Modellierung und Programmierung einer Stackmaschine mit grafischer Oberfläche, welche einfache mathematische Operationen ausführen kann und somit über eine Makrosprache verfügt, eines zugehörigen lexikalischen Scanners, eines syntaktischen Parsers und eines Compilers (Übersetzer und Interpreter) zur schrittweisen Überführung eines korrekt geklammerten, arithmetischen Ausdrucks in die Stackmaschinensprache.

Informatik

Fachlehrer: Herr Faßbender

Kurs: LK IF7

Schuljahr: 12

Erik Springer

Inhaltsverzeichnis

1 Einleitung

Im Informatik Unterricht der Q1 haben wir bereits reguläre Sprachen und endliche Automaten behandelt. Außerdem lernten wir einfache Scanner und Parser für reguläre Sprachen oder endliche Automaten zu implementieren. Dadurch haben wir nur einen Teilbereich des Compilerbaus behandelt. Die interessantere Anwendung des Compilerbaus geht jedoch über den Lehrstoff hinaus.
In dieser besonderen Lernleistung wird die Funktionsweise eines Compilers verdeutlicht. Dafür wurde eine Stackmaschine mit allen zugehörigen Elementen eines Interpreters implementiert. Die Stackmaschine berechnet anhand von bestimmten Steuerbefehlen einen korrekten arithmetischen Ausdruck. Bei dieser Stackmaschine handelt es sich also um einen Interpreter. Dieser ist einem Compiler sehr ähnlich. Allerdings erstellt er aus dem Quelltext, im Gegensatz zu einem Compiler oder Assembler, keine direkt ausführbare Datei[1]. Stattdessen liest der Interpreter den Quellcode ein, analysiert diesen und führt ihn anschließend aus. Er übersetzt den Quellcode also zur Laufzeit des Programms.
Um diesen arithmetischen Ausdruck, welcher sozusagen der Quellcode ist, auf Korrektheit zu überprüfen, wird zunächst ein lexikalischer Scanner den Ausdruck in eine Liste aus Token umwandeln und gleichzeitig den Term auf unerlaubte Terminalsymbole prüfen. Ein Parser führt anschließend eine syntaktische und semantische Analyse durch. Hier wird zum Beispiel darauf geachtet, dass alle Klammern, die auf gehen, auch wieder zu gehen, oder dass keine zwei Rechenoperatoren nebeneinander stehen. Ist der Ausdruck syntaktisch korrekt, wird die vom Scanner erstellte Tokenliste, in eine umgekehrte polnische Notation(kurz: UPN) umgewandelt. Dies kann durch den Shunting-Yard-Algorithmus umgesetzt werden. Eine Alternative ist die Erstellung eines Syntaxbaumes im Parser, welcher anschließend von einem Übersetzer in einer Postorder Traversierung ausgelesen wird. Die UPN wird dann in Steuerbefehle für die Stackmaschine, also in die Stackmaschinensprache übersetzt. Die Stackmaschine führt die Steuerbefehle aus. Dieser Vorgang soll auch in einer graphischen Benutzeroberfläche dargestellt werden.

Zielsetzungen dieser Lernleistung sind folgende:

- Modellierung und Programmierung einer Stackmaschine mit grafischer Oberfläche
- Implementation eines lexikalischen Scanners
- Implementation eines syntaktischen Parsers
- Implementation eines Übersetzer
- Implementation einer graphischen Benutzeroberfläche zur Darstellung der schrittweisen Überführung des Ausdrucks in Steuerbefehle für die Stackmaschine

In Abbildung 1 wird der Aufbau eines Compilers nochmal zusammengefasst.

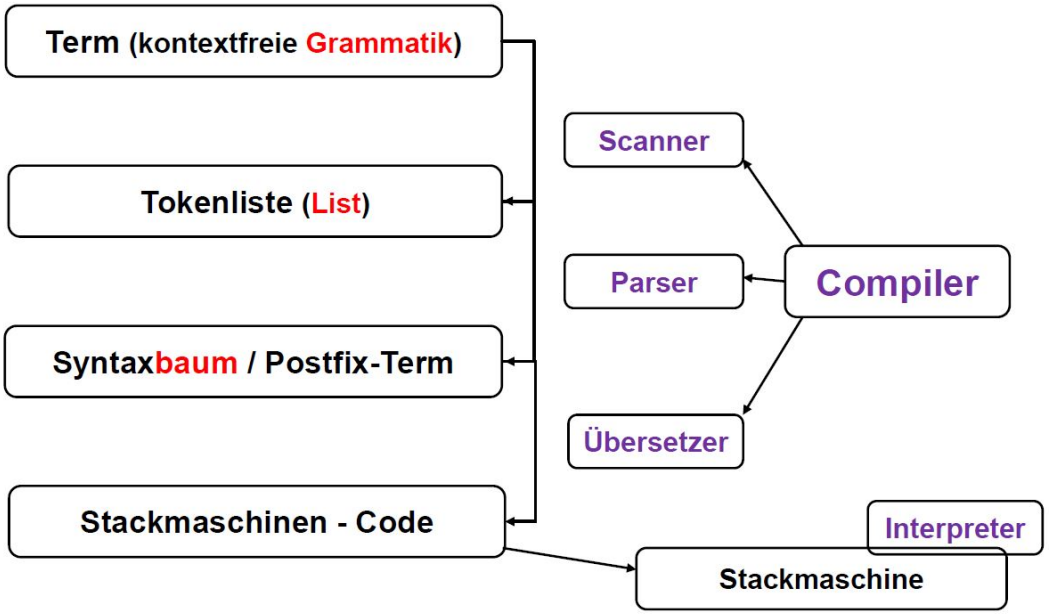

Abbildung 1: Aufbau eines Compilers [2]

Abschnitt 2 erläutert die lexikalische Analyse des Scanners und wie diese zu implementieren ist. Abschnitt 3, beschreibt wie der Parser bei der syntaktischen Analyse vorgeht und gleichzeitig einen semantisch verwertbaren Syntaxbaum aufbaut. Abschnitt 4 beschreibt wie der Übersetzer die Steuerbefehle für die Stackmaschine, durch den vom Parser erzeugten Syntaxbaum, erzeugt. Zum Schluss zeigt Abschnitt 5 wie diese Steuerbefehle verarbeitet werden und die dazu gehörige graphische Benutzeroberfläche.

Für dieses Projekt wird die objektorientierte Programmiersprache Java verwendet, welche 1995 erschienen ist und von Sun Microsystems entwickelt wurde[3]. Das Unternehmen wurde 2010 von Oracle aufgekauft. Java gehört zur Java-Technologie, welche eine Sammlung verschiedener Laufzeitumgebungen ist.
Das Besondere an Java ist, dass im Gegensatz zu anderen Programmiersprachen, der Maschinencode nicht für ein bestimmtes Betriebssystem ist. Der Quelltext wird in einen Bytecode kompiliert, welcher nur von der Java virtual Machine(JVM) ausgeführt werden kann.
Für die Erstellung des Programms wurde die speziell für Java integrierte Entwicklungsumgebung BlueJ benutzt. BlueJ visualisiert Klassen als verkürztes Klassendiagramm. Darüber hinaus kann man auch Quelltext bearbeiten und kompilieren[4].
Zur Erstellung der graphischen Benutzeroberfläche wird NetBeans IDE(integrated development environment) verwendet, welches auch hauptsächlich für Java entwickelt wurde[5]. NetBeans ist sehr beliebt für die Modellierung von Benutzeroberflächen, da entsprechende Frameworks integriert sind.

2 Scanner - Lexikalische Analyse

Der lexikalische Scanner, auch Tokenizer genannt, ist im Compilerbau dafür zuständig einen Quelltext zu durchlaufen und dabei in logisch zusammengehörende Einheiten zu zerlegen. Diese Einheiten werden als Token bezeichnet. Ein Token ist eine Zeichenkette, da auch mehrere Zeichen im Zuge einer lexikalischen Analyse zusammengefasst werden können. Der Inhalt der Token ist dabei gleichzusetzen mit den Terminalsymbolen der Grammatik des Parsers. Liest der Scanner also ein Zeichen ein, welches nicht zu einem Terminalsymbol der Grammatik gehört, wird ein lexikalischer Fehler gemeldet. Tritt kein Fehler auf, wird anschließend diese Folge dem Parser überreicht.

Der Scanner hat die Aufgabe eine Tokenliste zu erstellen. Hierfür benötigen wir zunächst eine Klasse Token. Dieser Klasse wird ein String im Konstruktor übergeben, der bestimmt, welche Eigenschaften das Token hat. Damit die Klasse Token bei späterer Verwendung nützlich ist, bekommt sie mehrere Booleans, wie zum Beispiel: ”istZahl”, ”istKlammerAuf” oder ”istADD”. Damit kann leicht abgefragt werden, um welchen Typ eines Tokens es sich handelt. Im Konstruktor werden mit den Hilfsmethoden alle diese Werte gesetzt. Es gibt auch einen leeren Konstruktor, welcher lediglich den Boolean ”istLeer” auf true setzt. Abbildung 2 zeigt eine Implementationsdiagramm der Klasse Token.

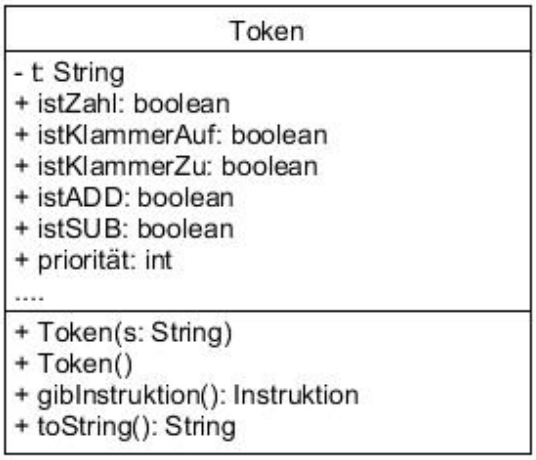

Abbildung 2: UML-Diagramm der Klasse Token

Der Scanner hat eine statische Methode scanne(String pWort), welche eine Liste aus Token zurück gibt. Der Eingabestring wird dabei mit einer while-Schleife durchlaufen und einzelne Zeichen werden abgearbeitet.
Durch die Methode substring(1) der Klasse String kann man den String um ein Zeichen verkürzen. Wenn es sich zum Beispiel um ein ”+” handelt, wird der Tokenliste ein neues Token, mit ”tokenliste.append(new Token(”+”))” angehängt und ”pWort” um ein Zeichen verkürzt.

Die Methode ”istZahl(String pWort)” der Klasse Hilfsmethode, überprüft, ob es sich beim ersten Zeichen des Eingabestrings, um eine Ziffer handelt.
Der Scanner soll auch Fließkommazahlen erkennen und zu einem Token zusammensetzen. Das ist normalerweise Aufgabe der syntaktischen Analyse. Es ist jedoch effektiver dies bereits im Scanner zu erledigen. Andernfalls müsste man die Grammatik der arithmetischen Ausdrücken erweitern und anschließend mehrere Token einer Fließkommazahl im Parser zu einer Zahl zusammensetzen. Um dies im Scanner umzusetzen, wird beim Erkennen einer Zahl am Anfang des Eingabestrings ein Zähler auf 1 gesetzt. Dieser Zähler gibt an, wie viele Stellen als ein Token zusammengefasst werden sollen. Um auch Zahlen mit mehr als einer Ziffer zu erkennen, wird erst eine while-Schleife im Eingabestring solange durchlaufen, bis etwas anderes als eine Zahl kommt. Danach wird abgefragt, ob ein Punkt kommt. Ist dies der Fall, wird der Zähler weiter gezählt, bis wieder etwas anderes als ein Punkt oder Komma gelesen wird. Anschließend muss nur noch ein Token mit dem String bis zur Stelle des Zählers der Tokenliste angehängt werden. Sollte nach dem Punkt keine Zahl eingelesen werden, dann wird ”null” als Fehlerwert zurückgegeben. Wird allgemein ein Eingabezeichen eingelesen, welches nicht in einem arithmetischen Ausdruck vorkommen darf, wird ”null” zurückgegeben.
Abbildung 3 stellt die lexikalische Analyse als Programmablaufplan dar.

