LR(k)-Analyse für Pragmatiker

Andreas Kunert

Version 2.218
(16. Juni 2017)

Humboldt-Universität zu Berlin
Institut für Informatik / ZE Rechenzentrum (CMS)

INHALTSVERZEICHNIS

1 Grundbegriffe 5
1.1 Grammatiken und Ableitbarkeit . . . 5
1.1.1 Definition Grammatik . . . 5
1.1.2 Kontextfreie Grammatiken und Ableitbarkeit . . . 6
1.1.3 LL(k)- und LR(k)-Sprachen . . . 6
1.1.4 Die Symbole ε und $ . . . 7
1.2 FIRST-Mengen . . . 8
1.2.1 Definition . . . 8
1.2.2 Algorithmus zur Berechnung von FIRST . . . 8
1.2.3 Beispiel . . . 9
1.3 FOLLOW-Mengen . . . 10
1.3.1 Definition . . . 10
1.3.2 Algorithmus zur Berechnung von FOLLOW . . . 11
1.3.3 Beispiel . . . 11

2 Bottom-Up-Syntaxanalyse 13
2.1 Aufbau einer Syntaxanalysetabelle . . . 13
2.2 Ablauf der Syntaxanalyse . . . 14
2.3 Beispiel . . . 15

3 LR(0)-Syntaxanalyse 17
3.1 Prinzipieller Ablauf und Grundlagen . . . 17
3.2 LR(0)-Elemente . . . 18
3.3 Der Hüllenoperator CLOSURE_0 . . . 19
3.3.1 Algorithmus zur Berechnung . . . 19
3.3.2 Beispiel . . . 19
3.4 Die Sprungoperation GOTO_0 . . . 20
3.4.1 Algorithmus zur Berechnung . . . 20

3.4.2 Beispiel 21
3.5 Das Aufstellen des Zustandsübergangsgraphen 21
3.5.1 Theorie 21
3.5.2 Beispiel 22
3.6 Das Erstellen der Syntaxanalysetabelle 22
3.6.1 Algorithmus 22
3.6.2 Beispiel 24

4 SLR(1)-Syntaxanalyse 27
4.1 Idee und Anwendung 27
4.2 Beispiel 28

5 LR(1)-Syntaxanalyse 31
5.1 Idee 31
5.2 LR(1)-Elemente 32
5.3 Der Hüllenoperator $\mathsf{CLOSURE}_1$ 32
5.3.1 Algorithmus zur Berechnung 32
5.3.2 Beispiel 33
5.4 Die Sprungoperation GOTO_1 34
5.5 Das Aufstellen des Zustandsübergangsgraphen 35
5.6 Das Erstellen der Syntaxanalysetabelle 36

6 LALR(1)-Syntaxanalyse 37
6.1 Idee 37
6.2 Beispiel 38

7 LR(k)-Syntaxanalyse für $k \geq 2$ 41
7.1 Motivation 41
7.2 FIRST_k und FOLLOW_k 42
7.2.1 Die Hilfsoperationen k-Präfix und k-Konkatenation 42
7.2.2 FIRST_k 43
7.2.3 FOLLOW_k 43
7.3 LR(k)-Elemente, $\mathsf{CLOSURE}_k$ und GOTO_k 44
7.4 LR(k)-Zustandsübergangsgraphen und LR(k)-Syntaxanalysetabellen 44
7.5 LR(k)-Syntaxanalyse 45
7.6 Beispiel 46

A Aufgaben 49
A.1 Einfache Grammatiken 50
A.2 Komplexere Grammatiken 56
A.3 Real existierende Grammatiken 60

B Literaturverzeichnis 63

VORWORT

Die LR(k)-Analyse ist innerhalb der Informatik mit zwei wichtigen Erkenntnissen verbunden:

1. Sie stellt die heute gebräuchlichste Form der Syntaxanalyse dar.
   Dies ist zum einen dadurch begründet, dass die LR(k)-Analyse das mächtigste Verfahren unter den effizienten Syntaxanalysealgorithmen darstellt und zum anderen, dass es schon seit längerer Zeit Parsergeneratoren gibt, die einem die fehlerträchtige Tabellenkonstruktion abnehmen.

2. Man kann als Informatikstudent Nächte damit verbringen.

Der vorliegende Text soll helfen, den Einstieg in die Materie etwas zu vereinfachen, indem er zuerst die notwendigen Grundbegriffe erklärt und anschließend die einzelnen Analyseverfahren (in der Theorie und am Beispiel) beschreibt.

Dabei kann (und soll) dieser Text kein Compilerbaubuch ersetzen. Er sollte vielmehr als Ergänzung angesehen werden; genau die Stellen erklären, die im Buch unverständlich waren (und vielleicht genau an den Stellen Lücken aufweisen, wo das Buch eine gute Beschreibung liefert. . .).

Was dem Leser vielleicht relativ früh in diesem Text auffallen wird, ist die vergleichsweise lockere Sprache. Diese ist zum einen begründet durch die Zielgruppe (Studenten im dritten Semester) und zum anderen, dass es sich um einen Versuch meinerseits handelt (vielleicht der letzte), ein relativ trockenes Thema halbwegs nett lesbar zu beschreiben („unterhaltsam“ wäre etwas zu hoch gegriffen).

Wie auch immer, es handelt sich um meinen ersten Versuch eines derartigen Pamphlets, so dass ich über Kommentare, Kritiken und vor allem Korrekturen und Verbesserungsvorschläge *jeder* Art dankbar bin. Dabei spielt es bei den letztgenannten Punkten keine Rolle, ob es sich um inhaltliche, grammatikalische oder Ausdrucksfehler handelt.

Die jeweils letzte Version dieses Texts sollte irgendwo auf meiner Homepage unter `www.informatik.hu-berlin.de/~kunert`[1] zu finden sein. Ich bitte darum, immer erst dort nachzusehen, ob der Fehler nicht eventuell bereits behoben wurde. Falls nicht, bitte einfach eine Mail an mich: `kunert@informatik.hu-berlin.de`

Als Abschluss würde ich mich gerne noch bei den Probelesern bedanken und zwar bei Dr. Klaus Ahrens, Prof. Klaus Bothe sowie den beiden angehenden Diplom-Informatikern Glenn Schütze und Konrad Voigt.

Andreas Kunert
Berlin, 2003-06-02

Vorwort zur zweiten Version

Es war der 17. Januar 2003, an dem zum ersten Mal eine Urversion des vorliegenden Skriptes kompiliert wurde. Es war wohl am frühen Nachmittag; eine genaue Zeit kann ich erst ab der 13. Kompilation geben (13:27:27 Uhr CET), da erst dann die von mir im makefile implementierte automatische Dokumentation der Kompilationshistorie funktionierte.

Dabei hatte ich ursprünglich gar nicht vor, ein Lehr-Skript für Studenten zu schreiben, sondern wollte lediglich eine Notizensammlung anfertigen, mittels derer ich einmal jährlich (und zwar genau dann, wenn Compilerbau-Praktika anstehen) innerhalb kurzer Zeit wieder in den LR-Stoff „reinkomme“ und mir nicht jedes Mal erneut alles aus etlichen Quellen zusammensuchen muss. Was mich damals bewogen hat, dann doch mehr zu schreiben, kann ich heute leider nicht mehr sagen...

Wie dem auch sei, am 11. August 2003 (13:35:42 CEST) war das Skript dann soweit gediehen, dass ich mich traute, ihm die Versionsnummer „1“ zu geben (erstes *Major Release*?). Im selben Zeitraum erfolgte auch die erste Veröffentlichung auf meiner Homepage.

Seitdem ist eine Menge Zeit vergangen. Insbesondere in den ersten Jahren gab es zahlreiche Fehlerhinweise, was nicht unwesentlich mit meiner Zusage verbunden war, alle „erfolgreichen“ Fehlerfinder in zukünftigen Versionen dieses Skriptes namentlich zu verewigen.[2] Die Zahl der gemeldeten Korrekturwünsche pro Jahr ist in der folgenden Zeit dann zwar deutlich zurückgegangen, insbesondere nachdem ich keine Compilerbau-Praktika mehr gehalten habe, hat aber nie den Wert null erreicht.

Was demgegenüber deutlich zugenommen und mich nachvollziehbar sehr erfreut hat, ist die Menge von Danksagungsmails aus den mitunter entlegensten Ecken des deutschsprachigen Raumes. Diese brachten mich dazu, mal meinem Skript „hinterherzugooglen“[3] mit folgenden, zumindest für mich überraschenden Ergebnissen (Achtung, es folgt eine schamlose Selbstbeweihräucherung!):

[1] Kleines Update: inzwischen ist die URL kürzer: `www.hu-berlin.de/~kunert`
[2] Eine Zusage, die übrigens weiterhin gilt.
[3] ein Wort, dass ich in der ersten Version dieses Skriptes sicherlich noch nicht benutzt hätte

- das Skript wird offiziell in Compiler-Lehrveranstaltungen von vier Universitäten[4] und einem Gymnasium eingesetzt,
- das Skript wird in mehreren Foren-Einträgen empfohlen und Seminararbeiten referenziert,
- Wikipedia kennt mein Skript[5]

Inzwischen arbeite ich als Datenbank- und System-Administrator am Rechenzentrum der Humboldt-Universität. Da ich allerdings das Lehren doch stark vermisst habe und mein Büro überdies nur ca. 200 m vom Institut für Informatik entfernt ist, halte ich seit diesem Semester auf freiwilliger Basis wieder Compilerbau-Praktika.

Sowohl die Dankesmails, die Resonanz im Web als auch meine erneute Aufnahme der Lehrtätigkeit haben zu der vorliegenden Überarbeitung geführt, wobei der Großteil der Änderungen kosmetischer Natur (Wechsel der Rechtschreibung, interne Links, netteres Layout, . . .) sind bzw. sich im Hintergrund (Aufbau der LaTeX-Dateien, Versionsverwaltung, Einbindung in den umgezogenen Web-Auftritt, . . .) abgespielt haben. Die Überarbeitung ist auch noch nicht wirklich abgeschlossen (wie ich mich kenne, wird sie es auch nie), hat aber zumindest einen Stand erreicht, der meiner bescheidenen Meinung nach eine neue Versionsnummer rechtfertigt.

Zum Abschluss möchte ich mich bei allen bedanken, die mir freundliche Mails geschickt haben (ich beantworte definitiv jede Skript-bezogene Mail, es kann nur manchmal leider etwas länger dauern[6]) sowie bei allen, die mich dazu bewogen haben, das Skript erneut zu überarbeiten.

Andreas Kunert
Berlin, 2011-05-27

[4] die Humboldt-Universität nicht mitgezählt
[5] . . . und ich lege Wert darauf, dass die dortigen Links *nicht* von mir stammen (auch wenn ich diese bei einem späteren Umzug meiner Homepage korrigiert habe)
[6] mein bisheriger Negativrekord liegt bei drei Jahren

KAPITEL 1

GRUNDBEGRIFFE

In diesem Kapitel werden einige Grundbegriffe geklärt, die im späteren Verlauf benötigt werden.

1.1 Grammatiken und Ableitbarkeit

Ich werde hier nicht sonderlich tief auf Grammatiken und deren Einteilung eingehen, da mit ziemlicher Sicherheit jeder, der sich diesen Text zu Gemüte führt, etwas damit anfangen kann. Es geht mir in diesem Kapitel primär darum, klarzustellen, welche Symbole ich standardmäßig für die einzelnen Elemente einer Grammatik benutzen werde. Dabei halte ich mich an die Notation, die im Buch von Uwe Schöning [Sch97] verwendet wird.

1.1.1 Definition Grammatik

Eine (Typ-0-)Grammatik ist ein 4-Tupel $G = (V, \Sigma, P, S)$, wobei die einzelnen Elemente folgende Bedeutung haben:

- V die Menge der *Variablen* (auch *Metasymbole* oder *Nichtterminale* genannt – ich persönlich präferiere die Bezeichnung *Metasymbole*)
- Σ die Menge der *Terminalsymbole* ($\Sigma \cap V = \emptyset$)
- P die Menge der *Produktionen* ($P \subseteq (V \cup \Sigma)^+ \times (V \cup \Sigma)^*$)
- S das *Startsymbol* ($S \in V$)

Für die Elemente aus P hat sich statt der sonst üblichen Paarschreibweise (α, β) die intuitivere Notation $\alpha \to \beta$ eingebürgert.[1]

[1] Im weiteren Verlauf dieses Skriptes wird zwar die intuitive Notation benutzt, aber zur besseren Unterscheidung vom restlichen Text trotzdem ein Klammerpaar gesetzt, d. h. das genannte Beispiel hat in diesem Skript die folgende Form: $(\alpha \to \beta)$.

1.1.2 Kontextfreie Grammatiken und Ableitbarkeit

Die Grammatikdefinition im vorherigen Abschnitt erschlägt zwar wirklich alle Grammatiken, ist für uns aber eher unpraktisch. Der Grund dafür liegt darin, dass man bei vielen Grammatiken (Typ-0) nicht entscheiden kann, ob ein gegebenes Wort Element der durch die Grammatik beschriebenen Sprache ist, oder diese Entscheidung zwar möglich, aber für praktische Anwendungen zu komplex ist (Typ-1/kontextsensitiv)[2]. Dummerweise ist es aber leider genau dieses sogenannte Wortproblem, das wir bei der Syntaxanalyse lösen müssen.

Für uns Compilerbauer beginnen die Grammatiken ab Typ-2/kontextfrei interessant zu werden. Eine kontextfreie Grammatik ist eine Grammatik, für deren Produktionen gilt, dass sie auf der linken Seite immer aus genau einem Metasymbol bestehen.

Für alle Produktionen $(\alpha \to \beta)$ gilt also $\alpha \in V$, so dass unsere Regelmenge P folgende Gestalt annimmt:

$$P \subseteq V \times (V \cup \Sigma)^*$$

Für die später folgenden Definitionen benötigen wir noch eine wichtige Relation, und zwar die Ableitung ($\Rightarrow$). Es gilt (mit $x, y, \gamma \in (V \cup \Sigma)^*$, $A \in V$):

$$x \Rightarrow y \;:\Longleftrightarrow\; \exists_{\alpha,\beta,\gamma,A} : \; x = \alpha A \beta \;\wedge\; y = \alpha\gamma\beta \;\wedge\; (A, \gamma) \in P$$

Diese Definition beschreibt die Ableitung einer Zeichenkette aus einer anderen, indem einfach ein Metasymbol mittels einer dem Metasymbol zugehörigen Regel durch die rechte Seite eben dieser Regel substituiert wird.

Die reflexiv-transitive Hülle ($\Rightarrow^*$) wird intuitiv gebildet ($x, y, w_i \in (V \cup \Sigma)^*$):

$$x \Rightarrow^* y \;:\Longleftrightarrow\; (x = y) \vee (x \Rightarrow y) \vee (\exists_{w_1 \ldots w_n} : x \Rightarrow w_1 \Rightarrow \ldots \Rightarrow w_n \Rightarrow y)$$

Eine Zeichenkette y ist dann mittels der Hülle aus einer Zeichenkette x ableitbar, wenn die beiden entweder identisch sind oder aber sich y über einen oder mehrere Zwischenschritte aus x ableiten lässt.

Mit Hilfe der obigen Definitionen können wir nun die Sprache L zu einer gegebenen Grammatik G definieren. Wie zu erwarten besteht diese aus allen Wörtern, die wir aus dem Startsymbol (direkt oder indirekt) ableiten können:

$$L(G) := \left\{ x \in \Sigma^* \middle| S \Rightarrow^* x \right\}$$

1.1.3 LL(k)- und LR(k)-Sprachen

Die Sprachen, die man durch kontextfreie Grammatiken beschreiben kann, nennt man naheliegenderweise kontextfreie Sprachen. Diese Sprachklasse ist für Compilerbauer deshalb so interessant, da man in ihr das Wortproblem in kubischer (also polynomialer) Zeit lösen kann.[3]

[2] Für die Einteilung und Benennung der Grammatiken nach Noam Chomsky siehe [Sch97]
[3] Für genauere Details siehe Abschnitt 1.3.4 „Der CYK-Algorithmus“ in [Sch97].

Tragischerweise ist eine kubische Laufzeit für praktische Zwecke immer noch zu langsam, so dass wir uns im Compilerbau mit noch eingeschränkteren Sprach- bzw. Grammatikklassen auseinandersetzen. Die prominentesten Vertreter sind die sogenannten LL(k)- und die LR(k)-Sprachen. In beiden lässt sich das Wortproblem in linearer Zeit lösen.

Eine der ersten Fragen, die dieses Schriftstück beantworten möchte und die gleichzeitig gerne in Compilerbauprüfungen auftaucht, lautet: „Wofür stehen die einzelnen Buchstaben in den Bezeichnungen LL(k) und LR(k)?“.

Der erste Buchstabe (also in beiden Fällen ein „L“) steht für die Leserichtung bei der Syntaxanalyse – es wird in beiden Fällen von *l*inks nach rechts gelesen. Der zweite Buchstabe gibt an, ob bei der Syntaxanalyse eine *L*inks- oder eine *R*echtsableitung entsteht, d. h. ob in jedem Ableitungsschritt immer das am weitesten links oder rechts stehende Metasymbol substituiert wird.

Das k steht für den sogenannten LOOKAHEAD. Dieser gibt an, wieviele Zeichen man in der Eingabe „vorausschauen“ muss, um in jedem Ableitungschritt eine deterministische Entscheidung treffen zu können, mittels welcher Substitution man fortfährt.

Auf LL(k)-Sprachen und -Grammatiken soll an dieser Stelle nicht näher eingegangen werden, da dies den Rahmen eines LR(k)-Skripts ein wenig sprengen würde. Die verschiedenen Klassen von LR(k)-Grammatiken werden in den Kapiteln ab Kapitel 3 ausführlich beschrieben.

Eine wichtige Teilmengenbeziehung von Sprachklassen soll an dieser Stelle aber noch erwähnt werden, da sie zum Grundwissen gehört:

$$\text{LL}(k) \;\subset\; \text{LR}(k) \;=\; (\text{det. kontextfreie Sprachen}) \;\subset\; (\text{kontextfreie Sprachen})$$

1.1.4 Die Symbole ε und \$

Neben den bereits beschriebenen Definitionen benötigen wir zusätzlich noch die zweier spezieller Symbole. Zunächst wäre da das ε, das für das leere Wort (ein Wort bestehend aus null Terminalsymbolen) steht.

Das ε tritt vor allem in Grammatikproduktionen der Form $(X \to \varepsilon)$ auf, was gleichbedeutend ist mit: „X kann in einem Ableitungsschritt mittels dieser Regel entfernt werden“. Metasymbole, die sich mittels einer oder mehrerer Grammatikregeln auf das ε reduzieren lassen, werden als „ε-ableitbar“ bezeichnet.

