

Formale Sprachen und Automaten

Unterlagen für die Vorlesung
an der DHBW Karlsruhe
im Wintersemester 2012

vorläufige Fassung vom 30. Oktober 2012

Dr. Thomas Worsch
Fakultät für Informatik
Karlsruher Institut für Technologie

Inhaltsverzeichnis

| | |
|---|-----------|
| 1 Grundlagen | 3 |
| 1.1 Zeichen, Alphabete, Wörter, Sprachen | 3 |
| 1.2 „Verarbeitung“ formaler Sprachen | 5 |
| 1.3 Ausblick | 5 |
| 2 Typ 3 | 7 |
| 2.1 Deterministische endliche Automaten | 7 |
| 2.2 Reguläre Ausdrücke | 13 |
| 2.3 Grammatiken | 17 |
| 2.4 Typ-3-Grammatiken | 20 |
| 2.5 Nichtdeterministische endliche Automaten | 22 |
| 2.6 Zustandsminimierung endlicher Automaten | 29 |
| 3 Regular expressions | 35 |
| 3.1 Anwendungsszenario für regular expressions | 35 |
| 3.2 Regular expressions bei egrep | 35 |
| 3.3 Bequemere Notation für reguläre Ausdrücke | 37 |
| 3.4 Metazeichen in Zeichenklassen | 39 |
| 3.5 Anker | 40 |
| 3.6 Gruppierungen und Rückwärtsverweise | 41 |
| 3.7 Verwendung von Metazeichen als reguläre Zeichen | 44 |
| 4 Typ 2 | 45 |
| 4.1 Kellerautomaten | 45 |
| 4.2 Kontextfreie Grammatiken | 49 |
| 4.3 Zusammenhang zwischen Kellerautomaten und Typ-2-Grammatiken | 51 |
| 5 Typ 1 und Typ 0 | 57 |
| 5.1 Kontextsensitive Grammatiken | 57 |
| 5.2 Turingmaschinen | 57 |
| 5.3 Jenseits von Typ 0 | 64 |
| 5.4 Ausblick | 67 |
| 6 Syntaxanalyse | 68 |
| 6.1 Einleitung | 68 |
| 6.2 Lexikalische Analyse | 70 |
| 6.3 Lexer-Generatoren | 71 |
| 6.4 Syntaktische Analyse | 76 |
| 6.5 Der Algorithmus von Cocke, Younger und Kasami | 76 |
| 6.5.1 Chomsky-Normalform für kontextfreie Grammatiken | 76 |
| 6.5.2 Der Algorithmus von Cocke, Younger und Kasami | 80 |
| 6.6 Der Algorithmus von Earley | 83 |
| 6.7 LR Parsing | 88 |
| 6.8 Nach der Syntaxanalyse | 94 |

1 Grundlagen

1.1 Zeichen, Alphabete, Wörter, Sprachen

1.1 Beispiel. Ein Java-Übersetzer liest als Eingabe eine Folge von Zeichen, von denen der Benutzer „verspricht“, dass sie alle aus einer gewissen Menge A (einem so genannten Zeichensatz, bei Java *mehr* als ASCII) stammen. Diese Menge ist *endlich* und man nennt sie auch ein *Alphabet*.

Ein Java-Programm ist eine „syntaktisch korrekte“ Zeichenfolge (wie auch immer das definiert sein mag).

1.2 Definition. Ein *Alphabet* ist eine endliche Menge von Zeichen. Ein *Wort* über einem Alphabet A ist eine Folge von Zeichen aus A . Die *Länge* $|w|$ eines Wortes w ist die Anzahl der Zeichen, aus denen es besteht.

Das *leere Wort* ε besteht aus 0 Zeichen: $|\varepsilon| = 0$. Damit man es trotzdem sieht, schreibt man ε . (Mit diesem ε ist *nicht* eines der Zeichen in A gemeint!)

Für die Menge aller Wörter über einem Alphabet A schreiben wir A^* .

Eine beliebige Teilmenge $L \subseteq A^*$ von Wörtern heißt eine *formale Sprache* über A .

Man beachte, in welcher Form hier das Wort *Wort* benutzt wird: eine beliebige Folge von Zeichen aus dem zu Grunde liegenden Alphabet. In diesem Sinne ist ein Java-Programm *ein* Wort über dem für solche Programme erlaubten Alphabet (und nicht mehrere Wörter).

1.3 Definition. Es sei A ein beliebiges Alphabet.

- Für zwei Wörter $w_1, w_2 \in A^*$ mit $w_1 = a_1 \cdots a_k$ mit $a_1, \dots, a_k \in A$ und $w_2 = b_1 \cdots b_\ell$ mit $b_1, \dots, b_\ell \in A$ ist ihre *Konkatenation* $w_1 \cdot w_2 = w_1 w_2 = a_1 \cdots a_k b_1 \cdots b_\ell$.
- Für das leere Wort gilt: $\varepsilon \cdot w = w = w \cdot \varepsilon$.
- Die *Potenzen* eines Wortes $w \in A^*$ sind so definiert: $w^0 = \varepsilon$ und $w^{k+1} = w^k w$ für alle $k \in \mathbb{N}_0$.

Wie auch von Zahlen gewohnt gilt für Wörter: $(w_1 w_2) w_3 = w_1 (w_2 w_3)$. Das ist das so genannte Assoziativgesetz. Aber Vorsicht: Nicht alles, was man von Zahlen gewohnt ist, gilt auch für Wörter:

1.4 Beispiele. $00 \cdot 10 = 0010$, aber $10 \cdot 00 = 1000$. Die Konkatenation von Wörtern ist also *nicht kommutativ*.

$$w^1 = w^{0+1} = w^0 \cdot w = \varepsilon \cdot w = w$$

$$0^5 = 00000, 0^3 1^3 = 000111, 01^3 = 0111 \text{ und } (01)^3 = 010101.$$

$$\text{Für alle } k \in \mathbb{N}_0 \text{ ist } \varepsilon^k = \varepsilon.$$

1.5 Man mache sich klar, dass für alle Wörter $w_1, w_2 \in A^*$ gilt: $|w_1 w_2| = |w_1| + |w_2|$. Und für alle $k \in \mathbb{N}_0$ ist daher $|w^k| = k|w|$.

In unseren Beispielen wird das Eingabealphabet meist $\{0, 1\}$ oder $\{a, b\}$ oder etwas ähnlich einfaches sein. Für eine Programmiersprache ist es oft ASCII oder Unicode oder ...

Die Operationen, die wir eben für einzelne Wörter definiert haben, kann man auf Mengen von Wörtern, also formale Sprachen „ausweiten“.

1.6 Definition. Für formale Sprachen $L_1, L_2, L \subseteq A^*$ definiert man:

- Die *Konkatenation* oder das *Produkt* zweier Sprachen ist $L_1 \cdot L_2 = L_1 L_2 = \{w_1 w_2 \mid w_1 \in L_1 \wedge w_2 \in L_2\}$.
- Die *Potenzen* einer formalen Sprache L sind so definiert: $L^0 = \{\varepsilon\}$ und für alle $k \in \mathbb{N}_0$ ist $L^{k+1} = L^k L$.

1.7 Beispiele. $\{01, 1\} \cdot \{000, 01\} = \{01000, 0101, 1000, 101\}$
 $\{010, 11\}^2 = \{010010, 01011, 11010, 1111\}$
 $\{0, 1\}^3 = \{000, 001, 010, 011, 100, 101, 110, 111\}$
 $A^k = \{w \in A^* \mid |w| = k\}$

1.8 Bei der Konkatenation gilt insbesondere:

$$\begin{aligned} \{\varepsilon\}L &= \{w_1 w_2 \mid w_1 \in \{\varepsilon\} \wedge w_2 \in L\} \\ &= \{w_1 w_2 \mid w_1 = \varepsilon \wedge w_2 \in L\} \\ &= \{\varepsilon w_2 \mid w_2 \in L\} \\ &= \{w_2 \mid w_2 \in L\} = L \end{aligned}$$

und analog $L\{\varepsilon\} = L$.

Man beachte auch, dass die formale Sprache $\{\varepsilon\}$ *nicht* die leere Menge ist, sondern eine Menge, die genau ein Element enthält.

1.9 Definition. Der ε -freie Konkatenationsabschluss L^+ einer formalen Sprache L und der Konkatenationsabschluss L^* von L sind so definiert:

$$\begin{aligned} L^+ &= \bigcup_{k=1} L^k \\ L^* &= \bigcup_{k=0} L^k = L^0 \cup L^+ = \{\varepsilon\} \cup L^+ \end{aligned}$$

Der $*$ heißt nach dem amerikanischen Mathematiker Stephen Kleene (1909–1994) auch *Kleene-Operator* oder *Kleene-Stern*. Diese Bezeichnung geht zurück auf die Arbeit von ?.

1.10 L^+ ist die Menge aller Wörter, die man als Produkt von einem oder mehreren Wörtern aus L schreiben kann.

Man beachte, dass auch in L^+ schon ε enthalten sein kann, nämlich dann, wenn $\varepsilon \in L$ ist.

1.11 Beispiel. Die Menge der syntaktisch korrekten Java-Programme ist eine formale Sprache $L_{\text{Java}} \subset A^*$ über dem Alphabet $A = \{a, \dots, z, A, \dots, Z, +, -, *, \dots\}$.

Ein Java-Übersetzer muss unter anderem überprüfen, ob eine Eingabezeichenkette ein syntaktisch korrektes Java-Programm ist (und es dann gegebenenfalls übersetzen) oder nicht (und dann gegebenenfalls möglichst mitteilen, an welcher Stelle die Eingabezeichenkette nicht den vorgeschriebenen Regeln genügt). Mit anderen Worten muss ein Übersetzer für jedes $w \in A^*$ feststellen können, ob $w \in L_{\text{Java}}$ ist oder nicht.

1.2 „Verarbeitung“ formaler Sprachen

Wir haben schon erwähnt, dass man z. B. die Menge der syntaktisch korrekten Java-Programme als formale Sprache auffassen kann. Damit stellen sich die unter anderem die folgenden (verwandten) Fragen:

1. Wie spezifiziert man (präzise), welche Zeichenfolgen syntaktisch korrekte Java-Programme sind?
2. Wie kann z. B. ein Algorithmus aussehen, der überprüft, ob die Eingabe ein syntaktisch korrektes Java-Programm ist?

Verallgemeinert auf beliebige formale Sprachen (denken Sie z. B. an verschiedene Programmiersprachen) lauten die Fragen dann:

1. Wie spezifiziert man eine formale Sprache?
2. Inwieweit kann man Algorithmen angeben, die für Eingabewörter entscheiden, ob sie zu einer vorher spezifizierten formalen Sprache gehören?

Ist man erst einmal so weit, dann ergeben sich als weitere Fragen:

3. Kann man den Entscheidungsalgorithmus womöglich automatisch aus der Spezifikation der formalen Sprachen erzeugen?
4. Gibt es vielleicht Teilklassen (besonders „einfacher“) formaler Sprachen, für die gewisse besonders „schöne“ Spezifikations- und Erkennungsmethoden benutzt werden können?

Natürlich kann man die Verarbeitung formaler Sprachen, also zum Beispiel die Entscheidung über die Zugehörigkeit eines Wortes zu einer formalen Sprache, durch Angabe von Algorithmen in irgendeiner gängigen Programmiersprache beschreiben. Es hat sich aber herausgestellt, dass man für spezielle Teilklassen formaler Sprachen mit Algorithmen einer bestimmten speziellen einfachen Struktur auskommt.

Eine übliche Darstellung ist die in Form gewisser Automaten. Darunter hat man sich typischerweise eine *Kontrolleinheit* (sozusagen mit einem fest verdrahteten Programm) vorzustellen, die auf einen (*Daten-*)*Speicher* zugreifen kann. Die verschiedenen Varianten unterscheiden sich insbesondere bei der Größe des Speichers und bei der Art der möglichen Speicherzugriffe.

Je nach „Kompliziertheit“ der formalen Sprache braucht man unterschiedlich „mächtige“ Automaten für die Erkennung.

1.3 Ausblick

Gegenstand der folgenden Kapitel sind Antworten auf die zuletzt angesprochenen Fragen.

Wir werden die grobe Einteilung der nach dem Linguisten Noam Chomsky so genannten *Chomsky-Hierarchie* zu Grunde legen. Danach unterscheidet man Typ-0-, Typ-1-, Typ-2- und Typ-3-Sprachen (mit abnehmendem „Schwierigkeitsgrad“).

In der Vorlesung wird das Schwergewicht auf den beiden Klassen einfacherer Sprachen, also den Typ-3- und den Typ-2-Sprachen (und Teilklassen hiervon) liegen, weil sie z. B. im Übersetzerbau (siehe Kapitel 6), aber auch anderswo immer wieder von Nutzen sind.

Typ-1- und Typ-0-Sprachen, sowie Sprachen, die noch „komplizierter“ als Typ-0-Sprachen sind, werden kürzer abgehandelt. Die fundamentalen Tatsachen werden aber auch hier besprochen.

Die Vorlesung beginnt mit der einfachsten Variante, den Typ-3-Sprachen und arbeitet sich dann „nach oben“ (das ist in diesem Zusammenhang in Richtung Typ-0) weiter vor.

Im zweiten Teil der Vorlesung wird dann gezeigt, wie insbesondere die Konzepte, die im Zusammenhang mit Typ-3- und Typ-2-Sprachen eingeführt wurden, im Syntaxanalyseteil eines Übersetzers bzw. bei der automatischen Erzeugung dieses Übersetzerteils zur Anwendung kommen.

Hinzu kommen in beiden Teilen der Vorlesung weitere interessante Themen wie Regular Expressions und andeutungsweise der Bau einfacher kleiner Interpreter mit Hilfe attributierter Grammatiken.

2 Typ 3

2.1 Deterministische endliche Automaten

2.1 Beispiel. Man stelle sich einen kleinen Parkplatz mit 5 Stellplätzen für Autos vor. An der Einfahrt befindet sich eine Schranke. Am Eingang (E) vor der Schranke und am Ausgang (A) befindet sich je eine Induktionsschleife, die ein Signal liefert, wenn ein Auto auf den Parkplatz einfahren will resp. wenn es ihn verlässt.

Der Einfachheit halber gehen wir davon aus, dass nicht beide Induktionsschleifen gleichzeitig ein Signal liefern.

Ein elektrisches System soll die Einfahrtschranke steuern und dazu einen Zähler mit der Anzahl momentan geparkter Autos verwalten. Die Einfahrtschranke soll nur aufgehen (\uparrow), wenn noch ein Parkplatz frei ist; ansonsten soll wartenden Autos an einer Ampel an der Einfahrt angezeigt werden, dass der Parkplatz voll ist (\otimes). Außerdem soll einem Operateur ein Warnsignal ($?!$) geliefert werden, wenn der Parkplatz eigentlich leer sein sollte, aber von der Ausfahrtsschleife das Signal eines wegfahrens Autos geliefert wird. Ansonsten wird dem Operateur mitgeteilt, dass sich die Zahl der Autos um 1 verringert hat ($-$).

Das formalisieren wir wie folgt. Das System verwaltet einen Zähler, für den die Werte in $Z = \{0, 1, 2, 3, 4, 5\}$ in Frage kommen. Es reagiert auf zwei mögliche Eingaben, die wir in einem Eingabealphabet $X = \{E, A\}$ zusammenfassen. Außerdem gibt es ein Ausgabealphabet $Y = \{\uparrow, -, ?!, \otimes\}$.

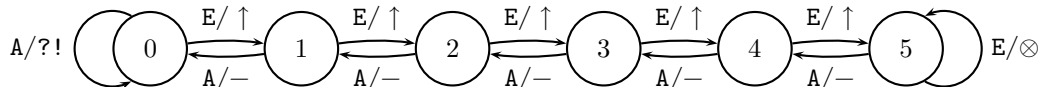
Die Arbeitsweise des Systems kann man z. B. durch zwei Tabellen beschreiben. Die Festlegung der Änderungen des Zählers sind (bis auf die Kombination $(0, A)$) offensichtlich:

| | | alter Zählerstand | | | | | |
|---------|---|-------------------|---|---|---|---|---|
| | | 0 | 1 | 2 | 3 | 4 | 5 |
| Eingabe | E | 1 | 2 | 3 | 4 | 5 | 5 |
| | A | 0 | 0 | 1 | 2 | 3 | 4 |

Die vom System zu erzeugenden Ausgabesignale sehen so aus:

| | | alter Zählerstand | | | | | |
|---------|---|-------------------|------------|------------|------------|------------|-----------|
| | | 0 | 1 | 2 | 3 | 4 | 5 |
| Eingabe | E | \uparrow | \uparrow | \uparrow | \uparrow | \uparrow | \otimes |
| | A | $?!$ | $-$ | $-$ | $-$ | $-$ | $-$ |

Grafisch kann man das Ganze auch so darstellen:



Ein Knoten in dem Graphen entspricht einem Zählerstand. Eine Kante von Knoten i zu Knoten j , die mit x/y beschriftet ist, bedeutet, dass das System von Zustand i bei Eingabe von x in Zustand j übergeht und dabei die Ausgabe y erzeugt.

2.2 Definition. Ein *endlicher Automat* ist festgelegt durch

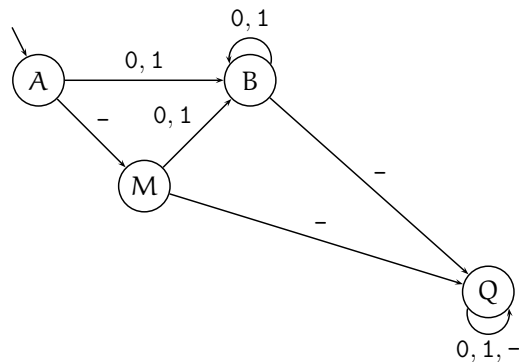
- eine endliche Zustandsmenge Z ,
- einen Anfangszustand $z_0 \in Z$,
- ein Eingabealphabet X ,
- eine Überföhrungsfunktion $f : Z \times X \rightarrow Z$,

Wir verzichten zunächst einmal auf Ausgaben.

2.3 Beispiel. Man betrachte den endlichen Automaten mit Zustandsmenge $Z = \{A, B, M, Q\}$, Anfangszustand A , Eingabealphabet $X = \{0, 1, -\}$ und einer Überföhrungsfunktion wie in der folgenden Tabelle dargestellt:

| | A | B | M | Q |
|-----|-----|-----|-----|-----|
| 0 | B | B | B | Q |
| 1 | B | B | B | Q |
| $-$ | M | Q | Q | Q |

Die grafische Darstellung sieht dann also so aus:



Der Anfangszustand eines endlichen Automaten wird üblicherweise durch einen Pfeil „ohne Anfangsknoten“ gekennzeichnet.

Wenn wir auch im Moment nichts darüber gesagt haben, „was der Automat tun soll“, sollte man doch einmal darüber nachdenken, welche Information er vielleicht liefern könnte.

Für einen Zustand $z \in Z$ und ein Eingabesymbol $x \in X$ ist $f(z, x)$ der Zustand nach Eingabe dieses einzelnen Symbols ausgehend von Zustand z . Manchmal möchte man auch über den nach Eingabe eines ganzen Wortes $w \in X^*$ erreichten Zustand oder gar über alle dabei durchlaufenen Zustände reden.

2.4 Definition. Am bequemsten hinzuschreiben ist das mit Hilfe zweier Abbildungen, die wir mit $f^* : Z \times X^* \rightarrow Z$ und $f^{**} : Z \times X^* \rightarrow Z^*$ bezeichnen wollen. Wir definieren zunächst:

$$f^*(z, \varepsilon) = z$$

$$\text{und für alle } w \in X^* \text{ und } x \in X \text{ sei } f^*(z, wx) = f(f^*(z, w), x)$$

Damit kann man nun f^{**} wie folgt festlegen:

$$\begin{aligned} f^{**}(z, \varepsilon) &= z \\ \text{und für alle } w \in X^* \text{ und } x \in X \text{ sei } f^{**}(z, wx) &= f^{**}(z, w)f(f^*(z, w), x) \end{aligned}$$

2.5 Dies ist nicht die erste rekursive Definition dieses Skriptes, aber eine, anhand derer noch einmal kurz generelle Prinzip festgehalten werden soll.

Man kann z. B. die Funktionswerte einer Funktion festlegen durch die Angabe von

- einem (oder evtl. mehreren) Funktionswert für den (oder evtl. mehrere) „kleinste“ Argumentwerte und
- eine Vorschrift, wie sich ein Funktionswert für einen „größeren“ Argumentwert aus dem/den Funktionswert(en) für „kleinere“ Argumentwerte ergibt.

In obigen Definitionen von f^* und f^{**} wird die Größe eines zweiten Arguments durch seine Länge gegeben. Der kleinste solche Argumentwert ist das leere Wort und von längeren Argumentwerten wird das erste Symbol abgespalten und der, dann kürzere, Rest als kleinerer Argumentwert benutzt.

Der Beweis einer Eigenschaft einer so definierten Funktion kann dann unter Umständen dieser Struktur folgen und eine Variante vollständiger Induktion sein, z. B. wenn die Größe immer eine natürliche Zahl ist.

2.6 Beispiel. Wir wollen zeigen, dass für alle $z \in Z$ und alle $w \in X^*$ gilt: $|f^{**}(z, w)| = 1 + |w|$.

Es sei z ein beliebiger Zustand. Wir führen eine Induktion über die Länge $n = |w|$ von w :

Induktionsanfang $n = 0$: Dann muss $w = \varepsilon$ sein.

In diesem Fall ist laut Definition $|f^{**}(z, w)| = |f^{**}(z, \varepsilon)| = |z| = 1$ und das ist $1 = 1 + 0 = 1 + |\varepsilon|$ wie behauptet.

Induktionsschritt $n - 1 \rightsquigarrow n$: Wenn w ein Wort der Länge $n \geq 1$ ist, dann kann man es schreiben als $w = w'x$ mit $|w'| = n - 1$ und $x \in X$. In diesem Fall ist $f^{**}(z, w'x) = f^{**}(z, w')f(f^*(z, w'), x)$. Also ist dann $|f^{**}(z, w'x)| = |f^{**}(z, w')f(f^*(z, w'), x)| = |f^{**}(z, w')| + |f(f^*(z, w'), x)|$. Nach *Induktionsvoraussetzung* (auch *Induktionsannahme* genannt) ist $|f^{**}(z, w')| = 1 + |w'| = 1 + n - 1 = n$. Außerdem ist natürlich $|f(f^*(z, w'), x)| = 1$, denn $f(\dots, x)$ ist ein einzelner Zustand. Damit ist insgesamt $|f^{**}(z, w'x)| = n + 1$ wie behauptet.

2.7 Angenommen, wir wollten zeigen, dass für alle Wörter $w_1, w_2 \in X^*$ und alle Zustände z gilt:

$$f^*(z, w_1w_2) = f^*(f^*(z, w_1), w_2) .$$

Was hier steht, ist inhaltlich völlig banal: Der Zustand, den man erreicht, wenn man in z startet und w_1w_2 eingibt, ist genau der Zustand, den man erreicht, wenn man in z startet und w_1 eingibt, und ausgehend vom dann erreichten Zustand w_2 eingibt.

Trotzdem muss man es genau genommen beweisen. Überlegen Sie sich, wie ein Induktionsbeweis aussehen könnte.

2.8 Aufgabe. Überlegen Sie sich eine rekursive Definition von f^{**} , in der nicht auf f^* (sondern nur auf f und f^{**}) Bezug genommen wird.

2.9 Definition. Sinnvollerweise liefert jeder Automat im weitesten Sinne irgendwelche Reaktionen. (Wozu sollte man ihn sonst laufen lassen?)

Die einfachste Möglichkeit wäre es wohl, $f^*(z_0, w)$ oder $f^{**}(z_0, w)$ als die zu einer Eingabe w gehörende Ausgabe zu betrachten. Üblicherweise geht man aber den folgenden Weg:

Man hat neben dem Eingabealphabet auch ein *Ausgabealphabet* Y festgelegt. Es kommen zwei Fälle vor: Im einen wird zu jedem Eingabesymbol auch eine Ausgabe erzeugt; man spricht von einem *Mealy-Automaten*. Im anderen gehört zu jedem Zustand eine Ausgabe; dann spricht man von einem *Moore-Automaten*.

Dementsprechend ist bei einem Mealy-Automaten $M = (Z, z_0, X, f, Y, g)$ die *Ausgabefunktion* $g : Z \times X \rightarrow Y$, und bei einem Moore-Automaten $M = (Z, z_0, X, f, Y, g)$ ist die *Ausgabefunktion* $g : Z \rightarrow Y$.

2.10 Beispiel. Der Automat aus Beispiel 2.3 kann benutzt werden, um Eingaben darauf hin zu überprüfen, ob sie gewisse sinnvolle Zahldarstellungen sind oder nicht.

Angenommen, man definiert z. B. $g : Z \rightarrow \{0, 1\}$ so, dass $g(B) = 1$ und $g(z) = 0$ für alle $z \neq B$. Dann liefert der Automat für ein Eingabewort genau dann als letzte Ausgabe eine 1, wenn es eine Dualzahl ist, der optional noch ein Minuszeichen vorangehen darf.

2.11 Aufgabe. Verallgemeinern Sie den Automaten aus Beispiel 2.3 so, dass er auch für Zahlen in Gleitkommadarstellung funktioniert.

1. Erlauben Sie zunächst nur optional einen Dezimalpunkt und nachfolgende Ziffern.
2. Ergänzen Sie in einem zweiten Schritt Zustände, um auch Suffixe wie z. B. E-23 verarbeiten zu können.

Wie ist jeweils die Ausgabefunktion zu wählen?

2.12 Wie im oben genannten Beispiel liegt manchmal bei einem Moore-Automaten der Spezialfall vor, dass einfach nur jedes Eingabewort w klassifiziert werden soll in der Art, dass entschieden wird, ob w eine gewisse Eigenschaft hat oder nicht.

Dann ist $Y = \{0, 1\}$ und man interessiert sich nur für das letzte Ausgabesymbol. Äquivalent dazu spezifiziert man in einem solchen Fall einfach statt Y und g die Teilmenge $F \subset Z$ der Zustände z mit Ausgabe $g(z) = 1$. Diese Zustände nennt man *akzeptierende Zustände*¹ und den Automaten mitunter einen *Akzeptor*. In grafischen Darstellungen werden akzeptierende Zustände durch doppelte Kreise gekennzeichnet.

2.13 Definition. Die von einem endlichen Akzeptor $M = (Z, z_0, X, f, F)$ erkannte Sprache $L(M)$ ist die Menge

$$L(M) = \{w \in X^* \mid f^*(z_0, w) \in F\}$$

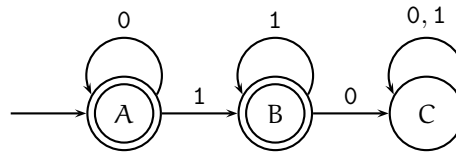
aller Wörter, bei deren Eingabe M vom Anfangszustand z_0 in einen akzeptierenden Zustand $z \in F$ übergeht.

2.14 Beispiel. Für die formale Sprache

$$L = \{0^k 1^\ell \mid k, \ell \in \mathbb{N}_0\} = \{w \in \{0, 1\}^* \mid \text{in } w \text{ kommen alle } 0 \text{ vor allen } 1\}$$

kann man einen endlichen Akzeptor M angeben, der L erkennt:

¹Wir vermeiden den von manchen Autoren auch benutzten Begriff *Endzustände*, weil das erfahrungsgemäß zu Missverständnissen führt.



Der Automat unterscheidet anhand seiner Zustände drei verschiedene Arten von Eingaben:

Zustand A: es kam noch keine 1 in der Eingabe vor;

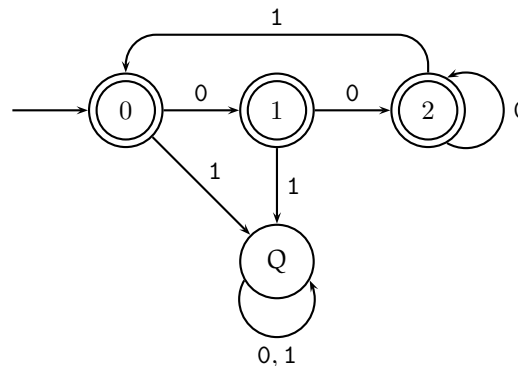
Zustand B: es kam schon mindestens eine 1 in der Eingabe vor, aber danach noch keine 0, d. h. die bisherige Eingabe ist in L ;

Zustand C: in der Eingabe w kam die Zeichenfolge 10 vor, d. h. w gehört nicht zu L und das kann sich durch weitere Eingabesymbole auch nicht mehr ändern.

2.15 Beispiel. Für die formale Sprache

$$L = \{w \in \{0, 1\}^* \mid \text{in } w \text{ kommen vor jeder } 1 \text{ mindestens zwei } 0 \text{ vor} \}$$

kann man einen endlichen Akzeptor M angeben, der L erkennt:



Man beachte, dass man — wie in den beiden vorangegangenen Beispielen — stets für alle Eingabewörter mit dem *immer gleichen* Automaten arbeiten muss, unabhängig davon, wie lang sie sind.

2.16 Gibt es auch eine formale Sprache, die von *keinem* endlichen Akzeptor erkannt werden kann, ganz egal, wieviele (aber endlich viele!) Zustände man verwenden darf? Ja! Und zwar schon dann, wenn man nur $X = \{0\}$ erlaubt.

Man kann das auf verschiedenen Wegen beweisen.

Eine Möglichkeit besteht darin, sich davon zu überzeugen, dass es nur abzählbar unendlich viele, also so viele wie natürliche Zahlen, endliche Akzeptoren gibt, aber überabzählbar unendlich viele, also so viele wie reelle Zahlen, formale Sprachen über $\{0\}$. Und wenn man dann weiß, dass es keine surjektive Abbildung von \mathbb{N} auf \mathbb{R} gibt (Georg Ferdinand Ludwig Philipp Cantor², 1845–1918, sei Dank), ist man fertig.

Man kann aber auch konkrete Beispiele nicht von endlichen Akzeptoren erkennbarer Sprachen angeben. Weil die Argumentation etwas einfacher wird und im Hinblick auf spätere Anwendungen wählen wir im folgenden $X = \{0, 1\}$.

²<http://www-groups.dcs.st-andrews.ac.uk/~history/Mathematicians/Cantor.html>

2.17 Lemma. *Die formale Sprache*

$$\begin{aligned}
L &= \{0^k 1^k \mid k \in \mathbb{N}\} \\
&= \{000 \cdots 0111 \cdots 1 \mid \text{die Anzahl der 0 und die Anzahl der 1 stimmen \u00fcberein}\}
\end{aligned}$$

wird von keinem endlichen Akzeptor erkannt.

2.18 Es ist also $L = \{\varepsilon, 01, 0011, 000111, \dots\}$, aber z. B. ist $00111 \notin L$ und $1010 \notin L$.

Warum kann diese Sprache von keinem endlichen Akzeptor erkannt werden? Betrachten wir einen beliebigen endlichen Akzeptor $M = (Z, z_0, X, f, F)$. Anschaulich gesprochen „muss sich M irgendwie merken, mit wievielen 0 eine Eingabe begonnen hat, bevor alle 1 kommen“, wollte M gerade L erkennen. M hat aber nur endlich viele Zust\u00e4nde; damit „kann man nicht beliebig weit z\u00e4hlen“, d. h. M kann nicht alle m\u00f6glichen L\u00e4ngen des 0-Pr\u00e4fixes unterscheiden.

2.19 Beweis. (von Lemma 2.17) Machen wir diese Idee pr\u00e4zise. Wir f\u00fchren den Beweis indirekt und nehmen an: Es gibt einen endlichen Akzeptor M , der genau L erkennt. Diese Annahme m\u00fcssen wir zu einem Widerspruch f\u00fchren.

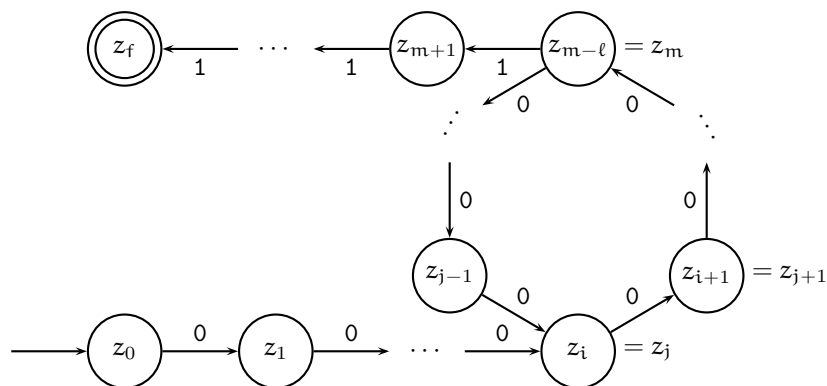
M hat eine gewisse Anzahl Zust\u00e4nde, sagen wir $|Z| = m$. Betrachten wir ein spezielles Eingabewort, n\u00e4mlich $w = 0^m 1^m$.

1. Offensichtlich ist $w \in L$. Wenn also $L(M) = L$ ist, dann muss M bei Eingabe von w in einen akzeptierenden Zustand z_f gelangen: $f^*(z_0, w) = z_f \in F$.
2. Betrachten wir die Zust\u00e4nde, die M bei Eingabe der ersten H\u00e4lfte des Wortes durchl\u00e4uft: $z_0, f(z_0, 0) = z_1, f(z_1, 0) = z_2, \dots, f(z_{m-1}, 0) = z_m$; mit anderen Worten: $f^{**}(z_0, 0^m) = z_0 z_1 \cdots z_m$. Offensichtlich gilt dann: $f^*(z_m, 1^m) = z_f$.

Andererseits besteht die Liste $z_0 z_1 \cdots z_m$ aus $m + 1$ Werten. Aber M hat nur m verschiedene Zust\u00e4nde. Also kommt mindestens ein Zustand doppelt vor.

D. h. der Automat befindet sich in einer Schleife. Sei etwa $z_i = z_j$ f\u00fcr gewisse $i < j$. Genauer sei z_i das erste Auftreten irgendeines mehrfach auftretenden Zustandes und z_j das zweite Auftreten des gleichen Zustandes. Dann gibt es eine „Schleife“ der L\u00e4nge $\ell = j - i > 0$. Und ist der Automat erst einmal in der Schleife, dann bleibt er nat\u00fcrlich darin, solange er weitere 0 als Eingabe erh\u00e4lt. Also ist auch $z_{m-\ell} = z_m$.

Die Folge der durchlaufenen Zust\u00e4nde ist in der folgenden Abbildung skizziert. (Man mache sich klar, dass sie nicht unbedingt den Automaten korrekt wiedergibt; es k\u00f6nnte ja z. B. $z_0 = z_f$ sein.)



3. Nun entfernen wir einige der 0 in der Eingabe, so dass die Schleife einmal weniger durchlaufen wird, d. h. wir betrachten die Eingabe $w' = 0^{m-\ell}1^m$. Wie verhält sich der Akzeptor bei dieser Eingabe? Nachdem er das Präfix³ $0^{m-\ell}$ gelesen hat, ist er in Zustand $z_{m-\ell}$. Dieser ist aber gleich dem Zustand z_m , d. h. M ist in dem Zustand in dem er auch nach der Eingabe 0^m ist. Und wir wissen: $f^*(z_m, 1^m) = z_f \in F$. Also ist $f^*(z_0, 0^{m-\ell}1^m) = f^*(f^*(z_0, 0^{m-\ell}), 1^m) = f^*(z_{m-\ell}, 1^m) = f^*(z_m, 1^m) = z_f$, d. h. M akzeptiert die Eingabe $w' = 0^{m-\ell}1^m$. Aber das Wort w' gehört nicht zu L , da es verschieden viele Nullen und Einsen enthält! Also ist $L(M) \neq L$. Widerspruch!

Also war die Annahme falsch und es gibt gar keinen endlichen Akzeptor, der L erkennt. ■

- 2.20 Beispiel.** Ähnlich kann man auch beweisen, dass die folgende formale Sprache nicht von einem endlichen Akzeptor erkannt werden kann:

$$L = \{0^k 1^i 0^k 1^j \mid i, j, k \in \mathbb{N}\}$$

Ein Wort gehört also genau dann zu L , wenn es zu $\{0\}^* \{1\}^* \{0\}^* \{1\}^*$ gehört und wenn der hintere 0-Block ganz am Anfang auch schon vorgekommen ist.

2.2 Reguläre Ausdrücke

Der Begriff *regulärer Ausdruck* geht ursprünglich auf Kleene zurück und wird heute in unterschiedlichen Bedeutungen genutzt. In diesem Abschnitt beschäftigen wir uns mit regulären Ausdrücken nach der „klassischen“ Definition. Etwas anderes sind die Varianten der *Regular Expressions*, von denen Sie möglicherweise schon im Zusammenhang mit dem ein oder anderen Programm (awk, emacs, grep, perl, sed, java, ...) gelesen haben. Damit sollten reguläre Ausdrücke nicht verwechselt werden. Auf Regular Expressions werden wir in Kapitel 3 genauer eingehen.

- 2.21 Definition.** Es sei A ein Alphabet, das keines der fünf Zeichen aus $Z = \{ |, (,), *, \emptyset \}$ enthält. Ein *regulärer Ausdruck* über A ist eine Zeichenfolge über dem Alphabet $A \cup Z$, die gewissen Vorschriften genügt. Die Menge der regulären Ausdrücke ist wie folgt festgelegt:

- \emptyset ist ein regulärer Ausdruck.
- Für jedes $a \in A$ ist a ein regulärer Ausdruck.
- Wenn R_1 und R_2 reguläre Ausdrücke sind, dann auch $(R_1 | R_2)$ und $(R_1 R_2)$.
- Wenn R ein regulärer Ausdruck ist, dann auch (R^*) .
- Nichts anderes sind reguläre Ausdrücke.

Um sich das Schreiben zu vereinfachen, darf man Klammern auch weglassen. Im Zweifelsfall gilt „Stern- vor Punkt- und Punkt- vor Strichrechnung“, d. h. $R_1 | R_2 R_3^*$ ist z. B. als $(R_1 | (R_2 (R_3^*)))$ zu verstehen. Bei mehreren gleichen binären Operatoren gilt das als links geklammert; zum Beispiel ist $R_1 | R_2 | R_3$ als $((R_1 | R_2) | R_3)$ zu verstehen.

- 2.22 Beispiele.** Die folgenden Zeichenketten sind, sogar im strengen Sinne der Definition, alle reguläre Ausdrücke über dem Alphabet $\{0, 1\}$:

³Nein, das ist kein Grammatikfehler; es heißt wirklich *das* Präfix.

- | | | | |
|-----------------------|--------------------------|------------------|-------------------|
| (a) \emptyset | (b) 0 | (c) 1 | |
| (d) (01) | (e) ((01)0) | (f) (((01)0)0) | (g) ((01)(00)) |
| (h) (\emptyset 1) | (i) (0 1) | (j) ((0(0 1)) 1) | (k) (0 (1 (0 0))) |
| (l) (\emptyset^*) | (m) (0*) | (n) ((10)(1*)) | (o) (((10)1)*) |
| (p) ((0*)*) | (q) (((((01)1)*)*) (0*)) | | |

Wendet man die Klammereinsparungsregeln an, so ergibt sich aus den Beispielen mit Klammern:

- | | | | |
|--------------------|----------------------------|--------------|-------------------|
| (d) 01 | (e) 010 | (f) 0100 | (g) 01(00) |
| (h) \emptyset 1 | (i) 0 1 | (j) 0(0 1) 1 | (k) (0 (1 (0 0))) |
| (l) \emptyset^* | (m) 0* | (n) 101* | (o) (101)* |
| (p) 0** | (q) (011)** \emptyset^* | | |

Die folgenden Zeichenketten sind dagegen auch bei Berücksichtigung der Klammereinsparungsregeln *keine* regulären Ausdrücke über $\{0, 1\}$:

- (|1) ist falsch, denn vor | muss ein regulärer Ausdruck stehen;
- | \emptyset | ist falsch, denn vor und hinter | muss je ein regulärer Ausdruck stehen;
- ()01 ist falsch, denn zwischen (und) muss ein regulärer Ausdruck stehen;
- ((01) ist falsch, denn Klammern müssen „in der üblichen Weise gepaart“ auftreten;
- *(01) ist falsch, denn vor * muss ein regulärer Ausdruck stehen;
- 2* ist falsch, denn 2 ist nicht Zeichen des Alphabetes;

Reguläre Ausdrücke werden benutzt, um formale Sprachen zu spezifizieren. Auch dafür bedient man sich wieder einer induktiven Vorgehensweise; man spricht auch von einer induktiven Definition:

2.23 Definition. Die von einem regulären Ausdruck R beschriebene formale Sprache $\langle R \rangle$ ist wie folgt definiert:

- $\langle \emptyset \rangle = \emptyset = \{\}$ (d. h. die leere Menge).
- Für $a \in A$ ist $\langle a \rangle = \{a\}$.
- Sind R_1 und R_2 reguläre Ausdrücke, so ist $\langle R_1 | R_2 \rangle = \langle R_1 \rangle \cup \langle R_2 \rangle$.
- Sind R_1 und R_2 reguläre Ausdrücke, so ist $\langle R_1 R_2 \rangle = \langle R_1 \rangle \cdot \langle R_2 \rangle$.
- Ist R ein regulärer Ausdruck, so ist $\langle R^* \rangle = \langle R \rangle^*$.

Ist w ein Wort über dem Alphabet eines regulären Ausdrucks R und $w \in \langle R \rangle$, dann sagt man auch, dass w zu dem Muster R passe oder dass der reguläre Ausdruck R das Wort w matcht (vom Englischen *to match*).

