Rheinische Friedrich-Wilhelms-Universität Bonn
Institut für Informatik
Abteilung V

Diplomarbeit

Implementierung eines Generators für erweiterte LR(k)-Parser

Heinz Christian Steinhausen
Matrikelnummer 1422315
30. Juni 2011

Erstgutachter: Prof. Dr. Norbert Blum

Rheinische Friedrich-Wilhelms-Universität Bonn
Institut für Informatik
Abteilung V

Diplomarbeit

Implementierung eines Generators für erweiterte LR(k)-Parser

Heinz Christian Steinhausen
Matrikelnummer 1422315
30. Juni 2011

Erstgutachter: Prof. Dr. Norbert Blum

Eidesstattliche Erklärung

gemäß §19 (7) der Prüfungsordnung von 1998

Hiermit versichere ich, dass ich die vorliegende Diplomarbeit „Implementierung eines Parsergenerators für erweiterte LR(k)-Parser“ selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Wörtliche und sinngemäße Wiedergaben aus anderen Quellen sind kenntlich gemacht und durch Zitate belegt. Dasselbe gilt sinngemäß für Tabellen und Abbildungen.
Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt.

Bonn, 30. Juni 2011 Heinz Christian Steinhausen

Warenzeichen

Die in dieser Arbeit genannten Unternehmens- und Produktbezeichnungen können geschützte Marken- oder Warenzeichen sein. Auch wenn sie nicht gesondert gekennzeichnet sind, gelten die entsprechenden Schutzbestimmungen und die Besitzrechte der jeweiligen eingetragenen Eigentümer.

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Vielen Computerprogrammen gemeinsam ist die Notwendigkeit, strukturierte Eingaben zu analysieren. Beispiele sind die Übersetzung eines Programm-Quellcodes aus einer höheren Programmiersprache in Maschinensprache, das Einlesen einer HTML-Datei zur Darstellung im Webbrowser oder das Öffnen eines Dokumentes im Office-Programm zur weiteren Bearbeitung. Die Eingaben müssen eingelesen, in eine für die Aufgabenstellung geeignete programminterne Darstellung gebracht und gegebenenfalls für die Ausgabe in ein anderes Format umgewandelt werden.

Das Einlesen der Eingabe geschieht häufig in zwei Schritten. Zunächst findet die lexikalische Analyse statt, die mit der Rechtschreibprüfung eines natürlichsprachlichen Textes vergleichbar ist. Die Eingabe wird dabei in einzelne Bausteine, die sogenannten Token zerlegt. Diese Token bilden die Eingabe für die nächste Analysestufe, die Syntaxanalyse. Dabei prüft eine spezielle Programmkomponente, der Parser, ob die Anordnung der Token den Regeln einer gegebenen Grammatik gehorcht. Gleichzeitig werden in dieser Phase die vom Programm benötigten Informationen in eine interne Darstellung gebracht und teilweise auch sofort weiter verarbeitet.

Grammatiken und die Sprachen, die durch sie erzeugt werden, lassen sich nach Zeit- und Platzbedarf für ihre Analyse in Kategorien einteilen. Einen ausführlichen Überblick über die Analysemethoden für verschiedene Klassen von Grammatiken bietet [GJ08], ergänzt um eine umfangreiche Liste kommentierter Literaturverweise zum Thema Parsing.

Die vorliegende Diplomarbeit befasst sich mit der Klasse der $LR(k)$-Grammatiken, die von Knuth in [Knu65] definiert wurde. Für eine $LR(k)$-Grammatik G kann stets ein Parser konstruiert werden, der die Eingabe von links nach rechts analysiert und dabei eine Rechtsableitung in linearer Zeit erzeugt, gemessen an der Länge n der Eingabe. Dafür stehen dem Parser k Zeichen Lookahead zur Verfügung, d. h. er trifft seine Entscheidungen nach Betrachten der nächsten maximal k ungelesenen

Symbole der Eingabe. Konstruktionsbedingt kann die Anzahl von Zuständen eines Automaten, der solch einem Parser zugrunde liegt, auf ein theoretisches Maximum von $2^{|G|^{k+1}}$ anwachsen. Dies kann schon für $k = 1$ zu groß für den praktischen Einsatz eines $LR(k)$-Parsers sein. In vielen praktischen Projekten finden daher Teilklassen von Grammatiken Anwendung, die gegenüber $LR(k)$ weiter eingeschränkt sind. Ein Beispiel sind die $LALR(1)$-Grammatiken, für die es unter anderem die Parsergeneratoren Yacc und Bison gibt.

Blum stellt in [Blu10a] ein neues Modell für $LR(k)$-Parser vor, deren Größe polynomiell in der Größe der Grammatik und der Länge des Lookaheads ist, nämlich $O(|G| + \#LA \cdot |N| \cdot k^2)$, wobei $|N|$ die Anzahl Nichtterminalsymbole der Grammatik G angibt und $\#LA$ die maximale Anzahl von Lookahead-Strings, die G ermöglicht. Dazu verwaltet der Parser einen Graphen $\mathcal{G}$ der Größe $O(|G| \cdot n)$. Die genannte Arbeit wurde auch in gekürzter Form als [Blu10b] veröffentlicht.

Um die Praxistauglichkeit von Blums Parsermodell zu prüfen, wurden für die vorliegende Arbeit Algorithmen für die in [Blu10a] vorgeschlagene Parserkonstruktion detailliert ausgearbeitet und ein Parsergenerator implementiert. Dabei bestand eine besondere Herausforderung darin, den Generator anschließend im Hinblick auf Größe und Geschwindigkeit der erzeugten Parser mit anderen Lösungen zu vergleichen.

1.2 Gliederung der Arbeit

Das folgende Kapitel gibt zunächst einen Überblick über die verwendeten Begriffe und das Konzept eines $LR(k)$-Parsers. Zuerst werden einige allgemeine Eigenschaften von $LR(k)$-Grammatiken beschrieben. Abschnitt 2.1 ist der Beschreibung des kanonischen $LR(k)$-Parsers gewidmet, und im zweiten Abschnitt wird anschließend Blums Modell des erweiterten $LR(k)$-Parsers erläutert.

In Kapitel 3 wird der Grundalgorithmus aus [Blu10a] um die dort nur textuell beschriebenen Konstruktionen ergänzt, die ihn zu einem $LR(k)$-Parser machen. Dabei wurde großer Wert darauf gelegt, die einzelnen Schritte des Parsers durch eine Unterteilung in Teilalgorithmen deutlich herauszuarbeiten. Nach einer Beschreibung des Basisalgorithmus wird in Abschnitt 3.1 die Durchführung der Expansionsschritte beschrieben. In Abschnitt 3.2 folgt eine ausführliche Beschreibung $LR(k)$-konformer Reduktions- und Leseschritte. Anschließend werden in den Abschnitten 3.3 und 3.4 Algorithmen für die Suche nach den Knoten, die in einem Reduktions-/Leseschritt zu behandeln sind, ausformuliert und im Detail erklärt. Diese Algorithmen imple-

mentieren die Verfahren, die in Abschnitt 7 von [Blu10a] erläutert, aber noch nicht als Algorithmus ausformuliert werden. In Abschnitt 3.3 wird zuerst der einfachere Fall $k = 1$ betrachtet, bevor in Abschnitt 3.4 das Suchverfahren für allgemeine (aber kleine) Werte von k beschrieben wird. Der anschließende Abschnitt 3.5 ist der Behandlung von Zyklen im Graphen während dieser Suche gewidmet, was in Blums Arbeit nur kurze Erwähnung findet. In Unterabschnitt 3.5.2 wird ein neu entwickeltes modifiziertes Suchverfahren vorgestellt, das keine Sonderbehandlung von Zyklen erfordert und für Werte von $k = 1$ oder $k = 2$ Vorteile verspricht. Abschnitt 3.6 befaßt sich mit der Behandlung der $FIRST_k$-Mengen im Generator und in den erzeugten Parsern und weicht ebenfalls von dem in [Blu10a] beschriebenen Vorgehen ab. Den Abschluß des Kapitels bildet in Abschnitt 3.7 eine Laufzeitanalyse des entwickelten Parsers.

In Kapitel 4 werden die Implementierungen des Parsergenerators (Abschnitt 4.1) und der erzeugten Parser (Abschnitt 4.2) erläutert. Insbesondere werden die Datenstrukturen beschrieben, die für den Parser und den von diesem verwalteten Graphen gewählt wurden, und deren Platzbedarf erörtert.

Die nach der Implementierung durchgeführten Praxistests, insbesondere im Vergleich mit anderen Parsergeneratoren, werden in Kapitel 5 beschrieben. Nach einer Vorstellung der verwendeten Grammatiken in Abschnitt 5.1 werden in Abschnitt 5.2 die eigentlichen Tests bezüglich Parsergröße, Laufzeit und Speicherbedarf beschrieben und die Ergebnisse gedeutet.

Abschließend folgt in Kapitel 6 eine Zusammenfassung der Ergebnisse der vorliegenden Arbeit und ein Ausblick auf Ansatzpunkte für eine weitere Erforschung des Themas.

2 LR-Grammatiken und -Parser

In diesem Kapitel werden die Begriffe beschrieben, die der vorliegenden Arbeit zugrunde liegen. Insbesondere werden die Konzepte des kanonischen und des erweiterten $LR(k)$-Parsers vorgestellt. Dabei wird weitgehend die Notation aus [Blu10a] verwendet.

Die ersten beiden Schritte beim Einlesen strukturierter Eingabedaten bilden üblicherweise die lexikalische und syntaktische Analyse. Bei der *lexikalischen Analyse* wird die Eingabe in *lexikalische Einheiten*, auch *Token* oder *Terminalsymbole* genannt, unterteilt. Dabei werden gegebenenfalls für die Anwendung irrelevante Eingabebestandteile wie Kommentare oder Leerstellen herausgefiltert. Der Programmteil, der diesen ersten Analyseschritt durchführt, wird *(lexikalischer) Scanner* oder auch *Lexer* genannt.

Für den nächsten Schritt, die *syntaktische Analyse*, dienen die von Scanner generierten Token als Eingabe. Deren Anordnung wird von einem *Parser* auf syntaktische Korrektheit überprüft. Zu diesem Zweck wird geprüft, ob die Reihenfolge der Token in der Eingabe mit den Regeln einer vorgegebenen Grammatik konform ist. Im Regelfall generiert der Parser für eine korrekte Eingabe einen *Syntaxbaum* (auch: *Ableitungsbaum*), aus dem die Struktur der Eingabe ablesbar ist. In den folgenden Abschnitten werden die Begriffe definiert, die für eine Syntaxanalyse anhand von $LR(k)$-Grammatiken relevant sind.

Für ein Alphabet Σ und eine natürliche Zahl k ist die Menge aller Wörter der Länge $\leq k$ über Σ definiert durch

$$\Sigma^{\leq k} = \{w \mid w = a_1 \dots a_l,\ l \leq k,\ \text{mit } a_i \in \Sigma,\ 1 \leq i \leq l\}.$$

Für ein Alphabet Σ bzw. eine Sprache L ist die *Kleenesche Hülle* Σ^* bzw. L^* die Menge aller Wörter, die durch eine endlich lange *Konkatenation*, d. h. Aneinanderreihung, der Zeichen aus Σ bzw. L gebildet werden können. Dies schließt auch das *leere Wort* ε mit ein. Formal:

$$\Sigma^* = \bigcup_{k \in \mathbb{N}_0} \Sigma^{\leq k}.$$

Definition 1 (Kontextfreie Grammatik): Eine *kontextfreie Grammatik* ist ein Vier-Tupel $G = (N, \Sigma, P, S)$, bestehend aus einer endlichen, nichtleeren Menge N von *Nichtterminalsymbolen*, auch *Variablen* genannt, einer endlichen, nichtleeren Menge Σ von *Terminalsymbolen*, einer endlichen Menge P von *Produktionen* und einem *Startsymbol* $S \in N$. Das gesamte *Vokabular* V der Grammatik setzt sich zusammen als $V = \Sigma \cup N$.

Die Produktionen, auch *(Ersetzungs-)Regeln* genannt, haben die Gestalt $A \to \alpha$, mit einer *linken Seite* $A \in N$ und einer *rechten Seite* $\alpha \in V^*$. α wird *Alternative* für A genannt.

Die *Größe* $|G|$ der Grammatik G ist definiert durch

$$|G| := \sum_{A \to \alpha \in P} lg(A\alpha),$$

wobei $lg(A\alpha)$ die Länge des Strings $A\alpha$ angibt. ◇

Solch eine Grammatik erzeugt Wörter, indem durch sukzessive Anwendung der Regeln, beginnend vom Startsymbol S, die Variablen durch rechte Regelseiten ersetzt werden. Bei einer kontextfreien Grammatik darf dabei das zu ersetzende Nichtterminalsymbol stets nur allein auf der linken Seite einer Regel stehen. Bei einer kontextsensitiven Grammatik hingegen sind auch Regeln erlaubt, in denen ein zu ersetzendes Nichtterminalsymbol A einen *Kontext* hat, d. h. die Auswahl der anwendbaren Regeln hängt auch von den Symbolen ab, die A in einer Satzform umgeben. In der folgenden Definition wird dieser Produktionsprozeß formalisiert:

Definition 2 (Satzform, Ableitung): Eine Folge $\alpha \in V^*$ von Symbolen heißt *Satzform*. Ein *Ableitungsschritt* $\alpha \Rightarrow \beta$ überführt eine Satzform α durch die Anwendung einer Regel aus P in eine Satzform β. Eine Folge von Ableitungsschritten heißt *Ableitung* und wird mit $\alpha \overset{*}{\Rightarrow} \beta$ notiert. In einer *Rechtsableitung* wird bei jedem Schritt eine Produktion auf die am weitesten rechts stehende Variable einer Satzform angewandt. Eine Satzform innerhalb einer in S beginnenden Rechtsableitung heißt *Rechtssatzform*. Die Begriffe *Linksableitung* und *Linkssatzform* sind analog definiert. ◇

Die Menge $L(G) = \{w \mid w \in \Sigma^* \wedge S \overset{*}{\Rightarrow} w\}$ aller Wörter, die aus dem Startsymbol der kontextfreien Grammatik G abgeleitet werden können, heißt die von G erzeugte *kontextfreie Sprache*.

Eine kontextfreie Grammatik heißt *mehrdeutig*, falls eine Zeichenfolge $x \in L(G)$ existiert, so dass es zwei verschiedene Rechtsableitungen von x aus dem Startsymbol S gibt. Eine kontextfreie Grammatik $G = (N, \Sigma, P, S)$ heißt *reduziert*, falls $P = \emptyset$ oder es für jedes $A \in V$ Strings $\alpha, \beta \in V^*, w \in \Sigma^*$ gibt, so dass $S \stackrel{*}{\Rightarrow} \alpha A \beta \stackrel{*}{\Rightarrow} w$.

Wenn nicht anders angegeben, sind im Folgenden alle Ableitungen Rechtsableitungen.

Für eine kontextfreie Grammatik $G = (N, \Sigma, P, S)$, eine natürliche Zahl k und $\alpha \in V^*$ enthält die Menge $FIRST_k(\alpha)$ alle Terminalstrings der Länge $\leq k$ und alle Präfixe der Länge k von Terminalstrings, die aus α in G abgeleitet werden können. Formal:

$$FIRST_k(\alpha) = \left\{ x \in \Sigma^* \mid \alpha \stackrel{*}{\Rightarrow} xy, (y = \varepsilon \wedge |x| \leq k) \vee (y \in \Sigma^* \text{ mit } |x| = k) \right\}.$$

Eine Teilmenge der kontextfreien Grammatiken, die im Folgenden betrachtete Klasse der *$LR(k)$-Grammatiken*, ist wie folgt definiert:

Definition 3 ($LR(k)$-Grammatik): Sei $k \geq 0$ eine natürliche Zahl. Eine reduzierte kontextfreie Grammatik $G = (N, \Sigma, P, S)$ ist $LR(k)$, wenn aus den drei Eigenschaften

1. $S \stackrel{*}{\Rightarrow} \alpha A w \Rightarrow \alpha \beta w$,
2. $S \stackrel{*}{\Rightarrow} \gamma B x \Rightarrow \alpha \beta y$, und
3. $FIRST_k(w) = FIRST_k(y)$

folgt: $\alpha = \gamma,\ A = B,\ x = y$. ◇

Die Bedeutung ist folgende: Die $LR(k)$-Eigenschaft garantiert, dass zwei Satzformen, die im Hinblick auf das bisher gelesene Präfix $\alpha\beta$ der Eingabe und den *Look-ahead*, d. h. die ersten k Zeichen des ungelesenen Suffixes der Eingabe ($FIRST_k(w)$ und $FIRST_k(y)$), identisch erscheinen, es auch tatsächlich sind. Ein Parser für G kann die Entscheidung über seinen nächsten Schritt also allein anhand dieser Informationen fällen.

2.1 Der kanonische $LR(k)$-Parser

Bei der Konstruktion eines Parsers für eine $LR(k)$-Grammatik G sind folgende Notationen hilfreich: Sei $S \stackrel{*}{\Rightarrow} \alpha A w \Rightarrow \alpha \beta w$ eine Rechtsableitung in G. Ein Präfix γ von $\alpha\beta$ heißt *lebensfähiges Präfix* von G.

Definition 4 (Item): Ein *Item* ist eine Produktion, zusammen mit einer Position in ihrer rechten Seite. Das heißt, für $p = X \to X_1 X_2 \dots X_{n_p} \in P$ ist das Paar $[p, i], 0 \leq i \leq n_p$, ein Item. Eine andere Darstellung dieses Items ist

$$[X \to X_1 X_2 \dots X_i \cdot X_{i+1} \dots X_{n_p}].$$ ◇

Der Punkt nach dem i-ten Symbol auf der rechten Regelseite zeigt dabei in einem Parse-Vorgang an, bis zu welcher Stelle die zugehörige rechte Regelseite schon eingelesen wurde. Das einzige mögliche Item für eine Regel $p = X \to \varepsilon$ wird $[X \to \cdot]$ geschrieben. Um die $LR(k)$-Eigenschaft aus Definition 3 nutzen zu können, muss jedes Item noch um die Information ergänzt werden, wie der Rest der Eingabe beschaffen sein muss, damit eine weitere Betrachtung dieses Items zu einem erfolgreichen Parsevorgang führen kann. Dies führt zur Definition des $LR(k)$-*Items*:

Definition 5 ($LR(k)$-Item): Ein $LR(k)$-*Item* ist ein Item, zusammen mit einem Terminalstring der Länge $\leq k$. Formal ist $[A \to \beta_1 \cdot \beta_2, u]$ mit $A \to \beta_1\beta_2 \in P$ und $u \in \Sigma^{\leq k}$ ein $LR(k)$-*Item*. Ein $LR(k)$-Item $[A \to \beta_1 \cdot \beta_2, u]$ heißt *gültig* für $\alpha\beta_1 \in V^*$, falls es eine Ableitung $S \overset{*}{\Rightarrow} \alpha A w \Rightarrow \alpha\beta_1\beta_2 w$ mit $u \in FIRST_k(\beta_2 w)$ gibt. Insbesondere kann ein $LR(k)$-Item nur für ein lebensfähiges Präfix von G gültig sein. ◇

Als Grundlage für die kanonische Konstruktion eines $LR(k)$-Parsers dient ein deterministischer Keller-Transducer:

Definition 6 (Kellerautomat): Ein *Kellerautomat* M ist ein Sieben-Tupel $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$.

- Q ist eine endliche, nichtleere Menge von *Zuständen*,
- Σ ist eine endliche, nichtleere Menge von *Eingabesymbolen*,
- Γ ist eine endliche, nichtleere Menge von *Kellersymbolen*,
- $q_0 \in Q$ ist der *Startzustand*,
- $Z_0 \in \Gamma$ ist das *Kellerstartsymbol*,
- $F \subseteq Q$ ist die Menge der *Endzustände*,
- und δ ist eine Abbildung von $Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma$ auf endliche Teilmengen von $Q \times \Gamma^*$.

Da δ auf Teilmengen von $Q \times \Gamma^*$ abbildet, ist ein Kellerautomat im Allgemeinen nichtdeterministisch. Ein Kellerautomat ist *deterministisch*, wenn für jedes $q \in Q$ und $Z \in \Gamma$ entweder $\delta(q, a, Z)$ höchstens ein Element für jedes $a \in \Sigma$ enthält und $\delta(q, \varepsilon, Z) = \emptyset$, oder falls $\delta(q, a, Z) = \emptyset$ für alle $a \in \Sigma$, und $\delta(q, \varepsilon, Z)$ enthält höchstens ein Element.

Ein *deterministischer Keller-Transducer* ist ein deterministischer Kellerautomat mit der zusätzlichen Fähigkeit, eine Ausgabe zu erzeugen. Formal ist ein deterministischer Keller-Transducer ein Acht-Tupel $(Q, \Sigma, \Gamma, \Delta, \delta, q_0, Z_0, F)$, wobei alle Symbole dieselbe Bedeutung haben wie für einen Kellerautomaten, außer, dass Δ ein endliches *Ausgabealphabet* ist und δ nun eine Abbildung

$$\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \mapsto Q \times \Gamma^* \times \Delta^*.$$

$\diamond$

Der kanonische LR-Parser ist ein sogenannter *Shift-Reduce-Parser*. Dabei handelt es sich um einen deterministischen Keller-Transducer, der die Eingabe von links nach rechts einliest und dabei rückwärts eine Rechtsableitung erzeugt. Als Zustandsmenge dient dem Automaten die Menge aller $LR(k)$-Items. Im Folgenden wird die Funktionsweise dieses Automaten informell beschrieben. Für eine Rechtsableitung $S \Rightarrow \alpha_o \Rightarrow \alpha_1 \Rightarrow \ldots \Rightarrow \alpha_{m-1} \Rightarrow \alpha_m = x$ der Zeichenfolge x aus dem Startsymbol S beginnt der Shift-Reduce-Parser mit der Rechtssatzform $\alpha_m := x$ als Eingabe und erzeugt nacheinander die Rechtssatzformen $\alpha_{m-1}, \alpha_{m-2}, \ldots, \alpha_1, \alpha_0$ und schließlich S. Die aktuelle Rechtssatzform ist stets die Konkatenation des Kellerinhaltes von unten nach oben mit dem noch ungelesenen Suffix der Eingabe. Zu Beginn ist der Keller leer. Sei y der unberührte Teil der Eingabe und $\alpha_i = \gamma y$ die aktuelle Rechtssatzform. Dann ist γ der aktuelle Inhalt des Kellers, wobei das letzte Symbol von γ ganz oben im Keller steht. Aus α_i soll nun die Rechtssatzform α_{i-1} konstruiert werden.

Wenn $\alpha_i = \gamma_1\gamma_2 y$ und $\alpha_{i-1} = \gamma_1 A y$, so befindet sich zur Zeit die Alternative γ_2 für die Variable A ganz oben auf dem Keller. Falls $\alpha_i = \gamma_1\gamma_2 y_1 y_2$ und $\alpha_{i-1} = \gamma_1 A y_2$, dann ist ein Teil der Alternative für A noch Präfix der unberührten Eingabe y. Der Parser sorgt nun dafür, dass die Alternative für A vollständig zuoberst auf dem Keller liegt. Erst wenn dies der Fall ist, wird diese Alternative auf dem Keller durch die Variable A ersetzt. Dazu benutzt der Shift-Reduce-Parser die folgenden, namensgebenden Operationen:

Shift: Das nächste Eingabesymbol wird gelesen und zuoberst auf den Keller gelegt.

Reduce: Der Parser stellt fest, dass eine Alternative von A vollständig zuoberst auf dem Keller liegt und ersetzt diese Alternative durch A. Es findet also eine Reduktion des Kellers statt.

In jedem Schritt kann ein Shift-Reduce-Parser jede dieser beiden Operationen ausführen. Da die Konstruktion auf einem nichtdeterministischen Automaten basiert, sind Shift-Reduce-Parser im Allgemeinen ebenfalls nichtdeterministisch. Die folgenden Erläuterungen zeigen, dass $LR(k)$-Grammatiken es ermöglichen, den Parser deterministisch zu machen.

Sei γy die aktuelle Rechtssatzform, mit γ der aktuelle Kellerinhalt und y das unberührte Suffix der Eingabe. Sei $u := FIRST_k(y)$ der aktuelle *Lookahead*. Sei $[A \to \beta_1 \cdot \beta_2, v]$ ein $LR(k)$-Item, das für γ gültig ist. Dann ist β_1 ein Suffix von γ. Falls $\beta_2 = \varepsilon$, so ist $v = u$, und das $LR(k)$-Item $[A \to \beta_1 \cdot, u]$ korrespondiert zu einer Reduktion, die der Parser durchführen kann. Falls $\beta_2 \in \Sigma V^*$ und $u \in FIRST_k(\beta_2 v)$, so korrespondiert das $LR(k)$-Item $[A \to \beta_1 \cdot \beta_2, v]$ zu einem Leseschritt, den der Shift-Reduce-Parser durchführen kann. Der folgende Satz besagt, dass die Menge aller $LR(k)$-Items, die für einen Kellerinhalt γ gültig sind, zu höchstens einem Schritt korrespondiert, den der Shift-Reduce-Parser durchführen kann.

Satz 1: *Sei $k \geq 0$ eine natürliche Zahl und $G = (V, \Sigma, P, S)$ eine kontextfreie Grammatik. G ist $LR(k)$ genau dann, wenn für alle $u \in \Sigma^{\leq k}$ und alle $\alpha\beta \in V^*$ die folgende Eigenschaft erfüllt ist: Wenn ein $LR(k)$-Item $[A \to \beta \cdot, u]$ gültig für $\alpha\beta$ ist, so existiert kein anderes $LR(k)$-Item $[C \to \beta_1 \cdot \beta_2, v]$ mit $\beta_2 \in \Sigma V^* \cup \{\varepsilon\}$ und $u \in FIRST_k(\beta_2 v)$, das ebenfalls für $\alpha\beta$ gültig ist.* □

Der Beweis kann in [Blu10a] nachgelesen werden.

