

Wissenschaftliche Hausarbeit im Rahmen der Ersten Staatsprüfung
für das Lehramt an Gymnasien im Fach Informatik,
eingereicht dem Landesschulamt - Prüfungsstelle Frankfurt am Main - .

Thema:

Visualisierung der Ableitungsbäume von Grammatik –
Eine Simulationsumgebung für den Informatikunterricht

Verfasser: Sebastian Beier
Mittelstraße 6
61231 Bad Nauheim

Gutachter: Prof. Dr. Jürgen Poloczek

Inhaltsverzeichnis

1	Einleitung	1
1.1	Ziel dieser Arbeit	1
1.2	Aufbau der wissenschaftlichen Hausarbeit	2
1.3	Beigefügter Datenträger	3
2	Formale Sprachen	4
2.1	Einführung und Definition	4
2.2	Beispiele für formale Sprachen	6
2.3	Eigenschaften von unendlichen Sprachen	7
2.3.1	Abzählbarkeit von Sprachen	8
2.3.2	Aufzählbarkeit von Sprachen	9
2.3.3	Entscheidbarkeit einer Sprache	10
2.3.4	Übersicht über die Menge der unendlichen Sprachen	11
3	Grammatiken	12
3.1	Einführung und Definition	12
3.2	Beispiele für Grammatiken	16
3.3	Die Chomsky-Hierarchie	18
3.3.1	Definition der Chomsky-Hierarchie	18
3.3.2	Weitere Eigenschaften und Definitionen	19
3.3.3	Die Backus-Naur-Form für kontextfreie Grammatiken	20
3.4	Ableitungsbäume	22
3.4.1	Ableitungsbäume für kontextfreie Grammatiken	22
3.4.2	Linksableitung und Rechtsableitung	23
3.4.3	Das Problem der Mehrdeutigkeit	24
3.5	Das Wortproblem	26
3.5.1	Das Wortproblem für Typ-0 Grammatiken bzw. Sprachen	26
3.5.2	Das Wortproblem für Typ-1 Grammatiken bzw. Sprachen	26
3.5.3	Das Wortproblem für Typ-2 Grammatiken bzw. Sprachen	28
3.5.4	Das Wortproblem für Typ-3 Grammatiken bzw. Sprachen	32
3.6	Erweiterung der Mengenhierarchie der formalen Sprachen	33
4	Die Simulationsumgebung: GrammatikSimulator	34
4.1	Vergleich zu alternativen Simulationsumgebungen	34
4.1.1	JFLAP	34
4.1.2	Machines	35
4.1.3	AtoCC	36
4.2	Zielsetzung für die Simulationsumgebung	37
4.3	Umsetzung der Darstellung von Ableitungsbäumen für kontextsensitive Grammatiken	40

4.4	Überblick über die Realisierung der Simulationsumgebung	43
4.4.1	Die Datenstrukturen für Grammatiken und Ableitungsbäume	43
4.4.2	Benutzeroberfläche und Funktionsweise des Grammatikeditors	51
4.4.3	Visualisierung der Ableitungsbäume mit JGraphX.....	55
4.4.4	Die verwendeten Algorithmen	57
4.4.5	Benutzeroberfläche des Hauptfensters.....	66
4.5	Anwendungsbeispiele.....	68
4.5.1	Erstellen einer Grammatik.....	68
4.5.2	Durchführung einer manuellen Ableitung.....	71
4.5.3	Prüfung eines Wortes	75
4.5.4	Erzeugen von Worten	78
4.5.5	Bekannte Problemfälle	82
5	Abbildungsverzeichnis	84
6	Literaturverzeichnis	85
7	Erklärung.....	86

1 Einleitung

1.1 Ziel dieser Arbeit

Nach dem noch aktuellen Lehrplan Informatik für den Unterricht an gymnasialen Oberstufen in Hessen ist für das Halbjahr Q3 der Qualifikationsphase das Thema „Konzepte und Anwendungen der Theoretischen Informatik“ vorgesehen. Als Teil dieses Themas gehört die Behandlung von Grammatiken als Mittel zur Beschreibung von formalen Sprachen zu den Pflichtthemen von Grundkurs und Leistungskurs. Der Lehrplan (vgl. [LpfInf]) empfiehlt dabei zunächst die Behandlung der verschiedenen Automatenmodelle, um dann auf diesem Weg zu den Grammatiken überzuleiten. Für die Behandlung der Automatenmodelle wird dabei explizit zum Einsatz geeigneter Simulationsumgebungen aufgefordert.

Es ist jedoch ein alternativer Weg zur Erarbeitung des Themas denkbar. Nach der Einführung der grundlegenden Theorien über formale Sprachen können zunächst Grammatiken als Mittel zur Beschreibung formaler Sprachen mit unterschiedlichen Eigenschaften eingeführt und anschließend die zu den Sprachklassen passenden Automatenmodelle behandelt werden. Um dabei den Unterricht, wie vom Lehrplan gefordert, möglichst anschaulich zu gestalten, bietet sich auch hier der Einsatz einer geeigneten Simulationsumgebung an.

Der Einsatz einer Simulationsumgebung für Grammatiken ist natürlich nicht nur bei dem skizzierten alternativen Vorgehen in der Unterrichtsreihe sinnvoll, sondern bietet den Schülerinnen und Schülern in jedem Fall die Möglichkeit zur aktiven Auseinandersetzung mit dem Thema Grammatiken.

Bereits existierende Programme bieten hierzu nicht alle wünschenswerten Funktionen, und es existiert noch kein Programm, das alle für den Einsatz in der Schule sinnvollen Funktionen in einem einzigen Programm bündelt. Der Grund hierfür ist, dass die verfügbaren Programme in der Regel vorwiegend für den Einsatz an Hochschulen konzipiert sind.

Ziel dieser Arbeit ist es auf Grund dieser Ausgangslage einer neuen Simulationsumgebung für den Einsatz in der Schule zu entwickeln, mit der die Lernenden sich aktiv mit dem Thema Grammatiken befassen können.

Das für diese Arbeit erstellte Programm bietet zum einen die Möglichkeit eigene Grammatiken erstellen zu können. Zum anderen werden verschiedene Wege bereitgestellt, um mit diesen Grammatiken Elemente der Sprache der jeweiligen Grammatik zu erzeugen. Um die im Lehrplan geforderte Anschaulichkeit für die Auseinandersetzung der Schülerinnen und Schüler mit dem Thema Grammatiken zu ermöglichen, liegt der Fokus des Programms auf der Erstellung und Visualisierung von Ableitungsbäumen für die erzeugten Elemente der Sprache.

1.2 Aufbau der wissenschaftlichen Hausarbeit

Zu Beginn dieser Arbeit möchte ich zuerst einen Überblick über die fachwissenschaftlichen Grundlagen dieses Themas geben. Hierzu gehe ich zunächst auf die allgemeinen Grundlagen zu formalen Sprachen ein, um grundlegende Begriffe und Eigenschaften einzuführen. Darauf aufbauend wird im nächsten Abschnitt das Thema Grammatiken behandelt und die für diese Arbeit relevanten Definitionen und Eigenschaften gestellt. Die zu den wissenschaftlichen Grundlagen gemachten Ausführungen bzw. Darstellungen basieren dabei auf der spezifischen Fachliteratur und wurden entsprechend angepasst bzw. erweitert.

Im zweiten Teil der Arbeit werde ich das für die Arbeit erstellte Java Programm GrammatikSimulator vorstellen. Um die Ausgangslage besser darstellen zu können, werden zunächst einige bereits existierende Simulationsumgebungen vorgestellt und anschließend wird die genaue Zielsetzung für die neue Simulationsumgebung erläutert. Des Weiteren werde ich die Art der Gestaltung und Umsetzung des Programms darstellen und ausführliche begründen. Abschließend wird die Funktionsweise des Programms anhand eines Anwendungsbeispiels vorgestellt.

1.3 Beigefügter Datenträger

Dieser Arbeit ist eine CD mit den für diese Arbeit erstellten Dateien beigefügt.

Im Einzelnen sind dies:

- Die Quellcode-Dateien für das Programm GrammatikSimulator.
- Das Programm GrammatikSimulator in kompilierter Version.
- Beispieldateien für Grammatiken zur Verwendung im Programm.
- Diese wissenschaftliche Hausarbeit als PDF-Dokument.

2 Formale Sprachen

2.1 Einführung und Definition

Wenn man über die Theorie von Sprachen spricht, wird ein Laie damit nicht unbedingt die Informatik als die zugehörige Wissenschaft assoziieren. Die Bedeutung von Sprachen in der Informatik wird erst ersichtlich, wenn man sich darüber bewusst wird, dass Handlungsanweisungen für Computer oder allgemeine Informatiksysteme einer Formulierung bedürfen, die von dem jeweiligen System auch verstanden werden kann.

In der Theorie wird zum einen unterschieden zwischen natürlichen Sprachen und formalen Sprachen. „Eine natürliche Sprache wird im Wesentlichen durch sprachliche Sätze charakterisiert, die von Menschen geäußert werden, wobei man grammatikalische Korrektheit und sinnvollen Gebrauch voraussetzt.“

([He12], S. 5) Natürliche Sprachen unterliegen dabei zwar einem Regelwerk, das ihre Syntax (also die grammatikalische Korrektheit) in der Regel eindeutig festlegt, aber die Semantik, also die Bedeutung der Sätze oder Texte, kann nicht eindeutig durch einen Formalismus bestimmt werden. Für einen Menschen erschließt sich die Semantik aus dem Kontext, in dem der Satz oder der Text steht. Dieser Kontext kann von heutigen Maschinen jedoch nicht interpretiert werden.

Deshalb braucht es für die Beschreibung von Aufgaben für Informatiksysteme Sprachen, deren Semantik weitgehend eindeutig erkannt werden kann. Diese Sprachen werden als formale Sprachen bezeichnet und ihre Theorie stellt eine eigene Teildisziplin in der Informatik dar.

Dabei werden Begriffe wie Alphabet, Wort und Sprache in einer allgemeineren Form als bei natürlichen Sprachen verwendet. Daher folgt zunächst eine Definition dieser Begriffe.

Definition 1 (Alphabet, Wort, Sprache) [He12]

Ein Alphabet ist eine endliche, nichtleere Menge von Zeichen (auch Symbole oder Buchstaben genannt). Dabei verwendet man meist V als Abkürzung für Alphabete (von engl.: vocabulary).

Sei V ein Alphabet und $k \in \mathbb{N}$. Dabei ist $\mathbb{N} = \{1, 2, 3, \dots\}$ die Menge der natürlichen Zahlen einschließlich der Null.

Eine endliche Folge (x_1, x_2, \dots, x_k) mit $x_i \in V$ ($i = 1, \dots, k$) heißt Wort über V der Länge k . Es gibt genau ein Wort der Länge 0, das Leerwort, das mit ε bezeichnet wird.

Die Länge eines Wortes x wird durch $|x|$ dargestellt.

Man verwendet, wann immer es das Alphabet erlaubt, die Kurzschreibweise $x_1 x_2 \dots x_k$ für Wörter, indem man die Zeichen einfach aneinanderfügt. (Beim Alphabet $\{1, 11\}$ wäre dies nicht möglich, denn man könnte z.B. bei dem Wort 111 nicht erkennen, aus welchen Zeichen es besteht.)

V^* bezeichnet die Menge aller Wörter über V .

Die Menge der nichtleeren Wörter über V ist $V^+ := V^* \setminus \{\varepsilon\}$.

Jede (beliebige) Teilmenge von V^* wird als Sprache oder formale Sprache (über dem Alphabet V) bezeichnet. Als Abkürzung für Sprache verwendet man meistens L (von engl.: language).

Aus der Definition ergeben sich einige Unterschiede zu natürlichen Sprachen, auf die ich hinweisen möchte:

- Bei natürlichen Sprachen werden über dem Alphabet (z.B. $V = \{a, b, c, \dots, z\}$) die einzelnen Worte gebildet. Diese bezeichnet man in der natürlichen Sprache als den Wortschatz. Im Sinne einer formalen Sprache bildet die Menge der Worte jedoch bereits eine Sprache.
- Die Zeichen bzw. Symbole einer formalen Sprache müssen keine einzelnen Zeichen im Sinne des natürlichen Sprachgebrauchs sein. Das Alphabet einer formalen Sprache kann aus Zeichen bestehen, die im Sinne einer natürlichen Sprache Worte oder zumindest Zeichenfolgen sind.

2.2 Beispiele für formale Sprachen

Um die Struktur von formalen Sprachen und den Unterschied zu natürlichen Sprachen nochmals zu verdeutlichen, möchte ich einige Beispiele für formale Sprachen vorstellen.

Beispiel 1

Es sei $V = \{0,1\}$. Die Menge aller Wörter V^* ist also die Menge aller endlichen Folgen aus 0 und 1 inklusive dem Leerwort ε . Eine Sprache L über diesem Alphabet V ist zum Beispiel die Menge aller Binärzahlen ohne Vorzeichen und ohne führende Nullen, also

$$L = \{0,1,10,11,100,101,110,111, \dots\}.$$

Beispiel 2

Es sei $V = \{a, b, c, \dots, z\}$. Mit diesem Alphabet können Wörter einer natürlichen Sprache gebildet werden. Erweitert man das Alphabet um die Großbuchstaben, Umlaute und Sonderzeichen, dann wäre eine Teilmenge von V^* die Sprache $L = \text{Die Menge aller deutschen Wörter}$.

Beispiel 3

Es sei $V = \text{Menge der Worteinträge im Duden}$. Während diese Menge V in Beispiel 2 selbst eine Sprache darstellt, können die deutschen Wörter natürlich auch als Alphabet genommen werden. Eine Sprache L über diesem Alphabet wäre dann die Menge aller grammatikalisch richtig gebildeten deutschen Sätze. Hierbei wäre ein „deutscher Satz“ aus Sicht der formalen Sprache dann ein Wort über dem Alphabet V .

Beispiel 4

Es sei $V =$ Menge der Eingabesymbole einer Programmiersprache.

Wie einleitend erwähnt, braucht es in der Informatik formale Sprachen, um Handlungsanweisungen für Informatiksysteme verständlich formulieren zu können. Ein typisches Beispiel für eine solche formale Sprache ist eine Programmiersprache. Ein korrektes Programm stellt damit ein Wort über der Menge der Eingabesymbole dar. Um nun die Teilmenge von V^* zu bilden, die der Sprache

$L =$ eine Programmiersprache (zum Beispiel Java) entspricht, bilden zu können, bedarf es analog zur natürlichen Sprache einer Grammatik, aus der sich die syntaktisch richtigen Worte über V ableiten lassen.

2.3 Eigenschaften von unendlichen Sprachen

Während das Alphabet V und die über V gebildeten Worte per Definition immer endliche Mengen sind, kann man anhand der vorhergehenden Beispiele leicht erkennen, dass V^* , die Menge aller Worte über dem Alphabet, immer eine unendliche Menge ist. Damit ist V^* auf Grund der Definition, unabhängig vom zugrunde liegenden Alphabet, immer eine unendliche Sprache.

Aus der Mathematik ist bekannt, dass man unendliche Mengen noch durch weitere Eigenschaften unterscheiden bzw. unterteilen kann. Dies ist auch in analoger Weise für Sprachen möglich, wie ich im Folgenden darstellen möchte. Da formale Sprachen nicht das eigentliche Thema dieser Arbeit sind, wird auf die Darstellung der ausführlichen Beweise für die Sätze in diesem Kapitel verzichtet. Bei Bedarf sind diese unter anderem in [He12] zu finden.

2.3.1 Abzählbarkeit von Sprachen

Die erste Eigenschaft für eine unendliche Menge ist die Abzählbarkeit. Unter der Abzählbarkeit einer Menge versteht man, dass man die Elemente der Menge so ordnen kann, dass man jedem Element eindeutig eine Nummer zu ordnen kann. Man kann also vom ersten, zweiten, dritten, ... Element der Menge sprechen. Aus dem Schulunterricht ist in der Regel bekannt, dass die Mengen der natürlichen, der ganzen oder auch der rationalen Zahlen abzählbar sind, während die Menge der reellen Zahlen diese Eigenschaft nicht mehr besitzt und als überabzählbar bezeichnet wird.

Formal wird diese Eigenschaft durch folgende Definition bestimmt:

Definition 2 (Abzählbarkeit) [He12]

Eine Menge M ist abzählbar, falls es eine Funktion $f: \mathbb{N} \rightarrow M$ gibt, die surjektiv ist (d.h. alle Elemente aus M kommen als Bildelement vor), oder falls $M = \emptyset$.

Ist eine Menge nicht abzählbar, so heißt sie überabzählbar.

Für formale Sprachen gelten nun folgende Eigenschaften:

Satz 1: [He12]

Die Menge V^* aller Wörter über einem Alphabet V ist abzählbar.

Satz 2: [He12]

Jede Teilmenge M' einer abzählbaren Menge M ist abzählbar.

Folgerung 1: [He12]

Nach Satz 1 ist V^* immer abzählbar. Da eine Sprache L nach Definition 1 eine Teilmenge von V^* ist, ist L nach Satz 2 abzählbar. Somit ist jede Sprache L abzählbar.

2.3.2 Aufzählbarkeit von Sprachen

Neben der Abzählbarkeit ist eine mögliche Eigenschaft für Mengen, dass ein Element der Menge direkt erzeugt werden kann, ohne diese erst durch abzählen in der Menge bestimmen zu müssen. Zudem gibt die Eigenschaft der Abzählbarkeit nur Auskunft über die Existenz eines Elementes, jedoch nicht über seine Gestalt. Um diese Konstruierbarkeit der Elemente einer Menge auszudrücken, wird die Eigenschaft der Aufzählbarkeit einer Menge wie folgt definiert:

Definition 3 (Aufzählbarkeit) [He12]

Eine Menge M ist aufzählbar (auch rekursiv aufzählbar oder semi-entscheidbar), falls es eine surjektive Funktion $f : \mathbb{N} \rightarrow M$ gibt und einen Algorithmus, der es gestattet, für jedes $n \in \mathbb{N}$ den Funktionswert $f(n)$ zu berechnen, oder falls $M = \emptyset$.

Die Folge $f(0), f(1), f(2), \dots$ heißt Aufzählung von M .

Sprachen, die diese Eigenschaft besitzen, werden als aufzählbare Sprachen bezeichnet. Von Informatiksystemen können nur aufzählbare Sprachen verarbeitet werden.

Es gelten weiterhin folgende Eigenschaften:

Satz 3: [He12]

Jede endliche Menge ist aufzählbar.

Satz 4: [He12]

Sei V ein Alphabet. Dann ist die Menge V^* aller Wörter über V aufzählbar.

Aus Definition 3 ist offensichtlich, dass jede abzählbare Menge auch abzählbar ist, während die Umkehrung nicht automatisch gültig ist. Entsprechend gibt es neben den abzählbaren Sprachen auch abzählbare Sprachen, die nicht abzählbar sind. Auf diese Sprachen wird im Folgenden jedoch nicht näher eingegangen.

2.3.3 Entscheidbarkeit einer Sprache

Ein wichtiges Problem in der Theorie der formalen Sprachen ist die Frage, ob Wort w Element einer Sprache L ist oder nicht. Für eine endliche Sprache L ist diese Frage leicht zu beantworten, denn es lässt sich einfach eine Liste der Worte von L aufstellen und mit dem Wort w vergleichen. Für unendliche Sprachen ist diese Frage viel schwieriger. Die bis jetzt stärkste Eigenschaft der Aufzählbarkeit hilft hier nur zu einem Teil weiter. Diese garantiert, dass jedes Wort w einer Sprache in endlicher Zeit erzeugt werden kann. Für den Fall, dass w zur Sprache L gehört, kann dies theoretisch in endlicher Zeit bestätigt werden, in dem alle Wörter der Sprache so lange gebildet werden, bis w gebildet wurde. Anders sieht dies für den Fall aus, dass w nicht zu L gehört. Da L eine unendliche Sprache ist, würde das Verfahren, um alle Worte zu konstruieren, unendliche Zeit benötigen. Es könnten in endlicher Zeit zwar beliebig viele Worte erzeugt werden, aber nie garantiert werden, dass das Wort w nicht zu den noch nicht konstruierten Worten gehört.

Die Frage „gehört das Wort w zur Sprache L oder nicht“ in endlicher Zeit entscheiden zu können, ist für eine Reihe von Anwendungen der Informatik von Bedeutung. Zum Beispiel muss ein Compiler beim Übersetzen eines Quellcodes in endlicher Zeit entscheiden können, ob der Quellcode ein zulässiges Programm formuliert.

Sprachen, für die diese Frage in endlicher Zeit beantwortet werden kann, heißen, entsprechend der folgenden Definition, entscheidbar.

Definition 4 (Entscheidbarkeit) [He12]

Gegeben sei ein Alphabet V . Eine Sprache $L \subseteq V^*$ heißt entscheidbar, falls es einen abbrechenden Algorithmus, Entscheidungsverfahren genannt, gibt, der für jedes $w \in V^*$ feststellt, ob $w \in L$ oder $w \notin L$.

Weiterhin gilt folgende Eigenschaft:

Satz 5: [He12]

Jede entscheidbare Sprache ist aufzählbar.

2.3.4 Übersicht über die Menge der unendlichen Sprachen

Die dargestellten Eigenschaften von formalen Sprachen bilden eine Hierarchie in der Menge der unendlichen Sprachen. Diese Mengenhierarchie wird in Abbildung 1 noch einmal durch ein entsprechendes Mengendiagramm graphisch dargestellt.

