



Bachelorarbeit

Visualisierung des LR(1)-Verfahrens zur Erkennung der deterministisch kontextfreien Sprachen

**Visualization of the LR(1)-Method for Recognizing the Deterministic
Context-Free Languages**

zur Erlangung des akademischen Grades

Bachelor of Science (B. Sc.)

vorgelegt dem

Fachbereich Mathematik und Informatik, Physik, Geographie
der Justus-Liebig Universität Gießen

von

Uranik Berisha

im September 2021

Referent: Prof. Dr. Martin Kutrib

Abstract

Parser sind in der Informatik weit verbreitet und sind eine der Schlüsselkomponenten für den Compilerbau, sodass diese meistens im Laufe des Informatikstudiums gelehrt werden. Deren Theorie ist für Studierende jedoch sehr komplex und bedarf daher weitere Lehrmittel. Zur Lösung dieses Problems wurde in einem vorangegangenen Projekt ein Lernprogramm in Form eines Parsergenerators entwickelt, welche das LR(1)-Verfahren implementiert und lehrt.

Ziel dieser Arbeit ist es, diesen Parsergenerator auf die Richtigkeit des Verfahrens und die Nützlichkeit des Lernprogrammes zu prüfen und damit das Projekt zu evaluieren. Dafür werden der Aufbau und die Funktionsweise des Programmes mithilfe der Theorie der formalen Sprachen analysiert und die Erkenntnisse aufbereitet.

Mit den gewonnenen Erkenntnissen wurde in dieser Bachelorarbeit gezeigt, dass der Parsergenerator das gewünschte LR(1)-Verfahren korrekt implementiert und durch viele Hilfestellungen auch als Lernprogramm nutzbar ist. Dabei konnte die Nutzbarkeit zwar nur theoretisch argumentiert werden und nicht durch praktische Evaluierungsmethoden statistisch belegt werden, nichtsdestotrotz besteht die Annahme, dass das Programm durchaus auch in der Praxis als Lernprogramm nutzbar ist.

Inhaltsverzeichnis

1	Grundlegendes zur Bachelorarbeit	1
1.1	Visualisierung durch den LR(1)-Parsergenerator	2
2	Einleitung	5
2.1	Motivation: Wofür werden Parser benötigt?	5
2.2	Problemstellung: Welches Problem löst der LR(1)-Parsergenerator?	6
2.3	Zielsetzung der Bachelorarbeit	7
3	Stand der Technik	9
3.1	Wie lassen sich Parsergeneratoren kategorisieren?	9
3.2	Welche Forschung gibt es für Parser?	11
4	Theorie der Formalen Sprachen	15
4.1	Kontextfreie Grammatik	15
4.1.1	Alphabet & Wörter	15
4.1.2	Grammatik	17
4.1.3	Ableitungen & Reduktionen	19
4.1.4	First- & Follow-Mengen	21
4.2	Kontextfreie Sprachen	23
4.2.1	Sprachen	23
4.3	Deterministische endliche Automaten	26
4.3.1	Kellerautomaten	26
4.3.2	Übergänge	31
4.3.3	Deterministische Sprache	33
4.4	LR(1)-Verfahren	35
4.4.1	Bestandteile des LR(1)-Verfahrens	35
4.4.2	Konstruktion des LR(1)-Parsers	38
5	Analyse des LR(1)-Parsergenerators	41
5.1	Datenstrukturen	42
5.1.1	Grammatik	43
5.1.2	First- & Follow-Mengen	46
5.1.3	LR(1)-Verfahren	47
5.2	Algorithmen des LR(1)-Verfahren	51
5.2.1	First	52
5.2.2	Follow	55
5.2.3	Hülle	58
5.2.4	Sprung	60

5.2.5	Zustandsberechnung	61
5.3	Funktionsweise der Interaktion	65
5.3.1	Abfrage- und Korrekturfunktion	65
5.3.2	Warnungen und Fehlermeldungen	66
6	Evaluierung des LR(1)-Parsergenerators	71
6.1	Funktionale Anforderungen	71
6.2	Nicht-Funktionale Anforderungen	72
7	Schluss	75
7.1	Zusammenfassung	75
7.2	Ausblick	76
	Literaturverzeichnis	77

Abbildungsverzeichnis

1.1	Abbildung: Website	3
4.1	Beispiel: Kellerautomat	28
5.1	Datenstruktur: Terminalsymbole	43
5.2	Datenstruktur: Terminalsymbole	44
5.3	Datenstruktur: Produktionsregeln	45
5.4	Datenstruktur: First- und Follow-Mengen	46
5.5	Datenstruktur: Zustände	48

1 Grundlegendes zur Bachelorarbeit

Diese Bachelorarbeit baut auf die Projektphase (Modulcode: [CS3004](#)) an der Technischen Hochschule Mittelhessen auf. Im Rahmen dieser Projektphase wurde in Zusammenarbeit mit Herrn Thomas Moissl und unter der Betreuung von Herrn Prof. Dr. Uwe Meyer das Ziel verfolgt, das sogenannte SLR(1)-Verfahren¹ zu visualisieren. Diese Visualisierung sollte zeitgleich als interaktives e-Learning-Programm für das Unterrichten von SLR(1)-Parsern² dienen.

Genauer wurde in diesem Projekt ein Generator für Parser implementiert, welches aus einer gegebenen Grammatik eine interaktive Visualisierung des entsprechenden Parsers generiert. Dieses kann im Themengebiet des Compilerbaus von Informatik-Studierenden genutzt werden, um das Verständnis der SLR(1)-Parser zu vertiefen. Das Projekt soll unter anderem als Hilfestellung für die Prüfungsvorbereitung dienen und wurde in Form einer Website ermöglicht, die unter:

<https://parser-generator.github.io>

erreichbar ist.

Die Website erlaubt durch die Eingabe einer Grammatik³ die schrittweise Generierung eines Parsers. Dabei wird der Nutzer bezüglich einzelner Schritte abgefragt, bis die Visualisierung als interaktive Parsertabelle ausgegeben wird.

Eine genauere Analyse des SLR(1)-Parsergenerators lässt sich in der Bachelorarbeit „Visualisierung des SLR(1)-Verfahrens für die Erkennung einer Teilmenge der deterministisch kontextfreien Sprachen“ von Herrn Thomas Moissl durchgeführt.

Diese Arbeit beschäftigt sich mit einer Erweiterung des vorhandenen Parsergenerators. Vorbereitend wurde daher das Projekt um das sogenannte LR(1)-Verfahren⁴ erweitert, um auch dieses visualisieren zu können. Dafür wurde die Website um eine neue Option erweitert, welche den Nutzern die Möglichkeit gibt, zu bestimmen, welches Verfahren verwendet werden soll.

¹Das SLR(1)-Verfahren wird in *Practical translators for LR(k) languages* von Franklin Lewis DeRemer (siehe [6]) definiert und ist ein Verfahren zur Erkennung einer Teilmenge der formalen Sprachen (formale Sprachen werden in Kapitel 4.2.1.1 definiert).

²Parser meinen ausführbare Programme, welche die Analyse und Verarbeitung von Eingaben übernehmen und diesen Strukturbeschreibungen befügen. Eine genauere Definition kann in [18] gefunden werden. SLR(1)-Parser sind Parser, welche das SLR(1)-Verfahren verwenden.

³Formale Grammatiken wird in 4.1.2.1 definiert.

⁴Das LR(1)-Verfahren wird in Kapitel 4.4 definiert.

Diese erweiterte Website ist die Grundlage dieser Bachelorarbeit und soll hier ausführlich vor dessen theoretischen Hintergrund analysiert werden.

1.1 Visualisierung durch den LR(1)-Parsergenerator

Da diese Arbeit auf dem entwickelten LR(1)-Parsergenerator aufbaut, wird an dieser Stelle die Funktionsweise des Generators beleuchtet.

Als Erweiterung des SLR(1)-Parsergenerators verfolgt der LR(1)-Parsergenerator die gleichen Ziele, weswegen nur inhaltliche Änderungen zum Vorgänger vorliegen. Insbesondere besteht eines dieser Ziele darin ein e-Learning-System⁵ zu entwickeln, welches eine interaktive Visualisierung erstellt, in das eigene Lösungsvorschläge eingegeben werden können. Diese Eingaben können anschließend von dem e-Learning-System korrigiert werden. Demnach kann das Programm für Studierende als ein hilfreiches Mittel fungieren, um die Konzepte des LR(1)-Parsers zu erlernen. Motiviert wird dieses Ziel durch die Bedeutsamkeit des Themas für den Compilerbau und der Tatsache, dass die Thematik zu Beginn durchaus komplex sein kann (siehe 2.2).

Zu der Funktionsweise ist wichtig anzumerken, dass diese mit Fokus auf die Nutzerfreundlichkeit umgesetzt ist, da dies ein wichtiger Erfolgsfaktor eines e-Learning-Systems ist (vgl. [15] S. 73). So ist beispielsweise das System als eine Website umgesetzt, um die Erreichbarkeit zu erhöhen. Des Weiteren wird die Abfrage der verschiedenen Schritte des LR(1)-Verfahrens stets in einer festgelegten Struktur geführt, welche wie folgt umgesetzt ist.

Anfangs hat der Nutzer die Option, eine eigene Grammatik einzugeben und sich den entsprechenden LR(1)-Parser zu dieser Grammatik sich anzeigen zu lassen. Falls der Nutzer keine Grammatik eingeben möchte, sind Beispielgrammatiken vorhanden, die genutzt werden können, indem eine ausgewählt wird.

Nach der Eingabe oder der Auswahl einer Grammatik erhält der Nutzer anschließend die Möglichkeit, sich die Schritte der Parsergenerierung nacheinander anzeigen zu lassen.

Mittels dieser Schritte wird der Nutzer zuerst aufgefordert, eigene Lösungsvorschläge zu erarbeiten und diese einzugeben. Nachdem die Eingabe erfolgt ist, kann der Nutzer sich dazu entscheiden, entweder die eigenen Fehler oder die korrekte Lösung anzeigen zu lassen. Als zusätzliche Hilfestellung bietet die Website die Möglichkeit, durch Klicken auf „Hilfe“ Pop-Ups aufzurufen, welche Anweisungen zu dem nächsten Schritt beinhalten. Diese Pop-ups sind kontextabhängig gestaltet und bieten für die jeweiligen Schritte entsprechende Nachrichten an, um die Nutzerfreundlichkeit weiter zu erhöhen (vgl. [20] S. 39).

Anschließend wird eine interaktive Visualisierung in Form einer Parsertabelle ausgegeben. Dabei erfolgt die Interaktion des Nutzers, in Form eines selbstständigen Herleitens der Parsertabelle, mit der Option, die eigene Lösung in die vorhandene Tabelle einzufügen. Hier kann erneut, durch entsprechende Optionen, eine Überprüfung durchgeführt werden oder die richtige Parsertabelle eingetragen werden. Im letzteren Fall werden jedoch falsche

⁵e-Learning beschreibt eine Art des Lernens mithilfe von elektronischen Hilfsmitteln wie digitalen Medien (vgl. [12] S. 6).

1.1 Visualisierung durch den LR(1)-Parsergenerator

Tabelleneinträge dennoch hervorgehoben, um die Aufmerksamkeit des Nutzers auf diese Stellen zu richten.

Zusammenfassend soll die geführte Bedienung und die mögliche Hilfestellung zu einer leicht bedienbaren Website führen, welche durch die Korrekturfunktion das Lernen unterstützen soll. Eindeutige Anweisungen dienen gleichzeitig dazu, die Effektivität des Lernens zu erhöhen (vgl. [20] S. 39).

Beschreibung

Diese Website soll als Hilfsmittel für das Lernen für Compilerbau dienen. Es können kontextfreie Grammatiken angegeben werden, dafür auch die First-/Followmengen, und die LR(1)-Parserstabelle, welche korrigiert werden. Die Zustandstabelle wird automatisch berechnet.

Anleitung

In der Grammatik müssen Symbole durch Leerzeichen getrennt werden. First und Followmengen können mit "f" oder "F" getrennt werden. Shift/Reduce wird mit "s/r" und der entsprechenden Zahl angegeben, die Endaktion mit "fertig". Durch "f" können mehrere Regeln in einem Feld angegeben werden. "-" ist das leere Symbol.

Beispiele

Kleine Grammatik
Mathematische Ausdrücke
Richtig gesetzte Klammern
Sprache mit ungleich vielen 'a' und 'b'
Übungsgrammatik 1
Übungsgrammatik 2

Eingabe

S' → Expr
Expr → Expr + Term
Expr → Term
Term → Term * Faktor
Term → Faktor
Faktor → (Expr)
Faktor → Konst

First

Expr: (, Konst
Term: (, Konst
Faktor: (, Konst

Zustand	$I_0 = \text{Hülle(Startregel)}$	$I_1 = \text{Sprung}(I_0, \text{'Expr'})$	$I_2 = \text{Sprung}(I_0, \text{'Term'})$	$I_3 = \text{Sprung}(I_0, \text{'Faktor'})$	$I_4 = \text{Sprung}(I_0, \text{'('})$	$I_5 = \text{Sprung}(I_0, \text{'Konst'})$	$I_6 = \text{Sprung}(I_0, \text{'+'})$	I_7
Elemente	[S' → Expr; \$] [Expr → Expr + Term; \$ +] [Expr → Term; \$ +] [Term → Term * Faktor; \$ +] [Term → Faktor; \$ +] [Faktor → (Expr; \$ +] [Faktor → Konst; \$ +]	[S' → Expr; \$] [Expr → Expr + Term; \$ +] [Expr → Term; \$ +]	[Expr → Term; \$ +] [Term → Term * Faktor; \$ +] [Term → Faktor; \$ +]	[Term → Faktor; \$ +]	[Faktor → (Expr; \$ +] [Expr → Expr + Term; \$ +] [Expr → Term; \$ +] [Term → Term * Faktor; \$ +] [Term → Faktor; \$ +] [Faktor → (Expr; \$ +] [Faktor → Konst; \$ +]	[Faktor → Konst; \$ +]	[Expr → Expr + Term; \$ +] [Term → Term * Faktor; \$ +] [Term → Faktor; \$ +] [Faktor → (Expr; \$ +] [Faktor → Konst; \$ +]	

Aktion

Zustand	+	(	)	Konst	\$	Expr	Term	Faktor
I_0		s4		s5		1	2	3
I_1	s6				Fertig			
I_2	s7				r2			
I_3	r4				r4			
I_4		s1		s12		8		10
I_5	s8			r6				
I_6		s4		s2			13	3
I_7		s4		s5				14
I_8	s16			s15				
I_9	r2	s17		r2				
I_{10}	r4			r4				
I_{11}		s11		s3		18	9	10
I_{12}	r6	r6		r6				
I_{13}	r1	s7			r1			
I_{14}	r3				r3			
I_{15}	s5	r5			r5			
I_{16}		s11		s12			19	10
I_{17}		s11		s12				25
I_{18}	s16			s21				
I_{19}	r1	s17						
I_{20}	r3							
I_{21}	r5	r5		r5				

Abbildung 1.1: Bildschirmaufnahme der Website

2 Einleitung

2.1 Motivation: Wofür werden Parser benötigt?

Das Verständnis von Parsern ist in der Informatik von großer Bedeutung. Sie werden für die Analyse und Verarbeitung von Eingaben (vgl. [18] S. 4) in einigen Bereichen der Informatik genutzt und sind daher ein fester Bestandteil in nahezu jedem Informatikstudium, wie in den öffentlich verfügbaren Modulbeschreibungen einiger Hochschulen gesehen werden kann (vgl. [26]).

Der Einsatz der Parser liegt meist im Bereich der Syntaxanalyse, in welcher Kontext-Beschreibungen für die zu verarbeitende Eingabe erzeugt werden. Somit wird die Eingabe durch ein bestimmtes Schema strukturiert und in der entsprechend vordefinierten Struktur umgesetzt. (vgl. [18] S. 4)

Zwar werden Parser in der Entwicklung oftmals innerhalb eines Projekts für das Interpretieren von Eingaben genutzt, jedoch lassen sich die bekanntesten Beispiele vor allem in den Programmiersprachen finden. Dort werden Parser genutzt, um den Quellcode für den Compiler aufzubereiten, indem der Code syntaktisch analysiert wird und diesem damit eine Strukturbeschreibung beigelegt wird. (vgl. [18] S. 3)

Auch textbasierte Auszeichnungssprachen wie HTML oder XML sind als reine Textform für den Gebrauch durch den Rechner ungeeignet. Die hierarchische Struktur muss zuerst durch die Anwendung eines Parsers analysiert werden und in eine, für den Rechner nutzbare, Form übersetzt. (vgl. [18] S. 4)

Wenngleich die Bedeutung des Wortes „Parser“ auf ein implementiertes, ausführbares Programm Bezug nimmt (vgl. [18] S. 4), dient das Parsen an sich jedoch nicht nur der Anwendung in der Softwareentwicklung, stattdessen kann das Konzept des Parsens auch außerhalb der Informatik Anwendung finden.

„Der Begriff ‚Parsing‘ wird heutzutage in vielen Disziplinen wie dem Compilerbau, der kognitiven Psychologie, der Computerlinguistik, sowie der sprachorientierten KI und der Sprachtechnologie [...] verwendet.“ ([18] S. 3)

In der Linguistik können Parser beispielsweise genutzt werden, um an von Menschen gesprochene Sprachen zu forschen. (vgl. [18] S. 3)

Insgesamt lässt sich beobachten, dass das Verständnis von Parsern besonders für Informatiker, aber auch in diversen anderen Gebieten von großem Nutzen sein kann. Aus diesem Grund ist nicht nur die weiterführende Forschung von Bedeutung, sondern auch die breite

Zugänglichkeit.

Ziel des LR(1)-Parsergenerators ist es, durch die interaktive Visualisierung einen Einstieg in die Theorie der Parser zu bilden und somit das Wissen zu verbreiten. Aufgrund der Bedeutsamkeit der Parser wird in dieser Bachelorarbeit einhergehend der theoretische Hintergrund aus der Informatik im Teilgebiet der formalen Sprachen¹ behandelt. Da die Theorie der formalen Sprachen als Ganzes zu umfangreich ist, beschränkt sich diese Arbeit konkret auf die Klasse der deterministisch kontextfreien Sprachen². Zumal diese eine Untermenge der allgemeineren kontextfreien Klassen³ (auch Typ-2-Sprachen der Chomsky-Hierarchie⁴) bilden, werden auch die kontextfreien Sprachen bei der Herleitung aufgegriffen. Dies erleichtert das allgemeine Verständnis der Parser und wird daher auch mit der oben genannten Bedeutsamkeit des Parsens motiviert.

2.2 Problemstellung: Welches Problem löst der LR(1)-Parsergenerator?

Da es bereits einige öffentlich verfügbare Parsergeneratoren gibt⁵ und einige sogar als Website leicht und sofort erreichbar sind, stellt sich die Frage nach der Notwendigkeit eines weiteren Generators. Selbst der Aspekt der Visualisierung wird in den meisten Generatoren aufgegriffen.

Das Ziel, welches der in Kapitel 1 genannte Parsergenerator zu erreichen versucht, ist jedoch nicht nur die Erstellung einer schlichten Visualisierung. Stattdessen soll eine interaktive Art der Visualisierung umgesetzt werden. Hierbei ist es die Absicht dem Nutzer durch eine praxisorientierte Anwendungsmöglichkeit, zu einem tieferen Verständnis zu verhelfen. Begründet wird dieses Ziel durch folgende Problemstellung.

Wie bereits in der Motivation 2.1 erwähnt, werden Studierende in fast allen Informatikstudiengängen in Compilerbau unterrichtet (vgl. [26]). Dieses gibt eine Einführung in vielerlei Gebiete, die für die Entwicklung von Compilern wichtig sind. Zu diesen Gebieten gehören die lexikalische, syntaktische und semantische Analyse des Codes, welche für die Übersetzung des Quellcodes benötigt werden (vgl. [19]). Für die syntaktische Analyse werden dabei Parser genutzt und dementsprechend auch im Compilerbau-Modul behandelt (vgl. [26]). Dabei werden formale Sprachen als Grundlage eingeführt, um unter anderem auch LR-Verfahren als Standard für die Erkennung deterministisch kontextfreier Sprachen zu erklären. [26]

Da dieser Teilbereich der theoretischen Informatik für Studierende sehr komplex ist, werden in der Lehre SLR(1)-Parser genutzt, um die Studierenden an die LR-Parser heranzuführen

¹Die formalen Sprachen werden in 4.2.1.1 definiert.

²Die deterministisch kontextfreien Sprachen werden in 4.3.3.3 definiert.

³Die kontextfreien Sprachen werden in 4.2.1.3 definiert.

⁴Dieser Begriff geht auf Noam Chomsky zurück, welcher in *Three models for the description of language*. formale Sprachen hierarchisch klassifiziert (siehe [5]).

⁵Öffentlich verfügbare Parsergeneratoren werden in Kapitel 3.1 behandelt.

(vgl. [26]). Dies liegt an dem simpleren SLR(1)-Verfahren, welches auf Grund seiner einfachen Funktionalität als einführendes Verfahren geeigneter ist.

Allerdings besteht die Problematik, dass die SLR(1)-Parser ein formal schwächeres Verfahren nutzen (vgl. [6]), sodass das Interesse an den LR-Parsern weiterhin bestehen bleibt. Da die Konzepte des LR(1)-Verfahrens gleichwohl kompliziert sind, bleibt die Aneignung der Verfahrensweise ohne interaktive Möglichkeiten schwierig.

Diese interaktive Möglichkeit bietet der LR(1)-Parsergenerator. Daneben ist zu beachten, dass es viele Parsergeneratoren gibt, die auch Visualisierungen anbieten. Solche sind jedoch nicht für das Ziel optimiert worden, das Verständnis der Nutzer für die unterliegenden Verfahren zu erhöhen. Dementsprechend bieten diese Parsergeneratoren keine weitere Hilfestellungen an und die entsprechenden Visualisierungen sind auch nicht interaktiv gestaltet (siehe 3.1).

2.3 Zielsetzung der Bachelorarbeit

Ziel dieser Bachelorarbeit ist es, den entwickelten LR(1)-Parser hinsichtlich der Funktionen als Parsergenerator zur Erkennung deterministisch kontextfreier Sprachen und der Funktionen als e-Learning-System zu evaluieren. Dafür findet im Hauptteil dieser Bachelorarbeit eine ausführliche Analyse des LR(1)-Parsergenerators statt, um mithilfe dieser Erkenntnisse die Evaluierung durchzuführen. Bei der Analyse sollen insbesondere folgende Aspekte betrachtet werden:

1. Der **Aufbau**: *Welche Datenstrukturen wurden für die Visualisierung verwendet?*
2. Die **Algorithmen** des LR(1)-Verfahren: *Wie wurden die Algorithmen des LR(1)-Verfahren umgesetzt?*
3. Die **Funktionsweise** der Interaktion: *Wie wurden die interaktiven Aspekte umgesetzt?*

Die Analyse beinhaltet die Erläuterung der Aspekte und eine entsprechende Begründung für die Art der Umsetzung dieser Aspekte.

Um die Analyse durchzuführen, muss zuerst eine Basis mithilfe des theoretischen Hintergrundes geschaffen werden, welche für die drei Aspekte aus den folgenden Gründen benötigt werden:

1. Für den Aufbau werden hauptsächlich formale Definitionen der bereits vorhandenen Literatur benötigt. Dafür wird in Kapitel 4 die gesamte benötigte Theorie der formalen Sprachen eingeführt.

2. Um die Änderungen der Algorithmen zu verstehen werden außerdem noch grundlegende Eigenschaften der deterministisch kontextfreien Sprachen und des LR(1)-Verfahrens benötigt. Diese werden auch in Kapitel 4 vorgestellt und sind meist als Bemerkung gekennzeichnet.
3. Eine Erklärung der Funktionsweise der Interaktion, benötigt an sich keine weitere Theorie, jedoch ist durch die gesamte Arbeit hinweg ein Verständnis der Begriffe notwendig, sodass im Kapitel 4 eine Begriffserklärung in Form von Definitionen und Anmerkungen vorhanden ist.

Um schließlich den entwickelten Parsergenerator evaluieren zu können, werden in Kapitel 3 existierende Visualisierungen und Parsergeneratoren betrachtet, und inwiefern sich diese vom entwickelten LR(1)-Parsergenerator unterscheiden. Dies wird benötigt, da nicht nur die in der Einleitung gesetzten Anforderungen erfüllt werden sollen, sondern weil das Projekt auch einen Mehrwert im Vergleich zu den anderen Generatoren liefern soll.

Aufbau der Arbeit Die Einführung beginnt mit der Theorie der Grammatiken, danach werden Sprachen, insbesondere deterministisch kontextfreie Sprachen, mithilfe der Theorie der Kellerautomaten hergeleitet. Zum Schluss der Theorie wird das LR(1)-Verfahren erläutert. (Kapitel 4)

Nachdem die theoretischen Grundlagen etabliert sind, können die gewählten Datenstrukturen erklärt werden und deren Umsetzung begründet werden. (Kapitel 5.1)

Da der Aufbau der Datenstrukturen eine große Auswirkung auf die Umsetzung der Algorithmen hat, werden notwendige Änderungen im Anschluss aufgezeigt, und deren Korrektheit wird mithilfe der behandelten Eigenschaften aus der Theorie verifiziert. (Kapitel 5.2)

Anschließend wird die Interaktion des Nutzers mit dem Programm genauer erläutert und die Funktionsweise einiger wichtiger Aspekte begründet. Diese Aspekte beinhalten die Abfragefunktionalität und die Fehlermeldungen an den Nutzer. (Kapitel 5.3)

Die Evaluation des gesamten Projekts bildet den Abschluss der Arbeit und nutzt die gewonnenen Erkenntnisse aus dem Hauptteil 5 für die Auswertung und den Stand der Technik 3 für den Vergleich mit anderen Parsergeneratoren. (Kapitel 6)

3 Stand der Technik

3.1 Wie lassen sich Parsergeneratoren kategorisieren?

Im Folgenden werden bereits vorhandene und bekannte Arten von öffentlich zugänglichen Parsergeneratoren benannt, um Vergleiche zwischen dem LR(1)-Parsergenerator und anderen Generatoren zu ermöglichen.

Es gibt vielerlei Arten von öffentlich verfügbaren Parsergeneratoren, welche zu einer vordefinierten Grammatik einen Parser erstellen. Dabei werden die Generatoren im Rahmen dieser Arbeit, ungeachtet der spezifischen Umsetzung, grob in drei nutzungsbezogene Kategorien einordnen. Diese Kategorien sollen keine Auskunft über die Verfahrensweise der Konstruktion oder den erstellten Parser geben, stattdessen soll damit die Nutzungsart beschrieben werden.

Kategorie 1 Einerseits gibt es Parsergeneratoren, welche Parser für die Nutzung von Entwicklern generieren, die also für ein gegebenes Projekt in der Entwicklung genutzt werden können. Bei dieser Art von Generatoren werden Parser erstellt, welche in das Produkt eingebunden werden können, um Eingaben einzulesen und schließlich weiterzuverarbeiten. Die bekanntesten Vertreter dieser Art von Generatoren sind „yacc“ und „bison“. Das sind frei verfügbare Parsergeneratoren, die kontextfreie Grammatiken in Programme übersetzen können, welche in den Programmiersprachen C, C++ oder Java als Parser genutzt werden können. (vgl. [19] S. 201)

Kategorie 2 Andererseits gibt es auch Parsergeneratoren, deren Nutzung nur zum Wissenserverb oder in der Forschung der Konzepte liegen. Hier werden die Parser nicht als Programm realisiert, sondern in Form einer abstrakten Repräsentation ausgegeben. Gleichzeitig soll hier nicht zwischen den Arten der Visualisierung unterschieden werden, vielmehr kann beispielsweise nur der Parser selbst visualisiert werden oder sogar die gesamte Generierung als Prozess. Diese Parser sind nicht für die Nutzung gedacht und bieten dementsprechend auch keine Option, Eingaben zu tätigen. Aus diesem Grund liegt der Fokus dieser Kategorie nicht im Parsen selbst. Stattdessen kann die Visualisierung verwendet werden, um Einblicke in den Prozess der Parsergenerierung oder sogar in die Eigenschaften einer Grammatik zu erhalten.

