1. Formale Grundlagen der Linguistik

Schein

Kursübersicht

- Mathematische Grundlagen
- Reguläre Mengen, Ausdrücke und Sprachen
- Endliche Automaten
- Kontextfreie Sprachen und ihre Grammatiken
- Kellerautomaten
- Kontextsensitive Sprachen und ihre Grammatiken
- Turingmaschinen
- Allgemeine Regelsprachen und die Chomsky-Hierarchie

Technisches

Schein: Fleißig kommen und die Aufgaben im Tutorium rechnen (in Zahlen: mindestens 50% der Punkte aus mindestens 70% der Blätter), Abschlusskolloquium oder Klausur.

Literatur

- Klabunde, R.: Formale Grundlagen der Linguistik, Narr 1998
- Vossen, G & Witt, K.-U.: Grundlagen der Theoretischen Informatik mit Anwendungen, Vieweg 2001
- Schöning, U.: Theoretische Informatik kurzgefasst, Spektrum 2001
- Partee, B. et al: Mathematical Methods in Linguistics, Kluwer 1990
- Hopcroft, J.E., & Ullmann, J.D.: Introduction to automata theory, languages, and computation, Addison Wesley 1979
- Hofstadter, D.: Gödel, Escher, Bach, passim

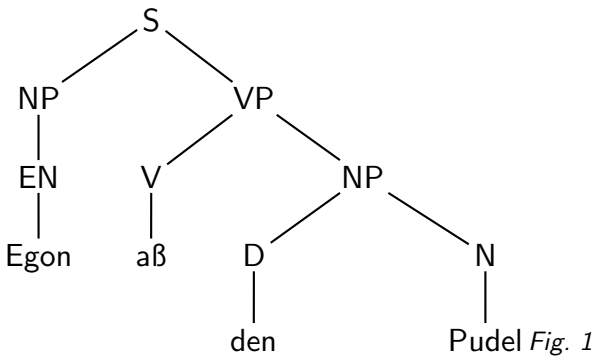


Fig. 1

2. Wohin geht die Reise?

Formale Sprachen sind Mengen von Wörtern (in linguistischen Anwendungen können das „Wörter“, Sätzen, Phrasen oder was immer sein, in Anwendungen aus der Informatik vielleicht eine ganze Datei), die ihrerseits aus Symbolen (diese können je nach Anwendung Buchstaben, „Wörter“ einer Natursprache oder allgemeiner „Tokens“, vorklassifizierte Bruchstücke einer Eingabe aus einer vorgeschalteten lexikalischen Analyse) aufgebaut sind. Was in der Menge ist, ist ein „grammatisch korrektes“ Wort, alles andere ist falsch.

In dieser Veranstaltung werden wir die Symbole meist klein halten: a, b, oder y (die Methoden funktionieren aber genauso mit Symbolen wie „der“, „Mann“, „Xylophon“, „Magnetostriktionskoeffizient“ oder auch $+$, $-$ und $\times$), dementsprechend sehen die Wörter meist albern aus: $abyby$.

Chomsky hat in den 50er und 60er Jahren angefangen, Natursprachen als formale Sprachen zu behandeln. Es ist zu bezweifeln, dass das ganze Geheimnis ist, *formale Syntaxanalyse* aber wird bei der Verarbeitung natürlicher Sprache immer irgendwo auftauchen. Kunstsprachen (Programmier- oder Konfigurationssprachen, Protokolle) basieren fast vollständig auf Theorien formaler Sprachen.

Wir wollen einem Rechner erklären, wie er eine Sprache „verstehen“ kann, und Abschätzungen gewinnen, wie schwierig das sein wird. Dabei werden wir nach „guten“ Sprachen suchen, in denen eine Entscheidung über die „Richtigkeit“ („Grammatikalität“) eines Wortes „leicht“ gefällt werden kann.

Näher an die Natursprache rücken: Seien Zeichen in etwa das, was im Duden als Wörter steht (plus Flexionen), Wörter in etwa das, was zwischen zwei Satzendepunkten steht. Dann kann man Wörter beispielsweise in ihre Phrasenstruktur zerlegen und als Baum darstellen:

(cf. Fig. 1)

Eine Grammatik soll aus Klauseln wie

$$S \to NP\ VP$$
$$N \to \text{Pudel}$$

bestehen. Das soll bedeuten, dass ich in einem Wort, in dem irgendwo das Symbol *S* steht, ich dieses durch die beiden Symbole *NP* und *VP* ersetzen kann – nochmal im Klartext: das Wort *NP VP* besteht aus zwei (und nicht vier) Symbolen.

Bevor wir uns aber wirklich mit Sprachen, Regeln, Bäumen und ähnlichem beschäftigen können, müssen wir zunächst einige mathematische Grundlagen schaffen. Das Ziel dabei ist, dass wir „zwingend“ argumentieren können wollen, d.h. unsere Begriffe sollen unzweideutig sein, und wenn wir etwas beweisen, muss dieser Beweis von allen, die die Grundlagen der Mathematik anerkennen (für uns heißen diese Grundlagen übrigens „Axiomensystem von Zermelo und Fränkl“ – aber so weit „runter“ wollen wir in diesem Rahmen nicht gehen) überprüfbar sein.

Für dieses Ziel gibt es zwei Gründe. Einerseits ist es eine Frage der Erkenntnistheorie – wir wollen vermeiden, dass sich Leute etwa endlos streiten, was nun ein Wort sei (und so ein Streit

alpha	α		A	nu	ν		N
beta	β		B	xi	ξ		Ξ
gamma	γ		Γ	omikron	o		O
delta	δ		Δ	pi	π		Π
epsilon	ϵ	ε	E	rho	ρ	ϱ	P
zeta	ζ		Z	sigma	σ	ς	Σ
eta	η		H	tau	τ		T
theta	θ	ϑ	Θ	upsilon	υ		Υ
iota	ι		I	phi	ϕ	φ	Φ
kappa	κ		K	chi	χ		X
lambda	λ		Λ	psi	ψ		Ψ
mu	μ		M	omega	ω		Ω

Fig. 2

ist im Rahmen natürlicher Sprachen erstaunlich schwer zu entscheiden). Andererseits soll unser Vorgehen so zwingend sein, dass sich ihm nicht mal ein Computer entziehen kann (und nicht mal ein Computer der auf dem Mars gebaut wurde). Das bedeutet, dass die Erkenntnisse, die wir in diesem Rahmen gewinnen, mehr oder minder direkt in Computerprogramme übersetzbar sind. Sowas hilft ungemein, wenn man Computerlinguistik treiben möchte...

3. Wilde Zeichen

In der Mathematik gibt es viel zu bezeichnen, und weil das, was bezeichnet wird, noch viel manipuliert werden soll, sollten die Zeichen kurz sein. Deshalb braucht man viele Zeichen. Im Laufe der Geschichte der Mathematik haben sich etliche Konventionen herausgebildet, mit denen man vertraut sein sollte.

Insbesondere solltet ihr die Namen und Sprechweisen kennen. Wenn ihr irgendwelche mathematischen Ausdrücke hinschreibt, prüft dann und wann, ob ihr sie auch „aussprechen" könnt – wenn nicht, seht hier nach, wie man das spricht oder fragt in der Vorlesung oder im Tutorium. Am Ende müsst ihr etwa in Prüfungen oder gegenüber Mitstudis und KollegInnen, ja doch immer wieder über Mathematik *reden*.

(cf. Fig. 2)

Fast alle Zeichen in der Mathematik sind mehrfach überladen, bedeuten also je nach Kontext verschiedene Dinge. Es gibt also keine Tabelle, die sagt „Σ ist ein Alphabet, während ν eine Frequenz ist."

Sprechweisen: f': „f Strich" (engl. f prime); $\bar{a}$: „a quer" (engl. a bar); $\hat{a}$: „a Dach" (engl. a hat); a_i: ahh ihh.

Ein paar Zeichen haben kanonische Bedeutungen. Hier eine Auswahl, ohne Gewähr, dass sie diese im konkreten Fall auch haben:

- ∞ – unendlich (schwieriger Begriff)
- i – imaginäre Einheit. Leider ist i auch der populärste Index, also eine Variable, die einfach nacheinander verschiedene Werte annimmt (es geht dann meist weiter mit j, k usf.) Hier wird also eifrig überladen. Da wir hier nicht mit komplexen Zahlen operieren, bedeutet i hier auch fast nie die imaginäre Einheit, und wenn, ist das Zeichen nicht kursiv, was es als Index immer ist.
- $\emptyset$ oder $\{\}$ – die leere Menge (engl. empty set)
- $\sum$ – Summe
- $\prod$ – Produkt
- $\in$ – Mitgliedschaft von Mengen (Sprechweise: $a \in B$ – „a Element B" oder „a ist in B", engl. „a is an element of B" oder „a in B")

- $\cup$ – Vereinigung von Mengen (Sprechweise: $A \cup B$ – „A vereinigt B“, engl. „A union B“)
- $\cap$ – Schnitt von Mengen (Sprechweise: $A \cap B$ – „A geschnitten B“, engl. „A intersection B“)
- $\setminus$ – Differenz von Mengen
- $\times$ – kartesisches Produkt von Mengen (Sprechweise: A kreuz B)
- $\circ$ – „Verkettung von Funktionen“ oder „generischer Operator“
- $\approx$ – Ungefähr gleich
- $\sim$ – Ungefähr gleich oder Proportional
- $\gg$, $\ll$ – Wesentlich größer, kleiner
- $\to$ – „Geht über auf“ (engl. „maps to“)

Aussagen

Aussageform

Problems

(3.1) Schreibt die Kleinbuchstaben des griechischen Alphabets drei Mal auf und legt euch die Zettel ins Federmäppchen (oder wohin auch immer), damit ihr in langweiligen Vorlesungen griechische Zeichen üben könnt.

Wenn ihr Probleme habt, einzelne Zeichen elegant zu malen, fragt in der Vorlesung nach den kanonischen Schwüngen.

(3.2) Lest die folgenden mathematischen Ausdrücke laut vor:

1. α'
2. $\hat{\alpha} \in \Gamma$
3. $A \cup B = \emptyset$
4. $\omega_i'' \gg \bar{\vartheta}$

(L)

4. Logische Aussagen

Aussagen

Aussagen sind Dinge, die wahr oder falsch sein können: „a ist im Wort Markus enthalten“ (W), „Alle Vögel fliegen hoch“ (F).

$2 + x = 7$ ist keine Aussage, solange x nicht belegt ist: *Aussageform*.

Junktoren

Junktoren sind Operatoren auf Aussagen: Sie bauen aus Aussagen neue Aussagen:

- $\neg$: Negation
- $\vee$: Disjunktion, „oder“
- $\wedge$: Konjunktion, „und“
- $\rightarrow$: Implikation, „folgt aus“
- $\leftrightarrow$: Äquivalenz „dann und nur dann“

Beispiele: Wenn $\neg A$ wahr ist, ist A falsch
Wenn A_1 wahr und A_2 falsch ist, ist $A_1 \vee A_2$ wahr und $A_1 \wedge A_2$ falsch.

Die Wirkungsweise der logischen Operatoren wird gerne durch Wahrheitstafeln dargestellt, hier für Disjunktion, Implikation und Äquivalenz.

A_1	A_2	$\vee$	$\rightarrow$	$\leftrightarrow$
W	W	W	W	W
W	F	W	F	F
F	W	W	W	F
F	F	F	W	W

Es mag zunächst etwas verwirren, dass etwas wie „folgt aus“ hier den intuitiv verstehbaren logischen Operatoren „und“ und „oder“ gleichgestellt sind. Das Problem dahinter ist, dass man unterscheiden sollte zwischen dem Erzeugen neuer Aussagen und den Aussagen selbst. Das „folgt aus“, das ihr in der Schule wahrscheinlich mit $\Rightarrow$ und (bzw. $\Leftrightarrow$ für „ist äquivalent mit“) bezeichnet habt, gehört zur ersten Kategorie – es macht eine Aussage über die Relation zwischen (anderen) Aussagen, gehört also in gewisser Weise zu einer Metasprache. Da die Trennung dieser Ebenen im harten Beweisleben meist nicht dringend nötig ist, wird sie auch oft nicht vorgenommen, und auch wir werden nach dieser Seite für unsere Schlüsse einfach $\Rightarrow$ schreiben.

Das $\leftarrow$, das hier gemeint ist, sollte man sich vorstellen wie in Aussagen der Form „Wer mehr als 50% der Punkte erreicht, wird zur Scheinprüfung zugelassen“ oder „Wenn der Computer läuft, wird er kalt“. Beide haben die Form $A \rightarrow B$ (macht euch klar, was jeweils A und B sind!), aber die eine ist für diese Veranstaltung wahr, die andere für unsere Computer falsch (das mit dem „für XY“ ist übrigens etwas, mit dem die Logik auch eifrig operiert, das Stichwort hier wären Belegungen).

Komisch an der Tabelle ist vor allem, dass $F \rightarrow X$ wahr, egal ob X selbst wahr oder falsch ist. In der Tat ist das nicht wirklich trivial (das Stichwort dazu ist „ex falso quodlibet“), und eine stichhaltige Begründung (sofern es so etwas an den Grundlagen geben kann) baut letztlich auf Kalküle auf, mit denen man Aussagen generiert. Es ist aber vielleicht einsichtig, dass etwas wie „Wenn die Sonne im Westen aufgeht, ist es 12 Uhr mittags“ eigentlich nicht falsch sein kann. Die Sonne geht eben nie im Westen auf, und deshalb ist es egal, ob es gerade 12 Uhr mittags ist, wenn sie das tut. In dem Sinn kann die Aussage jedenfalls nicht falsch sein. Wenn die Aussage aber nicht falsch ist, muss sie in der klassischen Logik (in der es nichts außer wahr und falsch gibt) aber wahr sein (ok, das war jetzt ein Taschenspielertrick – wie gesagt, letzlich kann man das akzeptieren oder nicht und kommt entsprechend auf verschiedene Logiken, und wer klassische Logik machen will, sollte es akzeptieren).

Die Äquivalenz („dann und nur dann“) wird auch gerne mit „gdw“ („genau dann wenn“) oder „iff“ („if and only if“) abgekürzt.

Quantoren

Quantoren

Häufig reichen einfache Aussagen nicht aus – man will Aussageformen mit Variablen belegen können, und dann soll wieder etwas herauskommen, was wahr oder falsch ist. Dazu gibt es *Quantoren*:

- $\forall$: Allquantor, Generalisator, „für alle“
- $\exists$: Partikularisator, „es gibt ein“

Beispiel:
$\forall\, a \in \mathbb{N}\ \exists\, b \in \mathbb{Z} : b = a - 1$ heißt: Für alle a aus der Menge $\mathbb{N}$ gibt es ein b in der Menge $\mathbb{Z}$, so dass die Gleichung $b = a - 1$ gilt

Die Quantoren kommen aus einer Sprache, die sich Prädikatenlogik nennt und zusätzlich noch genau sagt, was eigentlich alles hinter den Quantoren kommen kann. Für unsere Zwecke sind diese Details nicht sonderlich wichtig, ich möchte nur darauf hinweisen, dass richtige Logiker mit Schreibweisen wie „$\forall\, x \in \mathbb{N}$“ nicht unbedingt immer glücklich sind, weil man damit leicht den Rahmen dessen, was in dem *Formalismus* der Prädikatenlogik erster Stufe erlaubt ist, verlassen kann. Nicht erlaubt wäre beispielsweise $\forall\, A \in \{A \mid A$ ist für natürliche Zahlen wahr$\}$. Warum das so ist, soll uns hier nicht kümmern, mein Punkt ist, dass sind diese Zeichen *eigentlich* mehr als die Abkürzungen, als die wir sie hier verwenden.

Das ist in der Mathematik generell so: Die lustigen Zeichen, die hier verwendet werden, dienen nicht nur dazu, Leute zu beeindrucken, sie vermitteln auch eine präzise und tragfähige Semantik, die im Prinzip jedem/r ermöglicht, die Argumentationsweisen unzweideutig nachzuvollziehen und eventuelle Widersprüche aufzudecken. Ein leicht nachvollziehbares Beispiel dazu findet sich in den Aufgaben: Der Satz „Es regnet oder ich fahre Rad“ hat in Natursprache mindestens zwei mögliche Interpretationen („Inklusiv-oder“ oder „Exklusiv-oder“, wobei ersteres erlaubt, dass beides wahr ist, letzteres aber dem Entweder-oder entspricht) – die formalisierte Aussage hat genau eine, nämlich die Inklusiv-Variante.

Mehr zu all diesen Themen vermittelt die Vorlesung über Logik.

Problems

(4.1) Malt die Wahrheitstafeln für $\dot{\vee}$ (Exklusiv-Oder, also Entweder-Oder), $\wedge$ und $\neg$. **(L)**

(4.2) (a) Sei R die Aussage „es regnet“, F die Aussage „ich fahre Rad“ und N „ich werde nass“. Schreibt die folgenden Aussagen als formale Ausdrücke:

1. Es regnet nicht
2. Es regnet und ich werde nass
3. Es regnet oder ich fahre Rad
4. Wenn es regnet, fahre ich nicht Rad
5. Ich fahre genau dann Rad, wenn es nicht regnet
6. Wenn es nicht regnet, werde ich nicht nass

(b) Bewertet den Wahrheitsgehalt der folgenden Aussagen zum augenblicklichen Zeitpunkt:

1. $R \wedge F \to N$
2. $R \wedge N \to N$
3. $R \vee N \to N$

(L)

(4.3) Berechnet Wahrheitstafeln für die Aussagen $A \vee (B \wedge C)$ und $(A \vee B) \wedge (A \vee C)$. Da hier drei Wahrheitswerte eine Rolle spielen, haben diese Wahrheitstafeln $2^3 = 8$ Zeilen, eben alle Möglichkeiten, A, B und C zu belegen. **(L)**

5. Mengen

Selbstanwendungs-problem

set comprehension

Basis

Regeln

rekursiv

Naiver Mengenbegriff

Georg Cantor (1895): Eine Menge M ist eine Zusammenfassung von bestimmten wohlunterschiedenen Objekten unserer Anschauung oder unseres Denkens, welche Elemente der Menge M genannt werden, zu einem Ganzen.

Wichtige Folge dieser Definition: Mengen sind nicht geordnet ($\{a, b\} = \{b, a\}$), ein Element kann nicht „mehrmals“ Element einer Menge sein.

Die Definition ist weniger trivial, als sie klingen mag. Sie impliziert inbesondere bereits das unten erwähnte Extensionalitätsprinzip. Sie bedeutet aber wohl oder übel auch, dass Mengen als Elemente von Mengen in Frage kommen (was etwas ganz anderes ist als eine Teilmenge). Das ist zwar sehr nützlich (genau genommen entscheidend), kann aber, wenn man unvorsichtig ist, Probleme machen:

Russell'sche Antinomie

Sei M die Menge aller Mengen, die sich nicht selbst enthalten. Wenn $M \in M$, dann folgt nach Definition, dass $M \notin M$. Wenn $M \notin M$, folgt nach Definition, dass $M \in M$.

Also: Naiver Mengenbegriff führt zu Widersprüchen, offenbar sind zusätzliche Forderungen nötig: *Axiomatische Mengenlehre*, ZFC.

Für unsere Zwecke reicht der naive Mengenbegriff allerdings, wo er Probleme bereiten würde, weise ggf. darauf hin. Die Axiomatische Mengenlehre ist nämlich, richtig betrieben, Stoff für mehr als eine eigene Vorlesung.

Übrigens ist die Russell'sche Antinomie eng verwandt mit Aussagen wie „Alle Hochschullehrer lügen“, wenn ich ihn sage: Es nutzt ein *Selbstanwendungsproblem* aus, ein Formalismus (im einen Fall die Mengenlehre, im anderen die Sprache) wird auf sich selbst angewandt. Ein solches Selbstanwendungsproblem wird uns bei der Untersuchung von Turingmaschinen und ihren Beschränkungen noch einmal begegnen.

Mengen notieren wir entweder durch Aufzählung ihrer Elemente,

$$M = \{\text{rot}, \text{grün}, \text{gelb}\},$$

oder durch Definition der Elementeigenschaften,

$$M = \{x \in \mathbb{N} \mid x \text{ ist ohne Rest durch 2 teilbar}\}$$

(„alle x aus $\mathbb{N}$ mit der Eigenschaft“ – *set comprehension*)

oder durch *Basis* und *Regeln rekursiv*, z.B.:

1. $a \in A$
2. $\Phi \in A \Rightarrow \Phi a \in A$
3. Elemente von A sind nur Zeichenketten, die durch die obigen Regeln erzeugt wurden (diese Ausschlussbedingung werden wir künfig implizit annehmen)

erzeugt $A = \{a, aa, aaa, aaaa, \ldots\}$.

Problems

Teilmenge

echte Teilmenge

(5.1) Findet (etwa auf der unserer Webseite) alle Räume heraus, in denen Veranstaltungen des Lehrstuhls stattfinden und schreibt diese als Menge. **(L)**

(5.2) Welche Menge wird durch folgende Vorschrift erzeugt?

1. $x \in A$
2. $\Phi \in A \Rightarrow a \Phi b \in A$

(L)

(5.3) Schreibt die ersten paar Elemente der Menge

$$\{x/2 \,|\, x \text{ ist nichtnegative gerade Zahl}\}$$

hin. **(L)**

6. Mengen II

Teilmengen

B ist *Teilmenge* von A ($B \subseteq A$), wenn

$$x \in B \Rightarrow x \in A$$

gilt. Also: Jedes Element von B ist in A enthalten.B ist *echte Teilmenge* von (oder *echt enthalten* in) A ($B \subset A$), wenn

$$(B \subseteq A) \wedge (\exists x \in A : x \notin B).$$

Das heißt: B ist Teilmenge von A, und dazu existiert mindestens ein $x \in A$, das nicht in B ist.

Vorsicht: $B \in A$ ist etwas ganz anderes als $B \subset A$.

Extensionalitätsprinzip

Zwei Mengen sind gleich, wenn sie die gleichen Elemente haben: $A = B \Leftrightarrow B \subseteq A \wedge A \subseteq B$.

Leere Menge

Eine Menge ohne Element ist wegen Extensionalität immer gleich der Leeren Menge

$$\emptyset = \{x | x \neq x\}.$$

Es gilt: $\emptyset \subseteq M$ für alle Mengen M.

Der Beweis dazu ist eine nette Illustration von Schlussweisen in der Mathematik:

Die zu zeigende Aussage lässt sich als $x \in \emptyset \Rightarrow x \in M$ schreiben – hier wurde einfach die Definition der Teilmenge verwendet.

Dies ist äquivalent zu $x \neq x \Rightarrow x \in M$. Ist diese Aussage wahr oder falsch? Nun, $x \neq x$ ist immer falsch, d.h. auf der linken Seite der Aussage steht F. Eine Inspektion der Wahrheitstafel von $\Rightarrow$ liefert, dass damit unabhängig von vom Wahrheitswert der rechten Seite die Gesamtaussage immer wahr ist. Da alle unsere Aussagen äquivalent sind, ist auch die zu beweisende Ursprungssaussage wahr.

Auf ähnliche Weise können viele Eigenschaften der leeren Menge nachgewiesen werden – offenbar spielt die leere Menge in der Mengenlehre eine ähnlich exotische Rolle wie die Null in der Arithmetik.

Operationen auf Mengen

Vereinigung:

$$A \cup B = \{x \mid x \in A \vee x \in B\}.$$

Schnitt:

$$A \cap B = \{x \mid x \in A \wedge x \in B\}.$$

Differenz:

$$A \setminus B = \{x \mid x \in A \wedge x \notin B\}.$$

Komplementbildung (relativ zu Obermenge O):

$$\overline{A} = O \setminus A = \{x \mid x \in O \wedge x \notin A\}.$$

Bei der Komplementbildung ist entscheidend, dass wir eine Obermenge haben. Hätten wir sie nicht, wäre im Komplement „alles“, was nicht in der Menge ist, und „alles“ ist ein Begriff, der im Rahmen der naiven Mengenlehre nicht vorkommen darf.

Rechenregeln

Hier steht $\circ$ für $\cup$ oder $\cap$

- Kommutativität: $M \circ N = N \circ M$
- Assoziativität: $K \circ (M \circ N) = (K \circ M) \circ N$
- Idempotenz: $M \circ M = M$
- Distributivität: $K \cup (M \cap N) = (K \cup M) \cap (K \cup N)$ (und mit $\cap$ und $\cup$ vertauscht)

Problems

(6.1) Seien $A = \{1, 2, 3, a, b\}$, $B = \{4, a\}$ und $C = A \cup B$.

Rechnet C, $A \cap B$, $\mathcal{P}(B)$, $A \setminus B$, $B \setminus A$ aus. **(L)**

(6.2) Gilt $\emptyset \subseteq \emptyset$, ist also die leere Menge ihre eigene Teilmenge? **(L)**

(6.3) Wie ist das mit $\emptyset \subset \emptyset$, ist also die leere Menge auch *echte* Teilmenge ihrer selbst? Tipp: Versucht, einen Beweis nach dem Muster des $\emptyset \subseteq M$-Beweises oben zusammenzuzimmern. **(L)**

(6.4) Beweist mit Definitionen für Schnitt und Vereinigung sowie dem in einer Aufgabe zum Kapitel mit den logischen Aussagen bewiesenen „Distributivgesetz“ für Junktoren, dass $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$. **(L)**

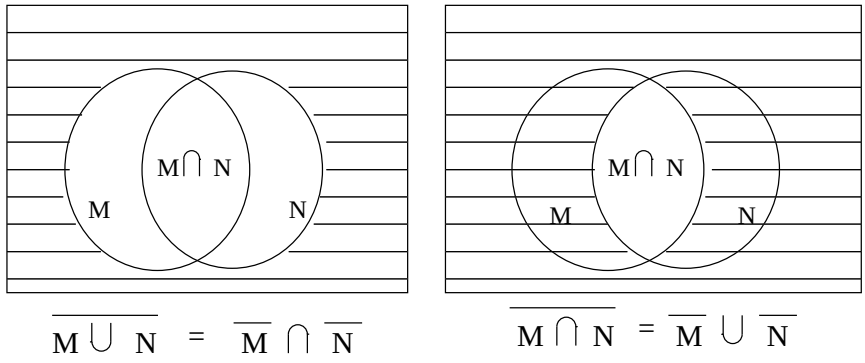


Fig. 3

7. Mengen III

Es gelten weiter:

Konsistenzprinzip:

$$M \cup N = N \Leftrightarrow N \supseteq M$$
$$M \cap N = M \Leftrightarrow M \subseteq N$$

Adjunktivität: $M \cap (M \cup N) = M$ und mit $\cap$ und $\cup$ vertauscht.

DeMorgan:

$$\overline{M \cup N} = \overline{M} \cap \overline{N}$$
$$\overline{M \cap N} = \overline{M} \cup \overline{N}$$

Einsehen über *Venn-Diagramme*

(cf. Fig. 3)

Man kann diese Sachen auch streng beweisen. Ein halbwegs strenges Beispiel: $\overline{M \cup N} = \overline{M} \cap \overline{N}$.