Für den Shift-Reduce-Parser ermöglicht dieser Satz eine eindeutige Arbeitsanweisung: Wenn während der Konstruktion der Rechtsableitung ein $LR(k)$-Item der Form $[A \to \beta \cdot, u]$ gültig für den aktuellen Kellerinhalt γ ist und u der aktuelle Lookahead ist, so befindet sich β zuoberst auf dem Keller. Somit ist die Reduktion anhand der Produktion $A \to \beta$ der einzig mögliche Schritt des Parsers. Falls ein $LR(k)$-Item $[C \to \beta_1 \cdot \beta_2, v]$ mit $\beta_2 \in \Sigma V^*$ für den aktuellen Kellerinhalt gültig ist und der aktuelle Lookahead u in $FIRST_k(\beta_2 v)$ liegt, so ist das Lesen des ersten Symbols von u der einzig durchführbare Schritt für den Shift-Reduce-Parser.

Ein Hindernis für eine größere Verbreitung von $LR(k)$-Parsern ist die Größe ihrer Zustandsmenge Q. Für eine Grammatik G gibt es $|G|$ viele Items und maximal $|\Sigma|^k \leq |G|^k$ mögliche Lookahead-Strings. Somit liegt die Obergrenze für die Anzahl

der $LR(k)$-Items für G bei $|G|^{k+1}$. Bei der Konstruktion eines deterministischen Shift-Reduce-Parsers repräsentiert jeder Zustand eine Menge von $LR(k)$-Items. Somit ist eine obere Schranke für die Größe der Zustandsmenge eines $LR(k)$-Parsers durch $2^{|G|^{k+1}}$ gegeben. Ukkonen zeigt zwar in [Ukk85], dass diese obere Schranke für viele relevante Grammatiken deutlich zu groß ist, stellt aber zugleich Grammatiken vor, für die eine praktische Implementierung des kanonischen $LR(k)$-Parsers dennoch zu viele Zustände hätte. Eine der dort erwähnten Grammatiken fand bei den praktischen Vergleichstests Anwendung, die in Kapitel 5 beschrieben sind. Die große Zustandsmenge von $LR(k)$-Parsern ist auch ein Grund dafür, das in der Praxis oft Generatoren verwendet werden, die Parser für die Teilklasse der $LALR(1)$-Grammatiken erzeugen. Ausführliche Informationen zum Aufbau von $LALR(1)$-Parsern sind in [GJ08] zu finden.

2.2 Der erweiterte $LR(k)$-Parser

Aufgrund der möglicherweise exponentiellen Größe eines kanonischen $LR(k)$-Parsers erscheint es sinnvoll, nach alternativen Implementierungsmöglichkeiten zu suchen. Blum stellt in [Blu10a] ein Parsermodell für $LR(k)$-Grammatiken vor, bei dem ein Parser für eine Grammatik G einen Graphen der Größe $O(|G|n)$ verwaltet, wobei n die Länge der Eingabe angibt. Solch ein *erweiterter $LR(k)$-Parser* hat, je nach Wahl der Datenstrukturen, eine Größe in $O(|G| + \#LA|N|2^k \log k)$ oder in $O(|G| + \#LA \cdot |N| \cdot k^2)$ und eine Laufzeit in $O(ld(input) + k|G|n)$. Dabei ist $\#LA$ die Anzahl möglicher Lookaheads im Bezug auf G, und $ld(input)$ gibt die Länge der Ableitung der Eingabe an. Im Folgenden wird die Konstruktion eines erweiterten $LR(k)$-Parsers P_G für eine Grammatik G erläutert.

Die Grundidee der erweiterten $LR(k)$-Parser ist es, nicht nur die einzig mögliche Rechtsableitung zu betrachten, sondern parallel die Menge aller Ableitungen, die im aktuellen Zustand des Parsers zu einer akzeptierenden Berechnung führen können. Um möglichst früh während des Parsevorgangs Ableitungen auszuschließen, die nicht mehr der korrekten Rechtsableitung der Eingabe entsprechen können, bietet es sich an, alle möglichen Linksableitungen zu simulieren. Diese haben den Vorteil, das zu einem möglichst frühen Zeitpunkt Terminalsymbole am linken Ende einer Satzform erzeugt werden. Dadurch kann früh verglichen werden, ob für eine Satzform $y\gamma, y \in \Sigma^*, \gamma \in NV^* \cup \{\varepsilon\}$, das Präfix y zugleich Präfix der Eingabe ist. Nur dann kann die aktuell betrachtete Satzform Teil der gesuchten Ableitung sein.

Für eine gegebene kontextfreie Grammatik $G = (N, \Sigma, P, S)$ wird der Kellerautomat M_G mit $L(M_G) = L(G)$ wie folgt konstruiert, damit er eine Linksableitung erzeugt: Als Zustandsmenge dient die Menge aller Items, ergänzt um zwei zusätzliche Items $[S' \to \cdot S]$ und $[S' \to S\cdot]$, die den Beginn und das Ende eines Parserlaufs signalisieren. Dem Automaten steht wieder ein Keller als Zwischenspeicher zur Verfügung, wobei das Kelleralphabet diesmal aus der Zustandsmenge und dem Symbol $\perp$ für den leeren Keller besteht. Bezeichne für ein Item I die Produktion, aus der das Item gebildet wurde, mit $p(I)$. Sei $q = I_l$ der aktuelle Zustand und $I_{l-1} \dots I_1 \perp$ der aktuelle Kellerinhalt des Automaten. Die Zustandsübergänge werden so definiert, dass die Folge von Produktionen $p(I_1), p(I_2), \dots, p(I_l)$ genau der Reihenfolge entspricht, in der die Produktionen der Grammatik in der aktuell simulierten Linksableitung angewendet wurden.

Definition 7 (Ableitungsautomat M_G): Für eine Produktion $p \in P$ sei n_p die Länge der rechten Seite von p. Sei $H_G = \{[p, i] \mid p \in P, 0 \leq i \leq n_p\}$ die Menge aller Items von G.

Dann ist der Automat $M_G = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ definiert durch

$$
\begin{aligned}
Q &= H_G \cup \{[S' \to \cdot S], [S' \to S\cdot]\}, \\
q_0 &= [S' \to \cdot S],\ F = \{[S' \to S\cdot]\}, \\
\Gamma &= Q \cup \{\perp\},\ Z_0 = \perp, \text{ und} \\
\delta &: Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \mapsto 2^{Q \times \Gamma^*}.
\end{aligned}
$$

Bei der Definition der nichtdeterministischen Übergangsfunktion δ werden drei Typen von Schritten unterschieden:

(E) Expansion:
$\delta([X \to \beta \cdot A\gamma], \varepsilon, Z) = \{([A \to \cdot\alpha], [X \to \beta \cdot A\gamma]Z) \mid A \to \alpha \in P\}$.
Die am weitesten links stehende Variable in der Linkssatzform wird durch eine ihrer Alternativen ersetzt. Der Keller wird expandiert.

(C) Lesen:
$\delta([X \to \phi \cdot a\psi], a, Z) = \{([X \to \phi a \cdot \psi], Z)\}$.
Das nächste Symbol der Eingabe wird gelesen.

(R) Reduktion:
$\delta([X \to \alpha\cdot], \varepsilon, [W \to \mu \cdot X\nu]) = \{([W \to \mu X \cdot \nu], \varepsilon)\}$.

Die Alternative α von X wurde vollständig hergeleitet. Somit kann der Punkt hinter X geschoben werden, und das entsprechende Item kann vom Keller genommen werden, um zum neuen Zustand zu werden. Der Keller wird also reduziert. $\diamond$

Wenn der aktuelle Zustand von M_G stets zuoberst auf den Keller gelegt wird, genügt es für eine deterministische Simulation des Automaten, alle möglichen Kellerinhalte parallel zu simulieren. Da jeder Kellerinhalt aus einer Teilmenge von H_G besteht, kann es allerdings eine exponentielle Anzahl unterschiedlicher Kellerinhalte geben. Eine direkte Simulation aller Kellerinhalte erscheint somit ineffizient. Da die Grammatik G und somit auch der Kellerautomat M_G eine feste Größe haben, kann es aber nur eine konstante Anzahl verschiedener Elemente geben, die zeitgleich auf allen Kellern zuoberst liegen. Es gibt also nur eine konstante Anzahl von Möglichkeiten, die exponentiell vielen Kellerinhalte zu modifizieren.

Alle Kellerinhalte werden parallel durch einen gerichteten Graphen $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ simuliert. Dabei wird jeder Knoten mit seinem Item identifiziert. Der Graph enthält genau einen Knoten mit Eingangsgrad 0, den *Startknoten* $[S' \to \cdot S]$. Die Knoten mit Ausgangsgrad 0 heißen *Endknoten*. Zu jedem Zeitpunkt besteht eine Bijektion zwischen den Pfaden vom Startknoten zu einem Endknoten und den möglichen Kellerinhalten.

Der Simulationsalgorithmus ist in Phasen unterteilt, während derer alle Endknoten simultan behandelt werden. Ein Endknoten der Form $[A \to \alpha_1 \cdot \alpha_2]$ mit $\alpha_2 \in NV^*$ heißt *expandierbar*, mit $\alpha_2 \in \Sigma V^*$ *lesbar* und mit $\alpha_2 = \varepsilon$ *reduzierbar*. Da für verschiedene Endknoten unterschiedliche Aktionen angebracht sein können, finden folgende Synchronisationsregeln Anwendung:

1. Solange es expandierbare Endknoten gibt, führe für diese Knoten eine Expansion durch.

2. Wenn alle Endknoten reduzierbar oder lesbar sind, dann führe für alle Endknoten mit $\alpha_2 = \varepsilon$ eine Reduktion durch.

3. Sind alle Endknoten lesbar, dann führe für sie einen Leseschritt durch.

Dabei wechseln sich solange Folgen von Expansionen (*Expansionsschritte*) mit Folgen von Reduktionen (*Reduktionsschritte*) ab, bis nur noch lesbare Endknoten im Graph verbleiben. Während jeder Phase wird genau ein Eingabesymbol gelesen,

so dass es genau n Phasen gibt. Da es zyklische Produktionen in G geben kann, kann auch der Graph $\mathcal{G}$ Zyklen enthalten. Dabei heißt eine Produktion $A \to \alpha$ mit $A \in N, \alpha \in V^*$ *zyklisch*, wenn A wiederum aus α herleitbar ist, wenn also gilt $\alpha \stackrel{*}{\Rightarrow} \beta_1 A \beta_2$ für $\beta_1, \beta_2 \in V^*$.

Um nun den erweiterten $LR(k)$-Parser P_G zu erhalten, sollen bei der Simulation von M_G Eigenschaften von $LR(k)$-Grammatiken berücksichtigt werden. Hierzu ist erneut Satz 1 von zentraler Bedeutung. Satz 1 trifft eine Aussage über gültige $LR(k)$-Items für ein lebensfähiges Präfix von G. Somit muss P_G bei der Simulation aller Linksableitungen nur solche Satzformen betrachten, die mit einem lebensfähigen Präfix von G beginnen. Im Graphen $\mathcal{G}$, den der Simulationsalgorithmus verwaltet, müssen also die maximalen lebensfähigen Präfixe gefunden und untersucht werden.

Sei $[A \to \alpha_1 \cdot \alpha_2]$ ein Item. Dann heißt der Teil α_1 links vom Punkt die *linke Seite* dieses Items.

Definition 8 (Maximales lebensfähiges Präfix): Sei $\mathcal{P}$ ein Pfad vom Startknoten zu einem Endknoten in $\mathcal{G}$. Dann ergibt sich das *maximale lebensfähige Präfix* $pref(\mathcal{P})$ *bezüglich* $\mathcal{P}$ durch die Konkatenation der linken Seiten der Items vom Startknoten aus bis zu den Endknoten von $\mathcal{P}$. Formal: Für

$$\mathcal{P} = [S' \to \cdot S], [S \to \alpha_1 \cdot A_2 \beta_1], [A_2 \to \alpha_2 \cdot A_3 \beta_2], \ldots, [A_t \to \alpha_t \cdot \beta_t]$$

ist das maximale lebensfähige Präfix bezüglich $\mathcal{P}$ definiert durch

$$pref(\mathcal{P}) = \alpha_1 \alpha_2 \ldots \alpha_t.$$

◇

Als nächstes folgt eine Charakterisierung gültiger $LR(k)$-Items für einen Pfad $\mathcal{P}$, dessen Endknoten reduzierbar oder lesbar ist. Sei $[B \to \alpha \cdot C\beta], C \in N, \beta \in V^*$ ein Item. Dann heißt β die *rechte Seite* des Items. Die *rechte Seite* eines Items $[B \to \alpha \cdot a\beta], a \in \Sigma, \beta \in V^*$, ist $a\beta$.

Definition 9 (Relevantes Suffix): Das *relevante Suffix* $suf(\mathcal{P})$ bezüglich eines Pfades $\mathcal{P}$ in $\mathcal{G}$ ist die Konkatenation der rechten Seiten vom Endknoten von $\mathcal{P}$ aus zum Startknoten, d. h.:

$$suf(\mathcal{P}) = \begin{cases} \beta_{t-1}\beta_{t-2}\ldots\beta_1, & \text{falls } \beta_t = \varepsilon, \\ a\beta'_{t-1}\beta_{t-2}\ldots,\beta_1, & \text{falls } \beta_t = a\beta'_t. \end{cases}$$

◇

Das relevante Suffix für einen Pfad $\mathcal{P}$ mit Endknoten w gibt Aufschluß darüber, wie der ungelesene Rest der Eingabe beschaffen sein muss, damit das zu w korrespondierende Item zu einem gültigen $LR(k)$-Item ergänzt werden kann. Dies wird in folgender Gültigkeitsdefinition zum Ausdruck gebracht:

Definition 10 (Für einen Pfad gültiges $LR(k)$-Item): Sei $\mathcal{P}$ ein Pfad im Graphen $\mathcal{G}$ und u der aktuelle Lookahead. Das $LR(k)$-Item $[A_t \to \alpha_t \cdot \beta_t, u]$ heißt *gültig für den Pfad* $\mathcal{P}$ genau dann, wenn $u \in FIRST_k(suf(\mathcal{P}))$. ◇

In Lemma 2 in [Blu10a] wird bewiesen, dass für alle Pfade vom Startknoten zu einem Endknoten in $\mathcal{G}$ das maximale lebensfähige Präfix identisch ist. Somit kann Satz 1 auf M_G angewendet werden, was zu einem neuen Kellerautomaten $LR(k)$-M_G führt. Dabei wird gefordert, dass während der Simulation des $LR(k)$-M_G folgendes Invariantenpaar erfüllt ist:

Definition 11 (Invarianten für $LR(k)$-M_G): Für die Endknoten des Graphen $\mathcal{G}$, den der Parser verwaltet, gilt stets:

1. Unmittelbar vor einem Expansionsschritt haben alle Endknoten von $\mathcal{G}$ die Form $[A \to \alpha \cdot B\beta]$ oder $[A \to \alpha \cdot a\beta]$, wobei $\alpha, \beta \in V^*, B \in N$, und $a \in \Sigma$ ist das nächste ungelesene Symbol der Eingabe. Alle Endknoten in $\mathcal{G}$ sind also expandierbar oder lesbar.

2. Unmittelbar vor einem Reduktions-/Leseschritt haben alle Endknoten von $\mathcal{G}$ die Form $[A \to \cdot]$ oder $[A \to \alpha \cdot a\beta]$, wobei $\alpha, \beta \in V^*$, und $a \in \Sigma$ ist das nächste ungelesene Symbol der Eingabe. Der Graph $\mathcal{G}$ enthält also ausschließlich reduzierbare und lesbare Endknoten. ◇

Die Details der Simulation sind im folgenden Kapitel zu finden.

3 Ausarbeitung der Algorithmen

In diesem Kapitel werden Algorithmen formuliert, in denen die Konzepte aus Abschnitt 2.2 umgesetzt werden. Als Grundlage dient der in [Blu10a] vorgestellte Algorithmus SIMULATION(M_G), der eine Simulation des nichtdeterministischen Ableitungsautomaten M_G aus Definition 7 beschreibt. Dieser Grundalgorithmus wurde für die vorliegende Arbeit in mehrere Teilalgorithmen untergliedert und zu einem Parser für $LR(k)$-Grammatiken ergänzt. Dazu wurden die Erkenntnisse eingearbeitet, die Blum in den Abschnitten 6 und 7 seiner Arbeit entwickelt und die in Abschnitt 2.2 beschrieben wurden. Nach einer kurzen Beschreibung des Rahmenalgorithmus werden in diesem Kapitel die Methoden zur Durchführung der Expansionen, Reduktionen und Leseschritte herausgearbeitet. Besonderes Augenmerk wird dabei auf die bei Blum nur kurz erwähnte Behandlung der Zyklen bei der Suche nach gültigen Items gelegt. Anschließend folgen eine Erläuterung zur Berechnung der vielfach verwendeten $FIRST_k$-Mengen und eine Laufzeitanalyse des entwickelten Parsers.

Im Folgenden bezeichnet $\mathcal{G}$ stets den aktuellen Zustand des Graphen, der von den Algorithmen manipuliert wird. Ein Knoten in $\mathcal{G}$ und das durch ihn repräsentierte Item werden synonym verwendet. Zur leichteren Unterscheidung erhalten einige Knoten als Index die Nummer derjenigen Phase, in welcher der Knoten erzeugt oder zuletzt verändert wurde. Wo dies nicht nötig ist, kann der Index auch fehlen. Außerdem gilt $[A \to \cdot\varepsilon] = [A \to \varepsilon\cdot]$. Die Eingabe des Parsers wird mit $x = a_1 a_2 \dots a_n$ bezeichnet, der aktuelle Lookahead mit u.

Der Algorithmus 3.1 enthält das Gerüst für den generierten Parser und orientiert sich an der Phasenunterteilung in [Blu10a]. Während der Ausführung des Algorithmus und aller untergeordneten Algorithmen wird das Invariantenpaar aus Definition 11 aufrecht erhalten, das Blum in Abschnitt 6 seiner Arbeit für die Simulation des Automaten $LR(k)$-M_G fordert. Zu Beginn jeder Phase führt der Parser zunächst alle möglichen Expansionen durch, bis der Graph nur noch reduzierbare und lesbare Endknoten enthält (EXPAND, Algorithmus 3.2). Anschließend wird ein Reduktions-/Leseschritt eingeleitet (REDUCE-READ, Algorithmus 3.3). Falls die dort ausgeführ-

Algorithmus 3.1 SIMULATION($LR(k)$-M_G)

Eingabe: Kontextfreie Grammatik $G = (V, \Sigma, P, S)$, Eingabestring $x = a_1 a_2 \dots a_n$.
Ausgabe: „accept“, falls $x \in L(G)$; „error“, sonst.

1: Initialisiere den Graphen $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ mit $\mathcal{V} := \{[S' \to \cdot\ S]_0\}$ und $\mathcal{E} := \emptyset$;
2: $H := \{[S' \to \cdot S]_0\}$; $K := \emptyset$;
   ($*$ Expandierbare und bereits expandierte Knoten. $*$)
3: $C := \emptyset$; $R := \emptyset$; ($*$ Lesbare und reduzierbare Endknoten. $*$)
4: $Pr := \emptyset$; ($*$ Zu untersuchende Vorgängerknoten. $*$)
5: $i := 0$;
6: **while** $i \leq n$ **do**
7: EXPAND; ($*$ Algorithmus 3.2 $*$)
8: REDUCE-READ; ($*$ Algorithmus 3.3 $*$)
9: **end while**
10: **if** $[S' \to S\cdot]_n \in \mathcal{V}$ **then**
11: *output* := „accept“
12: **else**
13: *output* := „error“
14: **end if**

te Aktion ein Leseschritt ist, wird darin der Phasenzähler i um 1 erhöht und die Phase somit beendet. Pro Phase wird also genau ein Eingabesymbol verbraucht, die **while**-Schleife in den Zeilen 5–8 von Algorithmus 3.1 terminiert daher, sobald i den Wert $n+1$ erreicht. Findet eine Reduktion statt, können neue expandierbare Knoten entstehen. In diesem Fall wird die **while**-Schleife iteriert, ohne die Phase zu beenden, d. h. ohne i zu inkrementieren. Im Unterschied zum Algorithmus SIMULATION(M_G) in [Blu10a] kann die Hauptschleife des Algorithmus also während einer Phase mehrmals durchlaufen werden. Dies stellt aber algorithmisch keinen Unterschied dar, sondern dient nur einer besseren Gliederung des Verfahrens in Teilalgorithmen.

Folgende Datenstrukturen verwaltet der Algorithmus: H ist die Menge der zu expandierenden Endknoten. K enthält Itemknoten, die in der aktuellen Phase bereits expandiert wurden. Dadurch werden Endlosschleifen bspw. beim Expandieren rekursiver Regeln vermieden. Knoten in C und R sind lesbar bzw. reduzierbar, und Pr nimmt Knoten auf, die untersucht werden müssen, weil ihre Vorgängerknoten reduziert wurden.

Im folgenden Abschnitt 3.1 wird die Durchführung der Expansionsschritte be-

schrieben, und in Abschnitt 3.2 wird die Durchführung der Reduktionen ausführlicher erläutert. Die Probleme, die durch das Vorhandensein von Zyklen im Graphen entstehen können, werden in Abschnitt 3.5 näher untersucht. Abschnitt 3.6 schließlich beschreibt die Berechnung der $FIRST_k$-Mengen, mit deren Hilfe der Algorithmus während eines Reduktions-/Leseschrittes seine Entscheidungen trifft.

3.1 Durchführung der Expansionen

Die Durchführung eines Expansionsschrittes unter Berücksichtigung der $LR(k)$-Eigenschaft der Grammatik wird in Algorithmus 3.2 beschrieben. Dabei wird für einen zu expandierenden Knoten $[A \to \alpha_1 \cdot B\alpha_2]_i \in H$ mit $B \to \beta \in P$ ein Knoten $[B \to \cdot\beta]_i$ für die Alternative β nur dann zu $\mathcal{G}$ hinzugefügt, wenn das neue Item Teil einer akzeptierenden Berechnung sein kann. Dazu darf β aus dem leeren Wort bestehen oder mit einem Nichtterminalsymbol oder dem nächsten zu lesenden Eingabesymbol a_{i+1} beginnen, nicht aber mit einem Terminalsymbol $a \neq a_{i+1}$. Sollte für einen expandierbaren Endknoten keine solche Alternative existieren, kann der Pfad zu diesem Knoten nicht mehr zu einer akzeptierenden Berechnung korrespondieren. In diesem Fall findet eine *Graphbereinigung* statt, bei der der betreffende Endknoten aus dem Graphen entfernt wird. Solange es noch Knoten im Graph gibt, deren Nachfolger alle bei der Graphbereinigung entfernt wurden, werden auch diese entfernt.

Möglicherweise hat in der aktuellen Phase allerdings schon eine Expansion anhand des Nichtterminalsymbols B stattgefunden, d. h. es existieren schon Knoten $[B \to \cdot\beta_j]_i$, $1 \leq j \leq q$, für die q Alternativen von B. Dann müssen nur noch die q Kanten $([A \to \alpha_1 \cdot B\alpha_2]_i, [B \to \cdot\beta_j]_i)$ zu $\mathcal{G}$ hinzugefügt werden. Dadurch ist die Größe der Knotenmenge beschränkt durch $O(|G|n)$, da jeder mögliche Knoten pro Phase höchstens einmal hinzugefügt werden kann. Durch mehrfache Expansion anhand desselben Nichtterminalsymbols innerhalb einer Phase kann aber die Anzahl der Kanten auf $O(|G|^2 n)$ anwachsen. Daher wird im hier vorgestellten Algorithmus eine Optimierung angewendet, die in Abschnitt 7 von [Blu10a] vorgeschlagen wird. Dazu wird ein neuer Knotentyp für Variablen eingeführt. Auch für diese Knoten wird im Folgenden oft der Knoten und das durch ihn repräsentierte Symbol synonym verwendet. Bei der ersten Expansion eines Endknotens der Form $[A \to \alpha_1 \cdot B\alpha_2]_i$ wird der Endknoten mit einem neuen Knoten B_i verbunden. Von diesem gehen dann die Kanten zu den einzelnen Alternativen ab (Zeilen 9–26). Bei einer weiteren Expansion

Algorithmus 3.2 EXPAND

1: **while** $H \neq \emptyset$ **do**
2: Wähle beliebigen Endknoten $v = [A \to \alpha_1 \cdot B\alpha_2]_i \in H$;
3: $H := H \setminus \{[A \to \alpha_1 \cdot B\alpha_2]_i\}$; ($*$ Entnimm v aus der Menge expandierbarer Items. $*$)
4: $K := K \cup \{[A \to \alpha_1 \cdot B\alpha_2]_i\}$; ($*$ Vermerke den Knoten als expandiert. $*$)
5: $ok := 0$;
6: **if** B wurde in dieser Phase schon expandiert **then**
7: ($*$ Es gibt also bereits einen Nichtterminalknoten B_i in $\mathcal{G}$, der nur noch mit v verbunden werden muss. $*$)
8: $ok := 1$;
9: **else** ($*$ B wurde in dieser Phase noch nicht expandiert $*$)
10: $\mathcal{V} := \mathcal{V} \cup \{B_i\}$;
11: **for** jede Alternative β von B **do**
12: **if** $\beta \in (NV^* \cup \{a_{i+1}\}V^* \cup \{\varepsilon\})$ **then**
13: $\mathcal{V} := \mathcal{V} \cup \{[B \to \cdot\beta]_i\}$;
14: $\mathcal{E} := \mathcal{E} \cup \{(B_i, [B \to \cdot\beta]_i)\}$;
15: **if** $\beta = \varepsilon$ **then** ($*$ Neuer Endknoten ist reduzierbar. $*$)
16: $R := R \cup \{[B \to \cdot\beta]_i\}$;
17: **else if** $\beta = a_{i+1}\beta',\ \beta' \in V^*$ **then** ($*$ Neuer Endknoten ist lesbar. $*$)
18: $C := C \cup \{[B \to \cdot a_{i+1}\beta']\}_i$;
19: **else if** $\beta \in NV^* \wedge [B \to \cdot\beta]_i \notin K$ **then**
20: ($*$ Neuer Endknoten ist expandierbar und wurde in der aktuellen Phase noch nicht bearbeitet. $*$)
21: $H := H \cup \{[B \to \cdot\beta]_i\}$
22: **end if**
23: $ok := 1$;
24: **end if**
25: **end for**
26: **end if**
27: **if** $ok = 1$ **then**
28: $\mathcal{E} := \mathcal{E} \cup \{([A \to \alpha_1 \cdot B\alpha_2]_i, B_i)\}$;
29: **else**
30: Entferne aus $\mathcal{G}$ den Nichtterminalknoten B_i, den Knoten $[A \to \alpha_1 \cdot B\alpha_2]_i$, und führe eine Graphbereinigung durch.
31: **end if**
32: **end while**

eines Endknotens $[A' \to \alpha'_1 \cdot B\alpha'_2]_i$ wird nur noch eine Kante $([A' \to \alpha'_1 \cdot B\alpha'_2]_i, B_i)$ zum Graphen hinzugefügt (Zeilen 6–8 und 27–29). Somit ist auch die Anzahl der Kanten mit $O(|G|n)$ nach oben beschränkt.

