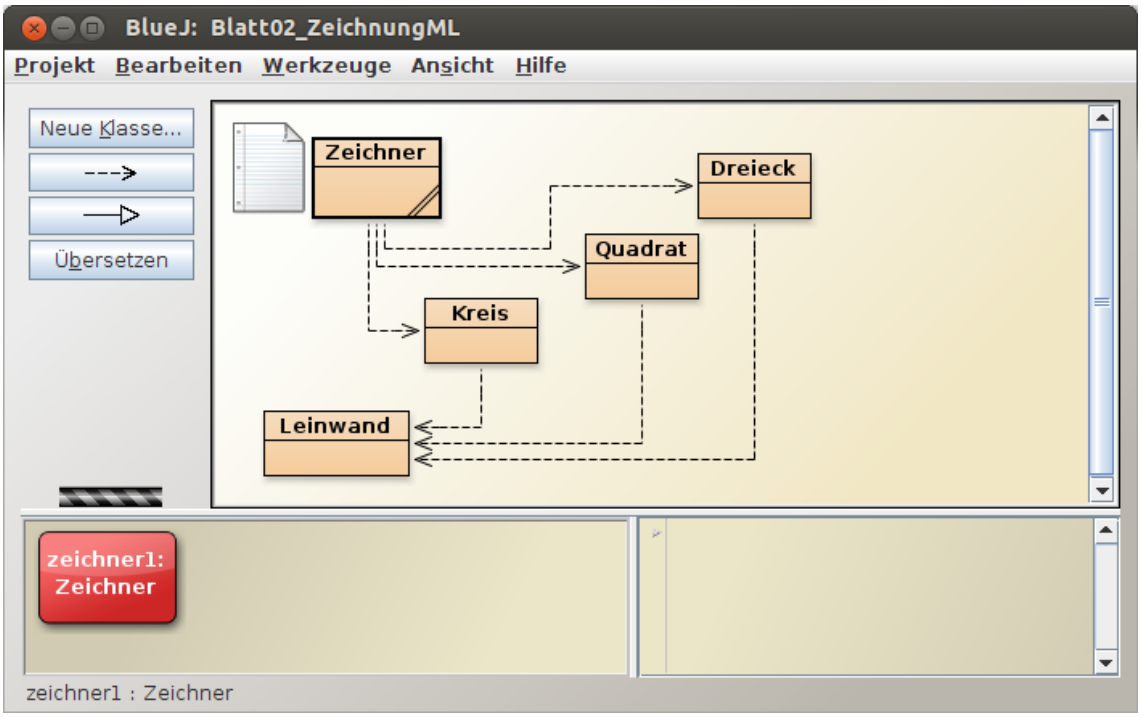


Abbildung 2.1: Benutzeroberfläche von BlueJ [Pot12, Abb. 3.1]

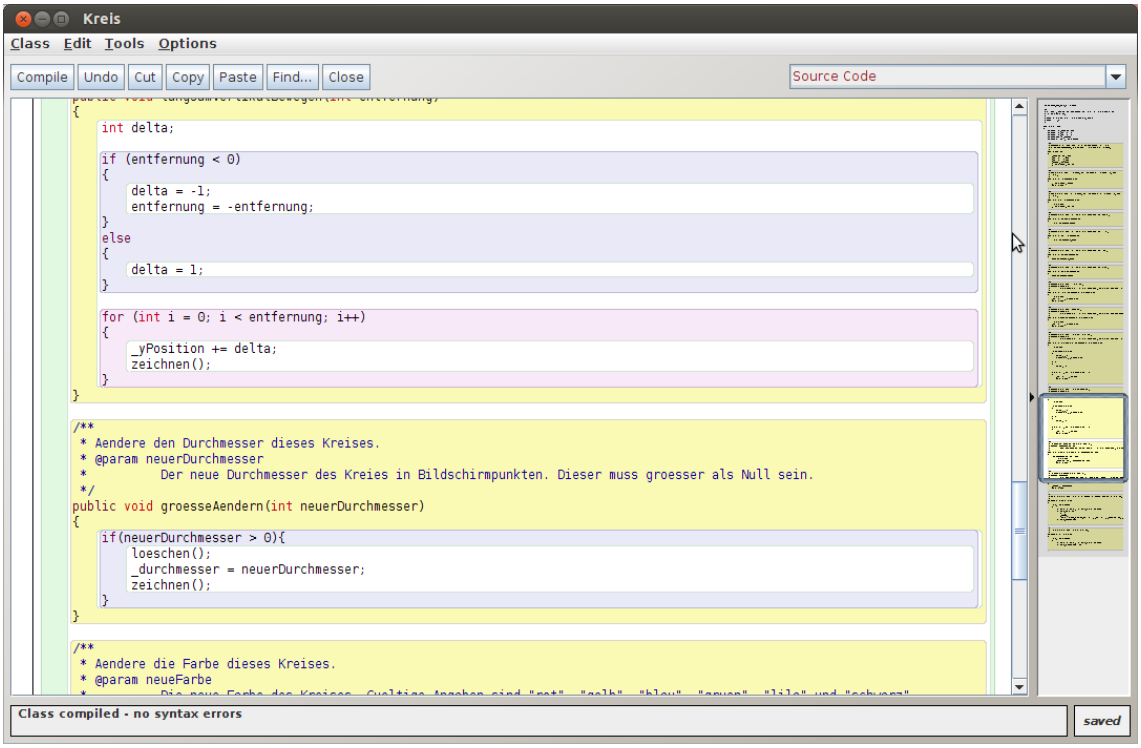


Abbildung 2.2: Quelltextansicht in BlueJ [Pot12, Abb. 3.2]

Ein Screenshot der Benutzeroberfläche von BlueJ 3.0.7 ist in Abbbildung 2.1 zu sehen. Im Hauptfenster werden die Klassen und Interfaces des aktuell geladenen Projekts in einem vereinfachten Klassendiagramm dargestellt. Objekte von diesen Klassen können durch Rechtsklick auf die Klasse aus dem Kontextmenü erzeugt werden und werden in dem Fenster unten links angezeigt. An ihnen können, wiederum aus dem Kontextmenü, beliebige Methoden aufgerufen werden. Ein Doppelklick auf ein Objekt öffnet den so genannten Objektinspektor, der den internen Zustand des Objekts, also alle Zustandsfelder und deren Belegung, anzeigt.

Unten rechts lässt sich die so genannte *Direkteingabe* einblenden, in der interaktiv Java-Code ausgewertet werden kann. Die meisten Anweisungen, die innerhalb einer Methode stehen könnten, können auch in die Direkteingabe eingegeben werden, inklusive Schleifen und Methodenaufrufen. Anweisungen, die nicht mit einem Semikolon abgeschlossen werden, werden von der Direkteingabe als auszuwertender Ausdruck angesehen und zu einem Ergebnis ausgewertet. Zu jedem Ergebnis wird auch der Typ mit angezeigt. Ein Beispiel für eine solche Auswertung ist in Abbildung 2.3 dargestellt.

```
> int i = 42;
> i
  42 (int)
> for(int j=0; j<i; j++) i--;
> i
  21 (int)
```

Abbildung 2.3: Beispiel-Auswertung in der Direkteingabe

Zur Quelltextansicht (Abb. 2.2) gelangt man durch Doppelklick auf eine Klasse. Der Editor in BlueJ bietet zwei verschiedene Ansichten: den Quelltext und die Javadoc-Dokumentation (in SE1 auch Schnittstellenansicht genannt). Die Quelltextansicht dient dem Bearbeiten der Klassen. Die verschiedenen Sichtbarkeitsebenen werden farblich hervorgehoben. Außerdem wird rechts eine Vorschau des Quelltexts zur schnelleren Navigation angezeigt.

BlueJ besitzt einen integrierten Debugger. Wird im Quelltext am linken Rand durch Doppelklick ein Haltepunkt gesetzt, so öffnet der Debugger sich automatisch beim nächsten Aufruf der betroffenen Methode. Der Debugger kann auch direkt aus dem Menü im Hauptfenster geöffnet werden. Er besitzt eine eigene grafische Oberfläche (Abb. 2.4). Hier werden alle aktuellen Klassen und aufgerufenen Methoden sowie alle sichtbaren Variablen an der betrachteten Programmzeile angezeigt und Möglichkeiten der Navigation durch den Programmablauf angeboten.

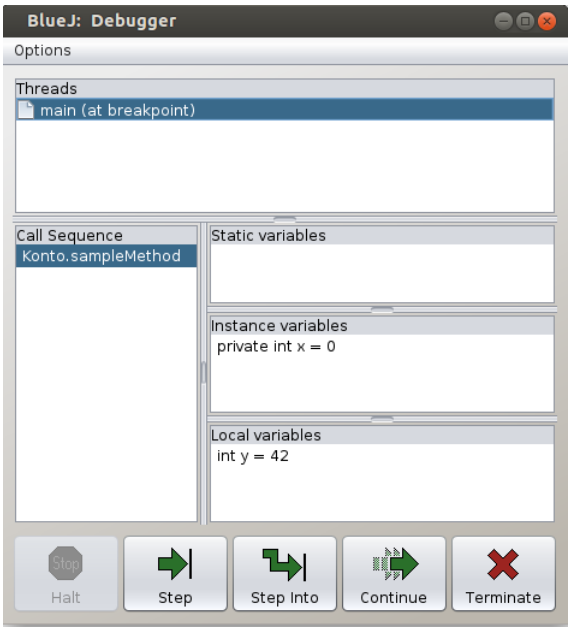


Abbildung 2.4: Debugger in BlueJ

2.3 Ausdrücke in SE1

Das SE1-Modul gliedert sich in vier Komplexitätsstufen, auch *Level* genannt [SZ+12]. In Level 1 werden beispielsweise Klassen und Objekte eingeführt und in Level 2 darauf aufbauend Klassen als Typ und Referenzen. Dabei werden in jeder Stufe bereits Inhalte aus höheren Stufen konsumiert. Jede Vorlesung und Übung ist eines dieser Level zugeordnet.

Für diese Arbeit relevant ist Level 1, das die ersten vier Wochen der Lehre umfasst. In diesen Wochen müssen von den Studenten sehr viele grundlegende Konzepte, sowohl von der Objektorientierung als auch der Programmiersprache, verstanden werden. Um Klassen und Objekte zu verstehen und zu benutzen, sind bestimmte Grundlagen nötig. Eine dieser Grundlagen ist die Ausdrucksauswertung. Als *Ausdruck* gilt alles, was zu einem Ergebnis ausgewertet werden kann, wie zum Beispiel eine Addition oder ein Methodenaufruf. Ausdrücke sind Bestandteil beinahe jeder Java-Anweisung. Die für Level 1 relevanten Ausdrücke sind in Tabelle 2.1 aufgelistet.

Neben arithmetischen Operatoren und Vergleichen gehören auch Methodenaufrufe, Typumwandlungen und Zuweisungen dazu. Die logischen Operatoren beschränken sich auf „Und“, „Oder“ und „Nicht“. Es können sowohl primitive Typen als auch Refenztypen, die bereits in Level 1 konsumiert werden, und Variablen von diesen Typen vorkommen.

Ausdrucksart	**Beispiele**
arithmetisch	+, -, *, /, %, ++, --
logisch	&&, ||, !
Vorzeichen	+, -
Vergleiche	==, !=, <, >, <= >=
Typumwandlung	explizit, implizit
Feld-/Methodenzugriff	o.e, o.m(), o.m(int, String),...
Zuweisung	=

Tabelle 2.1: Liste der für Level 1 relevanten Ausdrücke

3 Problemstellung und Anforderungen

In diesem Kapitel wird verdeutlicht, warum gerade das Verständnis der Ausdrucksauswertung ein Problem darstellt und welche Anforderungen an ein Tool, das diese in der Lehre anschaulich visualisieren soll, gestellt werden müssen.

3.1 Problemstellung

Eine der Grundlagen für das Lernen von Programmiersprachen ist ein Verständnis der Bedeutung von Ausdrücken.

Die Programmiersprache ist gerade in den ersten Wochen des Studiums etwas, das schwer zu erfassen ist. Mit wenig Code können bereits erstaunliche Ergebnisse erreicht werden. Und während intuitiv meist klar ist, dass bei `5+3` zwei Zahlen addiert werden, ist es doch ein großer Schritt hin zu dem Verständnis, dass das Ergebnis der Operation wie eine gewöhnliche Zahl weitergenutzt werden kann - und dass es sogar verworfen wird, falls man dies nicht tut. Auch ein Methodenaufruf führt Operationen aus, die an einer gänzlich anderen Stelle festgelegt sind, da liegt der Gedanke nicht fern, dass auch der Rückgabewert automatisch irgendwo gespeichert wird und man nur den richtigen Befehl kennen muss, um später darauf zuzugreifen. Besonders boolsche Ausdrücke sind in den Augen von Studienanfängern etwas höchst abstraktes und kompliziertes. Mit etwas wie „wahr“ und „falsch“ zu rechnen ist mindestens genauso anspruchsvoll wie die Einsicht, dass die Wirkung von

```
if(variable == true)
{
    return true;
}
else
{
  return false;
}
```

die gleiche ist, als hätte man einfach

```
return variable;
```

geschrieben. Gleitkommazahlen verhalten sich nicht einmal mehr so, wie man es von reellen Zahlen erwarten würde: `123456789f + 1` ergibt in Java `123456792f`[1].

Um all diese Konzepte zu verstehen, stehen bisher nur die Betreuer in den Präsenzübungen zur Verfügung. Doch auch Betreuer haben nur begrenzt Zeit für jeden Studenten und nicht jede Erklärung schafft es ins Langzeitgedächtnis. Manchmal werden selbst im an SE1 anschließenden Modul „Softwareentwicklung 2“ noch Wissenslücken entdeckt. Es fehlt etwas, das es den Studierenden ermöglicht, selbst herauszufinden, warum ein Ausdruck das beobachtete Ergebnis liefert. Etwas, das ihnen zeitnah Feedback gibt, an dem sie experimentieren können. Ein solches Tool, das die Ausdrucksauswertung visualisiert, soll im Rahmen dieser Arbeit entstehen.