Zu zeigen „$\supseteq$“, also nach Definition: „Jedes Element von $\overline{M} \cap \overline{N}$ ist auch Element von $\overline{M \cup N}$“.
Sei also $x \in \overline{M} \cap \overline{N}$
Dann ist nach der Definition der Schnittmenge $x \in \overline{M}$ und gleichzeitig $x \in \overline{N}$,
also nach Definition des Komplements $x \notin M$ und gleichzeitig $x \notin N$,
also ist mit einfacher Logik *nicht* $x \in M$ oder $x \in N$,
also nach Definition der Vereinigung $x \notin M \cup N$,
also nach Definition des Komplements $x \in \overline{M \cup N}$.

Zu zeigen „$\subseteq$“, also nach Definition: „Jedes Element von $\overline{M \cup N}$ ist auch Element von $\overline{M} \cap \overline{M}$.
Sei also $x \in \overline{M \cup N}$.
Dann ist nach Definition des Komplements $x \notin M \cup N$,
also nach Definition der Schnittmenge $x \notin M$ und zugleich $x \notin N$,
also nach Definition des Komplements $x \in \overline{M}$ und zugleich $x \in \overline{M}$,
also nach Definition der Schnittmenge $x \in \overline{M} \cap \overline{N}$, q.e.d.

Potenzmenge

Die *Potenzmenge* $\mathcal{P}(A)$ ist die Menge aller Teilmengen von A. Es gilt $|\mathcal{P}(A)| = 2^{|A|}$, wobei $|A|$ für die Zahl der Elemente in A steht. Beispiel: $\mathcal{P}(\{a, b\}) = \{\emptyset, \{a\}, \{b\}, \{a, b\}\}$.

Wir haben die Zahl der Elemente einer Menge A als $|A|$ recht nonchalant eingeführt. Wie so oft in der Mathematik ist das nicht ohne Fallen, denn niemand hat gesagt, dass man die Elemente einer Menge auch wirklich zählen kann – was wäre z.B. $|\mathbb{N}|$? Wir kehren unter dem Stichwort Kardinalität später nochmal zu dieser Frage zurück.

Kartesisches Produkt

Kartesisches Produkt
Tupel

$M \times N = \{\langle m, n\rangle \mid m \in M, n \in N\}$. heißt *Kartesisches Produkt* von M und N, dabei ist $\langle m, n\rangle$ ein *Tupel*.

Tupel sind anders als Mengen geordnet (d.h., $\langle m, n\rangle \neq \langle n, m\rangle$) und können einzelne Elemente öfter als ein Mal enthalten. Tupel mit zwei Elementen heißen auch Paare, solche mit drei und vier Tripel und Quadrupel, allgemein n-Tupel.

Der Begriff „kartesisch“ bezieht sich auf René Descartes, der erkannte, dass mit einem Kartesischen Produkt der reellen Zahlen Punkte in der Ebene beschrieben werden können – zweidimensionale Koordinaten eben.

Beispiel:

$$\{a, b, c\} \times \{0, 1\} = \\ \{\langle a, 0\rangle, \langle a, 1\rangle, \langle b, 0\rangle, \langle b, 1\rangle, \langle c, 0\rangle, \langle c, 1\rangle\}$$

Problems

(7.1) Seien wieder $A = \{1, 2, 3, a, b\}$, $B = \{4, a\}$ und $C = A \cup B$.

Rechnet $A \times B$, $C \setminus (A \cap B)$ und $(C \setminus A) \cup (C \setminus B)$ aus. **(L)**

(7.2) Zeigt dem Beweis für $\overline{M \cup N} = \overline{M} \cap \overline{N}$ folgend die andere DeMorgan'sche Formel.

8. Iteration in Zeichen

Häufig will man in der Mathematik Operationen auf eine Menge von Objekten durchführen – sie schneiden, vereinigen, addieren, was immer. Um so etwas elegant schreiben zu können, haben sich einige Schreibweisen entwickelt. Am häufigsten begegnet einem dabei die wiederholte Addition (Summe), während im Rahmen dieser Veranstaltung vor allem Mengenoperationen iteriert werden.

Seien n Größen $a_1, a_2, \ldots a_n$ gegeben. Wenn wir ihre Summe notieren wollten, könnten wir

$$a_1 + a_2 + \cdots + a_n$$

schreiben. Das ist aber lang und potenziell missverständlich. Mathematiker schreiben deshalb lieber:

$$\sum_{i=1}^{n} a_i$$

Das funktioniert genau wie das gewohnte Plus. Man kann also ausklammern:

$$\sum_{i=1}^{n} b\, a_i = b \sum_{i=1}^{n} a_i$$

oder Summen umorganisieren:

$$\sum_{i=1}^{n} (a_i + b_i) = \sum_{i=1}^{n} a_i + \sum_{i=1}^{n} b_i.$$

Dabei ist natürlich wichtig, zu prüfen, was jeweils von der „Schleifenvariablen“ i abhängt. In der ersten Regel ist b eine Konstante, in der zweiten sind die b_i für verschiedene i natürlich in der Regel verschieden. Es hilft am Anfang, die Summen auszuschreiben (zur Not für festes i) und mit Schulwissen nachzusehen, was geht und was nicht. So ist zum Beispiel

$$\sum_{i=1}^{n} b a_i = b a_1 + b a_2 + \cdots + b a_n = b(a_1 + a_2 + \cdots + a_n),$$

und wir erkennen im zweiten Faktor des Ergebnisses unsere Summe wieder. Analog ist

Symmetrieargumente

$$\sum_{i=1}^{n}(a_i+b_i) = (a_1+b_1)+(a_2+b_2)+\cdots+(a_n+b_n)$$
$$= (a_1+a_2+\cdots+a_n)+(b_1+b_2+\cdots+b_n),$$

und wieder erkennen wir die Summen.

Die Zählvariable i kann natürlich auch anders heißen und muss nicht in Indizes verwendet werden, und die Grenzen müssen nicht 1 und n sein:

$$\sum_{k=-3}^{3} k = (-3)+(-2)+(-1)+0+1+2+3 = 0$$

Im letzten Beispiel hätte man unabhängig von den konkreten Grenzen schon vorhersagen können, dass die Summe Null ist, weil zu jeder negativen Zahl eine positive Zahl gleichen Betrags addiert wird. Argumente dieser Art heißen *Symmetrieargumente* und helfen sehr oft, die fraglichen Ausdrücke entscheidend zu vereinfachen.

Es gilt:

$$\sum_{i=1}^{n} a_i = a_1 + \sum_{i=2}^{n} a_i.$$

und (jedenfalls, wenn $1 < m < n-1$):

$$\sum_{i=1}^{n} a_i = \sum_{i=1}^{m} a_i + \sum_{i=m+1}^{n} a_i.$$

Übrigens ist es mitnichten nötig, dass der Summenindex im Ausdruck in der Summe vorkommt. Es ist etwa $\sum_{i=1}^{n} 1 = n\cdot 1$. Wer das nicht glauben möchte, kann sich vorstellen, dass in Wirklichkeit $\sum_{i=1}^{n} 1i^0$ dasteht.

Analog zum Summenzeichen definieren Mathematiker gerne auch andere Mehrfachoperationen, etwa

$$\prod_{k=1}^{n} a_k = a_1 \cdot a_2 \cdots a_n$$

(das Zeichen kann als großes Pi gelesen werden und steht für Produkt in der gleichen Weise, in der das große Sigma für Summe steht),

$$\bigcup_{k=1}^{n} A_k = A_1 \cup A_2 \cup \cdots \cup A_n,$$

$$\bigcap_{k=1}^{n} A_k = A_1 \cap A_2 \cap \cdots \cap A_n$$

oder

$$\bigotimes_{k=1}^{n} A_k = A_1 \times A_2 \times \cdots \times A_n.$$

Für diese gelten natürlich andere Rechenregeln, so ist etwa

$$\prod_{k=1}^{n} a_k^x = \left(\prod_{k=1}^{n} a_k\right)^x$$

Häufig braucht man eine etwas verallgemeinerte Notation, wenn die Objekte nicht einfach durchgezählt werden. Will man beispielsweise alle Elemente von M vereinigen, die x enthalten, kann man

$$\bigcup_{N \in M \wedge x \in N} N$$

schreiben.

Relation
Definitionsbereich
Argumentbereich
Wertebereich

Problems

(8.1) Rechnet $\sum_{i=1}^{n} i$ für $n = 2$ bis $n = 7$ aus. Überlegt euch vorher, wie ihr das arbeitssparend hinbekommt (die Formel aus der nächsten Folie verwenden ist natürlich Betrug und in diesem Fall nicht mal arbeitssparend).

(8.2) Was ist $\prod_{i=1}^{n}(-1)$ in Abhängigkeit von n? Was ist $\sum_{i=1}^{n}(-1)^i$? **(L)**

(8.3) Wandelt $\sum_{i=1}^{n}(a_i - a_{i-1})$ in eine möglichst einfache Form um. Macht das dann auch noch mit $\prod_{i=1}^{n}(a_i / a_{i-1})$. **(L)**

(8.4) Sei $M_n = \{1, 2, 3, \ldots, n\}$, also $M_1 = \{1\}$, $M_2 = \{1, 2\}$ und so fort.

Schreibt $\bigcap_{i=1}^{3} M_i$ und $\bigcup_{i=1}^{3} M_i$ aus und berechnet den Wert dieser Ausdrücke. **(L)**

9. Relationen I

Eine (n-stellige) *Relation* R zwischen Mengen $M_1, \ldots, M_n$ ist eine Teilmenge des Kartesischen Produkts $M_1 \times \cdots \times M_n$.

Wir spezialisieren sofort auf $n = 2$ (zweistellige Relationen) und schreiben $R(a, b)$ oder aRb wenn $\langle a, b \rangle \in R$.

Beispiel: Sei T die Menge aller Tierarten, F die Menge aller Fortbewegungsarten (schwimmen, fliegen, tauchen, ...). Dann ist

$$\begin{aligned} T \times F \supset R = \{ &\langle \text{Schwan, fliegt} \rangle , \\ &\langle \text{Schwan, schwimmt} \rangle , \langle \text{Schwan, läuft} \rangle , \\ &\langle \text{Karpfen, taucht} \rangle , \langle \text{Fliege, fliegt} \rangle , \ldots \} \end{aligned}$$

eine mögliche Relation zwischen T (dem *Definitionsbereich*, auch *Argumentbereich* genannt) von R und F (dem *Wertebereich*).

Im Beispiel hat „Schwan“ mehrere *Bilder* und „fliegt“ mehrere *Urbilder*. In diesem Sinn ist die Relation eine Verallgemeinerung des Funktionsbegriffs.

Beispiel: $M = \{1, 2, 3\}$, $< \subset M \times M$. Relation „ist kleiner als“ ist $< = \{\langle 1, 2 \rangle , \langle 1, 3 \rangle , \langle 2, 3 \rangle\}$. Wir schreiben $1 < 2$, $1 < 3$ usf.

Vokabeln

reflexiv
symmetrisch
transitiv
irreflexiv
asymmetrisch
antisymmetrisch
intransitiv
antitransitiv

Sei $R \subseteq M \times M$.

Wir reden jetzt also von Relationen zwischen Elementen ein und derselben Menge.

R heißt

- *reflexiv*: $\forall\, x \in M : \langle x, x\rangle \in R$.
- *symmetrisch*: $\langle x, y\rangle \in R \Rightarrow \langle y, x\rangle \in R$.
- *transitiv*: $\langle x, y\rangle \in R \wedge \langle y, z\rangle \in R \Rightarrow \langle x, z\rangle \in R$.

Beispielsweise ist die Relation „$=$“ reflexiv, symmetrisch und transitiv, die Relation „$<$“ transitiv, die Relation „$\leq$“ reflexiv und transitiv.

Relationen können eine dieser Eigenschaften einfach nicht haben, es kann aber auch sein, dass es für alle Elemente irgendwelche Negationen der Bedingungen gegeben sind. Die Negation kann auf verschiedene Arten geschehen:

- *irreflexiv*: $\forall\, x \in M : \langle x, x\rangle \notin R$ – beachtet, dass das etwas ganz anderes ist, als nicht reflexiv. Ist etwa $R \subseteq \mathbb{N} \times \mathbb{N}$ einfach nur nicht reflexiv, kann es durchaus sein, dass zwar $\langle 1, 1\rangle \in R$, aber nicht $\langle 2, 2\rangle \in R$.
- *asymmetrisch*: $\langle x, y\rangle \in R \Rightarrow \langle y, x\rangle \notin R$ – das verlangt sozusagen das „Gegenteil“ der Symmetrie. Relationen wie $<$ oder $>$ wären Beispiele dafür.
- *antisymmetrisch*: $\langle x, y\rangle \in R \wedge \langle y, x\rangle \in R \Rightarrow x = y$ – im Gegensatz zur Asymmetrie wird hier erlaubt, dass sowohl $\langle x, y\rangle$ als auch $\langle y, x\rangle$ in der Relation sind. Dann aber müssen die Elemente gleich sein; das ist keine komische Forderung, sondern beschreibt, was Relationen wie $\leq$ oder auch $\subseteq$ (auf einer geeigneten Menge von Mengen) tun. Mit antisymmetrischen Relationen kann man Mengen ordnen.
- *intransitiv*: $\exists\, x, y, z \in R : \langle x, y\rangle \in R \wedge \langle y, z\rangle \in R \wedge \langle x, z\rangle \notin R$ – das ist eine eher schwache Forderung, denn sie wird von allen Relationen erfüllt, die die Forderung aus der Definition der Transitivität nicht erfüllen.
- *antitransitiv*: $\langle x, y\rangle \in R \wedge \langle y, z\rangle \in R \Rightarrow \langle x, z\rangle \notin R$ – dies ist demgegenüber eine starke Forderung, nämlich wieder quasi das „Gegenteil“ der Transitivität. Ein Beispiel wäre die (direkte) Vorgängerrelation auf den Natürlichen Zahlen: Wenn 2 auf 1 folgt und 3 auf 2, dann darf natürlich 3 nicht auch auf 1 folgen.

Problems

(9.1) Wir vergessen gerade mal alle (berechtigten) Einwände von LinguistInnen und nehmen an, wir könnten die Begriffe (a) „Gegenstand“, (b) „Möbel“, (c) „Tisch“, (d) „Sitzgelegenheit“, (e) „Stuhl“, (f) „Bürostuhl“ und (g) „Sofa“ in eine Relation $\succ$ „ist Oberbegriff von“ bringen. So gilt beispielsweise Sitzgelegenheit $\succ$ Sofa, was wir gleich, um ein Karpaltunnelsyndrom zu vermeiden, als $d \succ g$ abkürzen wollen. Schreibt die Relation aus. **(L)**

(9.2) Ist die Relation aus der letzten Aufgabe reflexiv, symmetrisch oder transitiv? Prüft das zunächst stur entlang der Definitionen und überlegt euch dann, ob eure Ergebnisse mit eurer Intuition übereinstimmen. **(L)**

(9.3) Ist $<$ eine antisymmetrische Relation?

10. Relationen II

inverse Relation
Komplement
Äquivalenzrelation
Äquivalenzklassen

Die *inverse Relation* R^{-1} entsteht durch Vertauschung von Definitions- und Wertebereich und der Tupelelemente:

$$R^{-1} = \{\langle b, a\rangle \mid \langle a, b\rangle \in R\}.$$

Davon wohl zu unterscheiden ist das *Komplement* einer Relation $R \subseteq M \times N$, nämlich $\bar{R} = (M \times N) \setminus R$.

Beispiel 1: $M = \{1, 2, 3\}$, $R \subset M \times M$. Relation „ist kleiner als“ ist $R = \{\langle 1, 2\rangle, \langle 1, 3\rangle, \langle 2, 3\rangle\}$. $\bar{R}$ ist dann „ist nicht kleiner als“ oder „ist größer oder gleich“, R^{-1} „ist größer als“.

Beispiel 2 (gilt nur für katholische Gegenden): Die Relation R, „ist Ehemann von“ mit Männern als Wertebereich. $\bar{R}$ ist die Relation „ist nicht verheiratet mit“, R^{-1} ist dagegen „ist Ehefrau von“ auf der Menge der Frauen.

Äquivalenzrelation

$R \subseteq M \times M$ heißt *Äquivalenzrelation*, wenn sie reflexiv, symmetrisch und transitiv ist.

Eine Äquivalenzrelation teilt eine Menge in *Äquivalenzklassen* (disjunkte Teilmengen, deren Vereinigung die Menge ist) ein.

Beispiel 1: Sei M die Menge aller deutschen Wörter. Die Relation

$$R = \{\langle a, b\rangle \mid a \text{ und } b \text{ fangen mit dem gleichen Buchstaben an}\}$$

ist eine Äquivalenzrelation.

Beispiel 2 (Klassiker): Restklassen. Dabei ist $R \subset \mathbb{N} \times \mathbb{N}$ mit $R = \{\langle x, y\rangle \mid x \bmod n = y \bmod n\}$ für ein festes n.

$x \bmod n$ ist dabei der Rest bei der ganzzahligen Division von x durch n. Beispielsweise ist $7 \bmod 2 = 1$, weil $5 = 2 \times 3 + 1$.

Die Äquivalenzklassen für $n = 3$ wären damit:

$$\begin{aligned}
\overline{0} &= \{0, 3, 6, 9, 12, \ldots\} \\
\overline{1} &= \{1, 4, 7, 10, 13, \ldots\} \\
\overline{2} &= \{2, 5, 8, 11, 14, \ldots\} \\
\overline{3} &= \{3, 6, 9, 12, 15, \ldots\}
\end{aligned}$$

Toll daran ist beispielsweise, dass die Menge der Äquivalenzklassen mit der gewohnten Addition und Multiplikation der natürlichen Zahlen genau dann Körper bilden, wenn n prim ist (aber das führt hier zu weit).

Hüllen

reflexive Hülle

transitive Hülle

Die *reflexive Hülle* R^{refl} einer Relation R entsteht, indem man zu jedem $\langle \alpha, \beta \rangle \in R$ noch $\langle \alpha, \alpha \rangle$ und $\langle \beta, \beta \rangle$ zu R hinzufügt:

$$\begin{aligned} R^{\text{refl}} = R &\cup \{\langle \alpha, \alpha \rangle \mid \langle \alpha, x \rangle \in R\} \\ &\cup \{\langle \beta, \beta \rangle \mid \langle x, \beta \rangle \in R\}. \end{aligned}$$

Beispiele: Die reflexive Hülle von $<$ ist $\leq$. Die reflexive Hülle von $\leq$ ist $\leq$ selbst (das gilt natürlich generell: die reflexive Hülle einer reflexiven Relation ist die Relation selbst). Die reflexive Hülle der Relation „x liebt y“ führt zu einer Population von NarzistInnen.

Die *transitive Hülle* R^+ einer Relation R entsteht, indem man für alle Paare von Tupeln $\langle \alpha, \beta \rangle$ und $\langle \beta, \gamma \rangle$ auch das Tupel $\langle \alpha, \gamma \rangle$ zu R hinzufügt und das so lange macht, bis sich an R nichts mehr ändert:

$$R^+ = \bigcap \{\tau \subseteq M \times M \mid R \subseteq \tau \wedge \tau \text{ transitiv}\}.$$

In der formalen Definition werden alle transitiven Relationen über M genommen und diejenigen, die R enthalten, geschnitten. Im Effekt ergibt sich die kleinste transitive Relation, die R enthält (weil diese Teilmenge aller anderen ist und der Schnitt einer Menge mit ihrer Obermenge die Menge selbst liefert) – eben die transitive Hülle.

Da in dieser Vorschrift letzlich über eine Potenzmenge iteriert wird, liefert sie sicher keinen guten Algorithmus zur Berechnung der Hülle. Der würde sich eher aus einer Formalisierung unserer intuitiven Definition ergeben.

Beispiele: Sei $R = \{\langle x, y \rangle \mid y = x + 1 \wedge x, y \in \mathbb{N}\}$ die Nachfolgerelation auf den natürlichen Zahlen. Dann ist R^+ die Größer-Relation auf $\mathbb{N}$.

Die transitive Hülle der Relation „x kennt y“ verbindet (vermutlich) alle Menschen auf der Welt direkt.

Die transitive Hülle von „x ist mit y durch eine Straße verbunden“ verbindet praktisch alle Orte eines Kontinents oder einer Insel miteinander.

Und noch ein „künstliches“ Beispiel, weil die Berechnung der transitiven Hülle manchmal etwas tricky aussieht. Gegeben sei die Relation

$$R = \{\langle a, b \rangle, \langle b, c \rangle, \langle b, e \rangle, \langle e, d \rangle\}.$$

Sie ist offensichtlich nicht transitiv, weil z.B. $\langle a, c \rangle$ fehlt. Um nun die Hülle zu bauen, sehen wir uns für jedes Tupel das zweite Element an und sehen nach, ob dieses als erstes Element eines Tupels auftaucht. Wenn ja, fügen wir ein neues Paar aus dem ersten Element des Ausgangstupels und dem zweiten Element des gefundenen Tupels hinzu. Das führt im Beispiel zu folgender Relation:

$$R' = \{\langle a, b \rangle, \langle a, c \rangle, \langle a, e \rangle, \langle b, c \rangle, \langle b, e \rangle, \langle b, d \rangle, \langle e, d \rangle\}$$

Auch diese Relation ist noch nicht transitiv, weil z.B. $\langle a, d \rangle$ fehlt. Wir wiederholen die Prozedur:

$$R^+ = \{\langle a, b \rangle, \langle a, c \rangle, \langle a, e \rangle, \langle a, d \rangle, \langle b, c \rangle, \langle b, e \rangle, \langle b, d \rangle, \langle e, d \rangle\}.$$

Diese Relation ist transitiv und mithin die transitive Hülle von R.

Problems

(10.1) Zeigt für die Restwertbildung in $\mathbb{N}$ zu festem Divisor n die drei Eigenschaften, die eine Äquivalenzrelation hat. Tipp: $x = y \bmod n$ heißt nichts anderes, als dass es eine Zahl a gibt, so dass $x = an + y$ mit $a \in \mathbb{N}$. **(L)**

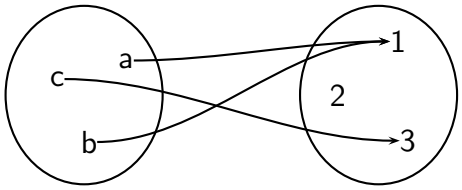


Fig. 4

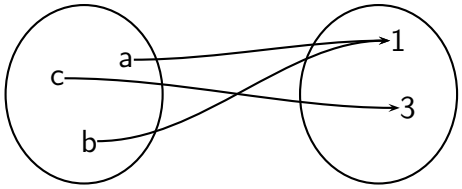


Fig. 5

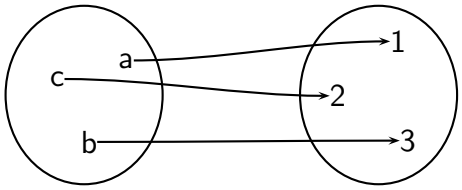


Fig. 6

Funktion
Abbildung
partielle Funktion
Urbild
surjektiv
injektiv
bijektiv

11. Funktionen

Eine (totale) *Funktion* (oder *Abbildung*) ist eine Relation, die jedem Element des Definitionsbereichs genau ein Element des Wertebereichs zuordnet. Für *partielle Funktion*en ist „genau ein“ durch „höchstens ein“ zu ersetzen.

Schreibweise: $f : M \to N$ statt $f \subset M \times N$ und $f(m) = n$ für $\langle m, n \rangle \in f$.

Vokabeln

Das Bild einer Menge $A \subseteq M$ unter einer Funktion f ist die Menge $f(A) = \{f(a) | a \in A\}$.

Das *Urbild* einer Menge N unter einer Funktion $f: M \to N$ ist die Menge $f^{-1}(N) := \{m \in M : f(m) \in N\}$.

Vorsicht: Jede Menge hat unter jeder Funktion ein Urbild; im Allgemeinen ist aber die Umkehrrelation keine Funktion. Wenn man totale Funktionen will, sind nur für bijektive Funktionen auch die Umkehrrelationen wieder Funktionen.

$f: M \to N$ heißt

- *surjektiv*, wenn $\forall\, n \in N\, \exists\, m \in M : f(m) = n$. N ist also gerade das Bild von M unter f.
- *injektiv*, wenn $\forall\, m_1 \in M\, \forall\, m_2 \in M : f(m_1) = f(m_2) \Rightarrow m_1 = m_2$, also nie zwei verschiedene Elemente von M das gleiche Bild haben. (Vgl. streng monotone Funktionen in der Analysis)
- *bijektiv*, wenn sie injektiv und surjektiv ist.

(cf. Fig. 4)

Nicht surjektiv, nicht injektiv

(cf. Fig. 5)

surjektiv, nicht injektiv

(cf. Fig. 6)

surjektiv, injektiv: bijektiv

Problems

(11.1) Erinnert euch an eure Schulmathematik und beurteilt, ob die folgenden Funktionen surjektiv, injektiv oder bijektiv sind:

1. $f_1: \mathbb{R} \to \mathbb{R}$, $f_1(x) = x$
2. $f_2: \mathbb{R} \to \mathbb{R}$, $f_2(x) = x^2$
3. $f_3: \mathbb{R} \to \mathbb{R}^+$, $f_3(x) = x^2$ ($\mathbb{R}^+$ sind dabei die nichtnegativen reellen Zahlen)
4. $f_4: \mathbb{R}^+ \to \mathbb{R}^+$, $f_4(x) = x^2$
5. $f_5: \mathbb{R} \to \mathbb{R}^+$, $f_5(x) = \exp(x)$

(L)

(11.2) Für streng monoton wachsende Funktionen gilt: $x < y \Rightarrow f(x) < f(y)$. Zeigt, dass streng monotone Funktionen immer injektiv sind. Tipp: Betrachtet die drei möglichen Fälle $x < y$, $x = y$ und $x > y$. **(L)**

12. Kardinalität und Induktion

Zwei Mengen M_1, M_2 haben gleiche *Kardinalität* (sind gleichmächtig) gdw. es eine Bijektion zwischen ihnen gibt.

Insbesondere hat M die Kardinalität n (in Zeichen: $|M| = n$), wenn es eine Bijektion $\{1, \ldots, n\} \to M$ gibt. Die Menge heißt dann *endlich*.

Eine Menge, die bijektiv auf die natürlichen Zahlen $\mathbb{N}$ abbildbar ist, heißt *abzählbar unendlich*. Dazu gehören beispielsweise die Brüche. Schreibweise: $|M| = \aleph_0$ (Aleph Null).

Es gibt *überabzählbar unendliche* Mengen.

Gilt eine Aussage für eine abzählbar unendliche Menge von Objekten, kann ein *Induktionsbeweis* geführt werden. Dieser besteht aus drei Schritten:

1. Induktionsanfang: Behauptung ist für n richtig (meistens $n = 0$ oder $n = 1$)
2. Induktionsannahme: Die Behauptung gelte für ein beliebiges $m \geq n$
3. Induktionsschluss: Aus der Richtigkeit für m folgt die Richtigkeit für $m + 1$

Beispiel: $\sum_{i=0}^{n} i = n(n+1)/2$.