Da nur während eines Expansionsschrittes neue Kanten zu Nichtterminalknoten in den Graph eingefügt werden, sind die Zyklen, die durch solche mehrfachen Expansionen eines Nichtterminalsymbols innerhalb einer Phase im Graphen entstehen können, die einzig möglichen Zyklen in $\mathcal{G}$. Da die äußere **while**-Schleife solange iteriert wird, bis alle expandierbaren Endknoten bearbeitet wurden, und in Zeile 18 nur solche lesbaren Endknoten in den Graphen aufgenommen werden, in denen das Symbol hinter dem Punkt mit dem nächsten ungelesenen Eingabesymbol identisch ist, ist die zweite Invariante von Definition 11 nach der Durchführung des Expansionsschrittes und somit vor der Durchführung des nächsten Reduktions-/Leseschrittes erfüllt.

3.2 Durchführung der Reduktions-/Leseschritte

Ein $LR(k)$-konformer Reduktions-/Leseschritt wird gemäß Algorithmus 3.3 durchgeführt. Zunächst wird nach Endknoten der Form $[A \to \varepsilon\cdot]$ oder $[A \to \alpha \cdot a_{i+1}\beta]$ gesucht, zu denen in $\mathcal{G}$ vom Startknoten aus ein Pfad $\mathcal{P}$ existiert, so dass das $LR(k)$-Item $[A \to \varepsilon\cdot, u]$ bzw. $[A \to \alpha \cdot a_{i+1}\beta, u]$ gültig bezüglich $\mathcal{P}$ ist. Solch ein Pfad $\mathcal{P}$ heißt *passend* für den Endknoten $[A \to \varepsilon\cdot]$ bzw. $[A \to \alpha \cdot a_{i+1}\beta]$. Das Verfahren für die Suche nach passenden Pfaden wird in den Abschnitten 3.3 und 3.4 ausformuliert. Anschließend findet, je nach Suchergebnis, ein Leseschritt oder eine Reduktion statt. Drei Fälle sind möglich:

Fall 1: Ist das Ergebnis der Suche eine Menge von lesbaren Endknoten, so können alle reduzierbaren Endknoten nicht mehr zu einer akzeptierenden Berechnung führen und werden aus $\mathcal{G}$ entfernt. Für alle verbliebenen Endknoten wird der Leseschritt durchgeführt. Dazu wird in diesen Knoten der Punkt um eine Stelle nach rechts verschoben. Die modifizierten Endknoten müssen anschließend noch daraufhin untersucht werden, ob sie expandierbar, reduzierbar oder lesbar sind oder zu keiner akzeptierenden Berechnung führen können. Sollten nämlich reduzierbare Endknoten entstanden sein, müssen diese noch vor Ende der Phase bearbeitet werden, damit vor dem ersten Expansionsschritt der nächsten Phase Invariante 1 aus Definition 11 weiterhin erfüllt ist. Diese Untersuchung geschieht in Algorithmus 3.5, der weiter

Algorithmus 3.3 REDUCE-READ

1: Suche in $C \cup R$ eine Menge W von Endknoten, für die ein passender Pfad existiert;
2: **if** $W = \emptyset$ **then**
3: $output$:= „error“;
4: **else if** $W = \{w\}$ mit $w = [A \to \varepsilon\cdot]$ **then**
5: Entferne aus $\mathcal{G}$ alle anderen Endknoten mit einer Graphbereinigung;
6: A := Der direkte Vorgänger von w in $\mathcal{G}$;
7: $\mathcal{E} := \mathcal{E} \setminus \{(A, [A \to \varepsilon\cdot])\}$;
8: $\mathcal{V} := \mathcal{V} \setminus \{[A \to \varepsilon\cdot]\}$;
9: Untersuche die Vorgänger von A mit TREATPREDECESSORS;
(∗ Algorithmus 3.4 ∗)
10: **else** (∗ W besteht aus lesbaren Endknoten der Form $[A \to \alpha \cdot a_{i+1}\beta]$ ∗)
11: Entferne aus $\mathcal{G}$ alle reduzierbaren Endknoten mit einer Graphbereinigung;
12: $i := i + 1$; $K := \emptyset$;
13: Bewege in allen verbliebenen Endknoten den Punkt um eine Stelle nach rechts;
14: Untersuche dabei jeden modifizierten Endknoten mit TESTNEWENDNODE;
(∗ Algorithmus 3.5 ∗)
15: **if** R enthält neue reduzierbare Endknoten **then**
16: Suche in R eine Menge W von Endknoten, für die ein passender Pfad existiert;
17: **if** $W = \{w\}$ mit $w = [A \to \alpha\cdot]$ **then**
18: Entferne aus $\mathcal{G}$ alle anderen Endknoten mit einer Graphbereinigung;
19: A := Der direkte Vorgänger von w in $\mathcal{G}$;
20: $\mathcal{E} := \mathcal{E} \setminus \{(A, w)\}$;
21: $\mathcal{V} := \mathcal{V} \setminus \{w\}$;
22: Untersuche die Vorgänger von A mit TREATPREDECESSORS;
23: **else** (∗ $W = \emptyset$ ∗)
24: Entferne aus $\mathcal{G}$ alle reduzierbaren Endknoten mit einer Graphbereinigung;
25: **end if**
26: **end if**
27: **end if**

unten erläutert wird. Dabei ist es wichtig, den Phasenindex i vor der Untersuchung der modifizierten Endknoten zu inkrementieren, da eventuelle Reduktionsentscheidungen schon auf Basis des Lookaheads der neuen Phase getroffen werden müssen. Andernfalls würde das in der aktuellen Phase verbrauchte Symbol a_{i+1} fälschlicherweise erneut für Reduktions- oder Leseentscheidungen herangezogen werden.

Fall 2: Wenn ein reduzierbarer Endknoten w gefunden wurde, so ist die entsprechende Reduktion der gemäß Satz 1 eindeutig bestimmte nächste Schritt des Parsers. Alle anderen Endknoten werden aus $\mathcal{G}$ entfernt, und die Reduktion wird wie folgt durchgeführt: Sei der Nichtterminalknoten A der direkte Vorgängerknoten von w. Der Knoten w wird aus dem Graphen entfernt, und für jeden Vorgängerknoten v von A müssen zwei Fälle unterschieden werden, die Algorithmus 3.4 behandelt:

a) Der Knoten A hat keine weiteren Nachfolger in $\mathcal{G}$. Dann kann der Punkt in v um eine Position nach rechts verschoben werden.

b) Der Knoten A hat noch andere Nachfolger. Dann kann v sowohl zu Berechnungen korrespondieren, in denen das Nichtterminalsymbol A bereits vollständig abgeleitet wurde, als auch zu solchen, in denen dies noch nicht zutrifft. Daher wird von v eine Kopie v' erzeugt und in v' der Punkt um eine Stelle nach rechts bewegt.

In beiden Situationen muss analog zu Fall 1 der neue Endknoten noch untersucht und gegebenenfalls eine erneute Reduktion durchgeführt werden, damit nach Beendigung des aktuellen Reduktions-/Leseschrittes Invariante 1 erfüllt bleibt. Dabei hat die parallele Behandlung aller neuen reduzierbaren Endknoten ab Zeile 16 von Algorithmus 3.3 und ab Zeile 11 von Algorithmus 3.4 gegenüber einer sofortigen einzelnen Untersuchung nach ihrer Entstehung den Vorteil, dass gemeinsame Teilpfade zu den Endknoten nicht mehrfach besucht werden müssen.

Fall 3: Falls das Suchergebnis leer oder widersprüchlich ist, wird die Analyse der Eingabe mit einer Fehlermeldung abgebrochen. Dies kann zwei Gründe haben:

a) Wenn kein Endknoten gefunden wird, für den ein passender Pfad in $\mathcal{G}$ existiert, kann keiner der vorhandenen Endknoten zu einer akzeptierenden Berechnung führen. In diesem Fall kann die Eingabe x nicht Teil der Sprache sein, die vom Parser erkannt wird.

b) Sollten mehrere reduzierbare Endknoten gefunden werden, für die ein passender Pfad existiert, oder sowohl lesbare als auch reduzierbare Endknoten mit dieser Eigenschaft, liegt ein Fehler vor. In diesem Fall ist die zugrunde liegende Grammatik nicht $LR(k)$ für das gewählte k.

Algorithmus 3.4 TreatPredecessors

Eingabe: Nichtterminalknoten A, dessen Vorgängerknoten zu überprüfen sind.

1: $Pr := \{v \in \mathcal{V} \mid v \text{ ist direkter Vorgänger von } A \text{ in } \mathcal{G}\}$;
2: **if** A hat in $\mathcal{G}$ keine anderen Nachfolger als den soeben reduzierten Knoten **then**
3: Entferne für jeden Knoten $v \in Pr$ die Kante (v, A) aus $\mathcal{G}$;
4: Bewege in jedem Knoten $v \in Pr$ den Punkt um eine Stelle nach rechts und überprüfe v mit TestNewEndnode;
5: $\mathcal{V} := \mathcal{V} \setminus A$;
6: **else** $(* \; A$ hat noch andere Nachfolger $*)$
7: Erzeuge von jedem Knoten $v \in Pr$ eine Kopie v' mit allen in v eingehenden Kanten;
8: Bewege in jeder Kopie v' den Punkt um eine Stelle nach rechts und überprüfe v' mit TestNewEndnode;
9: **end if**
10: **if** R enthält neue reduzierbare Endknoten **then**
11: Suche in R eine Menge W von Endknoten, für die ein passender Pfad existiert;
12: **if** $W' = \{w'\}$ mit $w' = [A' \to \alpha\cdot]$ **then**
13: Entferne aus $\mathcal{G}$ alle anderen Endknoten mit einer Graphbereinigung;
14: $A' :=$ Der direkte Vorgänger von w' in $\mathcal{G}$;
15: $\mathcal{E} := \mathcal{E} \setminus \{(A', w')\}$; $\mathcal{V} := \mathcal{V} \setminus \{w'\}$;
16: Untersuche die Vorgänger von A' wiederum mit TreatPredecessors;
17: **else** $(* \; W = \emptyset \; *)$
18: Entferne aus $\mathcal{G}$ alle reduzierbaren Endknoten mit einer Graphbereinigung;
19: **end if**
20: **end if**

Die Überprüfung der neuen Endknoten nach einer Reduktion oder einem Leseschritt ist in Algorithmus 3.5 beschrieben. Sollte der neue Knoten w expandierbar sein, wird er in die Menge H der zu expandierenden Knoten aufgenommen. Falls w nicht expandierbar, aber reduzierbar ist, wird er in die Menge R der zu reduzierenden Endknoten eingefügt und nach Rückkehr in die aufrufende Funktion (Reduce-

READ oder TREATPREDECESSORS) dort bearbeitet. Wenn der Knoten lesbar ist und das Symbol $a = a_{i+1}$ rechts vom Punkt stehen hat, wird er in die Menge C der lesbaren Endknoten eingefügt. Trifft keiner dieser Fälle zu, so ist der Knoten lesbar, aber mit einem Symbol $a \neq a_{i+1}$ hinter dem Punkt. Dieser Knoten kann nicht mehr zu einer akzeptierenden Berechnung führen und wird entfernt. Sollte $w = [S' \to S\cdot]$ sein, so darf w nicht reduziert werden. In diesem Fall darf w nur dann im Graph verbleiben, wenn die Eingabe verbraucht ist.

Algorithmus 3.5 TESTNEWENDNODE

Eingabe: Neuer Endknoten $w = [B \to \alpha \cdot \beta]$, aktueller Lookahead u.

1: **if** $w = [S' \to S\cdot]$ **then**
2: **if** $i \leq n$ **then**
3: Entferne w aus $\mathcal{G}$;
4: **end if**
5: **else if** $\beta \in NV^*$ **then** ($*$ Neuer Endknoten ist expandierbar. $*$)
6: $H := H \cup \{[B \to \alpha \cdot \beta]\}$;
7: **else if** $\beta = \varepsilon$ **then** ($*$ reduzierbar. $*$)
8: $R := R \cup \{[B \to \alpha\cdot]\}$;
9: **else if** $\beta \in \{a_{i+1}\}V^*$ **then** ($*$ lesbar mit richtigem Symbol $*$)
10: $C := R \cup \{[B \to \alpha \cdot \beta]\}$;
11: **else** ($*$ lesbar mit $a \neq a_{i+1}$ hinter dem Punkt $*$)
12: Entferne $[B \to \alpha \cdot \beta]$ mit einer Graphbereinigung aus $\mathcal{G}$;
13: **end if**

In Algorithmus 3.6 ist die Bereinigung des Graphen durch die Entfernung unnützer Knoten und ggf. ihrer Vorgänger beschrieben. Dabei wird der betreffende Endknoten aus dem Graphen entfernt und iterativ alle Knoten, für die sämtliche Nachfolger entfernt wurden. Damit auch Zyklen aus dem Graphen entfernt werden, von denen aus keine Endknoten mehr erreicht werden können, müssen auch Knoten entfernt werden, die in solchen Zyklen liegen. In Abbildung 3.1 wäre dies beispielsweise der Knoten v, nachdem w aus dem Graphen entfernt wurde. Je nach Implementierung des Suchalgorithmus nach Endknoten mit passenden Pfaden kann diese Zykleneliminierung aber auch ausbleiben, siehe die „zyklenignorierende Suche“ in Abschnitt 3.5.2. Dadurch verbleiben in einer Implementierung zwar Graphteile im Speicher, die nicht mehr benötigt werden, dafür entfällt aber die zusätzliche Suche nach solchen Zyklen bei einer Bereinigung.

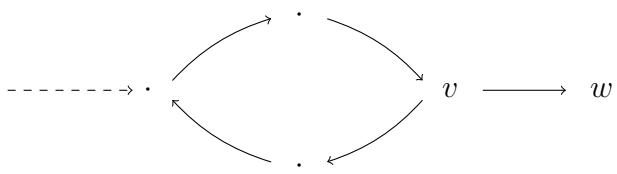

Abbildung 3.1: Nach Entfernen des Knotens w aus dem Graph kann von v aus kein Endknoten mehr erreicht werden.

Algorithmus 3.6 AdjustGraph

Eingabe: Menge D zu entfernender Knoten.

1: **while** $D \neq \emptyset$ **do**
2: Entnimm beliebigen Knoten $v \in D$;
3: **for all** $v' \in \mathcal{V} \mid v'$ ist in $\mathcal{G}$ direkter Vorgänger von v **do**
4: $\mathcal{E} := \mathcal{E} \setminus \{(v', v)\}$
5: **if** v' hat keine Nachfolger mehr in $\mathcal{G} \vee$ von v' aus ist kein Endknoten erreichbar **then**
6: $D := D \cup \{v'\}$;
7: **end if**
8: **end for**
9: $\mathcal{V} := \mathcal{V} \setminus \{v\}$;
10: **end while**

Für die Durchführung der Reduktions- und Leseschritte fehlt noch ein Algorithmus zum Finden eines Endknotens, für den ein passender Pfad existiert. In den folgenden beiden Abschnitten wird das Verfahren für kleine Lookahead-Längen k aus [Blu10a], Abschnitt 7.1, umgesetzt. Eine Version für größere Werte von k wird dort in Abschnitt 7.2 vorgestellt, im Rahmen dieser Diplomarbeit jedoch nicht weiter betrachtet. Zunächst wird in Abschnitt 3.3 der Suchalgorithmus für den Sonderfall $LR(1)$ ausformuliert. Darauf aufbauend folgt in Abschnitt 3.4 der Algorithmus für Lookaheadlängen $k > 1$.

3.3 Suchalgorithmus für Lookaheads der Länge $k = 1$

Der Suchalgorithmus RTSearch(1) (Algorithmus 3.7) verwaltet die Menge Q der zu untersuchenden Knoten in einer Schlange. Zwei Fälle sind zu unterscheiden:

Fall 1: Die Suchmenge Q enthält mindestens einen lesbaren Endknoten der Form $[A \to \alpha \cdot a\beta]$ mit $\alpha, \beta \in V^*$ und $a \in \Sigma$. Aufgrund der Invarianten für die Simulation des $LR(k)$-M_G befinden sich nur solche lesbaren Endknoten im Graphen, bei denen das Symbol a hinter dem Punkt identisch mit dem nächsten ungelesenen Symbol a_{i+1} der Eingabe ist. Daher ist für solche Endknoten für jeden Pfad $\mathcal{P}$ vom Startknoten zu $[A \to \alpha \cdot a\beta]$ die Bedingung $a_{i+1} \in \mathit{FIRST}_1(\mathit{suf}(\mathcal{P}))$ erfüllt. Gemäß Satz 1 ist dann der Leseschritt die eindeutig bestimmte nächste Aktion des Parsers. Alle reduzierbaren Endknoten können in diesem Fall aus $\mathcal{G}$ entfernt werden. Daher bietet es sich an, zuerst nach einem lesbaren Endknoten zu suchen.

Fall 2: Es gibt in Q nur Endknoten der Form $[A \to \varepsilon\cdot]$ bzw. $[A \to \alpha\cdot], \alpha \in V^*$. In diesem Fall muss der eindeutige reduzierbare Endknoten gefunden werden, für den es einen passenden Pfad in $\mathcal{G}$ gibt. Für diesen Knoten wird dann die Reduktion durchgeführt, alle anderen Endknoten werden entfernt. Das Auffinden des Endknotens geschieht durch rückwärtstopologische Suche, wie in [Blu10a] vorgeschlagen.

Definition 12 (Rückwärtstopologische Suche): Bei einer *topologischen Suche* auf einem gerichteten Graphen werden nur Knoten besucht, deren Vorgänger bereits alle besucht wurden. Eine *Rückwärtssuche* auf einem gerichteten Graphen durchquert die Kanten entgegen ihrer Richtung. Eine *rückwärtstopologische Suche* auf einem gerichteten Graphen ist eine Rückwärtssuche, die einen Knoten erst besucht, wenn alle seine Nachfolger bereits besucht wurden. ◇

Obwohl eine rückwärtstopologische Suche nur auf azyklischen Graphen anwendbar ist, stellen die Zyklen in $\mathcal{G}$ kein Problem dar, wie die Ausarbeitung des Suchalgorithmus zeigen wird.

Die Suche startet in einem Endknoten w. Da die Existenz lesbarer Endknoten schon in Fall 1 abgehandelt wurde, kann w nur ein reduzierbarer Knoten der Form $[A \to \alpha\cdot]$ sein. Der Endknoten wird gemäß den Zeilen 3–5 und 9–12 des Algorithmus RTSearch(1)-Item behandelt: Der direkte Vorgänger v von w wird in die Menge der zu besuchenden Knoten aufgenommen. Der Endknoten w wird mit v *assoziiert*, d. h. an v wird die Information weitergegeben, dass a_{i+1} aus den rechten Seiten von Vorgängerknoten von w hergeleitet werden muss. Dabei wird mit jedem Knoten des Graphen pro Suchlauf höchstens ein Endknoten assoziiert, da jeder Knoten während der Suche höchstens einmal besucht wird. Sollte nämlich ein innerer Knoten von zwei Endknoten aus besucht werden, so wäre die zugrunde liegende Grammatik

Algorithmus 3.7 RTSearch(1)

Eingabe: Menge Q von Knoten (als Schlange), in denen die Suche beginnen soll; aktuelles Lookaheadsymbol a_{i+1}.

Ausgabe: Falls Q lesbare Endknoten enthält: die Menge aller lesbaren Endknoten. Falls kein lesbarer Endknoten existiert, aber ein reduzierbarer Endknoten w, für den es einen passenden Pfad in $\mathcal{G}$ gibt: $\{w\}$.
Sonst: $\emptyset$.

1: **if** Q enthält mindestens einen Endknoten der Form $[A \to \alpha \cdot a\beta]$ **then**
2: **return** $\{w \mid w \in Q \wedge w = [A \to \alpha \cdot a\beta], a \in \Sigma, \alpha, \beta \in V^*\}$;
3: **end if**
4: $Result := \emptyset$;
5: **while** $Q \neq \emptyset \wedge Result = \emptyset$ **do**
6: Entnimm den ersten Knoten v aus Q;
7: **if** v repräsentiert ein Item **then**
8: $Result :=$ RTSearch(1)-Item(v); ($*$ Algorithmus 3.8 $*$)
9: **else** ($*$ v repräsentiert ein Nichtterminalsymbol C. $*$)
10: RTSearch(1)-Nonterminal(v); ($*$ Algorithmus 3.9 $*$)
11: **end if**
12: **end while**
13: **return** *Result*.

mehrdeutig und somit nicht $LR(1)$, wie in [Blu10a], Abschnitt 7.1, erläutert wird. Ein Endknoten ist stets mit sich selbst assoziiert.

Nachdem alle Endknoten aus Q besucht wurden, werden nun die inneren Knoten des Graphen bearbeitet, die in die Suchmenge aufgenommen wurden. Bezeichne im Folgenden v den aktuell betrachteten inneren Knoten und w den mit v assoziierten Endknoten. Wenn v ein Item der Form $[C \to \gamma \cdot D\delta]$ repräsentiert, $C, D \in N, \gamma, \delta \in V^*$, so wird überprüft, ob a_{i+1} aus der rechten Seite des Items herleitbar ist (Algorithmus 3.8). In diesem Fall ist die Reduktion von w der eindeutig bestimmte nächste Schritt des Parsers. Die Suche ist beendet mit $\{w\}$ als Rückgabewert.

Falls nicht a_{i+1}, aber ε aus δ herleitbar ist, so muss der direkte Vorgänger v' von v bei der Suche mit berücksichtigt werden. Der Endknoten w wird mit v' assoziiert und v' in die Suchmenge Q aufgenommen.

Falls weder a_{i+1} noch ε herleitbar sind, so kann a_{i+1} auf einem Pfad, auf dem v liegt, nicht hergeleitet werden. Die Suche wird in einem anderen Knoten $v' \in Q$ fortgesetzt, sofern vorhanden.

Algorithmus 3.8 RTSearch(1)-Item

Eingabe: Zu untersuchender Itemknoten v.

Ausgabe: Der mit v assoziierte Endknoten w, falls a_{i+1} durch die rechte Seite von v herleitbar ist; sonst: $\emptyset$.

1: **if** $v = [C \to \gamma \cdot D\delta], C, D \in N, \gamma, \delta \in V^*$ **then**
2: $\quad$ *right* $:= \delta$
3: **else** $(* \; v = [C \to \gamma\cdot], C \in N, \gamma \in V^* \; *)$
4: $\quad$ *right* $:= \varepsilon$;
5: **end if**
6: Sei w der mit v assoziierte Endknoten;
7: **if** $a_{i+1} \in FIRST_1(right)$ **then**
8: $\quad$ **return** $\{w\}$
9: **else if** $\varepsilon \in FIRST_1(right)$ **then**
10: $\quad$ Assoziiere den direkten Vorgänger v' von v mit w, falls v' existiert;
11: $\quad$ $Q := Q \cup \{v'\}$;
12: $\quad$ **return** $\emptyset$;
13: **else**
14: $\quad$ **return** $\emptyset$;
15: **end if**

Repräsentiert der aktuell untersuchte Knoten v ein Nichtterminalsymbol C, so wird er gemäß Algorithmus 3.9 behandelt. Zunächst wird geprüft, ob C Zielpunkt einer schließenden Kante ist, d. h., ob sich in C ein Zyklus schließt.

Definition 13 (Schließende Kante): Eine Kante $([B \to \gamma \cdot C\delta], C)$ heißt *schließende Kante*, wenn sie in $\mathcal{G}$ einen Zyklus

$$C \to [C \to \mu \cdot \nu] \to \cdots \to [B \to \gamma \cdot C\delta] \to C$$

schließt, $B \in N, \mu, \nu \in V^*$. Der Knoten $[B \to \gamma \cdot C\delta]$ wird *Finalknoten* genannt. $\diamond$

Wenn C nicht Ziel einer schließenden Kante ist, so wird w mit allen direkten Vorgängerknoten von C assoziiert und diese in die Menge zu untersuchender Knoten

aufgenommen (Zeilen 6 und 7 im Algorithmus). Der Knoten C ist damit abgearbeitet.

Wenn C Zielpunkt mindestens einer schließenden Kante ist, so muss zuerst für jede solche Kante $([B \to \gamma \cdot C\delta], C)$ überprüft werden, ob a_{i+1} durch die rechten Seiten von Knoten auf dem betreffenden Zyklus hergeleitet werden kann. Dies wird durch die Zeilen 2–5 in Algorithmus 3.9 eingeleitet. Erst nach der Überprüfung aller Zyklen kann die Suche bei den *echten Vorgängern* von C fortgesetzt werden, d. h. bei solchen Vorgängerknoten, die nicht gleichzeitig Nachfolger von C sind. Dazu wird w auch mit jedem Finalknoten $v' = [B \to \gamma \cdot C\delta]$ assoziiert und v' in der Suchmenge Q ganz vorne eingefügt. Dabei kann es nicht vorkommen, dass C von dem betrachteten Zyklus aus erneut besucht wird, da die Grammatik dann mehrdeutig wäre, siehe erneut Abschnitt 7.1 in [Blu10a]. Daher kann $\mathcal{G}$ trotz eventuell vorhandener Zyklen für die rückwärtstopologische Suche als azyklischer Graph angesehen werden. Wenn die Zyklenuntersuchung erfolglos verläuft, wird die Suche in den echten Vorgängerknoten von C fortgesetzt. Ausführlichere Anmerkungen zur Behandlung von Zyklen und zu den Problemen bei der Erkennung von schließenden Kanten befinden sich im nachfolgenden Abschnitt 3.5.

Algorithmus 3.9 RTSearch(1)-Nonterminal

Eingabe: Zu untersuchender Nichtterminalknoten C.

1: Sei w der mit C assoziierte Endknoten;
2: **for all** schließende Kanten (v', C) **do**
3: Assoziiere w mit dem Finalknoten v';
4: Füge v' in der Schlange Q ganz vorne ein;
5: **end for**
6: Assoziiere w mit allen echten Vorgängerknoten von v;
7: Füge die echten Vorgängerknoten von v ganz hinten in Q ein.

