

LEHRBUCH

Christian Wagenknecht
Michael Hielscher

Formale Sprachen, abstrakte Automaten und Compiler

Lehr- und Arbeitsbuch mit FLACI für
Grundstudium und Fortbildung

3. Auflage



Springer Vieweg

Formale Sprachen, abstrakte Automaten und Compiler

Christian Wagenknecht · Michael Hielscher

Formale Sprachen, abstrakte Automaten und Compiler

Lehr- und Arbeitsbuch mit FLACI für
Grundstudium und Fortbildung

3., überarbeitete und ergänzte Auflage



Christian Wagenknecht
Fakultät Elektrotechnik und Informatik
Hochschule Zittau/Görlitz
Görlitz, Deutschland

Michael Hielscher
Institut für Medien und Schule
Pädagogische Hochschule Schwyz
Goldau, Schweiz

ISBN 978-3-658-36852-4

ISBN 978-3-658-36853-1 (eBook)

<https://doi.org/10.1007/978-3-658-36853-1>

Die Deutsche Nationalbibliothek verzeichnetet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <http://dnb.d-nb.de> abrufbar.

© Springer Fachmedien Wiesbaden GmbH, ein Teil von Springer Nature 2009, 2014, 2022

Das Werk einschließlich aller seiner Teile ist urheberrechtlich geschützt. Jede Verwertung, die nicht ausdrücklich vom Urheberrechtsgesetz zugelassen ist, bedarf der vorherigen Zustimmung des Verlags. Das gilt insbesondere für Vervielfältigungen, Bearbeitungen, Übersetzungen, Mikroverfilmungen und die Einspeicherung und Verarbeitung in elektronischen Systemen.

Die Wiedergabe von allgemein beschreibenden Bezeichnungen, Marken, Unternehmensnamen etc. in diesem Werk bedeutet nicht, dass diese frei durch jedermann benutzt werden dürfen. Die Berechtigung zur Benutzung unterliegt, auch ohne gesonderten Hinweis hierzu, den Regeln des Markenrechts. Die Rechte des jeweiligen Zeicheninhabers sind zu beachten.

Der Verlag, die Autoren und die Herausgeber gehen davon aus, dass die Angaben und Informationen in diesem Werk zum Zeitpunkt der Veröffentlichung vollständig und korrekt sind. Weder der Verlag noch die Autoren oder die Herausgeber übernehmen, ausdrücklich oder implizit, Gewähr für den Inhalt des Werkes, etwaige Fehler oder Äußerungen. Der Verlag bleibt im Hinblick auf geografische Zuordnungen und Gebietsbezeichnungen in veröffentlichten Karten und Institutionsadressen neutral.

Planung: David Imgrund

Springer Vieweg ist ein Imprint der eingetragenen Gesellschaft Springer Fachmedien Wiesbaden GmbH und ist ein Teil von Springer Nature.

Die Anschrift der Gesellschaft ist: Abraham-Lincoln-Str. 46, 65189 Wiesbaden, Germany

Vorwort

Lernen und (erst recht) Studieren erfordern Aktivität. Die beste Vorlesung bleibt wirkungslos, wenn sie als „Kinoeffekt“ verpufft. Schnell sieht man ein, dass *praktische* Prozesse, die man beherrschen muss, *eingeübt* werden müssen. Abstrakte Denktechniken scheinen da pflegeleichter zu sein. Falsch! Die wirkliche Verinnerlichung abstrakter Inhalte erfordert beharrliche geistige *Aktivität*.

Dabei soll dieses *Arbeitsbuch* zur theoretischen Informatik helfen. *Benutzen* Sie es! Verinnerlichen Sie die anfangs unappetitlichen Happen. Sie werden zunehmend spüren, dass abstrakte Denktechniken Freude bereiten können. Dies gilt auch für verwandte Denkaktivitäten außerhalb der theoretischen Informatik. Die Reise lohnt sich und SIE sind der Kapitän. Wir reichen Ihnen Karte und Kompass.

Der Text basiert auf Vorlesungen zur theoretischen Informatik, die es seit 1993 im Fachbereich Informatik an der Hochschule Zittau/Görlitz gibt. Sie wurden kontinuierlich modifiziert, verbessert und durch Begleitmaterial im Web ergänzt. Didaktische Hinweise werden im Text offen gegeben: Was Lehrpersonen diskutieren und anstreben, soll kein Geheimnis für Studierende sein.

Das vorliegende Lehrbuch ist vor allem für Bachelorstudierende der Informatik aber auch für Studierende anderer Fachgebiete, die sich mit formalen Sprachen und abstrakten Automaten beschäftigen, geeignet. Darüber hinaus eignet es sich für die Fortbildung von Lehrkräften an Gymnasien sowie interessierte Schülerinnen und Schüler entsprechender Kurse. Es werden lediglich Schulwissen aus der Mathematik und Grundkenntnisse aus der Informatik vorausgesetzt.

Kernstück des Arbeitsbuches ist eine Lern- und Arbeitsumgebung. Von 2004 bis 2016 war dies AtoCC (from Automaton to Compiler Construction), <http://www.atocc.de>. Ab dem Jahr 2017 verwenden wir FLACI (Formal Languages, abstract Automata, Compiler and Interpreter), <https://flaci.com/>, – eine Weiterentwicklung von AtoCC, s. u. Bemerkungen zur dritten Auflage.

Die Lern- und Arbeitsumgebung bietet die Möglichkeit, Grundlagen der Theorie formaler Sprachen und Automaten mit Bezug auf sehr praxisbezogene Anwendungen im automatisierten Übersetzerbau (compiler compiler bzw. compiler construction) am Computer umzusetzen. Andere Teilgebiete der theoretischen Informatik (Berechenbarkeitstheorie, Komplexitätstheorie) werden hier lediglich mit Erwähnungen oder kurzen Kommentaren bedacht.

Der Dank der Autoren gilt allen Personen, die sich seit 2004 mit dem Material beschäftigt und mit ihrem Feedback zur Entstehung bzw. Qualifizierung des Textes beigetragen haben. Musterlösungen oder Lösungshinweise zu den Übungsaufgaben finden Sie bei den studienbegleitenden Ressourcen, die Sie über

die Seite <https://flaci.com/buch> erreichen. Für Ihre Fragen, kritischen Hinweise und Bemerkungen gibt es die Adresse mail@flaci.com.

In der zweiten Auflage wurden Schreibfehler korrigiert und verbesserte bzw. angepasste Algorithmen eingearbeitet. Beispielsweise wurde das Verfahren zur Eliminierung von Kettenregeln vereinfacht. Für Studierende schwierige Themen, wie der Satz von Myhill und Nerode und die Pumping Lemmata für reguläre und kontextfreie Sprachen, und weitere Textpassagen wurden didaktisch angereichert.

Mit der dritten Auflage wird FLACI integriert. FLACI ist eine Lern- und Arbeitsumgebung, die sich dem Studierenden als wartungsfreie Online-Software darstellt. Um FLACI zu verwenden ist lediglich ein Webbrowser (vorzugsweise Google Chrome oder Firefox) erforderlich. Insbesondere an der inhaltlichen Abfolge wurden deutliche Veränderungen vorgenommen: Die Modellierung von Übersetzungsprozessen wurde schon im Einführungskapitel integriert und durch ausführbare praktische Übungen vertieft. Dadurch soll der erforderliche Kenntniserwerb über die angewandten Methoden noch stärker motiviert werden.

Die trotz großer Sorgfalt nicht auszuschließenden Fehler bzw. Ungeschicklichkeiten sind allein den Autoren anzulasten. Bitte senden Sie uns Ihre Anmerkungen. Dafür sind wir sehr dankbar!

Vervielfältigungen beliebiger Teile dieses Textes oder deren Weitergabe sind nicht gestattet.

© 2022 by Christian Wagenknecht and Michael Hielscher
<http://www.christian-wagenknecht.de/>
<http://www.michael-hielscher.de/>

Inhaltsverzeichnis

| | |
|--|------------|
| 1 Einleitung | 1 |
| 1.1 Sprachen als Kommunikationsmittel | 1 |
| 1.2 Sprachübersetzer: Interpreter und Compiler | 2 |
| 1.3 Definition von Sprachen | 6 |
| 1.4 Alphabet und Zeichen | 7 |
| 1.5 Wort, Wortlänge und Verkettung | 9 |
| 1.6 Wortmenge | 12 |
| 1.7 Formale Sprache | 15 |
| 1.8 Konkrete Syntax | 18 |
| 1.9 Regeln, (E)BNF und Syntaxdiagramme | 18 |
| 1.10 Syntaxanalyse | 23 |
| 1.11 Abstrakte Syntax | 27 |
| 2 Formale Grammatiken | 31 |
| 2.1 Muster und formale Grammatiken | 31 |
| 2.2 Ableitung und definierte Sprache | 34 |
| 2.3 Nichtdeterminismus des Ableitungsprozesses | 36 |
| 2.4 Mehrdeutigkeit kontextfreier Grammatiken | 38 |
| 2.5 CHOMSKY-Hierarchie | 41 |
| 2.6 ϵ -Sonderregelungen | 43 |
| 2.7 Das Wortproblem | 46 |
| 3 Endliche Automaten und reguläre Sprachen | 49 |
| 3.1 Allgegenwärtige Zustandsmodelle | 49 |
| 3.2 Deterministischer Endlicher Automat (DEA, EA) | 52 |
| 3.3 Endlicher Automat und reguläre Grammatik | 57 |
| 3.4 Nichtdeterministischer endlicher Automat (NEA) | 60 |
| 3.5 Konstruktion eines äquiv. DEA aus einem NEA | 66 |
| 3.6 Abschlusseigenschaften regulärer Sprachen | 71 |
| 3.7 Satz von Myhill und Nerode | 77 |
| 3.8 Minimalautomat | 85 |
| 3.9 Das Pumping Lemma für reguläre Sprachen | 87 |
| 3.10 NEA mit ϵ -Übergängen | 91 |
| 3.11 Endliche Maschinen | 98 |
| 4 Reguläre Ausdrücke (RA) | 103 |
| 4.1 Reguläre Mengen | 103 |

| | | |
|----------|--|------------|
| 4.2 | Klammersparregeln und Äquivalenzen | 105 |
| 4.3 | Reguläre Ausdrücke und endliche Automaten | 107 |
| 4.4 | Reguläre Ausdrücke in der Praxis | 112 |
| 4.5 | Reguläre Ausdrücke in Scannergeneratoren | 117 |
| 5 | Sprachübersetzer | 121 |
| 5.1 | Compiler und Interpreter | 121 |
| 5.2 | Modellierung von Übersetzungsprozessen | 122 |
| 5.3 | Lexikalische Analyse | 129 |
| 5.4 | Syntaktische Analyse | 135 |
| 6 | Kellerautomaten und kontextfreie Sprachen | 139 |
| 6.1 | Grenzen endlicher Automaten | 139 |
| 6.2 | Nichtdeterministischer Kellerautomat (NKA) | 140 |
| 6.3 | Äquivalenz von NKA und kontextfreier Grammatik | 147 |
| 6.4 | Parsing kontextfreier Sprachen | 154 |
| 6.5 | Deterministischer Kellerautomat (DKA) | 156 |
| 6.6 | Deterministisch kontextfreie Sprachen | 159 |
| 6.7 | Parsegeneratoren für dkfS | 161 |
| 6.8 | Optimierung kontextfreier Grammatiken | 163 |
| 6.9 | CHOMSKY-Normalform | 166 |
| 6.10 | Das Pumping Lemma für kontextfreie Sprachen | 169 |
| 7 | LL(k)-Sprachen | 173 |
| 7.1 | Deterministische Top-down-Syntaxanalyse | 173 |
| 7.2 | Begriff und Einordnung | 174 |
| 7.3 | <i>LL(1)</i> -Forderungen | 176 |
| 7.4 | Top-down-Parser für <i>LL(1)</i> -Grammatiken | 181 |
| 7.5 | Methode des Rekursiven Abstiegs | 186 |
| 7.6 | Grammatiktransformationen | 191 |
| 8 | LR(k)-Sprachen | 199 |
| 8.1 | Begriff | 199 |
| 8.2 | Deterministische Bottom-up-Syntaxanalyse | 199 |
| 8.3 | Tabellengesteuerte <i>LR(k)</i> -Syntaxanalyse | 203 |
| 8.4 | Automatisierte Parsegenerierung | 208 |
| 9 | Sprachübersetzerprojekte | 211 |
| 9.1 | Syntaxgesteuerte Übersetzung | 211 |
| 9.2 | Audiocompiler und Musikinterpretation | 213 |
| 9.3 | Projekt: Schachnotation | 223 |
| 9.4 | Compiler für die Zeichenroboter-Sprache ZR | 233 |
| 9.5 | Präsentation eines Compilers | 241 |
| 9.6 | Datenvisualisierung | 244 |

| | |
|--|------------|
| 10 TURING-Maschine (TM) und CHOMSKY-Typ-0/1-Sprachen | 247 |
| 10.1 Grenzen von Kellerautomaten | 247 |
| 10.2 Die TURING-Maschine (TM) | 248 |
| 10.3 Die Arbeitsweise einer DTM | 250 |
| 10.4 Die DTM als Akzeptor | 253 |
| 10.5 Alternative TM-Definitionen | 255 |
| 10.6 DTM, NTM, LBTM und Sprachklassen | 255 |
| 10.7 TM in Komplexitäts- und Berechenbarkeitstheorie | 257 |
| 10.8 TM zur Berechnung von Funktionen | 259 |
| Sachverzeichnis | 263 |

FLACI

Formale Sprachen, abstrakte Automaten, Compiler und Interpreter

HOME BEISPIELE KONTAKT

DIE MODULE

Formale Sprachen

Ein Alphabet A ist eine endliche, nichtleere Menge von Zeichen.

Wählen Sie eines des Beispiel-Alphabete zum Experimentieren aus:

- $A_1 = \{0,1\}$
- $A_2 = \{a,b,c,\dots,z\}$
- $A_3 = \{0,1,2,3,4,5,6,7,8,9\}$
- $A_4 = \{\text{!}, \text{#}, \text{&}, \text{@}, \text{^}, \text{?}, \text{<}, \text{>}, \text{<} \text{>}, \text{<} \text{>} \text{<} \text{>}\}$
- $A_5 = \{\text{begin}, \text{end}, \text{for}, \text{while}, \text{do}, \text{repeat}, \text{until}\}$

Interaktives Minitutorial zu den Grundbegriffen formaler Sprachen

Reguläre Ausdrücke

Der einfachste reguläre Ausdruck a steht für ein einzeliges Wort "a". Kurz: a besteht aus genau einer Abhahmenzeichen a . Der Ausdruck beschreibt die Sprache $L = \{a\}$.

Reguläre Ausdrücke sind beliebig erweiterbar: a gefolgt von b , gefolgt von c ; wird kurz abc geschrieben.

Was ändert sich in der berechenbare Sprache, wenn man den regulären Ausdruck a zu ab oder abc verändert?

| Regulärer Ausdruck | Ungültiger Ausdruck |
|--------------------|---------------------|
| a | ab |
| $a b$ | abc |
| $a b c$ | |

Interaktives Minitutorial zu regulären Ausdrücken

Formale Grammatiken

Kontextfreie Grammatiken entwickeln, transformieren und konvertieren

Abstrakte Automaten

Abstrakte Automaten konstruieren, simulieren, transformieren und konvertieren

Compiler und Interpreter

Modellieren von Übersetzungsprozessen und Entwicklung von Compilern und Interpretern



1 Einleitung

1.1 Sprachen als Kommunikationsmittel

Menschen verwenden *Sprachen*, um miteinander zu kommunizieren. Hierfür bilden sie grammatisch (*syntaktisch*) korrekte Sätze, die eine Bedeutung (*Semantik*) besitzen. Auf diese Weise findet *Informationsübertragung/-austausch* statt.

Der Empfänger kann die *Semantik* des Satzes „entschlüsseln“, da er die Bedeutungen der einzelnen Wörter unter Berücksichtigung ihrer Anordnung im Satz kennt. „Bitte nehmen Sie Platz.“ führt dazu, dass sich der Angesprochene dankend auf den entsprechenden Stuhl begibt. Offenbar führt die *Interpretation* eines Satzes zur Feststellung einer Bedeutung und ggf. der Ausführung zugehöriger Aktionen.

Neben der Interpretation muttersprachlicher Sätze geht es auch um solche einer Fremd- oder einer Fachsprache. Heute verwenden viele Menschen die englische Sprache. Sie sind in der Lage, englischsprachige Texte zu lesen. Vielen Musiker*innen reicht ein Notenblatt, um die darauf notierte Melodie zu singen. Elektrotechniker*innen sind so ausgebildet, dass sie einen Schaltplan lesen und die darin angegebenen Kabelverbindungen vornehmen können.

Trotz der Kenntnis aller semantischer Details eines korrekt gebauten Satzes kann eine Satzinterpretation problematisch sein. Eine Quelle für Missinterpretationen sind unklare reflexive Bezüge, wie in: „Verena kaufte noch eine Vase. Sie war wieder einmal blau“. Sogar Definitionen können Missinterpretationen hervorrufen: „Ein Junggeselle ist ein Mann, dem zum Glück die Frau fehlt“. „Soll ich diesen Baum umfahren?“ Die wirklich zutreffende Interpretation ist dann nur mit Kenntnis des Kontextes (Mimik, Gestik, Tonfall, Situation) möglich und bleibt wohl auch stark subjektiv. Neben *Syntax* und *Semantik* spielt also auch die *Pragmatik* (kontextabhängige und nicht-wörtliche Bedeutungen) eine Rolle.

Nicht in jedem Fall ist es möglich, Sätze einer Sprache unmittelbar zu interpretieren. Europäer denken an Texte aus dem asiatischen Raum, die beispielsweise in chinesischer oder japanischer Sprache verfasst wurden. Um sie weltweit zugänglich zu machen, werden sie in die englische Sprache übertragen.

Diese Übersetzung ist keine Interpretation, sondern eine *Compilation*. Im Unterschied zur *Interpretation*, die den Text portionsweise bewertet und zugehörige Aktionen bewirkt, erzeugt eine *Compilation* einen neuen Text, d.h. die Folge übersetzter Sätze oder Wörter.

| |
|----------------|
| Sprache |
| Syntax |
| Semantik |
| Interpretation |
| Pragmatik |

1.2 Sprachübersetzer: Interpreter und Compiler

Beschreibungs- sprachen
API
Syntax
Semantik
Pragmatik

Die in Abschnitt 1.1 angestellten Überlegungen zur Übersetzung (Translation) natürlicher Sprachen wollen wir nun auf die „Computerwelt“ übertragen. Darin gibt es zwar keine einzige natürliche Sprache aber eine große Zahl von *Programmier-, Fach-, Beschreibungssprachen* und *Schnittstellen* für die Anwendungsprogrammierung (*API*). Diese „künstlichen“ Sprachen haben ebenso wie natürliche eine *Syntax* (grammatikalische Regeln), eine *Semantik* (Bedeutung) und eine *Pragmatik* (Beziehung der Zeichen zum Leser). Um diese Sprachen einsetzen zu können, muss man sie erlernen, d.h. sich mit diesen drei Bereichen beschäftigen.

Maschinencode
Betriebssystem

Hat ein Computer eine „Muttersprache“? Die schnelle und grundsätzlich korrekte Antwort lautet: Ja, den *Maschinencode*. Dieser wird vor allem von dem verwendeten *Betriebssystem* bestimmt, setzt auf der Hardware-Ebene (Befehlssatz des Prozessors) auf und repräsentiert interpretierbare 0-1-Folgen.

FLACI
JavaScript (JS)

virtuellen JS-Maschine

Unser *FLACI*¹-System, mit dem wir im Folgenden eine Reise in das Gebiet der Formalen Sprachen, abstrakten Automaten, Compiler und Interpreter antreten werden, „versteht“ *JavaScript* (JS) als Muttersprache, erkennbar an dem kleinen schwarzen Dreieck, das an einen in JS vorliegenden Baustein (automatisch) angefügt wird, s. Abbildung 1.1. Die Interpretation des JS-Codes erledigt FLACI für uns, was die Modellierung vereinfacht. Da aktuelle Web-Browser und Systeme wie NodeJS eine Reihe von JS-Interpretern, JS-engines und weitere JS-Software bereitstellen, ist die gedankliche Vorstellung von einer sog. *virtuellen JS-Maschine* sehr realistisch.

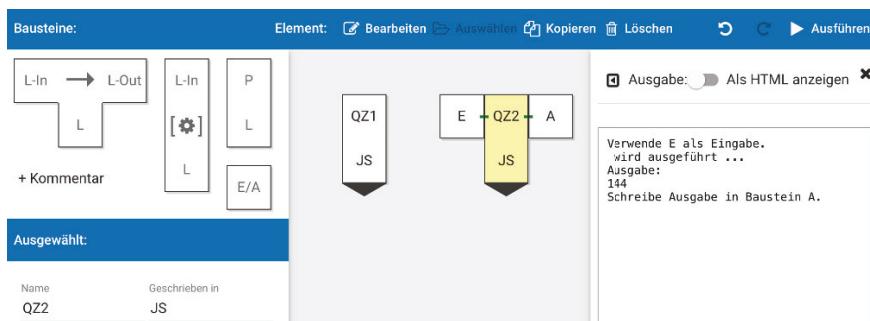


Abbildung 1.1: Interpretation von JS-Programmen (ohne/mit Eingabe) mit dem FLACI-Modul „Compiler und Interpreter“

Links im mittleren Teil (Modellierungsbereich) in Abbildung 1.1 wurde ein sehr einfaches in JS geschriebenes Programm QZ1 mit `var a=7; return a*a;` zur Quadratzahlberechnung modelliert. Das unten am Rechteck angesetzte schwarze

¹FLACI-Webanwendung: <https://flaci.com>

Dreieck symbolisiert, dass ein in JS geschriebenes Programm in FLACI *unmittelbar* interpretiert werden kann. Führt man es aus, liefert QZ1 das Ergebnis 49.

Um die gleiche Berechnung für eine wählbare Zahl (dem Wert des `input`-Parameters) vornehmen zu können, erhält das JS-Programm QZ2 mit `var a=input; return a*a;` rechts in Abbildung 1.1 als Eingabe (in dem angesetzten Eingabe-Öhrchen E) die Zahl 12 und liefert 144 im Ausgabe-Öhrchen A.

input

Computerübung 1.1

Verwenden Sie FLACI: <https://flaci.com> und registrieren Sie sich. Öffnen Sie den Bereich „Compiler und Interpreter“ und stellen Sie die in Abbildung 1.1 dargestellte Situation exakt nach. Die Bedienung ist intuitiv: Die Bausteine ziehen Sie mit der Maus in den Modellierungsbereich. Die Eigenschaften „Name“ und „Geschrieben in“ ändern Sie durch Auswahl mit dem Cursor und Texteingabe. Dann brauchen Sie noch „Bearbeiten“ und „Ausführen“. Das Speichern der von Ihnen vorgenommenen Veränderungen geschieht automatisch.



Im Folgenden betrachten wir eine *Musiksprache* ML. Abbildung 1.2 zeigt die seit langem übliche *grafische* Notensprache. „ML-Programme“ sind *textbasiert*, so wie die JS-Programme QZ1 und QZ2.

Musiksprache
(ML)

Abbildung 1.2: Notenblatt für den Song aus Abbildung 1.3

ML ist für FLACI keine Muttersprache. Ein Satz in dieser Sprache, also ein Lied, wie etwa in Abbildung 1.3 rechts. Es ist selbst Eingabe (aufgesetztes Rechteck) für einen mit FLACI entwickelten *Interpreter*, der es hörbar macht. Ein solcher ML-Interpreter muss deshalb in JS geschrieben, d.h. als JS-Programm, vorliegen.

Interpreter



G0-4 E0-4 E0-2 F0-4 D0-4 D0-2
C0-4 D0-4 E0-4 F0-4 G0-4 G0-4
G0-2 G0-4 E0-4 E0-2 F0-4 D0-4
D0-2 C0-4 E0-4 G0-4 G0-4 C0-2

Abbildung 1.3: Anwendung eines in JS geschriebenen ML-Interpreters (links) auf ein in ML geschriebenes Lied (links oben und rechts)

Abbildung 1.3 illustriert diese Interpretation. In Kapitel 5 wird der Herstellungsprozess des Interpreters betrachtet und in FLACI modelliert.

Compiler

Offensichtlich ist ein *Interpreter* ein Übersetzer, der „portionsweise“ (z.B. Note für Note) zugehörige Aktionen auslöst, jedoch *keinen* neuen Text erzeugt. Dies ist der wesentliche Unterschied zu einem *Compiler*, der einen Text (Satz einer Sprache; Programm) entgegennimmt und daraus einen (neuen) Text, d.h. das komplette Übersetzungsergebnis, erzeugt. In Kapitel 5 wird der Herstellungsprozess des Compilers, der ML in die Notenblattdarstellung (wie in Abbildung 1.2) übersetzt, betrachtet und in FLACI modelliert. Doch ganz gleich wie viele Compilations-schritte über beliebig viele Zwischensprachen stattfinden, ohne abschließende Interpretation geht es nicht.

Ganz analog zur Musiksprache in Abbildung 1.3 lässt sich die Interpretation eines in der Sprache EN (englisch) vorliegenden Buch-Textes, wie in Abbildung 1.4 links, mit FLACI modellieren: ML wird durch EN ersetzt und Lied durch Buch.

Auch wenn wir mit FLACI für das Beispiel aus Abbildung 1.4 keine ausführbare Lösung angeben können, ist die Modellierung recht aussagekräftig. Links und rechts unten erkennt man die bekannten Interpretationen, diesmal für ein englisch- bzw. (nach Übersetzung) deutschsprachiges Buch.

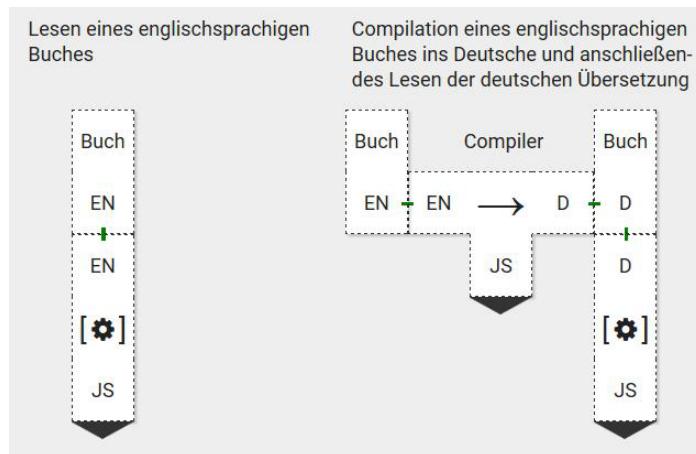


Abbildung 1.4: Buchübersetzung und Interpretation

T-Diagramm

Die an ein T erinnernde Gestalt eines Diagrammbausteins für einen Compiler (rechts in Abbildung 1.4) enthält drei Informationen:

Quellsprache

- die zu übersetzende *Quellsprache* – links im T-Diagramm: EN für englisch,

Zielsprache

- die *Zielsprache*, in die übersetzt wird – rechts: D für deutsch,

Implementations-sprache

- die *Implementationssprache*, d.h. die Sprache, in der der Compiler geschrieben wurde – unten im Sockel des T's: JS für JavaScript.

Die schwarzen Dreiecke lassen uns schnell erkennen, welche Programme/Texte

ausführbar vorliegen und welche nicht. Der Pfeil im T-Diagramm unterstreicht optisch die Übersetzungsrichtung.

Wir könnten die „musikalische Verwirrung“ noch etwas weiter treiben: Compiliert man das Lied, also den in ML geschriebenen Text, in die Sprache QR (Quick Response), verstummt selbst der letzte Musikfreund. Abbildung 1.5 zeigt links den Übersetzungsprozess und rechts das Ergebnis für den konkreten Song.

QR-Code



Abbildung 1.5: Lied als Text in ML compiliert in QR-Code

Diese Compilation kann mit FLACI ausgeführt werden. Ein TXT→QR-Compiler steht in FLACI zur Verfügung. Wir benennen ihn einfach um zu ML→QR-Compiler, ein Verfahren, mit dem wir später nicht mehr zufrieden sein werden.

Computerübung 1.2

Verwenden Sie Ihr Handy, um festzustellen, was der QR-Code in Abbildung 1.5 enthält.



Computerübung 1.3

Öffnen Sie einen neuen FLACI-Modellierungsbereich (Compiler und Interpreter) und kopieren Sie den TXT→QR-Compiler, QR-Code genannt, aus der öffentlichen Beispielsammlung dort hinein. Benennen Sie ihn um zu ML→QR-Compiler. Der Compiler nimmt einen Text in ML und generiert QR-Code. Der QR-Code ist selbst ein Text aus schwarzen und weißen Rechtecken. Er kann aus höchstens ca. 3000 Zeichen bestehen. Halten Sie einfach Ihr Smartphone über den QR-Code und starten Sie den QR-Scanner. Dann werden Sie rasch den in Abbildung 1.3 rechts angegebenen Song in ML wiedererkennen.



Natürlich hat die Notenschreibweise wie in Abbildung 1.2 eine Reihe von Vorteilen gegenüber der textuellen Notation aus Abbildung 1.3. Deshalb ist auch die Compilation von ML in die Noten-Notation (SVG) sinnvoll, s. Abbildung 1.6.

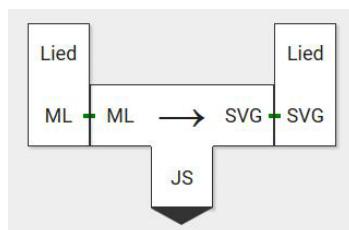
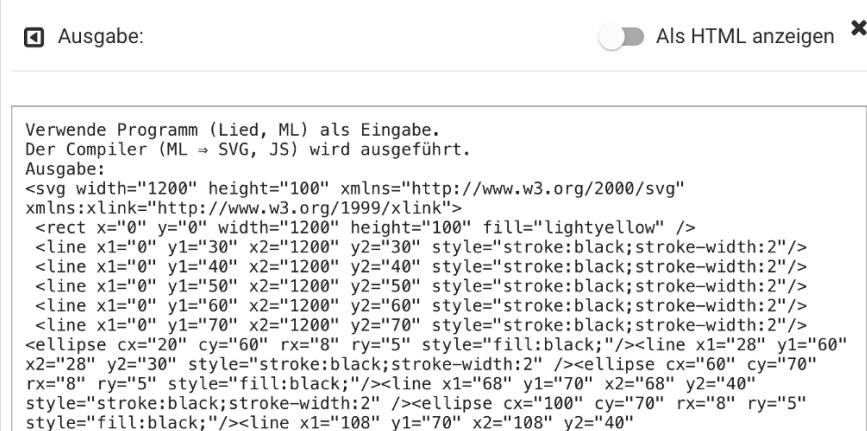


Abbildung 1.6: ML→SVG-Compiler

SVG SVG ist ein populäres Grafikformat (Vektorgrafik), mit dem wir auch Noten sehr gut darstellen können. Auch hier verschieben wir die Frage nach der Herkunft des in JS geschriebenen (und damit in FLACI lauffähigen) ML→SVG-Compilers.

Für das ausgewählte Lied ergibt sich ein Text in SVG wie in Abbildung 1.7. Es ist kein in FLACI lauffähiges JS-Programm – das schwarze Dreieck fehlt in Abbildung 1.6, rechts, unter SVG. Die Interpretation, z.B. per Schalter im Webbrowser, der SVG versteht, s. Abbildung 1.7, liefert das Notenblatt aus Abbildung 1.2.



The screenshot shows a web page with a text input field containing SVG code. Above the input field are two buttons: 'Ausgabe:' and 'Als HTML anzeigen'. The input field contains the following text:

```

Verwende Programm (Lied, ML) als Eingabe.
Der Compiler (ML → SVG, JS) wird ausgeführt.
Ausgabe:
<svg width="1200" height="100" xmlns="http://www.w3.org/2000/svg"
xmlns:xlink="http://www.w3.org/1999/xlink">
<rect x="0" y="0" width="1200" height="100" fill="lightyellow" />
<line x1="0" y1="30" x2="1200" y2="30" style="stroke:black;stroke-width:2"/>
<line x1="0" y1="40" x2="1200" y2="40" style="stroke:black;stroke-width:2"/>
<line x1="0" y1="50" x2="1200" y2="50" style="stroke:black;stroke-width:2"/>
<line x1="0" y1="60" x2="1200" y2="60" style="stroke:black;stroke-width:2"/>
<line x1="0" y1="70" x2="1200" y2="70" style="stroke:black;stroke-width:2"/>
<ellipse cx="20" cy="60" rx="8" ry="5" style="fill:black;"/><line x1="28" y1="60" x2="28" y2="30" style="stroke:black;stroke-width:2" /><ellipse cx="60" cy="70" rx="8" ry="5" style="fill:black;"/><line x1="68" y1="70" x2="68" y2="40" style="stroke:black;stroke-width:2" /><ellipse cx="100" cy="70" rx="8" ry="5" style="fill:black;"/><line x1="108" y1="70" x2="108" y2="40" style="stroke:black;stroke-width:2" />
```

Abbildung 1.7: SVG-Darstellung von Lied (Ausschnitt), SVG-Interpretation per Schalter

1.3 Definition von Sprachen

Um Sprachen, wie z.B. ML, TXT, QR und SVG, übersetzen zu können, müssen sie strikt definiert sein. Dies beschränken wir zunächst nur auf die Syntax, also die grammatischen Korrektheit.

Für eine Sprache, die nur 10 oder 20 Sätze umfasst, ist der Definitionsauwand nicht sehr hoch: Man listet alle diese Sätze einfach auf. Um festzustellen, ob ein vorgelegter Satz zu einer solchen Sprache gehört oder nicht, vergleicht man ihn 1:1 mit denen auf der Liste. Gibt es eine Übereinstimmung, ist der Satz syntaktisch korrekt und gehört zur Sprache, anderenfalls nicht.

endliche Sprache

Solche Sprachen, die eine feste Anzahl von Sätzen umfassen, nennen wir *endlich*. In der Tat gibt es endliche Sprachen mit praktischer Relevanz, wie etwa die der Uhrzeiten, die der Kfz-Kennzeichen oder die der Postleitzahlen. Im Falle der am 1. Juli 1993 in Deutschland eingeführten 5-stelligen Postleitzahlen umfasst die

(endliche) Sprache höchstens² 100000 Elemente, nämlich 00000, ..., 99999.

Wenn man nun feststellen möchte, ob 02826 eine syntaktisch korrekte deutsche Postleitzahl ist, ist ein manueller Vergleich mit jedem Element der (sortierten) Liste durchaus zumutbar. Die Entscheidung kann man aber viel schneller treffen, wenn alternativ *syntaktische Regeln* anwendet:

Prüfe für die vorgelegte Zeichenkette, ob sie genau die Länge 5 hat und jedes Zeichen, aus denen sie besteht, aus {’0’, ’1’, ’2’, ..., ’9’} stammt. Dabei ist es wichtig an *Wörter* und *nicht* an Ziffern, Zahlen und Arithmetik zu denken. Denn bei Zahlendarstellungen lässt man gern die Vornullen weg, was im Beispiel für 2826 zu einem Fehler führt, der in der Tat gelegentlich vorkommt.

Für Sprachen mit unendlich vielen Elementen gibt es gar keine andere Möglichkeit, als *grammatikalische Regeln* anzuwenden. Dies kennt man vom Fremdsprachenunterricht. Aber auch ML, TXT, Java, JS und SVG sind *unendliche Sprachen*, in denen man immer wieder neue Sätze bilden kann.

Wie im Sprachenunterricht werden wir die syntaktischen Regeln als Grammatik formulieren.

In Abgrenzung zu natürlichen Sprachen betrachten wir in der theoretischen Informatik *formale Sprachen*, also solche, die mit *formalen Grammatiken* definiert werden können. Einfach strukturierte Sprachen lassen sich mit *regulären Ausdrücken* beschreiben. Außerdem werden wir *abstrakte Automaten* verwenden, um die syntaktische Korrektheit vorgelegter Zeichenketten festzustellen.

Erst im Zusammenhang mit dem Übersetzungsprozess (*Compilerbau*) kommen semantische Aspekte hinzu.

Im Folgenden befassen wir uns mit den Grundbegriffen der Theorie der formalen Sprachen und der Automatentheorie.

1.4 Alphabet und Zeichen

Formale Sprachen enthalten Wörter³. Wörter bestehen aus Zeichen. Sämtliche Zeichen eines Wortes stammen aus genau einem sich nicht erschöpfenden *Alphabet*.

Damit haben wir bereits drei wichtige Grundbegriffe der Theorie der forma-

²Dass davon derzeit weniger als ein Drittel vergeben sind, spielt bei der syntaktischen Beurteilung keine Rolle.

³Wenigstens im Deutschen besitzt das Wort „Wort“ zwei Pluralformen, nämlich „Wörter“ und „Worte“. Wörter stehen ohne Bezug auf ihren Sinn-Zusammenhang. Man kann also mit vielen Wörtern nichts sagen. Hingegen spricht man von *Worten*, wenn damit Gedanken oder Gefühle ausgedrückt werden. Beispiele: die groß geschriebenen Wörter, Gottes Wort, geflügelte Worte. Er sprach ein paar andachtsvolle Worte. Bei formalen Sprachen spielt die Bedeutung von Worten keine Rolle. Folglich benutzen wir den Plural „Wörter“.

syntaktische
Regeln

grammatikalische
Regeln
unendliche
Sprachen

formale Sprachen
Grammatik
abstrakte
Automaten

Compilerbau

formale Sprachen
Wörter
Zeichen
Alphabet

len Sprachen angesprochen. Es folgen die zugehörigen Definitionen und einige Überlegungen zum konstruktiven Umgang mit Zeichen, Wörtern und Sprachen.



Definition 1.1

Ein *Alphabet* ist eine beliebige *endliche, nichtleere* Menge. Die Elemente dieser Menge heißen *Zeichen*.



Beispiel 1.1

Die folgenden Mengen sind Alphabete. Auf einige von ihnen werden wir uns im Folgetext mehrfach beziehen.

$$A_1 = \{a, b, c, \dots, z\}$$

$$A_2 = \{(,), +, -, *, /, a\}$$

$$A_3 = \{\text{begin}, \text{end}, \text{for}, \text{while}, \text{do}, \text{repeat}, \text{until}\}$$

$$A_4 = \{\triangle, \triangleleft, \triangleright\}$$

$$A_5 = \{ \begin{array}{c} A B C D E F G H \\ I J K L M N O P Q \\ R S T U V W X Y Z \\ 1 2 3 4 5 6 7 8 9 0 \end{array} \}$$

$$A_6 = \{ \begin{array}{c} \text{red flower} \\ \text{green sphere} \\ \text{yellow triangle} \\ \text{blue four-pointed star} \\ \text{purple heart} \\ \text{grey crescent} \\ \text{gold eight-pointed star} \\ \text{green hexagon} \end{array} \}$$

$$A_7 = \{ \begin{array}{c} \text{ability} \\ \text{CHANGE} \\ \text{Success} \\ \text{artists} \\ \text{DREAM} \\ \text{ATTENTION} \\ \text{Awareness} \\ \text{create} \\ \text{bright} \\ \text{creativity} \\ \text{THINKING} \\ \text{RECOGNITION} \\ \text{cornucopia} \\ \text{Art} \\ \text{Commitment} \end{array} \}$$

Man beachte insbesondere bei A_3 und A_7 , dass es sich um „atomare“ Zeichen handelt, die *nicht* etwa in einzelne Tastaturzeichen zerfallen. Man kann sie sich als beschriftete Kärtchen vorstellen.



Didaktischer Hinweis 1.1

Alphabetzeichen (und Terminale, wie in Abschnitt 1.9) haben wir mit einem Rahmen umgeben bzw. durch eine Typewriter-Schrift gekennzeichnet. Dies werden wir in den folgenden Kapiteln nicht konsequent durchhalten, weil es die Darstellung nicht in jedem Falle übersichtlicher macht. Es ist aber sehr wichtig, die entsprechenden Symbole (Alphabetzeichen und Terminale) als *atomare* Zeichen zu verstehen.



Computerübung 1.4

Öffnen Sie das FLACI-Modul „Formale Sprachen“ und definieren Sie in „Alphabet und Zeichen“ das dort selbst definierbare Alphabet A_6 so, dass es mit A_2 aus Beispiel 1.1 übereinstimmt, s. Abbildung 1.8.

alternative
Alphabet-
Definition

In mancher Literatur wird ein Alphabet auch als eine *geordnete* Menge von Zeichen, also als ein *Singletupel*, definiert. Das hat Vorteile für die Sortierung von Wörtern, die wir in Abschnitt 1.6 benötigen. Im Allgemeinen ist es jedoch nicht erforderlich, eine Ordnungsrelation als definierende Eigenschaft eines Alphabets anzusehen. Deshalb definieren wir Alphabete als Mengen, s. Definition 1.1.

Formale Sprachen

Alphabet und Zeichen Wort Wortmenge Sprache

i Ein Alphabet A ist eine endliche, nichtleere Menge von Zeichen.

Wählen Sie eines der Beispiel-Alphabete zum Experimentieren aus.

$A_1 = \{ 0, 1 \}$
 $A_2 = \{ a, b, c, \dots, z \}$
 $A_3 = \{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 \}$
 $A_4 = \{ \text{💻}, \text{☁️}, \text{👁️}, \text{⚠️}, \text{❤️}, \text{★} \}$
 $A_5 = \{ \text{begin}, \text{end}, \text{for}, \text{while}, \text{do}, \text{repeat}, \text{until} \}$
 $A_6 = \{ (,), +, -, *, /, a \}$

Alphabetzeichen (mit Komma getrennt eingeben)
 $(,), +, -, *, /, a$

WEITER ZUM WORT

Abbildung 1.8: Definition eines Alphabets

1.5 Wort, Wortlänge und Verkettung

Definition 1.2

Irgendeine (auch leere) Zeichenkette ist ein *Wort*. Man sagt: Eine Zeichenkette „ w “ ist ein Wort über dem Alphabet A , wenn sämtliche Zeichen von w aus A stammen.
 Das Symbol für das *leere Wort* "" ist ϵ .



Didaktischer Hinweis 1.2

Analog zur Vorgehensweise in der Mathematik, wo man die leere Menge $\{\}$ mit \emptyset bezeichnet, symbolisieren wir das leere (zeichenlose) Wort "" mit ϵ .
 Wird es in Operationen benutzt, muss man mit "" arbeiten.



Beispiel 1.2

"izabbix", "endbegin", s. Abbildung 1.9, und "b" sind Wörter über $A = \{a, b, \dots, z\}$, denn sie enthalten ausschließlich Zeichen aus A . Dabei spielt es keine Rolle, wie oft ein



ausgewähltes Zeichen im Wort verwendet wird. "i#jahs" ist kein Wort über A , weil es mit '#' ein Zeichen enthält, das nicht zu A gehört.

The screenshot shows a user interface for generating words over an alphabet. At the top, there's a navigation bar with tabs: 'Alphabet und Zeichen', 'Wort' (which is selected), 'Wortmenge', and 'Sprache'. On the right side of the header, there are icons for user login ('ANMELDEN') and language selection ('DE').

In the main content area, there's a large input field containing the word "endbegin". Above this field, a tooltip provides the definition of a word over an alphabet: "Ein Wort w über dem Alphabet A ist eine Zeichenkette, die ausschließlich aus Zeichen aus A besteht. Das leere Wort $\epsilon = ""$ ist ein Wort über jedem beliebigen Alphabet." Below the input field, another tooltip states: "Klicken Sie wiederholt auf Zeichen des Alphabets A , um ein Wort w zu konstruieren." The input field itself has a placeholder "Alphabet A = {a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z}" and contains the word "endbegin".

Below the input field, there's a message: "Wort $w = \boxed{e} \boxed{n} \boxed{d} \boxed{b} \boxed{e} \boxed{g} \boxed{i} \boxed{n}$ ". Underneath this message, another tooltip says: "Klicken Sie auf Zeichen von w , um diese zu entfernen." Further down, another tooltip explains word length: "Die Länge eines Wortes w , kurz: $|w|$, ist bestimmt durch die Anzahl aller Zeichen, die das Wort enthält. Mehrfachvorkommen werden entsprechend mehrfach gezählt." At the bottom left, it shows "Wortlänge $|w| = 8$ ". At the bottom right, there's a blue button labeled "WEITER ZUR WORTMENGE".

Abbildung 1.9: Wort über A_1 aus Beispiel 1.1

Eine vorgegebene Zeichenkette $z = z_1 z_2 \dots z_n$ ist genau dann ein Wort über dem Alphabet A , wenn für *jedes* einzelne Zeichen z_i gilt: $z_i \in A$. Damit ist auch klar, dass die Zeichenkette " ϵ " ein Wort – das *leere Wort* – über (jedem) Alphabet A ist. ϵ enthält kein einziges unzulässiges Zeichen.

Ist das Wort "endbegin" über $A_1 = \{a, b, c, \dots, z\}$, s. Abbildung 1.9, eigentlich auch ein Wort über $A_3 = \{\text{begin, end, for, while, do, repeat, until}\}$? Nein, denn "endbegin" ist ein Wort über A_1 und besteht folglich aus den Zeichen b, d, e, g, i und n. Diese Zeichen gibt es in A_3 nicht.

Das optisch davon ununterscheidbare Wort "endbegin" kann natürlich über A_3 gebildet werden. Es besteht aus den Zeichen begin und end, s. Abbildung 1.10.

Bei der Analyse von Wörtern muss also stets das zugrunde liegende Alphabet mit angegeben werden.

Notation von Wörtern

Wörter werden in Anführungszeichen eingeschlossen. Dies entspricht unseren Notationsgewohnheiten für Zeichenketten. Wenn keine Verwechslungsmöglichkeit besteht, können die Anführungszeichen entfallen: "hallo" oder hallo beschreiben also ein und dasselbe Wort über A_1 . 'a' und "a" stehen jedoch für zwei völlig verschiedene Dinge, nämlich das Zeichen 'a' bzw. das „einzeichige“ Wort "a".

Klicken Sie wiederholt auf Zeichen des Alphabets A, um ein Wort w zu konstruieren.

Alphabet A = { begin , end , for , while , do , repeat , until }

Wort w = "endbegin"

Klicken Sie auf Zeichen von w, um diese zu entfernen.

Abbildung 1.10: Wort über A_3 aus Beispiel 1.1

Auch in Programmiersprachen gilt diese Unterscheidung.

Didaktischer Hinweis 1.3

Leider sind hier immer wieder Fehler zu beobachten: Weder das leere noch irgendein anderes Wort dürfen in einem Alphabet vorkommen.



Jedes Wort hat eine bestimmte Länge. Würde man zur Definition des Begriffs der *Wortlänge* beispielsweise die Anzahl der paarweise verschiedenen Zeichen eines Wortes verwenden, wäre das höchst unpraktikabel: "abbacaba" und "cab" wären gleichlang. Wir verwenden die folgende Definition.

Wortlänge

Definition 1.3

Die *Länge eines Wortes w*, kurz: $|w|$, ist bestimmt durch die Anzahl aller Zeichen, die das Wort enthält. Dabei werden Mehrfachvorkommen entsprechend ihres Auftretens mehrfach gezählt.



Beispiel 1.3

Das in Abbildung 1.10 angegebene Beispielwort hat die Länge 2.



Wir wenden uns nun der *Wortbildung* zu. Um ein Wort über einem gegebenen Alphabet zu erzeugen, verwendet man eine sehr einfache Operation: die *Verkettung* oder *Konkatenation*. Gern schreibt man dafür das Symbol \circ (Kringel). Um beispielsweise das Wort aba über dem Alphabet {a, b, c} zu bilden, geht man folgendermaßen vor: Man wählt zunächst das Zeichen 'a' aus dem Alphabet und verkettet es mit dem Zeichen 'b', d.h. 'a' \circ 'b' = "ab". Durch die Verkettung zweier Zeichen entsteht ein Wort.

Wortbildung

Als nächstes erwarten wir, dass die Verkettungsoperation auf ein Wort (erster Operand) und ein Zeichen (zweiter Operand) anwendbar ist: "ab" \circ 'a' = "aba". Wieder entsteht ein Wort. Dies ist auch der Fall, wenn zwei Wörter über einem bestimmten Alphabet miteinander verkettet werden: "aste" \circ "rix" = "asterix".

Verkettung

Computerübung 1.5

In FLACI ist die Wortbildung sehr bequem: Das an der jeweiligen Position gewünschte Alphabetzeichen wird einfach angeklickt. Davon haben Sie bereits Gebrauch gemacht. Bilden Sie nun einige Wörter über einem beliebigen von Ihnen vorgegebenen Alphabet und bestimmen Sie die zugehörigen Wortlängen.



1.6 Wortmenge

Definition 1.4

Die Menge aller Wörter über A nennt man die *Wortmenge* A^* . Das leere Wort ϵ gehört zu jeder Wortmenge.



Beispiel 1.4

Sei $A = \{1\}$. Dann ist $A^* = \{\epsilon, 1, 11, 111, \dots\}$, wie in Abbildung 1.11 dargestellt, die Wortmenge der „Bierdeckelnotation“.

The screenshot shows a navigation bar with 'Formale Sprachen', 'ANMELDEN', and 'DE'. Below it, tabs for 'Alphabet und Zeichen', 'Wort', 'Wortmenge' (which is underlined), and 'Sprache' are visible. A note says: 'Die Menge aller Wörter über A nennt man die Wortmenge A^* . A^* ist eine abzählbar unendliche Menge und lässt sich systematisch notieren. Das leere Wort ϵ gehört zu jeder Wortmenge.' Below this, it shows 'Alphabet $A = \{1\}$ ' and a list of words: ' $A^* = \{\epsilon, 1, 11, 111, 1111, 11111, 111111, 1111111, 11111111, 111111111, \dots\}$ '. A button at the bottom says 'WEITERE ELEMENTE ANZEIGEN'.

Abbildung 1.11: Wortmenge über A aus Beispiel 1.4

Wie viele Elemente besitzt A^* für ein beliebiges Alphabet A mit $|A| = n$ Elementen? A^* ist stets eine *unendliche* Menge: Für jedes Wort aus A^* kann man ein weiteres Wort durch Verkettung bilden. Wie wir in Beispiel 1.4 gesehen haben, reicht schon ein Alphabet mit nur einem einzigen Zeichen aus, um damit unendlich viele Wörter zu erzeugen.

(A^*, \circ)

Innerhalb von A^* hat \circ die Eigenschaft, durch Anwendung auf je zwei beliebige Wörter aus A^* ein Wort zu erzeugen, das (ganz sicher) zu A^* gehört. Man sagt, A^* ist bezüglich \circ abgeschlossen. Das Verknüpfungsgebilde (A^*, \circ) hat einige (leicht beweisbare) Eigenschaften. Einige davon sind Gegenstand der folgenden Übung.



Übung 1.1

Bearbeiten Sie die folgenden Aufgaben in der angegebenen Reihenfolge.

- (a) Beweisen Sie die folgenden Eigenschaften der Operation „Verkettung“ in der Menge aller Wörter über einem gegebenen Alphabet A .

Für $u, v \in A^*$ gelten: $|u \circ v| = |u| + |v|$ und $|u^n| = \underbrace{|u \circ u \circ \dots \circ u|}_{n \text{ mal}} = n \cdot |u|$, mit $n \in \mathbb{N}$.

- (b) Argumentieren Sie für oder gegen die Gültigkeit der folgenden Eigenschaften der Verkettung:

- Abgeschlossenheit, d.h. für alle $x, y \in A^*$ gilt $x \circ y = xy \in A^*$.
 - Assoziativität, d.h. für alle $x, y, z \in A^*$ gilt $(x \circ y) \circ z = x \circ (y \circ z) = xyz$.
 - Kommutativität, d.h. für alle $x, y \in A^*$ gilt $x \circ y = y \circ x$.
 - Neutrales Element, d.h. es gibt ein $e \in A^*$ mit $e \circ x = x \circ e = x$.
- (c) Bilden Sie vier Wörter, die zu A^* gehören, und geben Sie zwei Wörter an, die nicht zu A^* gehören. Dabei ist $A = \{(), +, -, *, /, a\}$.
- (d) Ermitteln Sie die Anzahl der Wörter über einem Alphabet mit n Zeichen, deren Länge k nicht überschreitet.

Die Bildung von Wörtern über einem einelementigen Alphabet, wie in Beispiel 1.4, ist sehr einfach. Umfasst das Alphabet mehrere Elemente, wie etwa $A = \{a, b, c\}$, sollte man die Wortmenge systematisch notieren, um den Überblick nicht zu verlieren. Beispielsweise kann man die Wörter in Gruppen aufsteigender Wortlänge anordnen.

Beispiel 1.5

$$A = \{a, b, c\}. A^* = \underbrace{\varepsilon}_{A^0}, \underbrace{a, b, c}_{A^1}, \underbrace{aa, ab, ac, ba, bb, bc, ca, cb, cc}_{A^2}, \text{aaa}, \dots$$



Die gesamte Wortmenge über A setzt sich aus den jeweils endlich vielen Wörtern dieser unendlich vielen disjunkten Teilmengen A^i zusammen:

$$A^* = \bigcup_{i \in \mathbb{N}} A^i.$$

Die natürliche Zahl i kann beliebig groß gewählt werden. Man nennt solche Mengen *potentiell unendliche* Mengen. Manchmal verwendet man auch

$$A^+ = A^* \setminus \{\varepsilon\}.$$

Zu beachten ist, dass $A^0, A^1, A^2, \dots, A^i, \dots$ *endliche* Mengen sind. Sie besitzen jeweils $|A|^i$ Elemente.

Bei der Anordnung der Elemente von A^* in Beispiel 1.5 sind wir sogar einen Schritt weiter gegangen: Neben der Gruppierung nach der Wortlänge wurden die Wörter der einzelnen Gruppen zusätzlich lexikografisch⁴ angeordnet. Die Reihenfolge des Aufschreibs der Alphabetelemente (von links nach rechts) bestimmt deren aufsteigende Sortierung bezüglich der betrachteten Ordnungsrelation \prec . Für

Beispiel 1.5 bedeutet das: " a " \prec " b " \prec " c ". Fasst man beides zusammen, so ergibt sich die *längenlexikografische Ordnung*

$$\prec: A^* \times A^* \rightarrow \{\text{true, false}\} \text{ mit}$$

⁴Dies entspricht der alphabetischen Sortierung wie in einem Lexikon oder im Duden.

unendlich viele
endliche und
disjunkte
Teilmengen

\prec
 lex

längenlexiko-
grafische
Ordnung

\prec
 llo

$$w_1 \underset{llo}{\prec} w_2 = \begin{cases} \text{true,} & \text{wenn } |w_1| < |w_2| \\ w_1 \underset{\text{lex}}{\prec} w_2, & \text{wenn } |w_1| = |w_2| \\ \text{false,} & \text{sonst} \end{cases}$$

abzählbar
unendliche
Menge

In Beispiel 1.4 hatten wir bereits vermutet, dass eine Wortmenge stets eine *abzählbar unendliche* Menge ist. Was ist darunter zu verstehen? Gibt es noch andere „Arten“ von Unendlichkeit?



Definition 1.5

Zwei Mengen M_1 und M_2 heißen *gleichmächtig*, wenn es eine bijektive Abbildung von M_1 auf M_2 gibt.

Ist eine der beiden Mengen die der natürlichen Zahlen \mathbb{N} , so ist M eine *abzählbar unendliche* Menge.

Abbildung 1.12 veranschaulicht die geforderte mathematische Abbildung.

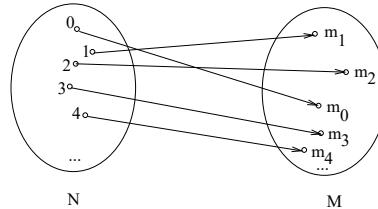


Abbildung 1.12: Bijektive Abbildung $f : \mathbb{N} \rightarrow M$



Didaktischer Hinweis 1.4

Auf die Existenz überabzählbar unendlicher Mengen sollte unbedingt hingewiesen werden. Während die Menge der natürlichen Zahlen \mathbb{N} eine abzählbar unendliche Menge ist, ist die Menge der reellen Zahlen \mathbb{R} nicht abzählbar, sondern *überabzählbar* unendlich. Es gibt also (sehr viel) mehr reelle Zahlen als natürliche.



Übung 1.2

Eine Funktion kann injektiv, surjektiv oder bijektiv sein. Wiederholen Sie diese Begriffe und drücken Sie diese Eigenschaften mit Mengenbeziehungen aus.

Die Elemente von M in Abbildung 1.12 könnte man mit m_0, m_1, m_2, \dots benennen. O.B.d.A.⁵ kann man eine bijektive Abbildung $f : \mathbb{N} \rightarrow M$ definieren, für die $f(i) = m_i$ gilt, wobei $i \in \mathbb{N}$. Dies legt die Idee nahe, die Elemente aus M einfach durchzumerken: $M = \{f(0), f(1), f(2), \dots\}$. Das Durchmerken wird durch eine Sortierung der Elemente erreicht, z.B. nach der Ordnungsrelation \prec_{llo} .

Damit ist der Beweis des folgenden Satzes gedanklich ausreichend vorbereitet.

⁵Ohne Beschränkung der Allgemeinheit

Satz 1.1

Die Menge A^* aller Wörter über A ist abzählbar unendlich.



Beweis

Die erforderliche bijektive Abbildung von \mathbb{N} auf A^* ergibt sich aus der Sortierung der Elemente von A^* nach der längenlexikografischen Ordnungsrelation \prec . □



Beispiel 1.6

Wir bestimmen $f(86)$ in der längenlexikografisch geordneten Wortmenge über $\{a, b, c, d\}$, beginnend mit $f(0)$: $|A| = 4$; $|A^0| = 1$, $|A^1| = 4$, $|A^2| = 16$, $|A^3| = 64$; $|A^0| + |A^1| + |A^2| + |A^3| = 85$. Wir suchen also das erste Wort aus A^4 . Das ist "aaaa", d.h. $f(86) = \text{aaaa}$, $f(87) = \text{aaab}$, $f(88) = \text{aac}$, s. Abbildung 1.13.

Alphabet $A = \{ a, b, c, d \}$

$$A^* = \{$$

}

Abbildung 1.13: A^* für $A = \{a, b, c, d\}$, angezeigt bis $f(96)$

1.7 Formale Sprache

Im Unterschied zu den in Abschnitt 1.1 betrachteten natürlichen Sprachen beschäftigt man sich in der theoretischen Informatik mit *formalen Sprachen*. Dabei geht es ausschließlich um die grammatischen – man sagt *syntaktischen* – Strukturen der Wörter⁶ einer Sprache.

formale Sprache

Im Unterschied zum Compilerbau (Praktische Informatik) spielt die *Semantik* (Bedeutung) hier keine Rolle. Dies gilt auch für die *Pragmatik*, d.h. die Untersuchung des zweckbestimmten Gebrauchs von Sprache und dessen, was sie bei Menschen bewirken kann. Die Pragmatik beschäftigt sich mit der Beschreibung von kontextabhängigen und nicht-wörtlichen Bedeutungen.

Semantik

Beispielsweise gehören die Wörter $w_1 = "ash::dstoi qqsq QU SQUHS !$
 $\ddot{u}; ;?W"$ und $w_2 = "a--t"$ lt. Definition 1.4 zur Wortmenge über dem deutschen

⁶Eine (beliebig komplexe) Zeichenkette nennt man in der theoretischen Informatik „Wort“ und im Compilerbau „Satz“.

Alphabet zzgl. einiger Interpunktionszeichen. Dies gilt auch für $w_3 = \text{"Die Ente frisst den Bär."}$, dessen Semantik uns nicht interessiert. Wir wissen bereits, dass sich über diesem Alphabet abzählbar unendlich viele Wörter bilden lassen.

Eine Voraussetzung für den erfolgreichen Einsatz von Compilern und Interpretern ist die syntaktische Korrektheit der zu verarbeitenden Sätze. Die theoretische Informatik liefert die Grundlagen der Syntaxanalyse und ist deshalb eng mit dem Compilerbau (Praktische Informatik) verbunden. Die Theorie der formalen Sprachen und die Automatentheorie versuchen die Frage zu beantworten, ob ein Wort w aus der Wortmenge A^* zur betrachteten Sprache L gehört oder nicht, d.h. entweder $w \in L$ oder $w \notin L$. Das ist ein Entscheidungsproblem.



Definition 1.6

Sei A ein Alphabet. Jede Teilmenge $L \subseteq A^*$ heißt *Sprache über A* .
(L erinnert an das englische Wort language für Sprache.)

Eine Sprache besteht also aus Wörtern. Die Definition lässt zwei extreme Sprachen zu: $L_1 = \emptyset$ und $L_2 = A^*$. Sowohl die leere Menge L_1 als auch die Allsprache L_2 sind wenig interessant: Dies sind gerade die beiden Sprachen, die kein einziges Wort enthält bzw. alle bildbaren Zeichenketten, da sie keinerlei Anforderungen an den Aufbau ihrer Wörter stellt.

Pünktchen-Notation

Sprachen sind entweder endliche oder (typischerweise) unendliche⁷ Mengen. In der Mathematik wird für unendliche Mengen die bekannte Pünktchen-Notation, wie bei $M = \{"aab", "bb", "bababa", "c", "ba", ...\}$ und auch in Abbildung 1.14, eingesetzt.

Klar, die Pünktchen deuten an, dass es sich um eine Menge mit unendlich vielen Elementen handelt. Aber wie lauten die nächsten fünf Elemente von M ? Wir müssen erkennen, dass sich diese deskriptive Repräsentationsform als unbrauchbar erweist, wenn wir ganz bestimmte Sprachen festlegen bzw. verarbeiten wollen.



Computerübung 1.6

Legen Sie eine Sprache $L \subset A^*$ für $A = \{a, b, c, d\}$ fest, indem Sie in FLACI („Formale Sprachen“) bestimmte Elemente aus A^* (durch Doppelklick) auswählen. Soll L eine unendliche Menge sein, müssen auch die Pünktchen übernommen werden. Diese stehen für die in A^* zum Zeitpunkt der Auswahl nicht angezeigten Elemente.

⁷Da die Grundmenge A^* eine abzählbar unendliche Menge ist, kann eine beliebige Teilmenge davon höchstens abzählbar unendlich sein.

Abbildung 1.14: Auswahl bestimmter Wörter für eine Sprache

Didaktischer Hinweis 1.5

Auch mit FLACI können die Einschränkungen, die sich aus der Pünktchen-Notation ergeben, nicht behoben werden. Die in A^* nicht angezeigten (unendlich vielen) Elemente können lediglich übernommen werden. Oft möchte man das nicht, sondern nur jedes zweite Element oder alle Elemente, die ein bestimmtes Zeichen enthalten. Dann wünscht man sich so etwas wie ein Filter, das man auf die Elemente dieser Teilmenge anwenden kann. Formale Grammatiken, die wir im nächsten Kapitel behandeln, können das Gewünschte leisten.



Übung 1.3

$$A_1 = \{0, 1, 2, 3, \dots, 9\}$$

$L_1 = \{a_1 a_2 \dots a_n \mid a_i \in A_1, \text{ für } 1 \leq i \leq n, n \in \mathbb{N}; a_1 \neq 0, \text{ falls } n \geq 2 \text{ gilt.}\}$ Welche Sprache beschreibt L_1 ?



Übung 1.4

Sei die Sprache über $A = \{a\}$, die aus allen Wörtern mit einer primzahligen Anzahl von a 's besteht. Geben Sie eine mathematische Notation dafür an.



1.8 Konkrete Syntax

Wir sind gedanklich bei der Syntax von Programmiersprachen und betrachten als Beispiel eine *Zuweisung*, wie sie in imperativen und objektorientierten Programmiersprachen typischerweise anzutreffen ist. Eine Zuweisung ist eine Anweisung, mit der eine Variable eine Wertbindung erfährt. Es entsteht ein Name-Wert-Paar.

In imperativen Sprachen gibt es dafür einen Operator, den man meist als `=`, `:=` oder `\leftarrow` notiert (Syntax). Mit $n \leftarrow n+1$ wird n dessen Nachfolger zugewiesen (Semantik). In JavaScript verwendet man dafür die Regel `Variable = Wert`, z.B.: `n = n + 1`. Damit wird der alte Wert der angegebenen Variable n verändert (mutiert).

Vergleicht man $n \leftarrow n+1$ und $n = n + 1$ miteinander, so stellt man fest, dass sich die *konkrete Syntax* in beiden Fällen – trotz übereinstimmender Semantik – deutlich unterscheidet. In der Praxis kommen `++n` bzw. `n++` oder `n += 1` hinzu. Welche Version sollte man bevorzugen?

Sprachdesign Die konkrete Syntax, also das *Design*, einer Programmiersprache ist demnach eine Frage der Pragmatik. Dabei können viele Gesichtspunkte eine Rolle spielen. Beispielsweise ist das Gleichheitszeichen `=` in Zuweisungen umstritten, weil es mit dem Vergleichsoperator „ist gleich“ verwechselt werden könnte. Dem wirkt man mit `==` bzw. `===` für das entsprechende Relationszeichen entgegen.

1.9 Regeln, (E)BNF und Syntaxdiagramme

MiniJavaScript Im Folgenden betrachten wir die Struktur von Programmen der (von uns erfundenen) Sprache MiniJavaScript. Der Name lässt erahnen, dass MiniJavaScript nur wenige Befehle kennt. Dem programmiererfahrenen Leser fällt sofort auf, dass es beispielsweise kein `else` gibt. Außerdem kennt MiniJavaScript nur die drei Variablen A, B und C, d.h. nur diese können in einem syntaktisch korrekten MiniJavaScript-Programm vorkommen.

Nim-Spiel Unser Programmbeispiel stellt eine Version des bekannten *Nim-Spiels* dar, das gegen den Computer gespielt werden kann. Zu Beginn des Spiels sind 20 Streichhölzchen vorhanden. Zwei Spieler nehmen abwechselnd ein, zwei oder drei Hölzchen weg. Derjenige Spieler, der das letzte Streichholz nehmen könnte, hat gewonnen; derjenige, der kein Streichholz vorfindet, verliert.

```
writeln(" __ Streichholzspiel __ ");
A = 20;
C = 1;
while (C == 1) {
    write("Es liegen noch:");
    writeln(A);
```

```

if (A == 0) {
    writeln("Du hast verloren ;-)); C = 0;
};

if (A > 0) {
    writeln("Wie viele willst du ziehen (1 2 3)?");
    B = prompt();
    if (B < 1) { writeln("Du schummelst ..."); C = 0; B = -1 };
    if (B > 3) { writeln("Du schummelst ..."); C = 0; B = -1 };
    if (B > A) { writeln("Du schummelst ..."); C = 0; B = -1 };
    if (B > 0) {
        A = A - B;
        write("Du ziehst ");
        writeln(B);
        write("Es liegen noch:");
        writeln(A);
        B = A;
        if (B > 10) { writeln("Ich ziehe 3"); A = A-3; B = -1 };
        if (B > 8) { writeln("Ich ziehe 2"); A = A-2; B = -1 };
        if (B > 3) { writeln("Ich ziehe 1"); A = A-1; B = -1 };
        if (B == 3) { writeln("Ich ziehe 3"); A = A-3; B = -1 };
        if (B == 2) { writeln("Ich ziehe 2"); A = A-2; B = -1 };
        if (B == 1) { writeln("Ich ziehe 1"); A = A-1; B = -1 };
        if (B == 0) {
            writeln("Ich habe verloren :-("); C = 0
        };
    };
};
};

```

Der Hauptzyklus `while (C == 1) {...}` wird solange durchlaufen, bis der Streichhölzchen-Pool vollständig aufgebraucht ist, oder der menschliche Spieler eine ungültige Anzahl von Hölzern gezogen hat. Die `if`-Bedingungen am Ende des Programmtextes bilden sozusagen die Intelligenz des Computerprogramms als Gegenspieler. Da es in unserem MiniJavaScript kein `else` gibt, behilft sich das Programm durch Zuweisung von `-1` an die Variable B nach einem Halbzug des Computers, um so ein weiteres Ziehen eines Hölzchens zu verhindern.

Da sich MiniJavaScript sehr stark am echten JavaScript orientiert, können wir MiniJavaScript-Programme in einem Web-Browser (wie Mozilla FireFox oder Google Chrome) abarbeiten lassen. Da jedoch die zwei Prozeduren *write* und *writeln* in JavaScript nicht definiert sind, müssen wir diese als Hilfsfunktionen extra bereitstellen:

Hinweise zur Programm- ausfhrung

```
<script language="javascript">
    function write(s) { document.write(s); }
    function writeln(s) { document.writeln(s+"<br>"); }
    ... Streichholzspiel - Programm Text ...
</script>
```

Browser Die folgenden Ausgaben im Browser protokollieren ein konkretes Spiel.



```
== Streichholzspiel ==
Es liegen noch:20
Wie viele willst du ziehen (1 2 3)?
Du ziehst 3
Es liegen noch:17
Ich ziehe 3
Es liegen noch:14
Wie viele willst du ziehen (1 2 3)?
Du ziehst 2
Es liegen noch:12
Ich ziehe 3
Es liegen noch:9
Wie viele willst du ziehen (1 2 3)?
Du ziehst 3
Es liegen noch:6
Ich ziehe 1
Es liegen noch:5
Wie viele willst du ziehen (1 2 3)?
Du ziehst 2
Es liegen noch:3
Ich ziehe 3
Es liegen noch:0
Du hast verloren ;-)
```

Das Nim-Spiel ist aus der Sicht der Spieltheorie durchaus reizvoll, da es eine vollständige Analyse der Gewinnsituationen ermöglicht. Uns interessiert hier jedoch lediglich die Syntax von MiniJavaScript-Programmen.

konkrete Syntax
von
MiniJavaScript

Die konkrete Syntax von MiniJavaScript wird durch die Regeln in Abbildung 1.15 bestimmt. Die für die Interpretation durch den Browser notwendigen `script`-Tags bleiben unberücksichtigt.

Die Form, wie wir die syntaktischen Regeln aufgeschrieben haben, bedarf einiger Erklärung.

Eingerahmte Zeichen sind genau die, die tatsächlich im Programm vorkommen können. Gleichgültig, ob sie über eine normale oder über eine „Spezialtast-



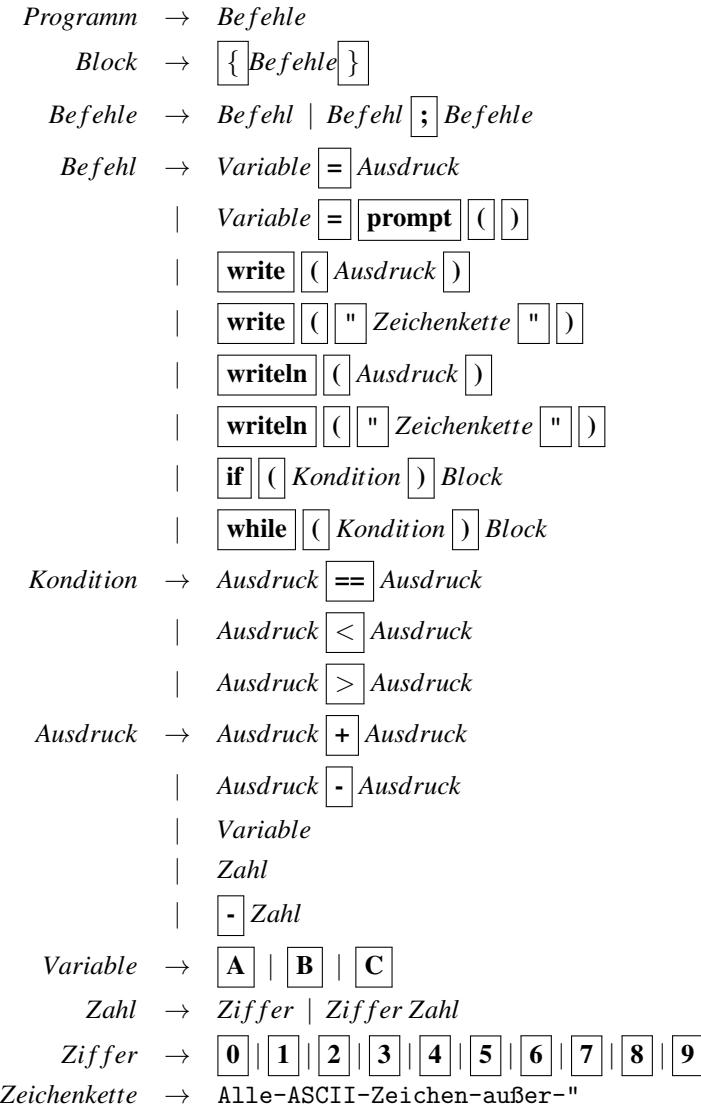


Abbildung 1.15: Regeln für syntaktisch korrekte MiniJavaScript-Programme

tatur“ (mit **write**, **prompt** usw.) eingegeben werden. Wir nennen sie *Terminale*. Dabei handelt es sich um nichts anderes als um Zeichen des zugrunde liegenden Alphabets.

Wörter in *Normalschrift* sind grammatischen Begriffe, die uns in die Lage ver-

| | |
|------------------|---|
| Nichtterminale | setzen, die Regeln überhaupt formulieren zu können. Prinzipiell sind diese Begriffe frei wählbar, anstelle von „Ausdruck“ könnten wir an jeder vorkommenden Stelle ebenso „XY11K19“ schreiben, was aus nahe liegenden Gründen nicht geschieht. Grammatikalische Begriffe heißen <i>Nichtterminale</i> . |
| Regeln | <i>Der Rechtspfeil</i> → trennt das auf der linken Seite einer <i>Regel</i> oder <i>Produktion</i> stehende Nichtterminal ⁸ von ihrer rechten. Man liest den Pfeil als „definiert“, „ist definiert als“ oder „steht für“, z.B. „Ausdruck definiert eine Zahl“. In englischsprachiger Literatur nimmt man einfach nur „is“. |
| Alternative | <i>Der Alternativ-Strich</i> verringert den Schreibaufwand für den Fall, dass es für ein und dasselbe Nichtterminal mehrere Regeln gibt. $X \rightarrow Y$ und $X \rightarrow Z$ dürfen also zu $X \rightarrow Y Z$ zusammengefasst werden. liest man als „oder“. |
| Regeln | <i>Alle-ASCII-Zeichen-außer-</i> steht für die Menge aller zulässigen ASCII-Zeichen, also wenigstens alle Ziffern, Buchstaben und Sonderzeichen, die über die Tastatur eingetippt werden können. Dies ist wesentlich kürzer, als wenn wir die gesamte Liste als alternative Regeln hier angeben müssten. Nur das Zeichen " ist nicht erlaubt. |
| Metasprache | Die in dieser Form vorgenommene Definition der konkreten Syntax einer Sprache ist selbst eine Sprache, genauer: eine <i>Metasprache</i> . Die oben erklärten Zeichen und → sind Bestandteile dieser Sprache, die von BACKUS ⁹ eingeführt wurde |
| Backus-Naur-Form | und <i>Backus-Naur¹⁰-Form</i> , kurz: BNF, heißt. „Form“ ist das englische Wort für Formular oder Formalismus. |
| | Die BNF wurde für Sprachen eingeführt, deren grammatischen Regeln auf sämtlichen linken Seiten aus jeweils genau einem Nichtterminal bestehen. Obwohl die BNF prinzipiell nicht darauf beschränkt ist, werden wir uns verstärkt dem Studium dieser Grammatikklasse zuwenden. |
| EBNF | Um rekursive Regeln der Form $X \rightarrow a aX$ verkürzt darstellen zu können, wurde die BNF durch die Metazeichen * und + ergänzt. Man spricht dann von der <i>Erweiterten Backus-Naur-Form</i> , kurz: EBNF. |
| Syntax-diagramme | Für $X \rightarrow a aX$ schreibt man dann $X \rightarrow a^+$. Um die Folgen von Terminalen und/oder Nichtterminalen, auf die sich * bzw. + beziehen, auszuweisen, sind ggf. geschweifte Klammern zu setzen. Dies würde bei unserer Sprache MiniJavaScript zu einer Kollision mit den zum Sprachumfang gehörenden geschweiften Klammern führen, die als Terminalen der Sprache nicht als metasprachliche Symbole verwendet werden dürfen. Manchmal werden <i>anstelle von</i> * geschweifte und <i>anstelle von</i> + eckige Klammern verwendet. Weitere Varianten sind anzutreffen. Die EBNF ist nicht standardisiert. |
| | Eine alternative Darstellungsform zur BNF bilden <i>Syntaxdiagramme</i> . Sie sind gra- |

⁸Bei Regeln dieser Art bestehen *sämtliche* linke Regelseiten aus *genau* einem Nichtterminal.

⁹John Warner Backus (geb. 3. Dezember 1924 in Philadelphia; gest. 17. März 2007 in Ashland, Oregon)

¹⁰Peter Naur (geb. 25. Oktober 1928 in Frederiksberg bei Kopenhagen)

fikorientiert und damit intuitiv sehr gut lesbar: Syntaxdiagramme bestehen aus Rechtecken, Ovalen und Pfeilen. Zwischen den Elementen der BNF und denen der Syntaxdiagramme gibt es folgende Entsprechungen:

| BNF | Syntaxdiagramm |
|---------------|---|
| Terminal | Oval |
| Nichtterminal | Rechteck |
| Regel | Verbindungspfeile zwischen Ovalen oder Rechtecken |

Jedes Syntaxdiagramm besitzt genau einen Eingangs- und genau einen Ausgangspfeil. Wie bei der BNF-Notation wird mit jedem Diagramm ein bestimmtes Nichtterminal erklärt. Für MiniJavaScript können die in Abbildung 1.16 dargestellten Diagramme angegeben werden. Sie entsprechen den Regeln in Abbildung 1.15. Es handelt sich allerdings um eine unvollständige Angabe: Für jedes Nichtterminal muss genau ein Syntaxdiagramm (durchaus mit alternativen Pfaden) angegeben werden.

Folgt man dem jeweiligen Pfeil, entstehen syntaktisch korrekte Teilwörter der definierten Art. Im Beispiel von `Befehle` bedeutet das entweder einen einzigen Befehl oder eine Folge durch Semikoli getrennter Befehle von der Art `Befehl`. Die Einzelheiten werden in Kapitel 2 behandelt.

Übung 1.5

Interpretieren Sie das MiniJavaScript-Programm in einem Browser mit JavaScript-Unterstützung und spielen Sie das Nim-Spiel gegen den Computer. Editieren Sie anschließend den Programmtext und verursachen Sie durch Einfügen von `()()` an irgendeiner Stelle einen Syntaxfehler. Sie werden dann feststellen, dass Ihnen der Browser (genauer: der JavaScript-Interpreter) keinerlei Fehlerhinweis gibt und lediglich eine leere Seite anzeigt. Justieren Sie ggf. Ihre persönliche Einstellung zu Fehlermeldungen in der Programmierung.



1.10 Syntaxanalyse

Ist das MiniJavaScript-Programm für das Nim-Spiel in Abschnitt 1.9 korrekt? Warum? Wie können wir das nachweisen?

Genau dies leistet die *Syntaxanalyse*. Bei einem Sprachübersetzer (Compiler) bildet die Syntaxanalyse – das *Parsing* – die erste Hauptphase. Im Erfolgsfall hinterlässt sie einen *Abstrakten Syntaxbaum (AST)*, s. Abschnitt 1.11 für die Weiterverarbeitung (Übersetzung in die Zielsprache). Wir verwenden FLACI für die Syntaxanalyse des folgenden, einfachen MiniJavaSkript-Beispielprogramms.

Syntaxanalyse
Parsing
AST

```
if (1>B) {while (7<C) {B=C}}; write(-5)
```

ist korrekt, was zu einer baumartigen Analysedarstellung führt. Ein *Ableitungsbaum* oder *Parsebaum* entsteht, wenn das Wort (Programm) syntaktisch korrekt ist.

Ableitungsbaum
Parsebaum

Computerübung 1.7

Verwenden Sie das FLACI-Modul „Kontextfreie Grammatiken“ Wählen Sie die MiniJava-



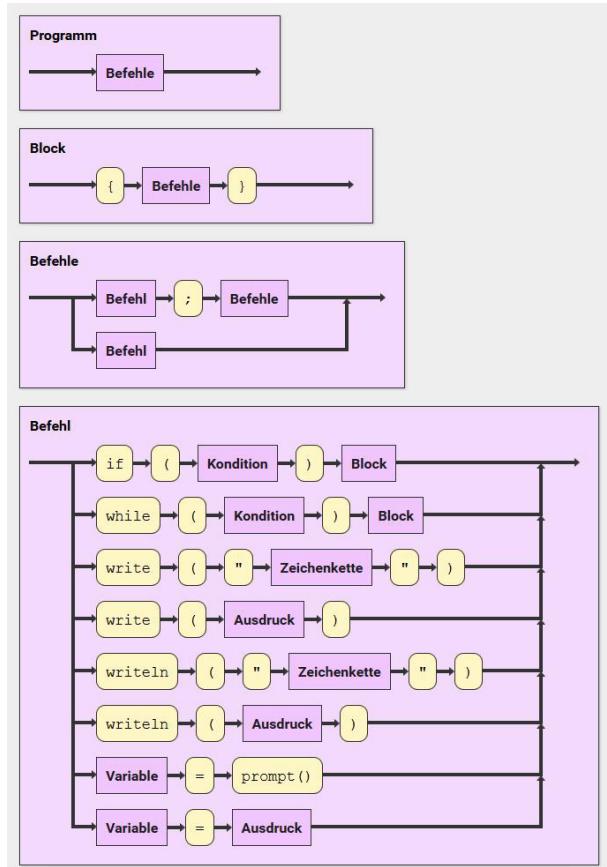


Abbildung 1.16: Syntaxdiagramme für MiniJavaScript (unvollständig)

Script-Definition aus der Beispielsammlung (Grammatiken). Überzeugen Sie sich auch von der syntaktischen Korrektheit des obigen Ausdrucks und des Nim-Spiel-Programms.

Die Analyse beginnt beim „obersten“ Nichtterminal. In unserem Beispiel ist das **Programm**. Dann folgen wir der Regel für **Programm** und allen weiteren für die auftretenden Nichtterminale. Die Terminalen müssen an den angegebenen Positionen der Regeln im Programmtext stehen.

Anhand des recht übersichtlichen Regelwerks in Beispiel 1.7 wollen wir das Parse-Verfahren noch einmal illustrieren.

Didaktischer Hinweis 1.6

Halten Sie sich strikt an die Vorstellung, dass es sich in Beispiel 1.7 um bedeutungslose Wörter (Zeichenketten) handelt, deren Aufbau durch Regeln festgelegt wird. Arithmetische Berechnungen sind damit *nicht* möglich.



**Beispiel 1.7**

Die folgenden Regeln beschreiben sehr einfache arithmetische Ausdrücke. Die Regeln für **Zahl** und **Ziffer** werden hier nicht wiederholt. Sie können Abbildung 1.15 entnommen werden.

$$\begin{array}{l}
 \text{Ausdruck} \rightarrow \text{Ausdruck } \boxed{+} \text{ Ausdruck} \\
 | \quad \text{Ausdruck} \boxed{-} \text{ Ausdruck} \\
 | \quad \text{Zahl}
 \end{array}$$

Wir wählen **3** **-** **4** **+** **5** als Eingabewort.

Die Vorgehensweise ist intuitiv: Beginnend mit dem Nichtterminal *Ausdruck* wenden wir eine Regel an, die den Aufbau dieses Nichtterminals erklärt. Das ist beispielsweise die zweite Regel: $\text{Ausdruck} \rightarrow \text{Ausdruck } \boxed{-} \text{ Ausdruck}$. Das *Start-Nichtterminal*, also *Ausdruck*, wird nun durch die komplette rechte Regelseite, d.h. $\text{Ausdruck } \boxed{-} \text{ Ausdruck}$, ersetzt, s. Tabelle unten.

Für den nächsten Schritt ergeben sich zwei zu substituierende Nichtterminale, nämlich *Ausdruck* und *Ausdruck*. Es ist nicht festgelegt, mit welchen passenden(!) Regeln die beiden *Ersetzungen* dieses Nichtterminals an zwei verschiedenen Positionen vorzunehmen sind. Es muss nicht in beiden Fällen die gleiche sein.

Die Substitution der noch vorhandenen Nichtterminale durch je eine zugehörige rechte Seite einer passenden Regel wird solange fortgesetzt, bis nur noch Terminalsymbole vorkommen.

Die folgenden beiden Darstellungen visualisieren dieses Vorgehen: in Tabellenform und in linearisierter Form. Die dabei verwendeten neuen Symbole, wie \Rightarrow , bzw. Begriffe, wie *Satzform* und *Ableitung*, werden in Kapitel 2 eingeführt.

Ersetzung

| Satzform | angewandte Regel |
|--|---|
| <i>Ausdruck</i> | $\text{Ausdruck} \rightarrow \text{Ausdruck } - \text{ Ausdruck}$ |
| <i>Ausdruck</i> - <i>Ausdruck</i> | $\text{Ausdruck} \rightarrow \text{Zahl}$ |
| Zahl - <i>Ausdruck</i> | $\text{Zahl} \rightarrow 3$ |
| 3 - <i>Ausdruck</i> | $\text{Ausdruck} \rightarrow \text{Ausdruck } + \text{ Ausdruck}$ |
| 3 - <i>Ausdruck</i> + <i>Ausdruck</i> | $\text{Ausdruck} \rightarrow \text{Zahl}$ |
| 3 - Zahl + <i>Ausdruck</i> | $\text{Zahl} \rightarrow 4$ |
| 3 - 4 + <i>Ausdruck</i> | $\text{Ausdruck} \rightarrow \text{Zahl}$ |
| 3 - 4 + Zahl | $\text{Zahl} \rightarrow 5$ |
| 3 - 4 + 5 | akzeptiert |

Ableitung für **3-4+5**: $\text{Ausdruck} \Rightarrow \text{Ausdruck } - \text{ Ausdruck} \Rightarrow \text{Zahl } - \text{ Ausdruck} \Rightarrow \text{3 } - \text{ Ausdruck} \Rightarrow \text{3 } - \text{ Ausdruck } + \text{ Ausdruck} \Rightarrow \text{3 } - \text{ Zahl } + \text{ Ausdruck} \Rightarrow \text{3 } - \text{ 4 } + \text{ Ausdruck} \Rightarrow \text{3 } - \text{ 4 } + \text{ Zahl} \Rightarrow \text{3 } - \text{ 4 } + \text{ 5}$

Für syntaktisch fehlerhafte Ausdrücke gibt es keine derartige Ersetzungskette (Ableitung). Protokolliert man diesen Prozess für unser Beispiel grafisch, so entsteht ein *Ableitungsbaum (Parsebaum)*, wie in Abbildung 1.17.

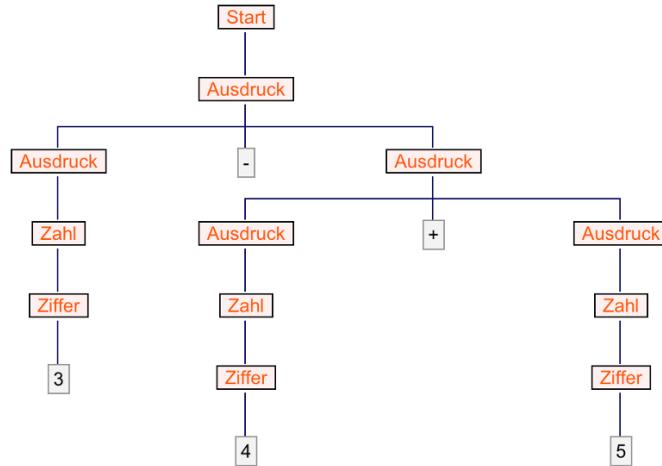


Abbildung 1.17: Ableitungsbaum für den Ausdruck $3 - 4 + 5$

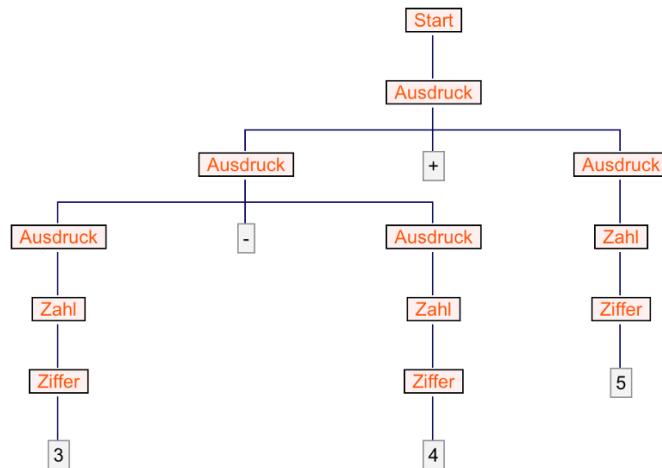


Abbildung 1.18: Noch ein Ableitungsbaum für den Ausdruck $3 - 4 + 5$

Computerübung 1.8

Legen Sie die Definition der Sprache arithmetischer Ausdrücke gemäß Beispiel 1.7 mit FLACI an. Wechseln Sie zwischen Syntax-Diagramm und Textmodus, um die Regeln anzuzeigen.

Neben dem in Abbildung 1.17 dargestellten Parsebaum erhalten Sie (auch mit FLACI, durch Auswahl) für das betrachtete Eingabewort einen zweiten, s. Abbildung 1.18. In FLACI ist die Benennung mit „Ableitungsbau 1 von 2“ und „Ableitungsbau 2 von 2“ willkürlich. Machen Sie sich vollständig klar, an welcher Stelle verschiedene Regeln eingesetzt wurden, um das jeweilige Ergebnis zu erzielen.

Gibt es für ein Wort einer Sprache mehrere strukturell verschiedene Ableitungsbäume, heißt die Sprache *mehrdeutig*. Dies werden wir in Abschnitt 1.11 näher untersuchen.

1.11 Abstrakte Syntax

Die Übersetzung eines Wortes (Satzes) aus der Quellsprache in eine bestimmte Zielsprache setzt dessen syntaktische Korrektheit voraus. Diese wird durch ein erfolgreiches Parsing, wie in Abschnitt 1.10 angedeutet, festgestellt.

Für die Berechnung des mathematischen Ausdrucks $23 + 8$ in Beispiel 1.7 ist es nicht mehr relevant, dass das Wort 23 aus den beiden Zeichen 2 und 3 gewonnen wurde. Wichtig ist die nun vorliegende Zahl 23. Auch dies ist eine der Möglichkeiten den Parsebaum zu einem *abstrakten Syntaxbaum* zu verdichten.

Von Bedeutung für weitere Verarbeitungsschritte ist also die *abstrakte Syntax*, d.h. lediglich der Extrakt, die (unbedingt notwendigen) Bestandteile des Satzes unter Beachtung der Satzstruktur.

Die Repräsentation der abstrakten Syntax eines syntaktisch korrekten Eingabewortes erfolgt ebenfalls in Form eines Baumes. Man nennt ihn einen *abstrakten Syntaxbaum*, kurz: *AST*, von engl. abstract syntax tree. Ein AST repräsentiert die Satzstruktur in kondensierter Weise.

Typischerweise werden *Zahl*, *Variablename* usw. als Terminate angesehen, zu deren Speicherung im AST die betreffende Zahl bzw. der vorgefundene Bezeichner als Zusatzinformation gehören. Für *Zahl* wird dies mit Bezug auf Beispiel 1.7 in Abbildung 1.19 durch Rechtecke gekennzeichnet.

Andererseits können bestimmte Elemente der *konkreten Syntax*, wie Klammern, das Semikolon als Anweisungstrennzeichen und das Zuweisungszeichen, weggelassen werden, da sie für die abstrakte Syntax keine Rolle spielen.

Man nennt einen solchen Baum für arithmetische Ausdrücke auch *Operatorbaum*. Operatorbäume werden in der funktionalen Programmierung für entsprechende Ausdrücke und bei Datenbanken zur Darstellung von Anfragen verwendet.

Wie wir aus Abschnitt 1.10 schon wissen, ergeben sich für das (syntaktisch korrekte) Wort $\boxed{3} \boxed{-} \boxed{4} \boxed{+} \boxed{5}$ aus Beispiel 1.7 zwei verschiedene Parsebäume und folglich auch zwei verschiedene zugehörige AST. Sie sind in Abbildung 1.19 zu

sehen.

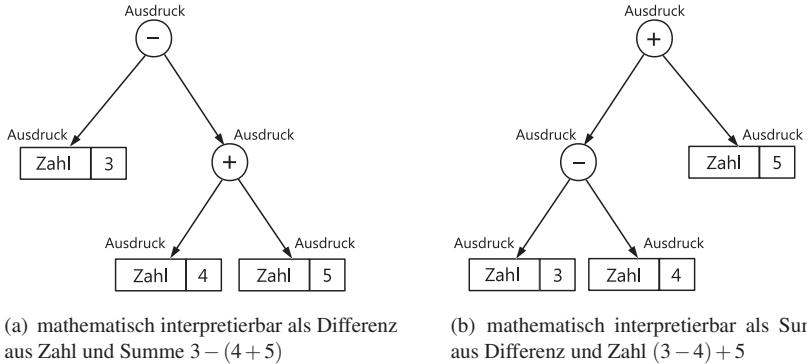


Abbildung 1.19: Zwei verschiedene Bäume (AST) für $3-4+5$

Schon für das kleine Beispiel 1.7 sehen wir, dass ein Ableitungsbaum (s. Abbildung 1.18) und ein abstrakter Syntaxbaum (s. Abbildung 1.19) für ein und denselben korrekten Ausdruck nicht identisch sind.

| konkrete Syntax | abstrakte Syntax |
|---|--|
| $\text{Ausdruck} \rightarrow \text{Ausdruck} + \text{Ausdruck}$ | <pre> graph TD A((+)) --> B(()) A --> C((...)) </pre> |
| $\text{Ausdruck} \rightarrow \text{Ausdruck} - \text{Ausdruck}$ | <pre> graph TD A((-)) --> B((...)) A --> C((...)) </pre> |
| $\text{Ausdruck} \rightarrow \text{Zahl}$ | <pre> graph TD A[Zahl] --> B((...)) </pre> |

Abbildung 1.20: Gewinnung eines AST (rechts) aus dem Parsebaum (links)

Obwohl wir auf eine strenge Definition der beiden Baum-Typen (Ableitungsbaum und AST) verzichten wollen, können wir angeben, wie man für einen syntaktisch korrekten Satz einen zugehörigen AST gewinnt. Wie Abbildung 1.20 zeigt, wird AST aus Parsebaum

dazu jeder Regel der konkreten Syntax genau ein Teilbaum zugeordnet: Für jedes Nichtterminal X in $X \rightarrow \alpha$ wird dabei genau ein Knotentyp definiert. Dessen Kindknoten entsprechen den in α vorkommenden Nichtterminalen.

Zwei oder mehrere verschiedene AST für ein und denselben Ausdruck (Satz) sind für einen Compiler im Allgemeinen unerwünscht. Für das betreffende Wort der *Quellsprache* entstünden (auf völlig korrektem Weg) *verschiedene Wörter* der *Zielsprache*. Um eine solche *syntaktische Mehrdeutigkeit* zu vermeiden, muss man versuchen, die syntaktischen Regeln für die Quellsprache zu verändern, s. Abschnitt 2.4. Dies ist nicht in jedem Fall erfolgreich: Für *inhärent mehrdeutige Sprachen* gibt es nur Grammatiken, die mehrdeutig sind.

In Abschnitt 9.1 werden wir Sprachübersetzungen vornehmen *ohne* den jeweiligen AST explizit zu definieren und in einer Datenstruktur zu speichern.

Syntaxgesteuerte Übersetzung (syntax-directed translation – SDT) ermöglicht es auf elegante Weise im Zuge der syntaktischen Analyse der einzelnen Kategorien die zugehörigen Codefragmente der Zielsprache zu erzeugen und zum Zusammenbau weiterzureichen. Hierfür werden die Knoten des Ableitungsbäumes *attribuiert* und *semantische Regeln* zur Berechnung dieser Attribute hinterlegt.

Quell- und Zielsprache

syntaktische Mehrdeutigkeit



2 Formale Grammatiken

2.1 Muster und formale Grammatiken

In Abschnitt 1.7 haben wir erfahren, dass eine Sprache L eine endliche oder im Allgemeinen eine abzählbar unendliche Teilmenge der Wortmenge A^* über einem bestimmten Alphabet A ist. Die Elemente jeder Sprache sind Wörter aus A^* .

Zur Definition einer bestimmten Sprache findet also ein *Auswahlprozess* statt.

Die einfachste Form dieses Prozesses könnte darin bestehen, einzelne Wörter aus A^* herauszugreifen. Auf diese Weise erzeugt man endliche Sprachen. Zur Bestimmung unendlicher Sprachen, wie wir sie beispielsweise mit der Menge aller JAVA-Programme vor uns haben, braucht man so etwas wie einen „Auswahlprozessor“. Dieser wird auf jedes Wort aus A^* angewandt und nimmt genau die Wörter in L auf, die die entsprechende Auswahlbedingung (i.S.v. „Aufnahmebedingung“) erfüllen. Abbildung 2.1 illustriert das Prinzip. Programmiersprachen, die mit streams¹ arbeiten können, sind in der Lage, Sprachen nach dieser Vorstellung (als streams) zu implementieren.

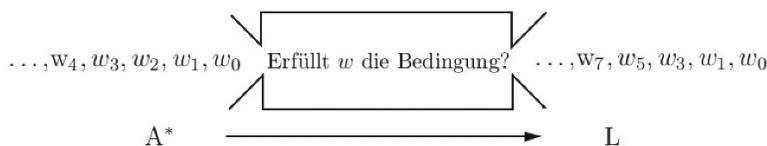


Abbildung 2.1: Auswahlprozess für $L \subseteq A^*$

Die *Auswahlbedingung* charakterisiert die Sprache L über A vollständig. Wie könnte eine solche Bedingung formuliert werden? Die schon in Abschnitt 1.7 genannte Idee eines Filters stellt sich als eine Art Schablone oder Muster dar, wie wir dies von Texteditoren kennen, s. Abbildung 2.2.

Unter Verwendung von Schablonen dieser Art gestaltet sich der Auswahlprozess als *Pattern matching* (Mustervergleich): Das zu beurteilende Wort w aus A^* wird mit dem Muster verglichen und wenn es passt in L aufgenommen, ansonsten ver-

Auswahlprozess

Schablone
Muster

pattern matching

¹Der Datentyp stream (Strom) ermöglicht die Arbeit mit potentiell unendlichen Mengen, wie etwa Zahlenfolgen. Eine wichtige Voraussetzung dafür ist, dass die Programmiersprache über das Konzept der verzögerten Evaluation verfügt. Dies gilt für eine Reihe funktionsorientierter Sprachen.

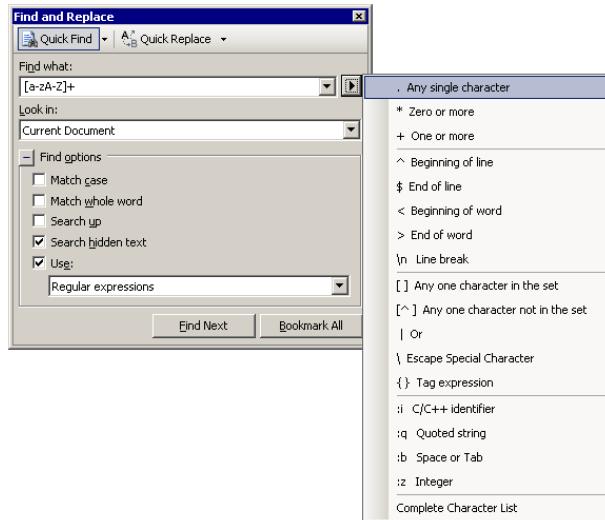


Abbildung 2.2: Muster (reguläre Ausdrücke) in Texteditoren

worfen. Das Verfahren ist sehr bequem. Leider ist es nicht mächtig genug, um beliebige Sprachen bzw. Sprachklassen zu beschreiben. In Kapitel 4 untersuchen wir die Leistungsfähigkeit solcher Muster.