1. Induktionsanfang: $n = 1$: $0 + 1 = 2/2$
2. Induktionsannahme: Behauptung ist für $m \geq 1$ richtig.
3. Induktionsschluss:

$$\begin{aligned}\sum_{i=0}^{m+1} i &= m + 1 + \sum_{i=0}^{m} i \\ &= m + 1 + \frac{m(m+1)}{2} \\ &= \frac{m^2 + 3m + 2}{2} = \frac{(m+1)(m+2)}{2}.\end{aligned}$$

Historische Anmerkung: Das Wort „Induktion“ kommt aus der klassischen Logik, wo – grob gesprochen – die Induktion der Schluss vom Speziellen aufs Allgemeine, die Deduktion der Schluss vom Allgemeinen aufs Spezielle ist. An und für sich ist die Induktion ein gefährliches Pflaster, das aus guten Gründen nicht als Beweistechnik zugelassen ist.

Zum Beispiel könnte jemand die speziellen Fälle 1, 3, 5, 7, 11, 13, 17 untersuchen und feststellen, dass alle ungeraden, die er/sie untersucht hat, prim sind. Der Schluss, alle ungeraden Zahlen seien prim, ist natürlich grottenfalsch.

Andererseits hilft die Induktion, die Richtung, in der man suchen soll, zu finden. Im Beispiel könnte man z.B. vermuten, dass alle Primzahlen ungerade sind. Das ist – bis auf den Sonderfall

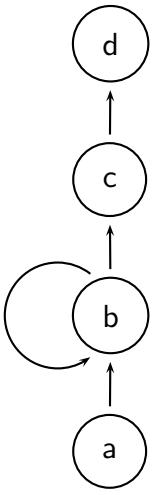


Fig. 7

Graph
Knoten
Kanten
Pfad
Zyklus
Baum
Blätter

der Zwei – auch richtig, was man dann deduktiv nachweisen kann (im Groben: in allen geraden Zahlen steckt der Primfaktor zwei).

Die *vollständige* Induktion ist auf der anderen Seite völlig in Ordnung, eben weil sie über abzählbar unendlichen Mengen operiert und auf diese Weise *alle* Spezialfälle sicher abdecken kann.

Problems

(12.1) Haben die Mengen $\{-n, -n+1, \ldots, -1, 0\}$ und $\{0, 1, \ldots, n\}$ die gleiche Kardinalität? Wenn ja, schreibt die Bijektion f_1 explizit hin.

Wie ist das mit den Mengen $\{0, 1, 2, \ldots, n\}$ und $\{0, 2, 4, 6, \ldots, 2n\}$? Und mit den Mengen $\mathbb{N}$ und $\{0, 2, 4, 6, \ldots\}$ (also alle geraden Zahlen)? Schreibt auch hier ggf. die Bijektionen f_2 bzw. f_3 hin. **(L)**

(12.2) Beweist durch vollständige Induktion:

$$\prod_{i=1}^{n}(-1) = \begin{cases} 1 & n \text{ gerade} \\ -1 & n \text{ ungerade} \end{cases}$$

Beachtet: $\prod_{i=1}^{1} a = a$. Wir wollen das nur für $n > 0$ wissen.

Tipp: Wenn schon in der Behauptung eine Fallunterscheidung steht, ist es wohl eine gute Idee, auch im Beweis die Fälle getrennt zu unterscheiden. **(L)**

13. Graphen

Ein (endlicher) gerichteter *Graph* G ist ein Paar $G = \langle K, E\rangle$ aus einer endlichen Menge von *Knoten* K und einer Relation $E \subset K \times K$, den *Kanten*.

Beispiel: $E = \{\langle a, b\rangle, \langle b, c\rangle, \langle c, d\rangle, \langle b, b\rangle\}$, $K = \{a, b, c, d\}$.

Ein *Pfad* der Länge n in einem Graphen ist eine Folge von Knoten k_i, so dass $\langle k_i, k_{i+1}\rangle \in E$ für alle $1 \leq i < n$ (es werden also $n + 1$ Knoten besucht).

Ein Pfad der Länge n heißt *Zyklus*, wenn $k_1 = k_n$.

Statt $\langle k_i, k_j\rangle \in E$ schreibt man auch $k_i \to k_j$.

Ein *Baum* ist ein Graph mit

- Genau ein Knoten hat keinen Vorgänger, der *Wurzelknoten*
- Alle anderen Knoten haben genau einen Vorgänger
- In einem Baum existieren keine Zyklen

Üblicherweise fordert man noch eine Ordnung auf der Menge $\{\langle k_i, k_j\rangle\}$ der Kanten ab einem gegebenen Knoten k_i.

Knoten ohne Nachfolger heißen *Blätter*.

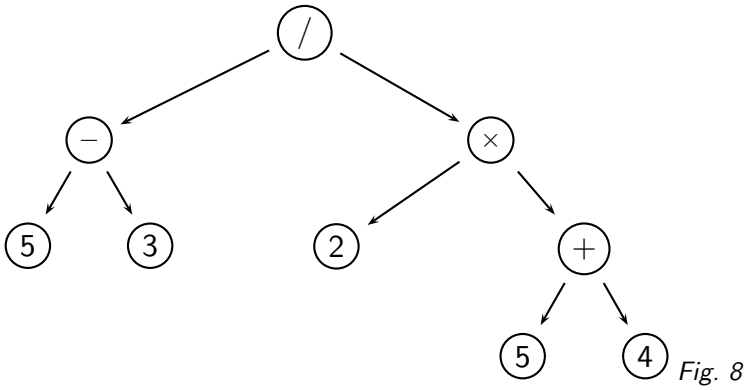


Fig. 8

Alphabet
Zeichen
Wort
Konkatenation
Verkettung
leeres Wort
Länge
Sternbildung

(cf. Fig. 7)

(cf. Fig. 8)

14. Vokabeln

Alphabet: Eine nicht-leere Menge von *Zeichen*. Hier betrachten wir nur endliche Alphabete. Σ bezeichnet im Folgenden immer ein Alphabet. Was wir dabei als Zeichen ansehen, ist eigentlich relativ egal. Wir werden der Einfachheit halber meist lateinische Groß- und Kleinbuchstaben nehmen, manchmal auch ASCII-Zeichen – aber wie schon gesagt wären „Wörter“ der Natursprache genauso möglich wie Morphe, Formelzeichen oder Tokens von Programmiersprachen.

Wort: Eine Folge $(x_i)_{i\in\{1\ldots n\}}$ von Zeichen $x_i \in \Sigma$. Es muss betont werden, dass sich dies deutlich von fast allen Definitionen von „Wort“ in der Linguistik unterscheidet, selbst wenn man als Alphabet tatsächlich ein Alphabet einer Natursprache verwendet. Allerdings wird auch und gerade Morphologie gern mit den hier besprochenen Methoden betrieben, und dann ist ein Wort auch ziemlich gut ein Wort, während ein Zeichen vielleicht ein Morph oder auch ein Buchstabe ist.

Konkatenation, auch *Verkettung*: Wenn $w = (w_i)_{i\in\{1\ldots n\}}$ und $v = (v_i)_{i\in\{1\ldots m\}}$ mit $w_i, v_i \in \Sigma$, dann die Verkettung $u = w\bullet v$ (meistens lassen wir das $\bullet$ weg und schreiben einfach wv) definiert als $u = (u_i)_{i\in\{1\ldots n+m\}}$ mit

$$u_i = \begin{cases} w_i & i \le n \\ v_{i-n} & n < i \le n+m \end{cases}$$

Das ist MathematikerInnensprache für „Wir schreiben die Zeichen halt hintereinander.“ Wie so oft, verbirgt sich hinter komplizierten Zeichen, die einfach die Operation genau und unzweideutig definieren, ein ganz einfacher Sachverhalt. Wenn $w = abc$ und $v = cba$, dann ist eben $w \bullet v = abccba$. Das Ganze kann man auch für Mengen machen: Wenn $M_1, M_2 \subset \Sigma^*$, dann ist

$$M_1 \bullet M_2 = \{w_1 \bullet w_2 \,|\, w_1 \in M_1 \wedge w_2 \in M_2\}.$$

leeres Wort: Das aus null Zeichen bestehende Wort $\epsilon = (\epsilon_i)_{i=0..0}$. Es ist für alle Wörter $w\bullet\epsilon = w$. ϵ spielt also für die Verkettung in etwa die Rolle des Neutralen Elements, ganz wie die Null bei der Addition. N.B. $\emptyset \neq \{\epsilon\}$.

Länge eines Worts: Die Zahl der Zeichen in einem Wort w, in Zeichen $|w|$. Eine elegante rekursive Definition ist: (a) $|\epsilon| = 0$, (b) $|aw| = |wa| = 1+|w|$ für ein Wort $w \in \Sigma^*$ und $a \in \Sigma$. Generell ist die Idee rekursiver (oder induktiver) Definitionen, eine Eigenschaft für etwas „Größeres“ durch das Nächstkleinere zu bestimmen. Hier wird die Länge eines Wortes mit einem angehängten Zeichen durch die Länge des nackten Wortes bestimmt.

Sternbildung: Die Menge aller Worte über einem Alphabet Σ wird mit Σ^* bezeichnet. Eine rekursive Definition ist: (a) $\epsilon \in \Sigma^*$, (b) $w \in \Sigma^* \wedge a \in \Sigma \Rightarrow aw \in \Sigma^*$. Für ein endliches Alphabet ist Σ^* abzählbar unendlich. Dies kann man zeigen durch die Angabe einer Bijektion

zu den natürlichen Zahlen. Eine injektive Abbildung in die natürlichen Zahlen kann man etwa bauen, indem man die ersten $n := |\Sigma|$ Primzahlen p_i mit den Elementen von Σ identifiziert und weiter $q = \prod_{i=1}^{n} p_i$ definiert. Die Abbildung eines Wortes $w = a_{i_1} a_{i_2} \dots a_{i_m}$ auf $\sum_{j=1}^{m} p_{i_j} q^j$ ist wegen der Eindeutigkeit der Primfaktorzerlegung injektiv. Es ist etwas schwieriger zu zeigen, dass diese Abbildung auch surjektiv ist, aber das soll uns jetzt nicht kümmern.

Potenz

Potenz eines Worts: a^i lässt sich induktiv definieren als (a) $a^0 = \epsilon$, (b) $a^i = a \bullet a^{i-1}$.

Formale Sprache

Formale Sprache über Σ: Eine Teilmenge von Σ^*. Das ist die zentrale Definition in unserem Kurs, sie sagt nämlich, worüber wir reden wollen.

Wortproblem

Wortproblem: Beim Wortproblem sucht man eine Funktion, die wahr zurückgibt, wenn ein Wort zu einer Sprache gehört, falsch sonst.

Berechenbarkeit

Berechenbarkeit: Eine Funktion, die nach endlich vielen Berechnungsschritten (wir müssen natürlich eigentlich noch erklären, was das ist, aber das muss vorerst warten) garantiert ein Ergebnis liefert, heißt berechenbar.

Entscheidbarkeit

Entscheidbarkeit: Eine Sprache, deren Wortproblem berechenbar ist, heißt entscheidbar.

Grammatik

Nichtterminalsymbol

Terminalsymbolen

Startsymbol

allgemeine Regelgrammatik

Problems

(14.1)* Wie würdet ihr ab^2 definieren, und was ist daran überhaupt zu definieren? **(L)**

(14.2)* Sei $A = \{1, 2, 3\}$. Was sind die kürzesten (sagen wir, bis Länge 2) Elemente von A^*? Was ist demgegenüber $\mathcal{P}(A)$? **(L)**

(14.3) Wie viele Wörter der Länge n gibt es über einem Alphabet mit N Elementen? **(L)**

(14.4) Wie viele Elemente mit einer Länge $\leq n$ hat die Sternmenge einer Menge mit N Elementen? (Anmerkung: Die Summe, die hier vorkommt, heißt geometrische Reihe – ihr findet Anleitungen zu ihrer Berechnung an vielen Stellen im Netz, und natürlich auch in der Lösung) **(L)**

15. Grammatiken und Ableitungen

Sprachen können wir im Prinzip wie Mengen definieren – sie sind ja nur Mengen. Einfach alle Wörter aufzählen ist aber im Regelfall unmöglich, weil „interessante“ Sprachen typischerweise abzählbar unendlich viele Wörter enthalten.

Da wäre die set comprehension schon viel netter, und eigentlich würde auch die rekursive Definition taugen, nur: wie konkret soll die Definition aussehen, wie können wir das Aussehen der Wörter beschreiben? „Ein a am Anfang, und dann kommt entweder ein o oder ein c, es kann aber auch sein, dass das Wort mit b anfängt, und dann...“ Nein – wir brauchen einen Formalismus für diese Beschreibung, und ein Industriestandard dafür sind Grammatiken, die sozusagen vorschreiben, wie man Wörter nach und nach aufbaut.

Eine *Grammatik* ist ein Tupel $G = \langle \Phi, \Sigma, R, S \rangle$ aus

- einem Alphabet Φ von *Nichtterminalsymbol*en – das sind Symbole, die am Schluss in den Wörtern der Sprache nicht mehr auftauchen sollen, ein bisschen so, wie der Quelltext eines Programms in seiner Ausgabe nichts verloren hat
- einem Alphabet Σ von *Terminalsymbolen*, wobei $\Sigma \cap \Phi = \emptyset$ – das sind die Symbole, die am Schluss in den fertigen Wörtern der Sprache stehen werden. Gelegentlich schreiben wir $\Gamma := \Sigma \cup \Phi$.
- einer Menge von Regeln $R \subset \Gamma^* \times \Gamma^* = \{\langle \alpha_i, \beta_i \rangle\}$, wobei $\alpha_i \notin \Sigma^*$.
- einem *Startsymbol* $S \in \Phi$.

Insbesondere ist $\alpha_i \neq \epsilon$. Also: Regeln ersetzen nie ausschließlich Folgen von Terminalsymbolen (daher deren Name) und ersetzen nie das leere Wort.

Diese Grammatik heißt auch *Typ 0-* oder *allgemeine Regelgrammatik*. Andere Grammatiktypen gehen durch weitere Forderungen an die Regeln hervor.

Den Grammatikbegriff hatten wir eingeführt, um Sprachen beschreiben zu können. Im Groben würden wir jetzt ganz gerne sagen: Die Sprache L besteht aus den Wörtern, die ich durch wiederholte Anwendung der Regeln aus dem Startsymbol erzeugen, wir werden sagen: ableiten kann. Das müssen wir jetzt etwas vornehmer formulieren.

direkt ableitbar

ableitbar

erzeugte Sprache

Seien $u, v \in \Gamma^*$. Dann heißt v aus u *direkt ableitbar*, wenn $u = u_1 w u_2$ und $v = u_1 z u_2$ und $\langle w, z\rangle \in R$.

In einer Grammatik

$$G = \langle \{S\}, \{a, b\}, \{\langle S, b\rangle, \langle S, aSa\rangle\}, S\rangle$$

gilt *abSba* $\to$ *abaSaba* und *bbbS* $\to$ *bbbb*, aber *nicht* $S \to ba$ oder *abS* $\to$ *aSa*. Im Zweifel kann man die Wörter rechts und links nach u_1, u_2, w und z aufteilen und dann sehen, ob die Definition von direkter Ableitbarkeit erfüllt ist.

Gut – wir haben jetzt eine Relation definiert, nämlich die der direkten Ableitbarkeit. Zur Übung kann man mal sehen, welche Eigenschaften sie hat: Sie ist nicht transitiv (es gilt mit der Grammatik oben zwar $S \to aSa$ und $aSa \to aba$, aber nicht $S \to aba$), sie ist nicht symmetrisch und auch nicht reflexiv (es gilt also insbesondere nicht $S \to S$).

Sie reicht aber eigentlich schon für eine rekursive Definition der Sprache $L(G)$, die aus allen Wörtern besteht, die die Grammatik G erzeugt. Probieren wir es erstmal mit einer Sprache L':

1. $S \in L'$
2. Wenn $w \in L'$ und $w \to w'$, dann ist auch $w' \in L'$.

Das Problem an dieser Definition ist, dass in L' allerlei Wörter wie *aaSaa* stehen – wir hatten aber oben verlangt, dass in den fertigen Wörtern nur noch Terminale, aber keine Nichtterminale stehen sollten. Das Problem ist lösbar, wir können einfach sagen, dass $L(G) = \{w \mid w \in L' \wedge w \in \Sigma^*\}$ sein soll.

Das ist keine schlechte Definition, aber die Leute verstecken die Rekursion lieber etwas. Dafür brauchen wir nun die transitive Hülle – wir hatten ja oben beklagt, dass nicht $S \to aba$ gilt, also die direkte Ableitbarkeit nicht transitiv ist. In der transititiven Hülle von $\to$ ist aber das Paar $\langle S, aba\rangle$ durchaus enthalten. Also definieren wir:

Seien $u, v \in \Gamma^*$. Dann heißt v aus u *ableitbar* (in Zeichen: $u \xrightarrow{*} v$), wenn $u = u_1 w u_2$ und $v = u_1 z u_2$ und $\langle w, z\rangle \in P^+$, wo P die durch die direkte Ableitbarkeit definierte Relation ist.

Damit können wir die erzeugte Sprache ganz ohne Rekursion definieren (denn die Rekursion haben wir schon bei der Definition der transitiven Hülle erledigt).

Die erzeugte Sprache

$L(G) = \{w \in \Sigma^* \mid S \xrightarrow{*} w\}$ ist die von einer Grammatik $G = \langle \Phi, \Sigma, R, S\rangle$ *erzeugte Sprache* (umgekehrt gibt es *keine* (eindeutige) von einer Sprache erzeugte Grammatik).

Beispiel: Sei

$$G = \langle \{S, A, B\}, \{b\}, \{S \to AB, \\ A \to BB, B \to b\}, S\rangle.$$

Dann ist

$$P = \{ \langle S, AB\rangle, \langle A, BB\rangle, \langle B, b\rangle, \langle AB, BBB\rangle, \\ \langle BBB, bBB\rangle, \langle bBB, bbB\rangle, \langle bbB, bbb\rangle, \\ \langle bbSbb, bbABbb\rangle, \ldots\}$$

(die drei Punkte deuten an, dass die Relation natürlich noch beliebig viele weitere Elemente hat – rechnet ein paar weitere aus –, die aber zum Glück keine Relevanz für die Bestimmung der erzeugten Sprache haben).

Damit ist

$$P^+ = \{ \langle S, AB\rangle, \langle S, BBB\rangle, \langle S, bBB\rangle, \\ \langle S, bbB\rangle, \langle S, bbb\rangle, \langle A, BB\rangle, \ldots\}.$$

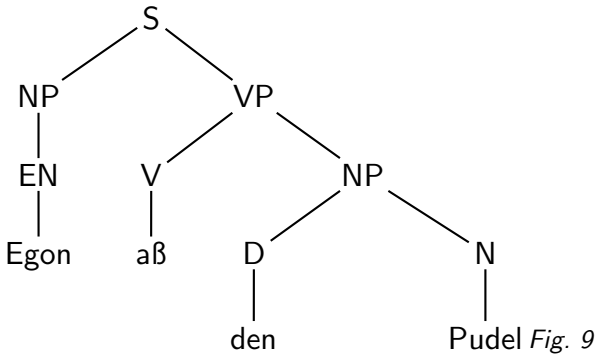


Fig. 9

Ableitung

Parse

Wieder hat diese Relation natürlich beliebig viele weitere Elemente. Allerdings ist *bbb* das einzige Element von Σ^*, das als zweites Element in einem Tupel mit S auftaucht, und damit hätten wir schon die erzeugte Sprache: $L(G) = \{bbb\}$.

Natürlich rechnet niemand wirklich auf *diese* Weise Sprachen aus. Aber das ist die Definition, und irgendwelche anderen Verfahren müssen die *Ergebnisse* dieses eingestandenermaßen umständlichen Verfahrens beweisbar reproduzieren.

Nebenbei bemerkt interessiert man sich in der Regel eher dafür, wie ein Wort erzeugt wurde. Sprachen wortweise Aufzählen ist eine eher ungewöhnliche Beschäftigung.

16. Ableitungen und Syntaxbäume

Eine *Ableitung* (auch *Parse* genannt) in einer Grammatik $G = \langle \Phi, \Sigma, R, S \rangle$ ist eine Folge von Wörtern w_i aus $(\Phi \cup \Sigma)^*$, so dass $w_i \to w_{i+1}$ und $w_1 = S$.

Gegeben sei

$$G = \langle \{S, NP, EN, VP, V, D, N\}, \\ \{\text{Egon}, \text{Pudel}, \text{den}, \text{aß}\}, R, S \rangle$$

mit

$$R = \{S \to NP\ VP, NP \to EN, NP \to D\ N, \\ VP \to V\ NP, EN \to \text{Egon}, N \to \text{Pudel}, \\ V \to \text{aß}, D \to \text{den}\}$$

Eine Ableitung des Wortes „Egon aß den Pudel“ ist

$S \to NP\ VP \to EN\ VP \to$
$Egon\ VP \to Egon\ V\ NP \to Egon\ aß\ NP \to$
$Egon\ aß\ D\ N \to Egon\ aß\ den\ N \to Egon\ aß\ den\ Pudel$

In der Regel gibt es zu einem Wort mehrere Ableitungen. Eine weitere Ableitung für „Egon aß den Pudel“ ist:

$S \to NP\ VP \to NP\ V\ NP \to$
$NP\ V\ D\ N \to NP\ V\ D\ Pudel \to NP\ V\ den\ Pudel \to$
$NP\ aß\ den\ Pudel \to EN\ aß\ den\ Pudel \to Egon\ aß\ den\ Pudel$

Beide Ableitungen führen auf den gleichen Baum:

(cf. Fig. 9)

Wie entsteht dieser Baum aus den Ableitungen? Nun, jeder innere Knoten (d.h. Knoten, der kein Blatt ist) k repräsentiert eine Regelanwendung, wobei die Wurzel des Unterbaums unterhalb von k der linken Seite der Regel entspricht, die direkten Nachfolger den Symbolen auf der rechten Seite der Regel. Um also den Baum zu malen, muss man eigentlich nur bei jedem Ableitungsschritt die angewandte Regel identifizieren und an der passenden Stelle die Symbole auf der rechten Seite

dieser Regel als neue Blätter hinmalen; die passende Stelle wiederum ist das Nichtterminalsymbol, das ersetzt wurde.

Dabei kann es vorkommen, dass zwei Mal das gleiche Nichtterminalsymbol als Blatt im augenblicklichen Parsebaum vorkommt – der dritte Schritt in der zweiten Ableitung oben ist ein Beispiel dafür, wir haben zwei Mal *NP*. Es ist dann natürlich entscheidend wichtig, dass man die richtige *NP* zum Ersetzen wählt.

Bäume dieser Art können natürlich nur gebildet werden, wenn auf der linken Seite aller Regeln nur ein Nichtterminalsymbol steht – andernfalls wäre jedes Symbol auf einer linken Seite Vorgänger jeden Symbols auf der rechten Seite und das Ergebnis wäre kein Baum mehr.

regulärer Ausdruck

reguläre Sprache

Invarianten

17. Reguläre Ausdrücke und Sprachen

Ein *regulärer Ausdruck* über einem Alphabet Σ ist definiert durch

- $\emptyset$ ist regulär
- $\{a_i\}$, $a_i \in \Sigma$ ist regulär
- A_1, A_2 regulär $\Rightarrow A_1 \cup A_2$
- A_1, A_2 regulär $\Rightarrow A_1 \bullet A_2$
- A regulär $\Rightarrow A^*$ regulär

Vorsicht: Die *Ausdrücke* hier sind sozusagen „verzögert“: Die Mengenoperationen werden nicht ausgeführt, sondern hingeschrieben. Werden sie ausgeführt, erhält man:

Eine *reguläre Sprache* ist eine formale Sprache, die durch einen regulären Ausdruck beschrieben werden kann.

Beispiele: Sei $\Sigma = \{a, b, c\}$. Reguläre Sprachen über Σ sind beispielsweise $((((\{a, b\} \bullet \{b\}) \cup \{c, a\}^*) \bullet \{c\})$ oder Σ^*.

Zählen wir ein paar Elemente der ersten Sprache auf. Wir haben „ganz außen“ eine Verkettung mit $\{c\}$ als zweiten Operanden. Der erste Operand dieser Verkettung ist kompliziert, aber wir können ihn nach und nach ansehen. Er besteht aus einer Vereinigung, deren erster Operand zur fertigen Sprache (d.h. nach der Verkettung mit dem $\{c\}$)

- abc
- bbc

beiträgt. Der zweite Operand der Vereinigung ist unendlich, wir können mit der Aufzählung der fertigen Wörter also nur anfangen:

- c (weil ϵ in jeder Sternmenge enthalten ist)
- cc
- ac
- ccc
- cac
- acc
- aac

Das sind alle Wörter der Sprache, die aus drei Zeichen aufgebaut sind. Man sieht, dass Sprachen selbst im einfachsten Fall (das sind die regulären Sprachen) schon sehr unübersichtlich werden können.

Deshalb ist es wichtig, *Invarianten* zu erkennen, Eigenschaften, die allen Wörtern gemein sind. Hier ist das beispielsweise, dass am Ende jeden Wortes immer ein c steht – das ist in diesem Fall bereits am regulären Ausdruck zu erkennen (die letzte Verkettung). Invarianten können aber auch etwas komplizierter sein – im Beispiel wäre eine weitere Invariante, dass alle Wörter, die ein b enthalten, mit bc enden (überzeugt euch durch Inspektion des Ausdrucks von dieser Eigenschaft).

Möglichkeiten und Grenzen

rechtslinear

Reguläre Ausdrücke werden z.B. von grep, emacs, vi, perl, python und vielen anderen verstanden. Schreibweise dort: Sternbildung durch $*$, Mengenbildung durch `[]` (für einzelne Zeichen) oder `(.|...)`, Vereinigung durch `|`, Verkettung durch hinereinanderschreiben. Die beiden regulären Ausdrücke oben würden damit etwa `([ab]b|[ac]*)c` bzw. `[abc]*` entsprechen.

Reguläre Ausdrücke können z.B. für Zahlen angegeben werden. Sie können aber nicht Elemente gegeneinander aufzählen: *Es gibt keinen regulären Ausdruck, der die Balanciertheit von Klammern feststellen kann* (das werden wir später beweisen).

Klassisch: $\{a^i b a^i \mid i \in \mathbb{N}\}$ ist *keine* reguläre Sprache.

Problems

(17.1) Im Satzsystem TEX folgen Kontrollsequenzen ohne weitere Einstellungen ungefähr folgendem regulären Ausdruck:

$$\{\backslash\} \bullet \Big((\{a, \ldots, z, A, \ldots, Z\} \bullet \{a, \ldots, z, A, \ldots, Z\}^*) \cup \{.\,, !\,, ", \$, \%, \&, (,), =, ?, *, +, -, @\} \Big)$$

(tatsächlich sind noch mehr Sonderzeichen erlaubt, und das System ermöglicht, diese Ausdrücke beliebig umzudefinieren, aber das ist hier nicht so relevant).