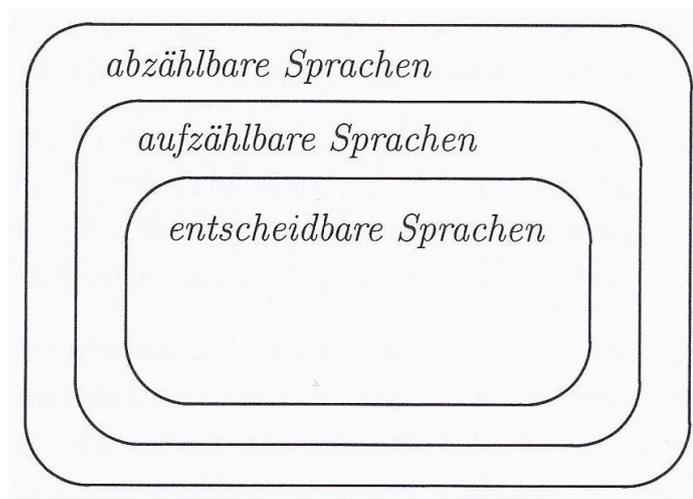


Abbildung 1: Mengenhierarchie der formalen Sprachen.
Entnommen aus [He12], S. 19

3 Grammatiken

3.1 Einführung und Definition

Im vorherigen Abschnitt wurden formale Sprachen zunächst nur allgemein definiert und einige Eigenschaften von unendlichen Sprachen vorgestellt. Auf dieser Basis haben wir bislang nur zwei einfache Möglichkeiten eine Sprache anzugeben: Entweder als die Menge aller Worte über dem zugrundeliegenden Alphabet oder durch die Aufzählung der Elemente der Sprache. Um unendliche Sprachen angeben zu können, die eine echte Teilmenge von V^* darstellen, benötigt man einen Algorithmus, um die Worte der Sprache erzeugen zu können. Ein solches Verfahren ist auch, wie bei der Eigenschaft der Entscheidbarkeit bereits aufgezeigt, notwendig, um konkrete Probleme mit formalen Sprachen lösen zu können.

Bei natürlichen Sprachen, die unendliche Sprachen darstellen, wird dieses Problem durch die Angabe eines Regelwerkes, der Grammatik einer Sprache, gelöst. Durch die Kenntnis dieser endlichen Beschreibung der Struktur einer Sprache ist es möglich, alle grammatikalisch korrekten Sätze zu bilden.

Dieses Prinzip lässt sich auch für die formalen Sprachen anwenden und wird auch hier als Grammatik bezeichnet. Das Grundprinzip einer Grammatik für formale Sprachen soll zunächst noch einmal am Beispiel des Satzbaus in der deutschen Sprache dargestellt werden.

Beispiel 5

Eine vereinfachte Grundregel für deutsche Sätze lautet: Ein Satz besteht aus einem Subjekt, einem Prädikat und einem Objekt.

Diese Regel lässt sich in folgender Kurzform schreiben:

Satz \rightarrow Subjekt Prädikat Objekt

In gleicher Form können weitere Regeln angegeben werden, die festlegen, was ein Subjekt, ein Prädikat oder ein Objekt ist:

Subjekt \rightarrow Artikel Substantiv

Prädikat \rightarrow Verb

Objekt \rightarrow Artikel Substantiv

Und weiter:

Artikel \rightarrow Der oder Artikel \rightarrow den

Substantiv \rightarrow Junge oder Substantiv \rightarrow Ball

Verb \rightarrow wirft

Mit diesen Regeln lässt sich nun ein einfacher deutscher Satz konstruieren, in dem man schrittweise die Regeln anwendet:

- 1) Satz
- 2) Subjekt Prädikat Objekt
- 3) Artikel Substantiv Verb Artikel Substantiv
- 4) Der Junge wirft den Ball

Um einen Satz, also ein Wort w über dem Alphabet V , das die Worte der deutschen Sprache als Elemente hat, zu bilden, legt man eine Menge von Hilfszeichen, in der Regel Variablen genannt, fest und bildet dann Regeln, wie diese Variablen durch weitere Variablen oder Zeichen aus dem Alphabet der Sprache ersetzt werden können. Nach diesem Schema können alle syntaktisch korrekten Sätze gebildet werden.

Zu beachten ist, dass auf diese Weise zwar syntaktische Korrektheit gegeben ist, nicht jedoch semantische Korrektheit. Im obigen Beispiel könnte nach den gegebenen Regeln genauso auch der Satz „Der Ball wirft den Jungen“ gebildet werden, der zwar grammatikalisch richtig, inhaltlich aber unsinnig ist. Die semantische Richtigkeit ist für formale Sprachen jedoch im Rahmen dieser Ar-

beit nicht weiter von Bedeutung, denn es soll im Folgenden um Grammatiken als Mittel zur Erzeugung der syntaktisch korrekten Elemente einer Sprache gehen.

Dazu gilt es nun genauer zu definieren, was eine Grammatik für formale Sprachen ist. Wie im obigen Beispiel gezeigt, müssen für eine Grammatik zwei Mengen von Zeichen festgelegt werden. Eine Menge von Hilfszeichen, die Variablen, und eine Menge von Zeichen, die die endgültigen Worte der Sprache bilden. Diese endgültigen Zeichen werden als Terminale bezeichnet. Dazu kommt eine Menge von Regeln, meistens Produktionen genannt, die festlegen, wie Zeichen durch andere Zeichen ersetzt werden können. Zudem muss festgelegt werden, wo man mit den Ersetzungen beginnen soll. Im Beispiel sollte am Ende ein Satz entstehen, weshalb mit dem Zeichen „Satz“ begonnen wurde. Das Zeichen, mit dem die Abfolge der Ersetzungen begonnen wird, wird als Startsymbol bezeichnet.

Eine formale Definition für Grammatiken, die für beliebige Sprachen angewendet werden kann, sieht wie folgt aus:

Definition 5 (Grammatik) [He12]

Eine Grammatik ist ein 4-Tupel $G = (V_N, V_T, P, S)$ mit

1. V_N, V_T sind endliche, nichtleere Mengen mit $V_N \cap V_T = \emptyset$.
 V_N ist die Menge der Variablen (oder auch Nichtterminale, nicht-terminal Symbole).
 V_T ist die Menge der Terminale (oder terminal Symbole).
2. P ist eine endliche Menge von Regeln der Form

$$\alpha \rightarrow \beta$$

mit $\alpha \in (V_N \cup V_T)^+, \beta \in (V_N \cup V_T)^*$.

Die Elemente von P werden Produktionen, Regeln, Produktionsregeln oder Grammatikregeln genannt.

3. $S \in V_N$ ist das Startsymbol.

Um im Verlauf der Arbeit weiter über die Eigenschaften von Grammatiken sprechen zu können, soll anhand der folgenden Definitionen noch weiteres Vokabular zu Grammatiken festgelegt werden.

Definition 6 (Ableitung) [He12]

Seien $G = (V_N, V_T, P, S)$, $V = V_N \cup V_T$ und $x, y \in V^*$.

1. Anwendung einer Grammatikregel

x wird unmittelbar überführt in y

$x \Rightarrow y$ genau dann, wenn: $x = \gamma\alpha\delta, y = \gamma\beta\delta$ und $\alpha \rightarrow \beta \in P$ wobei $\alpha \in V^+, \beta, \gamma, \delta \in V^*$.

Andere Redewendungen:

x führt unmittelbar zu y

y ist direkt (unmittelbar) aus x ableitbar

2. Anwendung mehrerer Grammatikregeln nacheinander

x wird übergeführt in y

$x \overset{*}{\Rightarrow} y$ genau dann, wenn: Es existiert eine endliche Folge

$(w_0, \dots, w_n), n \in \mathbb{N}$, so dass $x = w_0, w_n = y$ und $w_{i-1} \Rightarrow w_i$ ($i = 1, \dots, n$).

Diese Folge heißt Ableitung von y aus x .

Bezeichnung: $w_0 \Rightarrow w_1 \Rightarrow \dots \Rightarrow w_n$

Andere Redewendungen:

x führt zu y

y ist aus x ableitbar

Bemerkung: Es sind auch Ableitungen in 0 Schritten zugelassen,

deshalb gilt generell: $x \overset{*}{\Rightarrow} x$

Definition 7 (Erzeugte Sprache) [He12]

Die von einer Grammatik $G = (V_N, V_T, P, S)$ erzeugte (auch dargestellte oder definierte) Sprache:

$$L(G) = \{x \in V_T^* \mid S \overset{*}{\Rightarrow} x\}.$$

Definition 8 (Äquivalenz von Grammatiken) [He12]

Zwei Grammatiken G_1 und G_2 heißen äquivalent, falls gilt:

$$L(G_1) = L(G_2).$$

3.2 Beispiele für Grammatiken

Zum weiteren Verständnis von Grammatiken werden einige Beispiele für Grammatiken und die von ihnen erzeugten Sprachen vorgestellt:

Beispiel 6

Wir nehmen die Grammatik

$$G = (V_N = \{S\}, V_T = \{0\}, P = \{S \rightarrow 0S, S \rightarrow 0\}, S).$$

Die Ableitungen sehen allgemein wie folgt aus:

$$S \Rightarrow 0S \Rightarrow 00S \Rightarrow \dots \Rightarrow 0^{n-1}S \Rightarrow 0^n$$

Die dargestellte Sprache ist $L(G) = \{0^n | n \geq 1\}$.

Beispiel 7

Wir nehmen die Grammatik

$$G = (V_N = \{S\}, V_T = \{0,1\}, P = \{S \rightarrow 0S1, S \rightarrow 01\}, S).$$

Die Ableitungen sehen allgemein wie folgt aus:

$$S \Rightarrow 0S1 \Rightarrow 00S11 \Rightarrow \dots \Rightarrow 0^{n-1}S1^{n-1} \Rightarrow 0^n1^n$$

Die dargestellte Sprache ist $L(G) = \{0^n1^n | n \geq 1\}$.

Beispiel 8

Wir nehmen die Grammatik $G = (\{E, T, F\}, \{(,), a, +, *\}, P, E)$ mit

$$P = \{E \rightarrow T, E \rightarrow E + T, T \rightarrow F, T \rightarrow T * F, F \rightarrow a, F \rightarrow (E)\}.$$

Die von der Grammatik G dargestellte Sprache $L(G)$ umfasst die korrekt geklammerten arithmetischen Ausdrücke. Z.B. lässt sich die Ableitung

$$E \Rightarrow a * a * (a + a) + a$$
 bilden.

Beispiel 9

Wir nehmen die Grammatik $G = (V_N, V_T, P, S)$ mit $V_N = \{S, B, C\}$,

$V_T = \{a, b, c\}$ und

$P = \{S \rightarrow aSBC, S \rightarrow aBC, CB \rightarrow BC, aB \rightarrow ab, bB \rightarrow bb, bC \rightarrow bc, cC \rightarrow cc\}$.

Ein Beispiel für eine Ableitung in G ist:

$S \Rightarrow aSBC \Rightarrow aaSBCBC \Rightarrow aaaBCBCBC \Rightarrow aaaBBCCBC$
 $\Rightarrow aaaBBCBCC \Rightarrow aaaBBBCCC \Rightarrow aaabBBCCC \Rightarrow aaabbBCCC$
 $\Rightarrow aaabbbCCC \Rightarrow aaabbbcCC \Rightarrow aaabbbccC \Rightarrow aaabbbccc$
 $= a^3b^3c^3$

Wie sich anhand des Beispiels leicht vermuten lässt, ist die von G dargestellte Sprache

$$L(G) = \{a^n b^n c^n \mid n \geq 1\}.$$

3.3 Die Chomsky-Hierarchie

3.3.1 Definition der Chomsky-Hierarchie

In den Beispielen 6 bis 9 kann man sehen, dass diese sich in Hinblick auf ihre Struktur bei der Form ihrer Produktionen deutlich unterscheiden. Während in den Beispielen 6 bis 8 immer nur eine Variable auf der linken Seite der Produktionen steht, stehen bei Beispiel 9 auch mehrere Symbole auf den linken Seiten der Produktionen. Anhand dieses und weiterer Merkmale lassen sich die Grammatiken in verschiedene Klassen einteilen. Die folgende Definition für die Klassifizierung von Grammatiken geht auf den Linguisten Noam Chomsky zurück. Diese nach ihm benannte Klassifizierung drückt zudem die hierarchische Beziehung zwischen den einzelnen Typen aus.

Definition 9 (Chomsky-Hierarchie) [He12]

Sei $G = (V_N, V_T, P, S)$ eine Grammatik, $V = V_N \cup V_T$.

G ist vom

- Typ-0, falls keine Einschränkung vorliegt.
- Typ-1 oder kontextsensitiv, falls jede Produktion die Form

$$\alpha_1 A \alpha_2 \rightarrow \alpha_1 \beta \alpha_2$$

hat, wobei $A \in V_N, \alpha_1, \alpha_2 \in V^*, \beta \in V^+$, mit einer Ausnahme: $S \rightarrow \varepsilon$, dann darf S aber in keiner rechten Seite einer Produktion vorkommen.

- Typ-2 oder kontextfrei, falls jede Produktion die Form

$$A \rightarrow \beta$$

hat, wobei $A \in V_N, \beta \in V^+$, mit derselben Ausnahmeregelung wie in Typ 1.

- Typ-3 oder regulär, falls jede Produktion die Form

$$A \rightarrow aB \text{ oder } A \rightarrow a$$

hat, wobei $A, B \in V_N, a \in V_T$, mit derselben Ausnahmeregelung wie in Typ 1.

Anmerkungen

- 1) Die Definition für Typ-1 bzw. kontextsensitive Grammatiken ist in der Literatur mit großen Unterschieden angegeben. In [We05] wird für Typ-1 Grammatiken gefordert, dass die Produktionen die Form $u \rightarrow v$ mit $u \in V_N^+$, $v \in ((V_N \cup V_T) \setminus \{S\})^+$ und $|u| \leq |v|$ oder $S \rightarrow \varepsilon$ haben.

In [Sch09] wird sogar nur gefordert, dass für alle $w_1 \rightarrow w_2$ in P gilt: $|w_1| \leq |w_2|$.

Für den Zweck dieser Arbeit soll im Folgenden die unter Definition 9 gegebene Form für Typ-1 Grammatiken gelten. Die Entscheidung hierfür wird später noch ausführlicher begründet.

- 2) Für Grammatiken vom Typ-2 oder Typ-3 können auch Produktionen der Form $A \rightarrow \varepsilon$, mit $A \in V_N$ zugelassen werden. Dies erhöht nicht die Mächtigkeit der beiden Grammatiktypen, da jede Grammatik G , die Produktionen dieser Art enthält, in eine äquivalente Grammatik G' überführt werden kann, die der unter Definition 9 gegebenen Form entspricht (vgl. [Sch09], S. 10).
- 3) Reguläre Grammatiken in der angegebenen Form werden auch als rechtslinear bezeichnet. Analog dazu gibt es die linkslinearen Grammatiken, deren Produktionen dann die Form $A \rightarrow Ba$ oder $A \rightarrow a$ haben (vgl. [He12], S. 33). Ansonsten gelten die gleichen Forderungen wie in Definition 9.

3.3.2 Weitere Eigenschaften und Definitionen

Entsprechend der Chomsky-Hierarchie für Grammatiken kann auch den von den jeweiligen Grammatiktypen erzeugten Sprachen ein Typ zugeordnet werden:

Definition 10 (Typ einer Sprache) [He12]

Eine Sprache L ist vom Typ i ($i = 0,1,2,3$), falls eine Grammatik G vom Typ i existiert mit $L(G) = L$.

Satz 6 [He12]

Gilt $0 \leq i \leq j \leq 3$ und ist die Grammatik G (bzw. die Sprache L) vom Typ j , so ist G (bzw. L) auch vom Typ i .

Es ist leicht zu sehen, dass die Definitionen der spezielleren Typen auch jeweils den Definitionen der allgemeineren Typen genügen. Somit ist z.B. eine reguläre Grammatik bzw. Sprache auch immer kontextfrei.

3.3.3 Die Backus-Naur-Form für kontextfreie Grammatiken

Ein gängiger Formalismus zur Vereinfachung der Darstellung von kontextfreien Grammatiken ist die Angabe der Produktionen in der sogenannten Backus-Naur-Form, kurz BNF. Diese wurde von Backus und Naur im Zusammenhang mit der Konstruktion der Programmiersprache ALGOL 60 entwickelt. Die Backus-Naur-Form unterscheidet sich dabei nur in der Syntax von der theorieorientierten Schreibweise.

Die Darstellung der Produktionen in der Backus-Naur-Form hat folgende Gestalt (vgl. [He12], S. 38-39):

- Variablen werden mit spitzen Klammern ($\langle \dots \rangle$) umschlossen.
- Terminalsymbole werden ohne spezielle Kennzeichnung geschrieben.
- Statt „ \rightarrow “ wird das Symbol „ $::=$ “ verwendet.
- Alle Regeln, die die gleiche linke Seite besitzen, werden zu einer einzelnen Metaregel zusammen gefasst. Dabei werden die verschiedenen rechten Seiten durch „ $|$ “ voneinander getrennt.

In der Praxis wird oftmals nur die Schreibweise für die Metaregeln verwendet und auf die spitzen Klammern und das Symbol „ $::=$ “ verzichtet. Zum besseren Verständnis zeigt das folgende Beispiel eine kontextfreie Grammatik in der theorieorientierten Schreibweise und in der Backus-Naur-Form.

Beispiel 10

Wir nehmen die bereits aus Beispiel 8 bekannte Grammatik $G = (\{E, T, F\}, \{(\,), a, +, *\}, P, E)$, die wir nun entsprechend als kontextfrei einordnen können.

Die Produktionen in der theorieorientierten Schreibweise haben die Gestalt:

$$\begin{aligned} P = \{ & E \rightarrow T, \\ & E \rightarrow E + T, \\ & T \rightarrow F, \\ & T \rightarrow T * F, \\ & F \rightarrow a, \\ & F \rightarrow (E) \} \end{aligned}$$

Die äquivalente Darstellung in Backus-Naur-Form hat die Gestalt:

$$\begin{aligned} P = \{ & \langle E \rangle ::= \langle T \rangle \mid \langle E \rangle + \langle T \rangle, \\ & \langle T \rangle ::= \langle F \rangle \mid \langle T \rangle * \langle F \rangle, \\ & \langle F \rangle ::= a \mid (\langle E \rangle) \} \end{aligned}$$

3.4 Ableitungsbäume

Ableitungsbäume sind eine Darstellung, mit der sich visualisieren lässt, wie ein Wort w vom Startsymbol abgeleitet wird. In der Literatur ist diese Darstellung nur für kontextfreie bzw. Typ-2 Grammatiken definiert. Nach Satz 6 kann diese Darstellung selbstverständlich auch für reguläre bzw. Typ-3 Grammatiken angewendet werden. Später in dieser Arbeit wird eine Möglichkeit vorgestellt, wie auch die Ableitungen für kontextsensitive bzw. Typ-1 Grammatiken in Form von Bäumen visualisiert werden können, sofern die Produktionen die in Definition 9 angegebene Form haben.

3.4.1 Ableitungsbäume für kontextfreie Grammatiken

Bei kontextfreien Grammatiken G wird in jedem Ableitungsschritt ein Variablensymbol durch eine neue Symbolfolge ersetzt. In einer Baumstruktur kann man nun jedes Symbol einem Knoten zuordnen, wobei das Variablensymbol als Vaterknoten für die aus dem Variablensymbol abgeleiteten neuen Symbole genommen wird. So kann man jeder Ableitung $S \xRightarrow{*} w$ (wobei S das Startsymbol und $w \in L(G)$ ist) einen Baum zu ordnen mit S als Wurzel und den Terminalen, die das Wort w bilden, als Blätter.

Dies wird durch folgende Definition nochmals präzisiert:

Definition (Ableitungsbaum) [He12]

Sei $G = (V_N, V_T, P, S)$ eine kontextfreie Grammatik. Ein Ableitungsbaum (auch: Parsebaum, Strukturbaum oder Syntaxbaum) für G ist ein Baum mit Markierungen, wobei gilt:

- 1) Die Blätter sind mit Terminalen markiert.
- 2) Die inneren Knoten sind mit Variablen markiert.
- 3) Ist A die Markierung eines inneren Knoten und sind X_1, \dots, X_n die Markierungen der direkten Nachfolger, so ist $A \rightarrow X_1 \dots X_n \in P$.

Beispiel 11

Wir nehmen die kontextfreie Grammatik $G = (V_N, V_T, P, S)$ mit

$V_N = \{S\}$, $V_T = \{*, +, (,)\}$ und

$P: S \rightarrow S + S \mid S * S \mid (S) \mid a$.

Eine Ableitung für das Wort $w = a + (a + a) * a$ ist

$$\begin{aligned} S &\Rightarrow S + S \Rightarrow S + S * S \Rightarrow S + (S) + S \Rightarrow S + (S + S) * S \\ &\Rightarrow a + (S + S) * S \Rightarrow a + (a + S) * S \\ &\Rightarrow a + (a + a) * S \Rightarrow a + (a + a) * a \end{aligned}$$

Diese Ableitung kann durch den in Abbildung 2 gezeigten Ableitungsbaum dargestellt werden.

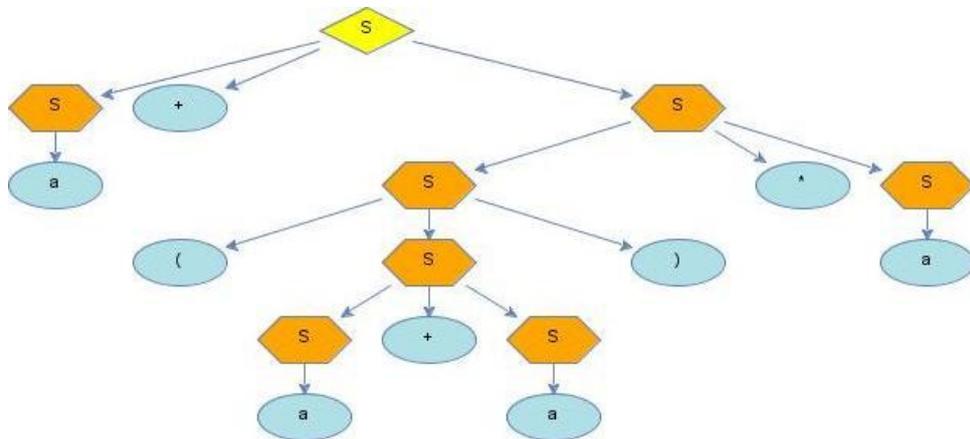


Abbildung 2: Ableitungsbaum für das Wort $w = a + (a + a) * a$

3.4.2 Linksableitung und Rechtsableitung

In Beispiel 11 kann man sehen, dass sich aus dem Baum nicht darauf schließen lässt, in welcher Reihenfolge die einzelnen Ableitungsschritte vorgenommen wurden. Wie man sich leicht überlegen kann, kann man die Schritte in verschiedenen Reihenfolgen ausführen, um den gleichen Ableitungsbaum zu erhalten.

