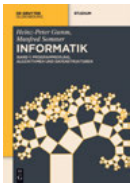
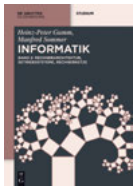
Heinz-Peter Gumm, Manfred Sommer
Informatik
De Gruyter Studium

Weitere empfehlenswerte Titel



Informatik, Band 1: Programmierung, Algorithmen und Datenstrukturen
H.P. Gumm, M. Sommer, 2016
ISBN 978-3-11-044227-4, e-ISBN (PDF) 978-3-11-044226-7,
e-ISBN (EPUB) 978-3-11-044231-1



Informatik, Band 2: Rechnerarchitektur, Betriebssysteme, Rechnernetze
H.P. Gumm, M. Sommer, 2017
ISBN 978-3-11-044235-9, e-ISBN (PDF) 978-3-11-044236-6,
e-ISBN (EPUB) 978-3-11-043442-2

IT-Sicherheit, 10. Auflage
C. Eckert, 2018
ISBN 978-3-11-055158-7, e-ISBN (PDF) 978-3-11-056390-0,
e-ISBN (EPUB) 978-3-11-058468-4

Maschinelles Lernen, 2. Auflage
E. Alpaydin, 2019
ISBN 978-3-11-061788-7, e-ISBN (PDF) 978-3-11-061789-4,
e-ISBN (EPUB) 978-3-11-061794-8

Rechnerorganisation und Rechnerentwurf, 5. Auflage
D. Patterson, J.L. Hennessy, 2016
ISBN 978-3-11-044605-0, e-ISBN 978-3-11-044606-7,
e-ISBN (EPUB) 978-3-11-044612-8

Heinz-Peter Gumm, Manfred Sommer

Informatik

Band 3: Formale Sprachen, Compilerbau,
Berechenbarkeit und Komplexität

DE GRUYTER
OLDENBOURG

Autoren
Prof. Dr. Heinz-Peter Gumm
Philipps-Universität Marburg
Fachbereich Mathematik
und Informatik
Hans-Meerwein-Straße
35032 Marburg
gumm@mathematik.uni-marburg.de

Prof. Dr. Manfred Sommer
Elsenhöhe 4B
35037 Marburg
manfred.sommer@gmail.com

ISBN 978-3-11-044238-0
e-ISBN (PDF) 978-3-11-044239-7
e-ISBN (EPUB) 9 978-3-11-043405-7

Library of Congress Control Number: 2019939344

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.dnb.de abrufbar.

Druck und Bindung: CPI books GmbH, Leck

www.degruyter.com

Formale Sprachen, Compilerbau, Berechenbarkeit und Komplexität

Inhalt

Vorwort

Dieses Buch ist der dritte und letzte Band unseres Buchprojekts zum Thema „Informatik“, das sich zum Ziel gesetzt hat, eine allgemeine Einführung in dieses faszinierende Gebiet zu geben.

Hier wird es deutlich theoretischer – aber die Theorie hat einen praktischen Nutzen. Formale Sprachen und Grammatiken bilden die Grundlage für die Spracherkennung, die fundamentaler Bestandteil eines jeden Compilers ist. Aus diesem Grunde haben wir auch ein Kapitel zum Compilerbau eingeschoben, das zeigt, wie die Theorie nutzbringend angewendet werden kann. In einem Hochschulkurs zur *Theoretischen Informatik* kann man dieses Kapitel getrost überschlagen oder aber Teile davon als Motivation einflechten. Leser, die speziell an dem Thema Compilerbau interessiert sind, erfahren hier, mit vielen Beispielen, wie Compiler arbeiten, welche Probleme es bei der Spracherkennung gibt, und welche Lösungen existieren.

Im Kapitel Berechenbarkeit werden die wichtigsten Berechenbarkeitsmodelle vorgestellt und deren Äquivalenz gezeigt. Auch die Aufgabenstellungen, die mathematisch-technisch zwar exakt definiert sind, die aber grundsätzlich algorithmisch nicht lösbar sind werden vorgestellt und die Unmöglichkeit ihrer Lösung bewiesen.

Im letzten Kapitel widmen wir uns den zwar theoretisch lösbaren, aber praktisch nur mit unzumutbarem Aufwand beherrschbaren Aufgaben und gelangen zur derzeit wichtigsten noch offenen Frage der theoretischen Informatik, auf die sogar ein Kopfgeld von 1 Million $ ausgesetzt ist:

$$P \neq NP?$$

Marburg an der Lahn, im April 2019

Heinz-Peter Gumm
Manfred Sommer

https://doi.org/10.1515/9783110442397-001

Kapitel 1

Formale Sprachen

Theoretische Informatik und Mathematik schaffen die Basis für viele der technischen Entwicklungen, die wir in den drei Bänden dieser Reihe besprechen. Die boolesche Algebra legt die theoretischen Grundlagen für den Bau digitaler Schaltungen, die Theorie formaler Sprachen zeigt auf, wie die Syntax von Programmiersprachen aufgebaut werden sollte, damit Programme einfach und effizient in lauffähige Programme übersetzt werden können, und die Theorie der Berechenbarkeit deckt exakt die Grenzen des Berechenbaren auf. Sie zieht eine klare Linie zwischen dem was man prinzipiell programmieren kann und dem was mit Sicherheit nicht von einem Rechner gelöst werden kann.

Ein fehlerfreies Programm ist noch lange nicht korrekt. Wünschenswert wäre es, wenn man feststellen könnte, ob jede Schleife auch terminiert. Dass diese und ähnliche semantischen Eigenschaften nicht automatisch geprüft werden können, ist eine der Konsequenzen der Berechenbarkeitstheorie. Mit dieser kann man auch zeigen, dass, wenn man einmal von Geschwindigkeit und Speicherplatz absieht, alle Rechner in ihren mathematischen Fähigkeiten identisch sind und damit das gleiche können bzw. nicht können.

Schließlich hilft uns die Komplexitätstheorie, Aussagen über den Aufwand zu machen, den man zur Lösung wichtiger Probleme treiben muss. Sie zeigt uns Grenzen des praktisch Machbaren auf und führt uns zu dem bekanntesten und seit Jahrzehnten ungelösten Problem der Informatik, ob gewisse relevante Probleme prinzipiell nicht effizienter gelöst werden können, als durch systematisches Ausprobieren von plausiblen Lösungsmöglichkeiten. Wenn man eines dieser Probleme effizient lösen könnte, dann würde das für alle anderen in dieser Problemgruppe auch gelten. Mathematisch kann man dies in der Unleichung $P \stackrel{?}{=} NP$ zusammenfassen – das Fragezeichen deutet an, dass die Antwort noch offen ist. Sogar ein Preisgeld von einer Million US$ ist auf die Beantwortung dieser Frage ausgesetzt.

In diesem Band wollen wir also einen Ausflug in die theoretische Informatik unternehmen. Wir werden sehen, dass diese Theorie nicht trocken ist, sondern unmit-

https://doi.org/10.1515/9783110442397-002

telbare praktische Anwendungen hat. Die Automatentheorie zeigt, wie man effizient die Wörter einer Programmiersprache festlegen und erkennen kann, die Theorie der kontextfreien Sprachen lehrt, wie man die Grammatik einer Programmiersprache definieren sollte, damit man weitestgehend automatisiert Übersetzer und Compiler dafür erzeugen kann.

1.1 Formale Beschreibungen

Bei der Kommunikation mit Rechnern bedienen wir uns formaler Beschreibungen der erlaubten Eingaben.

- Im einfachsten Fall sollen wir bei der Registrierung für eine Webseite ein Passwort wählen. Das Eingabeformular weist uns darauf hin, dass nur Buchstaben, Ziffern und bestimmte Sonderzeichen erlaubt sind, mindestens aber eine Ziffer und ein Sonderzeichen auftreten müssen. Gültig ist also z.B. `4u2p@3`, ungültig wären `test7` oder `@home`.
- Das Feld „Geburtsdatum“ erwartet eine Eingabe im Format `tt.mm.jjjj`, also mit zweistelliger Tages- und Monatsangabe, und einem vierstelligen Jahr zwischen 1900 und 2099, jeweils durch einen Punkt getrennt.
- In einem Java-Programm dürfen wir selbstgewählte Namen für Klassen, Methoden und Variablen benutzen. Allerdings müssen diese mit einem Buchstaben beginnen und dürfen ansonsten beliebige Buchstaben, Ziffern oder das Sonderzeichen „_“ beinhalten.
- Dezimalzahlen in Java können in verschiedenen Formaten eingegeben werden, wie die Beispiele 0.17, 3.1415927, 0.4 E -10 oder 42 zeigen.
- Integerzahlen dürfen nicht mit der Ziffer `0` beginnen, es sei denn, es handelt sich um die Zahl `0`, oder um eine Oktalzahl oder um eine Hexadezimalzahl. Integerzahlen dürfen zwischen je zwei Ziffern auch beliebig viele Unterstriche ’_’ enthalten.
- Oktalzahlen beginnen mit 0 und die folgenden Ziffern müssen aus dem Bereich `0...7` sein
- Hexadezimalzahlen beginnen mit `0x` und dürfen neben den Ziffern `0...9` auch die Buchstaben `a...f`, oder `A...F` verwenden.

In allen genannten Beispielen könnten wir ohne allzu große Mühen jeweils ein Programm erstellen, welches einen Input entgegennimmt und die Einhaltung der Regeln prüft. Allerdings würde dies heute niemand mehr von Hand machen. Zur Beschreibung der erlaubten Eingaben bedient man sich einer einfachen Notation, der sogenannten *regulären Ausdrücke,* mit denen man die erlaubten Eingaben beschreiben kann.

Eine solche formale Beschreibung hat zwei offensichtliche Vorteile. Zunächst wird präziser als in der Umgangssprache möglich, eindeutig festgelegt, was jeweils erlaubt ist und was nicht. Die obige Beschreibung der Integerzahlen in Java klärt nicht eindeu-

tig, welche der folgenden „Zahlen“ erlaubt sind und welche nicht: `007`, `008`, `077`, `7_7`, `-7_7`, `_77`, `1__0`, `0x_ab`, `0xa_B`.

Mit der Notation der *regulären Ausdrücke* könnte man die legalen Integerzahlen von Java zum Beispiel folgendermaßen spezifizieren:

```
0 | [1-9](_*[0-9])* | 0[0-7](_*[0-7])* | 0x[0-9a-fA-F](_*[0-9a-fA-F])*
```

Dieser Ausdruck spezifiziert, jeweils durch das Alternativenzeichen `'|'` getrennt, 4 verschiedene Muster, wie Integerkonstanten aussehen dürfen:

`0` : Dieses Muster passt nur auf die Ziffer 0 und sonst nichts

`[1-9](_*[0-9])*` :
Dieses Muster beschreibt eine dezimale Zahl, die mit einer Ziffer aus dem Bereich **`[1-9]`** beginnt. Der Rest der Zahl muss dem Muster **`(_*[0-9])*`** genügen. Der Stern '**`*`**' signalisiert optionale Wiederholungen eines Musters. In unserem Falle haben wir also beliebig viele Teile, die selber jeweils dem Muster **`_*[0-9]`** genügen. Letzteres erlaubt wegen '**`_*`**' beliebig viele Unterstriche '**`_`**', gefolgt von jeweils einer Ziffer. Insgesamt passen also zum Beispiel `9`, `9__8`, `9___8_76` und `987` auf dieses Muster, nicht aber `0`, `10__` , oder `_11`.

`0[0-7](_*[0-7])*` :
Eine 0 direkt gefolgt von mindestens einer Oktalziffer und danach ggf. weitere Oktalziffern, vor denen jeweils beliebig viele Unterstriche stehen können, wird als Oktalzahl interpretiert. Erlaubt sind z.B. `007` oder `00_7`, nicht aber `0_07` oder `008`.

`0x[0-9a-fA-F](_*[0-9a-fA-F])*` :
Der Bereich **`[0-9a-fA-F]`** steht für ein Zeichen aus den Bereichen **`[0-9]`**, **`[a-f]`**, oder **`[A-F]`**, also für eine Hexziffer. Das Muster beginnt mit **`0x`** gefolgt von einer Hexziffer. Danach kommen beliebig viele weitere Hexziffern denen jeweils eine Folge von Unterstrichen vorangestellt werden können.

1.1.1 Verwendung in Python

Der zweite Vorteil einer formalen Beschreibung zeigt sich darin, dass aus einer solchen automatisch ein tabellengesteuertes Programm erzeugt werden kann, das eine konkrete Eingabe daraufhin überprüft, ob sie die gesetzten Regeln erfüllt oder nicht. Wir können den oben gezeigten regulären Ausdruck mit Hilfe der Python-Bibliothek `re` ohne weiteres in ein Muster compilieren. Zunächst laden wir dazu die genannte Bibliothek aus der „Python Standard Library“:

```
import re
```

Die Alternativen

```
nul = '0'
dez = '[1-9](_*[0-9])*'
okt = '0[0-7]?(_*[0-7])*'
hx  = '0x[0-9a-fA-F](_*[0-9a-fA-F])*'
```

werden zu einem Muster zusammengefügt:

```
pattern = nul + '|' + dez + '|' + okt + '|' + hx
```

und zu einem Erkenner compiliert:

```
pos_int = re.compile(pattern)
```

Anschließend versuchen wir zu testen, ob eines unserer Beispiele von der gewünschten Bauart ist, z.B.:

```
tests = ['0', '007', '008', '4__2', '4_2_', '0xC_B','0x_CB']

for s in tests :
   if re.fullmatch(pos_int,s) != None :
      print(s,'\t : OK' )
   else :
      print(s,'\t : Illegal')
```

erzeugt die Ausgabe:

```
0     : OK
007   : OK
008   : Illegal
4__2  : OK
4_2_  : Illegal
0xC_B : OK
0x_CB : Illegal.
```

Im ersten Teil dieses Kapitels werden wir uns mit sogenannten *regulären Sprachen* beschäftigen. Dies sind Mengen von Zeichenketten, die sich wie oben durch einfache Muster definieren lassen. Anschließend werden wir erkennen, dass die Menge aller Zeichenfolgen, die ein korrektes Programm einer Programmiersprache ausmachen, nicht so einfach beschreibbar ist. Wir werden rekursive Beschreibungsmethoden benötigen, was uns zu den *kontextfreien Grammatiken* führt.

Sowohl reguläre Ausdrücke als auch kontextfreie Grammatiken lassen sich in Programme übersetzen, die einen String testen, ob er den Regeln genügt, oder nicht. Diese Aufgabe nennen wir *parsen* (engl. für *zerteilen*) und diese Aufgabe stellt die ers-

te Phase eines Compilers dar. Wie dies funktioniert, wie die zugehörigen Programme aussehen und was theoretisch möglich bzw. unmöglich ist, davon handelt dieses Kapitel.

1.2 Alphabete, Worte und Sprachen

1.2.1 Alphabete

Ein *Alphabet* definiert die Zeichen, die bei der Eingabe einer formalen Beschreibung verwendet werden dürfen. Für ein Programm in der Sprache C stehen beliebige druckbare ASCII-Zeichen zur Verfügung. Das Alphabet, aus dem C-Programme aufgebaut werden können, besteht dann aus den Buchstaben `a`,...,`z` und `A`,...,`Z`, den Ziffern `0`,...,`9` und Sonderzeichen wie `*`,`+`,`#`,`"`,`?`, etc.. Um Binärzahlen zu notieren, reichen die beiden Zeichen 0 und 1 aus, und für Programme in der Sprache LISP reichen die Großbuchstaben `A`,..,`Z`, die Ziffern `0`,...,`9` und die Klammern „`)`“ und „`(`“. Da jede beliebige Zeichenmenge als Alphabet für gewisse Zwecke dienen kann, definieren wir einfach:

Definition 1.2.1. Ein *Alphabet* ist eine endliche Menge Σ. Die Elemente von Σ nennen wir *Zeichen*.

Typischerweise benennen wir Alphabete mit den griechischen Buchstaben Σ oder Γ. Einige in der Praxis vorkommende Alphabete sind:

- $\Sigma_1 = \{1\}$, das unäre Alphabet,
- $\Sigma_2 = \{0, 1\}$, das Binäralphabet bzw. $\Sigma_{()} = \{(,)\}$ das Klammeralphabet,
- $\Sigma_M = \{-, \cdot, \text{␣}\}$, das Morse-Alphabet bestehend aus „lang“, „kurz“ und dem Leerzeichen,
- $\Sigma_{16} = \{0, 1, \ldots, 9, A, B, .., F\}$, das Hex-Alphabet,
- $\Sigma_{lat} = \{A, B, \ldots, Z, a, .., z\}$, das lateinische Alphabet,
- $\Sigma_{dt} = \Sigma_{lat} \cup \{\ddot{A}, \ddot{O}, \ddot{U}, \ddot{a}, \ddot{o}, \ddot{u}, ß\}$, das deutsche Alphabet,
- $\Gamma = \{\alpha, \beta, y, \ldots\}$, das griechische Alphabet,
- $\Sigma_{ASCII} = \{a, \ldots, z, A, \ldots, Z, 0, \ldots, 9, (,), \#, +, \sim, !, \text{␣}, \&, \ldots\}$, die ASCII-Zeichen.

1.2.2 Worte

Die Zeichen eines Alphabets dienen dazu, Worte zu bilden, indem man einige Zeichen des Alphabets aneinanderhängt. Somit definiert man:

Definition 1.2.2 (informal)**.** Ein *Wort* über einem Alphabet Σ ist eine endliche Folge von Zeichen aus Σ. Die Menge aller Worte über Σ bezeichnen wir mit Σ^*.

So ist z.B. „*Hello*“ ein Wort über Σ_{lat}, „0001011“ ein Wort über Σ_2 und „$(LIS(P))$“ ein Wort über Σ_{ASCII}. Mathematisch gesprochen ist also $Hello \in \Sigma_{ASCII}^{\star}$, $0001011 \in \Sigma_2^{\star}$ und $(LIS(P)) \in \Sigma_{ASCII}^{\star}$. Da Worte beliebig lang sein dürfen, ist klar,dass $\Sigma^{\star}$ immer unendlich viele Worte enthält. Ein besonderes Wort ist *das leere Wort*, das wir mit ε notieren. Für jedwedes Alphabet Σ gilt daher

$$\varepsilon \in \Sigma^{\star}.$$

Die *Länge* eines Wortes ist die Anzahl der Zeichen, aus denen es besteht. Beispielsweise hat ε die Länge 0 und 0001011 die Länge 7.

Obwohl $\Sigma^{\star}$ immer unendlich viele Elemente enthält, kann man alle Worte aus $\Sigma^{\star}$ systematisch aufzählen:

- $\{1\}^{\star} = \{\varepsilon, 1, 11, 111, 1111, \ldots\}$ kann man mit den natürlichen Zahlen identifizieren. Der Zahl n ordnen wir das Wort zu, das aus n vielen 1-en besteht, der Zahl 0 das leere Wort ε.
- $\{0, 1\}^{\star} = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, 101, 110, 111, 0000, \ldots\}$. Auch hier kann man die Worte aufzählen, also explizit durchnummerieren. Zuerst kommt das leere Wort ε, dann alle Worte der Länge 1, dann der Länge 2 und so weiter.
- $\{a, b, \ldots, z\}^{\star} = \{\varepsilon, a, b, \ldots, z, aa, ab, \ldots, ba, bb, bc, \ldots\}$. Diese Menge lässt sich analog aufzählen, indem man erst ε, dann alle Worte der Länge 1, dann alle Worte der Länge 2, etc., aufzählt, wobei die Worte gleicher Länge z.B. in lexikalischer Reihenfolge geordnet werden können.

Die meisten Programmiersprachen besitzen einen Datentyp *Character* den man als Ausgabealphabet verwenden kann. Die Worte über diesem Alphabet heißen dann *Strings* und das leere Wort wird mit “␣“ bezeichnet.

1.2.3 Konstruktion von Worten

Für die Angabe von Worten gibt es zwei Methoden. Entweder gibt man die Folge aller Zeichen (ohne Zwischenraum) an, wie z.B. ”$Hello$” , oder man schreibt ein Wort w als $a \cdot v$ wobei $a \in \Sigma$ das erste Zeichen ist und $v \in \Sigma^{\star}$ das Restwort.

Die Operation „$\cdot$“ des „Voranstellens“ eines Zeichens vor ein Wort ist eine *Konstruktorfunktion*, die ein Zeichen $a \in \Sigma$ und ein Wort $v \in \Sigma^{\star}$ nimmt und das neue Wort $a \cdot v$ liefert. Beginnend mit dem leeren Wort ε entsteht beispielsweise das Wort ”$Hello$” durch sukzessives Voranstellen der Zeichen o, l, l, e, H als $H \cdot (e \cdot (l \cdot (l \cdot (o \cdot \varepsilon))))$.

Die Klammern deuten die Reihenfolge an, in der die Operation „$\cdot$“ = *Voranstellen* ausgeführt wurde. Diese Klammern werden wir in Zukunft weglassen, da ohnehin jede andere Klammerung keinen Sinn ergäbe.

Jedes nichtleere Wort lässt sich eindeutig in der Form $a \cdot v$ darstellen: a ist dabei das erste Zeichen und v das Restwort.

1.2.4 Worte als Listen von Zeichen

In Programmiersprachen wird oft ein Datentyp *Character* (oder *char*) bereitgestellt, der die Zeichen eines Alphabets repräsentiert. Worte über diesem Alphabet heißen dann *Strings*. Manchmal werden Strings auch lediglich als Listen von Zeichen angesehen und implementiert. Der String `''Hello''` ist dann nichts anderes als die Liste `[H,e,l,l,o]`. Das erste Zeichen des Strings, `'H'`, ist der Kopf (engl.: *head*) der Liste und der Rest des Strings, `''ello''`, ist die Restliste (engl.: *tail*) von `[H,e,l,l,o]`.

Der Operation „$\cdot$“ des Voranstellens eines Zeichens a vor ein Wort w entspricht dann die Operation, die ein Element a vorne in eine Liste einfügt. Diese Operation wird je nach Programmiersprache `a:l` oder `a#l` oder `cons(a,l)` geschrieben. Jede nichtleere Liste lässt sich eindeutig in der Form $cons(a, l)$ darstellen – a ist der Kopf (engl.: *head*) der Liste und l der Rest (engl.: *tail*). Insofern kann man Worte auch als Liste von Zeichen definieren. Das leere Wort ε entspricht dabei der leeren Liste [ ].

Die bisherige Definition von Worten ist insofern noch informal, als sie den Begriff der „endlichen Folge“ benutzt, der zwar intuitiv klar ist, der sich aber in dieser Form noch nicht zur formalen Behandlung von Worten eignet. Daher präzisieren wir die obige intuitive Definition von Σ^* durch das folgende Konstruktionsprinzip, das angibt, wie Worte aus Zeichen mithilfe der Konstruktoren ε (leeres Wort) und „$\cdot$“ (Voranstellen) entstehen. Diese Definition hat den Vorteil, dass sie als Grundlage von induktiven Beweisen benutzt werden kann:

Definition 1.2.3. [formal] Σ^* ist die *kleinste Menge*, die folgende Eigenschaften erfüllt:

$$\varepsilon \in \Sigma^* \tag{1.2.1}$$

$$a \in \Sigma, u \in \Sigma^* \implies a \cdot u \in \Sigma^* \tag{1.2.2}$$

Eine Teilmenge $P \subseteq \Sigma^*$ von Worten, die das leere Wort enthält und mit jedem Wort $u \in P$ auch $a \cdot u$ muss schon ganz Σ^* sein. Dieser systematische (genauer: *induktive*) Aufbau von Worten ist in zweierlei Hinsicht relevant. Einmal für die Definition von rekursiven Funktionen auf Worten und zum anderen für induktive Beweise von Eigenschaften.

Auch wir werden Worte durch Angabe ihrer Zeichen notieren, also z.B. das Wort „$H \cdot e \cdot l \cdot l \cdot o \cdot \varepsilon$“ als „$Hello$“. Für rekursive Funktionen über Sprachen oder für Beweise werden wir uns dann erinnern, dass das Wort „$Hello$“ eigentlich aus einem ersten Zeichen $H \in \Sigma$ und einem Restwort $ello \in \Sigma^*$ zusammengesetzt ist. Wenn wir $a \cdot u$ schreiben, sollte klar sein, dass a ein Zeichen ($a \in \Sigma$) und u ein Wort ($u \in \Sigma^*$) ist.

1.2.5 Länge

Unter den Operationen auf Strings finden sich immer Operationen zur Bestimmung der Länge, oder eine Operation um zwei Strings aneinanderzuhängen, die wahlweise

als *append* oder *concatenate* oder einfach symbolisch als „$\circ$“ oder „+“ geschrieben wird. Die Funktion *length*, die die Anzahl der Zeichen in einem Wort misst, kann man rekursiv folgendermaßen definieren:

$$length(\varepsilon) = 0 \tag{1.2.3}$$
$$length(a \cdot u) = 1 + length(u) \tag{1.2.4}$$

Für den Definitionsbereich D der so definierten Funktion gilt offensichtlich $\varepsilon \in D$ und wenn $u \in D \Longrightarrow a \cdot u \in D$. Laut der formalen Definition von Σ^* muss also $D = \Sigma^*$ sein, folglich ist *length* für alle Worte $w \in \Sigma^*$ definiert.

Aufgabe 1.2.4. Sei Σ ein endliches Alphabet.

1. Konstruieren Sie (z.B. als Programm) eine bijektive Funktion $f : \Sigma^* \to \mathbb{N}$. Geben Sie die Umkehrfunktion f^{-1} konkret an. (Vorsicht: Falls z.B. $\Sigma = \{0, 1\}$ ist, muss $f(\varepsilon) \neq f(0) \neq f(00) \neq \ldots$ sein!)
2. Folgern Sie, dass es für beliebige endliche Alphabete Σ und Γ jeweils bijektive Codierungsfunktionen $c : \Sigma^* \to \Gamma^*$ und $c^{-1} : \Gamma^* \to \Sigma^*$ gibt, die Worte über Σ als Worte über Γ codieren und umgekehrt.

1.2.6 Das Induktionsprinzip für Worte

In der Praxis sind Teilmengen von Σ^* meist durch *Eigenschaften* beschrieben, etwa *„alle Worte, die mit ´a´ beginnen“*, oder *„alle Worte in denen gleich viele 'a'-s wie 'b'-s vorkommen“*, oder *„alle Worte, für die die Funktion $f : \Sigma^* \to X$ definiert ist“*.

Aus einer Teilmenge $P \subseteq \Sigma^*$ gewinnen wir unmittelbar die gleichnamige Eigenschaft

$$\bar{P}(w) :\Longleftrightarrow (w \in P)$$

und umgekehrt liefert jede Eigenschaft $\bar{P}(w)$ eine Teilmenge

$$P = \{w \in \Sigma^* \mid \bar{P}(w)\},$$

die aus denjenigen Worten $w \in \Sigma^*$ besteht, die $\bar{P}$ erfüllen. Somit folgt aus der formalen Definition 1.2.3 von Σ^* sofort das Induktionsprinzip für Worte:

Induktionsprinzip für Worte: Sei $P(w)$ eine Eigenschaft für Worte $w \in \Sigma^*$. Gilt $P(\varepsilon)$ und folgt aus $P(u)$ immer auch $P(a \cdot u)$, so gilt $P(w)$ für alle $w \in \Sigma^*$.

Mathematisch formuliert man dieses gerne als *Schlussregel*:

$$\frac{P(\varepsilon) \qquad \forall a \in \Sigma. \forall u \in \Sigma^*. P(u) \implies P(a \cdot u)}{\forall w \in \Sigma^*. P(w)}$$

Induktionsanfang

Induktionsschritt

Induktionsschluss

Der „Bruchstrich“ trennt dabei nur Voraussetzungen und Konsequenz. In diesem Falle haben wir zwei Voraussetzungen, die auch als

Induktionsanfang: $P(\varepsilon)$

Induktionsschritt: $\forall a \in \Sigma. \forall u \in \Sigma^*. P(u) \implies P(a \cdot u)$

bezeichnet werden. Der Induktionsanfang $P(\varepsilon)$ ist meist leicht nachzuprüfen. Für den *Induktionsschritt*

$$P(u) \implies P(a \cdot u)$$

nimmt man an, die Eigenschaft $P(u)$ sei für ein u bereits bewiesen. Dies nennt man oft auch *Induktionshypothese*. Daraus muss man folgern dass auch $P(a \cdot u)$ gilt.

Sind Induktionsanfang und Induktionsschritt nachgewiesen, so darf man den Induktionsschluss

$$\forall w \in \Sigma^*. P(w)$$

als bewiesen akzeptieren.

Typischerweise verwendet man solche Schlussregeln von hinten nach vorne: Man möchte eine Eigenschaft der Form $\forall w \in \Sigma^*. P(w)$ mittels Induktion zeigen.

Für den Induktionsanfang setzt man ε für w ein und zeigt $P(\varepsilon)$. Für den Induktionsschritt setzt man $P(u)$ voraus und beweist damit $P(a \cdot u)$.

1.3 Operationen mit Worten

Um zwei Worte u und v zusammenzufügen, definieren wir eine Funktion $\circ$:

$$\varepsilon \circ v = v \tag{1.3.1}$$

$$(a \cdot u) \circ v = a \cdot (u \circ v) \tag{1.3.2}$$

Wir wollen zeigen, dass für alle Worte w und v gilt:

$$length(w \circ v) = length(w) + length(v). \tag{1.3.3}$$

1.3.1 Ein Induktionsbeweis

Die Aussage (1.3.3) beweisen wir jetzt durch Induktion über w. Die Aussage hat die geeignete Form

$$\forall w \in \Sigma^*. P(w)$$

wenn wir

$$P(w) := \forall v \in \Sigma^*. length(w \circ v) = length(w) + length(v)$$

wählen. Für den Induktionsanfang ist

$$P(\varepsilon) = \forall v \in \Sigma^*. length(\varepsilon \circ v) = length(\varepsilon) + length(v)$$

zu überprüfen und für den Induktionsschritt $P(u) \Longrightarrow P(a \cdot u)$.

Induktionsanfang: $w = \varepsilon$:

$$\begin{aligned} length(\varepsilon \circ v) &\overset{(1.3.1)}{=} length(v) \\ &= 0 + length(v) \\ &\overset{(1.2.3)}{=} length(\varepsilon) + length(v). \end{aligned}$$

Induktionsschritt: $w = a \cdot u$: Die Induktionshypothese $P(u)$ lautet

$$\forall v \in \Sigma^*. length(u \circ v) = length(u) + length(v).$$

Daraus schließen wir $P(a \cdot u)$:

$$\begin{aligned} length((a \cdot u) \circ v) &\overset{(1.3.2)}{=} length(a \cdot (u \circ v)) \\ &\overset{(1.2.4)}{=} 1 + length(u \circ v) \\ &\overset{(IH)}{=} 1 + length(u) + length(v) \\ &\overset{(1.2.4)}{=} length(a \cdot u) + length(v) \end{aligned}$$

Somit ist die Aussage (1.3.3) aufgrund des Induktionsprinzips bewiesen.

1.3.2 Die Wahl der Induktionsvariablen

Eine wichtige Überlegung vor Beginn eines Induktionsbeweises ist immer, über welche Variable man die Induktion führt. Im Formel (1.3.3) gibt es eine weitere Variable vom Typ Σ^*, nämlich v. Man hätte die gewünschte Behauptung auch so formulieren können $\forall v. Q(v)$ mit :

$$Q(v) := \forall w \in \Sigma^*. length(w \circ v) = length(w) + length(v)$$

In diesem Falle würde der Induktionsanfang lauten:

$$Q(\varepsilon) = \forall w \in \Sigma^\star . length(w \circ \varepsilon) = length(w) + length(\varepsilon),$$

was wegen (1.2.3) zwar vereinfacht zu

$$\forall w \in \Sigma^\star . length(w \circ \varepsilon) = length(w),$$

aber da stockt der Beweis erst einmal, denn zwar gilt $\varepsilon \circ w = w$ aufgrund von 1.3.1, aber damit ist noch nicht klar, ob auch $w \circ \varepsilon = w$ stimmt. Wir müssen dies vorher beweisen, und dies ist auch durch eine einfache Induktion möglich, die wir dem Leser überlassen:

Aufgabe 1.3.1. Beweisen Sie mittels Induktion: $\forall w \in \Sigma^\star . w \circ \varepsilon = w$.

Auch die Assoziativität der Operation $\circ$

$$w \circ (x \circ y) = (w \circ x) \circ y \tag{1.3.4}$$

für beliebige $w, x, y \in \Sigma^\star$ folgt leicht durch eine Induktion über die Variable w. Hier ist die Behauptung $\forall w.P(w)$, diesmal mit

$$P(w) := \forall x, y \in \Sigma^\star . w \circ (x \circ y) = (w \circ x) \circ y.$$

Für den Induktionsanfang setzen wir $w := \varepsilon$ und erhalten $P(\varepsilon)$ mit

$$\begin{aligned} \varepsilon \circ (x \circ y) &\overset{1.3.1}{=} x \circ y && (1.3.5)\\ &\overset{1.3.1}{=} (\varepsilon \circ x) \circ y && (1.3.6) \end{aligned}$$

und für den Induktionsschritt $w = a \cdot u$ folgt mit der Induktionshypothese (IH)

$$P(u) = \forall x, y \in \Sigma^\star . u \circ (x \circ y) = (u \circ x) \circ y,$$

dass auch $P(a \cdot u)$ gilt, denn

$$\begin{aligned} (a \cdot u) \circ (x \circ y) &\overset{1.3.2}{=} a \cdot (u \circ (x \circ y)) && (1.3.7)\\ &\overset{IH}{=} a \cdot ((u \circ x) \circ y) && (1.3.8)\\ &\overset{1.3.2}{=} (a \cdot (u \circ x)) \circ y && (1.3.9)\\ &\overset{1.3.2}{=} ((a \cdot u) \circ x) \circ y && (1.3.10) \end{aligned}$$

1.3.3 $\Sigma^\star$ als Monoid

Eine Struktur, wie $\Sigma^\star$ mit einer assoziativen Verknüpfung $\circ$ und einem neutralen Element e nennt man ein *Monoid*:

Definition 1.3.2 (Monoid)**.** Ein Monoid besteht aus einer nichtleeren Menge M mit einer Konstanten $e \in M$ und einer zweistelligen Verknüpfung $\circ : M \times M \to M$ so dass die folgenden Gleichungen für alle $x, y, z \in M$ gelten:

$x \circ e = x = e \circ x$

$x \circ (y \circ z) = (x \circ y) \circ z$.

Die Menge Σ^* aller Worte über Σ bildet also ein Monoid mit neutralem Element $\varepsilon \in \Sigma^*$ und der Konkatenation $\circ$ als Verknüpfung.

Die Assoziativität erlaubt uns, Klammern und auch das Verknüpfungszeichen $\circ$ wegzulassen, wie wir es z.B. von der Multiplikation von Zahlen gewohnt sind. Infolgedessen schreiben wir zukünftig meist uv statt $u \circ v$. Wegen der Assoziativität muss man sich auch keine Sorgen machen, ob mit uvw nun $u(vw)$ oder $(uv)w$ gemeint sein soll.

1.3.4 Zeichen als Worte der Länge 1

Formal muss man zwischen dem Zeichen $a \in \Sigma$ und dem Wort $a \cdot \varepsilon \in \Sigma^*$ unterscheiden, so wie man beim Programmieren auch zwischen dem Zeichen e und dem String "e" unterscheiden muss. Zum Glück folgt mit (1.3.1) und (1.3.2) sofort, dass

$$(a \cdot \varepsilon) \circ v = a \cdot (\varepsilon \circ v) = a \cdot v \tag{1.3.11}$$

für jedes $a \in \Sigma$ und $v \in \Sigma^*$ gilt. Wenn wir also das Zeichen a mit dem Wort $a \cdot \varepsilon$ identifizieren, dann können wir einfach au schreiben statt $a \cdot u$.

Analog schreiben wir auch $u \cdot a$ oder einfach ua für das Wort, das aus u entsteht, indem wir das Zeichen a hinten anhängen:

$$u \cdot a := u \circ (a \cdot \varepsilon) \tag{1.3.12}$$

Wir können uns mit Hilfe dieser Definition und Gleichung 1.3.2 vergewissern, dass es gleichgültig ist, ob wir an das Wort u zuerst das Zeichen a vorne und dann das Zeichen b hinten anhängen, oder umgekehrt zuerst b hinten, dann a vorne:

$$\begin{aligned} (a \cdot u) \cdot b &\overset{1.3.12}{=} (a \cdot u) \circ (b \cdot \varepsilon) \\ &\overset{1.3.2}{=} a \cdot (u \circ (b \cdot \varepsilon)) \\ &\overset{1.3.12}{=} a \cdot (u \cdot b) \end{aligned}$$

Aufgabe 1.3.3. Rechnen Sie nach, dass für beliebige Worte $u, v \in \Sigma^*$ und beliebige Zeichen $a \in \Sigma$ gilt: $(u \cdot a) \circ v = u \circ (a \cdot v)$.

1.3.5 Reverse

Während diese informelle Kurznotation bequem und gut lesbar ist, ist es für Definitionen und Beweise oft jedoch sinnvoll, die Operationen explizit anzugeben. Dies wird

auch deutlich, wenn wir eine Funktion *reverse* definieren, die die Reihenfolge der Zeichen eines Wortes umdrehen soll. Statt $reverse(u)$ ist auch die Notation u^R üblich, die wir in der folgenden Definition verwenden:

$$\varepsilon^R = \varepsilon \tag{1.3.13}$$
$$(a \cdot u)^R = u^R \circ (a \cdot \varepsilon) \tag{1.3.14}$$

Man zeigt leicht durch Induktion über w, dass für alle Worte $w, v \in \Sigma^* gilt$:

$$(w \circ v)^R = v^R \circ w^R. \tag{1.3.15}$$

Palindrome

Ein *Palindrom* ist ein Wort u mit $u^R = u$. Beispiele für Palindrome sind: `rentner`, `otto`, `nun`, `neben`, `reliefpfeiler`, `regallager`, `1001`, und ε. Eine induktive Definition lautet:

- ε ist ein Palindrom

und für jedes Zeichen $a \in \Sigma$ und jedes Wort $u \in \Sigma^*$ gilt

- a ist ein Palindrom, und
- wenn u ein Palindrom ist, dann auch $a \cdot u \cdot a$.

1.3.6 Teilworte, Präfixe und Suffixe

Ein *Präfix* eines Wortes w ist ein Anfangsstück von w. Man schreibt dafür auch $u \preceq w$:

$$u \preceq w : \Longleftrightarrow \exists v. u \circ v = w.$$

Als induktive Definition geschrieben:

$$\varepsilon \preceq w : \Longleftrightarrow true$$
$$a \cdot u \preceq w : \Longleftrightarrow w = a \cdot v \wedge u \preceq v$$

Analog ist ist u ein *Suffix* von w und wir schreiben $w \succeq u$ falls ein $v \in \Sigma^*$ existiert mit $v \circ u = w$. Aus (1.3.15) folgt eine alternative Charakterisierung:

$$w \succeq u \iff u^R \preceq w^R.$$

s ist ein *Teilwort* von w, wenn es durch Abschneiden eines Präfix und eines Suffix aus w gewonnen werden kann, also:

$$s \; subword \; w : \Longleftrightarrow \exists u, v. \, w = u \circ s \circ v.$$

Auch hier kann man den Existenzquantor eliminieren und erhält eine rekursive Berechnungsvorschrift:

$$s \textit{ subword } w \;:\Longleftrightarrow\; s \preceq w \;\vee\; (w = a \cdot v \;\wedge\; s \textit{ subword } v).$$

Als *Teilfolge* (*subsequence*) von w bezeichnet man jedes Wort u, das durch Weglassen beliebiger Zeichen aus w entstehen kann:

$$s \textit{ subseq } w \;:\Longleftrightarrow\; s = \varepsilon \;\vee\; w = a \cdot u \;\wedge\; (s \textit{ subseq } u \vee (s = a \cdot v \wedge v \textit{ subseq } u)).$$

Für das Wort $kakao$ erhält man beispielsweise die

Präfixe:	$\{\varepsilon, k, ka, kak, kaka, kakao\}$,
Suffixe:	$\{\varepsilon, o, ao, kao, akao, kakao\}$,
Teilworte:	$\{\varepsilon, k, a, o, ka, ak, ao, kak, aka, kao, kaka, akao, kakao\}$.
Teilfolgen:	$\textit{Teilworte} \cup \{kk, aa, ko, kka, kko, kaa, aao, ako, kkao, kaao, kako\}$.

1.4 Sprachen

Eine Sprache besteht aus gewissen Worten über einem gemeinsamen Alphabet – und das ist auch schon die Definition:

Definition 1.4.1. Eine *Sprache L über Σ* ist eine Menge von Worten $L \subseteq \Sigma^*$.

Wir brauchen also nicht wählerisch zu sein, denn jede Teilmenge von Σ^* ist eine Sprache. Insbesondere auch:

Σ^*	die Menge aller Worte,
$\emptyset = \{\,\}$	die leere Menge,
$\{\varepsilon\}$	die Sprache, die nur das leere Wort enthält,
$\{a\}$	für jedes $a \in \Sigma$, die einelementige Menge bestehend aus dem Wort a der Länge 1.

Für ein konkretes Alphabet $\Sigma = \{a, b, c\}$ sind u.a. folgendes Sprachen:

$L_1 = \{abba, cbc, a, \varepsilon\}$
: eine Sprache mit 4 Worten, oder

$L_{a^n b^n} = \{a^n b^n \mid n \in \mathbb{N}\}$
: alle Worte die mit beliebig vielen a-s beginnen und mit gleich vielen b-s enden

$L_{\star c \star} = \{ucv \mid u, v \in \Sigma^*\}$
: alle Worte, in denen mindestens ein c vorkommt.

Praktisch relevanter sind z.B. über dem Alphabet $\Sigma = \{0, 1\}$ die Sprache L_{Bin} aller positiven ganzen Zahlen in Binärdarstellung:

$L_{Bin} = \{0\} \cup \{1w \mid w \in \Sigma^*\}$,

oder über dem Alphabet $\Sigma = \{0, \ldots, 9\}$ die Sprache aller positiven natürlichen Zahlen in Dezimaldarstellung:

$L_{\mathbb{N}} = \{0\} \cup \{xw \mid 0 \neq x \in \Sigma, w \in \Sigma^*\}$.
: Eine positive natürliche Zahl ist also entweder 0 oder eine Folge von Ziffern, die nicht mit 0 beginnt.

L_{py}
: die Menge aller (syntaktisch korrekten) Python Programme über dem Alphabet $\Sigma = ASCII$.

$L_{(,)}$
: die Menge aller wohlgeformten Klammerausdrücke, auch als *Dyck-Sprache* D_1 bekannt.

Die Dyck-Sprache über dem zwei-elementigen Alphabet $\Sigma = \{\,(\,,\,)\,\}$ ist umgangssprachlich gar nicht so einfach zu beschreiben. Man stelle sich einen arithmetischen Ausdruck vor, in dem alle Zeichen außer den Klammern wegradiert wurden. Somit gilt beispielsweise $(()(()())) \in L_{(,)}$ aber $())(()) \notin L_{(,)}$. Öffnende und schließende Klammern treten als Paare auf, wobei jede öffnende Klammer vor ihrer schließenden Partnerin erscheinen muss. Wir werden später sehen, dass diese Sprache sich sehr elegant mittels einer *kontextfreien Grammatik* definieren lässt. Die Dyck-Sprache D_2 erlaubt zusätzlich noch eckige Klammern ´[´ und ´]´, etc..

1.4.1 Exotische Sprachen

Wir beginnen mit der Zahl $\pi = 3.14159265358979323\ldots$ von der wir wissen, dass sie irrational ist und somit ihre Zifferndarstellung nie periodisch wird und erst recht nicht abbricht. Wenn wir den Dezimalpunkt ignorieren, haben wir eine unendliche Folge von Ziffern vor uns, die wir auch mit $\bar{\pi}$ bezeichnen, also

$$\bar{\pi} = 314159265358979323\ldots$$

Daraus bauen wir uns jetzt die folgenden Sprachen über dem Alphabet $\Sigma = \{0, 1, \ldots, 9\}$:

$L_{pre\pi} = \{u \mid u \preceq \bar{\pi}\} = \{\varepsilon, 3, 31, 314, 3141, 31415, \ldots\}$ und[1]
$L_{sub\pi} = \{u \mid u \text{ ist Teilwort von } \bar{\pi}\} = \{\varepsilon, 3, 1, 31, 4, 14, 314, 41, 5, 15, 415, \ldots\}$.

1 Die Definition von $u \preceq \bar{\pi}$ aus Abschnitt 1.3.6 funktioniert auch für „unendliche Worte“ wie $\bar{\pi}$.

Anhand dieser zugegebenermaßen exotischen Sprachen wollen wir einige grundsätzliche Probleme bei der Erkennung von Sprachen verdeutlichen.

1.5 Entscheidbarkeit – ein erster Blick

Wir fragen uns, ob ein Wort w zu einer der Sprachen $L_{pre\pi}$ oder $L_{sub\pi}$ gehört. Für die erste Sprache ist das ganz einfach zu beantworten. Da es einen Algorithmus gibt, welcher nacheinander die Ziffern von π ausdruckt, müssen wir nur die ersten $|w|$ Ziffern von $\bar{\pi}$ erzeugen und diese mit w vergleichen. Für die Sprache $L_{pre\pi}$ haben wir also einen Algorithmus,[2] der *entscheiden* kann, ob ein Wort w in der Sprache ist, oder nicht.

Für die zweite Sprache, $L_{sub\pi}$, ist das Problem schwieriger.

Falls w als Teilwort von $\bar{\pi}$ vorkommt, werden wir es nach endlicher Zeit entdecken, indem wir der Reihe nach die Ziffern von π erzeugen lassen. Wenn irgendwann die gesuchte Ziffernfolge auftaucht, melden wir Erfolg, andernfalls suchen wir weiter.

Falls w nicht als Teilwort vorkommt werden wir es auf diese Weise nie herausfinden – der Such-Algorithmus stoppt nie. Wir sagen daher, dass die Sprache $L_{sub\pi}$ zumindest *semi-entscheidbar* ist.

Es gibt die bisher noch nicht bewiesene mathematische Vermutung, dass jede denkbare endliche Ziffernfolge in $\bar{\pi}$ vorkommt. Falls diese Vermutung richtig ist, dann gilt natürlich $L_{sub\pi} = \Sigma^*$, und das Problem "$w \in L_{sub\pi}$" hat für alle w die triviale Antwort: *true*. Damit wäre also $L_{sub\pi}$ entscheidbar.

Im Internet gibt es zahlreiche Seiten, z.B. *mypiday.com* auf denen man sein Geburtsdatum in den Ziffern von π aufsuchen lassen kann. Das dortige Programm geht offensichtlich davon aus, dass jedes mögliche Datum in π auffindbar ist – die Geburtsdaten der Autoren wurden bei den Positionen 703779 und 50617 gefunden.

1.5.1 Entscheidbarkeit

Eine Sprache $L \subseteq \Sigma^*$ zerlegt die Menge aller Worte Σ^* in zwei Mengen, solche, die in der Sprache sind und solche die nicht zur Sprache gehören. Aber ist es immer so einfach festzustellen, ob ein Wort w zur Sprache L gehört? Die *charakteristische Funktion*

$$\chi_L : \Sigma^* \to Bool,$$

die einem Wort $w \in \Sigma^*$ den Wert $true$ (oder 1) zuordnet, wenn $w \in L$ ist und $false$ (oder 0) sonst, würden wir gerne als Algorithmus implementieren. Allgemein sprechen wir von einem Entscheidungsalgorithmus, wenn dieser für alle Worte $w \in \Sigma^*$ definiert ist, und als Ergebnis einen Wahrheitswert $true$ oder $false$ liefert. Insbesondere muss ein Entscheidungsalgorithmus für jede Eingabe terminieren.

2 Am 14.März 2019 (in amerikanischer Notation 3/14), dem π–day, gab Google bekannt, dass es gelungen sei, die ersten 31,4 Trillionen Dezimalziffern von π zu berechnen.

Es stellt sich heraus, dass nicht zu jeder Sprache $L \subseteq \Sigma^*$ ein Entscheidungsalgorithmus existieren kann. Es gibt einfach zu viele Teilmengen von Σ^* und damit zu viele mögliche Sprachen. Im Gegensatz dazu gibt es zu wenige mögliche Programme.

Wie kann das sein, wo es doch unendlich viele Programme, sogar unendlich viele Entscheidungsalgorithmen gibt? Um dies zu klären benötigen wir eine Definition:

Definition 1.5.1. Eine Teilmenge $L \subseteq \Sigma^*$ heißt *entscheidbar*, falls es einen Algorithmus $\mathcal{A}$ gibt, der als Input ein beliebiges Wort $w \in \Sigma^*$ erwartet und als Ausgabe $true$ erzeugt, falls $w \in L$ und $false$ sonst, also $L = \{w \in \Sigma^* \mid \mathcal{A}(w) = true\}$. Einen solchen Algorithmus nennen wir *Entscheidungsalgorithmus*.

1.5.2 Semi-Entscheidbarkeit

Eine Sprache $L \subseteq \Sigma^*$ heißt *semi-entscheidbar*, falls es einen Algorithmus $\mathcal{A}$ gibt, der Worte $w \in \Sigma^*$ entgegennimmt und genau dann die Ausgabe $true$ liefert, wenn $w \in L$ gilt. Falls $w \notin L$ ist, darf der Algorithmus $false$ liefern, er kann aber auch nie fertig werden, wie im Falle des Algorithmus für $L_{subπ}$.

Selbstverständlich ist jede entscheidbare Sprache auch semi-entscheidbar.

Ist L entscheidbar, dann ist auch ihr Komplement $\Sigma^* - L$ entscheidbar. Man muss die Ausgabe des Entscheidungsalgorithmus für L lediglich invertieren – aus $true$ mache $false$ und aus $false$ mache $true$.

Ist $L \subseteq \Sigma^*$ entscheidbar, so sind sowohl L als auch ihr Komplement $\Sigma^* - L$ semi-entscheidbar. Es gilt sogar die Umkehrung:

Satz 1.5.2. *Eine Sprache $L \subseteq \Sigma^*$ ist genau dann entscheidbar, wenn sowohl L als auch ihr Komplement $\Sigma^* - L$ semi-entscheidbar sind.*

Beweis. Angenommen wir haben sowohl einen Semi-Entscheidungs-Algorithmus $\mathcal{A}_L$ für L als auch einen $\mathcal{A}_{\Sigma^*-L}$ für ihr Komplement. Um zu entscheiden, ob ein Wort w in L ist, lassen wir beide Algorithmen mit Input w gleichzeitig laufen. Mindestens einer von beiden muss terminieren.

Terminiert $\mathcal{A}_L$, so liefert dessen Ergebnis die gewünschte Antwort $true$ bzw. $false$. Terminiert $\mathcal{A}_{\Sigma^*-L}$ so müssen wir dessen Ergebnis nur invertieren. □

1.5.3 Eine nicht-entscheidbare Sprache

Ohne näher festzulegen, wie ein Algorithmus aussehen darf und wie wir ihn beschreiben (z.B. als Python-Programm), wollen wir hier nur eine Eigenschaft betonen: Man muss den Algorithmus als Text hinschreiben können.

Ein Entscheidungsalgorithmus ist somit als Text beschrieben und er kann einen Text als Input entgegennehmen. Interessant wird es, wenn wir solche Algorithmen mit

ihrem eigenen Programmtext füttern. Wir demonstrieren dies mit unserer Funktion *length*, die wir zur Abwechslung als iteratives Python-Programm geschrieben haben:

Algorithmus 1.1 Aufruf einer Funktion mit eigenem Programmtext als Input

```
length("""def length(s):
       if s=='':
              return 0
       else:
              return 1+length(s[1:])""")
```

In diesem Falle erhielten wir als Wert des Aufrufs: 72. Nun ist bekannt, dass wir jeden Text sogar als Folge von 0-en und 1-en codieren können. Somit können wir auch jeden Algorithmus durch ein Wort $d \in \{0,1\}^*$ beschreiben. Selbstverständlich ist nicht jedes Wort $d \in \{0,1\}^*$ der Programmtext für einen Entscheidungsalgorithmus, uns interessieren ab jetzt aber nur solche Worte d, für die das der Fall ist.

Falls wir z.B. *Python* als unsere Beschreibungsmethode gewählt haben, interessieren uns nur Worte $d \in \{0,1\}^*$, die als Binärcode eines Python-Programms entstehen, welches einen Entscheidungsalgorithmus $\mathcal{E}$ codiert.

Sei also $L \subseteq \{0,1\}^*$ eine Sprache und $\mathcal{E}$ ein Entscheidungsalgorithmus für L. Für jedes Wort $w \in \{0,1\}^*$ gilt

$$w \in L \iff \mathcal{E}(w) = true. \tag{1.5.1}$$

Jetzt wenden wir einen Trick an, der auf den großen Mathematiker *Georg Cantor* zurückgeführt werden kann:

Wir fragen $\mathcal{E}$, ob sein eigener Binärcode $text(\mathcal{E})$ zu L gehört oder nicht, wir wenden also den Algorithmus auf seinen eigenen Code an:

$$\mathcal{E}(text(\mathcal{E})).$$

Da $\mathcal{E}$ ein Entscheidungsalgorithmus ist, liefert er entweder *true* oder *false*. Nicht genug damit, wir definieren eine Sprache:

$$L_D := \{text(\mathcal{E}) \in \{0,1\}^* \mid \mathcal{E} \text{ ist ein Entscheidungsalgorithmus und } \mathcal{E}(text(\mathcal{E})) = false\}. \tag{1.5.2}$$

Satz 1.5.3. *L_D ist nicht entscheidbar.*

Beweis. Angenommen, wir hätten einen Entscheidungsalgorithmus $\mathcal{A}$ für L_D, also

$$w \in L_D \iff \mathcal{A}(w) = true. \tag{1.5.3}$$

Wir fragen uns, ob $text(\mathcal{A}) \in L_D$ ist, oder nicht und finden

$$\begin{aligned} text(\mathcal{A}) \notin L_D &\overset{1.5.3}{\iff} \mathcal{A}(text(\mathcal{A})) = false \\ &\overset{1.5.2}{\iff} text(\mathcal{A}) \in L_D. \end{aligned}$$

Die Annahme, dass es einen Entscheidungsalgorithmus $\mathcal{A}$ gäbe ist also falsch, da sein Binärcode d die Absurdität $text(\mathcal{A}) \in L_D \iff text(\mathcal{A}) \notin L_D$ nach sich ziehen würde. □

Korollar 1.5.4. *Es gibt Sprachen $L \subseteq \Sigma^*$, die nicht entscheidbar sind.*

Wir haben der Einfachheit halber nur den Fall $\Sigma = \{0, 1\}$ betrachtet. Es ist allgemein bekannt, dass jede Information, die sich in einem Rechner speichern lässt, insbesondere auch jedes Programm als Folge von 0-en und 1-en und somit als Wort $w \in \Sigma^*$ darstellbar ist.

1.6 Operationen auf Sprachen

Definition 1.6.1. Aus vorhandenen Sprachen kann man neue Sprachen bauen. Dabei spielen die folgenden Operatoren eine wichtige Rolle. Seien dazu $L_1, L_2 \subseteq \Sigma^*$ gegeben. Wir definieren:

$$\begin{aligned} L_1 + L_2 &:= L_1 \cup L_2 \\ L_1 \circ L_2 &:= \{u \circ v \mid u \in L_1, v \in L_2\} \\ L^0 &:= \{\epsilon\} \\ L^{n+1} &:= L \circ L^n \\ L^* &:= \bigcup_{n \in \mathbb{N}} L^n \end{aligned}$$

Die Operation $(-)^*$ bezeichnet man auch als *Kleene-Stern*. Ein Wort w ist in L^*, wenn es eine natürliche Zahl n gibt und Worte $w_1, \ldots, w_n \in L$ so dass

$$w = w_1 w_2 \ldots w_n.$$

Für $n = 0$ ist insbesondere auch $\varepsilon \in L^*$.

Eine andere Beschreibung von L^* liefert das folgende Lemma:

Lemma 1.6.2. *L^* ist die kleinste Menge von M Worten, für die gilt:*

$$\varepsilon \in M \tag{1.6.1}$$

$$u \in L, v \in M \Rightarrow uv \in M. \tag{1.6.2}$$

Beweis. Wegen $L^0 \subseteq L^\star$ gilt natürlich $\varepsilon \in L^\star$. Sei $u \in L$ und $v \in L^\star = \bigcup_{n\in\mathbb{N}} L^n$, so gibt es ein $n \in \mathbb{N}$ mit $v \in L^n$. Damit ist $uv \in L \circ L^n = L^{n+1} \subseteq L^\star$. Folglich erfüllt $L^\star$ die Eigenschaften (1.6.1) und (1.6.2), umfasst also mindestens auch die kleinste Menge M mit (1.6.1) und (1.6.2), kurz: $M \subseteq L^\star$.

Sei nun M irgendeine Menge von Worten mit (1.6.1) und (1.6.2). Wir wollen zeigen, dass $M \supseteq L^\star$ gilt. Durch Induktion über n zeigen wir, dass $\forall n \in \mathbb{N}.L^n \subseteq M$ gilt, was sofort $L^\star = \bigcup_{n\in\mathbb{N}} L^n \subseteq M$ nach sich zieht. Für den Induktionsbeweis liefert (1.6.1) den Induktionsanfang und (1.6.2) den Induktionsschritt. □

Bei der Charakterisierung von $L^\star$ in Lemma 1.6.2 sticht sofort ins Auge, dass sie der Definition von $\Sigma^\star$ in (1.2.1) und (1.2.2) ähnelt, und in der Tat kann man $L^\star$ als Menge aller Worte über dem Alphabet L auffassen, wenn man L als potentiell unendliches Alphabet benutzt.

1.6.1 Gleichheiten

Seien L, R und S beliebige Sprachen über dem gleichen Alphabet. Dann gelten unter anderem folgende Gleichheiten:

$$\begin{aligned}(L+R)\circ S &= L\circ S + R\circ S\\ L\circ(R\circ L)^\star &= (L\circ R)^\star \circ L\\ (L+R)^\star &= (L^\star R^\star)^\star\\ L^0 + L\circ L^\star &= L^\star\end{aligned}$$

Wir weisen nur die erste Gleichheit nach, die anderen folgen analog:

Sei $w \in (L+R)\circ S$, dann gibt es $u \in L+R$ und $v \in S$ mit $w = u \circ v$, insbesondere $u \in L$ oder $u \in R$.

Ist $u \in L$ dann folgt $w = u \circ v \in L \circ S$, ist $u \in R$, dann folgt $w = u \circ v \in R \circ S$. In jedem Fall ist $w \in L\circ S + R\circ S$. Somit haben wir $(L+R)\circ S \subseteq L\circ S + R\circ S$ nachgewiesen.

Wegen $L \subseteq L+R$ sowie $R \subseteq L+R$ folgt

$$L\circ S + R\circ S \subseteq (L+R)\circ S + (L+R)\circ S \subseteq (L+R)\circ S.$$

1.6.2 Weitere Operatoren auf Sprachen

Bezeichnet man die leere Sprache mit $\mathbf{0}$ und die Sprache, die nur ε enthält, mit $\mathbf{1}$, also $\mathbf{0} := \{\}$ und $\mathbf{1} := \{\varepsilon\}$, so kann man die obigen Gleichungen suggestiv ergänzen um

$$\mathbf{0} \circ L = \mathbf{0} = L \circ \mathbf{0}$$

und

$$\mathbf{1} \circ L = L = L \circ \mathbf{1}.$$

Die Operatoren +, $\circ$ und * heißen auch *reguläre Operatoren* und spielen eine zentrale Rolle in der Theorie der formalen Sprachen. Den Operator $\circ$ schreibt man oft nicht aus, statt $L \circ M$ schreibt man dann kurz LM. Wir werden das später auch so machen.

Der Operator + wird in der praktischen Informatik und in vielen Anwendungsprogrammen auch durch den senkrechten Strich '|' dargestellt, man findet dann $L \mid M$ statt $L + M$. Das ist konsistent mit der Benutzung von '|' als '*oder*' in Programmiersprachen. Schließlich ist ein Wort w in $L + M$, wenn w in L ist oder in M.

Neben den Operatoren $'+'$, $'\circ'$ und * kann man noch weitere nützliche Operatoren definieren, nämlich den *Schnitt*, die *Differenz* und die *Ableitungen*:

Definition 1.6.3. Schnitt, Differenz und zusätzliche einstellige Operationen auf Sprachen sind folgendermaßen definiert:

$$\begin{aligned} L_1 \cap L_2 &:= \{w \in \Sigma^\star \mid w \in L_1 \wedge w \in L_2\} \\ L_1 - L_2 &:= \{w \in \Sigma^\star \mid w \in L_1 \wedge w \notin L_2\} \\ L^+ &:= L \circ L^\star \\ L^? &:= L + \{\varepsilon\} \\ L^R &:= \{w^R \mid w \in L\} \end{aligned}$$

sowie für jedes $a \in \Sigma$ die sogenannte *Ableitung* (engl.: *derivative*) *nach* a:

$$\partial_a(L) := \{u \in \Sigma^\star \mid a \cdot u \in L\}.$$

Für die Ableitung gelten Regeln, die vage an die Ableitung von Funktionen erinnern:

$$\begin{aligned} \partial_a(L_1 + L_2) &= \partial_a(L_1) + \partial_a(L_2) \\ \partial_a(L_1 \circ L_2) &= \begin{cases} \partial_a(L_1) \circ L_2 + \partial_a(L_2) & \text{falls } \varepsilon \in L_1 \\ \partial_a(L_1) \circ L_2 & \text{sonst} \end{cases} \\ \partial_a(L^\star) &= \partial_a(L) \circ L^\star \end{aligned}$$

Die letzte Gleichheit ergibt sich aus den Umformungen:

$$\begin{aligned} w \in \delta_a(L^\star) &\iff a \cdot w \in L^\star \\ &\iff a \cdot w \in (L^0 + L \circ L^\star) \\ &\iff a \cdot w \in L \circ L^\star \\ &\iff \exists u \in \Sigma^\star.\, \exists v \in L^\star.\, a \cdot u \in L \wedge u \circ v = w \\ &\iff \exists u \in \partial_a(L).\, \exists v \in L^\star.\, u \circ v = w \\ &\iff w \in \partial_a(L) \circ L^\star. \end{aligned}$$

Eine Sprache heißt *nullable*, falls sie das leere Wort enthält. Führen wir noch den Operator $\nu(L) := L \cap \mathbf{1}$ ein, dann gilt

$$\nu(L) = \begin{cases} \mathbf{1} & \varepsilon \in L \\ \mathbf{0} & \text{sonst} \end{cases}.$$

Mit Hilfe des *ν-Operators* lässt sich die obige Ableitungsregel für die Konkatenation eleganter formulieren:

$$\partial_a(L_1 \circ L_2) \quad = \quad \partial_a(L_1) \circ L_2 + \nu(L_1) \circ \partial_a(L_2).$$

Aufgabe 1.6.4. Zeigen Sie, dass für beliebige Sprachen $L, L_1, L_2 \subseteq \Sigma^*$ und $a \in \Sigma$ die folgenden Verträglichkeiten von ν mit den Operationen auf Sprachen gelten:

$\nu(L^*) = \mathbf{1} = \nu(\mathbf{1})$
$\nu(\{a\}) = \mathbf{0} = \nu(\mathbf{0})$
$\nu(L_1 + L_2) = \nu(L_1) + \nu(L_2)$
$\nu(L_1 \circ L_2) = \nu(L_1) \circ \nu(L_2)$.

Kapitel 2

Reguläre Sprachen

2.1 Reguläre Ausdrücke und reguläre Sprachen

Reguläre Ausdrücke dienen dazu, eine bestimmte Klasse einfacher Sprachen zu beschreiben. Diese Sprachen bezeichnet man als reguläre Sprachen. Sie haben eine besondere Bedeutung, da sie besonders einfach und effizient anhand einer Tabelle zu entscheiden sind. In der Praxis werden sie eingesetzt um die einfachsten Bestandteile einer Programmiersprache zu beschreiben, insbesondere die Schlüsselwörter, die Variablennamen, Zahl- und Stringkonstanten, etc..

Außerdem lassen sich reguläre Ausdrücke auch dazu verwenden, um effizient und gezielt nach bestimmten Textteilen in längeren Dokumenten zu suchen. Viele Programme, wie z.B. *OpenOffice*, *LibreOffice*, das Linux-Programm *grep*, viele Programmiereditoren, wie z.B. *notepad++, emacs* und *nano,* in beschränktem Umfang auch Microsoft *Word,* erlauben die Suche nach Textstücken im Gesamtdokument unter Zuhilfenahme regulärer Ausdrücke. Beispielsweise könnte man nach aktuellen Datumsangaben in einem Text suchen. Abbildung 2.1.1 zeigt, wie in *Notepad++* der reguläre Ausdruck

„[1 - 3] ? [0 - 9] [.] [0 - 1] ? [0 - 9] [.] (19 | 20) ? [0 - 9][0 -9][␣.]“

verwendet wird, um in einem Text nach Datumsangaben zwischen 1900 und 2099 zu suchen. Gefunden werden alle Daten im Format **„`tag.monat.jahr`“** wobei *tag* und *monat* ein- oder zweistellig sein dürfen, das *jahr* kann wahlweise mit zwei Ziffern oder mit 4 Ziffern notiert sein, im letzteren Fall muss es mit 19 oder mit 20 beginnen. Falls `tag` oder `monat` mit zwei Ziffern angegeben sind, darf die erste Ziffer höchstens 3 bzw. höchstens 1 sein. Das folgende Beispiel zeigt eine solche gezielte Suche in Notepad++. Der für die Suche verwendete reguläre Ausdruck endet mit einem Leerzeichen oder einem Punkt“.“, damit nicht versehentlich nach einem Datum aussehende Bestandteile

https://doi.org/10.1515/9783110442397-003

anderer Zahlenkolonnen, wie hier im Beispiel eine Linux-Versionsnummer, gefunden werden.

Neben den bereits bekannten Operatoren „**?**“, „**|**“, und Konkatenation, die wir, wie oben diskutiert, nicht ausschreiben, finden wir noch die eckigen Klammern „**[**“ und „**]**“, die für ein beliebiges Zeichen aus einem Bereich des Alphabets stehen. Dabei nutzt man aus, dass die meisten Alphabete, die in der Praxis vorkommen, eine natürliche Reihenfolge tragen. Beispielsweise steht **[0 – 3]** für ein beliebiges Zeichen aus der Menge $\{0, 1, 2, 3\}$. Auch Kombinationen von Bereichen sind erlaubt.

Beispielsweise steht **[0-9a-fA-F]** für eine beliebige Hexziffer und **[_a-zA-Z][_a-zA-Z0-9]*** für einen beliebigen Variablennamen, der mit einem beliebigen Buchstaben oder dem Unterstrich (underscore) „_“ beginnen darf und danach (wegen des „*“-Operators) beliebig viele Buchstaben, Ziffern oder Unterstriche benutzen darf und im Beispiel von Abbildung 2.1.1 steht „**[.]**“ für ein Leerzeichen oder einen Punkt.

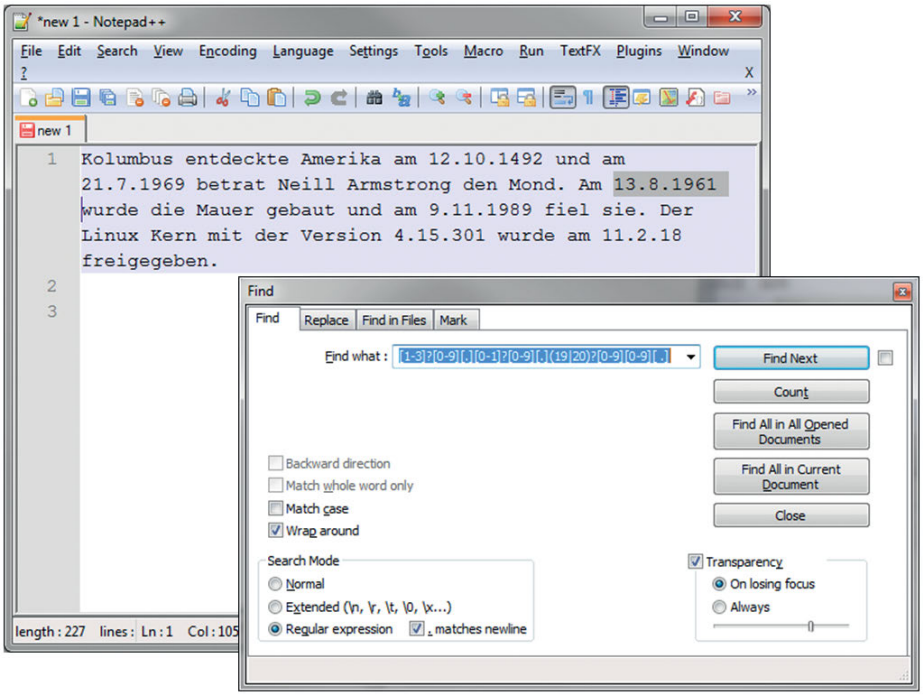


Abb. 2.1.1: Suche nach aktuellen Datumsangaben

Für die theoretische Diskussion kommen wir mit wenigen regulären Operatoren aus, mit deren Hilfe wir aus den Zeichen des Alphabets reguläre Ausdrücke bilden können. Diese regulären Ausdrücke sollen Sprachen beschreiben.

2.1.1 Reguläre Ausdrücke

Definition 2.1.1. Sei Σ ein Alphabet. *Reguläre Ausdrücke über* Σ sind dann:
- $\emptyset$, $\boldsymbol{\varepsilon}$, und $\boldsymbol{a}$ für jedes $a \in \Sigma$. [1]
- Sind $\boldsymbol{r}$ und $\boldsymbol{s}$ reguläre Ausdrücke, dann auch $\boldsymbol{rs}$, $\boldsymbol{r} + \boldsymbol{s}$, $\boldsymbol{r}^\star$ und $\boldsymbol{(r)}$.

Beispiele für reguläre Ausdrücke sind $\boldsymbol{a}$, $\boldsymbol{a} + \boldsymbol{b}$, $\boldsymbol{a(a + b)}$, $\boldsymbol{(ba^\star b^\star + \varepsilon)^\star}$.

2.1.2 Semantik der regulären Ausdrücke.

Reguläre Ausdrücke sollen Sprachen beschreiben. Dazu definiert man für jeden regulären Ausdruck r die zugehörige Sprache $[\![r]\!] \subseteq \Sigma^\star$.

Definition 2.1.2 (Semantik regulärer Ausdrücke)**.**

$$\begin{aligned}
[\![\emptyset]\!] &:= \{\} \\
[\![\varepsilon]\!] &:= \{\boldsymbol{\varepsilon}\} \\
[\![\boldsymbol{a}]\!] &:= \{a\} \text{ für jedes } a \in \Sigma \\
[\![r + s]\!] &:= [\![r]\!] \cup [\![s]\!] \\
[\![rs]\!] &:= [\![r]\!] \circ [\![s]\!] \\
[\![r^\star]\!] &:= [\![r]\!]^\star.
\end{aligned}$$

Diese Definition zeigt, wie man zu jedem regulären Ausdruck die zugehörige Sprache berechnen kann.

Beispiel 2.1.3. Dezimale Integerzahlen könnte man durch den folgenden regulären Ausdruck über $\Sigma = \{0, 1, \ldots, 9\}$ beschreiben:

$$\boldsymbol{0 + (1 + 2 + 4 + 5 + 6 + 7 + 8 + 9)(0 + 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9)^\star}$$

Die Bedeutung dieses Ausdrucks ist dann

$$\begin{aligned}
[\![\boldsymbol{0 + (1 + \ldots + 9)(0 + 1 + \ldots + 9)^\star}]\!] &= [\![\boldsymbol{0}]\!] \cup [\![\boldsymbol{(1 + \ldots + 9)(0 + 1 + \ldots + 9)^\star}]\!] \\
&= \{0\} \cup \{1, 2, \ldots, 9\} \circ \{0, 1, 2, \ldots, 9\}^\star
\end{aligned}$$

Hier zeigt sich, wie nützlich in der Praxis die zusätzlichen Notationen, wie etwa die Bereichsklammern [...] sind. Mit deren Hilfe könnte man den Ausdruck viel einfacher als $\boldsymbol{0 + [1 - 9][0 - 9]^\star}$ oder $\boldsymbol{0 | [1 - 9][0 - 9]^\star}$ notieren.

---

1 Falls zufällig $0, 1 \in \Sigma$ sind, so sollte man die regulären Ausdrücke **0** und **1** nicht verwechseln mit den Sprachen **0** und **1** wie sie in Abschnitt 1.6.2 verwendet wurden.

Für theoretische Untersuchungen sind zusätzliche Notationen oft nur Ballast, obwohl sie für praktische Zwecke äußerst hilfreich sind. Man kann sie einfach nur als Schreibabkürzungen ansehen, wie z.B.

$$\begin{array}{ll} \boldsymbol{r^+} & \text{für} \quad \boldsymbol{r(r)^\star} \\ \boldsymbol{r?} & \text{für} \quad \boldsymbol{(\varepsilon + r)} \\ \boldsymbol{r^n} & \text{für} \quad \boldsymbol{rr\ldots r} \ (n\text{-mal}). \end{array}$$

Hexzahlen kann man damit einfach durch den folgenden regulären Ausdruck spezifizieren:

$$\boldsymbol{0x\,[0-9a-fA-F]^2}.$$

2.1.3 Reguläre Sprachen

Definition 2.1.4. Eine Sprache L heißt *regulär*, falls es einen regulären Ausdruck r gibt mit $L = [\![r]\!]$.