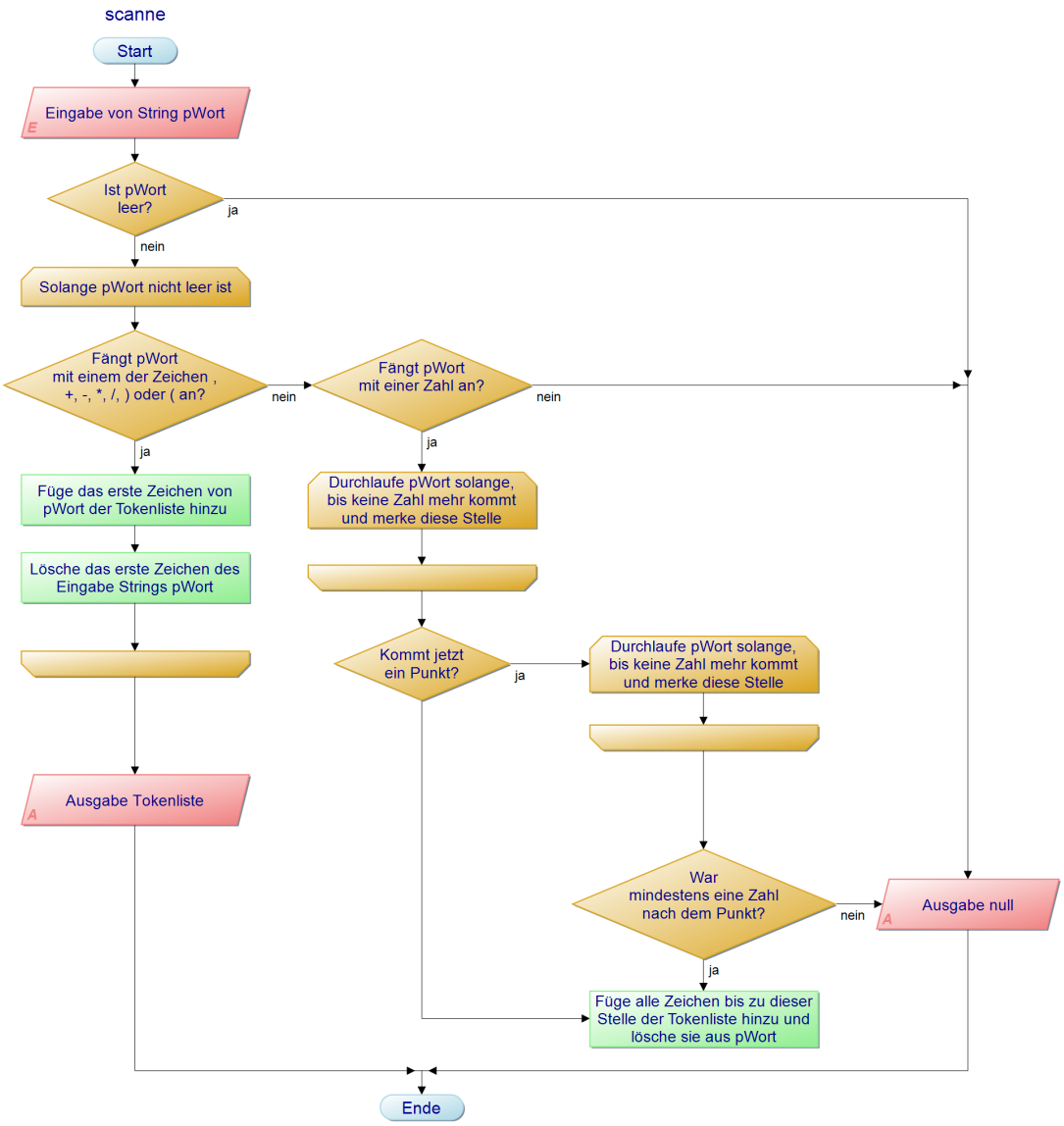

Abbildung 3: Programmablaufplan der Methode scanne

3 Parser

Der Parser hat die Aufgabe der Zerlegung und Weiterverarbeitung der Eingabe. Dafür überführt er die vom Scanner erstellte Tokenliste in ein geeignetes Format, wie zum Beispiel einen Syntaxbaum. Standardmäßig benutzt er eine eindeutige kontextfreie Grammatik für die syntaktische Analyse und gibt den während der Ableitung erstellten Syntaxbaum zurück. Dieser hilft im Anschluss die Semantik der Eingabe zu erschließen.
Abbildung 4 zeigt die Mengen aller eindeutigen kontextfreien Grammatiken. Eindeutige Grammatiken sind so definiert, dass die Herleitung eines Ausdrucks nur zu einem möglichen Syntaxbaum führen kann. Dabei wird unterschieden zwischen LR(k)-Grammatiken und den LL(K)-Grammatiken.

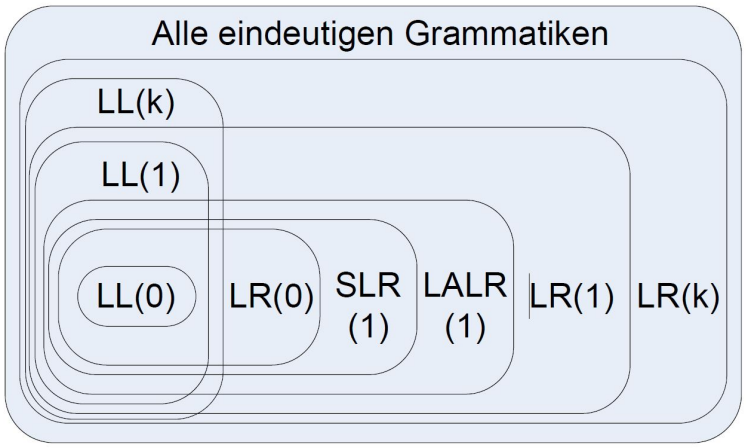

Abbildung 4: Menge der eindeutigen Grammatiken[6]

Eine LR(k)-Grammatik benutzt eine Anzahl von k Token die sie im Eingabewort maximal vorausschauen darf, um sich für eine Produktionsregel zu entscheiden. Außerdem wird die Eingabe von einem LR-Parser von links nach rechts abgearbeitet und bei der Produktion eine umgekehrten Rechtsableitung verfolgt. Der Unterschied zur LL(1)-Grammatik ist, dass diese eine Linksableitung vornimmt. Die Produktionsregeln der LL(1)-Grammatik haben speziellere Vorgaben. Somit ist sie nur eine Teilmenge aller LR(1)-Grammatiken. Links- und Rechtsableitung werden im Abschnitt 3.1 erklärt.
Je nach Grammatik werden unterschiedliche Implementierungstechniken verwendet. So kann eine Parser rekursiv absteigend, rekursiv aufsteigend oder tabellengesteuert sein. Für LR(k)-Grammatiken wird üblicherweise ein Shift-Reduce-Parser genutzt, welcher eine Bottom-Up-Analyse ausführt und dabei den Syntaxbaum von unten nach oben aufbaut. Der Nachteil dieser Parser ist, dass sie meistens aufwendig und fehleranfällig zu implementieren sind. LL(k)-Grammatiken können einfacher durch einen LL-Parser zum Beispiel mit rekursivem Abstieg umgesetzt werden. Er wird auch als Top-Down-Parser bezeichnet, da er den Syntaxbaum von oben nach unten aufbaut.

3.1 Syntax- und Ableitungsbäume

Ein Syntaxbaum, auch Ableitungs- oder Parsebaum genannt, ist in der theoretischen Informatik eine baumförmige Darstellung einer Ableitung[7]. Für jede Ableitung wird ein neuer Teilbaum mit einem Nichtterminal eingefügt. Wird auf ein Terminalsymbol produziert, handelt es sich um ein Blatt, da von dort aus keine weitere Ableitung mehr stattfinden kann.
Für eindeutige Sprachen gibt es nur einen Syntaxbaum. Dabei ist es egal, ob ein Wort der Sprache durch eine Links- oder Rechtsableitung hergeleitet wurde, der Baum stellt beide Ableitungen dar. Bei einer Linksableitung wird das linke Nichtterminalsymbol zuerst abgeleitet. Bei einer Rechtsableitung ist das anders herum[8]. Abbildung 5 zeigt eine Beispiel für eine Grammatik G und den zugehörigen Syntaxbaum für das Wort ”aab”. Liest man bei einem Syntaxbaum nur die Blätter von links nach rechts ab, erhält man das Wort.

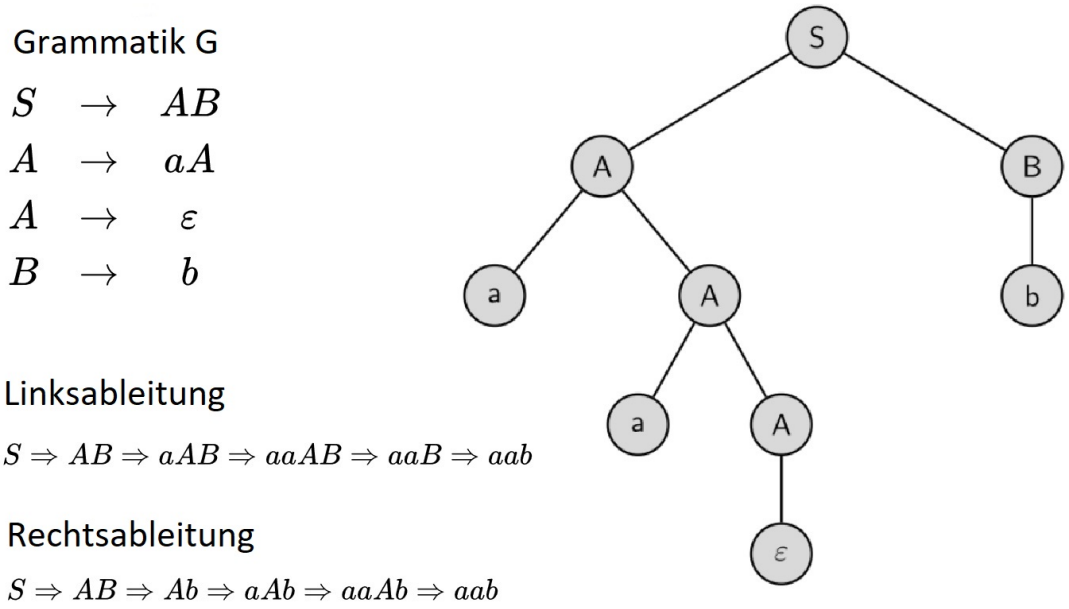

Abbildung 5: Beispiel für einen Syntaxbaum mit der Grammatik G[7]

In Abbildung 6 ist ein möglicher Syntaxbaum für die Sprache der korrekt geklammerten arithmetischen Ausdrücke dargestellt. Dieser ist mit einem Rechenbaum gleichzusetzen. Ersetzt man die Nichtterminale mit den Terminalen in den Blättern, erhält man als vereinfachte Darstellung den Rechenbaum.

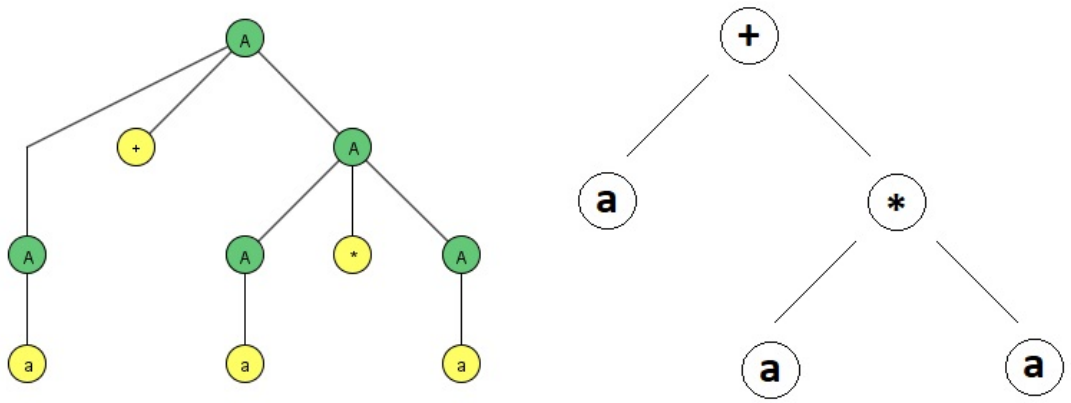

Abbildung 6: Beispiel für einen Syntaxbaum eines arithmetischen Ausdrucks und eines Rechenbaumes

3.2 Syntaktische Analyse und formale Sprachen

Um eine syntaktische Analyse durchzuführen, benötigen wir eine kontextfreie Grammatik, welche die Wörter der Sprache der korrekt geklammerten arithmetischen Ausdrücke erstellen kann. Die folgende Grammatik kann diese Sprache beschreiben:

Nichtterminalsymbole N = {Ausdruck}
Alphabet Σ = {a, (,), *, +, -, /}
Startsymbol S = Ausdruck

(1) Ausdruck → Ausdruck + Ausdruck
(2) Ausdruck → Ausdruck - Ausdruck
(3) Ausdruck → Ausdruck * Ausdruck
(4) Ausdruck → Ausdruck / Ausdruck
(5) Ausdruck → (Ausdruck)
(6) Ausdruck → a

Das Terminal ”a” steht dabei für eine beliebige positive Zahl.
Das Problem dieser Grammatik ist, dass sie nicht eindeutig ist. Will man zum Beispiel den Ausdruck ”a + a * a” von links herleiten, gibt es mehrere Möglichkeiten:

Möglichkeit 1:
Ausdruck (1) → Ausdruck + Ausdruck (6) → a + Ausdruck
(3) → a + Ausdruck * Ausdruck (6)(6) → a + a * a

Möglichkeit 2:
Ausdruck (3) → Ausdruck * Ausdruck (1) → Ausdruck + Ausdruck * Ausdruck
(6)(6)(6) → a + a * a

Dabei entstehen, wie Abbildung 7 zeigt, auch zwei unterschiedliche Syntaxbäume.