Regeln Wir suchen nach einem leistungsfähigeren Auswahlprozessor und stoßen auf die Beschreibung einer Sprache durch *Regeln*, wie in Abschnitt 1.9. Regeln sind (der wichtigste) Bestandteil einer *formalen Grammatik*.



Definition 2.1

Eine (*formale*) Grammatik G ist ein 4-Tupel $G = (N, T, P, s)$ mit folgenden Eigenschaften:

$N \dots$ Menge der *Nichtterminale* (grammatikalische Variablen)

$T \dots$ Menge der *Terminale* (Alphabetzeichen)

N, T sind nichtleere, endliche und disjunkte Mengen, d.h. $N \cap T = \emptyset$.

$P \dots$ endliche Menge von *Regeln* oder *Produktionen*

$P = \{\alpha \rightarrow \beta \mid \alpha \in (N \cup T)^* \setminus T^* \text{ und } \beta \in (N \cup T)^*\}$

$s \dots$ Start- oder Satz- oder Spitzensymbol, wobei $s \in N$.

Vokabular Im Grammatik-Kontext tritt die Terminalmenge T an die Stelle des Alphabets ($T \neq \emptyset$). Die Menge $N \cup T$ wird häufig als *Vokabular* V bezeichnet. $(N \cup T)^*$ bezeichnet die Menge aller *Zeichenketten über dem Vokabular*. Dies sind „gemischte Zeichenketten“, die aus beliebig vielen Nichtterminalen und Terminalen in beliebiger Reihenfolge bestehen. Man nennt sie *Satzformen*².

Satzform

²Manchmal wird *zusätzlich* gefordert, dass eine Satzform aus dem Spitzensymbol gemäß Definition 2.3 ableitbar ist. Unsere Definition sieht das nicht vor.

In Definition 2.1 wird eine sehr freizügige Regelgestalt $\alpha \rightarrow \beta$ erlaubt. Auf der rechten Seite von Produktionen (β) dürfen beliebige Satzformen stehen. Links (α) sind lediglich Folgen von Terminalen ausgeschlossen. Beispiel 2.1 zeigt eine Grammatik mit uneingeschränkter Regelgestalt. Die Notation erfolgt in BNF. Sie wurde in Abschnitt 1.9 eingeführt.

 $\alpha \rightarrow \beta$

Beispiel 2.1

$G = (N, T, P, s)$, mit

$$\begin{aligned} N &= \{S, A, B\} \\ T &= \{a, b, c\} \\ P &= \{S \rightarrow AS \mid ccSb, cS \rightarrow a, AS \rightarrow Sbb, cSb \rightarrow c\} \\ s &= S \end{aligned}$$

ist eine formale Grammatik i.S.v. Definition 2.1.



Wie in den Regeln der Sprache MiniJavaScript aus Abbildung 1.15 besitzen die Regeln der Grammatik in Beispiel 2.2 eine besondere Eigenschaft: Sämtliche linke Regelseiten bestehen aus genau je einem Nichtterminal. Grammatiken dieser Bauart nennt man *kontextfreie Grammatiken*, kurz: kfG. Eine fundierte Einordnung dieser Grammatikklasse erfolgt in Abschnitt 2.5. Da dieser Begriff jedoch im Folgenden sehr oft benötigt wird, führen wir ihn an dieser Stelle bereits ein. Auch die Regeln in Abbildung 1.15 haben die für eine kfG geforderte Form.

kontextfreie
Grammatik
kfG

Beispiel 2.2

Wir betrachten die folgende sehr einfache Grammatik: $G = (N, T, P, s)$, mit

$$\begin{aligned} N &= \{Zahl, Ziffer\} \\ T &= \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\} \\ P &= \{Zahl \rightarrow Ziffer \mid Ziffer Zahl, \\ &\quad Ziffer \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9\} \\ s &= Zahl \end{aligned}$$



FLACI kann nur mit kfG in BNF (nicht EBNF) umgehen. Die Regeln werden textuell oder grafisch als Syntaxdiagramme (ggf. mit alternativen Vorwärtspfeilen) notiert, s. Abbildung 2.3.

Die einzelnen Regel werden – beginnend mit der Regel für das Spitzensymbol³ – über die Tastatur eingegeben. Der Pfeil \rightarrow wird aus dem Zeichen – gefolgt von $>$ gebildet.

kfG in FLACI

Terminale werden bei der Eingabe der Regeln implizit definiert und erscheinen unverändert in schwarzer Farbe. Während der Eingabe kann es zu Umentscheidungen kommen: Steht ein (zunächst als Terminal identifiziertes) Zeichen auf der linken Seite einer Regel, so handelt es sich um ein Nichtterminal. Überall wo es in den Regeln vorkommt, wird es als solches rot hervorgehoben.

Computerübung 2.1

Wählen Sie die kfG-Komponente von FLACI aus und definieren Sie die in Beispiel 2.2 verwendete Grammatik. Orientieren Sie sich an Abbildung 2.3.



³Auf die Rolle des Spitzens- oder Startsymbols kommen wir in Abschnitt 2.2 zu sprechen.



Abbildung 2.3: Grammatik für Zahlwörter mit verschiedenen Regel-Darstellungsformen

Schalten Sie anschließend auf die Syntaxdiagramm-Darstellung um und machen Sie sich mit dem Editieren dieser Diagramme vertraut.

2.2 Ableitung und definierte Sprache

Die intuitive Anwendung grammatischer Regeln bei der Syntaxanalyse eines Wortes wurde in Abschnitt 1.10 (s. Tabelle auf Seite 25) vorgeführt. Von der dort angegebenen Ableitung für 3-4+5 zitieren wir an dieser Stelle den vorletzten „Ableitungsschritt“: $\underbrace{3 - Zahl}_{u} + \underbrace{Ausdruck}_{v} \Rightarrow 3 - 4 + \underbrace{Ausdruck}_{v}$.

Für $\underbrace{3 - Zahl}_{x \ y \ z} + \underbrace{Ausdruck}_{u}$ ist dieser direkte Übergang zu $\underbrace{3 - 4}_{x \ y' \ z} + \underbrace{Ausdruck}_{v}$,

d. h. $xyz \Rightarrow xy'z$, möglich, da die Regel $\underbrace{Zahl}_{y} \rightarrow \underbrace{4}_{y'}$ in P existiert, sodass

$Zahl$ durch 4 ersetzt werden kann. Die Teilwörter x und z werden unverändert übernommen. Definition 2.2 formalisiert $u \Rightarrow v$ aus dieser beispielbezogenen Beschreibung durch die Relationen \Rightarrow_G und $\stackrel{*}{\Rightarrow}_G$.

Definition 2.2

Mit $u \in (N \cup T)^* \setminus T^*$ und $v \in (N \cup T)^*$ stehen u und v in Relation \Rightarrow_G , d.h. $u \Rightarrow_G v$, und man sagt: „ u geht unter G unmittelbar über in v “, falls u und v die folgenden Formen besitzen: $u = xyz$, $v = xy'z$, $x, z \in (N \cup T)^*$ und $(y \rightarrow y') \in P$.

Für $d \Rightarrow_G e \Rightarrow_G \dots \Rightarrow_G f$ schreibt man verkürzend $d \stackrel{*}{\Rightarrow}_G f$.

Es ist üblich, das an den Doppelpfeil angehängte G wegzulassen und auch die Sprechweise etwas zu entschärfen: „ u geht unmittelbar über in v “.

α steht in Relation \Rightarrow mit β , wenn es in α (mindestens) eine Teilzeichenkette gibt,

die mit der linken Seite (mindestens) einer Regel aus P übereinstimmt.

Unter Verwendung von \Rightarrow bzw. $\stackrel{*}{\Rightarrow}$ kann nun der Auswahlprozess für jedes Wort w aus A^* gemäß G als *Ableitung*⁴ exakt beschrieben werden.

Ableitung

Definition 2.3

Eine Folge von Satzformen $(\alpha_0, \alpha_1, \dots, \alpha_n)$, mit $\alpha_0 = s$, $\alpha_n = w$, $w \in T^*$ und $s \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow \dots \Rightarrow w$, heißt *Ableitung von w*.



Wörter, die zu der von G definierten Sprache $L(G)$ gehören, müssen vom Startsymbol aus ableitbar sein.

Definition 2.4

Die durch G definierte^a Sprache $L(G)$ ist $L(G) = \{w \mid w \in T^* \text{ und } s \stackrel{*}{\Rightarrow}_G w\}$.



^aOft wird dies auch „erzeugte Sprache“ oder „erzeugbare Sprache“ genannt.

Computerübung 2.2

Verwenden Sie die Grammatik für die Sprache der Zahlwörter aus Beispiel 2.2 und leiten Sie das Wort 371 ab.



Das Ergebnis ist in Abbildung 2.4 zu sehen. Die im rechten Bereich protokolierte Ableitung (*Zahl*, *Ziffer Zahl*, *3 Zahl*, *3 Ziffer Zahl*, *3 7 Zahl*, *3 7 Ziffer*, *3 7 1*) schreibt man in der Form:

$Zahl \Rightarrow Ziffer Zahl \Rightarrow 3 Zahl \Rightarrow 3 Ziffer Zahl \Rightarrow 3 7 Zahl \Rightarrow 3 7 Ziffer \Rightarrow 3 7 1$.

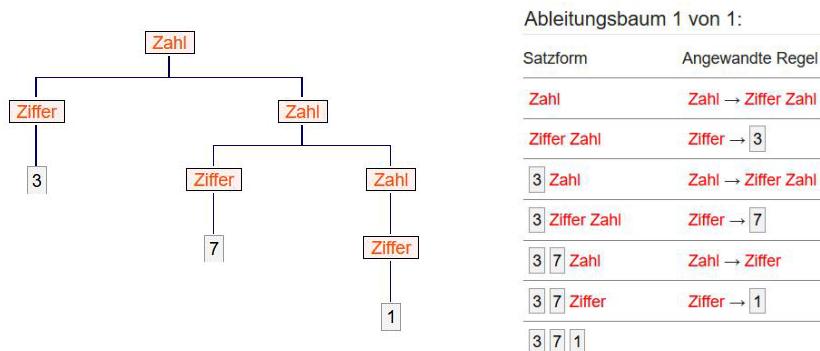


Abbildung 2.4: Ableitung des Zahlwortes 371

Computerübung 2.3

Erzeugen Sie die beiden Ableitungsbäume aus Abschnitt 1.10 für das Wort 3-4+5 der Sprache einfacher arithmetischer Ausdrücke unter Verwendung der mehrdeutigen Grammatik aus Beispiel 1.7. FLACI unterstützt Sie bei der Auswahl aus mehreren Ableitungsbäumen.



⁴Manchmal wird anstelle von Ableitung der Begriff „Herleitung“ verwendet und man spricht von einer „generativen Grammatik“.

2.3 Nichtdeterminismus des Ableitungsprozesses

Wie bereits erwähnt, beschränken wir uns zunächst auf kfG.

Im Protokoll der Ableitung für 371 rechts in Abbildung 2.4 ist gut zu erkennen, dass in jedem Schritt das am weitesten links stehende Nichtterminal ersetzt wurde.

Linksableitung

In diesem Fall spricht man von *Linksableitung*. Schaltet man in FLACI (durch Klick auf das „L“) auf Rechtsableitung um, wirkt sich das zwar auf das Protokoll jedoch nicht auf den Ableitungsbaum aus. In Abbildung 2.5 werden die beiden Protokolle zum direkten Vergleich nebeneinander gestellt.

Ableitungsbaum 1 von 1:

| Satzform | Angewandte Regel |
|---------------|--------------------|
| Zahl | Zahl → Ziffer Zahl |
| Ziffer Zahl | Ziffer → 3 |
| 3 Zahl | Zahl → Ziffer Zahl |
| 3 Ziffer Zahl | Ziffer → 7 |
| 3 7 Zahl | Zahl → Ziffer |
| 3 7 Ziffer | Ziffer → 1 |
| 3 7 1 | |

Ableitungsbaum 1 von 1:

| Satzform | Angewandte Regel |
|----------------------|--------------------|
| Zahl | Zahl → Ziffer Zahl |
| Ziffer Zahl | Zahl → Ziffer Zahl |
| Ziffer Ziffer Zahl | Zahl → Ziffer |
| Ziffer Ziffer Ziffer | Ziffer → 1 |
| Ziffer Ziffer 1 | Ziffer → 7 |
| Ziffer 7 1 | Ziffer → 3 |
| 3 7 1 | |

Abbildung 2.5: Links- vs. Rechtsableitung des Zahlwortes 371

Das analysierte Wort entsteht vom Spitzensymbol aus von links bzw. von rechts her. Durch die Wahl der Ersetzungsreihenfolge kann offenbar an der Menge der akzeptierten Wörter, d.h. der Sprache, nichts verändert werden. Das kann man sich rasch an der Baumstruktur (Ableitungsbaum) klar machen: Die Anwendungsmöglichkeit einer Regel nach Übereinstimmung eines zu ersetzenden Nichtterminals mit der linken Seite dieser Regel ist von der Bearbeitungsreihenfolge unabhängig. Ganz anders sieht es bei nicht-kfG aus, wie in Beispiel 2.1, deren Regelmenge wir hier wiederholen: $P = \{S \rightarrow AS \mid ccSb, cS \rightarrow a, AS \rightarrow Sbb, cSb \rightarrow c\}$. Dort kann es durchaus zu neuen Konstellationen für Satzformen kommen, auf die dann linke Regelseiten der Grammatik passen, die bei einem in der Reihenfolge veränderten Ableitungsprozess nicht anwendbar wären.

Linksableitung

Wir stellen fest: Die beiden Ableitungsäume (für Links- bzw. Rechtsableitung) entstehen zwar in unterschiedlicher Abfolge; strukturell sind sie jedoch identisch. Für kfG ist es deshalb sinnvoll, generell nur *Linksableitungen* zu verabreden.

Computerübung 2.4

Schalten Sie auf *Rechtsableitung* um und leiten Sie 371 erneut ab. Setzen Sie auch den Einzelschrittmodus ein, um die Aufbaudynamik des betrachteten Parsebaumes zu verfolgen.



Definition 2.5

Eine Ableitung für ein Wort w gemäß einer kfG heißt *Linksableitung* genau dann, wenn es für jedes $i = 0, 1, \dots, n - 1$ eine Produktion $Y_i \rightarrow Q_i$ derart gibt, dass $\gamma_i = \alpha_i Y_i \beta_i$, $\gamma_{i+1} = \alpha_i Q_i \beta_i$, wobei $X = \gamma_0 \Rightarrow \gamma_1 \Rightarrow \dots \Rightarrow \gamma_n = w$, $s = X$, $n \geq 1$, $\beta_i \in (N \cup T)^*$ gelten und $\alpha_i \in T^*$.

**Beispiel 2.3**

Gegeben sei die Grammatik $G = (N, T, P, s)$, mit $s = S$, $T = \{a, b, c, -\}$, $N = \{A, S\}$ und $P = \{S \xrightarrow{1} A, S \xrightarrow{2} S - A, A \xrightarrow{3} a, A \xrightarrow{4} b, A \xrightarrow{5} c\}$.



Mit Hilfe der Ziffern über den Pfeilen der fünf Produktionen werden wir uns im Folgenden auf die jeweilige Regel beziehen. Wegen $S \Rightarrow S - A \Rightarrow S - A - A \Rightarrow A - A - A \Rightarrow a - A - A \Rightarrow a - b - A \Rightarrow a - b - b$ gilt $a - b - b \in L(G)$. Die gewählte Ableitung ist $(S, S - A, S - A - A, A - A - A, a - A - A, a - b - A, a - b - b)$, die zugehörige Folge der Regelanwendungen lautet $(2, 2, 1, 3, 4, 4)$. Ebenso hätte man $S \Rightarrow S - A \Rightarrow S - b \Rightarrow S - A - b \Rightarrow A - A - b \Rightarrow A - b - A \Rightarrow a - b - b$ nehmen können, was der Folge $(2, 4, 2, 1, 4, 3)$ entspricht. Rechtsableitung hätte noch $(2, 4, 2, 4, 1, 3)$ hinzugefügt. Diese drei Ableitungen unterscheiden sich lediglich durch die Reihenfolge ihrer Elemente. Insfern ist es sinnvoll, für sie alle einen Repräsentanten, nämlich $(2, 2, 1, 3, 4, 4)$, auszuwählen. (Natürlich beschreibt nicht jede Permutation dieser Elemente eine mögliche Ableitung.)

Satz 2.1

Bei kontextfreien Sprachen, kurz: kfS, gibt es zu jedem Ableitungsbaum genau eine Linksableitung.

**Beweis**

Als Beweis gilt die Argumentation, die sich aus den vorangegangenen Bemerkungen ergibt. Ein strenger Beweis benötigt eine formale Definition des Begriffes Ableitungsbaum, worauf wir hier jedoch verzichten möchten. \square

Für ein bestimmtes Wort einer durch eine kfG definierten Sprache kann das Auffinden einer zugehörigen Linksableitung sehr aufwendig sein. Gibt es nämlich mehrere Regeln, deren linke Regelseiten aus dem zu ersetzenen Nichtterminal bestehen, muss man sich im nächsten Schritt für eine entscheiden. Alle passenden Regeln sind grundsätzlich gleichberechtigt anwendbar.

Abbildung 2.4 zeigt links den Linksableitungsbaum für das Wort 371 in Beispiel 2.2. Was wäre passiert, wenn wir im dritten Ableitungsschritt die Regel $Zahl \rightarrow Ziffer$ anstelle von $Zahl \rightarrow Ziffer Zahl$ angewandt hätten? Für $Zahl$ gibt es diese beiden Regeln.

In diesem und vergleichbaren Fällen gerät der Ableitungsprozess in eine *Sackgasse*. Im konkreten Beispiel wäre es unmöglich, aus dem *Ziffer*-Knoten die Blätter mit den Werten 7 und 1 zu erzeugen. Das Erreichen einer Sackgasse ist nicht gleichbedeutend damit, dass es für das analysierte Wort keine Linksableitung gibt. Durch Umentscheidung in der Regelauswahl am jeweils letzten Entscheidungspunkt kann man doch noch das Ziel erreichen. Dies setzt voraus, dass entsprechende Alternativen existieren.

Sackgasse

Backtracking Das beschriebene Verfahren heißt *Backtracking*. Erst wenn alle Möglichkeiten ausgeschöpft sind und sich darunter keine Ableitung für das zu analysierende Wort w ergab, ist nachgewiesen, dass w nicht zur Sprache gehört.

Wir stellen deshalb fest, dass der Ableitungsprozess – trotz verabredeter Linksableitung – bei *spontaner* Auswahl aus einer Menge prinzipiell anwendbarer Regeln in Sackgassen geraten kann. Diese Irrwege werden durch Backtracking korrigiert, was natürlich sehr aufwendig ist und wenn möglich durch bessere Verfahren (ggf. für eingeschränkte Grammatiken) vermieden werden sollte.

EARLEY Bei der Regelauswahl ist außerdem wichtig, solche Anwendungen zu vermeiden, die zu einem Endloszyklus führen. Dann würde der Ableitungsprozess nicht terminieren. Der in FLACI eingebaute EARLEY-Algorithmus, leistet die stets terminierende Analyse des Eingabewortes durch ein Memorieren bereits einmal durchgeführter Ableitungsschritte. Dies geschieht ohne die definierende kfG vorbereitend verändern zu müssen. Aus Satz 2.1 wissen wir, dass es für jedes Wort einer kfS, d. h. für jeden Ableitungsbaum, genau eine Linksableitung gibt. Gilt das auch umgekehrt? Gibt es zu jeder Linksableitung genau einen Ableitungsbaum? Je Nichtterminal kann es ja durchaus mehrere Regeln geben, die im nächsten Ersetzungsschritt völlig gleichberechtigt angewandt werden dürfen. Das muss nicht zwangsläufig in Sackgassen führen. Für ein betrachtetes Wort kann es durchaus mehrere strukturell verschiedene Ableitungsbäume geben. Solche kfG nennen wir *mehrdeutig*. Sie sind Gegenstand von Abschnitt 2.4.

mehrdeutig

Zusammenfassend stellen wir für kfG fest: Ein bestimmter Ableitungsbaum beschreibt genau ein Wort der Sprache. Umgekehrt kann es für ein betrachtetes Wort mehr als eine zugehörige Linksableitung (mit strukturgeleichen Ableitungsbäumen) und sogar mehrere *strukturell verschiedene* Ableitungsbäume geben.

2.4 Mehrdeutigkeit kontextfreier Grammatiken

mehrdeutige Grammatik

Wie wir in Beispiel 1.7 bereits gesehen haben, besitzt das Wort 3-4+5 wenigstens zwei voneinander verschiedene Ableitungsbäume, s. Abbildungen 1.17 und 1.18. Falls es wenigstens ein Wort gibt, für das mehrere verschiedene Ableitungsbäume existieren, heißen solche Grammatiken *mehrdeutig*. Zwei Ableitungsbäume sind verschieden voneinander, wenn sie sich *strukturell* unterscheiden. Die Entstehungsabfolge jeglicher Teilbäume spielt dabei keine Rolle.



Definition 2.6

Eine kfG ist *mehrdeutig*, wenn es (mindestens) ein Wort in $L(G)$ gibt, das (mindestens) zwei Linksableitungen besitzt. Andernfalls ist die Grammatik eindeutig.

Eine kfS L ist *eindeutig*, wenn es (mindestens) eine eindeutige kfG G , mit $L = L(G)$, gibt. Ansonsten ist L (inhärent) *mehrdeutig*, d. h., wenn es für die betrachtete kfS ausschließlich mehrdeutige kfG gibt.

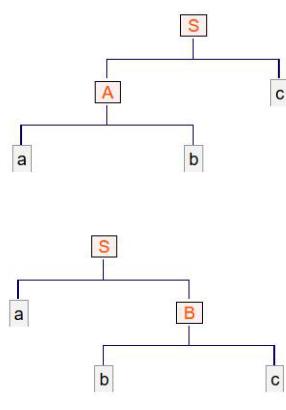
Beispiel 1.7 sollte uns daran erinnern, dass mehrere Ableitungsbäume, wie in Abbildung 1.19, zu verschiedenen Interpretationen und in der weiteren Verarbeitung zu unterschiedlichen Übersetzungen führen können. Mehrdeutige Grammatiken sollten daher vermieden werden.

Beispiel 2.4

Gegeben sei die Grammatik $G = (N, T, P, s)$, mit $s = S$, $T = \{a, b, c\}$, $N = \{A, B, S\}$ und $P = \{S \xrightarrow{1} aB, S \xrightarrow{2} Ac, A \xrightarrow{3} ab, B \xrightarrow{4} bc\}$.



Für das Wort abc gibt es zwei Linksableitungen: $S \xrightarrow{1} aB \xrightarrow{4} abc$ und $S \xrightarrow{2} Ac \xrightarrow{3} abc$.



Ableitungsbau 1 von 2:

| Satzform | Angewandte Regel |
|----------|---------------------------|
| S | $S \rightarrow A \quad c$ |
| A | $A \rightarrow a \quad b$ |
| a b c | |

Ableitungsbau 2 von 2:

| Satzform | Angewandte Regel |
|----------|---------------------------|
| S | $S \rightarrow a \quad B$ |
| a B | $B \rightarrow b \quad c$ |
| a b c | |

Abbildung 2.6: Zwei verschiedene Linksableitungen für das Wort abc

Beispiel 2.5 betrachtet einen sehr „unangenehmen“ Vertreter mehrdeutiger kfG.

Beispiel 2.5

Gegeben sei die Grammatik $G = (N, T, P, s)$, mit $s = S$, $T = \{a\}$, $N = \{S\}$ und $P = \{S \rightarrow S S \mid a \mid \epsilon\}$. $S \rightarrow \epsilon$ bedeutet S aus der Satzform ersatzlos zu entfernen. Obwohl diese Grammatik nur drei Regeln besitzt, gibt es für das Wort a unendlich viele Ableitungsbäume.



Ableitungsbau 1 von ?:

| Satzform | Angewandte Regel |
|----------|-------------------|
| S | $S \rightarrow a$ |
| a | |

Abbildung 2.7: Unendlich viele Linksableitungen für das Wort a

Computerübung 2.5

Verwenden Sie FLACI zur Ableitung des Wortes a für die in Beispiel 2.5 genannte mehrdeutige kfG. Sie erhalten den folgenden Hinweis.



Hinweis

Für das Wort **a** gibt es mehrere oder sogar unendlich viele Ableitungsbäume. Es wird nur ein möglicher Baum generiert.

Notieren Sie drei weitere Ableitungen und geben Sie die zugehörigen Bäume an.

Computerübung 2.6



$G = (N, T, P, s)$ mit $s = \text{Zahl}$, $T = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$, $N = \{\text{Zahl}, \text{Ziffer}\}$ und $P = \{\text{Zahl} \rightarrow \text{Ziffer} \mid \text{Zahl Zahl}$, $\text{Ziffer} \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9\}$ ist eine formale Grammatik für die Sprache der Zahlwörter für natürliche Zahlen. Für das Eingabewort 12345 gibt es 14 Ableitungsbäume. Prüfen Sie dies mit FLACI nach.

Verwenden Sie für **Zahl** alternativ die Regeln $\text{Zahl} \rightarrow \text{Ziffer} \mid \text{Ziffer Zahl}$, so ergibt sich eine eindeutige Grammatik.

Leider gibt es *keinen* Universalalgorithmus, der für eine beliebige kontextfreie Sprache entscheidet, ob sie mehrdeutig ist. Im Einzelfall gelingt es jedoch, solche Mehrdeutigkeiten durch Angabe einer *äquivalenten Grammatik* zu eliminieren.



Definition 2.7

Zwei Grammatiken G_1 und G_2 heißen *äquivalent*, geschrieben: $G_1 \cong G_2$, wenn die zugehörigen erzeugbaren Sprachen übereinstimmen, d.h. wenn $L(G_1) = L(G_2)$.

inhärent
mehrdeutig

Allerdings kann nicht zu jeder mehrdeutigen kfG eine äquivalente eindeutige angegeben werden. Gelingt dies nicht, spricht man von *inhärenter Mehrdeutigkeit*.

Beispielsweise ist $L = \{a^i b^j c^k \mid i = j \text{ oder } j = k\}$ eine inhärent mehrdeutige kfS.

Computerübung 2.7

Untersuchen Sie die folgende Grammatik auf Mehrdeutigkeit.

$G = (N, T, P, s)$, mit $s = X$, $T = \{a, b\}$, $N = \{X\}$ und $P = \{X \rightarrow a \mid b \mid XX\}$.

dangling else

Computerübung 2.8

Ein typisches Beispiel einer mehrdeutigen kfG ist das sog. „dangling else“-Problem bei Pascal-ähnlichen Sprachen, s. auch Abschnitt 7.6.

$G = (N, T, P, s)$, mit $s = \text{statement}$, $T = \{\text{if}, \text{then}, \text{else}, \text{A1}, \text{A2}, \text{S1}, \text{S2}\}$,

$N = \{\text{statement}, \text{expression}\}$ und $P = \{\text{statement} \rightarrow \text{if expression then statement} \mid \text{if expression then statement else statement} \mid \text{S1} \mid \text{S2}; \text{expression} \rightarrow \text{A1} \mid \text{A2}\}$

Experimentieren Sie mit verschiedenen Eingabewörtern. Die Frage ist: Zu welchem der beiden **if** gehört das **else** in dem Ausdruck **if A1 then if A2 then S1 else S2?** Finden Sie eine Möglichkeit, um durch möglichst geringfügige Modifikationen von Regeln in P das „lose, baumelnde“ **else** zu beheben.

Computerübung 2.9

Gegeben sei folgende Grammatik: $G = (\{E, T, F\}, \{(.,), a, +, *\}, P, E)$,

$P = \{E \rightarrow T \mid E + T, T \rightarrow F \mid T * F, F \rightarrow a \mid (E)\}$. Stellen Sie fest, ob $a+a*(a+a)$ und $a(a+a)$ zu der durch G definierten Sprache $L(G)$ gehören. Geben Sie – falls möglich – zugehörige Ableitungsbäume an. Stellen Sie fest, ob G eine mehrdeutige Grammatik ist.

Wenn man bei E , T bzw. F an **expression**, **term** bzw. **factor** denkt, wird klar, dass diese Grammatik die Sprache der korrekt geklammerten arithmetischen Ausdrücke beschreibt.

Erzeugen Sie weitere Wörter aus $L(G)$ z.B. mit dem FLACI-Zauberstab links neben der Worteingabe.

2.5 CHOMSKY-Hierarchie

Bisher haben wir den Begriff „kontextfreie Grammatik“ lediglich mit der Vorstellung verknüpft, dass sämtliche Produktionsregeln auf der linken Seite aus genau einem Nichtterminal bestehen. Dies wird nun durch eine Definition präzisiert. Außerdem wird die Klasse der kfG in eine Grammatik-Hierarchie eingeordnet.

CHOMSKY⁵ hat 1956 die formalen Grammatiken in 4 Klassen (Typen) eingeteilt und damit hierarchische Sprachklassen beschrieben. Auf den ersten Blick sieht das recht willkürlich aus. Mit zunehmender Beschäftigung mit dem Gegenstand wird deutlich, wie fundamental diese Typisierung für die Theorie der formalen Sprachen und die Automatentheorie ist.

Das klassifizierende Merkmal dieser Typen ist die *Regelgestalt*. Wir gehen von der allgemeinsten⁶ Form, wie sie bereits in Definition 2.1 angegeben wurde, aus, und vermerken in der Tabelle die zusätzlichen Forderungen (Einschränkungen):

$$\alpha \rightarrow \beta \mid \alpha \in (N \cup T)^* \setminus T^* \text{ und } \beta \in (N \cup T)^*.$$

| Typ | Bezeichnung | Regelgestalt |
|-----|-----------------|---|
| 0 | unbeschränkt | keine Einschränkung |
| 1 | kontextsensitiv | wie Typ 0 und zusätzlich: $ \alpha \leq \beta $ (längenmonoton) Ausnahme: $s \rightarrow \epsilon$ zulässig, wenn s in keiner Regel auf der rechten Seite steht. |
| 2 | kontextfrei | wie Typ 1 und zusätzlich: $\alpha \in N$ Ausnahme: Regeln der Form $\alpha \rightarrow \epsilon$ zulässig |
| 3 | regulär | wie Typ 2 und zusätzlich: $ \beta \leq 2$, genauer: <i>Entweder</i> $\alpha \rightarrow x$ und $\alpha \rightarrow Ax$ (linkslinear) <i>oder</i> $\alpha \rightarrow x$ und $\alpha \rightarrow xA$ (rechtslinear) mit $x \in T$ und $A \in N$. |

Die zugehörigen *Sprachklassen* heißen – wie die *Grammatiktypen* – *unbeschränkt* (uG; uS), *kontextsensitiv* (ksG; ksS), *kontextfrei* (kfG; kfS) und *regulär* (rG; rS).

Die von CHOMSKY definierten Sprachklassen bilden eine Hierarchie. Für die in Abbildung 2.8 dargestellten Sprachklassen gilt:

$$\mathcal{L}_{\text{Typ 3}} \subset \mathcal{L}_{\text{Typ 2}} \subset \mathcal{L}_{\text{Typ 1}} \subset \mathcal{L}_{\text{Typ 0}}.$$

Man beachte, dass die Elemente dieser Mengen Sprachen, also selbst Mengen, sind. Deshalb schreiben wir anstelle von L ein stilisiertes \mathcal{L} .

Übung 2.1

Ergänzen Sie die Klasse der endlichen Sprachen in Abbildung 2.8.

CHOMSKY-Hierarchie

Sprachklassen



⁵Noam Chomsky, geb. 1928

⁶Die in der Literatur hierfür verwendeten Definitionen differieren ein wenig. In allen Fällen werden aber die in der Tabelle angegebenen vier Sprachklassen beschrieben.

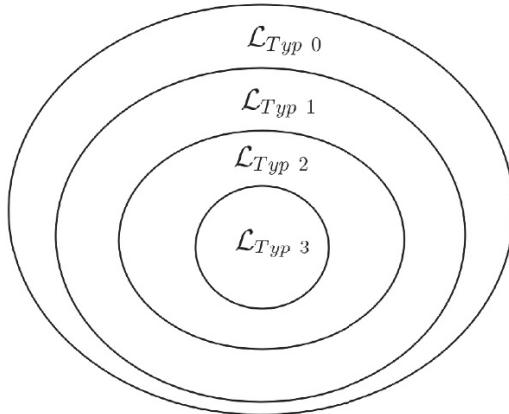


Abbildung 2.8: CHOMSKY-Hierarchie für Sprachklassen

Da wir zentrale Inhalte der Theorie der formalen Sprachen auf Aspekte der Sprachübersetzer anwenden werden, befassen wir uns vor allem mit Chomsky-Typ-2- und Chomsky-Typ-3-Sprachen. Dass wir uns auf kfS und aufgrund der Chomsky-Hierarchie auf rS konzentrieren, haben wir von Anfang an betont. Selbst wenn es sich in bestimmten Fällen um eine ksS (Typ 1) handelt, versucht man in der Praxis durch die Festlegung von (separat zu bearbeitender) Kontextbedingungen eine kfS herzustellen.

unbeschränkt uS nennt man auch *Phrasenstrukturgrammatik* oder *Semi-Thue*⁷-System.

kontextsensitiv Die Bezeichnungen *kontextfrei* vs. *kontextsensitiv* stützt die folgende Vorstellung: Soll ein Nichtterminal A einer ksG nur im Kontext u (links von A) und v (rechts von A) durch β ersetzt werden dürfen, reicht die Regel $A \rightarrow \beta$ (*kontextfreie Ersetzung*) nicht aus. $uAv \rightarrow x\beta y$ ermöglicht das Gewünschte. u, v, x, y sind beliebige Folgen von Vokabularzeichen. Es ist also notwendig, auf der linken Regelseite Satzformen zuzulassen, wie dies in der Definition (Tabelle) festgelegt wurde.

Beispiel 2.6

Ein (in einschlägiger Literatur reichlich strapaziertes) Beispiel einer ksG, die nicht kontextfrei ist, ist $G = (\{B, C, S\}, \{a, b, c\}, P, S)$ mit $P = \{S \rightarrow aSBC \mid aBC, CB \rightarrow BC, aB \rightarrow ab, bB \rightarrow bb, bC \rightarrow bc, cC \rightarrow cc\}$.

Die zugehörige Sprache ist $L(G) = \{a^n b^n c^n \mid n \in \mathbb{N}, n \geq 1\}$.

Beispiel 2.7

$G = (\{A, B\}, \{a, b, c\}, P, A)$ mit $P = \{A \rightarrow aABC \mid aBc, cB \rightarrow Bc, aB \rightarrow ab, bB \rightarrow bb\}$. Die zugehörige Sprache ist ebenfalls $L(G) = \{a^n b^n c^n \mid n \in \mathbb{N}, n \geq 1\}$.

Beispiel 2.8

$G = (\{S, A, C\}, \{a, b, c\}, P, S)$ mit $P = \{S \rightarrow aAbc \mid abc, A \rightarrow aAbC \mid abC, Cb \rightarrow bC, Cc \rightarrow cc\}$. Die zugehörige Sprache ist ebenfalls $L(G) = \{a^n b^n c^n \mid n \in \mathbb{N}, n \geq 1\}$.

⁷Norwegischer Mathematiker A. THUE, 1863-1922

Damit haben wir schon drei (echt) verschiedene Grammatiken angegeben, die ein und dieselbe Sprache definieren. Diese Grammatiken sind also äquivalent.

2.6 ε -Sonderregelungen

Etwas problematisch ist die Verwendung sog. ε -Regeln der Form: $\alpha \rightarrow \varepsilon$ in kontextsensitiven (hier nur für $\alpha = s$), kontextfreien und regulären Grammatiken. Sie verstößen gegen die *Längenmonotonie*, die für ksG und damit auch kfG sowie rG gefordert ist, s. Abschnitt 2.5. Die Längenmonotonie fordert für jede Regel in P , dass die rechte Seite (β) mindestens genauso lang ist, wie die linke (α), also $|\alpha| \leq |\beta|$. Wegen $|\alpha| > 0$ aber $|\beta| = |\varepsilon| = 0$ trifft dies bei $\alpha \rightarrow \varepsilon$ nicht zu. Dadurch wäre es auch nicht möglich, das leere Wort abzuleiten, sodass $\varepsilon \in L(G)$, denn dafür ist $s \rightarrow \varepsilon$ für Sprachen vom Typ 1, 2 und 3 definitiv erforderlich. Der folgende Satz rechtfertigt die o.g. Ausnahme in der Regelgestalt.

Satz 2.2

Zu jeder ε -freie ksG $G = (N, T, P, s)$ gibt es eine äquivalente ksG $G' = (N', T', P', s')$ mit $L(G') = L(G) \cup \{\varepsilon\}$.

Längen-
monotonie



Beweis

Die erforderliche Grammatiktransformation für kfG und analog für ksG geschieht folgendermaßen⁸:

1. Wähle ein noch nicht in N vorhandenes Nichtterminal s' als Spitzensymbol von G' .
2. Ergänze die Regeln $s' \rightarrow s$ und $s' \rightarrow \varepsilon$.

Aus G entsteht die Grammatik

$$G' = (N \cup \{s'\}, T, P \cup \{s' \rightarrow s \mid \varepsilon\}, s').$$

Will man $\varepsilon \in L(G)$ auch für reguläre Grammatiken erzielen, müssen wir die stark eingeschränkte Regelgestalt sorgfältig beachten. Eine Regel der Form $s' \rightarrow s$, wie oben verwendet, ist für Typ-3-Grammatiken nicht erlaubt. Folgende Transformation ist zielführend:

$\varepsilon \in L(G_{Typ1,2})$

1. Wähle ein noch nicht in N vorhandenes Nichtterminal s' als Spitzensymbol von G' .
2. Ergänze für alle Regeln $s \rightarrow \beta$ in P die Regeln $s' \rightarrow \beta$ sowie $s' \rightarrow \varepsilon$.

$\varepsilon \in L(G_{Typ1,2,3})$

Wir heben hervor, dass durch diese Transformation der jeweilige Typ der Grammatik unverändert bleibt. □

Beispiel 2.9

Durch simples Ergänzen der Regel $s \rightarrow \varepsilon$ gelingt die gewünschte Transformation *im Allgemeinen* nicht, wie dieses Beispiel zeigt. G_2 wurde aus G_1 durch pures Hinzufügen der Regel $K \rightarrow \varepsilon$ gewonnen. $G_1 = (\{K, H\}, \{i, j, l, r\}, \{K \rightarrow iHj, H \rightarrow IK \mid r\}, K)$.



⁸Ein alternativer Vorschlag lautet:

1. Benenne das Spitzensymbol s in allen Regeln in P um zu s' , wobei $s' \notin N$.
2. Ergänze die beiden Regeln $s \rightarrow s' \mid \varepsilon$.

In diesem Falle wird das Spitzensymbol von G in G' beibehalten, d.h. $s' = s$.

So geht es im
Allgemeinen
nicht!

$G_2 = (\{K, H\}, \{i, j, l, r\}, \{K \rightarrow iHj \mid \varepsilon, H \rightarrow lK \mid r\}, K)$. Offensichtlich gehört das Wort ilj , wegen $K \Rightarrow iHj \Rightarrow ilKj \Rightarrow ilj$, zu $L(G_2)$ jedoch nicht zu $L(G_1)$. Dies entspricht nicht unserer Absicht. G_2 erfüllt die geforderte Eigenschaft nicht!

Die Lösung unseres obigen Beispiels ist daher

$G' = (\{K, H, Q\}, \{i, j, l, r\}, \{Q \rightarrow K \mid \varepsilon, K \rightarrow iHj, H \rightarrow lK \mid r\}, Q)$. Das Wort ilj gehört weder zu $L(G_1)$ noch zu $L(G')$.

Computerübung 2.10



Verwenden Sie FLACI (kfG) zur Bearbeitung von Beispiel 2.9.

Nun wenden wir uns dem „umgekehrten“ Problem zu: Wie kann man für eine kfG ε -Freiheit herstellen? KfG mit ε -freien Regeln werden in Sätzen der folgenden Kapitel öfters vorausgesetzt. Dann wird immer wieder auf den folgenden Satz 2.3 und das im Beweis angegebene Transformationsverfahren verwiesen.



Satz 2.3

Zu jeder kfG $G = (N, T, P, s)$ mit ε -Regeln der Form $A \rightarrow \beta$, mit $A \in N$ und $\beta \in (N \cup T)^*$, gibt es eine äquivalente kfG $G' = (N', T', P', s')$ ohne ε -Regeln (ggf. bis auf $s \rightarrow \varepsilon$).

Beweis

Die Hauptidee dieses (konstruktiven) Beweises besteht darin, ein Verfahren anzugeben, das alle möglichen ε -Ersetzungen in den betreffenden Produktionen „per Hand“ ausführt, sodass sich die ε -Regeln erübrigten.

Hierfür müssen zunächst alle Nichtterminale $A_i \in N$ bestimmt werden, die in beliebig vielen Schritten zu ε abgeleitet werden können: Beginne mit $N_\varepsilon = \{A_i\}$, mit $A_i \rightarrow \varepsilon$ in P . Ergänze im nächsten Schritt A in N_ε , wenn $A \rightarrow A_1 A_2 \dots A_k$ in P , wobei $k \geq 1$, $A_i \in N$ und für alle A_i ($1 \leq i \leq k$) gilt $A_i \in N_\varepsilon$. Das Verfahren stoppt, wenn sich im jeweils nächsten Schritt keine weitere Veränderung in N_ε ergibt. Da N endlich ist, terminiert das Verfahren.

Anschließend entfernen wir alle Regeln der Gestalt $A_i \rightarrow \varepsilon$ aus P . Für jede Regel $B \rightarrow \beta A_i \gamma$ in P , mit $A_i \xrightarrow{*} \varepsilon$, d.h. $A_i \in N_\varepsilon$, ergänzen wir $B \rightarrow \beta \gamma$. β und γ sind Satzformen, von denen höchstens eine das leere Wort bezeichnet. Die ursprünglichen Regeln $B \rightarrow \beta A_i \gamma$ in P bleiben erhalten. \square

Damit ist nun gezeigt, dass die Verwendung von ε -Regeln in kfG eher „kosmetische Wirkung“ hat, denn sie sind verzichtbar ohne die Sprache zu verändern.



Beispiel 2.10

$G_1 = (\{X, B, K\}, \{a, c\}, \{X \rightarrow aB, B \rightarrow cB \mid K, K \rightarrow a \mid \varepsilon\}, X)$. Die zu G_1 äquivalente Grammatik G'_1 ohne ε -Regeln ergibt sich nach der oben (im Beweis) beschriebenen Transformation: $G'_1 = (N', T', P', s')$, mit $N' = N = \{X, B, K\}$, $T' = T = \{a, c\}$, $P' = \{X \rightarrow aB \mid a, B \rightarrow cB \mid c \mid K, K \rightarrow a\}$, $s' = s = X$.

Computerübung 2.11



Verwenden Sie FLACI (kfG) zur Bearbeitung von Beispiel 2.10.



Übung 2.2

Die nachfolgende Grammatik für mathematische Klammerausdrücke enthält ε -Regeln. Formen Sie die Grammatik G so um, dass eine ε -freie Grammatik G' entsteht.

$$\begin{aligned}
 G &= (N, T, P, s), \text{ mit} \\
 N &= \{S, E, T, \text{Zahl}, \text{Ziffer}\}, \\
 T &= \{+, (), 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}, \\
 P &= \{S \rightarrow (E) | \text{Zahl}, \\
 &\quad E \rightarrow ST, \\
 &\quad T \rightarrow + S | \varepsilon, \\
 &\quad \text{Zahl} \rightarrow \text{Ziffer WeitereZiffern}, \\
 &\quad \text{WeitereZiffern} \rightarrow \text{Ziffer WeitereZiffern} | \varepsilon, \\
 &\quad \text{Ziffer} \rightarrow 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9\}, \\
 s &= S
 \end{aligned}$$

Computerübung 2.12

Die durch die in Übung 2.2 erstellte ε -freie Grammatik G' beschriebene Sprache $L(G')$ soll um das leere Wort ε zu $L(G'')$ erweitert werden. Das Wort $(+)$ gehört nicht zur Sprache $L(G'')$. Sollte es mit Ihrer Grammatik ableitbar sein, dann überprüfen Sie erneut, ob Sie das Verfahren zum Hinzufügen von ε richtig angewandt haben. Fragen Sie FLACI!



Übung 2.3

Transformieren Sie die im Folgenden gegebene kfG G in eine mit ε -freien Regeln.

$$G = (N, T, P, s) \text{ mit}$$

$$\begin{aligned}
 N &= \{\text{variable}, \text{buchstabe}, \text{name}, \text{zeichen}\}, \\
 T &= \{a, b, c, d, e, A, B, C, D, E, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, !, -, +\}, \\
 P &= \{ \text{variable} \rightarrow \text{buchstabe name}, \\
 &\quad \text{name} \rightarrow \text{buchstabe name} | \text{zeichen name} | \varepsilon, \\
 &\quad \text{buchstabe} \rightarrow a | b | c | d | e | A | B | C | D | E, \\
 &\quad \text{zeichen} \rightarrow 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | ! | - | + \}, \\
 s &= \text{variable}.
 \end{aligned}$$

Es gibt noch ein anderes Verfahren, um für eine kfG mit ε -Regeln die Menge N_ε zu bestimmen. Hierzu bilden wir für jedes Nichtterminal alle möglichen Folgesatzformen, die sich durch Anwendung aller passenden Regeln auf die jeweilige Satzform ergeben. Das Verfahren verhält sich also wie ein *Simultan-Schachspieler*, der gleichzeitig je eine Partie gegen verschiedene Gegner spielt.

Simultaneous derivation

Auf unseren Fall angewandt bedeutet das: Für jede Satzform werden alle möglichen⁹ Regeln angewandt. In Anlehnung an das Schachspiel kann man das beschriebene Ableitungsverfahren als *Simultanableitung* bezeichnen. In der modernen Terminologie der Informatik spricht man wohl besser von einem *Brute-force-Ansatz*. Schon nach wenigen Schritten können gigantische Satzformen-Mengen entstehen und es wird schwierig, ja sogar unmöglich, diesen Prozess für große Schrittanzahlen praktisch auszuführen.

Die Simultanableitungen beginnen beim zu untersuchenden Nichtterminal A , genauer: mit $\{A\}$. Wie weit müssen sie fortgesetzt werden, um entscheiden zu können, ob $A \xrightarrow{*} \varepsilon$ gilt oder nicht?

⁹Es ist also durchaus möglich, dass sich aus einer Satzform mehrere Folgesatzformen ableiten lassen.

Da bei Simultanableitungen in jedem Schritt *alle* in einer Satzform enthaltenen Nichtterminale durch die jeweiligen rechten Regelseiten ersetzt werden, ergibt sich der längste mögliche „Weg zu ϵ “, wenn es für $N = \{A_1, A_2, \dots, A_n\}$ die Ableitung $A = A_1 \Rightarrow A_2 \Rightarrow \dots \Rightarrow A_n \Rightarrow \epsilon$ gibt. Es sind also höchstens $n = |N|$ Simultanableitungsschritte nötig. Damit ist gesichert, dass das Verfahren stets terminiert.

Wegen $N = \{X, B, K\}$ reichen in Beispiel 2.10 $|N| = 3$ Simultanableitungsschritte aus, um die zu ϵ ableitbaren Nichtterminale zu bestimmen.

2.7 Das Wortproblem

In Abschnitt 2.3 haben wir kfG betrachtet und die Frage aufgeworfen, ob das dort beschriebene Entscheidungsverfahren (Linksableitung ggf. mit Sackgassen) allgemeingültig ist, also ob es auch – sagen wir vorsichtig – für ksG gilt. Einen Ableitungsbaum können wir für ksG im Allgemeinen nicht erwarten, da die linken Regelseiten aus mehr als einem Symbol bestehen können. Gibt es überhaupt eine Möglichkeit, die Frage $w \stackrel{?}{\in} L(G)$ für ksS und uS zu entscheiden?

Entscheidungsproblem

Das Wesen eines (solchen) *Entscheidungsproblems* besteht darin, dass der zugehörige Bearbeitungsprozess nach endlicher Zeit stoppt und im Ergebnis genau einen Wahrheitswert (*true* oder *false*) liefert. Abbildung 2.9 zeigt den Sachverhalt. Im vorliegenden Fall erhält das gesuchte Entscheidungsverfahren ein Wort

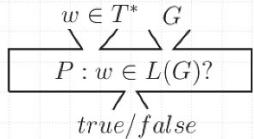


Abbildung 2.9: Wortproblem

Wortproblem

w aus T^* und eine Grammatik G der betrachteten Klasse als Eingaben. Genau dies sind die beiden Eingaben, die wir FLACI (kfG) geben, um zu entscheiden, ob $371 \in L(G)$ oder $371 \notin L(G)$, s. Abbildung 2.4 auf Seite 35. FLACI (kfG) entscheidet also die Frage: $w \stackrel{?}{\in} L(G)$. Wir nennen dies das *Wortproblem* für kfG.

Definition 2.8

Das (allgemeine) Wortproblem besteht aus der Frage nach der Existenz eines allgemeingültigen Entscheidungsverfahrens, das für jedes beliebige Wort w und jede beliebige Grammatik G in endlicher Zeit feststellt, ob entweder $w \in L(G)$ oder $w \notin L(G)$.



Um in endlich vielen Schritten festzustellen, ob ein Wort $w \in T^*$ zur Sprache $L(G)$ gehört oder nicht, ist zu prüfen, ob w aus dem Spitzensymbol s der betrachteten Grammatik G über beliebig viele Schritte ableitbar ist.

Mit der auf S. 45ff beschriebenen Simultanableitung werden aus jeder Satzform einer Satzformenmenge S_i alle möglichen Folgesatzformen erzeugt. Diese bilden zusammen¹⁰ mit S_i die Menge S_{i+1} , sodass $S_i \subseteq S_{i+1}$, bis auf folgende Ausnahmen:

- Wenn die betrachtete Satzform ausschließlich aus Terminalen besteht und mit w übereinstimmt, antwortet das Entscheidungsverfahren mit *true* und wird beendet.
- Wenn die betrachtete Satzform ausschließlich aus Terminalen besteht, jedoch nicht mit w übereinstimmt, wird diese Satzform nicht in S_{i+1} übernommen. Sie wäre eine (für die Ableitung von w) aussichtslose Kandidatin.
- Wenn die betrachtete Satzform eine Länge besitzt, die größer als $n = |w|$ ist, wird diese Satzform nicht in S_{i+1} übernommen. Aufgrund der für ksG geforderten Längenmonotonie wäre sie eine (für die Ableitung von w) aussichtslose Kandidatin.

Das Verfahren beginnt mit $S_0 = \{s\}$, wobei s das Spitzensymbol der betrachteten Grammatik bezeichnet, und endet

- entweder wenn $w \in S_k$, dann ist die Ausgabe des Algorithmus' *true*, s.o.,
- oder wenn $S_{k+1} = S_k$, d.h., es sind keine weiteren Satzformen ableitbar. Dann lautet die Ausgabe des Algorithmus' *false*.

Da es in $(N \cup T)^*$ nur *endlich* viele Satzformen gibt, deren Länge höchstens gleich $n = |w|$ ist, muss dieses Verfahren auch für $w \notin L(G)$ terminieren, denn es gibt ein solches k . Die Anzahl der Satzformen¹¹ mit Längen kleiner oder gleich n , die über $(N \cup T)^*$ mit $|N \cup T| = m \geq 2$ gebildet werden können, ergibt sich aus

$$\sum_{j=0}^n m^j = \frac{m^{n+1} - 1}{m - 1}, \quad m \geq 2.$$

(Variationen mit Wiederholung: m Elemente zur j -ten Klasse. Endliche geometrische Reihe mit $q = m$.) Hierin steht n im Exponenten, was auf gigantisches Wachstum hinweist. Aus theoretischer Sicht ist jedoch alles klar: Das Verfahren ist effektiv ausführbar, d.h., es ist algorithmisch beschrieben, und die einzige Frage ist, für welche Grammatik-*Typen* es terminiert (d.h. nach endlich vielen Schritten stoppt).

Für das folgende Beispiel 2.11 sind $m = 5$, $n = 6$. Damit sind höchstens $\frac{5^7 - 1}{4} = 19531$ Satzformen bildbar.

Beispiel 2.11

Wir verwenden die Grammatik $G = (\{A, B\}, \{a, b, c\}, P, A)$ mit $P = \{A \rightarrow aABc \mid$

¹⁰Das Archivieren früherer Satzformen ist notwendig, um ggf. Probleme bei zyklischen Ableitungen der Form $\alpha \xrightarrow{*} \alpha$ auszuschließen.

¹¹Das heißt natürlich nicht zwangsläufig, dass alle diese Satzformen in den S_i -Mengen auch wirklich vorkommen müssen.



$aBc, cB \rightarrow Bc, aB \rightarrow ab, bB \rightarrow bb\}$ aus Beispiel 2.7 mit $L(G) = \{a^n b^n c^n \mid n \in \mathbb{N}, n \geq 1\}$. Das zu untersuchende Wort ist $w = aabbcc, |w| = 6$. Die Folge der Satzformenmengen ergibt sich wie folgt:

$$S_0 = \{A\}$$

$$S_1 = \{A, aAbc, aBc\}$$

$$S_2 = \{A, aAbc, aBc, aBc, aaABcBc, aaBcBc, abc\}, \text{ da } |aaABcBc| = 7 > |w| \text{ und } abc \neq w$$

$$S_3 = \{A, aAbc, aBc, aaBcBc, aabcBc, aaBBcc\},$$

$$S_4 = \{A, aAbc, aBc, aaBcBc, aabcBc, aaBBcc, aabBcc\},$$

$$S_5 = \{A, aAbc, aBc, aaBcBc, aabcBc, aaBBcc, aabBcc, aabbcc\} \text{ und Stopp, da } aabbcc \in S_5.$$

Übung 2.4



Zeigen Sie, dass das Wort acb nicht in $L(G)$, mit $G = (\{A, B\}, \{a, b, c\}, P, A)$ mit $P = \{A \rightarrow aAbc \mid aBc, cB \rightarrow Bc, aB \rightarrow ab, bB \rightarrow bb\}$, enthalten ist.

Für CHOMSKY-Typ-0-Grammatiken versagt dieses Verfahren. Aufgrund der fehlenden Längenmonotonie ist es grundsätzlich möglich, aus Satzformen mit einer Länge größer als $|w|$ das Wort w abzuleiten. Dadurch kann die Menge der bildbaren Satzformen, aus denen das vorgegebene Wort abgeleitet werden kann, nicht beschränkt werden.

Beispiel 2.12



Wir betrachten die folgende Typ-0-Grammatik (unbeschränkte Grammatik) $G = (N, T, P, s)$ mit

$$N = \{A, B, C\},$$

$$T = \{a, b\},$$

$$\begin{aligned} P = & \{ A \rightarrow aBC \mid aA, \\ & aB \rightarrow bCBa, \\ & CBaC \rightarrow a \}, \end{aligned}$$

$$s = A.$$

Das Wort aba gehört offensichtlich zur Sprache $L(G)$, wegen $A \Rightarrow aA \Rightarrow aaBC \Rightarrow abCBaC \Rightarrow aba$. Es hat die Länge 3. Die zu dessen Ableitung verwendeten Satzformen $aaBC$ (Länge $4 > 3$) und $abCBaC$ (Länge $6 > 3$) wären von dem oben beschriebenen Verfahren als aussichtslose Kandidatinnen verworfen worden. Folglich hätte das Ergebnis nicht $aba \in L(G)$ lauten können.

Zusammenfassend stellen wir fest, dass die Längenmonotonie der Grammatik das entscheidende Kriterium für das Terminieren des beschriebenen Verfahrens ist.



Satz 2.4

Das Wortproblem ist für Typ-1,2,3-Sprachen allgemein entscheidbar, jedoch nicht für Sprachen vom Typ 0.

Beweis

Argumentativ, s.o.

□

3 Endliche Automaten und reguläre Sprachen

3.1 Allgegenwärtige Zustandsmodelle

Die Online-Reservierung eines Hotelzimmers ist heute eine sehr einfache Sache: Sofern ein Zimmer im gewünschten Zeitraum verfügbar ist, kann es entweder sofort bezogen (b) oder reserviert (r) werden. Dies wird in Abbildung 3.1 als *Zustandsmodell* aus der Perspektive eines Hotelzimmers systematisch dargestellt.

Zustandsmodell

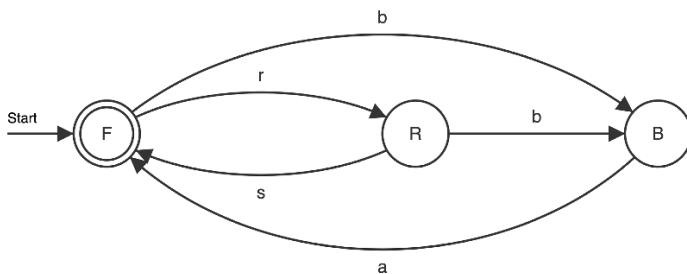


Abbildung 3.1: Unvollständiges Zustandsmodell „Hotelzimmer“

Alle *Zustände*, die das betrachtete *Zimmer* einnehmen kann, sind ...

Zustände

- F ... frei,
- R ... reserviert und
- B ... belegt.

In Abbildung 3.1 sind diese Zustände (Knoten eines Graphen) durch Kreise dargestellt. Der potenzielle Hotelgast (Kunde) hat die Aktionsmöglichkeiten b , r , s und a , d.h. das Zimmer ...

- b ... zu beziehen (zu belegen; check-in),
- r ... zu reservieren,
- s ... zu stornieren und
- a ... zu verlassen, also abzureisen (check-out).

akzeptiertes Wort

Mit diesen Kundenaktionen sind in Abbildung 3.1 die Kanten (Bögen, Pfeile, Übergänge) markiert. Wird ein zunächst freies Zimmer reserviert, danach storniert, belegt und durch Abreise des Gastes wieder frei, kann man dies mit dem Wort **rsba** beschreiben. Man sagt, dass **rsba** ein *akzeptiertes Wort* ist. (F, R, F, B, F) ist die zugehörige Zustandsfolge.

Ein Eingabewort wird genau dann akzeptiert, wenn die zugehörige Zustandsfolge mit einem *Endzustand* (hier F) abschließt. Dies ist für (F, R, F, B, F) erfüllt. Zur Markierung eines Endzustandes verwendet man zwei konzentrische Kreise.

Startzustand Der *Startzustand* (hier ebenfalls F) wird durch einen mit „Start“ markierten Pfeil ausgewiesen, s. Abbildung 3.1. Ein Zustandsmodell hat genau einen Startzustand und kann mehrere Endzustände (soweit vorhanden) besitzen. Dass es in unserem Beispiel genau einen Endzustand (F) gibt, der auch noch mit dem Startzustand übereinstimmt, ist sinnvoll, um den Prozess der Hotelzimmervergabe mit dem freien Zimmer zu beginnen und (meist nach Nutzung und Reinigung) zu beenden.

Sogar das leere Wort ϵ wird akzeptiert, da für den Übergang von F (Startzustand) zu F (Endzustand) kein einziges Zeichen benötigt wird. Die Interpretation ist klar: Das Zimmer steht bereit, aber es wird nicht nachgefragt.

Ausgehend von weiteren Beispielen, wie **bababars**, **rsrsba** und **rba**, gewinnt man die Einsicht, dass es in unserem Hotelzimmer-Zustandsmodell unendlich viele akzeptierte Wörter gibt. Deren Anfangsstück (Teilwort von links her) bis zur Einnahme des Endzustands F entspricht einem der folgenden drei Muster:

- **ba** ... Zimmernutzung ohne vorherige Reservierung,
- **rs** ... Stornierung einer früheren Reservierung ohne Zimmernutzung,
- **rba** ... eine Zimmernutzung nach vorausgegangener Reservierung.

Folgt man den Wegen im Graphen aus Abbildung 3.1 vom Startzustand aus, so kann man das unmittelbar nachvollziehen. Es ergeben sich genau diese Pfade, die von F (erstmals) zurück zu F führen.

Die dem Anfangsstück (eventuell) folgenden Teilwörter sind von gleicher „Bauart“ wie diese drei Muster, sodass sich insgesamt der Ausdruck $(ba|rs|rba)^*$ ergibt.

* bedeutet eine n -malige ($n \geq 0$) Verwendung des unmittelbar davor stehenden Zeichens bzw. der in runde Klammern gesetzten Zeichenkette.

nicht akzeptierte Wörter Es gibt aber auch Wörter, die *nicht akzeptiert* und folglich auch vom weiter oben angegebenen Ausdruck ausgefiltert werden. In unserem Anwendungskontext handelt es sich um für das Hotel unerwünschte Aktivitätenfolgen, d.h. *inakzeptable Wörter*, wie beispielsweise **rbs**: Der betrachtete Gast versucht sein bereits bezogenes Zimmer zu stornieren.

Verbotene Wörter müssen erkannt und abgewiesen werden. Dabei darf es nicht passieren, dass ...

- ... das Modell für den betrachteten Zustand (F, R, B) und ein aktionsanzeigendes Zeichen (b, r, s und a) keinen Übergang bereithält und
- ... eine fehlerhafte Eingabe in der Folge der Zustandsübergänge auf irgend-einem Weg doch noch einen Endzustand erreicht.

Wir erreichen dies durch Hinzunahme eines „Fehlerzustands“ (*trap state*) im Modell: Alle unzulässigen Übergänge führen zu diesem Zustand und mit allen verbleibenden Aktionen (a, b, r und s) kann dieser Zustand auch nicht mehr verlassen werden. Der Name *Trap* = Falle trifft es. Gleichzeitig können wir damit sicherstellen, dass bei jeder Eingabe immer klar ist, welcher Folgezustand eingenommen werden soll. Für unser Beispiel zeigt Abbildung 3.2 das vollständige Modell.

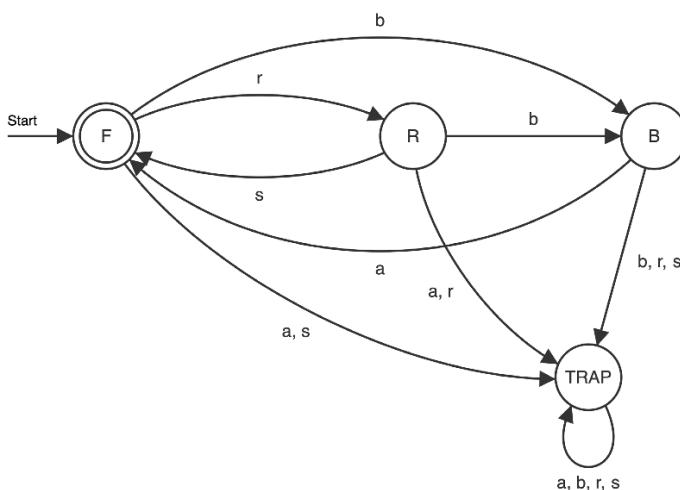


Abbildung 3.2: Vollständiges Zustandsmodell „Hotelzimmer“

Das Wort rbsabrasbbabssbababrrrr führt ebenso zum Trap-Zustand und wird folglich nicht akzeptiert, weil es keinen Endzustand erreichen kann.

Zustandsübergangsgraphen wie in Abbildung 3.2 bilden das Herzstück *abstrakter Automaten*. Sie sind mit einer *totalen* Überführungsfunktion ausgestattet. Ebenso wie formale Grammatiken dienen sie zur Definition formaler Sprachen, indem sie vorgegebene Wörter als Elemente einer bestimmten Sprache *akzeptieren* oder *ablehnen*. Daher werden sie auch *Akzeptoren* genannt.

Im konkreten Fall handelt es sich um einen *endlichen Automaten*, s. Abschnitt 3.2, der in jedem Schritt ein Zeichen des Eingabewortes liest und den zugehörigen Zustandswechsel vollzieht. Im Gegensatz zu Grammatiken, mit deren Hilfe bestimmte Wörter aus dem Spitzensymbol abgeleitet – im Sinne von *generiert* – werden, steht also bei Akzeptoren der *Analyseaspekt* im Vordergrund.

trap state

Trap

Abstrakter Automat

Akzeptor

Endlicher Automat



Didaktischer Hinweis 3.1

Aufgrund der Modellierung mit Zuständen stehen Automaten den Programmen des imperativen und objektorientierten Paradigmas sehr nahe. Obwohl Akzeptoren nichts anderes bewirken wie formale Grammatiken, fällt dem Programmierfreudigen der gedankliche Umgang mit Zustandsmaschinen oftmals wesentlich leichter. Dennoch handelt es sich in beiden Fällen um mentale Modelle, deren Zweck nicht darin besteht, in konkrete Produkte überführt zu werden.

3.2 Deterministischer Endlicher Automat (DEA, EA)

Automatentyp Wir beginnen mit dem einfachsten *Automatentyp*, den sog. endlichen Automaten (EA). EA sind Akzeptoren für Typ-3-Sprachen. Die Reichweite dieses Automatentyps ist also relativ stark beschränkt. Jeden einzelnen Automatentyp¹ ordnen wir einem entsprechenden CHOMSKY-Sprachtyp, s. Seite 41, zu.

Aufbau Endliche Automaten haben den in Abbildung 3.3 skizzierten Aufbau: Sie bestehen aus einem *Band* und einer *Kommunikationseinheit*. Das *potenziell unendliches Band* ist in aufeinanderfolgende *Felder* oder *Zellen* gegliedert.

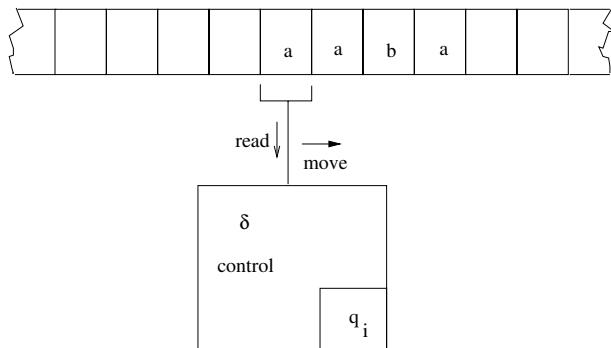


Abbildung 3.3: Aufbau eines endlichen Automaten

Kommunikationseinheit In jedes Feld des Bandes kann genau ein Zeichen des *Eingabealphabets* geschrieben werden. Die *Kommunikationseinheit* kann mit einem *Kopf* auf jeweils genau ein Feld dieses Bandes zugreifen und den entsprechenden Feldeintrag lesen und auswerten, *ohne* ihn dadurch zu entfernen oder zu überschreiben.

Kopf Dabei arbeitet die Kommunikationseinheit schritt- oder taktweise: Zu Beginn eines jeden Arbeitstaktes befindet sie sich in einem bestimmten *Zustand*. Danach

¹ Abstrakte Automaten werden als Modelle beschrieben. Dies unterstützt unsere Vorstellungskraft, darf aber deren abstrakten Charakter nicht verschleiern.

geht sie in einen Folgezustand, der sich nicht unbedingt von dem vorhergehenden unterscheidet, über. Am Ende eines Arbeitstaktes erfolgt eine *Kopfbewegung* um genau ein Feld nach rechts zu dem als nächstes zu inspizierenden Feld. Die beschriebene Arbeitsweise wird in einer *Überführungsfunktion* δ definiert.

Endliche Automaten, definieren reguläre Sprachen. Wie wir zeigen werden, stellen sie eine gleichberechtigte Alternative zu regulären Grammatiken dar.

Die folgende Definition formalisiert den beschriebenen Sachverhalt.

Definition 3.1

Ein *DEA*^a (auch EA) ist ein Quintupel $M = (Q, \Sigma, \delta, q_0, E)$, mit

- Q ... endliche Menge der Zustände,
- Σ ... Eingabealphabet, $Q \cap \Sigma = \emptyset$,
- δ ... Überführungsfunktion (*totale* Funktion), $Q \times \Sigma \rightarrow Q$,
- q_0 ... Startzustand, $q_0 \in Q$, und
- E ... endliche (nichtleere^b) Menge der Endzustände, $E \subseteq Q$.

^aDEA steht – wie in der Abschnittsüberschrift angegeben – für deterministischer endlicher Automat. In der Tat gibt es auch ein nichtdeterministisches Pendant. Die Hintergründe dieser Begriffswahl werden weiter unten erläutert.

^bFür die leere Sprache, d.h. $L = \emptyset$, wäre es notwendig, $E = \emptyset$ zuzulassen.

Überführungs-funktion
Endlicher Automat



Die in den einzelnen Zellen des Bandes eingetragenen Zeichen können *nur gelesen* werden. In der heutigen Sprechweise würde man dies wohl read-only-tape nennen. Ein DEA verfügt also über einen *Lesekopf*, visualisiert wie ein almodischer Kassetten- oder Video-Player. In jedem Takt wird

1. das aktuelle² Zeichen a gelesen,
2. ein Folgezustand gemäß δ eingenommen und
3. der Kopf um genau ein Feld nach rechts verschoben.

Die hier gewählte Reihenfolge ist einzuhalten.

Die Überführungsfunktion δ bestimmt den jeweiligen Folgezustand. Es handelt sich um eine *totale* Funktion, d.h. $\delta(q_i, a)$ muss für alle $q_i \in Q$ und $a \in \Sigma$ existieren. Für das Modell bedeutet dies, dass es niemals eine ungewisse Situation gibt, in der der Automat nicht „weiß“, wie es weiter gehen soll. Eine zugehörige Tabelle für diese zweistellige Funktion muss also stets *vollständig* ausgefüllt sein.

Für δ gibt es eine sehr anschauliche grafische Darstellung, die in Abbildung 3.4 angedeutet wird: Der Automat befindet sich zunächst in Zustand q_i , liest das Zeichen a und wechselt in den Zustand q_j . Dass der Kopf anschließend um genau ein Feld nach rechts bewegt wird, geht aus δ nicht³ hervor.

²Das „aktuelle Zeichen“ ist genau das Zeichen in dem Feld, auf/über dem sich der Lesekopf des Automaten gerade befindet.

³Dies wird mit der *erweiterten* Überführungsfunktion, die wir weiter unten betrachten, ausgedrückt.

Lesekopf

| δ | a_0 | a_1 | \dots | a_n |
|----------|----------|---------|---------|-------|
| q_0 | \dots | q_r | \dots | |
| q_1 | \dots | | | |
| \vdots | \vdots | \dots | | |
| q_m | \dots | | | |

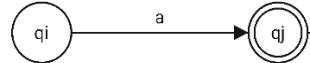


Abbildung 3.4: Überführungsfunktion eines DEA (links: Tabelle, rechts: Graph)

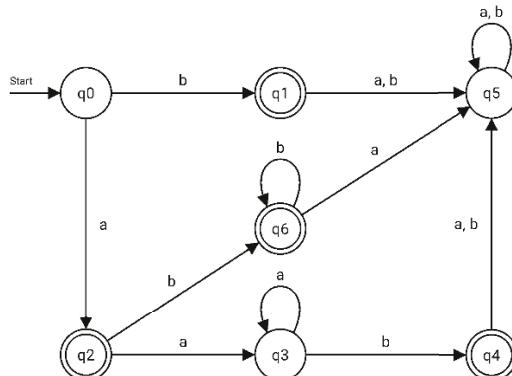
- Endzustand Falls q_j – wie in Abbildung 3.4 – ein *Endzustand* ist, wird dies durch zwei konzentrische Kreise visualisiert.
- Startzustand Jeder Automat besitzt genau einen *Startzustand*, in dem sich der Automat zu Beginn seiner Arbeit befindet. Es wird erwartet, dass das Eingabewort auf dem Band und der Lesekopf auf dem ersten Zeichen dieses Wortes stehen, s. Abbildung 3.3.
- Akzeptanzverhalten Das *Akzeptanzverhalten* des Automaten lässt sich so beschreiben: Ein DEA *stoppt*, wenn das Eingabewort vollständig⁴ abgetastet wurde und verharrt in dem dann aktuellen Zustand q_k . Falls q_k ein Endzustand ist, wird das betrachtete Eingabewort von diesem DEA akzeptiert, ansonsten wird es abgewiesen. Da δ total ist, kann es kein (q_i, a) -Paar ohne definierten Folgezustand geben. Dadurch ist gesichert, dass ein DEA für jedes vorgelegte Wort aus Σ^* bezüglich $w \in L$ in endlich vielen Schritten eine Entscheidung treffen kann.

Beispiel 3.1

Wir betrachten die Sprache aller Wörter über $\{a, b\}$, die *entweder* mit beliebig vielen a's beginnen und mit genau einem b enden *oder* mit genau einem a beginnen und mit beliebig vielen b's enden. „Beliebig viele“ schließt auch den Fall ein, dass das betreffende Zeichen überhaupt nicht vorkommt: $L = \{w \mid w = a^n b \text{ oder } w = a b^n, n \geq 0\}$.

Wir definieren nun einen DEA M als Akzeptor für die Sprache L .

$M = (\{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}, \{a, b\}, \delta, q_0, \{q_1, q_2, q_4, q_6\})$ mit



⁴D.h., dass sich der Kopf der Kommunikationseinheit auf dem ersten leeren Feld unmittelbar rechts neben dem Eingabewort befindet.

Der Zustand q_5 in Beispiel 3.1 wirkt wie eine Falle (engl.: *trap state*): Sämtliche Zeichen des Eingabealphabets führen wieder zu diesem trap state. Wird dieser Zustand im Verlauf der Analyse eingenommen, gibt es keine Chance, ihn wieder zu verlassen. Da er selbst kein Endzustand ist, endet die Analyse mit der Abweisung des Eingabewortes, d.h. $w \notin L$. Solche Fallen-Zustände sind notwendig, da δ eine totale Funktion ist. Man kann „unliebsame“ oder verbotene Übergänge also nicht einfach weglassen.

Die Analyse eines Eingabewortes w kann durch eine *Folge von Konfigurationen* protokolliert werden. Eine *Konfiguration* entspricht einem „Schnappschuss“, d.h. einer Momentaufnahme der Arbeit des Automaten, und umfasst den jeweils aktuellen Zustand q_i zusammen mit dem aktuellen Restwort r_k :

$$(q_{i_1}, r_{k_1}) \vdash (q_{i_2}, r_{k_2}) \vdash (q_{i_3}, r_{k_3}) \vdash \dots \vdash (q_{i_{|w|+1}}, \varepsilon).$$

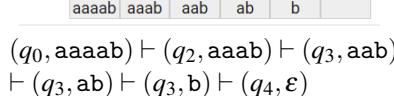
Konfigurationen werden auch gern in Tabellenform niedergeschrieben.

„Falle“
trap state

Konfiguration

FLACI

| Zustand | Restwort | Konfigurationenfolge für: a a a a b | | | | | | |
|---------|---------------|-------------------------------------|------|------|------|------|------|------|
| q_0 | aaaab | M0 | 0 | 1 | 2 | 3 | 4 | 5 |
| q_2 | aaab | | (q0) | (q2) | (q3) | (q3) | (q3) | (q4) |
| q_3 | aab | | | | | | | |
| q_3 | ab | | | | | | | |
| q_3 | b | | | | | | | |
| q_4 | ε | | | | | | | |



$$(q_0, \text{aaaab}) \vdash (q_2, \text{aaab}) \vdash (q_3, \text{aab}) \\ \vdash (q_3, \text{ab}) \vdash (q_3, \text{b}) \vdash (q_4, \varepsilon)$$

Abbildung 3.5: Konfigurationenfolge (links: Tabelle, rechts oben: mit FLACI)

Die Abarbeitung von aaaab endet in q_4 , s. Abbildung 3.5. Da q_4 ein Endzustand ist, wird das Wort aaaab vom Automaten akzeptiert und gehört somit zu der Sprache, die der Automat beschreibt. Allgemein gilt für die *von einem DEA M akzeptierte (erkannte, beschriebene, definierte) Sprache $L(M)$* :

$$L(M) = \{w \mid w \in \Sigma^* \text{ und } (q_0, w) \vdash^* (q_e, \varepsilon) \text{ und } q_e \in E\}.$$

Das Symbol \vdash^* steht für „Konfigurationenfolge beliebiger Länge“.

von DEA M
akzeptierte
Sprache $L(M)$

Didaktischer Hinweis 3.2

Der Entwurf von Automatenmodellen mit der FLACI-Komponente „Abstrakte Automaten“ ist ein Prozess, der von FLACI in vielfältiger Weise (lernertypadäquat usw.) unterstützt wird. Wie dies im Einzelnen geschieht und warum das so ist, kann hier nicht ausgeführt werden.



Computerübung 3.1

Verwenden Sie FLACI, um das Beispiel 3.1 nachzuvollziehen und experimentieren Sie sowohl mit verschiedenen Eingabewörtern als auch mit Modifikationen des DEA. Setzen Sie



den *Überprüfen*-Service von FLACI ein, um festzustellen, ob die Überführungsfunktion total ist. FLACI bietet eine automatische Vervollständigung an und lenkt alle fehlenden Übergänge zu einem trap state.

erweiterte Überführungsfunktion

Weiter oben hatten wir bereits darauf hingewiesen, dass die Überführungsfunktion $\delta : Q \times \Sigma \rightarrow Q$ eines Automaten dessen Arbeitsweise nicht vollständig beschreibt. Es müssen also noch mathematische Beschreibungen der Kopfbewegung (um genau ein Feld nach rechts in jedem Takt) und des Arbeitsendes (nach Erreichen des Wortendes) hinzugefügt werden. Dies geschieht mit der *erweiterten Überführungsfunktion* $\hat{\delta}$, lies „delta Dach“ mit

$$\hat{\delta} : Q \times \Sigma^* \rightarrow Q.$$

Sie ist eine Abstraktion der beschriebenen Arbeitsweise des DEA-Modells und nimmt als zweites Argument ein Wort anstatt ein Alphabetzeichen, wie δ .

$\hat{\delta}$ wird rekursiv definiert, wobei a für ein einzelnes Zeichen und w für das jeweilige Restwort (also das ursprüngliche Wort ohne das erste Zeichen) stehen.

$$\begin{aligned}\hat{\delta}(q, \varepsilon) &= q \\ \hat{\delta}(q, aw) &= \hat{\delta}(\delta(q, a), w)\end{aligned}$$

Die erste Zeile beschreibt das Arbeitsende (Stoppen) des Automaten für $w = \varepsilon$. In der zweiten Zeile wird der allgemeine Fall betrachtet, dass nämlich die Bearbeitung ($\hat{\delta}$) von aw nach einem Zustandswechsel von q zu $\delta(q, a)$ mit dem Restwort w fortgesetzt wird.

Unter Verwendung von $\hat{\delta}$ kann die von einem DEA akzeptierte Sprache wie folgt beschrieben werden:

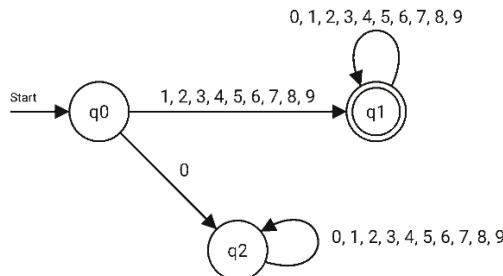
$$L(M) = \{w \mid w \in \Sigma^* \text{ und } \hat{\delta}(q_0, w) = q_e \text{ und } q_e \in E\}.$$

Beispiel 3.2 behandelt eine populäre Sprache, die von einem DEA erkannt wird.

Beispiel 3.2

Der DEA $M = (\{q_0, q_1, q_2\}, \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}, \delta, q_0, \{q_1\})$ mit folgendem δ

Beispiel



beschreibt die Sprache der Zahlwörter *ohne* Null, genauer: das Wort "0", und *ohne* Vornullen, wie in "00234". G ist eine Grammatik für $L(M)$. (Dabei handelt es sich um eine kfG,

obgleich es leicht möglich wäre, eine reguläre Grammatik für diese Sprache anzugeben.)

$$\begin{aligned}
 G &= (N, T, P, s), \\
 N &= \{\text{Zahl, Ziffern, Ziffer, ErsteZiffer}\}, \\
 T &= \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}, \\
 P &= \begin{aligned} \text{Zahl} &\rightarrow \text{ErsteZiffer Ziffern} \mid \text{ErsteZiffer}, \\ \text{Ziffern} &\rightarrow \text{Ziffer Ziffern} \mid \text{Ziffer}, \\ \text{Ziffer} &\rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9, \\ \text{ErsteZiffer} &\rightarrow 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \}, \end{aligned} \\
 s &= \text{Zahl}
 \end{aligned}$$

Übung 3.1

Geben Sie die Überführungsfunktion von M in Beispiel 3.2 als Tabelle an.

Entwickeln Sie eine reguläre Grammatik G' für die in Beispiel 3.2 definierte Sprache.



Computerübung 3.2

Verwenden Sie FLACI, um mit dem DEA aus Beispiel 3.2 zu experimentieren.



Computerübung 3.3

Zahlen mit 0x-Präfix, wie 0x2020, werden in Programmiersprachen als hexadezimale Zahlen verstanden. 2020 hingegen als dezimale. Ein DEA könnte nun verwendet werden, um zu erkennen, ob es sich bei der eingegebenen Zeichenfolge um eine Hex-Zahl handelt. Es ist zu beachten, dass bei Hex-Zahlen auch Buchstaben erlaubt sind, wie z.B. in 0x12FDAA oder 0xAF FE. Entwerfen Sie einen DEA für die Sprache aller Hex-Zahlen.



3.3 Endlicher Automat und reguläre Grammatik

In Beispiel 3.2 haben wir zwei verschiedene Definitionsmethoden (Grammatik und DEA) eingesetzt. Aus dieser Beobachtung ergibt sich eine interessante Frage: Gelingt es, zu einer gegebenen regulären Grammatik G einen äquivalenten DEA M , d.h. $L(M) = L(G)$, aus G automatisch zu gewinnen? Gemeint ist durch Anwendung eines allgemeingültigen Konstruktionsverfahrens (ohne Intelligenzeinsatz). Falls das möglich ist, erhebt sich sofort die Frage nach der Gegenrichtung: Kann man aus einem DEA eine äquivalente rG erzeugen?

Die Antworten gibt Satz 3.1.

Satz 3.1

Zu jeder regulären Grammatik G gibt es einen äquivalenten DEA M und umgekehrt.



Beweis

Dies zu beweisen, bedeutet zwei Teile zu behandeln:

Teil 1: Konstruiere aus M eine äquivalente CHOMSKY-Typ-3-Grammatik G .

Teil 2: Konstruiere aus G einen äquivalenten DEA M .



Konstruiere
G aus M

Beweis, Teil 1

Gegeben ist $M = (Q, \Sigma, \delta, q_0, E)$, gesucht ist $G = (N, T, P, s)$, mit $L(G) = L(M)$.

Offensichtlich gelten $N = Q$, $T = \Sigma$ und $s = q_0$. Die Produktionen in P ergeben sich aus δ und E : Für jedes $\delta(q_i, a) = q_j$ nehmen wir die Regel $q_i \rightarrow aq_j$ in P auf, wobei $q_i, q_j \in N$ und $a \in T$. Falls q_j ein Endzustand ist, kommt zusätzlich $q_i \rightarrow a$ hinzu. Falls $q_0 \in E$, d.h. $\epsilon \in L(M)$, wird zusätzlich $q_0 \rightarrow \epsilon$ in P aufgenommen.

Auf diese Weise wird schließlich eine zugehörige äquivalente Grammatik⁵ aufgebaut und es gilt für alle $w = a_1a_2\dots a_n \in \Sigma^*$: $w \in L(M)$ gdw.⁶ es eine Konfigurationenfolge $(q_0, a_1a_2\dots a_n) \vdash (q_{i_1}, a_2\dots a_n) \vdash \dots \vdash (q_{i_k}, \epsilon)$ gibt, wobei $q_{i_k} \in E$, gdw. $q_0 \Rightarrow a_1q_{i_1} \Rightarrow a_1a_2q_{i_2} \Rightarrow \dots \Rightarrow a_1a_2\dots a_n$ gdw. $w \in L(G)$. \square

Didaktischer Hinweis 3.3



Beweise dieser Art sind in diesem Text immer wieder anzutreffen, sodass es sich lohnt, über eine geeignete Strategie nachzudenken. Die Empfehlung lautet: *Ausschroten* der Vorgaben inkl. aller Begriffsdefinitionen. „Ausschroten“ bedeutet so viel wie „trage alles zusammen, was sich aus den Vorgaben herausholen lässt.“ Erst danach beginnt man, eine Beweiskette zu der gesuchten Seite hin aufzubauen. Häufig wird dieser zweite Schritt zu früh angegangen, sodass Beweisideen im Keim ersticken.

Beispiel 3.3



Wir konstruieren nach dem angegebenen Verfahren eine reguläre Grammatik G für den in Beispiel 3.2 angegebenen DEA zur Beschreibung der Sprache der Zahlwörter ohne Null und Vornullen. Vergleichen Sie mit Ihrer Lösung der Übungsaufgabe 3.1.

Das Ergebnis ist $G = (N, T, P, s) = (\{q_0, q_1, q_2\}, \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}, P, q_0)$ mit $P = \{$

$$q_0 \rightarrow 1q_1 \mid 1 \mid 2q_1 \mid 2 \mid 3q_1 \mid 3 \mid 4q_1 \mid 4 \mid 5q_1 \mid 5 \mid 6q_1 \mid 6 \mid 7q_1 \mid 7 \mid 8q_1 \mid 8 \mid 9q_1 \mid 9 \mid 0q_2,$$

$$q_1 \rightarrow 0q_1 \mid 0 \mid 1q_1 \mid 1 \mid 2q_1 \mid 2 \mid 3q_1 \mid 3 \mid 4q_1 \mid 4 \mid 5q_1 \mid 5 \mid 6q_1 \mid 6 \mid 7q_1 \mid 7 \mid 8q_1 \mid 8 \mid 9q_1 \mid 9,$$

$$q_2 \rightarrow 0q_2 \mid 1q_2 \mid 2q_2 \mid 3q_2 \mid 4q_2 \mid 5q_2 \mid 6q_2 \mid 7q_2 \mid 8q_2 \mid 9q_2\}$$
.

Computerübung 3.4



Führen Sie die in Beispiel 3.3 betrachtete Konvertierung des DEA in eine zugehörige reguläre Grammatik mit FLACI (Abstrakte Automaten) durch.

Optimierung der Grammatik

Nun sieht man allerdings, dass die entstandene Grammatik mit q_2 ein Nichtterminal enthält, für das es ausschließlich rekursive Regeln gibt. Im Ableitungsprozess kann es also niemals verschwinden. In M entspricht q_2 gerade der „Falle“. Sämtliche Regeln mit q_2 werden deshalb entfernt. Dies hat Konsequenzen für alle Regeln, die q_2 auf der rechten Regelseite enthalten. In unserem Beispiel ist das (neben q_2 selbst) nur q_0 , sodass wir genau die eine Regel $q_0 \rightarrow 0q_2$ ebenfalls entfernen. Optimierungsschritte dieser Art bieten sich an, um die resultierende Grammatik von unnötigem Ballast zu befreien. Dies führt zu Grammatik G' .

$$G' = (N', T', P', s') = (\{q_0, q_1\}, \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}, P, q_0) \text{ mit}$$

$$P' = \{$$

$$q_0 \rightarrow 1q_1 \mid 1 \mid 2q_1 \mid 2 \mid 3q_1 \mid 3 \mid 4q_1 \mid 4 \mid 5q_1 \mid 5 \mid 6q_1 \mid 6 \mid 7q_1 \mid 7 \mid 8q_1 \mid 8 \mid 9q_1 \mid 9,$$

$$q_1 \rightarrow 0q_1 \mid 0 \mid 1q_1 \mid 1 \mid 2q_1 \mid 2 \mid 3q_1 \mid 3 \mid 4q_1 \mid 4 \mid 5q_1 \mid 5 \mid 6q_1 \mid 6 \mid 7q_1 \mid 7 \mid 8q_1 \mid 8 \mid 9q_1 \mid 9\}$$

⁵Nachträgliche Optimierungen bzw. Umstrukturierungen sind denkbar.

⁶„gdw.“ steht für „genau dann, wenn“ (Äquivalenz). Manchmal schreibt man auch „gd“ und „w“, wenn es dem Satzbau zuträglich ist.

Computerübung 3.5

Vergleichen Sie die optimierte Lösung mit der Vereinfachung, die Sie auch mit FLACI vornehmen können.

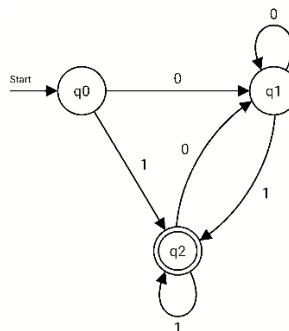
Computerübung 3.6

Gegeben ist ein DEA mit $M = (\{q_0, q_1, q_2\}, \{0, 1\}, \delta, q_0, \{q_2\})$ und



FLACI

| δ | 0 | 1 |
|----------|-------|-------|
| q_0 | q_1 | q_2 |
| q_1 | q_1 | q_2 |
| q_2 | q_1 | q_2 |



M akzeptiert alle Wörter aus $\{0, 1\}^*$, die auf mindestens eine 1 enden. Dazu gehören auch 1, 01 und 0111001, jedoch nicht das leere Wort und alle Wörter, die auf 0 enden.

Verwenden Sie das beschriebene Verfahren, um eine äquivalente reguläre Grammatik für die Sprache $L(M)$ herzustellen. Nach Vereinfachung ergibt sich $G = (\{X\}, \{0, 1\}, \{X \rightarrow 0X \mid 1X \mid 1\}, X)$. Überprüfen Sie anschließend das Ergebnis mit FLACI.



FLACI

Computerübung 3.7

Simulieren Sie das Verhalten von M angesetzt auf die o.g. Beispielwörter; auch für die, die abgewiesen werden. Benutzen Sie FLACI zur Ableitung dieser Wörter bezüglich G .



Übung

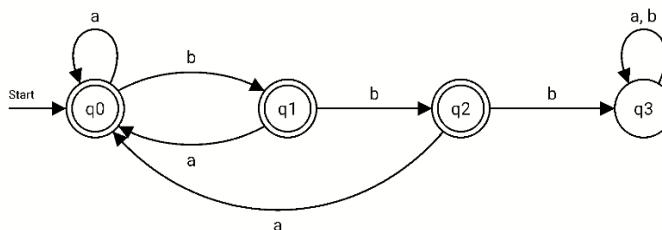
Übung 3.2

Entwerfen Sie einen DEA für die Sprache aller Wörter über $\{a, b\}$, die die Zeichenkette bbb nicht enthalten. Hinweis: Beginnen Sie beim DEA-Entwurf mit dieser „verbotenen“ Zeichenkette, die zwangsläufig in einen Fehlerzustand führt.



Vergleichen Sie anschließend Ihre Lösung mit dem folgenden Vorschlag:

$M = (\{q_0, q_1, q_2, q_3\}, \{a, b\}, \delta, q_0, \{q_0, q_1, q_2\})$ mit folgender Überführungsfunktion δ .

**Computerübung 3.8**

FLACI

Experimentieren Sie auch mit diesem DEA und der zugehörigen Grammatik unter Verwendung von FLACI.

Beweis, Teil 2

Wir setzen nun den Beweis fort, indem wir uns Teil 2 vornehmen: Gegeben ist eine reguläre

Konstruiere

M aus G

Grammatik G , gesucht ist ein äquivalenter DEA M .

Ein erster Konstruktionsversuch lässt Schwierigkeiten erkennen, denn Regeln, wie $q_2 \rightarrow aq_1 \mid aq_3 \mid \dots$ führen strikt zu der in Abbildung 3.6 dargestellten Situation, die mit einer DEA-Definition nicht vereinbar ist.

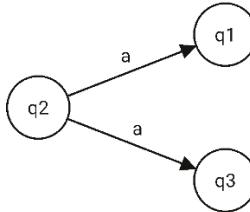


Abbildung 3.6: Verstoß gegen die DEA-Definition

Welcher Folgezustand sollte von einem DEA, der sich in Zustand q_2 befindet, nach dem Einlesen des Zeichens a eingenommen werden: q_1 oder q_3 ?

Wir kommen also nicht recht weiter und unterbrechen an dieser Stelle den Beweis zu Teil 1 des Satzes. Erst in Abschnitt 3.5 nehmen wir die Arbeit wieder auf. Dann verfügen wir über die erforderlichen Inhalte aus Abschnitt 3.4. \square

3.4 Nichtdeterministischer endlicher Automat (NEA)

NEA
RABIN
SCOTT Der Versuch, aus einer beliebigen regulären Grammatik einen äquivalenten DEA zu konstruieren, führt zu einem *nichtdeterministischen endlichen Automat (NEA)*. Dieser Automatentyp wurde von RABIN⁷ und SCOTT⁸ im April 1959 eingeführt. 1976 erhielten sie dafür den TURING-Award⁹ – die höchste Auszeichnung in der Informatik.

Abbildung 3.6 illustriert das Problem, wonach der Automat aus einem bestimmten Zustand, dort q_2 , mit ein und demselben Alphabetzeichen, dort a , in mehrere Folgezustände, q_1 und q_3 , übergehen kann. Eine Funktion kann so etwas nicht definieren, denn für jedes Argument wird jeweils *genau ein* Funktionswert erwartet, nicht zwei oder noch mehr.

⁷ Michael O. Rabin wurde 1931 in Wroclaw (Breslau) geboren. Von der dortigen Universität erhielt er 2007 einen Ehrendoktortitel. Seit 1981 ist er als Professor an der Harvard Universität (USA) tätig.

⁸ Dana S. Scott wurde 1932 in Berkeley (USA) geboren und arbeitete bis zu seiner Emeritierung im Jahre 2003 als Professor an der Carnegie Mellon University in Pittsburgh (USA).

⁹ In der Begründung heißt es: For their joint paper 'Finite Automata and Their Decision Problem', which introduced the idea of nondeterministic machines, which has proved to be an enormously valuable concept. Their (Scott & Rabin) classic paper has been a continuous source of inspiration for subsequent work in this field.

Ein kleiner Trick hilft uns weiter: Die sich ergebenden Folgezustände werden zu je einer *Menge* zusammengefasst. Diese Menge ist der jeweilige Funktionswert, wie in unserem Beispiel: $\delta(q_2, a) = \{q_1, q_3\}$.

| δ | ... | a | ... |
|----------|-----|----------------|-----|
| : | : | : | : |
| q_2 | ... | $\{q_1, q_3\}$ | ... |
| : | : | : | : |

In jedem Feld des Tabellenkörpers stehen also keine Zustände, sondern Zustandsmengen, genauer: Teilmengen von Q . Die Menge *aller* Teilmengen von Q ist die *Potenzmenge* von Q , kurz: $\wp(Q)$.

Potenzmenge

Beispiel 3.4

Die Potenzmenge von $M = \{a, b, c\}$ ist

$$\wp(M) = \{\emptyset, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\}\}.$$



Für eine endliche Menge M besitzt $\wp(M)$ genau $2^{|M|}$ Elemente. In Beispiel 3.4 sind das gerade $2^{|M|} = 2^3 = 8$.

Damit ist der Funktionscharakter von δ „gerettet“.

Aber wie sind diese Zustandsmengen-Funktionswerte im Hinblick auf die Arbeitsweise eines Automaten zu interpretieren? Bevor wir uns dieser Frage zuwenden, benötigen wir eine NEA-Definition.

Definition 3.2

Ein nichtdeterministischer endlicher Automat, kurz: NEA, wird durch ein Quintupel $M = (Q, \Sigma, \delta, q_0, E)$ definiert, wobei bis auf δ die Bedeutungen der Symbole aus der Definition des DEA übernommen werden.



Für die Überführungsfunktion eines NEA gilt: $\delta : Q \times \Sigma \rightarrow \wp(Q)$.

Auch die leere Menge $\emptyset \in \wp(Q)$ kann als Funktionswert von δ vorkommen, also $\delta(q_i, a) = \{\} = \emptyset$. Dies wird als *verbotener Übergang* interpretiert, denn für δ_i und a gibt es keinen einzigen möglichen Folgezustand.

verbotener
Übergang

Wir stellen nun die Frage nach der Arbeitsweise eines NEA-Akzeptors. Was bedeutet so etwas wie $\delta(q_2, a) = \{q_1, q_3\}$ im aktuellen Arbeitstakt? Welchen Folgezustand soll der NEA als Akzeptor einnehmen? Es gibt drei Interpretationsmuster: Ambiguous, Cloning und Backtracking.

NEA als
Akzeptor

Ambiguous Man stelle sich vor, es gäbe ein Orakel (eine gute Fee), das die Antwort auf die Frage $w \in L(M)$ kennt. Das Orakel „weiß“ genau, welchen der möglichen Folgezustände der NEA im jeweils nächsten Schritt einnehmen muss, um erfolgreich zu sein – vorausgesetzt, dass es ein erfolgreiches Ende

(Akzeptanz des Wortes) gibt. Das Orakel kann man sich als Ambiguous-Operator vorstellen¹⁰. Falls nur q_3 zur Akzeptanz des Eingabewortes führt, wird es ganz sicher q_3 und nicht etwa q_1 auswählen, d.h. $\text{amb}(q_1, q_3) = q_3$.

Cloning Für jeden Folgezustand (q_1 und q_3) wird je ein neuer Automat durch Klonen des Originalautomaten gebildet. Ein solcher Automat und der ursprüngliche sind identisch. Geklonte Automaten werden im entsprechenden Startzustand (q_1 und q_3) auf den zum Zeitpunkt ihrer Entstehung aktuellen Bandinhalt angesetzt. Sie arbeiten zeitlich parallel. Im Folgenden kann es zu weiteren Klonierungen der entsprechenden Automaten kommen. Das Eingabewort wird akzeptiert, wenn es (mindestens) einen Automaten gibt, der nach vollständigem Abtasten des Eingabewortes in einem Endzustand stoppt. Dann kann die Arbeit der anderen geklonten Automaten eingestellt werden.

Backtracking Der betrachtete Automat nimmt im aktuellen Arbeitstakt zunächst einen der möglichen Folgezustände (q_1) ein und wird auf den aktuellen Bandinhalt angesetzt. Dies geschieht wiederholt, bis das Wort entweder akzeptiert wird oder die Analyse in einer Sackgasse steckt. Im Falle einer Sackgasse wird die Arbeit des Automaten an der letzten möglichen Entscheidungsstelle für einen noch nicht verwendeten Folgezustand (q_3) fortgesetzt. Die alte Konfiguration (Bandinhalt, Kopfposition) wird rekonstruiert. Stehen keine weiteren Folgezustände zu Verfügung, muss noch weiter zurückgegangen werden. (Die Abarbeitung kann als Baum dargestellt werden, dessen Knoten die eingenommenen Zustände bilden und dessen Aufbau mit maximalem Vortrieb in die Tiefe statt in die Breite geschieht.)

verschiedene Modelle und zugehörige Abstraktionsniveaus für Nichtdeterminismus

Diese drei Modellvorstellungen haben gemein, dass ein nicht zur Sprache gehörendes Eingabewort erst dann abgelehnt werden kann, wenn sämtliche Möglichkeiten ausgereizt wurden, bzw. das Orakel mit seinem Latein am Ende ist. Beim Cloning sind das alle geklonten Automaten, die im zeitlichen Nebeneinander werkeln. Keiner von ihnen kommt zum Ziel. Bei der Backtracking-Vorstellung müssen auch wirklich alle Verzweigungen (im Zurück- und wieder Vorwärtsgehen) ausgeschöpft worden sein, bevor man sicher sein kann, dass das Wort nicht akzeptiert wird.

Für die Akzeptanz eines Wortes ist es hingegen ausreichend, wenn ein einziger Weg zum Ziel gefunden wurde.

