Prof. Dr. Hans-Jörg Kreowski, Dr. Sabine Kuske
Studiengang Informatik
Universität Bremen
Linzer Str. 9a, OAS 3001, 3005
Tel.: 2956, 2335, 3697 (Sekr.), Fax: 4322
E-Mail: {kreo,kuske}@informatik.uni-bremen.de
www.informatik.uni-bremen.de/theorie

Oktober 2007

Theoretische Informatik I

Wintersemester 2007/2008

Inhaltsverzeichnis

1 Zum Sinn der theoretischen Informatik

Theoretische Informatik ist der Teil der Informatik, der sich systematisch mit mathematischen Mitteln erfassen und durchdringen lässt. Welchen Sinn das macht, wird in diesem Kapitel am Beispiel einer korrekten Modellierung eines Datenverarbeitungsproblems und seiner algorithmischen und somit auf dem Computer realisierbaren Lösung veranschaulicht. Alle angesprochenen und verwendeten Begriffe und Konzepte (sowohl technischer als auch mathematischer Art) setzen lediglich ein naives, intuitives Verständnis voraus, formale Definitionen werden noch nicht gebraucht.

In vielen Gebieten der Informatik geht es um die Modellierung von Datenverarbeitungsproblemen und ihren (korrekten) Lösungen.

$$\begin{tikzcd} \boxed{\text{Problem}} \arrow[r, bend left, "\text{Modellierung}"] & \boxed{\text{Lösung}} \arrow[l, bend left, "\text{Korrektheit}"] \end{tikzcd}$$

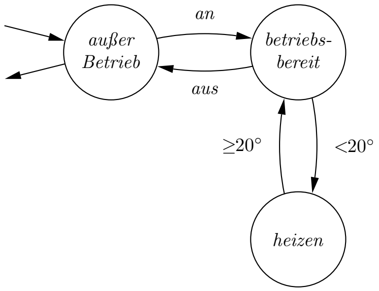
Am Beispiel eines sehr einfachen eingebetteten Systems soll illustriert werden, was das konkret heißen kann. Ziel ist die Modellierung einer Heizung (bzw. ihrer Steuerung), die heizt, wenn die Raumtemperatur zu niedrig ist. Etwas detaillierter ist folgende Aufgabe zu bewältigen:

Modelliere eine Heizung,

(1) die angeschaltet werden kann und dann betriebsbereit ist,
(2) die bei einer Temperatur unter 20° heizt,
(3) die wieder betriebsbereit wird, wenn die Temperatur mindestens 20° erreicht hat, und
(4) die ausgeschaltet werden kann, wenn sie nicht gerade heizt.

Ziel der Modellierung ist, diese informelle textuelle Beschreibung so zu präzisieren und zu explizieren, dass eine computerausführbare Lösung entsteht.

Der textuellen Beschreibung ist zu entnehmen, dass die Heizung mindestens drei verschiedene Zustände haben kann: *heizen*, *betriebsbereit* und *außer Betrieb*, wobei das den Zustand vor dem Anschalten und nach dem Ausschalten bezeichnet. Außerdem ist von vier Ereignissen die Rede, die eintreten können: anschalten (*an*), ausschalten (*aus*), Temperatur sinkt unter 20° (<20°), Temperatur steigt auf mindestens 20° (≥20°). Die Ereignisse ändern Zustände, was in der folgenden visuellen Darstellung festgehalten ist:



Die visuelle Darstellung ist ein sogenannter Zustandsgraph mit den Zuständen als Knoten, zu erkennen an den eingekreisten Namen, und den durch Ereignisse eintretenden Zustandsübergängen als Kanten, die als Pfeile gezeichnet sind mit dem Ereignis als Markierung am Pfeil, dem Pfeilende beginnend am Zustand vor dem Ereignis und der Pfeilspitze endend am Zustand nach dem Ereignis. Der Zustandsgraph präzisiert die textuelle Beschreibung in zweierlei Hinsicht:

(5) Der Zustand vor dem Anschalten ist derselbe wie nach dem Ausschalten.
(6) Dieser Zustand wird als Anfang und Ende aller Vorgänge im Zusammenhang mit der Heizung betrachtet, was einerseits durch den Pfeil, der von keinem Zustand ausgeht, und andererseits durch den Pfeil, der auf keinen Zustand zeigt, graphisch dargestellt ist.

Der Zustandsgraph spezifiziert die bei der modellierten Heizung *möglichen* Abläufe, die aus allen Ereignisfolgen bestehen, die vom Anfangs- zum Endzustand führen. Das sind alle Folgen $u_1 u_2 \dots u_n$ für $n \geq 1$, die aus *an*-*aus*-Zyklen u_i für $i = 1, \dots, n$ bestehen. Dabei ist ein *an*-*aus*-Zyklus eine Ereignisfolge, die mit *an* beginnt, mit *aus* endet und dazwischen immer abwechselnd <20° und ≥20° durchläuft, d.h. die Form

$$\mathit{an} <20^\circ \geq 20^\circ <20^\circ \geq 20^\circ \cdots <20^\circ \geq 20^\circ \mathit{aus}$$

hat. Außerdem wird die leere Folge als *möglich* betrachtet.

Bezeichnet man nun den Zustandsgraphen als *Heating* und die Menge aller möglichen Abläufe als $L(\mathit{Heating})$, dann kann man *Heating* als Modell betrachten, dessen Verhalten durch $L(\mathit{Heating})$ festgelegt ist und das damit die gestellte Aufgabe löst.

Aber ist die Lösung auch korrekt? Wird wirklich das gestellte Problem gelöst? Genau genommen ist *Heating* einfach konstruiert und dann zur Lösung erklärt worden. Dass alles richtig ist, bleibt der Intuition überlassen, was bei kleinen Problemen noch angeht. Aber was passiert bei Hunderten von Zuständen und Tausenden von Übergängen? Da kann die Intuition schnell versagen. Glücklicherweise kann man aber noch einen Schritt weiterkommen, indem man dem Lösungsmodell noch ein explizites Modell des Problems gegenüberstellt.

Um das von der bisherigen Beschreibung abzusetzen, soll es dadurch geschehen, dass angegeben wird, welche Ereignisfolgen bei der Heizung verboten sein sollen. Eine Ereignisfolge gilt als *verboten*, wenn sie mindestens eine der folgenden Teilfolgen enthält:

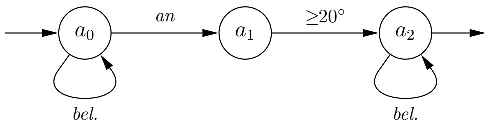
(a) *an* ≥20°
(b) <20° *aus*
(c) <20° <20°
(d) ≥20° ≥20°
(e) *an u an*
(f) *aus u aus*

wobei u eine Ereignisfolge ist, die nur <20° und ≥20° enthält. *Verboten* sind auch Folgen, die (g) nicht mit *an* beginnen oder (h) nicht mit *aus* enden. Wenn $L_{\mathit{forbidden}}$ die Menge aller verbotenen Ereignisfolgen bezeichnet, dann lässt sich diese Menge als Präzisierung des Problems durch negative Anforderungen auffassen, durch die ausgedrückt wird, was nicht passieren soll. Bezogen auf diese Anforderungen ist *Heating* ein korrektes Modell, wenn keine mögliche Ereignisfolge verboten ist. Mit anderen Worten ist *Heating korrekt* bzgl. $L_{\mathit{forbidden}}$, wenn $L(\mathit{Heating}) \cap L_{\mathit{forbidden}} = \emptyset$. Tatsächlich trifft das auch zu, denn vom Start aus muss man mit *an* beginnen (g) und zum Ende kommt man nur mit *aus* (h). Und ≥20° kann nur eintreten, wenn <20° direkt vorausgeht (a & d), so wie nach <20° nur ≥20° eintreten kann (b & c). Schließlich kann man *an* nur erhalten, wenn man mit *aus* zum Anfang zurückgekehrt ist (e), und ein weiteres *aus* setzt ein vorheriges *an* voraus (f).

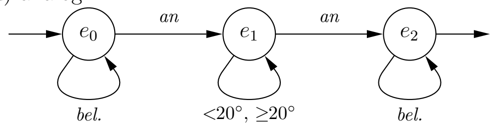
Insgesamt ist damit eine Modellierung gelungen, bei der das Problem durch die Menge $L_{\mathit{forbidden}}$ der verbotenen Ereignisfolgen präzisiert und die Lösung durch den Zustandsgraphen *Heating* modelliert ist, dessen Verhalten als die Menge $L(\mathit{Heating})$ der möglichen Ereignisfolgen bestimmt ist. Die Lösung ist in dem Sinne korrekt, dass keine bei *Heating* mögliche Ereignisfolge in der Problembeschreibung verboten ist.

Eine interessante Frage an dieser Stelle ist, ob Korrektheit automatisch überprüft werden kann. Es wird sich im Laufe der Lehrveranstaltung herausstellen, dass diese Frage bejaht werden kann, wenn man die negativen Anforderungen wie die Lösung mit Hilfe von Zustandsgraphen beschreibt.

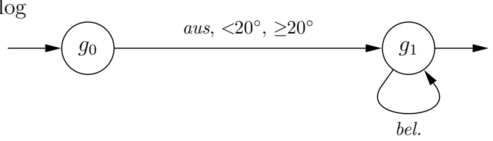
Die einzelnen Verbote im obigen Beispiel lassen sich sehr einfach durch Zustandsgraphen modellieren:

(a)


(b) – (d) analog

(e)


(f) analog

(g) 

(h) analog

Alle acht Zustandsgraphen, zusammengenommen und als Vereinigungsgraph betrachtet, ergeben im Prinzip einen Zustandsgraphen, der alle Verbote in sich vereinigt. Während allerdings bei Zustandsgraphen mehrere Endzustände erlaubt sind, wird meist verlangt, dass es nur einen Anfangszustand gibt. Das lässt sich aber auch erreichen, indem man noch einen neuen Zustand s_0 als einzigen Anfangszustand hinzunimmt (die bisherigen acht büßen diesen Status ein) und immer dann einen Zustandsübergang von s_0 nach x_i mit $x \in \{a, b, c, d, e, f, g, h\}$ und $i = 0, 1, 2$ vornimmt, wenn es einen von x_0 nach x_i bereits gibt.

Tatsächlich definieren Zustandsgraphen endliche Automaten, wie sie demnächst näher untersucht werden. Endliche Automaten gehören zu den einfachsten algorithmischen Instrumenten, die in der Informatik eine wichtige Rolle spielen. Endliche Automaten beschreiben Mengen von Wörtern, auch *Sprachen* genannt, wie beispielsweise die Mengen der möglichen und verbotenen Ereignisfolgen. Es wird unter anderem gezeigt, dass der Durchschnitt zweier solcher Sprachen wieder eine solche Sprache ist und dass die Leerheit einer solchen Sprache algorithmisch festgestellt werden kann, womit das Korrektheitsproblem in diesem Falle gelöst wäre. Diese Art der Verifikation wird *Model Checking* [MSS99, CS01] genannt und seit einigen Jahren äußerst erfolgreich in der Praxis eingesetzt.

In anderen Zusammenhängen ist es allerdings häufig sehr viel schwieriger Korrektheit sicherzustellen, so dass darauf oft verzichtet wird. Es muss jedoch beachtet werden, dass Modelle, die nicht sicher korrekt sind, Fehler machen können – teure Fehler oder vielleicht sogar fatale Fehler. Sich um Korrektheit zu bemühen, kann sich also lohnen. Sie setzt allerdings immer den Einsatz mathematischer Methoden voraus, weil nur dann ein echter Nachweis möglich ist. Denn während in dem kleinen Beispiel Korrektheit leicht einzusehen ist, wird die Angelegenheit äußerst unübersichtlich, wenn man es mit Systemen zu tun hat, die Hunderte von Zuständen besitzen und Hunderte oder Tausende von Verboten beachten müssen.

2 Lauter Wörter

Informatik ist ohne Zeichenketten, die in der Literatur oft auch Wörter genannt werden, undenkbar. Auch die Theorie ist stark auf sie angewiesen. Im vorigen Abschnitt spielen sie bereits als mögliche Abläufe, *an-aus*-Zyklen und verbotene Ereignisfolgen eine wichtige Rolle. Man muss sie hintereinander schreiben und Teilfolgen identifizieren können; man muss Gleichheit von Ereignisfolgen feststellen können. Den Benutzerinnen und Benutzern von Programmiersprachen sind Zeichenketten als Namen und Ausdrücke, als Dateien und Felder (Ketten konstanter Länge) u.ä. sowie in Gestalt der Programme selbst geläufig. Wörter und Sätze einer natürlichen Sprache wie Deutsch oder Englisch sind weitere wichtige Beispiele.

In diesem Abschnitt sind einige wichtige Informationen zum Umgang mit Zeichenketten zusammengestellt. Vieles davon wird verschiedentlich in die Überlegungen zur theoretischen Informatik während des kommenden Semesters eingehen, ohne dass diese Tatsache immer ausführlich gewürdigt wird.

2.1 Erzeugung von Wörtern

1. Um nicht bei jedem einzelnen Wort den Rahmen festlegen zu müssen, wird vorweg ein Zeichenvorrat A ausgewählt. A wird auch *Alphabet*, die Elemente von A werden *Zeichen* oder *Symbole* genannt.
2. *Wörter* (über A) sind rekursiv definiert durch:
   (a) λ ist ein *Wort*,
   (b) mit $x \in A$ und einem Wort v ist auch xv ein *Wort*.
3. Das initiierende Wort in (a) wird *leeres Wort*, der wiederholbare Vorgang in (b) *Linksaddition* (von x zu v) genannt. Für Wörter sind auch die Bezeichnungen Zeichenketten, Listen, Folgen, Sequenzen, Sätze, "files", Texte, Nachrichten, "strings" u.v.a.m. gebräuchlich. Die erste Bezeichnung wird im folgenden synonym für Wörter verwendet.
   Die Menge aller Wörter über A wird mit A^* bezeichnet. Für die Linksaddition $x\lambda$ von x zu λ wird kurz auch x geschrieben. In diesem Sinne gilt: $A \subseteq A^*$.
   Der Erzeugungsprozess für Wörter ist in Abbildung 1 illustriert.
4. Beispielsweise entsteht für das Alphabet $\{0, 1\}$ anfangs nur das leere Wort λ, weil der Teil (b) des Worterzeugungsprozesses nur verwendet werden kann, wenn schon Wörter vorhanden sind. Der Teil (a) muss nicht wieder angewendet werden, weil dadurch keine neuen Wörter mehr entstehen können. Die Anwendung von Teil (b) liefert nun zwei neue Wörter: $0\lambda, 1\lambda$ (bzw. $0, 1$ nach der Konvention zur Linksaddition mit dem leeren Wort). Darauf kann der Teil (b) erneut angewendet werden, was vier neue Wörter liefert: 00, 01, 10, 11. Durch Fortsetzung des Verfahrens (ad infinitum) entstehen nach und nach alle (endlichen) Bitstrings.
5. Die Erzeugung von Wörtern ist so gemeint, dass nur Gebilde, die durch den in Punkt 2 gegebenen rekursiven Prozess entstehen, Wörter über A sind und dass

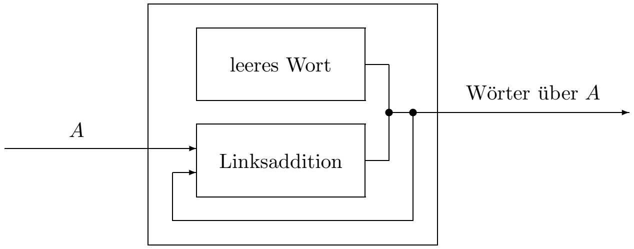


Abbildung 1: Rekursive Erzeugung von Wörtern

jedes Wort in eindeutiger Weise aus diesem Prozess hervorgeht. Letzteres bedeutet, dass für jedes Wort w entweder $w = \lambda$ gilt oder eindeutig $x \in A$ und ein Wort v mit $w = xv$ existieren. Im zweiten Fall wird x mit $head(w)$ und v mit $tail(w)$ bezeichnet. Bei dem Erzeugungsprozess wird stillschweigend vorausgesetzt, dass man durch die Bezeichnung λ ein Gebilde erhält, das sich von allen anderen Wörtern unterscheiden lässt, und dass das Nebeneinanderschreiben von Zeichen und Wörtern möglich ist. Beides mag intuitiv klar sein; es wird hier einfach vorausgesetzt.
Wenn das Alphabet selbst wieder Wörter enthält, wie z.B. $\{E, I, EI\}$, muss man mit dem Nebeneinanderschreiben aufpassen. Denn sonst kann beispielsweise EI ein Wort aus einem wie zwei Zeichen sein, was wegen der verlangten Eindeutigkeit verboten ist. In solchen Fällen muss man extra Vorsorge treffen, indem man die Zeichen in Hochkommata oder sonstige Klammern einschließt oder Zeichen und Wort beim Nebeneinanderschreiben durch ein Blank oder ein sonstiges Trennzeichen auseinanderhält.

6. Statt durch Linksaddition ließen sich alle Wörter auch durch Rechtsaddition erzeugen. In gewisser Weise wäre diese Alternative noch naheliegender, weil sie dem im europäischen Sprachraun verbreiteten Schreiben von links nach rechts entspricht.

Das durch diese Vereinbarungen verfügbare Textsystem ist noch recht bescheiden. Beginnend mit dem leeren Wort, kann jedes Wort von rechts nach links geschrieben werden; das zuletzt geschriebene Symbol kann gelesen (*head*) oder gelöscht (*tail*) werden. Wünschenswert wäre mehr Komfort, was mit den Punkten 2.2 und 2.4 ein Stück weit erreicht wird und sich analog noch wesentlich weitertreiben ließe.

2.2 Konkatenation

1. Analog zur Linksaddition und allgemeiner als diese lassen sich auch zwei Wörter v, w zu einem neuen Wort $v \cdot w$ zusammensetzen. $v \cdot w$ wird *Konkatenation* von v und w genannt und ist folgendermaßen rekursiv definiert:
   (a) für $v = \lambda$ gilt $\lambda \cdot w = w$,
   (b) für $v = xu$ mit $x \in A$ gilt $(xu) \cdot w = x(u \cdot w)$.

2. Die Linksaddition erweist sich als Spezialfall der Konkatenation:
$$xv \underset{1a}{=} x(\lambda \cdot v) \underset{1b}{=} (x\lambda) \cdot v \underset{2.1.3}{=} x \cdot v.$$
Das rechtfertigt die Schreibweise vw für $v \cdot w$.
3. Wie die Linksaddition ergibt sich auch die Rechtsaddition als Spezialfall der Konkatenation, indem man ein beliebiges Wort mit einem Symbol als zweites Argument konkateniert.
4. Die Konkatenation fügt zwei Wörter zusammen. Meist werden jedoch mehrere Konkatenationen kombiniert, z.B. ((((BE)I)(SP))((IE)L)). Das sieht hässlich aus; doch glücklicherweise kann man alle Klammern auch weglassen, weil die Reihenfolge, in der Wortteile verknüpft werden, keinen Einfluss auf das resultierende Wort hat. (Dagegen ist die Reihenfolge der Wortteile untereinander entscheidend, wie Punkt 2.5 zeigt).
Für alle Wörter u, v, w gilt $(uv)w = u(vw)$. Diese Eigenschaft ist auch als Assoziativität bekannt.
Die Behauptung ergibt sich für $u = \lambda$ nach Punkt 1a (in Verbindung mit der in Punkt 2 eingeführten Schreibweise):
$$(\lambda v)w = vw = \lambda(vw).$$
Für den Fall $u = xt$ mit $x \in A$ erhält man
$$((xt)v)w \underset{1b}{=} (x(tv))w \underset{1b}{=} x((tv)w) \underset{(*)}{=} x(t(vw)) \underset{1b}{=} (xt)(vw),$$
wenn die Gültigkeit der Aussage für t vorausgesetzt wird $(*)$. Das ist zulässig, da t um ein Zeichen kürzer ist als u, so dass die Eigenschaft für t als Induktionsvoraussetzung formuliert werden kann.

2.3 Induktionsprinzip

1. Wie in der vorangegangenen Überlegung liefert die rekursive Definition der Wörter ein für viele Situationen brauchbares Induktionsprinzip. Um eine Aussage THEOREM über Wörter zu beweisen, funktioniert oft folgendes Vorgehen:

Induktionsanfang (IA):	Zeige	THEOREM für $v = \lambda$.
Induktionsvoraussetzung (IV):	Nimm	THEOREM für v an.
Induktionsschluss (IS):	Zeige	THEOREM für xv mit $x \in A$.

2. Dieses Prinzip kann benutzt werden, um nachzuweisen, dass λ keinen Einfluss auf die Konkatenation hat. (Vergleiche Punkt 2.2.1a.)
Für alle Wörter v gilt $v\lambda = v$.
Denn es gilt:
$$\lambda\lambda \underset{2.2.1}{=} \lambda \text{ und } (xv)\lambda \underset{2.2.1}{=} x(v\lambda) \underset{(*)}{=} xv,$$
wobei $(*)$ die Anwendung der Induktionsvoraussetzung anzeigt.
Die oben gezeigte Assoziativität der Konkatenation beruhte bereits auf dem Induktionsprinzip. So lässt sich auch nachweisen, dass die Konkatenation eindeutig ist.

IA: Die Konkatenation von λ mit einem beliebigen Wort w liefert nach Definition dieses Wort, ist also eindeutig.

IV: Sei vw für zwei Wörter v, w ein eindeutig bestimmtes Wort.

IS: Betrachte nun $(xv)w$ für $x \in A$ und Wörter v, w.

Nach Definition gilt: $(xv)w = x(vw)$. Nach IV ist vw ein bestimmtes Wort, so dass nach den Festlegungen in Punkt 2.1.5 auch $x(vw)$ eindeutig bestimmt ist.

3. Allgemein lassen sich mit dem Induktionsprinzip Aussagen über alle Wörter eines Alphabets beweisen. Wenn mehrere Wörter in einer Aussage vorkommen und allquantifiziert sind, kann man sich eins aussuchen, aber nicht jede Wahl klappt gleich gut.
4. Das Indukttionsprinzip lässt sich analog für Wörter formieren, die mittels Rechtsaddition aufgebaut sind. Dabei ändert sich nur der Induktionsschlußin "Zeige THEOREM für vx mit $x \in A$".

2.4 Gleichheitstest, Länge und Zeichenzählen

1. Die eindeutige Zerlegbarkeit von Wörtern gemäß Punkt 2.1.5 gestattet auch, in einfacher Weise die Gleichheit zweier Wörter rekursiv festzustellen.
Zwei Wörter v, w sind *gleich* (in Zeichen: $v \equiv w$), wenn sie beide leer sind: $v = \lambda = w$, oder wenn sie beide nicht leer sind sowie $head(v) \equiv head(w)$ und $tail(v) \equiv tail(w)$, wobei ein Gleichheitstest auf dem Alphabet, der ebenfalls mit $\equiv$ bezeichnet wird, vorausgesetzt ist.
2. Ebenfalls rekursiv bestimmt werden kann, wie lang ein Wort ist und wie oft ein bestimmtes Zeichen darin vorkommt:

   (a) $length(\lambda) = 0$
   (b) $length(xv) = length(v) + 1$ für $x \in A, v \in A^*$
   (c) $count(x, \lambda) = 0$
   (d) $count(x, yv) =$ if $x \equiv y$ then $count(x, v) + 1$ else $count(x, v)$
   für $x, y \in A, v \in A^*$.

   Die in (d) verwendete Fallunterscheidung funktioniert wie üblich: Ist die Abfrage wahr, so wird der then-Teil wirksam, sonst der else-Teil.
   Dass durch (a) und (b) eine Abbildung $length\colon A^* \to \mathbb{N}$ definiert wird, lässt sich mit Hilfe des Induktionsprinzips zeigen:

   IA: $length(\lambda) = 0$ ist genau ein Wert aus $\mathbb{N}$ für λ.

   IV: Für ein Wort v sei $length(v)$ genau eine natürliche Zahl.

   IS: Betrachte nun xv für $x \in A$. Nach (b) gilt $length(xv) = length(v) + 1$. Nach IV ist $length(v)$ genau eine natürliche Zahl, so dass der Nachfolger auch genau eine natürliche Zahl ist.

   Analog erweist sich auch $count\colon A \times A^* \to \mathbb{N}$ als Abbildung.

3. Auf dieser Basis lassen sich auch diverse weitere Eigenschaften der eingeführten Operationen zeigen. Als Beispiel wird mit vollständiger Induktion bewiesen, dass die Länge einer Konkatenation gerade die Summe der Längen der Einzelwörter ist, d.h. es gilt für alle $v, w \in A^*$:

$$length(vw) = length(v) + length(w)$$

   IA: $length(\lambda w) = length(w) = 0 + length(w) = length(\lambda) + length(w)$.
   Dabei werden nacheinander die Definition der Konkatenation, eine bekannte arithmetische Eigenschaft und die Definition der Länge ausgenutzt.

   IV: Die Behauptung gelte für v und beliebige w.

   IS: Betrachte av mit $a \in A$ (und beliebiges w):

$$\begin{aligned} length((av)w) &= length(a(vw)) \\ &= length(vw) + 1 \\ &= length(v) + length(w) + 1 \\ &= length(v) + 1 + length(w) \\ &= length(av) + length(w). \end{aligned}$$

   Dabei werden nacheinander die Definition der Konkatenation, die Definition der Länge, die Induktionsvoraussetzung, eine arithmetische Eigenschaft und wieder die Definition der Länge ausgenutzt.

2.5 Iterative Darstellung

Die eindeutige Zerlegbarkeit von Wörtern nach Punkt 2.1.5 ergibt zusammen mit dem Induktionsprinzip eine sehr wichtige, gebräuchliche und vertraute Darstellung von Wörtern.

Für jedes Wort w existieren eindeutig $n \in \mathbb{N}$ und $x_i \in A$ für $i = 1, \ldots, n$ mit $w = x_1 \cdots x_n$.

Das schließt das leere Wort mit ein. In diesem Falle ist $n = 0$, und $x_1 \cdots x_0$ steht für λ. Insgesamt lässt sich also jedes Wort eindeutig in Zeichen als elementare Bausteine zerlegen. Auf den einfachen Beweis wird verzichtet.

2.6 Ansichten von der Menge aller Wörter

Während die vorausgegangenen Informationen über Wörter unverzichtbar für die weiteren Überlegungen sind, dient dieser Abschnitt zur Abrundung. Es wird gezeigt, dass sich die Menge aller Wörter in verschiedenen Formen darstellen lässt. In Punkt 1 wird ein interessanter Zusammenhang zur Algebra und universellen Algebra hergestellt. Punkt 2 charakterisiert A^* durch eine sogenannte Bereichsgleichung. Solche Gleichungen sind in der denotationellen Semantik von Programmiersprachen gebräuchlich. Die Punkte 3 und 4 liefern eine iterative Darstellung von Wörtern, die in der Literatur am häufigsten anzutreffen ist. Die in diesem Kapitel gewählte Einführung wird *axiomatisch* genannt. In

Punkt 5 wird die Nähe zu den berühmten Peano-Axiomen der natürlichen Zahlen aufgedeckt. Diese Art des Zugangs eignet sich besonders für den weiteren Umgang mit Wörtern nach Art der funktionalen Programmierung.