3.2 Anforderungen

Die größte Anforderung ist natürlich das Lösen der Problemstellung: die Visualisierung der Ausdrucksauswertung. Eine Darstellung des Abstrakten Syntaxbaumes ist beispielsweise bereits sehr hilfreich, um die Reihenfolge der Auswertung zu verstehen, doch sie gibt wenig Auskunft über Typen, Typumwandlung oder die Seiteneffekte von Operationen. Das Tool, das in dieser Arbeit entstehen soll, muss also in der Lage sein, mehrere verschiedene Lehrinhalte und Programmierkonzepte zu vermitteln:

- die Bedeutung von eventuell für Studierende unbekannte Operatoren, wie Modulo oder logische Operatoren
- die Auswertungsreihenfolge
- die Typen aller Teilausdrücke, statische Typprüfung und Typumwandlungen
- was bei einem Auswertungsschritt passiert
- Seiteneffekte auf Variablen
- Besonderheiten, wie Gleitkommaproblematik, Referenzen, Auswertungsstopp bei boolschen Ausdrücken
- den Unterschied zwischen Übersetzungszeit und Laufzeit

[1] Das `f` indiziert, dass es sich um eine Float-Zahl handelt. Float-Zahlen werden intern als Zweierpotenzen mit begrenzter Genauigkeit gespeichert. Nicht jede Zahl lässt sich mit der Genauigkeit von Float direkt als Zweierpotenz darstellen, so auch 123456789. Solche Zahlen werden an die nächstgelegene darstellbare Zahl angenähert, in diesem Fall 123456792. Addiert man darauf 1, erhält man wieder eine nicht darstellbare Zahl, die auf 123456792 angenähert wird.

Dabei sollte die Benutzungsoberfläche trotzdem einfach gehalten und intuitiv zu bedienen sein.

Das Tool sollte robust sein, in dem Sinne, dass es auch auf nicht auswertbare Eingaben korrekt reagiert. Es sollte keine Eingabe geben, die BlueJ zum Absturz bringt. Außerdem sollte das Tool in der Lage sein, hilfreiches Feedback zu geben, warum ein Ausdruck nicht auswertbar ist, zum Beispiel wegen nicht kompatibler Typen, unbekannten Variablenbezeichnern oder dem Tool unbekannten Java-Konstrukten.

Damit das Tool allen Übungsteilnehmern zur Verfügung gestellt werden kann, muss es als BlueJ-Addon bereitgestellt werden. Es sollte, wie BlueJ, im Internet frei verfügbar sein und eine Installationsanleitung haben. Da die vollständige Implementation im Rahmen einer Bachelorarbeit nicht möglich ist, sollte der Quelltext leicht verständlich und erweiterbar sein, indem bekannte Entwurfsmuster genutzt werden und der Quelltext gut kommentiert ist.

4 Grundlagen: Parsing von Ausdrücken

Um die Lösungsidee nachvollziehen zu können, ist ein grundlegendes Verständnis von Parsing notwendig.

> *Parsing is the process of structuring a linear representation in accordance with a given grammar.*
>
> - Dick Grune, Ceriel Jacobs [GJ90, S.13]

In dieser sehr abstrakten Definition von Grune und Jacobs kann die „lineare Repräsentation“ ein Satz sein, ein Computerprogramm oder sogar ein Stück Musik. Wichtig ist, dass es eine gewisse Ordnung gibt, eine Hierarchie. Ein (objektorientiertes) Computerprogramm besteht gewöhnlich aus Klassen, Klassen bestehen aus Methoden, der Rumpf jeder Methode ist gefüllt mit Anweisungen.

Eine „Grammatik“ ist, im normalen Sprachgebrauch wie auch in der Informatik, etwas, das der Beschreibung einer Sprache dient.

4.1 Sprache

Eine (Programmier-)Sprache kann schlicht als Menge von Sätzen betrachtet werden. Sätze bestehen aus Wörtern. Wörter sind wiederum Sequenzen von Buchstaben beziehungsweise *Symbolen* aus einem bestimmten *Alphabet*. Jedes Wort hält eine gewisse Bedeutung und trägt zur Bedeutung des ganzen Satzes bei.

Man unterscheidet zwischen der Snytax und der Semantik der Sprache. Die *Syntax* umfasst die Gestalt und den Aufbau des Programms und seiner Bestandteile. Sie ist die Vorschrift dafür, wie eine Anweisung aufzubauen ist, damit sie einen gültigen Befehl ergibt. Ein Beispiel hierfür sind arithmetische Operatoren, für die es zwei Operanden geben muss, üblicherweise Zahlen. Aber auch der Aufbau einer Methode oder Klasse ist Teil der Syntax. Die *Semantik* hingegen ist die Bedeutung der Ausdrücke und Programmteile und legt fest, was bei der Ausführung des Programms geschehen soll [Neb12, S.87]. Die Semantik des Operators „*“ ist beispielweise, dass die beiden Operanden multipliziert werden und, falls es noch weitere Operatoren im Ausdruck gibt, wie die Auswertungsreihenfolge ist. Man kann sagen, dass die syntaktische Struktur die semantische Auswertung steuert.

4.2 Grammatik

Die Syntax einer Programmiersprache kann auf zwei Wegen definiert werden: durch Erkennen oder Erstellen [Seb12, S.116]. Möchte man eine Programmiersprache durch Erkennen definieren, benötigt man einen Mechanismus um festzustellen, ob ein gegebenes Wort Teil der Sprache ist oder nicht. Ein solcher Mechanismus wird *Recognizer* genannt. Mittels eines Recognizers kann man ein gegebenes Wort, einen Satz oder ein Stück Code auf syntaktische Korrektheit überprüfen. Es ist nicht möglich, neue Sätze der Sprache zu erstellen.

Ein *Sprachgenerator* ist eine Möglichkeit zur Definition durch Erstellen. Er ist in der Lage, alle Wörter der Sprache zu erzeugen. Ein Wort ist also genau dann in der Sprache, wenn es vom Sprachgenerator erzeugt werden kann. Ein Sprachgenerator ist meist deutlich einfacher zu schreiben und zu verstehen als ein Recognizer. Oft lässt sich bereits an der Struktur des Generators erkennen, ob ein Wort in der Sprache enthalten ist. Es werden deshalb häufig Spreachgeneratoren zur Beschreubung einer Sprache genutzt,

Moderne Programmiersprachen nutzen als Sprachgenerator eine *kontextfreie Grammatik*:

> *[...] eine solche Grammatik ist zu sehen als eine Menge von Umformungsregeln, um eine Sprache zu erzeugen.*
>
> - Katrin Erk, Lutz Priese [EP08, S.54]

Den Anfang bildet dabei immer ein Startsymbol. Das *Startsymbol* ist eine Variable, die meist mit S bezeichnet wird. Nach den Regeln der Grammatik wird das Startsymbol durch ein Wort ersetzt. Dieses Wort kann wiederum Variablen (auch: *Nicht-Terminale*) enthalten, die dann weiter ersetzt werden, oder *Terminale*. Terminale sind Symbole aus dem Alphabet oder der Sprache und werden nicht weiter ersetzt. Die Umformung endet, wenn in dem Wort nur noch Terminale enthalten sind.

Eine Regel in der Grammatik beschreibt immer einen Umformungsschritt. Dabei wird das, was auf der „linken“ Seite steht, durch das auf der „rechten“ ersetzt. Dieser Prozess heißt *Ableitung*. Für die Ersetzung kann es mehrere Auswahlmöglichkeiten geben. Wenn wir das (sehr einfache) Beispiel

$$S \to HN$$
$$S \to H$$

betrachten, so kann S entweder durch HN oder durch H ersetzt werden.

Die Sprache, die von einer Grammatik erzeugt wird, ist somit die Menge aller terminalen

Wörter, die durch die Umformungsregeln aus dem Startsymbol erzeugt werden können [EP08, S.54].

Eine Grammatik heißt *kontextfrei*, wenn in jedem Schritt ein einzelnes Nicht-Terminal abgeleitet wird. Das bedeutet insbesondere, dass nur das Nicht-Terminal allein auf der Linken Seite der Ableitung stehen darf. Es wird also unabhängig von dem Kontext, in dem es auftritt, abgeleitet. Kontextfreie Grammatiken können auch in der *Erweiterten Backus-Naur-Form* (kurz EBNF, siehe [ISO96]) dargestellt werden. Der Unterschied zu der Notation kontextfreier Grammatiken ist in Abb. 4.1 zu sehen.

```
Expr -> Expr + Expr
Expr -> Expr * Expr
Expr -> Expr - Expr
Expr -> Expr / Expr
Expr -> ( Expr )
Expr -> Number
Number -> 1
Number -> 2
Number -> 3
```

```
Expr =  Number
        | Expr {Opr Expr}
        | '(' Expr ')';

Opr =  '+'
        | '*'
        | '-'
        | '/';

Number = '1' | '2' | '3' ;
```

Abbildung 4.1: Beispiel einer kontextfreien Grammatik (links) im Vergleich zur entsprechenden EBNF (rechts). Diese Grammatiken definieren Sätze der Form (1+2)*3. [Pot12, Abb. 2.1]

Während es in kontextfreien Grammatiken für jede Mögliche Ableitung eine eigene Regel gibt, sind in der EBNF alle Ableitungen für ein Nichtterminal in einer einzigen Regel zusammengefasst[2].

Die Ableitungsschritte lassen sich in einem *Syntaxbaum* (auch: Ableitungsbaum, Parsebaum) grafisch darstellen. Jeder Ableitungsschritt wird hierbei von einem Knoten repräsentiert. Der Syntaxbaum für die Ableitung des Satzes `(1+2)*3` aus der Grammatik in Abb. 4.1 ist in Abb. 4.2 zu sehen.

Der abgeleitete Satz ergibt sich durch Lesen der Terminale von links nach rechts. Dabei kann die Ableitung in unterschiedlicher Reihenfolge stattfinden. So könnte die Ersetzung von links nach rechts stattfinden (ähnlich einer Tiefensuche) oder zuallererst das Multiplikationszeichen abgeleitet werden. Man erhält jedoch unabhängig von der Reihenfolge der Ableitung immer den gleichen Satz.

[2] Es sei anzumerken, dass die Grammatiken nicht vollständig äquivalent sind. Sie erzeugen zwar die gleichen Ausdrücke, aber unterschiedliche Strukturen. In der kontextfreien Grammatik sind für den Ausdruck `a + b + c` beispielsweise zwei verschiedene Ableitungen (und damit Syntaxbäume) möglich, je nachdem, obzunächst nach `a + Expr` abgeleitet wird oder zu `Expr + c`. In der EBNF gibt es hingegen nur eine Ableitung.

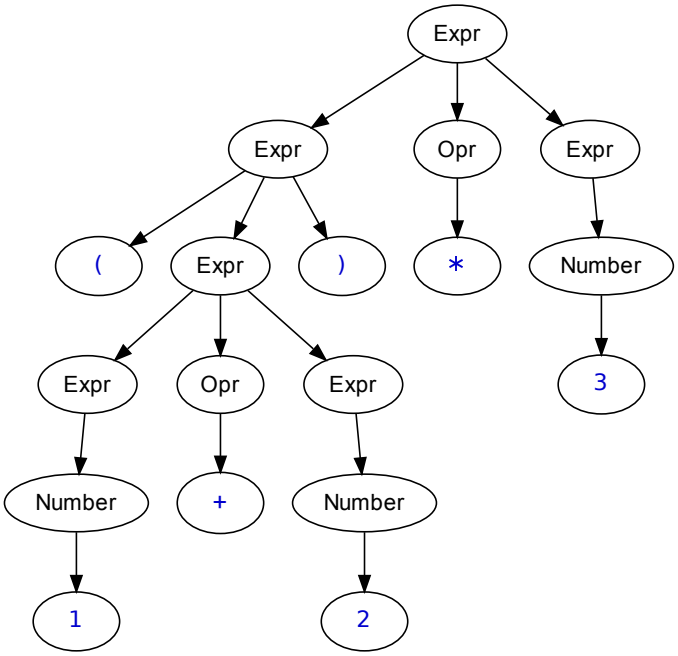


Abbildung 4.2: Syntaxbaum für den Satz `(1+2)*3`, abgeleitet aus der Grammatik in Abb. 4.1. Die Terminale sind blau hervorgehoben.

4.3 Der Parser

> *To parse a string according to a grammar means to reconstruct the production tree that indicates how the given string can be produced from the given grammar.*
>
> - Dick Grune, Ceriel Jacobs [GJ90, S.62]

Das Parsing einer Zeichenkette ist einfach ausgedrückt der Umkehrschritt zum Ableiten. Statt des Anwendens von Umformungsschritten zum Erzeugen eines Satzes werden anhand eines gegebenen Satzes die Umformungsschritte - und damit der Syntaxbaum - rekonstruiert. Ein Parser ermittelt die syntaktsiche Struktur eines Satzes und kann damit bereits erste Hinweise zur Semantik liefern, wie die Operatorpräzedenz. Daher ist er wichtiger Bestandteil von Übersetzungen zwischen Programmiersprachen.

Hochsprachen wie Java und C sind komfortabel zu benutzen, unsere Computer sprechen jedoch Maschinensprache. Die Maschinensprache ist die Menge an Befehlen, die von einem Prozessor direkt ausgeführt werden kann [Seb12, S.24]. Diese Befehle sind sehr primitive Anweisungen, wie das Verschieben des Programmpointers, Laden oder Speichern von Werten in Registern oder einfache arithmetischen Operationen. Bevor ein Java- oder C-Programm ausgeführt werden kann, muss es zunächst in Maschinensprache übersetzt werden. Ein solches Programm, das zwischen zwei Programmiersprachen übersetzt, heißt *Compiler* [ALSU08, S.3].