Lemma 2.1.5. *Jede endliche Sprache ist regulär. Ist L regulär, dann sind auch L^+, $L?$ und L^R regulär.*

Beweis. Für jedes Wort $w \in \Sigma^\star$ ist zunächst die einelementige Sprache $\{w\}$ regulär, denn $w = a_1 a_2 \ldots a_n$ für gewisse $a_i \in \Sigma$ und damit folgt

$$\{w\} = \{a_1\} \circ \ldots \circ \{a_n\} = [\![\boldsymbol{a_1}]\!] \circ [\![\boldsymbol{a_2}]\!] \circ \ldots \circ [\![\boldsymbol{a_n}]\!] = [\![a_1 a_2 \ldots a_n]\!] = [\![w]\!].$$

Für eine endliche Menge $S = \{w_1, \ldots, w_k\} \subseteq \Sigma^\star$ ist dann auch

$$\{w_1, \ldots, w_k\} = [\![w_1]\!] \cup \ldots \cup [\![w_k]\!] = [\![w_1 + \ldots + w_k]\!].$$

Offensichtlich gelten $L^+ = L \circ L^\star$ und $L? = \{\varepsilon\} \cup L$. Ist daher L regulär, also $L = [\![r]\!]$, so sind $L^+ = [\![rr^\star]\!]$ und $L? = [\![\varepsilon + r]\!]$ auch regulär.

Für L^R benötigen wir einen Induktionsbeweis. Ist r ein regulärer Ausdruck, so konstruieren wir einen regulären Ausdruck r^R so dass $[\![r^R]\!] = [\![r]\!]^R$. Dies geht ganz einfach, denn $\emptyset^R := \emptyset$, $\varepsilon^R = \boldsymbol{\varepsilon}$ und $\boldsymbol{a}^R := \boldsymbol{a}$. Weiter sei $(r+s)^R = r^R + s^R$ jedoch $(r \circ s)^R = s^R \circ r^R$ und $(r^\star)^R = (r^R)^\star$. Es ist leicht nachzuprüfen, dass jeweils $[\![r^R]\!] = [\![r]\!]^R$ gilt. □

Wir werden später feststellen, dass mit L und S auch $L - S$ regulär ist, insbesondere auch das Komplement $\Sigma^\star - L$, und der Schnitt $L \cap S$. Allerdings ist dies ganz und gar nicht offensichtlich. Insbesondere ist es nicht möglich aus regulären Ausdrücken r für L und s für S einen regulären Ausdruck aufzubauen, dessen Semantik gerade $L - S$ wäre. Daher gelten " $-$ " und " $\cap$ " auch nicht als reguläre Operatoren, obwohl die regulären Sprachen unter diesen Operatoren abgeschlossen sind.

Aufgabe 2.1.6. Sei $\Sigma = \{a, b, c\}$. Geben Sie reguläre Ausdrücke r, s und t an, so dass
$[\![r]\!] = \Sigma^*$
$[\![s]\!] = \Sigma^* - \{a\}$
$[\![t]\!] = \{w \in \Sigma^* \mid aw \in \Sigma^*\{b\}\Sigma^*\}$, alle Worte, die ein b enthalten.

Definition 2.1.7 (Reguläre Gleichungen)**.** Für zwei reguläre Ausdrücke r und s drückt die *reguläre Gleichung* "$r = s$" aus, dass $[\![r]\!] = [\![s]\!]$.

Wir gewinnen leicht folgende reguläre Gleichungen (r, s und t stehen für beliebige reguläre Ausdrücke):

$$\begin{aligned} r + (s + t) &= (r + s) + t \\ r + s &= s + r \\ r + \emptyset &= r \\ r + r &= r \end{aligned}$$

$$\begin{aligned} r(st) &= (rs)t \\ \varepsilon r &= r \\ r\varepsilon &= r \\ r(s + t) &= rs + rt \\ (r + s)t &= rt + st \end{aligned}$$

$$\begin{aligned} \emptyset r &= \emptyset \\ r\emptyset &= \emptyset \\ rr^* &= r^* r \\ \varepsilon + rr^* &= r^* \end{aligned}$$

Man sieht deutlich, dass bezüglich der „Addition" und der „Multiplikation" die Sprachen $\emptyset$ und ε sich wie die arithmetischen Verwandten 0 und 1 verhalten. Daher findet man in der Literatur oft auch die Bezeichnungen $\mathbf{0}$ und $\mathbf{1}$ statt $\emptyset$ und ε. Damit kann man einige der obigen Gleichungen suggestiver formulieren als:

$$\mathbf{1}r = r = r\mathbf{1},\ \mathbf{0}r = \mathbf{0} = r\mathbf{0},\ \mathbf{1} + rr^* = r^*.$$

Nur falls die Ziffern 0 oder 1 zum Alphabet Σ gehören, begegnet man der Verwechslungsgefahr, indem man die Zeichen $\emptyset$ und ε verwendet.

Aufgabe 2.1.8. Wir nehmen an, dass die Zeichen 0 und 1 nicht zu Σ gehören. Zeigen Sie, dass für beliebige Sprachen $L, L_1, L_2 \subseteq \Sigma^*$ und $a \in \Sigma$ die folgenden Ableitungsregeln (siehe Def 1.6.3) gelten und folgern Sie:

Die Ableitung einer regulären Sprache ist wieder eine reguläre Sprache.:

1. $\partial_a(\boldsymbol{a}) = \mathbf{1}$ und $\partial_a(\boldsymbol{b}) = \mathbf{0}$, falls $a, b \in \Sigma$ und $a \neq b$.
2. $\partial_a(L_1 + L_2) = \partial_a L_1 + \partial_a L_2$
3. $\partial_a(L_1 \circ L_2) = (\partial_a L_1) \circ L_2 + \nu(L_1) \circ \partial_a L_2$
4. $\partial_a(L^\star) = (\partial_a L) \circ L^\star$.

2.2 Maschinen und Automaten

Automaten sind gedachte Maschinen, deren Aufgabe es ist, einen Text zu lesen und und zu entscheiden, ob das gelesene Wort zu einer bestimmten Sprache gehört, oder nicht. Wir werden eine Reihe von Automatenmodellen kennenlernen: Deterministische endliche Automaten, Nichtdeterministische Automaten, Stackautomaten (Kellerautomaten), und sogenannte Turing-Maschinen. Allen gemeinsam ist, dass sie einen Input zeichenweise aus einer Datei lesen und je nach gelesenem Input und internem Zustand in einen neuen Zustand übergehen können. Stackautomaten können zusätzlich noch Informationen auf einem Stack speichern und Turing-Maschinen können den Inhalt der Eingabedatei verändern.

Diese Automatenmodelle wurden geschaffen bevor es real existierende Rechner gab. Die Turing-Maschine etwa, wurde von Alan Turing 1936 als mathematisches Gedankenmodell für einen Universalcomputer vorgestellt, ein halbes Jahrzehnt, bevor der erste Universalcomputer das Licht der Welt entdeckte. Daher spricht man heute noch von einem „Eingabeband“ von dem der Automat mit einem „Lesekopf“ seinen Input liest bzw. von einem „Schreib-Leseband“, auf dem er mit seinem „Schreib-Lesekopf“ lesen und schreiben kann, obwohl uns heute die Begriffe „Eingabedatei“ mit „Dateiposition“ bzw. „Editor“ mit „Cursorposition“ viel vertrauter sind.

Außerdem spricht man von „Automaten“ bzw. von „Turing-Maschinen“, wenn man eigentlich das Programm meint, was die Maschinen steuert. In diesem Kapitel werden wir zunächst sogenannte „Endliche Automaten“ betrachten und zeigen, dass diese genau das richtige Konzept liefern, um regulären Sprachen erkennen.

2.2.1 Endliche Automaten

Wir stellen uns einen Automaten als eine *black box* vor, die von einem Eingabeband zeichenweise einen Text liest. Je nach Zustand q und je nach gelesenem Zeichen a geht er in einen (neuen) Zustand $\delta(q, a)$ über. Ein bestimmter Zustand q_0 ist ausgezeichnet als *Startzustand* und gewisse Zustände $F \subseteq Q$ können als *akzeptierende Zustände* ausgezeichnet sein.

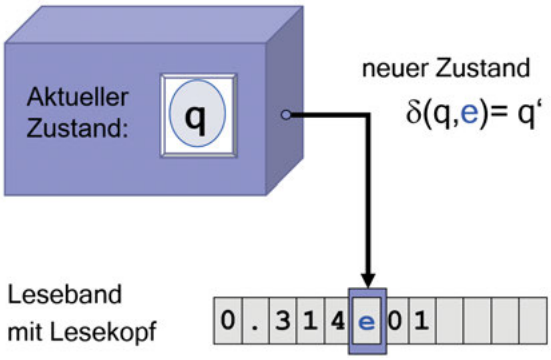


Abb. 2.2.1: Automat beim Erkennen einer Dezimalzahl

Als Beispiel stellen wir uns einen Automaten vor, der wie in Abbildung 2.2.1 angedeutet, erkennen soll, ob die Eingabe eine syntaktisch korrekte Dezimalzahl ist, wie z. B. 42, 31.415, 0.007, oder 9.46e12 in wissenschaftlicher Notation. Als Alphabet benötigen wir (mindestens) $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, \ ., e, E\}$. Während der Erkennung kann der Automat jeweils in einem von endlich vielen Zuständen Q sein. Intuitiv beschreiben Zustände den Fortschritt bei der Erkennung eines Inputs. Im Falle des Automaten für Dezimalzahlen könnten die folgenden Zustände relevant sein:

q_0 : noch nichts eingelesen, Anfangszustand,
q_1 : ein oder mehrere Ziffern eingelesen,
q_2 : Dezimalpunkt eingelesen,
q_3 : eine oder mehrere Ziffern nach dem Dezimalpunkt gesehen,
q_4 : eine Zahl mit Dezimalpunkt gesehen und dann ein „e“ oder ein „E“,
q_5 : erste Ziffer des Exponenten gesehen,
q_6 : zweite Ziffer des Exponenten gesehen,
q_7 : ein Fehler ist aufgetreten.

Die Übergänge von einem Zustand in einen anderen sind vom Input abhängig. In Zustand q_1 wissen wir, dass zumindest eine Ziffer schon eingelesen wurde. Jetzt darf also entweder eine weitere Ziffer kommen oder ein Dezimalpunkt.

Im ersten Fall verbleibt der Automat im gleichen Zustand, im zweiten geht er in Zustand q_2 über.

Falls in Zustand q_1 ein „e“ oder ein „E“ eingelesen wird, geht der Automat in den Fehlerzustand q_7. Mathematisch bedeutet das für unsere Zustandsübergangsfunktion δ, dass

$$\delta(q_1, 0) = \delta(q_1, 1) = \ldots = \delta(q_1, 9) = q_1, \delta(q_1, \text{“.”}) = q_2$$

und

$$\delta(q_1, e) = \delta(q_1, E) = q_7.$$

Auf diese Weise können wir eine Tabelle aufstellen, die alle erlaubten Zustandsübergänge beschreibt. Alle in Fig. 2.1 freigelassenen Kästchen sollen implizit den Eintrag q_7 enthalten.

	0	1	2	3	4	5	6	7	8	9	.	e	E
q_0	q_1	q_1	q_1	q_1	q_1	q_1	q_1	q_1	q_1	q_1			
$\underline{q_1}$	q_1	q_1	q_1	q_1	q_1	q_1	q_1	q_1	q_1	q_1	q_2		
q_2	q_3	q_3	q_3	q_3	q_3	q_3	q_3	q_3	q_3	q_3			
$\underline{q_3}$	q_3	q_3	q_3	q_3	q_3	q_3	q_3	q_3	q_3	q_3		q_4	q_4
q_4	q_5	q_5	q_5	q_5	q_5	q_5	q_5	q_5	q_5	q_5			
q_5	q_6	q_6	q_6	q_6	q_6	q_6	q_6	q_6	q_6	q_6			
$\underline{q_6}$													
q_7													

Tab. 2.1: Zustandsübergangstabelle für den Dezimalzahlautomaten

Die Tabelle beschreibt offensichtlich eine Abbildung

$$\delta : Q \times \Sigma \to Q,$$

wobei $\delta(q, a)$ der Eintrag ist, welcher sich in Zeile q und Spalte a befindet. Umgekehrt, kann man jede Abbildung $\delta : Q \times M \to Q$ als $|Q| \times |\Sigma|$-Tabelle schreiben: Die Elemente von Q nummerieren die Zeilen durch, die Elemente von Σ die Spalten. Der Eintrag im Schnittpunkt der q-ten Zeile mit der a-ten Spalte ist $\delta(q, a)$.

Als Startzustand vereinbaren wir den Zustand der ersten Tabellenzeile, q_0.

Jetzt müssen wir noch bestimmen, wann ein Input erkannt ist, oder nicht. Wenn der gesamte Input verarbeitet ist, und wir befinden uns in Zustand q_1, dann haben wir eine Integerzahl ohne Dezimalpunkt und ohne Exponent gesehen. Das ist nach unserer Vorgabe legal, wie im Beispiel der Zahl 42.

In Zustand q_3 haben wir eine Dezimalzahl gesehen wie 31.415 oder 0.007. Auch das ist eine korrekte Dezimalzahl. Schließlich haben wir in Zustand q_6 eine Dezimalzahl mit Exponenten gesehen.

Ist also unser Input aufgebraucht und befinden wir uns in einem der Zustände $\{q_1, q_3, q_6\}$, so haben wir erfolgreich eine Dezimalzahl erkannt. Daher nennen wir diese Zustände *akzeptierend* - andere Bezeichnungen sind *terminale Zustände* oder *Endzustände*. Die entsprechenden Tabellenzeilen haben wir durch Unterstreichung markiert.

Alle anderen Zustände markieren einen Zustand, in dem das gesuchte Wort noch nicht vollständig ist $\{q_0, q_2, q_4, q_5\}$ oder einen Fehlerzustand $\{q_7\}$.

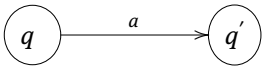
Als Beispiel verfolgen wir das Erkennen der Zahl $0.314e01$ wie in Figur 2.2.1. Wir beginnen im Zustand q_0. Vom Input lesen wir die Ziffer '0', daher entnehmen wir unsern nächsten Zustand aus Zeile q_0 und Spalte '0' der Tabelle als $\delta(q_0, 0) = q_1$. Unser

neuer Zustand q_1 ist bereits akzeptierend, aber das kümmert uns nicht, weil der Input noch nicht zu Ende ist.

Wir lesen als nächstes Zeichen ein '.' und finden als neuen Zustand in Zeile q_1 und Spalte '.' den Wert $\delta(q_1, '.') = q_2$. Dies ist kein akzeptierender Zustand, aber zum Glück ist unserer Input noch nicht komplett eingelesen. Wäre es anders, hätten wir also nur den Input '0.' vor uns, so würden wir diesen insgesamt nicht akzeptieren. Unser Automat akzeptiert also '0', '0.3' , '0.31', '0.314e01' etc., nicht aber '0.' oder '.3' oder '0.314e0'.

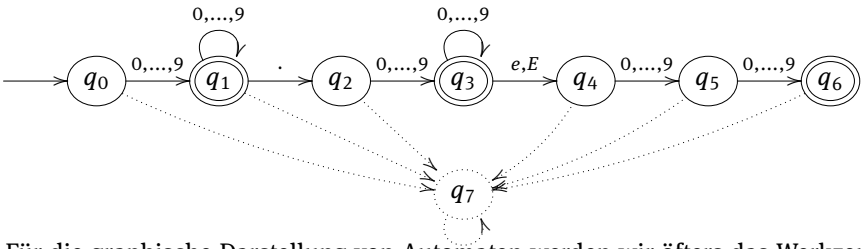
2.2.2 Graphische Darstellung von Automaten.

Automaten sind endliche Gebilde, sie lassen sich daher auch leicht in verschiedenen graphische Notationen darstellen. Eine beliebte Darstellung ist die des Transitionsgraphen. Zustände markiert man durch kleine Kreise, die den Namen des Zustands tragen. Wenn immer $\delta(q, a) = q'$ gilt, ziehen wir einen Pfeil von Zustand q zu Zustand q' und beschriften diesen mit a.



Gibt es mehrere Zeichen a, b, die von Zustand q zu Zustand q' führen, also mit $\delta(q, a) = \delta(q, b) = q'$, so schreiben wir alle diese Zeichen an den einen Pfeil von q nach q'. Den Startzustand markiert man als Ziel eines Pfeils der bei keinem Zustand beginnt und Endzustände werden doppelt eingekreist.

Eigentlich muss aus jedem Zustand $q \in Q$ für jedes Zeichen $a \in \Sigma$ ein Pfeil herausführen, der mit a beschriftet ist. In der Praxis stellt man häufig auch partielle Automaten dar. Den Fehlerzustand, in den alle fehlenden Pfeile hineinführen, muss sich der Benutzer dazu denken. In dieser Darstellung wird aus unserer obigen Tabelle die folgende Graphdarstellung, wobei wir den Fehlerzustand samt aller zugehörigen Pfeile gepunktet dargestellt haben.



Für die graphische Darstellung von Automaten werden wir öfters das Werkzeug JFLAP benutzen, welches auf der Seite *www.jflap.org* kostenlos angeboten wird. Man kann sich Automaten zusammenklicken, und diese in vielfältiger Weise bearbeiten und analysieren.

Als Beispiel wollen wir einen Automaten $\mathcal{A}$ mit Alphabet $\Sigma = \{0, 1\}$ entwerfen, der überprüft, ob eine Binärzahl durch 5 teilbar ist, mit anderen Worten soll

$$L(\mathcal{A}) = \{w \in \Sigma^* \mid (w)_2 \; mod \; 5 \; = 0\}$$

sein. Lesen wir Binärzahlen von links nach rechts ein, so bedeutet das Anhängen einer „0“ nichts anderes als Multiplikation mit 2 und Anhängen einer „1“ macht aus einer Binärzahl mit Zahlenwert n eine Binärzahl mit Zahlenwert $2n + 1$. War n vorher durch 5 teilbar, also $n = 5k$, dann lässt $2n + 1$ beim Teilen durch 5 den Rest 1, denn $2n + 1 = 2(5k)+1 = 10k+1$. Erneutes Anhängen einer „1“ liefert eine Zahl mit Wert $2(2n+1)+1 = 20k + 3$.

Dies legt nahe, die Teilerreste modulo 5 durch die Zustände des gesuchten Automaten zu repräsentieren. Wir betrachten also 5 Zustande $q_0, \ldots, q_4$ wobei Zustand q_i ausdrücken soll, dass die bisher eingelesenen Binärziffern beim Teilen durch 5 den Rest i ergeben. In Zustand q_i haben wir also $n = 5k + i$ gesehen.

Folgt eine 0, so ist die gesehene Zahl $2n = 2(5k + i) = 10k + 2i$, der neue Zustand muss also $q_{2i \, mod \, 5}$ sein. Das ergibt $\delta(q_0, 0) = q_0$, $\delta(q_1, 0) = q_2$, ... $\delta(q_4, 0) = q_3$. Analog erhalten wir $\delta(q_i, 1) = q_{(2i+1) \, mod \, 5}$, also z.B. $\delta(q_0, 1) = q_1, \delta(q_1, 1) = q_3$, etc..

In JFLAP klicken wir uns den Automaten zusammen und ziehen die Knoten des Graphen in eine ästhetisch angenehme Position.

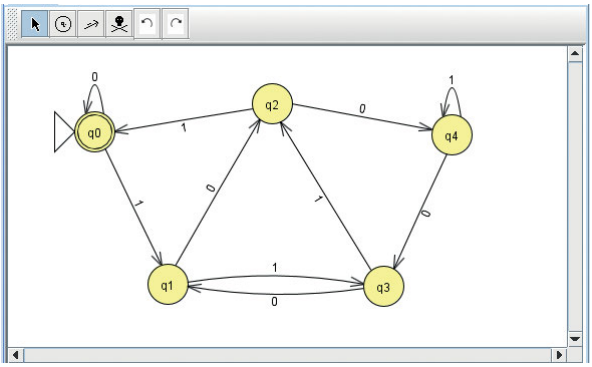


Abb. 2.2.2: Automat in JFLAP

Der Anfangszustand wird in JFLAP durch ein Dreieck markiert. Unser Automat ließe sich noch verbessern, denn da der Anfangszustand schon akzeptierend ist, akzeptiert er auch das leere Wort ε. Durch Hinzufügen eines neuen Anfangszustandes lässt sich das leicht korrigieren.

Aufgabe 2.2.1. Zeigen Sie, wie man aus einem Automaten $\mathcal{A}$ mit Sprache $L(\mathcal{A})$ einen neuen Automaten $\mathcal{A}'$ gewinnen kann mit $L(\mathcal{A}') = L(\mathcal{A}') - \{\varepsilon\}$.

2.2.3 Worte als Fahrpläne

Die graphische Darstellung von Automaten ist besonders intuitiv. Wir können sie lesen wie eine Landkarte und dabei das Eingabewort w als Fahrplan benutzen. In einem beliebigen Zustand q zeigt uns das nächste Eingabezeichen, wohin wir abbiegen sollen. Jedes Wort ist ein gültiger Fahrplan, da für jeden Zustand q und für jedes Zeichen $a \in \Sigma$ der nächste Zustand $\delta(q, a)$ definiert ist. Haben wir den Fahrplan abgearbeitet und sind wir in einem akzeptierenden Zustand gelandet, so ist unser Ziel erreicht. Die dabei besuchten Zustände nennt man auch einen *Lauf*.

2.2.4 Formale Definition

Insgesamt codieren sowohl die obige Tabelle mit den unterstrichenen Zeilen als auch die Zustandsübergangsgraphen mit den besonders gekennzeichneten Anfangs- und Endzuständen folgende Bestimmungsstücke:

- ein Alphabet Σ
- eine endliche Menge Q (von Zuständen)
- eine Übergangsfunktion $\delta : Q \times \Sigma \to Q$
- einen Startzustand $q_0 \in Q$
- eine Menge von akzeptierenden Zuständen $F \subseteq Q$.

Damit haben wir auch schon alle Bestandteile für eine mathematische Definition beisammen:

Definition 2.2.2 (Endlicher Automat)**.** Ein endlicher Automat ist ein 5-Tupel $\mathcal{A} = (\Sigma, Q, \delta, q_0, F)$ wobei Σ ein Alphabet ist Q eine endliche Menge, $\delta : Q \times \Sigma \to Q$, $q_0 \in Q$ und $F \subseteq Q$.

Die Elemente von Q nennt man *Zustände*, die Elemente von F *akzeptierende Zustände*, q_0 heißt Startzustand und δ heißt *Übergangsfunktion* oder *Transitionsfunktion*.

Um zu testen, ob ein Wort $w \in \Sigma^*$ von dem Automaten *akzeptiert* wird, schreibt man es auf das Eingabeband, setzt den Lesekopf auf das erste Zeichen des Wortes und initialisiert den aktuellen Zustand mit dem Startzustand q_0. Solange das Wort noch nicht komplett eingelesen ist, liest man das nächste Zeichen a und geht vom aktuellen Zustand q in den neuen Zustand $\delta(q, a)$ über und setzt den Lesekopf eine Position weiter. Ist das Wort fertig eingelesen und ist der Automat in einem akzeptierenden Zustand $q \in F$, dann wird das Wort akzeptiert, man sagt auch „*erkannt*“, ansonsten nicht.

Die englische Bezeichnung für „*endlicher Automat*“ heißt „*deterministic finite automaton*“ und wird gern mit *DFA* abgekürzt. Wir werden später auch noch „*nondeterministic finite automata*“ (*NFA*) kennenlernen.

Statt von akzeptierenden Zuständen spricht man oft auch von *Endzuständen*, *terminalen* oder *finalen* Zuständen. Diese Terminologie führt manchmal zu dem Missver-

ständnis, dass ein Automat stoppen würde, wenn ein Endzustand bzw. ein terminaler erreicht ist. Dem ist aber nicht so - der Automat stoppt erst, wenn das Inputwort aufgebraucht ist. Das wird an folgender algorithmischer Beschreibung deutlich:

Algorithmus 2.1 Arbeitsweise eines Automaten

- Setze $q = q_0$
- Solange noch ein Zeichen des Wortes vorhanden ist,
  - lese das aktuelle Zeichen a,
  - setze $q = \delta(q, a)$
  - bewege den Lesekopf eine Position weiter
- falls $q \in F$ wird das Wort akzeptiert, sonst nicht.

Die Übergangsfunktion δ dehnen wir aus zu einer Funktion $\delta^* : Q \times \Sigma^* \to Q$, welche statt Zeichen $a \in \Sigma$ ganze Worte $w \in \Sigma^*$ entgegennimmt und den resultierenden Zustand berechnet. Wir definieren δ^* induktiv über den Aufbau von Worten $w \in \Sigma^*$:

$$\delta^*(q, \varepsilon) = q \tag{2.2.1}$$

$$\delta^*(q, a \cdot u) = \delta^*(\delta(q, a), u) \tag{2.2.2}$$

Das leere Wort belässt uns in dem aktuellen Zustand, und wenn das Wort $w = au$ mit dem Zeichen a beginnt, gehen wir zunächst in den Zustand $\delta(q, a)$ und lesen von dort aus das Restwort u ein.

Die Sprache eines Automaten ist die Menge aller Worte, die von ihm erkannt werden. Mit Hilfe unserer Funktion δ^* können wir dies besonders einfach ausdrücken:

Definition 2.2.3. [Sprache eines Automaten] Sei $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ ein Automat. Die *Sprache von* $\mathcal{A}$ ist $L(\mathcal{A}) := \{w \in \Sigma^* \mid \delta^*(q_0, w) \in F\}$. Eine Sprache L heißt *regulär*[2], falls es einen Automaten $\mathcal{A}$ gibt mit $L = L(\mathcal{A})$. Man sagt, dass $\mathcal{A}$ die Sprache L *erkennt*.

Die Funktion δ^* arbeitet schrittweise das Wort in ihrem zweiten Argument ab. Insbesondere kann man für beliebige Worte $u, v \in \Sigma^*$ und beliebige $q \in Q$ durch Induktion über u leicht zeigen, dass für beliebige Worte $u, v \in \Sigma^*$ gilt:

$$\delta^*(q, u \circ v) = \delta^*(\delta^*(q, u), v). \tag{2.2.3}$$

Ist insbesondere $w = u \cdot a$ mit $a \in \Sigma$ und $u, w \in \Sigma^*$, dann folgt aus 2.2.3 und der Definition 1.3.12, dass man 2.2.2 ersetzen könnte durch

$$\delta^*(q, u \cdot a) = \delta(\delta^*(q, u), a). \tag{2.2.4}$$

2 Wir hatten den Begriff „reguläre Sprache“ schon früher im Zusammenhang mit regulären Ausdrücken eingeführt. Wir werden später erkennen, dass beide Begriffe übereinstimmen.

Der Begriff *regulär* deutet schon an, dass reguläre Sprachen mit regulären Ausdrücken beschrieben werden können. Dies ist ein Resultat, das wir in Abschnitt 2.9 erzielen werden.

2.2.5 Implementierung

Für manche Leser mögen sich die obigen Definitionen etwas trocken anfühlen. Sie lassen sich aber unmittelbar mit Leben füllen, wenn wir sie in ein objektorientiertes Programm einer modernen Programmiersprache wie z.B. *Scala* umsetzen. In Abbildung 2.2.3 sehen wir die entsprechende Scala-Klasse `Automaton`, parametrisiert über die Typvariablen `State` und `Alpha` welche in der mathematischen Definition einfach Q und Σ heißen. `init` bezeichnet den Anfangszustand q_0, `delta` die Übergangsfunktion δ und `isTerminal` prüft, ob ein Zustand terminal (akzeptierend) ist, oder nicht.

```
// Deterministischer Automat in Scala

class Automaton[State,Alpha](

  init        : State,

  delta       : (State,Alpha)=> State,

  isTerminal  : State => Boolean
  ){
  type Word = List[Alpha]

  def deltaStar(s:State,w:Word):State =
    w match {
      case Nil     => s
      case a::v    => deltaStar(delta(s,a),v)
    }

  def accepts(w:Word):Boolean
      = isTerminal(deltaStar(init,w))
 }
```

Abb. 2.2.3: Endlicher Automat in Scala

Den Typ `Word` als Abkürzung für Listen von Zeichen führen wir nur der besseren Lesbarkeit halber ein.

Die Methode `deltaStar` implementiert die Übergangsfunktion δ^*. Die Fähigkeit von Scala, Funktionen mit pattern-matching zu definieren erlaubt die unmittelbare Umsetzung der definierenden Gleichungen (2.2.1) und (2.2.2). Die Methode `accepts` prüft, ob ein Wort w akzeptiert wird, also zur Sprache des Automaten gehört, oder nicht.

Im folgenden `main()`-Programm instanziieren wir die Klasse `Automaton` mit dem Automaten aus Abbildung 2.2.4, der diejenigen Binärzahlen erkennen soll, welche durch drei teilbar sind. Wir benötigen als Zustände $Q = \{0, 1, 2\}$ und als Alphabet $\Sigma = \{0, 1\}$.

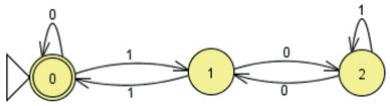


Abb. 2.2.4: Automat für die Sprache aller durch 3 teilbaren Binärzahlen

Der Bequemlichkeit halber wählen wir sowohl für `State` als auch für `Alpha` den Typ `Int`.

`init` ist 0 und `isTerminal` codieren wir als boolesche Funktion $x \mapsto (x == 0)$. Dies entspricht der Menge $\{0\}$ als Menge der terminalen Zustände.

Die Übergangsfunktion `delta:(Int,Int)==>Int` können wir auch als *map* explizit hinschreiben, der letzte Fall „`case(_,x)`“ erfasst alle restlichen Zustände und gibt für Input x den Zustand x aus. Hier ist nur relevant, dass $(0, \mathbf{0}) \mapsto \mathbf{0}$ und $(0, \mathbf{1}) \mapsto \mathbf{1}$ gelten soll.

```
// Jetzt ein Testlauf
 val binMod3 = new Automaton[Int,Int](

   init = 0,

   delta = {
     case (1,0) => 2
     case (1,1) => 0
     case (2,0) => 1
     case (2,1) => 2
     case (_,x) => x
     },

   isTerminal = { x => (x==0) }
   )

 def test(a:Automaton[_,Int]) =
   for(i <- 0 to 1; j <- 0 to 1; k <- 0 to 1)
     println("Wort "+i+j+k+" in L(A) : "
               + a.accepts(List(i,j,k)))

 test(binMod3)
```

Abb. 2.2.5: Erkennen aller durch 3 teilbaren Binärzahlen

Die `test`-Routine prüft, welche Bitfolgen der Länge 3 zur Sprache gehören.

```
Wort 000 in L(A) : true
Wort 001 in L(A) : false
Wort 010 in L(A) : false
Wort 011 in L(A) : true
Wort 100 in L(A) : false
Wort 101 in L(A) : false
Wort 110 in L(A) : true
Wort 111 in L(A) : false
```

Abb. 2.2.6: Ausgabe

2.2.6 Partielle Automaten.

In der Literatur findet man gelegentlich auch eine leicht unterschiedliche Definition von Automaten, wobei δ nur als partielle Funktion vorausgesetzt wird. Dies bedeutet, dass der Definitionsbereich von δ nicht ganz $Q \times \Sigma$ ist, sondern eine Teilmenge $D \subseteq Q \times \Sigma$. Wir werden diese Variante von Automaten *partielle Automaten* nennen.

Wenn wir aus unserer Tabelle 2.1 die letzte Zeile weglassen und als Definitionsbereich D alle Positionen (q, e) aus der Tabelle wählen, an denen ein Eintrag steht, haben wir einen partiellen Automaten vor uns.

Aus einem partiellen Automaten mit $D \subset Q \times \Sigma$ und

$$\delta : D \to Q \text{ mit } D \subseteq Q \times \Sigma$$

können wir immer einen richtigen Automaten machen, indem wir einen neuen Zustand q_{Err} als *Errorzustand* hinzufügen. Wir definieren die Übergangsfunktion

$$\delta' : (Q \cup \{q_{Err}\}) \times \Sigma \to Q \cup \{q_{Err}\}$$

durch

$$\delta'(q, e) = \text{if } (q, e) \in D \text{ then } \delta(q, e) \text{ else } q_{Err}.$$

Unser Automat in Tabelle 2.1 ist auf genau diese Weise entstanden.

Aufgabe 2.2.4. Unser Automat zur Erkennung von Dezimalzahlen erlaubt auch Zahlen, die mit mehreren Nullen beginnen, wie etwa „007“ oder „00.7”. Modifizieren Sie den Automaten, so dass dies nicht mehr erlaubt ist.

2.3 Konstruktionen mit Automaten.

2.3.1 Der Komplementautomat

Ist $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ ein Automat, der die Sprache $L \subseteq \Sigma^*$ akzeptiert, so lässt sich ganz einfach auch ein Automat $\mathcal{A}^c$ gewinnen, der $(\Sigma^* - L)$, das Komplement von L akzeptiert: Man tauscht einfach akzeptierende gegen nicht akzeptierende Zustände aus und behält alles andere bei, also $\mathcal{A}^c = (Q, \Sigma, \delta, q_0, (Q - F))$. Offensichtlich gilt

$$\begin{aligned} w \in L(\mathcal{A}^c) &\iff \delta^*(q_0, w) \in (Q - F) \\ &\iff \delta^*(q_0, w) \notin F \\ &\iff w \notin L \\ &\iff w \in (\Sigma^* - L). \end{aligned}$$

Ist also L regulär (von einem endl. det. Automaten erkennbar), dann auch $\Sigma^* - L$.

2.3.2 Produktkonstruktionen

Seien $\mathcal{A}_1 = (Q_1, \Sigma, \delta_1, q_0^1, F_1)$ und $\mathcal{A}_2 = (Q_2, \Sigma, \delta_2, q_0^2, F_2)$ Automaten über dem gleichen Alphabet Σ und L_1 sowie L_2 die zugehörigen Sprachen. Auf dem Produkt der Zustandsmengen $Q_1 \times Q_2$ definieren wir nun einen Automaten,

$$\mathcal{A}_1 \times \mathcal{A}_2 = (Q_1 \times Q_2, \Sigma, \delta, (q_0^1, q_0^2), F)$$

mit

$$\delta((q_1, q_2), a) := (\delta_1(q_1, a), \delta_2(q_2, a))$$

bedeutet, dass δ_1 für die erste Komponente zuständig ist, und δ_2 für die zweite. Durch Induktion über $w \in \Sigma^*$ folgt problemlos:

Lemma 2.3.1. *Für jedes $w \in \Sigma^*$ und alle $q_1 \in Q_1,\ q_2 \in Q_2$ gilt*

$$\delta^*((q_1, q_2), w) = (\delta_1^*(q_1, w), \delta_2^*(q_2, w)).$$

Somit gilt:

$$w \in L(\mathcal{A}_1 \times \mathcal{A}_2) \iff \delta^*((q_0^1, q_0^2), w) \in F \iff (\delta_1^*(q_0^1, w), \delta_2^*(q_0^2, w)) \in F.$$

Figur 2.3.1 zeigt zwei Automaten A und B mit 3 bzw. 2 Zuständen, sowie deren Produktautomat mit $3*2 = 6$ Zuständen. Der Input wird in A und B separat verarbeitet. Die erste Komponente des erreichten Zustandes wird von A bestimmt, die zweite von B. Für die Auswahl von F gibt es zwei natürliche Kandidaten:

$$F_\cap := (F_1, F_2)$$

oder

$$F_\cup := (F_1 \times Q_2 \cup Q_1 \times F_2).$$

Für beliebige (q_1, q_2) gilt:

$$(q_1, q_2) \in F_\cap \iff q_1 \in F_1 \textit{ und } q_2 \in F_2,$$

aber

$$(q_1, q_2) \in F_\cup \iff q_1 \in F_1 \textit{ oder } q_2 \in F_2.$$

Bei der Wahl von $F_\cap$ als akzeptierende Menge erhalten wir einen Automaten, der $L_1 \cap L_2$ erkennt, bei die Wahl von $F_\cup$ einen Automaten, der $L_1 \cup L_2 = L_1 + L_2$ erkennt. Abbildung 2.3.1 zeigt den Produktautomaten mit $F_\cap = F_1 \times F_2$.

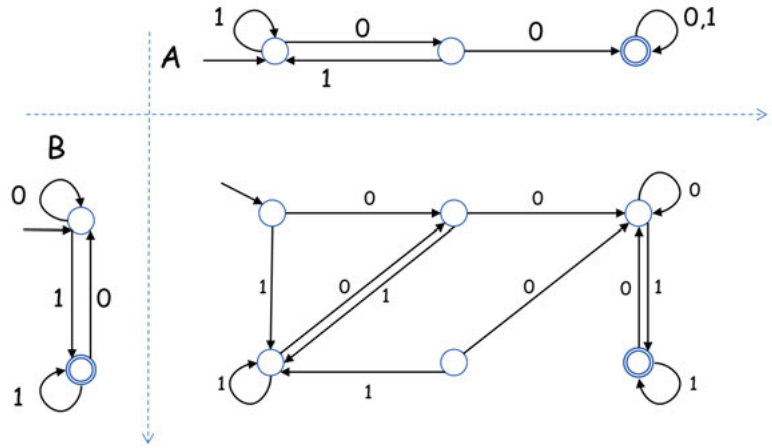


Abb. 2.3.1: Produktautomat

Sind also L_1 und L_2 von einem *DFA* erkennbar, dann auch $L + L$ und $L_1 \cap L_2$. Die erkennenden Automaten unterscheiden sich nur durch die Wahl der akzeptierenden Zustände.

2.3.3 Vergleich von Automaten

Seien $\mathcal{A} = (P, \Sigma, \delta_\mathcal{A}, p_0, F_\mathcal{A})$ und $\mathcal{B} = (Q, \Sigma, \delta_\mathcal{B}, q_0, F_\mathcal{B})$ zwei Automaten mit dem selben Eingabealphabet. Eine Abbildung $\varphi : P \to Q$ heißt *Homomorphismus* von $\mathcal{A}$

nach $\mathcal{B}$, falls für alle $p \in P$ und $a \in \Sigma$ gilt:

$$\varphi(p_0) = q_0 \tag{2.3.1}$$
$$\varphi(\delta_{\mathcal{A}}(p,a)) = \delta_{\mathcal{B}}(\varphi(p),a) \tag{2.3.2}$$
$$p \in F_{\mathcal{A}} \iff \varphi(p) \in F_{\mathcal{B}}. \tag{2.3.3}$$

Problemlos lässt sich 2.3.2 auf $\delta^\star$ ausdehnen: Für alle $w \in \Sigma^\star$ und alle $p \in P$ gilt:

$$\varphi(\delta^\star_{\mathcal{A}}(p,w)) = \delta^\star_{\mathcal{B}}(\varphi(p),w) \tag{2.3.4}$$

Lemma 2.3.2. *Ist $\varphi : \mathcal{A} \to \mathcal{B}$ ein Homomorphismus, so ist $L(\mathcal{A}) = L(\mathcal{B})$.*

Beweis. Wir rechnen nach:

$$\begin{aligned} w \in L(\mathcal{A}) &\overset{\text{Def. 2.2.3}}{\iff} \delta^\star_{\mathcal{A}}(p_0,w) \in F_{\mathcal{A}} \\ &\overset{2.3.3}{\iff} \varphi(\delta^\star_{\mathcal{A}}(p_0,w)) \in F_{\mathcal{B}} \\ &\overset{2.3.2}{\iff} \delta^\star_{\mathcal{B}}(\varphi(p_0),w) \in F_{\mathcal{B}} \\ &\overset{2.3.1}{\iff} \delta^\star_{\mathcal{B}}(q_0,w) \in F_{\mathcal{B}} \\ &\overset{\text{Def. 2.2.3}}{\iff} w \in L(\mathcal{B}). \end{aligned}$$

□

2.4 Vereinfachung von Automaten

2.4.1 Erreichbarkeit

Als interessantes Phänomen kann man in Abb. 2.3.1 erkennen, dass ein Zustand q im Produktautomaten nicht *erreichbar* ist: Es gibt kein Wort w mit $\delta^\star(q_0,w) = q$. Diesen Zustand mit allen ein- und ausgehenden Pfeilen kann man selbstverständlich entfernen ohne die Sprache zu verändern, welche von dem Automaten erkannt wird. Man kann den Automaten also verkleinern, ohne dass sich etwas an seinen Erkennungseigenschaften ändert.

Bemerkung 2.4.1. Im Folgenden werden wir annehmen, dass in unseren Automaten alle nicht erreichbaren Zustände schon entfernt wurden.

2.4.2 Äquivalente Zustände

Betrachten wir im Produktautomaten von Abb. 2.3.1 die beiden schraffierten Zustände am linken Rand, so erkennen wir, dass dass es absolut gleichgültig ist, ob wir uns in dem einen oder dem anderen befinden. Wenn irgendein Wort w uns von einem der beiden Zustände in einen Endzustand führt, dann auch von dem anderen. Solche Zustände heißen *äquivalent*.

Definition 2.4.2. Zwei Zustände p und q heißen *äquivalent*, und wir schreiben $p \sim q$, falls für jedes Wort $w \in \Sigma^*$ gilt: $\delta^*(p, w) \in F \iff \delta^*(q, w) \in F$.

Nicht äquivalente Zustände p und q nennt man auch *trennbar*, da es ein Wort w gibt, welches von p aus in einen Endzustand führt, von q aus aber nicht, oder umgekehrt. w ist dann ein *trennendes Wort* für p und q. In jedem Automaten trennt das leere Wort ε alle akzeptierenden Zustände $p \in F$ von allen nicht akzeptierenden $q \in (Q - F)$.

Beispiel 2.4.3. Im Automaten in Figur 2.3.1 sind die beiden schraffierten Zustände, nennen wir sie p und q, äquivalent, also nicht trennbar. Beide sind nicht akzeptierend, das leere Wort ε kann sie daher nicht trennen. Wegen $\delta(p, 0) = \delta(q, 0)$ und $\delta(p, 1) = \delta(q, 1)$ gilt für jedes längere Wort $w = 0u$ oder $w = 1u$ erst recht, dass $\delta^*(p, w) = \delta^*(q, w)$, somit kann auch kein nichtleeres Wort die beiden Zustände trennen.

Alle anderen (erreichbaren) Zustände sind voneinander trennbar: $w = \varepsilon$ trennt den akzeptierenden Zustand von allen anderen. $w = 1$ und $w = 01$ trennen die übrigen (erreichbaren) Zustände.

Die Idee ist jetzt, äquivalente Zustände zusammenzufassen und aus den Äquivalenzklassen einen Automaten mit weniger Zuständen zu gewinnen, der die gleiche Sprache erkennt. Wir können die relevanten Eigenschaften von $\sim$ in der folgenden Definition bündeln:

Definition 2.4.4. Sei $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ ein Automat. Eine Äquivalenzrelation θ auf Q heißt *Kongruenz*, falls sie im folgenden Sinne mit δ und F verträglich ist:

$$p \,\theta\, q \quad \Longrightarrow \quad \delta(p, a) \,\theta\, \delta(q, a) \quad \text{für alle } a \in \Sigma \tag{2.4.1}$$

$$p \,\theta\, q \quad \Longrightarrow \quad (p \in F \iff q \in F). \tag{2.4.2}$$

Für die in 2.4.2 definierte Relation $\sim$ überlegen wir uns zuerst, dass es sich um eine Kongruenz handelt. Dass $\sim$ eine Äquivalenzrelation auf Q ist kann man sofort aus der Definition ablesen. Sei nun $p \sim q$ vorausgesetzt. Wäre $\delta(p, q) \not\sim \delta(q, a)$, dann gäbe es also ein trennendes Wort w mit $\delta^*(\delta(p, a), w) \in F$ und $\delta^*(\delta(q, a), w) \notin F$ (oder umgekehrt). Dann wäre aber $\delta^*(p, aw) \in F$ und $\delta^*(q, aw) \notin F$ und somit wäre aw ein trennendes Wort für p und q im Widerspruch zu $p \sim q$. Die Eigenschaft 2.4.2 ist Teil der Definition von $\sim$. Es gilt sogar:

Lemma 2.4.5. *$\sim$ ist die größte Kongruenz auf Q.*

Beweis. Sei θ eine beliebige Kongruenz auf $\mathcal{A}$. Wir wollen zeigen, dass $\theta \subseteq \sim$, also dass kein Paar $(p, q) \in \theta$ durch ein Wort $w \in \Sigma^*$ trennbar ist. Wir führen einen Widerspruchsbeweis und nehmen an, dass es ein Wort gäbe, welches zwei Zustände aus θ

trennt. Dann gibt es auch ein Wort w minimaler Länge mit dieser Eigenschaft. Entweder ist $w = \varepsilon$ oder $w = au$. Beide Möglichkeiten führen wir zum Widerspruch:

$w = \varepsilon$: Wegen 2.4.2 kann kein Paar $(p, q) \in \theta$ durch $w = \varepsilon$ getrennt werden, da $\delta^*(p, \varepsilon) = p \in F \iff q = \delta^*(q, \varepsilon) \in F$.

$w = au$: Angenommen $p\theta q$ und w trennt p von q, also z.B. $\delta^*(p, au) \in F$ und $\delta^*(q, au) \notin F$, das heißt $\delta^*(\delta(p, a), u) \in F$ und $\delta^*(\delta(q, a), u) \notin F$. Wegen 2.4.1 ist aber $(\delta(p, a), \delta(q, a)) \in \theta$, somit haben wir mit u ein kürzeres Wort als w gefunden, welches ein Paar von Zuständen aus θ trennt. Das ist ein Widerspruch zur minimalen Länge von w.

□

Wir können die offensichtlich wichtige Relation $\sim$ auch auf eine andere Weise charakterisieren. Wenn wir von einem Automaten $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ ausgehen und lediglich einen anderen Zustand $q \in Q$ als Anfangszustand wählen, erhalten wir einen Automaten $\mathcal{A}_q := (Q, \Sigma, \delta, q, F)$. Für die zugehörigen Sprachen gilt:

Lemma 2.4.6. *Seien $p, q \in Q$.*
1. $L(\mathcal{A}_p) = L(\mathcal{A}_q) \iff p \sim q$
2. $\delta(p, a) = q \implies L(\mathcal{A}_q) = \partial_a(L(\mathcal{A}_p))$

Beweis. Für alle $w \in \Sigma^*$ gilt: $w \in L(\mathcal{A}_p) \iff \delta^*(p, w) \in F$. Für die zweite Aussage rechnen wir nach:

$$\begin{aligned} u \in \partial_a(L(\mathcal{A}_p)) &\iff au \in L(\mathcal{A}_p) \\ &\iff \delta^*(p, au) \in F \\ &\iff \delta^*(\delta(p, a), u) \in F \\ &\iff u \in L(\mathcal{A}_{\delta(p,a)}). \end{aligned}$$

□

2.4.3 Faktorautomaten

Sei $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ ein Automat und θ eine Kongruenz auf Q. Die Kongruenz θ zerteilt die Menge Q in Klassen paarweise äquivalenter Zustände. Sei p ein Zustand, so sei $p/\theta := \{q \in Q \mid p\,\theta\,q\}$ die Klasse, in der sich p befindet. Offensichtlich gilt:

$$p\theta q \iff p/\theta = q/\theta. \tag{2.4.3}$$

Die Menge aller Äquivalenzklassen ist $Q/\theta := \{q/\theta \mid q \in Q\}$. Auf ihr definieren wir den Faktorautomaten $\mathcal{A}/\theta := (Q/\theta, \Sigma, \delta_\theta, q_0^\theta, F_\theta)$ folgendermaßen:

$$\delta_\theta(q/\theta, a) := \delta(q, a)/\theta \tag{2.4.4}$$

$$q_0^\theta := q_0/\theta \tag{2.4.5}$$

$$q/\theta \in F_\theta :\Longleftrightarrow q \in F. \tag{2.4.6}$$

Die Definition erscheint problematisch, da die Klasse q/θ ja nicht das Element q eindeutig bestimmt. Falls $q/\theta = q'/\theta$ ist, soll das Ergebnis einerseits $\delta(q, a)/\theta$ sein, andererseits $\delta(q', a)/\theta$. Zum Glück besagt aber gerade 2.4.1 in Verbindung mit 2.4.3, dass beide Ergebnisse übereinstimmen. Es folgt durch eine einfache Induktion, dass für beliebige Worte $w \in \Sigma^\star$ gilt

$$\delta_\theta^\star(q/\theta, w) = \delta^\star(q, w)/\theta. \tag{2.4.7}$$

Analog ist wegen 2.4.2 auch F_θ wohldefiniert.

Wir haben jetzt also einen Automaten $\mathcal{A}_\theta$ auf den Kongruenzklassen von θ definiert. Die Berechnung von δ und F geschieht repräsentantenweise: Um $\delta(q/\sim, a)$ zu berechnen, greifen wir uns ein Element q' aus $q/\sim$ heraus, berechnen $\delta(q', a) = q''$ und liefern dessen Klasse $q''/\sim$ als Ergebnis. Um zu bestimmen, ob $q/\sim$ ein Endzustand ist, greifen wir uns wieder ein beliebiges $q' \in q/\sim$ und fragen, ob $q' \in F$ ist. Dass dies gut geht, dafür sorgen gerade die Eigenschaften 2.4.1 und 2.4.2.

Falls θ trivial ist, also falls $\theta = id_Q$ stimmt $\mathcal{A}_\theta$ mit $\mathcal{A}$ überein. Falls θ nichttrivial ist, falls also mindestens ein $p \neq q$ existiert mit $p\theta q$, so hat $\mathcal{A}_\theta$ weniger Zustände als $\mathcal{A}$ und trotzdem gilt:

Satz 2.4.7. *Für jede Kongruenz θ auf $\mathcal{A}$ ist $L(\mathcal{A}_\theta) = L(\mathcal{A})$.*

Beweis.

$$\begin{aligned}
w \in L(\mathcal{A}_\theta) &\overset{\text{Def. 2.2.3}}{\Longleftrightarrow} \delta_\theta^\star(q_0^\theta, w) \in F_\theta \\
&\overset{2.4.5}{\Longleftrightarrow} \delta_\theta^\star(q_0/\theta, w) \in F_\theta \\
&\overset{2.4.7}{\Longleftrightarrow} \delta^\star(q_0, w)/\theta \in F_\theta \\
&\overset{2.4.6}{\Longleftrightarrow} \delta^\star(q_0, w) \in F \\
&\overset{\text{Def. 2.2.3}}{\Longleftrightarrow} w \in L(\mathcal{A}).
\end{aligned}$$

□

Korollar 2.4.8. $L(\mathcal{A}/\sim) = L(\mathcal{A})$.

Durch Zusammenfassung äquivalenter Zustände haben wir mit $\mathcal{A}_{/\sim}$ also einen Automaten gewonnen, der potenziell weniger Zustände hat als der Originalautomat, und der trotzdem die gleiche Sprache erkennt wie $\mathcal{A}$. Aber ist dies bereits der Minimalautomat für die Sprache $L(\mathcal{A})$. Vielleicht war der Ausgangsautomat $\mathcal{A}$ schlecht gewählt

und hat sich durch den Übergang zu $A/\sim$ nur unwesentlich verbessert? Vielleicht sind in $\mathcal{A}/\sim$ neue Zustände entstanden, die jetzt äquivalent sind? Müssen wir vielleicht erneut äquivalente Zustände identifizieren und zu $(\mathcal{A}/\sim)$ übergehen, etc.?

Dass dies alles nicht der Fall ist, wird sich aus dem folgenden Kapitel ergeben.

2.4.4 Die Nerode-Kongruenz

Diesmal beginnen wir mit einer Sprache $L \subseteq \Sigma^*$, welche wir durch einen geeigneten Automaten $\mathcal{A}$ implementieren wollen. Wir suchen also einen Automaten $\mathcal{A}$ mit $\mathcal{L}(\mathcal{A}) = L$. Jedes Wort $u \in \Sigma^*$ führt uns in dem gesuchten Automaten zu einem Zustand $q_u := \delta(q_0, u)$. Wir haben aber unendlich viele Worte u in Σ^*, während der Automat nur endlich viele Zustände haben soll. Wir sind also öfter gezwungen, für q_u und für q_v den gleichen Zustand wählen.

Eine Identifikation von $q_u = q_v$ hat aber die Konsequenz, dass für jedes Wort w gilt: $\delta^*(q_u, w) \in F \iff \delta^*(q_v, w) \in F$, mithin $uw \in L \iff vw \in L$. Im Umkehrschluss gilt: Falls $uw \in L$ aber $vw \notin L$, dann muss in jedem Automat, der L erkennen will, $q_u = \delta(q_0, u) \neq \delta(q_0, v) = q_v$ sein.

Definition 2.4.9. Sei $L \subseteq \Sigma^*$ eine beliebige Sprache und $u, v \in \Sigma^*$. Wir definieren die sogenannte *Nerode-Kongruenz* $\sim_L$ auf Σ^* durch

$$u \sim_L v :\iff \forall w \in \Sigma^*.(uw \in L \iff vw \in L). \tag{2.4.8}$$

$\sim_L$ ist offensichtlich eine Äquivalenzrelation auf Σ^* und wir bezeichnen u und v als L-äquivalent , falls $u \sim_L v$. Folgende Eigenschaften, die unmittelbar aus der Definition folgen, sind verantwortlich für das Attribut “Kongruenz“:

$$u \sim_L v \implies uw \sim_L vw \text{ für alle } w \in \Sigma^* \tag{2.4.9}$$

$$u \sim_L v \implies u \in L \iff v \in L. \tag{2.4.10}$$

Im anderen Fall, wenn also $u \not\sim_L v$ gilt, gibt es ein Wort $w \in \Sigma^*$ mit $uw \in L$ aber $vw \notin L$ oder umgekehrt $uw \notin L$ aber $vw \in L$. u und v heißen dann L-*trennbar* mit *trennendem Wort* w.

Mit $index(L)$ bezeichnen wir die Anzahl der Äquivalenzklassen von $\sim_L$. Diese kann endlich oder unendlich sein.

Für den Index von L gilt die wichtige Beziehung:

Satz 2.4.10. *[Myhill-Nerode] Jeder Automat, der eine Sprache L erkennt, hat mindestens $index(L)$ viele Zustände.*

Beweis. Sei $u \not\sim_L v$, dann gibt es o.B.d.A ein w mit $uw \in L$ aber $vw \notin L$. Für jeden Automaten $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$, der L erkennt, müssen dann $q_u := \delta(q_0, u)$ und $q_v :=$

$\delta(q_0, v)$ verschiedene Zustände sein, denn $\delta^*(q_u, w) = \delta^*(q_0, uw) \in F$ aber $\delta(q_v, w) = \delta^*(q_0, vw) \notin F$ (bzw. umgekehrt). Somit benötigt $\mathcal{A}$ mindestens so viele Zustände wie es paarweise nicht $\sim_L$-äquivalente Worte gibt.

Um den Index einer Sprache abzuschätzen, sucht man eine Menge M von Worten aus Σ^*, die paarweise L-trennbar sind. Da sie zu verschiedenen Äquivalenzklassen von $\sim_L$ gehören, muss $index(L) \geq |M|$ sein. □

Beispiel 2.4.11. Wir betrachten noch einmal die Sprache L_5 die aus allen durch 5 teilbaren Binärzahlen besteht. Die Elemente der Menge $M = \{\varepsilon, 1, 10, 11, 111\}$ sind paarweise L-trennbar, also nicht $\sim_{L_5}$ - äquivalent, denn $\varepsilon\varepsilon \in L_5$ aber $w\varepsilon \notin L_5$ für alle anderen $w \in M$. Ebenso ist $101 \in L_5$ aber $w01 \notin L_5$ für alle anderen $w \in L_5$ etc.. Die linke der beiden folgenden Tabellen liefert für je zwei $u, v \in M$ ein trennendes Wort w:

	ε	1	10	11	111
ε					
1	ε				
10	ε	01			
11	ε	01	1		
111	ε	01	1	11	

	q_0	q_1	q_2	q_3	q_4
q_0					
q_1	ε				
q_2	ε	01			
q_3	ε	01	1		
q_4	ε	01	1	11	

Jeder Automat, der L_5 erkennt, muss also mindestens $|M| = 5$ Zustände besitzen. Figur 2.2.2 zeigt aber schon einen Automaten, der L_5 erkennt, somit ist klar, dass L_5 ein Automat für L_5 mit minimaler Zustandsmenge ist. Folglich sind je zwei Zustände des Automaten trennbar. Die Menge M besteht übrigens aus allen Pfaden zu den Zuständen dieses Automaten und die L-trennenden Worte in der Tabelle sind die gleichen Worte, welche die entsprechenden Zustände des Automaten trennen, wie in der rechten Tabelle ersichtlich.

Der Satz von Myhill und Nerode kann auch verwendet werden, um zu zeigen, dass eine vorgegebene Sprache L durch *keinen* endlichen Automaten erkennbar ist: Man sucht einfach eine unendliche Menge $M \subseteq \Sigma^*$ deren Elemente paarweise L-trennbar sind.

Beispiel 2.4.12. Die Sprache $L_{anbn} = \{a^n b^n \mid n \in \mathbb{N}\}$ ist nicht durch einen endlichen Automaten erkennbar, denn die Elemente der Menge $M = \{a, a^2, a^3, \ldots\}$ sind paarweise trennbar: b^n trennt jeweils a^n von jedem anderen $m \in M$, denn $a^n b^n \in L$ aber $a^m b^n \notin L$ falls $m \neq n$. Somit ist der Index von L_{anbn} unendlich, weshalb diese Sprache von keinem endlichen Automaten erkannt werden kann.

Als für die Praxis relevantes Beispiel zeigt man analog, dass die Dyck-Sprache $L_{(,)}$ der „wohlgeformten Klammerausdrücke“ nicht durch einen endlichen Automaten erkennbar ist. Eine unendliche Menge paarweise $L_{(,)}$-trennbarer Worte ist z.B. $\{(, ((, (((, \ldots\}$.

Ebenso ist leicht zu sehen, dass für $|\Sigma| > 1$ die Sprache aller Palindrome, also aller Worte w mit $w = w^R$ nicht durch einen endlichen Automaten erkennbar ist.

Der Satz von Myhill-Nerode liefert nicht nur ein notwendiges, sondern auch hinreichendes Kriterium dafür, dass eine Sprache durch einen endlichen Automaten erkennbar ist. Die Betonung liegt hier auf „*endlich*“. Lässt man **unendlich viele Zustände** zu, so kann man **jede** Sprache $L \subseteq \Sigma^*$ durch einen Automaten erkennen.

Satz 2.4.10 besagt, dass wir mindestens $index(L)$ viele Zustände benötigen. Der folgende Satz versichert uns, dass wir mit $index(L)$ vielen Zuständen auch auskommen. Somit ist $index(L)$ die minimale Anzahl von Zuständen für einen Automaten, der die Sprache L erkennt.

Satz 2.4.13. *Für jede Sprache L gibt es einen Automaten $\mathcal{A}$ mit index(L) vielen Zuständen und $L = L(\mathcal{A})$.*

Beweis. Wir definieren einen Automaten $\mathcal{A}_L$ auf den Äquivalenzklassen von $\sim_L$ durch

$$\begin{aligned} \delta_L(u/\sim_L, a) &:= (ua)/\sim_L \\ F_L &:= \{u/\sim_L \mid u \in L\}. \end{aligned}$$

Dass δ_L und F_L unabhängig von der Wahl der Repräsentanten und damit wohldefiniert sind folgt sofort aus 2.4.9 und 2.4.10. Für δ_L^* erhalten wir entsprechend:

$$\delta_L^*(u/\sim_L, w) = (uw)/\sim_L .$$

Wählen wir jetzt noch $\varepsilon/\sim_L$ als Anfangszustand für $\mathcal{A}_L$ so rechnen wir nach:

$$\begin{aligned} w \in L(\mathcal{A}_L) &\iff \delta_L^*(\varepsilon/\sim_L, w) \in F_L \\ &\iff (\varepsilon w)/\sim_L \in F_L \\ &\iff w/\sim_L \in F_L \\ &\iff w \in L. \end{aligned}$$

□

Falls L keinen endlichen Index hat, liefert die Konstruktion einen Automaten mit unendlich vielen Zuständen. Aus praktischen Gründen wollten wir den Begriff „Automat“ aber nur auf solche mit endlichen Zustandsmengen beschränken.

In jedem Falle folgt mit 2.4.10, dass die Konstruktion sofort einen Automaten mit minimaler Zustandsmenge liefert. Diesen nennt man auch „*Minimalautomat*“ für die Sprache L.

Die Kombination von 2.4.10 und 2.4.13 besagt, auch, dass unser Kriterium um festzustellen, ob eine Sprache durch einen endlichen Automaten erkennbar ist, immer funktioniert:

Korollar 2.4.14. *[Regularitätstest] Eine Sprache ist* ***nicht*** *durch einen endlichen Automaten erkennbar, wenn es eine unendliche L-trennbare Menge $M \subseteq \Sigma^*$ gibt.*

Aufgabe 2.4.15. Zeigen Sie mit Hilfe des Regularitätstests von Myhill-Nerode, dass folgende Sprachen nicht regulär sind:0pt

- $L_1 = \{a^n b^m \mid m \neq n\}$
- $L_2 = \{ww^R \mid w \in \Sigma^*\}$

Aufgabe 2.4.16. Stellen Sie sich ein Taxi in New York vor, dessen Navigationssystem einen Fahrplan als Folge von Richtungsangaben $w \in \{S, O, N, W\}^*$ erhält. Dabei bedeutet S bzw. O, N, W: fahre einen Block nach Süden, bzw. Osten, Norden, Westen. Ist die Menge L aller derjenigen Fahrpläne, die das Taxi zum Schluss wieder an seinen Ausgangspunkt bringen, regulär? Beispielsweise ist $NOSWWO \in L$ aber $NON \notin L$.

2.4.5 Minimierung

In der Praxis geht man bei der Konstruktion des minimalen Automaten zu einer Sprache L oft anders vor. Man beginnt mit irgendeinem Automaten $\mathcal{A}$, der die Sprache L erkennt. Dieser Automat kann zum Beispiel das Ergebnis einer Transformation eines regulären Ausdrucks in einen Automaten sein, oder das Ergebnis einer Konstruktion wie vielleicht eines Faktor- oder Produktautomaten. Meist erhält man dabei einen Automaten mit überflüssigen, nicht erreichbaren oder äquivalenten Zuständen. Davon ausgehend muss der Automat minimalisiert werden.

Im ersten Schritt entfernt man alle nicht erreichbaren Zustände. Durch eine Breitensuche ausgehend von q_0 findet man alle erreichbaren Zustände. Alle übrigen kann man entfernen.

In der nächsten Phase trennt man Zustände, die nicht äquivalent sein können. Dazu bedient man sich einer Tabelle, in der man für jedes Paar (q_i, q_j), von Zuständen (mit $i < j$) einträgt, ob q_i und q_j schon als trennbar erkannt wurden.

Sind q_i und q_j trennbar, dann auch q_j und q_i. Daher genügt es, nur diejenigen Paare $(q_{i,} q_j)$ zu tabellieren, für die $i < j$ ist.

Man beginnt mit je einem Eintrag für (q_i, q_j) wobei $q_i \in F$ und $q_j \notin F$ oder umgekehrt. In jeder Runde wählt man sich ein Paar (q_i, q_j) mit $i < j$, das noch nicht als trennbar markiert ist und ein Zeichen $e \in \Sigma$. Damit überprüft man, ob $\delta(q_i, e)$ und $\delta(q_j, e)$ schon als trennbar markiert sind. Wenn ja, markiert man auch (q_i, q_j).

Wenn in einer Runde kein neues Paar markiert werden kann, sind wir fertig.

Die folgende Figur zeigt einen Automaten und das Auffüllen der zugehörigen Tabelle in drei Phasen:

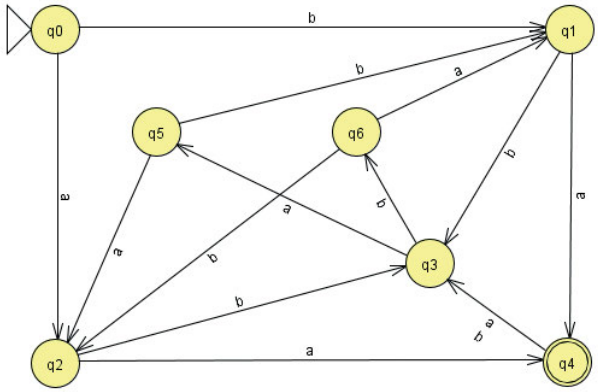


Abb. 2.4.1: Ausgangsautomat

Für das Beispiel des in Abb. 2.4.1 dargestellten Automaten beginnen wir damit, in der Tabelle die Paare zu markieren, bei denen einer in F ist und der andere nicht. Als nächstes wählen wir $e := a \in \Sigma$ und markieren alle (q_i, q_j) für die $(\delta(q_i, e), \delta(q_j, e))$ schon markiert ist. Im dritten Schritt wiederholen wir das gleiche mit $e := b$. Erneute Versuche mit $e := a$ und $e := b$ bringen keine neuen Markierungen, so dass wir abbrechen können.

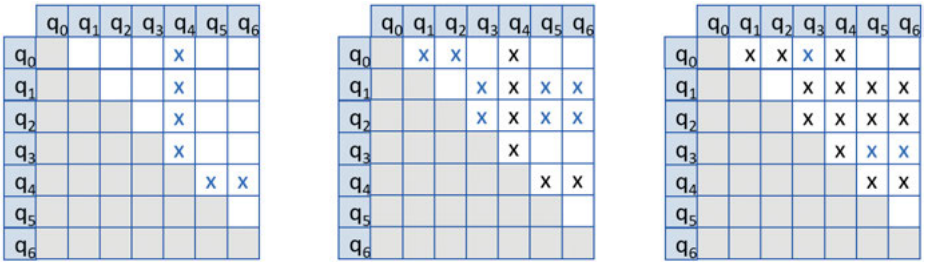


Abb. 2.4.2: Tabelle trennbarer Zustände

Die nicht markierten Positionen in der rechten oberen Hälfte der Tabelle zeigen, welche Zustände äquivalent sind. Hier bestehen die Äquivalenzklassen von $\sim$ aus $\{q_0, q_5, q_6\}, \{q_1, q_2\}, \{q_3\}, \{q_4\}$.

Diese Mengen sind die Zustände des Minimalautomaten, der von JFLAP folgendermaßen dargestellt wird. Die neu erzeugten Zustandsnamen q_0, q_1, q_2, q_3 stehen für diese Klassen, was in Figur 2.4.3 durch die zusätzlichen Beschriftungen ausgedrückt wird.

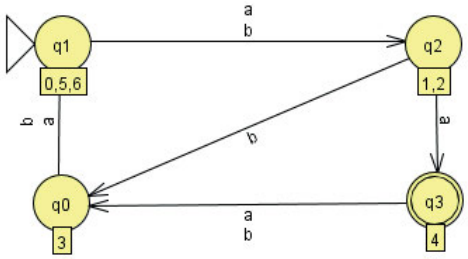


Abb. 2.4.3: Minimalautomat

2.4.6 Das Pumping Lemma

Wir haben bereits eine Methode kennengelernt, mit der wir nachweisen können, dass eine Sprache nicht durch einen endlichen Automaten erkennbar ist. Eine besser bekannte – und bei manchen Studenten gefürchtete – Methode beruht auf dem sogenannten „Pumping Lemma“. Dahinter verbirgt sich zunächst einmal folgende simple Beobachtung:

Ist $\mathcal{A}$ ein Automat mit k Zuständen, und $w \in L(\mathcal{A})$ ein Wort mit $|w| \geq k$. Dann muss im Verlauf der Erkennung des Wortes w mindestens ein Zustand zweimal besucht worden sein. q sei der erste solche Zustand. w lässt sich in drei Teile $w = xyz$ zerlegen:

–nach dem Einlesen von x gelangt man erstmalig in Zustand q, kurz: $\delta^*(q_0, x) = q$
–nach dem Einlesen von y gelangt man erneut zu q, also $\delta^*(q, y) = q$
–nach dem Einlesen von z gelangt man von q zu einem $q_f \in F$.

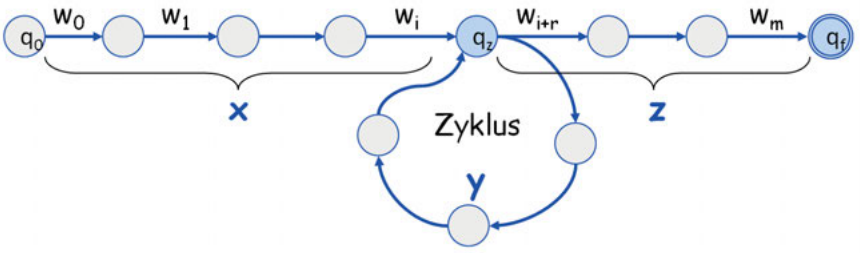


Abb. 2.4.4: Erkennung eines langen Wortes

Wenn aber $w = xyz$ vom Anfangszustand in einen akzeptierenden Zustand führt, dann auch $xyyz$, $xyyyz$, …, xy^nz für jedes $n \geq 0$, insbesondere auch xz. Man kann die Schleife von q nach q beliebig oft durchlaufen: 0-mal oder öfter. Man sagt auch, dass

man den Anteil y „aufpumpen“ oder „abpumpen“ kann. Für die Längen der Teilworte gilt, wenn q der erste Zustand war, der mehrfach besucht wurde: $0 < |y| \leq |xy| < k$. Wir haben gezeigt:

Lemma 2.4.17 (Pumping Lemma)**.** *Sei L eine reguläre Sprache. Dann gibt es ein k, so dass sich jedes Wort $w \in L$ mit $|w| \geq k$ zerlegen lässt als $w = xyz$ mit $0 < |y| \leq |xy| < k$ so dass jedes Wort $xy^n z$ zu L gehört.*

Intuitiv gesprochen gibt es ein k so dass jedes lange Wort ($|w| > k$) im vorderen Bereich ($|xy| < k$) einen nichtleeren Abschnitt ($|y| > 0$) besitzt, der sich beliebig auf- und abpumpen lässt, ohne die Sprache zu verlassen.

Beispiel 2.4.18. Zur Illustration betrachten wir den Automaten $\mathcal{A}$ in Figur 2.4.3. Jedes Wort aus $L(\mathcal{A})$ mit mehr als 3 Zeichen muss einen Zustand zweimal besuchen und somit eine Schleife im Automaten durchlaufen. So ist z.B. $ababa \in L(\mathcal{A})$. Das Teilwort bab beschreibt eine Schleife, die in q_2 beginnt und wieder endet. Daher gilt für jedes $n \in \mathbb{N}$, dass auch $a(bab)^n a \in L(\mathcal{A})$ ist. Insbesondere z.B. aa, $ababbaba$, $ababbabbaba$, ...

Das Pumping Lemma wird meist angewendet, um zu zeigen, dass bestimmte Sprachen *nicht* regulär sein können. Dazu nimmt man an, es gäbe ein k mit den im Pumping Lemma genannten Eigenschaften. Dann konstruiert man ein Wort $w \in L$ mit $|w| \geq k$ das im Bereich der ersten k Zeichen kein nichtleeres Teilwort y besitzt, welches sich beliebig auf- und abpumpen ließe, ohne L zu verlassen. Als umgangssprachliche Merkregel formuliert:

Fakt 2.4.19. *Um zu zeigen, dass eine Sprache L* nicht *regulär ist, finde für alle beliebig großen $k \in \mathbb{N}$ ein längeres Wort aus L, welches im k–vorderen Bereich keinen nichtleeren Abschnitt besitzt, der sich beliebig auf- und abpumpen ließe, ohne L zu verlassen.*

Beispiel 2.4.20. Die folgenden Sprachen sind nicht regulär. Um dies zu zeigen, wählen wir zu jedem k ein Wort w mit $|w| \geq k$, bei dem im vorderen Bereich kein Abschnitt beliebig auf- oder abgepumpt werden kann:

1. $L_{a^n b^n} = \{a^n b^n \mid n \in \mathbb{N}\}$ ist nicht regulär: Für beliebiges $k \in \mathbb{N}$ wählen wir $w = a^k b^k \in L$. Im vorderen Bereich (innerhalb der ersten k Zeichen) befinden sich nur a-s. Wenn wir w in diesem Bereich aufpumpen, hat das entstandene Wort mehr a-s als b-s, wir verlassen also die Sprache L.
2. Das gleiche Argument zeigt auch, dass die Dyck-Sprache $L_{(,)}$ nicht regulär ist: Zu gegebenem k können wir ebenfalls $w = (^k)^k \in L_{(,)}$ wählen und argumentieren, dass sich dieses Wort im vorderen Bereich nicht aufpumpen lässt.