Vertreter dieser Kategorie können oftmals im Internet gefunden werden, so zum Beispiel eine Website, welche das LR(1)-Verfahren mithilfe einer Tabelle visualisiert (siehe [25]).

Kategorie 3 Die dritte Kategorie beinhaltet Eigenschaften von beiden Kategorien. Von Generatoren in dieser Kategorie werden Parser erstellt, welche Eingaben tatsächlich parsen können, aber für das Erlernen oder das Erforschen der Theorie genutzt werden. Obwohl in dieser Kategorie der Nutzen auch im Parsen von Eingaben liegt, sind diese von der ersten Kategorie zu unterscheiden. Der Unterschied liegt darin, dass die Nutzung nicht für die Integration in ein bestehendes Projekt beabsichtigt ist. Stattdessen wird hier lediglich die geparsete Eingabe visualisiert, wobei auch an dieser Stelle durchaus mehrere Aspekte visualisiert werden können. Beispiele sind das Parsen als Verfahren zu visualisieren oder die geparsete Struktur der Eingabe visuell darzustellen. Diese Parser werden beispielsweise dazu verwendet, eine gegebene Eingabe zu strukturieren, um damit entweder Erkenntnisse über die Eingabe oder über den Parsingprozess zu gewinnen.

Ein Beispiel für diese Art von Parsergeneratoren und deren Entwicklung ist in „CUPV - a visualization tool for generated parsers“ von Alan Kaplan und Denise Shoup gegeben (vgl. [11]).

Auch Parsergeneratoren, die zeitgleich Eigenschaften der zweiten und dritten Kategorien besitzen, wurden durch Projekte an einigen Universitäten wie Princeton und Stanford erschaffen. Diese erstellen eine Visualisierung des Parsergenerators und erlauben gleichzeitig Eingaben, welche durch den gegebenen Parser wiederum geparkt werden können. (siehe [13], [23])

Das am weitesten entwickelte Werkzeug *JFLAP - Java Formal Languages and Automata Package* der Duke University ist zwar nicht als Website aufrufbar, ist jedoch aufgrund der umfangreichen Funktionalität weit verbreitet. [21]

Kategorie des LR(1)-Parsergenerators Insgesamt dienen diese Kategorien dazu, die Unterschiede des LR(1)-Generators zu anderen Parsergeneratoren zu verdeutlichen. Der LR(1)-Parsergenerator erstellt keinen funktionsfähigen Parser, sondern lediglich eine abstrakte Repräsentation davon als Parsertabelle. Dementsprechend gehört dieser zwangsweise zur zweiten Kategorie, erweitert die Eigenschaften der Kategorie jedoch um interaktive Elemente.

Der Prozess der Generierung wird nicht nur visualisiert, vielmehr wird er, vergleichbar zu der interaktiven Visualisierung des Parsingprozesses, interaktiv gestaltet. Das bedeutet, dass der Fokus, wie auch in der zweiten Kategorie, auf dem Prozess der Generierung liegt, aber durch die interaktiven Elemente ein größerer Wert auf das Erlernen der Verfahren für die Generierung gelegt wird.

Beispiele der letzten beiden Kategorien und auch die Kombinationen der zwei Kategorien sind zwar durchaus vorhanden und mit dem erstellten Programm gleichermaßen vergleichbar, allerdings ergab eine eigene Recherche, dass Visualisierungen mit Fokus auf die Generierung und gleichzeitigen interaktiven Aspekten nicht weit verbreitet sind, demnach liefert der LR(1)-Parsergenerator in diesem Aspekt einen Mehrwert gegenüber anderen Parsergeneratoren in Kategorie zwei.

Da Lernprogramme in dieser Hinsicht selten sind, wurde auch keine verfügbare Literatur gefunden, welche die besonderen Anforderungen, die in diesem Fall benötigt werden, aufgreift

und aufbereitet. Weiterhin sind die designtechnischen Aspekte für e-Learning Systeme nur in Form allgemeiner Anforderungen vorhanden (siehe [12], [15], [20]).

Aus diesen Gründen wurde die Implementierung des Systems auf nicht-geprüfte Annahmen darüber, welche Aspekte hilfreich für das Lernen sind, basiert. Infolgedessen wurde im Rahmen dieser Bachelorarbeit eine Evaluation über die Effektivität des Systems durchgeführt. Dafür wurde in dem Programm eine Feedback-Funktionalität eingebaut, welches den Nutzern erlaubt, auf sechs vordefinierten Fragen zur empfundenen Effektivität des Lernprogramms zu antworten. Des Weiteren wurde ein freies Textfeld für weiteres Feedback zur Verfügung gestellt.

3.2 Welche Forschung gibt es für Parser?

Wie bereits in der Motivation (siehe Kapitel 2.1) erwähnt, ist der Compilerbau ein prominentes Beispiel für die Anwendung von Parsern in der Informatik. Es wird im Compilerbau versucht, das Problem des Erkennens von Quelltexten einer Programmiersprache durch formale Sprachen¹ und Grammatiken² zu lösen. Ein Verfahren, welches dieses Problem löst, ist das sogenannte LR(1)-Verfahren³. Das Verfahren ist das Ergebnis von umfangreicher Forschung im Gebiet der formalen Sprachen und Grammatiken. Diese Forschung hat ihren Ursprung in der Linguistik, jedoch wurden formalen Sprachen und Grammatiken erst nach der Einführung in die Mathematik und Informatik formal definiert. Erst daraufhin fanden diese auch ihre Anwendung in der Informatik. (vgl. [24])

Ziel dieses Abschnittes ist es, die für die Entwicklung des LR(1)-Verfahrens wichtigsten Forschungsergebnisse chronologisch zu präsentieren.

In 1956 wurde von Noam Chomsky eine Klassifizierung von verschiedenen formalen Grammatiken in der sogenannten Chomsky-Hierarchie eingeführt. Diese Hierarchie klassifiziert Grammatiken nach ihrer Mächtigkeit⁴ in drei Sprachklassen⁵ *Typ-1-* (auch kontextsensitive Grammatiken genannt), *Typ-2-* (auch kontextfreie Grammatiken genannt) und *Typ-3-Grammatiken* (auch reguläre Grammatiken genannt), wobei die *Typ-1-Grammatiken* die mächtigsten Sprachen bilden und *Typ-3-Grammatiken* formal am schwächsten sind. Aufgrund des starken Zusammenhangs⁶ zwischen formalen Grammatiken und Sprachen, lässt sich diese Hierarchie auch auf formale Sprachen übertragen. (vgl. [5])

Im weiteren Verlauf werden nur die *Typ-2-Grammatiken* (beziehungsweise die *Typ-2-Sprachen*) betrachtet, da diese Kategorie in direkter Weise mit dem Problem des Erkennens von Quelltexten zusammenhängt.

Der Zusammenhang ist über die Unterklasse *deterministisch kontextfreien Sprachen* der

¹Die formalen Sprachen werden in Kapitel 4.2.1.1 definiert.

²Die formalen Grammatiken werden in Kapitel 4.1.2.1 definiert.

³Das LR(1)-Verfahren wird in Kapitel 4.4 definiert.

⁴Die Mächtigkeit meint hier die Größe der Sprachklasse.

⁵Sprachklassen sind Mengen von Sprachen, welche meist gemeinsame Eigenschaften haben.

⁶Dieser Zusammenhang wird in Kapitel 4.2.1.2 weiter erleutert.

kontextfreien Sprachen⁷ gegeben. Diese wurden 1965 in *Deterministic context free languages* von Ginsburg und Greibach, durch Einschränkungen an die kontextfreien Sprachen, als echte Teilmenge der kontextfreien Sprachen eingeführt. (vgl. [8])

Um herauszufinden, ob eine gegebene Eingabe zu einer Sprache gehört, existieren grundsätzlich zwei Vorgehensweisen. Es kann versucht werden, durch die gegebene Grammatik die Eingabe zu konstruieren oder ausgehend von der Eingabe, Regeln der Grammatik zu finden, welche mit der Eingabe übereinstimmen. Die Verfahren werden respektive Top-Down- und Bottom-Up-Verfahren genannt.

Ein Bottom-Up-Verfahren, für die Erkennung von deterministisch kontextfreien Sprachen, wurde nach der Veröffentlichung von *Deterministic context free languages* im selben Jahr 1965 von Donald E. Knuth hergeleitet. Dieses Verfahren ist als das LR-Verfahren bekannt und dient heute als Basis für die gängigsten Parser im Compilerbau (vgl. [14])

Auch im Rest des 20. Jahrhundert wurde weiter Forschung betrieben, um Verfahren für die Erkennung von verschiedenen Sprachklassen zu finden. So wurde zum Beispiel, vor allem nachdem das LR-Verfahren weitestgehend als Standard etabliert worden war, an der Herangehensweise geforscht, um diese zu simplifizieren und effizienter zu gestalten.

In diesem Rahmen wurden 1969 von Frank L. DeRemer erstmals zwei weitere Bottom-Up verfahren entwickelt. Die sogenannten SLR- und LALR-Verfahren basieren auf das LR-Verfahren und sind zwei der praktischen Anwendungsfälle des LR-Verfahrens (vgl. [6]). Dabei konnten diese Verfahren, durch Vereinfachungen der Konzepte, effizientere Parser generieren, welche allerdings nur eine Teilmenge aller deterministisch kontextfreien Sprachen erkennen können. (vgl. [6], S. 70)

Selbst heutzutage werden LALR-Parser⁸ für den Compilerbau genutzt, denn sie sind effizienter als LR-Parser⁹, erkennen aber noch einen Großteil der deterministisch kontextfreien Sprachen. (vgl. [6] S. 70, 86-88)

Aufgrund des simpleren Verfahrens können durch das SLR-Verfahren weniger Sprachen erkannt werden als durch das LALR-Verfahren (vgl. [6] S. 91-93). Dafür werden andererseits, wie bereits in Kapitel 2.2 erwähnt, diese Verfahren oft in der Lehre für die Einführung des LR-Verfahrens genutzt.

Damit ist die Präsentation der relevanten Forschungsergebnisse in der Anwendung im Compilerbau abgeschlossen. Um noch einen Überblick über andere Anwendungsgebiete zu erhalten, sind hier noch einige der jüngsten Entwicklungen im Bereich der formalen Sprachen aufgelistet. Diese neuere Forschung entsteht oft in Anbindung der formalen Sprachen an andere Forschungsfelder.

Ein prominentes Beispiel der neueren Forschung ist die Anwendung im Gebiet des Natural Language Processings (kurz NLP), welches sich mit der Verarbeitung und vor allem

⁷Die formalen Sprachen werden in Kapitel 4.2.1.1 definiert.

⁸LALR-Parser sind Parser, welche das LALR-Verfahren nutzen.

⁹LR-Parser sind Parser, welche das LR-Verfahren nutzen.

der Erkennung natürlicher Sprachen befasst. (vgl. [4]) Dies kann von Linguisten in der Analyse von Sprachen verwendet werden, aber auch von Biologen und Zoologen, um die Kommunikation zwischen Tieren zu betrachten. (vgl. [7])

Ein anderes Beispiel für die Verarbeitung von Sprachen ist die Paraphrasierung von Texten mithilfe von Neuronalen Netzen. Angewendet auf einer exakt spezifizierten, formalen Sprache, wie den Quellcodes von Programmen, hat die Theorie ein großes Anwendungspotential. Dabei kann die Sprachverarbeitung mithilfe von formalen Sprachen genutzt werden, um Quellcodes durch Refaktorisierung zu vereinfachen oder sogar effizienter zu gestalten. (vgl. [9])

Weitere Forschungsfelder in denen formale Sprachen Anwendung finden, ist zum Beispiel die sogenannte „symbolic regression“. In der „symbolic regression“ wird automatisiert eine mathematische Modellierung von beobachteten Daten erstellt. Hier kann die Angabe formaler Sprachbeschreibungen die Suchmenge verringern, um so die Suche effizienter zu gestalten. (vgl. [2])

4 Theorie der Formalen Sprachen

Für die formale Analyse der entwickelten Visualisierung muss erst eine grundlegende Begriffserklärung erfolgen. Im Folgenden findet genau diese Begriffserklärung in Form eines Grundlagen-Kapitels statt. Dabei werden die entsprechenden Strukturen definiert und wichtige Eigenschaften benannt, welche bei der Analyse aufgegriffen werden.

Die Einführung folgt einer linearen Struktur, in der die Definitionen in direkter Form aufeinander aufbauen. Dafür werden nur die notwendigen Definitionen für die Nutzung in der späteren Analyse vorgestellt. Diese werden zunächst jeweils in einem kurzen Abschnitt separat motiviert, um diese anschließend auch hinsichtlich deren Definition formal einzuführen, auf Basis dessen die daran angeknüpfte Analyse stattfindet.

Beispiele werden gegeben, um den jeweiligen Nutzen der Definitionen zu verdeutlichen und werden so gewählt, dass anhand der Beispiele die nächste Definition motiviert wird.

4.1 Kontextfreie Grammatik

Formale Grammatiken werden für die Erzeugung formaler Sprachen genutzt und bilden damit eine Grundlage für die Analyse von Sprachen. Viele der Eigenschaften von formalen Sprachen können aus gleichwertigen Eigenschaften der Grammatik hergeleitet werden. So kann beispielsweise die Grammatik dazu genutzt werden, um festzulegen, ob ein gegebenes Wort zu der entsprechenden Sprache gehört.

4.1.1 Alphabet & Wörter

Für die Einführung der kontextfreien Sprachen wird erst einmal die Definition einer kontextfreien Grammatik benötigt, welche selbst die Definition eines Alphabets und den Wörtern eines Alphabets nutzt.

4.1.1.1 Definition

Ein **Alphabet** Σ ist eine beliebige *endliche* ($|\Sigma| < \infty$) Menge.

Die Elemente $s \in \Sigma$ eines Alphabetes werden **Symbole** (seltener auch **Zeichen** oder **Buchstaben**) genannt.

Ein **Wort über** Σ (kurz geschrieben **Wort**) w ist eine *endliche* Folge von Symbolen aus dem Alphabet Σ . Die leere Folge von Symbolen ϵ wird als das **leere Wort** bezeichnet. Das Ende eines Wortes wird von dem **Endsymbol** $\$$ angegeben.

(vgl. [10] S. 7-8)

Alphabete bilden als Grundmenge der Wörter die Grundlage für alle weiteren Definitionen, sind aber aufgrund der schwachen Einschränkung als *endliche* Menge nicht sonderlich Interessant. Einige bekannte Alphabete und Beispiele von Wörtern, die daraus gebildet werden können, sind:

1. $\Sigma_{DE} = \{a, b, c, \dots, z, \ddot{a}, \ddot{o}, \ddot{u}, \beta\}$: Das deutsche Alphabet mit den deutschen Wörtern.
2. $\Sigma_{bin} = \{0, 1\}$: Das binäre Alphabet mit den Binärzahlen als Worte.
3. $\Sigma_{(,)} = \{ (,) \}$: Das Alphabet der Klammern mit Klammerpaaren als Worte.

Dabei ist zu beachten, dass die Alphabete Σ_{bin} und $\Sigma_{(,)}$ formal nicht unterscheidbar sind und somit aufgrund der einzigen Unterscheidung durch die Symbole als gleichwertig gesehen werden.

Des Weiteren bilden die angegebenen Mengen der Wörter nicht das gesamte Alphabets, sondern lediglich eine Teilmenge aller möglichen Wörter. Beispielsweise ist auch „aaa“ ein Wort des deutschen Alphabets Σ_{DE} .

Die Menge aller *nichtleeren* Wörter, die aus Symbolen des Alphabets Σ gebildet werden können, wird durch Σ^+ bezeichnet. Für die Menge aller (d.h. *leerer* und *nichtleerer*) Wörter wird die Schreibweise Σ^* verwendet.

4.1.1.2 Bemerkung

Da das Aneinanderreihen von Wörtern die meistbenutzte Operation auf Wörtern ist, wird diese als die **Konkatenation** von Wörtern definiert.

Sei Σ ein beliebiges Alphabet.

Die Konkatenation von zwei Wörtern $w \in \Sigma$ und $v \in \Sigma$ ist eine Abbildung:

$$\circ : \Sigma \times \Sigma \rightarrow \Sigma, \quad w \circ v = wv$$

Wenn offensichtlich ist, dass die Konkatenation $w \circ v$ genutzt wird, wird stattdessen auch wv geschrieben.

(vgl. [3] S. 21)

Dabei ist zu beachten, dass die Konkatenation von einem Wort mit dem leeren Wort ϵ die Identität bildet. Es gilt also:

$$\epsilon w = w = w \epsilon$$

Die n -fache Ausführung der Konkatenation eines Wortes w mit sich selbst wird der Einfachheit halber mit w^n bezeichnet.

Mithilfe der Konkatenation können die Mengen Σ^* und Σ^+ durch Induktion formal definiert werden.

4.1.1.3 Definition

Sei Σ ein beliebiges Alphabet.

Die **kleenesche Hülle** Σ^* von Σ ist eine Menge, die folgende Eigenschaften erfüllt:

1. $w \in \Sigma, v \in \Sigma^* \implies w \circ v \in \Sigma^*$
2. $\epsilon \in \Sigma^*$

Die **positive Hülle** Σ^+ kann damit definiert werden als $\Sigma^+ := \Sigma^* \setminus \{\epsilon\}$

(vgl. [10] S. 9)

Dabei besagt die Erste der Eigenschaften, dass Wörter beliebig durch das Aneinanderreihen von Symbolen gebildet werden können. Die zweite Eigenschaft bildet den Induktionsanfang der Definition, denn wie in Bemerkung 4.1.1.2 gesehen, bildet die Konkatenation mit ϵ die Identität.

Diese Erklärung zeigt gleichzeitig auch, dass diese Definition mit der anschaulichen Bedeutung als „Menge aller Wörter die aus Σ gebildet werden können“ übereinstimmt.

4.1.2 Grammatik

Durch die Nutzung der Definitionen von Alphabeten und deren kleenesche und positive Hülle können formale Grammatiken und kontextfreie Grammatiken als Einschränkung der formalen Grammatiken eingeführt werden.

4.1.2.1 Definition

Eine **formale Grammatik** $\mathcal{G} = (\mathcal{T}, \mathcal{N}, \mathcal{P}, \mathcal{S})$ (auch **Grammatik**) ist ein Tupel bestehend aus:

1. Einem Alphabet $\mathcal{T}$ von **Terminalsymbolen** (auch **Tokens** genannt)

2. Einem Alphabet $\mathcal{N}$ von **Nichtterminalsymbolen** (auch **Nonterminalsymbole** oder **Variablen** genannt) mit der Eigenschaft $\mathcal{T} \cap \mathcal{N} = \emptyset$
3. Einer Menge $\mathcal{P} \subseteq (\mathcal{T} \cup \mathcal{N})^* \times (\mathcal{T} \cup \mathcal{N})^*$ von Tupel namens **Produktionsregeln** (auch **Produktionen** oder **Ableitungsregeln** genannt)
4. Einem Startsymbol $\mathcal{S} \in \mathcal{N}$

(vgl. [10] S. 109, [3] S. 180)

Die Bedeutung der Tupelelemente lassen sich wie folgt erklären. Ausgehend von dem Startsymbol $\mathcal{S}$ werden Produktionsregeln angewendet, welche ein Teilwort aus Terminal- und Nichtterminalsymbolen durch das zweite und eventuell leere Tupelelement ersetzen. Dabei muss das zu ersetzende Teilwort genau dem ersten Wort der Produktionsregel entsprechen.

Die Untermenge der kontextfreien Grammatiken werden durch eine Einschränkung der Produktionsregeln wie folgt definiert.

4.1.2.2 Definition

Eine formale Grammatik $\mathcal{G} = (\mathcal{T}, \mathcal{N}, \mathcal{P}, \mathcal{S})$ heißt **kontextfreie Grammatik**, wenn die Produktionsregeln

$$\mathcal{P} \subseteq \mathcal{N} \times (\mathcal{T} \cup \mathcal{N})^*$$

auf Ersetzungen von einzelnen Nichtterminalsymbolen eingeschränkt werden.

(vgl. [3] S. 180)

Wichtig hierbei ist, dass das erste Element einer Produktionsregel immer aus genau einem Nichtterminalsymbol besteht. Der Begriff der Kontextabhängigkeit wird durch genau diese Eigenschaft erklärt. Jedes beliebige Vorkommen eines Nichtterminalsymbols kann durch eine entsprechende Produktionsregel ersetzt werden, ungeachtet der Symbole, die vor oder nach dem Nichtterminalsymbol stehen.

4.1.2.3 Bemerkung

Um die Richtung der Ersetzung zu verdeutlichen, werden Elemente der Produktionsregeln in folgender Form angegeben:

$$N \rightarrow w \quad \text{mit } N \in \mathcal{N} \text{ und } w \in (\mathcal{T} \cup \mathcal{N})^*$$

Aus dieser Schreibweise hervorgehend, wird das erste Element der Produktionsregel (hier N) **linke Seite** der Produktionsregel genannt und entsprechend das zweite Element (hier w) **rechte Seite**.

(vgl. [3] S. 178)

4.1.3 Ableitungen & Reduktionen

Die in 4.1.2.3 genannten Ersetzungen sind für die Definition der kontextfreien Sprachen wichtig und können folgendermaßen formal definiert werden.

4.1.3.1 Definition

Sei $\mathcal{G} = (\mathcal{T}, \mathcal{N}, \mathcal{P}, \mathcal{S})$ eine Grammatik.

Ein **Ableitungsschritt** $w \xrightarrow[\mathcal{G}]{} v$ ist eine Relation auf $(\mathcal{T} \cup \mathcal{N})^*$. Dabei stehen zwei Wörter $w = w'Nw'' \in (\mathcal{T} \cup \mathcal{N})^*$ und $v = w'xw'' \in (\mathcal{T} \cup \mathcal{N})^*$ genau dann in Relation, wenn eine Produktionsregel $N \rightarrow x$ existiert.

Eine **Ableitung (von w nach v)** $w \xrightarrow[\mathcal{G}]{}^* v$ ist eine Folge von Wörtern

$$w = w_1, w_2, \dots, w_n = v \quad \text{mit } w_i \in (\mathcal{T} \cup \mathcal{N})^*, \forall i \in \{1, \dots, n\}$$

mit der Eigenschaft:

$$w_i \xrightarrow[\mathcal{G}]{} w_{i+1} \quad \forall i \in \{1, \dots, n-1\}$$

(vgl. [3] S. 182)

Wenn zwischen zwei aufeinanderfolgenden Folgengliedern ein Ableitungsschritt existiert, ist w aus v **ableitbar (über $\mathcal{G}$)**. Außerdem heißen Ableitungen von w nach v **vollständige Ableitung (von w)**, falls das Wort $v \in \mathcal{T}^*$ vollständig aus Terminalsymbolen besteht.

(vgl. [3] S. 182)

Aufgrund der geringen Beschränkungen für die rechte Seite der Produktionsregel einer Grammatik können Ableitungen auch mehrdeutig sein. Sei beispielsweise

$$\mathcal{G} = (\{a, b, c\}, \{S, X, Y\}, \{S \rightarrow XY, X \rightarrow a, X \rightarrow ab, Y \rightarrow bc, Y \rightarrow c\}, S)$$

Die Ableitung $S \xrightarrow[\mathcal{G}]{}^* abc$ kann nach der obigen Definition aus den Ableitungsschritten

$$S \xrightarrow[\mathcal{G}]{} XY \xrightarrow[\mathcal{G}]{} abY \xrightarrow[\mathcal{G}]{} abc$$

bestehen, aber auch aus:

$$S \xrightarrow[\mathcal{G}]{} XY \xrightarrow[\mathcal{G}]{} Xbc \xrightarrow[\mathcal{G}]{} abc$$

Diese Mehrdeutigkeit motiviert die Definition der **Linksableitung** (oder der **Rechtsableitung**), in der immer das am weitesten links (oder rechts) stehende Nichtterminalsymbol abgeleitet wird.

(vgl. [19] S. 138)

Die Ableitungen von beliebigen Wörtern v nach w mit $w \in \mathcal{T}^*$ geben gewissermaßen das Ziel der Theorie der formalen Sprachen und Grammatiken an. Die grundlegenden Entscheidungsprobleme zu Sprachen und ihren Grammatiken sind genau die zwei Fragen:

1. Was ist die gesamte Menge der über $\mathcal{G}$ ableitbaren Wörter?
2. Ist ein gegebenes Wort ableitbar über $\mathcal{G}$?

Gleichzeitig verdeutlichen diese zwei Fragen die zwei grundlegenden Nutzungsarten von Grammatiken. Die letztere Form der Nutzung ist auch besser als „Parsen“ bekannt. Bei dem Parsen wird genau versucht herauszufinden, ob eine gegebene Folge von Symbolen durch Ableitungen erreicht werden kann.

Im Folgenden wird das zweite Entscheidungsproblem mit **Wortproblem** bezeichnet.

Auffällig ist hierbei, dass für die Untersuchung des Parsing-Problems Ableitungen nur bedingt geeignet sind. Intuitiv wirkt das zugrunde liegende Prinzip so als sei es falsch herum angewendet.

Bei dem sogenannten Bottom-Up Parsing wird nicht aus dem Startsymbol ein Wort abgeleitet, sondern es wird versucht, das Wort auf das Startsymbol zu **reduzieren**. Hierbei ist anzumerken, dass Ableitungen tatsächlich auch für das Parsen genutzt werden können, diese Verfahren werden entsprechend Top-Down Parsing genannt, werden jedoch nicht in dieser Arbeit behandelt.

Die Umkehrung der Ableitungen (**Reduktionen** genannt) sind analog zu Ableitungen definiert, werden jedoch im Laufe der Arbeit benötigt und daher der Vollständigkeit halber formal eingeführt.

4.1.3.2 Definition

Sei $\mathcal{G} = (\mathcal{T}, \mathcal{N}, \mathcal{P}, \mathcal{S})$ eine Grammatik.

Ein **Reduktionsschritt** $w \xleftarrow[\mathcal{G}]{} v$ ist eine Relation auf $(\mathcal{T} \cup \mathcal{N})^*$. Dabei stehen zwei Wörter $w = w'Nw'' \in (\mathcal{T} \cup \mathcal{N})^*$ und $v = w'xw'' \in (\mathcal{T} \cup \mathcal{N})^*$ genau dann in Relation, wenn eine Produktionsregel $N \rightarrow x$ existiert.

Eine **Reduktion (von v nach w)** $w \xleftarrow[\mathcal{G}]{}^* v$ ist eine Folge von Wörtern

$$w = w_1, w_2, \dots, w_n = v \quad \text{mit} \quad w_i \in (\mathcal{T} \cup \mathcal{N})^*, \forall i \in \{1, \dots, n\}$$

mit der Eigenschaft,

$$w_i \xleftarrow[\mathcal{G}]{} w_{i+1} \quad \forall i \in \{1, \dots, n-1\}$$

dass zwischen zwei aufeinanderfolgenden Folgenglieder ein Reduktionsschritt existiert. w ist dann aus v **reduzierbar (über $\mathcal{G}$)**.

(vgl. [10] S. 129)

Im Beispiel der Ableitungen wären die entsprechenden Reduktionsschritte der Reduktion $S \xrightarrow[\mathcal{G}]{*} abc$ die Folgenden:

$$S \xrightarrow[\mathcal{G}]{} XY \xrightarrow[\mathcal{G}]{} abY \xrightarrow[\mathcal{G}]{} abc$$

und:

$$S \xrightarrow[\mathcal{G}]{} XY \xrightarrow[\mathcal{G}]{} Xbc \xrightarrow[\mathcal{G}]{} abc$$

Mit der selben Begründung der Definitionen von Linksableitungen und Rechtsableitungen werden **Linksreduktionen** und **Rechtsreduktionen** analog definiert.