2.24 Beispiele.

- $R = 0|1$: Dann ist $\langle R \rangle = \langle 0|1 \rangle = \langle 0 \rangle \cup \langle 1 \rangle = \{0\} \cup \{1\} = \{0, 1\}$.
- $R = (0|1)^*$: Dann ist $\langle R \rangle = \langle (0|1)^* \rangle = \langle 0|1 \rangle^* = \{0, 1\}^*$.
- $R = (0^*1^*)^*$: Dann ist $\langle R \rangle = \langle (0^*1^*)^* \rangle = \langle 0^*1^* \rangle^* = (\langle 0^* \rangle \langle 1^* \rangle)^* = (\langle 0 \rangle^* \langle 1 \rangle^*)^* = (\{0\}^* \{1\}^*)^*$.
Kurzes Überlegen zeigt übrigens, dass das ebenfalls gleich $\{0, 1\}^*$ ist.

2.25 Wie man an den beiden letzten Beispielen in 2.24 sieht, kann man die gleiche formale Sprache durch verschiedene reguläre Ausdrücke beschreiben — wenn sie denn überhaupt so beschreibbar ist.

Damit klingen (mindestens) die beiden folgenden Fragen an:

1. Welche formalen Sprachen sind denn durch reguläre Ausdrücke beschreibbar?

2. Kann man algorithmisch von zwei beliebigen regulären Ausdrücken R_1, R_2 feststellen, ob sie die gleiche formale Sprache beschreiben, d. h. ob $\langle R_1 \rangle = \langle R_2 \rangle$ ist?

Auf die erste Frage werden wir gleich eine Antwort geben. Daraus folgt dann auch, dass die Antwort auf die zweite Frage *ja* lautet.

- 2.26** Allerdings hat das Problem, die Äquivalenz zweier regulärer Ausdrücke zu überprüfen, die Eigenschaft PSPACE-vollständig zu sein wie man in der Komplexitätstheorie sagt. Das bedeutet unter anderem, dass alle *bisher bekannten* Algorithmen im allgemeinen *sehr sehr langsam* sind: die Rechenzeit wächst „stark exponentiell“ mit der Länge der regulären Ausdrücke (z. B. wie 2^{n^2} o.ä.). Es sei noch einmal betont, dass dies für alle bisher bekannten Algorithmen gilt. Man weiß nicht, ob es vielleicht doch signifikant schnellere Algorithmen für das Problem gibt, aber man sie „nur noch nicht gefunden“ hat.

Nun aber zur ersten Frage:

- 2.27 Satz.** Für jede formale Sprache L sind die folgenden beiden Aussagen äquivalent:

1. L kann von einem endlichen Akzeptor erkannt werden.
2. L kann durch einen regulären Ausdruck beschrieben werden.

- 2.28 Definition.** Eine *reguläre Sprache* ist eine formale Sprache, die die Eigenschaften aus Satz 2.27 hat.

Den vollständigen Beweis von Satz 2.27 bleiben wir an dieser Stelle schuldig und zeigen nur, wie man zu jeder Sprache, die von einem endlichen Akzeptor erkannt wird, einen sie beschreibenden regulären Ausdruck konstruieren kann. Die umgekehrte Richtung vom regulären Ausdruck zum endlichen Akzeptor wird später (als Teil eines umfassenderen Satzes) nachgeholt. Die dann zur Verfügung stehenden Hilfsmittel werden uns die Arbeit leichter machen.

- 2.29 Lemma.** Für jeden endlichen Akzeptor M ist $L(M)$ durch einen regulären Ausdruck beschreibbar.

- 2.30 Beweisskizze.** Es sei $M = (Z, z_0, X, f, F)$ ein endlicher Akzeptor mit $m = |Z|$ Zuständen. Für die Zustände schreiben wir z_0, z_1, \dots, z_{m-1} . Für $0 \leq k \leq m$ sei $Z_k = \{z_\ell \mid \ell < k\}$; also ist $Z_0 = \emptyset$ und $Z_m = Z$.

Wir definieren nun für alle $i, j \in \{0, 1, \dots, m-1\}$ und alle $k \in \{0, 1, \dots, m\}$ die formale Sprache

$$L_{ij}^k = \left\{ w \in X^* \mid f^{**}(z_i, w) \in \{z_i\} Z_k^* \{z_j\} \vee (i = j \wedge w = \varepsilon) \right\}$$

L_{ij}^k beinhaltet also diejenigen Eingabewörter, die den Automaten von Zustand z_i in den Zustand z_j überführen und dazwischen nur Zustände mit Indizes kleiner als k besuchen.

Man kann sich nun zweierlei überlegen:

1. $L(M)$ kann man mit Hilfe der L_{ij}^k als Vereinigung ausdrücken.
2. Jede Sprache L_{ij}^k kann man durch einen regulären Ausdruck beschreiben.

Das geht so:

1. Bezeichnet I_F die Menge der Indizes der akzeptierenden Zustände, so ist $L(M) = \bigcup_{j \in I_F} L_{0j}^m$.

2. Als erstes überlegt man sich, wie die Sprachen L_{ij}^0 aussehen:

$$\begin{aligned}
 L_{ij}^0 &= \{w \in X^* \mid f^{**}(z_i, w) \in \{z_i\}Z_0^*\{z_j\} \vee (i = j \wedge w = \varepsilon)\} \\
 &= \{w \in X^* \mid f^{**}(z_i, w) \in \{z_i\}\{z_j\} \vee (i = j \wedge w = \varepsilon)\} \\
 &= \{w \in X^* \mid f^{**}(z_i, w) \in \{z_i z_j\} \vee (i = j \wedge w = \varepsilon)\} \\
 &= \{w \in X^* \mid f^{**}(z_i, w) = z_i z_j \vee (i = j \wedge w = \varepsilon)\} \\
 &= \{x \in X \mid f(z_i, x) = z_j\} \cup \begin{cases} \{\varepsilon\} & \text{falls } i = j \\ \{\} & \text{falls } i \neq j \end{cases}
 \end{aligned}$$

Diese Sprachen sind endliche Mengen bestehend aus einzelnen Eingabesymbolen und evtl. ε , also leicht durch reguläre Ausdrücke beschreibbar.

Und für $k > 0$ gilt folgende Gleichung:

$$L_{ij}^k = L_{ij}^{k-1} \cup L_{i,k-1}^{k-1} (L_{k-1,k-1}^{k-1})^* L_{k-1,j}^{k-1}$$

Damit kann man ausgehend von den regulären Ausdrücken für die L_{ij}^0 sukzessive die für die L_{ij}^1 , für die L_{ij}^2 , usw. bis zu den L_{ij}^m konstruieren.

Natürlich muss man beweisen, dass diese Konstruktion das korrekte Ergebnis liefert. Das unterlassen wir hier. Überlegen Sie sich aber als Übungsaufgabe, warum die rekursive Gleichung für die L_{ij}^k gilt. ■

2.31 Man kann sich fragen, warum in Definition 2.21 neben $|$ mit der Interpretation \cup hinsichtlich der beschriebenen Sprache nicht auch Symbole für die mengentheoretischen Operationen Durchschnitt und Komplementbildung mit hinzugenommen wurden.

Man könnte ja zum Beispiel Definition 2.21 um den Punkt

- Wenn R ein regulärer Ausdruck ist, dann auch $(\sim R)$.

erweitern und Definition 2.23 um

- Ist R ein regulärer Ausdruck, so ist $\langle (\sim R) \rangle = A^* \setminus \langle R \rangle$.

Ein Teil der Antwort ist: Man braucht es nicht, wie man am folgenden Satz sieht.

2.32 Satz. Sind L_1 und L_2 reguläre Sprachen über einem Alphabet A , dann sind auch $A^* \setminus L_1$ und $L_1 \cap L_2$ reguläre Sprachen.

Und das heißt, dass man (für reguläre L_1 und L_2) auch $A^* \setminus L_1$ und $L_1 \cap L_2$ durch reguläre Ausdrücke, so wie wir sie definiert haben, beschreiben kann; also nur mit Hilfe von $*$, $|$ und Konkatination. Solange man nur an reguläre Ausdrücke denkt, ist das nicht offensichtlich. Aber wir können ja Satz 2.27 benutzen. Damit wird der Beweis recht einfach. Damit auch ist die Nützlichkeit verschiedener Charakterisierungen einundderselben Eigenschaft deutlich vor Augen geführt wird.

2.33 Beweis. Es seien $M_1 = (Z_1, z_{01}, A, f_1, F_1)$ resp. $M_2 = (Z_2, z_{02}, A, f_2, F_2)$ endliche Akzeptoren von L_1 resp. L_2 . Die Konstruktionen der gesuchten Akzeptoren sehen wie folgt aus:

Komplement: Ein endlicher Akzeptor M' für $A^* \setminus L_1$ ist ganz leicht anzugeben; er muss genau dann akzeptieren, wenn M_1 nicht akzeptiert. Also leistet $M' = (Z_1, z_{01}, A, f_1, Z_1 \setminus F_1)$ das Gewünschte.

Durchschnitt: Wir konstruieren einen endlichen Akzeptor $M = (Z, z_0, A, f, F)$ mit $L(M) = L_1 \cap L_2$. Anschaulich gesprochen besteht die Idee darin, die beiden Akzeptoren M_1 und M_2 „gleichzeitig parallel“ laufen zu lassen. Mit anderen Worten „merkt sich“ M immer sowohl den Zustand, in dem M_1 nach einer Eingabe wäre, als auch den, in dem M_2 danach wäre. Man wählt also:

- $Z = Z_1 \times Z_2 = \{(z_1, z_2) \mid z_1 \in Z_1 \wedge z_2 \in Z_2\}$;
- für alle $(z_1, z_2) \in Z$ und alle $a \in A$: $f((z_1, z_2), a) = (f_1(z_1, a), f_2(z_2, a))$, denn M soll die Arbeit beider Akzeptoren M_i nachvollziehen;
- $F = F_1 \times F_2$, denn M soll genau dann akzeptieren, wenn ein Eingabewort in $L_1 \cap L_2$ liegt, also wenn beide Akzeptoren M_i es akzeptieren.
- $z_0 = (z_{01}, z_{02})$.

Genau genommen müsste man nun für beide Konstruktionen zeigen, dass sie das Gewünschte leisten. Da die Beweise aber im Wesentlichen nur Schreiarbeit wären, sparen wir uns das hier. ■

2.3 Grammatiken

Bei der Behandlung verschiedener Typen von Automaten ist es sinnvoll, mit der einfachsten Variante, den endlichen Automaten zu beginnen, und sie dann zu mächtigeren Modellen zu erweitern.

Bei Grammatiken ist der umgekehrte Weg natürlich. Die weniger mächtigen Varianten lassen sich bequem als Einschränkungen des allgemeinen Konzeptes darstellen. Deshalb ist jetzt ein bisschen Geduld von Nöten.

2.34 Definition. Eine (erzeugende) Grammatik $G = (N, T, S, P)$ ist festgelegt durch

- ein Alphabet N sogenannter *Nichtterminalsymbole*,
- ein Alphabet T sogenannter *Terminalsymbole*, das zu N disjunkt ist,
- ein ausgezeichnetes *Startsymbol* $S \in N$ und
- eine endliche Menge $P \subset V^*NV^* \times V^*$ sogenannter *Produktionen*.

Dabei schreiben wir zur Abkürzung $V = N \cup T$. Eine *Produktion* $(v, w) \in P$ wird üblicherweise in der Form $v \rightarrow w$ notiert. Dass die linke Seite $v \in V^*NV^*$ ist, bedeutet, dass in ihr mindestens ein Nichtterminalsymbol vorkommen muss; die rechte Seite darf ein beliebiges Wort über V sein, also auch das leere.

2.35 Möchte man mehrere Produktionen mit der gleichen linken Seite notieren, z. B. $v \rightarrow w_1$, $v \rightarrow w_2$ und $v \rightarrow w_3$, so schreibt dafür häufig kurz $v \rightarrow w_1 | w_2 | w_3$.

2.36 Beispiel. $G = (\{X\}, \{0, 1\}, X, \{X \rightarrow 01, X \rightarrow 0X1\})$ ist eine Grammatik. Dafür hätte man auch kürzer $G = (\{X\}, \{0, 1\}, X, \{X \rightarrow 01 \mid 0X1\})$ schreiben können.

2.37 Beispiel. $G = (\{B, Q, S, X, Y, Z\}, \{a\}, S, P)$ mit $P = \left\{ \begin{array}{l} S \rightarrow aXBZ, \\ XB \rightarrow Q, QB \rightarrow Q, QZ \rightarrow \varepsilon, \\ XB \rightarrow aXYB, YB \rightarrow aaBY, YZ \rightarrow BZ, \\ Xa \rightarrow aX, Ba \rightarrow aB \end{array} \right\}$

2.38 Beispiel. $G = (N, T, S, P)$ mit

- $N = \{X_z, X_m, X_v, X_n, X_e, X_b\}$

- $T = \{0, 1, -, ., E\}$
- $S = X_z$
- $P = \{ \begin{array}{l} X_z \rightarrow X_m X_v X_n X_e \\ X_m \rightarrow - \mid \varepsilon \\ X_v \rightarrow X_b \\ X_n \rightarrow . X_b \mid \varepsilon \\ X_e \rightarrow E X_m X_b \mid \varepsilon \\ X_b \rightarrow 0 X_b \mid 1 X_b \mid 0 \mid 1 \end{array} \}$

Eines der ersten Dinge, die man beim Programmieren lernt, ist, sinnvolle Bezeichner für Variablen usw. zu benutzen. Entsprechendes gilt natürlich auch hier. Eine besser lesbare Variante obiger Grammatik könnte zum Beispiel wie folgt aussehen. Dabei muss man sich nur klar machen, dass eine Folge deutscher Buchstaben umgeben von spitzen Klammern wie z. B. $\langle \text{bitfolge} \rangle$ als ein Nicht-terminalsymbol zu verstehen ist.

- $N = \{ \langle \text{zahl} \rangle, \langle \text{opt.minus} \rangle, \langle \text{vorkomma} \rangle, \langle \text{opt.nachkomma} \rangle, \langle \text{opt.exponent} \rangle, \langle \text{bitfolge} \rangle \}$
- $T = \{0, 1, -, ., E\}$
- $S = \langle \text{zahl} \rangle$
- $P = \{ \begin{array}{ll} \langle \text{zahl} \rangle & \rightarrow \langle \text{opt.minus} \rangle \langle \text{vorkomma} \rangle \langle \text{opt.nachkomma} \rangle \langle \text{opt.exponent} \rangle \\ \langle \text{opt.minus} \rangle & \rightarrow - \mid \varepsilon \\ \langle \text{vorkomma} \rangle & \rightarrow \langle \text{bitfolge} \rangle \\ \langle \text{opt.nachkomma} \rangle & \rightarrow . \langle \text{bitfolge} \rangle \mid \varepsilon \\ \langle \text{opt.exponent} \rangle & \rightarrow E \langle \text{opt.minus} \rangle \langle \text{bitfolge} \rangle \mid \varepsilon \\ \langle \text{bitfolge} \rangle & \rightarrow 0 \langle \text{bitfolge} \rangle \mid 1 \langle \text{bitfolge} \rangle \mid 0 \mid 1 \end{array} \}$

Das letzte Beispiel deutet schon darauf hin, dass auch Grammatiken zur Beschreibung der Struktur syntaktischer Gebilde benutzt werden (können), d. h. zur Definition formaler Sprachen.

Zur Vorbereitung der entsprechenden Definition legen wir zunächst fest, wie man mit Grammatiken „arbeitet“.

2.39 Definition. Ist $G = (N, T, S, P)$ eine Grammatik und sind $w_1, w_2 \in V^*$, dann schreibt man $w_1 \Rightarrow w_2$, falls es Wörter $u, u' \in V^*$ und eine Produktion $v \rightarrow w$ gibt, so dass $w_1 = uvu'$ und $w_2 = uwu'$ ist. Um deutlich zu machen, an welcher Stelle v durch w ersetzt wird, benutzen wir gelegentlich einen Unterstrich und notieren $u\underline{v}u' \Rightarrow u\underline{w}u'$.

Sind $w_1, w_2 \in V^*$, dann schreibt man $w_1 \Rightarrow^* w_2$, falls

- $w_1 = w_2$ ist oder
- es ein $w \in V^*$ gibt mit $w_1 \Rightarrow^* w$ und $w \Rightarrow w_2$.

Mit anderen Worten muss man mit einer endlichen Anzahl von Ableitungsschritten (\Rightarrow -Schritten) von w_1 zu w_2 gelangen können. Der Fall von 0 Schritten ist mit eingeschlossen. Man sagt auch, dass \Rightarrow^* die *reflexiv-transitive Hülle* von \Rightarrow sei.

2.40 Beispiel. Für die Grammatik aus Beispiel 2.38 gilt zum Beispiel:

$\underline{X_z} \Rightarrow \underline{X_m} X_v X_n X_e \Rightarrow -\underline{X_v} X_n X_e \Rightarrow -\underline{X_v} \varepsilon X_e \Rightarrow -\underline{X_b} X_e \Rightarrow -1 \underline{X_b} X_e \Rightarrow -10 \underline{X_b} X_e \Rightarrow -101 \underline{X_e} \Rightarrow -101$
und deswegen dann auch $X_z \Rightarrow^* -101$.

Was kann man aus X_b ableiten?

- $X_b \Rightarrow 0$ und $X_b \Rightarrow 1$.

- $X_b \Rightarrow 0X_b \Rightarrow 00$ und $X_b \Rightarrow 0X_b \Rightarrow 01$ sowie $X_b \Rightarrow 1X_b \Rightarrow 10$ und $X_b \Rightarrow 1X_b \Rightarrow 11$.
- „und so weiter“. Genauer gesagt:
Für jedes $k \in \mathbb{N}$ und jedes Wort $w \in T^k$ gilt $X_b \Rightarrow^* w$ und $X_b \Rightarrow^* wX_b$.

Diese letzte Behauptung muss man natürlich beweisen, naheliegenderweise durch vollständige Induktion:

Induktionsanfang $k = 1$: Dann ist die Aussage auf Grund der vorhandenen Produktionen „offensichtlich“ richtig.

Induktionsschritt $k \rightsquigarrow k + 1$: Wir setzen voraus, die Aussage sei für ein k richtig, und betrachten ein beliebiges $w \in T^{k+1}$. Wir müssen zweierlei zeigen:

- $X_b \Rightarrow^* w$
- $X_b \Rightarrow^* wX_b$

Das Wort w ist von der Form $w = w'x$ mit $x \in T$ und $w' \in T^k$. Also ist auf w' die Induktionsvoraussetzung anwendbar. Insbesondere gilt also $X_b \Rightarrow^* w'X_b$. Da $X_b \rightarrow x$ eine Produktion ist, gilt auch $w'X_b \Rightarrow w'x$, also insgesamt $X_b \Rightarrow^* w$. Außerdem ist auch $X_b \rightarrow xX_b$ eine Produktion. Folglich gilt auch $X_b \Rightarrow^* w'X_b \Rightarrow w'xX_b = wX_b$.

2.41 Definition. Die von einer Grammatik $G = (N, T, S, P)$ erzeugte Sprache ist

$$L(G) = \{w \in T^* \mid S \Rightarrow^* w\}.$$

Man beachte, dass man sich in obiger Definition nur für die Wörter aus Terminalsymbolen interessiert. Eine Grammatik kann also (unter anderem) auch als die Spezifikation einer formalen Sprache angesehen werden.

2.42 Beispiel. Greifen wir die Grammatik aus Beispiel 2.36 wieder auf, die nur zwei Produktionen hat: $P = \{X \rightarrow 01, X \rightarrow 0X1\}$. Die Ableitung eines Terminalwortes muss offensichtlich mit der Produktion $X \rightarrow 01$ enden; und vorher muss man (wenn überhaupt) stets die Produktion $X \rightarrow 0X1$ anwenden. Wenn das $k - 1$ mal geschieht, ergibt sich

$$X \Rightarrow 0X1 \Rightarrow 00X11 \Rightarrow \dots \Rightarrow 0^{k-1}X1^{k-1} \Rightarrow 0^k1^k.$$

Also ist $L(G) = \{0^k1^k \mid k \in \mathbb{N}\}$.

Von dieser Sprache haben wir in Punkt 2.19 gesehen, dass sie von keinem endlichen Akzeptor erkannt wird. Man kann mit Grammatiken also formale Sprachen spezifizieren, die man mit endlichen Akzeptoren nicht spezifizieren kann.

Damit stellt sich fast von selbst die Frage, was mit der Umkehrung dieser Aussage ist. Kann auch mit endlichen Akzeptoren formale Sprachen spezifizieren, die man nicht mit Grammatiken spezifizieren kann? Wie wir später sehen werden, ist die Antwort hierauf: Nein.

2.43 Beispiel. Komplizierter ist die Grammatik aus Beispiel 2.37: $G = (\{B, Q, S, X, Y, Z\}, \{a\}, S, P)$ mit

$$P = \left\{ \begin{array}{l} S \rightarrow aXBZ, \\ XB \rightarrow Q, QB \rightarrow Q, QZ \rightarrow \varepsilon, \\ XB \rightarrow aXYB, YB \rightarrow aaBY, YZ \rightarrow BZ, \\ Xa \rightarrow aX, Ba \rightarrow aB \end{array} \right\}$$

Was kann man mit dieser Grammatik ableiten?

- Für alle $k \in \mathbb{N}$ gilt: $XB^kZ \Rightarrow^* \varepsilon$.

Mit Hilfe der Produktionen aus der zweiten Zeile ist nämlich die folgende Ableitung möglich:

$$\underline{X}BBB \cdots BZ \Rightarrow \underline{Q}BB \cdots BZ \Rightarrow \underline{Q}B \cdots BZ \Rightarrow \cdots \Rightarrow QZ \Rightarrow \varepsilon.$$

- Für alle $k \in \mathbb{N}$ gilt: $XB^kZ \Rightarrow^* a^{2k+1}XB^{k+1}Z$.

Mit Hilfe der Produktionen aus der dritten Zeile ist nämlich die folgende Ableitung möglich:

$$\begin{aligned} \underline{X}BBB \cdots BZ &\Rightarrow aX\underline{Y}BBB \cdots BZ \Rightarrow aXaaB\underline{Y}BB \cdots BZ \Rightarrow aXaaBaaB\underline{Y}B \cdots BZ \Rightarrow \cdots \\ &\cdots \Rightarrow aXaaBaaB \cdots aaB\underline{Y}Z \Rightarrow aXaaBaaB \cdots aaBBZ = aX(aaB)^k BZ \text{ und mit Hilfe der Pro-} \\ &\text{duktionen aus der vierten Zeile kann man daraus } a^{2k+1}XB^{k+1}Z \text{ ableiten.} \end{aligned}$$

- Für alle $k \in \mathbb{N}$ gilt: $a^{k^2}XB^kZ \Rightarrow^* a^{(k+1)^2}XB^{k+1}Z$. Das ist wegen $k^2 + 2k + 1 = (k+1)^2$ so.

Man versuche als Übung zu beweisen, dass $L(G) = \{a^{k^2} \mid k \in \mathbb{N}\}$ ist.

2.44 Beispiel. Man kann immer verschiedene Grammatiken angeben, die die gleiche Sprache erzeugen. Betrachten wir $G = (N, T, S, P)$ mit

- $N = \{X_0, X_1, X_2, X_3, X_4, X_5, X_6, X_7\}$
- $T = \{0, 1, -, ., \varepsilon\}$
- $S = X_0$
- $P = \{ \begin{array}{l} X_0 \rightarrow -X_1 \mid X_1 \\ X_1 \rightarrow 0X_2 \mid 1X_2 \\ X_2 \rightarrow 0X_2 \mid 1X_2 \mid \varepsilon \mid X_3 \\ X_3 \rightarrow .0X_4 \mid .1X_4 \mid X_5 \\ X_4 \rightarrow 0X_4 \mid 1X_4 \mid \varepsilon \mid X_5 \\ X_5 \rightarrow E-X_6 \mid EX_6 \\ X_6 \rightarrow 0X_7 \mid 1X_7 \\ X_7 \rightarrow 0X_7 \mid 1X_7 \mid \varepsilon \end{array} \}$

Hinreichend langes „Herumspielen“ und Nachdenken zeigt, dass diese Grammatik die gleiche formale Sprache erzeugt wie die Grammatiken aus Beispiel 2.38.

Was ist bei der Grammatik aus Beispiel 2.44 anders als in 2.38? Auf der rechten Seite jeder Produktion kommt immer nur *ein* Nichtterminalsymbol vor. Und außerdem steht es immer am rechten Ende. Also werden bei jeder Ableitung eines terminalen Wortes die Symbole „der Reihe nach von links nach rechts“ erzeugt.

Ein endlicher Akzeptor andererseits liest immer die Symbole des Eingabewortes „der Reihe nach von links nach rechts“ ... Ob es da einen Zusammenhang gibt? Und wo lauern vielleicht Probleme?

2.4 Typ-3-Grammatiken

2.45 Definition. Eine *Typ-3-Grammatik* ist eine Grammatik, die der folgenden Einschränkung genügt: Jede Produktion ist entweder von der Form $X \rightarrow w$ oder von der Form $X \rightarrow wY$ mit $w \in T^*$ und $X, Y \in N$.

2.46 Auf der rechten Seite einer Produktion darf also höchstens ein Nichtterminalsymbol vorkommen, und wenn dann nur als letztes Symbol. Man spricht auch von *rechtslinearen Produktionen*. Typ-3-Grammatiken heißen auch *rechtslineare Grammatiken*, denn ihre Produktionen sind *alle* rechtslinear.

2.47 Beispiel. Die Grammatik aus Beispiel 2.44 ist vom Typ 3.

Unser Ziel ist es, letztendlich zu zeigen, dass die folgende Erweiterung von Satz 2.27 gilt:

2.48 Satz. Für jede formale Sprache L sind die folgenden drei Aussagen äquivalent:

1. L kann von einem endlichen Akzeptor erkannt werden.
2. L kann durch einen regulären Ausdruck beschrieben werden.
3. L kann von einer Typ-3-Grammatik erzeugt werden.

Einen Teil dieser Aussage haben wir schon in 2.30 bewiesen. Hier folgt nun ein weiterer Baustein. Den Rest der Beweisschuld lösen wir später ein.

2.49 Lemma. Wenn eine formale Sprache L von einem endlichen Akzeptor erkannt wird, dann kann sie auch von einer Typ-3-Grammatik erzeugt werden.

2.50 Beweisskizze. Es sei $M = (Z, z_0, X, f, F)$ ein endlicher Akzeptor. Der Beweis müsste in zwei Schritten geführt werden. Wir konstruieren hier nur eine Grammatik G , zeigen aber anschließend nicht, dass wirklich $L(G) = L(M)$ ist.

1. Wir definieren $G = (N, T, S, P)$ wie folgt:

- $N = \{X_z \mid z \in Z\}$
- $T = X$
- $S = X_{z_0}$
- $P = \{X_z \rightarrow xX_{z'} \mid f(z, x) = z'\} \cup \{X_z \rightarrow \varepsilon \mid z \in F\}$. Mit anderen Worten: Wenn M bei Eingabe x von Zustand z in Zustand z' übergeht, dann und nur dann erlaubt die Grammatik den Übergang vom Nichtterminalsymbol X_z zum Nichtterminalsymbol $X_{z'}$ mit Erzeugung von x .

2. Man müsste nun beweisen, dass $L(G) = L(M)$ ist. Dazu könnte man z. B. zeigen, dass die beiden Inklusionen $L(G) \supseteq L(M)$ und $L(G) \subseteq L(M)$ gelten, d. h. mit anderen Worten, dass die Grammatik die „richtigen“ Wörter erzeugt, aber keine „falschen“. In beiden Fällen könnte man z. B. mit einer Induktion über die Länge der Wörter in $L(M)$ bzw. $L(G)$ argumentieren. Das ist etwas aufwendig, aber nicht sehr lehrreich. Deswegen lassen wir es Interessierten als Übungsaufgabe.

■

2.51 Die Typ-3-Grammatiken, die im eben geführten Beweis auftreten, haben eine spezielle Struktur. Die rechte Seite jeder Produktion ist das leere Wort oder ein einzelnes Terminalsymbol gefolgt von einem Nichtterminalsymbol. Keine rechte Seite besteht also nur aus (einem oder mehreren) Terminalsymbolen und die Fälle $X \rightarrow Y$ und $X \rightarrow wY$ mit $|w| \geq 2$ kommen auch nicht vor.

Man kommt also für die „Simulation“ von DEA mit einem Spezialfall von Typ-3-Grammatiken (sogenannten Typ-3-Grammatiken in *Normalform*) aus. Es stellt sich die Frage, ob man mit Typ-3-Grammatiken im allgemeinen „mehr“ kann als mit solchen in Normalform. Wir werden im folgenden Abschnitt sehen, dass das nicht der Fall ist.

Zum Abschluss dieses Abschnittes beschäftigen wir uns noch mit einem Teilaspekt der Frage, welche Beziehung zwischen Typ-3-Grammatiken und regulären Ausdrücken besteht.

2.52 Lemma. Sind $G_1 = (N_1, T_1, X_{01}, P_1)$ und $G_2 = (N_2, T_2, X_{02}, P_2)$ zwei Typ-3-Grammatiken, dann gibt es Typ-3-Grammatiken G_V , G_K und G_S , so dass gilt:

$$a) L(G_V) = L(G_1) \cup L(G_2)$$

$$b) L(G_K) = L(G_1)L(G_2)$$

$$c) L(G_S) = L(G_1)^*$$

2.53 Beweisskizze. O. B. d. A. seien N_1 und N_2 disjunkt.

a) Man wählt ein „neues“ Nichtterminalsymbol X_V und setzt $G_V = (N_V, T_V, X_V, P_V)$ mit $N_V = N_1 \cup N_2 \cup \{X_V\}$, $T_V = T_1 \cup T_2$, und $P_V = P_1 \cup P_2 \cup \{X \rightarrow X_{01} | X_{02}\}$.

b) Man setzt $G_K = (N_K, T_K, X_{01}, P_K)$ mit $N_K = N_1 \cup N_2$, $T_K = T_1 \cup T_2$, und $P_K = ((P_1 \cup P_2) \setminus \{X \rightarrow w \mid w \in T_1^* \wedge X \rightarrow w \in P_1\}) \cup \{X \rightarrow wX_{02} \mid w \in T_1^* \wedge X \rightarrow w \in P_1\}$.

c) G_S kann man nach einer ähnlichen Idee konstruieren wie G_K : Übungsaufgabe. ■

2.54 Korollar. Zu jedem regulären Ausdruck r gibt es eine Typ-3-Grammatik G mit $L(G) = \langle r \rangle$.

2.55 Aufgabe. Wenn man Lemma 2.52 benutzt, dann bleibt nicht mehr viel zu zeigen. Überlegen Sie sich, was noch zu tun ist.

2.5 Nichtdeterministische endliche Automaten

Statt wie in Punkt 2.51 deterministische EA als Vorbild zu nehmen, um Typ-3-Grammatiken so lange ein zuschränken bis sie dazu passen, wollen wir nun dem umgekehrten Weg gehen und EA so weit verallgemeinern, bis sie zu allgemeinen Typ-3-Grammatiken passen.

Als Vorbereitung legen wir zunächst eine einfache Schreibweise für die Potenzmenge einer Menge fest.

2.56 Definition. Für eine Menge Z bezeichne 2^Z die *Potenzmenge* von Z , d. h. diejenige Menge, deren Elemente gerade die Teilmengen von Z sind.

Manchmal wird dafür auch $P(Z)$ geschrieben.

- 2.57 Beispiel.**
1. Ist $Z = \{0\}$, dann ist $2^Z = \{\emptyset, \{0\}\}$.
 2. Ist $Z = \{0, 1\}$, dann ist $2^Z = \{\emptyset, \{0\}, \{1\}, \{0, 1\}\}$.
 3. Ist $Z = \{0, 1, 2\}$, dann ist $2^Z = \{\emptyset, \{0\}, \{1\}, \{2\}, \{0, 1\}, \{0, 2\}, \{1, 2\}, \{0, 1, 2\}\}$.
 4. Ist $Z = \emptyset$, dann ist $2^Z = \{\emptyset\}$.

Die Schreibweise 2^Z rührt daher, dass für eine endliche Menge Z mit m Elementen ihre Potenzmenge gerade 2^m Elemente hat; also $|2^Z| = 2^{|Z|}$.

Nun können wir definieren:

2.58 Definition. Ein *nichtdeterministischer endlicher Akzeptor* (kurz NEA) $N = (Z, z_0, X, G, F)$ ist festgelegt durch:

- eine endliche Zustandsmenge Z ,
- einen Anfangszustand $z_0 \in Z$,
- ein Eingabealphabet X ,
- eine Menge $F \subseteq Z$ akzeptierender Zustände und

- eine Funktion $g : Z \times X^* \rightarrow 2^Z$ mit der Eigenschaft, dass nur für endlich viele Paare (z, w) gilt: $g(z, w) \neq \emptyset$.

2.59 Wegen der letzten Bedingung ist sichergestellt, dass man jeden NEA durch einen endlichen Text beschreiben kann.

Bei tabellarischer Darstellung hat man z. B. wieder für jeden Zustand eine Spalte und je eine Zeile für jedes Wort $w \in X^*$, für das es mindestens ein $z \in Z$ gibt mit $g(z, w) \neq \emptyset$. Der Eintrag in Zeile w , Spalte z ist natürlich $g(z, w)$.

2.60 Die Bedeutung von $z' \in g(z, w)$ soll sein, dass der NEA, wenn er sich in Zustand z befindet und die nächsten Eingabezeichen das Wort w bilden, in den Zustand z' übergehen *kann*. Wir schreiben dafür manchmal auch $z \xrightarrow{w} z'$.

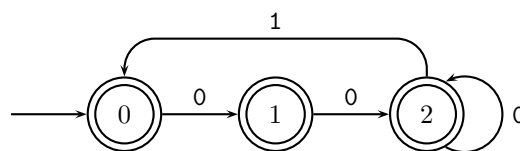
Bei einem deterministischen EA sind alle Übergänge von der Form $g(z, x) = \{z'\}$. Dabei ist $x \in X$ nur ein einzelnes Symbol und es gibt auch nur einen einzigen möglichen Nachfolgezustand z' . Bei NEA ist man weniger restriktiv: Das Wort w kann mehr als ein Zeichen umfassen, oder auch leer sein. Und für ein Paar (z, w) kann es mehrere Nachfolgezustände geben, oder auch gar keinen. Der Akzeptor kann dann in *keinen* neuen Zustand übergehen und dann auch keine weiteren Eingabesymbole verarbeiten.

Die zuletzt genannte Eigenschaft hat der im folgenden Beispiel dargestellte NEA. Die Ähnlichkeit mit dem deterministischen Akzeptor aus Beispiel 2.15 ist alles andere als zufällig:

2.61 Beispiel. Der folgende NEA hat Zustandsmenge $Z = \{0, 1, 2\}$, Anfangszustand 0, $X = \{0, 1\}$, alle Zustände sind akzeptierende Zustände ($F = Z$) und seine Überföhrungsfunktion lautet:

| | 0 | 1 | 2 |
|---|-------------|-------------|---------|
| 0 | $\{1\}$ | $\{2\}$ | $\{2\}$ |
| 1 | \emptyset | \emptyset | $\{0\}$ |

Eine grafische Darstellung des NEA sieht so aus:



2.62 Wenn bei einem deterministischen Akzeptor M alle Zustände akzeptierende Zustände sind, umfasst die von ihm erkannte Sprache $L(M) = A^*$ alle Eingabewörter, denn jedes Eingabewort gibt einen (sogar eindeutigen) Weg vom Anfangszustand zu einem Zustand vor und der ist dann eben auf jeden Fall akzeptierend.

Bei einem nichtdeterministischen Akzeptor ist das anders. Wenn wie in Beispiel 2.61 in manchen Fällen $g(z, w) = \emptyset$ ist, dann existiert für manche Eingabewörter überhaupt keine Möglichkeit des Akzeptors, die Eingabe abzuarbeiten. Ob alle Zustände akzeptierende Zustände sind oder nicht, spielt dann keine Rolle.

2.63 Wir wollen nun analog zu unserem Vorgehen bei DEA die Arbeitsweise von NEA formal festlegen, indem wir eine Funktion g^* definieren.

In Anlehnung an das Vorgehen bei Grammatiken (siehe Definition 2.39) legen wir die Idee zu Grunde, dass ein NEA von einem Zustand z bei Eingabe eines Wortes w in einen Zustand z'

übergehen kann, wenn man w so in Teilwörter $w = w_1 w_2 \cdots w_k$ aufteilen kann, dass es Zustände z_0, z_1, \dots, z_k gibt, für die gilt: $z = z_0 \xrightarrow{w_1} z_1 \xrightarrow{w_2} \cdots \xrightarrow{w_{k-1}} z_{k-1} \xrightarrow{w_k} z_k = z'$.

2.64 Definition. Für Zustände $z_1, z_2 \in Z$ eines NEA und ein $w \in X^*$ schreibt man $z_1 \xrightarrow{w}^* z_2$, falls

- $z_1 = z_2$ und $w = \varepsilon$ ist oder
- es ein $z' \in Z$ und Wörter $w', w'' \in X^*$ gibt mit $w = w' w''$ und $z_1 \xrightarrow{w'}^* z' \xrightarrow{w''}^* z_2$

Damit kann man nun einfach die Funktion $g^* : Z \times X^* \rightarrow 2^Z$ festlegen durch die Forderung

$$g^*(z_1, w) = \{z_2 \mid z_1 \xrightarrow{w}^* z_2\}.$$

2.65 Definition. Die von einem nichtdeterministischen endlichen Automaten N *erkannte Sprache* ist die Menge

$$L(N) = \{w \in X^* \mid g^*(z_0, w) \cap F \neq \emptyset\}$$

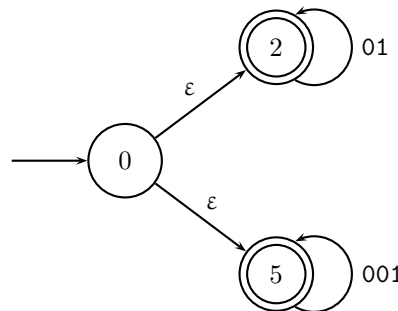
aller Wörter, bei deren Eingabe der Automat N vom Anfangszustand z_0 in einen akzeptierenden Zustand übergehen *kann*.

Man vergleiche diese Definition mit Definition 2.13.

2.66 Beispiel. $N = (\{0, 1, 2, 3, 4, 5\}, 0, \{0, 1\}, g, \{2, 5\})$, wobei g wie folgt festgelegt ist:

| | 0 | 2 | 5 |
|---------------|-------------|-------------|-------------|
| ε | $\{2, 5\}$ | \emptyset | \emptyset |
| 01 | \emptyset | $\{2\}$ | \emptyset |
| 001 | \emptyset | \emptyset | $\{5\}$ |

Im Bild sieht der NEA so aus:



Welche Eingabewörter akzeptiert N ?

Bevor das erste Eingabesymbol gelesen wird, kann der NEA „willkürlich“ in einen der Zustände 2 oder 5 übergehen. Tut er das nicht, können gar keine Eingabesymbole gelesen werden.

Angenommen, N geht von 0 nach 2 über, das ein akzeptierender Zustand ist. Da bisher das Eingabewort ε gelesen wurde, ist also $\varepsilon \in L(N)$. Wenn N erst einmal in Zustand 2 ist, kommt er nur weiter, wenn die nächsten beiden Eingabesymbole 01 sind, und N gelangt dann wieder in Zustand 2. Das muss immer so weiter gehen. Also gehören alle Wörter der Form $(01)^k$ mit $k \in \mathbb{N}_0$ zu $L(N)$.

Analog führt ein spontaner Übergang von Zustand 0 nach 5 dazu, dass alle Wörter der Form $(001)^k$ mit $k \in \mathbb{N}_0$ akzeptiert werden. Alle Eingaben, die nicht Präfix eines solchen Wortes sind, können gar nicht gelesen werden.

Insgesamt ist also $L(N) = \{01\}^* \cup \{001\}^*$. Ein regulärer Ausdruck, der diese Sprache beschreibt, ist z. B. $(01)^* \mid (001)^*$.

2.67 Aufgabe. Um sich klar darüber zu werden, wie „schön“ die Spezifikation einer formalen Sprache mittels nichtdeterministischer Akzeptoren sein kann, sollten Sie einmal versuchen, einen deterministischen Akzeptor zu entwerfen, der die obige Beispielsprache $\{01\}^* \cup \{001\}^*$ erkennt.

Wir wollen nun „von Typ-3-Grammatiken zu deterministischen endlichen Akzeptoren“. Dies geschieht im wesentlichen in zwei Schritten „über nichtdeterministische endliche Akzeptoren“. Nach den schon geleisteten Vorarbeiten ist der erste Schritt mehr oder weniger nur noch Formsache. Man muss sich im wesentlichen nur jede Produktion $X \rightarrow wY$ in der Form $X \xrightarrow{w} Y$ hinschreiben.

2.68 Lemma. Zu jeder Typ-3-Grammatik G gibt es einen NEA N mit $L(N) = L(G)$.

2.69 Beweisskizze. Es sei $G = (N, T, X_0, P)$ eine Typ-3-Grammatik, d.h. jede Produktion ist von einer der Formen $X \rightarrow w$ oder $X \rightarrow wY$ mit $X, Y \in N$ und $w \in T^*$.

Wir konstruieren einen NEA $N = (Z, z_0, X, g, F)$ wie folgt:

- $Z = \{z_X \mid X \in N\} \cup \{z_f\}$; dabei ist z_f ein „neuer“ Zustand;
- $z_0 = z_{X_0}$;
- $X = T$;
- $F = \{z_f\}$;
- $g(z_X, w) = \{z_Y \mid X \rightarrow wY \in P\} \cup \{z_f \mid X \rightarrow w \in P\}$.