Die Orakelvorstellung kann sehr leicht auf das Cloning oder auch auf systematische Suche (Tiefensuche, Backtracking) zurückgeführt werden. Ambiguous¹¹ bietet die höchste Abstraktion bei der Beschreibung des Konzepts des Nichtdeterminismus. Danach folgen Cloning und Backtracking in dieser Reihenfolge. In

¹⁰... und auch tatsächlich implementieren, was wir hier nicht vorhaben. Aufrufbeispiel: $\text{amb}(q_1, q_3)$ für $\delta(q_2, a) = \{q_1, q_3\}$.

¹¹Intern, also bei der Implementierung des amb-Operators, wird letztlich doch Backtracking eingesetzt.

diesem Text verwenden wir meist die *Cloning*-Vorstellung. Sie ist auch in FLACI fest eingebaut. Backtracking ist in den entsprechenden Simulationen durch eine sequentielle Anordnung der geklonten Automaten erkennbar.

Formal wird die Arbeitsweise des NEA-Modells durch die erweiterte Überführungsfunktion $\hat{\delta} : \wp(Q) \times \Sigma^* \rightarrow \wp(Q)$ beschrieben.

$$\begin{aligned}\hat{\delta}(Q', \varepsilon) &= Q', \text{ für alle } Q' \subseteq Q \\ \hat{\delta}(Q', aw) &= \bigcup_{q \in Q'} (\hat{\delta}(\delta(q, a), w))\end{aligned}$$

Gegenüber der erweiterten Überführungsfunktion von DEA wird hier nicht mit Einzelzuständen, sondern mit den entsprechenden Zustandsmengen gearbeitet. Mit $\hat{\delta}$ werden alle möglichen Folgezustände ermittelt, die sich von jedem Zustand einer Zustandsmenge aus für ein vorgelegtes Wort ergeben.

Übung 3.3

Machen Sie sich diese Definition vollständig klar und begründen Sie, inwiefern damit die Arbeitsweise eines NEA-Akzeptors beschrieben wird. Wo findet sich das Cloning wieder?

Damit kann die von einem NEA M akzeptierte Sprache $L(M)$ angegeben werden:

$$L(M) = \{w \mid w \in \Sigma^* \text{ und } \hat{\delta}(\{q_0\}, w) \cap E \neq \emptyset\}.$$

Mit anderen Worten: Die Menge $\hat{\delta}(\{q_0\}, w)$ der von q_0 aus mit w erreichbaren Zustände muss mindestens einen Endzustand q_e enthalten, kurz: $\hat{\delta}(\{q_0\}, w) \ni^{12} q_e$.

NEA haben den Vorteil, dass sie reguläre Sprachen im Allgemeinen kompakter definieren als zugehörige DEA. Es liegt auf der Hand, dass ein Fehlerzustand als „Sammelstelle“ für unzulässige Übergänge beim NEA entfällt. Für Zustände, die keine Folgezustände besitzen, ist der Funktionswert von δ die leere Menge.

Beispiel 3.5

Zum direkten Vergleich entfernen wir im Überführungsgraph in dem in Beispiel 3.1 auf Seite 54 entwickelten DEA für die Sprache $L = \{w \mid w = a^n b \text{ oder } w = ab^n, n \geq 0\}$ den Zustand q_5 mit allen zugehörigen Übergängen. Das Experimentieren wird auch von FLACI unterstützt, wenn man im DEA-Modell den Schalter „ δ als partielle Funktion erlauben“ einsetzt. Abbildung 3.7 zeigt das Ergebnis.

Computerübung 3.9

Wählen Sie **abb** als Eingabewort und simulieren Sie den Analyseprozess.

Ein Vorteil eines NEA gegenüber einem DEA ist die gute Versprachlichung. Für die einzelnen Ausprägungen der zu beschreibenden Sprache werden zugehörige Zustandsübergänge hinzugefügt.

Im NEA $M = (\{q_0, q_1, q_2, q_3, q_4\}, \{a, b\}, \delta, q_0, \{q_2, q_3, q_4\})$ in Abbildung 3.8 stehen die drei Pfade für die leicht zu identifizierenden Teilmengen b , ab^n und $a^n b$. Dieser Transparenzvorteil macht NEA attraktiv, auch wenn daraus keineswegs auf

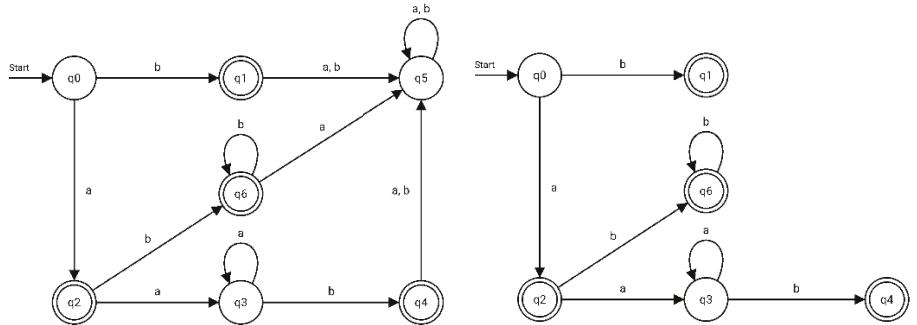
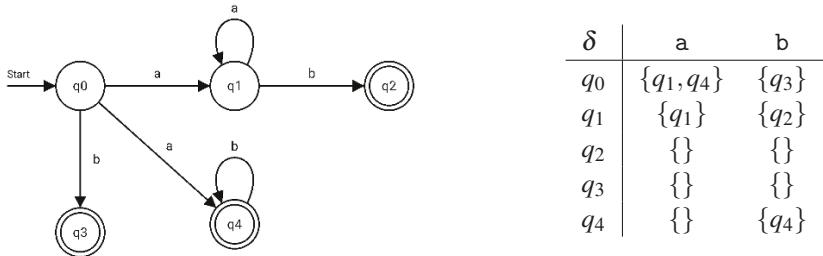
¹²Das Zeichen \ni liest man als „enthält“. $\hat{\delta}(\{q_0\}, w) \ni q_e$ bedeutet daher, dass q_e zur Menge(!) $\hat{\delta}(\{q_0\}, w)$ gehört.

FLACI verwendet Cloning.



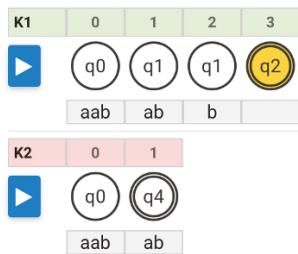
akzeptierte Sprache



Abbildung 3.7: DEA (links) und NEA (rechts) für $L = \{w \mid w = a^n b \text{ oder } w = ab^n, n \geq 0\}$ Abbildung 3.8: δ eines NEA für $L = \{w \mid w = a^n b \text{ oder } w = ab^n, n \geq 0\}$

Analysevorteile geschlossen werden kann. Im Gegenteil: Die (hier nicht relevante) Effizienz von Analysetechniken auf der Basis von NEA ist im Allgemeinen vernichtend schlecht. Simuliert man diesen NEA in Abbildung 3.8 angesetzt auf das Eingabewort aab mit FLACI, so entstehen zwei Konfigurationenfolgen K_1 und K_2 . Bereits im ersten Übergang, nämlich von q_0 aus mit a , haben wir nach dem Clo-

Konfigurationenfolge(n) für: a a b

Abbildung 3.9: Eine erfolgreiche NEA-Konfigurationenfolge (K_1) für das Wort aab

ning zwei Automaten. Wie man in Abbildung 3.9 sieht, ist nur der erste, der K_1

erzeugt, erfolgreich, was bedeutet, dass aab zu $L(M)$ gehört.

Dennoch stellt FLACI nach Möglichkeit die Arbeit aller geklonter Automaten vollständig dar. Dies geschieht aus gutem Grund: Gibt es nämlich mehr als eine erfolgreiche Konfigurationenfolge für das Eingabewort, so ist die zugrunde liegende Grammatik *mehrdeutig*. Beispielsweise gibt es für das Wort ab zwei erfolgreiche Konfigurationenfolgen.

Hat man für eine Sprache bereits eine reguläre Grammatik gefunden, kann man sie leicht in einen äquivalenten NEA konvertieren. Dies reduziert (trotz der weiter oben hervorgehobenen Beschreibungskraft von NEA) die (stets vorhandene) Fehlerträchtigkeit von „Neuentwicklungen“.

Ausgehend von einer regulären ε -freien¹³ Grammatik G ist es einfach, einen äquivalenten NEA M aufzubauen. Wir geben das allgemeine Konstruktionsprinzip an.

Gegeben: reguläre Grammatik $G = (N, T, P, s)$ ohne ε -Regeln

Gesucht: NEA $M = (Q, \Sigma, \delta, q_0, E)$, mit $L(M) = L(G)$

Klar sind $\Sigma = T$ und $q_0 = s$.

Bilde $Q = N \cup \{q_x\}$, mit $q_x \notin N$, und δ wie folgt:

Falls $A \rightarrow aB$ in P existiert, so füge $\delta(A, a) \ni B$ in δ hinzu.

Falls $A \rightarrow a$ in P existiert, so füge $\delta(A, a) \ni q_x$ in δ hinzu.

Für die Menge der Endzustände E des NEA M gilt

$$E = \begin{cases} \{q_0, q_x\}, & \text{wenn } s \rightarrow \varepsilon \text{ in } P \\ \{q_x\}, & \text{sonst} \end{cases}$$

Damit ist die Konstruktion von M aus G vollständig beschreiben. Auf einen formalen Beweis der Äquivalenz wird hier verzichtet.

Anzumerken ist allerdings, dass wir bei der oben vorgestellten NEA-Konstruktion von regulären Grammatiken ausgegangen sind, die (neben Regeln der Form $X \rightarrow a$) ausschließlich rechtslineare Regeln besitzen. Ist das Konvertierungsverfahren für Grammatiken mit linkslinearen Regeln unbrauchbar?

Erfreulicherweise gelingt die Transformation regulärer ε -freier Grammatiken mit linkslinearen Regeln in äquivalente rechtslineare immer. Dies geschieht in zwei Schritten:

1. Transformation aller¹⁴ Regeln in *linkslineare*: Regeln der Form $X \rightarrow Ya$ bleiben unverändert. Regeln der Form $X \rightarrow a$ werden ersetzt durch $X \rightarrow Ha$

Konstruiere
NEA M aus G

rechtslineare
Regeln

¹³Siehe dazu Satz 2.3, Seite 44. Die entsprechende Transformation zur Eliminierung von Regeln der Form $X \rightarrow \varepsilon$ wurde im zugehörigen Beweis – sogar für kfG – vorgeführt.

¹⁴Die Definition regulärer Grammatiken schließt eine Vermischung rechts- und linkslinearer Regeln aus.

und $H \rightarrow \epsilon$. H ist hierbei ein neues Nichtterminal. (Man braucht nur genau dieses eine neue Nichtterminal H , nicht etwa für jede Regel eines.)

2. Transformation der vorbereiteten Regeln in rechtslineare: Wähle H als neues Spitzensymbol und streiche die Regel $H \rightarrow \epsilon$. Ergänze $s \rightarrow \epsilon$, wobei s das Spitzensymbol der linkslinearen Grammatik ist. Ersetze jede Regel der Form $X \rightarrow Ya$ durch $Y \rightarrow aX$. Falls $\epsilon \in L(G)$, muss $H \rightarrow \epsilon$ wieder hinzugefügt werden.

Dies kann man auch mit weniger Text schreiben; Index l erinnert an linkslinear und Index r an rechtslinear: Gegeben sei $G_l = (N_l, T_l, P_l, s_l)$. Wir suchen $G_r = (N_r, T_r, P_r, s_r)$. Es gelten: $N_r = N_l \cup \{H\}$, $T_r = T_l$ und $s_r = H$. $P_r = \{H \rightarrow \epsilon\} \cup \{X_i \rightarrow t X_j \mid X_j \rightarrow X_i t \in P'_l\}$ mit $t \in \Sigma$, $X_i, X_j \in N_l$. P'_l entsteht aus P_l durch Substitution aller Regeln der Form $X \rightarrow t$ durch $X \rightarrow Ht$ und $H \rightarrow \epsilon$.

Beispiel 3.6

Beispiel

Gegeben sei die linkslineare reguläre Grammatik $G = (\{A, B, S\}, \{a, b, c\}, P, S)$ mit $P = \{S \rightarrow Bc \mid Ac; A \rightarrow a \mid Aa; B \rightarrow b \mid Bb\}$.

Hieraus entsteht im ersten Schritt: $G' = (\{A, B, H, S\}, \{a, b, c\}, P', S)$ mit $P' = \{S \rightarrow Bc \mid Ac; A \rightarrow Ha \mid Aa; B \rightarrow Hb \mid Bb; H \rightarrow \epsilon\}$.

Der zweite Schritt liefert: $G'' = (\{A, B, H, S\}, \{a, b, c\}, P'', H)$ mit $P'' = \{H \rightarrow aA \mid bB; A \rightarrow cS \mid aA; B \rightarrow cS \mid bB; S \rightarrow \epsilon\}$.

Didaktischer Hinweis 3.4



Linkslineare reguläre Grammatiken sind ohnehin recht „unhandlich“: Versucht man für ein vorgegebenes Wort eine Ableitung zu finden, so entpuppt sich diese Regelgestalt als *Linksrekursion*. Man „sieht“ also nicht auf das nächste zu erzeugende Terminal, sondern muss sich damit begnügen, der wiederholten Anwendung der ausgewählten Regel $X \rightarrow Xa$ zu folgen. Dadurch wird in jedem Schritt das rechts von diesem Nichtterminal X stehende Terminal a einmal in das Wort „hineingepumpt“. Wann muss man damit aufhören?

Computerübung 3.10



Verwenden Sie FLACI (Formale Grammatiken) zur Transformation der folgenden regulären linkslinearen Grammatik $G = (\{X\}, \{a, b\}, P, X)$ mit $P = \{X \rightarrow a \mid Xb\}$ für die Sprache ab+. Nehmen Sie nach Eingabe der Grammatik die folgenden Transformationsschritte vor.

1. Linksrekursionen entfernen
2. ϵ -Regeln entfernen
3. Nichtterminale umbenennen

Versuchen Sie das folgende Ergebnis zu erzielen: $G = (\{X, Y\}, \{a, b\}, P, X)$ mit $P = \{X \rightarrow a \mid aY, Y \rightarrow b \mid bY\}$

3.5 Konstruktion eines äquiv. DEA aus einem NEA

$G \rightarrow \text{NEA} \rightarrow \text{DEA}(\rightarrow G)$ Der Versuch zu zeigen, dass es zu jeder regulären Grammatik einen äquivalenten

DEA gibt, führte uns zunächst zu einem neuen Automatentyp, dem NEA, s. Abbildung 3.6 auf Seite 60. NEA leisten zur Beschreibung von Sprachen offenbar gute Dienste, wie wir in Abschnitt 3.4 gesehen haben.

Die eigentliche Beweisaufgabe ist damit jedoch nicht gelöst. Es bleibt zu zeigen, dass zu jedem NEA ein äquivalenter DEA konstruiert werden kann. Gelingt dies, so haben wir unser ursprüngliches Konstruktionsziel über den „Umweg“ NEA erreicht. Dies würde dann außerdem bedeuten, dass DEA und NEA die gleiche Leistungsfähigkeit zur Definition regulärer Sprachen besitzen.

Beweis

Die Beweisidee besteht darin, den oben angewandten Trick der Zusammenfassung mehrerer Zustände zu jeweils einer Zustandsmenge „rückgängig“ zu machen, indem jedem Element (also jeder Menge) aus $\wp(Q)$ genau ein Zustand z_i zugeordnet wird. Dann kann die Überführungsfunktion δ' des gesuchten DEA M' aus δ des betrachteten NEA $M = (Q, \Sigma, \delta, q_0, E)$ nach der Vorschrift

$$\delta'(z, a) = \bigcup_{q \in z} \delta(q, a)$$

konstruiert werden. Die anderen Bestandteile von $M' = (Q', \Sigma', \delta', z'_0, E')$ ergeben sich aus

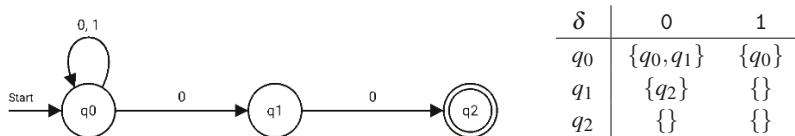
$$\begin{aligned} Q' &= \wp(Q) \text{ nach Umbenennung zu } z_i, \text{ mit } 0 \leq i < 2^{|Q|}, \\ z'_0 &= z_0 (= \{q_0\}) \text{ und} \\ E' &= \{R \mid R \subseteq Q \text{ und } R \cap E \neq \emptyset\} \text{ nach Umbenennung zu } z_i. \end{aligned}$$

□

Die sehr knappe Darstellung dieser (konstruktiven) Beweises erfordert ein Beispiel.

Beispiel 3.7

Wir zeigen die Konstruktion eines äquivalenten DEA $M' = (Q', \Sigma', \delta', z'_0, E')$ aus einem NEA $M = (\{q_0, q_1, q_2\}, \{0, 1\}, \delta, q_0, \{q_2\})$, mit



$$\wp(Q) = \underbrace{\{q_0\}}_{z_0}, \underbrace{\{q_1\}}_{z_1}, \underbrace{\{q_2\}}_{z_2}, \underbrace{\{q_0, q_1\}}_{z_3}, \underbrace{\{q_0, q_2\}}_{z_4}, \underbrace{\{q_1, q_2\}}_{z_5}, \underbrace{\{q_0, q_1, q_2\}}_{z_6}, \underbrace{\emptyset}_{z_7}$$

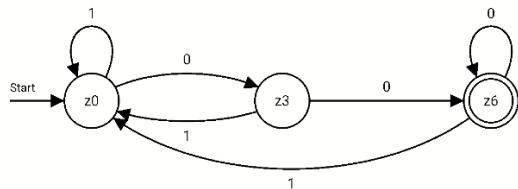
Die Reihenfolge der Elemente (Teilmengen) in $\wp(Q)$ spielt bei deren Umbenennung keine Rolle.

$$\begin{aligned} \text{z.B. } \delta'(z_3, 0) &= \delta'(\{q_0, q_1\}, 0) = \delta(q_0, 0) \cup \delta(q_1, 0) \\ &= \{q_0, q_1\} \cup \{q_2\} = \{q_0, q_1, q_2\} \\ &= z_6 \end{aligned}$$

Die vollständige Tabelle für δ' (links) kann anschließend vereinfacht werden (rechts). Hierfür sind Zustände, zu denen es keinen Übergang gibt (wie z_5) oder von denen kein Übergang wegführt (wie z_7), zu streichen. Streichungen können weitere Streichungen hervorrufen. Dass der optimierte DEA wie der gegebene NEA am Ende ebenfalls drei Zustände besitzt, ist nicht allgemeingültig und liegt am Beispiel.

| δ' | 0 | 1 |
|-----------|-------|-------|
| z_0 | z_3 | z_0 |
| z_1 | z_2 | z_7 |
| z_2 | z_7 | z_7 |
| z_3 | z_6 | z_0 |
| z_4 | z_3 | z_0 |
| z_5 | z_2 | z_7 |
| z_6 | z_6 | z_0 |
| z_7 | z_7 | z_7 |

| δ' | 0 | 1 |
|-----------|-------|-------|
| z_0 | z_3 | z_0 |
| z_3 | z_6 | z_0 |
| z_6 | z_6 | z_0 |



Wir erhalten den DEA $M' = (Q', \{0, 1\}, \delta', z'_0, E')$ mit

$$Q' = \{z_0, z_3, z_6\}$$

$z'_0 = z_0$, denn q_0 ist der Anfangszustand von M .

$E' = \{z_6\}$, denn q_2 ist der einzige Endzustand von M .

Bei Bedarf können die Zustände z_i zu geeigneten q_j umbenannt werden. Im Beispiel gilt offensichtlich $w \in L(M)$ genau dann, wenn $w \in L(M')$.



Didaktischer Hinweis 3.5

Bei $|Q| \geq 4$ und damit $|\wp(Q)| \geq 2^4$ kann die mitunter recht umfangreiche Tabelle durch eine geschickte Berechnungsreihenfolge unvollständig ausgefüllt werden. Im Beispiel hätte man zuerst $\delta'(z_0, 1)$, danach $\delta'(z_3, 1)$ und schließlich $\delta'(z_6, 1)$ berechnet.



Computerübung 3.11

Modellieren Sie den in Beispiel 3.7 vorgegebenen NEA mit FLACI (Abstrakte Automaten). Lassen Sie diesen in einen äquivalenten DEA konvertieren und vergleichen Sie das Resultat mit dem in diesem Beispiel konstruierten. Zum Vergleich der Überführungsfunktionen ist es ggf. empfehlenswert die Anordnung des Graphen durch Ziehen der Zustandsknoten mit der Maus anzupassen. Lassen Sie FLACI anschließend aus dem DEA eine äquivalente Grammatik generieren und vereinfachen Sie diese wiederum mit FLACI, sodass nur noch zwei Nichtterminale übrig bleiben. Konvertieren Sie diese Grammatik zu einem NEA mit Handrechnung und zur Kontrolle mit FLACI. Es entsteht ein äquivalenter NEA, der mit der anfänglichen NEA-Vorgabe nicht identisch ist.



Übung 3.4

Gegeben sei die reguläre Grammatik $G = (\{A, B, S\}, \{\mathbf{a}, \mathbf{b}\}, P, S)$, mit $P = \{S \rightarrow \mathbf{a}A \mid \mathbf{a}B, A \rightarrow \mathbf{b}A \mid \epsilon, B \rightarrow \mathbf{a}B \mid \mathbf{b}\}$.

- Zeigen Sie, dass G mehrdeutig ist.
- Stellen Sie ϵ -Freiheit her.
- Konstruieren Sie einen äquivalenten NEA M .
- Konstruieren Sie aus M einen äquivalenten DEA.
- Überführen Sie den erzeugten DEA in einen äquivalenten NEA und vergleichen Sie diesen mit M .
- Versuchen Sie, eine äquivalente reguläre Grammatik G' zu finden, die eindeutig ist.

Fazit: Typ-3-Grammatiken, DEA und NEA sind äquivalente Beschreibungsmittel regulärer Sprachen. Alle auch mit FLACI möglichen Konvertierungen sind in Abbildung 3.10 dargestellt. Während formale Grammatiken Wörter ab- bzw. herleiten

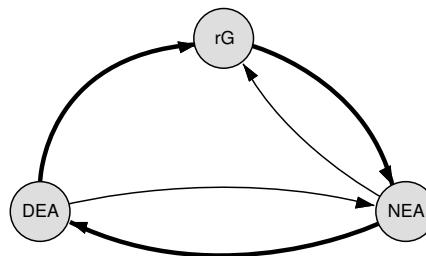


Abbildung 3.10: Konvertierungskette (fett): $rG \rightarrow NEA \rightarrow DEA \rightarrow rG$

und damit eher ein *generierendes* Vorgehen verwirklichen, sind abstrakte Automaten eher *analysierend* bzw. *akzeptierend*. Mit den *regulären Ausdrücken* kommt in Kapitel 4 noch ein weiteres *deskriptives* (beschreibendes) Konzept hinzu.

regulärer Ausdruck

Diese Gleichmächtigkeit von Definitionsmethoden für reguläre Sprachen ermöglicht es uns in konzeptionell völlig unterschiedlichen „Welten“ zu denken, um ein und dasselbe auszudrücken.

Computerübung 3.12

Gegeben ist ein NEA M mit $M = (\{q_0, q_1, q_2, q_3\}, \{a, b\}, \delta, q_0, \{q_1, q_3\})$, $\delta(q_0, a) = \{q_1, q_2\}$, $\delta(q_0, b) = \emptyset$, $\delta(q_1, a) = \emptyset$, $\delta(q_1, b) = \emptyset$, $\delta(q_2, a) = \emptyset$, $\delta(q_2, b) = \{q_3\}$, $\delta(q_3, a) = \{q_2\}$, $\delta(q_3, b) = \emptyset$.

Computer Übung

FLACI

- Geben Sie δ als Graph an.
- Stellen Sie $L(M)$ fest.
- Konstruieren Sie einen äquivalenten DEA für $L(M)$.

Verwenden Sie FLACI.

Computerübung 3.13

Gegeben ist ein NEA $M = (Q, \Sigma, \delta, q_0, E)$ mit $Q = \{q_0, q_1, q_2, q_3, q_4\}$, $\Sigma = \{a, b\}$, $E = \{q_2, q_4\}$ und δ ist bestimmt durch folgende Tafel:

Computer Übung

FLACI

| δ | a | b |
|----------|----------------|----------------|
| q_0 | $\{q_0, q_3\}$ | $\{q_0, q_1\}$ |
| q_1 | $\{\}$ | $\{q_2\}$ |
| q_2 | $\{q_2\}$ | $\{q_2\}$ |
| q_3 | $\{q_4\}$ | $\{\}$ |
| q_4 | $\{q_4\}$ | $\{q_4\}$ |

- (a) Geben Sie δ als Graph an.
 (b) Stellen Sie fest, ob der Automat die Eingabe $abaab$ akzeptiert, und geben Sie im positiven Fall eine zugehörige Konfigurationenfolge an. FLACI unterstützt Sie dabei.
 (c) Welche Sprache akzeptiert der Automat?



Computerübung 3.14

Gegeben ist die Sprache S , die aus allen Wörtern der Form a^n mit $n \geq 0$ besteht, deren Längen $|a^n| = n$ durch 3 oder durch 4 (oder durch beide) teilbar ist. Geben Sie einen NEA für S an. Setzen Sie FLACI ein.



Computerübung 3.15

Geben Sie einen DEA für die Menge der ganzen Zahlen (als Zahlwörter) an. Konstruieren Sie daraus eine zugehörige reguläre Grammatik, aus der Sie in einem dritten Schritt einen entsprechenden NEA konstruieren. Schließlich entwickeln Sie aus diesem NEA einen DEA und vergleichen ihn mit Ihrer Lösung am Anfang. Lassen Sie sich von FLACI entsprechende Musterlösungen bereitstellen.



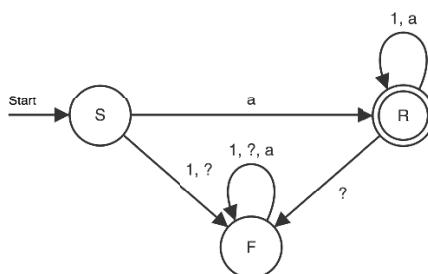
Computerübung 3.16

Erstellen Sie einen DEA für die Sprache aller Wörter über $\{a, b, c\}$, in denen jedes der drei Alphabetzeichen mindestens einmal vorkommt. Verwenden Sie FLACI.



Computerübung 3.17

Gegeben sei ein Automat $M = (\{S, R, F\}, \{a, ?, 1\}, \delta, S, \{R\})$ mit folgender Überfunktionsfunktion:



- (a) Geben Sie die vollständige Definition von M an und stellen Sie den Automatentyp fest.
 (b) Entscheiden Sie, ob M die Wörter $aa11a$, $aaa1a$ und $1a1a1a1a$ akzeptiert zuerst durch Handrechnung und anschließend mittels FLACI.
 (c) Wenden Sie das Verfahren zur Entwicklung einer zu M gehörenden regulären Grammatik G an und vergleichen Sie Ihr Ergebnis mit der Grammatik, die von FLACI erzeugt wird. Leiten Sie einige Beispielwörter ab.

3.6 Abschlusseigenschaften regulärer Sprachen

Wir haben nun drei vollkommen gleichberechtigte Beschreibungsmethoden für reguläre Sprachen kennengelernt: Abstrakte Automaten (DEA, NEA) und reguläre Grammatiken. Sie beschreiben Mengen, die im Allgemeinen abzählbar unendlich viele Elemente besitzen.

Aus der Mathematik wissen wir, welche Ergebnisse entstehen, wenn man zwei abzählbar unendliche Mengen miteinander verknüpft, z. B. vereinigt, bzw. eine einstellige Operation anwendet, wie etwa das Komplement oder die Potenzmenge.

Liefert die Anwendungen bestimmter Operationen über regulären Sprachen wieder reguläre Sprachen? Nur wenn dem so ist, können die genannten Beschreibungsmittel für reguläre Sprachen auf Zwischenergebnisse angewandt werden. Wir nennen diese Eigenschaft eines Verknüpfungsgebildes (M, \circ) *Abgeschlossenheit* und sagen „ M ist *abgeschlossen* unter \circ “, wenn die Anwendung von \circ auf beliebige Elemente aus M ausschließlich Ergebnisse in M erzeugt.

Wir untersuchen im Folgenden einige der *Abschlusseigenschaften* (closure properties) regulärer Sprachen. Man kann zeigen, dass reguläre Sprachen unter dem *Komplement*, der *Vereinigung*, der *Differenz*, dem *Durchschnitt*, der *Verkettung* und dem *Kleene-Stern-Produkt* abgeschlossen sind.

Als erstes zeigen wir die Abgeschlossenheit der Menge aller regulären Mengen unter dem *Komplement*. Dabei handelt es sich um das *relative Komplement* bezogen auf eine Grundmenge Σ^* : Die Komplementmenge \overline{M} einer Menge M enthält alle Elemente $x \in \Sigma^*$, die nicht zu M gehören. D.h. $\overline{M} = \Sigma^* \setminus M$.

Abgeschlossenheit

Satz 3.2

Ist L eine reguläre Sprache über Σ , so ist auch $\overline{L} = \Sigma^* \setminus L$ regulär.

Komplement



Beweis

Sei $A = (Q, \Sigma, \delta, q_0, E)$ ein DEA, der die Sprache $L(A)$ akzeptiert. Wir suchen einen DEA B für die Sprache $L(B) = \overline{L(A)} = \Sigma^* \setminus L(A)$. Dies ist offenbar gerade der DEA $B = (Q, \Sigma, \delta, q_0, Q \setminus E)$. Das bedeutet, dass B mit A vollkommen übereinstimmt, bis auf den Fakt, dass die Nichtend-/End-Zustände von A zu den End-/Nichtend-Zuständen von B werden. Dann gilt $w \in L(B)$ genau dann, wenn $\hat{\delta}(q_0, w) \in Q \setminus E$. Dies ist genau dann der Fall, wenn $w \notin L(A)$. \square

Didaktischer Hinweis 3.6

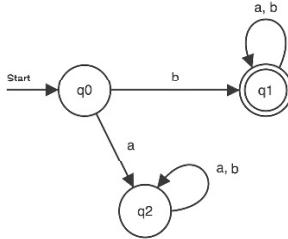
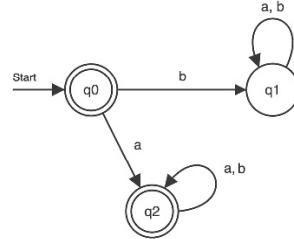
Hier kommt uns zugute, dass wir DEA mit einer *totalen* Funktion δ definiert haben. Damit ist gesichert, dass $\hat{\delta}(q_0, w)$ entweder in E oder in $Q \setminus E$ liegt.



Für die Bildung des Komplement-DEA von A gibt es in FLACI keine Transformation per Knopfdruck. Man kann dies manuell leicht dadurch erreichen, dass man die Klassifikation der Zustände eines gegebenen DEA verändert, d.h. aus Nichtendzustand wird Endzustand und umgekehrt.

Beispiel 3.8

Gegeben ist $L(A) = \{w \mid w = b \circ v, v \in \{a, b\}^*\}$, s. Abbildung 3.11. Folglich ist $L(B) = \overline{L(A)}$ die Menge aller Wörter über $\{a, b\}$, die mit a beginnen, inkl. das leere Wort ϵ :
 $L(B) = \{w \mid w = a \circ v, v \in \{a, b\}^*\} \cup \{\epsilon\}$, s. Abbildung 3.12.

Abbildung 3.11: $L(A)$ Abbildung 3.12: $L(B) = \overline{L(A)}$

Wir zeigen nun die Abgeschlossenheit der Menge der regulären Sprachen unter Durchschnitt

Satz 3.3

Sind L_1 und L_2 reguläre Sprachen, so ist auch $L_1 \cap L_2$ regulär.

**Beweis**

Gegeben sind $L_1 = L(M_1)$ und $L_2 = L(M_2)$ mit den DEA $M_1 = (Q_1, \Sigma, \delta_1, q_0, E_1)$ und $M_2 = (Q_2, \Sigma, \delta_2, p_0, E_2)$.

Die Beweisidee besteht darin, einen DEA M zu konstruieren, der M_1 und M_2 durch ein logisches UND verbindet und simuliert, sodass $L(M) = L_1(M_1) \cap L_2(M_2)$ gilt.

Für je zwei Übergänge $\delta_1(r, a) = x$ und $\delta_2(z, a) = y$ enthält M den Übergang $\delta((r, z), a) = (x, y)$, wobei man die Zustände von M als Zustandspaare, wie z.B. $q_i = (x, y)$, schreibt.

Der Anfangszustand von $M = (Q_1 \times Q_2, \Sigma, \delta, (q_0, p_0), E_1 \times E_2)$ ist (q_0, p_0) .

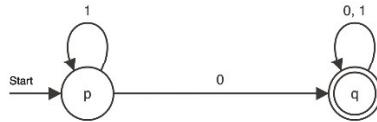
Für δ gilt $\delta((r, z), a) = (\delta_1(r, a), \delta_2(z, a))$. M akzeptiert ein Wort $w \in L(M)$, genau dann, wenn $w \in L(M_1)$ und $w \in L(M_2)$, d.h. $\hat{\delta}((q_0, p_0), w) = (\hat{\delta}_1(q_0, w), \hat{\delta}_2(p_0, w))$ mit $\hat{\delta}_1(q_0, w) \in E_1$ und $\hat{\delta}_2(p_0, w) \in E_2$. \square

Dieser konstruktive Beweis gibt uns die Möglichkeit, aus zwei DEA durch Produktbildung einen DEA zu berechnen, der nur Wörter akzeptiert, die zu beiden zugehörigen regulären Sprachen gehören.

Sollten die Alphabete der beiden zu kombinierenden DEA nicht völlig übereinstimmen, so verwendet man die Vereinigungsmenge der beiden Alphabete und muss einen Trap-Zustand nachrüsten. Dies erledigt FLACI per Knopfdruck. Sind die Alphabete der beiden zu multiplizierenden DEA disjunkt, so entsteht ein DEA, der kein einziges Wort akzeptiert.

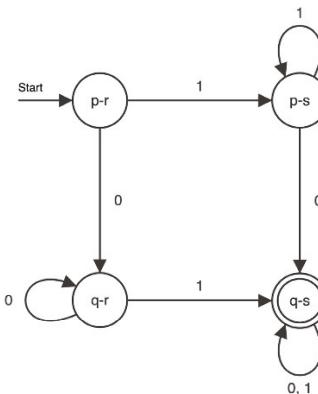
**Beispiel 3.9**

Gegeben sind die DEA $M_1 = (\{p, q\}, \{0, 1\}, \delta_1, p, \{q\})$ und $M_2 = (\{r, s\}, \{0, 1\}, \delta_2, r, \{s\})$ mit

Abbildung 3.13: δ_1 Abbildung 3.14: δ_2

Zur Berechnung von $M = M_1 \times M_2$ wie im Beweis von Satz 3.3 bestimmen wir zuerst die $|\{p, q\}| \cdot |\{r, s\}| = 2 \cdot 2 = 4$ Zustände von M : $(p, r), (p, s), (q, r)$ und (q, s) , die in FLACI mit $p\text{-}r, p\text{-}s, q\text{-}r$ und $q\text{-}s$ bezeichnet werden.

$$\begin{aligned}\delta((p, r), 0) &= (\delta_1(p, 0), \delta_2(r, 0)) = (q, r) \\ \delta((p, s), 0) &= (q, s) \\ \delta((q, r), 0) &= (q, s) \\ \delta((q, s), 0) &= (q, s) \\ \delta((p, r), 1) &= (p, s) \\ \delta((p, s), 1) &= (p, s) \\ \delta((q, r), 1) &= (q, s) \\ \delta((q, s), 1) &= (q, s)\end{aligned}$$

Abbildung 3.15: δ von $M = M_1 \times M_2$

(p, r) ist der Startzustand von $M = M_1 \times M_2$, da p der Startzustand von M_1 und r der Startzustand von M_2 sind. Da M_1 und M_2 nur jeweils genau einen Endzustand besitzen, hat M ebenfalls genau einen Endzustand, nämlich (q, s) . Die vollständige Definition lautet: $M = (\{(p, r), (p, s), (q, r), (q, s)\}, \{0, 1\}, \delta, (p, r), \{(q, s)\})$ mit δ aus Abbildung 3.15.

Diesen Ergebnis-DEA M können wir auch mit FLACI erzeugen. Hierzu öffnet man den ersten DFA und wählt „Automaten kombinieren“ aus „Transformieren“. Aus der Liste der (alphabetgleichen) kombinierbaren DFA greift man den für M_2 heraus. Um den dadurch überschriebenen ursprünglichen DFA M_1 zurück zu gewinnen, muss man ihn entweder vorher duplizieren oder mit der Änderungstaste („letzte Änderung zurücknehmen“) wiederbeschaffen.

Simulationen der DFA M_1, M_2 und M legen die folgenden regulären Sprachen nahe:
 $L(M_1) = 1*0(0|1)*, L(M_2) = 0*1(0|1)*$ sowie $L(M) = (1+0(0|1)*)(0+1(0|1)*)$.
FLACI liefert $L(M) = (0+1|1+(0|0(0|1)*))|0+1(0|1)*$. Die beiden regulären Ausdrücke für $L(M)$ sind äquivalent. Auch dies kann man mit FLACI feststellen.

$M_1 \times M_2$ mit
FLACI

Satz 3.3 ist nicht nur von formalem theoretischen Interesse wie in Beispiel 3.9, sondern findet auch Anwendung im Bereich der *Spieleprogrammierung*. Dabei werden Spielszenarien mit endlichen Automaten, die man aus DEA oder NEA für reguläre Sprachen durch (ggf. mehrfache) Produktbildung gewinnt, beschrieben.



Didaktischer Hinweis 3.7

Das folgende Beispiel 3.10 greift eine wohlbekannte Variante des Problems einer Flussüberquerung auf. Gern wird es für didaktische Zwecke in der logischen Programmierung genutzt. Um es für Operationen mit endlichen Automaten zu erschließen, wird die Fragestellung ein wenig modifiziert.



Flussüberquerungsproblem

Beispiel 3.10

Ein schon seit dem 9. Jahrhundert bekanntes *Flussüberquerungsproblem* erzählt man sich beispielsweise so: Ein Mann (👤) möchte mit seinem Boot einen Fluss überqueren und dabei einen Wolf (🐺), eine Ziege (🐐) und einen Kohlkopf (ccoli) mitnehmen. Neben dem Mann kann das Boot je Überfahrt nur *höchstens* einen „Passagier“, d.h. entweder 🐺 oder 🐐 oder 🥬, mitnehmen. Außerdem muss darauf geachtet werden, dass zu keinem Zeitpunkt ein Paar, bei denen der eine den anderen fressen könnte, an einem Ufer unbeaufsichtigt zurückbleibt. Zu Beginn befinden sich 🐺, 🐐, 🤷‍♂️ bzw. 🥬 am linken Ufer, während sie alle am Ende auf der rechten Flusseite sein sollen.

1. Gesucht sind alle möglichen in diesem Sinne zulässigen Passagierfolgen. Dies sind gerade die Wörter der Sprache des Fährmanns.
2. Wir nehmen an, dass die Ziege sehr leicht seekrank wird, sodass sie im Rahmen einer Flussüberquerung nur höchstens dreimal übersetzen darf. Gesucht ist ein DEA, der genau diese eingeschränkte Sprache akzeptiert.

Um die erste Aufgabe zu lösen, stellen wir zunächst sämtliche Zustände zusammen. Diese ergeben sich aus der Angabe aller Passagiere (🐺, 🐐 und 🤷‍♂️) auf der linken Uferseite. Die Passagiere, die sich dann jeweils auf der rechten Seite befinden, ergeben sich zwangsläufig, sodass wir sie nicht extra angeben müssen. Insgesamt sind das $\binom{4}{0} + \binom{4}{1} + \binom{4}{2} + \binom{4}{3} + \binom{4}{4} = 1 + 4 + 6 + 4 + 1 = 16$ (Kombinationen ohne Wiederholung) Zustände, von denen sechs aufgrund des Fressverhaltens von 🐺 und 🐐 verboten sind, wie in Tabelle 3.1 dargestellt.

Es bleiben $16 - 6 = 10$ Zustände übrig. Trägt man alle Übergänge zwischen diesen Zuständen ein und ergänzt einen für die DEA-Definition erforderlichen trap state (in FLACI per Knopfdruck), erhält man den Überführungsgraph aus Abbildung 3.16.

Bei den Markierungen der Übergänge ist zu beachten, dass das 🏠-Symbol für Leefahrten steht: Der Mann wechselt mit seinem Boot das Ufer ohne dabei einen Passagier (🐐, 🐺 oder 🥬) zu befördern. q_0 ist der Startzustand und q_{15} ist der Endzustand.

Es gibt unendlich viele Wörter, die der DEA für die Fährmannssprache akzeptiert. 🐐🐺🐺🐺🐺 ist ein solches. Die Wörter 🐐🐺🐺🐺🐺 und 🐐🐺🐺🐺🐺🐺 sind die beiden kürzesten. Sie haben die Länge 7. Man kann sie in Abbildung 3.16 leicht identifizieren.

Computerübung 3.18

Entsprechende Simulationen mit FLACI werden dringend empfohlen. Entsprechende Emo-



| linkes Ufer | rechtes Ufer | Zustand | Bemerkungen |
|-------------|--------------|----------|-----------------------------------|
| 👤, 🐑, 🐑, 🍃 | - | q_0 | |
| 🐺, 🐑, 🍃 | 👤 | q_1 | verboten, da 🐑 frist 🍃, 🐑 frist 🐑 |
| 🐺, 🍃 | 👤, 🐑 | q_2 | |
| 🐺, 🍃 | 👤, 🍃 | q_3 | verboten, da 🐑 frist 🍃 |
| 🐺, 🍃 | 👤, 🐑 | q_4 | verboten, da 🐑 frist 🍃 |
| 👤, 🐑, 🍃 | 🐺 | q_5 | |
| 👤, 🍃 | 🐺, 🍃 | q_6 | |
| 🐺 | 👤, 🐑, 🍃 | q_7 | |
| 葎 | 👤, 🐑, 🍃 | q_8 | |
| 🐺 | 👤, 🐑, 🍃 | q_9 | |
| 👤, 🐑 | 🐺, 🍃 | q_{10} | |
| 👤, 🐑, 🐑 | 🐺 | q_{11} | |
| 👤 | 🐺, 🐑, 🍃 | q_{12} | verboten, da 🐑 frist 🍃 |
| 葎 | 🐺, 🐑 | q_{13} | verboten, da 🐑 frist 🐑 |
| 👤, 🐑 | 🐺, 🍃 | q_{14} | verboten, da 🐑 frist 🍃 |
| - | 👤, 🐑, 🍃 | q_{15} | |

Tabelle 3.1: Zustände des Flussüberquerungsproblems

jis können z.B. von <https://www.getemojis.net/> kopiert (copy) und wie andere Alphabetzeichen in FLACI eingefügt (paste) und verwendet werden.

Zur Lösung der zweiten Aufgabe formalisieren wir eine „seekranke“ Ziege mittels DEA, s. Abbildung 3.17.

Der zugehörige DEA akzeptiert nur Wörter über $\{👤, 🐑, 🐑, 🍃\}$, die das Zeichen 🐑 höchstens dreimal enthalten.

Führt man nun die Produktbildung der beiden DEA mit δ in den Abbildungen 3.16 bzw. 3.17 aus, so ergibt sich für den Ergebnis-DEA zunächst eine unübersichtliche Überführungsfunktion. Dies kann mit FLACI sehr gut repräsentiert werden. Nach dem Duplizieren und Multiplizieren (Kombinieren) kann durch Minimalisierung (s. Abschnitt 3.7) die Zahl der Zustände minimiert werden. Schließlich kann ein trap state entfernt werden, um die Übersichtlichkeit der Darstellung der DEA-Überführungsfunktion zu erhöhen. Mit einem DEA mit partieller Überführungsfunktion sollten jedoch keine weiteren Transformationen vorgenommen werden.

Simulationen mit dem so gewonnenen DEA beachten die Flussüberquerungsregeln *und* lassen keine Wörter zu, die 🐑 mehr als dreimal enthalten. Die beiden oben angegebenen kürzesten Wörter werden ebenfalls akzeptiert. Entschließt sich der Fährmann zu einer „Extrarunde“ mit der Ziege, so ergibt sich das Wort 👤🐺🐺🐺🐺🐺. Es wird nicht akzeptiert, da es fünfmal das Zeichen 🐑 enthält.

Computerübung 3.19

Illustrieren Sie mit FLACI, dass die Fährmannssprache mit einer *sehr* seekranken Ziege kein einziges Wort enthält. Eine sehr seekranke Ziege kann ohne gesundheitliches Risiko pro Flussüberquerung nur höchstens zwei Überfahrten mitmachen.

Zur Vereinfachung der Simulation mit FLACI ist es möglich, Wörter, die von dem betrachteten DEA akzeptiert werden, zufällig erzeugen zu lassen.

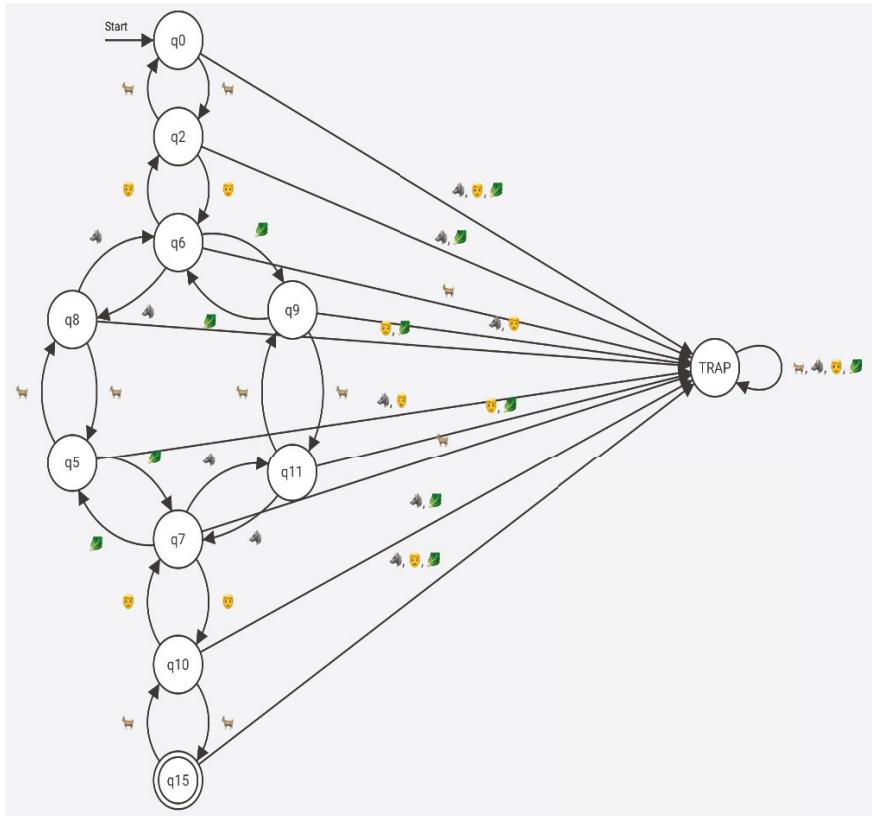


Abbildung 3.16: Zustandsübergänge eines DEA für das Flussüberquerungsproblem

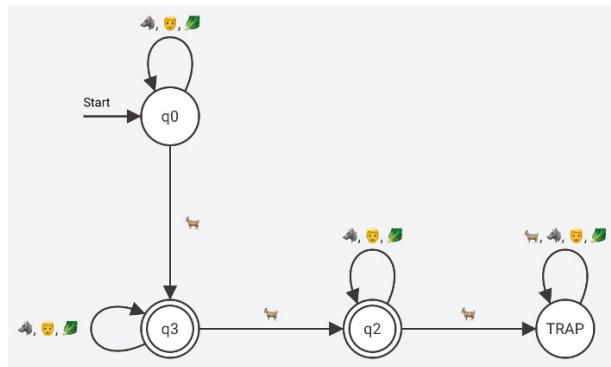


Abbildung 3.17: Überführungsfunktion eines DEA, der die eingeschränkten Überfahrten (max. 3) einer Ziege formalisiert

3.7 Satz von Myhill und Nerode

In Abschnitt 3.6 haben wir sichergestellt, dass bestimmte Operationen mit regulären Sprachen wiederum reguläre Sprachen hervorbringen, die Menge der Typ-3-Sprachen also nicht verlassen.

Gibt es eine Möglichkeit von einer vorliegenden Sprache (gegeben oder erzeugt) festzustellen, ob sie regulär ist oder nicht?

Beispiel 3.11

Wir scheitern kläglich, wenn wir versuchen, für L mit $L = \{w \mid w = a^n b^n, n \geq 1\}$ einen DEA oder NEA M mit $L = L(M)$ anzugeben: Mit ansteigenden Werten für n müssten immer neue Zustände hinzugefügt werden. Es gibt also keinen NEA, der (ohne „prozessbegleitende Anpassung“) die angegebene Sprache erkennt.

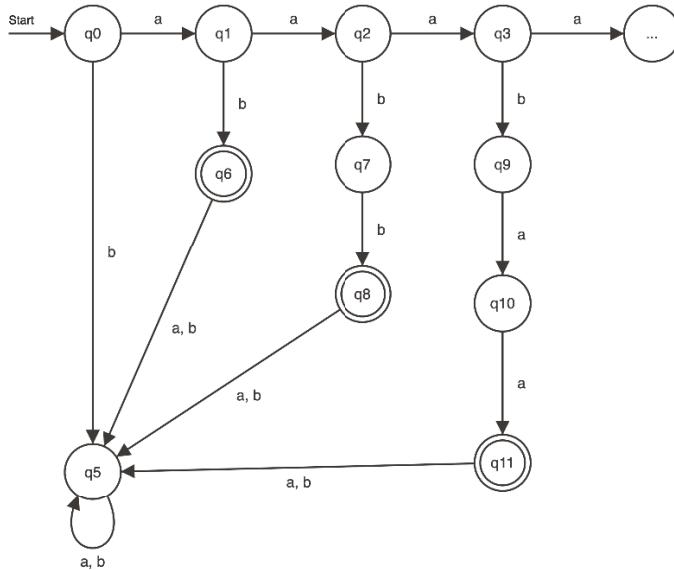


Abbildung 3.18: $L = \{w \mid w = a^n b^n, n \geq 1\}$ ist nicht regulär.

Natürlich ist es unbefriedigend, bei der Beurteilung der Regularität von Sprachen auf solche Experimente wie in Abbildung 3.18 angewiesen zu sein.

Der folgende Satz liefert ein notwendiges und hinreichendes Kriterium für die Regularität einer formalen Sprache.

**Satz 3.4**

Eine Sprache L ist genau dann regulär, wenn der Index^a von R_L endlich ist. (Satz von MYHILL^b und NERODE) Dabei ist R_L eine zweistellige Relation in der Wortmenge Σ^* , mit $L \subseteq \Sigma^*$. Für beliebige Zeichenketten $x, y \in \Sigma^*$ gilt xR_Ly genau dann, wenn für alle Zeichenketten $z \in \Sigma^*$ gilt: $xz \in L \Leftrightarrow yz \in L$.

^aUnter dem Index einer Äquivalenzrelation R in M versteht man die Anzahl der durch R definierten Äquivalenzklassen in M .

^bJohn R. Myhill, 11.08.1923 - 15.02.1987; Anil Nerode, geb. 04.06.1932;

http://www.buffalo.edu/cas/math/news-events/myhill.html#collapsible_0

Satz von
MYHILL und

NERODE

Beweis

Teil 1: Wenn L regulär ist, dann ist der Index von R_L endlich.

Teil 2: Wenn der Index von R_L endlich ist, dann ist L regulär.

Der Beweis für beide Teile wird im nachfolgenden Text erbracht. \square

R_L R_L ist die NERODE-Relation in Σ^* . Sie ist auf L beschränkt.

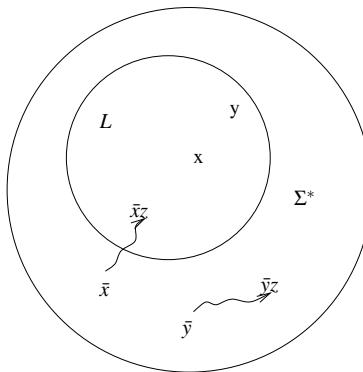


Abbildung 3.19: NERODE-Relation

xR_Ly gilt genau dann, wenn die Wörter xz und yz für alle $z \in \Sigma^*$, also auch $z = \epsilon$, entweder beide in L oder beide nicht in L , also in $\Sigma^* \setminus L$, liegen.

In Abbildung 3.19 sind zwei Wörter \bar{x} und \bar{y} eingezeichnet, für die $\bar{x}R_L\bar{y}$ nicht gilt, d.h. $(\bar{x}, \bar{y}) \notin R_L$, denn es gibt (offensichtlich) ein $z \in \Sigma^*$, sodass $\bar{x}z \in L$ und $\bar{y}z \notin L$.

Übung 3.5

Weisen Sie nach, dass R_L eine (rechtsinvariante¹⁵) Äquivalenzrelation ist.

Eine Äquivalenzrelation in Σ^* erzeugt eine vollständige Zerlegung von Σ^* in paarweise disjunkte Teilmengen, die Äquivalenzklassen $[x]_{R_L} = \{w \mid w \in \Sigma^* \text{ und } wR_Lx\}$ heißen. Deren Anzahl heißt Index der Äquivalenzrelation.

¹⁵Eine Äquivalenzrelation R heißt rechtsinvariant, wenn $xRy \Rightarrow \forall z \in \Sigma^* : xzRyz$.

Äquivalenzrelation
Äquivalenzklassen





Didaktischer Hinweis 3.8

.... hat im gleichen Jahr Geburtstag wie ...“ ist ein leicht verständliches Beispiel für eine Äquivalenzrelation aus unserer Lebenswirklichkeit. Es ist eine Relation über der *endlichen* Menge aller Menschen auf der Erde.

Die Zahltentheorie beschäftigt sich mit den ganzen Zahlen. Dies ist, wie Σ^* , eine abzählbar *unendliche* Menge. Als klassisches Beispiel zur Einführung von Äquivalenzrelationen werden gern Zahlenkongruenzen $b \equiv a \pmod{n}$ verwendet. Dabei betrachtet man den Rest r der Ganzzahldivision von a durch n . Offensichtlich können nur die Reste $r = 0, 1, 2, \dots, n-1$ auftreten.

Die betrachtete Äquivalenzrelation zerlegt die Grundmenge, also die ganzen Zahlen, in paarweise disjunkte Teilmengen, sog. Restklassen. Die Restklasse $a \pmod{n}$ ist eine Menge, die aus genau den ganzen Zahlen besteht, die bei Division durch n den gleichen Rest lassen wie a . Jede ganze Zahl befindet sich in genau einer Restklasse. Da n verschiedene Reste auftreten können, gibt es genau n Restklassen.

Es ist üblich, sie mit dem jeweiligen Rest r zu benennen, z.B. $[2]_3$. Man wählt also jeweils einen geeigneten Repräsentanten als Restklassennamen $[a]_n := \{b \mid b \equiv a \pmod{n}\}$.

Das folgende Beispiel 3.12 dient der Illustration. Der Anspruch, die angegebenen Äquivalenzklassen für eine beliebige Sprache selbst finden zu können, besteht nicht. Es bedarf schon einiger Übung bzw. der Nutzung von Ergebnissen, die auf anderem Weg gefunden wurden, um diese Klassen treffsicher zu bestimmen. Sicher ist jedoch, dass es sie gibt!

Beispiel 3.12



Wir betrachten die Menge der Wörter über $\{0, 1\}$, deren vorletztes Zeichen 0 ist: $L = \{w \mid w = u0v \text{ und } u \in \{0, 1\}^* \text{ und } v \in \{0, 1\}\} = \{00, 01, 000, 100, 001, 101, 1000, \dots\}$. Dann wird Σ^* durch R_L mit (wie in Satz 3.4 angegeben) xR_Ly genau dann, wenn für alle Zeichenketten $z \in \Sigma^*$ gilt: $xz \in L \Leftrightarrow yz \in L$, in folgende vier Äquivalenzklassen zerlegt:

$[11]_{R_L} = \{\epsilon, 1, 11, 011, 111, \dots\}$, denn x und y liegen beide außerhalb von L , sodass jeder Suffix z die L -Mitgliedschaft beider Wörter xz und yz übereinstimmend festlegt. Wählt man $z = \epsilon$, so liegen xz und yz (also jeweils beide) nicht in L .

$[01]_{R_L} = \{01, 001, 101, 0001, \dots\}$, denn x und y liegen beide in L . Auch hier bestimmt der jeweils ausgewählte Suffix z die L -Mitgliedschaft für xz und yz übereinstimmend: Wählt man $z = \epsilon$, so liegen xz und yz (also jeweils beide) in L .

Aufgrund der erläuterten Zugehörigkeiten der Wörter in $[11]_{R_L}$ und $[01]_{R_L}$ für $z = \epsilon$ ist sofort klar, dass $[11]_{R_L}$ und $[01]_{R_L}$ disjunkte Mengen sind.

Nun betrachten wir die Wörter der Äquivalenzklasse $[10]_{R_L} = \{0, 10, 110, 0010, \dots\}$. Sie bringen das Potenzial mit, durch Anhängen eines beliebigen einzeichenigen Wortes, d.h. $z = 0$ oder 1 , zu L zu gehören. Das ist weder bei $[11]_{R_L}$ noch bei $[01]_{R_L}$ der Fall. Für $z = \epsilon$ ergeben sich stets Wörter, die nicht zu L gehören.

$[00]_{R_L} = \{00, 000, 100, 0000, \dots\}$ fasst die Wörter zusammen, die zu L gehören ($z = \epsilon$) und zusätzlich auf 0 enden, sodass durch Anhängen eines einzeichenigen Wortes, d.h. $z = 0$ oder 1 , je ein Wort aus L entsteht.

Die gewählten Repräsentanten der Äquivalenzklassen, d.h. die Zeichenketten in den eckigen Klammern, entsprechen den beiden Zeichen, auf die Wörter der betrachteten Klasse enden. An den Wörtern $\varepsilon, 1 \in [11]_{R_L}$ und $0 \in [10]_{R_L}$ wird deutlich, dass dies nicht dogmatisch gilt.

Abbildung 3.20 zeigt links die Zerlegung von Σ^* durch R_L für dieses Beispiel. Auf die Zerlegung von Σ^* durch R_M (rechts dargestellt) gehen wir weiter unten ein.

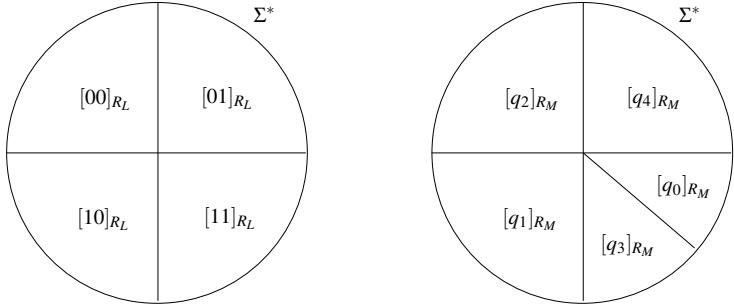


Abbildung 3.20: $Index(R_L) = 4$ (links) $Index(R_M) = 5$ (rechts)

Da Σ^* eine unendliche Menge ist, kann es im Extremfall unendlich viele Äquivalenzklassen geben, d.h. der Index ist unendlich. Es gilt also entweder

$$\begin{aligned}\Sigma^* &= [w_1]_{R_L} \cup [w_2]_{R_L} \cup \dots \cup [w_n]_{R_L} \text{ oder} \\ \Sigma^* &= [w_1]_{R_L} \cup [w_2]_{R_L} \cup \dots,\end{aligned}$$

wobei $[w_i]_{R_L} = \{w \mid wR_Lw_i\}$.

Beweis, Teil 1

Es ist zu zeigen: Wenn L eine reguläre Sprache ist, dann ist der Index von R_L endlich.

Für jede reguläre Sprache L gibt es einen DEA M mit $L = L(M)$. Die in Satz 3.4 definierte Relation R_L lässt sich leicht in die Welt der endlichen Automaten übertragen, s. Abbildung 3.21.

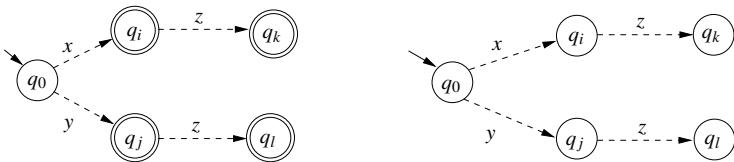


Abbildung 3.21: links: $xz, yz \in L$; rechts: $xz, yz \notin L$ für alle Wörter $z \in \Sigma^*$

Die in Abbildung 3.21 verwendeten (gestrichelten) Pfeile illustrieren die Anwendung der *erweiterten* Überführungsfunktion $\hat{\delta}$, z.B. $\hat{\delta}(q_0, x) = q_i$. An diesen Pfeilen stehen Wörter (hier x , y , und z) statt einzelne Zeichen, wie bei δ .

In Abbildung 3.22 betrachten wir den Spezialfall, der den DEA aus dem Startzustand q_0 sowohl mit x als auch mit y in ein und denselben Zustand q_i übergehen lässt: $\hat{\delta}(q_0, x) =$

Abbildung 3.22: Spezialfall $q_i = q_j$ gegenüber Abb. 3.21

$\hat{\delta}(q_0, y) = q_i$ und $\hat{\delta}(q_i, z) = q_k$ für alle Wörter $z \in \Sigma^*$. Es ist dann vollkommen klar, dass ein beliebiges Wort $z \in \Sigma^*$ zu einem gemeinsamen Zustand q_k führt. q_i und q_k sind entweder beide Endzustände (links in Abbildung 3.22) oder sie sind es *beide* nicht (rechts in Abbildung 3.22).

Da die auf diese Weise gewonnene zweistellige Relation R_M in Σ^* im Folgenden eine wichtige Rolle spielt, benötigen wir deren Definition.

Definition 3.3

Sei $M = (Q, \Sigma, \delta, q_0, E)$ ein DEA, der genau L akzeptiert. Dann sind die Zeichenketten $x, y \in \Sigma^*$ ununterscheidbar durch M , d.h. $xR_M y$, wenn gilt: $\hat{\delta}(q_0, x) = \hat{\delta}(q_0, y) = q_i$ mit $q_i \in Q$.



Übung 3.6

Zeigen Sie, dass R_M eine Äquivalenzrelation (reflexiv, transitiv, symmetrisch) in Σ^* ist.



Da ein DEA endlich viele Zustände besitzt, gilt für $q_k \in Q$, dass der Index von R_M durch die Zustandsanzahl nach oben beschränkt ist, d.h.

$$\text{Index}(R_M) \leq |Q|.$$

Von besonderem Interesse ist der Zusammenhang zwischen den beiden Äquivalenzrelationen R_L und R_M : Für alle $x, y \in \Sigma^*$ gilt $xR_M y \Rightarrow xR_L y$, d.h. wenn $xR_M y$, d.h. $\hat{\delta}(q_0, x) = \hat{\delta}(q_0, y)$, dann $xR_L y$, d.h. $xz \in L \Leftrightarrow yz \in L$ für alle $z \in \Sigma^*$:

$$\begin{aligned} xz \in L &\Leftrightarrow \hat{\delta}(q_0, xz) \in E \\ &\Leftrightarrow \hat{\delta}(\hat{\delta}(q_0, x), z) \in E \\ &\Leftrightarrow \hat{\delta}(\hat{\delta}(q_0, y), z) \in E, \text{ wegen } xR_M y \Leftrightarrow \hat{\delta}(q_0, x) = \hat{\delta}(q_0, y) \\ &\Leftrightarrow \hat{\delta}(q_0, yz) \in E \\ &\Leftrightarrow yz \in L \end{aligned}$$

Analog verfährt man mit $xz, yz \notin L$.

Für jede Äquivalenzklasse von R_M gilt folglich:

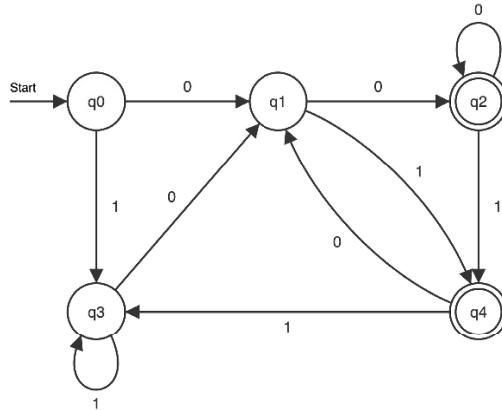
$$\begin{aligned} [x]_{R_M} &= \{y \mid y \in \Sigma^* \text{ und } xR_M y\} \\ &\subseteq \{y \mid y \in \Sigma^* \text{ und } xR_L y\} \text{ wegen } xR_M y \Rightarrow xR_L y \\ &\subseteq [x]_{R_L} \end{aligned}$$

Man sagt, R_M ist eine *Verfeinerung* von R_L . Wie viele Elemente in der jeweiligen Äquivalenzklasse enthalten sind, spielt dabei keine Rolle.

Da der Index von R_M endlich ist, ist wegen $[x]_{R_M} \Rightarrow [x]_{R_L}$ für alle $x \in \Sigma^*$ auch der Index von R_L endlich.

Damit ist der erste Teil des Satzes 3.4 bewiesen: Wenn L regulär ist, so ist der Index von R_L endlich. \square

Mit dem DEA $M = (\{q_0, q_1, q_2, q_3, q_4\}, \{0, 1\}, \delta, q_0, \{q_2, q_4\})$ und folgendem δ



greifen wir hier noch einmal Beispiel 3.12 auf. M akzeptiert genau die Menge aller Wörter über $\{0, 1\}$, deren vorletztes Zeichen 0 ist.

Didaktischer Hinweis 3.9

M ist nicht der Minimalautomat für L .

Wir ermitteln die Äquivalenzklassen für R_M , indem wir *alle* Wörter in eine Klasse aufnehmen, die M von q_0 aus in einen bestimmten Zustand q_k überführen.

Übung 3.7

Schreiben Sie die ersten 15 Wörter aus Σ^* in längenlexikografischer Reihenfolge auf und notieren Sie den nach Verarbeitung erreichten Zustand. FLACI kann Sie unterstützen.

Es ist sinnvoll, die Äquivalenzklassen nach diesen Zuständen zu benennen:

$$\begin{aligned}[q_0]_{R_M} &= \{\epsilon\} \subset [11]_{R_L} \\ [q_1]_{R_M} &= \{0, 10, 010, 110, \dots\} = [10]_{R_L} \\ [q_2]_{R_M} &= \{00, 000, 100, \dots\} = [00]_{R_L} \\ [q_3]_{R_M} &= \{1, 11, 011, 111, \dots\} \subset [11]_{R_L} \\ [q_4]_{R_M} &= \{01, 001, 101, \dots\} = [01]_{R_L}\end{aligned}$$

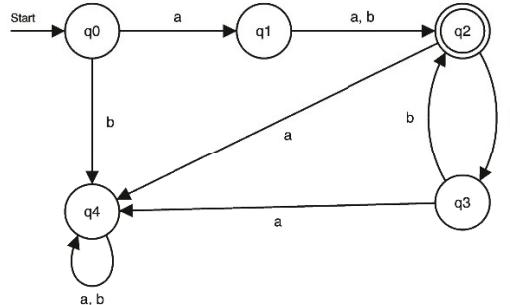
Die Verfeinerung, die mit R_M gegenüber R_L im Beispiel erzielt wird, ist in Abbildung 3.20 rechts dargestellt.

Computerübung 3.20

Verwenden Sie FLACI, um den zugehörigen Minimal-DEA zu erzeugen und bestimmen Sie anschließend die Äquivalenzklassen. Diese stimmen exakt mit den vier Klassen von R_L überein.

**Beispiel 3.13**

$M = (\{q_0, q_1, q_2, q_3, q_4\}, \{a, b\}, \delta, q_0, \{q_2\})$.



Man findet leicht heraus, dass $L(M) = \{w \mid w = aab^{2n} \text{ oder } w = abb^{2n}, n \geq 0\}$.

Die Relation R_M für diesen DEA zerlegt Σ^* in 5 Äquivalenzklassen. In diesem Beispiel kann jeder Zustand vom Anfangszustand q_0 aus erreicht werden. Die Zeichenketten, die von q_0 zu q_i „führen“, notieren wir als Elemente von $[q_i]_{R_M}$ mit $i = 0, 1, 2, 3, 4$.

$$[q_0]_{R_M} = \{\epsilon\}$$

$$[q_1]_{R_M} = \{a\}$$

$$[q_2]_{R_M} = \{w \mid w = aab^r \text{ oder } w = abb^r, r \text{ gerade}\}$$

$$[q_3]_{R_M} = \{w \mid w = aab^r \text{ oder } w = abb^r, r \text{ ungerade}\}$$

$[q_4]_{R_M}$ enthält alle restlichen Wörter.

Die folgenden Überlegungen helfen uns bei der Bestimmung der Äquivalenzklassen von R_L für $L = \{w \mid w = aab^{2n} \text{ oder } w = abb^{2n}, n \geq 0\}$:

- Es gilt aaR_Lab , denn $aaw \in L$ und $abw \in L$ gdw. $w = abb^{2n}, n \geq 0$ für ein beliebiges $w \in \Sigma^*$.
- Es gilt bR_Lba , da $bw \notin L$ und $baw \notin L$ für alle $w \in \Sigma^*$.
- Hingegen gilt $\neg(aR_Lab)$, da z.B. $abb \notin L$ aber $abbb \in L$.

Wir erhalten die folgenden Äquivalenzklassen von R_L , sind uns aber zunächst nicht ganz sicher, ob wir wirklich alle Klassen gefunden haben. Ein Vergleich mit den Äquivalenzklassen von R_M signalisiert Vollständigkeit, d.h.

$$[\epsilon]_{R_L} \cup [a]_{R_L} \cup [aa]_{R_L} \cup [aab]_{R_L} \cup [b]_{R_L} = \Sigma^* \text{ mit}$$

$$[\epsilon]_{R_L} = [q_0]_{R_M} = \{\epsilon\}$$

$$[a]_{R_L} = [q_1]_{R_M} = \{a\}$$

$$[aa]_{R_L} = [q_2]_{R_M} = \{w \mid w = aab^r \text{ oder } w = abb^r, r \text{ gerade}\}$$

$$[aab]_{R_L} = [q_3]_{R_M} = \{w \mid w = aab^r \text{ oder } w = abb^r, r \text{ ungerade}\}$$

$$[b]_{R_L} = [q_4]_{R_M}$$

sodass wir in diesem Beispiel nun sämtliche Äquivalenzklassen gefunden haben.

Wegen $[x]_{R_M} \Rightarrow [x]_{R_L}$ für alle $x \in \Sigma^*$ hätten wir R_L auch einfach aus R_M konstruieren können.

Beweis, Teil 2

Es ist zu zeigen: Wenn der Index von R_L endlich ist, dann ist die Sprache L regulär. Dann gibt es die Zeichenketten x_1, x_2, \dots, x_k , mit

$$\Sigma^* = [x_1]_{R_L} \cup [x_2]_{R_L} \cup \dots \cup [x_k]_{R_L}.$$

Wir konstruieren hieraus den DEA $M = (Q, \Sigma, \delta, q_0, E)$, mit

$$\begin{aligned} Q &= \{[x_1]_{R_L}, [x_2]_{R_L}, \dots, [x_k]_{R_L}\}, \\ \delta([x_i]_{R_L}, a) &= [x_i a]_{R_L}, \text{ mit } a \in \Sigma, \\ q_0 &= [\epsilon] \text{ und} \\ E &= \{[x_i]_{R_L} \mid x_i \in L\}, \end{aligned}$$

wobei $[x_i]_{R_L}$ für die entsprechenden Zustandsnamen stehen. Nun zeigen wir, dass ein Wort w genau dann zu L gehört, wenn es von einem so konstruierten DEA M akzeptiert wird.

$$\begin{aligned} w \in L(M) &\Leftrightarrow \hat{\delta}(q_0, w) \in E \\ &\Leftrightarrow \hat{\delta}([\epsilon]_{R_L}, w) \in E \\ &\Leftrightarrow [w]_{R_L} \in E \\ &\Leftrightarrow w \in L \end{aligned}$$

Da DEA reguläre Sprachen definieren, ist L regulär. \square

Regularität einer Sprache

Wir verwenden nun den Satz von MYHILL und NERODE, um zu zeigen, dass eine gegebene Sprache regulär bzw. nicht-regulär ist.

Beispiel 3.14

Zu zeigen: Die Sprache $L = \{a^n b^n \mid n \geq 1\}$ ist nicht-regulär.



Man erkennt, dass die Zeichenketten $a^k b$, mit $k \geq 1$, (paarweise) unterschiedlichen Äquivalenzklassen angehören müssen. Für ein beliebig ausgewähltes $r \geq 1$ gibt es nur eine einzige Zeichenkette, nämlich $z = b^{r-1}$, sodass $a^r b z \in L$. Für alle anderen Zeichenketten $a^k b$, mit $k \neq r$, gilt $a^k b z \notin L$. Folglich gibt es unendlich viele Äquivalenzklassen von R_L :

$$\begin{aligned} [ab]_{R_L} &= L \\ [a^2 b]_{R_L} &= \{a^2 b, a^3 b^2, a^4 b^3, \dots\} \\ [a^3 b]_{R_L} &= \{a^3 b, a^4 b^2, a^5 b^3, \dots\} \\ &\vdots \quad \vdots \\ [a^k b]_{R_L} &= \{a^{k+i-1} b^i \mid i \geq 1\} \\ &\vdots \quad \vdots \end{aligned}$$

L ist nicht-regulär. Eine genaue Betrachtung der Äquivalenzklassenstruktur ist nicht nötig.

Didaktischer Hinweis 3.10

Falsch wäre $[\epsilon]_{R_L}$, $[a]_{R_L} = \{a, a^2 b, a^3 b^2, \dots\}$, $[aa]_{R_L}, \dots$, denn z.B. $a^2 b R_L a \Leftrightarrow a^2 b \underline{a b^2} \notin L$ aber $a \underline{a b^2} \in L$.

**Übung 3.8**

Zeigen Sie mit Hilfe des Satzes von MYHILL und NERODE, dass die Sprache L mit $L = \{w \mid w \in \{0, 1\}^* \text{ und } w \text{ endet mit } 00\}$ regulär ist.

3.8 Minimalautomat

Aus dem Beweis des Satzes von MYHILL und NERODE kann man erkennen, dass der Minimal-DEA für eine reguläre Sprache L gerade der Äquivalenzklassenautomat ist. Bis auf Umbenennung der Zustände gibt es für jede reguläre Sprache genau einen Minimalautomat.

Das Auffinden des R_L -Äquivalenzklassenautomaten für eine Sprache L kann schwierig sein. Es erfordert schon einige Übung, um keinen Fehler zu machen.

Etwas besser geht es über R_M . Aber dafür müssten wir den Minimal-DEA bereits kennen.

Hier hilft uns ein Verfahren, das aus einem entwickelten DEA für L den Minimal-DEA konstruiert. Klar ist, dass ein DEA nicht minimal sein kann, wenn es zwei Zustände z und z' gibt, sodass für alle $x \in \Sigma^*$ gilt

$$\hat{\delta}(z, x) \in E \Leftrightarrow \hat{\delta}(z', x) \in E.$$

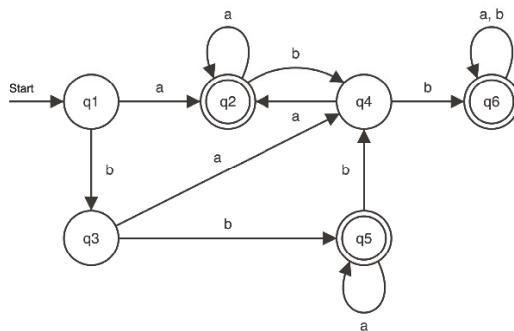
Solche Zustandpaare können zu jeweils einem Zustand verschmelzen.

Das von HOPCROFT und ULLMAN angegebene Verfahren markiert alle Zustandspaare, die nicht zusammengelegt werden können. Es hinterlässt alle verschmelzbaren Zustandspaare. Das Verfahren arbeitet sehr schnell (mit quadratischem Zeitaufwand).

Beispiel 3.15

DEA $M = (\{1, 2, 3, 4, 5, 6\}, \{a, b\}, \delta, 1, \{2, 5, 6\})$.

Beispiel



Die Idee des Verfahrens besteht darin, schrittweise alle die Zustandspaare auszufiltern, die *nicht* verschmelzbar sind. Dafür ist es notwendig, dass alle möglichen Zustandspaare gebildet werden. Genau genommen sind es Zustandszweiermengen, denn die Reihenfolge von z und z' in (z, z') spielt keine Rolle.

Hieraus erklärt sich die Vorbelegung bestimmter Felder in der weiter unten angegebenen Matrix mit einem Ausrufezeichen, was wir als ‚verboten‘ ansehen können. Interessiert man sich für die Belegung eines verbotenen Feldes (z, z') , so muss

man bei (z', z) nachsehen. Zustandspaare, die nicht verschmelzbar sind, werden (mit x) *markiert*. Die am Ende unmarkierten Paare sind verschmelzbar. Für alle Paare (z, z) gilt dies naturgemäß.

Schritt 1 Schritt 1: Wir stellen als erstes sämtliche Zustandspaare in einer Matrix zusammen.

| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 1 | | | | | | |
| 2 | ! | | | | | |
| 3 | ! | ! | | | | |
| 4 | ! | ! | ! | | | |
| 5 | ! | ! | ! | ! | | |
| 6 | ! | ! | ! | ! | ! | ! |

Schritt 2 Schritt 2: Markiere nun alle Paare (z, z') , wenn *genau* einer der beiden Zustände, also entweder z oder z' , ein Endzustand ist.

| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 1 | | x | | | x | x |
| 2 | ! | | x | x | | |
| 3 | ! | ! | | | x | x |
| 4 | ! | ! | ! | | x | x |
| 5 | ! | ! | ! | ! | | |
| 6 | ! | ! | ! | ! | ! | ! |

Schritt 3 Schritt 3: Prüfe für jedes noch unmarkierte Paar (z, z') , ob für wenigstens ein $a \in \Sigma$ gilt: $(\delta(z, a), \delta(z', a))$ ist bereits markiert. Wenn ja, dann markiere (z, z') .

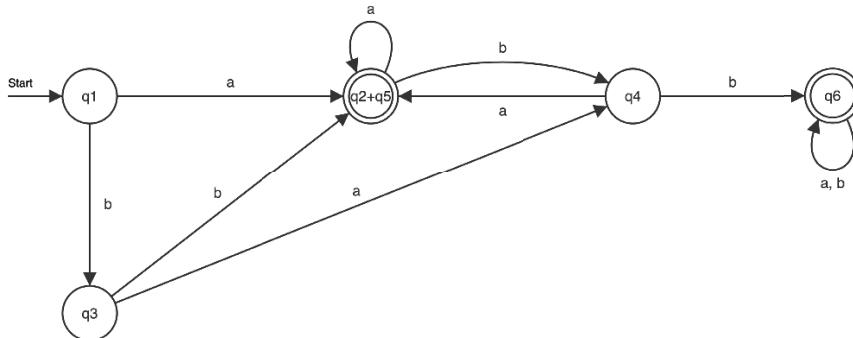
Beispiel für das Paar $(1, 4)$: $\delta(1, a) = 2$, $\delta(4, a) = 2$ Das Paar $(2, 2)$ ist unmarkiert. Also müssen noch $\delta(1, b) = 3$, $\delta(4, b) = 6$ gebildet werden. Das Paar $(3, 6) = (6, 3)$ ist bereits markiert. Folglich muss das Paar $(1, 4)$ markiert werden.

Schritt 4 Schritt 4: Wiederhole Schritt 3 solange, bis sich keine Veränderung der Matrixeinträge ergibt.

| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 1 | | x | x | x | x | x |
| 2 | ! | | x | x | | x |
| 3 | ! | ! | | x | x | x |
| 4 | ! | ! | ! | | x | x |
| 5 | ! | ! | ! | ! | | x |
| 6 | ! | ! | ! | ! | ! | ! |

Schritt 5 Schritt 5: Alle jetzt noch unmarkierten Zustandspaare sind verschmelzbar. Im Bei-

spiel sind dies die Zustände 2 und 5. Dies ergibt folgenden Minimal-DEA.



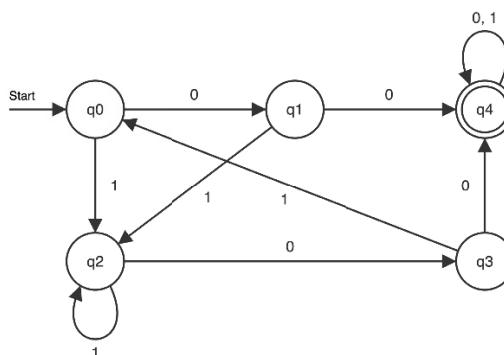
Computerübung 3.21

Verwenden Sie FLACI, um die Lösung des mitgeführten Beispiels zu überprüfen.



Übung 3.9

Der (nicht minimale) DEA $M = (\{q_0, q_1, q_2, q_3, q_4\}, \{0, 1\}, \delta, q_0, \{q_4\})$ mit



akzeptiert genau die Sprache $L(M) = \{x \mid x \in \{0, 1\}^* \text{ und in } x \text{ kommt } 00 \text{ vor.}\}$ Gesucht ist der zugehörige Minimalautomat.

Computerübung 3.22

Verwenden Sie FLACI, um die Lösung von Übungsaufgabe 3.9 zu überprüfen.



3.9 Das Pumping Lemma für reguläre Sprachen

Wir betrachten die Sprache $L = \{b, bab, babab, bababab, \dots\}$ für die wir problemlos eine kontextfreie Grammatik angeben können: $G = (N, T, P, s)$ mit $T = \{a, b\}$, $N = \{X\}$, $s = X$ und $P = \{X \rightarrow b \mid baX\}$.

$L = L(G)$ ist also eine kontextfreie Sprache. Aber ist sie auch regulär? Ja, denn in diesem einfachen Beispiel gelingt es auch, eine reguläre Grammatik für L anzugeben: $G = (N, T, P, s)$ mit $T = \{a, b\}$, $N = \{X, Y\}$, $s = X$ und $P = \{X \rightarrow b \mid bY, Y \rightarrow aX\}$.

Leider sieht man einer beliebigen kontextfreien Sprache im Allgemeinen nicht an, ob sie regulär ist, d.h. ob sie mit einer regulären Grammatik, einem DEA oder einem NEA (mit und ohne ϵ -Übergängen) definiert werden kann. Von daher wäre ein dementsprechendes Entscheidungswerkzeug wünschenswert.

Wenn man sich darunter ein Verfahren vorstellt, das für jede Sprache L entweder mit „Ja“ antwortet, wenn L regulär ist, und anderenfalls mit „Nein“, wird man enttäuscht sein. Ein solches Verfahren gibt es leider nicht. (Auch auf dem Satz von NERODE und MYHILL lässt sich kein praktikables Verfahren aufbauen, das in jedem Falle zu einer Entscheidung führt.)

Pumping Lemma
uvw-Theorem

Im folgenden betrachten wir das sog. *Pumping Lemma für reguläre Sprachen*. Es ist das diesbezüglich leistungsfähigste bekannte Entscheidungswerkzeug und auch unter den Namen *Iterationsatz*, Schleifenlemma oder *uvw-Theorem* bekannt.



Didaktischer Hinweis 3.11

Anstelle einen Satz an den Anfang zu stellen und dessen Beweis anzuschließen, geben wir eine Herleitung an. Diese mündet in die Formulierung des Pumping Lemmas für reguläre Sprachen und liefert rückblickend den Beweis. Die Motivation für das Pumping Lemma wäre viel geringer, wenn wir in traditionelle Satz-Beweis-Form vorgehen würden.

Herleitung
des Satzes

Herleitung: Wir dürfen davon ausgehen, dass es für jede reguläre Sprache L (mindestens) einen DEA gibt, der genau L akzeptiert. Ein solcher Automat hat endlich viele, sagen wir n , mit $n > 0$, Zustände. Wir betrachten ein Eingabewort $z = a_1a_2a_3 \dots a_n \dots a_m$ der Länge $m \geq n$, wobei keineswegs gefordert ist, dass alle in z vorkommenden Alphabetzeichen a_i paarweise verschieden sind.

Der DEA startet in q_0 und berechnet als ersten Folgezustand $\delta(q_0, a_1) = q_1$. Im zweiten Schritt geht er in den Zustand $\delta(q_1, a_2) = q_2$ über usw.¹⁶ Schließlich befindet sich der Automat in Zustand q_{n-1} . Dann liest er a_n und nimmt Zustand q_n ein. Bis dahin lautet die zugehörige Zustandsfolge $(q_0, q_1, q_2, \dots, q_n)$. Danach folgt noch das Restwort $a_{n+1}a_{n+2} \dots a_m$.

Da der Automat – nach Voraussetzung – genau n Zustände besitzt, die Folge $(q_0, q_1, q_2, \dots, q_n)$ jedoch $n+1$ Zustände umfasst, muss es in $(q_0, q_1, q_2, \dots, q_n)$ mindestens einen Zustand geben, der mehrfach vorkommt. Bezieht man das Restwort ein, so gilt das ebenso. Es reicht aber völlig aus, den Wortanfang der Länge n zu betrachten, denn bei dessen Analyse findet mindestens eine solche „Zustandswiederkehr“ zwangsläufig statt.

Es gibt also (mindestens) ein j und ein k , mit $0 \leq j < k \leq n$, sodass $q_j = q_k$. Dann ergibt sich folgendes Bild.

Der Teil des Wortes, der den schon einmal eingenommenen Zustand q_j wiederbringt, also $a_{j+1} \dots a_k$, wurde in Abbildung 3.23 als Schleife dargestellt. Die Länge

¹⁶O.B.d.A. gehen wir davon aus, dass die Zustände gerade so benannt sind, dass δ in der hier verwendeten Form definiert werden kann.

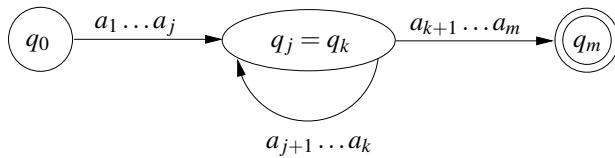


Abbildung 3.23: Beweisskizze für das Pumping Lemma regulärer Sprachen

dieses Teilworts ist mindestens gleich 1, denn $j < k$.

Wenn $q_m \in E$, dann gehört z zu L . Das Teilwort $a_{j+1} \dots a_k$ hat darauf keinen Einfluss. Es spielt keine Rolle, ob der Zustand q_j mehr als einmal vorkommt, wenn er wenigstens einmal eingenommen wird. Folglich kann die Schleife ebenso entfallen wie beliebig oft durchlaufen werden. Dies bedeutet anschaulich, dass das Teilwort $a_{j+1} \dots a_k$ aus z herausgeschnitten oder beliebig oft „hineingepumpt“ werden kann, was wohl zur Namensgebung des Satzes geführt hat. D.h., wenn $z = a_1 a_2 a_3 \dots a_n \dots a_m \in L$, dann gilt für beliebiges $i \geq 0$:

$$\underbrace{a_1 a_2 \dots a_j}_u \underbrace{(a_{j+1} \dots a_k)}_v^i \underbrace{a_{k+1} \dots a_n \dots a_m}_w \in L.$$

Damit ist der folgende Satz 3.5 ausreichend vorbereitet. Bei dessen Formulierung in der Wenn-Dann-Form muss auf die korrekte Angabe der Voraussetzungen geachtet werden.

Satz 3.5

Wenn L eine reguläre Sprache ist, dann existiert eine natürliche Konstante $n > 0$, sodass sich jedes Wort z aus L mit $|z| \geq n$ in der Form $z = uvw$ schreiben lässt und für alle natürlichen $i \geq 0$ gilt $uv^i w \in L$, mit $|v| \geq 1$ und $|uv| \leq n$.



Beweis

s. Herleitung oben

□

Beispiel 3.16

Für eine reguläre Sprache, wie $L = \{a^m \mid m \geq 0\} = \{\epsilon\} \cup \{a^m \mid m > 0\}$, kann man sich nun leicht davon überzeugen, dass es ein n gibt, sodass sich die in Satz 3.5 geforderte Zerlegung für jedes Wort aus $\{a^m \mid m > 0\}$ vornehmen lässt: $n = 1$ leistet das Gewünschte. Jedes Wort $z = a^k$, mit $k \geq n = 1$, lässt sich dann zerlegen in $z = uvw$, mit $u = \epsilon$, $v = a$ und $w = a^{k-1}$. Nun kann man sich leicht davon überzeugen, dass die drei geforderten Bedingungen erfüllt sind: $|v| = |a| = 1 \geq 1$ (OK), $|uv| = 1 \leq n = 1$ (OK), $uv^i w = a^i a^{k-1} = a^{k+i-1} = a^m \in L$ (OK).



In Beispiel 3.16 haben wir das Pumping Lemma für eine gegebene reguläre Sprache in der Wenn-Dann-Richtung angewandt. Das ist in Ordnung aber keineswegs ein Entscheidungswerkzeug für reguläre Sprachen, wie wir es eingangs angestrebt hatten.

Kann man Satz 3.5 einfach umkehren, d.h. aus der Existenz eines n mit den geforderten Eigenschaften (B), folgern, dass die Sprache regulär ist (A)? Nein, das dürfen wir nicht: Mit dem Pumping Lemma gilt $A \Rightarrow B$ und damit ist B eine *nötige, keinesfalls hinreichende* Bedingung für A , wie Beispiel 3.17 exemplarisch unterstreicht.

Beispiel 3.17



Beispiel

Die Sprache $L = \{a^h b^j c^k \mid h = 0 \text{ oder } j = k\}$ ist nicht-regulär, erfüllt aber trotzdem die Forderungen aus dem Pumping Lemma. Für $n = 1$ gibt es für die Zerlegung $z = uvw$, mit $u = \varepsilon$, jedes Wortes $z \in L$ folgende Fälle:

Fall 1: $v = a$, für $h \neq 0$, $w = a^r b^j c^j$

Fall 2: $v = b$, für $h = 0$, $j \neq 0$, $w = b^r c^k$

Fall 3: $v = c$, für $h = j = 0$, $w = c^r$

Die drei Bedingungen sind erfüllt: $|v| = 1 \geq 1$ (OK), $|uv| = 1 \leq n = 1$ (OK),

$uv^i w = a^i w = a^i a^r b^j c^j = a^s b^j c^j \in L$, also für Fall 1 OK.

$uv^i w = b^i w = b^i b^r c^k = b^q c^k \in L$, also für Fall 2 OK.

$uv^i w = c^i w = c^i c^r = c^p \in L$, also für Fall 3 OK.

Zum Nachweis der Nichtregularität dieser Sprache muss man auf den beschwerlich nutzbaren Satz von Myhill und Nerode (Satz 3.4, s. Seite 78) zurückgreifen.

Aber was kann man mit Satz 3.5 anfangen, wenn das eigentliche Untersuchungsziel (A) als zutreffend vorausgesetzt wird?

Die Aussagenlogik hilft uns weiter. Unter Verwendung der weiter oben vergebenen aussagenlogischen Variablen A und B gilt: $A \Rightarrow B$ ist äquivalent zu $\neg B \Rightarrow \neg A$.

Damit ist klar, dass wir mit dem Pumping Lemma die *Nichtregularität* einer Sprache ($\neg A$) nachweisen können, jedoch nicht ihre Regularität. Leider gibt es nicht-reguläre Sprachen, die sich beim Beweisversuch sehr sträuben. Dann bleibt wiederum nur der Satz von Myhill und Nerode (Satz 3.4), um zu zeigen, dass es für die betrachtete nicht-reguläre Sprache unendlich viele Äquivalenzklassen gibt.

Übung 3.10



Weisen Sie die Korrektheit dieser Folgerung mit den Mitteln der Aussagenlogik nach.

Beweistechnisch gesehen, ergeben sich zwei Möglichkeiten, um die Nichtregularität einer Sprache L mit dem Pumping Lemma nachzuweisen:

- Entweder man zeigt (indirekt) $\neg B$, d.h., dass es ein solches n nicht gibt, woraus sich dann mittels $\neg B \Rightarrow \neg A$ die gewünschte Aussage $\neg A$ ergibt (modus ponens, Abtrennungsregel).
- Oder man konstruiert aus der Annahme, L sei regulär, mittels $A \Rightarrow B$ einen Widerspruch, wodurch das Gegenteil der Annahme ($\neg A$) Gültigkeit erlangt (*indirekter Beweis*, s. Beispiel 3.18).

Die Kernaktivität läuft für diese beiden Beweisstrategien auf das Gleiche hinaus.

Beispiel 3.18



Beispiel

Es ist zu zeigen, dass $L = \{a^k b^k \mid k \geq 1\}$ nicht-regulär ist.

Annahme: L ist regulär. Dann existiert lt. Pumping Lemma ein wie in Satz 3.5 gefordertes

Beweis der
Nichtregularität
einer
Sprache

indirekter
Beweis

n. Wir betrachten das Wort $z = \underbrace{a \dots a}_{n} \underbrace{b \dots b}_{n}$. Es entsteht aus $a^k b^k$ für $k = n$, mit $n \geq 1$,

und gehört daher zu L . Wenn wir das n -lange Anfangsstück von z in der Form uv darstellen, ergibt sich wegen $|uv| \leq n$ zwingend, dass v , wegen $|v| \geq 1$, aus mindestens einem a besteht. Dann gilt für das spezielle Wort $uv^0 w = a^{n-|v|} b^n \notin L$. Damit haben wir den Widerspruch, denn jedes Wort der Form $uv^i w$ sollte zu L gehören. (Wenn wir weitere a 's hineinpumpen oder weglassen, die Anzahl der b 's jedoch nicht verändern, ist klar, dass die so entstehenden Wörter nicht zur Sprache gehören.) Die Annahme ist also falsch. L ist folglich nicht-regulär.

Didaktischer Hinweis 3.12

Im Beweis muss von einem beliebigen und damit unbekannten n ausgegangen werden. Es wäre absolut falsch, einfach ein festes n , z.B. $n = 4$, zu wählen und damit die Zerlegung von z vorzunehmen. Dieses fehlerhafte Vorgehen ist leider oft zu beobachten.



Übung 3.11

Gegeben sei die Sprache $L = \{0^2 \mid i > 0\}$. Welche Sprache wird mit L beschrieben? Zeigen Sie mit dem Pumping Lemma, dass L nicht-regulär ist.



Übung 3.12

Entscheiden (und begründen) Sie, welche der folgenden Sprachen regulär sind und welche nicht: $L_1 = \{a^i b^{2i} \mid i \geq 1\}$, $L_2 = \{(ab)^i \mid i \geq 1\}$, $L_3 = \{a^{2n} \mid n \geq 1\}$, $L_4 = \{a^{2^n} \mid n \geq 0\}$, $L_5 = \{a^n b^m a^{n+m} \mid n, m \geq 1\}$.



3.10 NEA mit ε -Übergängen

DEA-Definitionen verlangen die Angabe einer totalen Überführungsfunktion. Dies ist mitunter sehr aufwendig, denn alle „verbotenen“ Eingaben müssen vollständig aufgeführt werden. Ein äquivalenter NEA kommt meist mit weniger Zuständen aus; der zugehörige Zustandsgraph ist dann übersichtlicher. Im Folgenden werden wir sehen, dass man die Überführungsfunktion noch weiter verschlanken kann.

Beispiel 3.19

Wir geben zuerst einen DEA für die Sprache der ganzen Zahlen ohne Vornullen an: $M = (\{q_0, q_1, q_2, q_3, q_4\}, \{+, -, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}, \delta, q_0, \{q_1, q_4\})$ an, δ s. Abbildung 3.24.



Der Akzeptor $M = (\{q_0, q_1, q_3, q_2\}, \{+, -, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}, \delta, q_0, \{q_1, q_3\})$ mit der in Abbildung 3.25 gezeigten Überführungsfunktion ist ein entsprechender NEA für die gleiche Sprache. Er kommt mit nur vier Zuständen aus.

In diesem NEA gibt es für die Zeichen $1, \dots, 9$ Übergänge sowohl von q_0 nach q_3 als auch von q_2 nach q_3 . Durch Einführung eines sog. ε -Übergangs von q_0 zu q_2 können diese Übergänge zusammengefasst werden. Man spricht von einem *spontanen* Übergang. Das bedeutet, dass der Zustandswechsel ohne Verbrauch des nächsten Zeichens des zu analysierenden Wortes erfolgt.

spontaner
Übergang

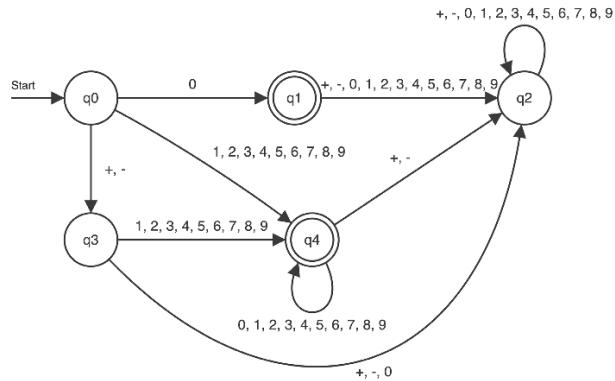


Abbildung 3.24: DEA für die Sprache der ganzen Zahlen ohne Vornullen

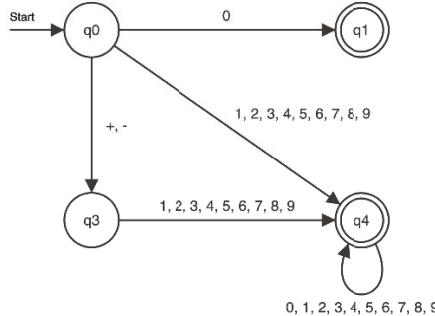
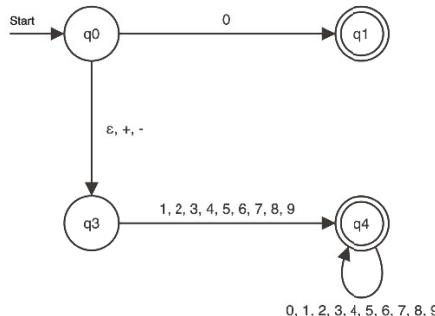


Abbildung 3.25: NEA für die Sprache der ganzen Zahlen ohne Vornullen

Wir erhalten $M = (\{q_0, q_1, q_3, q_2\}, \{+, -, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}, \delta, q_0, \{q_1, q_3\})$ mit der in Abbildung 3.26 gezeigten Überführungsfunktion.

Abbildung 3.26: NEA $_{\epsilon}$ für die Sprache der ganzen Zahlen ohne Vornullen

Der so entstandene Automat hat zwar ebenso viele Zustände wie obiger NEA ohne ε -Übergänge, aber er ist noch besser lesbar:

1. 0 ist eine ganze Zahl.
2. Eine vorzeichenbehaftete Ziffernfolge ohne führender Null ist eine ganze Zahl.
3. Eine vorzeichenlose Ziffernfolge ohne führender Null ist eine ganze Zahl.

Computerübung 3.23

Verwenden Sie FLACI, um die in Beispiel 3.19 beschriebenen Transformationen nachzuvollziehen.



Der Gebrauch von NEA mit ε -Übergängen, kurz: NEA_ε , zur Beschreibung regulärer Sprachen ist recht bequem. Momentan haben wir allerdings noch keine Berechtigung, NEA_ε s zu verwenden. Zuvor muss die Äquivalenz von NEA_ε und NEA nachgewiesen werden.