4.1.4 First- & Follow-Mengen

Bevor die Formalien der kontextfreien Sprachen hergeleitet werden, werden erst zwei essenzielle Mengen definiert, welche auf der Theorie der Grammatiken aufbauen. Diese werden jedoch erst später im Rahmen des LR(1)-Verfahrens benötigt.

4.1.4.1 Definition

Sei $\mathcal{G} = (\mathcal{T}, \mathcal{N}, \mathcal{P}, \mathcal{S})$ eine Grammatik

Die **First-Menge eines Nichtterminalsymbols** $N \in \mathcal{N}$ entspricht der Menge:

$$\text{First}(N) := \{a \in \mathcal{T} \cup \{\epsilon\} \mid \exists N \xrightarrow[\mathcal{G}]{*} aw, w \in (\mathcal{T} \cup \mathcal{N})^*\}$$

Für die **First-Menge eines Terminalsymbols (oder ϵ)** $a \in \mathcal{T} \cup \{\epsilon\}$ gilt:

$$\text{First}(a) := \{a\}$$

(vgl. [19] S. 163)

In Worten ausgedrückt, bildet die First-Menge eines Symbols $a \in (\mathcal{T} \cup \mathcal{N})^*$ ein Alphabet von Terminalsymbolen (und ϵ), welches alle Symbole enthält, die in der vollständigen Ableitung von a an erster Stelle stehen können.

4.1.4.2 Bemerkung

Zu beachten ist, dass auch die First-Mengen eines Wortes $w \in (\mathcal{T} \cup \mathcal{N})^*$ definierbar sind. Dafür wird die First-Menge eines Wortes $w = av$ rekursiv auf die Definition der First-Mengen von Symbolen a zurückgeführt.

$$\text{First}(av) := \begin{cases} \text{First}(a), & \epsilon \notin \text{First}(a) \\ \text{First}(a) \setminus \{\epsilon\} \cup \text{First}(v) & \epsilon \in \text{First}(a) \end{cases}$$

Auf diese Definition baut eine weitere Definition auf, welche der erleichterten Handhabung dient. Die First-Mengen einer Wörmmenge $M = \{w_1, \dots, w_n\}$ wird definiert als:

$$\text{First}(M) := \bigcup_{i=1}^n \text{First}(w_i)$$

Vorbereitend auf das LR(1)-Verfahren wird außerdem für ein Wort w

$$\text{First}(wM) := \bigcup_{i=1}^n \text{First}(ww_i)$$

definiert. Es wird also das Wort w mit jeweils allen Wörtern aus M konkateniert und davon die First-Mengen vereinigt.

Die zweite Menge wird mit **Follow-Menge von dem Nichtterminalsymbol** N bezeichnet und ist ein fundamentaler Bestandteil des SLR(1)-Verfahrens (vgl. [6]), wird jedoch im LR(1)-Verfahren nicht benötigt. Nichtsdestotrotz liefern Follow-Mengen nützliche Informationen über die Grammatik, welche in der Visualisierung genutzt werden. Die Mengen werden im Verlauf der Arbeit aufgegriffen, um die entsprechenden Aspekte der Visualisierung zu erläutern und werden daher formal definiert.

4.1.4.3 Definition

Sei $\mathcal{G} = (\mathcal{T}, \mathcal{N}, \mathcal{P}, \mathcal{S})$ eine Grammatik

Die **Follow-Menge** (eines Nichtterminalsymbols $N \in \mathcal{N}$) entspricht der Menge:

$$\text{Follow}(N) := \begin{cases} \text{First}(v), & \exists S \xrightarrow[\mathcal{G}]{*} wNv \vee \epsilon \notin \text{First}(v) \\ (\text{First}(v) \setminus \{\epsilon\}) \cap \{\$, \}, & \exists S \xrightarrow[\mathcal{G}]{*} wNv \vee \epsilon \in \text{First}(v) \end{cases}$$

(vgl. [19] S. 166)

Die Follow-Menge gibt eine Menge von Terminalsymbolen an, welche innerhalb einer vollständigen Ableitung von $\mathcal{S}$ direkt auf die vollständige Ableitung von N folgen. In der Definition wird hierbei auf die rekursive Definition der First-Mengen für Worte zurückgegriffen.

Das Ende des Wortes „\$“ ist genau dann in der Follow-Menge enthalten, wenn die vollständige Ableitung von $\mathcal{S}$ mit der vollständigen Ableitung von N enden kann.

Für die leichtere Handhabung im weiteren Verlauf der Arbeit werden hier einige wichtige Eigenschaften der Grammatiken benannt.

4.1.4.4 Bemerkung

Sei $\mathcal{G} = (\mathcal{T}, \mathcal{N}, \mathcal{P}, \mathcal{S})$ eine Grammatik, dann gelten folgende Eigenschaften:

1. Ein Nichtterminalsymbol $N \in \mathcal{N}$ wird **erreichbar** genannt, wenn $S \xRightarrow[\mathcal{G}]{*} wNw'$ für $w, w' \in (\mathcal{T} \cup \mathcal{N})^*$ existiert. Bei einem Fehlen dieser Eigenschaft wird N als **unerreichbar** bezeichnet.

(vgl. [19] S. 165)

2. Produktionsregeln der Form $N \rightarrow Nw$ mit $N \in \mathcal{N}, w \in (\mathcal{T} \cup \mathcal{N})^*$ heißen **links-rekursiv**. Analog kann auch die Eigenschaft **rechts-rekursiv** definiert werden.

(vgl. [19] S. 152)

3. Die Grammatik $\mathcal{G}$ heißt **eindeutig**, wenn es für alle Ableitungen der Grammatik genau eine Linksableitung gibt. Beispielsweise ist dies insbesondere dann der Fall, wenn jedes Nichtterminalsymbol nur einmal in den rechten Seiten der Produktionsregeln vorkommen. Gilt diese Eigenschaft nicht, wird die Grammatik als **mehrdeutig** bezeichnet.

(vgl. [19] S. 150)

4.2 Kontextfreie Sprachen

Mithilfe der Definition der kontextfreien Grammatiken können kontextfreie Sprachen eingeführt werden. Diese sind als Einschränkungen des Konzepts der formalen Sprachen zu verstehen. Im Folgenden werden aus diesem Grund formale Sprachen zuerst definiert. Anschließend werden die kontextfreien Sprachen als eine besondere Art der formalen Sprachen hergeleitet.

4.2.1 Sprachen

4.2.1.1 Definition

Eine **formale Sprache über Σ** (auch **Sprache über Σ** genannt oder **formale Sprache** bzw. nur **Sprache**, wenn Σ aus dem Kontext bekannt ist) $\mathcal{L}$ ist eine beliebige Teilmenge von Worten über Σ .

$$\mathcal{L} \subseteq \Sigma^*$$

(vgl. [10] S. 16)

Insgesamt hat damit die reine Struktur einer formalen Sprache keine besonderen Einschränkungen und ist somit auch nicht sehr komplex. Die formale Sprache ist an sich genommen nur eine Menge von Symbolfolgen. Insbesondere sind folgende Mengen streng genommen Sprachen:

1. $\emptyset$: Die leere Sprache.
2. $\{\epsilon\}$: Die Sprache mit dem leeren Wort.
3. $\Sigma \subseteq \Sigma^*$: Das Alphabet selbst als Sprache.

Diese Sprachen haben jedoch keinen großen Nutzen, sodass das Konzept der Sprache erst im Zusammenhang der Grammatiken für weitere Betrachtung interessant wird. Dafür werden Sprachen über den Terminalsymbolen einer Grammatik betrachtet. Wenn zu einer Grammatik $\mathcal{G} = (\mathcal{T}, \mathcal{N}, \mathcal{P}, \mathcal{S})$ eine Sprache über $\mathcal{T}$ betrachtet wird, wird diese auch als **Sprache über $\mathcal{G}$** bezeichnet.

4.2.1.2 Definition

Sei $\mathcal{G} = (\mathcal{T}, \mathcal{N}, \mathcal{P}, \mathcal{S})$ eine Grammatik.

Die **Sprache (über $\mathcal{G}$)** $\mathcal{L}(\mathcal{G})$ entspricht der Menge

$$\mathcal{L}(\mathcal{G}) := \{w \in \mathcal{T}^* \mid S \xrightarrow[\mathcal{G}]{}^* w\}$$

aller von $\mathcal{G}$ akzeptierbarer Wörter.

(vgl. [3] S. 182)

Die Sprache $\mathcal{L}(\mathcal{G})$ wird auch **von $\mathcal{G}$ erzeugt** genannt.

In Worten beschrieben entspricht $\mathcal{L}(\mathcal{G})$ genau der Menge von Wörtern über $\mathcal{T}$, die aus $\mathcal{S}$ ableitbar über $\mathcal{G}$ sind. Anders ausgedrückt sind $\mathcal{L}(\mathcal{G})$ alle Wörter aus Terminalsymbolen, die mit der Grammatik in kanonischer Weise übereinstimmen.

Damit lässt sich erklären, warum die Sprache $\mathcal{L}(\mathcal{G})$ auch mit **von $\mathcal{G}$ erzeugt** bezeichnet wird. Alle Wörter in $\mathcal{L}(\mathcal{G})$ lassen sich durch Ableitungsschritte mit den Produktionsregeln der Grammatik $\mathcal{G}$ erzeugen.

Es ist leicht erkennbar, dass die von $\mathcal{G}$ erzeugte Sprache tatsächlich auch eine Sprache ist, denn offensichtlich gilt $\{w \in \mathcal{T}^* \mid S \xrightarrow[\mathcal{G}]{}^* w\} \subseteq \mathcal{T}^*$.

In dem Beispiel

$$\mathcal{G} = (\{a, b, c\}, \{S, X, Y\}, \{S \rightarrow XY, X \rightarrow a, X \rightarrow ab, Y \rightarrow bc, Y \rightarrow c\}, S)$$

ist die Sprache über $\mathcal{G}$ durch

$$\mathcal{L}(\mathcal{G}) = \{ac, abc, abbc\}$$

gegeben, da dies alle Wörter sind die gebildet werden können.

4.2.1.3 Definition

Eine **kontextfreie Sprache** ist eine Sprache über einer kontextfreien Grammatik.

(vgl. [3] S. 183)

Die kontextfreie Sprache hängt damit eng mit den kontextfreien Grammatiken zusammen und übernimmt viele der Eigenschaften der Grammatik. Beispielsweise ist die Sprache sichtbar frei von kontextabhängigen Wörtern, da die Definition nur Wörter erlaubt, welche sich kontextfrei parsen lassen.

4.2.1.4 Bemerkung

Sprachen $\mathcal{L}$, die sich auch als kontextfreie Sprache schreiben lassen, also Sprachen für die gilt

$$\exists \text{ kontextfreie Grammatik } \mathcal{G} : \quad \mathcal{L} = \mathcal{L}(\mathcal{G})$$

werden auch als **kontextfrei** bezeichnet.

(vgl. [3] S. 183)

Beispiele von kontextfreien Sprachen über Σ sind:

1. Die Sprache $\mathcal{L}_{(,)}$ der wohlgeformten Klammerausdrücke. Diese beinhaltet beispielsweise $((()))()$ und $((()))()$ aber nicht $((())$ und $)($.
2. Die Sprache $\mathcal{L}_{a^n b^n} = \{a^n b^n \mid n \in \mathbb{N}\}$ mit $a, b \in \Sigma$ der Wörter mit gleich vielen a's und b's.
3. Die Sprache $\mathcal{L}_{a^n b^n c^m} = \{wc^m \mid w \in \mathcal{L}_{a^n b^n}, m \in \mathbb{N}\}$ der Wörter aus 2. mit beliebig vielen c's am Ende.

Diese sind kontextfrei, da die Grammatiken

1. $\mathcal{G}_{(,)} = (\Sigma_{(,)}, \{\mathcal{S}\}, \{\mathcal{S} \rightarrow \mathcal{S}\mathcal{S}, \mathcal{S} \rightarrow (\mathcal{S}), \mathcal{S} \rightarrow \epsilon\}, \mathcal{S})$
2. $\mathcal{G}_{a^n b^n} = (\Sigma_{DE}, \{\mathcal{S}\}, \{\mathcal{S} \rightarrow a\mathcal{S}b, \mathcal{S} \rightarrow \epsilon\}, \mathcal{S})$
3. $\mathcal{G}_{a^n b^n c^m} = (\Sigma_{DE}, \{\mathcal{S}, C\}, \{\mathcal{S} \rightarrow a\mathcal{S}bC, \mathcal{S} \rightarrow \epsilon, C \rightarrow cC, C \rightarrow \epsilon\}, \mathcal{S})$

existieren mit $\mathcal{L}_{(,)} = \mathcal{L}(\mathcal{G}_{(,)}), \mathcal{L}_{a^n b^n} = \mathcal{L}(\mathcal{G}_{a^n b^n})$ und $\mathcal{L}_{a^n b^n c^m} = \mathcal{L}(\mathcal{G}_{a^n b^n c^m})$

Analog zu 3. kann auch $\mathcal{L}_{a^m b^n c^n}$ mit einer entsprechenden kontextfreien Grammatik $\mathcal{G}_{a^m b^n c^n}$ betrachtet werden.

Leicht erkennbar ist, dass die Grammatiken $\mathcal{G}_{a^n b^n}$, $\mathcal{G}_{a^n b^n c^m}$ und $\mathcal{G}_{a^m b^n c^n}$ eindeutig sind. Das liegt daran, dass jedes Nichtterminal in diesen Grammatiken nur ein Mal in den rechten Seiten der Produktionsregeln vorkommen.

Dies gilt jedoch nicht für die Vereinigung der Sprachen $\mathcal{L}_{a^n b^n c^m}$ oder $\mathcal{L}_{a^m b^n c^n}$. Die Vereinigung

$$\mathcal{L}_{a^l b^m c^n} := \mathcal{L}_{a^n b^n c^m} \cup \mathcal{L}_{a^m b^n c^n} = \{a^l b^m c^n \mid l = m \wedge m = n, l, m, n \in \mathbb{N}\}$$

ist nicht mehr eindeutig. Da sowohl die Anzahl von a's und b's als auch die Anzahl der b's und c's gleich sein kann, müssen Produktionsregeln für beide Fälle enthalten sein. Der Fall $l = m = n$ reicht dabei schon, um die Eindeutigkeit zu widerlegen, da hier durch die verschiedenen Produktionsregeln Ableitungsschritte über verschiedene Produktionsregeln und damit mehrere Linksableitungen gegeben sind.

Tatsächlich ist hier in einem gewissen Sinne noch eine größere Mehrdeutigkeit gegeben, denn es ist nicht nur die Frage nach dem nächsten abzuleitenden Nichtterminalsymbol vorhanden, sondern auch die Frage, welche der Produktionsregeln angewendet werden muss.

Sogar die Betrachtung der umliegenden Symbole kann diese Mehrdeutigkeit nicht auflösen.

Die Definition der Sprachklasse von Sprachen welche diese Mehrdeutigkeit nicht aufweisen, wird mit dieser Motivation im folgenden Kapitel eingeführt und erläutert.

4.3 Deterministische endliche Automaten

Für den formalen Umgang mit Maschinen in der theoretischen Informatik werden Maschinen durch sogenannte Automaten modelliert. Dabei gibt es viele verschiedene Arten von Automaten, welche verschiedene Arten von Maschinen modellieren.

Von **Kellerautomaten** werden beispielsweise „Erkennungsmaschinen“ modelliert, die zu formalen Sprachen bestimmen, ob ein Wort zu der Sprache gehört. Dies entspricht genau der Beantwortung des Wortproblems.

4.3.1 Kellerautomaten

Formal werden Kellerautomaten zwar in erster Linie unabhängig von formalen Sprachen definiert, um das Konzept zu verallgemeinern, jedoch werden diese im Zuge der Bachelorarbeit ausschließlich im Kontext der formalen Sprachen benötigt, genauer gesagt für die Unterscheidung der Sprachklasse der deterministisch kontextfreien Sprachen und der kontextfreien Sprachen.

4.3.1.1 Definition

Ein **Kellerautomat** $\mathcal{M} = (\mathcal{Z}, \Sigma, \Gamma, \delta, \gamma_0, q_0, \mathcal{F})$ (auch **pushdown automaton** (kurz **pda**) genannt) ist ein Tupel bestehend aus:

1. Einer Menge $\mathcal{Z} \neq \emptyset$ von **Zuständen** mit der Eigenschaft $|\mathcal{Z}| < \infty$.

2. Einem Alphabet Σ von **Eingabesymbolen** mit der Eigenschaft $|\Sigma| < \infty$.
3. Einem Alphabet Γ von **Kellersymbolen** mit der Eigenschaft $|\Gamma| < \infty$.
4. Einer Funktion $\delta : \mathcal{Z} \times (\Sigma \cup \{\epsilon\}) \times \Gamma \rightarrow 2^{\mathcal{Z} \times \Gamma^*}$ (**Übergangsfunktion** genannt) mit der Eigenschaft $|\delta(q, w, \gamma)| < \infty \forall q \in \mathcal{Z}, w \in (\Sigma \cup \{\epsilon\}), \gamma \in \Gamma$.
5. Einem Element $\gamma_0 \in \Gamma$ (**Anfangssymbol** oder **Startsymbol** genannt).
6. Einem Element $q_0 \in \mathcal{Z}$ (**Anfangszustand** oder **Startzustand** genannt).
7. Einer Teilmenge der Zustände $\mathcal{F} \subseteq \mathcal{Z}$ von **Endzuständen**.

(vgl. [8] S. 622)

Der Begriff des Kellers entsteht aus der Nutzung eines Zwischenspeichers, welches nach dem FILO-Prinzip (first in last out) agiert, solche Speicher werden in der Informatik auch Keller (oder Stack) genannt. Das bedeutet, dass bei dem Nutzen des Speichers immer das zuletzt eingefügte Element ausgelesen wird. Dabei ist zu beachten, dass bei dem Auslesen aus einem Keller standardmäßig das ausgelesene Element entfernt wird.

In der Definition eines Kellerautomaten geben die Kellersymbole Γ an welche Symbole in dem Keller gespeichert werden können und das Anfangssymbol γ_0 gibt den Inhalt des Kellers zu Beginn an.

Zwar sind die Tupелеlemente individuell betrachtet nicht komplex, jedoch muss für die Interpretation als Maschine die Betrachtung des Tupels als Ganzes erfolgen.

Anschaulich gesehen führt ein Kellerautomat eine Verarbeitung eines Wortes $w \in \Sigma^*$ durch. Im Rahmen der formalen Sprachen entspricht diese Verarbeitung das Einlesen und Erkennen von Wörtern. Dabei bilden die Eingabesymbole Σ das Alphabet der Wörter, die erkannt werden sollen.

Wörter aus Σ^* werden durch die Übergabe an den Kellerautomaten $\mathcal{M}$ erkannt, indem anfangs, von dem Startzustand q_0 ausgehend, das erste Zeichen des Wortes und das Startsymbol γ_0 des Kellers eingelesen werden. Basierend auf der Übergangsfunktion δ , erfolgt dann ein sogenannter **Wechsel** zu einem anderen Zustand aus der Zustandsmenge $\mathcal{Z}$. Dabei gibt δ auch neue Kellersymbole an, welche in dem Keller zwischengespeichert werden. Iterativ wird dann das nächste Symbol der Eingabe und das letzte Symbol des Kellers eingelesen. Basierend auf den eingelesenen Symbolen und dem derzeitigen Zustand wird dann in den Zustand gewechselt, der von der Übergangsfunktion angegeben wird. Auch hier werden die angegebenen Kellersymbole in den Keller gelegt.

4.3.1.2 Bemerkung

Die Anwendung der Übergangsfunktion $\delta(q, w, \gamma) = (p, \gamma')$, wird im Folgenden **Zustandswechsel** (oder kurz **Wechsel**) **von q nach p über w mit Kellersymbolen γ'** genannt, wobei nicht benötigte Informationen weggelassen werden.

Anschaulich gesehen entspricht der Kellerautomat im Fall der formalen Sprachen in gewisser Weise der Grammatik der Sprache. Das Alphabet Σ ist dann genau das Alphabet der Terminalsymbole und die Verarbeitung entspricht dem Einlesen und Erkennen der Wörter. In dieser noch abstrakten Betrachtung bilden alle, von dem Kellerautomaten erkannten, Wörter die Sprache des Automaten.

Ein Beispiel eines Kellerautomaten ist:

$$\mathcal{M} = \{\{q_0, q_1, q_2\}, \{a, b\}, \{\alpha, \beta\}, \delta, \beta, q_0, \{q_2\}\}$$

mit:

$$\delta : \mathcal{Z} \times (\Sigma \cup \epsilon) \times \Gamma \rightarrow 2^{\mathcal{Z} \times \Gamma^*}, \quad \delta(q, w, \gamma) = \begin{cases} \{(q_1, \alpha), (q_2, \epsilon)\}, & q = q_0, w = a, \gamma = \beta \\ \{(q_0, \beta)\}, & q = q_1, w = b, \gamma = \alpha \end{cases}$$

Dieser Kellerautomat erkennt die Sprache $\mathcal{L} = \{a(ba)^n \mid n \in \mathbb{N}\}$ von Wörtern mit abwechselnden Buchstaben a und b , die mit a beginnen und enden.

Für eine anschaulichere Darstellungsweise werden Automaten oftmals als gerichtete Graphen visualisiert. Dabei bilden die Zustände die Ecken und die Zustandswechsel gerichtete Kanten von der Ecke, die dem Ursprungszustand des Wechsels entspricht, zu der Ecke, die dem Zielzustand des Wechsels entspricht. Die Eingabesymbole werden über der Kante vor einem Semikolon geschrieben und die neuen Kellersymbole hinter dem Semikolon.

Der Anfangszustand wird mit einer eingehenden Kante gekennzeichnet, die mit dem Anfangssymbol des Kellers gekennzeichnet ist und der Endzustand wird als zweifach umrandete Ecke gekennzeichnet.

Für das gegebene Beispiel sieht dementsprechend die Visualisierung wie folgt aus:

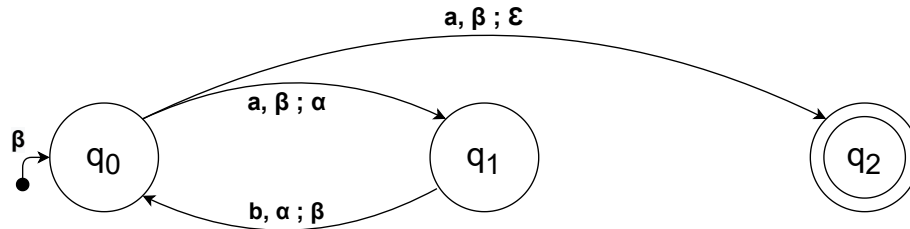


Abbildung 4.1: Beispiel für Kellerautomaten

Bei genauerer Betrachtung fällt auf, dass von dem Anfangszustand q_0 ausgehend, bei einer Eingabe des Wortes $a \in \{a, b\}^*$ der Zustandswechsel nicht eindeutig ist. Tatsächlich ist diese Mehrdeutigkeit immer gegeben, wenn im Zustand q_0 ein a als nächstes Eingabezeichen eingelesen wird.

Generelle Kellerautomaten können aufgrund fehlender Einschränkungen, eindeutige Wechsel nicht garantieren. Dies folgt aus der Möglichkeit, dass sowohl Wechsel über ϵ als auch Wechsel über andere *nichtleere* Wörter von dem selben Zustand aus definiert sein können. In diesem Fall kann zu jedem Wechsel über einem nichtleeren Symbol, stattdessen der Wechsel über ϵ durchgeführt werden. Eine reale Umsetzung des Automaten muss deshalb

alle mehrdeutigen Wechsel durchgehen, um ein Wort nicht fälschlicherweise als „der Sprache nicht zugehörend“ zu klassifizieren.

In dem gegebenen Beispiel muss für jedes eingelesene a im Zustand q_0 ein Schritt mehr durchgeführt werden, in der überprüft wird, ob das Wort mit dem nächsten a endet. In anderen Automaten kann diese Mehrdeutigkeit für noch größere Ineffizienzen sorgen.

Deterministische Kellerautomaten sind Automaten, die diese Mehrdeutigkeit nicht aufweisen und werden als Einschränkung der Kellerautomaten definiert.

4.3.1.3 Definition

Sei $\mathcal{M} = (\mathcal{Z}, \Sigma, \Gamma, \delta, \gamma_0, q_0, \mathcal{F})$ ein Kellerautomat

Der Kellerautomat $\mathcal{M}$ heißt **deterministisch**, wenn für alle $q \in \mathcal{Z}$ und $\gamma \in \Gamma$

$$\forall w \in \Sigma \cup \{\epsilon\} : |\delta(q, w, \gamma)| \leq 1$$

gilt und jeweils höchstens eine der folgenden Eigenschaften erfüllt ist:

1. $\exists w \in \Sigma \setminus \{\epsilon\} : |\delta(q, w, \gamma)| = 1$
2. $|\delta(q, \epsilon, \gamma)| = 1$

(vgl. [8] S. 623)

Die allgemeine Beschränkung der Übergangsfunktion auf höchstens ein Wechsel verhindert dabei die Mehrdeutigkeit in dem obigen Beispiel aus Abbildung 4.1. Für das Beispiel gilt:

$$\exists q_1 \in \{q_0, q_1, q_2\}, b \in \{a, b\} \cup \{\epsilon\}, \alpha \in \{\alpha, \beta\} : |\delta(q_1, b, \alpha)| = 2 > 1$$

Jedoch bleibt die andere beschriebene Mehrdeutigkeit aufgrund von Wechsel über ϵ trotz der allgemeinen Beschränkung möglich. Um diese Mehrdeutigkeit zu verhindern, wird die Beschränkung auf höchstens eine der genannten Eigenschaften benötigt. Diese haben zur Folge, dass in jedem Zustand und unabhängig von dem Kellerinhalt, Zustandswechsel nur existieren, wenn sie entweder nur über *nichtleere* Wörter erfolgen, oder nur über genau dem leeren Wort.

Wenn Wechsel über *nichtleere* Wörter existieren, kann es keinen Wechsel über ϵ geben. Dies würde Eigenschaft 2. erfüllen, obwohl bereits Eigenschaft 1. erfüllt ist. Genau so kann es umgekehrt keine anderen Wechsel geben, wenn es ein Wechsel über ϵ gibt, da dies Eigenschaft 1. erfüllen würde.

Folglich brauchen deterministischen Kellerautomaten für die Überprüfung, ob ein Wort zu einer Sprache gehört, im Allgemeinen weniger Berechnungsschritte als die allgemeine Definition von Kellerautomaten. Dies liegt daran, dass es zu jedem Eingabewort immer nur genau eine Folge von Zustandswechsel gibt. Diese kann dann in der Umsetzung linear

befolgt werden, um somit die Zugehörigkeit zur Sprache zu widerlegen oder zu bestätigen.

Allgemein kann jedoch auch für beliebige Kellerautomaten bestimmt werden, ob das Wort zu der jeweiligen Sprache gehört. Aus diesem Grund werden die folgenden Grundlagen der Verarbeitung allgemein definiert.

Während der Verarbeitung befindet sich der Automat bei jedem „Schritt“ in einem bestimmten Zustand $q \in \mathcal{Z}$, hat ein bestimmtes restliches Wort $w \in \Sigma$ und einen Kellerinhalt $\gamma \in \Gamma$. Da diese Angaben sich im Laufe einer Verarbeitung verändern, müssen diese für die eindeutige Beschreibung der „Lage“ zusammen mit der Definition des Kellerautomaten $\mathcal{M}$ angegeben werden. Diese „Lage“ wird im Folgenden als **Konfiguration** eines Automaten $\mathcal{M}$ definiert.