Sollte schließlich die Menge Q der zu besuchenden Knoten verbraucht sein, ohne dass a_{i+1} hergeleitet werden konnte, wird die Ausführung des Suchalgorithmus 3.7 mit der leeren Rückgabemenge beendet.

Für den Fall, dass die rückwärtstopologische Suche einen reduzierbaren Knoten zum Resultat hat, wird auf Seite 18f von [Blu10a] eine Möglichkeit aufgezeigt, eine ganze Folge von Reduktionen und Expansionen in einem Schritt abzuarbeiten. Sei $v = [B \to \gamma_1 \cdot C\gamma_2]$ der innere Knoten, in dem die Suche endete. Es gilt also $a_{i+1} \in \mathit{FIRST}_1(\gamma_2)$, und auf jedem Teilpfad $\mathcal{P}'$ zwischen v und dem assoziierten

Endknoten w wurde ε hergeleitet. Aufgrund der Anwendung von Satz 1 finden nun nach Reduktion von w für alle Pfade, die passend für w sind, dieselben Reduktionen und Expansionen statt. Alle Expansionen, die zwischendurch stattfinden, können nur zu neuen Reduktionen von Items der Form $[A' \to \cdot]$ führen. Gäbe es nämlich auf $\mathcal{P}'$ einen Knoten $v' = [D \to \delta_1 \cdot \delta_2 E \delta_3]$ mit $E \stackrel{*}{\Rightarrow} a_{i+1}\mu$, so würde $a_{i+1} \in FIRST_k(right(v'))$ gelten, und die Suche hätte schon in v' geendet.

Es findet also eine Kette von Reduktionen und Expansionen statt, die in jedem Fall in dem Knoten $[B \to \gamma_1 \cdot C\gamma_2]$ endet, in dem auch die rückwärtstopologische Suche ihren Abschluß fand. In diesem Knoten wird jetzt der Punkt um eine Stelle nach rechts geschoben, so dass daraus der neue lesbare oder expandierbare Endknoten $[B \to \gamma_1 C \cdot \gamma_2]$ entsteht. Nun kann der folgende Expansionsschritt eingeleitet werden.

Somit kann im Falle $k = 1$ Rechenzeit gespart werden, indem nach dem erfolgreichen Auffinden eines Endknotens w, für den ein passender Pfad existiert, die resultierende Kette aus Reduktions- und Expansionsschritten übersprungen wird und sofort der Knoten v manipuliert wird, in dem die Suche endete. Dies geschieht, indem die Anweisungen für die Fälle $w = [A \to \varepsilon\cdot]$ und $w = [A \to \alpha\cdot]$ in Algorithmus 3.3 (Zeilen 4–9 und 17–22) und Algorithmus 3.4 (Zeilen 12–16) durch Algorithmus 3.10 ausgetauscht werden.

Algorithmus 3.10 REDUCE(1)

1: Entferne aus $\mathcal{G}$ alle anderen Endknoten mit einer Graphbereinigung;
2: Sei $v = [B \to \gamma_1 \cdot C\gamma_2]$ der Knoten, in dem die Suche beendet wurde;
3: Entferne alle Knoten zwischen v und w.
4: Ersetze v durch $v' = [B \to \gamma_1 C \cdot \gamma_2]$;
5: Untersuche v' mit TESTNEWENDNODE; ($*$ Algorithmus 3.5 $*$)

Die Ausführungen im folgenden Unterabschnitt zeigen, dass sich das Suchverfahren im Fall $k > 1$ deutlich aufwändiger gestaltet.

3.4 Suchalgorithmus für Lookaheads der Länge $k > 1$

Für einen Lookahead u der Länge $k > 1$ muss auch für lesbare Endknoten der Form $[A \to \alpha \cdot a\beta]$ untersucht werden, ob es einen Pfad $\mathcal{P}$ im Graphen $\mathcal{G}$ gibt, für den $u \in FIRST_k(suf(\mathcal{P}))$ ist. Dazu wird der Graph wie zuvor mittels rückwärtstopologischer Suche durchmustert, wie in Algorithmus 3.11 beschrieben.

Algorithmus 3.11 RTSearch(k)

Eingabe: Initiale Suchmenge $E = R \cup C$ oder $E = R$, aktueller Lookahead u.

Ausgabe: Ein lesbarer oder reduzierbarer Endknoten w, falls ein passender Pfad $\mathcal{P}$ vom Startknoten zu w existiert; sonst: $\emptyset$.

1: $Q := \emptyset$; *Result* $:= \emptyset$;
2: **for all** Endknoten $w \in \mathcal{V}$ **do**
3: **if** $w = [A \to \alpha \cdot B\beta]$ **then** ($*$ expandierbar $*$)
4: $L(w) := \emptyset$
5: **else if** $w = [A \to \alpha\cdot]$ **then** ($*$ reduzierbar $*$)
6: $L(w) := \langle(\varepsilon, w)\rangle$
7: **else** ($* \; w = [A \to \alpha \cdot a\beta]$, d. h. lesbar $*$)
8: **if** $E = R$ **then**
9: $L(w) := \emptyset$;
10: **else**
11: **for all** $u' \in FIRST_k(a\beta)$ mit $\exists u'' : u'u'' = u$ **do**
12: $L(w) := L(w) \cup \langle(u', w)\rangle$
13: **end for**
14: **if** $L(w) = \emptyset$ **then**
15: Entferne w aus $\mathcal{G}$ mit einer Graphbereinigung;
16: **else if** $(u, w) \in L(w)$ **then**
17: **return** w;
18: **end if**
19: **end if**
20: **end if**
21: Sei v der direkte Vorgänger von w in $\mathcal{G}$;
22: $Q := Q \cup \{v\}$
23: **end for**
24: **while** $Q \neq \emptyset \wedge$ *Result* $= \emptyset$ **do**
25: Entnimm den ersten Knoten $v \in Q$;
26: **if** v repräsentiert ein Item **then**
27: *Result* := RTSearch(k)-Item ($*$ Algorithmus 3.12 $*$)
28: **else**
29: *Result* := RTSearch(k)-Nonterminal ($*$ Algorithmus 3.13 $*$)
30: **end if**
31: **end while**
32: **return** *Result*

Im Gegensatz zum Fall $k = 1$ kann ein innerer Knoten v nun von mehreren Endknoten aus besucht werden. Daher genügt es nicht mehr, mit jedem inneren Knoten nur einen Endknoten zu assoziieren. Stattdessen verwaltet der Algorithmus für jeden besuchten Knoten v eine Liste $L(v)$ der echten Präfixe von u, die aus den rechten Seiten der Items auf einem Pfad von v zu einem Endknoten hergeleitet werden können. Um am Ende der Suche erkennen zu können, für welchen Endknoten ein passender Pfad gefunden wurde, wird jedes Präfix mit einem eindeutigen Endknoten assoziiert. Die Präfixliste eines Knotens $v \in \mathcal{G}$ hat also die Form

$$L(v) = \left\langle (u'_1, w_1), (u'_2, w_2), \ldots, (u'_j, w_j) \right\rangle$$

mit $j \leq k$, da ein Lookahead-String u der Länge k genau die k echten Präfixe ε, x_1, x_1x_2, …, $x_1x_2 \ldots x_{k-1}$ hat. Folgendes Lemma besagt, dass diese Präfixliste stets eindeutig ist, es also nicht mehrere Einträge für dasselbe Präfix geben kann:

Lemma 1: *Während der rückwärtstopologischen Suche werden für einen Knoten v in der Liste $L(v)$ niemals zwei unterschiedliche Endknoten w und w' mit demselben Präfix u' von u assoziiert.* □

Beweis: Sei v der aktuell betrachtete Knoten. Angenommen, es gäbe in der Präfixliste $L(v)$ Einträge (p, w_1) und $(p, w_2), w_1 \neq w_2$, d. h. mit demselben Präfix p des aktuellen Lookaheadstrings u sind zwei verschiedene Endknoten assoziiert. Dies könnte dann eintreten, wenn ein Nichtterminalknoten dasselbe Präfix von zwei verschiedenen Nachfolgerknoten übernimmt. Dann allerdings gäbe es für den selben String zwei verschiedene Ableitungen, was im Widerspruch zur Eindeutigkeit einer $LR(k)$-Grammatik steht. ■

1.) Behandlung der Endknoten: Abhängig von der Situation im aufrufenden Algorithmus (Reduce-Read oder TreatPredecessors) erhält Algorithmus 3.11 als initiale Suchmenge die Endknotenmenge $E = R \cup C$ oder $E = R$. Letzteres ist der Fall, wenn der Suchalgorithmus während der Behandlung neuer reduzierbarer Endknoten zum Abschluss eines Reduktions-/Leseschrittes aufgerufen wird (Zeile 16 in Algorithmus 3.3, Zeile 11 in Algorithmus 3.4). Analog zu RTSearch(1)werden die zu besuchenden inneren Knoten in einer Schlange Q verwaltet. Die Suche beginnt mit der Menge aller Endknoten von $\mathcal{G}$ (Zeilen 2–19). Sei w der aktuell betrachtete Endknoten. Bei der Behandlung von w werden drei Fälle unterschieden, je nachdem, ob w expandierbar, reduzierbar oder lesbar ist:

a) Behandlung expandierbarer Endknoten (Zeilen 3 und 4): Falls der Suchalgorithmus im Anschluss an einen Reduktions-/Leseschritt zwecks Behandlung der Vorgängerknoten aufgerufen wurde, kann w auch ein expandierbarer Knoten sein. In diesem Fall wird w mit leerer Präfixliste als besucht markiert, und seine Vorgänger werden in die Suchmenge aufgenommen. Dieser Besuch ist nötig, da innere Knoten im Subalgorithmus RTSearch(k)-Nonterminal (Algorithmus 3.13) immer nur dann bearbeitet werden, wenn alle ihre Nachfolger schon besucht wurden. Andernfalls müssten Knoten, die für die aktuelle Suche irrelevant sind, in einem Vorlauf entsprechend markiert werden, was auf die Laufzeit des Verfahrens denselben Effekt hätte wie das Besuchen solcher Knoten während der Suche.

b) Behandlung reduzierbarer Endknoten (Zeilen 5 und 6): Wenn w reduzierbar ist, wird sein Präfixvektor mit $L(w) = \langle(\varepsilon, w)\rangle$ initialisiert und sein direkter Vorgänger v zu Q hinzugefügt.

c) Behandlung lesbarer Endknoten (Zeilen 7–19): Sei $w = [A \to \alpha \cdot a\beta]$ der aktuell betrachtete lesbare Endknoten. Sollte die initiale Suchmenge E nur aus der Menge R der reduzierbaren Endknoten bestanden haben, muss w nicht weiter berücksichtigt werden. Er wird aus den selben Gründen betreten wie ein expandierbarer Endknoten und ebenso behandelt. Ansonsten muss zunächst geprüft werden, welchen Anteil die rechte Seite $a\beta$ von w zur Herleitung von u leisten kann. Dazu werden alle Elemente der Menge

$$\{u' \mid u' \in \mathit{FIRST}_k(a\beta) \wedge u = u'u'' \text{ für } u'' \in \Sigma^*\}$$

mit w assoziiert und zur Präfixliste $L(w)$ von w hinzugefügt. Dies sind alle Präfixe von u, die durch $a\beta$ herleitbar sind. Sollte schon der ganze String u durch ein Element aus $\mathit{FIRST}_k(a\beta)$ hergeleitet sein, ist die Suche schon beendet. Ein Leseschritt anhand aller lesbaren Endknoten im Graphen ist dann gemäß Satz 1 die eindeutig bestimmte nächste Aktion des Parsers. Falls u noch nicht fertig hergeleitet ist, wird der direkte Vorgängerknoten v von w in die Suchmenge Q aufgenommen und die Suche fortgesetzt.

2.) Behandlung innerer Knoten: Wenn alle Endknoten bearbeitet wurden, ohne den Lookahead vollständig herzuleiten, folgt in den Zeilen 24–31 die Untersuchung der inneren Knoten, unterschieden nach Itemknoten und Nichtterminalknoten. Sei v der aktuell betrachtete innere Knoten.

Algorithmus 3.12 RTSearch(k)-Item

Eingabe: Zu untersuchender Itemknoten $v = [A \to \alpha \cdot B\beta]$.

Ausgabe: Ein Endknoten w, falls ein assoziiertes Präfix u' durch die rechte Seite von v zu u verlängert werden kann. Sonst: Keine.

```
1:  L(v) := ∅;
2:  for all (u', w) ∈ L(B) do
3:     Sei u = u'u'';
4:     if ∃ū ∈ Σ* : u''ū ∈ FIRST_k(β) then
5:        return w
6:     else
7:        Pref := {p ∈ FIRST_k(β) | p ist Präfix von u''};
8:        for all p ∈ Pref do
9:           L(v) := L(v) ∪ {(u'p, w)}
10:       end for
11:    end if
12: end for
13: Sei v' der direkte Vorgängerknoten von v;
14: Q := Q ∪ v';
```

a) Behandlung innerer Itemknoten: Falls v ein Item $[A \to \alpha \cdot B\beta]$ repräsentiert, $A, B \in N$, $\alpha, \beta \in V^*$, so muss überprüft werden, welchen Beitrag β zur Herleitung von u leisten kann. Dies geschieht in Algorithmus 3.12.

Der einzige direkte Nachfolgerknoten von v ist der Knoten für die Variable B. Für jeden Eintrag u' in der Präfixliste $L(B)$ wird getestet, ob u' durch Strings in $FIRST_k(\beta)$ verlängert werden kann. Falls u' durch $u'' \in FIRST_k(\beta)$ oder ein Präfix davon zu u verlängert werden kann, ist die Suche beendet. Das Resultat der Suche ist der Endknoten w, der mit dem Präfix u' in $L(B)$ assoziiert ist.

Falls u' durch einen String $p \in FIRST_k(\beta)$ nicht vollständig hergeleitet, aber zu einem Präfix der Länge $\geq |u'|$ von u verlängert werden kann, wird das Präfix $u'p$ mit w assoziiert und in die Präfixliste $L(v)$ von v eingetragen. Dabei ist auch $p = \varepsilon$ zulässig.

Sollte sich kein Eintrag aus $L(B)$ verlängern lassen und sich dadurch ein Teilpfad $\mathcal{P}$ in G als unnütz für die aktuelle Suche herausstellen, so können die Knoten auf $\mathcal{P}$ dennoch nicht aus dem Graphen entfernt werden. Zum einen könnte die Suche aus der Vorgängerbehandlung nach einem Reduktions-/Leseschritt aufgerufen worden

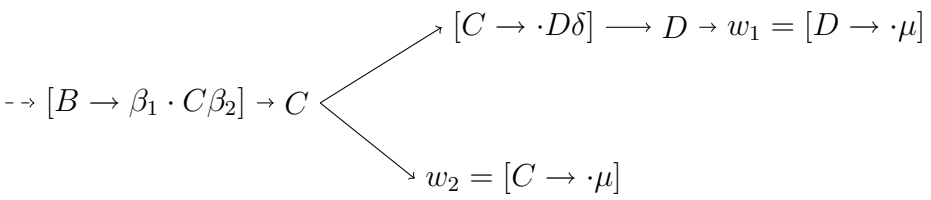

Abbildung 3.2: Der Nichtterminalknoten C wird ggf. sofort nach Betrachtung von w_2 besucht, obwohl noch unbesuchte Knoten zwischen C und w_1 liegen.

sein. Dann könnten durch das Entfernen von Knoten fälschlicherweise Pfade zu expandierbaren Endknoten entfernt werden. Zum anderem wäre es möglich, dass der aktuell betrachtete Knoten v auf einem Zyklus liegt, der für die Herleitung von u zwingend benötigt wird, falls ein vorheriger Durchlauf des Zyklus während der selben Suche zu einer Verlängerung des hergeleiteten Präfixes von u beitragen konnte.

b) Behandlung innerer Nichtterminalknoten: Nichtterminalknoten werden gemäß Algorithmus 3.13 behandelt. Dabei werden folgende Begriffe benutzt, um Knoten, die zu Zyklen gehören, von anderen zu unterscheiden:

Definition 14 (echte und zyklische Nachfolger/Vorgänger): Ein Knoten v' heißt *echter Nachfolger* eines Knotens v, falls v' nicht zugleich Vorgänger von v ist, falls also die beiden Knoten nicht zu einem gemeinsamen Zyklus gehören. Sollten v' und v zu einem gemeinsamen Zyklus gehören, heißt v' *zyklischer Nachfolger* von v. Die Begriffe *echter* und *zyklischer Vorgänger* sind analog definiert. ◇

Sei C das durch den aktuell besuchten Knoten v repräsentierte Nichtterminalsymbol. Da zwei verschiedene Endknoten w_1 und w_2 unterschiedlich weit von C entfernt sein können, kann der Fall eintreten, dass C besucht wird, obwohl noch nicht alle seine echten Nachfolger behandelt wurden (vgl. Abbildung 3.2). In diesem Fall muss die Betrachtung von C aufgeschoben werden (Zeilen 1 und 2).

Seien nun alle echten Nachfolger von C besucht. Wenn von C keine Zyklen ausgehen, entsteht die Liste $L(C)$ durch Vereinigung der Listen der direkten Nachfolgerknoten. Wie schon erwähnt, kann aufgrund der Eindeutigkeit von $LR(k)$-Grammatiken dabei dasselbe echte Präfix von u nicht in den Listen zweier Nachfolgerknoten von v zugleich enthalten sein. In einer Implementierung bietet sich daher

Algorithmus 3.13 RTSearch(k)-Nonterminal

Eingabe: Zu untersuchender Knoten v für ein Nichtterminalsymbol C.

Ausgabe: Falls während der Untersuchung eines Zyklus ein passender Pfad für einen Endknoten w gefunden wird: Dieser Knoten w. Sonst: $\emptyset$.

1: **if** C hat noch unbesuchte echte Nachfolger in G **then**
2: $Q := Q \cup \{C\}$;
3: **else**
4: $L(C) := \bigcup \{L(v') \mid v'$ ist echter Nachfolger von v in $\mathcal{G}\}$;
5: changed := 1;
6: **while** changed = 1 **do**
7: changed := 0;
8: **for all** schließende Kanten $e = ([B \to \gamma \cdot C\delta], C)$ **do**
9: $Q' := \{[B \to \gamma \cdot C\delta]\}$;
10: **while** $Q' \neq \emptyset$ **do**
11: Entnimm den ersten Knoten $v' \in Q'$;
12: **if** v' repräsentiert ein Item **then**
13: Result := RTSearch(k) − Item(v', Q');
14: **else if** v' repräsentiert ein Nichtterminalsymbol $\wedge\ v' \neq v$ **then**
15: Result := RTSearch(k) − Nonterminal(v', Q');
16: **end if**
17: **if** Result $\neq \emptyset$ **then**
18: **return** Result;
19: **end if**
20: **end while**
21: **end for**
22: $L' := \bigcup \{L(v') \mid v'$ ist zyklischer Nachfolger von v in $\mathcal{G}\}$;
23: **if** $L' \setminus L(C) \neq \emptyset$ **then**
24: $L(C) := L(C) \cup L'$;
25: changed := 1;
26: **end if**
27: **end while**
28: $Q := Q \cup \{v' \mid v'$ ist echter Vorgänger von v in $\mathcal{G}\}$;
29: **end if**
30: **return**

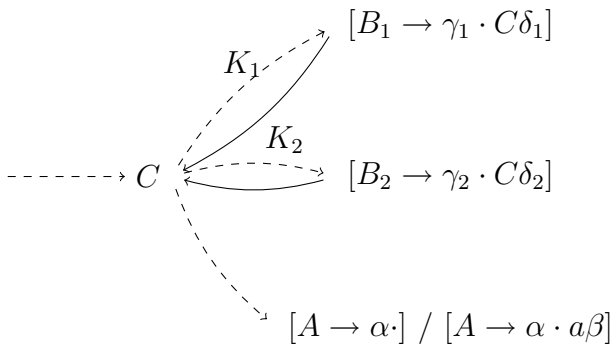

Abbildung 3.3: Vom Nichtterminalknoten C gehen zwei Zyklen K_1 und K_2 sowie ein Pfad zu einem Endknoten ab.

an dieser Stelle eine Möglichkeit zur Fehlererkennung. Sollte es nämlich doch vorkommen, dass dasselbe Präfix von u von zwei verschiedenen Nachfolgerknoten aus in $L(C)$ eingefügt werden soll, so ist die Grammatik mehrdeutig und somit nicht $LR(k)$.

Sollten aber Zyklen von C ausgehen, so müssen diese betrachtet werden, bevor die rückwärtstopologische Suche von C aus fortgesetzt wird. Dabei ist folgende, in Abbildung 3.3 skizzierte Situation denkbar: Zur Herleitung von u sind möglicherweise mehrere Zyklentraversierungen nötig, beispielsweise erst K_1, dann K_2, dann nochmals K_1, und dann erst weiter im Graphen von C aus. Daher wird eine parallele Betrachtung aller in C eingehenden schließenden Kanten so oft iteriert, bis u entweder vollständig hergeleitet ist oder bis keines der bisher hergeleiteten Präfixe von u durch eine erneute Iteration vergrößert werden kann. Dies findet sich in Algorithmus 3.13 in den Zeilen 6–27 wieder. Danach wird die Suche in den echten Vorgängerknoten von C fortgesetzt.

Sämtliche bisherigen Überlegungen basieren auf der Annahme, dass schließende Kanten im Graphen effizient erkannt werden können. Mit den Problemen, die bei der Erkennung und Behandlung von Zyklen im Graphen auftreten können, beschäftigt sich der folgende Abschnitt.

3.5 Behandlung von Zyklen im Parsegraphen

Folgende, auch in [Blu10a] offen gebliebene, Frage ist noch nicht beantwortet worden: Wie stellt man sicher, dass während einer rückwärtstopologischen Suche auf dem

Graphen $\mathcal{G}$ ein Nichtterminalknoten C erst untersucht wird, nachdem als erstes alle echten Nachfolger und danach alle in C endenden schließenden Kanten untersucht wurden?

Dazu wäre es wünschenswert, beim Besuch von C zu erkennen, welche Nachfolgerknoten echt und welche zyklisch sind, sodass eine genauere Betrachtung von C ggf. aufgeschoben werden kann. Daher folgt im nächsten Unterabschnitt ein Vergleich verschiedener Strategien der Zyklenerkennung. Danach wird in Abschnitt 3.5.2 eine Variante der rückwärtstopologischen Suche vorgestellt, die auf Kosten möglicherweise längerer Suchzeiten ohne eine Erkennung der Zyklen in $\mathcal{G}$ auskommt.

3.5.1 Erkennung von schließenden Kanten und Finalknoten

Im Folgenden werden vier Methoden betrachtet, mittels derer der Graph $\mathcal{G}$ auf das Vorhandensein von Zyklen untersucht werden kann. Die verwendeten Grundalgorithmen zu Breiten- und Tiefensuche und dem Erkennen starker Zusammenhangskomponenten sind in gängigen Lehrbüchern wie [Blu01] zu finden.

Erkennung von Zyklen schon beim Expandieren: Besonderer Suchaufwand könnte vermieden werden, wenn es möglich wäre, eine entsprechende Ordnung der Knoten bereits während ihrer Erstellung, d. h. während der Expansionsschritte, aufzubauen. Da in einer Phase mehrere Expansionsschritte erfolgen können, unterbrochen durch Reduktionsschritte, entspricht die Reihenfolge der Knotenerstellung aber nicht unbedingt der Traversierungsreihenfolge bei einer Breiten- oder Tiefensuche. Daher erscheint es nicht praktikabel, eine entsprechende Ordnung durch Nummerierung oder Färbung der Knoten und Kanten während der Expansionen aufrecht zu erhalten. Eine Variation der Idee, die Information über Zyklen bei jeder Änderung im Graphen zu aktualisieren, wird in der folgenden Strategie entwickelt. ◇

Verwalten von Vorgängerlisten: Unterhalte für jeden Knoten v des Graphen eine Liste $PrevNT(v)$, die alle Nichtterminalknoten aus derselben Phase enthält, die Vorgänger von v sind. So könnte man sofort beim Expandieren und auch später während einer rückwärtstopologischen Suche erkennen, ob für einen Itemknoten v sein nachfolgender Nichtterminalknoten C zugleich auch Vorgänger ist. Dabei genügt es, nur die Vorgängerknoten aus derselben Phase zu berücksichtigen, da Zyklen nur durch mehrfache Expansionen eines Symbols innerhalb ein und derselben Phase

entstehen können. Für dieses Verfahren ist es somit erforderlich, jeden Knoten mit der Phase seiner Erschaffung bzw. seiner letzten Modifikation zu indizieren.

Sobald ein Zyklus gefunden wird, müssen die *PrevNT*-Listen aller Knoten auf dem Zyklus um ihre neu entdeckten zyklischen Vorgänger ergänzt werden. Ebenso müssen die Listen von evtl. bereits besuchten echten Nachfolgerknoten von Knoten auf dem Zyklus aktualisiert werden. Dies bedarf entweder eines erneuten Durchlaufes durch den Zyklus und die betroffenen Nachfolger, oder man „verzeigert“ die Listen:

$$PrevNT(v) := \star PrevNT(C) \cup \{C\}.$$

Dabei soll nicht statisch der Inhalt der Menge $PrevNT(C)$ zum Zeitpunkt der Berechnung von $PrevNT(v)$ eingefügt werden. Stattdessen stellt $\star PrevNT(C)$ einen Zeiger auf den jeweils *aktuellen* Wert dar, vgl. dazu die Funktionsweise von Zeigern in imperativen Programmiersprachen wie C. Dadurch werden die *PrevNT*-Listen aller Nachfolgerknoten des Knotens C automatisch aktualisiert, sobald $PrevNT(C)$ sich ändert.

Leider genügt es nicht, nach Erkennen eines Zyklus diese Information bei der schließenden Kante zu vermerken, um die platzaufwändige Vorgängerliste am Phasenende löschen zu können. Die Tatsache, dass ein Zyklus vorliegt, muss nämlich von jedem Knoten darauf erkennbar sein, nicht nur von dem Knoten aus, in dem er entdeckt wurde. Deutlich wird das Problem am Beispiel eines Kreises mit „Ausgang“, wie in Abbildung 3.4 dargestellt: Die Existenz des Zyklus würde nur im Knoten C vermerkt werden. So könnte während einer anschließenden rückwärtstopologischen Suche im Knoten E das Problem auftreten, dass der Zyklus dort nicht erkannt wird. Der Suchalgorithmus bliebe stecken, weil der Vorgängerknoten von E erst besucht werden dürfte, nachdem alle Nachfolger bearbeitet sind. Der Nachfolger $[E \to \cdot C\gamma]$ würde aber aufgrund des Zyklus niemals besucht werden.