Um hier einem übersichtlichen System zu folgen, kann man nach zwei Prinzipien vorgehen. Entweder wird in jedem Ableitungsschritt immer die am weitesten links stehende Variable ersetzt, oder es wird analog dazu immer die am weitesten rechts stehende Variable ersetzt. Man nennt diese Vereinbarungen Links- bzw. Rechtsableitung, die durch folgende Definition noch einmal eindeutig festgelegt werden.

Definition 11 (Linksableitung – Rechtsableitung) [He12]

Eine Ableitung in einer kontextfreien Grammatik heißt Linksableitung, falls immer die am weitesten links stehende Variable ersetzt wird. Wird immer die am weitesten rechts stehende Variable ersetzt, so spricht man von einer Rechtsableitung.

Anmerkung:

Es ist leicht nachvollziehbar, dass jedem Ableitungsbaum eindeutig eine Linksableitung zugeordnet werden kann und umgekehrt. Weiterhin gelten folgende äquivalente Aussagen (vgl. [Sch09], S. 16-17):

$$\begin{aligned}x \in L(G) &\Leftrightarrow \text{es gibt (irgend)eine Ableitung von } x \\ &\Leftrightarrow \text{es gibt einen Ableitungsbaum mit } x \text{ an den Blättern} \\ &\Leftrightarrow \text{es gibt eine Linksableitung von } x \\ &\Leftrightarrow \text{es gibt eine Rechtsableitung von } x\end{aligned}$$

3.4.3 Das Problem der Mehrdeutigkeit

3.4.3.1 Definition

Wie oben dargelegt wurde, kann jedem Ableitungsbaum eindeutig eine Links- bzw. Rechtsableitung zugeordnet werden. Es kann jedoch vorkommen, dass es zu einem Wort w , das von einer kontextfreien Grammatik G erzeugt wird, mehrere Ableitungsbäume gibt, die sich in ihrer Struktur unterscheiden. Eine solche Grammatik ist also in Bezug darauf, wie die Worte ihrer Sprach $L(G)$ erzeugt werden, nicht eindeutig. Dies wird auch durch die folgende Definition zum Ausdruck gebracht.

Definition 12 (Mehrdeutigkeit) [He12]

Eine kontextfreie Grammatik G heißt mehrdeutig, falls es ein Wort in $L(G)$ gibt, das mindestens zwei verschiedene Ableitungsbäume (bzw. Linksableitungen oder Rechtsableitungen) hat, andernfalls heißt G eindeutig.

Eine kontextfreie Sprache L heißt inhärent mehrdeutig, falls alle Grammatiken, die L erzeugen, mehrdeutig sind, andernfalls heißt L eindeutig.

Satz 7 [Sch09]

Für eine kontextfreie Grammatik G ist das Entscheidungsproblem, ob G mehrdeutig ist, unentscheidbar.

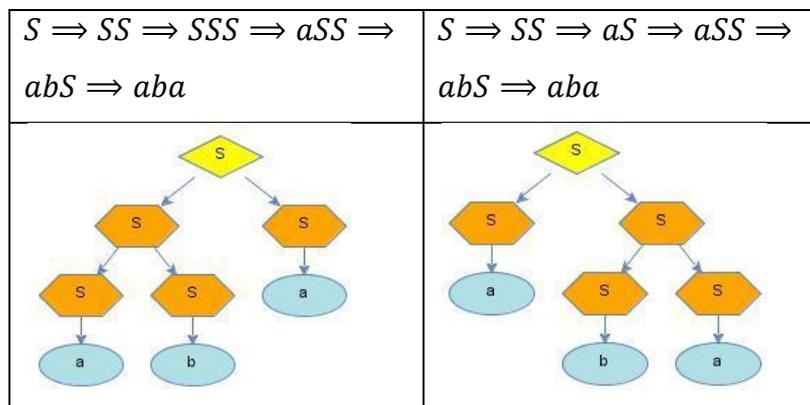
Der Beweis zu Satz 7 ist unter anderem in [Sch09] zu finden. Da für den Beweis umfassende zusätzliche Grundlagen zur Berechenbarkeitstheorie nötig sind, wird auf eine Darstellung in dieser Arbeit verzichtet.

3.4.3.2 Beispiele für Mehrdeutigkeit bei Grammatiken und Sprachen

Beispiel 12

Wir nehmen die Grammatik $G = (\{S\}, \{a, b\}, P, S)$ mit $P = \{S \rightarrow SS \mid a \mid b\}$.

Das Wort $w = aba$ hat in G zwei Linksableitung bzw. zwei Ableitungsbäume:



3.5 Das Wortproblem

Im Kapitel über formale Sprachen wurde bereits das Problem angesprochen, für eine Sprache L zu entscheiden, ob für ein gegebenes Wort w gilt: $w \in L$. Dieses Problem kann nun auch auf Grammatiken bzw. auf die von Grammatiken erzeugten Sprachen bezogen werden. Dieses Problem wird das Wortproblem genannt.

Eine wichtige Frage ist nun, für welche Typen von Grammatiken das Wortproblem in endlicher Zeit entschieden werden kann.

3.5.1 Das Wortproblem für Typ-0 Grammatiken bzw. Sprachen

Für Typ-0 Grammatiken ist das Wortproblem nicht entscheidbar. Sprachen vom Typ-0 gehören damit nicht zur Menge der entscheidbaren Sprachen. Sie sind jedoch aufzählbar bzw. rekursiv aufzählbar. Die entsprechenden Beweise können unter anderem in [Sch09] nachgelesen werden. Da diese Beweise wieder umfassende zusätzliche Grundlagen zur Berechenbarkeitstheorie erfordern, wird auf eine Darstellung der Beweise in dieser Arbeit wieder verzichtet.

3.5.2 Das Wortproblem für Typ-1 Grammatiken bzw. Sprachen

Das Wortproblem für Typ-1 Grammatiken bzw. Sprachen ist entscheidbar mit exponentieller Zeitkomplexität. Der nachfolgende Satz mit Beweis führt dies genauer aus.

Satz 8 [Sch09]

Das Wortproblem für Typ-1 Sprachen (und damit auch für Typ-2, Typ-3 Sprachen) ist entscheidbar.

Genauer:

Es gibt einen Algorithmus, der bei Eingabe einer Kontextsensitiven Grammatik $G = (V_N, V_T, P, S)$ und eines Wortes $w \in V_T^*$ in endlicher Zeit entscheidet, ob $w \in L(G)$ oder $w \notin L(G)$.

Beweis: [Sch09]

Für $m, n \in \mathbb{N}$ definiere Mengen T_m^n wie folgt:

$$T_m^n = \left\{ w \in (V_N \cup V_T) \mid \begin{array}{l} |w| \leq n \text{ und } w \text{ lässt sich aus } S \\ \text{in höchstens } m \text{ Schritten ableiten} \end{array} \right\}$$

Die Mengen T_m^n , $n \geq 1$, lassen sich induktiv über m definieren:

$$T_0^n = \{S\}$$

$$T_{m+1}^n = Abl_n(T_m^n), \text{ wobei}$$

$$Abl_n(X) = X \cup \{w \in (V_N \cup V_T)^* \mid |w| \leq n \text{ und } w' \Rightarrow w \text{ für ein } w' \in X\}$$

Diese Darstellung ist nur für Typ-1 Grammatiken korrekt. (Bei einer Typ-0 Grammatik könnte es sein, dass aus einem Wort der Länge $\geq n$ ein Wort der Länge $\leq n$ ableitbar ist.)

Da es nur endlich (nämlich exponentiell in n) viele Wörter der Länge $\leq n$ in $(V_N \cup V_T)^*$ gibt, ist

$$\bigcup_{m \geq 0} T_m^n$$

für jedes n eine endliche Menge (der Mächtigkeit $2^{O(n)}$). Daraus ergibt sich, dass es ein m gibt mit

$$T_m^n = T_{m+1}^n = T_{m+2}^n = \dots$$

Falls nun x , $|x| = n$, in $L(G)$ liegt, so muss x in $\bigcup_{m \geq 0} T_m^n$ und damit in T_m^n für ein m liegen. Damit ergibt sich der folgende Algorithmus:

```
INPUT ( $G, x$ );  $\{|x| = n\}$ 
 $T := \{S\}$ ;
REPEAT
     $T_1 := T$ ;
     $T := Abl_n(T_1)$ ;
UNTIL ( $x \in T$ ) OR ( $T = T_1$ );
IF  $x \in T$ 
    THEN WriteString('x liegt in  $L(G)$ ')
    ELSE WriteString('x liegt nicht in  $L(G)$ ')
END
```

q.e.d.

Satz 9 [Sch09]

Das Wortproblem für Typ-1 Sprachen ist NP-Hart.

Für den Beweis sei wieder auf [Sch09], S. 173 verwiesen, da die Darstellung des Beweises für den Zweck dieser Arbeit zu umfangreich ist.

Die Zugehörigkeit zur Komplexitätsklasse NP-Hart zeigt auch, dass Algorithmen für das Wortproblem für Typ-1 Sprachen kein polynomiales Laufzeitverhalten erzielen können. Es ist also unmöglich, dieses Problem effizient von einem Computer lösen zu lassen.

3.5.3 Das Wortproblem für Typ-2 Grammatiken bzw. Sprachen

Für eine Grammatik bzw. Sprache vom Typ-2 kann das Wortproblem in einer Zeit von $O(n^3)$ entschieden werden, wenn die Grammatik in der Chomsky-Normal-Form vorliegt. Der Algorithmus hierzu, der Cocke-Younger-Kasami-Algorithmus (kurz CYK-Algorithmus), wurde in den 1960er Jahren von Cocke, Younger und Kasami entwickelt. Das Wortproblem für kontextfreie Sprachen bzw. Grammatiken kann somit effizient von Computern gelöst werden.

Bevor im Folgenden der CYK-Algorithmus ausführlich dargestellt wird, erfolgt zunächst eine kurze Einführung der Chomsky-Normalform, die eine Grundlage des CYK-Algorithmus ist.

Definition 13 (Chomsky-Normalform) [Sch09]

Eine kontextfreie (bzw. Typ-2) Grammatik G mit $\varepsilon \notin L(G)$ heißt in Chomsky-Normalform (kurz CNF), falls alle Regeln eine der beiden Formen haben:

$$A \rightarrow BC$$

bzw.

$$A \rightarrow a$$

wobei $A, B, C \in V_N, a \in V_T$.

Satz 10 [Sch09]

Zu jeder kontextfreien (bzw. Typ-2) Grammatik G mit $\varepsilon \notin L(G)$ gibt es eine Chomsky-Normalform-Grammatik G' mit $L(G) = L(G')$.

Für den Beweis zu Satz 10 vgl. [Sch09], S. 45-46. Auf die ausführliche Darstellung des Beweises im Rahmen dieser Arbeit wird verzichtet, da die Normalisierung nicht von weiterer Bedeutung für das Thema dieser Arbeit ist.

Der CYK-Algorithmus

Für eine Typ-2 Grammatik G in Chomsky-Normalform gilt, dass ein Wort w der Länge 1, wie z.B. $w = a$, nur dann abgeleitet werden kann, wenn G eine Produktion der Form $A \rightarrow a$ enthält. Für längere Worte der Form $w = a_1 \dots a_n$ muss es jedoch zunächst eine Produktion $A \rightarrow BC$ so geben, dass $B \overset{*}{\Rightarrow} a_1 \dots a_k$ und $C \overset{*}{\Rightarrow} a_{k+1} \dots a_n$ gilt, wobei sich dieses Prinzip für B und C wiederholt. Man kann also sagen, dass von B der vordere Teil des Wortes abgeleitet wird und von C der hintere, wobei sich der Prozess für die Teilworte fortsetzt. Dazu muss es ein k mit $1 \leq k \leq n$ geben.

Das Wortproblem für ein Wort w der Länge n kann so reduziert werden auf zwei Teilprobleme für Wörter der Länge k und $n - k$, wobei k natürlich unbekannt ist.

Mit der Methode der dynamischen Programmierung können nun systematisch alle Teilprobleme berechnet werden, indem, beginnend mit allen Teilwörtern der Länge 1, für alle Teilwörter ihre Ableitbarkeit aus den Variablen der Grammatik geprüft wird. Diese Informationen werden in einer Tabelle vermerkt, so dass in Folgeschritten wieder darauf zurückgegriffen werden kann. Für den folgenden Algorithmus soll folgende Notation zu Grunde liegen: Für ein Wort w bezeichnet $w_{i,j}$ dasjenige Teilwort von w , das an der i -ten Stelle beginnt und Länge j hat.

Der Algorithmus verwendet entsprechend eine Tabelle $T[1 \dots n, 1 \dots n]$, wobei aber nicht alle Matrixelemente benötigt werden, sondern nur eine Dreiecksmatrix.

Für $j = 1, \dots, n$ und für $i = 1, \dots, n + 1 - j$ wird in $T[i, j]$ die Menge der Variablen gespeichert, aus denen $w_{i,j}$ abgeleitet werden kann.

Für das Eingabewort w gilt $w \in L(G)$ genau dann, wenn $S \in T[1, n]$.

Der eigentliche Algorithmus sieht nun wie folgt aus:

```
Eingabe:  $w = w_1 w_2 \dots w_n$ 
FOR  $i := 1$  TO  $n$  DO /*Fall  $j = 1$ */
     $T[i, 1] := \{A \in V_N \mid A \rightarrow w_i \in P\}$ ;
END
FOR  $j := 2$  TO  $n$  DO /*Fall  $j > 1$ */
    FOR  $i := 1$  TO  $n + 1 - j$  DO
         $T[i, j] := \emptyset$ ;
        FOR  $k := 1$  TO  $j - 1$  DO

             $T[i, j] := T[i, j] \cup \left\{ A \in V_N \mid \begin{array}{l} A \rightarrow BC \in P \wedge B \in T[i, k], \\ \wedge C \in T[i + k, j - k] \end{array} \right\}$ ;

        END
    END
END
IF  $S \in T[1, n]$  THEN
    WriteString('w liegt in  $L(G)$ ');
ELSE
    WriteString('w liegt nicht in  $L(G)$ ');
END
```

Der Algorithmus besteht aus drei ineinander verschachtelten Schleifen, welche jeweils $O(n)$ viele Elemente durchlaufen. Der gesamte Algorithmus benötigt daher die Zeit $O(n^3)$.

Beispiel 13

Die Sprache $L = \{a^n b^n c^m | n, m \geq 1\}$ kann durch die kontextfreie Grammatik G mit

$$V_N = \{S, A, B\}, V_T = \{a, b, c\}, P = \{S \rightarrow AB, A \rightarrow ab|aAb, B \rightarrow c|cB\}$$

und $S = S$ erzeugt werden.

Die entsprechend Grammatik G' in CNF mit

$$V'_N = \{S, A, B, C, D, E, F\}, V'_T = V_T,$$

$$P' = \{S \rightarrow AB,$$

$$A \rightarrow CD|CF,$$

$$B \rightarrow c|EB,$$

$$C \rightarrow a,$$

$$D \rightarrow b,$$

$$E \rightarrow c,$$

$$F \rightarrow AD\}$$

und $S' = S$.

Für das Wort $w = aaabbbcc$ erzeugt der CYK-Algorithmus die folgende Tabelle:

		$i \rightarrow$						
$w =$	a	a	a	b	b	b	c	c
j	C	C	C	D	D	D	E, B	E, B
\downarrow			A				B	
			F					
		A						
		F						
	A							
	S							
	S							

3.5.4 Das Wortproblem für Typ-3 Grammatiken bzw. Sprachen

Durch den Beweis für kontextfreie bzw. Typ-2 Grammatiken ist bereits klar, dass das Wortproblem auch für reguläre bzw. Typ-3 Grammatiken bzw. Sprachen effizient (also in polynomieller Zeitkomplexität) gelöst werden kann. Während kontextfreie Grammatiken, die nicht regulär sind, mindestens die Komplexität $O(n^3)$ haben, kann das Problem für eine reguläre Grammatik G noch effizienter gelöst werden. Die Voraussetzung dazu ist, dass ein deterministischer endlicher Automat (DFA) bekannt ist, der genau die Sprache $L(G)$ als Eingabe akzeptiert. In [Sch09] wird nachgewiesen, dass zu jeder regulären bzw. Typ-3 Grammatik ein nichtdeterministischer endlicher Automat (NFA) existiert, der genau die Sprache $L(G)$ als Eingabe akzeptiert. Ebenso wird gezeigt, dass zu jedem NFA ein DFA existiert, der die gleiche Sprache akzeptiert. In der Umkehrung gibt es wiederum zu jedem DFA eine reguläre Grammatik, die die Sprache erzeugt, die der DFA als Eingabe akzeptiert. Dies führt zu einem Ringschluss, wie in Abbildung 3 noch einmal dargestellt ist.

Ist ein DFA, der genau die Sprache $L(G)$ als Eingabe akzeptiert, bekannt, so kann das Wortproblem für eine reguläre Grammatik G und ein Wort w der Länge n wie folgt mit der Komplexität $O(n)$ gelöst werden:

Verfolge Zeichen für Zeichen die Zustandsübergänge im Automaten, die durch die Eingabe w hervorgerufen werden. Falls ein Endzustand erreicht wird, gilt $w \in L(G)$.

Da die regulären Grammatiken im Rahmen dieser Arbeit nur als Spezialfall der kontextfreien Grammatiken behandelt werden, wird auf eine ausführliche Darstellung der Theorie zu endlichen Automaten und den entsprechenden Beweisen verzichtet und nur auf die oben angegebene Literatur verwiesen.

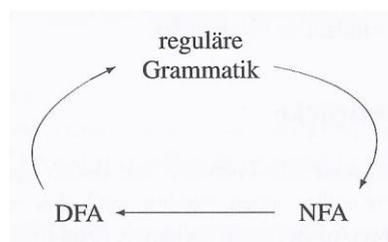


Abbildung 3: Ringschluss
Entnommen aus [Sch09], S. 27

3.6 Erweiterung der Mengenhierarchie der formalen Sprachen

Wie bisher in Abschnitt 3 dargestellt wurde, ergibt sich aus der Chomsky-Hierarchie eine weitere Einteilung für eine Teilmenge der formalen Sprachen. Zusammen mit der Frage nach der Entscheidbarkeit des Wortproblems für die verschiedenen Sprachtypen kann diese nun in die in Abschnitt 2.3.4 dargestellte Mengenhierarchie eingegliedert werden. Diese erweiterte Mengenhierarchie ist in Abbildung 4 in Form eines Mengendiagramms dargestellt.

In diesem Diagramm ist zu erkennen, dass die abzählbaren Sprachen gerade den Typ-0 Sprachen der Chomsky-Hierarchie entsprechen, während die kontextsensitiven (bzw. Typ-1) Sprachen eine echte Teilmenge der entscheidbaren Sprachen darstellen.

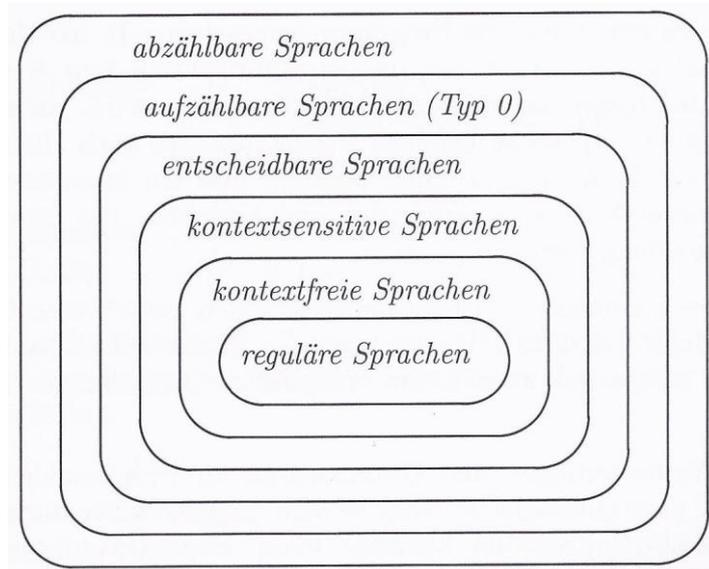


Abbildung 4: Erweiterung der Mengenhierarchie.
Entnommen aus [He12], S. 37

4 Die Simulationsumgebung: GrammatikSimulator

4.1 Vergleich zu alternativen Simulationsumgebungen

Bevor es im Weiteren um die für diese Arbeit erstellte Simulationsumgebung geht, möchte ich zuerst einen kurzen Überblick über eine kleine Auswahl von alternativen Simulationsprogrammen geben, die bereits existieren. Dies ermöglicht vor allem eine Vergleichbarkeit bei der Darstellung der Zielsetzung für die neue Simulationsumgebung.

4.1.1 JFLAP

Das Programm JFLAP [JFL], dessen Name sich als Abkürzung von Java Formal Language and Automata Package herleitet, existiert bereits seit Mitte der 1990er Jahre und wurde seit 1995 an der Duke University in den USA weiterentwickelt. Die Weiterentwicklung des Programms wurde dabei immer wieder von wechselnden Studenten durchgeführt und so die Funktionen des Programms erweitert und verändert.