Wie schon öfters müsste man nun zeigen, dass für den so konstruierten NEA tatsächlich gilt: $L(N) = L(G)$. Wir überlassen das wieder den Interessierten als Übungsaufgabe zum Thema vollständige Induktion. ■

Nun schließen wir den Kreis:

2.70 Lemma. Zu jedem NEA N gibt es einen DEA M mit $L(M) = L(N)$.

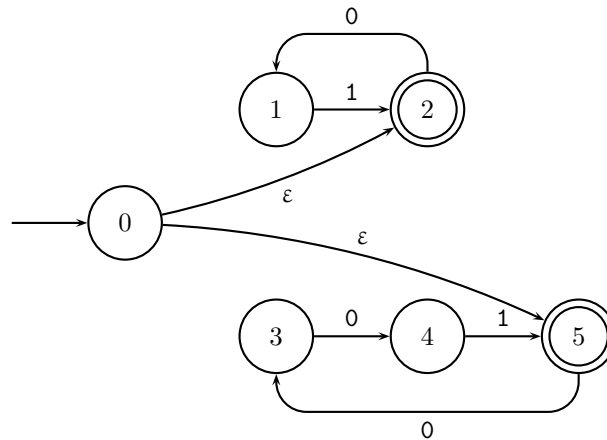
Wir werden den Beweis konstruktiv führen, indem wir einen beliebig vorgegeben NEA nacheinander in folgenden Schritten in einen DEA umwandeln:

1. Beseitigung von Übergängen $z \xrightarrow{w} z'$ mit $|w| \geq 2$;
2. Beseitigung von Übergängen $z \xrightarrow{\varepsilon} z'$; dafür muss man im allgemeinen auch die Menge akzeptierender Zustände anpassen;
3. Beseitigung von Situationen, die echt nichtdeterministisch sind, d.h. $|g(z, w)| \geq 2$. Bei dieser Gelegenheit werden die Fälle mit $g(z, w) = \emptyset$ auch gleich mit beseitigt.

2.71 Beweisskizze. (Schritt 1) Es sei ein beliebiger NEA $N = (Z_N, z_{0N}, X, g, F_N)$ gegeben.

Jeden Übergang $z \xrightarrow{w} z'$ mit $w = x_1 x_2 \cdots x_k$ mit $x_i \in X$ und $k \geq 2$ kann man beseitigen, indem man $k - 1$ neue Zustände z_1, \dots, z_{k-1} einführt und den Übergang $z \xrightarrow{w} z'$ streicht und ersetzt durch die Übergänge $z \xrightarrow{x_1} z_1, z_1 \xrightarrow{x_2} z_2, \dots, z_{k-1} \xrightarrow{x_k} z'$.

Führt man dies zum Beispiel für den NEA aus Beispiel 2.66 durch, so erhält man

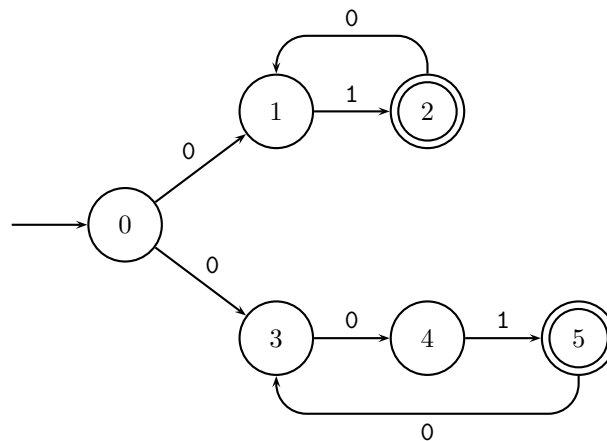


■

2.72 Beweisskizze. (Schritt 2) Als nächstes sollen alle ε -Übergänge aus dem NEA entfernt werden. Die grundsätzliche Idee besteht darin, sie durch Übergänge für einige geeignete einzelne Eingabesymbole zu ersetzen. Für jeden Übergang $z \xrightarrow{\varepsilon} z'$ macht man das folgende:

- Für jede Situation $z \xrightarrow{\varepsilon} z' \xrightarrow{a} z''$ fügt man den Übergang $z \xrightarrow{a} z''$ dem NEA hinzu.
- Für jede Situation $z'' \xrightarrow{a} z \xrightarrow{\varepsilon} z'$ fügt man den Übergang $z'' \xrightarrow{a} z'$ dem NEA hinzu.
- Man entfernt $z \xrightarrow{\varepsilon} z'$ aus dem NEA.

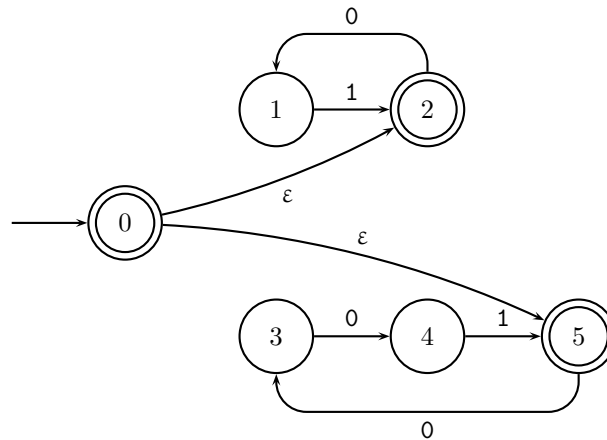
Achtung: Obwohl das auf den ersten Blick sehr sinnvoll aussieht, ergibt sich ohne weitere Vorsichtsmaßnahmen etwa in unserem Beispiel der folgende *falsche (!)* NEA:



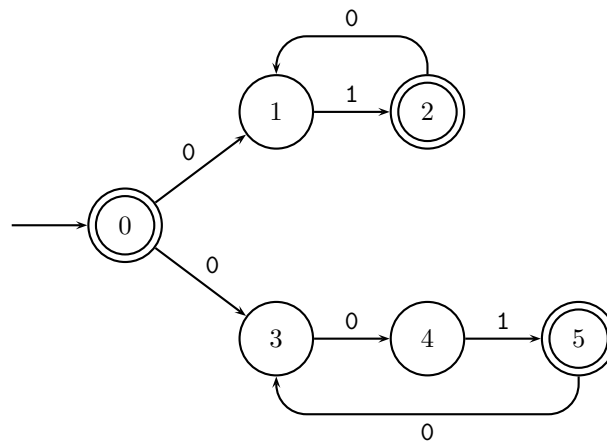
Man beachte, dass so entstandene NEA nicht mehr zum vorangegangenen äquivalent ist: Das leere Wort, das vorher noch zur akzeptierten Sprache gehörte, ist nun nicht mehr dabei.

Um diesen Fehler zu vermeiden, ist es ausreichend, vor der Entfernung der ε -Übergänge die Menge der akzeptierenden Zustände gegebenenfalls zu vergrößern. Ist $z \xrightarrow{\varepsilon} z'$ ein Übergang und ist $z' \in F$, dann kann man auf jeden Fall z ebenfalls zu F hinzunehmen (falls nicht ohnehin schon $z \in F$ ist), ohne an der vom NEA akzeptierten Sprache etwas ändern.

In unserem Beispiel erhält man den NEA



Die anschließende Entfernung aller ε -Übergänge liefert schließlich



■

2.73 Als Übungsaufgaben beschäftige man sich mit den folgenden Punkten:

1. Die Vergrößerung der Menge der akzeptierenden Zustände ändert nichts an der Menge der akzeptierten Wörter.
2. Es wäre *falsch*, auch die „umgekehrte“ Vergrößerung von F durchzuführen. Damit ist gemeint: Ist $z \xrightarrow{\varepsilon} z'$ ein Übergang und ist $z \in F$, dann kann die Hinzunahme von z' zu F dazu führen, dass sich die vom NEA akzeptierte Sprache ändert.
3. Die Entfernung aller ε -Übergänge und deren Ersetzung durch Übergänge der Form $z \xrightarrow{a} z''$ und $z'' \xrightarrow{a} z'$ wie oben beschrieben ändert nichts an der vom NEA akzeptierten Sprache.

2.74 Beweisskizze. (Schritt 3) Es liegt nun also ein NEA vor, bei dem alle Übergänge für einzelne Eingabesymbole stattfinden. Es fehlt nun noch das „Deterministisch-Machen“.

Hierzu stelle man sich vor, man bekäme einen entsprechenden NEA N und eine Eingabe w für ihn vorgelegt und sollte — mittels eines deterministischen Algorithmus — entscheiden, ob $w \in L(N)$ ist. Man müsste also feststellen, ob unter den mehreren möglichen Wegen durch den NEA bei Eingabe w mindestens einer ist, der in einem akzeptierenden Zustand endet. Wie könnte man das tun? „Irgendwie“ muss man alle für w möglichen Wege durch den NEA untersuchen. Das

könnte man nacheinander versuchen. Wenn man zeitsparend arbeiten möchte, könnte man aber zum Beispiel auch versuchen „alle Wege gleichzeitig zu verfolgen“.

Man würde also damit beginnen, dass man sich nur z_0 notiert als einzigen Zustand, in dem sich N am Anfang befinden kann. Aufgrund des ersten Eingabesymbols x_1 kann man alle Zustände bestimmen, in denen sich N nach Eingabe von x_1 befinden kann. Aus diesen Zuständen kann man aufgrund des zweiten Eingabesymbols x_2 man alle Zustände bestimmen, in denen sich N nach Eingabe von x_1x_2 befinden kann, usw. Diese Idee wird auch im Folgenden zu Grunde gelegt:

Zu einem gegebenen $N = (Z_N, z_{0N}, X, g, F_N)$ konstruiert man wie folgt einen passenden DEA $M = (Z_M, z_{0M}, X, f, F_M)$:

- M soll nach Eingabe jedes Wortes die Menge aller derjenigen Zustände gespeichert haben, in denen sich N nach der gleichen Eingabe befinden könnte. Also wählt man $Z_M = 2^{Z_N}$, d. h. die Teilmengen $S \subseteq Z_N$ von Zuständen von N .
- Zu Beginn kann sich N nur in einem einzigen Zustand, nämlich seinem Anfangszustand befinden. Also: $z_{0M} = \{z_{0N}\}$.
- M soll genau dann ein Wort w akzeptieren, wenn N es akzeptiert. Das ist der Fall, wenn einer der bei Eingabe von w erreichbaren Zustände von N akzeptierend ist. Also sollte M akzeptieren, wenn sich in der momentanen Menge von Zuständen von N , die M speichert, mindestens ein akzeptierender Zustand von N befindet, d. h. $F_M = \{S \subseteq Z_N \mid S \cap F_N \neq \emptyset\}$.
- Angenommen, N befindet sich in einem der Zustände aus einer Menge $S = \{s_1, s_2, \dots, s_k\} \subseteq Z_N$ und N liest ein Eingabesymbol x . In welche Zustände kann N dann übergehen? Das sind die, in die N von s_1 aus bei Eingabe von x übergehen kann, und die, in die N von s_2 aus bei Eingabe von x übergehen kann, usw. und die, in die N von s_k aus bei Eingabe von x übergehen kann. Also: $f : Z_M \times X \rightarrow Z_M$ mit $f(S, x) = \bigcup_{z \in S} g(z, x)$.

Nun müsste man beweisen, dass die Konstruktion das Gewünschte leistet. Das ist im Kern die Tatsache, dass für alle $S \subseteq Z_N$ und alle $w \in X^*$ gilt: $g^*(S, w) = f^*(S, w)$. ■

2.75 Bei der eben beschriebenen Konstruktion explodiert die Zustandszahl. Wenn der NEA k Zustände hat, dann hat der zugehörige äquivalente DEA 2^k Zustände. Man kann zeigen, dass es keine Konstruktion gibt, die diesbezüglich besser ist. Für manche formale Sprachen *muss* selbst der kleinste DEA so viele Zustände mehr haben als der kleinste NEA. (Genauer gesagt gibt es unendlich viele (also auch beliebig große) $m \in \mathbb{N}_0$, zu denen es reguläre Sprachen L_m gibt, für die die kleinsten NEA gerade m Zustände haben und die kleinsten DEA 2^m Zustände.)

2.76 Zusammenfassung. Damit sind wir fast am Ende des Teiles über reguläre Sprachen angelangt. Wir haben gesehen:

- Zu jedem DEA gibt es einen äquivalenten regulären Ausdruck.
- Zu jedem regulären Ausdruck gibt es eine äquivalente Typ-3-Grammatik.
- Zu jeder Typ-3-Grammatik gibt es einen äquivalenten NEA.
- Zu jedem NEA gibt es einen äquivalenten DEA.

Damit ist der Kreis geschlossen und es ist gezeigt, dass alle diese Konzepte zur Spezifikation formaler Sprachen im Prinzip gleichmächtig sind. Man kann damit gerade die regulären Sprachen charakterisieren. Gleichwohl kann je nach Anwendungsfall die eine oder die andere Methode unter Umständen sehr viel einfacher anzuwenden sein.

2.6 Zustandsminimierung endlicher Automaten

Zu jeder vorgegebenen regulären Sprache R gibt es immer beliebig viele, beliebig große DEA, die genau R erkennen. Unter Umständen ist man aber an besonders kleinen DEA, d. h. DEA mit besonders wenigen Zuständen, für R interessiert.

Zunächst einmal kann man sich überlegen, dass zu jeder regulären Sprache R mindestens einen R erkennenden DEA mit minimaler Zustandszahl gibt. Tun Sie das!

- 2.77** Zweitens kann man zeigen, dass dieser minimale DEA immer im wesentlichen eindeutig ist; man sagt genauer, dass er „bis auf Isomorphie“ eindeutig ist. Das bedeutet, dass sich „verschiedene“ minimale DEA für die gleiche reguläre Sprache nur durch die Bezeichnung der Zustände unterscheiden.

Den Beweis bleiben wir hier schuldig.

Drittens ist anzumerken, dass man zu einem gegebenen DEA M den äquivalenten kleinsten DEA K relativ einfach konstruieren kann. Im folgenden soll dargestellt werden, wie man das machen kann.

- 2.78** Als erstes sollte man sich klar machen, dass der folgende DEA offensichtlich unnötig groß ist. Es sei $M = (\{A, B, C, D\}, A, \{0, 1\}, f, \{A\})$ mit der Überföhrungsfunktion $f(z, x) = z$ für alle Zustände z und alle Eingabesymbole x . Ausgehend vom Anfangszustand A bleibt mit jedem Eingabesymbol in A . Die Zustände B , C und D sind mit keinem einzigen Eingabewort erreichbar. Solche unerreichbaren Zustände kann man aus einem DEA (oder auch NEA) entfernen, ohne an der erkannten Sprache etwas zu ändern.

Deswegen gehen wir ab sofort davon aus, dass in jedem endlichen Automaten alle Zustände vom Anfangszustand aus erreichbar sind.

Für endliche Automaten ist das Auffinden der überflüssigen Zustände so einfach, dass man es von einem Programm erledigen lassen kann. Das ergibt sich aus dem folgenden Lemma:

- 2.79 Lemma.** Für jeden endlichen Automaten $M = (Z, z_0, X, f, F)$ gilt für alle Zustände $z, z' \in Z$: Ist Zustand z' von z aus durch ein Eingabewort w erreichbar, d. h. $f^*(z, w) = z'$, dann gibt es auch ein Wort w' mit $f^*(z, w') = z'$ und die Länge von w' ist höchstens $|Z| - 1$.

- 2.80 Beweisskizze.** Es sei $f^*(z, w) = z'$. Ist $|w| < |Z|$, dann ist man schon fertig.

Sei andernfalls $w = x_1 \cdots x_n$ mit $x_i \in X$ und $|w| = n \geq |Z|$. In diesem Fall greifen wir eine der Ideen aus Beweis 2.19 auf und betrachten die Folge $z, z_1 = f^*(z, x_1), z_2 = f^*(z, x_1 x_2), \dots, z_n = f^*(z, x_1 \cdots x_n)$. Das sind $n + 1$ Zustände, also echt mehr als der Automat verschiedene hat; d. h. mindestens einer kommt doppelt vor. Also „macht der Automat Schleifen“.

Entfernt man aus w alle Symbole, die zu Schleifen gehören⁴, so ergibt sich ein Wort $w' = x'_1 \cdots x'_k$, das die folgenden Eigenschaften hat:

- $f^*(z, w') = z'$, d. h. auch w' belegt die Erreichbarkeit von z' von z aus.
- Die Folge $z, z'_1 = f^*(z, x'_1), z'_2 = f^*(z, x'_1 x'_2), \dots, z'_n = f^*(z, x'_1 \cdots x'_k)$ enthält keinen Zustand doppelt. Also muss $1 + k \leq |Z|$ sein, d. h. $|w'| < |Z|$.

■

- 2.81 Korollar.** Um zu prüfen, ob ein Zustand z eines endlichen Automaten vom Anfangszustand z_0 aus erreichbar ist, muss man nur für die endlich vielen Wörter $w \in \bigcup_{i=0}^{|Z|-1} X^i$ überprüfen, ob $f^*(z_0, w) = z$ ist.

⁴An dieser Stelle sind wir ungenau. Überlegen Sie sich, warum.

Schneller ist im allgemeinen die folgende Vorgehensweise:

- 2.82 Algorithmus.** Die Menge aller vom Anfangszustand z_0 aus erreichbaren Zustände befindet sich am Ende in der Variable S . S und S' speichern jeweils eine Teilmenge aller Zustände des Automaten:

```

 $S \leftarrow \{z_0\}$ 
repeat
   $S' \leftarrow S$ 
  for each  $(z, x) \in S' \times X$  do
     $S \leftarrow S \cup \{f(z, x)\}$ 
  od
until  $S' = S$ 

```

- 2.83** Die folgenden Überlegungen dienen als Vorbereitung auf Algorithmus 2.90 und dem Verständnis seiner Arbeitsweise.

Ein DEA kann sich mit Hilfe seiner Zustände gewisse Unterschiede bei verarbeiteten Eingaben merken. Etwa kam es in Beispiel 2.14 darauf an, zu wissen, ob in der bisher gelesenen Eingabe

- A: noch keine 1,
- B: schon eine 1, aber danach noch keine 0, oder
- C: schon eine 1 und danach eine 0

vorkamen.

Die entscheidende Frage ist: Welche Unterschiede *muss* sich ein DEA merken? Präziser: Es seien w_1 und w_2 zwei Eingabewörter. Unter welchen Umständen muss $f^*(z_0, w_1) \neq f^*(z_0, w_2)$ sein, damit der DEA das richtige tun kann?

Erinnern wir uns an Beweis 2.19, wo gezeigt wurde, dass $L = \{0^k 1^k \mid k \in \mathbb{N}\}$ nicht regulär ist. Der entscheidende Punkt dort war, dass der hypothetische Automat zwei Eingaben $w_1 = 0^m$ und $w_2 = 0^{m-\ell}$ nicht unterschied, aber man durch Anhängen von $w = 1^m$ einerseits ein Wort $w_1 w \in L$ und andererseits ein Wort $w_2 w \notin L$ erhält.

Das ist eine Katastrophe, die man nur so vermeiden kann: Wenn es zu w_1 und w_2 ein w gibt, so dass $w_1 w \in L$ und $w_2 w \notin L$ ist, dann *muss* ein DEA, der L erkennt, w_1 und w_2 unterscheiden, nämlich dadurch, dass $f^*(z_0, w_1) \neq f^*(z_0, w_2)$ ist.

Zum Beispiel unterscheidet der DEA aus Beispiel 2.14 die Eingaben $w_1 = 000$ und $w_2 = 01$, weil für $w = 0$ zwar $w_1 w = 0000 \in \{0^k 1^\ell \mid \dots\}$ ist, aber $w_2 w = 010$ eben nicht.

- 2.84 Definition.** Sind w_1 , w_2 und w drei Wörter, so dass $w_1 w \in L$ und $w_2 w \notin L$, dann sagen wir, dass (bezüglich L) w_1 und w_2 von w getrennt werden.

- 2.85** Mit der nun schon etwas vertrauten Technik könnte man beweisen: Es sei L eine reguläre Sprache, die von einem DEA $M = (Z, \dots)$ erkannt wird. Wenn w_1 und w_2 von einem Wort w getrennt werden, dann gibt es auch ein Wort w' mit einer Länge $|w'| < |Z|$, das w_1 und w_2 trennt.

Wir unterlassen das hier. Aber vielleicht haben Sie ja Lust auf eine zusätzliche Übungsaufgabe

...

- 2.86 Definition.** Eine Relation $R \subseteq M \times M$ auf einer Menge M ist eine *Äquivalenzrelation*, falls gilt:

- R ist *reflexiv*, d. h. für alle $x \in M$ gilt: $(x, x) \in R$;
- R ist *symmetrisch*, d. h. für alle $x, y \in M$ gilt: Wenn $(x, y) \in R$ ist, dann ist auch $(y, x) \in R$;

- R ist *transitiv*, d. h. für alle $x, y, z \in M$ gilt: Wenn $(x, y) \in R$ ist und $(y, z) \in R$ ist, dann ist auch $(x, z) \in R$.

Eine Äquivalenzrelation schreibt man häufig auch in Infixnotation und verwendet ein Symbol wie z. B. \equiv (oder \approx, \dots). Statt $(x, y) \in R$ schreibt man also etwa $x \equiv y$.

Durch eine Äquivalenzrelation wird die zu Grunde liegende Menge M in *Äquivalenzklassen* eingeteilt. Eine Äquivalenzklasse ist eine Teilmenge von M . Jedes Element x von M liegt in genau einer Äquivalenzklasse, für die man $[x]$ schreibt. Seine Definition lautet: $[x] = \{y \mid y \equiv x\}$.

Wenn $x \equiv y$ ist, dann ist $[x] = [y]$ und umgekehrt⁵.

Für die Menge aller Äquivalenzklassen schreibt man auch M/\equiv .

2.87 Beispiel. Gleichheit ist eine Äquivalenzrelation (auf jeder beliebigen Menge), denn

- es ist immer $x = x$,
- wenn $x = y$ ist, dann auch $y = x$, und
- wenn $x = y$ ist und $y = z$, dann ist auch $x = z$.

In diesem Fall besteht jede Äquivalenzklasse aus nur einem Element: $[x] = \{x\}$.

2.88 Beispiel. Man könnte z. B. zwei natürliche Zahlen als äquivalent bezeichnen, wenn sie die gleiche Anzahl von Primfaktoren haben. Dann sind zum Beispiel die Zahlen 6 und 77 äquivalent, weil beide das Produkt zweier Primzahlen sind; und 41 und 13 sind äquivalent, weil beide (das Produkt) eine(r) Primzahl sind; usw.. Die Äquivalenzklasse von 2 ist also z. B. die Menge aller Primzahlen: $[2] = \{2, 3, 5, 7, 11, 13, 17, \dots\}$, und das heißt auch, dass $[2] = [3] = [5] = [7] = \dots$ ist.

Und durch diese Äquivalenzrelation wird die Menge der natürlichen Zahlen in unendlich viele Äquivalenzklassen zerlegt, denn $2^1, 2^2, 2^3, 2^4$, usw. haben alle paarweise verschieden viele Primfaktoren. Diese Zahlen liegen also in paarweise verschiedenen Äquivalenzklassen.

2.89 Es sei ein beliebiger endlicher Akzeptor $M = (Z_M, z_{0M}, X, f_M, F_M)$ gegeben. Um einen anderen DEA zu finden, der die gleiche formale Sprache erkennt, versucht der folgende Algorithmus, eine Einteilung der Zustände von M in möglichst wenige Äquivalenzklassen zu finden, so dass man Zustände aus verschiedenen Klassen wirklich unterscheiden muss, aber Zustände aus der gleichen Klasse nicht.

Der Algorithmus findet die geeignete Äquivalenzrelation aber nicht auf Anhieb. Vielmehr beginnt er mit einer sehr groben Äquivalenzrelation \equiv_0 (mit nur zwei Klassen), und verfeinert diese dann nach und nach zu Äquivalenzrelationen \equiv_1, \equiv_2 , usw., bis das Ziel erreicht ist.

Die anschauliche Bedeutung der Äquivalenzrelation \equiv_i für ein $i \geq 0$ ist die folgende: Zwei Zustände $z, z' \in Z$ sollen genau dann als „ i -äquivalent“ angesehen werden, also $z \equiv_i z'$, wenn gilt: Man kann mit Hilfe von Eingabewörtern w der Länge $|w| \leq i$ anhand der „Ausgaben“ des DEA (d. h. w akzeptiert oder nicht) die beiden Zustände nicht unterscheiden. Also:

$$z \equiv_i z' \iff \text{für alle } w \in A^* \text{ mit } |w| \leq i \text{ gilt: } f^*(z, w) \in F \iff f^*(z', w) \in F$$

Was bedeutet das? Betrachten wir zunächst den Fall $i = 0$. Zwei Zustände sind 0-äquivalent, wenn sie durch das leere Wort nicht zu unterscheiden sind in dem Sinne, dass entweder beide Zustände akzeptierende Zustände sind oder beide nicht.

Angenommen, zwei Zustände z und z' sind $(i + 1)$ -äquivalent. Was wissen wir dann? Jedenfalls sind sie dann auch i -äquivalent, denn wenn eine Eigenschaft für alle $w \in A^*$ mit $|w| \leq i + 1$ gilt, dann erst recht für alle $w \in A^*$ mit $|w| \leq i$. Für jedes Eingabesymbol $x \in X$ müssen aber auch die

⁵Beweisen Sie das!

Nachfolgezustände $\bar{z} = f(z, x)$ und $\bar{z}' = f(z', x)$ i -äquivalent sein. Denn andernfalls gäbe es ein w , $|w| \leq i$, so dass etwa $f^*(\bar{z}, w) \in F$, aber $f^*(\bar{z}', w) \notin F$. Dann wäre aber auch $f^*(z, xw) \in F$, aber $f^*(z', xw) \notin F$, d. h. z und z' wären gar nicht $(i+1)$ -äquivalent.

Sind umgekehrt zwei Zustände z und z' *nicht* $(i+1)$ -äquivalent, dann sind sie schon nicht i -äquivalent oder es muss ein $x \in X$ geben, so dass die Nachfolgezustände $\bar{z} = f(z, x)$ und $\bar{z}' = f(z', x)$ *nicht* i -äquivalent sind.

Also gilt: $z \equiv_{i+1} z' \iff z \equiv_i z'$ und für alle $x \in X$ gilt: $f(z, x) \equiv_i f(z', x)$.

Mit anderen Worten zerfällt jede Äquivalenzklasse von \equiv_i in eine oder mehrere Äquivalenzklassen von \equiv_{i+1} . Es kann nicht passieren, dass Zustände, die *nicht* i -äquivalent sind, bei \equiv_{i+1} in die gleiche Äquivalenzklasse fallen.

2.90 Algorithmus. Gegeben sei ein beliebiger DEA $M = (Z_M, z_{0M}, X, f_M, F_M)$. Wir konstruieren einen DEA $K = (Z_K, z_{0K}, X, f_K, F_K)$ wie folgt.

Zunächst berechnen wir solange alle Äquivalenzklassen $[z]_i$ der Äquivalenzrelationen $\equiv_0, \equiv_1, \dots$, bis sich dabei nichts mehr ändert:

- Wir beginnen mit \equiv_0 ; hierzu gibt es zwei Äquivalenzklassen:

$$[z]_0 = \begin{cases} F_M & \text{falls } z \in F_M \\ Z_M \setminus F_M & \text{falls } z \notin F_M \end{cases}$$

- Dann arbeitet man sich in einer Schleife von \equiv_i zu \equiv_{i+1} weiter, bis sich nichts mehr ändert:

```

i ← 0
repeat
  for each ⟨Paar (z, z') mit z ≡i z'⟩ do
    z ≡i+1 z' falls für alle x ∈ X gilt: f(z, x) ≡i f(z', x)
  od
until ≡i+1 stimmt mit ≡i überein

```

Nun definieren wir:

- Z_K ist die Menge der Äquivalenzklassen der zuletzt berechneten Äquivalenzrelation \equiv_i .
- $z_{0K} = [z_{0M}]_i$.
- $f_K([z]_i, x) = [f(z, x)]_i$.
- $F_K = \{[z]_i \mid z \in F_M\}$.

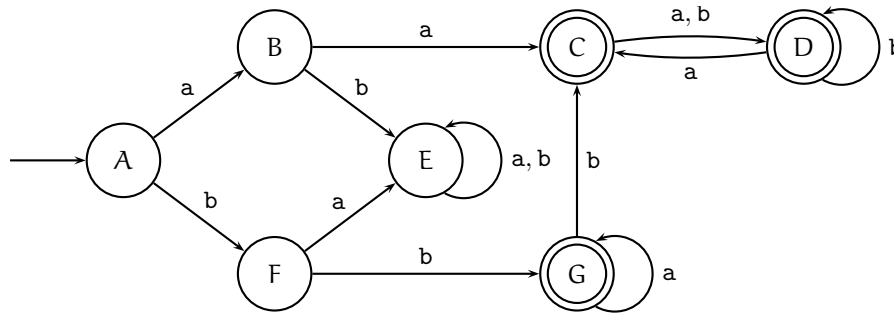
Man muss sich davon überzeugen, dass die obige Definition von f_K keine Widersprüche enthält. Überlegen Sie sich als Übung zumindest, wo einer stecken könnte!

2.91 Als zweites müsste man nun beweisen, dass es keinen DEA mit noch weniger Zuständen als in Z_K gibt, der die gleiche formale Sprache erkennt. Diesen Beweis bleiben wir hier schuldig; wir versprechen nur, dass es so ist.

2.92 Beispiel. Es sei der DEA $M = (\{A, B, \dots, G\}, A, \{a, b\}, f, \{C, D, G\})$ mit der folgenden Überföhrungs-

funktion gegeben:

| | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| a | B | C | D | C | E | E | G |
| b | F | E | D | D | E | G | C |



Wir berechnen nun nacheinander $\equiv_0, \equiv_1, \dots$ bis sich nichts mehr ändert:

\equiv_0 : Bei dieser Äquivalenzrelation gibt es immer genau zwei Äquivalenzklassen, nämlich die Menge der akzeptierenden Zustände und die Menge der nicht akzeptierenden Zustände. Im vorliegenden Beispiel gilt also: $Z/\equiv_0 = \{\{C, D, G\}, \{A, B, E, F\}\}$.

\equiv_1 : Es kann passieren, dass eine Äquivalenzklasse von \equiv_i in mehrere von \equiv_{i+1} zerfällt. Überprüfen wir nacheinander die beiden \equiv_0 -Äquivalenzklassen:

$\{C, D, G\}$: Wir müssen nacheinander die Paare (C, D) , (D, G) und (C, G) untersuchen:

$\{C, D\}$: Der Auszug aus der Überführungstabelle sieht so aus:

| | C | D |
|---|----------------|---|
| a | D \equiv_0 C | |
| b | D \equiv_0 D | |

Da die Nachfolgezustände jeweils 0-äquivalent sind, sind C und D 1-äquivalent.

$\{D, G\}$: Der Auszug aus der Überführungstabelle sieht so aus:

| | D | G |
|---|----------------|---|
| a | C \equiv_0 G | |
| b | D \equiv_0 C | |

Da die Nachfolgezustände jeweils 0-äquivalent sind, sind D und G 1-äquivalent.

$\{C, G\}$: Hierfür muss nicht mehr gerechnet werden, denn wegen $C \equiv_1 D$ und $D \equiv_1 G$ ist auch $C \equiv_1 G$.

Wenden wir uns nun der anderen 0-Äquivalenzklasse zu:

$\{A, B, E, F\}$: Wieder müssen im Prinzip alle Paare von Zuständen untersucht werden:

$\{A, B\}$: Der Auszug aus der Überführungstabelle sieht so aus:

| | A | B |
|---|--------------------|---|
| a | B $\not\equiv_0$ C | |
| b | F \equiv_0 E | |

Da die Nachfolgezustände in einem Fall *nicht* 0-äquivalent sind, sind A und B *nicht* 1-äquivalent.

$\{A, E\}$: Der Auszug aus der Überführungstabelle sieht so aus:

| | A | E |
|---|----------------|---|
| a | B \equiv_0 E | |
| b | F \equiv_0 E | |

Da die Nachfolgezustände jeweils 0-äquivalent sind, sind A und E 1-äquivalent.

$\{A, F\}$: Der Auszug aus der Überführungstabelle sieht so aus:

| | A | F |
|---|--------------------|---|
| a | B \equiv_0 E | |
| b | F $\not\equiv_0$ G | |

Da die Nachfolgezustände in einem Fall *nicht* 0-äquivalent sind, sind A und B *nicht* 1-äquivalent.

$\{B, E\}$: Da A und E 1-äquivalent sind, aber A und B nicht, können B und E ebenfalls nicht 1-äquivalent sein.

$\{E, F\}$: Da A und E 1-äquivalent sind, aber A und F nicht, können E und F ebenfalls nicht 1-äquivalent sein.

$\{B, F\}$: Der Auszug aus der Überführungstabelle sieht so aus:

| | B | F |
|---|--------------|-----|
| a | $C \neq_0 E$ | |
| b | $E \neq_0 G$ | |

Da die Nachfolgezustände *nicht* 0-äquivalent sind, sind B und F *nicht* 1-äquivalent.

Damit ergeben sich insgesamt die Äquivalenzklassen $Z/\equiv_1 = \{\{A, E\}, \{B\}, \{F\}, \{C, D, G\}\}$.

\equiv_2 : Einelementige Äquivalenzklassen können offensichtlich nicht weiter zerfallen. Für die anderen ergibt sich analog wie oben:

$\{C, D, G\}$: Da alle Nachfolgezustände in der Äquivalenzklasse selbst liegen und diese eben schon nicht zerfiel, zerfällt sie nun auch wieder nicht.

$\{A, E\}$: Der Auszug aus der Überführungstabelle sieht so aus:

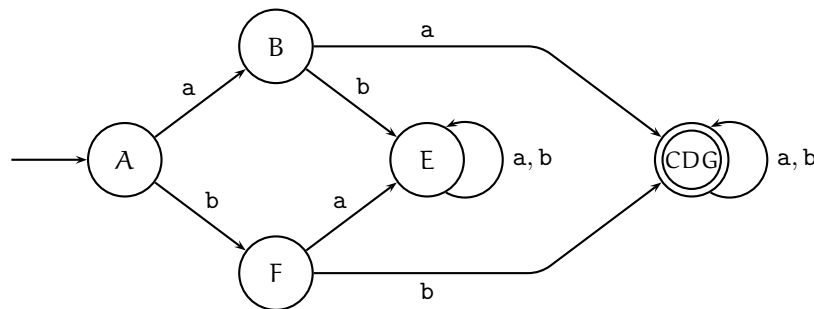
| | A | E |
|---|--------------|-----|
| a | $B \neq_1 E$ | |
| b | $F \neq_1 E$ | |

Da die Nachfolgezustände *nicht* 1-äquivalent sind, sind A und E auch *nicht* 2-äquivalent.

Damit ergeben sich insgesamt die Äquivalenzklassen $Z/\equiv_2 = \{\{A\}, \{B\}, \{E\}, \{F\}, \{C, D, G\}\}$.

\equiv_3 : Da $\{C, D, G\}$ nicht weiter zerfällt und die anderen Klassen nicht weiter zerfallen können, stimmt \equiv_3 mit \equiv_2 überein und man ist fertig.

Der reduzierte DEA hat damit die folgende Form:



3 Regular expressions

Es gibt eine ganze Reihe von Unix-Werkzeugen, für deren Benutzung es sinnvoll oder gar wesentlich ist, dass man mit etwas umgehen kann, was als *regular expression* oder kurz *Regex* bezeichnet wird. Bei diesem Begriff handelt es sich um die wörtliche Übersetzung von „regulärer Ausdruck“; wir benutzen hier aber die englische Beschreibung, weil es sich um allgemeinere Konzepte handelt. Regular expressions sind auch nicht gleich regular expressions. Je nach Werkzeug variiert die konkrete Syntax und unter Umständen auch die Mächtigkeit der zur Verfügung stehenden Konzepte.

Im folgenden gehen wir auf einige Punkte ein, die in diesem Zusammenhang von grundsätzlicher Bedeutung sind. Weitergehende Informationen findet man nicht nur in den üblichen Quellen (man pages, etc.), sondern z.B. auch in dem Buch „Mastering Regular Expressions“ von Jeffrey E. F. Friedl (erschienen bei O'Reilly, neueste (dritte) Auflage von 2006). Unter http://www.gnosis.cx/publish/programming/regular_expressions.html findet man eine gut lesbare Einführung von David Mertz.

3.1 Anwendungsszenario für regular expressions

Grundsätzlich hat man bei den Anwendungen immer ein Wort w und einen regulären Ausdruck R und das benutzte Programm teilt (auf die ein oder andere Weise) mit, ob w ein Teilwort enthält, das durch R beschrieben wird. Man spricht dann im Englischen auch davon einem *Match*, genauer „ R matches w “ oder noch genauer „ R matches in w “.

Im Deutschen werden wir in diesem Skript in Anlehnung an die Tatsache, dass der Duden das Verb „lunchen“ explizit aufführt (einschließlich der Beispielformulierung „du lunchst“) das Verb *matchen* benutzen und zum Beispiel sagen: „ R matcht w “. Ein weiteres solches Problem ist die Wahl des Genus für *Regex*. Wegen der Verwandtschaft mit „regulärer Ausdruck“ haben wir uns für *der* *Regex* entschieden. Als Plural benutzen wir „Regexes“ (in Analogie zum Duden-Eintrag für „Lunch“).

Zum Beispiel matcht der *Regex* $R = 001|11101$ das Wort $w = 11001001000$, denn w enthält (sogar an zwei Stellen) das Teilwort 001.

Werkzeuge wie z.B. *egrep* erwarten üblicherweise als Kommandozeilenargumente einen *Regex* R und einen (oder mehrere) Dateinamen. Es überprüft dann nacheinander jede Zeile der Datei(en) darauf, ob sie als Wort von R gematcht wird. Entsprechende Zeilen werden ausgegeben (oder weiterverarbeitet), alle anderen nicht.

Auf der WWW-Seite <http://www.pcre.org/> kann man ein Paket herunterladen, das unter anderem ein Programm *pcretest* enthält, mit dem man ausprobieren kann, ob und gegebenenfalls was/wie ein *Regex* in einem Wort matcht.

3.2 Regular expressions bei *egrep*

Es gibt nicht *die eine* Implementierung von *egrep*. Es gibt verschiedene, die sich in Details unterscheiden. Deswegen ist es leider immer notwendig, sich vor der Verwendung darüber zu informieren, welches Verhalten die vorliegende Version zeigt. Entsprechendes gilt auch für andere Werkzeuge. (Wer es genau wissen will: Alles, was wir im folgenden sagen, betrifft die extended *regexes* von GNU *egrep*.)

Im folgenden gehen wir davon aus, dass eine Datei namens `liste` existiert, deren Inhalt aus den vier Zeilen

```
0. huo
1. hugo
2. huggo
3. hugggo
```

bestehe. Jede Zeile beginnt mit einer Ziffer und endet mit einem o.

3.1 Beispiele. Ein typischer Aufruf von egrep hat die Form

```
> egrep 'hugo' liste
```

Dabei wird der Dateiname als zweites Argument übergeben und der Regex als erstes. Um (bei komplizierteren Regexes) zu verhindern, dass die Shell einige Zeichen wie z. B. `*` interpretiert, wird der ganze Regex in *single quotes* `' ... '` eingeschlossen.

Der Aufruf

```
> egrep 'hugo' liste
```

liefert als Ausgabe:

```
1. hugo
```

denn nur in dieser Zeile kommt das Teilwort `hugo` vor.

Der Aufruf

```
> egrep 'uo|ggg' liste
```

liefert als Ausgabe:

```
0. huo
3. hugggo
```

denn in der ersten Zeile kommt `uo` vor und in der letzten `ggg`.

Der Aufruf

```
> egrep 'ggg*' liste
```

ist gleichbedeutend mit

```
> egrep 'gg(g)*' liste
```

und liefert als Ausgabe:

```
2. huggo
3. hugggo
```

denn in diesen beiden Zeilen kommt `gg` gefolgt von Null oder mehreren weiteren `g` vor.

Der Aufruf

```
> egrep '(gg)*' liste
```

liefert als Ausgabe:

```
0. huo
1. hugo
2. huggo
3. hugggo
```

denn in allen Zeilen kommt ε als Teilwort vor, was ja von $(gg)^*$ gematcht wird.

- 3.2 Die obigen Beispiele demonstrieren, dass das, was wir als reguläre Ausdrücke kennen gelernt haben, (mit Ausnahme von \emptyset) auch von `egrep` als Regex akzeptiert und in der gewohnten Weise interpretiert wird.

3.3 Bequemere Notation für reguläre Ausdrücke

- 3.3 Der bequemeren Notationsmöglichkeiten wegen erweitert man die Syntax regulärer Ausdrücke. Ist R ein regulärer Ausdruck, so schreibt man

- ε statt \emptyset^* ,
- R^+ statt RR^* ,
- $R?$ statt $R|\varepsilon$

- 3.4 Diese Möglichkeiten bietet auch `egrep`. Der Aufruf

```
> egrep 'g+' liste
```

ist gleichbedeutend mit

```
> egrep 'gg*' liste
```

und liefert als Ausgabe:

```
1. hugo
2. huggo
3. hugggo
```

denn in diesen Zeilen kommt mindestens ein `g` vor.

Der Aufruf

```
> egrep 'g?' liste
```

liefert als Ausgabe:

```
0. huo
1. hugo
2. huggo
3. hugggo
```

denn in allen Zeilen kommt ein `g` vor oder auch nicht.

Die gleiche Ausgabe liefert z. B. auch

```
> egrep 'W?' liste
```

- 3.5 Für eine Reihe einzelner Zeichen x_1, \dots, x_k schreibt man manchmal statt $x_1 | \dots | x_k$ auch $[x_1 \dots x_k]$. Man spricht auch von einer *Zeichenklasse*. Ist auf den Zeichen eine naheliegende Reihenfolge definiert, z. B. bei den Buchstaben `a, \dots, z`, dann erlaubt man auch Angaben der Form $[a-z]$.

