

# Interpretation von Grammatiken und deren Simulation mit Zweikellerautomaten

Jan-Martin Kuhnigk

Angefertigt am Institut für Theoretische Informatik der  
Medizinischen Universität zu Lübeck

1998

# Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b>	<b>3</b>
<b>2</b>	<b>Theoretischer Hintergrund</b>	<b>5</b>
2.1	Präliminarien . . . . .	5
2.1.1	Zweikellerautomaten . . . . .	5
2.1.2	Wortersetzungssysteme . . . . .	6
2.1.3	Bewertungen und ihre Folgen . . . . .	7
2.1.4	Church-Rosser-Systeme . . . . .	8
2.2	Charakterisierung der CRL durch sDTPDA . . . . .	9
<b>3</b>	<b>Praktische Umsetzung</b>	<b>16</b>
3.1	Aufgaben des Programms . . . . .	16
3.2	Verwendete Algorithmen . . . . .	16
3.2.1	Die Simulation des Automaten . . . . .	16
3.2.2	Überprüfen eines Wortersetzungssystems auf Konfluenz . . . . .	17
3.2.3	Überprüfen, ob ein Wortersetzungssystem schrumpfend ist . . . . .	18
<b>4</b>	<b>Dokumentation: TPDAsim</b>	<b>19</b>
4.1	Installation . . . . .	19
4.1.1	Systemvoraussetzungen . . . . .	19
4.1.2	Herunterladen der Dateien . . . . .	19
4.1.3	Installation mittels RPM . . . . .	19
4.1.4	Installation der TGZ-Version . . . . .	20
4.2	Bedienungsanleitung . . . . .	20
4.2.1	Starten des Programms . . . . .	20
4.2.2	Neue Grammatik erstellen . . . . .	20
4.2.2.1	Definition der Steuersymbole . . . . .	20
4.2.2.2	Definition der Grammatik . . . . .	22
4.2.3	Neues Wortersetzungssystem erstellen . . . . .	23
4.2.4	Laden eines Wortersetzungssystems . . . . .	24
4.2.5	Speichern eines Wortersetzungssystems . . . . .	24
4.2.6	Beenden des Programms . . . . .	24
4.2.7	Editieren eines Wortersetzungssystems . . . . .	25
4.2.8	Parsieren eines Wortersetzungssystems . . . . .	25
4.2.9	Analysieren eines Wortersetzungssystems . . . . .	26
4.2.10	Anzeigen eines Wortersetzungssystems . . . . .	26
4.2.11	Überprüfen eines Wortersetzungssystems auf Konfluenz . . . . .	27
4.2.12	Überprüfen, ob ein Wortersetzungssystem schrumpfend ist . . . . .	27
4.2.13	Überprüfen eines Wortersetzungssystems auf Determinismus . . . . .	27
4.2.14	Die Simulation des Automaten . . . . .	28
4.2.15	Konfigurieren der Umgebung . . . . .	29
4.3	Auflistung und Erklärung der Fehlermeldungen . . . . .	30
4.3.1	Fehler bei der Dateiverwaltung . . . . .	30
4.3.2	Fehlermeldungen des Parsers . . . . .	31

# 1 Einleitung

Um einen Computer mit Rechenvorschriften oder Algorithmen zu instruieren, stellen Programmiersprachen das wichtigste Hilfsmittel dar. Eine Programmiersprache definiert sich durch ihre Syntax und Semantik. Bevor ein Programm vom Rechner in für ihn direkt verständliche Bitfolgen übersetzt werden kann, muß er die Bedeutung des Programmtextes erfassen, was wiederum voraussetzt, daß dieser der Syntax der verwendeten Programmiersprache genügt. Folglich muß zuerst die Korrektheit der Syntax überprüft werden, was, bedenkt man die noch bevorstehende und zumeist schwierigere Aufgabe der semantischen Interpretation, möglichst schnell erledigt sein sollte.

Wie bei natürlichen Sprachen wird die Syntax einer Programmiersprache durch eine Grammatik bestimmt. Die Geschwindigkeit der Syntaxanalyse hängt im wesentlichen von der Komplexität der verwendeten Grammatik ab. Für Programmiersprachen bieten sich daher einfach aufgebaute Grammatiken an, welche eine effiziente Syntaxüberprüfung ermöglichen. Der Preis, den man für die Einschränkung der Grammatiken zahlt, ist der Verlust semantischer Ausdruckstärke, und damit hoch. Das hier verfolgte Anliegen ist es, einen Ansatz vorzustellen, der auch für etwas komplexere Grammatiken einen effizienten Algorithmus zur Syntaxanalyse bereitstellt.

Um diesen Algorithmus formal darzustellen, werden wir im Abschnitt 2 ein spezielles Automatenmodell einführen. Automatenmodelle werden in der Informatik häufig zur Klassifikation von Sprachen verwendet. Dabei sagt man, daß ein Automat eine bestimmte Sprache *entscheidet*, falls er für jedes beliebige Eingabewort berechnen kann, ob dieses Wort zur Sprache gehört oder nicht. Jedes Automatenmodell definiert auf diese Weise eine eigene Sprachklasse, nämlich die Menge aller Sprachen, die durch einen Automaten dieses Modells entschieden werden können.

Auf diese Weise werden die deterministischen kontextfreien Sprachen von den deterministischen Einkellerautomaten definiert. Diese Automaten haben die Eigenschaft, daß sie für die Verarbeitung einer Eingabe höchstens um einen konstanten Faktor mehr Rechenschritte benötigen, als das Eingabewort Symbole hat. Damit besitzen sie exakt das Verhalten, welches man sich bezüglich der Geschwindigkeit der Syntaxüberprüfung eines Programmes wünscht: Die benötigte Zeit soll sich maximal linear zur Programmlänge verhalten. Allerdings sind einige Konstrukte, die man gerne in Programmiersprachen verwenden würde, nicht kontextfrei, sondern kontextsensitiv. Kontextsensitive Sprachen lassen sich jedoch meist nicht durch die oben genannten Einkellerautomaten entscheiden.

Gerhard Buntrock hat in seiner Habilitationsschrift [Bun96] die *deterministischen wachsenden kontextsensitiven Sprachen (DGCSL)* untersucht. Diese sind eine echte Obermenge der deterministischen kontextfreien Sprachen und werden durch deterministische schrumpfende Zweikellerautomaten, welche wie die Einkellerautomaten die Ausgabe in Linearzeit berechnen, definiert. Buntrock zeigt in dieser Arbeit unter anderem, daß mit den *Church-Rosser-Sprachen (CRL)* eine formalsprachliche Entsprechung zu DGCSL existiert. Dies ist für unsere Betrachtungsweise insofern interessant, als daß jede Sprache aus CRL durch

ein bestimmtes reduzierendes Wortersetzungssystem, *Church-Rosser-System* genannt, definiert wird, aus dem leicht sowohl ein Zweikellerautomat, der die Sprache akzeptiert, als auch eine Grammatik, welche sie erzeugt, konstruiert werden kann. Da also mit dem Wortersetzungssystem Richtlinien für den Programmiersprachenentwickler, mit dem Automaten ein effizientes Modell zur Realisierung und mit der Grammatik eine Sprachbeschreibung für den Programmierer zur Verfügung stehen, liegt die praktische Verwendung dieses Ansatzes nahe.

Daher wurde im Rahmen dieser Arbeit das Programm `TPDAsim` entwickelt, welches primär die Aufgabe hat, die Arbeitsweise eines Zweikellerautomaten bei gegebenem Reduktionssystem oder gegebener Grammatik zu simulieren und interaktiv zu visualisieren. Es können sowohl Grammatiken als auch reduzierende Wortersetzungssysteme zur Sprachdefinition verwendet werden, wobei deren Eingabe einfach und flexibel ist. Nach einem erfolgreichen Parserdurchlauf erfolgt optional eine Analyse, welche die Alphabetsymbole und Regeln des eingegebenen Systems nach Funktion zu klassifizieren versucht. Weiterhin bietet `TPDAsim` bei der Verifizierung, ob es sich bei dem betrachteten Wortersetzungssystem tatsächlich um ein Church-Rosser-System handelt, diverse unterstützende Funktionalität an. Falls das Ergebnis der Überprüfung positiv ausfällt, ist für den Anwender gesichert, daß die Automatensimulation korrekt abläuft. Allerdings kann in jedem Fall aus dem gegebenen Wortersetzungssystem nach dem im Beweis zu Satz 12 im Abschnitt 2 angegebenen Algorithmus automatisch ein Zweikellerautomat konstruiert werden, dessen Arbeitsweise nach Spezifikation seiner Eingabe simuliert werden kann.

Die in Abbildung 1 dargestellte Simulation selbst kann in beliebig großen Schritten wahlweise mit Verzögerung durchgeführt werden, wobei ihre graphische Realisierung an die anschaulichen Vorstellung eines Zweikellerautomaten angelehnt ist.

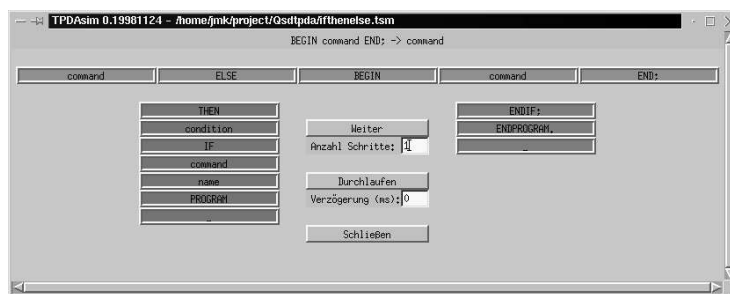


Abbildung 1: Die Simulation des Zweikellerautomaten

Des weiteren könnte `TPDAsim` für einen Programmiersprachenentwickler bereits ein Tool darstellen, mit welchem er für seine Programmiersprache erwünschte kontextsensitive Konstrukte auf Verwendbarkeit überprüfen kann. Dabei ist die automatische und effiziente Überprüfung einer notwendigen, nichttrivialen Eigenschaft (Konfluenz) der Systeme sicherlich auch für sich genommen eine interessante Programmfunktion.

## 2 Theoretischer Hintergrund

Das Programm TPDAsim, welches das wesentliche Produkt dieser Arbeit darstellt, behandelt die Simulation schrumpfender Zweikellerautomaten. Wir wollen in diesem Abschnitt aus theoretischer Sicht darlegen, warum die Betrachtung solcher Automaten in der Praxis lohnend sein kann.

### 2.1 Präliminarien

Bevor wir zum zentralen Punkt im Abschnitt 2.2 kommen, nämlich der konstruktiven Charakterisierung der Church-Rosser-Sprachen mit den durch schrumpfende Zweikellerautomaten definierten deterministisch wachsenden kontextsensitiven Sprachen, werden wir in diesem Abschnitt sämtliche im obigen Nebensatz eingeworfenen Bezeichnungen ausführlich erläutern.

#### 2.1.1 Zweikellerautomaten

Wie sehen Zweikellerautomaten aus, wie arbeiten sie, was besitzen sie für wichtige Eigenschaften, und wie kann man die Effizienz ihrer Arbeitsweise nachweisen?

##### Definition 1 [Bun96]

Ein *Zweikellerautomat* (TPDA) ist ein nichtdeterministischer Automat mit zwei Kellern und wird gegeben durch ein Siebentupel:  $M = (Q, \Sigma, \Gamma, \delta, q_0, \perp, F)$

- $Q$  ist eine endliche Menge, die *Zustandsmenge*.
- $\Sigma$  ist ein Alphabet, das *Eingabealphabet*.
- $\Gamma \supseteq \Sigma$  ist ein Alphabet, das *Arbeitsalphabet*.
- $q_0 \in Q$  ist ein ausgezeichneteter Zustand, der *Anfangszustand*.
- $\perp \in (\Gamma \setminus \Sigma)$  ist das *initiale Kellersymbol* oder *Kellerbodensymbol*
- $F \subseteq Q$  ist die Menge der *Endzustände*.
- $\delta$  ist eine totale Abbildung von  $(Q \times \Gamma \times \Gamma)$  in die endlichen Teilmengen von  $(Q \times \Gamma^* \times \Gamma^*)$ , das *Programm* von  $M$ .

Ist der Automat deterministisch, bildet also  $\delta$  ausschließlich auf einelementige Mengen ab, so verwendet man die Abkürzung DTPDA.

Aus technischen Gründen setzen wir voraus, daß  $\Gamma$  und  $Q$  disjunkt sind.

Eine Konfiguration des TPDA besteht aus sämtlichen in den Kellern befindlichen Symbolen sowie dem aktuellen Zustand, und kann folglich als ein Wort über  $\Gamma \cup Q$  notiert werden. Für eine Konfiguration  $\alpha_2 \alpha_1 q \beta_1 \beta_2 \in \Gamma^* Q \Gamma^*$ ,  $q \in Q$ , heißen  $\alpha_1$  und  $\beta_1$  jeweils *Topstück* des linken bzw. rechten Kellers (hierbei sei bemerkt, daß wir den linken Keller immer "von unten nach oben" notieren, so daß sich links und rechts neben dem Zustand in der Notation die Topsyndrome der Keller befinden).