JFLAP ist ein Tool zur graphischen Darstellung von Automaten, Grammatiken und weiteren Konzepten aus der Theorie der Automaten und formalen Sprachen und soll zur Unterstützung beim Erlernen dieser Konzepte dienen.

Für Grammatiken ist die Erstellung von entweder Kontextfreien Grammatiken oder unbeschränkten Grammatiken (womit anscheinend Typ-0 Grammatiken gemeint sind) möglich. Für Kontextfreie Grammatiken können mit verschiedenen Parser-Algorithmen Worte auf Zugehörigkeit zur Sprache der Grammatik getestet werden. Dazu wird auch ein entsprechender Ableitungsbaum erzeugt und angezeigt. Es wird jedoch wohl immer nur ein möglicher Baum erzeugt, so dass eine eventuelle Mehrdeutigkeit der Grammatik nicht ersichtlich wird. Für Typ-0 Grammatiken steht ein Brute-Force Parser zur Verfügung, der auch eine Art Ableitungsbaum erstellt, jedoch ist diese Darstellung meiner Ansicht nach schwer nachvollziehbar. Die Eingabe der Grammatiken erfolgt über die Eingabe der Produktionen, die Mengen der Variablen und Terminale sowie das Startsymbol können nicht explizit definiert werden. Die so erstellten Grammatiken können gespeichert werden und entsprechend können auch gespeicherte Grammatiken geladen werden.

Zudem ist JFLAP nur in englischer Sprache verfügbar, was für den Einsatz im Unterricht in Deutschland von Nachteil ist, da die Schülerinnen und Schüler nicht immer über ausreichend gute Englischkenntnisse verfügen.

Weiterhin können mit JFLAP keine Ableitungen manuell ausgeführt werden, sondern nur die Ableitungen für die geprüften Worte schrittweise nachverfolgt werden. Und auch die Erstellung einer Liste der Worte bis zu einer vorgegebenen Wortlänge ist nicht möglich.

Dafür bietet JFLAP jedoch umfangreiche Möglichkeiten zur Transformation der Grammatiken, wahlweise in ein Automatenmodell oder auch in die Chomsky-Normal-Form.

4.1.2 Machines

Das Programm Machines [MAC] wurde von Prof. Rolf Socher an der FH Oldenburg/Ostfriesland/Wilhelmshaven entwickelt und ist mittlerweile in der Version 1.5 verfügbar.

Machines ist ein Lern- und Visualisierungsprogramm für Automaten, reguläre Ausdrücke und Grammatiken, und die Beschriftungen im Programm sind im Gegensatz zu JFLAP in deutscher Sprache. Für Grammatiken bietet Machines die Möglichkeit, Grammatiken aller Typen zu erstellen. Die Erstellung erfolgt auch bei Machines wieder über die Eingabe der Produktionen, wobei das Startsymbol am Ende explizit festgelegt wird. Die Produktionen sind in Machines, in Anlehnung an die Backus-Naur-Form, so einzugeben, dass Variablen durch spitze Klammern („<>“) gekennzeichnet sind. Auch dieses Programm bietet die Möglichkeit, erstellte Grammatiken zu speichern und umgekehrt Grammatiken aus einer Datei zu laden.

Mit einer erstellten oder geladenen Grammatik können nun Eingabeworte auf Zugehörigkeit zur Sprache geprüft werden oder eine Liste der Worte bis zu einer vorgegebenen Wortlänge erstellt werden.

Machines bietet dabei jedoch keine Darstellung der Ableitungsbäume. Entsprechend gibt es auch keine Möglichkeit, eine Ableitung schrittweise manuell auszuführen.

4.1.3 AtoCC

AtoCC [ATCC] ist eine mittlerweile weit verbreitete Lernumgebung für Inhalte der theoretischen Informatik, die sich stark am Thema Compilerbau orientiert. Die ersten Versionen des Programms wurden von Dr. M. Hieschler bereits während seiner Studienzeit entwickelt und immer weiter ausgebaut. Da AtoCC eine über Jahre gewachsene Lernumgebung ist, bietet sie in ihrer aktuellen Version relativ umfassende Möglichkeiten zur Erstellung, Simulation, graphischen Darstellung und Transformation von verschiedenen Automatentypen, regulären Ausdrücken, kontextfreien Grammatiken und spezifisch für Compilerbau noch weitere Elemente. Das Programm unterteilt sich in sieben relativ eigenständige Module, deren Ergebnisse aber auch mitunter zur Anwendung in anderen Modulen transformiert werden können.

Das Modul kfgEdit für kontextfreie Grammatiken (und somit selbstverständlich auch reguläre Grammatiken) bietet die Möglichkeit, Grammatiken selbst zu erstellen, zu speichern und zu laden. Darüber hinaus können auch aus in anderen Modulen erstellten Automaten oder regulären Ausdrücken die entsprechenden Grammatiken generiert werden und in kfgEdit verwendet werden. In der Umkehrung kann aus einer Grammatik auch ein entsprechender Automat erzeugt werden.

Die Erstellung eigener Grammatiken erfolgt auch in AtoCC über die Eingabe der Produktionen. Die Mengen der Variablen und Terminale und das Startsymbol werden also auch hier nur implizit aus den Produktionen bestimmt. Im Gegensatz zu den anderen vorgestellten Programmen bietet AtoCC zumindest die Möglichkeit, die formale Definition zur eingegebenen Grammatik anzeigen zu lassen. Neben der Umwandlung in einen Automaten können erstellte Grammatiken auch in die Chomsky-Normal-Form überführt werden und die Darstellung der Produktionen verändert werden. Außerdem kann die Grammatik auf Richtigkeit, oder ob es sich um eine reguläre Grammatik handelt, geprüft werden.

Für die Simulation von Grammatiken bietet AtoCC die Möglichkeit, Worte zu prüfen, und bei Zugehörigkeit zur Sprache der Grammatik wird der entsprechende Ableitungsbaum angezeigt. Hierbei werden für mehrdeutige Grammatiken alle möglichen Ableitungsbäume gebildet und können angezeigt werden.

Weiterhin können auch manuelle Ableitungen durchgeführt werden, jedoch ist hier nur die Ausführung von Linksableitungen in einzelnen Schritten möglich. Wie bereits erwähnt, ist AtoCC nur auf die Verarbeitung von Typ-2 und Typ-3 Grammatiken ausgelegt und bietet somit keinerlei Möglichkeiten für Typ-1 oder sogar eventuell Typ-0 Grammatiken. Auch das Generieren von Worten bis zu einer maximalen Wortlänge ist in AtoCC nicht möglich.

4.2 Zielsetzung für die Simulationsumgebung

Für die Simulation von Grammatiken existieren bereits einige Programme, die in der Summe die meisten Funktionen im Umgang mit Grammatiken bereitstellen. Keines der Programme vereint dabei jedoch alle wünschenswerten Funktionen in einem Programm.

Die neue Simulationsumgebung soll nun eine möglichst große Auswahl der Funktionen, die auch andere Programme bieten, in einem Programm bündeln und, sofern technisch machbar, weitere Funktionen bieten, die noch nicht in einem anderen Programm in der Form vorhanden sind.

Mit den Möglichkeiten der bereits existierenden Programme im Bewusstsein, ergaben sich die folgenden Funktionen als anvisiertes Ziel für das neue Programm:

- Arbeit mit möglichst allgemeinen Grammatiken:

Der Typ der Grammatiken in der Chomsky-Hierarchie, mit denen das Programm arbeiten kann, soll möglichst allgemein sein. Nach Möglichkeit sollten überhaupt keine Einschränkungen an die Form der Grammatiken gestellt werden. Wie später noch ausgeführt wird, zeigte sich relativ früh, dass diese Zielsetzung nicht zu erreichen war.

- Möglichkeit zum Erstellen von eigenen Grammatiken:

Diese Funktion ist im gewissen Sinne für ein solches Simulationsprogramm natürlich selbstverständlich. Denkbar wäre zwar auch eine Art Bibliothek mit einer größeren Auswahl an fertigen Grammatiken, aber dies wäre im Vergleich zu den existierenden Programmen ein Rückschritt.

Die Erstellung einer Grammatik erfolgt bei den alternativen Programmen fast ausschließlich über das Definieren der Produktionen, während die Mengen der Variablen und Terminale automatisch aus den Produktionen bestimmt werden. Diese Vorgehensweise ermöglicht zwar ein schnelles Erstellen der Grammatiken, bietet aber keine Möglichkeit zur vertieften Auseinandersetzung mit der formalen Definition einer Grammatik. Im neuen Programm sollte daher die Erstellung einer Grammatik stärker die Definition der Mengen der Variablen und Terminale sowie die Festlegung des Startsymbols betonen. Auf diesem Weg setzen sich die Schülerinnen und Schüler genauer mit der Struktur einer Grammatik auseinander und können so ihre Kenntnis über die formale Definition einer Grammatik festigen.

Zudem sollte es möglich sein, erstellte Grammatiken zu speichern und zu laden, so dass auch mit fertigen Beispielen gearbeitet werden kann.

- Manuelle, schrittweise Ausführung von Ableitungen:

Den Schülerinnen und Schülern soll es möglich sein, in beliebiger Reihenfolge und in einzelnen Schritten Ableitungen auszuführen, um die Ableitung für ein Wort im Einzelnen nachvollziehen zu können und verschiedene Reihenfolgen der einzelnen Schritte vergleichen zu können. Dabei soll in jedem Schritt der entsprechende Ableitungsbaum gezeigt werden, in dem auch einzelne Knoten selektiert werden können, um diese als Ausgangspunkt (sofern zulässig) für den nächsten Ableitungsschritt zu wählen.

Des Weiteren sollen auch in jedem Schritt die entsprechende Linksableitung und Rechtsableitung ausführbar sein, ohne die entsprechende Variable bzw. den sie repräsentierenden Knoten im Baum manuell auswählen zu müssen.

Die Menge der für die betreffende Variable zulässigen Produktionen soll dabei in allen Fällen automatisch vom Programm bestimmt werden und den Schülerinnen und Schülern zu Auswahl gegeben werden.

Um bei einem falschen bzw. ungewollten Schritt nicht wieder von vorne beginnen zu müssen, muss natürlich auch die Möglichkeit bestehen, alle Schritte einzeln rückgängig zu machen.

- Worte prüfen und Ableitungsbaum erstellen lassen:

Verkürzt gesagt, soll das Programm das Wortproblem lösen können. Genauer gesagt, sollen die Schülerinnen und Schüler ein Wort eingeben können, und das Programm soll dann bestimmen, ob dieses Wort zur von der Grammatik erzeugten Sprache gehört. Darüber hinaus soll, falls das Wort zur Sprache gehört, auch der Ableitungsbaum bestimmt werden, so dass ersichtlich wird, wie das Wort gebildet wird. Dabei sollen für mehrdeutige Grammatiken auch alle verschiedenen Ableitungsbäume für das Wort gebildet und zur Anzeige bereitgestellt werden.

- Wörter unter Vorgabe einer maximalen Länge erzeugen lassen:

Neben der Möglichkeit, einzelne Wörter zu prüfen und ihre Ableitungsbäume erzeugen zu lassen, soll es auch möglich sein, eine maximale Zeichenlänge vorzugeben und dann die Wörter der Sprache, deren Länge kleiner oder gleich der Vorgabe ist, sowie deren Ableitungsbäume erzeugen zu lassen.

Eine technische Anforderung für die Simulationsumgebung ist, dass das Programm auf möglichst vielen verschiedenen Systemen genutzt werden kann. Zudem sollen keine ungewöhnlichen zusätzlichen Treiber oder andere Programme zur Ausführung des Programms notwendig sein. Die Wahl der Programmiersprache fiel deshalb auf Java. Java ist auf allen gängigen Betriebssystemen lauffähig und bedarf dazu nur der Installation der passenden Java-
Runtime-Environment, die zum Standard auf den meisten Schulsystemen gehört. Zudem bietet Java die Möglichkeit, das erstellte Programm bequem in der Form eines JAR-Archives zu veröffentlichen, das keine weitere Installation des Programms notwendig macht.

4.3 Umsetzung der Darstellung von Ableitungsbäumen für kontextsensitive Grammatiken

In Abschnitt 3.4 dieser Arbeit wurde bereits ausgeführt, dass Ableitungsbäume formal nur für Typ-2 (bzw. indirekt auch für Typ-3) Grammatiken definiert sind. Die Anwendung dieser Definition ist strenggenommen für Typ-1 oder gar Typ-0 Grammatiken nicht möglich.

Andererseits wurde im vorherigen Abschnitt 5.2 nun als Ziel angegeben, dass das Programm die Arbeit mit möglichst allgemeinen Grammatiken gestatten soll.

Wie bereits angedeutet, war eine Umsetzung ohne Einschränkung, also die Möglichkeit, mit Typ-0 Grammatiken arbeiten zu können, nicht realisierbar. Da die Ableitungen bei Typ-0 Grammatiken keinen Beschränkungen unterliegen, fand sich kein Ansatz, um die Ableitung eines Wortes als Baum zu modellieren. Die graphische Darstellung der Ableitung als Baum ist jedoch die zentrale Funktion des Programms und das Kernthema dieser Arbeit. Außerdem wurde unter Abschnitt 3.5 bereits erwähnt, dass das Wortproblem für Typ-0 Grammatiken nicht berechenbar ist. Zwei der Hauptfunktionen des Programms sind also für Typ-0 Grammatiken nicht (sinnvoll) umsetzbar.

Für Typ-1 Grammatiken zeigte sich, dass die Umsetzbarkeit abhängig ist von der Definition, die zugrunde gelegt wird, worauf im Folgenden noch genauer eingegangen wird. Zudem ist für Typ-1 Grammatiken das Wortproblem, wie in Abschnitt 3.5 bewiesen wurde, berechenbar. Es liegt aber in der Komplexitätsklasse NP-Hart, und somit ist ein exponentielles Laufzeitverhalten für den Algorithmus unumgänglich. Dies ist im schulischen Kontext jedoch vertretbar, da die verwendeten Beispiele in der Regel klein genug sind, um auch von einem Algorithmus mit exponentieller Laufzeit in einer vertretbaren Zeit bearbeitet zu werden.

Wie bereits gesagt, liegt die Kernfunktion des Programms in der graphischen Darstellung der Ableitung in Form von Ableitungsbäumen. Für Typ-1 Grammatiken sind dazu entsprechende Einschränkung bzw. die Zugrundelegung der passenden Definition notwendig.

In der Literatur werden die Typ-1 Grammatiken in der Chomsky-Hierarchie oftmals durch die Beschränkung der Produktionen in der Form

$$w_1 \rightarrow w_2 \text{ mit } |w_1| \leq |w_2|, w_1, w_2 \in (V_N \cup V_T)^+$$

definiert. Dies macht z.B. Produktionen der Form

$$AB \rightarrow cde \text{ mit } A, B \in V_N, c, d, e \in V_T$$

oder auch

$$Ab \rightarrow cD \text{ mit } A, D \in V_N, b, c \in V_T$$

zulässig.

In solchen Fällen kann man die Knoten für die neuen Zeichen nicht als Kinder einem einzelnen Vater-Knoten zu ordnen. Zudem wäre es notwendig, einzelne Knoten in gewisser Weise wieder verschwinden lassen zu können.

In Abschnitt 3.3, Definition 9 wurde deshalb bereits für Typ-1 Grammatiken die folgende, ebenfalls gebräuchliche, Einschränkung für die Form der Produktionen angegeben:

$$\alpha_1 A \alpha_2 \rightarrow \alpha_1 \beta \alpha_2 \text{ mit } A \in V_N, \alpha_1, \alpha_2 \in V^*, \beta \in V^+.$$

Bei dieser Form der Produktionen wird immer ein Variablen-Zeichen durch eine neue Zeichenfolge ersetzt, so dass, wie bei Typ-2 Grammatiken, eine eindeutige Vater-Kind Beziehung für die Knoten eines Ableitungsbaumes gegeben ist.

Für den Gebrauch im Schulunterricht sollte der Ableitungsbaum so dargestellt werden, dass einzelne Schritte in der Ableitung möglichst gut nachvollziehbar sind. Deshalb ist es bei den Typ-1 Grammatiken noch erforderlich, den Kontext, der bei einer Ableitung „auftritt“, geeignet darzustellen. Insbesondere soll erkenntlich sein, welche Knoten im Baum bei der Anwendung einer Produktion als Kontext interpretiert wurden, um nachträglich nachvollziehbar zu machen, wann die Anwendung dieser Produktion zulässig war. Dabei genügt es nicht, diese nur farbig oder in anderer Weise zu markieren, da bei mehreren Ausführungen von Produktionen mit Kontext nicht mehr nachvollziehbar wäre, welche Knoten zu welchem Schritt in der Ableitung gehören.

Für das Programm wurde von mir deshalb folgende Modellierung zur Darstellung der Ableitungen bei kontextsensitiven bzw. Typ-1 Grammatiken gewählt:

Die Ableitungsbäume werden weitestgehend analog zu den Ableitungsbäumen von Typ-2 Grammatiken erzeugt. Für neue Knoten, die aus der Anwendung einer Produktion mit Kontext resultieren, werden zum einen die Knoten im Baum, die als Kontext interpretiert wurden, farblich gekennzeichnet. Des Weiteren werden die als Kontext interpretierten Knoten als Kopie zusammen mit den neuen Knoten nochmals in den Baum eingefügt. Die Originale der Knoten werden dabei inaktiv bzw. passiv geschaltet. Wurde zum Beispiel eine Variable als Teil des Kontextes interpretiert, werden Ersetzungen für diese Variable an die Kopie angefügt. Die Kopien werden dabei so mit eingefügt, dass sie in der korrekten Ordnung links und rechts der neuen Knoten stehen.

Zum besseren Verständnis möchte ich dies noch an einem konkreten Anwendungsbeispiel mit dem Programm darstellen.

Beispiel 14:

Wir nehmen die kontextsensitive Grammatik $G = (V_N, V_T, P, S)$ mit

$$V_N = \{S, A, B, C\}, V_T = \{a, b, c\},$$

$$P = \{S \rightarrow aAb, A \rightarrow abc, aAb \rightarrow aBCb, aB \rightarrow ac, Cb \rightarrow Ab\}$$

und $S = S$.

Das Wort $w = acabcb$ kann wie folgt abgeleitet werden:

$$S \Rightarrow aAb \Rightarrow aBCb \Rightarrow acCb \Rightarrow acAb \Rightarrow acabcb$$

Diese Ableitung wird im Programm durch den in Abbildung 5 gezeigten Ableitungsbaum graphisch dargestellt. Die als Kontext interpretierten Knoten sind rot eingefärbt und das Wort w ist an den blau gefärbten ovalen Knoten von links nach rechts abzulesen.

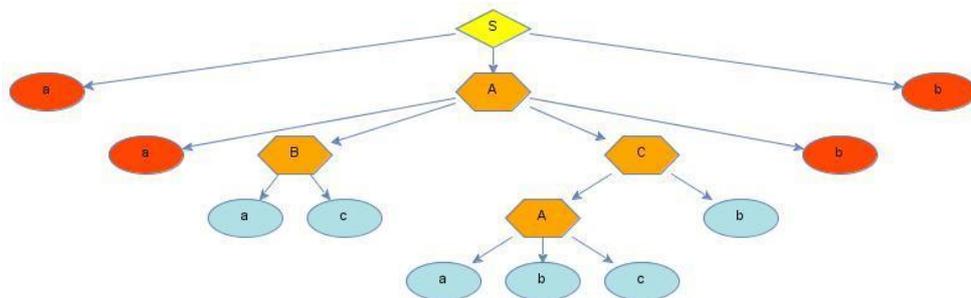


Abbildung 5: Ableitungsbaum für das Wort $acabcb$ aus Beispiel 14

4.4 Überblick über die Realisierung der Simulationsumgebung

In diesem Teilabschnitt möchte ich einen Überblick über die wichtigsten Entscheidungen zur Realisierung der Simulationsumgebung geben. Dazu werde ich im Folgenden zunächst die wichtigsten verwendeten Datenstrukturen darstellen, danach den Editor für das Erstellen und Bearbeiten von Grammatiken erläutern und die verwendete Bibliothek für die Visualisierung der Ableitungsbäume vorstellen. Darauffolgend werden die zentralen Algorithmen für die Ausführung der Ableitungen verkürzt dargestellt, um einen Einblick in die Funktionsweise der Simulationsumgebung zu geben. Abschließend werde ich noch den Aufbau der Benutzeroberfläche im Hauptfenster erläutern.

4.4.1 Die Datenstrukturen für Grammatiken und Ableitungsbäume

4.4.1.1 Die Datenstruktur für Grammatiken

Bekannterweise ist die Erstellung von Datenstrukturen in Java nur durch Klassen möglich. Im Folgenden möchte ich einen kurzen Überblick über die zu diesem Zweck erstellten Klassen geben.

In den verschiedenen Teilen des Programms ist immer wieder der Zugriff auf einzelne Teile der aktuellen Grammatik notwendig. Um diese Zugriffe sinnvoll zu strukturieren, wird eine Grammatik im Programm als Ganzes in einer entsprechenden Datenstruktur modelliert. Eine Grammatik kann so übersichtlich als einzelnes Objekt im Programm verwaltet werden.

Dabei werden vor allem verkettete Listen genutzt, um die Mengen der Variablen, Terminale und Produktion einer Grammatik zu speichern. Hierbei stellt Java mit der Klasse `LinkedList` eine entsprechende Implementierung bereit, die auch hilfreiche Methoden, wie die Prüfung ob ein Objekt bereits in der Liste enthalten ist, bereitstellt.