1. Die Konkatenation gemäß Punkt 2.2.1 bestimmt eine Abbildung $\cdot : A^* \times A^* \to A^*$, die nach Punkt 2.2.3 assoziativ ist und deshalb A^* zu einem *Monoid* macht mit dem leeren Wort als neutralem Element (vgl. Punkte 2.2.1a und 2.3.2). Da jedes Wort nach Punkt 2.5 eindeutig in "Atome" zerfällt, erweist sich A^* sogar als *freies Monoid*.
2. Die in Punkt 2.1.5 formulierte Zerlegbarkeitseigenschaft der Linksaddition lässt sich explizit dadurch erreichen, dass xv als Paar (x, v) geschrieben wird. Unter Verwendung der Mengenoperationen *disjunkte Vereinigung* $+$ und *kartesisches Produkt* $\times$ erfüllt demnach die Menge A^* aller Wörter über A die Gleichung

   $$A^* = \{\lambda\} + (A \times A^*).$$

   A^* ist sogar die kleinste Menge mit dieser Eigenschaft. Deshalb ließe sich diese Mengengleichung auch zur Definition von A^* und damit von Wörtern über A heranziehen.
3. Eine weitere Möglichkeit, A^* zu definieren, die häufig in der Literatur zu finden ist, besteht darin, Wörter direkt als beliebige Tupel von atomaren Zeichen zu wählen:

   (a) λ ist ein Wort,
   (b) für $n > 0$ und $x_i \in A$ $(i = 1, \ldots, n)$ ist $w = x_1 \cdots x_n$ ein Wort.

   In Tupelschreibweise lautet diese Definition:

   (c) das leere Tupel () ist ein Wort,
   (d) das n-Tupel $(x_1, \ldots, x_n)$ mit $n > 0$, $x_i \in A$, $i = 1, \ldots, n$ ist ein Wort.
4. Beachtet man, dass n-Tupel wie in (b) bzw. (d) von Punkt 3 Elemente des n-fachen kartesischen Produkts A^n sind, ergibt sich für die Menge aller Wörter über A folgende Darstellung

   $$A^* = \sum_{i=0}^{\infty} A^i,$$

   wobei Σ die disjunkte Vereinigung bezeichnet.
   Korrespondierend zur Linksaddition lassen sich die Potenzen A^i iterieren:

   $$A^0 = \{\lambda\} \text{ und } A^{i+1} = A \times A^i \text{ für } \; i \in \mathbb{N}.$$
5. Betrachtet man speziell das einelementige Alphabet $\{|\}$, so entstehen als Wörter Strichfolgen beliebiger Länge, wobei jedes Wort bereits eindeutig durch seine Länge festgelegt ist. Das leere Wort kann also als 0 gesehen werden, und die Linksaddition entspricht der Nachfolgerfunktion natürlicher Zahlen. Die Strichdarstellung natürlicher Zahlen ist als Bierdeckel-Arithmetik bekannt. Spezialisiert man außerdem das Induktionsprinzip für Wörter auf diesen Fall, erhält man ein bekanntes Induktionsprinzip für natürliche Zahlen. Insgesamt erweist sich also der in diesem Kapitel

gewählte Zugang zu Wörtern als Verallgemeinerung der durch die Peano-Axiome definierten natürlichen Zahlen.

Es sei noch folgendes angemerkt. Ergänzte man die Erzeugung von Wörtern explizit um ihre Länge:

(a) λ ist ein Wort der Länge 0,
(b) xv ist ein Wort der Länge $n + 1$, falls $x \in A$ und v ein Wort der Länge $n \in \mathbb{N}$ ist,

so ließe sich das Induktionsprinzip in 2.3 durch vollständige Induktion über die Länge von Wörtern beweisen.

3 Endliche Automaten

In Kapitel 1 wird exemplarisch eine typische Situation in der Datenverarbeitung mit Hilfe von Zustandsgraphen modelliert. Zustandsgraphen sind graphische Darstellungen endlicher Automaten, die in diesem Kapitel formal eingeführt werden. Sie bilden ein in der Informatik häufig und sehr erfolgreich eingesetztes Modellierungswerkzeug, das vergleichsweise einfach ist und sich deshalb als Einstieg in den Bereich der formalen Modellierungsmethoden eignet. Ein endlicher Automat spezifiziert in seiner einfachsten Form eine Sprache, d.h. eine Menge von Wörtern. Als ein erstes interessantes Ergebnis stellt sich heraus, dass sich jeder endliche Automat deterministisch machen lässt, ohne seine Sprache zu verändern. Das ist deshalb signifikant, weil ein deterministischer Automat sehr schnell erkennen kann, ob ein Wort zu seiner Sprache gehört oder nicht.

Unter einem endlichen Automaten hat man sich ein taktweise arbeitendes System vorzustellen, das sich zu jedem Zeitpunkt in einem bestimmten Zustand (von endlich vielen verfügbaren Zuständen) befindet und das dann innerhalb eines Taktes einen Buchstaben eines Eingabewortes einliest, daraufhin seinen Zustand ändert und den Lesekopf auf den nächsten Eingabebuchstaben rechts einstellt usw. Am Anfang befindet sich der Automat in einem Anfangszustand und der Lesekopf steht ganz links auf dem ersten Buchstaben des Eingabewortes. Ein Eingabewort gilt dann als erkannt, wenn vom Anfangszustand aus nach Verarbeitung des Wortes ein Endzustand erreicht wird. Die erkannten Wörter bilden die Sprache des Automaten.

3.1 Endlicher Automat, fortgesetzte Zustandsüberführung und erkannte Sprache

Die Veranschaulichung führt zu folgender Definition:

1. Ein *endlicher relationeller erkennender Automat* – kurz *endlicher Automat* – ist ein System $A = (Z, I, d, s_0, F)$, wobei
   - Z eine endliche Menge von *Zuständen* ist,
   - I ein endliches *Eingabealphabet*,
   - $s_0 \in Z$ der *Anfangszustand*,
   - $d \subseteq Z \times I \times Z$ eine Relation ist, geschrieben $d\colon Z \times I \rightsquigarrow Z$, die *Zustandsüberführung* genannt wird und jedem Zustand und jeder Eingabe eine Menge von Folgezuständen zuordnet und
   - $F \subseteq Z$ eine Menge von *Endzuständen* ist.
2. Die *Zustandsüberführung* d lässt sich rekursiv zu einer Relation $d^* \subseteq Z \times I^* \times Z$ bzw. $d^*\colon Z \times I^* \rightsquigarrow Z$ fortsetzen, die nicht nur Zeichen, sondern Eingabewörter verarbeitet:
   (i) $d^*(s, \lambda) = \{s\}$ für alle $s \in Z$ und
   (ii) $d^*(s, wx) = \bigcup_{t \in d^*(s,w)} d(t, x)$ für alle $s \in Z$, $x \in I$, $w \in I^*$.
   Die Eingabewörter sind hierbei rekursiv von rechts nach links aufgebaut.
3. Die von einem endlichen Automaten A *erkannte Sprache* $L(A)$ ist dann definiert durch:
   $$L(A) = \{w \in I^* \mid d^*(s_0, w) \cap F \neq \emptyset\}.$$
4. Ein Automat $A = (Z, I, d, s_0, F)$ heißt *deterministisch*, wenn d eine Abbildung ist; d.h. für alle $s \in Z$ und $x \in I$ existiert genau ein $s' \in Z$ derart, dass (s, x) und s' bzgl. d in Relation stehen. In diesem Fall wird auch $d\colon Z \times I \to Z$ und $d(s, x) = s'$ geschrieben, wie es für Abbildungen üblich ist.

Bemerkung zur Relationsschreibweise

Ist $R\colon A \rightsquigarrow B$ eine Relation (d.h. $R \subseteq A \times B$), dann wird statt $(a, b) \in R$ oft auch aRb geschrieben oder, wenn R eine Abbildung ist, $R(a) = b$. Ansonsten bezeichnet $R(a)$ die Menge aller Elemente, die bzgl. R zu a in Relation stehen: $R(a) = \{b \in B \mid aRb\}$.

3.2 Fortgesetzte Zustandsüberführung und erkannte Sprache von deterministischen Automaten

Für deterministische Automaten ist mit d auch d^* eine Abbildung, wie man leicht durch Induktion über w zeigt. Deshalb können (i) und (ii) aus Punkt 2 in diesem Fall auch ausgedrückt werden durch:

(i') $d^*(s, \lambda) = s$ und
(ii') $d^*(s, wx) = d(d^*(s, w), x)$.

Ferner ist die von deterministischen Automaten erkannte Sprache bestimmt durch:

$$L(A) = \{w \in I^* \mid d^*(s_0, w) \in F\}.$$

3.3 Zustandsgraph

Jeder endliche Automat A besitzt eine graphische Darstellung in Form eines *Zustandsgraphen*. Dabei werden die Zustände zu Knoten, und von einem Zustand s zu einem Zustand s' wird eine mit $x \in I$ markierte Kante gezogen, falls $s' \in d(s, x)$. Falls es mehrere Zeichen $x_1, \ldots, x_k \in I$ gibt mit $s' \in d(s, x_i)$ für alle $i \in \{1, \ldots, n\}$, so wird der Übersichtlichkeit halber nur eine Kante gezogen, an der dann $x_1, \ldots, x_k$ steht. Der Anfangszustand wird durch einen hineingehenden Pfeil, die Endzustände werden entsprechend durch herausgehende Pfeile gekennzeichnet.

3.4 Beispiel

In Abbildung 2 ist ein kleiner endlicher Automat dargestellt. Er erkennt gerade die Sprache der positiven ganzen Zahlen in Binärdarstellung ohne führende Nullen, d.h. die Menge $\{1w \mid w \in \{0,1\}^*\}$. Dieser Automat ist nichtdeterministisch; denn für die Zustandsüberführung d gilt: $d(s_0, 1) = \{s_0, s_1\}$ und $d(s_0, 0) = \emptyset$.

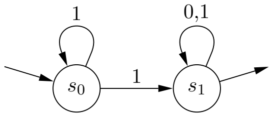


Abbildung 2: Graphische Darstellung eines endlichen Automaten

3.5 Verarbeitung von Wörtern in iterativer Darstellung

Die fortgesetzte Zustandsüberführung ist für Eingabewörter rekursiv definiert. Es gibt aber auch eine iterative Version, die vielleicht etwas anschaulicher ist, vor allem aber in manchen Situationen besser zu gebrauchen.

Sei $A = (Z, I, d, s_0, F)$ ein endlicher Automat und $w = a_1 \cdots a_n \in A^*$ mit $a_i \in A$ für $i = 1, \dots, n$. Dann ist $s' \in d^*(s, w)$ für $s, s' \in Z$ genau dann, wenn es eine Folge von Zuständen $t_0, \dots, t_n$ gibt, derart dass $s = t_0$, $s' = t_n$ und $t_i \in d(t_{i-1}, a_i)$ für $i = 1, \dots, n$.

Dabei ist der Fall $n = 0$ zugelassen und betrifft das leere Wort $w = a_1, \dots, a_0 = \lambda$ und die Zustandsfolge t_0 mit $s = t_0 = s'$.

Beweisen lässt sich das mit Induktion über die Struktur oder Länge der Eingabewörter, worauf hier allerdings verzichtet wird.

Im Zustandsgraph sieht die Verarbeitung eines Eingabewortes $w = a_1 \cdots a_n$ demnach so aus:

3.6 Schnelle Spracherkennung durch deterministische Automaten

Mit jeder Überführung des Zustandes eines deterministischen Automaten in den (eindeutigen) Nachfolgezustand wird ein Buchstabe des Eingabewortes abgearbeitet. Erst wenn

das Eingabewort vollständig durchlaufen ist, bleibt der Automat stehen. Es sind also n Schritte erforderlich, um zu entscheiden, ob ein Eingabewort der Länge n zur erkannten Sprache gehört. Damit ist das sogenannte Wortproblem für jede von einem deterministischen Automaten erkannte Sprache in linearer Zeit lösbar.

Gilt das auch, wenn der erkennende Automat nichtdeterministisch ist?

3.7 Der Potenzautomat

Offensichtlich ist Determinismus eine echte Einschränkung für endliche Automaten. Ist damit aber auch die Klasse der Sprachen, die von deterministischen Automaten erkannt werden, eine echte Teilmenge der von allgemeinen endlichen Automaten erkannten Sprachen? Das folgende Theorem verneint dies.

Theorem 1
Zu jedem endlichen Automaten A lässt sich effektiv ein deterministischer Automat $\mathcal{P}(A)$ konstruieren, der dieselbe Sprache erkennt; d.h.

$$L(A) = L(\mathcal{P}(A)).$$

Beweis.
Sei $A = (Z, I, d, s_0, F)$ ein endlicher Automat.

Daraus lässt sich der *Potenzautomat* konstruieren:

$$\mathcal{P}(A) = (\mathcal{P}(Z), I, D, \{s_0\}, F_{\mathcal{P}} := \{S \subseteq Z \mid S \cap F \neq \emptyset\}),$$

wobei $\mathcal{P}(Z)$ die Potenzmenge von Z ist und die Abbildung $D \colon \mathcal{P}(Z) \times I \to \mathcal{P}(Z)$ definiert ist durch $D(S, x) := \bigcup_{s \in S} d(s, x)$ für alle $S \in \mathcal{P}(Z)$ und $x \in I$.

Dann stehen die fortgesetzten Zustandsüberführungen d^* und D^* in der Beziehung

$$d^*(s, w) = D^*(\{s\}, w).$$

Denn für $w = \lambda$ gilt:

$$d^*(s, \lambda) = \{s\} = D^*(\{s\}, \lambda),$$

und für Wörter wx erhält man, vorausgesetzt, die Behauptung gilt bereits für w:

$$\begin{aligned} d^*(s, wx) &= \bigcup_{t \in d^*(s,w)} d(t, x) \\ &= D(d^*(s, w), x) \\ &= D(D^*(\{s\}, w), x) = D^*(\{s\}, wx). \end{aligned}$$

Für die Sprachen $L(A)$ und $L(\mathcal{P}(A))$ folgt somit:

$$\begin{aligned} w \in L(A) \quad &\text{gdw.} \quad d^*(s_0, w) \cap F \neq \emptyset \\ &\text{gdw.} \quad D^*(\{s_0\}, w)(= d^*(s_0, w)) \in F_{\mathcal{P}} \\ &\text{gdw.} \quad w \in L(\mathcal{P}(A)); \end{aligned}$$

$L(A)$ und $L(\mathcal{P}(A))$ sind also gleich. □

3.8 Beispiel

Der Potenzautomat zu dem endlichen Automaten aus Abbildung 2 ist in Abbildung 3 dargestellt. Zum Beispiel ist $D(\{s_0, s_1\}, 0) = d(s_0, 0) \cup d(s_1, 0) = \emptyset \cup \{s_1\} = \{s_1\}$ sowie $D(\{s_0, s_1\}, 1) = d(s_0, 1) \cup d(s_1, 1) = \{s_0, s_1\} \cup \{s_1\} = \{s_0, s_1\}$ und immer $D(\emptyset, x) = \emptyset$.

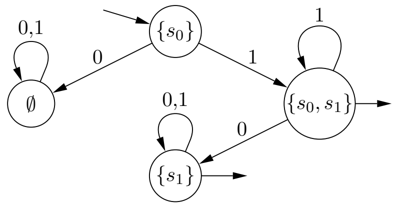


Abbildung 3: Der Potenzautomat zu dem endlichen Automaten aus Abbildung 2

3.9 Schnelle Spracherkennung durch endliche Automaten

Mit dem Theorem 1 folgt sofort, dass das Wortproblem für jede von einem beliebigen endlichen Automaten erkannte Sprache linear lösbar ist. Denn man kann erst den Potenzautomaten konstruieren, was nur konstant viel Zeit in Anspruch nimmt (zumindest bezogen auf die Länge der Eingabewörter). Danach ist das Wortproblem linear lösbar, weil der Potenzautomat deterministisch ist, aber immer noch dieselbe Sprache erkennt.

4 Produktautomat erkennt Durchschnitt

Wie im Kapitel 1 versprochen, kann ein endlicher Automat konstruiert werden, der den Durchschnitt zweier Sprachen erkennt, die selbst von endlichen Automaten erkannt werden. Eine einfache Modifikation der Endzustände befähigt diesen sogenannten Produktautomaten ebenfalls, die Vereinigung der beiden Sprachen zu erkennen.

Die Idee des Produktautomaten ist, zwei endliche Automaten über das kartesische Produkt ihrer Zustandsmengen so zu koppeln, dass sie Eingabewörter parallel abarbeiten.

4.1 Produktautomat

Seien $A_i = (Z_i, I, d_i, s_{0i}, F_i)$ für $i = 1, 2$ zwei deterministische Automaten. Dann ist der *Produktautomat* definiert durch

$$A_1 \times A_2 = (Z_1 \times Z_2, I, d, (s_{01}, s_{02}), F_1 \times F_2)$$

mit $d((s_1, s_2), x) = (d_1(s_1, x), d_2(s_2, x))$ für alle $(s_1, s_2) \in Z_1 \times Z_2$ und $x \in I$.

Der Produktautomat führt also die Zustandsübergänge der beiden Einzelautomaten parallel durch. Wie das Lemma im folgenden Abschnitt zeigt, gilt das auch für die fortgesetzte Zustandsüberführung.

4.2 Erkennung des Durchschnitts

Da der Anfangszustand des Produktautomaten aus den beiden einzelnen Anfangszuständen und die Endzustände Paare der einzelnen Endzustände sind, ergibt sich aus dem folgenden Lemma 4.1 die gewünschte Durchschnittseigenschaft.

Theorem 2

$$L(A_1 \times A_2) = L(A_1) \cap L(A_2).$$

Beweis.
$w \in L(A_1 \times A_2)$ gdw. nach Lemma 4.1

$$d^*((s_{01}, s_{02}), w) = (d_1^*(s_{01}, w), d_2^*(s_{02}, w)) \in F_1 \times F_2$$

gdw. $d_i^*(s_{0i}, w) \in F_i$ für $i = 1, 2$ gdw. $w \in L(A_i)$ für $i = 1, 2$. □

Lemma 4.1

$$d^*((s_1, s_2), w) = (d_1^*(s_1, w), d_2^*(s_2, w)) \text{ für alle } (s_1, s_2) \in Z_1 \times Z_2 \text{ und } w \in I^*.$$

Beweis.
Der Beweis wird mit vollständiger Induktion über den Aufbau von w geführt.

IA: $d^*((s_1,s_2),\lambda) = (s_1,s_2) = (d_1^*(s_1,\lambda), d_2^*(s_2,\lambda))$, wobei die Definition der fortgesetzten Zustandsüberführung verwendet wird.

IS: $d^*((s_1,s_2),wx) = d(d^*((s_1,s_2),w),x) =$
$d((d_1^*(s_1,w), d_2^*(s_2,w)),x) =$
$(d_1(d_1^*(s_1,w),x), d_2(d_2^*(s_2,w),x)) =$
$(d_1^*(s_1,wx), d_2^*(s_2,wx))$.

Dabei ergeben sich die Gleichheiten in der gegebenen Reihenfolge aus der Definition der fortgesetzten Zustandsüberführung, der Induktionsvoraussetzung, der Definition der Zustandsüberführung des Produktautomaten und erneut aus der Definition der fortgesetzten Zustandsüberführung. □

4.3 Erkennung der Vereinigung

Wenn beim Abarbeiten eines Eingabewortes einer der beiden Automaten A_1 und A_2 in einen Endzustand kommen, dann gehört das Wort gerade zu der Vereinigung der erkannten Sprachen. Nach Lemma 4.1 erkennt der Produktautomat aber genau diese Vereinigung $L(A_1)) \cup L(A_2)$, wenn man die Menge $(F_1 \times Z_2) \cup (Z_1 \times F_2)$ als Endzustände wählt.

Wie muss man Endzustände wählen, damit der Produktautomat die Differenzsprachen $L(A_1) \setminus L(A_2)$ und $L(A_2) \setminus L(A_1)$ erkennt?

5 Entscheidbarkeit des Leerheitsproblems

Nach den Überlegungen in Kapitel 1 muss nur noch gezeigt werden, dass sich algorithmisch feststellen lässt, ob eine von einem endlichen Automaten erkannte Sprache leer ist oder nicht, um Korrektheit nachzuweisen. Denn die Sprache des Automaten, der das System modelliert, enthält alle möglichen Abläufe und arbeitet korrekt, wenn kein möglicher Ablauf verboten ist. Wenn also die verbotenen Sequenzen auch durch einen endlichen Automaten erkannt werden, muss man nach Kapitel 4 nur den Produktautomaten konstruieren und diesen auf Leerheit seiner Sprache testen.

Sei $A = (Z, I, d, s_o, F)$ ein deterministischer Automat. Sei

$$H(A) = \{s \in Z \mid d^*(s,w) \in F \text{ für ein } w \in I^*\}$$

die Menge aller Zustände, die zu Endzuständen führen. Dann gilt offensichtlich:

$$L(A) \neq \emptyset \text{ gdw. } s_0 \in H(A).$$

$H(A)$ lässt sich folgermaßen iterieren:

$H_0 = F$ und $H_{i+1} = H_i \cup \{s \in Z \mid d(s,x) \in H_i \text{ für ein } x \in I\}$ sowie $H(A) = H_m$ für das kleinste m mit $H_{m+1} = H_m$.

Nach Definition gilt: $F = H_0 \subseteq H_1 \subseteq \cdots \subseteq H_i \subseteq H_{i+1} \subseteq \cdots \subseteq Z$. Da Z endlich ist, können nur endlich viele dieser Inklusionen echt sein, so dass es m geben muss. Dann gilt auch $H_{m+k} = H_m$ für alle $k \in \mathbb{N}$ (einfache Induktion über k). Daraus folgt die gewünschte Gleichheit. Denn mit Induktion über i ergibt sich $H_i \subseteq H(A)$:

IA: $H_0 = F \subseteq H(A)$ wegen $d^*(s,\lambda) = s \in F$ für alle $s \in F$.
IS: $H_{i+1} = H_i \cup \{s \in Z \mid d(s,x) \in H_i \text{ für ein } x \in I\}$
$\subseteq H(A) \cup \{s \in Z \mid d(s,x) \in H(A) \text{ für ein } x \in I\}$
$= H(A) \cup \{s \in Z \mid d^*((d(s,x),w)) \in F \text{ für ein } w \in I^*, x \in I\}$
$= H(A) \cup \{s \in Z \mid d^*(s,xw) \in F \text{ für ein } xw \in I^*\}$
$\subseteq H(A) \cup H(A) = H(A)$.
Insbesondere gilt $H_m \subseteq H(A)$.

Betrachte umgekehrt $s \in H(A)$. Dann gibt es $w \in I^*$ mit $d^*(s,w) \in F$. Das impliziert $s \in H_{length(w)}$, wie eine einfache Induktion über den Aufbau von w ergibt. Daraus folgt wie gewünscht: $s \in H_{length(w)} \subseteq H_m$.

6 Reguläre Sprachen und reguläre Ausdrücke

Neben dem Erkennen von Sprachen ist es interessant, ihre Kompositionalität oder Modularität zu untersuchen. Dabei geht es darum, wie sich umfangreiche und komplizierte Sprachen aus einfachen Bausteinen aufbauen lassen. Diese Frage ist auch für die Entwicklung großer Spezifikationen und informationstechnischer Systeme von zentraler Bedeutung. Es zeigt sich, dass sich die von endlichen Automaten erkannten Sprachen in besonders einfacher Weise modular aufbauen lassen.

Dieses Verhalten wird in vier Schritten verwirklicht. Im Abschnitt 6.1 wird zuerst eine modulare Komposition von Sprachen eingeführt, die zu den sogenannten regulären Sprachen führt. Diese Komposition lässt sich auch auf der Ebene der endlichen Automaten realisieren, so dass man für jede reguläre Sprache einen endlichen Automaten konstruieren kann, der sie erkennt, was im Abschnitt 6.2 gezeigt wird. Im Abschnitt 6.3 wird dann in einem nicht ganz einfachen Beiweis hergeleitet, dass jede von einem endlichen Automaten erkannte Sprachen schon selbst regulär ist, sich also modular aufbauen lässt. Im Abschnitt 6.4 schließlich werden reguläre Ausdrücke eingeführt, die eine syntaktische Beschreibung regulärer Sprachen liefern.

6.1 Reguläre Sprachen

Reguläre Sprachen lassen sich durch bestimmte Sprachoperationen aus kleinen, unzerlegbaren Bausteinen modular aufbauen. Diese kleinsten Sprachmoduln sind die leere Menge $\emptyset$, die einelementige Sprache $\{\lambda\}$, die das leere Wort enthält sowie die einelementige Sprache $\{x\}$ für jedes Wort x der Länge 1. Die aufbauenden Sprachoperationen sind die Vereinigung, die Konkatenation und die Kleene-Hülle. Die *Konkatenation* zweier Sprachen L_1 und L_2 wird mit L_1L_2 bezeichnet und besteht aus allen Wörtern, die man erhält, wenn man ein Wort der zweiten Sprache hinter ein Wort der ersten Sprache hängt, d.h. $L_1L_2 = \{w_1w_2 \mid w_1 \in L_1,\, w_2 \in L_2\}$. Die *Kleene-Hülle* einer Sprache L wird mit L^* bezeichnet und besteht aus allen Wörtern, die sich durch beliebig häufiges Aneinanderreihen von Wörtern aus L konstruieren lassen, d.h. $L^* = \bigcup_{i\in\mathbb{N}} L^i$ mit $L^0 = \{\lambda\}$ und $L^{i+1} = L^iL$.

Formal lässt sich die Klasse der regulären Sprachen wie folgt beschreiben:

Sei I ein endliches Alphabet. Dann ist die Klasse $\mathcal{L}_{REG(I)}$ der *regulären Sprachen über I* rekursiv definiert durch:

1. $\emptyset, \{\lambda\}, \{x\} \in \mathcal{L}_{REG(I)}$ für $x \in I$,
2. $L, L_1, L_2 \in \mathcal{L}_{REG(I)}$ impliziert $L_1 \cup L_2$, L_1L_2, $L^* \in \mathcal{L}_{REG(I)}$.

Die regulären Sprachen sind also vollständig modular aufgebaut.