Sei  $w \in \Gamma^*$  eine Eingabe. Dann steht zu Beginn im linken Keller nur das Kellerbodensymbol  $\perp$  und im rechten Keller  $w\perp$ . Der Automat befindet sich also in der

Konfiguration  $\perp q_0 w \perp$ .

Unter den Voraussetzungen

$$q, q_1, \dots, q_m \in Q; A, B \in \Gamma; \gamma_1, \gamma_2 \in \Gamma^*; \alpha_1, \beta_1, \alpha_2, \beta_2, \dots, \alpha_m, \beta_m \in \Gamma^*$$

und

$$\delta(q, A, B) = \{(q_1, \alpha_1, \beta_1), (q_2, \alpha_2, \beta_2), \dots, (q_m, \alpha_m, \beta_m)\}$$

kann der Automat in einem Rechenschritt mit beliebigem  $i \in \{1, 2, \dots, m\}$  in den Zustand  $q_i$  übergehen und die Topsybole  $A, B$  der beiden Keller durch  $\alpha_i$  bzw.  $\beta_i$  ersetzen, was wir wie folgt notieren:

$$\gamma_1 A q B \gamma_2 \xrightarrow{M} \gamma_1 \alpha_i q_i \beta_i \gamma_2$$

Mit  $\xrightarrow{M}^*$  meinen wir den reflexiven und transitiven Abschluß von  $\xrightarrow{M}$ . Bei uns soll die Menge

$$N(M) =_{\text{def}} \{w \in \Sigma^* \mid \exists q_{\text{fin}} \in F : \perp q_0 w \perp \xrightarrow{M}^* q_{\text{fin}}\}$$

die von  $M$  akzeptierte Sprache sein. Da also insbesondere der rechte Keller leer ist, hat der TPDA, wenn er akzeptiert, die gesamte Eingabe gelesen.

Offenbar hat der Automat die Aufgabe, die gesamte Eingabe durch sukzessive Wortersetzung auf ein leeres Wort zu reduzieren. Dies legt die Programmierung mit einem Wortersetzungssystem nahe.

### 2.1.2 Wortersetzungssysteme

**Definition 2** Eine Teilmenge  $R \subseteq \Sigma^* \times \Sigma^*$  ist ein *Wortersetzungssystem* über  $\Sigma$ , oft *Semi-Thue-System* oder *Regelsystem* genannt. Wir nehmen  $R$  stets als endlich an. Die Elemente von  $R$  heißen *Regeln*, und wir verwenden eine Regel  $(v, w) \in R$ , in dem wir in einem Wort  $xvy$  das  $v$  durch  $w$  ersetzen und notieren das durch:

$$xvy \xrightarrow{R} xwy$$

Mit  $\xrightarrow{R}^*$  notieren wir wieder den reflexiven und transitiven Abschluß von  $\xrightarrow{R}$ .

Außerdem nennen wir  $L(R) =_{\text{def}} \{w \in \Sigma^* \mid w \xrightarrow{R}^* \varepsilon\}$  die *von  $R$  akzeptierte Sprache*.

Nicht jedes Wortersetzungssystem ist allerdings in der Praxis zur Programmierung geeignet. So müssen wir voraussetzen, daß das Wortersetzungssystem keine unendliche Ableitungskette besitzt, die ein Terminieren des Automaten verhindern könnte.

### Definition 3 [BO93]

Ein Wortersetzungssystem  $R$  heißt *noethersch*, wenn es keine unbeschränkte Ableitungskette  $w_1 \xrightarrow{R} w_2 \xrightarrow{R} w_3 \xrightarrow{R} \dots$  gibt.

Wenn man nicht nur das Terminieren sichern möchte, sondern auch eine gewisse Effizienz der Reduktion, muß man die Menge der zugelassenen Wortersetzungssysteme noch weiter einschränken. Wir werden dafür verlangen, daß jede Regel auch wirklich reduzierend wirkt.

Offenbar ist diese Bedingung erfüllt, wenn man verlangt, daß jede Regel bezüglich der Wortlänge verringernd wirkt. Wortersetzungssysteme, die diese Voraussetzung erfüllen, werden *verkürzend* genannt.

### 2.1.3 Bewertungen und ihre Folgen

Wie die Praxis zeigt, erschwert das Bestehen auf verkürzende Regeln die Konstruktion erheblich; viele Wortersetzungssysteme werden dadurch groß und unhandlich.

Wir werden daher einen anderen Ansatz wählen: Jedem Symbol wird ein positives Gewicht zugeordnet, es wird also bewertet. Der Wert eines Wortes soll sich dann aus der Summe der Einzelwerte der Wortsymbole ergeben.

#### Definition 4 [Bun96]

Sei  $\Sigma$  ein beliebiges Alphabet. Die Fortsetzung einer Funktion  $f : \Sigma \rightarrow \mathbb{N}$  zu einem Homomorphismus  $f : \Sigma^* \rightarrow \mathbb{N}$  in die Struktur  $(\mathbb{N}, +)$  heißt *Bewertung*. Für ein Wort  $w \in \Sigma^*$  heißt dann  $f(w)$  der *Wert* von  $w$ .

Wenn man festlegt, daß jedes Wort nach der Anwendung einer Regel einen niedrigeren Wert haben muß als zuvor, erhält man auf diese Weise ebenfalls eine Art Verkürzungseigenschaft.

#### Definition 5 [Bun96]

Ein Wortersetzungssystem  $R$  heißt *schrumpfend*, falls es eine Bewertung  $f : \Sigma^* \rightarrow \mathbb{N}$  gibt, so daß für jedes  $(l, r) \in R$  gilt:  $f(l) > f(r)$

Offenbar benötigt ein schrumpfendes Wortersetzungssystem höchstens um einen konstanten Faktor mehr Regelanwendungen für die Reduktion des Eingabewortes als ein verkürzendes.

Des weiteren haben Gundula Niemann und Friedrich Otto in [NO98] nachgewiesen, daß man mit schrumpfenden Systemen nicht mehr Sprachen entscheiden kann als mit verkürzenden.

Durch die Verwendung schrumpfender Wortersetzungssysteme haben wir ein gewisses Maß an Flexibilität für die Gestaltung des Systems gewonnen (es sei angemerkt, daß wir an anderer Stelle einen Preis dafür zahlen (3.2.3)).

Nachdem wir schrumpfende Wortersetzungssysteme behandelt haben, wollen wir den Begriff „schrumpfend“ auf Zweikellerautomaten übertragen: Indem wir eine „normale“ Bewertungsfunktion, die auf einem Alphabet arbeitet, zusätzlich die Zustände des Automaten bewerten lassen, sind wir in der Lage, mit ihr Konfigurationen zu bewerten.

**Definition 6** [Bun96]

Ein TPDA  $M = (Q, \Sigma, \Gamma, \delta, q_0, \perp, F)$  heißt *schrumpfend* (*sTPDA*), wenn es eine Bewertungsfunktion  $f$  für  $\Gamma \cup Q$  gibt, so daß für alle  $q, q' \in Q$ ,  $A, B \in \Gamma$  und alle  $\alpha, \beta \in \Sigma^*$  gilt:

Falls  $\delta(q, A, B) \ni (q', \alpha, \beta)$  ist, dann gilt  $f(qAB) > f(q'\alpha\beta)$ . Wir sagen dann kurz „ $f$  ist eine Bewertung für  $M$ “.

Ist der Automat darüberhinaus deterministisch, verwenden wir die Abkürzung *sDTPDA*.

Die Sprachen, die durch schrumpfende Zweikellerautomaten akzeptiert werden können, bilden die Sprachklasse DGCSL:

**Definition 7** [Bun96] Eine Sprache heißt *deterministisch wachsend kontextsensitiv* genau dann, wenn sie von einem deterministischen schrumpfenden Zweikellerautomaten akzeptiert wird. Die so definierte Sprachklasse bezeichnen wir mit *DGCSL*.

Da jeder Arbeitsschritt eines sDTPDA schrumpfend wirkt, verringert sich jedesmal auch der Wert der Gesamtkonfiguration, da die restlichen Kellerinhalte unberührt bleiben. Also kann der Automat nicht mehr Schritte benötigen, als der Wert seiner Anfangskonfiguration, der sich offenbar linear zum Wert der Eingabe verhält, angibt.

Diese Folgerung wollen wir in einer Bemerkung festhalten, da sie der Grund ist, warum schrumpfende Zweikellerautomaten für die Praxis interessant sind.

**Bemerkung 8** Ein schrumpfender Zweikellerautomat entscheidet ein Eingabewort in Linearzeit.

Bevor wir uns endgültig dem Automaten zuwenden, wollen wir noch eine letzte Eigenschaft für die zur Programmierung zu verwendenden Wortersetzungssysteme voraussetzen, nämlich die Konfluenz.

**2.1.4 Church-Rosser-Systeme****Definition 9** [Bun96]

Ein Wortersetzungssystem  $R$  über  $\Sigma$  heißt *konfluent*, wenn für alle  $u, v, w \in \Sigma^*$  stets der Fall ist:

$$u \xrightarrow{*}_R v, u \xrightarrow{*}_R w \implies \exists z \in \Sigma^* : v \xrightarrow{*}_R z, w \xrightarrow{*}_R z$$

Gibt es mehrere Regeln, die zu einem Zeitpunkt auf ein Wort angewendet werden könnten, so ist es im Falle eines konfluenten Wortersetzungssystems unbedeutend, welche von ihnen gewählt wird, oder an welcher von den potentiell vielen möglichen Positionen im Wort sie ansetzen soll, da man unabhängig

von der Wahl der Regel am Ende der gesamten Ableitungskette aufgrund der Konfluenzbedingung in denselben irreduziblen Wort mündet. Somit genügt es beispielsweise, ausschließlich Linksableitungen zu berechnen.

Umrahmt man Eingabewort durch nicht zum Eingabealphabet gehörige Randmarkierungen, erweitert sich die Sprachklasse der von schrumpfenden konfluenten Wortersetzungssystemen akzeptierten Sprachen zu jener der *generalisierten Church-Rosser-Sprachen*:

**Definition 10** [Bun96]

Ein *Church-Rosser-System* (CRS) ist ein 5-Tupel  $(\Sigma, \Gamma, R, t_1, t_2)$ , wobei  $R$  ein konfluentes und verkürzendes Wortersetzungssystem über dem Alphabet  $\Gamma$  ist. Des weiteren seien  $t_1, t_2$  aus  $(\Gamma \setminus \Sigma)^*$  und irreduzibel. Weiterhin sei  $Y \in (\Gamma \setminus \Sigma)$  ebenso irreduzibel, dann heißt die Sprache

$$L =_{\text{def}} \left\{ w \in \Sigma^* : t_1 w t_2 \xrightarrow{*}_R Y \right\}$$

*Church-Rosser-Sprache* (CRL). Wenn  $R$  nur schrumpfend ist, heißt das zugehörige CRS *generalisiertes Church-Rosser-System* (GCRS) und  $L$  *generalisierte Church-Rosser-Sprache* (GCRL).

Da Gundula Niemann und Friedrich Otto in [NO98] zeigen konnten, daß schrumpfende Wortersetzungssysteme nicht mächtiger als verkürzende sind und somit CRL = GCRL gilt, werden wir in Zukunft ausschließlich von CRL sprechen und diese Sprachklasse nicht mehr durch CRS, sondern durch GCRS definieren.

Nun sind wir für den letzten Theorieabschnitt vorbereitet, dessen (für uns) zentrales Resultat die Programmierung eines Zweikellerautomaten mit einem GCRS aus dem Beweis zu Satz 14 sein wird.

## 2.2 Charakterisierung der CRL durch sDTPDA

Zuerst werden wir zeigen, daß sich aus jedem sDTPDA ein konfluentes schrumpfendes Wortersetzungssystem konstruieren läßt:

**Satz 11** [Bun96]

Jede Sprache, die von einem sDTPDA akzeptiert wird, liegt in CRL, d.h., für jeden sDTPDA  $M = (Q, \Sigma, \Gamma, \delta, q_0, \perp, \{q_{\text{fin}}\})$  existiert ein generalisiertes Church-Rosser-System  $S = (\Sigma, \Delta, R, t_1, t_2)$  mit

$$N(M) = L(S)$$

*Beweis:*

Das WES  $R$  soll konfluent über dem Alphabet  $\Delta =_{\text{def}} Q \cup \Gamma \cup \bar{\Gamma}$  sein. Dabei ist  $\bar{\Gamma}$  eine Kopie von  $\Gamma$ , d.h.,  $\bar{\Gamma}$  ist der Wertebereich eines Homomorphismus, der jedes Symbol  $a \in \Gamma$  auf ein Symbol  $\bar{a} \in \bar{\Gamma}$  abbildet. Des weiteren gelte  $\Gamma \cap \bar{\Gamma} = \emptyset$ .