Das \$ hingegen ist ein Terminalsymbol, das das Ende einer Eingabe bezeichnet, d. h. es ist das letzte Zeichen in *allen* Wörtern einer Sprache. In der Compilerbauliteratur gibt es verschiedene Ansichten, ob das \$ explizit in den Grammatikregeln notiert werden sollte, oder nicht. In diesem Skript werde ich das \$ implizit als Ende der Eingabe betrachten und es nicht extra in der Grammatik auftauchen lassen.

1.2 FIRST-Mengen

1.2.1 Definition

FIRST-Mengen werden eigentlich fast immer automatisch mit der LL(k)-Analyse in Verbindung gebracht (wo sie auch wirklich exzessiv genutzt werden), aber sie spielen auch bei der LR(k)-Analyse eine wichtige Rolle.

Genaugenommen betrachten wir in diesem Kapitel nur einen Spezialfall von FIRST-Mengen und zwar die FIRST_1-Mengen (statt FIRST-Mengen müsste man eigentlich FIRST_k-Mengen schreiben). Da man in der Praxis jedoch fast ausschließlich mit FIRST_1-Mengen konfrontiert wird, ist es in der Compilerbauliteratur üblich, statt FIRST_1 einfach FIRST zu schreiben und den allgemeinen Fall FIRST_k völlig zu ignorieren.[4] Dieser Tradition möchte dieses Skript zur Hälfte folgen: Auch hier wird im folgenden die vereinfachte Notation FIRST statt FIRST_1 benutzt, aber die FIRST_k-Mengen werden trotzdem noch behandelt und zwar in Kapitel 7.

FIRST-Mengen werden von beliebigen Zeichenketten, bestehend aus Terminal- und Metasymbolen, gebildet. Die FIRST-Menge einer solchen Zeichenkette gibt an, welche Terminalsymbole als jeweils erstes Symbol in einer von der Zeichenkette abgeleiteten Zeichenkette auftreten können (wobei alle möglichen Ableitungen berücksichtigt werden).

Diese Formulierung ist natürlich schrecklich informal und bringt nur demjenigen etwas, der bereits eine Vorstellung von einer FIRST-Menge hat, daher folgt eine schöne, formale, zweistufige Definition:[5]

$$
\begin{aligned}
\text{FIRST}^0 &: (V \cup \Sigma)^+ \to \mathcal{P}(\Sigma) \\
\text{FIRST}^0(x) &:= \left\{ y \,\middle|\, \exists_{\alpha \in (\Sigma \cup V)^*} : x \Rightarrow^* y\alpha,\ y \in \Sigma \right\} \\
\text{FIRST} &: (V \cup \Sigma)^+ \to \mathcal{P}(\Sigma \cup \{\varepsilon\}) \\
\text{FIRST}(x) &:= \begin{cases} \text{FIRST}^0(x) \cup \{\varepsilon\} & \text{, wenn } x \Rightarrow^* \varepsilon \\ \text{FIRST}^0(x) & \text{, sonst} \end{cases}
\end{aligned}
$$

Die Definition von FIRST ist zwar relativ leicht nachvollziehbar, kann jedoch nur bedingt zur konstruktiven Berechnung der FIRST-Mengen verwendet werden. Der Grund besteht darin, dass bei rekursiver Regelanwendung unendlich viele Ableitungen in die Definition einer FIRST-Menge eingehen können.

1.2.2 Algorithmus zur Berechnung von FIRST

Der folgende, dem Drachenbuch [ASU99] entnommene, Algorithmus zur FIRST-Mengenberechnung scheint da schon wesentlich geeigneter zu sein.

Zuerst werden für alle Meta- und Terminalsymbole die zugehörigen FIRST-Mengen berechnet, indem die folgenden drei Schritte solange iterativ angewendet werden, bis sich keine FIRST-Menge mehr ändert:

[4] Es gibt natürlich Ausnahmen, z. B. [Spe03] und [Wil97]

[5] Zur Notation: $\mathcal{P}$ steht für die Potenzmenge ($\mathcal{P}(X) := \{Y | Y \subseteq X\}$)

1. wenn $X \in \Sigma$ (also X ein Terminal), so ist FIRST$(X) = \{X\}$
2. gibt es eine Produktion der Form $(X \to \varepsilon)$, so füge ε zu FIRST(X) hinzu
3. gibt es eine Produktion der Form $(X \to Y_1 Y_2 \ldots Y_n)$, so füge zunächst FIRST$(Y_1) \setminus \{\varepsilon\}$ zu FIRST(X) hinzu. Sollte gelten $\varepsilon \in$ FIRST(Y_1), d. h. Y_1 lässt sich zu ε reduzieren, so füge auch FIRST$(Y_2) \setminus \{\varepsilon\}$ hinzu. Sollte wiederum Y_2 ε-ableitbar sein, fahre wie gehabt fort, bis du auf ein Y_i stößt, das nicht ε-ableitbar ist. Sollten alle Y_i ε-ableitbar sein, dann (und *nur* dann) füge auch ε zu FIRST (X) hinzu.

Anschließend kann man ohne weitere Probleme die FIRST-Menge jeder beliebigen Zeichenkette $X_1 X_2 \ldots X_n$ berechnen. Dazu nimmt man analog zum soeben praktizierten dritten Schritt erst FIRST$(X_1) \setminus \{\varepsilon\}$ zu FIRST $(X_1 X_2 \ldots X_n)$ hinzu. Sollte X_1 ε-ableitbar sein, nimm auch FIRST$(X_2) \setminus \{\varepsilon\}$ zu FIRST $(X_1 X_2 \ldots X_n)$ hinzu, usw. Sollten alle X_i ε-ableitbar sein, dann (und *nur* dann) füge auch ε zu FIRST $(X_1 X_2 \ldots X_n)$ hinzu.

1.2.3 Beispiel

Zum Abschluss des Abschnitts soll ein Beispiel die Anwendung des Algorithmus verdeutlichen. Gegeben sei die folgende Grammatik G_1 mit dem Startsymbol Z:

$$\begin{array}{llcl} (1) & Z & \to & S \\ (2) & S & \to & Sb \\ (3) & S & \to & bAa \\ (4) & A & \to & aSc \\ (5) & A & \to & a \\ (6) & A & \to & aSb \end{array}$$

Wir wollen nun die FIRST-Mengen der einzelnen Grammatiksymbole berechnen (wobei die Berechnung der FIRST-Mengen der Terminalsymbole aufgrund der ersten Regel im Algorithmus zugegebenermaßen etwas witzlos ist).

Fangen wir mit FIRST(A) an; die vierte Produktion liefert uns $(A \to aSc)$, worauf wir die dritte Regel anwenden können:

$$\underbrace{A}_{X} \to \underbrace{a}_{Y_1}\underbrace{S}_{Y_2}\underbrace{c}_{Y_3}$$

Wir müssen also FIRST$(Y_1) =$ FIRST(a) zu FIRST(A) hinzunehmen. Dank der ersten Regel wissen wir, dass FIRST$(a) = \{a\}$. Da a nicht ε-ableitbar ist (oder anders: die FIRST-Menge von a kein ε enthält), müssen wir Y_2 und Y_3 (also S und c) nicht weiter betrachten.

Die dritte Regel auf die Produktionen 5 und 6 angewendet, liefert uns keine weiteren Informationen (es wird stets a zu FIRST(A) hinzugefügt). Wendet man die

dritte Regel auf die dritte Produktion ($S \to bAa$) an, so erfährt man, dass man b zu $\mathsf{FIRST}(S)$ hinzunehmen soll.

Die zweite Produktion ist relativ nett: wir sollen $\mathsf{FIRST}(S)$ zu $\mathsf{FIRST}(S)$ hinzunehmen – dieser Schritt kann ignoriert werden. Zu guter Letzt wenden wir uns noch der ersten Produktion zu. Diese sorgt dafür, dass $\mathsf{FIRST}(S)$ zu $\mathsf{FIRST}(Z)$ hinzugenommen wird.

Nun gehen wir noch einmal im Kopf alle Regeln des Algorithmus und alle Produktionen der Grammatik durch und stellen fest, dass bei keiner erneuten Anwendung einer Regel auf eine Produktion eine Änderung an der bereits berechneten FIRST-Menge erfolgt.

Fasst man alle Berechnungen zusammen, so erhält man abschließend folgende Tabelle:

X	$\mathsf{FIRST}(X)$
Z	$\{b\}$
S	$\{b\}$
A	$\{a\}$

1.3 FOLLOW-Mengen

1.3.1 Definition

Neben den FIRST-Mengen haben auch die FOLLOW-Mengen im Compilerbau eine große Bedeutung. Auch sie werden exzessiv bei der LL(k)-Analyse benutzt und spielen dort insbesondere bei der Fehlerstabilisierung eine wichtige Rolle. Bei der LR(k)-Analyse werden sie vor allem beim SLR(1)-Verfahren benötigt.

Analog zu den FIRST-Mengen spielen auch bei den FOLLOW-Mengen (eigentlich FOLLOW_k-Mengen) die FOLLOW_1-Mengen eine so große Bedeutung, dass wir statt FOLLOW_1 einfach nur FOLLOW schreiben und uns mit den FOLLOW_k-Mengen erst in Kapitel 7 beschäftigen.

Die FOLLOW-Menge eines Metasymbols (FOLLOW-Mengen können nur von Metasymbolen gebildet werden) enthält alle Terminalsymbole, die in irgendeinem Ableitungsschritt unmittelbar rechts von diesem Metasymbol stehen können. Etwas formaler drückt es die folgende Definition aus:

$$\begin{aligned}
\mathsf{FOLLOW}^0 &: V \to \mathcal{P}(\Sigma) \\
\mathsf{FOLLOW}^0(X) &:= \left\{ y \,\middle|\, \exists_{\alpha,\beta \in (\Sigma \cup V)^*} : S \Rightarrow^* \alpha X y \beta,\ y \in \Sigma \right\} \\
\mathsf{FOLLOW} &: V \to \mathcal{P}\left(\Sigma \cup \{\$\}\right) \\
\mathsf{FOLLOW}(X) &:= \begin{cases} \mathsf{FOLLOW}^0(X) \cup \{\$\} & \text{, wenn } S \Rightarrow^* \alpha X \\ \mathsf{FOLLOW}^0(X) & \text{, sonst} \end{cases}
\end{aligned}$$

Anmerkung: Die Definition ist deshalb zweistufig (mit dem Umweg über FOLLOW^0) aufgebaut, da wir $ als nicht explizit in der Grammatik vorkommend definiert haben. In Compilerbaubüchern, die das $ bereits in den Grammatikregeln auftauchen lassen, wird dieser Umweg ausgelassen.

1.3.2 Algorithmus zur Berechnung von FOLLOW

Der folgende Algorithmus zur gleichzeitigen Ermittlung der FOLLOW-Mengen aller Metasymbole wurde ebenfalls dem Drachenbuch [ASU99] entnommen und besteht aus drei Schritten, die solange iterativ angewendet werden, bis sich keine der FOLLOW-Mengen mehr ändert:

1. nimm \$ zu FOLLOW(S) hinzu (S sei das Startsymbol)
2. gibt es eine Produktion $(A \to \alpha B \beta)$, so nimm FIRST$(\beta) \setminus \{\varepsilon\}$ zu FOLLOW(B) hinzu
3. gibt es eine Produktion $(A \to \alpha B)$ oder $(A \to \alpha B \beta)$ mit $\beta \Rightarrow^* \varepsilon$, so füge FOLLOW$(A)$ zu FOLLOW(B) hinzu

Im Prinzip ist jede der drei Regeln leicht nachvollziehbar. Nichtsdestotrotz folgt noch ein Beispiel, um die Handhabung zu demonstrieren.

1.3.3 Beispiel

Gegeben sei wieder die Grammatik G_1:

$$\begin{array}{llcl}
\text{(1)} & Z & \to & S \\
\text{(2)} & S & \to & Sb \\
\text{(3)} & S & \to & bAa \\
\text{(4)} & A & \to & aSc \\
\text{(5)} & A & \to & a \\
\text{(6)} & A & \to & aSb
\end{array}$$

Wir wollen nun die FOLLOW-Mengen der drei Metasymbole berechnen. In der folgenden Abbildung, die die Berechnung darstellt, stehen jeweils über den Pfeilen zuerst die angewendeten Regeln und unmittelbar dahinter (in Klammern) die Produktionen, die Opfer der jeweiligen Regel wurden.

X	FOLLOW(X)
Z	$\{\}$
S	$\{\}$
A	$\{\}$

$\overset{1}{\Longrightarrow}$

X	FOLLOW(X)
Z	$\{\$\}$
S	$\{\}$
A	$\{\}$

$\overset{2(3)}{\Longrightarrow}$

X	FOLLOW(X)
Z	$\{\$\}$
S	$\{\}$
A	$\{a\}$

$\overset{2(2/6)}{\Longrightarrow}$

X	FOLLOW(X)
Z	$\{\$\}$
S	$\{b\}$
A	$\{a\}$

$\overset{2(4)}{\Longrightarrow}$

X	FOLLOW(X)
Z	$\{\$\}$
S	$\{b, c\}$
A	$\{a\}$

$\overset{3(1)}{\Longrightarrow}$

X	FOLLOW(X)
Z	$\{\$\}$
S	$\{b, c, \$\}$
A	$\{a\}$

Der letzte Kasten stellt das Ende der Berechnung dar. Es wurden alle Regeln auf allen passenden Produktionen ausgeführt und eine erneute Anwendung wird keine neuen Elemente mehr in eine der FOLLOW-Mengen bringen.

KAPITEL 2

BOTTOM-UP-SYNTAXANALYSE

In diesem Abschnitt soll es darum gehen, wie man mit Hilfe einer gegebenen *Syntaxanalysetabelle* (in manchen Compilerbaubüchern auch *Parsertabellen* genannt; engl.: *parse table*) überprüft, ob ein Wort Element einer durch eine Grammatik beschriebenen Sprache ist. Dazu simuliert man einen Kellerautomaten, d. h. einen Zustandsautomaten mit „angeschlossenem" Kellerspeicher (auf gut deutsch: Stack), der sich in einem bestimmten Zustand befindet und, abhängig von diesem Zustand, dem obersten Symbol des Kellerspeichers und dem nächsten Eingabesymbol, eine Aktion ausführt und anschließend den Zustand wechselt, wobei die Aktion eine Schreiboperation auf dem Kellerspeicher zur Folge haben kann.

2.1 Aufbau einer Syntaxanalysetabelle

Syntaxanalysetabellen bestehen aus zwei Teilen: der Aktionstabelle und der Sprungtabelle. In beiden werden die Zustände des Automaten in den Zeilenköpfen aufgelistet. In den Spaltenköpfen befinden sich bei der Aktionstabelle die Terminalsymbole und in der Sprungtabelle die Metasymbole. Da sich die Spaltenköpfe beider Tabellen nicht überschneiden und die Zeilenköpfe identisch sind, werden die beiden Tabellen oft zu einer Syntaxanalysetabelle zusammengefasst.

Innerhalb der Aktionstabelle werden die Aktionen aufgelistet, die der Parser ausführen soll, wenn er sich in einem bestimmten Zustand (repräsentiert durch die Zeile) befindet und das nächste Eingabesymbol (repräsentiert durch die Spalte) liest. Diese Aktion kann eine der folgenden sein: Schieben (shift), Reduzieren (reduce), Akzeptieren (accept) oder Fehler (error).

In der Sprungtabelle werden nur Zustandsnummern eingetragen. Diese geben dem Parser an, in welchem Zustand er fortfahren soll, wenn er die Reduktion zu einem bestimmten Metasymbol durchgeführt hat.

Abbildung 2.1 auf Seite 15 zeigt eine Syntaxanalysetabelle für Grammatik G_1. Wir interessieren uns in diesem Kapitel noch nicht dafür, wie diese Tabelle zustande gekommen ist, sondern nehmen sie einfach mal als gegeben hin.

2.2 Ablauf der Syntaxanalyse

Mit Hilfe einer Syntaxanalysetabelle ein gegebenes Wort zu analysieren bedeutet einen Kellerautomaten zu simulieren. Auf dem Stack des Automaten werden sowohl Grammatiksymbole als auch Zustandsnummern gespeichert (genaugenommen kann man die Symbole auch weglassen – sie dienen jedoch sehr der Übersichtlichkeit).

Man beginnt mit einem fast leeren Stack, auf dem sich lediglich ein spezielles Symbol für den Kellerboden (meist # oder $) und die Nummer des Startzustandes befinden. Anschließend werden die folgenden Anweisungen iterativ ausgeführt:

1. Lesen des obersten Stackelementes (sollte eine Zustandsnummer sein)
2. Lesen des nächsten Zeichens in der Eingabe
3. Nachschlagen in der Aktionstabelle, welche Aktion dieser Kombination aus Zustand und Eingabezeichen zugeordnet ist
4. Ausführen der Aktion

Dabei bedeuten die verschiedenen möglichen Aktionen folgende Arbeitsschritte:

shift n
: Zuerst wird das nächste Zeichen in der Eingabe aus dieser entfernt und auf den Stack gelegt. Anschließend wird n als neuer Zustand auf den Stack gepackt.

reduce m
: Zunächst entfernt man doppelt soviele Elemente vom Stack, wie Symbole auf der rechten Seite von Produktion Nummer m stehen.[1] Anschließend liest man die Zustandsnummer z, die jetzt ganz oben auf dem Stack liegt. Nun wird das Metasymbol der linken Seite der Produktion auf den Stack gelegt und in der Sprungtabelle nachgeschlagen, welche Zustandsnummer diesem Metasymbol beim Zustand z zugeordnet ist. Zu guter Letzt wird auch diese neue Zustandsnummer auf den Stack gelegt.[2]

  Das Zeichen aus der Eingabe bleibt bei einer Reduktionsoperation unberührt und wird bei der folgenden Aktion erneut als Entscheidungskriterium benutzt.

[1] Doppelt soviele, weil wir ja pro Zustand auch noch ein Grammatiksymbol mit auf den Stack schreiben.

[2] Anmerkung: Die Zahl hinter dem r (z. B. 5 in r5) gibt wirklich *nur* die Nummer der anzuwendenden Produktion an. Es ist ein beliebter Anfängerfehler zu glauben, dahinter verberge sich eine Zustandsnummer (analog zur Schiebeoperation).

	Aktionstabelle				Sprungtabelle		
	a	b	c	$	A	S	Z
0		s3				1	
1		s2		acc			
2		r2		r2			
3	s6				4		
4	s5						
5		r3	r3	r3			
6	r5	s3				7	
7		s9	s8				
8	r4						
9	r6	r2	r2	r2			

Abbildung 2.1: Syntaxanalysetabelle für G_1

accept

Erreicht man diese Aktion, wurde das Eingabewort erfolgreich akzeptiert, und man ist fertig mit der Analyse.

error

Kommt man in die Verlegenheit, diese Aktion auszuführen, ist das Eingabewort nicht in der Sprache enthalten. Erroranweisungen befinden sich in Aktionstabellen in allen Zellen, in denen keine der anderen Aktionen steht (man lässt das explizite Hinschreiben von „error“ zugunsten der Übersichtlichkeit weg).