Äquivalenz von NEA_ε und NEA

Satz 3.6

Zu jedem NEA ohne ε -Übergänge gibt es einen äquivalenten NEA_ε und umgekehrt.



Beweis, Teil 1

Es ist zu zeigen, dass jedem NEA ohne ε -Übergänge ein äquivalenter NEA_ε zuordenbar ist. Dies ist trivial: Hierfür erweitert man δ_{NEA} um Übergänge der Form $\delta(q_i, \varepsilon) \ni q_i$. \square

Beweis, Teil 2

Für die Gegenrichtung ist zu zeigen, dass zu jedem beliebigen NEA_ε ein äquivalenter NEA ohne ε -Übergänge angegeben werden kann. Dieser Beweisteil ist jedoch nicht so offensichtlich, sodass wir uns im folgenden Text ausführlicher damit beschäftigen. Das Ende dieses Beweisteils ist entsprechend markiert.

Für die Überführungsfunktion eines NEA_ε gilt: $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow \wp(Q)$.

Um ε -Übergänge zu eliminieren, müssen für jeden Zustand sämtliche Folgezustände, die *ohne* Eingabezeichenverbrauch erreichbar sind, ermittelt werden. Man nennt dies die ε -Hülle des Zustands q .

ε -Hülle(q)

Definition 3.4

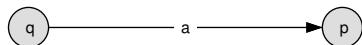
Die ε -Hülle eines Zustands q ist wie folgt definiert: ε -Hülle(q) = { $p \in Q \mid p$ ist von q erreichbar ohne das aktuelle Eingabezeichen zu verbrauchen.}



Lt. Definition 3.4 gilt ε -Hülle(q) $\ni q$. Das ist sehr sinnvoll, denn jeder Zustand ist natürlich von sich selbst aus ohne Eingabezeichenverbrauch erreichbar. Damit ist klar, dass die ε -Hülle(q) mindestens ein Element und höchstens $|Q|$ Elemente enthält.

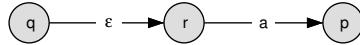
Natürlich kann es auch mehrere „hintereinander geschaltete“ ε -Übergänge geben. Ein Zustand p ist von q mit dem aktuellen Eingabezeichen a erreichbar, wenn

1. $\delta(q, a) \ni p$ („klassischer“ Fall)



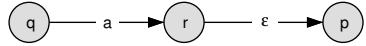
oder wenn es einen oder mehrere Zustände r gibt, mit

2. $\delta(q, \varepsilon) \ni r$, mit $r \neq q$, und $\delta(r, a) \ni p$



oder

3. $\delta(q, a) \ni r$, mit $r \neq q$, und $\delta(r, \varepsilon) \ni p$



Die letzten beiden Fällen besagen, dass der eigentliche „ a -Übergang“ sowohl *nach* als auch *vor* einem spontanen Übergang stattfinden kann. Hierbei müssen alle möglichen „Zwischenzustände“ r betrachtet werden. Im Übrigen kann es auch von r aus weitere spontane Übergänge geben. Die Fälle 2 und 3 sind diesbezüglich eingeschränkt.

ε -Hülle einer Menge von Zuständen



Definition 3.5

Die ε -Hülle einer Menge von Zuständen $\{q_{i_1}, q_{i_2}, \dots, q_{i_m}\}$ ist die Vereinigungsmenge der ε -Hüllen der Einzelzustände, d.h.

$$\begin{aligned} \varepsilon\text{-Hülle}(\emptyset) &= \emptyset \\ \varepsilon\text{-Hülle}(\{q_{i_1}, q_{i_2}, \dots, q_{i_m}\}) &= \bigcup_{k=1}^m \varepsilon\text{-Hülle}(q_{i_k}). \end{aligned}$$

Unter Hinzunahme der folgenden Definition sind wir in der Lage, die oben genannten drei Fälle ohne Einschränkung und kompakt zu beschreiben sowie eine Berechnungsvorschrift für die Überführungsfunktion δ' des zu konstruierenden äquivalenten NEA anzugeben.



Definition 3.6

Für $q \in Q$ und $a \in \Sigma$ definieren wir

$$d(q, a) = \{p \in Q \mid \text{Es gibt einen (direkten) Übergang von } q \text{ nach } p \text{ mit } a\}.$$

Verallgemeinert gilt für eine Zustandsmenge $\{q_{i_1}, q_{i_2}, \dots, q_{i_m}\}$ und $a \in \Sigma$

$$d(\{q_{i_1}, q_{i_2}, \dots, q_{i_m}\}, a) = \bigcup_{k=1}^m d(q_{i_k}, a).$$

Dann haben die folgenden Ausdrücke die jeweils angegebenen Bedeutungen:

$d(\varepsilon\text{-Hülle}(q), a)$ ist die Menge aller von q direkt oder nach vorausgegangenen spontanen Übergängen erreichbarer Zustände. Dies entspricht den Fällen 1 und 2 (verallgemeinert) in obiger Dreierliste.

$\varepsilon\text{-Hülle}(d(q, a))$ ist die Menge aller von q direkt erreichbarer Zustände, inklusive sich anschließender spontaner Übergänge. Dies entspricht gerade dem Fall 3 (verallgemeinert) in obiger Liste.

Dann ergibt sich für die gesuchte Überführungsfunktion δ' des zugehörigen NEA

$$\delta'(q, a) = \underbrace{\varepsilon\text{-H\"ulle}}_{\substack{\varepsilon \text{ vorher} \\ \text{mit } a}} \left(\underbrace{d(\varepsilon\text{-H\"ulle}(q), a)}_{\varepsilon \text{ nachher}} \right).$$

Die restlichen Bestandteile des gesuchten NEA M' sind

$$\begin{aligned} Q' &= Q \\ \Sigma' &= \Sigma \\ q'_0 &= q_0 \\ E' &= E \cup \{q \mid \varepsilon\text{-H\"ulle}(q) \cap E \neq \emptyset\} \end{aligned}$$

Da wir bei der Konstruktion von M' aus M lediglich die spontanen Übergänge eliminiert haben, hat sich am Akzeptanzverhalten des Automaten nichts geändert, womit der geforderte Beweis erbracht wurde. \square

Ende des
Beweisteils 2

Fasst man alles zusammen, so haben wir nachgewiesen, dass es für reguläre Sprachen eine weitere gleichwertige Beschreibungstechnik, nämlich NEA_ε , gibt.

Satz 3.7

NEA u. NEA_ε beschreiben reguläre Sprachen.



Beweis

argumentativ; als Übungsaufgabe \square

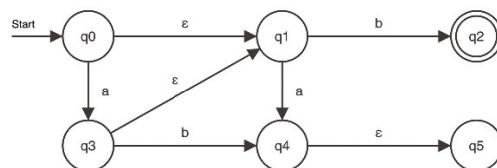
An folgendem Beispiel wird die Berechnung von δ' aus δ vorgeführt.

Beispiel 3.20

Gegeben sei der $\text{NEA}_\varepsilon M = (\{q_0, q_1, q_2, q_3, q_4, q_5\}, \{a, b\}, \delta, q_0, \{q_2\})$, mit



| δ | a | b | ε |
|----------|-----------|-----------|---------------|
| q_0 | $\{q_3\}$ | $\{\}$ | $\{q_1\}$ |
| q_1 | $\{q_4\}$ | $\{q_2\}$ | $\{\}$ |
| q_2 | $\{\}$ | $\{\}$ | $\{\}$ |
| q_3 | $\{\}$ | $\{q_4\}$ | $\{q_1\}$ |
| q_4 | $\{\}$ | $\{\}$ | $\{q_5\}$ |
| q_5 | $\{\}$ | $\{\}$ | $\{\}$ |



Die von M akzeptierte Sprache ist nicht sehr aufregend: $L(M) = \{b, ab\}$. Wir führen die Berechnung von $\delta'(q_0, a)$ vor.

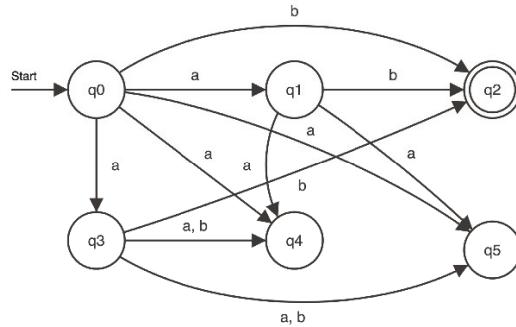
$$\begin{aligned} \delta'(q_0, a) &= \varepsilon\text{-H\"ulle}(d(\varepsilon\text{-H\"ulle}(q_0), a)) \\ &= \varepsilon\text{-H\"ulle}(d(\{q_0, q_1\}, a)) \\ &= \varepsilon\text{-H\"ulle}(\{q_3, q_4\}) \\ &= \{q_1, q_3, q_4, q_5\} \end{aligned}$$

Es empfiehlt sich, die Teilergebnisse in eine Tabelle aufzunehmen, um die Nacheinanderausführung obiger Funktionen angemessen verwalten zu können.

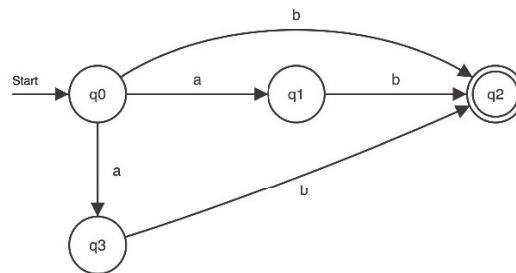
| q | a | ε -Hülle(q) | d (Spalte 3, a) | ε -Hülle(Spalte 4) = $\delta'(q, a)$ |
|-------|-----|-----------------------------|----------------------|--|
| q_0 | a | { q_0, q_1 } | { q_3, q_4 } | { q_1, q_3, q_4, q_5 } |
| q_0 | b | { q_0, q_1 } | { q_2 } | { q_2 } |
| q_1 | a | { q_1 } | { q_4 } | { q_4, q_5 } |
| q_1 | b | { q_1 } | { q_2 } | { q_2 } |
| q_2 | a | { q_2 } | \emptyset | \emptyset |
| q_2 | b | { q_2 } | \emptyset | \emptyset |
| q_3 | a | { q_1, q_3 } | { q_4 } | { q_4, q_5 } |
| q_3 | b | { q_1, q_3 } | { q_2, q_4 } | { q_2, q_4, q_5 } |
| q_4 | a | { q_4, q_5 } | \emptyset | \emptyset |
| q_4 | b | { q_4, q_5 } | \emptyset | \emptyset |
| q_5 | a | { q_5 } | \emptyset | \emptyset |
| q_5 | b | { q_5 } | \emptyset | \emptyset |

$E' = \{q_2\}$ kann man aus der dritten Spalte, also aus den Werten für ε -Hülle(q), direkt ablesen: Die einzige Teilmenge in dieser Spalte, die den Endzustand von M , nämlich q_2 , enthält, ist { q_2 }.

Nun können wir den gesuchten NEA M' ohne ε -Übergänge vollständig angeben: $M' = (\{q_0, q_1, q_2, q_3, q_4, q_5\}, \{a, b\}, \delta, q_0, \{q_2\})$, mit

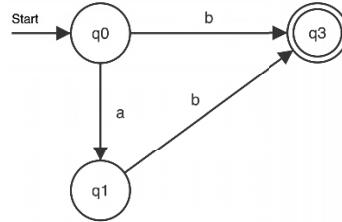


Schaut man etwas genauer auf den Graph der Überführungsfunktion δ' , fallen Zustände auf, von denen keinen Pfeile abgehen. Für den Endzustand q_2 ist das in Ordnung. Die Zustände q_4 und q_5 und mit ihnen alle zuführenden Übergänge können jedoch entfallen. Es entsteht die folgende reduzierte Überführungsfunktion:



Auch diese Funktion kann vereinfacht werden: Offensichtlich gibt es für ab und nur für

dieses Wort zwei gleichartige Wege von q_0 zu q_2 – einen über q_1 und einen über q_3 . Die beiden Wege können zu einem verschmolzen werden. Dies ergibt dann:



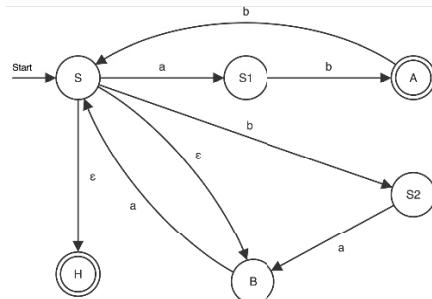
Computerübung 3.24

Verwenden Sie FLACI zur Definition des $\text{NEA}_\epsilon M$ aus Beispiel 3.20. Transformieren Sie M in einen äquivalenten NEA ohne spontane Übergänge M' und vereinfachen Sie ihn. Konvertieren Sie den so entstandenen NEA in einen DEA und minimieren Sie diesen. Erzeugen Sie daraus den finalen NEA und vergleichen Sie das Ergebnis mit dem im Beispiel angegebenen.



Übung 3.13

Konstruieren Sie für den folgenden $\text{NEA}_\epsilon M = (\{S, S_1, A, H, B, S_2\}, \{a, b\}, \delta, S, \{A, H\})$, mit



einen äquivalenten NEA ohne ϵ -Übergänge per Handrechnung.

Computerübung 3.25

Vergleichen Sie Ihr Ergebnis in Übungsaufgabe 3.13 mit der von FLACI durchgeföhrten Konvertierung. Welche(s) der Wörter ab und abb akzeptiert der Automat?



Schlussbetrachtung: Zur Beschreibung regulärer Sprachen sind NEAs mit spontanen Übergängen nicht nur geeignet, sondern für den Entwurf besonders zu empfehlen. Für uns Menschen ist dies das übersichtlichste Entwurfswerkzeug, wenn man bereit ist, das Konzept des Nichtdeterminismus anzunehmen. Ist man an praktischen Umsetzungen, also an lauffähigen Parsern bzw. Scannern, interessiert, sind sie jedoch aus Effizienzgründen die schlechteste Wahl. Hierfür wählt man Minimal-DEAs. Im vorangehenden Text haben wir die Sicherheit geschaffen, jeden beliebigen NEA_ϵ über die Stufen NEA und DEA in einen Minimal-DEA transformieren zu können. Und mit FLACI haben wir auch die reale Möglichkeit, dies bequem zu erledigen.

3.11 Endliche Maschinen

Endliche Automaten haben einen festen Platz in der theoretischen Informatik. Auch dann, wenn man ihre Praxisrelevanz beurteilt, schneiden sie gut ab. Als Akzeptoren für Sprachen können sie zur Modellierung von Prozessen mit sequentiellen Zustandswechseln (Robotersteuerung, Web-Applikationen etc.) ebenso Verwendung finden, wie zur Definition und Analyse zulässiger Eingaben in GUI.

Denkt man an reale Automaten, die als Getränke-, Park- und Fahrkartautomaten unser gesellschaftliches Lebensbild prägen, so gibt es jedoch Aspekte, die mit *Akzeptoren* nicht modelliert werden können: Zu diesen Defiziten gehört die *Ausgabe von Daten* nach einem oder sogar jedem Zustandswechsel. Solche Ausgaben sind nützlich, z.B. zur Zwischeninformation des menschlichen Nutzers über den noch zu zahlenden Restbetrag nach Einwurf einer Münze mit einem bestimmten Wert.

DEA + Ausgabe
= endliche
Maschine

Offensichtlich ist es sinnvoll, DEA um eine Ausgabe zu erweitern. DEA mit Ausgabe nennt man *endliche Maschinen*. Abstrakte Maschinen-Modelle dieser Art sind MEALY¹⁷- und MOORE¹⁸-Maschinen. Wie DEA besitzen sie eine Überführungsfunktion, jedoch keine Endzustandsmenge. Zusätzlich verfügen sie über eine Ausgabefunktion. Obwohl endliche Maschinen (im Unterschied zu DEA) keine Akzeptoren sind, können sie so beschaffen sein, dass die Akzeptanz eines Eingabewortes (z.B. Folge eingeworfener Münzen) an der Gestalt des erzeugten Ausgabewortes festgestellt werden kann: So könnte die Ausgabe beispielsweise mit *zu zahlen: 0 Euro* enden, um anzudeuten, dass das anfangs auf dem Eingabeband stehende „Münz-Wort“ (Getränkepreis) akzeptiert wird.

MOORE-
Maschine

Falls das in jedem Schritt auszugebende Zeichen ausschließlich von dem jeweils aktuellen Zustand der Maschine abhängt, d.h. der zugehörige Übergang keine Rolle spielt, handelt es sich um eine sog. MOORE-Maschine. Beispielsweise können die Zustände einer MOORE-Maschine, die einen Parkautomaten modelliert, den jeweils noch zu zahlenden Restbetrag repräsentieren. Die *Ausgabefunktion* einer MOORE-Maschine nimmt also nur den aktuellen Zustand als Argument.



Definition 3.7

Eine MOORE-Maschine ist ein Sechstupel $M = (Q, \Sigma, \Delta, \delta, \lambda, q_0)$, wobei Q, Σ, Δ und q_0 die von DEAs bekannten Bedeutungen haben. Δ ist das Ausgabealphabet und λ ist eine *totale* Funktion der Form $\lambda : Q \rightarrow \Delta$.

Eine MOORE-Maschine berechnet die Funktion $f : \Sigma^* \mapsto \Delta^*$, wobei $f(x) = y$ genau dann, wenn $(q_0, x, \lambda(q_0)) \xrightarrow{*} (q_i, \epsilon, y)$ und q_i ist irgendein Zustand aus Q .

Beispiel

¹⁷George H. Mealy, 1955

¹⁸Edward Forrest Moore (1925-2003) war einer der Mitbegründer der Automatentheorie und Erfinder des nach ihm benannten MOORE-Automaten (1956). Von 1966 bis 1985 lehrte er als Professor für Mathematik und Informatik an der University of Wisconsin.

Beispiel 3.21

$M_3 = (\{q_0, q_1, q_2\}, \{0, 1\}, \{\text{no, yes}\}, \delta, \lambda, q_0)$ ist eine MOORE-Maschine, die für je ein Eingabewort aus $\{0, 1\}^*$ ein entsprechendes Wort aus $\{\text{no, yes}\}^* \circ \{\text{yes}\}$ ausgibt, wenn das verarbeitete Eingabewort mit zwei gleichen Zeichen endet. Eine Simulation mit FLACI liefert für das Eingabewort 001011 die Ausgabe no no yes no no no yes.

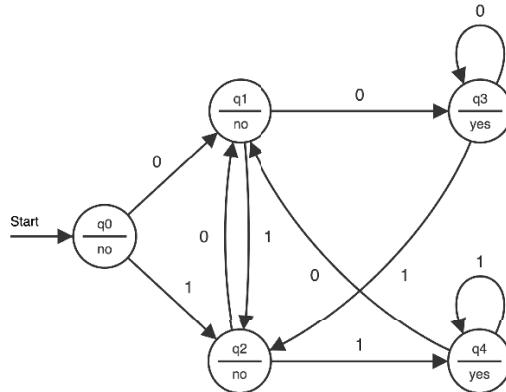


Abbildung 3.27: MOORE-Maschine für $L = \{w \mid w = v00 \text{ oder } w = v11, v \in \{0, 1\}^*\}$

Computerübung 3.26

Geben Sie die vollständige Definition von M_3 an. Notieren Sie für δ und λ die zugehörigen Tabellen. Verwenden Sie FLACI zur Simulation von M_3 .



MEALY-Maschinen unterscheiden sich etwas von MOORE-Maschinen: Die noch zu zahlenden bzw. die Rückgabebeträge (Ausgabe) werden zusammen mit dem nächsten möglichen Eingabe-Münzwert an den Kanten des entsprechenden Übergangsgraphen vermerkt. Jede Kante des Übergangsgraphen einer MEALY-Maschine ist also sowohl mit dem jeweiligen Eingabezeichen als auch dem zugehörigen Ausgabezeichen markiert.

MEALY-Maschine

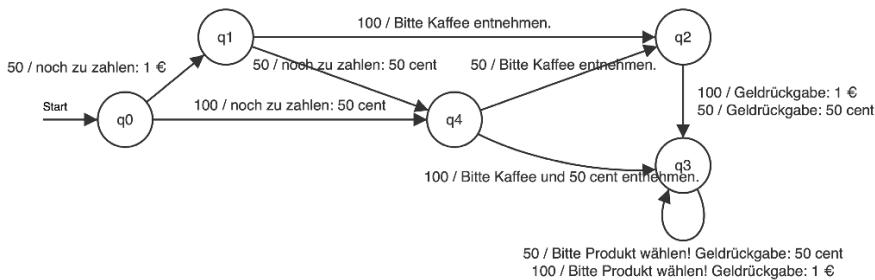


Abbildung 3.28: MEALY-Maschine zur Modellierung eines Kaffeeautomaten

Die MEALY-Maschine, deren Zustandsübergänge in Abbildung 3.28 dargestellt sind, treten in ähnlicher Form bei einem Kaffee-Automaten auf. Eine beim Start

Kaffee-Automat

(in q_0) bereits ausgewählte Tasse Kaffee kostet 1,50 €. Der Automat akzeptiert lediglich 50-cent- und 1-€-Münzen. Nach gestückelter Eingabe des Produktpreises gibt der modellierte Kaffeeautomat eingeworfene Münzen ausnahmslos zurück. Er akkumuliert also kein Guthaben für ggf. weitere Kaffeeprodukte.

Die *Ausgabefunktion* ist hierbei eine zweistellige Funktion, die neben dem aktuellen Zustand ein Zeichen des Eingabealphabets nimmt und das zugehörige Ausgabezeichen zurückgibt.

Im Übergangsgraphen für MEALY-Maschinen werden die *Kanten* in der Form a/b markiert, wobei $a \in \Sigma$ und $b \in \Delta$.

Zum Eingabewort 505050100100 (oder etwas übersichtlicher: 50 50 50 100 100) erzeugt die MEALY-Maschine, deren Überführungsfunktion in Abbildung 3.28 grafisch dargestellt ist, folgende Ausgabe:

noch zu zahlen: 1 €

noch zu zahlen: 50 cent

Bitte Kaffee entnehmen.

Geldrückgabe: 1 €

Bitte Produkt wählen! Geldrückgabe: 1 €

Computerübung 3.27



Verwenden Sie FLACI zum Editieren der Maschine aus Abbildung 3.28 und simulieren Sie die Verarbeitung selbstgewählter Eingabewörter. Geben Sie die jeweils zugehörige Konfigurationenfolge an.

Definition 3.8

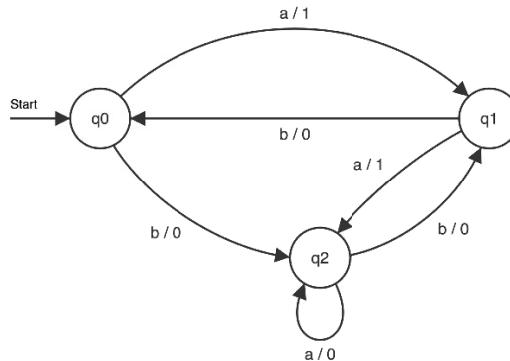
Eine MEALY-Maschine ist ein Sechstupel $M = (Q, \Sigma, \Delta, \delta, \lambda, q_0)$, wobei Q, Σ, δ und q_0 die von DEA bekannten Bedeutungen haben. Δ ist das Ausgabealphabet und λ ist eine *totale* Funktion der Form $\lambda : Q \times \Sigma \rightarrow \Delta$.



- | | |
|--|--|
| Start | Eine endliche Maschine startet mit einem Eingabewort aus Σ^* auf dem Band und stoppt, wenn dieses Wort vollständig eingelesen wurde. Dies geschieht zeichenweise von links nach rechts und wird von entsprechenden Zustandswechseln begleitet. Dieses Verhalten entspricht genau dem eines DEA. |
| Stopp | |
| Ausgabe | Die Ausgabe, die für jedes Eingabewort von einer endlichen Maschine erzeugt wird, ergibt sich durch Verkettung sämtlicher „Einzelfunktionswerte“ von λ vom Start bis zum Stopp. |
| Eine MEALY-Maschine berechnet die Funktion $f : \Sigma^* \mapsto \Delta^*$, wobei $f(x) = y$ genau dann, wenn $(q_0, x, \varepsilon) \xrightarrow{*} (q_i, \varepsilon, y)$ und q_i ist irgendein Zustand aus Q . y ist die von der Maschine produzierte Ausgabe. | |

**Beispiel 3.22**

Die MEALY-Maschine $M_1 = (\{q_0, q_1, q_2\}, \{a, b\}, \{0, 1\}, \delta, \lambda, q_0)$ mit folgendem δ und λ



liefert zur Eingabe von `babaab` das Ausgabewort `000100`.

**Übung 3.14**

Geben Sie die vollständige Definition von M_1 in Beispiel 3.22 an. Notieren Sie δ und λ in Tabellenform.

**Computerübung 3.28**

Erweitern Sie die Definition der in Abbildung 3.28 angegebenen MEALY-Maschine für den Kaffee-Automaten so, dass auch $2-\epsilon$ -Münzen als zulässiges Zahlungsmittel akzeptiert werden.



Mit geeigneten Ausgabefunktionen können MEALY-Maschinen die Akzeptanzanalyse endlicher Automaten simulieren.



Akzeptoren

Beispiel 3.23

Wir betrachten die Sprache $L = \{w \mid w = v00 \text{ oder } w = v11, v \in \{0, 1\}^*\}$. Es ist die Menge aller Wörter, die aus beliebig vielen 0en und 1en bestehen und mit mindestens zwei gleichen Zeichen enden.



Eine entsprechende MEALY-Maschine $M_2 = (\{q_0, q_1, q_2\}, \{0, 1\}, \{\text{no, yes}\}, \delta, \lambda, q_0)$, s. Abbildung 3.29, gibt jeweils ein no-yes-Wort aus, das genau dann mit yes endet, wenn das Eingabewort zu der gegebenen Sprache gehört. Hier wird also die Akzeptanzanalyse eines Automaten durch eine bestimmte Form des Ausgabewortes einer MEALY-Maschine simuliert. Eine Simulation mit FLACI liefert für das Eingabewort `001011` die Ausgabe `no yes no no no yes`. Man kann nachweisen, dass ein DEA für L mindestens fünf Zustände benötigt.

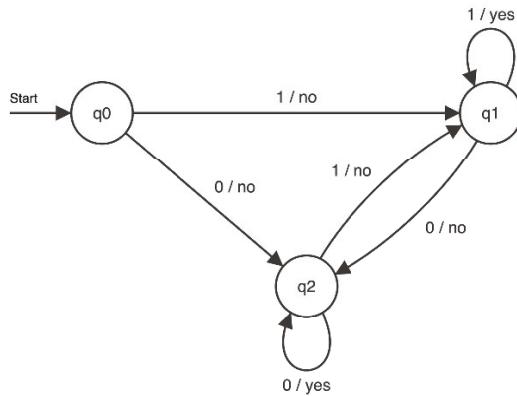
**Computerübung 3.29**

Geben Sie einen DEA für die Sprache $L = \{w \mid w = v00 \text{ oder } w = v11, v \in \{0, 1\}^*\}$ an.



Endliche Maschinen haben die gleiche Leistungsfähigkeit wie endliche Automaten. Außerdem kann man zeigen, dass MEALY- und MOORE-Maschinen äquivalent sind, d.h. zu jeder MOORE-Maschine existiert eine äquivalente MEALY-Maschine und umgekehrt. Außerdem gilt, dass MEALY-Maschinen im Allgemeinen mit weniger Zuständen auskommen als ihre MOORE-Maschinen-Pendants.

endliche
Automaten und
endliche
Maschinen

Abbildung 3.29: MEALY-Maschine für $L = \{w \mid w = v00 \text{ oder } w = v11, v \in \{0,1\}^*\}$

Anwendungsgebiete

Endliche Automaten und endliche Maschinen modellieren sequentielle Zustandsübergänge und eignen sich deshalb zur Beschreibung digitaler Schaltungen. Durch Komposition können sie auch zur *Modellierung paralleler Übergänge* verwendet werden. Dies tritt beispielsweise bei *zellulären Automaten* und *Petri-Netzen* auf. Sinnvolle Anwendungen endlicher Maschinen sind Software- und Systementwicklung, Workflow-Management, elektronischer Handel und Web Services. In Gestalt computergesteuerter Gegner (sog. Bots) kommen sie beispielsweise bei Spielen und KI-Projekten zum Einsatz.



4 Reguläre Ausdrücke (RA)

4.1 Reguläre Mengen

Am 2. Juli 2019 traf es Cloudflare¹: Innerhalb weniger Minuten verlor der Dienstleister 80 Prozent seiner Serverkapazität. Tausende Webseiten waren nicht mehr erreichbar. Dropbox, Cloudbase, Discord, Shopify und Zendesk gehörten zu den betroffenen Cloudflare-Kunden. Cloudflare brauchte Stunden, um die Infrastruktur wieder hochzufahren – alles nur wegen weniger eigentlich korrekter Zeichen. Die CPU-Überlastung wurde von einer einzigen Regel verursacht, die einen schlecht geschriebenen *regulären Ausdruck (RA)* enthielt, der ein enormes *Backtracking* auslöste. Dies ist der RA, der den Ausfall verursacht hat:

```
(?:(?:\"|'|\\]|\\}|\\\\\\d|(:nan|infinity|true|false|null|undefined|symbol|math)|\\'|-|\\+)+[]]*;?  
((?:\\s|-|~|!|{|\\||\\+)*.*(:.*=.*)))
```

Pattern Matching mit RA ist im Allgemeinen sehr schnell, aber man kann spezielle Ausdrücke konstruieren, die das Matching zum Erliegen bringen. Sie sind ein gutes Beispiel dafür, wie die Anwendung guter Theoriekenntnisse zu besseren Programmen führt.

Pattern matching und backtracking haben wir bereits als algorithmische Kerntechnologien der Informatik kennengelernt. Ken *Thompson* und Dennis *Ritchi* haben sich um die Einführung regulärer Ausdrücke und die Entwicklung von Tools, die sie verwenden, verdient gemacht. Ende der 1970er Jahre waren RA eine Schlüsseleigenschaft der UNIX-Landschaft. Dabei spielte es auch eine große Rolle, dass RA zeichenweise über die Tastatur eingetippt werden konnten.

Wir wissen, dass reguläre Sprachen mit *Chomsky*-Typ-3-Grammatiken definiert werden können. Außerdem haben wir gezeigt, dass sich DEA und NEA ebenso dafür eignen. RE stellen eine weitere, sehr kompakte Beschreibungsmöglichkeit für reguläre Sprachen dar.

Wir beginnen mit der Definition *regulärer Mengen*. Die Idee besteht darin, für deren Definition nur ganz wenige *Basismengen* und insgesamt nur drei *Mengenoperationen* zuzulassen. Durch deren wiederholte und kombinierte Anwendung kann

regulären
Ausdruck (RA)
Backtracking

Pattern Matching

Thompson
Ritchi

Chomsky

regulärer
Ausdruck

reguläre
Mengen

¹[https://blog.cloudflare.com/de-de/...
...details-of-the-cloudflare-outage-on-july-2-2019-de-de/](https://blog.cloudflare.com/de-de/...-details-of-the-cloudflare-outage-on-july-2-2019-de-de/)

jede beliebige reguläre Menge (Sprache) erzeugt werden und etwas anderes als reguläre Mengen kann auf diesem Wege nicht entstehen. Die Basismengen sind:

- \emptyset Die leere Menge $\{\}$, die kein einziges Wort enthält, auch nicht das leere.
- $\{\epsilon\}$ Die Menge, die nur das leere Wort $\epsilon = " "$ enthält.
- $\{a\}$ Alle Einermengen, die nur jeweils genau ein einzeichiges Wort, das aus dem zugehörigen Alphabetzeichen hervorgeht, enthalten.

Die so definierten Basismengen sind reguläre Mengen. Durch Anwendung der im Folgenden angegebenen Operationen – und nur durch diese – entstehen weitere reguläre Mengen.

- \cup Vereinigung: $A \cup B = \{w \mid w \in A \text{ oder } w \in B\}$
- \circ Konkatenation: $A \circ B = \{w \mid w = w_1 w_2 \text{ und } w_1 \in A \text{ und } w_2 \in B\}$
- * Kleene-Stern: $A^* = A^0 \cup A^1 \cup \dots \cup A^i \cup \dots$
(Wenn $\epsilon \notin A$, steht A^i für die Menge aller Wörter über A der Länge i .)

Kleene Stephen Cole Kleene² hat reguläre Ausdrücke eingeführt.

Auf die jeweils erzeugten Resultatmengen dürfen diese Operationen erneut angewandt werden, so dass wiederum reguläre Mengen über dem zugrunde liegenden Alphabet entstehen.

Ausschlussregel Reguläre Mengen werden nur auf die angegebene Weise gebildet. Alle anderen Mengen sind nicht regulär.

Die binäre Operation Verkettung (Konkatenation, \circ) wurde zunächst für Zeichen eingeführt und später auf Wörter und schließlich auf Mengen (Sprachen) ausgeweitet. Das Operationszeichen \circ wird vereinbarungsgemäß gern weggelassen.

Beispiel 4.1



Unterscheide Konkatenation und Vereinigung für zwei Mengen:

$$\begin{aligned} \{a\} \circ \{b\} &= \{a\} \{b\} = \{ab\} \text{ aber } \{a\} \cup \{b\} = \{a, b\}, \\ \{a, b\} \circ \{c, d\} &= \{ac, ad, bc, bd\}, \{a, b\} \circ \{\} = \{\} = \emptyset, \{a, b\} \circ \{\epsilon\} = \{a, b\} \end{aligned}$$

Beispiel 4.2



Der KLEENE-Stern³ wurde im Zusammenhang mit dem Begriff der Wortmenge eingeführt: $\{I\}^* = \{\epsilon, I, II, III, \dots\}$, $\{a, b\}^* = \{\epsilon, a, b, ab, ba, aa, bb, aaa, aab, aba, baa, abb, \dots\}$.

Mit Hilfe von regulären Mengen kann man *reguläre Sprachen* definieren. Beispielsweise beschreibt der Ausdruck

$$(((\{a\}((\{a\} \cup \{b\})^*))(\{b\}\{b\}))((\{a\} \cup \{b\})^*))$$

Mustervorstellung die Menge aller Wörter über $\{a, b\}$, die mit a beginnen und zwei b 's unmittelbar nebeneinander enthalten. Das kürzestes Wort, das auf dieses *Muster* (Pattern) passt, ist abb .

Notation Die Schreibweise des obigen Ausdrucks ist sehr unübersichtlich. Neben der

²sprich: klini; geb. 05.01.1909 in Hartford, Connecticut; gest. 25.01.1994 in Madison, Wisconsin

³Dieser Stern-Operator wird *KLEENESche Hülle* oder auch *endlicher Abschluss* genannt.

Häufung an runden Klammern fällt auf, dass eine kompaktere Notation durch das Weglassen der geschweiften Klammern, die die Mengen anzeigen, erzielt werden kann. Hierzu ist es nötig, geeignete Symbole für die jeweiligen Einermengen einzuführen. Diese müssen sich wiederum von den entsprechenden Mengenelementen unterscheiden. Deshalb verwendet man *Fettdruck* (gelegentlich auch Unterstreichungen) und spricht von einem *regulären Ausdruck* (engl.: regular expression, kurz: RegExp, RA oder Regex; deutsch: RA).

reguläre Ausdrücke

Definition 4.1

Reguläre Ausdrücke über einem Alphabet Σ sind wie folgt definiert.



1. \emptyset , ε und a , für jedes $a \in \Sigma$, sind reguläre Ausdrücke über Σ und bezeichnen die Mengen $\emptyset = \{\}$, $\{\varepsilon\}$ bzw. $\{a\}$.
2. Seien u und v reguläre Ausdrücke über Σ und $L(u)$ bzw. $L(v)$ die zugehörigen Sprachen. Dann sind auch
 - a) $(u + v) = L(u) \cup L(v)$,
 - b) $(u \cdot v) = L(u) \circ L(v)$ und
 - c) $(u^*) = L(u)^*$ sowie $(v^*) = L(v)^*$
 reguläre Ausdrücke über Σ .
3. Nur die mit diesen Regeln erzeugten Ausdrücke sind reguläre Ausdrücke über Σ .

Obwohl in obiger Definition nicht ausgewiesen, werden die Notationen $x^+ = xx^*$ und $x^n = \underbrace{x \dots x}_{n-mal}$ zur Abkürzung gerne benutzt.



Beispiel 4.3

Die oben betrachtete Sprache $((\{a\}((\{a\} \cup \{b\})^*))(\{b\}\{b\}))((\{a\} \cup \{b\})^*)$ wird durch den folgenden regulären Ausdruck $((a \cdot (a+b)^*) \cdot (b \cdot b)) \cdot (a+b)^*$ definiert. Es stören lediglich die vielen runden Klammern.

4.2 Klammersparregeln und Äquivalenzen

In manchen Programmiersprachen der LISP-Familie (wie Scheme und Racket) werden runde Klammern verpflichtend verwendet, um die Ausführungsreihenfolge der entsprechenden Operationen explizit festzulegen. In der Sprache Logo gibt es jedoch Klammersparregeln, was die Syntax deutlich „entschlackt“.

Auch die Klammern in RA können vermieden werden, wenn man bereit ist, Prioritätsfestlegungen zu akzeptieren. Die absteigend angegebene Prioritätenfolge lautet: $*$, \cdot , $+$. Da Vereinigung und Konkatenation *assoziative* Operationen sind, dürfen die betreffenden Klammern einfach weggelassen werden.

Wie bei arithmetischen Ausdrücken lässt man auch das Zeichen \cdot sehr gern weg, wenn dadurch keine Verwechslungen möglich sind.

Beispiel 4.4

Der obige Beispielausdruck kann dann noch kompakter als

$$\mathbf{a}(\mathbf{a} + \mathbf{b})^* \mathbf{b} \mathbf{b} (\mathbf{a} + \mathbf{b})^*$$

geschrieben werden. Bezogen auf den Schreibaufwand ist dies die kürzeste Form zur Definition regulärer Sprachen.

Die Wahl der Operationssymbole $*$, \cdot und $+$ für reguläre Ausdrücke suggeriert eine Verwandschaft mit dem Potenzieren, Multiplizieren und Addieren natürlicher Zahlen. Hier gilt die gleiche Prioritätenfolge. Es gibt aber auch Unterschiede, wie wir in Abschnitt 4.2 sehen werden.

So wie es mehrere äquivalente formale Grammatiken und DEA gibt, existieren verschiedene RE zur Definition ein und derselben regulären Sprache. Ähnlich wie bei äquivalenten Termumformungen dürfen folgende Gleichheiten benutzt werden.
Umformungsregeln

u, v und w symbolisieren reguläre Ausdrücke, **ε** für $\{\epsilon\} = \{""\}$ und **Ø** für $\{\} = \emptyset$.

Øu = uØ = Ø – mit Erinnerung an die Multiplikation mit 0, s. Beispiel 4.1.

εu = ue = u – mit Erinnerung an die Multiplikation mit 1.

Ø* = ε, denn $\{\}^* = \{\}^0 \cup \{\}^1 \cup \{\}^2 \cup \dots = \{\epsilon\} \cup \{\} \cup \{\} \cup \dots = \{\epsilon\} = \epsilon$.

ε* = ε, denn $\{\epsilon\}^* = \{\epsilon\}^0 \cup \{\epsilon\}^1 \cup \{\epsilon\}^2 \cup \dots = \{\epsilon\} \cup \{\epsilon\} \cup \{\epsilon\} \cup \dots = \{\epsilon\} = \epsilon$.

u + v = v + u – Erinnerung an die Kommutativität der Addition.

u + Ø = u – Erinnerung an die Addition von 0.

u + u = u – verstößt gegen die Analogie.

(u*)* = u*

u(v+w) = uv + uw – mit Erinnerung an die Distributivität.

(uv)*u = u(vu)*

(u+v)* = (u* + v*)*

Übung 4.1

Beweisen Sie einige der hier angegebenen Gleichungen.

**Übung 4.2**

Gegeben sind $L_1 = \{10, 1\}$ und $L_2 = \{011, 11\}$. Ermitteln Sie $L_1 L_2$ und $\{10, 11\}^*$.

**Übung 4.3**

Geben Sie einen regulären Ausdruck an, der die folgende Sprache beschreibt:
 $L = \{w \in \{a, b\}^* \mid w \text{ beginnt oder endet mit } a\}$.



Übung 4.4

Welche Sprache L wird durch den regulären Ausdruck $c^*(b+ac^*)^*$ über dem Alphabet $\{a, b, c\}$ beschrieben? Zeigen Sie außerdem, dass $acabacc$ und $bbaaacc$ zu L gehören.

Lösung: Jedes Wort über $\{a, b, c\}$, das die Zeichenkette bc nicht enthält.

**Übung 4.5**

Zeigen Sie in $(1+10)^i$ durch vollständige Induktion über i , dass die durch $(1+10)^*$ definierte Sprache keine Wörter mit zwei aufeinander folgenden Nullen enthält.



4.3 Reguläre Ausdrücke und endliche Automaten

Mit dem folgenden *Satz von KLEENE* wollen wir nun zeigen, dass RA genau die gleiche Sprachklasse definieren, wie endliche Automaten, nämlich reguläre Sprachen. Bisher haben wir diesen Zusammenhang nur angenommen, um die Palette der „bequemen“ Beschreibungsmöglichkeiten regulärer Sprachen auszuweiten.

Satz von
KLEENE

Satz 4.1

Eine Sprache L wird von einem DEA M genau dann akzeptiert, wenn L durch einen regulären Ausdruck R über dem zu M gehörenden Eingabealphabet beschrieben werden kann.



Wir haben eine Äquivalenz zu zeigen. Deshalb umfasst der Beweis zwei Teile.

Beweis

1. Teil: Konstruktion von M aus R (*Thompson-Konstruktion*),

Thompson-
Konstruktion

2. Teil: Konstruktion von R aus M ,

wobei für beide Teile $L(M) = L(R) = L$ gelten muss. \square

Bei der in Teil 1 geforderten Konstruktion eines äquivalenten DEA aus einem gegebenen regulären Ausdruck machen wir uns $\mathcal{L}_{NEA_\epsilon} \equiv \mathcal{L}_{DEA}$ zunutze. Es ist nämlich viel einfacher, einen zu R äquivalenten NEA $_\epsilon$ zu konstruieren als einen dementsprechenden DEA. Ein solcher NEA $_\epsilon$ für R besitzt genau einen Endzustand. Dies ist keine Einschränkung gegenüber der allgemeinen NEA-Definition, was wir mit Satz 4.2 nachweisen.

Satz 4.2

Zu jedem NEA $_\epsilon$ M gibt es einen äquivalenten NEA $_\epsilon$ M' mit genau einem Endzustand q_f .

**Beweis**

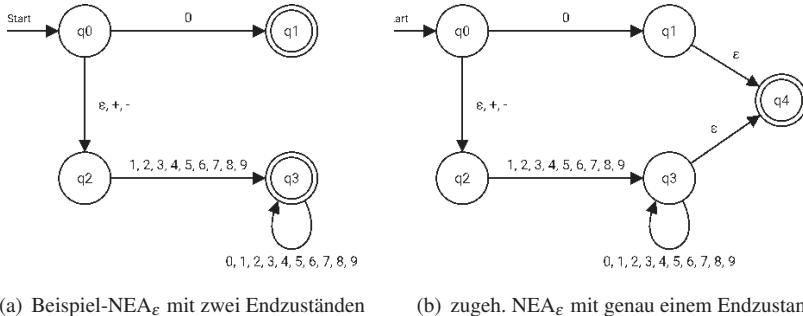
Man konstruiert M' aus M , indem man für alle Endzustände q_i von M spontane Übergänge $\delta'(q_i, \epsilon) = \{q_f\}$ in M' ergänzt. Es gilt $E' = \{q_f\}$. \square

Beispiel 4.5

Beispiel

Gegeben sei der NEA_ε $M = (\{q_0, q_1, q_2, q_3\}, \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, -\}, \delta, q_0, \{q_1, q_3\})$ mit dem in Abbildung 4.1(a) dargestellten δ .

Durch Anwendung des im Beweis formulierten Verfahrens erhält man einen zu M äquivalenten Automaten $M' = (\{q_0, q_1, q_2, q_3\}, \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, -\}, \delta', q_0, \{q_f\})$ mit dem in Abbildung 4.1(b) dargestellten δ' .



(a) Beispiel-NEA_ε mit zwei Endzuständen (b) zugeh. NEA_ε mit genau einem Endzustand

Abbildung 4.1: Transformation eines NEA_ε in einen NEA_ε mit genau einem Endzustand

Beweis, Teil 1

RA \leadsto NEA_ε

Wir beginnen mit dem Beweis von Teil 1 des Satzes 4.1, also der Konvertierung eines regulären Ausdrucks in einen äquivalenten NEA_ε mit genau einem Endzustand. In den folgenden Abbildungen sollen s an Startzustand und f an Finalzustand erinnern.

- 0** Der reguläre Ausdruck **0** beschreibt die leere Menge. Es gibt einen Anfangs- und einen Endzustand, aber es findet kein einziger Übergang statt. Da das Alphabet nach Definition nicht leer sein darf, geben wir hier das Zeichen **a** vor.



$$M = (\{s, f\}, \{a\}, \delta, s, \{f\}), L(M) = \emptyset$$

- e** Für den regulären Ausdruck **ε** entsteht ein Automat, der nur das leere Wort akzeptiert.



$$M = (\{s, f\}, \{a\}, \delta, s, \{f\}), L(M) = \{\epsilon\}$$

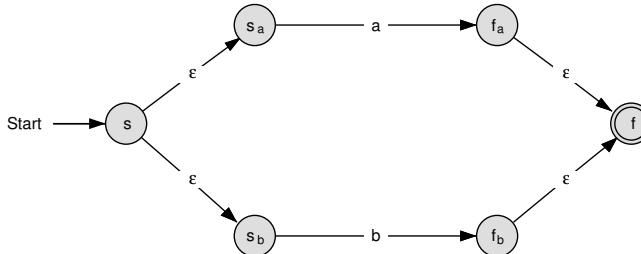
- a** Für alle Alphabetzeichen **a** ergibt sich jeweils der zu **a** gehörende NEA.



$$M = (\{s, f\}, \{a\}, \delta, s, \{f\}), L(M) = \{a\}$$

- a+b** Um den zu **a+b** gehörenden NEA zu erzeugen, brauchen wir zwei neue Zustände s (Anfangszustand) und f (Endzustand des zu konstruierenden NEA_ε). Dann folgen spontane Übergänge von s zu den Anfangszuständen s_a und s_b bzw. von den Endzuständen f_a und f_b

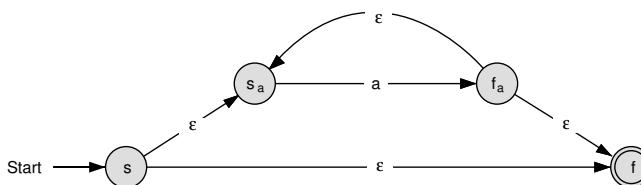
der „parallelgeschalteten“ Automaten zu f .



$$M = (\{s, s_a, s_b, f_a, f_b, f\}, \{\text{a, b}\}, \delta, s, \{f\}), L(M) = \{\text{a, b}\}$$

Nun konstruieren wir einen NEA für a^* .

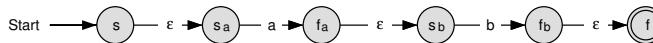
a^*



$$M = (\{s, s_a, f_a, f\}, \{\text{a}\}, \delta, s, \{f\}), L(M) = \{\epsilon, \text{a, aa, aaa, ...}\}$$

Ein NEA für $\text{a} \cdot \text{b} = \text{a b} = \text{ab}$ ergibt sich durch „Verkettung“ der zugehörigen Automaten in der angegebenen Reihenfolge.

$\text{a} \cdot \text{b}$



$$M = (\{s_a, f_a, s_b, f_b\}, \{\text{a, b}\}, \delta, s_a, \{f_b\}), L(M) = \{\text{ab}\}$$

Mit der Angabe dieses allgemeingültigen Verfahrens zur Konstruktion eines äquivalenten NEA $_{\epsilon}$ M aus einem gegebenen RA R ist der erste Teil des Beweises beendet. \square

Beweis, Teil 2

Wir beweisen Teil 2, indem wir aus einem gegebenen DEA⁴

$$M = (\{q_1, q_2, \dots, q_n\}, \Sigma, \delta, q_1, E)$$

einen äquivalenten regulären Ausdruck R konstruieren. Dabei gilt:

$$L(M) = L(R) = \bigcup_{e \text{ mit } q_e \in E} R_n(1, e).$$

Es werden also alle regulären Mengen vereinigt, die M aus dem Startzustand q_1 in einen Endzustand übergehen lassen. In obigem Ausdruck $R_n(1, e)$ werden nur die Zustandsindizes geschrieben, nicht die vollen Zustandsnamen: 1 für q_1 oder z.B. e für q_e , falls $q_e \in E$.

Man kann sich das Verfahren so vorstellen, dass man die Übergänge im Zustandsgraphen schrittweise durch RA ersetzt. Bis auf den Start- und die Endzustände werden die Zustände dadurch obsolet und können entfernt werden (*Zustandseliminierungsverfahren*). Am Ende

⁴Die Umbenennung der Zustände in der geforderten Form ändert nichts an der Sprache, die durch M akzeptiert wird, und kann für jeden DEA problemlos vorgenommen werden.

Zustands-eliminierungs-verfahren

kann der RA für den Übergang vom Start- zu den Endknoten abgelesen werden.

Um dies mathematisch beschreiben zu können, definieren wir Mengen

$$R_k(i, j) = \{w \mid w \in \Sigma^* \text{ und } \hat{\delta}(q_i, w) = q_j, \\ \text{wobei für alle eingenommenen Zwischenzustände } q_z \text{ gilt: } z \leq k\}$$

für $k = 0, 1, \dots, n - 1$ und $i, j = 1, 2, \dots, n$. q_k ist also der Index-größte Zwischenzustand, falls er auf dem „Weg“ von q_i zu q_j tatsächlich vorkommt.

$R_k(i, j)$ fasst alle Wörter aus Σ^* zusammen, die M aus q_i in q_j überführen. Dabei sind nur Zwischenzustände erlaubt, deren Indices nicht größer als k sind.

$k = 0$ bedeutet, dass es keine solchen Zwischenzustände gibt, also Direktübergang von q_i zu q_j . Abbildung 4.2 zeigt die beiden Möglichkeiten.

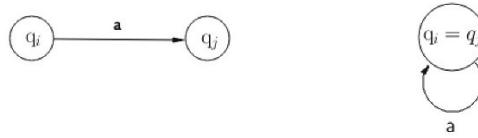


Abbildung 4.2: Direktübergänge von q_i zu q_j mit a

Dann gilt

$$R_0(i, j) = \begin{cases} \{"a" \mid \delta(q_i, a) = q_j\}, & \text{wenn } i \neq j; \\ \{"a" \mid \delta(q_i, a) = q_j\} \cup \{\varepsilon\}, & \text{wenn } i = j. \end{cases}$$

Abbildung 4.3 hilft uns, eine sinnvolle Definition für $R_{k+1}(i, j)$ zu finden.

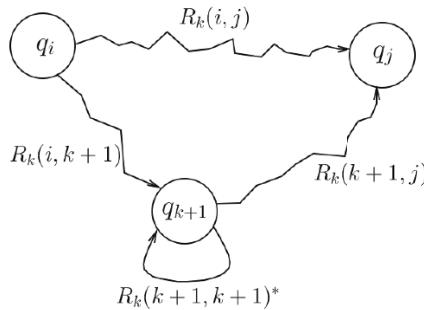


Abbildung 4.3: Konstruktion eines äquivalenten RA aus einem DEA

Es gibt praktisch zwei verschiedene „Wege“ von q_i zu q_j , nämlich der unter potenzieller Verwendung von Zwischenzuständen q_z , mit $z \leq k$, oder der Weg, der q_{k+1} einbezieht. Außerdem kann es noch q_{k+1} -Schleifen geben, also Wege, die beliebig oft von q_{k+1} wieder

zu q_{k+1} führen. Folgender Ausdruck fasst das zusammen:

$$R_{k+1}(i, j) = R_k(i, j) \cup R_k(i, k+1) \circ R_k(k+1, k+1)^* \circ R_k(k+1, j).$$

Mit dem oben angegebenen Ausdruck für R_0 ist damit die rekursive Definition von R_k für $k = 0, 1, 2, \dots, n-1$ komplett.

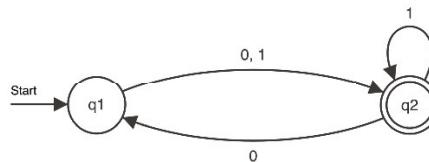
Es spielt im Grunde keine Rolle, ob man die so definierten Mengen als reguläre Ausdrücke notiert oder die mathematische Mengenschreibweise verwendet und nur die gesuchte Menge als RA angibt. Wie weiter oben bereits ausgeführt ist der zu ermittelnde RA

$$R = \bigcup_{e \text{ mit } q_e \in E} R_n(1, e).$$

□

Beispiel 4.6

Zur Demonstration der im Beweis (Teil 2) beschriebenen DEA-Transformation verwenden wir einen DEA $M = (\{q_1, q_2\}, \{0, 1\}, \delta, q_1, \{q_2\})$ mit nur zwei⁵ Zuständen und der folgenden Überführungsfunktion.



Gesucht ist $R = R_2(1, 2)$. Wir führen die rekursive Berechnung vollständig vor:

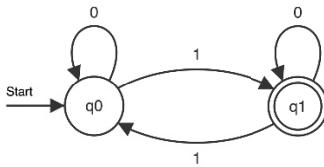
$$\begin{aligned} R_2(1, 2) &= R_1(1, 2) \cup R_1(1, 2) \circ R_1(2, 2)^* \circ R_1(2, 2) \\ R_1(1, 2) &= R_0(1, 2) \cup R_0(1, 1) \circ R_0(1, 1)^* \circ R_0(1, 2) \\ R_0(1, 2) &= \{0, 1\} \\ R_0(1, 1) &= \{\epsilon\} \\ R_1(1, 2) &= \{0, 1\} \cup \{\epsilon\} \circ \{\epsilon\}^* \circ \{0, 1\} = \{0, 1\} \\ R_1(2, 2) &= R_0(2, 2) \cup R_0(2, 1) \circ R_0(1, 1)^* \circ R_0(1, 2) \\ R_0(2, 2) &= \{\epsilon, 1\} \\ R_0(2, 1) &= \{0\} \\ R_1(2, 2) &= \{\epsilon, 1\} \cup \{0\} \circ \{\epsilon\}^* \circ \{0, 1\} = \{\epsilon, 1, 00, 01\} \\ R_2(1, 2) &= \{0, 1\} \cup \{0, 1\} \circ \{\epsilon, 1, 00, 01\}^* \circ \{\epsilon, 1, 00, 01\} \\ R &= \mathbf{0 + 1 + (0 + 1)} \cdot (\mathbf{\epsilon + 1 + 00 + 01})^* \cdot (\mathbf{\epsilon + 1 + 00 + 01})^* \\ &= \mathbf{0 + 1 + (0 + 1)} \cdot (\mathbf{\epsilon + 1 + 00 + 01})^* \\ &= (\mathbf{0 + 1}) \cdot (\mathbf{\epsilon + 1 + 00 + 01})^* \\ R &= (\mathbf{0 + 1})(\mathbf{1 + 00 + 01})^* \end{aligned}$$

Den finalen RA kann man gut interpretieren: Nach einem Übergang von q_1 zu q_2 mit 0 oder 1 ist der Endzustand (q_2) erreicht. Dort verbleibt man (ϵ oder mit beliebig vielen 1) oder wechselt mittels 00 oder 01 über q_1 zurück zu q_2 .



Übung 4.6

Gegeben sei ein DEA $M = (\{q_0, q_1\}, \{0, 1\}, \delta, q_0, \{q_1\})$ mit folgendem δ . Verwenden Sie



das in Beispiel 4.6 beschriebene Verfahren zur Bestimmung eines RA für $L(M)$ und vereinfachen Sie den entwickelten RA mit folgenden Regeln: $\mathbf{u} \cdot \mathbf{u}^* = \mathbf{u}^* \cdot \mathbf{u} = \mathbf{u}^*$ und $\mathbf{v} + \boldsymbol{\varepsilon} = \boldsymbol{\varepsilon} + \mathbf{v}$. Vergleichen Sie Ihr Ergebnis mit dem Ausdruck $R = \mathbf{0}^* \mathbf{1} (\mathbf{0} + \mathbf{1}^* \mathbf{1})^*$.

4.4 Reguläre Ausdrücke in der Praxis

In Definition 4.1 haben wir für RA die „wissenschaftliche Notation“ verwendet und anschließend mittels Klammensparregeln übersichtlicher gestaltet.

Texteditoren Im berufspraktischen Bereich der Softwareentwicklung haben sich andere Muster, s. Abbildung 2.2 (S. 32), etabliert. Sie leisten z.B. in Texteditoren gute Dienste beim Suchen, Ersetzen oder Filtern von Teilzeichenketten und werden bis heute eingesetzt.

Ein markanter Unterschied zwischen der wissenschaftlichen Notation regulärer Ausdrücke und der in der Praxis üblichen ist die Verwendung des Metazeichens $+$, wie in a^+ für aa^* . Da die Eingabe a^+ in Programmtexten oder Formularfeldern so nicht möglich ist, schreibt man a^+ . Dies ist zwar vernünftig, hat aber zur Folge, dass die ursprüngliche Bedeutung des Zeichens $+$ als „oder“, wie in Definition 4.1 auf Seite 105 festgelegt, nun durch ein anderes Symbol getragen werden muss. Hierfür hat man mit $|$ ein Symbol gewählt, welches wir auch in der BNF zur Grammatiknotation für alternative Regeln verwenden. Anstatt $(\alpha+\beta)$ müssen wir also $(\alpha|\beta)$ schreiben. α und β stehen hier für beliebige reguläre Ausdrücke.

| statt +

Praxisrelevante
Syntax regulärer
Ausdrücke

Wie Tabelle 4.1 zeigt, gibt es eine ganze Reihe zusätzlicher *Metazeichen*, wie `[`, `]`, `(`, `)`, `{`, `}`, `|`, `?`, `+`, `*`, `^`, `$`, `\` und `.`, die in einer solchen *praxisrelevanten Syntax* regulärer Ausdrücke verwendet werden können. Leider unterliegt diese Syntaxdefinition keiner Standardisierung, sodass die einzelnen Implementierungen (in verschiedenen Programmiersprachen) teilweise stark differieren.

Zusätzlich wurden eine Reihe von Makrozeichen-Sequenzen (bestehend aus zwei Metazeichen) vereinbart, um problematische ASCII-Zeichen, wie Zeilenumbrüche, in einen RA aufnehmen zu können. Tabelle 4.2 zeigt die Details.

⁵Natürlich kann eine solche Handrechnung für größere n recht aufwendig sein.

| Metazeichen | Bedeutung |
|-------------|--|
| . | ... steht für ein belieb. Zeichen (nicht \n, \r. (s. Tabelle 4.2)). |
| a^* | ... steht für 0 bis beliebig viele Vorkommen von a . |
| a^+ | ... steht für mindestens ein Vorkommen von a . |
| $a^?$ | ... steht für ein optionales Zeichen. Im Beispiel darf a auftreten oder auch nicht. (gleichbedeutend mit $a\{0,1\}$) |
| $\{m,n\}$ | ... steht für m bis n Vorkommen des Zeichens oder Metazeichens, $m, n \in \mathcal{N}$. |
| $\{n\}$ | ... steht für n Vorkommen des Zeichens oder Metazeichens. |
| (ab) | ... Klammern erlauben Teilausdrücke zu einer Einheit zusammenzufassen um anschließend Operationen wie +, * oder für den gesamten Teilausdruck anzuwenden. |
| $A B$ | ... steht für ein Zeichen A ODER B (kann auch auf komplexe geklammerte Teilausdrücke angewendet werden, z.B.: $(a b) [0-9]+$. |
| $[ABC]$ | Eckige Klammern beschreiben eine Auswahl (Zeichenklasse ohne Reihenfolge). Dieses Beispiel steht also für genau ein Zeichen A,B oder C. |
| $[a-zA-Z]$ | Der Bindestrich innerhalb eckiger Klammern bestimmt einen Bereich. Daher steht dieses Beispiel für genau ein Zeichen von a bis z oder A bis Z. |
| $[^A]$ | Das Dach (^) negiert die Auswahl. Daher wird hier ein beliebiges Zeichen außer A beschrieben. Wird gern verwendet um verbotene Zeichen in einem Ausdruck zu beschreiben. |

Tabelle 4.1: Praxis-relevante Syntax regulärer Ausdrücke: Metazeichen

| Metazeichen | Bedeutung |
|-------------|--|
| \n | ein Zeilenumbruch |
| \r | ein Wagenrücklauf (unter Windows steht \r\n für einen Zeilenumbruch) |
| \t | ein Tabulatorzeichen |
| \s | ein einzelnes Leerzeichen |
| \d | der reguläre Teilausdruck [0-9] |
| \w | ein Buchstabe, eine Ziffer oder der Unterstrich [a-zA-Z_0-9] |

Tabelle 4.2: Praxisrelevante Syntax regulärer Ausdrücke: Metazeichen-Sequenzen

Beispiel 4.7

Im Folgenden geben wir einige Beispiele für syntaktisch korrekte RA in Praxisnotation



zusammen mit der damit jeweils beschriebenen Sprache an.

| | |
|-------------|--|
| ... | trifft 3 beliebige Zeichen (außer \r und \n) |
| .{1,5} | trifft 1,2,3,4 oder 5 beliebige Zeichen (außer \r und \n) |
| [0-9]+ | trifft eine beliebig(mindestens ein Zeichen) lange Ziffernfolge wie etwa: 058237435 oder aber auch 0 |
| a*b | trifft beliebig viele a (oder keins) gefolgt von genau einem b |
| (a*) (b*) | trifft beliebig viele a ODER beliebig viele b oder auch das leere Wort! |
| \d+, \d\d | trifft eine Fließkommazahl der Form: 234,32 oder 0,98 identisch mit [0-9]+,[0-9][0-9] |
| [1-9][0-9]* | trifft eine beliebig lange Ziffernfolge aber ohne vorangestellte Nullen. |
| [\r\t\n\s] | trifft ein typisches Whitespace |
| [1-5]0 | trifft genau die Ziffernfolgen 10, 20, 30, 40 oder 50. |

Möchte man das Tastaturzeichen ? mit einem regulären Ausdruck beschreiben, muss das Symbol \ (Backslash) vor das Metazeichen gesetzt werden, um dessen Bedeutung (Option) aufzuheben: also \?. Wollen wir beispielsweise den arithmetischen Ausdruck $(234 + 23) * 4$ mit einem regulären Ausdruck erkennen, so eignet sich $\backslash([0-9]+\+[0-9]+\+)*[0-9]+$ dafür.



Didaktischer Hinweis 4.1

Man könnte auf die Idee kommen, einen Ausdruck wie [a*|c] zu formulieren, was jedoch nicht erlaubt ist. Eine Zeichenklasse [...] muss ausschließlich aus Zeichen bestehen. Ein gültiger Ausdruck wäre hingegen (a*)|(c) oder noch kürzer a*|c aufgrund der Priorisierung der Operationen .

Auch bei der Compilerentwicklung in FLACI werden wir mit *RA in der Praxisnotation* arbeiten. Indem wir uns am POSIX⁶-Standard orientieren, beschränken wir uns daher auf einen typischen Kern, s. Tabelle 4.1 und Tabelle 4.2.



Computerübung 4.1

Verwenden Sie das FLACI-Modul „Reguläre Ausdrücke“ und wählen Sie zunächst den Bereich „Definition“. Arbeiten Sie den interaktiven Lehrtext durch und prägen Sie sich die wichtigsten Notationen der (nicht standardisierten) Praxisnotation für RA ein.



Computerübung 4.2

Öffnen Sie den Bereich „Experimentieren“ des FLACI-Moduls „Reguläre Ausdrücke“ und übertragen Sie den Ergebnisausdruck $(0+1)(1+00+01)^*$ aus Beispiel 4.6 in das entsprechende Eingabefeld: $(0|1)(1|00|01)^*$. Prüfen Sie die Äquivalenz dieses Ausdrucks mit $0^*1(0+10^*)^*$. Weisen Sie nach, dass diese beiden Ausdrücke nicht äquivalent sind.



Computerübung 4.3

Konvertieren Sie den RA $(0+1)(1+00+01)^*$ in einen NEA_E. Simulieren Sie die Analyse des Wortes 10011100 mit diesem NEA. Was stellen Sie fest? Konvertieren Sie den NEA

⁶POSIX steht für „Portable Operating System Interface“ und ist eine Spezifikation der Benutzer- und Software-Schnittstelle des Betriebssystems. Die aktuelle Version stammt aus dem Jahr 2004. Auch Kommandozeileninterpreter sind darin erfasst. Die Standard-POSIX-Shell ist die aus UNIX bekannte Korn-Shell. Hilfsprogramme, wie awk, vi oder echo, sind ebenfalls Teil des POSIX-Standards.

danach in einen äquivalenten DEA und simulieren Sie die Analyse des gleichen Wortes. Beurteilen Sie das Ergebnis. Vereinfachen Sie den DEA und vergleichen Sie das Ergebnis mit dem DEA aus Beispiel 4.6.

Computerübung 4.4

Verwenden Sie den Bereich „Experimentieren“ des FLACI-Moduls „Reguläre Ausdrücke“ und entwerfen Sie einen RA für Datumsangaben der Form TT.MM.JJJJ, wobei lediglich die Jahresangaben 2000, 2001, ..., 2008 zugelassen sind. Außerdem sollen alle Monate (01, 02, ..., 12) vereinfachend die Tage 01,02,...,31 haben dürfen.



Lassen Sie sich von FLACI einige Eingabewörter generieren, die Ihrem RA entsprechen. Geben Sie sich einige Eingabewörter, wie 12.03.2008, 09.09.2007, 39.09.2207 und 08.06.2008, zur Analyse vor.

Vergleichen Sie Ihren RA mit dem unten angegebenen RA und führen Sie eine Äquivalenzprüfung durch.

Lassen Sie FLACI einen äquivalenten NEA_ε erzeugen und simulieren Sie die Syntaxprüfung für 12.03.2008. Wenn Sie (alles mit FLACI) die ε-Übergänge dieses NEA_ε entfernen, entstehen noch mehr Zustände. Simulieren Sie mit dem gleichen Eingabewort. Im Anschluss wandeln Sie den NEA in einen DEA um und minimieren diesen.

```
((0[1-9])|((1|2)[0-9])|(3(0|1)))\.(0[1-9]|(10|11|12))\.200[0-8]
```

In Skriptsprachen, wie JavaScript, Perl, Tcl, awk oder Python und Groovy, sind reguläre Ausdrücke ebenso integriert, wie in compilierenden Programmiersprachen, z.B. in Delphi, Java oder Visual C++.

Neben *deterministischen* Pattern-Matching-Verfahren finden auch *nichtdeterministische* Verwendung. Aus der Sicht der theoretischen Informatik ist dies keineswegs verwunderlich, denn sie bieten prinzipiell die gleiche Leistungsfähigkeit und können im Einzelfall sogar effizienter arbeiten.

Aktuelle matcher sind durch *Rückwärtsreferenztechnik* (*Backreference*) erweitert worden. Dabei werden Variablen in ein Muster eingebaut. Diese werden beim Abgleich an das passende Teilwort gebunden. Wenn dieser Prozess von links nach rechts abläuft, müssen dann alle weiter rechts stehenden Vorkommen dieser Variablen mit dem an diese Variable gebundenen Teilwort matchen. Es entstehen *dynamische Muster*.

Rückwärts-
referenztechnik
Backreference

Beispielsweise matcht der Ausdruck ([abc]*)!\\1 die Wörter ! und cab!cab. Dies wird mit exponentiellem Aufwand im worst case bezahlt.

Auf diese Weise können syntaktisch komplexer strukturierte Wörter verarbeitet werden. Zu beachten ist, dass der Bereich regulärer Mengen dadurch im Allgemeinen überschritten wird. RA mit Backreferences sind *keine* regulären Ausdrücke im Sinne der theoretischen Informatik. Ihre Beschreibungskraft geht über die regulären Sprachen hinaus.

In FLACI wird auf Rückwärtsreferenzen gänzlich verzichtet.

Übung 4.7

Machen Sie sich unter Verwendung entsprechender Informationsangebote (man, -?) mit der



Perl, egrep Rückwärtsreferenz-Technik in Perl und egrep (Weiterentwicklung von grep – global regular expression print) vertraut.



Computerübung 4.5

Geben Sie einen NEA (mit möglichst wenigen Zuständen) für folgende Sprache an: $(0+1)^*(000+111)(0+1)^*$. Da der RA nach wissenschaftlicher Definition vorliegt, muss er zuerst in die Praxisnotation übertragen werden.



Computerübung 4.6

Prüfen Sie durch Umformung, ob die folgenden regulären Ausdrücke äquivalent sind: $(a(b+(ab+ac))^*)^*a$ und $a((b^*+(a(b+c))^*)^*a)^*$. Verwenden Sie FLACI (Praxisnotation!), um Ihr Ergebnis zu verifizieren.



Computerübung 4.7

Konstruieren Sie einen EA für $10 + (0+1)0^*1$.



Übung 4.8

Gegeben sei die Sprache $L = \{w \mid w = aw' \text{ mit } w' \in \{a, b\}^*\}$. Entwerfen Sie einen DEA M für L und entwickeln Sie daraus einen äquivalenten regulären Ausdruck. Wenden Sie dabei das im Beweis (Teil 2) von Satz 4.1 vorgestellte Verfahren⁷ an. Setzen Sie FLACI ein.



Übung 4.9

Gegeben sei die Sprache L mit

$L = \{w \mid w \in \{a, b\} \text{ und } w \text{ endet auf } b \text{ und } w \text{ enthält nicht } aa\}$.

- (a) Geben Sie einen regulären Ausdruck an, der L definiert. Nennen Sie mindestens drei Wörter aus L .
- (b) Konstruieren Sie einen DEA für L .
- (c) Geben Sie für L eine Typ-3-Grammatik an.



Beispiel 4.8

Ein typisches Beispiel für die Anwendung von RA in der Praxis ist die *Eingabevalidierung* bei graphischen Bedienoberflächen sowohl in Webanwendungen als auch in Desktopanwendungen.

Eingabevalidierung

Meldet man sich etwa bei einem Internet-Portal an, wird oftmals eine gültige Emailadresse, ein Nutzernamen oder eine Kreditkartennummer gefordert. Für (syntaktisch) unzulässige Eingaben liefert die Webanwendung einen Hinweis wie in Abbildung 4.4. Die Webanwendung muss also überprüfen, ob eine gültige Emailadresse eingegeben wurde oder nicht.

Für die syntaktische Überprüfung können wir einen RA verwenden, was uns viele Zeilen Programmcode erspart. Dieser ist zwar komplex und für das ungeübte Auge schwierig lesbar, jedoch sehr kompakt und leistungsfähig.



Beispiel 4.9

Der folgende RA beschreibt eine typische, leicht abweichende Form einer syntaktisch korrekten E-Mail-Adresse:

$[a-zA-Z0-9\.\+\-\]+@[a-zA-Z0-9\.\-\]+\.[a-zA-Z]{2,4}$

⁷Der Rechenaufwand ist durchaus verkraftbar. Die Autoren haben es auch selbst auf sich genommen.

The screenshot shows the Wikipedia.de login page. At the top right, there's a link to 'Anmelden' (Login) and a message: 'Ihre Spenden helfen, Wikipedia zu betreiben.' Below that is a red-bordered error message: 'Fehler bei der Anmeldung: Die E-Mail-Adresse wird nicht akzeptiert, weil sie ein ungültiges Format (eventuell ungültige Zeichen) zu haben scheint. Bitte gib eine korrekte Adresse ein oder leere das Feld ([[Hilfe:E-Mail]]).'. The main form is titled 'Benutzerkonto anlegen' (Create User Account). It contains fields for 'Benutzername' (username), 'Passwort' (password), 'Passwort wiederholen' (repeat password), and 'E-Mail-Adresse' (email address). Below these fields is a note: 'Optional. Ermöglicht anderen Benutzern, über E-Mail Kontakt mit dir aufzunehmen, ohne dass du deine Identität offenlegen musst, sowie das Zustellen eines Ersatzpasswortes (Hilfe E-Mail).'. There are two checkboxes: one for 'Benutzer auf diesem Computer dauerhaft anmelden' (remember user on this computer) and one for 'Benutzerkonto anlegen' (create account). On the left sidebar, there are links for 'Spezialseite', 'Navigation' (including Hauptseite, Über Wikipedia, Themenportale, Von A bis Z, Zufälliger Artikel), 'Mitmachen' (Help, Autorenportal, Letzte Änderungen, Spenden), 'Suche' (Search), 'Werkzeuge' (Artikel, Volltext, Hochladen, Spezialseiten), and a search bar.

Abbildung 4.4: Anmeldung eines neuen Benutzers bei Wikipedia.de

Dabei steht der erste Teil bis zum Zeichen @ für eine Folge von Zeichen, die in E-Mail-Adressen im Allgemeinen erlaubt sind. Nach dem @ folgen mögliche Subdomains, der Domainname und eine zwei- bis vierstellige Landeskennung, wie .de.

4.5 Reguläre Ausdrücke in Scannergeneratoren

Übersetzungsprozesse werden in Kapitel 5 ausführlich behandelt. Als Quellsprachen werden im Allgemeinen kfS verwendet. Eine Übersetzung eines Wortes aus der Quell- in die Zielsprache beginnt mit der Syntaxanalyse des betrachteten Wortes, dem Parsing. Dies ist Aufgabe eines *Parsers*. Dieser bekommt Unterstützung durch einen *Scanner*, der sich um die regulären Sprachen, die gewöhnlich in kfS vorkommen, kümmert. Typische Beispiele sind Variablennamen, Schlüsselworte, E-Mail-Adressen, URLs, Zahlwerte usw.

Scanner sind also die Parser regulärer Subsprachen. Im Compilerbau nennt man diese auch Lexer oder Tokenizer. Die einzelnen *Tokenklassen* sind reguläre Sprachen, die zwischen keinem und unendlich vielen Wörtern (Lexemen) umfassen.

Tokenklassen können sehr bequem mit regulären Ausdrücken beschrieben werden. Diese Beschreibungen füttert FLACI einem *Scannergenerator*, der den jeweiligen Scanner erzeugt. Die „Hintergrundarbeit“ kann man sich so vorstellen, dass ein NEA erzeugt wird, der die einzelnen Subsprachen erkennt und folglich durch „Zusammenschaltung“ der Einzel-NEAs gebildet wird.

Wir illustrieren dies an einer einfachen Sprache eines Zeichenroboters (ZR).

Parser
Scanner

Tokenklassen

Scannergenerator

Turtle Zeichenroboter *ZR* ist die Sprache eines *imaginären Zeichenroboters*, den wir die *Turtle* nennen. Wir haben die Vorstellung, dass die Turtle einen kleinen Zeichenstift besitzt. Dieser Stift ist so angebracht, dass die Turtle bei ihren Vorwärtsbewegungen eine Spur (als Stiftstrich) hinterlässt.

VW *n* bedeutet, dass sich der Roboter *n* Schritte *vorwärts* bewegt.

RE *n* lässt ihn eine *Rechtsdrehung* um *n* Grad vollführen.

STIFT *n* bestimmt die *Strichstärke* *n* des Stiftes bis zur nächsten Veränderung.

FARBE *f* legt die aktuelle *Stiftfarbe* (weiß, rot, blau, grün, gelb, schwarz) fest. Sie gilt bis zur nächsten Veränderung.

WH *n* [...] führt dazu, dass die mit Pünktchen symbolisierte Anweisungsfolge *n*-mal hintereinander ausgeführt wird (*Wiederholung*).

Damit lassen sich *ZR*-Programme formulieren, deren Interpretationen den o.g. Anweisungen entsprechen. Man kann solche Programme als an die Turtle gerichtete Befehlsfolgen interpretieren. Beispielsweise ergibt

WH 10 [WH 36 [VW 10 RE 10] RE 40]

eine „Blume“, die in Abbildung 4.5 ganz rechts dargestellt ist.



Didaktischer Hinweis 4.2

In Spezialkursen zum Compilerbau greift man den historischen Zugang auf und wählt als Zielsprache den Maschinencode (*MC*) einer konkreten Rechnerarchitektur bzw. des jeweiligen Betriebssystems. Nicht nur aus didaktischen Gründen, sondern auch den vielfältigen Anwendungszusammenhängen entsprechend, verwenden wir eine Hochsprache als Zielsprache. Compiler mit dieser Eigenschaft werden auch *Transcompiler* genannt. Für eine belastbare Motivation der Lernenden ist es von Bedeutung, dass die Interpretationen von Zielsprachen-Programmen zu *grafischen* oder *akustischen* Wirkungen führen.



Übung 4.10

Interpretieren Sie jedes der folgenden *ZR*-Programme. Zeichnen Sie.

VW 50

RE 270

RE 45 WH 2 [VW 100]

WH 4 [VW 100 RE 100]

WH 36 [WH 4 [VW 100 RE 90] RE 10]



Übung 4.11

Geben Sie *ZR*-Programme an, die die Figuren 1-3 in Abbildung 4.5 beschreiben.

In Abbildung 5.17 auf Seite 135 findet man eine formale Grammatik für *ZR*. Die Nichtterminale dieser Ausgangsgrammatik, die die Subsprachen definieren, und für die der Scanner zuständig sein wird, verwandeln sich zu Terminalen der sog. reduzierten Grammatik *G'* für *ZR*, s. Abbildung 4.6.

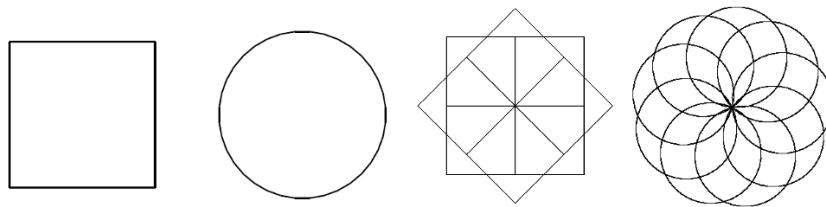


Abbildung 4.5: ZR-Figuren

$$\begin{aligned}
 G' &= (N', T', P', s') \\
 N' &= \{\text{Programm, Anweisungen, Anweisung}\} \\
 T' &= \{\text{WH, [,], FARBE, RE, STIFT, VW, Zahl, Farbwert}\} \\
 P' &= \begin{array}{l} \{\text{Programm} \rightarrow \text{Anweisungen} \\ \quad \text{Anweisungen} \rightarrow \text{Anweisung Anweisungen} \\ \quad \quad | \\ \quad \quad \varepsilon \\ \quad \text{Anweisung} \rightarrow \text{WH Zahl [Anweisungen]} \\ \quad \quad | \\ \quad \quad \text{FARBE Farbwert} \\ \quad \quad | \\ \quad \quad \text{RE Zahl} \\ \quad \quad | \\ \quad \quad \text{STIFT Zahl} \\ \quad \quad | \\ \quad \quad \text{VW Zahl } \} \end{array} \\
 s' &= \text{Programm}
 \end{aligned}$$

Abbildung 4.6: Reduzierte kfG für ZR (identisch mit Abbildung 5.18)

In G' sind *Farbwert* und *Zahl* Terminale. *Ziffern*, *Ziffer* und *ErsteZiffer* entfallen.

$$\begin{array}{l} \text{Farbwert} \rightarrow \text{blau} \mid \text{gelb} \mid \text{gruen} \mid \text{rot} \mid \text{schwarz} \\ \text{Zahl} \rightarrow \text{ErsteZiffer Ziffern} \\ \text{Ziffern} \rightarrow \text{Ziffer Ziffern} \mid \varepsilon \\ \text{Ziffer} \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \\ \text{ErsteZiffer} \rightarrow 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \end{array}$$

Zur Definition der Token *Farbwert* und *Zahl* verwenden wir reguläre Ausdrücke (in Praxisnotation). Für *Farbwert* ist das trivial:

`blau|gelb|gruen|rot|schwarz`

wobei zu beachten ist, dass die Zeichen `|` von keinen Leerzeichen umgeben sind.

Der zur Beschreibung von *Zahl* gewählte reguläre Ausdruck

`[1-9] [0-9]*`

sorgt für die Unterdrückung von Vornullen.



Computerübung 4.8

Entwickeln Sie einen regulären Ausdruck für die deutschen Autokennzeichen. Dabei können mehrere reguläre Ausdrücke „hintereinander geschaltet“ werden. Das entspricht, wie wir später sehen werden, durchaus dem Verhalten des Scanners im Compilerbau. Dieser besitzt jedoch noch einige zusätzliche Entscheidungsregeln, da mehrere reguläre Ausdrücke auf das Eingabewort passen könnten.



Computerübung 4.9

Eine Textdatei enthält Messwerte einer Anlage:

- 1: BauteilA17; 23.45; 12.87;
- 2: BauteilB3; 198.88; 553.99; 12.45;
- 3: Tor3; 0.53;
- 4: BauteilC1; 55.54; 24.82;
- 5: BauteilD1; 33.32;

Ein Eintrag beginnt stets mit einer Zeilennummer gefolgt von einem Doppelpunkt. Es folgt ein Komponentennamen der mit mindestens einem Buchstaben beginnt. Anschließend folgen beliebig viele Messwerte in der angegebenen Form.

Entwickeln Sie eine kfG für dieses Dateiformat. Definieren Sie anschließend geeignete Tokenklassen und geben Sie die zugehörige reduzierte Grammatik an. Verwenden Sie zunächst nur das Modul „kontextfreie Grammatiken“ von FLACI und die Suchkomponente Ihrer Textverarbeitungssoftware.



Computerübung 4.10

Ein Mitarbeiter möchte wissen, welche Bauteile Messwerte geliefert haben. Dazu soll die Textdatei (aus Computerübung 4.9) nach Komponentennamen durchsucht werden. Diese sollen ausgegeben werden.



Computerübung 4.11

Ein Buchhaltungssystem soll Daten aus einer externen Datei einlesen. Es handelt sich um eine einfache Textdatei, die Preisangaben enthält. Beispiel:

19,32
182,47
1,45
2,99

Entwickeln Sie eine reguläre Grammatik für einen korrekten Eingabewert (eine Preisangabe) aus dieser Datei.

Erstellen Sie aus dieser regulären Grammatik einen NEA.

Die im Folgenden angegebene Datei wurde von einer Sekretärin per Hand verfasst und kann Schreibfehler enthalten. Folgende Eingaben sind fehlerhaft:

0,,32
3,447
,43
20,9
74,

Natürlich ist es unmöglich solche Tippfehler automatisch zu korrigieren. Jedoch könnte ein Programm diese potenziellen Fehler anzeigen, damit sie die Sekretärin bereinigen kann.

Definieren Sie Token für korrekte und inkorrekte Eingabewerte. Testen Sie mit verschiedenen Eingabetexten.



5 Sprachübersetzer

5.1 Compiler und Interpreter

Programme, die Programme einer *Quellsprache* in zugehörige Programme einer *Zielsprache* überführen, nennt man *Compiler*, s. Abschnitt 1.2. Historisch hat sich der Bedarf an Compilern aus der Verwendung höherer, problemorientierter, algorithmischer Programmiersprachen (kurz: Hochsprachen) ergeben: Statt Anwendungsprogramme aus maschinennahen Instruktionen aufzubauen, fügt man leicht lesbare Sprachelemente in syntaktisch korrekter Weise zu einem Programm zusammen. Damit diese Programme von einer „primitiven Zielmaschine“, genauer: einem Prozessor und dem Betriebssystem, verarbeitet werden können, ist eine entsprechende Übersetzung (Compilation) notwendig.

Compiler

Ein weiterer Grund für das Aufkommen von Compilern bestand darin, Quellcodeprogramm-Übersetzungen für verschiedene Maschinen vornehmen zu können. Damit wurde es möglich, mit ein und derselben Sprache für mehrere Zielmaschinen zu programmieren.

Interpreter gehen anders vor als Compiler: Das vorgelegte Programm wird nicht vollständig (in eine neue Datei) übersetzt, sondern „portionsweise“ analysiert, in eine zugehörige Folge von (Prozessor-)Instruktionen übertragen und „ausgeführt“. Programmiersprachen, wie Python und JavaScript, besitzen Interpreter, auch in Form von Kommandozeilen-Interprettern. Man nennt sie auch *Skriptsprachen*. Eine abgrenzende Definition ist schwierig.

Interpreter

Skriptsprachen werden zunehmend und in verschiedensten Anwendungsbereichen eingesetzt. Man erspart sich den ggf. zeitaufwendigen Compilationsvorgang, bevor man „sieht“, ob das Programm das Gewünschte leistet. So erklärt sich die enorme Verbreitung von JavaScript. Auch andere Websprachen, wie PHP oder Python, gehören in diese Kategorie. Viele Werkzeuge bieten ihren Nutzern einfache Skriptsprachen zur Erweiterung der Werkzeugeigenschaften an. Ein Beispiel ist Lua mit vordergründiger Verwendung in Computerspielen und Software, die den Nutzer in die Lage versetzt, selbst Anpassungen am Spiel bzw. am Programm vorzunehmen.

Skriptsprachen

Übung 5.1

Setzen Sie sich aufgrund der beiden beschriebenen Verarbeitungskonzepte mit dem Argument auseinander, dass Programme in Interpretersprachen üblicherweise langsamer laufen als semantisch äquivalente Programme in Compilersprachen. Schließen Sie Zyklusstrukturen in Ihre Betrachtung ein.



Ein vollständiges Programmiersystem kommt nicht ohne Interpretation durch einen bestimmten Prozessor aus. Gleichgültig wie viele Übersetzungsschritte (Phasen) durch Anwendung diverser Compiler auf den Quelltext bzw. die in jedem Schritt erzeugten Programme in Zwischensprachen stattgefunden haben, muss das Programm in einer Zielsprache schließlich doch interpretiert werden.

Mischformen

Während Interpreter den interaktiven Programmentwurf bequemer gestalten, liefern Compiler im Allgemeinen performanten Zielcode, wenn es sich um Maschinencode handelt. Bestrebungen die Vorteile beider Systeme zu vereinen, führte zu Mischformen: *Compiler-Interpreter-Systeme* (Interpreter für den Entwurf, Compiler für die Ausführung), *inkrementelle Compiler*, die jeweils nur die editierten Programmteile übersetzen, *Zwischencode-Interpreter*, die ein (vor)compiliertes Programm in einem Zwischencode (z.B. Bytecode bei Java) interpretieren und damit eine *virtuelle Maschine* repräsentieren, und *Just-in-time-Compiler*, die zur Laufzeit (also gerade noch „rechtzeitig“) in Maschinencode übersetzen.

Die folgenden Betrachtungen sind auf Compiler und Interpreter in der klassischen Form beschränkt.

5.2 Modellierung von Übersetzungsprozessen

Wir wählen *ZR*, d.h. die Sprache eines Zeichenroboters, als Quell- und *PDF* als Zielsprache. *ZR* haben wir in Abschnitt 4.5 ausführlich eingeführt und *PDF* ist eine etablierte Dokumentbeschreibungssprache.

T-Diagramm

Im Folgenden betrachten wir den Übersetzungsprozess. Zur Modellierung verwenden wir T-Diagramme¹. Der Name dieser Diagramme röhrt von ihrer Grundform (rechts in Abbildung 5.1) her.

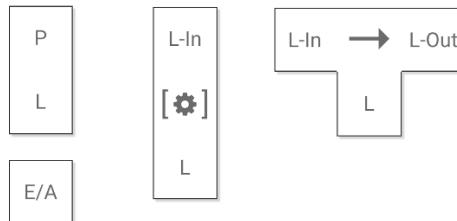


Abbildung 5.1: Diagrammdarstellung für Programm, Interpreter und Compiler

Links in Abbildung 5.1 ist Programm *P*, geschrieben in der Sprache *L*, dargestellt.

¹T-Diagramme gehen auf McKeeman, Horning und Wortman zurück, die diese grafische Darstellungsform zur Beschreibung von Übersetzungsprozessen 1971 einführten. Sie wurden von Kerner (1990) und anderen aufgegriffen. Watt und Brown (2000) verwenden hierfür Tombstone-Darstellungen (tombstone = Grabstein), die auch ein bisschen wie Bäckermützen aussehen.

Darunter befindet sich ein E/A-Baustein (Eingabe/Ausgabe), der in FLACI bei Bedarf links und rechts an P „geklebt“ werden kann. Dies modelliert dann die Berechnung $A=P(E)$.

Die Bauform eines Interpreters (Abbildung 5.1, Mitte) ist der eines Programms P ähnlich, wird jedoch unten an ein in L-In geschriebenes Programm geheftet. Der Interpreter ist in Sprache L geschrieben.

Der im rechten Teil von Abbildung 5.1 dargestellte Compiler übersetzt Programme aus Sprache L-In in L-Out-Programme und ist selbst in Sprache L geschrieben.

In FLACI können diese drei Grundformen sehr benutzerfreundlich miteinander kombiniert werden, was die Modellierung beliebiger Übersetzungsszenarien ermöglicht.

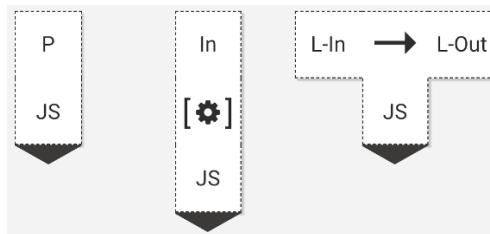


Abbildung 5.2: Darstellung lauffähiger Programme, Interpreter und Compiler

Handelt es sich bei der Implementationssprache L des betrachteten Programms um JavaScript (JS), so ist es in FLACI lauffähig. Wir sprechen von einer virtuellen JavaScript-Maschine, d.h. FLACI's Muttersprache ist JS. Damit abstrahieren wir von einem weiteren Übersetzungsschritt, nämlich von JavaScript in Maschinencode.

Optisch ist die Lauffähigkeit (Ausführbarkeit) eines JS-Programms an einem unten angesetzten ausgefüllten Dreieck-Sockel erkennbar, s. Abbildung 5.2.

Die gestrichelten Linien bedeuten in FLACI, dass diese Programme zu diesem Zeitpunkt noch nicht wirklich existieren. Das ist Potenzial für die Modellierung: Man baut sich das Entwicklungsziel zusammen und kann danach schrittweise alle verwendeten Bausteine vervollständigen - bis alle Linien durchgezogen sind. Anschließend können einzelne Bausteine oder das gesamte Diagramm ausgewählt und getestet bzw. ausgeführt werden.

Beispiel 5.1

Abbildung 5.3 enthält ein Diagramm für den $ZR \rightarrow PDF$ -Übersetzungsprozess, wie wir ihn am Ende von Abschnitt 4.5 beschrieben haben: $ZR \rightarrow PDF$ -Compiler und PDF -Interpreter.

Im Rahmen der Planung dieses (kleinen) Übersetzungsprojekts kommen einige Alternativen in Betracht, die im Vorfeld diskutiert werden sollten:

1. $ZR \rightarrow TG \rightarrow PDF$. In dieser zweiphasigen Compilation wird die Programmier-



sprache *LOGO* als Zwischensprache für *TG* (turtle geometry) verwendet. Im Programming Wiki (<https://programmingwiki.de/Startseite>) steht *LOGO* zur Verfügung.

2. $ZR \rightarrow SVG \rightarrow PDF$. Für die $SVG \rightarrow PDF$ -Compilation gibt es Standardprogramme, sodass wir uns nur um die $ZR \rightarrow SVG$ -Übersetzung kümmern müssen.
3. $ZR \rightarrow PS \rightarrow PDF$ -Compilation ist eine ganz ähnliche Variante, wie 2.
4. $ZR \rightarrow PDF$. Dies ist die Direktübersetzung, für die wir uns hier entscheiden. Sie ist durch die Bereitstellung entsprechender jsPDF-Bibliotheken möglich geworden.

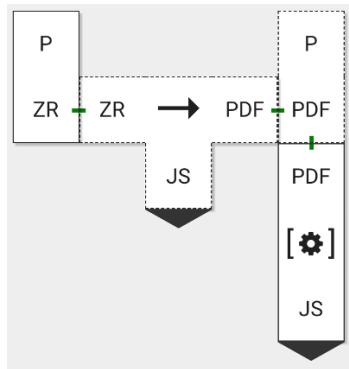


Abbildung 5.3: $ZR \rightarrow PDF$ -Compiler und PDF-Interpreter

Beim Zusammenziehen der Bausteine in Abbildung 5.3 mit der linken Maustaste ist die jeweils gewünschte Verbindung erst dann wirksam, wenn die kleinen Tacker, die übereinstimmende Sprachen verbinden, sichtbar sind.

Wie wir an den gestrichelten Linien sehen, ist lediglich der $ZR \rightarrow PDF$ -Compiler herzustellen. Dies wird endgültig in Abschnitt 9.1 geschehen. Der *PDF*-Interpreter ist in FLACI vorhanden oder kann aus einer Vorgabe leicht gewonnen werden.

Computerübung 5.1

Machen Sie sich mit FLACI (Compiler und Interpreter) vertraut, indem Sie die Modellierung gemäß Abbildung 5.3 selbst vornehmen.

Computerübung 5.2

Modellieren Sie die Verarbeitung eines Java-Programms. Stichwort: Bytecode! Beachten Sie, dass dies eine Modellierungsaufgabe ist, die keinen Anspruch auf Ausführung erhebt.

Computerübung 5.3

Modellieren Sie die vollständige Übersetzung eines LATEX-Programms in *PDF*. Es wird nicht erwartet, dass das Modell ausführbar ist.

Da Compiler selbst Programme sind, können sie auch auf Compiler angewendet werden. Dafür gibt es zwei klassische Übersetzungsszenarien: Bootstrapping und Cross-compilation.



Beispiel 5.2

Ein $S \rightarrow M$ -Compiler kann im Allgemeinen nicht in einem Schritt in („menschenunfreundlichem“ Maschinencode) M programmiert werden. Deshalb beginnt man beispielsweise mit einem C-Programm für den $S \rightarrow M$ -Compiler. Dieses kann mit Hilfe eines allgemein verfügbaren C-Compilers, auch in JavaScript z.B. CaptCC, in das gewünschte Zielprogramm compiliert werden, s. Abbildung 5.4.

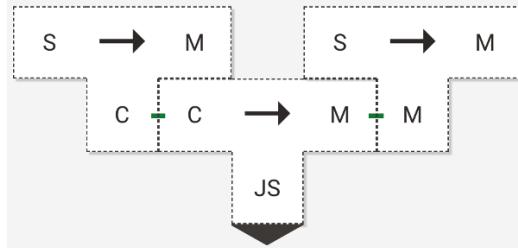


Abbildung 5.4: $S \rightarrow M$ -Compiler in C und M

Natürlich ist es auch möglich, den $S \rightarrow M$ -Compiler selbst in S zu schreiben. Dann kann dieser Compiler mit dem in M geschriebenen $S \rightarrow M$ -Compiler aus Abbildung 5.4 übersetzt werden, s. Abbildung 5.5. Dies muss allerdings auf einem Computer geschehen, der M versteht. Es entsteht ebenso ein $S \rightarrow M$ -Compiler für die Maschine M , was für Effizienzvergleiche der beiden gleichartigen Compiler von Interesse sein kann.

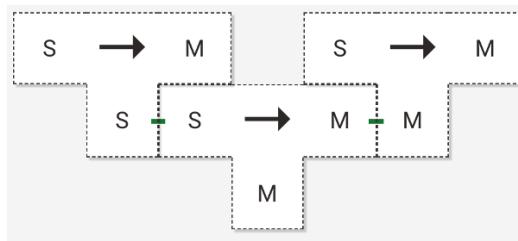


Abbildung 5.5: $S \rightarrow M$ -Compiler in S und M

Nun können wir den in S geschriebenen $S \rightarrow M$ -Compiler verwenden, um Spracherweiterungen von S zu S' per Hand in S zu programmieren, s. Abbildung 5.6. Nach einer entsprechenden Compilation erhalten wir einen $S' \rightarrow M$ -Compiler in M .

In Abbildung 5.6 sind die beiden Compilations von links nach rechts gut zu erkennen. Dies lässt sich mit Hilfe der T-Notation auch sehr gut in einem Diagramm darstellen, wie wir in Abbildung 5.7 sehen können.

Das Verfahren kann man weiter fortsetzen. Da es in jedem Schritt auf Compiler zurückgreift, die im vorhergehenden Schritt quasi mit „aufgestockten Bordmitteln“ erzeugt wurden, nennt man das Verfahren *Bootstrapping*. Der Begriff wird auch gern in Beziehung

Bootstrapping

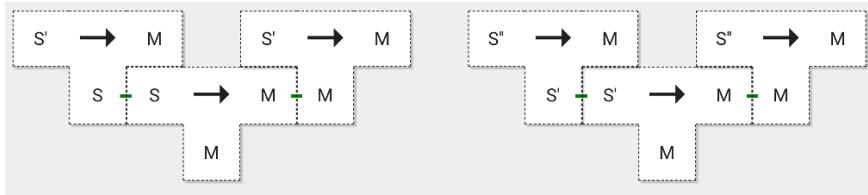


Abbildung 5.6: $S' \rightarrow M$ -Compiler in S und M , $S'' \rightarrow M$ -Compiler in S' und M

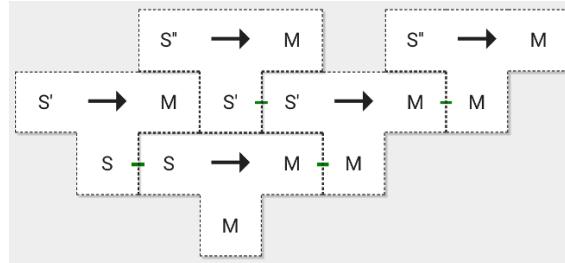


Abbildung 5.7: $S'' \rightarrow M$ -Compiler in M (Bootstrap)

zu Baron Karl Friedrich Hieronymus Freiherr von Münchhausen (Lügenbaron, 1720-1797) gebracht, der sich (samt Pferd) am eigenen Zopf aus dem Sumpf gezogen haben soll.

An Beispiel 5.2 zeigt sich die Leistungsfähigkeit von T-Diagrammen zur Modellierung mehrstufiger Übersetzungsprozesse. An mehreren Stellen in FLACI, werden Parser eingesetzt, die wiederum selbst mit dem Compiler Modul von FLACI geschrieben und übersetzt wurden.

Trojaner

Man sollte jedoch immer Sicherungskopien von älteren Versionen seines Compilers aufheben, wie eine Geschichte aus der OpenSource-Gemeinde erzählt: Bei der Entwicklung eines Compilers hatte jemand im Schritt von S zu S' einen Trojaner in den Compiler eingebaut, diesen in eine binäre Version übersetzt und anschließend den Trojaner wieder aus dem Quelltext entfernt (sodass niemand ihn im Code vorfinden konnte). Der binäre Compiler wurde nun von der Gemeinde weiterverwendet, um die nächste Generation zu erzeugen. Jedem mit diesem Compiler neu erzeugten Compiler wurde binär ebenfalls der Trojaner angehängt. Jedes Programm, welches mit einem solchen Compiler übersetzt wurde, trug auch den Trojaner in sich. Dies wurde erst mehrere Versionen später festgestellt. Er konnte nur mit großer Mühe wieder entfernt werden.

Computerübung 5.4

Wer dieses Trojaner-Beispiel konkret nachvollziehen und das umfangreiche T-Diagramm samt Compiler erproben möchten, kann ein entsprechendes Beispiel in der Beispielsammlung von FLACI unter „Bootstrapping Schadcode“ finden.

Übung 5.2

Bootstrap-Verfahren gibt es nicht nur im Compilerbau. Finden Sie heraus, wie eine Primzahlprüfung mit Bootstrapping durchgeführt werden kann.