Beispiele

1. Das Alphabet I ist eine reguläre Sprache, da man es mittels Vereinigung aller Bausteine $\{x\}$ $(x \in I)$ erhält, d.h. $I = \cup_{x\in I}\{x\}$. Mit der Operation der Kleene-Hülle ergibt sich unmittelbar, dass die Menge aller Wörter über I ebenfalls regulär ist, d.h. $I^* \in \mathcal{L}_{REG(I)}$. Da I endlich ist, erhält man auf die gleiche Weise für jede Teilmenge M von I die Regularität von M und M^*. Insbesondere ist mit $a \in I$ die Sprache L_a aller Wörter, die nur aus a's bestehen, regulär, d.h. $L_a = \{a^n \mid n \in \mathbb{N}\} \in \mathcal{L}_{REG(I)}$, denn $L_a = \{a\}^*$.
2. Für alle $b \in I$ ist die Sprache $L_{2b} = \{b^{2n} \mid n \in \mathbb{N}\}$ regulär, denn $L_{2b} = (\{b\}\{b\})^*$, d.h. L_{2b} setzt sich mittels Konkatenation und Kleene-Hülle aus der regulären Sprache $\{b\}$ zusammen.
3. Die Sprache $L_{a2b} = \{a^ib^{2j} \mid i,j \in \mathbb{N}\}$ ist regulär, denn $L_{a2b} = L_aL_{2b}$.
4. Die Regularität der Sprache $L_{a+} = \{a^n \mid n \geq 1\}$ kann man sich an der Gleichheit von L_{a+} und $\{a\}L_a$ klarmachen.
5. Die Sprache $L_{even(b)} = \{w \in \{a,b\}^* \mid count(b,w) \mod 2 = 0\}$ ist regulär, denn es gilt $L_{even(b)} = L_a \cup (L_a\{b\}L_a\{b\}L_a)^*$. Auf analoge Weise kann man die reguläre Sprache $L_{even2(b)} = \{w \in I^* \mid count(b,w) \mod 2 = 0\}$ konstruieren, nämlich $L_{even2(a)} = M^* \cup (M\{b\}M\{b\}M)^*$ mit $M = I \setminus \{b\}$; denn nach Punkt 1 ist M^* regulär.
6. Auch die Sprache L_{even} aller Wörter gerader Länge ist regulär, da sie sich mittels $(II)^*$ definieren lässt.

6.2 Reguläre Sprachen werden von endlichen Automaten erkannt

In diesem Abschnitt wird gezeigt, dass zu jeder regulären Sprache L ein endlicher Automat konstruiert werden kann, der L erkennt. Dabei werden zunächst endliche Automaten für die Basisbausteine $\emptyset$, $\{\lambda\}$ und $\{x\}$ gebaut. Anschließend werden für beliebige nichtdeterministische endliche Automaten A_1 und A_2 der Vereinigungsautomat $A_1 \cup A_2$ und der Konkatenationsautomat $A_1 \circ A_2$ konstruiert, wobei der erste die Vereinigung der von A_1 und A_2 erkannten Sprachen erkennt und der zweite die Konkatenation, d.h. $L(A_1 \cup A_2) = L(A_1) \cup L(A_2)$ und $L(A_1 \circ A_2) = L(A_1)L(A_2)$. Zusätzlich wird aus einem nichtdeterministischen endlichen Automaten A der Sternautomat A_* für die Kleene-Hülle konstruiert, d.h. $L(A_*) = L(A)^*$.

Die in Abbildung 4 gezeigten drei endlichen Automaten erkennen (a) die leere Menge $\emptyset$, (b) die Sprache $\{\lambda\}$, bzw. (c) die Sprache $\{x\}$. Somit erkennen diese Automaten die kleinsten, unzerlegbaren Sprachmodule.

Seien $A = (Z, I, d, s_0, F)$ und $A_i = (Z_i, I, d_i, s_{0_i}, F_i)$ (für $i = 1,2$) nichtdeterministische endliche Automaten. Ohne Beschränkung der Allgemeinheit nehmen wir an, dass die Zustandsmengen von A_1 und A_2 keine gemeinsamen Elemente enthalten, d.h. $Z_1 \cap Z_2 = \emptyset$. (Ist dies nicht der Fall, können die Zustände von A_2 so umbenannt werden, dass die

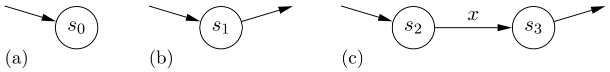


Abbildung 4: Endliche Automaten für atomare Sprachen: (a) $\emptyset$ (b) $\{\lambda\}$ (c) $\{x\}$ mit $x \in I$

Eigenschaft erfüllt ist.)

Bildlich gesprochen erhält man den Vereinigungsautomaten $A_1 \cup A_2$, indem man die beiden Automaten A_1 und A_1 nebeneinander setzt und ein neuer Startzustand s_0 hinzugefügt wird, von dem für jede Kante von s_{0_1} oder s_{0_2} eine neue mit derselben Markierung und demselben Ziel gezogen wird. Befindet man sich demnach im neuen Startzustand s_0, können beim Lesen von x genau die Zustände besucht werden, die von s_{0_1} oder s_{0_2} durch Lesen von x erreichbar sind.

Die Endzustände setzen sich aus allen Endzuständen der Automaten A_1 und A_2 zusammen, falls weder A_1 noch A_2 das leere Wort erkennen. Andernfalls wird s_0 in die Menge der Endzustände aufgenommen.

Folglich ist $A_1 \cup A_2 = (Z_1 \cup Z_2 \cup \{s_0\}, I, d, s_0, F)$ mit $s_0 \notin Z_1 \cup Z_2$, $d = d_1 \cup d_2 \cup \{(s_0, x, s) \mid (s_{0_i}, x, s) \in d_i, i = 1, 2\}$ und

$$F = \begin{cases} F_1 \cup F_2 \cup \{s_0\}, & \text{falls } s_{0_i} \in F_i, i \in \{1, 2\} \\ F_1 \cup F_2 & \text{sonst.} \end{cases}$$

Der so konstruierte Automat erkennt die Sprache $L(A_1) \cup L(A_2)$. Jedes Wort $w \in I^*$ ist genau dann in $L(A_1) \cup L(A_2)$ enthalten, wenn $w \in L(A_i)$ ist für ein $i \in \{1, 2\}$. Ist $w = \lambda$, gilt, dass w genau dann in $L(A_1) \cup L(A_2)$ enthalten ist, wenn s_{0_1} oder s_{0_2} ein Endzustand ist. Dies ist wiederum genau dann der Fall, wenn s_0 Endzustand von $A_1 \cup A_2$ ist, d.h. $\lambda \in L(A_1 \cup A_2)$. Andernfalls erhalten wir die Gleichheit von $L(A_1) \cup L(A_2)$ und $L(A_1 \cup A_2)$ aufgrund des folgenden Lemmas, das sich per vollständiger Induktion über den Aufbau von Wörtern beweisen lässt.

Lemma 6.1
Für alle $v \in I^*$, $x \in I$ gilt $d^*(s_0, vx) = d_1^*(s_{0_1}, vx) \cup d_2^*(s_{0_2}, vx)$.

Denn ein Wort w ist genau dann in $L(A_1 \cup A_2)$, wenn der Vereinigungsautomat nach dem Lesen von w einen Endzustand s erreichen kann, d.h. $s \in d^*(s_0, w)$ mit $s \in F$. Da $w \neq \lambda$ ist und man nach Definition von $A_1 \cup A_2$ den Startzustand s_0 kein zweites Mal besuchen kann, muss s in $F_1 \cup F_2$ liegen. Gemäß Lemma 6.1 kann s ebenfalls von A_1 oder A_2 durch Lesen von w erreicht werden, d.h. $s \in d_1^*(s_{0_1}, w) \cup d_2^*(s_{0_2}, w)$ mit $s \in F_1 \cup F_2$. Nach Voraussetzung sind die Zustandsmengen Z_1 und Z_2 disjunkt. Deshalb gilt für $i = 1, 2$, dass s genau dann in $d_i^*(s_{0_i}, w)$ ist, wenn $s \in F_1$ ist. Insgesamt bedeutet dies, dass $w \in L(A_1) \cup L(A_2)$ ist.

Der Konkatenationsautomat $A_1 \circ A_2$ setzt ebenfalls A_1 und A_2 nebeneinander. Der Startzustand ist s_{0_1}, und für jeden Endzustand $s' \in F_1$, jedes Zeichen $x \in I$ und jeden Zustand $s \in d_2(s_{0_2}, x)$ wird eine Kante von s' zu s mit der Markierung x gezogen. Folglich gelangt

man von s' beim Lesen von x zu den Zuständen in A_2, die beim Lesen von x von s_{0_2} erreichbar sind. Die Endzustände des Konkatenationsautomaten sind alle Endzustände von A_2, falls der Startzustand von A_2 nicht in F_2 liegt. Andernfalls werden alle Endzustände von F_1 dazugenommen, denn da in diesem Fall $\lambda \in L_2$ ist, müssen alle Wörter aus L_1 ebenfalls erkannt werden können.

Formal ist der Konkatenationsautomat von A_1 und A_2 der nichtdeterministische endliche Automat $A_1 \circ A_2 = (Z_1 \cup Z_2, I, d, s_{0_1}, F)$ mit $d = d_1 \cup d_2 \cup \{(s', x, s) \mid (s_{0_2}, x, s) \in d_2, s' \in F_1\}$ und

$$F = \begin{cases} F_1 \cup F_2, & \text{falls } s_{0_2} \in F_2 \\ F_2 & \text{sonst.} \end{cases}$$

Beim Abarbeiten eines Wortes $v_1 v_2$ mit $v_i \in L(A_i)$ $(i = 1, 2)$ gibt es die Möglichkeit zunächst v_1 zu lesen, indem A_1 vom Startzustand s_{0_1} bis zu einem Endzustand $s' \in F_1$ durchlaufen wird. Falls v_2 das leere Wort ist, wird $v_1 v_2$ erkannt, da F den Zustand s' enthält. Andernfalls kann anschließend mittels Lesen des ersten Zeichens von v_2 ein Folgezustand s von s_{0_2} besucht werden. Das restliche Teilwort von v_2 kann dann ausgehend von s von dem Automaten A_2 gelesen werden, so dass am Schluss ein Endzustand erreicht wird.

Der Beweis der Gleichheit von $L(A_1)L(A_2)$ und $L(A_1 \circ A_2)$ basiert auf folgendem Lemma, welches besagt, dass $A_1 \circ A_2$ ausgehend von jedem Endzustand in F_1 die Arbeitsweise von A_2 simulieren kann.

Lemma 6.2
Für alle $v \in I^*$, $x \in I$, $s \in F_1$ gilt $d_2^*(s_{0_2}, vx) \subseteq d^*(s, vx)$.

Der Sternautomat A_* wird aus A konstruiert, indem ein neuer Startzustand s_* geschaffen wird, der auch gleichzeitig in die Menge der Endzustände aufgenommen wird, damit das leere Wort erkannt werden kann. Von s_* und jedem Endzustand s' in F fügt man für jeden Zustand $s \in d(s_0, x)$ eine mit x markierte Kante nach s ein. Somit kann man von s_* oder s' beim Lesen von x genau die Zustände erreichen, die beim Lesen von x von s_0 erreichbar sind.

Der Sternautomat A_* ist folglich der nichtdeterministische endliche Automat

$$(Z \cup \{s_*\}, I, d_*, s_*, F \cup \{s_*\})$$

mit $s_* \notin Z$ und $d_* = d \cup \{(s', x, s) \mid (s_0, x, s) \in d, s' \in F \cup \{s_*\}\}$.

Der Automat A_* kann ein Wort der Form $w_1 \cdots w_n$ $(w_i \in L(A) \setminus \{\lambda\}$ für $i = 1, \ldots, n$ und $n \in \mathbb{N})$ lesen, indem er das n-malige Durchlaufen von A simuliert. Ist $n = 0$, d.h. $w_1 \cdots w_n = \lambda$, akzeptiert der Sternautomat $w_1 \cdots w_n$, da s_* ein Endzustand ist. Andernfalls nehmen wir an, dass A_* das Wort $w_1 \cdots w_{n-1}$ erkennt, d.h nach Lesen von $w_1 \cdots w_{n-1}$ befindet sich A_* in einem Endzustand. Von hier aus kann A_* das Wort w_n lesen, wobei zuerst eine der neu eingefügten Kanten durchlaufen und danach genau wie in A weiter gearbeitet wird.

Aufgrund der obigen Überlegungen gilt, dass jede reguläre Sprache von von einem endlichen Automaten erkannt wird, was die Folmulierung des folgenden Theorems erlaubt.

Theorem 3
Sei L eine reguläre Sprache. Dann gibt es einen endlichen Automaten A mit $L(A) = L$.

Bezeichnet $\mathcal{L}_{AUT(I)}$ die Klasse aller Sprachen über dem Alphabet I, die von endlichen Automaten erkannt werden, so gilt demnach, dass die Klasse aller regulären Sprachen in der Klasse der von endlichen Automaten erkannten Sprachen enthalten ist.

Korollar 4

$$\mathcal{L}_{REG(I)} \subseteq \mathcal{L}_{AUT(I)}.$$

6.3 Regularität der von endlichen Automaten erkannten Sprachen

Bemerkenswerterweise kann auch die Umkehrung nachgewiesen werden, so dass sich die von endlichen Automaten erkannten Sprachen als regulär erweisen und somit aus den atomaren Sprachmoduln allein durch endlich viele Vereinigungen, Konkatenationen und Kleene-Hüllen-Bildungen aufgebaut werden können.

Theorem 5
Sei $A = (Z, I, d, s_0, F)$ ein endlicher Automat. Dann ist $L(A)$ regulär.

Beweis.
Ohne Beschränkung der Allgemeinheit kann $Z = \{1, \ldots, n\}$ und $s_0 = 1$ angenommen werden. Dann ist für $i, j \in Z$ und $k \in \mathbb{N}$ die Sprache $L^k_{i,j}$ aller Wörter, die im Zustandsgraphen von i nach j führen, ohne zwischendurch Zustände jenseits von k zu besuchen, definiert als:

$$L^k_{i,j} = \{w \in I^* \mid j \in d^*(i, w),\ d^*(i, u) \subseteq \{1, \ldots, k\} \text{ für alle } u \text{ mit } w = uv,\ w \neq u \neq \lambda\}.$$

Mit Induktion über k kann gezeigt werden, dass $L^k_{i,j}$ für alle $i, j \in Z$ und $k \in \mathbb{N}$ regulär ist.

IA: Für $k = 0$ und $i \neq j$ enthält $L^k_{i,j}$ die Eingaben, die direkt von i nach j führen, weil alle Zustände jenseits von 0 liegen und deshalb nicht besucht werden dürfen. D.h. $L^0_{i,j} = \{x \in I \mid j \in d(i, x)\}$. Diese Sprache ist entweder leer oder die endliche Vereinigung von atomaren regulären Sprachen und deshalb selbst regulär.

Für $k = 0$ und $i = j$ gehört in jedem Fall noch λ zur Sprache, d.h. $L^0_{i,i} = \{\lambda\} \cup \{x \in I \mid i \in d(i, x)\}$, was sich analog als regulär erweist, weil $\{\lambda\}$ regulär ist.

IV: Als Induktionsvoraussetzung wird von der Regularität von $L^k_{i,j}$ für alle i und j ausgegangen.

IS: Betrachte nun im Induktionsschluss $L^{k+1}_{i,j}$ bzw. ein Element w daraus. Dieses Wort induziert einen Weg von i nach j im Zustandsgraph. Sei $l_0 \cdots l_p$ die durchlaufene Sequenz von Zuständen mit $i = l_0$ und $j = l_p$. Kommt der Zustand $k + 1$ gar nicht in $l_1 \cdots l_{p-1}$ vor, so liegt w in $L^k_{i,j}$. Ansonsten kommt der Zustand $k+1$ m-mal darin vor mit $0 < m < p$, so dass die Sequenz folgende Form hat:

$$l_0 \cdots l_{p_1-1}(k+1)l_{p_1+1} \cdots l_{p_2-1}(k+1) \cdots (k+1)l_{p_m+1} \cdots l_p,$$

wobei $p_1 < p_2 < \cdots < p_m$ die Stellen sind, wo $k + 1$ auftritt. Insbesondere kommt in den Abschnitten $l_1 \cdots l_{p_1-1}, l_{p_1+1} \cdots l_{p_2-1}, \ldots, l_{p_m+1} \cdots l_{p-1}$ der Zustand $k + 1$ nicht vor, sondern nur die Zustände $1, \ldots, k$. Es stellt sich also heraus, dass $w = w_0 w_1 \ldots w_m$ für Wörter $w_0 \in L^k_{i,k+1}$, $w_1, \ldots, w_{m-1} \in L^k_{k+1,k+1}$ und $w_m \in L^k_{k+1,j}$, d.h. $w \in L^k_{i,k+1}(L^k_{k+1,k+1})^* L^k_{k+1,j}$. Damit ist gezeigt, dass

$$L^{k+1}_{i,j} \subseteq L^k_{i,j} \cup L^k_{i,k+1}(L^k_{k+1,k+1})^* L^k_{k+1,j}.$$

Nach Definition von $L^k_{i,j}$ gilt auch die umgekehrte Inklusion, so dass insgesamt

$$L^{k+1}_{i,j} = L^k_{i,j} \cup L^k_{i,k+1}(L^k_{k+1,k+1})^* L^k_{k+1,j}$$

folgt. Nach Induktionsvoraussetzung sind $L^k_{i,j}$, $L^k_{i,k+1}$, $L^k_{k+1,k+1}$ und $L^k_{k+1,j}$ regulär, so dass auch die Kleene-Hülle der dritten Sprache und die Konkatenation mit den anderen beiden Sprachen sowie die Vereinigung mit der ersten Sprache regulär sind. Also ist $L^{k+1}_{i,j}$ regulär, was zu zeigen war.

Die Sprachen $L^k_{i,j}$ sind also für alle $i, j \in Z$ und $k \in \mathbb{N}$ regulär. Die von A erkannte Sprache ist eine Vereinigung endlich vieler dieser Sprachen, denn es gilt $L(A) = \bigcup_{j \in F} L^n_{1,j}$. Somit ist auch $L(A)$ regulär, wie behauptet. $\square$

6.4 Reguläre Ausdrücke

Reguläre Ausdrücke sind, analog zu arithmetischen Ausdrücken, aus Konstanten, Operationen und Klammern aufgebaute Zeichenketten, die einen Wert beschreiben. Der Wert soll in diesem Fall allerdings keine Zahl sein, sondern eine Sprache. Demzufolge müssen die Konstanten möglichst einfache Sprachen repräsentieren und die Operationen müssen es erlauben, aus diesen Sprachen komplexere zu bilden.

In der einen oder anderen Form dürften reguläre Ausdrücke den meisten schon einmal über den Weg gelaufen sein, da sie in Editoren mit komfortablen Suchfunktionen und in UNIX-Kommandos wie z.B. grep, find und sed Verwendung finden.

Sei I ein Alphabet mit $lambda, empty, +, \circ, *, (,) \notin I$. Die Menge $REX(I)$ der *regulären Ausdrücke über I* ist rekursiv definiert durch:

(i) $\mathit{empty}, \mathit{lambda} \in \mathit{REX}(I)$,
(ii) $I \subseteq \mathit{REX}(I)$ und
(iii) für alle $r, r_1, r_2 \in \mathit{REX}(I)$ sind auch die Wörter $(r_1 + r_2)$, $(r_1 \circ r_2)$ und (r^*) in $\mathit{REX}(I)$.

Jedem regulären Ausdruck r über I wird wie folgt eine Sprache $L(r)$ zugeordnet:

(i) $L(\mathit{empty}) = \emptyset$, $L(\mathit{lambda}) = \{\lambda\}$ und $L(x) = \{x\}$ für alle $x \in I$,
(ii) für alle $r, r_1, r_2 \in \mathit{REX}(I)$ sei
  (1) $L((r_1 + r_2)) = L(r_1) \cup L(r_2)$,
  (2) $L((r_1 \circ r_2)) = L(r_1)L(r_2) = \{w_1 w_2 \mid w_1 \in L(r_1),\ w_2 \in L(r_2)\}$ und
  (3) $L((r^*)) = L(r)^* = \{w_1 \cdots w_n \mid w_1, \ldots, w_n \in L(r),\ n \in \mathbb{N}\}$.

Bemerkungen

1. Nach der Interpretation regulärer Ausdrücke steht $+$ für die Vereinigung zweier Sprachen, $\circ$ für deren Konkatenation und $*$ für die Iteration einer Sprache. Die Verwendung der Symbole $+$ und $\circ$ für Vereinigung und Konkatenation hat einen guten Grund: Abstrakt gesehen, verhalten sich diese Operationen wie Addition und Multiplikation (genauer gesagt, die Menge der Wörter bildet mit diesen Operationen einen Semiring). Dabei ist $\emptyset$ das neutrale Element der Addition (die "Null") und $\{\lambda\}$ ist das der Multiplikation (die "Eins"). Wie gewohnt ergibt die Multiplikation mit Null immer Null: $\emptyset L = \emptyset = L\emptyset$. Beide Operationen sind offenbar assoziativ, aber nur die Vereinigung ist auch kommutativ, denn $L_1 L_2$ ist natürlich im allgemeinen nicht dasselbe wie $L_2 L_1$. Wie man sich leicht überzeugen kann, gelten auch die Distributivgesetze: $L(L_1 \cup L_2) = LL_1 \cup LL_2$ und $(L_1 \cup L_2)L = L_1 L \cup L_2 L$. Damit sind alle Zutaten beisammen, die man für einen Semiring benötigt (zu einem "echten" Ring fehlen die inversen Elemente bezüglich der Addition). Übrigens hat auch der Kleene-Stern einige interessante Eigenschaften. Insbesondere ist er idempotent, $L^{**} = L^*$, was praktisch direkt aus der Definition folgt.
2. Um reguläre Ausdrücke übersichtlicher zu machen, wird üblicherweise von der Konvention Gebrauch gemacht, dass $*$ stärker bindet als $\circ$ und dies wiederum stärker als $+$. Klammern, die nach dieser Konvention (oder der Assoziativität von $+$ und $\circ$) zur Vermeidung von Mehrdeutigkeiten unnötig sind, können weggelassen werden. Außerdem lässt man das Symbol $\circ$ oft weg, analog zur Schreibweise bei arithmetischen Ausdrücken, wo man ja auch oft xy für $x \cdot y$ schreibt. Demnach kann man beispielsweise statt $(((a \circ b)^*) \circ (c \circ (c^*)))$ auch kurz $(ab)^* cc^*$ schreiben.
3. Gelegentlich wird in der Literatur ein regulärer Ausdruck r angegeben, wenn eigentlich $L(r)$ gemeint ist, sofern dies aus dem Kontext genügend klar hervorgeht. Dies sollte aber nicht darüber hinwegtäuschen, dass r und $L(r)$ zwei grundverschiedene Dinge sind: r ist eine Zeichenkette, während $L(r)$ eine Sprache ist.

Aus der Definition der regulären Ausdrücke und ihrer Interpretation sowie der Definition der regulären Sprachen folgt unmittelbar, dass eine Sprache $L \subseteq I^*$ genau dann regulär ist, wenn es einen regulären Ausdruck $r \in \mathit{REX}(I)$ mit $L = L(r)$ gibt. Reguläre Ausdrücke

stellen somit einen weiteren Formalismus neben endlichen Automaten dar, um reguläre Sprachen zu spezifizieren. Formal gilt:

$$\mathcal{L}_{\mathit{REG}(I)} = \mathcal{L}_{\mathit{AUT}(I)} = \mathcal{L}_{\mathit{REX}(I)},$$

wobei $\mathcal{L}_{\mathit{REX}(I)} = \{L(r) \mid r \in \mathit{REX}(I)\}$ alle durch Interpretation der regulären Ausdrücke über I erhaltenen Sprachen enthält.

7 Pumping-Lemma für erkannte Sprachen

Endliche Automaten haben als Modellierungskonzept viele brauchbare Eigenschaften: Ihre Sprache lässt sich schnell erkennen, sie besitzen neben der graphisch-visuellen auch eine textuell-kompositionelle Beschreibung, und sie erlauben automatische Korrektheitsbeweise. Sie wären ein ideales Modellierungsinstrument, wenn es nicht vieles Interessante gäbe, was mit ihnen nicht modelliert werden kann. Um das einsehen zu können, soll zunächst eine Eigenschaft aller von endlichen Automaten erkannten Sprachen bewiesen werden. Wenn es eine Sprache gibt, die diese Eigenschaft nicht hat, so gibt es demnach keinen endlichen Automaten, der sie erkennt.

Es wird ein Pumping Lemma gezeigt, das aussagt, dass genügend lange Wörter einer von einem endlichen Automaten erkannten Sprache ein Teilwort besitzen, das beliebig oft wiederholt werden kann, ohne dass man dabei die Sprache verlässt.

Theorem 6 (Erstes Pumping-Lemma)
Sei L eine von einem endlichen Automaten erkannte Sprache.

Dann existiert eine natürliche Zahl $p \in \mathbb{N}$ derart, dass jedes Wort $w \in L$ mit $length(w) \geq p$ zerlegt werden kann in drei Teilwörter $w = xyz$ mit $length(xy) \leq p$ und $length(y) > 0$ und dass $xy^iz \in L$ ist für alle $i \geq 0$.

Beweis.
Nach Voraussetzung existiert ein Automat $A = (Z, I, d, s_0, F)$, der L erkennt. Wähle dann p als Anzahl der Zustände von A ($p = \#Z$). Gibt man nun ein Wort $w = x_1 \cdots x_n \in L$ ($x_i \in I$) mit $n \geq p$ in A ein, so erhält man durch buchstabenweises Abarbeiten eine Folge $s_0 \cdots s_n$ von Zuständen mit der Eigenschaft

$$s_i \in d(s_{i-1}, x_i) \text{ für } i = 1, \ldots, n.$$

Wegen $n \geq p$ gibt es unter den ersten $p+1$ Zuständen $s_0, \ldots, s_p$ zwei gleiche, etwa $s_j = s_k$ mit $0 \leq j < k \leq p$. Das liefert die gewünschte Zerlegung von w, denn mit $x = x_1 \cdots x_j$ kommt man von s_0 nach s_j, mit $y = x_{j+1} \cdots x_k$ von s_j nach $s_k = s_j$, was man dann aber auch immer wiederholen oder auslassen kann, und von s_k gelangt man mit $z = x_{k+1} \cdots x_n$ nach $s_n \in F$. Insgesamt erreicht man also mit den Wörtern xy^iz für $i \geq 0$ von s_0 den Endzustand s_n. Das aber bedeutet gerade $xy^iz \in L$, wie behauptet.