Als nächstes setzen wir  $t_1 =_{\text{def}} \bar{\perp} q_0$  und  $t_2 =_{\text{def}} \perp$ . Dann werden die Regeln von  $R$  mit der Übergangsfunktion  $\delta$  für alle  $q \in Q$  und  $a, b \in \Gamma$  definiert, wobei wir die Markierung der Symbole in  $\bar{\Gamma}$  auf die Wörter in  $\bar{\Gamma}^*$  fortsetzen:

$$\delta(q, a, b) = \{(q', u, v)\} \implies (\bar{a}qb, \bar{u}q'v) \in R$$

Das akzeptierende Symbol  $Y$  des GCRS soll  $q_{\text{fin}}$  sein. Folglich ist die von  $S$  erkannte Sprache wie folgt definiert:

$$L(S) = \left\{ w \in \Sigma^* \mid \bar{\perp} q_0 w \perp \xrightarrow{*}_R q_{\text{fin}} \right\}$$

Diese Sprache ist nach Konstruktion offensichtlich identisch zu

$$N(M) = \{w \in \Sigma^* \mid \perp q_0 w \perp \xrightarrow{*}_M q_{\text{fin}}\}$$

Nach Voraussetzung existiert außerdem eine Bewertung  $f$  für  $M$ , bezüglich welcher der Automat in jedem der durch  $\delta$  beschriebenen Arbeitsschritte schrumpft. Setzen wir die Bewertung auf  $\bar{\Gamma}$  gemäß  $f(\bar{a}) =_{\text{def}} f(a)$ , so ergibt sich sofort, daß das aus  $\delta$  konstruierte WES  $R$  ebenfalls schrumpfend ist.

Wir zeigen jetzt noch die Konfluenz von  $R$ . Da  $M$  deterministisch ist, ist es auch  $R$ . Somit bleibt nur zu überprüfen, ob sich zwei linke Regelseiten aus  $R$  "überschneiden", d.h., ob das Ende der einen der Anfang der anderen ist (vgl. Alg. 16). Da aber jede linke Regelseite nach Konstruktion die Form  $\bar{a}qb$  mit  $\bar{a} \in \bar{\Gamma}$  und  $b \in \Gamma$  besitzt, ist dies offenbar unmöglich.  $\diamond$

Jetzt zeigen wir, daß auch die Umkehrung dieses Satzes gilt. Dazu werden wir in Satz 12 zunächst zeigen, daß ein sDTPDA ein konfluentes schrumpfendes Wortersetzungssystem simulieren kann, um dann in Satz 14 die Aussage auf generalisierte Church-Rosser-Systeme auszuweiten.

### Satz 12 [Bun96]

Für jedes konfluente schrumpfende Wortersetzungssystem  $R \subseteq \Sigma^* \times \Sigma^*$  läßt sich ein sDTPDA  $M = (Q, \Sigma, \Gamma, \delta, q_0, \perp, \{q_{\text{fin}}\})$  konstruieren, so daß gilt:

$$N(M) = L(R)$$

*Beweis:*

Sei  $\mu = \max\{|l| : (l, r) \in R\}$ , also die maximale Länge einer linken Regelseite.

Besitzt  $R$  mehrere Regeln mit derselben linken Seite, so können wir diesen Nicht-determinismus einfach durch Streichen aller dieser Regeln bis auf eine auflösen. Die von dem WES erkannte Sprache ändert sich dadurch nicht, da alle diese Regeln wegen der Konfluenzeigenschaft sowieso letztendlich<sup>1</sup> auf dasselbe irreduzible Wort führen würden.

<sup>1</sup>d.h. nach Anwendung des Abschlußoperators auf die jeweiligen rechten Regelseiten

Eine Grundidee des Automaten ist, ein Wort der Länge  $\leq \mu$  im Zustand zu kodieren und ausschließlich darauf die Regeln aus  $R$  anzuwenden.

Zusätzlich zu diesen “gedachten” Symbolen werden im Zustand noch drei “echte” Zustände  $p_0, p_1$  und  $p_{\text{fin}}$  kodiert, die wir auch meinen werden, wenn wir in Zukunft sagen, “Der Automat geht über in den Zustand ...”.

Wir werden zunächst zwei Alphabetskopien  $\hat{\Sigma}$  und  $\bar{\Sigma}$  für das Alphabet  $\Sigma$  einführen, um in den verschiedenen Komponenten (linker und rechter Keller, Zustand) des Automaten disjunkte Alphabete benutzen zu können. Dies ist wichtig, um ein und dasselbe Symbol in verschiedenen Komponenten unterschiedlich bewerten zu können. So kann auch das Kopieren von einer Komponente in eine andere ein schrumpfender Vorgang sein.

Wir benutzen dazu die beiden Isomorphismen  $\bar{\varphi}$  und  $\hat{\varphi}$ :

$$\hat{\varphi} : \Sigma^* \longrightarrow \hat{\Sigma}^* \quad \bar{\varphi} : \Sigma^* \longrightarrow \bar{\Sigma}^*$$

Für ein Wort  $w \in \Sigma^*$  soll dann gelten:

$$\hat{\varphi}(w) =_{\text{def}} \hat{w} \quad \bar{\varphi}(w) =_{\text{def}} \bar{w}$$

Für  $M$  soll gelten:

$$\begin{aligned} \Gamma &=_{\text{def}} \Sigma \cup \bar{\Sigma} \cup \{\perp\} \\ Q &=_{\text{def}} \left( \bigcup_{j=0}^{\mu} \hat{\Sigma}^j \right) \times \{p_0, p_1, p_{\text{fin}}\} \\ q_0 &=_{\text{def}} [\varepsilon, p_0] \\ q_{\text{fin}} &=_{\text{def}} [\varepsilon, p_{\text{fin}}] \end{aligned}$$

Wir werden  $\bar{\Sigma}$  ausschließlich im linken Keller verwenden, während im rechten Keller natürlich das Originalalphabet  $\Sigma$  benutzt wird, denn  $w \in \Sigma^*$  wird dort eingegeben.

Es fehlt jetzt noch das Wichtigste, nämlich das Programm  $\delta$  des Automaten. Da der Automat deterministisch ist, lassen wir die Mengenklammern der Bilder unter  $\delta$  weg.

Sei zunächst  $\hat{u} \in \hat{\Sigma}^{\leq \mu}$ ,  $\bar{b} \in \bar{\Sigma} \cup \{\perp\}$ , und  $a \in \Sigma$ . Dann sind die ersten drei Fälle wie folgt festgelegt:

$$\delta([\hat{u}, p_0], \bar{b}, a) =_{\text{def}} \begin{cases} ([\hat{u}_l, p_1], \bar{b}, ra) & \text{falls } \hat{u} = \hat{u}_l \hat{r} \text{ mit } (l, r) \in R & (1) \\ ([\hat{u}\hat{a}, p_0], \bar{b}, \varepsilon) & \text{falls } |\hat{u}| < \mu & (2) \\ ([\hat{v}\hat{a}, p_0], \bar{b}\bar{c}, \varepsilon) & \text{falls } |\hat{u}| = \mu, \hat{u} = \hat{c}\hat{v}, \hat{c} \in \hat{\Sigma} & (3) \end{cases}$$

Gilt  $a = \perp$ , ist der rechte Keller also leer bis auf das Kellersymbol, ergeben sich die folgenden Fälle 4 bis 6, von denen Fall 4 zugebenermaßen nur einen Spezialfall von Fall 1 darstellt<sup>2</sup>.

$$\delta([\hat{u}, p_0], \bar{b}, \perp) =_{\text{def}} \begin{cases} ([\hat{u}_l, p_1], \bar{b}, r\perp) & \text{falls } \hat{u} = \hat{u}_l \hat{r} \text{ mit } (l, r) \in R & (4) \\ ([\varepsilon, p_{\text{fin}}], \varepsilon, \varepsilon) & \text{falls } \hat{b} = \perp \text{ und } \hat{u} = \varepsilon & (5) \\ ([\varepsilon, p_{\text{fin}}], \bar{b}\bar{u}, \varepsilon) & \text{sonst} & (6) \end{cases}$$

<sup>2</sup>Die Übersichtlichkeit soll hier Vorrang gegenüber einer kompakten Darstellung haben

Für  $\hat{u}, \bar{b}$  wie oben und  $a \in \Sigma \cup \{\perp\}$  legen wir die restlichen drei Fälle fest, die allesamt vom Zustand  $p_1$  ausgehen:

$$\delta([\hat{u}, p_1], \bar{b}, a) =_{\text{def}} \begin{cases} ([\hat{b}\hat{u}, p_1], \varepsilon, a) & \text{falls } \bar{b} \in \bar{\Sigma} \text{ und } |\hat{u}| < \mu - 1 & (7) \\ ([\hat{b}\hat{u}, p_0], \varepsilon, a) & \text{falls } \bar{b} \in \bar{\Sigma} \text{ und } |\hat{u}| = \mu - 1 & (8) \\ ([\hat{u}, p_0], \perp, a) & \text{falls } \bar{b} = \perp & (9) \end{cases}$$

Um die Bedeutung der einzelnen Fälle besser zu verstehen, wollen wir den Automaten jetzt noch einmal in Worten beschreiben, und uns dabei jeweils in runden Klammern auf die Fälle 1 bis 9 von oben beziehen.

- Der Automat speichert in seinen Zuständen das vom rechten Keller Gelesene als “gedachtes” Wort bis zur Länge  $\mu$  (2), und prüft jeweils beim Auffüllen, ob er eine Regel aus  $R$  darauf anwenden kann.
- Wenn dies geht (1, 4), schreibt er die entsprechende rechte Regelseite auf den rechten Keller, “löscht” die linke Regelseite im Zustand und geht über in  $p_1$ . Dort “füllt” er (7, 8, 9) das im Zustand kodierte Wort mit Symbolen aus dem linken Keller bis zur Länge  $\mu$  (oder bis sich im linken Keller keine Symbole mehr befinden) auf.
- Besitzt das im Zustand gespeicherte Wort die Länge  $\mu$ , und läßt sich immernoch keine Regel darauf anwenden (3), wirkt der Zustand quasi als Puffer: Nach dem first-in-first-out-Prinzip wird das nächste Symbol aus dem rechten Keller gelesen und rechts an das Zustandswort angefügt und das erste, also “älteste” Symbol des Zustandswortes aus diesem entfernt und auf den linken Keller geschrieben.
- Ist der rechte Keller irgendwann leer (4, 5, 6), und läßt sich auf das Zustandswort keine Regel anwenden (5, 6), “wirft” der Automat dieses noch auf den linken Keller, geht dann in den Endzustand über und hält. Falls sich im linken Keller nur das Kellerbodensymbol befindet (5), wird dieses in diesem Schritt auch gelöscht, damit die Konfiguration des Automaten nun der Bedingung zum Akzeptieren aus Definition 1 genügt. Somit gilt  $w \in N(M)$ .  
Befinden sich allerdings außer dem Kellerbodensymbol noch weitere Symbole im linken Keller (6), verwirft der Automat. Es gilt  $w \notin N(M)$ .

Offensichtlich gilt, daß dieser Automat deterministisch arbeitet, falls  $R$  keine zwei Regeln mit derselben linken Seite aufweist, was wir zu Beginn des Beweises ausgeschlossen haben.

Aus der Eindeutigkeit der Linksableitung folgt (i):

$$\begin{aligned} \forall w, u, z \in \Sigma^* : \\ \perp q_0 w \perp \vdash_M^* \perp \bar{u} [\hat{v}, p_i] z &\implies w \xrightarrow{R^*} uvz \\ \perp q_0 w \perp \vdash_M^* \perp \bar{u} [\hat{v}, p_i] z &\implies w \xrightarrow{R^*} uvz \end{aligned}$$

Nach Konstruktion gilt dann (ii):

$$\begin{aligned} \forall w \in L(R) : \perp q_0 w \perp \vdash_M^* q_{\text{fin}} \\ \forall w \in \Sigma^* \setminus L(R) : \perp q_0 w \perp \vdash_M^* \perp \overline{w_{\text{irr}}} q_{\text{fin}} \end{aligned}$$

Hierbei bezeichnet  $w_{\text{irr}}$  dasjenige<sup>3</sup> irreduzible Wort, für das  $w \xrightarrow{*}_R w_{\text{irr}}$  gilt.

Da wir für ein Akzeptieren des Automaten verlangt haben, daß beide Keller leer sein sollen, folgt aus (ii) sofort die zu zeigende Eigenschaft

$$\forall w \in \Sigma^* : w \in L(R) \iff w \in N(M)$$

Jetzt bleibt noch zu zeigen, daß  $M$  auch wirklich schrumpfend ist. Da der Beweis allerdings bereits lang genug ist, verpacken wir den Rest in ein eigenes Lemma.