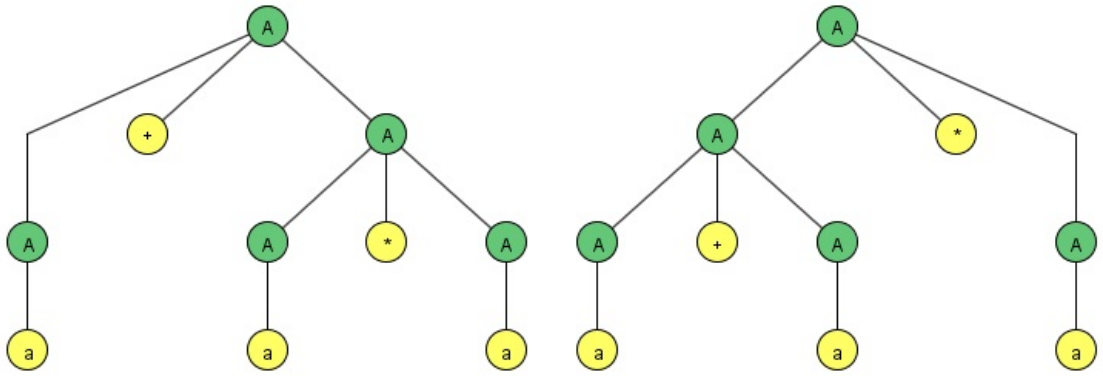

Abbildung 7: Zwei unterschiedliche Syntaxbäume für das selbe Wort(mit JFlap erstellt)

Die Syntaxbäume haben unterschiedliche Semantik(”a+a*a” und ”(a+a)*a”). Nur der Linke der beiden Syntaxbäume verfügt über die gewünschte Semantik, denn es gilt: Punktrechnung vor Strichrechnung. Überführt man den Syntaxbaum in einen Rechenbaum, dann fällt auf, dass der untere Teilbaum zuerst berechnet werden muss, um den oberen zu berechnen. Für den den linken Syntaxbaum würde das bedeuten, wir rechnen erst ”a*a” und addieren dann das daraus resultierende Ergebnis mit a. Beim rechten Syntaxbaum wäre es verkehrt herum und wir hätten die Rechenregel Punkt vor Strich missachtet.
Das bedeutet also, dass eine Grammatik beim Parsen nicht mehrdeutig sein darf. Wir benötigen eine eindeutige Grammatik.
Bei der folgenden Grammatik handelt es sich um eine LR(1)-Grammatik. Diese sind immer eindeutig.

Nichtterminalsymbole N = {Ausdruck, Summand, Faktor, Ziffer}
Alphabet Σ = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, (,), *, +, -, /}
Startsymbol S = Ausdruck

(1) Ausdruck → Summand
(2) Ausdruck → Ausdruck + Summand
(3) Ausdruck → Ausdruck - Summand
(4) Summand → Faktor
(5) Summand → Summand * Faktor
(6) Summand → Summand / Faktor
(7) Faktor → Ziffer
(8) Faktor → (Ausdruck)
(9) Ziffer → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 1 Ziffer | 2 Ziffer | ... | 9 Ziffer

Da Ziffer auch auf Fließkommazahlen abgeleitet werden soll, müsste man die Produkti-

onsregeln folgendermaßen erweitern:

Ziffer $\rightarrow$ 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 1 Zahl | 2 Zahl | ... | 9 Zahl
Zahl $\rightarrow$ 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0 Zahl | 1 Zahl | ... | 9 Zahl | . Komma
Komma $\rightarrow$ 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0 Komma | 1 Komma | ... | 9 Komma

Da es allerdings für die syntaktische Korrektheit keine Rolle spielt, ob man eine Zahl oder Fließkommazahl an einer bestimmten Stelle hat, ist diese Erweiterung umständlich und unnötig. Es ist daher besser, wenn man diesen Teil der syntaktischen Analyse, wie oben bereits beschrieben, im Scanner erledigt. Auch werden allgemein längere Zahlen vom Scanner als ein Token zusammengefasst, was uns die Implementation des Parsers viel einfacher macht.
Diese Grammatik ist eindeutig, dass es für jedes Wort nur eine mögliche Links- oder Rechtsableitung gibt. Die Linksableitung für das Wort ”a+a*a” würde so aussehen:
Ausdruck (2) $\rightarrow$ Ausdruck + Summand (1) $\rightarrow$ Summand + Summand
(4) $\rightarrow$ Faktor + Summand (9) $\rightarrow$ a + Summand(5) $\rightarrow$ a + Summand * Faktor
(4) $\rightarrow$ a + Faktor * Faktor (9) $\rightarrow$ a + a * Faktor (9) $\rightarrow$ a + a * a

Da die Produktion ”Summand $\rightarrow$ Summand * Faktor” später ausgeführt wird, als die Produktion ”Ausdruck $\rightarrow$ Ausdruck + Summand”, muss die Multiplikation im Syntaxbaum in einem tieferen Teilbaum sein als die Addition. Damit ist der einzig möglich Syntaxbaum auch semantisch korrekt. Das gleiche gilt auch für Subtraktion und Division.

3.2.1 Rekursive Abstieg

Der rekursive Abstieg ist eine Technik aus dem Compilerbau, welche von LL-Parsern verwendet wird, um eine LL(k)-Grammtik zu implementieren. Dabei wird für jedes Nichtterminalsymbol eine Prozedur definiert, welche das Eingabewort entsprechend der Produktionen durchläuft. Kann an einer Stelle keine Produktion mehr angewandt werden, so bricht der Vorgang an dieser Stelle ab und wir kennen somit die Fehlerstelle. Die oben angegebene LR(1)-Sprache führt bei der Implementation zu einem Problem, denn der rekursive Abstieg ist nur für LL(k)-Grammatiken gedacht. Eine Produktion wie zum Beispiel ”Ausdruck $\rightarrow$ Ausdruck + Summand” führt bei einer Rekursion zu einer Endlosschleife, da die Prozedur sich am Anfang immer wieder selber rekursiv aufrufen würde. Wir müssen die gegebene LR(1)-Grammatik also in eine LL(1)-Grammatik umwandeln. Für eine LL(1)-Grammatik gelten folgende Regeln:

1. Die Grammatik darf nicht mehrdeutig sein
2. Die Grammatik darf keine Linksrekursion enthalten
3. Die Grammatik muss linksfaktorisiert sein

Die gegebene Grammatik ist nicht mehrdeutig. Allerdings enthält sie zum Beispiel in der Produktion ”Ausdruck → Ausdruck + Summand” Linksrekursion. Um Linksrekursion aus einer Grammatik zu entfernen, gibt es bestimmte Algorithmen. Da es sich hier aber um eine einfache Grammatik handelt, reicht es, wenn die linksrekursiven Produktionen umgedreht werden. Die daraus entstandene Grammatik beschreibt weiterhin dieselbe Sprache. Durch das Umdrehen wird die Grammatik auch gleichzeitig linksfaktorisiert. Das bedeutet, dass Nichtterminalsymbole, welche in einer Produktion alleine stehen, allerdings auch in anderen Produktionsregeln vorhanden sind, am Anfang jeder dieser Produktionsregeln stehen. Für die Produktionen von Ausdruck heißt das zum Beispiel, dass Summand immer am Anfang in allen Produktionen von Ausdruck stehen muss.
Die folgende Grammatik[9] erfüllt nun diese Voraussetzungen und kann mit dem rekursiven Abstieg umgesetzt werden.

(1) Ausdruck → Summand
(2) Ausdruck → Summand + Ausdruck
(3) Ausdruck → Summand - Ausdruck
(4) Summand → Faktor
(5) Summand → Faktor * Summand
(6) Summand → Faktor / Summand
(7) Faktor → (Ausdruck)
(8) Faktor → a

Es gibt eine Methode für jedes Nichtterminalsymbol. Die Methode pruefeAusdruck() beschreibt alle Produktion von dem Nichtterminalsymbol Ausdruck. Diese Methode sollte in der Tokenliste alle Terminale abarbeiten, welche mit den Produktionsregeln von ”Summand” erzeugt werden können. Alle Methoden liefern den Wert ”false” zurück, falls die jeweilige Produktionen fehlgeschlagen ist. Dabei ruft man zuerst die Methode pruefeSummand() auf. Gibt diese ”false” zurück, müssen wir abbrechen, also auch ”false” zurückgeben. Sollte die Liste jetzt immer noch Zugang haben, und der Rückgabewert war ”true”, hat mindestens die Produktion ”Ausdruck → Summand” funktioniert. Die Produktion ist aber noch nicht abgeschlossen. Nun könnten noch Produktionsregel 2 oder 3 möglich sein. Dafür muss das aktuelle Token ein ”+” oder ”-” sein. Ist dies der Fall, laufen wir in der Liste eins weiter. Falls ein syntaktischer Fehler auftritt, sagt uns der Integer s an welcher Stelle der Fehler aufgetreten ist. Da wir nun die Produktion ”Ausdruck → Summand + Ausdruck” oder ”Ausdruck → Summand - Ausdruck” haben, müssen wir jetzt die Methode pruefeAusdruck() aufrufen. Auf dieselbe Weise funktioniert die Methode pruefeSummand(). Sollte allerdings nach der Produktion Ausdruck → Summand, das nächste Token weder + noch - sein, dann geben wir nur ”true” zurück.
Abbildung 8 zeigt einen Programmablaufplan für die Methode pruefeAusdruck(). Eine Liste hat ein aktuelles Element, welche auf jede beliebige Stelle der Liste zeigen kann.

Am Anfang wird dieses auf das erste Element der Liste gesetzt. Der Vorgang ”Betrachte nächstes Token” bedeutet bei der Implementierung, dass wir das nächste Element der Tokenliste mit ”tokenliste.next()” anschauen. Mit Abfrage ”Hat die Tokenliste Zugang” überprüfen wir, ob das aktuelle Element der Liste noch auf ein Teilelement der Liste oder auf ”null” zeigt.

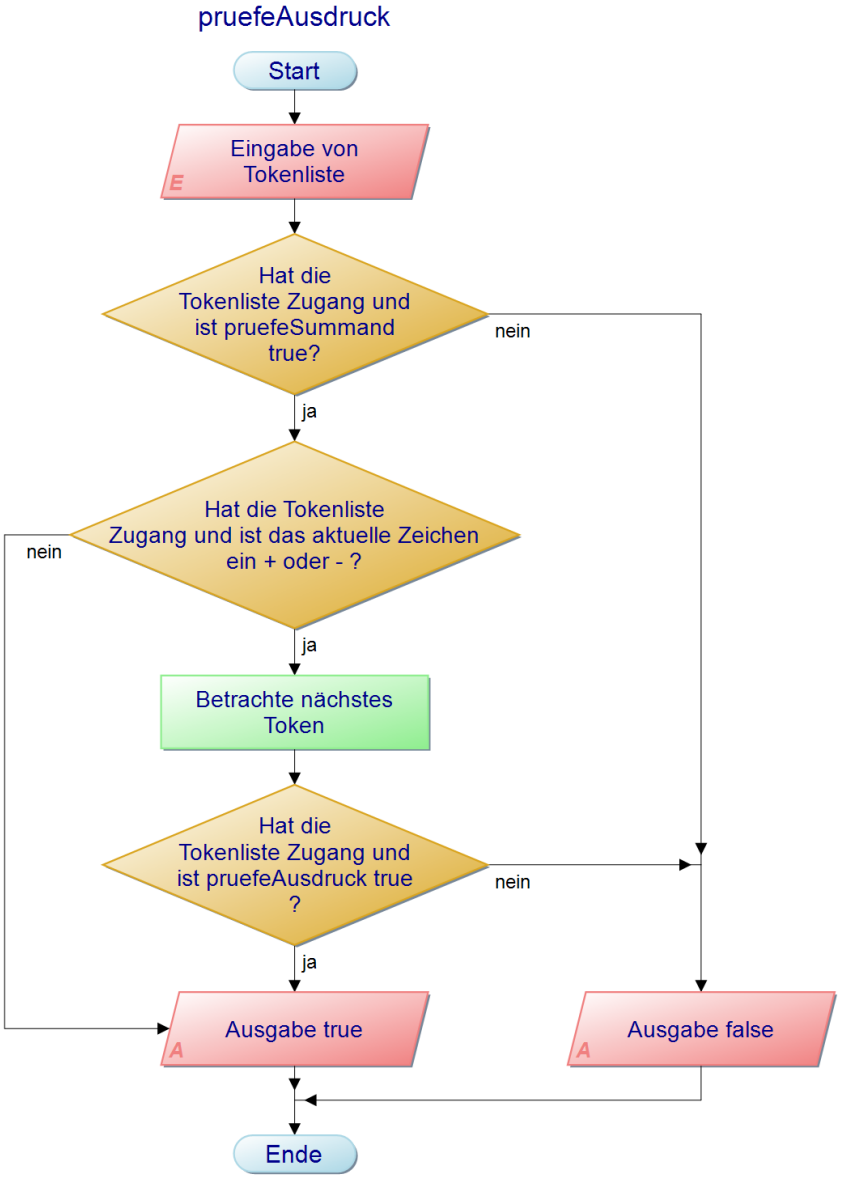

Abbildung 8: Programmablaufplan der Methode pruefeAusdruck()

Abbildung 9 zeigt einen Programmablaufplan für die Methode pruefeSummand(). Diese Methode implementiert die Produktionsregeln 4, 5 und 6.