Nach Konstruktion ist $length(xy) = k \leq p$ und $length(y) = k - j > 0$. □

Beispiel: Anwendung des Pumping-Lemmas

1. Das Pumping-Lemma kann benutzt werden, um zu zeigen, dass eine Sprache von keinem endlichen Automaten erkannt wird.

Betrachte die Sprache $L_{balance} = \{a^nb^n \mid n \in \mathbb{N}\}$. Angenommen, sie wird von einem endlichen Automaten erkannt. Sei dann $p \in \mathbb{N}$ die Konstante aus dem Pumping-Lemma, und wähle $w = a^pb^p \in L_{balance}$. Nach dem Pumping-Lemma kann w in drei Teilwörter $w = xyz$ mit $length(xy) \leq p$ und $y \neq \lambda$ zerlegt werden, so dass $xy^iz \in L_{balance}$ für alle $i \in \mathbb{N}$. Das Teilwort y liegt somit in der ersten Hälfte von w, d.h. $y = a^k$ für ein $k > 0$. Dies kann nicht sein, denn $xy^0z = xz = a^{p-k}b^p \notin L_{balance}$ für $k \neq 0$. Also gibt es keine Zerlegung von w mit den geforderten Eigenschaften, was bedeutet, dass die Annahme falsch gewesen sein muss.

2. Analog kann gezeigt werden, dass die Sprache $L' = \{w \in \{a, b\}^* \mid count_a(w) = count_b(w)\}$ von keinem endlichen Automaten erkannt werden kann. Intuitiv liegt das daran, dass beim Abarbeiten eines Eingabewortes eine beliebig große Differenz zwischen der Anzahl der gelesenen a's und der der b's entstehen kann, aber in den endlich vielen Zuständen eines endlichen Automaten kann maximal nur eine beschränkt große Differenz gespeichert werden.

8 Syntax von Programmiersprachen und Syntaxanalyse

Mit Hilfe des Pumping Lemmas wurde im vorigen Abschnitt gezeigt, dass schon einfachste Klammerstrukturen mit n öffnenden Klammern (repräsentiert durch a) gefolgt von n schließenden Klammern (repräsentiert durch b) für beliebige $n \in \mathbb{N}$ nicht von endlichen Automaten erkannt werden können. Das ist auch anschaulich klar. Denn beim Lesen von links nach rechts kann sich ein endlicher Automat mit seiner beschränkten Zahl von Zuständen nur beschränkt viele öffnende Klammern merken, während das Eingabewort beliebig viele enthalten kann, so dass später kein Vergleich mehr mit der Zahl der schließenden Klammern möglich ist. Da Klammerstrukturen in praktisch allen Programmiersprachen vielfältig vorkommen, für die dann eine analoge Beweisführung möglich ist wie für die einfachen Klammerstrukturen, stellt sich heraus, dass Programme von Programmiersprachen in der Regel nicht durch endliche Automaten erkannt werden können.

Wer ein Programm in einer Programmiersprache X schreiben möchte, wählt sich häufig einen Texteditor Y aus und erstellt mit dessen Hilfe eine bestimmte Zeichenkette, die dann als Eingabe für den Compiler von X dient. Dies ist in Abbildung 5 skizziert.

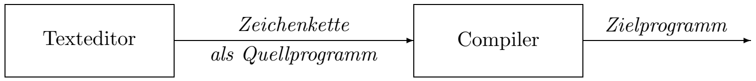


Abbildung 5: Erstellung eines Programms

Man kann nicht erwarten, dass jeder Text, der mit dem Editor Y entstehen kann, bereits ein Programm in X ist. Denn mit einem Texteditor lassen sich in der Regel beliebige Zeichenketten aufbauen, während ein Programm eine bestimmte Form haben muss. Zeichenketten jedoch, die keine Programme sind, wird der Compiler als unübersetzbar zurückweisen. Woher weiß aber eine Programmiererin oder ein Programmierer, wie die Zeichenkette aussehen muss, um ein Programm zu sein? Wie findet der Compiler heraus, ob irgendeine Zeichenkette ein übersetzbares Programm darstellt?

Die Form von Programmen einer Programmiersprache wird durch ihre Syntax festgelegt. Die Syntaxdefinition macht meist äußerst restriktive Vorschriften über die Anordnung und Plazierung von Zeichen, damit eine Zeichenkette ein syntaktisch richtiges Programm ist. Eine Person, die ein Programm mit Hilfe eines Texteditors erstellen will, sollte die Syntax recht gut kennen, weil andernfalls sicherlich häufig Syntaxfehler auftreten. Der Compiler dagegen übersetzt die eingegebene Zeichenkette in ein Zielprogamm, falls die Eingabe ein syntaktisch korrekt gebildetes Programm ist. Um das festzustellen, besitzen Compiler eine Syntaxanalyse-Komponente, die diese Aufgabe übernimmt.

Bei der Syntaxanalyse wird für eingegebene Zeichenketten untersucht, ob sie Programme

sind oder nicht. Im positiven Fall wird außerdem der syntaktische Aufbau ermittelt, weil diese Information bei der weiteren Übersetzung maßgeblich genutzt wird. Im negativen Fall werden meist noch Hinweise auf Syntaxfehler gegeben. Die Trennung in "richtige" und "falsche" Zeichenketten ist in der Regel das entscheidende algorithmische Problem, weil bei der Lösung die zusätzlichen Informationen ohne allzu große Mühe nebenbei gewonnen werden können.

Die syntaktisch richtigen Programme einer Programmiersprache bilden als Menge von Zeichenketten eine "formale Sprache". Solche Zeichenkettenmengen sind Gegenstand der Untersuchung in der Theorie formaler Sprachen, die Konzepte für die syntaktische Definition formaler Sprachen bereitstellt und Methoden liefert, um die Eigenschaften formaler Sprachen analysieren zu können. Zu den wichtigsten Anwendungsfeldern der Theorie formaler Sprachen gehören die Syntaxdefinition von Programmiersprachen und die Syntaxanalyse. Die Schlüsselfrage der Syntaxanalyse, ob eine Zeichenkette ein Programm ist oder nicht, wird auch *Wortproblem* genannt. Betrachtet man die Menge aller richtigen Programme als formale Sprache, dann besteht das Problem darin, ob eine Zeichenkette in der Sprache liegt oder nicht. Da die Lösung des Wortproblems eine zentrale Rolle bei der Implementierung von Programmiersprachen spielt, aber keineswegs immer auf der Hand liegt, zieht sich die Behandlung des Wortproblems wie ein roter Faden durch die Theorie formaler Sprachen.

Aber erst einmal zurück zur Syntaxdefinition. Eine grundlegende Weise, formale Sprachen, einschließlich Programmiersprachen, zu spezifizieren, besteht darin, die übliche Art der Begriffsbildung (unter *dem und dem* versteht man *das und das*) in formalisierter Form zu nutzen. Einem zu definierenden, also noch undefinierten, syntaktischen Konstrukt wird ein definierender Ausdruck zugeordnet. Beides zusammen bildet eine Syntaxregel, das (noch) Undefinierte wird linke, das Definierende rechte Regelseite genannt. Der definierende Ausdruck der rechten Seite ist eine Zeichenkette, die rekursiv auch wieder undefinierte Konstrukte enthalten darf. Ist das noch Undefinierte der linken Seite durch ein einziges Zeichen dargestellt, spricht man von einer kontextfreien Regel. Die Syntaxdefinition einer Programmiersprache besteht in einem ersten Anlauf meist in der Angabe von kontextfreien Regeln, die dann später um Syntaxteile ergänzt werden, die sich nicht durch kontextfreie Regeln ausdrücken lassen.

Der kontextfreie Anteil der Syntax von Programmiersprachen wird häufig in der sogenannten Backus-Naur-Form geschrieben, wobei die kontextfreien Regeln als linke Seiten nichtterminale Zeichen, die zu definierende syntaktische Konstrukte der Sprache benennen, und als rechte Seiten Zeichenketten aus terminalen und nichtterminalen Zeichen besitzen. Die rechten Seiten zur selben linken Seite werden als Alternativen nebeneinandergestellt, durch einen senkrechten Strich voneinander getrennt. Linke und rechte Seiten werden durch das Trennzeichen "::=" auseinandergehalten. Nichtterminale Zeichen sind in spitze Klammern eingeschlossen.

Für die Beschreibung der Form von Programmen – oder wie man auch sagt: ihrer Syntax – werden also andere Mittel gebraucht. Sehr häufig werden dafür kontextfreie Grammatiken verwendet, die in ähnlicher Form auch für die grammatikalische Beschreibung natürlicher

Sprachen eingesetzt werden. In diesem Abschnitt werden solche Grammatiken motiviert und informell eingeführt. Die formale Behandlung folgt dann in den folgenden Abschnitten.

Um das Prinzip zu illustrieren, sollen Boolesche Ausdrücke einfacher Art beschrieben werden. Ein solcher Ausdruck kann eine der Booleschen Konstanten $true$ oder $false$ sein, eine Boolesche Variable, ein negierter Boolescher Ausdruck oder die Komposition zweier Boolescher Ausdrücke durch einen Booleschen Operator. In den beiden letzten Fällen werden jeweils Klammern verwendet, damit Anfang und Ende der Ausdrücke eindeutig festgelegt sind. Neben den zu definierenden Ausdrücken sind auch die Variablen ein syntaktisches Konstrukt, das definiert werden muss. Der Einfachheit halber geschieht das durch jeweils ein $''b''$ gefolgt von einer Ziffernfolge. Desweiteren sind Boolesche Operatoren und Ziffernfolgen sowie Ziffern als syntaktische Konstrukte eingeführt und definiert. Die folgenden Syntaxregeln reflektieren die verbale Beschreibung:

$$
\begin{aligned}
&< boolexp >::= true \mid false \mid < var > \mid \\
&(\neg < boolexpr >) \mid (< boolexpr >< boolop >< boolexpr >) \\
&< boolop >::= \wedge \mid \vee \mid \Rightarrow \mid \Leftrightarrow \\
&< var >::= b < cipherseq > \\
&< cipherseq >::=< cipher > \mid < cipher >< cipherseq > \\
&< cipher >::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{aligned}
$$

Die Regeln können zum Aufbau syntaktisch korrekter Boolescher Ausdrücke benutzt werden, indem mit dem nichtterminalen Zeichen $< boolexpr >$ begonnen und dann nach und nach in den aktuellen Zeichenketten je ein nichtterminales Zeichen durch eine zugehörige rechte Regelseite ersetzt wird, bis keine nichtterminalen Zeichen mehr vorkommen. Ein Beispiel dafür ist:

$$
\begin{aligned}
< boolexp > &\to (< boolexpr >< boloop >< boolexpr >) \\
&\to (< boolexpr > \Leftrightarrow < boolexpr >) \\
&\to (< boolexpr > \Leftrightarrow true) \\
&\to ((< boolexpr >< boloop >< boolexpr >) \ \Leftrightarrow true) \\
&\to ((< boolexpr > \vee < boolexpr >) \Leftrightarrow true) \\
&\to ((< boolexpr > \vee (\neg < boolexpr >)) \ \Leftrightarrow true) \\
&\to ((< var > \vee (\neg < boolexpr >)) \ \Leftrightarrow true) \\
&\to ((< var > \vee (\neg < var >)) \Leftrightarrow true) \\
&\to ((b < cipherseq > \vee (\neg < var >)) \ \Leftrightarrow true) \\
&\to ((b < cipherseq > \vee (\neg\ b < cipherseq >)) \ \Leftrightarrow true) \\
&\to ((b < cipher > \vee (\neg\ b < cipherseq >)) \ \Leftrightarrow true) \\
&\to ((b < cipher > \vee (\neg\ b < cipher >)) \ \Leftrightarrow true) \\
&\to ((b < cipher > \vee (\neg\ b\ 0)) \ \Leftrightarrow true) \\
&\to ((b\ 0 \vee \ (\neg\ b\ 0)) \ \Leftrightarrow true)
\end{aligned}
$$

Wie dieses Gesetz vom ausgeschlossenen Dritten lassen sich alle Booleschen Ausdrücke mit Hilfe der Regeln herleiten.

9 Kontextfreie Grammatiken

Die im vorigen Abschnitt erläuterte Art, Zeichenketten einer bestimmten Form durch Syntaxregeln festzulegen, wird hier mit dem Konzept kontextfreier Grammatiken und ihrer erzeugten Sprachen formal definiert.

9.1 Definition kontextfreier Grammatiken

Eine kontextfreie Grammatik besteht aus endlich vielen Regeln, die in der Literatur oft auch Produktionen genannt werden. Eine kontextfreie Regel hat die Form $A ::= u$, wobei A ein nichtterminales Zeichen ist und u ein Wort, das aus nichtterminalen und terminalen Zeichen zusammengesetzt sein kann. Außerdem gibt es ein nichtterminales Startsymbol.

1. Eine *kontextfreie Grammatik* ist ein System $G = (N, T, P, S)$, wobei N eine Menge *nichtterminaler Zeichen*, T eine Menge *terminaler Zeichen*, P eine endliche Menge kontextfreier Produktionen und $S \in N$ ein *Startsymbol* ist. Dabei ist eine kontextfreie Produktion (Regel) ein Paar $(A, u) \in N \times (N \cup T)^*$, das meist als $A ::= u$ geschrieben wird.
2. Das Zeichen A wird *linke Seite*, die Zeichenkette u *rechte Seite* von p genannt. Zur Abkürzung können mehrere Produktionen $A ::= u_1, \ldots, A ::= u_k$ $(k \geq 2)$ mit derselben linken Seite zu $A ::= u_1 \mid \cdots \mid u_k$ zusammengefasst werden. Soweit nichts anderes gesagt wird, nimmt man an, dass kein nichtterminales Zeichen gleichzeitig terminal ist, d.h. $N \cap T = \emptyset$.

9.2 Ableitungsprozess

Produktionen werden auf Zeichenketten angewendet indem man in einer Zeichenkette ein Zeichen sucht, das die linke Seite einer Produktion ist, und es durch die rechte Seite ersetzt.

1. Seien $w, w', x, y, u, v \in (N \cup T)^*$. Dann wird w' aus w *direkt* durch Anwendung der Produktion $p = (A ::= u)$ *abgeleitet*, falls $w = xAy$ und $w' = xuy$. In diesem Falle wird $w \underset{p}{\longrightarrow} w'$ geschrieben.

   Die Anwendung einer Produktion wird *direkte Ableitung* genannt. Ist P eine Menge von Produktionen und $p \in P$, so kann man statt $w \underset{p}{\longrightarrow} w'$ auch $w \underset{P}{\longrightarrow} w'$ schreiben.
2. Die Iteration direkter Ableitungen ergibt das Konzept der *Ableitung*:

   $$w_0 \underset{p_1}{\longrightarrow} w_1 \underset{p_2}{\longrightarrow} \cdots \underset{p_n}{\longrightarrow} w_n$$

   für $w_0, \ldots, w_n \in (N \cup T)^*$ und Produktionen $p_1, \ldots, p_n$ $(n \geq 1)$. Stammen alle angewendeten Produktionen aus P, so kann man die obige Ableitung auch schreiben als $w_0 \underset{P}{\longrightarrow} \cdots \underset{P}{\longrightarrow} w_n$ oder $w_0 \overset{n}{\underset{P}{\longrightarrow}} w_n$. Für manche Zwecke ist es sinnvoll, auch *Nullableitungen* zuzulassen: $w \overset{0}{\underset{P}{\longrightarrow}} w$ für alle $w \in (N \cup T)^*$. Statt $w \overset{n}{\underset{P}{\longrightarrow}} w'$ für $n \in \mathbb{N}$

darf auch $w \xrightarrow[P]{*} w'$ geschrieben werden. Außerdem kann man bei Ableitungen und direkten Ableitungen das Subskript P weglassen, wenn die Produktionsmenge aus dem Kontext klar ist.

9.3 Erzeugte Sprache

Der Ableitungsprozess bildet die operationelle Semantik, die durch eine Produktionsmenge syntaktisch beschrieben ist. Betrachtet man diejenigen Zeichenketten, die aus dem Startsymbol einer kontextfreien Grammatik $G = (N, T, P, S)$ ableitbar sind und nur aus terminalen Zeichen bestehen, so erhält man auf der Basis des Ableitungsprozesses eine erzeugte Sprache.

Sei $G = (N, T, P, S)$ eine kontextfreie Grammatik. Dann enthält die von G *erzeugte Sprache* alle mit Produktionen in P aus dem Startsymbol S ableitbaren terminalen Zeichenketten:

$$L(G) = \{w \in T^* \mid S \xrightarrow[P]{*} w\}.$$

Auf diese Weise stellen kontextfreie Grammatiken ein syntaktisches Instrument dar, um formale Sprachen zu spezifizieren, die dann entsprechend *kontextfreie Sprachen* genannnt werden. Ausführliche Darstellungen von kontextfreien Grammatiken findet man in praktisch jedem Buch über formale Sprachen (siehe auch Kapitel 21 über Anmerkungen zur Literatur).

9.4 Beispiele

1. Mit der Produktion $S ::= aS$ lässt sich das Zeichen a hochzählen:

   $$S \to aS \to a^2S \to \cdots \to a^nS.$$

   Entsprechend kann man mit $S ::= a^kS$ ($k \in \mathbb{N}$) ein Vielfaches von k hochzählen. Terminieren lässt sich dieser Vorgang mit $S ::= \lambda$, so dass

   $$L((\{S\}, \{a\}, \{S ::= a^kS \mid \lambda\}, S)) = \{a^{n \cdot k} \mid n \in \mathbb{N}\}.$$

2. Auch das getrennte Zählen zweier (bzw. mehrerer) Größen ist kein Problem. Sei $T_l = \{a_1, \ldots, a_l\}$ ($l \geq 1$), $N_l = \{S\} \cup \{A_1, \ldots, A_l\}$ und $P_l = \{S ::= A_1 \cdots A_l\} \cup \{A_i ::= a_iA_i \mid i = 1, \ldots, l\} \cup \{A_i ::= \lambda \mid i = 1, \ldots, l\}$.
   Dann gilt:
   $$L((N_l, T_l, P_l, S)) = \{a_1^{n_1} \cdots a_l^{n_l} \mid n_i \geq 0, i = 1, \ldots, l\}.$$

3. Fast genauso einfach ist es, zwei Größen gleichzeitig hochzuzählen:

   $$L((\{S\}, \{a, b\}, \{S ::= aSb \mid \lambda\}, S)) = \{a^nb^n \mid n \in \mathbb{N}\}.$$

   Es erweist sich also als ausgesprochen einfach, Klammerausdrücke, an denen endliche Automaten scheitern, durch eine kontextfreie Grammatik zu spezifizieren.

4. Auch kompliziertere Klammerausdrücke mit mehreren Klammern, die nicht nur ineinander, sondern auch nebeneinander gesetzt werden können, wie das in Programmiersprachen üblich ist, lassen sich kontextfrei spezifizieren.
   Seien $(a_1, b_1), \ldots, (a_k, b_k)$ k Klammerpaare und $T_{bracket} = \{a_1, \ldots a_k\} \cup \{b_1, \ldots, b_k\}$. Dann erzeugt die kontextfreie Grammatik $G_{bracket} = (\{S\}, T_{bracket}, P_{bracket}, S)$ mit den Produktionen
   $$S ::= SS \mid a_ib_i \mid a_iSb_i \text{ für } i = 1, \ldots, k$$
   alle Klammerstrukturen mit den gegebenen Klammerpaaren.
   Für die Klammerpaare [,] und $<, >$ lässt sich beispielsweise folgende Struktur ableiten:

   $$S \to SS \to S[S] \to < S > [S] \to < SS > [S] \to < []S > [S] \to$$
   $$< []S > [<>] \to < [][] > [<>]$$

10 Übersetzung endlicher Automaten in rechtslineare Grammatiken

Kontextfreie Grammatiken können nicht nur Sprachen erzeugen, die endliche Automaten nicht erkennen, sondern stellen selbst eine Verallgemeinerung endlicher Automaten dar. Genauer gesagt, lässt sich ein Übersetzer von endlichen Automaten in kontextfreie Grammatiken angegeben, bei dem im Prinzip nur die Zustandsüberführung in Regelform umgewandelt wird. Es wird gezeigt, dass der Übersetzer korrekt ist, d.h. dass der eingegebene Automat die Sprache erkennt, die die ausgegebene Grammatik erzeugt.

Die bei der Übersetzung entstehenden Regeln haben eine spezielle Form, die rechtslinear genannt wird.

Theorem 7
Sei $A = (Z, I, d, s_0, F)$ ein endlicher Automat. Dann wird durch $GRA(A) = (Z, I, P_A, s_0)$ mit

$$P_A = \{s ::= xs' \mid s' \in d(s, x)\} \cup \{s'' ::= \lambda \mid s'' \in F\}$$

eine rechtslineare Grammatik konstruiert, so dass gilt: $L(A) = L(GRA(A))$.

Diese Übersetzung (einschließlich der semantischen Verträglichkeit) lässt sich mit dem Diagramm in Abbildung 6 illustrieren.

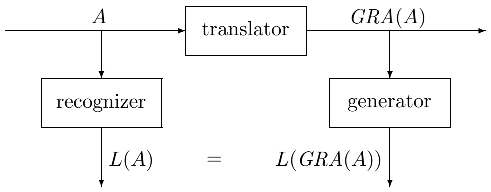


Abbildung 6: Korrekte Übersetzung endlicher Automaten in rechtslineare Grammatiken

Beweis.
Da $GRA(A)$ offensichtlich eine rechtslineare Grammatik ist, muss nur die Sprachgleichheit gezeigt werden. Dazu wird ein Zusammenhang zwischen der fortgesetzten Zustandsüberführung von A und den Ableitungen von $GRA(A)$ im anschließenden Lemma 8 hergestellt. Die behauptete Sprachgleichheit folgt dann vergleichsweise einfach:

$w \in L(A)$ bedeutet $d^*(s_0, w) \cap F \neq \emptyset$, d.h. es gibt einen Zustand $s' \in F$ mit $s' \in d^*(s_0, w)$. Nach Definition der Regeln und Lemma 8 ist das gleichbedeutend zu $s' ::= \lambda \in P_A$ und $s_0 \xrightarrow[P_A]{*} ws'$. Daraus lässt sich die Ableitung $s_0 \xrightarrow[P_A]{*} ws' \xrightarrow[s' ::= \lambda]{} w\lambda = w$ zusammensetzen. Die Ableitung $s_0 \xrightarrow[P_A]{*} w$ liefert aber gerade $w \in L(GRA(A))$.

Umgekehrt zerfällt eine Ableitung der Form $s_0 \xrightarrow[P_A]{*} w$ immer in Ableitungen $s_0 \xrightarrow[P_A]{*} ws'$ und $ws' \xrightarrow[s' ::= \lambda]{} w$, weil das die einzige Möglichkeit zum Terminieren ist. Damit lässt sich die gesamte Überlegung auch umdrehen. $\square$

Lemma 8
Seien A und $GRA(A)$ wie in Theorem 7.
Dann gilt für alle $s, s' \in Z$ und $w \in I^*$: $s \xrightarrow[P_A]{*} ws'$ gdw. $s' \in d^*(s, w)$.

Beweis (mit Induktion über die Länge von w)**.**

IA (für $w = \lambda$):
$s \xrightarrow[P_A]{*} \lambda s' = s'$ gdw. $s = s'$ gdw. $s' \in \{s\} = d^*(s, \lambda)$,
wobei in der zweiten Äquivalenz die Definition von d^* ausgenutzt wird und bei der ersten die Tatsache, dass jede Regelanwendung in $GRA(A)$ ein terminales Zeichen erzeugt oder das nichtterminale löscht.

IV: Die Behauptung gelte für $w \in I^*$.

IS (für wx mit $w \in I^*$ und $x \in I$):
Eine Ableitung der Form $s \xrightarrow[P_A]{*} wxs'$ zerfällt immer in zwei Ableitungen der Form $s \xrightarrow[P_A]{*} w\bar{s}$ und $w\bar{s} \xrightarrow[\bar{s} ::= xs']{} wxs'$, weil nur Regeln der Form $\bar{s} ::= xs'$ Wörter verlängern und weil das nur am rechten Ende geschehen kann. Nach der Induktionsvoraussetzung korrespondiert die Ableitung $s \xrightarrow[P_A]{*} w\bar{s}$ zu $\bar{s} \in d^*(s, w)$. Die im Ableitungsschritt angewendete Regel korrespondiert zu $s' \in d(\bar{s}, x)$. Beides zusammen bedeutet nach Definition von d^* gerade $s' \in \bigcup_{t \in d^*(s,w)} d(t, x) = d^*(s, wx)$. $\square$

11 Kellerautomaten

Die Erkennung von Sprachen durch endliche Automaten ist angemessen schnell, aber in ihrem Anwendungsspektrum ziemlich eingeschränkt, weil ein endlicher Automat während des Abarbeitens eines Eingabewortes nur eine beschränkte Zahl von Informationen speichern kann, die durch die Zustände vorgegeben ist. Insbesondere kann nur bis zu einer Schranke gezählt werden, und nur beschränkte Abschnitte des Eingabewortes können für spätere Vergleiche aufgehoben werden. Um dagegen eine Sprache wie

$$L_{balance} = \{a^n b^n \mid n \in \mathbb{N}\}$$

zu erkennen, muss man unbeschränkt zählen können. Oder um das Wortproblem von

$$L_{palindrom} = \{w \in T^* \mid w = trans(w)\}$$

zu lösen, ist es nötig, die erste Hälfte des Wortes zwischenzuspeichern, damit sie mit der zweiten Hälfte des Wortes verglichen werden kann.[1]

Um die Technik des Erkennens endlicher Automaten beibehalten zu können, aber gleichzeitig auch in der Lage zu sein, mitzuzählen und sich beliebige Teile des gelesenen Wortes zu merken, werden die endlichen Automaten um einen Keller (Stapel, stack) als zweites Speichermedium erweitert. Ein Zustandsübergang wird dann auch vom obersten Kellersymbol abhängig gemacht, das dabei durch eine Sequenz von Kellersymbolen ersetzbar ist. Auf dem Keller werden in jedem Schritt also eine POP-Operation sowie eine Folge von PUSH-Operationen ausgeführt, die auch leer sein kann. Außerdem werden noch – anders als bei endlichen Automaten gemäß Abschnitt 3.1 – Zustandsübergänge erlaubt, bei denen kein Eingabesymbol gelesen wird.

11.1 Konzept des Kellerautomaten

Das Modell des Kellerautomaten formalisiert diese Beschreibung. Die Arbeitsweise des Kellerautomaten wird durch fortgesetzte Übergänge zwischen Konfigurationen beschrieben, wobei eine Konfiguration den aktuellen Zustand, das noch zu lesende Eingabewort und ein aktuelles Kellerwort umfasst. Einzelne Übergänge sind durch die Zustandsüberführung festgelegt. Ziel ist es, eine Anfangskonfiguration mit Anfangszustand, dem vollständigen Eingabewort und dem initialen Kellersymbol als Startkeller in eine Endkonfiguration zu überführen, bei der der aktuelle Zustand ein Endzustand und die Eingabe vollständig gelesen ist. Wenn das gelingt, ist das Eingabewort vom Kellerautomaten erkannt.