**Lemma 13** [Bun96] Der in Satz 12 definierte Automat ist schrumpfend.

*Beweis* (entspricht weitgehend dem aus [Bun96]):

Nach Voraussetzung existiert eine Bewertung  $g$ , mit der  $R$  schrumpft. Darauf aufbauend werden wir nun eine Bewertungsfunktion  $f : \Gamma \cup Q$  für  $M$  definieren. Dazu legen wir zuerst einmal zwei Konstanten fest:

$$\begin{aligned} g_{\max} &=_{\text{def}} \max \{g(a) : a \in \Sigma\} \\ \nu &=_{\text{def}} g_{\max} \cdot \max \{|r| : (l, r) \in R\} + 1 \end{aligned}$$

Mit diesen definieren wir alle Werte von  $f$ :

$$\begin{aligned} f(p_0) &=_{\text{def}} 2 \\ f(p_1) &=_{\text{def}} 3 \\ f(p_{\text{fin}}) &=_{\text{def}} 1 \\ f(\perp) &=_{\text{def}} \mu \cdot g_{\max} \end{aligned}$$

$\forall a \in \Sigma$  und  $\forall [\hat{u}, p] \in Q$ :

$$\begin{aligned} f(a) &=_{\text{def}} g(a) \cdot 2\nu + 2g_{\max} \\ f(\hat{a}) &=_{\text{def}} g(a) \cdot 2\nu \\ f(\bar{a}) &=_{\text{def}} g(a) \cdot 2\nu + g_{\max} \\ f([\hat{u}, p]) &=_{\text{def}} f(\hat{u}) + f(p) \end{aligned}$$

Jetzt müssen wir zeigen, daß unter dieser Bewertung  $f$  jede Regel aus  $\delta$  schrumpfend wird. Dabei beziehen wir uns auf die neun Fälle der Definition von  $\delta$  aus Satz 12.

$$\begin{aligned} (1) \quad & \delta([\hat{u}, p_0], \bar{b}, a) = ([\hat{u}_l, p_1], \bar{b}, ra) \text{ mit } \hat{u} = \hat{u}_l \hat{l} \text{ für ein } (l, r) \in R : \\ & f([\hat{u}_l \hat{l}, p_0]) + f(\bar{b}) + f(a) - (f([\hat{u}_l, p_1]) + f(\bar{b}) + f(ra)) \\ &= f(\hat{u}_l) + f(\hat{l}) + f(p_0) + f(\bar{b}) + f(a) - f(\hat{u}_l) - f(p_1) - f(\bar{b}) - f(r) - f(a) \\ &= f(\hat{l}) + f(p_0) - f(p_1) - f(r) \\ &= g(l) \cdot 2\nu - g(r) \cdot 2\nu - 2g_{\max} \cdot |r| - 1 \\ &\geq (g(r) + 1) \cdot 2\nu - g(r) \cdot 2\nu - 2g_{\max} \cdot |r| - 1 \\ &= 2\nu - 2g_{\max} \cdot |r| - 1 \\ &> 0, \text{ da } \nu > g_{\max} \cdot |r| \text{ definiert wurde.} \end{aligned}$$

<sup>3</sup> dasjenige ist korrekt, da jedes Wort durch die Konfluenzbedingung nur zu einem einzigen irreduziblen Wort reduziert werden kann

- (2)  $\delta([\hat{u}, p_0], \bar{b}, a) = ([\hat{u}\hat{a}, p_0], \bar{b}, \varepsilon) :$   
 $f(\hat{u}) + f(p_0) + f(\bar{b}) + f(a) - f(\hat{u}) - f(\hat{a}) - f(p_0) - f(\bar{b})$   
 $> 0$ , da  $f(a) > f(\hat{a})$ .
- (3)  $\delta([\hat{u}, p_0], \bar{b}, a) = ([\hat{v}\hat{a}, p_0], \bar{b}\bar{c}, \varepsilon)$  mit  $\hat{u} = \hat{c}\hat{v} :$   
 $f(\hat{c}) + f(\hat{v}) + f(p_0) + f(\bar{b}) + f(a) - f(\hat{v}) - f(\hat{a}) - f(p_0) - f(\bar{b}) - f(\bar{c})$   
 $> 0$ , da  $f(\bar{c}) - f(\hat{c}) < f(a) - f(\hat{a})$ .
- (4)  $\delta([\hat{u}, p_0], \bar{b}, \perp) = ([\hat{u}_l, p_1], \bar{b}, r\perp)$  mit  $\hat{u} = \hat{u}_l\hat{l}$  für ein  $(l, r) \in R :$   
 $f(\hat{u}_l) + f(\hat{l}) + f(p_0) + f(\bar{b}) + f(\perp) - f(\hat{u}_l) - f(p_1) - f(\bar{b}) - f(r) - f(\perp)$   
 $> 0$  ( $f(\perp)$  fällt heraus, Rest wie Fall (1)).
- (5)  $\delta([\varepsilon, p_0], \perp, \perp) = ([\varepsilon, p_{fin}], \varepsilon, \varepsilon) :$   
 $f(\varepsilon) + f(p_0) + f(\perp) + f(\perp) - f(\varepsilon) - f(p_{fin}) - f(\varepsilon) - f(\varepsilon)$   
 $> 0$ , da  $f(p_0) > f(p_{fin})$ .
- (6)  $\delta([\hat{u}, p_0], \bar{b}, \perp) = ([\varepsilon, p_{fin}], \bar{b}\bar{u}, \varepsilon) :$   
 $f(\hat{u}) + f(p_0) + f(\bar{b}) + f(\perp) - f(\varepsilon) - f(p_{fin}) - f(\bar{b}) - f(\bar{u})$   
 $> 0$ , da  $f(\bar{u}) - f(\hat{u}) = |u| \cdot g_{\max} \leq \mu \cdot g_{\max} = f(\perp)$  und  $f(p_0) > f(p_{fin})$ .
- (7)  $\delta([\hat{u}, p_1], \bar{b}, a) = ([\hat{b}\hat{u}, p_1], \varepsilon, a) :$   
 $f(\hat{u}) + f(p_1) + f(\bar{b}) + f(a) - f(\hat{b}) - f(\hat{u}) - f(p_1) - f(a)$   
 $> 0$ , da  $f(\bar{b}) > f(\hat{b})$ .
- (8)  $\delta([\hat{u}, p_1], \bar{b}, a) = ([\hat{b}\hat{u}, p_0], \varepsilon, a) :$   
 $f(\hat{u}) + f(p_1) + f(\bar{b}) + f(a) - f(\hat{b}) - f(\hat{u}) - f(p_0) - f(a)$   
 $> 0$ , da  $F(\bar{b}) > f(\hat{b})$  und  $f(p_1) > f(p_0)$ .
- (9)  $\delta([\hat{u}, p_1], \perp, a) = ([\hat{u}, p_0], \perp, a) :$   
 $f(\hat{u}) + f(p_1) + f(\perp) + f(a) - f(\hat{u}) - f(p_0) - f(\perp) - f(a)$   
 $> 0$ ,  $f(p_1) > f(p_0)$ .

Somit ist  $M$  schrumpfend bezüglich  $f$ .  $\diamond$

Mit obigem Lemma 13 ist auch der Beweis von Satz 12 abgeschlossen.  $\diamond$

Somit können wir schrumpfende konfluente Wortersetzungssysteme effizient mit einem Automaten simulieren. Wenn wir den Algorithmus aus dem Beweis zu Satz 12 so ergänzt haben, daß der resultierende Automat auch generalisierte Church-Rosser-Systeme simulieren kann, sind wir am Ziel. Der Beweis des folgenden Satzes wird das Problem lösen, daß durch die Randmarkierungen  $t_1$  und  $t_2$  entsteht, die vor der Simulation des eigentlichen Systems an das Eingabewort in schrumpfender Art und Weise angefügt werden müssen.

**Satz 14** [Bun96] Jede Church-Rosser-Sprache wird von einem sDTPDA akzeptiert, d.h., für jedes generalisierte Church-Rosser-System  $S = (\Sigma, \Gamma_S, R, t_1, t_2)$  existiert ein sDTPDA  $M = (Q, \Sigma, \Gamma_M, \delta, q_0, \perp, \{q_{fin}\})$ , so daß gilt:

$$L(S) = N(M)$$

*Beweis:*

Im rechten Keller steht zu Beginn die Eingabe  $w$ . Wir werden jetzt einen Weg angeben, wie  $M$  auf schrumpfende Weise die Eingabe mit  $t_1$  und  $t_2$  umrahmt, so daß daraufhin der Automaten aus Satz 12 das WES  $R$  simulieren kann.

Dazu nehmen wir wieder Alphabetkopien an, die wir  $\Gamma_M$  hinzufügen:  $\bar{\Gamma}_S$  und  $\bar{\bar{\Gamma}}_S$  seien jeweils Kopien von  $\Gamma_S$  und  $f$  eine Bewertung für  $M$  mit:

$$\forall a \in \Gamma_S : f(a) > f(\bar{a}) > f(\bar{\bar{a}})$$

Wir führen zwei neue Zustände  $q_{t_1}, q_{t_2} \in Q$  ein. Dann soll für  $f$  gemäß obiger Bedingung mit  $f(t_1), f(t_2)$  beliebig und  $f(p_0)$  wie in Satz 12 gelten:

$$\begin{aligned} f(q_{t_1}) &=_{def} 1 + f(\bar{\bar{t}}_1) + f(p_0) \\ f(q_{t_2}) &=_{def} 1 + f(\bar{\bar{t}}_2) + f(q_{t_1}) \end{aligned}$$

Des weiteren werden die Regeln

$$\begin{aligned} \delta(q_{t_2}, \bar{b}, a) &=_{def} \begin{cases} (q_{t_2}, \bar{\bar{b}a}, \varepsilon) & \text{falls } a \neq \perp \\ (q_{t_1}, \bar{b}, \bar{\bar{t}}_2\perp) & \text{falls } a = \perp \end{cases} \quad \begin{matrix} (1) \\ (2) \end{matrix} \\ \delta(q_{t_1}, \bar{b}, \bar{\bar{a}}) &=_{def} \begin{cases} (q_{t_1}, \varepsilon, \bar{\bar{b}a}) & \text{falls } \bar{b} \neq \perp \\ ([\varepsilon, p_0], \perp, \bar{\bar{t}}_1\bar{\bar{a}}) & \text{falls } \bar{b} = \perp \end{cases} \quad \begin{matrix} (3) \\ (4) \end{matrix} \end{aligned}$$

hinzugefügt. Diese sind mit obiger Bewertung offenbar schrumpfend.

Im Zustand  $q_{t_2}$  kopiert der Automat also die Eingabe vom rechten in den linken Keller und führt dabei einen Alphabetwechsel nach  $\bar{\Gamma}_S$  durch (1). Ist der rechte Keller leer, schreibt er  $t_2$  hinein und wechselt über in Zustand  $q_{t_1}$  (2).

Jetzt wird von links nach rechts zurückkopiert (3), allerdings ist das Zielalphabet nun  $\bar{\bar{\Gamma}}_S$ . Ist dann wiederum der linke Keller leer, wird  $t_1$  vor das Eingabewort in den rechten Keller geschrieben und in den Zustand  $[\varepsilon, p_0]$  übergegangen (4), in dem der Automat aus Satz 12 seine Arbeit beginnt<sup>4</sup>.  $\diamond$

Jetzt können wir sofort den abschließenden Satz folgern:

**Satz 15** Die Church-Rosser-Sprachen werden durch schrumpfende deterministische Zweikellerautomaten charakterisiert, d.h., es gilt  $CRL = DGCSL$

*Beweis:*

Folgt sofort aus Satz 11 und Satz 14.  $\diamond$

---

<sup>4</sup>Das Eingabealphabet dieses Automaten entspricht dann genau  $\bar{\bar{\Gamma}}_S$ , so daß auch die neuen Symbole  $\bar{\bar{t}}_1$  und  $\bar{\bar{t}}_2$  darin enthalten sind.

## 3 Praktische Umsetzung

### 3.1 Aufgaben des Programms

In erster Linie wurde das Programm entwickelt, um die Funktionsweise eines deterministischen Zweikellerautomaten graphisch und möglichst interaktiv zu veranschaulichen.

### 3.2 Verwendete Algorithmen

#### 3.2.1 Die Simulation des Automaten

Um den Simulationsalgorithmus, der in den Beweisen zu Satz 12 und Satz 14 angegeben wird, auf dem Rechner umzusetzen, muß man zunächst entscheiden, welche praktischen Voraussetzungen dafür zu machen sind.

Wir haben versucht, bereits im theoretischen Abschnitt unter Berufung auf die praktische Umsetzung darauf hinzuweisen, warum bestimmte Voraussetzungen gemacht werden. Dies wollen wir an dieser Stelle vertiefen.