2.3 Beispiel

Wir wollen nun versuchen, ein Wort der zur Grammatik G_1 gehörenden Sprache zu analysieren. Die Syntaxanalysetabelle ist in Abbildung 2.1 zu sehen. Das zu analysierende Wort sei $baab$.

Zunächst befinden sich auf dem Stack der Kellerboden ($) und die Anfangszustandsnummer (0). In der Eingabe steht das Wort, gefolgt vom Eingabeendesymbol (ebenfalls das Zeichen $[3]):

Stack:	Eingabe:
$\$ \; 0$	$b\,a\,a\,b\,\$$

In der Aktionstabelle steht für den Zustand 0 und das Eingabezeichen b die Aktion s3, also ein Schieben des Eingabesymbols und des neuen Zustands 3 in den Stack:

Stack	Eingabe
$\$ \; 0 \; b \; 3$	$a\,a\,b\,\$$

[3]Im Normalfall macht die Unterscheidung, wann der Kellerboden und wann das Eingabeende gemeint ist, keine Probleme.

Für die Kombination aus Zustand 3 und Eingabezeichen a kann man in der Aktionstabelle die Aktion s6 ausfindig machen:

$\$\,0\,b\,3\,a\,6$	$a\,b\,\$$

Nun erfolgt die erste Reduktion (r5 – abgelesen in der Aktionstabelle bei Zustand 6 und Eingabezeichen a). Produktion 5 lautet: $(A \to a)$. Wir entfernen doppelt soviele Elemente aus dem Stack, wie die Produktion auf der rechten Seite Symbole hat ($2 \times 1 = 2$ Elemente). Anschließend legen wir das Metasymbol der linken Seite (A) auf den Stack und bilden aus den beiden (nun) obersten Elementen des Stacks (Zustand 3 und Symbol A) mit Hilfe der Sprungtabelle den nächsten Zustand (4), den wir ebenfalls auf den Stack legen. Das Ergebnis der Bemühungen ist folgendes:

$\$\,0\,b\,3\,A\,4$	$a\,b\,\$$

Wir fahren nun nach dem Schema immer weiter fort, bis wir auf die Aktion Akzeptieren (acc) stoßen. Die folgende Tabelle zeigt den gesamten Analyseablauf (also auch die bereits erläuterten drei Anfangsschritte). Die jeweils auszuführende Aktion steht in der dritten Spalte:

Stack	Eingabe	Aktion
$\$\,0$	$b\,a\,a\,b\,\$$	shift 3
$\$\,0\,b\,3$	$a\,a\,b\,\$$	shift 6
$\$\,0\,b\,3\,a\,6$	$a\,b\,\$$	reduce 5 $(A \to a)$
$\$\,0\,b\,3\,A\,4$	$a\,b\,\$$	shift 5
$\$\,0\,b\,3\,A\,4\,a\,5$	$b\,\$$	reduce 3 $(S \to bAa)$
$\$\,0\,S\,1$	$b\,\$$	shift 2
$\$\,0\,S\,1\,b\,2$	$\$$	reduce 2 $(S \to Sb)$
$\$\,0\,S\,1$	$\$$	accept

Das Analysieren eines Wortes mit Hilfe einer gegebenen Syntaxanalysetabelle ist also recht einfach. Etwas komplizierter ist es, die Tabelle selbst herzustellen – ein Thema, dem wir uns in den nächsten Kapiteln zuwenden werden.

KAPITEL 3

LR(0)-SYNTAXANALYSE

In den nun folgenden Kapiteln soll es darum gehen, die Syntaxanalysetabelle zu einer gegebenen Grammatik G mittels verschiedener Methoden selbst herzustellen. Das in diesem Kapitel verwendete Verfahren ist das einfachste, aber leider auch am wenigsten mächtige der hier vorgestellten Herangehensweisen.

3.1 Prinzipieller Ablauf und Grundlagen

Der Ablauf der Erstellung einer Syntaxanalysetabelle ist (unabhängig vom konkreten Verfahren) immer der folgende:

1. Hinzufügen einer neuen Startproduktion $(S' \to S)$ zu G, wobei S das alte Startsymbol und S' ein völlig neues Startsymbol ist.

   Hintergrund dieses Vorganges ist das Problem zu entscheiden, wann die Syntaxanalyse beendet ist. Während der Syntaxanalyse werden fortlaufend Produktionen der Grammatik in umgekehrter Richtung angewendet. Wenn man nun bei der Syntaxanalyse in einen Zustand gelangt, in dem mittels der (künstlichen) Startproduktion $(S' \to S)$ reduziert werden muss, weiß man, dass die gesamte Eingabe erfolgreich zum eigentlichen Startsymbol (S) reduziert wurde, d. h. das Wort in der Eingabe ist tatsächlich ein Element aus der Sprache der ursprünglichen Grammatik.

2. Bildung des Zustandsübergangsgraphen des zukünftigen Kellerautomaten (ein Beispiel eines solchen Graphen ist in Abbildung 3.1 auf Seite 23 zu sehen).

3. Aufstellen der Syntaxanalysetabelle mit Hilfe des Zustandsübergangsgraphen.

Punkt 1 kann als trivial betrachtet werden. Wesentlich interessanter sind die Punkte 2 und 3. Widmen wir uns zunächst der Erstellung des Zustandsübergangsgraphen.

3.2 LR(0)-Elemente

Bevor wir mit der Konstruktion beginnen, müssen einige Grundlagen geschaffen werden. Fangen wir mit der Definition von sogenannten LR(0)-Elementen an.

Sehr vereinfacht ausgedrückt ist ein LR(0)-Element nichts weiter als eine Grammatikproduktion, die irgendwo auf der rechten Seite einen Punkt enthält. Nehmen wir also z. B. die dritte Produktion aus G_1: $(S \to bAa)$, so können wir daraus die folgenden LR(0)-Elemente bilden:

$$\begin{array}{lcl} [S & \to & .bAa] \\ [S & \to & b.Aa] \\ [S & \to & bA.a] \\ [S & \to & bAa.] \end{array}$$

So weit, so gut, aber was soll der Punkt nun eigentlich bedeuten? Im Prinzip stellt ein LR(0)-Element einen Zustand während einer laufenden Syntaxanalyse dar. Alle Symbole links vom Punkt befinden sich in diesem Zustand bereits auf dem Stack, während alle Zeichen rechts vom Punkt noch nicht eingelesen wurden.

Befindet sich der Parser in einem durch ein LR(0)-Element charakterisierten Zustand, so können die folgenden drei disjunkten Fälle auftreten:

1. Der Punkt befindet sich ganz am rechten Ende des LR(0)-Elementes. Das bedeutet, die gesamte rechte Seite der zugrundeliegenden Regel befindet sich auf dem Stack; im nächsten Schritt kann also mittels dieser Regel reduziert werden (Reduce[1]).

2. Unmittelbar rechts vom Punkt befindet sich ein Terminalsymbol. In diesem Fall wird der Parser im folgenden Schritt prüfen, ob es sich beim nächsten Zeichen von der Eingabe um dieses Terminalsymbol handelt. Im Positivfall legt er es auf den Stack (Shift) und wechselt den Zustand. Für diesen Zustandswechsel definieren wir uns eine Sprungoperation. Im Negativfall wird eine Fehlermeldung generiert (Error).

3. Unmittelbar rechts vom Punkt befindet sich ein Metasymbol. In diesem Fall kommen wir auf direktem Weg nicht weiter (wir lesen schließlich keine Metasymbole von der Eingabe). Stattdessen müssen wir es schaffen, die nächsten Eingabezeichen zum benötigten Metasymbol zu reduzieren. Als Hilfestellung dafür (dieser Prozess kann selbstverständlich rekursiv über viele Ebenen gehen) definieren wir uns im folgenden einen Hüllenoperator.

Bevor wir mit der nächsten Hilfskonstruktion fortfahren soll noch kurz ein hinreichend oft vorkommender Spezialfall erwähnt werden: Zu jeder Grammatikregel mit einem ε auf der rechten Seite existiert genau ein LR(0)-Element und zwar $[X \to .]$, d. h. man lässt das ε-Zeichen weg.

[1] bzw. Accept, falls es sich bei der Regel um die künstlich hinzugefügte Startregel handelt

3.3 Der Hüllenoperator $\mathsf{CLOSURE}_0$

3.3.1 Algorithmus zur Berechnung

Beschäftigen wir uns zunächst mit dem Operator $\mathsf{CLOSURE}_0$ (Hülle). Dieser dient, wie oben im dritten Fall erwähnt, der Erkennung abgeleiteter Metasymbole während der Bearbeitung einer Regel. Die Bildung einer Hülle $\mathsf{CLOSURE}_0(I)$ zu einer Menge I von LR(0)-Elementen ist relativ einfach. Die Bildungsvorschrift besteht aus zwei Regeln:

1. füge I zu $\mathsf{CLOSURE}_0(I)$ hinzu (dieser Schritt dürfte keinen verwundern, der bei Hüllenoperatoren in Theoretischer Informatik I aufgepasst hat)

2. gibt es ein LR(0)-Element $[A \to \alpha.B\beta]$ aus $\mathsf{CLOSURE}_0(I)$ und eine Produktion $(B \to \gamma)$, so füge $[B \to .\gamma]$ zu $\mathsf{CLOSURE}_0(I)$ hinzu

Die Bildungsvorschrift anzuwenden, funktioniert also folgendermaßen: Zunächst fügt man alle Elemente aus I zu $\mathsf{CLOSURE}_0(I)$ hinzu. Anschließend überprüft man in allen Elementen aus $\mathsf{CLOSURE}_0(I)$ (also im ersten Schritt dasselbe wie I), ob sich Metasymbole unmittelbar rechts von einem Punkt befinden. Ist das der Fall, so fügt man alle Produktionen, die aus diesem Metasymbol ableiten, zu $\mathsf{CLOSURE}_0(I)$ hinzu, wobei der Punkt an den Anfang der rechten Seite gesetzt wird. Dabei ist zu beachten, dass es durchaus mehrere LR(0)-Elemente zu einer gleichen Produktion in $\mathsf{CLOSURE}_0(I)$ geben kann (also zwei Elemente, die sich nur in der Position des Punktes unterscheiden).

Hat man alle in Frage kommenden LR(0)-Elemente hinzugefügt, überprüft man diese neu hinzugekommenen Elemente auf Metasymbole unmittelbar rechts vom Punkt. Sollten neue Metasymbole hinzugekommen sein, so werden auch deren Produktionen als LR(0)-Elemente zu $\mathsf{CLOSURE}_0(I)$ hinzugefügt. Diese Prozedur wiederholt man solange, bis kein neues Metasymbol mehr unmittelbar rechts von einem Punkt in irgendeinem LR(0)-Element aus $\mathsf{CLOSURE}_0(I)$ existiert.

3.3.2 Beispiel

Nehmen wir einmal mehr Grammatik G_1 und berechnen die Hülle des aus der Startproduktion entstandenen LR(0)-Elementes $[Z \to .S]$.

$$
\begin{array}{llll}
(1) & Z & \to & S \\
(2) & S & \to & Sb \\
(3) & S & \to & bAa \\
(4) & A & \to & aSc \\
(5) & A & \to & a \\
(6) & A & \to & aSb
\end{array}
$$

Im ersten Schritt fügen wir alle Elemente aus $\{[Z \to .S]\}$ zur Hülle hinzu:

$$[Z \to .S]$$

Nun sehen wir nach, ob wir in einem der enthaltenen LR(0)-Elemente ein Metasymbol unmittelbar rechts vom Punkt finden. Wir finden das Metasymbol S, also fügen wir alle Produktionen der Grammatik, die S auf der linken Seite enthalten, zur Hülle hinzu, wobei wir den Punkt an den Anfang der rechten Seite setzen:

$[Z \to .S]$
$[S \to .Sb]$
$[S \to .bAa]$

Nun prüfen wir bei allen den neu hinzugekommenen Elementen, ob ein Metasymbol direkt rechts von einem Punkt steht. Dies trifft hier nur auf das S zu, dessen Produktionen wir aber bereits hinzugenommen haben. Da wir also keine neuen Metasymbole mehr finden, mit denen wir fortfahren müssten, sind wir fertig (und können den Kasten schließen):

$[Z \to .S]$
$[S \to .Sb]$
$[S \to .bAa]$

Wir haben also erfolgreich berechnet:

$$\mathsf{CLOSURE}_0\Big(\big\{[Z \to .S]\big\}\Big) = \big\{[Z \to .S], [S \to .Sb], [S \to .bAa]\big\}$$

3.4 Die Sprungoperation GOTO_0

3.4.1 Algorithmus zur Berechnung

Die zweite Hilfsoperation, die benötigt wird, ist $\mathsf{GOTO}_0(I, X)$. Diese erzeugt zu einem Zustand, der durch eine Menge I von LR(0)-Elementen charakterisiert wird, den Folgezustand unter der Annahme, dass das Symbol $X \in \Sigma \cup V$ erfolgreich gelesen bzw. hergeleitet wurde.

Die Bildung ist nicht sonderlich kompliziert: Man nimmt zuerst alle Elemente aus I und fügt davon diejenigen zu $\mathsf{GOTO}_0(I, X)$ hinzu, in denen sich X unmittelbar rechts vom Punkt befindet, wobei man den Punkt hinter das X verschiebt. Anschließend wird von der neuen Menge die Hülle gebildet, und schon ist man fertig.

Die semantische Bedeutung ist klar: Man hat ein benötigtes Zeichen X (rechts vom Punkt stehend) gelesen (bzw. im Falle eines Metasymbols hat man selbiges durch Reduktion erhalten) und wechselt jetzt in den Zustand, wo sich das Zeichen auf dem Stack (also links vom Punkt) befindet.

Man kann die gesamte Definition selbstverständlich auch etwas formaler ausdrücken:

$$\mathsf{GOTO}_0(I, X) = \mathsf{CLOSURE}_0\left(\left\{ [A \to \alpha X.\beta] \;\middle|\; [A \to \alpha.X\beta] \in I \right\}\right)$$

3.4.2 Beispiel

Nehmen wir noch einmal Grammatik G_1. Aufgabe sei nun, folgende Menge zu berechnen:

$$\mathsf{GOTO}_0\Big(\big\{[Z \to .S], [S \to .Sb], [S \to .bAa]\big\}, b\Big)$$

Zunächst suchen wir alle LR(0)-Elemente, die b unmittelbar rechts von einem Punkt stehen haben (genau eins: $[S \to .bAa]$) und fügen sie zur Menge hinzu, wobei wir den Punkt eine Position nach rechts (also hinter das b) verschieben:

$[S \to b.Aa]$

Der noch offene Kasten soll bereits andeuten, dass wir noch nicht fertig sind: Es fehlt noch die Hüllenbildung. Das einzige Metasymbol unmittelbar rechts von einem Punkt ist A, so dass wir also noch alle von A ableitenden Produktionen hinzunehmen müssen:

$[S \to b.Aa]$
$[A \to .aSc]$ $[A \to .a]$ $[A \to .aSb]$

Nun sind wir auch schon fertig, da keine LR(0)-Elemente hinzugekommen sind, die weitere Metasymbole unmittelbar rechts des Punktes stehen haben. Damit haben wir also erfolgreich berechnet:

$$\mathsf{GOTO}_0\Big(\big\{[Z \to .S], [S \to .Sb], [S \to .bAa]\big\}, b\Big)$$
$$= \big\{[S \to b.Aa], [A \to .aSc], [A \to .a], [A \to .aSb]\big\}$$

3.5 Das Aufstellen des Zustandsübergangsgraphen

3.5.1 Theorie

Hat man einmal die Bildung der Mengen GOTO_0 und $\mathsf{CLOSURE}_0$ verinnerlicht, so erweist sich das Aufstellen des Zustandsübergangsgraphen als relativ einfache Übung. Man beginnt mit der künstlich eingefügten Startproduktion $(S' \to S)$. Von dem zugehörigen LR(0)-Element $[S' \to .S]$ bildet man nun die Hülle, interpretiert die erhaltene Menge von LR(0)-Elementen als Zustand und gibt ihm eine Nummer (welche ist völlig egal – 0 oder 1 scheinen für den Startzustand sinnvoll).

Nun bildet man für jedes (Terminal- oder Meta-)Symbol X, das sich unmittelbar rechts vom Punkt in einem LR(0)-Element des Startzustandes befindet, die Menge GOTO_0. Die dabei entstehenden Mengen bekommen wieder jeweils eigene Nummern und werden durch Pfeile mit dem Startzustand verbunden, an die man das verwendete Symbol X schreibt.

Nun fährt man mit diesen Zuständen wie mit dem Startzustand fort (also GOTO_0-Mengenbildung und Pfeile malen). Sollte man beim Erstellen einer GOTO_0-Menge

feststellen, dass ein identischer Zustand bereits vorhanden ist, so erzeugt man keinen neuen, sondern verwendet stattdessen den bereits vorhandenen. Die gesamte Spielerei führt man solange fort, bis sich kein noch nicht betrachtetes Symbol mehr unmittelbar rechts von einem Punkt in irgendeinem LR(0)-Element irgendeines Zustandes befindet.

Hat man dies erreicht, sollte man einen vollständigen LR(0)-Zustandsübergangsgraphen vor sich haben.

3.5.2 Beispiel

Zum Beispiel ist nicht viel zu sagen, wenn Hüllen- und Sprungoperation bereits „sitzen". Nehmen wir die Grammatik G_1 und bilden davon den Zustandsübergangsgraphen.

Wir beginnen mit der Regel $(Z \to S)$.[2] Die Hülle des zur Startproduktion gehörenden LR(0)-Elementes $[Z \to .S]$ haben wir bereits in Abschnitt 3.3.2 ausgerechnet. Sie sieht folgendermaßen aus:

$[Z \to .S]$
$[S \to .Sb]$
$[S \to .bAa]$

Nun bildet man die Sprungoperationen dieser Menge mit allen Symbolen, die in irgendeinem LR(0)-Element dieser Menge unmittelbar rechts von einem Punkt auftauchen (hier also S und b). Anschließend werden Pfeile, die mit dem entsprechenden Symbol beschriftet werden, von dem alten Zustand zu den einzelnen neuen Zuständen gezogen. Diesen Vorgang wiederholt man nun mit den neuen Zuständen, bis es nichts mehr zu tun gibt.