Cross-Compiler sind Compiler, die auf einer bestimmten Maschine laufen, und Programmcode für eine andere Plattform generieren. Auf diese Weise können beispielsweise Programme für verschiedene Handys produziert werden, ohne dass dem Hersteller dieser Programme das entsprechende Handy jemals vorlag. Cross-Compiler spielen heute eine wichtige Rolle für *eingebettete Systeme* (embedded systems), die selbst über keine (üppigen) Softwareentwicklungsumgebungen verfügen. Ähnlich verhält es sich mit Spielkonsolen.

Beispiel 5.3

Gewünscht sei ein C++-Compiler für das IBM-Unix-Betriebssystem *Alpha*, wie in Abbildung 5.8.

Cross-Compiler

eingebettete
Systeme
Spielkonsole

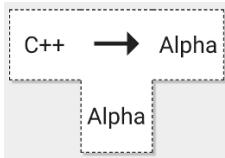


Abbildung 5.8: $\text{C}++ \rightarrow \text{Alpha}$ -Compiler, lauffähig auf *Alpha*

Wir schreiben den $\text{C}++ \rightarrow \text{Alpha}$ -Compiler natürlich nicht im *Alpha*-Maschinencode, sondern in C++ auf unserem Linux-Laptop mit x86-Befehlssatz. Dann nehmen wir den unter Linux laufenden C++-Compiler GCC und erzeugen einen $\text{C}++ \rightarrow \text{Alpha}$ -Compiler.

Wie aus Abbildung 5.9 hervorgeht, hat dies noch einen Haken: Der erzeugte Compiler läuft zwar auf unserem Linux-Laptop, jedoch nicht auf der *Alpha*. Also nehmen wir den oben hergestellten Compiler und wenden ihn wieder auf den in C++ geschriebenen $\text{C}++ \rightarrow \text{x86}$ -Compiler an, s. Abbildung 5.10. Damit erhalten wir den gewünschten Compiler. Wie Abbildung 5.11 zeigt, kann dieser auf der *Alpha* zur Übersetzung von C++-Programmen verwendet werden.

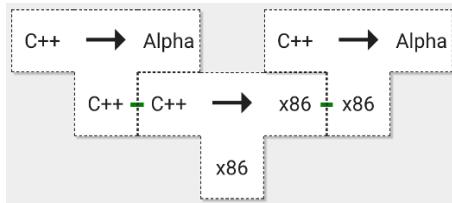


Abbildung 5.9: $\text{C}++ \rightarrow \text{Alpha}$ -Compiler lauffähig auf *x86* (z.B. Linux-Laptop)

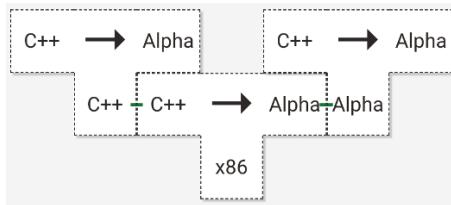


Abbildung 5.10: $C++ \rightarrow \text{Alpha}$ -Compiler lauffähig auf Alpha

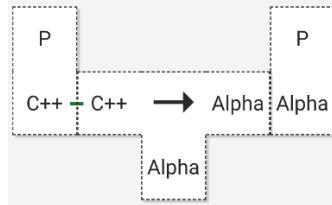


Abbildung 5.11: $C++ \rightarrow \text{Alpha}$ -Compiler übersetzt $C++$ -Programme auf der Alpha

Nachdem wir Compiler bei der Entwicklung von Übersetzungsprinzipien konzeptionell eingesetzt haben, betrachten wir die Struktur der zu entwickelnden Parser und Interpreter.

Phasen eines Compilers

Die klassischen *Phasen eines Compilers* sind

1. die lexikalischer Analyse (Scanner)
2. die syntaktische Analyse (Parser)
3. die Zwischencodegenerierung (AST)
4. die semantische Analyse
5. die Code-Optimierung
6. die Speicherplatzvergabe
7. die Zielcodegenerierung

Die ersten drei Phasen gelten analog auch für Interpreter, bezogen auf Quellcode-Abschnitte.

In diesem Kapitel werden wir uns auf den Analyseteil konzentrieren.



Didaktischer Hinweis 5.1

In der Tat gibt es noch einige in der Liste unerwähnte Aktionen: So wird beispielsweise eine Symboltabelle eingerichtet, die je nach Möglichkeit Typinformationen usw. aufnimmt. Bei streng typisierten Sprachen ist diese Tabelle auch eine Voraussetzung für das type checking (semantische Analyse) und scoping. Da „echte“ Programmiersprachen im Allgemeinen kontextsensitiv sind, jedoch durch kontextfreie Grammatiken beschrieben werden, sind zusätzlich Prüfungen von Kontextbedingungen erforderlich.

Vermutlich überrascht es Sie, dass nur die letzte Phase, nämlich die Zielcodegenerierung maschinenabhängig ist. Dass die gesamte Compilerentwicklung, also alle hier angegebenen Phasen, zielmaschinenfern auf einem beliebigen Computer ausgeführt werden können, haben wir schon in Beispiel 5.3 ausgenutzt (Cross-Compiler).

Wir sehen uns nun die ersten beiden Phasen etwas genau an. Bisher haben wir nur das Parsing zur Syntaxanalyse genannt, jedoch noch nichts darüber gesagt, wie ein Parser konzeptionell arbeitet.

5.3 Lexikalische Analyse

Abbildung 5.12 zeigt eine Pipeline: dem Scanning folgt das Parsing.

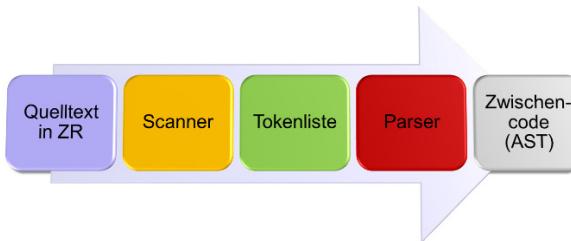


Abbildung 5.12: Scanner und Parser in einem Compiler

Motivation für die Verwendung eines Scanners sind eine bessere Strukturierung des Analyseprozesses, höhere Effizienz und leichtere Portierbarkeit.

Dabei erwartet der Parser, dass der lexikalische *Scanner* (auch *Lexer* oder *Tokenizer*) eine hilfreiche Vorarbeit leistet, die sich in Form einer *Tokenliste*, auch *Tokenstrom* genannt, darstellt.

Token sind syntaktische Elementarbausteine des Eingabewortes. Genauer: Ein *Token* ist ein Paar, das aus dem Namen der *Tokenklasse* (*Tokentyp*) und dem jeweils

Scanner

Tokenliste

Token

Lexem zugehörigen *Lexem* besteht, z. B. [keyword, "begin"]. Für Programmiersprachen typische Tokenklassen sind beispielsweise Schlüsselwörter, Variablennamen, Wörter für Zahlen diverser Bereiche und Operatoren. Zur Definition der zu einer Tokenklasse gehörenden Lexeme dient ein regulärer Ausdruck.

Für das Festlegen von Tokenklassen gibt es keine Definition oder Vorschrift, es ist eher eine Erfahrungsentscheidung. Typischerweise werden die (regulären) Typ-3-Anteile der Grammatik der kontextfreien Quellsprache als Tokenklassen herausgegriffen.

Im Zeichenstrom (Eingabewort) identifiziert der Scanner die Token, klassifiziert die Lexeme und hinterlässt eine Folge von *Tokenklasse-Lexem-Paaren*. Obwohl die jeweilige Tokenerwartung im Prozess der *syntaktischen Analyse* vom Parser ausgeht, darf man sich das als separaten Durchlauf (Vorverarbeitung) vorstellen. Anschließend werden die *einzelnen gelesenen Zeichen* nie wieder angefasst.

Beispiel 5.4

Je nach Grammatik könnte sich aus dem Eingabewort

```
var radius=12; writeln(radius);
```

beispielsweise der Tokenstrom

```
([keyword, "var"] [identifier, "radius"] [assignment, "="] [number, "12"]
 [delimiter, ";"] [keyword, "writeln"] [leftpar, "("] [identifier, "radius"]
 [rightpar, ")"] [delimiter, ";"])
```

ergeben.

Wie man in Beispiel 5.4 sehen kann, gibt es pro Tokenklasse unterschiedlich viele Lexeme: Im Extremfall gibt es nur eins, wie bei `assign`, `leftpar` und `rightpar`, oder es sind unendlich viele, wie bei `number`.

Didaktischer Hinweis 5.2

Hier bietet sich eine Analogie zur Objekt-orientierten Programmierung an: Die Lexeme bilden die *Instanzen* der entsprechenden *Token-Klasse*.

Whitespaces

Typischerweise enthält ein Eingabewort Leerzeichen, Zeilenwechsel, Tabulatoren, kurz: *Whitespaces*, die zur besseren Lesbarkeit dieses Eingabewortes dienen, für die Übersetzung jedoch keine Bedeutung haben. Diese *Whitespaces* verstehen wir als Lexeme der Tokenklasse IGNORE, etwa mit der Definition wie in der Tabelle von Beispiel 5.5 (letzte Zeile). Sie werden genauso behandelt wie andere Token. Die Lexeme gehören jedoch nicht zu den Terminalen der Grammatik. Die IGNORE-Token werden nicht in den Tokenstrom aufgenommen.

IGNORE

Wichtig ist, dass man kein Lexem von *IGNORE innerhalb* eines anderen Tokens verwendet. Dieser Leerzeichen-Fehler kann sich rasch einschleichen: Beispielsweise sind bei einer Zuweisung wie: `var x = "Hallo Welt!"`; die Leerzeichen bis auf die innerhalb einer Zeichenkette, hier also zwischen o und W, irrelevant. Um diese Unterschiede korrekt zu erfassen, vereinbaren wir weiter unten

zusätzliche Regeln, welche Tokenklasse bei mehreren Möglichkeiten ausgewählt werden soll.

Ein *Scanner-Generator* nimmt sämtliche Tokenbeschreibungen und erzeugt für jede Tokenklasse einen endlichen Automaten (möglichst Minimal-DEA). Diese können zu einem einzigen NEA zusammengefügt werden. Wird der erzeugte Scanner auf ein Eingabewort angewandt, so kann man aus den eingenommenen Endzuständen auf das jeweilige Token der Tokenfolge schließen. Im der Praxis verwendet ein solcher Scanner eine ausgeklügelte Puffertechnik, um nach der Tokenerkennung (Erreichen des entsprechenden Endzustands) das zugehörige Lexem herauszugreifen. Manchmal ist auch eine Vorausschau auf das potenzielle Lexem-Ende notwendig.

Dabei erfordern „überlappende“ Tokendefinitionen, sofern sie nicht vermeidbar sind, transparente Regularien des zu entwickelnden Scanners. Beispiel 5.5 ist so gewählt, dass wir diese Problematik exemplarisch studieren können.

Beispiel 5.5

Wir betrachten drei Tokenklassen: NUM, IF und ID, für Wörter für ganze Zahlen (number), das Schlüsselwort if und Variablennamen (identifier). Zur Definition der Tokenklassen werden reguläre Ausdrücke verwendet, genau einer je Tokenklasse, und vom Scanner-Generator erwartet:

Scanner-
Generator



| Tokenklasse | regulärer Ausdruck | |
|-------------|--|---|
| NUM | $[\backslash+\backslash-]?\backslash[0-9]+\backslash+$ | Man beachte die Backslashes vor + und -. |
| IF | if | Tokenklasse umfasst genau ein Lexem. |
| ID | [a-z] [a-z0-9]* | Tokenklasse umfasst unendlich viele Lexeme. |
| IGNORE | $[\backslashn\backslasht\backslashr\backslashs]$ | \s für space (und weitere Leerzeichenvarianten), nicht etwa für whitespace, wie bei regulären Ausdrücken. |

Abbildung 5.13 zeigt einen Scanner-NEA für die in der Tabelle angegebenen Tokenklassen. Schaut man in dieser Abbildung von oben nach unten, so erkennt man die Teil-Automaten für NUM, IF und ID. Vom jeweiligen Endzustand gibt es je einen spontanen Übergang zu q_0 , dem Startzustand des Gesamtautomaten. Als IGNORE-Zeichen ist zusätzlich ein Leerzeichen ' ' aufgenommen worden. Es darf in beliebiger Wiederholung zwischen den Token stehen.

Die FLACI-Simulation dieses NEA für das Eingabewort "aif+4" erzeugt 19 Konfigurationenfolgen. Genau fünf davon akzeptieren das gesamte Eingabewort. Sie sind in Abbildung 5.14 zu sehen.

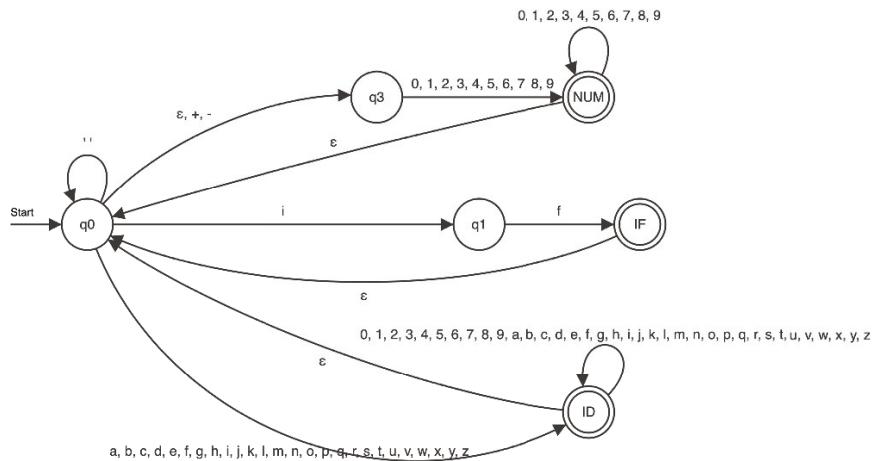


Abbildung 5.13: Gesamt-NEA für den Tokenizer

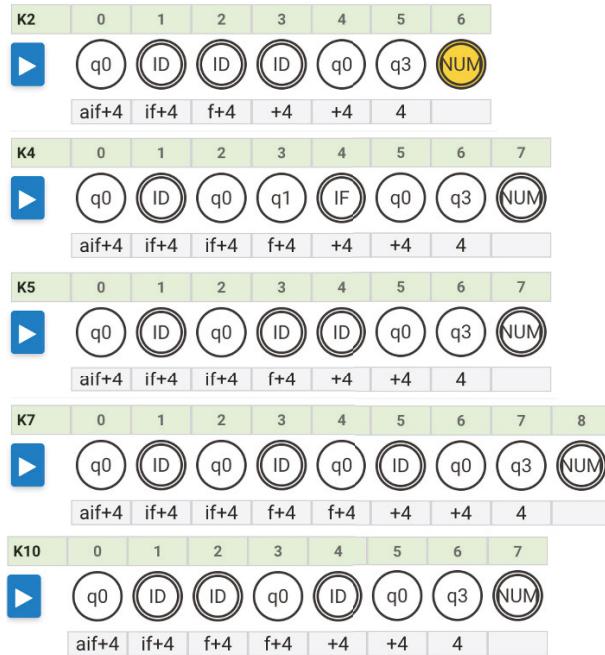


Abbildung 5.14: Scanner-NEA: Konfigurationenfolgen für aif+4

Diese mit FLACI generierten erfolgreichen Konfigurationenfolgen für das Eingabewort "aif+4" entsprechen den Tokenfolgen:

K2 ([ID, "aif"], [NUM, "+4"])

K4 ([ID, "a"], [IF, "if"], [NUM, "+4"])

K5 ([ID, "a"], [ID, "if"], [NUM, "+4"])

K7 ([ID, "a"], [ID, "i"], [ID, "f"], [NUM, "+4"])

K10 ([ID, "ai"], [ID, "f"], [NUM, "+4"])

Welche dieser korrekten Tokenfolgen ist die gewünschte? Nehmen wir an, dies wäre die Tokenliste für K2. Dann wären wir mit der *longest-prefix-Strategie* erfolgreich. Diese hat sich bei modernen Programmiersprachen, wie beispielsweise Java und C, durchgesetzt. In K2 ist der längste Präfix für ID "aif" und erst das Zeichen + beendet den Bezeichner.

longest-prefix-
Strategie

Die longest-prefix-Strategie lässt sich informell wie folgt beschreiben:

1. Die längste von mehreren auf eine Tokenklasse passende Zeichenkette bestimmt diese Klasse.
2. Bei gleich langen passenden Zeichenketten bestimmt die zuerst definierte Klasse die Tokenklasse.

Punkt 2 bedeutet, dass die Reihenfolge (von oben nach unten), in der die Token-Definitionen angegeben wurden, die bestimmt, welche der grundsätzlich passenden Tokenklasse zur Anwendung kommt.

Während dies für longest-prefix nur im Falle gleich langer Zeichenketten gilt, wird dies in der *First-wins-Strategie* bedingungslos angewandt. D.h. die Reihenfolge des Tokenbeschreibungsaufschreibs von *oben nach unten* entspricht abnehmender Priorität und bestimmt das Matching.

First-wins-
Strategie

FLACI verwendet First wins: Das (von oben nach unten) erste passende Muster wird genommen. Die für die entsprechende Tokendefinition in FLACI eingebauten Werkzeuge werden im Zusammenhang mit dem Modul „Compiler und Interpreter“ behandelt. In den Abbildungen 5.15 und 5.16 gibt es schon einen kleinen Vorgeschmack.

In diesen beiden Abbildungen werden links die Token X und Y definiert. Das Eingabewort "hallo" passt sowohl auf Y als auch (zeichenweise) auf X. Je nach Definitionsposition der beiden Tokenklassen entstehen verschiedene Tokenfolgen. Die Ergebnisse sind ebenfalls in den beiden Abbildungen angegeben.

Computerübung 5.5

Simulieren Sie gedanklich und mit FLACI (Modul „Abstrakte Automaten“) die Verarbeitung der Eingabewörter "ai f+4", "if" und "aifb+4" nach Beispiel 5.5.



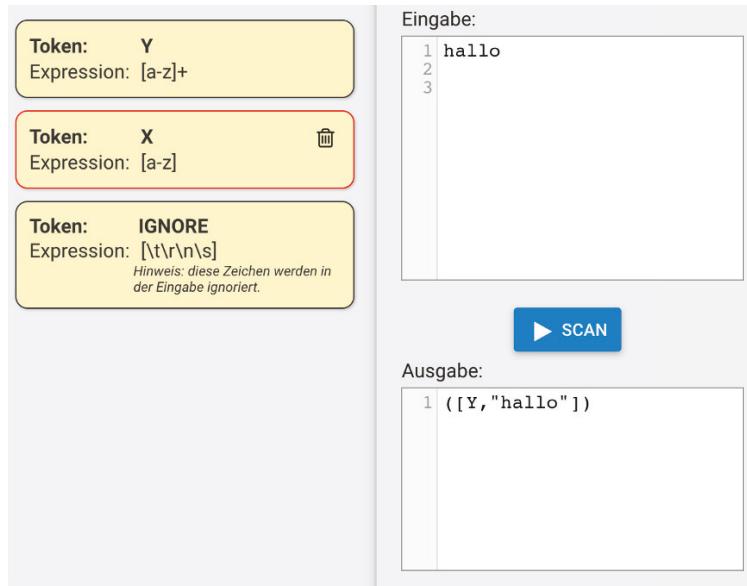


Abbildung 5.15: First wins: Definition von Y vor X

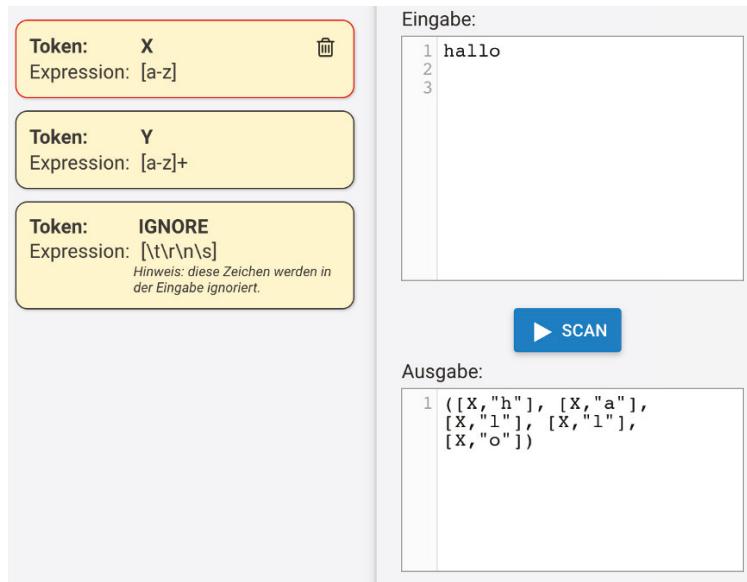


Abbildung 5.16: First wins: Definition von X vor Y

5.4 Syntaktische Analyse

Als Nächstes kommt der Parser zum Einsatz. Durch die Bereitstellung des Tokenstroms braucht er sich nicht mehr um die Analyse der Lexeme, die dem jeweiligen Token zugeordnet wurden, zu kümmern. Der Parser versteht sie als Terminale einer *reduzierten Grammatik*.

Zum Vergleich einer formalen Grammatik G einer Quellsprache mit einer zugehörigen reduzierten Grammatik G' kehren wir nun zu unserem Zeichenroboter-Beispiel aus Abschnitt 4.5 (S. 117) zurück. Der erste Schritt der $ZR \rightarrow PDF$ -Compilation ist die Definition der Quellsprache ZR . Dies leistet die formale Grammatik in Abbildung 5.17.

reduzierte
Grammatik

$$\begin{aligned}
 G &= (N, T, P, s), \\
 N &= \{Programm, Anweisungen, Anweisung, Farbwert, Zahl, Ziffern, \\
 &\quad Ziffer, ErsteZiffer\}, \\
 T &= \{\text{WH}, [,], \text{FARBE}, \text{RE}, \text{STIFT}, \text{VW}, \text{blau}, \text{gelb}, \text{gruen}, \\
 &\quad \text{rot}, \text{schwarz}, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}, \\
 P &= \begin{aligned} \{Programm &\rightarrow \text{Anweisungen}, \\ \text{Anweisungen} &\rightarrow \text{Anweisung Anweisungen} \\ &\quad | \quad \epsilon, \\ \text{Anweisung} &\rightarrow \text{WH Zahl [Anweisungen]} \\ &\quad | \quad \text{FARBE Farbwert} \\ &\quad | \quad \text{RE Zahl} \\ &\quad | \quad \text{STIFT Zahl} \\ &\quad | \quad \text{VW Zahl}, \\ \text{Farbwert} &\rightarrow \text{weiss} | \text{blau} | \text{gelb} | \text{gruen} | \text{rot} | \text{schwarz}, \\ \text{Zahl} &\rightarrow \text{ErsteZiffer Ziffern}, \\ \text{Ziffern} &\rightarrow \text{Ziffer Ziffern} | \epsilon, \\ \text{Ziffer} &\rightarrow 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9, \\ \text{ErsteZiffer} &\rightarrow 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 \}, \\ s &= \text{Programm} \end{aligned}
 \end{aligned}$$

Abbildung 5.17: kfG für ZR

$$\begin{aligned}
 G' &= (N', T', P', s'), \\
 N' &= \{\text{Programm, Anweisungen, Anweisung}\}, \\
 T' &= \{\text{WH, [,], FARBE, RE, STIFT, VW, Zahl, Farbwert}\}, \\
 P' &= \{\text{Programm} \rightarrow \text{Anweisungen}, \\
 &\quad \text{Anweisungen} \rightarrow \text{Anweisung Anweisungen} \\
 &\quad \quad | \quad \varepsilon, \\
 &\quad \text{Anweisung} \rightarrow \text{WH Zahl [Anweisungen]} \\
 &\quad | \quad \text{FARBE Farbwert} \\
 &\quad | \quad \text{RE Zahl} \\
 &\quad | \quad \text{STIFT Zahl} \\
 &\quad | \quad \text{VW Zahl } \}, \\
 s' &= \text{Programm}
 \end{aligned}$$

Abbildung 5.18: Reduzierte kfG für ZR

Computerübung 5.6

Geben Sie den Ableitungsbaum des Wortes WH 4 [VW 10 RE 90] aus ZR, gemäß der Grammatik in Abbildung 5.17, mit FLACI (kontextfreie Grammatiken) an.

Übung 5.3

Geben Sie die Definitionen der in G' , s. Abbildung 5.18, verwendeten Token an (Klassen und jeweils zugehörige Lexeme). Vergleichen Sie G' mit der Ausgangsgrammatik G für ZR.

Didaktischer Hinweis 5.3

Wie weiter oben bereits mitgeteilt wurde, darf man sich die Zusammenarbeit zwischen Scanner und Parser als einen Prozess mit zwei aufeinander folgenden Durchläufen vorstellen: Zuerst tastet der Scanner das Eingabewort vollständig ab und danach analysiert der Parser den vom Scanner hinterlassenen Tokenstrom. Obwohl es heute sehr ausgeklügelte Hand-in-Hand-arbeitende und inkrementelle Verarbeitungstechniken gibt, ist diese phasenbezogene Sicht konzeptionell und didaktisch sehr zu empfehlen. Die Analysephase wird auch das Frontend und der Rest (ab einschl. semantischer Analyse) das Backend eines Compilers genannt.

Top-down-
Analyse
Bottom-up-
Analyse

Die Arbeitsweise von Recognizern und Parsern folgt dem Ableitungsprozess eines Wortes bei formalen Grammatiken. Von einem Spitzensymbol aus wird das zu analysierende Wort hergeleitet. Man spricht von einer *Top-down-Analyse*. Ebenso gibt es Parser, die in entgegengesetzter Richtung arbeiten und das Eingabewort schrittweise zum Satzsymbol reduzieren. Letzteres nennt man *Bottom-up-Analyse*. Für beide Analysetechniken werden wir noch typische Verfahren kennenlernen.

In der Praxis ist es häufig so, dass der Scanner vom Parser bei Bedarf aufgerufen wird. Wenn der Parser an der aktuellen Analysestelle ein Zahlwort erwartet, dann beauftragt er den Scanner, ihm das nächste Token zu liefern. Falls es ein Token der Klasse `number` ist, wird auch das zugehörige Lexem, z.B. `12`, zum Aufbau eines zugehörigen AST benutzt. Kann der Scanner den Parserwunsch nicht erfüllen, entsteht ein Syntaxfehler.

Außerdem gibt es Geräte mit geringem Speicher, wie etwa einfache Taschenrechner, die eine zeichenweise Eingabe unmittelbar in einen Tokenstrom verwandeln. Der aus der funktionalen Programmierung, wie etwa mit Racket, bekannte `reader` erledigt das analog.

scan on demand



6 Kellerautomaten und kontextfreie Sprachen

6.1 Grenzen endlicher Automaten

Endliche Automaten reichen zur Beschreibung beliebiger kontextfreier Sprachen nicht aus. Dies belegen die folgenden Beispielsprachen.

Beispiel 6.1

$L_1 = \{w \mid w \in \{a, b\}^* \text{ und } w \text{ enthält ebenso viele Zeichen } a \text{ wie } b\}$



Beispiel 6.2

Ein Wortpalindrom ist ein Wort w , das rückwärts gelesen (w') dasselbe Wort ergibt, d.h. $w = w'$. Die Sprache der Wortpalindrome über dem Alphabet $\{a, b\}$ ist $L_2 = \{w \mid w \in \{a, b\}^* \text{ und } w = \text{umkehr}(w)\} = \{w \mid w = vv' \text{ oder } w = vav' \text{ oder } w = vbv', v' = \text{umkehr}(v), v, v' \in \{a, b\}^*\}$. Die Symmetriearchse durch ein solches Wortpalindrom verläuft also zwischen zwei oder durch ein Alphabetzeichen.

$L_2 = \{\epsilon, a, b, aa, bb, aaa, bbb, aba, bab, abba, baab, ababa, baaab, \dots\}$.



Beispiel 6.3

L_3 : Sprache der geklammerten arithmetischen Ausdrücke, wie beispielsweise $(x \cdot (x+y))$ und nicht etwa $x \cdot (x) + (y)$, obwohl die gleichen Alphabetzeichen verwendet wurden.



Woran liegt es, dass endliche Automaten nicht in der Lage sind, die Sprache L_1 in Beispiel 6.1 zu akzeptieren? Offensichtlich hängt das damit zusammen, dass die Anzahl n der a's bzw. b's nicht vorgegeben ist. n wird erst durch das konkrete Eingabewort, wie etwa $babaababbba$ - hier gilt $n = 6$, bestimmt.



Eine naheliegende Idee zur Erweiterung endlicher Automaten mit Ziel, deren diesbezügliche „Defizite auszubügeln“, ist so eine Art „Merkzettel“, auf dem wir jedes gelesene Zeichen a durch einen Merker, z. B. $|$, notieren. Jedes gelesene b führt dann zur Streichung eines Merkstriches. Das Eingabewort gehört zur Sprache, wenn der Merkzettel am Ende leer ist.

Merkzettel

Leider hat die Sache einen Haken: Was geschieht, wenn das Eingabewort im Anfangsstück mehr b's als a's enthält? Dann haben wir noch gar keinen (nun zu streichenden) Merker gesetzt oder es sind zu wenige. Anstelle eines neutralen Merkstriches ist es offenbar notwendig, Eingabezeichen-spezifische Merker, z.B. I_a für a und I_b für b, zu verwenden. Wird ein Zeichen gelesen, das sich vom jeweils vorhergehenden Zeichen unterscheidet, wird der zugehörige Merker entfernt. Ansonsten wird der Eingabezeichen-Merker auf einen als *Stapel* organisierten Merkzettel

Stapel

Keller
push
pop
gelegt. Abbildung 6.1 unterstreicht die anschauliche Vorstellung von einem Stapel (auch *Keller* genannt) und illustriert die Wirkung der Operationen *push* und *pop*.

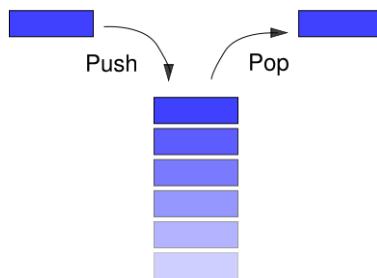


Abbildung 6.1: Stapel (stack) und Operationen *push* und *pop*

top of stack
Die beiden Operationen *push* und *pop* beziehen sich stets auf das oberste Kellerzeichen, das *top of stack* (*tos*). Die Ausführung von *push(x,stack)* legt das Zeichen *x* auf den Stapel *stack*, sodass *x* fortan das neue *tos* von *stack* ist. Durch *pop(stack)* wird das zum Zeitpunkt der Ausführung der Operation gültige *tos* vom Stapel entfernt und kann für eine bestimmte Weiterverarbeitung herangezogen werden.

Der Einsatz eines Stapels ist auch zur Beschreibung der Sprache L_2 aus Beispiel 6.2 unumgänglich. Bei L_2 ist nicht nur die Anzahl der Zeichen *a* und *b* von Bedeutung, sondern zusätzlich deren Reihenfolge. Offensichtlich muss der gelesene Teil *v* eines Eingabewortes $w = vv'$ zeichenweise auf den Stapel gelegt - man sagt: gekellert - werden, bis die Wortmitte erreicht ist. Ist das letzte Zeichen von *v* das *tos*, so wird geprüft, ob das folgende Zeichen, also das erste Zeichen von *v'*, mit dem aktuellen *top of stack* übereinstimmt. Wenn das der Fall ist, wird der Kellerinhalt dabei verbrauchend gelesen (*pop*) usw. Falls *v'* das Umkehrwort von *v* ist, so gehört *w* zur betrachteten Sprache. Dies gilt analog auch für Palindrome, deren Symmetrieachsen durch ein Zeichen (statt zwischen zwei aufeinanderfolgenden Zeichen) verlaufen. Dann gelten $w = vxv'$, wobei $v, v' \in \{a, b\}^*$ und $x \in \{a, b\}$. Allerdings bleibt noch die Frage, wie die Wortmitte erkannt und damit das Entkellern eingeleitet wird.

Übung 6.1

Illustrieren Sie die Verwendung eines Stapels zur Erkennung korrekt geklammerter arithmetischer Ausdrücke, s. L_3 in 6.3.



6.2 Nichtdeterministischer Kellerautomat (NKA)

Nach dieser beispielbezogenen Betrachtung der Erweiterung endlicher Automaten durch ein „Gedächtnis“, das im Stapelbetrieb beschrieben und gelöscht wird, kann

eine Definition des neuen Automatentyps angegeben werden.

Definition 6.1

Ein nichtdeterministischer Kellerautomat, kurz: NKA oder KA, (engl.: NPDA = nondeterministic pushdown^a automaton) ist durch ein 7-Tupel $M = (Q, \Sigma, \Gamma, \delta, q_0, k_0, E)$ definiert. Die verwendeten Symbole haben folgende Bedeutungen:

- Q ... endliche Menge der Zustände,
- Σ ... Eingabealphabet,
- Γ ... Kelleralphabet,
- δ ... partielle Überführungsfunktion, $Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \rightarrow \wp_{\text{endlich}}(Q \times \Gamma^*)$,
- q_0 ... Anfangszustand, $q_0 \in Q$,
- k_0 ... Kellervorbelegungszeichen, $k_0 \in \Gamma$, $k_0 \notin \Sigma$ und
- E ... Menge von Endzuständen, $E \subseteq Q$.

^abedeutet so viel wie ‚nach unten drücken (winden)‘, nicht etwa ‚umschubsen‘.



Die Definition bedarf noch einiger Erläuterung und einer Ergänzung. Die Überführungsfunktion δ bestimmt die Arbeitsweise eines NKA-Modells, dessen Aufbau in Abbildung 6.2 dargestellt ist.

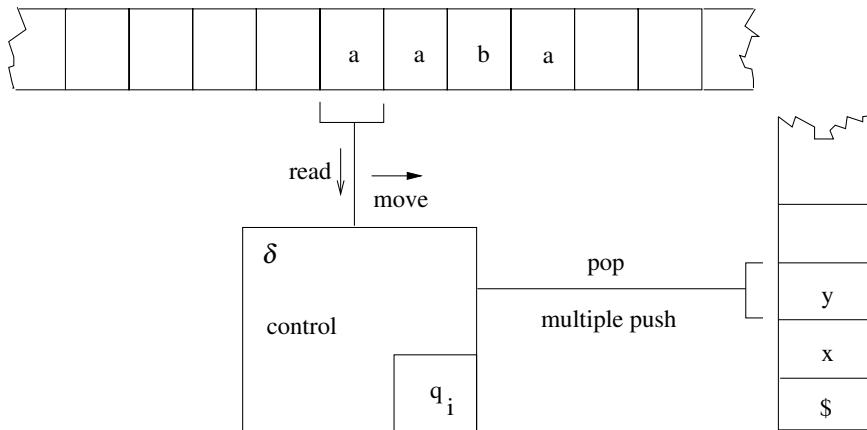


Abbildung 6.2: Aufbau eines Kellerautomaten-Modells

Die Arbeitsweise des Modells kann folgendermaßen beschreiben: Ein NKA wechselt von einem Zustand q_i in einen Zustand q_j , wenn k das oberste Kellerzeichen (tos) und a das aktuelle Eingabezeichen sind.

Achtung: Das tos wird stets *verbrauchend* gelesen, was der Operation *pop* für Kellerspeicher entspricht. Anschließend wird ein Kellerwort $k_1 k_2 \dots k_r$ auf den Stapel geschrieben. Hier findet die Operation *push* r -mal unmittelbar hintereinander Anwendung. In Abbildung 6.2 wurde dies als *multiple push* bezeichnet.

| | |
|------|---------------|
| pop | push |
| push | multiple push |

Wichtig ist die Reihenfolge. Wie in Abbildung 6.3 dargestellt, verabreden wir, dass die Zeichen in umgekehrter Reihenfolge auf den Keller geschoben werden. Nach dem Kellern von $k_1 k_2 \dots k_r$ ist k_1 (und *nicht* etwa k_r) das neue top of stack, darunter liegt k_2 usw.

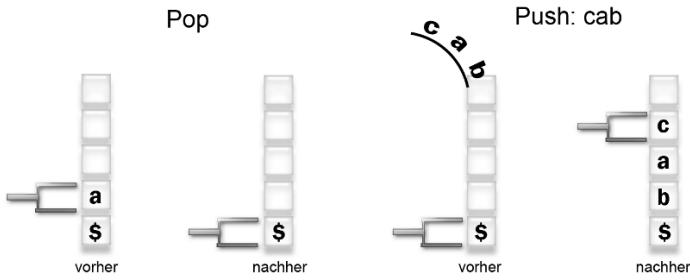


Abbildung 6.3: Operationen *pop* und *multiple push* angewandt auf eine Zeichenkette

Das in einem Schritt zu kellernde Wort darf auch leer sein. In diesem Fall schrumpft der Kellerinhalt.

Der beschriebene Vorgang wird in der Überführungsfunktion durch

$$\delta(q_i, a, k) \ni (q_j, k_1 k_2 \dots k_r)$$

ausgedrückt, und $\delta(q_i, a, k) \ni (q_j, \epsilon)$ beschreibt den Spezialfall des leeren Kellerwortes.

spontane Übergänge Außerdem sind spontane Übergänge ohne Verbrauch eines Eingabezeichens erlaubt:

$$\delta(q_i, \epsilon, k) \ni (q_j, k_1 k_2 \dots k_r).$$

Auch in diesem Fall wird das oberste Kellerzeichen verbrauchend gelesen.

Die Überführungsfunktion eines NKA hat damit folgende allgemeine Gestalt:

$$\delta : Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \rightarrow \wp_{\text{endlich}}(Q \times \Gamma^*),$$

\wp_{endlich} wobei $\wp_{\text{endlich}}(X)$ die Menge aller *endlichen* Teilmengen einer Menge X bezeichnet. Dies wird auf Seite 145 noch einmal aufgegriffen und begründet.

Crash Außerdem ist δ eine *partielle* Funktion. Gibt es für ein Tripel aus $Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma$ keinen Funktionswert, so stoppt der Automat durch Crash. Dies tritt auch dann ein, wenn der Keller leer ist: $\delta(q_i, a, \epsilon)$ ist nicht definiert. Ein bis zu einem Crash nicht vollständig eingelesenes Eingabewort, wird abgewiesen.

Zu Beginn des Analyseprozesses steht der Lesekopf auf dem erstem Zeichen des Eingabewortes auf dem Band. Der Keller enthält nur das Kellervorbelegungszeichen als aktuelles top of stack. Der Automat stoppt, wenn das Wort vollständig

gelesen wurde. Ggf. finden dann noch spontane Übergänge statt. Ein NKA akzeptiert das Eingabewort w genau dann, wenn

1. w vollständig abgetastet wurde und
2. der dann eingenommene Zustand ein Endzustand ist.

Der Kellerinhalt spielt in diesem Fall keine Rolle.

Der Analyseprozess einer NKA für ein Eingabewort wird durch eine Folge von Konfigurationen beschrieben. Eine Konfiguration eines NKA ist ein Tripel aus $Q \times \Sigma^* \times \Gamma^*$. Die zugehörige Liste von Konfigurationen heißt *Konfigurationenfolge*.

Didaktischer Hinweis 6.1

Eine tabellarische Darstellung der Konfigurationenfolge wird nachdrücklich angeraten.

Die von einem NKA definierte (oder akzeptierte oder erkannte) Sprache ist

$$L(M) = \{w \in \Sigma^* \mid (q_0, w, k_0) \xrightarrow{*} (q_e, \varepsilon, K), \text{ mit } q_e \in E, K \in \Gamma^*\}.$$

Im Gegensatz zu endlichen Automaten kann die dreistellige Überführungsfunktion eines NKA nicht als (zweidimensionale) Tabelle aufgeschrieben werden. Ein Zustandsgraph des NKA, der eine grafische Darstellung für δ enthält, ist oben bereits angegeben.

Die Markierungen der Kanten des Zustandsgraphen eines NKAs, wie z.B. $(0, b) : b0$ in Beispiel 6.4, haben folgenden Aufbau:

Akzeptanz

Konfiguration
Konfiguratio-
nenfolge

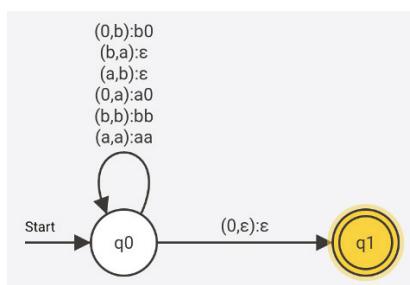
Transitionen

(oberstes Kellerzeichen, Eingabezeichen): Kellerwort.

Damit soll die wichtige Rolle des jeweils obersten Kellerzeichens hervorgehoben werden. Selbst bei einem spontanen Übergang wird es verbrauchend gelesen. Ausnahmslos! Dies unterstreicht die Verwandtschaft und Erweiterung des weniger mächtigen DEA und dessen Überführungsfunktion durch das Hinzufügen eines weiteren Parameters.

Beispiel 6.4

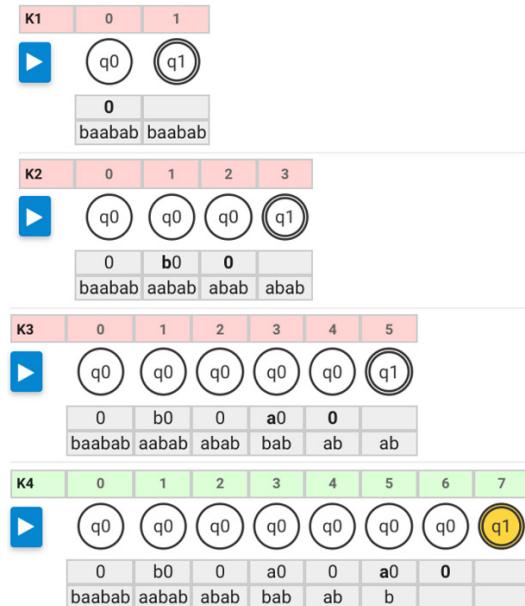
Der NKA $M_1 = (\{q_0, q_1\}, \{\text{a, b}\}, \{0, \text{a, b}\}, \delta, q_0, 0, \{q_1\})$ mit folgendem δ



| | | |
|-----------------------------------|-------|----------------------|
| $\delta(q_0, \text{a}, \text{b})$ | \ni | (q_0, ε) |
| $\delta(q_0, \text{a}, 0)$ | \ni | $(q_0, \text{a}0)$ |
| $\delta(q_0, \text{a}, \text{a})$ | \ni | (q_0, aa) |
| $\delta(q_0, \text{b}, \text{a})$ | \ni | (q_0, ε) |
| $\delta(q_0, \text{b}, 0)$ | \ni | $(q_0, \text{b}0)$ |
| $\delta(q_0, \text{b}, \text{b})$ | \ni | (q_0, bb) |
| $\delta(q_0, \varepsilon, 0)$ | \ni | (q_1, ε) |

akzeptiert genau die in Beispiel 6.1 angegebene Sprache L_1 . Die grundlegende Idee zur Konstruktion von M_1 wurde bereits vorgetragen. Die Simulation dieses NKA angesetzt auf das zu L_1 gehörende Eingabewort $baabab$ liefert 4 Konfigurationenfolgen, von denen eine (K4) erfolgreich ist:

Konfigurationenfolge(n) für: b a a b a b



Computerübung 6.1



Verwenden Sie das Modul „Abstrakte Automaten“ und legen Sie einen Automaten vom Typ NKA an. Simulieren Sie M_1 aus Beispiel 6.4 angesetzt auf das Wort $bbabaa$ mit FLACI und protokollieren Sie den Analyseprozess als Tabelle. Vergleichen Sie Ihre Lösung mit der im Folgenden angegebenen Konfigurationenfolge.

| Zustand | Eingabezeichen | Keller (links: top of stack) |
|---------|----------------|------------------------------|
| q_0 | bbabaa | 0 |
| q_0 | babaa | b0 |
| q_0 | abaa | bb0 |
| q_0 | baa | b0 |
| q_0 | aa | bb0 |
| q_0 | a | b0 |
| q_0 | ϵ | 0 |
| q_1 | ϵ | ϵ |

Machen Sie sich die Wirkung des Nichtdeterminismus klar. Verwenden Sie auch Wörter, die nicht zu $L(M_1)$ gehören.

In der Tabelle aus Computerübung 6.1 ist ein wichtiger Unterschied zwischen endlichen Automaten und Kellerautomaten deutlich erkennbar: Die mit NKA zu

leistende Berechnung der jeweiligen Folgekonfiguration erfordert nämlich nicht nur das oberste Kellerzeichen, wie das die Überführungsfunktion suggeriert. Zusätzlich *muss* der gesamte Kellerinhalt mitgeführt werden.

Obwohl ein NKA nur endlich viele Zustände q_i besitzt, gibt es abzählbar unendlich viele Funktionswerte (q_i, K) , denn es sind abzählbar unendlich viele Kellerwörter $K \in \Gamma^*$ bildbar. Die Potenzmenge $\wp(Q \times \Gamma^*)$ ist überabzählbar unendlich. Streicht man alle unendlichen Mengen in $\wp(Q \times \Gamma^*)$, so bleiben abzählbar *unendlich* viele endliche Mengen (als Elemente) übrig. Diese unendliche Menge $\wp_{\text{endlich}}(Q \times \Gamma^*)$ ist gerade der Wertebereich von δ .

Es ist also nicht die Endlichkeit der Zustandsmenge, die einen Automat als endlich charakterisiert, sondern die Endlichkeit des Wertebereiches von δ . NKA sind *keine* endlichen Automaten. (Die Überführungsfunktionen aller Automatenmodelle besitzen grundsätzlich endliche Argumentmengen.)

Computerübung 6.2

Geben Sie einen NKA für die Sprache der Palindrome über $\{a, b\}$, s. L₂ in Beispiel 6.2, an.

NEA vs. DEA



Definition 6.2

Zur NKA-Definition 6.1 gibt es eine alternative, gleichwertige *NKA-Definition als 6-Tupel – ohne Endzustandsmenge*: $M = (Q, \Sigma, \Gamma, \delta, q_0, k_0)$. Dieses Automatenmodell akzeptiert ein Eingabewort, wenn es vollständig gelesen wurde, der Automat also stoppt, und der Keller leer ist. Man sagt: Der Automat „akzeptiert per leerem Keller“.



Die Tatsache, dass die beiden alternativen NKA-Definitionen äquivalent sind, werden wir in Satz 6.1 beweisen.

FLACI ist auf die Arbeit mit der 7-Tupel-Definition beschränkt, unterstützt aber bei der wechselseitigen Transformation. Insofern ist die Angabe eines entsprechenden Transformationsverfahrens $NKA_6 \rightsquigarrow NKA_7$ nicht nur von theoretischem, sondern auch von praktischem Interesse, nämlich um 6-Tupel-NKAs für die Arbeit mit FLACI zu erschließen. Der Beweisteil 2 von Satz 6.1 beschreibt eine solche allgemeingültige Transformation.

Didaktischer Hinweis 6.2

Die Einführung von NKA in der 7-Tupel-Definition hat didaktische Vorteile, die sich aus der Nähe zu endlichen Automaten ergeben. Die Endzustandsmenge hat dort prinzipiell die gleiche Aufgabe wie bei 7-Tupel-Kellerautomaten.



Andererseits wird die 6-Tupel-Definition in einigen Beweisen sehr vorteilhaft verwendet. Insofern ist es notwendig, dass wir uns mit beiden Definitionen befassen.

**Satz 6.1**

Zu jedem 7-Tupel-NKA $M' = (Q', \Sigma', \Gamma', \delta', q'_0, k'_0, E')$ gibt es einen äquivalenten 6-Tupel-NKA $M = (Q, \Sigma, \Gamma, \delta, q_0, k_0)$ und umgekehrt. Die beiden Definitionen sind äquivalent.

$NKA_7 \rightsquigarrow NKA_6$

Beweis, Teil 1

Zu jedem NKA $M' = (Q', \Sigma', \Gamma', \delta', q'_0, k'_0, E')$ gibt es einen äquivalenten NKA $M = (Q, \Sigma, \Gamma, \delta, q_0, k_0)$ mit $L = L(M') = L(M)$.

Die Beweisaufgabe besteht in der Konstruktion von M aus M' , so dass M die Abarbeitung von M' für ein beliebiges Eingabewort simuliert. Ganz einfach die Endzustände zu eliminieren ist nicht die Lösung. Man muss nämlich beachten, dass M' einen (beliebigen) Endzustand einnehmen kann, um entweder die Akzeptanz des Analysewortes anzuzeigen oder mit dessen Verarbeitung fortzufahren. Diese Überlegung führt unter Ausnutzung des Nichtdeterminismus dazu, dass wir bei der Konstruktion von M aus M' für alle Endzustände $q'_{e_i} \in E'$ und sämtliche Kellerzeichen $A \in \Gamma$ spontane Übergänge zu einem neuen Zustand $q_e \in Q$ hinzufügen, d.h. $\delta(q'_{e_i}, \varepsilon, A) \ni (q_e, \varepsilon)$. In q_e wird der (ggf. nichtleere) Keller von M' geleert, d.h. $\delta(q_e, \varepsilon, A) \ni (q_e, \varepsilon)$ für alle $A \in \Gamma$ hinzufügen.

Wenn M' nach vollständigem Lesen des Eingabewortes w seinen Keller leert ohne einen Endzustand einzunehmen, wird w bekanntlich nicht akzeptiert. Bei der Konstruktion von M muss darauf geachtet werden, dass dies nicht etwa zur Akzeptanz von w führt. Man erreicht dies durch ein neues Kellervorbelegungszeichen $k_0 \in \Gamma$. Zu Beginn der Arbeit von M wird es unter das Vorbelegungszeichen $k'_0 \in \Gamma'$ „geschoben“: $\delta(q_0, \varepsilon, k_0) = \{(q'_0, k'_0 k_0)\}$. Hierzu benötigt M einen neuen Anfangszustand q_0 .

Alle anderen Funktionswerte für δ werden von δ' übernommen. Auf diese Weise ergibt sich $M = (Q' \cup \{q_0, q_e\}, \Sigma', \Gamma' \cup \{k_0\}, \delta, q_0, k_0)$, mit $q_0, q_e \notin Q'$ und $k_0 \notin \Gamma'$. \square

$NKA_6 \rightsquigarrow NKA_7$

Beweis, Teil 2

Zu jedem NKA $M = (Q, \Sigma, \Gamma, \delta, q_0, k_0)$ gibt es einen NKA $M' = (Q', \Sigma', \Gamma', \delta', q'_0, k'_0, E')$ mit $L = L(M') = L(M)$.

Wir konstruieren also M' aus M : Wähle $Q' = Q \cup \{q'_0, q'_e\}$, mit $q'_0 \neq q'_e$ und $q'_0, q'_e \notin Q$, $\Sigma' = \Sigma$, $\Gamma' = \Gamma \cup \{k'_0\}$, $E' = \{q'_e\}$. M' wird so konstruiert, dass er M simuliert: $\delta'(q_i, a, A) = \delta(q_i, a, A)$ für alle $q_i \in Q$, $a \in \Sigma$, $A \in \Gamma$.

Für alle Zustände $q_i \in Q$ ergänzen wir $\delta'(q_i, \varepsilon, k'_0) \ni (q'_e, \varepsilon)$, um dafür zu sorgen, dass es genau dann einen spontanen Übergang zum Endzustand q'_e von M' gibt, wenn M mit leerem Keller stoppt. Genau dann ist nämlich k'_0 das oberste Kellerzeichen.

Schließlich müssen wir noch dafür sorgen, dass vor dem Start der Simulation von M dessen Kellervorbelegungszeichen k_0 auf den nur mit k'_0 belegten Keller gelegt wird ohne k'_0 dabei zu entfernen: $\delta'(q'_0, \varepsilon, k'_0) = \{(q_0, k_0 k'_0)\}$. \square

**Beispiel 6.5**

Wir wenden die Transformation aus dem Beweisteil 1 auf M_1 aus Beispiel 6.4 an. Die gesuchte 6-Tupeldefinition lautet: $M'_1 = (\{q_0, q_1, q_2, q_3\}, \{a, b\}, \{0, a, b, x\}, \delta, q_2, x)$ mit

$$\begin{array}{lll}
 \delta(q_2, \varepsilon, x) &= \{(q_0, 0x)\} & \delta(q_1, \varepsilon, 0) \ni (q_3, \varepsilon) \\
 \delta(q_0, a, b) &\ni (q_0, \varepsilon) & \delta(q_1, \varepsilon, a) \ni (q_3, \varepsilon) \\
 \delta(q_0, a, 0) &\ni (q_0, a0) & \delta(q_1, \varepsilon, b) \ni (q_3, \varepsilon) \\
 \delta(q_0, a, a) &\ni (q_0, aa) & \delta(q_1, \varepsilon, x) \ni (q_3, \varepsilon) \\
 \delta(q_0, b, a) &\ni (q_0, \varepsilon) & \delta(q_3, \varepsilon, 0) \ni (q_3, \varepsilon) \\
 \delta(q_0, b, 0) &\ni (q_0, b0) & \delta(q_3, \varepsilon, a) \ni (q_3, \varepsilon) \\
 \delta(q_0, b, b) &\ni (q_0, bb) & \delta(q_3, \varepsilon, b) \ni (q_3, \varepsilon) \\
 \delta(q_0, \varepsilon, 0) &\ni (q_1, \varepsilon) & \delta(q_3, \varepsilon, x) \ni (q_3, \varepsilon)
 \end{array}$$

Für das Wort $bbabaa$ ergibt sich die folgende Konfigurationenfolge:

| Zustand | Eingabezeichen | Keller (links: top of stack) |
|---------|----------------|------------------------------|
| q_2 | bbabaa | x |
| q_0 | bbabaa | 0x |
| q_0 | babaa | b0x |
| q_0 | abaa | bb0x |
| q_0 | baa | b0x |
| q_0 | aa | bb0x |
| q_0 | a | b0x |
| q_0 | ε | 0x |
| q_1 | ε | x |
| q_3 | ε | ε |

Das Eingabewort wird also von M'_1 akzeptiert, denn der Keller ist am Ende des Analyseprozesses leer.

Die im Beweis beschriebene Transformation $NKA_6 \rightsquigarrow NKA_7$ wird in Beispiel 6.6 und ganz am Ende von Beispiel 6.7 angewandt.

Übung 6.2

Zeigen Sie, dass man eine weitere gleichwertige Definition für NKA durch Aufnahme einer Menge von Startzuständen formulieren könnte.



6.3 Äquivalenz von NKA und kontextfreier Grammatik

Bisher haben wir NKA als Beschreibungsmittel für kontextfreie Sprachen (kfs) kennengelernt und beispielbezogen angewandt. Wir wissen aber noch nicht, ob

dies für jede kfS gelingt, oder ob es sogar kontextsensitive Sprache gibt, die mit NKA beschrieben werden können. Der folgende Satz beantwortet diese Fragen.



Satz 6.2

Eine Sprache L ist kontextfrei genau dann, wenn L von einem NKA akzeptiert wird.

$$\mathcal{L}_{kfs} = \mathcal{L}_{NKA}$$

Damit steht fest, dass die Menge der durch NKA akzeptierten Sprachen und die Menge der kfS identisch sind: $\mathcal{L}_{kfs} = \mathcal{L}_{NKA}$.

Vor uns steht der Beweis dieses Satzes. Dieser ist konstruktiv und liefert als Nebenprodukte ein Verfahren zur Erzeugung eines äquivalenten NKA aus einer gegebenen kfG (Teil 1) und eines zur Gewinnung einer äquivalenten kfG aus einem gegebenen NKA (Teil 2). Wir verwenden die alternative NKA-Definition 6.2 als 6-Tupel ohne Endzustandsmenge.

Beweis, Teil 1

Wir konstruieren einen äquivalenten NKA $M = (Q, \Sigma, \Gamma, \delta, q_0, k_0)$ aus einer (beliebig) vorgegebenen kfG $G = (N, T, P, s)$.

Die Idee besteht darin, den Ableitungsprozess (Grammatik) durch die Arbeitsweise des NKA zu simulieren. Hierfür braucht man nur die rechte Seite der jeweils angewandten Regel als Kellerwort auf den Stapel zu legen. Da vereinbarungsgemäß das erste Zeichen des jeweiligen Kellerwortes das neue top of stack ist, wird praktisch eine Linksableitung simuliert. Ist das top of stack ein Nichtterminal, wird eine der passenden Regeln angewandt, d.h. dieses Nichtterminal wird durch eine zugehörige rechte Regelseite (im Keller) ersetzt. Wenn ein Terminalzeichen ganz oben auf dem Stapel liegt, muss es mit dem aktuellen Zeichen des Eingabewortes übereinstimmen. Das Zeichen wird dann einfach vom Keller entfernt. Andernfalls ist dieser Analyseweg für das betrachtete Wort nicht erfolgreich. Da wir einen nichtdeterministischen Kellerautomat verwenden, können für mehrere alternative Ersetzungsregeln für ein Nichtterminal später mehrere Maschinen geklont werden. Es findet also eine Simulatableitung statt. Die Simulation der Ableitung beginnt mit dem Spitzen-symbol der Grammatik als Kellervorbelegungszeichen und besteht ausschließlich aus Stapelarbeit im Zustand q_0 . Der so konstruierte NKA benötigt also nur einen einzigen Zustand. Die weiteren Bestandteile des NKA ergeben sich unmittelbar: $M = (\{q_0\}, T, N \cup T, \delta, q_0, s)$. Für δ gilt:

1. Für jede Regel $A \rightarrow \alpha \in P$, mit $\alpha \in (N \cup T)^*$ und $A \in N$, setze $\delta(q_0, \varepsilon, A) \ni (q_0, \alpha)$.
Mit anderen Worten: Die Substitution eines Nichtterminals erfolgt ohne Verbrauch des aktuellen Zeichens auf dem Eingabeband. Die rechte Regelseite wird gekellert.
2. Für alle Terminals $a \in T$ setze $\delta(q_0, a, a) \ni (q_0, \varepsilon)$.
Das top of stack wird verbrauchend gelesen und muss mit dem aktuellen Zeichen auf dem Eingabeband übereinstimmen.

Linksableitung
mit Keller
simulieren

Für alle Wörter $w \in T^*$ gilt $w \in L(G)$ genau dann, wenn

$$\Leftrightarrow w \in L(M)$$

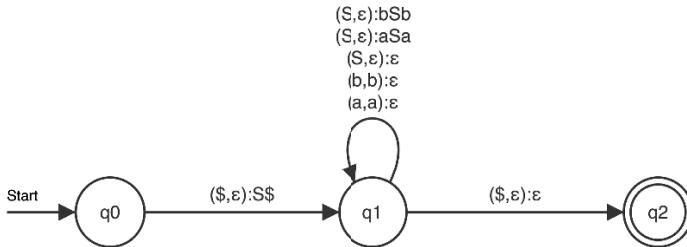
- es eine Ableitung der Form $s \Rightarrow \dots \Rightarrow w$ in G gibt;

- es eine Konfigurationenfolge der Form $(q_0, w, k_0) \xrightarrow{*} (q_0, \varepsilon, \varepsilon)$ gibt, d. h. wenn $w \in L(M)$. Ein formaler Beweis verwendet für diesen letzten Teil vollständige Induktion.

□

Beispiel 6.6

Gegeben sei die kfG $G = (\{S\}, \{a, b\}, \{S \rightarrow \epsilon \mid aSa \mid bSb\}, S)$ für die Sprache der Palindrome über $\{a, b\}$ mit geradzahliger Länge. Der zu G äquivalente NKA $M = (\{q_0, q_1, q_2\}, \{a, b\}, \{a, b, \$, S\}, \delta, q_0, \$, \{q_2\})$ hat folgende Überführungsfunktion.



Es ist zu beachten, dass zur Konstruktion des hier angegebenen NKA nach der 7-Tupel-Definition 6.1 das Verfahren aus dem Beweis, Teil 2, von Satz 6.1 zusätzlich angewandt wurde: kfG \rightsquigarrow NKA(6) \rightsquigarrow NKA(7).

Nun wird die Akzeptanz des Wortes $baab$ nachgewiesen. Da der betrachtete NKA nichtdeterministisch arbeitet, entstehen durch Cloning dieses NKA insgesamt 11 Konfigurationenfolgen von denen nur genau eine (K4) erfolgreich ist. Dafür sind neun Takte notwendig:

| Zustand | Eingabe | Keller | Konfigurationenfolge(n) für: b a a b | | | | | | | | | | |
|---------|------------|------------|--------------------------------------|------|------|--------|-------|--------|--------|-------|------|------|----|
| q_0 | baab | \$ | | | | | | | | | | | |
| q_1 | baab | \$ \$ | K4 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| q_1 | baab | bSb \$ | ▶ | (q0) | (q1) | (q1) | (q1) | (q1) | (q1) | (q1) | (q1) | (q1) | q2 |
| q_1 | aab | Sb \$ | | \$ | SS | bSb \$ | Sb \$ | aSb \$ | Sab \$ | ab \$ | b \$ | \$ | |
| q_1 | aab | aSab \$ | | baab | baab | baab | aab | aab | ab | ab | b | | |
| q_1 | ab | Sab \$ | Konfigurationenfolge(n) für: b a a b | | | | | | | | | | |
| q_1 | ab | ab \$ | K3 | 0 | 1 | 2 | 3 | 4 | | | | | |
| q_1 | b | b \$ | ▶ | (q0) | (q1) | (q1) | (q1) | (q1) | | | | | |
| q_1 | ϵ | \$ | | \$ | SS | bSb \$ | Sb \$ | b \$ | | | | | |
| q_2 | ϵ | ϵ | | baab | baab | baab | aab | aab | | | | | |

Computerübung 6.3

Verwenden Sie FLACI, um das Beispiel 6.6 zu bearbeiten. Beginnen Sie bei der Definition der angegebenen kfG und führen Sie auch die Konvertierung in einen NKA mit FLACI aus. Erzeugen Sie die 11 Konfigurationenfolgen und geben Sie K3 (erfolglos) als Tabelle an.

**Computerübung 6.4**

Geben Sie für die Sprache L_A der korrekt geklammerten arithmetischen Ausdrücke einen äquivalenten NKA an und simulieren Sie dessen Arbeitsweise für einige korrekte und fehlerhafte Eingabewörter.

L_A sei durch folgende Grammatik $G = (\{E, T, F\}, \{a, +, *, (,)\}, P, E)$ definiert, mit $P = \{E \rightarrow T \mid E+T, T \rightarrow F \mid T*T, F \rightarrow a \mid (E)\}$.

**Beweis, Teil 2**

Wir konstruieren eine äquivalente kfG $G = (N, T, P, s)$ aus einem gegebenen NKA

$\text{NKA} \rightsquigarrow \text{kfG}$

$M = (Q, \Sigma, \Gamma, \delta, q_0, k_0)$. Auch hier verwenden wir die 6-Tupel-Definition.

Welcher Zusammenhang besteht zwischen den beiden verschiedenen „Welten“: Grammatik und Automat? Ein Wort v kann genau dann von X abgeleitet¹ werden, d.h. $X \xrightarrow{*} v$, wenn gilt $(\underline{q}_i, v, \underline{k}_r) \xleftarrow{*} (\underline{q}_j, \varepsilon, \varepsilon)$. Dabei spielt es keine Rolle, wie viele Schritte die zugehörige Konfigurationsfolge umfasst. Doch wo steckt das X ? Nichtterminale (Grammatikwelt) müssen aus den Konfigurationsübergängen (Automatenwelt) gewonnen werden. Sie werden aus genau drei Informationen gespeist:

- dem alten Zustand q_i ,
- dem obersten Kellerzeichen (top of stack) k_r und
- dem neuen Zustand q_j .

Notation für
Nichtterminale
 $[q_i, k_r, q_j]$

Von daher ist es sinnvoll, diese Bestandteile im Namen der zu bildenden Nichtterminale zu verankern. Wir schreiben die entsprechenden Tripel aus $Q \times \Gamma \times Q$ in eckigen Klammern, um eben diese *Nichtterminal-Namen*² von den Konfigurationen des Automaten abzugrenzen. Ein Wort w gehört zu $L(M)$ genau dann, wenn $(q_0, w, k_0) \xleftarrow{*} (q_i, \varepsilon, \varepsilon)$. In der Grammatik wird dies durch die Regel $[q_0, k_0, q_i] \xrightarrow{*} w$ ausgedrückt. Werfen wir nun einen Blick auf die Überführungsfunktion eines NKA. Die allgemeine Gestalt von δ ist

$$\delta(q, a, A) \ni (q', B_1 \dots B_k), \text{ mit } k \geq 0 \text{ und } a \in \Sigma \text{ oder } a = \varepsilon. \quad (6.1)$$

Hier und im Folgenden gelten $A, B, B_i \in \Gamma$.

Normierung:
 $k = 0, 1, 2$

Wir können uns nicht allen Fällen mit $k \geq 0$ zuwenden und nehmen deshalb eine *Normierung* vor, so dass $0 \leq k \leq 2$ gilt. Hierfür zeigen wir, dass jede Gestalt von δ in die zugehörige normierte Form transformiert werden kann. Zur Umformung werden weitere Zustände, q_1, q_2, \dots, q_{k-2} , eingeführt. Aus (6.1) wird eine Folge von „ δ -Regeln“:

$$\begin{aligned} \delta(q, a, A) &\ni (q_1, B_{k-1} B_k) \\ \delta(q_1, \varepsilon, B_{k-1}) &\ni (q_2, B_{k-2} B_{k-1}) \\ \delta(q_2, \varepsilon, B_{k-2}) &\ni (q_3, B_{k-3} B_{k-2}) \\ &\vdots \\ \delta(q_{k-2}, \varepsilon, B_2) &\ni (q', B_1 B_2) \end{aligned}$$

Aufgrund der vorgenommenen Normierung für δ können bezüglich der Konfigurationsübergänge $(q, a\beta, A\gamma) \xleftarrow{*} (q', \beta, \gamma)$, mit $a \in \Sigma, \beta \in \Sigma^*, A \in \Gamma$ und $\gamma \in \Gamma^*$, nur die folgenden drei Fälle auftreten, nämlich für $k = 0$ (1. Fall), $k = 1$ (2. Fall) und $k = 2$ (3. Fall).

1. Fall: *reines Entkellern* bei $\delta(q, a, A) \ni (q', \varepsilon)$

| Zustand | Eingabe | Keller |
|---------|----------|-----------|
| q | $a\beta$ | $A\gamma$ |
| q' | β | γ |

¹Dabei ist X nicht notwendigerweise das Spitzensymbol der Grammatik.

²Darstellungen, wie $[q_i, k_r, q_j]$, repräsentieren also nichts anderes als ein Nichtterminal. Mag sein, dass die Namensgebung etwas merkwürdig ausschaut. Auf keinen Fall darf man beim Gebrauch dieser Nichtterminale in der zu konstruierenden Grammatik die Namenselemente „auseinandernehmen“. Es handelt sich also um atomare Gebilde, wie A, B oder X .

In der „Welt der Grammatiken“ bedeutet das

$$[q, A, q'] \rightarrow a.$$

2. Fall: *genau ein Kellerzeichen* bei $\delta(q, a, A) \ni (q_1, B)$

| Zustand | Eingabe | Keller |
|----------|----------|-----------|
| q | $a\beta$ | $A\gamma$ |
| q_1 | | $B\gamma$ |
| \vdots | | |
| q' | β | γ |

In der „Welt der Grammatiken“ bedeutet das

$$[q, A, q'] \rightarrow a[q_1, B, q'].$$

$[q, A, q']$ und $[q_1, B, q']$ sind Nichtterminale in G für alle $q' \in Q$.

3. Fall: Kellerwort der Länge 2 bei $\delta(q, a, A) \ni (q_1, BC)$

| Zustand | Eingabe | Keller |
|----------|----------|------------|
| q | $a\beta$ | $A\gamma$ |
| q_1 | | $BC\gamma$ |
| \vdots | | |
| q_2 | | $C\gamma$ |
| \vdots | | |
| q' | β | γ |

In der „Welt der Grammatiken“ bedeutet das

$$[q, A, q'] \rightarrow a[q_1, B, q_2][q_2, C, q'].$$

$[q, A, q'], [q_1, B, q_2]$ und $[q_2, C, q']$ sind Nichtterminale in G für alle $q', q_2 \in Q$.

Diese intuitiven Vorüberlegungen genügen, um sich klar zu machen, wie man die Menge der Produktionen von G bestimmt. Für $a \in (\Sigma \cup \{\epsilon\})$ ergibt sich

$$\begin{aligned} P &= \{s \rightarrow [q_0, k_0, q] \mid \text{für alle } q \in Q\} \\ &\cup \{[q, A, q'] \rightarrow a \mid \delta(q, a, A) \ni (q', \epsilon)\} \\ &\cup \{[q, A, q'] \rightarrow a[q_1, B, q'] \mid \delta(q, a, A) \ni (q_1, B), \text{ für alle } q' \in Q\} \\ &\cup \{[q, A, q'] \rightarrow a[q_1, B, q_2][q_2, C, q'] \mid \delta(q, a, A) \ni (q_1, BC), \text{ für alle } q', q_2 \in Q\} \end{aligned}$$

Mit vollständiger Induktion kann man nun für jede Zeichenkette $w \in \Sigma^*$ nachweisen, dass $w \in L(M) \Leftrightarrow w \in L(G)$ und damit $L(M) = L(G)$ gelten. \square

Beispiel 6.7

Wir wenden das Konstruktionsverfahren aus diesem Beweis, Teil 2, s. Seite 149ff, zur Erzeugung einer äquivalenten kfG G aus dem NKA $M = (\{q_0, q_1\}, \{0, 1\}, \{X, Y\}, \delta, q_0, Y)$



mit folgendem δ an.

$$\begin{aligned}\delta(q_0, 0, Y) &\ni (q_0, XY) \\ \delta(q_0, 0, X) &\ni (q_0, XX) \\ \delta(q_0, 1, X) &\ni (q_1, \varepsilon) \\ \delta(q_1, 1, X) &\ni (q_1, \varepsilon) \\ \delta(q_1, \varepsilon, X) &\ni (q_1, \varepsilon) \\ \delta(q_1, \varepsilon, Y) &\ni (q_1, \varepsilon)\end{aligned}$$

Da in der Überführungsfunktion δ ausschließlich Kellerwörter vorkommen, deren Längen höchstens 2 betragen, liegt die gewünschte Normierung bereits vor. Wir können also gleich mit der eigentlichen Transformation beginnen. Als erstes ergeben sich die Produktionen für Spitzensymbol

$$s \rightarrow [q_0, Y, q_0] \mid [q_0, Y, q_1].$$

Nun folgen die Konstruktionen nach den 3 betrachteten Fällen. Für Fall 1 erhalten wir:

$$\begin{aligned}[q_0, X, q_1] &\rightarrow 1 \\ [q_1, X, q_1] &\rightarrow 1 \\ [q_1, X, q_1] &\rightarrow \varepsilon \\ [q_1, Y, q_1] &\rightarrow \varepsilon\end{aligned}$$

Da in δ kein einziger Funktionswert ein Kellerwort der Länge 1 enthält, entfällt Fall 2. Im Fall 3 stellen wir (zuerst für $\delta(q_0, 0, Y) \ni (q_0, XY)$) das für alle Zustandskombinationen auszufüllende Muster voran:

$$[q_0, Y, \square] \rightarrow 0[q_0, X, \diamondsuit][\diamondsuit, Y, \square]$$

Die darin enthaltenen Symbole \diamondsuit und \square sind so durch Zustände zu belegen, dass gleiche Symbole gleiche Zustände repräsentieren. Andererseits ist jedoch nicht ausgeschlossen, dass verschieden symbolisierte Zustandsbelegungen übereinstimmen.

$$\begin{aligned}[q_0, Y, q_0] &\rightarrow 0[q_0, X, q_0][q_0, Y, q_0] \\ [q_0, Y, q_0] &\rightarrow 0[q_0, X, q_1][q_1, Y, q_0] \\ [q_0, Y, q_1] &\rightarrow 0[q_0, X, q_0][q_0, Y, q_1] \\ [q_0, Y, q_1] &\rightarrow 0[q_0, X, q_1][q_1, Y, q_1]\end{aligned}$$

Analog gehen wir nun für $\delta(q_0, 0, X) \ni (q_0, XX)$ vor.

$$\begin{aligned}\underline{[q_0, X, \square]} &\rightarrow \underline{0[q_0, X, \diamondsuit][\diamondsuit, X, \square]} \\ [q_0, X, q_0] &\rightarrow 0[q_0, X, q_0][q_0, X, q_0] \\ [q_0, X, q_0] &\rightarrow 0[q_0, X, q_1][q_1, X, q_0] \\ [q_0, X, q_1] &\rightarrow 0[q_0, X, q_0][q_0, X, q_1] \\ [q_0, X, q_1] &\rightarrow 0[q_0, X, q_1][q_1, X, q_1]\end{aligned}$$

Wir erhalten 14 Regeln. Diese Regelmenge kann allerdings noch deutlich optimiert werden: Als erstes entfernen wir jede Regel, auf deren rechter Seite (mindestens) ein Nichtterminal

vorkommt, für das es keine Regel gibt. Von den verbleibenden Produktionen werden die rekursiven Regeln, wie $X \rightarrow \alpha X \beta$, für solche Nichtterminale gestrichen, für die es *auschließlich* rekursive Regeln gibt. Daraufhin muss der erste Optimierungsschritt ggf. wiederholt werden.

Im Beispiel bleiben nur noch drei Nichtterminale übrig. Wir benennen sie um, damit sich besser lesbare Namen ergeben: $[q_0, X, q_1] := A$, $[q_1, X, q_1] := B$, und $[q_0, Y, q_1] := C$.

Schließlich erhalten wir folgende Grammatik $G = (\{A, B, C\}, \{0, 1\}, P, C)$ mit $P = \{A \rightarrow 1 | 0AB, B \rightarrow 1 | \epsilon, C \rightarrow 0A\}$. Je nach Wunsch kann G nun in eine ϵ -freie kfG transformiert werden. Offensichtlich wird mit G die Sprache $L = L(G) = \{0^n 1^m | n \geq 1, 1 \leq m \leq n\}$ beschrieben. Einer gewissen Anzahl von Nullen (mindestens einer Null) folgen höchstens ebenso viele Einsen.

Übung 6.3

Arbeiten Sie das Beispiel 6.7 komplett und gewissenhaft durch.



Computerübung 6.5

Wenden Sie die in FLACI eingebaute automatische Transformation von NKA (7-Tupel-Definition) in äquivalente kfG auf den in Beispiel 6.7 angegebenen NKA an. Führen Sie im Anschluss entsprechende Optimierungen durch: Umbenennung der Nichtterminale, Streichen unnötiger Regeln. Vergleichen Sie die auf diese Weise erzeugte Grammatik mit der aus Beispiel 6.7.



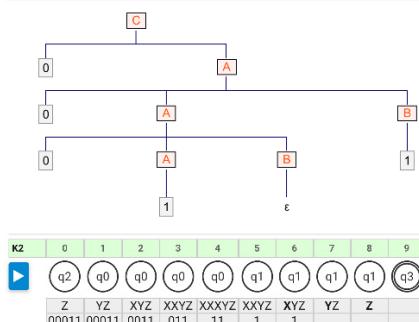
Computerübung 6.6

Verwenden Sie FLACI, um mit Bezug auf Beispiel 6.7 die Anwendung von M auf das Eingabewort 00011 zu simulieren. Beginnen Sie mit der folgenden 7-Tupel-Definition für den betrachteten NKA $M = (\{q_0, q_1, q_2, q_3\}, \{0, 1\}, \{X, Y, Z\}, \delta, q_2, Z, \{q_3\})$ mit $\delta(q_2, \epsilon, Z) = \{(q_0, YZ)\}$, $\delta(q_0, 0, Y) \ni (q_0, XY)$, $\delta(q_0, 0, X) \ni (q_0, XX)$, $\delta(q_0, 1, X) \ni (q_1, \epsilon)$, $\delta(q_1, 1, X) \ni (q_1, \epsilon)$, $\delta(q_1, \epsilon, X) \ni (q_1, \epsilon)$, $\delta(q_1, \epsilon, Y) \ni (q_1, \epsilon)$, $\delta(q_0, \epsilon, Z) \ni (q_3, \epsilon)$, $\delta(q_1, \epsilon, Z) \ni (q_3, \epsilon)$.

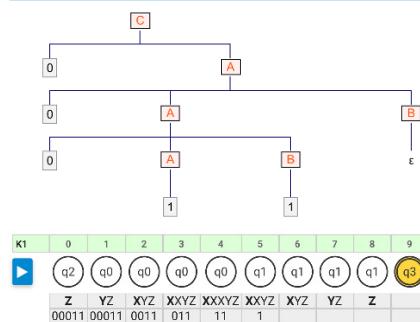
Lassen Sie FLACI daraus eine äquivalente kfG erzeugen und benennen Sie die Nichtterminale anschließend so um³, dass die in Beispiel 6.7 angegebene Form entsteht. Für das Eingabewort 00011 ergeben sich zwei Ableitungsbäume, die den beiden möglichen Konfigurationenfolgen der NKA-Simulation entsprechen.



Ableitungsbau 1 von 2:



Ableitungsbau 2 von 2:



³Dafür gibt es in FLACI einen Knopf.

6.4 Parsing kontextfreier Sprachen

Wir wissen, dass kfS mit kfG (CHOMSKY-Typ 2) und NKA beschrieben werden können. Für praktische Anwendungen sind Parser auf der Basis von NKA jedoch unbrauchbar, da sie im worst case mit exponentiellem Aufwand⁴ arbeiten.

**Effizientes
kfS-Parsing**

Es gibt zwei verschiedene Wege, diesem Problem zu begegnen:

1. Suche nach *effizienten* Parsing-Verfahren für alle kfS und
2. Definition spezieller Untermengen der Menge aller kfS, für die es sehr effiziente Parsing-Verfahren gibt.

allgemeine kfS

Für beide Arbeitsrichtungen gibt es interessante Resultate. Während wir uns weiter unten mit Sprachfamilien befassen werden, die in der Menge aller kfS echt enthalten sind, beleuchten wir hier Variante 1. D.h., wir fragen nach effizienten Parsing-Verfahren für ganz *allgemeine* kfS, also auch für mehrdeutige Grammatiken, die sogar ϵ - und (in)direkte linksrekursive Regeln besitzen können.

Packrat-Parser

Im Übrigen hat die aufgeworfene Fragestellung inzwischen eine historische Dimension: Insbesondere (Computer-)Linguisten haben sich schon in den 1950er Jahren damit beschäftigt. Sie haben es naturgemäß mit eher „unbequemen“ Sprachen zu tun, da sie eine gewisse Nähe zu den natürlichen Sprachen anstreben.

**Chart-Parser
dynamisches
Programmieren**

Der Funktionsweise eines rekursiv absteigenden Parsers (recursive descent parser) ähnlich, ist ein *Packrat-Parser* (BRYAN FORD, 2002) ein spezieller „sammelwütiger“ Parser, der während des Parsing-Prozesses die Zwischenergebnisse aller rekursiven Aufrufe behält und damit *viele* kontextfreie Grammatiken sowie alle sogenannten PEG (parsing expression grammar) in linearer Zeit parst.

**Chart-Parser
dynamisches
Programmieren**

Mit der Erfindung sog. *Chart-Parser* für allgemeine kfG durch MARTIN KAY (1996) gelingt ein Effizienzgewinn durch dynamisches Programmieren⁵. Grob gesagt, wird dabei das eigentliche Problem dadurch gelöst, dass (sämtliche) gleichartige Teilprobleme (beginnend mit den kleinsten) gelöst werden, von denen oft nur einige benötigt werden, um aus deren Lösung die Lösung des Ausgangsproblems zu gewinnen. Meist kann man die Teilprobleme mit den zugehörigen Lösungen in Form einer $n \times m$ -Tabelle darstellen.

Zum Ausfüllen der gesamten Tabelle (mit den Einzellösungen) benötigt der Berechnungsprozess eine Zeit in $\mathcal{O}(nm)$. Braucht es zur Berechnung jedes dieser Tabelleneinträge beispielsweise einen linearen Zeitaufwand (in $\mathcal{O}(p)$), so ergibt sich insgesamt ein kubischer Aufwand: $\mathcal{O}(nm) \cdot \mathcal{O}(p) = \mathcal{O}(nmp) = \mathcal{O}(r^3)$, mit $r = \max(m, n, p)$. Charts (Tabellen) bezeichnen hier eine geeignete Datenstruktur,

⁴Zur Angabe des zeitlichen Berechnungsaufwandes kann eine exponentielle Funktion T der Form $y = T(n) = k^n$ ($k > 1$, n ist die Problemgröße, hier die Länge des Eingabewortes) in gewissem Sinn nicht „unterboten“ werden. Man sagt: „ T liegt in $\mathcal{O}(k^n)$.“

⁵Dynamisches Programmieren (eigentlich Optimieren) ist ein fundamentales Entwurfsmuster für Algorithmen und wird nicht nur für Parsing-Prozesse verwendet.

nicht etwa eine Parsing-Strategie. Die Parsing-Strategie kann gewissermaßen „unter der Haube“ in gewissen Grenzen frei gewählt werden. Ein solches Vorgehen wurde schon viel früher (1970) von dem Psychologen JAY EARLEY vorgeschlagen.

Ein *Earley-Parser* arbeitet mit einem Aufwand in $\mathcal{O}(n^3)$, wobei n die Länge des Eingabewortes ist. Der Quelltext des Linux Kernels ist mehrere Millionen Zeilen lang. Aber selbst bei kleineren Projekten mit wenigen hundert Zeilen Code ergeben sich schnell tausende Token im Tokenstrom und damit die Länge n . Ein exponentieller Aufwand in $\mathcal{O}(k^n), k \geq 1$ ist damit bereits für den leistungsfähigsten Computer ein praktisch unlösbares Problem. Ein Aufwand von $\mathcal{O}(n^3)$ ist im Vergleich dazu mit wachsendem n von großem Vorteil. Für eindeutige Grammatiken liegt der Zeitaufwand sogar in $\mathcal{O}(n^2)$ und für eine in Kapitel 8 behandelte Klasse der $LR(k)$ -Sprachen ist er linear, liegt also in $\mathcal{O}(n)$.

Ein Algorithmus von *Harrison*, der sich am CYK-Algorithmus orientiert und versucht den Algorithmus von Earley zu verbessern, löst das Wortproblem für kfS im Allgemeinen auch in kubischer Laufzeit.

Der *CYK*-Algorithmus (COOK, YOUNGER, KASAMI, 1965) erwartet, dass die kfG in CNF (Chomsky-Normalform, s. Kapitel 8) vorliegt. Eine solche Grammatiktransformation, die mit Veränderungen der Regelmenge einher geht, kann einem didaktischen Kontext, wie mit FLACI, entgegen stehen: Durch die Umformung der Regelmenge verwenden die Ableitungen nicht mehr genau die ursprünglichen Nichtterminale, was das Nachvollziehen des Ableitungsprozesses deutlich erschwert.

GLR-Parser (Generalized LR-Parser) sind eine Erweiterung der LR-Parser (s. Kapitel 8), die Nichtdeterminismus verarbeiten können, indem mehrere Threads erzeugt werden, die parallel arbeiten und eine Breitensuche ausführen.

Von aktuellem Forschungsinteresse sind Generalized LL-Parser (GLL-Parser) (ELIZABETH SCOTT and ADRIAN JOHNSTONE, 2010). Dieser Parsing-Typ basiert auf der Technik des rekursiven Abstiegs (wie bei LL(1)-Parseern, s. Kapitel 8) und ermöglicht es, alle kfG (inkl. Linksrekursion) im worst case mit kubischem Aufwand zu parsen. Für LL(1)-Sprachen geschieht das mit linearem Aufwand.

Wir stellen fest, dass wir in diesem Abschnitt auf Parsing-Verfahren für allgemeine kfS hingewiesen haben, die in Originalform der jeweils gegebenen kfG oder nach spezieller Umformung (falls möglich) effizient arbeiten. Die dabei angesprochenen Untermengen kontextfreier Sprachen (z.B. $LR(k)$, $LL(1)$), für die die zugehörigen Verfahren besonders effizient sind, werden wir im in Folgekapiteln genauer betrachten.

Jay Earley

Earley-Parser

Harrison

CYK

GLR-Parser

GLL-Parser

6.5 Deterministischer Kellerautomat (DKA)

In Analogie zu endlichen Automaten gibt es auch bei Kellerautomaten den deterministische Typ, also den deterministischen Kellerautomaten, kurz: DKA. Spricht man nur vom Kellerautomaten schlechthin, so ist der NKA gemeint. Beim Vergleich der Beschreibungsmächtigkeiten der deterministischen und nichtdeterministischen Form beider Automatentypen (EA und KA) werden wir auch sehen, dass diese Analogie ihre Grenzen hat.



Definition 6.3

Ein DKA ist wie ein NKA definiert. Allerdings gibt es drei Abweichungen:

1. $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \rightarrow Q \times \Gamma^*$, anstelle von $\wp_{\text{endlich}}(Q \times \Gamma^*)$ auf der rechten Seite bei NKA. Die Funktionswerte sind also Paare und keine Mengen.
2. Wenn $\delta(q, \varepsilon, A)$ definiert ist, dann ist für alle $a \in \Sigma$ der Funktionswert $\delta(q, a, A)$ undefiniert oder umgekehrt.
3. Es gibt eine Menge von Endzuständen $E \subseteq Q$, d.h. es wird immer die NKA-Definitionsform als 7-Tupel $M = (Q, \Sigma, \Gamma, \delta, q_0, k_0, E)$ zugrunde gelegt. Die bei NKA zulässige 6-Tupel-Definition darf für DKA nicht verwendet werden.

Der in Abbildung 6.4 links angegebene Graph-Ausschnitt zeigt einen Verstoß gegen Punkt 1 der Definition 6.3, der auf der rechten Seite verstößt gegen Punkt 2. Das Verbot solcher Konstellationen ist notwendig, um eine deterministische Arbeitsweise zu garantieren.

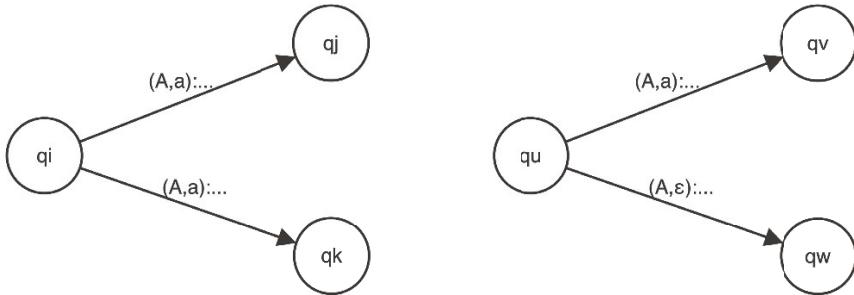


Abbildung 6.4: Verstöße gegen die DKA-Definition 6.3

Die von einem DKA akzeptierte Sprache ist

$$L(M) = \{w \in \Sigma^* \mid (q_0, w, k_0) \xrightarrow{*} (q_e, \varepsilon, K), K \in \Gamma^*, q_e \in E\}.$$

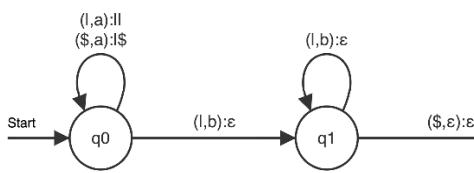
Ein DKA stoppt, wenn das Eingabewort vollständig abgetastet wurde. Ist der dann eingenommene Zustand ein Endzustand, wird das Wort akzeptiert. Ist es kein End-

zustand, so ist zu prüfen, ob ggf. vorhandene spontane Zustandsübergänge zu einem Endzustand führen. Diese werden ausgeführt, sodass sich für das Eingabewort eine Akzeptanzkonfiguration ergibt.

Beispiel 6.8

Für die kfs $L = \{a^n b^n \mid n \geq 1\}$ kann ein DKA wie folgt angegeben werden.

$M_{DKA} = (\{q_0, q_1, q_2\}, \{\text{a, b}\}, \{|\$, |\$\}, \delta, q_0, \$, \{q_2\})$ mit folgendem δ :



$$\begin{aligned}\delta(q_0, a, \$) &= (q_0, |\$) \\ \delta(q_0, a, |) &= (q_0, ||) \\ \delta(q_0, b, |) &= (q_1, \varepsilon) \\ \delta(q_1, b, |) &= (q_1, \varepsilon) \\ \delta(q_1, \varepsilon, \$) &= (q_2, \$)\end{aligned}$$

Dem Entwurf des Automatenmodells liegt folgende Überlegung zugrunde: Beginnt das Eingabewort mit b, wird es abgewiesen.

Für jedes eintreffende a wird ein Merkstrich gesetzt. Das erste eingelesene b führt zum Zustandswechsel, sodass eventuell folgende a's die Akzeptanz des Wortes verhindern. Jedes b trägt einen Merkstrich ab. Der ganz am Ende der Analyse erforderliche spontane Übergang findet nur statt, wenn das oberste Kellerzeichen \$ ist, d.h. es wurden genauso viele b's gelesen, wie es Merkstriche (also a's) im Keller gibt.

Die mit FLACI durchgeführte Simulation der Akzeptanz des Eingabewortes aabb ist erfolgreich, s. Abbildung 6.5.

| | Zustand | Eingabe | Keller |
|---|---------|---------------|--------|
| Konfigurationenfolge(n) für: a a b b | | | |
| K1 | q_0 | aabb | \$ |
| | q_0 | abb | \$ |
| | q_0 | bb | \$ |
| | q_1 | b | \$ |
| | q_1 | ε | \$ |
| | q_2 | ε | \$ |

$(q_0, aabb, \$) \vdash (q_0, abb, |\$) \vdash (q_0, bb, ||\$) \vdash (q_1, b, |\$) \vdash (q_1, \varepsilon, \$) \vdash (q_2, \varepsilon, \$)$ ist die zugehörige Konfigurationenfolge.

Abbildung 6.5: DKA: Simulation der Akzeptanz des Eingabewortes aabb

Wie Beispiel 6.8 unterstreicht, zeigt der zugehörige DKA deterministisches Verhalten: In jedem Arbeitstakt sind der Folgezustand und das Kellerwort eindeutig bestimmt. Es gibt *kein* Cloning, sondern nur genau eine Maschine, die so konstruiert ist, dass sie für jedes Eingabewort mit dessen Akzeptanz oder Ablehnung stoppt.

Computerübung 6.7

Verwenden Sie FLACI, um Beispiel 6.8 nachzuvollziehen. Erschließen Sie, wie der DKA



dafür sorgt, dass aab und abb abgewiesen werden.

Beispiel 6.9



Auch für die kfS $L = \{w \mid w \in \{a, b\}^* \text{ und } \text{Anzahl}(a) = \text{Anzahl}(b)\}$ kann ein DKA M mit $L = L(M)$ angegeben werden. Die Lösung ist aber nicht so naheliegend, dass man sie aus der in Beispiel 6.8 kopieren könnte. Da die a 's und b 's keiner vorgegebenen Anordnungs-vorschrift unterliegen, versagt die Strichlistentechnik aus Beispiel 6.8, wenn es in w einen b -lastigen Präfix gibt. Das „Abstreichen“ von b 's findet für gekellerte a 's und von a 's für gekellerte b 's statt. Falls nicht vorhanden muss gekellert werden. Abbildung 6.6 zeigt einen ersten Vorschlag, der sich jedoch rasch als NKA entpuppt.

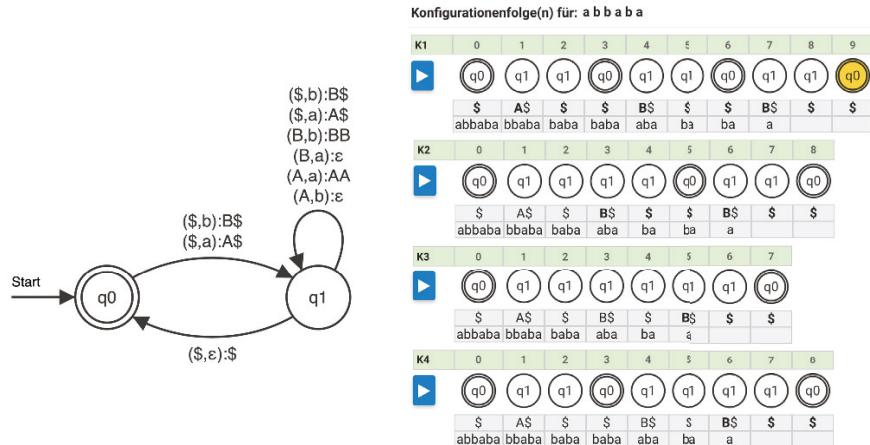


Abbildung 6.6: NKA für L aus Beispiel 6.9

Die beiden Übergänge $(\$, a) : A\$$ und $(\$, b) : B\$$, die den NKA im Zustand q_1 verharren lassen, finden auch beim Übergang von q_0 nach q_1 Anwendung. Unter Verwendung des spontanen Übergangs aus q_1 in den Endzustand q_0 können die genannten Übergänge in q_1 entfallen. Auf diese Weise ergibt sich ein DKA für L , s. Abbildung 6.7.

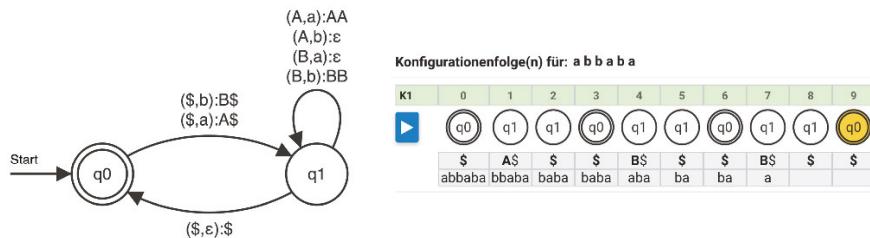


Abbildung 6.7: DKA für L aus Beispiel 6.9

6.6 Deterministisch kontextfreie Sprachen

Nachdem wir in Beispiel 6.9 sowohl einen NKA als auch einen äquivalenten DKA für die dort betrachtete Sprache angeben konnten, drängt sich die Frage nach der Beschreibungsmächtigkeit von DKA förmlich auf. Erinnern wir uns an die Situation bei endlichen Automaten:

$$\mathcal{L}_{DEA} = \mathcal{L}_{NEA}$$

$$\mathcal{L}_{DEA} = \mathcal{L}_{NEA}.$$

DEAs und NEAs sind gleichermaßen leistungsfähig im Hinblick auf die Beschreibungskraft regulärer Sprachen. Für DKA und NKA gilt dies jedoch *nicht*. Wir werden rasch feststellen, dass

$$\mathcal{L}_{DKA} \subseteq \mathcal{L}_{NKA}.$$

Daraus folgt, dass es im Unterschied zu endlichen Automaten kein Verfahren gibt, das beliebige NKA in zugehörige DKA transformiert.

Für manche kfS kann sehr wohl ein NKA, jedoch kein entsprechender DKA angegeben werden. In Beispiel 6.10 wird eine solche Sprache betrachtet, für die sich kein DKA angeben lässt. Sie unterscheidet sich nur geringfügig von der Sprache in Beispiel 6.11, für die es einen zugehörigen DKA gibt.

Beispiel 6.10

Für die Sprache der Wortpalindrome



$$L = \{w \mid w \in \{\text{a}, \text{b}\}^* \text{ und } w = \text{umkehr}(w)\}$$

gibt es keinen DKA, der genau L akzeptiert. Ein DKA verfügt über keine Mittel, um festzustellen, ob die Wortmitte erreicht ist. Genau dies ist aber die Voraussetzung, um den Vorgang des Kellerns zu beenden und auf Entkellern umzuschalten.

Ein NKA für L ist $M = (\{q_0, q_1, q_2\}, \{\text{a}, \text{b}\}, \{\$\}, \delta, q_0, \$, \{q_2\})$ mit folgender Übergangsfunktion δ .

$$\begin{array}{llll} \delta(q_0, a, \$) & = & \{(q_0, a\$), (q_1, \$)\} & \delta(q_0, \epsilon, \$) = \{(q_1, \$)\} \\ \delta(q_0, b, \$) & = & \{(q_0, b\$), (q_1, \$)\} & \delta(q_0, \epsilon, a) = \{(q_1, a)\} \\ \delta(q_0, a, a) & = & \{(q_0, aa), (q_1, a)\} & \delta(q_0, \epsilon, b) = \{(q_1, b)\} \\ \delta(q_0, b, a) & = & \{(q_0, ba), (q_1, a)\} & \delta(q_1, a, a) = \{(q_1, \epsilon)\} \\ \delta(q_0, a, b) & = & \{(q_0, ab), (q_1, b)\} & \delta(q_1, b, b) = \{(q_1, \epsilon)\} \\ \delta(q_0, b, b) & = & \{(q_0, bb), (q_1, b)\} & \delta(q_1, \epsilon, \$) = \{(q_2, \$)\} \end{array}$$

Computerübung 6.8

Bearbeiten Sie den NKA aus Beispiel 6.10 mit FLACI und führen Sie entsprechende Simulationen durch. Zum Vergleich geben wir den Übergangsgraph für den NKA aus Beispiel 6.10 an:



FLACI

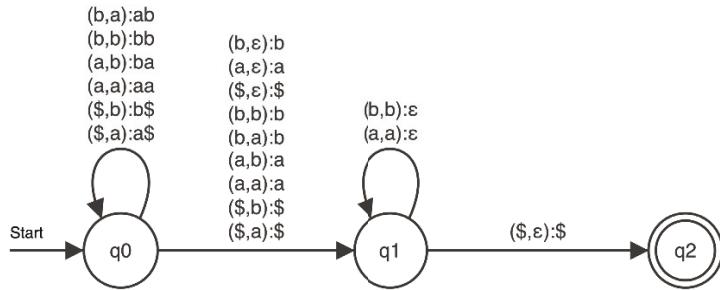


Abbildung 6.8: Graph der Überführungsfunktion des NKA aus Beispiel 6.10

Beispiel 6.11

Beispiel

Wir betrachten die Sprache der Palindrome mit markierter Wortmitte:

$$L = \{w \in \{a, b, c\}^* \mid w = vcv' \text{ und } v' = \text{umkehr}(v), \text{ mit } v \in \{a, b\}^*\}.$$

$M = (\{q_0, q_1, q_2\}, \{a, b, c\}, \{\$\}, \delta, q_0, \$, \{q_2\})$ mit

$$\begin{array}{lll} \delta(q_0, a, \$) & = & (q_0, a\$) \\ \delta(q_0, a, a) & = & (q_0, aa) \\ \delta(q_0, a, b) & = & (q_0, ab) \\ \delta(q_0, b, \$) & = & (q_0, b\$) \\ \delta(q_0, b, a) & = & (q_0, ba) \\ \delta(q_0, b, b) & = & (q_0, bb) \\ & & \\ \delta(q_0, c, \$) & = & (q_1, \$) \\ \delta(q_0, c, a) & = & (q_1, a) \\ \delta(q_0, c, b) & = & (q_1, b) \\ \delta(q_1, a, a) & = & (q_1, \epsilon) \\ \delta(q_1, b, b) & = & (q_1, \epsilon) \\ \delta(q_1, \epsilon, \$) & = & (q_2, \epsilon) \end{array}$$

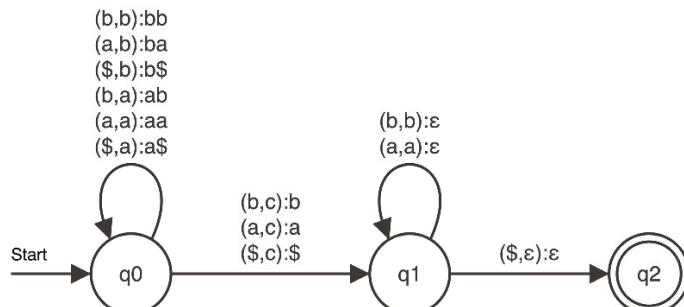
ist ein DKA für L . Ergänzend geben wir die Überführungsfunktion grafisch an:

Abbildung 6.9: Graph der Überführungsfunktion des DKA aus Beispiel 6.11

Computerübung 6.9

Bearbeiten Sie den DKA aus Beispiel 6.11 mit FLACI und führen Sie entsprechende Simulationen durch.