Findet Invarianten der von diesem regulären Ausdruck erzeugten Sprache. **(L)**

(17.2) Bildet den Partizip Perfekt von einer Handvoll deutscher Verben. Findet ihr in der dadurch definierten Sprache Invarianten?

Geht auf eine nicht allzu öde Webseite und lasst euch von eurem Browser die darin eingebetteten Links anzeigen (bei einem englischsprachigen Firefox geht das z.B. durch Tools/Page Info, Tab „Links“). Findet ihr in der durch diese Links (Feld „Address“) definierten Sprache Invarianten?

18. Einseitig Lineare Grammatiken

Ich hatte oben verkündet, dass wir Sprachen durch Grammatiken definieren wollten – und nun haben wir stattdessen einen ganz anderen Formalismus eingeführt, nämlich reguläre Ausdrücke. Auch diese beschreiben, wie die Wörter einer Sprache aussehen sollen. Wenn der Grammatikformalismus auch nur irgendetwas taugt, dann sollte er eigentlich auch „können“, was die regulären Ausdrücke können. Das ist in der Tat so.

Eine Grammatik $G = \langle \Phi, \Sigma, R, S \rangle$ heißt *rechtslinear* (linkslinear), wenn alle Regeln eine der beiden Formen

- $A \to w$ oder
- $A \to wB$ ($A \to Bw$)

haben, wobei $A, B \in \Phi$ und $w \in \Sigma^*$ *(Chomsky-3-Grammatik)*.

Lemma: Jede reguläre Sprache wird durch eine rechtslineare Grammatik erzeugt.

Wir wollen die beiden Behauptungen des Satzes getrennt beweisen. Dabei beschränken wir uns auf den rechtslinearen Fall, der linkslineare wäre ähnlich zu behandeln.

Beweis: Wir geben für alle Operatoren in einem regulären Ausdrucks rechtslineare Grammatiken.

1. $\emptyset$: $G_\emptyset = \langle \{S\}, \Sigma, \{S \to S\}, S \rangle$.
2. $\{a_i\}$: $G_{a_i} = \langle \{S\}, \Sigma, \{S \to a_i\}, S \rangle$.
3. $L(G_1) \cup L(G_2)$: $G_\cup = \langle \Phi_1 \cup \Phi_2, \Sigma_1 \cup \Sigma_2, R_1 \cup R_2 \cup \{S \to S_1, S \to S_2\}, S \rangle$.Dabei sollen $G_1 = \langle \Phi_1, \Sigma_1, R_1, S_1 \rangle$ und $G_2 = \langle \Phi_2, \Sigma_2, R_2, S_2 \rangle$ rechtslineare Grammatiken sein, wobei $\Phi_1 \cap \Phi_2 = \emptyset$ (das ist durch Umbenennen immer möglich).
4. $L(G_1) \bullet L(G_2)$: $G_\bullet = \langle \Phi_1 \cup \Phi_2, \Sigma_1 \cup \Sigma_2, \{A \to wS_2 \mid A \to w \in R_1\} \cup \{A \to wB \mid A \to wB \in R_1\} \cup R_2, S_1 \rangle$. Hier hätte man versucht sein können, einfach wie im Fall eben eine Regel $S \to S_1 S_2$ zur Vereinigung der beiden Regelmengen hinzuzufügen. Das würde auch

in der Tat die Verkettung erzeugen, nur wäre die resultierende Regelmenge nicht mehr rechtslinear. Das würde unseren Beweis kaputt machen, und deshalb machen wir den etwas aufwändigeren Trick, einfach alle Regeln, die ein Wort aus $L(G_1)$ vollenden würden (die also die Form $A \to w$ haben) so umzuschreiben, dass ein Wort aus $L(G_2)$ folgt (wir hängen also S_2 an). konstruktiv

5. $L(G)^*$: $G_* = \langle \Phi, \Sigma, \{A \to wS \,|\, A \to w \in R\} \cup \{A \to wB \,|\, A \to wB \in R\} \cup \{S \to \epsilon\}, S\rangle$.

Lemma: Jede durch eine rechtslineare Grammatik erzeugte Sprache wird auch durch einen regulären Ausdruck beschrieben.

Der folgende Beweis ist *konstruktiv*, d.h. er beweist nicht einfach nur, *dass* es einen regulären Ausdruck zu einer einseitig linearen Grammatik gibt, sondern gibt sogar ein Verfahren an, wie er zu konstruieren sei. Wir wollen den Beweis eher informell und wortreich angehen, quasi zur Eingewöhnung.

Wir haben eine rechtslineare Grammatik $G = \langle \Phi, \Sigma, R, A_1\rangle$. Φ soll zwecks einfacher Notation aus A_1, A_2,..., A_n bestehen (das ist durch Umbenennen immer zu garantieren). Σ ist beliebig, R darf nur rechtslineare Regeln enthalten.

Der Hit an den rechtslinearen Grammatiken ist, dass die Art, wie sie ihre Wörter erzeugen, recht einfach ist: Sie wachsen immer nur nach hinten. Vor jeder Ableitung steht immer ein Nichtterminalsymbol ganz am Ende des augenblicklichen Wortes. Nach dieser Ableitung steht dann entweder wieder ein Nichtterminalsymbol am Ende, oder aber ein Terminalsymbol. Im zweiten Fall ist die Ableitung zu Ende, weil wir ohne Nichtterminalsymbol nicht weiter ableiten können.

Damit kommt der letzten Ableitung eine besondere Bedeutung zu, und wir wollen sehen, was dabei passieren kann.

Wir definieren dazu die Mengen $N(j)$ als die Nichtterminale, die jeweils von A_j aus direkt erzeugt werden, formal

$$N(j) = \{w \,|\, w \in \Sigma^* \wedge A_j \to w \in R\}.$$

Damit wir nicht wahnsinnig werden, schreiben Formeln dieser Art ab jetzt als $N(j) = \{w \,|\, A_j \to w\}$ – die Notation ist kürzer und ähnlich klar.

Diese Mengen sagen uns gerade, was am Ende eines Wortes stehen kann. Wenn A_j überhaupt jemals in Ableitungen vorkommt, muss es auch Wörter geben, in denen ein Element von $N(j)$ am Ende steht, denn auch das A_j kann in rechtslinearen Grammatiken immer nur am Ende eines Wortes (aus Γ^* – in Wörtern der erzeugten Sprache hat es als Nichtterminalsystem natürlich nichts verloren) auftreten. Wenn es also überhaupt einen regulären Ausdruck gibt, der $L(G)$ beschreibt, muss er die Form $\ldots \bullet N(j)$ haben.

Leider ist das, was an die Stelle der drei Punkte kommt, unübersichtlich. Es ist naheliegend, das Problem aufzuspalten („divide et impera“), aber wie? Nun, angenommen, wir dürften nur zwei Ableitungsschritte machen. Dann müssten wir im ersten Schritt irgendeine Regel der Art $A_1 \to w' A_j$ anwenden, im zweiten Schritt käme der eben skizzierte „Abschluss“ dran.

Was sind die möglichen w', wenn wir über A_j zum Ende kommen? Das muss einfach die Menge $S(j) = \{w' \,|\, A_1 \to w' A_j\}$ sein. Dann wäre unsere Sprache $\bigcup_{j=1\ldots n} S(j) \bullet N(j)$, die Menge aller Wörter, die durch Verkettung von Kram, der von A_1 kommend vor A_j stehen kann, verkettet mit allem Kram, der aus A_j werden kann, ohne dass ein weiteres Nichtterminal auftaucht, und den Kram dann wieder für alle A_j vereinigt.

Halt – da ist noch Fehler: Es kann ja auch passieren, dass es Regeln $A_1 \to w$ gibt, und die w, die dabei auftauchen, wären noch nicht erfasst. Das ist schnell repariert:

$$L(G) \overset{\text{Leider nicht wirklich}}{=} N(1) \cup \bigcup_{j=1\ldots n} \big(S(j) \bullet N(j)\big).$$

Damit hätten wir den regulären Ausdruck, weil sowohl $S(j)$ als auch $N(j)$ endlich und damit regulär sind.

Leider ist es nicht so einfach, weil es eben mehr als zwei Ableitungsschritte geben kann. Aber die Idee trägt ja vielleicht: Definieren wir mal die Menge aller $w \in \Sigma^*$, die vor der letzten Ableitung vor einem A_j stehen können, $\tilde{M}(j)$. Mit dieser *Schreibweise* sieht der resultierende Ausdruck immer noch ziemlich regulär aus:

$$L(G) = N(1) \cup \bigcup_{j=1\ldots n} \big(\tilde{M}(j) \bullet N(j)\big).$$

In Wirklichkeit sind wir aber nicht viel weiter, denn ein regulärer Ausdruck ist das nur dann, wenn die $\tilde{M}(j)$ wirklich regulär sind. Um das zu zeigen, braucht es einen Trick, der in der Mathematik populär ist – wir sehen uns etwas allgemeinere Gebilde an, nämlich die Mengen der Wörter in $w \in \Sigma^*$, die bei einer Ableitung $A_i \stackrel{*}{\to} wA_j$ auftreten können,

$$\hat{M}(i,j) = \{w \in \Sigma^* \mid A_i \stackrel{*}{\to} wA_j\}.$$

Speziell ist $\tilde{M}(j) = M(1,j)$, wenn wir also zeigen können, dass die $\hat{M}(i,j)$ regulär sind, sind wir auch schon fertig.

Leider ist auch den $\hat{M}$ die Regularität noch nicht anzusehen. Hier kommt der eigentliche „kreative Akt“ des Beweises, wir müssen nämlich besonders geschickt aufteilen. Dazu definieren wir Mengen $M(i,j,k)$, die Teilmengen von $\hat{M}(i,j)$, bei denen während der Ableitung nur die ersten k Nichtterminalsymbole $A_1, \ldots, A_k$ verwendet werden. So seltsam diese Mengen zunächst auch sind, es ist klar, dass

$$\hat{M}(i,j) = M(i,j,n).$$

Dieser Umweg lohnt sich, weil man die $M(i,j,k)$ relativ leicht auf „einfachere“ $M(i,j',k')$ zurückführen kann. Mit einem Induktionsbeweis kann man dann zeigen, dass sie selbst und damit auch die $\hat{M}(i,j)$ als endliche Vereinigung von $M(i,j,k)$ regulär sind. Mit ihnen sind auch alle $\hat{M}(i,j)$ regulär, also auch alle $\tilde{M}(j)$ und damit schließlich nach der Formel oben auch $L(G)$ selbst.

Diesen entscheidenden Beweisschritt lagern wir in einen Hilfssatz aus. In der Mathematik sagt man statt Hilfssatz gerne auch Lemma – wir schließen uns dem an.

Lemma: Die Mengen $M(i,j,k)$ sind regulär.

Beweis durch Induktion.

Induktionsanfang: $k = 0$. $M(i,j,0)$ ist die Menge der w, die beim Übergang von A_i auf A_j unter Verwendung *keiner* anderen Nichtterminale (A_i und A_j eingeschlossen) entstehen können. Das ist aber gerade die Menge

$$M(i,j,0) = \{w \mid A_i \to wA_j\}.$$

Da es nur endlich viele Regeln gibt, ist das eine endliche Menge und deshalb regulär.

Induktionssannahme: Sei $M(i,j,k)$ für $k \geq 0$ regulär.

Induktionsschluss: Von k auf $k+1$. Wir müssen also aus der Annahme, dass $M(i,j,k)$ regulär ist, auf die Regularität von $M(i,j,k+1)$ schließen.

Was steht in $M(i,j,k+1)$? Nun, auf jeden Fall alles, was schon in $M(i,j,k)$ drin ist. Die Wörter, die dazu kommen, verwenden irgendwo das Nichtterminal A_{k+1}. Auf dem Weg zu dieser Verwendung wird ein Element aus $M(i,k+1,k)$ beigetragen – wir laufen bei A_i los, verwenden die ersten k Nichtterminale und kommen dann schließlich bei A_{k+1} raus. Von dort müssen wir dann wieder zu A_j kommen – dabei wird dann ein Element von $M(k+1,j,k)$ beigetragen. Die Überlegung hat noch eine Lücke: Wir dürfen *mehrmals* bei A_{k+1} vorbeischauen, wir dürfen das sogar machen, so oft wir wollen. Bei jedem dieser Wege laufen wir mit A_{k+1} los und kommen

wieder bei A_{k+1} raus, bekommen also Elemente aus $M(k+1,k+1,k)$ zu unserem Wort dazu. Zusammen heißt das:

$$\begin{aligned} M(i,j,k+1) =& M(i,j,k) \cup \\ & \Big(M(i,k+1,k) \bullet \big(M(k+1,k+1,k)\big)^* \bullet \\ & M(k+1,j,k) \Big). \end{aligned}$$

Nach der Induktionsannahme sind aber $M(i,j,k)$ alle regulär. In diesem Ausdruck werden also reguläre Mengen verkettet, vereinigt und versternt – es ist also ein regulärer Ausdruck, und das zeigt die die Behauptung.

Zusammengefasst gilt also:

Wenn $G = \langle \{A_1, \ldots, A_n\}, \Sigma, R, A_1 \rangle$ rechtslinear ist, gilt

$$L(G) = N(1) \cup \bigcup_{j=1\ldots n} M(1,j,n) \bullet N(j),$$

wobei N und M reguläre Mengen sind.

Insgesamt gilt:

Satz: Reguläre Ausdrücke und einseitig lineare Grammatiken sind äquivalente Formalismen zur Beschreibung von Sprache.

Mit „äquivalent“ ist dabei gemeint, was die beiden „großen“ Lemmata sagen: Zu jedem regulären Ausdruck finde ich eine rechtslineare Grammatik und zu jeder rechtslinearen Grammatik finde ich einen regulären Ausdruck, die jeweils identische Sprachen beschreiben.

Nachgewiesen haben wir das nur für rechtslineare Grammatiken, aber die Beweise für linkslineare Grammatiken funktionieren analog. Linkslineare Grammatiken sind allerdings meistens langweiliger.

Beachtet nochmal, dass ihr nicht links- und rechtslineare Regeln in einer Regelmenge mischen dürft und erwarten könnt, dass das Ergebnis immer noch regulär ist. Die Grammatik

$$G = \langle \{S, A, B\}, \{a, b\}, \{S \to Ba, S \to aA, A \to Sa, B \to aS, S \to b\}, S \rangle$$

beispielsweise erzeugt die Sprache $\{a^i b a^i\}$, die, wie wir noch zeigen werden, nicht regulär ist.

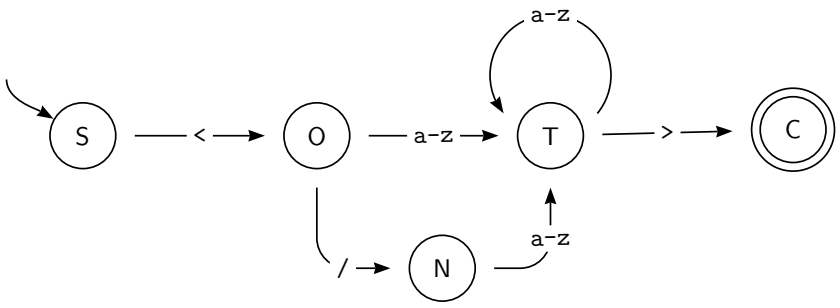


Fig. 10

deterministischer endlicher Automat
Zustandsalphabet
Eingabealphabet
Übergangsfunktion
Endzustand
akzeptiert
markierte Graphen

19. Automaten I

Wir können jetzt reguläre Sprachen entweder durch eine einseitig linearen Grammatik oder aber durch Hinschreiben eines regulären Ausdrucks definieren. Diese Formalismen bieten Methoden, mit deren Hilfe man sämtliche Elemente einer Sprache L aufzählen kann. Zur Entscheidung, ob ein Wort $\tilde{w}$ in einer Sprache ist oder nicht, ist es jedoch höchst mühsam, einfach ein Wort nach dem anderen zu erzeugen und darauf zu warten, dass irgendwann $\tilde{w}$ vorkommt – zumal auf diese Weise ohne weitere Maßnahmen eine Entscheidung, dass $\tilde{w}$ nicht in L ist, für den Regelfall unendlicher Sprachen nicht möglich ist, denn wir können eben nicht ausschließen, dass gerade das nächste Wort in unserer Aufzählung $\tilde{w}$ ist.

Deshalb wollen wir jetzt in gewisser Weise den umgekehrten Weg gehen: Wir möchten ein Verfahren, das möglichst einfach entscheidet, ob ein Wort in einer irgendwie definierten Sprache ist oder nicht („das Wortproblem löst“). Solche Verfahren verwenden gemeinhin Automaten, im Fall der regulären Sprachen die deterministischen endlichen Automaten.

Ein *deterministischer endlicher Automat* (DEA) ist ein Tupel $\langle \Phi, \Sigma, \delta, S, F \rangle$ aus

1. Einem *Zustandsalphabet* Φ
2. Einem zu Φ disjunkten *Eingabealphabet* Σ
3. Einer *Übergangsfunktion* $\delta\colon \Phi \times \Sigma \to \Phi$
4. Einem Startzustand $S \in \Phi$
5. Einer Menge $F \subset \Phi$ von Endzuständen

Ein Automat befindet sich am Anfang im Startzustand S und liest das erste Zeichen $x_1 \in \Sigma$ eines Eingabewortes w. Er geht daraufhin in den Zustand $T = \delta(S, x_1)$ und liest x_2, um damit nach $\delta(T, x_2)$ zu gehen usf. Ist er in einem *Endzustand* (einem Zustand aus F), wenn die Eingabe abgearbeitet ist, ist w *akzeptiert*, sonst nicht.

Der folgende DEA akzeptiert beispielsweise einfache HTML-Tags (sie folgen dem regulären Ausdruck `</?[a-z]+>` (in Unix-Schreibweise) oder

$$\{<\} \bullet \Big((\{/\} \bullet \{a, \ldots, z\}) \cup \{a, \ldots, z\} \Big) \bullet \{a, \ldots, z\}^* \bullet \{>\}$$

Man stellt DEAs im Allgemeinen als Graphen der Übergangsfunktion δ dar. Leider ist δ aber kein Graph (dazu müsste er eine Relation zwischen Φ und Φ, also eine Abbildung $\Phi \to \Phi$ sein). Die Komplikation mit dem Σ lösen wir durch Ausweichen auf *markierte Graphen* (labeled graphs). Dabei malen wir an jede Kante $\langle \varphi_1, \varphi_2 \rangle$, $\varphi_{1,2} \in \Phi$ das Zeichen x, das im in δ enthaltenen Tupel $\langle \langle \varphi_1, x \rangle, \varphi_2 \rangle$ steht. Das klingt kompliziert, ist aber ganz einfach:

(cf. Fig. 10)

Endzustände zeichnet man konventionell als doppelte Kreise, den Anfangszustand markiert man manchmal durch einen kleinen Pfeil „aus dem Unendlichen“.

Sehen wir, ob der HTML-Tag `<br>` von diesem Automaten erkannt wird. Dazu laufen wir den Automaten einfach durch und schreiben über jeden Übergang das Zeichen, was wir für diesen Übergang gelesen haben:

$$S \xrightarrow{<} O \xrightarrow{b} T \xrightarrow{r} T \xrightarrow{>} C$$

C ist ein Endzustand, deshalb ist das ein korrekter HTML-Tag.

akzeptierte Sprache

Demgegenüber wollen wir </g< zurückweisen. Konventionell bleibt ein Automat einfach stehen, wenn ein Zeichen gelesen wird, für das es keinen Übergang gibt (alternativ könnte man einen „Fehlerzustand“ E einführen, und δ durch $\langle\langle\varphi, x\rangle, E\rangle$ ergänzen, wenn es für alle $\psi \in \Phi$ keinen Übergang $\langle\langle\varphi, x\rangle, \psi\rangle \in \delta$ gibt). Dann sieht unser Weg durch den Automaten so aus:

$$S \xrightarrow{<} O \xrightarrow{/} N \xrightarrow{g} T \xrightarrow{<} \text{DEA steht}$$

Wir sind nicht in einem Endzustand angekommen, das Wort wird nicht akzeptiert.

Analog zur Verallgemeinerung der direkten Ableitung $\to$ zur Ableitbarkeit $\xrightarrow{*}$ bei Grammatiken können wir für DEAs eine Übergangsfunktion $\delta^* : \Phi \times \Sigma^* \to \Phi$ für ganze Wörter definieren, etwa durch

- $\delta^*(T, \epsilon) = T$ und
- $\delta^*(T, wx) = \delta\big(\delta^*(T, w), x\big)$.

Wieder ist das mehr eine formale Geschichte – für nichttriviale Grammatiken ist δ^* ein unendlicher Graph, der in der Praxis nicht ganz berechnet werden kann. Die Definition erlaubt uns aber, einen weiteren Begriff zu fassen.

Analog zur von einer Grammatik erzeugten Sprache ist die von einem Automaten $A = \langle\Phi, \Sigma, \delta, S, F\rangle$ *akzeptierte Sprache* $L(A) = \{w \in \Sigma^* \mid \delta^*(S, w) \in F\}$.

Das ist einfach die Menge der Wörter, die auf einen Endzustand führen.

Problems

(19.1) Lasst den Automaten für einfache HTML-Tags auf
< laufen. Schreibt die Zustandsfolge nach dem Muster oben auf. Wird das Wort akzeptiert? **(L)**

(19.2) Reale HTML (na ja, sagen wir XML)-Tags können auch Attribute mit Werten haben. Das sieht dann etwa so aus:

```
<tag att="some stuff">
```

Erweitert den diskutierten Automaten so, dass er damit auch noch zurecht kommt (Hinweis: Ihr wollt den Übergang beim Attributwerte mit einem „Nicht Anführungszeichen“-Label versehen, das dürfen wir jetzt mal)

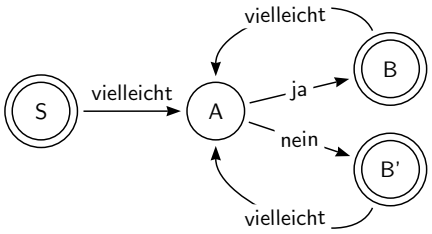


Fig. 11

20. Automaten können reguläre Sprachen

Satz: Jede von einem DEA akzeptierte Sprache ist regulär.

Beweisidee: Konstruiere aus DEA $\langle \Phi, \Sigma, \delta, S, F \rangle$ die Grammatik $G = \langle \Phi, \Sigma, R, S \rangle$, wobei die Regelmenge

$$R = \{B \to xC \mid \delta(B, x) = C\} \cup \{B \to \epsilon \mid B \in F\}$$

ist. Korrekterweise muss jetzt noch per Induktion über die Wortlänge nachgewiesen werden, dass $S \overset{*}{\to} wT \Leftrightarrow \delta^*(S, w) = T$ – ggf. Klabunde, 59.

Umkehrung lässt sich leider nicht so direkt zeigen; Gegenbeispiel:

$$\begin{aligned} R = \{&S \to \text{vielleicht } A, S \to \text{vielleicht } B, \\ &S \to \epsilon, A \to \text{ja } S, B \to \text{nein } S\} \end{aligned}$$

Beim dem Lesen von „vielleicht“ müsste sich der Automat entscheiden, ob er nach A *oder* B gehen und dann entweder nur „ja“ (im Fall A) oder „nein“ (im Fall B) akzeptieren soll. Die ÜF eines DEA muss aber zu jedem Eingabezeichen *eindeutig* den nächsten Zustand geben.

Überlegt man sich, dass die mit diesen Regeln erzeugte Sprache

$$L(G) = \Big(\text{vielleicht} \bullet (\{\text{ja}\} \cup \{\text{nein}\})\Big)^*$$

ist, ist es nicht schwer, folgenden DEA zu konstruieren, der $L(G)$ akzeptiert:

(cf. Fig. 11)

Das Problem ist nur, dass wir nicht wissen, ob sowas wirklich immer geht, ganz abgesehen davon, dass wir ohnehin lieber ein systematisches Verfahren hätten, aus beliebigen einseitig linearen Grammatiken DEAs zu machen. Dazu nehmen wir einen Umweg:

NDEAs

Abhilfe: Nichtdeterministische endliche Automaten. Diese sehen aus wie DEAs, nur ist das Startsymbol jetzt eine Teilmenge von Φ und

$$\delta\colon \Phi \times \Sigma \to \mathcal{P}(\Phi).$$

Außerdem dürfen NDEAs natürlich auch mehrere Startzustände haben. Man kann sich vorstellen, dass ein NDEA systematsich alle Möglichkeiten durchprobiert, wenn er mehrere mögliche Folgezustände hat. Meist ist aber einfacher, sich vorzustellen, der NDEA habe die Gabe des Hellsehens und er könne mit traumwandlerischer Sicherheit jeweils den Übergang nehmen, der schließlich in einen Endzustand führt (so es diesen gibt) oder es gleich aufgibt (wenn nicht). Manchmal konstruiert man für so etwas auch ein so genanntes Orakel, das der NDEA in Ermangelung eigener transzendenter Fähigkeiten zu Rate ziehen kann.

Diese Orakel gibt es natürlich nicht wirklich, und so muss man, wenn man einen NDEA simuliert, eben doch alle Möglichkeiten durchprobieren. Das ist unter Umständen viel verlangt – es gibt potenziell eine gewaltige Menge möglicher Zustandsfolgen. In realen Computern ist die Auswertung von NDEAs daher nicht ganz einfach (mit Quantencomputern könnte sich das ändern, für sie sind NDEAs quasi wie geschaffen, aber wir haben noch keine wirklich brauchbaren Quantencomputer).

Welche Sprache akzeptieren nun NDEAs? Das lässt sich recht analog zur Entwicklung bei DEAs beantworten, also durch Bildung einer erweiterten Übergangsfunktion δ^*, nur, dass wir dieses Mal damit rechnen müssen, dass wir nicht einen, sondern mehrere Zustände zurückbekommen.

Definieren wir also

$$\delta^*(T, \epsilon) = T \quad \text{für alle } T \subseteq \Phi$$
$$\delta^*(T, wx) = \bigcup_{Z \in \delta^*(T,w)} \delta(Z, x).$$

Mit anderen Worten berechnen wir die Menge der Zustände in denen sich der NDEA nach dem Lesen von wx befinden kann dadurch, dass wir erst alle Zustände berechnen in denen er nach dem Lesen von w stehen kann und dann für jeden dieser Zustände sehen, in welche er beim Lesen von x von dort aus gehen kann. Die Ergebnisse $\delta(Z, x)$ sind Mengen (wir haben ja einen NDEA), und die vereinigen wir, fertig. Wir definieren dann

$$L(A) = \{w \in \Sigma^* \mid \delta^*(S, w) \cap F \neq \emptyset\}.$$

Verglichen mit der entsprechenden Definition für DEAs hat sich nur die Entscheidung geändert, ob wir akzeptieren wollen oder nicht – haben wir damals geprüft, ob *der* Zustand am Ende der Eingabe in der Menge der Endzustände ist, müssen wir jetzt sehen, ob *einer* der potenziell vielen Zustände, in denen wir uns am Ende der Eingabe befinden können, ein Endzustand ist – ob also die Menge der Endzustände F mit der Menge $\delta^*(S, w)$ der möglichen Zustände nach dem Lesen von w mindestens ein gemeinsames Element hat.