Im Einzelnen besteht die Struktur aus folgenden Elementen:

In einem String wird eine Benennung der Grammatik hinterlegt, die dem User angezeigt werden kann. Die Mengen der Variablen und Terminale werden einfach als Listen von Strings gehalten. Hierbei wird bewusst String statt Char verwendet, da so auch Worte im Sinne natürlicher Sprachen als Zeichen in einer Grammatik verwendet werden können. Das Startsymbol ist entsprechend wieder ein einfacher String.

Die einzelnen Produktionen in einer Grammatik haben selbst wieder eine komplexere Struktur. Diese wurden daher wiederum als eigene Datenstruktur modelliert, um so die algorithmische Verarbeitung zu erleichtern. Auf die entsprechende Klasse Produktion werde ich im Folgenden noch etwas näher eingehen. In der Erstellung eines Objektes der Klasse Grammatik wird für den Anwender nicht sichtbar geprüft, ob die Grammatik nur Produktionen ohne Kontext besitzt. Diese Prüfung wird im Grammatikeditor durchgeführt, der in einem späteren Abschnitt noch ausführlich dargestellt wird. Da diese folglich kontextfrei ist, wird die Grammatik entsprechend über die Variable kfg als kontextfrei markiert. Im Programm wird dies an verschiedenen Stellen genutzt, um Algorithmen in einer einfacheren bzw. effizienteren Fassung zu verwenden, die aber nur für kontextfreie Grammatiken gültig sind. Dies wird bei der Erläuterung der Realisierung der Funktionen des Programms genauer ausgeführt. Da erstellte Grammatiken auch gespeichert werden, wird in der Klasse das Interface „Serializable“ implementiert. Dadurch können Objekte der Klasse Grammatik einfach als Bytestrom serialisiert und in einer Datei gespeichert werden. Umgekehrt kann der Bytestrom einfach wieder aus der Datei gelesen werden und durch Deserialisierung wieder das Objekt der Klasse Grammatik erzeugt werden.

Im folgenden Auszug aus dem Quellcode der Klasse Grammatik wurden die Set- und Get-Methoden für die Variablen der Klasse ausgelassen. Diese haben die in Java übliche Gestalt und wurden daher für eine bessere Übersicht in diesem Auszug ausgelassen.

4.4.1.2 Auszug aus der Klasse Grammatik.java

```
public class Grammatik implements Serializable {

    private String name;
    private String startsymbol;
    private LinkedList<String> variablen;
    private LinkedList<String> terminale;
    private LinkedList<Produktion> produktionen;
    private boolean kfg;

    public Grammatik(String startsymbol, LinkedList<String> variablen,
        LinkedList<String> terminale,
        LinkedList<Produktion> produktionen, boolean kfg) {
        this.startsymbol = startsymbol;
        this.variablen = variablen;
        this.terminale = terminale;
        this.produktionen = produktionen;
        this.kfg = kfg;
    }

    :

}
```

4.4.1.3 Die Datenstrukturen für Produktionen

Wie bereits erwähnt, werden auch einzelne Produktionen durch eine eigene Klasse modelliert.

Eine Produktion kann im Groben in die linke und die rechte Seite unterteilt werden. Die linke Seite besteht aus der Variablen, die als einfacher String gehalten werden kann, und gegebenenfalls dem Kontext, der noch einmal in den Kontext links der Variablen und rechts der Variablen unterteilt werden kann. In genau dieser Unterteilung wird die linke Seite auch durch die Klasse modelliert. Der Kontext wird dabei als Erleichterung für die Verwendung im Programm einmal als einfache Strings gehalten und einmal zeichenweise in verketteten Listen. Die Listen werden bei der Überprüfung der Produktionen bei der Erstellung im Grammatikeditor benötigt. Für die rechte Seite werden die Zeichen, mit denen die Variable zu ersetzen ist, wieder einzeln in einer verketteten Liste gehalten. Da zu den Zeichen zusätzliche Informationen anfallen,

wurde hierfür noch eine weitere kleine Klasse für die Datenstruktur angelegt. In der Klasse RSProdzeichen wird ein Zeichen aus dem Alphabet der Grammatik modelliert, in dem das eigentliche Zeichen als String gehalten wird und als Ganzzahl kodiert der Typ (Variable, Terminal, Startsymbol oder Leerwort). Dazu wird wieder zur Erleichterung in der Programmierung in einer booleschen Variablen vermerkt, ob dieses Zeichen ein Variablenzeichen ist, das in einem späteren Schritt wieder ersetzt werden darf.

Ähnlich wie bei den Grammatiken wird bei den Produktionen in einer booleschen Variablen vermerkt, ob bei dieser Produktion ein Kontext existiert, der geprüft werden muss, bevor die Produktion angewendet werden kann. Dies geschieht auch hier aus dem Grund, damit unnötige Schritte in den Algorithmen ausgelassen werden können.

Zudem gibt es zwei Konstruktoren, die sich darin unterscheiden, ob ein Kontext übergeben wird oder nicht. Es ist möglich, dass Produktionen mit Sicherheit kontextfrei sind, für die dann der Konstruktor ohne Parameter für den Kontext genutzt wird. In den allgemeineren Fällen wird der Konstruktor mit Übergabe des Kontextes verwendet. Die Prüfung für den Wert von „kontext“ erfolgt natürlich nur, wenn ein solcher übergeben werden kann, ansonsten wird „kontext“ immer auf falsch gesetzt.

4.4.1.4 Auszüge aus den Klassen Produktion.java und RSProdZeichen.java

```
public class Produktion implements Serializable {

    private String var;
    private String kontext_l;
    private String kontext_r;
    private LinkedList<String> konl_list;
    private LinkedList<String> konr_list;
    private LinkedList<RSProdZeichen> rechteseite;
    public boolean kontext;

    public Produktion(String var, String kont_l, LinkedList<String> konl_list,
String kont_r, LinkedList<String> konr_list, LinkedList<RSProdZeichen> rs)
    {
        this.var = var;
        this.kontext_l = kont_l;
        this.konl_list=konl_list;
        this.kontext_r = kont_r;
        this.konr_list=konr_list;
        this.rechteseite = rs;
        if (kontext_l.length() > 0 || kontext_r.length() > 0) {
            kontext = true;
        } else {
            kontext = false;
        }
    }

    public Produktion(String var, LinkedList<RSProdZeichen> rs) {
        this.var = var;
        this.rechteseite = rs;
        kontext = false;
    }

    :
}
```

⋮

```
@Override
public String toString() {
    String s = new String();
    if (!kontext_l.equals("")) {
        s = s + kontext_l + "--";
    }
    s = s + var;
    if (!kontext_r.equals("")) {
        s = s + "--" + kontext_r;
    }
    s = s + " -->";
    if (!kontext_l.equals("")) {
        s = s + kontext_l + "--";
    }
    for (int i = 0; i < rechteSeite.size(); i++) {
        s = s + (rechteSeite.get(i)).zeichen;
    }
    if (!kontext_r.equals("")) {
        s = s + "--" + kontext_r;
    }
}
```

```
public class RSProdZeichen implements Serializable {

    public String zeichen;
    public int typ;
    public boolean ableitbar;

    public RSProdZeichen(String zeichen, int typ, boolean ableitbar) {
        this.zeichen = zeichen;
        this.typ = typ;
        this.ableitbar = ableitbar;
    }
}
```

4.4.1.5 Die Datenstrukturen für Ableitungsbäume

Als Datenstruktur für Ableitungsbäume wird die Java eigene Klasse `DefaultTreeModel` verwendet. Diese wird normaler Weise als Model für die Daten in einem `JTree` verwendet. Nach der Verwendung eines `JTree` zur Visualisierung der Ableitungsbäume im Prototypen Stadium des Programms wurde die Klasse `DefaultTreeModel` weiter verwendet, da diese benötigte Methoden bereits komfortabel bereitstellt. Ein Beispiel für solche Methoden ist die Erzeugung einer Enumeration der Knoten, wahlweise in Post- oder BreathFirst-Order. Diese Darstellung der Knoten wird für die Algorithmen zum manuellen Ableiten, Worte prüfen und Worte erzeugen mehrfach benötigt.

Da eine eigene Baum-Klasse im Grunde nur ein Nachbau von `DefaultTreeModel` wäre, wurde darauf verzichtet.

Um in den Knoten die zusätzlichen notwendigen Informationen halten zu können, wird mit der Klasse `Ableitungsknoten` eine entsprechende Datenstruktur modelliert, die in den Knoten des Baumes als sogenanntes User-Object gehalten werden kann. Einem Knoten im Ableitungsbaum kann auf diesem Wege hinzugefügt werden, für welches Zeichen er steht, welchen Typ (analog zur in `RSProdZeichen` gemachten Modellierung) dieses Zeichen hat, ob diesem Knoten noch weitere Knoten angehängt werden können, und ob er als Kontext interpretiert wurde.

Für den einfacheren Gebrauch gibt es zwei Konstruktoren: In der einen Version wird „kontext“ auf den Standard-Startwert falsch gesetzt, da dieser Wert in der Regel nur bei bereits erstellten Knoten auf wahr gesetzt wird. Da aber auch in den Algorithmen Kopien von bestehenden Bäumen erzeugt werden müssen, gibt es noch den zweiten Konstruktor mit dem Knoten mit einem entsprechenden Wert für „kontext“ belegt werden können.

Als Methoden enthält die Klasse im Wesentlichen nur die üblichen Set- und Get-Methoden.

4.4.1.6 Auszug aus der Klasse Ableitungsknoten.java

```
public class Ableitungsknoten {

    private String zeichen;
    private int typ;
    private boolean ableitbar;
    private boolean kontext;

    public Ableitungsknoten(String zeichen, int typ, boolean ableitbar) {
        this.zeichen = zeichen;
        this.typ = typ;
        this.ableitbar = ableitbar;
        kontext = false;
    }

    public Ableitungsknoten(String zeichen, int typ, boolean ableitbar,
                             boolean kontext) {

        this.zeichen = zeichen;
        this.typ = typ;
        this.ableitbar = ableitbar;
        this.kontext = kontext;
    }

    :

    @Override
    public String toString() {
        return zeichen;
    }
}
```

4.4.2 Benutzeroberfläche und Funktionsweise des Grammatikeditors

Mit dem Editor werden im Programm zwei wichtige Funktionen realisiert: Zum einen das Erstellen neuer Grammatiken und zum anderen das Editieren von bereits bestehenden Grammatiken. Der Editor ist ein gesondertes Fenster im Programm, das auf Basis eines JDialog erstellt wurde. Auf diesem Weg erhält der Editor vor allem den notwendigen Platz für eine übersichtliche Eingabe, und zudem können so Eingaben im eigentlichen Programm blockiert werden, während der Editor geöffnet ist. Dadurch ist gewährleistet, dass es nicht zu Fehlern im Programm durch eine falsche Bedienung während der Bearbeitung einer Grammatik kommt, und es wird die Überprüfung der Eingabe der Grammatik erleichtert.

Eine Ansicht des Editor-Fensters im Startzustand für die Erstellung einer Grammatik ist nachfolgend in Abbildung 6 zu sehen. Die Benutzeroberfläche im Editor-Dialog kann grob in drei Bereiche unterteilt werden. Das linke obere Viertel beinhaltet die Elemente zur Eingabe, die rechte Hälfte die Elemente zur Anzeige der bisherigen Eingaben. Im linken unteren Viertel wird eine Kurzanleitung angezeigt, und darunter liegen noch die Buttons, um den Editor zu beenden.

Zunächst möchte ich auf die Elemente im Viertel oben links eingehen: Über Textfelder können die Zeichen für die Variablen- und Terminalmengen eingegeben werden. Mit den darunter liegenden Buttons wird jeweils das Hinzufügen zu den Listen ausgelöst, wobei geprüft wird, dass neue Variablen nicht schon als Terminale definiert sind und umgekehrt. Sobald mindestens eine Variable festgelegt wurde, kann über die JComboBox das Startsymbol festgelegt werden. Wie bereits bei den Zielvorgaben für das Programm festgelegt wurde, erfolgt die Eingabe der Variablen- und Terminalzeichen nicht indirekt über die Eingabe der Produktionen. Die Variablen- und Terminalmenge wird dadurch stärker in den Fokus gerückt, und die Bedeutung und Unterschiede dieser Mengen in einer Grammatik sollen so besser aufgezeigt werden.

Während der Erstellung werden die Mengen jeweils in einem JList-Element gehalten und angezeigt, womit ich auch zu den Elementen in der rechten Hälfte der Benutzeroberfläche kommen möchte. Dort sind entsprechend drei JList-Elemente zu sehen, in denen Variablen, Terminale und Produktionen angezeigt werden. Unter den Listen befindet sich jeweils ein Button, mit dem sich ein

ausgewähltes Zeichen bzw. eine Produktion aus der Liste wieder entfernen lässt, um fehlerhafte Eingaben wieder korrigieren zu können.

Bei der Initialisierung des Editors wird zur Liste für die Terminale automatisch das Zeichen ϵ hinzugefügt, das im Programm das Leerwort repräsentiert. Zusätzlich gibt es unter dem Textfeld für die Eingabe der Terminale den Button „Epsilon hinzufügen“, um es erneut hinzuzufügen zu können, falls es gelöscht wurde.

Unter der Eingabe für die Variablen und Terminale findet sich noch die Eingabe für die Produktionen. Die Eingabe ist dabei auf mehrere Eingabeelemente verteilt, um die grundlegende Richtigkeit der Eingabe überprüfen zu können. Die Erstellung der Produktionen wird erst dann möglich, wenn die entsprechenden Variablen und Terminale zu den Listen hinzugefügt wurden. Unter anderem wird die Variable, die in der Produktion ersetzt wird, über eine JComboBox festgelegt, deren Elemente über die JList der Variablen generiert wird. Die Zeichenfolge für die rechte Seite der Produktion wird über ein Textfeld eingegeben, wobei zwischen den einzelnen Zeichen des Grammatikalphabets ein Leerzeichen erforderlich ist. Diese Leerzeichen werden benötigt, da die Zeichenfolge beim Hinzufügen in eine Liste der einzelnen Zeichen umgewandelt wird. Dies erfolgt wiederum, um für jedes Zeichen zu prüfen, dass es in der Liste der Terminale oder Variablen enthalten ist. Diese Prüfung wird beim Hinzufügen jeder einzelnen Produktion ausgeführt, und es wird eine entsprechende Fehlermeldung ausgegeben, wenn die Produktion ungültige Zeichen enthält. Umgekehrt werden, wenn Elemente aus den Listen der Variablen- und Terminalzeichen gelöscht werden, alle bereits erstellten Produktionen wieder geprüft, da sie nun ungültige Zeichen enthalten könnten. Ist dies der Fall, werden die entsprechenden Produktionen aus der Liste entfernt und eine passende Meldung angezeigt.

In das Textfeld für die rechte Seite können Zeichen auch hinzugefügt werden, in dem man einen Doppelklick auf das gewünschte Zeichen in der JList für Variablen bzw. der für Terminale ausführt. Dabei wird auch automatisch das notwendige Leerzeichen mit eingefügt.

Über die unter den Eingabefeldern liegende JCheckBox kann die Eingabe des Kontextes für Produktionen aktiviert werden. Wenn die Eingabe erlaubt ist, wird der Kontext analog zur rechten Seite der Produktion eingegeben. Dabei

gibt es entsprechend jeweils ein Textfeld für einen möglichen Kontext links der Variablen und rechts der Variablen. Auch beim Kontext wird geprüft, dass nur zulässige Zeichen verwendet werden. Das Hinzufügen von Zeichen in die Textfelder durch Doppelklick auf die Listen ist hier jedoch nicht möglich. Um die Eingabe zu erleichtern, muss der Kontext für die rechte Seite nicht noch einmal eingegeben werden. Die entsprechenden Textfelder werden vom Programm automatisch an die Eingaben in den Textfeldern für die linke Seite angepasst.

Beim Hinzufügen einer Produktion zur Liste wird, nach der erwähnten Prüfung der Gültigkeit der Zeichen, direkt ein Objekt der Klasse Produktion erstellt. Für die Anzeige in der Liste wurde in der Klasse entsprechend die Methode *toString()* überschrieben.

Sind die Variablen, Terminale, Produktionen und das Startsymbol vollständig angelegt, kann der Anwender den Editor über den Button „Fertig“ beenden. Es erfolgt eine abschließende Prüfung, ob die Daten für die Grammatik soweit korrekt angegeben wurden, so dass eine Grammatik erzeugt werden kann. Nur wenn dies der Fall ist, wird ein neues Grammatik-Objekt erzeugt und der Editor beendet. Wird ein Fehler erkannt, wird eine entsprechende Meldung angezeigt und der Editor noch nicht beendet. Zudem wird dabei erfasst, ob die Grammatik nur Produktionen ohne Kontext enthält (jedes Produktion-Objekt hat eine entsprechende boolesche Variable). Ist dies der Fall, wird bei der Grammatik die entsprechende boolesche Variable über den Konstruktor auf wahr gesetzt und die Grammatik entsprechend als kontextfreie Grammatik gekennzeichnet. Für den Anwender ist dies jedoch nicht sichtbar, da es nur programminternen Zwecken dient.

Über den Button „Abbrechen“ kann der Editor jederzeit ohne die Erzeugung einer Grammatik beendet werden. Das Programm kehrt dann im Prinzip in den Status vor dem Aufruf des Editors zurück.

Wie zu Beginn bereits erwähnt, wird der Editor auch zum Editieren bereits bestehender Grammatiken genutzt. Hierzu gibt es einen zweiten Konstruktor in der Klasse des Editors, über den eine Grammatik mit übergeben werden kann. Mit den Daten der Grammatik werden dann im Konstruktor die entsprechenden Listen belegt.

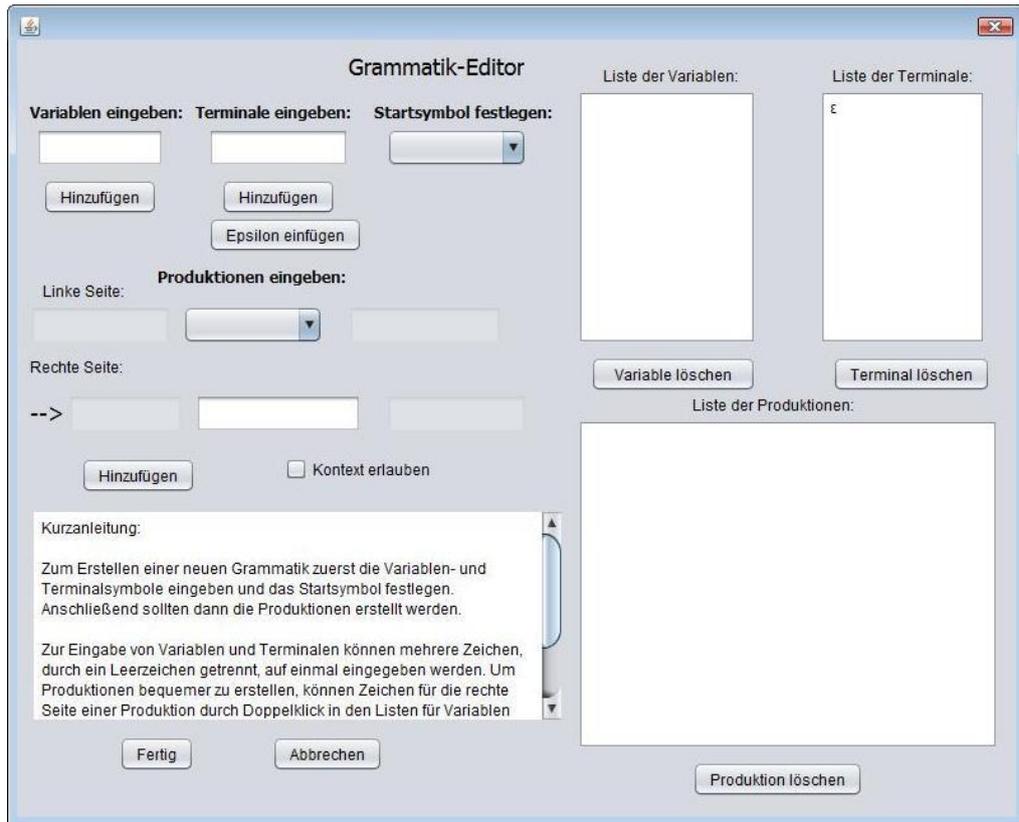


Abbildung 6: Benutzeroberfläche des Grammatik-Editor in Startkonfiguration

4.4.3 Visualisierung der Ableitungsbäume mit JGraphX

Die Darstellung der Ableitungsbäume erfolgte im Prototypenstadium des Programms zunächst über eine JTree Komponente. Da Ableitungsbäume jedoch Bäume im Sinne der Graphentheorie darstellen, war diese Anzeige in der Form von Verzeichnisbäumen nicht zufriedenstellend. Java bietet keine direkte Unterstützung für die Darstellung von Graphen im Sinne der Graphentheorie, so dass nach einer Drittanbieterlösung gesucht wurde, die die Anforderungen an die Darstellung erfüllt.

Gefunden wurde diese mit der Open Source Bibliothek JGraphX [JGX], die von der englischen Firma JGraph Ltd. unter der BSD-Lizenz zur Verfügung gestellt wird. Die BSD-Lizenz erlaubt die Verwendung der Bibliothek unter der Bedingung, dass die Erklärung der Urheberrechte der JGraph Ltd. an der Bibliothek in der Dokumentation bzw. dem Quellcode enthalten ist.

Die Bibliothek stellt umfassende Möglichkeiten zur Visualisierung und automatisierten Anordnung (sog. Layoutmanager) von mathematischen Graphen zur Verfügung. Somit ist im Speziellen auch die Darstellung eines Baumes im Sinne der Graphentheorie mit JGraphX möglich.