4.3.1.4 Definition

Sei $\mathcal{M} = (\mathcal{Z}, \Sigma, \Gamma, \delta, \gamma_0, q_0, \mathcal{F})$ ein Kellerautomat

Eine **Konfiguration (von $\mathcal{M}$)** $K = (q, w, \gamma) \in \mathcal{K}$ ist ein Tupel bestehend aus:

1. Dem derzeitigen Zustand $q \in \mathcal{Z}$
2. Dem restlichen Wort $w \in \Sigma^*$
3. Dem Inhalt des Kellers $\gamma \in \Gamma^*$

Die **Menge der (möglichen) Konfigurationen (von $\mathcal{M}$)** ist gegeben durch eine Teilmenge $\mathcal{K} \subseteq \mathcal{Z} \times \Sigma^* \times \Gamma^*$

(vgl. [8] S. 623)

Durch die Angabe der Konfiguration kann damit jeder Verarbeitungsschritt, in dem sich der Kellerautomat befinden kann, eindeutig beschrieben werden. Eine Konfiguration enthält genau die Elemente, welche die veränderlichen „Komponenten“ des Automaten bilden. Zusammen mit der Definition des Kellerautomaten wird damit ein eindeutiges Vokabular für die Handhabung der Kellerautomaten geschaffen.

Für das Beispiel in Abbildung 4.1 ist die Menge aller möglichen Konfigurationen $\mathcal{K}$, aufgrund von der simplen Struktur der Beispiels gegeben durch:

$$\mathcal{K} = M_0 \cup M_1 \cup M_2$$

mit:

$$M_0 = \{(q_0, w, \beta) : w \in a(ba)^*\}$$

$$M_1 = \{(q_1, w, \alpha) : w \in ba(ba)^*\}$$

$$M_2 = \{(q_2, w, \epsilon) : w \in (ab)^*\}$$

4.3.2 Übergänge

Des Weiteren dient die Definition der Konfiguration als Basis für die weitere Betrachtung von Eigenschaften der Automaten. So können beispielsweise die Wechsel der Übergangsfunktion weiter betrachtet werden.

Außerdem basiert die Definition des Analogons der Ableitung (genannt **Übergang**) im Kontext der Kellerautomaten auf die möglichen Konfigurationen und deren Übergänge.

4.3.2.1 Definition

Sei $\mathcal{M} = (\mathcal{Z}, \Sigma, \Gamma, \delta, \gamma_0, q_0, \mathcal{F})$ ein Kellerautomat

Ein **Übergangsschritt** $K \xRightarrow{\mathcal{M}} K'$ ist eine Relation auf der Menge der Konfigurationen $\mathcal{K}$. Dabei stehen zwei Konfigurationen $(q, xw, \alpha\gamma) \in \mathcal{K}$ und $(p, w, \alpha\gamma') \in \mathcal{K}$ mit $x \in \Sigma \cup \{\epsilon\}$ und $\gamma \in \Gamma$ genau dann in Relation, wenn $(p, \gamma') \in \delta(q, x, \gamma)$.

Ein **Übergang (von K nach K')** $K \xRightarrow{*}_{\mathcal{M}} K'$ ist eine Folge von Konfigurationen

$$K = (q, w_1, \gamma) = K_1, K_2, \dots, K_n = K' = (p, w_n, \gamma')$$

$$\text{mit } K_i = (q_i, w_i, \gamma_i) \in \mathcal{K} \quad \forall i \in \{1, \dots, n\}$$

mit der Eigenschaft,

$$K_i \xRightarrow{\mathcal{M}} K_{i+1} \quad \forall i \in \{1, \dots, n-1\}$$

dass zwischen zwei aufeinanderfolgenden Folgenglieder ein Übergangsschritt existiert.

(vgl. [8] S. 622-623)

Unter Beachtung der vorherigen Interpretation, dass Konfigurationen Schritte in der Eingabeverarbeitung eindeutig identifizieren, kann die Definition eines Übergangs als eine Reihe von Verarbeitungsschritten verstanden werden.

Ein einzelner Verarbeitungsschritt entspricht in dem Fall einem Übergangsschritt. Dieser existiert, wenn nach dem Einlesen des ersten Symbols x der Eingabe xw und des letzten Symbols γ des Kellerinhalts $\alpha\gamma$ ein Wechsel des Zustandes von q nach p existiert. Der hinzuzufügende Kellerinhalt γ' kann beliebig in Γ^* sein. Dies ist implizit durch die Definition der Übergangsfunktion gegeben und bedeutet, dass insbesondere auch $\gamma' = \epsilon$ gelten kann. Mit der selben Begründung ist $w \in \Sigma^*$ und kann damit auch das leere Wort sein.

Die Betrachtung von Übergängen als Reihe von Schritten der Eingabeverarbeitung liefert im Fall $w = \epsilon$ nicht nur Schritte der Eingabeverarbeitung, sondern die gesamte Verarbeitung des Wortes. Wenn also solcher Übergang existiert, bedeutet dies, dass das Wort vollständig verarbeitet werden kann. Der Zustand, in dem eine Verarbeitung endet, sagt dann aus, ob das Wort zu der Sprache gehört oder nicht.

Diese Eigenschaft wird oftmals benötigt und wird daher als eigenständiger Begriff formal definiert. Die **Akzeptanz** eines Wortes wird zwar genauso wie der Kellerautomat, unabhängig von den formalen Sprachen definiert, bedeutet aber im Fall der formalen Sprachen, dass das Wort in der jeweiligen Sprache enthalten ist.

Da die Übergänge bestimmen, welche Wörter von dem Automaten erkannt werden, werden diese in der Definition eines **akzeptierten** Wortes benötigt.

4.3.2.2 Definition

Sei $\mathcal{M} = (\mathcal{Z}, \Sigma, \Gamma, \delta, \gamma_0, q_0, \mathcal{F})$ ein Kellerautomat

Ein Wort w wird (**von dem Kellerautomaten $\mathcal{M}$**) **akzeptiert** wenn ein Übergang

$$K \xrightarrow[\mathcal{M}]{*} K' \text{ von } K = (q_0, w, \gamma_0) \in \mathcal{K} \text{ nach } K' = (q, \epsilon, \gamma) \in \mathcal{K}$$

existiert mit der Eigenschaft,

$$q \in \mathcal{F} \text{ und } \gamma \in \Gamma^*$$

dass q ein Endzustand ist und γ ein beliebiger Kellerinhalt.

(vgl. [8] S. 623)

Zu beachten ist hierbei, dass ein Halten des Automaten nicht garantiert ist. Beispielsweise kann eine mögliche Endlosschleife von ϵ -Übergängen entstehen, da dies der Definition eines Kellerautomaten nicht widerspricht. In diesem Fall wird das Wort definitionsgemäß nicht akzeptiert, da kein Endzustand erreicht wird.

Wenn der Automat hält, kann nach der vorherigen Überlegung der Übergang nach $K' = (q, \epsilon, \gamma)$ als vollständigen Verarbeitung des Eingabewortes aufgefasst werden. Dementsprechend ist der Zustand q aus der Konfiguration K' der letzte Zustand der Verarbeitung, im Falle einer endlichen Berechnung. Folglich wird dann dieser Zustand für die Bestimmung genutzt, ob das Wort akzeptiert wird.

Ein Wort wird genau dann von dem Kellerautomaten akzeptiert, wenn der letzte Zustand in der Menge der Endzustände $\mathcal{F}$ enthalten ist. Andersherum wird ein Wort nicht akzeptiert, wenn während des Einlesens ein Wechsel über dem nächsten Symbol fehlt, also kein Übergang nach $K' = (q, \epsilon, \gamma)$ existiert, oder wenn der letzte Zustand kein Endzustand ist.

Zu beachten ist, dass die Eindeutigkeit des letzten Zustandes nur für deterministische Kellerautomaten gegeben ist. Aus diesem Grund wird lediglich die Existenz eines Übergangs gefordert. Im Allgemeinen können Kellerautomaten zu einer gegebenen Eingabe mehrere Zustände haben, welche für die Bestimmung der Akzeptanz des Wortes genutzt werden können.

4.3.3 Deterministische Sprache

Mithilfe der Definition eines akzeptierten Wortes kann die gesamte Sprache, die von einem Kellerautomaten erkannt werden kann, definiert werden. Das sind genau die Sprachen, die bisher als „die Sprache des Kellerautomaten“ oder „die vom Kellerautomat erkannte Sprache“ bezeichnet wurden.

4.3.3.1 Definition

Sei $\mathcal{M} = (\mathcal{Z}, \Sigma, \Gamma, \delta, \gamma_0, q_0, \mathcal{F})$ ein Kellerautomat

Die **(von dem Kellerautomaten $\mathcal{M}$) akzeptierte Sprache $\mathcal{L}(\mathcal{M})$** entspricht der Menge

$$\mathcal{L}(\mathcal{M}) := \{w \in \Sigma^* \mid \exists q \in \mathcal{F}, \gamma \in \Gamma^* : (q_0, w, \gamma_0) \xrightarrow[\mathcal{M}]^* (q, \epsilon, \gamma)\}$$

aller von $\mathcal{M}$ akzeptierten Wörter.

(vgl. [8] S. 623)

Die Definition der Menge bedient sich hierbei der rein mathematischen Formulierung von akzeptierten Wörtern des Kellerautomaten. Die gesamte Menge dieser Wörter bildet die gewünschte Sprache. Auch hier ist klar, dass $\mathcal{L}(\mathcal{M})$ eine Sprache ist, da offensichtlich $\{w \in \Sigma^* \mid \exists q \in \mathcal{F}, \gamma \in \Gamma^* : (q_0, w, \gamma_0) \xrightarrow[\mathcal{M}]^* (q, \epsilon, \gamma)\} \subseteq \Sigma^*$ gilt.

Weiter ist folgende Bemerkung für das Verständnis des Zusammenhangs zwischen Kellerautomaten und kontextfreien Sprachen wichtig.

4.3.3.2 Bemerkung

Es gilt:

1. $\forall \mathcal{G}$ kontextfreie Grammatik $\exists \mathcal{M}$ Kellerautomat : $\mathcal{L}(\mathcal{G}) = \mathcal{L}(\mathcal{M})$
2. $\forall \mathcal{M}$ Kellerautomat $\exists \mathcal{G}$ kontextfreie Grammatik : $\mathcal{L}(\mathcal{M}) = \mathcal{L}(\mathcal{G})$

Beweise dieser Aussagen lassen sich in vielen Vorlesungen zu kontextfreien Sprachen finden, zum Beispiel im Studiengang theoretische Informatik an der Ludwigs-Maximilians-Universität in München [22].

Das bedeutet, dass Kellerautomaten genau die kontextfreien Sprachen erkennen können und jede kontextfreie Sprache von einem Kellerautomaten erkannt werden kann.

Für die Bemerkung kann jedoch nicht angenommen werden, dass die Kellerautomaten deterministisch sind. Deterministischen Kellerautomaten erkennen nur eine echte Teilmenge der kontextfreien Sprachen und definieren so die deterministisch kontextfreien Sprachen.

4.3.3.3 Definition

Eine **deterministische** Sprache ist eine Sprache, die von einem deterministischen Kellerautomaten akzeptiert wird.

(vgl. [8] S. 624)

Hierbei ist wichtig anzumerken, dass deterministische Sprachen nach der vorherigen Bemerkung auch kontextfrei sind. Aus diesem Grund werden deterministische Sprachen auch **deterministisch kontextfrei** genannt.

4.3.3.4 Bemerkung

Sprachen $\mathcal{L}$, die sich auch als **deterministisch kontextfreie** Sprache schreiben lassen, also Sprachen für die gilt:

$$\exists \text{ deterministischer Kellerautomat } \mathcal{M} : \mathcal{L} = \mathcal{L}(\mathcal{M})$$

werden auch als **deterministisch kontextfrei** bezeichnet.

(vgl. [8] S. 624)

Die Motivation deterministisch kontextfreie Sprachen einzuführen, wurde am Anfang des Kapitels mit der effizienteren Methode begründet, das Wortproblem zu beantworten. Tatsächlich ist das Problem für deterministisch kontextfreie Sprachen in linearer Zeit, proportional zur Länge der Eingabe, lösbar. Dies wurde 1965 von D. Knuth in *On the Translation of Languages from Left to Right* gezeigt.

In derselben Arbeit wurde auch das LR(1)-Verfahren herangeführt, welche deterministisch kontextfreie Sprachen in linearer Zeit parsen kann.

4.4 LR(1)-Verfahren

Das bisher genannte LR(1)-Verfahren ist ein spezifisches Verfahren der allgemeinen LR(k)-Verfahren mit $k \in \mathbb{N}_0$.

Bezüglich LR(k) gibt es viele Begriffe, die vor der formalen Einführung erläutert werden müssen. Generell beschreibt das LR(k)-Verfahren einen Algorithmus für das Parsen von Sprachen nach dem Bottom-Up Verfahren, es wird also das Wort eingelesen und versucht durch Reduktionen das Wort auf das Startsymbol zu reduzieren.

Sprachen, die sich durch das LR(k)-Verfahren parsen lassen, werden LR(k)-Sprachen genannt. Des Weiteren werden diese Sprachen von bestimmten Grammatiken erzeugt, die LR(k)-Grammatiken heißen. Insgesamt werden daher Parser die dieses Verfahren implementieren, als LR(k)-Parser bezeichnet und werden von sogenannten LR(k)-Parsergeneratoren erstellt.

Der Begriff LR(k) selbst, steht abkürzend für eine Beschreibung der Funktionsweise des LR(k)-Verfahrens. Das „L“ beschreibt die Leserichtung des Parsers, es wird von links nach rechts gelesen. Das „R“ beschreibt die Anwendung der Reduktionen. Es werden Rechtsreduktionen angewendet. Das „k“ beschreibt die Anzahl der Symbole, die der Parser vorausschauen kann. Diese werden genutzt, um bei mehreren möglichen Reduktionen zu entscheiden, welche ausgeführt werden muss.

Mit $k \in \mathbb{N}_0$ gibt es damit theoretisch abzählbar unendlich viele dieser Verfahren, jedoch finden in der Praxis nur zwei Arten Anwendung, diese sind LR(0) und LR(1). Begründet wird dies durch die Tatsache, dass die LR(k)-Verfahren für $k > 1$ genau so mächtig sind wie das LR(1)-Verfahren. Gleichmächtig bedeutet in diesem Fall, dass genau dieselbe Sprachklasse erkannt werden kann. Diese Begründung wird durch zwei Implikationen bewiesen, die D. Knuth in *On the Translation of Languages from Left to Right* zeigt.

1. $\forall k \in \mathbb{N}_0 : \forall \text{ LR}(k)\text{-Grammatik } \mathcal{G} : \mathcal{L}(\mathcal{G}) \text{ ist deterministisch}$
2. $\forall \text{ deterministische Sprache } \mathcal{L} : \exists \text{ LR}(1)\text{-Grammatik } \mathcal{G} : \mathcal{L} = \mathcal{L}(\mathcal{G})$

Insgesamt folgt, dass $\forall \text{ LR}(k)\text{-Grammatik } \mathcal{G} \exists \text{ LR}(1)\text{-Grammatik } \mathcal{G}' : \mathcal{L}(\mathcal{G}) = \mathcal{L}(\mathcal{G}')$.

Aus diesem Grund wird im weiteren Verlauf nur das LR(1)-Verfahren betrachtet. Dieses Verfahren ist im Vergleich zu dem LR(0)-Verfahren insofern mächtiger, als ein Vorschau-symbol verwendet wird, um nicht durchführbare Reduktionen zu vermeiden und so einige Mehrdeutigkeiten aufzulösen.

4.4.1 Bestandteile des LR(1)-Verfahrens

Für die Konstruktion des LR(1)-Parsers mithilfe des LR(1)-Verfahrens, werden Grammatiken benötigt, welche gewisse Eigenschaften erfüllen. Diese Grammatiken werden dem Verfahren nach **LR(1)-Grammatiken** genannt und wie folgt definiert.

4.4.1.1 Definition

Sei $\mathcal{G} = (\mathcal{T}, \mathcal{N}, \mathcal{P}, \mathcal{S})$ eine kontextfreie Grammatik.

Die Grammatik $\mathcal{G}$ heißt **LR(1)-Grammatik**, wenn es zu jeder Rechtsableitung möglich ist mit dem nächsten Symbol das Vorkommen einer rechten Seite einer Produktion zu erkennen.

(vgl. [1] S. 242)

Diese Definition kann analog auch durch Reduktionen ausgedrückt werden. Dadurch wird sichergestellt, dass der entstehende Parser zu jedem Zeitpunkt höchstens einen Reduktionsschritt vornehmen kann und folglich der Parsingvorgang eindeutig bestimmt ist. Am ende dieses Kapitels wird außerdem folgende Definition klar werden: Eine Grammatik ist genau dann LR(1) wenn sie durch einen LR(1)-Parser geparkt werden kann.

Den Grundbaustein des entstehenden Parsers bildet das sogenannte LR(0)-Element, welches im Fall von dem LR(1)-Verfahren, um die Menge der Vorschau symbole ergänzt wird.

4.4.1.2 Definition

Sei $\mathcal{G} = (\mathcal{T}, \mathcal{N}, \mathcal{P}, \mathcal{S})$ eine Grammatik und $\Sigma = \mathcal{T} \cup \mathcal{N}$.

Ein **LR(0)-Element (der Produktion $N \rightarrow w \in \mathcal{P}$)** entspricht der Produktion mit einem Punkt in der rechten Seite. Ein LR(0)-Element wird geschrieben als:

$$N \rightarrow u.v \quad \text{mit } uv = w \text{ und } (u \in \Sigma^* \wedge v \in \Sigma^+) \vee (u \in \Sigma^+ \wedge v \in \Sigma^*)$$

Ein **LR(1)-Element (der Produktion $N \rightarrow w \in \mathcal{P}$)** ist ein Tupel bestehend aus einem LR(0)-Element und einer Menge $\{x_1, \dots, x_n\} \subseteq \Sigma$ von Symbolen. Ein LR(1)-Element wird geschrieben als:

$$[N \rightarrow u.v ; x_1, \dots, x_n] \quad \text{mit } (N \rightarrow u.v) \text{ ein LR(0)-Element von } N \rightarrow w \text{ und } x_1, \dots, x_n \in \Sigma$$

(vgl. [19] S. 180, 196)

Zwei besondere Elemente, welche oftmals benötigt werden, sind das **Startelement (einer Produktion $N \rightarrow w$)**, von der Form $N \rightarrow .w$ (mit dem Punkt an der ersten Stelle) und das **Endelement (einer Produktion $N \rightarrow w$)**, von der Form $N \rightarrow w.$ (mit dem Punkt an letzter Stelle). Alle Elemente die keine Endelemente sind, haben ein Symbol nach dem Punkt, welches für einige Beschreibungen wichtig ist. Dieses Symbol kriegt eine eigene

Bezeichnung als **Folgesymbol**.

Innerhalb des LR(1)-Elements wird das LR(0)-Element als **Kern** bezeichnet und die Menge von Symbolen heißen **Vorschausymbole**. (vgl. [19] S. 196)

4.4.1.3 Bemerkung: Shift- und Reduce-Aktionen des LR(1)-Verfahrens

Die Begründung warum LR(1)-Elemente die Grundbausteine des LR(1)-Verfahren bilden liegt in der Nutzung der Elemente im Parsing-Verfahren. Die Zustände des zugehörigen Kellerautomaten entsprechen Mengen von LR(1)-Elementen. Des Weiteren verschieben Wechsel über einem Symbol die Punkte der entsprechenden Kerne, die dieses Symbol als Folgesymbol enthalten um ein Symbol nach rechts.

In dem LR(1)-Parser wird der Stack des Kellerautomaten dafür genutzt diesen Fortschritt zu speichern. Dafür wird das nächste Symbol der Eingabe gelesen und auf den Stack gelegt. Diese Aktion wird **shift** genannt, wird aber nicht ausgeführt wenn das Wort im Stack nicht mehr reduziert werden kann. In diesem Fall bricht die Eingabe Verarbeitung ab und das Wort ist mit der gegebenen Grammatik nicht erkennbar.

Wenn im Stack ein Wort steht, welches reduziert werden kann, wird eine sogenannte **reduce**-Aktion durchgeführt, indem der entsprechende Reduktionsschritt ausgeführt und das Nichtterminalsymbol der zugehörigen Produktionsregel in den Stack gelegt wird. Dies ist genau dann der Fall, wenn die Kollektion ein Endelement besitzt, welches auf der Produktionsregel basiert, die das Wort erzeugt.

Aus diesem Grund beschreibt der Punkt des LR(0)-Element den Parsing-Fortschritt innerhalb der Produktionsregel.

Wichtig zu beachten ist dabei, dass die Reduktion nicht bei jeder beliebigen Eingabe durchgeführt werden muss. Verschiedene Verfahren haben verschiedene Methoden, um die Menge der Eingaben zu bestimmen, bei denen die Reduce-Aktion durchgeführt wird. Es können nämlich je nach Verfahren, einige Eingaben ausgeschlossen werden, bei denen bereits bekannt ist, dass mit dieser Eingabe das Wort nicht geparkt werden kann. Im Fall des LR(1)-Verfahrens sind dies genau die Vorschausymbole. Daher ist es auch üblich, analog zu den shift-Aktionen die reduce-Aktionen abhängig von Symbolen zu machen.

In den LR(1)-Elementen bleibt die Bedeutung des LR(0)-Elements die selbe. Hinzu kommen jedoch die Vorschausymbole. Diese geben an, welche Symbole direkt nach der Symbolfolge der vollständigen Ableitung von N in der gesamten Sprache $\mathcal{L}(\mathcal{G})$ vorkommen können. Damit gibt die Vorschaumenge an, in welchen Fällen tatsächlich auch die Reduktion erfolgen kann. Die Vorschausymbole können so genutzt werden, um Mehrdeutigkeiten zu vermeiden und werden daher im Verlauf des LR(1)-Verfahren mit berechnet.

Wie bereits erwähnt, spielen Mengen von LR(1)-Elementen eine wichtige Rolle in dem LR(1)-Verfahren. Sie werden daher gesondert definiert.

4.4.1.4 Definition

Eine **Kollektion** I im LR(1)-Verfahren ist eine Menge von LR(1)-Elementen.

(vgl. [19] S. 181, 196)

4.4.2 Konstruktion des LR(1)-Parsers

Da die Kollektionen als Zustände dienen, werden Operationen auf Kollektionen definiert, welche für die Berechnung der Zustände benötigt werden. Diese Operationen sind:

1. **Hülle(I) einer Kollektion I :** Der Hüllenoperator fügt der Kollektion weitere Elemente hinzu. Die Kollektion wird, für alle Elemente mit Nichtterminal als Folgesymbol, um das Startsymbol von jeder Produktion des Nichtterminals erweitert.
2. **Sprung(I, X) einer Kollektion I über dem Nichtterminalsymbol X :** Der Sprungoperator ordnet jeder Kollektion eine andere Kollektion zu. Mit allen Elementen mit X als Folgesymbol wird eine Kollektion erstellt, in denen die Punkte der Elemente um X verschoben werden. Der Sprung liefert dann die Hülle dieser Kollektion.

Dabei ist zu beachten, dass die Vorschausymbole entsprechend angepasst werden müssen. Formal werden die Operatoren wie folgt definiert.

4.4.2.1 Definition

Sei $\mathcal{G} = (\mathcal{T}, \mathcal{N}, \mathcal{P}, \mathcal{S})$ eine LR(1)-Grammatik, I eine Kollektion und $X \in \mathcal{T} \cup \mathcal{N}$

Die **Hülle einer Kollektion I** ist eine Kollektion:

$$\text{Hülle}(I) := I \cup \{ [N \rightarrow .w ; \text{First}(bM)] \mid (N \rightarrow w) \in \mathcal{G} \wedge \exists [N' \rightarrow a.Nb ; M] \in I \}$$

Der **Sprung einer Kollektion I über X** ist eine Kollektion:

$$\text{Sprung}(I, X) := \text{Hülle}(\{ [N \rightarrow aX.b ; M] \mid \exists [N \rightarrow a.Xb ; M] \in I \})$$

(vgl. [19] S. 181, 196, 197)

Durch die formalen Definition wird deutlich, wie die Vorschausymbole berechnet werden und über den Sprüngen aus anderen Kollektionen hervorgehen. Bei der Sprung-Berechnung werden die vorhandenen Vorschaumengen nur weitergegeben. Erst die anschließende Hüllen-Berechnung verändert die Vorschausymbole. Dort entsteht die Vorschaumenge der Kollektion durch die Berechnung der First-Menge von bM (mithilfe der dritten Formel zur First-Berechnung in 4.1.4.2) wobei b das Wort ist, welches auf das Nonterminal des Kerns folgt

und M die Vorschau Menge des bereits enthaltenen Elements ist.

Durch die Betrachtung aller Wörter nach dem Nonterminal des Kerns wird damit bei der Erstellung der Kollektionen sichergestellt, dass in den Vorschausymbolen genau die Symbole stehen, welche nach der vollständigen Ableitung von N vorkommen können.

Dies begründet, warum einige Eingaben bei den reduce-Aktionen ausgeschlossen werden können, da bereits bekannt ist, dass mit dieser Eingabe das Wort nicht geparkt werden kann. Ein späterer Vergleich mit dem tatsächlichen Symbol kann dadurch nicht realisierbare Reduktionen effizient verhindern.

4.4.2.2 Bemerkung: Algorithmus des LR(1)-Verfahrens

In Pseudocode beschrieben lautet das LR(1)-Verfahren wie folgt:

Algorithm 1: Zustandsberechnung (vgl. [1] S. 261)

Input: Startelement s , Terminalsymbole T , Nichtterminalsymbole N

Output: Zustände Z

```

1  $Z \leftarrow \{\text{HÜLLE}(\{[s; \$]\})\}$ 
2 do
3   foreach Kollektion  $K$  in  $Z$  do
4     foreach Symbol  $X$  in  $N \cup T$  do
5       if  $\text{SPRUNG}(K, X) \neq \emptyset$  then
6          $Z \leftarrow Z \cup \{\text{SPRUNG}(K, X)\}$ 
7 while  $Z$  geändert
8 return  $Z$ 

```

Zuerst wird die Grammatik um ein weiteres Nichtterminalsymbol S' und einer Produktion $S' \rightarrow S$ erweitert, welche **Startproduktion** genannt wird.

Den Startzustand q_{0LR} bildet dann die Hülle der Startproduktion. Diese wird anschließend in die Menge aller Zustände Z_{LR} aufgenommen. Für die Berechnung der anderen Zustände werden dann die folgenden Schritte wiederholt, bis alle Zustände berechnet worden sind. Das ist genau dann der Fall, wenn sich die Zustandsmenge durch eine Iteration nicht verändert wird.

Es wird zu jeder Kollektion in der Zustandsmenge der Sprung-Operator angewendet, wobei für jede Kollektion, die Sprünge über alle möglichen Symbole (Terminal- und Nichtterminalsymbole) betrachtet werden.

Wenn der Sprung existiert, also die entstehende Kollektion nicht leer ist, wird der neue entstehende Kollektion in die Zustandsmenge aufgenommen und bildet einen neuen Zustand. Nachdem alle Zustände berechnet worden sind, wird der Endzustand F_{LR} , als der Zustand, welcher aus dem Sprung des Startzustandes über S gebildet wurde, definiert. Dieser Zustand beinhaltet nämlich das Endelement der Startproduktion und trägt somit die Bedeutung, dass die gesamte Eingabe erfolgreich reduziert wurde.

Die dabei genutzten Sprünge dienen gleichzeitig als Basis, um die Shift-Aktionen entsprechend der Bemerkung 4.4.1.3 zu berechnen. Diese Shift-Aktionen bilden zusammen mit den Reduce-Aktionen, welche auch nach Bemerkung 4.4.1.3 berechnet werden, die Übergänge zwischen den Zuständen.

4.4.2.3 Bemerkung: LR(1)-Parser

Der dadurch entstehende LR(1)-Parser der Grammatik $\mathcal{G} = (\mathcal{T}, \mathcal{N}, \mathcal{P}, \mathcal{S})$ kann nun als Kellerautomat $\mathcal{M} = (\mathcal{Z}, \Sigma, \Gamma, \delta, \gamma_0, q_0, \mathcal{F})$ aufgefasst werden, indem die berechneten Strukturen als die nötigen Tupelelemente des Kellerautomaten interpretiert werden.