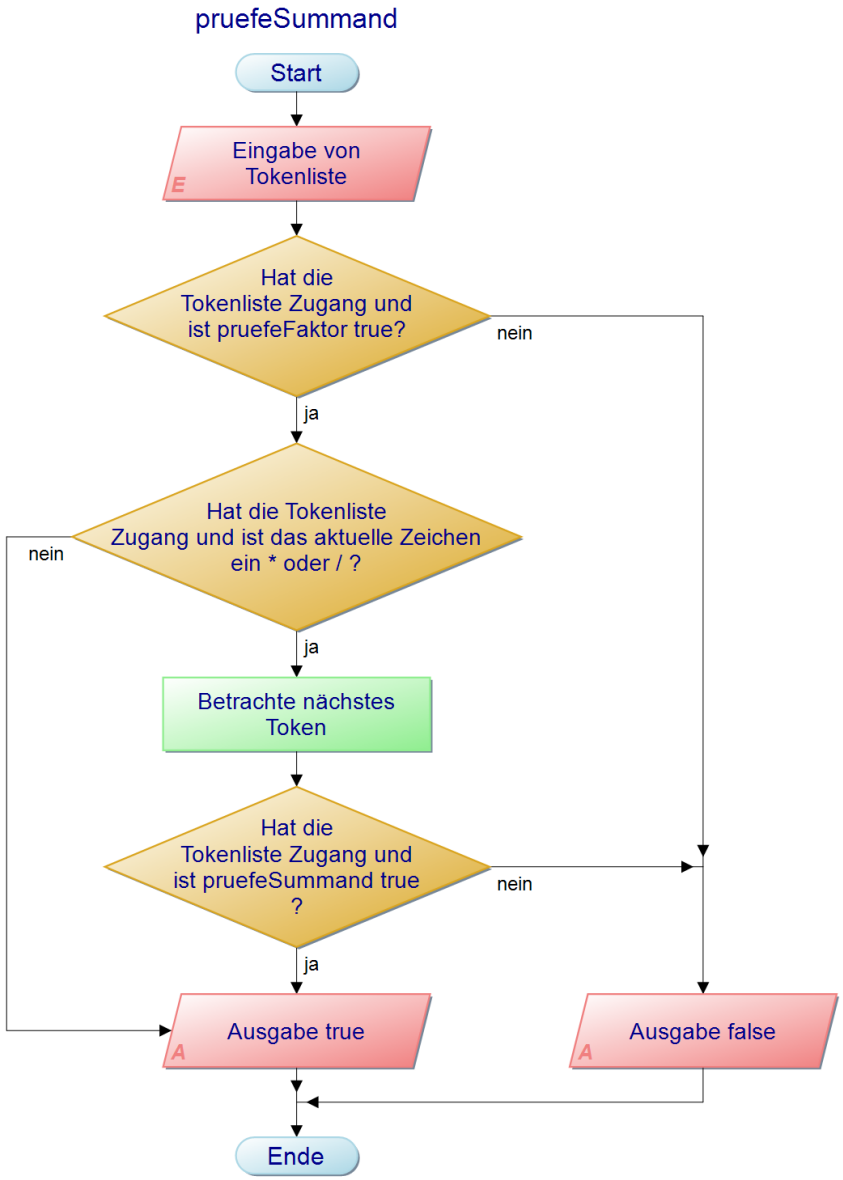

Abbildung 9: Programmablaufplan der Methode pruefeSummand()

Faktor kann entweder auf ”(Ausdruck)” oder ”a” abgeleitet werden. Beide Fälle müssen in der pruefeFaktor() Methode abgefragt werden. Ist das aktuelle Zeichen eine öffnende Klammer, dann wird anschließend die Methode pruefeAusdruck() aufgerufen. Sollte diese ”true” zurück liefern, wird überprüft, ob das aktuelle Zeichen nun eine schließende Klammer ist. Wenn das nicht der Fall ist, ist die Produktion fehlgeschlagen und wir geben ”false” zurück. Sollte das aktuelle Zeichen keine öffnende Klammer sein, darf es nur noch eine Zahl sein. Triff keins von beidem zu, liefert die Methode ”false” zurück.
Abbildung 10 verdeutlicht diese Prozedur als Programmablaufplan.

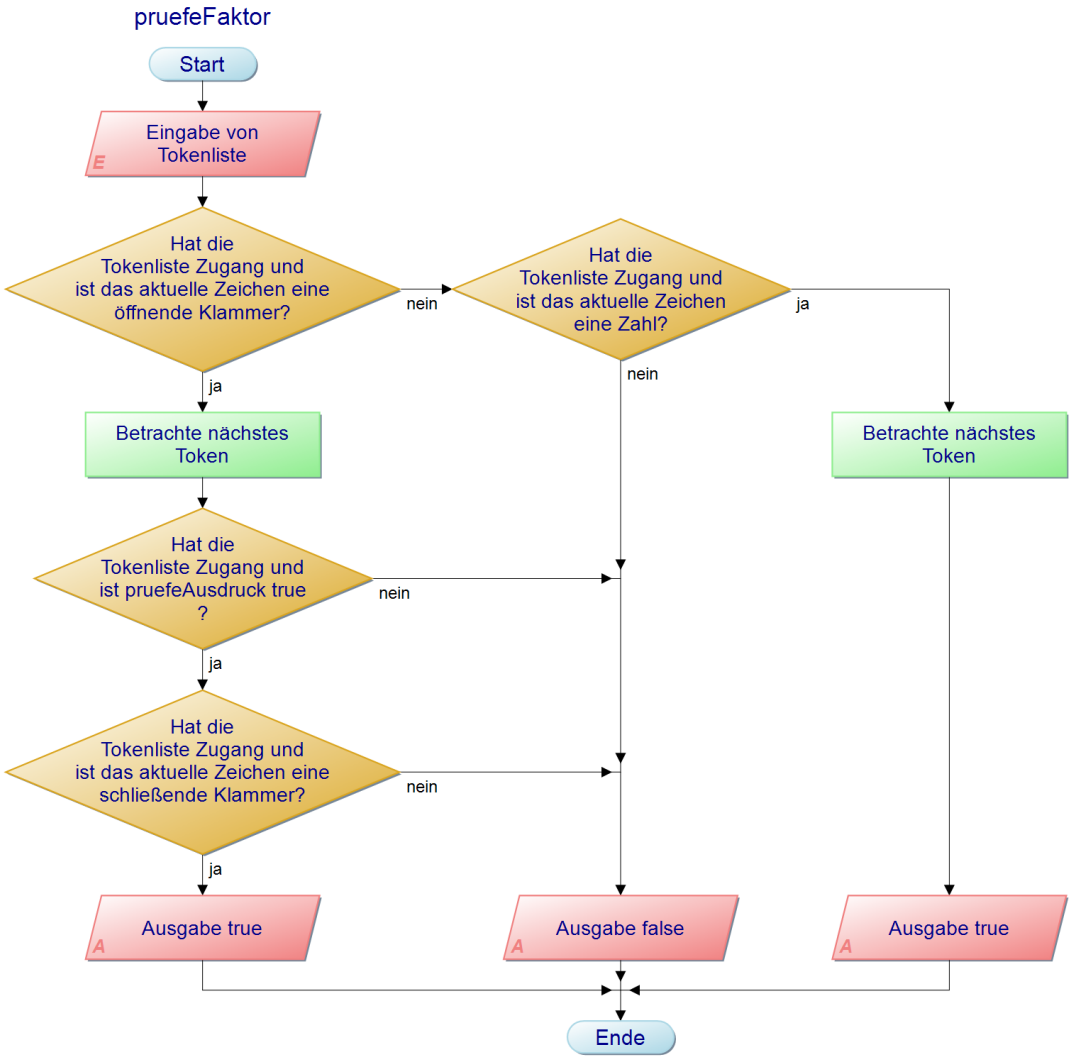

Abbildung 10: Programmablaufplan der Methode pruefeFaktor()

3.2.2 Kellerautomat

Die Sprache der korrekt geklammerten arithmetischen Ausdrücke kann auch als Menge der Wörter, die von einen Kellerautomaten erkannt werden, definiert sein. Es kann kein endlicher Automat für diese Sprache verwendet werden, da es sich um eine kontextfreie Sprache handelt. Das kommt daher, dass wir die Anzahl der öffnenden Klammer zählen müssen, um zu überprüfen, ob die gleiche Anzahl auch wieder zu geht. Ein endlicher Automat wäre dazu nicht in der Lage. Ein Kellerautomat hat einen zusätzlichen Kellerspeicher, normalerweise auch Kellerstapel genannt, welcher über ein Kelleralphabet verfügt. Die Übergänge in einem Kellerautomaten werden nicht nur vom aktuellen Zeichen des Eingabewortes bestimmt, sondern auch vom obersten Kellerzeichen. Beim Lesen des Kellerspeichers handelt es sich um ein sogenanntes zerstörendes Lesen, denn das oberste Kellerzeichen wird eingelesen und anschließend gelöscht. Je nach Übergang wird ein weiteres Kellerzeichen auf den Kellerspeicher gelegt. Eine Übergangsfunktion setzt sich also aus dem Einlesen des aktuellen Eingabezeichens, dem Löschen und Anschauen des obersten Kellerzeichen und dem Auflegen eines Kellerzeichens auf den Kellerstapel zusammen.
Der deterministische Kellerautomat, welcher in Abbildung 11 dargestellt ist, wurde mit dem Programm JFlap erstellt. Ein λ ist ein leeres Zeichen und bedeutet hier, dass beim Lesen des Kellerspeichers dieser Übergang unabhängig von dem aktuellen Kellerzeichen passiert. Also wird das oberste Kellerzeichen weder eingelesen, noch gelöscht.

Zustandsmenge Z = {q0, q1, q2}
Eingabealphabet Γ = {a, +, -, *, /, (,)}
Kelleralphabet Σ = {Z, (}
Endzustandsmenge F = {q2}

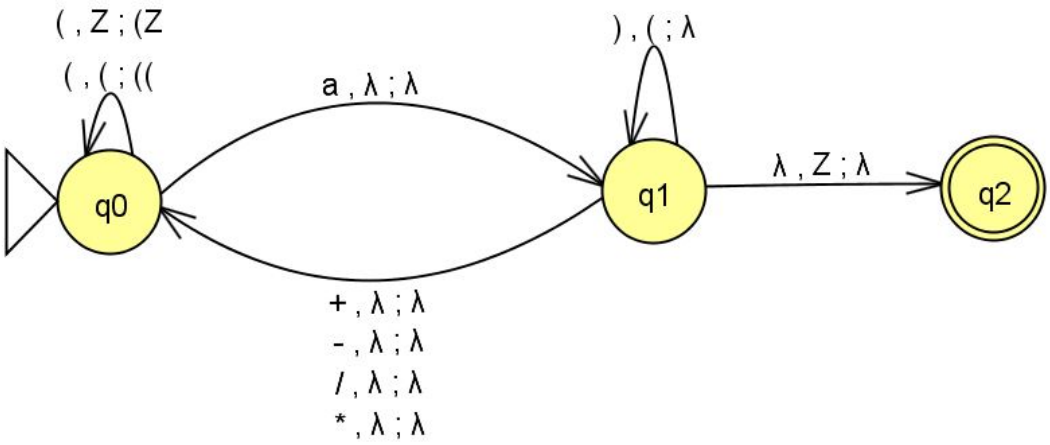

Abbildung 11: Kellerautomat

Der Kellerautomat kann auch, statt des rekursiven Abstiegs, zur syntaktischen Überprüfung eines arithmetischen Ausdrucks verwendet werden. Bei der Implementation des Kellerautomaten wurde kein Stapel verwendet, da wir mit ”(” nur ein Kellerzeichen haben. Stattdessen zählt jetzt ein Integer die Anzahl der öffnenden Klammern, welche auf dem Stack liegen müssten. In Abbildung 12 ist ein Programmablaufplan zur Implementation des Kellerautomaten.