Die Sprachen, die durch DKA beschrieben werden können, heißen *deterministisch kontextfreie Sprachen*, kurz: dkfS. Sie bilden die für die Programmiersprachen wichtigste Sprachklasse und stimmen mit den sog. LR(k)-Sprachen überein, s. Abbildung 6.10. Diese werden im Gebiet der Sprachübersetzer (Compilerbau) ausführlich thematisiert.

dkfS

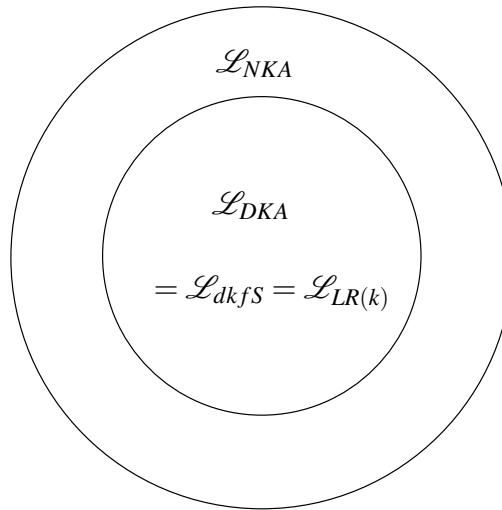


Abbildung 6.10: Darstellung relevanter Sprachklassen: kfS

6.7 Parsergeneratoren für dkfS

An dieser Stelle greifen wir unsere Zeichenrobotersprache aus Abbildung 5.17 wieder auf.

Es ist klar, dass NKA zur Beschreibung von kfS, die den Kern von Programmiersprachen bilden, ungeeignet sind. Schon bei kleineren Programmen würde die Analyse viel zu viel Zeit beanspruchen. Cloning und damit Backtracking sind sehr zeitaufwendige Strategien.

Computerübung 6.10

Konvertieren Sie einen NKA aus der kfG für die Sprache ZR aus Abbildung 5.17. Simulieren Sie die Syntaxanalyse für das Eingabewort WH 4 [VV 100 RE 90] mit diesem NKA.



Das Cloning wird bei 50 Maschinen abgebrochen und das Wort daraufhin verworfen. Dies ist jedoch falsch, denn das Eingabewort ist syntaktisch korrekt.

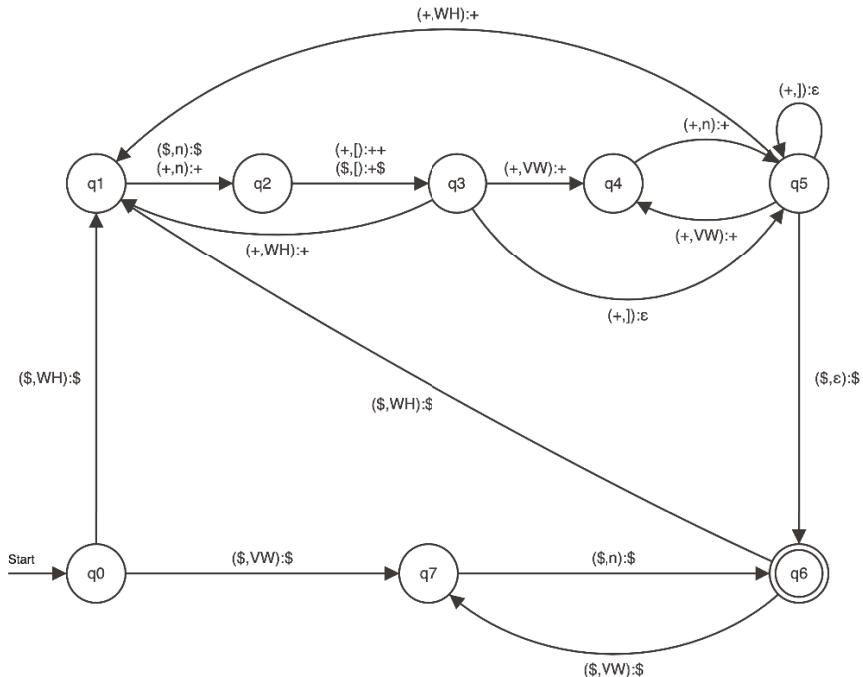
Also konzentrieren wir uns auf DKAs und verwenden zunächst einen DKA exemplarisch zur Beschreibung eines kleinen Ausschnitts unserer ZR-Sprache.

Beispiel 6.12

Die folgenden Regeln beschreiben eine simple „Subsprache“ von ZR, s. Abbildung 5.17.

$$\begin{array}{lcl} \text{Anweisungen} & \rightarrow & \text{Anweisung Anweisungen} \\ | & & \epsilon \\ \text{Anweisung} & \rightarrow & \text{WH n [Anweisungen]} \\ | & & \text{VW n} \end{array}$$

Die Terminale sind in der gewohnten Form notiert. n ist also hier wirklich ein verwendbares Zeichen und nicht etwa ein Platzhalter für eine Zahl. Das Nichtterminal *Anweisungen* ist das Startsymbol. Ein DKA für diese Subsprache ist $M = (\{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7\}, \{[], n, VW, WH\}, \{\$, +\}, \delta, q_0, \$, \{q_6\})$ mit



M akzeptiert das Wort $VW\ n\ VW\ n\ WH\ n\ [VW\ n\ VW\ n\ WH\ n\ [VW\ n]]\ VW\ n\ WH\ n\ [WH\ n\ [VW\ n]\ VW\ n\ VW\ n]$ nach 36 Schritten.

Schaut man etwas genauer hin, erkennt man, dass der angegebene DKA das leere Wort – obwohl lt. Sprachdefinition zulässig – nicht akzeptiert. Diese Ausnahme nehmen wir in Kauf, da zur formalen Korrektur ein NKA erforderlich wäre.

Hätten Sie den DKA in Beispiel 6.12 auch angeben können? Offenbar ist es schon für kfG mit wenigen Regeln eine Herausforderung, einen äquivalenten DKA anzugeben. Deshalb wurden dafür hilfreiche Werkzeuge entwickelt. Man nennt sie *Parsergeneratoren*. Sie erwarten die Definition einer Sprache als Eingabe und generieren daraus einen zugehörigen Parser. Diese Prinzip kennen wir bereits vom Scannergenerator, s. Abschnitt 4.5 ab Seite 117. Dort haben wir darauf hingewiesen, dass Parser und Scanner zusammenarbeiten. Darauf kommen wir hier zurück, um unseren Parser dementsprechend zu strukturieren.

Als „menschenwürdige“ Beschreibungsmittel für kontextfreie Sprachen (Parser) verwenden wir kontextfreie Grammatiken und für reguläre Sprachen reguläre Ausdrücke (Scanner). In FLACI arbeitet JISON „unter der Haube“ und bietet durch die Einbettung in FLACI eine komfortable Nutzererfahrung für Scanner- und Parsergenerierung.

Parsergeneratoren

6.8 Optimierung kontextfreier Grammatiken

Der Entwurf einer kfG für eine Sprache mit echter Anwendungsrelevanz ist ein komplexer Prozess. Um ihn zu beherrschen, gehen wir meist top-down vor, so dass die grundlegenden Eingabewort-Strukturen als erste beschrieben und danach deren innerer Aufbau nachgereicht werden. Dabei verwenden wir aussagefähige Nichtterminale (speaking symbols) in Regeln, die die jeweilige Teilstruktur für den menschlichen Leser verständlich benennen. Dies fördert auch die Wartbarkeit, d.h., wir können die Rolle, die die betrachtete Regel spielt, rasch reproduzieren, um Erweiterungen oder ggf. Korrekturen anzubringen.

Was für die Lesbarkeit und die Beherrschung der Komplexität des Entwurfsprozesses zuträglich ist, kann für den Prozess der Syntaxanalyse unnötigen Ballast bedeuten. Unnötige Nichtterminale, die wir aus o.g. Gründen gern etwas üppig verwendet haben, führen aus der Sicht des praktischen Parsings zu vermeidbaren Zuständen und zeitaufwendigen Übergängen. Außerdem lässt manche Ableitung erkennen, dass die eine oder andere Regel nach einem kleinen Eingriff ganz entfallen kann.

Beispiel 6.13

Eine kleine „ZR-ähnliche“ Sprache könnte im ersten Entwurf mit folgender kfG definiert worden sein: $G = (\{befehl, befehl_mit_parameter, einfacher_befehl\}, \{START, n, VW, RE, STIFTFARBE, START\}, P, befehl)$ mit $P =$

$$\begin{aligned} \textit{befehl} &\rightarrow \textit{befehl_mit_parameter} \mid \textit{einfacher_befehl}, \\ \textit{befehl_mit_parameter} &\rightarrow \textit{VW} \ n \mid \textit{RE} \ n \mid \textit{STIFTFARBE} \ n \mid \textit{WH} \ n \ \textit{befehl} \\ \textit{einfacher_befehl} &\rightarrow \textit{START} \} \end{aligned}$$



Offensichtlich ist die sog. Kettenregel $\textit{befehl} \rightarrow \textit{einfacher_befehl}$ überflüssig, wenn man $\textit{befehl} \rightarrow \textit{START}$ ergänzt. $\textit{einfacher_befehl} \rightarrow \textit{START}$ entfällt.

Entfernt man sog. Kettenregeln und unnütze Nichtterminale, ergibt sich die folgende Regelmenge, die nur noch ein einziges Nichtterminal, nämlich *befehl*, enthält.

$$\textit{befehl} \rightarrow \text{START} \mid \text{VW } n \mid \text{RE } n \mid \text{STIFTFARBE } n \mid \text{WH } n \text{ } \textit{befehl}$$

Im Folgenden wird gezeigt, wie man Grammatik-Optimierungen in Bezug auf *unnütze Nichtterminale* und *Kettenregeln* algorithmisch durchführt.

Entfernung
unnützer
Nichtterminale
Kettenregeln

Unnütze Nichtterminale (useless nonterminals) sind solche, die nichts zur Erzeugung der Sprache $L(G)$ beitragen. Um sie herauszufiltern, fragen wir umgekehrt nach Nichtterminalen, die *unverzichtbar* sind:

1. Auf das Spitzensymbol s kann natürlich nicht verzichtet werden, da es Ausgangspunkt jeder Ableitung ist: $s \xrightarrow{*} w$.
2. Für alle $X \in N$ mit $X \neq s$ gilt:
 - a) X darf nicht gestrichen werden, wenn es mindestens ein Wort $w \in T^+$ gibt, mit $X \xrightarrow{*} w$
 - b) und wenn $s \xrightarrow{*} \alpha X \beta$, wobei $\alpha, \beta \in (N \cup T)^*$, d.h. wenn X in wenigstens einer vom Spitzensymbol aus erreichbaren Satzform vorkommt.

Die nach der zweiten Forderung unverzichtbaren Nichtterminale werden iterativ „eingesammelt“. Zuerst wenden wir uns der Forderung 2a zu:

$$\begin{aligned} M_1 &= \{X \mid X \in N \text{ und } (X \rightarrow \alpha) \in P, \text{ mit } \alpha \in T^+\} \\ M_{i+1} &= M_i \cup \{X \mid X \in N \text{ und } (X \rightarrow \alpha) \in P, \text{ mit } \alpha \in (M_i \cup T)^+\}, i \geq 1. \end{aligned}$$

Die Bildung der Mengen M_i wird bei Menge M_n beendet, wenn sich M_n nicht von M_{n+1} unterscheidet, d.h. wenn $M_n = M_{n+1} = \dots = M_{n+k}$. n ist also der kleinste Index mit dieser Eigenschaft. Aus $M_i \subseteq N$ für alle $1 \leq i \leq n$ folgt, dass dieser Prozess in jedem Fall terminiert. M_n enthält sämtliche Nichtterminale aus N , die zu einem Wort $w \in T^+$ führen.

Die ursprüngliche Grammatik $G = (N, T, P, s)$, mit $L(G) \neq \emptyset$, wird reduziert zu $G' = (N', T', P', s') = (M_n, T, \{X \rightarrow \alpha \mid (X \rightarrow \alpha) \in P \text{ und } X \in M_n\}, s)$.

In P' werden alle Regeln $X \rightarrow \alpha$ gestrichen, die auf der rechten Seite Nichtterminale Y aus $(N \setminus M_n)$ enthalten, für die also $Y \in \alpha$ gilt.

Nun ist Forderung 2b an der Reihe: Wieder werden die unverzichtbaren Nichtterminale iterativ „eingesammelt“:

$$\begin{aligned} M'_1 &= \{s\} \\ M'_{i+1} &= M'_i \cup \{X \mid X \in N' \text{ und } (Z \rightarrow \alpha X \beta) \in P', \\ &\quad \text{mit } \alpha, \beta \in (N' \cup T')^*, Z \in M'_i\}, \text{ für } i \geq 1. \end{aligned}$$

M'_n mit dem kleinsten n mit Eigenschaft $M'_n = M'_{n+1}$ enthält alle Nichtterminale, die von $s' = s$ aus erreichbar sind. Der Prozess terminiert ganz sicher. Die gesuchte,

gegenüber G reduzierte Grammatik ist $G'' = (N'', T'', P'', s'') = (M'_n, T, \{X \rightarrow \alpha \mid (X \rightarrow \alpha) \in P' \text{ und } X \in M'_n\}, s)$.

Computerübung 6.11

Für die Bearbeitung von Beispiel 6.14 mit FLACI (Transformation „Grammatik vereinfachen“) ist zu beachten, dass Nichtterminale (wie F im Beispiel) einer kfG, die in keiner Produktion auf der linken Seite stehen, als Terminale angesehen werden. Diese Verabredung sorgt dafür, dass die Eingabe der Grammatik in FLACI auf die Eingabe der Regeln reduziert wird. Hieraus ergeben sich alle Bestandteile der definierten kfG.

Um nun FLACI mitzuteilen, dass F ein Nichtterminal ist, fügen wir die Regel $F \rightarrow F$ hinzu.

Beispiel 6.14

Gegeben ist $G = (\{A, B, C, D, E, F\}, \{a, b\}, P, A)$, mit $P = \{A \rightarrow aBb \mid aA \mid D, D \rightarrow E, E \rightarrow ab, B \rightarrow Fa, C \rightarrow abba\}$.

Gesucht ist die zugehörige reduzierte Grammatik G'' . Diese wird in den beiden oben beschriebenen Schritten hergestellt:

$$M_1 = \{E, C\}, M_2 = \{E, C, D\}, M_3 = \{E, C, D, A\} = M_4, n = 3.$$

$G' = (N', T', P', s') = (\{A, C, D, E\}, \{a, b\}, P', A)$, mit $P' = \{A \rightarrow aA \mid D, D \rightarrow E, E \rightarrow ab, C \rightarrow abba\}$. Da $B \notin N'$, entfällt auch Regel $A \rightarrow aBb$.

$$M'_1 = \{A\}, M'_2 = \{A, D\}, M'_3 = \{A, D, E\} = M'_4, n = 3.$$

$$G'' = (N'', T'', P'', s'') = (\{A, D, E\}, \{a, b\}, \{A \rightarrow aA \mid D, D \rightarrow E, E \rightarrow ab\}, A).$$

Nun wenden wir uns den *Kettenregeln* (unit productions) zu. Darunter versteht man Produktionen der Form $X \rightarrow Y$ mit $X, Y \in N$, d.h. Regeln, auf deren linken und rechten Seite genau ein Nichtterminal steht. In Beispiel 6.14 ist gut zu erkennen, dass die resultierende Grammatik G'' mit $A \rightarrow D, D \rightarrow E$ zwei derartige Kettenregeln enthält. G'' kann also weiter reduziert werden.

Das im Folgenden beschriebene Verfahren zur *Beseitigung von Kettenregeln* kann auch für kfG mit ϵ -Regeln angewandt werden. Zum schnellen Verständnis fügen wir der Erklärung ein sehr einfaches Beispiel bei:

$$G = (\{A, B, S\}, \{b\}, \{S \rightarrow A, A \rightarrow B, B \rightarrow b\}, S).$$

1. Bilde die Menge M aller Paare $(A, B) \in N \times N$ mit $A \xrightarrow{*} B$. M ist eine endliche Menge von Paaren mit höchstens $|N|^2$ Elementen. Folglich ist M in jedem Fall algorithmisch konstruierbar.

Im Beispiel: $M = \{(S, S), (S, A), (S, B), (A, A), (A, B), (B, B)\}$

2. Bilde die reduzierte Grammatik $G' = (N, T, P', s)$ mit

$$P' = \{X \rightarrow \beta \mid (X, Y) \in M \text{ und } (Y \rightarrow \beta) \in P, \beta \notin N\}.$$

Im Beispiel: $P' = \{S \rightarrow b, A \rightarrow b, B \rightarrow b\}$

Im Anschluss sollten die unnützen Nichtterminale in G' mit dem weiter oben beschriebenen Verfahren beseitigt werden.

Beispiel 6.15

Wir nehmen das Ergebnis $G'' = (\{A, D, E\}, \{a, b\}, \{A \rightarrow aA \mid D, D \rightarrow E, E \rightarrow ab\}, A)$ aus Beispiel 6.14 auf und entfernen die darin enthaltenen Kettenregeln.

1. Schritt: $M = \{(A, A), (A, D), (A, E), (D, D), (D, E), (E, E)\}$
2. Schritt: $G''' = (\{A, D, E\}, \{a, b\}, \{A \rightarrow aA \mid ab, D \rightarrow ab, E \rightarrow ab\}, A)$



Kettenregeln

Offensichtlich enthält G''' unnötige Nichtterminale. Diese können mit dem weiter oben beschriebenen Verfahren eliminiert werden: $M_1 = \{A, D, E\} = M_2$. $M'_1 = \{A\}$, $M'_2 = M'_1 \cup \{A\} = \{A\} = M'_1$. Schließlich bleiben nur noch die beiden Regeln $A \rightarrow aA \mid ab$ übrig.

Computerübung 6.12



Verwenden Sie auch FLACI zur Bearbeitung von Beispiel 6.15.

6.9 CHOMSKY-Normalform

Grammatiktransformationen der in Abschnitt 6.8 beschriebenen Form werden nicht nur zur Reduktion eingesetzt, sondern können auch zur Herstellung von Regelstrukturen verwendet werden, die in gewisser Hinsicht vorteilhaft sind.

Definition 6.4



Eine kfG $G = (N, T, P, s)$ ist in CHOMSKY-Normalform, kurz: *CNF*, wenn jede Regel aus P entweder die Form $X \rightarrow a$ oder $X \rightarrow YZ$, mit $a \in T$ und $X, Y, Z \in N$, besitzt.

Falls es in der betrachteten kfG ϵ -Regeln gibt, kann man das Verfahren aus dem Beweis von Satz 2.3 nutzen, um eine äquivalente kfG ohne ϵ -Regeln herzustellen. Sollte ϵ zur Sprache $L(G)$ gehören, wird ein neues Spitzensymbol s' mit $s' \rightarrow \epsilon \mid s$ eingeführt und nur die Sub-Grammatik mit s als Spitzensymbol hinsichtlich der geforderten Regelgestalt beurteilt und ggf. transformiert.

CNF Die speziellen Regelgestalten einer CNF sind in Definition 6.4 angegeben. Diese gewährleisten, dass Ableitungsbäume stets Binäräbäume sind, da das jeweils linke bzw. rechte Nichtterminal Wurzel des zugehörigen Ableitungsteilbaums ist. Eine Reihe theoretischer Untersuchungen können auf Basis der CNF eleganter oder überhaupt erst geführt werden, da sich Schlussfolgerungen über Binäräbäumen durch deren kalkulierbare Tiefe usw. anbieten.

CYK Außerdem kann für kfG in CNF ein sehr effizientes Parse-Verfahren angegeben werden: der *Cocke-Younger-Kasami-Algorithmus* (CYK). Die Grundlage für die Suche nach diesem Algorithmus bildet der folgender Satz.

Satz 6.3



Zu jeder kfG G , mit $\epsilon \notin L(G)$, gibt es eine äquivalente kfG G' in CHOMSKY-Normalform.

Beweis

Der Beweis ist konstruktiv und zeigt, wie G' aus G algorithmisch gewonnen wird. Dabei ist es sinnvoll, die weiter oben besprochenen Grammatik-Optimierungen (Entfernen unnötiger Nichtterminale und Kettenregeln) vorzuschalten.

1. Übernimm' alle Regeln der Form $X \rightarrow a$ mit $X \in N$ und $a \in T$ aus P in P' .
2. Ersetze in jeder Regel der Form $X \rightarrow \beta$ mit $\beta \in (N \cup T)^*$ und $|\beta| \geq 2$ jedes Terminal a durch X_a und ergänze die Regeln $X_a \rightarrow a$. X_a ist ein neues Nichtterminal. Übertrage die so veränderten Regeln in P' .

3. Ersetze alle Regeln der Form $X \rightarrow Y_1 Y_2 \dots Y_n$, $n \geq 3$, durch

$$\begin{array}{lcl} X & \rightarrow & Y_1 Z_1 \\ Z_1 & \rightarrow & Y_2 Z_2 \\ Z_2 & \rightarrow & Y_3 Z_3 \\ & \vdots & \\ Z_{n-2} & \rightarrow & Y_{n-1} Y_n \end{array}$$

wobei Z_i ($1 \leq i \leq n-2$) neue (paarweise verschiedene) Nichtterminale sind, und notiere sie in P' .

□

Beispiel 6.16

Wir betrachten die kfG $G = (\{S, A, B\}, \{a, b\}, P, s)$, mit $P = \{S \rightarrow aSa \mid aa \mid A, A \rightarrow bA \mid B, B \rightarrow b\}$.



1. Schritt: Kettenregeln eliminieren (z.B. mit FLACI)

$S \rightarrow aSa \mid aa \mid bA \mid b, A \rightarrow bA \mid b$

2. Schritt: Neue Nichtterminale für Terminale ergänzen

$S \rightarrow X_a SX_a \mid X_a X_a \mid X_b A \mid b, A \rightarrow X_b X_2 \mid b, X_a \rightarrow a, X_b \rightarrow b$

3. Schritt: Reduzierung rechter Regelseiten

Nur die Regel $S \rightarrow X_a SX_a$ hat eine rechte Seite der Länge 3. Nach Substitution erhält man

$S \rightarrow X_a Z_1 \mid X_a X_a \mid X_b A \mid b, Z_1 \rightarrow SX_a, A \rightarrow X_b A \mid b, X_a \rightarrow a, X_b \rightarrow b$

Das Ergebnis lautet:

$G = (\{S, X_a, X_b, A, Z_1\}, \{a, b\}, \{S \rightarrow X_a Z_1 \mid X_a X_a \mid X_b A \mid b, Z_1 \rightarrow SX_a, A \rightarrow X_b A \mid b, X_a \rightarrow a, X_b \rightarrow b\}, S)$.

Computerübung 6.13

Führen Sie die in Beispiel 6.16 betrachtete Transformation mit FLACI durch und nutzen Sie die Möglichkeit, die verwendeten Nichtterminale ggf. so umzubenennen, dass sie mit denen aus dem Beispiel übereinstimmen.



Beispiel 6.17

Das CYK-Verfahren ist ein relativ effizientes Parseverfahren für kfS. Die Zeitkomplexität des Verfahrens liegt in $O(n^3)$, wobei n die Länge des Eingabewortes ist. Im Unterschied zum Earley-Parser muss die vorgelegte kfG „vorbehandelt“, d.h. in CNF transformiert werden.

Das für CYK verwendete algorithmische Entwurfsmuster ist *dynamisches Programmieren*. Traditionell⁶ wird eine Tabelle vorbereitet, in der schrittweise jeweils die Nichtterminale eingetragen werden, aus denen das analysierte Teilwort abgeleitet werden kann. Die Berechnung beginnt bei den einzeichenigen Teilwörtern, aus denen das Eingabewort besteht und setzt sich (unter Berücksichtigung aller prinzipiell möglichen Zerlegungen) bis zum gesamten Wort fort. Dabei wird konsequent auf vorhandene Teilresultate zurückgegriffen.



⁶Eine alternative Implementierung ergibt sich durch ein Top-Down-Verfahren gemäß einer meist rekursiv vorliegenden Resultatbeschreibung unter Einsatz von Memoizing.

CYK-Verfahren
dynamisches
Programmieren

Für die Entscheidung, ob das betrachtete Eingabewort w zu $L(G)$ gehört oder nicht, ist lediglich von Interesse, ob sich das Spitzensymbol s im obersten Tabellenfeld, das für das Gesamtwort w steht, befindet.

Wenn man die Akzeptanz des Wortes $baaba$ für $G = (\{S, A, B, C\}, \{a, b\}, P, S)$ mit $P = \{S \rightarrow AB \mid BC, A \rightarrow BA \mid a, B \rightarrow CC \mid b, C \rightarrow AB \mid a\}$ untersucht, baut man einen „Turm“ (Tabelle) wie in Abbildung 6.11.

| | | | | |
|---------------|------------|---------------|------------|------------|
| $\{S, C, A\}$ | | | | |
| $\{\}$ | | $\{S, C, A\}$ | | |
| $\{\}$ | | $\{B\}$ | | $\{B\}$ |
| $\{S, A\}$ | $\{B\}$ | $\{S, C\}$ | $\{S, A\}$ | |
| $\{B\}$ | $\{A, C\}$ | $\{A, C\}$ | $\{B\}$ | $\{A, C\}$ |
| b | a | a | b | a |

Abbildung 6.11: $baaba \in L(G)$ für die obige Grammatik G mit CYK

Wie man aus der Regelmenge unmittelbar entnimmt, können a genau aus A und C sowie b aus B abgeleitet werden. Die betreffenden Nichtterminale fassen wir zu Mengen zusammen, d.h. $\{A, C\}$ bzw. $\{B\}$. Dies spiegelt die unterste Schicht des „Turmes“ für $baaba$ in Abbildung 6.11 wider.

Für das Teilwort ab notieren wir in der zweiten Ebene $\{S, C\}$, denn lt. Grammatik gelten nur $S \Rightarrow AB$ und $C \Rightarrow AB$. Außerdem gibt es kein Nichtterminal, aus dem man die Satzform AB ableiten kann.

Wir erklären, wie der Inhalt des fett umrahmten Feldes entsteht. Es enthält alle Nichtterminale, aus denen das Wort aba abgeleitet werden kann, d.h. gesucht sind $N_?$ mit $N_? \Rightarrow XY \stackrel{*}{\Rightarrow} aba$. Für das Wort aba gibt es die folgenden zwei Zerlegungsmöglichkeiten:

1. $X \stackrel{*}{\Rightarrow} a$, $Y \stackrel{*}{\Rightarrow} ba$, d.h. $\{A, C\} - \{S, A\}$, sodass $N_? \Rightarrow AS$, $N_? \Rightarrow CS$, $N_? \Rightarrow AA$, $N_? \Rightarrow CA$. Hierfür passt keine Regel der Grammatik.
2. $X \stackrel{*}{\Rightarrow} ab$, $Y \stackrel{*}{\Rightarrow} a$, d.h. $\{S, C\} - \{A, C\}$, sodass $N_? \Rightarrow SA$, $N_? \Rightarrow CA$, $N_? \Rightarrow SC$, $N_? \Rightarrow CC$. Die einzige passende Regel ist $B \rightarrow CC$. Deshalb lautet der Eintrag in dem betrachteten Feld $\{B\}$.

Übung 6.4



Folgen Sie dem CYK-Verfahren und füllen Sie die Tabelle in Abbildung 6.11 vollständig aus.

Der CYK kann so erweitert werden, dass man den Ableitungsbaum rekonstruieren kann. Hierfür werden nicht nur alle die Nichtterminale (als Menge), aus denen das betrachtete Teilwort abgeleitet werden kann, notiert, sondern zusätzlich die jeweils angewandte Regel und die dabei gewählte Zerlegungsposition.

6.10 Das Pumping Lemma für kontextfreie Sprachen

In Abschnitt 3.5 haben wir bereits das Pumping Lemmas für reguläre Sprachen behandelt. Einen entsprechenden Satz gibt es auch für kfS. Der zugehörige Beweis macht direkten Gebrauch von der im vorangehenden Abschnitt betrachteten CHOMSKY-Normalform für kfG.

Wir wissen, dass jede kfS durch eine kfG in CHOMSKY-Normalform definiert werden kann. Aufgrund der Beschränkung auf die einzigen beiden Regelformen $X \rightarrow a$ und $X \rightarrow YZ$, mit $a \in T$ und $X, Y, Z \in N$, hat jeder CNF-Ableitungsbaum stets die in Abbildung 6.12 dargestellte Gestalt. Es handelt sich um einen Binärbaum, dessen Wurzel mit dem Spitzensymbol s der betrachteten CNF-Grammatik beschriftet ist. Man sagt, dass sich die Wurzel auf Baum-*Niveau 0* befindet. Dieses und die folgenden Niveaus bis einschließlich Nummer $m + 1$ sind in Abbildung 6.12 angegeben.

Die Blätter des Baumes bestehen aus Terminalen der Grammatik. Sie liegen auf Niveau $m + 1$. Da diese Terminal-Blätter durch Anwendungen der Regeln $X \rightarrow a$, mit $X \in N$ und $a \in T$ erzeugt werden, stimmt die Anzahl der Blätter mit der Anzahl der Knoten des unmittelbar vorhergehenden Niveaus m überein.

Die Anwendung der Regeln vom Typ $X \rightarrow YZ$, mit $X, Y, Z \in N$, führt auf dem jeweiligen Niveau hingegen zur Verdopplung der Knotenzahl gegenüber dem vorhergehenden.

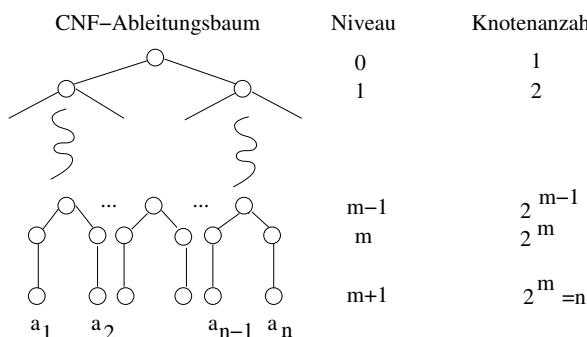


Abbildung 6.12: Struktur eines Ableitungbaumes für ein Wort einer kfG in CNF

Für ein beliebiges Wort $z = a_1 a_2 \dots a_n \in L(G)$ der Länge $n = 2^m$ besitzt ein CNF-Ableitungsbaum $m + 2$ Niveaus (von 0 bis $m + 1$) und 2^m Terminalknoten, die von links nach rechts verkettet z ergeben.

Sämtliche inneren Knoten des Baumes sind mit Nichtterminal-Symbolen markiert. Die maximale Länge eines Pfades von der Wurzel s zu einem beliebigen Terminal-Blatt ist $m + 1$: Jeder Schritt von Knoten zu Knoten vollzieht sich im Einklang

maximale Pfadlänge

mit der jeweiligen Niveaunummer des Baumes beginnend bei 0. Ein solcher Pfad enthält (einschließlich des Wurzelknotens s) $m+2$ Knoten $(0, 1, 2, \dots, m+1)$, wovon $m+1$ auf Nichtterminale entfallen.

Wählt man nun ein $z \in L(G)$ mit $|z| = 2^m$ gerade so, dass $m = |N|$, d.h. m ist die Anzahl der Nichtterminale der CNF-Grammatik G für L , so gibt es im Ableitungsbaum für z mindestens einen Pfad von der Wurzel s zu a_i , der über wenigstens $m+1$ Nichtterminal-Knoten führt. Da es aber nur m Nichtterminale in G gibt, muss es mindestens ein Nichtterminal A geben, das auf diesem Pfad mehrfach (also mindestens doppelt) vorkommt.

Den beschriebenen Sachverhalt illustriert Abbildung 6.13.

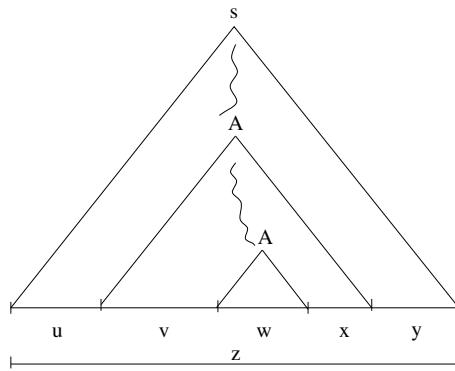


Abbildung 6.13: Doppelung eines Nichtterminals A im CNF-Ableitungsbaum

Da A mit einer Regel der Form $A \rightarrow BC$ weiter abgeleitet wird, gilt $vx \neq \epsilon$, d.h. $|vx| \geq 1$.

Gilt das auch für die Sonderfälle $A \rightarrow AB$, $A \rightarrow BA$ und $A \rightarrow AA$, bei denen A auf der rechten Seite steht, wie in Abbildung 6.14 dargestellt? Jawohl!

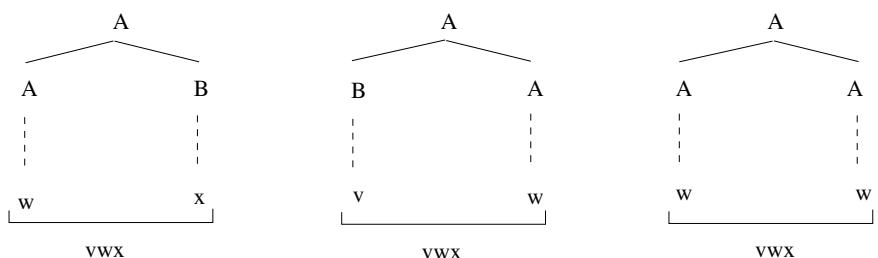


Abbildung 6.14: $vwvx$ -Zerlegung bei verschiedenen Regeln für A

1. Fall: $A \rightarrow AB, A \xrightarrow{*} w, B \xrightarrow{*} x$

Ergebnis: Aus $vwx = wx$ folgt $v = \epsilon$ und $w, x \neq \epsilon$, also gilt $vx \neq \epsilon$.

2. Fall: $A \rightarrow BA, A \xrightarrow{*} w, B \xrightarrow{*} v$

Ergebnis: Aus $vwx = vw$ folgt $x = \epsilon$, und $w, v \neq \epsilon$, also gilt $vx \neq \epsilon$.

3. Fall: $A \rightarrow AA, A \xrightarrow{*} w$

Ergebnis: Aus $vwx = ww$ folgt $v = \epsilon$ und $x = w$ bzw. $v = w$ und $x = \epsilon$, also gilt stets $vx \neq \epsilon$.

Übung 6.5

Skizzieren Sie den allgemeinen Fall $A \rightarrow BC$.



Aus der weiter oben angestellten Betrachtung des CNF-Ableitungsbaumes wird außerdem klar, dass das aus A abgeleitete Teilwort vwx nicht länger sein kann als z selbst, d.h. $|vwx| \leq 2^m$. Auch diese Eigenschaft wird sich in Satz 6.4 wiederfinden.

Analog zum Pumping Lemma für reguläre Sprachen wird die Pumping-Eigenschaft intuitiv augenscheinlich. Abbildung 6.15 unterstützt die Beobachtung.

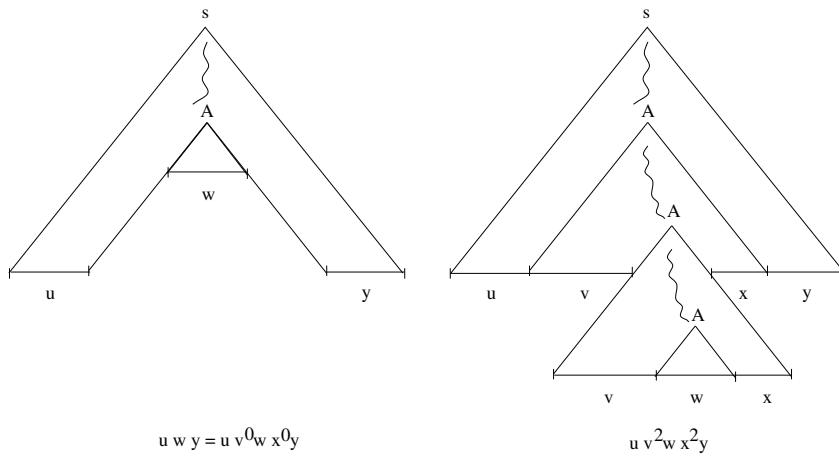


Abbildung 6.15: Struktur eines Ableitungbaumes für ein Wort einer kfG in CNF

Während im linken Teil von Abbildung 6.15 das Teilwörter v und x aus z „herausgeschnitten“ wurden, sind sie im rechten einmal mehr „hineingepumpt“ worden. In beiden Fällen entsteht ein Wort, das zu $L(G)$ gehört, denn der Teilbaum mit Wurzel A kann entfernt oder beliebig oft eingehangen werden, ohne am Erfolg der Ableitung $s \xrightarrow{*} z$ etwas zu ändern.

Nun sind wir gut vorbereitet, um die Formulierung des Pumping Lemmas für kfS in Satz 6.4 verstehen zu können.



Satz 6.4

Sei L eine kontextfreie Sprache. Dann existiert eine Konstante n , sodass sich jedes Wort z aus L mit $|z| \geq n$ in der Form $z = uvwxy$, mit $u, v, w, x, y \in T^*$, schreiben lässt und für alle $i \geq 0$ gilt $uv^iwx^i y \in L$, mit $|vx| \geq 1$ ($vx \neq \epsilon$) und $|vwx| \leq n$.

Beweis

Fall 1: L sei eine endliche kfS. Wähle n größer als die Länge des längsten Wortes in L . Dann gibt es kein einziges Wort $z \in L$ mit $|z| \geq n$, d.h. die Menge aller z , für die die Eigenschaften des Pumping Lemmas gelten sollen, betrifft eine leere Menge. Da man in einer leeren Menge kein einziges Element antrifft, das die Eigenschaften nicht besitzt, ist die Gültigkeit des Satzes für Fall 1 bewiesen. Man spricht auch von einem sog. „Schubfachschluss“ (engl. pigeonhole principle, d.h. Taubenschlagprinzip).

Fall 2: $L = L(G)$ ist eine unendliche kfS. $G = (N, T, P, s)$ ist eine kfG in CNF, wobei $|N| = m$. O.B.d.A. sei $\epsilon \notin L$. Dann gelten die obigen Überlegungen, die wir zur Herleitung des Satzes angestellt haben. \square

Beweistechnisch wird das Pumping Lemma für kfS genauso eingesetzt, wie das für reguläre Grammatiken, s. Satz 3.5: Wir zeigen, dass die betrachtete Sprache nicht kontextfrei ist.

Beispiel 6.18

Beispiel

$$L = \{a^n b^n c^n \mid n \geq 0\}$$

Die Sprache $L = \{a^n b^n c^n \mid n \geq 0\}$, ist kontextsensitiv. Wir zeigen mit dem Pumping Lemma, dass L nicht kontextfrei ist.

Angenommen L wäre eine kfS. Dann existiert ein n mit den im Pumping Lemma angegebenen Eigenschaften. Wähle $k > \frac{n}{3}$ mit $k \in \mathbb{N}$. Das Wort $z = a^k b^k c^k$ hat eine Länge von $3 \cdot k > n$. Die Bedingung $|z| \geq n$ ist also erfüllt.

Mit $z = uvwxy \in L$ gilt lt. Pumping Lemma auch $uv^iwx^i y \in L$, für $i \geq 0$. Man sieht, dass v (und ebenso x) aus genau einem Buchstaben – also entweder aus a oder b oder c – bestehen muss, da ansonsten ein Hineinpumpen von z.B. ab zu einem unerlaubten Mix, wie beispielsweise $ababab = (ab)^3$ als Teilwort, führen würde. Ein Wort, das eine solche Zeichenkette enthält, verstößt gegen die Form $a^n b^n c^n$ und gehört deshalb nicht zu L .

Aber auch wenn v aus genau einem Zeichen besteht, führt dessen wiederholte Verwendung dazu, dass die Anzahlen der verwendeten as , bs und cs nicht übereinstimmen.

Im Widerspruch zur Aussage des Pumping Lemmas liegen also nicht alle diese Wörter $uv^iwx^i y$ in L und die Negation der Annahme gilt: L ist nicht kontextfrei.



7 LL(k)-Sprachen

7.1 Deterministische Top-down-Syntaxanalyse

Aus Kapitel 6 wissen wir, dass

- Chomsky-Typ-2-Grammatiken und NKAs gleichberechtigte Beschreibungs-mittel für die Klasse der kfS sind, und
- die sog. *deterministisch-kontextfreien Sprachen* (dkfS), die durch DKAs be-schrieben werden, eine echte Untermenge der kontextfreien Sprachen bil-den.

Top-down-Verfahren zur Syntaxanalyse *kontextfreier Sprachen* beginnen mit dem Spitzensymbol und erzeugen das Analysewort vereinbarungsgemäß durch Links-ableitungen. Wir wissen, dass dieser Prozess durchaus in Sackgassen führen kann. Der danach durch Backtracking verursachte Zeitaufwand ist für praktisches Par-sing ggf. vernichtend inakzeptabel. Von daher ist es sehr sinnvoll, nach Sprachklas-sen zu suchen, für die das Parsing Backtrack-frei stattfindet. DkfS (als Untermenge der kfS) bilden eine solche Klasse.

Deterministisches Top-down-Parsing bedeutet, dass die in jedem Ableitungsschritt anzuwendende Regel eindeutig bestimmbar ist. Jede Regelauswahl erfolgt al-so „irrtumsfrei“, d.h. ohne Sackgassen, ohne nachträgliche Korrektur – also oh-ne Backtracking. Die im jeweils nächsten Ableitungsschritt anzuwendende Pro-duktion ist treffsicher *vorhersagbar*. Hieraus ergibt sich auch die Bezeichnung *prädiktive Syntaxanalyse*, engl.: to predict = vorhersagen.

Um dies zu erreichen, muss in jeder aktuellen Analysesituation auf ein gewis-ses Anfangsstück des noch zu analysierenden Restwortes vorausgeschaut werden. Man spricht daher auch von *lookahead*-Verfahren. Natürlich ist man daran interes-siert, diese Treffsicherheit mit möglichst wenigen *Vorausschauzeichen* bzw. *-Token* zu erzielen. Prädiktive Parser für *LL(k)*-Sprachen, s. Abschnitt 7.2, sind schon für $k = 1$ Vorausschauzeichen sehr leistungsfähig und es gibt transparente Verfahren zu deren Implementierung.

Bei aller Leistungsfähigkeit prädiktiver Parser darf jedoch nicht der Eindruck ent-stehen, *LL(k)*-Sprachen und dkfS seien identisch. *LL(k)*-Sprachen bilden eine ech-te Teilmenge der dkfS, d.h. $\mathcal{L}_{LL(k)} \subset \mathcal{L}_{dkfS}$, sodass unsere Suche nach umfas-senderen Sprachklassen mit effizienten Parsing-Algorithmen im nächsten Kapitel weitergeht. In Abbildung 7.1 sind die erwähnten Mengen dargestellt. Das stilisier-

dkfS

Top-down-Verfahren

Deterministisches Top-down-Parsing

prädiktive Syntaxanalyse

Vorausschau-zeichen

$\mathcal{L}_{LL(k)} \subset \mathcal{L}_{dkfS}$

te \mathcal{L} symbolisiert die jeweils mit Index benannte Sprachklasse, also eine *Menge* von Sprachen.

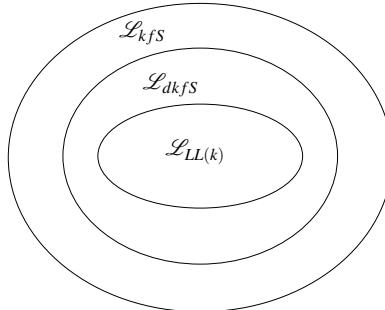


Abbildung 7.1: kfS und zwei relevante Teilmengen

Zur effizienten Analyse zahlreicher praktisch relevanter Sprachen reicht die Klasse der $LL(1)$ -Sprachen vollkommen aus. Die Beschäftigung mit dieser Sprachklasse lohnt sich also sehr.

7.2 Begriff und Einordnung

Die im Namen $LL(k)$ verwendeten Buchstabenkürzel haben folgende Bedeutungen:

- $L \dots$ Das erste (ganz links stehende) L steht für „Analyse des Eingabewortes von links nach rechts“. Wie bei DKAs definiert, arbeitet sich der Lesekopf konsequent von links nach rechts voran. Es gibt nicht etwa einen Rückgriff auf frühere Zeichen, etwa durch Zurückfahren (Linksbewegung) des Kopfes auf dem Eingabeband.
- $L \dots$ Das zweite L steht für „Linksableitung des Analysewortes“. Das am weitesten links stehende Nichtterminal wird zuerst ersetzt. Das Analysewort wird vom Spitzensymbol her erzeugt.
- $k \dots$ Die in Klammern stehende natürliche Zahl k (sinnvoll: $k \geq 1$) gibt die Anzahl der Vorausschauzeichen (genauer: Vorausschautoken) auf das noch nicht analysierte Restwort (die Resttokenfolge) an, sodass eine irrtumsfreie (Sackgassen ausschließende) Entscheidung des jeweils nächsten Ableitungsschrittes gewährleistet ist.

Definition 7.1

 $LL(k)$ -Sprachen, für deren *deterministische* Syntaxanalyse die Vorausschau auf k Folgezeichen ($k \geq 1$) ausreicht, nennt man *stark-LL(k)*-Sprachen, kurz: *SLL(k)*-Sprachen, oder *LF(k)*-Sprachen.

Im Keller des jeweils zugehörigen DKA stehen Informationen zur Analyse des bereits verarbeiteten Teilworts zur Verfügung. Diese könnten $LL(k)$ -Parser für die in jedem Schritt zu treffende Regelauswahlentscheidung einbeziehen. Aber wir verwenden den Begriff $LL(k)$ - für $SLL(k)$ -Parser, die das nicht tun. Dies wird durch folgende Befunde gestützt:

1. *Nicht* jede $LL(k)$ -Sprache ist auch stark- $LL(k)$. Aber man kann zu jeder $LL(k)$ -Grammatik eine äquivalente stark- $LL(k)$ -Grammatik angeben. Das ist sehr erfreulich, sodass wir uns im Folgenden nur noch mit stark- $LL(k)$ -Sprachen (ohne Auswertung des Kellerinhalts) befassen.
2. Jede $LL(1)$ -Grammatik ist eine $SLL(1)$ -Grammatik. Dies gilt nicht für beliebige k in $LL(k)$.
3. Jede $SLL(k)$ -Grammatik ist auch eine $LL(k)$ -Grammatik.
4. Eine $LL(k)$ -Grammatik ist niemals mehrdeutig und besitzt keine linksrekursiven Regeln.
5. Es gibt kfG, die für kein k $LL(k)$ -Grammatiken sind, s. Abbildung 7.1.
6. Für eine gegebene kfG ist *nicht allgemein entscheidbar*, ob sie durch eine $LL(k)$ -Grammatik definiert werden kann.
7. Die Frage, ob eine gegebene kfS *für ein festes k* vom $LL(k)$ -Typ ist, kann hingegen allgemein entschieden, d.h. für jede beliebige kfS entweder mit „ja“ oder mit „nein“ beantwortet werden.

Natürlich sind wir bestrebt, die Anzahl k ($k \geq 1$) der Vorausschauzeichen möglichst klein zu halten.

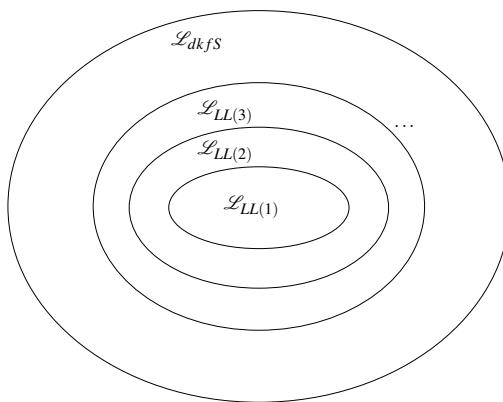


Abbildung 7.2: Hierarchie der $LL(k)$ -Sprachen

Dies hat jedoch Konsequenzen für die Leistungsfähigkeit des jeweiligen Parsing-Verfahrens: Wie in Abbildung 7.2 dargestellt, bilden die $LL(k)$ -Sprachen mit auf-

steigendem k eine Hierarchie, d.h.

$$\mathcal{L}_{LL(1)} \subset \mathcal{L}_{LL(2)} \subset \dots \subset \mathcal{L}_{LL(i)} \subset \mathcal{L}_{LL(i+1)} \subset \dots \subset \mathcal{L}_{dkfS}.$$

Wie weiter oben bereits angedeutet, ist sogar die (aus Effizienzgründen sehr beliebte) Klasse der *LL(1)-Sprachen* erfreulicherweise mächtig genug, um in der Praxis bestehen zu können. Beispielsweise können Pascal-Programme von einem *LL(1)-Parser* analysiert werden.

7.3 LL(1)-Forderungen

Welche Bedingungen muss eine kfS erfüllen, um *LL(1)*-Sprache zu sein?



Beispiel 7.1

Wir betrachten die Sprache $D = L(G) = \{x^n y, x^n z \mid n \geq 0\}$, mit $G = (N, T, P, s)$, wobei $N = \{A, B, S\}$, $T = \{x, y, z\}$, $s = S$ und $P = \{S \rightarrow A \mid B, A \rightarrow xA \mid y, B \rightarrow xB \mid z\}$. D ist keine *LL(1)*-Sprache. Warum?

Versucht man beispielsweise das offensichtlich zu D gehörende Wort xxz aus S durch Verwendung genau eines Vorausschauzeichens abzuleiten, kann die Analyse in eine Sackgasse geraten. Schon für den ersten Ableitungsschritt gibt es zwei Möglichkeiten. Die Vorausschau auf x lässt sowohl $S \rightarrow A$ als auch $S \rightarrow B$ zu.

Ein Syntaxdiagramm für S (Abbildung 7.3), in dem alle Nichtterminale außer S substituiert wurden, lässt die Unbestimmtheit der Regelauswahl unmittelbar erkennen. Man spricht von

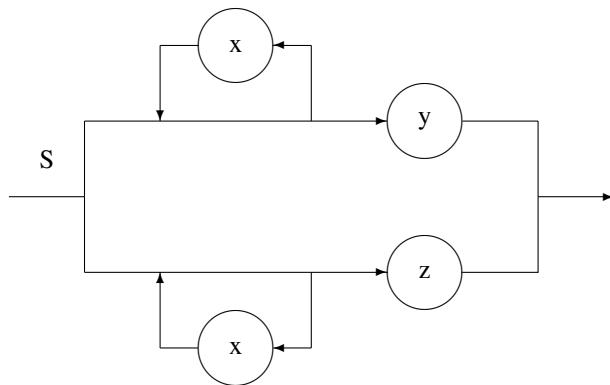


Abbildung 7.3: Syntaxdiagramm für S aus Beispiel 7.1

nichtdetermin.
Graphensystem

einem *nichtdeterministischen Graphensystem*. Für welchen Weg soll man sich entscheiden, wenn das erste Zeichen eines Wortes x ist? Für den oberen oder unteren Weg von S aus?

Aus Beispiel 7.1 lernen wir, dass die sog. *Zielsymbolmengen* alternativer Regeln *keine gemeinsamen Elemente* besitzen dürfen. Nur so kann man erreichen, dass

durch Vorausschau genau eines Zeichens die Auswahl der entsprechenden Regel ohne nachträgliche Korrektur stattfinden kann. Genau dies besagt Definition 7.2.

LL(1)-Ford. 1

Definition 7.2

Eine kfG G erfüllt die *LL(1)-Forderung 1*, wenn für jedes Nichtterminal X von G , mit $X \rightarrow \alpha_1 | \alpha_2 | \dots | \alpha_n$ und $\alpha_i \in (N \cup T)^*$, gilt

$$FIRST(\alpha_i) \cap FIRST(\alpha_j) = \emptyset \quad \text{für alle } i \neq j.$$



Anschaulich ist $FIRST(\alpha)$ die Menge aller Terminalzeichen, die bei allen möglichen Ableitungen von α an der jeweils ersten Stelle aller bildbaren Satzformen auftreten können.

FIRST

Definition 7.3

Für eine Satzform α gilt

$$FIRST(\alpha) := \{t \mid t \in T, \alpha \xrightarrow{*} t\beta, \beta \in (N \cup T)^*\}.$$



Falls $\alpha \xrightarrow{*} \varepsilon$, gilt zusätzlich $FIRST(\alpha) \ni \varepsilon$. (Lies „ \ni “ als „enthält“.)

Aus dieser Definition kann man noch kein Verfahren für die Konstruktion der *FIRST*-Mengen erkennen. In der Literatur gibt es verschiedene iterativ arbeitende Algorithmen für $FIRST(\alpha)$. Man erkennt sie an der Formulierung: „Beenden Sie dieses Verfahren, wenn sich an der/den aktuell gebildeten *FIRST*-Menge/n nichts mehr verändert, d.h. wenn keine weiteren Elemente hinzukommen“.

Verfahren zur Ermittlung von FIRST

Didaktischer Hinweis 7.1

Die im Folgenden angegebene rekursive Definition mag auf den ersten Blick etwas abschrecken, hat aber den großen Vorteil, dass sie mit einer höheren Programmiersprache unmittelbar umgesetzt werden kann. Da bei rekursiven Definitionen die *Beschreibung des Resultats* und nicht die des Berechnungsprozesses im Vordergrund steht, ist sie didaktisch wertvoll, da sie zur nochmaligen Auseinandersetzung mit dem eigentlichen Inhalt anregt.



Definition 7.4

Rekursive Definition der *FIRST*-Mengen

$$FIRST(\alpha) = \left\{ \begin{array}{ll} \{\}, & \alpha \text{ ist leer} \\ \{\varepsilon\}, & \alpha = \varepsilon \\ \{a\}, & \alpha = a \\ \bigcup_{i=1}^n FIRST(\beta_i), & \alpha = X \text{ und } X \rightarrow \beta_1 | \beta_2 | \dots | \beta_n \\ FIRST(Y_1), & \alpha = Y_1 Y_2 \dots Y_n, \varepsilon \notin FIRST(Y_1) \\ & (\text{gilt auch für } Y_1 \in T) \\ FIRST(Y_1) \cup FIRST(Y_2 \dots Y_m), & \alpha = Y_1 Y_2 \dots Y_m, \forall j : \varepsilon \in FIRST(Y_j) \\ (FIRST(Y_1) \cup FIRST(Y_2 \dots Y_m)) \setminus \{\varepsilon\}, & \alpha = Y_1 Y_2 \dots Y_m \end{array} \right.$$



wobei $a \in T$, $X \in N$, $\beta_i \in (N \cup T)^*$ und $Y_j \in N \cup T$.

Beispiel 7.2

Es ist zu prüfen, ob die kfG $G = (N, T, P, s)$ mit $N = \{K, S, E\}$, $T = \{a, b, d, c\}$, $P = \{K \rightarrow S \mid \epsilon, S \rightarrow aSb \mid E, E \rightarrow d \mid cE\}$ und $s = K$ die $LL(1)$ -Forderung 1 aus Definition 7.2 erfüllt.

Wir berechnen zunächst die $FIRST$ -Mengen für die Satzformen, die die rechten Regelseiten des jeweils betrachteten Nichtterminals bilden und prüfen anschließend deren paarweise Disjunktheit.

Für K : $FIRST(S) = \{a, d, c\}$, $FIRST(\epsilon) = \{\epsilon\}$, $FIRST(S) \cap FIRST(\epsilon) = \emptyset$.

Für S : $FIRST(aSb) = \{a\}$, $FIRST(E) = \{c, d\}$, $FIRST(aSb) \cap FIRST(E) = \emptyset$.

Für E : $FIRST(d) = \{d\}$, $FIRST(cE) = \{c\}$, $FIRST(d) \cap FIRST(cE) = \emptyset$.

Die Untersuchung zeigt, dass G die $LL(1)$ -Forderung 1 erfüllt.

Übung 7.1

Untersuchen Sie, ob die Grammatik G , mit $G = (\{X\}, \{a\}, \{X \rightarrow Xa \mid a\}, X)$, die $LL(1)$ -Forderung 1 erfüllt. Was stellen Sie fest? Verallgemeinern Sie Ihre Beobachtung.

Grammatik mit linksrekursiven Regeln
 $LL(1)$ -Ford. 2

Aus der Lösung von Übungsaufgabe 7.1 wissen wir, dass kfG mit wenigstens einer linksrekursiven Regel *nicht* vom $LL(1)$ -Typ sind. Deshalb werden wir uns in Abschnitt 7.6 überlegen, wie man Linksrekursionen beseitigen kann.

Wie das folgende Beispiel zeigt, reicht Forderung 7.2 nicht aus, um die $LL(1)$ -Eigenschaft einer beliebigen kfG sicherzustellen.

Beispiel 7.3

$G_2 = (\{A, S\}, \{x\}, \{S \rightarrow Ax, A \rightarrow \epsilon \mid xA\}, S)$. Gehört das Wort xxx zu $L(G_2)$?

Obwohl G_2 in Beispiel 7.3 die $LL(1)$ -Forderung 1 erfüllt, ist sie offensichtlich keine $LL(1)$ -Grammatik. Worin liegt das?

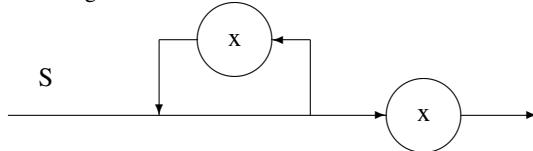


Abbildung 7.4: Syntaxdiagramm für S aus Beispiel 7.3

Die Anwendung der Produktionen $S \rightarrow Ax$ und $A \rightarrow xA$ führen dazu, dass das Terminal x in einer aus S abgeleiteten Satzform *vor* und *nach* dem Nichtterminal A stehen kann. Da in G_2 für das Nichtterminal A die Regel $A \rightarrow \epsilon$ existiert, ist nicht klar, ob das x in einer Satzform durch $S \rightarrow Ax$ oder $A \rightarrow xA$ erzeugt wurde. Dies führt zu folgender Erkenntnis.

Nur bei ϵ -Ableitungen, d.h. $X \stackrel{*}{\Rightarrow} \epsilon$, müssen die Initialsymbolmenge $FIRST(X)$ und die Folgesymbolmenge $FOLLOW(X)$ des betreffenden Nichtterminals X disjunkt sein.

Definition 7.5

Eine Grammatik G erfüllt die $LL(1)$ -Forderung 2, wenn für jedes Nichtterminal X , mit $X \stackrel{*}{\Rightarrow} \epsilon$, gilt

$$FIRST(X) \cap FOLLOW(X) = \emptyset.$$

Die Menge $FOLLOW(X)$ enthält also alle Terminalen, die einem Nichtterminal X in jeder aus s ableitbaren Satzform unmittelbar folgen können. „Unmittelbar“ bedeutet, dass ein solches Terminal nur dann in $FOLLOW(X)$ aufgenommen wird, wenn es direkt rechts neben X steht.

Definition 7.6

$$FOLLOW(X) := \{t \mid t \in T, s \xrightarrow{*} \alpha X t \beta, \text{ mit } \alpha, \beta \in (N \cup T)^*, X \in N\}$$

 $FOLLOW$ 

Zur Ermittlung von $FOLLOW(X)$ für ein Nichtterminal X mit o.g. Eigenschaft geht man folgendermaßen vor: Für jede in P enthaltene Regel, in der X auf der rechten Seite vorkommt, d.h. $A \rightarrow \alpha X \beta$, wobei $\alpha, \beta \in (N \cup T)^*$ und $A \neq X, \dots$

Berechnung
der $FOLLOW$ -Menge

1. ... wird $FIRST(\beta) - \text{ohne } \varepsilon$ – in $FOLLOW(X)$ aufgenommen.

Es ist offensichtlich, dass die Terminalen, die auf X folgen, genau die sind, die bei Ableitung von β ganz vorn stehen. Diese Terminalen bilden gerade die $FIRST$ -Menge von β .

2. ... und wenn $\beta \xrightarrow{*} \varepsilon$, d.h. $FIRST(\beta) \ni \varepsilon$, wird (zusätzlich) jedes Zeichen aus $FOLLOW(A)$ in $FOLLOW(X)$ aufgenommen.

Wenn es möglich ist, dass β „verschwindet“, endet A mit X und alle Terminalen, die in einer Satzform dem Nichtterminal A folgen, folgen nach Substitution von A auch X .



Definition 7.7

Für sämtliche Regeln^a der Form $A \rightarrow \alpha_1 X \beta_1 | \alpha_2 X \beta_2 | \dots | \alpha_n X \beta_n$ mit $\alpha_i, \beta_i \in (N \cup T)^*$ ist $FOLLOW(X)$ wie folgt definiert:

$$FOLLOW(X) = \begin{cases} \bigcup_{i=1}^n FIRST(\beta_i), & \text{falls } \varepsilon \notin \bigcup_{i=1}^n FIRST(\beta_i) \\ (\bigcup_{i=1}^n FIRST(\beta_i) \setminus \{\varepsilon\}) \cup FOLLOW(A), & \text{sonst} \end{cases}$$

^aWir gehen davon aus, dass die jeweils betrachtete Grammatik ausschließlich echte Regeln besitzt, also solche, deren Nichtterminal auf der linken Seite mindestens einmal in irgendwelcher rechten Regelseite vorkommen.

Eine besondere Rolle spielt das *Eingabeabschlusszeichen* $\$$, das das Ende des Eingabewortes markiert, vergleichbar mit eof (end of file) zur Kennzeichnung des Dateiendes. Bei der Berechnung der $FOLLOW$ -Menge wird es wie ein Terminal behandelt. In Abbildung 7.5 (2. Z. v. u.) taucht es in $FOLLOW(X)$ auf.

Eingabe-
abschlusszeichen

Beispiel 7.4

Um festzustellen, ob $G = (\{S, X, C, B\}, \{d, a\}, \{S \rightarrow X d, X \rightarrow B a \mid C, C \rightarrow \varepsilon, B \rightarrow d\}, S)$ die LL(1)-Forderung 2 erfüllt oder nicht, nehmen wir uns das Nichtterminal X vor: Da $X \xrightarrow{*} \varepsilon$ bilden wir $FIRST(X) = \{d, \varepsilon\}$ und $FOLLOW(X) = \{d\}$. Wegen $FIRST(X) \cap FOLLOW(X) = \{d\} \neq \emptyset$ ist die LL(1)-Forderung 2 verletzt.



Beispiel



Didaktischer Hinweis 7.2

Während die *FIRST*-Mengen-Bildung für Forderung 1 stets für Satzformen^a gebildet werden, geschieht die *FOLLOW*-Mengen-Bildung für (ausgewählte) Nichtterminale. Dies wird immer wieder gern verwechselt.

^aEine Satzform kann im Speziellen natürlich aus einem einzigen Nichtterminal bestehen.

Die Forderungen 1 und 2 stellen die definierenden Eigenschaften für *LL(1)*-Grammatiken dar. Eine *LL(1)*-Grammatik kann daher niemals linksrekursiv oder mehrdeutig sein.



Definition 7.8

Wir nennen eine kfG genau dann eine *LL(1)*-Grammatik, wenn sie die beiden Forderungen gemäß den Definitionen 7.2 und 7.5 erfüllt.



Beispiel 7.5

Im Folgenden verwenden wir FLACI, um für die Grammatik $G = (N, T, P, s)$ mit

$$\begin{aligned} N &= \{E, X, T, Y, F\} \\ T &= \{+, *, (,), a\} \\ P &= \{E \rightarrow T X, \\ &\quad X \rightarrow + T X \mid \varepsilon, \\ &\quad T \rightarrow F Y, \\ &\quad Y \rightarrow * F Y \mid \varepsilon, \\ &\quad F \rightarrow (E) \mid a\} \\ s &= E. \end{aligned}$$

zu bestätigen, dass sie eine *LL(1)*-Grammatik ist, s. auch Abbildung 7.5.

Die in Übungsaufgabe 7.2 angegebene Grammatik für die Sprache einfacher arithmetischer Ausdrücke verstößt offenkundig gegen *LL(1)*-Forderung 1 und ist deshalb keine *LL(1)*-Grammatik.



Übung 7.2

Stellen Sie zuerst durch „Rechnung“ und überprüfend mit FLACI fest, dass $G = (N, T, P, s)$ mit

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

keine *LL(1)*-Grammatik ist.

Überprüfung der $LL(1)$ Forderungen

- ✓ Forderung 1 ist erfüllt.
- ✓ Forderung 2 ist erfüllt.

$E \rightarrow TX$

Forderung 1 erfüllt:
 $\text{FIRST}(TX) = \{\langle, a\}$

Forderung 2 erfüllt:
 $\text{FIRST}(E) = \{\langle, a\}$, Da ϵ in der FIRST-Menge nicht vorkommt, ist Forderung 2 erfüllt.

$X \rightarrow +TX | \epsilon$

Forderung 1 erfüllt:
 $\text{FIRST}(+TX) = \{+\}$
 $\text{FIRST}(\epsilon) = \{\epsilon\}$

Forderung 2 erfüllt:
 $\text{FIRST}(X) = \{+, \epsilon\}$, $\text{FOLLOW}(X) = \{\$\}$,
 $\text{FIRST}(X) \cap \text{FOLLOW}(X) = \emptyset$

Abbildung 7.5: Prüfung der $LL(1)$ -Forderungen für G mit FLACI

Für kfG, die keine $LL(1)$ -Grammatiken sind, kann man versuchen, die gewünschte Form durch Transformation herzustellen, Abschnitt 7.6. Es ist also nicht aussichtslos, obgleich diese Transformation nicht für jede kfG möglich ist.

Ein lohnendes Ziel ist die Erstellung einer $LL(1)$ -Grammatik (falls sie existiert) allemal, denn für diese Sprachklasse gibt es sehr effiziente Parse-Verfahren, die wir in den folgenden Abschnitten behandeln werden. Deren Effizienz ist besser als die von CYK und Earley.

7.4 Top-down-Parser für $LL(1)$ -Grammatiken

Wir haben nun alles vorbereitet, um zu erklären, wie ein deterministisch arbeitender Top-down-Parser für $LL(1)$ -Sprachen arbeitet. Ein $LL(1)$ -Parser wird immer dort, wo es für ein Nichtterminal (X) mehrere alternative Übergänge $X \rightarrow \alpha_1 | \alpha_2 | \dots | \alpha_n$ gibt, mit $X \rightarrow \alpha_i$ genau die Regel auswählen, für die das aktuelle Vorausschauzeichen v in $\text{FIRST}(\alpha_i)$ enthalten ist.

Es müssen also zunächst die Mengen $\text{FIRST}(\alpha_1), \text{FIRST}(\alpha_2), \dots, \text{FIRST}(\alpha_i)$, $i \leq n$, berechnet werden, bis wir auf eine Menge $\text{FIRST}(\alpha_i)$ stoßen, die das aktuelle Vorausschauzeichen (Token) v enthält. Durch die $LL(1)$ -Forderung 1 ist ausgeschlossen, dass sich v in mehr als einer FIRST -Menge befindet. Gehört das betrachtete Token zu keiner dieser FIRST -Mengen, endet die Analyse von w mit dem Ergebnis $w \notin L(G)$.

deterministische
 $LL(1)$ -Parsing-
Strategie

vorgezogene
FIRST-Mengen-
Berechnung

Memoizing

Wenn wir diese Vorschrift strikt befolgen, erhalten wir einen *LL(1)*-Parser, der zwar deterministisch aber immer noch recht ineffizient arbeitet, da er zahlreiche *FIRST*-Mengen mehrfach berechnet. Es ist also sinnvoll, einmal berechnete *FIRST*s zu speichern, damit die Resultate später wiederverwendet werden können. Diese Methode nennt man *Memoizing*. Sie sichert, dass während der Analyse nur die wirklich erforderlichen *FIRST*-Mengen genau einmal ermittelt werden.

Mehrfachnutzung des Parsers für diverse Eingabewörter vorausgesetzt, wird diese mit Memoizing erzielte Effizienzverbesserung überboten, wenn wir alle *FIRST*-Mengen vor der Parsezeit (einmalig) berechnen. Damit wird die Analyse von diesen Berechnungen völlig befreit, da sämtliche Ergebnisse (in einer Tabelle; Hashmap) vorliegen und bei Bedarf direkt zugegriffen werden kann.



Beispiel 7.6

Wir betrachten eine kfS für symbolische Ausdrücke, wie sie z. B. in Sprachen der LISP-Familie (Lisp, Racket, Scheme) verwendet werden. Wörter, wie $((x_{x_x})_x)$, sollen zu dieser Sprache gehören. Die Suche nach einer dementsprechenden kfG ist rasch erfolgreich:

$$G = (\{S\}, \{x, (,), -\}, \{S \rightarrow x \mid (S) \mid (S - S)\}, S)$$

Mit FLACI kann man die Analyse des o.g. Eingabewortbeispiels mit dem Ergebnis $w \in L(G)$ durchführen, d.h. es entsteht ein Ableitungsbäum. Leider ist die Grammatik offensichtlich nicht *LL(1)*-konform, was man mit Prüfen in FLACI bestätigt.

$$G = (\{S, R\}, \{x, (,), -\}, \{S \rightarrow x \mid (S R ; R \rightarrow) \mid - S R \}, S)$$

ist eine *LL(1)*-Grammatik für die gleiche Sprache. Prüfen! Wir berechnen

für S : $FIRST(x) = \{x\}$, $FIRST((SR)) = \{\}\}$ und

für R : $FIRST(\)) = \{\}\}$, $FIRST(_SR) = \{-\}$.

Parsertabelle

Daraus ergibt sich die für die Parsersteuerung erforderliche *Parsertabelle* (parse table): Die

| | lookahead | | |
|---|-----------|---|--------|
| | x | (|) |
| S | x | (| $S R$ |
| R | |) | $_S R$ |

Tabelle 7.1: Parsertabelle für *LL(1)*-Grammatik

Spalten von Tabelle 7.1 werden durch die Terminale gebildet. In den Feldern stehen die jeweils anzuwendenden Regeln, allerdings nur die rechten Regelseiten. Die jeweils linke Seite wird durch die Zeilenangabe (Nichtterminal) bestimmt. Die Langform des Eintrags in Tabellenfeld [$R,$] ist also $R \rightarrow$) und bedeutet die Anwendung genau dieser Regel für R , wenn) das aktuelle Vorausschauzeichen ist. Leere Felder bedeuten Fehler und Abbruch des Analyseprozesses mit Ablehnung des betrachteten Eingabewortes.

Abbildung 7.6 protokolliert den weiter oben beschriebenen sackgassenfreien Ableitungsprozess mit Tabellensteuerung für das angegebene Eingabewort.

| Satzform | Angewandte Regel |
|---|-------------------------|
| S | $S \rightarrow (_ S R$ |
| $(_ S R$ | $S \rightarrow (_ S R$ |
| $(_ (_ S R R$ | $S \rightarrow x$ |
| $(_ (_ x _ R R$ | $R \rightarrow _ S R$ |
| $(_ (_ x _ _ S R R R$ | $S \rightarrow (_ S R$ |
| $(_ (_ x _ _ (_ S R R R R$ | $S \rightarrow x$ |
| $(_ (_ x _ _ (_ x _ R R R$ | $R \rightarrow _ S R$ |
| $(_ (_ x _ _ (_ x _ _ S R R R$ | $S \rightarrow x$ |
| $(_ (_ x _ _ (_ x _ _ x R R R$ | $R \rightarrow \}$ |
| $(_ (_ x _ _ (_ x _ _ x _ R R$ | $R \rightarrow \}$ |
| $(_ (_ x _ _ (_ x _ _ x _) R$ | $R \rightarrow _ S R$ |
| $(_ (_ x _ _ (_ x _ _ x _) _ S R$ | $S \rightarrow x$ |
| $(_ (_ x _ _ (_ x _ _ x _) _ _ x R$ | $R \rightarrow \}$ |
| $(_ (_ x _ _ (_ x _ _ x _) _ _ x _)$ | |

Abbildung 7.6: Satzformenfolge für das $LL(1)$ -parsing von $((x_)(x_x))_x$

Dieses $LL(1)$ -parsing mit Tabellensteuerung können wir auch aus der Sicht eines zu gehörigen NKA illustrieren. Hierfür wird die gegebene kfG ($LL(1)$ -Grammatik) in einen äquivalenten NKA konvertiert:

$$M = (\{q_0, q_1, q_2\}, \{(., , - , x), \{ \$, (,) , - , R, S, x \}, \delta, q_0, \$, \{q_2\} } \text{ mit}$$

$$\begin{aligned}\delta(q_0, \varepsilon, \$) &= \{(q_1, \$\$)\} \\ \delta(q_1, \varepsilon, \$) &= \{(q_2, \varepsilon)\} \\ \delta(q_1, (, () &= \{(q_1, \varepsilon)\} \\ \delta(q_1, (,)) &= \{(q_1, \varepsilon)\} \\ \delta(q_1, -, -) &= \{(q_1, \varepsilon)\} \\ \delta(q_1, x, x) &= \{(q_1, \varepsilon)\} \\ \delta(q_1, \varepsilon, S) &= \{(q_1, (S R), (q_1, x)\} \\ \delta(q_1, \varepsilon, R) &= (q_1,), (q_1, _S R)\}\end{aligned}$$

Für das Eingabewort $((x_)(x_x))_x$ ergibt sich die in Tabelle 7.2 ausschnittsweise wie dergegebene Konfigurationenfolge. In Spalte 3 wird das jeweilige Vorausschauzeichen angegeben, das den im aktuellen Schritt verwendeten Übergang eindeutig bestimmt.

In den Zeilen 2 und 4 von Tabelle 7.2 sieht man, dass das Vorausschauzeichen $($ den

| Zustand | Eingabe | Vorausschau | Keller | Übergang |
|----------|--------------------|-------------|------------|---|
| q_0 | $((x_{-}(x_x))_x)$ | (| \$ | $\delta(q_0, \epsilon, \$) = (q_1, S\$)$ |
| q_1 | $((x_{-}(x_x))_x)$ | (| S\$ | $\delta(q_1, \epsilon, S) = (q_1, (SR))$ |
| q_1 | $((x_{-}(x_x))_x)$ | (| (SR\$ | $\delta(q_1, (, () = (q_1, \epsilon)$ |
| q_1 | $(x_{-}(x_x))_x)$ | (| SR\$ | $\delta(q_1, \epsilon, S) = (q_1, (SR))$ |
| q_1 | $(x_{-}(x_x))_x)$ | (| (SRR\$ | $\delta(q_1, (, () = (q_1, \epsilon)$ |
| q_1 | $x_{-}(x_x)_x)$ | x | SRR\$ | $\delta(q_1, \epsilon, S) = (q_1, x)$ |
| q_1 | $x_{-}(x_x)_x)$ | x | xRR\$ | $\delta(q_1, x, x) = (q_1, \epsilon)$ |
| \vdots | \vdots | \vdots | \vdots | \vdots |
| q_1 |) |) |)\$ | $\delta(q_1,),) = (q_1, \epsilon)$ |
| q_1 | ϵ | | \$ | $\delta(q_1, \epsilon, \$) = (q_2, \epsilon)$ |
| q_2 | ϵ | | ϵ | Akzeptiert! |

Tabelle 7.2: $LL(1)$ -Parsing unter Verwendung der Parser-Tabelle 7.1

Übergang $\delta(q_1, \epsilon, S) = (q_1, (SR))$ verlangt, s. Tabelle 7.1: $Tabelle[S, ()] = (SR)$.

Das Verfahren lässt sich folgendermaßen beschreiben: Das in jedem Arbeitstakt aktuelle oberste Kellerzeichen (top of stack, kurz: tos) entspricht entweder einem Terminal (Token) oder einem Nichtterminal der Grammatik.

Handelt es sich um ein Terminal, muss es mit dem aktuellen Zeichen des Eingabewortes w übereinstimmen. Andernfalls endet die Analyse mit $w \notin L(G)$. Ist das tos ein Nichtterminal, so entnehmen wir Tabelle 7.1 die Satzform, die das aktuelle tos im Keller substituiert.



Computerübung 7.1

Definieren Sie $G = (\{S, R\}, \{x, (,), _, \}, \{S \rightarrow x \mid (S R; R \rightarrow) \mid _ S R \}, S)$ aus Beispiel 7.6 mit FLACI und betrachten Sie den Ableitungsbaum für $((x_{-}(x_x))_x)$. Konvertieren Sie diese $LL(1)$ -Grammatik G in einen äquivalenten NKA mit FLACI und betrachten Sie die Konfigurationenfolge für $((x_{-}(x_x))_x)$.

Greifen Sie aus den 14 angegebenen Konfigurationenfolgen genau die (hervorgehobene) Folge heraus, die zur Akzeptanz des Wortes führt. Diese sollte exakt mit der in Tabelle 7.2 angedeuteten Konfigurationenfolge für $((x_{-}(x_x))_x)$ mit Vorausschau genau eines Zeichens übereinstimmen. Vervollständigen Sie die 28 Schritte dieser Konfigurationenfolge.

Hinweis: An dieser Stelle ist es ggf. möglich, die Erzeugung eines $LL(1)$ -Parsers für die Grammatik $G = (\{S, R\}, \{x, (,), _, \}, \{S \rightarrow x \mid (S R; R \rightarrow) \mid _ S R \}, S)$ mit FLACI (Modul: „Compiler und Interpreter“) vorzuführen. Man wähle $LL(1)$.

Wir haben noch nicht formalisiert, wie $LL(1)$ -Parsestabellen vom Parsergenerator hergestellt werden. Das *Konstruktionsprinzip* ist unter Rückgriff auf *FIRST* und *FOLLOW* sofort verständlich: Für jede Regel $A \rightarrow \alpha$ in P gilt bzw. gelten:

Konstruktion von $LL(1)$ -Parsestabellen

- Wenn $\alpha \stackrel{*}{\Rightarrow} a\beta$, d.h. $a \in FIRST(\alpha)$, schreibe $A \rightarrow \alpha$ in die Felder $Tabelle[A, a]$, für alle diese Terminals a .

- Wenn $\alpha \xrightarrow{*} \epsilon$, d.h. $\epsilon \in FIRST(\alpha)$, dann betrachte $FOLLOW(A)$, wobei $\$ \in FOLLOW(A)$ ¹ gelten kann. Schreibe $A \rightarrow \alpha$ in die Felder $Tabelle[A, b]$, mit $b \in FOLLOW(A)$.
- Sämtliche noch freien Felder der Zeile für A bleiben leer.

Sollten sich bei Anwendung dieser Vorschrift mehrere Regeleinträge in (wenigstens) ein Tabellenfeld erforderlich machen, so ist die betrachtete Grammatik nicht vom $LL(1)$ -Typ.

Beispiel 7.7

Für die $LL(1)$ -Grammatik $G = (\{E, E', T, T', F\}, \{+, *, (,), a\}, P, E)$, mit den Produktionen $P = \{E \rightarrow TE', E' \rightarrow +TE', E' \rightarrow \epsilon, T \rightarrow FT', T' \rightarrow *FT', T' \rightarrow \epsilon, F \rightarrow a, F \rightarrow (E)\}$ ergibt sich folgende Parsestabelle:



| Parse-tabelle | lookahead | | | | | |
|---------------|---------------------|---------------------------|-----------------------|---------------------|---------------------------|---------------------------|
| | a | + | * | (|) | \$ |
| E | $E \rightarrow TE'$ | | | $E \rightarrow TE'$ | | |
| E' | | $E' \rightarrow +TE'$ | | | $E' \rightarrow \epsilon$ | $E' \rightarrow \epsilon$ |
| T | $T \rightarrow FT'$ | | | $T \rightarrow FT'$ | | |
| T' | | $T' \rightarrow \epsilon$ | $T' \rightarrow *FT'$ | | $T' \rightarrow \epsilon$ | $T' \rightarrow \epsilon$ |
| F | $F \rightarrow a$ | | | $F \rightarrow (E)$ | | |

Didaktischer Hinweis 7.3

In der Tabelle in Beispiel 7.7 erkennt man eine gewisse Redundanz, auf die in Bezug auf Tabelle 7.1 bereits hingewiesen, dort jedoch unterdrückt, wurde: Die Einträge in den Tabellenfeldern bestehen aus vollständigen Regeln. Dabei könnten sämtliche linken Regelseiten weggelassen werden, denn sie stimmen mit dem ganz links in der jeweiligen Zeile angegebenen Nichtterminal überein. In der Praxis wird von dieser verkürzten Schreibweise natürlich Gebrauch gemacht, auch bei entsprechenden Implementierungen.



Übung 7.3

Verwenden Sie die $LL(1)$ -Grammatik G und die Parsestabelle aus Beispiel 7.7 und protokollieren Sie den Analyseprozess für das Wort a^*a+a unter Angabe des jeweils aktuellen Stapelinhals, des entsprechenden Restwort im Eingabepuffer und der anzuwendenden Produktion:



| Keller | Eingabewort | Regel |
|--------|-------------|---------------------|
| $E \$$ | $a * a + a$ | $E \rightarrow TE'$ |

Übung 7.4

Konstruieren Sie die $LL(1)$ -Parsestabelle aus Beispiel 7.7 nach der allgemeinen Vorschrift zur $LL(1)$ -Parsegenerierung und vergleichen Sie Ihr Ergebnis mit der in Beispiel 7.7 angegebenen Tabelle.



¹Dies bedeutet, dass das Eingabeabschlusszeichen wie ein Terminal behandelt wird.



Übung 7.5

Für die in Beispiel 7.6 verwendete Grammatik soll eine tabellengesteuerte Syntaxanalyse für das Wort (x_x_x) durchgeführt werden. Bereiten Sie die $LL(1)$ -Parse-Tabelle vor und protokollieren Sie den Analyseprozess.

7.5 Methode des Rekursiven Abstiegs

Anstelle die Syntaxanalyse von $LL(1)$ -Sprachen durch einen deterministisch arbeitenden tabellengesteuerten NKA zu bewerkstelligen, gibt es eine naheliegende Idee, das Parsing-Verfahren direkt aus der Grammatik abzuleiten. Dies führt zu der Methode des *rekursiven Abstiegs* (recursive descent).

| | |
|----------------------|---|
| recursive descent | <p>Nichtterminal</p> <p>Die Grundidee besteht darin, zu jedem Nichtterminal eine parameterfreie (nullstellige) Prozedur (Methode) anzugeben, deren Definition den Regeln für dieses Nichtterminal entspricht: Für jedes Nichtterminal <i>auf der rechten Regelseite</i> entsteht dann der zugehörige Prozederaufruf. Im Verlauf der Analyse können sich auf diese Weise völlig „verstrickte“ (direkt und indirekt) rekursive Aufrufe ergeben. Eine Implementierung dieses Verfahrens setzt eine Sprache voraus, die Rekursion anbietet, was heute eher selbstverständlich ist. In FLACI ist das JavaScript.</p> |
| Terminal | <p>Für ein Terminal gibt es keinen Regelauftrag. In diesem Falle muss das betreffende Symbol mit dem ersten Zeichen des Eingaberestwertes übereinstimmen. Trifft dies zu, so wird es an aktueller Position im Eingabepuffer entfernt, anderenfalls muss die Analyse mit einer Fehlermeldung beendet werden. Der Analyseprozess beginnt mit dem Aufruf der zum Spitzensymbol gehörenden Prozedur.</p> <p>In einem solchen Fehlerfall ist zu beachten, dass es sich um einen Sofortabbruch handelt, der nicht etwa die bisher stattgefundene rekursive Abarbeitung rückkehrend bedient. Fortgeschrittene <i>Fehlerbehandlung</i> (error recovery) ist einem Spezialkurs über Compilerbau vorbehalten.</p> |
| Produktionen | <p>Mehrere Produktionen für ein und dasselbe Nichtterminal erscheinen als alternative Fälle in den zugehörigen Prozeduren bzw. Methoden. Das jeweils aktuelle Vorausschauzeichen bestimmt in einer Verzweigung, welcher dieser Fälle ausgewählt wird. Aufgrund der $LL(1)$-Eigenschaften ist gesichert, dass in jedem Schritt genau eine der angegebenen Alternativen zutrifft. Analog zum „NKA mit Vorausschau“ ist zu beachten, dass es sich bei der Vorausschau <i>nicht</i> um verbrauchendes Lesen handelt, so dass es dadurch zu <i>keiner</i> Veränderung des Wortes im „Eingabepuffer“ kommt.</p> |
| Parsergenerator | <p>Die Methode des rekursiven Abstiegs ist für $LL(1)$-Sprachen ein etabliertes Verfahren. Da die Regeln in der oben beschriebenen Art und Weise in entsprechende Prozeduren überführt werden, ergibt sich ein transparenter <i>Parsergenerator</i>. Dieser nimmt eine $LL(1)$-Grammatik und generiert den zugehörigen Parser. FLACI</p> |

stellt einen *LL(1)*-Parsergenerator im Modul „Compiler und Interpreter“ bereit.

Von NIKLAUS WIRTH übernehmen wir eine formale Grammatik für die Sprache PL/0. Die großbuchstabigen Wörter sind die Nichtterminale. Die Terminatele sind die Symbole `begin` `call` `procedure` `end` `while` `do` `if` `then` `const` `var` `? !` `odd` `+ - * / < <= = > >= # () . := , ; .`. Weiterhin gehören alle Ziffernsymbole (0...9) und die Kleinbuchstaben (a...z) zur Menge der Terminatele.

Die übrigen Zeichen, die zur Angabe der Produktionen in *P* dienen, sind gängige Metazeichen der *EBNF*: Die mit einer geschweiften Klammer gruppierten Satzglieder dürfen beliebig oft wiederholt oder auch weggelassen werden. Eckige Klammern markieren optionale Satzformen, die entweder genau einmal an dieser Stelle auftreten oder vollständig fehlen.

Die Regelmenge *P* ist wie folgt definiert.

```

PROGRAM    -> BLOCK .
BLOCK      -> [ const IDENT = NUMBER { , IDENT = NUMBER } ; ]
                  [ var IDENT { , IDENT } ; ]
                  { procedure IDENT ; BLOCK ; }
                  STATEMENT
STATEMENT  -> [ IDENT := EXPRESSION | call IDENT |
                  ? IDENT | ! EXPRESSION |
                  begin STATEMENT { ; STATEMENT } end |
                  if CONDITION then STATEMENT |
                  while CONDITION do STATEMENT ]
CONDITION   -> odd EXPRESSION | EXPRESSION RELATION EXPRESSION
RELATION    -> = | # | < | > | <= | >=
EXPRESSION  -> [ + | - ] TERM { SIGN TERM }
SIGN        -> + | -
TERM         -> FACTOR { OPERATION FACTOR }
OPERATION   -> * | /
FACTOR       -> IDENT | NUMBER | ( EXPRESSION )
CHARACTER   -> a | b | c | ... | z
IDENT        -> CHARACTER | CHARACTER IDENT
NUMBER       -> DIGIT | DIGIT NUMBER
DIGIT        -> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

```

Das folgende PL/0-Programmbeispiel wurde ebenfalls von WIRTH übernommen.

Beispielsprache
PL/0

EBNF

PL/0-
Programmbeispiel

```
var x, y, z, q, r, n, f;
procedure multiply;
    var a, b;
    begin
        a := x; b := y; z := 0;
        while b > 0 do
            begin
                if odd z then z := z + a;
                a := 2 * a;
                b := b/2
            end
    end;
procedure divide;
    var w;
    begin
        r := x; q := 0; w := y;
        while w <= r do w := 2 * w;
        while w > y do
            begin
                q := 2 * q; w := w / 2;
                if w <= r then
                    begin r := r - w; q := q + 1
                    end
            end
    end;
procedure gcd;
    var f, g;
    begin
        f := x; g := y;
        while f # g do
            begin
                if f < g then g := g - f;
                if g < f then f := f - g
            end;
        z := f
    end;
procedure fact;
begin
    if n > 1 then
        begin
            f := n * f;
            n := n - 1;
            call fact
        end
end
```

```

    end;
begin
?x; ?y; call multiply; !z;
?x; ?y; call divide; !q; !r;
?x; ?y; call gcd; !z;
?n; f := 1; call fact; !f
end.

```

Zur Erhöhung der Lesbarkeit des Programmtextes können Leerzeichen zwischen je zwei Terminalen eingefügt werden. Sie sind ohne syntaktische Relevanz.

Da FLACI eine Grammatik grundsätzlich in BNF erwartet, eliminieren wir die o.g. EBNF-Metazeichen durch entsprechende Transformationen. Wir illustrieren dies an der Regel

BLOCK → [const IDENT = NUMBER {, IDENT = NUMBER } ;].

Die BNF benötigt zusätzliche Nichtterminale. Dies sind hier CD?, CONST und CONST2. Mit Blick auf FLACI schreiben wir bereits EPSILON für ϵ .

EBNF \rightsquigarrow BNF

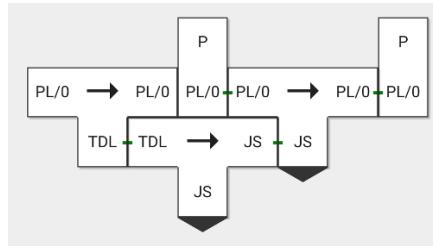
| | |
|--------|--------------------------------------|
| BLOCK | -> CD? |
| CD? | -> CONST EPSILON |
| CONST | -> const IDENT = NUMBER CONST2 ; |
| CONST2 | -> , IDENT = NUMBER CONST2 EPSILON |

In der gesamten Grammatik können wir die vier ursprünglich vorhandenen Nichtterminale, nämlich DIGIT und NUMBER sowie CHARACTER und IDENT, zusammen mit den Regeln in denen sie vorkommen, streichen. Genauer gesagt: Wir übergeben deren Beschreibungen dem *Scannergenerator* (Lexer). Der von diesem generierte Scanner erkennt diese Token, so dass sie der Parser als Terminaler versteht. Es entsteht die folgende Regelmenge der nun reduzierten Grammatik in BNF.

Scannergenerator

| | |
|-------------------|---|
| PROGRAM | -> BLOCK . |
| BLOCK | -> CD? VD? PDs STATEMENT |
| CONST-DECLARATION | -> const IDENT = NUMBER IDNs ; |
| CD? | -> CONST-DECLARATION EPSILON |
| VAR-DECLARATION | -> var IDENT IDENTs ; |
| VD? | -> VAR-DECLARATION EPSILON |
| PROC-DEFINITION | -> procedure IDENT ; BLOCK ; |
| PDs | -> PROC-DEFINITION PDs EPSILON |
| STATEMENT | -> STMTBODY EPSILON |
| STMTBODY | -> ? IDENT IDENT := EXPRESSION begin STATEMENT STATEMENTS end |

Greduziert in BNF

Abbildung 7.7: $PL/0 \rightarrow PL/0$ -Compiler mit LL(1)-Parsergenerator

| | |
|------------|------------------------------------|
| | call IDENT |
| | if CONDITION then STATEMENT |
| | while CONDITION do STATEMENT |
| | ! EXPRESSION |
| STATEMENTS | -> ; STATEMENT STATEMENTS |
| | EPSILON |
| CONDITION | -> odd EXPRESSION |
| | EXPRESSION RELATION EXPRESSION |
| RELATION | -> # < <= = > >= |
| EXPRESSION | -> SIGN? TERM SIGTERMs |
| SIGTERMs | -> SIGN TERM SIGTERMs EPSILON |
| SIGN | -> + - |
| SIGN? | -> SIGN EPSILON |
| TERM | -> FACTOR OPFACs |
| OPFACs | -> OPERATION FACTOR OPFACs |
| | EPSILON |
| OPERATION | -> * / |
| FACTOR | -> (EXPRESSION) IDENT NUMBER |
| IDENTs | -> , IDENT IDENTs EPSILON |
| IDNs | -> , IDENT = NUMBER IDNs |
| | EPSILON |

Computerübung 7.2

Definieren Sie diese reduzierte Grammatik (in BNF) mit FLACI und bekräftigen Sie, dass $G_{reduziert}$ eine LL(1)-Grammatik ist.

automatisierte
Generierung
eines
 $LL(1)$ -Parsers
mit FLACI

Wir verwenden $G_{reduziert}$, um im FLACI-Modul „Compiler und Interpreter“ einen „neuen Compiler (mit TDL von Grammatik)“ zu erzeugen. Als Generator dürfen wir nun LL(1) auswählen. LALR(1) ist voreingestellt. Unter Verwendung des Parsergenerators $TDL \rightarrow JS$ erzeugen wir einen $PL/0$ -Parser, genauer gesagt, einen $PL/0 \rightarrow PL/0$ -Compiler, s. Abbildung 7.7, den wir später für eine andere Zielsprache qualifizieren können.

Ein Blick in die generierte JS-Codierung des $PL/0$ -Parser lässt die LL(1)-Struktur

des rekursiven Abstiegs etwa an Definitionen 0-stelliger Funktionen für Nichtterminale sehr gut erkennen: `function rule_PROGRAM () { ... }`,

Schließlich müssen noch die Tokenklassen-Definitionen für NUMBER und IDENT durch reguläre Ausdrücke (in Praxisnotation) für den Scannergenerator hinzugefügt werden: `[0-9] [0-9]*` bzw. `[a-z] [a-z]*`. Damit Schlüsselworte, wie `begin`, nicht als Lexem der Tokenklasse IDENT „erkannt“ werden, verschieben wir die Definition von IDENT weit nach unten. Da die Suche von oben beginnt, wird damit den Schlüsselworten gegenüber Variablennamen Vorrang eingeräumt.

Außerdem empfiehlt sich wieder die Hinzunahme von IGNORE mit den möglichen Lexemen `\t\r\n\$`.



Computerübung 7.3

Folgen Sie bei der PL/0-Parserentwicklung den oben angegebenen Erläuterungen. Orientieren Sie sich an dem Übersetzungsschema (Modellierung mit T-Diagrammen) aus Abbildung 7.7. Testen Sie den Compiler und bauen Sie im Beispiel-PL/0-Programm absichtlich einen Syntaxfehler ein.



Computerübung 7.4

Betrachten Sie die folgende Grammatik: $G = (\{E, T, F\}, \{a, (,), +, -, *, /\}, P, E)$ mit den in EBNF notierten Regeln: $E \rightarrow T\{[+ | -]T\}$, $T \rightarrow F\{[* | /]F\}$, $F \rightarrow (E) \mid a$. Transformieren Sie als erstes die Regeln in BNF. Wie auf S. 187 bedeuten

`{}` ... beliebig oft oder keinmal und

`[]` ... einmal oder keinmal.

Die Transformationsregeln lauten demnach:

- Transformiere $X \rightarrow \{Y\}$ zu $X \rightarrow YX \mid \varepsilon$ und
- Transformiere $X \rightarrow [Y]$ zu $X \rightarrow Y \mid \varepsilon$.

Verwenden Sie den von FLACI generierten $LL(1)$ -Parser, um festzustellen, ob das Wort `a*(a+a)` zu $L(G)$ gehört.

7.6 Grammatiktransformationen

Wir haben gesehen, dass Rekursive-Abstiegs-Parser ein leicht zu programmierendes effizientes Syntaxanalyseverfahren darstellen. Ihre Anwendung setzt $LL(1)$ -Grammatiken voraus. Ist eine betrachtete kfG nicht $LL(1)$ -Grammatik, weil sie gegen eine oder beide Forderungen verstößt, ist noch nicht alles verloren. Man kann dann versuchen, eine äquivalente $LL(1)$ -Grammatik zu finden. Dafür gibt es verschiedene Transformationen, wie beispielsweise die Herstellung der ε -Freiheit, die wir bereits in Satz 2.3 und dessen Beweis kennengelernt haben.

Im Folgenden behandeln wir drei typische Transformationen, deren Anwendungen im Allgemeinen gute Erfolgsaussichten haben.

- die Beseitigung linksrekursiver Produktionen
- die Beseitigung von Mehrdeutigkeit
- die Beseitigung gemeinsamer Präfixe durch Linksfaktorisierung

zyklenfrei,
ohne ϵ -Regeln

$LL(1)$ -Sprachen können niemals linksrekursive Regeln enthalten. Besitzt eine zyklenfreie² kfG G ohne ϵ -Regeln³ wenigstens eine linksrekursive Produktion, muss diese eliminiert werden. Die Frage, ob die gewünschte Transformation immer möglich ist, beantwortet der folgende Satz 7.1 positiv. Auch die Herstellung einer ϵ - und zyklenfreien kfG ist immer möglich. Auf Seite 165 findet sich das Transformationsverfahren zur Kettenregel-Eliminierung.



Satz 7.1

Zu jeder kfG G gibt es eine äquivalente kfG G' ohne Linksrekursivität.

Beweis

direkte
Linksrekursivität

Mit $\alpha, \beta \in (N \cup T)^*$ und β beginnt nicht mit $X \in N$, wenn $(X \rightarrow \beta) \in P$, können *direkt-linksrekursive Regeln* der Form

$$X \rightarrow X\alpha \mid \beta$$

am einfachsten eliminiert werden. Offenbar gilt: $X \xrightarrow{*} \beta\alpha\alpha\dots\alpha$. Die aus X ableitbaren Satzformen können auch durch die folgenden drei Regeln beschrieben werden.

$$\begin{array}{rcl} X & \rightarrow & \beta X' \\ X' & \rightarrow & \alpha X' \mid \epsilon \end{array}$$

X' ist ein neu hinzugefügtes Nichtterminal.

Diese Transformationsidee können wir nun für den Fall verallgemeinern, dass es für X mehrere linksrekursive Regeln gibt:

$$X \rightarrow X\alpha_1 \mid X\alpha_2 \mid \dots \mid X\alpha_m \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_n,$$

Das Ergebnis lautet

$$X \rightarrow \beta_1 X' \mid \beta_2 X' \mid \dots \mid \beta_n X' \text{ und } X' \rightarrow \alpha_1 X' \mid \alpha_2 X' \mid \dots \mid \alpha_m X' \mid \epsilon.$$

(Beweis wird fortgesetzt!) □



Beispiel 7.8

$$G = (\{A, S\}, \{a, b, c, d\}, \{S \rightarrow Aa \mid b, A \rightarrow Ac \mid Sd \mid \epsilon\}, S)$$

Zuerst muss ϵ -Freiheit hergestellt werden. Dies führt zu

$$G' = (\{A, S\}, \{a, b, c, d\}, \{S \rightarrow Aa \mid b \mid a, A \rightarrow Ac \mid Sd \mid c\}, S).$$

²Zyklenfrei bedeutet, dass es für kein einziges Nichtterminal X einer kfG eine Ableitung der Form $X \xrightarrow{*} X$ gibt. Dies ist dann der Fall, wenn eine ϵ -freie kfG keine Kettenregeln besitzt.

³Eine kfG nennt man ϵ -frei, wenn sie keine Regel der Form $X \rightarrow \epsilon$ besitzt, bis auf den Fall, dass X das Startsymbol der Grammatik ist. Falls $s \rightarrow \epsilon$ in P existiert, darf s auf keiner rechten Seite einer beliebigen Regel in P vorkommen.

Danach beseitigen wir die Linksrekursion in der Regel $A \rightarrow Ac$, was schließlich $G'' = (\{A, A', S\}, \{a, b, c, d\}, \{S \rightarrow Aa \mid b \mid a, A \rightarrow SdA' \mid cA', A' \rightarrow cA' \mid \varepsilon\}, S)$ ergibt.

Nun kann wieder ε -Freiheit hergestellt werden.

$G''' = (\{A, A', S\}, \{a, b, c, d\}, \{S \rightarrow Aa \mid b \mid a, A \rightarrow SdA' \mid Sd \mid cA' \mid c, A' \rightarrow cA' \mid c\}, S)$.

Die Transformation der ursprünglichen Grammatik ist damit allerdings noch nicht getan, denn es gibt *indirekte Linksrekursivität*. Diese lässt sich im Allgemeinen nicht durch „scharfes Hinsehen“ aufspüren. In unserem Beispiel ist dies jedoch möglich: $S \Rightarrow Aa \Rightarrow Sda$.