1. $\mathcal{Z} \hat{=}$ Die berechnete Zustandsmenge $\mathcal{Z}_{LR}$.
2. $\Sigma \hat{=}$ Die Terminalsymbole $\mathcal{T}$.
3. $\Gamma \hat{=}$ Die Gesamtheit aller Symbole $\mathcal{T} \cup \mathcal{N}$.
4. $\delta \hat{=}$ Die Shift- und Reduce-Aktionen.
5. $\gamma_0 \hat{=}$ ϵ .
6. $q_0 \hat{=}$ q_{0LR} .
7. $\mathcal{F} \hat{=}$ $\{F_{LR}\}$.

Die Shift-Aktionen gehen von bestimmten Zuständen aus, lesen dabei das nächste Symbol aus der Eingabe ein und lesen ein ϵ aus dem Keller ein. Dadurch bilden sie die Übergänge, zu den Sprüngen des Ausgangszustands über dem eingelesenen Symbol, und legen dieses Symbol in den Keller ab.

Reduce-Aktionen gehen von dem Ausgangszustand aus und lesen nichts aus der Eingabe ein, dafür aber alle Symbole im Keller, welche der Produktion entsprechen, über der reduziert wird. Dabei bilden sie die Übergänge von dem Ausgangszustand zurück zu dem vorherigen Zustand.

Nach Bemerkung 4.3.3.2 ist insbesondere sichtbar, dass nur eine Teilmenge der kontextfreien Grammatiken von LR(1)-Parsern eingelesen werden kann, tatsächlich ist es sogar der Fall, dass der Kellerautomat deterministisch ist, sodass das LR(1)-Verfahren, eine Vorgehensweise beschreibt Parser zu generieren, die genau die deterministisch-kontextfreien Sprachen erkennen. Grammatiken welche nicht durch die LR(1)-Parser geparkt werden können, führen in der Berechnung zu Konflikten in den Übergängen und damit zu nicht-deterministischen Automaten. Dies ist genau dann der Fall, wenn die Grammatik keine LR(1)-Grammatik ist. Der Beweis dazu kann in *On the Translation of Languages from Left to Right* [14] von Donald Knuth nachgelesen werden.

5 Analyse des LR(1)-Parsergenerators

Mithilfe der Theorie des LR(1)-Verfahrens und der deterministisch kontextfreien Sprachen kann nun die Umsetzung des LR(1)-Parsergenerators analysiert werden. Diese Analyse wird in vier thematisch getrennten Abschnitten durchgeführt, welche die Fragestellungen aus der Zielsetzung der Bachelorarbeit (siehe Kapitel 2.3) behandeln. Dabei wird die zweite Frage nach den Algorithmen für das LR(1)-Verfahren und Algorithmen für die Visualisierung aufgeteilt

Der erste Abschnitt behandelt den Systemaufbau. Das Ziel dieses Abschnitts liegt darin die Wahl der einzelnen Aspekte der Datenstrukturen anhand des theoretischen Hintergrundes zu erläutern.

Insbesondere müssen hier zeitgleich sowohl die theoretischen Anforderungen als auch die Anforderung der Visualisierung betrachtet werden.

Im weiteren Verlauf werden die Anpassungen der aus der Literatur [1] bereits bekannten Algorithmen zur Berechnung des LR(1)-Parsers begründet. Diese Anpassungen basieren auf der spezifischen Umsetzung der Datenstrukturen und haben abhängig von der Datenstruktur Änderungen der Algorithmen zur Folge. Anschließend werden diese Änderungen einer genaueren Überprüfung unterzogen, um deren Korrektheit sicherzustellen.

Aufgrund der Gestaltung des Parsergenerators als didaktisches Mittel, basieren viele der Funktionalitäten der Visualisierung auf der Fehlerüberprüfung und Fehlerbehebung der Nutzereingaben als auch der Korrektur der Abfragen. Fehler in der Nutzereingabe entsprechen dabei Fehlern der eingegebenen Grammatik und werden im LR(1)-Parsergenerator durch entsprechende Algorithmen zur geeigneten Überprüfung der Grammatik entdeckt und korrigiert. Die Korrektheit dieser Überprüfungen und Korrekturen ist dementsprechend auch Thema dieses Kapitels.

Nachdem die Analyse abgeschlossen ist, können die verwendeten Erkenntnisse genutzt werden, um eine Evaluierung des Parsergenerators durchzuführen.

5.1 Datenstrukturen

Die Verwendung der richtigen Datenstrukturen ist ein kritischer Aspekt des gesamten Projekts, da die Umsetzung der Algorithmen stark von den benötigten Datenstrukturen abhängt. Weiterhin müssen die Datenstrukturen der Darstellungsweise dienlich sein, da die verschiedenen Daten ausgegeben werden müssen, während der Nutzer die interaktiven Visualisierungsmöglichkeiten des Programms nutzt.

Dadurch begründet lassen sich die Anforderung an die Datenstrukturen in zwei Kategorien klassifizieren:

1. Die Anforderungen der theoretischen Definitionen, welche dem Ziel dienen, dass die Datenstrukturen in den Algorithmen des LR(1)-Verfahrens für die Erkennung deterministisch kontextfreier Sprachen genutzt werden können.
2. Die Anforderungen der interaktiven Visualisierung, welche dem Ziel dienen, dass die Datenstrukturen für die interaktiven Aspekte der Visualisierung genutzt werden können.

Die erste Anforderung ist dadurch begründet, dass die Algorithmen des LR(1)-Verfahrens bereits bewiesen sind und die Verwendung dieser Algorithmen dem generierten Parser die Erkennung deterministisch kontextfreier Sprachen durch die bereits behandelten Theorie 4 garantiert.

Die Anforderungen an die Strukturen durch die Interaktivität sind nicht in dem Kapitel behandelt worden, daher soll deren Wichtigkeit hier kurz argumentiert werden.

Es müssen für die Interaktion passende Wrapper-Funktionen¹ in den Datenstrukturen verfügbar sein, um die Umsetzung der interaktiven Aspekte zu erleichtern.

Zu den wichtigsten Beispielen solcher Funktionen zählt neben dem Vergleichsoperator **equals**, welcher prüft, ob zwei Strukturen gleich² sind, auch die Mengenoperation **append**, die einer bestehenden Instanz der Datenstruktur weitere Daten beifügt. Diese relativ simplen Operatoren sind jedoch in der Umsetzung nicht trivial, da die theoretischen Formulierungen der Algorithmen abstrakt definierte Strukturen umfasst, wie beispielsweise Mengen, deren Eigenschaften formal einfach definiert werden können. Zum Beispiel können die Eigenschaften: „In einer Menge wird die Reihenfolge nicht beachtet“ und „In einer Menge wird nicht beachtet, ob ein Element mehrfach enthalten ist.“ zwar einfach definiert werden, dennoch müssen diese Eigenschaften bei der Umsetzung dieser Operationen beachtet werden. Die **append**-Operation sollte ein Element nicht hinzufügen, wenn dieses bereits enthalten ist. Entsprechend muss bei dem Vergleichen zweier Strukturen durch die **equals**-Operation oder bei der Ausgabe der Daten beachtet werden, dass die Reihenfolge der Elemente einer Menge keine Rolle spielt.

¹Funktionen, welche die Notwendigkeit, die spezifische Umsetzungsmethode zu kennen, durch Abstraktion entfernen.

²Die Gleichheit ist abhängig von der Datenstruktur und beinhaltet bei verschiedenen Strukturen verschiedene Vergleiche.

Im Folgenden wird die interne Repräsentation der verschiedenen Daten anhand von UML-Klassendiagrammen erläutert. Angesichts der Komplexität mancher abstrakten Strukturen, muss die Umsetzung, analog zu den in Kapitel 4 vorgestellten theoretischen Grundlagen, stückweise erfolgen und die Gesamtstruktur aus einzelnen Datenstrukturen zusammengesetzt werden.

5.1.1 Grammatik

Weil in der Umsetzung keine Operationen auf die Gesamtheit einer Grammatik angewendet werden müssen und eine Grammatik keine übergreifenden Einschränkungen³ beinhaltet, sondern als Tupel nur als Sammlung verschiedener Strukturen dient, wird keine gesonderte Datenstruktur für die Grammatik an sich benötigt. Da zur Laufzeit jeweils nur eine Grammatik betrachtet wird, können stattdessen die einzelnen Tupelemente einzeln gespeichert werden. Auf diese werden jedoch Operationen angewendet, sodass die Konstruktion von gesonderten Datenstrukturen benötigt wird.

5.1.1.1 Terminal- & Nichtterminalsymbole

Symbol
- symbol: String
+ append(symbol: String): Boolean + equals(symbol: String): Boolean

Abbildung 5.1: UML-Klassendiagramm eines Symbols

Die einfachsten Strukturen bilden die Symbole (Definition 4.1.1.1), welche mit Blick auf die Ein- und Ausgabe als einfache Strings umgesetzt werden. Die **append**-Operation entspricht hierbei der Konkatenation (Bemerkung 4.1.1.2) und ist in JavaScript bereits gemäß der theoretischen Definition durch den „+“-Operator für Strings integriert. Genau so ist ein entsprechender „===“-Vergleichsoperator in JavaScript umgesetzt, welcher genau dann den Wahrheitswert **true** annimmt, wenn die Strings gleich sind. Insgesamt sind damit alle Anforderungen an ein einzelnes Symbol durch den primitiven Datentypen String erfüllt und es bedarf daher keinem zusätzlichen Datentyp, um Symbole darzustellen.

Weiterhin haben Wörter (Definition 4.1.1.1) die gleichen technischen Anforderungen und können daher auch durch Strings umgesetzt werden. Somit wären zwar alle Anforderungen erfüllt, jedoch entsteht dadurch das Problem, dass Symbole und Wörter strukturell nicht mehr unterschieden werden können. Dieses Problem wird dadurch gelöst, dass Wörter als String-Arrays gespeichert werden. Die vom Nutzer vorgegebenen Symbole werden in einer

³Einschränkungen die sich auf mehrere der Unterliegenden Strukturen gleichzeitig auswirken.

gesonderten Datenstruktur gespeichert, als Menge von Symbolen. Es eignet sich daher die Struktur des Alphabets (Definition 4.1.1.1).

Alphabet
- symbols: String[]
+ append(symbol: String): Boolean + has(symbol: String): Boolean

Abbildung 5.2: UML-Klassendiagramm eines Alphabets

Da die Symbole als Strings umgesetzt werden, eignet sich zwar ein String-Array, um die Menge von Symbolen zu speichern, jedoch erfüllen die Arrays die Eigenschaften der Menge nicht. Um diese Anforderungen eines Alphabets als endliche Menge zu wahren, muss der Zugriff auf die Datenmenge durch die **append**-Operation geregelt werden. Dies bedeutet, dass bereits enthaltene Elemente nicht mehrfach eingefügt werden können.

Für die Überprüfung, ob es sich bei einem String um ein Symbol handelt, wird die **has**-Operation gebraucht, welche prüft, ob ein String in dem Alphabet enthalten ist und einen entsprechenden Wahrheitswert zurückliefert. Diese ist bereits für Arrays in JavaScript als die sogenannte **includes**-Funktion umgesetzt und nutzt für Strings intern auch den notwendigen „===“-Operator.

In der Implementierung gibt es für die Menge der Terminal- und Nichtterminalsymbole jeweils eine globale Instanz eines Alphabets. Dies wird mit der Definition 4.1.2.1 begründet, in der die Terminal- und Nichtterminalsymbole beides Alphabete sind und dementsprechend auch die gleichen Anforderungen haben. Für die weitere Verarbeitung muss jedoch zwischen den beiden Arten der Symbole unterschieden werden. Diese Unterscheidung kann durch die jeweilige **includes**-Funktion durchgeführt werden.

5.1.1.2 Produktionsregeln

Die Produktionsregeln (Definition 4.1.2.1) benötigen eine komplexere Struktur, um die nötige Information zu speichern und die notwendigen Operationen bereitzustellen. Da jedoch nur kontextfreie Grammatiken betrachtet werden, müssen nach Bemerkung 4.1.2.3 nur Regeln der folgenden Form gespeichert werden.

$$i. \quad N \rightarrow w$$

Die Informationen die in dieser Darstellung gezeigt sind, werden für die anschließende Berechnung genutzt und müssen daher in der Datenstruktur gespeichert werden. Das i entspricht der Nummerierung der Produktionsregel und das N dem Nichtterminalsymbol, welches durch die rechte Seite der Produktion w ersetzt werden kann. Da die Beziehung

Rules
- symbolToProduction: Map(String, Int[]) - productions: String[][]
+ of(symbol: String): String[][] + has(symbol: String, production: String[]): Boolean + append(symbol: String, production: String[]): Boolean

Abbildung 5.3: UML-Klassendiagramm der Menge der Produktionsregeln

zwischen Nichtterminalsymbol und der rechten Seite der Produktion einseitig⁴ ist, muss lediglich zu jedem Nichtterminalsymbol gespeichert werden mit welchen rechten Seiten das Symbol ersetzt werden kann. Infolgedessen eignet sich eine Map von den Nichtterminalsymbolen zu dessen rechten Seiten für diese Anforderung gut.

Weiterhin kann die Reihenfolge zwar beliebig sein, jedoch müssen die Regeln durchnummeriert werden, um das i der darstellungsweise zu speichern, zumal dieses für die spätere Berechnung benötigt wird.

Das Problem dabei ist, dass durch eine Map die Reihenfolge der Regeln zerstört wird, sodass eine direkte Map von Nichtterminalsymbolen zu den rechten Seiten nicht möglich ist. Stattdessen muss die Datenstruktur entsprechend angepasst werden, indem die Produktionsregeln in einer festen Reihenfolge in einem gesonderten Array gespeichert und durch die Map nur referenziert werden. Da die Symbole durch Strings repräsentiert werden, wird eine rechte Seite als Array von Strings umgesetzt. Dementsprechend bildet die Gesamtheit aller rechten Seiten ein zweidimensionales Array von Strings. Die Map referenziert die Produktionen durch die Indizes und wird daher als Map von Strings zu Int-Arrays umgesetzt, wobei das Array wegen der Möglichkeit von mehreren Produktionen pro Nichtterminalsymbol benötigt wird.

Wegen der relativ komplexen Datenstruktur wird auch hier das Einfügen neuer Produktionen durch eine **append**-Funktion kontrolliert. Diese fügt gleichzeitig die rechte Seite ein, erstellt die Referenz und fügt diese in die Map ein. Dies geschieht jedoch nur, wenn die entsprechende Produktionsregel nicht bereits eingefügt worden ist. Dafür baut die Operation auf eine Funktion **has** auf, welche zu einer gegebenen Produktionsregel prüft, ob diese bereits vorhanden ist. Auch der Zugriff wird durch eine Funktion **of** erleichtert, die für ein Nichtterminalsymbol die rechten Seiten dereferenziert und als Array zurückliefert.

Damit fehlt für die Speicherung einer Grammatik nur noch das Startsymbol, welches einfachheitshalber als globale Variable vom Datentyp String gespeichert werden kann.

⁴Einseitig bedeutet jede Produktionsregel enthält ein eindeutiges Nichtterminalsymbol.

Mit diesen Datenstrukturen kann somit bereits die gesamte Eingabeverarbeitung vorgenommen werden. Bei dem Nutzen des Programms werden entsprechende Textfelder bereitgestellt, in denen der Nutzer die gewünschte Grammatik eingeben kann. Diese können anschließend eingelesen und durch die **append**-Funktionen zu der jeweiligen Instanz der dazugehörigen Datenstruktur hinzugefügt werden, und liegen infolgedessen für die weitere Verarbeitung vor.

5.1.2 First- & Follow-Mengen

Nach dem die Eingabe eingelesen wurde, beginnt die Verarbeitung der Daten, um diese für die Parsergenerierung vorzubereiten. Die Vorbereitung sieht die Berechnung von den First-Mengen (Definition 4.1.4.1) vor, da diese für das LR(1)-Verfahren benötigt werden, aber auch die Follow-Mengen (Definition 4.1.4.3), welche zwar nicht für die Algorithmen des LR(1)-Verfahrens benötigt werden, jedoch Auskunft über mögliche Fehler geben können und daher für die Interaktivität benötigt werden.

5.1.2.1 First-Mengen

Nach Bemerkung 4.1.4.2 ist bekannt, dass die beliebige First-Mengen sich rekursiv durch die First-Mengen aller möglichen Symbole eindeutig berechnen lassen. Aus diesem Grund lässt sich die Erstellung der Datenstruktur auf einzelne Symbole beschränken. Für die First-Mengen von Wörtern wird eine entsprechende rekursive Funktion implementiert.

MapSet
+ symbols: Map(String, String[])
+ append(symbol: String, first: String): Boolean + equals(symbol: String, input: String[]): Boolean

Abbildung 5.4: UML-Klassendiagramm der First- und Follow-Mengen

Da die First-Mengen von Symbolen abhängen, kann $\text{First}(\cdot)$ (Definition 4.1.4.1) als Abbildung von Symbolen zu Symbolmengen aufgefasst werden. Insofern eignet sich für die Konstruktion der Datenstruktur eine Map von Strings zu String-Arrays. Da jedoch aufgrund der Mengeneigenschaften der First-Menge auch hier die Nutzung einer gewöhnlichen Map ausgeschlossen ist, gilt es stattdessen die Wrapper-Funktionen **append** und **equals** umzusetzen. Die **append**-Operation fügt auch in diesem Fall ein Symbol der entsprechenden First-Menge hinzu, sofern dieses nicht bereits enthalten ist. Da die Verarbeitung der Daten auch ein wichtiger Bestandteil der Parsergenerierung ist, wird dieser im interaktiven Teil des Programms abgefragt. Für die Korrekturen wird daher der Vergleichsoperator

equals gebraucht, welcher die Gleichheit zweier Mengen durch das Prinzip der gegenseitigen Inklusion prüft.

5.1.2.2 Follow-Mengen

Da die Follow-Mengen nicht für die Parsergenerierung benutzt und folglich nicht abgefragt werden, ist keine **equals**-Operation notwendig. Dennoch sind diese mit den gleichen Anforderungen verbunden, wodurch auch in diesem Fall eine Umsetzung mittels der MapSet-Datenstruktur (Abbildung 5.4) erfolgt.

5.1.3 LR(1)-Verfahren

Im Anschluss an die Verarbeitung der Grammatik zu deren First- und Follow-Mengen, können mithilfe der First-Mengen die LR(1)-Zustände (Definition 4.4.1.4) berechnet werden. Durch dessen Größe ist die Menge aller Zustände als gesamte Struktur sehr komplex.

Aus diesem Grund liegt das Schlüsselkonzept der folgenden Datenstrukturen darin, dass alle Aspekte aufbauend erstellt werden, wodurch sich die Komplexität der einzelnen Implementierungen verringern lässt. Dazu gehören auch die Hilfsfunktionen die oftmals auf die unterliegende Struktur aufbauen müssen und dafür selbst die unterliegenden Hilfsfunktionen nutzen.

Wenn von den zusätzlichen Daten abstrahiert wird, wird die Menge der Zustände als eine Menge von Kollektionen (Definition 4.4.1.4) definiert. Kollektionen selbst lassen sich als Mengen von LR(1)-Elementen (Definition 4.4.1.2) auffassen, welche selbst wiederum Tupeln von Symbolen entsprechen werden können. Dadurch ergibt sich eine Menge von Mengen von Tupel von Symbolen als Struktur. Diese einzelnen Teilstrukturen sind leicht vom Gesamten trennbar, sodass sich die komplexe Struktur in kleinere weniger komplexe Strukturen aufteilen lässt. Insgesamt ergibt sich der in Abbildung 5.5 gezeigte Aufbau.

5.1.3.1 Elemente

Elemente können nach Definition 4.4.1.2 vergleichbar zu einer Produktionsregel durch Verwendung der folgenden informationsbeogenen Schreibweise gespeichert werden.

$$[N \rightarrow w.v ; x_1, \dots, x_n]$$

Dies beinhaltet die Angabe einer Produktionsregel welche, wie in 5.1.1.2 behandelt, aus einem String, der das Nichtterminalsymbol angibt, und einem Array von Strings, welcher die Symbole der Produktion speichert, besteht. Da für die Angabe der Reduktion der Index der Regel innerhalb der Reihenfolge aller Produktionsregeln benötigt wird, wird dieser auch gespeichert. Damit ist die Produktionsregel durch die Referenz über den Index eindeutig bestimmt, sodass das Speichern der rechten Seite der Produktion selbst redundant ist, da diese stattdessen über dem *productions*-Array der *Rules*-Datenstruktur (Abbildung 5.3) bezogen werden kann.

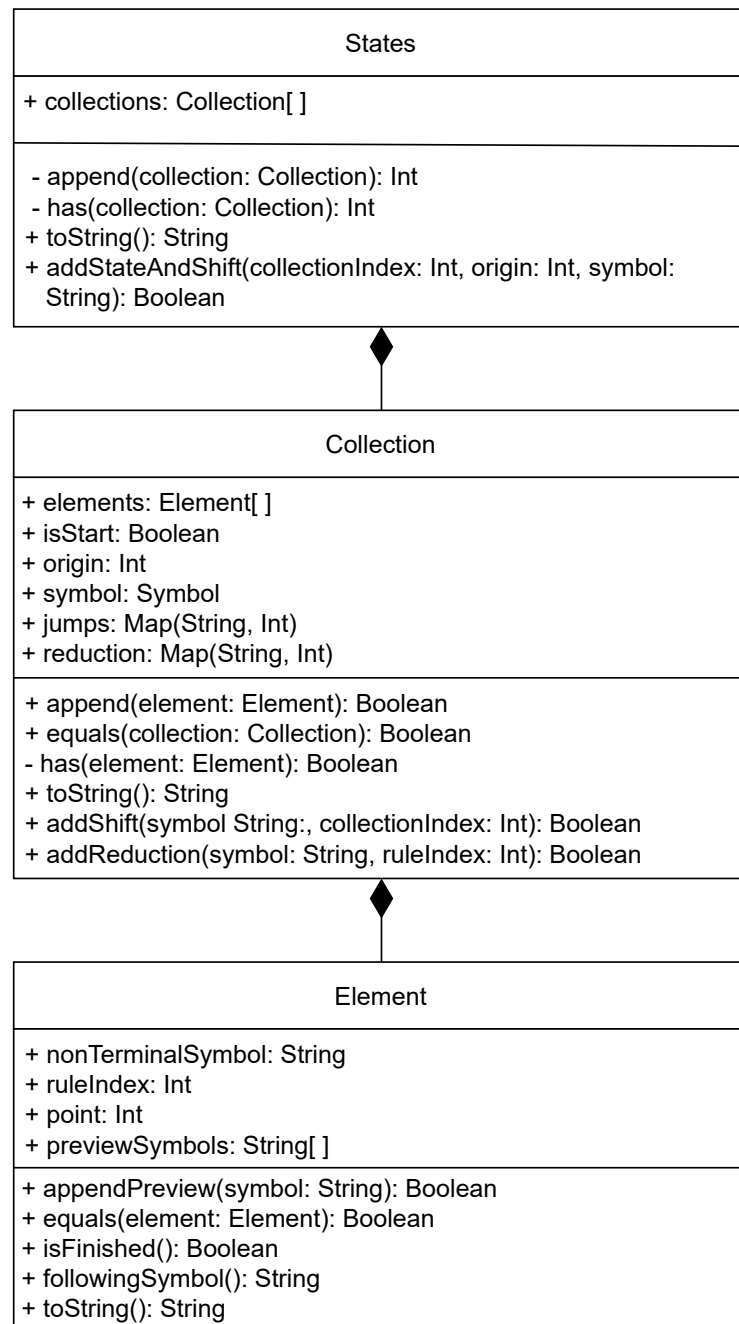


Abbildung 5.5: UML-Klassendiagramm der Menge der Zustände

Aus der obigen Schreibweise ist ersichtlich, dass neben der Produktionsregel noch weitere Daten gespeichert werden müssen. Dazu gehören die Vorschau Symbole $x_1, \dots, x_n$, welche auf Grund der Mengeneigenschaften wieder durch eine Hilfsfunktion **appendPreview** in das String-Array *previewSymbols* eingefügt werden. Außerdem muss der Parsing-Fortschritt

gespeichert werden, welcher innerhalb der Regel durch den Punkt „.“ angegeben wird. Umgesetzt wird dieser Fortschritt als ein Index auf die Stelle des Punktes, da sich eine solche Umsetzung als Vorteilhaft erweist. Beispielsweise wird dadurch die Implementierung der zwei Hilfsfunktionen **isFinished** und **followingSymbol** stark vereinfacht. **isFinished** gibt an, ob die Regel in Bezug auf den Parsing-Fortschritt fertig ist. Vereinfacht kann durch einen Vergleich des **point**-Indizes mit der Länge der Regel berechnet werden. Wenn die Regel noch nicht fertig ist, kann das Folgesymbol (Definition 4.4.1.2) bestimmt werden. Dies kann durch die Umsetzung des Fortschrittes als Index durch den Arrayzugriff *Rules productions[ruleIndex][point]* (siehe Abbildung 5.3) auch leicht berechnet werden. Anders als die Vergleiche von Symbolen, welche als String umgesetzt sind, gibt es hier keinen bereitgestellten **equals**-Vergleich von JavaScript. Folglich muss der Vergleich erst in der **equals**-Operation umgesetzt werden, im Zuge dieser die Produktionsregel, der Punkt und die Vorschausymbole miteinander verglichen werden. Für die Ausgabe auf die graphische Nutzeroberfläche überschreibt die Struktur außerdem die **toString**-Methode, sodass die Elemente in der oben genannten Schreibweise an den Nutzer ausgegeben werden.

5.1.3.2 Kollektion

Die darauf aufbauende Struktur einer Kollektion enthält ein Array aller Elemente. Als Hilfsfunktion wird hier wieder die **append**-Mengenoperationen benötigt, da die Kollektion eine Menge ist. Weitere Hilfsfunktionen sind die **has**- und **equals**-Operatoren.

Mithilfe des **has**-Operators findet die Überprüfung statt, ob ein Element bereits enthalten ist. Dafür wird auf dem **equals**-Vergleichsoperator eines Elements aufgebaut, indem dieser für die Vergleiche genutzt wird. Da die **includes**-Funktion von JavaScript nicht so konfiguriert werden kann, dass der **equals**-Operator eines Elements genutzt wird, muss die **has**-Operation hier explizit umgesetzt werden.

Weil die Datenstruktur *States* 5.5 auch eine eigene **has**-Operation benötigt, wird in der Kollektion-Datenstruktur analog ein **equals**-Operator benötigt, um den notwendigen Vergleich durchzuführen. Dieser prüft die Gleichheit zweier Kollektionen durch gegenseitige Inklusion und baut dafür auf die bereits vorgestellten **has**-Operatoren auf.

Obwohl die Datenstruktur damit bereits alle theoretischen Anforderungen der Definition 4.4.1.4 erfüllt, benötigt diese dennoch weitere Daten, um auch die Anforderungen an die Darstellung zu erfüllen.

So müssen beispielsweise die Zustände eine Bezeichnung erhalten. Die Zustände können dafür durch eine Nummerierung eindeutig gekennzeichnet werden, zumal diese Definitionsgemäß immer in einer eindeutigen Reihenfolge über die Hülle- und Sprung-Funktionen 4.4.2.1 berechnet werden.

Damit der Nutzer die Schritte der Berechnung nachvollziehen kann, wird zudem noch

eine Angabe benötigt, über welchen Sprung der Zustand erstmals erreicht wurde. Dafür kann analog zu den First-Mengen die Kollektion $\text{Sprung}(I, X)$ als mehrdimensionale Abbildung $\text{Sprung}(\cdot, \cdot)$ mit dem Zustand und Symbol als Parameter verstanden werden. Um dann die Berechnungsschritte nachvollziehen zu können, müssen beide Angaben des Sprung-Operators angegeben werden. Der vorherige Zustand wird durch die oben genannte Nummerierung angegeben und das Symbol als String.