3. $L_{a^m b^n} = \{a^m b^n \mid m \geq n\}$ ist nicht regulär: Zu gegebenem k wählen ebenfalls $w = a^k b^k \in L_{ambn}$. Wenn wir dieses Wort im vorderen Bereich aufpumpen, gelangen wir nicht zum ersehnten Widerspruch, denn mehr a-s als b-s sind in dieser Sprache ja erlaubt. Wenn wir w aber im vorderen Bereich „abpumpen“, erhalten wir aber ein Wort, das weniger a-s als b-s hat, und daher nicht mehr in L_{ambn} ist.

Nicht immer gestaltet sich die Suche nach einem Wort w zu einem gegebenen k so einfach wie in den vorigen Beispielen.

Beispiel 2.4.21. Dazu betrachten wir die Sprache $L = \{uu \mid u \in \{0,1\}^*\}$. Wir werden sehen, dass diese Sprache nicht regulär ist. Zu gegebenem k müssen wir ein Wort w finden, so dass $w \in L$ ist, $|w| \geq k$ und w im vorderen Bereich nirgends beliebig auf- und abgepumpt werden kann.

Erster Versuch: $w = 0^k 1^k$.
Dieser Versuch ist ungültig, da $w \notin L$ ist.

Zweiter Versuch: $w = (01)^k (01)^k$.
Für $k \geq 1$ lässt sich w im k-vorderen Bereich aufpumpen, ohne die Sprache L zu verlassen. Dies erkennt man anhand der Zerlegung $w = (01)^k(01)^k = (0101)(01)^{2k-2}$. Offensichtlich kann man den vorderen Teil beliebig auf- und abpumpen und bleibt in L. Somit ist auch dieser Beweisversuch nicht zielführend.

Dritter Versuch: $w = 0^k 10^k 1$.
Dieser Versuch ist erfolgreich. Zunächst ist $w \in L$ und $|w| \geq k$. Im k-vorderen Teil von w befinden sich nur 0-en. Aufpumpen liefert in jedem Fall ein Wort $u = 0^r 10^k 1$ mit $r > k$, welches offensichtlich nicht in L ist.

Das Pumping Lemma ist unter Studenten nicht gerade beliebt, erst recht nicht, wenn es strikt in mathematischer Sprache formuliert ist:

$$
\begin{aligned}
&L \text{ regulär} \implies \\
&\quad \exists k \in \mathbb{N}.\, \forall w \in L.\, |w| \geq k \implies \\
&\quad\quad \exists x, y, z \in \Sigma^*.\, 0 < |y| \leq |xy| \leq k \wedge w = xyz \wedge \forall n \in \mathbb{N}.\, xy^n z \in L
\end{aligned}
$$

Für die Anwendung, nämlich um zu zeigen, dass eine Sprache nicht regulär ist, verwendet man die Kontraposition dieser Aussage:

$$
\begin{aligned}
&(\forall k \in \mathbb{N}.\, \exists w \in L.\, |w| \geq k \wedge \forall x, y, z \in \Sigma^*.\, w = xyz \wedge 0 < |y| \leq |xy| \leq k \\
&\quad \implies \exists n \in \mathbb{N}. xy^n z \notin L\,) \\
&\quad\quad \implies L \text{ nicht regulär.}
\end{aligned}
$$

Eine derart gewaltige Formel für eine einfache Tatsache ist nicht jedermanns Sache. Zudem führt das Pumping Lemma nicht immer zum Ziel, denn in den obigen Formeln kann man die Implikation nicht durch eine Äquivalenz ersetzen.

Im Gegensatz dazu kommen wir mit der Nerode-Methode immer zum Ziel, das garantiert uns Korollar 2.4.14. Zur Illustration betrachten wir das folgende Beispiel:

Beispiel 2.4.22. $L_{diff} = \{a^m b^n \mid m \neq n\}$ ist nicht regulär.

Wollten wir das Pumping Lemma anwenden, müssten wir zu beliebigen $k \in \mathbb{N}$ ein $w \in L_{diff}$ mit $|w| \geq k$ finden können, das im Bereich der ersten k Zeichen keinen Abschnitt y hat, der beliebig auf- und abgepumpt werden kann ohne L zu verlassen. Das ist eine Herausforderung, die wir unseren Lesern überlassen wollen.

Mit dem Nerode-Lemma ist dagegen der Nachweis, dass L_{diff} nicht regulär ist, ganz einfach: $M := \{a^m \mid m \in \mathbb{N}\} \subseteq \Sigma^*$ ist eine L-trennbare Menge, denn das Wort b^m trennt a^m von jedem anderen Element von M.

Oft argumentiert man aber auch mit der Konstruktion von anderen Sprachen, von denen man bereits weiß, dass sie regulär sind. Im Falle von L_{diff} könnte man folgendermaßen argumentieren: Wäre L_{diff} regulär, dann gäbe es einen Automaten $\mathcal{A}$, der L_{diff} erkennt. Dessen Komplementation würde dann die Sprache

$$L = \Sigma^* - L_{diff} = \{a^m b^m \mid m \in \mathbb{N}\}$$

erkennen. Wir wissen aber bereits, dass diese nicht regulär ist.

2.5 Automaten mit Ausgabe – Transducer

Ein Automat erzeugt keine eigentliche Ausgabe, er erkennt ein Wort, wenn es in einen akzeptierenden Zustand führt, ansonsten erkennt er es nicht. In der Praxis kann man verschiedenen akzeptierenden Zuständen eines Automaten oft auch verschiedene Bedeutungen zumessen. Ein Wort kann man danach klassifizieren, welcher akzeptierende Zustand mit ihm erreicht wurde. Diese Klassifizierung kann als Ausgabe des Automaten verstanden werden.

Allgemein spricht man von einem *Transducer*, der aus einem Strom von Eingabezeichen einen Strom von Ausgabezeichen erzeugt. Ein- und Ausgabe dürfen verschiedene Alphabete Σ und Γ verwenden. Wie „normale“ Automaten besitzen Transducer einen Startzustand $q_0 \in Q$ und eine Übergangsfunktion $\delta : Q \times \Sigma \to Q$, aber nicht notwendigerweise akzeptierende Zustände. Stattdessen werden zu gewissen Zeiten Ausgaben $a \in \Gamma$ erzeugt. Transducer werden als Scanner oder auch zum Entwurf von Schaltungen verwendet.

2.5.1 Scanner

Die folgende Figur zeigt einen Automaten, der das Schlüsselwort „`if`“ oder Variablennamen oder natürliche Zahlen erkennen soll. Variablennamen dürfen Ziffern enthalten, allerdings nicht am Anfang. Außerdem dürfen sie nicht mit einem Schlüsselwort der Sprache, wie z.B. „`if`“ übereinstimmen. In dem gezeigten Automat klassifizieren die akzeptierenden Zustände q_1, q_4 und q_3 der Reihe nach *Zahlen*, *Variablennamen* und das Schlüsselwort „`if`“.

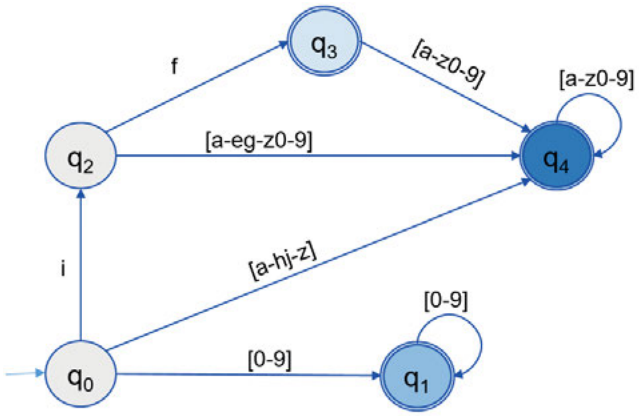


Abb. 2.5.1: Scanner als klassifizierender Automat

Solche klassifizierende Automaten heißen auch *Scanner*. Statt ein fertiges Wort zu verarbeiten, nimmt ein Scanner einen Strom von Daten entgegen, zerlegt diesen in Worte und klassifiziert diese. Wenn ein akzeptierender Zustand erreicht ist, kann das bisher gelesene Anfangswort abgeschnitten und klassifiziert werden. Es wird dann ein sogenanntes *Token* ausgegeben, das die Klasse des Wortes repräsentiert und der Scanner springt anschließend wieder in seinen Startzustand.

In der Praxis ist es oft erlaubt, dass ein akzeptiertes Wort ein Präfix eines anderen akzeptierten Wortes sein kann. So könnte im folgenden Beispiel eines Scanners sowohl das Schlüsselwort „`if`“ als auch die Variable „`ifen`“ akzeptiert werden. Für solche Fälle lesen Scanner meist die Eingabe bis zum nächsten Zeilenende-Zeichen oder zum nächsten Leerzeichen und wählen den längsten Präfix aus, der zu einem akzeptierenden Zustand führt.

Aus dem ursprünglichen String, der vielleicht ein Java-Programm darstellt, entsteht somit eine Folge von Token. Für spätere Phasen des Compilierens ist es vielleicht gar nicht wichtig, ob der eingelesene String „JahresEinkommen + 27“ hieß oder „x+1“. In beiden Fällen würde eine Tokenfolge, wie etwa ***id addOp num*** erzeugt, es wurde also ein Bezeichner (***id*** für *identifier*) dann die Additionsoperation (***addOp***) und an-

schließend ein Numeral ***num*** erkannt. Der konkrete Namen des Bezeichners, oder der Wert des Numerals sind uninteressant, wenn es nur darum geht, zu entscheiden, ob das Programm syntaktisch korrekt ist.

Falls es nicht nur um die syntaktische Korrektheit eines Programmtextes geht, sondern das Originalprogramm auch übersetzt werden soll, müssen gewisse Token auch noch Werte mitschleppen – numerische Token ihren Zahlenwert und Bezeichner auch ihren Namen.

2.5.2 Moore-Automaten

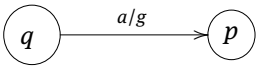
Moore-Automaten sind Transducer mit einer Ausgabefunktion $y : Q \to \Gamma$. Die Ausgabe hängt also nur von dem aktuellen Zustand ab. In der graphischen Darstellung ergänzt man jeden Zustand q durch seine Ausgabe s und markiert die Zustandsknoten daher durch „q/r". „Normale" Automaten könnte man als Moore-Automaten mit einer Ausgabefunktion $y : Q \to \{0, 1\}$ auffassen, wobei $y(q) = 1$ ausdrückt, dass q akzeptierend ist.

2.5.3 Mealy-Automaten

Mealy-Automaten sind Transducer, deren Ausgabe vom aktuellen Zustand und dem gerade eingelesenen Zeichen abhängen darf, also

$$y : Q \times \Sigma \to \Gamma.$$

In der graphischen Darstellung zeichnet man einen Pfeil von q nach p und beschriftet ihn mit a/g, falls $\delta(q, a) = p$ und $y(q, e) = g$:



Man kann einen Moore-Automaten auch als Mealy-Automaten ansehen, bei dem y nicht von ihrem zweiten Argument abhängt. Umgekehrt kann man auch Mealy-Automaten als Moore-Automaten kodieren, die Anzahl der Zustände kann sich dabei aber vervielfachen.

2.6 Nichtdeterministische Automaten

Automaten sind extrem effiziente Mechanismen um Sprachen zu erkennen. Eine Automatentabelle kann direkt in ein Maschinenspracheprogramm umgewandelt werden. Aus Zuständen q_i werden Programmzeilen und aus $\delta(q_i, a) = q_j$, was dem Eintrag q_j in Zeile q_i und Spalte z in der Automatentabelle entspricht, wird eine Programmzeile mit Label i Sprung zu Programmzeile j.

```
qi : switch(letter) {
       case a : goto δ(q_i, a);
       case b : goto δ(q_i, b);
        ...
     }
```

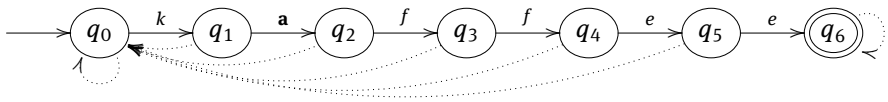
2.6.1 Spezifikation mit deterministischen Automaten

Automaten können beispielsweise für die Suche nach Worten oder Begriffen in einem Text eingesetzt werden. Als Beispiel betrachten wir die Suche nach einem Wort wie z.B. `kaffee` in einem Text. Konzentrieren wir uns zunächst nur auf die Frage, ob das Wort in einem Text t auftaucht, so benötigen wir einen Automaten für die Sprache

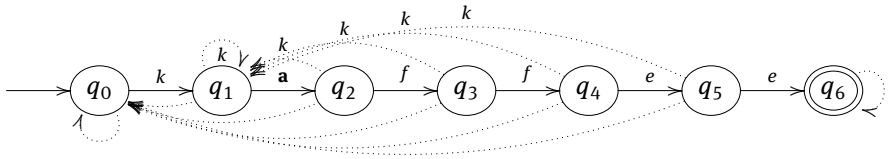
$$L = \Sigma^* \text{kaffee} \Sigma^*$$

Ein Text t enthält genau dann das Wort `kaffee`, wenn $t \in L$ ist.

Ein Automat für die Sprache ist schnell gebaut: Wir benötigen 7 Zustände $q_0, \ldots, q_6$, die dem Fortschritt der Erkennung des Wortes `kaffee` entsprechen – Zustand q_i signalisiert, dass die ersten i Zeichen von `kaffee` schon erkannt wurden und Zustand q_6 ist folglich akzeptierend. Falls in Zustand q_i das $(i+1)$-te Zeichen von `kaffee` gelesen wird geht es in Zustand q_i ansonsten zurück in den Anfangszustand q_0. Der Automat scheint auf den ersten Blick dem folgenden partiellen Automaten zu entsprechen, bei dem q_0 gleichzeitig als Start- und als Fangzustand fungiert:



Allerdings gibt es noch einen Sonderfall zu beachten: falls das fehlerhafte Zeichen zufälligerweise k war, geht es zurück nach q_1, weil mit `k` erneut das erste Zeichen des gesuchten Wortes gesehen worden sein kann. Diese Situation tritt ein, wenn beispielsweise das Wort *kafkaffee* gelesen wird. Von Zustand q_3 geht es mit k in Zustand q_1 weil das in q_3 fehlerhafte k der Beginn eines erfolgreichen Wortes sein kann. Der korrekte Automat ist damit:



Noch schwieriger wird die Sache, wenn wir stattdessen das Wort *kakao* erkennen wollen. Hier wird der Automat noch einen Tick komplizierter. Wenn beispielsweise nach dem Einlesen von kaka erneut das Zeichen k kommt, dürfen wir nicht zu Zustand q_1 übergehen, sondern in Zustand q_3.

2.6.2 Nichtdeterminismus

Wie diese einfachen Beispiele zeigen, kann eine Spezifikation durch endliche Automaten durchaus schwierig und fehleranfällig sein. Ein Problem liegt darin, dass jeder Zustand q für jedes mögliche Inputzeichen e nur eine Transition $\delta(q, e)$ hat. Würden wir diese Beschränkung lockern, dann wäre δ nicht mehr eine Funktion

$$\delta : Q \times \Sigma \to Q,$$

sondern eine Relation

$$\Delta \subseteq Q \times \Sigma \times Q.$$

Alternativ können wir δ durch eine Funktion in die Potenzmenge von Q ersetzen

$$\tau : Q \times \Sigma \to \mathbb{P}(Q),$$

die jedem Paar (q, e) nicht einen Zustand, sondern eine Menge von Zuständen zuordnet. Die Übersetzung ist:

$$(q, e, p) \in \Delta \iff p \in \tau(q, e).$$

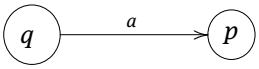
Damit haben wir den Begriff des Nichtdeterministischen Automaten gewonnen:

Definition 2.6.1 (Nichtdeterministischer Automat)**.** Ein *Nichtdeterministischer Automat* $\mathcal{A} = (Q, \Sigma, \tau, q_0, F)$ zu einem Alphabet Σ besteht aus einer endlichen Menge Q von Zuständen, einem Startzustand $q_0 \in Q$, einer Menge $F \subseteq Q$ von Endzuständen und einer Abbildung

$$\tau : Q \times \Sigma \to \mathbb{P}(Q)$$

in die Potenzmenge $\mathbb{P}(Q)$.

Statt $p \in \tau(q, a)$ schreiben wir auch $(q, a, p) \in \Delta$ und stellen dies wie vorher graphisch dar durch



Der Unterschied zu einem deterministischen Automaten ist lediglich, dass p durch q und a nicht eindeutig bestimmt ist. Es kann weitere p' geben, so dass $(q, a, p) \in \Delta$ und auch $(q, a, p') \in \Delta$ gilt, andererseits muss nicht zu jedem q, a ein entsprechendes p existieren. Das ist der Fall, wenn $\tau(q, a) = \{\}$ ist.

Definition 2.6.2. [Läufe und Akzeptanz] Ein *Lauf* mit einem Wort $w = w_1 \ldots w_n \in \Sigma^*$ ist eine Folge von Zuständen $q_0, \ldots q_n$, so dass $(q_i, w_{i+1}, q_{i+1}) \in \Delta$ gilt für $0 \leq i < n$. Ein Wort w wird *akzeptiert*, falls es einen Lauf mit w gibt, der im Anfangszustand startet und dessen letzter Zustand akzeptierend ist. Die *Sprache* eines Nichtdeterministischen Automaten ist die Menge aller Worte, die akzeptiert werden.

Einen Lauf von q nach q_n mit dem Wort $w = w_1 w_2 \ldots w_n$ stellen wir als

$$q \xrightarrow{w_1} q_1 \xrightarrow{w_2} \ldots \xrightarrow{w_{n-2}} q_{n-1} \xrightarrow{w_n} q_n$$

dar oder als

$$q \xrightarrow{w} q_n \ .$$

Definition 2.6.3. Ein Wort w wird *akzeptiert*, falls es einen Lauf mit w gibt, dessen letzter Zustand akzeptierend ist. Die *Sprache* eines nichtdeterministischen Automaten ist die Menge aller Worte, die akzeptiert werden. Formal:

$$L(\mathcal{A}) := \{w \in \Sigma^* \mid \text{es gibt einen Lauf mit } w \text{ von } q_0 \text{ zu einem } q \in F\}.$$

Als erstes Beispiel betrachten wir die Aufgabe, einen Nichtdeterministischen Automaten zu spezifizieren, der ein gewünschtes Wort, demonstriert an dem Wort kakao, in einem Text finden kann:

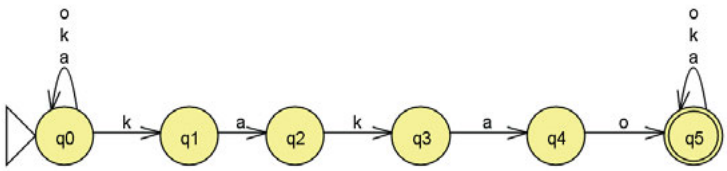


Abb. 2.6.1: Nichtdeterministischer Automat zur Erkennung eines Wortes

Wir ignorieren hier der Einfachheit halber alle Zeichen, die nicht in dem gesuchten Wort vorkommen, wählen also $\Sigma = \{k, a, o\}$. Ein Wort w enthält genau dann das Wort kakao, wenn w einen Lauf in einen Endzustand hat. Die Sprache des Automaten ist daher genau $\Sigma^* kakao \Sigma^*$. Nichtdeterminismus liegt hier in zwei Formen vor: Aus dem Anfangszustand q_0 führen zwei k-Transitionen heraus, d.h. $\delta(q_0, k) = \{q_0, q_1\}$ und die Zustände $q_1 \ldots q_4$ haben nicht-definierte Transitionen. Beispielsweise gilt $\tau(q_1, a) = \{\}$, etc..

Offensichtlich sind deterministische endliche Automaten Spezialfälle nichtdeterministischer Automaten. Ein deterministischer Automat ist ein nichtdeterministischer Automat bei dem $\tau(q, e)$ stets einelementig ist. Mit dieser Übersetzung stimmt auch

die Sprache des deterministischen Automaten mit der Sprache des zugehörigen nicht-deterministischen Automaten überein.

2.6.3 Erreichbarkeit

Ein Zustand q eines Automaten ist *erreichbar*, wenn es ein Wort w und einen Lauf l mit dem Wort w vom Anfangszustand in den Zustand q gibt. Es ist nützlich, die Menge aller Zustände, die von einer gegebenen Zustandsmenge aus mit einem Wort w erreicht werden kann, zu betrachten:

Sei $w \in \Sigma^\star$. Wir definieren $Q(w)$ durch Induktion über den Aufbau von w. Dabei ist es natürlicher, sich den Aufbau von w durch fortgesetztes Anfügen von Zeichen auf der rechten Seite vorzustellen, also $w = \varepsilon$ oder $w = u \cdot a$ mit $u \in \Sigma^\star$ und $a \in \Sigma$:

$$\begin{aligned} Q(\varepsilon) &:= q_0 \\ Q(u \cdot a) &:= \bigcup \{\tau(q, a) \mid q \in Q(u)\}. \end{aligned}$$

Die letzte Gleichung besagt, dass mit einem Wort $u \cdot a$ ein Zustand p erreicht werden kann, wenn zunächst mit Hilfe von u ein (Zwischen-)Zustand q erreicht werden kann, so dass $p \in \tau(q, a)$ ist. Mit dieser Definition kann man die Sprache eines nicht-deterministischen Automaten auch so ausdrücken:

$$L(\mathcal{A}) = \{w \in \Sigma^\star \mid Q(w) \cap F \neq \emptyset\}.$$

2.6.4 Spezifizieren mit nicht-deterministischen Automaten.

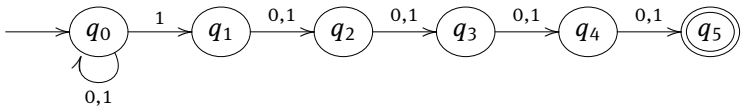
Nichtdeterministische Automaten bieten große Vorteile für die Spezifikation von Sprachen. Wir haben dies bereits am Beispiel der „Kakao“-Sprache gesehen. Weitere Beispiele liefern die Sprachen L_k über dem Alphabet $\Sigma = \{0, 1\}$ die aus allen Worten bestehen, deren k–letztes Symbol eine 1 ist.

L_1 besteht also aus allen Worten, welche mit 1 enden, L_2 aus allen Worten, deren vorletztes Zeichen eine 1 ist. Für L_1 und L_2 lassen sich noch leicht deterministische endliche Automaten $\mathcal{A}_1$ und $\mathcal{A}_2$ finden. $\mathcal{A}_1$ hat 2 Zustände, $\mathcal{A}_2$ hat 4 und $\mathcal{A}_5$ wird mindestens $32 = 2^5$ Zustände haben.

Die Menge aller k-stelligen Binärzahlen ist eine L_k–trennbare Menge von Worten. Seien nämlich u und v zwei verschiedene k-stellige Binärzahlen. Sei $1 \leq i \leq k$ eine Position, an der sie sich unterscheiden, beispielsweise sei $u(i) = 0$ und $v(i) = 1$. Dann sind u und v trennbar bezüglich L_k. Hängen wir nämlich ein beliebiges Wort w der Länge $|w| = i - 1$ an, so ist $uw \notin L_k$ aber $vw \in L_k$.

Für den Fall $k = 5$ und die Worte $u = 10\mathbf{0}10$ und $v = 10\mathbf{1}11$, die sich u.a. an der 2-ten Stelle unterscheiden, haben wir z.B. $w = 00$ als trennendes Wort, denn $uw = 10\mathbf{0}1000 \notin L_5$, aber $vw = 10\mathbf{1}1100 \in L_5$.

Diese Überlegung zeigt, dass jeder Automat für L_k mindestens 2^k Zustände haben muss. Wir sparen es uns daher, einen deterministischen Automaten für L_5 zu spezifizieren. Für einen nichtdeterministischen Automaten reichen 6 Zustände statt 2^5, und einen solchen Automaten können wir leicht angeben:



2.7 Von nicht-deterministischen zu deterministischen Automaten

Wir zeigen jetzt, dass man aus jedem nichtdeterministischen Automaten $\mathcal{A}_{nd}$ einen deterministischen Automaten $\mathcal{A}_d$ gewinnen kann, der die gleiche Sprache erkennt. Das vorige Beispiel zeigt, dass wir nicht verhindern können, dass dabei die Anzahl der benötigten Zustände möglicherweise exponentiell anwächst.

Die Idee ist ganz einfach. Während der Erkennung eines Wortes w befindet sich ein deterministischer Automat $\mathcal{A}_d$ jederzeit in genau einem bestimmten Zustand. Ein nichtdeterministischen Automaten kann sich in einem von mehreren Zuständen befinden. Diese Menge von möglichen Zuständen, in denen wir sein könnten, werden wir nun als Zustand des zugehörigen deterministischen Automaten erklären.

Ein Zustand dieses deterministischen Automaten ist damit eine *Zustandsmenge* des nichtdeterministischen Automaten. Ist Q die Menge aller Zustände von A_{nd} dann setzen wir also $\mathbb{P}(Q)$, die Potenzmenge von Q als Zustandsmenge des deterministischen Automaten an. Die Konstruktion heißt daher auch *Potenzmengenkonstruktion*.

Definition 2.7.1 (Potenzmengenautomat)**.** Sei $\mathcal{A} = (Q, \Sigma, \tau, q_0, F)$ ein nichtdeterministischer Automat. Den zugehörigen Potenzmengenautomaten definieren wir als

$$\mathbb{P}(\mathcal{A}) = (\mathbb{P}(Q), \Sigma, \delta, \{q_0\}, \Diamond F),$$

wobei

$$\delta(S, a) := \bigcup \{\tau(s, a) \mid s \in S\} \text{ für } S \in \mathbb{P}(Q) \text{ und } a \in \Sigma \tag{2.7.1}$$

$$\Diamond F := \{S \in \mathbb{P}(Q) \mid S \cap F \neq \emptyset\}. \tag{2.7.2}$$

Für die praktische Konstruktion des Potenzmengenautomaten wird man nicht mit der Zustandsmenge $\mathbb{P}(Q)$ beginnen, sondern stattdessen, beginnend mit dem Anfangszustand $\{q_0\}$ lediglich alle erreichbaren Zustände schrittweise konstruieren.

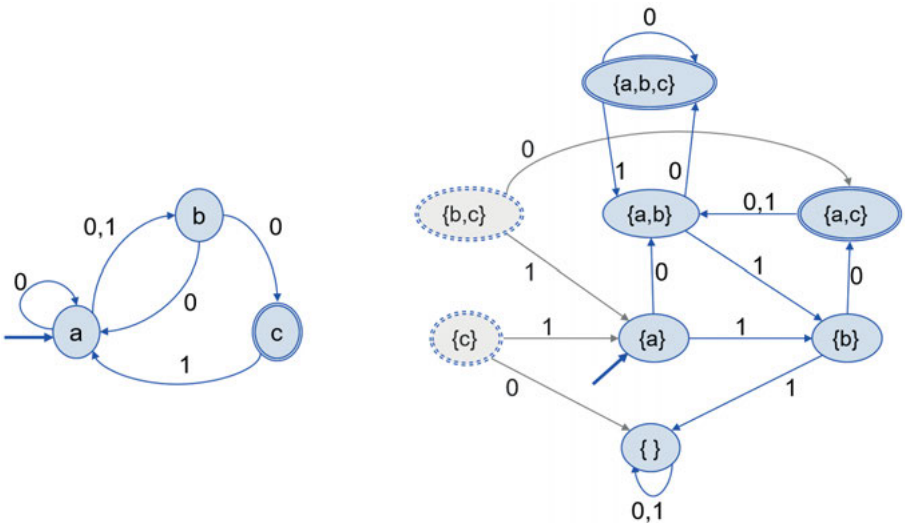


Abb. 2.7.1: Potenzmengenkonstruktion

Beispielsweise ist der Zustand $\{\} \in \mathbb{P}(Q)$ nie erreichbar. Abbildung 2.7.1 zeigt links einen nichtdeterministischen Automaten und rechts den zugehörigen (deterministischen) Potenzmengenautomaten, wobei die nicht-erreichbaren Zustände ausgegraut sind.

Der Potenzmengenautomat ist ein deterministischer Automat, und wir zeigen:

Satz 2.7.2. *Sei $\mathcal{A}$ ein beliebiger endlicher nichtdeterministischer Automat mit n Zuständen. Der zugehörige Potenzmengenautomat $\mathbb{P}(\mathcal{A})$ ist ein deterministischer endlicher Automat mit höchstens 2^n Zuständen, der die gleiche Sprache erkennt.*

Beweis. Für $q \in Q$ und ein beliebiges $w \in \Sigma^*$ zeigen wir durch Induktion nach w:

$$\forall S \subseteq Q.\, (q \in \delta^*(S, w) \iff \exists s \in S.\, s \xrightarrow{w} q)\ . \tag{2.7.3}$$

$\underline{w = \varepsilon}$:

$$q \in \delta^*(S, \varepsilon) \iff q \in S \iff \exists s \in S.\, s \xrightarrow{\varepsilon} q\ .$$

$\underline{w = au}$:

$$\begin{aligned} q \in \delta^*(S, au) &\overset{\text{Def. } \delta^*}{\iff} q \in \delta^*(\delta(S, a), u) \\ &\overset{\text{Ind. Hyp. 2.7.3}}{\iff} \exists s' \in \delta(S, a).\, s' \xrightarrow{u} q \\ &\overset{\text{Def. } \delta \text{ 2.7.1}}{\iff} \exists s \in S. \exists s' \in \tau(s, a).\, s' \xrightarrow{u} q \\ &\overset{\text{Def. 2.6.2}}{\iff} \exists s \in S.\, s \xrightarrow{au} q. \end{aligned}$$

Wenden wir (2.7.3) auf $S = \{q_0\}$, den Anfangszustand des Potenzmengenautomaten an, so folgt für jedes $q \in Q$ und jedes $w \in \Sigma^*$:

$$q \in \delta^*(\{q_0\}, w) \iff q_0 \xrightarrow{w} q \ .$$

Damit folgt schließlich

$$\begin{aligned} w \in L(\mathbb{P}(\mathcal{A})) &\iff \delta^*(\{q_0\}, w) \in \Diamond F \\ &\iff \delta^*(\{q_0\}, w) \cap F \neq \emptyset \\ &\iff \exists q \in F.\, q \in \delta^*(\{q_0\}, w) \\ &\iff \exists q \in F.\, \exists s \in \{q_0\}.\, s \longrightarrow q \\ &\iff \exists q \in F.\ q_0 \xrightarrow{w} q \\ &\iff w \in L(\mathcal{A}). \end{aligned}$$

□

Nichtdeterministische Automaten geben uns mehr Freiheit beim Spezifizieren, deterministische Automaten lassen sich leicht (tabellengesteuert) implementieren. Der gerade gezeigte Äquivalenzsatz kombiniert damit das beste aus beiden Welten.

Aufgabe 2.7.3. Beweisen Sie Satz 2.7.2, indem Sie den Begriff der Erreichbarkeit (Abschnitt 2.6.3) verwenden. Für die notwendige Induktion stellen Sie sich w als von rechts aufgebaut vor, betrachten Sie also die Fälle $w = \varepsilon$ und $w = u \cdot a$ mit $u \in \Sigma^*$ und $a \in \Sigma$.

2.7.1 ε-Transitionen

Eine Schwierigkeit gilt es noch zu überwinden. Wenn wir Automaten $\mathcal{A}_1$ für die Sprache L_1 und $\mathcal{A}_2$ für L_2 haben, wie finden wir einen Automaten für $L_1 \circ L_2$? Das sollte eigentlich möglich sein, und am einfachsten wäre es, wenn wir von jedem beliebigen akzeptierenden Zustand von $\mathcal{A}_1$ ohne weiteres, also ohne ein weiteres Zeichen zu verbrauchen, in den Anfangszustand von $\mathcal{A}_2$ springen könnten. Solche Sprungmöglichkeiten schaffen wir uns jetzt mit sogenannten ε-Transitionen. Sie erlauben den Sprung von einem Zustand in gewisse andere, ohne dabei ein Zeichen zu konsumieren. Dies erreichen wir technisch dadurch, dass wir die Transitionsfunktion auf $\Sigma \cup \{\varepsilon\}$ ausdehnen. $\tau(q, \varepsilon)$ liefert dann die Menge aller Zustände, in die wir ausgehend von Zustand q „spontan“ springen können.

Definition 2.7.4. Ein ε-Automat $\mathcal{A} = (Q, \Sigma, \tau, q_0, F)$ besitzt eine Transitionsfunktion $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to \mathbb{P}(Q)$ und ist ansonsten definiert wie ein nichtdeterministischer Automat. Mit einer Zustandsmenge $S \subseteq Q$ ist automatisch auch jeder weitere Zustand erreichbar, der mit Hilfe einer oder mehrerer ε-Transitionen erreicht werden kann.

Man definiert daher die ε–Hülle $\varepsilon(S)$ einer Zustandsmenge S als kleinste Zustandsmenge, die S umfasst und die gegen ε-Transitionen abgeschlossen ist:

$$\begin{aligned} S &\subseteq \varepsilon(S) \\ q \in S &\Longrightarrow \tau(q, \varepsilon) \subseteq \varepsilon(S). \end{aligned}$$

Ein *ε–Lauf* für ein Wort $w = w_1 w_2 \ldots w_n$ von q zu q_n darf jetzt neben den Transitionen für die einzelnen Zeichen von w auch noch beliebig viele ε–Transitionen beinhalten, beispielsweise:

$$q \xrightarrow{\varepsilon} q_1 \xrightarrow{w_1} q_2 \xrightarrow{\varepsilon} q_3 \xrightarrow{\varepsilon} q_4 \xrightarrow{w_2} \ldots \xrightarrow{\varepsilon} q_{m-2} \xrightarrow{w_n} q_{m-1} \xrightarrow{\varepsilon} q_m$$

Definition 2.7.5. Die Sprache des nichtdeterministischen ε–Automaten ist die Menge aller Worte w, für die es einen ε–Lauf vom Anfangszustand q_0 in einen Endzustand gibt.

Auch hier können wir wieder zeigen, dass es einen deterministischen Automaten gibt, der die gleiche Sprache erkennt. Es handelt sich wieder um die Potenzmengenkonstruktion. Als Variation zu der bereits gesehenen Konstruktion betrachten wir diesmal von vornherein nur erreichbare Zustandsmengen. Dazu definieren wir für beliebige Worte $w \in \Sigma^\star$:

$$Q(w) := \{q \in Q \mid \text{es gibt einen } \varepsilon\text{-Lauf für } w \text{ von } q_o \text{nach } q\}.$$

Aus der Definition folgt sofort:

$$Q(\varepsilon) = \varepsilon(\{q_0\}) \tag{2.7.4}$$

$$Q(w) = Q(w') \implies Q(w \cdot a) = Q(w' \cdot a) \tag{2.7.5}$$

$$Q(w) \cap F \neq \emptyset \iff w \in L(\mathcal{A}). \tag{2.7.6}$$

Das nächste Resultat verallgemeinert Satz 2.7.2. Der konstruierte endliche Automat ist deterministisch und bereits minimal. Allerdings ist seine Zustandsmenge nicht mehr so einfach bestimmbar:

Satz 2.7.6. *Zu jedem endlichen nichtdeterministischen Automaten $\mathcal{A} = (Q, \Sigma, \tau, q_0, F)$ mit ε–Übergängen gibt es einen deterministischen endlichen Automaten $\mathcal{A}_d$, der die gleiche Sprache erkennt.*

Beweis. Definiere $\mathcal{A}_d = (Z, \Sigma, \delta, z_0, T)$ mit $Z := \{Q(w) \mid w \in \Sigma^\star\}$ sowie $z_0 = Q(\varepsilon)$ und

$$T := \{Q(w) \mid Q(w) \cap F \neq \emptyset\}. \tag{2.7.7}$$

Die Transitionsfunktion $\delta : Z \times \Sigma \to Z$ definieren wir durch

$$\delta(Q(w), a) := Q(w \cdot a). \tag{2.7.8}$$

Durch Induktion über den Aufbau von w zeigen wir zunächst:

$$\forall w \in \Sigma^\star . \forall v \in \Sigma^\star . \, \delta^\star(Q(v), w) = Q(vw). \tag{2.7.9}$$

[Induktionsanfang] $w = \varepsilon$:

$$\delta^\star(Q(v), \varepsilon) = Q(v) = Q(v\varepsilon)$$

[Induktionsschritt] $w = a \cdot u$:

$$\begin{aligned}
\delta^\star(Q(v), a \cdot u) &\overset{Def\ \delta^\star}{=} \delta^\star(\delta(Q(v), a), u) \\
&\overset{Def\ 2.7.8}{=} \delta^\star(Q(v \cdot a), u) \\
&\overset{Ind.Hyp}{=} Q((v \cdot a)\, u) \\
&\overset{Assoziativität}{=} Q(v\,(a \cdot u)).
\end{aligned}$$

Jetzt müssen wir nur noch die Definition von $L(\mathcal{A})$ entwickeln:

$$\begin{aligned}
w \in L(\mathcal{A}_d) &\iff \delta^\star(z_0, w) \in T \\
&\iff \delta^\star(Q(\varepsilon), w) \in T \\
&\iff Q(\varepsilon w) \in T \\
&\iff Q(w) \in T \\
&\iff Q(w) \cap F \neq \emptyset \\
&\iff w \in L(\mathcal{A})
\end{aligned}$$

□

Wir haben jetzt also die Freiheit sowohl Nichtdeterminismus als auch ε–Transitionen für die Spezifikation von Automaten zu verwenden und wir haben Methoden gesehen daraus einen deterministischen Automaten zu gewinnen. Wir wollen dies an zwei Beispielen ausprobieren. Angenommen, wir wollen einen von zwei Strings s_1 oder s_2 in einem Text finden. Für diese Strings haben wir je einen Automaten $\mathcal{A}_1$ und $\mathcal{A}_2$. Nun fügen wir einen neuen Anfangszustand s hinzu mit ε–Transitionen in die (ehemaligen) Anfangszustände von $\mathcal{A}_1$ und $\mathcal{A}_2$.

Bereits angedeutet hatten wir die Konstruktion eines Automaten für die Sprache $L_1 \circ L_2$, wenn für L_1 und L_2 die Automaten $\mathcal{A}_1$ und $\mathcal{A}_2$ vorliegen. Von jedem Endzustand von $\mathcal{A}_1$ ziehen wir eine ε–Transition in den Anfangszustand von $\mathcal{A}_2$. Dessen Endzustände werden die Endzustände des kombinierten Automaten. Im folgenden Abschnitt werden wir der Frage systematischer nachgehen.

2.8 Automaten für reguläre Ausdrücke

Wir hatten zu Beginn des Kapitels reguläre Ausdrücke und die zugehörigen Sprachen kennengelernt. In diesem Abschnitt werden wir zeigen, dass zu jedem regulären Ausdruck ein deterministischer endlicher Automat (*DFA*) konstruiert werden kann, der genau die Sprache akzeptiert, welche von dem regulären Ausdruck beschrieben wird. Wir werden also beweisen:

Satz 2.8.1. *[Kleene]Zu jedem regulären Ausdruck r gibt es einen deterministischen endlichen Automaten* $\mathcal{A}_r$ *mit* $L(\mathcal{A}_r) = [\![r]\!]$.

Beweis. Zu jedem regulären Ausdruck r konstruieren wir einen ε–NFA $\mathcal{A}_r$ mit genau einem Endzustand.

Für die konstanten regulären Ausdrücke $\mathbf{0}$, $\mathbf{1}$ und $\boldsymbol{a}$ für $a \in \Sigma$ ist die Sache einfach. Der zugehörige Automat hat keine Transition, bzw. nur die Transition a oder ε die jeweils vom Anfangs- zum Endzustand führen.

Abb. 2.8.1: Automaten für triviale reguläre Ausdrücke

Seien jetzt Automaten für reguläre Ausdrücke e und f gegeben, so dass $L(\mathcal{A}_e) = [\![e]\!]$ und $L(\mathcal{A}_f) = [\![f]\!]$. Wir können annehmen, dass die Zustandsmengen der beteiligten Automaten disjunkt sind. Die folgenden Diagramme zeigen, wie wir Automaten für $e + f$ und für $(e \circ f)$ gewinnen können. Im ersten Fall werden die vormals terminalen Zustände von $\mathcal{A}_e$ und $\mathcal{A}_f$ zu nicht-terminalen Zuständen. Dann werden ein neuer Anfangszustand und ein neuer Endzustand hinzugefügt, die mit den vormaligen Anfangs- bzw. Endzuständen durch ε-Transitionen verbunden werden. Im zweiten Fall werden die terminalen Zustände von $\mathcal{A}_e$ als nicht-terminal erklärt und ε-Transition führen von dort in den Anfangszustand von $\mathcal{A}_f$.

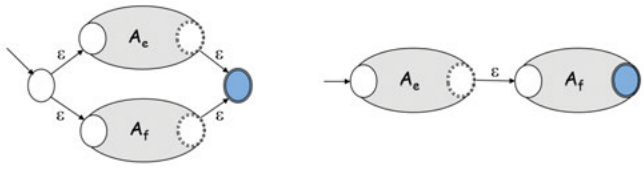


Abb. 2.8.2: ε-NFAs für reguläre Ausdrücke: $\mathcal{A}_{e+f}$ und $\mathcal{A}_{e\circ f}$

Es bleibt noch die Aufgabe, aus einem Automaten $\mathcal{A}_e$ mit $L(\mathcal{A}_e) = [\![e]\!]$ einen Automaten für $e^\star$ zu gewinnen. Die Lösung zeigt die folgende Figur. Es bedarf eines neuen Zustands, der sogleich Anfangs- und Endzustand wird. Von den vormaligen Endzuständen führt jeweils eine ε-Transition in diesen neuen Zustand, und von dort eine weitere ε-Transition in den vormaligen Anfangszustand.

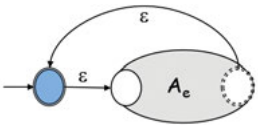


Abb. 2.8.3: ε-NFA für $\mathcal{A}_{e^\star}$

□

2.8.1 Anwendung regulärer Ausdrücke in Python

Der Übergang von regulären Ausdrücken zu Automaten hat durchaus praktische Relevanz. Die Python-Methode `re.compile` erzeugt aus einem *pattern*, also einem regulären Ausdruck, eine Automatentabelle

$$delta = re.compile(pattern)$$

Dabei werden alle bisher genannten Operationen (z.T. in einer optimierten Version) ausgeführt:

1. Aus dem *pattern* wird wie oben ein ε-NFA erzeugt.
2. Mit Hilfe der Potenzmengenkonstruktion gewinnt man daraus einen deterministischen Automaten,
3. Der DFA wird minimiert.

Die so gewonnene Automatentabelle $\delta : Q \times \Sigma \to Q$ kann anschließend die schrittweise Abarbeitung eines Wortes w steuern. Um etwa festzustellen, ob ein Wort w zur Sprache $L(pattern)$ gehört, kann man dies mit der Funktion *fullmatch* testen:

$$fullmatch(delta, w).$$

2.9 Äquivalenz

Aus einem regulären Ausdruck r haben wir also einen nichtdeterministischen ε-Automaten $\mathcal{A}_r$ gewonnen, der die gleiche Sprache beschreibt, das heißt $[\![r]\!] = L(\mathcal{A}_r)$. In diesem Abschnitt zeigen wir, dass auch der umgekehrte Weg möglich ist, dass

man also zu einem beliebigen Automaten $\mathcal{A}$ einen regulären Ausdruck $r_{\mathcal{A}}$ gewinnen kann, so dass $L(\mathcal{A}) = [\![r_{\mathcal{A}}]\!]$ gilt. Dabei spielt es keine Rolle, ob $\mathcal{A}$ deterministisch oder nichtdeterministisch ist.

Im ersten Schritt bereiten wir den Automaten vor, so dass er keine nicht erreichbaren Zustände mehr enthält. Durch Hinzunahme eines neuen Endzustandes und entsprechenden ε-Transitionen können wir leicht erreichen, dass unser Automat stets die folgende Invariante *Inv* erfüllt:

Inv

- es gibt genau einen Startzustand und genau einen Endzustand
- Startzustand und Endzustand sind verschieden
- keine Transition erreicht den Startzustand oder verlässt den Endzustand.

Der Algorithmus eliminiert schrittweise alle inneren Knoten des Zustandsübergangsgraphen, bis nur noch der Startknoten s und der Endknoten t übrigbleiben. Dabei werden neue Kanten zwischen den verbleibenden Zuständen gezogen und mit regulären Ausdrücken beschriftet. Die Invariante *Inv* wird dabei nicht verletzt, sie ist auch noch gültig für den endgültigen Graphen, welcher dann nur noch aus dem Start- und Endknoten besteht und einer Kante, welche mit dem gesuchten regulären Ausdruck beschriftet ist. Im einzelnen sind die Schritte:

Knotenelimination: Sei k ein innerer Knoten.

- Für jeden Weg der Länge 2, der k benutzt, also jedes Paar (p, q) von Knoten, so dass Kanten $p \xrightarrow{e} k \xrightarrow{f} q$ existieren, füge eine neue Kante $p \xrightarrow{ef} q$ ein. Falls k eine Kante zu sich selbst besitzt, welche mit g beschriftet ist, beschrifte die Kante stattdessen mit eg^*f, also $p \xrightarrow{eg^*f} q$.
- Entferne anschließend k.

Elimination paralleler Kanten:

- Falls $p \xrightarrow{e} q$ und $p \xrightarrow{f} q$ zwei parallele Kanten zwischen denselben Knoten sind, eliminiere diese und ersetze sie durch die Kante $p \xrightarrow{(e+f)} q$.

Es ist leicht zu sehen, dass in jedem Schritt die Invariante eingehalten wird und dass der reguläre Ausdruck r zwischen zwei benachbarten Knoten p und q genau die Menge aller Worte codiert, die in dem vormaligen Automaten von p nach q führten.

Zum Schluss bleibt aufgrund der Invarianten genau eine Kante $s \xrightarrow{r} t$ vom Start- zum Endknoten übrig. Der reguläre Ausdruck r, mit dem diese beschriftet ist, ist der gesuchte reguläre Ausdruck mit $L(\mathcal{A}) = [\![r]\!]$. Dies ist aufgrund der Konstruktion intuitiv klar.

Abbildung 2.9.1 zeigt die Entfernung zweier Knoten aus einem Automatengraphen und das Entstehen regulärer Ausdrücke an den Kanten.

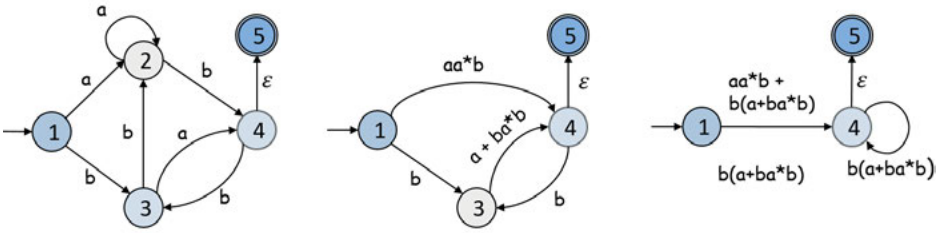


Abb. 2.9.1: Schrittweise Elimination zweier Knoten

Der vormalige Endknoten 4 ist bereits durch Hinzunahme eines neuen Endknotens (5) und einer ε-Transformation ersetzt worden, damit der Automat die Invariante *Inv* erfüllt. Im ersten Schritt wird Knoten 2 entfernt. Dabei müssen die Wege $1 \to 2 \to 4$ und $3 \to 2 \to 4$ durch Kanten von 1 nach 4 und von 3 nach 4 substituiert werden. Diese tragen die Label aa^*b bzw. ba^*b. Neben der bereits vorhandenen Kante $3 \xrightarrow{a} 4$ entsteht also eine weitere Kante $3 \xrightarrow{ba^*b} 4$. Diese zwei parallelen Kanten können mit „+“ zusammengefasst werden, so entsteht die mit $a + ba^*b$ beschriftete Kante von 3 nach 4.

Im nächsten Schritt wird Knoten 3 entfernt, was eine Kante von 1 nach 4 nach sich zieht und eine Schleife bei Knoten 4, die mit $b(a + ba^*b)$ beschriftet werden muss. Im allerletzten Schritt muss noch Knoten 4 eliminiert werden, so dass der Graph allein aus einer Kante vom Anfangs- zum Endknoten besteht. Dessen Beschriftung

$$(aa^*b + b(a + ba^*b))(b(a + ba^*b))^*$$

ist der gesuchte reguläre Ausdruck, der die Sprache des ursprünglichen Automaten beschreibt.

Für einen formalen Beweis der Korrektheit des Verfahrens kann man zu einem beliebigen Paar (p, q) von Knoten die Sprache $L_{p,q}$ definieren als die Menge aller Worte w, die von Knoten p zu Knoten q führen. Da zwischenzeitlich die Kanten mit regulären Ausdrücken e_i beschriftet sind bedeutet dies:

$$\begin{aligned} w \in L_{p,q} \iff & \quad \exists n \in \mathbb{N}, p_0, \ldots, p_n, e_1, .., e_n.\ w \in [\![e_1 e_2 \ldots e_{n-1} e_n]\!] \\ \text{mit} \quad & p = p_0 \xrightarrow{e_1} p_2 \xrightarrow{e_2} \ldots \xrightarrow{e_{n-1}} p_{n-1} \xrightarrow{e_n} p_n = q. \end{aligned}$$

Man überzeugt sich leicht, dass in jedem Schritt unserer Konstruktion $L_{p,q}$ nicht verändert wird. Insbesondere gilt zu Beginn $L(\mathcal{A}) = L_{s,t}$ und zum Schluss $L_{s,t} = [\![r]\!]$, woraus $L(\mathcal{A}) = [\![r]\!]$ folgt.

Wir können nun die Ergebnisse der letzten Abschnitte in folgendem Satz zusammenfassen:

Satz 2.9.1. *Für eine Sprache $L \subseteq \Sigma^*$ sind äquivalent:*
1. *L ist regulär (durch einen regulären Ausdruck definierbar).*
2. *L ist die Sprache eines ε-NFA.*
3. *L ist die Sprache eines DFA.*

In der Praxis sind vor allem die Implikationen $1. \to 2. \to 3.$ nützlich. Sie erlauben uns, zu einem gegebenen regulären Ausdruck r auf dem Umweg über einen nichtdeterministischen Automaten schließlich einen äquivalenten endlichen Automaten zu gewinnen, der dann tabellengesteuert Worte analysieren kann, um zu entscheiden, ob diese zu $[\![r]\!]$ gehören, oder nicht. In Python wird dies, wie gesehen, durch die Funktion `compile` aus dem Modul `re` realisiert.

Kapitel 3

Grammatiken und Stackautomaten

3.1 Der zweistufige Aufbau von Sprachen

3.1.1 Natürliche Sprachen

Natürliche Sprachen sind zweistufig aufgebaut

–die unterste Ebene bilden die *Wörter*.

–die nächste Ebene kombiniert die Wörter zu Sätzen.

Wörter kann man im Wörterbuch nachschlagen. Dabei werden sie klassifiziert, etwa als *Artikel*, *Hauptwort*, *Hilfsverb, Verb, etc.*.

Wie man Wörter korrekt zu Sätzen kombiniert wird durch die Grammatik der Sprache festgelegt. Auf dieser Ebene kommen Kategorien wie *Subjekt*, *Prädikat*, *Objekt* ins Spiel. Grob vereinfacht ist ein Satz dann entweder eine Folge

Subjekt Prädikat

oder

Subjekt Prädikat Objekt.

Ein *Subjekt* besteht aus einem *Artikel* und einem *Hauptwort* und ein *Prädikat* ist ein *Verb* oder ein *Hilfsverb*..

Der Satz „`Der Autor schreibt ein Buch`“ ist dann ein *Satz*, denn er hat die geforderte Struktur *Subjekt Prädikat Objekt*, wobei *Subjekt* und *Objekt* jeweils von der Form *Artikel Hauptwort* sind („`der Autor`“ und „`ein Buch`“) und „`schreibt`“ ist ein „*Verb*“.

Eine derart vereinfachte Grammatik kann man formal durch Regeln beschreiben:

https://doi.org/10.1515/9783110442397-004

$$
\begin{array}{rcll}
\mathit{Satz} & ::= & \mathit{Subjekt} \quad \mathit{Prädikat} & \\
 & | & \mathit{Subjekt} \quad \mathit{Prädikat} \quad \mathit{Objekt} & \\
 & & & \\
\mathit{Prädikat} & ::= & \mathit{Verb} & \\
 & | & \mathit{Hilfsverb} & \\
 & & & \\
\mathit{Subjekt} & ::= & \mathit{Artikel} \quad \mathit{Hauptwort} & \\
\mathit{Objekt} & ::= & \mathit{Artikel} \quad \mathit{Hauptwort} &
\end{array}
$$

Auf der linken Seite stehen die Begriffe, die definiert werden sollen. In unserem Falle sind dies *Satz*, *Prädikat*, S*ubjekt* und *Objekt*. Davon jeweils durch ein Definitionszeichen „: : =“ getrennt findet sich auf der rechten Seite die Definition als eine Folge von weiteren Begriffen. Falls es mehrere Alternativen gibt, werden diese durch das Zeichen `'|'` voneinander getrennt. Ein Satz ist also alternativ eine Folge *Subjekt Prädikat* oder eine Folge *Subjekt Prädikat Objekt*. Ein *Prädikat* ist ein *Verb* oder ein *Hilfsverb*. Sowohl *Subjekt* als auch *Objekt* bestehen aus einem *Artikel* gefolgt von einem *Hauptwort*.

Die Begriffe *Verb*, *Hilfsverb*, *Artikel* und *Hauptwort* werden nicht definiert. Aus der Sicht der Grammatik sind dies undefinierte Begriffe, die je nach Anwendung auch *Terminale* oder *Token* genannt werden. Für jedes dieser Token gibt es ein Klassifikationsprogramm, das entscheidet, ob ein vorgelegter String wie z.B. „schreibt“ ein *Artikel*, ein *Verb*, ein *Hilfsverb* oder ein *Hauptwort* ist.

Die *Erkennung* eines Satzes wie „`Der Autor schreibt ein Buch`“ beinhaltet die Analyse, ob die vorgelegte Wortfolge den Regeln der Grammatik entspricht. Sie beinhalte zwei Phasen:

Lexikalische Analyse: In dieser Phase werden die Wörter klassifiziert. Aus „`Der Autor schreibt ein Buch`“ wird die Tokenfolge „*Artikel Hauptwort Verb Artikel Hauptwort*“.

Syntaktische Analyse: In dieser zweiten Stufe, auch *grammatischen Analyse* genannt, wird anhand der Grammatikregeln versucht, diese Tokenfolge als *Satz* zu erkennen.

Die syntaktische Analyse versucht einen Baum aufzustellen, der den gesamten Programmtext und dessen syntaktische Zusammenhänge repräsentiert. Dabei kann man entweder mit der Wurzel beginnen und schrittweise (z.B. von links nach rechts) die Söhne der Wurzel aufbauen, dann deren Söhne, etc. oder man kann alternativ mit den Blättern beginnen und dann stets kleine Bestandteile (Teilbäume) zu größeren zusammenzufassen, bis alle relevanten Teile zu einem Baum zusammengewachsen sind. Die erste Vorgehensweise nennt man *top down*, die zweite *bottom up*. In jedem Fall hat man die Grammatik zur Verfügung sowie eine Tokenfolge, die einen Satz der

zugehörigen Sprache repräsentieren soll. Etwas präziser kann man den Unterschied so beschreiben:

bottom up: Man versucht in der zu analysierenden Tokenfolge die rechte Seite einer Regel zu erkennen. Diese Bestandteile ersetzt man durch die entsprechende linke Seite. Aus „*Artikel Hauptwort Verb Artikel Hauptwort*“ wird dabei in einem ersten Durchgang „*Subjekt Prädikat Objekt*“ und daraus in einem zweiten Durchgang „*Satz*“.

top down: Man beginnt mit dem gewünschten Ergebnis *Satz* und ersetzt dieses durch eine zugehörige rechte Seite, z.B. „*Subjekt Prädikat Objekt*“. Sodann ersetzt man „*Subjekt*“ durch „*Artikel Hauptwort*“, „*Prädikat*“ durch „*Verb*“ und „*Objekt*“ durch „*Artikel Hauptwort*“. Insgesamt ist die vermutete Struktur jetzt „*Artikel Hauptwort Verb Artikel Hauptwort*“ und dies ist in der Tat die zu „Der Autor schreibt ein Buch“ gehörende Tokenfolge.

3.1.2 Programmiersprachen

Programmiersprachen sind ähnlich zweistufig konstruiert wie auch natürliche Sprachen. Die sogenannten „*lexikalischen Bestandteile*“ entsprechen den Wörtern einer Sprache und eine *Grammatik* beschreibt deren syntaktisch korrekte Kombinationen zu Programmen. Zu den lexikalischen Bestandteilen einer Programmiersprache gehören u.a.

- die Variablennamen (z.B.: `x,y,betrag, wohn_ort`)
- Konstanten (`''Otto''`,`42`,`3.71E-10`)
- die Schlüsselwörter (**`if`**, **`else`**, **`while`**, etc.)
- Operatoren (+, -, *, /, %, **, etc.)
- Sonderzeichen (Klammern, Trennzeichen)

Jeder dieser Tokentypen wird durch einen regulären Ausdruck spezifiziert. In die nächste Phase gelangen nur diese Token, wie z.B. ***num*** (für Numeral), ***id*** (für identifier/Bezeichner), ***while*** für das Schlüsselwort 'while' bzw. 'WHILE', ***bOp*** für den binären Operator +, und ***(*** sowie ***)*** für die öffnende Klammer und schließende Klammer.

Eine *Grammatik* beschreibt dann, wie aus diesen lexikalischen Bestandteilen korrekte Programme gebildet werden können. Aus Sicht der Grammatik bilden die Token ihr Alphabet das wir mit T bezeichnen. In unserem Falle ist also

$$T = \{\boldsymbol{num}, \boldsymbol{id}, \boldsymbol{bOp}, \boldsymbol{(}, \boldsymbol{)}\}.$$

Die Aufgabe der Grammatik ist dann, festzulegen, ob ein vorgelegtes Wort aus T^* zur (Programmier-)Sprache gehört oder nicht.

Als Beispiel wollen wir eine einfache imperative Sprache *WHILE* diskutieren. Die Grammatik definiert, was als Programm zugelassen sein soll. Zunächst aber definieren wir nur die Syntax für arithmetische Ausdrücke (engl.: *expressions*):

Beispiel 3.1.1 (Expression Syntax)**.**

$$
\begin{array}{rcl}
\mathit{Expr} & ::= & \boldsymbol{num} \\
& | & \boldsymbol{id} \\
& | & \mathit{Expr}\ \ \boldsymbol{bOp}\ \ \mathit{Expr} \\
& | & \boldsymbol{(}\ \ \mathit{Expr}\ \ \boldsymbol{)}
\end{array}
$$

Ein Ausdruck (*Expr* für engl.: *expression*) ist also entweder eine Zahl (***num***), eine Variable (allgemeiner ein Bezeichner, engl.: ***id***entifier), eine Verknüpfung zweier *Expr* mittels einer zweistelligen Operation (engl.: ***bi***nary ***op***eration) oder ein geklammerter *Expr*. Ein Beispiel für einen Ausdruck ist etwa die Formel für einen mit Zinssatz z verzinsten Betrag :

„`betrag * (1 + z/100) ** jahre`“.

Die lexikalische Analyse klassifiziert die Bestandteile und liefert die Tokenfolge

$$\boldsymbol{id\ bOp\ (\ num\ bOp\ id\ bOp\ num\ )\ bOp\ id}.$$

Anhand der ersten beiden Zeilen der Grammatik sind ***num*** und ***id*** jeweils *Expr*, wir haben also eine Folge

$$\mathit{Expr}\ \boldsymbol{bOp}\ \boldsymbol{(}\ \mathit{Expr}\ \boldsymbol{bOp}\ \mathit{Expr}\ \boldsymbol{bOp}\ \mathit{Expr}\ \boldsymbol{)}\ \boldsymbol{bOp}\ \mathit{Expr}$$

vor uns.

Mit den Regeln der Grammatik sollen wir dies zu einem *Expr* reduzieren. Eine Grammatikregel besteht aus der linken Seite (hier *Expr*), einem Pfeil → und einer der alternativen rechten Seiten. Die Regeln der obigen Grammatik wären dann

$$\mathit{Expr} \to \boldsymbol{num},\ \mathit{Expr} \to \boldsymbol{id},\ \mathit{Expr} \to \mathit{Expr}\ \boldsymbol{bOp}\ \mathit{Expr},\ \mathit{Expr} \to (\ \mathit{Expr}\ ).$$

Eine *Reduktion* besteht darin, die rechte Seite einer Regel in dem Text zu finden und diese durch die linke Seite zu ersetzen.

Die dritte Grammatikregel können wir auf den Teilausdruck zwischen den Klammern zweimal anwenden, erhalten zunächst „**(** *Expr* ***bOp*** *Expr* **)**“ und durch erneute Anwendung „**(** *Expr* **)**“. Insgesamt haben wir die obige Tokenfolge also reduziert zu

$$\mathit{Expr}\ \boldsymbol{bOp}\ \boldsymbol{(}\ \mathit{Expr}\ \boldsymbol{)}\ \boldsymbol{bOp}\ \mathit{Expr}$$

Schließlich wenden wir die vierte Regel an und ersetzen „**(** *Expr* **)**“ durch *Expr*. Übrig bleibt

$$\mathit{Expr}\ \boldsymbol{bOp}\ \mathit{Expr}\ \boldsymbol{bOp}\ \mathit{Expr}$$

Zweimalige Anwendung der dritten Regel reduziert dies über *Expr* ***bOp*** *Expr* zu *Expr*. Wir sind am Ziel, denn wir haben festgestellt, dass „`betrag * (1 + z/100) ** jahre`“ tatsächlich ein gemäß der Grammatik gültiger Ausdruck (*Expr*) ist.

Wir könnten umgekehrt auch den gewünschten String gemäß den Grammatikregeln als Expr *ableiten*. Zur Ableitung von „`betrag * (1 + z/100) ** jahre`“ beginnen wir mit *Expr* und expandieren jeweils die im aktuellen Text gefundene linke Seite einer Regel durch ihre rechte Seite. Zur Illustration markieren wir jeden Ableitungsschritt durch die Nummer der verwendeten Regel:

$$
\begin{aligned}
\mathit{Expr} &\overset{3}{\Rightarrow} \mathit{Expr}\ \boldsymbol{bOp}\ \mathit{Expr} \\
&\overset{2}{\Rightarrow} \boldsymbol{id\ bOp}\ \mathit{Expr} \\
&\overset{3}{\Rightarrow} \boldsymbol{id\ bOp}\ \mathit{Expr}\ \boldsymbol{bOp}\ \mathit{Expr} \\
&\overset{4}{\Rightarrow} \boldsymbol{id\ bOp\ (}\ \mathit{Expr}\ \boldsymbol{)\ bOp}\ \mathit{Expr} \\
&\overset{3}{\Rightarrow} \boldsymbol{id\ bOp\ (}\ \mathit{Expr}\ \boldsymbol{bOp}\ \mathit{Expr}\ \boldsymbol{)\ bOp}\ \mathit{Expr} \\
&\overset{1}{\Rightarrow} \boldsymbol{id\ bOp\ (\ const\ bOp}\ \mathit{Expr}\ \boldsymbol{)\ bOp}\ \mathit{Expr} \\
&\overset{3}{\Rightarrow} \boldsymbol{id\ bOp\ (\ const\ bOp}\ \mathit{Expr}\ \boldsymbol{bOp}\ \mathit{Expr}\ \boldsymbol{)\ bOp}\ \mathit{Expr} \\
&\overset{1}{\Rightarrow} \boldsymbol{id\ bOp\ (\ const\ bOp\ id\ bOp\ const\ )\ bOp}\ \mathit{Expr} \\
&\overset{2}{\Rightarrow} \boldsymbol{id\ bOp\ (\ const\ bOp\ id\ bOp\ const\ )\ bOp\ id} \\
&\quad \texttt{betrag * (1 + z/100) ** jahre}
\end{aligned}
\tag{3.1.1}
$$

Im letzten Schritt haben wir zur Illustration die Token durch ihre Inhalte ersetzt, beispielsweise ***id*** durch `betrag`, `z`, `jahre`, und ***num*** durch 1 bzw. 100. Die Token **')'** und **'('** stehen allein für die schließenden und öffnenden Klammern „)“ und „(“.

Wir vervollständigen nun unsere Grammatik, um komplette Programme schreiben zu können:

Programm	::=	*Stmt*					
Stmt	::=	***id***	:=	*Expr*			
	|	*Stmt*	**;**	*Stmt*			
	|	**if**	*BExpr*	**then**	*Stmt*	**else**	*Stmt*
	|	**if**	*BExpr*	**then**	*Stmt*		
	|	**while**	*BExpr*	**do**	*Stmt*		
	|	***skip***					

Ein *Programm* ist also eine Anweisung (engl.: ***St**ate**m**ent*) und ein *Stmt* ist entweder eine Zuweisung eines Ausdrucks an eine Variable (z.B. `betrag := betrag + betrag*z/100`) oder die Hintereinanderausführung zweier Statements. Schließlich erlauben wir noch eine bedingte (if-then-else) Anweisung und eine **while**-Schleife, die durch eine Bedingung *BExpr* kontrolliert, ob ein *Stmt* (der sogenannte Körper der while-Schleife) ausgeführt werden soll, oder nicht. Das *skip* in der letzten Zeile steht für die leere rechte Seit, vergleichbar mit ε, dem leeren Wort. Mit ihm könnten wir beispielsweise die **if**-Anweisung ohne zugehöriges **else** realisieren als **if** BExpr

then Stmt **else** *skip*. Im Programmtext steht das Token ***skip*** für das leere Wort oder Leerzeichen, oder Kommentare.

Offen ist nur noch die Syntax für bedingte oder Boolesche Ausdrücke ***BExpr***. Diese liefern wir jetzt nach. Das Token ***relOP*** steht für einen der relationalen oder Vergleichs-Operatoren „=“, „<=“, „>=“, „<“, „>“, „!=“. Die Token **`and`**, **`or`** , **`not`**, **`true`** und **`false`** sind Schlüsselworte.

BExpr	::=	*Expr*	***relOp***	*Expr*
	|	*Bexpr*	**and**	*Bexpr*
	|	*BExpr*	**or**	*BExpr*
	|	**not**	*BExpr*	
	|	(	*BExpr*	)
	|	**true**		
	|	**false**		

3.2 Kontextfreie Grammatiken

Nachdem wir einige Anwendungen von Grammatiken gesehen haben, wollen wir diese auch formal definieren, um ihre Möglichkeiten und Grenzen zu erkennen. Zunächst einmal benötigen wir Grammatiken um Sprachen zu definieren. Aufgrund der gesehenen Anwendungen bezeichnet man das Alphabet über dem die Worte der Sprache gebildet werden, meist nicht mit Σ sondern mit T. Diese Bezeichnung soll daran erinnern, dass die „Zeichen“ in diesem Zusammenhang „Token“ sind. Die Sprache $L(G)$ die durch eine Grammatik G beschrieben wird soll dann eine Teilmenge von $T^\star$ sein, also $L(G) \subseteq T^\star$. Eine neutrale Deutung der Bezeichnung T bezeichnet diese als „Terminale“.

Außerdem verwendet eine Grammatik Namen für Begriffe und Hilfsbegriffe, die durch die Regeln definiert werden. Diese nennt man auch *Variablen* oder im Gegensatz zu den Terminalen als *Nonterminale*. Die Grammatik der While-Sprache verwendete als Variablen *Programm*, *Stmt*, *BExpr*, und *Expr* und als Token u.a. ***num***, ***id***, ***bOp***, „**;**“, **if**, ***skip***, etc.. Jede Regel der Grammatik besteht aus einer Variablen als linker Seite und einer Folge von Terminalen und Nichtterminalen, kurz einem Wort aus $(V \cup T)^\star$ als rechter Seite. Jede Regel ein Paar aus $V \times (V \cup T)^\star$. Damit sind wir auch schon bei der Definition:

Definition 3.2.1. Eine *kontextfreie*[1] *Grammatik* $G = (V, T, P, S)$ besteht aus
- einer endlichen Menge V von *Variablen,*
- einer endlichen Menge T von *Terminalen*, wobei $V \cap T = \emptyset$ vorausgesetzt ist,

1 Der Begriff „kontextfrei“ betont den Gegensatz zu allgemeineren „kontextabhängigen“ Grammatiken (siehe Abschnitt 3.9), bei denen die Ersetzung einer Variablen durch ihre rechte Seite auf bestimmte Kontexte eingeschränkt werden können.

– einer endlichen Menge $P \subseteq V \times (V \cup T)^*$ von *Produktionen,*
– einem Startsymbol $S \in V$.

Eine *Satzform* α ist ein Wort aus $(V \cup T)^*$. Eine Produktion (A, α) notiert man auch als $A \to \alpha$.

Beispiel 3.2.2. Die Grammatik der Binärzahlen ist $BZ = (V, T, P, S)$ mit

$$V = \{B\},\ T = \{0, 1\},\ P = \{B \to 0,\ B \to 1,\ B \to 0B,\ B \to 1B\} \text{ und } S = B.$$

Inhaltlich sind die Produktionen des vorigen Beispiels leicht zu interpretieren als

> 0 und 1 sind Binärzahlen, und wenn B eine Binärzahl ist, so erhält man durch Voranstellen von 0 oder 1 wieder eine Binärzahl.

Das folgende Beispiel liefert eine alternative Grammatik für arithmetische Ausdrücke. Wir beschränken uns auf die Operatoren '+' und '*'.

Beispiel 3.2.3. $G_{AExp} := (T, V, P, S)$ mit
$T = \{\ \textbf{\textit{id}},\ \textbf{\textit{num}}, +, *, \textbf{)}, \textbf{(}\ \}$,
$V = \{AExpr,\ Term,\ Faktor\}$ und
$P = \{AExpr \to Term,\ AExpr \to AExpr + Term,\ Term \to Factor, Term \to Term * Factor,\ Factor \to id, Factor \to num, Factor \to (Expr)\}$.

Übersichtlicher wird eine Grammatik, wenn wir die Produktionen für das gleiche Nonterminal zusammenfassen und die rechten Seiten durch ein Alternativzeichen trennen.

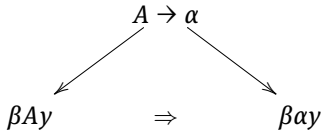
$$\begin{array}{rcl} AExpr & ::= & AExpr + Term \\ & | & Term \\ Term & ::= & Term * Factor \\ & | & Factor \\ Factor & ::= & (AExp) \\ & | & id \\ & | & num \end{array}$$

Mit der Konvention, dass
– das *Startsymbol* die linke Seite der ersten Regel ist und dass
– *Nichtterminale* genau die Namen sind, welche links auftauchen,
– *Terminale* die übrigen Namen,

beinhaltet diese Darstellung alle für eine Grammatik benötigten Informationen. Im Folgenden werden wir Grammatiken auch nur noch auf diese Art darstellen.

3.2.1 Ableitungen

Jede Produktion $A \to \alpha$ erlaubt es in einer Satzform $\beta A y$ das Nonterminal A durch α zu ersetzen. Aus $\beta A y$ erhalten wir dann mittels der Produktion $A \to \alpha$ die neu Satzform $\beta\alpha y$. Wir schreiben dann $\beta A y \implies \beta\alpha y$.



Die transitive Hülle von $\Rightarrow$ bezeichnen wir mit $\overset{\star}{\Rightarrow}$. Konkret definieren wir für beliebige Satzformen σ und τ:

Definition 3.2.4. $\sigma \overset{\star}{\Rightarrow} \tau :\Leftrightarrow \exists n.\exists \alpha_1, \ldots, \alpha_n.\, \sigma = \alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \Rightarrow \alpha_n = \tau$. In diesem Fall sagen wir, dass τ aus σ *ableitbar* ist.

Eine *Ableitung* $Expr \overset{\star}{\Rightarrow}$ ***id binOp (const binOp id binOp const) binOp id*** haben wir in 3.1.1 bereits gesehen. Die gleiche Satzform hätten wir auch auf andere Weise ableiten können. Schon im zweiten Schritt hätten wir statt dem ersten *Expr* auch das zweite *Expr* in *Expr binOp Expr* expandieren können. Stattdessen haben wir stets das von links gesehen erste Nonterminal expandiert. Bei einer solchen Ableitung spricht man auch von einer Linksableitung. Im Ergebnis spielt es keine Rolle, ob eine Linksableitung gewählt wird oder nicht. Eine umso größere Rolle spielt die Auswahl der Regeln für die Expansion eines Nonterminals. Will man z.B. den Ausdruck $x * (1 + y)$ in Form der Tokenfolge ***id binOp (const binOp id)*** erkennen, so kann die Ableitung

$$\begin{aligned} Expr &\Rightarrow Expr\ \mathbf{binOp}\ Expr \\ &\Rightarrow \mathbf{(}\ Expr\ \mathbf{)}\ binOp\ Exp \\ &\Rightarrow \ldots \end{aligned}$$

keinesfalls mehr zum Ziel führen.