Der wohl wichtigste Aspekt bei der Übersetzung eines theoretischen Algorithmus in die Praxis ist die Sicherheit, daß dieser immer terminiert. Unser Automat wurde für schrumpfende Wortersetzungssysteme entworfen, jedoch soll der Anwender nicht daran gehindert werden, auch beliebige andere Systeme von dem Automaten simulieren zu lassen (oder dies zumindest zu versuchen), ohne daß das Programm beispielsweise in eine Endlosschleife gerät.

Eine einfache Möglichkeit, eine nichtterminierende Simulation zu verhindern, ist, eine maximale Anzahl an auszuführenden Schritten zu spezifizieren. Diese Lösung wurde hier auch verwendet: Erreicht der Automat diese Grenze, wird der Automat angehalten und der Anwender informiert. Dieser kann dann selbst entscheiden, ob er dem Automat nochmals eine gewisse Anzahl von Schritten spendiert, oder eben nicht.

Des weiteren tritt bei einer praktischen Umsetzung stets die Frage auf, wie exakt man die theoretische Vorlage umsetzen sollte. Im Falle eines Programms, daß den theoretischen Algorithmus demonstrieren soll (so wie das hier der Fall ist) muß die Antwort „möglichst genau“ lauten.

Andererseits soll die Demonstration in diesem Fall auch Interesse an einer „echten“ Anwendung der Methode, nämlich bei der Syntaxüberprüfung, wecken. Allzu viele technische Details können diesem Zweck entgegenwirken. Daher weicht die praktische Umsetzung in folgenden Punkten von der theoretischen Vorlage ab:

- Es wurde auf die Benutzung der Alphabetskopien verzichtet. Diese wurden in den Beweisen verwendet, um ein und dasselbe Symbol in verschiedenen Komponenten des Automaten unterschiedlich zu bewerten und so nachzuweisen, daß der Automat schrumpfend ist. Da der Automat jedoch letztendlich symbolweise arbeitet, können wir uns diesen Overhead in der

Praxis sparen und die Originalsymbole in jeder Automatenkomponente verwenden.

- Das in Satz 14 beschriebene Erweitern des Ausgangsalgorithmus zum Akzeptieren von Sprachen CRL wird vereinfacht. Anstatt das Wortanfangs- und -endsymbol durch Verfrachten der Eingabe in den linken Keller und dann wieder zurück in den rechten Keller anzufügen, umrahmen wir das Eingabewort mit diesen Symbolen schon vor der Eingabe in den Automaten. So benötigen wir lediglich konstant zwei statt linear in der Eingabelänge viele Schritte für diese eigentlich einfachen Operationen. Von der theoretischen Vorlage weichen wir also dem praktisch interessierten Anwender zuliebe ab: Falls die Eingabe beispielsweise ein Programm ist, das auf korrekten Syntax überprüft werden soll, spart man sich somit zwei komplette „Durchläufe“ des unter Umständen sehr langen Programms.

### 3.2.2 Überprüfen eines Wotersetzungssystems auf Konfluenz

**Algorithmus 16** [BO93] *Überprüfung eines noetherschen Wotersetzungssystems  $R$  auf Konfluenz*

1. Finde “kritische” Wortpaare, d.h für jedes Regelpaar  $(l_i, r_i), (l_j, r_j)$  aus  $R$  überprüfe
  - ob  $i \neq j$  und  $l_i = xl_jy$  ( $x, y \in \Sigma^*$ ) gilt.  
Falls ja, merke man sich das Wortpaar  $(r_i, xl_iy)$  als kritisch. Achtung:  $l_j$  kann auch mehrfach in  $l_i$  auftauchen, jedes so entstehende Wortpaar ist dann kritisch.
  - ob  $l_ix = yl_j$  ( $x, y \in \Sigma^*$ ) mit  $|x| < |l_j|$  gilt.  
Falls ja, merke man sich das Wortpaar  $(r_ix, yr_j)$  als kritisch.
2. Überprüfe unter Anwendung des Zweikellerautomaten (dieser terminiert, da das eingegebene System nach Voraussetzung noethersch ist), ob beide Wörter eines kritischen Paares durch wiederholte Linksableitung auf dasselbe irreduzible Wort (welches ebenfalls wegen der Noetherzität existiert) reduziert werden. Falls dies nicht zutrifft, kann  $R$  nach Definition nicht konfluent sein.  
Tritt dieser Fall für keines der Paare ein, ist  $R$  konfluent.

Die meisten Leser stellen sich nach dem ersten Betrachten des Algorithmus vermutlich folgende Fragen:

- (a) Sind das wirklich *alle* kritischen Paare? Warum ergibt sich aus einem Wort  $w = xl_1yl_2z$  für  $x, y, z \in \Sigma^*$  und  $(l_1, r_1), (l_2, r_2) \in R$  kein kritisches Paar?
- (b) Warum benutzen wir in Schritt 2 die Linksableitung; es ist doch möglich daß die Ableitungen zwar über eine Linksableitung “zusammenfließen”, daß es aber auf dem Weg zu den jeweiligen irreduziblen Wörtern auch andere Ableitungsmöglichkeiten gab?

zu (a):

Gibt es keine Überschneidung der linken Regelseiten  $l_1$  und  $l_2$  in  $w$ , so bedeutet dies, daß man nach Anwenden einer der beiden Regeln immernoch genauso die jeweils andere anwenden kann. Zerstört eine andere Regel irgendwann im Laufe der weiteren Ableitungskette diese Möglichkeit, so muß sie dies durch eine Überschneidung gemäß 1. geschafft haben, und diese wird oder wurde bereits zu einem anderen Zeitpunkt separat behandelt, da wir in 1. wirklich *alle* kritischen Paare herausuchen.

zu (b):

Sei ein kritisches Paar  $(u, v)$  gegeben.

Wir leiten nun  $u$  und  $v$  auf irgendeine Weise zu den irreduziblen Wörtern  $u_{\text{irr}}$  und  $v_{\text{irr}}$  ab. Es existieren die folgenden Möglichkeiten:

1. Es gilt  $u_{\text{irr}} \neq v_{\text{irr}}$ . Wir sind fertig, denn die Konfluenzbedingung ist verletzt.
2. Es gilt  $u_{\text{irr}} = v_{\text{irr}}$ . Wir behaupten, daß wir dieses Paar nicht weiter betrachten müssen. Wir liegen damit richtig, wie wir durch die Betrachtung der möglichen Unterfälle zeigen:
  - 2.1. Alle anderen Ableitungspfade, die wir nicht ausprobiert haben, führen zu demselben Ergebnis  $u_{\text{irr}} = v_{\text{irr}}$ , aus diesem Paar kann man folglich keine Verletzung der Konfluenz folgern.
  - 2.2. Es existiert ein Paar von Ableitungspfaden für  $u$  und  $v$ , so daß  $u_{\text{irr}} \neq v_{\text{irr}}$  gilt. Daraus folgt aber, daß es für  $u$  oder für  $v$  zwei verschiedene irreduzible Entwörter geben muß. Sei dies o.B.d.A. für  $u$  der Fall, so existiert auf dem Ableitungspfad von  $u$  offenbar ein weiteres kritisches Paar, so daß von diesem wiederum Ableitungspfade mit verschiedenen irreduziblen Endwörtern gibt. Denkt man diesen Gedanken zuende, so muß es irgendwann in der Ableitungskette ein kritisches Paar geben, dessen sämtliche Ableitungsmöglichkeiten zu dem Ergebnis 1. führen müssen. Da wir nach Definition des Algorithmus alle kritischen Paare betrachten, wird auch dieses noch überprüft, und die Verletzung gemäß 1. korrekt erkannt.

Somit ist die Wahl der Ableitungsart frei, sie darf sogar innerhalb eines Ableitungsstranges beliebig variiert werden.

### 3.2.3 Überprüfen, ob ein Wortersetzungssystem schrumpfend ist

Es ist leicht, zu prüfen, ob ein Wortersetzungssystem längenverkürzend ist. Aber zu prüfen, ob es schrumpfend ist, ist ungleich schwerer, da *die Existenz* einer schrumpfenden Bewertung nachzuweisen ist.

Versucht man, eine solche Bewertung zu finden, so ergibt sich aus der Bedingung, daß jede Regel schrumpfend sein muß, ein lineares Ungleichungssystem der Form  $Ax > b$ , wobei sowohl für  $A$  als auch für  $b$  ausschließlich ganzzahlige und für den Gewichtsvektor  $x$  zunächst sogar nur natürliche Einträge zugelassen sind. Da jedoch der Vektor  $b$  in unserem Fall die minimale Differenz zwischen den Wortwerten der linken und rechten Seite einer Regel darstellt (also  $b = (1 \ 1 \ \dots \ 1)^T$ )

und somit in jeder Komponente positiv ist, kann man aus jeder Lösung  $x \in \mathbb{Q}^+$  durch Multiplikation mit dem kleinsten gemeinsamen Vielfachen der Nenner der einzelnen Lösungskomponenten einen Vektor  $\tilde{x} \in \mathbb{N}$  berechnen, welcher das Ungleichungssystem ebenfalls löst.

Peter van Emde Boas schreibt nun in [vEB79], daß obiges Problem des Findens einer positiven rationalen Lösung polynomiell äquivalent zum Problem des allgemeinen linearen Programmierens ist. Demzufolge existiert ein Lösungsalgorithmus für unser Problem, welcher lediglich polynomielle Zeit benötigt. Aus Zeitgründen wurde in der Version 1.0 von `TPDAsim` jedoch noch kein solcher Algorithmus implementiert, hinter dem Menüpunkt "Testen ob schrumpfend" verbirgt sich lediglich das Überprüfen einer manuell eingegebenen Bewertung.

## 4 Dokumentation: TPDAsim

### 4.1 Installation

#### 4.1.1 Systemvoraussetzungen

`TPDAsim` benötigt die freie Qt-Bibliothek von Troll Tech ab Version 1.3. Entwickelt und getestet wurde das Programm unter dem Linux X Window System.

#### 4.1.2 Herunterladen der Dateien

Quellcode und Binaries liegen auf der `TPDAsim`-Webseite <http://www.tus.muebebeck.de/homepages/kuhnigk/tpdasim/index.html> als RPM- und TGZ-Dateien für Linux 2.0 oder höher zum Herunterladen bereit.

`TPDAsim` unterliegt der GNU GENERAL PUBLIC LICENSE, und darf somit frei kopiert werden.

Nach der Installation liegt die ausführbare Datei `tpdasim` im Verzeichnis `/usr/local/bin/` und alle weiteren zur Ausführung benötigte Dateien im Verzeichnis `/usr/local/lib/tpdasim/`. Des Weiteren wurden einige Beispielgrammatiken nach `/usr/local/lib/tpdasim/samples/` kopiert.

#### 4.1.3 Installation mittels RPM

Um das Programm mit Hilfe des `Redhat Package Managers` zu installieren muß man erstens die `.rpm`-Version von `TPDAsim` vorliegen haben als auch die `root`-Rechte auf dem entsprechenden Linux-Rechner.

Aus einer `root`-Shell heraus erfolgt die Installation dann über

```
rpm -i <Name des RPM-Files>
```

, während die Deinstallation durch das Kommando

```
rpm -e <Name des RPM-Files ohne .rpm>
```

erreicht wird.

#### 4.1.4 Installation der TGZ-Version

Die zu installierende Datei hat dabei die Endung `.tgz` oder `.tar.gz`. Zuerst müssen die Quelldateien mit

```
tar -xvzf <TGZ-Dateiname>
```

entpackt werden. Dabei wird ein Unterverzeichnis erstellt, welches den Namen `tpdasim-<Versionsnummer>` besitzt. Wechselt man von einer *root*-Shell aus in dieses Verzeichnis, kann man das Programm mit den Kommandos

```
make install
make uninstall
```

wahlweise installieren oder deinstallieren.

## 4.2 Bedienungsanleitung

### 4.2.1 Starten des Programms

Das Programm wird durch Eingabe von `tpdasim <return>` aus einer Shell heraus gestartet. Dabei muß der Pfad der ausführbaren Datei ( $\hookrightarrow$  4.1) im Suchpfad des Anwenders eingetragen oder dem Programmaufruf vorangestellt sein. Optional ist ein Parameter, mit dem man eine direkt nach Programmstart zu ladende Datei spezifizieren kann, beispielsweise `tpdasim thue.tsm <return>`.

### 4.2.2 Neue Grammatik erstellen

Über Auswahl des Menüpunkts DATEI/NEU/GRAMMATIK wird eine leere Grammatikdatei mit dem vorläufigen Namen `neueGRM.tsm` in den Editor ( $\hookrightarrow$  4.2.7) geladen. Diese Datei ist nicht leer, sondern erhält bereits die Steuersymboldefinitionen ( $\hookrightarrow$  4.2.2.1) und die wichtigsten Schlüsselwörter. So müssen im einfachsten Fall lediglich die Regeln eingetragen werden.