Zuletzt muss für den Startzustand, welcher nicht durch einen Sprung berechnet wird, eine Variable angelegt werden, die angibt, ob es sich um den Startzustand handelt.

5.1.3.3 Shift & Reduce-Aktionen

Da die Shift- und Reduce-Aktionen den Übergängen von einem Zustand über einem Symbol zu einem anderen Zustand entsprechen, ist es naheliegend für einen erleichterten Zugriff diese zusammen mit dem Ausgangszustand zu speichern. Dadurch entfällt die Notwendigkeit diesen Zustand explizit zu speichern, sodass die Shift-Aktionen als Map von dem Symbol zu dem Index des Zielzustandes gespeichert werden kann. Analog können die Reduktionen als Map von Symbolen zu dem Index der Produktionsregel des Endelements, über dem reduziert wird, gespeichert werden. Um auch hier das Einfügen zu erleichtern, werden von der Datenstruktur die Hilfsfunktionen **addShift** und **addReduction** bereitgestellt.

5.1.3.4 Zustände

Durch die Aufteilung der Strukturen, braucht die Menge aller Zustände zur Speicherung der Daten nur noch ein Array von Kollektionen. Wie bei den Kollektionen wird eine **append**-Operation benötigt, welche das geregelte Hinzufügen übernimmt, sofern der Zustand nicht enthalten ist. Die dafür notwendige **has**-Operation baut auch hier auf die **equals**-Operation der Kollektionen auf. Dadurch wird der komplexe Vergleich, der für die Hinzunahme von Zuständen benötigt wird, analog zu der gesamten Struktur auf den unterliegenden Funktionen aufgebaut.

Weil die Algorithmen bei neuen Zuständen auch die Speicherung des Übergangs vorsehen, wird eine Hilfsfunktion bereitgestellt, die ein erleichtertes hinzufügen von Zuständen erlaubt. Diese fügt gleichzeitig die Sprünge zu dem nächsten Zustand und den Zustand selbst hinzu.

Für das Ausgeben der Zustände wird eine **toString**-Methode benötigt, welche nach dem gleichen Konzept die unterliegende **toString**-Methode der Elemente nutzt.

5.2 Algorithmen des LR(1)-Verfahren

Durch die Anpassung der Datenstrukturen müssen auch die Algorithmen angepasst werden. Im Folgenden werden genau diese Anpassungen auf deren Korrektheit geprüft und die Laufzeit der veränderten Algorithmen berechnet. Die Laufzeit wird dann in der Landau-Notation der Worst Case Laufzeit $\mathcal{O}$ berechnet. Weiter wird die Implementierung der Algorithmen mithilfe der Datenstrukturen erläutert. Dafür werden die angepassten Algorithmen mit den Algorithmen aus der Literatur [1] in Pseudocode gegenübergestellt. Dabei ist zu beachten, dass der bekannte Algorithmus aus der Literatur als korrekt angenommen wird, woraufhin ein Vergleich mit dem angepassten Algorithmus des LR(1)-Parsergenerator erfolgt.

Für den Vergleich sind dabei folgende Anmerkungen zu beachten:

1. Der Algorithmus aus der Literatur ist in der Gegenüberstellung immer links zu finden. Der angepasste Algorithmus ist immer rechts zu sehen.
2. Der verwendete Pseudocode basiert auf den gängigsten Programmiersprachen und wird daher nicht weiter erklärt.
3. Verwendete Variablen, welche aus dem Kontext klar sind, werden nicht gesondert definiert. Wenn nicht anders angegeben, tragen die Variablen folgende Bedeutung: $T \hat{=}$ Terminalsymbole; $N \hat{=}$ Nichtterminalsymbole; $P \hat{=}$ Produktionsregeln; $S \hat{=}$ Startsymbol; $X, Y \in N \cup T$; $u, v, w \in (\mathcal{T} \cup \mathcal{N})^*$; $t \in T$; $A, B, C \in N$; $p \in P$
4. Um den Algorithmus im Text zu referenzieren, wird folgende Schreibweise verwendet. „Z.“ steht für Zeile und gibt den Beginn der Referenz an. Die darauf folgende Zahl oder der folgende Intervall gibt die Zeile oder mehrere Zeilen an. Der letzte Buchstabe gibt den Algorithmus an, wobei „l“ den linken und „r“ den rechten Algorithmus referenziert. Eine Angabe der Form Z.6l entspricht der sechsten Zeile des linken Algorithmus. Analog entspricht Z.3-5r den Zeilen drei bis fünf des rechten Algorithmus.

5.2.1 First

Algorithm 2: First (vgl. [1] S. 221)

Input: Terminalsybole T ,
Nichtterminalsymbole N ,
Produktionen P
Output: Map(Symbol, First-Menge) F

```
1 do
2   foreach Symbol  $X \in T \cup N$  do
3     if  $X \in T$  then
4        $F[X] \leftarrow \{X\}$ 
5     else if  $X \in N$  then
6       if  $X \rightarrow \epsilon \in P$  then
7          $F[X] \leftarrow F[X] \cup \{\epsilon\}$ 
8       else if  $X \rightarrow Y_1 \dots Y_n \in P$ 
9         then
10        for  $i \leftarrow 2, \dots, n$  do
11          if  $Y_1 \dots Y_{i-1} \xrightarrow{*}_G \epsilon$ 
12            then
13               $F[X] \leftarrow$ 
14                 $F[X] \cup F[Y_i] \setminus \{\epsilon\}$ 
15              if  $Y_1 \dots Y_n \xrightarrow{*}_G \epsilon$  then
16                 $F[X] \leftarrow F[X] \cup \{\epsilon\}$ 
17 while  $F$  geändert
18 return  $F$ 
```

Algorithm 3: Anpassung von First

Input: Terminalsybole T ,
Nichtterminalsymbole N ,
Produktionen P
Output: Map(Symbol, First-Menge) F

```
1 foreach Terminal  $t$  in  $T$  do
2    $F[t] \leftarrow \{t\}$ 
3 do
4   foreach Produktion
5      $p = X \rightarrow A \in P$  do
6     if  $A = \epsilon$  then
7        $F[X] \leftarrow F[X] \cup \{\epsilon\}$ 
8     else if  $A = Y_1 \dots Y_n$  then
9       for  $i \leftarrow 1, \dots, n$  do
10        if  $i < n$  then
11           $F[X] \leftarrow$ 
12             $F[X] \cup F[Y_i] \setminus \{\epsilon\}$ 
13          if  $\epsilon \notin F[Y_i]$  then
14            break
15        if  $i = n$  then
16           $F[X] \leftarrow F[X] \cup F[Y_n]$ 
17 while  $F$  geändert
18 return  $F$ 
```

Korrektheit In dem Algorithmus der First-Berechnung ist zu beachten, dass in der Anpassung nicht über Produktionsregeln mit gesuchten Eigenschaften (z.B.: $X \rightarrow \epsilon \in P$ in Z.6l oder $X \rightarrow Y_1 \dots Y_n \in P$ in Z.8l) iteriert werden kann. Stattdessen wird über alle Regeln der Form $X \rightarrow A$ iteriert (in Z.4r) und die gewünschten Eigenschaften (z.B.: $p = X \rightarrow \epsilon$) werden über **if**-Anweisung gefiltert (z.B.: **if** $A = \epsilon$ in Z.5r). Diese Anpassung ist mit der Datenstruktur (siehe 5.1.1.2) zu begründen, denn darin sind alle rechten Seiten in einem Array gespeichert, sodass eine Iteration notwendig ist, um die gesuchte Produktionsregel zu finden.

Des Weiteren muss in dem angepassten Algorithmus nicht mehr über alle Nichtterminalsymbole (wie in Z.2l) iteriert werden, da alle benötigten Nichtterminalsymbole bereits in

den linken Seiten der Produktionsregeln enthalten sind. Ferner können definitionsgemäß (4.1.2.2) nur Nichtterminale in der linken Seite auftreten, sodass die Überprüfung $X \in N$ aus Z.5l, in dem angepassten Algorithmus redundant ist.

Da nun jedoch die Schleife in Z.2l umgangen wird, welche über alle Symbole (auch über die Terminalsymbole) iteriert, wird eine weitere Schleife benötigt, die über alle Terminalsymbole iteriert und die entsprechende Zuordnung in Z.4l durchführt, um Z.3-4l umzusetzen. Diese Zuordnung ist unabhängig von den berechneten First-Mengen und kann daher in Z.1-2r außerhalb der **do-while**-Schleife umgesetzt werden.

Mit den identischen Zuweisungen in Z.7l und Z.6r, und auch Z.4l und Z.2r und den nach der obigen Begründungen äquivalenten **if**-Abfragen, ist die Korrektheit der Zeilen Z.1-7r bewiesen.

Damit bleibt nur noch die Äquivalenz der Zeilen Z.9-13l und Z.8-13r zu zeigen. Diese sind mit Bemerkung 4.1.4.2 zu begründen, da daraus hervorgeht, dass ein Wort sich nur zu ϵ ableiten lässt, wenn sich alle Symbole des Wortes zu ϵ ableiten lassen. Diese Überprüfung wird in dem rechten Algorithmus durch Z.13-14r vorgenommen, da dort die Schleife abgebrochen wird, wenn sich ein Symbol nicht zu ϵ ableiten lässt.

7 Genauer gesagt werden die First-Mengen in Z.9-10r nur dann hinzugenommen, wenn alle vorherigen Symbole sich zu ϵ ableiten lassen. Sofern dies für alle Symbole der Fall ist, kommt mit Z.11-12r das ϵ hinzu. Damit sind diese Zeilen äquivalent zu den bedingten Zuordnungen in Z.10-11l und Z.12-13l und die Korrektheit des gesamten Algorithmus gezeigt.

Die Terminierung der Algorithmen folgt aus der Äquivalenz der Zeilen Z.3-13l und Z.4-14r und der Endlichkeit der Produktionsmenge P .

Laufzeit Vor der Laufzeitanalyse des angepassten Algorithmus müssen Beobachtungen bezüglich der Laufzeit folgender Umsetzungen gemacht werden.

In der Implementierung werden die Vereinigungen aus Z.6r, Z.10r und Z.12r durch die in Kapitel 5.1 behandelten **append**-Funktionen umgesetzt. Diese benötigen aufgrund der Umsetzung der First-Mengen als Array für die Überprüfung, ob das Element bereits enthalten ist, eine Laufzeit von $\mathcal{O}(|T|)$, da im Worst Case die First-Menge alle Terminalsymbole enthalten kann.

Mit derselben Begründung benötigt die **includes**-Funktion in Z.13r auch eine Laufzeit von $\mathcal{O}(|T|)$.

Folglich kann die Laufzeitanalyse mit der Betrachtung der **for-each**-Schleife in Z.1-2r beginnen. Offensichtlich gilt, dass diese Schleife $|T|$ Schleifendurchläufe benötigt und daher im Vergleich zu der **do-while**-Schleife in Z.3-15r in der $\mathcal{O}$ -Notation nicht relevant ist. Dies liegt daran, dass allein die **do-while**-Schleife $|N| \cdot |T|$ -mal durchlaufen wird. Um dies zu verdeutlichen wird von einer Mindestanzahl an Änderungen ausgegangen, welche nötig sind, damit die Schleife nicht abgebrochen wird. Dadurch läuft die Schleife so lang wie möglich und es kann die Worst Case Laufzeit der Schleife berechnet werden. Bezüglich der Änderungen gilt dabei Folgendes. Es können höchstens die First-Mengen aller Nichtterminale N

jeweils alle Terminale T enthalten, sodass insgesamt $|N| \cdot |T|$ Symbole zu den First-Mengen hinzugefügt werden. Da bei jedem Schleifendurchlauf mindestens ein Terminalsymbol hinzugefügt werden muss, damit eine Änderung erfolgt, ergeben sich $|N| \cdot |T|$ Durchläufe.

Innerhalb dieser Schleife ergehen anschließend zwei geschachtelte **for-each**-Schleifen. Die Schleifen sind die Schleife aller Produktionen in Z.4-14r, welche daher $|P|$ Durchläufe benötigt und die Schleife aller Symbole der rechten Seite einer Produktionsregel in Z.8-14r. Im Worst Case wird diese Schleife so oft durchlaufen, wie die Anzahl der Symbole der längsten Produktionsregel ($\sup_{p \in P} |p|$, wobei $|p|$ die Anzahl der Symbole ist).

Die bedingte Zuweisung in Z.5-6r kann auch hier wieder vernachlässigt werden, da diese nach der ersten Beobachtung eine maximale Laufzeit von $\mathcal{O}(|T|)$ benötigt, welche im kleiner ist als die Laufzeit des **else**-Zweigs.

Im **else**-Zweig findet die zweite geschachtelte **for**-Schleife in Z.8r statt. Die darin befindlichen **if**-Bedingungen in Z.9r und Z.11r werden konstanter Zeit bearbeitet, folglich bleiben nur noch die Zuweisungen in Z.10r und Z.12r wie auch die **if**-Bedingung in Z.13r, welche nach der Beobachtung alle in $\mathcal{O}(|T|)$ berechnet werden. Da diese nicht verschachtelt sind, ergibt sich insgesamt eine Laufzeit von $\mathcal{O}(|N| \cdot |T| \cdot |P| \cdot \sup_{p \in P} |p| \cdot |T|) = \mathcal{O}(|N| \cdot |T|^2 \cdot |P| \cdot \sup_{p \in P} |p|)$.

Implementierung Die konkrete Implementierung nutzt dann die bereits genannten Hilfsfunktionen. So wird beispielsweise für die Vereinigung in den Zeilen 2,6,10 und 12 die **append**-Funktion des *MapSets* (siehe 5.1.2.1) genutzt. Diese geben einen boolschen Wahrheitswert zurück, welcher angibt, ob die **append**-Funktion tatsächlich ein Element eingefügt hat. Dabei ist zu beachten, dass die Vereinigung zweier Mengen keine Änderung liefert, wenn eine Menge in der anderen Menge enthalten ist. Dieser Wert wird dann an die **do-while**-Schleife weitergegeben, wodurch anschließend für die Prüfung, ob es eine Änderung gegeben hat, nur noch dieser Wert ausgelesen werden muss.

5.2.2 Follow

Algorithm 4: Follow (vgl. [1] S. 222)

Input: Nichtterminalsymbole N ,
Startsymbol S , Produktionen P ,
First-Mengen $FIRST$

Output: Map(Symbol, Follow-Menge)
 F

```

1  $F[S] \leftarrow \{\$ \}$ 
2 do
3   foreach Symbol  $X \in N$  do
4     foreach Produktion
5        $p = A \rightarrow uXv \in P$  do
6         if  $v \neq \epsilon$  then
7            $F[X] \leftarrow F[X] \cup$ 
8              $FIRST[v] \setminus \{\epsilon\}$ 
9           if  $\epsilon \in FIRST[v]$  then
10             $F[X] \leftarrow F[X] \cup F[A]$ 
11         else if  $v = \epsilon$  then
12            $F[X] \leftarrow F[X] \cup F[A]$ 
11 while  $F$  geändert
12 return  $F$ 

```

Algorithm 5: Anpassung von Follow

Input: Nichtterminalsymbole N ,
Startsymbol S , Produktionen P ,
First-Mengen $FIRST$

Output: Map(Symbol, Follow-Menge)
 F

```

1  $F[S] \leftarrow \{\$ \}$ 
2 do
3   foreach Produktion
4      $p = A \rightarrow w \in P$  do
5     foreach Symbol  $X \in w$  do
6       if  $X \in N$  then
7         if  $w = uXY_1 \dots Y_n$  then
8            $F[X] \leftarrow F[X] \cup$ 
9              $FIRST[Y_1 \dots Y_n] \setminus \{\epsilon\}$ 
10          for  $i \leftarrow 1, \dots, n$  do
11            if  $\epsilon \notin FIRST[Y_i]$ 
12              then
13                break
14            else if  $i = n$  then
15               $F[X] \leftarrow$ 
16                 $F[X] \cup F[A]$ 
13          else if  $w = uX$  then
14             $F[X] \leftarrow F[X] \cup F[A]$ 
15 while  $F$  geändert
16 return  $F$ 

```

Korrektheit Offensichtlich sind Z.11 und Z.1r identisch, und die **do-while**-Schleife auch, wenn die Inneren der Schleifen (Z.3-10l und Z.3-14r) übereinstimmen. Damit bleibt nur diese Äquivalenz zu zeigen.

Erneut gilt hier, dass nicht über Produktionen mit bestimmten Eigenschaften iteriert werden kann, wie es in Z.4l der Fall ist, wo alle Produktionen gesucht werden, welche das Nichtterminal X aus der Schleife in Z.3l enthalten. Aus diesem Grund, werden alle Produktionen der Form $A \rightarrow w$ betrachtet und über die **if**-Anweisung in Z.5r prüft mit $X \in w$, ob das Symbol in der rechten Seite enthalten ist.

In Z.6r wird dann überprüft, ob nach X weitere Symbole kommen. Dies entspricht der

Überprüfung $v \neq \epsilon$ in Z.5l. Analog wird in Z.13r geprüft, ob nach X die Produktion beendet ist, wobei es sich um die inhaltlich selbe Überprüfung wie in Z.9l handelt. Da außerdem Z.7r identisch zu Z.6l ist und Z.14r identisch zu Z.10l ist, bleibt nur noch die Äquivalenz von Z.7-8l und Z.8-12r zu zeigen.

In Z.7-8l wird die Follow-Menge des Nichtterminalsymbols der derzeitigen Produktionsregel in der First-Menge des Wortes v aller Symbole nach X hinzugefügt, wenn ϵ enthalten ist. Dies ist nach Bemerkung 4.1.4.2 nur dann der Fall, wenn alle Symbole von v ein ϵ in dessen First-Mengen enthalten. Damit muss in Z.8-11r genau diese Überprüfung stattfinden, da Z.10l und Z.12r identisch sind.

Das zugrundeliegende Prinzip ist das selbe wie bei dem First-Algorithmus in Z.8-14r. Es werden in der **for**-Schleife in Z.8r alle Symbole einzeln durchgegangen. Wenn ein Symbol in dessen First-Menge das ϵ nicht enthält, wird die Überprüfung beendet und die Zuweisung in Z.12r wird nicht ausgeführt. Nur wenn die First-Mengen aller Symbole das ϵ enthalten, wird die Anweisung ausgeführt.

Damit folgt die Äquivalenz von Z.7-8l und Z.8-12r und damit die Äquivalenz der **do-while**-Schleifen. Insgesamt ist also die Korrektheit des Algorithmus gezeigt.

Die Terminierung der Algorithmen folgt wieder aus der Äquivalenz der Algorithmen und der Endlichkeit der Produktionsmenge P , jedoch wird hier auch die Endlichkeit aller Produktionsregeln benötigt, welche im Rahmen dieses Programms offensichtlich ist.

Laufzeit Auch die Analyse der Follow-Berechnung benötigt die Beobachtung, dass die **append**-Funktionen als Umsetzung der Vereinigung eine Laufzeit von $\mathcal{O}(|T|)$ pro hinzuzufügendem Symbol hat. Entsprechend benötigt auch die **if**-Bedingung in Z.9r eine Laufzeit von $\mathcal{O}(|T|)$. Im Follow-Algorithmus befindet sich jedoch eine weitere Abfrage, die betrachtet werden muss. Dies ist die **if**-Bedingung in Z.5r, welche durch Anwendung der **includes**-Funktion des Alphabets der Nichtterminalsymbole 5.1.1.1 in $\mathcal{O}(|N|)$ prüft, ob das Symbol X ein Nichtterminal ist.

Analog zur Laufzeitanalyse der First-Mengen-Berechnung wird die Anzahl der Schleifendurchläufe der **do-while**-Schleife mit $|N| \cdot |T|$ abgeschätzt und auch hier enthält die **do-while**-Schleife weitere geschachtelte Schleifen.

Die erste **for-each**-Schleife in Z.3-14r iteriert über alle Produktionsregeln und damit $|P|$ mal. Bei der **for-each**-Schleife in Z.4-14r muss beachtet werden, dass die **if**-Bedingung in Z.5r nach der Beobachtung in $\mathcal{O}(|N|)$ erfolgt, sodass diese zwei Zeilen zusammen maximal $|P| \cdot |N|$ Berechnungen durchführen.

Im Gegensatz dazu hat die **if**-Bedingung in Z.6r zusammen mit der **else-if**-Bedingung in Z.13r keine Auswirkung auf die Worst Case-Laufzeit. Nach der Prüfung, ob das Symbol X das letzte Symbol der Produktionsregel ist, wird im Falle des letzten Symbol die First-Berechnung in Z.7r und die **for**-Schleife in Z.8-12r übersprungen. Ohne **if**-Bedingung würde die innere Schleife in Z.8-12r $|p|$ -mal betreten werden und mit der **if**-Bedingung $|p - 1|$ -mal.

Dies macht für die Worst Case-Laufzeit in $\mathcal{O}$ -Notation jedoch keinen Unterschied.

Die bedingte Vereinigung in Z.13-14r kann wiederum im Vergleich zu Z.6-12r vernachlässigt werden. Folglich hängt die Gesamtlaufzeit entweder von der Berechnung der First-Menge in Z.7r ab oder entspricht der Anzahl an Iterationen der inneren Schleife (Z.8-12r).

Für die Funktion, welche die First-Berechnung in Z.7r übernimmt, gilt, dass die Umsetzung nach Bemerkung 4.1.4.2 linear rekursiv mit nur einem konstanten Zugriff pro Aufruf umgesetzt wird, also in $\mathcal{O}(\sup_{p \in P} |p| \cdot |T|)$ umgesetzt werden kann, dabei fällt die $|T|$ wegen der Überprüfung auf vorhandene Symbole an. Des Weiteren muss die berechnete Menge anschließend mit $F[X]$ vereinigt werden. Da die $FIRST[Y_1 \dots Y_n]$ maximal alle Terminalsymbole enthält kann eine Gesamtlaufzeit von $\mathcal{O}(|T|^2 \cdot \sup_{p \in P} |p|)$ für diese Zeile anfallen.

Für die Schleife gilt, dass das n mit jedem Durchlauf der Schleife in Z.4r um eins verringert wird. Daher ergibt sich pro Produktionsregel p der **for-each**-Schleife in Z.3r die folgende Anzahl an Iterationen der inneren Schleife:

$$\sum_{i=1}^{|p|} |p| - i = \sum_{i=1}^{|p|} i = \frac{|p|(|p| - 1)}{2} = \frac{|p|^2 - |p|}{2}$$

und für die längste Produktionsregel:

$$\frac{(\sup_{p \in P} |p|)^2 - \sup_{p \in P} |p|}{2}$$

Innerhalb dieser Schleife fällt sowohl im **if**-Fall als auch im **else**-Fall nach der anfänglichen Beobachtung eine Laufzeit von $\mathcal{O}(|T|)$ an, sodass die Berechnung in Z.8-12r insgesamt eine Laufzeit von $\mathcal{O}((\sup_{p \in P} |p|)^2 - \sup_{p \in P} |p| \cdot |T|) = \mathcal{O}((\sup_{p \in P} |p|)^2 \cdot |T|)$ hat.

Insgesamt wird diese Schleife in Z.4-14 der obigen Begründung $|N| \cdot |T| \cdot |P|$ -mal betreten, sodass die Laufzeit $\mathcal{O}(|N| \cdot |T|^3 \cdot |P| \cdot \sup_{p \in P} |p|)$ oder $\mathcal{O}(|N| \cdot |T|^2 \cdot |P| \cdot (\sup_{p \in P} |p|)^2)$ beträgt. Die genaue Laufzeitsklasse hängt dabei von der eingegebenen Grammatik ab. Genauer ist es abhängig davon, ob die Menge an Terminalsymbolen oder die größte Produktionsregel größer ist.

Implementierung Bei der Implementierung ist auch hier zu beachten, dass für die Vereinigung die **append**-Funktion genutzt wird, welche die Angabe, ob eine Änderung vorgenommen wurde, an die **while** F geändert Schleife übergibt.

Um hier die Abfrage $X \in N$ in Z.5r durchzuführen, wird die **has**-Funktion des Alphabets (siehe 5.1.1.1) der Nichtterminale genutzt.

Weiterhin benötigt Z.7r eine eigene Funktion zur Berechnung der First-Mengen nach dem Prinzip der Bemerkung 4.1.4.2, da die First-Mengen in 5.2.1 nur für einzelne Symbole berechnet werden. Diese Funktion soll hier aber nicht weiter behandelt werden.

5.2.3 Hülle

Algorithm 6: Hülle (vgl. [1] S. 261)

Input: Kollektion K , Produktionen P ,
First-Mengen FIRST
Output: Kollektion K'

```
1 do
2   foreach Element
3      $e = [A \rightarrow u.Xv; M]$  in  $K$  do
4       foreach Produktion  $p = X \rightarrow w$ 
5         in  $P$  do
6            $K \leftarrow K \cup \{[X \rightarrow .w;$ 
7             FIRST[ $vM$ ] ]}
8 while  $K$  geändert;
9 return  $K$ 
```

Algorithm 7: Anpassung von Hülle

Input: Kollektion K , Produktionen P ,
Nichtterminalsymbole N ,
First-Mengen FIRST
Output: Kollektion K'

```
1 do
2   foreach Element
3      $e = [A \rightarrow u.Xv; M]$  in  $K$  do
4       if  $X \in N$  then
5          $F \leftarrow FIRST[vM]$ 
6         foreach Produktion
7            $p = X \rightarrow w$  in  $P$  do
8            $K \leftarrow K \cup \{[X \rightarrow .w; F]\}$ 
9 while  $K$  geändert
10 return  $K$ 
```

Korrektheit In der Hüllen-Berechnung gibt es nur wenige Veränderungen, die gemacht werden müssen. Die erste Änderung ist die **if**-Abfrage in Z.3r, welche auf Grund der Umsetzung der Produktionsregeln als Map von Nichtterminalsymbolen zu den Produktionen mit $X \in N$ prüfen muss, ob das Folgesymbol des derzeitigen Elements ein Nichtterminalsymbol ist, da sonst die Produktionen von X undefiniert wären.

Die zweite Änderung ist nicht zwingend notwendig, wurde jedoch gemacht, um den Algorithmus effizienter zu gestalten. Da die First-Mengen nur für Symbole gespeichert werden, muss $FIRST[vM]$ zur Laufzeit des Algorithmus berechnet werden, daher wurde die Berechnung in Z.4l aus der **for-each**-Schleife in Z.3l beziehungsweise Z.5r gezogen. Dies ist möglich, da die Berechnung unabhängig von der Produktionen der Schleife ist. Somit kann die mehrfache Berechnung der gleichen Menge verhindert werden.

Da die Algorithmen sonst identisch sind, ist auch hier die Korrektheit gezeigt.

Die Korrektheit reicht in diesem Fall, um zu zeigen, dass der Algorithmus terminiert. Es muss aufgrund der nahezu identischen Algorithmen keine weitere Argumentation erfolgen.

Laufzeit Um die Laufzeitanalyse des Hüllen-Algorithmus durchzuführen, müssen auch an dieser Stelle zwei Beobachtungen über den Algorithmus festgehalten werden.

Die erste Beobachtung ist die selbe wie bei der Laufzeitanalyse der First- und Follow-Algorithmen. Die Überprüfung in Z.3r findet in $\mathcal{O}(|N|)$ statt. In Z.6r wird die Vereinigung

in $\mathcal{O}(|K|)$ umgesetzt.

Die zweite Beobachtung ist, dass in einer Kollektion theoretisch alle möglichen Elemente einmal vorkommen können. Das sind für jede Produktionsregel p genau $|p| + 1$ Stück. Die maximale Anzahl an Elementen kann daher mit $|P| \cdot (\sup_{p \in P} |p|)$ abgeschätzt werden, dabei fällt das „+1“ durch die spätere Nutzung der $\mathcal{O}$ -Notation weg. Insbesondere gilt damit, dass die Vereinigung in $\mathcal{O}(|P| \cdot \sup_{p \in P} |p|)$ umgesetzt wird.