Definition 3.2.5. Die *Sprache* einer Grammatik $G = (V, T, P, S)$ besteht aus allen Satzformen, die aus dem Startsymbol mit Hilfe der Produktionen abgeleitet werden können, und die nur aus Terminalen bestehen, kurz

$$\mathcal{L}(G) := \{u \in T^\star \mid S \overset{\star}{\Rightarrow} u\}.$$

Eine Sprache $L \subseteq T^*$ heißt *kontextfrei*, falls es eine kontextfreie Grammatik G gibt mit $L = \mathcal{L}(G)$,

Beispiel 3.2.6. Die folgenden Sprachen kann man jeweils durch eine kontextfreie Grammatik definieren:

1. $L_{a^n b^n} = \{a^n b^n \mid n \in \mathbb{N}\}$:

$$\begin{array}{rcl} S & ::= & a\,S\,b \\ & | & \varepsilon \end{array}$$

2. Die Dyck-Sprache der wohlgeformten Klammerausdrücke:

$$\begin{array}{rcl} S & ::= & \varepsilon \\ & | & S\,S \\ & | & (\,S\,) \end{array}$$

3. $L = \{a^m b^n \mid m \geq n\}$:

$$\begin{array}{rcl} S & ::= & AU \\ A & ::= & aA \\ & | & \varepsilon \\ U & ::= & aUb \\ & | & \varepsilon \end{array}$$

Von den im obigen Beispiel gerade gezeigten Sprachen hatten wir gesehen, dass sie nicht regulär sind, also weder durch einen regulären Ausdruck definierbar sind, noch von einem endlichen Automaten – gleichgültig ob deterministisch oder nicht – erkannt werden können. Somit scheinen kontextfreie Grammatiken mächtiger zu sein als reguläre Ausdrücke. Um dies zu einzusehen, müssen wir nur noch zeigen, dass jede reguläre Sprache auch kontextfrei ist. Dies lässt sich durch eine einfache Induktion über den Aufbau regulärer Sprachen zeigen und wir empfehlen dies dem Leser als Übung:

Aufgabe 3.2.7. Zeigen Sie durch Induktion, dass jede reguläre Sprache auch kontextfrei ist. Konstruieren Sie dazu zunächst kontextfreie Grammatiken für die Sprachen $\emptyset$, $\{\varepsilon\}$ und $\{a\}$ für $a \in Sigma$. Nehmen Sie als dann an, dass Sie schon kontextfreie Grammatiken für die Sprachen L_1 und L_2 haben. Zeigen Sie, wie Sie daraus Grammatiken für $L_1 L_2$,$L_1 \cup L_2$ und L_1^* konstruieren können.

Wir können sogar jedem regulären Ausdruck eine kontextfreie Grammatik einer bestimmten einfachen Bauweise zuordnen:

Definition 3.2.8. Eine Grammatik heißt *rechtslinear*, wenn jede Produktion entweder von der Form $A \to tB$ oder von der Form $A \to t$ ist mit $t \in (T \cup \varepsilon)$. Eine Sprache L

heißt rechtslinear, falls es eine rechtslineare Grammatik G gibt mit $L = \mathcal{L}(G)$. Analog definiert man linkslineare Grammatiken.

Jetzt können wir der Charakterisierung der regulären Sprachen (Satz 2.9.1) eine weitere Äquivalenz hinzufügen:

Satz 3.2.9. *Eine Sprache ist regulär genau dann wenn sie rechtslinear ist.*

Beweis. Ist eine Sprache L regulär, so gibt es einen Automaten $\mathcal{A}$ mit $L = \mathcal{L}(\mathcal{A})$. Aus diesem gewinnen wir folgendermaßen eine rechtslineare Grammatik: Jedem Zustand q von $\mathcal{A}$ ordnen wir ein Nonterminal Q zu und für jeden Übergang $\delta(q, a) = q'$ nehmen wir die Produktion $Q \to aQ'$ auf. Der Startzustand s des Automaten wird zum Startsymbol S der Grammatik und für jeden Endzustand q_f von $\mathcal{A}$ fügen wir die Produktion $Q_f \to \varepsilon$ hinzu. Einem Pfad

$$s \overset{a_1}{\to} q_1 \overset{a_2}{\to} q_2 \to \ldots \overset{a_n}{\to} q_f$$

im Automaten entspricht dann genau die Ableitung

$$S \Rightarrow a_1 Q_1 \Rightarrow a_2 Q_2 \Rightarrow \ldots \Rightarrow a_1 a_2 \ldots a_n Q_f \Rightarrow a_1 a_2 \ldots a_n.$$

Umgekehrt können wir einer rechtslinearen Grammatik einen (i.A. nichtdeterministischen) Automaten zuordnen. Aus den Nonterminalen der Grammatik werden die Zustände des Automaten, zusätzlich fügen wir einen Endzustand Q_f hinzu.

Jede Produktion $A \to tB$ der Grammatik führt zu einem Übergang $B \in \delta(A, t)$ des Automaten. Jede Produktion $A \to t$ mit $t \in T \cup \varepsilon$ führt zu einer entsprechenden Automatentransition $Q_f \in \delta(A, t)$. □

Aufgabe 3.2.10. Definieren Sie *linkslineare Grammatiken* und zeigen Sie, dass jede rechtslineare Sprache auch linkslinear ist und umgekehrt.

3.2.2 Ableitungsbäume

Während eine Ableitungsfolge belegen kann, dass ein Wort in der Sprache ist, so ist am Ende der Ableitung der Beleg, warum es in der Sprache ist, verschwunden. In einem Ableitungsbaum kann man dagegen die Ableitung selbst visualisieren. Es handelt sich um einen Baum, dessen Wurzel der Startknoten der Grammatik ist. Jeder Knoten entspricht der Anwendung einer Regel. Er ist mit dem Nonterminal der linken Seite der Regel beschriftet, seine Söhne mit den Token oder Nonterminalen der rechten Seite. Die Blätter des Baumes stellen das hergeleitete Wort aus $T^\star$ dar.

Wir betrachten dazu als Beispiel die Ableitung

$$\mathit{Expr} \overset{\star}{\Rightarrow} \textbf{\textit{id binOp id binOp num}}$$

mit Hilfe der *Expr*-Grammatik aus Beispiel 3.1.1, die zeigt, dass z.B.

```
betrag + kredit * 0.03
```

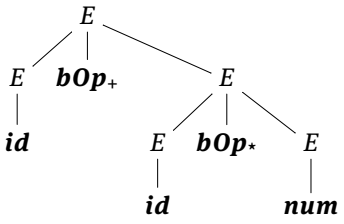
ein korrekter *Expr* ist. Wir beginnen eine Linksableitung mit dem gewünschten *Expr*, das zur Wurzel des Ableitungsbaumes wird. Die Anwendung der Regel

$$Expr \rightarrow Expr\ \mathbf{bOp}\ Expr$$

expandiert die Wurzel zu den drei Söhnen *Expr*, ***bOp*** und *Expr*. (In unserer Baumdarstellung schreiben wir *E* statt *Expr* und indizieren zur einfacheren Orientierung das Token ***bOp*** jeweils mit der konkreten Operation für die das Token in unserem Quelltext steht, also mit $'+'$ oder mit $'*'$.) Im zweiten Schritt haben wir jetzt die Auswahl zwischen Regel 2 ($Expr \rightarrow \mathbf{id}$) oder Regel 3 ($Expr \rightarrow Expr\ \mathbf{bOp}\ Expr$). Beide Auswahlen sind in der Lage die gewünschte Tokenfolge zu produzieren, wie die folgenden Ableitungen demonstrieren. Im zweiten Schritt einer Linksableitung können wir entweder Regel Nr. 2 oder Regel Nr. 3 wählen.

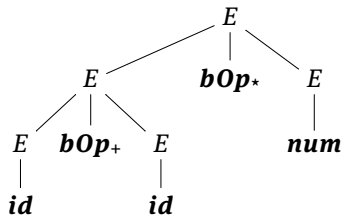
Im ersten Versuch wählen wir Regel 2 ($Expr \rightarrow \mathbf{id}$). Die folgenden Schritte sind eindeutig. Wir erkennen, dass der entstehende Ableitungsbaum die gewünschte Tokenfolge liefert wenn man seine Blätter von links nach rechts durchläuft.

$$\begin{aligned}
Expr &\Rightarrow Expr\ \mathbf{bOp}\ Expr \\
&\overset{2}{\Rightarrow} \mathbf{id}\ \mathbf{bOp}\ Expr \\
&\Rightarrow \mathbf{id}\ \mathbf{bOp}\ Expr\ \mathbf{bOp}\ Expr \\
&\Rightarrow \mathbf{id}\ \mathbf{bOp}\ \mathbf{id}\ \mathbf{bOp}\ Expr \\
&\Rightarrow \mathbf{id}\ \mathbf{bOp}\ \mathbf{id}\ \mathbf{bOp}\ \mathbf{num}
\end{aligned}$$



Im zweiten Versuch wählen wir stattdessen Regel 3 ($Expr \rightarrow Expr\ \mathbf{bOp}\ Expr$). Die aus den Blättern des entstandenen Baumes gewonnene Tokenfolge ist die gleiche wie vorhin, aber der Baum hat eine andere Struktur. Im letzten Fall entspricht er der Linksklammerung, semantisch würde der eingegebene Programmtext also als Produkt einer Summe mit einem Numeral aufgefasst, im vorigen Fall jedoch als Summe eines ***id*** und eines Produktes.

$$\begin{aligned}
Expr &\Rightarrow Expr\ \mathbf{bOp}\ Expr \\
&\overset{3}{\Rightarrow} Expr\ \mathbf{bOp}\ Expr\ \mathbf{bOp}\ Expr \\
&\Rightarrow \mathbf{id}\ \mathbf{bOp}\ Expr\ \mathbf{bOp}\ Expr \\
&\Rightarrow \mathbf{id}\ \mathbf{bOp}\ \mathbf{id}\ \mathbf{bOp}\ Expr \\
&\Rightarrow \mathbf{id}\ \mathbf{bOp}\ \mathbf{id}\ \mathbf{bOp}\ \mathbf{num}.
\end{aligned}$$



Offensichtlich war die erste Ableitung der zweiten vorzuziehen, da sie die gewünschten Präzedenzen der Operatoren widerspiegelt: „Punktrechnung vor Strichrechnung“. Will man garantieren, dass die zweite Ableitung, die den Ausdruck als Produkt interpretiert, ausgeschlossen ist, so kann man natürlich Klammern verwenden, es gibt aber eine bessere Methode.

3.3 Eingebaute Präzedenzregeln

Um Präzedenzen von Operatoren durch die Grammatik zu erzwingen, kann man zusätzliche Variablen einführen - für jede Präzedenzstufe eine. Im Falle der arithmetischen Ausdrücke in Beispiel 3.2.3 kann man drei Präzedenzstufen unterscheiden: ’+’ hat niedrigere Präzedenz als ’*’ und dieses wiederum niedrigere Präzedenz als geklammerte Ausdrücke. Demzufolge führen wir drei Nonterminale ein. *AExpr* für Summen, *Term* für Produkte und *Factor* für Terminale oder geklammerte Ausdrücke. Die Grammatik ist dann einfach erzeugt:

Ein *AExpr* ist eine Summe von Termen:

$$AExpr \ ::= \ Term \mid AExpr \ + \ Term$$

ein *Term* ist ein Produkt von Faktoren :

$$Term \ ::= \ Factor \mid Term \ * \ Factor$$

und ein *Factor* ist ein ***num***, ein ***id*** oder ein geklammerter Ausdruck

$$Factor \ ::= \ num \mid id \mid (\ AExpr\ ).$$

Zur Abwechslung haben wir diesmal ’+’, ’*’ und die Klammern **’(’** und **’)’** direkt als Token übernommen.

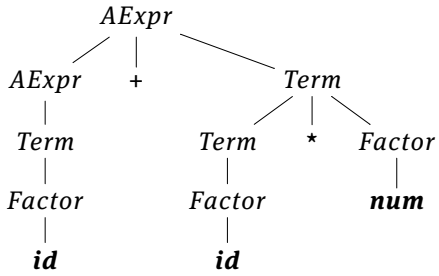
Ein stets wiederkehrendes Muster, das man auch in dieser Grammatik antrifft ist die Modellierung von Listen, ggf. mit einem Trennzeichen versehen: Ein *AExpr* ist eine durch ’+’ getrennte nichtleere Liste von Termen und ein *Term* ist eine durch ’*’ getrennte nichtleere Liste von Faktoren.

Versuchen wir nun mit dieser Grammatik den Programmtext

```
betrag + kredit * 0.03
```

abzuleiten, so werden die Präzedenzen automatisch korrekt im Ableitungsbaum sichtbar: Der gesamte Ausdruck ist eine Summe, nicht ein Produkt.

$$
\begin{aligned}
\mathit{AExpr} &\Rightarrow \mathit{AExpr} + \mathit{Term}\\
&\Rightarrow \mathit{Term} + \mathit{Term}\\
&\Rightarrow \mathit{Factor} + \mathit{Term}\\
&\Rightarrow \mathbf{\mathit{id}} + \mathit{Term}\\
&\Rightarrow \mathbf{\mathit{id}} + \mathit{Term} * \mathit{Factor}\\
&\Rightarrow \mathbf{\mathit{id}} + \mathit{Factor} * \mathit{Factor}\\
&\Rightarrow \mathbf{\mathit{id}} + \mathbf{\mathit{id}} * \mathit{Factor}\\
&\Rightarrow \mathbf{\mathit{id}} + \mathbf{\mathit{id}} * \mathbf{\mathit{num}}
\end{aligned}
$$



Ein nettes kleines Tool das aufgrund einer selbst ausgedachten Grammatik Ableitungsbäume zu vorgegebenen Tokenfolgen generieren und anzeigen kann ist der *Grammar Editor* von C. Burch. Alternativ kann man auch zufällige Tokenfolgen erzeugen lassen. Das folgende Bildschirmfoto zeigt in einem Fenster die Definition unserer Grammatik und eine Tokenfolge und in einen zweiten Fenster den erzeugten Baum. Aufgrund der Beschränkung der Tokensyntax auf Buchstaben und Zeichen haben wir ´+´ durch '*plus*' und '*' durch '***mal***' repräsentiert. '***auf***" und '***zu***' ersetzen jeweils die geöffnete und geschlossene Klammer.

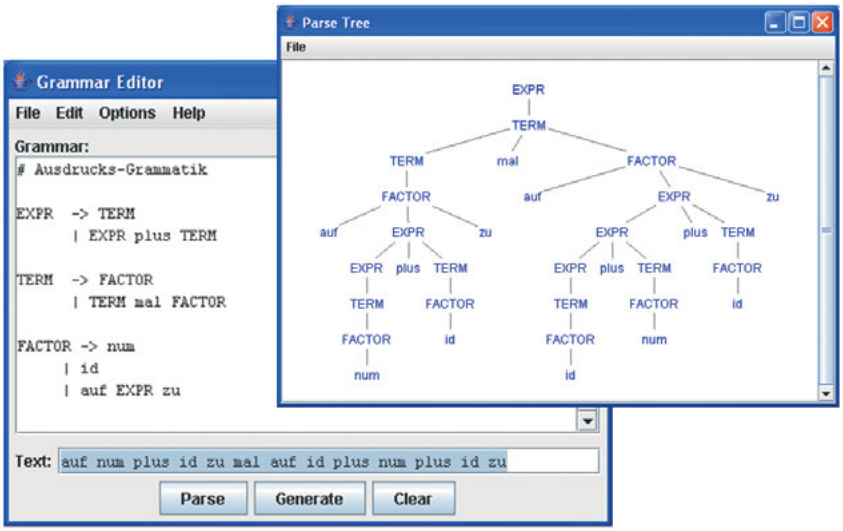


Abb. 3.3.1: Grammar Editor von C.Burch

3.4 Formale Analyse von Grammatiken

3.4.1 Konstruktion

Die Konstruktion von Grammatiken für Sprachen oder Bestandteile von Sprachen, wie auch die Analyse der von einer Grammatik erzeugten Sprache sind elementare Fertigkeiten die ein Informatiker trainieren muss. Mit etwas Übung erkennt man wiederkehrende Strukturen oder Tricks. Manchmal muss man aber mathematisch kombinatorische Methoden zu Rate ziehen, wie wir anhand eines Beispiels demonstrieren wollen.

Wir beginnen mit der Analyse der Sprachen in Beispiel 3.2.6. Im ersten Beispiel wird durch die Grammatik

$$S ::= a\,S\,b \mid \varepsilon$$

die Sprache $L = \{a^n b^n | \in \mathbb{N}\}$ definiert. Obwohl dies *offensichtlich* zu sein scheint, wollen wir gerade an diesem einfachen Beispiel demonstrieren, wie wir eine solche Behauptung auch formal mathematisch beweisen können. Der Beweis übt eine Vorgehensweise ein, die wir später in komplizierteren Situationen anwenden können.

Lemma 3.4.1. *Die Grammatik $S ::= a\,S\,b \mid \varepsilon$ erzeugt die Sprache $L = \{a^n b^n | \in \mathbb{N}\}$.*

Beweis. Nach Definition 3.2.6 ist $L(S) = \{w \in \{a,b\}^* \mid S \overset{*}{\Rightarrow} w\}$. Da als Zwischenschritte der Ableitung $\overset{*}{\Rightarrow}$ nicht nur Tokenfolgen, sondern beliebige Satzformen auftauchen können, charakterisieren wir allgemeiner alle Satzformen αmit $S \overset{*}{\Rightarrow} \alpha$ und spezialisieren diese in einem zweiten Schritt zu allen α aus $\{a,b\}^*$. Dazu definieren wir

$$F := \{a^n S b^n \mid n \in \mathbb{N}\} \cup \{a^n b^n \mid n \in \mathbb{N}\}$$

und behaupten, dass für eine beliebige Satzform αgilt

$$S \overset{*}{\Rightarrow} \alpha \iff \alpha \in F. \tag{3.4.1}$$

Für die Implikation „$\Longrightarrow$ " reicht es zu überprüfen, dass $S \in F$ ist und dass jede Anwendung einer der Grammatikregeln auf eine Satzform τ in F eine Satzform liefert, die wieder in F ist:

– ist $\tau = a^n S b^n$, so liefert die erste Regel $\tau' = a^{n+1} S b^{n+1} \in F$ und die zweite Regel liefert $\tau' = a^n b^n \in F$

– ist $\tau = a^n b^n$, so ist keine Regel anwendbar und die Behauptung daher wahr.

Für die Implikation " $\Longleftarrow$ " kann man zunächst durch eine leichte Induktion nach n zeigen, dass mit Hilfe der ersten Regel jede Satzform $a^n S b^n$ aus S abgeleitet werden kann. Daraus erhält man $a^n b^n$ durch eine abschließende Anwendung der zweiten Regel. Nach Definition ist nun

$$\begin{aligned} L(S) &= F \cap \{a,b\}^* \\ &= (\{a^n S b^n \mid n \in \mathbb{N}\} \cup \{a^n b^n \mid n \in \mathbb{N}\}) \cap \{a,b\}^* \\ &= \{a^n b^n \mid n \in \mathbb{N}\}. \end{aligned}$$

□

Die zweite Grammatik aus Beispiel 3.2.6 soll die Dyck-Sprache, also alle „wohlgeformten“ Klammerungen liefern. Da wir nicht präzise definiert haben, was wir mit „wohlgeformt“ meinen, kann schwerlich bewiesen werden, dass die angegeben Grammatik die richtige ist. Vielmehr sollte man die angegebene Grammatik als (elegante) Definition für die gewünschte Sprache ansehen. In Definition 1.4.1 hatten wir die Sprache als Überbleibsel eines arithmetischen Ausdrucks bezeichnet, wenn man alles außer der Klammern weglässt. Starten wir also mit der *Expr*-Sprache von Beispiel 3.1.1 und lassen alle Terminale außer den Klammern weg, so erhalten wir tatsächlich exakt die Grammatik der Dyck-Sprache.

Die dritte Grammatik für die Sprache $L = \{a^m b^n \mid m \geq n\}$ lässt sich aus den bekannten Versatzstücken leicht konstruieren. Dazu stellen wir uns jedes Wort $a^m b^n$ aus L aus zwei Teilen zusammengesetzt vor:

$$a^m b^n = a^{m-n} a^n b^n.$$

Eine beliebige Folge von a's gefolgt von einem Wort aus $\{a^n b^n \mid n \in \mathbb{N}\}$. Die Grammatik für letztere Sprache hatten wir bereits als

$$U ::= a\, U\, b \mid \varepsilon$$

gekennzeichnet. Beliebig viele vorangestellte a's, also Listen von 0 oder mehreren a's, liefert die Grammatik

$$A ::= \varepsilon \mid a\, A.$$

Die beiden Sprachen müssen wir nur noch kombinieren durch

$$S ::= AU$$

wobei S zum Startsymbol der fertigen Grammatik wird.

3.4.2 Analyse

Jetzt beginnen wir mit einer Grammatik und wollen die Sprache dieser Grammatik analysieren. Ausgangspunkt sei

$$\begin{aligned}
S &::= aB \\
&\mid\ bA \\
&\mid\ \varepsilon \\
A &::= aS \\
&\mid\ bAA \\
B &::= bS \\
&\mid\ aBB
\end{aligned}$$

Erzeugt man einige Worte $\varepsilon, ab, ba, abab, abba, baab, baba, ababba$, so scheint es immer der Fall zu sein, dass gleich viele a's wie b's vorkommen.

Für ein Wort w seien $|w|_a$ bzw. $|w|_b$ die Anzahl der Vorkommen von a bzw. b in w. Die Vermutung lautet somit:

Lemma 3.4.2. $L(S) = \{w \in \{a, b\}^\star \mid |w|_a = |w|_b\}$.

Beweis. Die gewünschte Behauptung erhalten wir als Konsequenz einer stärkeren Behauptung, die zusätzlich feststellt, dass die Worte, die aus A abgeleitet werden können, ein überschüssiges a besitzen und analoges gilt für B. Wir wollen also zeigen:

$$\begin{aligned} S \overset{\star}{\Rightarrow} w &\iff |w|_a = |w|_b \\ A \overset{\star}{\Rightarrow} w &\iff |w|_a = |w|_b + 1 \\ B \overset{\star}{\Rightarrow} w &\iff |w|_a + 1 = |w|_b \end{aligned}$$

Wir können diese Äquivalenzen simultan durch Induktion über die Länge von w herleiten.

Induktionsanfang $w = \varepsilon$: Es gilt $S \to \varepsilon$ und gleichzeitig $|\varepsilon|_a = 0 = |\varepsilon|_b$. Ebenso gilt $A \overset{\star}{\not\Rightarrow} \varepsilon$ und gleichzeitig $|\varepsilon|_a \neq |\varepsilon|_b + 1$ und analoges gilt für B, somit sind alle Äquivalenzen erfüllt.

Für den Induktionsschritt sei $|w| = k+1$ und beginne mit a also $w = au$ mit $|u| = k$. (Den symmetrischen Fall $w = bv$ überlassen wir dem Leser.)

Jetzt gilt

$$\begin{aligned} S \overset{\star}{\Rightarrow} w &\iff S \overset{\star}{\Rightarrow} a\,u \\ &\iff B \Rightarrow^\star u \\ &\overset{IndHyp}{\iff} |u|_a + 1 = |u|_b \\ &\iff |au|_a = |au|_b \\ &\iff |w|_a = |w|_b \end{aligned}$$

sowie

$$\begin{aligned} B \overset{\star}{\Rightarrow} w &\iff B \overset{\star}{\Rightarrow} au \\ &\iff u = u_1u_2 \wedge B \overset{\star}{\Rightarrow} u_1 \wedge B \overset{\star}{\Rightarrow} u_2 \\ &\overset{IndHyp}{\iff} u = u_1u_2 \wedge |u_1|_b = |u_1|_a + 1 \wedge |u_2|_b = |u_2|_a + 1 \\ &\iff w = au_1u_2 \wedge |au_1|_b = |u_1|_a + 1 \wedge |u_2|_b = |u_2|_a + 1 \\ &\iff w = au_1u_2 \wedge |w|_b = 1 + |u_1|_a + |u_2|_a + 1 \\ &\iff w = au_1u_2 \wedge |w|_b = |au_1u_2|_a + 1 \\ &\iff |w|_b = |w|_a + 1 \end{aligned}$$

Für die Rückrichtung der letzten Äquivalenz folgt aus $w = au$ und $|w|_b = |w|_a + 1$, dass $|u|_b = |u|_a + 2$. Wir können also u in zwei Teile $u = u_1 u_2$ zerlegen, so dass $|u_1|_b = |u_1|_a + 1$ gilt und $|u_2|_b = |u_2|_a + 1$. □

Aufgabe 3.4.3. Ein alternativer Beweisweg für die obige Behauptung betrachtet beliebige Satzformen α und zeigt, dass:

$$\begin{aligned} S \overset{\star}{\Rightarrow} \alpha \quad &\Longleftrightarrow \quad |\alpha|_a + |\alpha|_A = |\alpha|_b + |\alpha|_B \\ A \overset{\star}{\Rightarrow} \alpha \quad &\Longleftrightarrow \quad |\alpha|_a + |\alpha|_A = |\alpha|_b + |\alpha|_B + 1 \\ B \overset{\star}{\Rightarrow} \alpha \quad &\Longleftrightarrow \quad |\alpha|_a + |\alpha|_A + 1 = |\alpha|_b + |\alpha|_B. \end{aligned}$$

Versuchen Sie, dies zu beweisen. Es ist ratsam die Implikationen " $\Longrightarrow$ " und " $\Longleftarrow$ " getrennt zu beweisen.

3.4.3 Äquivalenz

Zwei Grammatiken G_1 und G_2 heißen *äquivalent*, wenn sie die gleiche Sprache erzeugen. Beispielsweise sind die folgenden drei Grammatiken äquivalent:

$$\begin{array}{lcl} E & ::= & num - E \\ & | & num \end{array} \qquad \begin{array}{lcl} E & ::= & E - E \\ & | & num \end{array} \qquad \begin{array}{lcl} E & ::= & E - num \\ & | & num \end{array}$$

In jedem Falle werden Differenzen von ***num*** definiert, also Listen von ***num***, die durch **„-“** getrennt werden. Was die Spracherzeugung angeht, so sind alle Grammatiken gleichwertig. Dennoch erkennen wir Unterschiede:

Die erste und die letzte Grammatik sind linear. Die zugehörigen Sprachen sind regulär, sie lassen sich also durch einen regulären Ausdruck angeben, etwa durch $(\boldsymbol{num}\,-)^{\star}\boldsymbol{num}$ oder durch $\boldsymbol{num}\,(-\boldsymbol{num})^{\star}$. Die zweite Grammatik ist nicht linear, beschreibt aber trotzdem die gleiche Sprache wie die beiden anderen. Die linke Grammatik ist endrekursiv, die rechte ist endrekursiv und die mittlere nur „rekursiv“.

Welche der äquivalenten Grammatiken vorzuziehen ist, hängt von vielen Überlegungen ab, vordringlich von der Frage der Semantik einer Grammatik aber auch davon, welche Art von *Parser* (Spracherkenner) aus der Grammatik erzeugt werden soll.

3.4.4 Links-Rekursion

Eine Grammatik-Regel ist *links-rekursiv*, wenn sie von der Form $A \to A\alpha$ ist. Links-Rekursion taucht in vielen Grammatiken auf, so sind beispielsweise in der Ausdrucks-Grammatik von Beispiel 3.2.3:

$$\begin{aligned} AExpr &::= AExpr + Term \\ &\mid Term \\ Term &::= Term * Factor \\ &\mid Factor \\ Factor &::= (AExp) \\ &\mid id \\ &\mid num \end{aligned}$$

die Regeln $AExpr \to AExpr + Term$ und auch $Term \to Term * Factor$ linksrekursiv.

Für gewisse Zwecke stören solche Links-Rekursionen nicht. Will man allerdings einen *recursive descent* Parser bauen, dann führen links-rekursive Regeln zu rekursiven Funktionen, die sich sofort und ohne einen Fortschritt erzielt zu haben, wieder selber aufrufen. Für solche Zecke und, wie wir später sehen werden, auch zur Konstruktion der Greibach-Normalform, müssen wir die Grammatik transformieren, bis keine der Regeln mehr linksrekursiv sind.

In der obigen Grammatik werden *AExpr* als Summen von *Term*en und Terme als Produkte von Faktoren definiert. In beiden Fällen haben wir nichtleere Listen gleichartiger Dinge (Term bzw. Factor), die mit Trennzeichen (* bzw. +) getrennt sind. Diese können wir auch durch die folgende Grammatik beschreiben, die die gleiche Liste durch ihr erstes Element und eine Restliste beschreibt:

$$\begin{aligned} AExpr &::= Term\, RestExpr \\ RestExpr &::= +Term\, RestExpr \mid \varepsilon \end{aligned}$$

und genauso ist ein Term eine durch '*' getrennte Liste von Faktoren: :

$$\begin{aligned} Term &::= Factor\, RestTerm \\ RestTerm &::= *Term\, RestTerm \mid \varepsilon \end{aligned}$$

Im Allgemeinen wird ein Nonterminal A durch linksrekursive Regeln definiert, wenn die Produktionen für A die Form haben:

$$\begin{aligned} A &::= A\alpha_1 \mid .... \mid A\alpha_n \\ &\mid \beta_1 \mid ... \mid \beta_k \end{aligned}$$

wobei die α_i und β_j Satzformen sind und die β_i nicht mit A beginnen. Eine solche Grammatik ist äquivalent zu folgender Grammatik:

$$\begin{aligned} A &::= \beta_1 R \mid ... \mid \beta_k R \\ R &::= \alpha_1 R \mid ... \mid \alpha_n R \mid \varepsilon. \end{aligned}$$

Die Äquivalenz der beiden Grammatiken ergibt sich aus der Tatsache, dass in beiden Grammatiken jede Satzform, die anhand der entsprechenden Regeln aus A abgeleitet werden können jeweils eine Folge bestehend aus einem β_i gefolgt von beliebig vielen α_j's ist.

3.5 Grammatik-Transformationen und Normalformen

Wir haben gesehen, dass wir zu einer gegebenen Sprache durchaus verschiedene Grammatiken angeben können. In diesem Kapitel werden wir sehen, dass wir zu jeder kontextfreien Sprache sogar eine Grammatik besonders einfacher Bauform angeben können. Es handelt sich zunächst einmal um die Chomsky-Normalform, die von dem Sprachtheoretiker Noam Chomsky 1958 gefunden wurde und um die Greibach-Normalform von der Sprachtheoretikerin Sheila Greibach 1965.

Bevor wir dies in Angriff nehmen, wollen wir die Grammatik $G = (V, T, P, S)$ ein wenig aufräumen, indem wir „offensichtlich unnütze" Variablen entfernen. Unnütze Variablen sind solche, die bei der Ableitung eines Wortes $S \overset{\star}{\Rightarrow} w \in L(G)$ sowieso nicht vorkommen können, insbesondere wenn sie *nicht erreichbar* oder *nicht produktiv* sind.

Definition 3.5.1. Eine Variable $A \in V$ nennen wir
erreichbar, falls $S \overset{\star}{\Rightarrow} \alpha A \beta$ mit geeigneten Satzformen α und β möglich ist.
produktiv, falls es ein Wort $w \in T^\star$ gibt mit $A \overset{\star}{\Rightarrow} w$.

Eine Variable V ist also *erreichbar*, falls sie in mindestens einer Satzform vorkommt, die aus S abgeleitet werden kann und sie ist produktiv, falls aus ihr erfolgreich ein Wort $w \in T^\star$ gewonnen werden kann. Nicht-erreichbare Variablen, wie auch nicht-produktive Variablen können auf keinen Fall in einem Zwischenschritt einer erfolgreichen Ableitung $S \overset{\star}{\Rightarrow} w \in L(G)$ vorkommen. Daher können wir beide Typen von Variablen und sämtliche Produktionen in denen solche vorkommen, entfernen ohne die erkannte Sprache zu ändern. Dies setzt aber voraus, dass wir Methoden haben, diese Variablen in der gegebenen Grammatik G aufzufinden. Dazu kann man folgende rekursive Charakterisierungen benutzen, die jeweils sofort in ein rekursives Programm übersetzt werden können:

Lemma 3.5.2. *Rekursive Charakterisierung*
1. *A ist* erreichbar $\iff$ *A = S oder es gibt eine* erreichbare *Variable E und eine Regel* $E \to \alpha A \beta$ *in der A also auf der rechten Seite vorkommt.*
2. *A ist* produktiv $\iff$ *Es gibt eine Regel* $A \to \alpha$ *bei der A nicht in* α *vorkommt und jede (andere) in* α *vorkommende Variable produktiv ist.*

Nicht produktive Variablen und auch nicht erreichbare Variable kommen in der realen Welt kaum vor – kein Sprachdesigner wird eine Grammatik vorstellen, in der nicht erreichbare oder nicht produktive Variablen vorkommen. Dennoch erlaubt un-

sere Definition einer Grammatik als (i.W.) eine Menge von Produktionen solche Sonderfälle und diese können wir ab jetzt als erledigt betrachten.

3.5.1 ε-Freiheit

Praktisch nützlich dagegen sind sogenannte ε-Produktionen, also Produktionen der Form $A \to \varepsilon$. Diese kommen in vielen realen Grammatiken vor, stören uns aber beim Erreichen der gewünschten Normalformen. Daher möchten wir sie beseitigen soweit möglich. Falls die Sprache das leere Wort enthält, also $\varepsilon \in L(G)$ werden wir nicht umhin kommen dies mit einer Produktion $S \to \varepsilon$ zu ermöglichen. Alle anderen ε-Transitionen kann man aber entfernen:

Definition 3.5.3. Sei $G = (V, T, P, S)$ eine Grammatik. Als *ε-Regel* bezeichnet man jede Regel der Form $A \to \varepsilon$. Eine Grammatik G heißt *ε-frei*, falls die einzige erlaubte ε-Regel die Regel $S \to \varepsilon$ ist. In diesem Fall darf S bei keiner Regel auf der rechten Seite vorkommen.

Es ist einfach, festzustellen, ob $\varepsilon \in L(G)$ ist. Man muss nur feststellen, ob S *kollabierend* (engl.: *nullable*) ist. Dabei ist eine Variable $A \in V$ kollabierend, wenn es eine Ableitung $A \overset{\star}{\Rightarrow} \varepsilon$ gibt. Eine rekursive Definition, und damit ein Entscheidungsverfahren ist:

A *ist kollabierend* $\iff$ Es gibt eine Regel $A \to \varepsilon$ oder es gibt eine Regel $A \to B_1 B_2 \ldots B_n$ so dass jede Variable B_i kollabierend ist.

Um eine zur Ausgangsgrammatik G äquivalente ε-freie Grammatik zu erhalten, prüft man, ob das Startsymbol kollabierend ist. In diesem Fall wählt man ein neues Startsymbol S' und fügt die Regeln $S' \to S$ und $S' \to \varepsilon$ hinzu. Anschließend ersetzt man, wie unten beschrieben, alle ε-Regeln $A \to \varepsilon$ mit $A \neq S'$ bis keine solche mehr übrig ist. Dies geschieht auf folgende Weise:

- Falls G keine weitere Produktion $A \to \alpha$ mit $\alpha \neq \varepsilon$ hat, kann man in jeder Regel, $B \to \beta$ bei der A rechts vorkommt, alle A's in β durch ε ersetzen.
- Falls G weitere Produktionen $A \to \alpha$ mit $\alpha \neq \varepsilon$ besitzt, füge für jede Regel $B \to \beta$, bei der A in β vorkommt, alle Varianten dieser Regel hinzu, bei denen ein oder mehrere $A's$ weggelassen wurden. Besitzt G also eine Produktion $B \to \alpha A \beta$, so muss, bevor $A \to \varepsilon$ entfernt wird, die Regel $B \to \alpha\beta$ zur Grammatik hinzugefügt werden. Taucht A rechts sogar mehrfach auf, etwa in der Form $B \to \alpha A \beta A y$, dann müssen folgende Regeln hinzugefügt werden: $B \to \alpha\beta A y \mid \alpha A \beta y \mid \alpha\beta$.

Aus G kann man anschließend die Regel $A \to \varepsilon$ weglassen. Die entstandene Grammatik ist offensichtlich äquivalent zur Ausgangsgrammatik.

Somit haben wir also einen konstruktiven Beweis für die folgende Beobachtung:

Lemma 3.5.4. *Zu jeder Grammatik G lässt sich eine Grammatik G^+ konstruieren, so dass $L(G^+) = L(G) - \{\varepsilon\}$ und G^+ keine ε-Regeln enthält. Falls erforderlich muss man nur noch eine Regel $S \to \varepsilon$ hinzunehmen, um eine zu G äquivalente Grammatik zurückzuerhalten.*

Korollar 3.5.5. *Jede kontextfreie Grammatik kann in eine äquivalente ε-freie Grammatik transformiert werden.*

Im folgenden können wir uns darauf beschränken, Grammatiken G^+ zu betrachten, die $\varepsilon \notin L(G^+)$ erfüllen.

3.6 Chomsky-Normalform

Satz 3.6.1 (N.Chomsky)**.** *Zu jeder kontextfreien Grammatik G^+ gibt es eine äquivalente Grammatik, deren Regeln folgende einfache Bauart haben:*

$$\begin{aligned} A &\to a \\ A &\to BC. \end{aligned}$$

mit $A, B, C \in V$ und $a \in T$. Auf der rechten Seite darf also entweder nur ein Terminalzeichen vorkommen, oder eine Kombination von zwei Nonterminalen.

Beweis. Wir können davon ausgehen, dass unsere Grammatik bereits ε-frei ist, so dass also $L(G^+) = L(G) - \{\varepsilon\}$ und G^+ schon ε-frei ist.

Eine Variablenregel ist von der Form $A \to B$ mit $A, B \in V$. Für eine solche Regel und jede Regel $B \to \beta$ fügen wir die Regel $A \to \beta$ hinzu. Dann streichen wir $A \to B$. (Im Spezialfall $A \to A$ können wir diese Regel sofort streichen.)

Regeln der Form $A \to a$ mit $a \in T$ sind erlaubte Regeln und Bestandteile der fertigen Chomsky-Normalform. Neben ihnen haben wir nur noch Regeln der Form $A \to \alpha$ wobei $|\alpha| \geq 2$ eine Folge von Terminalen und Nichtterminalen ist.

Für jedes in α vorkommende Terminalzeichen a führen wir eine neue Variable X_a und nehmen in die Grammatik die Regel $X_a \to a$ auf.

Jetzt stören uns einzig noch Regeln der Bauart $A \to B_1 B_2 .. B_n$ mit $n > 2$. Jede solche können wir aber unter Zuhilfenahme von $n-2$ neuen Nonterminalen $C_1, \ldots, C_{n-2}$ ersetzen durch $A \to B_1 C_1$, $C_1 \to B_2 C_2$,..., $C_{n-2} \to B_{n-1} B_n$. □

Beispiel 3.6.2. Wir betrachten als Beispiel die Grammatik der Dyck-Sprache aus Beispiel 3.2.6:

$$\begin{aligned} S &::= \varepsilon \\ &\mid SS \\ &\mid (S) \end{aligned}$$

Das Startsymbol S ist kollabierend und kommt auch auf der rechten Seite vor. Bevor wir die Grammatik ϵ-frei machen, führen wir daher ein neues Start-Symbol S' ein, zusammen mit den Regeln

$$S' \quad ::= \quad S \mid \varepsilon.$$

Jetzt müssen wir die ε-Regel $S \to \varepsilon$ entfernen. Aus der neu hinzugekommenen Regel $S' \to S$ gewinnen wir $S' \to S \mid \varepsilon$, also nichts Neues. Aus der Regel $S \to (S)$ werden die Regeln $S \to (S) \mid ()$ und aus der Regel $S \to SS$ gewinnen wir $S \to SS \mid \varepsilon S \mid S\varepsilon \mid \varepsilon$, was zu $S \to SS \mid S \mid \varepsilon$ vereinfacht. Dann entfernen wir die Regel $S \to \varepsilon$ und behalten folgende neue Grammatik zurück:

$$\begin{aligned} S' \quad &::= \quad S \mid \varepsilon \\ S \quad &::= \quad (S) \mid () \mid SS \mid S \end{aligned}$$

Die Variablenregel $S \to S$ können wir sofort streichen. Die Variablenregel $S' \to S$ führt zu

$$\begin{aligned} S' \quad &::= \quad (S) \mid () \mid SS \mid \varepsilon \\ S \quad &::= \quad (S) \mid () \mid SS \end{aligned}$$

Als nächstes ersetzen wir die Terminalzeichen '(' und ')' durch Nonterminale L und R (die Abkürzungen sollen an *L*inks und *R*echts erinnern) :

$$\begin{aligned} S' \quad &::= \quad LSR \mid LR \mid SS \mid \varepsilon \\ S \quad &::= \quad LSR \mid LR \mid SS \\ L \quad &::= \quad (\\ R \quad &::= \quad) \end{aligned}$$

Jetzt müssen wir nur noch die jeweils ersten Regeln für S' und für S auf zwei Nonterminale verkürzen. Dazu führen wir erneut ein Nonterminal C mit der Regel $C \to SR$ ein und erhalten die endgültige Chomsky-Normalform:

$$\begin{aligned} S' \quad &::= \quad LC \mid LR \mid SS \mid \varepsilon \\ S \quad &::= \quad LC \mid LR \mid SS \\ C \quad &::= \quad SR \\ L \quad &::= \quad (\\ R \quad &::= \quad) \end{aligned}$$

3.6.1 Entscheidbarkeit

Chomsky publizierte seine Normalform 1959. Sie ist vor allem für theoretische Zwecke interessant. Wenn man sie auf praktische Grammatiken anwendet spiegeln die transformierten Grammatiken nicht unbedingt die Intention der ursprünglichen Grammatik wieder. Die Zuordnung einer Semantik zu einem entsprechenden Syntaxbaum wird

zu unübersichtlich um von praktischem Nutzen zu sein. Immerhin ergibt sich als Korollar aus der Chomsky Normalform, dass die Frage, ob ein Wort w zu einer Sprache $L(G)$ gehört, für kontextfreie Sprachen prinzipiell entscheidbar ist. Falls $w = \varepsilon$ ist, ist nur zu überprüfen, ob S nullable ist. Falls $w \neq \varepsilon$, kann man die Grammatik zunächst in Chomsky Normalform transformiert. Diese Transformation kann man leicht durch ein Programm $derivable(A, w)$ erledigen lassen, die für ein beliebiges Wort $w \in T^*$ und $A \in V$ überprüft, ob $A \overset{*}{\Rightarrow} w$ anhand der Chomsky Grammatik gilt:

$derivable(A, w)$ gdw.

- $|w| = 1$, also $w = a \in T$ und es gibt eine Regel $A \to a$
- $|w| \geq 2$ und es gibt eine Zerlegung $w = uv$ so dass $u, v \neq \varepsilon$ und es gibt eine Regel $A \to BC$ mit $derivable(B, u)$ und $derivable(C, v)$.

Wir haben es hier also mit einer rekursiven Prozedur zu tun, die offensichtlich terminiert, da in den inneren Aufrufen die zu überprüfenden Wörter u und v immer kürzer werden. Sicherlich ist diese Prozedur nicht die effizienteste, da viele Teilaufgaben immer wieder berechnet werden, ähnlich wie bei der unbedarften Programmierung der Fibonacci-Folge, daher kann durch Tabellierung von Zwischenergebnissen dieser Algorithmus auch in eine effizientere Form gebracht werden.

Dies ist eine Standardmethode im funktionalen Programmieren. Im vorliegenden Fall hat der entstehende Algorithmus sogar einen Namen, der aus den Anfangsbuchstaben der Autoren geformt wurde, der sogenannte *CYK-Algorithmus*.

Allerdings ist schon die Transformation der Grammatik in Chomsky-Normalform sehr aufwendig und nur für theoretische Überlegungen von Interesse, so dass eine Beschleunigung eines darauf aufbauenden Algorithmus nicht so spannend ist. Wichtig sind die theoretischen Ergebnisse, die wir bis jetzt gewinnen können, dass nämlich die folgenden Fragen algorithmisch, also durch ein Programm entschieden werden können:

Satz 3.6.3. *Für eine kontextfreie Grammatik G sind folgende Fragen entscheidbar:*
- *Ist* $L(G) \neq \emptyset$?
- *Ist* $\varepsilon \in L(G)$?
- *Ist* $w \in L(G)$ *? für ein vorgegebenes* $w \in T^*$.

3.6.2 Greibach-Normalform

1965 publizierte Sheila Greibach von der Harvard University eine neue Normalform, die einige Vorteile gegenüber der Chomsky-Normalform hat. Auch hier geht man von einer bereits ε-freien Grammatik aus:

Satz 3.6.4 (S. Greibach). *Zu jeder Grammatik G^+ gibt es eine äquivalente Grammatik, deren Regeln folgende Bauart haben:*

$$\begin{aligned} A &\to a \\ A &\to a\,B_1 \ldots B_n \end{aligned}$$

mit $B_1, \ldots, B_n \in V$ und $a \in T$. Die rechte Seite jeder Regel beginnt also mit einem Terminalzeichen und darauf dürfen nur noch Nonterminale folgen.

Es gibt zahlreiche Vorschläge zur Transformation einer Grammatik in eine äquivalente Grammatik in Greibach Normalform. Wir skizzieren hier eine der Möglichkeiten und demonstrieren diese anhand unserer Dyck-Sprache.

Geht es allein um die Frage der Existenz einer Greibach-Normalform, beginnt man im Allgemeinen mit der Chomsky-Normalform. Die Regeln der Form $A \to a$ mit $a \in T$ sind bereits in der gewünschten Form, mit $n = 0$. Die übrigen Regeln sind jetzt alle von der Form $A \to BC$ mit Nonterminalen A, B, C.

Wir versuchen nun die Nonterminale in eine Reihenfolge zu bringen, indem wir sie mit $A_0, A_1, \ldots, A_n$ neu benennen. Dazu beginnen wir mit dem Startsymbol S welches in A_0 umbenannt wird. Die auf der rechten Seite einer Regel auftretenden Nonterminale, die noch nicht umbenannt wurden, erhalten dann aufsteigend die Namen A_i, A_{i+1}, etc..

Im Folgenden versuchen wir zu erreichen, dass jede Regel $A_i \to A_j\alpha$, deren rechte Seite also mit einem Nonterminal beginnt, *geordnet* ist, in dem Sinne, dass $i < j$ gilt.

Falls noch eine Regel $A_i \to A_j\alpha$ vorhanden ist mit $i > j$, ersetzen wir das erste Nonterminal A_j durch alle rechten Seiten der vorhandenen Regeln für A_j. Diesen Vorgang müssen wir ggf. mehrfach wiederholen, solange noch eine nicht geordnete Regel existiert (oder neu entstanden ist). Es bleibt noch der Fall zu behandeln in dem $i = j$ ist. Solche Regeln sind von der Form $A_i \to A_i\alpha$, also *links-rekursiv* und das Ersetzungsverfahren würde hier nicht fruchten, da nacheinander u.a. die Regeln $A_i \to A_i\alpha$, $A_i \to A_i\alpha\alpha$,$A_i \to A_i\alpha\alpha\alpha$, ... entstehen würden. Wir haben aber bereits in Abschnitt 3.4.4 gesehen, wie man Links-Rekursionen entfernen kann, und dies werden wir für Regeln der Form $A_i \to A_i\alpha$ anwenden. Die hierbei eingeführten ε-Regeln sind leicht wieder zu entfernen.

Haben wir nun die Grammatik so transformiert, dass nur noch geordnete Regeln der Form $A_i \to A_j\alpha$ mit $j > i$ vorhanden sind, so kann für das größte j die Regel $A_j \to \alpha$ nur mit einem Nonterminal beginnen! Durch systematische Substitution erreichen wir dies auch für A_{j-1},A_{j-2}, etc.. Nachdem also alle Regeln von der Bauart $A \to ay$ sind, mit Terminalzeichen a und Satzform y, haben wir fast die Greibach-Normalform erreicht. Es bleibt nur noch, alle Terminalzeichen b in y durch ein neues Nonterminal X_b zu ersetzen und die Regel $X_b \to b$ einzufügen.

Beispiel 3.6.5. Für die Dyck-Sprache haben wir oben bereits die Chomsky Normalform hergeleitet. Verzichten wir auf das leere Wort, so vereinfacht sich dies zur fol-

genden Grammatik G^+:

$$\begin{aligned} S &::= L\,C \mid L\,R \mid S\,S \\ C &::= S\,R \\ L &::= (\\ R &::=) \end{aligned}$$

Statt die Nonterminale umzubenennen, vereinbaren wir folgende Ordnung: $S \prec L \prec C \prec R$. Die ersten beiden Regeln $S \to L\,C$ und $S \to L\,R$ sind bereits geordnet, denn $S \prec L$. Die Regel $S \to S\,S$ ist linksrekursiv. Mit einer zusätzlichen Variablen *Rest* entfernen wir die Rekursion, wie in Abschnitt 3.4.4 besprochen. Der Übersicht halber ersetzen wir jetzt bereits L und R durch die rechten Seiten ihrer Regeln:

$$\begin{aligned} S &::= (\,C\,Rest \mid (\,)\,Rest \\ Rest &::= \varepsilon \mid S\,Rest \\ C &::= S\,) \end{aligned}$$

Entfernung der ε-Regel ergibt:

$$\begin{aligned} S &::= (\,C \mid (\,) \mid (\,C\,Rest \mid (\,)\,Rest \\ Rest &::= S \mid S\,Rest \\ C &::= S\,) \end{aligned}$$

Substituieren wir nun in den letzten beiden Regeln S durch alle rechten Seiten der Produktionen für S, so beginnen sämtliche Regeln mit einem Terminalzeichen:

$$\begin{aligned} S &::= (\,C \mid (\,) \mid (\,C\,Rest \mid (\,)\,Rest \\ Rest &::= (\,C \mid (\,) \mid (\,C\,Rest \mid (\,)\,Rest \mid (\,C\,Rest\,Rest \mid (\,)\,Rest\,Rest \\ C &::= (\,C\,) \mid (\,)) \mid (\,C\,Rest\,) \mid (\,)\,Rest\,) \end{aligned}$$

Das Ersetzen von $')'$ durch ein entsprechendes Nonterminal (z.B. R) zusammen mit der alten Regel $R \to)$ ist jetzt nur noch Formsache. Übersichtlicher ist die Grammatik nicht geworden, aber sie erfüllt die Bedingungen der Greibach Normalform. Wir erinnern uns, dass wir mit der Grammatik G^+ begonnen hatten, wobei $L(G^+) = L(G) - \{\varepsilon\}$ war. Falls also $\varepsilon \in L(G)$ ist, müssen wir noch S' als neues Startsymbol zusammen mit den Regeln

$$S' ::= \varepsilon \mid S$$

hinzufügen.

3.6.3 Grenzen kontextfreier Sprachen

Formale Sprachen in der Informatik, seien es Programmiersprachen, Protokolle, Datenbeschreibungssprachen oder Dateiformate, werden fast ausschließlich mit Hilfe kontextfreier Grammatiken definiert. Allerdings gibt es meist Randbedingungen, die man entweder nur schwer oder gar nicht in eine kontextfreie Beschreibung pressen will oder kann.

So lässt sich das folgende relevante Problem auch mit einer Grammatiktransformation nicht beheben: In vielen Programmiersprachen sind Variablen zu deklarieren bevor sie zum ersten Mal benutzt werden. Ein Programmfragment

```
{ int a,b ; a = 5; b = 7; }
```

wäre korrekt in Java, das Fragment

```
{ int a,b; aa = 5; b = 7; }
```

dagegen nicht – es würde den Fehler ,,`undeclared variable aa`'' erzeugen. Wir können den abstrakten Kern der Situation extrahieren, indem wir die Sprache

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

betrachten, oder die ähnlich aussehende Sprache

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

Beide Sprachen sind nicht kontextfrei – sie lassen sich also nicht mit Hilfe einer kontextfreien Grammatik definieren. Dies wird aus dem Pumping Lemma für kontextfreie Sprachen folgen.

Immerhin lässt sich $L_{a^n b^n c^n}$ als Schnitt kontextfreier Sprachen bilden:

$$L_{a^n b^n c^n} = L_{a^m b^m c^n} \cap L_{a^m b^n c^n}.$$

Da man leicht sieht, dass die Vereinigung kontextfreier Sprachen wieder kontextfrei ist, und da sich der Schnitt zweier Sprachen mittels Komplement und Vereinigung bilden lässt, gilt auch:

Kontextfreie Sprachen sind zwar unter Vereinigung abgeschlossen, nicht aber unter Schnitt oder Komplementbildung.

Aufgabe 3.6.6 (Operationen auf kontextfreien Sprachen)**.** Sind L_1 und L_2 kontextfreie Sprachen. Dann sind auch die folgenden Sprachen kontextfrei:

- die Vereinigung $L_1 \cup L_2$
- die Konkatenation $L_2 \cdot L_2$
- der Kleene-Stern $L_1^\star$.

3.6.4 Das Pumping Lemma für kontextfreie Sprachen

Satz 3.6.7. *Für jede kontextfreie Sprache L gibt es eine Zahl k, so dass jedes Wort $w \in L$ mit $|w| > k$ sich zerlegen lässt als $w = xuyvz$, so dass*

- $0 < |uv| < |uyv| \leq k$ *und*
- $\forall n \in \mathbb{N}.\ xu^n yv^n z \in L$.

Auch dieses können wir mit der Metapher des Aufpumpens so formulieren: Es gibt ein k so dass jedes längere Wort sich an zwei Stellen simultan aufpumpen lässt. Man beachte, dass u oder v leer sein dürfen, nicht aber beide. Beide müssen sich in einem Umkreis von k Zeichen befinden, aber nicht notwendig im vorderen Bereich wie beim Pumping Lemma für reguläre Sprachen.

Bevor wir dieses Pumping Lemma beweisen, wollen wir es auf die Sprache $L = L_{a^n b^n c^n}$ anwenden. Angenommen, diese Sprache sei kontextfrei, dann gäbe es ein k wie im Pumping Lemma gefordert. Für das Wort $w = a^k b^k c^k$ gilt dann $w \in L$. Für jedes Teilwort uyv von w der Länge $\leq k$ gilt aber, dass nicht sowohl a als auch c in uyv vorkommen kann. Jedes Auf- oder Abpumpen von $w = xuyvz$ zerstört damit die Balance der Anzahl der a's und der Anzahl der c's.

Beweis des Pumping Lemmas: Als kontextfreie Sprache hat L eine Grammatik $G = (V, T, P, S)$ in Chomsky-Normalform. Alle Regeln sind also von der Form $A \to a$ oder $A \to BC$. Die zugehörigen Syntaxbäume haben nur Knoten mit zwei Nachfolgern bzw. Blätter mit einem Nachfolger. Da ein Binärbaum der Tiefe t höchstens 2^t viele Blätter besitzt, muss der Syntaxbaum für ein Wort $w \in L$ der Länge $|w| \geq 2^t$ mindestens Tiefe t haben. Für das gesuchte k wählen wir $k := 2^t$ mit irgendeinem $t > |V|$.

Sei jetzt w ein beliebiges Wort aus L mit $|w| > k$, dann hat der Ableitungsbaum von w eine Tiefe t, die größer als $|V|$ ist. Es folgt, dass auf einem Pfad des Ableitungsbaumes ein Nonterminal A mindestens zweimal vorkommt. Dies bedeutet, dass im Syntaxbaum von w ein Pfad existiert, der der folgenden Ableitung entspricht – mit geeigneten α_i, β_i:

$$S \overset{*}{\Rightarrow} \alpha_1 A \alpha_2 \Rightarrow \ldots \Rightarrow \alpha_1 \beta_1 A \beta_2 \alpha_2 \overset{*}{\Rightarrow} w.$$

Seien jetzt x, u, y, v, z die Teilworte von w, die sich dabei durch Ableitung aus $\alpha_1, \beta_1, A, \beta_2, \alpha_2$ ergeben, d.h. $\alpha_1 \overset{*}{\Rightarrow} x$, $\beta_1 \overset{*}{\Rightarrow} u$, etc. und $w = xuyvz$. Aus der obigen Ableitung liest man zusätzlich ab, dass

$$A \overset{*}{\Rightarrow} \beta_1 A \beta_2.$$

Damit können wir die obige Ableitung fortsetzen mit den Zwischenschritten

$$S \overset{*}{\Rightarrow} \alpha_1 A \alpha_2 \Rightarrow \ldots \Rightarrow \alpha_1 \beta_1 A \beta_2 \alpha_2 \overset{*}{\Rightarrow} \alpha_1 \beta_1 \beta_1 A \beta_2 \beta_2 \alpha_2 \overset{*}{\Rightarrow} xuuyvvz$$

und so fort, wir können also jedes Wort $xu^n yv^n z$ ableiten, wie versprochen, siehe auch Figur 3.6.1.

□

Mit Hilfe des Pumping Lemmas können wir zeigen, dass gewisse Sprachen nicht kontextfrei sind.

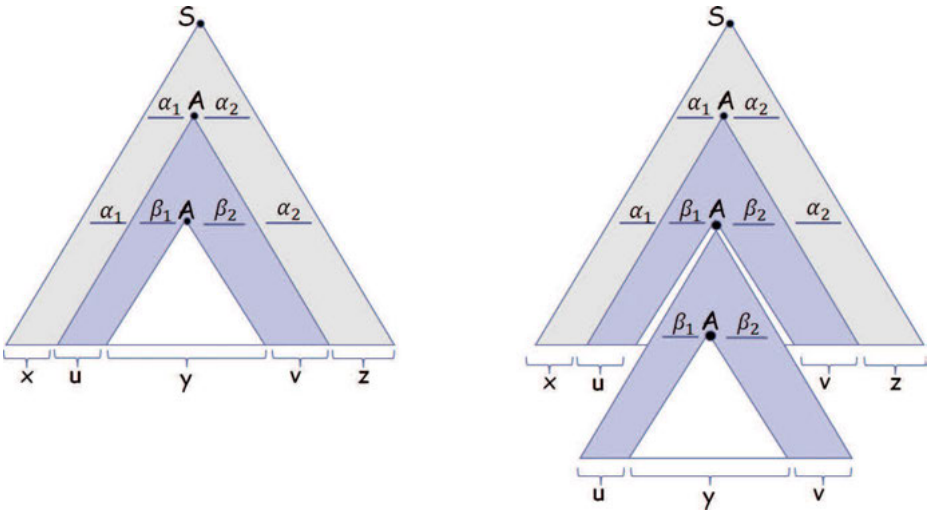


Abb. 3.6.1: Pumping Lemma für kontextfreie Sprachen

Beispiel 3.6.8. $L_{a^n b^n c^n} = \{a^n b^n c^n \mid n \in \mathbb{N}\}$ ist nicht kontextfrei.

Intuitiv könnte eine Stackmaschine den Stack zwar benutzen, um zu garantieren dass auf eine bestimmte Anzahl von a's genauso viele b's folgen. Um dann aber die gleiche Anzahl von c's zu akzeptieren, hätte man sich auch die Anzahl der b's merken müssen und dafür wäre ein zweiter Stack nötig gewesen. Dieses intuitive Argument kann allerdings nur als Anhaltspunkt dienen. Der Beweis, dass $L_{a^n b^n c^n}$ nicht kontextfrei ist benutzt das Pumping Lemma zu einem Widerspruchsbeweis.

Wir nehmen also an, $L_{a^n b^n c^n}$ sei kontextfrei, dann gibt es ein k so dass man jedes längere Wort an zwei Stellen gemeinsam aufpumpen kann. Wir wählen nun $w = a^k b^k c^k$. Dann ist $w \in L$ und $|w| > k$. Jeder Versuch es als $xuyvw$ zu zerlegen, so dass $|uyv| \leq k$ ist und jedes $xu^m yv^m z$ zur Sprache gehört, scheitert. Aus $a^k b^k c^k = xuyvz$ mit $|uyv| \leq k$ folgt, dass entweder kein c im Bereich uyv vorkommen kann, oder kein a. Pumpt man nun u und v auf, so hat das Ergebnis entweder zu wenige c's oder zu wenige a's, so dass das Ergebnis nicht mehr in $L_{a^n b^n c^n}$ sein kann.

Lemma 3.6.9. $L_{a^n b^m c^n} = \{a^n b^m a^n \mid m, n \in \mathbb{N}\}$ *ist kontextfrei.*

Beweis. Eine Grammatik für $L_{a^n b^m a^n}$ ist z.B.:

$$\begin{aligned} S &::= a\,S\,a \mid B \\ B &::= b\,B \mid \varepsilon \end{aligned}$$

□

Modifiziert man die Sprache nur leicht, so ist die resultierende Sprache nicht kontextfrei:

Lemma 3.6.10. $L_1 := \{a^n b^m a^n \mid m > n\}$ *ist nicht kontextfrei.*

Beweis. Zu gegebenem k wählen wir das Wort $w = a^k b^{k+1} a^k$ aus der Sprache. Egal wie wir ein Teilwort uyv von w der Länge $\leq k$ markieren, können wir durch auf- oder abpumpen der zu u und v gehörigen Bestandteile immer ein Wort w' erzeugen, das nicht zu L gehört:

– wenn einer der mit u oder v markierten Teile a's enthält, so stammen diese entweder alle aus dem vorderen Bereich von w oder alle aus dem hinteren Bereich. Durch Aufpumpen gelangen wir zu einem Wort w', das entweder vorne oder hinten zu viele a's enthält.

– wenn weder u noch v a's enthält, so stammt der ganze Teil uyv aus dem mittleren Bereich, enthält aber nur b's. Wegen $|uv| > 0$ gehen beim Abpumpen nur b's verloren, es entsteht also ebenfalls ein Wort, was nicht zu L gehört.

□

$L_2 = \{ww^R \mid w \in \{a, b\}^*\}$ ist kontextfrei, aber $L_3 = \{ww \mid w \in \{a, b\}^*\}$ nicht. Letztere Behauptung ist etwas aufwendig zu zeigen, daher vereinfachen wir uns die Aufgabe ein bisschen, indem wir eine Marke in der Mitte setzen und eine ähnliche Sprache betrachten:

Lemma 3.6.11. $L_4 := \{wcw \in \{a, b, c\}^* \mid w \in \{a, b\}^*\}$ *ist nicht kontextfrei.*

Beweis. Angenommen, L_4 wäre kontextfrei, dann gäbe es ein k so dass jedes Wort der Länge > k sich zerlegen lässt als $w = xuyvz$ mit den Eigenschaften, die im Pumping Lemma spezifiziert sind. Insbesondere gälte das für das Wort $w = a^k b^k c a^k b^k$, dass wir einen höchstens k langen Bereich uyv finden müssten, den wir dann an den mit u bzw. v bezeichneten Abschnitten beliebig auf- und abpumpen können, ohne die Sprache L_4 zu verlassen. Klarerweise darf der Mitte-Marker c nicht zu u oder zu v gehören. Ebenso dürfen nicht sowohl u als auch v im linken Teil bzw. beide im rechten Teil des Wortes sein, denn Aufpumpen würde das Wort vorne bzw. hinten zu lang werden lassen. Es bleibt nur noch die Möglichkeit, dass u in der linken Hälfte und v in der rechten Hälfte zu liegen kommt. Wegen $|uyv| < k$ folgt aber, das u nur aus b's besteht und v nur aus a's. Auf- oder Abpumpen bewirkt, dass das Ergebnis nicht mehr von der Form wcw sein kann. □

3.7 Stackmaschinen (Kellerautomaten)

Die schöne Korrespondenz von regulären Sprachen und Automaten möchten wir auch auf kontextfreie Sprachen ausweiten. Da jede reguläre Sprache kontextfrei ist – sogar mit Hilfe einer links-linearen kontextfreien Grammatik – muss unser Maschinenmodell mehr Fähigkeiten besitzen, als ein Automat. Die Sprache $L = \{a^n b a^n \mid n \in \mathbb{N}\}$ ist nicht regulär, aber offensichtlich kontextfrei. Um sie aber erkennen zu können, muss ein Automat sich bis zur Wortmitte gemerkt haben, wie viele a's gesehen wurden, damit nach der Mitte gleich viele a's – nicht mehr und nicht weniger – akzeptiert werden. Diese Merkfähigkeit fehlt unserem Standard-Automatenmodell.

Automaten, die kontextfreie Sprachen erkennen sollen, müssen daher eine gewisse Speicherfähigkeit haben. Diese verleihen wir ihnen mit Hilfe eines Stacks (auch als *Keller* bezeichnet). Eine Transition des Automaten resultiert nicht nur in einem Zustandsübergang, sondern auch in einer Stackoperation. Auf dem Stack können Zeichen eines eigenen Alphabets abgelegt werden. In jedem Schritt werden, abhängig von dem gelesenen Zeichen a, dem aktuellen Zustand q und dem obersten Stacksymbol s ein neuer Zustand s' gewählt und das oberste Stacksymbol durch ein neues Stackwort ersetzt. Die Erkennung endet, wenn der Stack leer ist. In der Literatur wir oft noch eine Menge F von Endzuständen gefordert, wobei zu Ende der Erkennung die Maschine auch zusätzlich in einem Endzustand sein soll. Allerdings berührt dies nicht die Mächtigkeit der Stackmaschine, so dass wir dies hier weglassen können.

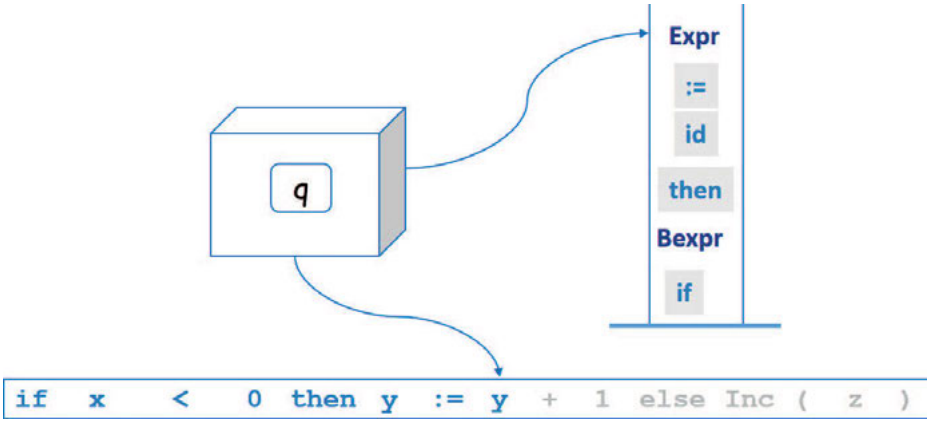


Abb. 3.7.1: Stackmaschine

Definition 3.7.1. Eine Stackmaschine $M = (Q, \Sigma, \Gamma, \delta, q_0, s_0, F)$ ist ein $7-$ *Tupel* bestehend aus

Q	einer endlichen Menge von Zuständen
Σ	einem endlichen Alphabet
Γ	einem endlichen Alphabet (dem Stackalphabet)
δ	einer nichtdeterministischen Transitionsrelation
	$\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \to \mathbb{P}_\omega(Q \times \Gamma^\star)$
$q_0 \in Q$	dem Anfangszustand
$Z \in \Gamma$	dem Stack-Startsymbol
$Q_f \subseteq Q$	einer Menge von Endzuständen.

Dabei stehe $\mathbb{P}_\omega(X)$ für die Menge aller endlichen Teilmengen von X.

Wie die bereits bekannten endlichen Automaten liest der Automat ein Wort $w \in \Sigma^\star$ im Input. Dabei kann er aber nicht nur seinen Zustand verändern, er liest auch immer das oberste Zeichen g des Stacks. Dieses wird (mit *pop*) entfernt und durch ein Wort $y = g_1 \ldots g_n \in \Gamma^\star$ ersetzt, indem diese Zeichen des Stack-Alphabets oben auf dem Stack abgelegt werden. Dann geht der Automat in einen neuen Zustand $q^{'}$. Die Wahl des neuen Zustands $q^{'}$ und des zu stapelnden Wortes $y \in \Gamma^\star$ wird durch δ eingeschränkt. Dabei kann der nächste Buchstabe a des Eingabestrings berücksichtigt werden oder nicht.

Sei der Automat im Zustand q und sei g das oberste Zeichen des Stacks. Entweder wird kein Zeichen des Inputs eingelesen, dann wird $(q^{'}, y)$ aus der endlichen Menge $\delta(q, g, \varepsilon)$ gewählt, oder das nächste Eingabezeichen a wird eingelesen und $(q^{'}, y) \in \delta(q, g, a)$. Der Automat muss auf jeden Fall terminieren, wenn der Stack leer ist. Das Wort $w \in \Sigma^{*}$ gilt als akzeptiert, wenn es komplett eingelesen wurde und danach der Stack leer ist und der Automat sich in einem Endzustand befindet.

Mathematisch können wir diese Handlungsanweisung folgendermaßen codieren: Eine *Konfiguration* $\kappa = (q, w, s)$ bestehe aus dem aktuellen Zustand q, dem noch nicht eingelesenen Eingabewort $w \in \Sigma^\star$ und dem Stackinhalt $y \in \Gamma^\star$. Wir schreiben $\kappa \vdash \sigma$, wenn der Automat von der Konfiguration κ in eine Konfiguration σ übergehen kann. Dabei wird entweder ein Inputzeichen verbraucht, also

$$(q, a.u, g.\alpha) \vdash (p, u, y\alpha) \text{ falls } (p, y) \in \delta(q, a, g)$$

oder es wird kein Inputzeichen gelesen, dann hat man den Übergang

$$(q, w, g.\alpha) \vdash (p, w, y\alpha) \text{ falls } (p, y) \in \delta(q, \varepsilon, g).$$

Als *Sprache* eines Automaten bezeichnet man die Menge aller Worte, die eine *Anfangskonfiguration* in eine *Endkonfiguration* überführen können, wobei eine Endkonfiguration eine solche ist, bei der das Eingabewort vollständig verbraucht ist, der Stack

leer ist und der Zustand des Automaten ein Endzustand ist. Notieren wir Folgen von Zustandsübergängen $\overset{*}{\vdash}$, so können wir definieren:

Definition 3.7.2. Die von einer Stackmaschine $M = (Q, \Sigma, \Gamma, \delta, q_0, Z, Q_f)$ akzeptierte Sprache ist

$$L(M) := \{w \in \Sigma^* \mid \exists q_f \in Q_f . (q_0, w, Z) \overset{*}{\vdash} (q_f, \varepsilon, \varepsilon)\}.$$

Beispiel 3.7.3. Eine Stackmaschine M für die Sprache $L = \{a^n b^n \mid n \in \mathbb{Z}\}$ hat $\Sigma = \{a, b\}$ und kommt mit drei Zuständen q_0, q_1, q_2 und einem Zeichen Z im Stackalphabet $\Gamma = \{Z\}$ aus. Die zugehörige Funktion $\delta : \{q_0, q_1, q_2\} \times \{a, b, \varepsilon\} \times \{Z\} \to \mathbb{P}(\{q_0, q_1, q_2\} \times \{Z\})$ wird durch folgende Tabelle beschrieben, wobei für alle fehlenden Argumente δ den Wert $\emptyset$ haben soll. q_0 dient als Startzustand und q_2 als einziger Endzustand:

x	$\delta(x)$
(q_0, a, Z)	$\{(q_0, ZZ)\}$
(q_0, ε, Z)	$\{(q_1, Z)\}$
(q_1, b, Z)	$\{(q_1, \varepsilon)\}$
(q_1, ε, Z)	$\{(q_2, \varepsilon)\}$
sonst	$\{\}$

Obwohl die Stackmaschine *nicht deterministisch* ist – wenn z.B. in Zustand q_0 ein a gesehen wird, hat die Maschine die Wahl, in dem Zustand q_0 zu bleiben, das a einzulesen und ein weiteres Z auf dem Stack abzulegen, oder in Zustand q_1 zu wechseln und weder etwas einzulesen noch den Stack zu verändern. Dennoch gibt es für jedes Wort aus $L(M)$ nur einen möglichen Lauf, der das Wort akzeptiert. Wir demonstrieren dies anhand des Wortes $a^2b^2 = aabb$:

$$\begin{aligned}
(q_0, aabb, Z) &\vdash (q_0, abb, ZZ) \\
&\vdash (q_0, bb, ZZZ) \\
&\vdash (q_1, bb, ZZZ) \\
&\vdash (q_1, b, ZZ) \\
&\vdash (q_1, \varepsilon, Z) \\
&\vdash (q_2, \varepsilon, \varepsilon).
\end{aligned}$$

Für eine graphische Notation von Stackmaschinen können wir die Übergangsgraphen nichtdeterministischer ε-Automaten aufbohren, indem wir jeder Transition zusätzlich die Stackveränderung in der Form g/y wobei g das oberste Stacksymbol ist

und $y \in \Gamma^*$ das Wort, das g ersetzt. Der folgende Übergangsgraph gehört zu der gerade besprochenen Stackmaschine.