- 3.6 Zeichenklassen wie $[a-z]$ sind offensichtlich deutlich kompakter und lesbarer als das Ungetüm `a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z`. Die Variablennamen (ohne Unterstriche) in C werden zum Beispiel durch den regulären Ausdruck $[a-zA-Z][a-zA-Z0-9]^*$ beschrieben.

3.7 Beispiele. Der Aufruf

```
> egrep '[defg]' liste
```

liefert als Ausgabe:

```
1. hugo
2. huggo
3. hugggo
```

denn nur in diesen Zeilen findet sich ein Vorkommen von mindestens einem der Buchstaben d, e, f und/oder g.

Völlig äquivalent dazu ist der Aufruf

```
> egrep '[d-g]' liste
```

3.8 Man kann eine Zeichenklasse auch durch Angabe der *ausgeschlossenen* Zeichen festlegen. Man schreibt $[\sim x_1 \cdots x_k]$ für die Menge aller Zeichen, ausgenommen x_1, \dots, x_k . Wir sprechen hier auch von einer *negierten Zeichenklasse*.

3.9 Beispiel. Der Aufruf

```
> egrep '[^1234hugo. ]' liste
```

liefert als Ausgabe:

```
0. hu0
```

denn nur in dieser Zeile kommt ein Zeichen vor (nämlich die 0), das *nicht* eines der Zeichen 1, 2, 3, 4, h, u, g, o, . oder Leerzeichen ist.

3.10 Manchmal benutzt man für nichtnegative ganze Zahlen $x < y$ die folgenden Notationen um Schreibarbeit zu sparen:

- $R\{x\}$ steht für $\underbrace{(R) \cdots (R)}_{x \text{ mal}}$.
- $R\{x, \}$ steht für $R\{x\}R^*$.
- $R\{x, y\}$ steht für $\underbrace{(R) \cdots (R)}_{x \text{ mal}} \underbrace{(R?) \cdots (R?)}_{y - x \text{ mal}}$.

Es ist also z. B.

- ε äquivalent zu $R\{0\}$.
- R äquivalent zu $R\{1\}$.
- R^* äquivalent zu $R\{0, \}$.
- R^+ äquivalent zu $R\{1, \}$.
- $R?$ äquivalent zu $R\{0, 1\}$.

3.11 Beispiele. Der Aufruf

```
> egrep '[oguh]{5,}' liste
```

liefert als Ausgabe:

```
2. huggo
3. hugggo
```

denn nur in diesen beiden Zeilen finden sich mindestens fünf Vorkommen der Buchstaben o, g, u und/oder h direkt hintereinander.

Dagegen liefert der Aufruf

```
> egrep '[ouh]{5,}' liste
```

als Ausgabe eine leere Liste, denn in keiner Zeile finden sich mindestens fünf Vorkommen der Buchstaben o, u und/oder h direkt hintereinander.

Auch der Aufruf

```
> egrep '[ogh]{5,}' liste
```

liefert als Ausgabe eine leere Liste, denn in der letzten Zeile finden sich zwar fünf Vorkommen der Buchstaben o, u und/oder h, aber nicht direkt hintereinander.

- 3.12 Schließlich gibt es noch die Abkürzung `.` (also einen einzelnen Punkt), der für ein x-beliebiges Zeichen steht.

3.4 Metazeichen in Zeichenklassen

- 3.13 Innerhalb von Zeichenklassen spielen `-` und `^` eine besondere Rolle. Es stellt sich also die Frage, was man tun soll, wenn man eines von ihnen als reguläres Zeichen aufführen will. Gleiches gilt für `[` und `]`. Man spricht auch davon, dass diese Zeichen innerhalb von Zeichenklassen *Metazeichen* sind.

- 3.14 Um in einer Zeichenklasse eines der Metazeichen als reguläres Zeichen interpretieren zu lassen, gelten (jedenfalls in GNU egrep, aber auch bei anderen Werkzeugen) die folgenden Regeln:

- Ein `-` unmittelbar am Anfang einer Zeichenklasse, d. h. unmittelbar nach `[` bzw. nach `[^` bei einer negierten Zeichenklasse steht für sich als reguläres Zeichen.
- Ein `^`, das *nicht* unmittelbar nach der öffnenden Klammer `[` der Zeichenklasse kommt, steht für sich als reguläres Zeichen.
- Eine schließende Klammer `]` unmittelbar am Anfang einer Zeichenklasse, d. h. unmittelbar nach `[` bzw. nach `[^` bei einer negierten Zeichenklasse steht für sich als reguläres Zeichen.
- Eine öffnende Klammer `[` unmittelbar am Anfang einer Zeichenklasse, d. h. unmittelbar nach `[` bzw. nach `[^` bei einer negierten Zeichenklasse steht für sich als reguläres Zeichen.

- 3.15 **Beispiel.** Die Datei `klammern` enthalte zwei Zeilen mit jeweils einer Klammer als einzigem Zeichen:

```
[
]
```

Dann liefert der Aufruf

```
> egrep '[]' klammern
```

als Ausgabe:

```
[
```

und der Aufruf

```
> egrep '[][]' klammern
```

als Ausgabe:

```
]
```

Umgekehrt liefert der Aufruf

```
> egrep '[^[]' klammern
```

als Ausgabe:

```
]
```

und der Aufruf

```
> egrep '[^]]' klammern
```

als Ausgabe:

```
[
```

Der Aufruf

```
> egrep '[^^]' klammern
```

als Ausgabe:

```
[
```

```
]
```

denn bei Zeilen enthalten ein Zeichen, das nicht das ^ ist.

3.5 Anker

Manchmal interessiert man sich nur für Matches, die am Anfang der untersuchten Zeichenkette beginnen und/oder am Ende der Zeichenkette enden. Hierfür gibt es die sogenannten *Anker*.

3.16 Der Anker für den Zeichenkettenanfang ist das ^ und der Anker für das Zeichenkettenende ist das \$.

Ein Ankersymbol matcht also *nicht* das entsprechende Symbol in der Eingabe, sondern sozusagen das leere Wort, und zwar nur an einer ganz bestimmten Stelle in der Eingabe.

3.17 Beispiele. Der Aufruf

```
> egrep 'ugg' liste
```

als Ausgabe:

```
2. huggo  
3. hugggo
```

denn diese Zeilen enthalten die Zeichenkette ugg.

Aber der Aufruf

```
> egrep 'ugg$' liste
```

liefert als Ausgabe die leere Liste denn keine Zeile endet mit der Zeichenkette ugg.

Das gleiche gilt für den Aufruf

```
> egrep '^ugg' liste
```

denn keine Zeile beginnt mit der Zeichenkette ugg.

Andererseits liefert der Aufruf

```
> egrep 'ggo$' liste
```

als Ausgabe:

| |
|-------------------------------|
| <pre>2. huggo 3. hugggo</pre> |
|-------------------------------|

denn diese Zeilen enden sogar mit der Zeichenkette ggo.

3.6 Gruppierungen und Rückwärtsverweise

Bislang haben wir uns darauf beschränkt festzustellen, ob ein Regex R ein Wort w matcht oder nicht. Im Hinblick auf typische Anwendungen von Regexes ist aber auch die Frage interessant, welcher Teil eines Wortes genau gematcht wird, d. h. welches Teilwort v von w zu $\langle R \rangle$ gehört. Das ist natürlich im allgemeinen zunächst einmal nicht eindeutig festgelegt.

3.18 Beispiele.

1. Angenommen, $R = aa|bb$ und $w = _aa_bb_$. Dann gibt es ja zwei Möglichkeiten, „warum“ R in w matcht, nämlich das Vorkommen von aa in w und das Vorkommen von bb .
2. Bei $R = bb^*$ und $w = abbbbc$ liegt ein ähnliches Problem vor: Wird nur ein b gematcht, oder 2 b oder 3 oder 4?
3. Analog bei $R = b|bbb$ und $w = abbbc$: Wird nur b gematcht oder bbb ?

3.19 Die erste Regel zu den oben gestellten Fragen lautet:

1. *Ein weiter vorne in der Zeichenkette beginnender exakter Match wird einem weiter hinter beginnenden immer bevorzugt.*

Diese Regel ist so für (nahezu?) alle Werkzeuge, die mit Regexes umgehen, gültig.

Schwieriger ist es, Regeln anzugeben, die aussagen, wie der gematchte Text ausgewählt wird, wenn mehrere Möglichkeiten bestehen. Hier gibt es verschiedene Spezifikationen und verschiedene Werkzeuge implementieren dann eben auch diese verschiedenen Spezifikationen.

3.20 Zur Länge des gematchten Textes:

- 2a. POSIX Spezifikation: *Bei gleichem Startpunkt in der Zeichenkette wird ein längerer Match einem kürzeren immer bevorzugt.*
- 2b. Bei einigen anderen Werkzeugen (z. B. Perl) ist die Festlegung des gematchten Textes komplizierter, erlaubt aber dafür die Implementierung einfacherer und schneller Verfahren. An Details Interessierte seien auf das zu Beginn des Abschnittes erwähnte Buch von Jeffrey Friedl verwiesen.

3.21 Beispiele.

1. Im Beispiel $R = aa|bb$ und $w = _aa_bb_$ wird also wegen Regel 1 „das aa gematcht“ und nicht das bb (weil das weiter hinten in w steht).
2. Im Beispiel $R = bb^*$ und $w = abbbbc$ wird bei Anwendung von Regel 2a „das bbbb gematcht“ (weil das das längste matchende Teilwort ist). Tatsächlich würde in diesem Fall z. B. auch von Perl noch bbbb gematcht werden.
3. Der Regex $.^*$ matcht also immer *alles*.

Weitere Beispiele folgen im Anschluss an die Einführung von Gruppierungen.

- 3.22** Es ist erlaubt, in einem Regex einen (sinnvollen) Teil durch Klammerung mit (und) zu einem Unterausdruck zu gruppieren auch wenn die Klammereinsparungsregeln es nicht erfordern. Gruppen dürfen natürlich geschachtelt sein. Bei manchen Werkzeugen gibt es Implementierungsschranken bei der Anzahl von Gruppen insgesamt oder bei der Schachtelungstiefe.

Üblicherweise ist der gesamte Regex die Gruppe 0, auch wenn gar nicht geklammert wurde. Im Folgenden wird stets angenommen, dass der gesamte Regex *nicht* geklammert ist. Die Nummern der weiteren Gruppen ergeben sich durch Zählen der öffnenden Klammern bis zur interessierenden Gruppe einschließlich.

3.23 Beispiele.

1. Ist $w = zzbbcbzbz$ und $R = (bb)c(bb)$, dann ergibt sich also die folgende Struktur:

```

zzbbcbzbz
 1  2
 0

```

Jedes Teilwort, das exakt einer Gruppierung entspricht, ist unterstrichen und darunter die Gruppennummer angegeben.

2. Ist $w = zzbbcbzbzzbbccbb$ und $R = (bb)c^*(bb)$, dann ergeben sich die folgenden Gruppierungen:

```

zzbbcbzbzzbbccbb
 1  2
 0

```

- 3.24 Beispiel.** Wir betrachten nun den Regex $(.*)((.))$ und die Zeichenkette $abcdefg$. Um zu verstehen, welche Gruppe welchen Teil matcht, jedenfalls wenn man die POSIX-Regeln zu Grunde legt, mache man sich einmal noch einmal die Regeln aus Punkt 3.19 und aus Punkt 3.20 klar:

```

abcdefg
 1
 0

```

Die zweite Gruppe matcht nichts, weil die erste ein möglichst langes Teilwort matcht und nichts „übrig lässt“.

- 3.25** Sogenannte *Rückwärtsverweise* (engl. *back references*) erweitern die Möglichkeiten von Regexes deutlich. Nachdem Gruppe i abgeschlossen ist, steht ein Ausdruck der Form $\backslash i$ für das Teilwort, das genau von Gruppe i gematcht wurde.

Mitunter gibt es die Implementierungseinschränkung, dass nur einstellige Gruppennummern zulässig sind.

- 3.26 Beispiel.** Der reguläre Ausdruck $(abc|xyz)\backslash 1$ matcht z. B. $abcabc$ und $xyzxyz$. Aber Zeichenketten wie $abcxyz$ werden *nicht* gematcht.

Man mache sich den Unterschied zu $(abc|xyz)(abc|xyz)$ klar.

3.27 Beispiel. Die Datei liste2 enthalte die Zeilen

```
0. hu0
1. hugo
2. huggo
3. hugggo
4. huggggo
5. hugggggo
6. huggggggo
7. hugggggggo
8. huggggggggo
```

Der Aufruf

```
> egrep '((gg)+)\1' liste2
```

liefert als Ausgabe:

```
4. huggggo
5. hugggggo
6. huggggggo
7. hugggggggo
8. huggggggggo
```

Der Aufruf

```
> egrep 'u((gg)+)\1o' liste2
```

liefert als Ausgabe:

```
4. huggggo
8. huggggggggo
```

Am folgenden Beispiel werden die verschiedenen Vorgehensweisen zum Finden des gematchten Textes deutlich:

3.28 Beispiel. Als Regex diene `(.*)*(\1)` und als Eingabewort `abcxabca`. Aufgrund der ersten Regel ist klar, dass der Match auf jeden Fall mit dem ersten Symbol des Wortes beginnt. Aber es gibt mehrere Möglichkeiten; unter anderem:

1. Die erste Gruppe ist maximal „gefräßig“ und matcht `abc`. Damit ist der Rest des Matches eindeutig festgelegt:

```
abcxabca
 1  2
 0
```

2. Die erste Gruppe matcht `a`, `*` ist maximal „gefräßig“ und der Rückwärtsverweis matcht erst am Wortende:

```
abcxabca
 1      2
 0
```

Wie man sieht, ist in dem Fall, dass die erste Gruppe gefräßig ist, die Länge des Gesamtmatches *kleiner*, und in dem Fall, dass die Länge des ersten Gruppenmatches kleiner ist, die Länge des Gesamtmatches *größer*.

Der erste Fall zeigt das Verhalten, wie man es z. B. bei Perl beobachten würde, der zweite Fall entspricht der Forderung des POSIX-Standards.

3.7 Verwendung von Metazeichen als reguläre Zeichen

- 3.29 Aufgrund der Originaldefinition regulärer Ausdrücke haben die Symbole \emptyset , (,), | und * eine besondere Bedeutung; sie können deswegen nicht (ohne Weiteres) zum Alphabet der „eigentlichen“ Zeichen, der auch sogenannten *Literale*, gehören, aus denen die Wörter, die gematcht werden, aufgebaut sind. Diese Symbole heißen deswegen auch *Metazeichen*.

Aufgrund der verallgemeinerten Notationen aus dem vorangegangenen Abschnitt müssen auch zumindest ., +, ?, [,], { und } zu den Metazeichen gezählt werden. Dies betrifft die Vorkommen *außerhalb* von Zeichenklassen.

Die übliche Regel, um Metazeichen auch als Literale „benutzbar“ zu machen besteht in der Vereinbarung, dass ihnen dafür ein Backslash \ vorangestellt werden muss. Hierdurch wird natürlich auch der Backslash zu einem Metazeichen.

4 Typ 2

4.1 Kellerautomaten

Wir haben zum Beispiel in 2.17 gesehen, dass es formale Sprachen gibt, die von keinem DEA erkannt werden können. Gleiches gilt für die Sprache der wohlgeformten Klammerausdrücke aus den Übungsaufgaben. Für diese Sprache sollte außerdem klar sein, dass man bei der Übersetzung von Programmen aus gängigen Programmiersprachen aber im Zusammenhang mit arithmetischen Ausdrücken sozusagen als Teilproblem Syntaxanalyse für Klammerausdrücke machen muss.

Also brauchen wir eine Verallgemeinerung des Konzeptes endlicher Akzeptoren. Das sind die sogenannten Kellerautomaten, die als nächstes eingeführt werden.

4.1 Vorweg schon einmal eine Warnung: Bei endlichen Akzeptoren hatte sich herausgestellt, dass es im Prinzip gleichgültig ist, ob man die deterministische oder nichtdeterministische Variante betrachtet. *Bei Kellerautomaten gibt es hinsichtlich der Erkennungsmächtigkeit einen Unterschied zwischen der deterministischen und der nichtdeterministischen Variante.*

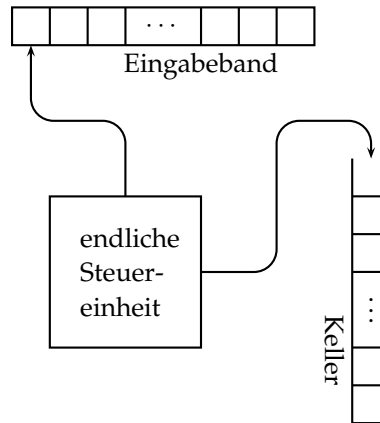
4.2 Wie kann man endliche Automaten erweitern ohne die wesentliche Eigenschaft der endlichen Beschreibbarkeit aufzugeben? Eine der Möglichkeiten besteht jedenfalls in der Hinzunahme von Speicher.

Im „Extremfall“ von unbeschränkt viel, unbeschränkt zugreifbarem Speicher bekommt man das Konzept eines „allgemeinen“ Prozessors. Diesen Fall werden wir später in Abschnitt 5.2 auch noch betrachten. Wir wollen aber nicht gleich mit der Tür ins Haus fallen. Wie also könnten beschränkte Erweiterungen aussehen? Zwei Ideen bieten sich an:

- Die Größe des Speichers wird eingeschränkt. Sie muss aber in Abhängigkeit von der Wortlänge immerhin noch mehr als nur konstant sein, denn sonst hat man nur einen endlichen Automaten.
Ein Modell mit einer solchen Beschränkung werden wir in Abschnitt 5.1 betrachten.
- Die Zugriffsmöglichkeiten auf den Speicher werden eingeschränkt. Die im folgenden betrachteten Kellerautomaten sind ein solches Modell.
 - Man kann sich verschiedene Zugriffsbeschränkungen ausdenken. Man könnte zum Beispiel Zähler vorsehen, die in- und dekrementiert werden können; die Steuereinheit kann aber nicht den aktuellen Inhalt auslesen, sondern nur in Abhängigkeit davon, ob der Zählerinhalt 0 ist oder nicht, im Programm verzweigen. Bei diesem Modell macht es übrigens einen großen Unterschied, ob ein oder zwei Zähler zu Verfügung stehen!
 - Eine andere Zugriffsbeschränkung, nämlich die, die bei Kellerautomaten angewendet wird, besteht darin, dass in jedem Schritt nur auf das zuletzt abgespeicherte Datum zurückgegriffen werden kann.
 - Überlegen Sie sich noch andere interessant aussehende Zugriffsbeschränkungen!

4.3 Definition. Ein *nichtdeterministischer Kellerautomat* (NKA) besteht aus einer endlichen Steuereinheit, einem Eingabeband und einem Keller(speicher). Die Steuereinheit ist stets in einem von endlich vielen Zuständen. Ihre Aktionen können zweierlei bewirken. Zum einen kann (muss aber nicht) in jedem Schritt ein weiteres Eingabesymbol gelesen werden. Die Eingabesymbole stehen nacheinander auf einem Eingabeband, über das der KA einmal von links nach rechts Symbol für Symbol

hinweg gehen kann. Zum anderen hat die Steuereinheit Zugriff auf einen *Keller* (siehe nachfolgende Abbildung). Im Keller sind im allgemeinen mehrere Symbole „übereinander“ abgespeichert. Das Charakteristikum des Kellers ist, dass der KA immer nur das „oberste“ Symbol herausnehmen (und lesen) und eventuell durch ein oder mehrere neue Kellersymbole ersetzt.



Formal ist ein nichtdeterministischer Kellerautomat festgelegt durch

- eine endliche Zustandsmenge Z ,
- einen Anfangszustand $z_0 \in Z$,
- ein Kelleralphabet Y ,
- ein Anfangskellersymbol $y_0 \in Y$
- ein Eingabealphabet X ,
- eine Überföhrungsfunktion $f : Z \times Y \times (X \cup \{\varepsilon\}) \rightarrow 2^{Z \times Y^*}$ und
- eine Menge $F \subseteq Z$ akzeptierender Zustände.

Ist ein NKA im Zustand z und liest auf dem Keller Symbol y , so kann er im Prinzip zunächst einmal „entscheiden“, ob er ein Eingabesymbol lesen will oder nicht.

- Liest der NKA kein Eingabesymbol, so gibt $f(z, y, \varepsilon)$ eine (unter Umständen leere) Menge von möglichen „Aktionen“ $(z', v) \in Z \times Y^*$ vor. Dabei ist z' der neue Zustand und v ein Wort aus Kellersymbolen, das statt des entfernten y im Keller gespeichert wird (erstes Symbol zuoberst, letztes zuunterst).
- Liest der NKA ein Eingabesymbol, so gibt $f(z, y, x)$ eine (unter Umständen leere) Menge von möglichen „Aktionen“ $(z', v) \in Z \times Y^*$ vor für den Fall, dass das Eingabesymbol gerade x ist. Die Bedeutung von z' und v ist die gleiche wie eben.

Um zu verhindern, dass der Keller jemals ganz leer wird, verlangen wir, dass immer, wenn das Kelleraufangssymbol y_0 aus dem Keller gelesen wird, es auch wieder zuunterst auf den Keller gelegt werden muss, d. h. in diesem Fall muss bei jeder „Aktion“ (z', v) das Wort v mit y_0 beginnen.

4.4 Definition. Ein *deterministischer Kellerautomat* (DKA) ist im Prinzip wie ein NKA definiert, muss aber den folgenden Einschränkungen genügen:

- Bei vorgegebenen z und y muss eindeutig festgelegt sein, ob ein Eingabesymbol gelesen wird oder nicht. D. h. entweder ist $f(z, y, \varepsilon) = \emptyset$ oder für alle $x \in X$ ist $f(z, y, x) = \emptyset$.
- Ist $f(z, y, \varepsilon) = \emptyset$, dann enthält $f(z, y, x)$ für alle $x \in X$ genau eine Aktion (z', v) .
- Ist $f(z, y, \varepsilon) \neq \emptyset$, dann enthält es genau eine Aktion (z', v) .

4.5 Definition. Die von einem KA K erkannte Sprache ist die Menge aller Eingabewörter mit der folgenden Eigenschaft: Wenn man K mit dieser Eingabe und einem Keller, der nur das Kelleraufangssymbol enthält, startet, dann hat K nach einigen Schritten alle Eingabesymbole gelesen und die Steuereinheit ist in einem akzeptierenden Zustand.

4.6 Beispiel. Der folgende KA überprüft, ob ein Eingabewort $w \in X^* = \{0,1\}^*$ von der Form $0^k 1^k$ ist. Von diesem Problem hatten wir in Lemma 2.17 gesehen, dass es nicht von einem endlichen Akzeptor gelöst werden kann.

Wir benutzen Kelleralphabet $Y = \{*, 0\}$, wobei $*$ das Anfangssymbol sei. Zustandsmenge ist $Z = \{z_0, z_1, z_+, z_-\}$, wobei z_0 der Anfangszustand sei und es nur einen akzeptierenden Zustand gebe: $F = \{z_+\}$.

Die wesentlichen Zustandsübergänge sind in der folgenden Tabelle aufgeführt. In allen anderen Fällen werde kein weiteres Eingabesymbol mehr gelesen, der neue Zustand sei stets z_- und das auf den Keller gelegte Symbol gleich dem zuletzt entfernten.

| z | y | x | z' | v |
|-------------------------|-----|---------------|-------|---------------|
| z_0 | $*$ | 0 | z_0 | $0*$ |
| z_0 | 0 | 0 | z_0 | 00 |
| z_0 | 0 | 1 | z_1 | ε |
| z_1 | 0 | 1 | z_1 | ε |
| z_1 | $*$ | ε | z_+ | $*$ |
| in allen anderen Fällen | z | y | z_- | y |

Ist zum Beispiel das Eingabewort 00001111 , dann werden nacheinander dessen Symbole gelesen und die durchlaufenen Zustände und Kellerinhalte sind:

| | | | | | | | | | | | | | | | | | | |
|-------|-----|-------|-----|-------|-----|-------|-----|-------|-----|-------|-----|-------|-----|-------|-----|-------|---------------|-------|
| z_0 | 0 | z_0 | 0 | z_0 | 0 | z_0 | 0 | z_0 | 1 | z_1 | 1 | z_1 | 1 | z_1 | 1 | z_1 | ε | z_+ |
| $*$ | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | $*$ | | $*$ |
| | | $*$ | | 0 | | 0 | | 0 | | 0 | | 0 | | $*$ | | | | |
| | | | | $*$ | | 0 | | 0 | | 0 | | $*$ | | | | | | |
| | | | | | | $*$ | | 0 | | $*$ | | | | | | | | |
| | | | | | | | | $*$ | | | | | | | | | | |

Im Hinblick auf spätere Anwendungen stellen wir den Ablauf noch einmal etwas anders dar:

| gelesene Eingabe | neuer Zustand | neuer Kellerinhalt |
|---------------------|------------------|-----------------------|
| | z_0 | $*$ |
| 0 | z_0 | $0*$ |
| 00 | z_0 | $00*$ |
| 000 | z_0 | $000*$ |
| 0000 | z_0 | $0000*$ |
| 00001 | z_1 | $000*$ |
| 000011 | z_1 | $00*$ |
| 0000111 | z_1 | $0*$ |
| 00001111 | z_1 | $*$ |
| 00001111 | z_+ | $*$ |

Haben zwei aufeinander folgende Zeilen die Struktur

| | | |
|------|------|-------|
| w | z | yv |
| wx | z' | $v'v$ |

so bedeutet das, dass der Kellerautomat, als er im Zustand z und sein Kellerinhalt yv (mit oberstem sichtbarem Symbol y) war, Eingabesymbol x gelesen hat, in Zustand z' übergegangen ist und das Wort v' gekellert hat. Analog ist

| | | |
|-----|------|-------|
| w | z | yv |
| w | z' | $v'v$ |

zu interpretieren für den Fall, dass kein Eingabesymbol gelesen wird.

4.7 Definition. Für ein Wort $w \in A^*$ bezeichne w^R das Spiegelbild von w . Es sei genauer $\varepsilon^R = \varepsilon$ und für $x \in A$ und $w \in A^*$ sei $(xw)^R = w^R x$.

4.8 Überlegen Sie sich, warum gilt: $(ww^R)^R = ww^R$. Mit anderen Worten: Ist ein Wort w' von der Form $w' = ww^R$, dann ist w' ein Palindrom, d. h. $w' = w'^R$.

Überlegen Sie sich, dass umgekehrt jedes Palindrom gerader Länge von der Form ww^R ist. Wie sehen Palindrome ungerader Länge aus?

4.9 Beispiel. Es sei L_{pal} die formale Sprache aller Palindrome gerader Länge über dem Alphabet $A = \{a, b\}$, d. h. $L_{pal} = \{ww^R \mid w \in A^*\}$.

Wir wollen einen Kellerautomaten angeben, der L_{pal} erkennt. Die Idee besteht darin, zunächst die erste Hälfte der Eingabe im Keller zu speichern und anschließend jedes Symbol der zweiten Eingabehälfte mit dem obersten, ausgelesenen Kellersymbol zu vergleichen. Bei Ungleichheit kann die Eingabe „abgelehnt“ werden, bei Gleichheit macht man mit dem nächsten Eingabesymbol und dem nächsten auf dem Keller erschienenen Symbol weiter bis der Keller „leer“ ist.

Das Problem besteht darin, dass der Kellerautomat „nicht wissen kann“, wann die Mitte der Eingabe erreicht ist und er von Einkellern auf Vergleichen umschalten muss. Man hat daher intuitiv das Gefühl, dass der Kellerautomat nichtdeterministisch sein *muss*. Das ist auch tatsächlich so; einen entsprechenden Beweis führen wir hier aber nicht.

Der Kellerautomat benutzt vier Zustände z_{in} , z_{out} , z_- und z_+ , von denen der erste der Anfangszustand und nur der letztgenannte akzeptierend sei. Wie schon weiter vorne beschränken wir uns darauf, den „wesentlichen“ Teil der Überföhrungsfunktion anzugeben. In allen anderen Fällen tue der Kellerautomat „nichts“.

| z | y | x | z' | v |
|-------------------------|-----|---------------|-----------|---------------|
| z_{in} | * | a | z_{in} | a* |
| z_{in} | * | b | z_{in} | b* |
| z_{in} | a | a | z_{in} | aa |
| z_{in} | a | b | z_{in} | ba |
| z_{in} | b | a | z_{in} | ab |
| z_{in} | b | b | z_{in} | bb |
| z_{in} | a | ε | z_{out} | a |
| z_{in} | b | ε | z_{out} | b |
| z_{out} | a | a | z_{out} | ε |
| z_{out} | a | b | z_- | ε |
| z_{out} | b | a | z_- | ε |
| z_{out} | b | b | z_{out} | ε |
| z_{out} | * | ε | z_+ | * |
| in allen anderen Fällen | z | y | z | y |

Zum Beispiel wird das Eingabewort abaaaaba wie folgt verarbeitet:

| z_{in} | a | z_{in} | b | z_{in} | a | z_{in} | a | z_{in} | ε | z_{out} | a | z_{out} | a | z_{out} | b | z_{out} | a | z_{out} | ε | z_+ |
|----------|---|----------|---|----------|---|----------|---|----------|---------------|-----------|---|-----------|---|-----------|---|-----------|---|-----------|---------------|-------|
| * | | a | | b | | a | | a | | a | | a | | b | | a | | * | | * |
| | | * | | a | | b | | a | | a | | b | | a | | * | | | | |
| | | | | * | | a | | b | | b | | a | | * | | | | | | |
| | | | | | | * | | a | | a | | * | | | | | | | | |
| | | | | | | | | * | | * | | | | | | | | | | |

- 4.10** Überlegen Sie sich, was passiert, wenn der Kellerautomat „zum falschen Zeitpunkt“ von Kellern auf Vergleichen umschaltet.

Überlegen Sie sich, wie der Kellerautomat Eingaben verarbeiten kann, die keine Palindrome sind.

- 4.11** Es sei noch einmal erwähnt, dass man beweisen kann, dass zum Beispiel L_{pal} (und unendliche viele andere formale Sprachen auch) zwar von nichtdeterministischen Kellerautomaten erkannt werden können, aber *nicht* von deterministischen Kellerautomaten.

Im Unterschied zu endlichen Automaten gibt es bei Kellerautomaten also einen echten Unterschied in der Mächtigkeit zwischen der deterministischen und der nichtdeterministischen Variante.

- 4.12** Außerdem gibt es formale Sprachen, die auch von keinem nichtdeterministischen Kellerautomaten erkannt werden können. Ein Beispiel ist:

$$L = \{0^k 1^k 2^k \mid k \in \mathbb{N}\}$$

Für einen einfach aufzuschreibenden Beweis würde man ein sogenanntes Pumping-Lemma (z. B. das von Bar-Hillel oder das von Ogden) benötigen; darauf werden wir im Rahmen dieser Vorlesung aber nicht näher eingehen.

4.2 Kontextfreie Grammatiken

- 4.13 Definition.** Eine *kontextfreie Grammatik* ist eine Grammatik, deren Produktionen alle die Eigenschaft haben, dass ihre linke Seite aus einem einzelnen Nichtterminalsymbol besteht. Kontextfreie Grammatiken heißen auch *Typ-2-Grammatiken* (T2G).

Eine formale Sprache, die von einer kontextfreien Grammatik erzeugt werden kann, heißt *kontextfreie Sprache*.

- 4.14 Beispiele.** Die Grammatiken in den Beispielen 2.36 und 2.38 sind kontextfrei.

Die Grammatik in Beispiel 2.37 ist *nicht* kontextfrei, weil z. B. die Produktion $XB \rightarrow Q$ eine bei T2G verbotene linke Seite hat.

Jede T3G (siehe Def. 2.45) ist eine T2G, denn jede Produktion muss von der Form $X \rightarrow wY$ mit $w \in T^*$ und $X, Y \in N$ sein; also ist insbesondere jede linke Seite ein Nichtterminalsymbol.

- 4.15 Beispiel.** Die Grammatik $G = (\{X, Y_0, Y_1\}, \{0, 1\}, X, P)$ mit den Produktionen $P = \{X \rightarrow 0Y_01Y_1, Y_0 \rightarrow X0, Y_1 \rightarrow X1\}$ ist kontextfrei. (Sie ist aber nicht von Typ 3!)

- 4.16 Beispiel.** Die Grammatik $G = (\{S\}, \{(\cdot, \cdot)\}, S, \{S \rightarrow \varepsilon \mid (S) \mid SS\})$ ist kontextfrei.

- 4.17 Definition.** Ist $uXu' \Rightarrow uwu'$ ein Ableitungsschritt mittels einer Produktion $X \rightarrow w$ einer T2G, so heißt er ein *Linksableitungsschritt*, falls das ersetzte X das am weitesten links stehende Nichtterminalsymbol in uXu' ist, falls also $u \in T^*$ ist. In einem solchen Fall schreibt man auch $uXu' \xRightarrow{\ell} uwu'$.

Eine Ableitung(sfolge) $w_0 \Rightarrow w_1 \Rightarrow \dots \Rightarrow w_k$ ist eine *Linksableitung(sfolge)* falls jeder einzelne Schritt ein Linksableitungsschritt ist. In einem solchen Fall schreibt man auch $w_0 \xRightarrow{\ell}^* w_k$.

Analog definiert man *Rechtsableitungsschritte* (\xRightarrow{r}) und *Rechtsableitungen* (\xRightarrow{r}^*).

4.18 Beispiel. Wir betrachten wieder die Grammatik $G = (\{S\}, \{ (,) \}, S, \{S \rightarrow \varepsilon \mid (S) \mid SS\})$.

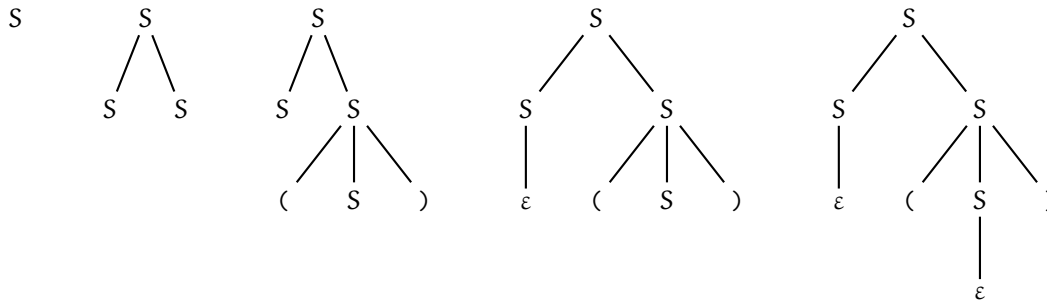
Die Ableitungsfolge $S \Rightarrow SS \Rightarrow \underline{S}(S) \Rightarrow (\underline{S}) \Rightarrow ()$ ist *keine* Linksableitung, denn im zweiten Schritt wird *nicht* das am weitesten links stehende Nichtterminalsymbol ersetzt. Es ist auch *keine* Rechtsableitung, denn im dritten Schritt wird *nicht* das am weitesten rechts stehende Nichtterminalsymbol ersetzt.

Aber die Ableitungsfolge $\underline{S} \Rightarrow SS \Rightarrow \underline{S} \Rightarrow (\underline{S}) \Rightarrow ()$ ist eine Linksableitung und $\underline{S} \Rightarrow SS \Rightarrow S(\underline{S}) \Rightarrow \underline{S}() \Rightarrow ()$ ist eine Rechtsableitung.

4.19 Definition. Ein *Ableitungsbaum* zu einer Ableitung $X \Rightarrow^* w$ ist ein Graph, dessen Knoten mit Nichtterminalsymbolen, Terminalsymbolen oder ε beschriftet sind. Der Aufbau des Baumes ergibt sich wie folgt:

- Die Wurzel des Baumes ist mit X beschriftet.
- Die Blätter des Baumes von links nach rechts aneinandergereiht ergeben w .
- Gehört zu einer Ableitung $X \Rightarrow^* wYw'$ der Ableitungsbaum B' , so ergibt sich der Ableitungsbaum B zur Ableitung $X \Rightarrow^* wYw' \Rightarrow wwv'$ mit zuletzt angewendeter Produktion $Y \rightarrow v$ daraus, indem an das mit Y beschriftete Blatt von B' Nachfolgeknoten angehängt werden, die mit den Symbolen von v beschriftet sind.

4.20 Beispiel. Für obige Grammatik und die Ableitung $S \Rightarrow SS \Rightarrow \underline{S}(S) \Rightarrow (\underline{S}) \Rightarrow ()$ ergibt sich schrittweise von links der im folgenden rechts dargestellte Ableitungsbaum:



Mit anderen Worten sind von links nach rechts die Ableitungsbäume für die Ableitungen S , $S \Rightarrow SS$, $S \Rightarrow SS \Rightarrow \underline{S}(S)$, $S \Rightarrow SS \Rightarrow \underline{S}(S) \Rightarrow (\underline{S})$ und $S \Rightarrow SS \Rightarrow \underline{S}(S) \Rightarrow (\underline{S}) \Rightarrow ()$ dargestellt.

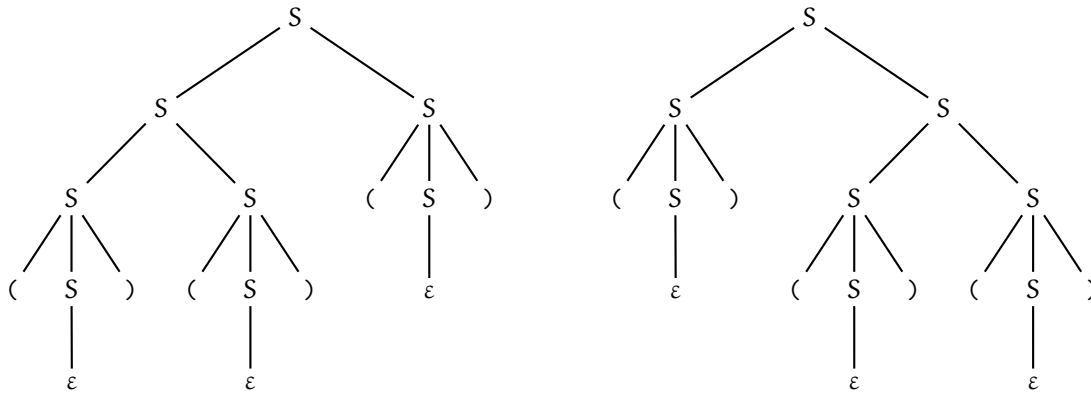
4.21 Wie man sieht, ist der Begriff des Ableitungsbaums „allgemeiner“ als der der Ableitung. Auch die Ableitungen $\underline{S} \Rightarrow SS \Rightarrow \underline{S} \Rightarrow (\underline{S}) \Rightarrow ()$ und $\underline{S} \Rightarrow SS \Rightarrow S(\underline{S}) \Rightarrow \underline{S}() \Rightarrow ()$ liefern den gleichen, in Punkt 4.20 ganz rechts dargestellten Ableitungsbaum.

4.22 Umgekehrt kann man aus jedem Ableitungsbaum eine (und nur eine) Linksableitung ablesen (und analog genau eine Rechtsableitung). Das heißt auch, dass es zu jeder Ableitung eine „äquivalente“ Links- (bzw. Rechts-)ableitung gibt.

4.23 Definition. Eine Typ-2-Grammatik ist *mehrdeutig*, falls es für ein Wort mindestens zwei verschiedene Ableitungsbäume (also auch zwei verschiedene Linksableitungen bzw. Rechtsableitungen) gibt. Sonst heißt die Grammatik *eindeutig*.

Eine kontextfreie Sprache ist (*inhärent*) *mehrdeutig*, falls jede sie erzeugende T2G mehrdeutig ist.

4.24 Beispiel. Die oben angegebene Grammatik ist mehrdeutig. Zum Beispiel kann man das Wort $()()()$ auf mehrere Weisen (links-)ableiten: $S \xRightarrow{\ell} \underline{S} \xRightarrow{\ell} \underline{SS} \xRightarrow{\ell}^* ()\underline{S} \xRightarrow{\ell}^* ()()()$ und $S \xRightarrow{\ell} \underline{S} \xRightarrow{\ell}^* ()()()$ haben die verschiedenen Ableitungsbäume:



Die formale *Sprache* ist aber *nicht* inhärent mehrdeutig. Man kann für sie nämlich auch eine eindeutige Grammatik angeben: $G = (\{S, T\}, \{ (,) \}, S, \{ S \rightarrow \varepsilon \mid ST, T \rightarrow (S) \})$.

- 4.25 Gibt es überhaupt mehrdeutige kontextfreie Sprachen, also kontextfreie Sprache, für die *jede* sie erzeugende Grammatik mehrdeutig ist? Ja. Zum Beispiel ist die Sprache

$$L = \{a^i b^j c^k \mid i = j \text{ oder } j = k \text{ oder } i = k\}$$

über dem Alphabet $\{a, b, c\}$ inhärent mehrdeutig. Einen Beweis bleiben wir schuldig (er ist nämlich nicht ganz einfach).

Im Hinblick auf Übersetzerbau will man eindeutige Grammatiken, d. h. man ist bei der Syntax von Programmiersprachen eingeschränkt, jedenfalls soweit man sie mit T2G beschreiben will.

4.3 Zusammenhang zwischen Kellerautomaten und Typ-2-Grammatiken

- 4.26 **Satz.** Eine formale Sprache kann genau dann von einem Kellerautomaten erkannt werden, wenn sie von einer Typ-2-Grammatik erzeugt werden kann.

Ein exakter Beweis dieses Satzes ist nicht ganz einfach, weshalb wir ihn hier weglassen. Im Hinblick auf Übersetzerbau wollen wir aber zwei mögliche Vorgehensweisen skizzieren, wie man mit Hilfe von Kellerautomaten kontextfreie Sprachen erkennen kann.