Auch für die Bereinigung des Graphen wäre es wichtig, einen Zyklus in jedem seiner Knoten erkennen zu können. Würde der Parser hier nämlich erkennen, dass ein Knoten nur noch Nachfolgerknoten hat, die keinen Endknoten erreichen, sondern zu einem Zyklus gehören, können dieser Knoten und der ganze Zyklus, zu dem er gehört, entfernt werden. Dies ist nötig, damit die oben erwähnten unnützen Kreise so früh wie möglich aus dem Graphen entfernt werden können.

Bei einer Implementierung der *PrevNT*-Mengen mit dieser Zeigerstruktur beträgt der Zusatzaufwand bei der Erschaffung oder Manipulation eines Knotens $O(1)$. Zur Überprüfung, ob für einen Knoten v ein Nichtterminalknoten $C \in PrevNT(v)$ ist,

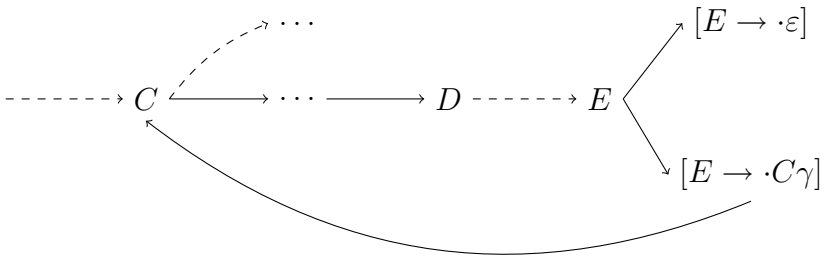

Abbildung 3.4: Wenn die Information über den Zyklus nur im Knoten C vermerkt wird, kann der Parser bei der Untersuchung des Knotens E in eine Endlos-Warteschleife geraten.

müssen aber schlimmstenfalls die Listen von $O(|G|)$ Knoten durchsucht werden, die aus der gleichen Phase stammen.

Wenn in einer Implementierung des Parsers die Nichtterminalsymbole durchgehend nummeriert sind, könnte man $PrevNT(v)$ auch als Bitvektor der Länge $|N|$ speichern. Dann soll gelten: $PrevNT(v)[j] = 1$ genau dann, wenn der Nichtterminalknoten für das Symbol mit der Nummer j zu den Vorgängern von v gehört und denselben Phasenindex wie v hat. Dann ist das Nachsehen in Zeit $O(1)$ möglich, aber der Vorteil der Verzeigerung geht verloren, sodass nach jeder Änderung des Vektors für einen Nichtterminalknoten gegebenenfalls die Vektoren all seiner $O(|G|)$ vielen Nachfolger ebenfalls aktualisiert werden müssen. Somit erscheint dieses Verfahren gegenüber einer Breitensuche bei Bedarf, die als nächstes vorgestellt wird, keine wesentlichen Vorteile zu haben. ◇

Erreichbarkeitssuche: Mittels einer Breitensuche (Breadth-first search, BFS) läßt sich für einen Knoten v in einem Graphen bestimmen, welche anderen Knoten in $\mathcal{G}$ von v aus erreichbar sind. Eine solche Breitensuche wäre immer dann durchzuführen, wenn während einer rückwärtstopologischen Suche ein Nichtterminalknoten C besucht wird, der noch unbesuchte Nachfolger hat. Die Suche startet mit der Suchmenge

$$Q_{\mathrm{BFS}} := \{v \mid v \text{ ist direkter Nachfolgerknoten von } C\}.$$

Folgende Situationen werden während der BFS unterschieden:

- Die Suche erreicht einen Endknoten w. Für C wird vermerkt, dass ein weiterer Endknoten erreichbar ist. Sofern Q_{BFS} noch Knoten enthält, wird die Breiten-

suche fortgesetzt. Bei Rückkehr in die rückwärtstopologische Suche wird die Betrachtung von C aufgeschoben, weil die Teilpfade zwischen C und w noch nicht vollständig untersucht wurden.

- Die Suche erreicht einen Knoten v', der während der aktuellen rückwärtstopologischen Suche bereits betrachtet wurde. Dann ist von v' aus ein Endknoten erreichbar, und C kann wie im obigen Fall behandelt werden.

- Es wird ein Knoten v erreicht, der aus einer anderen Phase stammt als C. Da Zyklen in $\mathcal{G}$ allein durch mehrfache Expansionen anhand desselben Symbols innerhalb einer Phase entstehen können, kann v nicht Teil eines Zyklus sein, der C enthält. Seine Nachfolgerknoten können also für die BFS ignoriert werden, und C wird behandelt wie in den beiden vorigen Fällen. Dies setzt allerdings voraus, dass der Parser für jeden Knoten speichert, in welcher Phase er erschaffen oder zuletzt modifiziert wurde.

- Die BFS erreicht C über eine schließende Kante $e = ([B \to \beta_1 \cdot C\beta_2], C)$. In diesem Fall ist C Teil eines Zyklus, der noch nicht untersucht wurde. Für C wird vermerkt, dass bei der Fortsetzung der rückwärtstopologischen Suche nach der Betrachtung aller echten Nachfolger von C die schließende Kante e betrachtet werden muss. Erst danach kann mit der Untersuchung der Vorgänger von C begonnen werden.

- Die Breitensuche endet, weil $Q_{\mathrm{BFS}} = \emptyset$. Dann wurde mindestens ein Endknoten oder Finalknoten erreicht, oder beides. Nun wird die rückwärtstopologische Suche in C so fortgesetzt, wie es der Befund aus der BFS erfordert.

  Der Fall, dass von C aus weder ein Endknoten noch ein Finalknoten erreichbar ist, kann nicht eintreten. Dies wäre nämlich nur möglich, wenn von C aus ein *unnützer Kreis* erreichbar wäre, also ein Kreis, von dem aus kein Endknoten erreichbar ist, vgl. Abbildung 3.5. Dieser Fall wird aber durch die Graphbereinigung ausgeschlossen.

Der Aufwand für die zusätzliche Breitensuche läßt sich in den meisten Fällen als konstanter Faktor auf den Aufwand der rückwärtstopologischen Suche aufschlagen, von der aus die BFS gestartet wurde. Jeder Knoten, der von C aus erreichbar ist, wird während der Breitensuche höchstens einmal besucht, ebenso wie bei der rückwärtstopologischen Suche. Die Breitensuche wird von C aus höchstens einmal pro

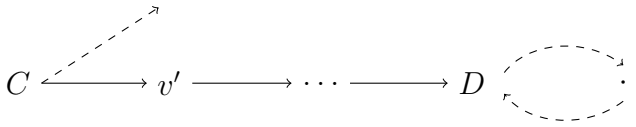

Abbildung 3.5: Von C aus ist ein unnützer Kreis erreichbar.

rückwärtstopologischer Suche gestartet, sofern die Listen, die währenddessen angelegt wurden, jeweils nach Abarbeiten eines bisher unbesuchten Nachfolgers von C aktualisiert werden. Nur in dem Fall, dass von C aus ein Nichtterminalknoten D erreichbar ist, für den später ebenfalls eine BFS nötig wird, kann der Aufwand größer sein. Falls D nämlich während der aktuellen rückwärtstopologischen Suche noch nicht besucht wurde, werden die Teilpfade zwischen D und den von dort erreichbaren Endknoten des Graphen während der Untersuchung von D möglicherweise erneut betreten. Im schlimmsten Fall werden also einzelne Knoten während einer rückwärtstopologischen Suche in $O(|N|)$ vielen Breitensuchen besucht. Da somit auch bei diesem Verfahren nicht auszuschließen ist, dass manche Knoten mehrfach betrachtet werden müssen, wird als letztes eine Strategie betrachtet, bei der zwar der gesamte Graph durchmustert werden muss, dafür aber jeder Knoten und jede Kante nur konstant oft durchlaufen wird. ◇

Suche nach starken Zusammenhangskomponenten: Mittels einer Tiefensuche (Depth-first search, DFS) lassen sich für einen Graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ die starken Zusammenhangskomponenten in Zeit $O(|\mathcal{V}| + |\mathcal{E}|)$ bestimmen. Man könnte also vor jedem Reduktions-/Leseschritt die starken Zusammenhangskomponenten von $\mathcal{G}$ bestimmen und in einer geeigneten Datenstruktur speichern. Um während einer rückwärtstopologischen Suche für einen Nichtterminalknoten C und einen Nachfolgerknoten v zu erfahren, ob sie auf einem gemeinsamen Zyklus liegen, müsste man dann nur nachsehen, ob sich beide Knoten in der selben starken Zusammenhangskomponente befinden. Da die Anzahl der Knoten und Kanten des Graphen durch $O(|G|n)$ nach oben beschränkt ist, würde dieses Verfahren einen zeitlichen Mehraufwand von $O(|G|n)$ je rückwärtstopologischer Suche bewirken. Geht man davon aus, dass während einer rückwärtstopologischen Suche schlimmstenfalls jeder Knoten einmal besucht werden muss, bis ein gültiger Endknoten identifiziert ist, läßt sich der Aufwand für die zusätzliche DFS mit diesem Suchaufwand verrechnen. ◇

Als Alternative zu den unterschiedlich aufwändigen Verfahren zum Erkennen von

Zyklen im Graphen wird im nächsten Unterabschnitt ein modifiziertes Verfahren für die rückwärtstopologische Suche vorgestellt, das keine Kenntnis über die Zyklen im Graphen erfordert.

3.5.2 Zyklenignorierende Suche

Da alle bisher aufgeführten Strategien zum Erkennen und Behandeln von Zyklen zusätzlichen Verwaltungsaufwand im Graphen verursachen, werden in der aktuellen Version des Generators Parser erzeugt, die die Existenz von Zyklen ignorieren. Beim Betreten eines Nichtterminalknotens C während einer rückwärtstopologischen Suche werden sofort alle seine direkten Vorgänger in die Suchmenge aufgenommen. Sollte C in einem Zyklus liegen, kann es nun vorkommen, dass Teilpfade zwischen dem Startknoten $[S' \rightarrow \cdot S]$ und C mehrfach besucht werden. Dadurch entsteht zwar zusätzlicher Rechenaufwand, was aber die Korrektheit des Verfahrens nicht schmälert. Weitere Überlegungen zur Korrektheit folgen nach der Vorstellung des Suchverfahrens.

Da bei dieser Variante der rückwärtstopologischen Suche mit der Betrachtung einzelner Knoten nicht gewartet wird, bis alle Nachfolgerknoten besucht wurden, ist es auch nicht zwingend nötig, während der Graphbereinigung mittels AdjustGraph (Algorithmus 3.6) unnütze Kreise aus $\mathcal{G}$ zu entfernen. Außerdem ist es dadurch im Unterschied zum Suchalgorithmus in Abschnitt 3.4 möglich, die Suchmenge auf eine Teilmenge aller Endknoten einzuschränken. So ist es nun nicht mehr nötig, zum Aufrechterhalten der „besucht“-Relation auch expandierbare Endknoten zu besuchen, und bei einer Untersuchung neuer Endknoten nach einem Reduktions-/Leseschritt können auch die lesbaren Endknoten übergangen werden. Dadurch kann Rechenzeit für den Besuch von Knoten eingespart werden, die für die aktuelle Suche irrelevant sind.

Das modifizierte Verfahren wird durch die Algorithmen 3.14, 3.15 und 3.16 implementiert. Der Aufbau von Algorithmus 3.14 orientiert sich an Algorithmus 3.11 aus Abschnitt 3.4. Der Parser unterhält einen globalen Wert *searchID*, durch den jeder Suchlauf der rückwärtstopologischen Suche einen eindeutigen Index erhält (Zeile 1). Knoten, die während eines Suchlaufs besucht werden, werden mit diesem Wert indiziert (Zeilen 18 und 28), so dass leicht zu prüfen ist, ob ein Knoten während der aktuellen Suche schon besucht wurde oder nicht.

Um Endlosschleifen bei der Untersuchung von Zyklen zu vermeiden, muss der

Algorithmus 3.14 RTSearch'(k)

Eingabe: Menge Q zu durchsuchender reduzier- und/oder lesbarer Endknoten; aktueller Lookahead-String u.

Ausgabe: Ein lesbarer oder reduzierbarer Endknoten w, falls in $\mathcal{G}$ ein passender Pfad von $[S' \to \cdot S]$ zu w gefunden wird. Sonst: $\emptyset$.

1: *searchID* := *searchID* + 1;
2: **for all** Endknoten $w \in Q$ **do**
3: *prefixCode*(w) := 0;
4: **if** $w = [A \to \alpha\cdot]$ **then**
5: $L(w) := \langle(\varepsilon, w)\rangle$;
6: *prefixCode*(w)[1] := 1;
7: **else** ($* \; w = [A \to \alpha \cdot a\beta] \; *$)
8: **for all** $u' \in FIRST_k(a\beta)$ **do**
9: $L(w) := L(w) \cup \langle(u', w)\rangle$;
10: *prefixCode*(w)[$|u'| + 1$] := 1;
11: **end for**
12: **if** $L(w) = \emptyset$ **then**
13: AdjustGraph(w);
14: **else if** $(u, w) \in L(w)$ **then**
15: **return** w;
16: **end if**
17: **end if**
18: $Q := Q \setminus \{w\}$; *visited*(w) := *searchID*;
19: $Q := Q \cup \{v \mid v$ ist direkter Vorgänger von w in $\mathcal{G}\}$;
20: **end for**
21: **while** $Q \neq \emptyset \wedge$ *Result* $= \emptyset$ **do**
22: Entnimm den ersten Knoten $v \in Q$;
23: **if** v repräsentiert ein Item **then**
24: *Result* := RTSearch'(k)-Item(v); ($*$ Algorithmus 3.15 $*$)
25: **else** ($* \; v$ repräsentiert eine Variable $*$)
26: RTSearch'(k)-Nonterminal(v); ($*$ Algorithmus 3.16 $*$)
27: **end if**
28: *visited*(v) := *searchID*;
29: **end while**
30: **return** *Result*.

Parser beim Besuch eines Nichtterminalknotens v in Algorithmus 3.16 erkennen können, ob sich ein erneuter Besuch in seinem Vorgängerknoten v' lohnt, d. h. ob sich die Präfixliste $L(v)$ seit dem letzten Besuch von v' verändert hat. In diesem Falle wurde nämlich seit dem letzten Besuch des Vorgängers v' ein neues Präfix entdeckt, das beim letzten Betreten von v' noch nicht berücksichtigt werden konnte. Somit muss v' erneut in die Suchmenge aufgenommen werden.

Für jeden Itemknoten v' muss also gespeichert werden, wie die Präfixliste des nachfolgenden Nichtterminalknotens zum Zeitpunkt des letzten Besuchs von v' aussah (Zeile 1 in Algorithmus 3.15). Dies geschieht hier mit Hilfe eines Bitvektors *prefixCode* der Länge k, der zusätzlich zur Präfixliste für jeden Knoten gespeichert wird. Ein gesetztes erstes Bit $\mathit{prefixCode}(v)[1]$ bedeutet dann, dass das Präfix ε in $L(v)$ enthalten ist, das Bit j, $j \leq k$, wird gesetzt, wenn $L(v)$ das Präfix der Länge $j+1$ enthält. In einer Rechnerimplementierung kann dieser Bitvektor als Integer-Zahl gespeichert werden, wobei die Zahl 0 dann für den leeren Vektor steht. Das Setzen des Eintrags für ein Präfix der Länge l für den Knoten v geschieht dann durch $\mathit{prefixCode}(v) := \mathit{prefixCode}(v)$ **OR** 2^l, wobei der Operator **OR** ein bitweises „Oder“ auf zwei Integer-Zahlen durchführt.

Die Behandlung eines Nichtterminalknotens v gestaltet sich dann folgendermaßen, siehe Algorithmus 3.16: Falls der Nichtterminalknoten v während des aktuellen Suchlaufs noch nicht besucht wurde, stammen evtl. vorhandene Einträge in $L(v)$ von vorhergehenden Suchläufen und müssen überschrieben werden (Zeilen 2–4). Sollte v jedoch während der aktuellen Suche schon einmal besucht worden sein, wird seine Präfixliste nicht ersetzt, sondern nur ergänzt um alle Präfixe, die bis zu seinen direkten Nachfolgern hergeleitet werden konnten (Zeile 5). Außerdem wird der Vektor *prefixCode* entsprechend angepasst, damit er den neuen Zustand von $L(v)$ widerspiegelt (Zeile 6). Schließlich werden, wie oben erwähnt, genau diejenigen Vorgängerknoten v' von v in die Suchmenge aufgenommen, für die eine der beiden folgenden Bedingungen gilt (Zeile 8 im Algorithmus):

1. v' wurde während der aktuellen Phase noch nicht besucht ($\mathit{visited}(v') \neq \mathit{searchID}$), oder

2. seit des letzten Besuchs in v' hat sich die Präfixliste von v verändert ($\mathit{prefixCode}(v') \neq \mathit{prefixCode}(v)$).

Die Korrektheit des Verfahrens ergibt sich aus den folgenden Überlegungen, getrennt nach Endknoten, inneren Knoten, von denen aus kein Zyklus erreichbar ist,

Algorithmus 3.15 RTSearch'(k)-Item

Eingabe: Zu untersuchender Knoten v für ein Item $[A \to \alpha \cdot B\beta]$; aktueller Lookahead-String u.

Ausgabe: Ein Endknoten w, falls ein zu w assoziiertes Präfix u' von u durch die rechte Seite von v zu u verlängert werden kann. Sonst: Keine.

```
1:  prefixCode(v) := prefixCode(B);
2:  for all (u', w) ∈ L(B) do
3:     Sei u = u'u'';
4:     if ∃ū ∈ Σ* : u''ū ∈ FIRST_k(β) then
5:        return w;
6:     else
7:        Prefs := {p ∈ Σ* | p ist Präfix von u'' ∧ p ∈ FIRST_k(β)};
8:        for all p ∈ Prefs do
9:           L(v) := L(v) ∪ {(u'p, w)};
10:       end for
11:    end if
12: end for
13: if L(v) ≠ ∅ then
14:    Q := Q ∪ {v' ∈ V | v' ist direkter Vorgänger von v in G};
15: end if
```

und inneren Knoten, die Teil eines Zyklus sind oder von denen aus Zyklen erreichbar sind.

Endknoten: Endknoten, die zum Suchergebnis beitragen können, werden einmal besucht, genau wie in Algorithmus 3.11. Endknoten, die RTSearch'(k) ausläßt, wären auch im anderen Algorithmus nur mit einer leeren Präfixliste initialisiert worden, hätten also zum Ergebnis der Suche ebensowenig beigesteuert.

Innere Knoten, die keine Zyklen erreichen: Ein innerer Knoten v, von dem aus kein Zyklus erreichbar ist, wird unter Umständen mehrfach besucht, wobei sich die Präfixliste, auf deren Grundlage der Knoten untersucht wird ($L(B)$ in Zeile 1 von Algorithmus 3.15, $L(v)$ in Zeile 5 von Algorithmus 3.16) bei jedem Besuch anders darstellt. Konstruktionsbedingt gilt aber für jede solche Folge $L_1, L_2, \ldots, L_l$ von

Algorithmus 3.16 RTSearch'(k)-Nonterminal

Eingabe: Zu untersuchender Knoten v für ein Nichtterminalsymbol C, aktueller Lookahead u.

Ausgabe: Keine, da direkt auf den globalen Datenstrukturen des Parsers operiert wird.

1: $Succs := \{v' \mid v' \text{ ist direkter Nachfolger von } v \wedge visited(v') = searchID\}$;
2: **if** $visited(v) \neq searchID$ **then**
3: $\quad L(v) := \emptyset$; $prefixCode(v) := 0$;
4: **end if**
5: $L(v) := L(v) \cup \bigcup \{L(v') \mid v' \in Succs\}$;
6: $prefixCode(v) := prefixCode(v) \vee \bigvee \{prefixCode(v') \mid v' \in Succs\}$;
7: ($*$ Es müssen nun alle diejenigen Vorgänger von v besucht werden, die in der aktuellen Suche noch nicht berücksichtigt wurden oder die zuletzt besucht wurden, als die Präfixliste von v anders als zur Zeit aussah: $*$)
8: $Q := Q \cup \{v' \mid v' \text{ ist direkter Vorgänger von } v \text{ in } \mathcal{G} \wedge$
$\quad ((visited(v') \neq searchID) \vee (prefixCode(v') \neq prefixCode(v)))\}$.

Listen, die in der angegebenen Reihenfolge behandelt werden:

$$L_1 \subset L_2 \subset \ldots \subset L_l.$$

L_l ist dabei die Liste, die v erhält, nachdem alle Nachfolger vollständig bearbeitet worden sind, und entspricht somit der Liste, die auch in RTSearch(k) an v übergeben worden wäre.

Innere Knoten, die einen Zyklus erreichen: Für Nichtterminalknoten, die Teil eines Zyklus sind, stellt sich die Situation ganz ähnlich dar. Sei C ein solcher Knoten, der Endpunkt mindestens einer schließenden Kante (v', C) mit Finalknoten v' ist. Der Suchalgorithmus in Abschnitt 3.4 veranlasst, dass vor einem Besuch der echten Vorgänger von C zuerst solange alle in C endenden schließenden Kanten behandelt werden, bis sich $L(C)$ durch eine erneute Kreistraversierung nicht mehr verändern kann. Dieselben Schritte werden auch während der zyklenignorierenden Suche durchgeführt. Der Unterschied ist, dass zwischendurch auch Vorgängerknoten von C auf der Grundlage einer unvollständigen Teilliste $L' \subset L(C)$ betrachtet werden können. Nachdem aber alle Zyklen behandelt wurden, werden C und seine Vorgänger betrachtet, diesmal anhand der vollständigen Liste $L(C)$.

Somit stellt die zyklenignorierende Suche ein Verfahren dar, dass die rückwärtstopologische Suche ebenfalls korrekt durchführt und dabei keine Sonderbehandlung von Zyklen erfordert. Dies geschieht allerdings um den Preis möglicher mehrfacher Knotenbesuche und somit eines höheren Zeitbedarfs. Eine Analyse für den Gesamtzeitbedarf des Parsers, der in diesem Kapitel entwickelt wurde, liefert Abschnitt 3.7. Vorher folgen in Abschnitt 3.6 einige Überlegungen zur Behandlung der $FIRST_k$-Mengen, die während der Suche nach gültigen Endknoten benötigt werden.

3.6 Berechnung der $FIRST_k$-Mengen

Die in den vorigen Abschnitten vorgestellten Suchalgorithmen setzen voraus, dass man für eine Teilsatzform $\alpha \in V^*$ die Menge $FIRST_k(\alpha)$ bestimmen kann. Daher beschäftigt sich dieser Abschnitt mit der Berechnung dieser $FIRST_k$-Mengen. Zur Erinnerung: Für $\alpha \in V^*$ enthält $FIRST_k(\alpha)$ alle Terminalstrings der Länge $\leq k$, die durch endliche Anwendung der Regeln in P herleitbar sind, sowie von allen herleitbaren Strings x mit $|x| \geq k$ deren Präfixe der Länge k. Formal:

$$FIRST_k(\alpha) := \left\{ x \in \Sigma^* \mid \alpha \stackrel{*}{\Rightarrow} xy, (y = \varepsilon \text{ und } |x| \leq k) \vee (y \in \Sigma^* \text{ und } |x| = k) \right\}.$$

Für $a \in \Sigma$ gilt offensichtlich $FIRST_k(a) = \{a\}$. Ferner seien die Operatoren $\cdot$ und $\oplus_k$ für $L_1, L_2 \subseteq V^*$ und $k \in \mathbb{N}_0$ folgendermaßen definiert:

$$L_1 \cdot L_2 := \{\alpha\beta \mid \alpha \in L_1, \beta \in L_2\}$$

und

$$L_1 \oplus_k L_2 := \{x \in \Sigma^* \mid \exists \alpha \in L_1, \beta \in L_2 : x \in FIRST_k(\alpha\beta)\}.$$

Ein Verfahren zur Berechnung der $FIRST_k$-Mengen für die Variablenmenge N einer Grammatik wird in [AU72], Kapitel 5.1.6 vorgestellt. Der für diese Diplomarbeit implementierte Parsergenerator verwendet dieses Verfahren, das auf dem Konzept der Fixpunktiteration basiert. Ein effizienteres Verfahren, dass für große Längen k des Lookaheads vorteilhafter erscheint, wird in Kapitel 6.4 der ersten Auflage von [Blu01] entwickelt.

Zunächst berechnet der Parsergenerator mittels des erwähnten Fixpunktverfahrens für alle Variablen $A \in N$ die Mengen $FIRST_k(A)$. Anschließend werden für alle Items $[A \to \alpha \cdot \beta],\ \beta \in V^*$, die $FIRST_k$-Mengen ihrer rechten Seiten β berechnet.

Dies geschieht für $\beta = A_1 A_2 \dots A_r \in V^*$ gemäß der ebenfalls [Blu01] entnommenen Regel

$$FIRST_k(\beta) = FIRST_k(A_1) \oplus_k FIRST_k(A_2) \oplus_k \dots \oplus_k FIRST_k(A_r).$$

Diese Mengen schließlich werden im erzeugten Parser in der Datenstruktur für die Items hinterlegt, um dort während der rückwärts-topologischen Suche abgerufen zu werden.

In Blums Arbeit [Blu10a] wird vorgeschlagen, für jede Menge U von aktuell betrachteten Präfixen des Lookaheadstrings u im Voraus die Folgemenge U' zu bestimmen, die durch die Hinzunahme der $FIRST_k$-Menge eines weiteren Symbols A_j entsteht. Dies bietet sich an, da U' nur von u und den Mengen U und $FIRST_k(A_j)$ abhängt. In den Parsergenerator, der für diese Arbeit implementiert wurde, wurde diese Berechnung allerdings nicht aufgenommen. Der Grund liegt im Platzbedarf der vorausberechneten Tabellen, der in [Blu10a] mit

$$O(|\Sigma|^k \cdot |N| \cdot 2^k \cdot k \log k)$$

angegeben ist. Die folgenden Rechenbeispiele sollen dies verdeutlichen.