1. Ein *Kellerautomat* ist ein System $K = (Z, I, C, d, s_0, F, c_0)$ mit einer endlichen Menge Z von *Zuständen*, einem endlichen Alphabet I von *Eingaben*, einem endlichen Alphabet C von *Kellersymbolen*, einer *Zustandsüberführung* d, die jedem Zustand s, jeder Eingabe x und jedem Kellersymbol c eine Menge von Paaren aus Zuständen und Kellerwörtern $d(s, x, c) \subseteq Z \times C^*$ sowie jedem Zustand s und jedem Kellersymbol c eine entsprechende Menge $d(s, -, c) \subseteq Z \times C^*$ zuordnet, einem *Anfangszustand* $s_0 \in Z$, einer Menge von *Endzuständen* $F \subseteq Z$ und einem *initialen Kellersymbol* c_0.
2. Ein Kellerautomat lässt sich ähnlich einem endlichen Automaten graphisch darstellen. Unterschiedlich ist lediglich die Beschriftung der Kanten, wie Abbildung 7 illustriert.

[1]Dabei wird vorausgesetzt, dass das Eingabewort nur einmal von links nach rechts gelesen werden kann.

für $(s', \alpha) \in d(s, x, c)$ für $(s', \alpha) \in d(s, -, c)$

Abbildung 7: Kantenbeschriftungen bei Kellerautomaten

3. Eine *Konfiguration* (s, v, γ) besteht aus einem Zustand $s \in Z$, einem Eingabewort $v \in I^*$ und einem Kellerwort $\gamma \in C^*$. Für $w \in I^*$ ist (s_0, w, c_0) die *Anfangskonfiguration* von w. Und (s'', λ, γ) wird *Endkonfiguration* genannt, falls $s'' \in F$ (bei beliebigem Kellerwort γ).
Falls $(s', \alpha) \in d(s, x, c)$, so hat die Konfiguration $(s, xv, c\gamma)$ die *Folgekonfiguration* $(s', v, \alpha\gamma)$; falls $(s', \alpha) \in d(s, -, c)$, so hat die Konfiguration $(s, v, c\gamma)$ die *Folgekonfiguration* $(s', v, \alpha\gamma)$.
Ist con' eine Folgekonfiguration von con, so schreibt man dafür auch $con \vdash\!\!\!- con'$.
Eine Folge von n solchen direkten Übergängen
$$con = con_0 \vdash\!\!\!- con_1 \vdash\!\!\!- \cdots \vdash\!\!\!- con_n = con'$$
(für $n \in \mathbb{N}$) kann durch $con \overset{n}{\vdash\!\!\!-} con'$ oder $con \overset{*}{\vdash\!\!\!-} con'$ abgekürzt werden.
4. Ein Wort $w \in I^*$ wird von K *erkannt*, falls die Anfangskonfiguration von w in eine Endkonfiguration überführbar ist, d.h. es existieren $s'' \in F$ und $\gamma \in C^*$ mit $(s_0, w, c_0) \overset{*}{\vdash\!\!\!-} (s'', \lambda, \gamma)$.
Die Menge aller von K erkannten Wörter bildet die *erkannte Sprache* $L(K)$.

11.2 Deterministische Kellerautomaten

Bei einem Kellerautomaten kann eine Konfiguration mehrere Folgekonfigurationen haben, so dass das Erkennungsverfahren nichtdeterministisch ist. Es wird deterministisch, wenn es jeweils höchstens eine Folgekonfiguration gibt, was offenbar für folgende Kellerautomaten gilt:

Ein Kellerautomat $K = (Z, I, C, d, s_0, F, c_0)$ ist *deterministisch*, wenn für jedes $s \in Z$ und $c \in C$ die Mengen $d(s, x, c)$ für alle $x \in I$ und $d(s, -, c)$ leer oder einelementig sind und $d(s, -, c)$ höchstens dann nicht leer ist, wenn alle $d(s, x, c)$ leer sind.

Ohne Beweis sei angemerkt, dass es nicht möglich ist, zu jedem Kellerautomaten einen deterministischen Kellerautomaten zu konstruieren, der dieselbe Sprache erkennt. So gibt es z.B. einen Kellerautomaten, der die Sprache $L_{mirror} = \{w\,trans(w) \mid w \in \{a,b\}^*\}$ erkennt, aber keinen deterministischen Kellerautomaten.

Ebenfalls ohne Beweis sei festgehalten, dass deterministische Kellerautomaten das Wortproblem ihrer erkannten Sprachen wie endliche Automaten in linearer Zeit lösen, d.h. die Zahl der Berechnungsschritte ist proportional zur Länge der Eingabewörter. Deshalb wird in der Praxis des Compilerbaus in der Regel versucht, die Syntaxanalyse von Programmiersprachen mit Hilfe von deterministischen Kellerautomaten oder ähnlich funktionierenden Verfahren zu bewerkstelligen.

11.3 Beispiel: Reguläre Ausdrücke

Reguläre Ausdrücke über I (vgl. Abschnitt 6.4) werden von dem in Abbildung 8 dargestellten deterministischen Automaten erkannt (wobei $y \in I \cup \{empty, lambda\}$, $c \in \{op, br, c_0\}$ und $\oplus \in \{+, \circ\}$).

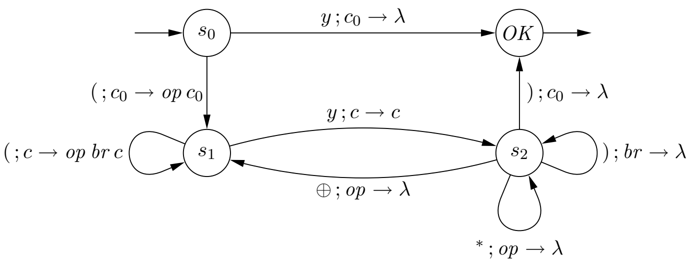


Abbildung 8: Ein deterministischer Kellerautomat, der $REX(I)$ erkennt

Durch folgende Konfigurationsfolge wird der Ausdruck $(((x+empty)^*)\circ lambda)$ als regulär erkannt:

$$\begin{array}{lllr}
 & (s_0, & (((x+empty)^*)\circ lambda), & c_0) \\
\vdash & (s_1, & ((x+empty)^*)\circ lambda), & op\,c_0) \\
\vdash & (s_1, & (x+empty)^*)\circ lambda), & op\,br\,op\,c_0) \\
\vdash & (s_1, & x+empty)^*)\circ lambda), & op\,br\,op\,br\,op\,c_0) \\
\vdash & (s_2, & +\,empty)^*)\circ lambda), & op\,br\,op\,br\,op\,c_0) \\
\vdash & (s_1, & empty)^*)\circ lambda), & br\,op\,br\,op\,c_0) \\
\vdash & (s_2, &)^*)\circ lambda), & br\,op\,br\,op\,c_0) \\
\vdash & (s_2, & ^*)\circ lambda), & op\,br\,op\,c_0) \\
\vdash & (s_2, &)\circ lambda), & br\,op\,c_0) \\
\vdash & (s_2, & \circ\, lambda), & op\,c_0) \\
\vdash & (s_1, & lambda), & c_0) \\
\vdash & (s_2, &), & c_0) \\
\vdash & (OK, & \lambda, &)
\end{array}$$

Die folgende Konfigurationsfolge zeigt, dass der Ausdruck $((x+empty)^*)\circ lambda)$ nicht regulär ist:

$$\begin{array}{lllr}
 & (s_0, & ((x+empty)^*)\circ lambda), & c_0) \\
\vdash & (s_1, & (x+empty)^*)\circ lambda), & op\,c_0) \\
\vdash & (s_1, & x+empty)^*)\circ lambda), & op\,br\,op\,c_0) \\
\vdash & (s_2, & +\,empty)^*)\circ lambda), & op\,br\,op\,c_0) \\
\vdash & (s_1, & empty)^*)\circ lambda), & br\,op\,c_0) \\
\vdash & (s_2, &)^*)\circ lambda), & br\,op\,c_0) \\
\vdash & (s_2, & ^*)\circ lambda), & op\,c_0) \\
\vdash & (s_2, &)\circ lambda), & c_0) \\
\vdash & (OK, & \circ\, lambda), &)
\end{array}$$

Denn die letzte Konfiguration besitzt keine Folgekonfiguration, ist aber auch keine Endkonfiguration.

12 Von kontextfreien Grammatiken zu Kellerautomaten

Kontextfreie Grammatiken haben nur Regeln, deren linke Seiten einzelne nichtterminale Zeichen sind. Da die Syntax von Programmiersprachen üblicherweise mit Hilfe solcher Regeln definiert wird, sind kontextfreie Grammatiken und die Lösung ihres Wortproblems besonders interessant. Es trifft sich deshalb gut, dass kontextfreie Grammatiken korrekt in Kellerautomaten übersetzt werden können. Dabei bedeutet Korrektheit, dass jede eingegebene Grammatik dieselbe Sprache erzeugt wie der aus der Eingabe konstruierte Automat erkennt. Der konstruierte Automat löst also das Wortproblem der Eingabegrammatik. Da Kellerautomaten im allgemeinen nichtdeterministisch arbeiten, ist diese Lösung jedoch nicht polynomiell (wenn man den Nichtdeterminismus z.B. durch Breitensuche vermeidet). Leider lässt sich im Gegensatz zu endlichen Automaten nicht jeder Kellerautomat in einen deterministischen umbauen, ohne dass sich die erkannte Sprache ändert.

12.1 Der Übersetzer

Zu jeder kontextfreien Grammatik wird ein Kellerautomat konstruiert, in dessen Keller praktisch der Ableitungsprozess der Eingabegrammatik abläuft. Sonstige Zustandsübergänge dienen lediglich dem richtigen Starten und Halten.

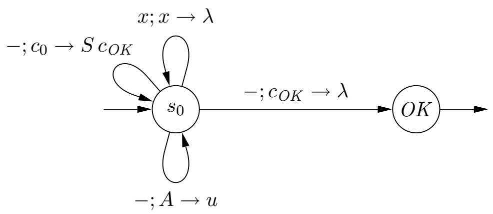
Sei $G = (N, T, P, S)$ eine kontextfreie Grammatik und $c_0, c_{OK} \notin N \cup T$. Dann ist der zugehörige Kellerautomat $PDA(G)$ [2] gegeben durch:

$$PDA(G) = (\{s_0, OK\}, T, N \cup T \cup \{c_0, c_{OK}\}, d_G, s_0, \{OK\}, c_0)$$

mit

(i) $d_G(s_0, -, c_0) = \{(s_0, S\, c_{OK})\}$,
(ii) $d_G(s_0, -, A) = \{(s_0, u) \mid A ::= u \in P\}$,
(iii) $d_G(s_0, x, x) = \{(s_0, \lambda)\}$ für alle $x \in T$ und
(iv) $d_G(s_0, -, c_{OK}) = \{(OK, \lambda)\}$.

Der konstruierte Automat kann wie folgt veranschaulicht werden (wobei $x \in T$ und $A ::= u \in P$):



[2] *PDA* steht für die englische Bezeichnung *pushdown automaton* für Kellerautomat.

Theorem 9 (Korrektheit der Übersetzung)
Sei G eine kontextfreie Grammatik und PDA(G) der zugehörige Kellerautomat. Dann gilt: $L(G) = L(PDA(G))$.

Die Situation der Übersetzung von kontextfreien Grammatiken in Kellerautomaten und ihre Korrektheit lassen sich durch das Diagramm in Abbildung 9 veranschaulichen.

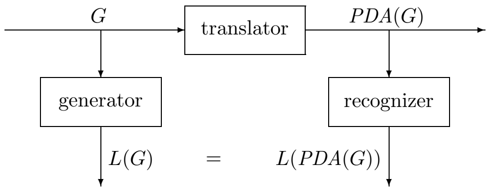


Abbildung 9: Korrekte Übersetzung kontextfreier Grammatiken in Kellerautomaten

Der Beweis von Theorem 9 wird auf ein späteres Kapitel verschoben, da dafür einige Eigenschaften von kontextfreien Grammatiken benötigt werden, die erst in den folgenden Kapiteln erarbeitet werden.

Zunächst soll die Übersetzung mit einem Beispiel veranschaulicht werden.

12.2 Beispiel: Klammergebirge

Die kontextfreie Grammatik $G = (\{A\}, \{\,[\,,\,]\,\}, \{A ::= AA \mid [A] \mid \lambda\}, A)$ erzeugt alle korrekten Klammerungen über dem Klammerpaar [und] . Der zugehörige Kellerautomat $PDA(G)$ ist in Abbildung 10 dargestellt.

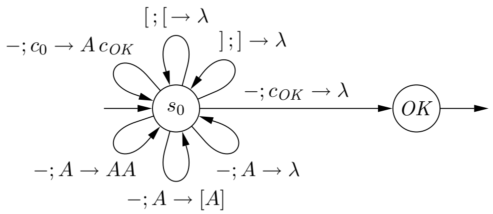


Abbildung 10: Ein zu einer kontextfreien Grammatik gehörender Kellerautomat

Wie die in der linken Hälfte von Abbildung 11 angegebene Berechnung zeigt, wird die Klammerung [ ][ ][[ ]] von $PDA(G)$ erkannt. Dieser Berechnung entspricht die rechts dane-

$$
\begin{array}{llrrcll}
 & (s_0, & [\,][\,][[\,]], & c_0) & & & \\
\vdash & (s_0, & [\,][\,][[\,]], & A\,c_{OK}) & & & \boldsymbol{A} \\
\vdash & (s_0, & [\,][\,][[\,]], & AA\,c_{OK}) & \longrightarrow & & \boldsymbol{A}A \\
\vdash & (s_0, & [\,][\,][[\,]], & AAA\,c_{OK}) & \longrightarrow & & \boldsymbol{A}AA \\
\vdash & (s_0, & [\,][\,][[\,]], & [A]AA\,c_{OK}) & \longrightarrow & & [A]AA \\
\vdash & (s_0, &][\,][[\,]], & A]AA\,c_{OK}) & = & [& \boldsymbol{A}]AA \\
\vdash & (s_0, &][\,][[\,]], &]AA\,c_{OK}) & \longrightarrow & [&]AA \\
\vdash & (s_0, & [\,][[\,]], & AA\,c_{OK}) & = & [\,] & \boldsymbol{A}A \\
\vdash & (s_0, & [\,][[\,]], & [A]A\,c_{OK}) & \longrightarrow & [\,] & [A]A \\
\vdash & (s_0, &][[\,]], & A]A\,c_{OK}) & = & [\,][& \boldsymbol{A}]A \\
\vdash & (s_0, &][[\,]], &]A\,c_{OK}) & \longrightarrow & [\,][&]A \\
\vdash & (s_0, & [[\,]], & A\,c_{OK}) & = & [\,][\,] & \boldsymbol{A} \\
\vdash & (s_0, & [[\,]], & [A]\,c_{OK}) & \longrightarrow & [\,][\,] & [A] \\
\vdash & (s_0, & [\,]], & A]\,c_{OK}) & = & [\,][\,][& \boldsymbol{A}] \\
\vdash & (s_0, & [\,]], & [A]]\,c_{OK}) & \longrightarrow & [\,][\,][& [A]] \\
\vdash & (s_0, &]], & A]]\,c_{OK}) & = & [\,][\,][[& \boldsymbol{A}]] \\
\vdash & (s_0, &]], &]]\,c_{OK}) & \longrightarrow & [\,][\,][[&]] \\
\vdash & (s_0, &], &]\,c_{OK}) & = & [\,][\,][[\,] &] \\
\vdash & (s_0, & \lambda, & c_{OK}) & = & [\,][\,][[\,]] & \\
\vdash & (OK, & \lambda, & \lambda) & & &
\end{array}
$$

Abbildung 11: Eine Berechnung des Kellerautomaten und die zugehörige Ableitung in der Grammatik

ben angegebene Ableitung in G, wobei das jeweils im nächsten Schritt ersetzte nichtterminale Zeichen fett gedruckt ist.

Offenbar arbeitet die Zustandsüberführung von $PDA(G)$ wie folgt: Zur Anfangskonfiguration passt nur der Zustandsübergang nach (i), so dass in der Folgekonfiguration das Startsymbol zuoberst auf dem Keller steht. Ein Zustandsübergang nach (ii) kann ausgeführt werden, falls oben auf dem Keller ein Nichtterminal der Grammatik steht. Wenn es mehrere Übergänge für dieses Zeichen gibt, "rät" der Kellerautomat nichtdeterministisch, welche Produktion in der Ableitung angewendet wird, um das nächste Teilstück des Eingabewortes zu erzeugen. Ein Zustandsübergang nach (iii) wird ausgeführt, wenn das oberste Kellersymbol ein Terminalzeichen ist und dieses auch gerade dem nächsten gelesenen Zeichen gleicht. Damit das Eingabewort erkannt wird, muss zuletzt der Zustandsübergang nach (iv) ausgeführt werden.

Insgesamt wird beim Vergleich der Konfigurationsfolge und der Ableitung deutlich, dass der Kellerautomat alle in dem entsprechenden abgeleiteten Wort links von der Lücke stehenden Zeichen bereits gelesen und alle rechts stehenden Zeichen gerade auf dem Keller hat. Weiter fällt auf, dass in jedem Schritt der Ableitung das am weitesten links stehende Nichtterminal ersetzt wird, d.h. die Ableitung ist eine sogenannte *Linksableitung*. Dies ist kein Zufall, sondern liegt daran, dass der Kellerautomat immer nur das oberste Zeichen aus seinem Keller entfernen kann.

Damit liegt die Vermutung nahe, dass der zu einer kontextfreien Grammatik gehörige Kellerautomat genau dann ein Wort erkennt, wenn dieses Wort mit einer Linksableitung der Grammatik erzeugt werden kann. In Theorem 9 wird aber behauptet, dass der Kellerautomat jedes von der Grammatik erzeugte Wort erkennt, unabhängig davon, wie dieses Wort abgeleitet wurde. Um dieses Problem zu lösen, wird später gezeigt, dass jede terminierende Ableitung einer kontextfreien Grammatik in eine Linksableitung umgebaut werden kann, ohne das erzeugte Wort zu verändern. Für den Beweis, dass diese Umbautechnik das Gewünschte leistet, wird eine wesentliche Eigenschaft von kontextfreien Grammatiken benötigt, die im nächsten Kapitel als *Kontextfreiheitslemma* formuliert ist.

Ohne das näher auszuführen, sei angemerkt, dass sich auch umgekehrt jeder Kellerautomat K korrekt in eine kontextfreie Grammatik $G(K)$ übersetzen lässt, d.h. $L(K) = L(G(K))$.

13 Kontextfreiheitslemma

Da die linke Seite einer kontextfreien Regel nur aus einem Zeichen besteht, ist die Stelle eines Wortes, auf die diese Regel angewendet wird, in einem sehr strengen Sinn lokalisierbar: Wenn das Wort zerteilt wird, ist das ersetzte Zeichen in genau einem der Teile. Auf Ableitungen übertragen, bedeutet das, dass die einzelnen Ableitungsschritte auf Teile des Ausgangswortes verteilbar sind, wobei eine horizontale Zerlegung von Ableitungen entsteht. Diese als Kontextfreiheitslemma bezeichnete Eigenschaft ist recht offensichtlich, hat aber viele interessante Konsequenzen.

Theorem 10 (Kontextfreiheitslemma)
Sei $G = (N, T, P, S)$ eine kontextfreie Grammatik, $u \xrightarrow[P]{n} v$ eine Ableitung der Länge n und $u = u_1u_2 \cdots u_k$ eine Zerlegung von u in k Teilwörter.

Dann gibt es k Ableitungen $u_i \xrightarrow[P]{n_i} v_i$, so dass $v = v_1 \cdots v_k$ und $n = \sum\limits_{i=1}^{k} n_i$.

Beweis (mit Induktion über n)**.**

IA: $n = 0$. Dann wähle $v_i = u_i$ und $u_i \xrightarrow[P]{0} u_i$.

IV: Die Behauptung gelte für n.

IS: Betrachte $u \xrightarrow[P]{n+1} v$. Der erste Schritt hat dann die Form $u = u'Au'' \xrightarrow[A::=r]{} u'ru'' = \bar{u}$. Da A ein einzelnes Zeichen ist, existiert ein i_0, so dass $u_{i_0} = u'_{i_0}Au''_{i_0}$ und $u' = u_1 \cdots u_{i_0-1}u'_{i_0}$ sowie $u'' = u''_{i_0}u_{i_0+1} \cdots u_k$. Wähle nun $\bar{u}_i = u_i$ für $i \neq i_0$ und $\bar{u}_{i_0} = u'_{i_0}ru''_{i_0}$, so dass insbesondere $u_{i_0} \xrightarrow[A::=r]{} \bar{u}_{i_0}$ gilt. Außerdem gilt nach Konstruktion:

$$\bar{u} = u'ru'' = u_1 \cdots u_{i_0-1}u'_{i_0}ru''_{i_0}u_{i_0+1} \cdots u_k = \bar{u}_1 \cdots \bar{u}_k.$$

Auf die restlichen n Ableitungsschritte $\bar{u} \xrightarrow[P]{n} v$ ist nun die Induktionsvoraussetzung anwendbar, was Ableitungen $\bar{u}_i \xrightarrow[P]{n_i} v_i$ mit $v = v_1 \cdots v_k$ und $\sum\limits_{i=1}^{k} n_i = n$ ergibt. Für i_0 kann das zu $u_{i_0} \xrightarrow[A::=r]{} \bar{u}_{i_0} \xrightarrow[P]{n_{i_0}} v_{i_0}$, einer Ableitung der Länge $n_{i_0}+1$, zusammengesetzt werden. Für $i \neq i_0$ können die Ableitungen wegen $u_i = \bar{u}_i$ bleiben, wie sie sind. Die Zerlegung von v bleibt unverändert, die Summe der Ableitungslängen ergibt $n + 1$. Damit ist alles gezeigt. □

Es sei noch angemerkt, dass die Umkehrung des Kontextfreiheitslemmas ebenfalls gilt. Ableitungen $u_i \xrightarrow[P]{n_i} v_i$ für $i = 1, \ldots, k$ können immer zu einer Ableitung

$$u = u_1 \cdots u_k \xrightarrow[P]{n} v_1 \cdots v_k = v$$

mit $n = \sum\limits_{i=1}^{k} n_i$ zusammengesetzt werden.

14 Linksableitungen

Wenn man in einer kontextfreien Grammatik beim Ableiten in jedem Schritt das ganz links stehende nichtterminale Zeichen ersetzt, spricht man von Linksableitungen.[3] Trennt man dabei die links stehenden terminalen Zeichen vorher ab, so handelt es sich gerade um Schritte, die der aus einer kontextfreien Grammatik konstruierte Kellerautomat auf dem Keller vollführt. In diesem Kapitel wird gezeigt, dass jede Ableitung einer kontextfreien Grammatik in eine Linksableitung umgebaut werden kann. Es stellt sich also heraus, dass das Ableiten in der Grammatik dieselbe Leistungsfähigkeit hat wie das Bilden von Folgekonfigurationen im zugehörigen Kellerautomaten.

Sei $G = (N, T, P, S)$ eine kontextfreie Grammatik. Eine *Linksableitung* ist eine Ableitung $u_1 \xrightarrow[P]{} u_2 \xrightarrow[P]{} \cdots \xrightarrow[P]{} u_n$, bei der in jedem Schritt $u_i \xrightarrow[A_i::=v_i]{} u_{i+1}$ $(1 \leq i < n)$ das am weitesten links stehende Nichtterminal ersetzt wird, d.h. $u_i = x_iA_iy_i$ und $u_{i+1} = x_iv_iy_i$ mit $x_i \in T^*$. Um anzudeuten, dass eine Ableitung eine Linksableitung ist, wird der Pfeil $-\!\ell\!\!\rightarrow$ statt $\longrightarrow$ verwendet.

Das folgende Lemma impliziert, dass man sich bei kontextfreien Grammatiken auf die Betrachtung von Linksableitungen beschränken kann, ohne dass dies eine Auswirkung auf die erzeugte Sprache hat.

Lemma 11
Sei $G = (N, T, P, S)$ eine kontextfreie Grammatik. Dann lässt sich jede Ableitung $A \xrightarrow[P]{*} v$ mit $A \in N$, $v \in T^*$ in eine Linksableitung $A \overset{*}{\underset{P}{-\!\ell\!\!\rightarrow}} v$ umformen.

Beweis (Per Induktion über die Länge von Ableitungen)**.**

IA: Eine Ableitung $A \xrightarrow[P]{0} v$ mit $v \in T^*$ existiert nicht, also muss nichts gezeigt werden.

IV: Gelte die Behauptung für alle Ableitungen der Länge $\leq n$.

IS: Betrachte eine Ableitung $A \xrightarrow[P]{} u \xrightarrow[P]{n} v$. Dann lässt u sich in $u = u_0A_1u_1 \cdots A_mu_m$ mit $u_0, \ldots, u_m \in T^*$ und $A_1, \ldots, A_m \in N$ zerlegen. Für $i = 0, \ldots, m$ gilt $u_i \xrightarrow[P]{0} u_i$ und aufgrund des Kontextfreiheitslemmas auch $A_i \xrightarrow[P]{n_i} v_i$ für $i = 1, \ldots, m$, wobei $v = u_0v_1u_1 \cdots v_mu_m$ und $n_i \leq \sum\limits_{i=1}^{m} n_i = n$. Also kann nach Induktionsvoraussetzung jede der Ableitungen $A_i \xrightarrow[P]{n_i} v_i$ in eine Linksableitung $A_i \overset{*}{\underset{P}{-\!\ell\!\!\rightarrow}} v_i$ umgeformt werden.

[3] In der Literatur ist es auch gebräuchlich, eine Ableitung Linksableitung zu nennen, wenn alle nichtterminalen Zeichen links vom gerade ersetzten im Rest der Ableitung nicht mehr ersetzt werden. Bei Ableitungen von terminalen Wörtern macht das keinen Unterschied, aber für den Nachweis der Korrektheit von Theorem 9 ist diese Definition weniger geeignet.

In der richtigen Reihenfolge zusammengesetzt, erhält man

$$\begin{aligned} A &\xrightarrow[P]{\ell} u_0 A_1 u_1 \cdots A_m u_m \\ &\xrightarrow[P]{\ell\,*} u_0 v_1 u_1 A_2 u_2 \cdots A_m u_m \\ &\xrightarrow[P]{\ell\,*} u_0 v_1 u_1 v_2 u_2 A_3 u_3 \cdots A_m u_m \\ &\;\vdots \\ &\xrightarrow[P]{\ell\,*} u_0 v_1 u_1 \cdots v_m u_m\,, \end{aligned}$$

also insgesamt eine Linksableitung $A \xrightarrow[P]{\ell\,*} v$. □

15 Korrektheit der Übersetzung von kontextfreien Grammatiken in Kellerautomaten

In diesem Kapitel wird der Beweis für Theorem 9 nachgeliefert. Dazu wird zunächst der enge Zusammenhang zwischen dem Ableiten in einer kontextfreien Grammatik und dem Bilden von Folgekonfigurationen in dem zugehörigen Kellerautomat festgehalten.