- 4.27 (**Beweisidee**) Zu einer Typ-2-Grammatik $G = (N, T, X_0, P)$ kann man leicht einen Kellerautomaten K konstruieren, der genau $L(G)$ erkennt. K hat nur drei Zustände z_0, z und z_+ . Das Kelleralphabet umfasst ein Kellern Anfangssymbol $*$ und die Terminalsymbole $x \in T$ und Nichtterminalsymbole $X \in N$ der Grammatik.

Die Arbeitsweise von K wird durch die folgenden Aktionen beschrieben:

- $f(z_0, *, \varepsilon) = \{(z, X_0*)\}$
- Für jede Produktion $X \rightarrow w$ hat K die Möglichkeit, einen „Produktionsschritt“ zu machen, wenn X gerade oben auf dem Keller liegt, d. h. $(z, w) \in f(z, X, \varepsilon)$; in diesem Fall wird kein Eingabesymbol gelesen.
- Für jedes $x \in T$ hat K die Möglichkeit, in einem „Leseschritt“ x in der Eingabe zu lesen, wenn x gerade oben auf den Keller liegt, d. h. $(z, \varepsilon) \in f(z, x, x)$. Für den Fall $x \neq x'$ gibt es aber keine Aktion: $f(z, x, x') = \emptyset$.

- $f(z, *, \varepsilon) = \{(z_+, *)\}$

Im ersten Schritt wird K also stets ohne Lesen eines Eingabesymbols zunächst einmal X_0 kellern.

Anschließend kann K wie folgt arbeiten: Wenn das oberste Kellersymbol ein Nichtterminalsymbol X ist, ersetzt K es in einem Produktionsschritt durch die rechte Seite einer Produktion $X \rightarrow w$. Wenn das oberste Kellersymbol ein Terminalsymbol x ist, versucht K einen Leseschritt und entfernt x vom Keller, sofern das nächste Eingabesymbol ebenfalls x ist. Andernfalls ist K in einer Sackgasse.

Die Unterscheidung, ob ein Produktionsschritt oder Leseschritt zu versuchen ist, und gegebenenfalls welcher Leseschritt, ist immer eindeutig festgelegt. Bei der Wahl des Produktionsschrittes wird der Kellerautomat aber im allgemeinen nichtdeterministisch sein, da die Grammatik mehrere Produktionen mit der gleichen linken Seite haben kann.

Man kann sich nun überlegen: Wenn es für ein Wort w eine Linksableitung in G gibt, dann gibt es eine Berechnung von K , nach der w akzeptiert wird. K führt nämlich gemäß den angewandten Produktionen von G in der gleichen Reihenfolge die entsprechenden Aktionen aus, unterbrochen durch die passenden Leseschritte. Irgendwann ist die Eingabe vollständig gelesen und der Keller ist bis auf $*$ leer, so dass K in den Zustand z_+ übergehen und w akzeptieren kann.

Kann umgekehrt K ein Wort w akzeptieren, dann muss es eine Linksableitung dafür in G geben, also ist auch $w \in L(G)$.

Wir demonstrieren diese Beweisidee an einem Beispiel.

4.28 Beispiel. Wir betrachten wieder die Grammatik $G = (\{S\}, \{(\,,\,)\}, S, \{S \rightarrow \varepsilon \mid (S) \mid SS\})$ und die Linksableitung $S \Rightarrow SS \Rightarrow (S)S \Rightarrow ((S))S \Rightarrow (())S \Rightarrow (())(S) \Rightarrow (())()$.

Entsprechend dieser Linksableitung kann der Kellerautomat wie folgt arbeiten:

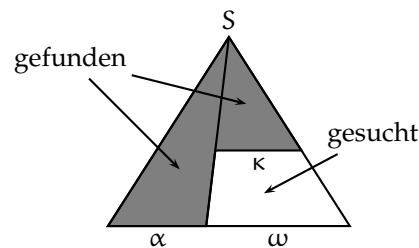
| gelesene Eingabe | neuer Zustand | neuer Kellerinhalt | verwandte Produktion |
|---------------------|------------------|-----------------------|-----------------------------|
| | z_0 | $*$ | |
| | z | $S*$ | |
| | z | $SS*$ | $S \rightarrow SS$ |
| | z | $(S)S*$ | $S \rightarrow (S)$ |
| $($ | z | $S)S*$ | |
| $($ | z | $(S))S*$ | $S \rightarrow (S)$ |
| $(($ | z | $S))S*$ | |
| $(($ | z | $))S*$ | $S \rightarrow \varepsilon$ |
| $(()$ | z | $)S*$ | |
| $(())$ | z | $S*$ | |
| $(())$ | z | $(S)*$ | $S \rightarrow (S)$ |
| $(())($ | z | $S)*$ | |
| $(())($ | z | $)*$ | $S \rightarrow \varepsilon$ |
| $(())()$ | z | $*$ | |
| $(())()$ | z_+ | $*$ | |

Es sei aber betont, dass dies sozusagen der „gute Fall“ ist. Der Kellerautomat ist nichtdeterministisch und könnte auch die „falschen“ Produktionsschritte machen:

| gelesene Eingabe | neuer Zustand | neuer Kellerinhalt | verwandte Produktion |
|---------------------|------------------|-----------------------|-------------------------|
| | z_0 | $*$ | |
| | z | $S*$ | $S \rightarrow SS$ |
| | z | $SS*$ | $S \rightarrow (S)$ |
| | z | $(S)S*$ | |
| $($ | z | $S)S*$ | $S \rightarrow (S)$ |
| $($ | z | $(S))S*$ | |
| $(($ | z | $S))S*$ | $S \rightarrow (S)$ |
| $(($ | z | $(S)))S*$ | |

Wenn die tatsächliche Eingabe $((()))$ ist, dann geht es an dieser Stelle nicht weiter, weil das nächste Eingabesymbol $)$ nicht gleich dem obersten Kellersymbol $($ ist.

4.29 (Top-down-Syntaxanalyse) Die eben beschriebene Vorgehensweise bezeichnet man auch als *Top-down-Syntaxanalyse*. Der Kellerautomat „konstruiert“ den Ableitungsbaum sozusagen von oben nach unten. Bezeichnet man den bereits gelesenen Teil der Eingabe mit α , den ausstehenden Teil von ω und den Kellerinhalt mit κ , dann lassen sich die zwischenzeitlich auftretenden Situationen unmittelbar vor einem Produktionsschritt so darstellen:



Den grau unterlegten Teil des Ableitungsbaumes hat der Kellerautomat schon gefunden, den weißen gilt es noch zu finden. Es gilt also stets $S \xRightarrow{\ell}^* \alpha\kappa$ und es ist noch zu überprüfen ob auch $\kappa \xRightarrow{\ell}^* \omega$ und damit $S \xRightarrow{\ell}^* \alpha\omega$ gilt.

Im Hinblick auf Syntaxanalyse stellen wir uns im folgenden Kellerautomaten so vor, dass sie in jedem Schritt auch eine endliche Ausgabe produzieren dürfen. Dies kann z. B. die Produktion (der Grammatik) sein, die bei der Top-down-Analyse gerade angewendet wurde.

4.30 Bei der Top-down-Syntaxanalyse erzeugt der Kellerautomat eine Linksableitung.

4.31 Im Hinblick auf die Anwendung in einem konkreten Übersetzer, der als deterministischer Algorithmus implementiert ist, sind *nichtdeterministische* Kellerautomaten natürlich nicht unmittelbar übertragbar.

Mehrere Auswege sind denkbar:

- Man simuliert den Kellerautomaten, indem man eine systematische Suche nach einer akzeptierenden Berechnung durchführt. Dabei ist aber zunächst einmal große Vorsicht geboten, damit sich der Algorithmus nicht in einer unendlichen Rekursion „verläuft“.

Aber auch bei „richtiger“ Implementierung ist das kein besonders gut gangbarer Weg, denn erstens kann der Zeitaufwand immer noch so groß werden, dass er Benutzern nicht mehr zugemutet werden kann. Zweitens hat ein Übersetzer Aufgaben zu erledigen, die über pure Syntaxanalyse hinausgehen, aber vorteilhaft mit der Syntaxanalyse verzahnt werden. Dafür

werden zusätzliche Datenstrukturen benötigt, deren Aktualisierung bei Backtracking eine zumindest sehr unerfreuliche Aufgabe sein kann.

- Man schränkt sich bei den Kellerautomaten ein. Im Extremfall würde man nur die deterministische Variante zulassen.

Man kann auch unsere bisherige Definition von Kellerautomaten etwas aufweichen, indem man erlaubt, dass die Steuereinheit im Eingabestrom einige Zeichen vorausschaut und ihre aktuellen Arbeitsschritte von den „demnächst“ auftretenden Symbolen abhängig macht. Dies ist eine in der Praxis bedeutsame Vorgehensweise. Wir werden im nächsten Kapitel genauer darauf eingehen.

- Man vergisst alles über Kellerautomaten und macht die Syntaxanalyse mit „anderen“ Algorithmen. Auch hierzu werden wir in Kapitel 6 ein Beispiel kennenlernen.

Außerdem ist diese Vorgehensweise natürlich erzwungen, wenn man mit Programmiersprachen zu tun hat, bei denen die Syntax nicht kontextfrei ist. Das ist allerdings nur sehr selten der Fall.

4.32 Als Alternative zur Top-down-Syntaxanalyse wollen wir zum Abschluss dieses Abschnittes noch die generelle Vorgehensweise bei der sogenannte *Bottom-up-Syntaxanalyse* kennenlernen. Wieder beschränken wir uns hier auf den vom Konzept her einfach handzuhabenden Fall nichtdeterministischer Kellerautomaten.

Weshalb betrachten wir überhaupt Bottom-up-Syntaxanalyse, wenn man doch wieder nichtdeterministische Kellerautomaten braucht und trotzdem nur die gleichen, i. e. die kontextfreien Sprachen erkennen kann?

Der Grund ist, dass die Konzepte von Top-down- bzw. Bottom-up-Syntaxanalyse verschiedene Leistungsfähigkeit bekommen, wenn man die Einschränkung auf deterministische Kellerautomaten mit Vorausschau macht.

4.33 (Modifizierte Kellerautomaten) Für Bottom-up-Syntaxanalyse ist es bequem, von einer modifizierten Art von Kellerautomat auszugehen:

- Es ist praktisch sich vorzustellen, der Kellerautomat habe die Möglichkeit, in einem Schritt nicht nur das oberste Kellersymbol zu lesen, sondern eine größere (konstante) Anzahl.

Das kann man aber natürlich mit einem „normalen“ Kellerautomaten nachbilden, indem man die Zustandszahl vergrößert und der neue Kellerautomat sozusagen die obersten „Kellersymbole“ gar nicht im Keller, sondern in seiner Steuereinheit speichert.

Die Vorstellung, dass immer sofort mehrere Kellersymbole sichtbar sind, erleichtert aber die Beschreibung wesentlich. Deshalb gehen wir im folgenden immer davon aus, dass ein Kellerautomat das kann.

- Außerdem soll man sich im folgenden vorstellen, dass beim „Auslesen“ eines Wortes, dessen Symbole oben auf dem Keller liegen, das oberste Kellersymbol das letzte Wortsymbol ist, das zweitoberste Kellersymbol das vorletzte Wortsymbol, usw.

Die modifizierten Kellerautomaten werden nur so benutzt werden, dass immer nur einzelne (Terminal- oder Nichtterminal-)symbole eingekellert werden. Eine Verwechslung der Richtungen ist hier also ausgeschlossen.

4.34 Beispiel. Als Grammatik diene wieder $G = (\{S\}, \{(\,,\,)\}, S, \{S \rightarrow \varepsilon \mid (S) \mid SS\})$.

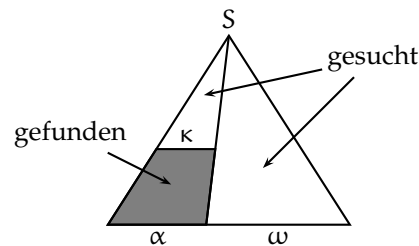
Nachfolgend ist für die Beispieleingabe $((()))$ eine mögliche Arbeitsweise eines modifizierten Kellerautomaten beschrieben. Man beachte, dass in der Tabelle im Unterschied zu Beispiel 4.28 der Kellerinhalt in der linken Spalte dargestellt ist (oberstes Kellersymbol rechts) und in der rechten Spalte der noch nicht gelesene Teil des Eingabewortes.

Bei einem *Leseschritt* wird das nächste Eingabesymbol gekellert. Bei einem *Reduktionsschritt* bildet eine Reihe oberster Kellersymbole die rechte Seite einer Produktion, die durch das Nichtterminalsymbol der zugehörigen linken Seite ersetzt wird:

| neuer Kellerinhalt | neuer Zustand | noch nicht ge- lesene Eingabe | verwandte Produktion |
|-----------------------|------------------|----------------------------------|-----------------------------|
| * | z_0 | $((()))$ | |
| * (| z | $((()))$ | |
| * ((| z | $((()))$ | |
| * ((S | z | $((()))$ | $S \rightarrow \varepsilon$ |
| * ((S) | z | $((()))$ | |
| * (S | z | $((()))$ | $S \rightarrow (S)$ |
| * (S) | z | $((())$ | |
| * S | z | $((())$ | $S \rightarrow (S)$ |
| * S(| z | $((())$ | |
| * S(S | z | $((())$ | $S \rightarrow \varepsilon$ |
| * S(S) | z | $((())$ | |
| * SS | z | $((())$ | $S \rightarrow (S)$ |
| * S | z | $((())$ | $S \rightarrow SS$ |
| * S | z_+ | | |

Der Kellerautomat geht in einen akzeptierenden Zustand, wenn im Keller nur noch das Startsymbol der Grammatik steht.

4.35 (Bottom-up-Syntaxanalyse) Im vorangegangenen Punkt handelt es sich um ein Beispiel für *Bottom-up-Syntaxanalyse*. Der Kellerautomat versucht, eine Ableitung des Eingabewortes gemäß der zu Grunde gelegten Grammatik zu finden. Er beginnt bei der Konstruktion sozusagen „unten links“ im Ableitungsbaum, wie nachfolgend dargestellt:



Bezeichnen α und ω wieder bereits gelesenen und noch zu lesenden Teil des Eingabewortes und κ den momentanen Kellerinhalt des Automaten, dann ist jederzeit sichergestellt, dass gilt: $\kappa \xrightarrow{r}^* \alpha$. Was noch fehlt ist eine Ableitung $S \xrightarrow{r}^* \kappa\omega$.

Der Kellerautomat arbeitet wie folgt:

Leseschritt: Solange oben auf dem Keller nicht die rechte Seite einer Produktion liegt, wird ein weiteres Symbol aus der Eingabe gelesen und gekellert.

Reduktionsschritt: Wenn oben auf dem Keller die rechte Seite mindestens einer Produktion liegt, dann *kann* der Automat sie durch das zugehörige Nichtterminalsymbol der linken Seite ersetzen.

Es sollte klar sein, dass durch eine Leseschritt die Eigenschaft $\kappa \xrightarrow{r}^* \alpha$ nicht zerstört wird, denn ein Leseschritt bedeutet einfach das Anhängen des gelesenen Terminalsymbols sowohl an α als auch an κ .

Bei einem Reduktionsschritt wird die Eigenschaft ebenfalls erhalten: Ist nämlich $\kappa = \kappa'\gamma$ und ist $X \rightarrow \gamma$ die Produktion, die zur Reduktion des oberen (rechten) Endes des Kellers benutzt wird, dann gilt ja gerade $\kappa'X \xrightarrow{r} \kappa'\gamma = \kappa \xrightarrow{r}^* \alpha$.

Wenn es dem Kellerautomaten also gelingt, in die Situation zu kommen, dass $\kappa = S$ das Startsymbol und die gesamte Eingabe w gelesen (also $\omega = \varepsilon$) ist, dann ist also tatsächlich eine Rechtsableitung $S \xrightarrow{r}^* \alpha = w$ möglich.

Und es ist auch klar, dass bei möglicher Rechtsableitung gemäß der Grammatik der Kellerautomat sie finden kann.

4.36 Man beachte, dass der Kellerautomat bei Bottom-up-Syntaxanalyse die Produktionen einer Rechtsableitung „von hinten nach vorne“ findet.

4.37 Bei der Bottom-up-Syntaxanalyse hat der Kellerautomat unter Umständen mehrere „Freiheitsgrade“:

- Es kann sowohl ein Lese- als auch ein Reduktionsschritt möglich sein.
- Bei einem Reduktionsschritt können unterschiedlich lange Kellerenden reduziert werden können.
- Bei einem Reduktionsschritt kann unter mehreren Produktionen mit gleicher rechter Seite ausgewählt werden können.

Wenn man einen deterministischen Algorithmus will, muss man also die Grammatik geeignet konstruieren.

Eine weitergehende Beschreibung von Verfahren für die Syntaxanalyse (mancher) kontextfreier Sprachen wird in Kapitel 6 gegeben werden.

5 Typ 1 und Typ 0

5.1 Kontextsensitive Grammatiken

Da kontextsensitive Sprachen für den weiteren Verlauf dieser Vorlesung nicht von besonderer Bedeutung sind, fassen wir uns in diesem Abschnitt *extrem* kurz.

5.1 Definition. Eine *Typ-1-Grammatik* (T1G) oder auch *kontextsensitive Grammatik* ist eine Grammatik $G = (N, T, X_0, P)$, deren Produktionen alle von einer der folgenden Formen sind:

- $uXv \rightarrow uvw$ mit $u, v \in V^*$, $w \in V^+$ und $X \in N$ oder
- $X_0 \rightarrow \varepsilon$. Falls diese Produktion existiert, kommt aber X_0 in keiner Produktion auf der rechten Seite vor.

Eine formale Sprache ist vom Typ 1 oder kontextsensitiv, wenn es eine T1G gibt, die sie erzeugt.

5.2 Wie man sieht, erlaubt eine kontextsensitive Produktion $uXv \rightarrow uvw$ „im Effekt“, ein Nichtterminalsymbol X durch ein Wort w zu ersetzen. Im Unterschied zu kontextfreien Grammatiken ist das aber nicht immer möglich, sondern nur, wenn das X in einem gewissen *Kontext* vorkommt: links von X muss u stehen und rechts davon v , sonst ist die Produktion nicht anwendbar.

5.3 Wie schon für Typ-3- und Typ-2-Grammatiken gibt es auch hier wieder ein Automatenmodell, mit dem man genau die Typ-1-Sprachen erkennen kann. Diese sogenannten *linear beschränkten Automaten* sind ein leicht zu definierender Spezialfall von Turingmaschinen, denen wir uns im nächsten Abschnitt widmen werden. Wir holen die Definition daher dort in Punkt 5.21 nach.

5.4 Der Vollständigkeit halber erwähnen wir, dass die Sprache $L = \{0^k 1^k 2^k \mid k \in \mathbb{N}\}$, von der in Punkt 4.12 mitgeteilt worden war, dass sie nicht kontextfrei ist, kontextsensitiv ist. Andererseits ist jede kontextfreie Sprache kontextsensitiv.

Mit kontextsensitiven Grammatiken kann man also echt mehr formale Sprachen erzeugen als mit kontextfreien. Es gibt aber auch formale Sprachen, die nicht kontextsensitiv (sondern noch „schwieriger“) sind.

5.2 Turingmaschinen

Wir erweitern nun Kellerautomaten im wesentlichen dadurch, dass der Keller, auf den nur in eingeschränkter Weise zugegriffen werden kann, durch einen Speicher mit wahlfreiem Zugriff ersetzt wird. Das resultierende Modell heißt Turingmaschine. Von Turingmaschinen werden heute üblicherweise diverse Varianten betrachtet. Wir beschränken uns auf einen einfachen, in Abbildung 5.1 dargestellten Fall.

5.5 Turingmaschinen sind nach dem bedeutenden englischen Mathematiker Alan Mathison Turing¹ (1912 – 1954) benannt, der sie 1936 in seiner bahnbrechenden Arbeit „*On Computable Numbers, with an application to the Entscheidungsproblem*“ (Proceedings of the London Mathematical Society, Band 42, Heft 2, S. 230–265) eingeführt hat.

¹<http://www-groups.dcs.st-andrews.ac.uk/~history/Mathematicians/Turing.html>

5.6 Definition. Eine *Turingmaschine* (TM) besteht aus einer endlichen Steuereinheit und einem unendlichen Arbeitsband, das in eine Folge von Feldern aufgeteilt ist, auf denen jeweils ein Symbol des Bandalphabetes gespeichert ist. Über einen Schreib-Lese-Kopf hat die TM Zugriff auf jeweils ein Feld des Bandes. Der Kopf kann in jedem Schritt um ein Feld weiter gerückt werden.

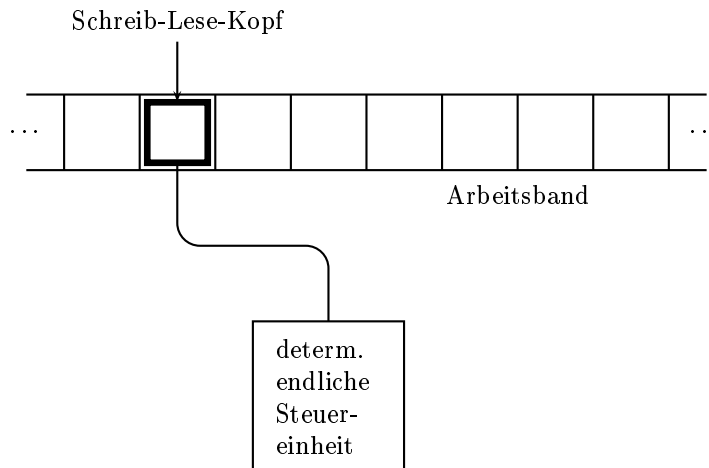


Abbildung 5.1: Eine Turingmaschine mit einem Band und einem Kopf

Formal ist eine Turingmaschine festgelegt durch

- eine endliche Zustandsmenge Z ,
- einen Anfangszustand $z_0 \in Z$,
- ein Bandalphabet Y ,
- ein Blanksymbol $\square \in Y$,
- ein Eingabealphabet $X \subseteq Y$,
- eine nichtleere Menge $F_+ \subseteq Z$ akzeptierender Endzustände und eine nichtleere Menge $F_- \subseteq Z \setminus F_+$ ablehnender Endzustände und
- eine Überföhrungsfunktion $f : Z \times Y \rightarrow Z \times Y \times \{-1, 0, 1\}$.

Wir verlangen, dass für alle $z \in F_+ \cup F_-$ und alle $y \in Y$ gilt: $f(z, y) = (z, y, 0)$, d. h. wenn eine TM einen (akzeptierenden oder ablehnenden) Endzustand erreicht hat, dann verläßt sie ihn nicht mehr, ändert nicht mehr die Bandbeschriftung und bewegt den Kopf nicht mehr. Die TM „hält an“.

5.7 Definition. Bei der Überföhrungsfunktion bedeutet $f(z, y) = (z', y', d)$, dass die TM, wann immer sie sich in Zustand z befindet und auf dem Band Symbol y liest, in Zustand z' übergeht, auf das Band Symbol y' schreibt und den Kopf um d Felder nach rechts bewegt.

Eine Berechnung für ein Eingabewort w beginnt so, dass auf $|w|$ nebeneinander liegenden Feldern die Symbole von w der Reihe nach notiert sind, alle anderen Felder „leer“, d. h. mit \square beschriftet sind, der Kopf auf dem ersten Symbol von w steht und die Steuereinheit in Zustand z_0 ist.

Wenn die Berechnung für ein Eingabewort w dazu führt, dass die TM in einen akzeptierenden Endzustand übergeht, dann gilt w als akzeptiert. Wenn die Berechnung für ein Eingabewort w dazu führt, dass die TM in einen ablehnenden Endzustand übergeht, dann gilt w als abgelehnt. Solange keines von beidem geschehen ist, „arbeitet die TM weiter“.

Die von einer TM erkannte Sprache ist die Menge aller Wörter $w \in X^*$, bei deren Eingabe die TM irgendwann in einen akzeptierenden Endzustand übergeht.

5.8 Formal ist das Band zweiseitig unendlich. Aus oben Gesagtem ergibt sich aber, dass zu jedem Zeitpunkt nur ein endlicher Abschnitt „interessant“ ist, in dem nicht alle Felder mit \square beschriftet sind. Und wenn eine Turingmaschine nach endlich vielen Schritten hält, dann hat sie nur endlich viele Felder besucht. Der tatsächlich benutzte Speicherbereich ist also endlich.

5.9 Beispiel. Wir beschreiben eine Turingmaschine zur Erkennung der formalen Sprache aller Palindrome (vgl. auch Beispiel 4.9) über dem Alphabet $X = \{a, b\}$. Wir benutzen die Zustandsmenge $Z = \{r, r_a, r_b, l, l_a, l_b, f_+, f_-\}$ mit Anfangszustand r und den Endzustandsmengen $F_+ = \{f_+\}$ und $F_- = \{f_-\}$. Als Bandalphabet genügt $Y = X \cup \{\square\}$.

Die Überföhrungsfunktion ist durch die folgende Tabelle gegeben:

| alter Zustand | gelesenes Symbol | neuer Zustand | neues Symbol | Kopf-bewegung | Bemerkung |
|---------------|------------------|---------------|--------------|---------------|-----------------------------------|
| r | a | r_a | \square | +1 | Symbol am linken |
| r | b | r_b | \square | +1 | Ende merken |
| r | \square | f_+ | \square | 0 | Palindrom gerader Länge erkannt |
| r_a | a | r_a | a | +1 | erstes Blanksymbol |
| r_a | b | r_a | b | +1 | rechts der Eingabe suchen |
| r_a | \square | l_a | \square | -1 | |
| r_b | a | r_b | a | +1 | erstes Blanksymbol |
| r_b | b | r_b | b | +1 | rechts der Eingabe suchen |
| r_b | \square | l_b | \square | -1 | |
| l_a | a | l | \square | -1 | Symbole gleich |
| l_a | b | f_- | b | 0 | Symbole ungleich |
| l_a | \square | f_+ | \square | 0 | Palindrom ungerader Länge erkannt |
| l_b | a | f_- | a | 0 | Symbole ungleich |
| l_b | b | l | \square | -1 | Symbole gleich |
| l_b | \square | f_+ | \square | 0 | Palindrom ungerader Länge erkannt |
| l | a | l | a | -1 | erstes Blanksymbol |
| l | b | l | b | -1 | links der Eingabe suchen |
| l | \square | r | \square | +1 | |

In Abbildung 5.2 ist die Arbeit der zugehörigen Turingmaschine für das Eingabewort *abbba* dargestellt.

Für f_+ und f_- sind in der Tabelle keine Aktionen spezifiziert, da Definition 5.6 ohnehin eindeutig festlegt, wie sie auszusehen haben.

5.10 Definition. Eine formale Sprache L heißt *rekursiv*, wenn es eine TM gibt, die für jedes $w \in L$ als Eingabe irgendwann in einen akzeptierten Endzustand übergeht und für jedes $w \notin L$ als Eingabe irgendwann in einen ablehnenden Endzustand übergeht.

Eine formale Sprache L heißt *rekursiv aufzählbar*, wenn es eine TM gibt, die für jedes $w \in L$ als Eingabe, und auch nur für diese Wörter, irgendwann in einen akzeptierten Endzustand übergeht. Für Eingaben $w \notin L$ wird nur gefordert, dass sie nicht akzeptiert werden; die TM darf aber irgendwann in einen ablehnenden Endzustand übergehen oder *unendlich arbeiten ohne je zu halten*.

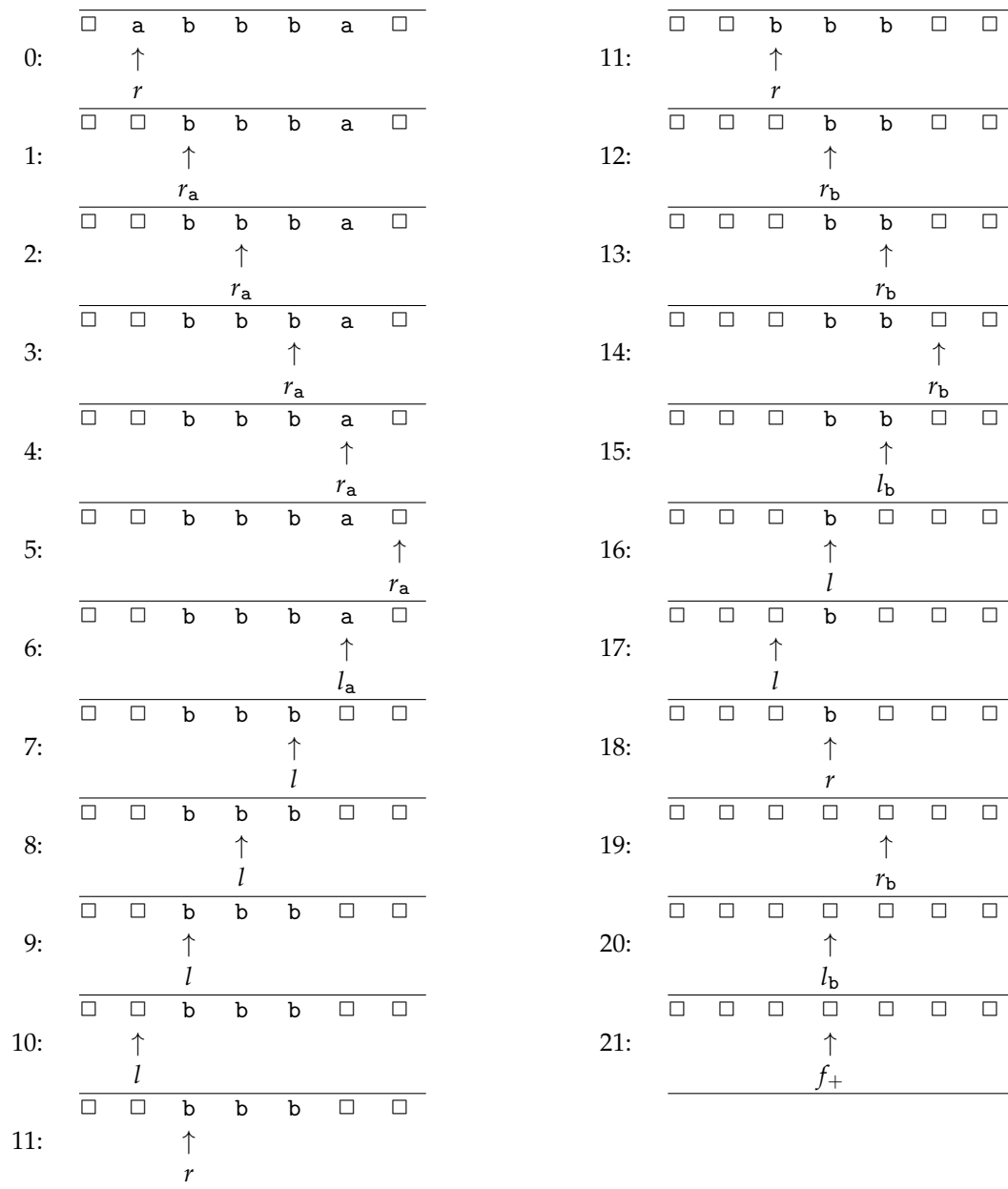


Abbildung 5.2: Überprüfung eines Palindromes mit der Turingmaschine aus Beispiel 5.9.

5.11 Wie man sieht, ist jede rekursive Sprache L auch rekursiv aufzählbar. Wenn L rekursiv ist, dann ist außerdem auch das Komplement $\bar{L} = X^* \setminus L$ rekursiv aufzählbar.

Man kann zeigen, dass auch die Umkehrung gilt: Wenn sowohl eine Sprache L als auch ihr Komplement \bar{L} rekursiv aufzählbar sind, dann ist L rekursiv.

Daraus folgt: Wenn eine Sprache L rekursiv ist, dann ist auch ihr Komplement \bar{L} rekursiv.

Es gibt aber formale Sprachen L , die zwar rekursiv aufzählbar sind, aber ihr Komplement nicht, d. h. solche L sind zwar rekursiv aufzählbar aber nicht rekursiv. Wir werden das später anhand eines Beispiels in Satz 5.34 sehen.

5.12 Grammatiken wie in Definition 2.34 ganz allgemein eingeführt bezeichnet man auch als *Typ-0-Grammatiken* (T0G).

5.13 Satz. Eine formale Sprache kann genau dann von einer Typ-0-Grammatik erzeugt werden, wenn sie rekursiv aufzählbar ist.

5.14 Ein Beweis dieses Satzes ist im Prinzip nicht sooo schwer. Wir geben kurz zwei Hinweise:

- Wenn ein Wort w von einer Typ-0-Grammatik erzeugt wird, kann man das algorithmisch dadurch feststellen, dass man systematisch erst alle Ableitungsfolgen der Länge 1, dann alle Ableitungsfolgen der Länge 2, dann alle Ableitungsfolgen der Länge 3 usw. erzeugt. Irgendwann wird bei einer Ableitung am Ende w erzeugt. Das ist offensichtlich eine, wenn auch nicht schnelle, Methode, die auch von einer TM angewendet werden kann.

Man beachte aber: Wenn ein Wort nicht von der Grammatik erzeugt wird, dann werden ewig Ableitungsfolgen erzeugt, die alle nicht zu w führen. Wegen des zweiten Teiles von Punkt 5.11 hat man auch überhaupt keine Chance, für hinreichend unangenehme Grammatiken eine bessere Vorgehensweise zu finden, bei der man im Fall $w \notin L$ nach endlich vielen Schritten abbrechen könnte.

- Für die umgekehrte Richtung des Satzes: Was hat wohl z. B. $f(z, y) = (z', y', +1)$ mit der Produktion $zy \rightarrow y'z'$ zu tun? Und $f(z, y) = (z', y', -1)$ mit $\bar{y}zy \rightarrow z'\bar{y}y'$?

5.15 Jeder (der Hörer dieser Vorlesung jedenfalls) hat ein gewisse intuitive Vorstellung davon, was ein *Algorithmus* ist und was nicht. Wesentliche Punkte sind, dass man

- eine *endliche Beschreibung* vorliegen hat, die in
- *elementare Schritte* zerfällt, die offensichtlich intuitiv berechenbar sind.
- Die Abarbeitung des Algorithmus soll *schrittweise* erfolgen und
- nach jedem Schritt soll *eindeutig* der als nächstes durchzuführende Schritt festliegen.
- Der Algorithmus soll für *beliebig große Eingaben* (also unendlich viele) die richtigen Ergebnisse produzieren.
- Der Algorithmus soll für jede Eingabe *nach endlich vielen Schritten halten*.

(Außerdem gibt es unter Umständen – sogar gute – Gründe, hier oder da vielleicht doch etwas weniger zu fordern. Darauf gehen wir hier aber nicht weiter ein.)

Wenn eine Funktion durch einen Algorithmus dieser Art berechnet werden kann, könnte man die Funktion *intuitiv berechenbar* nennen. Dies ist aber offensichtlich keine mathematisch harte Definition des Berechenbarkeitsbegriffes.

5.16 Mathematisch hart ist zum Beispiel die Definition von Turingmaschinen. Daneben hat man im Laufe der Jahrzehnte noch viele zum Teil deutlich andere Formalismen untersucht. Als Beispiele seien Registermaschinen, μ -rekursive Funktionen und Markov-Algorithmen genannt. Von all diesen Konzepten hat sich herausgestellt, dass sie zu Turingmaschinen äquivalent sind.

Das kann man zum Anlass nehmen, die folgende These aufzustellen:

5.17 (These von Church-Turing) Alles, was intuitiv berechenbar ist, ist auch von einer Turingmaschine berechenbar.

5.18 Diese Aussage kann man nicht in einem strengen Sinn beweisen, denn der Begriff der intuitiven Berechenbarkeit ist ja gerade kein formaler. Man könnte die These eventuell widerlegen, wenn man ein Problem fände, das zwar von keiner TM gelöst werden kann, von dem aber (hinreichend viele) Leute sagen, dass es intuitiv berechenbar sei.

Allerdings wäre das wegen der in Punkt 5.16 erwähnten Tatsache doch recht erstaunlich.

Andererseits gibt es in jüngster Vergangenheit durchaus Ansätze (und zwar nicht nur gedanklicher, sondern auch physikalischer Art), die unter Umständen einmal darauf hinauslaufen könnten, dass man bewusst vom bisher üblichen intuitiven Algorithmusbegriff abgeht.

Wenn man schon einmal versucht hat, zur Lösung etlicher verschiedener Probleme Algorithmen zu entwerfen, hat man wohl den Eindruck bekommen, dass das erstens je nach Problem unterschiedlich schwierig ist und dass zweitens Algorithmen für verschiedene Probleme unterschiedlich lange für gleich große Eingaben brauchen. Zum Beispiel ist nicht sehr schwer die Summe von zwei n -stelligen Zahlen in einer Zeit proportional zu n zu berechnen. Aber für das Produkt zweier solcher Zahlen haben sie vermutlich keinen ebenso schnellen Algorithmus gefunden.

Tatsächlich kann man „harte Modelle“ wie zum Beispiel Turingmaschinen benutzen, um zu *beweisen*, dass es Probleme gibt, für die man (z.B.) mehr Zeit in die algorithmische Lösung investieren *muss* als bei anderen. Entsprechendes gilt auch für den nötigen Arbeitsspeicher.

5.19 Man kann die von einem Algorithmus „verrichtete Arbeit“ quantifizieren, indem man den Verbrauch von „Ressourcen“ misst. Die wichtigsten Ressourcen sind die erforderliche Rechenzeit und der benötigte Speicherplatz. Sie hängen im allgemeinen natürlich davon ab, welche konkrete Eingabe zu verarbeiten ist.

Es ist allerdings üblich, keine so feine Unterscheidung zu treffen, sondern den Aufwand nur in Abhängigkeit von der *Größe der Eingabe* anzugeben. Bei der Erkennung formaler Sprachen ist zum Beispiel die Länge eines Eingabewortes seine Größe.

Die Rechenzeit (oder der Speicherplatzbedarf) wird ausgedrückt durch eine Funktion $f : \mathbb{N} \rightarrow \mathbb{N}$; $f(n)$ gibt dann an, wieviele Schritte (bzw. Speicherworte) der Algorithmus für Eingaben der Größe n benötigt.

5.20 Definition. Eine Turingmaschine ist $t(n)$ -*zeitbeschränkt*, wenn für alle $n \in \mathbb{N}$ gilt, dass sie für Eingabewörter w der Länge n höchstens $t(n)$ Schritte macht bis sie anhält.

Eine Turingmaschine ist $s(n)$ -*raumbeschränkt* oder $s(n)$ -*platzbeschränkt*, wenn für alle $n \in \mathbb{N}$ gilt, dass sie für Eingabewörter w der Länge n höchstens $s(n)$ Felder auf dem Arbeitsband besucht bis sie anhält.

5.21 Definition. Ein *linear beschränkter Automat* ist eine Turingmaschine, die $n + 2$ -platzbeschränkt ist. M. a. W. können also nur die n Felder, auf denen die Eingabesymbole stehen und (um die Enden der Wörter zu erkennen) jeweils das erste Feld links und rechts daneben besucht werden.

Man kann zeigen, dass linear beschränkte Automaten das Modell sind, das genau zu Typ-1-Grammatiken „passt“.

5.22 Man kann sowohl für die Zeit- als auch für Speicherplatzbeschränkungen zeigen, dass Ergebnisse der folgenden Art gelten: Wenn $f_1(n)$ und $f_2(n)$ zwei „hinreichend nette“ Funktionen sind, derart

dass $f_1(n)$ „etwas schwächer“ wächst als $f_2(n)$, dann kann man mit Turingmaschinen, deren Zeit- oder Platzbedarf durch $f_1(n)$ beschränkt ist, *echt* weniger formale Sprachen erkennen als mit Turingmaschinen, deren Zeit- oder Platzbedarf durch $f_2(n)$ beschränkt ist. Was „hinreichend nett“ ist und was „etwas schwächer“, kann man präzise definieren; für den hier gegebenen Überblick ist das aber entbehrlich.

Beispiele für solche Funktionen sind etwa n und n^2 , oder auch n^2 und n^3 , usw., allgemeiner $\lfloor n^x \rfloor$ und $\lfloor n^y \rfloor$ für beliebige rationale Exponenten $1 < x < y$.

Untersuchungen dieser Art sind Gegenstand der Komplexitätstheorie (engl. *computational complexity*).

- 5.23** Das bedeutet zum Beispiel, dass Turingmaschinen, die für Wörter der Länge n Speicherplatzbedarf n^2 haben, *echt* mehr rekursive formale Sprachen erkennen können als Turingmaschinen, die Speicherplatzbedarf $n + 2$ haben.

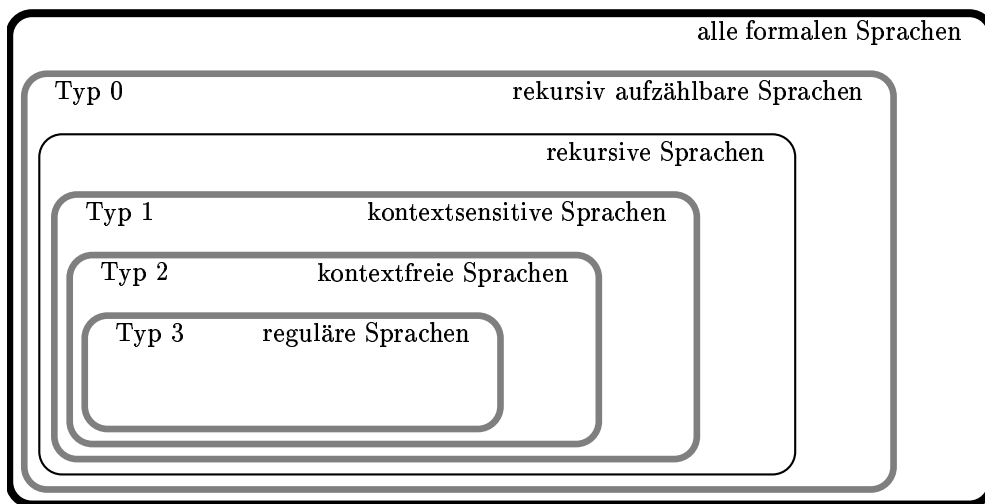
Hieraus folgt durch weitere Überlegungen auch, dass es rekursive Sprachen gibt, die nicht kontextsensitiv sind.