Die dritte Beobachtung ist, dass die Berechnung der First-Menge in Z.4r nach Bemerkung 4.1.4.2 als mehrfacher Aufruf von der First-Funktion für Wörter berechnet wird. Genauer gesagt muss die Funktion $|M|$ -mal angewendet werden. Da nach Definition 4.4.1.2 $M \subseteq T$ gilt, ist $|M|$ im Worst Case gleich $|T|$. Außerdem wissen wir nach 5.2.2, dass die First-Funktion eine Laufzeit von $\mathcal{O}(\sup_{p \in P} |p|)$ hat. Somit hat die Funktion in Z.4r eine Laufzeit von $\mathcal{O}(|T| \cdot \sup_{p \in P} |p|)$.

Für die Gesamtlaufzeit muss zunächst die maximale Anzahl an Durchläufen der **do-while**-Schleife berechnet werden. Da die einzige Änderung in Z.6r stattfindet und diese Zeile für jede Produktionsregel maximal einmal eine Änderung bewirkt, kann die **do-while**-Schleife nur $|P|$ -mal durchlaufen werden. Anschließend wird die **for-each**-Schleife nach der Abschätzung der zweiten Beobachtung $|P| \cdot \sup_{p \in P} |p|$ -mal durchlaufen. Auch die **if**-Bedingung wurde bereits in der Beobachtung behandelt und wird mit $|N|$ abgeschätzt. Für Z.4-6r muss wiederum zwischen der Laufzeit der First-Berechnung in Z.4r und die Laufzeit der gesamten **for-each**-Schleife in Z.5-6r unterschieden werden.

Die First-Berechnung wurde bereits behandelt. Für die Schleife gilt, dass die Laufzeit $\mathcal{O}(|P| \cdot |K|) = \mathcal{O}(|P|^2 \cdot (\sup_{p \in P} |p|))$ ist, da die Schleife für alle Produktionen in P die Vereinigung aus der Beobachtung anwendet.

Insgesamt folgt daher eine Laufzeit von

$$\mathcal{O}(|P| \cdot |P| \cdot \sup_{p \in P} |p| \cdot |N| \cdot |T| \cdot \sup_{p \in P} |p|) = \mathcal{O}(|P|^2 \cdot (\sup_{p \in P} |p|)^2 \cdot |N| \cdot |T|)$$

oder

$$\mathcal{O}(|P| \cdot |P| \cdot \sup_{p \in P} |p| \cdot |N| \cdot |P|^2 \cdot \sup_{p \in P} |p|) = \mathcal{O}(|P|^4 \cdot (\sup_{p \in P} |p|)^2 \cdot |N|)$$

als Abschätzung gewählt werden kann, abhängig von der eingegebenen Grammatik.

Zu beachten ist, dass durch geeignete Methoden möglicherweise auch eine kleinere Laufzeitklasse berechnet werden kann. Es wurden für die Berechnung schließlich Annahmen getroffen, welche in der Form gleichzeitig nicht auftreten können. Diese liegen darin, dass die Kollektion im Worst Case $|P| \cdot (\sup_{p \in P} |p| + 1)$ Elemente enthält und gleichzeitig $|P|$ Änderungen möglich sind. Dies ist jedoch nicht möglich, wenn die Kollektion bereits alle Elemente enthält, jedoch würde eine genauere Berechnung der Laufzeit den Rahmen dieser Analyse überschreiten.

Implementierung Auch hier findet eine Iteration statt, solange Änderungen an der Kollektion vorgenommen wurden. Diese **do-while**-Schleife wird nach demselben Konzept der modifizierten **append**-Funktion umgesetzt.

Die eigentliche Berechnung der Hülle beginnt mit der Iteration über die Elemente des

Zustandes in Z.2r. Dabei wird für den Zugriff auf X die Hilfsfunktion **followingSymbol**-Funktion verwendet, welche das Folgesymbol zurückgibt.

Die anschließende Berechnung der First-Menge bezieht sich hier auf die First-Mengen einer Wörtermenge aus Bemerkung 4.1.4.2 und benötigt erneut eine entsprechende Funktion, welche jedoch im Rahmen dieser Arbeit nicht weiter beleuchtet wird.

5.2.4 Sprung

Algorithm 8: Sprung (vgl. [1] S. 261)

Input: Kollektion K , Symbol X

Output: Kollektion K'

```

1  $S \leftarrow \{\}$ 
2 foreach Element  $e = [A \rightarrow u.Xv; M]$  in
    $K$  do
3    $S \leftarrow S \cup \{[A \rightarrow uX.v; M]\}$ 
4 return HÜLLE( $S$ )

```

Algorithm 9: Anpassung von Sprung

Input: Kollektion K , Symbol X

Output: Kollektion K'

```

1  $S \leftarrow \{\}$ 
2 foreach Element  $e = [A \rightarrow u.Bv; M]$  in
    $K$  do
3   if  $B = X$  then
4      $S \leftarrow S \cup \{[A \rightarrow uX.v; M]\}$ 
5 return HÜLLE( $S$ )

```

Korrektheit Der Sprung benötigt auf Grund des kurzen Algorithmus lediglich eine Änderung. Diese wird erneut benötigt, da auf Grund der Umsetzung der Datenstrukturen eine gezielte Iteration über die Elemente mit X als Folgesymbol nicht möglich ist. Stattdessen muss ohne weiteres Differenzieren über sämtliche Elemente iteriert werden und zu jedem Element mit der **if**-Abfrage in Z.3r geprüft werden, ob das Folgesymbol B des Elements dem Symbol X entspricht, über dem der Sprung stattfindet.

Da nach der Begründung in 5.2.3 eine Kollektion maximal $|P| \cdot \sup_{p \in P} |p|$ Elemente enthält, durchläuft die **for-each**-Schleife in Z.2r eine endliche Anzahl von Iterationen. Des Weiteren ist die Terminierung der Hüllen-Berechnung bereits gezeigt. Folglich terminiert der Algorithmus.

Laufzeit In dem Sprung wird definitionsgemäß auch die Hüllen-Berechnung ausführt. Da sonst nur die **for-each**-Schleife in Z.2r berechnet, aber eine Äquivalente Schleife bereits in der Hüllen-Berechnung enthalten ist und damit vernachlässigt werden kann, liegt der Sprung in derselben Laufzeitklasse $\mathcal{O}(|P|^2 \cdot (\sup_{p \in P} |p|)^2 \cdot |N| \cdot |T|)$ oder $\mathcal{O}(|P|^4 \cdot (\sup_{p \in P} |p|)^2 \cdot |N|)$ (siehe 5.2.3), abhängig davon, welche Laufzeit zu der gegebenen Grammatik größer ist.

Implementierung Zwar gibt es hier keine **do-while**-Schleife, es wird aber trotzdem die **append**-Funktion genutzt, um die Mengeneigenschaften zu wahren.

Außerdem wird auch hier die **followingSymbol**-Funktion verwendet, um auf das Folgesymbol B zuzugreifen. In der Implementierung muss vorher jedoch mit der **isFinished**-Funktion kontrolliert werden, ob es überhaupt ein Folgesymbol gibt.

Wenn dies der Fall ist, kann die Verschiebung des Punktes durch die Umsetzung als Index leicht implementiert werden, indem der Index inkrementiert wird.

5.2.5 Zustandsberechnung

Algorithm 10: Zustandsberechnung
(vgl. [1] S. 261)

Input: Startelement der
Startproduktion s ,
Terminalsymbole T ,
Nichtterminalsymbole N

Output: Zustände Z

```

1  $Z \leftarrow \{\text{HÜLLE}(\{[s; \$]\})\}$ 
2 do
3   foreach Kollektion  $K$  in  $Z$  do
4     foreach Symbol  $X$  in  $N \cup T$  do
5       if  $\text{SPRUNG}(K, X) \neq \emptyset$  then
6          $Z \leftarrow Z \cup \{\text{SPRUNG}(K, X)\}$ 
7 while  $Z$  geändert
8 return  $Z$ 

```

Algorithm 11: Anpassung der Zustandsberechnung

Input: Startelement der
Startproduktion s

Output: Zustände Z

```

1  $Z \leftarrow \{\text{HÜLLE}(\{[s; \$]\})\}$ 
2 do
3   foreach Kollektion  $K$  in  $Z$  do
4     foreach Element  $e$  in  $K$  do
5       if  $e = [A \rightarrow u.Xv; M]$  then
6          $J \leftarrow \text{SPRUNG}(K, X)$ 
7          $Z \leftarrow Z \cup \{J\}$ 
8          $K.\text{jumps}[X] \leftarrow J.\text{index}$ 
9       else if  $e = [A \rightarrow w.; M]$  then
10        foreach Symbol  $Y$  in  $M$ 
11          do
12             $K.\text{reductions}[Y] \leftarrow e.\text{ruleIndex}$ 
12 while  $Z$  geändert
13 return  $Z$ 

```

Korrektheit Bei der Zustandsberechnung werden in der Anpassung zeitgleich die Berechnungen der Shift- und Reduce-Aktionen durchgeführt. Dies liegt daran, dass auf Grund der Datenstruktur diese innerhalb des jeweiligen Zustandes gespeichert werden (siehe Kapitel 5.1.3.3).

Dies hat zur Folge, dass der rechte Algorithmus größere Anpassungen benötigt, um die benötigten zusätzlichen Daten zu berechnen. Für die Korrektheit der strikten Zustandsberechnung werden daher Z.8-11r erstmal nicht beachtet.⁵

⁵Die Korrektheit der Berechnung der Shift- und Reduce-Aktionen werden in dem Abschnitt 5.2.5.1 behandelt.

Erneut gilt hier, dass nur die Äquivalenz der inneren **do-while**-Schleifen (Z.3-6l und Z.3-7r) zu zeigen ist, da die **do-while**-Schleifen und die ersten Zeilen identisch sind. Da auch die **for-each**-Schleifen in Z.3l und Z.3r identisch sind, muss schließlich nur gezeigt werden, dass Z.4-6l und Z.4-7r die selbe Berechnung durchführen.

Um diese Gleichheit zu zeigen, ist es notwendig die Berechnung nachzuvollziehen. In dem linken Algorithmus wird zu jedem Symbol $X \in N \cup T$ (Z.4l) überprüft, ob der Sprung über dieses Symbol nicht leer ist. Dies ist genau dann der Fall, wenn der Sprung existiert.

Die Existenz kann dabei auch wie folgt ausgedrückt werden. Wenn die Kollektion K ein Element enthält, welches das Symbol X als Folgesymbol enthält, dann existiert der Sprung(K, X). Dies ist leicht in der Definition des Sprungs: Hülle($\{[N \rightarrow aX.b; M] \mid \exists [N \rightarrow a.Xb; M] \in K\}$) in 4.4.2.1 ersichtlich. Die Hülle einer Kollektion, ist definitionsgemäß schließlich genau dann leer, wenn die Kollektion leer ist. Damit folgt, dass die Menge $\{[N \rightarrow aX.b; M] \mid \exists [N \rightarrow a.Xb; M] \in K\}$ nichtleer sein muss und dies ist genau der Fall, wenn gilt $\exists [N \rightarrow a.Xb; M] \in K$.

Insgesamt werden nach der Schleife in Z.4l und der **if**-Bedingung in Z.5l daher nur Sprünge betrachtet, die in der Kollektion K überhaupt möglich sind.

Damit kann die Gleichheit gezeigt werden, da in dem rechten Algorithmus in der Schleife in Z.4r über alle Elemente der Kollektion iteriert wird, wohingegen in der **if**-Bedingung in Z.5r nur die Elemente erlaubt sind, welche ein Folgesymbol besitzen. Offensichtlich sind die Folgesymbole genau die Symbole über denen Sprünge in der Kollektion möglich sind, so dass sowohl Z.4-5l als auch Z.4-5r genau diejenigen Symbole erlauben, über denen Sprünge möglich sind.

Da Z.6l und Z.6-7r genau die selbe Berechnung durchführen, mit dem einzigen Unterschied, dass in dem rechten Algorithmus das Ergebnis zwischengespeichert wird, ist die Korrektheit der Zustandsberechnung gezeigt.

Die restlichen Berechnungen verändern ausschließlich Attribute der einzelnen Kollektionen und beeinflussen damit die Korrektheit der Zustandsberechnung nicht.

Der Algorithmus terminiert auch hier, da analog zur Sprung-Berechnung 5.2.4 die **for-each**-Schleife in Z.4r maximal $|P| \cdot \sup_{p \in P} |p|$ Elemente durchläuft und der Algorithmus sonst im Bezug auf die Schleifen identisch ist und außerdem die Terminierung der Sprung-Berechnung in Z.6r bereits gezeigt wurde.

Laufzeit Die Laufzeitanalyse der gesamten Berechnung kann auf Grund von fehlenden Abschätzungsmöglichkeiten für die Menge der Zustände und der darin enthaltenen Menge, analog zur Laufzeitanalyse der Hüllen-Berechnung, nur approximativ erfolgen.

Bereits die Menge aller Zustände ist $Z \subseteq 2^E$ eine Teilmenge der Potenzmenge der Menge aller Elemente E und muss im Worst Case nach der Begründung in 5.2.3 mit $2^{|P| \cdot (\sup_{p \in P} |p|)}$ Elementen abgeschätzt werden.

In dem Algorithmus, gibt es zwei geschachtelte Schleifen, welche über diese Menge iterieren.

Die **do-while**-Schleife in Z.2-12r kann pro Änderung einer Kollektion durchlaufen werden und die **for-each**-Schleife in Z.3-11r durchläuft explizit alle Kollektionen.

Dadurch ergeben sich für diese Schleifen eine Abschätzung der Iterationen als:

$$2^{|P| \cdot (\sup_{p \in P} |p|)} \cdot 2^{|P| \cdot (\sup_{p \in P} |p|)} = (2^{|P| \cdot (\sup_{p \in P} |p|)})^2 = 4^{|P| \cdot (\sup_{p \in P} |p|)}$$

Außerdem hat die innere Schleife in Z.4r mit derselben Begründung $|P| \cdot (\sup_{p \in P} |p|)$ Iterationen. Diese Iterationen müssen jedoch aufgeteilt werden, da Elemente, welche für die **if**-Bedingung in der Sprung-Berechnung genutzt werden und Elemente welche die **else-if**-Bedingung erfüllen, eine weitere **for-each**-Schleife betreten. Das sind nach dieser Bedingung genau die Endelemente. Folglich betreten maximal $|P|$ Elemente die Schleife, welche selbst $|M|$ Iterationen durchlaufen. Nach der Begründung in 5.2.3 wird dies mit $|T|$ abgeschätzt.

Zusammen hat der **else**-Zweig damit maximal $|P| \cdot |T|$ Zuordnungen von konstanter Zeit, welche im Vergleich zu der Laufzeit der Sprung-Funktion vernachlässigt werden kann.

Auch die **append**-Funktion in Z.7r und Zuordnung in Z.8r fallen in der Laufzeitanalyse durch den Vergleich zu der Sprung-Funktion weg.

Insgesamt liegt der Algorithmus durch die Laufzeit der Sprungfunktion in der Laufzeitklasse von

$$\mathcal{O}(4^{|P| \cdot (\sup_{p \in P} |p|)} \cdot |P| \cdot (\sup_{p \in P} |p|) \cdot |P|^2 \cdot (\sup_{p \in P} |p|)^2 \cdot |N| \cdot |T|) = \mathcal{O}(4^{|P| \cdot (\sup_{p \in P} |p|)} \cdot |P|^3 \cdot (\sup_{p \in P} |p|)^3 \cdot |T|)$$

oder

$$\mathcal{O}(4^{|P| \cdot (\sup_{p \in P} |p|)} \cdot |P| \cdot (\sup_{p \in P} |p|) \cdot |P|^4 \cdot (\sup_{p \in P} |p|)^2 \cdot |N|) = \mathcal{O}(4^{|P| \cdot (\sup_{p \in P} |p|)} \cdot |P|^5 \cdot (\sup_{p \in P} |p|)^3 \cdot |N|)$$

abhängig davon, welche Abschätzung größer ist.

Implementierung Mithilfe der Hülle- und Sprung-Funktionen (siehe 5.2.3, 5.2.4) kann schließlich die Zustandsberechnung erfolgen.

In der Initialisierung wird die Menge der Zustände Z als einelementige Menge aus der Hülle der Startproduktion gesetzt. In der Umsetzung wird dafür eine neue Instanz eines Elements mit der Startproduktion und dem Punkt an dem Index 0 erstellt und mithilfe der **Hülle**-Funktion der Startzustand berechnet.

Wie in den meisten anderen Algorithmen nutzt die **do-while**-Schleife in der Implementierung, die **append**-Funktion, um die Änderungen zu verfolgen.

Innerhalb der Schleife gibt es für die **if**- und **else-if**-Bedingungen in Z.5r und Z.9r die **isFinished**-Funktion, welche die beiden überprüften, ob es ein Folgesymbol gibt durchführen kann. Wenn dieses in Z.6-8r gebraucht wird, kann auch hier die **followingSymbol**-Funktion genutzt werden.

Mit dem Folgesymbol wird der neue Zustand mit der **Sprung**-Funktion berechnet. Der daraus resultierende Zustand lässt sich durch die entsprechende Funktion anschließend zu der Menge der Zustände hinzufügen .

5.2.5.1 Shift- und Reduce-Aktionen

Korrektheit Mit Blick auf die Sprünge werden diese in dem rechten Algorithmus innerhalb der inneren **for-each**-Schleife in Z.4r berechnet. Da die entstehenden Kollektionen der Sprünge zwangsläufig einmal als Zustände in die Zustandsmenge aufgenommen werden, woraufhin diese in der Menge der Sprünge gespeichert werden müssen, wird die berechnete Kollektion in Z.6r in J zwischengespeichert. Dadurch, dass die Aufnahme der Kollektion in die Menge der Sprünge in der vorherigen Zeile geschieht, kann auch an dieser Stelle der entsprechende Index, mit dem die Kollektionen eindeutig gekennzeichnet sind, direkt in Z.8r genutzt werden. Die Korrektheit ist hier bereits mit der Zustandsberechnung gezeigt worden, da für die Shift-Aktion nur die Angaben gespeichert werden, wie der Zustand J erreicht wurde und diese nach 4.4.1.3 richtig sind.

Die Reduce-Aktionen werden in dem rechten Algorithmus berechnet, indem in Z.9r geprüft wird, ob es sich bei dem derzeitigen Element um ein Endelement handelt. Wenn dies der Fall ist, kann nach der Begründung in Kapitel 4.4.1.3 zu jedem Symbol in der Vorschau-Menge M die Reduktion hinzugefügt werden. Die Korrektheit folgt hierbei aus der Begründung der Theorie.

Implementierung Da die Shift- und Reduce-Aktionen einen Wechsel der Zustände bedeuten, können diese zeitgleich zum Erstellen der Zustände berechnet und gespeichert werden. In der Implementierung werden Zustände und Shift-Aktionen schließlich jedoch nicht getrennt, sondern zusammen durch die **addStateAndJump**-Funktion hinzugefügt. Diese Funktion nutzt in diesem Zusammenhang die zugrundeliegende **append**- und **addShift**-Funktion.

Zu beachten ist hierbei, dass nur der Index des jeweiligen Zustandes zu den Shift-Aktionen hinzugefügt wird. Dieser wird dabei von der **append**-Funktion der Zustands-Menge als Int-Wert zurückgegeben.⁶ Um noch die Anforderung zu erfüllen, dass eine Änderung der Zustands-Menge angezeigt wird, erfolgt eine Rückgabe des Index nurunter der Voraussetzung dass eine entsprechende Änderung auch festzustellen ist, ansonsten werden negative Int-Werte zurückgeben.

Auch im Falle der reduce-Aktionen wird nur der Index der Regel gespeichert. Dieser kann hier jedoch aus der Datenstruktur gewonnen werden, da diese dort gespeichert ist (siehe 5.1.3.4).

⁶Dies ist eine Ausnahme, da hier der Index benötigt wurde. Alle anderen **append**-Funktionen liefern einen booleschen Wert.

5.3 Funktionsweise der Interaktion

In Anbetracht des Ziels ein Lernprogramm zu erstellen, ist die Interaktion mit dem Programm von großer Wichtigkeit. Genauer beinhaltet die Interaktion zwei wichtige Aspekte, dessen Funktionsweise im Folgenden genauer beleuchtet wird. Von besonderer Relevanz sind hier neben der Abfrage der verschiedenen Schritte des LR(1)-Verfahrens mit der einhergehenden Korrektur der Eingaben auch die Angabe einiger Warnungen und Fehlermeldungen für den Fall, dass die Grammatik fehlerhaft eingegeben worden ist. Aspekte die diesbezüglich nicht erläutert werden sind die didaktischen Designentscheidungen oder die genauen Funktionalitäten der graphischen Nutzeroberfläche.

5.3.1 Abfrage- und Korrekturfunktion

Bevor die Berechnung der Schritte des LR(1)-Verfahrens erfolgt, muss die Eingabeverarbeitung stattfinden. Dafür werden auf der Website nur Felder zur Eingabe von Produktionsregeln zur Verfügung gestellt. Diese reichen aus, wenn die Eingabe wie folgt verarbeitet wird.

1. Die Produktionsregeln werden exakt übernommen.
2. Die Nichtterminale werden aus den linken Seiten der Produktionsregeln übernommen.
3. Die Nichtterminale werden aus den übrigen Symbolen der rechten Seiten der Produktionsregeln übernommen.
4. Das Startsymbol wird zusammen mit der Startproduktion, welche im LR(1)-Verfahren hinzugefügt wird (siehe [4.4.2.2](#)) von dem Programm vorgegeben.

Nachdem die Eingabeverarbeitung vorgenommen wurde, welche der Nutzer durch eine Bestätigung der Eingabe einleitet, beginnt die Berechnung des LR(1)-Parsers. Dabei wird initial bereits die gesamte Berechnung durchgeführt, damit potenzielle Fehler in der Grammatik erkannt werden können. Die jeweiligen Zwischenschritte, zu denen beispielsweise die Berechnung der First-Mengen gehört, werden dabei in den vorhandenen Datenstrukturen gespeichert, um für die Abfrage genutzt werden zu können.

Nachdem die Berechnung durchgeführt wurde, erscheinen weitere Eingabefelder mit Korrekturfunktion für die First-Mengen aller Symbole. Die eingegebenen Mengen werden anschließend bei erneuter Bestätigung des Nutzers durch die **equals**-Operation mit den tatsächlich berechneten First-Mengen verglichen. Da die Funktion auf gegenseitige Inklusion prüft, ist die Reihenfolge der Mengenelemente nicht relevant. Vor diesem Hintergrund hat der Nutzer darüberhinaus auch die Möglichkeit auf die Abfrage mit Korrektur zu verzichten und sich nur die Lösungen anzeigen zu lassen.

Nach Eingabe aller First-Mengen oder dem alternativen Anzeigen der Lösungen, werden

die berechneten Zustände in tabellarischer Form angegeben, jedoch nicht abgefragt. Begründet wird dies dadurch, dass bereits kleine Grammatiken zu großen Zustandsmengen führen können. Da außerdem die Zustände eine zu komplexe Struktur besitzen (siehe 5.5), kann die Abfrage der Zustände nicht umgesetzt werden.

Aus diesem Grund werden die Zustände lediglich in einer übersichtlichen Tabelle ausgegeben. Dabei werden die gespeicherten Informationen zur Berechnungsweise des Zustandes und die Zustände als Menge von Elementen angegeben. Diese werden mithilfe der bereits implementierten **toString**-Methoden nacheinander ausgegeben.

Da hier keine Eingabe des Nutzers gefragt ist, wird zeitgleich die Visualisierung des berechneten LR(1)-Parsers als interaktive Parsertabelle ausgegeben. Diese ist zu Beginn bis auf die Beschriftung leer, lediglich die Angabe des Zustandes in den Zeilen und die Symbole in den Spalten sind enthalten. Somit kann der Nutzer die Berechnung der Shift- und Reduce-Aktionen selbst durchführen, mit der Option, die eigene Lösung in die vorhandene Tabelle einzutragen.

Die Beschriftung der Tabelle ist auf Grund der Datenstrukturen mit geringem Aufwand verbunden. Sowohl die Terminalsymbole als auch die Nichtterminalsymbole sind als Arrays gespeichert und können wegen der Trennung der Symbole in der Parsertabelle bei der Ausgabe nacheinander ausgegeben werden.

Des Weiteren können die Zeilen nur durchnummeriert werden, um die Zustände zu referenzieren, da die Anzahl und die Reihenfolge der Zustände eindeutig bestimmt sind.

Nach der letzten Bestätigung aller Eingaben erfolgt die Korrektur der eingegebenen Aktionen, welche jedoch nicht durch eine Hilfsfunktion der Datenstrukturen durchgeführt wird, sondern durch einen Stringvergleich, der für das Korrigieren bereits ausreicht. Die eingegebene Aktion wird eingelesen, und mit den berechneten Shift- und Reduce-Aktionen verglichen. Dafür kann die Zeile genutzt werden, um die Kollektion zu referenzieren und die Spaltenbeschriftung für den Zugriff mit der *jumps*- und *reductions*-Map.

5.3.2 Warnungen und Fehlermeldungen

Um Fehler in der Eingabe zu erkennen und dem Nutzer einen entsprechenden Hinweis geben zu können, wird bereits nach der Eingabeverarbeitung die gesamte Berechnung durchgeführt. Dabei gibt es vier Arten von Fehlern, welche sich darin unterscheiden wie schwerwiegend der Fehler ist.

1. Art: Korrigierbarer Fehler ohne Auswirkung auf die Berechnung.
2. Art: Nicht korrigierbarer Fehler mit Auswirkung auf die Berechnung.
3. Art: Nicht korrigierbarer Fehler, welcher die Fortsetzung der Berechnung verhindert.

Bei dem Auftreten eines Fehlers müssen zwei Schritte unternommen werden, welche im Folgenden exemplarisch für einige Fehler betrachtet werden sollen. Erstmal muss eine Fehlererkennung erfolgen und schließlich eine Fehlerkorrektur, wenn der Fehler von 1. Art ist, oder die Ausgabe eines Hinweises, wenn der Fehler 2. oder 3. Art ist.

Fehler 3. Art Wichtig zu beachten ist, dass die Schwere der Fehler nicht zwangsweise mit der Komplexität des Fehlers zusammenhängt. So ist beispielsweise der folgende Fehler als Fehler 3. Art zu klassifizieren:

- Mehr als ein Nichtterminal wurde als linke Seite einer Produktionsregel eingegeben.

Dieser Fehler ist nicht vermeidbar und führt bei einem Auftreten dazu, dass die Berechnung nicht durchgeführt werden kann. Begründet wird dies dadurch, dass die Datenstrukturen und Algorithmen alle auf eine kontextfreie Grammatik eingerichtet sind und durch den Fehler die Grammatiken nicht mehr kontextfrei sind.

Erkannt werden kann dieser Fehler bereits in der Eingabeverarbeitung durch eine entsprechende **if**-Bedingung.

Da diese jedoch nicht korrigierbar ist, wird nur ein Popup-Hinweis ausgegeben, welcher in roter Farbe den Fehler markiert.

Andere Fehler 3. Art sind vergleichsweise leicht erkennbar, führen jedoch auch zum Abbruch der Berechnung und können daher nur durch einen Hinweis angezeigt werden.

Fehler 2. Art Auch die Fehler 2. Art können nicht korrigiert werden, wobei diese jedoch Auswirkungen auf die Berechnung haben. Hierbei muss die Berechnung nicht abgebrochen werden, da die Algorithmen trotz des Fehlers noch angewendet werden können. Diese Fehler sind insonfern selten, dass der Fehler nur bestimmte, kleine Auswirkungen haben darf. Sie treten aber dennoch auf, wie anhand des folgenden Beispiels zu erkennen ist:

- Eine nicht-deterministisch kontextfreie Grammatik wurde eingegeben.