Sei in diesem Kapitel $G = (N, T, P, S)$ eine gegebene kontextfreie Grammatik und $PDA(G)$, der zu ihr gehörige Kellerautomat, wie in Abschnitt 12.1 definiert.

Lemma 12
Sei $w \in (N \cup T)^*$. Wenn es eine Linksableitung $S \xrightarrow{\ell\,*} w = u\bar{u}$ gibt, wobei $u \in T^*$ das längste terminale Anfangsstück von w ist, dann gibt es für beliebiges $v \in T^*$ eine Konfigurationsfolge $(s_0, uv, S\,c_{OK}) \vdash^* (s_0, v, \bar{u}\,c_{OK})$.

Beweis (mittels Induktion über die Ableitungslänge)**.**

IA: Für $S \xrightarrow{\ell\,0} w$ gilt $w = S$, d.h. $u = \lambda$ und $\bar{u} = S$, und daher

$$(s_0, uv, S\,c_{OK}) = (s_0, v, \bar{u}\,c_{OK}) \vdash^0 (s_0, v, \bar{u}\,c_{OK}).$$

IV: Die Behauptung gelte für alle Linksableitungen der Länge n.

IS: Betrachte nun eine Linksableitung $S \xrightarrow{\ell\,n} u_1 A\bar{u}_1 \xrightarrow{\ell} u_1 r\bar{u}_1 = w$ der Länge $n+1$ und eine Zerlegung $w = u\bar{u}$, wobei u das längste terminale Anfangsstück von w ist. Da die Ableitung eine Linksableitung ist, gilt $u_1 \in T^*$. Also ist u_1 ein terminales Anfangsstück von w und damit auch Anfangsstück von u, d.h. es gibt ein $t \in T^*$ mit $u = u_1 t$ und $t\overline{u} = r\overline{u}_1$. Somit kann man die Konfigurationsfolge

$$\begin{aligned} (s_0, uv, S\,c_{OK}) &= (s_0, u_1 tv, S\,c_{OK}) \\ &\vdash^* (s_0, tv, A\overline{u}_1\,c_{OK}) \\ &\vdash (s_0, tv, r\overline{u}_1\,c_{OK}) \\ &= (s_0, tv, t\overline{u}\,c_{OK}) \\ &\vdash^* (s_0, v, \overline{u}\,c_{OK}) \end{aligned}$$

konstruieren, wobei sich die zweite Zeile aus der Anwendung der Induktionsvoraussetzung für die Linksableitung $S \xrightarrow{\ell\,*} u_1 A\bar{u}_1$, die dritte Zeile aus Zeile (ii) der Definition von d_G und die letzte Zeile aus Zeile (iii) der Definition von d_G ergibt.

□

Lemma 13
Seien $u, v \in T^*$. Wenn es eine Konfigurationsfolge $(s_0, uv, S\,c_{OK}) \vdash^* (s_0, v, \bar{u}\,c_{OK})$ gibt, dann auch eine Linksableitung $S \xrightarrow{\ell\,*} u\bar{u}$.

Beweis (mittels Induktion über die Länge der Konfigurationsfolge)**.**

IA: Für $(s_0, uv, S\,c_{OK}) \vdash^{0} (s_0, v, \bar{u}\,c_{OK})$ gilt $u = \lambda$ und $\bar{u} = S$, woraus $S \xrightarrow[\ell]{0} S = u\bar{u}$ folgt.

IV: Gelte die Behauptung für Konfigurationsfolgen der Länge n.

IS: Betrachte $(s_0, uv, S\,c_{OK}) \vdash^{n+1} (s_0, v, \bar{u}\,c_{OK})$. Da die Zustandsüberführung nie c_0 auf den Keller schreibt, kann keine der Folgekonfigurationen nach Zeile (i) der Definition von d_G gebildet worden sein. Damit scheidet auch Zeile (iv) aus, denn sonst würde c_{OK} nicht mehr auf den Keller stehen. Also ist die letzte Folgekonfiguration nach Zeile (ii) oder (iii) der Definition von d_G gebildet worden.

1. Entsprechend Zeile (ii) hat die Konfigurationsfolge die Form
$$(s_0, uv, S\,c_{OK}) \vdash^{n} (s_0, v, A\bar{u}_1\,c_{OK}) \vdash (s_0, v, \bar{u}_0\bar{u}_1\,c_{OK}) = (s_0, v, \bar{u}\,c_{OK})$$
und $A ::= \bar{u}_0$ ist eine Regel in P. Zusammen mit der Induktionsvoraussetzung für die Konfigurationsfolge $(s_0, uv, S\,c_{OK}) \vdash^{n} (s_0, v, A\bar{u}_1\,c_{OK})$ ergibt sich dann die Linksableitung
$$S \xrightarrow[\ell]{*} uA\bar{u}_1 \xrightarrow[\ell]{} u\bar{u}_0\bar{u}_1 = u\bar{u}.$$

2. Entsprechend Zeile (iii) hat die Konfigurationsfolge die Form
$$(s_0, uv, S\,c_{OK}) \vdash^{n} (s_0, xv, x\bar{u}\,c_{OK}) \vdash (s_0, v, \bar{u}\,c_{OK}).$$
Dann hat also u die Form $u = u_0x$, und aus der Induktionsvoraussetzung für die Konfigurationsfolge $(s_0, u_0xv, S\,c_{OK}) \vdash^{n} (s_0, xv, x\bar{u}\,c_{OK})$ ergibt sich die Linksableitung $S \xrightarrow[\ell]{*} u_0x\bar{u} = u\bar{u}$. □

Mit Hilfe der beiden Lemmata kann nun folgendermaßen geschlossen werden.

Beweis von Theorem 9**.**
Sei $w \in L(G)$, d.h. $S \xrightarrow{*} w$ und $w \in T^*$. Damit existiert nach Lemma 11 eine Linksableitung $S \xrightarrow[\ell]{*} w$. Da w schon das längste terminale Anfangsstück von w ist, folgt mit Lemma 12, dass es eine Konfigurationsfolge $(s_0, w, S\,c_{OK}) \vdash^{*} (s_0, \lambda, \lambda\,c_{OK})$ gibt, wobei $\bar{u} = \lambda$ sein muss und $v = \lambda$ gewählt wurde. Mit den Zeilen (i) und (iv) der Definition von d_G lässt sich diese Folgekonfigurationsbildung vorn und hinten verlängern zu:
$$(s_0, w, c_0) \vdash (s_0, w, S\,c_{OK}) \vdash^{*} (s_0, \lambda, c_{OK}) \vdash (OK, \lambda, \lambda). \qquad (*)$$
Das bedeutet aber gerade $w \in L(PDA(G))$.

Sei umgekehrt $w \in L(PDA(G))$, d.h. $(s_0, w, c_0) \vdash^{*} (OK, \lambda, \gamma)$ für irgendein γ. Das lässt sich immer in die Form $(*)$ zerlegen, denn der erste und letzte Schritt können nicht anders aussehen. Der mittlere Teil impliziert nach Lemma 13 $S \xrightarrow{*} w\lambda = w$. Somit gilt $w \in L(G)$.

Insgesamt sind die beiden Sprachen also gleich. □

16 Ableitungsbäume

Dass das Wortproblem kontextfreier Sprachen mit Hilfe von Kellerautomaten gelöst werden kann, liegt wesentlich am Konzept der Linksableitungen. Denn sie erlauben, Ableitbarkeit und Erzeugbarkeit von Wörtern durch Operationen auf einem Keller zu realisieren. Ableitungsbäume sind ein vergleichbares Konzept zur Repräsentation kontextfreier Ableitungen. Die Konstruktion von Ableitungsbäumen beruht auf dem Kontextfreiheitslemma und wird in Abschnitt 16.1 vorgestellt. Sie gestattet, bekannte Eigenschaften von Bäumen für die Untersuchung von Ableitungen zu nutzen.

Um das zu demonstrieren, werden einige Beispiele für die vorteilhafte Nutzung der Baumstruktur gegeben. Sie alle werden im Kapitel 17 zum Nachweis des Pumping-Lemmas für kontextfreie Sprachen verwendet. In Abschnitt 16.2 wird nachgewiesen, dass man aus der Länge des Resultats eines Ableitungsbaumes dessen Höhe abschätzen kann. In Abschnitt 16.3 wird zu jedem Ableitungsbaum ein Weg von der Wurzel zu einem Blatt konstruiert, der so lang ist, wie der Baum hoch ist. In Abschnitt 16.4 schließlich werden das An- und Abhängen von Ableitungsbäumen betrachtet.

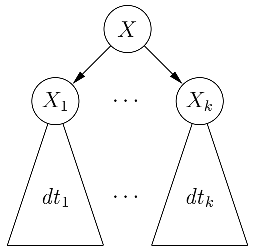
16.1 Konstruktion

Sei $G = (N, T, P, S)$ eine kontextfreie Grammatik ohne λ-Produktionen, d.h. ohne Produktionen mit dem leeren Wort als rechter Seite.

Dann kann jeder Ableitung, die in einem einzelnen Symbol beginnt, folgendermaßen rekursiv ein *Ableitungsbaum* zugeordnet werden:

(i) Für $X \xrightarrow{0} X$ mit $X \in N \cup T$ besteht der zugeordnete *Ableitungsbaum* aus einem mit X markierten Knoten mit der *Höhe* 0 und X als *Resultat*. Der einzelne Knoten ist sowohl *Wurzel* als auch *Blatt*. Dieser Ableitungsbaum wird mit $dt(X \xrightarrow{0} X)$ bezeichnet.

(ii) Für $X \xrightarrow{n+1} U$ sei $X ::= X_1 \cdots X_k$ mit $X_i \in N \cup T$ die erste angewendete Regel, und seien $X_i \xrightarrow{n_i} U_i$ die korrespondierenden Ableitungen gemäß dem Kontextfreiheitslemma. Wegen $n_i \leq n$ kann angenommen werden, dass Ableitungsbäume $dt_i = dt(X_i \xrightarrow{n_i} U_i)$ mit einer jeweiligen Höhe, U_i als Resultat und einem Knoten X_i als Wurzel bereits existieren. Dann hat der Ableitungsbaum $dt(X \xrightarrow{n+1} U)$ die Form



mit einer *Höhe*, die um 1 größer als die größte Höhe der angehängten Ableitungsbäume dt_i ist, mit U als *Resultat*, dem obersten Knoten als *Wurzel* und den Blättern der angehängten Teilbäume als *Blätter*.

Ist dt ein Ableitungsbaum und bezeichnet man seine Höhe mit $height(dt)$ und sein Resultat mit $res(dt)$, so gilt nach Konstuktion:

(i) $height(dt(X \xrightarrow{0} X)) = 0$ und $res(dt(X \xrightarrow{0} X)) = X$,

(ii) $height(dt(X \xrightarrow{n+1} U)) = 1 + max\{height(dt_i) \mid i = 1, \ldots, k\}$
und $res(dt(X \xrightarrow{n+1} U)) = U = U_1 \cdots U_k = res(dt_1) \cdots res(dt_k)$.

16.2 Beziehung zwischen Baumhöhe und Resultatslänge

Gemäß der rekursiven Definition erhöht sich die Höhe um 1, wenn man k Bäume an eine Wurzel hängt, während sich die Länge des Resultats als Summe der Längen der Resultate der angehängten Bäume ergibt. Daraus ergibt sich folgende Beziehung zwischen Höhe und Resultatslänge.

Lemma 16.1
Sei b die Länge der längsten rechten Seite von Produktionen aus P. Dann gilt für alle Ableitungsbäume dt:

$$|res(dt)| \leq b^{height(dt)}.$$

Beweis (mit Induktion über die Höhe):

IA: $height(dt) = 0$ bedeutet $dt = dt(X \xrightarrow{0} X)$ für ein geeignetes $X \in N \cup T$. Somit gilt:

$$|res(dt)| = |res(dt(X \xrightarrow{0} X)))| = |X| = 1 \leq 1 = b^0 = b^{height(dt)}.$$

IV: Die Behauptung gelte für alle Höhen bis m.

IS: Betrachte nun $height(dt) = m+1$. Dann korrespondiert dt gemäß der rekursiven Definition zu einer Ableitung $X \xrightarrow{n+1} U$ (mit $n \geq m$), einer ersten Regelanwendung $X \longrightarrow X_1 \cdots X_k$ und induzierten Ableitungen $X_i \xrightarrow{n_i} U_i$ $(i = 1, \ldots, k)$. Bezeichne dt_i den korrespondierenden Ableitungsbaum $dt(X_i \xrightarrow{n_i} U_i)$. Wegen $height(dt) = 1 + max\{height(dt_i) \mid i = 1, \ldots, k\}$ ist dann $height(dt_i) \leq m$, so dass nach Induktionsvoraussetzung gilt:

$$|res(dt_i)| \leq b^{height(dt_i)}.$$

Daraus folgt wunschgemäß :

$$\begin{aligned}
|res(dt)| &= |res(dt_1) \cdots res(dt_k)| \\
&= \sum_{i=1}^{k} |res(dt_i)| \\
&\leq \sum_{i=1}^{k} b^{height(dt_i)} \\
&\leq \sum_{i=1}^{k} b^{m} \\
&= k \cdot b^m \\
&\leq b \cdot b^m \\
&= b^{height(dt)}.
\end{aligned}$$

□

16.3 Ein langer Weg von der Wurzel zu einem Blatt

In jedem Baum gibt es mindestens einen Weg von der Wurzel zu einem Blatt, der so lang ist, wie der Baum hoch ist.

Lemma 16.2
Sei dt ein Ableitungsbaum. Dann gibt es in dt einen Weg

$$A_0 \longrightarrow A_1 \longrightarrow \cdots \longrightarrow A_l$$

mit $l = height(dt)$, wobei der erste Knoten die Wurzel und der letzte ein Blatt ist.

Beweis (mit Induktion über die Höhe):

IA: $height(dt) = 0$ bedeutet $dt = dt(X \xrightarrow{0} X)$ für ein $X \in N \cup T$. Als Weg der Länge 0 kann (und muss) man dann den einzigen Knoten wählen, d.h. $A_0 = X$.

IV: Die Behauptung gelte für alle Ableitungsbäume der Höhe m.

IS: Betrachte einen Ableitungsbaum dt mit $height(dt) = m + 1$. Mit den Bezeichnungen aus dem Beweis von Lemma 16.1 gibt es dann ein $i_0 \in \{1, \ldots, k\}$, so dass $height(dt_{i_0}) = m$ ist, sowie eine Kante von der Wurzel von dt zur Wurzel von dt_{i_0}. Nach Induktionsvoraussetzung gibt es in dt_{i_0} einen Weg

$$A_1 \longrightarrow A_2 \longrightarrow \cdots \longrightarrow A_{m+1}$$

wobei der erste Knoten die Wurzel von dt_{i_0} ist, d.h. insbesondere $A_1 = X_{i_0}$, und der letzte Knoten ein Blatt von dt_{i_0} ist. Dieser Knoten ist auch ein Blatt von dt, so dass der gesuchte Weg entsteht, wenn man die eine Kante davorsetzt, wobei man $A_0 = X$ wählt. $\square$

16.4 Anhängen und Abhängen von Ableitungsbäumen

Nach Konstruktion führt von der Wurzel eines Ableitungsbaumes genau eine Kante zu der Wurzel jedes angehängten Ableitungsbaumes, was rekursiv bedeutet, dass von der Wurzel zu jedem Knoten genau ein Weg führt und jeder Knoten Wurzel genau eines Teilbaumes ist. Wenn man den abhängt (bis auf seine Wurzel), bleibt wieder ein Ableitungsbaum übrig. Genauer gilt für $dt = dt(X \xrightarrow{*} U)$ und für jeden Weg

$$X \longrightarrow A_1 \longrightarrow \quad \cdots \quad \longrightarrow A_l$$

von der Wurzel zu einem Knoten folgendes:

Es existieren Ableitungen $X \xrightarrow{*} U_1 A_l U_3$ und $A_l \xrightarrow{*} U_2$, so dass $dt_2 = dt(A_l \xrightarrow{*} U_2)$ der Teilbaum von dt ist, der A_l als Wurzel hat, und $dt_1 = dt(X \xrightarrow{*} U_1 A_l U_3)$ der Baum ist, der durch Löschen von dt_1 aus dt entsteht. Insbesondere liegt der obere Weg auch in dt_1 und führt dort zu einem Blatt.

Sind umgekehrt dt_1 und dt_2 Ableitungsbäume, so dass dt_1 den obigen Weg von der Wurzel zu einem Blatt enthält, und die Wurzel von dt_2 mit A_l markiert ist, kann man dt_2 an das Blatt A_l von dt_1 hängen, und man erhält einen Ableitungsbaum $dt_3 = dt(X \xrightarrow{*} U_1 A_l U_3 \xrightarrow{*} U_1 U_2 U_3)$. Sind dt_1 und dt_2 die beiden Bäume, die beim Abhängen aus dt entstehen, dann gilt $dt_3 = dt$ und damit insbesondere $U = U_1 U_2 U_3$.

Auf die vollständige Formalisierung dieser beiden Konstruktionen wird hier verzichtet.

17 Pumping-Lemma für kontextfreie Sprachen

In diesem Kapitel wird ein Pumping-Lemma für kontextfreie Sprachen formuliert und eine Beweisskizze gegeben. Analog zum Pumping-Lemma für reguläre Sprachen in Kapitel 7[4] eignet es sich zum Nachweis, dass bestimmte Sprachen nicht kontextfrei sind.

Das Pumping-Lemma für reguläre Sprachen beruht darauf, dass beim Erkennen eines genügend langen Wortes in einem endlichen Automaten eine Zustandsfolge durchlaufen wird, die einen Kreis enthält. Übertragen auf rechtslineare Grammatiken heißt das, dass es Ableitungen der Form $S \xrightarrow{*} xA, A \xrightarrow{*} yA$ und $A \xrightarrow{*} z$ mit $x, y, z \in T^*$ und $|y| > 0$ gibt, woraus sich Ableitungen der Form $S \xrightarrow{*} xy^i z$ für $i \in \mathbb{N}$ bilden lassen. Bei beliebigen kontextfreien Grammatiken kann man nicht erwarten, dass das nichtterminale Zeichen A immer ganz rechts steht, sondern irgendwo im abgeleiteten Wort. Aber auch Ableitungen der Form $S \xrightarrow{*} uAy, A \xrightarrow{*} vAx$ und $A \xrightarrow{*} w$ lassen sich zu Ableitungen $S \xrightarrow{*} uv^i wx^i y$ für $i \in \mathbb{N}$ zusammensetzen, was ebenfalls einen Pumpeffekt ergibt, bei dem allerdings zwei Teilwörter gleichzeitig iteriert werden. Es bleibt, die drei Ableitungen zu finden. Der schwierige Teil dabei ist, ein A zu finden, das in einem seiner abgeleiteten Wörter wieder auftaucht. Um das zu erreichen, werden die Ergebnisse über Ableitungsbäume herangezogen.

Sei $G = (N, T, P, S)$ eine kontextfreie Grammatik ohne λ-Produktionen (d.h. ohne Produktionen der Form $A ::= \lambda$) und ohne Kettenregeln (d.h. ohne Produktionen der Form $A ::= B$ mit $B \in N$). Für $z \in L(G)$ gibt es dann einen Ableitungsbaum $dt = dt(S \xrightarrow{*} z)$. Dessen Höhe ist nach Lemma 16.1 größer als $\sharp N$, falls $|z| > b^{\sharp N}$, wobei b wieder die Länge der längsten rechten Seite einer Produktion von G ist. Nach Lemma 16.2 gibt es dann in dt einen Weg von der Wurzel zu einem Blatt

$$A_0 \longrightarrow A_1 \longrightarrow \quad \cdots \quad \longrightarrow A_l$$

mit $l = \mathit{height}(dt)$ und $A_0 = S$. Die l Zeichen $A_0, \ldots, A_{l-1}$ sind nichtterminal, weil in einem Ableitungsbaum höchstens Blätter terminal sind. Wegen $l > \sharp N$ muss es Indizes $i < j$ geben mit $A_i = A_j$. Dieses nichtterminale Zeichen wird im folgenden auch mit A bezeichnet. Nach Abschnitt 16.4 induziert der Teilweg

$$S = A_0 \longrightarrow A_1 \longrightarrow \quad \cdots \quad \longrightarrow A_j = A$$

[4]Dort ist das Pumping-Lemma für die von endlichen Automaten erkannten Sprachen formuliert. Reguläre Sprachen sind genau die von endlichen Automaten erkannten; Regularität ist aber die viel griffigere Bezeichnung und wird deshalb hier verwendet.

zwei Ableitungen $S \xrightarrow{*} z_1Az_2$ und $A \xrightarrow{*} w$, derart dass $z = z_1wz_2$ und der obige Teilweg in $dt_1 = dt(S \xrightarrow{*} z_1Az_2)$ liegt und dort von der Wurzel zu einem Blatt führt. Entsprechend induziert der Teilweg

$$S = A_0 \longrightarrow A_1 \longrightarrow \cdots \longrightarrow A_i = A$$

von dt_1 zwei Ableitungen $S \xrightarrow{*} uAy$ und $A \xrightarrow{*} vAx$ mit $z_1Az_2 = uvAxy$. Beachte dabei, dass z_1 und z_2 terminal sind, weil z terminal ist, und dass deshalb auch u und y terminal sein müssen. Das nichtterminale Zeichen A in z_1Az_2 kann also nur in der zweiten Ableitung abgeleitet werden.

Damit sind die drei gesuchten Ableitungen gefunden. Daraus lassen sich induktiv Ableitungen der folgenden Form iterieren:

1. $S \xrightarrow{*} uAy (= uv^0Ax^0y)$; und
2. wenn $S \xrightarrow{*} uv^iAx^iy$ bereits konstruiert ist, erhält man $S \xrightarrow{*} uv^iAx^iy \xrightarrow{*} uv^ivAxx^iy = uv^{i+1}Ax^{i+1}y$.

Schließlich lässt sich jede dieser Ableitungen terminieren: $S \xrightarrow{*} uv^iAx^iy \xrightarrow{*} uv^iwx^iy$, so dass $uv^iwx^iy \in L(G)$ für alle $i \in \mathbb{N}$ folgt.

Wegen $i < j$ ist $A \xrightarrow{*} vAx$ keine Nullableitung. Und da G keine Kettenregeln enthält, können v und x nicht beide leer sein; d.h. $vx \neq \lambda$.

Wählt man darüber hinaus den ursprünglichen Weg als den längst möglichen und i ebenfalls so groß wie möglich, dann muss $l - i \leq \sharp N$ gelten, weil sonst zwei andere Knoten unterhalb von i gleich markiert wären, was der Wahl von i widerspricht. In diesem Fall kann in dem Ableitungsbaum $dt_2 = dt(A \xrightarrow{*} vAx \xrightarrow{*} vwx)$ kein Weg länger als $\sharp N + 1$ sein, weil sonst der ursprüngliche Weg nicht der längste gewesen wäre. Dann ist dt_2 aber auch nicht höher, und vwx hat höchstens die Länge $b^{\sharp N+1}$.

Die vorausgehenden Überlegungen ergeben das folgende Pumping-Lemma für kontextfreie Sprachen.

Theorem 14
Zu jeder kontextfreien Sprache L existiert eine natürliche Zahl $p \in \mathbb{N}$, so dass gilt:

Ist $z \in L$ mit $length(z) \geq p$, dann lässt sich z schreiben als $z = uvwxy$, wobei $length(vwx) \leq p$ ist und $vx \neq \lambda$, und für alle $i \in \mathbb{N}$ gilt:

$$uv^iwx^iy \in L.$$

Beweis: Zu jeder kontextfreien Sprache L existiert eine kontextfreie Grammatik G ohne λ-Produktionen und Kettenregeln, derart daß $L(G) = L - \{\lambda\}$ (vgl. z.B. [HMU02]). Wählt man nun $p = b^{\sharp N+1}$, dann bilden die Überlegungen vor dem Theorem den Rest des

Beweises. Beachte, daß es keine Rolle spielt, ob $\lambda \in L$ gilt, weil ohnehin nur Wörter ab einer bestimmten Länge pumpbar sein müssen.

Mit dem Pumping Lemma für kontextfreie Sprachen lässt sich z.B. zeigen, dass die Sprache $L_0 = \{a^nb^nc^n \mid n \in \mathbb{N}\}$ nicht kontextfrei ist. Dieses Ergebnis kann wiederum benutzt werden, um zu zeigen, dass die Klasse der kontextfreien Sprachen nicht abgeschlossen gegenüber Durchschnitt ist.

Korollar 15

1. *Die Sprache $L_0 = \{a^nb^nc^n \mid n \in \mathbb{N}\}$ ist nicht kontextfrei.*
2. *Die Klasse der kontextfreien Sprachen ist nicht abgeschlossen gegenüber Durchschnitt, d.h. es gibt kontextfreie Sprachen, deren Durchschnitt nicht kontextfrei ist.*

Beweis.

1. Angenommen, L_0 wäre kontextfrei. Sei p die zu L_0 gehörende Zahl aus dem Pumping-Lemma. Dann gibt es für $n \in \mathbb{N}$ mit $3n \geq p$ eine Zerlegung $a^nb^nc^n = uvwxy$, derart dass $uv^iwx^iy \in L$ für alle $i \leq p$, $length(vwx) \leq p$ und $vx \neq \lambda$.
   Enthielte v oder x a's und b's oder b's und c's, so käme in uv^2wx^2y im Widerspruch zur Definition von L_0 ein a hinter einem b bzw. ein b hinter einem c vor. Also ist $v = a^r$ oder $v = b^r$ oder $v = c^r$ und entsprechend $x = c^s$ oder $x = b^s$ oder $x = a^s$, wobei $r + s \geq 1$. Da v in z stets vor x liegt, sind das insgesamt sechs Fälle. Im Fall $v = a^r$ und $x = c^s$ gilt $u = a^j$, $w = a^kb^nc^l$, $y = c^m$ und $z = uvwxy = a^ja^ra^kb^nc^lc^sc^m$ mit $j + r + k = n = l + s + m$. Damit erhält man $uv^0wx^0y = a^j\lambda a^kb^nc^l\lambda c^m = a^{n-r}b^nc^{n-s} \in L_0$, was wegen $r + s \geq 1$ der Definition von L_0 widerspricht. Analog führen die anderen fünf Fälle zum Widerspruch.
2. Die durch die Produktionen $S_1 ::= S_1c|A$ und $A ::= aAb|\lambda$ sowie $S_2 ::= aS_2|B$ und $B ::= bBc|\lambda$ definierten Grammatiken erzeugen die kontextfreien Sprachen $L_1 = \{a^mb^mc^n \mid m, n \in \mathbb{N}\}$ und $L_2 = \{a^mb^nc^n \mid m, n \in \mathbb{N}\}$, deren Durchschnitt gerade die nicht kontextfreie Sprache L_0 aus Punkt 1 ist. □

18 PASCALchen macht Funktionen berechenbar

Die bisher betrachteten Konzepte haben eine eingeschränkte Berechnungskapazität. Endliche Automaten erkennen reguläre Sprachen und kontextfreie Grammatiken erzeugen kontextfreie Sprachen, wobei es aber mit Hilfe des jeweiligen Pumping-Lemmas möglich ist, auch algorithmische erkenn- oder erzeugbare Sprachen zu finden, die nicht regulär und nicht kontexfrei sind. Diese Beobachtung legt einige Fragen nahe:

Mit welchen Berechnungskonzepten lassen sich auch Sprachen algorithmisch behandeln, die nicht kontextfrei sind? Was ist eigentlich überhaupt "berechenbar"? Wie sehen Probleme aus, die nicht "berechenbar" sind? Gibt es das überhaupt?