4.4 Compiler-Aufbau

Der Übersetzungsvorgang eines Compilers lässt sich unterteilen in Analye und Synthese. In der Analyse-Phase wird der Quelltext in seine Bestandteile zerlegt und aus ihnen der Syntaxbaum aufgebaut. Durch das Erstellen des Baumes geschieht bereits die Syntaxprüfung. Am Syntaxbaum wird anschließend auch die Semantik kontrolliert. Aus dem Baum entsteht dann eine Zwischendarstellung des Programms, die der Maschinensprache bereits näher ist. Bei der Synthese wird aus der Zwischendarstellung schließlich das Zielprogramm konstruiert.

Die Analyse besteht wiederum aus vier Phasen (Abb. 4.3). Jede wandelt die Darstellung des Quellcodes in eine andere Darstellung um. Es gibt außerdem eine Symboltabelle, die von allen Phasen gemeinsam genutzt wird und in der zusätzliche Informationen gespeichert werden. Für diese Arbeit sind die ersten drei Phasen relevant: Lexikalische Analyse, Syntaxanalyse und Semantische Analyse.

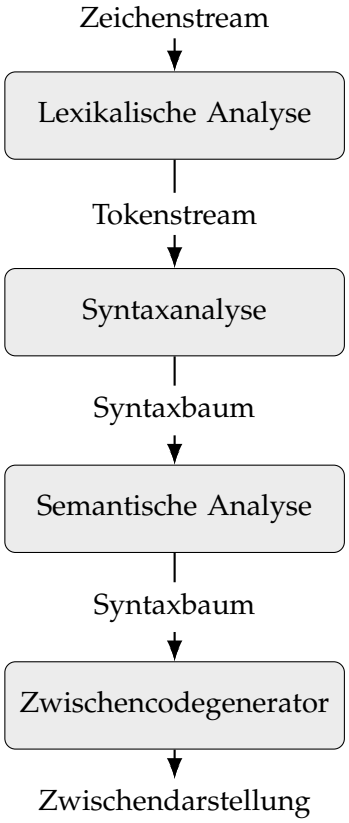


Abbildung 4.3: Analysephase eines Compilers, nach [ALSU08, Abb. 1.6]

In der *Lexikalischen Analyse* wird der Zeichenstream des Quellprogramms gelesen und in bedeutungsvolle Sequenzen, die so genannten Lexeme zerlegt. *Lexeme* sind Terminale der Grammatik und können reservierte Wörter wie `if` und `else` sein, Variablenbezeichner wie `pi` oder Zahlen wie `3.14159`. Zu jedem Lexem wird ein Token der Form

$$< Tokenname, Attributwert >$$

ausgegeben. Der Tokenname ist ein abstraktes Symbol, zum Beispiel `if` und `else`, `id` für Variablen oder `number` für Zahlen. Der Attributwert zeigt auf einen Eintrag in der Symboltabelle, in dem zusätzliche Informationen enthalten sind, wie der Name des Variablenbezeichners oder der Wert einer Zahl. Die Tokens werden dann an die Syntaxanalyse weitergegeben. In der Lexikalischen Analyse können auch bereits Leerzeichen oder Kommentare entfernt werden.

Die *Syntaxanalyse* wird auch *Parsing* genannt. Hier wird festgestellt, wie ein Terminal von der Grammatik generiert werden kann. Dieser Schritt ist vergleichbar mit der Erstellung eines Syntaxbaumes. Es gibt zwei große Klassen von Syntaxanalyse-Methoden:

Top-Down und *Bottom-Up*. Wie die Namen suggerieren, beginnt die Top-Down-Analyse bei der Wurzel und arbeitet sich zu den Blättern vor, während Bottom-Up-Analyse in umgekehrter Reihenfolge erfolgt. Mit der Bottom-Up-Methode lassen sich zwar umfangreichere Klassen von Grammatiken verarbeiten, Top-Down-Parser sind jedoch einfacher per Hand zu schreiben und daher beliebter [ALSU08, S.76].

Ausgehend vom Startsymbol werden bei der Top-Down-Analyse Linksableitungen, von der Reihenfolge her analog einer Tiefensuche, vorgenommen. Es entsteht eine baumartige Zwischendarstellung, der *Abstrakte Syntaxbaum* [ALSU08, S. 87]. Im Unterschied zum Syntaxbaum, der die Reihenfolge der Ableitungen visualisiert, wird in einem Abstrakten Syntaxbaum die grammatische Struktur des Tokenstreams gezeigt. Jeder innere Knoten steht für eine Operation und seine Kindknoten für die Operanden. Der Abstrakte Syntaxbaum des Ausdrucks `(1+2)*3` ist in Abbildung 4.4 dargestellt. Er zeigt die Reihenfolge, in der die einzelnen Operationen ausgeführt werden.

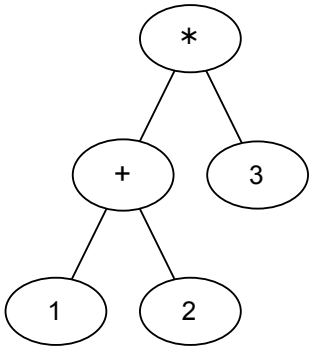


Abbildung 4.4: Abstrakter Syntaxbaum für den Ausdruck `(1+2)*3`

In den anschließenden Compiler-Phasen wird diese Darstellung genutzt, um das Quellprogramm zu analysieren und das Zielprogramm zu erzeugen.

Bei der *Semantischen Analyse* wird mithilfe des Syntaxbaumes und der Symboltabelle das Quellprogramm auf semantische Konsistenz überprüft. So findet beispielsweise eine *Typprüfung* statt, um sicherzustellen, dass jeder Operator die richtigen Operanden hat und nicht etwa zwei Wahrheitswerte miteinander addiert werden oder ein String als Arrayindex verwendet wird. In der Sprachspezifikation können einige Typkonvertierungen erlaubt sein. So kann der Compiler, falls kompatible Typen wie eine Integer- und eine Fließkommazahl addiert werden sollen, die Ganzzahl in eine Fließkommazahl umwandeln. Die Typumwandlung wird dann als eigener Knoten in den Baum eingefügt.

4.5 Parsergeneratoren

Nicht jeder Parser muss per Hand geschrieben werden. Es gibt Parsergeneratoren, die aus einer gegebenen Grammatik einen Parser generieren, üblicherweise in Form von Quelltext. Die Grammatik muss dabei in einer speziellen Form der Backus-Naur-Form gegeben sein, die sich jedoch von Generator zu Generator unterscheidet. Für jede Änderung des Parsers muss zunächst die Grammatik geändert und dann der Parser neu generiert werden. Der Vorteil an Parsergeneratoren ist, dass die zugrundeliegende Grammatik einfacher verständlich ist, da sie näher an der ursprünglichen Syntaxdefinition der Sprache ist als ein komplexer Parser. *Yacc* („Yet another compiler-compiler“) ist ein Beispiel für einen C-Parsergenerator, ANTLR („ANother Tool for Language Recognition“) für Java.

5 Entwurf der Visualisierung

Es gibt viele verschiedene Aspekte der Ausdrucksauswertung in Java, wie Operatorpräzedenz, allgemeine Auswertungsreihenfolge, Typen der Teilausdrücke, Typumwandlungen oder die Belegung von Variablen. Das Ziel ist es, all diese Aspekte so zu visualisieren, dass ein größtmögliches Verständnis der Ausdrucksauswertung erreicht wird.

5.1 Möglichkeiten zur Visualisierung

5.1.1 Visualsierung direkt am Ausdruck

Betrachten wir beispielsweise den folgenden Ausdruck:

$$1 + 2 * \texttt{pi} < \texttt{addierer.addiere}(5, 3 * 4)$$

wobei `pi` und `addierer` Variablenbezeichner sind.

Ein allererster Schritt könnte sein, den Ausdruck so zu klammern, dass die Operatorpräzedenz (nach der Java Language Secification [GJS+13]) sichtbar wird.

$$(1 + (2 * \texttt{pi})) < (\texttt{addierer.addiere}(5, 3 * 4))$$

Nun ist bereits klar, dass die Multiplikation vor der Addition stattfindet und der Vergleich ganz zuletzt. Diese Darstellung verrät jedoch noch nichts über die Typen der Teilausdrücke oder des Gesamtausdrucks und wie diese Typen zustande kommen. Genauso wenig erklärt sie die Auswertungsreihenfolge bei dem Methodenaufruf. Nun könnten die Typen der Operanden einfach am Ausdruck notiert werden.

$$(\underset{\texttt{int}}{1} + (\underset{\texttt{int}}{2} * \underset{\texttt{float}}{\texttt{pi}})) < (\texttt{addierer.addiere}(\underset{\texttt{int}}{5}, \underset{\texttt{int}}{3} * \underset{\texttt{int}}{4}))$$

Doch wie veranschaulicht man die Typen der restlichen Teilausdrücke? Natürlich ist es möglich, auch an jeden Operator den Ergebnistyp zu schreiben, wie in folgendem Beispiel dargestellt.

$$(\underset{\texttt{int}}{1} \underset{\texttt{float}}{+} (\underset{\texttt{int}}{2} \underset{\texttt{float}}{*} \underset{\texttt{float}}{\texttt{pi}})) \underset{\texttt{boolean}}{<} (\texttt{addierer.addiere}(\underset{\texttt{int}}{5}, \underset{\texttt{int}}{3} \underset{\texttt{int}}{*} \underset{\texttt{int}}{4}))$$

Aber dabei wird der Ausdruck eher unübersichtlich, was dazu führt, dass das Verständnis mehr erschwert als erleichtert wird. Die alleinige Visualisierung in Textform erweist sich also als schwierig.

5.1.2 Syntaxbaum

Die Operatorpräzedenz lässt sich sehr leicht und übersichtlich in einem Syntaxbaum visualisieren (Abb. 5.1). Nimmt man zusätzlich die Leserichtung von links nach rechts an, ergibt sich auch die Auswertungsreihenfolge direkt aus dem Baum.

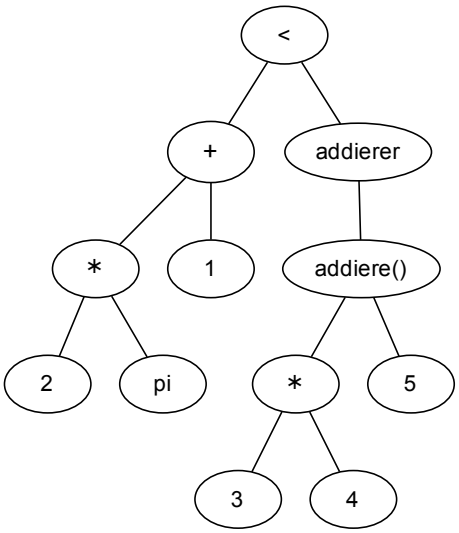


Abbildung 5.1: Baumdarstellung des Audrucks $1 + 2 * \texttt{pi} < \texttt{addierer.addiere}(3 * 4, 5)$

Im Syntaxbaum können auch die Typen notiert sein, ohne dass es unübersichtlich wird. Die Visualisierung mit einem Syntaxbaum ist damit bereits sehr hilfreich. Gleichzeitig ist sie jedoch auch sehr abstrakt und verlangt viel Vorstellungsvermögen. Es ist nicht möglich wirklich zu „sehen“, wie ein Auswertungsschritt ausgeführt wird.

5.1.3 Ausführen eines Auswertungsschritts

Um einen Auswertungsschritt zu veranschaulichen, ist die einfachste Möglichkeit, ihn am Ausdruck auszuführen. Für den (ungeklammerten) Ausdruck könnte das beispielsweise wie folgt aussehen:

$$
\begin{aligned}
&1 + 2 * \texttt{pi} < \texttt{addierer.addiere}(3 * 4, 5)\\
&1 + 2 * 3.141 < \texttt{addierer.addiere}(3 * 4, 5)\\
&1 + 6.238 < \texttt{addierer.addiere}(3 * 4, 5)\\
&7.238 < \texttt{addierer.addiere}(3 * 4, 5)\\
&7.238 < \texttt{addierer.addiere}(12, 5)\\
&7.238 < 17\\
&\texttt{true}
\end{aligned}
$$

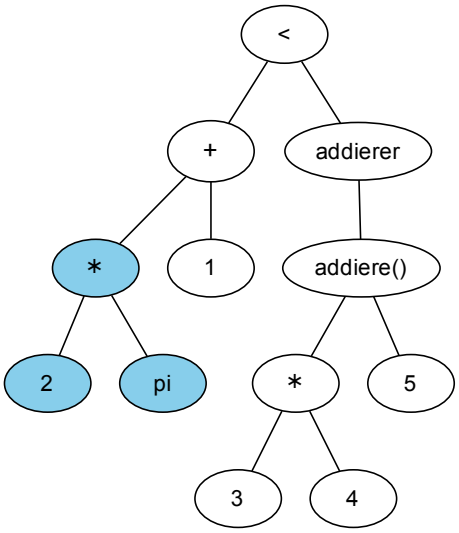
Auf ähnliche Weise lässt sich auch die statische Typprüfung sehr einfach visualisieren. Anstatt das Ergebnis direkt zu berechnen, wird lediglich der Ergebnis-Typ angezeigt:

$$
\begin{aligned}
&1 + 2 * \texttt{pi} < \texttt{addierer.addiere}(3 * 4, 5)\\
&1 + 2 * \texttt{float} < \texttt{addierer.addiere}(3 * 4, 5)\\
&1 + \texttt{float} < \texttt{addierer.addiere}(3 * 4, 5)\\
&\texttt{float} < \texttt{addierer.addiere}(3 * 4, 5)\\
&\texttt{float} < \texttt{addierer.addiere}(\texttt{int}, 5)\\
&\texttt{float} < \texttt{int}\\
&\texttt{boolean}
\end{aligned}
$$

So lässt sich gut der Unterschied zwischen der Compilerzeit, zu der lediglich die Typen der Teilausdrücke bekannt sind, und der eigentlichen Auswertung, zu der die Teilergebnisse bekannt sind, aufzeigen. Gleichzeitig sollte sich mit jedem Schritt auch der Syntaxbaum entsprechend verändern. Nun ist sichtbar, wie bei einem Methodenaufruf zunächst die Operanden ausgewertet werden und auch der Inhalt der Variablen `pi` wird angezeigt, bevor er verwendet wird.