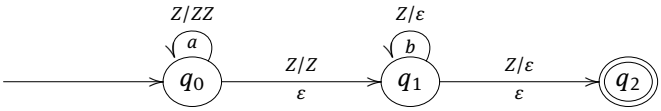


Abb. 3.7.2: Stackmaschine für $\{a^n b^n \mid n \in \mathbb{N}\}$

Im Startzustand q_0 können a's eingelesen werden. Dabei wird jedesmal das oberste Z auf dem Stack durch $Z\,Z$ ersetzt. Faktisch wird also ein zusätzliches Z auf dem Stack abgelegt und auf diese Weise mitgezählt, wie viele $a^{'}s$ gesehen wurden. Eine ε-Transition wechselt in Zustand q_1 ohne den Stack zu verändern – $\delta(q_0, \varepsilon, Z) = \{(q_1, Z)\}$ woraus $(q_0, w, Z) \vdash (q_1, w, Z)$ für jedes $w \in T^*$ folgt. Anschließend wird mit jedem eingelesenen b ein Z vom Stack entfernt. Eine ε-Transition kann ein Z vom Stack entfernen und in den Endzustand q_2 übergehen. Falls dabei der Stack leer geworden ist, müssen es gleich viele $b^{'}s$ wie a's gewesen sein. Die erkannte Sprache ist also in der Tat $\{a^n b^n \mid n \in \mathbb{N}\}$.

3.8 Stackmaschinen für kontextfreie Sprachen

Offensichtlich verallgemeinert der Begriff des Stackautomaten den des Nichtdeterministischen ε-Automaten. Dazu muss man dem ε-Automaten nur pro forma einen Stack mitgeben, und jeder Transition eine Stacktransformation S/S wie oben gesehen. Damit am Schluss der Stack auch leer werden kann, fügt man für jeden Endzustand q_f noch die Transition $(q_f, S) \in \delta(q_f, \varepsilon, \varepsilon)$ hinzu.

In der Literatur findet man auch andere Akzeptanzbegriffe. Dabei wird ein Wort bereits akzeptiert, wenn der Stack leer geworden ist. So kann man ganz auf Endzustände verzichten, oder man erklärt einfach jeden Zustand als Endzustand. Zu einem gegebenen Automaten M kann man so auch eine Sprache L_e definieren, die alle Worte enthält, die den Stack leeren können:

$$L_e(M) := \{w \in T^* \mid \exists q \in Q.\, (q_0, w, Z) \overset{*}{\vdash} (q, \varepsilon, \varepsilon)\}.$$

Wie wir gleich sehen werden, kann man sogar vollends auf Zustandsübergänge und damit auf Zustände verzichten, indem man nur einen einzigen Zustand benutzt und diesen nie verlässt. Wir können nämlich zeigen:

Satz 3.8.1. *Für jede kontextfreie Sprache L gibt es einen Stackautomaten M (mit einem einzigen Zustand) der L erkennt, also $L = L(M)$.*

Beweis. Sei dazu $G = (V, T, P, S)$ die kontextfreie Grammatik mit $L = L(G)$. Für das Stackalphabet wählen wir $\Gamma := V \cup T$. Den einzigen Zustand der Maschine können wir ignorieren, da er sich nie ändert und natürlich auch Endzustand sein muss. Als Startsymbol für den Stack wählen wir das Startsymbol S der Grammatik. Wir definieren $\delta : (T \cup \{\varepsilon\}) \times \Gamma \to \mathbb{P}_\omega(\Gamma^\star)$ durch

– $\delta(a, a) = \{\varepsilon\}$ für jedes $a \in T$
– $\delta(\varepsilon, A) = \{y \mid (A \to y) \in P\}$.

Die Stackmaschine simuliert dann eine Linksableitung. Auf dem Stack findet sich zu jedem Zeitpunkt eine Satzform, die im Input erwartet wird.

– Liegt auf dem Stack ein Terminal, so bedeutet dies, dass als nächstes dieses Terminal im Input erwartet wird. Wird die Erwartung erfüllt, verschwindet dieses Terminalzeichen vom Stack und aus dem Input.
– Liegt auf dem Stack ein Nonterminal A, so wird A durch eine rechte Seite y einer Produktion $(A \to y) \in P$ ersetzt.

Die Maschine endet, wenn der Stack leer ist, also ein S im Input erfolgreich erkannt worden ist. Die Maschine simuliert also eine Linksableitung. Konkateniert man nämlich in jedem Schritt den bereits erkannten vorderen Teil des Eingabewortes $u_k \preceq w$ mit der Satzform σ_k, die noch auf dem Stack liegt, so vollführt die Maschine eine Linksableitung: $S \overset{\star}{\Rightarrow} u_k\sigma_k \Rightarrow u_{k+1}\sigma_{k+1} \overset{\star}{\Rightarrow} w$, im Einzelnen:

– beginnt σ_k mit einem Terminal a, dann ist $\sigma_k = a \cdot \sigma_{k+1}$. Die Maschine kann jetzt $a$einlesen und vom Stack entfernen. Insgesamt wurde jetzt also $u_{k+1} = u_k a$ gelesen und σ_{k+1} befindet sich auf dem Stack, so dass $u_k\sigma_k = u_k(a \cdot \sigma_{k+1}) = (u_k a)\sigma_{k+1} = u_{k+1}\sigma_{k+1}$ gilt.

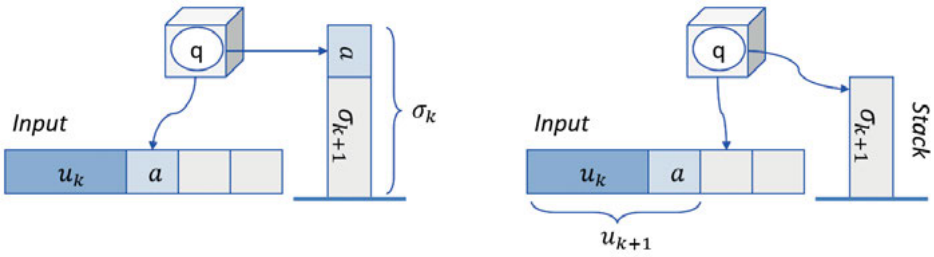


Abb. 3.8.1: Erkennung mit Terminal auf dem Stack

– beginnt σ_k mit einem Nichtterminal A, also $\sigma_k = A \cdot y$ und ist $A \to \alpha$ eine Produktion der Grammatik, so wird nichts eingelesen, also $u_{k+1} = u_k$ und σ_{k+1} wird zu αy. Dieser Schritt entspricht der Linksableitung $u_k\sigma_k = u_k Ay \Rightarrow u_k\alpha y = u_{k+1}\sigma_{k+1}$.

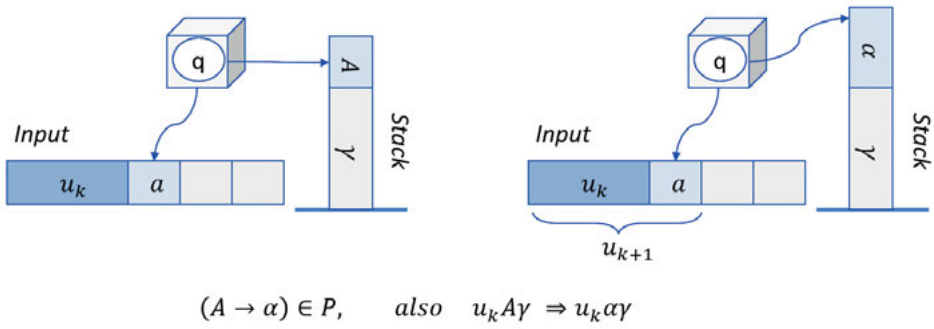


Abb. 3.8.2: Erkennung mit Nonterminal auf dem Stack

Die Maschine beginnt mit $S = \varepsilon S = u_0\alpha_0$ und endet, falls das Wort w in der Grammatik ist mit $u_n = w$ und $\sigma_n = \varepsilon$, also $u_n\sigma_n = w\varepsilon = w$. □

Satz 3.8.1 lässt sich auch umkehren: *Zu jeder Stackmaschine M kann man eine kontextfreie Grammatik G angeben mit* $L(M) = L(G)$. Da diese Richtung aber keinen offensichtlichen praktischen Nutzen zu haben scheint, wollen wir zum Beweis auf die Literatur verweisen. Immerhin können wir feststellen, dass es auch für die kontextfreien Sprachen ein äquivalentes Maschinenkonzept gibt.

3.8.1 Inhärent nichtdeterministische Sprachen

Wünschenswert wäre ein deterministischer Parser, d.h. ein deterministischer Stackautomat. Darunter versteht man einen Stackautomaten – evtl. mit nichtleerer Zustandsmenge Q – für den alle Mengen $\delta(q, a, A) \cup \delta(q, \varepsilon, A)$ höchstens einelementig sind. In jedem Zustand mit beliebigem Eingabezeichen und Stacksymbol gibt es also höchstens eine mögliche Aktion des Parsers. Die Implementierung könnte dann ohne Backtracking auskommen.

Leider gibt es kontextfreie Sprachen, für die ein deterministischer Parser nicht möglich ist. Ein einfaches Beispiel einer solchen nichtdeterministischen Sprache besteht aus allen Binärfolgen der Form uu^R, wobei u^R das zu u „reverse“ (gespiegelte) Wort bedeutet. Man kann sie durch folgende Grammatik definieren:

$$A \quad ::= \quad \varepsilon \mid 0\,A\,0 \mid 1\,A\,1.$$

Das Problem, das diese Sprache einem Parser stellt, ist, zu entscheiden, wann die Mitte des Wortes erreicht ist. Genau zu diesem Zeitpunkt muss die Produktion $A \to \varepsilon$ verwendet werden. Wo die Mitte des Wortes liegt, kann man aber erst sagen, wenn dieses komplett eingelesen ist, da eine beliebige Zeichenfolge u stets zu einem Wort uu^R aus $L(A)$ fortgesetzt werden kann. Wir folgern:

Korollar 3.8.2. *Es gibt Sprachen, für die kein deterministischer Parser existiert.*

Selbst die Vereinigung zweier deterministischer Sprachen kann nichtdeterministisch sein, wie die folgende Aufgabe demonstriert:

Aufgabe 3.8.3. (Sprachen und Stackmaschinen)

1. Seien $L_1 = \{a^m b^n c^n \mid m, n \in \mathbb{N}\}$ und $L_2 = \{a^m b^m c^n \mid m, n \in \mathbb{N}\}$. Zeigen Sie, dass L_1 und L_2 deterministisch sind, $L_1 \cup L_2$ aber nichtdeterministisch.
2. Konstruieren Sie aus einem Automaten A eine Stackmaschine, die die (reguläre) Sprache $L(A)$ erkennt.
3. Seien Stackmaschinen für die Sprachen L_1 und L_2 gegeben. Konstruieren Sie daraus je eine Stackmaschine für
   (a) $L_1 \cup L_2$
   (b) $L_1 \cdot L_2$
   (c) $L_1^\star$.

Dass die regulären Sprachen in den kontextfreien Sprachen enthalten sind, wissen wir bereits: auf der Grammatik-Ebene entsprechen die ersteren den links-linearen Sprachen, auf der Maschinenebene entsprechen sie den Stackautomaten ohne (bzw. mit trivialem) Stack. Wir haben auch schon gesehen, dass die kontextfreien Sprachen eine echte Obermenge der regulären Sprachen bilden. Immerhin sind z.B. die Sprachen L_{anbn} oder die Dyck-Sprachen kontextfrei. Im folgenden Kapitel werden wir auch die Grenzen der kontextfreien Sprachen aufzeigen.

3.9 Kontextabhängige Grammatiken

Für die Beschreibung aller Arten von formalen Sprachen der Informatik hat sich die Kombination

–reguläre Ausdrücke für die lexikalische Definition der Tokens

–kontextfreie Grammatik für die Syntax der Sprache

als Standard etabliert. Es gibt in der Praxis kaum einen Grund allgemeinere Sprachen als kontextfreie Sprachen zu benutzen. Ein lauffähiges Programm erfordert neben einer korrekten Syntax ohnehin noch die Einhaltung weiterer semantischer Bedingungen – dazu gehört die Prüfung, ob alle benutzten Variablen deklariert worden sind, und Operationen nur auf Variablen geeigneten Typs angewendet werden, etc. Theoretisch könnte man einige dieser Prüfungen noch auf der Ebene einer komplexeren Grammatik abhandeln, aber der Preis wäre, dass die Grammatik für den Anwender zu schwer verständlich sein könnte. Ein Beispiel für einen solchen Pragmatismus haben wir schon bei dem *dangling-else*-Problem gesehen. Dies könnte man theoretisch auf der grammatischen Ebene lösen, aber der Preis wäre, dass die Grammatik umständlich und unnatürlich würde.

Von informatischer Seite könnte man daher auf allgemeinere Grammatiken verzichten, aus theoretischen Gesichtspunkten ist jedoch noch eine weitere Klasse von Grammatiken interessant, die sogenannten *kontextabhängigen Grammatiken*. Wir werden diese daher hier kurz streifen, aber auf Beweise verzichten. Abgeleitet von der englischen Bezeichnung *context sensitive* verwendet man statt *kontextabhängig* auch den äquivalenten eingedeutschten Begriff *kontextsensitiv*.

Definition 3.9.1. Eine *kontext-abhängige* (*kontextsensitive*) Regel hat die Form

$$\alpha A \beta \to \alpha y \beta$$

mit $y \neq \varepsilon$. Die Regel $S \to \varepsilon$ ist erlaubt, falls S in keiner Regel auf der rechten Seite auftaucht.

Auf der linken Seite haben wir eine Variable A eingeschlossen zwischen Satzformen α und β. Nur wenn die Variable A auf diese Weise zwischen den Satzformen α und β eingerahmt angetroffen wird, kann man sie durch y ersetzen. Das Paar (α, β) bildet den *Kontext*, in dem die Ersetzung von A durch y erlaubt ist. Dies erklärt jetzt auch endlich den Begriff der *kontextfreien Sprachen*: Bei ihren Regeln ist der Kontext der Variablen A während der Ersetzung irrelevant, sie kann frei von irgendwelchen Kontexteinschränkungen durch ihre rechte Seite ersetzt werden.

Definition 3.9.2. Eine *kontextabhängige Grammatik* ist also ein Tupel $G = (V, T, P, S)$ wobei P eine Menge von Regeln $\alpha A \beta \to \alpha y \beta$ mit einer Variablen $A \in V$ und Satzformen $\alpha, \beta, y \in (V \cup T)^*$ und $y \neq \varepsilon$ ist.

– Eine *Ableitung* $\sigma \Rightarrow \tau$ mit Hilfe der Regel $A \to y$ ist möglich, falls $\sigma = \sigma_1 \alpha A \beta \sigma_2$ ist, also A im Kontext (α, β) angetroffen wird, und $\tau = \sigma_1 \alpha y \beta \alpha_2$ aus der Ersetzung von A durch y hervorgeht.

– Mit $\overset{*}{\Rightarrow}$ bezeichnen wir wie vorher den reflexiv-transitiven Abschluss von $\Rightarrow$. Die *Sprache von* G ist dann $L(G) = \{w \in T^* \mid S \overset{*}{\Rightarrow} w\}$.

Offensichtlich ist jede kontextfreie Sprache auch kontextsensitiv, umgekehrt gilt dies nicht. Dies wollen wir an dem Standardbeispiel einer Sprache demonstrieren, von der wir schon wissen, dass sie nicht kontextfrei ist: $L_{a^n b^n c^n}$. Wir können eine kontextsensitive Grammatik für L angeben:

Beispiel 3.9.3. [Eine kontextsensitive Sprache]

$$\begin{array}{rcl} S & ::= & aSBC \\ & | & \varepsilon \\ CB & \rightarrow & BC \\ aB & \rightarrow & ab \\ bB & \rightarrow & bb \\ bC & \rightarrow & bc \\ cC & \rightarrow & cc \end{array}$$

Die erste Regel ist eigentlich verboten, weil $S \rightarrow \varepsilon$ vorhanden ist und das Startsymbol S auf der rechten Seite auftaucht. Wir könnten sie aber durch folgende äquivalente Regeln ersetzen:

$$\begin{array}{rcl} S & \rightarrow & S' \\ S' & \rightarrow & aS'BC \end{array}$$

Die letzten vier Regeln sind kontextabhängig: $aB \rightarrow ab$ besagt, dass B durch b ersetzt werden kann, sofern B hinter einem a steht. Der Kontext, in dem B durch b ersetzt werden darf, ist also (b, ε). Ähnlich verhält es sich mit den folgenden Regeln.

Die Regel $CB \rightarrow BC$ ist als einzige nicht kontextfrei. Aber auch diese kann mit Hilfe eines zusätzlichen Nonterminals X durch folgende drei kontextabhängige Regeln ersetzt werden.

$$\begin{array}{rcl} CB & \rightarrow & CX \\ CX & \rightarrow & BX \\ BX & \rightarrow & BC \end{array}$$

Dies erinnert an den Programmiertrick, bei dem man mittels einer Hilfsvariablen den Inhalt zweier Variablen vertauscht. Betrachten wir die komplette Grammatik, so erkennen wir, dass zunächst mit der ersten Regel eine Satzform $a^n S(BC)^n$ erzeugt werden kann. Erst nachdem dann S zu ε reduziert wurde, greifen die restlichen Regeln. Das erste B wird zu einem b, aber bevor es weitergeht, muss mit der Regel $CB \rightarrow BC$ die zukünftige Reihenfolge (b's vor c's) ermöglicht werden. Der Reihe nach kann man jetzt mit den anderen Regeln die B's in b's und die C's in c's verwandeln. Wir führen dies am Beispiel des Wortes $a^2b^2c^2$ vor. :

$$\begin{aligned} S &\Rightarrow aSBC \\ &\Rightarrow aaSBCBC \\ &\Rightarrow aaBCBC \\ &\Rightarrow aabCBC \\ &\Rightarrow aabBCC \\ &\Rightarrow aabbCC \\ &\Rightarrow aabbcC \\ &\Rightarrow aabbcc. \end{aligned}$$

Die Sprache $L_{a^n b^n c^n} = \{a^n b^n c^n \mid n \in \mathbb{N}\}$ haben wir schon vorher (siehe Beispiel 3.6.8) als nicht kontextfrei erkannt. Damit ist die obige Grammatik auch nicht äquivalent zu einer kontextfreien Sprache.

Satz 3.9.4. *Kontextsensitive Sprachen sind entscheidbar.*

Dies kann man auf die Tatsache zurückführen, dass mit Ausnahme der Regel $S \to \varepsilon$ für alle anderen Regeln $\alpha \to \beta$ gilt: $|\alpha| \leq |\beta|$. Bei der Ableitung $S \overset{\star}{\Rightarrow} w$ entstehen zwischenzeitlich nur Satzformen, deren Länge kleiner oder gleich $|w|$ ist. Da P endlich ist, könnte man systematisch alle Ableitungen bis zu einer Tiefe, die durch $|w|$ bestimmt ist, erzeugen. Falls man dabei w nicht angetroffen hat, kann es nicht in der Sprache sein.

Dies ist der Anlass für folgende Definition:

3.10 Allgemeine Grammatiken

Definition 3.10.1. Eine allgemeine Grammatik $G = (V, T, P, S)$ lässt beliebige Produktionen $\alpha \to y$ zu, wobei α und y Satzformen sind.

Ein Ableitungsschritt mit der Produktion $\alpha \to \beta$ besteht in der Ersetzung einer Satzform $\sigma_1 \alpha \sigma_2$ durch $\sigma_1 \beta \sigma_2$. Man schreibt dann $\sigma_1 \alpha \sigma_2 \Rightarrow \sigma_1 \beta \sigma_2$. Die Sprache einer Grammatik ist

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

Allgemeine Grammatiken, die nicht kontextsensitiv sind, spielen in der Praxis keine Rolle. Sie existieren zwar, aber es ist nicht möglich, ein einfaches Beispiel einer solchen Grammatik anzugeben, die nicht schon äquivalent zu einer kontextsensitiven Grammatik wäre. Daher werden wir nur zur Kenntnis nehmen, dass es solche Grammatiken gibt. Man kann zeigen, dass die durch eine allgemeine Grammatik definierbaren Sprachen $L \subseteq T^\star$ genau die aufzählbaren Teilmengen von $T^\star$ sind, also

diejenigen Teilmengen, deren Elemente man durch ein Computerprogramm systematisch erzeugen lassen kann. Wir nehmen ohne Beweis zur Kenntnis:

Satz 3.10.2. *Eine Sprache $L \subseteq T^*$ ist genau dann durch eine allgemeine Grammatik definierbar, wenn sie rekursiv aufzählbar ist.*

Unter diesen Sprachen finden sich auch *unentscheidbare* Sprachen, also solche $L \subseteq T^*$, für die es nachweislich kein Programm gibt, das zu einem vorgelegten Wort $w \in T^*$ entscheiden kann, ob w zu L gehört oder nicht. Der Grund, warum dies so schwierig sein kann, ist, dass beliebige Grammatiken auch k*ontraktive Regeln* besitzen können, also solche Regeln $\alpha \to \beta$, für die $|\beta| > |\alpha|$ ist. Eine Ableitung $S \Rightarrow^* \ldots\ y \Rightarrow \delta \ldots^* \Rightarrow w$ kann damit zwischenzeitlich größere und kleinere Satzformen produzieren.

Beispiele unentscheidbarer Sprachen wird uns Kapitel 5 als Konsequenz des „Halteproblems“ liefern.

An dieser Stelle wollen wir eine Sprache vorstellen, die zwar aufzählbar ist, aber als Konsequenz des Halteproblems nicht entscheidbar:

Sei $L = \{w \in ASCII^* \mid w \text{ ist terminierendes Python Programm}\}$. Da das Halteproblem (siehe S. 5.7.3 auf Seite 213) besagen wird, dass es kein Entscheidungsverfahren für diese Sprache gibt, kann sie nicht vom Chomsky-Typ 1 sein, denn kontextabhängige Sprachen sind entscheidbar. Um nachzuweisen, dass L wenigstens vom Typ 0 ist, müssen wir ein Verfahren skizzieren, mit denen wir alle Elemente von L aufzählen können:

Zunächst können wir alle Texte $w \in Ascii^*$ aufzählen - erst alle der Länge 1, dann alle der Länge 2 etc. Jedesmal füttern wir den Text in einen Compiler oder Interpreter, und werfen den Text weg, wenn er syntaktisch falsch ist. Andernfalls starten wir einen Interpreter mit dem Programmtext. Ohne auf das Ergebnis zu warten, machen wir schon mit dem nächsten Text weiter - wir geben diesen an eine andere Python-Maschine, etc.. Mit der Zeit haben wir viele - und stets mehr Maschinen, die den jeweils ihnen überlassenen Programmtext abarbeiten. Sobald eine davon terminiert, spuckt unser Hauptprogramm den von ihr getesteten Programmtext w aus. Auf diese Weise erhalten wir in einer nicht kontrollierbaren Reihenfolge alle Worte der Sprache L, wir haben also ein (für die Praxis hoffnungslos ineffizientes) Aufzählungsverfahren.

Definition 3.10.3. Eine Regel $\alpha \to \beta$ heißt *beschränkt*, falls $|\alpha| \leq |\beta|$ ist. Eine Grammatik heißt beschränkt, wenn jede ihrer Regeln beschränkt ist und eine Sprache L heißt beschränkt, falls es eine beschränkte Grammatik G gibt mit $\mathcal{L}(G) = L$.

Jede kontextsensitive Regel $\kappa_1 A \kappa_2 \to \kappa_1 \alpha \kappa_2$ (mit $\alpha \neq \varepsilon$) ist selbstverständlich beschränkt. Damit erhalten wir sofort eine Richtung des folgenden Satzes. Die Regel $CB \to BC$ in der Grammatik von Beispiel 3.9.3 ist nicht kontextfrei, sie ist aber beschränkt. Wir hatten gesehen, wie wir sie durch eine kontextfreie Regel ersetzen

konnten. Dass eine analoge Ersetzung beschränkter Regeln durch kontextfreie Regeln immer möglich ist, wollen wir hier ohne Beweis akzeptieren:

Satz 3.10.4. *Eine ε-freie Sprache $\mathcal{L}(G)$ ist genau dann beschränkt, wenn sie kontextabhängig ist.*

3.11 Die Chomsky-Hierarchie

Chomsky klassifizierte Grammatiken in drei Typen, die man heute zu seinen Ehren als als Chomsky-0, Chomsky-1, Chomsk-2 und Chomsky-3 bezeichnet.

Typ-0	alle Grammatiken,
Typ-1	die kontextsensitiven Grammatiken. Diese sind äquivalent zu den beschränkten Grammatiken,
Typ-2	die kontextfreien Grammatiken,
Typ-3	die links-linearen Grammatiken.

Offensichtlich ist jede linkslineare Grammatik auch kontextfrei und jede kontextfreie Grammatik auch kontextsensitiv, somit haben wir die Inklusionen

$$Typ\ 3 \subset Typ\ 2 \subset Typ\ 1 \subset Typ\ 0.$$

Genauso sagt man, dass eine Sprachen $L \subseteq T^*$ vom Typ $Chomsky - i$ ist, wenn es eine Grammatik G von Chomsky-Typ i gibt mit $L = L(G)$.

Dass die Inklusionen echt sind, haben wir in den meisten Fällen bereits durch Beispiele belegt:

- L_{Dyck} und $L_{a^n b^n} = \{a^n b^n \mid n \in \mathbb{N}\}$ sind Chomsky-2 (Bsp.: 3.2.6), aber nicht Chomsky-3 (Bsp.: 2.4.20).
- $L_{a^n b^n c^n} = \{a^n b^n c^n \mid n \in \mathbb{N}\}$ ist Chomsky-1 (siehe Bsp. 3.9.3), aber nicht Chomsky-2 (Bsp. 3.6.8),
- $L_{py} = \{w \in ASCII^* \mid w$ ist ein terminierendes Python-Programm$\}$ ist Chomsky-0 aber nicht Chomsky-1.

Zu den einzelnen Sprachklassen gibt es entsprechende Maschinenmodelle, die genau die Sprachen der zugehörigen Klasse erkennen:

Typ-3	die Sprachen, die von endlichen Automaten erkannt werden. Dabei spielt es keine Rolle, ob wir uns auf deterministische Automaten beschränken, oder nicht, auch nicht, ob wir ε-Transitionen zulassen, oder nicht.
Typ-2	diese Sprachen kann man mit einem Stackautomaten (Kellerautomaten) erkennen. Dabei spielt es keine Rolle, ob Zustände vorhanden sind,

oder nicht, bzw. ob als Erkennung nur der leere Stack dient, oder der leere Stack in Verbindung mit einem Endzustand. Es gibt zwar auch den Begriff des deterministischen Stackautomaten. Allerdings gibt es Typ-2-Sprachen, die nicht von einem deterministischen Stackautomaten erkannt werden können.

Typ-1 solche Sprachen lassen sich von linear beschränkten Turingmaschinen erkennen. An dieser Stelle können wir uns den Begriff der Turingmaschine als Synonym für einen Rechner vorstellen. *Linear beschränkt* heißt, dass wir dem Rechner zur Erkennung von w einen begrenzten Speicherraum zur Verfügung stellen, dessen Größe linear von der Länge $|w|$ des Eingabewortes abhängen darf.

Typ-0 dies sind genau die aufzählbaren Sprachen, also diejenigen, für die ein Computerprogramm geschrieben werden kann, das jedes Element der Sprache ausdruckt. Es gibt keine Begrenzung des Speicherplatzes. Allerdings darf man nicht erwarten, dass die Wörter der Sprache in einer bestimmten Reihenfolge ausgedruckt werden. Wenn ein Wort w, sei es noch so kurz, nach beliebig langer Zeit noch nicht erschienen ist, ist dies kein Hinweis darauf, ob es noch kommen wird, oder nicht.

Figur 3.11.1 fasst diese Ergebnisse noch einmal zusammen und zeigt auch je eine Sprache, die in Typ i ist, nicht aber in Typ $(i + 1)$:

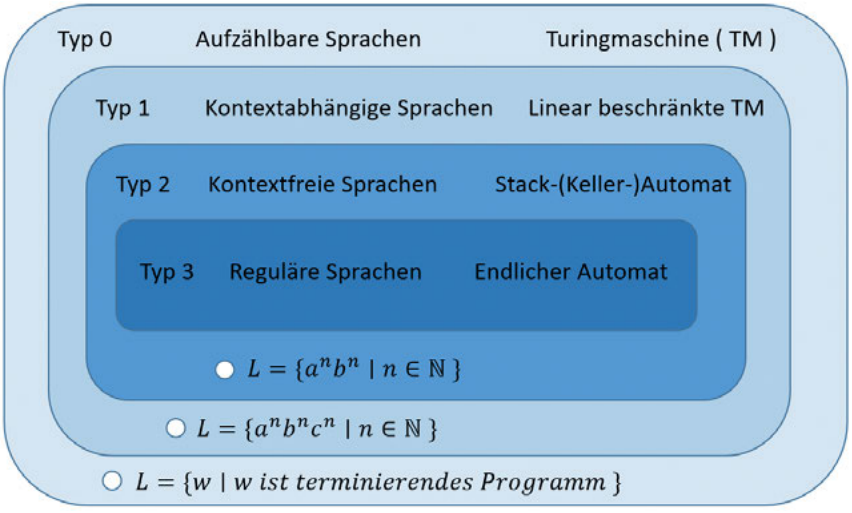


Abb. 3.11.1: Chomsky-Hierarchie

Kapitel 4

Compilerbau

Ein *Compiler* für eine Programmiersprache muss als erstes prüfen, ob eine vorgelegte Datei ein syntaktisch korrektes Programm enthält. Ihm liegt der Quelltext als String, also als Folge von Zeichen vor.

Die Analyse des Textes zerfällt in zwei Phasen – die lexikalische Analyse und die *syntaktische Analyse*. Dies entspricht in etwa auch unserem Vorgehen bei der Analyse eines fremdsprachlichen Satzes: In der ersten Phase erkennen wir die Wörter, aus denen der Satz besteht – vielleicht schlagen wir sie in einem Lexikon nach – und in der zweiten Phase untersuchen wir, ob und wie die Wörter zu einem grammatikalisch korrekten Satz zusammengefügt sind. Als laufendes Beispiel betrachten wir eine Grammatik die so oder ähnlich vielen imperativen Programmiersprachen zugrunde liegt:

Programm	::=	*Sequenz* **.**
Anweisung	::=	*Zuweisung* | *Schleife* | *Alternative* | *Sequenz*
Zuweisung	::=	*id* **:=** *Expr*
Schleife	::=	***WHILE*** *Bexpr* ***DO*** *Anweisung*
Alternative	::=	***IF*** *Bexpr* ***THEN*** *Anweisung*
	|	***IF*** *Bexpr* ***THEN*** *Anweisung* ***ELSE*** *Anweisung*
Sequenz	::=	***BEGIN*** *Anweisungen* ***END***
Anweisungen	::=	$\boldsymbol{\varepsilon}$ | *Anweisung* | *Anweisung* **;** *Anweisungen*
Expr	::=	*Expr* + *Expr* | *Expr* – *Expr* | *Expr* * *Expr* | (*Expr*)
	|	***id*** | ***num***
Bexpr	::=	*Expr* = *Expr* | *Expr* < *Expr*
	|	***NOT*** *Bexpr* | ***True*** | ***False***

Ein Programm ist demnach eine in **BEGIN** und **END** eingeschlossene Folge von Anweisungen, gefolgt von einem Punkt. Es gibt 4 verschiedene Anweisungen: eine

https://doi.org/10.1515/9783110442397-005

Zuweisung, eine *Schleife*, eine *Alternative* oder eine *Sequenz*. Neben den (hier groß geschriebenen Schlüsselwörtern (**WHILE**, **DO**, **IF**, **ELSE, THEN**, **BEGIN**, **END**, **NOT**) gibt es die Token **id** und **num**, welche für Variablennamen (***id****entifier*) und Zahlkonstanten (***num****bers*) stehen.

Bei der Spracherkennung wird der konkrete Wert (Name und Typ) des ***id***, bzw. die konkrete Zahl die sich hinter dem Token ***num*** verbirgt, noch keine Rolle spielen. Erst später, bei der Codeerzeugung, wird dieser Wert benötigt. Alle anderen Token sind entweder Operatoren (+, **-**, *****,<, =, **:=**) oder „Satzzeichen“ wie der Punkt „**.**“, oder die Klammern „**(**“, „**)**“. Das Semikolon „**;**“ kann man als Operator deuten, der aus einer *Anweisung* und einer Folge von *Anweisungen* eine neue Folge von *Anweisungen* macht.

4.1 Lexikalische Analyse

In dieser ersten Phase wird der vorgebliche Programmtext in Wörter zerlegt. Alle Trennzeichen (Leerzeichen, *tab*, *newLine*) und alle Kommentare werden entfernt. Aus einem einfachen WHILE-Programm, wie

```
BEGIN
   x := 54;
   y := 30;
   WHILE NOT x = y DO
      IF x > y THEN x := x-y
               ELSE y:= y-x;
   writeln(x)
END.“
```

wird dabei eine Folge von *Token*. Diesen englischen Begriff kann man mit Gutschein übersetzen. Für jede ganze Zahl erhält man z.B. ein Token ***num***, für jeden Bezeichner ein Token ***id***, für jeden String ein Token ***str***. Gleiche Token stehen für gleichartige Wörter. Andere Token, die in dem Beispielprogramm vorkommen, sind

eq (=) , ***gt*** (>), ***minus*** (**-**), ***assignOp***(**:=**), ***klAuf*** (**(**),
klZu(**)**), ***komma*** (**,**), ***semi*** (**;**), ***punkt*** (**.**).

Jedes Schlüsselwort bildet ein Token für sich. Nach dieser Zerlegung ist aus dem Programm eine Folge von Token geworden. Aus dem obigen Programm wurde:

begin id assignOp num semi id assignOp num semi while
not id eq id do if id gt id then id assignOp id minus id else id
assignOp id minus id semi id klAuf id klZu end punkt.

Aus den Token wie sie der Benutzer schreibt ist die interne Repräsentation der Token

geworden, also aus „x := 54 ;“ beispielsweise „***id assignOp num semi***. Damit ist die erste Phase, die lexikalische Analyse, abgeschlossen.

4.1.1 Endlicher Automat als Scanner

Diese erste Phase ist nicht schwer, aber auch nicht trivial. Es muss u.a. entschieden werden, ob „*BeGin*“ das gleiche ist, wie „*BEGIN*“, ob „beginnen“ ein ***id*** ist, oder das Schlüsselwort „**begin**“ gefolgt von dem Bezeichner „nen“. Schließlich muss erkannt werden, ob 00.23e-001, .523e, 314.e-2 das Token ***num*** repräsentieren, oder nicht.

Jedes Token ist durch einen regulären Ausdruck beschrieben und besitzt daher einen erkennenden Automaten. Ein Scanner schaltet alle diese Automaten zusammen. Derjenige Automat, der durch einen möglichst langen Präfix einer Zeile des Inputstrings in einen Endzustand überführt wird, gewinnt und liefert sein Token ab. Der zugehörige Teil des Inputstrings wird abgeschnitten und der Scanner setzt seine Arbeit auf dem Rest des Inputs fort. So entsteht in der ersten Phase eine Folge von Tokens.

Heutzutage werden Scanner aus einer Beschreibung der Token durch reguläre Ausdrücke automatisch erzeugt. Ein *Scannergenerator* transformiert eine Beschreibung von Token durch reguläre Ausdrücke in einen erkennenden Automaten, der z.B. durch ein C-Programm realisiert wird. Der bekannteste frei erhältlicher *Scannergenerator* heißt *flex* und gehört zu jeder Linux-Distribution. *flex* erwartet als Input eine Textdatei mit regulären Ausdrücken und zugehörigen Token. In dem Beispiel von Algorithmus 4.1 wird dem regulären Ausdruck `{alpha}{alpha_num}*` das Token `ID` zugeordnet und dem Input „WHILE“ das Token ***while***. Diese Token sind in flex einfach als C-Konstanten definiert.

Algorithmus 4.1 flex-Input

```
white_space                [ \t]*
digit                      [0-9]
unsigned_integer           {digit}+
alpha                      [A-Za-z_]
alpha_num                  ({alpha}|{digit})
hex_digit                  [0-9A-F]
%%
{white_space}              /* do nothing */
{alpha}{alpha_num}*        Return(ID);
{unsigned_integer}                          {   printf(atof(yytext));
return(UNSIGNED_INTEGER);}
0x{hex_digit}{hex_digit}*  return(HEX_INTEGER);
while                     return(WHILE);
%%
```

Eine flex-Datei besteht aus drei Abschnitten, die durch '**%%**' getrennt werden. Im ersten Teil werden Abkürzungen für reguläre Ausdrücke definiert. Im zweiten Teil wird in jeder Zeile einem regulären Ausdruck eine C-Anweisung zugeordnet. Wenn der Ausdruck im Input erkannt wurde, wird die zugehörige Anweisung ausgeführt. Im einfachsten Falle besteht diese Anweisung aus der Rückgabe eines Tokens oder gar der leeren Anweisung. Oft tragen die Token aber noch Zusatzinformationen, so ist beispielsweise für ein Integer-Token auch dessen Zahlenwert von Relevanz. In der Variablen **yytext** befindet sich jederzeit dasjenige Textstück des Quelltextes, das dem erkannten regulären Ausdruck entspricht. Wird etwa ein **UNSIGNED_INTEGER** erkannt, so kann durch Umwandlung des aktuellen Strings in `yytext` dessen Zahlenwert (z.B. mit der C-Funktion **atof( )**, die eine in ASCII-Text geschriebene Zahl in eine *float*-Zahl umwandelt) ermittelt und verarbeitet werden.

Der dritte und letzte Abschnitt der *flex*-Datei enthält typischerweise weitere C-Definitionen, gegebenenfalls sogar eine **main()**-Funktion. Mit Hilfe dieser lässt sich *flex* auch jenseits des Compilerbaus als nützliches Werkzeug verwenden.

4.2 Syntaxanalyse

Wir nehmen an, dass unser Programm die lexikalische Analyse gut überstanden hat. Dann folgt als nächstes die sogenannte Syntaxanalyse. Hier wird geprüft, ob die gefundenen Token in der vorliegenden Reihenfolge ein grammatikalisch korrektes Programm bilden.

Fasst man die Menge der Token T als neues Alphabet auf, so ist ein Programm ein Wort über T, also ein Element von T^*. Die Menge aller korrekten Programme ist also eine Sprache über der Tokenmenge T. Auf dieser Ebene kommen wir nicht mehr mit endlichen Automaten aus, da wir hier geschachtelte Strukturen vor uns haben — Funktionen mit Unterfunktionen, Klassen mit Unterklassen, Ausdrücke mit Unterausdrücken, etc.. In unserer obigen Beispielgrammatik sehen wir, dass z.B. eine *Anweisung* mittelbar über *Schleife*, *Alternative* oder *Sequenz* wiederum auf den Begriff *Anweisung* rekurriert. Die Dyck-Sprache hat uns schlaglichtartig gezeigt, dass geschachtelte Strukturen mindestens die Beschreibung durch kontextfreie Grammatiken zwingend erfordern. Alleine das Regelfragment

Expr ::= *Expr* + *Expr* | **(** *Expr* **)** | **id** | ...

beinhaltet die Dyck-Sprache wenn man die Token '+' und **id** ignoriert (bzw. durch ε ersetzt).

Wir benötigen also zumindest kontextfreie Grammatiken. Andererseits gibt es wenig Anlass, darüber hinaus Sprachen durch kontext-abhängige Grammatiken zu beschreiben. Lieber greift man von Hand ein, um gewisse Sonderfälle, wie auch z.B. das „*if-else-Problem*“ zu behandeln. Wir bleiben also dabei, dass wir Programmierspra-

chen durch kontextfreie Grammatiken beschreiben, und für die Erkennung solcher Sprachen reicht, wie wir in Abschnitt 3.7 gesehen haben, prinzipiell eine nichtdeterministische Stackmaschine.

4.2.1 Parsing

Als *Parsen* bezeichnet man den zur Ableitung umgekehrten Weg. Gegeben eine Grammatik G und ein Programm P, finde heraus, ob $P \in L(G)$ ist. Für den Compilerbau will man sogar etwas mehr wissen:

„*Wie kann man P mit Hilfe von G aus dem Startsymbol S ableiten?*“

4.2.2 Parsen durch rekursiven Abstieg (recursive descent)

Bevor wir diese Frage in voller Allgemeinheit beantworten, wollen wir eine einfache und für viele praktisch relevante Grammatiken anwendbare Methode verraten, die Methode des *rekursiven Abstiegs* (engl.: *recursive descent*). Derart konstruierte Parser heißen *RD-Parser*.

Die Kernidee ist, für jedes Nonterminal A einen eigenen Parser *parse_A* zu schreiben. Ist

$$A \to \alpha$$

eine Produktion für A, so übersetzt sich im Rumpf von *parse_A* jedes Nonterminal B in in einen Aufruf von *parse_B*.

Terminale werden im Inputstream *erwartet* und *akzeptiert* oder auch „konsumiert“. Dabei rückt die Leseposition im Inputstream der Nonterminale um eine Position weiter.

Die Struktur des Parsers folgt genau den Regeln der Grammatik:

Jede einzelne Regel beschreibt eine Prozedur zum Erkennen der auf der linken Seite der Regel angegebenen syntaktischen Einheit.

Die Regel für eine Schleife

Schleife → ***while*** *Bexpr* ***do*** *Anweisung*

besagt demnach:

Um eine Schleife zu erkennen
    akzeptiere ein ***while***
    erkenne ein Bexpr
    akzeptiere ein ***do***

erkenne eine Anweisung.

Analog werden alle Regeln in Prozeduren (Methoden) umgesetzt. Die rekursive Struktur der Regeln hat entsprechend rekursive Prozeduren zur Folge. Der rekursive Abstieg endet jeweils mit dem Akzeptieren eines erwarteten Tokens.

Die Methode des Parsers für das Erkennen einer Schleife lautet nun:

```
void parse_Schleife(){
    akzeptiere(while);
    parse_Bexpr();
    akzeptiere(do);
    parse_Anweisung(); }
```

Ein Problem tritt auf, wenn es zu einem Nonterminal mehrere rechte Seiten gibt, wie z.B. im Falle von *Anweisung* oder *Anweisungen*. Wir verschieben für einen Moment das Problem, indem wir uns eine magische *ODER*-Anweisung vorstellen, mit der wir die verschiedenen Möglichkeiten in der *Parse*-Funktion kombinieren. Aus

$$Anweisung \rightarrow Zuweisung \mid Schleife \mid Alternative \mid Sequenz$$

wird dann zu einer nichtdeterministischen Auswahl

```
void parse_Anweisung( ){
    parse_Zuweisung();
  ODER
    parse_Schleife();
  ODER
    parse_Alternative();
  ODER
    parse_Sequenz();
}
```

und aus

$$Anweisungen \rightarrow \varepsilon \mid Anweisung \mid Anweisung\,;\, Anweisungen$$

wird.

```
void parse_Anweisungen( ){
    return; //       leeres Wort
  ODER
    parse_Anweisung();
```

```
  ODER
    parse_Anweisung();
    akzeptiere(semi);
    parse_Anweisungen();
}
```

Der Aufruf `akzeptiere(t)` erwartet das Token `t` im Input. Ist das nächste Token tatsächlich `t`, so war `akzeptiere` erfolgreich und der Inputzeiger rückt vor zum nächsten Token. Ist dies nicht der Fall, wird eine Fehlermeldung „`Token t expected`“ erzeugt.

Zu jedem Nonterminal *A* gehört eine entsprechende Methode `parse_A`. Diese Methode, einen Parser zu schreiben ist einfach verständlich und in jeder Programmiersprache, die Rekursion zulässst, einfach zu realisieren. Allerdings gibt es zwei Probleme, die dabei auftauchen können, und die man größtenteils schon schon auf der Ebene der Grammatik lösen kann.

Dazu kann man die Grammatik in eine äquivalente Grammatik umwandeln, die diese Probleme vermeidet. Eines dieser Probleme stellen sogenannte *linksrekursive Produktionen* dar, wie wir sie z.B. in jeder *Expression*-Grammatik vorfinden.

4.2.3 Äquivalente Grammatiken

Die Produktion

$$Expr ::= Expr + Expr | \ldots$$

führt bei unmittelbarer Übersetzung zu dem Code

```
void parse_Expr(){
  parse_Expr() ;
  akzeptiere(plus);
  parseExpr();
ODER ....
}
```

Die Methode `parse_Expr` ruft sich unmittelbar selbst auf, bevor sie irgendetwas aus dem Input verarbeitet wurde. Es ist klar, dass eine solche Methode nie terminieren wird. Glücklicherweise haben wir aber bereits in Abschnitt 3.4.4 gesehen, wie man die Grammatik umschreiben kann, um sich solcher Linksrekursionen zu entledigen.

4.2.4 Linksfaktorisierung

Das zweite Problem ist der Nichtdeterminismus, den wir oben mit dem mysteriösen „ODER“ überspielt haben. Im Falle der Produktionen für Boolesche Ausdrücke, *Bexpr*, gibt es zwei Produktionen, die potenziell mit demselben Token beginnen können:

$$\begin{aligned} Bexpr &::= Expr = Expr \\ &\mid Expr < Expr \\ &\mid NOT\ Bexpr \mid True \mid False \end{aligned}$$

Hier fassen wir einfach die Produktionen, die mit der gleichen Satzform beginnen, zusammen und führen für den verschiedenen Rest ein neues Nonterminal ein. Aus dem obigen wird dann

$$\begin{aligned} Bexpr &::= Expr\ RestBexpr \\ &\mid NOT\ Bexpr \mid True \mid False \end{aligned}$$

$$\begin{aligned} RestBexpr &::= \text{'='}\ Expr \\ &\mid \text{'<'}\ Expr \end{aligned}$$

Man nennt diese Vorgehensweise auch Links-Faktorisierung. Es ist eine Art Distributivgesetz, das die gemeinsamen Anteile einer Alternative zusammenfasst und herauszieht. Jetzt kann man sowohl `parse_Bexpr` als auch `parse_RestBexpr` deterministisch implementieren. In der Variablen `lookahead` liegt stets das nächste Token des Eingabestroms:

```
void parse_Bexpr(){
  if (lookahead == not){
    akzeptiere(not);
    parse_Bexpr();
  else
    parse_Expr();
    parse_RestBexpr();
  }
```

In `parse_RestBexpr` können wir uns ebenfalls anhand des ersten Tokens (=, <) festlegen, in welche der Produktionen wir verzweigen müssen.

Allerdings kann ein Sonderfall auftreten, nämlich, dass eine der Alternativen ε wird. Das wäre beispielsweise der Fall, wenn wir die *Alternative* faktorisieren würden:

$$
\begin{aligned}
\mathit{Alternative} &::= \textbf{\textit{if}}\ \mathit{Bexpr}\ \textbf{\textit{then}}\ \mathit{Anweisung}\ \mathit{RestAlternative} \\
\mathit{RestAlternative} &::= \textbf{\textit{else}}\ \mathit{Anweisung} \\
&\mid\ \varepsilon
\end{aligned}
$$

Falls wir eine *RestAlternative* parsen und ein **else** im Input sehen, ist es dann sicher, dass dass nur die erste Klausel richtig ist? Könnte das **else** nicht von einer umfassenden Anweisung übrig sein, hier aber die RestAlternative beendet sein?

In der Tat ist dies hier der Fall. Stellen wir uns einen Programmtext der folgenden Struktur vor:

$$\textbf{\textit{if}}\ B_1\ \textbf{\textit{then if}}\ B_2 \textbf{\textit{then}}\ A_1 \textbf{\textit{else}}\ A_2.$$

Während wir die innere Alternative $\textbf{\textit{if}}\ B_2\ \textbf{\textit{then}}\ A_1$ parsen, gelangen wir zu dem ***else***. Wir müssen jetzt entscheiden, ob das ***else*** im Input noch zu der aktuellen Alternative gehören soll, oder zu der äußeren Alternative $\textbf{\textit{if}}\ B_1\ \textbf{\textit{then}} \ldots \textbf{\textit{else}}\ A_2$.

4.3 LL(1)-Grammatiken

In vielen Fällen ist der Nichtdeterminismus der aus einer Regel

$$A \to \alpha_1 \mid \alpha_2 \mid \ldots \alpha_n \tag{4.3.1}$$

erwächst, leicht auflösbar. Insbesondere ist dies der Fall, wenn jedes α_i mit einem Token beginnt und diese untereinander verschieden sind. In diesem Falle kann sich ein Parser *parse_A* anhand des nächsten Tokens im Eingabestrom für eine der Alternativen entscheiden. Oft kann man, wie oben gesehen, durch Linksfaktorisierung die Grammatik noch in eine geeignete Form bringen, so dass aufgrund des nächsten Tokens jeweils die Entscheidung für die richtige Alternative getroffen werden kann. Dies geht aber nicht immer, wie wir am Beispiel des *if-then-else* gesehen haben.

Allgemein verstehen wir unter $\mathit{First}(\alpha)$ für eine beliebige Satzform α die Menge aller Token, mit denen ein aus α abgeleitetes Wort beginnen kann. Gilt dann

$$\mathit{First}(\alpha_i) \cap \mathit{First}(\alpha_j) = \emptyset$$

für alle $i \neq j$, so kann der Nondeterminismus in (4.3.1) eindeutig anhand des nächsten Tokens aufgelöst werden. Eine Grammatik mit dieser Eigenschaft nennt man *LL(1)-Grammatik*.

In unserer *WHILE*-Grammatik ist diese Bedingung insbesondere im Falle der Anweisung erfüllt, da offensichtlich: $\mathit{First}(\mathit{Zuweisung}) = \{\textbf{\textit{id}}\}$, $\mathit{First}(\mathit{Alternative}) = \{\textbf{\textit{if}}\}$, $\mathit{First}(\mathit{Schleife}) = \{\textbf{\textit{while}}\}$, und $\mathit{First}(\mathit{Sequenz}) = \{\textbf{\textit{begin}}\}$ somit können wir das „ODER“ aus dem obigen Pseudocode für *parse_Anweisung* entfernen:

```
parse_Anweisung( ){
    if (lookahead == id) parse_Zuweisung();
    else if (lookahead == if) parse_Alternative();
    else if (lookahead == while) parse_Schleife();
    else if (lookahead == begin) parse_Sequenz();
    else error("id, if oder while erwartet"); }
```

4.3.1 First und Follow

Ein zusätzliches Problem tritt auf, wenn eine oder mehrere Alternativen einer Regel optional (d.h. ε) sind. Im Falle der *RestAlternative* kann das Token **else** einerseits das erste Token der *RestAlternative* sein, andererseits aber auch auf eine leere RestAlternative folgen. Wir müssen daher auch diejenigen token betrachten, die auf eine *RestAlternative* folgen können. Daher zählt man auch ε zu $First(\alpha)$, dann nämlich, wenn $\alpha \rightarrow^* \varepsilon$ in der Grammatik möglich ist. Formal lautet damit die Definition von *First* jetzt für eine beliebige Satzform α:

$$First(\alpha) = \{t \in T \mid \alpha \rightarrow^* t\beta\} \cup \{\varepsilon \mid A \rightarrow^* \varepsilon\}.$$

Im Fall der *Anweisungen* kann man entweder das Erkennen von *Anweisungen* gleich beenden oder die zweite Alternative versuchen. Dazu betrachtet man wieder das *lookahead*, also das nächste Token im Input. Ist es ein Token, das niemals auf *Anweisungen* folgen kann, so ist die zweite oder dritte Alternative angesagt. Auf *Anweisungen* kann in der bisherigen Grammatik nur das Token ***end*** folgen, wie man durch Inspektion der Regeln leicht sieht. Eine Anweisung muss aber mit einem der Token ***id***, ***if*** oder ***while*** beginnen. Somit können wir auch diese Unbestimmtheit in dem Parser für Anweisungen beseitigen:

```
void parse_Anweisung( ){
  switch (lookahead){
    case end: return; // keine weitere Anweisung
    case while:
    case if:
    case id:
         parse_Anweisung(); parseRestAnweisungen();
    default: error("end,id,if oder while erwartet");
  }
}
```

Mit $Follow(A)$ bezeichnet man die Menge aller Token, die in der Grammatik auf ein A folgen können. Die genaue Definition ist:

$$Follow(A) := \{t \in T \mid S \to^* \alpha A t \beta\}.$$

In unserer WHILE-Grammatik haben wir u.a.:

Follow(Anweisungen) = { **end** },
Follow(Anweisung) = { **;** , **else**, **end** },
Follow(Expr) = { **;** , + , **−** , * , **)** , = }.

4.3.2 Berechnung der First- und Follow-Mengen

$First(\alpha)$ lässt sich für jede Satzform α einfach algorithmisch bestimmen. Es besteht aus Terminalen und ggf. dem leeren Wort ε:

falls $\alpha = ty$ für ein Token t, setzen wir $First(\alpha) = \{t\}$
falls $\alpha = By$ für ein Nonterminal B ist und $B \to \beta$ eine Produktion, dann muss auch $First(\beta) - \{\varepsilon\} \subseteq First(\alpha)$ sein

falls $B \Rightarrow^* \varepsilon$ zusätzlich $First(y) \subseteq First(\alpha)$
falls $B \Rightarrow^* \varepsilon$ und $y \Rightarrow^* \varepsilon$ dann auch $\varepsilon \in First(\alpha)$

$Follow(A)$ ist für alle Nonterminale zu berechnen. Am besten führt man dies simultan für alle Nonterminale durch. Ein spezielles Token **eof** signalisiere das Ende des Inputs:

1. **eof** $\in$ Follow(S), wobei S das Startsymbol der Grammatik ist
2. Für jede Produktion $A \to \alpha X \beta$ muss $First(\beta) - \{\varepsilon\} \subseteq Follow(X)$.
3. Falls $\beta \Rightarrow^* \varepsilon$ muss zusätzlich $Follow(A) \subseteq Follow(X)$ sein.

Definition 4.3.1. Eine Grammatik heißt LL(1), falls für alle Regeln $A \to \alpha_1 | \ldots | \alpha_n$ und alle $i \neq j$ die folgenden Bedingungen erfüllt sind:

- $First(\alpha_i) \cap First(\alpha_j) = \emptyset$
- $\varepsilon \in First(\alpha_i) \implies Follow(A) \cap First(\alpha_j) = \emptyset$.

Für jede LL(1) Grammatik lässt sich auf die geschilderte Weise leicht ein recursive-descent Parser gewinnen, sofern Linksrekursionen vermieden werden können.

4.4 Ableitungsbaum, Syntaxbaum

Mit jeder Grammatik kann man eine Klasse von Bäumen assoziieren. Zu jeder Produktion $A \to \tau_1 \ldots \tau_k$, wobei jedes τ_i ein Token oder ein Nonterminal ist, gibt es einen

Knotentyp mit Wurzel A und Kindern $\tau_1, \ldots, \tau_n$. Ein aus diesen Knotentypen zusammengesetzter Baum mit dem Startsymbol S als Wurzel heißt *Ableitungsbaum*.

Die Konstruktion eines Ableitungsbaumes beginnt mit dem Startsymbol S als Wurzelknoten. In jedem Schritt wird ein mit einem Nonterminal, sagen wir A, beschriftetes Blatt des Baumes ausgewählt und eine Produktion für A, zum Beispiel $A \rightarrow \tau_1\tau_2\ldots\tau_k$. Sodann werden die Kinder $\tau_1, \tau_2, \ldots, \tau_k$ an den besagten Knoten angehängt.

Wir betrachten als Beispiel den algebraischen Ausdruck $x-2*(y+1)$, repräsentiert durch die Tokenfolge

$$id, -, num, *, (, id, +, num,)$$

für die wir eine Ableitung

$$Expr \rightarrow \ldots \rightarrow x - 2 * (y + 1)$$

suchen. Zwei der vielen möglichen Ableitungen sind:

$$\begin{aligned}
Expr &\Rightarrow Expr - Expr \\
&\Rightarrow id - Expr \\
&\Rightarrow id - Expr * Expr \\
&\Rightarrow id - num * Expr \\
&\Rightarrow id - num * (Expr) \\
&\Rightarrow id - num * (Expr + Expr) \\
&\Rightarrow id - num * (id + Expr) \\
&\Rightarrow id - num * (id + num)
\end{aligned}
\qquad
\begin{aligned}
Expr &\Rightarrow Expr * Expr \\
&\Rightarrow Expr - Expr * Expr \\
&\Rightarrow id - Expr * Expr \\
&\Rightarrow id - num * Expr \\
&\Rightarrow id - num * (Expr) \\
&\Rightarrow id - num * (Expr + Expr) \\
&\Rightarrow id - num * (id + Expr) \\
&\Rightarrow id - num * (id + num)
\end{aligned}$$

Die zugehörigen Ableitungsbäume sind in Figur 4.4.1 dargestellt.

In beiden Fällen liest man die Folge der Eingabetoken von den Blättern ab. Offensichtlich fasst der linke Ableitungsbaum den Ausdruck als Differenz auf, der rechte als Produkt. Letztere Interpretation ist aber nicht erwünscht. Die höhere Priorität von '*' über '-' wird also von der Grammatik nicht berücksichtigt.

Solange es nur darum geht, zu erkennen, ob ein vorgelegtes Programm ein Wort aus einer Sprache darstellt, ist die Mehrdeutigkeit irrelevant. Anders sieht es aus, wenn der konstruierte Ableitungsbaum als Basis für eine Übersetzung oder Auswertung dienen soll.

4.4.1 Abstrakte Syntaxbäume

Abstrakte Syntaxbäume sind kompaktere Darstellungen von Ableitungsbäumen. Zu jeder Produktion $A \rightarrow \alpha$ gibt es einen Knotentyp, der nur *zu jedem Nonterminal in* α einen Sohn besitzt. Terminale werden in der Regel nicht dargestellt, da aus dem

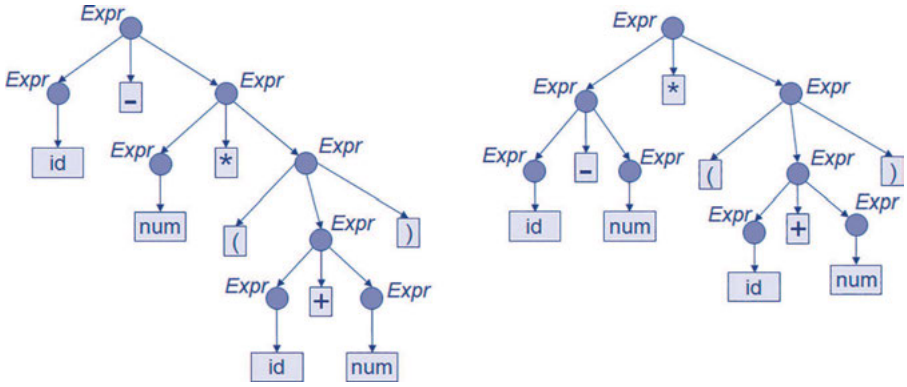


Abb. 4.4.1: Zwei Ableitungsbäume für $x - 2 * (y + 1)$

Knotentyp auf ihre Existenz geschlossen werden kann. Ausgenommen sind Terminale, für die eine Zusatzinformation bewahrt werden muss – für einen *id* der Name, oder für ein *num* der Zahlenwert. Abstrakte Syntaxbäume mit dieser Zusatzinformation sind Ausgangspunkte für Übersetzer und Interpreter. Im Falle von arithmetischen Ausdrücken entsprechen die Terminale außer *id* und *num* gerade den Operatoren, wie z.B. in $Expr \rightarrow Expr + Expr$ oder in $Expr \rightarrow -Expr$, so dass die Benutzung dieser Regeln zum Aufbau eines „+-Knotens“ bzw. eines „—Knotens“ im abstrakten Syntaxbaum führt. Klammern kann man in diesem Zusammenhang weglassen, denn eine Regel wie

$$Expr \rightarrow (\, Expr \,)$$

führt zwar zu einem Knoten mit einem Sohn, allerdings ist der zugehörige Operator die Identität. Figur 4.4.2 zeigt die abstrakten Syntaxbäume, die zu den vorhin gesehenen Ableitungen des Ausdrucks $x - 2 * (y + 1)$ gehören. Wieder wird deutlich, dass der rechte Syntaxbaum unerwünscht ist.

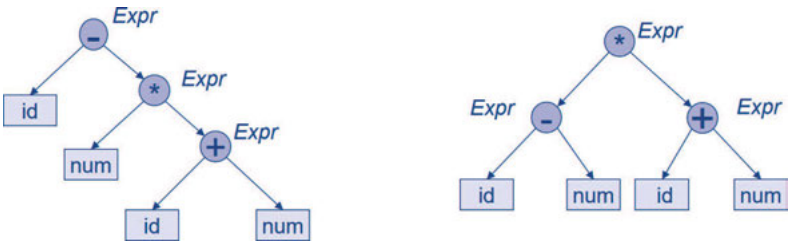


Abb. 4.4.2: Abstrakte Syntaxbäume für x-2*(y+1)

Erwünscht wäre eine *eindeutige* Grammatik, die für jeden arithmetischen Ausdruck nur einen Ableitungsbaum zulässt. Um unsere *Expr*-Grammatik eindeutig zu machen, führen wir neue Nonterminale ein, die Unterausdrücke einer bestimmten Prioritätsstufe kennzeichnen (siehe auch Abschnitt 3.3). Wir verwenden dazu die Nonterminale *Expr*, *Term* und *Faktor* sowie die Regeln:

$$\begin{aligned} Expr &\rightarrow Expr + Term \mid Expr - Term \mid Term \\ Term &\rightarrow Term * Faktor \mid Faktor \\ Faktor &\rightarrow id \mid num \mid (Expr) \end{aligned}$$

Expr sind mit dieser Grammatik also Summen und Differenzen von Termen, *Terme* sind Produkte von Faktoren und die *Faktor*en *id*, *num* und *(Expr)* sind die Ausdrücke der höchsten Prioritätsstufe. Mit dieser Grammatik sind arithmetische Ausdrücke eindeutig ableitbar.

Betrachten wir beispielsweise unseren vorigen Ausdruck $x - 2 * (y + 1)$, so ist die fälschliche Ableitung als Produkt von $x - 2$ und $(y + 1)$ nicht mehr möglich, da $x - 2$ nicht als *Term* ableitbar ist. Auch Linksklammerung wird korrekt gehandhabt: $x - y - z$ kann nur als Differenz von $x - y$ und z abgeleitet werden, nicht jedoch als Differenz von x und $y - z$.

4.5 Top down Parsing

Für praktische Zwecke soll ein Parser nicht bloß feststellen, ob ein Wort zur Sprache gehört, er soll auch den zugehörigen Syntaxbaum erstellen. Im Falle eines RD-Parsers beginnt das Wachstum des Ableitungsbaumes und damit auch des Syntaxbaumes, in der Wurzel. Allgemein kommt für ein beliebiges Nonterminal A mit Grammatik-Regel

$$A ::= \alpha_1 | \ldots | \alpha_n$$

und ein konkretes Wort w im Input eine bestimmte Alternative α_i zum Zuge. Im Ableitungsbaum wächst ein α_i entsprechender Knoten.

Im Syntaxbaum hat dieser Knoten zu jedem Nonterminal in α_i einen Sohn. Insgesamt wächst der Syntaxbaum von der Wurzel zu den Söhnen und schließlich zu den Blättern. Die entspricht der Aufrufhierarchie der *parse*-Methoden, denn beginnend mit `parse_S` ruft ein beliebiges `parse_A` die parse-Routinen auf, die den Nonterminalen in entsprechen. Da man in der Informatik Bäume immer mit der Wurzel nach oben darstellt, sagt man, dass der Baum top-down (engl. für: *von der Spitze nach unten*) wächst.

Abbildung 4.5.1 zeigt die Situation eines RD-Parsers für unsere WHILE-Sprache mit einem konkreten Programmtext als Eingabe zu einem Zeitpunkt als von der Eingabe schon der Text „`begin if x > y then x := x-y`“ gelesen worden ist. Die Blätter des bereits fertiggestellten Teils des Ableitungsbaumes ergeben von links nach rechts gelesen genau die Folge der Token, die bereits verarbeitet wurden. Als nächstes

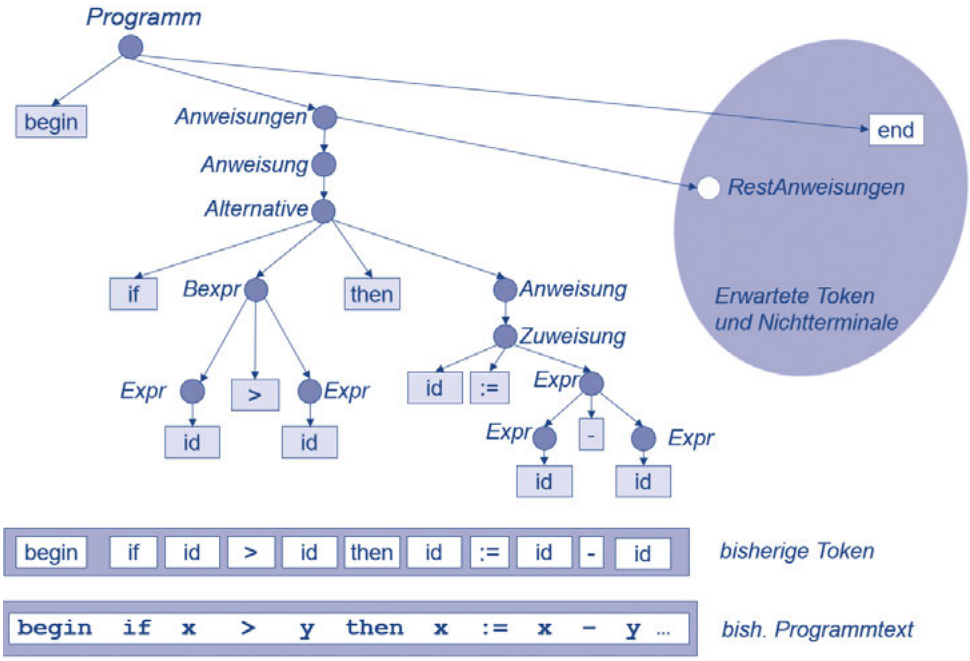


Abb. 4.5.1: Top-Down-Parsing

werden *RestAnweisungen* erwartet, also ein Semikolon gefolgt von Anweisungen und schließlich ein ***end***. Recursive descent ist demnach eine top-down-Methode. Die Knoten des Abstrakten Syntaxbaumes werden in *preorder* Reihenfolge, beginnend mit der Wurzel, dann in den linken Teilbaum absteigend, aufgebaut.

4.5.1 Nachteile der Recursive-Descent-Compiler

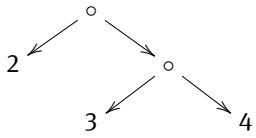
Zwar ist Top-Down-Parsing in Form der Recursive-Descent Technik sehr einfach zu verstehen und ebenso einfach zu implementieren, allerdings können die nötigen Grammatik-Modifikationen, mit der die Grammatik in eine LL(1)-Form gebracht wird, zu Syntaxbäumen führen, die mit der intendierten Semantik nur schwer in Einklang zu bringen ist. Wir demonstrieren dies am Beispiel der Grammatik

$$\begin{aligned} Expr &::= Expr - Expr \\ &\mid \ \textbf{\textit{num}} \end{aligned}$$

Entfernen wir die Linksrekursion, so erhalten wir

$$\begin{aligned} Expr &::= \textbf{\textit{num}}\ RestExpr \\ RestExpr &::= -\textbf{\textit{num}}\ RestExpr \\ &\mid \ \varepsilon \end{aligned}$$

Diese Grammatik ist LL(1) lässt sich also sofort in einen RD-Parser umsetzen. Parsen wir aber einen Ausdruck wie „2 – 3 – 4“ so spiegelt die Aufrufhierarchie einen Syntaxbaum der folgenden Form wider, der offensichtlich für die Auswertung des Ausdrucks ungeeignet ist:



4.6 Shift-Reduce Parser

Während Recursive-Descent Parser leicht und elegant in jeder Programmiersprache zu erstellen sind, bedient man sich für die professionelle Compilerentwicklung meist einer anderen Methode – des sogenannten Shift-Reduce Parsens. Dass Shift-Reduce Parser schwieriger zu erstellen sind als Recursive-Descent Parser fällt deswegen nicht ins Gewicht, weil man dazu mächtige Werkzeuge (engl.: *tools*) benutzen kann, die die Erstellung eines solchen Parsers automatisieren. Da solche Tools aus einer Syntaxbeschreibung automatisch den Syntaxanalyseteil eines Compilers erzeugen, heißen sie auch *compiler-compiler*. Der bekannteste heißt *yacc* (yet another compiler compiler). Die kostenfreie Variante, die unter der GNU Public License auf allen Plattformen erhältlich ist, heißt *bison*.

Wozu aber komplizierte Werkzeuge, wenn es einfachere Programmiertechniken gibt? Nun, wollte man lediglich die Syntax prüfen, also parsen, so gäbe es in der Tat keinen Grund. In der Praxis will man aber während des Parsens sofort – und möglichst automatisch – den Syntaxbaum erstellen. Da erweist es sich als Nachteil, wenn man die ursprüngliche Grammatik umschreiben muss, um sie in LL(1)-Form zu bringen. Dabei werden immer wieder aus links-geklammerten Ausdrücken rechtsgeklammerte. Ein Grund ist das Bemühen, Linksrekursionen zu vermeiden. Gleich mehrere Beispiele finden wir in der Grammatik der While-Sprache. Auch um die Präzedenzen der Operatoren korrekt zu parsen hatten wir die Grammatik umformen müssen und uns bei dieser Gelegenheit mehrere Linksrekursionen eingehandelt.

Bei der Abarbeitung der Syntaxbäume ist die falsche Repräsentation wieder auszugleichen. Shift-Reduce Parser haben im Gegensatz zu RD-Parsern keine Probleme mit der Linksrekursion. Für einen Shift-Reduce Parser dürfte man direkt die „richtigen“ Regeln verwenden, die der Linksklammerung arithmetischer Operatoren entsprechen.

$$
\begin{array}{rcl}
\mathit{Expr} & ::= & \mathit{Expr} + \mathit{Term} \\
& | & \mathit{Term} \\
\mathit{Term} & ::= & \mathit{Term} * \mathit{Factor} \\
& | & \mathit{Factor} \\
\mathit{Factor} & ::= & (\,\mathit{Expr}\,) \mid \textbf{\textit{num}} \mid \textbf{\textit{id}}
\end{array}
$$

4.6.1 Die Arbeitsweise von Shift-Reduce-Parsern

Shift-Reduce Parser versuchen immer, die rechte Seite einer Regel im Input zu erkennen und diese dann zur linken Seite zu reduzieren. Im Allgemeinen hat also ein Shift-Reduce-Parser bereits eine Satzform im Input erkannt und sieht als nächstes ein Token, worauf dann der Rest des Eingabewortes folgt. Wir stellen diesen Zustand des Parsers durch die Notation

$$\alpha \bullet au$$

dar. Dabei ist α die bereits erkannte Satzform, $\bullet$ die Position des Parsers, a das nächste Token und u der noch nicht gelesene Rest des Inputs. Der Parser kann jetzt zwei mögliche Aktionen vollführen:

Shift: das nächste Zeichen im Input lesen und in den Zustand $\alpha a \bullet u$ übergehen, oder
Reduce: ein Endstück α_2 von α als rechte Seite einer Regel $A \to \alpha_2$ erkennen, und dieses durch A ersetzen:

$$\alpha_1 \alpha_2 \bullet au \rightarrow \alpha_1 A \bullet au.$$

Ein Reduktionsschritt ist also die Umkehrung eines Ableitungsschrittes. Der gesamte Erkennungsprozess eines Wortes beginnt mit dem Zustand $\bullet w$ und endet mit $S\bullet$, wobei S das Startsymbol der Grammatik ist.

Wir zeigen in der folgenden Sequenz die einzelnen Schritte im Erkennungsprozess für das Wort $x * (y + 1)$, also für die Tokenfolge **id * (id + num)**. Die Nonterminale *Expr*, *Term* und *Faktor* kürzen wir mit E, T und F ab.

$$
\begin{array}{lcll}
\bullet * (id + num) & \twoheadrightarrow & id \bullet *(id + num) & \text{shift} \\
& \twoheadrightarrow & F \bullet *(id + num) & \text{Red mit } F \to id \\
& \twoheadrightarrow & T \bullet *(id + num) & \text{Red mit } T \to F \\
& \twoheadrightarrow & T * (id \bullet + num) & \text{3 Shifts} \\
& \twoheadrightarrow & T * (F \bullet + num) & \text{Red mit } F \to id \\
& \twoheadrightarrow & T * (T \bullet + num) & \text{Red mit } T \to F \\
& \twoheadrightarrow & T * (E \bullet + num) & \text{Red mit } E \to T \\
& \twoheadrightarrow & T * (E + num \bullet) & \text{2 Shifts}
\end{array}
$$

$\rightarrow\!\!\!\!\rightarrow$	$T * (E + F \bullet)$	Red mit $F \rightarrow id$
$\rightarrow\!\!\!\!\rightarrow$	$T * (E + T \bullet)$	Red mit $T \rightarrow F$
$\rightarrow\!\!\!\!\rightarrow$	$T * (E \bullet)$	Red mit $E \rightarrow E + T$
$\rightarrow\!\!\!\!\rightarrow$	$T * (E) \bullet$	Shift
$\rightarrow\!\!\!\!\rightarrow$	$T * F \bullet$	Red mit $F \rightarrow (E)$
$\rightarrow\!\!\!\!\rightarrow$	$T \bullet$	Red mit $T \rightarrow T * F$
	$E \bullet$	Red mit $E \rightarrow F$

Verfolgt man die Aktionen rückwärts, so erhält man in der Tat eine Rechtsableitung des Eingabewortes. Ein Shift-Reduce-Parser vollführt immer eine Rechtsableitung rückwärts. Damit ist insbesondere klar, dass für jede Grammatik ein nichtdeterministischer Shift-Reduce Parser existiert. Für praktische Zwecke ist aber immer noch der Nichtdeterminismus zu beseitigen.