Da eine solche Grammatik noch kontextfrei ist, kann diese zwar eingegeben werden, führt im LR(1)-Verfahren aber zu Konflikten in der Parsertabelle, da wie in 4.4.2.3 behandelt das LR(1)-Verfahren nur deterministisch kontextfreie Grammatiken erkennt. Bei nicht-deterministisch kontextfreien Grammatiken kann das Verfahren genutzt werden, führt aber in der Parsertabelle bei gleichen Eingaben zu widersprüchlichen Aktionen. Dabei kann ein shift-shift-Konflikt, welcher zu gleichen Eingaben verschiedene Shift-Aktionen vorgibt, ein shift-reduce-Konflikt, in dem eine shift- und gleichzeitig eine reduce-Aktion vorgegeben wird oder ein reduce-reduce-Konflikt, welcher bei der gleichen Eingabe verschiedene reduce-Aktionen anzeigt entstehen.

In den Algorithmen macht sich dieser Fehler dadurch bemerkbar, dass in einer Kollektion, über dem selben Symbol entweder mehrere Sprünge, ein Sprung zusammen mit einer Reduktion oder mehrere Reduktionen zu der *jump*- oder *reductions*-Map hinzugefügt werden. Die Erkennung ist damit auch ziemlich einfach, indem entweder nach der Berechnung für alle Map-Einträge eine Überprüfung stattfindet oder während der Berechnung ein mehrfacher Aufruf der **addShift**- oder **addReduction**-Funktion mit den selben Parametern erfolgt.

Um den Nutzer auf diese Fehler aufmerksam zu machen, wurde die Datenstruktur in der Hinsicht entsprechend angepasst, dass die Konflikte aufgezeichnet werden können. Dafür wurden die beiden Maps als String zu Int-Array Maps umgesetzt. Die Konflikte werden dann in der Parsertabelle durch mehrere Einträge in der Zelle des Symbols inkludiert, wo der Konflikt auftritt.

Formal gesehen ist diese Tabelle zwar fehlerhaft, da sie mehrdeutig ist, jedoch wurde die Umsetzung absichtlich angepasst, um diesen Fehler zu „erlauben“. Begründet wird dies dadurch, dass die Tabelle mit dem Konflikt für den Nutzer einen größeren Wert liefert, anstatt lediglich über einer Warnung den Nutzer darüber zu informieren, dass die Grammatik nicht deterministisch ist.

Ein weiteres Beispiel eines wichtigen Fehlers 2. Art ist folgender Fehler:

- Links-Rekursive Produktionsregel ohne Rekursionsanker.

Hierbei kann die Berechnung fortgesetzt werden, da alle Algorithmen mit der endlosen Rekursion umgehen können, jedoch kann das Ergebniss dadurch verfälscht werden.

Um diesen Fehler zu erkennen, kann die Mächtigkeit der First-Mengen betrachtet werden. Diese sind schließlich leer, falls die Grammatik keinen Rekursionsanker enthält. Nichtsdestotrotz fällt auf, dass mit den First-Mengen die Existenz des Rekursionsankers nur für Links-Rekursive Produktionsregeln geprüft werden kann. Die Prüfung vor diesem Hintergrund speziell auf die Tatsache gestützt, dass der First-Algorithmus nur so lange rechnet, bis keine Änderung mehr erfolgt. Dadurch bleibt die First-Menge des Nichtterminalsymbols der rekursiven Produktionsregel folglich leer, zumal diese zur Berechnung die eigene First-Menge benötigt. Diese kann jedoch aus keiner anderen Produktionsregel berechnet werden, sofern keine weiteren Produktionsregeln in der Grammatik enthalten ist, welche das Nichtterminal als linke Seite enthalten.

Auch hier kann mit einer Fortsetzung der Berechnung und einem entsprechenden Hinweis ein größerer Wert für den Nutzer entstehen, als wenn die Berechnung abgebrochen wird. Der Nutzer wird somit verleitet mit den Grammatiken zu experimentieren, wodurch das Verständnis für diese erhöht werden kann.

Fehler 1. Art Fehler 1. Art führen nicht zwangsweise zu Fehlern in der Berechnung und werden daher nur angezeigt, um den Nutzer auf die Formalien aufmerksam zu machen. Diese beinhalten beispielsweise folgende Fehler:

- Kein Symbol oder das leere Symbol als linke Seite einer Produktionsregel eingetragen.
- Unerreichbare Produktionsregel enthalten.⁷

Obwohl hier die Fehler nicht vermeidbar sind, kann die Berechnung trotzdem durchgeführt werden, indem die entsprechenden Regeln in der Berechnung ignoriert werden. Das erste

⁷Eine unerreichbare Produktionsregel ist eine Regel, welche durch keine Ableitung des Startsymbols erreichbar ist (Bemerkung 4.1.4.4).

Beispiel hinsichtlich des Fehlers 1. Art ist analog zu dem Beispiel des Fehlers 4. Art erkennbar. Es muss lediglich eine entsprechende Prüfung durch eine **if**-Abfrage durchgeführt werden, sobald die Eingabe eingelesen wird.

Das zweite Beispiel illustriert einen der Fehler, dessen Erkennung durch die Betrachtung der Follow-Mengen erfolgen kann und stellt somit einen der Gründe für die Berechnung der Follow-Mengen dar. Ferner gilt für unerreichbare Produktionsregeln, dass die Follow-Menge der linken Seite der Produktionsregel leer ist. Damit kann nach der Berechnung der Follow-Mengen dieser Fehler durch entsprechende Prüfungen der Mächtigkeit der Follow-Mengen erkannt werden.

Bei beiden Fehlern werden in der Zustandsberechnung die Produktionsregeln ignoriert, da diese durch keinen Sprung erreichbar sind und auch nicht durch die Hüllen-Berechnung aufgenommen werden. Dies ist darauf zurückzuführen, dass der Sprung ohne Hüllen-Bildung keine weiteren Elemente hinzufügt, sondern lediglich den Punkt verschiebt und außerdem in der Hüllen-Berechnung ausschließlich die Startelemente aller Folgesymbole aufgenommen werden. Dennoch ist das Nichtterminal (beziehungsweise das leere Symbol oder das fehlende Symbol) der Produktionsregel unerreichbar und damit insbesondere kein Folgesymbol, sodass die Produktionsregeln ohne weitere Veränderungen der Algorithmen ignoriert werden.

6 Evaluierung des LR(1)-Parsergenerators

Mithilfe aller Erkenntnisse aus der Analyse des LR(1)-Parsergenerators kann nun eine Evaluierung des Projekts durchgeführt werden. Um diese Evaluierung durchzuführen, wird geprüft, wie die zwei grundlegenden Ziele des Projekts erreicht wurden, welche auch bereits in dem Kapitel der Motivation 2.1 und der Problemstellung 2.2 aus der Einleitung eingeführt sind. Die zwei Ziele beinhalten zum einen die deterministisch kontextfreien Sprachen zu erkennen und zum anderen die Möglichkeit zur Nutzung des Parsergenerators als e-Learning-Programm.

Dafür werden die funktionalen und nicht-funktionalen Anforderungen des Projekts betrachtet. Außerdem wird mit der Argumentation aus der Analyse begründet, ob und wie diese erfüllt wurden.

6.1 Funktionale Anforderungen

Wie bereits in Kapitel 5.1 behandelt, sind die funktionalen Anforderungen in zwei Kategorien aufzuteilen. Diese sind nach den beiden Zielen orientiert und sind einerseits die Anforderungen der Theorie, um das Parsen der gewünschten Sprachklasse zu ermöglichen und andererseits die Anforderungen der interaktiven Visualisierung, um den Nutzen als e-Learning-Programm zu liefern.

Anforderungen der Theorie Aus der Theorie entstehen folgende Anforderungen. Die Datenstrukturen müssen definitionsgemäß umgesetzt werden, damit alle bewiesenen Eigenschaften aus der Theorie gelten. Somit können die Datenstrukturen folglich in den Algorithmen verwendet werden.

Dies ist wichtig, um so die Erkennung deterministisch kontextfreier Grammatiken zu garantieren, wie auch Knuth in „On the Translation of Languages from Left to Right“ [14] bewies, indem er belegte, dass die von LR(1)-Parsern erkannte Sprachklasse genau die Sprachklasse der deterministisch kontextfreien Sprachen ist.

Um die eben genannte Anforderung zu erfüllen, wurden beispielsweise entsprechende **append**-Funktionen umgesetzt, welche die Vereinigung aus den Algorithmen implementieren. Weiterhin wurde auch ein großer Fokus darauf gelegt, dass diese Anforderung erfüllt wird, indem die definierten Eigenschaften durch Wrapper-Funktionen gewahrt wurden. Jedoch konnte die Umsetzung aller definierten Eigenschaften aufgrund der Anforderungen

der Visualisierung nicht vollständig umgesetzt werden. Sodass die Nutzung der Datenstrukturen in den Algorithmen nicht immer möglich war. Um dennoch die Algorithmen verwenden zu können, wurden Anpassungen an den Algorithmen vorgenommen. Die Korrektheit der angepassten Algorithmen konnte dann in Kapitel 5.2 überprüft werden, indem die Äquivalenz der angepassten Algorithmen und der bereits bekannten Algorithmen gezeigt wurde. Dadurch waren zwar die Anforderung, die Algorithmen exakt so zu verwenden, nicht erfüllt, jedoch zeigte die Äquivalenz der Algorithmen, dass das Ziel deterministisch kontextfreier Sprachen zu erkennen, erreicht wurde.

Anforderungen der Visualisierung Die Anforderungen der Visualisierung beinhalten, dass die Datenstrukturen alle Informationen enthalten, welche für die Nutzer der Anwendung sichtbar sein müssen. Dies ist in jedem Fall notwendig für die Verwendung als Lernprogramm.

Zur Erfüllung dieser Anforderung wurde beispielsweise die Benennung der Zustände benötigt. Die Informationen mussten demnach von den Datenstrukturen durch erleichterte Zugriffsmöglichkeiten verfügbar gemacht werden. Im Falle der Benennung der Zustände, wurde dafür die eindeutig festgelegte Reihenfolge genutzt und in den **toString**-Methoden zur Verfügung gestellt, damit diese Ausgegeben werden konnten.

Des Weiteren mussten Hilfsfunktionen verfügbar sein, welche die Umsetzung der interaktiven Aspekte der Visualisierung erleichtern. Dazu gehören zum Beispiel die entwickelten **equals**-Funktionen, welche die Korrekturfunktion, unter Einhaltung der theoretischen Anforderungen, ermöglichen.

Somit konnte in Kapitel 5.1 gezeigt werden, dass zumindest aus funktionaler Sicht diese Anforderung zufriedenstellend erfüllt wurde.

6.2 Nicht-Funktionale Anforderungen

Um das Ziel, eine Nutzbarkeit als e-Learning Programm zu erreichen, reicht die reine Erfüllung der Funktionalität nicht. Damit der Nutzer das Programm als Mittel zum Lernen verwenden kann, muss dieser sich auf den Inhalt konzentrieren können. Dementsprechend muss der Nutzerfreundlichkeit eine hohe Priorität zugeteilt werden.

Einen großen Aspekt der Nutzerfreundlichkeit bildet die Reaktionszeit des Programms und um dieses zu erhöhen wurde das Programm als Serverless Website entwickelt, welches alle Berechnungen lokal durchführt. Diesbezüglich stellt sich jedoch die Frage, ob die lokale Rechenleistung ausreichend ist und wie groß die Laufzeiten der Algorithmen sind.

Für die Beantwortung dieser Frage wurde in Kapitel 5.2 zu jedem Algorithmus eine Laufzeitanalyse durchgeführt. Diese ergab dabei jedoch, dass die Algorithmen nicht sehr effizient arbeiten. Hierbei ist zu beachten, dass in der tatsächlichen Implementierung der

Algorithmen noch weitere Details umgesetzt werden müssen, welche die Laufzeit zusätzlich verschlechtern können. Ein Beispiel dafür sind die First-Funktionen in den Algorithmen 5.2.2 und 5.2.3, welche aus Gründen der Simplizität in größerer Laufzeit umgesetzt wurden als notwendig. Außerdem konnten in der Laufzeitanalyse nicht die kleinsten Laufzeitenklassen bestimmt werden, sodass auch die tatsächliche Laufzeit nicht exakt berechnet werden konnte. Weiterhin sind diese Laufzeiten stark an den Datenstrukturen gebunden, da beispielsweise die `append`- und `includes`-Funktionen mit einer Hashmap in $\mathcal{O}(1)$ hätten umgesetzt werden können.

Um dennoch die Evaluierung auch in diesem Aspekt durchführen zu können, wurden eigene Messungen der tatsächlichen Berechnungszeit durchgeführt, indem die JavaScript interne **`console.time`**-Funktion genutzt wurde. Dafür wurden verschiedene Grammatiken mit bis zu 20 Produktionsregeln getestet. Dabei ergab sich eine durchschnittliche Berechnungszeit von 0,236 Sekunden, sodass die Berechnung zumindest für kleine Eingaben schnell genug ist.

Da diese Parsergenerierung nicht in weiteren Applikationen genutzt werden kann, und außerdem für die Nutzung als Lernprogramm keine großen Grammatiken benötigt werden, ist die schnelle Berechnung für kleine Eingaben ausreichend, um diese Anforderung zu erfüllen. Die graphische Nutzeroberfläche kann so sofort aktualisiert werden. Die Effizienz hat bei geringen Berechnungszeiten somit keinen großen Einfluss auf die Nutzerfreundlichkeit.

Des Weiteren wurden, um die Nutzerfreundlichkeit weiter zu erhöhen und den Nutzen des Lernprogramms zu erweitern, Hinweise, Warnungen und Fehlermeldungen implementiert, damit das Programm auch fehlerhafte Eingaben sinnvoll verarbeiten kann.

Für die Implementierung der Meldungen an den Nutzer wurden dabei weitere Anpassungen an den Datenstrukturen vorgenommen. Diese sind in Kapitel 5.3.2 zusammen mit der Funktionsweise der Meldungen erläutert.

Begründet wird das Nutzen von Fehlermeldungen damit, dass die Nutzer durch das Abfangen von Fehlern einfach und ohne negative Auswirkungen experimentieren können. Dadurch werden Nutzer ermutigt verschiedene Grammatiken auszuprobieren, um deren Auswirkungen zu erkunden. Dieses Experimentieren führt zu einem besseren Lernerlebnis und kann somit das Verständnis fördern. (vgl. [15] S.17)

Insgesamt kann somit zumindest mit der theoretischen Evaluierung die Nutzbarkeit als Lernprogramm bestätigt werden.

Ursprünglich war es geplant, dass die Effizienz des Lernprogramms, im Bezug auf das Lernen, mithilfe von Fragebögen von den Nutzern evaluiert werden kann. Dafür wurde ein Feedbackbogen in die Website integriert, welche über einen Knopf in dem Menü der Seite verfügbar war, und automatisch erscheint, wenn die Parsertabelle korrekt ausgefüllt wurde. Diese Evaluation konnte aufgrund von fehlenden Rückmeldungen nicht vorgenommen werden, sodass nur die Funktionsweise und die theoretische Wirksamkeit des Lernprogramms evaluiert werden konnten.

Fazit Der LR(1)-Parsergenerator erfüllt beide gesetzten Ziele und ist daher als erfolgreiches Projekt zu bewerten. Es konnte im Rahmen dieser Bachelorarbeit gezeigt werden, dass eine Visualisierung des LR(1)-Verfahrens zur Erkennung der deterministisch kontextfreien Sprachen erstellt werden konnte, welche für Studierende im Fach Compilerbau auch als Lernprogramm verwendet werden kann.

Den Vorteil gegenüber anderen Parsergeneratoren liefert der LR(1)-Parsergenerator auf Grund des Fokus auf die Interaktivität. Dadurch ist dieses besonders gut als e-Learning Programm geeignet und liefert insbesondere im Vergleich zu anderen Parsergeneratoren den gewünschten bedeutsamen Mehrwert für den Nutzer. Um zu sehen, dass dieser Vorteil einzigartig für den erstellten Generator ist, muss der Parsergenerator nur mit anderen Parsergeneratoren verglichen werden, welche Visualisierungen ohne Parsingfunktionalität erstellen. In Kapitel 3.1 wurde dafür gezeigt, dass sich nur diese als Lernprogramm eignen. Dies sind genau die Generatoren, welche in Kapitel 3.1 in die zweite Kategorie klassifiziert wurden und in das auch der LR(1)-Parsergenerator zu klassifizieren ist.

In dem Kapitel über den Stand der Technik wurden unter anderem Parsergeneratoren der zweiten Kategorie beispielhaft vorgestellt. Dabei kann leicht gesehen werden, dass diese keine e-Learning-Funktionalitäten liefern, womit der Mehrwert bestätigt ist.

7 Schluss

7.1 Zusammenfassung

In dieser Arbeit wurden ausführlich die Hintergründe eines Parsergenerators erläutert, welcher das Ziel hat, als e-Learning Programm zu dienen. Dieser Parsergenerator nutzt das LR(1)-Verfahren von Donald Knuth [14] für das Erstellen von Parsern, die deterministisch kontextfreie Sprachen erkennen können.

Es fand sowohl eine Erklärung der Umsetzung der Datenstrukturen als auch eine Erläuterung der Umsetzung der Algorithmen. Des Weiteren wurden die Funktionsweisen einiger weiterer Funktionalitäten behandelt, welche für den e-Learning Aspekt des Programms von Relevanz sind.

Genauer erfolgte die Begründung der Designentscheidungen der Datenstrukturen durch die Zurhilfenahme der formalen Definitionen aus der Theorie der Formalen Sprachen. Dafür fand eine Herleitung der relevanten Definitionen und Eigenschaften statt. Beginnend bei Alphabeten und Wörtern wurden anschließend formale Grammatiken und Sprachen eingeführt. Aufbauend auf diesen Grundlagen konnten schließlich deterministisch kontextfreie Sprachen mithilfe der Theorie über Kellerautomaten hergeleitet werden, da dies die Sprachklasse ist, die von den LR(1)-Parsern erkannt werden können.

Aufgrund der speziellen Anforderungen des e-Learning-Aspekts an die Ausgabe konnten die Datenstrukturen dennoch nicht definitionsgemäß umgesetzt werden. Diese fehlende exakte Übereinstimmung mit den Definitionen führte dazu, dass die Algorithmen auch angepasst werden mussten.

Die dafür notwendigen Änderungen an den Algorithmen wurden folgend auf Korrektheit geprüft. Des Weiteren fand eine Laufzeitanalysen für die Nutzung in der Auswertung des Projektes statt. Vorbereitend auf diese Analysen wurde das LR(1)-Verfahren als Teil der Theorie eingeführt und erläutert. Somit konnte die Funktionstüchtigkeit des Projektes bewiesen werden.

Für die Aspekte des Programmes, welche für die Interaktion und den Lerneffekt zuständig sind, erfolgte im Anschluss eine genauere Betrachtung derer Funktionsweisen. Dieser Teil der Arbeit zeigte die Nutzbarkeit des Parsergenerators als e-Learning-Programm und schloßte damit den Hauptteil der Arbeit ab.

Die gesamten Erkenntnisse aus der Analyse bildeten dann die Basis für die Evaluation des Programms, um so zu demonstrieren, dass der LR(1)-Parsergenerator alle funktionalen Ziele erreicht hat und infolgedessen als Projekt erfolgreich abgeschlossen wurde.

Um außerdem den Nutzen des Projektes aufzuzeigen, wurde die Kategorisierung vorhandener Parsergeneratoren aus dem Stand der Technik genutzt, indem ein Vergleich der Nutzbarkeit zwischen dem LR(1)-Parsergenerator und den vorhandenen Generatoren erfolgte, welcher aufwies, dass der LR(1)-Parsergenerator besser als e-Learning-Programm geeignet ist.

7.2 Ausblick

Wie die meisten Projekte ist auch dieses Projekt trotz des Abschlusses noch erweiterbar. Die Berechnung der Parsertabelle ist in dem Projekt zwar beendet, jedoch gibt es für die Benutzeroberfläche noch viele Dinge, welche mit mehr Zeit umzusetzen wären. Sollte das Projekt fortgesetzt werden, bieten sich vor allem folgende Aspekte an:

Die offensichtlichen Punkte beinhalten die Erweiterung der bereits vorhandenen Funktionalitäten, wie beispielsweise die Erweiterung der Fehlermeldungen, sodass mehr Fehler erkannt werden.

Im Lernprogramm ist außerdem die Abfrage der Zwischenschritte lückenhaft, da die Menge der Zustände lediglich Ausgegeben wird und diesbezüglich keine Abfrage stattfindet. Durch eine Vereinfachung der Datenstruktur oder eine komplexe Umsetzung einer Abfragemöglichkeit, kann die Abfrage erweitert und der Nutzen des Programms damit erhöht werden. Dies kann beispielsweise durch die Angabe der Hüllen oder Sprung-Funktion mit Parametern als Fragestellung, und eine Liste von Textfeldern für die Elemente als Eingabe des Lösungsvorschlags implementiert werden.

Um jedoch das Projekt als Gesamtes zu erweitern, sind folgende Erweiterungen um fehlende Funktionalitäten notwendig:

Es könnten weitere Parsergenerierungsverfahren, wie das LALR-Verfahren (siehe 3.2) umgesetzt werden. Außerdem ist für die Fortsetzung der Parsertabelle eine Umsetzung der Visualisierung des entsprechenden Parserautomaten möglich. Dies kann beispielsweise helfen das Verständnis der LR(1)-Parser als Automaten weiter zu vertiefen und somit das Lernprogramm zu erweitern.

Schließlich kann durch die Umsetzung und Zurverfügungstellung des generierten Parsers, mit anschließender Visualisierung des Parsingverfahren die Nutzungsform der dritten Kategorien aus 3.1 hinzugefügt werden. Dadurch kann der Nutzer, neben den bereits vorhandenen Lernmitteln über den Parser als Konstrukt, mehr über das Parsen als Aktion lernen.

Insgesamt kann das Projekt also in der eigenen Funktionalität verbessert werden, aber auch durch das Hinzufügen weiterer Funktionalitäten erweitert werden. Diese Verbesserungen und Erweiterungen sind aber strikt als Ergänzungen zu betrachten, da das Projekt bereits eine eigenständige Funktionstüchtigkeit besitzt und den gewünschten Nutzen erfüllt.

Literaturverzeichnis

- [1] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, *Compilerbau Teil 1*, 2. Auflage, De Gruyter Oldenbourg, 1999.
- [2] Iwo Bladek, Krzysztof Krawiec *Solving symbolic regression problems with formal constraints* 2019, <https://doi.org/10.1145/3321707.3321743>
- [3] Hans-Joachim Böckenhauer, Juraj Hromkovič *Formale Sprachen - Endliche Automaten, Grammatiken, lexikalische und syntaktische Analyse* 1. Auflage, Springer, 2013 <https://doi.org/10.1007/978-3-658-00725-6>
- [4] Gang Chen, Huiming Jiang, Mei Liu, Peng Wei *Formal Language Generation for Fault Diagnosis with Spectral Logic via Adversarial Training*, 2020, <https://doi.org/10.1109/TII.2020.3040743>
- [5] Noam Chomsky, *Three models for the description of language*, in: Transactions on Information Theory, Volume 2, Issue 3, 1956. <https://doi.org/10.1109/TIT.1956.1056813>
- [6] Franklin Lewis DeRemer, *Practical translators for LR(k) languages*, 1965. <http://hdl.handle.net/1721.1/13628>
- [7] Horacio Feliciangeli, Gábor T. Herman *Algorithms for Producing Grammars from Sample Derivations: a Common Problem of Formal Language Theory and Developmental Biology*, in: Ferenc Papp, György Szépe, Papers in Computational Linguistics 2020, <https://doi.org/10.1515/9783110806700-045>
- [8] Seymour Ginsburg, Sheila Greibach, *Deterministic context free languages*, 1965 <https://doi.org/10.1109/FOCS.1965.7>
- [9] Rachel Greenstadt, Levi Kapllani, Spiros Mancoridis, Aviel J. Stein *Exploring Paraphrasing Techniques on Formal Language for Generating Semantics Preserving Source Code Transformations*, 2020, <https://doi.org/10.1109/ICSC.2020.00051>
- [10] Heinz-Peter Gumm, Manfred Sommer *Formale Sprachen, Compilerbau, Berechenbarkeit und Komplexität*, in: Heinz-Peter Gumm, Manfred Sommer, Grundlagen der Informatik, Band 3, 1. Auflage, De Gruyter Oldenbourg, 2019, <https://doi.org/10.1515/9783110442397>
- [11] Alan Kaplan, Denise Shoup, *CUPV - a visualization tool for generated parsers*, in: ACM Sigcse Bulletin, Volume 32, p. 11-15 2000, <https://doi.org/10.1145/330908.331801>

- [12] Michael Kerres, *Mediendidaktik*, 5. Auflage, De Gruyter Oldenbourg, München, 2018.
- [13] Zak Kincaid, Shaowei Zhu, Princeton University *LR(0) Parser Visualization*, <https://www.cs.princeton.edu/courses/archive/spring20/cos320/LR0/> Abruf 07.07.2021
- [14] Donald Knuth, *On the Translation of Languages from Left to Right*, in: Information and Control, Volume 8, Issue 6, 1965. [https://doi.org/10.1016/S0019-9958\(65\)90426-2](https://doi.org/10.1016/S0019-9958(65)90426-2)
- [15] Christian Kreidl, *Akzeptanz und Nutzung von E-Learning-Elementen an Hochschulen. Gründe für die Einführung und Kriterien der Anwendung von E-Learning*, 1. Auflage, Waxmann, Münster, 2011.
- [16] A. J. Korenjak, *A practical method for constructing LR(k) processors*, in: Communications of the ACM, Volume 12, Issue 11, 1969. <https://doi.org/10.1145/363269.363281>
- [17] Andreas Kunert, *LR(k)-Analyse für Pragmatiker*, 2015. <https://amor.cms.hu-berlin.de/~kunert/papers/lr-analyse/lr.pdf>,
- [18] Hagen Langer, Sven Naumann, *Was ist Parsing* in: Parsing. Leitfäden und Monographien der Informatik, Vieweg+Teubner https://doi.org/10.1007/978-3-322-90877-3_1, 1994.
- [19] Uwe Meyer *Grundkurs Compilerbau*, 1. Auflage, Rheinwerk Verlag, 2021.
- [20] Ulrike Rockmann, *Qualitätskriterien für IT-basierte Lernmedien - nützlich oder unsinnig?*, in: Tergan, Sigmar-Olaf (Hrsg.), Schenkel, Peter (Hrsg.) *Was macht E-Learning erfolgreich?*, 1. Auflage, Springer, Berlin, 71-81, 2004.
- [21] Susan H. Rodger, Duke University *JFLAP*, <http://jflap.org> Abruf 08.07.2021
- [22] David Sabel, Ludwigs-Maximilians-Universität München, Vorlesung zu kontextfreien Sprachen, <https://www.tcs.ifi.lmu.de/lehre/ss-2019/fsk/material/folien/05-druck> Abruf 15.08.2021
- [23] Rion Snow, Stanford University *Stanford Parser Parse Visualization Tool*, http://ai.stanford.edu/~rion/parsing/stanford_viz.html Abruf 07.07.2021
- [24] Heiko Vogler, Technische Universität Darmstadt, *Parsing von natürlichen Sprachen*, 2017, <https://www.orchid.inf.tu-dresden.de/research/parsing.de/> Abruf 12.06.2021
- [25] o.V., *LR(1) Parser Generator*, <http://jsmachines.sourceforge.net/machines/lr1.html> Abruf 07.07.2021
- [26] o.V. (Fachbereich 06 Mathematik, Naturwissenschaften und Informatik (MNI) der Technischen Hochschule Mittelhessen), *Modulhandbuch zur Prüfungsordnung für den Bachelorstudiengang Informatik*, Version 9, geänderte Fassung vom 07. Oktober 2020 und 06. Januar 2021