Damit schnell sichtbar wird, an welcher Stelle im Ausdruck sich gerade etwas verändert, sollte der Teilausdruck, der gerade ausgewertet wird, graphisch hervorgehoben werden,

zum Beispiel durch Einfärben (Abb. 5.2).



$1 + 2 * \texttt{pi} < \texttt{addierer.addiere}(3 * 4, 5)$

Abbildung 5.2: Baum und Ausdruck, bei denen der Teilausdruck, der als nächstes ausgewertet wird, blau hervorgehoben ist.

5.1.4 Typumwandlung, Variablen und andere Besonderheiten

Es gibt einige Konzepte, die sich selbst mit den bisher beschriebenen Mitteln nur schwer visualisieren lassen. In Java gibt es keine Multiplikation zwischen zwei verschiedenen Typen und doch werden Zahlen von verschiedenen Typen multipliziert und später verglichen. Dass an dieser Stelle implizite Typumwandlungen geschehen, muss ebenfalls verdeutlicht werden. In der Auswertung lässt sich dies leicht als zusätzlicher Schritt einfügen:

$2 * 3.141$

$\mathbf{2.0} * 3.141$

6.238

Auch im Syntaxbaum kann es einen entsprechenden Knoten für implizite Typumwandlung (ITU) geben (Abb. 5.3).

Variablen lassen sich in beiden Repräsentation, dem Ausdruck und Syntaxbaum, leicht

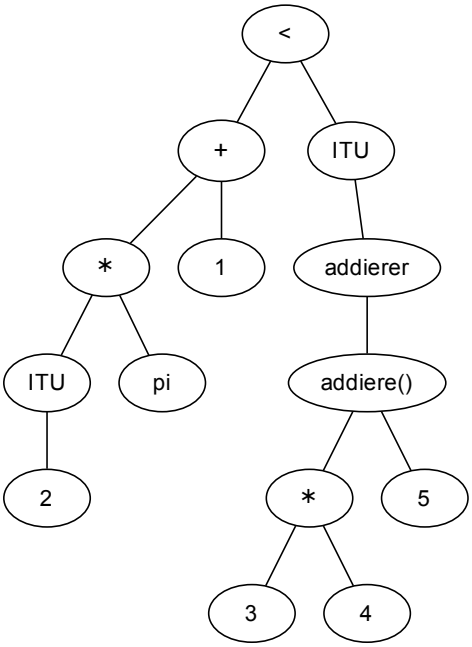


Abbildung 5.3: Baumdarstellung mit Knoten für implizite Typumwandlung (ITU)

durch ihren Bezeichner darstellen. Wird der Wert einer Variablen verwendet, so lässt sich auch dies als eigener Schritt einbauen: der Bezeichner wird durch den Wert ersetzt (Abb. 5.4). Um nicht den Überblick über alle Variablenbelegungen zu verlieren, kann in einer Variablenliste festgehalten werden, welche Belegung die Variablen zu jedem Zeitpunkt haben.

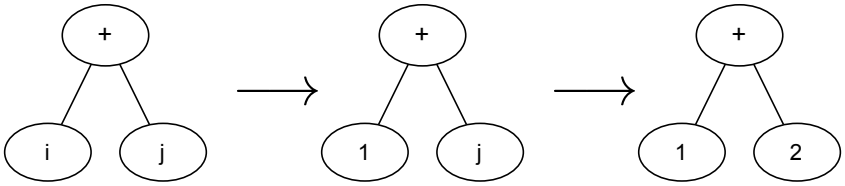


Abbildung 5.4: Auswertung von Variablen

Es gibt jedoch auch Besonderheiten, die sich nur schwer am Ausdruck selbst erschließen lassen, wie Gleitkommaungenauigkeiten oder der Auswertungsstopp bei boolschen Ausdrücken, wenn das Ergebnis bereits feststeht. Selbst wenn die betroffene Gleitkommazahl farblich hervorgehoben würde, wäre noch immer unklar, was nun das Problem sein könnte. Am einfachsten ist es hier tatsächlich, einen textuellen Hinweis zu dem Auswertungsschritt zu geben:

`123456789f` + 1 Achtung: Ungenauigkeit von Gleitkommazahlen!

`1.23456792E8` + 1

5.1.5

All diese Aspekte der Visualisierung sind wichtig und sollten deshalb in einem Tool, das der Visualisierung von Ausdrücken dient, berücksichtigt werden. Das Tool sollte also in der Lage sein,

- einzelne Schritte an einem Ausdruck auszuführen (vorwärts und rückwärts)
- den Abstrakten Syntaxbaum zu visualisieren
- die Typen der Teilausdrücke im Baum anzuzeigen
- implizite Typumwandlung und Variablenzugriff als eigenen Schritt durchzuführen
- den Ausdruck zu klammern
- eine Compiler-Sicht der Auswertung anzuzeigen und
- Tipps zum Verständnis des Auswertungsschrittes anzubieten.

5.2 Implementation in BlueJ vs. eigenständiges Tool

Es gibt zwei verschiedene Möglichkeiten, ein Tool zur Visualisierung der Ausdrucksauswertung zu erstellen.

Die erste Möglichkeit ist ein völlig eigenständiges Tool. Das Tool würde ein Feld zur Eingabe des Ausdrucks bereitstellen und diesen Ausdruck dann visualisieren. Der große Vorteil, der auffällt, ist, dass sehr leicht mit dem Ausdruck experimentiert werden kann. Die Eingabe muss nur verändert werden und die Auswirkungen sind sofort sichtbar. Außerdem wäre es leicht, das Tool allen Übungsteilnehmern zur Verfügung zu stellen. Jedoch müssten Variablen innerhalb des Tools angelegt und verwaltet werden. Während dies für primitive Typen leicht möglich ist, stellt sich die Frage, woher Referenzvariablen genommen werden sollen. Das Tool stünde in keinem Kontext zu existierenden Klassen oder Projekten. Das bedeutet, es gäbe eine Einschränkung auf String und Klassenobjekte wie Math. Besonders Referenzvariablen sind jedoch wichtig zur Veranschaulichung der Auswertung von Methodenaufrufen. Ohne weitere Unterstützung bei der Ausdrucksauswertung müsste außerdem Reflection verwendet werden, um Methoden aufzurufen.

Die Alternative ist eine Verwendung des Tools in einem Kontext. Den idealen Kontext lie-

fert die Entwicklungsumgebung BlueJ. Sie wird ohnehin bereits in der Lehre verwendet und bietet die Möglichkeit, leicht interaktiv Objekte von Klassen zu erzeugen. Nimmt man zum Beispiel die Direkteingabe als Kontext, so muss man die deklarierten Variablen lediglich importieren und muss sich nicht um deren Verwaltung kümmern. Auch die Auswertung von Methodenaufrufen kann von BlueJ übernommen werden und muss nicht selbst implementiert werden. Das entstehende Szenario könnte also sein, dass ein Ausdruck in die Direkteingabe eingegeben wird und sich das Tool öffnet, alle deklarierten Variablen lädt und anzeigt und dann die Auswertung visualisiert. Das Verändern des Ausdrucks ist hier umständlicher, denn er muss jedes Mal in die Direkteingabe eingegeben werden, bevor eine Visualisierung möglich ist. Es ist außerdem schwieriger, das Tool allen Teilnehmen zur Verfügung zu stellen, da nur eine Implementation als BlueJ-Extension auf allen Rechnern installiert werden kann und die Extension-Schnittstelle sehr restriktiv ist. Der große Vorteil ist jedoch, dass Referenzvariablen sehr einfach erzeugt und verwendet werden können.

Eine dritte Möglichkeit wäre eine Mischung aus den beiden vorigen: das Tool zwar eigenständig zu machen, aber den Code, den BlueJ zur Objekterzeugung und Ausdrucksauswertung verwendet, zu übernehmen. Damit löst man zwar das Problem, dass kein Kontext für Referenzvariablen existiert, jedoch bedeutet es wahrscheinlich einen erheblich höheren Arbeitsaufwand als die beiden anderen Implementationen.

Die Vor- und Nachteile sind noch einmal in Tabelle 5.1 zusammengefasst.

	Eigenständiges Tool	**Implementation in BlueJ**	**Mischung**
Experimentieren	schnell	umständlich	schnell
Erstellen von Variablen	innerhalb des Tools	in BlueJ	innerhalb des Tools
Referenzvariablen möglich	nein	ja	ja
Methodenaufrufe	über Reflection	in BlueJ	in BlueJ-Code
Bereitstellung des Tools	leicht	nur als BlueJ-Extension	leicht
Geschätzter Aufwand	normal	normal	hoch

Tabelle 5.1: Zusammenfassung der Vor- und Nachteile der Implementationsmöglichkeiten des Tools

Der Aufwand für die Mischung wird im Vergleich zu den beiden anderen Möglichkeiten höher eingeschätzt. Der bestehende Code müsste mit allen Abhängigkeiten aus BlueJ extrahiert und für das Tool angepasst werden. Das erfordert deutlich mehr Einarbeitungszeit in die Funktionalität von BlueJ als das reine Verständnis der Direkteingabe und einen Mehraufwand durch das Anpassen des Codes das Tool.

Da auf Referenzvariablen und Methodenaufrufe nicht verzichtet werden soll und die Mischung beider Ansätze zuviel Zeit in Anspruch nähme, wird das Tool in BlueJ integriert werden mit der Aussicht, es als Extension bereitzustellen.

5.3 Möglichkeiten der Einbindung in BlueJ

Die Direkteingabe ist eine der möglichen Einbindungsstellen des Tools. Sie ist bereits eine Unterstützung zum Verständnis der Ausdrucksauswertung innerhalb BlueJs und bietet die Möglichkeit, leicht Variablen anzulegen und Objekte zu erzeugen.

Eine zweite Variante wäre der Editor. Das Tool könnte durch markieren eines Ausdrucks im Quelltext und anschließendes Wählen des Eintrags im Kontextmenü gestartet werden. Den Kontext bilden dann die in der Methode oder Klasse bekannten Variablen. Da man für jeden Einsatz des Tools erst einmal eine Methode in einer Klasse schreiben müsste und die Belegung einiger Variablen erst zur Laufzeit bekannt ist, eignet sich der Editor nicht als Haupteinbindungsstelle.

Schließlich ist auch ein Eintrag im Tools-Menü möglich. Obwohl dieser Ansatz sehr einfach als Extension umzusetzen wäre, läuft er wieder auf ein eigenständiges Tool mit fehlendem Kontext hinaus.

Da die Direkteingabe die meisten Vorteile bietet, wird sie als Einbindungsstelle genutzt werden. Es lohnt sich aber sicher, in einer eventuellen Anschlussarbeit auch die anderen Möglichkeiten zur Verfügung zu stellen.

5.4 Ziel: Funktionalität des Tools

Das Tool soll aus der Direkteingabe durch Eingabe eines Ausdrucks gestartet werden. Dabei soll jeder Java-Level-1-konforme Ausdruck vom Tool akzeptiert werden. Zusätzlich zu der Berechnung des Ergebnisses in der Direkteingabe wird die Auswertung dann innerhalb einer eigenen grafischen Oberfläche visualisiert.

Es gibt einen Bereich, in dem die schrittweise Auswertung stattfindet. Der gerade betrachtete Teilausdruck wird dabei hervorgehoben. Die vom Tool verwendeten Variablen sollen in einer eigenen Liste angezeigt werden. Zusätzlich wird der Syntaxbaum des Ausdrucks dargestellt. Auch hier werden die Knoten des aktuellen Teilausdrucks eingefärbt. Es soll Buttons zum Ausführen eines Evaluationsschritts und dem Umkehren eines Schritts geben. Mit jedem Schritt werden links der aktualisierte Ausdruck hinzugefügt und Baum und Variablen aktualisiert.

Gibt es bei einem Schritt Erklärungsbedarf, wie zum Beispiel wenn implizite Typumwandlung stattfindet, so wird ein Hinweis angezeigt, der den Schritt kurz erläutert. Ist zu einem Schritt eine längere Erklärung notwendig, so soll diese in einem entsprechen-

den Tooltip untergebracht werden.

Bei dem Auswertungsfeld sind verschiedene Ansichten möglich: Detailauswertung mit Hinweisen, Auswertung mit geklammerten Ausdrücken und die Sicht zur Übersetzungszeit. Zwischen den Sichten soll zu jedem Zeitpunkt der Auswertung beliebig gewechselt werden können.

Der Baum beinhaltet Knoten für Operatoren und Operanden, aber auch für implizite Typumwandlung. Zu jedem Knoten ist auch der Typ notiert. Wenn mit dem Cursor über einen Knoten gefahren wird, so wird ein Tooltip angezeigt, der zusätzliche Informationen über den Knoten enthält.

Es soll möglich sein, das Tool mehrmals zu starten, um verschiedene Ausdücke zu vergleichen.

Sollte ein Ausdruck vom Tool nicht auswertbar sein, so soll stattdessen eine entsprechende Fehlermeldung in der Direkteingabe angezeigt werden. Alle Ausdrücke, die in die Direkteingabe eingegeben werden, sollen, unabhängig davon, ob sie nur in der Direkteingabe oder auch im Tool ausgewertet werden, weiterhin mit den Pfeiltasten wiederverwendet werden können.

6 Implementation eines Prototyps

Im Rahmen dieser Arbeit ist ein Prototyp des Tools entstanden. Zunächst wird kurz die Funktionalität des Prototyps mit dem Entwurf verglichen und anschließend die Implementation erläutert. Die Implementation des Tools wird von „innen“ nach „außen“ beschrieben. Dabei ist „innen“ der Start der Auswertung und Kern des Tools: die Grammatik; „außen“ ist die Umgebung in der es Verwendet wird: BlueJ.