Für den Test des Parsergenerators bzw. der erzeugten Parser wurde unter anderem eine Grammatik für die Programmiersprache Pascal durch Einfügen zusätzlicher Regeln derart modifiziert, dass sie $LALR(2)$ ist. Die Grammatik hat $|\Sigma| = 70$ Terminalsymbole und $|N| = 140$ Variablen. Legt man für jedes einzelne Datum einen Speicherbedarf von nur einem Bit zugrunde, ergibt sich eine obere Schranke für den Speicherbedarf $\mathfrak{S}$ als

$$\begin{aligned} \mathfrak{S} &\leq 70^2 \cdot 140 \cdot 2^2 \cdot 2 \cdot \log 2 \text{ Bit} \\ &\approx 601,7 \text{ KByte}. \end{aligned}$$

Würde man eine vergleichbare Grammatik erzeugen, die nicht $LR(2)$, aber $LR(3)$ ist, stiege der Platzbedarf bei ansonsten gleichen Bedingungen bereits auf

$$\begin{aligned} \mathfrak{S} &\leq 70^3 \cdot 140 \cdot 2^3 \cdot 3 \cdot \log 3 \text{ Bit} \\ &\approx 65,5 \text{ MByte}. \end{aligned}$$

Daher wurde für diese Arbeit auf eine vollständige Vorausberechnung der Präfixlisten im Parsergenerator verzichtet und stattdessen in den Parserfunktionen eine Berechnung im Bedarfsfall implementiert.

Der folgende Abschnitt schließt das Kapitel mit einer Analyse der Laufzeit des Parsers ab, der in den bisherigen Abschnitten vorgestellt wurde.

3.7 Laufzeitanalyse

In einigen Punkten unterscheiden sich die in dieser Arbeit vorgestellten Algorithmen und somit auch die Implementierung des Parsers von Blums Vorschlägen in [Blu10a]. Durch die Wahl einer anderen Methode zum Umgang mit Zyklen und den Verzicht auf die Vorausberechnung der Präfixlisten hat sich der Zeitaufwand für die rückwärtstopologischen Suchen verändert. Somit ergeben sich die folgenden Überlegungen für eine Laufzeitabschätzung des Parsers:

Bezeichne in diesem Abschnitt stets $x \in L(G)$ eine Eingabefolge für den Parser, $n = |x|$ die Länge der Eingabe und k die Länge des Lookaheads. Weiterhin ist der Begriff der Länge einer Ableitung nützlich:

Definition 15 (Länge einer Ableitung): Sei $G = (N, \Sigma, P, S)$ eine eindeutige Grammatik und x eine Symbolfolge aus $L(G)$. Die *Länge einer Ableitung* für x, $\mathrm{ld}(x)$, ist die Anzahl der Ableitungsschritte, die benötigt werden, um x aus dem Startsymbol S abzuleiten. ◇

Die Länge einer Ableitung gibt also an, wieviele Produktionsanwendungen nötig sind, um x zu erhalten, bzw. wie viele Reduktionen ein Shift-Reduce-Parser durchführen muss, um diese Ableitung zu berechnen. Obige Definition kann nicht auf mehrdeutige Grammatiken angewendet werden, da diese für dieselbe Symbolfolge mehrere verschiedene Ableitungen erlauben können. Da für $LR(k)$-Grammatiken Eindeutigkeit gefordert wird, kann der Begriff bei den folgenden Betrachtungen angewandt werden.

Für das Erzeugen und Entfernen einzelner Knoten und Kanten sowie für die Operationen zur Verwaltung des Graphen und der diversen Knotenmengen, d. h. für das Einfügen, Suchen oder Entfernen eines Knotens, wird ein Aufwand von $O(1)$ angenommen. In der Praxis hängen die Kosten dieser Operationen von der Implementierung der verwendeten Datenstrukturen ab, vgl. dazu die Messungen in Unterabschnitt 5.2.1. Unter dieser Annahme können die Kosten für die Initialisierung des Graphen und für die Ausgabe in SIMULATION($LR(k)$-M_G) (Algorithmus 3.1) mit $O(1)$ angesetzt werden. Dominanter Faktor für den Zeitaufwand des Parsers sind also die Kosten für die Expansionen, Reduktionen und Leseschritte.

Zunächst werden die Kosten für Knoten und Kanten betrachtet, die zu Reduktionen gehören. Dies umfasst für jede Reduktion den reduzierbaren Endknoten w, für den während der rückwärtstopologsichen Suche ein passender Pfad gefunden wurde, den Nichterminalknoten v, der sein direkter Vorgänger ist, und die Kante zwischen

v und w. Der Argumentation bei der Analyse in [Blu10a] folgend, können die Kosten für das Einfügen und Löschen von Knoten, die zu Reduktionen korrespondieren, gegen die entsprechenden Produktionen aufgerechnet werden, die während der Ableitung der Eingabe Anwendung finden. Während eines Parserlaufes finden $O(\mathrm{ld})$ viele Reduktionen statt, und da für jeden Pfad vom Startknoten $[S' \to \cdot S]$ zu einem Endknoten dieselben Reduktionen stattfinden, beträgt der Aufwand für die betroffenen Knoten und Kanten insgesamt $O(\mathrm{ld}(x))$. Während derjenigen rückwärtstopologischen Suche, die zur Reduktion eines Endknotens w führt, wird w genau einmal besucht. Während dieses Besuchs wird seine Präfixliste mit dem leeren Wort ε initialisiert, dieser Besuch verursacht also einen Aufwand von $O(1)$. Somit verursachen die Knoten und Kanten, die direkt an Reduktionen beteiligt sind, über den gesamten Parserlauf einen Zeitaufwand von insgesamt

$$O(\mathrm{ld(x)}). \tag{3.1}$$

Knoten und Kanten, die nicht an einer Reduktion beteiligt sind, werden durch die Expansionen während einer Phase maximal so viele erzeugt, wie die Grammatik Produktionen und Nichtterminalsymbole aufweist, also $O(|G|)$ viele, vgl. dazu auch die Argumentation in Abschnitt 3.1. Dazu kommen noch einige Itemknoten der Form $[A \to \alpha B \cdot \beta]$, die nach einer Reduktion eines Knotens $[B \to \gamma\cdot]$ durch Kopieren eines seiner Vorgängerknoten $[A \to \alpha \cdot B\beta]$ gemäß der Zeilen 6 bis 9 in TreatPredecessors (Algorithmus 3.4) entstehen. Da solche Knoten nicht durch Expansionen erzeugt werden können und der Wert $|G|$ die Gesamtanzahl der Items umfasst, die es für die Grammatik G geben kann, geht die Anzahl dieser Knoten auch in den $O(|G|)$ vielen Knoten auf, die pro Phase erschaffen werden. Im Laufe der n Phasen werden also insgesamt $O(n \cdot |G|)$ Knoten und Kanten erzeugt, die nicht zu einer Reduktion korrespondieren. Für jeden dieser Knoten können eine oder mehrere der folgenden Operationen in einer Phase stattfinden:

- Der Knoten wird gelöscht.
- Der Punkt des zugehörigen Items wird um eine Position nach rechts verschoben.
- Der Knoten ist ein Itemknoten $v = [A \to \alpha \cdot B\beta]$ und ist Ausgangspunkt einer Expansion. Dann wird eine neue Kante (v, B_i) in den Graphen eingefügt, was zu einem Kostenbeitrag von $O(1)$ für v führt.

- Der Knoten repräsentiert ein Nichtterminalsymbol, das während einer Expansion mit neuen Vorgänger- und Nachfolgerknoten verbunden wird. Sollte C neu erschaffen werden, verursacht dies Kosten von $O(1)$. Der Zeitaufwand für die Verbindung mit dem neuen Vorgänger v wurde oben dem Knoten v zugerechnet. Die Kosten für die Erzeugung der Kanten zu den Nachfolgern von C können diesen mit je $O(1)$ zugeteilt werden.

Setzt man auch für das Löschen eines Knotens und für das Verschieben des Punktes in einem Item einen Zeitbedarf von je $O(1)$ voraus, ergibt sich für die Menge aller Knoten, die nicht an einer Reduktion teilnehmen, ein Zeitbedarf von

$$O\left(n \cdot |G| + \mathcal{R}\right). \tag{3.2}$$

Dabei bezeichnet $\mathcal{R}$ den Zeitaufwand, der durch alle Knotenbesuche während der rückwärtstopologischen Suchen verursacht wird. Für die Untersuchung dieses Aufwands wird sich das folgende Lemma als nützlich erweisen:

Lemma 2: *Ein Knoten $v \in \mathcal{G}$ wird während einer zyklenignorierenden rückwärtstopologischen Suche maximal k-mal besucht.* □

Beweis: Im Algorithmus RTSearch'(k) (Algorithmus 3.14, Zeile 2–20) erkennt man, dass jeder Endknoten $w \in \mathcal{G}$ höchstens einmal während eines Suchlaufs betreten wird. Der Rest des Beweises beschränkt sich daher auf eine Betrachtung der inneren Knoten des Graphen. Ein innerer Knoten v kann aus folgenden Gründen in die Suchmenge Q aufgenommen werden:

1. Er ist direkter Vorgänger eines Endknotens und wird deshalb in Zeile 19 von Algorithmus 3.14 in die Suchmenge eingetragen,

2. Er ist als Nichtterminalknoten Vorgänger eines Itemknotens, der zur Herleitung des Lookaheads einen Beitrag leisten konnte (Zeilen 13–15 in Algorithmus 3.15).

3. v ist ein Itemknoten und wurde während der aktuellen Suche noch nicht besucht ($\mathit{visited}(v) \neq \mathit{searchID}$, Zeile 8 in Algorithmus 3.16).

4. v ist ein Itemknoten, und die Präfixliste des Nichtterminalknotens v', der direkter Nachfolger von v ist, hat sich seit dem letzten Besuch von v verändert ($\mathit{prefixCode}(v') \neq \mathit{prefixCode}(v)$, ebenfalls Zeile 8 von Algorithmus 3.16).

Da in Unterabschnitt 3.5.2 gezeigt wurde, dass während der zyklenigorierenden Suche höchstens solche Präfixe auftreten, die auch während des normalen Suchverfahrens gefunden worden wären, kann auch hier Lemma 1 angewandt werden. Da in allen vier Situationen ein Knoten v immer nur dann besucht wird, wenn sich die Präfixlisten seiner Nachfolger geändert haben, kann v also maximal k-mal betreten werden. Somit ist die Gesamtzahl der Besuche in einem Knoten während einer zyklenignorierenden rückwärtstopologischen Suche durch k beschränkt. ∎

Nun können die Kosten betrachtet werden, die im Laufe aller rückwärtstopologischen Suchen für diejenigen Knoten des Graphen aufgewendet werden müssen, die nicht an einer Reduktion teilnehmen. Da vor jeder Reduktion und jedem Leseschritt genau eine rückwärtstopologische Suche stattfindet, gibt es insgesamt während eines Parserlaufes für eine Eingabefolge x genau $n + \mathrm{ld}(x)$ viele Suchläufe. Ein Knoten, der sehr weit „vorne“ im Graphen liegt, d. h. nur wenige Kanten vom Startknoten $[S' \to \cdot S]$ entfernt ist, kann dabei schlimmstenfalls an allen Suchläufen während eines Parsevorgangs teilnehmen. Der Suchaufwand wird für Nichtterminalknoten und Itemknoten getrennt behandelt.

Nichtterminalknoten: Wird ein Nichtterminalknoten C besucht, so werden die Präfixlisten aller Nachfolgerknoten zusammengefasst zur neuen Präfixliste von C. Dabei können maximal k Werte übertragen werden, da gemäß Lemma 1 dasselbe Präfix nicht in der Präfixliste von zwei Nachfolgerknoten zugleich enthalten sein kann. Jeder Besuch eines Nichtterminalknotens während einer rückwärtstopologischen Suche benötigt also $O(k)$ Zeit für das Zusammenfügen der Listen. Da jeder Knoten während eines Suchlaufs maximal k Mal besucht wird, verursacht ein Nichtterminalknoten also je Suchlauf einen Zeitaufwand von $O(k^2)$. In jeder der n Phasen wird für jedes der $|N|$ Nichtterminalsymbole der Grammatik höchstens ein Knoten angelegt. Somit betragen die Kosten für die Behandlung aller Nichtterminalknoten während einer rückwärtstopologischen Suche $O(k^2 \cdot n \cdot |N|)$. Der Gesamtaufwand, den die Untersuchung aller Nichtterminalknoten während der Analyse der Eingabe verursacht, beträgt also

$$O\left((n + \mathrm{ld}(x)) \cdot k^2 \cdot n \cdot |N|\right). \tag{3.3}$$

Itemknoten: Beim Besuch eines Itemknotens v müssen die maximal k Präfixeinträge desjenigen Nichtterminalknotens, der sein einziger direkter Nachfolger ist, auf

Verlängerbarkeit getestet werden. Da bei der Implementierung des Parsergenerators, wie in Abschnitt 3.6 erklärt, auf eine Vorberechnung der Präfixlisten verzichtet wurde, muss jede dieser Zeichenfolgen einzeln darauf überprüft werden, ob sie durch die Strings der Länge $\leq k$ in $FIRST_k(right(v))$ verlängert werden kann. Dabei bezeichnet $right(v)$ die *rechte Seite des Items*, das durch den Knoten v repräsentiert wird. Für jeden einzelnen String $u'_j \in L(v)$ muss also jeder String in $FIRST_k(right(v))$ mit dem noch nicht hergeleiteten Rest u''_j des Lookaheadstrings $u = u'_j u''_j$ verglichen werden. Der Vergleich eines solchen Stringpaares benötigt $O(k)$ viel Zeit, folglich hat die Folge der Vergleiche von u''_j mit allen Einträgen von $FIRST_k(right(v))$ eine Dauer von $O(k \cdot |FIRST_k(right(v))|)$. Solche Vergleichsfolgen müssen so oft durchgeführt werden, wie $L(v)$ Präfixeinträge enthält, also höchstens k viele. Sei

$$f_{max} := \max_{v \in \mathcal{V}} \{|FIRST_k(right(v))|\}\,.$$

Dann beträgt der Aufwand für einen Besuch eines inneren Itemknotens v:

$$O(k^2 \cdot f_{max}).$$

Gemäß Lemma 2 kann auch ein Itemknoten während einer Suche maximal k Besuche erfahren. Somit beträgt der Gesamtzeitaufwand für einen Itemknoten, der nicht zu einer Reduktion korrespondiert, während eines Suchlaufs

$$O(k^3 \cdot f_{max}).$$

Der Gesamtaufwand aller Suchläufe für alle Itemknoten zusammen beträgt also

$$O\left((n + \mathrm{ld}(x)) \cdot k^3 \cdot f_{max} \cdot n \cdot |G|\right). \tag{3.4}$$

Eine obere Grenze für f_{max} stellt $|\Sigma|^k$ dar, womit sich der vollständige Aufwand für einen Lauf des Parsers durch Addition der Terme (3.1), (3.2), (3.3) und (3.4) abschätzen läßt durch

$$\begin{aligned} O\,(&\mathrm{ld}(x) \\ &+ n \cdot |G| \\ &+ (n + \mathrm{ld}(x)) \cdot k^2 \cdot n \cdot |N| \\ &+ (n + \mathrm{ld}(x)) \cdot k^3 \cdot |\Sigma|^k \cdot n \cdot |G|\ )\,. \end{aligned}$$

Da alle betrachteten Werte ≥ 1 sind und $|N| \leq |G|$, läßt sich das Ergebnis dieses Kapitels in folgendem Satz zusammenfassen:

Satz 2: *In diesem Kapitel wurde ein Parser für eine LR(k)-Grammatik G entwickelt, der eine Laufzeit von*

$$O\left((n + \mathrm{ld}(x)) \cdot k^3 \cdot |\Sigma|^k \cdot n \cdot |G|\right)$$

hat, wobei x eine Eingabefolge der Länge n ist, $\mathrm{ld}(x)$ *die Länge einer Ableitung für x angibt und der Lookahead eine Länge von k hat.* □

Für größere Werte von k könnte bei dieser Implementierung der Fall eintreten, dass manche Knoten während einer zyklenignorierenden Suche so oft besucht werden müssen, dass insgesamt eine vorgeschaltete Tiefensuche zu geringeren Kosten führen würde. Für kleine Werte, besonders für $k = 1$ oder $k = 2$, überwiegt aber der Nachteil der Tiefensuche, dass dort gegebenenfalls viele Knoten unnötig besucht würden, die während der anschließenden rückwärtstopologischen Suche keine Berücksichtigung finden.

Im folgenden Kapitel sind nähere Angaben zur Implementierung des Parsergenerators und der davon erzeugten Parser ausgeführt. Im Rahmen einer Erläuterung der für den Parser verwendeten Datenstrukturen werden dort auch der Platzbedarf für den Parser und für die Verwaltung des Graphen bestimmt.

4 Implementierung des Parsergenerators

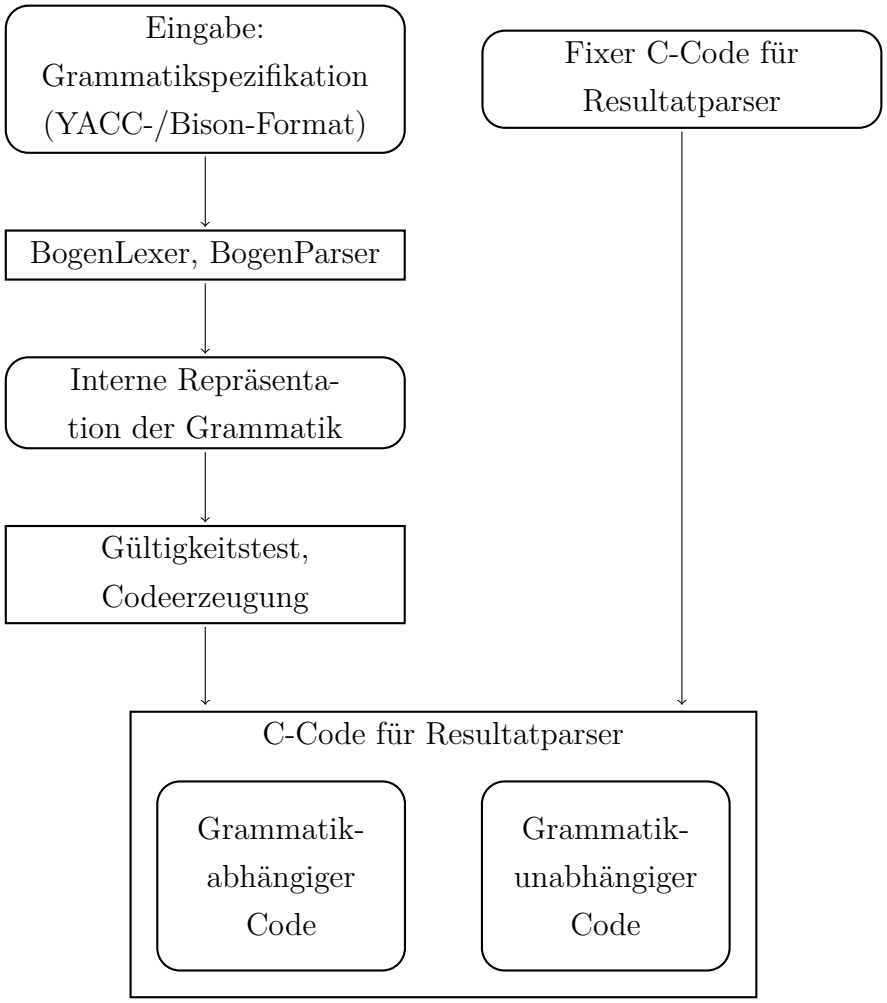

Abbildung 4.1: Aufbau des Parsergenerators BogenLR.

Für diese Arbeit wurde ein Parsergenerator namens *„BogenLR“* (Bonn Generator für $LR(k)$-Parser) implementiert, dessen Funktionsweise schematisch in Abbildung 4.1

dargestellt ist. Die Eingabe des Parsergenerators ist die Spezifikation einer Grammatik G in einem Format, das an die Eingaben für die gebräuchlichen Parsergeneratoren Yacc bzw. GNU Bison angelehnt ist, siehe [Joh] und [BIS]. Diese wird von den Modulen BogenLexer und BogenParser des Generators eingelesen und in eine interne Darstellung umgewandelt. Nachdem diese interne Grammatikrepräsentation einigen Gültigkeitstests unterzogen wurde, wird daraus anschließend der Programmcode für einen Parser für G erzeugt.

Die Verwendung des erzeugten $LR(k)$-Parsers ist in Abbildung 4.2 skizziert. Für einen Eingabestring $x \in \Sigma$ endet der Parserlauf mit Rückgabewert 0, falls x in der von G erzeugten Sprache $L(G)$ enthalten ist, ansonsten mit einer Fehlermeldung.

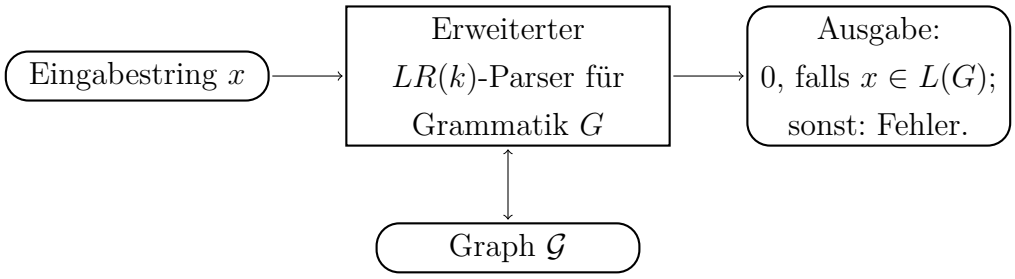

Abbildung 4.2: Verwendung des erzeugten Parsers.

Die Quellcodedateien für den Parsergenerator und für den fixen Anteil des Resultatparsers befinden sich auf der CD-ROM, die dieser Arbeit beiliegt, im Verzeichnis `src`. Auf derselben CD-ROM, im Verzeichnis `doc`, befindet sich auch eine Projektdokumentation, in der die implementierten Datenstrukturen und Funktionen ausführlicher erläutert werden.

Sämtliche Programmier- und Testarbeiten fanden unter Linux- und NetBSD-Betriebssystemen statt, die beschriebenen Programme sollten aber auf jedem System übersetzbar sein, auf dem mindestens ein OCaml-System [OCA] und der C-Compiler GCC ([GCC]) zur Verfügung stehen.

4.1 Verwendung des Parsergenerators

Der Parsergenerator ist in der Programmiersprache Objective Caml (OCaml) implementiert, die funktionale, imperative und objektorientierte Programmierung unterstützt. Informationen zur Sprache sind auf der Webseite [OCA] des „Institut national de recherche en informatique et en automatique“ (INRIA) zu finden.

Für die Übersetzung des Programms wird das Vorhandensein eines Compilers für OCaml erwartet, der ebenfalls unter [OCA] zu finden ist. Hilfreich ist zudem das Vorhandensein des Hilfsprogramms GNU Make ([MAKb]), durch das der Übersetzungsvorgang automatisiert werden kann.

Angenommen, die Dateien aus dem Ordner `src` der beiliegenden CD-ROM wurden in das Verzeichnis `/bogenlr/src/` auf einem beschreibbaren Datenträger kopiert, so wird der Übersetzungsvorgang durch Aufruf von `make` in diesem Verzeichnis aufgerufen. Im Erfolgsfall befinden sich anschließend im Verzeichnis `/bogenlr/bin/` die ausführbare Programmdatei `bogenlr` und die Datei `bogenfix.c`, die den Code für den fixen Anteil jedes generierten Parsers enthält.

Der Aufruf des Programms, um für eine im selben Dateiverzeichnis befindliche Grammatikdatei `grammar.y` einen Parser zu generieren, lautet dann wie folgt:

```
./bogenlr -l 2 -o grammar.c -r -d grammar.h grammar.y
```

Durch die Zahl hinter der Option `-l` wird festgelegt, wieviele Symbole Lookahead der Parser verwenden soll. Mit `-o` wird der Name der Datei festgelegt, in die der erzeugte Programmcode für den Parser geschrieben werden soll. Durch die Option `-r` wird veranlasst, dass der Generator einen Bericht mit Informationen über die generierten Datenstrukturen für die Grammatiksymbole und -regeln in eine Datei `grammar.y.output` schreibt. Wenn die Option `-d` angegeben ist, werden in eine Datei des angegebenen Namens (hier `grammar.h`) die Definitionen der Terminalsymbole geschrieben, wie sie beispielsweise die von Flex ([FLE]) generierten Programme zur lexikalischen Analyse benötigen. Der letzte Aufrufparameter ist der Name der Datei, in welcher sich die Grammatikspezifikation befindet.

Für eine Grammatik $G = (N, \Sigma, P, S)$ enthält diese Eingabedatei die Deklarationen sämtlicher Terminalsymbole von G sowie die Menge der Produktionen. Ergänzend darf die Datei noch Code in der Programmiersprache C enthalten, der vor und/oder nach den erzeugten Datenstrukturen im erzeugten Parsercode erscheinen soll. Der Aufbau dieser Datei entspricht dem von Eingaben für den $LALR(1)$-Parsergenerator Bison, allerdings mit einigen Einschränkungen. So erlaubt Bison die Angabe von Assoziativitäts- und Präzedenzregeln, mit deren Hilfe der Parser sogenannte Konflikte auflösen kann, wenn andernfalls die nächste Aktion des Parsers nicht eindeutig bestimmt werden kann. Außerdem können in der aktuellen Version von BogenLR zwar semantische Aktionen angegeben werden; diese werden aber ignoriert und nicht in den generierten Parser eingefügt. Dabei beschreibt eine *seman-*

tische Aktion eine Anweisungsfolge, die nach Reduktion anhand einer bestimmten Regel vom Parser durchzuführen ist.

Falls eine Eingabedatei syntaktisch fehlerhaft ist oder die darin beschriebene Grammatik einige einfache Kriterien nicht erfüllt, so bricht BogenLR mit einer Fehlermeldung ab. Eine Grammatik wird dabei zurückgewiesen, falls

- ein Startsymbol ausdrücklich deklariert wurde, das aber durch keine Regel definiert ist,
- Nichtterminale existieren, die zwar auf der rechten Seite von Regeln auftreten, die aber nicht durch eigene Regeln definiert sind,
- nicht alle Nichtterminalsymbole erreichbar sind, d. h. falls es Symbole gibt, für die zwar Regeln definiert sind, die aber in keiner rechten Regelseite Verwendung finden.