Beweis

(Fortsetzung des Beweises von Satz 7.1)

Der folgende Algorithmus nach PAULL transformiert jede zyklusfreie kfG ohne ε -Produktionen in eine äquivalente linksrekursionsfreie kfG. Eine Vorbehandlung durch Beseitigung direkter Linksrekursivität ist nicht erforderlich.

- 1: Benenne die Nichtterminale der Grammatik mit X_1, X_2, \dots, X_n .
- 2: **for** $i = 1, \dots, n$ **do**
- 3: **for** $j = 1, \dots, i - 1$ **do**
- 4: // Innerer Zyklus wird erst ab $i = 2$ durchlaufen.
- 5: Ersetze alle Regeln der Form $X_i \rightarrow X_j \gamma$ durch $X_i \rightarrow \delta_1 \gamma \mid \delta_2 \gamma \mid \dots \mid \delta_k \gamma$,
- 6: wobei $X_j \rightarrow \delta_1 \mid \delta_2 \mid \dots \mid \delta_k$ alle aktuellen Regeln für X_j in P sind.
- 7: **end for**
- 8: Beseitige direkte Linksrekursivität in den X_i -Regeln (nicht in allen Regeln!).
- 9: **end for**

Zur Herstellung der ε -Freiheit steht ein Verfahren bereit. Die Zyklensfreiheit kann durch Kettenregel-Eliminierung, s. Seite 165, hergestellt werden: Besitzt eine ε -freie kfG Zyklen der Form $\underline{X} \Rightarrow A_i \Rightarrow A_j \Rightarrow \dots \Rightarrow A_k \Rightarrow \underline{X}$ mit $X, A_i, A_j, \dots, A_k \in N$, muss es Kettenregeln der Form $A_i \rightarrow A_j$ geben. Diese können wir mit dem zugehörigen Verfahren eliminieren.

Auf die so vorbereitete kfG kann der obige Algorithmus erfolgreich angewandt werden.

Durch die Schleifen werden Regeln erzeugt, deren Nichtterminale auf der jeweils rechten Regelseite einen Index haben, der mindestens so groß ist wie der auf der linken Seite. D.h. die Nichtterminale mit kleinerem Index werden ersetzt. \square

indirekte
Linksrekursivität

PAULL's
Algorithmus

Didaktischer Hinweis 7.4

In der Literatur findet man Hinweise darauf, dass die Effizienz des Paull's Algorithmus' für bestimmte Grammatiken unzureichend ist. Dies ist jedoch erst dann relevant, wenn das Verfahren in der Praxis eingesetzt werden soll.



Beispiel 7.9

Wir greifen die kfS aus dem vorangehenden Beispiel wieder auf und beginnen mit der zyklischen- und ε -freien kfG $G = (\{A, S\}, \{a, b, c, d\}, \{S \rightarrow Aa \mid b \mid a, A \rightarrow Ac \mid Sd \mid c\}, S)$.



Nun erfolgt die Umbenennung der Nichtterminale: S wird X_1 und A wird X_2 . Dadurch entstehen die Regeln: $X_1 \rightarrow X_2a \mid b \mid a$, $X_2 \rightarrow X_2c \mid X_1d \mid c$.

Wir wenden nun den Paull's Algorithmus an: Für $i = 1$ wird der innere Zyklus nicht durchlaufen. Auch in Zeile 8 ist nichts zu tun, da es keine (direkt) linksrekursive Regel der Form $X_1 \rightarrow X_1 \gamma$ gibt. Also nehmen wir uns den Fall $i = 2$ und $j = 1$ vor: Wir müssen

$X_2 \rightarrow X_1d$ durch $X_2 \rightarrow X_2ad \mid bd \mid ad$ ersetzen. Dies ergibt insgesamt $X_1 \rightarrow X_2a \mid a \mid b$ und $X_2 \rightarrow X_2c \mid X_2ad \mid bd \mid ad \mid c$. Wir beseitigen die beiden direkt-linksrekursiven Regeln für X_2 nach der im Beweisanfang angegeben Transformation und erhalten insgesamt

$$\begin{aligned} X_1 &\rightarrow X_2a \mid b \mid a \\ X_2 &\rightarrow bdX_3 \mid adX_3 \mid cX_3 \\ X_3 &\rightarrow cX_3 \mid adX_3 \mid \epsilon \end{aligned}$$

Nach optionaler Herstellung der ϵ -Freiheit erhält man eine verbesserte Grammatik $G' = (\{X_1, X_2, X_3\}, \{a, b, c, d\}, P, X_1)$, mit folgender Regelmenge P :

$$\begin{aligned} X_1 &\rightarrow X_2a \mid b \mid a \\ X_2 &\rightarrow bdX_3 \mid adX_3 \mid bd \mid ad \mid cX_3 \mid c \\ X_3 &\rightarrow cX_3 \mid adX_3 \mid c \mid ad \end{aligned}$$

G' ist *nicht* unbedingt eine LL(1)-Grammatik.

Computerübung 7.5



Gegeben ist $G = (N, T, P, s)$, $N = \{E, T, F\}$, $T = \{a, +, *, (,)\}$, $s = E$, mit $P = \{E \rightarrow E + T \mid T, \quad T \rightarrow T * F \mid F, \quad F \rightarrow (E) \mid a\}$. Analysieren Sie das Wort $a + a * a$ mit FLACI. Bewerten Sie die entsprechenden Ableitungsbäume. Prüfen Sie, ob G eine LL(1)-Grammatik ist, und begründen Sie Ihre Entscheidung. Geben Sie eine zu G äquivalente Grammatik G' ohne Linksrekursionen an. Weisen Sie die LL(1)-Eigenschaft von G' nach.

Wie das bereits in Computerübung 2.8 auf Seite 40 angesprochene *dangling-else-Problem* zeigt, ist die Beseitigung der Linksrekursivität noch keine Garantie dafür, dass eine LL(1)-Grammatik entsteht.

Für die in vielen (imperativen) Programmiersprachen vorhandene *if-then-else*-Anweisung gibt es verschiedene Definitionsmöglichkeiten. Eine naheliegende kfG ist $G = (\{S\}, \{\text{if, b, then, else, other}\}, P, S)$ mit

$$P = \{S \rightarrow \text{if } b \text{ then } S \mid \text{if } b \text{ then } S \text{ else } S \mid \text{other}\}.$$

Diese Grammatik ist mehrdeutig: Abbildung 7.8 zeigt zwei strukturell verschiedene Ableitungsbäume für die Syntaxanalyse des Wortes

if b then if b then other else other .

Die folgende kfG $G = (\{S, S'\}, \{\text{if, b, then, else, other}\}, P, S)$ mit $P =$

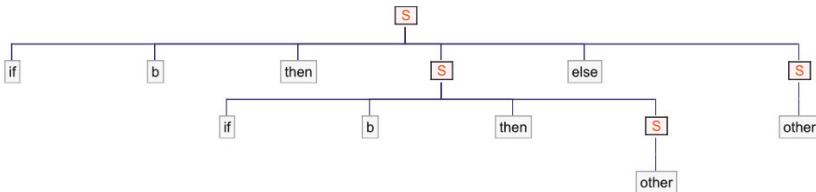
$$\begin{aligned} \{S &\rightarrow \text{if } b \text{ then } S' \mid \text{other} \\ S' &\rightarrow \text{else } S \mid \epsilon\} \end{aligned}$$

ist ebenfalls mehrdeutig. Für obiges Eingabewort liefert sie sogar drei verschiedene Ableitungsbäume.

Mit Hilfe der zusätzlichen Festlegung „else gehört zum nächstgelegenen if“ gelingt es für *if-then-else*-Ausdrücke eine eindeutige Grammatik anzugeben, so dass folglich nur der zweite Ableitungsbau in Abbildung 7.8 der richtige ist.

Unter Berücksichtigung dieser Festlegung kann die folgende *eindeutige* Gramma-

Ableitungsbaum 1 von 2:



Ableitungsbaum 2 von 2:

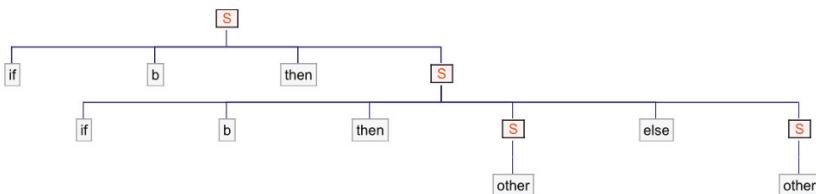


Abbildung 7.8: Mehrdeutige Grammatik für die if-then-else-Anweisungen

tik angegeben werden: $G = (\{S, S1, S2\}, \{\text{if}, \text{b}, \text{then}, \text{else}, \text{other}\}, P, S)$ mit $P =$

$$\begin{aligned} S &\rightarrow S1 \mid S2 \\ S1 &\rightarrow \text{if } b \text{ then } S1 \text{ else } S1 \mid \text{other} \\ S2 &\rightarrow \text{if } b \text{ then } S \mid \text{if } b \text{ then } S1 \text{ else } S2 \end{aligned}$$

Gelingt eine solche Transformation für jede kfS? Die Antwort ist leider: Nein. Bei sog. *inhärent mehrdeutigen Sprachen* haben wir keine Chance, eine eindeutige Grammatik zu finden. Dabei handelt es sich um kfS, für deren Definition *auschließlich* mehrdeutige kfG angegeben werden können.

Wie man (mit FLACI) leicht überprüft, ist diese Grammatik zwar eindeutig aber keine LL(1)-Grammatik. Es gibt auch keine. Im Web findet man dennoch verschiedene Vorschläge, wie etwa

$G = (\{S, S1, optionaltail, tail\}, \{\text{if}, \text{b}, \text{then}, \text{else}, \text{other}\}, P, S)$ mit $P =$

$$\begin{aligned} S &\rightarrow \text{if } b \text{ then } S1 \text{ optionaltail } \mid \text{other} \\ S1 &\rightarrow \text{if } b \text{ then } S1 \text{ else } S1 \mid \text{other} \\ optionaltail &\rightarrow \text{else tail } \mid \epsilon \\ tail &\rightarrow \text{if } b \text{ then tail } \mid \text{other} \end{aligned}$$

Dies ist tatsächlich eine LL(1)-Grammatik, allerdings nicht für die hier betrachtete Sprache. Beispielsweise wird das korrekte Wort

inhärent
mehrdeutige
Sprachen

if b then if b then if b then other else other
nicht akzeptiert.

Eine LL(1)-Grammatik für die zweiseitige Entscheidung kann nur erzielt werden, wenn man die konkrete Syntax (leicht) verändert, sodass sie die weiter oben besprochene Zugehörigkeit von `else` *syntaktisch* regelt. In verschiedenen Programmiersprachen anzutreffende Konstrukte sind `begin/end-` oder `if-fi-`-Klammerungen, geschweifte Klammern, `endif`, wie in Algol 60 und Ada, usw. In Python geschieht das durch veränderte Einrückungen des entsprechenden Codes. In Racket muss der `else`-Zweig stets vorhanden sein. Für einseitige Entscheidungen gibt es die Sprachelemente `when` und `unless`.

In JavaScript implementiert man die beiden Fälle unter Verwendung geschweifter Klammern folgendermaßen:

```
function teilbar1 (x) {
  if ((x % 3) == 0) {
    if ((x % 4) == 0) {
      return (x + " ist durch 3 und 4 teilbar.")
    } else {
      return (x + " ist nicht durch 4 teilbar.")
    } } }
```

`teilbar1` setzt die o.g. Forderung „`else` belongs to the nearest `if`“ um. Aufrufbeispiele sind

```
> teilbar1(7)
> teilbar1(6)
"6 ist nicht durch 4 teilbar."
> teilbar1(8)
> teilbar1(12)
"12 ist durch 3 und 4 teilbar."
```

hingegen gehört bei `teilbar2` das `else` zum äußeren (entfernten) `if`.

```
function teilbar2 (x) {
  if ((x % 3) == 0) {
    if ((x % 4) == 0) {
      return (x + " ist durch 3 und 4 teilbar.")
    }
  } else {
    return (x + " ist nicht durch 3 teilbar.")
  }
}
```

Aufrufbeispiele sind

```
> teilbar2(7)
"7 ist nicht durch 3 teilbar."
> teilbar2(6)
> teilbar2(8)
8 ist nicht durch 3 teilbar."
> teilbar2(12)
"12 ist durch 3 und 4 teilbar."
```

Durch den Einsatz von geschweiften Klammern für die Blockbildung entstehen unterschiedliche Resultate.

Computerübung 7.6

Folgen Sie dem angegebenen Dialog mit JavaScript.



Computerübung 7.7

Machen Sie sich damit vertraut, wie dem dangling-else-Problem in Python begegnet wird.



Da es algorithmisch nicht allgemein entscheidbar ist, ob es für eine (nicht mehrdeutige) kfG eine äquivalente $LL(1)$ -Grammatik gibt, versuchen wir das in diesem Fall Unmögliche durch eine weitere Transformation zu erreichen. Dabei lernen wir noch eine wichtige Transformationstechnik für kfG kennen.

Wir verwenden die erste der weiter oben verwendeten Grammatiken:

$G = (\{S\}, \{\text{if, b, then, else, other}\}, P, S)$ mit

$$P = \{S \rightarrow \text{if } b \text{ then } S \mid \text{if } b \text{ then } S \text{ else } S \mid \text{other}\}.$$

Wie wir bereits wissen, ist diese Grammatik mehrdeutig.

In den beiden ersten Produktionen für S ist der Verstoß gegen die erste $LL(1)$ -Forderung offensichtlich. Dagegen hilft *Linksfaktorisierung*.

Links-
faktorisierung

Hierbei wird der längste gemeinsame Anfangsteil von Satzformen der rechten Regelseiten für das betrachtete Nichtterminal wie ein gemeinsamer Faktor „ausgeklammert“. Die Restsatzformen werden zur Bildung neuer Regeln unter Verwendung eines neuen Nichtterminals benutzt: Aus

$$X \rightarrow \alpha\beta_1 \mid \alpha\beta_2$$

werden die Regeln

$$X \rightarrow \alpha X' \text{ und } X' \rightarrow \beta_1 \mid \beta_2$$

gebildet, wobei $\alpha \in (N \cup T)^+$, $\beta_i \in (N \cup T)^*$. $\beta_i = \varepsilon$ ist dabei durchaus zulässig.

Wenn wir dies anwenden, ergibt sich (nach Umbenennung eines Nichtterminals) die Grammatik $G = (\{S\}, \{\text{if, b, then, else, other}\}, P, S)$ mit $P =$

$$\begin{aligned} \{S &\rightarrow \text{if } b \text{ then } S X \mid \text{other} \\ X &\rightarrow \text{else } S \mid \epsilon\} \end{aligned}$$

Der Einsatz von Linksfaktorisierung bringt uns in diesem Beispiel leider auch nicht weiter. Wir erhalten eine mehrdeutige Grammatik.



8 LR(k)-Sprachen

8.1 Begriff

Die im Namen $LR(k)$ verwendeten Abkürzungen haben folgende Bedeutungen:

$L \dots$ Das erste (ganz links stehende) L steht für „Analyse des Eingabewortes von links nach rechts“. Wie bei DKAs definiert, arbeitet sich der Lesekopf konsequent von links nach rechts voran. Es gibt nicht etwa einen Rückgriff auf frühere Zeichen, etwa durch Zurückfahren (Linksbewegung) des Kopfes auf dem Eingabeband.

$R \dots$ Das an zweiter Position stehende R steht für „Rechtsableitung des Analysewortes“. Wäre es ein Top-Down-Verfahren¹, das beim Spitzensymbol beginnt, dann würde das am weitesten rechts stehende Nichtterminal zuerst ersetzt. Das Analysewort würde von rechts her entstehen. Da es sich aber um eine *Bottom-up-Analyse* handelt, muss man das Ganze „auf den Kopf stellen“, wie in Abschnitt 8.2 beschrieben.

$k \dots$ Die in Klammern stehende Zahl k gibt die Anzahl der Vorausschauzeichen (genauer: Vorausschautoken) auf das noch nicht analysierte Restwort (die Resttokenfolge) an, sodass eine irrtumsfreie (Sackgassen ausschließende) Entscheidung des jeweils nächsten Analyseschrittes gewährleistet ist. $k = 1$ ist der Vorgabewert, d.h. LR steht für $LR(1)$.

8.2 Deterministische Bottom-up-Syntaxanalyse

$LR(k)$ -Sprachen stellen die umfassendste Klasse deterministisch analysierbarer kontextfreier Sprachen dar. Aus der Theorie formaler Sprachen ist bekannt, dass genau diese Klasse durch deterministische Kellerautomaten beschrieben wird.

Da es sich um ein Bottom-up-Verfahren handelt, wird die jeweils betrachtete Satzform, die einer rechten Regelseite entspricht, durch das zugehörige Nichtterminal auf der linken Seite dieser Regel ersetzt.

Bottom-up-Verfahren

Beispiel 8.1

Wir wollen versuchen, das Wort $a+a*(a+a)$ für die Grammatik $(N, \{+, *, a, (,)\}, P, E)$ mit



¹Die Namensgebung folgt der Top-Down-Analyse-Vorstellung. Eigentlich handelt es nämlich um eine *Linksreduktion*.

$P = \{E \rightarrow E + T, E \rightarrow T, T \rightarrow T * F, T \rightarrow F, F \rightarrow (E), F \rightarrow a\}$ zum Spitzensymbol E zu reduzieren, d.h. $a+a*(a+a) \Leftarrow \dots \Leftarrow E$.

Computerübung 8.1



Verwenden Sie FLACI (kfG), um für die kfG aus Beispiel 8.1 einen NKA zu erzeugen. Bei der Simulation werden Sie feststellen, dass FLACI wegen sehr schlechter Effizienz abbricht. Dies gilt sogar für Wörter geringer Länge.

Übung 8.1



Analysieren Sie das Wort $a+a*(a+a)$ mit Papier und Bleistift „bottom up“ (von links her zum Spitzensymbol reduzierend) unter Verwendung genau eines Vorausschauzeichens. Welche Erfahrung machen Sie dabei? Was stellen Sie fest?

Übung 8.2



Analysieren Sie das im Beispiel gegebene Wort „top down“ unter Verwendung genau eines Vorausschauzeichens. Notieren Sie die Ableitung. Vergleichen Sie die beiden Methoden. Was stellen Sie fest?

Mit FLACI (kfG) erhält man die folgende Rechtsableitung für das Wort $a+a*(a+a)$ und die Grammatik aus Beispiel 8.1: $E \Rightarrow E + T \Rightarrow E + T * F \Rightarrow E + T * (E) \Rightarrow E + T * (E + T) \Rightarrow E + T * (E + F) \Rightarrow E + T * (E + a) \Rightarrow E + T * (T + a) \Rightarrow E + T * (F + a) \Rightarrow E + T * (a + a) \Rightarrow E + F * (a + a) \Rightarrow E + a * (a + a) \Rightarrow T + a * (a + a) \Rightarrow F + a * (a + a) \Rightarrow a + a * (a + a)$

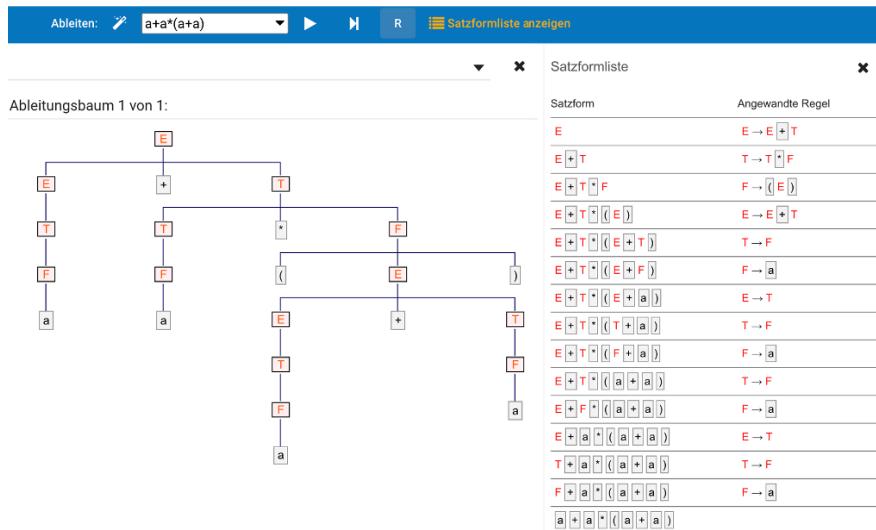


Abbildung 8.1: Rechtsableitung (R – statt L – auswählen!) für $a+a*(a+a)$

Wir kehren nun die Reihenfolge der Rechtsableitung um und notieren die einzelnen Satzformen als Konfigurationenfolge eines DKA, der soetwas wie eine Links-

reduktion (vom Wort zum Spitzensymbol hin) simuliert. Um das Ende des Eingabewortes zu kennzeichnen, verwenden wir ein Dollarzeichen \$. Als Aktion vermerken wir *shift*, *reduce* bzw. *accepted* mit folgenden Bedeutungen:

shift bedeutet, dass das nächste Token aus dem Eingabepuffer (Restwort) entfernt und auf den Stapel gelegt wird. shift

reduce: $X \rightarrow \beta$ bedeutet, dass der Stapelinhalt gemäß Regel $X \rightarrow \beta$ reduziert wird. Die rechte Regelseite β stimmt genau mit dem obersten Stapel(teil)wort überein. Genau dieser Stapelinhalt wird durch die linke Regelseite, also X , ersetzt. reduce

accepted steht ganz am Ende, wenn das Startsymbol der Grammatik als einziges Zeichen im Keller steht und der Puffer für das Eingabewort leer ist. accepted

| Stapel ↓ | Restwort | Aktion |
|------------|-------------|-------------------------------|
| \$ | a+a*(a+a)\$ | shift |
| \$a | +a*(a+a)\$ | reduce: $F \rightarrow a$ |
| \$F | +a*(a+a)\$ | reduce: $T \rightarrow F$ |
| \$T | +a*(a+a)\$ | reduce: $E \rightarrow T$ |
| \$E | +a*(a+a)\$ | shift |
| \$E + | a*(a+a)\$ | shift |
| \$E+a | *(a+a)\$ | reduce: $F \rightarrow a$ |
| \$E+F | *(a+a)\$ | reduce: $T \rightarrow F$ |
| \$E+T | *(a+a)\$ | shift !!! |
| \$E+T* | (a+a)\$ | shift |
| \$E+T*(| a+a)\$ | shift |
| \$E+T*(a | +a)\$ | reduce: $F \rightarrow a$ |
| \$E+T*(F | +a)\$ | reduce: $T \rightarrow F$ |
| \$E+T*(T | +a)\$ | reduce: $E \rightarrow T$ |
| \$E+T*(E | +a)\$ | shift |
| \$E+T*(E+ | a)\$ | shift |
| \$E+T*(E+a |)\$ | reduce: $F \rightarrow a$ |
| \$E+T*(E+F |)\$ | reduce: $T \rightarrow F$ |
| \$E+T*(E+T |)\$ | reduce: $E \rightarrow E + T$ |
| \$E+T*(E |)\$ | shift |
| \$E+T*(E) | \$ | reduce: $F \rightarrow (E)$ |
| \$E+T*F | \$ | reduce: $T \rightarrow T * F$ |
| \$E+T | \$ | reduce: $E \rightarrow E + T$ |
| \$E | \$ | accepted |

Tabelle 8.1: Erfolgreiche LR-Analyse des Wortes a+a*(a+a)

shift oder reduce? Die Angabe der Aktionen in Tabelle 8.1 ist nicht in jedem Schritt eindeutig. Beispielsweise wäre in der durch drei Ausrufezeichen gekennzeichneten Zeile 9 anstelle der gewählten shift-Aktion ein reduce mit Regel $E \rightarrow T$ prinzipiell möglich. Hätten wir uns jedoch für diese Reduktion entschieden, wäre die Analyse in eine Sackgasse geraten. Wie man der Konfigurationenfolge in Tabelle 8.1 entnimmt, ist $F \rightarrow a$ die (von unten gezählte) 6. angewandte Regel. Damit dies möglich ist, müssen vorher drei shift-Aktionen stattfinden. Wir halten also fest, dass wir ohne Zuhilfenahme der vorbereiteten Rechtsableitung nicht in der Lage gewesen wären, im betrachteten Beispiel eine sackgassenfreie Bottom-up-Syntaxanalyse durchzuführen.

Handle

Für eine *deterministische* Bottom-up-Analyse ist es notwendig, die als nächstes stattfindende Aktion (reduce oder shift) irrtumsfrei zu bestimmen: Entweder das nächste Zeichen des restlichen Eingabewortes wird auf den Stapel gelegt (shift) oder die Satzform β am oberen Stapelrand – das sog. *Handle* (Henkel oder An-satz)² wird reduziert (reduce).

Bezeichnet β eine Satzform (als Teilzeichenkette der aktuell betrachteten Satzform) und γ ein entsprechendes Restwort, so gilt: Eine Reduktion $\alpha\beta\gamma \Leftarrow \alpha X\gamma$, mittels Regel $X \rightarrow \beta$, findet statt, wenn $\alpha X\gamma \stackrel{*}{\Leftarrow} s$ gilt. Dies entspricht der Empfehlung „Reduktion genau dann, wenn die grundsätzlich erfolgreiche Analyse dadurch nicht in eine Sackgasse führt.“ Aus praktischer Sicht bringt das zunächst gar nichts. Aber wir können damit den Begriff der $LR(k)$ -Sprachen definieren.

Definition 8.1

Eine kFG ist **$LR(k)$ -Grammatik**, wenn für jede Satzform $\alpha\beta\gamma$ in einer Rechtsableitung, mit $\gamma \in T^*$, $\beta \in (N \cup T)^+$ und $\alpha \in (N \cup T)^*$, das Handle β durch Vorausschau auf die ersten k Zeichen von γ eindeutig identifiziert werden kann. Es gilt $k \geq 0$.



Welche Informationen stehen bei $LR(k)$ -Sprachen überhaupt zur Verfügung, um ein Handle zu bestimmen?

- Die Vorausschau auf die ersten k Zeichen des noch unverbrauchten Eingabewortes und
- der gesamte Stapelinhalt.

Vorausschauzeichen

In Bezug auf die Anzahl k der Vorausschauzeichen reicht es aus, wenn wir uns auf $k = 0$ und $k = 1$ konzentrieren. Dies sichert der folgende Satz.

Satz 8.1

Zu jeder $LR(k)$ -Grammatik, mit $k > 1$, gibt es eine äquivalente $LR(1)$ -Grammatik.



Beweis

s. Literatur □

Stapelinhalt

Würde in jedem Schritt der (gesamte) Stapelinhalt ausgewertet (gelesen und ver-

²In mancher Literatur wird auch die gesamte zugehörige Regel, $X \rightarrow \beta$, als Handle bezeichnet.

arbeitet), hätte dies ein unakzeptables Absinken der Arbeitsgeschwindigkeit eines LR -Parsers zur Folge. Deshalb besteht der Trick darin, bestimmte Konfigurationen durch Zustände (eines potentiellen DEA³) auszudrücken und im Wechsel mit dem Vokabular (Terminale, Nichtterminale) auf den Stapel zu legen. Diese Zustände werden konzeptionell gern mit $0, 1, 2, \dots, n$ angegeben. Die im Vorab ermittelte Steuerung (Aktion und Folgezustandsbestimmung) wird in Tabellenform angegeben.

8.3 Tabellengesteuerte $LR(k)$ -Syntaxanalyse

Zur Illustration verwenden wir im Folgenden wiederum die kfG aus Beispiel 8.1 und zitieren hier zur besseren Lesbarkeit die Regeln dieser Grammatik.

- (1) $E \rightarrow E + T,$
- (2) $E \rightarrow T,$
- (3) $T \rightarrow T * F,$
- (4) $T \rightarrow F,$
- (5) $F \rightarrow (E),$
- (6) $F \rightarrow a$

| Z | action | | | | | goto | | | | | | | | | |
|----|--------|-------|-------|---|-------|--------|---|---|---|---|----|----|----|---|---|
| | a | + | * | (|) | \$ | a | + | * | (|) | \$ | E | T | F |
| 0 | s | | | s | | | 5 | | 4 | | | | 1 | 2 | 3 |
| 1 | | s | | | | accept | | 6 | | | | | | | |
| 2 | | r_2 | s | | r_2 | r_2 | | | | 7 | | | | | |
| 3 | | r_4 | r_4 | | r_4 | r_4 | | | | | | | | | |
| 4 | s | | | s | | | 5 | | 4 | | | 8 | 2 | 3 | |
| 5 | | r_6 | r_6 | | r_6 | r_6 | | | | | | | | | |
| 6 | s | | | s | | | 5 | | 4 | | | 9 | 3 | | |
| 7 | s | | | s | | | 5 | | 4 | | | | 10 | | |
| 8 | | s | | | s | | | 6 | | | 11 | | | | |
| 9 | | r_1 | s | | r_1 | r_1 | | | | 7 | | | | | |
| 10 | | r_3 | r_3 | | r_3 | r_3 | | | | | | | | | |
| 11 | | r_5 | r_5 | | r_5 | r_5 | | | | | | | | | |

Tabelle 8.2: LR -Parsetabelle für die Grammatik aus Beispiel 8.1

An dieser Stelle wollen wir die *LR-Parsetabelle* als gegeben ansehen und uns nur *LR-Parsetabelle*

³Man kann sich das als Handle-basierten DEA vorstellen.

mit deren Verwendung im Analyseprozess vertraut machen.

Eine *LR*-Parsetabelle, wie etwa in Tabelle 8.2, besteht aus zwei Teilen: einem *action*-Teil und einem *goto*-Teil. Die Tabelleneinträge haben folgende Bedeutungen:

- Das Zeichen $\$$ im Kopf der Tabelle ist das Eingabeabschlusszeichen⁴.
- s steht für *shift*: Schiebe das aktuelle Eingabezeichen auf den Stapel.
- r_i steht für *reduce* mit Regel i .
- *accept* steht für „Eingabewort akzeptiert“.
- Leere Felder bedeuten *Fehler*.

shift-reduce-Syntaxanalyse

Zur Dokumentation der Analyse verwenden wir einen Stapel, auf dem sowohl Zustandsnamen (hier: $0, 1, \dots, n$) als auch Vokabularzeichen, d.h. Zeichen aus $N \cup T$, liegen.

Für die Bearbeitung des Wortes $(a+a)$ ergeben sich die in Abbildung 8.2 gezeigten Kellerinhalte (von links nach rechts).

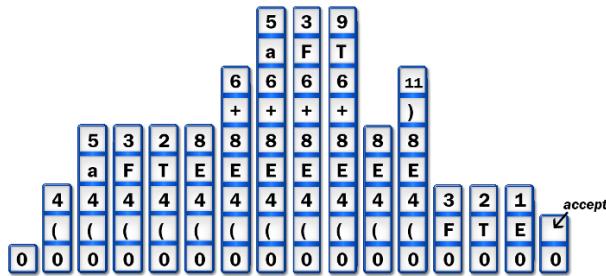


Abbildung 8.2: Shift-reduce-Parsing für $(a+a)$ mit Steuerung nach Tabelle 8.2

Zu Beginn bildet der Zustandsname 0 das top of stack. Als Erstes wird die *action*-Funktion bemüht, um aus dem aktuellen Zustand und dem nächsten Eingabezeichen eine der o.g. drei Aktionen (*shift*, *reduce*, *accept*) durchzuführen. Im Falle von *shift* wird das jeweils nächste Zeichen des Eingabewortes auf den Stapel geschoben. Die bei r_i auszuführende Reduktion mit Regel i bewirkt, dass der rechte Regelseite von r_i entsprechende Stapelinthalte, inkl. Zustandsnamen, durch das Nichtterminal auf der linken Seite von r_i ersetzt wird.

Anschließend kommt die *goto*-Funktion zum Einsatz. Sie nimmt (ebenfalls) den aktuellen Zustand und das oberste Vokabularzeichen, um den neuen Zustand zu

⁴Das Eingabeabschlusszeichen markiert das Ende des Eingabewortes und soll den Abschluss des Eingabevorgangs erkennbar machen. Selbstverständlich könnte dies auch anders, etwa durch das Ende einer Liste, bewirkt werden. Deshalb wird es in mancher Literatur auch nicht explizit verwendet. Wir benutzen das Zeichen $\$$ unter Hinweis auf die Forderung, dass es mit keinem Terminalsymbol übereinstimmt.

bestimmen. Der Name dieses Zustandes wird gekellert und bildet das neue top of stack. Damit ist ein Arbeitstakt beendet.

Die Analyse stoppt erfolgreich, wenn der Zustand *accept* erreicht ist, bzw. mit einem Fehler, wenn sie auf ein leeres Feld in der Tabelle trifft. Das Wort (a+a) wird akzeptiert.

Übung 8.3

Arbeiten Sie den Analyseprozess auf Papier gründlich durch.



Didaktischer Hinweis 8.1

Die klassische Form einer LR(1)-Parse-tabelle weicht von der in Tabelle 8.2 ein wenig ab: Anstelle der $s_i(\text{shift})$ -Einträge im action-Teil verwendet man s_i -Angaben, s für shift und i für den nach shift eingenommenen Zustand. Auf diese Weise spart man die Spalten für Terminals im goto-Teil ein.

In diesem Einführungsbeispiel machen wir davon jedoch keinen Gebrauch, um die Bestimmung der jeweiligen Aktion von der Berechnung des jeweiligen goto-Zustands konzeptionell zu entkoppeln.

In Abschnitt 8.4 kommen wir mit dieser komakteren Notation in Berührung, wenn wir die von FLACI generierte Parse-tabelle einsetzen.

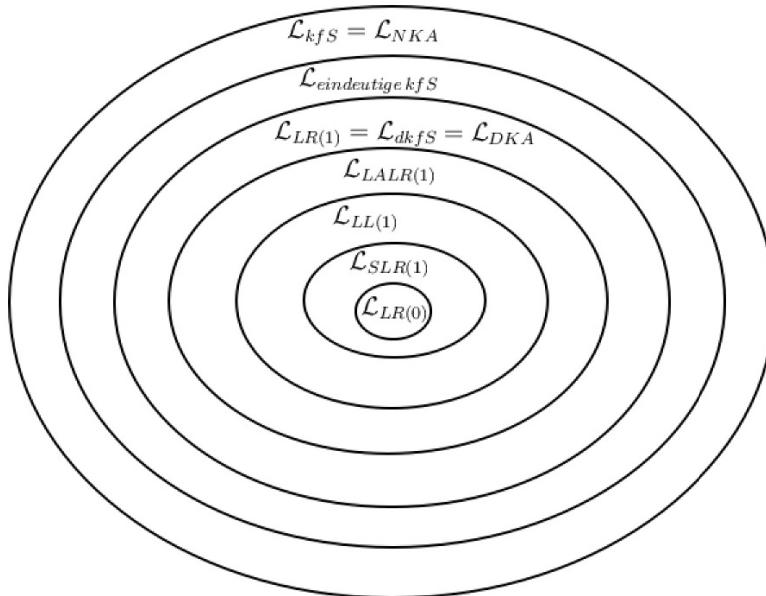


Abbildung 8.3: $LR(1)$ -Sprachen mit relevanten Teilmengen

Abbildung 8.3 enthält neben den bekannten auch Sprachklassen, die bisher noch nicht behandelt bzw. erwähnt wurden. In den folgenden Abschnitten werden wir

dies nachholen und dabei immer wieder Bezug auf diese Abbildung nehmen. Sie hat also strukturierende Wirkung und hilft uns den Überblick zu behalten.

Wie auch in Abbildung 8.3 dargestellt, können *LR(1)*-Sprachen niemals mehrdeutig sein.



Didaktischer Hinweis 8.2

In der Praxis gibt es durchaus einige Gründe, von mehrdeutigen Grammatiken auszugehen und gewisse Maßnahmen zur Schadensbegrenzung hinzuzufügen. Wir sehen Mehrdeutigkeit hier eher als schädlich für die Sprachverarbeitung und trachten nach deren Vermeidung.

shift/reduce-

Konflikt

reduce/reduce-

Konflikt

Wie Abbildung 8.3 zeigt, ist nicht jede eindeutige kfG *LR(1)*-Grammatik. Bei Nicht-*LR(1)*-Grammatiken treten *shift/reduce*- und *reduce/reduce*-Konflikte auf: Für den betrachteten Schritt ist sowohl eine *shift*- als auch eine *reduce*-Operation möglich oder für ein *reduce* kommen mindestens zwei Regeln infrage.

Eine typische „Reparaturmaßnahme“ für solche kfG besteht in der Praxis darin, die Assoziativität von Operationen zu beschreiben. Manche Parsergeneratoren, wie etwa der Yacc (s. Abschnitt 8.4), bieten sprachliche Möglichkeiten, um dies auszudrücken. Wie der FLACI-Parsergenerator mit Nicht-*LR*-Grammatiken umgeht, wird ebenfalls in Abschnitt 8.4 erwähnt.

Konstruktion von
LR-Parsetabellen

Nachdem nun klar ist, wie *LR*-Parsetabellen eingesetzt werden, betrachten wir deren Konstruktion. Dabei gehen wir nicht ins Detail. Grundsätzlich gibt es drei Haupt-Konstruktionsverfahren. Sie führen zu sog. *LR(1)*-, *SLR(1)*- und *LALR(1)*-Sprachen. *SLR(1)*-Sprachen sind die einfachsten. Die *SLR*-Methode hat die geringste Reichweite (Anzahl der Grammatiken, für die sie arbeiten), ist aber am einfachsten zu implementieren. Der Buchstabe *S* im Namen steht für „simple“.

Eine allgemeine *LR(k)*-Parsetabelle, mit $k = 0$ bzw. $k = 1$, entsteht in zwei Schritten:

1. Erzeugung aller Zustände, die den aktuellen Stand der Analyse – die Stapelbelegung – repräsentieren
2. Konstruktion des zugehörigen Überführungsgraphen (DEA) – Repräsentation als Tabelle

Je nach Anzahl der Vorausschauzeichen, die für den ersten bzw. zweiten Arbeitsschritt herangezogen werden, unterscheidet man folgende Sprachklassen:

| Sprachklasse | Anzahl der Vorausschauzeichen für die ... | |
|---------------|---|-----------------------|
| | Erzeugung der Zustände | Erzeugung der Tabelle |
| <i>LR(0)</i> | 0 | 0 |
| <i>SLR(1)</i> | 0 | 1 |
| <i>LR(1)</i> | 1 | 1 |

SLR(1)-Sprachen

Wie in Abbildung 8.3 dargestellt, liegen die *SLR(1)*-Sprachen gerade zwischen

den in obiger Tabelle genannten Sprachklassen. $SLR(1)$ -Grammatiken beschreiben *fast* alle Konstrukte in Programmiersprachen und beanspruchen im Vergleich zu $LR(1)$ -Sprachen wesentlich kleinere Parsetabellen.

Für reine $LR(1)$ -Sprachen muss man mit Tabellen rechnen, deren Konstruktion im Allgemeinen per Hand unzumutbar ist. Für Sprachen, die über den Status von Übungsbeispielen hinausgehen, kann ein zugehöriger DEA durchaus mehrere Hundert Zustände besitzen. Handarbeit ist da nicht möglich. Wir brauchen ein Werkzeug, das die DEA-Generierung leistet. Da dieser Erzeugungsprozess algorithmisch⁵ beschreibbar ist, gibt es solche Werkzeuge tatsächlich. In Abschnitt 8.4 befassen wir uns einführend mit der automatisierten LR -Parsergenerierung.

Die Suche nach einer Möglichkeit Parsetabellen zu verkleinern und dennoch *nahezu* über den Beschreibungsumfang von $LR(1)$ -Grammatiken zu verfügen, führte zu $LALR(1)$ -Grammatiken. Nach der Erzeugung der $LR(1)$ -Zustandsmenge werden zusammenfassbare Zustände verschmolzen. Dadurch entstehen auch kleinere Tabellen. Die Bezeichnung $LALR(1)$ steht für LookAhead- $LR(1)$. Für die meisten Praxisanforderungen stellen sie den besten Kompromiss dar und sind auch Gegenstand von Parsergeneratoren.

Aus dem Gebiet des Compilerbaus übernehmen wir für (schnelles) tabellengesteuertes Parsing in Abhängigkeit von der Vorausschauzeichenanzahl k folgende Befunde mit praktischer Relevanz:

$LR(0)$ -Sprachen sind einfach zu analysieren aber nicht mächtig genug, um alle gängigen Konstrukte in (imperativen) Programmiersprachen zu beschreiben.

$LR(1)$ -parsing, kurz: LR -Parsing, ist die leistungsfähigste und in der Praxis am häufigsten benutzte Analysetechnik.

$LR(k)$ -parsing, $k > 1$, ist wegen Satz 8.1 (auf Seite 202) praktisch ohne Interesse.

Interessant ist vielmehr der Zusammenhang zwischen $LR(k)$ - und $LL(k)$ -Sprachen:

$LALR(1)$ -Sprachen

Lookahead- $LR(1)$

Satz 8.2

Jede $LL(k)$ -Sprache ist auch $LR(k)$ -Sprache.



Beweis

s. Literatur



Die Umkehrung von Satz 8.2 gilt nicht, s. Abbildungen 8.3 und 7.2 (s. Seite 175). Die Grammatik in Beispiel 8.1 ist eine $LR(1)$ -, jedoch keine $LL(1)$ -Grammatik.

Auch für die $LR(1)$ -Grammatik $G = (\{A, S\}, \{a, b\}, \{S \rightarrow aS \mid A, A \rightarrow aAb \mid \epsilon\}, S)$ mit $L(G) = \{a^i b^j \mid i \geq j\}$ gibt es keine äquivalente $LL(k)$ -Alternative.

⁵Dies ist Gegenstand eines Kurses zum Compilerbau und wird hier nicht thematisiert.

8.4 Automatisierte Parsergenerierung

Für die meisten der praxisrelevanten Sprachen kommen *LALR(1)*-Parser zum Einsatz. Wie bereits erwähnt, werden sie nicht per Hand geschrieben, sondern mit entsprechenden *Parsergeneratoren* automatisch erzeugt.

Yacc

Ein traditioneller Compiler-Generator ist *Yacc* als Kürzel für *yet another compiler compiler*⁶. Scanner, mit denen die mittels Yacc erzeugten Parser kooperieren, werden mit einem *Scannergenerator*, meist mit *Lex*, erzeugt. Lex und Yacc sind ursprünglich in C geschriebene Programme, die seit langem als Bestandteil von Linux-Distributionen zur Verfügung stehen. Die verbesserten Versionen *Flex* und *Bison* sind auch für andere Betriebssysteme aus dem Web herunterladbar.

Scannergenerator

Lex

Flex

Bison

Es gibt eine Reihe von Problemen der angewandten Informatik, die sich auf die Übersetzung einer Sprache in eine andere zurückführen lassen. Insofern ist es lohnenswert, Theoriekenntnisse zu besitzen, die uns zusammen mit einem intuitiv zu bedienenden Werkzeug befähigen, hierfür komplexe Software zu entwickeln.

Didaktischer Hinweis 8.3

Aus der Erfahrung in der Arbeit mit Studierenden wissen wir, dass es bei der Verwendung von Yacc immer wieder technische Probleme gibt, die fern der eigentlichen Aufgabenstellung zeitaufwendige Lösungen erfordern. Lex und Yacc sind also klassische Werkzeuge für Informatiker*innen, die viel Erfahrung im Umgang mit diesen konkreten Programmen mitbringen bzw. aufbauen wollen.

Jison

In FLACI ist der Compilergenerator *Jison* eingebaut. Der Name erinnert an *Bison* mit einem vorn stehenden J für Javascript, der Implementierungssprache von *Jison*. Damit ist die Ausführbarkeit von *Jison* in der Javascript-Umgebung von FLACI gesichert. *Jison* erwartet die Beschreibung eines Übersetzungsprozesses einer Sprache L_1 in eine Sprache L_2 und erzeugt daraus einen $L_1 \rightarrow L_2$ -Compiler in Javascript.

Wie man der Dokumentation von *Jison* entnimmt, ähneln die textuellen Beschreibungen sehr denen, die man in *Bison* verwendet. Um den o.g. Problemen entgegen zu wirken, ermöglicht FLACI einen visuellen Entwurf der Compilerdefinition unter Verwendung der Sprache *TDL* (Translation description language).

TDL

Sie ermöglicht grafische Repräsentationen mit textuellen und programmiersprachlichen Ergänzungen. *TDL* integriert vorhandene Werkzeuge, wie etwa Syntaxdiagramme, die schon für die Definition von kfG Verwendung fanden.

TDL bietet *LL(1)*- und *LALR(1)*-Parsing, ist FLACI-Bestandteil und dadurch in der Lage, mit den anderen FLACI-Komponenten zusammenzuarbeiten. Für mutmaßliche *LR(1)*-Sprachen behandelt *TDL* shift-reduce-Konflikte durch Bevorzugung von shift und reduce-reduce-Konflikte durch Verwendung der in der Liste zuerst aufgeschriebenen Produktion. Ein Eingriff in die *Jison*-Verarbeitung durch die

Konflikt-behandlung

⁶Für Compilergeneratoren wird auch der Begriff Compiler Compiler verwendet.

Veränderung der Reihenfolge grammatischer Regeln ist also dadurch möglich und ggf. nötig. Noch besser ist es, mehrdeutige Grammatiken zu vermeiden.

In Abschnitt 7.5 ab Seite 186 haben wir das Thema „Parsergenerator“ bereits angesprochen und in FLACI (TDL und Jison) benutzt, um einen $LL(1)$ -Parser für PL/0 in Javascript herzustellen. Hierzu wurde eine reduzierte Grammatik der zu analysierenden Sprache (PL/0) in BNF übergeben und ein entsprechender $LL(1)$ -Parser in JavaScript erzeugt. Die $LALR(1)$ -Parsergenerierung geschieht analog, wobei eben anstelle des $LL(1)$ -Typs $LALR(1)$ gewählt wird.

Wir wollen nun die Arbeitsweise des Parsers für die Grammatik aus Beispiel 8.1 auf Seite 199 und das Eingabewort $a+a*(a+a)$ mit FLACI nachvollziehen.

Zu diesem Zweck ergänzen wir eine Regel für das (neue) Spitzensymbol *Start*:

$$Start \rightarrow E \boxed{EOF}.$$

Mit \boxed{EOF} (end of file) wird ein Eingabe-Abschlusszeichen, das selbst kein Terminal ist, hinzugefügt. Im Parseprozess wird es de facto wie ein Terminal behandelt, was den Vorgang der Erkennung für das Eingabewort des Wortendes vereinfacht.

Die Parserbeschreibung für Beispiel 8.1 umfasst sechs Tokendefinitionen, die man auf der linken Seite in Abbildung 8.4 gut erkennen kann.

Computerübung 8.2

Führen Sie einen Scanner- und einen Parsertest für das Eingabewort $a+a*(a+a)$ durch und überprüfen Sie die Ausgabe.

$LALR(1)$ -Parser
für PL/0

TDL und Jison



Abbildung 8.5 zeigt die von Jison generierte Parsetabelle für die Grammatik in Beispiel 8.1. Das farbig hinterlegte a steht für accept.

Übung 8.4

Protokollieren Sie den Analyseprozess nach dem Vorbild von Abbildung 8.2.



Computerübung 8.3

Provozieren Sie shift/reduce- und reduce/reduce-Konflikte, indem Sie in der Grammatik die Regel $F \rightarrow E$ ergänzen. Wenn Sie den Parser danach wiederum für $a+a*(a+a)$ testen, finden Sie ein Dreiecksymbol, hinter dem sich eine Liste solcher Konflikte verbirgt.



Computerübung 8.4

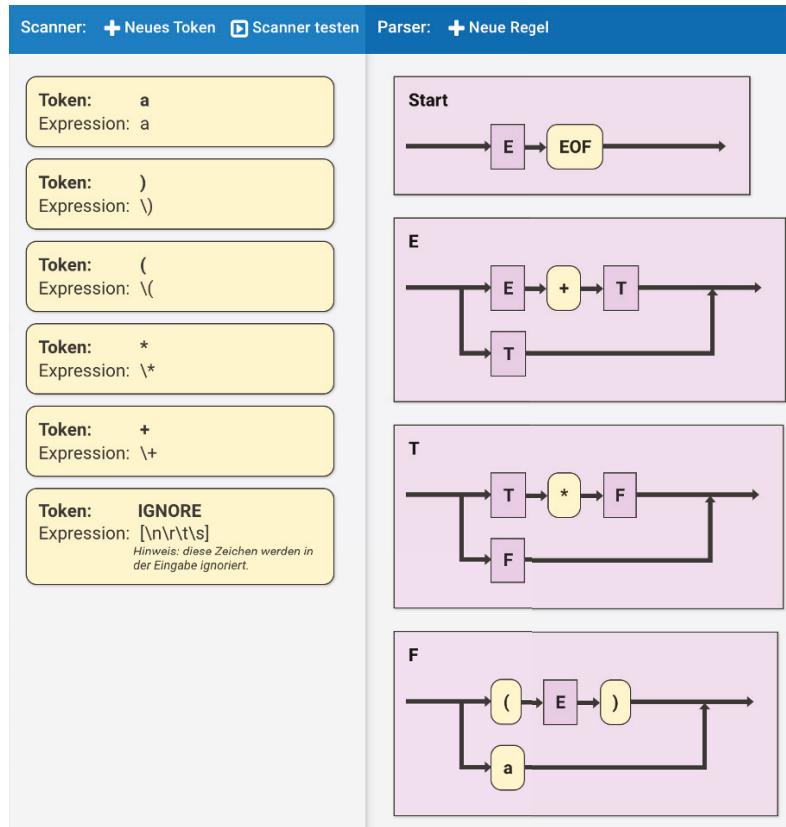
Wir betrachten eine Grammatik für die Sprache einfacher arithmetischer Ausdrücke: $G = (N, T, P, s), N = \{E, T, F\}, T = \{+, -, *, /, num\}, s = E. P = \{E \rightarrow E + T \mid E - T \mid T, T \rightarrow T * F \mid T / F \mid F, F \rightarrow num\}$ num steht für eine beliebige ganze Zahl. Verwenden Sie Jison zur automatisierten Generierung eines $LALR(1)$ -Parsers für diese $L(G)$. Verwenden Sie das Eingabewort $2+4*5$ als Analysebeispiel, ebenso wie $2++4*5 \notin L(G)$.



Computerübung 8.5

Verwenden Sie TDL/Jison zur Erzeugung eines $LALR(1)$ -Parsers für PL/0 aus Abschnitt 7.5 ab Seite 186 und testen Sie diesen Parser für syntaktisch korrekte und fehlerhafte PL/0-Programme.



Abbildung 8.4: Parserdefinition für *LALR*(1)-Parsegenerierung mit TDL

| states | \$end | (|) | * | + | EOF | IGNORE | a | E | F | Start | T |
|--------|-------|----|-----|----|----|-----|--------|----|----|----|-------|----|
| 0 | | | s5 | | | | | s6 | 2 | 4 | 1 | 3 |
| 1 | a | | | | | | | | | | | |
| 2 | | | | | s8 | s7 | | | | | | |
| 3 | | | r3 | s9 | r3 | r3 | | | | | | |
| 4 | | | r5 | r5 | r5 | r5 | | | | | | |
| 5 | | s5 | | | | | | s6 | 10 | 4 | | 3 |
| 6 | | | r7 | r7 | r7 | r7 | | | | | | |
| 7 | r1 | | | | | | | | | | | |
| 8 | | s5 | | | | | | s6 | | 4 | | 11 |
| 9 | | s5 | | | | | | s6 | | 12 | | |
| 10 | | | s13 | s8 | | | | | | | | |
| 11 | | | r2 | s9 | r2 | r2 | | | | | | |
| 12 | | | r4 | r4 | r4 | r4 | | | | | | |
| 13 | | | r6 | r6 | r6 | r6 | | | | | | |

Abbildung 8.5: Parse-tabelle für Grammatik aus Beispiel 8.1



9 Sprachübersetzerprojekte

9.1 Syntaxgesteuerte Übersetzung

In Abschnitt 5.2, s. Seite 128, wurde die „klassische Architektur“ eines Compilers beschrieben. Sie besteht aus einem *Analyseteil* (Parsing eines gegebenen Wortes im Quellcode) und einem *Syntheseteil* (Zielcodegenerierung).

Das klassische Vorgehen besteht gemäß Abbildung 5.12 auf Seite 129 darin, für ein syntaktisch korrektes Wort einen abstrakten Syntaxbaum (AST) zu erzeugen. Dieser dient als „Skelett“ für die Herstellung des Wortes in der Zielsprache.

Didaktischer Hinweis 9.1

In der Theorie formaler Sprachen betrachten wir die Fragestellung $w \stackrel{?}{\in} L(G)$. Zu den Ergebnissen gehören effiziente Parsingverfahren für praxisrelevante Sprachklassen, vor allem *LL(1)* und *LR(1)*. Vollständige Übersetzungsprozesse erfordern einen Syntheseteil, um ein Wort aus einer in eine andere zu übersetzen. Dies betrifft zahlreiche Anwendungsbereiche, wie man sie auch in diesem Kapitel findet.

Analyseteil
Syntheseteil

Aus dem Gebiet des Compilerbaus sind diverse Synthesemethoden bekannt. Insbesondere für *LR(1)*-Sprachen bietet sich die *syntaxgesteuerte Übersetzung* (syntax directed translation = SDT) an.

syntaxisgesteuerte
Übersetzung

Die Idee besteht darin, den Syntaxanalyseprozess mit der Zielcodegenerierung zu kombinieren. Dies geschieht, indem für jede syntaktische Regel der verwendeten (reduzierten) Grammatik eine *semantische Regel* hinterlegt wird.

semantische
Regel

Der Parsebaum wird um Attribute erweitert, die für jeden mit einem Vokabularzeichen markierten Knoten des Baums eingerichtet werden. Man nennt dies einen *attribuierten Parsebaum*.

attribuierten
Parsebaum

Die Attribute selbst werden durch Anwendung der semantischen Regeln berechnet. Diese Regeln können so gestaltet werden, dass sie Seiteneffekte oder komplexere Reaktionen bewirken, so dass man auch von *semantischen Aktionen* spricht.

abstrakte
Spezifikation
TDL

Eine kfG zusammen mit semantischen Regeln nennt man *syntaxgesteuerte Definition*. Sie stellt eine *abstrakte Spezifikation* für die Übersetzung dar.

In FLACI gibt es dafür eine *Übersetzungsbeschreibungssprache – TDL* (translation description language). Mit TDL können auch Variablen-deklarationen und Hilfsvariablen angegeben werden. Handelt es sich dabei um Code, der für alle semantischen Aktionen genutzt werden kann, wird dies als globaler Code hinterlegt.

| | |
|--------------------------------------|---|
| Globaler Code | <i>Globaler Code</i> für alle semantischen Regeln ein und derselben Compilerbeschreibung ist in FLACI editierbar, nachdem der entsprechende Bereich durch Mausklick innerhalb eines (beliebigen) Nichtterminals (nicht innerhalb einer syntaktischen Regel) ausgewählt wird. |
| synthetisierte Attribute S-Attribute | Ein attributierter Parsebaum für <i>LR(1)</i> -Parsing entsteht bekanntlich von den Blättern her und endet mit der Wurzel. Die (meist zusammengesetzten) Attribute beziehen die Werte, die in die Berechnung eingehen, ausschließlich von Nachfolgerknoten. Solche Attribute heißen <i>synthetisierte Attribute</i> , kurz: <i>S-Attribute</i> . Im Gegensatz dazu greifen <i>ererbte Attribute</i> auf Vorgänger- und Geschwisterknoten zu. Mit (verzichtbaren) ererbten Attributen ¹ befassen wir uns hier nicht. Die Attributwerte von Terminalen/Tokens liefert der Scanner in Form der Lexeme, die in der lexikalischen Analyse gewonnen wurden. Auf diese Weise ergibt sich für <i>LR(1)</i> -Sprachen im Idealfall eine höchst effiziente <i>Einpass</i> -Compilation. |
| Aufbau der semantischen Regeln | In Abschnitt 7.5, Seite 186, haben wir einen <i>LL(1)</i> -Parser für PL/0 erzeugt. Dieser arbeitet top down nach der Methode des rekursiven Abstiegs. Auch solche Parser können mit Attributen (L-Attribute) dekoriert werden. Deren Berechnungsreihenfolge kann sich aber als etwas „unhandlich“ erweisen, da u.U. nicht alle Attributwerte, die zur Berechnung anderer Attribute benötigt werden, zur Verfügung stehen. (Abhilfe schaffen Abhängigkeitsgraphen und topologische Sortierung.) Semantische Regeln haben folgenden Aufbau: Die zu deren Definition verwendeten Symbole der Gestalt $\$n$ bezeichnen den Wert des n -ten Vokabularzeichens (Nichtterminale und Terminate) in der betreffenden Regel. Die Zählung beginnt mit dem ersten Vokabularzeichen auf der jeweils <i>rechten</i> Seite der betrachteten Produktion mit 1, d.h. mit $\$1$. Dem Nichtterminal auf der linken Regelseite wird das Zeichen $\$\$$ zugeordnet. Die Ausdrücke $\$n$ und $\$\$$ werden von FLACI als Zeichenketten verarbeitet. Bei Bedarf können aber auch andere JavaScript Objekte an $\$\$$ zugewiesen werden. Das ist dann sinnvoll, wenn eine komplexe Objektstruktur generiert werden soll. Wie wir in Abschnitt 9.4.2 sehen werden, kann man $\$\$$ und die $\$n$ wie Variablennamen in beliebigen Ausdrücken verwenden. Ändert man nichts an den Vorgabe-Einstellungen, die in FLACI grau hervorschimmern, erzeugt der Generator den Parser für die Quellsprache. Es erfolgt eine 1:1-Übersetzung, bei der das syntaktisch korrekte Eingabewort (evtl. ohne die zu ignorierenden Zeichen) als Übersetzungsergebnis entsteht, oder es wird einfach <code>true</code> ausgegeben. Es gibt eine große Klasse syntaxgesteuerter Definitionen, für die Compiler mit |

¹Der Verzicht auf ererbte Attribute bedeutet lediglich einen Verzicht auf Bequemlichkeit, jedoch keine sprachliche Einschränkung: Man kann jedes Programm so umschreiben, dass ererbte Attribute eliminiert werden, sodass man mit S-Attributen auskommt.

sehr wenig zusätzlichem Programmieraufwand von einem Generator konstruiert werden können. Deshalb beschränken wir uns hier auf die Betrachtung der *Bottom-up-Auswertung S-attributierter Definitionen*.

9.2 Audiocompiler und Musikinterpretation

9.2.1 Die Musiksprache ML

Für den folgenden Ausflug in die Welt der Musik brauchen Sie weder Musikproduzent noch leidenschaftlicher Musikkonsument zu sein. Wie schon in Abschnitt 1.2 angedeutet, betrachten wir hier eine sehr einfache textuelle Notation, die wir Musiksprache (*ML* = music language) nennen, und deren Wörter Lieder (Songs) sind:

ML

- Ein Musikstück ist ein Song.
- Eine (auch leere) Folge von Noten ist ein Song.
- Eine Note, wie etwa G0-1, besteht aus dem Ton (C, D, E, F, **G**, A, H), der Oktave (**0** oder 1), gefolgt von einem Strich und der Länge des Tons.
- Für die Tonlänge ist je einer der Werte **1**, 2, 4, 8, 16, 32 zulässig. (Die Werte 16 und 32 werden im Folgenden weggelassen.) Man spricht von einer ganzen, halben, Viertel- oder Achtel-Note.

Aus dieser verbalen Beschreibung gewinnen wir eine formale Grammatik $G = (N, T, P, s)$, $N = \{\text{Musikstueck}, \text{Oktave}, \text{Laenge}, \text{Song}, \text{Note}, \text{Ton}\}$, $T = \{-, \text{C}, \text{D}, \text{E}, \text{F}, \text{G}, \text{A}, \text{H}, 0, 1, 2, 4, 8\}$, $s = \text{Musikstueck}$, mit den Regeln:

| | |
|-------------|---|
| Musikstueck | \rightarrow Song |
| Song | \rightarrow EPSILON Note Song |
| Note | \rightarrow Ton Oktave - Laenge |
| Ton | \rightarrow C D E F G H A |
| Oktave | \rightarrow 0 1 |
| Laenge | \rightarrow 1 2 4 8 |

Damit ist die Syntax der Quellsprache definiert.

Übung 9.1

Verwenden Sie FLACI (Kontextfreie Grammatiken), um eine kfG für einen ML-Parser zu definieren. Testen Sie die syntaktische Korrektheit des Wortes G0-4 E0-4 E0-2 F0-4 D0-4 D0-2 C0-4 D0-4 E0-4 F0-4 G0-4 G0-4 G0-2 G0-4 E0-4 E0-2 F0-4 D0-4 D0-2 C0-4 E0-4 G0-4 G0-4 C0-2.



Wir wenden uns nun dem Übersetzungskonzept für Wörter aus ML zu. Zwei Wünsche gibt es:

- akustischer ML-*Interpreter*, hörbar über den Lautsprecher,
- ML → SVG-*Compiler* mit anschließender SVG-Interpretation für die heute übliche grafische Notendarstellung.

9.2.2 Der ML-Interpreter

Um die Musikstücke hörbar zu machen, verwenden wir die JS-Bibliothek `musical.js`, die in FLACI (ohne zusätzliche Maßnahmen) zur Verfügung steht. Ausführliche Informationen dazu findet man auf <https://github.com/PencilCode/musical.js?files=1>. Das in Abbildung 9.1 dargestellte Übersetzungsmodell umfasst die Herstellung und Anwendung (ganz rechts) eines ML-Interpreters.

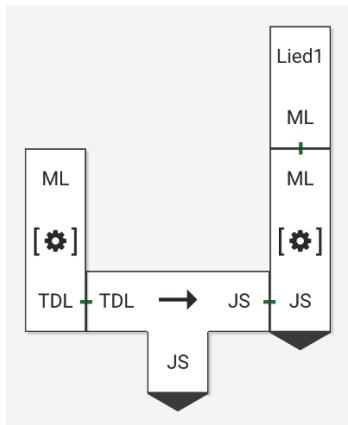


Abbildung 9.1: ML-Interpreter mit `musical.js`

Bei der Erarbeitung der Interpreterbeschreibung in TDL gehen wir zunächst genauso vor wie bei einer Compilerbeschreibung. Zielcode (Ausgabedatei) wird beim Interpreter jedoch *nicht* erzeugt.

Die weiter oben angegebenen Produktionen für ML legen die Tokenbildung für Laenge, Ton, Oktave und natürlich - nahe.

Dies drückt sich auch in der folgenden reduzierten Grammatik $G = (N, T, P, s)$ aus, mit $N = \{\text{Musikstueck}, \text{Song}, \text{Note}\}$, $s = \text{Musikstueck}$, $T = \{0, 1, -, \text{Laenge}, \text{Ton}, \text{Oktave}\}$ und den Produktionen:

| | |
|--------------------------|---|
| <code>Musikstueck</code> | \rightarrow <code>Song</code> |
| <code>Song</code> | \rightarrow <code>EPSILON</code> <code>Note Song</code> |
| <code>Note</code> | \rightarrow <code>Ton Oktave - Laenge</code> |



Computerübung 9.1

Um einen Parser für die reduzierte Grammatik G zu entwickeln, gehen Sie wie folgt vor: FLACI-Grammatik für die Musiksprache ML duplizieren und das Duplikat anschließend reduzieren und umbenennen. Achtung: Ein Wort der Sprache ML gehört nicht zu der durch die reduzierte Grammatik definierten Sprache.

Sie sollten in FLACI (Compiler und Interpreter) die Option „neuen Interpreter (mit TDL von Grammatik)“ auswählen und danach „ML-reduziert“ verwenden.

Bei der wie oben vorgeschlagenen Tokenbildung, die dieser reduzierten Grammatik folgt, gibt es jedoch eine „Überschneidung“: Das Zeichen 1 kann sowohl ein Lexem der Tokenklasse Laenge als auch der Tokenklasse Oktave sein.

Da FLACI zuerst die *Tokendefinitionen von oben nach unten* anzuwenden versucht, muss die Definition von Oktave vor der von Laenge stehen. Dann käme allerdings die 1 als Tonlänge niemals in Betracht. Der Parser (nicht der Scanner) müsste die erforderliche Entscheidung treffen.

Durch eine kleine Veränderung der Tokendefinition erreichen wir es dennoch, das Matching für Oktave von dem für Laenge im Scanner zu unterscheiden: Wir nehmen zur Tokendefinition von Oktave den Bindestrich – hinzu: Die längste für Oktave passende Zeichenfolge ist dann 1–, s. Abbildung 9.2.

Token-
definitionen

Scanner: + Neues Token

Token: Oktave
 Expression: 0-|1-

Token: Laenge
 Expression: 1|2|4|8

Token: Ton
 Expression: A|H|C|D|E|F|G

Token: IGNORE
 Expression: [\t\r\n\s]
Hinweis: diese Zeichen werden in der Eingabe ignoriert.

Scanner testen

Eingabe:

```
1 G0-4 E1-4 E1-1 F0-4
```

Ausgabe:

▶ SCAN

```
1 ([Ton, "G"], [Oktave, "0-"], [Laenge, "4"], [Ton, "E"], [Oktave, "1-"], [Laenge, "4"], [Ton, "E"], [Oktave, "1-"], [Laenge, "1"], [Ton, "F"], [Oktave, "0-"], [Laenge, "4"])
```

Abbildung 9.2: ML-Scanner unter Beachtung der Definitionsreihenfolge der Token



Computerübung 9.2

Vertauschen Sie die Reihenfolge der Tokendefinition von Oktave mit der von Laenge und wiederholen Sie den Scan. Erklären Sie die dann auftretende Fehlermeldung.

Schließlich muss noch die syntaktische Regel für Note angepasst werden:

Note → Ton Oktave Laenge

Nun sind Scanner und Parser vollständig beschrieben und wir können uns den semantischen Aktionen zuwenden.

Der *globale Code* für die semantischen Regeln enthält lediglich den JS-Code für die Instanziierung eines Pianos

```
var piano = new Instrument();
```

und die Initialisierung einer globalen Variablen für die jeweils aktuelle Zeit:

```
var currentTime = 0;
```

Dieser Zeitwert wird benötigt, um den jeweiligen Ton zur richtigen Zeit abzuspielen. Weitere Erklärungen folgen. Die Qualität der Musikwiedergabe kann bei Bedarf beispielsweise durch die Angabe einiger Attribute, wie

```
piano.setTimbre({wave:'sawtooth', gain:0.15,
    attack:0.008, decay:0.2, release:0.2,
    cutoff:0, cutfollow:20, resonance:3});
```

verbessert werden. Das ist hier nicht das Ziel. Die *den Produktionen zugeordneten semantischen Regeln* (nicht: globaler Code) sind im Folgenden dargestellt:

```

Musikstueck      -> Song           $$ = $1;
Song            -> EPSILON        $$ = '';
                    | Note Song       $$ = $1 + $2;
Note           -> Ton Oktave Laenge ...  

...             if ($2 == "0-") $1 = $1; // bleibt groß-C
                if ($2 == "1-") $1 = $1.toLowerCase(); // c
                if ($1 == "H") $1 = "B";
                if ($1 == "h") $1 = "b";  

currentTime += 1000/parseInt($3);  

function spielMusik(frequenz, laenge) {
    setTimeout(function(){
```

```

        piano.tone(frequenz,1,1/parseInt(laenge));
        // Frequenz, Lautst., Dauer
    }, currentTime);
}

spielMusik($1,$3);

$$ = $1 + $2 + $3 + " "; // wenn Compiler

```

Um diese semantische Regel formulieren zu können, braucht man ein wenig Kenntnis von der Zielsprache. Kern der Interpretation einer Note ist natürlich die Tonerzeugung mit

```
piano.tone(frequency, volume, duration)
```

`frequency` nimmt die Töne C,D,E,F,G,A,B und c,d,e,f,g,a,b und bestimmt über die Groß- bzw. Kleinschreibung die eingestrichene (0) bzw. zweigestrichene (1) Oktave. Die semantische Regel empfängt den jeweiligen Ton-Buchstaben in `$1`, verwendet `$2` zur Oktaven- und damit Schreibweisenbestimmung (groß/klein) und muss `H` zu `B` und `h` zu `b` umwandeln, da es `H` und `h` in der Zielsprache nicht gibt. Die Ton-Dauer (`duration`) ist $1, \frac{1}{2}, \frac{1}{4}$ oder $\frac{1}{8}$.

Damit sind die ersten vier `if`-Zeilen erklärt. Würden wir nun einfach mit `piano.tone($1,1,1/parseInt($3));` fortsetzen, ergäbe sich ein „Mischgeräusch“, denn die Berechnungen erfolgen sehr viel schneller als die Tonerzeugung und die einzelnen Töne würden sich überlagern.

Wir benötigen also den jeweils richtige Zeitpunkt zur Tonerzeugung. Dafür wird der Wert der globalen Variablen `currentTime` für jede Note mit `1000/Laenge` inkrementiert (`Laenge = 1,2,4,8` in `$3`). Dies entspricht gerade $1s, \frac{1}{2}s, \frac{1}{4}s$ oder $\frac{1}{8}s$, da `1000` (ohne Einheit) für `1000 ms = 1s` steht.

Nun müssen wir dafür sorgen, dass die Tonerzeugung zu dem jeweils berechneten Zeitpunkt stattfindet. Mit anderen Worten: Der Aufruf von `piano.tone` darf nicht sofort, sondern erst nach einer „Zeitverzögerung“ in Höhe von `currentTime` stattfinden. Man spricht von *Scheduling a call*. Dafür gibt es in der Web-API der etablierten Web-Browser (nicht in JavaScript selbst) eine Funktion (`WindowTimers.setTimeout()`) mit folgender Syntax:

```
setTimeout(f[, t, x0, x1, x2, ..., xn])
```

Ein entsprechender Aufruf bewirkt, dass `f(x0, x1, x2, ..., xn)`, d.h. der Aufruf der anonymen Funktion, *frühestens* nach `t ms` (ms = Millisekunden; `1000 ms = 1 s`; Vorgabewert: 0) ausgeführt wird.

Die (anonyme) Funktion `f` in unserem Beispiel ist nullstellig:

Scheduling a call

```
setTimeout(
    function(){piano.tone(frequenz,1,1/parseInt(laenge));},
    currentTime);
```

Didaktischer Hinweis 9.2

An dieser Stelle verlassen wir die gewohnte Welt der synchronen Programmierung, die der Anweisungssequenz eines Skripts (Programms) folgt. Ein kleiner Exkurs „Asynchrone Verarbeitung in JavaScript“ (inkl. Web-API) ist erforderlich, um das Sprachelement `setTimeout` im Web-Browser richtig zu verstehen und korrekt einzusetzen.

Beispiel 9.1

Um eine wertmäßige Parametervermittlung zu sichern, umgibt man den `setTimeout`-Aufruf mit einer Prozedur, in folgendem Codebeispiel mit `setTimeoutProc`.

```
function setTimeoutProc(wert) {
    setTimeout(function() {
        console.log(wert);
    }, 0);
}

for (var i = 0; i < 3; i++) setTimeoutProc(i);
// Ausgabe: 0,1,2
```

Nun ist die semantische Regel

```
function spielMusik(frequenz, laenge) {
    setTimeout(function(){
        piano.tone(frequenz,1,1/parseInt(laenge));
        // Frequenz, Lautst., Dauer
    }, currentTime);
}

spielMusik($1,$3);
```

in unserem Musikbeispiel komplett.

Didaktischer Hinweis 9.3

Möglichkeit zur Binnendifferenzierung: Alternativ ist es möglich, auf die Betrachtung dieser JS-spezifischen Verarbeitungsaspekte zu verzichten und diesen kleinen Codeabschnitt einfach zu übernehmen.

Computerübung 9.3

Verwenden Sie FLACI zur Erzeugung des vollständig vorbereiteten ML-Interpreters und lassen Sie wenigstens das weiter oben angegebene Musikstück erklingen.

Computerübung 9.4

Verwenden Sie FLACI zur Erzeugung des ML-Interpreters und bauen Sie einen Syntaxfehler in das Eingabewort ein. Beschreiben und begründen Sie die Reaktion von FLACI.



9.2.3 Der ML→SVG-Compiler

Im Folgenden widmen wir uns dem Wunsch nach einem ML→SVG-Compiler. Die Notendarstellung soll der heute vorherrschenden ähnlich sein. Für die Darstellung der Noten auf den fünf Linien ist SVG als Zielsprache eine sehr gute Wahl.

Der Koordinatenursprung befindet sich in der linken oberen Ecke. Die x-Achse verläuft mit aufsteigenden Werten waagerecht nach rechts und die y-Achse senkrecht nach unten. Dies wird auch in Abschnitt 9.3.4 beschrieben bzw. angewandt.

Im globalen Code für die semantischen Regeln vermerken wir deshalb als Startposition für die erste Note in der Zeile: `var x = 20;`.

SVG steht für scalable vector graphics und ist ein Vektorgrafikformat, das ohne Zusatzmaßnahmen von einem Web-Browser interpretiert werden kann. Wenn wir – wie üblich – in den Inhalt der Zielcode-Datei schauen und den Schalter auf „Als HTML anzeigen“ stellen, möchten wir ein Notenblatt sehen. In Abbildung 9.3 ist dafür kein separater SVG-Interpreter-Baustein angegeben.

SVG

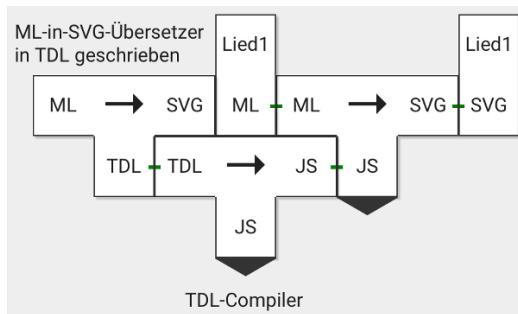


Abbildung 9.3: ML→SVG-Compiler mit SVG-Interpretation

Didaktischer Hinweis 9.4

Die Sprache SVG im Detail zu beschreiben und zu erlernen ist für dieses Projekt keinesfalls erforderlich. Die drei beschriebenen Sprachbefehle genügen für die Darstellung. Sie können bei Bedarf nachgeschlagen werden.



Für ein Notenblatt geben wir zunächst ein Grundgerüst in SVG vor, s. Abbildung 9.4. Die gewählte Maximalbreite des Blattes beträgt 1200. Damit sind die

Notenzeilen auf etwa 30 Noten beschränkt. Dies reicht für unsere kleinen Song-Beispiele vollkommen aus. Möchte man längere Lieder dargestellen, kann dieser Wert entsprechend erhöht werden.

S-Attribut für Musikstück -> Lied

```

1 $$ = '<svg width="1200" height="100" xmlns="http://www.w3.org/2000/svg" xmlns:xlink="http://www.w3.org/1999/xlink">+'\n';
2 $$ = $$ + ' <rect x="0" y="0" width="1200" height="100" fill="lightyellow" />+'\n';
3 $$ = $$ + ' <line x1="0" y1="30" x2="1200" y2="30" style="stroke:black;stroke-width:2"/>+'\n';
4 $$ = $$ + ' <line x1="0" y1="40" x2="1200" y2="40" style="stroke:black;stroke-width:2"/>+'\n';
5 $$ = $$ + ' <line x1="0" y1="50" x2="1200" y2="50" style="stroke:black;stroke-width:2"/>+'\n';
6 $$ = $$ + ' <line x1="0" y1="60" x2="1200" y2="60" style="stroke:black;stroke-width:2"/>+'\n';
7 $$ = $$ + ' <line x1="0" y1="70" x2="1200" y2="70" style="stroke:black;stroke-width:2"/>+'\n';

```

Abbildung 9.4: Semantische Regeln für Musikstück

Das Dokument beinhaltet die typischen XML-Header-Informationen, die für SVG-Code notwendig sind. Darauf hinaus legt es ein gelbes Rechteck mit fünf schwarzen Notenlinien an. Diese befinden sich in den Y-Positionen 30, 40, 50, 60 und 70. Die Notenlinien werden für die Zeichenanweisungen als Referenzpunkte dienen. Die Note C0 hat damit den Y-Wert 80 und die Note F0 hat $Y = 65$. Je Tonhöhe ergibt sich so ein Abstand von 5 Einheiten auf der Y-Achse.

```

<?xml version="1.0" ?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 20010904//EN"
"http://www.w3.org/TR/2001/REC-SVG-20010904/DTD/svg10.dtd">
<svg width="1200" height="100" xmlns="http://www.w3.org/2000/svg"
      xmlns:xlink="http://www.w3.org/1999/xlink">
    <rect x="0" y="0" width="1200" height="100" fill="lightyellow" />
    <line x1="0" y1="30" x2="1200" y2="30" style="stroke:black;stroke-width:2"/>
    <line x1="0" y1="40" x2="1200" y2="40" style="stroke:black;stroke-width:2"/>
    <line x1="0" y1="50" x2="1200" y2="50" style="stroke:black;stroke-width:2"/>
    <line x1="0" y1="60" x2="1200" y2="60" style="stroke:black;stroke-width:2"/>
    <line x1="0" y1="70" x2="1200" y2="70" style="stroke:black;stroke-width:2"/>
</svg>';

```

Didaktischer Hinweis 9.5

Für den Musikfreund sind Unterschiede zur Notenlehre zu beachten, s. Abbildung 9.5. So fehlt hier beispielsweise der entsprechende Notenschlüssel und der Notenhals wird ab der dritten Notenlinie nicht nach unten ausgerichtet, s. Abbildung 9.7. Die beiden darstellten Oktaven entsprechen denen in Abbildung 9.5. Diese werden durch Groß-/Kleinschreibung der Tonhöhen identifiziert, s. Abbildung 9.6.



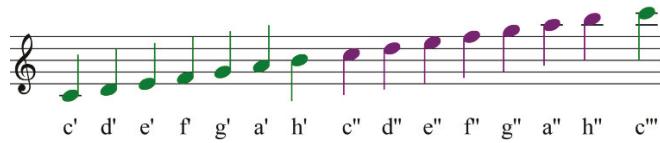


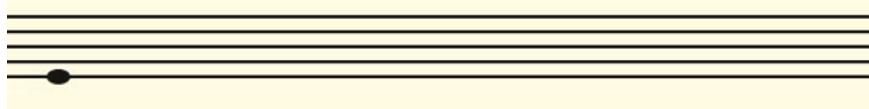
Abbildung 9.5: Zwei Oktaven in der Musik

| Abc | Noten |
|---------------|-------|
| C D E F G A B | |
| c d e f g a b | |

Abbildung 9.6: Zwei Oktaven in der Zielsprache: Groß-/Kleinbuchstaben; b/B für h/H

Eine Note zeichnen wir als einfache Ellipse. Die zugehörige *SVG*-Anweisung ist:

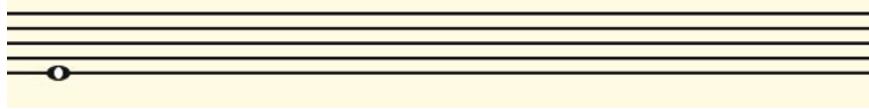
```
<ellipse CX="20" CY="70" RX="8" RY="5" style="fill:black;"/>
```



Dabei handelt es sich um eine E-Note, was leicht an CY zu erkennen ist. CX legt die Position der Note auf der Notenzeile fest. Mit jeder gezeichneten Note muss dieser Wert ansteigen. RX und RY definieren die Größe der Ellipse und bleiben konstant.

Das Aussehen der Noten richtet sich nach ihrem Wert: Für ganze und halbe Noten können wir eine kleinere weiße Ellipse in die Notenmitte zeichnen, damit der bekannte Hohlraum-Effekt entsteht:

```
<ellipse cx="20" cy="70" rx="3" ry="4" style="fill:white;"/>
```



C0-4 D0-4 E0-4 F0-4 G0-4 A0-4 H0-4

C1-2 D1-2 E1-2 F1-2 G1-2 A1-2 H1-2



Abbildung 9.7: Oktave 0 und Oktave 1 in ML und SVG

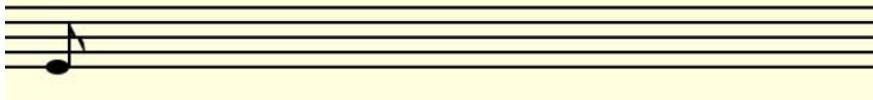
Für halbe, Viertel- und Achtelnoten benötigen wir einen Notenhals. Diesen können wir durch eine einfach Line (etwa 30 Einheiten lang) darstellen. Da dieser ans rechte Ende der Ellipse gezeichnet wird, ergibt sich $X1 = X2 = CX + RX$:

```
<line X1="28" Y1="70" X2="28" Y2="40" style="stroke:black; stroke-width:2"/>
```



Zusätzlich brauchen wir für halbe, Viertel- und Achtelnoten noch ein Fähnchen am Notenhals. Hierfür verwenden wir einen Pfad (Path). Die Startposition (20,40) muss für die jeweilige Note angepasst werden, die restlichen Anweisungen und Parameter bleiben unverändert:

```
<path d="M 20,40 q 0,10 5,10 t 5,10" style="stroke:black;" />
```



Verwendet man Groß-/Kleinbuchstaben für diese Befehle, beziehen sich die Parameter auf absolute/relative Koordinaten.

Damit sind die in Abbildung 9.8 angegebenen semantischen Regeln vollständig vorbereitet.

Computerübung 9.5



Vervollständigen Sie die $ML \rightarrow SVG$ -Beschreibung und erzeugen Sie einen dementsprechenden Compiler.

Abschließend lassen wir uns noch einmal das Lied

```
G0-4 E0-4 E0-2 F0-4 D0-4 D0-2  
C0-4 D0-4 E0-4 F0-4 G0-4 G0-4 G0-2  
G0-4 E0-4 E0-2 F0-4 D0-4 D0-2  
C0-4 E0-4 G0-4 G0-4 C0-2
```

vorspielen und betrachten dessen SVG-Darstellung, s. Abbildung 9.9.

In Abbildung 9.9 ist (bei entsprechender Schalterstellung in FLACI) die Interpretation der SVG-Datei als Wort der Sprache HTML vorgenommen worden. Anstelle der HTML-Interpretation sind beliebige andere Interpretationen denkbar. Der jeweils konstruierte Interpreter-Baustein wird dann einfach unter die SVG-Datei gestellt.

Computerübung 9.6



Entwickeln Sie einen nonsense-Interpreter, der die Anzahl der Elipsen bestimmt und als Interpretationsergebnis zurückgibt.

S-Attribut für Note -> Ton Oktave Laenge

```

1 var cy = 0;
2
3 $2 = $2.substring(0,1); // - am Ende entfernen
4
5 // auf die Oktaven 0 und 1 beschränkt
6 if ($1+$2 == 'C0') cy = 80; if ($1+$2 == 'C1') cy = 45;
7 if ($1+$2 == 'D0') cy = 75; if ($1+$2 == 'D1') cy = 40;
8 if ($1+$2 == 'E0') cy = 70; if ($1+$2 == 'E1') cy = 35;
9 if ($1+$2 == 'F0') cy = 65; if ($1+$2 == 'F1') cy = 30;
10 if ($1+$2 == 'G0') cy = 60; if ($1+$2 == 'G1') cy = 25;
11 if ($1+$2 == 'A0') cy = 55; if ($1+$2 == 'A1') cy = 20;
12 if ($1+$2 == 'H0') cy = 50; if ($1+$2 == 'H1') cy = 15;
13
14 $$ = '<ellipse cx='+(x)+'" cy="'+(cy)+'" rx="8" ry="5" style="fill:black;"/>';
15
16 // Notenhals für Halbe-, Viertel- und Achtel-Noten
17 if ($3 == '2' || $3 == '4' || $3 == '8')
18 $$ = $$ + '<line x1='+(x+8)+'" y1="'+(cy)+'" x2='+(x+8)+'" y2="'+
19 +(cy-30)+'" style="stroke:black;stroke-width:2"/>';
20
21 // Ganze und halbe Note sind hohl = werden weiß gefüllt
22 if ($3 == '1' || $3 == '2')
23 $$ = $$ + '<ellipse cx='+(x)+'" cy="'+(cy)+'" rx="3" ry="4" style="fill:white;"/>';
24
25 // Fähnchen exemplarisch für die Achtelnote (16-tel und 32-tel haben auch Fähnchen)
26 if ($3 == '8')
27 $$ = $$ + '<path d="M '+ (x+8) +' q 0,10 5,10 t 5,10" style="stroke:black;"/>';
28
29 // Abstand der Noten auf der Linie
30 x = x + 40;
31

```

Abbildung 9.8: S-Attribut-Note



Abbildung 9.9: S-Attribut-Note

Computerübung 9.7

Übersetzen Sie ein in ML vorliegendes Musikstück in QR-Code und scannen Sie ihn mit einem Smartphone oder Tablet ein.



Computerübung 9.8

Entwickeln Sie einen Compiler, der wie ein Filter wirkt und einen beliebigen Text „vorspielt“. Entscheiden Sie sich für eine geeignete Zielcode-Interpretation.



9.3 Projekt: Schachnotation

9.3.1 Die Forsyth-Edwards-Notation (FEN)

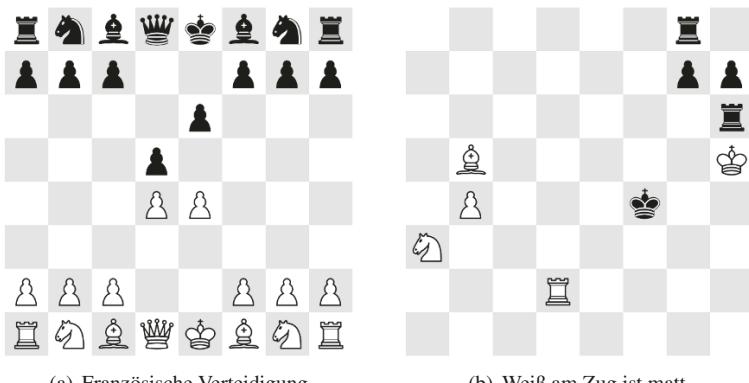
Schachspielen ist eine feine Sache. Aber auch eine weltmeisterliche Partie ist vergänglich, wenn sie für die Nachwelt nicht festgehalten wird. Um den *Spiel-*

algebraische Notation *verlauf* zu dokumentieren, notieren deshalb beide Spieler alle Halbzüge von Weiß und Schwarz in einer *algebraischen Notation*. Man kann dann das ganze Spiel rekonstruieren, indem man die Partie Zug für Zug nachspielt.

textuelle
Notation
FEN
SVG

Möchte man jedoch ausgewählte *Figurenstellungen* betrachten, etwa um deren Bewertung zu diskutieren oder eine Aufgabe im *Problemschach* zu formulieren, wäre ein fotografischer Schnappschuss eine angemessene Technologie. Diese hat jedoch den Nachteil, wenigstens nicht mit einfachen Mitteln verarbeitbar zu sein: Im Rahmen der Stellungsanalyse werden alternative (Halb)Züge diskutiert, vorgeschlagene Fortsetzungen ein stückweit ausgeführt und wieder zurückgestellt. Mit grafischen Repräsentationen lässt sich das schlecht machen, aber eine *textuelle Notation*, die jeweils eine Stellung beschreibt, ist dafür sehr gut geeignet.

Wir befassen uns hier mit einer solchen textbasierten Notation: der *FEN*. Natürlich erwarten wir in deren Verarbeitung optisch ansprechende Schachdiagramme, d.h. grafische Darstellungen von Figurenstellungen auf einem 8x8-Schachbrett, wie in Abbildung 9.10. Dies kann man (wie die Beschreibung der Notenblätter in Abschnitt 9.2) mit *SVG* erreichen.



(a) Französische Verteidigung

(b) Weiß am Zug ist matt.

Abbildung 9.10: Grafische Notation zweier Schachstellungen

Nach der Eröffnung einer Partie im Stil der französischen Verteidigung ergibt sich die im linken Teil von Abbildung 9.10 dargestellte Stellung. Rechts davon ist das Ende einer Partie gezeigt: Schwarz setzt Weiß matt.

9.3.2 Definition und Anwendung der FEN

Es geht also um ein *Übersetzungsprojekt aus der Sprache FEN in SVG*.

Figuren Schnell lernt man, wie die einzelnen *Figuren* heißen und welche Symbole dafür verwendet werden: K (king, König), Q (queen, Dame), R (rook, Turm), B (bishop, Läufer), N (knight, Springer), P (pawn, Bauer). Die angegebenen Symbole gibt es

auch als Kleinbuchstaben k, q, r, b, n und p. Während die Großbuchstaben für die weißen Figuren stehen, bezeichnen die Kleinbuchstaben die schwarzen.

Die Stellungsnotation mit FEN erfolgt reihenweise (*Reihe* im Schach = Zeile einer Matrix), oben links beginnend. In der Anfangsaufstellung steht dort der schwarze Turm. Jede der acht Reihen-Beschreibungen wird mit einem Schrägstrich von der jeweils nächsten getrennt.

Jede Reihe wird durch eine Folge von Symbolen für die positionierten Figuren oder Ziffern (1-8) für die *Anzahl der aufeinanderfolgenden Leerfelder* beschrieben. Die Summe der Figuren und der Leerfelder in einer Reihe muss gleich 8, d.h. gleich der Anzahl der *Linien* (Spalten) eines Schachbretts, sein. Die Zahl 8 (und nichts weiter) bedeutet, dass in der betreffenden Reihe keine einzige Figur steht.

Für die in Abbildung 9.10 gezeigten Stellungen lauten die zugehörigen FEN:

- (a) rnbqkbnr/ppp2ppp/4p3/3p4/3PP3/8/PPP2PPP/RNBQKBNR
- (b) 6r1/6pp/7r/1B5K/1P3k2/N7/3R4/8

Die FEN ist damit noch nicht ausgeschöpft. Sie ist insgesamt in sechs Gruppen aufgeteilt, die durch Leerzeichen voneinander getrennt sind. Diese geben jeweils an: *Figurenstellung*, Zugrecht, Rochaderechte, möglicher En-passant-Schlag, gespielte Halbzüge seit dem letzten Bauernzug oder dem Schlagen einer Figur und Nummer des nächsten Zuges.

Wir benötigen nur den ersten Teil der FEN, nämlich die Figurenstellung, und beschränken uns im Folgenden darauf, ohne das extra zu betonen.

Übung 9.2

Verifizieren Sie die oben angegebenen FEN für die beiden Stellungen aus Abbildung 9.10.

Reihe

Linien



9.3.3 Eine kfG für die FEN (Quellsprache)

Als erstes brauchen wir eine kontextfreie Grammatik $G = (N, T, P, s)$ für die Quellsprache FEN: $T = \{1, 2, 3, 4, 5, 6, 7, 8, R, N, B, Q, K, P, r, n, b, q, k, p, /\}$, $N = \{\text{Schachstellung}, \text{Reihe}, \text{FigurOderLeerfelder}, \text{Schachfigur}, \text{Leerfelder}\}$, $s = \text{Schachstellung}$ und $P = \{$

```

Schachstellung -> Reihe / Reihe / Reihe / Reihe / Reihe /
                           Reihe / Reihe / Reihe
Reihe          -> FigurOderLeerfelder
FigurOderLeerfelder -> Feld | Feld FigurOderLeerfelder
Feld -> Schachfigur | Leerfelder
Schachfigur -> R | N | B | Q | K | P | r | n | b | q | k | p
Leerfelder -> 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8
}
```

Aus Abschnitt 9.3.2 gewinnen wir zwei Forderungen:

1. *Ziffernbedingung*: Es dürfen nicht zwei Ziffern aus 1-8 unmittelbar nebeneinander stehen.
2. *Längenbedingung*: Die Summe aus der Anzahl der Figur-Symbole und der Anzahl der leeren Felder muss für jede Reihe genau 8 ergeben.

Diese beiden Forderungen lassen wir zunächst außer Acht. Wie man sie nachträglich und fundiert einbezieht, zeigen Folgeabschnitte.

Eigentlich müsste man noch prüfen, ob die gezeigten Figuren überhaupt in einem Schachspiel möglich wären. Beispielsweise sollten nicht mehrere gleichfarbige Könige auf dem Brett stehen. Auf eine solche Plausibilitätsprüfung wollen wir hier verzichten, um das Beispiel übersichtlich zu gestalten.



Didaktischer Hinweis 9.6

Die Nichtterminale sollten konsequent so gewählt werden, dass deren Bedeutungen im Anwendungskontext erkennbar sind. Beispielsweise **Leerfelder** statt **Ziffer**.



Computerübung 9.9

Überprüfen Sie mit FLACI (Modul „kontextfreie Grammatik“), ob die beiden in Abbildung 9.10 angegebenen Stellungen syntaktisch korrekten Wörtern der Sprache FEN entsprechen. Geben Sie außerdem vier Wörter an, die zu FEN gehören, die Forderung 1 oder 2, s. Seite 226, jedoch nicht erfüllen.

9.3.4 FEN→SVG-Compiler

Nach dem Vorbild des ML→SVG-Compilers aus Abschnitt 9.2.3 entwerfen wir nun einen FEN→ SVG-Compiler. Wir beginnen mit der Modellierung unter Verwendung der *T-Diagramme* in Abbildung 9.11. Neben dem Entwicklungsprozess

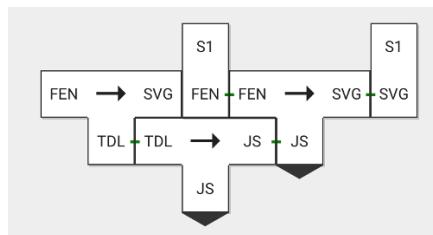


Abbildung 9.11: FEN→SVG-Compiler: Modellierung

(*Compilergenerierung*) ist in Abbildung 9.11 auch die *Anwendung* des erzeugten und lauffähigen FEN→ SVG-Compilers dargestellt. Unter Verwendung des „Compiler-und-Interpreter“-Moduls von FLACI ergeben sich die Scanner- und Parser-Definitionen für FEN wie in Abbildung 9.12.

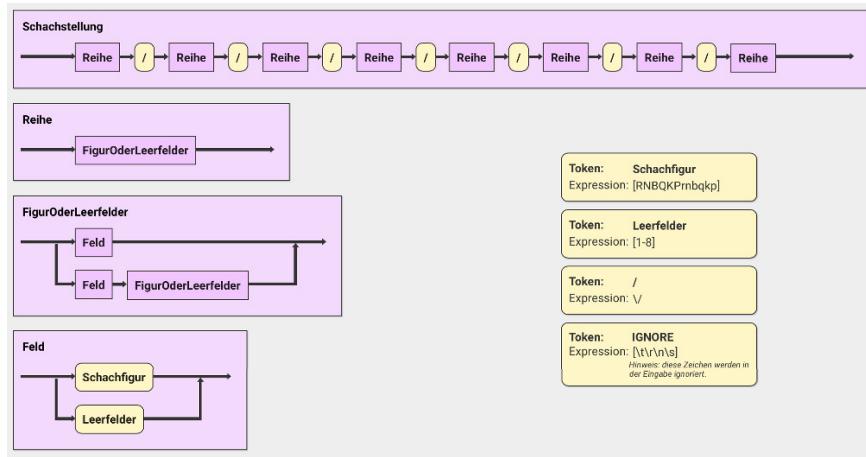


Abbildung 9.12: FEN→SVG-Compiler: Scanner-, Parsergenerierung

Die Definitionen der Tokenklassen (gelb) wurden in Abbildung 9.12 aus Darstellungsgründen rechts hineinkopiert.

Computerübung 9.10

Folgen Sie der in Abbildung 9.12 dargestellten Compilerentwicklung, indem Sie den FEN→SVG-Übersetzungsprozess modellieren und den Parser für FEN aus der oben entwickelten Grammatik G gewinnen. Erzeugen Sie eine *reduzierte Grammatik* durch Einführung der Tokenklassen **Schachfigur** und **Leerfelder**. Prüfen Sie die Arbeit von Scanner und Parser.



reduzierte Grammatik

Im nächsten Schritt ordnen wir den syntaktischen Regeln S-Attribute zu, deren Berechnung den Zielcode erzeugt. Dazu werden nur ein paar wenige Grundkenntnisse zu SVG benötigt.

Computerübung 9.11

Machen Sie sich soweit nötig mit SVG vertraut. Schauen Sie sich auch die Positionierung einer Gruppierung mit dem `transform`-Attribut an.



Die Schachfiguren werden zeilenweise mit Bezug zum Koordinatenursprung von SVG (linke obere Ecke) angeordnet. Zur Positionsberechnung verwenden wir $x = 50 \cdot x + 2.5$ und $y = 50 \cdot y + 2.5$ für $x = 0, 1, 2, \dots, 7$, wobei sich die x -Achse nach rechts und die y -Achse nach unten verlängert.

Jede Schachfigur bringt deshalb in SVG eine Positionsangabe mit:

```
'<g transform="translate('+(x*50+2.5)+','+(y*50+2.5)+')" ... >
...
</g>;'
```

Wir benötigen zwei, jeweils mit 0 zu initialisierende globale Variablen x und y :

```
var x = 0;
var y = 0;
```

Die SVG-Codierungen der Schachfiguren werden für die syntaktische Regel Feld -> Schachfigur als Werte einzelner (lokal gültiger) Variablen hinterlegt. Weiße und schwarze Schachfiguren stehen im SVG-Format zur Verfügung: <https://programmingwiki.de/Schach>. Außerdem brauchen wir eine Auswahlanweisung, die für den betreffenden Tokenwert die zugehörige Figurcodierung bestimmt und zurückgibt.

```
// Token -> SVG für die betreffende Figur
switch ($1) {
    case "r": $$ = blackRook; break;
    case "n": $$ = blackKnight; break;
    case "b": $$ = blackBishop; break;
    case "q": $$ = blackQueen; break;
    case "k": $$ = blackKing; break;
    case "p": $$ = blackPawn; break;

    case "R": $$ = whiteRook; break;
    case "N": $$ = whiteKnight; break;
    case "B": $$ = whiteBishop; break;
    case "Q": $$ = whiteQueen; break;
    case "K": $$ = whiteKing; break;
    case "P": $$ = whitePawn; break;
    default: $$ = "";
}

x = x + 1; // ein Feld weiter (nach rechts)
```

Um Schachfiguren in eine bestimmte Reihe „zu stellen“, d.h. für Reihe -> FigurOderLeerfelder, werden die x-y-Koordinaten beeinflusst:

```
y = y + 1; // nächste Zeile
x = 0; // wieder an den Anfang
$$ = $1;
```

Dies gilt auch für das „Einfügen“ von Leerfeldern, d.h. Feld -> Leerfelder, was in der Zielsprache eine Verschiebung der Schreibposition auf das damit bestimmte Feld bedeutet:

```
// Anzahl freier Felder als Zahl
x = x + parseInt($1);
$$ = "";
```

Für die beiden syntaktischen Regeln

FigurOderLeerfelder -> **Feld | Feld FigurOderLeerfelder**
 werden lediglich die (verketteten) S-Attributwerte der genannten Nichtterminale zurück gegeben: **\$\$=\$\$1**; bzw. **\$\$=\$\$1+\$2**; .

Für die Regel des Spitzensymbols ist noch der SVG-Code, in den die SVG-Beiträge der Figuren in den acht Reihen eingebettet werden, anzugeben. Außerdem muss das Schachbrett mit SVG beschrieben werden.

```
var SchachbrettSVG =
'<svg xmlns="http://www.w3.org/2000/svg" width="300"
height="300" viewBox="0 0 400 400"><rect width="400"
height="400" fill="#DDD"/><path fill="#FFF" d="M0
0H400v50H0zm0 100H400v50H0zm0 100H400v50H0zm0
100H400v50H0zM50 0V400h50V0zm100 0V400h50V0zm100
0V400h50V0zm100 0V400h50V0z"/>' ;
var Figuren = $1 + $3 + $5 + $7 + $9 + $11 + $13 + $15;
return SchachbrettSVG + Figuren + '</svg>' ;
```

Computerübung 9.12

Vervollständigen Sie die S-Attribute im FEN→SVG-T-Diagramm und führen Sie mindestens für die in Abbildung 9.10 angegebenen FEN-Eingabewörter Compilationen in SVG durch. Wählen Sie die HTML-Ansicht (per Schalter) für die Ausgabe, um die jeweils erwarteten Schachdiagramme zu sehen. Ein vollständiges Ergebnis finden Sie unter <https://flaci.com/T54bqdrgr>.