In Abbildung 3.1 ist der vollständige Zustandsübergangsgraph zu sehen. Die Numerierung der einzelnen Zustände kann beliebig erfolgen, da sie auf die spätere Analyse keinen Einfluss hat.

3.6 Das Erstellen der Syntaxanalysetabelle

3.6.1 Algorithmus

Hat man den Zustandsübergangsgraphen erfolgreich aufgestellt, so kann man mit Hilfe dessen die Syntaxanalysetabelle erzeugen. Das Verfahren an sich ist nicht schwer. Zunächst erstellt man eine leere Tabelle, deren Spaltenköpfe erst die Terminalsymbole (für die Aktionstabelle) und dann die Metasymbole (für die Sprungtabelle) enthalten. In die Zeilenköpfe trägt man die Zustandsnummern ein.

Nun fängt man an, die Tabelle auszufüllen. Man beginnt in einem beliebigen Zustand (wobei es der Übersichtlichkeit halber sinnvoll ist, sie in der Reihenfolge durchzugehen, in der sie numeriert sind). Für jeden vom Zustand weggehenden

[2] Z ist in G_1 bereits von vornherein als künstliches Startsymbol S' eingebaut worden.

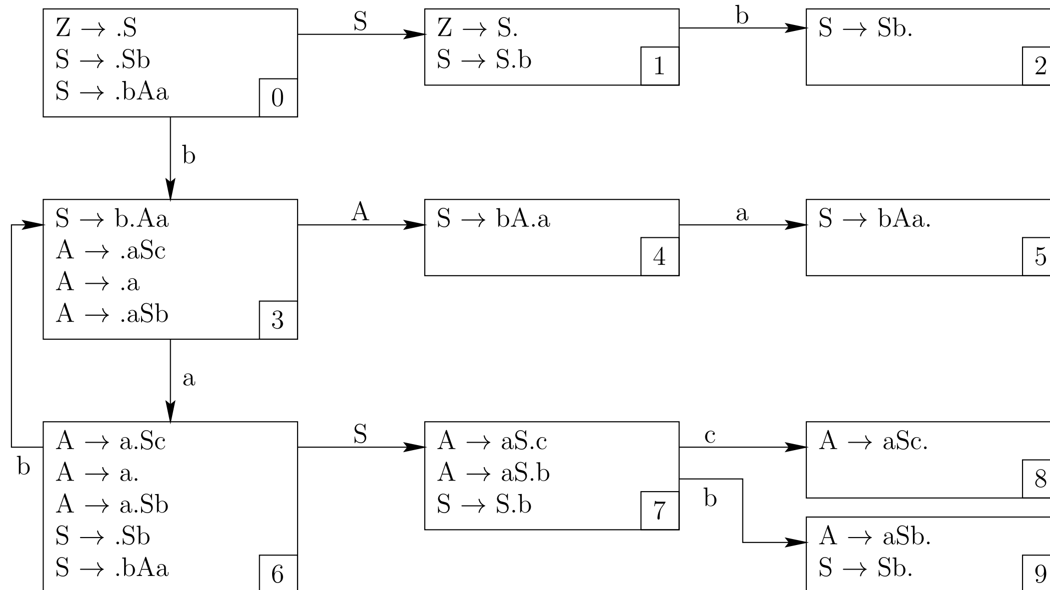

Abbildung 3.1: LR(0)-Zustandsübergangsgraph für Grammatik G_1

Pfeil, der mit einem Terminalsymbol beschriftet ist, erstellt man einen Eintrag in der zugehörigen Tabellenzelle (die sich innerhalb der Aktionstabelle befinden sollte). Dieser Eintrag erhält als Aktion „shift“ und die Nummer des Zustandes, auf den der Pfeil verweist. Für jeden Pfeil, der mit einem Metasymbol beschriftet ist, erstellt man analog einen Eintrag in der Sprungtabelle, nur dass man die Aktion weglässt, also nur die Nummer des Folgezustandes notiert.

Anschließend trägt man im Zustand, in dem die künstliche Startregel vollständig akzeptiert wird (also sich der Punkt am Ende der rechten Seite befindet ($[Z \to S.]$)), die Aktion *accept* beim Terminalsymbol $ ein. Wir hatten ja diese künstliche Produktion damit motiviert, dass wir bei ihrer Anwendung die Syntaxanalyse positiv beenden können.

Nun bleiben noch die „echten“ Reduktionen übrig: Für jedes LR(0)-Element (aus allen Zuständen), in dem sich der Punkt am Ende der rechten Seite befindet, trägt man für *alle* Terminalsymbole in der dem Zustand entsprechenden Zeile als Aktion „reduce“ und die Nummer der zum LR(0)-Element gehörenden Regel (die man aus der Definition der Grammatik erhält) ein.

Sollte man beim Eintragen einer Reduktionsaktion feststellen, dass in der zu beschreibenden Tabellenzelle bereits ein Eintrag vorhanden ist, so hat man einen Konflikt. Und zwar (je nachdem, ob sich in der Zelle eine Schiebe- oder Reduktionsanweisung befand) einen Schiebe/Reduktions-(shift-reduce bzw. s/r-) oder einen Reduktions/Reduktions-(reduce/reduce bzw. r/r-)konflikt. Sollte dieser Fall auftreten, so ist die gegebene Grammatik nicht vom Typ LR(0) und kann daher nicht mit diesem einfachen Verfahren behandelt werden.

Hat man hingegen alle Zustände vollständig betrachtet und beim Aufbau der Tabelle keinen Konflikt erhalten, so erhält man gleich zwei Dinge: erstens, das Wissen, dass die Grammatik wirklich vom Typ LR(0) ist und zweitens, eine (hoffentlich) korrekte Syntaxanalysetabelle, mit der man nun zur Syntaxanalyse schreiten kann.

3.6.2 Beispiel

Versuchen wir nun selbst eine Syntaxanalysetabelle für G_1 herzustellen. Als Hilfsmittel benötigen wir den Zustandsübergangsgraphen aus Abbildung 3.1 und die Definition der Grammatik mit durchnumerierten Regeln:

$$
\begin{array}{lrcl}
\text{(1)} & Z & \rightarrow & S \\
\text{(2)} & S & \rightarrow & Sb \\
\text{(3)} & S & \rightarrow & bAa \\
\text{(4)} & A & \rightarrow & aSc \\
\text{(5)} & A & \rightarrow & a \\
\text{(6)} & A & \rightarrow & aSb
\end{array}
$$

Wir fangen mit Zustand 0 an. Es gibt genau zwei Pfeile, die diesen Zustand verlassen (einen b-Pfeil nach Zustand 3 und einen S-Pfeil nach Zustand 1). Nach dem obigen Algorithmus muss die erste Zeile in unserer Tabelle also folgendermaßen aussehen:

	Aktionstabelle				Sprungtabelle		
	a	b	c	$	A	S	Z
0		s3				1	

Da es in diesem Zustand keine vollständig akzeptierten Regeln (Punkt am Ende der rechten Seite) gibt, können wir mit dem nächsten Zustand fortfahren.

In Zustand 1 gibt es nur einen ausgehenden Pfeil (b nach Zustand 2). Dieser veranlasst uns, in Zeile 1 der Tabelle in der b-Spalte ein „shift 2“ einzutragen. Nun gibt es jedoch in Zustand 1 noch die vollständig akzeptierte, künstliche Startregel $[Z \rightarrow S.]$. Diese verpflichtet uns, in der Aktionstabelle in der $-Spalte der Zeile 1 ein „accept“ einzutragen.

Die erste Reduktion taucht im Zustand 2 auf und zwar nach Regel 2 (das ist die Grammatikregel, die dem vollständig akzeptiertem LR(0)-Element $[S \rightarrow Sb.]$ entspricht). Die Reduktionsoperation wird in der dem Zustand entsprechenden Zeile bei allen Terminalsymbolen eingetragen.

Im weiteren Verlauf stellt man fest, dass die Grammatik leider zu zwei Konflikten führt, d. h. Grammatik G_1 lässt sich nicht mit dem LR(0)-Verfahren behandeln. Der Vollständigkeit halber (und um zu sehen, ob man den Tabellenaufbau verstanden hat) ist dennoch im folgenden die LR(0)-Syntaxanalysetabelle inklusive der Konflikte zu sehen:

	Aktionstabelle				Sprungtabelle		
	a	b	c	$	A	S	Z
0		s3				1	
1		s2		acc			
2	r2	r2	r2	r2			
3	s6				4		
4	s5						
5	r3	r3	r3	r3			
6	r5	s3/r5	r5	r5		7	
7		s9	s8				
8	r4	r4	r4	r4			
9	r2/r6	r2/r6	r2/r6	r2/r6			

Wie wir sehen, hat die Grammatik G_1 beim LR(0)-Verfahren in zwei Zuständen (den Zuständen 6 und 9) Konflikte (einen s/r- und einen r/r-Konflikt).

KAPITEL 4

SLR(1)-SYNTAXANALYSE

4.1 Idee und Anwendung

Das SLR(1)-Verfahren könnte man als „aufgebohrtes“ LR(0)-Verfahren bezeichnen. Es wird zwar zur Vermeidung von Konflikten bereits eine Art Symbolvorgriff verwendet, ohne jedoch auf die erst in Kapitel 5 behandelten (komplizierteren) Lookahead-Mengen zurückzugreifen.

Das „S“ in SLR(1) steht übrigens für „simple“ und das nicht ohne Grund. Das Verfahren ist bis auf die Erstellung der Syntaxanalysetabelle mit dem LR(0)-Verfahren identisch, lediglich die Anzahl der (unnötigen) Reduktionen in der Syntaxanalysetabelle wird verringert.

Die Idee beim SLR(1)-Verfahren ist folgende: In der LR(0)-Analyse wurde immer reduziert, wenn die rechte Seite einer Regel erfolgreich erkannt wurde. Das SLR(1)-Verfahren nimmt nun eine zusätzliche Information hinzu: Bevor reduziert wird, fragt sich ein SLR(1)-Parser, ob das Symbol, was direkt hinter der erkannten rechten Seite steht, überhaupt dem Metasymbol auf der linken Seite folgen darf und reduziert nur in diesem Fall mit dieser Regel. Man benötigt also zu jedem Metasymbol eine Menge aller Symbole, die diesem folgen können – und *genau diese Menge* wird zufälligerweise (!?!) durch die bereits bekannte FOLLOW-Menge beschrieben (wir erinnern uns kurz an Abschnitt 1.3).

Um also eine SLR(1)-Syntaxanalysetabelle aufzubauen, benötigen wir (im Unterschied zur LR(0)-Analyse) drei Dinge: den Zustandsübergangsgraphen, die Grammatik mit durchnumerierten Produktionen und die FOLLOW-Mengen aller Metasymbole.

Der Aufbau funktioniert, was Schiebe- und Akzeptieren-Operationen in der Aktionstabelle sowie Zustandswechsel in der Sprungtabelle angeht, genauso wie bei LR(0), d. h. man geht Zustand für Zustand durch den Zustandsübergangsgraphen und erzeugt für die einzelnen Pfeile die entsprechenden Einträge.

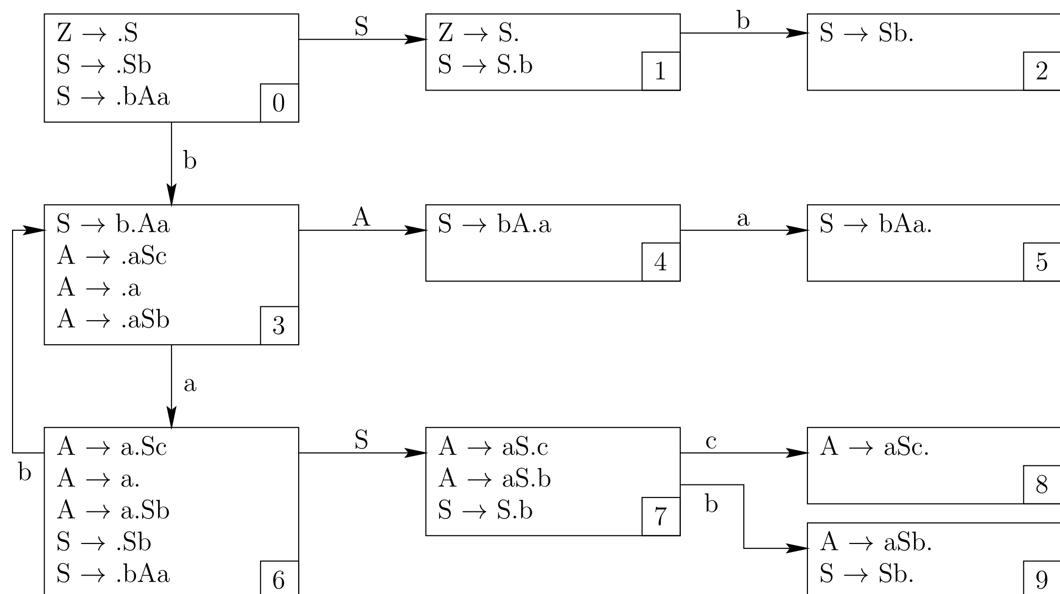

Abbildung 4.1: LR(0)-Zustandsübergangsgraph für Grammatik G_1

Nun wendet man sich allen LR(0)-Elementen zu, die einen Punkt am Ende der rechten Seite stehen haben, also vollständig akzeptiert wurden. Man trägt in der Aktionstabelle beim zugehörigen Zustand die Reduktion mittels der zugehörigen Regel bei allen Terminalsymbolen ein, die in der FOLLOW-Menge des Metasymbols auf der linken Seite der Regel enthalten sind.

Dabei wird man feststellen, dass viele Konflikte des LR(0)-Verfahrens beim SLR(1)-Verfahren einfach verschwinden. Sollten dennoch welche übrig bleiben, ist die gegebene Grammatik leider nicht vom Typ SLR(1), man muss also ein noch mächtigeres Verfahren verwenden.

Sollten hingegen keine Konflikte mehr auftauchen, so ist die Grammatik vom Typ SLR(1) und man hat eine funktionierende Syntaxanalysetabelle, die man zur Analyse verwenden kann.

4.2 Beispiel

Betrachten wir einmal mehr Grammatik G_1. Die Aufgabe soll darin bestehen, eine SLR(1)-Syntaxanalysetabelle aufzubauen, nachdem der LR(0)-Ansatz ja aufgrund zweier Konflikte fehlgeschlagen ist. Der LR(0)-Zustandsübergangsgraph ist noch einmal in Abbildung 4.1 dargestellt, und es folgt auch noch einmal die Grammatik selbst:

$$
\begin{array}{lrcl}
(1) & Z & \rightarrow & S \\
(2) & S & \rightarrow & Sb \\
(3) & S & \rightarrow & bAa \\
(4) & A & \rightarrow & aSc \\
(5) & A & \rightarrow & a \\
(6) & A & \rightarrow & aSb
\end{array}
$$

Bevor wir fortfahren, benötigen wir noch die FOLLOW-Mengen aller Metasymbole. Diese sind (siehe Abschnitt 1.3.3):

$$
\begin{array}{rcl}
\mathsf{FOLLOW}(Z) & = & \{\$\} \\
\mathsf{FOLLOW}(S) & = & \{b, c, \$\} \\
\mathsf{FOLLOW}(A) & = & \{a\}
\end{array}
$$

Und nun frisch ans Werk: Da Zustand 0 keine vollständig erkannte Regel enthält, können wir die erste Zeile der Tabelle wie gehabt ausfüllen:

	Aktionstabelle				Sprungtabelle		
	a	b	c	$	A	S	Z
0		s3				1	

In Zustand 1 fügen wir zunächst die Schiebeoperation ein:

	Aktionstabelle				Sprungtabelle		
	a	b	c	$	A	S	Z
0		s3				1	
1		s2					

Nun gibt es in diesem Zustand noch eine vollständig akzeptierte Regel, angezeigt durch das LR(0)-Element $[Z \rightarrow S.]$. Da es sich dabei um die künstliche Startregel handelt, tragen wir beim Eingabeendezeichen ($) die Aktion „accept“ ein. Wir erhalten als vollständige zweite Zeile:

	Aktionstabelle				Sprungtabelle		
	a	b	c	$	A	S	Z
0		s3				1	
1		s2		acc			

Im Zustand 2 taucht erstmals eine „echte“ Reduktion auf $(S \rightarrow Sb)$, die wir in den Spalten von b, c und $ ($\mathsf{FOLLOW}(S) = \{b, c, \$\}$) eintragen:

	Aktionstabelle				Sprungtabelle		
	a	b	c	$	A	S	Z
0		s3				1	
1		s2		acc			
2		r2	r2	r2			

Führt man das gesamte Verfahren mit den restlichen Zuständen durch, so erhält man schließlich die folgende Syntaxanalysetabelle:

	Aktionstabelle				Sprungtabelle		
	a	b	c	$	A	S	Z
0		s3				1	
1		s2		acc			
2		r2	r2	r2			
3	s6				4		
4	s5						
5		r3	r3	r3			
6	r5	s3				7	
7		s9	s8				
8	r4						
9	r6	r2	r2	r2			

Vergleicht man diese mit der LR(0)-Syntaxanalysetabelle auf Seite 24, so stellt man fest, dass mehrere Reduktionsoperationen entfernt und dadurch beide Konflikte beseitigt wurden.

KAPITEL 5

LR(1)-SYNTAXANALYSE

5.1 Idee

In der bisher als Beispiel verwendeten Grammatik G_1 waren wir bereits mit dem SLR(1)-Verfahren in der Lage, eine konfliktfreie Syntaxanalysetabelle aufzubauen. Im allgemeinen reicht dieses Verfahren jedoch leider noch nicht aus, um gängige Programmiersprachenkonstrukte zu behandeln. Die Lösung bringt das mächtigste aller (deterministischen) LR-Verfahren: das LR(1)-Verfahren (manchmal auch „kanonisches LR(1)-Verfahren" genannt, um es besser von den anderen zu unterscheiden).

Um mächtiger als die SLR(1)-Analyse sein zu können, benötigt das LR(1)-Verfahren mehr Informationen. In der SLR(1)-Analyse wurde mit Hilfe der FOLLOW-Mengen entschieden, ob das nächste Terminalsymbol überhaupt dem Metasymbol, zu dem reduziert werden soll, folgen darf. Nur im Positivfall wurde wirklich reduziert.

Die LR(1)-Analyse geht nun noch einen Schritt weiter. Zwar wird auch hier entsprechend einer Regel reduziert, wenn das nächste Symbol dem Metasymbol folgen kann. Dies geschieht aber nur dann, wenn dieses Metasymbol *genau durch diese Regel* entstanden ist. Mit anderen Worten, man betrachtet nicht mehr Mengen von Terminalsymbolen, die generell einem Metasymbol folgen können (FOLLOW-Mengen), sondern Mengen von Terminalsymbolen, die einem Metasymbol, nachdem es durch Anwendung einer bestimmten Regel entstanden ist, folgen können. Diese Mengen nennt man LOOKAHEAD-Mengen.[1]

Da diese LOOKAHEAD-Mengen nun pro Regel und nicht mehr pro Metasymbol erzeugt werden müssen, ist die Berechnung etwas komplizierter und wird während der Erstellung des Zustandsübergangsgraphen durchgeführt.