Ein Kardinalitätsargument wie wir es schon in Punkt 2.16 skizziert hatten, um zu zeigen, dass nicht alle formalen Sprachen von endlichen Automaten erkennbar sind, liefert auch die analoge Aussage für Turingmaschinen:

- 5.24 Satz.** *Es gibt formale Sprachen, die nicht rekursiv aufzählbar sind.*

Wir werden im nächsten Abschnitt noch konkrete Beispiele solcher Sprachen kennenlernen und sehen, dass es sich bei nicht entscheidbaren Problemen um sehr praxisrelevante Aufgaben handeln kann.

- 5.25** Zusammenfassend ergibt sich für die Welt der formalen Sprachen also das folgende Bild. Mit grauen Rahmen sind die vier Typen gekennzeichnet, für die wir eine Charakterisierung durch eine Art von Grammatiken kennengelernt haben. Man spricht diesbezüglich auch von der *Chomsky-Hierarchie*, benannt nach dem amerikanischen Linguisten Noam Chomsky.



Alle Inklusionen in der Abbildung sind *echt*: Es gibt immer eine (sogar unendlich viele) formale Sprachen, die in einer Klasse enthalten sind, aber nicht in der nächst kleineren.

5.3 Jenseits von Typ 0

5.26 In diesem Abschnitt wird es nützlich sein, davon reden zu können, dass eine Turingmaschine als Eingabe nicht nur ein Wort sondern „mehrere“ bekommt. Zur Präzisierung sei folgendes vereinbart.

- Das Bandalphabet umfasst neben den „eigentlichen“ Eingabesymbolen auch die davon verschiedenen Zeichen „Klammer auf“ $[$, „Semikolon“ $;$ und „Klammer zu“ $]$.
- Die Eingabe von k Argumenten w_1, \dots, w_k geschehe durch die Beschriftung des Bandes mit $[w_1; \dots; w_k]$ (umgeben von lauter Blanksymbolen, Kopf auf dem $[$ -Symbol).
- Möchte man von einer Turingmaschine eine Ausgabe $f(w_1, \dots, w_k)$, die über ein Bit hinausgeht, kann man vereinbaren, dass auf Eingabe $[w_1; \dots; w_k]$ nach endlich vielen Schritten ein Endzustand erreicht wird und auf dem nur noch $[f(w_1, \dots, w_k)]$ steht. (Die Unterscheidung zwischen akzeptierenden und ablehnenden Endzuständen ist dann natürlich hinfällig.) In einem solchen Fall schreiben wir auch $T(w_1, \dots, w_k) = f(w_1, \dots, w_k)$.

Eine Funktion heißt (*Turing-*)*berechenbar*, wenn es eine Turingmaschine gibt, die sie in obigem Sinne berechnet.

5.27 **Übung.** Konstruieren Sie eine Turingmaschine T , die Zahlen in Binärdarstellung addiert. Wenn z. B. die Anfangsbandbeschriftung $[101; 1001]$ lautet, dann soll am Ende auf dem Band $[1110]$ stehen. Mit anderen Worten soll sozusagen gelten: $T(w_1, w_2) = w_1 + w_2$.

5.28 Die Idee der *Gödelisierung* ist nach dem bedeutenden tschechischen Mathematiker Kurt Gödel² (1906 – 1978) benannt.

Dabei handelt es sich darum, z. B. *jede* Turingmaschine so mit einer natürlichen Zahl als Nummer zu versehen, dass man von einer Nummer feststellen kann, ob sie eine Turingmaschine bezeichnet, und gegebenenfalls aus der Nummer die Turingmaschine rekonstruieren kann, und das so einfach ist, dass man es sogar algorithmisch (also z. B. mit einer Turingmaschine) durchführen kann.

Das ist tatsächlich möglich.

Wir gehen davon aus, dass von nun an eine Gödelisierung aller Turingmaschinen festgelegt ist. Für die Turingmaschine mit einer Nummer x schreiben wir auch T_x .

5.29 **Definition.** Eine *universelle Turingmaschine* U ist eine Turingmaschine mit folgenden Eigenschaften:

- Eingabealphabet ist $\{0, 1, [,], ;\}$.
- Für jede Eingabe von der Form $[w_0; w_1; \dots; w_k]$
- überprüft U zunächst, ob w_0 die Nummer einer Turingmaschine T_{w_0} ist.
- Wenn das der Fall ist, berechnet U als Funktionswert $T_{w_0}(w_1, \dots, w_k)$.

5.30 **Satz.** *Es gibt eine universelle Turingmaschine.*

Für den Beweis hätten wir als erstes eine Gödelisierung aller Turingmaschinen fixieren müssen. Da wir schon das nicht gemacht haben, können wir auch keinen Beweis für Satz 5.30 geben.

5.31 Im folgenden sprechen wir statt von formalen Sprachen auch von *Problemen*. Inhaltlich ist das aber das Gleiche: Für ein Problem P besteht die Aufgabe immer darin, für jede Eingabe z. B. der Form $[w]$ (oder $[w_1; w_2]$, oder ...) festzustellen, ob eine bestimmte Eigenschaft E erfüllt ist oder nicht.

Man kann mit P naheliegenderweise die formale Sprache $L(P)$ aller Eingaben, die E erfüllen, identifizieren.

²<http://www-groups.dcs.st-andrews.ac.uk/~history/Mathematicians/Godel.html>

5.32 Definition. Ein Problem P heißt *entscheidbar*, wenn $L(P)$ rekursiv ist (siehe Definition 5.10), wenn es also eine Turingmaschine gibt, die alle Eingaben, die zu $L(P)$ gehören, akzeptiert und die alle Eingaben, die nicht zu $L(P)$ gehören, ablehnt.

5.33 Definition. Das *Halteproblem* H für Turingmaschinen besteht darin, für zwei Argumente x und y festzustellen, ob T_x für Eingabe y hält oder nicht.

Es ist also $L(H) = \{[x; y] \mid x \text{ ist Nummer einer TM und } T_x \text{ hält für Eingabe } y\}$.

Das *Selbstanwendungsproblem* K für Turingmaschinen besteht darin, für ein Argument x festzustellen, ob T_x bei Eingabe der eigenen Nummer x hält oder nicht.

5.34 Satz. $L(H)$ und $L(K)$ sind rekursiv aufzählbar.

Aber: Das Halteproblem und das Selbstanwendungsproblem für Turingmaschinen sind unentscheidbar, d. h. $L(H)$ und $L(K)$ sind nicht rekursiv.

5.35 Beweis. Dass $L(H)$ rekursiv aufzählbar ist, kann man z. B. so einsehen: Eine entsprechende Turingmaschine T wird für Eingabe $[x; y]$ zunächst wie eine universelle Turingmaschine U für die gleiche Eingabe arbeiten. U hält genau dann, wenn T_x für Eingabe y hält. Ist das der Fall, dann akzeptiert T die Eingabe.

Ähnlich zeigt man, dass K rekursiv aufzählbar ist.

Angenommen, das Halteproblem wäre entscheidbar. Dann wäre auch das Selbstanwendungsproblem entscheidbar. Folglich gäbe es auch eine Turingmaschine T , die das Komplement \bar{K} erkennt, die also genau für diejenigen Eingaben w hält und sie akzeptiert, die nicht in $L(K)$ sind, für die also gilt: w ist nicht die Nummer einer TM oder T_w hält nicht für die Eingabe w . Daraus kann man eine Turingmaschine T' konstruieren, die die gleichen Eingaben akzeptiert und für alle anderen Eingaben nicht hält.

Folglich gilt: T' hält für eine Turingmaschinenummer w (und akzeptiert sie) genau dann, wenn T_w nicht für die Eingabe w hält.

Es sei v die Nummer dieser Turingmaschine T' . Dann gilt also: T_v hält für eine Turingmaschinenummer w (und akzeptiert sie) genau dann, wenn T_w nicht für die Eingabe w hält.

Insbesondere: T_v hält für die Eingabe v (und akzeptiert sie) genau dann, wenn T_v nicht für die Eingabe v hält.

Widerspruch! Das kann nicht sein. Folglich war die Annahme, es gäbe eine Turingmaschine H , die das Halteproblem löst, falsch.

■

5.36 Definition. Das *Äquivalenzproblem* für Turingmaschinen besteht darin, für zwei Argumente x und y festzustellen, ob T_x und T_y für die gleichen Eingaben halten und gegebenenfalls für *alle* Eingaben das gleiche Ergebnis berechnen oder nicht.

5.37 Satz. Die Äquivalenz von Turingmaschinen ist unentscheidbar.

5.38 Beweis. Wir führen den Beweis indirekt. Aus der Annahme, das Problem sei entscheidbar, wird abgeleitet, dass dann auch das Halteproblem entscheidbar wäre, was im Widerspruch zu Satz 5.34 steht.

Angenommen es gäbe eine Turingmaschine A für das Äquivalenzproblem.

Es sei $[x; y]$ eine Eingabe, für die die Haltefrage zu beantworten ist. Dazu könnte man so vorgehen:

1. Man konstruiert eine TM T , die prüft, ob ihre einzige Eingabe genau y ist. Falls ja, hält sie. Falls nein, geht sie in eine Endlosschleife.

2. Man konstruiert eine TM M_1 , die erst wie T arbeitet und falls die hält, produziert M_1 die Ausgabe 1.
3. Man konstruiert eine TM M_2 , die erst wie T arbeitet. Falls die hält, arbeitet M_2 wie T_x . Falls auch die hält, produziert M_2 die Ausgabe 1.

Wie man sieht halten M_1 und M_2 beide nicht für alle Eingaben ungleich y . Für Eingabe y produziert M_1 Ausgabe 1. Für Eingabe y produziert M_2 Ausgabe 1, falls T_x für y hält, und keine Ausgabe sonst. Also sind M_1 und M_2 genau dann äquivalent, wenn T_x für y hält.

Wäre die Äquivalenz von Turingmaschinen entscheidbar, dann also auch das Halteproblem. Da letzteres nicht der Fall ist, ist der Satz bewiesen. ■

5.39 Wir führen ohne Beweis einige weitere unentscheidbare Probleme auf:

- Gegeben ein Java-Programm P , eine Eingabe w dafür und eine Zeilennummer n des Programms. Frage: Erreicht P für die Eingabe w jemals Zeile n ?
- Gegeben ein Java-Programm P und eine Zeilennummer n des Programms. Frage: Gibt es eine Eingabe für die P jemals Zeile n erreicht?
- Gegeben eine Turingmaschine T . Frage: Hält T für mindestens eine Eingabe?
- Gegeben eine Turingmaschine T . Frage: Ist $L(T)$ regulär?

Als letztes noch ein Beispiel einer nicht berechenbaren Funktion.

5.40 Definition. Ein *fleißiger Biber* ist eine Turingmaschine mit Bandalphabet $\{\square, 1\}$, die, wenn man sie mit einem anfangs völlig leeren Band startet, eventuell einige 1 auf das Band schreibt und danach hält.

Die *Busy-beaver-Funktion* ist eine einstellige Funktion $BB : \mathbb{N} \rightarrow \mathbb{N}$. Dabei ist $BB(n)$ die maximale Anzahl von 1 sein, die ein fleißiger Biber mit n Nichtend-Zuständen und einem Endzustand auf ein anfangs völlig leeres Band schreibt.

5.41 Wenn man einmal spaßhalber versucht, einen möglichst fleißigen Biber zu konstruieren, hat man zunächst nicht den Eindruck, dass $BB(n)$ besonders groß werden kann. Das ist ein Irrtum. Diese Funktion wächst nämlich schneller als jede berechenbare Funktion.

Die folgende Tabelle enthält einige Hinweise auf Werte von $BB(n)$ für kleine n :

| n | 1 | 2 | 3 | 4 | 5 | 6 |
|---------|---|---|---|----|-------------|-----------------------------|
| $BB(n)$ | 1 | 4 | 6 | 13 | ≥ 4098 | $\geq 3.5 \cdot 10^{18276}$ |

Die *Busy-beaver-Funktion* ist nicht berechenbar.

Derzeitiger Weltrekordhalter für die Konstruktion von Busy-beaver-Turingmaschinen (wie wir sie definiert haben) mit 6 Zuständen ist Pavel Kropitz. Interessenten können über die WWW-Seite http://www.drb.insel.de/~heiner/BB/simKro62_b.html zu weiteren Informationen finden.

Hier ist noch die Tabelle des fleißigen Bibers mit 6 Nichtendzuständen von Pavel Kropitz. Dabei stehen L und R für die Kopfbewegungen, A ist der Anfangszustand und Z ist der Endzustand:

| | A | B | C | D | E | F | Z |
|-----------|-------|-------|--------------|--------------|--------------|-------|-----|
| \square | $B1R$ | $C1R$ | $D1L$ | $E1R$ | $A1L$ | $Z1L$ | |
| 1 | $E1L$ | $F1R$ | $B\square R$ | $C\square L$ | $D\square R$ | $C1R$ | |

5.4 Ausblick

Neben den vorgestellten Automatenmodellen gibt es noch viele viele andere. Manche bestehen aus Ansammlungen vieler endlicher Automaten, zum Beispiel Zellularautomaten, andere operieren auf Registern, in denen man eine natürliche Zahl speichern kann oder größeren Ansammlungen solcher Automaten. Neben deterministischen und nichtdeterministischen Modellen werden auch sogenannte alternierende und stochastische und durch Quantenphysik inspirierte Varianten untersucht.

Genauso gibt es neben den Chomsky-Grammatik-Typen auch noch viele viele andere Erzeugungssysteme für formale Sprachen. Darunter befinden sich auch etliche, die durch Konzepte aus der Biologie motiviert sind, zum Beispiel Lindenmayer-Systeme, Membran-Systeme, usw.

6 Syntaxanalyse

Der Inhalt dieses Kapitels basiert auf den Büchern *Compilers — Principles, Techniques, and Tools* von Aho, Sethi und Ullman und *Compiler Design in C* von Holub sowie auf den Folien von Prof. Gerhard Goos zur Vorlesung *Übersetzerbau* an der Universität Karlsruhe, die im WWW unter http://i44www.info.uni-karlsruhe.de/~i44www/lehre/uebersetzerbau_WS00-Folien/ verfügbar sind, und der Vorlesung *Compiler* von Prof. Klaus Alber an der TU Braunschweig.

6.1 Einleitung

- 6.1 Ein Übersetzer hat die Aufgabe, einen als Eingabe vorliegenden Quelltext darauf hin zu analysieren, ob er ein korrekter Text in einer Sprache S ist, und gegebenenfalls daraus als Ausgabe einen Zieltext in einer Sprache T zu erzeugen. Dabei soll die Ausgabe in irgendeinem Sinne „äquivalent“ zur Eingabe sein.

Das Beispiel schlechthin für Übersetzer sind solche für Programme in einer höheren Programmiersprache, die Maschinencode für einen bestimmten Prozessortyp erzeugen.

Es gibt aber auch andere Übersetzer. Ein ganz einfacher Fall ist der Teil eines Programms, der Konfigurationsdateien einliest, den Inhalt analysiert und das weitere Verhalten des Programmes entsprechend parametrisiert.

Ein unter Umständen deutlich komplizierterer Fall liegt vor, wenn Texte aus z. B. einer ASCII-Notation in etwas visuell (hoffentlich) Ansprechenderes übersetzt werden sollen. Dabei kann es sich um HTML-Quellen handeln, die mittels CSS in eine Bildschirmdarstellung transformiert werden sollen, oder etwa um \TeX -Quellen, aus denen eine dvi-Datei erzeugt werden soll.

Wer mit Schlagworten wie XML, XSLT, usw. etwas anfangen kann, weiß, dass sich auch hier Übersetzungsprobleme in Hülle und Fülle finden.

Für dieses Kapitel gehen wir aber von der Annahme aus, dass es um die Übersetzung von Programmen geht, die in einer Programmiersprache wie C, Java, usw. verfasst sind, in der es die Konzepte von Variablen, Ausdrücken, Anweisungen, usw. gibt.

- 6.2 Ein Übersetzer besteht im allgemeinen aus einem *Analyseteil* und einem *Syntheseteil*.

Jeder dieser beiden Teile zerfällt bei einem modernen Übersetzer in mehrere sogenannte *Phasen*. Bei jeder Phase handelt es sich um eine Reihe logisch zusammenhängender Aufgaben, die von den anderen Aufgaben sinnvoll getrennt bearbeitet werden können.

Daneben werden bei einem Übersetzer manchmal eine Reihe von *Durchläufen* (engl. *passes*) unterschieden. Ein Durchlauf ist dabei dadurch gekennzeichnet, dass am Ende Ausgabedateien geschrieben werden, die im nächsten Durchlauf gelesen werden. Dieser Aspekt ist aber von nachgeordneter Bedeutung, weshalb wir ihn im folgenden ignorieren werden.

- 6.3 (**Phasen eines Übersetzers**) Wir beginnen mit einer bloßen Aufzählung der heute in vielen Übersetzern anzutreffenden Phasen:

- Vorverarbeitung (Makroersetzung)
- lexikalische Analyse
- syntaktische Analyse
- semantische Analyse
- Erzeugung des Zwischencodes

- Optimierung des Zwischencodes
- Erzeugung des Maschinencodes
- Optimierung des Maschinencodes

Mit Makroersetzung werden wir uns nicht beschäftigen.

Im Idealfall bilden lexikalische, syntaktische und semantische Analyse zusammen den Analyse- teil des Übersetzers. Allerdings kann man einen Übersetzer durchaus so schreiben, dass er schon gleichzeitig mit der Codeerzeugung zumindest auch beginnt.

Bei der sogenannten semantischen Analyse handelt es sich genauer gesagt um den Teil der Syntaxanalyse, der nicht mit Hilfe der benutzten, (üblicherweise) eingeschränkten kontextfreien Grammatik spezifiziert werden kann. Dazu gehören unter Umständen die Überprüfung, ob jede benutzte Variable vorher deklariert wurde oder ob Anzahl und Typen der aktuellen Parameter bei einem Prozeduraufruf mit Anzahl und Typen der formalen Parameter bei der Prozedurdeklaration übereinstimmen (vorausgesetzt, die Programmiersprache stellt solche Forderungen).

Die Erzeugung von Maschinencode erfolgt üblicherweise in zwei Schritten: erst in eine Zwischensprache für einen idealisierten Prozessor und von dort in die eigentliche Maschinensprache.

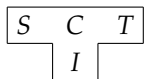
Neben den oben aufgezählten Punkten gibt es zwei weitere wichtige Aspekte bei Übersetzern, die nicht streng in die zeitliche Abfolge gepresst werden können, nämlich

- Verwaltung der Symboltabelle und
- Fehlerbehandlung.

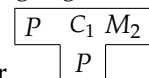
Die *Symboltabelle* ist eine Datenstruktur, in der für jeden Namen Informationen zusammengefasst sind. Dazu gehören etwa für eine Variable ihr Name und (soweit schon bekannt) ihr Typ, ihr Gültigkeitsbereich, der ihr zugeordnete Speicher, usw.. Im Falle von Prozedurnamen werden z. B. auch Anzahl und Typen ihrer Parameter in der Symboltabelle festgehalten.

Benutzerfreundliche *Fehlerbehandlung* ist schwierig. Sobald ein Übersetzer festgestellt hat, dass die Eingabe einen Syntaxfehler enthält, könnte er diese bloße Tatsache mitteilen und seine Arbeit beenden. Aber natürlich möchte man mehr: Erstens möchte man mitgeteilt bekommen, „wo der Fehler steckt“. Diese Formulierung setzt voraus, dass es immer eine solche Stelle gibt. Ist das überhaupt so? Und wenn ja, wie findet man sie? Zweitens möchte man mitgeteilt bekommen, um welchen Fehler es sich handelt. Geht das? Drittens möchte man, dass der Übersetzer nicht beim ersten „kleinen Tippfehler“ seine Arbeit beendet, sondern ihn womöglich behebt, oder zumindest an einer möglichst frühen Programmstelle hinter dem Fehler wieder aufsetzt, nach weiteren Fehlern sucht, und diese auch mitteilt. Wie findet man solche „günstigen“ Wiederaufsetzpunkte? Das alles sind wichtige Fragen, auf die in der Vorlesung „Programmiersprachen und Übersetzer“ eingegangen werden wird.

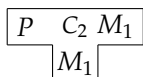
- 6.4 Falls ein Übersetzer C selbst in der Programmiersprache I implementiert ist und aus der Sprache S in die Sprache T übersetzt, stellt man ihn auch gelegentlich durch ein T-Diagramm der Form

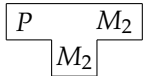



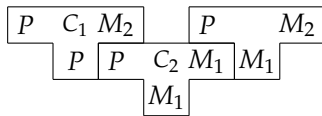
dar. Solche Diagramme können zum Beispiel benutzt werden, um die Erzeugung neuer



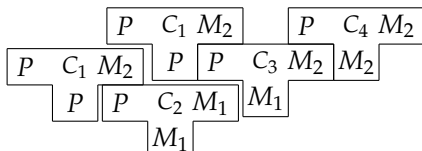
Übersetzer mit Hilfe alter zu beschreiben. Gegeben seien zum Beispiel zwei Übersetzer



und , und gesucht sei ein Übersetzer . Den kann man in zwei Schritten erzeugen:

1. Schritt: Durch

erzeugt man einen Übersetzer, der noch „in M_1 geschrieben“ ist.

2. Schritt: Den kann man benutzen, um durch

den gewünschten Übersetzer zu erzeugen.

Wir betrachten nun noch den Sonderfall $M_1 = M_2$. Dann leistet der zuletzt erzeugte Übersetzer C_4 im Prinzip das Gleiche wie der ursprünglich gegebene C_2 . Ist C_2 ein schlechter Übersetzer, aber C_1 das Programm für einen guten Übersetzer, dann ist C_4 ein guter, lauffähiger Übersetzer.

6.2 Lexikalische Analyse

Im Prinzip kann man zum Beispiel die Syntax von Variablenamen mit Hilfe einer kontextfreien Grammatik spezifizieren. Aber der Ableitungsbaum für einen Variablenamen ist für die Überprüfung, ob z. B. eine Zuweisung syntaktisch korrekt ist, völlig uninteressant (von Bedeutung ist nur, wo gegebenenfalls ein Variablenname steht und vielleicht noch, welcher). Außerdem ist die Menge der zulässigen Variablenamen üblicherweise vom Typ 3, d. h. der Ableitungsbaum wäre ohnehin degeneriert und hätte nur einen langen rechten Ast.

Ähnliches gilt für die Syntax von Zahldarstellungen (z. B. $12.3E4$), Operatoren (z. B. $<=$), usw..

6.5 Ein Programm(teil), das die lexikalische Analyse durchführt, heißt auch *Lexer* oder *Scanner*. Im Deutschen benutzen manche den Begriff *Symbolentschlüssler*.

6.6 Aufgabe der lexikalischen Analyse ist, den Eingabetext an den passenden Stellen in eine Folge sogenannter *Lexeme* aufzuteilen. Jedes Lexem passt zu einem *Muster* (engl. *pattern*). Es ist üblich, die Muster durch reguläre Ausdrücke zu beschreiben. Das ermöglicht eine kurze und präzise Beschreibung der Arbeitsweise des Lexers. Daraus kann eine effizient arbeitende Implementierung des Lexers erzeugt werden, und zwar sogar automatisch.

Ein Lexer stellt neben Prozeduren zur Initialisierung und Beendigung als Schnittstelle zur syntaktischen Analyse eine Prozedur bereit, die im folgenden stets *nextToken* heißen soll. Ein Aufruf führt dazu, dass der Lexer im Eingabestrom nach dem nächsten Lexem sucht, für das er ein sogenanntes *Token* zurückzuliefern hat. Ein Token besteht aus einem *Typ* und unter Umständen noch weiteren Informationen, den sogenannten *Attributen* des Tokens. Bei einem Token für einen Identifikator könnte das der eigentliche Name der Variable oder ein Verweis auf den Eintrag des Namens in der Symboltabelle sein, also z. B. (*id*,hugo).

Die Token, die vom Lexer geliefert werden können, bilden das Terminalsymbolalphabet für die Grammatik, die den „kontextfreien Anteil“ der Syntax der Programmiersprache beschreibt.

6.7 Jeden Kommentar kann man als Lexem auffassen, für das kein Token erzeugt wird. Ebenso sollte der Lexer alle Leerzeichen überlesen, jedenfalls insoweit sie für das Programm bedeutungslos sind. Es gibt allerdings Programmiersprachen (Fortran, Occam, Python, ...), bei denen die Position eines Lexems im Quellprogramm von Bedeutung sein kann. In einem solchen Fall ist der Lexer entsprechend aufwendiger zu implementieren.

6.8 **Beispiel.** Angenommen, der Lexer hätte als Eingabe die Zuweisung

```
erna = hugo + 1;
```

zu verarbeiten. Dann könnte er bei aufeinanderfolgenden Aufrufen von `nextToken` die folgenden Funktionswerte liefern:

```
(id,erna) (assignOp,) (id,hugo) (plusOp,) (num,1) (semi,)
```

Die syntaktische Analyse müsste dann überprüfen, ob die Zeichenfolge

```
id assignOp id plusOp num semi
```

in der Grammatik ableitbar ist.

6.9 Im allgemeinen wird ein Lexer die Quelldatei von Platte einlesen. Dies ist ein langsamer Vorgang, weshalb man darauf achten muss, es möglichst effizient zu implementieren. Es verbietet sich daher, für jedes einzelne Zeichen einen Prozeduraufruf an das Betriebssystem abzusetzen. Besser ist es, immer gleich einen ganzen sogenannten *Block* von Platte in einen Puffer zu schreiben und dann dort zu lesen.

Immer gleich die gesamte Datei vollständig in den Hauptspeicher zu laden wäre wohl noch effizienter, kann aber bei hinreichend vielen gleichzeitig offenen Dateien einen sehr großen Speicherbedarf nach sich ziehen.

Blockweises Lesen hat gegenüber zeichenweisem Lesen noch einen weiteren Vorteil. Bei manchen Programmiersprachen gibt es den Fall, dass der Lexer in der Eingabe mehrere Zeichen vorausschauen muss um zu entscheiden, wo das nächste Lexem endet. In diesem Fall ist eine Pufferung ohnehin nicht zu umgehen.

6.3 Lexer-Generatoren

6.10 Es gibt Programme zur automatischen Erzeugung von Lexern, sogenannte *Lexer-Generatoren*. Als Beispiele seien `lex`, `flex` und `JLex` genannt, aber auch andere Übersetzergenerator-Suiten wie *cocktail* und *Eli* enthalten solche Werkzeuge.

Lexergeneratoren erhalten als Eingabe eine Datei, in der insbesondere mit Hilfe regulärer Ausdrücke beschrieben ist, Lexeme welcher Struktur zu suchen sind, und welche Token daraus zu erzeugen sind. Als Ausgabe erzeugt ein Lexergenerator die Quelle eines Programmes, das als Lexer auf die gewünschte Art arbeitet.

Im folgenden werden einige in diesem Zusammenhang wichtige Punkte erläutert.

6.11 Der besseren Lesbarkeit wegen erlaubt man, die Beschreibung eines regulären Ausdrucks zu strukturieren, indem man Teilausdrücke mit Namen bezeichnen kann. Wir wollen das als *reguläre Definitionen* bezeichnen. Statt einer formalen Definition möge ein Beispiel genügen:

```

digit → [0-9]
letter → [a-zA-Z]
id → letter (letter|digit)*
num → digit+ (\.digit+) ? (∅*[-+]?digit+) ?

```

Der entscheidende Punkt ist, dass man auf der rechten Seite einer regulären Definition nur Namen regulärer Ausdrücke benutzen darf, die *vorher* definiert wurden. Direkte oder indirekte Rekursionen sind verboten. Folglich kann man durch banales Einsetzen der Namen durch ihre Bedeutung alle rechten Seiten in „normale“ Regexp's umwandeln.

Der Backslash \ vor dem Punkt ist notwendig, weil ein Punkt als „echtes“ Zeichen gematcht werden soll (und nicht das Metazeichen gemeint ist).

- 6.12** Die Eingabedatei für einen Lexergenerator enthält zum einen reguläre Definitionen, und zum anderen *Regeln*, die einigen regulären Ausdrücken *Aktionen* zuordnen. Eine Aktion gibt an, was der zu erzeugende Lexer tun soll, wenn `nextToken` aufgerufen wird und das zum nächsten Token gehörende Lexem zum jeweiligen regulären Ausdruck gehört.

Zum Beispiel könnte für den regulären Ausdruck **id** als Aktion spezifiziert sein, dass

- der gefundene Name in der Symboltabelle zu suchen und gegebenenfalls einzutragen ist und
- der Rückgabewert das Token **id** und als Attribut der Verweis auf den Symboltabelleneintrag zu liefern ist.

Außerdem können üblicherweise noch Programmstücke spezifiziert werden, die in den erzeugten Lexer mit eingebaut werden.

- 6.13** In diesem Zusammenhang sind wieder zwei Punkte wichtig, von denen der eine schon früher erwähnt wurde.

1. Wenn der Lexer versucht, zu einem regulären Ausdruck eine passende Folge von Eingabezeichen zu finden, dann eine *möglichst lange*.

Wenn zum Beispiel in der Eingabe 123.45E67 steht, dann wird der Lexer *ein* Token **num** mit Attribut 123.45E67 zurückliefern, und nicht etwa nacheinander zwei **num**-Token mit Attributen 123.4 und 5E67.

2. Es kann passieren, dass das gleiche maximal lange Präfix der Eingabe auf die regulären Ausdrücke mehrerer Lexerregeln passt. Jedenfalls bei vielen Lexergeneratoren wird in diesem Fall ein Lexer erzeugt, der immer die *erste passende* Regel benutzt.

Wenn zum Beispiel eine reguläre Definition **keyword** → `for|do|if|then` usw. existiert und eine Aktion für **keyword** *vor* der Aktion für **id** (wie in Punkt 6.11 angegeben) spezifiziert wird, dann wird der Lexer für die Zeichenfolge `if` in der Eingabe ein Token **keyword** zurückliefern und nicht ein Token **id**.

- 6.14** Die Programme `lex` und das GNU-Pendant `flex` sind (in C programmiert und) dafür gedacht, in C geschriebene Lexer zu erzeugen. Dementsprechend sind die zu den Regeln gehörigen Aktionen ebenfalls in C anzugeben. Das Programm `JLex` ist (in Java programmiert und) dafür gedacht, in Java geschriebene Lexer zu erzeugen. Dementsprechend sind die zu den Regeln gehörigen Aktionen ebenfalls in Java anzugeben.

Die Syntax für die Eingabedateien für `flex` und `JLex` sind ähnlich, aber im Detail verschieden. Nachfolgend beschreiben wir anhand eines einfachen Beispiels die Prinzipien für `JLex`.

Die Aufgabe besteht darin, in einem Strom von Eingabezeichen alle Lexeme zu finden, die „Variablenamen“ oder Darstellungen von ganzen und gebrochenen Zahlen sind. Sogenannter „white-space“ soll ignoriert werden. Alle anderen Zeichen sollen als „unpassend“ gekennzeichnet werden.

6.15 (Lexerspezifikation für JLex) Eine Eingabedatei für JLex hat die Grobstruktur

```

<Benutzer-Code>
%%
<reguläre Definitionen>
%%
<Regeln mit Aktionen>

```

Für das konkrete Beispiel könnte man zum Beispiel wie folgt vorgehen:

- *<Benutzer-Code>*:

```

import java.lang.System;

class Simple {
    public static void main(String argv[]) throws java.io.IOException {
        Yylex yy = new Yylex(System.in);
        Ytoken t;
        while ((t = yy.yylex()) != null)
            System.out.println(t);
    }
}

class Ytoken {
    Ytoken(int index, String text) {
        m_index = index;
        m_text = new String(text);
    }

    private static final String tokName[] = { "UNKNOWN", "FLOAT", "INT", "ID" };
    public int m_index;
    public String m_text;
    public String toString() {return tokName[m_index]+": "+m_text+"";}
}

```

- %%

- *<reguläre Definitionen>*:

```

ALPHA = [A-Za-z]
DIGIT = [0-9]

ID = {ALPHA}({ALPHA}|{DIGIT}|_)*
INT = {DIGIT}+
FLOAT = {INT}\.{INT}

WHITE_SPACE_CHAR = [\n\ \t\b\012]

```

- %%
- $\langle \text{Regeln mit Aktionen} \rangle$:

```
{WHITE_SPACE_CHAR}+ { }
```

```
{FLOAT} {
    return (new Ytoken(1,yytext()));
}
```

```
{INT} {
    return (new Ytoken(2,yytext()));
}
```

```
{ID} {
    return (new Ytoken(3,yytext()));
}
```

```
. {
    return (new Ytoken(0,yytext()));
}
```

6.16 Befände sich obige JLex-Spezifikation in einer Datei namens `simple.lex` und wäre JLex irgendwo im CLASSPATH verfügbar, dann würde das Kommando

```
java JLex.Main simple.lex
```

in der Datei `simple.lex.java` die Java-Quelle eines Lexer-Programms erzeugen. Die Fortschritte von JLex während der Arbeit werden so protokolliert:

```
Processing first section -- user code.
Processing second section -- JLex declarations.
Processing third section -- lexical rules.
Creating NFA machine representation.
NFA comprised of 40 states.
Working on character classes.:.:.:.:.::
NFA has 9 distinct character classes.
Creating DFA transition table.
Working on DFA states.....
Minimizing DFA transition table.
8 states after removal of redundant states.
Outputting lexical analyzer code.
```

Wie man sieht, erzeugt JLex erst einen nichtdeterministischen Automaten, der dann in einen deterministischen umgewandelt wird. Wie man das im Prinzip macht, haben wir in Kapitel 2 gesehen.

Anschließend kann man mit

```
javac simple.lex.java
```

den lauffähigen Lexer erzeugen, den man mit

```
java Simple
```

laufen lassen kann. Probieren Sie es aus.

- 6.17 Im Prinzip benutzen Lexer zum Auffinden des nächsten Lexems endliche Automaten wie sie in Kapitel 2 beschrieben wurden.

Dabei übersetzt der Lexergenerator die regulären Ausdrücke zunächst in (einen oder mehrere) nichtdeterministische EAs. Da der erzeugte Lexer aber deterministisch ist, muss auf die ein oder andere Weise die Potenzmengenkonstruktion zum Deterministischemachen angewendet werden. Eine Möglichkeit besteht darin, dass der Generator dies tut und im Lexer einen DEA implementiert. Theoretisch kann sich dabei die Zustandsmenge exponentiell aufblähen. In der Praxis zeigt sich aber, dass das bei geschickter Konstruktion unter Beschränkung auf die tatsächlich erreichbaren Zustände oft nicht passiert.

Die andere Möglichkeit besteht darin, den Lexer bei der Arbeit immer nur die eine Teilmenge der aktuell für den NEA möglichen Zustände mitzuführen. Beide Vorgehensweisen werden in der Praxis eingesetzt.

- 6.18 Man beachte, dass das Suchen nach einem maximal langen Eingabeprefix, das zu einer Regel passt, gegenüber der „einfachen“ Simulation zusätzlichen Aufwand erfordert.

Erstens wird man in der Regel in der Eingabe über das letzten Endes zu wählende Prefix hinaus schauen müssen. Gibt es zum Beispiel zwei Regeln

```
lessEqOp → <=
lessOp  → <
```

dann muss der Lexer, wenn er ein < gefunden hat, nachsehen, welches Zeichen darauf folgt. Im Falle eines = ist der Tokentyp **lessEqOp**, ist es z. B. ein a, so ist der Tokentyp **lessOp** und das a „muss zurück in die noch nicht verarbeitete Eingabe“. Wie schlimm das werden kann, sieht man z. B. in Fortran. Dort ist zum Beispiel

- DO 100 I=1,10 die erste Zeile einer Schleife, aber
- DO 100 I=1.10 die Zuweisung des Wertes 1.10 an die Variable D0100I.

Tatsächlich nutzen Fortran-Übersetzer wegen dieser Probleme andere Vorgehensweisen.

Zweitens muss man, solange die regulären Ausdrücke mehrerer Regeln oder mehrere Alternativen eines regulären Ausdrucks der Form $R_1 | R_2$ o.ä. passen, sozusagen mehrere endliche Automaten gleichzeitig simulieren und nicht nur einen.

- 6.19 Viele (aber nicht alle) Lexergeneratoren erzeugen *tabellengesteuerte* Lexer. Sie bestehen aus dem im wesentlichen immer gleichen Code, der die endlichen Automaten simuliert, die in Form einer Tabelle gespeichert sind. Man sehe sich z. B. `simple.lex.java` an, um einmal zu sehen, wie JLex es macht.

Bei der naiven Vorgehensweise, für jede Kombination von Zustand (das können viele sein) und Zeichen (das sind viele) getrennt die Arbeitsweise zu spezifizieren können aber schnell unnötig große Tabellen entstehen. Zum Glück gibt es gute Heuristiken, um solche Tabellen zu komprimieren ohne bei der Simulation der EA Einbußen bei der Rechenzeit in Kauf nehmen zu müssen.

- 6.20 Nehmen wir einmal als gegeben hin, dass erstens während der lexikalischen Analyse alle Kommentare aus dem Programm entfernt werden, und dass zweitens die lexikalische Analyse durch einen endlichen Automaten erledigt werden soll. Dann wird klar, warum es in etlichen Programmiersprachen, bei denen Kommentare durch ausgezeichnete Zeichenketten am Anfang und am Ende gekennzeichnet sind (z. B. /* und */ in C), verboten ist, Kommentare ineinander zu schachteln.

Man müsste die Eingabe auf korrekte Klammerstruktur überprüfen, und das kann ein endlicher Automat nicht. Andererseits würde es genügen, einen einfachen Zähler für die Schachtelungstiefe hinzuzunehmen, um des Problems Herr zu werden.

6.4 Syntaktische Analyse

6.21 Wir hatten im vorangegangenen Kapitel gesehen, wie man mit Hilfe nichtdeterministischer Kellerautomaten Syntaxanalyse für kontextfreie Grammatiken durchführen kann. In der Praxis muss man die Aufgabe aber mit deterministischen Algorithmen lösen. Dafür gibt es zwei Möglichkeiten:

- Man benutzt zur Syntaxanalyse nicht Kellerautomaten, sondern „andere Algorithmen“. In den Abschnitten 6.5 und 6.6 werden wir klassische Beispiele hierfür kennenlernen. Außerdem werden die Grundlagen für die andere Möglichkeit gelegt.
- Man schränkt sich auf eine echte Teilklasse aller kontextfreien Grammatiken ein, damit man mit deterministischen Kellerautomaten oder wenigstens einer „einfachen“ Erweiterung davon auskommt. Dies wird Gegenstand der Abschnitte 6.7 und folgender sein.

Es gibt zwei große Klassen kontextfreier Grammatiken, für die Parser-Generatoren zu Verfügung stehen. Das eine sind die sogenannten *LL(k)-Grammatiken*, das andere die *LR(k)-Grammatiken*. In beiden Fällen steht das k für die Anzahl Zeichen, die in der Eingabe vorausgeschaut wird, um zu entscheiden, wie beim Parsen weiter verfahren werden soll. Für *LL(k)-Grammatiken* gibt es ein deterministisches Top-down-Verfahren zur Syntaxanalyse, für *LR(k)-Grammatiken* ein deterministisches Bottom-up-Verfahren. Wir werden uns in dieser Vorlesung nur mit letzteren beschäftigen.

Bevor wir uns (zumindest ansatzweise) mit diesen Verfahren und Beispielen von Parser-Generatoren dafür beschäftigen, gehen wir zuvor in den folgenden Abschnitten auf zwei allgemeinere deterministische Verfahren zur kontextfreien Syntaxanalyse ein. Sie sind nicht so sehr im Hinblick auf Programmiersprachen von Bedeutung, sondern mehr beim Parsen natürlicher Sprache. Ihr Verständnis erleichtert aber den Zugang zu den *LR*-Verfahren.

6.5 Der Algorithmus von Cocke, Younger und Kasami

Die in diesem und dem nachfolgenden Abschnitt beschriebenen Algorithmen werden in Compilern für Programmiersprachen nicht verwendet. Sie sind aber erstens zur Hinführung auf dort angewendete Methoden nützlich, und zweitens in der Linguistik bei der Verarbeitung natürlicher Sprache (automatische Übersetzer etc.) von Bedeutung.

6.5.1 Chomsky-Normalform für kontextfreie Grammatiken

6.22 **Definition.** Eine kontextfreie Grammatik $G(N, T, S, P)$ ist in *Chomsky-Normalform*, wenn sie den folgenden Einschränkungen genügt:

- Jede Produktion $X \rightarrow w$ mit $X \neq S$ hat als rechte Seite entweder ein Wort $w \in N^2$ (genau zwei Nichtterminalsymbole) oder ein Wort $w \in T$ (genau ein Terminalsymbol).
- Wenn es die Produktion $S \rightarrow \varepsilon$ gibt, dann kommt S bei keiner Produktion auf der rechten Seite vor.

6.23 Satz. Zu jeder kontextfreien Grammatik G gibt es eine kontextfreie Grammatik G' , die zu G äquivalent ist (also $L(G) = L(G')$) und in Chomsky-Normalform ist.

Als Vorbereitung zeigen wir zunächst die folgenden Aussage, die auch in späteren Abschnitten noch Verwendung finden wird.

6.24 Lemma. Für jede kontextfreie Grammatik G kann man die Menge $EPS(G) = \{X \in N \mid X \Rightarrow^* \varepsilon\}$ aller Nichtterminalsymbole berechnen, aus denen das leere Wort ableitbar ist.