Die Aktionen eines *Shift-Reduce Parsers* sind in vielerlei Hinsicht invers zu denen eines RD-Parsers. Dazu gehört auch, dass der Ableitungsbaum bzw. Syntaxbaum nicht top-down, sondern bottom-up generiert wird. Das soll bedeuten, dass das Wachstum mit den Blättern, den Terminalen, beginnt. Blätter wachsen zu Bäumen zusammen, diese schließlich zu einem einzigen Ableitungsbaum. Zu einem beliebigen Zeitpunkt hat ein SR-Parser bereits eine Satzform erkannt. Jedes Zeichen dieser Satzform ist bereits die Wurzel eines eigenen kleinen Baumes. Dieser besteht nur aus einem Blatt, falls ein Terminalzeichen ist, bzw. aus einem Knoten mit Unterbäumen, falls er durch Reduktion mit einer Produktion entstanden ist. Bei jedem shift-Schritt kommt ein aus einem einzelnen Blatt bestehendes Bäumchen hinzu, bei jedem Reduktionsschritt mit Produktion werden bestehende Bäume unter einer gemeinsamen Wurzel zusammengefasst.

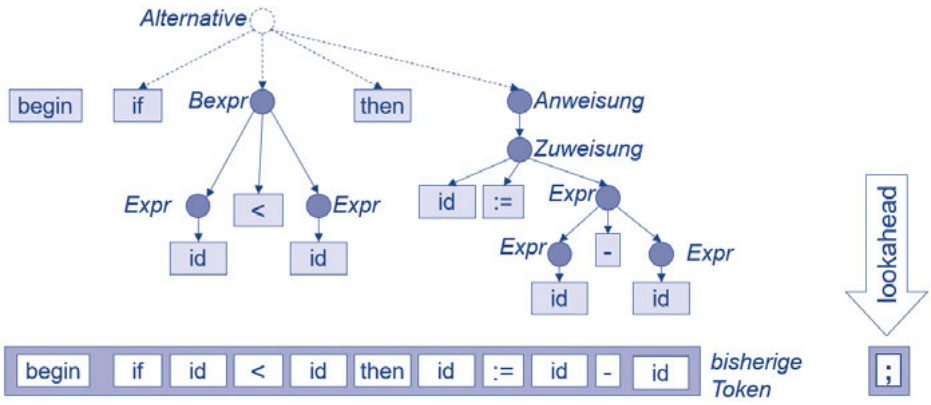


Abb. 4.6.1: Bottom-Up-Parsing

Abbildung 4.6.1 zeigt einen Schnappschuss eines Bottom-Up Parsers für unsere While-Sprache. Die bereits erkannte Satzform ist „*begin if Bexpr then Anweisung*“. Dem entsprechen 5 Ableitungsbäume – drei die jeweils nur aus einem Blatt bestehenden (***begin***, ***if***, ***then***) und die beiden Bäume mit Wurzeln *Bexpr* bzw. *Anweisung*. Wäre das nächste Zeichen des Inputs ein ***else***, so würde dieses als weiterer trivialer Baum hinzukommen. Da das nächste Zeichen „**;**“ aber zu einer Reduktion mit der Regel

$$\mathit{Alternative} :: \textbf{\textit{if}}\ \mathit{Bexpr}\ \textbf{\textit{then}}\ \mathit{Anweisung}$$

führt, werden als nächstes, wie in der Figur angedeutet, die letzten vier Bäume durch einen *Alternative*-Knoten zusammengefasst. Es bleiben zwei Bäume übrig. Ihre Wurzeln, von links nach rechts gelesen, repräsentieren die Satzform „***begin*** *Alternative*“.

4.6.2 Konflikte

Solange noch ein Zeichen im Input vorhanden ist, könnte der SR-Parser shiften. Allerdings sieht man leicht, dass dies in Sackgassen führen könnte. Würde man zum Beispiel in der Situation $T * (E + \textbf{\textit{num}} \bullet)$ shiften, was $T * (E + \textbf{\textit{num}}) \bullet$ produziert, so könnte man nie mehr reduzieren, denn die Expr-Grammatik hat keine Regel, deren rechte Seite ein Endstück von $T * (E + \textbf{\textit{num}})$ ist.

Jede Situation, in der der Parser zwei verschiedene Aktionen vollführen könnte, nennt man einen *Konflikt*. Kann er sowohl ein *Shift*, als auch ein *Reduce* vollführen, wie in der gerade besprochenen Situation, so nennt man dies einen *shift-reduce-Konflikt*. Ein *reduce-reduce-Konflikt* liegt vor, wenn zwei verschiedene Reduktionen möglich wären. Ein Beispiel ist die Situation $T * (E + T \bullet)$. Hier sind zwei Reduktionen möglich – mit Regel $E \to T$ zu $T * (E + E \bullet)$, oder mit Regel $E \to E + T$ zu $T * (E \bullet)$. Offensichtlich kann nur letztere zum Ziel führen.

4.6.3 Ein nichtdeterministischer Automat mit Stack

Eine Grammatik heißt LR(1), falls es möglich ist, nur anhand des nächsten Zeichens im Input alle Konflikte eindeutig auflösen zu können. Zu diesem Zweck setzen wir die bisher noch ungenutzte Möglichkeit ein, die Aktionen des Stackautomaten von Zuständen Q abhängig zu machen. Anhand des gegenwärtigen Zustands $q \in Q$ und anhand des nächsten Zeichens im Input soll man entscheiden können, ob ein *shift* oder ein *reduce* angebracht ist, und im letzteren Falle auch, mit welcher Regel reduziert werden soll.

Als Zustandsmenge wählt man sogenannte *items*. Das sind Produktionen der Grammatik, deren rechte Seite eine Markierung trägt. Diese soll andeuten, an welcher Position in der rechten Seite der Regel der Parser sich gerade befindet. Als Beispiel wählen wir im Folgenden einen Ausschnitt aus unserer *Expr*-Grammatik. Wir haben ein Startsymbol S und ein end-of-file-token **eof** beigefügt und für spätere Zwecke die

Produktionen durchnummeriert:

$$
\begin{array}{lrcl}
(0) & S & ::= & T\ \textbf{\textit{eof}} \\
(1) & T & ::= & T\ \texttt{*}\ F \\
(2) & & | & F \\
(3) & F & ::= & \texttt{(}\ T\ \texttt{)} \\
(4) & & | & \textbf{\textit{id}}
\end{array}
$$

Aus diesen Produktionen gewinnen wir die folgenden items:

$$
\begin{array}{llll}
S \to \bullet\, T\ \textbf{\textit{eof}} & S \to T \bullet\ \textbf{\textit{eof}} & S \to T\ \textbf{\textit{eof}} \bullet & \\
T \to \bullet\, T\ \texttt{*}\ F & T \to T \bullet\ \texttt{*}F & T \to T\ \texttt{*} \bullet F & T \to T\ \texttt{*}\ F \bullet \\
T \to \bullet\, F & T \to F \bullet & & \\
F \to \bullet\, \texttt{(}\ T\ \texttt{)} & F \to \texttt{(} \bullet\, T\ \texttt{)} & F \to \texttt{(}\ T \bullet\ \texttt{)} & F \to \texttt{(}\ T\ \texttt{)} \bullet \\
F \to \bullet\, \textbf{\textit{id}} & F \to \textbf{\textit{id}} \bullet & &
\end{array}
$$

Ein SR-Parser im Zustand $F \to \texttt{(}\ T \bullet\ \texttt{)}$, beispielsweise, ist im Begriff unter Benutzung der Regel (3) $F \to \texttt{(}\ T\ \texttt{)}$ ein F zu erkennen. Den ersten Teil der rechten Seite, „**(** T “, hat er bereits gesehen. Er erwartet jetzt noch ein „**)**“, bevor er zu F reduzieren kann.

Die Menge aller items bildet die Zustandsmenge Q eines nichtdeterministischen Automaten über dem Alphabet Γ, das aus allen Terminalen und Nonterminalen der Grammatik besteht. Der Automat hat Transitionen

$$A \to \alpha \bullet \kappa y \quad \to \quad A \to \alpha\kappa \bullet y$$

für jedes Inputsymbol $\kappa \in \Gamma$, und eine ε-Transition

$$A \to \alpha \bullet B\, y \quad \to \quad B \to \bullet\, \beta$$

für jede Regel $B \to \beta$ der Grammatik. In unserem Beispiel haben wir also u.a. die folgenden Transitionen:

$$
\begin{aligned}
\delta(F \to (\bullet\, T), T) &= \{F \to (T \bullet)\} \\
\delta(F \to (\bullet\, T), \varepsilon) &= \{T \to \bullet\, T\ \texttt{*}\ F,\ T \to \bullet\, F\}
\end{aligned}
$$

Auf diese Weise entsteht der Automat in Fig. 4.6.2. Darin haben wir die ε-Transitionen durch gestrichelte Linien kenntlich gemacht.

Items der Form $A \to \alpha \bullet$ mit dem Punkt am Ende heißen *Reduce-Items*. In einem solchen Zustand *kann* der Stackautomat mit der Regel reduzieren. Eine solche Reduktion entspricht gleichzeitig einer Transition mithilfe des Nonterminals A.

Wir verfolgen dies anhand des Inputs „**id * (id) eof**“. Die Aufgabe des nichtdeterministischen Automaten ist es, mit dem gegebenen Input vom Anfangszustand

$$S \to \bullet\, T\ \textbf{\textit{eof}}$$

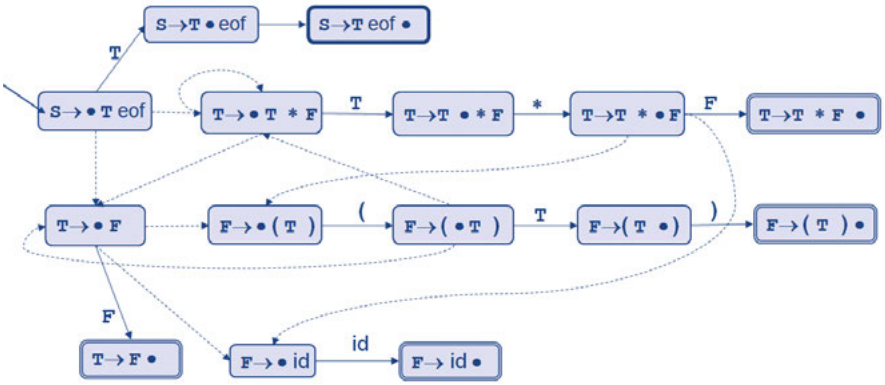


Abb. 4.6.2: Nondeterministischer Automat der Items

in den Zustand

$$S \to T\ \textbf{\textit{eof}}\ \bullet$$

zu gelangen.

Zuerst ist also ein Nonterminal T zu erkennen. Dazu wird der gegenwärtige Zustand $S \to \bullet\ T\ \textbf{eof}$ auf dem Stack abgelegt und mit einer ε-Transition zu einer der Produktionen für T gesprungen. Wir wählen $T \to \bullet\ T * F$.

Erneut müssen wir ein T erkennen, wir legen also auch $T \to \bullet\ T * F$ auf den Stack und springen diesmal zu $T \to \bullet\ F$.

Jetzt ist ein Nonterminal zu erkennen, also kommt auch $T \to \bullet\ F$ auf den Stack und wir nehmen die ε-Transition zu $F \to \bullet\ \textbf{\textit{id}}$.

Dies ist ein Shift-Zustand, von dem wir mit Hilfe des ersten Input-Zeichens ***id*** in den Zustand $F \to \textbf{\textit{id}}\ \bullet$ gelangen. Wir können nun reduzieren, was gleichbedeutend damit ist, ein F im Input erkannt zu haben.

Auf dem Stack liegen mittlerweile die Zustände $T \to \bullet\ F$, $T \to \bullet\ T * F$, $S \to \bullet\ T$. Sie entsprechen den noch nicht gelösten Aufgaben. Da wir aber gerade ein F erkannt haben, wurde zuletzt auf dem Stack abgelegte Aufgabe $T \to \bullet\ F$ gelöst. Wir entfernen dieses oberste Stack-Element, führen die F-Transition im Automaten aus und gelangen in den Zustand $T \to F\ \bullet$.

Dieser *reduce*-Zustand besagt wiederum, dass wir gerade ein T erkannt haben, womit wir auch den ersten Schritt der mittlerweile zuoberst auf dem Stack liegenden Aufgabe $T \to \bullet\ T * F$ gelöst haben. Wir entfernen diesen Zustand vom Stack und gehen in den Zustand $T \to T\ \bullet\ * F$ über. Dieser *shift*-Zustand veranlasst uns zum Einlesen eines „*“ vom Input, was auch gelingt und uns nach $T \to T * \bullet\ F$ bringt. Wir legen diesen Zustand auf den Stack und springen in eine Produktion für F, diesmal wählen wir natürlich $F \to \bullet\ T$.

Nach einigen weiteren Schritten haben wir das Eingabewort vollständig erkannt und sind dabei im Zielzustand $S \to T\ eof\ \bullet$ angelangt.

Die Regeln können wir allgemein also je nach erreichtem Zustand so zusammenfassen. Dabei seien A, B nonterminal, t terminal und αund β beliebige Satzformen:

$A \to \alpha \bullet ty$ Dies ist ein Shift-Zustand Shift-Zustand. Im Input wird ein t erwartet und eingelesen. Der neue Zustand ist dann $A \to \alpha t \bullet y$.

$A \to \alpha \bullet By$ Dieses Item wir auf dem Stack abgelegt. Mit einer ε-Transition springen wir in einen Zustand der Form $B \to \bullet\beta$. Wenn diese erfolgreich abgearbeitet ist, haben wir das B erfolgreich erkannt.

$B \to \beta \bullet$ Ein *Reduce*-Zustand $B \to \beta \bullet$ entspricht dem Erkennen des Nonterminals B. Oben auf dem Stack sollte ein Zustand der Form $A \to \alpha \bullet By$ liegen. Dieser wird entfernt der neue Zustand ist $A \to \alpha B \bullet y$.

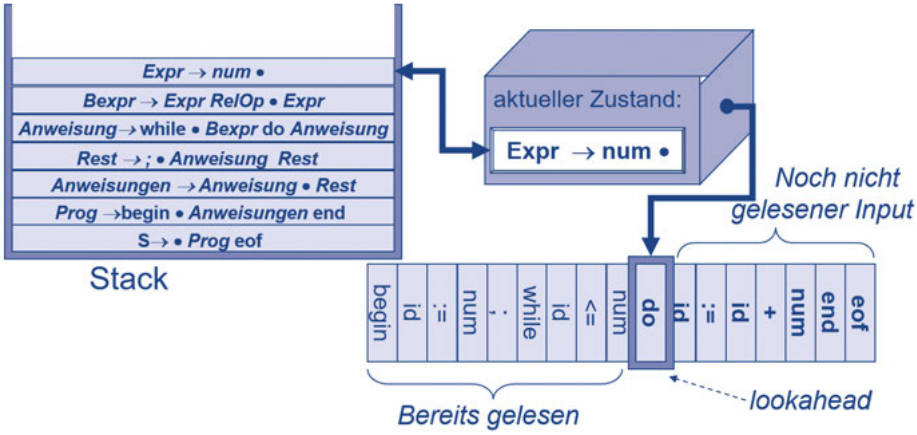


Abb. 4.6.3: Stackautomat beim nichtdeterministischen Shift-Reduce-Parsen

Die Abbildung 4.6.3 zeigt einen Shift-Reduce Parser, implementiert durch einen Stackautomaten, beim Lesen eines *while*-Programms. Der aktuelle Zustand

$$Expr \to \textbf{\textit{num}} \bullet$$

führt bei lookahead **do** zu einem *reduce*, was gleichbedeutend damit ist, dass ein *Expr* im Input erkannt wurde. Der oberste Stackzustand wird entfernt, der darunterliegende durch

$$Bexpr \to Expr\ Relop\ Expr \bullet$$

ersetzt. Eine erneute Reduktion bei unverändertem lookahead führt zu

$$Anweisung \to \textbf{\textit{while}}\ Bexpr \bullet \textbf{\textit{do}} Anweisung$$

auf dem Stack. Als nächstes wird ***do*** geshiftet etc., bis am Ende durch ein Shift von ***eof*** der Stack entleert ist.

4.6.4 Übergang zum deterministischen Automaten

Unser nächstes Ziel ist, den Nichtdeterminismus aus dem Automaten zu entfernen. Dazu führen wir die Potenzmengenkonstruktion von Abschnitt 2.7 in Kapitel 2 durch, wobei ein Zustand des neuen Automaten jeweils eine Teilmenge der Zustände des ursprünglichen Automaten repräsentiert, und zwar jeweils diejenige Teilmenge, die aus allen Zuständen besteht, welche durch das gleiche Wort erreichbar sind.

In unserem Falle sind z.B.

$$
\begin{aligned}
D &:= \{ S \to \bullet T\,\textbf{\textit{eof}}\,, T \to \bullet F, F \to \bullet id\,, F \to \bullet\textbf{(}\mathrm{T}\textbf{)}, T \to \bullet T * F \} \\
B &:= \{ S \to T \bullet \textbf{\textit{eof}}\,, \mathrm{T} \to T \bullet *F \} \\
G &:= \{ T \to T * \bullet F, F \to \bullet\,\textbf{\textit{id}}\,, F \to \bullet\,\textbf{(}T\textbf{)} \}
\end{aligned}
$$

Zustände des deterministischen Automaten und es gilt z.B.:

$$\delta(A, T) = B \text{ sowie } \delta(B, *) = G.$$

Man kann die Zustände auch so erzeugen, dass man mit dem Start-Item $S \to \bullet\, T\,\textbf{eof}$ beginnt und alle durch ε erreichbaren Items hinzufügt. Man erhält dann gerade die Itemmenge A, welche somit den Anfangszustand des deterministischen Automaten repräsentiert.

Weiter erhält man z.B. $\delta(A, F) = \{T \to F\bullet\}$, denn $T \to F\bullet$ ist das einzige Item in A, für das eine F-Transition möglich ist.

Für die öffnende Klammer ´**(**´ erhält man zunächst $F \to \textbf{(}\bullet\, T\textbf{)}$ als Element von $\delta(A, \textbf{(})$. Da aber von diesem Zustand ε-Transitionen ausgehen, muss man noch $T \to \bullet\, T * F$ und $T \to \bullet F$ hinzunehmen. Das letzte Item hat erneut ε-Transitionen nach $F \to \bullet\,\textbf{\textit{id}}$ und $F \to \bullet\,\textbf{(}T\textbf{)}$, so dass das Ergebnis $\delta(A, \textbf{(}) = D$ ist mit

$$D := \{F \to (\bullet T),\ T \to \bullet T * F,\ T \to \bullet F,\ F \to \bullet\textbf{\textit{id}}, \mathrm{F} \to \bullet\textbf{(}\,T\textbf{)}\}.$$

In der Compilerliteratur nennt man die Tabelle des so entstehenden deterministischen Automaten auch *Goto-Tabelle*. Der vollständige Automat für unsere kleine Grammatik ist in der folgenden Abbildung dargestellt. Reduce-Zustände sind besonders hervorgehoben. Beispielsweise sind C bzw. J *reduce*-Zustände für $T \to F\bullet$ bzw. $T \to T * F\bullet$, und E bzw. K sind *reduce*-Zustände für $F \to \textbf{\textit{id}}\bullet$ bzw. $F \to \textbf{(}\,T\,\textbf{)}\bullet$ und ***OK*** ist *reduce*-Zustand für $S \to T\,\textbf{eof}\,\bullet$.

Kommt also der Parser in den Zustand J, dann hat er gerade die Satzform $T * F$ im Input gesehen. Er kann jetzt

- die 3 obersten Elemente des Stacks entfernen
- mit dem dann obersten Stackelement X die Transition $\delta(X, T)$ ausführen und
- das Ergebnis auf den Stack legen.

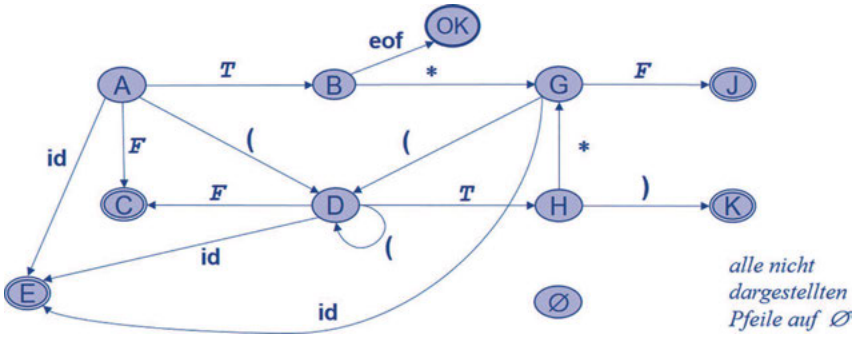


Abb. 4.6.4: Deterministischer Goto-Automat

Ein Zustand des deterministischen Automaten kann sowohl shift-items als auch reduce-items beinhalten. Ein Konflikt kann dennoch meist vermieden werden, wenn man das nächste Zeichen des Inputs, das *lookahead*, berücksichtigt. Ein *shift-reduce Konflikt* liegt nur vor, wenn ein Zustand sowohl ein *shift*-item $A \to \alpha \bullet ty$ als auch ein *reduce*-item $B \to \beta \bullet$ enthält und zusätzlich gilt, dass t$\in Follow(B)$. Man sagt, der Zustand sei ein shift-reduce-Konflikt für t.

Ein *reduce-reduce-Konflikt* für t ist ein Zustand mit zwei reduce-items $A \to \alpha\bullet$ und $B \to \beta \bullet$ so dass $t \in Follow(A) \cap Follow(B)$ gilt.

4.6.5 Goto-Tabelle

Wenn wir unter Berücksichtigung des lookaheads keinen Konflikt mehr haben, dann können wir den deterministischen Parser in einer Parsertabelle darstellen. Die Zeilen entsprechen den Zuständen, die Spalten den Terminalen und Nichtterminalen. In Zeile X und Spalte κ steht dann jeweils

$\delta(X, \kappa)$, falls X ein Item der Form $A \to \alpha \bullet \kappa y$ enthält.

$reduce\,(A \to \alpha)$, falls $(A \to \alpha\bullet) \in X$ und $\kappa \in Follow(A)$.

Alle Informationen, die der Parser benötigt, sind in dieser Tabelle enthalten. Jeder Eintrag der Form $\delta(X, \kappa) = Y$ bedeutet, dass ein Terminal oder Nichtterminal κ im Input erkannt wurde. Statt dieses selber auf den Stack zu bringen, kann man genauso gut den neuen Zustand Y ablegen.

Ein Eintrag $r(i)$ bedeutet: Mit Regel (i) (aus der Termgrammatik von Seite 132) reduzieren. Ist Regel (i) also $A \to \alpha$, so besagt $r(i)$:

- entferne die obersten $|\alpha|$-vielen Elemente vom Stack,
- ist X jetzt das oberste Stack-Element, dann lege $\delta(X, A)$ auf den Stack.

	id	**(**	**)**	*****	T	F	**eof**
A	E	D			B	C	
B				G			OK
C			r(2)	r(2)			r(2)
D	E	D			H	C	
E			r(4)	r(4)			r(4)
G	E	D				J	
H			K	G			
J			r(1)	r(1)			r(1)
K			r(3)	r(3)			r(3)

Tab. 4.2: Parsertabelle für die Termgrammatik

Die Entwicklung des Parserstacks bei Eingabe des Wortes „**(id * id)eof**“ zeigt die folgende Figur. Hervorgehoben sind die Zustände X, aus denen ein shift $\delta(X, \kappa)$ – dargestellt durch die Pfeile – ausgeführt wurde. Dabei ist κ entweder ein Terminalzeichen aus dem Input, oder ein Nonterminal der Grammatik – als Folge einer zugehörigen Reduktion. Zum Abschluss könnte noch mit Regel $r(0)$ reduziert werden, so dass der Stack wieder – bis auf A entleert würde.

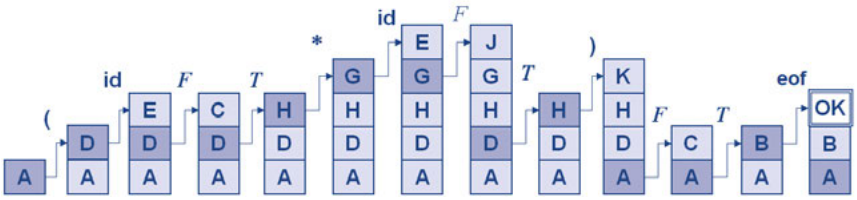


Abb. 4.6.5: Stack beim SR-Parsen von **(id * id) eof**

4.6.6 Präzedenz

Für recursive-descent Parser war es notwendig, die Grammatik umzuschreiben, um die gewünschten Präzedenzen der arithmetischen Operatoren zu erreichen. Im obigen Beispiel haben wir dies auch für das Beispiel unseres Shift-Reduce Parsers gemacht. Es stellt sich heraus, dass diese Umschreibung nicht notwendig ist.

Auch dies macht SR-Parser besonders attraktiv. Wir beginnen also gleich mit der gewünschten Grammatik, so wie sie ursprünglich für die While-Sprache spezifiziert wurde. Wir beschränken uns auf einen vereinfachten *Expr*-Teil der Grammatik und kürzen *Expr* wieder mit *E* ab:

$$
\begin{array}{lll}
(0) & E & ::= \quad E - E \\
(1) & & \mid \quad E * E \\
(2) & & \mid \quad \mathbf{id}
\end{array}
$$

Wenn wir jetzt den nichtdeterministischen Automaten der *items* konstruieren, dann daraus den deterministischen Automaten, dessen Zustände aus *item*-Mengen bestehen, stoßen wir u.a. auf den folgenden Zustand:

$$A = \{E \to E - E\bullet, E \to E \bullet -E, E \to E \bullet *E\}$$

Da sowohl „–“ als auch „*“ in $Follow(E)$ liegen, hat A je einen *shift-reduce* Konflikt für das *lookahead* „–“ und für das *lookahead* „*“. Tritt aber der Konfliktfall mit „–“ als lookahead auf, dann wurde bereits ein Ausdruck der Form $E - E$ im Input erkannt, und es kommt das nächste „–“. Offensichtlich führt daher ein *reduce* zu einer Linksklammerung einer Folge von Subtraktionen und ist somit angebracht.

Für das lookahead „*“ ist dagegen ein *shift* angezeigt. Im Konfliktfall haben wir eine Differenz erkannt und sehen ein „*“ vor uns. Das zweite E ist also Teil eines Produktes. Dieses muss aufgrund der Präzedenz von „*“ über „–“ zuerst ausgewertet und danach die Differenz gebildet werden. Somit wird dieser Konflikt durch ein *shift* korrekt gelöst. Folglich muss in der Parsertabelle in Zeile A und Spalte „–“ ein *reduce* stehen und in Spalte „*“ ein *shift*..

Analoge shift-reduce Konflikte bei lookahead „–“ und „*“ enthält der Zustand

$$B = \{E \to E * E\bullet,\ E \to E \bullet -E,\ E \to E \bullet *E\}.$$

In diesem Falle sind sowohl bei lookahead „–“ als auch bei lookahead „*“ *reduce*-Aktionen angebracht. Im ersten Fall, weil „*“ höhere Präzedenz als „–“ hat, im zweiten Fall, weil „*“ linksgeklammert werden soll.

Präzedenzen von Operatoren kann man so allgemein durch geeignete Auflösung von Shift-Reduce-Konflikten festlegen. Compiler-compiler wie *yacc* oder *bison*, die aus einer Grammatik automatisch eine Parsertabelle erzeugen, erlauben, die gewünschte *Präzedenz* und *Assoziativität* von Operatoren in der Grammatik anzugeben. Letztere kann die Werte *left* oder *right* haben, was bedeutet, dass der betreffende Operator links bzw. rechts geklammert sein soll.

4.6.7 LR(1) und LALR(1)

Die besprochene Methode des Parsens nennt man auch *LR-Parsen*. Der erste Buchstabe „L“ bedeutet, dass der Input von Links nach rechts gelesen wird, und das „R“, dass dabei eine Rechtsableitung nachvollzogen wird. Entsprechend handelte es sich bei den früher besprochenen recursive-descent Parsern um *LL-Parser*.

Schließlich unterscheidet man die Parsermethoden noch danach, wieviele Zeichen im Input als *lookahead* berücksichtigt werden. Im Falle der RD-Parser hatten wir stets ein *lookahead* berücksichtigt, man spricht also von $LL(1)$-Parsern. Im Falle des

Shift-Reduce Parsers hatten wir bei der Konstruktion der items die lookaheads ignoriert, erst bei der Konstruktion der Parsertabelle wurden sie einbezogen.

Es handelt sich hierbei um einen Kompromiss zwischen einem $LR(0)$ und einem $LR(1)$-Parser, den man auch als *SLR*-Parser (*Simple LR*) bezeichnet. Ein richtiger $LR(1)$-Parser bezieht schon bei der Konstruktion der *items* die möglichen lookaheads ein. Ein *LR(1)*-Item ist dann ein Paar

$$[A \to \alpha \bullet \beta, t], \text{ mit } t \in Follow(A).$$

Beginnend mit $[S \to \bullet A, \textbf{\textit{eof}}]$ konstruieren wir den deterministischen Automaten mit den X-Transitionen

$$[A \to \alpha \bullet Xy, t] \dashrightarrow [A \to \alpha \bullet uy, t]$$

für X terminal oder nonterminal und den ε-Transitionen

$$[A \to \alpha \bullet By, a] \dashrightarrow [B \to \bullet\beta, b],$$

falls $b \in First(ya)$.

Nach dem Übergang zum deterministischen Automaten sind in der Regel mehr Zustände entstanden, als bei der SLR-Methode. Daher verschmilzt man in der Praxis am Ende zwei Zustände, wenn sie die gleichen LR(0)-Items, aber verschiedene lookaheads beinhalten. Den so entstehenden Parser nennt man $LALR(1)$-Parser.

Mit $LALR(1)$-Parsern kann man in der Praxis alle üblichen syntaktischen Konstrukte von Programmiersprachen erkennen. Daher erzeugen die meistgenutzten Parsergeneratoren, *yacc* (bzw. *bison*) $LALR(1)$-Parser.

4.7 Parsergeneratoren

Parsergeneratoren automatisieren die Konstruktion eines Parsers aus einer kontextfreien Grammatik. Sie implementieren die obigen Schritte der Konstruktion einer Parsertabelle aus einer Grammatik und deren Umsetzung in ein lauffähiges Programm. Eine *yacc*-(bzw. *bison*-) Datei besteht aus drei Teilen, die jeweils durch `%%` getrennt sind. Im ersten Teil werden die Token deklariert, im zweiten Teil folgen die Grammatik-Regeln und im dritten Teil kann eine C-Funktion angegeben werden, die durch einen Aufruf von `yyparse()` den Parser startet.

Um zu vergleichen, wie das soeben theoretisch besprochene mit *bison* nachvollzogen werden kann, bereiten wir eine Datei `expr.y` vor, von der wir hier nur einen kleinen Ausschnitt zeigen:

```
%token ID
%%
S : T
T : T '*' F
```

```
  | F
  ;
F : '(' T ')'
  | ID
  ;
%%
```

Wir können dabei Token als C-Konstanten deklarieren, wie im Beispiel: `%token ID`. Token, die nur aus einem Zeichen bestehen, können wir direkt durch Apostrophe markieren. Semikola „;“ schließen Regeln mit mehreren Alternativen ab.

Der Befehl bison `-v expr.y` erzeugt jetzt die Datei `expr_tab.c`, die u.a. die C-Funktion `yyparse()` definiert. Wegen der Option `-v` (`v` wie *verbose=wortreich*) entsteht zusätzlich eine Diagnosedatei `expr.output`. Diese stellt die Zustände als Menge von items, aber in kompakter Form (ohne die durch ε erreichbaren Nachfolgezustände) dar. Außerdem wird angegeben, bei welchen lookahead geshiftet und in welchen Zustand gegangen, bzw. mit der wievielten Regel reduziert werden soll. Wir zeigen hier nur die ersten drei Zustände aus `expr.output` und ermutigen den Leser selber mit dem frei verfügbaren `bison` zu experimentieren.

```
state 0

    ID shift, and go to state 1
    '(' shift, and go to state 2
    S go to state 9
    T go to state 3
    F go to state 4

state 1

    F -> ID . (rule 5)
    $default reduce using rule 5 (F)

state 2

    F -> '(' . T ')' (rule 4)
    ID shift, and go to state 1
    '(' shift, and go to state 2
    T go to state 5
    F go to state 4
```

4.7.1 lex/flex & yacc/bison

lex und *yacc* arbeiten als Team. *lex* produziert aus einer Beschreibungsdatei, z.B. `while.l` eine C-Datei `lex.yy.c`. Darin wird eine Funktion `yylex()` bereitgestellt. Analog produziert *yacc* aus der Definitionsdatei, etwa `while.y`, eine C-Datei `while_tab.c`. Wie der Name schon andeutet, beinhaltet diese die Parsertabelle und stellt sie in Form der Funktion `yyparse()` zur Verfügung. Zusätzliche C-Routinen, insbesondere die obligatorische `main()`-Funktion können aus externen Dateien eingebunden werden oder auch in dem dritten Abschnitt der *lex*- oder der *yacc*-Datei definiert werden.

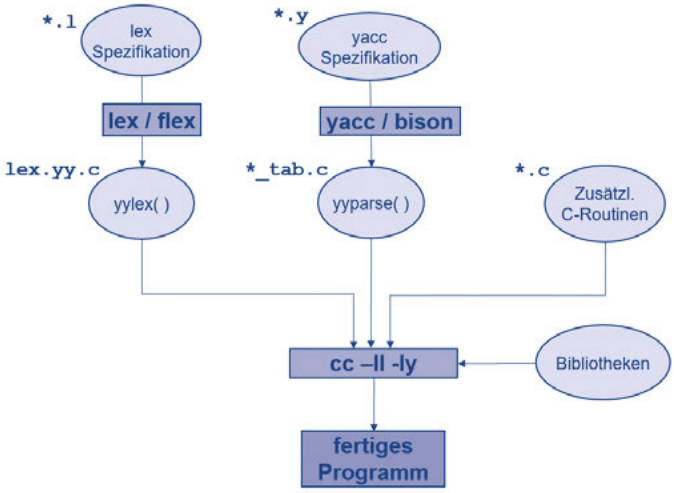


Abb. 4.7.1: Parserentwicklung mit lex und yacc

Für unsere While-Sprache haben wir also z.B. die lexikalische Definition in der Datei `while.l` und unsere grammatische Definition in `while.y`. Mit den Befehlen

```
lex while.l
bison while.y
```

erzeugen wir die C-Dateien `lex.yy.c` und `while_tab.c`. Wir rufen den C-Compiler `cc` auf, um `while_tab.c` zu übersetzen. Da es ein include `#include lex.yy.c` enthält, wird dieses Programm mitübersetzt. Aus den *lex*- und *yacc*-Bibliotheken werden durch die Optionen `-ll` und `-ly` noch Routinen hinzugebunden:

```
cc -o while y.tab.c -ll -ly
```

Das fertige Program heißt dann `while`, bzw. `while.exe`. Wird es gestartet, dann ruft seine Hauptfunktion `main()` die Parserfunktion `yyparse()` auf. Diese fordert jeweils von `yylex()` das nächste Token an. `yylex()` liest Zeichen aus dem Input, bis es ein komplettes Token gefunden hat und gibt dann dessen Nummer an `yyparse()` zurück. Kann `yyparse()` die gefundenen Token zu einem gültigen Wort der Grammatik gruppieren, so gibt es den Wert 0 an das Hauptprogramm zurück.

Die Phasen eines Compilers, *scannen, parsen, Codeerzeugung* laufen also nicht wirklich nacheinander ab, sondern ineinander verschränkt.

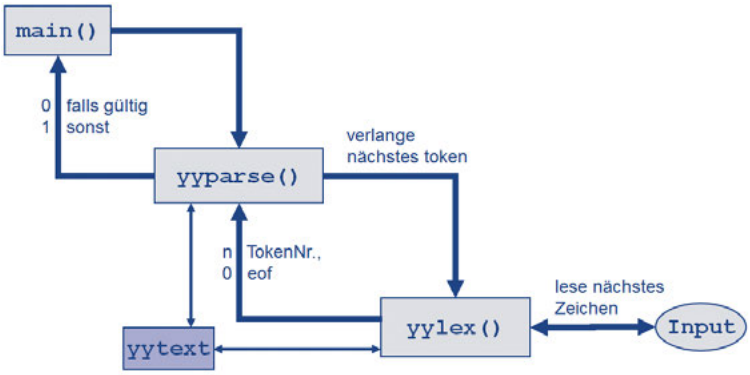


Abb. 4.7.2: Arbeitsweise von lex und yacc

Einige Token tragen eine Zusatzinformation, die für die Codeerzeugung wichtig ist. Für ein ***id*** ist insbesondere der Name der Variablen relevant, weil der Compiler jeder Variablen einen Speicherplatz zuordnen möchte, für ein ***num***-Token der konkrete Zahlenwert. Aus diesem Grund kann sich `yyparse()` zu jedem gelieferten Token auch noch dessen textuelle Repräsentation in der String-Variablen `yytext` besorgen. Analog gibt es eine zweite Variable `yylval` für den Zahlenwert von numerischen Token.

4.8 Grammatische Aktionen

In der Praxis soll ein Parser nicht nur prüfen, ob der Eingabestring ein syntaktisch korrektes Programm ist, er muss dieses auch übersetzen. Meist ist das Übersetzungsziel die Maschinensprache einer konkreten Prozessorarchitektur oder im Falle von Java die Sprache der Java Virtual Machine (JVM). Letztere ist ein Stackprozessor, insbesondere heißt das, dass für die Auswertung von Operationen die Operanden zuerst mit einem *push*- oder *load*-Befehl auf den Stack gelegt werden müssen.

Man muss unterscheiden, ob es sich um einen konstanten Operanden handelt (z.B. LOADC 42) oder um einen Operanden, dessen Wert zunächst aus dem Speicher

geladen werden muss (z.B. LOAD x). Danach wird der Operationsbefehl (etwa ADD, SUB, MUL, DIV, etc.) ausgeführt. Dabei werden die obersten Stackelemente mit der entsprechenden Operation verknüpft. Sie werden vom Stack entfernt und durch das Ergebnis der Operation ersetzt.

Beispielsweise entspricht der Ausdruck $2 * betrag * (zins + 1)$ folgenden Befehlen des Stackprozessors:

```
LOADC 2
LOAD betrag #
MUL
LOAD zins
LOADC 1
ADD MUL
```

In einem wirklichen Compiler müssten noch die Namen der Bezeichner durch ihre Speicheradressen ersetzt werden.

Sowohl beim RD-Parsen als auch beim SR-Parsen können sogenannte semantische Aktionen des Parsers in Form von C-Code in die Grammatik eingefügt werden. Diese semantischen Aktionen können u.a. dazu dienen, den übersetzten Programmtext auszugeben, einen Syntaxbaum aufzubauen, einen Wert aus den Werten der Unterausdrücke zu berechnen.

Die *yacc*-Eingabedatei erlaubt semantische Aktionen als C-Blöcke (in {}–Klammern eingeschlossene Anweisungsfolgen) einzubauen.

Die folgende *yacc*-Datei zeigt, wie man einen einfachen Übersetzer von arithmetischen Operationen in Befehle für einen Stackprozessor erhält. Dabei benutzen wir die im vorigen Abschnitt erklärten globalen Variablen `yytext` und `yylval`.

```
%token ID, NUM
%left PLUS, MINUS
%left TIMES, QUOT
%start expr
%%
expr : expr PLUS expr { printf(" add "); }

    | expr MINUS expr { printf(" sub "); }
    | expr TIMES expr { printf(" mult "); }
    | expr QUOT expr { printf(" div "); }
    | NUM { printf("LOADC %d ",yylval);}
    | ID { printf("LOAD %s ",yytext);}
    ;

%% #include "lex.yy.c"
```

```
int main(){

    printf("Bitte geben Sie einen Ausdruck ein :\n");
    yyparse(); }
```

Auch RD-Parser können mit semantischen Aktionen umgehen. Die Aktionen müssen meist aber in die ursprüngliche Grammatik eingebaut werden – bevor die Grammatik umgeschrieben wird, um Linksrekursionen und gemeinsame Links-Faktoren zu beseitigen. Bei der Umschreibung der Grammatik behandelt man diese semantischen Aktionen dann wie Terminale.

Wir zeigen im Folgenden, wie eine vereinfachte Version der obigen Grammatik umgeschrieben werden kann. Wir gehen aus von

$$\begin{aligned} Expr &::= Expr + Expr \,\{printf(" add ");\} \\ &\mid Expr \ * \ Expr \,\{printf(" mult ");\} \\ &\mid \textbf{\textit{num}}\{printf("LOADC\ \%d ", yylval);\} \end{aligned}$$

und führen zunächst Präzedenzen ein, um die Grammatik eindeutig zu machen:

$$\begin{aligned} Expr &::= Expr + Term \,\{printf(" add ");\} \\ &\mid Term \\ Term &::= Term * \textbf{\textit{num}} \,\{printf(" mult ");\} \\ &\mid \textbf{\textit{num}}\{printf("LOADC\ \%d ", yylval);\} \end{aligned}$$

Jetzt entfernen wir die Linksrekursionen. Dabei behandeln wir grammatische Aktionen wie Terminale:

$$\begin{aligned} Expr &::= Term\, ERest \\ ERest &::= \varepsilon \\ &\mid +Term \,\{printf(" add ");\}\, ERest \end{aligned}$$

$$\begin{aligned} Term &::= num \,\{printf("LOADC\ \%d ", yylval);\}\, TRest \\ TRest &::= \varepsilon \\ &\mid * Term \ \{printf(" mult ");\} \end{aligned}$$

Diese Grammatik lässt sich jetzt unmittelbar in ein C- (oder Java-)Programm umschreiben.

4.8.1 Fehlererkennung

Fehler im Quelltext eines Programms können sowohl vom Scanner als auch vom Parser entdeckt werden. Der Scanner kann Zeichenfolgen erhalten, die sich nicht zu gültigen Token gruppieren lassen, wie z.B. „§“ oder „!?“. Diese können einfach durch „Invalid Token“ gemeldet werden, wobei noch die Zeile und Spalte im Quelltext angegeben wird, oder der Editor kann die Eingabemarke an der fraglichen Stelle platzieren. Fehler treten beim Parsen auf, wenn sich die eingehenden Token nicht zu der gewünschten Satzform gruppieren lassen. Entsprechende Meldungen kann man schon bei der yacc-Spezifikation als grammatische Aktionen einbauen, etwa in der Form:

$$
\begin{array}{lcl}
\mathit{Zuweisung} & ::= & \mathit{id}\ :=\ \mathit{Expr} \\
& | & \mathit{id}\,\mathit{fehler}(\text{”}\,\mathit{Expr}\,\mathit{erwartet}\,\text{”}); \\
& | & \mathit{id}\,\mathit{fehler}(\text{”}\ =\ \mathit{erwartet}\,\text{”});
\end{array}
$$

Nützlich ist die Fehlermeldung aber nur, wenn auch die Fundstelle angegeben wird. Zu diesem Zweck kann der Scanner jedem Token seine Fundstelle im Quelltext als Attribut mitgeben.

4.8.2 Synthetisierte Werte

yacc und *bison* können im Input gefundenen Terminalen und Nonterminalen Werte zuordnen, die während des Parsens berechnet und propagiert werden. Klassisch ist das Beispiel der Spezifikation eines Taschenrechners durch eine *yacc*-Grammatik.

Jedes ***num***-Token erhält seinen Zahlenwert als Wert. Alle relevanten Grammatikregeln ergänzt man um Angaben, wie der neue Wert berechnet werden soll, wenn mit der Regel reduziert wird.

Beispielsweise besagt die erste Zeile der Regel

$$
\begin{array}{lcl}
\mathit{Expr} & ::= & \mathit{Expr}\ \mathit{PLUS}\ \mathit{Expr}\ \{\ \$\$ = \$1\ +\ \$3\ \} \\
& | & \boldsymbol{num}\ \{\$\$\ =\ \$1;\ \}
\end{array}
$$

dass der neue Wert ($$) des *Expr* auf der linken Seite sich aus den Werten des ersten und dritten Knotens der Regel ergibt.

Die Aktion der zweiten Zeile ist offensichtlich. Da sie die default-aktion ist, kann sie auch weggelassen werden. Das Token ***num*** erhält seinen Wert im Scanner. Dieser speichert ihn in der gemeinsamen Variablen `yylval`.

Solche synthetisierte Werte kann man auch zur Typüberprüfung verwenden. In der obigen Regel würde man aus den Typen der rechten Seiten (etwa `float` und `int`) einen umfassenden Typ berechnen und diesen der linken Seite zuordnen. Bei der Zuweisung

$$\mathit{Zuweisung} ::= \boldsymbol{id} = \mathit{Expr}$$

würde überprüft, dass der deklarierte Typ des ***id*** mit dem Typ des *Expr* kompatibel ist. Um aber festzuhalten, welcher ***id*** welchen Typ haben soll, benötigt man eine global zugreifbare Datenbasis, eine sogenannte *Symboltabelle*.

4.8.3 Symboltabellen

Während der Syntaxprüfung legt ein Parser (mindestens) eine Symboltabelle an, in der wichtige Bestimmungsstücke von Variablen, zumindest aber Name, Typ und Speicherplatz festgehalten werden.

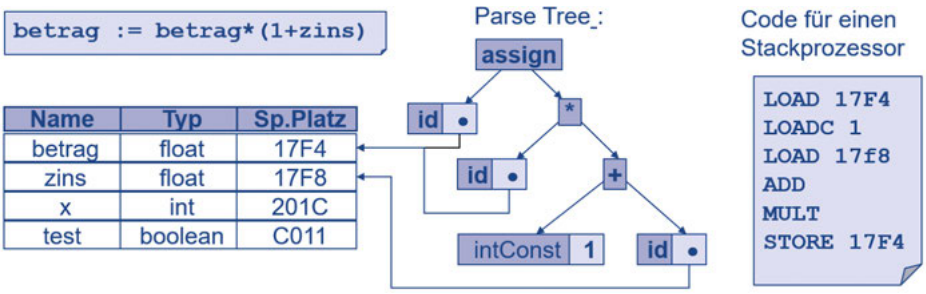


Abb. 4.8.1: Symboltabelle, Ableitungsbaum und erzeugter Code

Trifft der Parser auf einen neuen Bezeichner, so prüft er, ob dieser bereits in der Symboltabelle vorhanden ist. Beim Parsen einer Deklaration sollte dies noch nicht der Fall sein. Der Bezeichner zusammen mit seinem Typ wird eingefügt, und ihm wird eine Speicheradresse zugeordnet. Beim Parsen einer Anweisung muss verifiziert werden, dass jeder darin vorkommende Bezeichner schon in der Symboltabelle verzeichnet ist. Viele Sprachen erlauben geschachtelte Gültigkeitsbereiche. Dementsprechend muss in der Praxis meist eine Hierarchie von Symboltabellen verwaltet werden.

4.8.4 Codeoptimierung

Nach der Codeerzeugung eines Compilers schließt sich eine Phase der *Codeoptimierung* an. Dabei wird, meist mit speziellen Regeln und Heuristiken, versucht, das automatisch erzeugte Maschinenprogramm zu beschleunigen. Einige der Regeln sind:

Erkennen identischer Teilausdrücke, z.B. kann man die zweimalige Auswertung des Teilausdrucks `x*x` in `if(x*x < 32767) x = x*x;` vermeiden, indem man seinen Wert an eine temporäre Variable bindet:

```
temp=x*x; if (temp<10) x=temp;
```

Berechnung und Propagierung von Konstanten, Entfernung toten Codes: Beispiel `debug=false; ...` und später im gleichen Programm

```
... if(debug)System.out.println("x ist"+x);
```

Algebraische Tricks zur Beschleunigung von Berechnungen: `x+x` oder `x*2` können effizienter mittels `left shift x<<1` berechnet werden.

Ausdrücke, deren Variablen sich nicht ändern, in Schleifen: Dies betrifft sowohl den Schleifenrumpf:

```
while(sum<100){sum += y*y%x; x = x+1; }
```

wird zu

```
tmp=y*y; while(sum<100){sum += tmp%x; x = x+1}
```

als auch die Abbruchbedingung:

```
while(x<y*y){sum += x++;}
```

wird zu

```
tmp=y*y; while(x<tmp){sum+=x++;}
```

Bei den meisten der obigen Beispiele könnte man einwenden, dass ein Programmierer sich ziemlich dumm angestellt hat, wenn er derart ineffizienten Code schreibt. Allerdings können solche Codeteile auch als Zwischenschritte vorangegangener Transformationen entstanden sein.

Kapitel 5

Berechenbarkeit

5.1 Unendliche Mengen und Cantor's Trick

Heutzutage bereitet es uns keine Schwierigkeiten von endlichen und unendlichen Mengen zu reden. Das war nicht immer so. Die Einführung und Erforschung unendlicher Mengen durch Georg Cantor traf auf großen Widerstand bei einflussreichen Mathematikern seiner Zeit. Unendliche Mengen waren vielleicht als Sprechweise zulässig aber nicht als wirkliche mathematische Objekte, deren Eigenschaften man studieren kann. Cantors beharrliche Forschung förderte aber soviel interessante und tiefliegende Mathematik zutage, dass einer der größten und besonders an den Grundlagen interessierter Mathematiker, David Hilbert, mit dem Ausspruch zitiert wird: "*Aus dem Paradies, das Cantor uns geschaffen, soll uns niemand vertreiben können.*"

5.1.1 Abbildungen

Definition 5.1.1. Eine *Abbildung* oder *Funktion* $f : M \to S$ zwischen zwei Mengen M und S ordnet jedem Element $m \in M$ ein eindeutig bestimmtes Element $f(m) \in S$ zu. $f(m)$ heißt auch das *Bild* von m. Als Formel geschrieben:

$$\forall m \in M . f(m) \in S.$$

Aus der Schule kennt man Funktionen wie $g(x) = x^2$ oder $h(x) = \sqrt{x}$. Start- und Zielmenge werden dabei meist als klar vorausgesetzt, so z.B. $g : \mathbb{R} \to \mathbb{R}$. Informatiker sind aber schon gewöhnt, diese explizit als Typinformation bei einer Funktionsdefinition anzugeben, etwa in der Form `def g(x:Real):Real` oder ähnlich.

Zu jeder Wahl von $r \in \mathbb{R}$ ist $g(r) = r * r$ ein eindeutig bestimmtes Element aus der Bildmenge $\mathbb{R}$. Daher ist $g : \mathbb{R} \to \mathbb{R}$ tatsächlich eine Funktion.

Es ist durchaus erlaubt, dass verschiedene Argumente den gleichen Wert liefern, so z.B. $g(-1) = g(1) = 1$, es ist aber *nicht erlaubt*, dass ein Argument mehrere Bildwerte liefert, etwa $p(x) = \pm\sqrt{x}$.

https://doi.org/10.1515/9783110442397-006

Mit $\sqrt{x}$ ist die eindeutig bestimmte positive Wurzel von x gemeint. Trotzdem ist $h(x) = \sqrt{x}$ noch keine Abbildung von $\mathbb{R}$ nach $\mathbb{R}$, da $h(x)$ für negative Zahlen x nicht definiert ist. Daher schränken wir die erlaubten Argumente von h auf alle positiven reellen Zahlen $\mathbb{R}_+ := \{r \in \mathbb{R} \mid r \geq 0\}$ ein und erhalten mit $h(x) = \sqrt{x}$ eine Abildung von $\mathbb{R}_+$ nach $\mathbb{R}$, also $h : \mathbb{R}_+ \to \mathbb{R}$.

Definition 5.1.2. Eine Abbildung $f : M \to S$ heißt
injektiv, falls verschiedene Elemente auch verschiedene Bilder haben, also falls

$$\forall m_1, m_2 \in M. f(m_1) = f(m_2) \implies m_1 = m_2. \tag{5.1.1}$$

surjektiv, falls jedes Element $s \in S$ Bild mindestens eines Elementes von M ist, also:

$$\forall s \in S. \exists m \in M. f(m) = s. \tag{5.1.2}$$

bijektiv, falls f sowohl injektiv als auch surjektiv ist.

Beispiel 5.1.3. Die Abbildung $g : \mathbb{R} \to \mathbb{R}$ mit $g(x) = x * x$ ist nicht injektiv, denn es gilt z.B. $g(-3) = g(3)$ aber $-3 \neq 3$. Sie ist auch nicht surjektiv, denn es gibt z.B. kein x mit $g(x) = -4$.

Die Abbildung $h : \mathbb{R}_+ \to \mathbb{R}$ mit $h(x) = \sqrt{x}$ ist injektiv, denn aus $\sqrt{r_1} = \sqrt{r_2}$ folgt durch quadrieren: $r_1 = r_2$. Sie ist nicht surjektiv, da beispielsweise kein $x \in \mathbb{R}_+$ existiert mit $\sqrt{x} = -1$.

Injektive Abbildungen kennzeichnen wir manchmal durch einen Pfeil mit gespaltenem Anfang wie in $f : M \rightarrowtail S$ und surjektive Abbildungen durch einen Pfeil mit Doppelspitze, wie in $f : M \twoheadrightarrow S$. Konsequenterweise bedeutet $f : M \rightarrowtail\!\!\!\!\twoheadrightarrow S$, dass f bijektiv ist.

Die identische Abbildung einer Menge M auf sich selbst bezeichnen wir mit id_M. Falls $M \subseteq S$ ist, kann die *Inklusionsabbildung* $\iota(m) = m$ auch als Abbildung von M nach S aufgefasst werden – sie ist immer injektiv, aber nur dann surjektiv, wenn $M = S$ ist. Diese spezielle Abbildung kennzeichnen wir durch einen Pfeil, dessen Beginn an das Zeichen für Mengeninklusion erinnert: $M \xhookrightarrow{\iota} S$.

Sind $f : A \to B$ und $g : B \to C$ Abbildungen, so bezeichnen wir mit $g \circ f : A \to C$ die *Komposition*, die ein Element $a \in A$ zuerst mit f abbildet und dann das Ergebnis mit g, also

$$(g \circ f)(a) := g(f(a)).$$

Wir erhalten sofort das Lemma:

Lemma 5.1.4. *Seien $f : A \to B$ und $g : B \to C$ Abbildungen.*
– *Sind f und g injektiv (surjektiv, bijektiv), dann ist auch $g \circ f$ injektiv (surjektiv, bijektiv).*
– *Ist $g \circ f$ injektiv, dann ist auch f injektiv.*

– *Ist $g \circ f$ surjektiv, dann ist auch g surjektiv.*

Beweis. Wir zeigen exemplarisch nur die letzte Behauptung. Die übrigen sind genauso leicht und als Übung empfohlen:

Um zu zeigen, dass g surjektiv ist, müssen wir zeigen: $\forall c \in C.\exists b \in B.g(b) = c$.

Sei also $c \in C$. Weil $g \circ f$ als surjektiv vorausgesetzt ist, gibt es ein $a \in A$ mit $(g \circ f)(a) = c$, also $g(f(a)) = c$. Mit $b := f(a)$ folgt jetzt $g(b) = c$. □

Lemma 5.1.5. *Ist $M \neq \emptyset$ dann ist $f : M \to S$ injektiv genau dann wenn es eine Abbildung $g : S \to M$ gibt mit $g \circ f = id_M$.*

Beweis. Da $M \neq \emptyset$ gibt es ein $m_0 \in M$. Definiere für beliebige $s \in S$:

$$g(s) := \begin{cases} m & \text{falls } f(m) = s \\ m_0 & \text{sonst.} \end{cases}$$

Da f injektiv ist, gibt es zu jedem $s \in S$ höchstens ein m mit $f(m) = s$. Daher ist g eine Abbildung und wir rechnen für beliebige $m \in M$ nach:

$$(g \circ f)(m) = g(f(m)) = m = id_M(m),$$

also $g \circ f = id_M$.

Umgekehrt sei $g \circ f = id_M$. Da id_M bijektiv ist, insbesondere also auch injektiv, folgt aus $g \circ f = id_M$ mit dem vorigen Lemma, dass f injektiv ist. □

Korollar 5.1.6. *$f : M \to S$ ist genau dann bijektiv, wenn es eine Abbildung $g : S \to M$ gibt mit $g \circ f = id_M$ und $f \circ g = id_S$. In diesem Fall ist g eindeutig und wir nennen es f^{-1}.*

Beweis. Sei $f : M \to S$ bijektiv. Falls $M = \emptyset$ muss auch $S = \emptyset$ sein und $f = g = id_\emptyset$. Ansonsten liefert Lemma 5.1.5 eine Abbildung $g : S \to M$ mit $g \circ f = id_M$. Weil f surjektiv ist, gibt es für jedes $s \in S$ ein $m \in M$ mit $f(m) = s$. Es folgt

$$(f \circ g)(s) = (f \circ g)(f(m)) = (f \circ g \circ f)(m) = (f \circ id_M)(m) = f(m) = s,$$

also $f \circ g = id_S$.

Für die umgekehrte Richtung folgt aus $f \circ g = id_S$ und $g \circ f = id_M$ mit Lemma 5.1.4, dass f surjektiv und injektiv ist, also bijektiv.

Gäbe es eine weitere Abbildung g' mit $f \circ g' = id_S$ dann hätten wir

$$g' = id_M \circ g' = g \circ f \circ g' = g \circ id_S = g.$$

Somit ist $g = f^{-1}$ eindeutig. □

5.1.2 Endliche und unendliche Mengen

Als *endliche Menge* bezeichnet man eine Menge M, deren Elemente man mit den Zahlen von $1 \ldots k$ durchzählen kann. Alle anderen Mengen sind unendliche Mengen.

„Durchzählen“ bedeutet: *Eine surjektive Abbildung von den Zahlen* $\{1, 2, \ldots, k\}$ *auf M angeben,* und *Surjektivität* bedeutet hier, dass bei der Durchzählung jedes Element von M mindestens einmal vorkommt.

Gibt es eine solche Abbildung

$$m : \{1, \ldots, k\} \twoheadrightarrow M,$$

so können wir dies durch die Schreibweise $M = \{m_1, m_2, \ldots, m_k\}$ ausdrücken, wobei man traditionell m_i statt $m(i)$ schreibt. Es ist in dieser Darstellung durchaus erlaubt, dass zwei oder mehrere der m_i gleich sein können.

Der Spezialfall $k = 0$ liefert uns die leere Menge $\emptyset = \{\}$ die damit auch als endliche Menge mit 0 Elementen gilt.

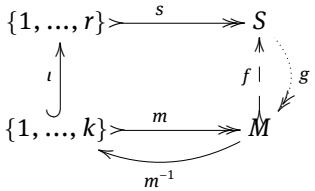
Definition 5.1.7. Eine Menge M heißt *endlich*, wenn es eine natürliche Zahl k gibt und eine surjektive Abbildung $m : \{1, 2, \ldots, k\} \twoheadrightarrow M$.

Wenn es ein k wie in der Definition gefordert gibt, dann gibt es auch ein kleinstes solches k. Diese Zahl heißt dann die *Mächtigkeit* der Menge M und man schreibt $|M| = k$. Die zugehörige Abbildung $m : \{1, \ldots, k\} \to M$ ist in diesem Fall nicht nur surjektiv, sondern auch injektiv, also bijektiv. Es folgt:

Satz 5.1.8 (Mächtigkeit)**.** *Für zwei endliche Mengen $M \neq \emptyset$ und S sind äquivalent:*

1. $|M| \leq |S|$
2. *Es gibt eine injektive Abbildung $f : M \rightarrowtail S$*
3. *Es gibt eine surjektive Abbildung $g : S \twoheadrightarrow M$*

Beweis. $M \neq \emptyset$ und S sind endlich, es gibt also natürliche Zahlen $k > 0$ und r sowie bijektive Abbildungen $m : \{1, \ldots, k\} \to M$ und $s : \{1, \ldots, r\} \to S$.



1. $\to$ 2. : Aus $|M| \leq |S|$ folgt $k \leq r$, also $\{1, \ldots, k\} \subseteq \{1, .., r\}$. Die Abbildung $\iota : \{1, \ldots, k\} \hookrightarrow \{1, \ldots, r\}$ mit $\iota(x) = x$ ist somit wohldefiniert und injektiv.
Da m bijektiv ist, gibt es eine (ebenfalls bijektive) Umkehrabbildung $m^{-1} : M \to \{1, \ldots, k\}$, somit ist die Komposition

$$f := (s \circ \iota \circ m^{-1})$$

eine Komposition injektiver Abbildung und daher selber eine injektive Abbildung $f : M \rightarrowtail S$.

2. → 3. : Wegen $M \neq \emptyset$ liefert Lemma 5.1.5 eine Abbildung $g : S \to M$ mit $g \circ f = id_M$. Wegen Lemma 5.1.4 ist g surjektiv.

3. → 1. : Aus einer surjektiven Abbildung $g : S \twoheadrightarrow M$ und den bijektiven m und s gewinnen wir eine surjektive Abbildung

$$h := m^{-1} \circ g \circ s : \{1, \ldots, r\} \twoheadrightarrow \{1, \ldots, k\}.$$

Das bedeutet aber $r \geq k$, also $|S| \geq |M|$.

□

5.1.3 Mächtigkeiten

Bei unendlichen Mengen können Effekte auftreten, welche man von endlichen Mengen nicht kennt. So können echte Teilmengen gleich viele Elemente haben, wie die ganze Menge, weil es eine bijektive Abbildung zwischen der Menge und einer echten Teilmenge geben kann.

Als einfaches Beispiel vergleichen wir die Menge aller natürlichen Zahlen $\mathbb{N}$ mit der Teilmenge aller echt positiven natürlichen Zahlen

$$\mathbb{N}_+ := \{n \in \mathbb{N} \mid n > 0\}.$$

Die Abbildung $succ$ mit $succ(n) = n + 1$ liefert offensichtlich eine bijektive Abbildung zwischen $\mathbb{N}$ und ihrer echten Teilmenge $\mathbb{N}_+$.

Ähnlich liefert $g(n) := 2 \cdot n$ eine bijektive Funktion zwischen allen natürlichen Zahlen und der Teilmenge aller geraden Zahlen und

$$f(z) := \begin{cases} 2 * z & z \geq 0 \\ 2 * (-z) - 1 & z < 0 \end{cases}$$

stellt eine bijektive Abbildung $f : \mathbb{Z} \to \mathbb{N}$ her.

Für unendliche Mengen wie $\mathbb{N}$, $\mathbb{Z}$, und $\mathbb{Q}$, die Menge der rationalen Zahlen (Bruchzahlen) oder $\mathbb{R}$, die Menge der reellen Zahlen haben wir keine „Maßzahl" zur Verfügung, um ihre Größe zu messen.

Ein naiver Vorschlag wäre, $|M| = \infty$ für alle unendlichen Mengen zu schreiben. Dann hätten wir aber $|\mathbb{R}| = \infty$ und $|\mathbb{N}| = \infty$ und nach den üblichen Gleichheitsregeln müssten wir daraus $|\mathbb{R}| = |\mathbb{N}|$ folgern.

Dass dies aber nicht der Fall sein kann, hat *Cantor* mit seinem cleveren *Diagonalschluss* bewiesen. Dieser Trick diente später auch als Vorlage für die berühmte *Russellsche Antinomie* wie auch für die berühmten Unmöglichkeitsbeweise von Alan Turing und Kurt Gödel.

Daher verzichten wir bei unendlichen Mengen M darauf, zu definieren, was „$|M|$" als alleinstehender Begriff sein soll. Stattdessen definieren wir eine Relation zwischen zwei Mengen M und S, die wir einfach mit $|M| \leq |S|$ bezeichnen. Satz 5.1.8 liefert die Idee. Wir definieren:

Definition 5.1.9. Für beliebige Mengen M und S bedeute $|M| \leq |S|$, dass es eine injektive Abbildung $f : M \to S$ gibt. Kann f sogar bijektiv gewählt werden, dann schreiben wir $|M| = |S|$ und sagen, dass M und S *gleichmächtig* sind. Falls $|M| \leq |S|$ ist aber $|M| \neq |S|$, dann schreiben wir $|M| < |S|$.

Da die Komposition injektiver Abbildungen injektiv ist, gilt für beliebige Mengen M, S und T: Aus $|M| \leq |S|$ und $|S| \leq |T|$ folgt $|M| \leq |T|$. Ebenfalls gilt $|M| \leq |M|$, denn id_M, die Identität auf M, ist injektiv.

Der folgende harmlos anmutende Satz ist nicht so einfach zu beweisen wie es den Anschein haben mag. Vermutet worden war er schon von Georg Cantor. Dessen (damals 19-jähriger) Schüler Felix Bernstein und unabhängig davon Ernst Schröder fanden fast gleichzeitig jeweils einen Beweis. Deswegen ist das Resultat auch als *Satz von Schröder-Bernstein* bekannt:

Satz 5.1.10 (Schröder, Bernstein)**.** *Für beliebige Mengen M und S gilt: Aus $|M| \leq |S|$ und $|S| \leq |M|$ folgt $|M| = |S|$.*

Der Beweis des Satzes ist leider nicht ganz einfach, daher wollen wir ihn hier nicht führen, sondern auf die Literatur verweisen. Wichtig ist vor allem, zu erkennen, dass da etwas zu beweisen ist:

Aus einer injektiven Abbildung $f : M \rightarrowtail S$ und einer injektiven Abbildung $g : S \rightarrowtail M$ sollen Sie auf die Existenz einer bijektiven Abbildung h zwischen M und S schließen. Auf „konstruktive“ Weise ist dies nicht möglich, die Existenz einer solchen Abbildung h kann man nur über einen Widerspruchsbeweis absichern.

5.1.4 Das Auswahlaxiom

Die Äquivalenz von 1. und 2. in Satz 5.1.8 gilt auch für unendliche Mengen M und S aus dem einfachen Grund, weil sie dort zur Definition von $|M| \leq |S|$ benutzt wird..

Für die Beweisrichtung, $2. \Rightarrow 3.$ kann der Beweis wörtlich auch für unendliche Mengen übernommen werden. Die Richtung $3. \Rightarrow 2.$ ist für unendliche Mengen jedoch äquivalent zu einem Axiom der Mengenlehre:

Fakt 5.1.11 (Auswahlaxiom)**.** *Zu jeder surjektiven Abbildung $f : M \to S$ gibt es eine injektive Abbildung $g : S \to M$ mit $f \circ g = id_S$.*

Das Auswahlaxiom wird von den meisten Mathematikern als Axiom akzeptiert, obwohl es äquivalent zu Aussagen ist, die überhaupt nicht plausibel scheinen. So folgt aus dem Auswahlaxiom unter anderem, dass man eine Kugel in 5 Teile zerlegen kann, die, wenn man sie wieder zusammensetzt, zwei Kugeln ergeben, deren jede genauso groß ist, wie die Kugel.

Andererseits ist das Auswahlaxiom äquivalent zu der Forderung, dass es zu jeder Familie $(A_i)_{i\in I}$ von nichtleeren Mengen eine Funktion $f : I \to \bigcup_{i\in I} A_i$ geben soll mit $f(i) \in A_i$ für jedes $i \in I$.

Bertrand Russel erklärte es einmal so:[1] „*Aus unendlich vielen Paaren von Schuhen kann man leicht eine Menge von Schuhen auswählen, die genau einen Schuh von jedem Paar enthält – beispielsweise die Menge aller linken Schuhe. Aus unendlich vielen Paaren von Socken ist es schwieriger eine Menge auszuwählen, die aus jedem Paar genau eine Socke enthält. Dazu benötigt man das Auswahlaxiom.*"

In der Tat, können wir im ersten Falle die Auswahlfunktion konkret angeben – *wähle den linken von jedem Paar von Schuhen* – im zweiten Fall denken wir, dass eine Funktion machbar ist, *wähle irgendeine von dem Paar von Socken*, aber konkret können wir die Funktion nicht benennen.

Man kann das Auswahlaxiom mit dem Parallelenaxiom der Euklidischen Geometrie vergleichen. Es folgt nicht aus den elementaren Axiomen der Mengenlehre und man kann es zu den Basisaxiomen hinzunehmen (oder stattdessen seine Verneinung), ohne einen Widerspruch zu erzeugen.

Die überwiegende Mehrheit aller Mathematiker akzeptiert das Auswahlaxiom, denn ohne dieses können viele liebgewordene Sätze, so etwa der *Zwischenwertsatz der Analysis*, nicht bewiesen werden. Daher werden auch wir das Auswahlaxiom stets benutzen. Damit können wir Satz 5.1.8 auch für unendliche Mengen übernehmen. Die Äquivalenz von 1. und 2. wird zu Definition 5.1.9 und die Äquivalenz von 2. mit 3. ist das Auswahlaxiom.

Korollar 5.1.12. *Für beliebige Mengen $M \neq \emptyset$ und S sind äquivalent:*

1. $|M| \leq |S|$
2. *Es gibt eine injektive Abbildung $g : M \to S$*
3. *Es gibt eine surjektive Abbildung $h : S \to M$*

Ebenso folgt mit Hilfe des Satzes von Schröder-Bernstein und dem Auswahlaxiom:

Korollar 5.1.13. *Für beliebige Mengen M und S sind äquivalent :*

1. $|M| = |S|$
2. $|M| \leq |S|$ *und* $|S| \leq |M|$
3. *Es gibt Abbildungen $f_1, f_2 : M \to S$ so dass f_1 injektiv ist und f_2 surjektiv.*

1 "The Axiom of Choice is necessary to select a set from an infinite number of socks, but not an infinite number of shoes." — Bertrand Russell

5.1.5 Der Satz von Cantor

Nach diesen Vorbereitungen können wir den Satz von Cantor formulieren. Dieser besagt, dass die *Potenzmenge*

$$\mathbb{P}(M) := \{U \mid U \text{ ist Teilmenge von } M\}$$

immer eine echt größere Mächtigkeit hat, als M selber.

Theorem 5.1.14 (Cantor)**.** *Für jede Menge M (endlich oder unendlich) gilt*

$$|M| < |\mathbb{P}(M)|.$$

Beweis. Eine injektive Funktion von M nach $\mathbb{P}(M)$ ist schnell gefunden; Wir ordnen jedem $m \in M$ die Einermenge $\{m\} \in \mathbb{P}(M)$ zu.

Eine surjektive Funktion von M nach $\mathbb{P}(M)$ ist allerdings unmöglich. Dies zeigen wir durch einen Widerspruchsbeweis:

Angenommen, wir hätten eine surjektive Abbildung $f : M \to \mathbb{P}(M)$. Dann betrachten wir folgende Teilmenge von M:

$$C := \{x \in M \mid x \notin f(x)\}. \tag{5.1.3}$$

C ist eine Teilmenge von M, also ein Element der Potenzmenge: $C \in \mathbb{P}(M)$. Wegen der Surjektivität von f muss es daher ein $m \in M$ geben mit

$$f(m) = C \tag{5.1.4}$$

Wir stellen uns nun die Frage, ob $m \in f(m)$ gilt oder nicht und gelangen zu dem merkwürdigen Widerspruch

$$m \in f(m) \overset{(5.1.4)}{\iff} m \in C \overset{(5.1.3)}{\iff} m \notin f(m).$$

Daher ist unsere Annahme, dass es eine surjektive Abbildung $f : M \to \mathbb{P}(M)$ gibt, falsch. □

Für den Fall $M = \mathbb{N}$ können wir uns den Sachverhalt geometrisch veranschaulichen: Gäbe es eine surjektive Abbildung $f : \mathbb{N} \twoheadrightarrow \mathbb{P}(\mathbb{N})$, dann könnten wir uns eine unendliche Tabelle F vorstellen mit

$$F[m, n] = \begin{cases} 1 & \text{falls } n \in f(m) \\ 0 & \text{sonst.} \end{cases}$$

Jede Zeile $F[i, -]$ von F codiert dann dieTeilmenge $f(i) = \{n \in \mathbb{N} \mid F[i, n] = 1\}$. Wäre f surjektiv, so müsste jede Teilmenge $C \subseteq \mathbb{N}$ irgendwann als Tabellenzeile vorkommen.

F	0	1	2	3	4	5	...
$f(0)$	**0**	1	1	0	0	1	...
$f(1)$	1	**1**	0	1	1	1	...
$f(2)$	1	0	**1**	1	0	1	...
$f(3)$	0	0	0	**0**	0	0	...
$f(4)$	1	0	1	0	**1**	0	...
$f(5)$	1	1	1	0	0	**0**	...
...	...	...	...	...	...	...	...

es fehlt:

$f(i)$	**1**	**0**	**0**	**1**	**0**	**1**	...