Erstaunlicherweise gilt der folgende Satz von Rabin und Scott:

NDEAs und DEAs sind äquivalent.

Beweisidee: $\Phi_{\text{DEA}} = \mathcal{P}(\Phi_{\text{NDEA}})$

Formal wollen wir beweisen, dass es zu jeder durch einen NDEA $A = \langle \Phi, \Sigma, \delta, S, F \rangle$ (Vorsicht: S ist hier eine Menge!) akzeptierten Sprache L auch einen DEA A' gibt, der gerade L akzeptiert. Dieser DEA wird nach Beweisidee die folgende Form haben:

$$A' = \langle \mathcal{P}(\Phi), \Sigma, \delta', S', F' \rangle$$

Was muss nun δ' sein? Wir wollen unseren NDEA durch den DEA simulieren, indem wir uns nicht gleich entscheiden, in welchen Zustand wir übergehen, sondern erstmal in mehreren NDEA-Zuständen gleichzeitig sind – also in einer Teilmenge von Φ. Deswegen wollen wir aus einem DEA-Zustand Z' einfach in den DEA-Zustand übergehen, der aus allen NDEA-Zuständen besteht, die wir von allen NDEA-Zuständen φ in Z' unter Lesen des augenblicklichen Zeichens x erreichen können, also

$$\delta'(Z', x) = \bigcup_{\varphi \in Z'} \delta(\varphi, x).$$

Die Teilmenge von Φ, in der sich der NDEA am Anfang befinden kann, ist gerade S, also muss der Startzustand $S' = S$ sein.

Ein NDEA akzeptiert, wenn einer der der möglichen Zustände am Ende der Eingabe in der Menge der Endzustände enthalten ist; Endzustände des DEA sind mithin alle Zustände, die mindestens ein Element aus F enthalten.

Damit haben wir den DEA konstruiert:

$$A' = \langle \mathcal{P}(\Phi), \Sigma, \delta', S, \{Z' \in \mathcal{P}(\Phi\rangle \mid Z' \cap F \neq \emptyset\})$$
$$\text{mit } \delta'(Z', x) = \bigcup_{Z \in Z'} \delta(Z, x)$$

Prüfen wir, ob $L(A) = L(A')$ gilt, wir also bei unserer Konstruktion keinen Fehler gemacht haben. Sei zunächst $w = x_1 \dots x_n \in L(A)$. Wir müssen dazu zeigen, dass die Zustandsfolge, die der DEA beim Lesen von w durchläuft, schließlich in einem Endzustand des DEA endet.

Was ist diese Zustandsfolge? Nun, interessanterweise sind die $Z_i = \delta^*(S, x_1 \dots x_{i-1})$ (wir vereinbaren $x_1 \dots x_0 = \epsilon$, um die Schreibweise zu vereinfachen), also die Werte von δ^* nach dem Lesen der ersten $i-1$ Zeichen des Wortes, gerade DEA-Zustände (nämlich Teilmengen von Φ). Könnte es nicht sein, dass $Z_1, \dots, Z_{n+1}$ die gesuchte DEA-Zustandsfolge ist?

Dem ist tatsächlich so; um das zu sehen, reicht etwas Indexzauberei:

$$\begin{aligned}\delta'(Z_i, x_i) &= \bigcup_{Z \in Z_i} \delta(Z, x_i) \\ &= \bigcup_{Z \in \delta^*(S, x_1 \dots x_{i-1})} \delta(Z, x_i) \\ &= Z_{i+1}\end{aligned}$$

(prüft das mit den Formeln für δ', δ^* und Z_i!)

Um sicher zu sein, dass DEA und NDEA auch an der gleichen Stelle loslaufen, vergewissern wir uns noch, dass in der Tat

$$Z_1 = \delta^*(S, \epsilon) = \{S\} = S'.$$

Schließlich müssen wir noch sehen, dass der DEA das Wort auch akzeptiert. Das tut er aber, weil

$$\emptyset \neq F \cap \delta^*(S, w) = F \cap Z_{n+1}$$

– die Ungleichung gilt, weil $w \in L(A)$ ist, mit dem Gleichheitszeichen gilt dann aber nach Definition $Z_{n+1} \in F'$, also $w \in L(A')$, und das war es, was wir wissen wollten.

Als Beispiel für die Wandlung eines NDEA in einen DEA betrachten wir den Vielleicht-Automaten. Dessen Übergangsfunktion ist

$$\begin{aligned}\delta_{\mathsf{NDEA}}(S, \mathsf{vielleicht}) &= \{A, B\} \\ \delta_{\mathsf{NDEA}}(A, \mathsf{ja}) &= \{S\} \quad , \\ \delta_{\mathsf{NDEA}}(B, \mathsf{nein}) &= \{S\}\end{aligned}$$

die Potenzmenge der Zustandsmenge

$$\Phi_{\mathsf{DEA}} = \mathcal{P}(\Phi_{\mathsf{NDEA}}) = \Big\{\emptyset, \{S\}, \{A\}, \{B\}, \{A, B\}, \{A, S\}, \{B, S\}, \{A, B, S\}\Big\}$$

Der Startzustand ist einfach: $S' = \{S\}$, weil auch der NDEA nur einen Startzustand hat.

Zulässige Endzustände des DEA sind alle Mengen, die Endzustände des NDEA enthalten. Endzustand des NDEA ist wieder nur S, wir haben also insgesamt vier DEA-Endzustände.

Die Übergangsfunktion rechnen wir nur für die wirklich interessanten Fälle aus, also die Zustände, die auch wirklich erreicht werden. Das ist zunächst

$$\delta'(\{S\}, \mathsf{vielleicht}) = \delta(S, \mathsf{vielleicht}) = \{A, B\} =: A'.$$

Beachtet, wie sich δ' aus dem Wert von δ ergibt. $\delta'(\{S\}, \mathsf{ja})$ ergibt sich auf diese Weise zur leeren Menge, und analog zum nein-Fall sparen wir uns das Hinschreiben.

Wir haben jetzt aber einen neuen Zustand $A' = \{A, B\}$ und müssen δ' für ihn ausrechnen:

$$\begin{aligned}\delta'(A', \mathsf{ja}) &= \delta(A, \mathsf{ja}) \cup \delta(B, \mathsf{ja}) = \{S\} \\ \delta'(A', \mathsf{nein}) &= \delta(A, \mathsf{nein}) \cup \delta(B, \mathsf{nein}) = \{S\} \\ \delta'(A', \mathsf{vielleicht}) &= \delta(A, \mathsf{vielleicht}) \cup \delta(B, \mathsf{vielleicht}) = \emptyset\end{aligned}$$

Alle anderen Zustände sind vom Startsymbol aus nicht erreichbar und daher uninteressant; nicht erreichbare Zustände können wir natürlich auch aus der Zustandsmenge streichen, so dass wir insgesamt zwei Zustände mit insgesamt drei Übergängen haben.

In diesem speziellen Fall hat uns dieses Verfahren also einen kleineren Automaten geliefert als den oben von mir vorgeschlagenen. Das ist aber sehr untypisch. In der Regel bekommt man aus so einer Umwandlung gewaltige DEAs heraus – was auch nicht so überraschend ist, ist doch hier

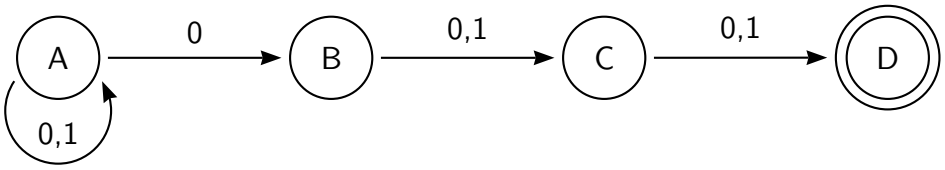
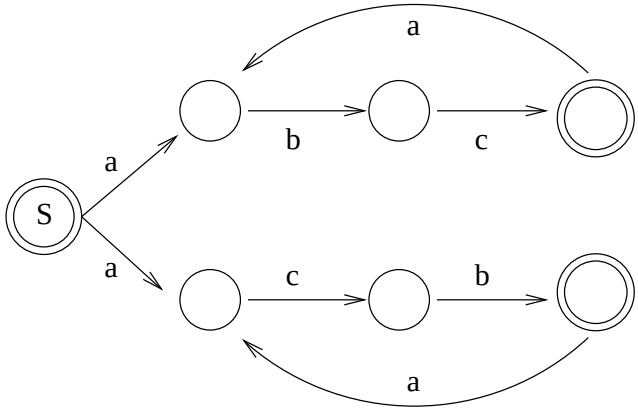


Fig. 12



. *Fig. 13*

eine Potenzmenge im Spiel. Man kann die DEAs auch etwas besser konstruieren oder aber einige bekannte Verfahren einsetzen, um die DEAs automatisch zu verkleinern.

Es ist aber keineswegs sicher, dass das viel bringt. Die Sprache aller Wörter aus $\{0,1\}^*$ mit einer 0 an der k-letzten Stelle ist z.B. mit einem NDEA mit $k+1$ Zuständen zu erkennen und damit regulär. Für $k = 3$ sieht so ein NDEA etwa so aus:

(cf. Fig. 12)

Ein DEA, der diese Sprache akzeptiert, muss aber mindestens 2^k Zustände haben (der Beweis dafür ist nicht schwierig und findet sich z.B. bei Schöning) – und das macht schon bei relativ moderaten k von vielleicht 30 dieses Problem für DEAs unzugänglich.

In gewisser Weise könnte man NDEAs also als „Konstruktionshilfen für DEAs“ bezeichnen.

NDEA für $\{abc\}^* \cup \{acb\}^*$

(cf. Fig. 13)

DEA für $\{abc\}^* \cup \{acb\}^*$

(cf. Fig. 14)

Problems

(20.1)* Gegeben sei der Automat $A = \langle \Phi, \Sigma, \delta, S, F \rangle$ mit

$$\Phi = \{ S, U, L, V, K, D\}$$
$$\Sigma = \{ \text{unsigned, long, short, int, \_, a, b, c, ;}\}$$
$$F = \{ D\}$$

(stellt euch den Unterstrich als Komma vor – ich habe das hier ausgewechselt, weil das sonst etwas konfus aussähe) und

$$\begin{array}{ccc} \delta(S, \text{unsigned}) = U & \delta(S, \text{short}) = L & \delta(S, \text{long}) = L \\ \delta(U, \text{long}) = L & \delta(U, \text{short}) = L & \delta(L, \text{int}) = V \\ \delta(V, \text{a}) = K & \delta(V, \text{b}) = K & \delta(V, \text{c}) = K \\ \delta(K, \text{\_}) = V & \delta(K, \text{;}) = D & \end{array}$$

Dieser Automat akzeptiert einige Variablendeklaration aus der Sprache C (die ihr nicht können müsst, um mit dieser Aufgabe etwas anfangen zu können).

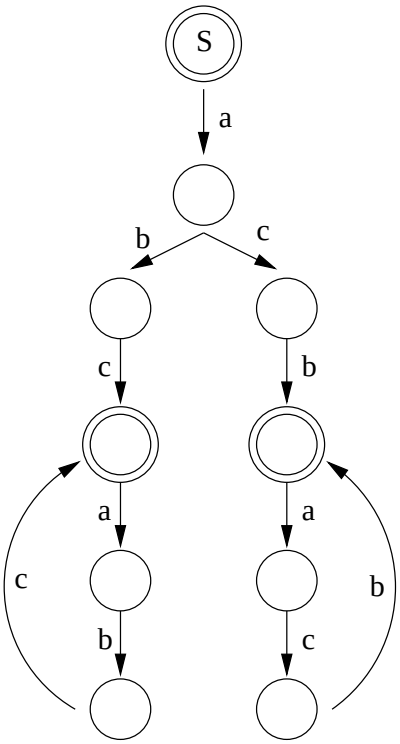


Fig. 14

Skizziert diesen Automaten und schreibt eine rechtslineare Grammatik, die die von ihm akzeptierte Sprache erzeugt. Erzeugt mit der Grammatik ein paar Wörter und schickt sie durch den Automaten, um nachzusehen, ob sie auch akzeptiert werden. **(L)**

21. Transducer

Eine *Transducer* ist ein „Automat mit Drucker“, genauer ein Tupel $\langle \Phi, \Sigma_i, \Sigma_o, \delta, \lambda, S \rangle$ mit

- Einem Zustandsalphabet Φ
- Einem Eingabealphabet Σ_i
- Einem Ausgabealphabet Σ_o
- Einer Übergangsfunktion $\delta\colon \Phi \times \Sigma_i \to \Phi$
- Einer Ausgabefunktion $\lambda\colon \Phi \times \Sigma_i \to \Sigma_o^*$
- Einem Startzustand $S \in \Phi$

Gegenüber einem DEA kommen also Σ_o und λ dazu.

Alternativ könnte man auch die Übergangsfunktion eines DEA zu einer Funktion $\delta\colon \Phi \times \Sigma_i \to \Phi \times \Sigma_o^*$ erweitern. Dann ist natürlich λ nicht mehr nötig.

Endliche Transducer sind ein Konzept, das in der Sprachverarbeitung eine wichtige Rolle spielt – klassisch ist etwa die Two-Level-Morphology, eine der verbreitetsten Methoden für die Computermorphologie.

Man kann endliche Transducer auch auffassen als einen Automaten, der auf Paaren von Eingabesymbolen läuft, der sozusagen von zwei Eingabebändern liest, und zwar so, dass der äquivalente „Standard-Transducer“ (bei vielen Autoren ist Standard und Variante dabei gegenüber unserer Definition vertauscht) bei Eingabe des einen Bandes gerade das andere als Ausgabe erzeugt. Der so entstandene DEA akzeptiert eine Folge von Paaren von Eingabesymbolen genau dann, wenn die Folge der ersten Elemente der Paare den Transducer zur Ausgabe der zweiten Elemente

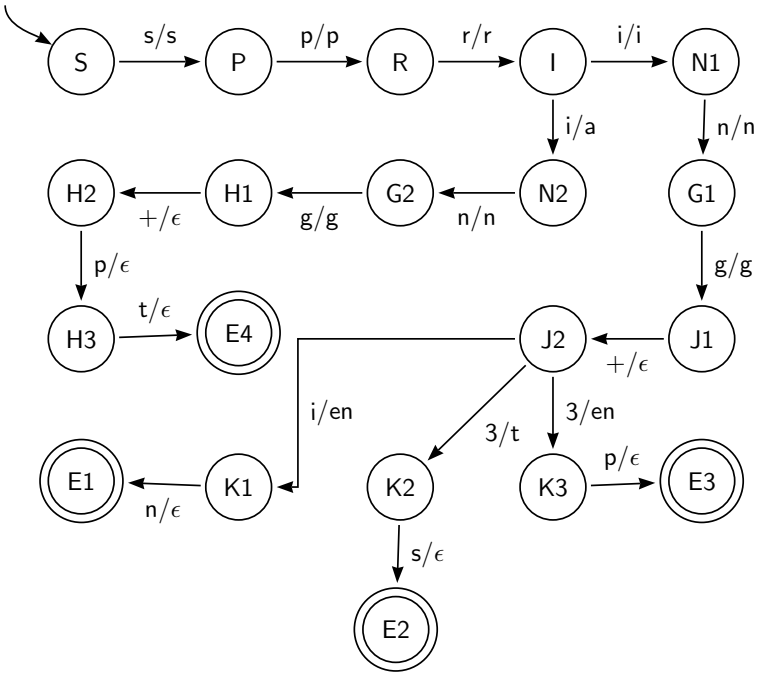


Fig. 15

bringen würde. In diesem Sinn kann man Transducer mit endlichen Automaten simulieren, woraus auch folgt, dass Ein- und Ausgaben endlicher Transducer *zusammen* eine reguläre Sprache bilden.

Ein Grund für die breite Anwendung von Transducern liegt in den Abschlusseigenschaften der regulären Sprachen: Transducer lassen sich sehr einfach verketten, vereinigen, komponieren usf. Die fsmtools[1] bieten einen fertig durchprogrammierten Werkzeugkasten dazu.

Die Paradeanwendung von Transducern in der Computerlinguistik ist die Morphologie, in der Transducer etwa zur Produktion von Wortformen dienen. Hier ein Beispiel, wie so ein Transducer aussehen könnte:

(cf. Fig. 15)

Dieser Transducer kann aus „Tiefenstrukturen“ wie spring+3p („springen in der 3. Person Plural“) oder spring+pt („springen in 3. Person Singular Präteritum“) „Oberflächenstrukturen“ wie „springen“ oder „sprang“ machen. Es ist schon aus diesem beliebig vereinfachten Beispiel klar, dass solche Transducer nur maschinell generierbar sind. Eingebeformalismen für diese Generierung gibt es etliche – mit der bekannteste wurde durch das Urkimmo-System definiert. Bei dieser Generierung wird gleich noch dafür gesorgt, dass die resultierenden Transducer deterministisch sind und nicht wie hier nichtdeterministisch.

Ähnlich lassen sich Transducer definieren, die aus einem Text nach gewissen Regeln – letztlich natürlich regulären Ausdrücken – bestimmte „interessante“ Konstrukte herausfischen, beispielsweise Eigennamen. Dabei operieren die Transducer dann meistens nicht mehr auf einzelne Zeichen, sondern auf morphosyntaktische Merkmale der einzelnen Wörter (meist so genannte Tags, die zu jedem Wort eine irgendwie definierte Wortklasse angeben).

Ein weiteres Beispiel ist die Addition im Binärsystem. Wir stellen Zahlen dar als $\ldots b_2 b_1 b_0$ mit $b_i \in \{0, 1\}$ und $z = \sum b_i 2^i$ (binäre oder 2-adische Darstellung). Addition in Binärdarstellung einfach:

$$0 + 0 = 0 \quad 0 + 1 = 1 + 0 = 1 \quad 1 + 1 = 0 + \text{Carry}$$

[1] `http://www.research.att.com/sw/tools/fsm/`

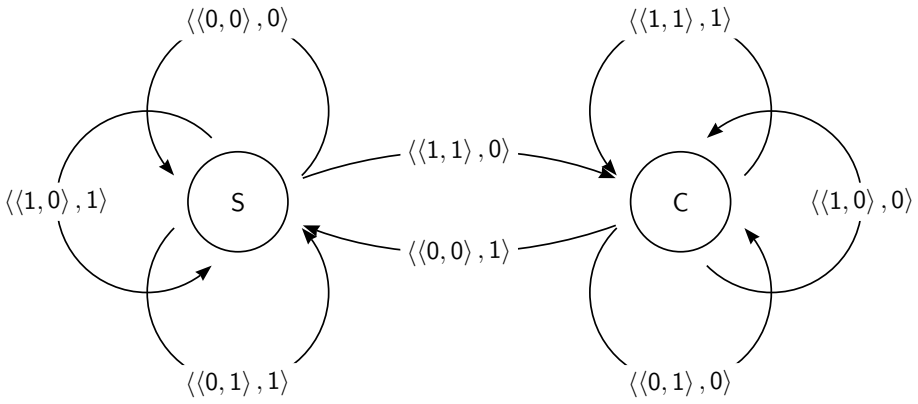


Fig. 16

Carry heißt so viel wie Übertrag – es entspricht dem „Merke eins“ aus der Grundschuladdition. Wir müssen also noch jeweils sagen, was passiert, wenn wir einen Übertrag haben. Auch das ergibt sich sehr einfach:

$$0 + 0 = 1 \quad 0 + 1 = 1 + 0 = 0 + \text{Carry} \quad 1 + 1 = 1 + \text{Carry}.$$

Wir brauchen einen Automaten, der ein 2-Tupel von Bits $\left\langle b_{1,i}, b_{2,i} \right\rangle$ liest und je nach Tupel entweder 0 oder 1 ausgibt und zusätzlich ein Carry verwalten kann.

(cf. Fig. 16)

Eine Beispielberechnung: $35 = 14 + 21 = 01110_2 + 10101_2$. Die Maschine erwartet die Eingabe „Least significant bit first“, also rückwärts. Das Eingabewort ist also

$$\langle 0, 1\rangle \langle 1, 0\rangle \langle 1, 1\rangle \langle 1, 0\rangle \langle 0, 1\rangle \langle 0, 0\rangle$$

Dargestellt in Tripeln $\left\langle \text{Zustand}, \text{Eingabe}, \text{Ausgabe} \right\rangle$ läuft die Maschine so:

$\left\langle S, \langle 0, 1\rangle \ldots, \epsilon \right\rangle$
$\rightarrow \quad \left\langle S, \langle 1, 0\rangle \ldots, 1 \right\rangle$
$\rightarrow \quad \left\langle S, \langle 1, 1\rangle \ldots, 11 \right\rangle$
$\rightarrow \quad \left\langle C, \langle 1, 0\rangle \ldots, 110 \right\rangle$
$\rightarrow \quad \left\langle C, \langle 0, 1\rangle \ldots, 1100 \right\rangle$
$\rightarrow \quad \left\langle C, \langle 0, 0\rangle, 11000 \right\rangle$
$\rightarrow \quad \left\langle S, \epsilon, 110001 \right\rangle$

In der Tat: $100011_2 = 35$

22. Reguläre Sprachen II

Satz: Ist L eine reguläre Sprache, so gibt es einen endlichen Automaten A, der L akzeptiert.

Beweis: Wieder konstruktiv über die fünf Eigenschaften regulärer Sprachen.

(1) $\emptyset$ wird von allen Automaten ohne Endzustände erzeugt

(2) Einzelne Zeichen aus dem Alphabet gehen mit

$$\begin{aligned}\delta(S,x) &= \begin{cases} B & \text{wenn } x = a_i \\ C & \text{sonst} \end{cases} \\ \delta(B,x) &= C \\ \delta(C,x) &= C,\end{aligned}$$

wenn nur B ein Endzustand ist – d.h., das gesuchte Zeichen treibt den Automaten in den Endzustand, jedes andere oder ein weiteres in einen nicht-akzeptierenden.

(3) Vereinigung: Φ_1 und Φ_2 disambiguieren wie gehabt, einen neuen Startzustand machen, der als Folgezustände die Folgezustände der alten Startzustände hat. Das geht, weil nicht zwischen Φ_1 und Φ_2 gesprungen wird. Der neue Startzustand muss in die neuen Endzustände aufgenommen werden, wenn einer der alten Startzustände Endzustand war.

(4) Verkettung: Startzustand von A_1 wird neuer Startzustand, alle Endzustände von A_1 werden mit den Startzuständen von A_2 vereinigt (und nicht in F aufgenommen, es sei denn Startzustand von A_2 ist in F_2), Endzustände sind F_2

(5) Sternbildung: Startzustand wird zu Endzustand (wg. ϵ). Dann alle Endzustände wieder auf Startzustand verweisen lassen.

Diesen Satz hätten wir eigentlich gar nicht mehr beweisen müssen. Aus der Äquivalenz von einseitig linearen Grammatiken und regulären Ausdrücken auf der einen und der zwischen einseitig linearen Grammatiken DEAs auf der anderen Seite folgt nämlich sogar die Äquvialenz von DEAs und regulären Ausdrücken.

Abschlusseigenschaften

Da Sprachen zunächst nur Mengen sind, kann man durch die üblichen Mengenoperationen neue Sprachen erzeugen – es wäre beispielsweise toll, wenn man eine Grammatik für Hauptsätze hätte und eine für Relativsätze und durch Vereinigung eine Grammatik bekäme, die beides abdeckt. Die Frage ist aber: gibt es für die – in diesem Fall – vereinigte Menge immer noch eine – beispielsweise – reguläre Grammatik?

Seien L_1, L_2 zwei reguläre Sprachen. Dann sind $L_1 \bullet L_2$, $L_1 \cup L_2$ und L_i^* trivial regulär.

$\Sigma^* \setminus L = \overline{L}$ ist regulär: Es existiert ein Automat $A = \langle \Phi, \Sigma, \delta, S, F \rangle$, der L akezptiert. Der Automat $A' = \langle \Phi, \Sigma, \delta, S, \Phi \setminus F \rangle$ akzeptiert genau alle Wörter nicht, die A nach F treiben und endet für alle anderen in einem Endzustand, akzeptiert also $\overline{L}$. Daher muss $\overline{L}$ regulär sein

$L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$ (deMorgan), Regularität ist aber unter Vereinigung und Komplementbildung erhalten.

$L_1 \setminus L_2 = L_1 \cap \overline{L_2}$, also auch regulär.

Das Pumping Lemma

Satz: Zu jeder regulären Sprache L existiert eine Zahl n, so dass sich alle $w \in L$ mit $|w| \geq n$ als $w = x \bullet y \bullet z$ darstellen lassen mit

- $x, y, z \in \Sigma^*$
- $|y| \geq 1$
- $|x \bullet y| \leq n$
- $x \bullet y^i \bullet z \in L$.

Beweis: Der entscheidende Punkt ist die vierte Bedingung. Weil L regulär, gibt es einen Automaten A mit Zuständen T_i ($i = 1, \ldots, n$), der L akzeptiert.

Wenn es kein w mit $|w| \geq n$ gibt, ist nichts zu beweisen. Sei also $|w| \geq n$. Dann wird mindestens ein T_k mehrfach durchlaufen. Wir zerlegen w, so dass

$$\begin{aligned}\delta^*(S, x) &= T_k \\ \delta^*(T_k, y) &= T_k \\ \delta^*(T_k, z) &\in F.\end{aligned}$$

Durch triviale Induktion folgt damit, dass auch $x \bullet y^i \bullet z$ in von A akzeptiert und damit in L ist.

In Prosa: Wörter in ausreichend komplexen regulären Sprachen lassen sich „normalerweise“ aufblasen. Deshalb lassen sich balancierte Klammern nicht mit regulären Ausdrücken prüfen:

Satz: $L = \{a^i b a^i\}$ ist nicht regulär.

Beweis: Annahme: L ist regulär. Sei i, so dass $z = a^i b a^i$ länger als n aus dem Pumping Lemma ist. Dann gibt es eine Zerlegung $z = uvw$ mit $v \neq \epsilon$. Da nach Pumping Lemma auch $uv^j w \in L$, ist kein b in v. Also muss $v = a^l$ mit $l > 0$ sein. Damit liegt auch $a^{i+l} b a^i$ (wenn b in w ist) oder $a^i b a^{i+l}$ (sonst) in L, Widerspruch.

Satz: $L = \{a^p \mid p \text{ Quadratzahl}\}$ ist nicht regulär.

Beweis: L enthält auf jeden Fall Wörter, die länger als n aus dem Pumping Lemma sind. OBdA $n > 1$. Sei nun speziell $p = n^2$. Dann ist $w = xyz$ und $|xy| < n$ sowie $|y| \geq 1$ nach Pumping Lemma. Außerdem ist $xy^2z \in L$ (Vorsicht: Hier haben wir zwei Bedeutungen der Potenz: In n^2 bedeutet es das gewohnte Quadrat, in y^2 die Verkettung).