6.25 Beweisskizze. Wir behaupten dass der folgende einfache Algorithmus das Gewünschte leistet:

```

i ← 0
M0 ← {X | X → ε ∈ P}
do
  i ← i + 1
  Mi ← Mi-1 ∪ {X | ∃w ∈ Mi-1* : X → w ∈ P}
until Mi = Mi-1
return Mi

```

Man beginnt mit den Nichtterminalsymbolen, aus denen offensichtlich das leere Wort ableitbar ist, und erweitert diese Menge gegebenenfalls sukzessive um weitere Nichtterminalsymbole, aus denen indirekt ε ableitbar ist.

Den Nachweis, dass dieser Algorithmus korrekt ist, überlassen wir als Übung. ■

Offensichtlich kann man mit Hilfe der Berechnung von $EPS(G)$ auch entscheiden, ob $\varepsilon \in L(G)$ ist oder nicht. Man muss nur prüfen, ob $S \in EPS(G)$ ist.

6.26 Beispiel. Es sei die Grammatik $G = (\{X, Y, Z\}, \{a\}, X, P)$ gegeben mit der Produktionsmenge $P = \{X \rightarrow YZ, Y \rightarrow ZZ, Z \rightarrow a|\varepsilon\}$.

Der obige Algorithmus berechnet nacheinander:

$$\begin{aligned}
 M_0 &= \{Z\} \\
 M_1 &= \{Z, Y\} \text{ wegen } Y \rightarrow ZZ \in P \\
 M_2 &= \{Z, Y, X\} \text{ wegen } X \rightarrow YZ \in P \\
 M_3 &= \{Z, Y, X\} \text{ (keine Änderung mehr)}
 \end{aligned}$$

Also ist $EPS(G) = \{X, Y, Z\}$.

6.27 Lemma. Zu jeder kontextfreien Grammatik $G = (N, T, S, P)$ kann man eine kontextfreie Grammatik $G' = (N, T, S, P')$ berechnen, für die gilt:

- $L(G') = L(G) \setminus \{\varepsilon\}$ und
- keine Produktion von G' hat ε als rechte Seite.

6.28 Beweisskizze. In P' nimmt man alle Produktionen $X \rightarrow w'$ auf, für die gilt:

- $w' \neq \varepsilon$ und
 - es gibt eine Produktion $X \rightarrow w \in P$, so dass w' aus w entsteht, indem man einige Vorkommen von Symbolen aus $EPS(G)$ entfernt.
-

6.29 Lemma. Zu jeder kontextfreien Grammatik $G = (N, T, S, P)$ kann man eine äquivalente kontextfreie Grammatik $\bar{G} = (N, T, \bar{S}, \bar{P})$ berechnen, für die gilt:

- Wenn $\varepsilon \notin L(G)$ ist, dann hat \bar{G} keine Produktion ε als rechte Seite.
- Wenn $\varepsilon \in L(G)$ ist, dann ist $\bar{S} \rightarrow \varepsilon$ die einzige Produktion von \bar{G} mit ε als rechter Seite und \bar{S} kommt bei keiner Produktion auf der rechten Seite vor.

6.30 Beweisskizze. Man konstruiert zunächst wie in Lemma 6.27 beschrieben eine Grammatik G' mit $L(G') = L(G) \setminus \{\varepsilon\}$. Wenn $\varepsilon \notin L(G)$ ist, kann man $\bar{G} = G'$ setzen und man ist wegen $L(\bar{G}) = L(G') = L(G)$ schon fertig.

Ist $\varepsilon \in L(G)$, dann erweitert man G' um ein neues Nichtterminalsymbol \bar{S} , das Startsymbol von \bar{G} wird und setzt $\bar{P} = P' \cup \{\bar{S} \rightarrow \varepsilon, \bar{S} \rightarrow S'\}$. ■

Nun können wir beweisen:

6.31 Beweis. (von Satz 6.23) Es sei $G_0 = (N_0, T, S, P_0)$ eine beliebige kontextfreie Grammatik. Wir konstruieren eine äquivalente kontextfreie Grammatik in Chomsky-Normalform in mehreren Schritten:

1. Konstruiere aus G eine äquivalente Grammatik G_1 , bei der Terminalsymbole $a \in T$ nur in Produktionen der Form $X \rightarrow a$ vorkommen.
2. Konstruiere aus G_1 eine äquivalente Grammatik G_2 wie in Lemma 6.29 beschrieben.
3. Konstruiere aus G_2 eine äquivalente Grammatik G_3 , die keine Produktionen der Form $X \rightarrow Y$ enthält.
4. Konstruiere aus G_3 eine äquivalente Grammatik G_4 , die keine Produktionen der Form $X \rightarrow Y_1 Y_2 \cdots Y_k$ mit $k \geq 3$ enthält.

Grammatik G_4 hat dann die gewünschte Eigenschaft.

1. Zunächst führt man für jedes Terminalsymbol $a \in T$ ein neues Nichtterminalsymbol Z_a ein: $N_1 = N_0 \cup \{Z_a \mid a \in T\}$. Die Produktionsmenge P_1 enthält zum einen alle Produktionen der Form $Z_a \rightarrow a$ und zum zweiten alle Produktionen, die man erhält, indem man in jeder Produktion aus P_0 jedes Vorkommen eines Terminalsymboles a durch Z_a ersetzt.
2. Nach dem zweiten Schritt sind alle Produktionen von der Form $x \rightarrow a$ oder von der Form $X \rightarrow Y_1 \cdots Y_k$ mit $k \geq 1$ (mit der eventuellen Ausnahme $S \rightarrow \varepsilon$).
3. Für jedes $X \in N_2$ berechnet man die Menge $M(X) = \{X' \mid X \Rightarrow^* X'\}$ aller daraus ableitbaren Nichtterminalsymbole.

Dies erledigt der folgende Algorithmus:

```

i ← 0
M0(X) ← {X}
do
  i ← i + 1
  Mi(X) ← Mi-1(X) ∪ {X' | ∃Y ∈ Mi-1(X) : Y → X' ∈ P}
until Mi(X) = Mi-1(X)
return Mi(X)

```

P_3 erhält man aus P_2 , indem man zunächst alle Produktionen der Form $X \rightarrow X'$ streicht. Für jede verbliebene Produktion $X' \rightarrow Y_1 \cdots Y_k$ fügt man alle Produktionen $X \rightarrow Y'_1 \cdots Y'_k$ hinzu, sofern $X' \in M(X)$ und für alle i , $1 \leq i \leq k$, auch $Y'_i \in M(Y_i)$ ist.

4. P_4 erhält man aus P_3 , indem man jede Produktion der Form $X \rightarrow Y_1 \cdots Y_k$ mit $k \geq 3$ entfernt und ersetzt durch die $k - 1$ Produktionen $X \rightarrow Y_1 V_1$, $V_1 \rightarrow Y_2 V_2$, \dots , $V_{k-1} \rightarrow Y_{k-1} Y_k$. Dabei seien die V_i jeweils neue Nichtterminalsymbole (die natürlich alle zu N_4 hinzugenommen werden).

■

6.32 Beispiel. Wir betrachten die Grammatik mit Startsymbol E und den folgenden Produktionen:

$$\begin{aligned} E &\rightarrow E + F \mid F \\ F &\rightarrow (E) \mid a \end{aligned}$$

1. Im ersten Schritt der obigen Konstruktion werden daraus die Produktionen

$$\begin{aligned} E &\rightarrow E Z_+ F \mid F & Z_+ &\rightarrow + & Z_a &\rightarrow a \\ F &\rightarrow Z_-(E Z_-) \mid Z_a & Z_- &\rightarrow (& Z_& \rightarrow) \end{aligned}$$

Für das Terminalsymbol a ist das Vorgehen natürlich überflüssig. Aber da der Algorithmus es vorschreibt, haben wir es mit gemacht.

2. Da keine ε -Produktionen vorhanden sind, ändert sich im zweiten Schritt nichts.
3. Im dritten Schritt werden zunächst die Mengen $M_i(X)$ berechnet:

| X | $M_0(X)$ | $M_1(X)$ | $M_2(X)$ | $M_3(X)$ |
|-----|----------|--------------|-----------------|-----------------|
| E | $\{E\}$ | $\{E, F\}$ | $\{E, F, Z_a\}$ | $\{E, F, Z_a\}$ |
| F | $\{F\}$ | $\{F, Z_a\}$ | $\{F, Z_a\}$ | $\{F, Z_a\}$ |

Die Produktionen $E \rightarrow F$ und $F \rightarrow Z_a$ werden entfernt.

Zum „Ausgleich“ werden neue Produktionen aufgenommen. Aus $E \rightarrow E Z_+ F$ entstehen:

$$E \rightarrow E Z_+ F \mid F Z_+ F \mid Z_a Z_+ F \mid E Z_+ Z_a \mid F Z_+ Z_a \mid Z_a Z_+ Z_a$$

Aus $F \rightarrow Z_-(E Z_-)$ ergeben sich

$$\begin{aligned} F &\rightarrow Z_-(E Z_-) \mid Z_-(F Z_-) \mid Z_-(Z_a Z_-) \\ E &\rightarrow Z_-(E Z_-) \mid Z_-(F Z_-) \mid Z_-(Z_a Z_-) \end{aligned}$$

4. Im vierten Schritt wird jede Produktion mit drei Nichtterminalzeichen auf der rechten Seite ersetzt durch zwei Produktionen mit nur zwei Nichtterminalzeichen auf der rechten Seite. Zum Beispiel wird $E \rightarrow E Z_+ F$ ersetzt durch Produktionen $E \rightarrow E V_1$ und $V_1 \rightarrow Z_+ F$. Analog geht man in allen anderen Fällen vor, wobei man analog zu V_1 neue zusätzliche Nichtterminalsymbole V_2, \dots, V_{24} einführen muss.

Jedenfalls verlangt das der Algorithmus so. Scharfes Hinsehen ergibt im vorliegenden Beispiel aber, dass man etwas sparsamer sein kann und mit den folgenden Produktionen auskommt:

$$\begin{array}{ll}
 E \rightarrow E V_1 \mid F V_1 \mid Z_a V_1 & V_1 \rightarrow Z_+ F \\
 E \rightarrow E V_2 \mid F V_2 \mid Z_a V_2 & V_2 \rightarrow Z_+ Z_a \\
 E \rightarrow Z_{\zeta} V_3 \mid Z_{\zeta} V_4 \mid Z_{\zeta} V_5 & V_3 \rightarrow E Z_{\zeta} \\
 F \rightarrow Z_{\zeta} V_3 \mid Z_{\zeta} V_4 \mid Z_{\zeta} V_5 & V_4 \rightarrow F Z_{\zeta} \\
 & V_5 \rightarrow Z_a Z_{\zeta}
 \end{array}$$

6.5.2 Der Algorithmus von Cocke, Younger und Kasami

In diesem und im folgenden Abschnitt (Algorithmus von Earley) beschränken wir uns darauf, die Zugehörigkeit des Eingabewortes zur erzeugten Sprache zu entscheiden. Die Verallgemeinerung der Algorithmen, um gegebenenfalls auch Ableitungsbäume zu erhalten, lassen wir zu Gunsten besserer Verstehbarkeit der Algorithmen weg.

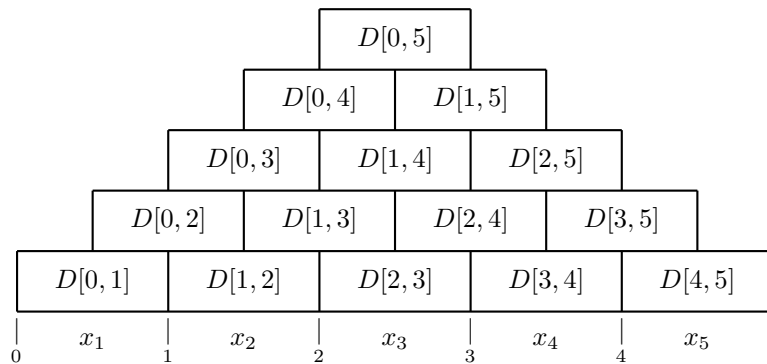
- 6.33** Der *Algorithmus von Cocke, Younger und Kasami* (kurz CYK) kann für jede Grammatik $G = (N, T, S, P)$ in Chomsky-Normalform und jedes Wort $w \in T^*$ auf deterministische Weise feststellen, ob $w \in L(G)$ ist oder nicht.

Es ist im folgenden nützlich, sich eine Eingabe $w = x_1 \cdots x_n \in T^n$ so vorzustellen, dass sie mit $n + 1$ „Trennstellen“ versehen ist, die neben den einzelnen Symbolen liegen. Wir bezeichnen die Trennstellen mit $|_0, |_1, \dots, |_n$, z. B.:

$$\begin{array}{cccccc}
 | & a & | & b & | & b & | & a & | & b & | \\
 0 & 1 & 2 & 3 & 4 & 5
 \end{array}$$

Für $0 \leq i < j \leq n$ bezeichnen wir mit $w(i, j]$ dasjenige Teilwort von $w = x_1 \cdots x_n$, das zwischen den Trennstellen $|_i$ und $|_j$ liegt, also $w(i, j] = x_{i+1} \cdots x_j$. Zum Beispiel ist also $w(0, n] = w$. Für den Fall, dass $i = j$ ist, sei $w(i, j] = \varepsilon$ vereinbart. Es ist z. B. immer $|w(i, j]| = j - i$ und für $i \leq m \leq j$ gilt: $w(i, m]w(m, j] = w(i, j]$.

- 6.34 Algorithmus. (Cocke, Younger, Kasami)** CYK berechnet eine Datenstruktur, die man sich wie in Abbildung 6.1 dargestellt als Pyramide vorstellen kann. Die einzelnen Komponenten $D[i, j]$ für $0 \leq i < j \leq n$ sind (möglicherweise leere) Teilmengen von Nichtterminalsymbolen. Das Ziel ist es, zu erreichen, dass $D[i, j] = \{X \mid X \Rightarrow^* w(i, j]\}$ ist. Hat man das erreicht, dann weiß man insbesondere, ob das Eingabewort in der Grammatik ableitbar ist: Es ist genau dann $x_1 \cdots x_n \in L(G)$, wenn $S \in D[0, n]$ ist.

Abbildung 6.1: Die von CYK aufgebaute Datenstruktur für den Fall $n = 5$

```

for  $i \leftarrow 0$  to  $n - 1$  do
     $D[i, i + 1] = \{X \mid X \rightarrow x_{i+1} \in P\}$ 
od

for  $k \leftarrow 2$  to  $n$  do
    for  $i \leftarrow 0$  to  $n - k$  do
         $D[i, i + k] \leftarrow \emptyset$ 
        for  $m \leftarrow i + 1$  to  $i + k - 1$  do
            if exists  $X \rightarrow YZ \in P$  such that  $(Y \in D[i, m] \wedge Z \in D[m, i + k])$  then
                 $D[i, i + k] \leftarrow D[i, i + k] \cup \{X\}$ 
            fi
        od
    od
od

```

Wie man sieht, enthält der Algorithmus 3 ineinander geschachtelte Schleifen, von denen keine mehr als n Werte durchläuft; andererseits wird für die ersten $n/4$ Durchläufe der äußersten und die letzten $n/4$ Durchläufe der mittleren Schleife die innerste jeweils mindestens $n/4$ mal durchlaufen. Der Test für die **if**-Anweisung benötigt konstante Zeit. Also ist die Laufzeit des Algorithmus in $\Theta(n^3)$.

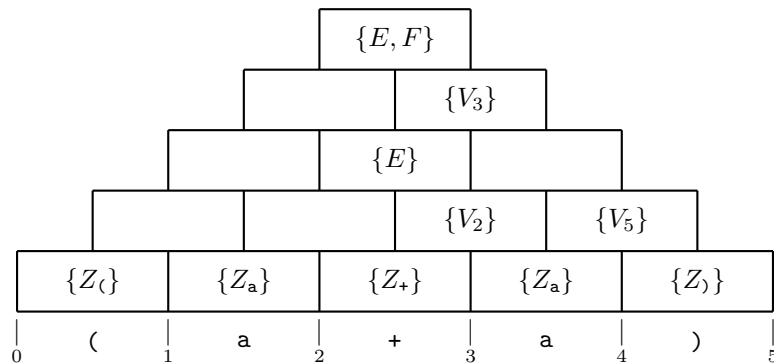
Es ist auch nicht schwer, einzusehen, dass am Ende in der Tat $D[i, j] = \{X \mid X \Rightarrow^* x_{i+1} \cdots x_j\}$ ist. Also ist $x_1 \cdots x_n$ genau dann aus $L(G)$, wenn $S \in D[0, n]$ ist.

6.35 Beispiel. Wir wenden CYK auf die Grammatik an, für die wir in Beispiel 6.32 eine Chomsky-Normalform bestimmt hatten. Als Eingabewort betrachten wir $(a+a)$.

Die Eintragungen in der untersten Schicht sind klar. In allen darüberliegenden Schichten bedeuten Kästchen ohne Angabe einer Menge von Nichtterminalzeichen, dass die entsprechenden Menge $D[i, j]$ leer ist.

Der Eintrag $D[1, 5] = \{V_3\}$ ergibt sich zum Beispiel auf Grund der Produktion $V_3 \rightarrow EZ$, beim Durchlauf der innersten Schleife für den Wert $m = 4$ aus dem Eintrag $D[1, 4] = \{E\}$ unmittelbar links unterhalb und dem Eintrag $D[4, 5] = \{Z\}$ ganz unten rechts. Für die anderen Werte von m ergibt sich kein Beitrag zu $D[1, 5]$. Analoges gilt für die anderen Einträge.

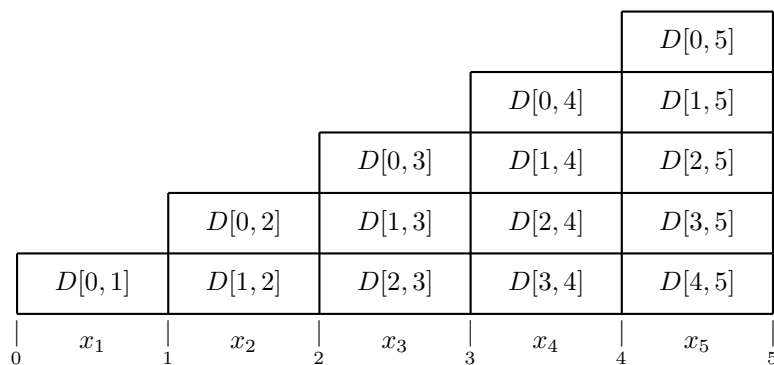
Dass in diesem Beispiel jede Menge $D[i, j]$ leer oder nur einelementig ist, ist Zufall. Bei anderen Grammatiken können die Mengen durchaus mehrere Nichtterminalzeichen enthalten.

Abbildung 6.2: CYK für die Beispielgrammatik und Eingabe $(a+a)$

Der Rest dieses Abschnittes und der nachfolgende Abschnitt (Algorithmus von Earley) können unter Umständen aus Zeitgründen nicht behandelt werden. Ob sie prüfungsrelevant oder sind, wird in der Vorlesung rechtzeitig bekannt gegeben.

Im folgenden transformieren wir nun CYK, um uns nach und nach dem Algorithmus von Earley zu nähern.

6.36 Algorithmus. Zunächst ordnen wir die Kästchen wie in Abbildung 6.3 dargestellt einfach etwas anders an.

Abbildung 6.3: Alternative Darstellung der von CYK aufgebauten Datenstruktur für den Fall $n = 5$

Die nächste Beobachtung ist, dass man die $D[i, j]$ nicht nur „Schicht für Schicht“ von unten nach oben (und innerhalb jeder Schicht von links nach rechts) berechnen kann, sondern auch „Spalte für Spalte“ von links nach rechts (und innerhalb jeder Spalte von unten nach oben). Damit ergibt sich die folgende Variante des Algorithmus:

```

for  $j \leftarrow 1$  to  $n$  do
   $D[j-1, j] \leftarrow \{X \mid X \rightarrow x_j \in P\}$ 
  for  $i \leftarrow j-1$  downto  $0$  do
     $D[i, j] \leftarrow \emptyset$ 

```

```

    for  $m \leftarrow i + 1$  to  $j$  do
        if exists  $X \rightarrow YZ \in P$  such that  $(Y \in D[i, m] \wedge Z \in D[m, j])$  then
             $D[i, j] \leftarrow D[i, j] \cup \{X\}$ 
        fi
    od
od

```

6.37 Wir fassen nun alle für jedes j mit $1 \leq j \leq n$ alle Mengen $D[i, j]$ ($0 \leq i \leq j - 1$) zu einer Menge S_j zusammen und fügen schon mal eine vorläufig noch leere Menge S_0 hinzu. Dies ist in Abbildung 6.4 dargestellt.

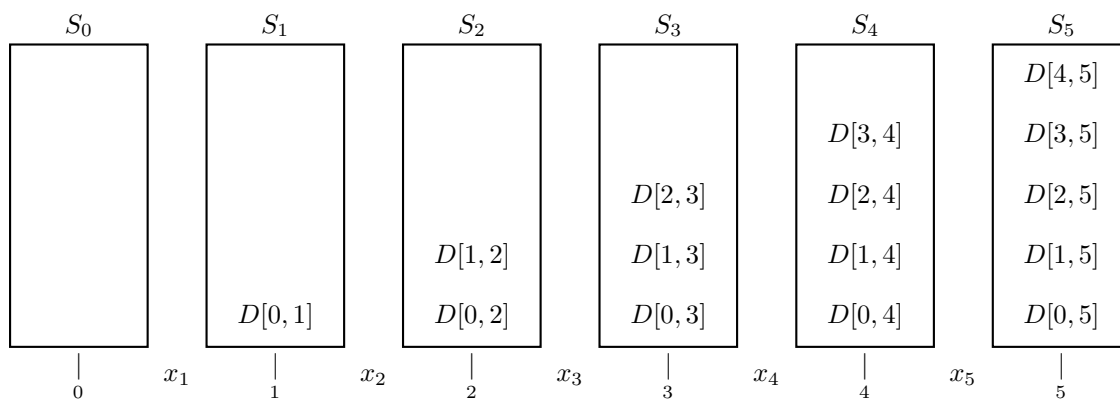


Abbildung 6.4: Alternative Darstellung der von CYK aufgebauten Datenstruktur für den Fall $n = 5$

6.38 **Beispiel.** Wir passen nun das Bild aus Beispiel 6.35 an die neue Darstellungsweise an. Dabei tun wir zusätzlich noch zweierlei: Zum einen geben wir zu jedem Nichtterminalsymbol auch noch an, auf Grund welcher Produktion es in eine Menge $D[i, j]$ aufgenommen wurde. Zum anderen geben wir auf der rechten Seite der Produktion noch die Trennstellen i und j mit an. Es ergibt sich Abbildung 6.5.

Nun ist es natürlich so, dass die Nummer der rechten Trennstelle immer mit dem Index der Menge S_j übereinstimmt, in der die Produktion notiert ist. Man kann also die Zahl weglassen. Außerdem ist es üblich, diese Stelle mit einem dicken Punkt • zu markieren. Es ergibt sich Abbildung 6.6.

Damit sind wir nun so weit, dass wir übergehen können zu ...

6.6 Der Algorithmus von Earley

Der Algorithmus von Earley kann als Verallgemeinerung des Algorithmus von Cocke, Younger und Kasami angesehen werden, bei der die Grammatik nicht mehr in Chomsky-Normalform vorliegen muss.

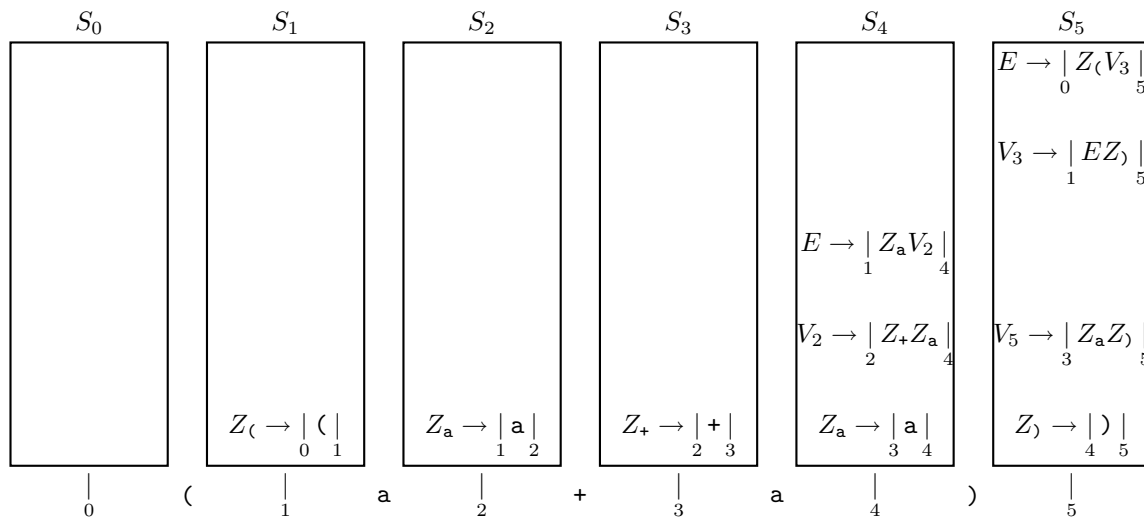


Abbildung 6.5: Alternative Darstellung des Beispiels für CYK

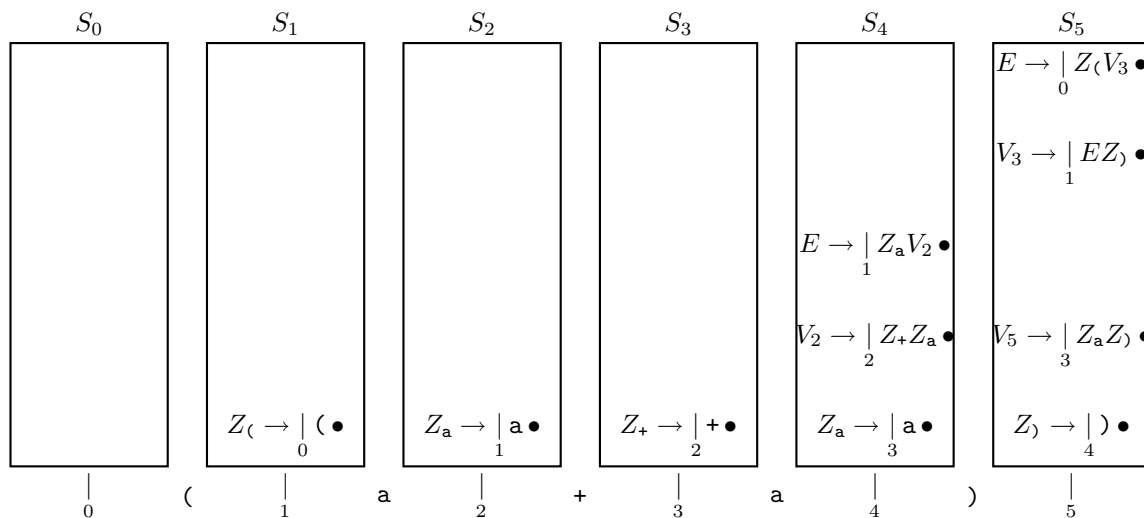


Abbildung 6.6: Alternative Darstellung des Beispiels für CYK

6.39 Wir gehen aus von der Darstellung des Algorithmus von CYK wie in Abbildung 6.6 beispielhaft angegeben: Für eine Eingabe der Länge n gibt es Mengen S_0, S_1, \dots, S_n , die sogenannte *Items* enthalten: Das sind Produktionen, die einen Markierungspunkt \bullet enthalten, und bei denen an der gleichen Stelle oder links davon noch eine Trennstelle angegeben ist.

6.40 So, wie CYK aufgebaut war, weiß man:

- Hat die Trennstelle Nummer i und ist das Item in S_j , so ist sicher, dass man aus dem, was auf der rechten Seite der Produktion zwischen Trennstelle und \bullet steht, das Teilwort $w(i, j]$ der Eingabe ableiten kann.

- Eine Produktion $X \rightarrow \underset{i}{|} YZ \bullet$ wird in S_j aufgenommen, wenn es ein m gibt, so dass ein Item $Y \rightarrow \underset{i}{|} v \bullet$ in S_m ist und ein Item $Z \rightarrow \underset{m}{|} v' \bullet$ in S_j .

Die eben genannte Regel zur Berechnung von markierten Produktionen in CYK wird beim Algorithmus von Earley durch etwas Komplizierteres ersetzt. Bislang war die Markierung \bullet immer am Ende einer Produktion. Das wird nun verallgemeinert.

6.41 Definition. Wir betrachten im folgenden *Items* der Form $X \rightarrow \underset{i}{|} v_1 \bullet v_2$. Dabei ist immer $X \rightarrow v_1 v_2$ eine der ursprünglichen Grammatik-Produktionen, deren rechte Seite an einer beliebigen Stelle in Präfix v_1 und Suffix v_2 zerlegt werden darf. Insbesondere kann die Form $X \rightarrow \underset{i}{|} \bullet w$ vorliegen.

Ein Item $X \rightarrow \underset{i}{|} v_1 \bullet v_2$ wird immer nur zu Mengen S_j mit $i \leq j$ gehören.

6.42 Was könnte die Bedeutung eines solchen Items sein? Wie schon in der modifizierten Darstellung von CYK soll der \bullet wieder die Stelle markieren, „bis zu der man schon gekommen ist“, und es soll wieder sichergestellt sein, dass man aus dem, was auf der rechten Seite der Produktion zwischen Trennstelle und \bullet steht, das Teilwort $w(i, j]$ der Eingabe ableiten kann.

Für das Suffix, das im Item hinter dem Punkt steht, wird dagegen noch offen sein, ob aus ihm ein weiteres Stück der Eingabe ableitbar ist.

6.43 Algorithmus. (von Earley) Damit ist die Frage zu beantworten, nach welchen Regeln man ein Item $X \rightarrow \underset{i}{|} v_1 \bullet v_2$ zu einer Menge $S_{j'}$ hinzunimmt.

Lexer Regel: Der einfachste Fall liegt vor, wenn ein S_j ein Item von der Form $X \rightarrow \underset{i}{|} v_1 \bullet a v_2$ mit einem *Terminalsymbol* a unmittelbar hinter dem \bullet enthält. In diesem Fall muss geprüft werden, ob das nächste Eingabesymbol gerade a ist oder nicht:

- Wenn das der Fall ist, dann kann man das Item $X \rightarrow \underset{i}{|} v_1 a \bullet v_2$ zu S_{j+1} hinzufügen.
- Andernfalls unterlässt man das.

Die Lexer Regel besagt:

Wenn $X \rightarrow \underset{i}{|} v_1 \bullet a v_2$ in S_j und $a = w(j, j+1]$, dann kommt $X \rightarrow \underset{i}{|} v_1 a \bullet v_2$ zu S_{j+1} hinzu.

Completer Regel: Betrachten wir nun den Fall, dass ein S_m ein Item $X \rightarrow \underset{i}{|} v_1 \bullet Y v_2$ mit einem *Nichtterminalsymbol* Y nach dem \bullet enthält. Wann kann man, intuitiv gesprochen, den \bullet an Y vorbei schieben? Das ist dann sinnvoll, wenn man aus Y ein Teilwort der Eingabe ableiten kann, das an Trennstelle m beginnt. Das soll man aber gerade daran ablesen können, dass ein Item $Y \rightarrow \underset{m}{|} v \bullet$ in einem S_j ist für ein $j > m$.

Die Completer Regel besagt:

Wenn $X \rightarrow \underset{i}{|} v_1 \bullet Y v_2$ in S_m ist und $Y \rightarrow \underset{m}{|} v \bullet$ in S_j , dann kommt $X \rightarrow \underset{i}{|} v_1 Y \bullet v_2$ zu S_j hinzu.

Predictor Regel: Nachdem wir nun wissen, wie man den \bullet in einem Item nach hinten schieben kann, bleibt noch zu klären, woher Items der Form $Y \rightarrow \underset{i}{|} \bullet v_2$ mit dem \bullet am Anfang herkommen. Es erscheint jedenfalls sinnvoll, solch ein Item nur dann zu einem S_j hinzuzunehmen, wenn man „das Y brauchen kann“.

Die Predictor Regel besagt:

Wenn $X \rightarrow \underset{i}{|} v_1 \bullet Y v_2$ in S_j ist, dann kommt $Y \rightarrow \underset{j}{|} \bullet v$ zu S_j hinzu für jede Produktion $Y \rightarrow v$.

Der Algorithmus von Earley (genauer gesagt die Variante, die ohne sogenannten Look-Ahead in der Eingabe arbeitet) für eine Grammatik $G = (N, T, S, P)$ ergibt sich nun einfach wie folgt:

1. Man beginnt mit der Menge S_0 , die genau alle Items der Form $S \rightarrow \underset{0}{|} \bullet w$ für jede Produktion der Form $S \rightarrow w \in P$ (S ist das Startsymbol der Grammatik).
2. Auf eine Menge S_j werden solange die Regeln für Lexer, Completer und Predictor angewendet, bis sich nichts mehr ändert. Dann fährt man mit S_{j+1} fort.
Ist einmal ein $S_j = \emptyset$, dann ist die Eingabe nicht ableitbar.
Andernfalls muss man am Ende überprüfen, ob S_n ein Item der Form $S \rightarrow \underset{0}{|} w \bullet$ enthält. Wenn das der Fall ist (und nur dann), ist die Eingabe in der Grammatik ableitbar.

Der Nachweis, dass dieser Algorithmus wirklich das Richtige tut, wurde von Jay Earley geführt.

6.44 Beispiel. Um ein Beispiel vollständig und übersichtlich durchführen zu können, betrachten wir nun die folgende sehr einfache Grammatik $G = (N, T, S, P)$ mit $N = \{S, X, A, B, C\}$, $T = \{a, b, c\}$, Startsymbol S und Produktionsmenge $P = \{S \rightarrow AX, X \rightarrow BC, A \rightarrow a, B \rightarrow b, C \rightarrow c\}$.

In Abbildung 6.7 sind für die Eingabe abc die vollständigen Mengen S_0, \dots, S_3 angegeben, wie sie der Algorithmus von Earley konstruiert.

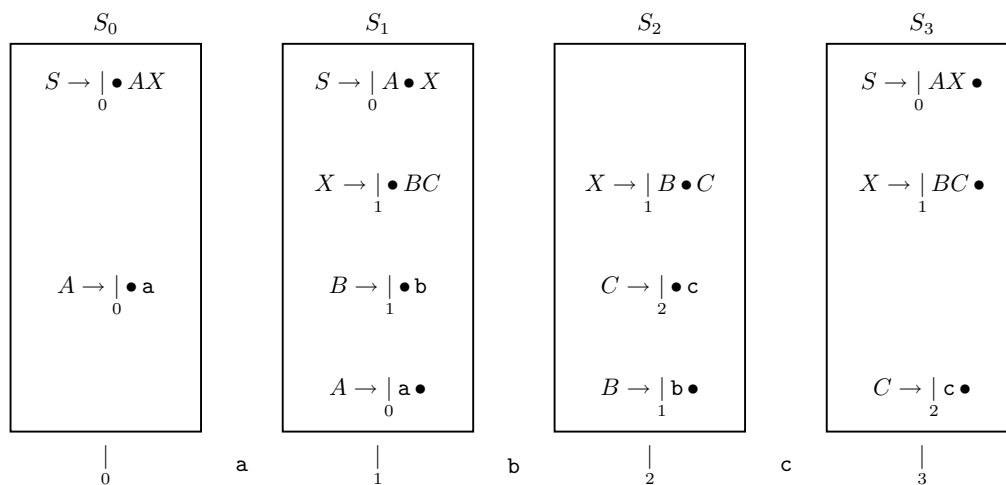


Abbildung 6.7: Der Algorithmus von Earley für eine einfache Beispielgrammatik

Zum besseren Verständnis ist in Abbildung 6.8 zusätzlich noch durch Pfeile angegeben, nach welchen Regeln sich die einzelnen Items ergeben. Von der Lexer-Regel bewirkte Schritte sind durch durchgezogene Linien gekennzeichnet, von der Predictor-Regel bewirkte durch gestrichelte Linien und von der Completer-Regel bewirkte durch gepunktete Linien.

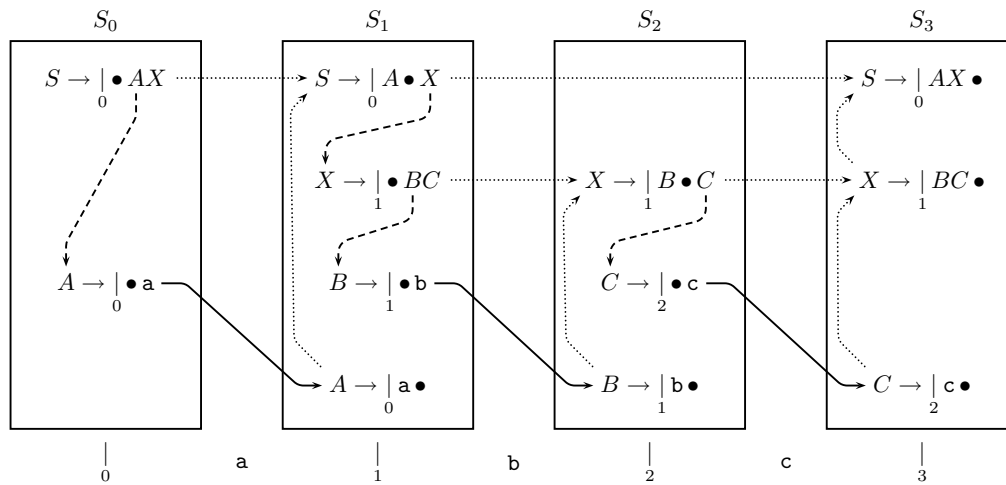


Abbildung 6.8: Der Algorithmus von Earley für eine einfache Beispielgrammatik

Der Rest dieses Kapitels ab hier wird behandelt und ist prüfungsrelevant.

6.7 LR Parsing

Für den später noch definierten Spezialfall der *LALR(1)*-Grammatiken stehen z. B. mit yacc resp. der GNU-Variante bison und anderen mehrere Parser-Generatoren in und für C zur Verfügung; in und für Java ist zum Beispiel Java CUP gedacht.

Zu Demonstrationszwecken benutzen wir im Weiteren vor allem die folgende Grammatik zur Beschreibung einfacher arithmetischer Ausdrücke.

- 6.45 Beispiel.** Nichtterminalsymbole sind E , T und F (die für die englischen Ausdrücke expression, term und factor stehen). Terminalsymbole sind $*$, $+$, $($, $)$ und a . Das Startsymbol ist E und die Produktionen lauten:

$$\begin{aligned} E &\rightarrow E+T \mid T \\ T &\rightarrow T*F \mid F \\ F &\rightarrow (E) \mid a \end{aligned}$$

- 6.46 Vereinbarung.** Im Folgenden wollen wir immer nur *ergänzte* Grammatiken betrachten, bei denen ein zusätzliches Nichtterminalsymbol S' eingeführt wird, das das neue Startsymbol wird. Für S' gibt es genau eine Produktion, nämlich $S' \rightarrow \neg S \vdash$, wobei S das Startsymbol der ursprünglichen Grammatik sei. Die Zeichen \neg und \vdash sind genau genommen neue Terminalsymbole. Das \neg wird immer als unterstes Kellersymbol fungieren, und \vdash markiert das Ende der Eingabe.

Aus der obigen Grammatik wird dann also die mit der Produktionsmenge

$$\begin{aligned} E' &\rightarrow \neg E \vdash \\ E &\rightarrow E+T \mid T \\ T &\rightarrow T*F \mid F \\ F &\rightarrow (E) \mid a \end{aligned}$$

- 6.47 Definition.** Eine Grammatik hat die Eigenschaft $LR(k)$ für ein $k \in \mathbb{N}_0$, wenn für zwei beliebige Rechtsableitungen

$$\begin{aligned} S &\xRightarrow{*} \xi_1 A_1 \omega \eta_1 \xRightarrow{r} \xi_1 \gamma_1 \omega \eta_1 \xRightarrow{*} \alpha \omega \eta_1 \in T^* \\ S &\xRightarrow{*} \xi_2 A_2 \tau_2 \xRightarrow{r} \xi_1 \gamma_1 \omega \eta_2 \xRightarrow{*} \alpha \omega \eta_2 \in T^* \end{aligned}$$

mit $|\omega| = k$ oder $\eta_1 = \eta_2 = \varepsilon$ immer gilt: $\xi_2 A_2 \tau_2 = \xi_1 A_1 \omega \eta_2$.

Mit anderen Worten: Wenn $\tau_2 \in T^*$ ist, wenn also A_2 das zuletzt durch eine Reduktion entstandene Zeichen ist, dann ist $A_1 = A_2$, $\xi_1 = \xi_2$ und $\tau_2 = \omega \eta_2$. Das heißt, *durch den Kellerinhalt ($\xi_1 \gamma_1$) und die nächsten k Eingabezeichen (ω) ist der nächste Reduktionsschritt bei der Bottom-Up-Syntaxanalyse eindeutig bestimmt.*

- 6.48** In der Abkürzung LR steht das L dafür, dass die Eingabe von links gelesen wird, und das R steht dafür, dass eine Rechtsableitung konstruiert wird. Im folgenden werden wir uns auf den Fall $k \leq 1$ beschränken.

Für eine Grammatik G ist zu gegebenem k entscheidbar, ob sie $LR(k)$ ist oder nicht. Für eine Grammatik G ist *nicht* entscheidbar, ob sie für irgendein k die Eigenschaft $LR(k)$ hat.

6.49 (Prinzip der Syntaxanalyse für $LR(k)$ -Grammatiken) Wenn eine Grammatik $LR(k)$ ist, dann gibt es eine Funktion act der Art $act(\kappa, \omega) = \text{„nächste Aktion des KA“}$ für jeden Kellerinhalt κ und jede(s) bevorstehende Eingabe(bruchstück) ω .