Ziel dieses Abschnitts ist, den Begriff der berechenbaren Funktion zu präzisieren. Die Betonung liegt dabei auf einem exakten Vorgehen, um einen Teil der gestellten Fragen möglichst zuverlässig zu beantworten und den Spielraum für Einwände einzuengen. Der übliche Weg, zu berechenbaren Funktionen zu kommen, ist die Einführung einer Sprache zum Schreiben von Algorithmen (Syntax) und die Bereitstellung eines Kalküls, mit dem Algorithmen ausgeführt werden können (Semantik). Es gibt eine breite Palette an Möglichkeiten, das zu konkretisieren; sie reicht von maschinenorientierten bis zu problemorientierten Sprachen. Hier wird eine für Informatikerinnen und Informatiker naheliegende Wahl getroffen: die imperative Programmiersprache PASCALchen.

PASCALchen ist eine magere Sprache, die vage an PASCAL erinnert. PASCALchen verzichtet nahezu ganz auf Benutzungskomfort. Die wesentlichen Unterschiede zu üblichen imperativen Programmiersprachen sind, dass der einzige erlaubte Datentyp die natürlichen Zahlen sind und – vorläufig – Rekursion verboten ist. Das wesentliche Konstrukt von PASCALchen ist die Iteration auf der Basis der *while*-Schleife; deshalb werden die Programme der Sprache *while*-Programme genannt.

18.1 Die Sprache PASCALchen

1. Die Programme sind aus folgenden *Zeichen* aufgebaut:
   (a) *Variablennamen* sind Wörter, die mit X beginnen, worauf eine natürliche Zahl folgt.
   (b) *Operatorensymbole*, die zum Ändern der Variablenwerte dienen, sind *succ* für die Nachfolgerfunktion, *pred* für die Vorgängerfunktion und 0 für die Nullkonstante.
   (c) Als *Relationssymbol* zum Vergleich von Variablenwerten steht $\neq$ für die Ungleichheitsabfrage zur Verfügung.
   (d) *Programmsymbole* sind := für die Zuweisung, ; für die Reihung sowie *begin*, *end*, *while*, *do*.
   (e) Als *Hilfssymbole* zum Anwenden der Operatoren gibt es runde Klammern (und).
2. Programme werden nach folgender *Syntax* gebildet:
   (a) Ein *while-Programm* ist eine Reihungsanweisung.

   (b) Eine *Reihungsanweisung* (kurz: *Reihung*) hat die Form

$$\textit{begin } S_1; S_2; \ldots; S_m \textit{ end},$$

   wobei $S_1, \ldots, S_m$ für $m \geq 0$ Anweisungen sind. Der Fall $m = 0$ ist so gemeint, dass die Reihungsanweisung dann einfach die Form *begin end* hat.
   (c) Eine *Wiederholungsanweisung* (kurz: *Wiederholung*) hat die Form

$$\textit{while } X_i \neq X_j \textit{ do } S,$$

   wobei S eine Anweisung ist und X_i, X_j zwei Variablen.
   (d) Eine *Zuweisungsanweisung* (kurz: *Zuweisung*) hat die Form

$$X_i := 0 \quad \text{oder} \quad X_i := \textit{succ}(X_j) \quad \text{oder} \quad X_i := \textit{pred}(X_j),$$

   wobei X_i, X_j zwei Variablen sind.
   (e) Eine *Anweisung* kann eine Reihung, eine Wiederholung oder eine Zuweisung sein. Mit den folgenden Bemerkungen werden dann auch noch Makroanweisungen als Anweisungen zugelassen.
3. Die Variablen müssen nicht deklariert werden, da sie alle vom Typ der natürlichen Zahlen $\mathbb{N}$ sind, für die es, korrespondierend zu den syntaktischen Größen, eine Konstante 0 gibt, die Nachfolger- und Vorgängerfunktionen $\textit{succ}, \textit{pred}\colon \mathbb{N} \to \mathbb{N}$ und einen Ungleichheitstest $\neq\colon \mathbb{N}^2 \to \{\mathsf{T}, \mathsf{F}\}$. Dabei ist *pred* definiert durch $\textit{pred}(0) = 0$ und $\textit{pred}(n+1) = n$ für alle $n \in \mathbb{N}$. Die anderen Funktionen arbeiten in bekannter Weise.

Bemerkungen

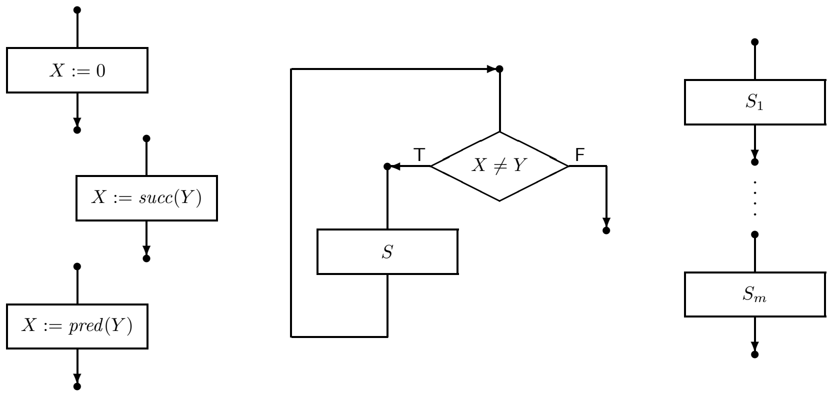
1. Dass PASCALchen so knapp zugeschnitten ist, hat einen pragmatischen und einen theoretischen – fast philosophischen – Grund.
   Es muss noch definiert werden, was *while*-Programme leisten. Das ist für wenige Konstrukte oft einfacher als für ein breites Spektrum. Andererseits ist das Ziel, das Berechenbare aufzuspüren. Von Vorgaben jedoch lässt sich nicht beweisen, dass sie Berechenbares darstellen, sondern es ist allenfalls plausibel und muss geglaubt werden. Je weniger vorausgesetzt wird, desto besser sind die Chancen, dass die Berechenbarkeit davon einsichtig ist und den Erfahrungen entspricht. Aus diesem Grunde sind insbesondere auch die natürlichen Zahlen als einziger Datentyp gewählt worden, weil Menschen seit Jahrtausenden mit ihnen rechnen. Anschaulich wird erlaubt, ein leeres Blatt Papier als 0 zu nehmen, für jedes *succ* einen Strich zu machen, für jedes *pred* einen Strich auszuradieren und von zwei Blättern festzustellen, ob sie gleich viele Striche enthalten oder nicht. Die Annahmen sind also prozesshaft und von einfacher Art. Ist eigentlich die Unterstellung der Berechenbarkeit bei Zuweisung, Wiederholung und Reihung genauso selbstverständlich?
2. So klein, wie es aussieht, ist PASCALchen gar nicht. Da jedes *while*-Programm selbst eine Anweisung ist, kann es in anderen Programmen weiterverwendet werden.

Gibt man *while*-Programmen Bezeichnungen und erlaubt, diese statt der Anweisungen weiterzuverwenden, lässt sich mit diesem Mechanismus von *Makroanweisungen* sogar sehr strukturiert programmieren. Viele vertraute Operationen, Programmkonstrukte und Abfragen erweisen sich als Makroanweisungen, d.h. sie lassen sich als *while*-Programme realisieren, und können deshalb wie in PASCAL und ähnlichen Sprachen verwendet werden. Dazu gehören die Zuweisungsformen $X := Y$, $X := n$ (Konstante), $X := Y + Z$, $X := Y - Z$, $X := Y * Z$, $X := Y \operatorname{div} Z$, $X := Y \bmod Z$, $X := 2^Y$, ...; die Wiederholungsanweisungen der Form *while B do S*, wobei B ein Boolescher Ausdruck ist, der aus Variablen, Konstanten, $=$, $\neq$, $<$, *and*, *or*, *not* aufgebaut ist, und die Kontrollstrukturen *if-then*, *if-then-else*, *repeat-until*.
Näheres dazu finden die Leserinnen und Leser im Abschnitt 2.2 des Buches (Kfoury, Moll, Arbib 1982). Dieses Buch ist auch sonst für das Thema Berechenbarkeit zu empfehlen.

3. In Bemerkung 1 wird begründet, dass das begrenzte Repertoire von PASCALchen die Gefahr mindert, mit den gewählten Beschreibungshilfsmitteln über das Berechenbare hinauszuschießen. In Bemerkung 2 wird erläutert, dass das einzige, was gegenüber Sprachen wie PASCAL wirklich fehlt, das Konzept der Rekursion ist. Damit ist auch die Gefahr klein, dass PASCALchen hinter den Möglichkeiten des Berechenbaren zurückbleibt. Denn eine Programmiersprache wie FORTRAN kennt auch keine Rekursion; trotzdem sind keine Klagen bekannt, dass damit bestimmte algorithmische Probleme nicht lösbar seien. Tatsächlich gilt, dass Rekursion auf Iteration zurückgeführt werden kann, wie in nahezu jedem Buch über Berechenbarkeit nachlesbar ist (z.B. Abschnitt 5.2 in (Kfoury, Moll, Arbib 1982)).
4. Um die Semantik von Programmen anzugeben, wird ausgenutzt, dass jede Anweisung durch ein Flussdiagramm wie in Abbildung 12 dargestellt werden kann. Dabei werden die drei Zuweisungen und der Test *Instruktionen* genannt. Die Definition ist rekursiv. Die mit $S, S_1, \ldots, S_m$ bezeichneten Kästen müssen solange durch eines der fünf Diagramme ersetzt werden, bis nur noch Instruktionen vorkommen. Jedes Flussdiagramm hat genau einen Eingang und einen Ausgang. Die fünf Diagramme sind genau nach den Syntaxregeln gebildet. Darüber hinaus lässt sich zeigen, dass jedes *while*-Programm genau ein Flussdiagramm besitzt.
5. Wie bereits in Punkt 2 geschehen, werden im folgenden manchmal die Variablen durch beliebige Großbuchstaben benannt, wenn die spezielle Form Xi nicht gebraucht wird.

18.2 Berechnungen von PASCALchen-Programmen

Mit Hilfe der Flussdiagramme soll nun geklärt werden, was *while*-Programme berechnen. Das Abarbeiten eines Programms mit bestimmten Anfangswerten für die Variablen entspricht einem Durchlauf durch das zugehörige Flussdiagramm. Wird dabei eine Zuweisung überquert, ändert sich der Wert der betroffenen Variablen entsprechend. Wird ein Test erreicht, bleiben alle Werte unverändert. Aber anhand der aktuellen Werte entscheidet sich, wie weitergegangen wird (in Richtung T oder F). Die jeweiligen Werte der Variablen



(a) Zuweisungen (b) Wiederholung (c) Reihung

Abbildung 12: Flussdiagramme für PASCALchen-Anweisungen

während der Auswertung werden im Zustandsvektor festgehalten. Um dessen Definition zu vereinfachen, wird folgende Konvention getroffen.

Vereinbarung

Eine Anweisung (und damit auch ein *while*-Programm) wird *k-variabel* genannt, wenn höchstens die Variablen $X1, \ldots, Xk$ darin vorkommen.

Nutzt man die Reihenfolge der Variablen aus, können ihre Werte als eine Folge von k Zahlen $x_1, \ldots, x_k$ notiert werden, wobei x_i der Wert von Xi ist.

Definition (Berechnungszustand)
Ein *Berechnungszustand* eines k-variablen *while*-Programms ist ein k-dimensionaler Vektor über den natürlichen Zahlen

$$a = (x_1, \ldots, x_k) \in \mathbb{N}^k,$$

der auch *Zustand* oder *Zustandsvektor* genannt wird.

Gezielte Veränderungen der Berechnungszustände während eines Durchlaufs durch das Flussdiagramm ergeben eine Berechnung eines Programms.

Definition (Berechnung)
Eine *Berechnung* durch ein k-variables *while*-Programm ist eine (möglicherweise unendliche) Sequenz der Form

$$a_0 A_1 a_1 A_2 a_2 \cdots a_{i-1} A_i a_i A_{i+1} a_{i+1} \cdots,$$

wobei die a_i Berechnungszustände und die A_i Instruktionen mit folgenden Eigenschaften sind:

(a) Es gibt einen Weg durch das zugehörige Flussdiagramm, der beim Eingang beginnt und genau die Instruktionen $A_1, A_2, \dots, A_i, \dots$ in dieser Reihenfolge durchläuft.
(b) a_0 ist frei wählbar.
(c) Für alle $i \geq 1$ ergibt sich a_i aus a_{i-1} und A_i nach folgenden Regeln:
  – Ist A_i ein Test, dann ist $a_i = a_{i-1}$.
  – Ist A_i eine Zuweisung der Form $Xu := g(Xv)$, wobei $g(Xv)$ für $succ(Xv)$ oder $pred(Xv)$ oder 0 steht und $u, v \in \{1, \dots k\}$; ist ferner $a_{i-1} = (x_1, \dots, x_k)$, dann gilt:
$$a_i = (x_1, \dots, x_{u-1}, g(x_v), x_{u+1}, \dots, x_k).$$
Ist A_i die letzte Instruktion der Berechnung, so erreicht man hinter A_i den Ausgang. Ansonsten ist A_{i+1} die nächste Instruktion hinter A_i.
(d) Ist A_i ein Test der Form $Xu \neq Xv$, $a_{i-1} = (x_1, \dots, x_k)$ sowie $x_u \neq x_v$, dann ist A_{i+1} die erste Instruktion in der Anweisung der zugehörigen Wiederholung. Ist jedoch $x_u = x_v$, muss man dem F-Zweig folgen. Dann erreicht man den Ausgang, falls A_i die letzte Instruktion der Berechnung ist. Ansonsten ist A_{i+1} die nächste Instruktion nach dieser Wiederholung.
(e) Ist die Sequenz endlich, endet sie mit einem Berechnungszustand und hat deshalb die Form:
$$a_0 A_1 a_1 \cdots a_{n-1} A_n a_n \qquad (n \geq 0).$$
Dabei ist A_n die letzte Instruktion vor dem Ausgang des Flußdiagramms. Ist A_n ein Test $Xu \neq Xv$, muss ferner $x_u = x_v$ gelten für $a_{n-1} = a_n = (x_1, \dots, x_k)$.
(f) Im Falle von (e) wird n die *Länge* der Berechnung genannt.

Unmittelbar aus der Konstruktion ergibt sich das folgende Ergebnis.

Beobachtung 16
Zu jedem k-variablen *while*-Programm und jedem Berechnungszustand a_0 gibt es genau eine Berechnung mit a_0 als Anfang.

Bemerkungen

1. Für den Umgang mit Berechnungen sind einige Sprechweisen hilfreich:
   (a) Fängt eine Berechnung mit a_0 an, wird a_0 *Eingabe* genannt.
   (b) Für eine endliche Berechnung $a_0 A_1 a_1 \cdots A_n a_n$ sagt man, dass sie (in n Schritten) *terminiert*; a_n wird dann *Ausgabe* genannt.
   (c) Entsprechend *terminiert* eine unendliche Berechnung *nicht*; ihre Ausgabe ist *undefiniert*.
2. Die hier beschriebene Auswertung von *while*-Programmen ist ein typisches Beispiel für eine operationelle Semantik, wobei anhand des Programmtextes eine Eingabe schrittweise zur Ausgabe umgeformt wird. Da die Semantik dadurch entsteht, dass Berechnungszustände umgeformt werden, wird sie auch als Zustandstransformationssemantik bezeichnet.

18.3 Semantik von PASCALchen-Programmen

Die obige Beobachtung legt nahe, die Semantik eines Programms als partielle Funktion auf den natürlichen Zahlen zu definieren, denn Berechnungen sind pro Eingabe eindeutig, liefern jedoch nur bei Termination Ergebnisse. Aus technischen Gründen wird dabei von den verfügbaren k Variablenwerten in modifizierter Form Gebrauch gemacht. Während als Funktionswert nur der Wert der Variablen $X1$ betrachtet wird, wird die Funktion auf j Argumente angewendet. Ist $j < k$, bekommen die ersten j Variablen die Argumente als Eingangswerte, während die restlichen durch Nullen aufgefüllt werden. Ist $j \geq k$, fallen die eventuell überschüssigen Argumente bei der Berechnung weg. Durch diesen "Trick" wird die Zahl der Argumente unabhängig von der Zahl der Variablen.

Definition (Semantikfunktion)
Sei P ein k-variables *while*-Programm und $j \in \mathbb{N}$. Dann ist die j-stellige *Semantikfunktion* von P
$$SEM_P \colon \mathbb{N}^j \to \mathbb{N}$$
für die Argumente $(x_1, \dots, x_j) \in \mathbb{N}^j$ nach folgenden Regeln definiert:
(a) Aus den Argumenten wird eine Eingabe $a \in \mathbb{N}^k$ hergestellt, wobei $a = (x_1, \dots, x_k)$ ist für $j \geq k$ und $a = (x_1, \dots, x_j, 0, \dots, 0)$ mit $k - j$ Nullen für $j < k$.
(b) P wird mit Eingabe a berechnet.
(c) Terminiert die Berechnung mit der Ausgabe $(y_1, \dots, y_k)$, so ist $SEM_P(x_1, \dots, x_j) = y_1$.
(d) Terminiert sie nicht, ist $SEM_P(x_1, \dots, x_j)$ undefiniert.

Bemerkungen

1. Beachte, dass die Semantikfunktion wegen (d) im allgemeinen nicht immer definiert ist. Sie ist genau dann eine totale Funktion, wenn jede Berechnung terminiert.
2. Jedes Programm kann wegen der Wahlfreiheit von j unendlich viele Funktionen berechnen, die sich allerdings nur wenig voneinander unterscheiden.
3. Soll das j betont werden, wird $SEM_P^{(j)}$ statt SEM_P geschrieben.
4. Ist $SEM_P(x_1, \dots, x_j)$ undefiniert, wird im folgenden manchmal die Bezeichnung $SEM_P(x_1, \dots, x_j) = \perp$ dafür verwendet.

18.4 Berechenbarkeit

Nach diesen Vorbereitungen ist es nun möglich, den zentralen Begriff der Berechenbarkeitstheorie einzuführen. Berechenbare Funktionen präzisieren durch Computer lösbare Aufgaben.

Definition (Berechenbare Funktion)
Eine partielle Funktion $f \colon \mathbb{N}^j \to \mathbb{N}$ heißt *berechenbar*, wenn ein *while*-Programm existiert

mit

$$f = SEM_P^{(j)}.$$

Mit Hilfe dieser exakten Version von Berechenbarkeit lässt sich nun auch die sogenannte Churchsche These für die Zwecke dieser Lehrveranstaltung formulieren. Der folgende Abschnitt zeigt, dass sie sehr bequem ist, auch wenn man ihr nicht unbedingt glaubt. Sie besagt, dass man mit PASCALchen alles berechnen kann, was überhaupt möglich ist. Es muss allerdings darauf hingewiesen werden, dass die Berechenbarkeitstheorie und mathematische Logik bisher nur Belege für ihre Richtigkeit gefunden haben und dass sie auch den Erfahrungen der meisten Informatikerinnen und Informatiker entspricht.

Churchsche These
Jede partielle Funktion $f\colon \mathbb{N}^j \to \mathbb{N}$, die durch irgendeinen Mechanismus oder auf Grund irgendeiner Überlegung algorithmisch berechnet werden kann, ist bereits berechenbar (durch ein *while*-Programm).

19 Programme sind aufzählbar – doch ihr (Ver-)Halten macht Kummer

Zu den technologischen Durchbrüchen der Computertechnik in den 40er Jahren zählt das von Neumann entdeckte Prinzip der Programmspeicherung, durch das Programme jederzeit abrufbar und wie Daten bearbeitbar werden. Heutzutage ist allen Informatikerinnen und Informatikern angesichts von Editoren, Compilern u.ä. vertraut, dass Programme Programme verarbeiten können. Theoretisch wurde die Tatsache, dass Algorithmen Daten von Algorithmen sein können, noch einige Jahre früher von Gödel erkannt und hat in Logik und Berechenbarkeitstheorie zu fundamentalen Erkenntnissen geführt. Um einiges davon vorstellen zu können, ist noch eine Schwierigkeit zu überwinden: Der Algorithmusbegriff des vorigen Kapitels arbeitet auf natürlichen Zahlen; damit *while*-Programme Daten werden und umgekehrt, müssen natürliche Zahlen als Programme und Programme als natürliche Zahlen aufgefasst werden können. Zu diesem Zweck erhält jedes *while*-Programm einen Index. Außerdem wird ein algorithmisches Verfahren angegeben, wie aus einer natürlichen Zahl ein Programm entsteht, dessen Index sie ist. Auf diese Weise werden *while*-Programme "effektiv" aufgezählt.

19.1 Bestimmung des Indexes eines Programms

1. Nach Punkt 18.1.1 in Verbindung mit der Konvention in Abschnitt 18.2 besteht der Zeichensatz A von PASCALchen aus 22 Zeichen.
2. Für jedes Zeichen $a \in A$ wird nun eine eindeutige 6-Bit-Darstellung $code(a) = b_1 \cdots b_6$ mit $b_1 = 1$ und $b_2, \ldots, b_6 \in \{0,1\}$ ausgewählt. Es wird also eine injektive Abbildung $code\colon A \to \{0,1\}^*$ fixiert. Eine solche Wahl ist möglich, weil es 32 verschiedene 6-Bit-Muster mit 1 am Anfang gibt.
3. Die Abbildung *code* lässt sich auf A^* fortsetzen, indem jedes Zeichen eines Wortes von links nach rechts durch seine 6-Bit-Darstellung ersetzt wird. Das ergibt eine Abbildung $code^*\colon A^* \to \{0,1\}^*$, die definiert ist durch
   (i) $code^*(\lambda) = \lambda$ und
   (ii) $code^*(av) = code(a)code^*(v)$ für $a \in A$, $v \in A^*$.
4. Jedes *while*-Programm ist ein Wort über A und besitzt deshalb ein Bitmuster. Jedes Bitmuster lässt sich als Binärdarstellung einer natürlichen Zahl auffassen. Das erlaubt folgende Definition.
5. Der *Index* eines *while*-Programms P ist die natürliche Zahl, deren Binärdarstellung $code^*(P)$ ist.
6. Diese Verwandlung syntaktischer Objekte in Zahlen wird *Arithmetisierung* oder *Gödelnumerierung* genannt. Sie macht es prinzipiell möglich, mit Programmen zu rechnen.

Es ist äußerst wichtig, dass verschiedene Programme verschiedene Indizes erhalten. Das jedoch folgt unmittelbar aus folgendem Lemma.

Lemma 17 (Injektivität der Gödelnumerierung)
Die Gödelnumerierung $code^*$ ist injektiv.

Beweis.
Betrachte die Abbildung $decode\colon \{0,1\}^* \to A^*$, die definiert ist für alle $B \in \{0,1\}^*$ und $b_1, \ldots, b_6 \in \{0,1\}$ durch

(i) $decode(B) = \lambda$ für alle B kürzer als 6,

(ii) $decode(b_1 \cdots b_6 B) = \begin{cases} a\, decode(B) & \text{wenn } code(a) = b_1 \cdots b_6 \\ \lambda & \text{sonst.} \end{cases}$

Dann gilt $decode(code^*(w)) = w$ für alle $w \in A^*$, wie sich durch vollständige Induktion beweisen lässt.

IA: $decode(code^*(\lambda)) = decode(\lambda) = \lambda$.

IV: Die Aussage gelte für $v \in A^*$.

IS: $decode(code^*(av)) = decode(code(a)code^*(v)) = a\, decode(code^*(v)) = av$.

Daraus ergibt sich die behauptete Injektivität:
$code^*(u) = code^*(v)$ impliziert $decode(code^*(u)) = decode(code^*(v))$, also $u = v$. $\square$

Umgekehrt kann jeder Zahl algorithmisch ein Programm zugeordnet werden, dessen Index sie ist.

19.2 Bestimmung des Programms eines Indexes

Betrachte zu jeder natürlichen Zahl n ihre Binärdarstellung $B(n)$, und wende darauf die Abbildung *decode* aus dem vorangegangenen Beweis an. Ist $decode(B(n))$ ein *while*-Programm, so wird es als von n aufgezählt betrachtet:

$$AUFZÄHLUNG(n) = decode(B(n)).$$

Andernfalls wird n ein Programm zugeordnet, das niemals hält:

$$\begin{aligned} AUFZÄHLUNG(n) = \ & begin \\ & \quad X1 := 0;\ X2 := 1; \\ & \quad while\ X1 \neq X2\ do\ X1 := X1 \\ & end. \end{aligned}$$

Damit ist insgesamt eine Abbildung $AUFZÄHLUNG$ definiert, die jeder Zahl ein Programm zuordnet.

Bemerkung
Durch $AUFZÄHLUNG$ wird insbesondere der Index n eines Programms P auf P abgebildet.