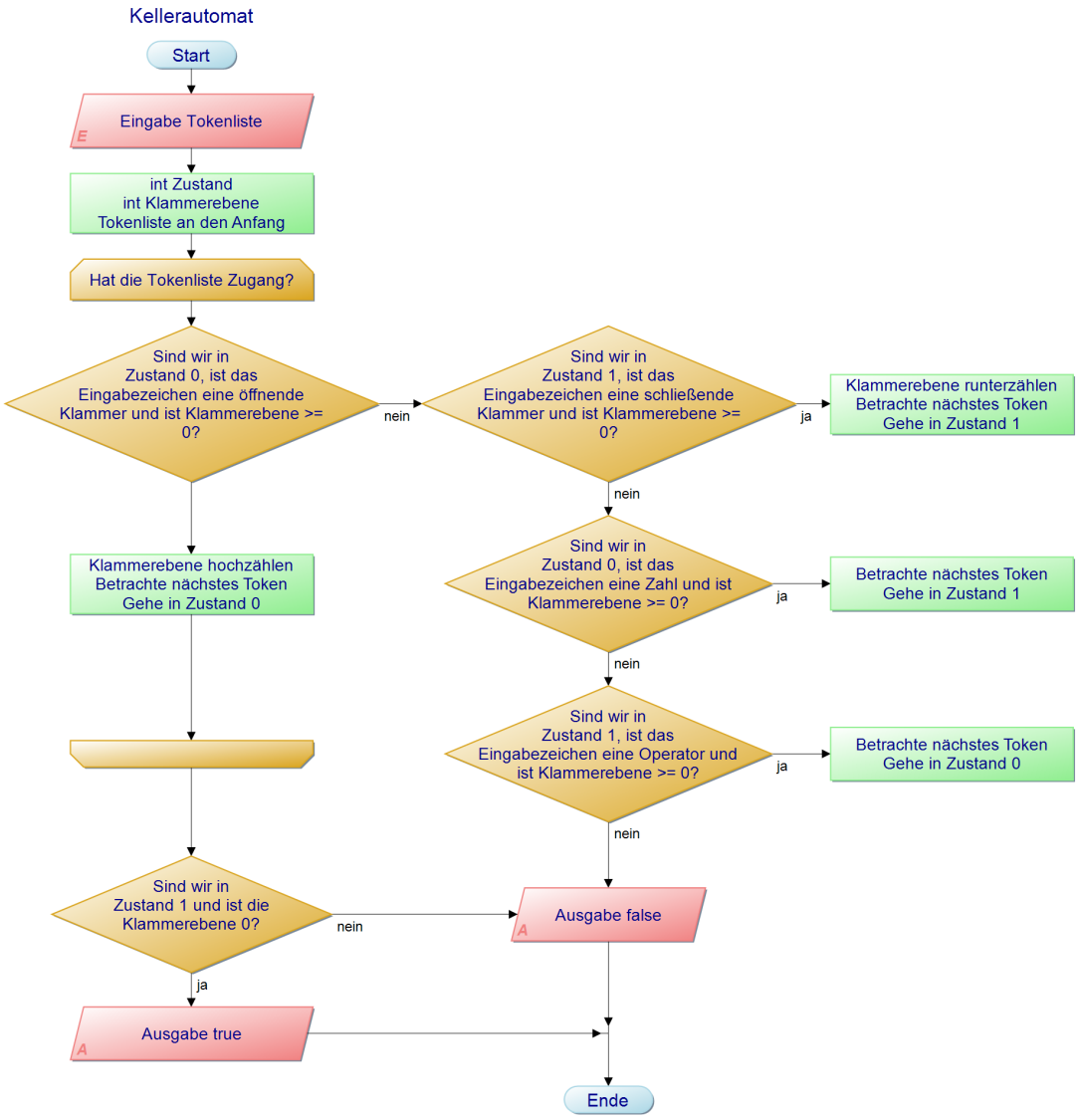

Abbildung 12: Programmablaufplan zur Implementation des Kellerautomaten

3.3 Semantische Analyse - Umwandlung in ein UPN

Die semantische Analyse eines Compilers folgt nach der syntaktischen Analyse. Hier wird überprüft, ob der Quelltext, welcher syntaktisch korrekt ist, auch semantisch korrekt ist. Bei einer Programmiersprache wäre ein semantischer Fehler zum Beispiel, wenn eine Variable vorher nicht definiert wurde.[10] Auch werden bestimmte Knoten des Syntaxbaumes mit Attributen versehen, welche Informationen zusammenfassen.

In unserem Fall kann man nicht wirklich von einer semantischen Analyse sprechen. Dennoch gibt der Syntaxbaum die Semantik der Eingabe wieder, da dieser später in einer Post-Order Traversierung ausgelesen wird, um eine umgekehrt polnische Notation zu erhalten. Es ist also wichtig, dass der Syntaxbaum, welcher aus der syntaktische Analyse hervorgeht, auch dem entspricht, was sich der Nutzer bei der Eingabe gedacht hat. Nehmen wir beispielsweise die Eingabe ”5-4-3”. Betrachtet man den zugehörigen Syntax- und Rechenbaum, welcher durch den rekursiven Abstieg also die LL(1)-Grammatik hervorgeht, würde er wie in Abbildung 13 aussehen.

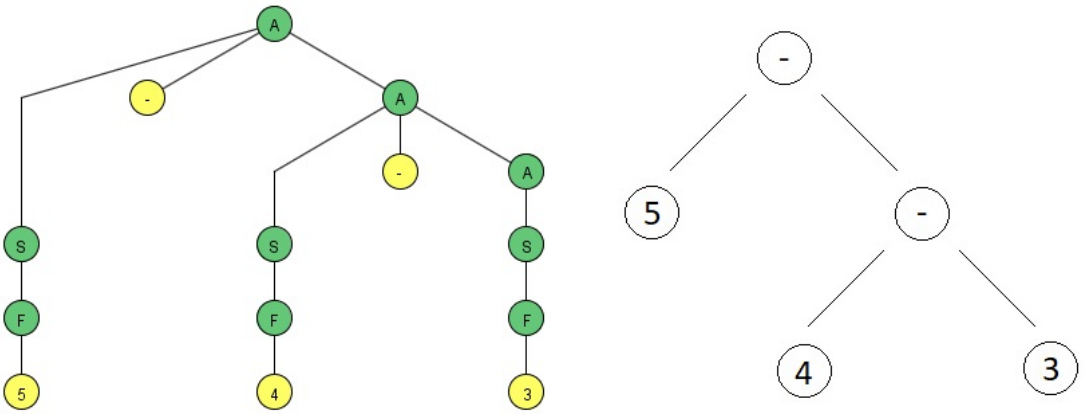

Abbildung 13: Syntaxbaum und Rechenbaum der LL(1)-Grammatik für 5-4-3

Der Rechenbaum gibt vor, erst ”4-3” zu berechnen und dann ”5-1”. Also beschreibt der Rechenbaum den Ausdruck ”5-(4-3)”. Das ist ein semantischer Fehler. Für arithmetische Ausdrücke gilt das Assoziativgesetz, welches vorschreibt, dass die Operatoren +, -, * und / von links nach rechts berechnet werden. Dieses Problem kommt daher, dass wir durch die Umwandlung der LR(1)-Grammatik in eine LL(1)-Grammatik die Linksrekursion entfernt haben. An der Herleitung des Ausdrucks ”5-4-3” in Abbildung wird deutlich, was damit gemeint ist:

Herleitung mit linksrekursiver LR(1)-Grammatik	Herleitung mit LL(1)-Grammatik
Ausdruck → Ausdruck – Summand → Ausdruck – Summand – Summand → Summand – Summand – Summand → Faktor – Faktor – Faktor → 5 – 4 – 3	Ausdruck → Summand – Ausdruck → Summand – Summand – Ausdruck → Summand – Summand – Summand → Faktor – Faktor – Faktor → 5 – 4 – 3

Abbildung 14: Herleitung des Ausdrucks ”5-4-3” mit LR- und LL-Grammatik

Der aus der LR(1)-Grammatik hervorgehende Syntaxbaum zur Ableitung ist semantisch korrekt. Abbildung 15 zeigt diesen Syntaxbaum.

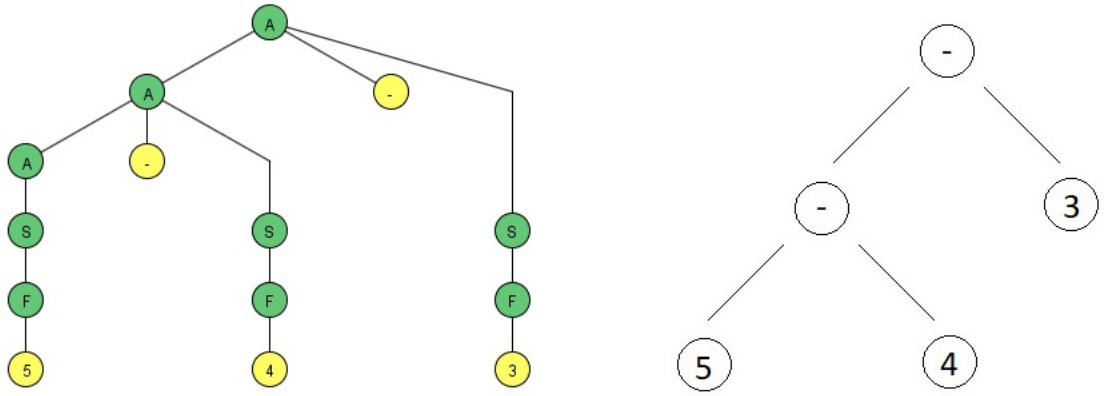

Abbildung 15: Syntaxbaum des Ausdrucks ”5-4-3” mit LR(1)-Grammatik

Um also einen semantisch korrekten Syntaxbaum zu erhalten, müssten wir also einen Parseverfahren implementieren, welches in der Lage ist, mit LR(1)-Grammatiken umzugehen. Dafür baut man einen Shift-Reduce-Parser. Dieser schaut sich das Lookahead-Symbol an und entscheidet dann anhand einer Action- und GoTo-Tabelle, ob er Shiften oder Reducen soll. Shiften bedeutet, dass das erste Zeichen der Tokenliste weggenommen und auf einen Stack gelegt wird. Reducen bedeuten, dass eine Produktionsregel rückwärts angewandt wird. Daher ist es auch eine umgekehrte Rechtsableitung. Währenddessen wird ein Syntaxbaum von unten nach oben aufgebaut.
Man kann allerdings den rekursiven Abstieg auch so erweitern, dass er einen semantisch korrekten Rechenbaum zurückliefert. Diesen kann man nicht mehr als Syntaxbaum bezeichnen, da er nicht die Reihenfolge der Ableitungsschritte wiedergibt. Da wir aber nur einen Rechenbaum benötigen, ist eine solche Erweiterung leichter als ein komplett neues Parseverfahren für LR(1)-Grammatiken zu implementieren, welches eigentlich für komplexere Programmiersprachen gedacht ist. In Abschnitt 3.3.1 werden 2 mögliche Erweiterungen erklärt, welche auch implementiert wurden.

3.3.1 Syntaxbaum durch den rekursiven Aufstieg oder rekursiven Abstieg

Die erste Möglichkeit ist es, die Tokenliste umzudrehen. Daher arbeiten wir den Ausdruck von rechts nach links ab. Es handelt sich damit um einen rekursiven Aufstieg. Die Methoden pruefeAusdruck(), pruefeSummand() und pruefeFaktor() werden so abgeändert, dass sie einen Binärbaum zurückgeben. Weiterhin haben sie nur die Tokenliste als Parameter. Am Anfang jeder Methode wird ein neuer Binärbaum erstellt. Außerdem brauchen wir eine Hilfsmethode inRechtenTeilbaumKopieren(), welche einen Binärbaum übergeben bekommt. Dieser wird so geändert, dass der rechte Teilbaum eine Kopie des original Baumes referenziert und das die Wurzel und der linke Teilbaum ein leeres Token referenzieren.
Jede Methode des rekursiven Aufstiegs trägt ihr abgefragtes Terminalsymbol in den Binärbaum ein. Die Methode pruefeFaktor() trägt eine Zahl in die Wurzel dieses Baumes schreiben und gibt ihn zurück. Lesen die Methoden pruefeAusdruck() und pruefeSummand() einen Operator ein, wenden sie die Methode inRechtenTeilbaumKopieren() an und tragen den Operator in die Wurzel ein. Als linker Teilbaum wird dann der Rückgabebaum des nächsten rekursiven Aufrufes gesetzt.
Tritt an einer Stelle ein syntaktischer Fehler wird ”null” zurückgegeben. Daher muss nach jedem rekursiven Aufruf auch abgefragt werden, ob der jeweilige Baum ”null” referenziert, da es sonst zu einer NullPointerException kommen würde.
Dadurch, dass man die Tokenliste umdreht und dann das linke Nichtterminal in den rechten Teilbaum ableitet, hat man eine Produktion der Form ”Ausdruck $\rightarrow$ Ausdruck + Summand — Ausdruck - Summand”. Dies ist die LR(1)-Grammatik, welche zur gewünschten Semantik des Syntaxbaumes führt. Diese Erweiterung funktioniert allerdings nicht mehr, wenn wir rechtsassoziative Operatoren, wie Potenzieren einführen würde.

Um die zweite Variante zu erläutern, habe ich deren Vorgehensweise mit der Eingabe ”5-4-3*4*(5+4)” in den Abbildungen 23, 24, 25 und 26 aus dem Anhang dargestellt. Dieses Beispiel besteht aus 13 Schritten. Bei jedem Schritt wird angegeben welches Token wir gerade einlesen, wie sich der Fall ändert, welche Methode dieses Token einliest und die Stelle im zugehörige Rechenbaum, bei der wir das Token eintragen. Bei dieser Vorgehensweise braucht man drei globale Binärbäume returnBaum, aBaum und sBaum. Auch wird ein globaler Integer benötigt, welcher festhält, in welchen dieser Binärbäume ein Token eingetragen wird. Dieser wird ”fall” genannt. Die Referenz von ”returnBaum” wird nie verändert. Dies ist der Baum, den wir schrittweise aufbauen und am Ende zurück geben. Die Referenzen aBaum und sBaum werden je nach Bedarf gesetzt. Es wurde eine weitere Ableitung ”Faktor $\rightarrow$ D” und ”D $\rightarrow$ num” hinzugefügt, um die Implementation übersichtlicher zu gestalten. Die Methode pruefeD() prüft nur auf eine Zahl. Anhand des Falls entscheidet die Methode ob die eingelesene Zahl an die Wurzel von aBaum eingetragen wird (Fall 1), in den rechten Teilbaum von aBaum (Fall 2) oder in den rechten

Teilbaum von sBaum (Fall 3).
Die Methode pruefeA() wendet bei einem + oder - die Hilfsmethode inLinkenTeilbaum-Kopieren() auf aBaum an und setzt den Fall auf 2. Die Methode pruefeS() macht das gleiche mit sBaum und setzt den Fall auf 3.
Tritt eine öffnende Klammer auf, wird auf den aktuellen aBaum eine temporäre Referenz tmpBaum gesetzt. Der aktuelle Fall entscheidet dann, ob aBaum und sBaum auf den rechten Teilbaum von aBaum(Fall 2) oder von sBaum(Fall 3) gesetzt werden.
Wird eine schließende Klammer eingelesen, werden die Referenzen von aBaum und sBaum zurück auf ”tmpBaum” gesetzt, falls vor der öffnenden Klammer der Fall 1 war. Das wird sich durch den temporären ”boolean tmpB” gemerkt. Sonst wird die Referenz sBaum zusätzlich auf den rechten Teilbaum von ”tmpBaum” gesetzt.