Wegen der unendlich vielen möglichen κ kann man diese Funktion nicht als Tabelle speichern. (Das möchte man aber gerne, damit man Parser leicht automatisch erzeugen kann.) Deswegen bildet man Äquivalenzklassen von Kellerinhalten, die zum gleichen Verhalten führen. Diese Äquivalenzklassen heißen auch *Zustände*. Die Menge aller solcher Zustände heie Z . Für den Zustand, der zu einem Kellerinhalt κ gehört, schreiben wir auch $z(\kappa)$.

Die Äquivalenzrelation soll erstens so sein, dass es nur endlich viele Äquivalenzklassen gibt. Außerdem soll act in der Form $act(z(\kappa), \omega) = \text{„nächste Aktion des KA“}$ aufgeschrieben werden können.

Wird auf einen Keller κ ein weiteres Symbol X oben auf gelegt, dann möchte man gerne $z(\kappa X)$ „leicht“ berechnen können. Am bequemsten ist eine Funktion $goto$ mit $goto(z(\kappa), X) = z(\kappa X)$.

Angenommen, man hat eine solche Äquivalenzrelation und Funktionen act und $goto$. Dann kann man für $LR(k)$ -Grammatiken einen deterministischen Algorithmus für Bottom-Up-Syntaxanalyse angeben. Wir beschreiben zunächst eine kleine Erweiterung unseres Modelles der Kellerautomaten und anschließend das Prinzip des Analysealgorithmus:

Automatenmodell: Man erweitert den KA um einen zweiten Keller, in dem Zustände gespeichert werden können. (Alternativ kann man sich auch vorstellen, dass es bei einem Keller bleibt, in dem aber streng abwechselnd Grammatiksymbole und Zustände abgelegt werden). Beide Keller sind immer gleich weit gefüllt. Für das i -te Element z des Zustandskellers gilt stets $z = z(\kappa)$, wobei κ der Inhalt der unteren i Stellen des Symbolkellers ist:

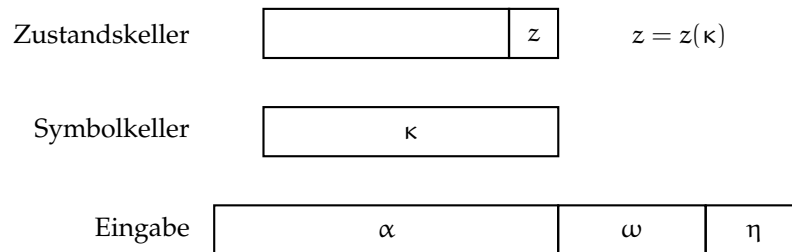


Abbildung 6.9: Modifizierter Kellerautomat

Algorithmus: Im wesentlichen sind zwei Fälle zu betrachten, nämlich dass ein weiteres Eingabesymbol gekellert werden soll ($act(z(\kappa), \omega) = shift$), und dass das obere Ende γ des Symbolkellers mittels einer Produktion $A \rightarrow \gamma$ reduziert werden soll ($act(z(\kappa), \omega) = reduce\ A \rightarrow \gamma$).

Shift: In diesem Fall wird das nächste Eingabesymbol a auf den Symbolkeller gelegt und auf den Zustandskeller $goto(z, a)$, wobei z der vorher zuoberst liegende Zustand sei, der ebenfalls im Keller bleibt. Aus Abbildung 6.9 ergibt sich dann die in Abbildung 6.10 dargestellte Situation.

Reduce: In diesem Fall wird das obere Ende γ aus dem Symbolkeller $\kappa = \xi\gamma$ entfernt und durch A ersetzt. Aus dem Zustandskeller werden ebenfalls $|\gamma|$ viele Elemente entfernt. Bezeichnet z'' den darunter befindlichen Zustand, so wird darauf als neuer Zustand $goto(z'', A)$ gelegt. Aus Abbildung 6.9 ergibt sich dann die in Abbildung 6.11 dargestellte Situation.

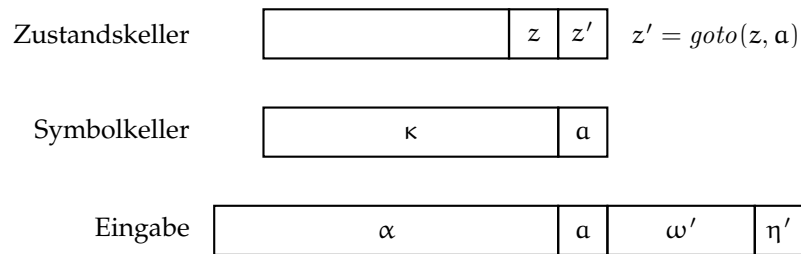


Abbildung 6.10: Modifizierter KA nach einer Shift-Operation

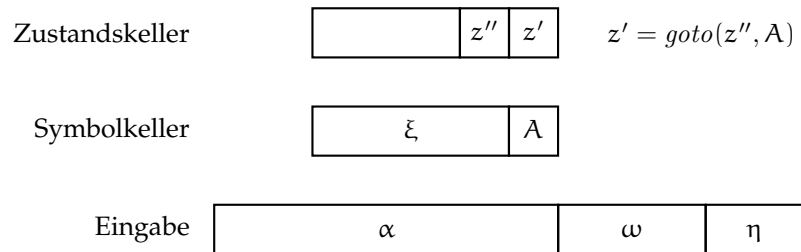


Abbildung 6.11: Modifizierter KA nach einer Reduce-Operation

Im übrigen sind für $act(z, \omega)$ noch zwei weitere Fälle möglich: Zum einen gibt es die Aktion „Akzeptieren“ (*accept*) und zum anderen die Aktion „Fehler“ (*error*).

Zu Beginn der Analyse einer Eingabe wird in den Symbolkeller als unterstes Symbol ein \vdash gelegt und in den Zustandskeller entsprechend $z(\vdash)$.

Der Automat ist fertig, wenn im Symbolkeller nur noch $\vdash S \vdash$ steht und das zu S' reduziert werden kann. Genau genommen kann der Kellerautomat auch schon akzeptieren, wenn im Symbolkeller $\vdash S$ steht und als letztes Eingabesymbol \vdash ansteht.

6.50 Beispiel. Für die Grammatik aus Beispiel 6.45 seien die Produktionen wie nachfolgend angegeben durchnummeriert:

$$\begin{array}{ll}
 E' \rightarrow \vdash E \vdash & \\
 E \rightarrow E+T \mid T & R_1 \mid R_2 \\
 T \rightarrow T * F \mid F & R_3 \mid R_4 \\
 F \rightarrow (E) \mid a & R_5 \mid R_6
 \end{array}$$

Die folgende Tabelle skizziert eine Menge von Zuständen und Funktionswerte für act und $goto$. Wie man auf diese Tabelle kommen kann, werden wir später sehen. Zunächst geht es nur darum, sie anschließend beispielhaft für die Analyse einer Eingabe zu benutzen.

| | | $act(z, \omega)$ | | | | | | $goto(z, X)$ | | | | | | | |
|-----|------------------------------|------------------|-------|-------|-----|-------|----------|--------------|-----|-----|-----|-----|-----|-----|-----|
| z | κ | a | $+$ | $*$ | $($ | $)$ | \vdash | a | $+$ | $*$ | $($ | $)$ | E | T | F |
| 0 | \neg | S | | | | S | | 11 | | | 9 | | 1 | 5 | 8 |
| 1 | $\neg E$ | | S | | | | A | | 3 | | | 10 | | | |
| 2 | $\dots (E$ | | S | | | S | | | 3 | | | | | | |
| 3 | $\dots E+$ | S | | | | S | | 11 | | | 9 | | | 4 | 8 |
| 4 | $\dots E+T$ | | R_1 | S | | R_1 | R_1 | | | 6 | | | | | |
| 5 | $\neg T, \dots (T$ | | R_2 | S | | R_2 | R_2 | | | 6 | | | | | |
| 6 | $\dots T*$ | S | | | | S | | 11 | | | 9 | | | | 7 |
| 7 | $\dots T*F$ | | R_3 | R_3 | | R_3 | R_3 | | | | | | | | |
| 8 | $\dots +F, \dots (F, \neg F$ | | R_4 | R_4 | | R_4 | R_4 | | | | | | | | |
| 9 | $\dots ($ | S | | | | S | | 11 | | | 9 | | 2 | 5 | 8 |
| 10 | $\dots (E)$ | | R_5 | R_5 | | R_5 | R_5 | | | | | | | | |
| 11 | $\dots a$ | | R_6 | R_6 | | R_6 | R_6 | | | | | | | | |

In der Tabelle für act stehe

- S für eine Shift-Operation,
- jedes R_i für die Reduce-Operation mit der entsprechenden Produktion,
- A für Akzeptieren und
- ein leeres Feld für einen Fehlerfall.

Anhand der Tatsache, dass die Spalten mit jeweils einem einzelnen Symbol beschriftet sind, erkennt man, dass $|\omega| = 1$ genügt, d. h. in der Eingabe muss nur ein Zeichen vorausgelesen werden, um die richtige Entscheidung fällen zu können.

In der Tabelle für $goto$ stehe jede Zahl für den entsprechenden Zustand.

In Tabelle 6.1 ist die Arbeitsweise des modifizierten KA für die Eingabe $(a+a)*a \vdash$ vollständig angegeben.

Es stellt sich nun die Frage, wie man zu Tabellen wie für die Beispielgrammatik kommt.

6.51 Definition. Zu jeder Produktion $A \rightarrow w = X_1 X_2 \dots X_k = \gamma$ mit beliebigen Terminal- und Nicht-terminalsymbolen X_i gehört die Menge $I(A \rightarrow \gamma)$ der folgenden $k+1$ sogenannten *markierten Produktionen* (engl. *items* oder engl. *LR(0) items*), bei denen jeweils eine Stelle neben einem Symbol durch einen Punkt gekennzeichnet ist: $I(A \rightarrow w) = \{A \rightarrow \bullet X_1 X_2 \dots X_k, A \rightarrow X_1 \bullet X_2 \dots X_k, A \rightarrow X_1 X_2 \bullet \dots X_k, \dots, A \rightarrow X_1 \dots \bullet X_k, A \rightarrow X_1 \dots X_k \bullet\}$. Zu einer Produktion $A \rightarrow \varepsilon$ gibt genau ein Item, nämlich $A \rightarrow \bullet$.

6.52 Die Zustände werden nun als Mengen von markierten Produktionen konstruiert, und zwar so, dass $z(\kappa)$ gerade diejenigen $A \rightarrow \mu \bullet \nu$ enthält, für die gilt:

- $A \rightarrow \mu \nu \in P$,
- $\exists \xi \in V^* : \kappa = \xi \mu$ und
- $\exists \eta \in T^* : S' \xRightarrow{r} \xi A \eta$

Das macht man so: Für eine Menge M markierter Produktionen bezeichne $kompl(M)$ die Vervollständigung gemäß der Predictor-Regel des Algorithmus von Earley. Falls Sie diesen Abschnitt überlesen haben, ist hier eine explizite Beschreibung:

- Es ist $M \subseteq kompl(M)$.
- Wenn $A \rightarrow \mu \bullet B \nu \in kompl(M)$ und $B \rightarrow \gamma \in P$ ist, dann ist auch $B \rightarrow \bullet \gamma \in kompl(M)$.

| | | κ | ω | η | \dots |
|--|--|------------|----------|--------------|--------------------|
| | | 0 | | | |
| | | \vdash | (| a | + a) * a \vdash |
| | | 0 9 | | | |
| | | \vdash | (| a | + a) * a \vdash |
| | | 0 9 11 | | | |
| | | \vdash | (| a | + a) * a \vdash |
| | | 0 9 8 | | | |
| | | \vdash | (| F | + a) * a \vdash |
| | | 0 9 5 | | | |
| | | \vdash | (| T | + a) * a \vdash |
| | | 0 9 2 | | | |
| | | \vdash | (| E | + a) * a \vdash |
| | | 0 9 2 3 | | | |
| | | \vdash | (| E | + a) * a \vdash |
| | | 0 9 2 3 11 | | | |
| | | \vdash | (| E | + a) * a \vdash |
| | | 0 9 2 3 8 | | | |
| | | \vdash | (| E | + F) * a \vdash |
| | | 0 9 2 3 4 | | | |
| | | \vdash | (| E | + T) * a \vdash |
| | | 0 9 2 | | | |
| | | \vdash | (| E |) * a \vdash |
| | | 0 9 2 10 | | | |
| | | \vdash | (| E |) * a \vdash |
| | | 0 8 | | | |
| | | \vdash | F | * a \vdash | |
| | | 0 5 | | | |
| | | \vdash | T | * a \vdash | |
| | | 0 5 6 | | | |
| | | \vdash | T | * a \vdash | |
| | | 0 5 6 11 | | | |
| | | \vdash | T | * a \vdash | |
| | | 0 5 6 7 | | | |
| | | \vdash | T | * F \vdash | |
| | | 0 5 | | | |
| | | \vdash | T | \vdash | |
| | | 0 1 | | | |
| | | \vdash | E | \vdash | |

Tabelle 6.1: Verarbeitung der Eingabe (a+a)*a

(iii) Punkt (ii) iteriert man, bis sich nichts mehr ändert.

Mit anderen Worten: $\text{kompl}(M) = M \cup \{B \rightarrow \bullet \gamma \mid A \rightarrow \mu \bullet Bv \in \text{kompl}(M)\}$. Wir sagen auch, dass $\text{kompl}(M)$ durch *Komplettierung* oder *Vervollständigung von M* entsteht.

Die $z(\kappa)$ kann man nun rekursiv wie folgt konstruieren:

- Man beginnt mit einer Menge $M_0 = \{S' \rightarrow \cdot S \vdash\} \subseteq z(\cdot)$.
- Wann immer man eine Menge M hat, komplettiert man sie zu einem Zustand $z = \text{kompl}(M)$.
- Wenn man einen Zustand $z(\kappa)$ hat, erzeugt man durch die folgende *Fortschreibungsregel* Mengen, die sogenannte Kerne für Zustände $z(\kappa X)$ sind: Wenn $A \rightarrow \mu \bullet X \nu \in z(\kappa)$ ist, dann ist $A \rightarrow \mu X \bullet \nu \in z(\kappa X)$. Diese werden ihrerseits komplettiert usw.

6.53 Beispiel. Wir führen die Konstruktion der Zustände anhand unserer Beispielgrammatik für arithmetische Ausdrücke durch.

$z_0 = z(\cdot)$: Wir beginnen mit der markierten Produktion $E' \rightarrow \cdot E \vdash \in z_0$. Durch Vervollständigung kommen zunächst $E \rightarrow \bullet E+T$ und $E \rightarrow \bullet T$ hinzu, dann $T \rightarrow \bullet T*F$ und $T \rightarrow \bullet F$ und schließlich $F \rightarrow \bullet (E)$ und $F \rightarrow \bullet a$.

Also $z_0 = \{E' \rightarrow \cdot E \vdash, E \rightarrow \bullet E+T, E \rightarrow \bullet T, T \rightarrow \bullet T*F, T \rightarrow \bullet F, F \rightarrow \bullet (E), F \rightarrow \bullet a\}$. Durch die Fortschreibungsregel kommt man nun weiter. Für jedes (Terminal- oder Nichtterminal-)Symbol, an dem die Marke vorbeigeschoben werden kann, ergibt sich ein neuer Zustand.

$z_1 = z(\cdot E)$: Zunächst hat man $E' \rightarrow \cdot E \vdash$ und $E \rightarrow E \bullet +T$. Durch Vervollständigung kommen keine weiteren markierten Produktionen hinzu.

$z_2 = z(\cdot T)$: Zunächst hat man $E \rightarrow T \bullet$ und $T \rightarrow T \bullet *F$. Durch Vervollständigung kommen keine weiteren markierten Produktionen hinzu.

$z_3 = z(\cdot F)$: Zunächst hat man $T \rightarrow F \bullet$. Durch Vervollständigung kommen keine weiteren markierten Produktionen hinzu.

$z_4 = z(\cdot a)$: Zunächst hat man $F \rightarrow a \bullet$. Durch Vervollständigung kommen keine weiteren markierten Produktionen hinzu.

$z_5 = z(\cdot ())$: Zunächst hat man $F \rightarrow (\bullet E)$. Durch Vervollständigung kommen sukzessive die folgenden markierten Produktionen hinzu: $E \rightarrow \bullet E+T, E \rightarrow \bullet T, T \rightarrow \bullet T*F, T \rightarrow \bullet F, F \rightarrow \bullet (E), F \rightarrow \bullet a$.

$z_6 = z(\cdot) = \dots$: Dies ist der Fehlerzustand $z_6 = \emptyset$.

$z_7 = z(\cdot E+)$: Aus z_1 erhält man zunächst durch die Fortschreibungsregel $E \rightarrow E+ \bullet T$. Durch Vervollständigung kommen sukzessive die folgenden markierten Produktionen hinzu: $T \rightarrow \bullet T*F, T \rightarrow \bullet F, F \rightarrow \bullet (E), F \rightarrow \bullet a$.

$z_8 = z(\cdot E\vdash)$: Das ist der akzeptierende Zustand (aus z_1 durch Fortschreibungsregel).

Damit ist für z_1 die Fortschreibungsregel vollständig ausgenutzt.

$z_9 = z(\cdot T*)$: Aus z_2 erhält man zunächst durch die Fortschreibungsregel $T \rightarrow T* \bullet F$. Durch Vervollständigung kommen sukzessive die folgenden markierten Produktionen hinzu: $F \rightarrow \bullet (E), F \rightarrow \bullet a$.

Für z_3 und z_4 ist die Fortschreibungsregel nicht anwendbar.

$z_{10} = z(\cdot (E))$: Aus z_5 erhält man zunächst durch die Fortschreibungsregel $F \rightarrow (E \bullet)$ und $E \rightarrow E \bullet +T$. Durch Vervollständigung kommt nichts hinzu.

$z_{11} = z(\cdot (E+T))$: Aus z_7 erhält man zunächst durch die Fortschreibungsregel $E \rightarrow E+T \bullet$ und $T \rightarrow T \bullet *F$. Durch Vervollständigung kommt nichts hinzu.

$z_{12} = z(\neg T * F)$: Aus z_9 erhält man durch die Fortschreibungsregel $T \rightarrow T * F \bullet$.

$z_{13} = z(\neg (E))$: Aus z_{10} erhält man durch die Fortschreibungsregel $F \rightarrow (E) \bullet$.

6.54 Mit anderen Worten: $z(\kappa X) = \text{kompl}(\{A \rightarrow \mu X \bullet v \mid A \rightarrow \mu \bullet Xv \in z(\kappa)\})$. Das definiert auch gleich die Funktion $\text{goto}(z(\kappa), X)$.

6.55 Beispiel. Damit ergibt sich für die Beispielgrammatik folgende Tabelle für goto .

| | E | T | F | a | $($ | $)$ | $+$ | $*$ | \vdash |
|----------|----------|----------|----------|-------|-------|----------|-------|-------|----------|
| z_0 | z_1 | z_2 | z_3 | z_4 | z_5 | z_6 | z_6 | z_6 | z_6 |
| z_1 | z_6 | z_6 | z_6 | z_6 | z_6 | z_6 | z_7 | z_6 | z_8 |
| z_2 | z_6 | z_6 | z_6 | z_6 | z_6 | z_6 | z_6 | z_9 | z_6 |
| z_3 | z_6 | z_6 | z_6 | z_6 | z_6 | z_6 | z_6 | z_6 | z_6 |
| z_4 | z_6 | z_6 | z_6 | z_6 | z_6 | z_6 | z_6 | z_6 | z_6 |
| z_5 | z_{10} | z_2 | z_3 | z_4 | z_5 | z_6 | z_6 | z_6 | z_6 |
| z_6 | z_6 | z_6 | z_6 | z_6 | z_6 | z_6 | z_6 | z_6 | z_6 |
| z_7 | z_6 | z_{11} | z_3 | z_4 | z_5 | z_6 | z_6 | z_6 | z_6 |
| z_8 | | | | | | | | | |
| z_9 | z_6 | z_6 | z_{12} | z_4 | z_5 | z_6 | z_6 | z_6 | z_6 |
| z_{10} | z_6 | z_6 | z_6 | z_6 | z_6 | z_{13} | z_7 | z_6 | z_6 |
| z_{11} | z_6 | z_6 | z_6 | z_6 | z_6 | z_6 | z_6 | z_9 | z_6 |
| z_{12} | z_6 | z_6 | z_6 | z_6 | z_6 | z_6 | z_6 | z_6 | z_6 |
| z_{13} | z_6 | z_6 | z_6 | z_6 | z_6 | z_6 | z_6 | z_6 | z_6 |

6.56 Es bleibt die Frage, wie man die Funktion act findet. Die prinzipielle Idee besteht darin, sogenannte *Look-ahead-Mengen* $la(z, a \rightarrow \mu \bullet v)$ und $la(z, \text{shift})$ von Terminalzeichen zu finden, so dass für jedes z alle diese Mengen disjunkt sind. Dann definieren die folgenden Festlegungen act konsistent und sinnvoll:

$$f(z, a) = \begin{cases} \text{reduce } A \rightarrow \gamma & \text{falls } a \in la(z, A \rightarrow \mu \bullet) \\ \text{shift} & \text{falls } a \in la(z, \text{shift}) \\ \text{accept} & \text{falls } z = z(\neg S \vdash \bullet) \\ & \text{(oder falls } \neg S \bullet \vdash \in z \text{ und } a = \vdash) \\ \text{error} & \text{sonst} \end{cases}$$

Je nachdem, wieviel Mühe man sich gibt und wie kompliziert die Grammatik ist, führen mehr oder weniger aufwendige Konstruktionen ans Ziel. Wir beschränken uns hier auf folgende Mitteilung:

6.57 Definition. Eine Grammatik ist $LALR(1)$ (engl. *look ahead LR(1)*), falls für jedes z die wie folgt definierten la -Mengen paarweise disjunkt sind:

$$\begin{aligned} la(z, S' \rightarrow \neg \bullet S \vdash) &= \emptyset \\ a \in la(z, A \rightarrow \mu \bullet Bv) \wedge b \in \text{anf}(va) &\Rightarrow b \in la(z, B \rightarrow \bullet \gamma) \\ a \in la(z, A \rightarrow \mu \bullet Xv) &\Rightarrow a \in la(\text{goto}(z, X), A \rightarrow \mu X \bullet v) \end{aligned}$$

6.8 Nach der Syntaxanalyse

Die Analyse eines Eingabewortes auf syntaktische Korrektheit gemäß z. B. einer $LALR(1)$ -Grammatik ist nicht schwer. Dementsprechend macht auch z. B. der dafür nötige Quellcode eines modernen Übersetzers nur einen relativ kleinen Anteil aus.

- 6.58** Aber an die Syntaxanalyse schließen sich noch weitere Übersetzerteile an. Der erste ist die *semantische Analyse*. Damit bezeichnet man üblicherweise den Teil der Analyse, der sich mit der Definition desjenigen Teiles der Syntax befasst, der nicht mit Hilfe kontextfreier Grammatiken behandelt werden kann. Ein typisches Beispiel ist die Überprüfung aller Verwendungen von Variablen darauf hin, ob sie auch deklariert sind und ob die bei der Deklaration angegebenen Typen an der Verwendungsstelle passen.

Auf die semantische Analyse folgen *Codeerzeugung* und *Codeoptimierung*. Moderne Compiler gehen üblicherweise so vor, dass sie das Programm zunächst in eine Zwischensprache übersetzen, die sozusagen von einem idealisierten Prozessor als Grundlage ausgeht. Auf dieser Ebene werden auch schon Optimierungen vorgenommen. Anschließend wird das Zwischenspracheprogramm in ein Assemblerprogramm für die tatsächliche Zielarchitektur übersetzt, das anschließend u. U. noch weiter optimiert wird.

Ziel dieses Abschnittes ist es im wesentlichen, noch eine Verallgemeinerung kontextfreier Grammatiken einzuführen, bei denen den Produktionen auf die ein oder andere Weise auch noch „semantische Regeln“ hinzugefügt sind. Vor allem werden wir anhand von Beispielen deren Nützlichkeit für semantische Analyse und Codeerzeugung kennenlernen. Außerdem werden sehen, dass für Spezialfälle gewisser sogenannter attributierter Grammatiken die zusätzliche Arbeit gut von dem Kellerautomaten übernommen werden kann, der die Syntaxanalyse macht.

Wir werden im folgenden gelegentlich kleine Programmfragmente notieren.

Grammatiken mit semantischen Aktionen

- 6.59** Eine *Grammatik mit semantischen Aktionen* ist eine kontextfreie Grammatik, bei der jede rechte Seite einer Produktion aus einer Folge von Terminal- und Nichtterminalsymbolen besteht, neben/zwischen denen zusätzlich ein Programmfragment eingefügt ist. Im allgemeinen hat eine Produktion also die Struktur

$$A \rightarrow \{code_0\} X_1 \{code_1\} X_2 \cdots X_m \{code_m\}$$

Codefragmente dürfen leer sein; in diesem Fall lässt man auch die geschweiften Klammern weg.

Ein Codefragment am Ende einer Produktion wird vom Übersetzer ausgeführt, nachdem er auf dem Keller eine Reduktion gemäß der entsprechenden kontextfreien Produktion durchgeführt hat.

Für die anderen Codestücke kann man sich vorstellen, man hätte zusätzliche neue Nichtterminalsymbole M_0, \dots, M_{m-1} eingeführt und die obige Schreibweise wäre eine Abkürzung für

$$\begin{aligned} A &\rightarrow M_0 X_1 M_1 X_2 \cdots X_m \{code_m\} \\ M_0 &\rightarrow \varepsilon \{code_0\} \\ M_1 &\rightarrow \varepsilon \{code_1\} \\ &\vdots \\ M_{m-1} &\rightarrow \varepsilon \{code_{m-1}\} \end{aligned}$$

- 6.60 Beispiel.** Wir erweitern unsere Grammatik für arithmetische Ausdrücke wie folgt:

```

 $E \rightarrow E+T$  { printf("Reduktion mit E -> E+T \n"); }
 $E \rightarrow T$  { printf("Reduktion mit E -> T \n"); }
 $T \rightarrow T * F$  { printf("Reduktion mit T -> T * F \n"); }
 $T \rightarrow F$  { printf("Reduktion mit T -> F \n"); }
 $F \rightarrow (E)$  { printf("Reduktion mit F -> (E) \n"); }
 $F \rightarrow a$  { printf("Reduktion mit F -> a \n"); }

```

Das führt dazu, dass der Übersetzer jedes Mal, wenn er eine Reduktion auf dem Keller durchführt, die benutzte Produktion ausgibt.

Dieser Seiteneffekt ist natürlich nichts, was man von einem echten Übersetzer will, aber für Lehrzwecke ist es sehr hilfreich.

Wozu also kann man solche semantischen Aktionen benutzen?

- 6.61** Bei der Syntaxanalyse wird (zumindest implizit) der Ableitungsbaum für die Eingabe aufgebaut. Viele der danach noch zu erledigenden Aufgaben lassen sich in der folgenden Art auffassen: Zu jedem inneren Knoten des Ableitungsbaumes gehört eine Datenstruktur, die im Allgemeinen als Verbund (engl. *record*) angesehen werden kann. Die Komponenten dieses Verbundes nennt man auch *Attribute*. Die Aufgabe besteht darin, alle Attributwerte aller inneren Knoten zu bestimmen.

Üblicherweise werden einige Attributwerte der Blätter (i. e. der Terminalsymbole) von der lexikalischen Analyse geliefert, und einige Attributwerte des Startsymbols sind auch klar. Die Attribute eines inneren Knotens hängen von den Attributwerten anderer Knoten „in der Umgebung“ ab. Dazu gehören die Nachfolgeknoten im Ableitungsbaum, der Vorgängerknoten und z. B. evtl. noch die Knoten mit gleichem Vorgänger links vom betrachteten Knoten.

Wegen der Einschränkungen auf benachbarte Knoten ist es oft möglich, Attributwerte mit Hilfe semantischer Aktionen zu berechnen.

- 6.62 Beispiel.** Wir betrachten die folgende Variante unserer Grammatik für arithmetische Ausdrücke, in der als Terminalsymbole nur ganzzahlige Konstanten erlaubt seien, für die die lexikalische Analyse das Token **num** liefere. Wir wollen nun den Fall betrachten, dass schon der Übersetzer solche Ausdrücke auswerten soll. Dazu gebe es für alle Terminal- und Nichtterminalsymbole ein Attribut *val* vom Typ *integer*.

Für das Terminalsymbol **num** liefere die lexikalische Analyse als Attributwert für *val* den Wert der zugehörigen Zahlkonstante. Damit ergibt sich zum Beispiel für die Eingabe $2+3*4$ der Ableitungsbaum wie in Abbildung 6.12 auf der linken Seite dargestellt. Den Wert des Attributes *val* eines Knotens des Ableitungsbaumes, für den ein indiziertes Nichtterminalsymbol X steht, notieren wir als $X.val$.

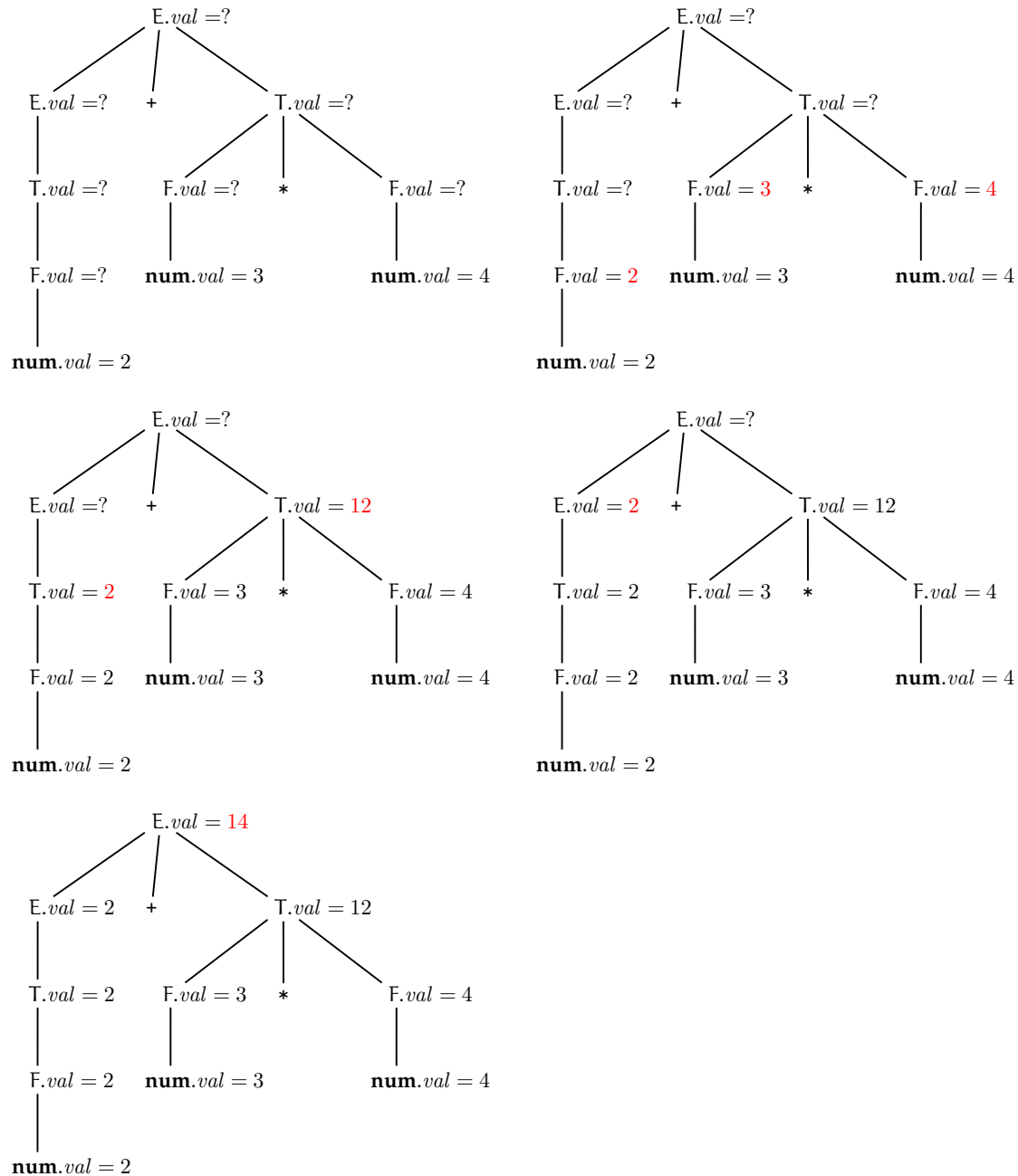
Es ist aber klar, dass man durch Hinaufreichen der Attributwerte von Knoten zu ihrem Vorgänger und gegebenenfalls Anwenden der naheliegenden, der Produktion entsprechenden Operation den Attributwert *val* für alle Nichtterminalknoten berechnen kann. Es ergibt sich dann der in Abbildung 6.12 auf der rechten Seite dargestellte attributierte Ableitungsbaum.

Notieren kann man das zum Beispiel wie folgt. Da ein Nichtterminalsymbol mehrfach in einer Produktion auftreten kann, unterscheiden wir die verschiedenen Vorkommen durch Indizes.

```

 $E_0 \rightarrow E_1+T_1$  {  $E_0.val = E_1.val + T_1.val$  }
 $E_0 \rightarrow T_1$  {  $E_0.val = T_1.val$  }
 $T_0 \rightarrow T_1 * F_1$  {  $T_0.val = T_1.val \cdot F_1.val$  }
 $T_0 \rightarrow F_1$  {  $T_0.val = F_1.val$  }
 $F_0 \rightarrow (E_1)$  {  $F_0.val = E_1.val$  }
 $F_0 \rightarrow \text{num}$  {  $F_0.val = \text{num.val}$  }

```

Abbildungung 6.12: Schrittweise Berechnung der Werte eines Attributs in einem Ableitungsbaum

Da die Berechnung der Attributwerte „von unten nach oben“ erfolgt, kann man sie gut in eine Bottom-Up-Syntaxanalyse integrieren. Bei jedem Reduktionsschritt $X \rightarrow X_1 \cdots X_k$ werden die Attributwerte der Kellersymbole X_1, \dots, X_k verarbeitet und daraus der Attributwert für das neue Kellersymbol X berechnet.

Attributierte Grammatiken

- 6.63** Eine *attributierte Grammatik* ist eine kontextfreie Grammatik $G(N, T, S, P)$ zusammen mit einer endlichen Menge A von *Attributen* $a \in A$ (mit je einem Wertebereich V_a).

Zu jedem Symbol $X \in N \cup T$ sind zwei disjunkte Attributmengen $D_X \subset A$ und $I_X \subset A$ sogenannter *synthetisierter* (engl. *synthesized* oder *derived*) resp. *vererbter* (engl. *inherited*) Attribute festgelegt.

Zu jeder Produktion $p : X_0 \rightarrow X_1 \cdots X_m$ ist für jedes $a \in D_{X_0}$ eine Funktion spezifiziert, die in Abhängigkeit von allen Attributen aller auf der rechten Seite vorkommenden Symbole einen Wert für das Attribut a festlegt.

Zu jeder Produktion $p : X_0 \rightarrow X_1 \cdots X_m$ ist für jedes $i = 1, \dots, m$ und jedes $a \in I_{X_i}$ eine Funktion spezifiziert, die in Abhängigkeit von allen Attributen aller vorkommenden Symbole einen Wert für das Attribut a festlegt.

Ein Wort gilt als von einer attribuierten Grammatik erzeugt, wenn es von der kontextfreien Grammatik erzeugt wird und es eine Belegung aller im Ableitungsbaum vorkommenden Attribute mit Werten gibt, die den vorgeschriebenen Funktionen genügen.

- 6.64** Eine Grammatik heißt *S-attribuiert*, wenn sie nur synthetisierte Attribute enthält. Solche Grammatiken sind sehr nützlich. Zum Glück sind sie auch für Bottom-up-Syntaxanalyse gut geeignet.

In diesem Fall kann der Kellerautomat wie folgt vorgehen: Wann immer er die rechte Seite einer Produktion $A \rightarrow w$ reduziert, kann er auch aus den Attributwerten der Symbole in w die Attributwerte für A berechnen. Der Kellerautomat muss nur zusammen mit jedem Symbol im Symbolkeller auch seine Attributwerte speichern.

- 6.65 Beispiel.** Ähnlich wie in Beispiel 6.50 betrachten wir wieder eine Grammatik für arithmetische Ausdrücke (die nun z. B. 2, 3 und 4 als Konstanten zulasse) und die Eingabe $(2+3)*4$. Abbildung 6.2 zeigt die Verarbeitung durch den Kellerautomaten. Im Unterschied zu Abbildung 6.1 ist im Keller *nicht* zu jedem Symbol der Kellerzustand angegeben, sondern (soweit sinnvoll) der Wert des Attributs *val*.

Eine nochmalige Betrachtung dieses Beispiels zeigt, dass die Berechnung der Attributwerte auch noch in einem etwas allgemeineren Fall im wesentlichen genauso funktioniert: nämlich dann, wenn die Attributwerte eines neu in den Keller gelegten Symbols auch noch von Attributwerten abhängen, die zu Symbolen gehören, die schon oben im Keller liegen. Mit anderen Worten kann man auch noch „manche“ vererbten Attribute behandeln:

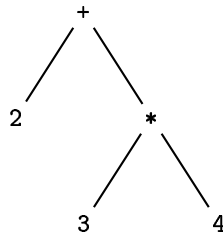
- 6.66** Eine Grammatik heißt *L-attribuiert*, wenn sie nur synthetisierte Attribute enthält und solche vererbten Attribute, für die gilt: In jeder Produktion $A \rightarrow X_1 \cdots X_m$ hängen die vererbten Attribute jedes X_j nur von den vererbten Attributen von A und den links von X_j stehenden Symbolen X_1, \dots, X_{j-1} ab.

Wir geben nun noch zwei Beispiele dafür, wozu synthetisierte Attribute benutzt werden können.

| Kellersymbole Attributwerte (<i>val</i>) | noch nicht gelesene Eingabe |
|---|-----------------------------|
| \neg | (2 + 3) * 4 \vdash |
| \neg (| 2 + 3) * 4 \vdash |
| \neg (2 2 | + 3) * 4 \vdash |
| \neg (F 2 | + 3) * 4 \vdash |
| \neg (T 2 | + 3) * 4 \vdash |
| \neg (E 2 | + 3) * 4 \vdash |
| \neg (E + 2 | 3) * 4 \vdash |
| \neg (E + 3 2 3 |) * 4 \vdash |
| \neg (E + F 2 3 |) * 4 \vdash |
| \neg (E + T 2 3 |) * 4 \vdash |
| \neg (E) 5 |) * 4 \vdash |
| \neg (E) 5 | * 4 \vdash |
| \neg F 5 | * 4 \vdash |
| \neg T 5 | * 4 \vdash |
| \neg T * 5 | 4 \vdash |
| \neg T * 4 5 4 | \vdash |
| \neg T * F 5 4 | \vdash |
| \neg T 20 | \vdash |
| \neg E 20 | \vdash |

Tabelle 6.2: Verarbeitung der Werte von Attribut *val* für die Eingabe (2+3)*4

6.67 Beispiel. Ableitungsbäume enthalten manchmal einige „überflüssige“ Knoten. Zum Beispiel könnte man die Information aus dem Ableitungsbaum in Abbildung 6.12 auch kompakter so darstellen:



Dies ist ein Beispiel eines so genannten (abstrakten) Syntaxbaumes. Analog kann man etwa bei Produktionen für Kontrollstrukturen (**if-then-else**, usw.) vorgehen.

Die Konstruktion eines Syntaxbaumes aus dem (u. U. nur implizit konstruierten) Ableitungsbaum kann man mit Hilfe eines synthetisierten Attributes *ast* (die englische Abkürzung für „abstract syntax tree“) durchführen. Attributwert für ein Nichtterminalsymbol *A* ist dabei ein Zeiger auf die Wurzel des Syntax(teil)baumes, der *A* entspricht.

In unserer Beispielgrammatik ist dann etwa für die Produktion $E_0 \rightarrow E_1 + T_1$ die zugehörige semantische Regel: $E_0.ast = makeASTNode('+', E_1.ast, T_1.ast)$. Dabei erzeuge $makeASTNode(c, t_1, t_2)$ einen neuen Knoten des „Typs“ $+$, dessen linker bzw. rechter Nachfolger durch die Zeiger t_1 bzw. t_2 gegeben sind.

6.68 Beispiel. Die Erzeugung von Code durch den Übersetzer kann (jedenfalls in Teilen) ebenfalls durch ein synthetisiertes Attribut *code* von Nichtterminalsymbolen *A* beschrieben werden, dessen Wert gerade der Code ist, der dem Programmausschnitt entspricht, für den *A* steht.

Will man zum Beispiel arithmetische Ausdrücke auswerten (die nun auch wieder Variablen enthalten dürfen und nicht schon vom Übersetzer ausgewertet werden können), dann kann man etwa für die Produktion $E_0 \rightarrow E_1 + T_1$ wie folgt vorgehen. Man benutzt ein zusätzliches synthetisiertes Attribut *temp*, das angibt, in welcher Hilfsvariablen —z. B. der Form H4711 — der arithmetische Wert letztendlich abgelegt ist.

Zunächst wird der Code für E_1 und T_1 übernommen. Dann wird der Name *v* einer neuen Hilfsvariable $E_0.temp = v$ erzeugt, indem z. B. die maximale bisher benutzte Nummer um 1 erhöht wird. Als neue Codezeile noch angefügt: $v = E_1.temp + T_1.temp$.

Dies ist natürlich noch eine sehr plumpe Methode der Codeerzeugung. Für die Demonstration des Prinzips ist sie aber ausreichend.

Literaturverzeichnis

Stephen C. Kleene. Representation of events in nerve nets and finite automata. In Claude E. Shannon and John McCarthy, editors, *Automata Studies*, chapter 1, pages 3–40. Princeton University Press, 1956.