Beweis.
Nach Definition des Indexes n eines *while*-Programms P gilt: $B(n) = code^*(P)$. Daraus folgt mit der Injektivität der Gödelnumerierung:

$$decode(B(n)) = decode(code^*(P)) = P.$$

Nach obiger Definition von $AUFZÄHLUNG$ ergibt sich wie behauptet:
$AUFZÄHLUNG(n) = decode(B(n)) = P$. $\square$

Bemerkungen

1. Im folgenden wird eine natürliche Zahl n *Index* des *while*-Programms P genannt, wenn $AUFZÄHLUNG(n) = P$. Wegen der obigen Beobachtung umfasst diese Definition die bisherigen Indizes. Statt $AUFZÄHLUNG(n)$ wird oft kurz P_n geschrieben. P_n bezeichnet also das Programm mit dem Index n.
   Jede natürliche Zahl ist nun Index, so dass *while*-Programme mühelos Indizes manipulieren können. Von $AUFZÄHLUNG$ erfährt man dann, wie sich das auf die indizierten Programme auswirkt. Beachte, dass dadurch alle Programme erfasst werden, weil jedes Programm einen Index hat.
2. Da jedes Programm P für jede Argumentzahl j eine berechenbare Funktion $SEM_P^{(j)}$ bestimmt, kann jede Zahl auch als Index einer berechenbaren Funktion aufgefasst werden:
   $$SEM_n^{(j)} := SEM_{P_n}^{(j)} \text{ für alle } n \in \mathbb{N}.$$
   Statt $SEM_n^{(1)}$ wird kürzer SEM_n geschrieben, denn dieser Fall kommt am häufigsten vor.
   Beim "Aufzählen" der natürlichen Zahlen werden also gleichzeitig alle *while*-Programme und damit insbesondere alle berechenbaren Funktionen mit einem Argument "mitaufgezählt".
   $$\begin{array}{ccccccccl} 0 & 1 & 2 & \ldots & n & \ldots & & & \mathbb{N} \\ \downarrow & \downarrow & \downarrow & & \downarrow & & & & \downarrow AUFZÄHLUNG \\ P_0 & P_1 & P_2 & \ldots & P_n & \ldots & & & \textit{while}\text{-Programme} \\ \downarrow & \downarrow & \downarrow & & \downarrow & & & & \downarrow SEM \\ SEM_0 & SEM_1 & SEM_2 & \ldots & SEM_n & \ldots & & & \text{berechenbare Funktionen} \end{array}$$
   Für jede andere Argumentzahl kann die Aufzählung entsprechend durchgeführt werden; das wird aber im folgenden nicht gebraucht.
3. Beachte, dass die Indizes nicht für das praktische Rechnen mit Programmen geeignet sind, weil sie sehr groß ausfallen. Ein Programm aus nur 10 Zeichen beispielsweise hat bereits einen kleinsten Index zwischen 2^{59} und 2^{60}.

19.3 Effektive Aufzählbarkeit und nicht berechenbare Funktionen

1. Man nennt eine Menge M von Objekten *abzählbar* oder *aufzählbar*, wenn es eine surjektive Abbildung $num\colon \mathbb{N} \to M$ gibt. M heißt *effektiv aufzählbar*, wenn diese Numerierung durch einen Algorithmus vorgenommen wird.
2. Da die Binärdarstellung einer Zahl berechnet werden kann, da die Werte von *decode* algorithmisch bestimmt sind, da es schließlich einen Algorithmus gibt, der entscheidet, ob ein Text über A ein *while*-Programm ist oder nicht, erweist sich $AUFZÄHLUNG$ als effektiv. Graphisch sei das ausgedrückt durch die Darstellung in Abbildung 13.

Abbildung 13: Effektivität von $AUFZÄHLUNG$

3. Dass die effektive Aufzählung etwas Besonderes ist, zeigt folgende Überlegung: Abzählbar ist insbesondere jede Teilmenge der natürlichen Zahlen, von denen es überabzählbar viele gibt. Effektiv aufzählbar sind jedoch nur abzählbar viele Teilmengen, weil es nach Abschnitt 19.2 nur abzählbar viele Algorithmen gibt. Die dabei verwendete Überabzählbarkeit der Teilmengen von $\mathbb{N}$ ergibt sich analog zur folgenden Überlegung.
   Die Konstruktionen in Abschnitt 19.2 zeigen, dass es nur abzählbar viele berechenbare Funktionen gibt. Mit einem bekannten Argument (dem Diagonalisierungsverfahren von Cantor, mit dem dieser gezeigt hat, dass es überabzählbar viele reelle Zahlen gibt) stellt sich heraus, dass die (partiellen) Funktionen auf $\mathbb{N}$ nicht abzählbar sind. Es muss also Funktionen geben, die nicht berechenbar sind.

Theorem 18 (Existenz nicht-berechenbarer Funktionen)
Es gibt eine Funktion $f\colon \mathbb{N} \to \mathbb{N}$, die nicht berechenbar ist.

Beweis (durch Widerspruch)**.**
Angenommen, jede Funktion $f\colon \mathbb{N} \to \mathbb{N}$ wäre berechenbar. Dann gäbe es zu f einen Index i mit $f = SEM_i$. Die (totalen) Funktionen lassen sich dann folgendermaßen aufzählen: f_0 ist die erste Funktion in der Aufzählung $SEM_0, SEM_1, SEM_2, \ldots$. Sind $f_0, \ldots, f_k$ bereits bestimmt, dann ist f_{k+1} die nächste Funktion hinter f_k in $SEM_0, SEM_1, SEM_2, \ldots$. Betrachte dann die Funktion $plus\colon \mathbb{N} \to \mathbb{N}$, die definiert ist durch

$$plus(n) = f_n(n) + 1.$$

Fasst man die Argumente als Spaltennummern und die Indizes der Funktionen als Zeilennummern auf und trägt in die Felder (m, n) immer gerade $f_m(n)$ ein, dann entsteht *plus*

dadurch, dass jede Zahl in der Diagonalen dieser unendlichen Matrix um 1 hochgezählt wird.

Nach Annahme existiert ein Index l mit $plus = f_l$. Das aber führt zu einem Widerspruch, denn *plus* erweist sich als verschieden zu f_l:

$$plus(l) = f_l(l) + 1 \neq f_l(l).$$

Da die Annahme falsch ist, muss die Behauptung richtig sein. □

19.4 Unlösbarkeit des Halteproblems

Die allgemeine Überlegung zur Nicht-Berechenbarkeit hat den Nachteil, dass sich eine ziemlich uninteressante Funktion als nicht-berechenbar erweist. Es ist nicht übermäßig wahrscheinlich, dass irgendjemand auf die Idee gekommen wäre, gerade diese Funktion zu berechnen. Anders verhält es sich mit dem folgenden Problem, bei dem versucht wird, eine äußerst wichtige Eigenschaft von Programmen zu ermitteln, nämlich ob bestimmte Berechnungen terminieren oder nicht. In dem hier betrachteten Fall des "Halteproblems" wird jedes Programm auf den eigenen Index angewendet, weshalb es in der Literatur auch häufig "Selbstanwendungsproblem" genannt wird. Es stellt sich heraus, dass es unmöglich ist, zu berechnen, ob ein Programm bei Eingabe seines Indexes hält oder nicht.

Theorem 19 (Unlösbarkeit des Halteproblems)
Die (totale) Funktion $HALTEPROBLEM : \mathbb{N} \to \mathbb{N}$, die definiert ist für alle $i \in \mathbb{N}$ durch

$$HALTEPROBLEM(i) = \begin{cases} 1 & \text{wenn } SEM_i(i) \text{ definiert ist} \\ 0 & \text{sonst} \end{cases}$$

ist nicht berechenbar.

Beweis (durch Widerspruch)**.**
Angenommen, die Funktion $HALTEPROBLEM$ wäre durch das *while*-Programm $HALT$ berechenbar. Dann ist auch die partielle Funktion $konfus : \mathbb{N} \to \mathbb{N}$ mit

$$konfus(i) = \begin{cases} 1 & \text{für } HALTEPROBLEM(i) = 0 \\ \perp & \text{sonst} \end{cases}$$

berechenbar durch das Programm:

$$\begin{array}{l} \mathit{begin} \\ \quad HALT; \\ \quad \mathit{while}\ X1 \neq 0\ \mathit{do}\ X1 := X1; \\ \quad X1 := 1 \\ \mathit{end} \end{array}$$

Es existiert also ein Index j mit $\mathit{konfus} = \mathit{SEM}_j$. Betrachte nun die Anwendung von SEM_j auf j.

1. Fall: $\mathit{SEM}_j(j)$ ist definiert.
Dann ist $\mathit{HALTEPROBLEM}(j) = 1$ und damit $\mathit{konfus}(j) = \mathit{SEM}_j(j)$ undefiniert.

2. Fall: $\mathit{SEM}_j(j)$ ist undefiniert.
Dann ist $\mathit{HALTEPROBLEM}(j) = 0$ und damit $\mathit{konfus}(j) = \mathit{SEM}_j(j) = 1$.

Mehr Fälle gibt es nicht, und $\mathit{SEM}_j(j)$ kann nicht gleichzeitig definiert und undefiniert sein. Also muss schon die Annahme falsch und damit die Behauptung des Theorems richtig sein. □

Wem es immer noch "exotisch" vorkommt, dass man Programme auf sich selbst bzw. ihre Indizes anwendet, den mag die Unlösbarkeit des allgemeinen Halteproblems mehr überzeugen, bei dem es darum geht, ob ein Programm für irgendeine Eingabe hält oder nicht. Das ist auch ein erstes Beispiel für das hilfreiche Reduktionsprinzip, bei dem die Nicht-Berechenbarkeit eines Problems dadurch gezeigt wird, dass das Problem auf eines zurückgeführt wird, dessen Nicht-Berechenbarkeit bereits bekannt ist.

Korollar 20 (Unlösbarkeit des allgemeinen Halteproblems)
Die Funktion $\mathit{HALTEPROBLEM2}\colon \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ *mit*

$$\mathit{HALTEPROBLEM2}(i,j) = \begin{cases} 1 & \text{wenn } \mathit{SEM}_i(j) \text{ definiert ist} \\ 0 & \text{sonst} \end{cases}$$

ist nicht berechenbar.

Beweis (durch Widerspruch).
Angenommen, $\mathit{HALTEPROBLEM2}$ wäre durch das *while*-Programm $\mathit{HALT2}$ berechenbar. Dann wäre auch $\mathit{HALTEPROBLEM}$ berechenbar durch das Programm

```
begin
  X2 := X1;
  HALT2
end
```

im Widerspruch zum obigen Theorem. □

Bemerkung
Für die Nicht-Berechenbarkeit von Problemen, die auch als Unlösbarkeit angesprochen wird, kann man häufig auch die Bezeichnung *Nichtentscheidbarkeit* finden, wenn das Problem eine ja/nein-Frage ist, d.h. wenn die zugehörige Funktion für jede Eingabe einen von zwei Werten (in der Regel 0 und 1) liefert.

20 Ein Programm rechnet für alle

Die Nicht-Berechenbarkeit des Halteproblems wird verursacht durch den Wunsch, auch erfahren zu wollen, wenn Programme nicht halten. Verzichtet man darauf und begnügt sich damit, die Werte von Programmberechnungen zu bekommen, soweit sie existieren, so erweist sich diese abgeschwächte Aufgabe als berechenbar.

20.1 PASCALchen-Interpreter als universelle Funktion

Um diese Behauptung einzusehen, muss ein Interpreter konstruiert werden, der bei Eingabe eines Programms P und der Argumente $(x_1,\ldots,x_j)$ der von P berechneten Funktion SEM_P den Wert $\mathit{SEM}_P(x_1,\ldots,x_j)$ liefert, vorausgesetzt P terminiert für diese Eingabe. Graphisch ist dies in Abbildung 14 dargestellt.

Abbildung 14: Ein Interpreter für *while*-Programme

Ein solcher Algorithmus $\mathit{INTERPRETER}_0$ ergibt sich gerade aus den im 6. Kapitel eingeführten Konzepten und Konstruktionen:

(1) Wandle das k-variable Programm P in sein Flussdiagramm um.
(2) Wandle $(x_1,\ldots,x_j)$ in eine Eingabe a um.
(3) Berechne P für Eingabe a.
(4) Gib bei Termination den Wert von $X1$ aus.

Zwei Klippen sind noch zu umschiffen, bevor dieser Vorgang sich als berechenbar herausstellen kann. Der Interpreter verarbeitet Programme, *while*-Programme und berechenbare Funktionen nur natürliche Zahlen. Diese Diskrepanz lässt sich beseitigen, wenn der Interpreter wie in Abbildung 15 mit der Aufzählung des 7. Kapitels gekoppelt wird, da die Aufzählung ja gerade dazu dient, Zahlen stellvertretend für Programme als Eingabe zu akzeptieren.

Diese neue Interpreterfunktion hat jetzt nur noch natürlichzahlige Argumente, und zwar genau ein Argument mehr als die Funktionen, die berechnet werden. Es gibt nach den bisherigen Überlegungen einen Algorithmus, der die Werte dieser Funktion ermittelt; aber gibt es auch ein *while*-Programm, das das leistet? Die bejahende Antwort ergibt sich aus der CHURCHSCHEN THESE. Das ist ein interessantes Beispiel für ihren vorteilhaften Gebrauch: Um Berechenbarkeit sicherzustellen, muss nicht immer ein *while*-Programm geschrieben werden; irgendein Algorithmus tut es auch. Beachte, dass es gerade auch für

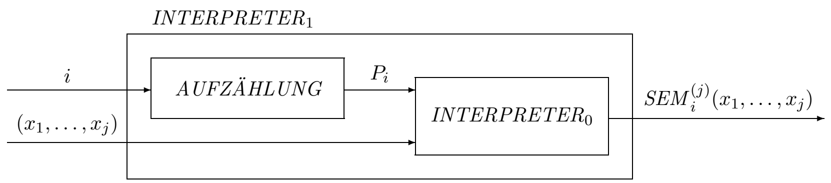


Abbildung 15: Eine Interpreter-Funktion

die Konstruktionen des 6. und 7. Kapitels reichlich mühsam wäre, simulierende *while*-Programme zu entwickeln.

Zusammengefasst hat sich folgende Erkenntnis ergeben:

Theorem 21 (universelle Funktion)
Für alle $j \in \mathbb{N}$ *ist die partielle Funktion* $interpret\colon \mathbb{N}^{j+1} \to \mathbb{N}$, *die definiert ist durch*

$$interpret(i, x_1, \ldots, x_j) = SEM_i^{(j)}(x_1, \ldots, x_j),$$

berechenbar.

Bemerkung
Da *interpret eine* Funktion ist, die *alle* berechenbaren Funktionen mit j Argumenten ausrechnet, wird sie *universelle Funktion* genannt. So verblüffend dieser Sachverhalt klingen mag, eigentlich passiert etwas Simples. Die Indizes der berechenbaren Funktionen werden zu einem zusätzlichen Argument. Und die Berechnung eines *while*-Programms ist ein algorithmischer Vorgang.

Zum Abschluss dieses Kapitels wird an fünf Beispielen die Grenzlinie zwischen Berechenbarem und Unberechenbarem kommentiert, die universelle Funktionen vom Halteproblem trennt. Vier der Beispiele nutzen dabei die Existenz universeller Funktionen in spezieller Form:

Es gibt ein k-variables *while*-Programm *INTERPRETER* mit

$$SEM_{INTERPRETER}(i, x) = SEM_i(x) \text{ für alle } i, x \in \mathbb{N}.$$

20.2 Beispiele

1. Abhängig von den Berechnungen des Interpreters lassen sich auch andere Aufgaben erledigen. Ein einfaches Beispiel dafür ist die partielle Funktion $halt_1\colon \mathbb{N}^2 \to \mathbb{N}$ mit

$$halt_1(x, y) = \begin{cases} y & \text{wenn } SEM_x(x) \text{ definiert ist} \\ \perp & \text{sonst.} \end{cases}$$

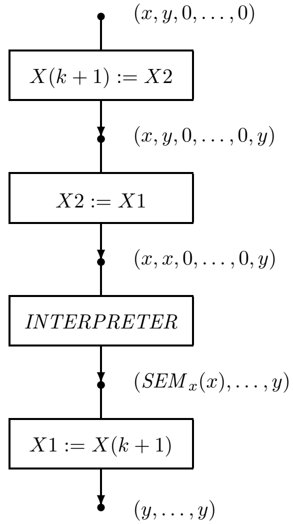


Abbildung 16: Das Programm $HALT_1$

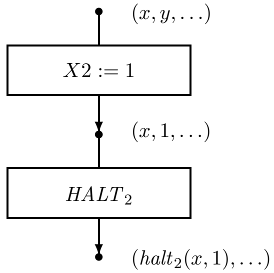


Abbildung 17: Reduktion des Halteproblems auf $halt_2$

Die Funktion $halt_1$ erweist sich als berechenbar durch das in Abbildung 16 dargestellte $(k+1)$-variable *while*-Programm $HALT_1$. Rechts sind die wichtigsten Änderungen der Zustandsvektoren verzeichnet, wobei der vorletzte Zustand nur entsteht, wenn der Interpreter hält. Die Berechnungen zeigen, dass die partielle Funktion $halt_1$ berechnet wird.

2. Die Grenze zur Unberechenbarkeit wird wieder gerade überschritten, wenn über $halt_1$ hinaus auch Werte produziert werden sollen, wo der Interpreter nicht hält. Deshalb ist die Funktion $halt_2\colon \mathbb{N}^2 \to \mathbb{N}$ nicht berechenbar, falls

$$halt_2(x, y) = \begin{cases} y & \text{wenn } SEM_x(x) \text{ definiert ist} \\ 0 & \text{sonst.} \end{cases}$$

Angenommen, $halt_2$ wird durch das *while*-Programm $HALT_2$ berechnet. Betrachte dann das in Abbildung 17 dargestellte Programm. Die rechts aufgezeichneten Berechnungen zeigen (für $y = 0$), dass dieses Programm P die einstellige Funktion $SEM_P\colon \mathbb{N} \to \mathbb{N}$ berechnet mit

$$SEM_P(x) = halt_2(x, 1) = \begin{cases} 1 & \text{wenn } SEM_x(x) \text{ definiert ist} \\ 0 & \text{sonst.} \end{cases}$$

Das ist das Halteproblem, das gar nicht lösbar ist; also muss die Annahme falsch sein.

Am Unterschied von $halt_1$ und $halt_2$ wird noch einmal deutlich, wodurch Unberechenbarkeit verursacht wird. Solange der Interpreter läuft, kann nicht gesagt werden, ob die Berechnung nach weiteren Schritten abbricht oder nie zum Stehen kommt. Denn die Schrittzahl von Berechnungen kann jede endliche Schranke überschreiten. Ob der Wert 0 sein muss, lässt sich beim Berechnen nicht erfahren.

3. Die Verhältnisse können aber auch wesentlich undurchsichtiger sein. Die (partielle) Funktion $halt_3 \colon \mathbb{N}^2 \to \mathbb{N}$ mit

$$halt_3(x,y) = \begin{cases} \mu(y) & \text{wenn } SEM_x(x) \text{ definiert ist} \\ \nu(y) & \text{sonst,} \end{cases}$$

wobei $\mu, \nu \colon \mathbb{N} \to \mathbb{N}$ beide berechenbare Funktionen sind, hat große Ähnlichkeit mit $halt_2$. Und doch ist sie unter bestimmten Umständen berechenbar, wenn nämlich $\nu \leq \mu$ ist, d.h. wann immer $\nu(y)$ definiert ist, so ist auch $\mu(y)$ definiert, und in diesem Fall gilt: $\mu(y) = \nu(y)$.
Ein Beispiel für solche Funktionen wäre etwa gegeben durch

$$\mu(y) = \begin{cases} y^2 & \text{für } y < 200 \\ \perp & \text{sonst} \end{cases} \qquad \text{und} \qquad \nu(y) = \begin{cases} y^2 & \text{für } y < 100 \\ \perp & \text{sonst.} \end{cases}$$

Die Berechenbarkeit von $halt_3$ lässt sich folgendermaßen einsehen, wobei $\nu \leq \mu$ vorausgesetzt wird.
   (a) Berechne gleichzeitig $\nu(y)$ und $SEM_x(x)$.
   (b) Terminiert zuerst $SEM_x(x)$, tritt der erste Fall in Kraft. Beende deshalb die Berechnung von $\nu(y)$ und beginne die von $\mu(y)$.
   (c) Terminiert zuerst die Berechnung von $\nu(y)$, dann ist nach Voraussetzung auch $\mu(y)$ definiert, und es gilt: $\mu(y) = \nu(y)$. Der Wert von $halt_3(x,y)$ ist damit gefunden, unabhängig davon, ob $SEM_x(x)$ terminiert.
   (d) Terminieren beide nicht, so ist auch $halt_3(x,y)$ undefiniert.

   Deshalb beschreiben die Punkte (a)-(c) einen Algorithmus, der $halt_3$ ausrechnet. Nach der Churchschen These ist $halt_3$ also berechenbar.

4. Eine sehr praktische Möglichkeit, die Schwierigkeiten mit dem Halteproblem zu umgehen, besteht darin, nach bestimmter “Zeit” die Berechnung gezielt abzubrechen. Im Beispiel wird das erreicht, indem die Rechenschritte gezählt werden und bei Überschreiten einer vorgegebenen Schranke aufgehört wird.
Die Funktion $halt_4 \colon \mathbb{N}^3 \to \mathbb{N}$ mit

$$halt_4(x,y,z) = \begin{cases} 1 & \text{wenn } P_x \text{ mit Eingabe } y \text{ nach spätestens } z \text{ Schritten hält} \\ 0 & \text{sonst} \end{cases}$$

ist berechenbar.
Denn ein Algorithmus für $halt_4$ kann wie folgt arbeiten:
   (a) Berechne P_x für Eingabe y.
   (b) Zähle die Schritte (Länge des Berechnungsweges im Flussdiagramm).
   (c) Gib 1 aus, wenn der Zähler bei $count \leq z$ stehen bleibt.

   (d) Sonst gib 0 aus.

   Die Berechenbarkeit folgt wieder aus der Churchschen These.
   Diese Vorgehensweise wird bei vielen Problemen der Informatik verwendet. Ihre “Philosophie” ist, eine Frage als unbeantwortet zu betrachten, wenn die Antwort zu lange ausbleibt.

5. Das letzte Beispiel ist nicht auf den Interpreter bezogen, sondern soll zeigen, dass auch im Bereich des Berechenbaren sehr undurchsichtige Situationen entstehen können. Es rankt sich um die Dezimalentwicklung von $\pi = 3,14159265\ldots$. Gesucht werden Sequenzen aufeinanderfolgender 5en:

$$3,14\ldots\underbrace{55\ldots5}_{x-\text{mal}}\ldots$$

Dabei spielen die vorausgehenden und nachfolgenden Ziffern keine Rolle (insbesondere dürfen auch sie 5en sein).
Aus diesen Elementen lässt sich eine Funktion $foggy \colon \mathbb{N} \to \mathbb{N}$ bilden mit

$$foggy(x) = \begin{cases} 1 & \text{wenn } \pi \text{ eine 5er-Sequenz der Länge } x \text{ enthält} \\ 0 & \text{sonst.} \end{cases}$$

Verglichen mit $halt_2$ könnte *foggy* für unberechenbar gehalten werden. Denn die Dezimalentwicklung von π lässt sich beliebig weit algorithmisch herstellen, wird jedoch nie fertig. So betrachtet, kann man nie wissen, ob der Wert 0 sein muss oder noch eine geeignete Sequenz beim weiteren Entwickeln der Dezimalstellen kommt.
Mit einer anderen Überlegung erweist sich *foggy* als berechenbar. Entweder gibt es in π 5er-Sequenzen jeder Länge. Dann ist $foggy(x) = 1$ für alle $x \in \mathbb{N}$ und als konstante Funktion berechenbar. Oder es existiert eine 5er-Sequenz mit maximaler Länge max. Dann gilt

$$foggy(x) = \begin{cases} 1 & \text{für } x \leq max \\ 0 & \text{sonst.} \end{cases}$$

Für jede natürliche Zahl $max \in \mathbb{N}$ lässt sich ein *while*-Programm schreiben, das *foggy* berechnet (*if* $x \leq max$ *then* 1 *else* 0).
Es gibt für *foggy* damit unendlich viele Möglichkeiten; in jedem Fall aber ist die Berechenbarkeit sichergestellt. Der einzige Haken: Niemand weiß, welcher Fall zutrifft.

21 Anmerkung zur Literatur

Wer sich über das Skript hinaus über Automaten und formale Sprachen informieren möchte, denen sei [HMU02] als das grundlegende Werk zur theoretischen Informatik empfohlen. Die englische Ausgabe [HMU01] ist 2001 unter dem Titel Introduction to Automata Theory, Languages, and Computation im selben Verlag erschienen.

Weitere Bücher, in denen die Themen dieses Skripts unter anderen behandelt werden, sind z.B. [LP81, Sch03, ST99, EP00, Hed00, VW00, AB02, Soc03, Sud06, Hol07]. Eine Einführung in die Berechenbarkeit anhand von *while*-Programmen ist in [KMA82] zu finden.

Literatur

[AB02] Alexander Asteroth and Christel Baier. *Theoretische Informatik – Eine Einführung in Berechenbarkeit, Komplexität und formale Sprachen mit 101 Beispielen*. Pearson Education, 2002.

[AKM81] Michael A. Arbib, A.J. Kfoury, and Robert N. Moll. *A Basis for Theoretical Computer Science*. Springer, 1981.

[CS01] Edmund M. Clarke and Bernd-Holger Schlingloff. Model Checking. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, chapter 24, pages 1635–1790. Elsevier und MIT Press, 2001.

[EP00] Katrin Erk and Lutz Priese. *Theoretische Informatik. Eine umfassende Einführung*. Springer, 2000.

[Hed00] Ulrich Hedtstück. *Einführung in die Theoretische Informatik, Formale Sprachen und Automatentheorie*. Oldenbourg, 2000.

[HMU01] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 2001.

[HMU02] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie*. Addison-Wesley, 2002.

[Hol07] Boris Hollas. *Grundkurs Theoretische Informatik mit Aufgaben und Prüfungsfragen*. Spektrum Akademischer Verlag, 2007.

[KMA82] A.J. Kfoury, Robert N. Moll, and Michael A. Arbib. *A Programming Approach to Computability*. Springer, 1982.

[LP81] Harry R. Lewis and Christos H. Papadimitriou. *Elements of the Theory of Computation*. Prentice Hall, 1981.

[MSS99] Markus Müller-Olm, David Schmidt, and Bernhard Steffen. Model-checking a tutorial introduction. In *Proc. Static Analysis Symposium*, volume 1694 of *Lecture Notes in Computer Science*, pages 330–354, 1999.

[Sch03] Uwe Schöning. *Theoretische Informatik – kurzgefaßt* (4. Auflage). Spektrum Akademischer Verlag, 2003.

[Soc03] Rolf Socher. *Theoretische Grundlagen der Informatik*. Fachbuchverlag Leipzig, 2003.

[ST99] Dan A. Simovici and Richard L. Tenney. *Theory of Formal Languages with Applications*. World Scientific, 1999.

[Sud06] Thomas A. Sudkamp. *An Introduction to the Theory of Computer Science, Third Edition*. Pearson Education, 2006.

[VW00] Gottfried Vossen and Kurt-Ulrich Witt. *Grundlagen der Theoretischen Informatik mit Anwendungen*. Vieweg, 2000.