Zunächst eine Anmerkung zur Bezeichnung: Wenn wir das Wort *Symbol* benutzen, meinen wir nicht zwangsläufig ein einzelnes Zeichen, es kann sich auch eine zusammenhängende Zeichenkette handeln. Als Trennglied zwischen zwei Symbolen wirkt das *Separatorsymbol* ( $\hookrightarrow$  4.2.2.1).

Eine Datei gliedert sich in zwei große Abschnitte:

**4.2.2.1 Definition der Steuersymbole** In diesem notwendig am Anfang der Datei stehenden Abschnitt werden die Steuersymbole für den zweiten Abschnitt festgelegt. Diese Sektion ist notwendig, da das Alphabet der eingegebenen Grammatik disjunkt zu der Menge der Steuersymbole sein muß. Kommen einige der Steuersymbole in der gewünschten Grammatik vor, so muß der Programmierer nicht seine Grammatik sondern nur die Steuersymbole ändern. Wir können den folgenden Steuersymbole durch Verwendung des Syntax

$$\langle \text{Steuersymbolname} \rangle = \text{„}\langle \text{Steuerzeichen(-kette)} \rangle\text{”}$$

einen String zuweisen:

- **schluesselwort-beginn, schluesselwort-ende:** Mit diesen Strings wird ein Schlüsselwort geklammert, damit es als solches erkannt werden kann.  
*Voreinstellung:* „<“, „>“  
*Beispiel:*  
<Regeln>
- **zuweisung:** Das Zuweisungssymbol wird im Abschnitt *Alphabet* verwendet, um Symbolen einen Wert zuzuweisen.  
*Voreinstellung:* „=“  
*Beispiel im Unterabschnitt Alphabet (↔ 4.2.2.2.2):*  
Programm = 2
- **separator:** Ein Separatorsymbol trennt zwei Symbole. Dies ist notwendig, um auch Grammatiksymbole mit mehr als einem Zeichen zuzulassen.  
*Voreinstellung:* „ „  
*Beispiel im Unterabschnitt Regeln (↔ 4.2.2.2.3):*  
Programm -> Beginn Anweisungssequenz Ende
- **pfeil:** Der Pfeil trennt im Unterabschnitt *Regeln* (↔ 4.2.2.2.3) die linke Seite einer Regel von der rechten.  
*Voreinstellung:* „->“  
*Beispiel:*  
Programm -> Beginn Anweisungssequenz Ende
- **einzeiliger-kommentar:** Das hier definierte Steuersymbol steht einem sinzeiligen Kommentar voran. Dieser wird bei der Interpretation nicht berücksichtigt und endet mit dem Ende der Zeile.  
*Voreinstellung:* „//“  
*Beispiel:*  
<Alphabet>  
// Umrahmende Symbole:  
Beginn = 10  
Ende = 10  
...
- **mehrzeiliger-kommentar-beginn, mehrzeiliger-kommentar-ende:** Mit diesen Zeichen muß ein mehrzeiliger Kommentar umrahmt werden, damit er vom Parser ignoriert wird. Auf diese Art kann man ebenfalls nicht mehr gewünschte Abschnitte auskommentieren, auch die Verschachtelung von Kommentaren ist zugelassen.  
*Voreinstellung:* „/\*“, „\*/“  
*Beispiel:*  
<Regeln>  
/\*Zuerst kommt die Regel, welche aus einem Programm eine Anweisungssequenz erzeugt, die von 'Beginn' und 'Ende' umrahmt wird.\*/  
Programm -> Beginn Anweisungssequenz Ende  
...

- **weiter-in-naechster-zeile:** Um beispielsweise eine Regel in der nächsten Zeile fortsetzen zu können, wird am Ende der ersten Zeile dieses Symbol benötigt, da der Parser im Normalfall das Regelende am Zeilenende annimmt.

Voreinstellung: „\”

Beispiel im Abschnitt *Regeln* ( $\leftrightarrow$  4.2.2.2.3):

```

Programm -> \
Beginn \
Anweisungssequenz \
Ende
...

```

Wird eine dieser Definitionen ganz weggelassen, so gilt die jeweilige Voreinstellung. Außerdem ist die Reihenfolge der Definitionen beliebig, da die neuen Symbole erst ab dem ersten Lesen eines Schlüsselwort-Beginn-Symbols gelten. **Innerhalb des Steuersymboldefinitionsteils gelten immer die voreingestellten Symbole.**

**4.2.2.2 Definition der Grammatik** Die Grammatikdatei wird in diesem Teil von sogenannten Schlüsselwörtern gegliedert, die die Art des nächsten Abschnittes anzeigen, der dann bis zum nächsten Auftauchen eines Schlüsselworts reicht.

Wir werden die verwendbaren Schlüsselwörter im folgenden vorstellen. Ein Schlüsselwort wird von den oben definierten Schlüsselwortbeginn- und -endensymbolen geklammert. Groß- und Kleinschreibung ist dazwischen unbedeutend.

Die Grammatikdefinition beginnt mit dem Schlüsselwort `<Beginn Grammatik>`<sup>5</sup>. Der Abschnitt und damit auch der interpretierte Teil der Datei endet mit dem Schlüsselwort `<Ende>`.

Dazwischen sind die folgenden Unterabschnitte zulässig, bei deren Reihenfolge lediglich zu beachten ist, daß der `<Alphabet>`-Teil vor dem `<Regeln>`-Unterabschnitt kommt:

#### 4.2.2.2.1 Der Unterabschnitt `<Info>` (*optional*)

Hier kann man die Grammatik in ein paar Zeilen kommentieren. Dieser Kommentar besitzt eine Sonderstellung, er muß durch keine Steuersymbole gekennzeichnet sein und endet mit dem nächsten Auftreten eines Schlüsselwort-Beginn-Symbols.

#### 4.2.2.2.2 Der Unterabschnitt `<Alphabet>` (*optional*)

Hier können die in der Grammatik vorkommenden Symbole aufgezählt und auf ihnen eine Bewertung definiert werden:

$$Symbol [ = Symbolwert ]$$

Der in eckigen Klammern angegebene Teil ist optional, wird er weggelassen, erhält das so definierte Symbol den Wert 1.

<sup>5</sup>Beim Beschreiben der Definition einer Grammatik werden wir der Einfachheit wegen die voreingestellten Steuersymbole benutzen.

Andernfalls sollte der angegebene Wert eine nichtnegative ganze Zahl sein, wobei die 0 eigentlich nur zur Bewertung des akzeptierenden Symbols sinnvoll ist. Wird ein Symbol nicht in diesem Unterabschnitt angegeben, kommt aber später in einer Regel vor, so wird es automatisch dem Alphabet mit dem Standardwert 1 hinzugefügt.

#### 4.2.2.2.3 Der Unterabschnitt `<Regeln>` (*obligatorisch*)

Dies ist der einzige Unterabschnitt, der in jeder Grammatik auftauchen muß. Die Regeln werden zeilenweise gemäß dem Syntax

*Ausgangswort -> erzeugtesWort*

eingetragen, wobei ein Wort aus beliebig vielen, durch Separatorsymbole getrennten Grammatiksymbolen besteht.

Durch die Regeln kann außerdem implizit ein Startsymbol definiert werden. Lesen Sie dazu bitte den nächsten Abschnitt 4.2.2.2.4.

#### 4.2.2.2.4 Der Unterabschnitt `<Startsymbol>` (*optional*)

Damit eine Grammatik erfolgreich vom Parser verarbeitet werden kann, muß für den Parser ein Startsymbol erkennbar sein.

Er erkennt es normalerweise im Regelabschnitt ( $\leftrightarrow$  4.2.2.2.3) automatisch daran, daß es entweder durch eine explizite Erzeugungsregel mit leerer linker Seite ( $->$  *Startsymbol*) angezeigt wird, oder, wenn diese Regel fehlt, dadurch, daß es das einzige Symbol ist, was ausschließlich auf linken Regelseiten vorkommt (dort muß es zusätzlich alleine stehen).

Sind diese Bedingungen für kein Symbol erfüllt, oder treffen sie auf mehrere verschiedene Symbole zu, und will man trotzdem eines der Symbole zum eindeutigen Startsymbol bestimmen, so benutzt man das Schlüsselwort `<Startsymbol>`. Der einzige Inhalt dieses Abschnittes ist dann das Startsymbol.

Da die automatische Erkennung in aller Regel ausreicht, ist dieser Abschnitt nicht standardmäßig in einer neuen Datei vorhanden.

### 4.2.3 Neues Wortersetzungssystem erstellen

Über Auswahl des Menüpunkts DATEI/NEU/WES wird eine Datei mit einem leeren Wortersetzungssystem und dem vorläufigen Namen `neueWES.tsm` geöffnet.

Eine WES-Datei unterscheidet sich von einer Grammatik-Datei lediglich in folgenden Punkten:

- Der Beginn des Definitionsabschnitts wird mit dem Schlüsselwort `<Beginn WES>` angezeigt. So kann der Parser erkennen, daß es sich um eine WES-Datei handelt.
- Im Regelabschnitt werden die Regeln „umgedreht“ angegeben, d.h., besitzt eine Grammatik die erzeugende Regel `A -> B C`, so enthält das äquivalente Wortersetzungssystem die reduzierende Regel `B C -> A`. Der Syntax lautet also:

*Ausgangswort -> reduziertesWort*

- Es muß nun statt eines eindeutigen Startsymbols ein eindeutiges akzeptierendes Symbol existieren. Dies kennzeichnet man entweder explizit mit einer löschenden Regel (z.B.  $Y \rightarrow$ ), oder man stellt sicher, daß das akzeptierende Symbol das einzige ist, was ausschließlich auf der rechten Seite von Regeln vorkommt, und dort auch noch „solo“. Will man den Parser zu einem bestimmten akzeptierenden Symbol „zwingen“, so heißt das zu verwendende Schlüsselwort `<Akzeptierendes Symbol>`.

---

**Da Grammatiken durch den Parser zu Wortersetzungssystemen umgeformt werden, sprechen wir in Zukunft ausschließlich von Wortersetzungssystemen.**

---

#### 4.2.4 Laden eines Wortersetzungssystems

Wählt man im DATEI-Menü den Unterpunkt ÖFFNEN aus, so erscheint eine Dialogbox, in der man die zu öffnende Datei auswählen kann. Dabei wird als Standard-Dateiendung `.tsm` angenommen.

Anschließend wird sogleich der Editor ( $\leftrightarrow$  4.2.7) gestartet.

#### 4.2.5 Speichern eines Wortersetzungssystems

Wählt man im DATEI-Menü den Unterpunkt SPEICHERN aus, und ist die editierte Datei nicht mehr neu, so wird die Datei unter ihrem aktuellen Namen abgespeichert.

Wurde die Datei noch nie gespeichert oder wurde im DATEI-Menü SPEICHERN UNTER ausgewählt, so erscheint zunächst ein Dialogfenster, in dem man den Dateinamen spezifizieren kann. Dabei wird als Standard-Dateiendung `.tsm` angenommen, gibt man keine Endung an, so wird `.tsm` automatisch angefügt.

Existiert bereits eine Datei des gewählten Namens, so wird der Benutzer um eine Bestätigung des Überschreibevorgangs gebeten.

Wurde die Datei seit dem letzten Speichern verändert, wird in der Statuszeile des Editors das Wort 'UNGESPEICHERT' eingeblendet. Dieses verschwindet wieder nach einem erfolgreichen Speichervorgang.

#### 4.2.6 Beenden des Programms

Durch den Menübefehl DATEI/BEENDEN wird das Programm beendet.

Ist zu diesem Zeitpunkt eine ungespeicherte Datei geöffnet, so wird der Benutzer gefragt, ob er diese zunächst speichern will. Wählt er ja, so passiert dasselbe, als hätte man den Menüpunkt DATEI/SPEICHERN UNTER ( $\leftrightarrow$  4.2.5) aufgerufen. Beim Klick auf ABBRECHEN wird das Beenden des Programms abgebrochen, also mit dem Programm fortgefahren.

#### 4.2.7 Editieren eines Wortersetzungssystems

Befindet sich bereits eine Datei im Speicher, so kann über den Menüpunkt WES/EDITIEREN der Editor aufgerufen werden.

Dieser besteht aus zwei Teilen: Dem eigentlichen Eingabeteil, in dem das zu editierende Wortersetzungssystem angezeigt wird, und einer am unteren Ende anschließenden Statuszeile.

Dort wird am linken Rand die Zeilennummer angezeigt, während rechts die Wörter 'UNGESPEICHERT' ( $\leftrightarrow$  4.2.5) und 'UNPARSIERT' ( $\leftrightarrow$  4.2.8) stehen können.