Abb. 5.1.1: Diagonalargument

Aus den Diagonalelementen der Tabelle gewinnen wir nun eine Menge, die garantiert als Zeile der Tabelle vergessen wurde. Dazu negieren wir die Einträge der Diagonale – aus 0 wird 1 und umgekehrt:

$$C := \{n \in \mathbb{N} \mid F[n, n] = 0\} = \{n \in \mathbb{N} \mid n \notin f(n)\}.$$

$C \in \mathbb{P}(\mathbb{N})$ ist klar, aber C unterscheidet sich von jeder bisher codierten Menge $f(i)$, denn aus der Annahme

$$C = f(i)$$

folgt

$$i \in f(i) \iff i \in C \iff F[i, i] = 0 \iff i \notin f(i).$$

Somit ist $C \neq f(i)$ für jedes i, also kann f nicht surjektiv gewesen sein.

5.1.6 Abzählbarkeit

Definition 5.1.15. Eine Menge M heißt *abzählbar*, falls $|M| \leq |\mathbb{N}|$ gilt, falls es also eine injektive Abbildung $\iota : M \rightarrowtail \mathbb{N}$ gibt. Unendliche Mengen, die nicht abzählbar sind, heißen *überabzählbar*.

Aufgrund von Korollar 5.1.12 ist M abzählbar falls $M = \emptyset$ ist oder es eine surjektive Abbildung $g : \mathbb{N} \twoheadrightarrow M$ gibt.

Endliche Mengen sind abzählbar:

Ist eine Menge abzählbar, so gilt dies offensichtlich auch für jede ihrer Teilmengen $U \subseteq M$. Insbesondere ist jede endliche Menge abzählbar.

Kartesische Produkte abzählbarer Mengen sind abzählbar:

Die Menge aller Paare von natürlichen Zahlen, also

$$\mathbb{N} \times \mathbb{N} = \{(n_1, n_2) \mid n_1, n_2 \in \mathbb{N}\}$$

ist ebenfalls abzählbar. Wir werden in Abschnitt 5.2.5 eine Bijektion

$$pair : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$$

mit Umkehrfunktion

$$unpair : \mathbb{N} \to \mathbb{N} \times \mathbb{N}$$

konstruieren. Damit ist auch $\mathbb{N} \times \mathbb{N}$ abzählbar. Es folgt, dass für beliebige abzählbare Mengen A und B auch deren kartesisches Produkt $A \times B$ abzählbar ist: Aus surjektiven Abbildungen $f : \mathbb{N} \twoheadrightarrow A$ und $g : \mathbb{N} \twoheadrightarrow B$ erhalten wir nämlich die surjektive Abbildung

$$(f, g) \circ unpair : \mathbb{N} \twoheadrightarrow A \times B,$$

wobei

$$(f, g) : \mathbb{N} \times \mathbb{N} \to A \times B$$

die Abbildung ist, welche jedem Paar $(x, y) \in \mathbb{N} \times \mathbb{N}$ das Paar $(f(x), g(y)) \in A \times B$ zuordnet.

$\mathbb{Z}$ ist abzählbar:

Die Abbildung $f(n) := (-1)^n \cdot (n + 1)/2$, wobei wir $'/'$ als Ganzzahldivision ohne Rest auffassen wollen, liefert eine bijektive Funktion von $\mathbb{N}$ auf die ganzen Zahlen $\mathbb{Z}$ mit Umkehrfunktion $f^{-1}(z) = \mathsf{if}\,(z \geq 0)\,\mathsf{then}\ 2z\ \mathsf{else}\ -2z - 1$. Somit ist $\mathbb{Z}$ abzählbar.

$\mathbb{Q}$ ist abzählbar:

Die Menge $\mathbb{Q}$ der rationalen Zahlen ist ebenfalls abzählbar. Eine rationale Zahl r ist ein Bruch $\frac{p}{q}$ wobei p eine ganze Zahl ist und $q \neq 0$ eine natürliche Zahl ist. Zwei Brüche $\frac{p}{q}$ und $\frac{r}{s}$ sind gleich, falls $p \cdot s = r \cdot q$ gilt.

Da jede natürliche Zahl n auch als Bruch $\frac{n}{1}$ geschrieben werden kann, und aus $m \neq n$ auch $\frac{m}{1} \neq \frac{n}{1}$ folgt, haben wir mit $f(n) := \frac{n}{1}$ bereits eine injektive Funktion $f : \mathbb{N} \to \mathbb{Q}$.

Eine surjektive Funktion von $f : \mathbb{N} \times \mathbb{N}_+ \to \mathbb{Q}$ ordnet jedem Paar (m, n) mit $n \neq 0$ den Bruch $\frac{m}{n}$ zu, somit ist

$$f \circ (id, succ) : \mathbb{N} \times \mathbb{N} \to \mathbb{Q}$$

eine surjektive Abbildung, sodass mit $f \circ (id, succ) \circ unpair$ auch eine surjektive Abbildung von $\mathbb{N}$ nach $\mathbb{Q}$ gefunden ist.

Ebenso kann man zeigen, dass $\mathbb{N}^2$, $\mathbb{N}^3$, ... und sogar $\mathbb{N}^\star$ abzählbar sind. Dies werden wir im nächsten Kapitel sehen, wenn wir sogar eineindeutige Codierungen zwischen $\mathbb{N}$ und $\mathbb{N}^k$ bzw. $\mathbb{N}^\star$ untersuchen.

$\mathbb{P}(\mathbb{N})$ ist überabzählbar:

Dass $|\mathbb{N}| < |\mathbb{P}(\mathbb{N})|$ ist und folglich $\mathbb{P}(\mathbb{N})$ überabzählbar, folgt unmittelbar aus dem Satz von Cantor.

Die Vermutung, dass es keine Menge S gibt mit $|\mathbb{N}| < |S| < |\mathbb{P}(\mathbb{N})|$ war als *Kontinuumshypothese* bekannt, bis sie 1963 von Paul Cohen gelöst wurde.[2]

$\mathbb{R}$ ist überabzählbar:

Dazu brauchen wir nur die rellen Zahlen aus dem Intervall $[0, 1)$ zu betrachten. Stellen wir uns diese als binäre Kommazahlen in der Form $r = 0.a_1a_2a_3....$ aufgeschrieben vor, so können wir jeder solchen Zahl eine Teilmenge U_r der ganzen Zahlen zuordnen:

$$U_r := \{i \mid a_i = 0\}$$

und umgekehrt erhalten wir zu jeder Teilmenge $U \subseteq \mathbb{N}$ von die reelle Zahl $r_U := 0.a_1a_2a_3...$ wobei $a_i = \text{if } (i \in U) \text{ then } 1 \text{ else } 0$. Damit scheint es so, als hätten wir mindestens so viele reelle Zahlen im Intervall $[0, 1)$ wie Teilmengen von $\mathbb{N}$.

Leider hat diese Übersetzung noch einen kleinen Schönheitsfehler, denn so wie man Dezimalzahlen, die hinter dem Komma irgendwann mit 9 periodisch werden, mit der nächsten abbrechenden Dezimalzahl identifizieren muss (z.B. $0.4299999... = 0.42\bar{9} = 0.43\bar{0} = 0.43$) so muss man es in Binärdarstellung mit Dezimalzahlen halten, die irgendwann mit 1 periodisch werden:

$$(0.10101111111...)_2 = (0.1010\bar{1})_2 = (0.1011\bar{0})_2 = (0.1011)_2.$$

Daher verlassen wir das Binärsystem und betrachten (z.B. im Dezimalsystem) lediglich diejenigen Kommazahlen, die ausschließlich mit den Ziffern 0 und 1 gebildet sind. Da 9 nicht vorkommt, umgehen wir die Komplikation mit der Periode 9. Das liefert eine echte Teilmenge $D \subset [0,1)$ mit einer injektiven Abbildung

$$r : \mathbb{P}(\mathbb{N}) \to D \subseteq [0, 1) \subseteq \mathbb{R},$$

wobei $r(U)$ formal wie oben definiert ist, aber das Ergebnis als Dezimalzahl aufgefasst wird: $r(U) := (0.a_1a_2a_3...)_{10}$ mit $a_i = \text{if } (i \in U) \text{ then } 1 \text{ else } 0$.

---

2 Die Kontinuumshypothese ist weder wahr noch falsch – sie folgt nicht aus den üblichen Axiomen der Mengenlehre, so wie auch das Parallelenaxiom nicht aus den üblichen Axiomen von Euklid folgt.

5.2 Algorithmen und berechenbare Funktionen

5.2.1 Algorithmen

Ein *Algorithmus* ist im allgemeinen Sprachgebrauch eine Vorgehensbeschreibung zur Lösung eines Problems. Allgemein verlangt man, dass ein Algorithmus in diskreten Schritten abläuft und zu jedem Zeitpunkt klar sein muss, was als nächstes gemacht werden soll:

- Algorithmen steuern Maschinen, Fahrzeuge, Produktionsprozesse, Fabrikanlagen,
- Algorithmen analysieren und organisieren riesige Datenmengen,
- in der Schule lernen wir Algorithmen zur schriftlichen Multiplikation oder Division, zur Lösung quadratischer Gleichungen und linearer Gleichungssysteme.
- Algorithmen kann man in vielen Sprachen hinschreiben – vorzugsweise in einer Programmiersprache, denn solche sind präziser als Alltagssprachen. Es ist unerheblich ist, ob diese C heißen, Java, Python oder Scala.

Es ist unerheblich, welche Maschine von einem Algorithmus gesteuert wird. Wichtig sind die von dem Algorithmus ausgeführten Anweisungen und die dabei produzierten Ausgaben. Die Umsetzung in Steuersignale, Töne, Pixel oder andere Ausgaben ist ein technisches Problem, das hier keine Rolle spielen soll.

Schließlich gibt es Algorithmen, die zu einer vorgegebenen Eingabe(Input) eine Ausgabe(Output) berechnen und auch solche, die einmal gestartet, dauernd laufen sollen. Wir werden uns mit letzteren hier nicht beschäftigen, sondern nur mit solchen, die

- Eingaben lesen,
- dann ihre Anweisungen ausführen
- terminieren (fertig werden)
- Ausgaben erzeugen.

In diesem Kapitel werden wir Algorithmen hauptsächlich dazu benutzen, um Funktionen zu berechnen.

5.2.2 Partielle und totale berechenbare Funktionen

Berechenbare Funktionen sind solche für deren Berechnung ein Algorithmus existiert. Beispielsweise ist die Addition eine berechenbare Funktion $+ : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$. Den Algorithmus zur Berechnung der Addition lernen wir schon in der Grundschule. Das gleiche können wir auch für die anderen zahlentheoretischen Funktionen $*$, $-$, mod sagen.

Die Division ist ebenfalls berechenbar, allerdings ist ihr Definitionsbereich nur eine Teilmenge von $\mathbb{Z} \times \mathbb{Z}$, da eine Division durch 0 nicht definiert ist. Man spricht

daher von einer *partiellen Funktion*, denn ihr Definitionsbereich $\mathbb{Z} \times (\mathbb{Z} \setminus \{0\})$ ist eine echte Teilmenge von $\mathbb{Z} \times \mathbb{Z}$.

5.2.3 Definitionsbereiche

Wir schreiben $f :: A \to B$ falls f eine partielle Funktion ist. Der Definitionsbereich $dom(f) \subseteq A$ ist die Teilmenge von A für die f definiert ist. Selbstverständlich könnte man f dann als totale Funktion $f : dom(f) \to B$ bezeichnen, dies ist aber nicht üblich, da man sich gerne auf Standardbereiche für Input- und Outputwerte von Funktionen konzentriert, wie z.B. Zahlen, Strings, etc. und dann lieber von der *partiellen Funktion*

$$div :: \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$$

spricht, statt von der *Funktion*

$$div : \mathbb{Z} \times (\mathbb{Z} \setminus \{0\}) \to \mathbb{Z}.$$

Interessanter sind die Funktionen, deren Definitionsbereiche nicht so offensichtlich sind. Sei zum Beispiel $\pi pos :: \mathbb{N} \to \mathbb{N}$ die Funktion, die einer Zahl n den Index des ersten Vorkommens von n nach dem Komma in der Dezimaldarstellung von

$$\pi = 3.1415926535\ldots$$

zuordnet. Beispielsweise wäre $\pi pos(15) = 3$ und $\pi pos(265) = 6$; nicht ganz so offensichtlich ist $\pi pos(999999) = 762$. Somit sind 15, 265 und 999999 im Definitionsbereich von πpos. Was ist aber mit 777777 oder mit 100000? Einige Mathematiker vermuten, dass $dom(\pi pos) = \mathbb{N}$ ist, diese Vermutung ist aber bisher unbewiesen.

Die Funktion πpos ist *berechenbar*. Es gibt Algorithmen, die nacheinander immer mehr Dezimalstellen von π generieren können. Um $\pi pos(n)$ zu berechnen, muss man nur einen solchen Algorithmus starten und warten, bis die Ziffernfolge von n entsteht. Es ist aber möglich, dass es ein n_0 gibt, für das dies nie passieren wird. Dann würde man ewig warten, ohne je sicher zu wissen, ob noch Hoffnung besteht, oder nicht.

Der Grund, warum eine Funktion partiell ist, kann einfach bestimmbar sein, wie im Falle der Division, er kann aber auch unklar bleiben, wie im Falle der Funktion πpos. In beiden Fällen könnte man die partiellen Funktionen durch einen fiktiven Wert $\perp$ vervollständigen: Jedem Input, der nicht in $dom(f)$ ist, ordnen wir den fiktiven Wert $\perp$ zu. $f(x) = \perp$ ist dann synonym mit $x \notin dom(f)$.

Für die Funktion πpos erhielte man z.B.

$$\pi pos(n) = \begin{cases} k & k \text{ ist erster Index von } n \text{ in } \pi \\ \perp & n \text{ kommt nicht in } \pi \text{ vor.} \end{cases}$$

Anders als bei der Divisionsfunktion kann der fiktive Wert $\perp$ bei der Funktion πpos nie festgestellt oder gar ausgegeben werden, da wir nie erkennen können, dass die Berechnung der Funktion nicht terminiert.

Einen Extremfall stellt die Funktion Ω mit

$$\Omega(x) = \perp$$

dar, die überall undefiniert ist. Sie ist zweifellos berechenbar, beispielsweise durch einen Algorithmus, der eine Endlosschleife realisiert, wie etwa

„while $true$ do $x := x$“.

Definition 5.2.1. Ist $f : A \to B$ eine berechenbare Funktion, so sprechen wir von einer *totalen berechenbaren Funktion*, falls f für jeden Wert des Inputbereiches definiert ist, äquivalent, falls für alle $a \in A$ gilt: $f(a) \neq \perp$. Ansonsten heißt f *partielle berechenbare Funktion*. Fehlt das Adjektiv *total* bzw. *partiell*, so sollte aus dem Kontext hervorgehen, was gemeint ist.

Die Funktion

$$g(n) = min\{p \mid p \text{ prim und } n \leq p\}$$

ist offensichtlich berechenbar. man muss nur sukkzessive die Zahlen $n, n+1, n+2, \ldots$ testen, ob sie prim sind. Die erste so gefunde Zahl ist das Ergebnis. Bereits Euklid wusste, dass man dabei immer erfolgreich sein wird. Die Funktion

$$h(n) = min\{p \mid p \text{ prim und } (p+2) \text{ prim und } n \leq p\}$$

sollte durch eine leichte Abwandlung des Algorithmus für g ebenfalls berechenbar sein. Sie sucht oberhalb einer Zahl n nach dem ersten Primzahlzwilling, das ist ein Paar von Zahlen $(p, p+2)$ die beide prim sind, wie z.B. $(3, 5)$ oder $(17, 19)$.

Allerdings ist es unbekannt, ob es unendlich viele Primzahlzwillinge gibt, so dass die Suche nach einem Primzahlzwilling oberhalb n immer abbrechen wird. Möglicherweise bricht für ein großes n die Suche nie ab und der Algorithmus terminiert nicht. Bis heute weiss daher niemand, ob es sich bei g um eine partielle oder totale Funktion handelt.

Dass ein Algorithmus zur Berechnung existiert, heißt noch nicht notwendigerweise, dass wir ihn kennen. Es kann sein, dass man von einer Funktion beweisen kann, dass sie berechenbar ist, obwohl niemand weiß, wie die Berechnung durchgeführt werden kann. Ein ganz einfaches Beispiel ist die Funktion

$$f(n) = \begin{cases} 0 & \text{es existieren unendlich viele Primzahlzwillinge} \\ 1 & \text{sonst.} \end{cases}$$

Die Funktion f ist offensichtlich nicht von n abhängig. Es handelt sich entweder um die Funktion c_0 mit konstantem Wert 0 oder um c_1 mit konstantem Ergebnis 1. Da derzeit noch unbekannt ist, ob es unendlich viele Primzahlzwillinge gibt, wissen wir auch nicht, wie wir f berechnen können – aber f ist berechenbar, entweder durch „`return 0`“ oder durch „`return 1`“.

5.2.4 Input und Output

Funktionen von $\mathbb{R}$, oder Funktionen mit Ergebnis in $\mathbb{R}$, können wir algorithmisch nur näherungsweise berechnen. Es beginnt damit, dass die Dezimaldarstellung reeller Zahlen nicht abbrechen muß, wie z.B. bei π. Wie soll ein Algorithmus also π als Input entgegennehmen und wie soll er eine Zahl wie e ausgeben. Ausserdem könnte er nicht einmal testen, ob tatsächlich $(\pi + \pi)/2 = \pi$ gilt? Wie sollte das Ergebnis von

$$\sqrt{2} = 1.41421356237309504880168872420969807856967187537694\ldots$$

ausgegeben werden?

Daher beschränken wir uns darauf, Inputdaten und Outputdaten als endliche Zeichenfolgen über einem Alphabet zu codieren. Wir können daher davon ausgehen, dass der Input als Wort $w \in \Sigma^*$ vorliegt, für ein Eingabealphabet Σ, und dass der Output als Wort über einem Ausgabealphabet Γ beschrieben werden soll. Außerdem wollen wir davon ausgehen, dass der Output $v \in \Gamma^*$ nur von dem Input $w \in \Sigma^*$ abhängt, so dass wir einen funktionalen Zusammenhang $v = f(w)$ vorliegen haben.

Die Funktion f bezeichnet man auch als *Semantik* des Algorithmus $\mathcal{A}$ und bezeichnet sie mit $[\![\mathcal{A}]\!]$.

Definition 5.2.2. Seien Σ und Γ Alphabete. Eine *berechenbare Funktion* ist eine (ggf. partielle) Abbildung $f : \Sigma^* \to \Gamma^*$, für die es einen Algorithmus $\mathcal{A}$ gibt mit $[\![\mathcal{A}]\!] = f$.

Falls f nur auf einem Teilbereich von Σ^* definiert ist, sprechen wir von einer *partiellen berechenbaren* Funktion, ansonsten von einer *totalen berechenbaren* Funktion. Der Definitionsbereich $dom(f)$ besteht aus allen Eingabeworten $w \in \Sigma^*$ für die der Algorithmus terminiert. Gehört ein Wort $w \in \Sigma^*$ nicht zu $dom(f)$, so schreiben wir $f(w) = \perp$.

Abstrakt beschreiben Algorithmen also immer Funktionen zwischen Σ^* und Γ^*, sie nehmen als Input also ein Wort aus Σ^* und liefern ein Wort aus Γ^* ab. Für praktische Zwecke sind auch Funktionen relevant, die als Input ein Tupel oder eine Liste von Zahlen entgegenenehmen und ggf. auch ein Tupel oder eine Liste von Zahlen als Ausgabe liefern. Für diese Zwecke sollten wir zunächst untersuchen, wie wir Werte und Tupel von natürlichen, ganzen oder rationalen Zahlen in jeweils ein Wort $u \in \Sigma^*$ bzw. $v \in \Gamma^*$ codieren können.

5.2.5 Codierungen

Zunächst werden wir einige *Codierungen* angeben, das sind Algorithmen, die es uns erlauben, Informationen, die in einem bestimmten *Alphabet* formuliert sind, in ein anderes Alphabet zu übersetzen. Am Ende wird die Erkenntnis stehen, dass wir uns bei der Betrachtung berechenbarer Funktionen allein auf Funktionen von $\mathbb{N}$ nach $\mathbb{N}$ beschränken können.

Stellenwertsysteme Eine natürliche Zahl kann man im Dezimalsystem codiern, im Zweiersystem (Dualsystem) oder sogar im Einersystem über dem einelementigen Alphabet $\{|\}$. In letzterem codieren wir eine Zahl durch eine Anzahl Striche, also z.B. die Zahl 6 als „||||||". Umgekehrt repräsentiert jedes Wort $w \in \{|\}^*$ eine Zahl im Einersystem.

In allen anderen Systemen kann man sich streiten, ob führende Nullen, z.B. wie in 007, erlaubt sein sollen, oder nicht. An dieser Stelle wollen wir sie einfach ausschließen, weil viele Programmiersprachen dies ebenso halten. Die Menge aller Wörter $w \in \{0,1\}^*$, die legale natürliche Zahlen beschreiben, könnten wir dann z.B. mit dem regulären Ausdruck $r = 0+1\cdot(1+0)^*$ beschreiben und damit setzen wir $\mathbb{N}_{(2)} := [\![r]\!]$ als Menge aller Binärzahlen. Analog wollen wir es auch mit den anderen Zahlensystemen halten.

Die Umrechnung von einem System in ein anderes ist ganz einfach und kann leicht als Algorithmus, sprich Programm, codiert werden. Wir setzen an dieser Stelle die elementaren Grundrechenarten $(, +, -, /, mod)$ als gegeben (und algorithmisch beschreibbar) voraus.

$\{|\}^*$ und $\mathbb{N}_{(2)}$

Die Codierfunktion $c : \{|\}^* \to \mathbb{N}_{(2)}$ beschreiben wir rekursiv durch $c(\varepsilon) = 0$ und $c(u|) := c(u) + 1$. Hier ist mit $'+'$ die Addition von 1 zur Binärzahl $c(u)$ gemeint. Die Dekodierfunktion $d : \mathbb{N}_2 \to \{|\}^*$ beschreiben wir rekursiv durch $d(0) = \varepsilon$, $d(1) = |$ und $d(ub) := d(u) + d(u) + d(b)$. Hier ist es gleichgültig, ob wir $'+'$ als Addition oder als Stringkonkatenation lesen. Offensichtlich (siehe Aufgabe 5.2.3) gilt

$$c(d(n)) = n$$

sowie

$$d(c(w)) = w.$$

Σ^* in $\mathbb{N}$ für $|\Sigma| = r \geq 1$

Eine Möglichkeit wäre, die Zeichen aus Σ den Ziffern $0, \ldots, r-1$ zuzuordnen und dann ein Wort als Zahlendarstellung im r-System aufzufassen. Allerdings müssen wir dann das Problem der führenden Nullen lösen, denn 007 und 7, beispielsweise, sind als Elemente von $\{0, \ldots, r-1\}^*$ verschiedene Worte, haben aber den gleichen Zahlenwert.

Führt man einfach nur für die Elemente von Σ eine Reihenfolge ein, so kann man die Worte in Σ^* lexikographisch anordnen. Auch dies ergibt keine brauchbare Anordnung, da zwischen zwei verschiedenen Worten immer wieder unendlich viele Worte passen. Beispielsweise passen lexikographisch zwischen die Worte abc und abd über dem Alphabet $\Sigma = \{a, \ldots, z\}$ die unendlich vielen Worte $abca, abcaa, abcaaa, \ldots$.

Daher ordnen wir die Worte von Σ^* zuerst nach ihrer Länge und anschließend jeweils die Worte gleicher Länge alphabetisch.

Für das Alphabet $\Sigma = \{a, b, c\}$ erhält dann das Wort abc beispielsweise die Nummer $1 + 3 + 9 + 5 = 18$, denn vor ihm kommen alle Worte der Länge 0,1 und 2, sowie 5 Worte der Länge 3, nämlich aaa, aab, aac, aba, abb.

Umgekehrt kann man zu einer gegebenen Zahl n das zugehörige Wort w bestimmen, indem man von n zunächst die größte Zahl der Form

$$1 + r + r^2 + \ldots + r^k \leq n$$

subtrahiert und die Differenz im r-System darstellt. Beispielsweise erhalten wir für $\Sigma = \{a, b, c\}$ mit der üblichen alphabetischen Anordnung die 64-ste Zahl zwischen $40 = 1 + 3 + 3^2 + 3^3$ und $1 + 3 + 3^2 + 3^3 + 3^4 = 121$. Somit ist $k = 3$ und $64 - 40 = 24$, was im 3-er System die Darstellung 0220 hat. Das entspricht dem Wort $acca \in \Sigma^*$.

$\mathbb{N} \times \mathbb{N}$ in $\mathbb{N}$

Man stelle sich die Elemente von $\mathbb{N} \times \mathbb{N}$ als Gitterpunkte im ersten Quadranten der Zahlenebene vor. Jetzt durchläuft man, beginnend bei $(0, 0)$ systematisch alle Nebendiagonalen, also die Punkte (x, y) mit $x + y = k$ für steigendes k.

$(0, 0)$,
$(0, 1), (1, 0)$,
$(0, 2), (1, 1), (2, 0)$,
$(0, 3), (1, 2), (2, 1), (3, 0)$,
$(0, 4), (1, 3), (2, 2), (3, 1), (4, 0)$,
etc.

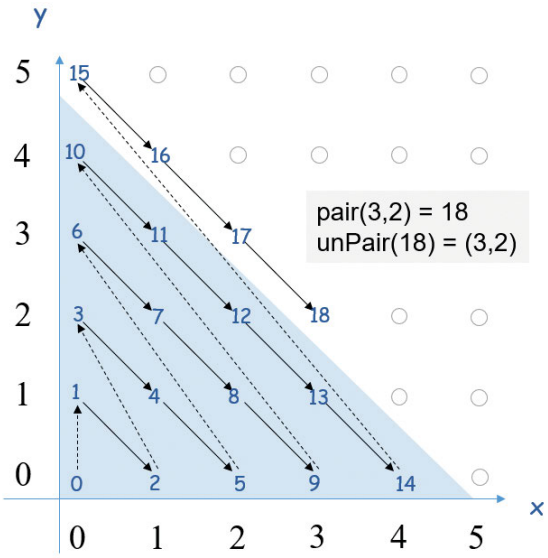


Abb. 5.2.1: Aufzählung aller Punkte in $\mathbb{N} \times \mathbb{N}$

Algorithmus 5.1 pair und unpair

```
# pair : Nat x Nat --> Nat
def pair(x,y):
 return (x+y)*(x+y+1)//2+x

#unpair : Nat -> Nat x Nat
def unpair(n):
 x, d = n, 1;
 while x >= d:
  x = x-d
  d = d+1
 y = (d-1)-x
 return (x,y)
```

Nachdem die k-te Zeile (die k-te Nebendiagonale in Figur 5.2.1) gefüllt ist, hat man insgesamt $1+2+\ldots+k$ Punkte verarbeitet. Dies ist k-te *Dreieckszahl*: $D(k) = k\star(k+1)/2$.

- Für die Codierung von $pair : \mathbb{N}\times\mathbb{N} \to \mathbb{N}$ hat man nun die einfache Formel $pair(x,y) = D(x+y)+x = (x+y)\star(x+y+1)/2+x$.
- Für die Dekodierung $unpair(n) = (x,y)$ muss man von n die größte Dreieckszahl mit $D(k) \le n$ subtrahieren. Dazu braucht man nur der Reihe nach $k = 1, 2, \ldots$ von n zu subtrahieren, solange der Rest positiv bleibt. Mit $x = n - D(k) = n-1-2-3-\ldots-k$ hat man somit schon die x-Koordinate. Weil (x,y) auf der Nebendiagonale $x+y=k$ liegen muss, folgt: $y = k-x$.

$\mathbb{N}^r$ in $\mathbb{N}$

Eine zugehörige Codierung c_r mit Decodierung d_r kann man durch Iteration der vorigen Codierungen $c_2 = pair$ und $d_2 = unpair$ erreichen. Für $r \ge 3$ also

- $c_r(x_1,\ldots,x_r) := c_2(x_1, c_{r-1}(x_2,\ldots,x_r))$.
- $d_r(n) = (x_1, x_2, \ldots, x_r)$ wobei $(x_1,u) = d_2(n)$ und $(x_2,\ldots,x_r) = d_{r-1}(u)$.

$\mathbb{N}^+$ in $\mathbb{N}$

Die Elemente von $\mathbb{N}^+$ kann man als nichtleere Listen natürlicher Zahlen darstellen. Zu gegebener Länge k sind die Listen der Länge k nichts anderes als die k-Tupel von Elementen von $\mathbb{N}$.

Mit anderen Worten ist $\mathbb{N}^+$ die disjunkte Vereinigung aller $\mathbb{N}^k$, in mathematischer Schreibweise

$$\mathbb{N}^\star = \biguplus_{k>0} \mathbb{N}^k = \{(k,\sigma) \mid 1 \le k \in \mathbb{N}, \sigma \in \mathbb{N}^k\}.$$

Diese Notation legt sofort folgende Codierung nahe:

$$c^+(w) = c_2(k, c_k(w))$$

mit $k = |w|$. Für die Dekodierung erhalten wir $d^+(n) = d_k(y)$ wobei $(k, y) = d_2(n)$.

$\mathbb{N}^\star$ in $\mathbb{N}$

Zu $\mathbb{N}^+$ kommt nur das leere Wort hinzu. Um Platz zu machen, schieben wir wie in *Hilbert's Hotel*[3] jeden Gast ein Zimmer weiter und platzieren ε in das 0-te Zimmer:

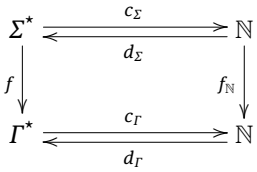
$c^\star(\varepsilon) := 0$ und $c^\star(w) = c^+(w) + 1$ für $w \neq \varepsilon$, und $d^\star(0) = \varepsilon$ sowie $d^\star(n) = d^+(n-1)$ für $n > 0$.

Elemente aus $\mathbb{N}^\star$ kann man auch folgendermaßen in $\mathbb{N}$ codieren: Zu einer gegebene Folge $\sigma = (n_1, n_2, \ldots, n_k)$ konkateniert man $2^{n_1}, \ldots, 2^{n_k}$ zu einer Binärzahl.
Für die Umkehrfunktion startet man mit einer Binärzahl b und rekonstruiert die Liste von Zahlen, aus der sie hervorgegangen sind. Die Anzahl der $1 - en$ codiert die Länge der Liste und jede maximale nebeneinanderliegende Gruppe von 0-en entspricht einer Zahl n_i. Die Liste $(3, 4, 1, 0, 2)$ wird durch 100010000101100 repräsentiert. Die leere Liste entspricht der Zahl 0.

Fazit

Aufgrund der zahlreichen Codierungen, die zudem noch durch einfache Algorithmen beschreibbar sind, können wir jede berechenbare Funktion zwischen $\Sigma^\star$,$\mathbb{N}^+$, $\mathbb{N}^\star$, $\mathbb{N}^k$ und $\mathbb{N}$ auf eine berechenbare Funktion von $\mathbb{N}$ nach $\mathbb{N}$ zurückspielen. Beispielsweise könnten wir jede gewünschte Funktion $f : \Sigma^\star \to \Gamma^\star$ zurückspielen auf $f = d_\Gamma \circ f_\mathbb{N} \circ c_\Sigma$ mit

- einer Codierfunktion $c_\Sigma : \Sigma^\star \to \mathbb{N}$
- einer partiellen berechenbaren Funktion $f_\mathbb{N} : \mathbb{N} \to \mathbb{N}$
- einer Dekodierfunktion $d_\Gamma : \mathbb{N} \to \Gamma^\star$



Aufgabe 5.2.3. Codierung und Decodierung

1. Zeigen Sie induktiv, dass für alle $w \in \{|\}^\star$ gilt, dass $d(c(w)) = w$.
2. Zeigen Sie induktiv $c(d(n)) = n$ für alle $n \in \mathbb{N}_2$.
3. Zeigen Sie, dass $f(x, y) := (2 \star x + 1) \star 2^y$ eine bijektive Codierung von $\mathbb{N} \times \mathbb{N}$ nach $\mathbb{N}$ liefert. Wie berechnet man die inverse Funktion f^{-1}?

3 siehe z.B. https://de.wikipedia.org/wiki/Hilberts_Hotel

5.2.6 Totale und partielle berechenbare Funktionen

Eine Funktion $f : A \to B$ heißt *partiell*, wenn sie nicht für alle Werte der Ausgangsmenge definiert ist. In der Theorie der Berechenbarkeit kann man solche Situationen leider nicht ausschließen. Dies liegt daran, dass jedes vernünftige Algorithmenkonzept auch die Möglichkeit nichtterminierender Berechnungen einschließt. Im Falle von WHILE-Programmen werden diese von Endlosschleifen verursacht. Manchmal sind solche einfach zu erkennen, oft können sie auch sehr versteckt sein. So ist es beispielsweise bis heute unbekannt, für welche Eingaben das folgende Programm jemals zu einem Ende kommt. Es handelt sich natürlich um eine Formulierung des bisher ungelösten Collatz-Problems:

Algorithmus 5.2 Collatz

```
def collatz(n):
 x = n
 while not(x==1):
  if x % 2==0:    ## % = mod
   x = x // 2   ## // = div
  else:
   x = 3*x+1
 return(x)
```

Die bisher unbewiesene Vermutung besagt, dass dieses Programm für jeden Input $n \in \mathbb{N}$ terminiert und als Ergebnis den Wert 1 liefert. Es ist aber auch denkbar, dass es große Zahlen n gibt, für die das Programm nie terminiert.

Für andere Programme ist die Situation klarer, z.B. berechnet das folgende Programm den ganzzahligen Zweierlogarithmus $\lfloor log_2(n) \rfloor$ einer Zahl $n \in \mathbb{N}$ – allerdings nur, falls es terminiert. Es terminiert nur für diejenigen n für die $\lfloor log_2(n) \rfloor$ gerade ist. Für alle anderen n terminiert es nicht, was sich praktisch in einem Stacküberlauf äußert.

Algorithmus 5.3 $\lfloor log_2 \rfloor$

```
def ilog2(n):
 if n==1 :
  return 0
 else:
  return 2+ilog2(n//4)
```

Wir wollen in diesem Kapitel noch gewisse Idealisierungen vornehmen, insofern als es

- keine Begrenzung des Speicherplatzes und
- keine Begrenzung von Heap und Stack

geben soll, insbesondere findet dann kein Stacküberlauf mehr statt. Stattdessen arbeitet das Programm ohne zu stoppen weiter. Inputs, für die diese Situation eintritt, gehören nicht zum Definitionsbereich der Funktion, die das Programm berechnet. Wir haben also eine *partielle* berechenbare Funktion vor uns. Wir können dann annehmen, dass $f(n) = \perp$ gleichbedeutend damit ist, dass f mit Input n nicht terminiert.

5.2.7 Die meisten Funktionen sind nicht berechenbar.

Zunächst wollen wir abschätzen, wieviele berechenbare Funktionen es überhaupt geben kann. Für die Menge aller Funktionen einer Menge M in die Menge N schreibt man N^M und für die Menge $\{0, 1\}$ schreibt man kurz $\mathbf{2}$. Damit ist $\mathbf{2}^{\mathbb{N}}$ die Menge aller Abbildungen von $\mathbb{N}$ in die zweielementige Menge.

$\mathbf{2}^{\mathbb{N}}$ können wir mit der Potenzmenge $\mathbb{P}(\mathbb{N})$ identifizieren, denn einerseits liefert jede Abbildung $f : \mathbb{N} \to \{0, 1\}$ eine Teilmenge

$$f^{-}(1) := \{n \in \mathbb{N} \mid f(n) = 1\}$$

und andererseits entsteht jede Teilmenge $U \subseteq \mathbb{N}$ auf diese Weise aus ihrer charakteristischen Funktion $\chi_U : \mathbb{N} \to \{0, 1\}$ mit

$$\chi_U(n) := \begin{cases} 1 & \text{falls } n \in U \\ 0 & \text{sonst.} \end{cases}$$

Insbesondere gibt es also genauso viele Abbildungen von $\mathbb{N}$ in die zweielementige Menge wie es Teilmengen von $\mathbb{N}$ gibt, also

$$|\mathbf{2}^{\mathbb{N}}| = |\mathbb{P}(\mathbb{N})|.$$

Zu jeder berechenbaren Funktion muss es aber einen Algorithmus geben und jeden Algorithmus kann man z.B. als Java-Programm formulieren. Der Programmtext jedes einzelnen Programms ist damit ein Wort $w \in \Sigma^*$. Wegen der bijektiven Korrespondenz von Σ^* mit $\mathbb{N}$ gilt insbesondere $|\Sigma^*| \leq |\mathbb{N}|$, es gibt also höchstens abzählbar viele Algorithmen und daher höchstens abzählbar viele berechenbare Funktionen.

Andererseits gibt es überabzählbar viele Teilmengen von $\mathbb{N}$ und damit erst recht überabzählbar viele Abbildungen $g : \mathbb{N} \to \mathbb{N}$, kurz

$$|\Sigma^*| = |\mathbb{N}| < |\mathbb{P}(\mathbb{N})| = |2^{\mathbb{N}}| \leq |\mathbb{N}^{\mathbb{N}}|.$$

Daraus folgt:

Theorem 5.2.4. *Es gibt Funktionen $f : \mathbb{N} \to \mathbb{N}$, die nicht berechenbar sind.*

Die Anzahl $|\mathbb{N}|$ ist so viel kleiner als $|\mathbb{N}^{\mathbb{N}}|$, dass man sogar sagen kann, dass „die meisten“ Funktionen von $\mathbb{N} \to \mathbb{N}$ nicht berechenbar sind.

Das klingt schlimmer als es ist. Diejenigen Funktionen, welche nicht berechenbar sind, haben den Nachteil, dass man sie auch nicht konstruktiv angeben kann. Sie verdanken ihre Existenz Axiomen wie z.B. dem Auswahlaxiom, welches die Existenz von gewissen Funktionen fordert, diese aber nicht konkret anzugeben in der Lage ist.

Es gibt durchaus Mathematiker und Informatiker, die vorschlagen, auf das Auswahlaxiom zu verzichten und eine entsprechende konstruktive Mathematik betreiben. Gegenwärtig ist dies unter Mathematikern noch eine kleine Minderheit, unter Informatikern ist diese Denkrichtung verständlicherweise stärker verbreitet.

Die Existenz nicht berechenbarer Funktionen wird im Endeffekt auf einen Widerspruchsbeweis – im Satz von Cantor – zurückgeführt. Kann man aber *konkret* eine nicht berechenbare Funktion vorzeigen? Wir suchen also eine Funktion $f : \mathbb{N} \to \mathbb{N}$ sowie einen Beweis, das es keinen Algorithmus gibt, der f berechnet. Dies kann aber nur gelingen, wenn wir vorher den Begriff „Algorithmus“ mathematisch präzise fassen.

5.2.8 Algorithmenbegriff und Churchsche These

Der Begriff der berechenbaren Funktionen erfordert den Begriff des Algorithmus. Dessen Definition ist bisher allerdings noch ziemlich vage geblieben. Benutzt haben wir bisher nur, dass jeder Algorithmus hingeschrieben werden kann und damit durch ein Wort w einer Sprache $L \subseteq \Sigma^*$ repräsentiert wird. Aus Anzahlgründen folgt daraus schon die Existenz nicht-berechenbarer Funktionen.

Wollen wir aber von einer konkreten Funktion zeigen, dass sie nicht berechenbar ist, dass also kein Algorithmus existieren kann, der sie berechnet, so müssen wir zuvor eine mathematisch saubere Definition des Begriffes *Algorithmus* zugrunde legen.

Für eine präzise Definition müssen wir sowohl die „Hardware“ beschreiben als auch die „Aktionen“, die diese ausführen kann, sowie die vorhandenen *Kontrollstrukturen*, mit denen man festlegen kann, wann und unter welchen Bedingungen eine bestimmte Aktion ausgeführt werden soll.

Streit scheint vorprogrammiert, da jeder seine eigene Hardware und seine Lieblings-Programmiersprache als Grundlage einer mathematischen Definition von „Algorithmus“ sehen möchte. In der Tat hat es viele und sehr unterschiedliche Vorschläge zur präzisen Definition des Begriffes „Algorithmus“ gegeben. Eine Möglichkeit, die verschiedenen Konzepte zu vergleichen, besteht darin, die Menge aller berechenbaren Funktionen zu bestimmen, die mit einem der vorgeschlagenen Algorithmenkonzepte beschrieben werden können.

Überraschenderweise hat sich herausgestellt, dass man stets auf die gleiche Menge berechenbarer Funktionen gestoßen ist. Diese Erfahrung mündete in die Überzeugung, die von Stephen Kleene seinem Lehrer Alonzo Church zugeschrieben wird die heute als Churchsche These bezeichnet wird:

Churchsche These: *Jede (vernünftige) Präzisierung des Begriffes Algorithmus führt auf die gleiche Menge partieller berechenbarer Funktionen.*

Offensichtlich kommt es nicht auf die Marke des eingesetzen Rechners oder dessen Betriebssystem an, allein schon, weil wir Algorithmen in einer höheren Programmiersprache formulieren. Außerdem wissen wir, dass im Kern jede CPU auf einem Satz einfacher arithmetischer und boolescher Operationen aufbaut, in die auch Java-Programme im Endeffekt übersetzt werden. In jedem Rechnermodell ist es daher einfach, ein Programm zu schreiben, das die CPU der anderen Marke simuliert, so dass prinzipiell Mac, PC, Raspberry-Pi und der teuerste Supercomputer die gleichen Funktionen berechnen können.

Somit erscheint uns die Churchsche These einleuchtend, solange wir uns auf handelsübliche Rechner und Programmiersprachen beziehen. Es gibt aber von der Denkweise der gängigen Programmiersprachen abweichende – und trotzdem nicht weniger interessante Ideen, den Begriff des Algorithmus zu fassen. Einige davon haben sogar zu völlig neuartigen Programmiersprachenkonzepten geführt, ein bekanntes Beispiel ist die Sprache *Prolog*. Ein weiteres Konzept entsteht aus dem Church'schen *Lambda-Kalkül*, der die Grundlage funktionaler Programmiersprachen bildet und dessen Konzepte immer stärker in gängigen Programmiersprachen (Scala, C#, ...) Einzug halten.

Schließlich eröffnet die Churchsche These die Möglichkeit, minimale Kernkonzepte bestehender Sprachen und deren Äquivalenzen herauszuarbeiten. Dies wollen wir im Folgenden tun, wenn wir einige der bekanntesten Algorithmenkonzepte vorstellen.

5.3 Turing-Berechenbarkeit

5.3.1 Turings Analyse

Wir werden nun eine Maschine definieren, die nach ihrem Erfinder, Alan Turing, „Turingmaschine“ genannt wird. Obwohl man sie konstruieren könnte, manche Zeitgenossen haben sich sogar den Spaß gemacht (siehe *„A Turing Machine - Subtraction“*, *www.youtube.com/watch?v=E3keLeMwfHY*), dient sie als mathematisches Konstrukt zur Begriffsklärung. Insbesondere wird sich herausstellen, dass eine Turingmaschine prinzipiell jede partielle berechenbare Funktion berechnen kann, also jede Funktion die auch von einem modernen Computer berechnet werden kann.

Wir stellen uns zunächst vor, dass ein Mensch mit Hilfe von kariertem Papier und einem Bleistift einen Algorithmus ausführt, beispielsweise zwei lange dezimal geschriebene Zahlen $(y_m \ldots y_0)_{10}$ und $(x_n \ldots x_0)_{10}$ multipliziert.

Dazu schreibt er zunächst die Ziffern an bestimmte Stellen in dem kariertem Papier, als nächstes führt er elementare Operationen (Multiplikation von Ziffern, Additi-

on von Ziffern) aus und schreibt die Ergebnisse wieder an bestimmte andere Positionen.

In der dritten Berechnungsphase werden die entstandenen untereinender stehenden Zahlen addiert.

Die Multiplikation von Ziffern könnten beispielsweise aus den Tabellen des *Kleinen Einmaleins* abgelesen werden, entsprechend könnte man auch die Addition zweier Ziffern aus einer Additionstabelle ablesen.

Die Berechnung erfolgt in einzelnen Schritten und der jeweils nächste Schritt ist von dem aktuellen Zustand, also der Phase der Berechnung, abhängig, aber auch von dem Zeichen, was gerade gelesen wurde.

Wir wollen dies in einem möglichst einfachen mathematischen Maschinenmodell erfassen und anschließend die Addition zweier Binärzahlen detailliert beschreiben. Als Hardware benötigen wir zunächst kariertes Papier und einen Schreibstift, mit dem wir eine Ziffer in ein Kästchen schreiben können. Zusätzlich muss der Schreibstift sich von Kästchen zu Kästchen bewegen können. Außerdem müssen wir verschiedene Phasen unterscheiden:

Q_1 : Hinschreiben der Multiplikanden, getrennt durch ein '*'
Q_2 : Für jede Ziffer x_i der zweiten Zahl: multipliziere die erste Zahl mit x_i
Q_3 : Addition der versetzt geschriebenen Ergebnisse aus Phase Q_3

Eventuell muss man die Phasen $Q_1, \ldots, Q_3$ noch einmal in kleinere Aufgaben zerlegen, die man mit den Operationen

- Stift bewegen,
- Inhalt des Kästchens lesen,
- Ziffer schreiben

spezifizieren kann. Die „Phasen“ werden wir später *Zustände* nennen.

5.3.2 Turingmaschinen

Wir wollen ein möglichst einfaches Maschinenmodell zum Ausgangspunkt nehmen. Zunächst könnten wir das karierte Blatt Papier auch durch einen langen Streifen ersetzen, in dem die Kästchen nur nebeneinanderliegen. Statt eine Zeile nach unten oder oben auf dem Originalblatt müssten wir auf dem Streifen dann 80 Kästchen nach rechts bzw. nach links gehen.

Damit uns der Platz niemals ausgeht, wollen wir der Einfachheit halber annehmen, dass unser Streifen rechts und links unbegrenzt ist, oder – was auf das gleiche hinausläuft — dass wir jederzeit, wenn nötig, rechts oder links noch ein Stück ankleben dürfen.

Auf jedem der Kästchen kann ein Zeichen stehen, oder es kann leer sein. Es gibt immer ein aktuelles Kästchen, auf dem unser Focus liegt. Früher sprach man von ei-

nem „Schreib-Lesekopf“, heute würde man dazu „cursor“ sagen, der immer auf genau auf einer Zelle der Bandes steht.

Auf dem Band können sich endlich viele Zeichen aus einem Alphabet Σ befinden. Die restlichen Positionen des Bandes sind leer. Um das Leerzeichen sichtbar zu machen verwenden wir $'\#'$ als Metasymbol und setzen $'\#' \notin \Sigma$ voraus. Die Maschine wird daher formal mit Zeichen des erweiterten *Bandalphabets* $\Gamma = \Sigma \uplus \{'\#'\}$ umgehen müssen.

Ein *Schritt* der Maschine besteht darin, in Abhhängigkeit von dem aktuellen Zustand q und dem Zeichen $x \in \Gamma$ unter dem Cursor

- das gelesenen Zeichen x mit einem neuen Zeichen x' zu überschreiben
- den cursor ein Kästchen nach rechts (R) oder nach links (L) zu bewegen
- in den neuen Zustand q' zu wechseln.

Legen wir uns auf ein endliches Bandalphabet $\Gamma = \Sigma \uplus \{'\#'\}$ und eine endliche Zustandsmenge Q fest, so können wir die möglichen Aktionen der Turingmaschine in einer *Turingtabelle* δ codieren. Deren Zeilen entsprechen den möglichen Zuständen, und die Spalten den gelesenen Zeichen. Der Eintrag der Tabelle in Zeile q und Spalte x ist entweder ein Tripel

$$\delta(q, x) = (x', q', d)$$

mit $x' \in \Gamma$, $q' \in Q$ oder er bleibt leer. In letzterem Fall betrachten wir die Tabelle an der Position (x, q) als undefiniert und notieren dies auch als $\delta(q, x) = \perp$.

Wir können die Turing-Tabelle als partielle Funktion

$$\delta :: Q \times \Gamma \to \Gamma \times Q \times \{R, L\}$$

modellieren. Befindet die Maschine sich in Zustand q mit dem Cursor auf einem Feld mit dem Zeichen $x \in \Gamma$, so wird die Aktion $\delta(q, x)$ ausgeführt, die in Zeile q und Spalte x der Tabelle steht.

So gelangen wir zu der offiziellen Definition:

Definition 5.3.1. Sei Σ ein endliches Alphabet, $'\#' \notin \Sigma$ und $\Gamma = \Sigma \uplus \{'\#'\}$. Sei Q eine endliche Menge von Zuständen und $q_0 \in Q$.

Eine *Turingtabelle* ist eine partielle Abbildung

$$\delta :: Q \times \Sigma \to \Sigma \times Q \times \{L, R\},$$

also eine Tabelle mit $|Q|$ Zeilen und $|\Sigma|$ Spalten, die an der Position (q, x) den Wert $\delta(q, x)$ enthält, falls (q, s) aus dem Definitionsbereich von δ ist.

Ein *Berechnungsschritt*, oder *Transition*, einer Turingmaschine, die in Zustamd $q \in Q$ ist und deren Cursor sich auf einer Bandposition mit Inhalt $x \in \Gamma$ befindet, besteht darin, die Instruktion auszuführen, die in der Tabelle an der Position $\delta(q, x)$ steht. Ist dieses Feld leer, so stoppt die Maschine. Ansonsten sei

$$\delta(q, x) = (x', q', d)$$

mit $x' \in \Gamma$, $q' \in Q$, $d \in \{L, R\}$. Diese Instruktion weist die Maschine an:

– überschreibe das Zeichen an der Cursorposition mit x'
– falls $d = R$, setze Cursor eine Position nach rechts, falls $d = L$, nach links
– gehe in den neuen Zustand q'.

Eine *Berechnung* startet im Anfangzustand $q_0 \in Q$ und führt Berechnungsschritte aus, bis die Maschine in Zustand q das Zeichen x liest und der Tabelleneintrag $\delta(q, x)$ nicht vorhanden ist.

Es ist nicht sicher, dass die Maschine jemals stoppt, sie kann auch endlos weiterlaufen, wenn sie nie einen leeren Tabelleneintrag antrifft.

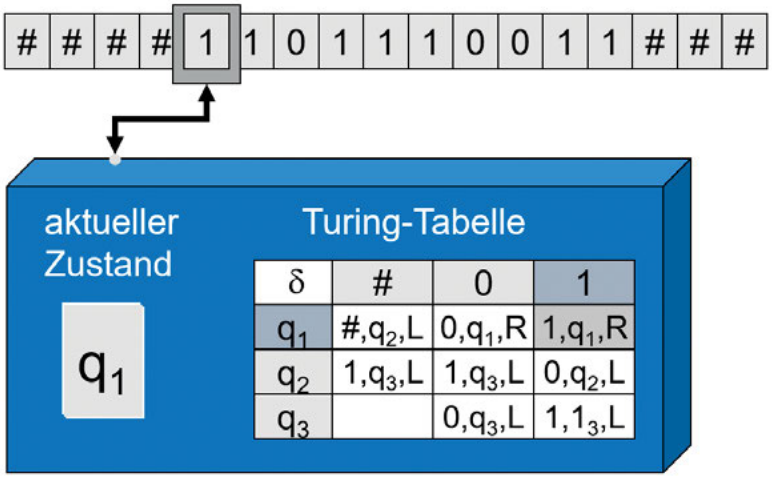


Abb. 5.3.1: Turingmaschine

Figur 5.3.1 zeigt eine Turingmaschine mit den Zuständen $Q = \{q_1, q_2, q_3\}$ und dem Alphabet $\Gamma = \Sigma \uplus \{'\#\}'$ wobei $\Sigma = \{0, 1\}$. Die Turing Tabelle δ enthält keinen Eintrag in Zeile q_3 mit Spalte $'\#'$.

Im aktuellen Zustand q_1 steht auf dem Band das Wort $1101110011 \in \Sigma^\star = \{0, 1\}^\star$, der Rest des Bandes enthält nur Leerzeichen '#'. Der Cursor befindet sich auf dem ersten Zeichen von 1101110011, einer $'1'$. Als nächstes wird daher die Instruktion $\delta(q_1, 1) = (1, q_1, R)$ ausgeführt. Dies bedeutet, dass das Zeichen unter dem Cursor unverändert bleibt (eine '1' wurde gelesen und eine '1' wird geschrieben) ebenso ist

der neue Zustand gleich dem alten Zustand, nur der Cursor wandert um eins nach rechts. Dies wiederholt sich auch im folgenden Schritt.

Nach einem weiteren Schritt ist sie immer noch in Zustand q_1 und erreicht eine Zelle mit Inhalt ´0´. Auch deren Inhalt sowie der Zustand q_1 bleiben unverändert, bis der Cursor schließlich am rechten Ende der Binärzahl auf ein Leerzeichen '#' trifft.

Wegen $\delta(q_1, \#) = (\#, q_2, L)$ bleibt auch dieses unverändert, allerdings wechselt die Maschine jetzt in den Zustand q_2 und geht wieder einen Schritt nach links. Insgesamt kann man die erste Zeile der Tabelle, und damit den Zustand q_1 folgendermaßen charakterisieren:

q_1 : bewege den Cursor an das Ende der Binärzahl und gehe in Zustand q_2.

Im neuen Zustand q_2 bewegt sich der Cursor nach links. Dabei ändert sie jede '1' in eine '0'. Sobald sie zum ersten Mal eine '0' oder ein Leerzeichen '#' antrifft, ändert sie dieses in eine '1' und geht in Zustand q_3, kurz:

q_2 : scanne die Zahl von rechts nach links und wandle jede ´1´ in eine ´0´ um. Sobald eine '0' oder ein '#' angetroffen wird ersetze das Zeichen durch ´1´ und gehe in den Zustand q_3.

Im Zustand q_3 ist das Ergebnis eigentlich schon fertig. Der Cursor läuft nur noch nach links, um sich vor dem Ergebnis zu positionieren. Sobald er dort steht, hält die Maschine, weil $\delta(q_3, ´\#´)$ undefiniert ist:

q_3 : positioniere den Cursor vor dem Ergebnis und stoppe.

Sobald die Maschine hält, ist aus der Binärzahl 1101110011 die Binärzahl 1101110100 entstanden. Man überzeugt sich leicht, dass auch für jede ander Binärzahl die Maschine 1 addiert und dann stoppt. Daher kann man sagen, dass die Maschine die Funktion $suc : \mathbb{N} \to \mathbb{N}$ mit $succ(n) = n + 1$ berechnet.

5.3.3 Simulatoren

Selbstverständlich ist es leicht, eine Turingmaschine durch ein Programm zu simulieren. Im Internet wimmelt es geradezu von Simulatoren für Turingmaschinen. Eine besonders gelungene Umsetzung stammt von Gregor Buchholz. Sie ist in der folgenden Figur dargestellt und wird z.B. auf *www.joefox.de/turing/* oder auf *www.info-wsf.de/* zum freien Download angeboten. Der folgende Screenshot zeigt, dass man offenbar ohne weiteres mit dem Programm umgehen kann.

Dargestellt wird der Simulator mit einem Programm, welches zwei Binärzahlen addieren kann. Rechts erkennt man die Turing-Tabelle, deren Zeilen $z_{01}, z_{02}, \ldots, z_{06}$ den Zuständen $Q = \{z_{01}, z_{02}, \ldots, z_{06}\}$ entsprechen. Das Datenalphabet ist $\Sigma = \{0, 1\}$, somit ist $\Sigma \cup \{´\#´\} = \Gamma = \{\#, 0, 1\}$ das Bandalphabet, was den Spalten der Tabelle ent-

spricht. z_{01} ist der Anfangszustand. Die Tabelle ist immer definiert, außer für $(z_{03}, \#)$, dies ist als eine Endkonfiguration definiert.

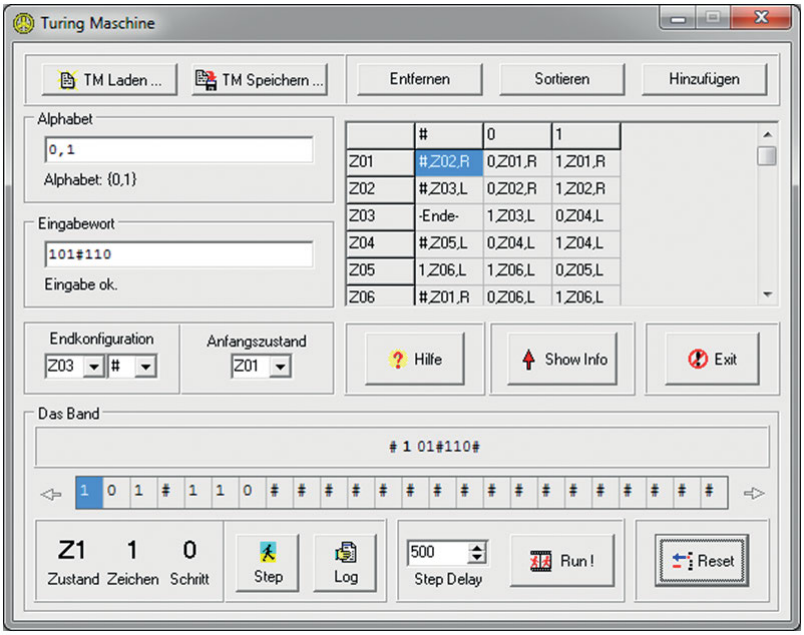


Abb. 5.3.2: Turing-Simulator

Wir behaupten, dass die Turingmaschine mit der gezeigten Turingtabelle zwei Binärzahlen addieren kann. Dazu schreibt man die beiden Summanden durch ein Leerzeichen '#' getrennt auf das ansonsten leere Band und setzt den Cursor auf das erste Zeichen des linken Summanden. Die Zustände $\{z_{01}, \ldots, z_{06}\}$ kann man so beschreiben:

z_{01} : Laufe nach rechts zum Beginn des zweiten Summanden
z_{02} : weiter nach rechts bis zur letzten Ziffer
z_{03} : subtrahiere 1 von der rechten Zahl
z_{04} : laufe nach links zum ersten Summanden
z_{05} : addiere 1 zur linken Zahl
z_{06} : weiter zum Anfang der linken Zahl und beginne von vorne.

Das Programm hält, wenn in Zustand z_{03} keine '1' in der zweiten Zahl mehr angetroffen wurde bevor das Trennzeichen '#' zwischen den Zahlen erreicht wird.

Es ist kein Zufall, dass die Darstellung der Zustände in der obigen Turingmaschine an Zeilennummern in einem Programm erinnern.

In der Literatur versteht man unter dem Begriff *Turingmaschine* meist die gedachte Hardware zusammen mit der Tabelle δ. Wir werden den Begriff *Turingmaschine* für die gedachte Hardware verwenden und *Turing-Tabelle* oder *Turingprogramm* für die Software δ.

5.3.4 Konfigurationen

Unsere bisherigen Erläuterung der Turingmaschine waren eher technischer Natur schließlich handelt es sich um eine – wenn auch gedachte – Hardware. In diesem Kapitel werden wir die entsprechenden mathematischen Begriffe einführen. Dies hat nicht nur den Vorteil, dass wir mit den bewährten Methoden mathematische Beweise führen können, sondern auch, dass es uns wertvolle Hinweise für die elegante Implementierung eines Simulationsprogramms gibt.

Definition 5.3.2. Eine *Turingmaschine* ist ein Tripel $TM = (Q, \Gamma, q_0, E, \delta)$ bestehend aus einem endlichen Alphabet Γ mit $'\#' \in \Gamma$, einer endlichen Menge Q von Zuständen mit Anfangszustand $q_0 \in Q$ und einer Menge $E \subseteq Q$ von Endzuständen, sowie einer Turingtabelle

$$\delta : Q \times \Gamma \to \Gamma \times Q \times \{L, R\}.$$

Eine *Konfiguration* ist ein Tripel (α, q, β) mit $\alpha, \beta \in \Gamma^*$ und $q \in Q$.

Eine Konfiguration (α, q, β) soll die Situation der Turingmaschine während einer Berechnung vollständig beschreiben: α und β beschreiben den Bandinhalt links und rechts vom Cursor, wobei wir willkürlich festlegen, dass das Zeichen unter dem Cursor β zugeschlagen werden soll und der Rest des Bandes mit Leerzeichen '#' gefüllt ist. α und β müssen zusammen mindestens den Bereich des Bandes abdecken, in dem von '#' verschiedene Zeichen vorkommen. q ist der aktuelle Zustand.

Wollten wir den Lauf einer Turingmaschine unterbrechen, müssten wir lediglich ihre Turingtabelle und ihre aktuelle Konfiguration speichern und könnten sie später allein mit dieser Information weiterlaufen lassen, als sei nichts geschehen.

Definition 5.3.3. Die *Übergangsrelation* „$\vdash$“ beschreibt formal den Übergang von einer Konfiguration (α, q, β) zu einer Folgekonfiguration (α', q', β'). Für $\alpha = a_1 \ldots a_m$ und $\beta = b_1 b_2 \ldots b_n$ definieren wir:

$$(a_1 \ldots a_m, q, b_1 \ldots b_n) \vdash \begin{cases} (a_1 \ldots a_m \boldsymbol{c}, q', b_2 \ldots b_n) & \text{falls } \delta(q, b_1) = (\boldsymbol{c}, q', R) \\ (a_1 \ldots a_{m-1}, q', a_m \, \boldsymbol{c} \, b_2 \ldots b_n) & \text{falls } \delta(q, b_1) = (\boldsymbol{c}, q', L) \end{cases}.$$

Falls $\alpha = \varepsilon$ oder $\beta = \varepsilon$ ist, wird es durch $'\#'$ ersetzt.

Eine Konfiguration heißt *Endkonfiguration*, falls im aktuellen Zustand q mit dem aktuellen Zeichen x unter dem Cursor der entsprechende Tabelleneintrag leer ist, falls also $\delta(q, x) = \perp$ ist. Wenn eine Endkonfiguration erreicht ist, stoppt die Maschine.

Mit $\vdash^*$ bezeichnen wir die reflexiv-transitive Hülle von $\vdash$. Dies bedeutet, dass $\sigma_0 \vdash^* \sigma_n$ für Konfigurationen σ_0 und σ_n gilt, falls es Zwischenkonfigurationen $\sigma_1, \ldots, \sigma_{n-1}$ gibt mit

$$\sigma_0 \vdash \sigma_1 \vdash \ldots \vdash \sigma_{n-1} \vdash \sigma_n.$$

Die Maschine beginnt also mit der Konfiguration σ_1 und gelangt über $\sigma_2, \ldots, \sigma_{n-1}$ schließlich zur Konfiguration σ_n. Falls σ_n eine *Endkonfiguration* ist, also $\sigma_n = (\alpha, q_e, x\beta)$ mit $\delta(q_e, x) = \bot$, hält die Maschine.

5.3.5 Berechnungen

Jetzt können wir Turingmaschinen auch benutzen, um Funktionen $f : \Sigma^* \to \Sigma^*$ zu berechnen, wobei wir natürlich $'\#' \notin \Sigma$ voraussetzen: Um $f(u)$ zu berechnen, schreiben wir u auf das ansonsten leere Band, setzen den Cursor auf das erste Zeichen von u, starten die Maschine im Anfangszustand q_0 und warten bis wir eine Endkonfiguration erreicht haben. Das Ergebnis ist das Wort $w \in \Sigma^*$, welches mit dem Zeichen unter dem Cursor beginnt und bei dem nächsten '#' endet.

Falls für ein Argument u die Berechnung die Maschine nicht endet, weil sie nie eine Endkonfiguration erreicht, so ist $f(u)$ nicht definiert. Auf diese Weise können Turingmaschinen auch partielle Funktionen $f :: \Sigma^* \to \Sigma^*$ berechnen. Der Definitionsbereich $dom(f)$ besteht einfach aus allen $u \in \Sigma$, für die die Berechnung nach endlicher Zeit endet. Formal lautet die Definition:

Definition 5.3.4. Eine partielle Abbildung $f :: \Sigma^* \to \Sigma^*$ heißt *Turing-berechenbar*, falls es eine Turingmaschine gibt, so dass für alle $u \in \Sigma^*$ gilt:

$$u \in dom(f) \Leftrightarrow (\varepsilon, q_0, u) \vdash^* (\varepsilon, q_f, f(u)) \text{ ist Endkonfiguration.}$$

Um $f(u)$ zu berechnen, schreiben wir also u auf das ansonsten leere Band, setzen den Cursor auf das erste Zeichen von u, starten im Anfangszustand q_0 und warten bis wir eine Endkonfiguration erreicht haben. Das Ergebnis ist das Wort $w \in \Sigma^*$, welches mit dem Zeichen unter dem Cursor beginnt und vor dem ersten '#' endet.

5.3.6 n-stellige Funktionen

Mit Turingmaschinen können wir selbstverständlich auch mehrstellige (partielle) Funktionen $f :: \mathbb{N}^k \to \mathbb{N}$ berechnen. Dazu können wir beispielsweise das binäre Alphabet $\Sigma = \{0, 1\}$ im Bandalphabet $\Gamma = \{0, 1, \#\}$ verwenden und die Zahlen $n \in \mathbb{N}$ binär codiert auf das Band schreiben. Ein Eingabetupel $(n_1, \ldots, n_k)$ schreiben wir als Folge der Binärcodierungen, getrennt durch jeweils ein Leerzeichen '#' auf das Band:

$$\#bin(n_1)\#bin(n_2)\ldots bin(n_k)\#$$

Dann setzen wir den Schreib-Lesekopf auf das erste Zeichen des ersten Arguments und starten die Maschine im Anfangszustand. Kommt sie zum Halten, lesen wir das Ergebnis vom Cursor bis zum ersten Leerzeichen '#' als Ergebnis ab. Formal ausgedrückt:

Die Funktion $f :: \mathbb{N}^k \to \mathbb{N}$ heißt Turing-berechenbar, falls es eine Turingmaschine gibt, so dass für beliebige $n_1, \ldots, n_k, m \in \mathbb{N}$ gilt:

$$f(n_1, \ldots, n_k) = m \Leftrightarrow (\#, q_0, bin(n_1)\#\ldots\#bin(n_k)\#) \vdash^* (\#, q_e, bin(m)\#).$$

Im obigen Bild des Turing-Simulators steht auf dem Band „#101#110#“, alle anderen Bandfelder tragen das Leerzeichen „#“, der Kopf steht auf der ersten 1 und die Maschine befindet sich im Zustand Z_{01}. Als Konfiguration ausgedrückt haben wir also $\sigma_0 = (\#, Z_{01}, 101\#110\#)$. Die Folgekonfigurationen sind dann $\sigma_1 = (\#1, Z_{01}, 01\#110\#)$, $\sigma_2 = (\#10, Z_{01}, 1\#110\#)$, $\sigma_3 = (\#101, Z_{01}, \#110\#)$, etc.. Es ist leicht zu erkennen, dass die Maschine die Addition zweier Binärzahlen berechnet.

5.3.7 Variationen

In der Literatur findet sich eine Reihe leicht veränderter Definitionen von Turingmaschinen. Die einfachste Modifikation ist, neben den Bandbewegungen R und L auch zu erlauben, dass die Maschine den Cursor nicht bewegt, was man durch ein '–' anstelle von $'L'$ oder $'R'$ kennzeichnen kann.

Eine dem Eintrag $\delta(q, x) = (x', q', -)$ entsprechende Aktion können wir aber auch mit unserer Maschine simulieren, welche zwingend immer eine Cursorbewegung verlangt. Dazu führen wir einfach einen neuen Zwischenzustand r ein. Den Eintrag für $\delta(q, x) = (x', q', -)$ ersetzen wir durch $\delta(q, x) = (x', r, L)$ und für jedes $y \in \Gamma$ ergänzen wir die neue Zeile r der Turingtabelle an der Spalte y durch $\delta(r, y) = (y, q', R)$.

Statt stehenzubleiben gehen wir also nach links, und benutzen den neuen Zustand r, um den Kopf wieder nach rechts zu bewegen, ohne etwas neues zu schreiben.

Andere Definitionen gehen von einer Menge $E \subseteq Q$ von Endzuständen aus. Die Maschine hält dann nur, wenn ein Endzustand erreicht ist, unabhängig von dem Zeichen unter dem Cursor.

Unsere Turingmaschine können wir leicht zu einer solchen Maschine modifizieren. Dazu führen wir einen zusätzlichen Zustand q_f ein, den wir als Endzustand bestimmen. In jede bisher leer gebliebene Tabellenzelle (q, x) tragen wir dann $\delta(q, x) = (x, q_f, -)$ ein. Dies versetzt die Maschine in den Endzustand, ohne die Bandinschrift oder den Cursor zu verändern.

Die Original-Definition von Turing ging von einem links begrenzten und nach rechts unendlichen Band aus. Solche Maschinen können aber auch ein beidseitig unendliches Band simulieren. Dazu markiert man das Bandende mit einem Sonderzeichen und betrachtet die geraden Bandpositionen als rechten Bandteil und die ungeraden Positionen als linken Bandteil.

Eine Zustandskomponente hält fest, in welchem Bandteil man sich gerade befindet. Eine Cursorbewegung nach links oder rechts im rechten Bandteil ersetzt man durch zwei entsprechende Bewegungen auf dem halbseitigen Band. Eine Bewegung auf dem linken Bandteil ersetzt man durch zwei Bewegungen in der jeweils umgekehrten Richtung. Beim Erreichen des Sonderzeichens muss vom linken Bandteil auf den rechten gewechselt werden und umgekehrt. Die Details sind hier nicht weiter interessant.

5.3.8 Turing-Post-Sprache

Die Zustände einer Turingmaschine als Zeilen der Turingtabelle erinnern an die Zeilennummern in einer sehr einfachen Programmiersprache. Bisher spezifizierte jede Instruktion der Maschine immer, abhängig von Zustand und gelesenem Zeichen eine Kombination von drei verschiedene Aktionen – dem Schreiben eines Zeichens, einer Cursorbewegung und einem Zustandsübergang.

Zerlegt man diese Kombination in einzelne Aktionen so ergeben sich Befehle wie man sie von alten Programmiersprachen kennt.

Martin Davis beschreibt in dem sehr lesenswerten Aufsatz „*What is a Computation*“ eine Sprache, die er *Turing-Post-Language* nennt und die wir hier leicht modifiziert weitergeben. Wir wollen sie als *TPL* abkürzen.

Ein TPL-Program besteht aus Zeilen, die fortlaufend durchnummeriert sind und jeweils einen der folgenden Befehle enthalten:

`LEFT, RIGHT, WRITE x, CASE x JUMP` n`, HALT.`

Der Befehl `HALT` stoppt die Programmausführung. Ansonsten werden die Befehle in der Reihenfolge ausgeführt, in der sie im Programm erscheinen. Eine Ausnahme bildet der *Case*-Befehl. Die Zeile

m`: CASE` x `JUMP` n

befiehlt einen bedingten Sprung: Falls sich das Zeichen x unter dem Cursor befindet, geht es mit Programmzeile n weiter, ansonsten mit der Folgezeile.

Das Turing-Post Programm „+1“ für die Addition von 1 zu einer Binärzahl können wir dann folgendermaßen als TP-Programm formulieren, welches wir zur besseren Verständlichkeit mit Kommentaren versehen haben:

Algorithmus 5.4 TPL-Programm für „+1“

```
// Über die erste Zahl hinweg nach rechts
 10: RIGHT
 20: CASE 0 JUMP 10
 30: CASE 1 JUMP 10

// Von rechts kommend 1-en zu 0-en umwandeln
 40: LEFT
 50: CASE 1 JUMP 80
 60: CASE 0 JUMP 100
 70: CASE # JUMP 100
 80: WRITE 0
 90: CASE 0 JUMP 40
100: WRITE 1

// Nach links laufen
110: LEFT
120: CASE 0 JUMP 110
130: CASE 1 JUMP 110
140: HALT
```

5.3.9 Sprach-Erweiterungen

Die TPL-Sprache können wir um Befehle erweitern, die der Bequemlichkeit des Programmierers und der Übersichtlichkeit des Programms dienen, ansonsten aber die Mächtigkeit der Sprache nicht tangieren. Ein Beispiel ist der unbedingte Sprung ’`JUMP` n’. Falls $\Gamma = \{x_1, \ldots, x_n\}$ ist, kann man diesen als Abkürzung für die Befehlsfolge

```
CASE x_1 JUMP n
...
CASE x_k JUMP n
```

ansehen. Ebenso könnte man auf diejenigen Zeilennummern verzichten, die nicht als Adressen eines Sprungbefehls vorkommen. Schliesslich ist es sogar unerheblich, dass die Zeilen durchnummeriert sind, es reicht, nur die Zeilen durch eine Marke (engl.: *label*) zu kennzeichnen, die Ziel eines Sprunges sind. Auf diese Weise kann man den obigen Algorithmus auch ohne Kommentare verstehen:

Algorithmus 5.5 '+1' als erweitertes TPL-Program

```
rechts: RIGHT
        CASE # JUMP links
        JUMP rechts

links:  LEFT
        CASE 0 JUMP eins
        CASE # JUMP eins

null:   WRITE 0
        JUMP links

eins:   WRITE 1

done:   LEFT
        CASE # JUMP ende
        JUMP done

ende:   HALT
```

5.3.10 TPL-Programme sind Turing-Tabellen

Es ist leicht zu sehen, wie die TPL-Befehle durch Tabelleneinträgen in einer Turingtabelle δ simuliert werden können. Zunächst wird jede Zeile n : zu einem Zustand q_n sodann übersetzt sich

n_:_WRITE_x in die Einträge $\delta(q_n, y) = (x, q_{n+1}, -)$ in jeder Spalte $y \in \Gamma$,

n_:_LEFT in $\delta(q_n, y) = (y, q_{n+1}, L)$ in jeder Spalte $y \in \Gamma$,

$$n\text{\_:\_CASE\_}x\text{\_JUMP\_}m \quad \text{in} \quad \delta(q_n, y) = \begin{cases} (x, q_m, -) & \text{falls } x = y \\ (y, q_{n+1}, -) & \text{sonst.} \end{cases}$$

5.3.11 Turing-Tabellen sind TPL-Programme

Umgekehrt lässt sich jede Turing-Tabelle durch ein TPL-Programm ersetzen. Sei dazu $Q = \{q_1, \ldots, q_n\}$ vorausgesetzt und $\Gamma = \{x_1, \ldots, x_m\}$. Zunächst führen wir für jeden Zustand $q_i \in Q$ eine Sprungmarke $[i]$ ein und einen Codeschnipsel, der abhängig von dem gelesenen Zeichen x_j an die Sprungmarke $[i, j]$ verzweigt:

```
// In Zustand q_i :
[i]:  CASE x_1  JUMP [i,1]
      ....
      CASE x_m  JUMP [i,k]
```

Für jeden Tabelleneintrag $\delta(q_i, x_k) = (y, q_j, L)$ benötigen wir den folgenden Codeschnipsel. Der Fall $\delta(q_i, x_k) = (y, q_j, R)$ ist selbstverständlich analog.

```
// Code für δ(q_i, x_k) = (y, q_j, L) :
[i,k]:  WRITE y
        LEFT
        JUMP [j]
```

Da jeder der Codeschnipsel mit einem Sprung an eine bestimmte Marke endet, ist es fast gleichgültig, in welcher Reihenfolge wir diese Schnipsel zu einem TP-Programm zusammenfügen. Wichtig ist nur, dass das Programm mit dem Code für q_1 beginnen muss, falls dieses der Anfangszustand sein soll.