In der Simulationsumgebung wurde JGraphX für die Visualisierung der Ableitungsbäume verwendet, obwohl die Bibliothek auch die entsprechenden Datenstrukturen bzw. Klassen implementiert, um auch das Datenmodell über JGraphX zu realisieren. Da aber nicht sicher ist, wie sicher der Support für die Bibliothek in der Zukunft sein wird, muss einkalkuliert werden, dass eventuell Inkompatibilitäten bei zukünftigen Java-Versionen auftreten könnten. Deshalb wurde der Einsatz der Bibliothek JGraphX so begrenzt, dass bei Bedarf auf eine alternative Bibliothek für die Visualisierung von Graphen gewechselt werden kann, ohne die komplette Architektur der Simulationsumgebung neu aufbauen zu müssen.

Wie bereits ausgeführt wurde, wird deshalb das Datenmodell der Ableitungsbäume in Objekten der Klasse DefaultTreeModel gehalten, die zum Standardumfang der Java-Bibliotheken gehört. Alle Änderungen im Datenmodell werden zunächst auf diesen Objekten ausgeführt, dann wird aus ihnen ein Objekt der Klasse mxGraph, die die Klasse zur Modellierung von Graphstrukturen in JGraphX darstellt, erzeugt. Auf diesen mxGraph wird dann ein Layoutmanager zur Erzeugung eines Baum-Layouts angewendet, also eine automatische Posi-

tionierung der Knoten für die Anzeige durchgeführt. Da JGraphX das in Java übliche MVC-Prinzip umsetzt, muss dann aus dem mxGraph-Objekt (in dem Fall das Model) ein Objekt der Klasse mxGraphComponent (der View für den Graphen) erzeugt werden. Abschließend wird noch der Viewport einer JScrollPane auf das mxGraphComponent-Objekt gelegt, um den Graphen im Programm sichtbar zu machen.

Für die Interaktion mit dem Graphen wird in der Bibliothek auch eine Reihe von entsprechenden Listnern implementiert. Dadurch können Knoten im Graphen bequem per Mausklick selektiert werden und die aktuelle Selektion über das mxGraph-Objekt abgefragt werden. Diese Funktion wird in der Simulationsumgebung für die Durchführung einer manuellen Ableitung benötigt.

4.4.4 Die verwendeten Algorithmen

Wie einleitend angekündigt, folgt hier eine verkürzte Darstellung der algorithmischen Umsetzungen für die verschiedenen Möglichkeiten zur Erzeugung von Ableitungsbäumen. Hierzu stelle ich vorab einige Hilfsmethoden vor, die in den Algorithmen wiederholt verwendet werden. Dabei wird in Wesentlichen nur die Funktion bzw. der Zweck der Methode erläutert, ohne auf sämtliche Details einzugehen.

Danach folgen die Erläuterungen der Umsetzungen für das Ausführen von manuellen Ableitungen, Prüfen von Worten und Erzeugen von Worten. Die Darstellungen sind hier, wie bereits gesagt, verkürzt, um den Rahmen dieser Arbeit nicht zu sprengen.

Für genauere Details zu diesen Umsetzungen möchte ich auf die entsprechenden Java-Quellcode Dateien verweisen.

4.4.4.1 Überblick über viel verwendete Hilfsmethoden

Die hier dargestellten Methoden werden für jede Weise zur Erstellung einer Ableitung benötigt und finden sich deshalb, teilweise in leicht angepasster Form, in allen Algorithmen wieder.

1. *getLeafs(...)*

In den meisten Fällen werden die Blätter eines Ableitungsbaums für die weiteren Schritte benötigt. Unter diesen Knoten finden sich zum einen die Terminale, die am Ende das Wort bilden, und zum anderen die Variablen, die noch ersetzt werden können. Um mit den Blättern korrekt arbeiten zu können, müssen diese in der korrekten Reihenfolge vorliegen. Dies wird gewährleistet, in dem aus dem Baummodell eine Post-Order- bzw. Depth-First-Enumeration (dies liefert in Java das gleiche Ergebnis) der Knoten erzeugt wird. Diese Enumeration wird nun noch durchlaufen und alle Blätter (dies kann über eine Methode der Knotenklasse bestimmt werden) in eine verkettete Liste übernommen. Dabei werden auch jene Blätter aussortiert, die als Kontext markiert wurden (siehe Abschnitt 4.3). Diese Liste wird dann als Ergebnis zurückgegeben.

Für manuelle Ableitungen hat diese Methode keine Parameter, da es nur ein Baummodell gibt. Für das Prüfen bzw. Erzeugen von Worten wird die Referenz auf das jeweilige Baummodell als Parameter übergeben.

2. *bestimmeWort(...)*

Diese Methode bestimmt den String, der durch die Zeichen der Blätter eines Baummodells gebildet wird. Hierzu wird mit *getLeafs(...)* zunächst die Liste der Blätter abgerufen. Dann wird in einer Schleife aus den einzelnen Knoten der String aufgebaut, wobei das Zeichen „ε“ ausgelassen wird.

3. *get_VarKnoten(...)*

Diese Methode erzeugt eine Liste der Variablen, die noch ersetzt werden dürfen. Hierzu wird wieder über *getLeafs(...)* die Liste der Blätter erzeugt und aus dieser anschließend die Knoten, die eine entsprechende Variable repräsentieren, in eine neue Liste übernommen. Die neue Liste wird dann als Ergebnis zurückgegeben.

Analog zu *getLeafs(...)* wird auch diese Methode in zwei verschiedenen Varianten bezüglich der Parameter verwendet.

4. *get_TermKnoten(...)*

Diese Methode ist analog zu *get_VarKnoten(...)*, wobei hier, wie der Name bereits impliziert, eine Liste der Knoten, die ein Terminal repräsentieren, als Ergebnis zurückgegeben wird. Das Zeichen „ε“ wird auch hier wieder ausgelassen.

Ebenso wird diese Methode in zwei Varianten bezüglich der Parameter verwendet.

5. *bestimme_Produktionen(...)*

Die Methode erzeugt eine Liste der Produktionen, die auf eine bestimmte Variable in einem Ableitungsbaum angewendet werden können. Auch diese Methode wird bezüglich der Parameter wieder in zwei Varianten verwendet. In beiden Fällen wird die Referenz auf den Kno-

ten, der die Variable repräsentiert, übergeben und in den Fällen, in denen mehrere Baummodelle existieren, wird auch die Referenz auf das entsprechende Baummodell übergeben.

Im Ablauf der Methode wird eine Fallunterscheidung gemacht:

Für kontextfreie Grammatiken genügt es, das Zeichen der Variablen (bzw. des Knotens) mit der linken Seite aller Produktionen zu vergleichen und bei Übereinstimmung die Produktion in die Ergebnisliste zu übernehmen.

Für kontextsensitive Grammatiken sind mehr Schritte nötig bzw. es muss weiter unterschieden werden. Enthält eine einzelne Produktion keinen Kontext, genügt wieder der Vergleich mit dem Zeichen. Gibt es jedoch einen Kontext, so muss noch geprüft werden, ob dieser erfüllt ist. Dazu wird mit *bestimmeWort(...)* die aktuelle Zeichenfolge erzeugt. Diese wird dann in die Teilfolgen links der Variable und rechts der Variable zerlegt und dann mit dem jeweiligen Kontext, sofern vorhanden, verglichen. Stimmen sowohl Zeichen als auch Kontext überein, wird die Produktion in die Liste übernommen.

6. *ableiten(...)*

Diese Methode führt einen einzelnen Ableitungsschritt auf ein Baummodell aus, also eine Überführung $w \Rightarrow w'$. Hierzu wurde zuvor das entsprechende Variablenzeichen V und die passende Produktion p festgelegt, und diese Information wird als entsprechende Parameter an die Methode übergeben. Auch hier gibt es wieder zwei Varianten, die sich darin unterscheiden, wie der Knoten, der das Variablenzeichen V repräsentiert, identifiziert wird.

Im Ableitungsbaum wird eine Ersetzung bekannterweise dadurch repräsentiert, dass die neuen Zeichen als Knoten an den Knoten des zu ersetzenden Zeichens angehängt werden. Für kontextfreie Grammatiken bzw. bei Produktionen ohne Kontext werden also nur die neuen Knoten, die durch die Produktion vorgegeben werden, erzeugt und an den Knoten von V als Kinder angefügt. Zudem wird V als nicht mehr als ableitbar markiert (siehe Datenstruktur Ableitungsknoten).

Für kontextsensitive Grammatiken bzw. Produktionen mit Kontext wurde das Vorgehen bereits in Abschnitt 4.3 skizziert: An den Knoten von V werden zunächst Kopien des linken Kontextes, sofern vorhanden, angefügt und die Originalknoten als Kontext markiert. Dann folgen wie bei kontextfreien Produktionen die neuen Zeichen. Anschließend werden analog zum linken Kontext Kopien des rechten Kontextes, sofern vorhanden, angefügt und die Originale markiert. Ebenso wird natürlich der Knoten von V als nicht mehr ableitbar markiert.

4.4.4.2 Manuelle Ableitungen

Für eine bessere Übersicht im Quellcode des Programms wurde der Code für die Ausführung einer manuellen Ableitung in eine eigene Klasse `Manuelle_Ableitung.java` zusammengefasst. Wird im Programm eine neue manuelle Ableitung begonnen, wird ein neues Objekt der Klasse erzeugt. Die Klasse stellt zu einen die notwendigen Methoden zur Verfügung und verwaltet zum anderen auch das `TreeModell` Objekt, das durch den Konstruktor der Klasse erzeugt wird.

Die Klasse `Manuelle_Ableitung` enthält dabei keinen größeren Algorithmus im eigentlichen Sinne, da die Ableitung schrittweise vom Benutzer gesteuert wird. Sie stellt daher nur sämtliche Hilfsmethoden gebündelt bereit, die je nach Eingabe des Benutzers aufgerufen werden.

Ein einzelner Schritt in der Ableitung kann vom Nutzer in drei Varianten ausgeführt werden: Linksableitung, Rechtsableitung oder ausgewählter Knoten.

Bei Links- und Rechtsableitung wird über `get_VarKnoten()` die Liste der Variablen bestimmt und entsprechend das erste bzw. letzte Element genommen.

Anschließend wird über `bestimme_Produktionen()` die Liste der passenden Produktionen bestimmt. Über einen Dialog wählt der Benutzer eine Produktion aus. Die Referenz auf den Knoten und die Produktion werden an `ableiten(Knoten, Produktion)` (der Knoten wird hier also direkt über die Referenz identifiziert) übergeben und der entsprechende Schritt ausgeführt.

In der dritten Variante kann der Benutzer die zu ersetzende Variable auch durch die Auswahl des Knotens mit der Maus in der Visualisierung des Baumes festlegen. Hierzu wird entsprechend vom Graphen das selektierte Element

abgefragt, analog zur Links- und Rechtsableitung die Produktion bestimmt und anschließend über *ableiten(Knoten, Produktion)* der Schritt ausgeführt.

Um falsche Schritt korrigieren oder alternative Möglichkeiten probieren zu können, ohne von neuem beginnen zu müssen, wird auch die Funktionalität „Rückgängig machen“ bereitgestellt. In der Klasse wird hierzu eine Liste geführt, in der alle einzelnen Schritte gemerkt werden. Ein einzelner Schritt ist hierbei wieder durch eine Liste repräsentiert, die alle Knoten enthält, die in einem Schritt geändert (z.B. Kinder angefügt oder als Kontext markiert) wurden. Um einen Schritt rückgängig zu machen, wird entsprechend die Liste des letzten Schrittes, die als letzte Liste in der Liste aller Schritt steht, genommen und durchlaufen. Bei jedem Knoten werden die Attribute wieder so gesetzt, dass der Knoten kein Kontext ist und, sofern der Knoten eine Variable repräsentiert, wieder ableitbar ist. Einer der Knoten muss zudem Kinder haben, welche wieder entfernt werden.

4.4.4.3 Prüfung von Worten

Wie für die manuelle Ableitung wurde der Code für die Prüfung von Worten in eine eigene Klasse *Wort_testen.java* zusammengefasst.

Der durch *Wort_testen* bereitgestellte Algorithmus leistet dabei im Endeffekt zwei Dinge: Zum einen wird festgestellt, ob ein Wort durch die Grammatik erzeugt werden kann oder nicht. Zum zweiten werden auch, falls das Wort erzeugt wird, alle Ableitungsbäume, die das Wort erzeugen, gebildet und zur Anzeige bereitgestellt.

Der Algorithmus orientiert sich dabei an dem in Abschnitt 3.5.2 dargestellten Ansatz aus [Sch09], in dem solange alle möglichen Ableitungsbäume mit zunehmender Größe gebildet werden, bis diese ein Abbruchkriterium erfüllen. Die Verwendung dieses Algorithmus begründet sich vor allem darin, dass auch kontextsensitive Grammatiken damit zu behandeln sind. Da der Algorithmus alle möglichen Ableitungsbäume für ein vorgegebenes Wort finden soll, kann die in [Sch09] angegebene Bedingung, mit dem ersten Auffinden einer Lösung abubrechen, nicht verwendet werden. In Abweichung wurde deshalb als Abbruchkriterium für den Algorithmus die Anzahl der Terminale bzw. Variablen im Vergleich mit der Länge des zu prüfenden Wortes gewählt. Sobald ein Baum mehr Terminale enthält als die Länge des Wortes, wird dieser nicht wei-

ter in neue Bäume überführt. Da Terminale nur bei Typ-0 Grammatiken wieder verschwinden können, kann von diesen Bäumen das vorgegebene Wort nicht erzeugt werden bzw. können sie nicht zu diesem führen. Die Vorgabe, auch die Anzahl der Variablen in gleichem Maße zu beschränken, ist dagegen mehr eine Vernunftentscheidung: Für Grammatiken ohne Produktionen der Form $V \rightarrow \varepsilon$ mit $V \in V_N$ gilt die gleiche Argumentation wie bei den Terminalen, da das vorgegebene Wort nicht mehr durch den Baum erzeugt werden kann. Formal sind Produktionen dieses Typs für Typ-1 Grammatiken nicht zugelassen. Im Programm wird die Einhaltung dieser Bedingung jedoch nicht algorithmisch überprüft, so dass hier der Benutzer für die Erstellung einer formal korrekten Grammatik verantwortlich ist.

Sind solche Produktionen enthalten, ist es theoretisch möglich, das vorgegebene Wort noch zu erreichen, aber nicht sehr wahrscheinlich, sofern es sich nicht um eine sehr ungewöhnliche Grammatik handelt. Da im schulischen Kontext nicht mit solchen außergewöhnlichen Fällen zu rechnen ist, wurde dieses Abbruchkriterium gewählt, das sich bislang als sinnvoll erwiesen hat.

Über den Konstruktor der Klasse werden für den Algorithmus zunächst das zu prüfende Wort und die aktuelle Grammatik gesetzt. Über den Aufruf der Methode *ablbaeume_bestimmen()* wird dann der eigentliche Algorithmus gestartet, der im Folgenden in Pseudocode dargestellt ist:

```
Wortlänge := Länge des zu prüfenden Wortes;  
Baum := neues Baummodell mit Startsymbol als einzigen Knoten;  
Ergebnisse := neue leere Liste; //Liste für die Baummodelle, die das Wort  
erzeugen;  
Bäume_alt := neue leere Liste;  
Bäume_neu := neue leere Liste;  
Bäume_alt.hinzufügen(Baum);
```

Nach der Initialisierung dieser Variablen erfolgt eine Fallunterscheidung:

Für kontextfreie Grammatiken ist es ausreichend, in jedem Schritt alle verschiedenen Linksableitungen zu bilden. Bereits in Abschnitt 3.4.2 dieser Arbeit wurde angemerkt, dass zu jedem Ableitungsbaum eine Linksableitung existiert. Für eine eventuelle Mehrdeutigkeit sind zudem nur verschiedene Ableitungsbäume, die jeweils eine Linksableitung besitzen, interessant, so dass es ausrei-

chend ist, nur die Linksableitungen zu verfolgen. Alternativ könnten natürlich auch nur die Rechtsableitungen gebildet werden.

Für Kontextsensitive Grammatiken ist dies nicht möglich. Es ist denkbar, dass eine Produktion in einem gewissen Schritt anwendbar ist und so zum gesuchten Wort führt, dieser Schritt aber weder eine Links- noch eine Rechtsableitung darstellt. Deshalb müssen für kontextsensitive Grammatiken alle Variablen, die ersetzt werden dürfen, durchlaufen werden und alle jeweils möglichen Ersetzungen ausgeführt werden.

Im Abschnitt über die Datenstruktur für Grammatiken wurde bereits erwähnt, dass zu einer Grammatik bei ihrer Erstellung geprüft wird, ob diese kontextfrei oder kontextsensitive ist und diese Information entsprechend in einem entsprechenden Attribut der Grammatik hinterlegt wird. Entsprechend prüft der Algorithmus für diese Fallentscheidung nur das entsprechende Attribut der aktuellen Grammatik.

Fall 1: Grammatik ist kontextfrei

```
Solange Bäume_alt nicht leer ist wiederhole:
  Für jedes Element von Bäume_alt als Baum_akt:
    Wenn (Baum_akt enthält weniger Terminale als Wortlänge
    und enthält weniger Variablen als Wortlänge):
      Variablen := get_VarKnoten(Baum_akt);
      Var := Erstes Element von Variablen
      Produktionen = bestimme_Produktionen(Var);
      Für jedes Element von Produktionen als Prod:
        Baum_kopie := kopiererzeugen(Baum_akt);
        Index := Bestimme den Index von Var in
        Baum_akt um die entsprechende Kopie in
        Baum_kopie zu finden;
        ableiten(baum_kopie, index, prod);
        Bäume_neu.hinzufügen(Baum_kopie);
      Ende Schleife
    Ende Klausel
  akt_wort := bestimmeWort(Baum_akt);
  Wenn (akt_wort gleich zu prüfendes Wort):
    Ergebnis.hinzufügen(Baum_akt)
  Ende Klausel
Ende Schleife
Bäume_alt := Bäume_neu;
Bäume_neu := neue leere Liste;
Ende Schleife
```

Zum Erzeugen eines neuen Baumes muss eine Kopie des Baumes erstellt werden, der den Ausgangspunkt darstellt, da sonst die Änderungen immer auf dem gleichen Objekt ausgeführt würden. Die in Java bereitgestellten Methoden erzeugen jedoch nur flache Kopien, also nur eine Kopie der Referenz, und erfüllen somit nicht den geforderten Zweck. Deshalb wurde eine eigene Methode *kopierzeugen(Baum)* erstellt, die eine echte Kopie erzeugt. Auf eine detailliertere Ausführung zum Aufbau der Methode wird an dieser Stelle verzichtet, da diese keinen relevanten Beitrag zu dieser Arbeit darstellt. Bei Bedarf möchte ich nochmals auf den entsprechenden Quellcode des Programms verweisen.

Fall 2: Grammatik ist kontextsensitiv

```

Solange Bäume_alt nicht leer ist wiederhole:
  Für jedes Element von Bäume_alt als Baum_akt:
    Wenn (Baum_akt enthält weniger Terminale als Wortlänge
    und enthält weniger Variablen als Wortlänge):
      Variablen := get_VarKnoten(Baum_akt);
      Für jedes Element von Variablen als Var:
        Produktionen = bestimme_Produktionen(Var);
        Für jedes Element von Produktionen als Prod:
          Baum_kopie :=
            kopierzeugen(Baum_akt);
          Index := Bestimme den Index von Var
            in Baum_akt um die entsprechende Ko-
            pie in Baum_kopie zu finden;
          ableiten(Baum_kopie, Index, prod);
          Bäume_neu.hinzufügen(Baum_kopie);
        Ende Schleife
      Ende Schleife
    Ende Klausel
  akt_wort := bestimmeWort(Baum_akt);
  Wenn (akt_wort gleich zu prüfendes Wort):
    Ergebnis.hinzufügen(Baum_akt)
  Ende Klausel
Ende Schleife
Bäume_alt := Bäume_neu;
Bäume_neu := neue leere Liste;
Ende Schleife

```

Nach der Fallunterscheidung wird die Liste Ergebnisse als Ergebnis zurückgegeben. Ist die Liste leer, gehört das Wort nicht zur Sprache der Grammatik.

4.4.4.4 Erzeugung von Worten

Der Algorithmus zur Erzeugung von Worten ist fast identisch mit dem Algorithmus für das Prüfen von Worten. Wie dargestellt wurde, nutzt der Algorithmus zur Prüfung von Worten die Wortlänge als Abbruchkriterium und bildet alle möglichen, unterschiedlichen Bäume. Dabei werden automatisch auch alle Bäume gebildet, die Worte erzeugen, die kürzer als die Vorgabe sind.

Für die Erzeugung von Worten wurde der Algorithmus daher lediglich wie folgt abgewandelt:

- Als Vorgabe wird dem Algorithmus eine maximale Wortlänge gegeben statt einem Wort, über das sich die Länge bestimmt.
- In die Liste Ergebnisse werden alle Baummodelle übernommen, deren Blätter nur Terminale repräsentieren.

4.4.5 Benutzeroberfläche des Hauptfensters

Der Aufbau der Benutzeroberfläche im Hauptfenster der Simulationsumgebung kann im Wesentlichen in fünf Bereiche unterteilt werden, wie in Abbildung 7 zu sehen ist.

- 1) Dieser größte Bereich des Fensters wird für die Anzeige der erstellten Ableitungsbäume und der Liste erzeugter Worte genutzt. Die Anzeige liegt dabei auf eine JScrollPane, so dass bei größeren Anzeigen automatisch Scrollbalken erscheinen.
- 2) Über die Menüleiste am oberen Rand des Fensters erfolgt die grundlegende Bedienung der Simulationsumgebung. Über das Menü „Datei“ kann zum einen das Programm beendet werden und zum anderen können über das Untermenü „Grammatik“ die Funktionen „Neu“ (Erstellen einer neuen Grammatik), „Laden“, „Bearbeiten“, „Speichern“ und „Speichern unter“ aufgerufen werden.