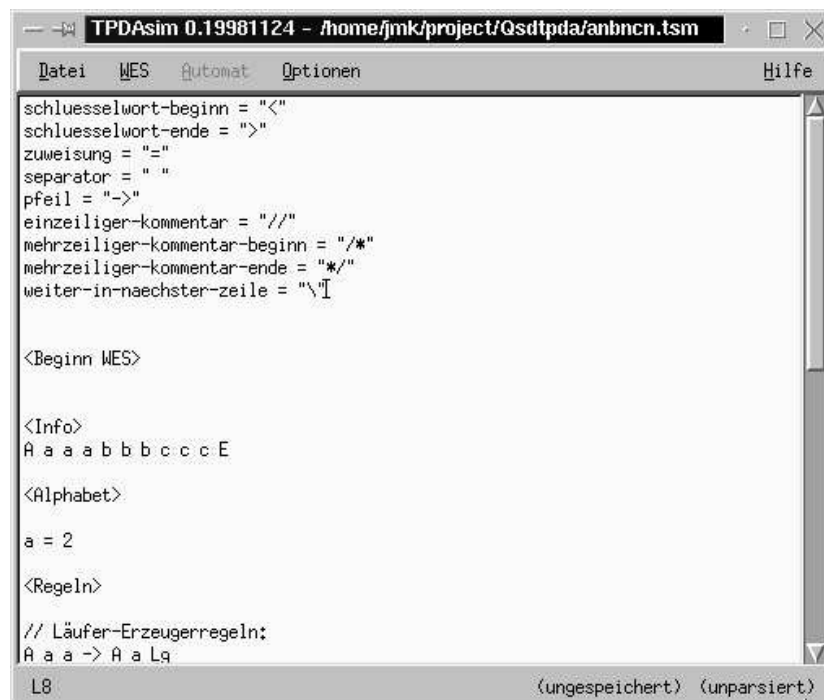


Abbildung 2: Der Editor

#### 4.2.8 Parsieren eines Wortersetzungssystems

Über den Menüpunkt WES/PARSIEREN wird der Parser gestartet, der das Programm zu interpretieren versucht.

Dabei wird immer das aktuell im Editor befindliche Wortersetzungssystem parsiert. Wurde seit der letzten Änderung im Editor noch nicht parsiert, so steht am rechten Rand der Statuszeile im Editor das Wort 'UNPARSIERT', um anzuzeigen, daß sich zu diesem Zeitpunkt sämtliche Aktionen (beispielsweise die Simulation des Automaten) auf eine von der im Editor befindlichen möglicherweise abweichendes, nämlich zu einem früheren Zeitpunkt parsiertes Wortersetzungssystem bezieht.

Scheitert der Parser, so gibt er eine entsprechende Fehlermeldung ( $\leftrightarrow$  4.3.2) aus, und springt mit dem Cursor in die Programmzeile, an der der Fehler aufgetreten ist. Im Erfolgsfall dagegen werden sämtliche Menüpunkte aktiviert. Ist man sich

der Richtigkeit des eingegebenen Systems sicher, so kann man jetzt bereits den Automaten starten.

---

**Die folgenden Funktionen sind erst nach einem erfolgreichen Parserdurchlauf erhältlich.**

---

#### 4.2.9 Analysieren eines Wortersetzungssystems

Falls das Programm versuchen soll, die Symbole des Wortersetzungssystems ihrer Bedeutung nach einzuordnen, kann man im DATEI-Menü den Punkt ANALYSIEREN anwählen. Standardmäßig wird das Wortersetzungssystem allerdings automatisch direkt nach dem Parsieren ( $\leftrightarrow$  4.2.8) analysiert, also ist dieser Menüpunkt normalerweise nicht mehr anwählbar<sup>6</sup>.

Beim Analysieren versucht TPDASIM, das Eingabealphabet und die Wortanfangs- und Wortendesymbole des Wortersetzungssystems zu identifizieren.

Eingabesymbole werden daran erkannt, daß sie ausschließlich auf der linken Seite vorkommen; Wortanfangs- und Wortendesymbole daran, daß sie immer nur am linken bzw. rechten Ende eines Wortes einer beliebigen Regelseite auftauchen.

Wurde ein Wortersetzungssystem erfolgreich analysiert, kann die Darstellung über die ANZEIGEN-Menüpunkte entsprechend angepaßt werden ( $\leftrightarrow$  4.2.10). Konnten außerdem genau ein Wortanfangssymbol und ein Wortendesymbol ausfindig gemacht werden, die gleichzeitig im Eingabealphabet enthalten sind (was gerade für Church-Rosser-Systeme interessant ist), so besteht später bei der Simulation des Automaten die Möglichkeit, diese Symbole automatisch an das Eingabewort einfügen zu lassen ( $\leftrightarrow$  4.2.14).

#### 4.2.10 Anzeigen eines Wortersetzungssystems

Wählt man im DATEI-Menü den Punkt ANZEIGEN, so kann man sich wahlweise die per Info-Abschnitt ( $\leftrightarrow$  4.2.2.2.1) eingegebene Zusatzinformation, das Alphabet oder die Regeln des Wortersetzungssystems anzeigen lassen.

Falls dieses nicht analysiert wurde, werden die Eingabesymbole und Regeln ganz normal aufgelistet, im anderen Falle werden sie nach ihrer Klassifizierung angeordnet:

- Bei der Anzeige des Alphabets steht zuoberst das akzeptierende Symbol, dann folgt das Eingabealphabet, dann die erkannten Wortanfangs- und Wortendesymbole und zuletzt alle restlichen Symbole.
- Bei der Anzeige der Regeln werden dementsprechend zuerst alle Regeln angezeigt, die das akzeptierende Symbol enthalten. Es folgen diejenigen, in denen Eingabesymbole vorkommen, vor denjenigen, welche Wortanfangs- und Wortendesymbole enthalten. Schließlich folgen noch alle übrigen Regeln. Die Abschnitte werden jeweils durch eine Leerzeile getrennt.

---

<sup>6</sup>Im Menü OPTIONEN ( $\leftrightarrow$  4.2.15) läßt sich diese Funktion deaktivieren

Die Regeln werden durchnummeriert, damit man eventuelle Fehlermeldungen des Konfluenztests oder die Überprüfung, ob das Wortersetzungssystem schrumpfend ist, nachvollziehen kann. Des Weiteren werden vor jeder Regel die Werte beider Regelseiten angezeigt, die sich aus den Einzelwerten der Symbole ergeben, die man beispielsweise bei der Anzeige des Alphabets nachsehen kann.

#### **4.2.11 Überprüfen eines Wortersetzungssystems auf Konfluenz**

Über den Menüpunkt WES/ÜBERPRÜFEN AUF KONFLUENZ startet man den Konfluenztest. Dort wird der in 16 beschriebene Algorithmus auf dem aktuellen Wortersetzungssystem ausgeführt. Stellt das Programm eine Verletzung der Konfluenzbedingung fest, so werden die beiden verantwortlichen Regeln angegeben, die dasjenige kritische Wortpaar erzeugt haben, welches zu dieser Verletzung geführt hat. Dabei können durchaus weitere Verletzungen gegeben sein, es wird jedoch nur die zuerst gefundene angegeben.

Da bei der Überprüfung der Automat verwendet wird, um den jeweiligen irreduziblen Deszendenten der Wörter des kritischen Paares zu ermitteln, ist es möglich daß dieser (für den Fall daß das Wortersetzungssystem nicht noethersch ist) nicht terminiert. Daher wird die Reduktion nach einer über das OPTIONEN-Menü ( $\leftrightarrow$  4.2.15) einstellbaren Schrittzahl angehalten. Die Meldung, die der Benutzer erhält lautet dann „BEI DER ÜBERPRÜFUNG TERMINIERTE DER AUTOMAT NICHT“ mit der Angabe der zwei Regeln, die das kritische Paar erzeugt haben, wessen Überprüfung das Nicht-Terminieren des Automaten zur Folge hatte.

#### **4.2.12 Überprüfen, ob ein Wortersetzungssystem schrumpfend ist**

Wie bereits erwähnt, überprüfen wir hier nur, ob das Wortersetzungssystem mit der angegebenen Bewertung ausschließlich schrumpfende Regeln enthält. Das kann man bei kleinen Wortersetzungssystemen noch gut selber überblicken ( $\leftrightarrow$  4.2.10), bei großen greift man besser auf den Menüpunkt WES/ÜBERPRÜFEN OB SCHRUMPFEND zurück.

#### **4.2.13 Überprüfen eines Wortersetzungssystems auf Determinismus**

Über den Menüpunkt WES/ÜBERPRÜFEN OB DETERMINISTISCH kann man nachprüfen lassen, ob ein Wortersetzungssystem deterministisch ist. Ist dies nicht der Fall, so werden mit der entsprechenden Meldung die Nummern derjenigen Regeln angegeben, die den Determinismus verletzen.

Allerdings sei hier daran erinnert, daß nicht der Determinismus eines Wortersetzungssystems eine notwendige Bedingung für die Korrektheit der Arbeitsweise des Automaten darstellt, sondern ausschließlich die Konfluenz und die Noetherzität des Systems.

#### 4.2.14 Die Simulation des Automaten

Um die Simulation des durch das Wortersetzungssystem definierten Zweikellerautomaten zu beginnen, wählt man im Menü TPDA den einzigen Menüpunkt STARTEN aus.



Abbildung 3: Eingabefenster für das Eingabewort

Zunächst erscheint ein Dialogfenster ( $\leftrightarrow$  Abb. 3). In diesem kann man entweder ein Eingabewort von Hand eingeben, oder eine Datei zur Eingabe auswählen. Diese Datei sollte einfach eine Textdatei sein, welche die kompletten Eingabe enthält. Sämtliche Zeilenumbrüche darin werden intern als Separatorsymbole ( $\leftrightarrow$  4.2.2.1) interpretiert.

Bei der Eingabe ist normalerweise darauf achten, daß man die Symbole wie bei der Definition des Wortersetzungssystems durch Separatorsymbole voneinander trennt. Allerdings ist es möglich, das Programm anzuweisen, die Separatorsymbole zwischen den Zeichen selbst einzufügen. Dabei ist allerdings zu beachten, daß der Algorithmus lediglich die Eingabe von Anfang bis Ende durchläuft, und nach jedem erkannten Symbol ein Separatorsymbol einfügt. **Das Anwählen der Option „Separatorsymbole automatisch einfügen“ kann somit zu Fehlinterpretationen führen, falls ein Symbol der Grammatik ein anderes als Präfix besitzt.**

Hat man das Wortersetzungssystem vorher analysiert ( $\leftrightarrow$  4.2.9), und wurden bei der Analyse eindeutige Wortanfangs- und Wortendesymbole gefunden, so hat man die Möglichkeit, den Mechanismus „AUTO-EINFÜGEN DER WORTANFANGS- UND WORTENDESIMBOLE“ zu aktivieren.

Nach Aktivierung des START-Buttons wird der Automat angezeigt. Der Zustand ist zu diesem Zeitpunkt „leer“, im linken Keller befindet sich ausschließlich das Kellerbodensymbol, während in den rechten Keller zusätzlich die

Eingabe „eingeworfen“ wurde.

Das Automatenfenster besteht aus den folgenden Teilen ( $\leftrightarrow$  Abb. 4):

- Einer Textzeile am oberen Fensterrand, in der in dem Schritt *vor* dem Anwenden einer Regel aus dem eingegebenen Wortersetzungssystem eben diese angezeigt wird
- Den Knöpfen und Eingabefeldern zur Automatensteuerung im mittleren Fensterteil:
  - Der zuunterst befindliche Knopf SCHLIESSEN beendet die Simulation sofort. Eine Weiterführung zu einem späteren Zeitpunkt ist dann nicht möglich.
  - Im Eingabefeld ANZAHL SCHRITTE kann festgelegt werden, wieviele Schritte der Automat beim Anklicken des Knopfes WEITER nacheinander durchführen soll.
  - Im Eingabefeld VERZÖGERUNG legt man die Länge der Pause fest, die der Automat zwischen zwei aufeinanderfolgenden Schritten einlegen soll.
  - Durch Anwählen des WEITER-Knopfes wird die im Eingabefeld ANZAHL SCHRITTE angegebene Nummer von Schritten simuliert.
  - Klickt man den DURCHLAUFEN-Knopf an, so simuliert der Automat so viele Schritte, bis er entweder terminiert oder die im OPTIONEN-Menü ( $\leftrightarrow$  4.2.15) spezifizierte maximal zulässige Anzahl von Schritten überschritten ist.
- Den beiden Kellern, die sich als einzige Elemente beim vertikalen Abrollen bewegen.

Ist die Simulation des Automaten irgendwann zuende, erscheint (falls dies nicht im OPTIONEN-Menü ( $\leftrightarrow$  4.2.15) deaktiviert wurde) eine Meldung, die den Benutzer über das Ergebnis der Simulation informiert.

In dem Hauptfenster des Programms werden während der Simulation die meisten Menüpunkte deaktiviert. Damit man die Arbeit des Automaten besser nachvollziehen kann, ist es allerdings weiterhin möglich, die ANZEIGEN-Befehle ( $\leftrightarrow$  4.2.10) im WES-Menü anzuwählen.