[1]Achtung: LOOKAHEAD- und FOLLOW-Mengen werden von Studenten *sehr* gerne verwechselt!

5.2 LR(1)-Elemente

Doch zunächst müssen wir mehrere, bereits bekannte Grundkonstruktionen erweitern und zwar die LR(0)-Elemente sowie die darauf aufbauenden Operationen $\mathsf{CLOSURE}_0$ und GOTO_0.

Ein LR(1)-Element ist ein Paar bestehend aus einem LR(0)-Element, welches in diesem Kontext „Kern“ genannt wird und einer LOOKAHEAD-Menge[2]. Diese gibt pro LR(1)-Element an, welche Terminalsymbole dem Metasymbol auf der linken Seite der Regel (nach Entstehung durch Reduktion entsprechend dieser Regel) folgen können. Es ergibt sich, dass diese Menge nur dann aussagekräftig ist, wenn sich der Punkt im Kern am rechten Ende befindet, also die rechte Seite vollständig akzeptiert wurde.

Ein Beispiel für ein LR(1)-Element aus Grammatik G_1 ist:

$$[Z \rightarrow .S\,,\,\{\$\}]$$

Dieses Element kann wie folgt gelesen werden: Bei der Bearbeitung der Regel $(Z \rightarrow S)$ befinden wir uns in der Analyse noch vor dem S. Sollten wir irgendwann die gesamte rechte Seite (also hier nur das S) erkannt haben, so reduzieren wir nach der Regel, aber nur unter der Voraussetzung, dass das nächste Zeichen ein Element der LOOKAHEAD-Menge (also hier $\$$) ist.

Ein LR(1)-Zustandsübergangsgraph unterscheidet sich dadurch von einem LR(0)-Zustandsübergangsgraphen, dass er statt aus LR(0)- aus LR(1)-Elementen aufgebaut ist.

5.3 Der Hüllenoperator $\mathsf{CLOSURE}_1$

5.3.1 Algorithmus zur Berechnung

Um nun von einer Grammatik über einen LR(1)-Zustandsübergangsgraphen zur LR(1)-Syntaxanalysetabelle zu kommen, müssen noch die Hilfsoperationen $\mathsf{CLOSURE}_0$ und GOTO_0 zu $\mathsf{CLOSURE}_1$ und GOTO_1 erweitert werden. Dies ist notwendig, um die LOOKAHEAD-Mengen berechnen zu können.

Die Menge $\mathsf{CLOSURE}_1(I)$ einer Menge I von LR(1)-Elementen wird wie folgt berechnet:

1. füge I zu $\mathsf{CLOSURE}_1(I)$ hinzu (also dasselbe wie bei $\mathsf{CLOSURE}_0$)

2. gibt es ein Element $[A \rightarrow \alpha.B\beta\,,\,L]$ in $\mathsf{CLOSURE}_1(I)$ (wobei $\alpha = \varepsilon$ und/oder $\beta = \varepsilon$ sein darf) und eine Produktion $(B \rightarrow \gamma)$, so füge auch $[B \rightarrow .\gamma\,,\,\mathsf{FIRST}(\beta L)]$ zu $\mathsf{CLOSURE}_1(I)$ hinzu

[2] In manchen Compilerbaubüchern korrekt übersetzt „Vorausschaumenge“ genannt

Dem kritischen Leser werden an dieser Stelle gleich zwei problematische (da noch nicht eingeführte) Konstruktionen ins Auge stechen: zum einen die Konkatenation von einer Zeichenkette mit einer Menge (βL) und zum anderen die FIRST-Mengenbildung von einer Menge (wir haben bisher nur die FIRST-Mengen von Zeichenketten definiert).

Die Konkatenation von einer Zeichenkette β mit einer Menge von Zeichenketten L ist nicht weiter kompliziert: Das Ergebnis ist eine Menge von Zeichenketten, die alle durch die Konkatenation von β mit den einzelnen Elementen aus L entstanden sind. Wesentlich theoretischer und genauer drückt es die folgende Definition aus ($\beta \in (V \cup \Sigma)^*, L \subseteq (V \cup \Sigma)^*$):

$$\beta L = \{y \mid \exists_{x \in L} : y = \beta x\}$$

Nun bleibt noch die Definition von FIRST-Mengen auf Mengen offen, diese ist aber erfreulicherweise recht intuitiv, da sie einfach die Vereinigung der FIRST-Mengen der einzelnen Elemente der gegebenen Menge darstellt:

$$\mathsf{FIRST}(L) = \bigcup_{\alpha \in L} \mathsf{FIRST}(\alpha)$$

Da insbesondere die zweite Regel der Hüllendefinition etwas komplizierter zu sein scheint, erfolgt nun die Beschreibung des gesamten Algorithmus zur Berechnung von CLOSURE$_1(I)$ noch einmal mit Worten.

Zunächst nimmt man alle Elemente aus I und fügt sie der Hülle hinzu. Nun betrachtet man alle Elemente der Hülle, die einen Punkt unmittelbar links von einem Metasymbol (B) stehen haben. Man fügt alle Produktionen der Art ($B \to \ldots$) als neue LR(1)-Elemente der Hülle hinzu, die man wie folgt bildet: Den Regelteil erhält man, indem man in der Produktion einen Punkt an den Anfang der rechten Seite setzt (also wie bei CLOSURE$_0$).

Die neue LOOKAHEAD-Menge ist gleich die FIRST-Menge aller Symbole, die dem Metasymbol folgen (β), verkettet mit der LOOKAHEAD-Menge des ursprünglichen LR(1)-Elementes (erst Verkettung, dann FIRST-Mengenbildung).

Das heißt, wenn dem Metasymbol ein Terminalsymbol folgt, besteht die neue LOOKAHEAD-Menge aus genau diesem Terminalsymbol (Konsequenz aus der FIRST-Mengenbildung). Folgt dem Metasymbol kein weiteres Symbol, ist die neue LOOKAHEAD-Menge gleich der alten (da $\mathsf{FIRST}(\varepsilon L) = \mathsf{FIRST}(L)$). Sollte hingegen dem Metasymbol eine andere Zeichenkette folgen, so ist die LOOKAHEAD-Menge gleich die FIRST-Menge von dieser Zeichenkette, wobei die alte LOOKAHEAD-Menge hinzugenommen wird, falls die Zeichenkette ε-ableitbar war.

5.3.2 Beispiel

Betrachten wir noch einmal das LR(1)-Element $[Z \to .S\,,\ \{\$\}]$ aus Grammatik G_1 und bilden davon CLOSURE$_1$. Zunächst wissen wir, dass das Element selbst Teil der Hülle ist:

$[Z \to .S\,, \{\$\}]$

Nun sehen wir, dass sich in einem Element der Hülle ($[Z \to .S\,,\ \{\$\}]$) ein Metasymbol ($S$) unmittelbar rechts von einem Punkt befindet. Diesem Metasymbol folgt kein weiteres Symbol. Daher fügen wir alle Produktionen, die ein S auf der linken Seite haben, als neue LR(1)-Elemente der Hülle hinzu. Die LOOKAHEAD-Menge dieser neuen Elemente ist gleich FIRST$(\varepsilon\$) = \{\$\}$. Damit sieht unsere Hülle jetzt so aus:

$[Z \to .S\quad, \{\$\}]$
$[S \to .Sb\quad, \{\$\}]$
$[S \to .bAa\,, \{\$\}]$

Nun schauen wir, ob in den neuen Regeln wieder ein Metasymbol unmittelbar rechts von einem Punkt steht und dem ist tatsächlich so ($[S \to .Sb\,,\ \{\$\}]$). Wir müssen also wieder alle von S abgeleiteten Produktionen als LR(1)-Elemente hinzunehmen, diesmal allerdings mit neuer LOOKAHEAD-Menge und zwar FIRST$(b\$) = \{b\}$. Das Ergebnis ist:

$[Z \to .S\quad, \{\$\}]$
$[S \to .Sb\quad, \{\$\}]$
$[S \to .bAa\,, \{\$\}]$
$[S \to .Sb\quad, \{b\}]$
$[S \to .bAa\,, \{b\}]$

Versucht man nun noch einmal den letzten Algorithmusschritt durchzuführen, stellt man fest, dass keine neuen Elemente mehr hinzukommen, d. h. der obige Kasten enthält CLOSURE$_1$ ($[Z \to .S\,,\ \{\$\}]$). Die Notation ist jedoch noch verbesserungswürdig.

Um (insbesondere bei großen LOOKAHEAD-Mengen) den Schreibaufwand zu reduzieren, ist es üblich, LR(1)-Elemente, die sich nur in den LOOKAHEAD-Mengen unterscheiden, zusammenzufassen. Wir bekommen damit als Endergebnis:

$[Z \to .S\quad, \{\$\}]$
$[S \to .Sb\quad, \{b, \$\}]$
$[S \to .bAa\,, \{b, \$\}]$

5.4 Die Sprungoperation GOTO$_1$

Im Gegensatz zur Hüllenoperation ist die Erweiterung der Sprungoperation minimal. GOTO$_1$ ist wie folgt definiert:

$$\mathsf{GOTO}_1(I, X) = \mathsf{CLOSURE}_1\left(\left\{\ [A \to \alpha X.\beta\,,\ L]\ \middle|\ [A \to \alpha.X\beta\,,\ L] \in I\ \right\}\right)$$

Auf eine weitere Erklärung oder ein Beispiel verzichte ich hier, da GOTO$_1$ bis auf das Kopieren der LOOKAHEAD-Menge mit GOTO$_0$ identisch ist.

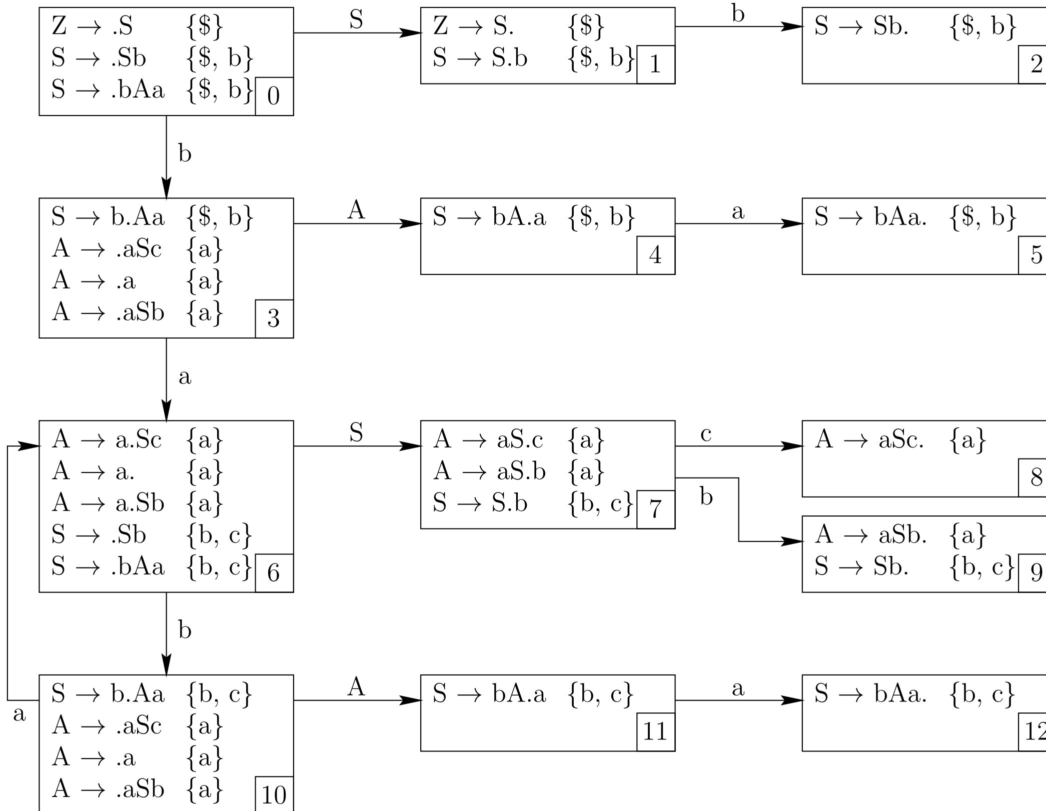

Abbildung 5.1: LR(1)-Zustandsübergangsgraph für Grammatik G_1

5.5 Das Aufstellen des Zustandsübergangsgraphen

Auch zum Aufstellen des Zustandsübergangsgraphen ist nicht viel neues zu berichten, da das Verfahren analog zum LR(0)-Verfahren funktioniert. Lediglich das Startelement ist nun ein LR(1)-Element und zwar (wenn S' das künstliche und S das eigentliche Startsymbol ist):

$$[S' \to .S\,,\ \{\$\}]$$

In Abbildung 5.1 ist der vollständige LR(1)-Zustandsübergangsgraph für Grammatik G_1 angegeben. Man beachte, dass er mehr Zustände besitzt als sein LR(0)-Pendant, da mehrere Zustände auseinandergenommen wurden.

Es ist bei der Konstruktion penibel darauf zu achten, dass zwei Zustände nur dann identisch sind, wenn sowohl die Regeln, als auch die LOOKAHEAD-Mengen aller enthaltenen LR(1)-Elemente identisch sind.[3]

[3]wirklich *sehr* beliebte Fehlerquelle

5.6 Das Erstellen der Syntaxanalysetabelle

Das Aufstellen der Syntaxanalysetabelle ist eigentlich leichter als beim SLR(1)-Verfahren, da alle benötigten Informationen (von der Numerierung der Grammatikregeln mal abgesehen) im Graphen enthalten sind. Die Tabelle wird bis auf die Reduktionsoperationen genauso aufgebaut, wie wir es bereits vom LR(0)/SLR(1)-Verfahren her kennen.

Reduktionsoperationen werden hingegen zwar auch wieder in den entsprechenden Zeilen (siehe LR(0)/SLR(1)) eingetragen, aber diesmal nur in den Spalten der Terminalsymbole, die in der LOOKAHEAD-Menge der entsprechenden Regel enthalten sind.

Nehmen wir beispielsweise den Zustandsübergangsgraphen zu Grammatik G_1 aus Abbildung 5.1 und schauen uns zusätzlich noch einmal die Definition von G_1 (wegen der Numerierung der Produktionen) an:

$$
\begin{array}{lrcl}
(1) & Z & \rightarrow & S \\
(2) & S & \rightarrow & Sb \\
(3) & S & \rightarrow & bAa \\
(4) & A & \rightarrow & aSc \\
(5) & A & \rightarrow & a \\
(6) & A & \rightarrow & aSb
\end{array}
$$

Dann sieht die zugehörige Syntaxanalysetabelle wie folgt aus:

	Aktionstabelle				Sprungtabelle		
	a	b	c	$	A	S	Z
0		s3				1	
1		s2		acc			
2		r2		r2			
3	s6				4		
4	s5						
5		r3		r3			
6	r5	s10				7	
7		s9	s8				
8	r4						
9	r6	r2	r2				
10	s6				11		
11	s12						
12		r3	r3				

In der Syntaxanalysetabelle sieht man bereits den größten Nachteil des LR(1)-Verfahrens: die Tabellen werden größer, als bei den anderen Verfahren (im allgemeinen sehr viel größer!). Dafür sind wir nun jedoch in der Lage mit allen relevanten Programmiersprachenkonstrukten umgehen zu können.

KAPITEL 6

LALR(1)-SYNTAXANALYSE

6.1 Idee

Als allererstes sollte man, um Missverständnisse von vornherein zu vermeiden, folgendes festhalten: Der Name LALR, also lookahead-LR, ist einer der unglücklichsten, den man für dieses Verfahren wählen konnte, da sowohl LR(1) als auch SLR(1) mittels Symbolvorgriff arbeiten.

Nachdem wir festgestellt haben, dass das mächtigste aller deterministischen LR-Verfahren mit einem Zeichen Vorgriff, das LR(1)-Verfahren, doch eigentlich machbar ist, stellt sich natürlich die Frage, wozu man sich noch mit einem weiteren Verfahren quälen soll, das zudem weniger mächtig ist. Die Antwort wurde bereits in Ansätzen am Ende des vorherigen Abschnitts sichtbar: Die Syntaxanalysetabellen werden beim LR(1)-Verfahren zu groß. Allerdings konnte man in dem Beispiel auch sehen, dass sich einige Zustände verblüffend ähnlich (soll heißen: bis auf die LOOKAHEAD-Mengen identisch) sind.

Das LALR(1)-Verfahren funktioniert nun folgendermaßen: man generiert zunächst einen LR(1)-Zustandsübergangsgraphen. Anschließend fasst man alle Zustände zusammen, die sich ausschließlich in den LOOKAHEAD-Mengen unterscheiden (d. h. die jeweiligen Kerne sind identisch), indem man diese bei den entsprechenden LR(1)-Elementen vereinigt. Anschließend (oder besser: dabei) „biegt“ man auch die Pfeile entsprechend um. Im Ergebnis sollte man einen LALR(1)-Zustandsübergangsgraphen erhalten, der die gleichen Elemente wie der LR(0)-Zustandsübergangsgraph enthält, nur dass diese hier LOOKAHEAD-Mengen besitzen.

Anmerkung: Man kann auch direkt aus einem LR(0)- einen LALR(1)-Zustandsübergangsgraphen durch bloßes Hinzufügen der LOOKAHEAD-Mengen generieren, was vor allem viele Parsergeneratoren praktizieren, der Algorithmus ist jedoch etwas komplizierter und weniger einsichtig, als der hier beschriebene.