6.1 Funktionalität des Prototyps

Der Protoyp lässt sich aus der Direkteingabe durch bestätigen eines Ausdrucks mit `Strg+Enter` starten. Ein Screenshot der grafischen Oberfläche ist in Abbildung 6.1 zu sehen.

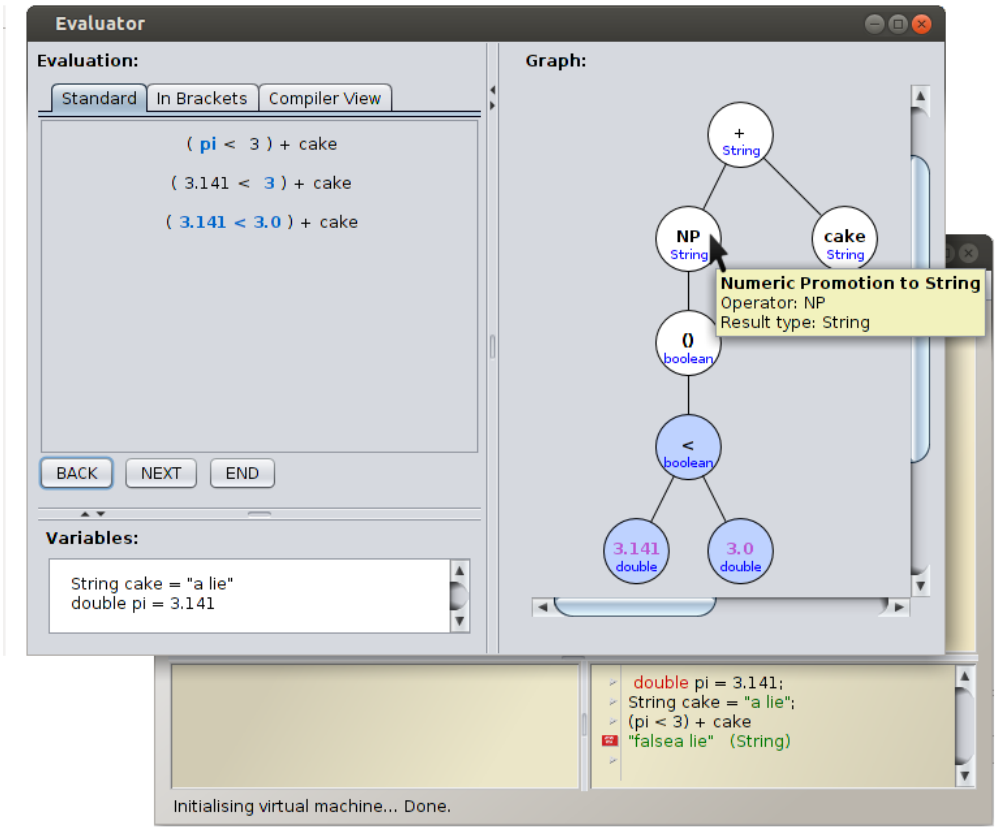


Abbildung 6.1: Starten und Ansicht des Tools

Links befindet sich das Auswertungsfeld. Die einzelnen Schritte werden hier alle untereinander aufgelistet. In jedem Ausdruck wird der zunächst auszuwertende Teilausdruck farblich hervorgehoben. Das Auswertungsfeld bietet außerdem die drei gewünschten Ansichten – Normale Auswertung, geklammert und Compiler-Sicht – zwischen denen zu jedem Zeitpunkt beliebig gewechselt werden kann.

Unter dem Auswertungsfeld befinden sich die Buttons zum Navigieren durch die einzelnen Auswertungsschritte. Zusätzlich zum Entwurf gibt es auch einen Button zum Springen an das Endergebnis der Auswertung (`END`).

Rechts neben den Buttons ist Platz für die Kurzbeschreibug der Tipps. Zu jedem Tipp gibt es auch einen Tooltip, in dem nach Bedarf eine längere Erläuterung zu finden ist. Es gibt solche Hinweise für den Modulo-Operator, Zuweisung, Vergleich, logische Operatoren, implizite Typumwandlung sowie `++` und `--`.

Unter den Buttons befindet sich die Variablenliste. Hier werden nur diejenigen Variablen angezeigt, die auch im Ausdruck vorkommen. Werden also beispielweise drei Variablen in der Direkteingabe deklariert, aber keine verwendet, so ist diese Liste leer. Die Variablenliste aktualisiert sich außerdem automatisch, sobald sich innerhalb des Prototypss die Variable ändert, zum Beispiel durch eine Zuweisung. Die Variablenliste ist nach dem Start des Prototyps unabhängig von der Direkteingabe. Sie ist lediglich eine Kopie des Stands der Direkteingabe von dem Zeitpunkt, als die Visualisierung gestartet wurde. Es gibt anschließend keine Wechselwirkungen mehr zwischen der Variablenliste des Tools und der Direkteingabe. Tatsächlich haben die Variablen in der Direkteingabe nach Starten des Prototyps bereits den neuen Wert nach Auswertung des Ausdruck.

Rechts ist der abstrakte Syntaxbaum visualisiert. Hier finden sich auch die Typen der Teilausdrücke. Außerdem bietet er Interaktionsmöglichkeiten. Die Standardfunktionen von JUNG sind Zoom, Verschieben, Drehen und perspektivische Verzerrung. Die Befehle hierfür sind in Tabelle 6.1 zusammengefasst.

Befehl	**Aktion**
Maus-Scrollrad	Zoom
Drag & Drop	Verschieben
Shift + Drag & Drop	Drehen
Strg + Drag & Drop	Perspektivische Verzerrung
Hover	Tooltip

Tabelle 6.1: Interaktionsmöglichkeiten mit dem Baum

Zu jedem Knoten gibt es einen Tooltip, der die Knotenbezeichnung, den Typ und, je nachdem, den Operator beziehungsweise den Wert des Literals oder Namen des Bezeichners

beinhaltet. Somit kann auch bei starkem Verkleinern des Baumes noch immer auf die Information der einzelnen Knoten zugegriffen werden. Falls der Baum sehr groß ist, kann es passieren, dass in dem Bereich oben links, der vom Prototyp zu Beginn standardmäßig angezeigt wird, kein Knoten enthalten ist und damit das Baum-Panel leer erscheint. In dem Fall muss ein Stück nach rechts und/oder unten gescrollt oder der Baum verkleinert werden.

Im Gegensatz zum Entwurf wird die Gleitkommaproblematik wird nicht thematisiert. Sie erfordert eine Möglichkeit zum Erkennen von Genauigkeitsverlusten und einer entsprechenden Visualisierung, was im Rahmen dieser Arbeit nicht umsetzbar war.

Ausdrücke mit Referenzvariablen oder Methodenaufrufen sind ebenfalls nicht möglich. Um im Tool angemessen mit Referenzen arbeiten zu können, müssten sie auch innerhalb des Tools verwaltbar und darstellbar sein. Bisher wurde sich von Variablen primitven Typs lediglich eine Kopie für das Tool erstellt, um Seiteneffekte auf die Direkteingabe zu verhindern. Das ist mit Referenzvariablen jedoch nur schwer möglich. Zusätzlich gibt es die Problematik von verändernden Methoden. Innerhalb des Tools gibt es die Möglichkeit, Schritte rückwärts zu machen. Wird die Methode im nächsten Schritt erneut aufgerufen, würde sie jedoch unter Umständen ein anderes Ergebnis liefern. Das Problem lässt sich umgehen, indem die Ergebnisse nach der ersten Auswertung gespeichert und beim erneuten Aufruf einfach wiederverwendet werden. Dafür müsste jedoch die bisherige Verwaltung von Variablen im Tool angepasst werden. Noch schwieriger ist der entstehende Seiteneffekt auf die Direkteingabe, wenn diese den Ausdruck ebenfalls auswertet und dann zu einem unterschiedlichen Ergebnis kommt.

Der Vergleich des Prototyps zum Entwurf ist noch einmal in Tabelle 6.2 zusammengefasst.

6.2 Implementation des Tools

In diesem Kapitel soll die Implementation des Tools an sich erklärt werden: wie ein Ausdruck geparst wird, wie er anschließend in Java repräsentiert wird und wie die schrittweise Auswertung und Visualisierung realisiert wurden.

	Entwurf	Prototyp
Ausdrücke	Java-Level-1-Ausdrücke möglich	nur primitive Typen
	Typen der Teilausdrücke sichtbar	✓
	Implizite Typumwandlung	✓
	Variablenzugriff	✓
	Tips für Besonderheiten in Ausdrücken	✓
	Gleitkommaproblematik	nicht thematisiert
Visualisierung	Auswertungsschritte möglich	✓
	Abstrakter Syntaxbaum	✓
	Variablenliste	✓
	Klammerungs-Sicht	✓
	Compiler-Sicht	✓
Funktionalität	mehrfach startbar	✓
	Fehlerausgabe in Direkteingabe	✓

Tabelle 6.2: Vergleich von Entwurf und Prototyp

6.2.1 Grammatik und ANTLR

Parsergenerator

Für das Parsing des Ausdrucks ist ein Parser notwendig. Der Parser für dieses Tool wurde mit dem ANTLR-Parsergenerator erzeugt. ANTLR[Par13] erlaubt das Generieren von Lexern, Parsern, Baum-Parsern und kombinierten Lexer-Parsern. Von ANTLR generierte Parser können automatisch Abstrakte Syntaxbäume generieren, die von Baum-Parsern weiterverarbeitet werden können. Das macht ANTLR für dieses Tool besonders geeignet. Außerdem bietet ANTLR eine konsistente Notation zur Spezifikation von Lexer, Parser und Baum-Parser. Dadurch hebt es sich von anderen Parsergeneratoren ab und ist leicht zu benutzen.

Grammatik für die Baum-Gererierung

Für das Generieren des Parsers wurde eine eigene Grammatik für Java-Level-1-Ausdrücke erstellt (Anhang A.1). Diese Grammatik definiert, wie ein gültiger Java-Level-1-Ausdruck auszusehen hat. In der Grammatik sind lexikalische Analyse (die Definition, wie Tokens aussehen) und Syntaxanalyse (die syntaktische Struktur von Ausdrücken) kombiniert. Sie besteht aus *Lexerregeln* für die Tokendefinition und *Parserregeln*, die die Baumstruktur definieren, die aus diesen Tokens aufgebaut werden soll.

Analog zur ANTLR Reference[Par07] beginnt die Grammatik mit der Regel `prog`:

```
prog   :       expr
       ;
```

Der Doppelpunkt steht hierbei für die Ableitung von der „linken“ Seite zur „rechten“. Das Semikolon schließt eine Regel ab. Mehrere Ableitungsmöglichkeiten werden mit einem `|` voneinander getrennt, wie zum Beispiel:

```
BOOL   :       'true'
       |       'false'
       ;
```

Es ist Konvention in ANTLR, dass diese drei Zeichen(`:`,`|` und `;`) zur besseren Übersicht immer in einer Linie untereinander stehen. Genauso werden nach Konvention Nichtterminale, die später nicht im Abstrakten Syntaxbaum auftauchen sollen (wie `prog` oder `expr`), kleingeschrieben. Sie bilden die Parserregeln. Alle Terminale der Grammatik werden automatisch von ANTLR als Tokens erkannt. Man kann ihnen jedoch selbstdefinierte Namen geben, wie `BOOL` für Booleanliterale. Tokennamen werden nach Konvention großgeschrieben und können am Anfang der Grammatik per Hand zur Tokenliste hinzugefügt werden, um eine Ableitungsregel zu sparen. Sie bilden die Lexerregeln und sorgen außerdem dafür, dass die Bezeichner in der generierten Parser-Datei besser lesbar sind.

```
tokens {
        PAR;            // Klammern
        ADD = '+';      // Arithmetisch
        SUB = '-';
        ...
}
```

Alle Ableitungen ohne Verzweigung werden von ANTLR als solche erkannt und entsprechend in den (später entstehenden) Baum eingefügt. Bei Verzweigungen ist es jedoch nötig, von Hand anzugeben, welches der Tokens oder Nichtterminale nun der Vater-Knoten werden soll und welches das Kind. Das geschieht einfach, indem an das Vater-Token ein Dach-Symbol (`^`) angehängt wird. Im Fall der Addition und Subtraktion kann das beispielsweise folgendermaßen aussehen:

```
arithExpr      :       multExpr ((ADD^|SUB^) multExpr)*
               ;
```

Ohne das Dach-Symbol wäre es eine einfache Ableitungsregel, nun ist es eine Ableitungsregel mit zusätzlicher Information darüber, wie der abstrakte Syntaxbaum später aussehen soll: mit dem Additions- oder Subtraktionszeichen als Vaterknoten.

Eine andere mögliche Schreibweise für den Vater-Knoten ist

```
^(parent child child ...)
```

Dabei wird das erste Element der Liste als Vaterknoten angesehen und alle folgenden Elemente als Kinder.