Wenn wir allerdings ein Eingabewort wählen, das wenigstens eine der beiden Forderungen auf Seite 226 verletzt, sehen wir „grafische Verwerfungen“ (ohne Fehlermeldungen), wie beispielsweise in Abbildung 9.13 für **rnbqkbnr/ppp2ppp/4p3/3p4/311qqPP2/8/PPP2PPP/RNBQKBNR** .



Abbildung 9.13: Verstoß gegen die Forderungen 1 und 2

Das oben angegebene Eingabewort enthält für **311qqPP2** gleich mehrere Verstöße gegen die FEN-Definition (s. Seite 226): Mit 311 stehen sogar drei Ziffern nebeneinander und die Summe der leeren Felder und Figuren dieser Reihe ist 11 statt 8.



Computerübung 9.13

Probieren Sie es aus und compilieren Sie das fehlerhafte Wort: Zwei schwarze Damen „drängen“ die beiden weißen Bauern nach rechts außen.

9.3.5 Mehrphasen-Compilation

Wörter, die eine oder beide syntaktischen FEN-Forderungen auf Seite 226 nicht erfüllen und damit unzulässige Schachstellungen repräsentieren, dürfen nicht akzeptiert werden.

Es gibt nun verschiedene Möglichkeiten damit umzugehen:

- Wir könnten den FEN→SVG-Compiler hernehmen und modifizieren (erweitern). Dies ist ein fehleranfälliges Vorhaben, denn wir würden riskieren, dass ein (höchstwahrscheinlich) korrekter Baustein nicht mehr funktioniert.
- Von daher ist es angebracht, den FEN→SVG-Compiler unverändert zu übernehmen und einen FEN1-Parser für Forderung 1 von Seite 226 (Ziffernbedingung) und danach einen FENE-Parser für Forderung 2 (Längenbedingung) voranzustellen. Durch eine geschickte Wahl einer Zwischensprache FENE (zwischen FEN1 und SVG), kann man den zugehörigen Compilationsprozess vereinfachen: FEN1→FENE→FEN→SVG, s. Abbildung 9.14. Dieses Verfahren nennt man *Mehrphasen-Compilation*.

Zwischensprache

Mehrphasen-Compilation

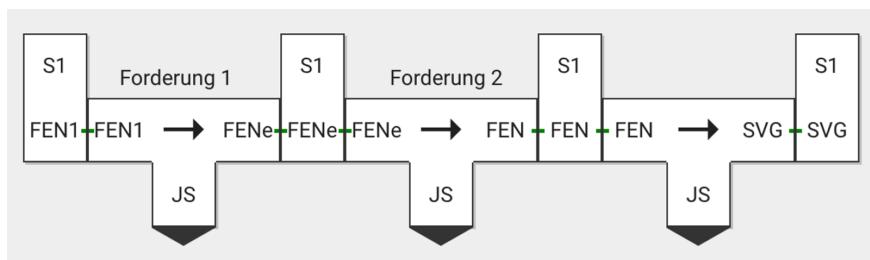


Abbildung 9.14: Mehrphasen-Compilation: FEN1→FENE→FEN→SVG

Ein FEN1-Parser ist mit dem Compiler-Interpreter-Modul von FLACI leicht herzustellen, indem die S-Attribute der Vorgabe (grau) unverändert übernommen werden. Lediglich für die Regel des Spitzensymbols ist ein `return $$;` zu ergänzen. Auf diese Weise entsteht ein FEN1→FEN1-Compiler, der genau alle korrekten Wörter aus FEN1 „durchlässt“ und anderenfalls mit einem Syntaxfehler abbricht.

Den dann folgenden Parser für die Prüfung der Längenbedingung können wir *vereinfachen*, indem wir die Zielsprache FEN1 des FEN1→FEN1-Compilers leicht verändern zu FENe. Man spricht dann von einer *Vorübersetzung* (precompilation). Die Wörter aus FENe enthalten auf den *freien* Feldern der Schachstellung ein e für empty („leere Figur“). Dann braucht der FENe-Parser nur noch zu prüfen, ob *genau alle* acht Zeichen *aller* acht Reihennotationen entweder aus einem Figursymbol oder einem e bestehen. Statt eines FEN2→FEN-Compilers haben wir dann einen FENe→FEN-Compiler. Abbildung 9.14 zeigt die beschriebene Mehrphasen-Compilation.

Der bereits entwickelte FEN→SVG-Compiler wird, wie in Abbildung 9.14 zu sehen, in der letzten Übersetzungsphase direkt angeschlossen. Ein Beispiel für eine erfolgreiche Mehrphasen-Compilation ist

Der Compiler (FEN1 => FENe, JS) wird ausgeführt.

Eingabe: 6r1/6pp/7r/1B5K/1P3k2/N7/3R4/8

Ausgabe: eeeeeere/eeeeeepp/eeeeeeeer/eBeeeeek/
ePeeekee/Neeeeeee/eeeReeee/eeeeeeee

Der Compiler (FENe => FEN, JS) wird ausgeführt.

Eingabe: eeeeeere/eeeeeepp/eeeeeeeer/eBeeeeek/
ePeeekee/Neeeeeee/eeeReeee/eeeeeeee

Ausgabe: 6r1/6pp/7r/1B5K/1P3k2/N7/3R4/8

Der Compiler (FEN => SVG, JS) wird ausgeführt.

Eingabe: 6r1/6pp/7r/1B5K/1P3k2/N7/3R4/8

Ausgabe: <svg> ... </svg>

Für das Wort rnbqbnr/ppp2ppp/4p3/3p4/**311qqPP2/8/PPP2PPP/RNBQKBNR** (beide Forderungen verletzt) wird nun (schon im ersten Schritt) ein Syntaxfehler gemeldet, statt „widerspruchslös“ eine unerwünschte Darstellung erzeugt.

Exemplarisch wollen wir die Entwicklung des FEN1→FENe mit einem NEA beginnen. FLACI stellt alle Werkzeuge bereit, die wir brauchen, um daraus eine Grammatik FEN1-Grammatik zu machen. Zur besseren Übersicht beschränken wir den NEA auf nur eine (der acht „baugleichen“) Reihen, s. Abbildung 9.15.

Computerübung 9.14

Machen Sie sich klar, dass die gewünschte Ziffernbedingung (nicht zwei Ziffern unmittelbar nebeneinander) mit dem NEA aus Abbildung 9.15 erreicht wird.

Aus dem NEA in Abbildung 9.15 erhalten wir die gesuchte kfG für FEN1 durch

1. Konvertierung: NEA in kfG,
2. Transformation: Vereinfachung der Grammatik,

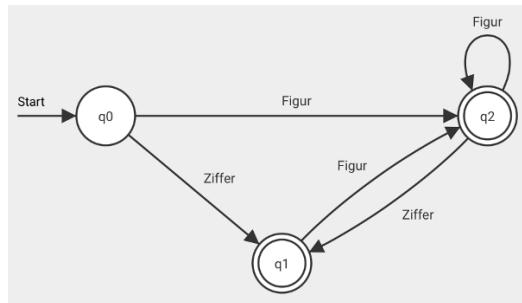


Abbildung 9.15: Überführungsfunktion eines NEA für FEN1 (genau eine Reihe)

3. Umbenennung des Spitzensymbols zu Reihe und
4. Ergänzung der Regel für Schachstellung.

Wir verwenden dabei eine reduzierte Grammatik, d.h., dass Ziffer für ein Ziffernsymbol aus [1-8] und Figur für ein Symbol aus [PRNQKBprnqkb] stehen.

| | | |
|----------------|---------------|--|
| Schachstellung | \rightarrow | Reihe / Reihe / Reihe / Reihe / Reihe |
| | | / Reihe / Reihe / Reihe |
| Reihe | \rightarrow | Figur Reihe Figur Ziffer q1 Ziffer |
| q1 | \rightarrow | Figur Reihe Figur |

Computerübung 9.15

Erzeugen Sie zunächst einen FEN1→FEN1-Compiler und testen Sie ihn.

Wir ergänzen den FEN1→FEN1-Compiler zu einem FEN1→FENe-Compiler, so dass dieser - wie weiter oben erwähnt - die leeren Felder mit e beschreibt. Zu diesem Zweck müssen nur die S-Attribute der Regeln, die auf der rechten Seite Ziffer enthalten, angepasst werden. Für Reihe \rightarrow Ziffer q1 gilt:

```

$$ = "";
var n = parseInt($1);
for (var i = 0; i < n; i++) {
    $$ = $$ + "e";
}
$$ = $$ + $2; // entfällt für Reihe -> Ziffer
  
```

Dadurch werden genau so viele e's geschrieben, wie Ziffer vorgibt und wie über q1 geliefert werden.

Computerübung 9.16

Vollenden Sie den FEN1→FENe-Compiler und testen Sie ihn.

Der Parser des noch zu entwickelnden FENe→FEN-Compilers kennt nur zwei Regeln, nämlich für Schachstellung und Reihe.

```
Schachstellung -> Reihe / Reihe / Reihe / Reihe / Reihe
                           / Reihe / Reihe
Reihe -> FigurOderLeer FigurOderLeer FigurOderLeer FigurOderLeer
                  FigurOderLeer FigurOderLeer FigurOderLeer FigurOderLeer
```

wobei FigurOderLeer ein Zeichen aus [eQPBNKRqpbnkr] ist. (Bitte das e beachten!) Das zugehörige S-Attribut für Schachstellung ist:

```
$$ = $1 + $2 + $3 + $4 + $5 + $6 + $7 + $8 + $9 + $10 + $11
      + $12 + $13 + $14 + $15;
return $$;
```

und die syntaktische Regel für Reihe kann folgendermaßen angegeben werden:

```
var ausgabe = "";
var leer = 0;
var Reihe = [$1,$2,$3,$4,$5,$6,$7,$8];
for(var i = 0; i < 8; i++){
    var symbol = Reihe[i];
    if(symbol == "e") leer++;
    if(symbol != "e") {
        if(leer > 0) ausgabe = ausgabe + leer.toString();
        ausgabe = ausgabe + symbol;
        leer = 0;
    }
}
if(leer > 0) ausgabe = ausgabe + leer.toString();
$$ = ausgabe;
```

Computerübung 9.17

Vervollständigen Sie den FENe→FEN-Compiler und „bestücken“ Sie das Vorüberzeugungsschema aus Abbildung 9.14. Ergänzen Sie außerdem die FEN→SVG-Compilation und testen Sie mit verschiedenen Eingabewörtern.



9.4 Compiler für die Zeichenroboter-Sprache ZR

9.4.1 ZR→PS→PNG-Compiler

An dieser Stelle kehren wir wieder zur Zeichenroboter-Sprache ZR zurück, die ZR

wir in Abschnitt 4.5, s. Seite 117, eingeführt und ausführlich beschrieben haben. In Abschnitt 5.4 wurde der Parse-Prozess von ZR-Programmen angedeutet.

Die in Abbildung 5.17 angegebene vollständige Grammatik für ZR kann durch Scanner-Unterstützung zu einer reduzierten Grammatik, s. Abbildung 5.18, vereinfacht werden. Dann gelten die in Abbildung 9.16 zusammengestellten Tokendefinitionen:

| | |
|--|---|
| Token: VW Expression: VW | Token: KlammerZu Expression: \] |
| Token: RE Expression: RE | Token: KlammerAuf Expression: \ |
| Token: FARBE Expression: FARBE | Token: Zahl Expression: [1-9][0-9]* |
| Token: Farbwert Expression: weiss rot gelb gruen | Token: STIFT Expression: STIFT |
| Token: WH Expression: WH | Token: IGNORE Expression: [\n\r\t\\$] <small>Hinweis: diese Zeichen werden in der Eingabe ignoriert.</small> |

Abbildung 9.16: Tokendefinition für ZR (reduzierte Grammatik)

Nach dem Vorbild von Abschnitt 9.3.5 wollen wir den Compilationsprozess in mehrere Phasen gliedern. Anstelle einer Direktübersetzung eines ZR-Programms in eine pdf-Datei, wie das in Abschnitt 9.4.2 unter Verwendung von jsPDF betrachtet wird, kommt auch eine *Mehrphasencompilation* infrage: Mehrere Compiler werden verkettet, d.h. nacheinander auf das vom jeweils vorhergehenden Compiler erzeugte Compilat angewandt.

Mehrphasen-compilation
PNG
Zwischensprache

Für unser Beispiel modellieren wir in Abbildung 9.17 eine Zweiphasencompilation: ZR→PS→PNG. Die Zielsprache ist *PNG* (Portable Network Graphics) und als *Zwischensprache* verwenden wir *PS* (Postscript).



Didaktischer Hinweis 9.7

Um für ein ZR-Wort den zugehörigen Zielcode in PS zu generieren, benötigen wir nur Basiswissen in PostScript. Da jedes PS-Dokument zur Deklaration einen bestimmten Kopf benötigt, müssen wir diesen am Spitzensymbol vorbereiten.

Postscript

Postscript ist eine Seitenbeschreibungssprache. Sie beschreibt die Elemente einer Seite als geometrische Objekte, d.h. als Linien, Kreise, Rechtecke und Texte, unabhängig von der Druckertechnologie. Es gibt einen Interpreter, der die in *PS*-Programmen enthaltenen Befehle in grafische Darstellungen umsetzt.

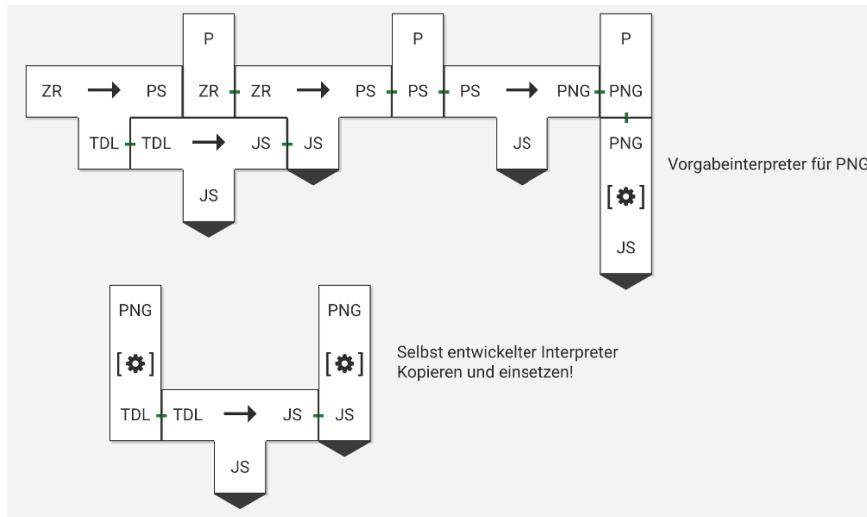


Abbildung 9.17: ZR→PS→PNG-Compilation

Wie der Name andeutet, sind die Befehle so aufgebaut, dass die Operatoren, wie `moveto` oder `lineto`, den Operanden, wie etwa die x - y -Koordinaten eines Punktes, *nachgestellt* sind. Ein solcher Befehl lautet beispielsweise: `120 200 moveto`. Dessen Interpretation setzt den imaginären Zeichenstift auf die Position (120, 200). Intern wird dies durch Stack-Arbeit umgesetzt.

Nutzerdefinierte Prozeduren werden mit dem Kommando `def` vorgenommen und können fortan mit dem gewählten Namen und ggf. entsprechenden konkreten Parametern verwendet werden. In der semantischen Regel für die erste Produktion machen wir davon Gebrauch.

Die Basislängeneinheit in Postscript ist 1 Point = 0.3528 mm. Die Positionen der Objekte einer Postscriptgraphik werden in einem *Koordinatensystem* angegeben. Dieses folgt den Abmessungen einer DIN-A4-Seite: Ursprung (0,0) im Hochformat links unten. Nach rechts in x -Richtung: 210 mm = 595 Pixel, nach oben in y -Richtung: 297 mm = 841 Pixel.

Koordinaten-
system

Didaktischer Hinweis 9.8

Um die (auf die ZR-Befehle eingeschränkte) Arbeit mit PS bequemer zu gestalten, definieren wir uns eigene Befehle und Hilfsvariablen. Diese Definitionen können für eine gut lesbare Beschreibung der damit festzulegenden ZR-Semantik vorgegeben werden.



Programm → Anweisungen:

```
// DIN-A4-Hochformat mit 3x Skalierung (21x29cm)
$$= '0 0 630 870 .gbox \n\
/orient 180 def \n\
```

```

/PI 3.14159265359 def \n\
/xpos 0 def \n\
/ypos 0 def \n\
0 0 0 setrgbcolor \n\
/goto { /ypos exch def /xpos exch def xpos ypos moveto} def \n\
/turn { /orient exch orient add def} def \n\
/draw { /len exch def newpath xpos ypos moveto \n\
/xpos xpos orient PI mul 180 div sin len mul add def \n\
/ypos ypos orient PI mul 180 div cos len mul add def \n\
xpos ypos lineto stroke \n\
} def \n\
315 435 goto \n';
return($$+$1);

```

Computerübung 9.18



Machen Sie sich mit dem hier angegebenen Postscript-Programmfragment unter Verwendung von Internet-Beiträgen vertraut und fügen Sie diesen Code für die angegebene Regel in Ihr T-Diagramm ein. Verfahren Sie mit den folgenden Attributen genauso.

Anweisung → Anweisung Anweisungen:

```
$$ = $1 + "\n" + $2;
```

Anweisung → Anweisung:

```
$$ = $1;
```

Anweisung → WH Zahl KlammerAuf Anweisungen KlammerZu:

```
$$ = "";
for (var i = 0; i < $2; i++)
$$ = $$ + $4;
```

Anweisung → FARBE Farbwert:

```
if ($2 == "blau")    $$ = "0 0 255 setrgbcolor ";
if ($2 == "rot")     $$ = "255 0 0 setrgbcolor ";
if ($2 == "gruen")   $$ = "0 255 0 setrgbcolor ";
if ($2 == "gelb")    $$ = "255 255 0 setrgbcolor ";
if ($2 == "schwarz") $$ = "0 0 0 setrgbcolor ";
if ($2 == "weiss")   $$ = "255 255 255 setrgbcolor ";
```

Anweisung → RE Zahl:

```
$$ = $2 + " turn ";
```

Anweisung → STIFT Zahl:

```
$$ = $2+" setlinewidth ";
```

Anweisung → VW Zahl:

```
$$ = $2+" draw ";
```

Der ZR→PS-Compiler ist nun vollständig beschrieben. Globaler Code ist für diese semantischen Regeln nicht erforderlich.

Zur Vollendung der ZR→PNG-Compilation ist ein PS→PNG-Compiler erforderlich. Dieser wird von FLACI bereitgestellt und kann in dem entsprechenden Dialog ausgewählt werden.

Die erzeugte PNG Datei kann aus dem T-Diagramm heruntergeladen und mit einer Bildbearbeitung betrachtet werden. Da dies mehrere Schritte erfordert, ist es praktischer einen PNG-Interpreter direkt im Browser und somit in FLACI zu nutzen. Zur Interpretation der erzeugten PNG-Datei kann wiederum ein in FLACI vorhandener PNG-Interpreter verwendet werden.

Alternativ zeigt Abbildung 9.18 ein Übersetzungsmodell für einen „selbstgebauten“ PNG-Interpreter.

Die Funktion `btoa()` steht für „binary to ascii“ und codiert eine (binäre) Zeichenkette nach Base64. In der Beispielsammlung befindet sich das vollständige Übersetzungsmodell.

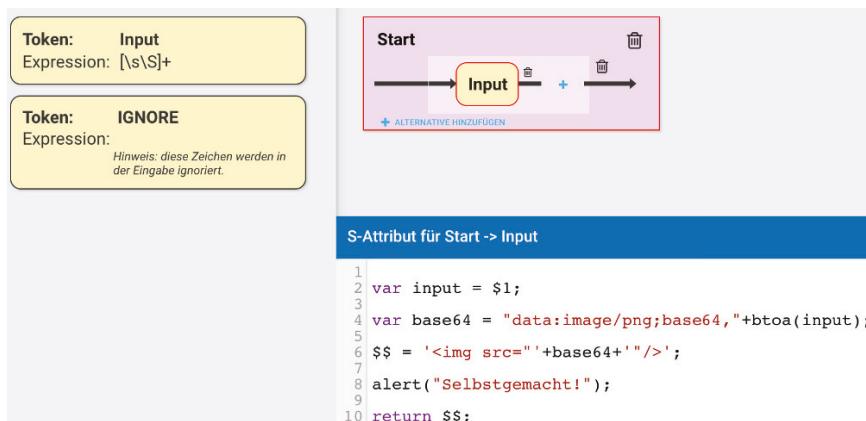


Abbildung 9.18: ZR→PS→PNG-Compilation

Abbildung 9.19 zeigt ein kleines Anwendungsbeispiel (Rosette).

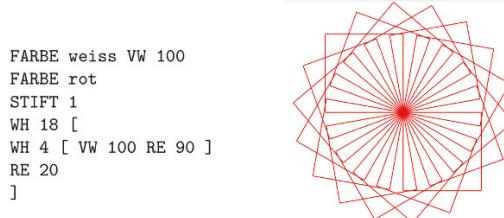


Abbildung 9.19: Rosette: als Wort in ZR (links) und als interpretiertes PDF (rechts)

Computerübung 9.19



Verwenden Sie den Compilergenerator zur Erzeugung des beschriebenen $ZR \rightarrow PS$ -Compilers. Vervollständigen Sie das entsprechende T-Diagramm und starten Sie die Übersetzung eines ZR -Programms mit nachfolgender PNG-Interpretation.

Beachten Sie die Ausgaben der beteiligten Compiler und wählen Sie am Ende die HTML-Anzeige, um tatsächlich eine „Blume“ zu erhalten.

Computerübung 9.20



Für das Zeichnen komplexer Figuren ist ein Sprungbefehl von Vorteil, um mehrfaches „Zurückfahren“ einzusparen. Wir haben bereits im PostScript-Kopf einen passenden Befehl definiert und verwendet: 300 400 goto. Erweitern Sie den $ZR \rightarrow PS$ -Compiler so, dass er einen neuen Befehl der Form GOTO Zahl, Zahl „versteht“ und entsprechend verarbeitet. Hierfür benötigen Sie zwei neue Tokenklassen, nämlich für GOTO und das Komma, und eine neue rechte Regelseite für Anweisung.

Computerübung 9.21



Bereits ausgewählte Teile von PostScript bieten eine Vielzahl an einfachen Befehlen, um unseren Zeichenroboter mit weiteren Funktionen auszurüsten. Experimentieren Sie unter Zuhilfenahme entsprechender Einführungstexte und Online-Tutorials zu PostScript.

Computerübung 9.22



Entwerfen Sie einen „Farbübersetzer“, der einen Fließtext nimmt, und alle Vorkommen der Wörter blau, rot, grün, gelb, schwarz und weiß durch einen Tag der Form `rot` ersetzt. Alle anderen Zeichen sollen wieder direkt ausgegeben werden. Hierfür ist eine entsprechende Tokenklasse nötig. Zur Beschreibung eines beliebigen ASCII-Zeichen (inklusive Zeilenumbrüchen und Sonderzeichen) eignet sich der reguläre Ausdruck `[^a] | a`.

9.4.2 ZR→PDF-Compiler

jsPDF

Unter Verwendung von *jsPDF* wollen wir nun einen Compiler entwickeln, der ZR -Programme *direkt* in PDF übersetzt. *jsPDF* ist eine JavaScript-Bibliothek, mit der man PDFs mit JavaScript generieren kann, s. Abbildung 9.20.

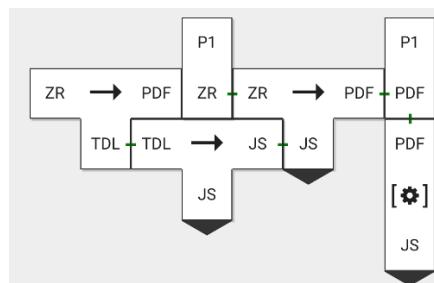


Abbildung 9.20: Entwicklung und Anwendung eines ZR→PDF-Compilers

Als erstes verweisen wir auf die Grammatikdefinition der Quellsprache ZR, s. Abbildung 5.17, und konzentrieren uns auf die reduzierte Grammatik gemäß Abbildung 5.18 auf Seite 136.

Wieder verwenden wir die TDL, um die Übersetzung aus ZR in PDF zu beschreiben. Für den vom Scannergenerator zu erzeugenden Scanners werden die in Abbildung 9.16 dargestellten Token verwendet. Die Idee ist nun, mit `var doc = new jsPDF("p", "pt", "A4");` ein jsPDF Dokument zu erstellen und darin gemäß ZR-Programm eine Folge von JavaScript-Zeichenbefehlen zusammenzustellen. Dieser Befehlsstapel wird am Ende evaluiert, wodurch ein zugehöriges PDF entsteht.

Zum Verständnis der semantischen Regeln in Abbildung 9.21 ist die Kenntnis des Koordinatenursprungs in der *linken oberen Ecke* erforderlich.

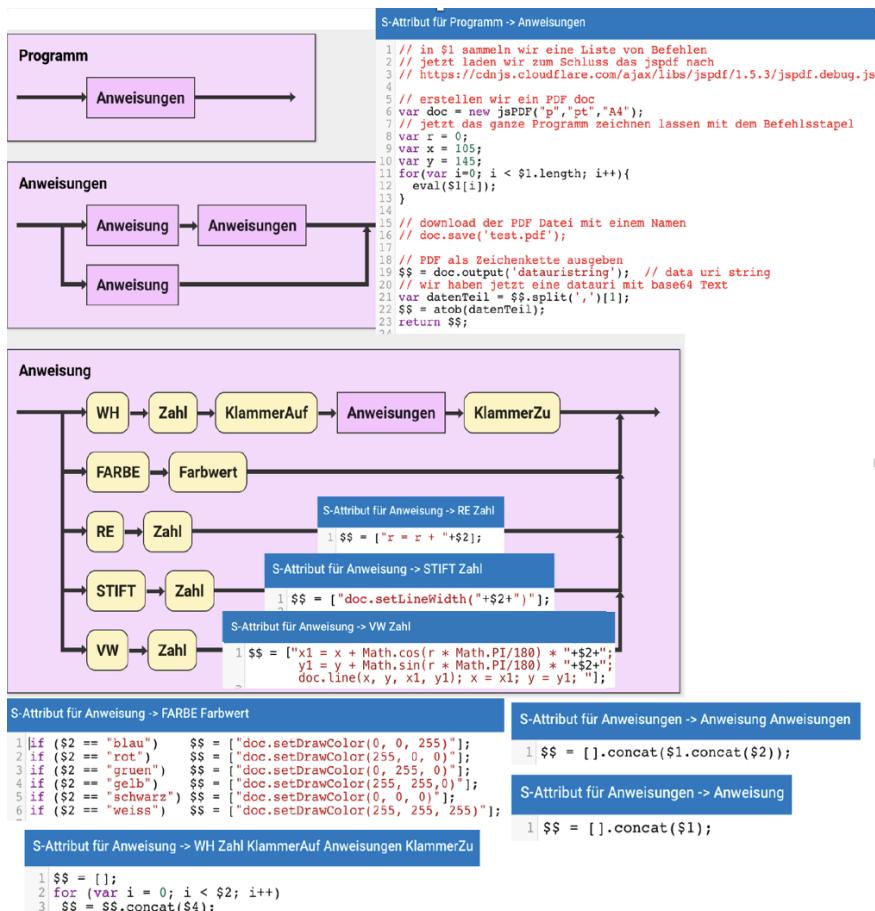


Abbildung 9.21: Semantische Regeln für den *ZR → PDF*-Compilergenerator

Für die Übersetzung von beispielsweise VW zahl1 (Quellsprache ZR) in einen Linienzug doc.line(x,y,x1,y1) sind die Koordinaten (Zielsprache PDF) wie in Abbildung 9.22 zu berechnen.

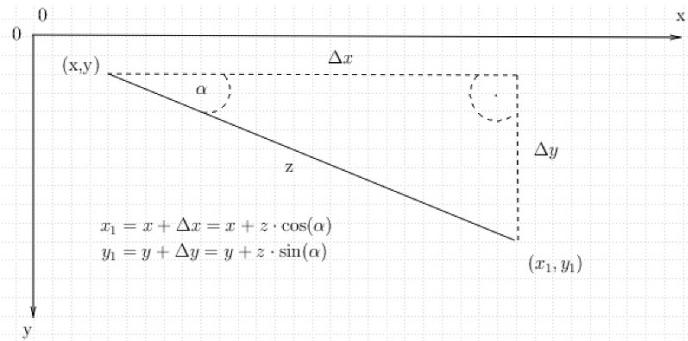


Abbildung 9.22: Berechnung der Koordinaten geometrischer Operationen

Abbildung 9.22 illustriert, dass sich die Turtle von Position (x, y) z Schritte in aktueller Blickrichtung bewegt und danach auf den Punkt (x_1, y_1) gelangt. Mathematisch gesehen, haben wir es also mit einer Transformation einer koordinatenfreien Geometrie (ZR) in eine mit kartesischen Koordinaten (PDF) zu tun. Damit wir das tun können, müssen wir x_1 und y_1 immer aus der aktuellen, d.h. jeweils letzten Position und dem Richtungswinkel über trigonometrische Funktionen bestimmen.

Übung 9.3



Informieren Sie sich über die JS-Kommandos zur Erzeugung einer dementsprechenden PDF (Zielsprache) unter http://www.rotisedapsales.com/snr/cloud_staging/website/jsPDF-master/docs/index.html.

Übung 9.4



Entscheiden Sie selbst, wie stark Sie die Lösung Zeichenroboter: ZR->PDF aus der FLACI-Beispieldatenbank (unter Zeichenroboter) einbeziehen.

Interessant ist noch die Wiederholungsanweisung von ZR. Die syntaktische Regel lautet: Anweisung -> WH Zahl KlammerAuf Anweisungen KlammerZu. Sie wird verwendet, um die in eckigen Klammern eingeschlossene Anweisungsfolge n -mal auszuführen ($n > 0$). WH 4 [VW 100 RE 90] lässt die Turtle ein Quadrat der Seitenlänge 100 zeichnen.

Anstelle dieser geklammerten Zeichenbefehle direkt auszuführen, schreiben wir sie in eine Liste und zwar so oft, wie es n angibt. Erst am Schluss werden sie nacheinander abgearbeitet, um die Ausgabegrafik (PDF) zu erzeugen.

```
$$ = [];
for (var i = 0; i < $2; i++)
  $$ = $$ . concat($4);
```

Der Compiler-Baustein aus Abbildung 9.20 (links) ist mittels TDL in Abbildung 9.21 komplett beschrieben; der Compiler kann generiert werden.

Nach Ausführung der $ZR \rightarrow PDF$ -Übersetzung steht das PDF-Dokument im entsprechenden Ausgabebaustein und kann bei Bedarf herunter geladen werden. Innerhalb von FLACI kann das PDF-Dokument mit einem bereitgestellten oder einem selbst hergestellten (s. Abbildung 9.23) PDF-Interpreter interpretiert werden.

Dabei machen wir uns die vielfältigen Möglichkeiten moderner Webbroweser zu nutze: Man kann nämlich eine PDF-Datei bei entsprechender Zeichenkodierung in ein iFrame einbetten und das so entstandene HTML vom Standardbrowser interpretieren lassen. Der Schalter in der Ausgabeprotokollierung ist auf HTML zu stellen.

Start → Alles:

```
var input = $1;
var base64 = "data:application/pdf;base64,"+btoa(input);
$$ = '<iframe src="'+base64
'" style="width:100%;height:100%;min-height:300px;border:0">'
+ '</iframe>';
return $$;
```



Abbildung 9.23: Beschreibung eines *PDF*-Interpreters

Computerübung 9.23

Entwerfen Sie das Übersetzungsmodell als T-Diagramm und führen Sie die entsprechende Compilergenerierung durch.



9.5 Präsentation eines Compilers

9.5.1 Anwendung und Dokumentation

Neben der Erzeugung eines Compilers ist auch seine spätere Verwendung und Dokumentation in der Praxis ein wichtiges Thema. Oftmals werden Compiler auf der Kommandozeile zum Einsatz gebracht und mit vielen Parametern gesteuert.

Die Verwendung von JavaScript als Ausführungssprache ermöglicht aber eine viel-

seitigere Nutzung. So lassen sich die mit FLACI erzeugten Compiler bzw. Interpreter auch unmittelbar in Web-Applikationen bzw. beliebigen Webseiten integrieren und anwenden. Beispielsweise kann mit einem FEN→SVG-Compiler (s. Abschnitt 9.3.4) in einer Webseite zum Thema Schach eine graphische Darstellung von mit FEN beschriebenen Brettstellungen eingeblendet werden.

Möchte man keine eigene Webseite betreiben, können alternativ auch Web-2.0-Dienste genutzt werden, die eine einfache Erstellung von Webseiteninhalten ermöglichen. Ein Beispiel dafür sind Wikis.

Unter <https://programmingwiki.de> steht ein Wiki für informatikdidaktische Inhalte bereit, welches bereits einige Komfortfunktionen für die unmittelbare Nutzung von FLACI-Compilern bereitstellt. Im Unterschied zu gewöhnlichen Wikis, bietet das *ProgrammingWiki* (PW) die Möglichkeit, Quellcode verschiedener Programmiersprachen direkt im Browser zu editieren, auszuführen und das Ergebnis der Abarbeitung zu betrachten. Damit können interaktive Arbeitsblätter zu diversen Themen erstellt werden, ähnlich wie mit Jupyter Notebooks.

Unser ProgrammingWiki (<https://programmingwiki.de/Startseite>) bietet zusätzlich die Möglichkeit, *editierbaren* Programmcode genau einer Programmiersprache in den Text einzufügen und diesen per Knopfdruck zu *evaluieren*. Dies ist nicht nur für eine beachtliche Liste von Programmiersprachen (Python, JavaScript, Logo, Scheme, Prolog usw.) möglich, sondern sogar für Compiler, die mit FLACI entwickelt wurden.

ProgrammWiki

FLACI+PW



Didaktischer Hinweis 9.9

Der Einsatz des ProgrammingWikis reicht damit von vorbereiteten Einführungskursen in bestimmte Programmiersprachen und *interaktiven Seminaren* zu Themen, wie etwa „Algorithmen und Komplexität“, bis hin zur Dokumentation von Übersetzungsprozessen.

In unserem Beispiel zur FEN-Schachnotation machen wir davon im Rahmen einer *Projektpräsentation* Gebrauch. Es wird empfohlen, sich am ProgrammingsWiki anzumelden. Exemplarisch legen wir die Seite „Schach“ an oder modifizieren die existierende. Zum Schreiben der Seite wechseln wir in den Bearbeiten-Modus. Wählt man als Programmiersprache FLACI aus, fragt das ProgrammingWiki nach der FLACI-URL des gewünschten Compilers. Im Falle des FEN→SVG-Compilers ist der URL aus dem zugehörigen T-Diagramm in FLACI leicht zu beschaffen: <https://flaci.com/c2716g54bqdgrgr.js>.

In die geöffnete ProgrammingWiki-Seite schreiben wir an der vorgesehenen Stelle

```
<run id="..."><!-- ID wird automatisch vergeben -->
    rnbqkbnr/ppp2ppp/4p3/3p4/3PP3/8/PPP2PPP/RNBQKBNR
</run>
<rendersvg></rendersvg>
```

und speichern die Seite. Im Browser wird dieser Teil der Webseite anschließend

wie in Abbildung 9.24 dargestellt. Werden die beiden Knöpfe nacheinander ge-



Abbildung 9.24: Verwendung des FEN→SVG-Compiler im ProgrammingWiki

drückt, verwandelt sich die Darstellung zu Abbildung 9.25.

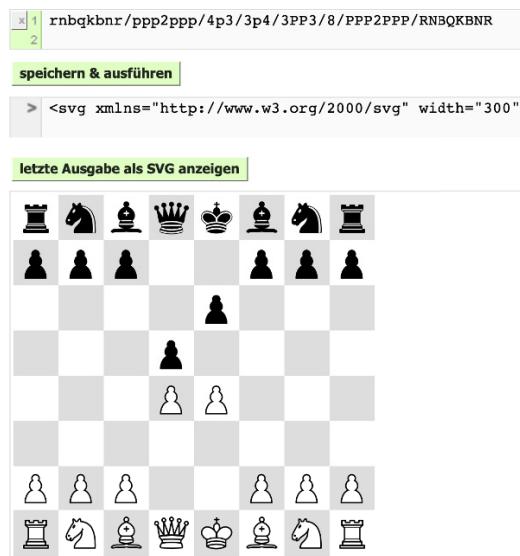


Abbildung 9.25: Verwendung des FEN→SVG-Compiler im ProgrammingWiki

Der im Beispiel angegebene FEN-Code ist editierbar, sodass die Webseite für das zweite Beispiel (Matt-Stellung) unverändert übernommen werden kann.

Computerübung 9.24

Schauen Sie sich das ProgrammingWiki <https://programmingwiki.de/Schach> für unser Projektbeispiel an und experimentieren Sie damit.

9.5.2 Präsentation und Diskussion von Projektergebnissen

Das Dreamteam der aktuellen Web-Programmierung ist HTML, JavaScript und CSS. Web-basierte Entwicklungsumgebungen, wie JS Bin, CodePen, Thimble und *JSFiddle*, ermöglichen es uns, mit HTML, JavaScript und CSS zu experimentieren ohne dafür separate lokale Installationen vornehmen und deren Zusammenspiel organisieren zu müssen. Auch das Beschaffen und Integrieren spezieller Bibliotheken bleibt uns erspart, wenn wir beispielsweise mit JSFiddle, dem „Urvater“ Browser-basierter Entwicklungsumgebungen arbeiten.

In JSFiddle kann man HTML-, JavaScript- und CSS-Code editieren und sich das Evaluationsergebnis unmittelbar danach anzeigen lassen. Durch die Möglichkeit, die eigenen Entwürfe, Probleme und Fehler anderen Entwicklern via JSFiddle „vorzutragen“, kann man Defizite rasch beheben und einen guten Arbeitsfortschritt bewirken. Diese Form der Projektdokumentation richtet sich vor allem an Studierende und Software-Entwickler.

Abbildung 9.26 zeigt, wie der mit FLACI entwickelte FEN→SVG-Compiler in JSFiddle eingebaut wird.



The screenshot shows a JSFiddle interface. On the left, the HTML tab displays the following code:

```

HTML ▾


<div id="Stellung1"></div>
<div>Französische Verteidigung</div>
</div>


<div id="Stellung2"></div>
<div>Schwarz setzt Weiß matt.</div>
</div>
</div>


```

Below the HTML tab is the JavaScript + No-Library (pure JS) tab, which contains the following code:

```

// FEN zu SVG Compiler laden
$.getScript("https://flaci.com/c2688gpkdghdr.js", function() {
    var p = generateParser();
    var svg = p.parse("rnbqkbnr/ppp2ppp/4p3/3p4/3PP3/8/PPP2PPP/RNBQKBNR");
    $('#Stellung1').append($(svg));
    var p = generateParser();
    var svg = p.parse("6r1/6pp/7r/1B5K/1P3k2/N7/3R4/8");
    $('#Stellung2').append($(svg));
});

```

Abbildung 9.26: Projektdiskussion in jsfiddle

9.6 Datenvisualisierung

Auch dieses Projektbeispiel verwendet (aktuelle) Daten, die zum Herunterladen via Webservices vorgehalten werden: https://earthquake.usgs.gov/earthquakes/feed/v1.0/summary/4.5_week.geojson liefert Erdbebenda-

ten im JSON-Format. In Abbildung 9.27 werden sie in dem Programmsteinein „Quake-JSON“ nach Angabe des genannten URL bereitgestellt.

Computerübung 9.25

Öffnen Sie das vorbereitete Projekt aus der FLACI-Beispielsammlung und verfolgen Sie diese Erklärungen an den einzelnen Bausteinen.

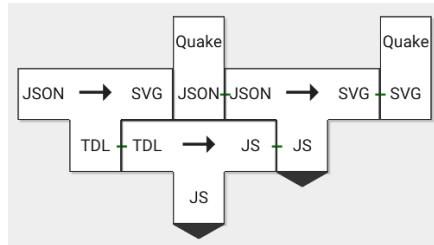


Abbildung 9.27: Grafische Darstellung von Erdbebengebieten

Um einen Informationsmehrwert zu erzielen, sollen die beschafften Daten mit einem (veränderten) Kontext in Verbindung gebracht werden. Im vorliegenden Fall werden die Erdbebendaten auf eine Weltkarte (https://upload.wikimedia.org/wikipedia/commons/4/41/Simple_world_map.svg), wie in Abbildung 9.28, projiziert.



Abbildung 9.28: Weltkarte

Die Karte (von Wikipedia) liegt im SVG-Format vor. Um die geographischen Koordinaten aufgezeichnete Erbebenen auf Pixelkoordinaten auf der SVG-Karte abzubilden, verwenden wir die D3 Bibliothek. Die Koordinaten aller Punkte werden der beschafften Datei entnommen und in einem globalen Feld gespeichert.

Der Compiler in Abbildung 9.27 liest JSON und sucht darin die Koordinaten der Erdbebenorte. Diese werden als ausgefüllte rote Kreise ins SVG gezeichnet. Ausgegeben wird ein SVG-Bild. Stellt man die Ausgabe auf HTML, ergibt sich die in Abbildung 9.29 angegebene Darstellung.

Den JSON-Compiler findet man in der FLACI-Beispielsammlung. Er ist ein guter Ausgangspunkt für Projekte, die Daten im JSON-Format verarbeiten.

Verwende Compilerdefinition (JSON → SVG, TDL) als Eingabe.
Der Compiler (TDL → JS, JS) wird ausgeführt.
Ausgabe: JSON → SVG Compiler ist JS
Schreibe Ausgabe in Compiler-Baustein (JSON → SVG, JS).
Verwende Programm (Quake, JSON) als Eingabe.
Der Compiler (JSON → SVG, JS) wird ausgeführt.
Ausgabe:



Schreibe Ausgabe in Programm-Baustein (Quake, SVG).

Abbildung 9.29: Erdbebengebiete

Mit JSFiddle wird wieder eine geeignete Projektdokumentation gegeben:
<https://jsfiddle.net/kj5d4szn/1/>.



Didaktischer Hinweis 9.10

Dieses Projekt eignet sich zur Binnendifferenzierung als auch zur Anregung für Projekt mit „bequemer“ Datenbeschaffung und geeigneter Visualisierung.



FLACI

Computerübung 9.26

Modifizieren Sie das Erdbeben-Beispiel zu einer PLZ-in-SVG-Compilation. Dabei sollen Postleitzahlen (ZIP-code) von Orten auf der ganzen Welt auf der Weltkarte markiert werden. Der erforderliche Webservice ist <https://api.zippopotam.us/>. Eine Lösung der Aufgabe findet sich in der FLACI-Beispieldatensammlung.

Die in den vorangehenden Abschnitten betrachteten Miniprojekte wurden so gewählt, dass sie nur geringe Programmierkenntnisse erfordern. Trotzdem liefern sie einen überzeugenden Effekt und lassen (zurecht) vermuten, dass es zahlreiche Probleme gibt, die sich auf Sprachen und deren Übersetzung zurückführen lassen. Den verschiedenen Probleminstanzen entsprechen verschiedene Eingabewörter.

Außerdem gewinnen die Inhalte der Theorie formaler Sprachen und abstrakter Automaten an Attraktivität, wenn man sie in Kontexte des Compilerbaus einbettet.



10 TURING-Maschine (TM) und CHOMSKY-Typ-0/1-Sprachen

10.1 Grenzen von Kellerautomaten

In den vorangehenden Kapiteln haben wir vor allem reguläre und kontextfreie Sprachen betrachtet. Ihre besondere Bedeutung erwächst aus deren praktischen Anwendungen.

Dennoch sind CHOMSKY-Typ-0- und -1-Sprachen von theoretischem Interesse: Im Satz 2.4 haben wir festgestellt, dass das Wortproblem für diese Typ-0-Sprachen nicht allgemein entscheidbar ist, da die wichtige Eigenschaft der Längenmonotonie fehlt.

Nicht nur aus theoretischer Sicht ist es wichtig zu wissen, ob es zur Definition *jeder* formalen Sprache einen Kellerautomaten (NKA) gibt oder nicht. Vielleicht sind die passenden NKA für gewisse Sprachen nur sehr schwer zu finden? Andernfalls müssten wir nach einem anderen Automatentyp suchen, der gerade die Sprachklassen beschreibt, die außerhalb der kontextfreien liegen.

Der Chomsky-Hierarchisierung formaler Grammatiken in Abschnitt 2.5 entnehmen wir, dass NKA für kontextsensitive Sprachen (ksS) und CHOMSKY-Typ-0-Sprachen nicht ausreichen. Beispielsweise gibt es keinen NKA, der die Sprache

$$L_1 = \{a^n b^n c^n \mid n \geq 0\},$$

akzeptiert, s. Beispiel 6.18 auf Seite 172.

Man kann nun versuchen, neue Automatenmodelle zu kreieren, die gerade die Typ-1- und Typ-0-Sprachen erkennen. Einfach ist diese Aufgabe nicht. Vielleicht entwerfen Sie einen erweiterten NKA-Typ, der mit *zwei* oder allgemein mit k Stapeln arbeitet. Auch die Arbeitsweise eines solchen Multistack-Automaten lässt sich leicht verallgemeinern: Der Automat M befindet sich zunächst in einem Zustand q , liest ggf.(!) das aktuelle Eingabezeichen a , ersetzt jedes der k obersten Kellerzeichen X_i durch α_i , mit $i = 1, 2, \dots, k$, und wechselt in den Zustand p . Die Überführungsfunktion δ fasst dies zusammen:

$$\delta(q, a, X_1, X_2, \dots, X_k) = (p, \alpha_1, \alpha_2, \dots, \alpha_k).$$

Stoppt der Multistack-Automat in einem Endzustand, wird das vorgelegte Wort akzeptiert.

kontextsensitive
Sprachen

Multistack-
Automat

Zur Analyse von $L = \{a^n b^n c^n \mid n \geq 0\}$ mit einem solchen Multistack-Automaten könnte man für a , b und c jeweils einen Stack vorsehen. Nach einem „Durchlauf“ zum Wortende wird eine „Multipop“-Operation auf den drei Stapeln ausgeführt. Sind die Stapel mit der letzten Anwendung allesamt leer, wird das Wort akzeptiert.

Diese Strategie ist zwar für L_1 erfolgreich, für

$$L_2 = \{ss \mid s \in \{a, b\}^*\}$$

Warteschlangen-Automat
jedoch ungeeignet. Anstelle der Stapel wäre hier vermutlich eine *Warteschlange* (queue) angebracht. Ein *Warteschlangen-Automat* wäre für die erste Sprache jedoch nicht zuträglich.

Es ist also alles andere als einfach, ein allgemeingültiges Maschinenmodell zu entwickeln, das exakt die gewünschten Sprachklassen (CHOMSKY-Typ-0/1-Sprachen) abdeckt.

10.2 Die TURING-Maschine (TM)

Alan TURING Alan Mathison TURING¹ ersann 1936 ein Automatenmodell, indem er statt des Kellers nur einen sequentiellen Speicher (wie bei EA) vorsah. Dieses Eingabeband ist (in der bekannten Weise) in unendlich viele aufeinanderfolgende Felder gegliedert. Jeder Feldinhalt kann gelesen oder überschrieben werden. Zu der aus anderen Modellen bekannten Lesefunktion des Kopfes tritt eine Schreibfunktion hinzu. Jedes Feld des Eingabebandes enthält genau ein Zeichen.

Lese/Schreib-Kopf Der Lese/Schreib-Kopf darf entweder um genau ein Feld nach links (L) oder nach rechts (R) bewegt werden. Alternativ darf er im entsprechenden Arbeitstakt auch seine aktuelle Position beibehalten, d.h. es findet keine Kopfbewegung statt (N), s. Abbildung 10.1.

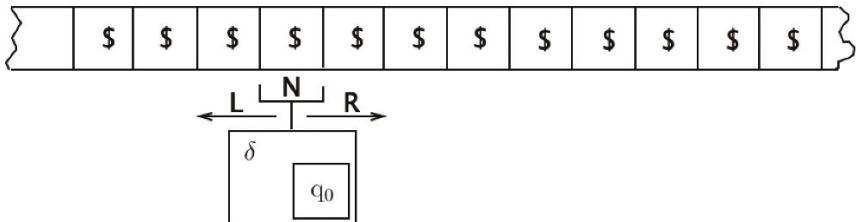


Abbildung 10.1: Aufbau einer TM

¹ Alan M.TURING: geb. 23.06.1912 in London, gest. 07.06.1954 in Wilmslow

Schon während seiner Zeit bei Alonzo CHURCH (1903-1995) an der Princeton University (USA), wo er 1938 promovierte, erarbeitete TURING *theoretische Grundlagen* für automatische Rechenmaschinen. Die Ergebnisse dieser Überlegungen führten zu herausragenden Beiträgen zur *Berechenbarkeitstheorie* (theory of computation; computability). Damit war er (zusammen mit CHURCH) einer der ersten, die sich mit den Möglichkeiten bzw. absoluten Grenzen maschinellen Rechnens auseinandersetzen. Dies geschah zu einer Zeit, nämlich Mitte/Ende der 19-Hundert-dreißiger Jahre, als es noch keine Digitalrechner (Computer) als physischen Gegenstand gab².

Bei seinen Überlegungen ließ sich TURING von dem Vorhaben leiten, den geistigen Prozess des Rechnens mit Papier und Bleistift und dessen Leistungsfähigkeit („computing power“) abstrakt zu beschreiben. Heute finden Bleistiftrechnungen zwar kaum noch statt, aber man darf ersatzweise getrost an die Arbeit von Computerprogrammen denken.

Das Papier oder eben den Speicher zur Aufnahme der zu verarbeitenden Daten abstrahiert TURING zu einem (potentiell) unendlichen Band mit Zellen für jedes einzelne Zeichen.

Wenn ein Mensch eine Berechnung ausführt, so nimmt sein Denken bestimmte Zustände, sog. „states of mind“, ein. Die Abstraktion mündet in eine endliche Zustandsmenge.

Es wird angenommen, dass die Zustände nacheinander eingenommen werden und dass während des Berechnungsprozesses keine Interaktion des „menschlichen Rechners“ mit der Umwelt stattfindet³.

Das Schreiben und Wegradieren von Zeichen wird durch einen Lese/Schreib-Kopf abstrakt beschrieben.

Man kann sich leicht überlegen, dass die Ideen aus Abschnitt 10.1 von der Aufnahme zusätzlicher Stapel bzw. Warteschlangen in NKAs durch das TURING-Maschinenmodell mit erfasst werden. Hierfür ist es allerdings notwendig, die Arbeitsweise von TM zu kennen. Diese wird in Abschnitt 10.3 erläutert.

Da die ksS von geringem praktischen Interesse sind, konzentrieren wir unsere Be trachtung auf TM, von denen wir wissen (s. Satz 10.1 auf S. 256), dass sie für die CHOMSKY-Typ-0-Sprachen „zuständig“ sind. Außerdem reicht es aus, wenn wir uns nur um deterministische TM, kurz: DTM, kümmern. Wie bei EA stellt sich nämlich heraus, dass NTM und DTM in ihrer Leistungsfähigkeit als Akzeptoren gleichwertig sind.

Die folgende Definition präzisiert das Automatenmodell.

²Mit der Entwicklung des Z3 im Jahre 1941 gilt Konrad ZUSE (geb. 22.06.1910 in Berlin, Abitur in Hoyerswerda; gest. 18.12.1995 in Hünfeld bei Fulda) als Erfinder des Computers.

³Beide Annahmen sind anfechtbar und Gegenstand gewisser wissenschaftlicher Debatten darüber, ob das Modell der TURING-Maschine wirklich alle Aspekte beliebiger Berechnungsprozesse erfasst.

Berechenbarkeitstheorie

(potentiell) unendliches Band

endliche Zustandsmenge

Lese- und Schreibkopf



Definition 10.1

Eine TURING-Maschine (TM) ist ein 7-Tupel $M = (Q, \Sigma, \Gamma, \delta, q_0, \$, E)$.

- Q ... endliche Menge von Zuständen
- Σ ... Eingabealphabet
- Γ ... Bandalphabet, wobei $\Sigma \subseteq \Gamma \setminus \{\$\}$
- δ ... partielle Überführungsfunktion: $Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, N, R\}$. Für eine nicht-deterministische TM – kurz: NTM – gilt: $Q \times \Gamma \rightarrow \wp(Q \times \Gamma \times \{L, N, R\})$.
- q_0 ... Anfangszustand, $q_0 \in Q$
- $\$$... Bandvorbelegungszeichen, kurz: Blankzeichen, mit $\$ \in \Gamma \setminus \Sigma$
- E ... endliche Menge von Endzuständen, mit $E \subseteq Q$

10.3 Die Arbeitsweise einer DTM

Blankzeichen

Jedes einzelne Feld des potentiell unendlichen Band einer DTM enthält zunächst ein Blank- oder Vorbelegungszeichen. In Abbildung 10.2 ist das das Zeichen \$.

Eingabealphabet

Vorbereitend wird das Eingabewort w , das ausschließlich aus Zeichen des Eingabealphabets Σ besteht, auf das Band geschrieben. Die Vorbelegungszeichen in den entsprechenden Feldern werden dadurch verdrängt. Der Lese/Schreib-Kopf befindet sich über dem Feld mit dem ersten Zeichen von w und die Maschine nimmt den Anfangszustand q_0 ein. Wir nennen dies eine *Anfangskonfiguration* und schreiben dafür $q_0 w$.

Anfangskonfiguration

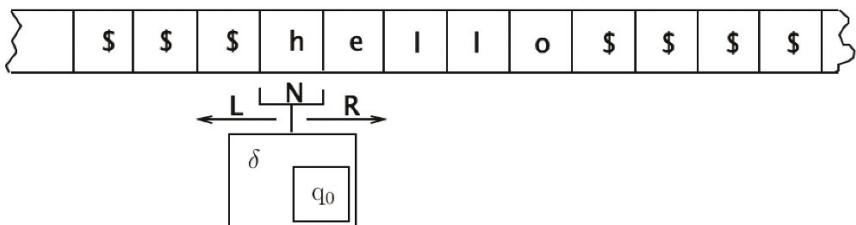


Abbildung 10.2: TM mit Eingabewort `he$lo`

Es ist zu beachten, dass die Initialisierung der betrachteten DTM mit einem Eingabewort von außen (durch den agierenden Menschen) geschieht. Damit handelt es sich um einen externen Vorgang, der nicht zur Arbeitsweise der DTM gehört.

- partielle Überführungs-funktion
- Die *partielle* Überführungsfunktion δ
- $$\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, N, R\}$$

ist das Herzstück einer DTM. Mit ihr wird die Aktion der Maschine in jedem Arbeitstakt festgelegt. Diese Festlegung geschieht in Abhängigkeit

- vom Zustand q_i ($q_i \in Q$), den die DTM gerade eingenommen hat,
- und vom Zeichen k ($k \in \Gamma$), das sich in dem Feld der aktuellen Lese/Schreib-Kopf-Position befindet.

$\delta(q_i, k) = (q_j, m, b)$, mit $q_i, q_j \in Q$, $k, m \in \Gamma$ und $b \in \{L, N, R\}$, falls der Funktionswert (q_j, m, b) überhaupt existiert, bestimmt die nächste Aktion der Maschine:

1. Die DTM stoppt, wenn $\delta(q_i, k)$ nicht definiert ist. Man sagt auch: „Die Maschine stoppt per crash.“
2. Die DTM geht aus q_i in einen (vom aktuellen nicht unbedingt verschiedenen) Zustand q_j über, schreibt das entsprechende Zeichen m in das aktuelle Feld und bewegt *anschließend* ggf. den Kopf um eine Position nach links bzw. rechts, gemäß $b \in \{L, N, R\}$.

Arbeitstakt

Wird eine DTM in der oben beschriebenen Anfangskonfiguration für ein bestimmtes Eingabewort gestartet, so gibt es zwei Möglichkeiten:

1. Die DTM stoppt nach endlich vielen Schritten (per crash).
2. Die DTM läuft endlos weiter.

Für den zweiten Fall gibt es drei Realisierungsmöglichkeiten:

- (a) unendlicher Rechtslauf,
- (b) unendlicher Linkslauf,
- (c) Endloszyklus, wobei die zum Zyklus gehörende „Taktlänge“ keine Rolle spielt.

Beispiel 10.1

Wir betrachten die folgende sehr einfache DTM

$M_1 = (\{q_0, q_1, q_2\}, \{a, b, c\}, \{\$\}, a, b, c, x, \delta, q_0, \$, \{q_2\})$, mit



| δ | $\$$ | a | b | c | x |
|----------|----------------|---------------|-----|---------------|-----|
| q_0 | - | (q_1, x, R) | - | - | - |
| q_1 | $(q_1, \$, N)$ | (q_1, x, R) | - | (q_2, x, L) | - |
| q_2 | - | - | - | - | - |

Enthält ein Tabellenfeld einen Strich, so gibt es keinen Funktionswert. Dies ist für partielle Funktionen erlaubt und bedeutet crash für M_1 , wenn δ ein solches Feld erreicht.

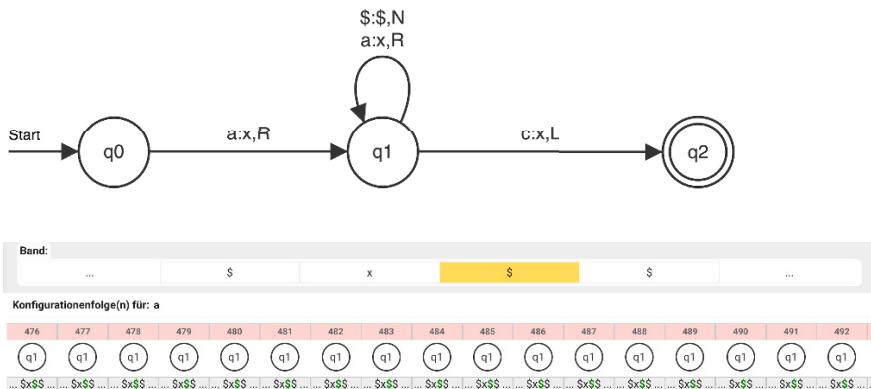
Die Überführungsfunktion kann auch als Graph dargestellt werden:

Für das Eingabewort aac ergibt sich folgende Verarbeitung:

$q_0 aac \vdash x q_1 ac \vdash x x q_1 c \vdash x x q_2 xx$.

Wählt man a als Eingabewort, stoppt M_1 wegen $\delta(q_1, \$) = (q_1, \$, N)$ nicht:

Welche Wörter von M_1 akzeptiert werden, können wir erst in Übungsaufgabe 10.2 nach



Behandlung von Abschnitt 10.4 klären. Hier geht es zunächst nur um die Betrachtung der Arbeitstakte des Automaten.

Konfiguration Zur Protokollierung dieser Arbeitstaktsequenz haben wir die zugehörige Folge der Konfigurationen $\alpha q_i \beta$, mit $\alpha \in \Gamma^*$, $q_i \in Q$ und $\beta \in \Gamma^+$, beginnend mit der Anfangskonfiguration $q_0 w$ notiert. α bzw. β bestimmt den Bandinhalt links bzw. rechts von der Kopfposition, wobei der Feldinhalt der aktuellen Kopfposition das erste Zeichen von β bildet.



Definition 10.2

Eine **Konfiguration** einer TM ist ein Wort aus $\Gamma^* \circ Q \circ \Gamma^+$.

Ein **Konfigurationsübergang** $\vdash \subseteq (\Gamma^* \circ Q \circ \Gamma^+) \times (\Gamma^* \circ Q \circ \Gamma^+)$ ist eine binäre Relation mit der oben kommentierten Definition.

Eine Folge von Konfigurationsübergängen $k_1 \vdash k_2 \vdash \dots \vdash k_n$ kann verkürzt durch $k_1 \stackrel{*}{\vdash} k_n$ angegeben werden.



Übung 10.1

Geben Sie nacheinander die Folgen der Konfigurationsübergänge für die Eingabewörter $aacaaba$, $abaaca$ und aaa in der DTM aus Beispiel 10.1 an. Kommentieren Sie die Befunde.



Übung 10.2

Begründen Sie, weshalb die DTM aus Beispiel 10.1 genau alle Wörter akzeptiert, die mit dem regulären Ausdruck $a+c[abc]^*$ beschrieben werden können. Das kürzeste akzeptierte Wort ist also ac .



Übung 10.3

Überlegen Sie, warum eine DTM prinzipiell mit einem einzigen Endzustand q_e auskommt. Welcher Vorteil ergibt sich ggf., wenn man mehrere Endzustände zulässt?

10.4 Die DTM als Akzeptor

Analog zu den bereits behandelten Automatenmodellen dienen auch DTM als Akzeptoren formaler Sprachen: Ein Wort w aus Σ^* wird akzeptiert, wenn die auf w angesetzte DTM in endlich vielen Schritten *in einem Endzustand* stoppt. Dabei ist es völlig gleichgültig, was auf dem Band steht, wo sich der Lese-/Schreibkopf befindet und ob das Wort überhaupt ganz oder teilweise gelesen wurde.

Aus der Lösung von Übungsaufgabe 10.1 geht hervor, dass es DTM gibt, die lediglich das erste Zeichen eines Eingabewortes lesen, und das (gesamte) Wort akzeptieren. Dies steht in strengem Gegensatz zu endlichen und Kellerautomaten, die das Eingabewort stets vollständig abtasten. Hält die Maschine nicht oder in einem Nicht-Endzustand, so wird das Eingabewort nicht akzeptiert.

Unter Verwendung der oben eingeführten Begriffe Konfiguration und Konfigurationsübergang (s. Definition 10.2) kann die von einer DTM akzeptierte Sprache recht kompakt definiert werden.

Definition 10.3

Sei $M = (Q, \Sigma, \Gamma, \delta, q_0, \$, E)$ eine DTM. Dann heißt $L(M)$ die von M akzeptierte Sprache, mit $L(M) = \{w \in \Sigma^* \mid q_0 w \xrightarrow{*} \alpha q_e \beta, \text{ mit } \alpha, \beta \in \Gamma^* \text{ und } q_e \in E\}$.

$w \in L(M)$

$w \notin L(M)$

Beispiel 10.2

Gesucht ist eine TM M_2 für die Sprache $L(M_2) = \{a^n b^n c^n \mid n \geq 0\}$.

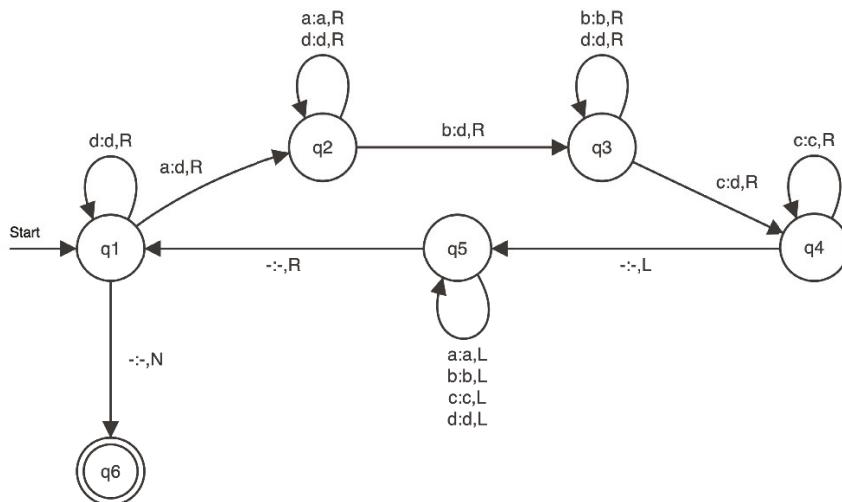
Lösungsidee: Auf der Basis der folgenden Überlegung kann eine vollständige Definition einer DTM M_2 für $L(M_2) = \{a^n b^n c^n \mid n \geq 0\}$ wie folgt angegeben werden:

$M_2 = (\{q_1, q_6, q_2, q_3, q_4, q_5\}, \{a, b, c\}, \{-, a, b, c, d\}, \delta, q_1, -, \{q_6\})$ mit



Beispiel

$L = \{a^n b^n c^n \mid n \geq 0\}$



Bei jedem Rechtslauf des Kopfes – beginnend mit dem ersten Zeichen des Eingabewortes – wird das jeweils zuerst gefundene a durch d , das jeweils zuerst gefundene b durch d und das jeweils zuerst gefundene c durch d ersetzt. Anschließend fährt der Kopf über eventuell vorhandene c 's bis an's Wortende, um dabei sicherzustellen, dass keine von c verschiedenen Zeichen folgen. Danach wird der Kopf auf das am weitesten links stehende d zurückgeführt. (Die linke Begrenzung erkennt der Kopf durch Erreichen des Bandvorbelegungszeichens, so dass eine unbegrenzte Linksbewegung ausgeschlossen ist.) Stehen am Ende (abgesehen den Vorbelegungszeichen) nur noch d 's auf dem Band, wird das Eingabewort akzeptiert.



Computerübung 10.1

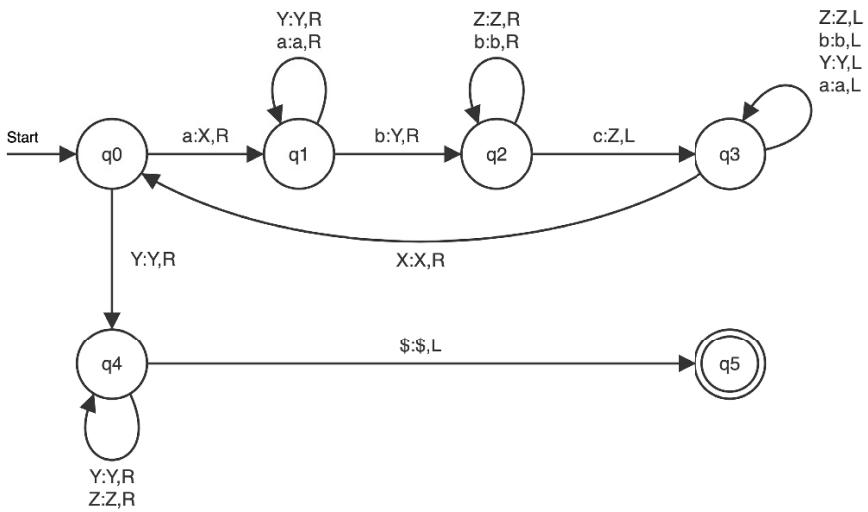
Simulieren Sie M_2 für mehrere Wörter über $\{a, b, c\}$, die zu $L(M_2) = \{a^n b^n c^n \mid n \geq 0\}$ gehören, und auch für solche, die nicht zu $L(M_2)$ gehören.



Computerübung 10.2

Betrachten Sie $M_3 = (\{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}, \{a, b, c\}, \{\$\$, a, b, c, X, Y, Z\}, \delta, q_0, \$, \{q_6\})$, mit der unten angegebenen Überführungsfunktion als alternative DTM für $\{a^n b^n c^n \mid n \geq 1\}$, wobei $L(M_2) = L(M_3) \cup \{\epsilon\}$.

Beschäftigen Sie sich mit M_3 und ergründen Sie die zugrunde liegende Konstruktionsidee.



Nun kommen wir noch einmal zurück auf die auf Seite 247 angedeutete Multistack-Maschine. In der Tat kann man zeigen, dass jede von einer TM akzeptierte Sprache auch von einem 2-Stapel-Automat akzeptiert wird. Der Beweis simuliert die Arbeitsweise einer TM mittels 2-Stapel-Automat. Dabei wird der erste Stapel für den vom Kopf der TM aus links stehenden Bandinhalt (außer Blankzeichen) verwendet und der zweite Stapel für den rechten, inkl. der aktuellen Zelle.



Didaktischer Hinweis 10.1

Grundsätzlich war die Idee eines Multistack-Automaten also nicht schlecht. In der Literatur findet man weitere leistungsfähige Modelle, wie etwa „Counter machines“, auf deren Definition wir an dieser Stelle jedoch verzichten.

10.5 Alternative TM-Definitionen

Schaut man in die Fachliteratur, so finden sich zahlreiche alternative Definitionen für die TM. Man kann zeigen, dass sie (in Bezug auf die Beschreibung berechenbarer Probleme bzw. Akzeptanz von Sprachen und Sprachklassen) äquivalent sind. D.h., grundsätzlich ist es gleichgültig, welche Definition wir verwenden. Spezielle Vorlieben für die eine oder andere Definitionsvariante ergeben sich aus dem vorgeesehenen Verwendungszweck, etwa für Beweise bestimmter Sätze. Die geschickte Definitionsauswahl kann die Arbeit dabei enorm erleichtern.

Die folgende unvollständige Liste trägt kann diese Vielfalt lediglich andeuten.

- Reduziert man in Definition 10.1 die möglichen Kopfbewegungen auf links und rechts, also $\{L, R\}$, ergibt sich eine gleichwertige TM-Definition. Das Verweilen des Kopfes auf der alten Position kann durch schreibneutrales Zurückführen des Kopfes mit L bzw. R erreicht werden.
- $\delta : (Q \setminus E) \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$. Der Ausschluss von Endzuständen im ersten Argument von δ macht Sinn, wenn zusätzlich gefordert wird, dass die Maschine bei Erreichen eines Endzustandes grundsätzlich stoppt.
- Forderung: $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, N, R\}$ ist total. Hieraus folgt für den Stopps einer TM: $(q_e, k) \rightarrow (q_e, k, N)$, mit $q_e \in E$ und $k \in \Gamma$. Man spricht in diesem Falle von einem dynamischen Stopps.
- Es gibt mehrere Spuren auf einem Band, die mit einem oder mit mehreren separaten Köpfen verwaltet werden. Dies ist gleichbedeutend mit der Vorstellung von der Mehrband-TM.
- Das Band wird links beschränkt, so dass sich der Kopf nicht über das Feld des ersten Eingabewort-Zeichens zu Beginn der Arbeit hinausbewegen kann. Dies gilt aber nur für die Linksbewegung.

Wichtiger Hinweis: Diese Definitionsvariante darf nicht mit der linear beschränkten TM (LBTM oder LBA), deren Kopfbewegung nur innerhalb des durch das Eingabewort belegten Bandbereiches erlaubt ist, verwechselt werden. Die (nichtdeterministischen) LBTM beschreiben genau die Menge der CHOMSKY-Typ-1-Sprachen.

Alternative
äquivalente
TM-Defini-
tionen

Linear be-
schränkte TM

10.6 DTM, NTM, LBTM und Sprachklassen

Wie wir bereits festgestellt haben, gibt es drei verschiedene Arbeitsergebnisse von DTM:

1. Die DTM stoppt und der erreichte Zustand ist *ein* Endzustand.
2. Die DTM stoppt und der erreichte Zustand ist *kein* Endzustand.

3. Die DTM stoppt *nicht*. (Unendlicher Rechtslauf, Linkslauf bzw. Endloszyklus)

Nur im ersten Fall wird das Eingabewort akzeptiert.



Definition 10.4

Eine Sprache L heißt **rekursiv aufzählbar** (semientscheidbar), wenn es eine DTM M gibt, mit $L = L(M)$.

Die Elemente solcher Sprachen können algorithmisch effektiv erzeugt, aufgelistet oder eben im anschaulichen Sinne aufgezählt werden. Dies wird in der Berechenbarkeitstheorie genauer untersucht.



Satz 10.1

Die Menge $\mathcal{L}_{re} = \{L(M) \mid M \text{ ist eine DTM}\}$ aller rekursiv aufzählbaren Sprachen stimmt genau mit der Menge aller CHOMSKY-Typ-0-Sprachen überein.

Beweis

Einen entsprechenden Beweis findet man in der Literatur. □

recursive enumerable
„re“ steht für „recursive enumerable“. Dabei handelt es sich um einen Begriff, der sich historisch etabliert hat. Dies gilt auch für den weiter unten verwendeten Begriff der „rekursiven Sprache“.

stets anhaltende
DTM
DTM, die für alle möglichen Eingabewörter w über dem zugehörigen Alphabet (akzeptierend: $w \in L(M)$ oder ablehnend: $w \notin L(M)$) stoppen, nennt man auch *stets anhaltende DTM*. Die DTM M_2 aus Beispiel 10.2 besitzt diese Eigenschaft, M_1 aus Beispiel 10.1 jedoch nicht. Wie Übungsaufgabe 10.1 erkennen lässt, stoppt M_1 beispielsweise nicht auf a .



Definition 10.5

Eine Sprache L heißt **rekursiv** (entscheidbar), wenn es eine stets anhaltende DTM M gibt, mit $L = L(M)$.

Man kann zeigen, dass die Menge aller rekursiven (algorithmisch erkennbaren) Sprachen \mathcal{L}_r eine Untermenge der Menge der rekursiv aufzählbaren Sprachen \mathcal{L}_{re} ist, die Klasse der ksS (Typ 1) jedoch vollständig enthält.

$$\mathcal{L}_{Typ\ 1} \subset \mathcal{L}_r \subset \mathcal{L}_{re}.$$

Für die Feststellung, ob eine rekursiv aufzählbare Sprache entscheidbar ist oder nicht, gibt es kein allgemein anwendbares Verfahren. In der Berechenbarkeitstheorie wird gezeigt, dass es keinen Algorithmus geben kann, der diese Prüfung durchführt.

Die Klasse der CHOMSKY-Typ-1-Sprachen wird durch linear beschränkte TM, kurz: LBTM, beschrieben, d.h.

LBTM

$$\mathcal{L}_{LBTM} = \mathcal{L}_{\text{Typ } 1}.$$

Bei LBTM handelt es sich dabei um eine eingeschränkte Form nichtdeterministischer TM (NTM), deren Kopf nur innerhalb des Bandbereiches, der zu Beginn der Abarbeitung durch das Eingabewort belegt wird, agieren, d.h. sich bewegen und schreiben, darf.

Beispiel 10.3

$L = \{a^n b^n c^n \mid n \geq 1\}$ ist eine Typ-1-Sprache. Sie kann durch eine ksG $G = (\{A, B, S\}, \{a, b, c\}, P, S)$ mit



$$\begin{aligned} P = \{ &S \rightarrow abc \mid aAbc \\ &Ab \rightarrow bA \\ &Ac \rightarrow Bbcc \\ &bB \rightarrow Bb \\ &aB \rightarrow aa \mid aaA \} \end{aligned}$$

definiert werden.

Rückblickend stellen wir fest, dass die in Beispiel 10.2 angegebene DTM M_2 den Eingabewort-Bandbereich nicht verlässt und folglich leicht zu einer LBTM umkonstruiert werden könnte.

Da die Arbeitsweise von NTMs mit DTMs simuliert werden kann, gilt zusammenfassend:

$$\mathcal{L}_{NTM} = \mathcal{L}_{DTM} = \mathcal{L}_{\text{Typ } 0} = \mathcal{L}_{re}.$$

Abbildung 10.3 fasst die relevanten Sprachklassen zusammen.

10.7 TM in Komplexitäts- und Berechenbarkeitstheorie

In der *Komplexitätstheorie* werden TM (DTM und NTM) als Modelle verwendet, um den Aufwand algorithmischer Berechnungen abstrakt, d.h. z.B. Prozessor-unabhängig, zu definieren. Algorithmische Operationen werden (wenigstens gedanklich) durch die atomaren TM-Instruktionen ausgedrückt. Ein Aufwandsmaß ergibt sich dann aus der Anzahl der zur Ausführung der Operationen jeweils erforderlichen TM-Schritte.

Während sich die Komplexitätstheorie mit der *Effizienz von Algorithmen*

⁴Es ist bisher nicht bekannt, ob linear beschränkte DTM dafür ebenso ausreichen würden, d.h. ob $\mathcal{L}_{DLBTM} = \mathcal{L}_{(N)LBTM}$ gilt oder nicht.

Komplexitäts-theorie

Effizienz von Algorithmen

- Überabzählbar unendliche Mengen

Bsp.: Menge aller Wortmengen über einem Alphabet

- Abzählbare Mengen

Bsp.: Menge der wahren arithmetischen Formeln

- rekursiv aufzählbare Spr. = DTM/NTM-Spr. = Chomsky-Typ-0-Spr.

Bsp.: Menge aller Algorithmen, Halteproblem

- rekursive Sprachen = Sprachen der stets stoppenden TM

- ksS = LBTM-Sprachen = Chomsky-Typ-1-Sprachen

Bsp.: $L = \{a^n b^n c^n \mid n \in N\}$

- kfS = NKA-Sprachen = Chomsky-Typ-2-Sprachen

Bsp.: $L = \{a^n b^n \mid n \in N\}$

- dkfS = DKA-Sprachen

Bsp.: Palindrome mit markierter Wortmitte

- rS = regExp-Sprachen = Chomsky-Typ-3-Sprachen

Bsp.: Zahlwörter

- endliche Sprachen (alle Elemente angebar)

Abbildung 10.3: Hierarchie der relevanten Sprachklassen (ohne *LR*- und *LL*-Sprachen, s. Abbildung 8.3 auf Seite 205)

Berechenbarkeitstheorie

beschäftigt, gehören Fragen nach der Entscheidbarkeit von Sprachen und Lösbarkeit von Problemen in die *Berechenbarkeitstheorie*. TM haben dabei eine fundamentale Bedeutung, da sie zur Präzisierung der dort verwendeten Begriffe dienen.

TURING-Berechenbarkeit

Betrachtet man den auf dem Band nach dem Stopps einer TM hinterlassenen Inhalt in einem verabredeten Bereich als Funktionswert, so kann man TM zur Berechnung n -stelliger Wortfunktionen $(\Sigma^*)^n \rightarrow \Sigma^*$ (s. Abschnitt 10.8) einsetzen. Der Begriff der Berechenbarkeit wird durch die *TURING-Berechenbarkeit* präzisiert. Dies vervollkommnet die herausragende Leistung TURINGs deutlich und hebt ihn auf die Stufe der bedeutendsten Informatiker überhaupt.

In der Berechenbarkeitstheorie gibt es eine Reihe von Entscheidungsproblemen, die mit TM präzisiert und für bestimmte Sprachklassen beantwortet werden. Beispielsweise kann gezeigt werden, dass das bereits in Abschnitt 2.7 eingeführte Wortproblem nur für Chomsky-Typ-1(2,3)-Sprachen allgemein entscheidbar ist, jedoch nicht für Typ 0. Seit 1997 wissen wir von Sénizergues (Universität Bordeaux), dass das Äquivalenzproblem⁵ für dkfS entscheidbar ist. Hingegen ist das Eindeutigkeitsproblem⁶ für kfS nicht entscheidbar.

Um Entscheidungsprobleme für Sprachklassen zu bearbeiten, bedarf es jeweils einer TM, die in der Lage ist, die Arbeitsweise jeder anderen TM (der betrachteten Klasse) zu simulieren. Eine TM mit dieser Eigenschaft nennt man *universelle TM*,

⁵Sind zwei gegebene Sprachen L_1 und L_2 äquivalent?

⁶Gibt es für eine Sprache L eine eindeutige Grammatik G ?

kurz: UTM. Wohlgemerkt: Dabei handelt es sich um definitionstreue TM (s. Definition 10.1), nicht etwa um einen neuen Typ. Auf dem Band erwartet eine UTM U zwei Eingabewörter, zwischen denen genau ein neues Eingabezeichen $\#$ steht:

$$\langle M \rangle \# w.$$

M ist die TM, deren Verhalten von U simuliert werden soll. $\langle M \rangle$ ist eine Codierung von M , d.h. das zu M gehörende Wort.

Die Überführungsfunktion von U wird also so konstruiert, dass sie die Überführungsfunktion δ_M von M , die sich innerhalb des Wortes $\langle M \rangle$ befindet, abtastet und interpretiert. $U(\langle M \rangle, w)$ liefert dann das gleiche Resultat, wie $M(w)$, und zwar für alle $w \in \Sigma^*$.

Das beschriebene Verhalten der UTM ist dem Prinzip der bis heute aktuellen VON-NEUMANN-Rechner verwandt.

Da die Angabe einer UTM angesichts der doch sehr elementaren TM-Instruktionen überaus aufwendig ist, wollen wir hier darauf verzichten. Grundsätzlich ist es aber möglich, bei Vorgabe einer (beliebigen) Kodierung $\langle M \rangle$ eine solche UTM U_{cod} anzugeben. Bei der Definition der Überführungsfunktion einer UTM muss die gewählte Kodierung cod von M berücksichtigt werden.

Für Typ-0-Sprachen ist das Wortproblem bekanntlich nicht allgemein entscheidbar. Der für die Entscheidung von Typ-1-Sprachen erforderliche exponentielle Aufwand, der durch die Erzeugung aller Satzformen bis zur Länge des Eingabewortes verursacht wird, ist für die Praxis vollkommen unbrauchbar. Während für kfS der sog. CYK⁷-Algorithmus (s. Seite 166) das Wortproblem mit einer Zeiteffizienz in $\mathcal{O}(n^3)$ löst, geschieht dies für deterministisch kfS – ebenso wie für reguläre Sprachen – mit linearem Aufwand, d.h. in $\mathcal{O}(n)$.

10.8 TM zur Berechnung von Funktionen

TM berechnen *partielle* Wortfunktionen der Form

$$f : (\Sigma^*)^n \rightarrow \Sigma^*.$$

Dass es sich im Allgemeinen um *partielle* Funktionen handelt, folgt aus der Arbeitsweise einer TM, die ggf. für bestimmte Argumente nicht anhält. Partielle n -stellige Wortfunktionen sind die allgemeinsten Funktionsklassen überhaupt. Sie umfassen die totalen Funktionen als eine echte Teilmenge.

Die n Argumente w_1, w_2, \dots, w_n , mit $w_i \in \Sigma^*$, werden durch jeweils genau ein neues Eingabezeichen $\#$ untereinander getrennt auf das Band geschrieben. Für einen

⁷COOKE, YOUNGER, KASAMI

UTM

Arbeitsweise
einer UTMVON-
NEUMANN-
RechnerEffizienz von
Algorithmen zur
Lösung der
Wortproblemepartielle
Wortfunktionen

Bandeintrag der Form

$$w_1 \# w_2 \# \dots \# w_n$$

Funktionswert

berechnet die darauf angesetzte TM M den Funktionswert $f_M(w_1, w_2, \dots, w_n)$, falls dieser existiert. Nach dem Stoppen von M wird das zugehörige Ausgabewort vom Band abgelesen: Der Lese/Schreib-Kopf steht auf dem ersten Zeichen dieses Wortes, das sich von links nach rechts bis zum Feldinhalt unmittelbar vor dem ersten auftretenden Blankzeichen erstreckt.

Ebenso, wie das Aufbringen von Eingabeworten (Argumenten), sind das Ablesen und Dekodieren des Ausgabewortes (Funktionswertes) externe Vorgänge, die nicht zur TM-Definition gehören.



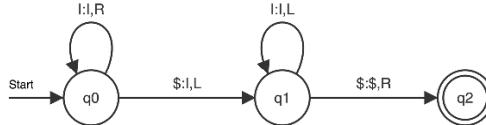
Beispiel 10.4

Nachfolger-Maschine: Eine *Nachfolger-Maschine* ist eine TM, die zu einer in unärer Darstellung (Eingabealphabet umfasst nur genau ein Zeichen, meist als I symbolisiert) auf das Band geschriebenen Zahl deren Nachfolger berechnet und diesen in unärer Darstellung (im Jargon: Bierdeckelnotation) auf dem Band hinterlässt.

Nachfolger-Maschine

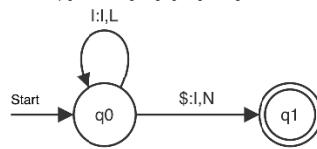
Eine Lösungsidee besteht darin, den Lese/Schreib-Kopf schrittweise nach rechts zu verschieben, wobei sämtliche Vorkommen des Eingabealphabetzeichens I reproduziert werden. In das (von links her) erste Feld, mit einem Blankzeichen $\$$ wird I geschrieben. Danach fährt der Kopf an die Anfangsposition zurück und bleibt dort stehen.

$$M = (\{q_0, q_1, q_2\}, \{I\}, \{\$\}, \delta, q_0, \$, \{q_2\}) \text{ mit}$$



Eine alternative Lösungsidee ist überraschend kurz: Der Kopf geht einfach einen Schritt nach links, schreibt dort I und bleibt auf dieser Position stehen.

$$M = (\{q_0, q_1\}, \{I\}, \{\$\}, \delta, q_0, \$, \{q_1\}) \text{ mit}$$



Übung 10.4



Erproben Sie beide Lösungen. Verwenden Sie auch „Extrem-Eingaben“, wie etwa das leere Wort ϵ und I .

Übung 10.5



Geben Sie eine DTM mit $\Sigma = \{I\}$ an, die den Vorgänger einer beliebigen natürlichen Zahl n ($n > 0$) berechnet. n und $n - 1$ sollen unär als Worte über Σ dargestellt werden.



Busy beaver

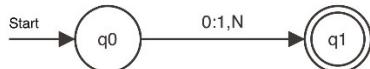
Beispiel 10.5

Busy beaver: Ein *fleißiger Biber (busy beaver)* ist eine DTM

$M = (\{q_0, q_1, \dots, q_{n-1}, q_n\}, \{0\}, \{0, 1\}, \delta, q_0, 0, \{q_n\})$, die auf einem mit Nullen (0) vorbelagerten Eingabeband nach endlich vielen Schritten stoppt und eine maximale Anzahl von Einsen (1) hinterlässt. (An DTM, die nicht stoppen und dabei ggf. unendlich viele Einsen hinterlassen, ist hier nicht gedacht.) Dabei sollen alle Einsen gezählt werden, auch wenn sie nicht zu einer zusammenhängenden Kette gehören, also durch Nullen unterbrochen wurden. Außerdem spielt die Position des Kopfes nach dem Halt keine Rolle.

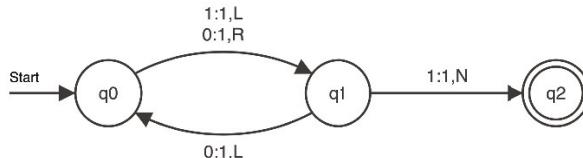
Für $n = 1$ ist die Lösung einfach zu sehen: Der Biber produziert genau *eine* Eins auf dem Band und „legt sich wieder schlafen“.

$M = (\{q_0, q_1\}, \{1\}, \{0, 1\}, \delta, q_0, 0, \{q_1\})$ mit



Auch für $n = 2$ kann die Lösung leicht verifiziert werden.

$M = (\{q_0, q_1, q_2\}, \{1\}, \{0, 1\}, \delta, q_0, 0, \{q_2\})$ mit



Nach sechs Schritten hat der Biber 4 Einsen produziert.

Für $n > 0$ entsteht auf diese Weise die RADOsche⁸ Funktion $n \rightarrow E(n)$, wobei $E(n)$ RADO-Funktion die maximale Einsen-Anzahl ist.

| n | $E(n)$ | Bemerkungen |
|-----|---|--|
| 1 | 1 | |
| 2 | 4 | |
| 3 | 6 | (1962: Tibor Rado, Shen Lin) |
| 4 | 13 | (1973: Bruno Weimann) |
| 5 | 501(?) | (1983: Schult) |
| | 1915(?) | (1984: Uhing) |
| 6 | 2075(?) | (Schult, mit spezieller Hardware) |
| : | : | |
| 12 | $6 \cdot 4096^{4096^{4096^{\dots^{4096^4}}}}$ | rechnerisch: Die Punkte stehen für insgesamt 162 mal 4096. |

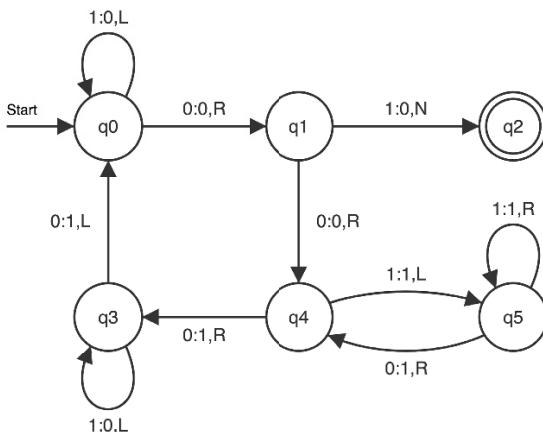
⁸Tibor RADO: ungarischer Mathematiker: geb.: 02.06.1895 in Budapest; gest. 12.12.1965 in New Smyrna Beach/Florida

Manche Funktionswerte in der Tabelle sind mit einem Fragezeichen markiert: Man vermutet, dass die bisher gefundene größtmögliche Anzahl von Einsen doch noch übertroffen werden kann.

Beim Experimentieren mit busy-beaver-TM gelingt es nicht immer, die eigentliche Aufgabenstellung zu erfüllen. Mitunter entstehen dabei recht „merkwürdige Biber“, wie etwa die folgende, ironisch als „Wissenschaftsbiber“ bezeichnete DTM.

Ein Wissenschaftsbiber ist unheimlich fleißig und strampelt sich ab, aber produziert meist nicht viel: Die folgende TM macht 187 Schritte, rückt dabei 8 Felder vor, aber schreibt keine einzige Eins.

$M = (\{z_0, z_1, z_3, z_4, z_5, z_6\}, \{0\}, \{0, 1\}, \delta, z_0, 0, \{z_3\})$ mit



Fragestellungen „rund um“ diese Funktion sind vor allem bei Informatik-Olympiaden beliebt. In der Berechenbarkeitstheorie lernt man, dass die (wohldefinierte) RADO-Funktion *nicht berechenbar* ist.

Übung 10.6



Entwickeln Sie eine DTM, die zu zwei in unärer Darstellung („Bierdeckelnotation“) auf das Band geschriebenen Wörtern für die Zahlen a und b deren Summe $a + b$ berechnet, indem sie das jeweilige Resultatwort in unärer Darstellung auf dem Band hinterlässt.

Übung 10.7



Entwickeln Sie eine DTM mit $\Sigma = \{a, b, c\}$, die als *Bandlöschmaschine* arbeitet. Eine Bandlöschmaschine entfernt ein Eingabewort, indem es die Felder des Bereiches ab Startposition bis zum ersten Blankzeichen mit dem Blankzeichen überschreibt.

Übung 10.8



Konstruieren Sie eine DTM, die die Sprache
 $L = \{w \mid w \in \{a, b\}^* \text{ und } w = a^{2n} \mid n \geq 0\}$ akzeptiert.

Sachverzeichnis

- Überführungsfunktion, 53
 \emptyset , 108
 ϵ , 108
- Abgeschlossenheit, 71
Ableitungsbaum, 23, 26
abstrakte Automaten, 7
abstrakte Spezifikation, 211
Abstrakten Syntaxbaum, 23
action, 204
Äquivalenzklassen, 78
Äquivalenzrelation, 78
Akzeptanzverhalten
 EA, 54
akzeptiertes Wort, 50
Akzeptor, 51
Alphabet, 7, 8
Analyseteil, 211
API, 2
AST, 23, 27
attribuierten Parsebaum, 211
Ausdruck
 regulärer, 103, 105
 regulärer, 69
Automat
 abstrakter, 51
 endlicher, 51–53
 nichtdeterministischer endlicher, 60,
 61
 zellularer, 102
- Backreference, 115
Backtracking, 38, 103
Backus-Naur-Form, 22
Bandlöschnmaschine, 264
Berechenbarkeitstheorie, 251
Beschreibungssprache, 2
Betriebssystem, 2
Bison, 208
Bootstrapping, 125
- Bottom-up-Analyse, 136, 199
Brute force, 45
Busy beaver, 263
- Chart-Parser, 154
Chomsky, 103
Chomsky, Noam, 41
Chomsky-Hierarchie, 41
CNF, 166
Compilation, 1
Compiler, 4, 121
 inkrementelle, 122
Compilerbau, 7
Cross-Compiler, 127
CYK, 155, 166
CYK-Verfahren, 167
- dangling-else-Problem, 40, 194
DEA, 52
Durchschnitt, 72
dynamisches Programmieren, 167
- Earley-Algorithmus, 38
Earley-Parser, 155
EBNF, 22, 187
Effizienz, 259
Eingabeabschlusszeichen, 179
Eingabeverifikierung, 116
Endzustand, 50, 54
Entscheidungsproblem, 46
 ϵ -Hülle, 93
 ϵ -freie Regeln, 44
- Fachsprache, 2
Falle, 55
FEN, 224
Figuren, 224
FIRST, 177
First-wins-Strategie, 133
FLACI, 2

- Flex, 208
- Flussüberquerungsproblem, 74
- FOLLOW, 179
- formale Sprachen, 7
- formalen Sprachen, 15
- GLL-Parser, 155
- Globaler Code, 212
- GLR-Parser, 155
- goto, 204
- Grammatik, 32
 - äquivalente, 40
 - formale, 7, 32
 - kontextfreie, 33, 41
 - kontextsensitive, 41
 - mehrdeutige, 38
 - reduzierte, 135
 - reguläre, 41
 - unbeschränkte, 41
- grammatikalische Regeln, 7
- Grammatiktransformationen, 163
- Handle, 202
- Harrison, 155
- IGNORE, 130
- Implementationssprache, 4
- inhärent mehrdeutig, 40
- Interpretation, 1
- Interpreter, 3, 4, 121
- JavaScript, 2
- Jison, 208
- JIT-Compiler, 122
- JSFiddle, 243
- jsPDF, 238
- Keller, 140
- Kettenregeln, 164, 165
- kfG, 33
- Kleene, 104
- Komplement, 71
- Komplexitätstheorie, 259
- Konfigurationenfolge, 55, 143
- Konkatenation, 11
- kontextfrei, 42
- kontextsensitiv, 42
- Koordinatensystem, 235
- Längenmonotonie, 43
- längenlexikografische Ordnung, 13
- LALR(1)*-Sprachen, 207
- LALR-Sprachen, 206
- Lesekopf, 53
- Lex, 208
- Lexem, 130
- Linien, 225
- Linksableitung, 36
- Linksfaktorisierung, 197
- LL(1)*-Forderung 2, 179
- LL(1)*-Forderung 1, 177
- LL(1)*-Parser, 176
- LL(1)*-Sprachen, 176
- longest-prefix-Strategie, 133
- lookahead, 173
- Lookahead-*LR(1)*-Sprachen, 207
- LR(k)*-Sprachen, 199
- LR(1)-Sprachen, 206
- LR-Parsetabelle, 203
- Maschine
 - endliche, 98
 - virtuelle, 122
- Maschinencode, 2
- MEALY, 98
- mehrdeutig, 27, 38, 65
- Mehrdeutigkeit
 - syntaktische, 29
- Mehrphasen-Compilation, 230
- Mehrphasencompilation, 234
- Memoizing, 182
- Menge
 - überabzählbar unendlich, 14
 - abzählbar unendlich, 14
- MiniJavaScript, 18
- ML, 213
- MOORE, 98
- Multistack-Automat, 249
- Musiksprache, 3
- Nachfolgermaschine, 262
- NEA, 60, 61
- Nichtterminale, 22, 32
 - unnütze, 164
- Operatorbaum, 27

- Packrat-Parser, 154
- Parsebaum, 23, 26
- Parser, 117
- Parsergenerator, 186
- Parsergeneratoren, 163, 208
- Parsertabelle, 182
- Parsing, 23
- Pattern Matching, 103
- Pattern matching, 31
- Paull's Algorithmus, 193
- Petri-Netz, 102
- Phasen eines Compilers, 128
- Phrasenstrukturgrammatik, 42
- PL/0, 187
- PNG, 234
- pop, 140
- POSIX, 114
- Postscript, 234
- Potenzmenge, 61
- Pragmatik, 1, 2, 15
- Produktion, 22
- Produktionen, 32
- ProgrammingWiki, 242
- Programmieren
 - dynamisches, 154
- Programmiersprache, 2
- Pumping Lemma, 88
 - für kfS, 169
- push, 140
- QR, 5
- QR-Scanner, 5
- Quellsprache, 4, 29, 121
- Rückwärtsreferenztechnik, 115
- reduce, 201
- reduce/reduce-Konflikt, 206
- reduzierte Grammatik, 227
- Regel, 22
- Regeln, 32
- regulären Ausdruck (RA), 103
- regulärer Ausdruck, 7
- Reihe, 225
- rekursiv aufzählbar, 258
- rekursiver Abstieg, 186
- Ritchi, 103
- S-Attribute, 212
- Sackgasse, 37
- Satz von KLEENE, 107
- Satzform, 32
- Satzsymbol, 32
- Scanner, 117, 129
- Scanner-Generator, 131
- Scannergenerator, 189
- Scannergenerator, 117, 208
- Scheduling a call, 217
- Semantik, 1, 2, 15
- semantische Regel, 211
- Semi-Thue-System, 42
- shift, 201
- shift/reduce-Konflikt, 206
- Simultaneous derivation, 45
- Skriptsprachen, 121
- SLR(1)-Sprachen, 206
- SLR-Sprachen, 206
- Spieleprogrammierung, 74
- Spielkonsole, 127
- Spitzensymbol, 32
- Sprachdesign, 18
- Sprache, 1, 16
 - akzeptierte, 55
 - deterministisch-kontextfrei, 173
 - endliche, 6
 - formal, 7
 - kontextfreie, 37
 - kontextsensitive, 249
 - rekursiv aufzählbar, 258
- Sprachklassen, 41
- Stapel, 139
- stark-*LL(k)*-Sprache, 175
- Startsymbol, 32
- Startzustand, 50, 54
- SVG, 6, 219, 224
- syntaktische Regeln, 7
- Syntax, 2
 - abstrakte, 27
 - konkrete, 18, 27
- Syntaxdiagramme, 22
- Syntaxanalyse, 23
 - prädiktive, 173
- Syntaxbaum
 - abstrakter, 27
- Syntheseteil, 211
- synthetisierte Attribute, 212

- Systeme
 - eingebettete, 127
- T-Diagramm, 5, 122
- T-Diagramme, 226
- TDL, 208, 211
- Terminale, 21, 32
- textuelle Notation, 224
- Thompson, 103
- Thompson-Konstruktion, 107
- Token, 129, 130
- Tokendefinitionen, 215
- Tokenklasse, 129
- Tokenklassen, 117
- Tokenliste, 129
- Tokentyp, 129
- top of stack, 140
- Top-down-Analyse, 136, 174
- Top-down-Syntaxanalyse, 173
- Transcompiler, 118
- Trap, 51
- trap state, 51, 55
- Trojaner, 126
- Turing, 250
- Turing-Berechenbarkeit, 260
- Turingmaschine, 250
 - linear beschränkte, 257
 - universelle, 261
- Turtle, 118
- Überführungsfunktion, 53
 - erweiterte, 56, 63
- Übergang
- spontaner, 91, 142
- Übersetzung
 - syntaxgesteuerte, 29, 211
- unendliche Sprachen, 7
- uvw-Theorem, 88
- Verkettung, 11
- virtuellen JS-Maschine, 2
- von-Neumann-Rechner, 261
- Vorübersetzung, 231
- Vorausschau, 173
- Vorausschauzeichen, 202
- Warteschlangen-Automat, 250
- Whitespaces, 130
- Wort, 7, 9
 - leeres, 9, 10
- Wortbildung, 11
- Wortlänge, 11
- Wortmenge, 12
- Wortproblem, 46
- Yacc, 208
- Zeichen, 7, 8
- Zeichenroboter, 118
- Zielsprache, 4, 29, 121
- Zustände, 49
- Zustand, 52
- Zustandseliminierungsverfahren, 109
- Zustandsmodell, 49
- Zuweisung, 18
- Zwischensprache, 230, 234