Jetzt ist

$$\begin{aligned}n^2 &= |xyz| && \text{Nach Konstruktion} \\ &< |xy^2z| && \text{Weil } |y^2| > |y| \text{ wegen } |y| \geq 1 \\ &= |xyz| + |y| \\ &\leq n^2 + n && \text{wegen } |xy| \leq n, \text{ insbesondere } |y| \leq n \\ &< n^2 + 2n + 1 \\ &= (n+1)^2.\end{aligned}$$

Damit liegt $|xy^2z|$ zwischen zwei aufeinanderfolgenden Quadratzahlen, kann also selbst keine Quadratzahl sein.

Problems

(22.1)* Das Wort amtmta über dem Alphabet $\{a, m, t\}$ möge von einem Automaten akzeptiert werden können, der drei Zustände hat. Schließt daraus, dass die von dem Automaten erzeugte Sprache unendlich ist. **(L)**

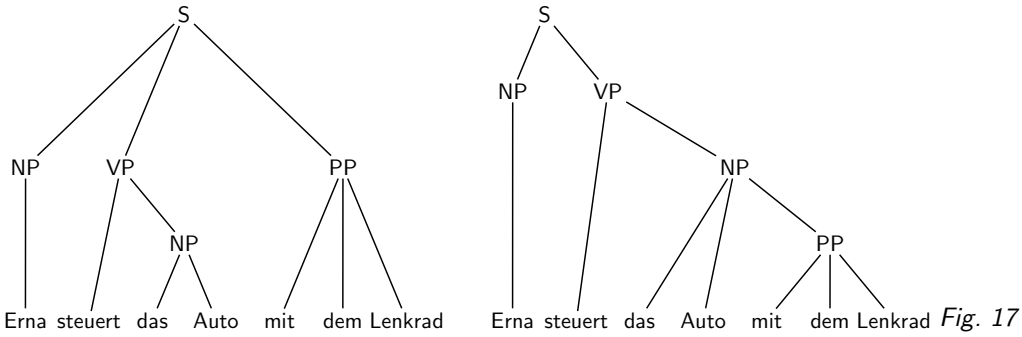


Fig. 17

kontextfrei

Linksableitung

ambig

23. Kontextfreie Grammatiken

Eine Grammatik $G = \langle \Phi, \Sigma, R, S \rangle$ heißt *kontextfrei* oder Chomsky-2-Grammatik, wenn jedes Element von R die Form $A \to \alpha$ hat mit $A \in \Phi$ und $\alpha \in (\Sigma \cup \Phi)^*$.

Kontextfreie Grammatiken schließen trivial reguläre Grammatiken ein, sind aber eine echte Obermenge.

$$G = \langle \{S\}, \{a, b\}, \{S \to b, S \to aSa\}, S \rangle$$

erzeugt z.B. $a^i b a^i$.

Ableitungen in kontextfreien Grammatiken sind nicht eindeutig. Sei

$$G = \langle \{S, A, B\}, \{a, b\}, \{S \to A\,B,\ A \to a,\\ A \to aa,\ B \to b,\ B \to ab\}, S \rangle.$$

aab kann abgeleitet werden:

(1) $S \to AB \to aaB \to aab$
(2) $S \to AB \to Ab \to aab$
(3) $S \to AB \to aB \to aab$

Zwei Phänomene: (1) und (2) sind verschieden abgeleitet, führen aber auf gleiche Bäume, (1) und (3) führen auf verschiedene Bäume.

Lösung für (1) vs. (2): Linksableitungen. Eine *Linksableitung* ist eine Ableitung, bei der immer das am weitesten links stehende Nichtterminal ersetzt wird. Offenbar gibt es zu jeder Ableitung eine zugehörige Linksableitung, wir verlieren also nichts, wenn wir diese Einschränkung machen.

Analoge Probleme können in einseitig linearen Grammatiken nicht auftreten, weil in den Wörtern, die während einer Ableitung auftreten, per definitionem immer nur ein Nichtterminal steht (und zwar entweder ganz links oder ganz rechts).

Linksableitungen und Ableitungsbäume entsprechen sich eindeutig. Dass eine Ableitung eindeutig einen Baum bestimmt, hatten wir bereits ganz am Anfang gesehen. Um umgekehrt aus einem Ableitungsbaum eine Linksableitung zu erzeugen, fängt man bei der Wurzel an und wertet jeweils den am weitesten links oben stehenden noch unbearbeiteten Knoten aus.

(1) vs. (3) ist nicht „lösbar“, die Zweideutigkeit liegt hier in der Natur kontextfreier Grammatiken.

Eine kontextfreie Grammatik heißt *ambig*, wenn es für mindestens ein $w \in L(G)$ mehrere Linksableitungen $S \overset{*}{\to} w$ gibt, eindeutig sonst. Eine kontextfreie Sprache heißt ambig, wenn jede sie erzeugende kontextfreie Grammatik ambig ist.

Ambiguität ist eigentlich nicht schlecht – sie taucht in der natürlichen Sprache überall auf:

(cf. Fig. 17)

Hier wurde der Satz „Erna steuert das Auto mit dem Lenkrad“ mit einer hypothetischen Gammatik des Deutschen geparst. Dabei sind zwei Lesungen denkbar – entweder, Erna verwendet das Lenkrad, um das Auto zu steuern, oder das Auto, das sie fährt, hat ein Lenkrad und nicht etwa einen Joystick oder einen Steuerknüppel oder etwas ähnliches.

Ambiguitäten dieser Art sind typisch für die natürliche Sprache. Dass wir Menschen meistens doch recht gut wissen, was gemeint ist, liegt vor allem an semantischen Informationen, zumal in Sprachen mit deutlich ärmerer Morphologie. Eine gute Demontration dieses Effekts ist das Satzpaar „time flies like an arrow“ und „fruit flies like a banana“ (wers nicht sieht: „fruit flies“ sind hier Fruchtfliegen). Ohne Wissen, was Zeit, Pfeile, Fruchtfliegen und Bananen sind, ist ein Parser hier aufgeschmissen.

schwach äquivalent

Tatsächlich sind Ambiguitäten in „klassischen“ Phrasenstrukturgrammatiken (also Grammatiken, die so beschrieben sind, wie wir das hier machen) ein riesiges Problem, denn es ist nicht ungewöhnlich, dass Sätze mit ein paar dutzend Wörtern ohne weitere Information auf Millionen von Bäumen führen. Unter anderem diese Feststellung hat zur Einsicht geführt, dass ohne Anleihen aus den stark lexikalisierten Dependenzgrammatiken bei der Verarbeitung natürlicher Sprache kaum etwas zu gewinnen ist. Aber das ist ein Thema für die Syntax-Veranstaltung.

Problems

(23.1)* Ambiguitäten lassen sich auch bei Komposita, die ja quasi wie kleine Sätze gebaut werden, beobachten. Seht euch beispielsweise folgende Grammatik an:

$$B \to \textit{fahrer} \mid SB$$
$$S \to QQ \mid QS \mid SQ$$
$$Q \to \textit{last} \mid \textit{personen} \mid \textit{kraft} \mid \textit{pferde}$$

Dabei sind Nichtterminale groß-, Terminale klein geschrieben. Der Kürze halber habe ich $A \to B$, $A \to C$ als $A \to B \mid C$ geschrieben. Startsymbol soll B (und nicht S!) sein.

Schreibt ein paar Linksableitungen für „lastkraftwagenfahrer“ auf, malt die zugehörigen Ableitungsbäume. Welche Ableitung scheint euch die „richtige“ zu sein? Wie würde sowas für reguläre Grammatiken aussehen? **(L)**

24. Chomsky-Normalform

Eine Grammatik heißt *1-frei*, wenn sie keine Regeln der Form $A \to B$ ($A, B \in \Phi$ – *1-Regeln*) enthält.

Satz: Sei G eine kontextfreie Grammatik. Dann gibt es eine 1-freie Grammatik G', so dass $L(G) = L(G')$ (die Grammatiken sind *schwach äquivalent*).

Beweis: Die Grundidee dieser Umformung ist: Wenn ich zwei Regeln $A \to B$ und $B \to x$ habe, führe ich eine neue Regel $A \to x$ ein und brauche dann kein $A \to B$.

Im Allgemeinen allerdings muss man etwas genauer aufpassen. Das erste Problem sind Zyklen: Damit ist gemeint, dass durchaus $A \to B$ und $B \to A$ gemeinsam in einer Regelmenge vorkommen können. Ein naives Ersetzen würde damit in eine Endlosschleife führen. Die Lösung ist einfach: Alle Nichtterminale in einem Zyklus sind äquivalent und können (in allen Regeln!) durch ein einziges Nichtterminalsymbol ersetzt werden.

Die restlichen 1-Regeln lassen sich nach dem Algorithmus

Solange noch 1-Regeln in der Regelmenge enthalten sind:

Wähle eine 1-Regel r (sagen wir $A \to B$), so dass es keine 1-Regel mit B auf der linken Seite gibt:

Für alle Regeln $B \to \alpha$:

Führe eine neue Regel $A \to \alpha$ ein

Entferne r.

entfernen.

Die Forderung, B solle in keiner 1-Regel auf der linken Seite vorkommen, verhindert, dass bei diesem Verfahren neue 1-Regeln entstehen. Dass diese Forderung immer für mindestens eine 1-Regel erfüllbar ist, liegt daran, dass wir Zyklen schon im Vorfeld eliminiert haben.

Um zu sehen, wie das geht, können wir die Regelmenge

ϵ-frei

$$S \to A \qquad A \to B \qquad B \to xy$$

mit zwei 1-Regeln betrachten. Würden wir im ersten Schritt $S \to A$ eliminieren, kämen wir auf

$$S \to B \qquad A \to B \qquad B \to xy$$

(überzeugt euch, dass die Regeln in der Tat schwach äquivalent sind). Das sieht nicht nach Fortschritt aus. Wenn ihr das Verfahren weiter durchführt, kommt ihr schließlich auch zu einem Ende, aber es ist günstiger, bei $A \to B$ anzufangen, eben weil es keine 1-Regel mit B auf der linken Seite gibt:

$$S \to A \qquad A \to xy \qquad B \to xy$$

– eine 1-Regel weniger, und jetzt hilft es auch, $S \to A$ zu eliminieren:

$$S \to xy \qquad A \to xy \qquad B \to xy.$$

Natürlich braucht es jetzt die Regeln mit A und B auf der linken Seite nicht mehr. Aber dazu später.

Eine kontextfreie Grammatik heißt ϵ-*frei*, wenn sie keine Regeln der Form $A \to \epsilon$ enthält ($\Rightarrow$ Wörter in Ableitungen werden nie kürzer).

Satz: Wenn $\epsilon \notin L(G)$, gibt es eine zu G schwach äquivalente ϵ-freie Grammatik.

Beweis: Zunächst alle $A_i \in \Phi$ mit $A_i \overset{*}{\to} \epsilon$ in V sammeln (zuerst alle A_i mit $A_i \to \epsilon$, dann alle A_i mit $A_i \to A_{k_1} \cdots A_{k_n}$, wo alle $A_{k_j} \in V$). Dann alle $A_i \to \epsilon$-Regeln entfernen und für jede Regel $B \to xA_iy$ mit $xy \neq \epsilon$ und $A_i \in V$ eine Regel $B \to xy$ hinzufügen.

Eine Grammatik ist in *Chomsky-Normalform* (CNF), wenn alle Regeln eine der Formen

- $A \to a$
- $A \to BC$

mit $A, B, C \in \Phi$ und $a \in \Sigma$ haben. Schon der Begriff der Normalform baut auf der Tatsache auf, dass es mehrere Grammatiken zu einer gegeben Sprache geben kann. Normalformen können zur Vereinfachung von Beweisen dienen, aber auch zur Implementation effizienter Parser.

Satz: Sei G eine kontextfreie Grammatik mit $\epsilon \notin L(G)$. Dann gibt es eine kontextfreie Grammatik G' in CNF mit $L(G) = L(G')$.

Beweis: oBdA G ϵ-frei und 1-frei. Wir konstruieren $G' = \langle \Phi', \Sigma', R', S' \rangle$.

1. Regeln $A \to a$ werden nach R' übernommen
2. Für alle Terminalsymbole x_i auf der rechten Seite, die nicht alleine stehen, werden Regeln $C_{x_i} \to x_i$ eingeführt und die x_i in den Regeln durch C_{x_i} ersetzt.
3. Die restlichen Regeln haben die Form $A \to Y_1 \ldots Y_n$. Wenn $n \neq 2$, werden weitere Nichtterminale Z_i eingeführt, so dass $A \to Y_1 Z_1$ bis $Z_{n-2} \to Y_{n-1} Y_n$.

Nach der Umwandlung in CNF kann die Grammatik in der Regel noch „aufgeräumt“ werden.

Dazu werden Symbole, die vom Startsymbol aus nicht in einer Ableitung zu einem nur aus Terminalen bestehenden Wort führen, aus Φ und Terminale, die nicht erzeugt werden können, aus Σ entfernt.

Man kann das algorithmisch formulieren, wir bescheiden uns mit Intuition – Nichtterminale, die nie auf der linken Seite einer Regel auftreten, können offenbar nie aus Wörtern verschwinden; solche Wörter können, da sie ein Element von Φ enthalten, dann auch nie Wörter aus $L(G) \subseteq \Sigma^*$ sein. Demnach darf man alle Regeln, die sie erwähnen, entfernen. Darurch können natürlich weitere Nichtterminale nicht mehr auf der linken Seite auftauchen. Kommen zwei Terminale nur mit jeweils identischen rechten Seiten vor, können sie verschmolzen werden.

Ein Beispiel: Vorgelegt sei die Regelmenge

A	$\to$	aCD	B	$\to$	bCD	C	$\to$	D
C	$\to$	ϵ	D	$\to$	C	S	$\to$	ABC

Wir wollen sie in CNF umformen, machen sie also ϵ-frei. Zunächst ist C in der Menge der auf ϵ führenden Symbole, wegen $D \to C$ aber auch D. Wir ergänzen die Regelmenge und werfen die ϵ-Regel raus:

Kellerautomat

$A \to a$ $A \to aC$ $A \to aCD$ $A \to aD$
$B \to b$ $B \to bC$ $B \to bCD$ $B \to bD$
$C \to D$ $D \to C$ $S \to AB$ $S \to ABC$

Im nächsten Schritt eliminieren wir Zyklen. Davon gibt es hier nur einen, nämlich zwischen C und D. Wir vereinigen die beiden Symbole zu einem neuen Symbol E (was hier nur der Klarheit halber geschieht – natürlich hätten wir auch einfach alle D durch C ersetzen können):

$A \to a$ $A \to aE$ $A \to aEE$ $B \to b$
$B \to bE$ $B \to bEE$ $S \to AB$ $S \to ABE$

Jetzt haben wir keine 1-Regeln mehr, können also mit der Prozedur aus dem Beweis anfangen. Wir haben zwei Terminale, die mit Nichtterminalen zusammenstehen, nämlich a und b. Glücklicherweise brauchen wir dafür keine neuen Nichtterminale einzuführen, weil bereits A und B diese beiden erzeugen:

$A \to a$ $A \to AE$ $A \to AEE$ $B \to b$
$B \to BE$ $B \to BEE$ $S \to AB$ $S \to ABE$

Wir haben jetzt noch drei Regeln mit drei Nichtterminalen auf der rechten Seite, alle anderen Regeln sehen schon aus wie in CNF gewünscht. Wir führen ein Nichtterminal für EE ein, für AB haben wir schon eins:

$A \to a$ $A \to AE$ $A \to AF$ $B \to b$ $F \to EE$
$B \to BE$ $B \to BF$ $S \to AB$ $S \to SE$

Das ist jetzt schon in CNF, aber komplizierter, als es sein muss, wie deutlich wird, wenn wir nicht verwendete Symbole rauswerfen. Insbesondere wird E nicht verwendet, weil es nie auf der linken Seite einer Regel vorkommt. Damit sind auch alle Regeln, die E enthalten, überflüssig, weil sie am Ende immer auf ein Wort mit E führen würden, also nie ganz Terminal werden könnten; da die Regel $F \to EE$ wegfällt, gilt dies auch für F. Unsere Anfangsgrammatik in CNF und reduziert ist also platterdings:

$A \to a$ $B \to b$ $S \to AB$

Also kleine Anmerkung am Rande: Diese Sprache ist endlich und damit sogar regulär. In der Tat reicht $S \to ab$ als Regelmenge aus (aber das wäre natürlich nicht mehr in CNF).

25. Kellerautomaten

Auch bei kontextfreien Grammatiken würden wir gerne das Wortproblem lösen. Endliche Automaten reichen dazu aber garantiert nicht aus, weil sie genau die regulären Sprachen erkennen, es aber kontextfreie Sprachen gibt, die nicht regulär sind.

Die Lösung des Wortproblems für kontextfreie Sprachen führt ziemlich schnell zu den klassischen Parsern, die neben der Information, ob ein Wort Element einer Sprache ist oder nicht auch noch z.B. Parsebäume erzeugen. Das ist eine Theorie für sich, die z.B. in unserer Parsing-Veranstaltung oder auch in Compilerbau-Vorlesungen der Informatik ausgebreitet wird.

Man braucht aber keinen „vollständigen“ Computer, um das Wortproblem für kontextfreie Sprachen zu lösen – tatsächlich reicht schon eine eher kleine Erweiterung der DEAs.

Ein (nichtdeterministischer) *Kellerautomat* ist ein Tupel $K = \langle \Phi, \Sigma, \Delta, \diamondsuit, \delta, S, F \rangle$ aus

- Einem Alphabet Φ von Zuständen
- Einem zu Φ disjunktem Alphabet Σ von Eingabesymbolen
- Einem zu Φ disjunktem Alphabet Δ von Kellersymbolen
- Einem Kelleranfangssymbol $\diamondsuit$
- Einer Übergangsfunktion $\delta\colon \Phi \times \Sigma \times (\Delta \cup \{\diamondsuit\}) \to \mathcal{P}(\Phi \times \Delta^*)$
- Einem Startzustand $S \in \Phi$

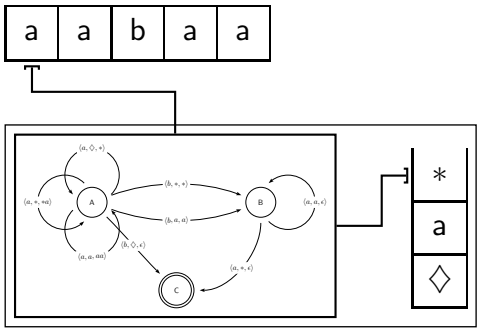


Fig. 18

- Einer Menge von Endzuständen $F \subseteq \Phi$.

Ein Übergang in einem Kellerautomat wird als

$$\langle S, x, p \rangle \rightarrow \{\langle S_i, w_i \rangle\}$$

notiert und bedeutet: Wenn im Zustand S das nächste Zeichen x ist und p als oberstes Element am Keller (besser: Stapel) liegt, gehe in einen der Zustände S_i und ersetze das oberste Element des Kellers durch w_i.

Eine Konfiguration $\langle T, v, p \rangle$ besteht aus

- Einem Zustand $T \in \Phi$ (dem augenblicklichen Zustand)
- Einem Wort $v \in \Sigma^*$ (dem noch nicht gelesenen Teil des Eingabeworts)
- Einem Wort $p \in \Delta^* \bullet \{\diamondsuit\}$ (dem augenblicklichen Kellerinhalt)

Die Startkonfiguration eines Kellerautomaten hat immer die Form $\langle S, w, \diamondsuit \rangle$, eine Endkonfiguration $\langle T, \epsilon, z \rangle$ mit $z \in \Delta^* \bullet \{\diamondsuit\}$ und $T \in F$.

Eine äquivalente Definition verzichtet auf Endzustände und definiert Akzeptanz durch den Kellerzustand am Wortende (z.B. Keller leer).

(cf. Fig. 18)

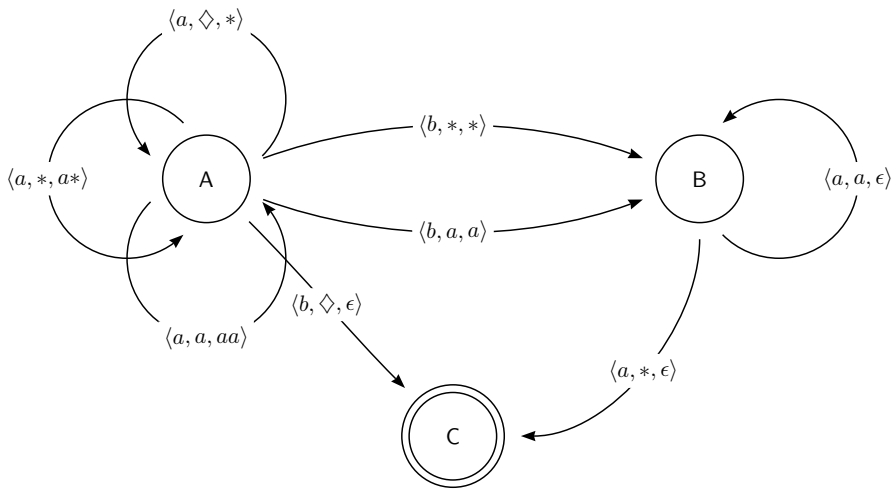


Fig. 19

26. Kellerautomaten und CFGs

Satz: Zu jeder CFG G gibt es einen Kellerautomaten, der $L(G)$ akzeptiert, und zu jedem Kellerautomaten A gibt es eine CFG, die $L(A)$ erzeugt.

Der Beweis ist aufwändig. Zunächst Beispiel: $\{a^i b a^i\}$ wird akzeptiert von folgendem Kellerautomaten:

(cf. Fig. 19)

Beispielableitungen: Wir leiten b ab:

$$\langle A, b, \diamondsuit \rangle \quad \to \quad \langle C, \epsilon, \epsilon \rangle$$

Wir leiten $aabaa$ ab:

$$\begin{aligned} \langle A, aabaa, \diamondsuit \rangle &\to \langle A, abaa, * \rangle \\ &\to \langle A, baa, a* \rangle \\ &\to \langle B, aa, a* \rangle \\ &\to \langle B, a, * \rangle \\ &\to \langle C, \epsilon, \epsilon \rangle \end{aligned}$$

$abaa$ wird zurückgewiesen:

$$\begin{aligned} \langle A, abaa, \diamondsuit \rangle &\to \langle A, baa, * \rangle \\ &\to \langle B, aa, * \rangle \\ &\to \langle C, a, \epsilon \rangle \end{aligned}$$

– kein gültiger Endzustand, da restliche Eingabe $\neq \epsilon$.

27. Kellerautomaten aus CFGs

Die Wandlung von CFGs in Kellerautomaten wird einfacher, wenn wir auf Endzustände verzichten und Aktzeptanz durch leeren Keller definieren. Außerdem lassen wir „spontane“ Übergänge zu (formal: Lesen von ϵ). Man müsste an dieser Stelle nachweisen, dass sich diese Sorte Kellerautomat in Kellerautomaten nach der ursprünglichen Definition umformen lässt und umgekehrt jeder „klassische“ Kellerautomat einen äquivalenten Kellerautomaten dieses Typs hat – da die entsprechende Transformation aber recht haarig ist, wollen wir das einfach glauben.

Sei $G = \langle \Phi, \Sigma, R, S \rangle$ kontextfrei. Der zugehörige Kellerautomat wird charakterisiert durch

- Das Kelleralphabet $\Delta = \Phi \cup \Sigma$,
- Das Kelleranfangssymbol S (d.h. das Startsymbol der Grammatik)
- Die Zustandsmenge $\Phi_M = \{z\}$ (d.h. es gibt nur einen Zustand)
- Die Übergangsfunktion δ, wo für jede Regel $A \to \alpha$ aus R ein Übergang $\langle z, \epsilon, A \rangle \to \langle z, \alpha \rangle$ und außerdem $\langle z, a, a \rangle \to \langle z, \epsilon \rangle$ gesetzt wird.

Wir wandeln $\{S \to b, S \to aSa\}$; die Ü-Funktion ist:

$$\begin{aligned} \langle z, \epsilon, S \rangle &\to \{\langle z, b \rangle, \langle z, aSa \rangle\} \\ \langle z, a, a \rangle &\to \{\langle z, \epsilon \rangle\} \\ \langle z, b, b \rangle &\to \{\langle z, \epsilon \rangle\} \end{aligned}$$

Test:

$\langle z, aba, S \rangle \to \langle z, aba, aSa \rangle \to \langle z, ba, Sa \rangle \to$
$\langle z, ba, ba \rangle \to \langle z, a, a \rangle \to \langle z, \epsilon, \epsilon \rangle \to$ ok

$\langle z, abba, S \rangle \to \langle z, abba, aSa \rangle \to \langle z, bba, Sa \rangle \to$
$\langle z, bba, ba \rangle \to \langle z, ba, a \rangle \to$ boink

Die Idee dieses Automaten ist eigentlich sehr einfach: Wenn wir oben auf dem Keller ein Nichtterminalsymbol haben, expandieren wir es mit irgendeiner passenden Regel (unsere Kellerautomaten sind ja nichtdeterministisch) und lesen nichts, wenn wir ein Terminalsymbol haben, entfernen wir es, wenn wir gerade das passende Terminalsymbol lesen.

Man kann diese Automaten natürlich zu „klassischeren“ Kellerautomaten mit mehr Zuständen und ohne ϵ-Übergange reduzieren, aber so genau wollen wir es jetzt nicht wissen. Das Grundproblem – dass wir nämlich nichtdeterministische Kellerautomaten basteln – bleibt, es gibt kein Verfahren, das aus nichtdeterministischen Kellerautomaten deterministische macht.

Erzeugung einer Grammatik aus einem Automaten ist komplizierter. Im Wesentlichen kehrt man das Verfahren oben um und benutzt Tupel aus (Zustand-vor, Kellersymbol, Zustand-nach) als Nichtterminale der Grammatik.

Genauer bestehen die Tupel aus eine Zustand am Anfang eines Verarbeitungsschritts, dem Kellersymbol, das zu diesem Zeitpunkt ganz oben auf dem Keller liegt und dem Zustand, in dem der Automat ist, wenn der Keller kürzer wird als er am Anfang des Verarbeitungsschritts war.

Die Idee dabei ist, dass wir eine Regel $\langle z, A, z' \rangle \xrightarrow{*} x$ genau dann einführen, wenn der zugehörige Kellerautomat durch die Eingabe x ggf. über mehrere Zwischenschritte, die natürlich ihrerseits Dinge auf den Keller legen und diese dann auch wieder runternehmen müssen, von z nach z' geht und A verarbeitet hat, also das Symbol *unterhalb* von A (das ja auf das, was der Automat auf dem Weg von z nach z' tut, keinen Einfluss haben kann) oben auf dem Keller liegt.

Wir gehen weiter von unseren modifizierten Kellerautomaten aus, wir akzeptieren also auf leeren Keller und lassen ϵ-Übergänge zu.