Es kann vorkommen, dass es schwierig ist, ein Token direkt in der zugehörigen Ableitungsregel zu definieren. Ein Paar von öffnenden und schließenden Klammern muss beispielweise zu einem Token zusammengefasst werden. Hierfür gibt es eine weitere Möglichkeit, die Baum-Erstellung zu steuern, den Rechtspfeil (->):

```
literal     :     ...
            |     '(' expr ')' -> ^(PAR expr)
            ;
```

Er kann sinnbildlich gelesen werden als „Wenn die linke Seite gematcht wird, dann schreibe sie um zu der Form auf der rechten Seite“. Auf diese Weise lassen sich auch leicht Ableitungsregeln der Form `Token : Zeichenkette` sparen, wie im Fall der Postfix-Operatoren ++ und --:

```
postfixOp   :     '++' -> PINCR
            |     '--' -> PDECR
            ;
```

Falls die Ableitungsregel eines Tokens sehr lang ist, kann sie zur besseren Lesbarkeit zergliedert werden. Damit die Teil-Ableitungen nicht als Tokens angesehen werden, markiert man sie mit dem Schlüsselwort `fragment`. Ein Gleitkomma-Literal kann zum Beispiel in der Exponenten-Darstellung vorliegen. Um den Exponent besser lesbar zu machen und Code-Duplizierung zu vermeiden, wird er als Fragment eingefügt.

```
DOUBLE      :     ...
            |     ('0'..'9')+ EXPONENT ('d'|'D')?
            |     ...
            ;

fragment
EXPONENT    |     ('e'|'E') ('+'|'-')? ('0'..'9')+
            ;
```

Zusätzlich zu den Ableitungsregeln muss in der Grammatik noch angegeben werden, in welcher Programmiersprache der Parser generiert wird und welche Ausgabe er liefern soll. Da für die Auswertungsvisualisierung der Abstrakte Syntaxbaum interessant ist, wurde als Ausgabetyp AST gewählt. Die generierten Parser-Java-Dateien werden später Teil des Java-Projekts, daher kann man zusätzlich noch den Datei-Header angeben (Paketname, Imports) und auftretende Fehler bei der Ableitung als Exception weitergeben, damit sie im Programm behandelt werden können. Fehler werden sonst standardmäßig an der Konsole ausgegeben und könnten nicht in die grafische Oberfläche eingebunden werden.

Baum-Grammatik für die Code-Generierung

ANTLR generiert aus der Grammatik einen Parser, der aus einem Ausdruck einen Abstrakten Syntaxbaum erstellt. Dieser Baum muss jedoch durchlaufen werden können,

um den Ausdruck, den er repräsentiert, auszuwerten. Die Struktur, die der Syntaxbaum dafür ausweisen muss, lässt sich in einer so genannten *Baum-Grammatik* [Par07, S.79] festlegen.

Die Baum-Grammatik für diese Arbeit befindet sich in Anhang A.2. Die Notation ist sehr ähnlich zu der Parser-Grammatik. Auch hier gibt es Optionen und auch diese Grammatik beginnt mit einer Startregel `prog`. Ein Unterschied ist jedoch, dass die Baum-Grammatik keinen Ausdruck erhält, sondern die Struktur des Abstrakten Syntaxbaums mit Tokens, die durch die Parser-Grammatik erzeugt wurde. Sie muss somit für jedes Token nur noch entscheiden, welcher Befehl ausgeführt werden soll. Damit schrumpft die Grammatik auf eine einzelne Regel:

```
expr returns [Node node]
      :         ^(ADD a=expr b=expr) {$node = new AdditionNode(a, b);}
      |         ^(SUB a=expr b=expr) {$node = new SubstractionNode(a, b);}
      |         ...
      ;
```

Dabei sind `AdditionNode` und `SubstractionNode` selbst definitiert Java-Klassen. Für jedes mögliche Token werden sich die zukünftigen Kinder als `a` oder `b` gemerkt und dann die entsprechende Java-Anweisung ausgeführt. Diese erzeugt ein Java-Objekt, das dieses Token repräsentiert. Dadurch wird aus dem Tokenstream in Form eines Abstrakten Syntaxbaums ein Baum aus selbst definierten Java-Objekten aufgebaut. Diese Objekte können nun nach Belieben angepasst und durchlaufen werden.

6.2.2 Java-Darstellung des Baums

Für jedes Token der Eingabe gibt es genau ein Knoten-Objekt. Jeder Knoten hält Informationen über sich, wie zum Beispiel seine Kinder, seinen Typ und seinen Operator oder Wert. Ein Knoten kann selbst einen Auswertungsschritt auf sich (und seinen Kindern) durchführen. Das Durchführen eines Auswertungsschrittes generiert einen neuen Baum, in dem dieser Schritt umgesetzt ist. Der ursprüngliche Baum bleibt erhalten. Außerdem akzepiert ein Knoten Besucher und bietet sondierende Methoden für die spätere Visualisierung an der grafischen Oberfläche.

Knoten sind nach dem Schema *TokennameNode* benannt und implementieren das gemeinsame Interface Node. Die vollständige Interface- und Vererbungshierarchie ist in Anhang A.3 dargestellt.

Der Baum wird von der Klasse `EvalGraph` verwaltet. Sie parst den Ausdruck, erzeugt den zugehörigen Baum und validiert diesen. Zur Validation gehört das Zuweisen der Werte zu Variablen, Typprüfung und das Hinzufügen von Knoten für die implizite Ty-

pumwandlung. Alle Überprüfungen geschehen mit *Besuchern*.

Besucher-Muster

Das *Besucher-Muster* (Visitor pattern) ist ein objektorientiertes Entwurfsmuster. Es bedeutet die Kapselung von Operationen, die auf allen Elementen einer Objektstruktur ausgeführt werden sollen, als ein Objekt. Dadurch ist es möglich, neue Operationen zu definieren, ohne die Klassen der betroffenen Objektstruktur verändern zu müssen [GHJV12, S. 301].

Um diese recht abstrakte Definition zu veranschaulichen, wird das Besucher-Muster im Folgenden anhand eines Beispiel erläutert.

Auf dem Abstrakten Syntaxbaum müssen beispielsweise Operationen für die Semantikanalyse, wie die Typprüfung oder die Prüfung ob alle Variablen definiert sind, angeboten werden. Außerdem möchte man eine Repräsentation des Baumes für die grafische Oberfläche erzeugen oder überprüfen, welcher Teilausdruck als nächstes ausgewertet wird. Die meisten dieser Operationen müssen Knoten, die eine Zuweisung repräsentieren anders behandeln als Knoten, die beispielsweise einen Integer-Wert darstellen. Das führt dazu, dass die Operation auf alle existierenden Knoten verteilt wird. Damit ist das System nicht nur schwer zu verstehen, auch Änderungen sind sehr aufwändig, da jedes mal alle Knotenklassen bearbeitet werden müssen.

Wünschenswert wäre es, wenn die neue Operation vom Baum unabhängig hinzugefügt werden könnte – und genau das wird durch das Besucher-Muster realisiert. Die Operation wird in einem separaten Objekt, dem *Besucher* (Visitor), gekapselt und dem Syntaxbaum bei der Traversierung übergeben. Wenn ein Element den Besucher akzeptiert, so ruft es an diesem eine dem Element ensprechende Methode auf und übergibt dabei sich selbst als Parameter. Der Besucher führt dann die Operation an diesem Element aus.

Um Besucher für mehr als eine Operation verwenden zu können, wird eine abstrakte Klasse *Besucher* für alle Besucher des Baumes benötigt. Diese muss eine Operation für jede Knotenklasse deklarieren[3]. Außerdem wird ein neues Interface für die bearbeiteten Klassen hinzugefügt, das die accept-Methode bereitstellt (Abb. 6.2).

Dadurch können neue Operationen erzeugt werden, indem einfach eine neue Klasse in die Besucher-Hierarchie eingefügt wird. Natürlich ist es nun schwieriger, neue Knotenklassen hinzuzufügen, denn für jede müssen nun alle Besucher-Klassen geändert wer-

[3] Da es in diesem Projekt sehr viele Knoten-Klassen gibt - und ähnliche Knoten oft das gleiche Verhalten zeigen - wurde zur besseren Übersicht nur eine Operation für jede Oberklasse deklariert: eine Operation für Knoten, die arithmetische Ausdrücke repräsentieren, eine für logische, für Literale, etc.

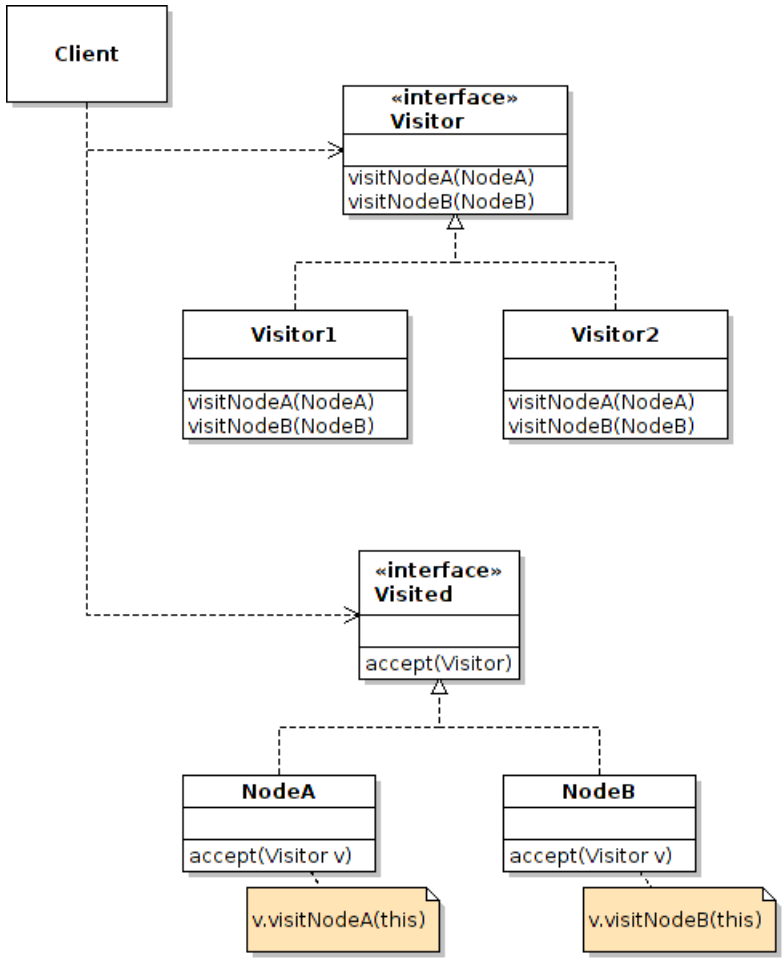


Abbildung 6.2: Aufbau des Besucher-Musters

den. Außerdem erfordert die Verwendung des Besuchers manchmal ein Aufbrechen der Kapselung, da die Schnittstellen der Knotenklassen mächtig genug sein müssen, um dem Besucher das Ausführen seiner Aufgabe zu ermöglichen. Das führt mitunter dazu, dass es notwendig wird, private Methoden und Zustandsfelder öffentlich zu machen.

In dieser Arbeit werden Besucher für Operationen genutzt, für die der Baum direkt nach seiner Erstellung traversiert werden muss:

- Hinzufügen von Knoten für die implizite Typumwandlung
- Prüfung, ob alle Variablen deklariert sind
- Bestimmung, welcher Teilausdruck im nächsten Schritt ausgewertet wird (für die Visualisierung im Abstrakten Syntaxbaum)

Es gibt jedoch auch einen Besucher, der die Labels für die UI-Darstellung sammelt und daher mehrmals in den Baum gereicht wird.

6.2.3 Visualisierung des Baums

Zeichnen von Bäumen in Java

Java bietet zwar Swing-Tools, mit denen einige grafische Objekte erzeugt werden können, doch sie eignen sich wenig dazu, bequem und schnell Graphen zu zeichnen, ohne sich viele Gedanken über räumliche Anordnung oder Ähnliches machen zu müssen. Es gibt Tools, die mit wenig Aufwand einfach Graphen zeichnen können. Meist müssen sie jedoch vorher vom Nutzer installiert werden. Besser eignet sich eine Bibliothek, die mit dem Tool mitgeliefert werden kann. Da das Tool in Java geschrieben ist, bietet sich eine Java-Bibliothek an. Frei im Netz gibt es dafür nur eine bekannte Bibliothek: *JUNG - the Java Universal Network/Graph Framework* [O+13]. Sie wird daher für die Visualisierung des Baumes in diesem Tool verwendet.

Visualisierung mit JUNG

Neben der Möglichkeit, die Auswertung kleinschrittig nachzuvollziehen, wird mit JUNG auch die Baumdarstellung an der grafischen Oberfläche visualisiert. Diese Visualisierung findet in der Klasse `TreePanel` statt. Das Hauptwerkzeug hierfür ist der Visualization-Viewer, ein parametrisierter Typ. Die Parameter sind die Typen der Knoten und Kanten. An dem Viewer können Einstellungen vorgenommen werden. Dazu gehören Größe und Farbe der Knoten, Labels, Kantenarten oder Tooltips. Dafür muss man meist zunächst auf den RenderContext zugreifen, um an diesem dann einen so genannten Transformer zu überschreiben.

JUNG baut laut ihrer Website auf Java-Swing-Tools auf. Diese Tools sind sehr limitiert und Transformer sind eine Möglichkeit, sie zu verändern. Daher muss beispielsweise der Tranformer, der dafür zuständig ist, dass Knoten eine bestimmte Größe haben, durch einen eigenen Transformer ersetz werden, falls man die Knotengröße ändern will. Die Transformer selbst werden zwar von der JUNG-Bibliothek mitgeliefert, sind jedoch nicht dokumentiert.

6.2.4 Paket-Struktur des Tools

Das Tool ist nach den Entwurfsregeln des *Werkzeug- und Material-Ansatzes* (WAM) [Zü05] in drei große Pakete unterteilt: Materialien, Services und Werkzeuge. Fachwerte, die vierte Kategorie, die es nach dem WAM-Ansatz gibt, existieren in diesem Projekt nicht.

Materialien sind anwendungsspezifische, veränderliche Gegenstände, wie Knoten, Karteikarten oder Plätze in einem Kinosaal. Ein Material hat ausschließlich fachliche Aufgaben, das heißt, die modellieren Konzepte des Anwendungsbereiches. Ein Material hat nur Beziehungen zu anderen Materialien (und Fachwerten) und informiert nicht selbstständig andere Objekte über Änderungen. Das Paket im Tool beinhaltet die Knoten des Baums und die Variablen.

Services bieten systemweite fachliche Dienstleistungen an, wie materialübergreifende Operationen oder Persistenz. Im Gegensatz zu Materialien gibt es von jedem Service nur ein Exemplar. Services arbeiten mit Materialen und anderen Services und verwalten oft Materialien. Sie können aktiv über Änderungen ihres eigenen Zustands oder dem der verwalteten Materialien informieren. Die Services im Prototyp dienen der Verwaltung des abstrakten Syntaxbaums: Sie führen Konsistenzprüfungen durch und bieten Operationen am Baum an. Außerdem verwalten sie die Variablenliste.