Im Erfolgsfall ist das Ergebnis eine Datei, die für die Grammatik G einen Parser in Form von C-Code enthält, der im folgenden Abschnitt genauer erläutert wird.

4.2 Erläuterung der generierten Parser

Für die von BogenLR generierten Parser wurde die Programmiersprache C gewählt. Dadurch ist es möglich, für die Generierung eines Scanners das Programm Flex zu verwenden, das ebenfalls Programmcode in der Sprache C ausgibt. Außerdem erlaubt die Wahl von C einen direkten Vergleich der von BogenLR generierten Parser mit denen, die Bison erzeugt. Entsprechende Vergleichstests folgen in Kapitel 5.

Die erzeugten Parser bestehen aus einem variablen Teil, der von der Eingabegrammatik abhängt, und einem grammatikunabhängigen Teil, der aus Funktionen für die Expansionen, Reduktionen, Leseschritte und die Verwaltung der benötigten Datenstrukturen besteht.

Der in Abhängigkeit von einer Grammatik $G = (N, \Sigma, P, S)$ erzeugte Teil des Parsers enthält Datenstrukturen für die Symbolmenge $V = N \cup \Sigma$ der Grammatik und für die Items, die sich aus der Produktionenmenge P ergeben. Für die Implementierung im Rahmen dieser Arbeit wurde für die Datenstrukturen des Parsers der Aufbau gewählt, der im folgenden beschrieben wird, zusammen mit dem daraus resultierenden Platzbedarf:

Terminalsymbole: Für die Terminalsymbole wird ein Aufzählungstyp erstellt, in dem jedem Symbol eine eindeutige Nummer zugewiesen ist. Diese Nummern werden vom Scanner als Eingabe für den Parser erwartet.

Geht man davon aus, dass das Alphabet Σ keine Terminalsymbole enthält, die nicht in mindestens einer rechten Regelseite auftreten, gilt eine grobe Abschätzung von $|\Sigma| \leq |G|$. Bei $|\Sigma|$ Symbolen ist für die Terminalsymbole also ein Speicherplatzbedarf von $O(|G|)$ anzusetzen.

Variablen: Für jede Variable A ist angegeben, welche Items der Form $[A \rightarrow \cdot\alpha]$, $\alpha \in V^*$, es gibt, d. h. in welchen Items die linke Regelseite aus der Variablen A besteht. Dies beschreibt die Menge all derjenigen Items, für die im Graphen $\mathcal{G}$ neue Knoten angelegt werden müssen, sobald eine Expansion anhand des Symbols A stattfindet.

Diese Teilmenge der Menge aller Items enthält so viele Elemente, wie die Grammatik Produktionen enthält. Die Gesamtzahl aller Items beträgt $|G|$, also gibt es weniger als $|G|$ Einträge, die sich auf die $|N|$ Listen verteilen. Somit ergibt sich für den Platzbedarf der Nichtterminalsymbole eine obere Schranke in $O(|G|)$.

Items: Für jedes Item sind angegeben: Typ (Terminal, Nichtterminal oder ε) und Nummerncode des Symbols unmittelbar hinter dem Punkt, sowie die $FIRST_k$-Menge für die rechte Seite des Items.

Die Anzahl der Items, die für die Grammatik G erzeugt werden, beträgt $|G|$. Für die Angabe von Typ und Code des Symbols hinter dem Punkt kann ein Platzbedarf von $O(1)$ je Item angesetzt werden. Die $FIRST_k$-Menge kann maximal soviele Strings enthalten, wie die Menge $\Sigma^{\leq k}$ enthält, also $O(|\Sigma|^k)$. Jeder einzelne dieser Strings hat eine Länge $l \leq k$. Somit kann der Speicherbedarf für ein einzelnes Item mit $O(1 + |\Sigma|^k \cdot k)$ angesetzt werden, was für die Menge aller Items einen Platzverbrauch von $O(|G| + |G| \cdot |\Sigma|^k \cdot k)$ ergibt.

Insgesamt ergibt sich für die erzeugten Datenstrukturen ein Platzbedarf von

$$O(|G| + |G| \cdot |\Sigma|^k \cdot k).$$

Da das Alphabet eine Größe von $|\Sigma| \geq 1$ hat und $k \geq 1$ vorausgesetzt wird, läßt sich dies zusammenfassen zu folgendem Satz:

Satz 3: *Die asymptotische Größe des Parsers für eine Grammatik $G = (N, \Sigma, P, S)$, der im Rahmen dieser Arbeit implementiert wurde, beträgt*

$$O(|G| \cdot |\Sigma|^k \cdot k).$$

Dabei gibt $|\Sigma|$ die Anzahl der Terminalsymbole in G an und k die Länge des Lookaheads. □

Der Unterschied zu Blums Abschätzung von $O(|G| + \#LA \cdot |V| \cdot 2^k \cdot k \log k)$ im Abschnitt 7.1 von [Blu10a] (vgl. Abschnitt 2.2, Seite 11) liegt darin begründet, dass bei der aktuellen Implementierung des Parsergenerators auf die Vorabgenerierung der Präfixlisten verzichtet wurde.

Die Funktionen im fixen Teil des Parsers implementieren genau die Algorithmen, die in Kapitel 3 beschrieben sind. Dabei wird für die rückwärts-topologische Suche nach gültigen Endknoten das Verfahren für die zyklenignorierende Suche aus Algorithmus 3.14 angewandt. Von AdjustGraph (Algorithmus 3.6) konnte dadurch, wie auf Seite 44 beschrieben, die modifizierte Variante ohne die Entfernung unnützer Zyklen implementiert werden. Alle Algorithmen wurden dabei in ihrer allgemeinen Version verwendet, d. h. ohne Berücksichtigung der Vereinfachungen, die im Fall $k = 1$ möglich sind.

Die Knoten des Graphen werden vom Parser als Struktur mit folgendem Inhalt gespeichert:

- Nummer des repräsentierten Items oder Nichtterminalsymbols,
- Menge von Zeigern auf die Vorgängerknoten in $\mathcal{G}$,
- Menge von Zeigern auf die Nachfolgerknoten in $\mathcal{G}$,
- Index der rückwärts-topologischen Suche, während derer der Knoten zuletzt besucht wurde,
- Präfixvektor als Array der Länge k mit Zeigern auf die assoziierten Endknoten,
- Bitvektor der Länge k, der für einen Itemknoten angibt, wie der Präfixvektor des nachfolgenden Nichtterminalknotens zur Zeit des letzten Besuchs beschaffen war.

Da ein Zeiger in der Programmiersprache C eine konstante Größe besitzt, beträgt somit der Platzbedarf für einen Knoten des Graphen $O(k)$. Wie in Abschnitt 3.1

argumentiert, ist auch die Gesamtanzahl der Kanten im Graphen mit $O(|G| \cdot n)$ beschränkt, so dass insgesamt gilt:

Satz 4: *Der im Rahmen dieser Diplomarbeit implementierte Parser für eine Grammatik G benötigt zur Laufzeit einen asymptotischen Speicherplatz von*

$$O(k \cdot |G| \cdot n).$$

Dieser Platz wird für die Verwaltung eines Graphen aufgewendet. Dabei ist k die Länge des Lookaheads und n die Länge einer Eingabefolge x. □

Der Graph wird dann repräsentiert durch eine Liste von Startknoten. Diese Liste enthält die meiste Zeit über genau einen Knoten, nämlich den für das Startitem $[S' \rightarrow \cdot S]$. Wenn aufgrund einer Reduktion das Item $[S' \rightarrow S\cdot]$ entstanden ist, wird dieses ebenfalls in die Liste der Startknoten aufgenommen, sofern die Eingabe verbraucht ist. Dadurch wird am Ende der Hauptschleife des Parsers erkannt, dass die Eingabe x zur Sprache $L(G)$ gehört, die durch die Grammatik G definiert ist, vgl. Algorithmus 3.1.

Sämtliche weiteren Knotenmengen wurden als Hashtafeln implementiert, um die mittlere Suchzeit nach vorhandenen Knoten gegenüber einer Implementierung mit verketteten Listen zu beschleunigen. Das verwendete Verfahren ist in [Blu01] beschrieben als Hashing mit Kollisionsbehandlung mittels verketteter Listen. Dort wird auch gezeigt, dass für eine derartige Implementierung der Erwartungswert für die mittleren Kosten einer Operation $O(1)$ ist. Eine *Hashtafel* T der Größe m ist eine Tabelle mit m Zeilen, wobei der Zugriff auf die Zeilen über eine ganzzahlige Hashfunktion $h : U \rightarrow [0, m-1]$ gesteuert wird. Jedes Element x aus einer Datenmenge U wird durch h einer Tabellenzeile $T[h(x)]$ zugeteilt. Dabei kann es zu einer *Kollision* kommen, einer Situation, bei der zwei Elemente $x \neq y \in U$ auf die gleiche Tabellenzeile verteilt werden sollen, für die also $h(x) = h(y)$ gilt. Bei der Kollisionsbehandlung mittels verketteter Listen besteht jede Tabellenzeile aus einer Liste von Elementen aus U. Wenn ein Element $x \in U$ in T eingefügt, dort gesucht oder aus T entfernt werden soll, wird die entsprechende Operation auf der Liste $T(h(x))$ durchgeführt. In Abbildung 4.3 ist die oben beschriebene Situation skizziert. Sei n die aktuelle Anzahl von Elementen in T. Wenn der *Belegungsfaktor* $\beta = n/m$ zu 1 wird, werden alle Einträge in eine neue Tabelle der Größe $2m$ verschoben, um die Zugriffzeiten nicht zu groß werden zu lassen. Sinkt β auf den Wert $1/4$, so wird die

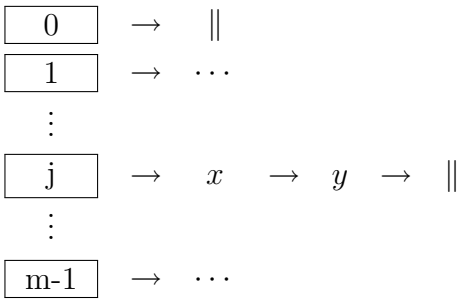

Abbildung 4.3: Hashtafel mit Kollisionsbehandlung mittels verketteter Listen.

Tabellengröße wieder halbiert. In der Implementierung der Parser wurde allerdings von dieser Halbierung abgesehen.

Im Erfolgsfall liefert die Parser-Funktion keine Ausgabe, sondern nur den Rückgabewert `0`. Im Fehlerfall wird das Programm abgebrochen und eine Fehlermeldung ausgegeben. Auf Wunsch, d. h. nach Definition bestimmter Makros beim Übersetzen, kann der Parser jedoch auch Ableitungsbäume erzeugen. Dabei besteht die Wahl zwischen der Ausgabe eines Ableitungsbaumes für die gesamte Eingabe am Ende des Parsevorgangs und der Ausgabe der Menge von partiellen Ableitungsbäumen für jede Phase, d. h. jeweils nach dem Verbrauchen eines Eingabesymbols. Dazu wird auf dem Standardausgabekanal des Systems eine Ausgabe erzeugt, die dem Programm Dot aus dem Programmpaket Graphviz als Eingabe dienen kann. Die Dokumentation zu Graphviz ist über die Website [GRA] erreichbar. Zum Nutzen dieser Option muss der von BogenLR erzeugte Parsercode auf eine der folgenden beiden Weisen übersetzt werden: Wird nur die Ausgabe des endgültigen Ableitungsbaums am Ende der Syntaxanalyse gewünscht, so lautet die Befehlsfolge, sofern der erzeugte Parser den Namen `parser.c` trägt und ein Scanner bereits als compilierte Objektdatei `parser.yy.o` bereitsteht:

```
gcc -DPARSE_TREE -c parser.c
gcc -oparser parser.yy.o parser.o
```

Ist eine Ausgabe des Waldes aus partiellen Ableitungsbäumen am Ende jeder Phase gewünscht, startet man die Übersetzung mittels

```
gcc -DPARSE_TREE -DPARSE_TREE_PHASES -c parser.c
gcc -oparser parser.yy.o parser.o
```

Details dazu, wie die Generierung dieser Bäume im Parser stattfindet, sind in der Projektdokumentation auf der beiliegenden CD-ROM im Verzeichnis `doc` zu finden.

Im nächsten Kapitel folgt eine Beschreibung der Vergleichstests, mit denen die Praxisrelevanz des implementierten Parsermodells überprüft werden sollte. Dabei wird für verschiedene Grammatiken und Eingaben getestet, wie effizient die implementierten Hashtafeln sind und wie sich Parsergröße, Laufzeiten und Speicherbedarf zur Laufzeit der von BogenLR generierten Parser mit den Resultaten anderer Generatoren verhalten.

5 Praktische Tests der erzeugten Parser

Im Rahmen dieser Diplomarbeit wurden die Parser, die BogenLR für einige Testgrammatiken erzeugt, mit den Parsern verglichen, die für die selben Grammatiken von den Parsergeneratoren Bison und MSTA erzeugt werden. Sämtliche Zeitmessungen fanden auf einem Rechner mit einem Intel Pentium 4-Prozessor mit 3200 MHz Taktfrequenz statt, der mit 1 GB Arbeitsspeicher ausgestattet ist. Als Betriebssystem fand NetBSD 5.1 Anwendung, für das sich Dokumentation und herunterladbare Installationsdateien im Web unter [BSD] finden lassen.

Bison ([BIS]) ist ein Parsergenerator, der aus einer textuellen Beschreibung einer kontextfreien Grammatik einen $LALR(1)$-Parser erzeugt. Dabei wird ein fixes Parserskelett aus Funktionen, die in der Programmiersprache C implementiert sind, um Tabellen ergänzt, die die Zustandsübergänge des $LALR(1)$-Automaten für die Eingabegrammatik repräsentieren. Das Eingabeformat für Bison diente auch bei der Entwicklung von BogenLR als Vorbild.

Ein weiterer Parsergenerator, MSTA, wurde von Vladimir Makarov im Rahmen von COCOM entwickelt, eines Satzes von Werkzeugen zur Erstellung von Compilern. Informationen und Quellcode-Dateien zu COCOM und MSTA sind unter [Maka] zu finden. Als Eingabe für MSTA können Grammatikspezifikationen im selben Format wie für Bison dienen, bei Bedarf ergänzt um einige MSTA-spezifische Erweiterungen. Als Ergebnis des Generationsvorgangs entsteht Programmcode für einen Parser in einer der Programmiersprachen C oder C++. MSTA kann mit $LALR(k)$- und $LR(k)$-Grammatiken umgehen, wobei zur Verkleinerung der Parsertabellen einige Optimierungen durchgeführt werden. Beispielsweise prüft der Generator, ob für eine Grammatik G die vom Benutzer vorgegebene Lookahead-Länge k tatsächlich nötig ist, um Eingaben aus der Sprache $L(G)$ korrekt zu parsen. Sollte auch ein kleinerer Wert k' genügen, werden entsprechend kürzere Lookahead-Strings und somit kleinere Parsertabellen erzeugt.

Bei den Tests mit einer Grammatik für die Programmiersprache Pascal und einer modifizierten Pascal-Grammatik (beide siehe Abschnitt 5.1) wurde das Programm Flex [FLE] zur Erzeugung eines Scanners benutzt. Dieser Scanner führt auf den Pascal-Programmen, die den Parsern als Testeingaben dienen sollen, die lexikalische Analyse durch und wandelt die Eingaben in Folgen von Tokens um, die die Parser weiterverarbeiten können.

Im folgenden Abschnitt werden zunächst die Grammatiken vorgestellt, die für die Vergleichstests der Parsergeneratoren Anwendung fanden. Anschließend werden in Abschnitt 5.2 die durchgeführten Tests genauer beschrieben und die Ergebnisse aufgeführt und erläutert.

5.1 Die Testgrammatiken

Sämtliche Grammatiken, Eingabedateien und Shellskripte zur Testautomatisierung, die in diesem Kapitel erwähnt werden, befinden sich auf der beiliegenden CD-ROM im Verzeichnis `test`.

Für einen Vergleich zwischen den von BogenLR erzeugten Parsern und den Parsern, die Bison und MSTA erzeugen, wurde eine $LALR(1)$-Grammatik G_P für die Programmiersprache Pascal verwendet. Eine Beschreibung dieser von Niklaus Wirth entworfenen Programmiersprache liefert [JW78]. Die im Rahmen dieser Tests verwendeten Grammatik- und Scannerspezifikationen `pascal.y` und `pascal.l` für das standardisierte ISO-Pascal sind auf der Website [Moo] zu finden, zusammen mit Programmbeispielen und weiteren Materialien zur Sprache. Für die Verwendung im Rahmen dieser Diplomarbeit wurden die beiden Dateien lediglich an einigen Stellen korrigiert. Die Pascal-Programme, die als Testeingaben für die generierten Parser fungierten, stammen ebenfalls aus diesem Internetangebot. Es handelt sich dabei um die fünf Dateien `basics.pas`, `pascals.pas`, `plzero.pas`, `prettyp.pas` und `startrek.pas`.

Eine weitere Grammatik wurde erzeugt, um die Tauglichkeit des Generators BogenLR für Lookahead-Längen $k > 1$ zu testen und das Resultat mit dem Erzeugnis von MSTA zu vergleichen. Zu diesem Zweck wurde die Pascal-Grammatik zu einer $LALR(2)$-Grammatik G_{P2} modifiziert. Dazu wurden die Produktionen in G_P, die gültige Programmanweisungen in Pascal definieren, gemäß Abbildung 5.1 ergänzt, um eine neue Spezifikationsdatei `pascal2.y` zu erhalten. Diese Regeln entsprechen einer Beispielgrammatik, die in Kapitel 9.6.2 von [GJ08] vorgestellt wird. Dadurch

```
statement :
open_statement {}
| closed_statement {}
| foo_statement {} /* NEU */
;

foo_statement :
cap_a AAA {}
| cap_b BBB {}
| cap_c EEE CCC {}
| cap_d EEE DDD {}
;

cap_a : QQQ cap_e {} ;
cap_b : QQQ cap_e {} ;
cap_c : QQQ {} ;
cap_d : QQQ {} ;
cap_e : EEE {} ;
```

Abbildung 5.1: Modifikationen an der Pascal-Grammatik zwecks Erzeugung einer $LALR(2)$-Grammatik.

werden in einem Pascal-Programm folgende Befehlszeilen zusätzlich erlaubt:

```
QQQ EEE AAA;
QQQ EEE BBB;
QQQ EEE CCC;
QQQ EEE DDD;
```

Einige Tests mit der Grammatik G_{P2} wurden mit den Eingaben für G_P benutzt. Zusätzlich wurde ein Satz von fünf neuen Eingabedateien `basics2.pas`, `pascals2.pas`, `plzero2.pas`, `prettyp2.pas` und `startrek2.pas` erstellt, indem die oben angegebenen neuen Befehlszeilen an unregelmäßigen Positionen in die Originaleingaben eingefügt wurden.

Weitere Tests wurden mit einer Familie G_n von $LR(0)$-Grammatiken durchgeführt, die Ukkonen in [Ukk83] und [Ukk85] erwähnt. Diese Grammatiken haben die

Form $G_n = (N_n, \Sigma_n, P_n, S)$ mit

$$\begin{aligned} P_n:\ & S \to A_i && (1 \le i \le n),\\ & A_i \to a_j A_i && (1 \le i \ne j \le n),\\ & A_i \to a_i B_i \mid b_i && (1 \le i \le n),\\ & B_i \to a_j B_i \mid b_i && (1 \le i, j \le n). \end{aligned}$$

Die Größe einer Grammatik G_n beträgt $m_n = |G_n| = 6n^2 + 6n$. In [Ukk83] wird bewiesen, dass eine Konstante c existiert, so dass ein kanonischer $LR(k)$-Parser für G_n eine Größe von $\Theta(2^{c\sqrt{m_n}})$ hat. Die zugehörigen Dateien mit den Grammatikspezifikationen tragen die Namen `g01.y` bis `g10.y`. Außerdem wird der Quellcode für ein C-Programm `create-grammar` zur Verfügung gestellt, mit dem für beliebige Werte von n eine Spezifikation für G_n automatisch erzeugt werden kann.

Im folgenden Abschnitt werden die Parser, die von Bison, BogenLR und MSTA für eine Grammatik G erzeugt wurden, mit Bison-G, BogenLR-G bzw. MSTA-G bezeichnet. Der Parser, den MSTA für G_P erzeugt, heißt also beispielsweise MSTA-G_P. Im nachfolgenden Abschnitt werden die durchgeführten Tests beschrieben.

5.2 Durchführung der Vergleichstests

Um die Praxistauglichkeit von BogenLR zu überprüfen, fanden einige Tests mit den Grammatiken statt, die im vorigen Abschnitt vorgestellt wurden. Ein erster, in Unterabschnitt 5.2.1 beschriebener Test sollte zeigen, wie effizient die Knotenverwaltung mittels Hashtafeln in der Praxis ist. Anschließend folgt in Unterabschnitt 5.2.2 ein Größenvergleich der Parser, die BogenLR, Bison und MSTA für die verschiedenen Grammatiken erzeugen. In der nächsten Testreihe wurden die Laufzeiten der Parser für einige Eingaben gemessen, siehe Unterabschnitt 5.2.3, und schließlich fand ein Vergleich des tatsächlichen Speicherplatzbedarfs zur Laufzeit der Parser statt, dem sich Unterabschnitt 5.2.4 widmet. Im Folgenden werden die durchgeführten Tests genauer beschrieben und die Ergebnisse vorgestellt.

5.2.1 Kosten von Operationen auf Hashtafeln

Zunächst fanden einige Stichproben statt, um die Annahme aus Abschnitt 4.2 zu überprüfen, dass die mittleren Kosten einer Operation auf einer Hashtafel $O(1)$ betragen. Dazu wurde nach jeder Operation, die die Belegung einer Hashtafel verän-

dert, gemessen, wie viele Elemente die Tafel insgesamt enthält, und wie viele Felder der Tafel belegt sind. Operationen, nach denen eine Messung stattfand, waren:

- Hinzufügen eines Elementes zu einer Hashtafel.
- Entfernen eines Elementes aus einer Hashtafel, sofern die Ergebnistafel nicht leer ist.
- Verdoppeln der Größe einer vollen Hashtafel.

Sei in einer Messung c die Gesamtanzahl der Elemente einer Hashtafel und l die Anzahl der belegten Felder in dieser Tafel. Dann gibt der Quotient $d := c/l$ an, wieviele Elemente ein belegtes Feld der betrachteten Tabelle durchschnittlich enthält. Ideal ist ein Wert von $d = 1$, denn dann gibt es keine Kollisionen in der Hashtafel, so dass ein Elementzugriff tatsächlich in Zeit $O(1)$ möglich ist.

Mit dem Parser BogenLR-G_P wurde für jede der fünf Eingabedateien eine Folge von d-Werten für einen ganzen Parserlauf erzeugt. Für jede dieser Folgen wurde anschließend anhand verschiedener Schwellenwerte $s \in \{2, 3, 4, 5\}$ gezählt, wie viele Werte von d größer als s sind und wie viele kleiner oder gleich s sind. Seien $gr_s = |\{d \mid d > s\}|$ und $kl_s = |\{d \mid d \leq s\}|$ die ermittelten Anzahlen. Schließlich wurden daraus die Quotienten

$$q_s := \frac{gr_s}{gr_s + kl_s}$$

gebildet, die angeben, welcher Anteil der durchgeführten Änderungsoperationen eine Hashtafel zurückläßt, auf der zum Finden eines Elementes mehr als s Zugriffe nötig sind.

Das Testskript, das die beschriebenen Messungen durchführt, befindet sich in der Datei `hashtest.csh`. Für seine Ausführung wird vorausgesetzt, dass beim Übersetzen des Parsers das Makro `DEBUG_ALL` definiert war und die Ausgabe eines Parserlaufes in eine Datei mit der Namensendung `.log` umgeleitet wurde, beispielsweise `datei.log`. Der Aufruf des Skripts geschieht dann mit folgendem Befehl:

```
./hashtest.csh datei
```

Das Ergebnis wird dann in eine Datei `datei.hash` geschrieben.

Eine Übersicht über die für BogenLR-G_P gemessenen Werte liefert Tabelle 5.1. Sämtliche Prozentwerte wurden dabei auf eine Nachkommastelle gerundet. Aus der Tabelle geht hervor, dass für die Testeingaben weniger als 14% der Änderungsoperationen eine Hashtafel mit einer durchschnittlichen Feldbelegung von 2 oder größer

Tabelle 5.1: Prozentsatz der Änderungsoperationen, die in den Hashtafeln des Parsers BogenLR-G_P eine durchschnittliche Zeilenbelegung $d > s$ verursachen.

Eingabe	Anzahl Token	$s = 2$	$s = 3$	$s = 4$	$s = 5$
basics.pas	5 620	12,6%	9,1%	8,7%	8,6%
pascals.pas	14 617	12,9%	9,0%	8,7%	8,6%
plzero.pas	3 467	12,9%	9,1%	8,7%	8,5%
prettyp.pas	4 081	13,8%	9,5%	9,1%	9,0%
startrek.pas	5 229	13,9%	9,4%	8,9%	8,8%

verursachen. Für Schwellenwerte von 3 oder höher sind es sogar weniger als 10% der Operationen. Eine Abhängigkeit von der Länge der Eingabe ist dabei nicht festzustellen. Zur Optimierung könnte möglicherweise eine andere Strategie zur Kollisionsbehandlung versucht werden. Weitere Ansatzpunkte könnten eine Anpassung der Hashfunktion oder der initialen Tabellengröße sein. Zudem ist nicht auszuschließen, dass ein Teil der beobachteten Operationen während der Speicherbereinigung stattfindet, die sich an den eigentlichen Parsevorgang anschließt. Durch eine Modifikation des Speichermanagements wäre an dieser Stelle möglicherweise eine Zeitersparnis möglich. Würde nämlich nur eine beschränkte Anzahl größerer Speicherbereiche am Stück für den Parserlauf reserviert, so könnten diese bei Programmende mit einem Zeitaufwand von $O(1)$ wieder freigegeben werden.

Zusammenfassend läßt sich aber sagen, dass in den gemessenen Stichproben nach 90% aller Änderungsoperationen ein Zugriff auf Elemente der modifizierten Hashtafel in $O(1)$ Schritten möglich ist, wie in Abschnitt 3.7 angenommen wurde. Somit wären weitere Versuche, die Datenstrukturen zu optimieren, zwar versuchenswert, jedoch nicht zwingend nötig, da die weiteren Operationen zur Verwaltung des Graphen und für die rückwärtstopologische Suche den größten Teil des Zeitverbrauchs der Parser ausmachen.