Die drei Fälle sind aus der Sichtweise der Methode pruefeD() in Abbildung 16 visualisiert.

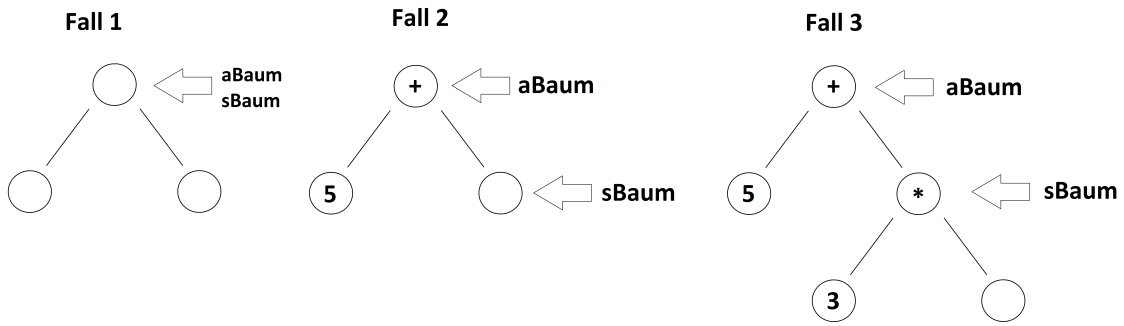

Abbildung 16: Unterscheidung der drei Fälle

3.3.2 Rechenbaum erstellen

Alternativ können wir eine separate Methode implementieren, welche einen Rechenbaum aufbaut. Dies ist die Methode erstelleRechenbaum(). Dafür muss die Tokenliste zunächst auf syntaktische Korrektheit geprüft werden. Das kann zum Beispiel der Kellerautomat übernehmen. Zur Modellierung des Rechenbaumes wurde ein Binärbaum verwendet. Um einen Rechenbaum aus einem arithmetischen Ausdruck zu erstellen, suchen wir den Hauptoperator. Der Hauptoperator ist der Operator, welcher ein + oder - in der höchsten Klammerebene ist und am weitesten rechts steht. Sollte weder + noch - in der höchsten Klammerebene vorhanden sein, ist der Hauptoperator das am weitesten rechts vorkommende * oder /. Die Methode sucheHauptOperator() wurde in der Klasse ”Hilfsmethode” implementiert. Diese bekommt eine Liste übergeben und setzt das aktuelle Element der Liste auf den Hauptoperator. Anschließend wird die Liste zurückgegeben. Dieser Hauptoperator kommt dann in die Wurzel des Binärbaums. Die Tokenliste wird dann in eine Liste links und rechts vom Hauptoperator aufgeteilt. Beide Listen rufen wir rekursiv auf und setzen den zurückgelieferten Binärbaum als linken und rechten Teilbaum. Die Abbruchbedingung der Rekursion ist erfüllt, wenn die linke oder rechte Liste keinen Operator mehr enthält. Trifft das auf eine der Listen zu, setzen wir den jeweiligen Teilbaum auf die Zahl, welche in der Liste enthalten ist. Diese Aufgabe übernimmt die Hilfsmethode pruefeZahl(), welche ein leeres Token zurück gibt, wenn mindestens ein Operator in der Liste vorkommt oder die jeweilige Zahl als Token. Bei einem leerem Token wissen wir also, dass ein weiterer rekursiver Aufruf nötig ist.
Der Nachteil dieser Methode ist, dass durch das Suchen des Hauptoperator, die Liste mehrfach durchlaufen wird. Das bringt eine erhöhte Rechenzeit mit sich.

4 Übersetzer - Erzeugung der Steuerbefehle

Der Übersetzer hat die Aufgabe, den vom Parser aufbereiteten Syntaxbaum, in eine für den Interpreter lesbare Sprache zu verwandeln. Diese Sprache ist bei der Stackmaschine eine Liste aus Instruktionen. Die Klasse Instruktion ist abstrakt und besitzt 5 Unterklassen: ADD, SUB, MUL, DIV und PUSH. Jede von ihnen überschreibt die drei abstrakten Methoden execute(), toString() und toStringZeichen(). Das hat erstens den Vorteil, dass auch einfach weitere Rechenoperationen, wie zum Beispiel Wurzelziehen oder Potenzieren, ergänzt werden könnten und zweitens haben die Klassen Zugriff auf den Stapel der Stackmaschine, sodass der Interpreter einfach nur die Liste aus Instruktionen durchiteriert und für jedes Element die Methode execute() aufruft. Die Instruktion ”ADD” ist so implementiert, dass sie die obersten beiden Zahlen vom Stapel der Stackmaschine nimmt, diese addiert und das Ergebnis auf den Stack legt. Die anderen Rechenoperatoren werden genauso implementiert, nur das bei ”SUB” und ”DIV” das untere Element vom oberen Element subtrahiert, beziehungsweise dividiert wird. Die Instruktion ”PUSH” bekommt im Konstruktor einen ”double”. Dieser wird beim Aufruf von execute() auf den Stack gelegt.
Die Methode übersetzeTokenlisteInInstruktion() kann eine Tokenliste in eine Liste aus Instruktionen übersetzen. Dafür wird die Tokenliste durchlaufen und für jedes Token die zugehörige Instruktion einer Liste angehängt. Die Klasse ”Token” wird um die Methode gibInstruktion() erweitert, welche anhand der Wahrheitswerte istZahl, istADD, istSUB, istMUL und istDIV entscheidet, welche Instruktion zurückgeliefert wird. Die Rückgabe dieser Methode ist die Stackmaschinensprache.
Die Liste der Instruktionen muss in einer Postfixnotation, welche auch als umgekehrte polnische Notation bekannt ist, angeordnet sein. Ein arithmetischer Ausdruck in der Infixnotation wäre ”3+4”. In der Postfixnotation würde man ”3 4 +” schreiben. Der zugehörige Stackcode lautet ”PUSH 3; PUSH 4; ADD”.

4.1 Postorder Traversierung

Liest man einen Rechenbaum in der Postorder-Traversierung aus und hält die Reihenfolge in einer Liste fest, erhält man die umgekehrte polnische Notation. Eine Postorder-Traversierung wird auch als Nebenreihenfolge bezeichnet. Dabei wird zuerst der linke Teilbaum durchlaufen, dann der rechte Teilbaum und zuletzt wird die Wurzel betrachtet. Das Durchlaufen eines Teilbaumes ist dabei als rekursiver Aufruf mit diesem Teilbaum zu verstehen.
Die Methode postOrderTraversal() gibt uns eine Tokenliste zurück, welche wir mittels der Methode übersetzeTokenlisteInInstruktion() in die Stackmaschinensprache überführen. Abbildung 17 beschreibt den Ablauf der Methode postOrderTraversal().

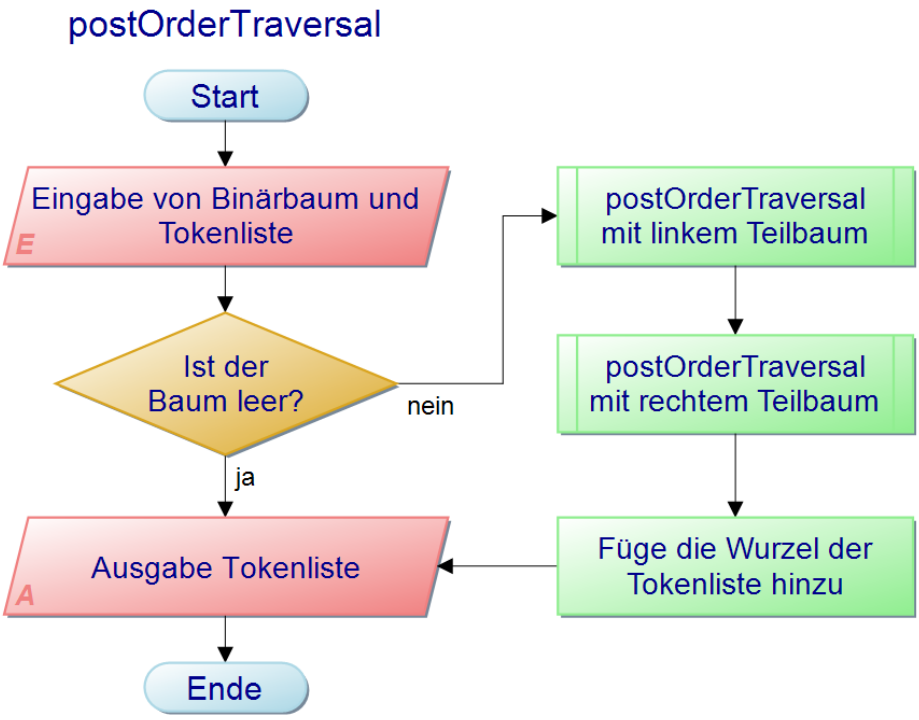

Abbildung 17: Programmablaufplan der Methode postOrderTraversal()

4.2 Shunting-Yard-Algorithmus

Der von Edsger W. Dijkstra erfundene Shunting-yard-Algorithmus kann einen Infixterm in einen Postfixterm umwandeln. Die Vorgehensweise erinnert an einen ”Rangierbahnhof”. Daher auch der englische Name ”shunting yard”. Der Algorithmus braucht einen Zeichenstapel, auf den Operatoren und Klammern kommen. Der Postfixterm wird aufgebaut, während man den Infixterm durchläuft und folgende Regeln beachtet:

1. Operanden(Zahlen) werden an den Postfixterm angehängt.

2. Eine öffnende Klammer kommt auf den Zeichenstapel.

3. Bei einer schließenden Klammer werden solange Operatoren vom Zeichenstapel geholt und an den Postfixterm angehängt, bis eine öffnende Klammer erscheint. Diese wird dann vom Zeichenstapel gelöscht.

4. Kommt ein Operator, und der Stapel ist leer, oder es liegt eine öffnende Klammer oben auf dem Zeichenstapel, so wird der Operator auf den Zeichenstapel gelegt. Liegt ein Operator auf dem Zeichenstapel, der eine höhere Priorität hat, wird dieser vom Stapel genommen und dem Postfixterm angehängt und der eingelesene Operator kommt auf den Zeichenstapel. Hat dagegen der eingelesene Operator eine höhere Priorität als der auf dem Zeichenstapel, wird er nur auf den Stapel gelegt.

5. Ist der Infixterm beendet, wird der restliche Zeichenstapel dem Postfixterm angehängt.

Die Methode shuntingYard() wurde in der Klasse ”Übersetzer” implementiert und bekommt eine Tokenliste und eine GUI übergeben, welche jeden Schritt des Algorithmus darstellt. Die entsprechende GUI ist in Abbildung 18 zu sehen. Die Tokenliste ist der Infixterm, welcher mit einer for-Schleife durchlaufen wird und die jeweiligen Regeln abgefragt werden. Die Methode gibt dann zum Schluss eine Tokenliste zurück, welche in der Postfixnotation ist. Diese wird anschließend von der Methode übersetzeTokenlisteInInstruktion() in die gewünschte Stackmaschinensprache überführt.

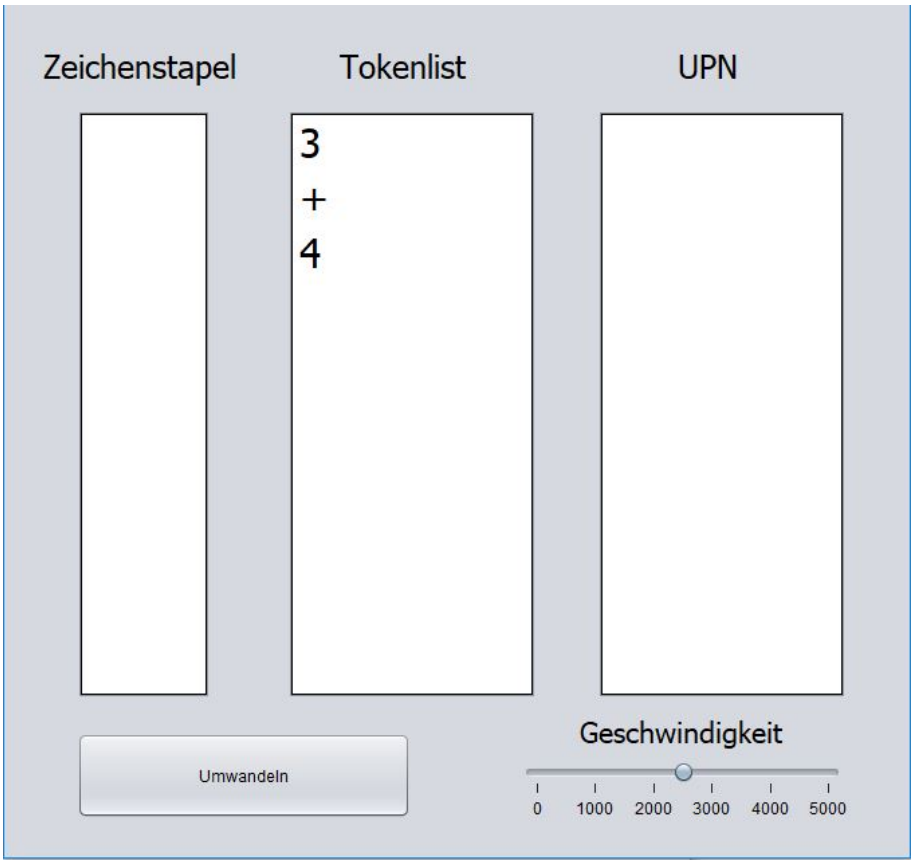

Abbildung 18: Bild der GUI zur Darstellung des Shunting-Yard-Algorithmus

5 Interpreter und Stackmaschine

Die Stackmaschine soll jetzt anhand des Stackcodes den ursprünglichen arithmetischen Ausdruck berechnen. Dafür hat sie einen Stack vom Typ ”double”. Die Methode auswerten() führt Schrittweise jede Instruktion des ”Stackcodes” aus. In Abbildung 19 ist die dazu erstellte GUI zu sehen. Der Schieberegler ”Geschwindigkeit” gibt an, wie viele Millisekunden zwischen den einzelnen Schritten gewartet werden.