Ohne Beschränkung der Allgemeinheit können wir annehmen, dass unser Automat nie mehr als zwei Symbole auf einmal auf den Keller schiebt – das lässt sich durch Einführen zusätzlicher Zustände immer erreichen. Wir können dann folgende Regeln schreiben (darin stehen alle z und ihre Verzierungen für Elemente aus der Zustandsmenge, alle a, b, c mit Verzierungen für

Elemente des Kelleralphabets, z_0 ist der Startzustand, T ist das Startsymbol der Grammatik, S das Kelleranfangssymbol, gemäß unserer Definition oben):

1. $T \to \langle z_0, S, z\rangle$ für alle z aus der Zustandsmenge. $\langle z_0, S, z\rangle$ ist das Symbol, aus dem alles ableitbar ist, was den Automaten dazu bringt, den Keller vom Kelleranfangssymbol aus zu leeren, d.h. eben alle Elemente der Sprache.
2. $\langle z, a', z'\rangle \to a$, wenn in der Ü-Funktion $\langle z, a, a'\rangle \to \langle z', \epsilon\rangle$ steht. Die sind einfach: Die Maschine nimmt ein a' vom Stack und liest ein a. Dazwischen kann nichts passieren, und sie muss im Zustand z' enden. Aus $\langle z, a', z'\rangle$ kann also nur a werden.
3. $\langle z, a', z'\rangle \to a\langle z_1, b, z'\rangle$ für alle $z' \in \Phi$, wenn in der Ü-Funktion $\langle z, a, a'\rangle \to \langle z_1, b\rangle$ steht. Das ist schon kitzliger: Der Automat nimmt a' vom Keller und legt b drauf, liest derweil aber a. Damit müssen wir, um zu sehen, as $\langle z, a', z'\rangle$ so werden kann, erstmal ein a haben und dann sehen, was wir alles machen können, wenn wir von z_1 irgendwo anders hingehen. Wesentlich ist aber, dass beim zweiten Schritt am Anfang das b auf dem Keller liegt, das beim Übergang $z \to z_1$ dort landet.
4. $\langle z, a', z'\rangle \to a\langle z_1, b, z_2\rangle\langle z_2, c, z'\rangle$ für alle $z', z_2 \in \Phi$, wenn in der Ü-Funktion $\langle z, a, a'\rangle \to \langle z_1, bc\rangle$ steht. Das lässt sich analog zum letzten Fall einsehen, nur müssen wir dieses Mal zwei Symbole vom Keller nehmen und haben also zwei Mal die Freiheit, das in irgendwelchen geeigneten Zuständen des Automaten zu tun.

Das sieht konfus und willkürlich aus, wäre aber eigentlich einfach, wenn man beim Nachdenken drüber nicht leicht den Überblick verlieren würde. Wer sehen will, warum das so aufgeht, kann den Beweis bei Schöning goutieren – im Wesentlichen muss man nur per Induktion zeigen, dass die Grammatik in jedem Ableitungsschritt genau das produziert, was der zugehörige Automat mit den durch die Symbole der Grammatik beschriebenen Schritten verbraucht.

Als Beispiel nehmen wir den Automaten oben:

$\langle z, \epsilon, S\rangle \to \langle z, b\rangle \qquad \langle z, \epsilon, S\rangle \to \langle z, aSa\rangle$
$\langle z, a, a\rangle \to \langle z, \epsilon\rangle \qquad \langle z, b, b\rangle \to \langle z, \epsilon\rangle$

Zunächst führen wir einen neuen Zustand ein, so dass nie mehr als zwei Symbole auf den Stack gepusht werden (die Spitze des Stacks soll wieder vorne sein):

$\langle z, \epsilon, S\rangle \to \langle z, b\rangle \qquad \langle z, \epsilon, S\rangle \to \langle y, Sa\rangle \qquad \langle y, \epsilon, S\rangle \to \langle z, aS\rangle$
$\langle z, a, a\rangle \to \langle z, \epsilon\rangle \qquad \langle z, b, b\rangle \to \langle z, \epsilon\rangle$

Sehen wir zunächst, wohin wir vom Startzustand aus kommen (Regeln vom Typ 1):

$$T \to zSz \qquad T \to zSy.$$

Dabei wollen wir im folgenden $\langle z, S, y\rangle$ als *zSy* abkürzen und analog für die anderen Nichtterminale.

Als nächstes verarbeiten wir die Regeln vom Typ (2), Suchen also alle Übergänge, die auf $\langle z, \epsilon\rangle$ enden. Dabei haben wir jeweils $z = z$, $z' = z$ und einmal $a' = a = a$ und einmal $a' = a = b$ (dabei kommen die Symbole rechts jeweils aus unserem Kellerautomaten, die links aus der Regeldefinition). Wir erhalten die folgenden Regeln:

$$zaz \to a \qquad zbz \to b.$$

Für die Regeln vom Typ (3) brauchen wir die Übergänge, die ein Zeichen am Keller hinterlassen. Davon haben wir nur eine, und für diese ist (mit der Konvention aus dem letzten Absatz) $z = z$, $a = \epsilon$, $a' = S$, $z_1 = z$, $b = b$. In unserer Zustandsmenge befinden sich z und y, wir erhalten also die folgenden beiden Regeln:

$$zSz \to \epsilon\, zbz \qquad zSy \to \epsilon\, zby.$$

Soweit wars einfach. Für die Regeln des Typs (4) müssen wir ernsthaft arbeiten. Bei $\langle z, \epsilon, S\rangle \to \langle y, Sa\rangle$ ist $z = z$, $a = \epsilon$, $a' = S$, $z_1 = y$, $b = S$ und $c = a$. Wir müssen sowohl z' als auch z_2 über beide Zustände iterieren, erhalten also vier Regeln:

$$zSz \to ySz\, zaz \quad zSz \to ySy\, yaz \quad zSy \to ySz\, zay \quad zSy \to ySy\, yay.$$

Für den anderen Übergang dieses Typs haben wir $z = y$, $a = \epsilon$, $a' = S$, $z_1 = z$, $b = a$ und $c = S$ und erhalten analog die Regeln

$$ySz \to zaz\,zSz \quad ySz \to zay\,ySz \quad ySy \to zaz\,zSy \quad ySy \to zay\,ySy.$$

Das hat keine große Ähnlichkeit mit der Ausgangsgrammatik. Wir können aber einen Haufen Regeln wegwerfen, wenn wir bedenken, dass Regeln, auf deren linker Seite etwas steht, was in keiner Regel auf der rechten Seite vorkommt, nie in einer Ableitung vorkommen können, die bei T anfängt und umgekehrt Regeln, in denen Nichtterminale vorkommen, die nie zu Terminalen werden können, ebenfalls nicht in Ableitungen von Wörtern vorkommen können. Mit letzterem Argument kann man alle Regeln streichen, die *yaz*, *yaz*, *yay* oder *zay* enthalten. Damit ist dann aber auch *ySy* nicht mehr erreichbar, so dass auch die Regel mit diesem Nichtterminalsymbol auf der linken Seite gestrichen werden kann.

Übrig bleiben

$T \to zSz$ $\quad T \to zSy$ $\quad zaz \to a$ $\quad zbz \to b$
$zSz \to zbz$ $\quad zSy \to zby$ $\quad zSz \to ySz\,zaz$ $\quad ySz \to zaz\,zSz$

Das ist immer noch ziemlich viel, lässt sich aber weiter vereinfachen, indem man die eigentlich überflüssigen Nichtterminale *zSy*, *zaz* und *zbz* durch Einsetzen eliminiert:

$T \to zSz$ $\quad T \to b$ $\quad zSz \to b$ $\quad zSz \to ySz\,a$ $\quad ySz \to a\,zSz$

Jetzt kann man noch fleißig Regeln ineinander einsetzen und erhält die Ausgangsgrammatik

$T \to a\,T\,a$ $\quad T \to b$

LR-Sprachen

Die Mengen der von deterministischen ($|\delta(z, a, A)| + |\delta(z, \epsilon, A)| \leq 1$) und von nichtdeterministische Kellerautomaten akzeptierten Sprachen sind verschieden.

Die von deterministischen Kellerautomaten akzeptierten Sprachen sind die LR(k)-Sprachen; sie sind eine Untermenge der kontextfreien Sprachen und können geparst werden, ohne jemals weiter als k Zeichen vorauszuschauen (bei allgemeinen CFGs können im Prinzip beliebig lange Sackgassen auftreten).

28. Abschlusseigenschaften

Seien $G_i = \langle \Phi_i, \Sigma_i, R_i, S_i \rangle$, $i = 1, 2$ kontextfreie Grammatiken. Was passiert unter Vereinigung, Verkettung, Sternbildung, Durchschnitt, Komplement und Differenz der erzeugten Sprachen?

Vereinigung:

$$\begin{aligned} G = &\langle \Phi_1 \cup \Phi_2 \cup \{S\}, \Sigma_1 \cup \Sigma_2, R, S \rangle \quad \text{mit} \\ &R = R_1 \cup R_2 \cup \{S \to S_1,\ S \to S_2\} \end{aligned}$$

Verkettung:

$$\begin{aligned} G = &\langle \Phi_1 \cup \Phi_2 \cup \{S\}, \Sigma_1 \cup \Sigma_2, R, S \rangle \quad \text{mit} \\ &R = R_1 \cup R_2 \cup \{S \to S_1 S_2\} \end{aligned}$$

Sternbildung:

$$\begin{aligned} G = &\langle \Phi_1 \cup \{S\}, \Sigma_1, R, S \rangle \quad \text{mit} \\ &R = R_1 \cup \{S \to S_1 S,\ S \to \epsilon\} \end{aligned}$$

Durchschnitt: CFGs sind nicht abgeschlossen gegen Durchschnitt. Gegenbeispiel: $L_1 = \{a^i b^i a^j\}$ und $L_2 = \{a^i b^j a^j\}$. Der Schnitt der beiden Mengen ist $\{a^i b^i a^i\}$ – dies ist nicht kontextfrei.

Komplement: Aus Abgeschlossenheit gegen Komplement würde Abgeschlossenheit gegen Durchschnitt folgen, deshalb sind CFGs auch nicht gegen Komplementbildung abgeschlossen.

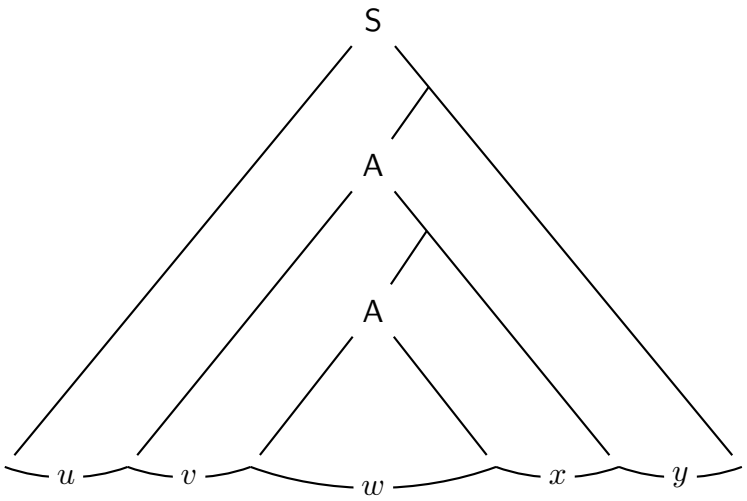


Fig. 20

Aus gegebenem Anlass die Schlussweise: Angenommen, $\overline{L}$ wäre kontextfrei. Dann würde aus der Kontextfreiheit von $L_1 \cup L_2$ die Kontextfreiheit von $\overline{L_1 \cup L_2}$ und durch erneute Komplementbildung die von $\overline{L_1} \cap \overline{L_2}$ folgen. Damit könnten wir aus der Abgeschlossenheit gegen Vereinigung die gegen Schnitt folgern – gegen Schnitt ist die Menge der kontextfreien Sprachen aber nicht abgeschlossen, Widerspruch.

Differenz: Komplement ließe sich auf Differenz zurückführen, deshalb: Wieder nicht abgeschlossen.

29. Pumping Lemma für CFGs

Satz: Sei L eine kontextfreie Sprache. Dann gibt es eine Konstante n, so dass für jedes Wort $z \in L$, $|z| \geq n$ gilt:

- $z = uvwxy$
- $|vx| \geq 1$
- $|vwx| \leq n$
- $uv^iwx^iy \in L$ für alle $i \geq 0$.

Beweis: Erzeuge G L, sei G in CNF. Da G binär verzweigend, können Ableitungsbäume der Höhe i höchstens Wörter der Länge 2^i ableiten.

Sei $n = 2^{|\Phi|+1}$, d.h., mindestens ein Nichtterminal kommt mehr als einmal in der Ableitung vor. Sei $|z| > n$. Der Ableitungsbaum von z muss die Form

(cf. Fig. 20)

haben (die Seite, von der aus jeweils A abgeleitet wird, ist dabei natürlich egal, es könnte auch der linke Ast sein). Durch mehrmaliges Durchlaufen der Ableitung $uAy \overset{*}{\rightarrow} uvAxy$ folgt die Behauptung.

Satz: $L = \{a^ib^ia^i\}$ ist nicht kontextfrei.

Beweis: Annahme: Behauptung falsch. Dann existiert ein $z = a^nb^na^n$ mit n der Schranke aus dem PL und $z = uvwxy$.

Weil nach PL $|vwx| \leq n$, kann vwx nur eine der Formen b^{l_1}, a^{l_1} (oBdA aus den linken a), $a^{l_1}b^{l_2}$ oder $b^{l_1}a^{l_2}$ haben. Dann hat aber $uwy \in L$ ($i = 0$ im PL) eine der Formen $a^nb^{n-l'}a^n$, $a^{n-l'}b^na^n$, $a^{n-l'}b^{n-l''}a^n$ oder $a^nb^{n-l'}a^{n-l''}$, wobei $l' \leq l_1$ und $l'' \leq l_2$. Weil $|vx| \geq 1$, ist aber auch mindestens eines von l' und l'' größer als Null, und deshalb liegt keines dieser Wörter in L.

Aus dem Pumping Lemma folgt auch, dass Sprachen wie

$$\{a^p \mid p \text{ Quadratzahl}\}$$

nicht kontextfrei sein können.

30. Nutzen der CFGs

Kunstsprachen

Fast alle Programmiersprachen sind durch kontextfreie (meist $LR(k)$) Grammatiken beschrieben

Natursprachen

Umstritten – stärkstes Argument: CFGs sind das Komplexeste, das wir noch vernünftig beherrschen.

Klabunde S. 115: Schweizerdeutsch ist komplizierter als CFG. Man mag durchaus darüber diskutieren, wie viel solche Argumente wert sind. Zunächst nämlich sind Natursprachen de facto endlich (alle Äußerungen, über deren Richtigkeit man befinden müsste, müssen in finiter Zeit gemacht werden, und eigentlich dürfte schon unstrittig sein, dass „Sätze“ oberhalb eine gewissen Länge als zumindest ungeschickt zurückgewiesen werden sollten). Daher ließen sie sich *im Prinzip* schon durch reguläre Sprachen modellieren. Die Frage ist, ob dies eine *adäquate* Beschreibungsweise ist, und diese Frage ist für reguläre Grammatiken auf jeden Fall mit Nein zu beantworten. Für kontextfreie Grammatiken sieht das anders aus – und durch geeignetes „Ausrollen“ für einfache oder zweifache Verkreuzung lassen sich auch scheinbar kontextsensitive Sprachteile kontextfrei repräsentieren.

Aber: Zur Modellierung von Sprachteilen oder zum Teilparsen insbesondere in statistischen Verfahren reichen CFGs. Weitere Anwendungen in Morphologie, Phonetik, Semantik...

Wieder auf einem anderen Blatt steht, dass reine Phrasenstrukturgrammatiken nicht mehr als ernsthafte Modelle für die Modellierung von Natursprachen verwendet werden. In der Regel kommen irgendwelche Formen von Unifikation von außen dazu, häufig geht man auch ganz von der Phrasenstruktur ab und baut auf Dependenz; darin fungieren bestimmte Wörter als „Köpfe“, von denen dann wiederum andere Wörter „abhängen“. In solchen Formalismen ist die Beurteilung des Sprachtyps nicht immer ganz einfach (aber natürlich befinden sich die Sprachen irgendwo in der Chomsky-Hierarchie, denn die syntaktisch zulässigen Wörter (also Sätze) formen eine Menge, die prinzipiell auch mit Phrasenstrukturgrammatiken zu erzeugen wäre, nur eben mit weit unübersichtlicheren).

EBNF

EBNF (Extended Backus-Naur Form) ist Sprache zur Darstellung von kontextfreien Grammatiken. Jede EBNF-Klausel definiert eine Ersetzungsregel, die vorschreibt, wie ein Nichtterminalsymbol in andere Nichtterminalsymbole oder Terminalsymbole umgesetzt wird. Eine Regel sieht so aus:

$$\textit{ganzeZahl} \ ::= [\textit{vorzeichen}]\textit{ziffer}\,\{\textit{ziffer}\}$$

Eine ganze Zahl besteht aus einem optionalen Vorzeichen (die eckigen Klammern), einer Ziffer und null oder mehr weiteren Ziffern (die geschweiften Klammern). Die (kursiv geschriebenen) Nichtterminalsymbole *vorzeichen* und *ziffer* müssen noch erklärt werden. Das geht mit

$$\begin{aligned} \textit{ziffer} \ ::= & \texttt{"0"} \mid \texttt{"1"} \mid \texttt{"2"} \mid \texttt{"3"} \mid \texttt{"4"} \mid \\ & \texttt{"5"} \mid \texttt{"6"} \mid \texttt{"7"} \mid \texttt{"8"} \mid \texttt{"9"} \mid \\ \textit{vorzeichen} \ ::= & \texttt{"+"} \mid \texttt{"-"} \end{aligned}$$

In `typewriter` gedruckt sind Terminalsymbole , der vertikale Strich steht für „oder“. EBNF lediglich eine kompakte Darstellung unserer gewohnten Regelmengen. Die Regel für *ganzeZahl* z.B. ließe sich auch schreiben als:

$$\begin{aligned} G &\to Z \\ G &\to VZ \\ Z &\to NZ \\ Z &\to N, \end{aligned}$$

wo G die Ganze Zahl, Z eine Ziffernfolge, V ein Vorzeichen und N eine Ziffer ist.

Mithin ist die Menge der durch Sätze von EBNF-Klauseln beschriebenen Sprachen gleich der Menge der kontextfreien Sprachen.

Vorsicht: Es gibt so viele Varianten von BNF und EBNF wie es AutorInnen gibt, die etwas damit ausdrücken. Häufige Varianten:

- Variation von „::=“
- Nichtterminale in spitzen Klammern
- Terminale ohne Anführungszeichen

31. Kontextsensitive Sprachen

kontextsensitiv
längenmonoton
Kuroda-Normalform

Eine Grammatik $G = \langle \Phi, \Sigma, R, S\rangle$ heißt *kontextsensitiv* (oder vom Chomsky-1-Typ), wenn alle Regeln eine der beiden Formen

$$\begin{aligned} \alpha A \gamma &\rightarrow \alpha\beta\gamma \\ S &\rightarrow \epsilon \end{aligned}$$

haben. Dabei ist $A \in \Phi$, $\alpha, \gamma \in (\Sigma \cup \Phi)^*$ und $\beta \in (\Sigma \cup \Phi)^* \setminus \{\epsilon\}$. Falls $\langle S, \epsilon\rangle \in R$, darf weiter S nicht auf der rechten Seite einer Regel vorkommen.

Kontextsensitive Grammatiken sind *längenmonoton*, d.h. die Länge von Wörtern in einer Ableitung nimmt nie ab.

Satz: Jede längenmonotone Sprache ist kontextsensitv.

Beweisidee: Das Problem sind hier lediglich Regeln, in denen mehr als ein Nonterminal ersetzt wird; diese lassen sich durch Einführung von „Zwischen-Nichtterminalen“ aufteilen.

Man geht dabei schrittweise so vor, dass in jedem Schritt nur genau ein Nichtterminal ersetzt wird. Dazu führt man neue Nichtterminale ein, die nur genau im passenden Kontext ersetzt werden können. Eine Regel $ABC \rightarrow CAB$ könnte dabei in folgende Regeln zerfallen:

$ABC \rightarrow X_1BC$ $\quad X_1BC \rightarrow X_1X_2C$ $\quad X_1X_2C \rightarrow X_1X_2X_3$
$X_1X_2X_3 \rightarrow CX_2X_3$ $\quad CX_2X_3 \rightarrow CAX_3$ $\quad CAX_3 \rightarrow CAB$

Wenn in Regeln dieser Art bereits auf der linken Seite Terminale stehen (z.B. $aX \rightarrow Xa$), müssen für die betreffenden Terminale neue Nichtterminale eingeführt werden, die nach dem Muster der entsprechenden Manipulationen zur Wandlung kontextfreier Grammatiken in CNF dann in allen Regeln ersetzt werden.

Kontextsensitve Sprachen sind eine echte Obermenge kontextfreier Sprachen.

$\{a^ib^ia^i\}$ ist nicht kontextfrei, kann aber durch folgende Regeln erzeugt werden:

$S \rightarrow A$ $\quad A \rightarrow aABC$ $\quad A \rightarrow abC$
$CB \rightarrow BC$ $\quad bB \rightarrow bb$ $\quad C \rightarrow a$

Bei Klabunde befindet sich hier ein subtiler Fehler: Er hat in der Regel $A \rightarrow abC$ ein großes B und dann zusätzlich eine Regel $aB \rightarrow ab$. Dies führt zu Übergenerierung, da im Prinzip sofort alle C in a umgewandelt werden können und dann die zusätzliche Regel die B direkt in b umwandeln kann.

Beispielableitung: $aabbaa$

$S \rightarrow A \rightarrow aABC$
$aabCBC \rightarrow aabBCC$
$aabbCC \rightarrow aabbaC \rightarrow aabbaa$

Man beachte, dass hinter dieser Grammatik schon eine regelrecht algorithmische Idee steckt: Wir sorgen zunächst dafür, dass immer nur aBC gemeinsam erzeugt werden und die keines der Nichtterminale verschwinden kann, bevor es zu einem Terminal wird. Dann bleibt das Problem, dass wir die B und C (die b und „rechten“ a entsprechen) noch durchmischt haben.

Wir müssen also dafür sorgen, dass sie kontrolliert entmischt werden (und genau das geht mit kontextfreien Grammatiken nicht). Die Idee ist, dass wir, wenn wir genug a haben, das Nichtterminal A, das das Wachstum bewirkt, „sterilisieren“ (das ist die Regel $A \rightarrow aBC$) und dann anfangen, die C nach hinten zu treiben. Das geht einerseits durch die kontextsensitive Regel zur Vertauschung von B und C, andererseits durch den Zwang, ein B nur dann in ein b verwandeln zu können, wenn vor ihm ein b steht.

Chomsky-1-Grammatiken, die nicht ϵ erzeugen, können in die *Kuroda-Normalform* gebracht werden, in der alle Regeln eine der Formen

$$A \rightarrow a \quad A \rightarrow B \quad A \rightarrow BC \quad AB \rightarrow CD$$

haben.

Erkennung von kontextsensitiven Sprachen durch *Turingmaschinen*.

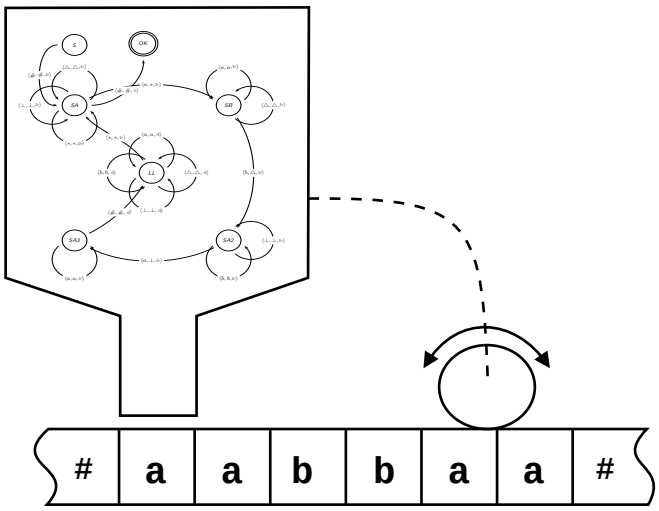


Fig. 21

Problems

(31.1) Überzeugt euch davon, dass die Regeln, die wir oben aus $ABC \to CAB$, tatsächlich kontextfrei sind (indem ihr α, γ und β angebt). Schreibt danach eine wilde längenmonotone Regel hin und wandelt sie in kontextfreie Regeln.

32. Turingmaschinen I

Eine *Turingmaschine* $T = \langle \Phi, \Sigma, \#, \delta, S, F \rangle$ besteht aus

- Einem Alphabet von Zuständen Φ
- Einem Alphabet von Bandsymbolen Σ
- Einem Nullsymbol $\#$
- Einer Übergangsfunktion $\delta\colon \Phi \times (\Sigma \cup \{\#\}) \to \Phi \times (\Sigma \cup \{\#\}) \times \{\triangleleft, \triangleright, \circ\}$
- Einem Startzustand $S \in \Phi$
- Einer Menge von Endzuständen $F \subseteq \Phi$

Analog zu den Automaten lassen sich hier nichtdeterministische Varianten definieren, indem δ Werte in der Potenzmenge des hier verwendeten Wertebereichs nimmt. Wie endliche Automaten sind auch deterministische und nichtdeterministische Turingmaschinen äquivalent, solange wir keine Zeit- und Bandbeschränkungen haben.

Die Bandsymbole stehen auf einem Band, das „gespult“ werden kann. Je nach dritter Komponente des Werts der Übergangsfunktion wird es nach links ($\triangleleft$), rechts ($\triangleright$) oder gar nicht ($\circ$) bewegt. Ein $\#$ steht für „Hier steht nichts“ und kann als Begrenzungssymbol verwendet werden. Grundsätzlich ist das Band einer Turingmaschine aber unendlich. Das bedeutet, dass, wann immer der Lesekopf auf ein Feld kommt, das zuvor noch nicht berührt wurde, dort $\#$ steht.

Das erste Element des Werts der ÜF ist das zu schreibende Zeichen.

Die Übergangsfunktion einer Maschine, die nach links hin alles mit Einsen zukleistert, wäre demnach $\delta(z, \#) = \langle z, 1, \triangleleft \rangle$.

(cf. Fig. 21)

Im Unterschied zu einem Kellerautomaten kann die Turingmaschine überall im Speicher lesen und schreiben, und Speicher und „Eingabeband“ sind identisch.