5.2.2 Vergleich der Parsergrößen

Für einen Vergleich der Größen der generierten Parser wurde die Dateigröße des jeweils erzeugten C-Programmcodes in KByte ermittelt. Diese Werte sind in Tabelle 5.2 eingetragen. Für die Pascal-Grammatik G_P hat der mit BogenLR erzeugte Parser die 4,9-fache Größe des Parsers Bison-G_P und die 2,5-fache Größe von

Tabelle 5.2: Größen der generierten Parser in KByte (1 KByte = 1024 Byte).

	BogenLR	Bison	MSTA
G_P	368	75	149
G_{P2}	484	–	168
G_1	143	39	15
G_2	149	41	19
G_3	157	45	27
G_4	167	54	45
G_5	181	76	92
G_6	197	127	214
G_7	216	255	540
G_8	237	568	1 395
G_9	261	1 343	3 559
G_{10}	288	3 199	8 760

MSTA-G_P. Für G_{P2} können nur BogenLR-G_{P2} und MSTA-G_{P2} verglichen werden, da die $LALR(1)$-Tabelle, die Bison erzeugt, zwei Konflikte enthält. Der von Bison erzeugte Parser der Größe 98 KByte würde also möglicherweise in manchen Situationen unkorrekte Ergebnisse liefern, da er die korrekte Entscheidung über den nächsten Schritt nicht in jeder Situation eindeutig erkennen könnte. Die Größe von BogenLR-G_{P2} beträgt ein 2,9-faches der Größe von MSTA-G_{P2}. Für diesen deutlichen Größenvorteil von Bison und MSTA gegenüber BogenLR kann es mehrere Erklärungen geben. Zum einen ist die vorliegende Pascal-Grammatik für den Gebrauch mit einem $LALR(1)$-Parsergenerator optimiert, und die gegenüber G_P nur geringfügig modifizierte Grammatik G_{P2} gehört zur Klasse $LALR(2)$. Von den speziellen Eigenschaften von $LALR(k)$-Grammatiken gegenüber $LR(k)$-Grammatiken macht BogenLR keinen Gebrauch. Zum anderen bestehen die Resultatparser von Bison und MSTA aus kompakten Tabellen und Funktionen zum Zugriff darauf. Von BogenLR erzeugte Parser hingegen müssen zur Laufzeit einen Graphen verwalten und enthalten daher einen großen Anteil von Funktionen für die dabei anfallenden Aufgaben.

Für die Grammatikfamilie $G_n,\ n \in \mathbb{N}$, stellt sich die Situation anders dar: Für Werte bis $n = 5$ erzeugen sowohl Bison als auch MSTA kleinere Parser als BogenLR. Man kann aber bereits erkennen, dass das Wachstum der Parser für BogenLR-G_n

Tabelle 5.3: Vergleich der für die Parsergröße dominanten Werte „Itemanzahl“ für die von BogenLR erzeugten Parser und „Zustandsanzahl“ für die von Bison und MSTA erzeugten Parser.

Grammatik	BogenLR: Items	Bison: Zustände	MSTA: Zustände
G_P	765	410	401
G_{P2}	793	424	416
G_1	14	10	8
G_2	38	25	25
G_3	74	54	54
G_4	122	111	111
G_5	182	228	228
G_6	254	477	477
G_7	338	1 018	1 018
G_8	434	2 203	2 203
G_9	542	4 800	4 800
G_{10}	662	10 473	10 473

geringer ist als das der anderen Parser. Bereits ab $n = 6$ sind die MSTA-Parser größer als die BogenLR-Parser und wachsen für jeden Schritt von n nach $n + 1$ auf mehr als die doppelte Größe an. Bei Bison ist der gleiche Effekt ab einem Wert von $n = 7$ zu erkennen. Bei einer Stichprobe für $n = 20$ erzeugte BogenLR den Parser BogenLR-G_{20} mit einer Größe von 714 KByte innerhalb von drei Sekunden, wogegen Bison nach 24 Stunden noch kein Ergebnis lieferte und MSTA die Berechnung nach ca. 7 Minuten wegen Speichermangels abbrach. Bei dieser Grammatikfamilie zeigt sich also, dass die von Ukkonen bewiesene Zustandsanzahl eines $LALR$-Parsers von mindestens $2^{c\sqrt{|G_n|}}$ für eine Konstante c im praktischen Einsatz zu groß sein kann.

Die Parser, die BogenLR erzeugt, implementieren im Gegensatz zu Bison und MSTA keinen Automaten. Stattdessen machen hier die Datenstrukturen für die Items und deren $FIRST_k$-Mengen den größten Anteil des generierten Programmcodes aus. Die Größe der Parser, die Bison und MSTA erzeugen, wird dagegen hauptsächlich durch die Anzahl der Zustände des repräsentierten Automaten bestimmt. Da diese Zustände auf Basis der Items und der $FIRST_k$-Mengen der Grammatiksymbole konstruiert werden, liegt es nahe, die Itemanzahlen der von BogenLR erzeugten Parser mit den Zustandsanzahlen der Bison- und MSTA-Parser zu ver-

Tabelle 5.4: Vergleich der durchschnittlichen Laufzeiten der Parser für G_P in Millisekunden für verschiedene Eingaben.

Eingabedatei	Anzahl Token	BogenLR-G_P	Bison-G_P	MSTA-G_P
basics.pas	5 620	242,62	3,12	2,77
pascals.pas	14 617	667,34	7,12	6,34
plzero.pas	3 467	156,52	1,84	1,67
prettyp.pas	4 081	178,46	2,62	2,36
startrek.pas	5 229	234,53	2,47	2,17

gleichen. Dies geschieht in Tabelle 5.3, aus welcher derselbe Effekt wie in Tabelle 5.2 abzulesen ist: Für $G = G_P$ und $G = G_{P2}$ liegt die Zustandsanzahl von Bison-G und MSTA-G unter der Itemanzahl von BogenLR-G, ebenso für G_1 bis G_4. Ab G_5 kehrt sich dieses Verhältnis jedoch um, sodass für $n \geq 5$ die Parser, die BogenLR erzeugt, stets weniger Items enthalten, als für die Parser, die Bison und MSTA erzeugen, Zustände konstruiert werden.

5.2.3 Vergleich der Laufzeiten der Parser

Zum Messen der Programmlaufzeiten der Parser wurde ein Skript `times.csh` verwendet, das für jede Eingabe nach vier unprotokollierten Parserläufen die Zeit für 100 Durchläufe des Parsers misst. Dieser Wert wurde anschließend durch 100 geteilt, um die durchschnittliche Zeit je Parserlauf zu bestimmen. Zur Zeitmessung wurde dabei das Kommando `time` des Kommandointerpreters `tcsh` verwendet, dessen Ausgabe das folgende Format hat:

```
0.017u 0.172s 0:02.29 7.8%      0+0k 0+0io 0pf+0w
```

Die ersten beiden Werte in der Ausgabe von `time` geben die Zeit für die Ausführung des eigentlichen Programms (Wert u) und den Zeitbedarf für durch das Programm verursachte Systemaufrufe (Wert s) an. Die Summe dieser beiden Werte bildet das Ergebnis der Messung.

Die Ergebnisse der Zeitmessungen für Bison-G_P, BogenLR-G_P und MSTA-G_P sind in Tabelle 5.4 zusammengefasst. Tendenziell führen für die gegebenen Eingabedateien bei allen drei Parsern längere Eingaben auch zu längeren Laufzeiten der Parser. Bei Bison-G_P und MSTA-G_P gibt es allerdings jeweils eine Ausnahme bei

Tabelle 5.5: Vergleich der durchschnittlichen Laufzeiten der Parser für G_{P2} in Millisekunden für verschiedene Eingaben.

Eingabedatei	Anzahl Token	BogenLR-G_{P2}	MSTA-G_{P2}
basics.pas	5 620	297,08	2,76
pascals.pas	14 617	798,62	6,27
plzero.pas	3 467	189,65	1,65
prettyp.pas	4 081	211,13	2,33
startrek.pas	5 229	279,05	2,18
basics2.pas	5 664	300,07	2,77
pascals2.pas	14 669	798,49	6,25
plzero2.pas	3 495	190,30	1,67
prettyp2.pas	4 125	212,86	2,36
startrek2.pas	5 297	281,54	2,19

den Messwerten zu den Dateien `prettyp.pas` und `startrek.pas`. Somit liegt der Schluß nahe, dass nicht nur die Länge einer Eingabe über die Zeit für einen Parserlauf bestimmt, sondern auch ihr Aufbau. Die deutlich höheren Laufzeiten des BogenLR-Parsers im Vergleich zu den Parsern, die Bison und MSTA erzeugen, sind durch den größeren Aufwand erklärbar, den BogenLR-G_P betreiben muss. Während bei Bison und MSTA die nächste Aktion des Parsers in Abhängigkeit vom aktuellen Parserzustand und dem Lookahead lediglich in einer Tabelle nachgeschlagen werden muss, muss BogenLR-G_P in jedem Schritt einen Teil des Graphen durchmustern oder manipulieren. Zu vergleichbaren Ergebnissen führten auch die Messungen, die für G_{P2} mit den Resultatparsern von BogenLR und MSTA durchgeführt wurden. Die ermittelten Laufzeiten sind sowohl für die Originaleingaben als auch für die modifizierten Versionen in Tabelle 5.5 zu finden.

Eine systematische Laufzeitmessung für die Grammatik-Familien $G_n, n \in \mathbb{N}$, konnte aus den nachfolgend erläuterten Gründen nicht durchgeführt werden: Mit den für G_{10} generierten Parsern wurde eine Stichprobe für eine Eingabe $x = a_2^{9998}a_1b_1$ der Länge 10 000 durchgeführt. Der Parser BogenLR-G_{10} hatte die Eingabe nach 371 ms bearbeitet, aber die beiden anderen Parser terminierten nach jeweils 2 ms mit Fehlermeldungen, nämlich Bison-G_{10} mit `memory exhausted` und MSTA-G_{10} mit `states stack is overfull`.

Tabelle 5.6: Vergleich des Speicherplatzbedarfs der Parser für G_P zur Laufzeit in KByte.

Eingabedatei	Anzahl Token	BogenLR-G_P	Bison-G_P	MSTA-G_P
basics.pas	5 620	16 180	2 868	2 868
pascals.pas	14 617	40 756	2 868	2 868
plzero.pas	3 467	12 084	2 868	2 868
prettyp.pas	4 081	13 108	2 868	2 868
startrek.pas	5 229	16 180	2 868	2 868

5.2.4 Vergleich des Laufzeit-Speicherbedarfs der Parser

Für die Messung des Speicherplatzbedarfs zur Laufzeit wurde jedem Parserlauf ein Programm vorgeschaltet, das bei jeder Speicheranforderung, die mittels der C-Funktion `malloc` stattfindet, den aktuell zugewiesenen Speicherplatz des Parsers bestimmt. Aus diesen Werten bildet das Testprogramm beim Beenden des Parsers das Maximum. Der Quellcode dieses Programms ist auf der beiliegenden CD-ROM als `/src/mmax.c` abgelegt.

In Tabelle 5.6 wird der maximale Speicherplatzbedarf der Parser für G_P verglichen, in Tabelle 5.7 für G_{P2}. Für beide Grammatiken liegt der Platzbedarf der Bison- und MSTA-Parser konstant bei 2 868 KByte. Bei den Parsern, die mittels BogenLR generiert wurden, ist eine Abhängigkeit der Größe des benötigten Speichers von der Eingabegröße zu beobachten. So wächst der Platzbedarf von BogenLR-G_P auf das 4,2-fache bis 14,2-fache des Platzbedarfs von Bison-G_P und MSTA-G_P, und BogenLR-G_{P2} hat einen 4,2- bis 14,6-fachen Speicherverbrauch im Vergleich zu MSTA-G_{P2}. Der Zuwachs an Platzbedarf zwischen BogenLR-G_P und BogenLR-G_{P2} liegt allerdings nur bei maximal 6,3% für die Eingabedatei `basics.pas`. Der deutlich größere Speicherplatzbedarf der LR-Parser ist analog zu den Überlegungen in Unterabschnitt 5.2.3 auch hier durch den Aufwand für die Verwaltung des Graphen erklärbar. Aus denselben Gründen wie dort konnte für die G_n-Grammatiken keine Messung des Laufzeit-Platzverbrauchs erfolgen.

Auch diese Ergebnisse bestätigen, dass $LALR(1)$-Generatoren für entsprechend optimierte Grammatiken Vorteile gegenüber BogenLR aufweisen. Erneut wird erkennbar, dass die Stärke von BogenLR darin liegt, Parser für solche Grammatiken zu erzeugen, für die andere Generatoren keine brauchbaren Resultate liefern.

Tabelle 5.7: Vergleich des Speicherplatzbedarfs der Parser für G_{P2} zur Laufzeit in KByte.

Eingabedatei	Anzahl Token	BogenLR-G_{P2}	MSTA-G_{P2}
basics.pas	5 620	17 204	2 868
pascals.pas	14 617	41 780	2 868
plzero.pas	3 467	12 084	2 868
prettyp.pas	4 081	13 108	2 868
startrek.pas	5 229	16 180	2 868
basics2.pas	5 664	17 204	2 868
pascals2.pas	14 669	41 780	2 868
plzero2.pas	3 495	12 084	2 868
prettyp2.pas	4 125	13 108	2 868
startrek2.pas	5 297	16 180	2 868

5.2.5 Testfazit

Die Tests der vorigen Unterabschnitte haben gezeigt, dass es Situationen gibt, in denen Parsergeneratoren wie Bison und MSTA überlegen sind, und Situationen, in denen BogenLR überlegen ist. Für Grammatiken, die auf den Gebrauch mit *LALR*-Parsergeneratoren optimiert sind, erzeugen solche Generatoren folgerichtig die effizienteren Parser im Hinblick auf Parsergröße, Laufzeit und Speicherplatzbedarf. Das Nachschlagen der nächsten Parseraktion in einer vorberechneten Tabelle ist für solche Grammatiken mit geringerem Zeit- und Speicherverbrauch zu bewerkstelligen als die Verwaltung eines Graphen. Ist die Struktur der Grammatik jedoch so beschaffen, dass klassische *LALR*-Verfahren einen Automaten mit einer sehr großen Zustandsanzahl hervorbringen, zeigen sich die besonderen Vorteile des erweiterten $LR(k)$-Parsers. Insbesondere erlaubt BogenLR es, Parser für Grammatiken zu generieren, für die andere Werkzeuge versagen oder unbrauchbare Parser erzeugen. Hier überwiegt der Vorteil von BogenLR, dass nicht eine tabellarische Repräsentation einer möglicherweise sehr großen Zustandsübergangsfunktion erzeugt werden muss, sondern nur Strukturen für die Symbolmenge der Grammatik, für die Items und für deren $FIRST_k$-Mengen.

Die Test haben außerdem offenbart, dass Optimierungen des Parsergenerators BogenLR, insbesondere bezüglich Zeit- und Platzeffizienz der erzeugten Parser, wün-

schenswert wären. Dazu bieten sich mehrere Ansatzpunkte an: Als erstes könnte eine andere Strategie zur Behandlung der Zyklen im Graphen dazu führen, dass weniger Knoten in einem Suchlauf besucht werden müssen. Weiterhin bietet es sich an, die Effizienz der verwendeten Datenstrukturen im Hinblick auf deren Zugriffsgeschwindigkeit und den Platzbedarf zu optimieren. Denkbar wäre beispielsweise die Wahl einer anderen Methode zur Verwaltung von Knotenmengen oder eine andere Repräsentation der Daten, die für jeden einzelnen Knoten verwaltet werden müssen. Schließlich wäre zu testen, ob eine Vorausberechnung der Präfixlisten, wie in [Blu10a] vorgeschlagen, zu einer Zeitersparnis im Parsevorgang führen kann.

6 Zusammenfassung und Ausblick

Nach einer Einführung in die Theorie der Syntaxanalyse mittels $LR(k)$-Parser und in das Modell des erweiterten $LR(k)$-Parsers, das Blum in [Blu10a] vorgestellt hat, wurden detaillierte Algorithmen für eine Rechnerimplementierung dieses Verfahrens entwickelt. Dabei wurde eine Einschränkung auf kleine Werte der Lookaheadlänge k gelegt, weshalb das bei Blum vorgeschlagene Vorgehen für große Werte von k nicht beachtet wurde. Da der erweiterte $LR(k)$-Parser auf der Verwaltung eines Graphen basiert, bei dessen Konstruktion Zyklen entstehen können, wurde der Erkennung und Behandlung dieser Zyklen besondere Beachtung geschenkt. Im Rahmen dieser Arbeit wurde ein Parsergenerator BogenLR programmiert, mit dessen Hilfe aus formalen Grammatikspezifikationen Parser erzeugt werden können, die die erwähnten Algorithmen implementieren. Die Entscheidungen, die dabei für bestimmte Vorgehensweisen und Datenstrukturen getroffen wurden, wurden erläutert, insbesondere dort, wo sie von den Vorschlägen in Blums Arbeit abweichen. So wurde ein anderer Weg gewählt, die oben erwähnten Zyklen im Graphen zu behandeln, und auch mit den $FIRST_k$-Mengen für Symbole und Items wird im Generator und den erzeugten Parsern anders verfahren als in [Blu10a].

Im Rahmen dieser Überlegungen wurde in der vorliegenden Arbeit bewiesen, dass die Größe der generierten Parser durch

$$O(|G| \cdot |\Sigma|^k \cdot k).$$

beschränkt ist, sich ihr Zeitaufwand mit

$$O\left((n + \mathrm{ld}(x)) \cdot k^3 \cdot |\Sigma|^k \cdot n \cdot |G|\right)$$

abschätzen läßt und dass sie

$$O(k \cdot |G| \cdot n).$$

viel Speicher zur Laufzeit verbrauchen.

Nach der Implementierung des Parsergenerators BogenLR wurden Vergleichstests für einige Grammatiken und Testeingaben durchgeführt. Dabei wurden die erzeugten

Parser im Hinblick auf Größe, Geschwindigkeit und Speicherbedarf mit denen verglichen, die von den Parsergeneratoren Bison und MSTA erzeugt werden. Ein erstes Resultat dieser Tests betrifft die Effizienz der gewählten Implementierung der diversen Knotenmengen, die während der Verwaltung des Graphen zu unterhalten sind, als Hashtafeln. Dabei zeigte sich bei Stichproben, dass man in der Praxis annehmen kann, dass Anfragen und Manipulationen auf den verwendeten Hashtafeln im Regelfall amortisierte Kosten von $O(1)$ verursachen. Die weiteren Tests haben gezeigt, dass für eine Grammatik, die auf den Gebrauch mit einem $LALR(1)$-Generator optimiert ist oder die durch geringe Modifikationen aus einer solchen optimierten Grammatik entstanden ist, Bison und MSTA gegenüber BogenLR kleinere und schnellere Parser erzeugen, die weniger Speicherplatz benötigen. Wählt man jedoch Grammatiken aus, die im Hinblick auf eine große Anzahl resultierender Zustände eines automatenbasierten Parsers konstruiert wurden, erscheint BogenLR überlegen, da er auch für solche Grammatiken funktionierende Parser erzeugt hat, für die Bison und MSTA keine Parser generieren konnten oder solche, die nicht einsetzbar waren.

Für eine weitere Erforschung des Parsermodells „Erweiterter $LR(k)$-Parser“ bietet es sich an, zunächst einige Designentscheidungen zu hinterfragen, die für die Implementierung im Rahmen dieser Arbeit getroffen wurden. Insbesondere für große Werte von k könnte sich eine andere als die gewählte Strategie zur Behandlung von Zyklen als effizienter erweisen. Wächst der Wert von k, bietet es sich zudem an, für die Suche nach denjenigen Knoten des Graphen, die in der nächsten Aktion des Parsers bearbeitet werden müssen, die modifizierte Strategie zu implementieren, die in Abschnitt 7.2 von [Blu10a] vorgeschlagen wird. Für den Sonderfall $k = 1$ sind außerdem Vereinfachungen des Verfahrens möglich, die zwar erwähnt, aber nicht implementiert wurden. Auch die Wahl der Datenstrukturen wäre dabei zu überdenken, um die Anzahl der Fälle zu reduzieren, in denen ein Zugriff auf die Elemente einer Menge mehr als zwei oder drei Schritte erfordert.

Für einen Einsatz des Parsergenerators in einem praktischen Arbeitsumfeld wäre es außerdem sinnvoll, auch die Implementierung des Generators auf mögliche Optimierungen zu untersuchen, besonders im Hinblick auf die Geschwindigkeit, mit der ein Parser erzeugt wird. Insbesondere bei der Berechnung der $FIRST_k$-Mengen, die der Generator im Programmcode des Parsers hinterlegt, wird in Lehrbüchern wie [Blu01] ein Verfahren beschrieben, dass für große Werte von k schneller zum Ergebnis führen kann. Zudem fehlen dem Parsergenerator und somit den erzeugten Parsern noch einige Funktionen, die für einen praktischen Einsatz nützlich oder gar

unerläßlich sind. So erlaubt die aktuelle Version von BogenLR es zwar, semantische Aktionen zu definieren, die nach einer Reduktion auszuführen sind. In die generierten Parser werden diese jedoch noch nicht übernommen, so dass das Erzeugnis des Generators in der aktuellen Version kein Parser im eigentlichen Sinne ist, sondern nur ein Akzeptor. Würden die Möglichkeiten, die BogenLR bei der Spezifikation einer Grammatik bietet, zudem noch stärker an die von Bison angepasst, so wäre es möglich, in einer bestehenden Arbeitsumgebung den Parsergenerator auszutauschen, ohne dass die Arbeitsweise angepasst werden müsste.

Es könnte weiterhin näher erforscht werden, ob sich die Eigenschaften solcher Grammatiken genauer formalisieren lassen, für die der erweiterte $LR(k)$-Parser die überlegene Strategie zur Syntaxanalyse darstellt. Weitere Tests mit Grammatiken, die in der Praxis bereits Anwendung finden, könnten dazu ebenfalls Aufschluß liefern.

Ein weiterer Ansatz für die Erforschung von Strategien zur Syntaxanalyse wäre die Kombination des erweiterten $LR(k)$-Parsers mit anderen Verfahren. Beispielsweise verwenden die Parser, die im Rahmen des SAIDE-Projekts an der „Federal University of Minas Gerais“ in Brasilien erzeugt werden, keinen fixen Wert für die Lookaheadlänge k, siehe [SAI]. Stattdessen werden stets nur so große Lookaheadstrings betrachtet, wie nötig sind, um einen Konflikt des Parsers aufzulösen. Solche Kombinationen könnten ebenfalls helfen, die Effizienz der erzeugten Parser zu erhöhen.

Abbildungsverzeichnis

Algorithmenverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[AU72] Aho, Alfred V. ; Ullman, Jeffrey D.: *The Theory of Parsing, Translation, and Compiling. Volume I: Parsing.* Prentice-Hall, Inc., 1972

[BIS] Free Software Foundation, Inc.: *Bison - GNU parser generator.* `http://www.gnu.org/software/bison/`, Abruf: 30.06.2011

[Blu01] Blum, Norbert: *Theoretische Informatik. Eine anwendungsorientierte Einführung.* Oldenbourg Wissenschaftlicher Verlag, 2001

[Blu10a] Blum, Norbert: *On LR(k)-parsers of polynomial size.* Version: 2010. `http://theory.cs.uni-bonn.de/blum/papers/lr4.pdf`, Abruf: 30.06.2011

[Blu10b] Blum, Norbert: *On LR(k)-parsers of polynomial size.* in: *Proceedings of the 37th international colloquium conference on Automata, languages and programming: Part II.* Berlin, Heidelberg : Springer-Verlag, 2010 (ICALP'10), 163–174

[BSD] The NetBSD Foundation, Inc.: *The NetBSD Project.* `http://www.netbsd.org/`, Abruf: 30.06.2011

[FLE] The Flex Project: *flex: The Fast Lexical Analyzer.* `http://flex.sourceforge.net/`, Abruf: 30.06.2011

[GCC] Free Software Foundation, Inc.: *GCC, the GNU Compiler Collection.* `http://gcc.gnu.org/`, Abruf: 30.06.2011

[GJ08] Grune, Dick ; Jacobs, Ceriel J.: *Parsing Techniques. A Practical Guide.* Second Edition. Springer Science+Business Media, LLC, 2008

[GRA] *Graphviz – Graph Visualization Software.* `http://www.graphviz.org/`, Abruf: 30.06.2011

[Joh] Johnson, Stephen C.: *Yacc: Yet Another Compiler-Compiler.* in: *Unix Seventh Edition Manual.* http://cm.bell-labs.com/7thEdMan/, Abruf: 30.06.2011

[JW78] Jensen, Kathleen ; Wirth, Niklaus: *PASCAL User Manual and Report.* Second Corrected Reprint of the Second Edition. Springer-Verlag New York Inc., 1978

[Knu65] Knuth, D. E.: *On the Translation of Languages from Left to Right.* in: *Information and Control* 8 (1965), S. 607–639

[Maka] Makarov, Vladimir: *Compiler toolset COCOM.* http://cocom.sourceforge.net/, Abruf: 30.06.2011

[MAKb] Free Software Foundation, Inc.: *GNU Make.* http://www.gnu.org/software/make/, Abruf: 30.06.2011

[Moo] Moore, S.A.: *The ISO 7185 Standard PASCAL Page.* http://www.moorecad.com/standardpascal/index.html, Abruf: 30.06.2011

[OCA] Institut national de recherche en informatique et en automatique: *The Caml Language.* http://caml.inria.fr/, Abruf: 30.06.2011

[SAI] Computer Science Department of the Federal University of Minas Gerais, Brazil: *SAIDE – Syntax Analyser Integrated Development Environment.* http://homepages.dcc.ufmg.br/~leonardo/saide/, Abruf: 30.06.2011

[Ukk83] Ukkonen, Esko: *Lower bounds on the size of deterministic parsers.* in: *Journal of Computer and System Sciences* 26 (1983), Nr. 2, S. 153 – 170

[Ukk85] Ukkonen, Esko: *Upper bounds on the size of LR(k) parsers.* in: *Information Processing Letters* 20 (1985), Nr. 2, S. 99 – 103