Abbildung 19: Bild der GUI zur Darstellung der Stackmaschine

Um den ganzen Ablauf der Übersetzung darzustellen, wurde ein Hauptfenster implementiert. Abbildung 20 zeigt dieses. Dort hat man ganz oben ein Textfeld für die Eingabe des arithmetischen Ausdrucks. Der Button ”Ausdruck scannen” führt die lexikalische Analyse des Scanners aus und speichert die daraus resultierende Tokenliste global ab.
Anschließend wird mit dem Button ”Syntaktische Analyse” die Methode ”pruefeAusdruckKellerautomat()” des Parsers aufgerufen.
War diese Analyse auch erfolgreich, kann der Ausdruck mit dem Button ”In UPN umwandeln” einen Rechenbaum erstellen. Dieser Button öffnet eine weitere GUI, welche in Abbildung 21 zu sehen ist, bei der man die Auswahl zwischen drei Verfahren hat. Dort ist dann im Textfeld die Postorder-Traversierung des Rechenbaumes, also die umgekehrte polnische Notation, angezeigt. Der Rechenbaum dann wird in Abbildung 22 dargestellt.
Mit dem Button ”In Stackcode umwandeln” erfolgt die Übersetzung in Stackmaschinensprache. Hierfür wird die Postorder Traversierung in eine Liste aus Instruktionen übersetzt. Da aber auch der Shunting-Yard-Algorithmus Anwendung finden soll, wird hier die GUI aus Abbildung 18 aufgerufen. Der Stackcode wird dann im Hauptfenster im Textfeld ”Stackcode” eingeblendet. Es ist auch möglich diesen als Textdatei zu exportieren oder eine Textdatei mit Stackcode einzulesen. Anschließend wird mit dem Aufruf von

”Stackcode ausführen” die GUI aus Abbildung 19 geöffnet und der Stackcode abgearbeitet.

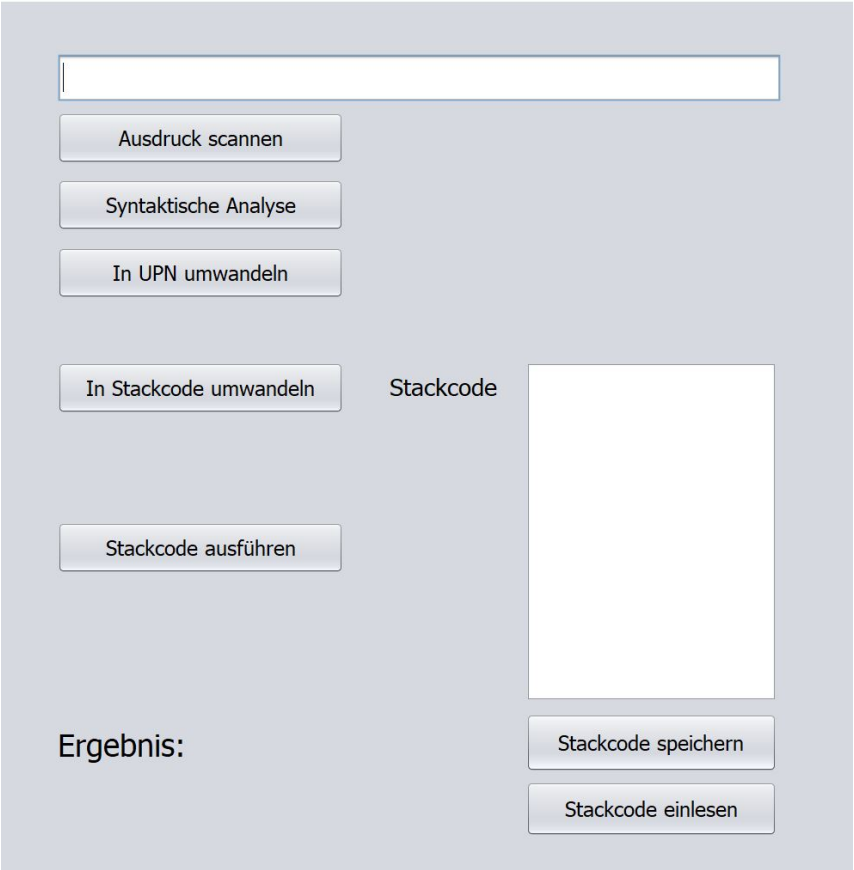

Abbildung 20: Bild der Hauptfensters

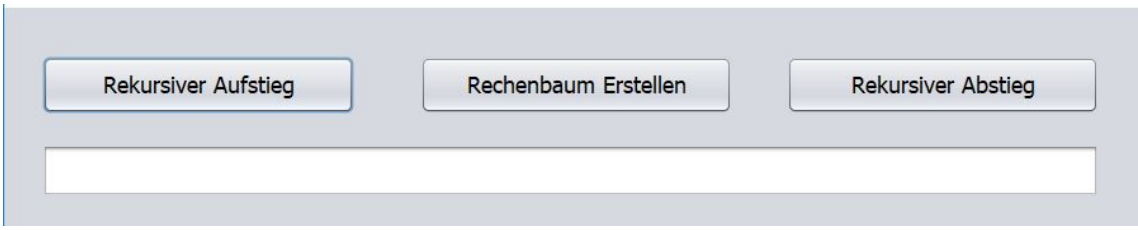

Abbildung 21: Bild der Parser GUI

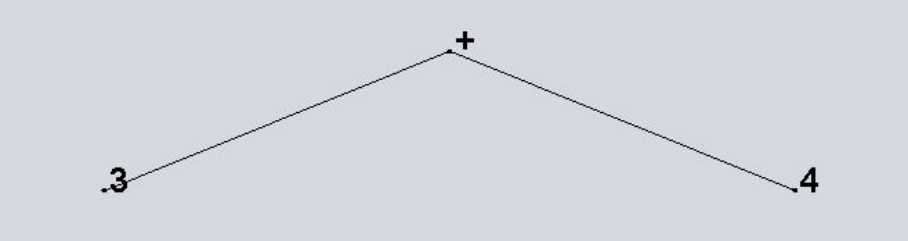

Abbildung 22: Bild der GUI zur Darstellung des Rechenbaumes

6 Fazit

Das Ziel der besonderen Lernleistung war es eine Stackmaschine mit grafischer Oberfläche, einem zugehörigen lexikalischen Scanner, einem syntaktischen Parsers und Übersetzer zur schrittweisen Überführung eines arithmetischen Ausdrucks in die Stackmaschinensprache, zu implementieren.
Die syntaktische Analyse kann entweder mit einem Kellerautomaten oder mit dem rekursivem Abstieg durchgeführt werden.
Der rekursive Abstieg wurde so erweitert, dass er einen Syntaxbaum mit der gewollten Semantik ausgibt. Hierfür bietet der Parser drei Möglichkeiten:

- Rekursiver Aufstieg(Abstieg mit umgedrehter Tokenliste)
- Rechenbaum erstellen durch Suchen des Hauptoperators(setzt syntaktische Korrektheit voraus)
- Rekursiver Abstieg mit zwei Binärbäumen zur Hilfe

Der Übersetzer kann auf zwei unterschiedlichen Wegen den Stackcode erzeugen:

- Den Syntaxbaum in einer Postorder-Traversierung auslesen um eine umgekehrte polnische Notation zu erhalten
- Die Tokenliste mittels Shunting-yard-Algorithmus in eine umgekehrte polnische Notation überführen

Alle Optionen stehen in der GUI zur Auswahl. Mithilfe eines Schiebereglers kann die Verarbeitungsgeschwindigkeit verstellt werden, sodass man die einzelnen Schritte der Stackmaschine und des Shunting-yard-Algorithmus nachvollziehen kann.
Ein Problem war, dass der Syntaxbaum einer LL(1)-Grammatik nicht die gewollte Semantik beschreibt. Die erste Erweiterung des rekursiven Abstiegs zur Ausgabe eines Syntaxbaumes war falsch. Zu diesem Zeitpunkt war noch nicht klar, dass durch die Entfernung der Linksrekursion der Syntaxbaum nicht mehr die gewünschte Semantik wiedergibt. Mit den oben beschriebenen Möglichkeiten des Parsers wurden andere Möglichkeiten gefunden, um dennoch einen semantisch korrekten Rechenbaum zu erstellen.
Die eigentliche Lösung des Problems wäre es, einen LR-Parser für die LR(1)-Grammatik zu schreiben. Dieser Parser wäre jedoch sehr umständlich und für dieses Projekt überflüssig. Außerdem war dafür die Zeit zu knapp.
Das Projekt hat mir geholfen mein Wissen über Compilerbau zu vertiefen. Bei der Umsetzung konnte ich praktische Erfahrung beim Programmieren in Java und beim Planen eines Projektes sammeln.

Literatur

[1] "Interpreter." https://www.xovi.de/wiki/Interpreter, März 2018.

[2] R. Faßbender, "Ein Ausflug in den Compilerbau - oder wie Computer Code verstehen,"

[3] "Java(Programmiersprache)." https://de.wikipedia.org/wiki/Java_(Programmiersprache), März 2018.

[4] "BlueJ." https://de.wikipedia.org/wiki/BlueJ, März 2018.

[5] "NetBeans IDE." https://de.wikipedia.org/wiki/NetBeans_IDE, März 2018.

[6] F. Berlin, "Syntaktische Analyse - Bottom-up,"

[7] "Syntaxbaum." https://de.wikipedia.org/wiki/Syntaxbaum, März 2018.

[8] D. Hoffmann, *Theoretische Informatik*. Carl Hanser Verlag GmbH Co KG, 2015.

[9] T. Brandes, "Ein Ausflug in den Compilerbau - Syntaktische Analyse und Berechnung arithmetischer Ausdrücke,"

[10] "Compiler design - semantic analysis." https://www.tutorialspoint.com/compiler_design/compiler_design_semantic_analysis.htm, März 2018.

Erklärung

Ich erkläre, dass ich die Facharbeit ohne fremde Hilfe angefertigt und nur die im Literaturverzeichnis angeführten Quellen und Hilfsmittel benutzt habe.