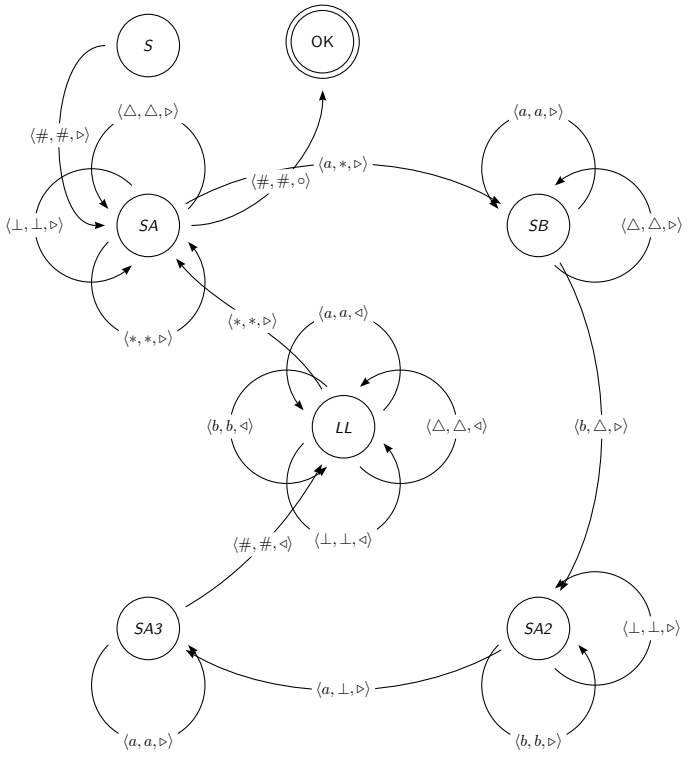


Fig. 22

33. Turingmaschinen II

(cf. Fig. 22)

Die Kantenkennzeichnung ist ein Tripel aus (anstehendem Zeichen, geschriebenem Zeichen, Bandbewegung).

Eine Turingmaschine mit dieser Übergangsfunktion erkennt die Sprache $\{a^i b^i a^i\}$.

Idee dabei: Die Maschine läuft i-Mal über die Eingabe und ersetzt dabei jeweils genau ein linkes a (im Zustand *SA*), ein b (im Zustand (*SB*) und ein rechtes a (im Zustand *SA2*), durch Hilfssymbole; Hilfssymbole werden jeweils überlesen. Dann spult die Maschine im Zustand *LL* zurück, nachdem sie in *SA3* das rechte Ende gefunden hat.

Wenn alles nur noch aus Hilfssymbolen besteht, läuft *SA* durch und erreicht das Ende des Bands – die Maschine landet in *OK*, sonst „stürzt sie ab".

Den Zustand einer Turingmaschine stellen wir dar als ein Wort $k \in \Sigma^* \bullet \Phi \bullet \Sigma^*$, so dass links und rechts der Bandinhalt links und rechts vom Kopf steht und in der Mitte der augenblickliche Zustand.

Wir leiten $aabbaa$ ab:

$S\ \#aabbaa\#$	$\to$	$\#\ SA\ aabbaa\#$	$\to$
$\#*\ SB\ abbaa\#$	$\to$	$\#*a\ SB\ bbaa\#$	$\to$
$\#*a\triangle SA2baa\#$	$\to$	$\#*a\triangle bSA2aa\#$	$\to$
$\#*a\triangle b\perp SA3a\#$	$\to$	$\#*a\triangle b\perp aSA3\#$	$\to$
$\#*a\triangle b\perp\ LL\ a\#$	$\to$	$\#*a\triangle b\ LL\ \perp a\#$	$\to$
$\#*a\triangle\ LL\ b\perp a\#$	$\to$	$\#*a\ LL\ \triangle b\perp a\#$	$\to$
$\#*\ LL\ a\triangle b\perp a\#$	$\to$	$\#\ LL\ *a\triangle b\perp a\#$	$\to$
$\#*\ SA\ a\triangle b\perp a\#$	$\to$	$\#**\ SB\ \triangle b\perp a\#$	$\to$
$\#**\triangle\ SB\ b\perp a\#$	$\to$	$\#**\triangle\triangle SA2\perp a\#$	$\to$
$\#**\triangle\triangle\perp SA2a\#$	$\to$	$\#**\triangle\triangle\perp\perp SA3\#$	$\to$
$\#**\triangle\triangle\perp\ LL\ \perp\#$	$\to$	$\#**\triangle\triangle\ LL\ \perp\perp\#$	$\to$
$\#**\triangle\ LL\ \triangle\perp\perp\#$	$\to$	$\#**\ LL\ \triangle\triangle\perp\perp\#$	$\to$

$$
\begin{array}{rcl@{\qquad}rcl}
\#* \mathit{LL} *\triangle\triangle\bot\bot\# & \to & & \# \mathit{SA} ** \triangle\triangle\bot\bot\# & \to \\
\#* \mathit{SA} *\triangle\triangle\bot\bot\# & \to & & \#** \mathit{SA} \triangle\triangle\bot\bot\# & \to \\
\#**\triangle \mathit{SA} \triangle\bot\bot\# & \to & & \#**\triangle\triangle \mathit{SA} \bot\bot\# & \to \\
\#**\triangle\triangle\bot \mathit{SA} \bot\# & \to & & \#**\triangle\triangle\bot\bot \mathit{SA} \# & \to \\
\#**\triangle\triangle\bot\bot \mathit{OK} \# & & & & &
\end{array}
$$

linear beschränkt

34. Turingmaschinen und Chomsky-0,1-Sprachen

Eine Turingmaschine heißt *linear beschränkt* (LBA), wenn alle von ihr erreichbaren Konfigurationen $\alpha z \beta$ die Bedingung $|\alpha \bullet \beta| \leq n$ erfüllen.

Satz: Zu jeder Chomsky-1-Grammatik $G = \langle \Phi, \Sigma, R, S \rangle$ gibt es eine nichtdeterministische LBA $M = \langle \Phi', \Sigma', \#, \delta, S', F \rangle$, die $L(G)$ akzeptiert und umgekehrt.

Beweisidee: ($G \to M$) Sei $G = \langle \Phi, \Sigma, R, S \rangle$ gegeben. Die zu konstruierende Maschine M mit $\Phi \cup \Sigma$ als Bandalphabet bekommt $x = a_0 \dots a_n$ als Eingabe. Wir wählen nichtdeterministisch eine Regel $u \to v \in R$ und wenden sie auf das erste Vorkommen von v in x an. Dabei kann das Eingabewort nicht länger werden. Der Automat setzt dies fort, bis das Band nur noch $\#S\#$ enthält.

($M \to G$) Konfigurationsübergänge der Maschine lassen sich auf Symbole der Grammatik abbilden (Schöning, S. 86).

Anmerkung: Hier ist wichtig, dass M *nichtdeterministisch* ist. Zwar können auch deterministische TMen Chomsky-1-Sprachen akzeptieren, ob sie dann aber noch linear beschränkt sind, ist offen (LBA-Problem).

Satz: Chomsky-0-Sprachen werden genau durch allgemeine Turingmaschinen akzeptiert

Chomsky-1 unterscheidet sich nur durch die Längenmonotonie von Chomsky-0, und der einzige Punkt, an dem die Längenmonotonie im „Beweis“ oben relevant war, war die lineare Beschränktheit.

Abschlusseigenschaften

Chomsky-1-Sprachen sind abgeschlossen unter Schnitt, Vereinigung, Komplement (anders als Chomsky-2!), Verkettung und Sternbildung.

Diese Eigenschaften lassen sich durch geeignete Verschaltung von Turingmaschinen zeigen. Problematisch dabei ist allenfalls die lineare Beschränkung.

Interessant ist vor allem die Komplementbildung. Die Idee dabei ist, dass man eine Turingmaschine bauen kann, die alle Wörter erzeugt, die nicht länger als ein zu untersuchendes sind und die dann abklappert. Man kann zeigen, dass das mit einem linear beschränkten Band geht. Die Aufzählung geht für allgemeine Regelsprachen nicht mehr, deshalb:

Chomsky-0-Sprachen unter Komplementbildung nicht abgeschlossen. Dieser Umstand hat recht profunde Konsequenzen. Hätte man z.B. eine Sprache aller „wahren“ Aussagen innerhalb eines formalen Systems irgendeiner Art und wäre sie eine allgemeine Regelsprache, so wäre das Komplement möglicherweise keine allgemeine Regelsprache mehr – da wir aber für noch allgemeinere Sprachen das Wortproblem nicht mehr lösen können, würde das bedeuten, dass wir zwar garantiert ausrechnen können, dass eine Aussage wahr ist, aber möglicherweise nicht, dass sie falsch ist.

35. Berechenbarkeit

Turing-berechenbar

charakteristische Funktion

halbe charakteristische Funktion

entscheidbar

semi-entscheidbar

rekursiv aufzählbar

Turingmaschinen sind sehr allgemeine Konstrukte. Glaubenssatz:

Church'sche These: Die Klasse der „intuitiv" berechenbaren Funktionen stimmt mit der der Turing-berechenbaren Funktionen überein *oder:* Turing-Maschinen können alles, was Computer können (die Umkehrung ist sofort offensichtlich, wenn wir ein Programm zur Simulation einer Turing-Maschine schreiben können, und das geht schon für recht simple Systeme, z.B. TEX).

Dabei heißt eine (partielle) Funktion $f: \Sigma^* \to \Sigma^*$ *Turing-berechenbar*, wenn es eine (deterministische) Turing-Maschine M gibt, so dass $f(x) = y$ genau dann, wenn $Sx \overset{*}{\to} \#Zy\#$ mit Z Endzustand von M.

Wir verwenden das Symbol $\overset{*}{\to}$ hier etwas schlampig – es soll hier natürlich heißen „es gibt eine Zustandsfolge, die die linke in die rechte Seite überführt". Wichtig ist, dass keine Aussage über das Verhalten der Maschine an den Stellen gemacht wird, an denen f nicht definiert ist.

Die *charakteristische Funktion* χ_M einer Menge $M \subseteq N$ ist

$$\chi_M : N \to \{0,1\} \qquad \chi_M(x) = \begin{cases} 1 & x \in M \\ 0 & x \notin M \end{cases}$$

die *halbe charakteristische Funktion* χ'_M ist

$$\chi'_M : N \to \{0,1\} \qquad \chi'_M(x) = \begin{cases} 1 & x \in M \\ \text{undef} & x \notin M \end{cases}$$

Beispiel: Sei L eine Chomsky-0-Sprache. Für jedes $w \in L$ stoppt eine Turingmaschine mit $L = L(M)$. Also sind die charakteristischen Funktionen

- $\chi_L: L \to \{0,1\}$ konstant eins, also berechenbar;
- $\chi_{\bar{L}}: \Sigma^* \to \{0,1\}$ nicht berechenbar, weil für $x \in \Sigma^* \setminus L$ das Verhalten von M nicht definiert ist
- $\chi'_L: \Sigma^* \to \{0,1\}$ berechenbar, weil das Verhalten nur für $x \in L$ entscheidend ist.

Eine Sprache L heißt *entscheidbar*, wenn ihre charakteristische Funktion berechenbar ist, *semi-entscheidbar*, wenn ihre halbe charakteristische Funktion berechenbar ist.

Eine Sprache L heißt *rekursiv aufzählbar*, wenn $L = \emptyset$ oder eine (totale) berechenbare Funktion $f: \mathbb{N} \to \Sigma^*$ existiert, so dass $L = \{f(1), f(2), \ldots\}$.

Satz: L rekursiv aufzählbar $\Leftrightarrow$ L semi-entscheidbar.

Beweisidee: „$\Rightarrow$": Der Algorithmus prüft einfach alle $f(i)$ durch.

„$\Leftarrow$": Wir definieren zunächst $w: \mathbb{N} \to \Sigma^*$ (das geht, weil Sternbildung sehr simpel und Σ endlich ist). Sei χ'_L durch den Algorithmus M realisiert. Wir nehmen ein Wort $a \in L$ als „default" und definieren eine Funktion

$$\tilde{f}(x,y) = \begin{cases} w(x) & M(w(x)) \text{ stoppt in } y \text{ Schritten} \\ a & \text{sonst} \end{cases}$$

Wenn wir die zwei Parameter von f in eine natürliche Zahl verpacken (es gibt rekursiv berechenbare Funktionen, die das können), haben wir die gewünschte Aufzählungsfunktion.

Anmerkung: Der Unterschied zwischen rekursiver Aufzählbarkeit und Abzählbarkeit ist, dass bei der Abzählbarkeit nicht die Berechenbarkeit der Bijektion gefordert ist (wichtig bei Teilmengenbildung!).

36. Entscheidbarkeitsprobleme

Das Wortproblem

Ist $L(G)$ entscheidbar?

Ja für 1, 2, 3. Für Chomsky-3 ist das Wortproblem sogar in linearer Zeit lösbar. Ansonsten Argument über zunehmende Länge der Ableitungen – die Zahl der möglichen Vorgänger eines Wortes ist endlich. Das Wortproblem für Chomsky-1 ist allerdings mindestens NP-hart.

Nein für 0 (das Wortproblem für Chomsky-0-Sprachen ist nur semi-entscheidbar).

Das Leerheitsproblem

Ist $L(G) = \emptyset$ berechenbar?

Ja für 2 und 3. Bei 3 kann man allen Pfaden im DEA folgen und sehen, ob wenigstens einer in einem Endzustand landet. Bei 2 in CNF werden erst alle Nonterminale in einer Regel $A \to a$ markiert, dann alle Nonterminale in $A \to BC$, wenn B und C schon markiert sind. Wenn am Ende S markiert ist, ist die Sprache nicht leer. Auf diese Weise kann man übrigens auch Parser bauen: CYK-Parser.

Nein für 0 und 1.

Das Endlichkeitsproblem

Ist $|L(G)| < \infty$ berechenbar?

Ja für 2 und 3. Suche nach Zyklen in den zugehörigen Automaten oder Verwendung des Pumping Lemma (wenn es ein Wort über der Schranke um Pumping Lemma gibt, ist die Sprache unendlich aufpumpbar, es müssen also nur endlich viele Wörter geprüft werden).

Nein für 0 und 1.

Das Äquivalenzproblem

Ist $L(G_1) = L(G_2)$ berechenbar?

Ja für 3, weil unter Schnitt, Vereinigung und Komplementbildung abgeschlossen. Da

$$L_1 = L_2 \Leftrightarrow (L_1 \cap \overline{L_2}) \cup (L_2 \cap \overline{L_1}) = \emptyset,$$

kann Äquivalenzproblem auf Leerheitsproblem zurückgeführt werden.

Nein für 0, 1 und 2. Für deterministisch kontextfreie Sprachen ist es allerdings entscheidbar.

Die Behauptungen, für die hier keine Argumente angegeben sind, sind eher schwierig zu beweisen. Häufig eine wichtige Rolle dabei spielt ein weiteres unentscheidbares Problem, das in der theoretischen Informatik an vielen Stellen lauert: Das Post'sche Korrespondenzproblem (PCP), zu dem später in diesem Skript noch etwas mehr gesagt wird.

37. Das Halteproblem

spezielle Halteproblem

Sei $\Sigma = \{a_1, \ldots, a_n\}$ und $\Phi = \{Z_1, \ldots, Z_n\}$. Wir können statt Zeichen bzw. Zuständen auch ihre Indizes verwenden. Den Turing-Übergang

$$\delta(Z_i, a_j) = (Z_{i'}, a_{j'}, y)$$

können wir schreiben als

$$w_{i,j,i',j',y} = \#\#i\#j\#i'\#j'\#m,$$

wobei die Zahlen binär dargestellt sind und $m = 0, 1, 2$ ist, wenn $y = \triangleleft, \triangleright, \circ$.

Schließlich kodieren wir

$$0 \to 00 \quad 1 \to 01 \quad \# \to 11$$

Damit haben wir Turingmaschinen injektiv auf die Sprache $\{0,1\}^*$ abgebildet und damit auf die natürlichen Zahlen.

Es gibt Turingmaschinen, die diese Darstellung ausführen können; „programmierbare Turingmaschine“, *universelle Turingmaschine*.

Das *spezielle Halteproblem* ist die Sprache

$$K = \{w \in \{0,1\}^* \mid M_w \text{ angewendet auf } w \text{ hält}\}.$$

Es gibt dabei das Problem, dass nicht jedes Element von $\{0,1\}^*$ eine Maschine kodiert (die Abbildung der Turingmaschinen in die natürlichen Zahlen ist injektiv, aber nicht surjektiv). Wir umgehen es, indem wir definieren, dass eine (beliebige) „Fallback-Maschine“ $\hat{M}$ läuft, wenn wir gerade keine gültige Maschinenkodierung haben (auf diese Weise wird die Abbildung surjektiv, aber nicht mehr injektiv – dafür ist die von den natürlichen Zahlen in die Turingmaschinen injektiv, und das zählt in dieser Definition).

Satz: Das spezielle Halteproblem ist nicht entscheidbar.

Beweis: Angenommen, K ist entscheidbar. Wir bauen eine Maschine M, die χ_K berechnet und eine weitere Maschine M', die stoppt, wenn M 0 ausgibt und sonst endlos läuft. M' ist eine Turingmaschine und kann daher durch das Wort $w' \in \{0,1\}^*$ ausgedrückt werden. Wir wenden M' auf w' an:

Dann gilt:

$$\begin{aligned} M'(w') \text{ hält} \quad &\Leftrightarrow M(w') = 0 \\ &\Leftrightarrow \chi_K(w') = 0 \\ &\Leftrightarrow w' \notin K \\ &\Leftrightarrow M'(w') \text{ hält nicht} \end{aligned}$$

Diese Argumentation braucht keine Turingmaschine, sondern lediglich die Selbstanwendbarkeit. Möge folgender (geeignet ergänzter) C-Quelltext als haelt.c gespeichert sein:

```
int haelt(char *prog, char *in)
/* gibt 1 zurück, wenn das C-Programm prog auf der Eingabe in
hält, sonst 0 */
{ /* gibt es nicht :-) */
}

int main(void)
{ char *mysource = readFile("haelt.c");

  if (haelt(mysource, mysource)) while (1);
  return 1;
}
```

geht genauso: Wenn das Halteproblem entscheidbar ist, muss `haelt(mysource, mysource)` eins zurückgeben und damit main in eine Endlosschleife treiben. Dann aber dürfte `haelt` keine eins zurückgeben, weil das Programm ja nicht terminiert.

Das *allgemeine Halteproblem* ist die Sprache

allgemeine Halteproblem

$$H = \{w\#x \mid M_w(x) \text{ hält}\}$$

Satz: Das allgemeine Halteproblem ist nicht entscheidbar.

Beweis: Rückführung auf spezielles Halteproblem: Schon $w\#w$ ist nicht entscheidbar.

Auch das *Halteproblem auf leerem Band* ist nicht entscheidbar.

38. Ergänzung: PCP I

Mit die bequemste Methode, die Unentscheidbarkeit etlicher Probleme der Theoretischen Linguistik nachzuweisen, bedient sich des so genannten Post'schen Korrespondenzproblems PCP.

Beim PCP hat man eine endliche Folge von Wortpaaren $K = (\langle v_i, w_i\rangle)_{i=1,\ldots,n}$ und sucht eine Folge von Indizes $i_k \in \{1,\ldots,n\}$, so dass

$$v_{i_1} v_{i_2} \ldots v_{i_l} = w_{i_1} w_{i_2} \ldots w_{i_l}$$

gilt. Man möchte also aus einander entsprechenden Wortbruchstücken gleiche Wörter aufbauen.

Beispiel: Mit $K = (\langle 1, 101\rangle, \langle 10, 00\rangle, \langle 011, 11\rangle)$ ist *eine* Lösung des PCP $(1, 3, 2, 3)$. Das kann man durch Einsetzen überprüfen:

$$v_{i_1} \bullet v_{i_2} \bullet v_{i_3} \bullet v_{i_4} = v_1 v_3 v_2 v_3 = 101110011$$
$$w_{i_1} \bullet w_{i_2} \bullet w_{i_3} \bullet w_{i_4} = w_1 w_3 w_2 w_3 = 101110011$$

Wir können zeigen: Das so harmlos aussehende PCP ist unentscheidbar. Dazu braucht man den folgenden Begriff:

Seien $A \subset \Sigma^*$ und $B \subset \Gamma^*$ Sprachen. Dann gilt $A \leq B$ („A ist auf B reduzierbar"), wenn

$$x \in A \Leftrightarrow f(x) \in B$$

mit $f\colon \Sigma^* \to \Gamma^*$ total und berechenbar.

Reduzierbarkeit bedeutet insbesondere, dass die charakteristische Funktion von A einfach $\chi_B \circ f$ ist, also die Verkettung des eben definierten f mit der charakteristischen Funktion von B. Ist nun letztere berechenbar, so ist es auch χ_A. Das bedeutet aber weiter, dass sich (Semi-) Entscheidbarkeit durch die Reduktion von Problemen durchzieht.

Indem wir nun eine totale und berechenbare Funktionen angeben (oder eher so tun, als ob), die das PCP auf das Halteproblem reduziert, können wir aus der Unentscheidbarkeit des Halteproblems auf die des PCP schließen.

Diese Reduktion ist vor allem ein technisches Problem, bei dem zunächst das PCP auf ein modifiziertes Problem, das MPCP, bei dem $i_1 = 1$ gefordert wird, reduziert wird. Es ist nicht überraschend, dass das PCP auf das MPCP reduziert werden kann, da ja die Reihenfolge der Paare in der Folge kaum Einfluss auf die Lösbarkeit haben dürfte und durch Umordnung $i_1 = 1$ immer erreicht werden kann. Diese Argumentation hat allerdings Lücken, ein soliderer Beweis findet sich bei Schöning im Kapitel 2.7.

Satz: H $\leq$ MPCP (H steht dabei für das Halteproblem).

Zum Beweis lassen wir das MPCP die Folge der Konfigurationen der Turingmaschine M angesetzt auf w erzeugen und sorgen dafür, dass gleiche $x = x_1 \bullet x_2 \bullet \cdots \bullet x_n$ und y nur dann herauskommen können, wenn M in einem Endzustand ist.

Dazu setzen wir $K = (\langle \$, \$z_0 w \$\rangle, \ldots)$. Dabei ist \$ ein Trennzeichen, das sonst nicht im Bandalphabet von M vorkommt und z_0 der Startzustand von M. Das bewirkt, dass in y zu Anfang die Anfangskonfiguration steht, während x im Wesentlichen leer ist.

Die restlichen Paare ergeben sich aus der Definition von M. So dürfen wir etwa das Band von y nach x „kopieren", indem für alle $a \in \Sigma$ ein $\langle a, a\rangle$ zu K hinzugefügt wird.

Wo wir einen Zustand haben, müssen wir die Änderung von einer zur anderen Konfiguration übernehmen, indem etwa aus $\delta(z,a) = \langle z', c, \triangleright\rangle$ ein Paar wie $\langle za, cz'\rangle$ in K wird. Beachtet, wie hier das Verhalten der Turingmaschine simuliert wird. Man braucht drei Schablonen dieser Art plus ein paar weitere, weil wir am Anfang und am Ende jeder Konfiguration jeweils ein etwas anderes Verhalten brauchen, um unser Trennzeichen \$ einzufügen.

Ist M in einem Endzustand, können wir den Bandinhalt löschen, etwa durch die Paare $\langle az_e, z_e\rangle$ und $\langle z_e a, z_e\rangle$ in K. Dabei ist $a \in \Sigma$ und $z_e \in F$.

Wenn dann nur noch der Endzustand in der Konfiguration steht, lassen wir x aufholen, was durch Anhängen von $\langle z_e\$\$, \$\rangle$ an K erreicht wird.

Die Idee des ganzen Spiels ist, dass zu jeder Zeit während der Lösung des MPCP

$$x = \$k_1\$k_2\$\cdots\$k_{n-1}$$
$$y = \$k_1\$k_2\$\cdots\$k_{n-1}\$k_n$$

ist. Dabei sind die k_i jeweils gültige Konfigurationen von M, und an den jeweils letzten wird gerumgedoktort. x ist also immer um eine Konfiguration kürzer als y – bis eben ganzu zum Schluss, wenn durch $\langle z_e\$\$, \$\rangle$ zwei Trennzeichen an x kommen. Zu diesem Zeitpunkt haben die Löschregeln bereits dazu geführt, dass in y alles leer ist.

Den vollen Regelsatz gibt es wieder im Kapitel 2.7 des Schöning.

39. Ergänzung: PCP II

Das PCP lässt sich auf viele Probleme, die im Zusammenhang mit Sprachen auftreten, reduzieren. Das Prinzip ist dabei die Zuordnung zweier Grammatiken G_1 und G_2 über $\Sigma_1 = \Sigma_2 = \{0, 1, \$, a_1, \ldots, a_n\}$ zum PCP $K = (\langle x_i, y_i\rangle)_{i=1\ldots n}$ mit $x_i, y_i \in \{0,1\}^*$ wie folgt:

$$R_1 = \{S \to A\$B, \quad A \to a_i A x_i, \quad A \to a_i x_i,$$
$$B \to \tilde{y}_i B a_i, \quad B \to \tilde{y}_i a_i\}$$

und

$$R_2 = \{S \to a_i S a_i, \quad S \to T, \quad T \to 0T0,$$
$$T \to 1T1, \quad T \to \$\}.$$

Dabei läuft i jeweils von 1 bis n, und $\tilde{x}$ ist die Umkehrung von x – z.B. ist für $x = abc$ dann $\tilde{x} = cba$. Diese Grammatiken wurden so *konstruiert*, dass sie eine Abbildung des PCP erlauben. Dazu betrachtet man die erzeugten Sprachen:

$$L_1 = \{a_{i_n} \cdots a_{i_1} x_{i_1} \cdots x_{i_n} \$ \tilde{y}_{j_m} \cdots \tilde{y}_{j_1} a_{j_1} \cdots a_{j_m}\}$$
$$L_2 = \{uv\$\tilde{v}\tilde{u} \mid u \in \{a_i\}^*, v \in \{0,1\}^*\}.$$

Die Elemente von L_2 sind gerade die an \$ „symmetrischen“ Wörter, die von L_1 gerade die, in denen Kandidaten für Lösungen des PCP rechts und links vom \$ stehen (y ist dabei gespiegelt). Der Schnitt der beiden Sprachen sind die symmetrischen Wörter, bei denen rechts und links Kandidaten für die Lösung zum PCP stehen – und weil sie symmetrisch sind, sind die Kandidaten tatsächlich Lösungen.

Damit gilt: Das PCP ist auf das Schnittproblem $L_1 \cap L_2 \stackrel{?}{=} \emptyset$ für kontextfreie Sprachen reduzierbar, letzteres ist daher nicht entscheidbar.

Ebenfalls nicht entscheidbar ist: Ist eine CFG G ambig?

Sei nämlich $L(G_3) = L_1 \cup L_2$. Wegen der Abgeschlossenheit der CFGs unter Vereinigung ist auch G_3 eine CFG. Sie ist ambig genau dann, wenn ein Wort sowohl eine Ableitung in G_1 als auch in G_2 hat, denn diese sind notwendig verschieden. Dies ist aber dann so, wenn $L_1 \cap L_2 \neq \emptyset$.

Weiter sind unentscheidbar: $\overline{L(G)}$ kontextfrei? $L(G)$ regulär?

Für kontextsensitive Sprachen ist schon das Leerheitsproblem $L(G) \stackrel{?}{=} \emptyset$ nicht entscheidbar. Das liegt an der Abgeschlossenheit dieser Sprachen gegen Schnitt. Wäre das Leerheitsproblem für G entscheidbar, müsste es auch das Schnittproblem für $L(G_1)$ und $L(G_2)$ mit $L(G) = L(G_1) \cap L(G_2)$ sein.