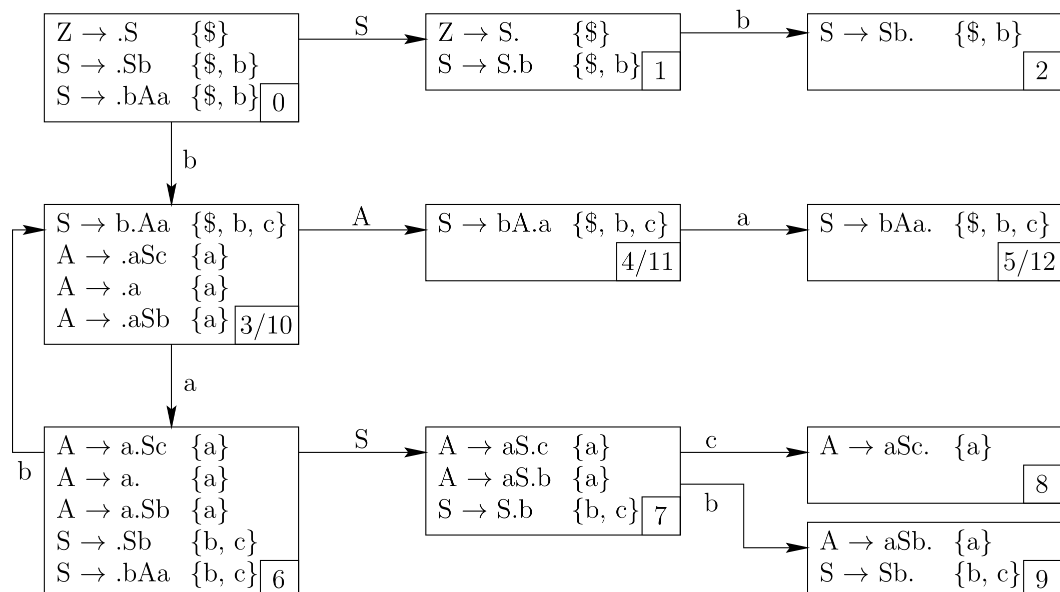

Abbildung 6.1: LALR(1)-Zustandsübergangsgraph für Grammatik G_1

Die ganze Zusammenfasserei hat jedoch einen Nachteil: Es können dabei neue Konflikte entstehen und zwar r/r-Konflikte. Im allgemeinen lassen sich jedoch alle gängigen Programmiersprachenkonstrukte von LALR(1)-Parsern verarbeiten.

Zum Aufbau der Syntaxanalysetabelle ist nichts weiter hinzuzufügen, da der Algorithmus mit dem aus dem LR(1)-Verfahren identisch ist.

6.2 Beispiel

Zur Selbstkontrolle ist in Abbildung 6.1 der LALR(1)-Zustandsübergangsgraph zu Grammatik G_1 zu sehen. Die daraus generierte LALR(1)-Syntaxanalysetabelle ist in Abbildung 6.2 dargestellt.

	Aktionstabelle				Sprungtabelle		
	a	b	c	$	A	S	Z
0		s3/10				1	
1		s2		acc			
2		r2		r2			
3/10	s6				4/11		
4/11	s5/12						
5/12		r3	r3	r3			
6	r5	s3/10				7	
7		s9	s8				
8	r4						
9	r6	r2	r2				

Abbildung 6.2: LALR(1)-Syntaxanalysetabelle für Grammatik G_1

KAPITEL 7

LR(K)-SYNTAXANALYSE FÜR $K \geq 2$

7.1 Motivation

Nachdem wir in den vergangenen Kapiteln verschiedene LR-Verfahren mit keinem und einem Zeichen Vorgriff betrachtet haben, wollen wir uns nun dem kanonischen LR(k)-Verfahren für beliebige k nähern. Bevor wir uns jedoch in die Tiefen dieses mächtigsten aller LR-Syntaxanalyseverfahren stürzen, sollten wir einige Vorbetrachtungen anstellen.

In der verfügbaren Compilerbauliteratur sind Bücher, die sich umfassend mit LR(k)[1] beschäftigen, eher selten zu finden, was vor allem daran liegt, dass LR(k)-Parser mit $k \geq 2$ in der Praxis eher selten benutzt werden. Dafür gibt es mehrere Gründe, und diese sollte jeder kennen, der beabsichtigt, einen LR(k)-Parser zu implementieren.

Das Hauptargument gegen LR(k) liefert die deprimierende Erkenntnis, dass bei den LR(k)-Syntaxanalyseverfahren (im Gegensatz zu den LL(k)-Syntaxanalyseverfahren) *nicht* gilt, dass mit steigendem k die Anzahl der analysierbaren Sprachen wächst. Es gilt statt dessen folgender Satz:

Zu jeder LR(k)-Sprache gibt es eine LR(1)-Grammatik.

Als Konsequenz dieses Satzes ergibt sich, dass die einzelnen Mengen der LR(k)-Sprachen für beliebige $k \geq 1$ untereinander identisch sind (nur die Menge der LR(0)-Sprachen ist weniger mächtig).

Des weiteren gibt es Algorithmen, die eine LR(k)-Grammatik in eine äquivalente LR(1)-Grammatik umformen.[2] Man kann also schlussfolgern, dass sich jede mit dem

[1]In diesem Abschnitt bedeutet LR(k) an manchen Stellen „LR(k) für beliebige k“ und an anderen Stellen „LR(k) für $k \geq 2$“. Welches jeweils gemeint ist, sollte sich aus dem jeweiligen Kontext ergeben.

[2]Genaugenommen gibt es einen Algorithmus, der eine LR(k)-Grammatik ($k > 1$) in eine LR($k-1$)-Grammatik umwandelt. Iterativ angewendet ergibt sich der gewünschte Effekt.

LR(k)-Verfahren analysierbare Sprache auch mit dem bereits vorgestellten kanonischen LR(1)-Verfahren erschlagen lässt.

Neben diesen theoretischen Überlegungen gibt es auch praktische Gründe, die gegen LR(k) sprechen: Der Aufwand der LR(k)-Analyse überwiegt dem der LR(1)-Analyse und zwar umso mehr, je höher das k ist. Wieviel genau wird sich im Laufe des Kapitels herausstellen.

Den genannten Argumenten zum Trotz gibt es aber auch Gründe, die für die allgemeine LR(k)-Analyse sprechen (ansonsten wäre dieses Kapitel auch ziemlich überflüssig) und zwar werden diese sehr schön in [Par96] beschrieben. Das Hauptargument besteht darin, dass man zwar zu jeder LR(k)-Grammatik eine LR(1)-Grammatik aufstellen kann, aber zum einen deren Umfang deutlich größer als der der Ausgangsgrammatik ist und zum anderen bei ihrer Generierung ein Großteil der Struktur der Ausgangsgrammatik verlorengeht.

Damit ist es zwar immer noch problemlos möglich, einen reinen Akzeptor (also einen Parser, der nur die Syntaxanalyse durchführt) zu konstruieren, aber in der Regel möchte man keine Akzeptoren, sondern Compiler implementieren. Letztere unterscheiden sich von reinen Akzeptoren dadurch, dass während der Syntaxanalyse semantische Aktionen durchgeführt werden (Syntaxbaumkonstruktion, Symboltabellenaufbau, Codegenerierung, ...). Diese bei der Konstruktion des Compilers sinnvoll in selbigen einzubauen ist schon bei einer „schönen" Grammatik[3] ein anspruchsvolles Problem; bei Grammatiken, die durch theoretische Umformungen entstanden sind, ist dies eine inakzeptable, da fehleranfällige und sehr aufwendige, Aufgabe.

Zusammenfassend kann man also sagen, dass es mehrere (vor allem theoretische) Gründe gibt, die LR(k) überflüssig erscheinen lassen, es aber aufgrund praktischer Anforderungen trotzdem durchaus Sinn macht, sich mit der allgemeinen LR(k)-Analyse zu beschäftigen.

7.2 FIRST$_k$ und FOLLOW$_k$

Wie bereits mehrfach im Kapitel zu den Grundbegriffen (Kapitel 1) angedeutet, werden jetzt einige der dort vorgestellten Definitionen vom jeweiligen Spezialfall 1 auf den allgemeinen Fall k erweitert.

Doch zunächst benötigen wir einige Hilfskonstruktionen, die uns die später folgenden Definitionen deutlich erleichtern.

7.2.1 Die Hilfsoperationen k-Präfix und k-Konkatenation

Insbesondere für die formale Definition von FIRST$_k$-Mengen wird eine spezielle Hilfsoperation benötigt und zwar der sogenannte k-Präfix. Dieser ist für eine Zei-

[3] Soll heißen: eine Grammatik, die die Sprache sinnvoll in semantische Einheiten gliedert.

chenkette $\alpha = a_1 \dots a_n$ mit $a_i \in (\Sigma \cup V)$ wie folgt definiert:

$$\alpha_{|k} := \begin{cases} a_1 \dots a_n \quad (= \alpha) & \text{, wenn } n \leq k \\ a_1 \dots a_k & \text{, sonst} \end{cases}$$

Der k-Präfix für ein Wort α ist also eine Zeichenkette, bestehend aus den ersten k Zeichen von α bzw. α selbst, falls α aus weniger als k Symbolen besteht.

Eine weitere Hilfsoperation ist die sogenannte k-Konkatenation ($\oplus_k$). Diese beschreibt, welche Zeichenkette die ersten k Symbole einer Konkatenation von zwei gegebenen Zeichenketten bilden. Verständlicher und formaler ist die folgende Definition:

$$\alpha \oplus_k \beta = (\alpha\beta)_{|k}$$

Beide Hilfsoperationen lassen sich nach den folgenden Definitionen auch auf Sprachen anwenden:

$$\begin{aligned} L_{|k} &:= \left\{ \alpha_{|k} \,\middle|\, \alpha \in L \right\} \\ L_1 \oplus_k L_2 &:= \left\{ \alpha \oplus_k \beta \,\middle|\, \alpha \in L_1 \wedge \beta \in L_2 \right\} \end{aligned}$$

7.2.2 FIRST$_k$

Die FIRST$_k$-Mengen stellen eine Erweiterung der in Abschnitt 1.2 auf Seite 8 eingeführten FIRST$_1$-Mengen dar.

Die FIRST$_k$-Menge einer gegebenen Zeichenkette X gibt an, welche Zeichenketten, bestehend aus k Symbolen, als Anfänge der von X abgeleiteten Zeichenketten auftreten können, wobei alle möglichen Ableitungen berücksichtigt werden (sollten Ableitungen von X mit weniger als k Symbolen existieren, gehören diese ebenfalls zu FIRST$_k(X)$).

Mit Hilfe des k-Präfixes lassen sich FIRST$_k$-Mengen recht unproblematisch formal definieren:

$$\begin{aligned} \text{FIRST}_k &: (V \cup \Sigma)^+ \to \mathcal{P}(\Sigma^*)_{|k} \\ \text{FIRST}_k(x) &:= \left\{ y_{|k} \,\middle|\, x \Rightarrow^* y \right\} \end{aligned}$$

7.2.3 FOLLOW$_k$

Auch die FOLLOW$_k$-Mengen stellen eine Erweiterung dar, und zwar die der in Abschnitt 1.3 auf Seite 10 vorgestellten FOLLOW$_1$-Mengen.

Die FOLLOW$_k$-Menge eines Metasymbols enthält alle aus k Terminalen bestehenden Zeichenketten, die in irgendeinem Ableitungsschritt unmittelbar rechts von diesem Metasymbol stehen können (wobei auch Zeichenketten aus weniger als k Terminalen berücksichtigt werden, wenn unmittelbar nach ihnen die Eingabe endet).

Dank der bereits erfolgten Definitionen von FIRST_k und k-Konkatenation lässt sich die Definition von FOLLOW_k sehr elegant formulieren ($\alpha, \beta, \gamma \in (\Sigma \cup V)^*$):

$$\begin{aligned} \mathsf{FOLLOW}_k &: V \rightarrow \mathcal{P}(\Sigma^*) \oplus_k \{\$\} \\ \mathsf{FOLLOW}_k(X) &:= \left\{\gamma \;\middle|\; S \Rightarrow^* \alpha X \beta \;\wedge\; \gamma \in \mathsf{FIRST}_k(\beta)\right\} \;\oplus_k\; \{\$\} \end{aligned}$$

7.3 LR(k)-Elemente, $\mathsf{CLOSURE}_k$ und GOTO_k

Auch die LR(1)-Elemente müssen für das allgemeine LR(k)-Verfahren zu LR(k)-Elementen erweitert werden. Jedes Konstrukt der folgenden Form heißt LR(k)-Element:

$$[X \rightarrow \alpha_1.\alpha_2\,,\, L] \qquad (\alpha_1, \alpha_2 \in (\Sigma \cup V)^*,\ L \subseteq \mathcal{P}(\Sigma^*) \oplus_k \{\$\}),$$

Ein LR(k)-Element besteht also aus einem Kern (dem LR(0)-Element) und einer Vorausschaumenge, wobei letztere nun aus Terminalketten besteht, die entweder genau k Zeichen lang ist oder aber kürzer ist und mit einem Eingabeendesymbol aufhört.

Nach kurzem Überlegen wird man feststellen, dass die in Abschnitt 5.2 auf Seite 32 eingeführten LR(1)-Elemente wirklich einen Spezialfall der hier definierten LR(k)-Elemente bilden.

Die Berechnung von $\mathsf{CLOSURE}_k$ folgt analog zur Berechnung von $\mathsf{CLOSURE}_1$, nur dass nun auf die allgemeinen Hilfskonstruktionen zurückgegriffen wird. Die beiden Schritte zur Berechnung von $\mathsf{CLOSURE}_k(I)$ lauten:

1. füge I zu $\mathsf{CLOSURE}_k(I)$ hinzu
2. gibt es ein Element $[A \rightarrow \alpha.B\beta\,,\, L]$ in $\mathsf{CLOSURE}_k(I)$ (wobei $\alpha = \varepsilon$ und/oder $\beta = \varepsilon$ sein darf) und eine Produktion $(B \rightarrow \gamma)$, so füge auch $[B \rightarrow .\gamma\,,\, \mathsf{FIRST}_k(\beta L)]$ zu $\mathsf{CLOSURE}_k(I)$ hinzu

Die Definitionen von GOTO_k und GOTO_1 unterscheiden sich ebenfalls nur marginal (es wird $\mathsf{CLOSURE}_k$ statt $\mathsf{CLOSURE}_1$ benutzt):

$$\mathsf{GOTO}_k(I, X) = \mathsf{CLOSURE}_k\left(\left\{\ [A \rightarrow \alpha X.\beta\,,\, L] \;\middle|\; [A \rightarrow \alpha.X\beta\,,\, L] \in I\ \right\}\right)$$

7.4 LR(k)-Zustandsübergangsgraphen und LR(k)-Syntaxanalysetabellen

Die Konstruktion des LR(k)-Zustandsübergangsgraphen verläuft (wie man vermuten würde) analog zur Erstellung von LR(1)-Zustandsübergangsgraphen, nur dass man nun mit den Operationen $\mathsf{CLOSURE}_k$ und GOTO_k auf Mengen von LR(k)-Elementen arbeitet.

Die Erstellung der LR(k)-Syntaxanalysetabelle ist hingegen etwas komplizierter als die ihres LR(1)-Pendants. Der Unterschied zwischen den beiden Tabellen besteht darin, dass die Aktionstabelle der LR(k)-Syntaxanalysetabelle deutlich mehr Spalten enthält. Der Grund dafür liegt darin, dass in den Spaltenköpfen keine Terminalsymbole sondern alle möglichen Zeichenketten, bestehend aus jeweils k Terminalen, untergebracht sind. Dazu kommen noch alle möglichen Zeichenketten, die aus weniger als k Terminalen bestehen und mit einem $ enden.

Als Anzahl s der Spalten der Aktionstabelle ergibt sich bei t Terminalen und einem LOOKAHEAD von k Symbolen ($k \geq 1$):

$$s = \sum_{i=0}^{k} t^i = \begin{cases} \frac{t^{k+1}-1}{t-1} & , \; t \neq 1 \\ k+1 & , \; t = 1 \end{cases}$$

Die folgende Tabelle soll das Größenwachstum der Spaltenanzahl verdeutlichen. Man kann leicht nachvollziehen, warum in der Praxis versucht wird, sich auf möglichst kleine k, am besten $k = 1$, zu beschränken:

t \ k	1	2	3	4	5	6	7	8	9
0	1	1	1	1	1	1	1	1	1
1	2	3	4	5	6	7	8	9	10
2	3	7	15	31	63	127	255	511	1023
3	4	13	40	121	364	1093	3280	9841	29524
4	5	21	85	341	1365	5461	21845	87381	349525
5	6	31	156	781	3906	19531	97656	488281	2441406
6	7	43	259	1555	9331	55987	335923	2015539	12093235
7	8	57	400	2801	19608	137257	960800	6725601	47079208
8	9	73	585	4681	37449	299593	2396745	19173961	153391689
9	10	91	820	7381	66430	597871	5380840	48427561	435848050

Das Ausfüllen der Syntaxanalysetabelle geschieht wieder analog zum LR(1)-Verfahren, nur dass man in der Aktionstabelle nun mit Zeichenketten statt mit einzelnen Terminalsymbolen hantieren muss.

7.5 LR(k)-Syntaxanalyse

Hat man eine Syntaxanalysetabelle fertiggestellt, so kann man sich an die Syntaxanalyse wagen. Obwohl auch diese völlig analog zum LR(1)-Verfahren abläuft, ist sie beim LR(k)-Verfahren am Anfang etwas gewöhnungsbedürftig. Der Grund dafür liegt darin, dass man zwar in der Syntaxanalysetabelle mit Zeichenketten, in der Eingabe und im Kellerspeicher aber weiterhin mit einzelnen Terminalsymbolen arbeitet.

In jedem Analyseschritt bestimmt sich die auszuführende Aktion mittels der Syntaxanalysetabelle aus dem aktuellen Zustand (oberstes Symbol im Kellerspeicher) und der Zeichenkette bestehend aus den ersten k Terminalsymbolen in der Eingabe

(bzw. der Eingabe selbst, falls diese kürzer als k sein sollte). Die so ermittelte Aktion wird dann genau wie beim LR(1)-Verfahren durchgeführt (dies gilt insbesondere für Schiebeoperationen: auch beim LR(k)-Verfahren wird nur ein einzelnes Terminalsymbol geschoben – die folgenden Terminale werden im folgenden Schritt wieder als LOOKAHEAD benutzt!).

Der Syntaxanalyseablauf am Ende des folgenden Beispiels sollte Klarheit bringen.

7.6 Beispiel

Wir besitzen nun alle notwendigen Informationen für die LR(k)-Syntaxanalyse, um aus Grammatiken LR(k)-Syntaxanalysetabellen zu generieren. Wir wollen uns im folgenden Beispiel mit der einfachsten Form der LR(k)-Syntaxanalyse für $k > 1$ beschäftigen: der LR(2)-Syntaxanalyse.

Als Beispielgrammatik soll folgende Grammatik G_2 dienen:

$$
\begin{array}{lrcl}
(1) & S & \rightarrow & Yaa \\
(2) & S & \rightarrow & Xa \\
(3) & X & \rightarrow & b \\
(4) & Y & \rightarrow & b
\end{array}
$$

Man kann sich schnell davon überzeugen, dass die Grammatik nicht vom Typ LR(1) ist, da der LR(1)-Zustandsübergangsgraph in einem Zustand einen reduce/reduce-Konflikt besitzt.