Werkzeuge modellieren die interaktiven Komponente eines Systems und haben eine grafische Benutzeroberfläche (UI). Sie visualisieren die Materialien und nutzen dafür Services. Werkzeuge erlauben es dem Benutzer, Materialien zu bearbeiten. Ein Werkzeug lässt sich unterteilen in eine technische UI-Klasse und eine fachliche Werkzeug-Klasse. Die UI-Klasse hat die Aufgabe, die benötigten UI-Komponenten und Layouts zu erzeugen und die relevanten Komponenten durch Getter an ihrer Schnittstelle zur Verfügung zu stellen. Die Werkzeug-Klasse verarbeitet Benutzereingaben und veranlasst entsprechende fachliche Operationen an den betroffenen Materialien oder Services. Sie vermittelt zwischen grafischer Benutzeroberfläche und fachlichen Klassen. Die Werkzeug-Klasse übernimmt dabei selbst keine fachlichen Aufgaben, sondern delegiert alle fachlichen Operationen an Materialien und Services. Außerdem aktualisiert sie die grafische Oberfläche.

Zusätzlich gibt es ein Paket *Grammar*, in dem die beiden Grammatiken, sowie die daraus generierten Parser und Lexer zu finden sind.

6.3 Integration in BlueJ

Das Tool ist in die Direkteingabe von BlueJ integriert. Daher wird zunächst kurz die Architektur von BlueJ beleuchtet und anschließend die Einbindung des Tools erklärt.

6.3.1 BlueJ: Auszug aus Architektur und Design

Der für diese Arbeit relevante Teil der BlueJ-Paketstruktur in Abbildung 6.3 zu sehen.

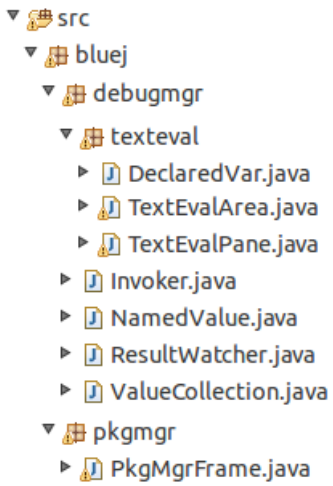


Abbildung 6.3: Auszug aus der Paket-Struktur von BlueJ

BlueJ ist ein System zum Verwalten und Bearbeiten von Projekten [Dok]. Jedes Projekt besteht aus Paketen und ein Paket wird in einem einzigen UI-Fenster repräsentiert (`bluej.pkmgr.PkgMgrFrame`). In der PkgMgrFrame-Klasse befindet sich der Großteil der UI-Funktionalität. Die meisten Interaktionsmöglichkeiten können von Menüs oder Buttons im Hauptfenster aus aufgerufen werden, dazu zählt auch die Direkteingabe (`bluej.debugmgr.texteval.TextEvalPane`).

Zum Auswerten von Code wird ein Invoker (`bluej.debugmgr.Invoker`) verwendet. Der Invoker schreibt den auszuwertenden Quelltext in eine „Shell"-Klasse und lässt diese die Auswertung durchführen. Die Shell-Klasse macht die interaktiv erzeugten Objekte und die in der Direkteingabe deklarierten Variablen verfügbar, indem sie die Werte aus einer Map bezieht, die in der Debug-Virtual-Machine existiert.

Das ResultWatcher-Interface wird vom Invoker genutzt, um Ergebnisse zurück an den Aufrufer zu geben. Tritt beim Compilieren der Shell-Klasse ein Fehler auf, so wird dieser an den Watcher zurückgegeben. Falls das Compilieren erfolgreich ist, wird die Shell-Klasse im Debugger ausgeführt und das Ergebnis der Auswertung an den Watcher gegeben.

6.3.2 Einbindung des Tools

Veränderte Klassen und Integration in die Paketstruktur

Das Tool hat eine eigene grafische Benutzeroberfläche, daher muss nur die Stelle in BlueJ verändert werden, an der es aufgerufen werden soll: die Direkteingabe. Die komplette

Logik der Direkteingabe findet in der Klasse `TextEvalPane` statt. Folglich ist dies die einzige zu verändernde Klasse. Die in der Direkteingabe vorhandenen Informationen müssen jedoch in eine Form gebracht werden, mit der das Tool arbeiten kann. Für diesen Zweck gibt es zwei neue Klassen. `EvalVisualizationStarter` nimmt die Verarbeitung der Daten vor und startet anschließend das Tool. Da die Werte der Variablen dem Tool bekannt sein müssen, wird hierfür ein `VariableResultWatcher` benötigt, der die Ergebnisse der Invoker-Aufrufe verarbeitet.

Außerdem muss auch der Quelltext des Tools in die Paketstruktur von BlueJ integriert werden. Da das Tool thematisch ein Werkzeug zum Debuggen ist und einen starken Bezug zur Direkteingabe hat, befindet sich der Quelltext ebenfalls im `debugmgr`-Paket, zusammen mit den zwei neuen Klassen (Abb. 6.4).

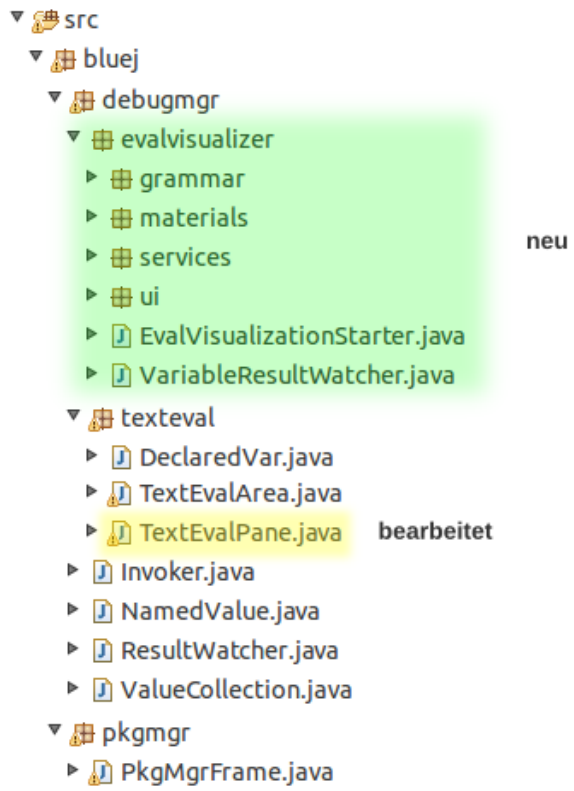


Abbildung 6.4: Auszug aus der Paket-Struktur von BlueJ mit integriertem Tool

TextEvalPane: Einfügen einer neuen Aktion

Das Tool soll aus der Direkteingabe heraus durch den Befehl `Strg+Enter` gestartet werden. Die Klasse `TextEvalPane` hat hierfür bereits eine Keymap, in die hierfür lediglich eine neuer Eintrag hinzugefügt werden muss (Abb. 6.5).

Die Aktion, die beim Drücken des Befehls `Strg+Enter` ausgeführt werden soll, wird in

```
private void defineKeymap()
{
    ...
    // OpenVisualizerAction
    action = new OpenVisualizerAction();
    newmap.addActionForKeyStroke(
      KeyStroke.getKeyStroke(KeyEvent.VK_ENTER, Event.CTRL_MASK), action);
    ...
}
```

Abbildung 6.5: Eintrag in die Keymap

der neuen inneren Klasse `OpenVisualizerAction` definiert. Das Verhalten ist ähnlich der normalen Auswertung eines Ausdrucks in der Direkteingabe. Entsprechend ist diese Klasse an der inneren Klasse `ExecuteCommandAction` orientiert, die für die Standardauswertung nach Drücken von `Enter` verantwortlich ist. Es wird jedoch zunächst nur das Tool zur Visualisierung gestartet.

Erst nach erfolgreichem Starten des Tools wird auch die Auswertung in der Direkteingabe angestoßen. Hierfür muss zusätzlich das Keymap-Objekt für `ExecuteCommandAction` (die Klasse, in der die Auswertung der Direkteingabe stattfindet) als Exemplarvariable gehalten werden.

EvalVisualizationStarter und VariableResultWatcher: Variablen erhalten ihre Werte

Der `EvalVisualizationStarter` erhält den auszuwertenden Ausdruck und die Liste der deklarierten Variablen. Falls keine Variablen deklariert wurden, so kann das Tool direkt gestartet werden. Ansonsten müssen zunächst die Werte der Variablen evaluiert werden. Hierfür werden einfach alle Variablen nacheinander einem Invoker übergeben. Da der Invoker nicht direkt ein Ergebnis zurückgibt, sondern als eigener Thread arbeitet, muss ihm ein ResultWatcher mitgegeben werden, an den er seine Ergebnisse meldet.

Die Auswertung der Variablen geschieht echt parallel und wir erhalten immer nur das Ergebnis. Es gibt jedoch keine Möglichkeit festzustellen, von welchem Aufruf des Invokers dieses Ergebnis kommt. Um das Ergebnis hinterher noch einer Variable zuordnen zu können, muss sich der ResultWatcher also merken, auf den Wert welcher Variable er gerade wartet. Aus diesem Grund gibt es eine eigene Klasse `VariableResultWatcher`, die neben dem Warten auf das Ergebnis auch noch den Namen der Variable hält.

Erst wenn alle Aufrufe des Invokers zurückgekehrt sind, wird, je nach Fall, entweder ein Fehler an der Direkteingabe ausgegeben oder das Tool gestartet. Das Tool erhält nun eine Map, in der jeder Variable der entsprechende Wert zugeordnet ist und kann damit arbeiten.

Nachdem es diese Werte einmal ausgelesen hat, wird jedoch nicht mehr darauf zugegriffen. Das bedeutet insbesondere, dass weitere Eingaben in der Direkteingabe keine Seiteneffekte auf die aktuelle Auswertung des Tools haben – und dass umgekehrt die Auswertung auch keine Seiteneffeckte auf die Direkteingabe hat. Die Direkteingabe verhält sich im Anschluss also genauso, als wäre der Ausdruck mit `Enter` eingegeben worden.

7 Zusammenfassung und Ausblick

Das entstandene Tool visualisiert die Ausrucksauswertung auf mehreren Ebenen. Durch Hinweise zu den einzelnen Schritten können auch Ausdrücke mit den Studenten unbekannten Operatoren oder unerwartetem Verhalten, wie der impliziten Typumwandlung, nachvollzogen werden. Die Auswertungsreihenfolge wird durch schrittweises Aufrollen des Ausdrucks sichtbar gemacht und zusätzlich in Form eines Abstrakten Syntaxbaumes dargestellt. Die Typen der Teilausdrücke sind ebenfalls im Abstrakten Syntaxbaum visualsiert, der auch Typumwandlungen beinhaltet. Die statische Typprüfung lässt sich mit einer Option zur Darstellung der Übersetzungssicht nachvollziehen. Hier wird auch der Unterschied zwischen Übersetzungszeit und Laufzeit sichtbar gemacht. Seiteneffekte auf Variablen (von primitiven Typen) können in der integrierten Variablenliste beobachtet werden.

Auf die Gleitkommaproblematik wird bisher überhaupt nicht eingegangen. Es sollte zumindest einen Hinweis dafür geben. Wünschenswert wäre es, dass dies auch direkt in der Auswertung auf irgendeine Art sichtbar wird. Das war im Rahmen dieser Arbeit jedoch nicht umsetzbar. Referenzvariablen und Methodenaufrufe sind ebenfalls nicht integriert. Hier liegt vermutlich der größte Mangel des Prototyps.

Um die Verständlichkeit des Codes zu erhöhen, wurden Programmierparadigmen, wie der Werkzeug-Materialansatz, Kapselung von Schnittstelle und Implementierung oder das Besuchermuster verwendet.

Da das Tool direkt in BlueJ eingebaut ist, ist nur schwer verteilbar. Um es an die Übungsteilnehmer zu verteilen, sollte es als BlueJ-Extension bereitgestellt werden.

Außerdem ist die grafische Oberfläche bisher komplett in Englisch gehalten. Eine Sprachverwaltung fehlt noch.