#### 4.2.15 Konfigurieren der Umgebung

Durch die Unterpunkte des OPTIONEN-Menüs kann man folgende Aspekte des Programms konfigurieren<sup>7</sup>:

- Über PARSER/NACH DEM PARSIEREN ANALYSIEREN kann ein- und ausgeschaltet werden, ob das Wortersetzungssystem nach einem erfolgreichen Durchlauf des Parsers ( $\leftrightarrow$  4.2.8) automatisch analysiert ( $\leftrightarrow$  4.2.9) werden soll.

*Voreinstellung: AN*

---

<sup>7</sup>Die Einstellungen werden bei jedem Programmstart auf die Voreinstellung zurückgesetzt



Abbildung 4: Der Automat bei der Arbeit

- Über TPDA/NACH TPDA-LAUF ERGEBNISFENSTER EINBLENDEN ist einstellbar, ob das Programm nach dem Terminieren der Automaten-simulation eine den Erfolg des Durchlaufs wiedergebende Meldung ausgeben soll.  
*Voreinstellung: AN*
- Über ALLGEMEIN/NUR WARN- UND FEHLERMELDUNGEN EINBLENDEN kann die Ausgabe der durch Mausklick zu bestätigenden Erfolgsmeldungen (z.B.: „Das WES wurde erfolgreich parsiert und analysiert.“) unterdrückt werden. Es werden dann lediglich noch Warn- und Fehlermeldungen angezeigt.  
*Voreinstellung: AUS*
- Wählt man den Unterpunkt ERWEITERT an, so erscheint ein neues Fenster, wo man derzeit lediglich die schon mehrfach erwähnte Obergrenze an Schritten, die der Automat am Stück machen darf, festsetzen kann.

### 4.3 Aufistung und Erklärung der Fehlermeldungen

#### 4.3.1 Fehler bei der Dateiverwaltung

##### 100 DATEI NICHT GEFUNDEN ODER ERLAUBNIS VERWEIGERT:

*Auftreten:*

- Beim Versuch, eine Datei zu öffnen  
*Bekannte Ursachen:*

- Die gewählte Datei existiert nicht
- Man besitzt kein Leserecht für die ausgewählte Datei
- Beim Versuch, eine neue Datei zu erstellen  
*Bekannte Ursachen:*
  - Eine der Dateien `neuGRM.tsm` oder `neuWES.tsm` konnte im Ausführungspfad nicht gefunden werden. Diese beiden Dateien stellen die Vorlagen für neue Programme dar.

**101** DATEI KONNTE NICHT GESCHRIEBEN WERDEN (KEINE ERLAUBNIS?):

*Auftreten:*

- Beim Versuch des Speicherns einer Datei

*Bekannte Ursachen:*

- Es existiert bereits eine Datei des gewählten Namens, für die man keine Schreibrechte hat. Das gilt beispielsweise für die Dateien `neueGRM.tsm` und `neuesWES.tsm`, die beim erstellen einer neuen Datei geöffnet werden. Die aktuelle Datei ist dann unter einem anderen Namen zu speichern
- Man besitzt keine Schreibrechte auf dem gewählten Verzeichnis

**102** SCHREIBFEHLER (PLATTE VOLL?):

*Auftreten:*

- Das Betriebssystem meldet während des Speicherns einer Datei einen Fehler

*Bekannte Ursachen:*

- Platzmangel auf dem gewählten Speichermedium

### 4.3.2 Fehlermeldungen des Parsers

**203** '=' ERWARTET:

*Bekannte Ursachen:*

- Bei der Definition eines Steuersymbols ( $\leftrightarrow$  4.2.2.1) wurde das Zuweisungssymbol nicht gefunden

**204** HIER IST KEIN STEUERSYMBOL ERLAUBT:

*Bekannte Ursachen:*

- Es trat ein Steuersymbol ( $\leftrightarrow$  4.2.2.1) an einer ungültigen Stelle auf

**205** KOMMENTARENDE FEHLT:

*Bekannte Ursachen:*

- Ein geöffneter mehrzeiliger Kommentar ( $\leftrightarrow$  4.2.2.1) wurde nicht geschlossen

**206** KOMMENTARENDE OHNE VORANGEHENDEN BEGINN:

*Bekannte Ursachen:*

- Ein Kommentar wurde geschlossen, ohne daß er vorher geöffnet wurde (↔ 4.2.2.1)

**207 UNBEKANNTER BEZEICHNER:**

*Bekannte Ursachen:*

- Bei der Definition der Steuersymbole wurde ein unbekannter Steuersymbolname angegeben (↔ 4.2.2.1)

**209 UNERWARTETES DATEIENDE:**

*Bekannte Ursachen:*

- Das Dateiende wurde erreicht, ohne daß das Schlüsselwort **Ende** gelesen wurde (↔ 4.2.2.2)

**210 UNBEKANNTES SCHLÜSSELWORT:**

*Bekannte Ursachen:*

- Ein Schlüsselwort ist dem Parser unbekannt (↔ 4.2.2.2)

**211 NUR EIN SCHLÜSSELWORT **Beginn** ... ERLAUBT:**

*Bekannte Ursachen:*

- Das erste Schlüsselwort muß entweder **Beginn Grammatik** oder **Beginn WES** lauten (↔ 4.2.2.2)

**212 UNGÜLTIGE POSITION DES ABSCHNITTES 'INFO':**

*Bekannte Ursachen:*

- Der **Info**-Abschnitt (↔ 4.2.2.2.1) wurde vor einem **Beginn ...**-Schlüsselwort oder nach dem **Ende**-Schlüsselwort plaziert
- Es ist mehr als ein **Info**-Abschnitt vorhanden

**213 UNGÜLTIGE POSITION DES ABSCHNITTES 'ALPHABET':**

*Bekannte Ursachen:*

- Der **Alphabet**-Abschnitt (↔ 4.2.2.2.2) wurde vor einem **Beginn ...**-Schlüsselwort oder nach dem **Ende**-Schlüsselwort plaziert
- Der **Alphabet**-Abschnitt wurde erst nach dem **Regeln**-Abschnitt eingegeben
- Es ist mehr als ein **Alphabet**-Abschnitt vorhanden

**214 DIESES SCHLÜSSELWORT IST NUR IN GRAMMATIK-DATEIEN ERLAUBT:**

*Bekannte Ursachen:*

- Das Schlüsselwort **Startsymbol** (↔ 4.2.2.2.4) wurde in einer **WES**-Datei (↔ 4.2.3) verwendet, obwohl seine Benutzung auf **Grammatik**-Dateien (↔ 4.2.2.2) eingeschränkt ist.

**215 UNGÜLTIGE POSITION DES ABSCHNITTES:**

*Bekannte Ursachen:*

- Ein Abschnitt wurde vor einem **Beginn ...**-Schlüsselwort oder nach dem **Ende**-Schlüsselwort plaziert (↔ 4.2.2.2)

- Es ist mehr als ein Abschnitt derselben Art vorhanden

**216** DIESES SCHLÜSSELWORT IST NUR IN WES-DATEIEN ERLAUBT:

*Bekannte Ursachen:*

- Das Schlüsselwort **Akzeptierendes Symbol** wurde in einer Grammatik-Datei ( $\leftrightarrow$  4.2.2.2) verwendet, obwohl seine Benutzung auf WES-Dateien ( $\leftrightarrow$  4.2.3) eingeschränkt ist.

**217** UNGÜLTIGE POSITION DES ABSCHNITTES 'REGELN':

*Bekannte Ursachen:*

- Der **Regeln**-Abschnitt ( $\leftrightarrow$  4.2.2.3) wurde vor einem **Beginn ...**-Schlüsselwort oder nach dem **Ende**-Schlüsselwort plaziert
- Es ist mehr als ein **Regeln**-Abschnitt vorhanden

**218** SCHLÜSSELWORT 'BEGINN ...' ERWARTET:

*Bekannte Ursachen:*

- Das Schlüsselwort **Ende** wurde gelesen, ohne daß vorher ein **Beginn ...**-Schlüsselwort gelesen wurde ( $\leftrightarrow$  4.2.2.2)

**220** FEHLENDER 'REGELN'-ABSCHNITT:

*Bekannte Ursachen:*

- Das Schlüsselwort **Ende** wurde gelesen, ohne daß vorher ein **Regeln**-Abschnitt vorkam ( $\leftrightarrow$  4.2.2.2)

**221** FEHLENDES SCHLÜSSELWORTENDE:

*Bekannte Ursachen:*

- Ein begonnenes Schlüsselwort ( $\leftrightarrow$  4.2.2.2) wurde nicht ordnungsgemäß beendet

**230** UNERLAUBTES STEUERSYMBOL GEFUNDEN:

*Bekannte Ursachen:*

- An dieser Stelle ist das verwendete Steuersymbol ( $\leftrightarrow$  4.2.2.1) nicht erlaubt

**231** ZUWEISUNGSSYMBOL ERWARTET:

*Bekannte Ursachen:*

- Im Abschnitt **Alphabet** ( $\leftrightarrow$  4.2.2.2) wurde nach einem Alphabetsymbol in derselben Zeile ein weiteres gelesen, welches nicht dem Zuweisungssymbol ( $\leftrightarrow$  4.2.2.1) entspricht

**232** GANZZAHLIGER WERT AUF DER RECHTEN SEITE ERWARTET:

*Bekannte Ursachen:*

- Im Abschnitt **Alphabet** ( $\leftrightarrow$  4.2.2.2) wurde versucht, einem Alphabetsymbol einen ungültigen Wert zuzuweisen

**241** SCHLÜSSELWORT ERWARTET:

*Bekannte Ursachen:*

- Nach dem Lesen Schlüsselwort-Beginn-Symbols ( $\leftrightarrow$  4.2.2.1) folgte kein bekanntes Schlüsselwort
- 251** PFEILSYMBOL ERWARTET:  
*Bekannte Ursachen:*
- Im Abschnitt **Regeln** ( $\leftrightarrow$  4.2.2.2.3) wurde in einer Regeldefinition kein Pfeilsymbol ( $\leftrightarrow$  4.2.2.1) gefunden
- 255** LEERE REGEL (  $\rightarrow$  ) NICHT ERLAUBT:  
*Bekannte Ursachen:*
- Ein Pfeilsymbol ( $\leftrightarrow$  4.2.2.1) steht alleine in einer Zeile im **Regeln**-Abschnitt ( $\leftrightarrow$  4.2.2.2.3)
- 256** DAS AKZEPTIERENDE SYMBOL MUSS ALLEINE STEHEN:  
*Bekannte Ursachen:*
- Das akzeptierende Symbol ( $\leftrightarrow$  4.2.2.2.3) steht auf einer rechten Regelseite nicht alleine
- 257** NUR EINE REGEL MIT LEERER RECHTER SEITE ERLAUBT:  
*Bekannte Ursachen:*
- Es wurden mehrere Regeln der Form  $w \rightarrow$  gefunden
- 258** MEHRERE MÖGLICHKEITEN FÜR DAS AKZEPTIERENDE SYMBOL GEFUNDEN: :  
*Bekannte Ursachen:*
- Es kommen mehrere Symbole als akzeptierende Symbole in Frage ( $\leftrightarrow$  4.2.2.2.3)
- 259** ES KONNTE KEIN AKZEPTIERENDES SYMBOL ERMITTELT WERDEN:  
*Bekannte Ursachen:*
- Jedes Symbol kommt sowohl auf einer rechten als auch auf einer linken Regelseite vor ( $\leftrightarrow$  4.2.2.2.3)
- 261** UNBEKANNTES SYMBOL IM EINGABEWORT:  
*Bekannte Ursachen:*
- Dieser Fehler tritt beim Parsieren eines Eingabewortes auf, wenn es ein Symbol enthält, daß in dem aktuellen Wortsatzungssystem nicht vorkommt
- 280-299** INTERNER PARSER-FEHLER:  
*Bekannte Ursachen:*
- Diese Fehler *sollten* nicht auftreten. Tritt doch einer auf, so kontaktieren Sie mich bitte (Email: [kuhnigk@informatik.mu-luebeck.de](mailto:kuhnigk@informatik.mu-luebeck.de)) und schicken mir Ihre Grammatik.

## Literatur

- [BO93] Ronald V. Book and Friedrich Otto. *String-Rewriting Systems*. Texts and Monographs in Computer Science. Springer, 1993.
- [Bun96] Gerhard Buntrock. *Growing context-sensitive languages*. Habilitation thesis, Fakultät für Mathematik und Informatik, Universität Würzburg, July 1996. In German (*Wachsende kontextsensitive Sprachen*).
- [NO98] Gundula Niemann and Friedrich Otto. The Church-Rosser languages are the deterministic variants of the growing context-sensitive languages. In Maurice Nivat, editor, *Proceedings of the First Conference on Foundations of Software Science and Computation Structures (1<sup>st</sup> FoSSaCS)*, volume 1378 of *LNCS*, pages 243–257. Springer, 1998.
- [vEB79] Peter van Emde Boas. Complexity of linear problems. In *FCT2nd*, pages 117–120, 1979.