Rheinbach, 20. März 2018

Erik Springer

Anhang

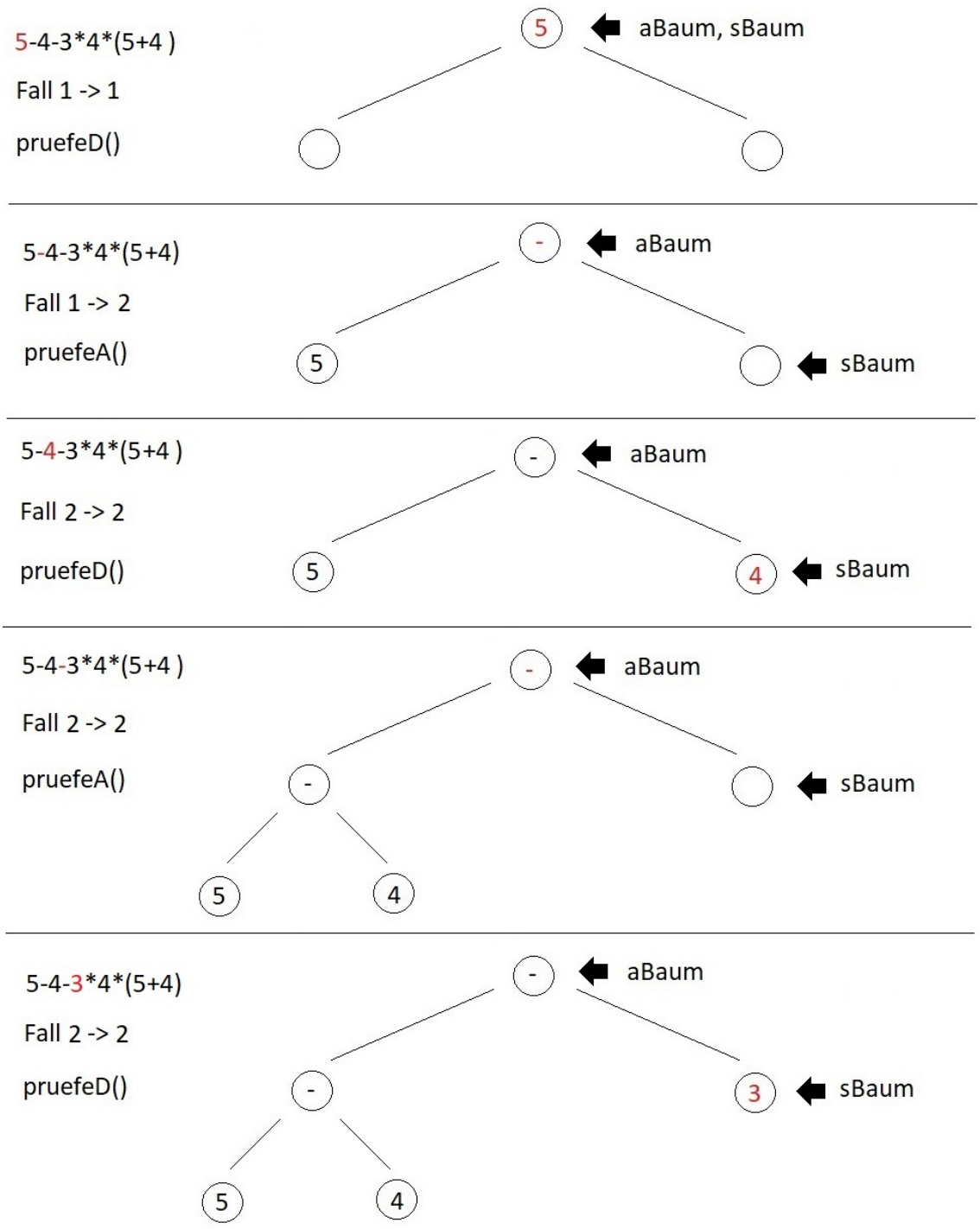

Abbildung 23: Schritte 1 bis 5

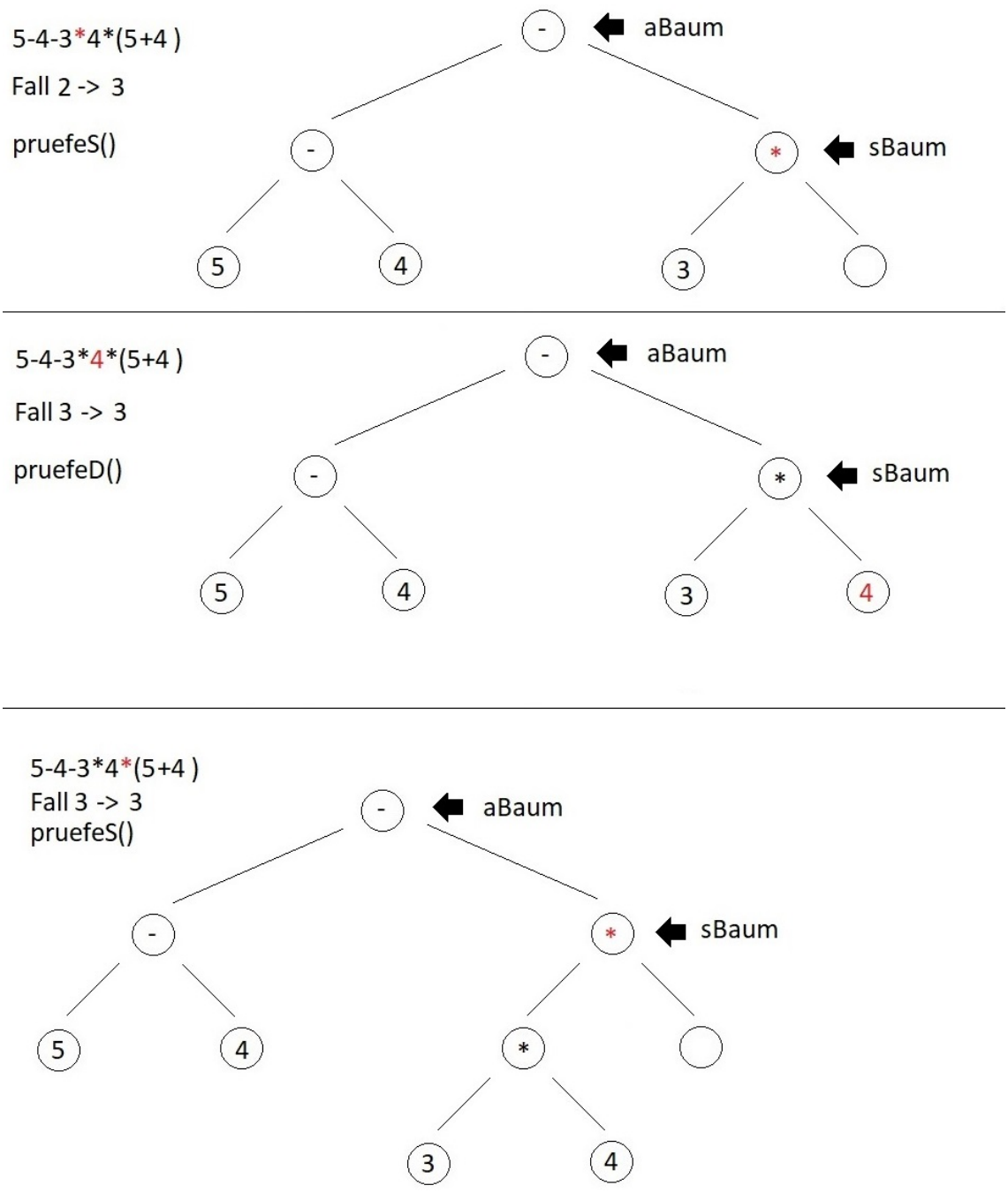

Abbildung 24: Schritte 6 bis 8

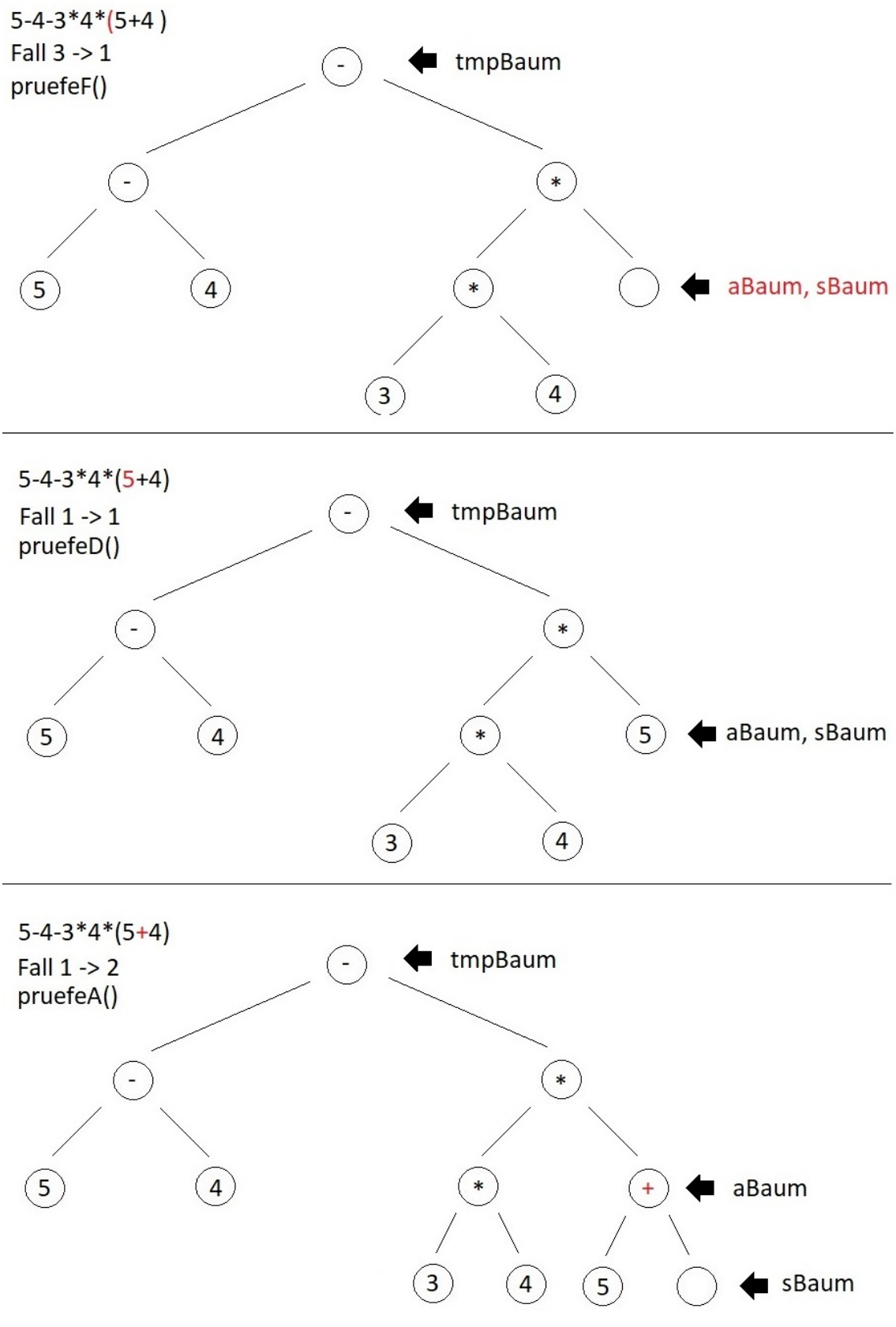

Abbildung 25: Schritte 9 bis 11

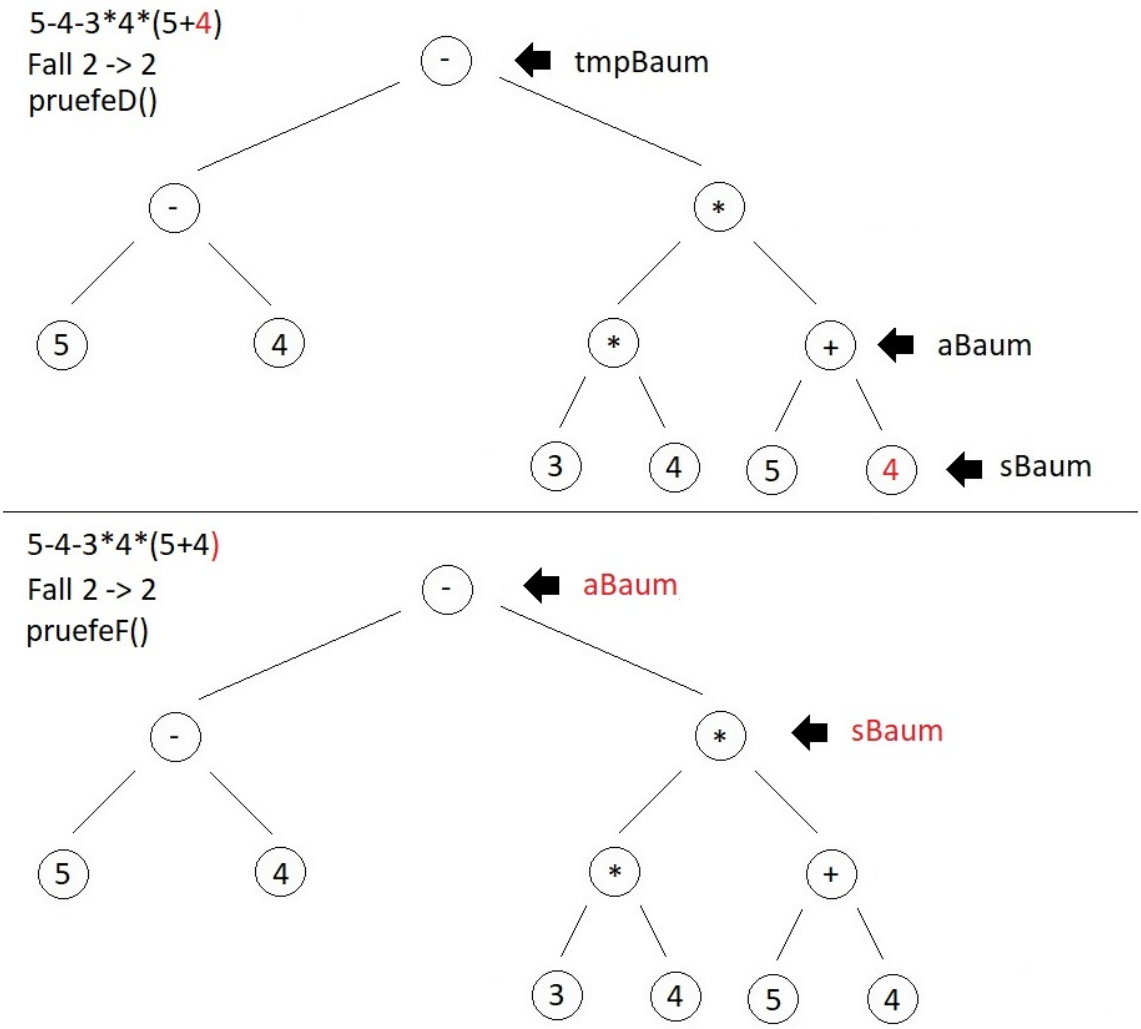

Abbildung 26: Schritte 12 bis 13