Über das Menü „Modus“ können die drei verschiedenen Modi zur Erstellung von Ableitungsbäumen gestartet werden.

Einzelne Menüeinträge werden im Programm erst dann aktiviert, wenn eine Grammatik erstellt oder geladen wurde. Um Bedienfehler zu vermeiden, können auf diesem Wege z.B. die Modi zur Erstellung von Ableitungsbäumen erst dann ausgewählt werden, wenn auch eine Grammatik vorhanden ist.

- 3) In diesem Bereich am rechten Rand des Fensters liegen die Buttons, über die die verschiedenen Funktionen innerhalb der Varianten zur Erzeugung von Ableitungsbäumen aufgerufen werden. Beispielsweise wird über den Button „Linksableitung“ ein Linksableitungsschritt ausgeführt. Dies wird in Abschnitt 5.5 anhand eines Beispiels genauer erläutert.

Die Buttons werden dabei jeweils nur im entsprechenden Modus aktiviert bzw. wenn ihre Funktion im Programm genutzt werden kann.

- 4) Im Bereich unter den Buttons zur Bedienung innerhalb der Modi werden für den Benutzer interessante Informationen angezeigt. Bei der Erstellung einer Ableitung wird hier die aktuell gebildete Zeichenkette bzw. das abgeleitete Wort und darunter wird die aktuell aktive Grammatik in der Mengenschreibweise angezeigt. Im Modus „Worte erzeugen“

4.5 Anwendungsbeispiele

In diesem Abschnitt möchte ich die Benutzung und Funktion der Simulationsumgebung an einem konkreten Beispiel vorstellen. Im Einzelnen möchte ich noch einmal zusammengefasst darstellen, wie eine neue Grammatik erstellt wird und dann die Möglichkeiten zur Erzeugung von Ableitungsbäumen mit dieser Grammatik erläutern.

Die Grammatik für dieses Beispiel ist die Grammatik $G = (V_N, V_T, P, S)$ mit

$V_N = \{Start, Klammerung\}$

$V_T = \{\varepsilon, (,)\}$

$P = \{Start \rightarrow \varepsilon|(Klammerung)|Klammerung()|()Klammerung, Klammerung \rightarrow \varepsilon|(Klammerung)|Klammerung()|()Klammerung\}$

und $S = Start$.

Die von der Grammatik erzeugte Sprache umfasst die wohlgeformten Klammersausdrücke. Diese Grammatik ist als Beispieldatei unter dem Namen Klammersausdrücke.gds im Beispiel-Verzeichnis der Simulationsumgebung beigelegt.

4.5.1 Erstellen einer Grammatik

Über die Menüleiste wird über Datei \rightarrow Grammatik \rightarrow neu der Editor zum Erstellen einer neuen Grammatik geöffnet. Der Editor-Dialog erscheint in der bereits aus Abbildung 6 bekannten Startkonfiguration.

Der erste Schritt sollte nun die Eingabe der Variablenzeichen sein, alternativ könnte jedoch auch mit den Terminalzeichen begonnen werden. Die zweite Zeichenmenge sollte dann entsprechend anschließend eingegeben werden. Die Zeichen werden in das jeweilige Textfeld eingegeben und über den jeweiligen „Hinzufügen“-Button zur entsprechenden Liste hinzugefügt. Dabei können auch mehrere Zeichen auf einmal hinzugefügt werden, in dem diese durch ein Leerzeichen getrennt in das Textfeld eingegeben werden. Das Programm prüft dabei, dass es nicht zu Überschneidungen zwischen den Mengen der Variablen und Terminalen kommt und gibt gegebenenfalls eine entsprechende Fehlermeldung aus.

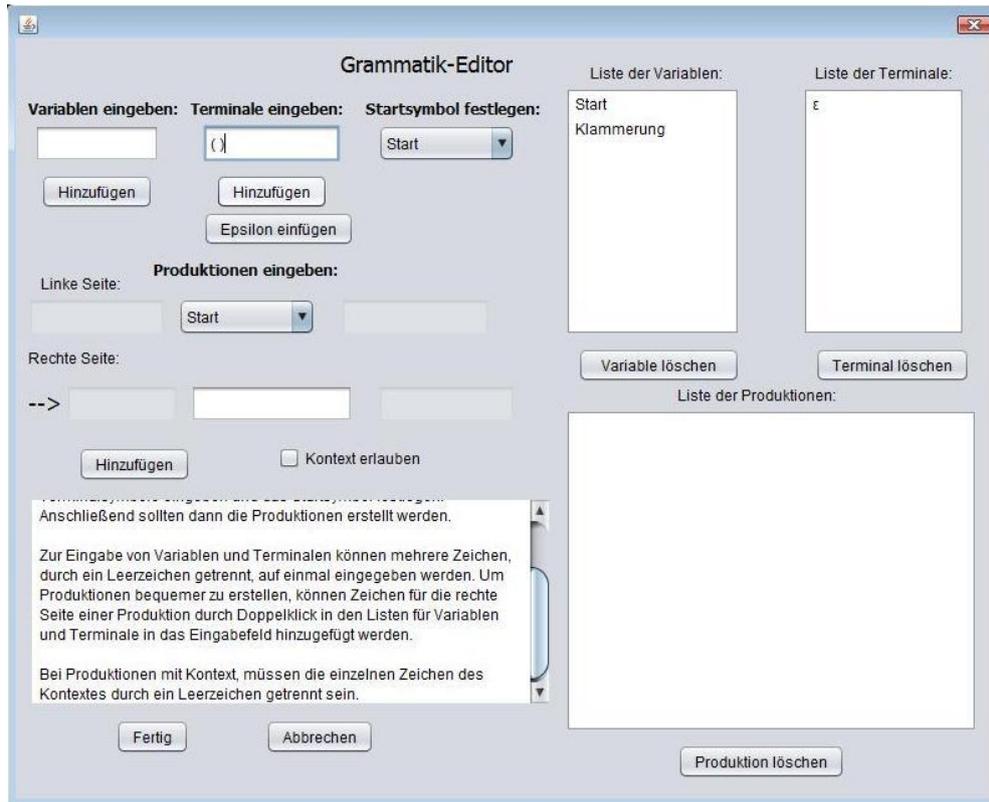


Abbildung 8: Eingabe der Variablen und Terminale

Sobald die Variablenzeichen eingegeben wurden, kann das Startsymbol festgelegt werden. Nach der Eingabe der ersten Variablenzeichen wird vom Programm automatisch das erste Zeichen aus der Liste als Startsymbol ausgewählt. Über die ComboBox kann diese Auswahl vom Benutzer angepasst werden. Werden Zeichen aus der Liste der Terminale entfernt, ist dieser Schritt gegebenenfalls zu wiederholen.

Nachdem nun das Alphabet der Grammatik eingegeben wurde, können nun die Produktionen erstellt werden. Hierzu wird die linke Seite der Produktion über die entsprechende ComboBox ausgewählt. Die rechte Seite kann entweder von Hand eingegeben werden oder es können einzelne Zeichen durch Doppelklick auf das gewünschte Zeichen in der Liste der Variablen bzw. Terminale in das Textfeld hinzugefügt werden. Bei der Eingabe der rechten Seite ist zu beachten, dass einzelne Zeichen durch ein Leerzeichen voneinander getrennt sind. Eine fertig eingegebene Produktion wird entsprechend wieder über den „Hinzufügen“-Button in die Liste der Produktionen übernommen. Dabei erfolgt wieder eine Prüfung der Gültigkeit der enthaltenen Zeichen, und es wird ge-

benenfalls eine entsprechende Fehlermeldung angezeigt. Weitere Produktionen können entsprechend nacheinander eingegeben werden.

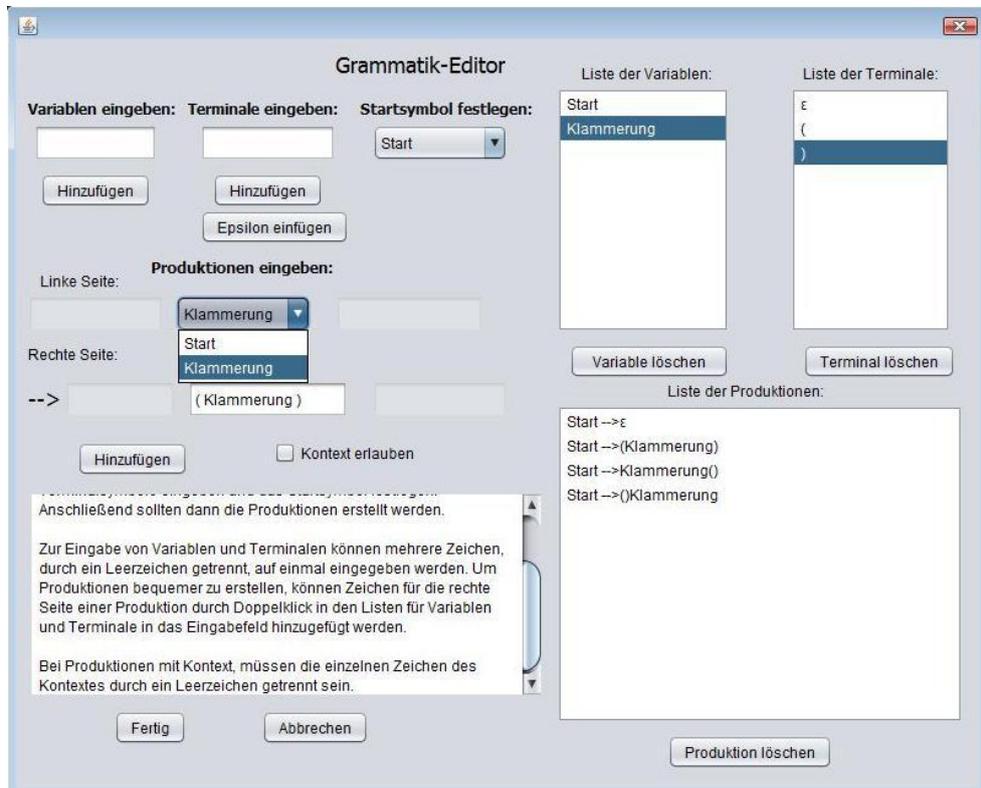


Abbildung 9: Erstellen der Produktionen

Wurde die Grammatik vollständig eingegeben, kann der Editor über den „Fertig“-Button beendet werden. Dabei erfolgt noch eine weitere Prüfung der Eingabe, die zu entsprechenden Fehlermeldungen führen kann. Werden keine weiteren Fehler gemeldet, wird die neue Grammatik endgültig erzeugt und der Editor-Dialog schließt sich.

Im Hauptfenster wird nun im Anzeigebereich rechts die soeben erstellte Grammatik in der Mengenschreibweise angezeigt. Der Name der Grammatik wird dabei noch mit „Unbenannt“ angegeben. Optional kann die Grammatik nun über Datei → Grammatik → speichern (oder speichern unter) in einer Datei gespeichert werden. Dabei erhält die Grammatik dann den Dateinamen als Namen.

Zudem sind nun auch die weiteren Menüeinträge aktiviert, so dass die Grammatik nun auch bearbeitet werden und einer der Modi zur Erstellung einer Ableitung gestartet werden kann.

4.5.2 Durchführung einer manuellen Ableitung

Mit der, wie zuvor beschrieben, erstellten Grammatik kann eine manuelle Ableitung ausgeführt werden. Das Vorgehen hierzu wird im Folgenden für das Beispielwort „ $()(())()$ “ dargestellt.

Eine neue manuelle Ableitung wird über Modus \rightarrow Manuelle Ableitung gestartet. In der Anzeige erscheint ein neuer Ableitungsbaum, der zunächst nur aus der Wurzel, die das Startsymbol repräsentiert, besteht. Weiterhin sind nun die Buttons „Linksableitung“, „Rechtsableitung“, „Auswahl ableiten“ und „Rückgängig“ im rechten oberen Bereich des Hauptfensters aktiv. Die Steuerung der manuellen Ableitung erfolgt über diese Buttons.

Um nun einen Ableitungsschritt auszuführen, kann einer der drei Buttons „Linksableitung“, „Rechtsableitung“ oder „Auswahl ableiten“ genutzt werden. Werden nun Links- oder Rechtsableitung genutzt, sucht das Programm selbstständig die entsprechende Variable bzw. den Knoten (für Details verweise ich hier auf den Abschnitt 5.4.4.2 über die algorithmische Umsetzung). Es erscheint der Dialog zur Auswahl der gewünschten Produktion, und nach der Auswahl wird im Hauptfenster der aktualisierte Ableitungsbaum mit den neu eingefügten Knoten angezeigt.

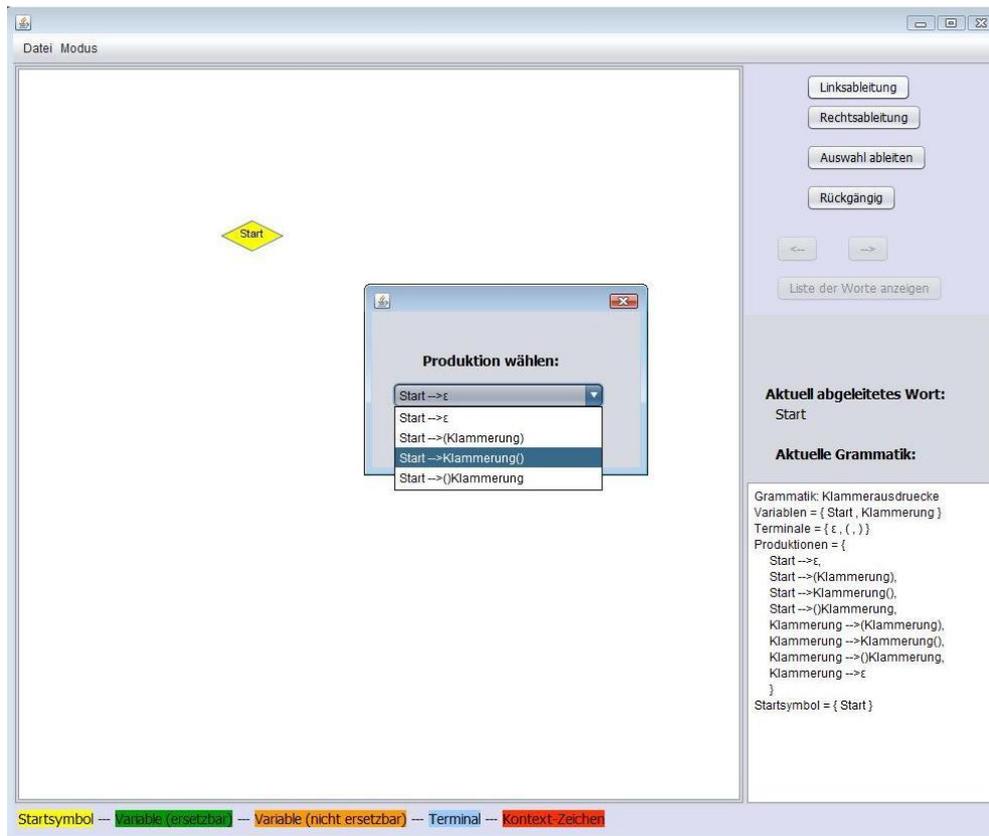


Abbildung 10: Beginnen einer manuellen Ableitung

Der nächste Schritt kann nun wieder über einen der genannten Buttons ausgelöst werden. Beispielsweise kann er nun über den Button „Auswahl ableiten“ ausgeführt werden. Hierzu muss die zu ersetzende Variable mit einem Mausklick auf den entsprechenden Knoten im Ableitungsbaum gewählt werden. Die getroffene Auswahl wird durch einen blauen Rahmen markiert. Ersetzbare Variable werden im Ableitungsbaum als grüne Sechsecke dargestellt. Wurde so eine gültige Variable für den nächsten Schritt ausgewählt, erscheint nach Klick auf den Button „Auswahl ableiten“ wieder der Dialog zur Wahl der Produktion, analog zur Links- bzw. Rechtsableitung.

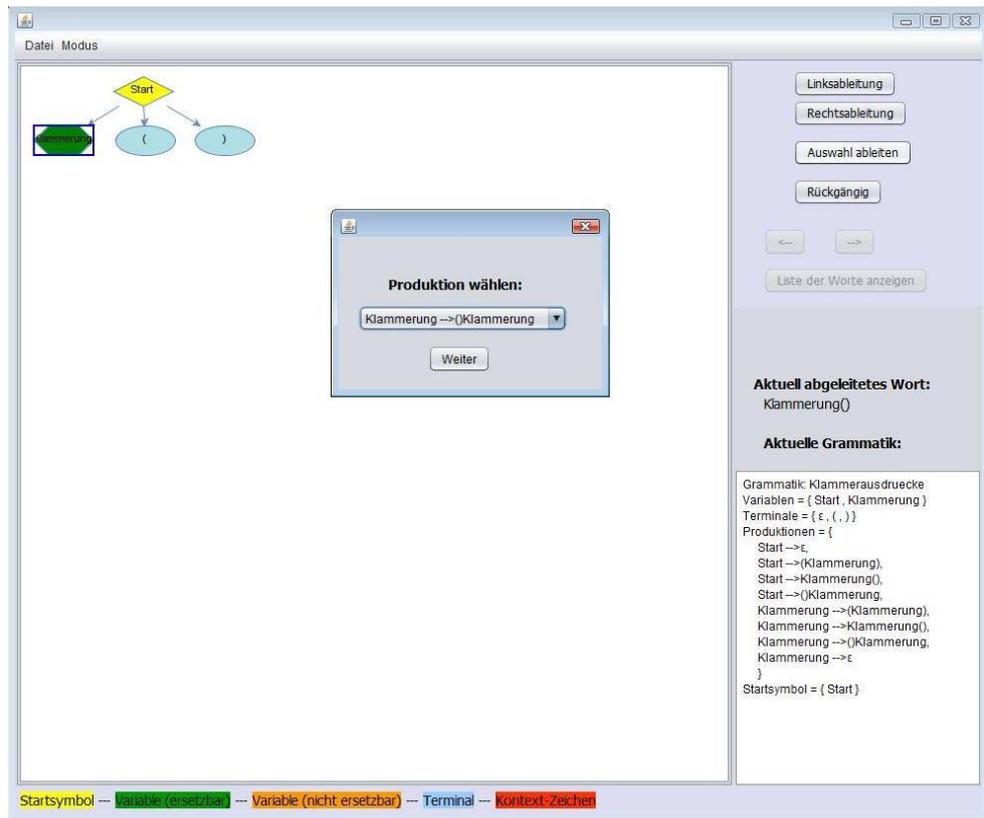


Abbildung 11: Ersetzung einer ausgewählten Variable

Anschließend wird entsprechend wieder die Anzeige des Ableitungsbaumes aktualisiert. Dabei wechselt die Farbe der zuvor gewählten Variable von Grün auf Orange, um so kenntlich zu machen, dass diese nicht mehr für eine Ersetzung gewählt werden kann.

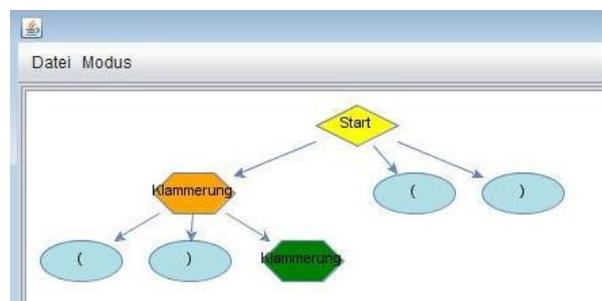


Abbildung 12: Farbcodierung von Variablen

Auf diese Weise können nun die weiteren Schritte ausgeführt werden, bis das gesuchte Wort gebildet wurde.

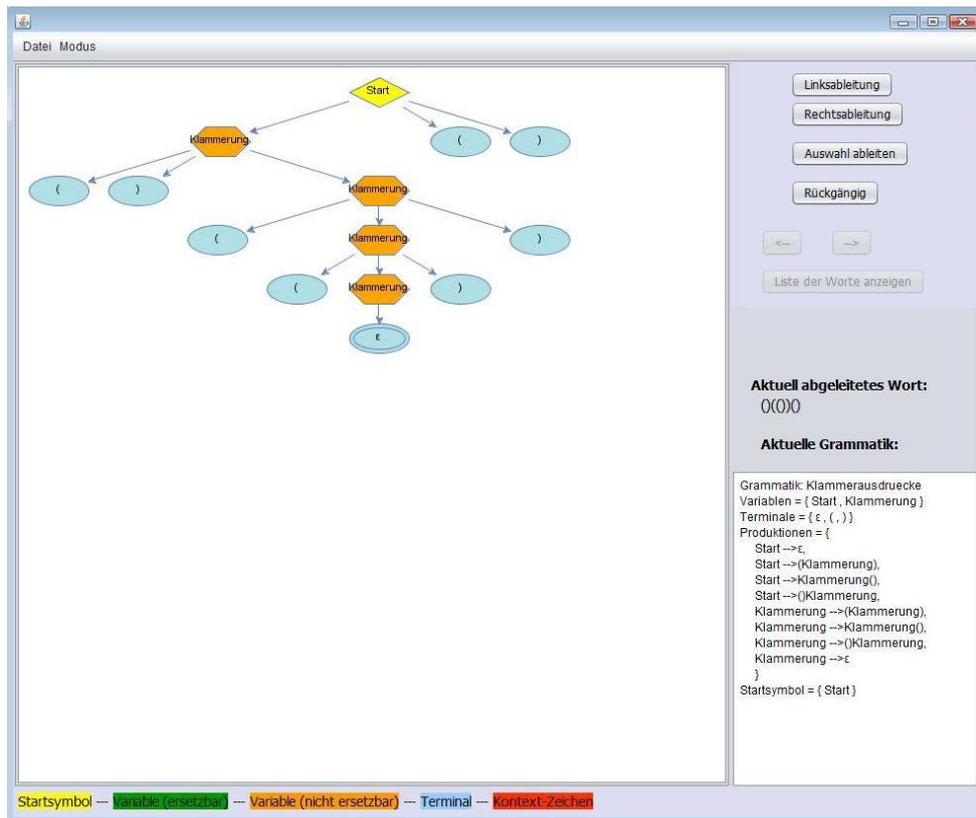


Abbildung 13: Endzustand einer manuellen Ableitung

Das aktuelle gebildete Wort bzw. die aktuelle Zeichenkette, die durch die Terminale und Variablen der Blätter des Ableitungsbaumes gebildet wird, wird auch in jedem Schritt rechts, im Anzeigenbereich des Hauptfensters, angezeigt. Die einzelnen, ausgeführten Schritte können über den Button „Rückgängig“ bis zum Start zurück rückgängig gemacht werden. Hierzu genügt wiederholtes Klicken auf den Button, bis der gewünschte Schritt in der Ableitung wieder erreicht ist.