Betrachten wir zunächst den durch die Hülle CLOSURE$_k([Z \rightarrow S\,,\ \{\$\}])$ gebildeten Startzustand:

$[Z \rightarrow .S\ \ ,\ \{\$\}]$
$[S \rightarrow .Yaa\,,\ \{\$\}]$
$[S \rightarrow .Xa\ \ ,\ \{\$\}]$
$[X \rightarrow .b\ \ ,\ \{a\$\}]$
$[Y \rightarrow .b\ \ ,\ \{aa\}]$

Der einzige Unterschied zum Startzustand des LR(1)-Zustandsübergangsgraphen besteht in den letzten beiden LR(2)-Elementen, deren LOOKAHEAD-Mengen nun aus Zeichenketten der Länge zwei bestehen (aa und $a\$$).

Betrachtet man den gesamten Zustandsübergangsgraphen (zu sehen in Abbildung 7.1), so stellt man fest, dass nur ein weiterer Zustand einer näheren Betrachtung bedarf und zwar Zustand 7:

$[X \rightarrow b.\,,\ \{a\$\}]$
$[Y \rightarrow b.\,,\ \{aa\}]$

An diesem Zustand kann man gut erkennen, warum die Grammatik nicht vom Typ LR(1) ist und warum LR(2) das Problem löst.

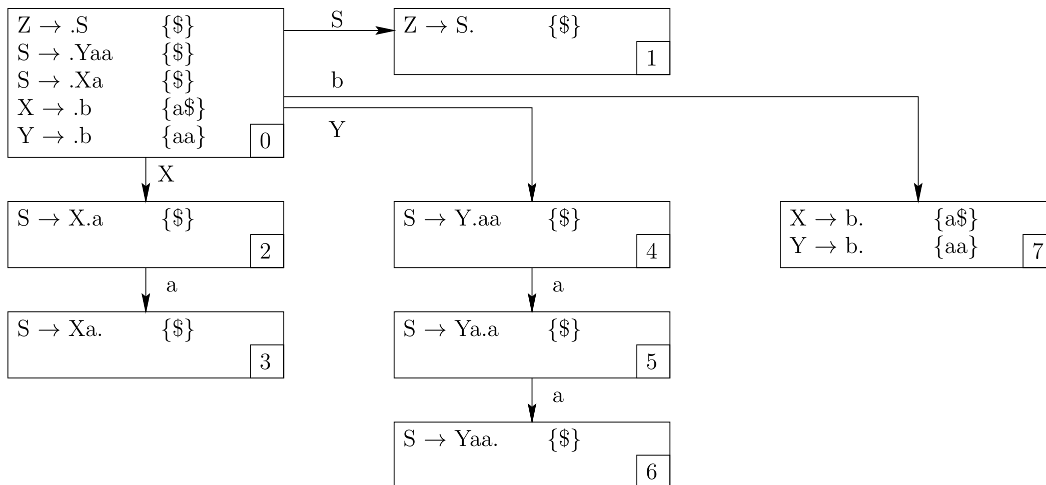

Abbildung 7.1: LR(2)-Zustandsübergangsgraph für Grammatik G_2

Begibt man sich nun an den Aufbau der Syntaxanalysetabelle, sollte man zu folgendem Ergebnis kommen (es ist relativ auffällig, dass die Tabelle durch die „Zweierzeichenkettenbildung“ etwas an Breite gewonnen hat):

	Aktionstabelle							Sprungtabelle		
	aa	*ab*	*a$*	*ba*	*bb*	*b$*	$	*S*	*X*	*Y*
0				s7	s7	s7		1	2	4
1							acc			
2	s3	s3	s3							
3							r2			
4	s5	s5	s5							
5	s6	s6	s6							
6							r1			
7	r4		r3							

Um die Syntaxanalysetabelle zu prüfen (genaugenommen: um zu zeigen, dass sie zumindest für ein Beispiel funktioniert), soll im folgenden das Wort baa mit ihrer Hilfe abgeleitet werden. Der dabei entstehende Ablauf ist:

Stack	Eingabe	Aktion
$\$ 0$	$b\ a\ a\ \$$	shift 7
$\$ 0\ b\ 7$	$a\ a\ \$$	reduce 4 ($Y \rightarrow b$)
$\$ 0\ Y\ 4$	$a\ a\ \$$	shift 5
$\$ 0\ Y\ 4\ a\ 5$	$a\ \$$	shift 6
$\$ 0\ Y\ 4\ a\ 5\ a\ 6$	$\$$	reduce 1 ($S \rightarrow Yaa$)
$\$ 0\ S\ 1$	$\$$	accept

ANHANG A

AUFGABEN

Zur Selbstüberprüfung habe ich in diesem Anhang mehrere Übungsaufgaben zusammengestellt. So ich ausreichend Zeit finde, werde ich weitere hinzufügen. Für alle, die „Opfer“ einer von mir verfassten Klausuraufgabe wurden oder noch in diese Situation kommen können, weise ich zusätzlich bei den aus Klausuren entnommenen Grammatiken auf ihre Herkunft hin.

Die Einteilung in einfache und kompliziertere Grammatiken ist natürlich rein subjektiv (schließlich erschlagen die in diesem Pamphlet beschriebenen Algorithmen alle LR-Grammatiken unabhängig von ihrem Umfang oder Aufbau).

Im unmittelbar folgenden Abschnitt sind die Grammatiken enthalten, die mir selbst als eher einfach erschienen und daher z. B. als „Punktebringer“ in Klausuren verwendet wurden.

Im direkt anschließenden Abschnitt folgen dann die etwas komplexeren Grammatiken, die entweder einige Spezialfälle enthalten oder einfach nur umfangreich sind. Diese Grammatiken tauchen in Klausuren in der Regel nur mit einer vereinfachten Aufgabenstellung auf (z. B. Analyse eines Wortes mit vorgegebener Syntaxanalysetabelle).

Der letzte Abschnitt widmet sich schließlich den „real existierenden“ Grammatiken. Hier sind Grammatiken aufgelistet, die nicht frei erfunden sind, sondern wirklich so (oder so ähnlich) „in der freien Natur“ vorkommen.

Bitte beim Vergleich der eigenen mit der abgedruckten Lösung beachten, dass die Zustände in den Zustandsübergangsgraphen (und damit die Zeilennummern in den Syntaxanalysetabellen) eventuell anders numeriert wurden.

A.1 Einfache Grammatiken

Aufgabe 1.1
Erstellen Sie Zustandsübergangsgraph und Syntaxanalysetabelle nach dem LR(0)-Verfahren. Woran erkennt man, dass es sich um eine LR(0)-Grammatik handelt? Erstellen Sie eine SLR(1)-, eine LR(1)- und eine LALR(1)-Syntaxanalysetabelle und zeigen Sie, dass die drei Tabellen den gleichen Inhalt haben.

Diese Grammatik ist ein Abfallprodukt einer Klausurvorbereitung. Sie ist eine von vielen Grammatiken, die es nicht in die Klausur geschafft haben.

$$
\begin{array}{llcl}
(1) & S & \to & aB \\
(2) & B & \to & bA \\
(3) & B & \to & a \\
(4) & A & \to & S
\end{array}
$$

Lösung 1.1
Die LR(0)-Syntaxanalysetabelle ist im folgenden abgebildet. Die Grammatik ist vom Typ LR(0), da es im Zustandsübergangsgraphen (und daher auch in der Tabelle) keine Konflikte gibt. Man beachte noch einmal in der Syntaxanalysetabelle, dass beim LR(0)-Verfahren Reduktionen immer jeweils eine ganze Zeile in der Aktionstabelle (die Spalten der Teminalsymbole) einnehmen. Mit anderen Worten: es wird immer reduziert unabhängig vom nächsten Terminalsymbol.

	a	b	\$	A	B	S
0	s2					1
1			acc			
2	s3	s5			4	
3	r3	r3	r3			
4	r1	r1	r1			
5	s2			6		7
6	r2	r2	r2			
7	r4	r4	r4			

Die SLR(1)/LALR(1)/LR(1)-Syntaxanalysetabelle sieht folgendermaßen aus:

	a	b	\$	A	B	S
0	s2					1
1			acc			
2	s3	s5			4	
3			r3			
4			r1			
5	s2			6		7
6			r2			
7			r4			

Aufgabe 1.2
Erstellen Sie Zustandsübergangsgraphen und Syntaxanalysetabelle nach dem SLR(1)-Verfahren.

Diese Grammatik habe ich beim Durchsehen alter Praktikumsvorbereitungen gefunden. Sie ist zwar nicht sonderlich spektakulär, aber dennoch eine nette Übung für Zwischendurch.

$$
\begin{array}{llll}
\text{(1)} & X & \rightarrow & XaY \\
\text{(2)} & X & \rightarrow & Y \\
\text{(3)} & Y & \rightarrow & b
\end{array}
$$

Lösung 1.2
Die Syntaxanalysetabelle fällt erwartungsgemäß sehr klein aus:

	a	b	\$	X	Y
0		s3		1	5
1	s2		acc		
2		s3			4
3	r3		r3		
4	r1		r1		
5	r2		r2		

Aufgabe 1.3
Erstellen Sie einen LR(0)-Zustandsübergangsgraphen. Zeigen Sie, dass die Grammatik nicht vom Typ LR(0) ist und erstellen Sie dann eine Syntaxanalysetabelle nach dem SLR(1)-Verfahren.

Bei dieser Grammatik handelt es sich um Grammatik G_1 aus dem Text. Der geneigte Leser kann diese Aufgabe wahlweise als Test für das Verständnis der benötigten Algorithmen oder als Herausforderung an sein Erinnerungsvermögen betrachten.

$$
\begin{array}{llll}
\text{(1)} & S & \rightarrow & Sb \\
\text{(2)} & S & \rightarrow & bAa \\
\text{(3)} & A & \rightarrow & aSc \\
\text{(4)} & A & \rightarrow & a \\
\text{(5)} & A & \rightarrow & aSb
\end{array}
$$

Lösung 1.3
Im LR(0)-Graphen sollten zwei Konflikte sichtbar sein: ein Reduce/Reduce-Konflikt und ein Shift/Reduce-Konflikt. Damit kann die Grammatik nicht vom Typ LR(0) sein. Die beiden Konflikte lassen sich jedoch bereits mit FOLLOW-Mengen (SLR(1)-Verfahren) auflösen. Als Fleißarbeit kann man sich jetzt noch die Mühe machen und eine LALR(1)-Syntaxanalysetabelle aufstellen. Im Vergleich mit der SLR(1)-

Syntaxanalysetabelle kann man dann gut erkennen, wie die Anzahl der unnötigen Reduktionen verringert wird.

Die SLR(1)-Syntaxanalysetabelle sieht folgendermaßen aus:

	a	b	c	\$	A	S
0		s3				1
1		s2		acc		
2		r1	r1	r1		
3	s6				4	
4	s5					
5		r2	r2	r2		
6	r4	s3				7
7		s9	s8			
8	r3					
9	r5	r1	r1	r1		

Aufgabe 1.4

Erstellen Sie Zustandsübergangsgraphen und Syntaxanalysetabelle nach dem SLR(1)-Verfahren. Woran erkennt man, dass es sich um keine LR(0)-Grammatik handelt? Zeigen Sie, dass die LR(1)- und die LALR(1)-Syntaxanalysetabellen den gleichen Inhalt haben wie die SLR(1)-Syntaxanalysetabelle.

Diese Grammatik ist eine Abwandlung einer Klausuraufgabe. In der Originalaufgabe lautete die vierte Regel ($B \rightarrow d$). Außer der obigen Aufgabenstellung sollte in der Klausur auch noch das Wort bdc mit Hilfe der selbst erstellten Tabelle analysiert werden.

$$
\begin{array}{lrcl}
\text{(1)} & S & \rightarrow & bAc \\
\text{(2)} & S & \rightarrow & bBa \\
\text{(3)} & A & \rightarrow & d \\
\text{(4)} & B & \rightarrow & \varepsilon
\end{array}
$$

Lösung 1.4

Die SLR(1)/LALR(1)/LR(1)-Syntaxanalysetabelle ist im folgenden abgebildet. Dass es sich bei der Grammatik um keine LR(0)-Grammatik handelt, kann man in Zei-

le 2 der Tabelle gut sehen. Hier würde es beim Terminal d zu einem Shift/Reduce-Konflikt kommen.

	a	b	c	d	\$	A	B	S
0		s2						1
1					acc			
2	r4			s3		4	5	
3			r3					
4			s6					
5	s7							
6					r1			
7					r2			

Aufgabe 1.5

Erstellen Sie einen LR(0)-Zustandsübergangsgraphen und eine SLR(1)-Syntaxanalysetabelle. Benennen Sie alle konfliktbehafteten Zustände, in denen ein LR(0)-Parser versagen würde. Analysieren Sie mit Hilfe der von Ihnen erstellten Syntaxanalysetabelle das Wort $aabab$.

Diese Aufgabenstellung habe ich 1:1 aus einer Klausur übernommen. Es gab für diese Aufgabe damals 14 Punkte, was bei einer Klausurgesamtpunktzahl von 100 und einer Gesamtdauer von 3 Stunden bedeutete, dass man sich für die Aufgabe ungefähr 25 Minuten (und 12 Sekunden) Zeit lassen konnte. Es gab allerdings zugegebenermaßen in der gleichen Klausur andere Aufgaben, bei denen Zeit nötiger war…

$$
\begin{array}{lrcl}
(1) & S & \to & aXb \\
(2) & S & \to & aYa \\
(3) & X & \to & S \\
(4) & X & \to & b \\
(5) & Y & \to & b
\end{array}
$$

Lösung 1.5

Im folgenden ist die SLR(1)-Syntaxanalysetabelle zu sehen; der konfliktbehaftete Zustand befindet sich in Zeile 8.

	a	b	$	X	Y	S
0	s2					1
1			acc			
2	s2	s8		3	5	7
3		s4				
4		r1	r1			
5	s6					
6		r2	r2			
7		r3				
8	r5	r4				

Aufgabe 1.6

Erstellen Sie einen LR(0)-Zustandsübergangsgraphen. Fügen Sie in diesen nachträglich die LOOKAHEAD-Mengen ein (keine Angst: der LR(1)-Graph besitzt bei dieser Grammatik genau dieselben Zustände) und zeigen Sie, dass LOOKAHEAD-Mengen den einzigen bestehenden Konflikt beseitigen.

Diese Grammatik habe ich dem Skript zur Vorlesung von Prof. Bothe entnommen. Sie ist ein sehr einfaches Beispiel für die Funktionsweise von LOOKAHEAD-Mengen.

$$
\begin{aligned}
&\text{(1)} \quad S \rightarrow X + X \\
&\text{(2)} \quad S \rightarrow Y * Y \\
&\text{(3)} \quad X \rightarrow a \\
&\text{(4)} \quad Y \rightarrow a
\end{aligned}
$$

Lösung 1.6

Die resultierende LR(1)/LALR(1)-Syntaxanalysetabelle ist im folgenden abgebildet. Im LR(0)-Zustandsübergangsgraphen gibt in einem Zustand (in meinem Fall Zustand 2) einen Konflikt zwischen (X→a) und (Y→a). Dieser wird durch die LOOKAHEAD-Mengen dahingehend gelöst, dass im Falle eines folgenden „+“ die

erste und im Falle eines folgenden „$*$“ die zweite Regel angewendet wird.

	a	$+$	$*$	$	X	Y	S
0	s2				3	6	1
1				acc			
2		r3	r4				
3		s4					
4	s8				5		
5				r1			
6			s7				
7	s9					10	
8				r3			
9				r4			
10				r2			

Aufgabe 1.7

Erstellen Sie einen LR(1)-Zustandsübergangsgraphen und eine LR(1)-Syntaxanalysetabelle. Warum ist die Grammatik vom Typ LR(1)/LALR(1)? Ist sie auch vom Typ LR(0)/SLR(1)?

Diese Grammatik ist wieder eine Wegwerfgrammatik einer Klausurvorbereitung.

$$
\begin{array}{llll}
(1) & S & \to & Aa \\
(2) & S & \to & bBa \\
(3) & A & \to & c \\
(4) & B & \to & c
\end{array}
$$

Lösung 1.7

Im folgenden ist die LR(1)-Syntaxanalysetabelle abgebildet. Man kann der Syntaxanalysetabelle leicht entnehmen, dass die Grammatik sogar LR(0) ist.

	a	b	c	$	A	B	S
0		s5	s4		2		1
1				acc			
2	s3						
3				r1			
4	r3						
5			s6			7	
6	r4						
7	s8						
8				r2			

A.2 Komplexere Grammatiken

Aufgabe 2.1
Erstellen Sie Zustandsübergangsgraphen und Syntaxanalysetabelle nach dem LALR(1)-Verfahren. Diese Grammatik hat eine kleine Gemeinheit implementiert: Wenn man den LR(1)-Zustandsübergangsgraphen aufbaut, ist es an einer Stelle sehr verlockend (aber leider falsch), einen Pfeil zu einem vorherigen Zustand einzubauen – bei diesem Zustand bitte auf die LOOKAHEAD-Mengen achten!

Diese Grammatik tauchte in einer Klausur als Punktebringer auf, allerdings mit einer stark vereinfachten Aufgabenstellung. In der Originalaufgabe wurde bereits die Syntaxanalysetabelle gegeben, und man sollte mit ihrer Hilfe das Wort $abaaba$ analysieren. In dieser Aufgabe haben übrigens weit über 90% der Studenten volle Punktzahl erhalten.

$$
\begin{array}{llll}
\text{(1)} & S & \rightarrow & aBa \\
\text{(2)} & B & \rightarrow & bAb \\
\text{(3)} & B & \rightarrow & \varepsilon \\
\text{(4)} & A & \rightarrow & S
\end{array}
$$

Lösung 2.1
Die LALR(1)-Syntaxanalysetabelle sieht folgendermaßen aus:

	a	b	$	A	B	S
0	s2					1
1			acc			
2	r3	s5			3	
3	s4					
4		r1	r1			
5	s2			7		6
6		r4				
7		s8				
8	r2					

Aufgabe 2.2
Erstellen Sie Zustandsübergangsgraphen und Syntaxanalysetabelle nach dem LR(1)- und nach dem LALR(1)-Verfahren.

Auch diese Grammatik wurde in einer Klausur als Punktebringer verwendet. Es wurde die gesamte Syntaxanalysetabelle vorgegeben, und man sollte das Wort $abaabcaabc$ analysieren.

Beim Aufstellen des Graphen wird einem ziemlich schnell klar, warum diese Grammatik in die Kategorie der etwas komplizierteren eingeordnet wurde. Insbesondere die Hüllenbildung kann bei einigen LOOKAHEAD-Mengen zum Alptraum werden.