7.1 Erweiterungsvorschläge

Für mögliche Anschlussarbeiten sind an dieser Stelle Erweiterungsvorschläge zusammengefasst:

- Referenzvariablen und Methodenaufrufe integrieren

- Gleitkommaproblematik hervorheben
- das Tool als BlueJ-Extension bereitstellen
- das Tool auch in Editor und Tools-Menü integrieren
- Benutzeroberfläche verbessern
- im Baum-Panel standardmäßig zum Wurzelknoten scrollen
- Sprachverwaltung an BlueJ anpassen und Deutsch anbieten

A Anhang

A.1 EBNF-Grammatik für Java-Level1-Ausdrücke

```
grammar Expr;

options {
        language=Java;
        output=AST;
        ASTLabelType=CommonTree;
}

// Liste der Tokens
tokens {
        PAR;            // Klammern
        ADD='+';        // Arithmetisch
        SUB='-';
        MUL='*';
        DIV='/';
        MOD='%';
        NEG;
        POS;
        IINCR;          // ++, -- infix und postfix
        IDECR;
        PINCR;
        PDECR;
        NOT='!';
        EQ='==';        // Vergleich
        NEQ='!=';
        LE='<';
        LEQ='<=';
        GT='>';
        GEQ='>=';
        AND='&&';       // Logisch
        OR='||';
        INTCAST;        // Typcast
        LONGCAST;
        FLOATCAST;
        DOUBLECAST;
        CHARCAST;
        BOOLCAST;
}


@header {
```

```
package grammar;
}

@lexer::header {
package grammar;
}

@rulecatch {
    catch (RecognitionException e) {
        throw e;
    }
}

@lexer::members {
  @Override
  public void reportError(RecognitionException e) {
    throw new IllegalArgumentException(e);
  }
}

// Startpunkt der Grammatik
prog    :       expr
        ;

expr    :       orExpr
        ;

orExpr  :       andExpr (OR^ andExpr)*
        ;

andExpr :       equalityExpr (AND^ equalityExpr)*
        ;

equalityExpr    :       relationExpr ((EQ^|NEQ^) relationExpr)*
                ;

relationExpr    :       arithExpr ((GT^|LE^|GEQ^|LEQ^) arithExpr)*
                ;

arithExpr       :       multExpr ((ADD^|SUB^) multExpr)*
                ;

multExpr        :       unaryExpr ((MUL^|DIV^|MOD^) unaryExpr)*
                ;

unaryExpr       :       '-' postfixExpr -> ^(NEG postfixExpr)
                |       '+' postfixExpr -> ^(POS postfixExpr)
                |       '!' postfixExpr -> ^(NOT postfixExpr)
                |       '++' postfixExpr -> ^(IINCR postfixExpr)
                |       '--' postfixExpr -> ^(IDECR postfixExpr)
```

```
                |       '(' typecast ')' postfixExpr -> ^(typecast postfixExpr)
                |       postfixExpr
                ;

typecast        :       'int' -> INTCAST
                |       'long' -> LONGCAST
                |       'float' -> FLOATCAST
                |       'double' -> DOUBLECAST
                |       'char' -> CHARCAST
                |       'boolean' -> BOOLCAST
                ;

postfixExpr     :       atom (postfixOp^)?
                //|     accessExpr
                ;

postfixOp       :       '++' -> PINCR
                |       '--' -> PDECR
                ;

atom    :       literal
        |       identifier
        ;

literal :       INT
        |       LONG
        |       FLOAT
        |       DOUBLE
        |       BOOL
        |       CHAR
        |       STRING
        |       '(' expr ')' -> ^(PAR expr)
        ;

identifier      :       ID
                ;

INT     :       ('0'..'9')+
        ;

LONG    :       INT ('l'|'L')
        ;

DOUBLE  :       ('0'..'9')+ '.' ('0'..'9')* EXPONENT? ('d'|'D')?
        |       '.' ('0'..'9')+ EXPONENT? ('d'|'D')?
        |       ('0'..'9')+ EXPONENT ('d'|'D')?
        |       INT ('d'|'D')
        ;

FLOAT   :       (DOUBLE|INT)('f'|'F')
```

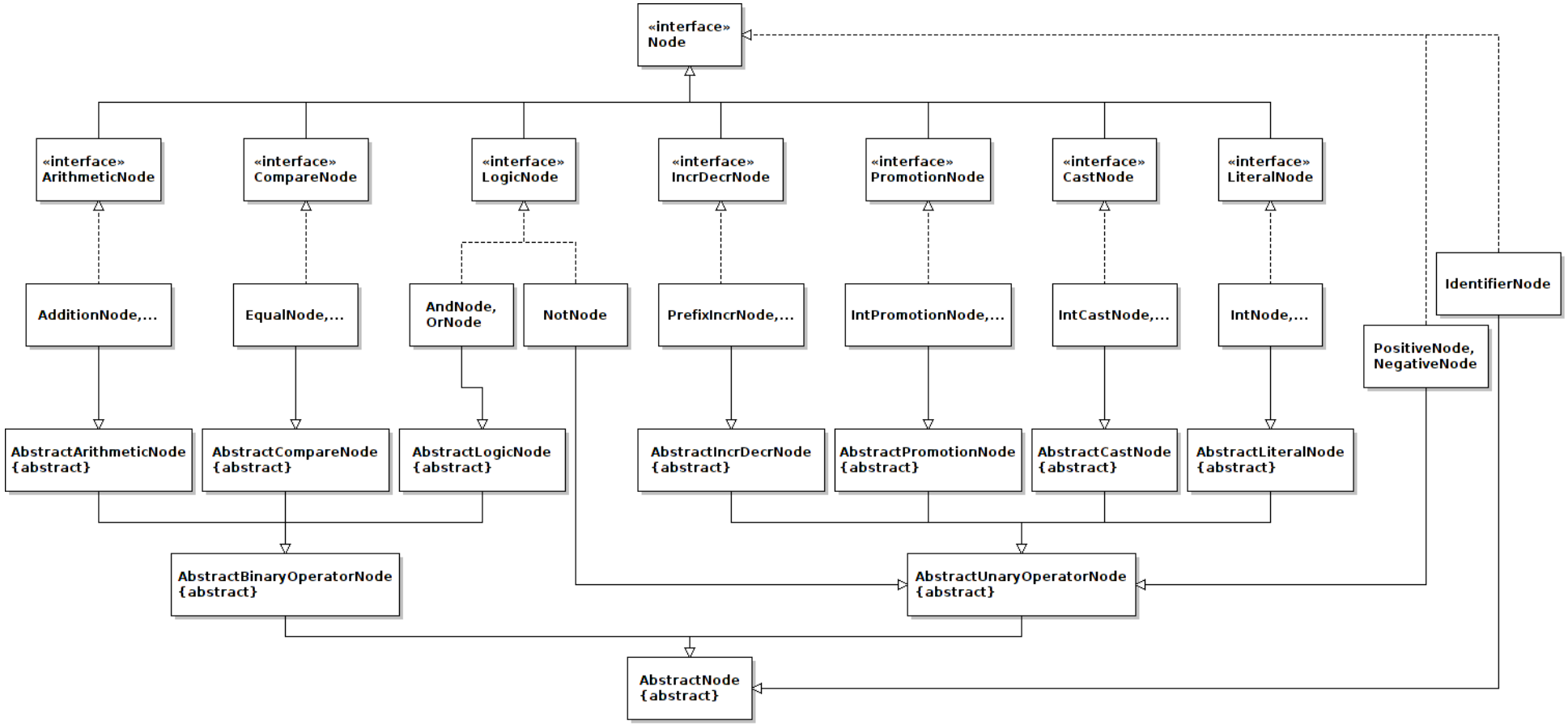
```
        ;

fragment
EXPONENT        :       ('e'|'E') ('+'|'-')? ('0'..'9')+
                ;

BOOL    :       'true'
        |       'false'
        ;

CHAR    :       '\''.'\''
        ;

STRING
        :       '"' ( ESC_SEQ | ~('\\'|'"') )*  '"'
        ;


fragment
HEX_DIGIT : ('0'..'9'|'a'..'f'|'A'..'F') ;

fragment
ESC_SEQ
    :   '\\' ('b'|'t'|'n'|'f'|'r'|'\"'|'\''|'\\')
    |   UNICODE_ESC
    |   OCTAL_ESC
    ;

fragment
OCTAL_ESC
    :   '\\' ('0'..'3') ('0'..'7') ('0'..'7')
    |   '\\' ('0'..'7') ('0'..'7')
    |   '\\' ('0'..'7')
    ;

fragment
UNICODE_ESC
    :   '\\' 'u' HEX_DIGIT HEX_DIGIT HEX_DIGIT HEX_DIGIT
    ;



ID      :       ('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'0'..'9'|'_')*
        ;

WS      :       (' ' | '\t' | '\n' | '\r')+ {skip();};
```

A.2 Grammatik zum Erzeugen der internen Baum-Repräsentation

```
/* Grammatik zum Erzeugen der internen Baum-Repraesentation */

tree grammar Eval;

options {
        tokenVocab=Expr; // read token types from Expr.tokens file
        ASTLabelType=CommonTree; // what is Java type of nodes?
}

@header {
package grammar;
import graph.node.*;
import graph.node.arithmetic.*;
import graph.node.compare.*;
import graph.node.literal.*;
import graph.node.logic.*;
import graph.node.unary.*;
import graph.node.typecast.*;
}

prog    :       expr
        ;

expr returns [Node node]
        :       ^(ADD a=expr b=expr) {$node = new AdditionNode(a, b);}
        |       ^(SUB a=expr b=expr) {$node = new SubstractionNode(a, b);}
        |       ^(MUL a=expr b=expr) {$node = new MultiplicationNode(a, b);}
        |       ^(DIV a=expr b=expr) {$node = new DivisionNode(a, b);}
        |       ^(MOD a=expr b=expr) {$node = new ModuloNode(a, b);}
        |       ^(NEG a=expr) {$node = new NegativeNode(a);}
        |       ^(POS a=expr) {$node = new PositiveNode(a);}
        |       ^(IINCR a=expr) {$node = new InfixIncrNode(a);}
        |       ^(IDECR a=expr) {$node = new InfixDecrNode(a);}
        |       ^(PINCR a=expr) {$node = new PostfixIncrNode(a);}
        |       ^(PDECR a=expr) {$node = new PostfixDecrNode(a);}
        |       ^(NOT a=expr) {$node = new NotNode(a);}
        |       ^(EQ a=expr b=expr) {$node = new EqualNode(a, b);}
        |       ^(NEQ a=expr b=expr) {$node = new NotEqualNode(a, b);}
        |       ^(LE a=expr b=expr) {$node = new LessNode(a, b);}
        |       ^(LEQ a=expr b=expr) {$node = new LessEqualNode(a, b);}
        |       ^(GT a=expr b=expr) {$node = new GreaterNode(a, b);}
        |       ^(GEQ a=expr b=expr) {$node = new GreaterEqualNode(a, b);}
        |       ^(AND a=expr b=expr) {$node = new AndNode(a, b);}
        |       ^(OR a=expr b=expr) {$node = new OrNode(a, b);}
        |       ^(PAR a=expr) {$node = new ParenthesisNode(a);}
        |       ^(INTCAST a=expr) {$node = new IntCastNode(a);}
```

```
|       ^(LONGCAST a=expr) {$node = new LongCastNode(a);}
|       ^(FLOATCAST a=expr) {$node = new FloatCastNode(a);}
|       ^(DOUBLECAST a=expr) {$node = new DoubleCastNode(a);}
|       ^(BOOLCAST a=expr) {$node = new BooleanCastNode(a);}
|       ^(CHARCAST a=expr) {$node = new CharCastNode(a);}
|       INT {$node = new IntNode(Integer.parseInt($INT.text));}
|       LONG {$node = new LongNode(Long.parseLong(
        ($LONG.text).substring(0,($LONG.text).length()-1)));}
|       FLOAT {$node = new FloatNode(Float.parseFloat($FLOAT.text));}
|       DOUBLE {$node = new DoubleNode(Double.parseDouble($DOUBLE.text));}
|       BOOL {$node = new BooleanNode(Boolean.parseBoolean($BOOL.text));}
|       CHAR {$node = new CharNode(($CHAR.text).charAt(1));}
|       STRING {$node = new StringNode(
        ($STRING.text).substring(1, ($STRING.text).length()-1));}
|       ID {$node = new IdentifierNode($ID.text);}
;
```

A.3 Klassendiagramm für Knoten



Literaturverzeichnis

[ALSU08] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compiler : Prinzipien, Techniken und Werkzeuge*. Pearson Studium, 2008.

[Dok] BlueJ 3.1.0 Sourcecode Dokumentation. Bluej-architecture-and-design.txt.

[EP08] Katrin Erk and Lutz Priese. *Theoretische Informatik : Eine umfassende Einführung*. Springer-Verlag, 2008.

[GHJV12] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Entwurfsmuster. Elemente wiederverwendbarer objektorientierter Software*. Addison-Wesley Verlag, Pearson Education, 2012.

[GJ90] Dick Grune and Ceriel Jacobs. *Parsing Techniques*. Ellis Horwood Limited, 1990.

[GJS+13] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. The Java Language Specification. Java SE 7 Edition. `http://docs.oracle.com/javase/specs/jls/se7/html/index.html`, 2013. [Online; Letzter Zugriff 30-Nov-2013].

[ISO96] Information technology - Syntactic metalanguage - Extended BNF. Number ISO/IEC 14977 : 1996(E). ISO, Geneva, Switzerland, 1996.

[KB13] Michael Kölling and David J. Barnes. *Java lernen mit BlueJ: eine Einführung in die objektorientierte Programmierung*. Pearson, 2013.

[Köl13] Michael Kölling. BlueJ – Source download. `http://bluej.org/download/source-download.html`, 2013. [Online; Letzter Zugriff 8-Aug-2013].

[Neb12] Markus E. Nebel. *Formale Grundlagen der Programmierung*. Vieweg+Teubner Verlag, 2012.

[O+13] J. O'Madadhain et al. JUNG - the Java Universal Network/Graph Framework. `http://jung.sourceforge.net/`, 2013. [Online; Letzter Zugriff 8-Aug-2013].

[Par07] Terence Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. The Pragmatic Bookshelf, May 2007.

[Par13] Terence Parr. ANTLR Parser Generator, Homepage. `http://www.antlr.org/`, 2013. [Online; Letzter Zugriff 8-Aug-2013].

[Pot12] Max Potente. Ein Parser für Java Level 1 als Erweiterung BlueJs. Bachelorarbeit, Universität Hamburg, 2012.

[Seb12] Robert W. Sebesta. *Concepts of Programming Languages*. Pearson Education Canada, 2012.

[SS07] Christian Späh and Axel Schmolitzky. Consuming before Producing as a Helpful Metaphor in Teaching Object-Oriented Concepts. *Eleventh Workshop on Pedagogies and Tools for the Teaching and Learning of Object Oriented Concepts, ECOOP 2007*, Berlin, Germany, 2007.

[SZ+12] Axel Schmolitzky, Heinz Züllighoven, et al. Folien der SE1-Vorlesung, 2012.

[Zü05] Heinz Züllighoven. *Object-oriented construction handbook : developing application-oriented software with the tools and materials approach*. dpunkt, 2005.

Erklärung

Ich versichere, dass ich die vorstehende Arbeit selbstständig und ohne fremde Hilfe angefertigt und mich anderer als der im beigefügten Verzeichnis angegebenen Hilfsmittel nicht bedient habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht.

Ich bin mit einer Einstellung in den Bestand der Bibliothek des Fachbereiches einverstanden.

Hamburg, den _____ Unterschrift: _____