4.5.3 Prüfung eines Wortes

Analog zur manuellen Ableitung wird auch dieser Modus anhand des Beispielwortes „()()()“ dargestellt. Wie in Abschnitt 5.4.4.3 über die algorithmische Umsetzung bereits erwähnt wurde, wird über diesen Modus zum einen geprüft, ob das eingegebene Wort zur Sprache der Grammatik gehört und zum anderen werden für Worte, die zur Sprache gehören, alle Ableitungsbäume gebildet, die das Wort erzeugen.

Über die Menüauswahl Modus → Wort prüfen wird eine neue Wortprüfung gestartet, und es erscheint zunächst ein Dialog zur Eingabe des zu prüfenden Wortes. Über den Button „Weiter“ wird das Wort dann an den Algorithmus übergeben und dieser gestartet. Je nach Komplexität der Grammatik und der Länge des eingegebenen Wortes kann der Algorithmus gegebenenfalls eine etwas längere Laufzeit benötigen, so dass es zu einer Wartezeit kommt.

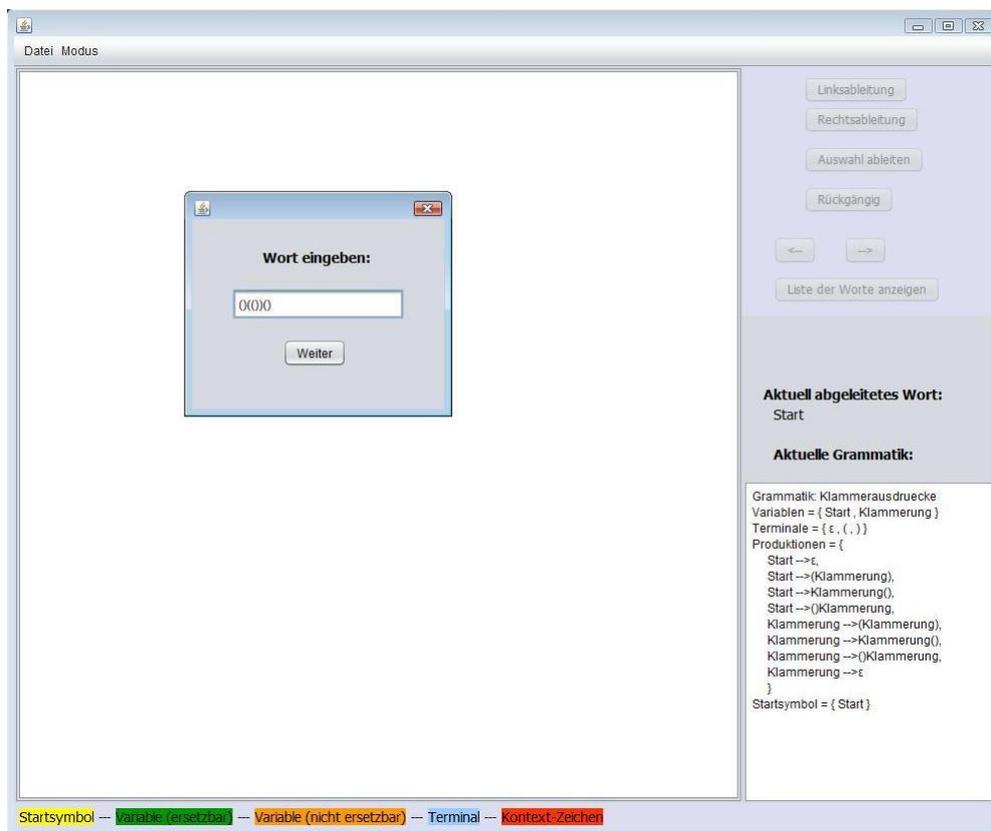


Abbildung 14: Beginnen einer Wort Prüfung

Konnte der Algorithmus erfolgreich enden, wird nun der erste von gegebenenfalls mehreren Ableitungsbäumen für das eingegebene Wort angezeigt. Ist die Grammatik für das eingegebene Wort mehrdeutig, sind nun im Bereich oben rechts entsprechend die Buttons zum Durchschalten der gefundenen Bäume aktiviert. Die Anzahl der gefundenen Bäume wird stets neben diesen Buttons angezeigt. Die Anzahl der gefundenen Bäume wird stets neben diesen Buttons angezeigt.

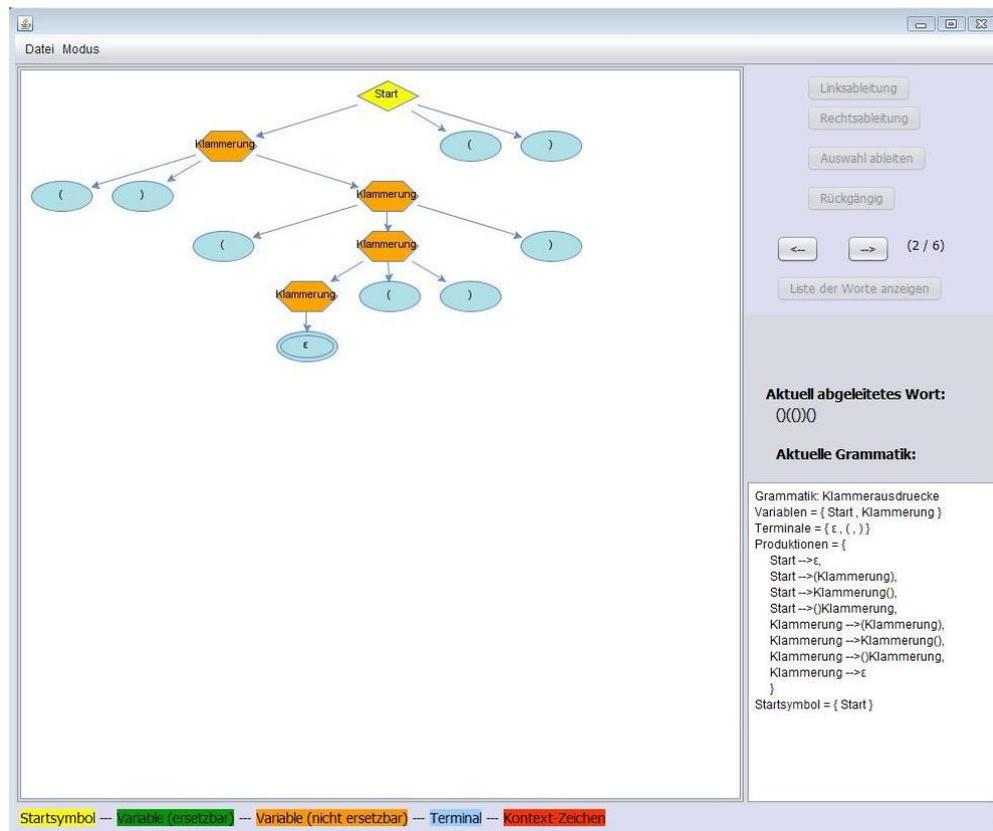


Abbildung 15: Ergebnis einer Wortprüfung

Bei gewissen Grammatiken und ab einer gewissen Länge des eingegebenen Wortes muss der Algorithmus eine zu große Zahl von Möglichkeiten untersuchen bzw. bilden. In der Folge wird der dem Programm zugewiesene Platz im Hauptspeicher nicht mehr ausreichen. Der Algorithmus bricht in diesem Fall ohne Ergebnis ab und es wird eine entsprechende Fehlermeldung angezeigt. Gehört das eingegebene Wort nicht zur Sprache der Grammatik und der Algorithmus konnte entsprechend keinen zum Wort passenden Ableitungsbaum bilden, wird ebenfalls eine entsprechende Meldung angezeigt.



Abbildung 16: Abbruchmeldung bei Speicherüberlauf



Abbildung 17: Meldung für $w \notin L(G)$

4.5.4 Erzeugen von Worten

Als Beispiel für diesen Modus dient die Erzeugung der Worte bis zu einer maximalen Länge von 8 Zeichen. Analog zu den anderen Modi wird die Erzeugung über die Menüauswahl Modus → Worte erzeugen gestartet und es erscheint ein Dialog zur Eingabe der gewünschten maximalen Länge. Wie beim Modus Wort prüfen wird über den Button „Weiter“ die Vorgabe an den Algorithmus gegeben und dieser gestartet. Wie bei der Prüfung von Worten kann es hier zu einer Wartezeit kommen, bis der Algorithmus endet.

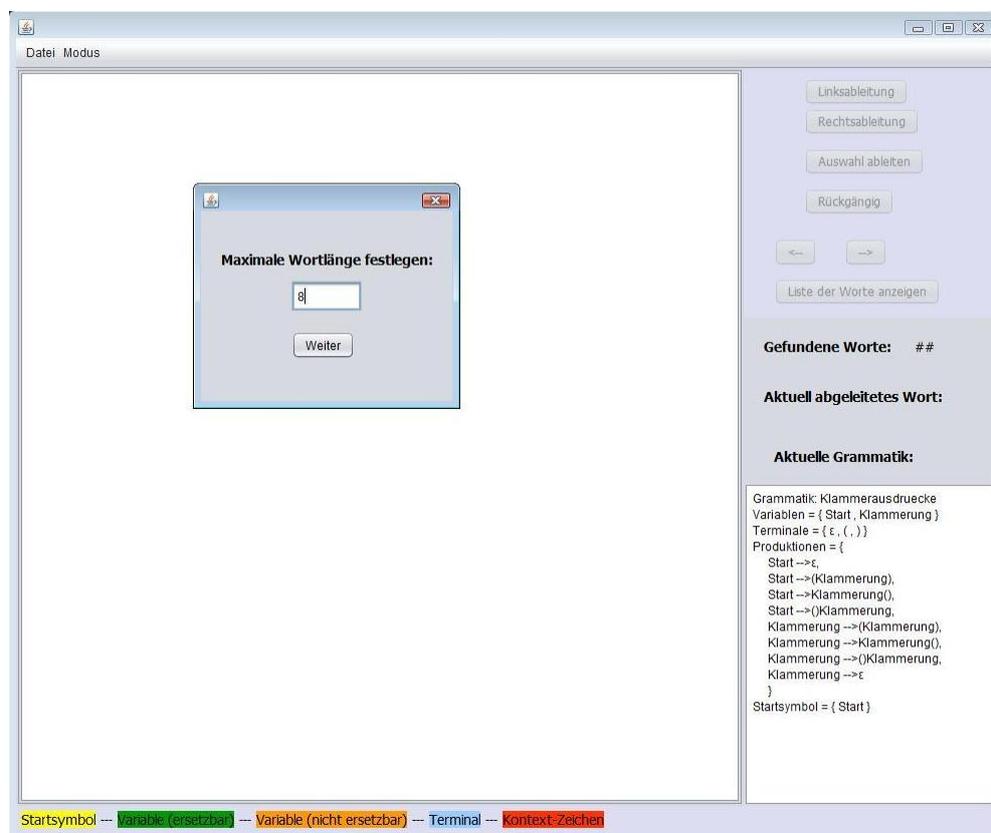


Abbildung 18: Beginn einer neuen Erzeugung von Worten

Kann der Algorithmus erfolgreich die Worte bis zur vorgegebenen Länge bilden, wird aus den Worten eine Liste gebildet und im Hauptfenster angezeigt. Bei mehrdeutigen Grammatiken sind die entsprechenden Worte mehrfach in der Liste vorhanden, da der Algorithmus auf der Bildung unterschiedlicher Bäume und nicht nur auf der Bildung unterschiedlicher Worte beruht. Im Anzeigenbereich wird zudem die Gesamtzahl der gefundenen Worte angezeigt,

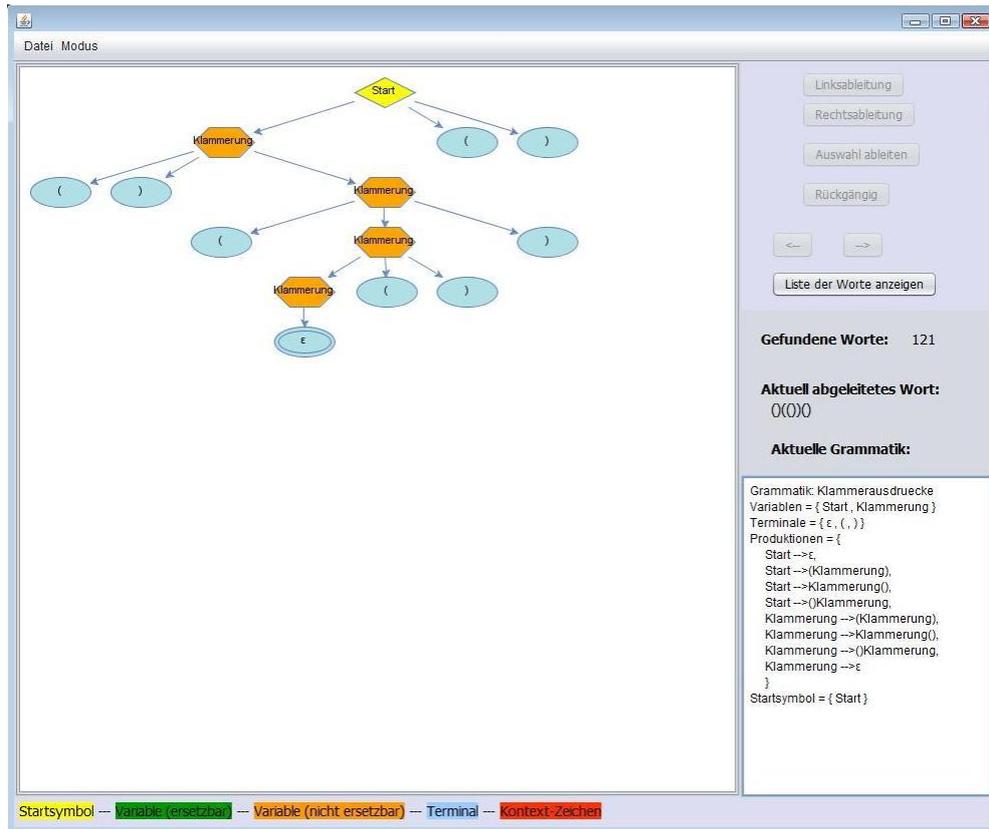


Abbildung 20: Baumansicht bei erzeugten Worten

In der Ansicht eines Ableitungsbaumes ist nun der Button „Liste der Worte anzeigen“ aktiv, über den der Benutzer, wie der Text impliziert, die Ansicht wieder auf die Liste der erzeugten Worte zurück schalten kann.

Wie auch beim Modus „Wort prüfen“ kann es bei der Erzeugung von Worten ab einer gewissen Längenvorgabe dazu kommen, dass der verfügbare Speicher nicht ausreicht. Auch hier bricht der Algorithmus ab und es wird eine entsprechende Fehlermeldung ausgegeben.



Abbildung 21: Abbruchmeldung bei Speicherüberlauf

4.5.5 Bekannte Problemfälle

Wie in den Abschnitten 4.5.3 und 4.5.4 angesprochen wurde, wird das Programm in gewissen Fällen bei der Ausführung der Funktionen „Wort prüfen“ und „Worte erzeugen“ einen Speicherüberlauf melden und die aktuelle Operation abbrechen.

Dieses Verhalten ist dem verwendeten Algorithmus für diese Funktionen geschuldet. In Abschnitt 4.4.4.3 und 4.4.4.4 wurde erläutert, dass der Algorithmus schrittweise jeden möglichen Ableitungsbaum erzeugt, um so alle Bäume für ein vorgegebenes Wort, oder für Worte mit einer vorgegebenen maximalen Länge, zu finden. Dies führt zwangsläufig zu einem exponentiellen Wachstum der Anzahl der Schleifendurchläufe im Algorithmus und ebenso der Anzahl der zu speichernden Baum-Objekte. In der Folge wird der verfügbare Platz im Hauptspeicher irgendwann nicht mehr ausreichen, was zwangsläufig zu einem Abbruch des Algorithmus durch die Java-VM führt. Der Nutzer wird dann entsprechend durch eine Fehlermeldung darauf hingewiesen.

In der Regel kann nach dem Bestätigen der Fehlermeldung normal mit dem Programm weiter gearbeitet werden.

Wann es zu so einem Abbruch kommt, kann nicht pauschal vorhergesagt werden. Das Wachstum der Anzahl an erstellten Bäumen ist immer von der Grammatik abhängig. Je nach der Gesamtanzahl der Produktionen in der Grammatik und abhängig davon wie viele Produktionen durchschnittlich in jedem Schritt angewendet werden können, kann auch bei relativ kurzen und vermeintlich „einfachen“ Worten das Programm bereits an seine Grenzen stoßen. Ein Beispiel für einen solchen Fall ist die dem Programm beigelegte Grammatik „Arithmetik2“. Zu dieser Grammatik können nur Worte mit maximal 9 Zeichen über die Algorithmen erzeugt werden. Das Wort „(a+a)*(a+a)“ kann nicht erfolgreich von „Wort prüfen“ verarbeitet werden, obwohl eine Ableitung für das Wort aus den Regeln der Grammatik leicht ersichtlich ist.

Es ist grundsätzlich ratsam sich von unten an die Grenze für die maximale Wortlänge heranzutasten. Am einfachsten gelingt dies über die Funktion „Worte erzeugen“, da bei dieser die maximale Wortlänge direkt vorgegeben wird. Nach Wunsch kann auch der, für das Programm verfügbare, Hauptspeicher vergrößert werden. Als Standardeinstellung wird von der Java-VM ein Viertel

des verfügbaren Hauptspeichers reserviert. Durch eine Batchdatei kann die Größe manuell gewählt werden. Anleitungen zur Erstellung einer solchen Batchdatei sind im Web verfügbar.

Neben diesem speziellen Problem, ist bekannt, dass es zu Problemen mit Java-Programmen unter Windows 8 und 8.1 kommt. Unter anderem kann es notwendig sein eine Batchdatei zu erstellen, um das Programm starten zu können. Zudem wurde beobachtet, dass die oben beschriebene Fehlermeldung nicht korrekt angezeigt wird. Dadurch wird das Programm blockiert und kann nur noch über den Task-Manager beendet werden.

Die Ursache für dieses Problem liegt meiner Kenntnis nach beim Betriebssystem und kann daher derzeit nicht behoben werden.

5 Abbildungsverzeichnis

1	Mengenhierarchie der formalen Sprachen.	11
2	Ableitungsbaum für das Wort $w = a + (a + a) * a$	23
3	Ringschluss	32
4	Erweiterung der Mengenhierarchie.	33
5	Ableitungsbaum für das Wort $acabcb$ aus Beispiel 14	42
6	Benutzeroberfläche des Grammatik-Editor in Startkonfiguration ...	54
7	Benutzeroberfläche des Hauptfensters	67
8	Eingabe der Variablen und Terminale	69
9	Erstellen der Produktionen	70
10	Beginnen einer manuellen Ableitung	72
11	Ersetzung einer ausgewählten Variable	73
12	Farbcodierung von Variablen	73
13	Endzustand einer manuellen Ableitung	74
14	Beginnen einer Wort Prüfung	75
15	Ergebnis einer Wortprüfung	76
16	Abbruchmeldung bei Speicherüberlauf	77
17	Meldung für $w \notin L(G)$	77
18	Beginn einer neuen Erzeugung von Worten	78
19	Ergebnisliste für erzeugte Worte	79
20	Baumansicht bei erzeugten Worten	80
21	Abbruchmeldung bei Speicherüberlauf	81

6 Literaturverzeichnis

- [Sch09] Schöning, Uwe: *Theoretische Informatik – kurz gefasst*. Heidelberg: Sketrum Akademischer Verlag, 5. Auflage, 2009.
- [He12] Hedtstück, Ulrich: *Einführung in die Theoretische Informatik*. München: Oldenburg Verlag, 5. Auflage, 2012.
- [LpInf] Hessisches Kultusministerium: *Lehrplan Informatik*. Gymnasialer Bildungsgang, Gymnasiale Oberstufe.
- [We05] Wegener, Ingo: *Theoretische Informatik: eine algorithmenorientierte Einführung*. Wiesbaden: Teubner Verlag, 3. überarb. Auflage, 2005.
- [JFL] *JFLAP*. Verfügbar unter: www.jflap.org [21.01.2014]
- [MAC] *Machines*. Verfügbar unter: <http://zeus.fh-brandenburg.de/~socher/tgi/> [21.01.2014]
- [ATCC] *AtoCC*. Verfügbar unter: www.atocc.de [21.01.2014]
- [JGX] *JGraphX*. Verfügbar unter: www.jgraph.com [21.01.2014]

7 Erklärung

Ich versichere hiermit, dass ich die Arbeit selbstständig verfasst, keine anderen als die angegebenen Hilfsmittel verwandt und die Stellen, die anderen benutzten Druck- und digitalisierten Werken im Wortlaut oder dem Sinn nach entnommen sind, mit Quellenangaben kenntlich gemacht habe.