Wer es ganz schwierig haben möchte, kann mal versuchen, direkt den LALR(1)-Graphen herzustellen (also ohne vorher den LR(1)-Graphen zu erzeugen).

$$\begin{array}{lrcl}
(1) & S & \rightarrow & aXab \\
(2) & S & \rightarrow & Y \\
(3) & X & \rightarrow & bYa \\
(4) & X & \rightarrow & \varepsilon \\
(5) & Y & \rightarrow & Sc
\end{array}$$

Lösung 2.2

Da es diese Grammatik wirklich in sich hat, sind im folgenden nicht nur die LR(1)- und LALR(1)-Syntaxanalysetabellen dargestellt. In den Abbildungen A.1 und A.2 sind auch noch die Zustandsübergangsgraphen zu sehen.

Die LR(1)-Syntaxanalysetabelle sieht folgendermaßen aus:

	a	b	c	$	X	Y	S
0	s4					3	1
1			s2	acc			
2			r5	r5			
3			r2	r2			
4	r4	s8			5		
5	s6						
6		s7					
7			r1	r1			
8	s13					9	11
9	s10		r2				
10	r3						
11			s12				
12	r5		r5				
13	r4	s8			14		
14	s15						
15		s16					
16			r1				

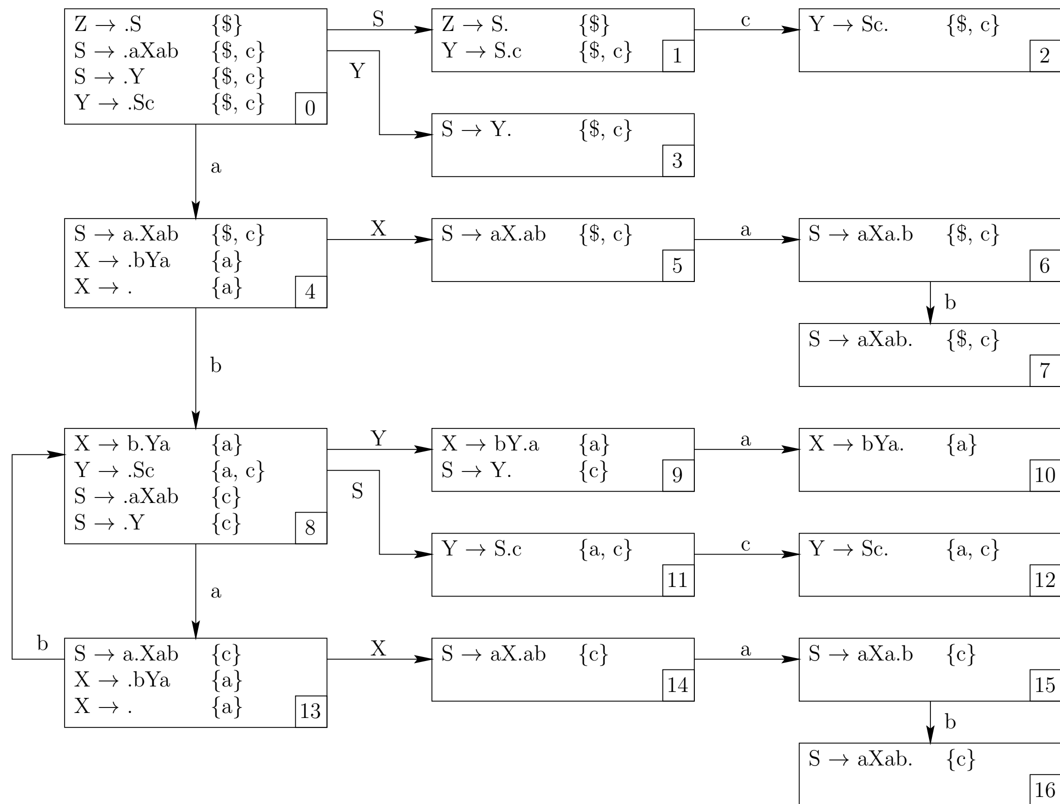

Abbildung A.1: LR(1)-Zustandsübergangsgraph für Aufgabe 2.2

In der LALR(1)-Syntaxanalysetabelle sind fünf Zustände weniger enthalten (siehe dazu auch den Zustandsübergangsgraphen):

	a	b	c	$	X	Y	S
0	s4					3	1
1			s2	acc			
2	r5		r5	r5			
3			r2	r2			
4	r4	s8			5		
5	s6						
6		s7					
7			r1	r1			
8	s4					9	11
9	s10		r2				
10	r3						
11			s2				

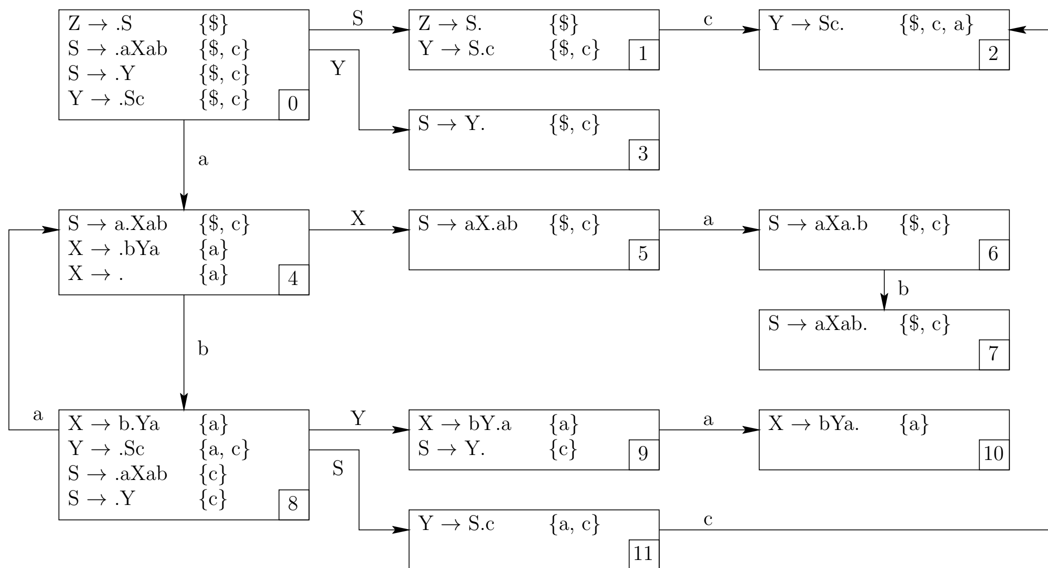

Abbildung A.2: LALR(1)-Zustandsübergangsgraph für Aufgabe 2.2

Aufgabe 2.3

Erstellen Sie einen LR(1)-Zustandsübergangsgraphen und daraus eine LR(1)-Syntaxanalysetabelle. Zeigen Sie, dass die Grammatik zwar vom Typ LR(1) aber nicht vom Typ LALR(1) ist.

Diese Grammatik habe ich vom Zustandsübergangsgraphen rückwärts entwickelt, da ich unbedingt einmal Studenten mit einer Nicht-LALR(1)-Aber-LR(1)-Grammatik konfrontieren wollte. Die Aufgabe tauchte dann auch in einer Klausur auf, fand dort allerdings weniger Anklang...

$$
\begin{array}{llcl}
\text{(1)} & S & \rightarrow & aXa \\
\text{(2)} & S & \rightarrow & aYb \\
\text{(3)} & S & \rightarrow & bXb \\
\text{(4)} & S & \rightarrow & bYa \\
\text{(5)} & X & \rightarrow & c \\
\text{(6)} & Y & \rightarrow & c
\end{array}
$$

Lösung 2.3

Die LR(1)-Syntaxanalysetabelle ist im folgenden abgebildet. Im konstruierten Zustandsübergangsgraphen sollten einem genau zwei Zustände ins Auge springen, die man beim LALR(1)-Verfahren zusammenlegen würde. Bei dieser Zusammenlegung entsteht ein Reduce/Reduce-Konflikt, so dass diese Grammatik wirklich vom Typ

LR(1), aber nicht vom Typ LALR(1) ist.

	a	b	c	\$	X	Y	S
0	s2	s3					1
1				acc			
2			s8		4	6	
3			s13		9	11	
4	s5						
5				r1			
6		s7					
7				r2			
8	r5	r6					
9		s10					
10				r3			
11	s12						
12				r4			
13	r6	r5					

A.3 Real existierende Grammatiken

Aufgabe 3.1

Erstellen Sie einen LR(1)-Zustandsübergangsgraphen und daraus eine LR(1)-Syntaxanalysetabelle.

Diese Grammatik ist die vermutlich am häufigsten in der Compilerbauliteratur vorkommende. Es handelt sich um eine Ausdrucksgrammatik (Expression, Term, Faktor). Das besondere an dieser Grammatik ist, dass die damit erstellten Ableitungsbäume bereits die Vorrangsregelung (Punkt- vor Strichrechnung) automatisch enthalten. Die Grammatik ist in vielen realen Programmiersprachengrammatiken als Teilmenge enthalten.

$$
\begin{array}{lrcl}
(1) & E & \rightarrow & E + T \\
(2) & E & \rightarrow & T \\
(3) & T & \rightarrow & T * F \\
(4) & T & \rightarrow & F \\
(5) & F & \rightarrow & id \\
(6) & F & \rightarrow & (E)
\end{array}
$$

Lösung 3.1

Da diese Grammatik wirklich *sehr* fehleranfällig ist, ist in Abbildung A.3 der LR(1)-Zustandsübergangsgraph zu sehen (der übrigens drei Leute zwei Tage lang beschäf-

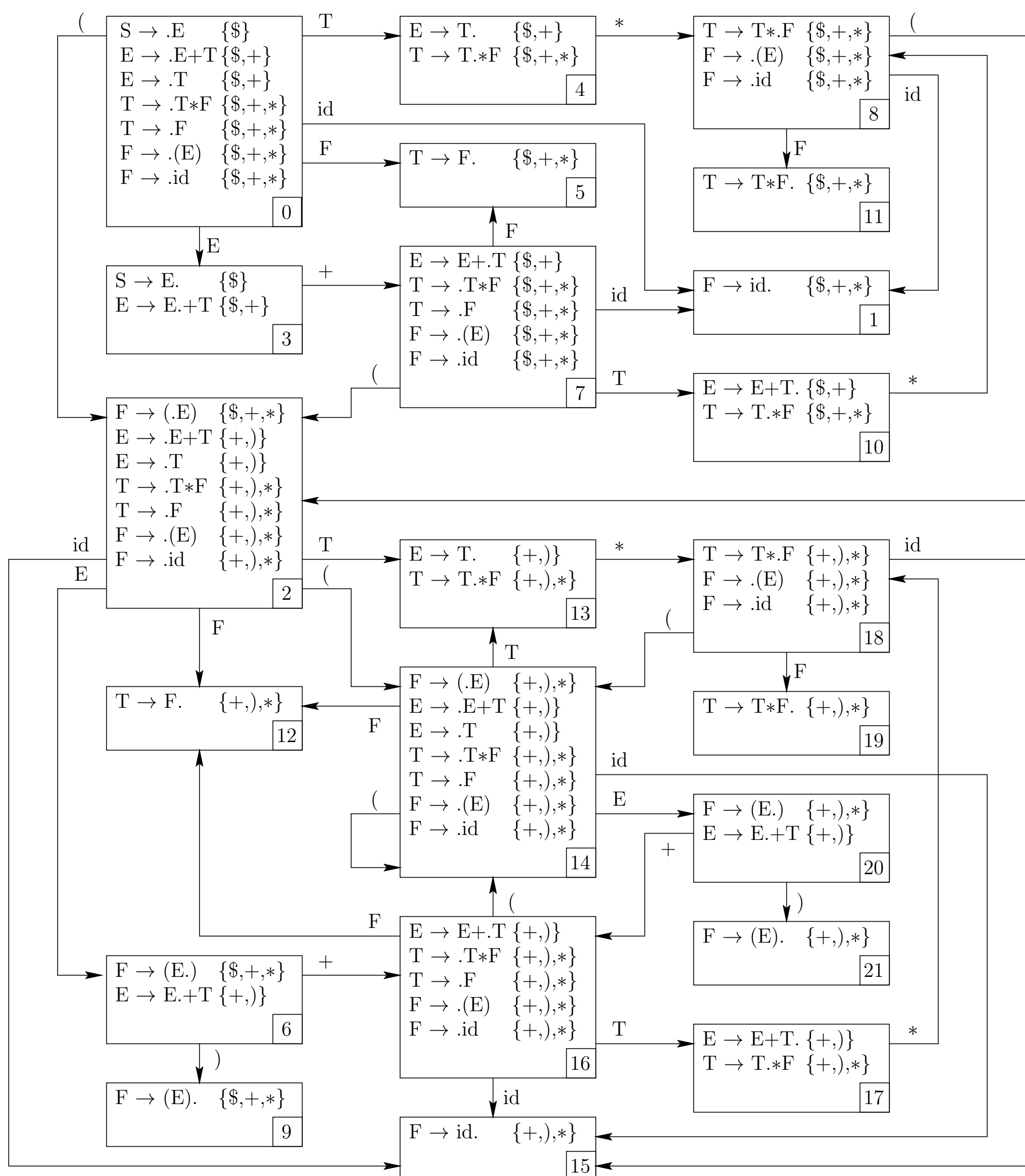

Abbildung A.3: LR(1)-Zustandsübergangsgraph für Aufgabe 3.1

tigt hat). Die daraus resultierende Syntaxanalysetabelle (deren Erstellung nicht weniger aufwendig war) sieht folgendermaßen aus:

	$+$	$*$	(	)	id	\$	E	T	F	S
0			s2		s1		3	4	5	
1	r5	r5				r5				
2			s14		s15		6	13	12	
3	s7					acc				
4	r2	s8				r2				
5	r4	r4				r4				
6	s16			s9						
7			s2		s1			10	5	
8			s2		s1				11	
9	r6	r6				r6				
10	r1	s8				r1				
11	r3	r3				r3				
12	r4	r4		r4						
13	r2	s18		r2						
14			s14		s15		20	13	12	
15	r5	r5		r5						
16			s14		s15			17	12	
17	r1	s18		r1						
18			s14		s15				19	
19	r3	r3		r3						
20	s16			s21						
21	r6	r6		r6						

ANHANG B

LITERATURVERZEICHNIS

[ASU99] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullmann: *Compilerbau Teil 1 und 2, 2., durchgesehene Auflage*, Oldenbourg Wissenschaftsverlag, München 1999

Aufgrund des Einbandes als „Drachenbuch“ bekannt. Die Bibel für jeden Compilerbauer – enthält prinzipiell alles, was man wissen muss. Leider ist das Buch als Einstieg in die Materie nur bedingt zu empfehlen, da zum einen die Reihenfolge der Themen an mehreren Stellen pädagogisch eher suboptimal ist und zum anderen viele Algorithmen und Sachverhalte unnötig kompliziert dargestellt werden.

[Bot97] Klaus Bothe: *Skript zur Vorlesung „Praktische Informatik III – Compilerbau“ WS1997/98*, notiert von Ralf Heese, 1997

Durchaus lesenswertes Skript zur Compilerbauvorlesung von Prof. Bothe, das allerdings vermutlich nur von Hörern der Veranstaltung wirklich nachvollziehbar ist.

[Hol90] Allen I. Holub: *Compiler Design in C*, Prentice-Hall, New Jersey, 1990

Sehr gutes Compilerbaubuch, das leider nicht mehr im Handel erhältlich ist. Besonders hervorzuheben sind vor allem zwei Tatsachen: Holub verwendet auch durchaus große Beispiele (Zustandsübergangsgraphen über zwei Buchseiten, ...) und er geht sehr detailliert auf Spezialfälle von Algorithmen ein.

[Par96] Terence Parr and Russel Quong: *LL and LR Translator Need k*, SIGPLAN Notices Volume 31 #2, Februar 1996
online unter: `http://www.antlr2.org/article/needlook.html`

Netter Artikel zur Fragestellung, ob bzw. warum LR(k) für $k \geq 2$ sinnvoll ist. Der erste Autor ist übrigens der Hauptprogrammierer von `ANTLR` und

gleichzeitig ein Beweis dafür, dass Compilerbauer Humor haben (man betrachte seine Homepage).

[Sch97] Uwe Schöning: *Theoretische Informatik – kurzgefaßt*, Spektrum Akademischer Verlag, Heidelberg; Berlin 1997

Das prägnanteste (mir bekannte) Buch zur theoretischen Informatik. Eignet sich sehr gut zum Selbststudium und enthält wirklich gute Beispiele für jeden behandelten Aspekt.

[Spe03] Michael Sperber, Peter Thiemann: *Systematic Compiler Construction; 2003*

Ein Skript, das mir sehr gut gefallen hat, da es zum einen sehr ausführlich und zum anderen verständlich ist.

[Wil97] Reinhard Wilhelm, Dieter Maurer: *Übersetzerbau, 2. überarbeitete und erweiterte Auflage; Springer-Verlag Berlin, Heidelberg 2003*

Das mit Abstand theoretischste Werk, das ich im Bereich Compilerbau kenne. Als Einstieg in die Materie meiner Meinung nach eher ungeeignet, aber ein sehr gutes Nachschlagewerk für alle, die es ganz genau wissen wollen.

DANKSAGUNG

Hiermit möchte ich mich noch einmal bei allen Personen bedanken, die mir bei der Verbesserung dieses Schriftstückes geholfen haben.

Da wären zum einen die eifrigen Probeleser/Korrektoren/Ideengeber:

- Prof. Dr. Klaus Bothe
- Dr. Klaus Ahrens
- Prof. Dr. Lothar Schmitz (UniBw München)
- Stefan Kirchner
- Constanze Kurz
- Julia Böttcher
- Konrad Voigt
- Glenn Schütze
- Martin Stigge

und zum anderen die Fehlerfinder (in chronologischer Reihenfolge):

- Jörg Peters (Uni Karlsruhe)
- Aylin Aydoğan
- Ayşegül Gündoğan
- Laura Obretin
- Ralf Heese

- Elena Filatova
- Martin Sommerfeld
- Elżbieta Jasińska
- Frank Butzek
- Matthias Kubisch
- Kristian Klaus
- Kornelius Kalnbach
- Lukas Moll
- Lars Sadau
- Alexander Borisov
- Kai Stierand (Uni Hamburg)
- Richard Müller
- Robert Müller
- Moritz Kaltofen
- Peter Hinkel (FSU Jena)
- Andreas Rentschler (Uni Karlsruhe)
- Mario Lehmann
- Wolfgang Nádasi-Donner
- Frederic Beister
- Jochen Taeschner
- Manuel Mohr (Karlsruher Institut für Technologie)
- Johannes Bechberger (Karlsruher Institut für Technologie)
- sowie einer, der nicht namentlich erwähnt werden wollte

Anmerkung: Jeder, der einen Fehler in diesem Skript findet, wird in kommenden Versionen ebenfalls auf dieser Seite verewigt.