Jetzt lassen sich leicht interessantere zahlentheoretische Funktionen, wie z.B. Addition, beschränkte Subtraktion ($a \dot{-} b := max(0, a - b)$), Multiplikation, ganzzahlige Division, ggT, etc. als Turing-berechenbar nachweisen. Wir können dazu entweder ein TPL-Programm schreiben oder eine Turing Tabelle und den Code mithilfe eines der Simulatoren testen.

Allerdings stellen wir fest, dass der Umgang mit Zwischenergebnissen, für die wir in einer „normalen“ Programmiersprache Hilfsvariablen verwenden würden, zwar möglich ist, aber echt mühsam. Wir müssen Zwischenergebnisse in festgelegten Reihenfolgen auf das Band schreiben, zum Lesen und Schreiben den Kopf mühsam an die gewünschte Stelle dirigieren. Dazu müssen genügend neue Zustände eingeführt werden, um die gewünschten Zwischenwerte aufzufinden und zu verändern, etc..

An dieser Stelle weicht das Modell der Turingmaschine erheblich von der Arbeitsweise heutiger Rechner ab, bei denen man über eine Adresse direkt auf Speicherinhalte zugreifen kann. Daher wollen wir als zweites und bequemeres Berechnungsmodell eine Registermaschine einführen.

5.3.12 Registermaschinen

Eine *Registermaschine* besteht aus einem gedachten Speicher bestehend aus einem Array von Zellen, $c[0]$, $c[1]$, $c[2]$, … in denen wir jeweils eine natürliche Zahl speichern können. Es gibt keine Obergrenze, weder für die Anzahl der Speicherzellen, noch für die Größe einer Zahl, die wir in jeder einzelnen der Zellen speichern können.

Die Zelle $c[0]$ wird uns als *Accumulator* dienen, nur mit dieser können wir Rechenoperationen durchführen. Auch die Rechenoperationen sind anfangs bescheiden, wir verfügen nur über Addition und beschränkte Subtraktion. Daneben gibt es noch ein *Register*, den *Programmzähler* PC.

Unsere Registermaschinen-Programme (kurz: *RM-Programme*) werden durchnummerierte Programmzeilen haben und der Programmzähler PC beinhaltet immer die Nummer der als nächstes auszuführenden Programmzeile.

Der folgende Befehlssatz steht uns zur Verfügung.

Befehl	Semantik
Load n	$c[0] := n$
Load $[n]$	$c[0] := c[n]$
Store $[n]$	$c[n] := c[0]$
Add $[n]$	$c[0] := c[0] + c[n]$
$\text{Sub}^+[n]$	$c[0] := max(0, c[0] - c[n])$
Goto i	$PC := i$
JZero i	if $c[0] = 0$ then $PC := i$

Tab. 5.1: Registermaschinenbefehle und ihre Semantik

Der erste Befehl `Load` n lädt die Zahl n in den Accumulator.

Argumente in eckigen Klammern sind Indizes in den Speicher. So lädt `Load` $[n]$ den Inhalt der Zelle $c[n]$ in den Accumulator.

Sprünge (`Goto` i) erreicht man durch Eintrag der Sprungadresse i in den Programmzähler.

Der Befehl `JZero` realisiert einen bedingten Sprung, welcher nur ausgeführt wird, falls der Accumulator den Wert 0 enthält.

5.3.13 Berechnung einer Funktion

Um eine Funktion $f :: \mathbb{N}^k \to \mathbb{N}^r$ mit Hilfe eines RM-Programms zu berechnen, schreibt man die Argumente $n_1, \ldots, n_k$ in die Zellen $c[1], \ldots, c[k]$. Alle anderen Zellen enthalten den Wert 0. Dann startet man das RM-Programm. Wenn es terminiert, erwartet man das Ergebnis in den Zellen $c[1], \ldots, c[r]$.

Eine Funktion heißt *RM-berechenbar*, falls sie auf diese Weise von einer Registermaschine berechnet werden kann.

Das folgende RM-Programm zeigt die Berechnung des $ggT(x, y)$ zweier Zahlen x und y.

Algorithmus 5.6 RM-Programm zur Berechnung des ggT

```
0: Load [1]   // x
1: Sub+ [2]   // x − y
2: JZero 5    // x <= y

  // hier ist x > y
3: Store [1]  // x := x − y
4: Goto 0

  // hier ist x <= y
5: Load [2]   // y
6: Sub+ [1]   // y-x
7: JZero 11

  // hier ist y > x
8: Store [2] //y := y − x
9: Goto 0

  // hier gilt: x = y
11: Halt
```

Es ist leicht zu sehen, dass man weitere nützliche Operationen der Registermaschine als Makro definieren kann. Insbesondere lässt sich die Multiplikation $x * y$ als fortgesetzte Addition von x implementieren, wobei y als Zähler dient und bei jedem Additionsschritt dekrementiert wird.

Daher könnten wir auch annehmen, dass die Operation $Mult\,[n]$ zum Repertoire der Maschine gehört.

Analog sind die Operationen $Div\,[n]$ der ganzzahligen Division mit Rest, sowie die Restbildung $Mod\,[n]$ durch fortgesetzte Subtraktion implementierbar. Die Details überlassen wir dem Leser als Übungsaufgabe.

5.4 Iterative Paradigmen

5.4.1 Goto-Programme

Registermaschinen sind durch ihren unmittelbaren Speicherzugriff schon deutlich einfacher zu programmieren, als Turingmaschinen, dennoch sind zwei Details einigermaßen störend:

- die explizite Auswahl von Speicherzellen für die Aufnahme von Werten
- die Benutzung des Akkumulators für jede arithmetische Operation.

Diese beiden Probleme werden durch Goto-Programme behoben. Diese benutzen Variablennamen als Abkürzung für Speicherorte und Inhalte von Speicherzellen. Auch für Goto Programme wollen wir annehmen, dass es für die Größe einer Zahl, die in einer Variablen gespeichert wird, keine Einschränkung gibt. Die grundlegende Operation ist die sogenannte Zuweisung

$$v := e$$

bei der eine Variable v einen neuen Wert erhält, welcher sich durch Auswertung eines Ausdrucks e ergibt.

Im Ausdruck e können Konstanten, Operationen und Variablen vorkommen. Insbesondere kann die Variable v in dem Ausdruck e auftauchen, wie z.B. in $x := x + 1$. Daher ist eine Zuweisung nicht als Gleichung zu lesen, dies macht schon die Zeichenkombination „:=“ deutlich. Vielmehr wird $e$mit dem alten Wert von v ausgewertet, das Ergebnis wird dann zum neuen Wert von v.

In der Mathematik oder der Physik benutzt man traditionell gerne gestrichene Variablen v', x'... etc. um den neuen Wert einer Variablen im Vergleich zu ihrem alten Wert, v, x, ... zu kennzeichnen. In dieser Notation würde einer Zuweisung tatsächlich eine Gleichung entsprechen, wie etwa $x' = x + 1$. Leider hat sich, beginnend mit der Programmiersprache *C*, die Unsitte etabliert, Zuweisungen als Gleichung zu schreiben: x= x+1; was dazu führt, dass für die Gleichheit ein neues Zeichen erfunden werden musste, z.B. '=='.

Goto-Programme haben als elementare Aktionen Zuweisungen $v := e$, wobei für den Ausdruck e lediglich Variablen, die Konstante 0 und die Nachfolgeroperation $succ$ zugelassen sind. Die Grammatik definiert also Ausdrücke (engl.: expression) durch

$$Expr :: id \mid 0 \mid succ(Expr).$$

Boolesche Ausdrücke $BExpr$ sind nötig, um Bedingungen für Sprünge zu formulieren. Wir kommen mit dem Größenvergleich aus:

$$BExpr :: Expr \leq Expr.$$

Als Anweisungen sind Zuweisungen zugelassen, sowie bedingte Sprünge an eine Marke. Der Einfachheit halber wollen wir annehmen, dass die Programmzeilen durch-

nummeriert sind, so dass die Zeilennummern als Marken dienen können. Jede Anweisung beginnt also mit einer Zeilennummer.

Es gibt genau zwei Typen von Anweisungen: die Zuweisung $v := e$ und den bedingten Sprung:

if *BExpr* **Goto** <*m*>.

Hierbei steht $< m >$ für eine beliebige Zeilennummer. Insgesamt sind Anweisungen (engl.: *Statements*) durch folgende Grammatik gegeben:

5.4.2 Syntax von Goto

$$\begin{aligned} Stmt \;::\; & < n >: \quad v := Expr \mid \\ \mid \; & < n >: \quad \text{if } BExpr \text{ Goto } < m > \end{aligned}$$

Ein Programm ist eine durch ';' getrennte Folge von *Stmts*:

$$Prog \;::\; \varepsilon \mid Stmt\,;\, Prog$$

Die Abarbeitung des Programms exekutiert die Anweisungen in der im Programm vorgegebenen Reihenfolge. Davon wird nur abgewichen, wenn die Bedingung einer Sprunganweisung erfüllt ist ist. In diesem Fall wird die Ausführung mit der angegebenen Marke fortgesetzt.

Um eine Funktion $f : \mathbb{N}^k \to \mathbb{N}^r$ mit einem GoTo-Programm zu berechnen, beginnt man mit einer Speicherung der Funktionsargumente $(n_1, \ldots, n_k)$ in den Variablen $x_1, \ldots, x_k$ und startet das Programm. Alle anderen im Programm vorkommenden Variablen seien anfangs mit 0 initialisiert. Das Funktionsergebnis erwartet man bei Beendigung des Programms in den Variablen $y_1, \ldots, y_r$. Das leere Programm ε berechnet somit jede konstante Funktion $c(x_1, \ldots, x_r) = (0, \ldots, 0)$.

Selbstverständlich kann man auch mit Goto-Programmen absichtlich oder versehentlich oder abhängig von den Eingabewerten in Endlosschleifen geraten. Daher berechnen Goto-Programme gelegentlich auch nur partielle Funktionen. So definieren wir auch hier:

Definition 5.4.1. Eine partielle Funktion $f :: \mathbb{N}^k \to \mathbb{N}^r$ heißt *Goto-berechenbar*, wenn es ein Goto-Program gibt, das genau dann terminiert, wenn $f(n_1, \ldots, n_k) \neq \perp$ ist und f in dem obigen Sinne berechnet.

5.4.3 Äquivalenz von Goto mit RM-Programmen

Goto-Programme ähneln offensichtlich RM-Programmen. Zwar fehlen jenen elementare Operationen wie $z := x + y$ oder $z := x \dot{-} y$, aber diese sind leicht durch folgenden Code-Schnipsel zu ersetzen:

Daher ist es klar, dass jede RM-berechenbare Funktion auch Goto-berechenbar ist. Ebenso wie dort lassen sich auch hier die Operationen $*$, div und mod nachrüsten.

Algorithmus 5.7 Goto Programm für $z := x + y$

```
10 : z := x
20 : k := 0
30 : if y <= k goto 70
40 : z := succ(z)
50 : k := succ(k)
60 : if k <= y goto 40
70 :
```

Andererseits ist auch jede Goto-berechenbare Funktion RM-berechenbar. Was die Variablen angeht, braucht man nur jeder Variablen des Goto-Programms ein bestimmtes Register $c[i]$ zuzuordnen. Eine Zuweisung $x := e$ verlangt zuerst einer Berechnung von e mit anschließender Speicherung in (dem Register) der Variablen x. Die Berechnung eines Ausdrucks, wie z.B. $succ(c[n])$ muss über den Akkumulator laufen, z.B. als
$Load\,1; Add\,[n];\ Store\,n.$
Somit stimmen die RM-berechenbaren Funktionen und die Goto-berechenbaren Funktionen überein.

5.5 WHILE Programme

Programme mit Marken und Sprüngen galten schon in den Jugendjahren der Informatik als unschick. Größere Programmen wurden schnell unübersichtlich, daher wurden als Ersatz für bedingte und unbedingte Sprünge bedingte Anweisungen und Schleifen erfunden.

Bedingte Anweisungen „`If` *BExpr* `then` *Stmt1* `else` *Stmt2*“ werten zuerst die Bedingung *BExpr* aus. Falls das Ergebnis wahr ist, wird $Stmt1$ ausgeführt, ansonsten $Stmt2$. Hier ensteht zum ersten Mal eine *geschachtelte Anweisung*, die aus einfacheren Anweisungen aufgebaut ist.

Schleifen „`While` *BExpr* `do` *Stmt*“ prüfen zunächst, ob die Bedingung *BExpr* wahr ist. Wenn ja wird die Anweisung *Stmt* ausgeführt. Anschließend geht es wieder von vorne los. Wenn zum ersten Mal die Überprüfung der Bedingung 'false' ergibt, wird die Schleife beendet.
Die Syntax von WHILE-Programmen ist durch folgende Grammatik gegeben:

$$
\begin{array}{rcl}
Stmt & :: & id := Expr \\
 & | & Stmt\,;\,Stmt \\
 & | & \textbf{if}\ BExpr\ \textbf{then}\ Stmt_1\ \textbf{else}\ Stmt_2 \\
 & | & \textbf{while}\ BExpr\ \textbf{do}\ Stmt
\end{array}
$$

WHILE-Programme ersetzen die Goto-Anweisung durch die WHILE-Schleife. Da es keine Sprünge gibt, benötigt man auch keine Marken. Das Zeichen ′;′ signalisiert die Hintereinanderausführung zweier Statements, erst das linke, dann das rechte.

5.5.1 While-Programme als Goto-Programme

Es ist leicht einzusehen, dass man WHILE-Programme mit Goto-Programmen simulieren kann. Konkret kann man eine Schleife „**while** $E_1 \leq E_2$ **do** $Stmt$ “ in das Fragment

n_1 : **If** $succ(E_2) \leq E_1$ **Goto** n_2
... $U(Stmt)$ ...
Goto n_1
n_2 :

übersetzen, wobei für $U(Stmt)$ die Übersetzung von $Stmt$ in das Programm-Fragment einzufügen ist. Analog lässt sich das Konstrukt „**if** $E_1 \leq E_2$**then**$Stmt_1$ **else** $Stmt_2$“ übersetzen durch das Fragment

n_1 : **If** $E_1 \leq E_2$ **Goto** n_2
... $U(Stmt_2)$ ...
Goto n_3
n_2 :
... $U(Stmt_1)$ ...
n_3 :

wobei auch hier $U(Stmt_2)$ und $U(Stmt_1)$ die Übersetzungen der geschachtelten Anweisungen $Stmt_2$ und $Stmt_1$ sind.

5.5.2 Goto-Programme als While-Programme

Wir gehen von einem Goto-Programm aus, dessen Zeilen wir uns als fortlaufend durchnummeriert vorstellen können.

1: G_1
2: G_2
...
k: G_k

Das zugehörige While-Programme führt eine Variable PC als Programmzähler (*program counter*) ein. Sprünge im Originalprogramm realisiert man durch Veränderung dieses Programmzählers, und ein Rahmenprogramm entscheidet in einer Schleife, welche Anweisung auszuführen ist.

Konkret übersetzen wir zunächst jede Zeile G_i des Gotoprogramms in eine entsprechendes Codefragment W_i unseres While-Programms:

G_i	W_i
$< n >: id := Expr$	$id := Expr\,;\ PC := succ(PC)$
$< n >:$ **if** $BExpr$ **Goto** $< k >$	**if** $BExpr$ **then** PC:= k **else** $PC := succ(PC)$

Jetzt kommt das zu dem ursprünglichen Goto-Programm äquivalente While-Programm als Interpretierer daher:

$PC := 1\ ;$
While $PC \leq k$ **do**
  if $PC \leq 1$ **then** W_1
  else if $PC \leq 2$ **then** W_2
    ...
  else if $PC \leq (k-1)$ **then** W_{k-1}
  else W_k

Damit ist gezeigt, dass die Goto-berechenbaren Funktionen und die While-berechenbaren Funktionen übereinstimmen. Wir werden das Ergebnis im nächsten Abschnitt auch noch auf RM-berechenbare Fuznktionen und Turingmaschinen ausdehnen.

Doch zunächst machen wir eine interessante Beobachtung, die auf S.C.Kleene zurückgeht:

Satz 5.5.1. *[S.C.Kleene] Jede berechenbare Funktion lässt sich in der Form*

$$I\,;\ While\ B\ do\ S$$

schreiben, wobei in I, B und S kein While *mehr vorkommt.*

Korollar. *Jedes Programm lässt sich als While–Programm mit einer einzigen Schleife programmieren.*

5.5.3 Die Äquivalenz zu Turing-Berechenbarkeit

Die RM-berechenbaren, die Goto-berechenbaren und die While-berechenbaren Funktionen stimmen miteinander überein. Wie verhält es sich mit den Turing-berechenbaren Funktionen? Dazu genügt es, wenn wir zeigen, dass wir jede Registermaschine mit einer Turingmaschine simulieren können, und jede Turingmaschine mit einem While-Programm.

Wie man mit einer Turingmaschine ein Goto-Programm simulieren kann, wollen wir nur vage andeuten. Die genaue Konstruktion ist mühsam und nicht sehr erhellend. Eine der Schwierigkeiten ist die Speicherung und der Zugriff auf Variablen und deren Inhalte auf dem Turingband. Dabei muss man ja auch berücksichtigen, dass die

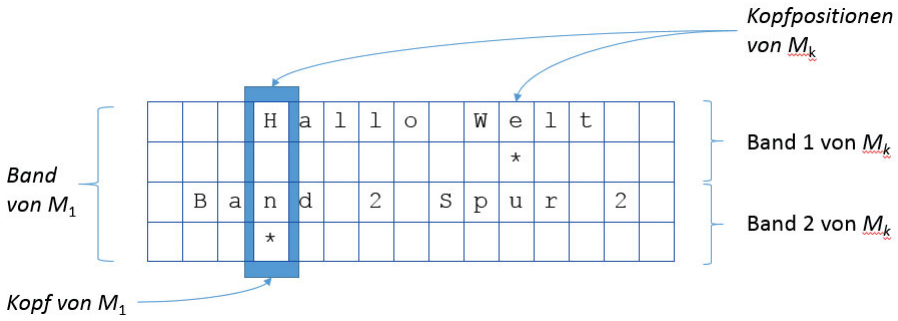


Abb. 5.5.1: Turingmaschine M_1 simuliert 2-Band Maschine M_k

Speicherung einer Zelle der Registermaschine je nach Inhalt mehr oder weniger Zellen des Bandes in Anspruch nehmen kann. Zudem können die gespeicherten Werte wachsen und dann mehr Platz auf dem Band in Anspruch nehmen, was eventuell eine Verschiebung der anderen Variablenwerte auf dem Band nach sich ziehen würde.

Daher greift man zu dem Kunstgriff der k-Mehrband-Turingmaschine. Diese besteht aus einer gemeinsamen Zustandsmenge, aber k Bändern, die jeweils mit eigenen Lese-Schreibköpfen ausgestattet sind, welche sich unabhängig voneinander bewegen lassen. Die Überführungsfunktion δ hat also die Signatur

$$\delta : Q \times \Gamma^k \to Q \times \Gamma^k \times \{L, R, -\}^k$$

Zunächst sieht diese Modifikation harmlos aus, da das Ersetzen eines Alphabets Γ durch ein Alphabet Γ^k uns geläufig ist. So stellen wir ja auch das ASCII-Alphabet als $\{0,1\}^8$ mittels des Binäralphabets $\{0,1\}$ dar. Gravierender ist der Anteil $\{L, R, -\}^k$, der bedeutet, dass sich die Lese-Schreibköpfe unabhängig voneinander bewegen können.

Wollen wir eine k-Band-Turingmaschine M_k mit einer „normalen“ 1-Band-Turingmaschine M_1 simulieren, so stellen wir uns je das Band in $2 * k$ parallele Spuren aufgeteilt vor. Jedes Paar von Spuren repräsentiert ein Band der Mehrbandmaschine, sowie die Position des zugehörigen Lese-Schreibkopfes, diese werde durch ein Zeichen, etwa '*' dargestellt. Das Alphabet von M_1 ist damit zumindest $(Q \times \{\#, *\})^k$ wobei ´#´ wieder das Leere Bandzeichen darstellt, so dass ihr Kopf nicht nur die Zeichen auf allen Bändern registriert, sondern gleichzeitig auch, ob sich der Kopf des i-ten Bandes an der gleichen Stelle befindet, oder nicht.

Um die Zeichen unter allen k Köpfen zu lesen, muss die Maschine über das Band laufen, bis sie alle k Köpfe gesehen hat, und sich „merken“ welches Zeichen sich unter welchem Kopf befindet. Dazu benötigt sie genügend viele Zustände - auf die Details gehen wir hier auch hier nicht ein. Sodann bestimmt δ, welche Aktion jeder Kopf auszuführen hat, so dass die Maschine M_1 wieder an die entsprechenden Kopfpositionen wandert, um die Aktionen auszuführen.

5.5.4 Implementierung einer Turingmaschine als WHILE-Programm

Interessanter und lehrreicher ist die Simulation einer Turingmaschine durch ein WHILE-Programm. Da wir gesehen haben, wie wir Goto-Programme in WHILE implementieren, reicht es hier auf die Implementierung der einzelnen Befehle eines Turing-Post-Programms einzugehen. Die einzige Schwierigkeit ist jedoch die Repräsentation und Manipulation des beidseitig unbegrenzten Bandes der Turingmaschine, da die Variablen in WHILE-Programmen lediglich mit Zahlen $n \in \mathbb{N}$ umgehen können. Hier machen wir uns zu Nutze, dass zu jedem Zeitpunkt nur ein endlicher wenn auch beliebig großer Teil des Bandes beschriftet ist und dass die Zahlen, die in Variablen des WHILE-Programmes gespeichert sind, beliebig groß werden dürfen.

Der Einfachheit halber können wir annehmen, dass $Q = \{0, \ldots, n-1\}$ die Zustände der TM sind. Die $|\Gamma| = d$ vielen Zeichen des Bandalphabets fassen wir als Ziffern des Zahlensystems mit Basis d, wobei wir $\Sigma = \{1, \ldots, d-1\}$ und $\Gamma = \Sigma \uplus \{0\}$ annehmen dürfen. Eine Konfiguration (α, q, β) der Turingmaschine können wir jetzt als Zahlentripel (x, q, y) in drei Variablen der Registermaschine speichern. Da α und β Folgen von Ziffern zur Basis d sind, können wir α als Darstellung einer Zahl x zur Basis d auffassen,

$$x = (\alpha)_d.$$

Statt β auf die gleiche Weise zu speichern, fassen wir die Ziffernfolge von β als umgedrehte Zifferndarstellung einer Zahl auf, mit anderen Worten speichern wir in der Variablen y den Zahlenwert von β^R:

$$y = (\beta^R)_d.$$

Auf diese Weise erzielen wir zwei Vorteile:

- führende Nullen spielen bei der Speicherung von α und β keine Rolle, denn 0 repräsentiert das Leerzeichen '#' und leere Bandfelder links von α sowie rechts von β ändern nichts an dem gespeicherten numerischen Wert.
- obwohl der Lese-Schreibkopf am rechten Ende von α und am linken Ende von β agiert, erfolgen die entsprechenden numerischen Manipulationen jeweils am rechten Ende der Zahlen x und y.

Jetzt müssen wir nur noch die Befehle der Turing-Post-Language implementieren. Dabei übersetzen sich die Bewegungen des Schreib-Lesekopfes, wie auch die Schreib-Operationen in arithmetische Operationen auf den Zahlen x und y. Betrachten wir beispielsweise die Rechtsbewegung des Schreib-Lesekopfes. Dabei geht die Konfiguration $(\alpha, q, b\beta)$ in die Konfiguration $(\alpha b, q', \beta)$ über. Das linkeste Zeichen von β, also das rechteste von β^R wird abgetrennt und rechts an α angehängt. Für die Repräsentation durch die Zahlen $x = (\alpha)_d$ und $y = (\beta^R)_d$ bedeutet das, dass die rechteste Ziffer von y abgetrennt und an $x$angehängt werden muss.

Mit einer Hilfsvariablen a für die vorübergehende Speicherung einer Ziffer ergeben sich auf diese Weise folgende Übersetzungen:

TPL-Befehl	WHILE-Code
`Left`	$a := x \; mod \; d \,;\; y := y * d + a;\; x := x \; div \; d$
`Right`	$a := y \; mod \; d \,;\; x := x * d + a;\; y := y \; div \; d$
`Write` i	$y := y \; div \; d \; + \; i$
`Case` x `Jump` i	IF $y \; mod \; d \; = x$ THEN $PC = i$

Hier wird deutlich, welche zentrale Rolle die arithmetischen Operationen +, *, *div* und *mod* spielen.

5.5.5 Die Macht der zwei Stacks

Betrachtet man sich die Repräsentation einer Bandinschrift samt Cursor, also einer Konfiguration durch ein Paar bestehend aus dem Wort links vom Cursor und das Wort rechts vom Cursor, so erinnert dies nicht zufällig an den Datentyp der „Stackpaare“ aus dem ersten Band dieser Buchreihe (§5.5.6, auf S.369 ff.). Wir hatten dazu zwei Stacks benutzt, um einen Texteditor mit Cursor zu modellieren. Schaut man sich die Implementierung der TPL-Befehle an, so haben wir die gleiche Situation vorliegen:

Zahlen x und y welche α und β^R repräsentieren können wir als Stacks ihrer Ziffern ansehen. Die Operationen *mod* und *div* entsprechen genau den Stackoperationen *top* und *pop*, und die Operation $y := y * d + a$ nichts anderes als $push(y, a)$.

Stackmaschinen haben wir auch in Kapitel 1 dieses Bandes kennengelernt und wir erinnern uns, dass sie genau die kontextfreien Sprachen erkennen können. Würden wir ihnen einen zweiten Stack spendieren, so könnten wir mit dem Trick der Stackpaare ein Turing-Band simulieren. Damit würden 2-Stack-Maschinen die gleichen Sprachen erkennen können wie Turingmaschinen.

Mit einem dritten Stack würden wir keine größere Ausdruckskraft gewinnen, denn mit einer Turingmaschine bzw. mit einem äquivalenten While-Programm könnte man Maschinen mit beliebig vielen Stacks implementieren.

5.5.6 Berechenbarkeit und Churchsche These

Es gibt viele Vorschläge für eine saubere mathematische Definition des Begriffes „Algorithmus“ und des zugehörigen Begriffes der partiellen berechenbaren Funktion. Wir haben jetzt einige davon gesehen:

- Turingmaschinen
- Turing-Post Programme
- Registermaschinen
- Goto-Programme
- WHILE-Programme.

Alle diese Konzepte haben sich als äquivalent herausgestellt. Es ist immer wieder die gleiche Klasse von partiell berechenbaren Funktionen, die durch diese Algorithmenbegriffe definiert werden.

Das ist aber nicht alles. Wir könnten dem leicht gängige Programmiersprachen wie C, Java, Python, Ruby, etc. hinzufügen und wir würden immer wieder auf die gleiche Klasse berechenbarer Funktionen stoßen. Auch funktionale und logische Programmiersprachen wie Haskell oder Prolog machen da keine Ausnahme. Es ist kein Problem, in jeder dieser Sprachen Turingmaschinen zu programmieren und umgekehrt ist es möglich – wenn auch vielleicht mühsam – einen Compiler für C in der WHILE-Sprache zu erstellen.

Neben den oben erwähnten gibt es noch unzählig viele andere Versuche, den intuitiven Begriff des Algorithmus und den zugehörigen Begriff der berechenbaren Funktion mathematisch sauber zu definieren. Immer wieder stellte sich das gleiche Ergebnis ein – Algorithmus und berechenbare Funktionen scheinen universelle Begriffe zu sein, die nicht von den Details einer physikalischen oder logischen Manifestierung abhängen. Abstrakt ausgedrückt lautet diese nur empirisch nachzuprüfende Erkenntnis:

Theorem (Churchsche These)**.** *Jede vernünftige Definition von Berechenbarkeit führt auf die gleiche Klasse der partiell berechenbaren Funktionen.*

Diese Church'sche These ist immer wieder bestätigt worden und sie gilt als eine Art Naturgesetz. Das Wörtchen „vernünftig" lässt natürlich noch ein kleines Hintertürchen offen. Es soll bedeuten, dass man entweder sofort sieht, dass die Definition nicht das beabsichtigte Konzept fasst, oder dass man eine richtige Definition hat. Wir wollen dies an dem Beispiel der Sprache LOOP verdeutlichen.

5.5.7 LOOP-Programme

Gängige Programmiersprachen enthalten neben der WHILE-Schleife weitere Konstrukte, um wiederkehrende Berechnungen zu spezifizieren. Für Schleifen mit einer vorher bestimmten Anzahl von Iterationen besitzen gängige Sprachen noch sogenannte FOR-Schleifen. Diese sind dafür gedacht, über eine gegebene Kollektion von Dingen zu iterieren. Leider ist die Verwendung und die Semantik in verschiedenen Programmiersprachen nicht einheitlich. Daher werden wir das Konstrukt LOOP nennen.

Ein *Loop*-Programm unterscheidet sich von einem *While*-Programm nur dadurch, dass es das Schleifenkonstrukt

$$\text{WHILE } Bexpr \text{ DO } Stmt$$

ersetzt durch

$$\text{LOOP } Expr \text{ DO } Stmt.$$

Die vollständige Syntax ist also:

$$
\begin{aligned}
\mathit{Stmt} \quad :: \quad & \mathit{id} := \mathit{Expr} \\
| \quad & \mathit{Stmt}\,;\, \mathit{Stmt} \\
| \quad & \textbf{if } \mathit{BExpr} \textbf{ then } \mathit{Stmt}_1 \textbf{ else } \mathit{Stmt}_2 \\
| \quad & \textbf{loop } \mathit{Expr} \textbf{ do } \mathit{Stmt} \textbf{ end}
\end{aligned}
$$

Die Schleife wird von einem Ausdruck *Expr* kontrolliert, statt von einem Booleschen Ausdruck *BExpr*. Die Semantik schreibt vor, dass der Ausdruck *Expr* zunächst zu einer Zahl n ausgewertet wird. Anschließend wird das *Stmt*, das den Körper der Schleife darstellt, n mal ausgeführt. Es folgt sofort, dass Loop-Programme immer terminieren muss.

Offensichtlich ist jedes *Loop*-Programm auch durch ein *While*-Programm ersetzbar. Mit zwei neuen Variablen i und k (die nicht in *Stmt* vorkommen) kann man schreiben

$k := \mathit{Expr}; i := 0$;
while $i < k$ **do**
  Stmt ; $i := succ(i)$

Andererseits sieht man unmittelbar, dass Loop-Programme immer terminieren müssen. Es folgt also, dass es sich bei den *Loop*-berechenbaren Funktionen um eine echte Teilmenge der partiell berechenbaren Funktionen handeln muss. Die Hoffnung, dass es sich genau um die Teilmenge der partiell berechenbaren Funktionen handelt, welche total sind, wird sich leider nicht erfüllen. Wir werden später die Ackermann-Funktion kennenlernen. Dies ist eine berechenbare und totale Funktion, die aber nicht durch ein Loop-Programm berechnet werden kann.

5.5.8 LOOP 1.5

Wir werden jetzt LOOP syntaktisch etwas erweitern, damit wir die folgenden Programme etwas leichter formulieren können. An der Mächtigkeit der Sprache ändert sich nichts, wenn wir die folgende **FOR**-Anweisung zulassen.

for $x :=\mathit{Expr}_1$ **to** Expr_2 **do** *Stmt*

berechnet zunächst die Werte n_1 und n_2 von Expr_1 und Expr_2. Dann durchläuft x die natürlichen Zahlen im Intervall $[n_1, n_2]$ und führt für jeden Wert von x die Anweisung *Stmt* aus. Die Semantik der for-Schleife kann man auch mit einem loop-Programm spezifizieren. Allerdings benötigt dieses eine neue Variable y, die in *Stmt* nicht vorkommen sollte.

$x := \mathit{Expr}_1$;
loop $(\mathit{Expr}_2 - \mathit{Expr}_1 + 1)$ **do**
  Stmt; $x := succ(x)$

Mit der for-Konstruktion können wir die arithmetischen Operationen leicht ausdrücken:

```
// Addition                    // Multiplikation
z := x;                        z := 0;
for x := 1 to y do             for x := 1 to y do
   z := succ(z)                   z := z + x
```

Es folgt, dass jedes Polynom $p(x) = a_n x^n + \ldots + a_1 x + + a_0$ loop-berechenbar ist. Der fundamentale Unterschied zwischen while- und loop-Schleifen ist, dass man bei den ersteren nicht weiß, wie oft der Schleifenkörper ausgeführt wird. Immer wenn man eine obere Schranke `maxstep` für die Anzahl der Iterationen ausrechnen kann, könnte man eine while-Schleife

$$\textbf{while } \mathit{BExpr} \textbf{ do } \mathit{Stmt}$$

auch durch folgende for-Schleife ersetzen:

$$\begin{array}{l}\textbf{loop } \mathit{maxstep} \quad \textbf{do} \\ \quad \textbf{if } \mathit{BExpr} \quad \textbf{then} \quad \mathit{Stmt}\end{array}$$

Aus dieser Tatsache lässt sich sofort folgender Satz leicht herleiten.

Satz 5.5.2. *Jeder terminierende Algorithmus polynomialer Komplexität ist Loop-berechenbar.*

Beweis. Wir beschränken uns auf Algorithmen mit einer Input-Variablen x. Es gibt ein Polynom $c * x^n$ so dass $A(x)$, der Algorithmus mit Input x höchstens $c * x^n$ Schritte benötigt. Den Algorithmus können wir aufgrund des Satzes von Kleene in der Form

$$S_1; \textbf{ while } \mathit{BExpr} \textbf{ do } S_2$$

schreiben. Aus dem input x können wir mit Hilfe eines loop-Programms $maxstep := c * x^n$ berechnen. Sodann ersetzen wir wie oben gesehen die while-Schleife durch eine loop-schleife. □

5.5.9 Die Grenzen Loop-berechenbarer Funktionen.

Wir haben gesehen, wie man mit dem Loop-Konstrukt die Addition und die Multiplikation berechnen kann. Man könnte dies leicht durch die Exponentiation fortsetzen.

```
z := 1;
for x := 1 to y do
   z := z * x
```

Man könnte so fortfahren und Hyper-Exponentiation, Hyper-Hyper-Exponentiation etc. definieren. Es war die Idee von Wilhelm Ackermann, diese Reihe fortzusetzen zu einer dreistelligen Operation $f(x, y, k)$, in der das dritte Argument k bestimmt, um welche der Operation in der Reihe es sich handeln solle, also $f(x, y, 1) = x + y$, $f(x, y, 2) = x * y$, $f(x, y, 2) = x^y$ etc. Man kann dann leicht sehen, dass die Funktion $f(x, y, k)$ sowohl total als auch berechenbar ist, allerdings konnte Ackermann zeigen, dass sie nicht durch ein LOOP-Programm berechenbar ist.

Somit bilden LOOP-Programme kein „vernünftiges“ Berechenbarkeitskonzept im Sinne der Churchschen These.

5.6 Rekursive Funktionen

Die bisherigen Berechnungskonzepte orientierten sich an der Ausführung physikalischer Maschinen. Schritt für Schritt arbeiten diese ein Programm ab, wobei allerdings Sprünge erlaubt sind, und wenn sie fertig sind, liegt das Ergebnis der Berechnung vor.

5.6.1 Funktionale Sprachen

Funktionale Programmierparadigmen orientieren sich an der mathematischen Vereinfachung von Ausdrücken. Ein Funktionsaufruf, evtl. mit Parameter, ist ein mathematischer Ausdruck, der schrittweise vereinfacht wird, wobei jeweils eine Seite einer Gleichung durch die andere Seite ersetzt wird. Wenn keine Ersetzung mehr vorzunehmen ist, ist die Berechnung fertig. Dies führt dazu, dass man ein funktionales Programm weniger als eine Handlungsanweisung als eine Gleichung auffasst, die es zu vereinfachen gilt. Insbesondere haben funktionale Programme keine Zuweisungen und auch keine explizite Schleifenkonstruktionen. Deren Rolle werden von dem Begriff der *Rekursion* übernommen. Dies sind Funktionsdefinitionen, in denen die zu definierende Funktion – mit einfacheren Argumenten – bereits vorkommt.

Der Vorteil einer solchen Programmierung ist, das sie mathematisch klarer und weniger fehleranfällig ist und sehr schnell zu einem lauffähigen Programm führt, was man auch unter dem Schlagwort „rapid prototyping“ verbucht. Der Nachteil des funktionalen Programmierens ist, dass es zu wenige mathematisch gebildete Programmierer gibt und auch diese oft mit dem iterativen Programmierstil aufgewachsen sind, und das Umdenken in eine mathematisch eigentlich logischere Methodik scheuen. Gegenwärtig ist aber eine Tendenz zur Integration funktionaler Konzepte in bestehende Mainstream-Sprachen zu erkennen, der sich vermutlich weiter fortsetzen wird. Rekursion wird schon länger von allen gängigen Sprachen unterstützt, aber auch die klassischen Methoden und Ausdrucksmittel wie garbage collection, funktionale Argumente und Funktionsobjekte („lambdas“) werden immer populärer.

Die einfachsten Funktionalen Programme sind einfach nur Abkürzungen. Wir können dies an der Python-Funktion wie beispielsweise „*abstand*“ betrachten, mit

der wir beispielsweise den Abstand eines Punktes Beispielsweise könnte man eine Funktion definieren, die den Abstand eines Punktes (x, y) vom Nullpunkt berechnet:

```
def abstand(x,y):
      return sqrt(x*x+y*y)
```

Diese Funktion ersetzt lediglich jeden Ausdruck der Form `abstand(x,y)` durch den Ausdruck `sqrt(x*x+y*y)`, konkret also beispielsweise `abstand(3,4)` durch `sqrt(3*3+4*4)`. Dies ist erneut ein Funktionsaufruf, diesmal der Funktion `sqrt`, deren Argumente aber zuerst berechnet werden müssen. Das bedeutet, dass zunächst 3*3 zu 9 vereinfacht werden muss, 4*4 zu 16 und anschließend 9+16 zu 25. Dann started der Aufruf `sqrt(25)` und liefert das Endergebnis.

Selbstverständlich ist die Funktion `sqrt`, die die Wurzel einer Zahl berechnet, in allen herkömmlichen Sprachen eingebaut, aber nehmen wir einmal an, wir wollten die Funktion `sqrt(x)` selber programmieren und nehmen wir weiter an, wir wären nur an dem ganzzahligen Anteil der Wurzel interessiert, also an $\lfloor \sqrt{x} \rfloor$. Iterativ erscheint die Lösung klar: Wir benutzen eine Variable k, die wir mit 0 initialisieren ($k := 0$) und anschließend erhöhen wir k schrittweise, wobei wir immer testen, ob $k * k \leq x$ ist: `while k*k <= n do k := k+1`. Zum Schluß geben wir `k-1` als Ergebnis zurück.

Wir können diese Vorgehensweise auch mit Hilfe von Rekursion ausdrücken.

```
def sqrt(n):
      return sqrt_aux(n,0)

def sqrt_aux(n,k):
      if k*k > n: do
            return k-1
      else:
            return sqrt_aux(n,k+1)
```

Der Aufruf `sqrt(n)` führt zu einem Aufruf einer Hilfsfunktion `sqrt_aux` mit einem zusätzlichen Argument, das wir wie die Hilfsvariable `k` einsetzen und das mit dem Wert 0 beginnt. `sqr_aux(n,k)` prüft, ob `k*k` schon größer als n ist. Wenn ja, liefert sie `k-1` andernfalls wird der Aufruf von `sqrt(n,k)` durch den Aufruf von `sqrt(n,k+1)` ersetzt.

5.6.2 Rekursion

So wie man iterative Programmierkonzepte auf die allernötigsten Bestandteile reduziert und dann zu Turingmaschinen, Registermaschinen, While- oder Goto-Programme stößt, so wollen wir es auch mit funktionalen Sprachen machen, die von Rekursion und Vereinfachung von Ausdrücken leben. Hier wollen wir mit einer möglichst einfachen Datenstruktur – den natürlichen Zahlen mit der Nachfolgeroperation – beginnen

und zusammen mit dem Prinzip der Rekursion neue Funktionen definieren. Wir werden am Ende die Churchsche These bestätigen – die so definierbaren Funktionen sind genau die uns bereits bekannten partiellen berechenbaren Funktionen.

Im ersten Schritt werden wir ein Rekursionsschema einführen, das als *primitives Rekursionsschema* bezeichnet wird, und mit dem wir nur totale Funktionen definieren können. In einem zweiten Schritt werden wir das Schema der μ*–Rekursion* einführen und zeigen, dass wir damit exakt alle partiellen berechenbaren Funktionen erfassen können.

Primitiv rekursive Funktionen entstehen aus einfachen

- Basisfunktionen

mit Hilfe von zwei Operationen, mit denen wir aus bereits vorhandenen Funktionen neue zu bilden dürfen, nämlich

- Komposition und
- primitive Rekursion.

5.6.3 Basisfunktionen

Wir beginnen mit den *Basisfunktionen* bestehend aus allen konstanten Funktionen, allen Projektionen und der Nachfolgeroperation

- $f(x) = k$ für eine beliebige Konstante $k \in \mathbb{N}$,
- $\pi_i^n(x_1, \ldots, x_n) = x_i$ für alle n und alle $i \leq n$
- $succ(x) = x + 1$

5.6.4 Komposition

Bei der *Komposition* handelt es sich um das Einsetzen von Funktionen in die Argumentpositionen anderer Funktionen. Für einstellige Funktionen entsteht aus einer Funktion $f : \mathbb{N} \to \mathbb{N}$ und einer zweiten Funktion $g : \mathbb{N} \to \mathbb{N}$ durch Einsetzen die Funktion $f \circ g$ mit der Definition $(f \circ g)(x) = f(g(x))$. Allgemeiner sei f eine k–stellige Funktion und seien $g_1, \ldots, g_k$ jeweils n-stellige Funktionen. Dann ist auch die Funktion $f(g_1, \ldots, g_k)$ primitiv rekursiv. Sie ist definiert durch

$$f(g_1, \ldots, g_k)(x_1, .., x_n) = f(g_1(x_1, .., x_n), \ldots, g_k(x_1, .., x_n))$$

Man rechnet also zuerst alle Werte $y_1 = g(x_1, \ldots, x_n), \ldots, y_k = g_k(x_1, .., x_n)$ aus und wendet auf das Ergebnistupel $(y_1, .., y_n)$ die Funktion g an.

Die Definition der Komposition erscheint etwas umständlich und wir werden sie nicht immer so explizit hinschreiben, wie es der Definition entspricht. Prinzipiell gestatten sie uns

- Funktionen ineinander einzusetzen,
- Variablen zu vertauschen,
- Variablen gleichzusetzen.

Sei zum Beispiel die Funktion $g(x, y, z)$ bereits vorhanden, dann können wir auch die Funktion $h(x, y) := g(y, y, succ(x))$, auf folgende Weise aus den Basisfunktionen gewinnen:

$h = g(f_1, f_2, f_3)$ wobei $f_1 = f_2 = \pi_2^2$ und $f_3 = succ(\pi_1^2)$ ist. In der Tat gilt:

$$\begin{aligned} g(f_1, f_2, f_3)(x, y) &= g(f_1(x, y), f_2(x, y), f_3(x, y)) \\ &= g(\pi_2^2(x, y), \pi_2^2(x, y), succ(\pi_1^2)(x, y)) \\ &= g(y, y, succ(x)). \end{aligned}$$

5.6.5 Primitives Rekursionsschema

Dies ist das wichtigste Prinzip zur Konstruktion primitiv rekursiver Funktionen, wobei man beachten sollte, dass der Begriff „primitiv“ in der Mathematik gerne in der Bedeutung „einfach“, „nicht verfeinerbar“ benutzt wird. Insofern ist die primitive Rekursion eine einfachste Form der Rekursion, nämlich über den natürlichen Zahlen. Wir nehmen an, wir hätten bereits eine k-stellige Funktion $g : \mathbb{N}^k \to \mathbb{N}$ und eine $k + 2$-stellige Funktion $h : \mathbb{N}^{k+2} \to \mathbb{N}$ gegeben. Dann gewinnen wir eine $k + 1$-stellige Funktion $f : \mathbb{N}^{k+1} \to \mathbb{N}$ durch die Definition

$$f(0, x_1, \ldots, x_k) = g(x_1, \ldots, x_k) \tag{5.6.1}$$

$$f(n + 1, x_1, \ldots, x_k) = h(\underline{f(n, x_1, .., x_k)}, n, x_1, \ldots, x_k). \tag{5.6.2}$$

Hier läuft die Fallunterscheidung über das erste Argument von f. Falls es 0 ist, das ist der Basisfall, liefert g sofort das Ergebnis. Ansonsten berechnet man zunächst $w = f(n, x_1, .., x_k)$ und erstellt daraus mit Hilfe von n sowie $x_1, \ldots, x_n$ das Ergebnis $h(w, n, x_1, \ldots, x_k)$.

Wir betrachten nun einige primitiv rekursive Funktionen, und wie diese definiert werden können.

```
add(0,x)   = x
add(n+1,x) = succ(add(n,x))

pred(0)    = 0
pred(n+1) = n

sub(x,0) = x
sub(x,y+1) = pred(sub(x,y))

mult(x,0) = 0
mult(x,y+1) = add(mult(x,y),x)

not(0) = 1
not(x+1) = 0
```

Im ersten Fall benutzen wir das primitive Rekursionsschema mit $k = 1$, $g = \pi_1^1 = id$ und $h(w, n, x) = \pi_3^3(w, n, x) = x$. Im zweiten Fall ist $k = 0$, die konstante Funktion mit Wert 0 und $h(w, n) = n$.

Im Falle der Funktionen sub und $mult$ lassen wir die Induktion über die zweite Variable, y, laufen. Dies ist zulässig, da wir mit Hilfe der Projektionsfunktionen, wie oben schon erwähnt, die Argumente beliebig vertauschen oder gleichsetzen können.

Die Funktion not hilft uns, Zahlen auch als Repräsentanten von Wahrheitswerten zu benutzen wobei wir willkürlich $0 = false$ und ansonsten $n + 1 = true$ wählen. Mit dessen Hilfe können wir erkennen:

Lemma 5.6.1. *Seien $B : \mathbb{N}^k \to \mathbb{N}$ swie $f, g : \mathbb{N}^k \to \mathbb{N}$ (primitiv) rekursiv, dann auch*

$$ite(x_1, \ldots, x_k) := \textit{if } B(x_1, \ldots, x_k) \geq 1 \textit{ then } f(x_1, \ldots, x_k) \textit{ else } g(x_1, \ldots, x_k).$$

Beweis. $not(not(x))$ liefert 1 falls $x \neq 0$ und 0 sonst. Wir benutzen zusätzlich die oben definierten Funktionen add und $mult$, die wir aber wie üblich infix und mithilfe der Zeichen $'+'$ und $'\star'$ notieren:

$$\begin{aligned} ite(x_1, \ldots, x_k) &= not(not(B(x_1, \ldots, x_k))) \star f(x_1, \ldots, x_k) \\ &\quad + \; not(B(x_1, \ldots, x_k)) \star g(x_1, \ldots, x_k) \end{aligned}$$

ist Komposition von p.r. Funktionen und hat die gewünschte Semantik. □

Wir haben den Begriff „primitiv“ im obigen Lemma in Klammern gesetzt, weil es auch für beliebige rekursive Funktionen gilt, die wir unten noch definieren werden.

Die Definition von $sqrt$ bzw. $sqrt\_aux$ in dem eingangs erwähnten Beispiel folgt nicht dem primitiven Rekursionsschema. Trotzdem werden wir sehen, dass beide Funktionen primitiv rekursiv sind. Man könnte sie auch mithilfe des Schemas definieren, was wir aber hier nicht unternehmen wollen, denn die Aussage wird sich aus der Charakterisierung der primitiv-rekursiven Funktionen sofort ergeben. Zunächst stellen wir fest:

Satz 5.6.2. *Jede primitiv rekursive Funktion ist total.*

Beweis. Offensichtlich sind alle Basisfunktionen totale Funktionen von $\mathbb{N}^k$ nach $\mathbb{N}$ und die Komposition totaler Funktionen führt wieder zu totalen Funktionen. Sind g und h im primitiven Rekursionsschem total, so folgt, dass auch die definierte Funktion f total ist. Offensichtlich ist nämlich $f(0, x_1, \ldots, x_n) = g(x_1, \ldots, x_n) \neq \perp$ und falls $f(n, x_1, \ldots, x_n) = w \neq \perp$ ist, folgt auch, dass $f(n + 1, x_1, \ldots, x_n) = h(w, n, x_1, \ldots, x_n) \neq \perp$. Vollständige Induktion liefert also sofort die gewünschte Aussage.

Interessanterweise stoßen wir mit den primitiv rekursiven Funktionen auf alte Bekannte, denn es gilt: □

Lemma 5.6.3. *Jede primitiv rekursive Funktion ist Loop-berechenbar.*

Beweis. Zunächst sind die Basisfunktionen offensichtlich Loop-berechenbar.

$y_1 := x_i$ berechnet π_i^n

$y_1 := x_1 + 1$ berechnet die Nachfolgerfunktion und

$y_1 := 1 + \ldots + 1$ berechnet die gewünschte konstante Funktion mit $n = 1 + \ldots + 1$.

Sind f und $g_1, \ldots, g_k$ Loop-berechenbar, so berechenen wir der Reihe nach $g_1(x_1, .., x_n), \ldots, g_k(x_1, .., x_n)$ wobei wir die Ergebnisse in den Variablen $z_1, \ldots, z_k$ speichern. Zum Schluss kopieren wir diese Werte in die Variablen $x_1, \ldots, x_k$ und starten die Berechnung von f. Das Ergebnis ist $f(g_1(x_1, .., x_n), \ldots, g_k(x_1, .., x_n)) = f(g_1, \ldots, g_k)(x_1, .., x_n)$.

Für die durch das primitive Rekursionsschema (5.6.1,5.6.2) gegebene Funktion f können wir davon ausgehen, dass g und h bereits durch Loop-Programme berechnet werden können. Ihr Ergebnis, speichern wir in der Variablen w. Anfangs gilt dann $w = f(0, x_1, \ldots, x_k)$ und in jedem Schleifendurchlauf berechnen wir aus $w = f(z, x_1, \ldots, x_k)$ den neuen Wert $f(z + 1, x_1, \ldots, x_k)$. In jedem Schleifendurchlauf wird aus $w = f(z, x_1, \ldots, z_k)$ der neue Wert $f(z + 1, x_1, \ldots, x_k)$ berechnet. □

```
w := g(x_1, ..., x_k)
for z := 1 to n do
    w := h(w, z, x_1, ..., x_k)
```

Es gilt aber auch die Umkehrung:

Lemma 5.6.4. *Jede Loop-berechenbare Funktion ist primitiv rekursiv.*

Wir verallgemeinern die Aussage leicht, indem wir Funktionen von $f : \mathbb{N}^n \to N^n$ betrachten. Eine solche Funktion soll primitiv rekursiv heißen, falls jede Koordinatenfunktion $f_i := \pi_i^n \circ f$ primitiv rekursiv ist. Nach Ausführung von f können wir annehmen, dass der Wert von f_i in der i-ten Variablen x_i gespeichert ist. Seien jetzt also $x_1, \ldots, x_n$ alle Variablen in einem Loop-Programm P, dann ist P durch eine primitiv rekursive Funktion f_P beschrieben. Wir lassen P alle Formen von Loop-Programmen durchlaufen:

1. Zuweisung : $x_i := e(x_1, \ldots, x_n)$
   - $f_{:=}(x_1, \ldots, x_k) = (x_1, \ldots, x_{i-1}, e(x_1, \ldots, x_n), x_{i+1}, \ldots, x_k)$
2. Hintereinanderausführung $P; Q$
   - $f_{P;Q}(x_1, \ldots, x_n) = f_Q(f_P(x_1, \ldots, x_n))$
3. Bedingung $if\ B\ then\ P\ else\ Q$
   - $f_{if}(x_1, \ldots, x_n) = ite(B(x_1, \ldots, x_n), f_P(x_1, \ldots, x_n), f_Q(x_1, \ldots, x_n))$
4. Iteration for $i := 0$ to $e(x_1, \ldots, x_n)$ do P
   - $f_{FOR}(x_1, \ldots, x_n) = g(e(x_1, \ldots, x_n), x_1, \ldots, x_n)$ wobei

$$\begin{aligned} g(0, x_1, \ldots, x_n) &= (x_1, \ldots, x_n) \\ g(k + 1, x_1, \ldots, x_n) &= f_P(g(k, x_1, \ldots, x_n)) \end{aligned}$$

Zusammengefasst besagen die beiden Lemmata also:

Satz 5.6.5. *Eine Funktion ist genau dann primitiv rekursiv, wenn sie Loop-berechenbar ist.*

5.6.6 Die Ackermann-Funktion

Die primitiv rekursiven Funktionen sind offenbar auch eine besondere Klasse von Funktionen, auf die wir auf ganz verschiedenen Wegen gestoßen sind. Jede primitiv rekursive Funktion ist total, und jede totale berechenbare Funktion polynomialer Komplexität ist primitiv rekursiv. Leider ist aber nicht jede totale berechenbare Funktion primitiv rekursiv. Ein Gegenbeispiel muss also sehr komplex sein. Das bekannteste Beispiel einer solchen Funktion stammt von Wilhelm Ackermann, einem Schüler von David Hilbert in Göttingen. Ein anderes Beispiel wurde von Gabriel Sudan, einem andern Schüler Hilberts gefunden. Die heute als Ackermann-Funktion bezeichnete Funktion ack, die wir auch hier betrachten wollen, ist eine zweistellige Version der ursprünglichen Ackermannschen Funktion, welche von Rózsa Péter, einer ungarischen Mathematikerin, 1934 gefunden wurde:

$$\begin{aligned} ack(0, y) &= y + 1 \\ ack(x + 1, 0) &= ack(x, 1) \\ ack(x + 1, y + 1) &= ack(x, ack(x + 1, y)) \end{aligned}$$

Diese Funktion ist total und berechenbar. In jeder Sprache, die Rekursion unterstützt kann man *ack* direkt programmieren. Versucht man die Funktion von Hand auszurechnen, erkennt man, dass die Rechnung sehr aufwendig sein kann. Als Beispiel wollen wir den Beginn der Berechnung von $ack(3, 3)$ ansehen:

$$\begin{aligned} ack(3, 3) &= ack(2, ack(3, 2)) \\ &= ack(2, ack(2, ack(3, 1))) \\ &= ack(2, ack(2, ack(2, ack(3, 0)))) \\ &= ack(2, ack(2, ack(2, ack(2, 1)))) \\ &= \ldots \text{ ca. 1200 Zeilen später} \ldots \\ &= 61 \end{aligned}$$

Die Ackermann Funktion ist nicht nur sehr aufwendig zu berechenen, sie wächst auch extrem schnell. Aus der Literatur entnehmen wir:

$$ack(4, 2) \approx 10^{20000}$$

$$ack(4,4) \approx 10^{10^{10^{21000}}}.$$

Dennoch ist die Funktion berechenbar, sie terminiert immer. In jedem rekursiven Aufruf wird entweder x kleiner, oder aber x bleibt gleich und y wird kleiner. Daher haben wir es hier mit einer totalen und berechenbaren Funktion zu tun.

Nun wollen wir zeigen, dass ack nicht primitiv rekursiv sein kann. Dazu zeigen wir, dass *ack* schneller wächst, als jede primitiv rekursuive Funktion:

Satz 5.6.6. *Für jede primitiv rekursive Funktion $f : \mathbb{N}^r \to \mathbb{N}$ gibt es ein $k \in \mathbb{N}$, so dass für alle $x_1, \ldots, x_r \in \mathbb{N}$ gilt*

$$f(x_1, \ldots, x_r) < ack(k, x_1 + \ldots + x_r).$$

Bevor wir zum Beweis schreiten wollen wir die für uns wichtigste Folgerung ziehen:

Korollar 5.6.7. *ack ist total, berechenbar, aber nicht primitiv rekursiv.*

Beweis. Angenommen *ack* wäre primitiv rekursiv, dann würde das auch für $f(n) = ack(n, n)$ gelten. Aus dem Satz könnten wir dann die Existenz eines k folgern so dass $f(n) < ack(k, n)$ für alle $n \in \mathbb{N}$. Insbesondere wäre dann

$$ack(k, k) = f(k) < ack(k, k),$$

was offensichtlich absurd ist. □

Der Beweis von Satz 5.6.6 wird sich aus einer Reihe von Lemmatas ergeben, deren Beweise jeweils relativ einfach mittels Induktion auf die jeweils vorigen zurückgeführt werden können.

Lemma 5.6.8. *Für alle $x, y \in \mathbb{N}$ gilt:*

1. $ack(1, y) = y + 2$
   - Induktion über y
2. $ack(2, y) = 2y + 3$
   - Induktion über y
3. $y < ack(x, y)$
   - Induktion über x mit Nebeninduklтion über y
4. $ack(x, y) < ack(x, y + 1)$
   - Induktion über x
5. $ack(x, y + 1) \leq ack(x + 1, y)$
   - Induktion über y
6. $ack(x, y) < ack(x + 1, y)$
   - aus dem Vorigen.

Aus diesen induktiv zu gewinnenenden Hilfssätzen folgern wir jetzt das Lemma:

Lemma 5.6.9. *Zu $a_1, a_2 \in \mathbb{N}$ gibt es stets ein c mit $ack(a_1, y) + ack(a_2, y) < ack(c, y)$.*

Beweis. Setze $d := max(a_1, a_2)$, dann gilt wegen der gerade bewiesenden Lemmata und der Definition von ack:

$$\begin{aligned} ack(a_1, y) + ack(a_2, y) &\leq ack(d, y) + ack(d, y) \\ &< 2 * ack(d, y) + 3 \\ &= ack(2, ack(d, y)) \\ &< ack(d + 2, ack(d + 3, y)) \\ &= ack(d + 3, y + 1) \\ &\leq ack(d + 4, y). \end{aligned}$$

Nach diesen Vorbereitungen können wir endlich den Beweis von Satz 5.6.6 angehen:

Für die Basisfunktionen ist die Behauptung klar.

Wir betrachten der Einfachheit halber nur Komposition und primitive Rekursion von zweistellige Funktionen. Seien also g und h primitiv rekursiv. Dann gibt es nach Induktionsvoraussetzung schon Zahlen $a, b \in \mathbb{N}$ mit $g(y) < ack(a, y)$ und $h(y) < ack(b, y)$.

Komposition:

$$\begin{aligned} g(h(y)) &< ack(a, h(y)) \\ &\leq ack(a, ack(b, y)) \\ &< ack(m, ack(m + 1, y) \text{ mit } m = max(a, b) \\ &= ack(m + 1, y). \end{aligned}$$

Primitive Rekursion: Sei $f(0, y) = g(y)$ und $f(x + 1, y) = h(f(x, y), x, y)$. Wir können voraussetzen, dass es $a, b \in \mathbb{N}$ gibt mit $g(y) < ack(a, y)$ und $h(x, y, z) < ack(b, x+y+z)$. Es folgt $g(y)+y < ack(a', y)$ sowie $h(x, y, z)+x+y+z < ack(b', x+y+z)$ für geeignete $a' > a$ und $b' > b$.

Für $e = max(a', b') + 1$ zeigt man nur durch Induktion nach x :

$$f(x, y) + x + y < ack(e, x + y).$$

□

5.6.7 Minimalisierung und μ-Rekursion

Primitive Rekursioon liefert nicht alle berechenbaren Funktionen, nicht einmal alle totalen, wohl aber alle „effizient berechenbaren“. Wir benötigen also ein weiteres Rekursionsprinzip, mit dem wir, analog wie mit der while-schleife, auch Berechnungen

ausdrücken können, von denen wir nicht wissen, ob und wann sie jemals terminieren. Der μ–Operator, den wir dazu einführen, sucht eine Zahl als Lösung für ein Problem und er geht so vor, dass er alle Zahlen der Reihe nach durchprobiert, ob sie vielleicht das Problem lösen. Das ist eine sehr primitive Methode, aber sie kann theoretisch eine Lösung finden, wenn eine solche existiert.

Ein Problem mit Parametern $x_1, \ldots, x_k$ können wir durch eine berechenbare Funktion $P : \mathbb{N}^{k+1} \to \mathbb{N}$ codieren. Beispielsweise können wir ein n suchen, so dass $P(n, x_1, \ldots, x_k) = 0$ ist. Wir suchen nun ein n, das das Problem löst. Wenn P berechenbar ist, sollte die Lösung, falls sie existiert, auch gefunden werden können, wir probieren zuerst $n = 0$ und berechnen $P(0, x_1, \ldots, x_k)$, dann $n = 0$ und $P(1, x_1, \ldots, x_k)$ etc.. Sobald wir zum ersten Mal ein n gefunden haben, mit $P(n, x_1, \ldots, x_k) = 0$, geben wir dieses n aus. Auf diese Weise, so scheint es, finden wir sogar das kleinste n für das $P(n, x_1, \ldots, x_k) = 0$ ist. Unser Algorithmus berechnet dann

$$min(P)(x_1, \ldots, x_k) = min\{n \in \mathbb{N} \mid P(n, x_1, \ldots, x_k)\}. \tag{5.6.3}$$

Der vorgeschlagene Operator min macht offenbar aus einer $(k+1)$-stelligen Funktion P eine k-stellige Funktion $min(P)$.

Allerdings kann es auch vorkommen, dass für eine bestimmte Wahl der Parameter überhaupt keine Lösung existiert, also kein n mit $P(n, x_1, \ldots, x_n) = 0$. In diesem Fall terminiert die Suche nicht und damit ist $min(P)$ ist eine partielle berechenbare Funktion. Auf jeden Fall kommen durch diese Minimumsberechnung auch partiell berechenbare Funktionen ins Spiel, selbst wenn P total ist. Wenn aber P nur partiell berechenbar ist, ist es denkbar, dass durch die oben angedeutete systematische Suche das gewünschte Ergebnis $r = min\{n \mid P(n, x_1, \ldots, x_k) = 0\}$ gar nicht gefundne wird. Das ist etwa der Fall, wenn für ein $m < r$ einmal $P(m, x_1, \ldots, x_k) = \perp$ ist. Dann bleibt die oben beschriebene Suche bei dem Test $P(m, x_1, \ldots, x_k) \stackrel{?}{=} 0$ stecken ohne jemals $n = r$ zu erreichen.

Wegen dieser Komplikation wird aus dem ursprünglich gewünschten Minimumsoperator der sogenannte μ–Operator, wobei das μ selbstverständlich die Nähe zu dem Begriff „min“ ausdrückt.

Sei $P : \mathbb{N}^k \to \mathbb{N}$ eine $(k+1)$–stellige partiell berechenbare Funktion. Der μ–Operator liefert eine k-stellige partiell berechenbare Funktion $\mu(P)$ mit der Definition

$$\mu(P)(x_1, \ldots, x_k) := \begin{cases} n & P(n, x_1, \ldots, x_k) = 0 \wedge \forall_{m<n}.P(m, x_1, \ldots, x_k) \in \mathbb{N} \setminus \{0\} \\ \perp & \text{sonst.} \end{cases}$$

Bleibt also bei der Suche nach n für ein $m < n$ die Suche stecken, weil $P(m, x_1, \ldots, x_k) = \perp$ ist, dann ist die erste Bedingung in der Definition von $\mu(P)$ nicht erfüllt und konsequenterweise ist dann auch $\mu(P)(x_1, \ldots, x_k) = \perp$.

Ist P berechenbar, so ist offensichtlich auch $\mu(P)$ berechenbar. Das gleiche lässt sich von $min(P)$ nicht sagen. Daher werden wir den μ–Operator zu unserem Werkzeugkasten der rekursiven Funktionen hinzunehmen und gelangen zur Definition der μ–rekursiven Funktionen:

Definition 5.6.10. Die μ–rekursiven Funktionen entstehen aus den Basisfunktionen (konstante Funktionen, Projektionen und successor-Operation) durch Abschluss unter Komposition, primitiver Rekursion und der Anwendung des μ-Operators.

Bei der Anwendung des μ–Operators können partielle Funktionen entstehen, auf die dann auch wieder Komposition, primitive Rekursion und μ–Operator angewendet werden dürfen. Insbesondere ist auch die überall undefinierte Funktion Ω μ–rekursiv. Wir wählen z.B. $P_{=1}(n, x_1) := 1$ als zweistellige Funktion mit konstantem Wert 1 und definieren

$$\Omega(x_1) := \mu(P_{=1})(x_1) = \begin{cases} n & P_{=1}(n, x_1) = 0 \wedge \ldots \\ \bot & \text{sonst,} \end{cases} = \begin{cases} n & 1 = 0 \wedge \ldots \\ \bot & \text{sonst} \end{cases} = \bot.$$

Man ahnt es schon:

Satz 5.6.11. *Die μ–rekursiven Funktionen sind genau die partiellen berechenbaren Funktionen.*

Beweis. Wir zeigen die Äquivalenz zu While-Programmen:

Den μ–Operator durch ein While Programm zu implementieren, ist einfach:

```
n := 0 ;
while f(n, x_1, ..., x_k) ≠ 0 do
        n := n + 1
```

Umgekehrt sei f durch ein While-Programm gegeben. Nach dem Satz von Kleene (5.5.1) können wir annehmen, dass dieses die Form

P ; while B do Q

besitzt, wobei in P und in Q kein *while* vorkommt. Wir müssen eine μ–rekursive Funktion angeben, die die Semantik des Programms beschreibt. Die Idee ist, mit Hilfe des μ-Operators die Anzahl der benötigten Schleifendurchläufe zu finden, und dann die While-Schleife durch eine *For*-Schleife zu ersetzen, von der wir wissen, dass sie primitiv rekursiv ausdrückbar ist.

Konkret sei $g(n, x_1, \ldots, x_n)$ die semantische Funktion von

for i := 0 to n do Q

und $f_B(x_1, \ldots, x_k)$ die semantische Funktion der Bedingung B. Dann gilt für

$$h(n, x_1, \ldots, x_k) := f_B(g(n, x_1, \ldots, x_k)) = \begin{cases} 1 & \text{falls } B \text{ wahr nach } n \text{ mal } P \\ 0 & \text{sonst,} \end{cases}$$

dass $\mu(h)(x_1, \ldots, x_k)$ die in der Schleife 5.6.7 benötigten Durchläufe berechnet. Folglich ist

while B do P

äquivalent zu

for $i := 0$ to $\mu(h)(x_1, \ldots, x_k)$ do P

und daher durch eine μ–rekursive Funktion berechenbar.

Wieder einmal haben wir die Churchsche These bestätigt, dass jedes „vernünftige“ Berechenbarkeitskonzept genau die partiell berechenbaren Funktionen definiert. Zusammenfassend könne wir also den Begriff eines Algorithmus mit einem von vielen konkurrierenden Konzepten gleichsetzen. Ob wir Algorithmen mit Turingtabellen, mit Goto- oder While-Programmen, oder mit μ–rekursiven Funktionen gleichsetzen, immer erhalten wir die gleiche Semantik, in dem Sinne, dass die partiellen berechenbaren Funktionen für alle diese Konzepte übereinstimmen. □

5.7 Grenzen der Berechenbarkeit

Wir wissen bereits, dass es Funktionen $f : \mathbb{N} \to \mathbb{N}$ geben muss, die nicht berechenbar sind. Der Grund ist, kurz gesagt, dass es zu viele zahlentheoretische Funktionen $f : \mathbb{N} \to \mathbb{N}$ gibt, nämlich überabzählbar viele und zu wenige berechenbare, nämlich nur abzählbar viele.

Schließlich ist jede berechenbare Funktion durch einen endlichen Text beschreibbar – Turingmaschinen durch eine endliche Turingtabelle, While-Programme durch einen endlichen Programmtext, Rekursive Funktionen durch eine endliche Konstruktion, die mit den Basisfunktionen beginnt und eine endliche Kombination von Komposition, primitive Rekursion und μ–Rekursion beschreibt.

Alle Konzepte haben gemeinsam, dass sich jedes „Programm“ nach Wahl eines geeigneten Alphabets Σ, z.B. ASCII, als endlicher Textstring $s \in \Sigma^*$ darstellen lässt. Da es aber nur abzählbar viele Elemente von Σ^* gibt, (siehe 5.2.5) ist die Menge aller partiellen berechenbaren Funktionen (damit erst recht die Menge aller totalen berechenbaren Funktionen) höchstens abzählbar.

Sei $\mathbb{N}^{\mathbb{N}}$ die Menge aller Abbildungen $g : \mathbb{N} \to \mathbb{N}$ von $\mathbb{N}$ nach $\mathbb{N}$. Diese Menge ist überabzählbar. Schon die Teilmenge derjenigen Funktionen, die nur die Ergebnisse 0 oder 1 haben dürfen, also die Menge aller Funktionen $\chi : \mathbb{N} \to \{0, 1\}$, kurz $\{0, 1\}^{\mathbb{N}}$, ist überabzählbar, denn sie entspricht eineindeutig der Menge $\mathbb{P}(\mathbb{N})$ aller Teilmengen von $\mathbb{N}$:

Jeder Funktion $\chi : \mathbb{N} \to \{0, 1\}$ können wir die Teilmenge $U_\chi = \{n \in \mathbb{N} \mid \chi(n) = 1\}$ zuordnen und umgekehrt jeder Menge $U \subseteq \mathbb{N}$ die *charakteristische Funktion*

$$\chi_U(n) := \begin{cases} 1 & n \in U \\ 0 & \text{sonst.} \end{cases}$$

Diese Idee lag ja schon Cantor's Diagonalargument (siehe Fig.5.1.1) zugrunde. Aus dem Satz von Cantor folgt insbesondere, dass $|\mathbb{N}| < |\mathbb{P}(\mathbb{N})|$ ist, also insgesamt $|\Sigma^\star| = |\mathbb{N}| < |\mathbb{P}(\mathbb{N})| = |\{0, 1\}^{\mathbb{N}}| \leq |\mathbb{N}^{\mathbb{N}}|$. Folglich muss es Funktionen $f \in \mathbb{N}^{\mathbb{N}}$ geben, die nicht berechenbar sind.

Aus Anzahlgründen muss es also nicht berechenbare Funktionen geben. Wir möchten in diesem Abschnitt eine solche Funktion finden. Ackermann's Funktion taugt übrigens nicht als Beispiel – sie ist berechenbar, wenn auch nicht LOOP-berechenbar.

5.7.1 Aufzählbarkeit, Entscheidbarkeit und Semi-Entscheidbarkeit

Die Begriffe „aufzählbar" und „entscheidbar" hatten wir informell bereits benutzt. Nach der jetzt nachgelieferten mathematisch sauberen Definition des Begriffs *Algorithmus* können wir sie jetzt auch als mathematisch exakte Definitionen betrachten.

Definition 5.7.1. Eine Menge $U \subseteq \Sigma^\star$ heißt *aufzählbar*, wenn es eine surjektive berechenbare Funktion $f : \mathbb{N} \twoheadrightarrow U$ gibt.

U ist also genau dann aufzählbar, wenn es einen Algorithmus gibt, der genau alle Elemente von U produzieren kann, während der Input alle natürlichen Zahlen durchläuft. Wenn wir unserem WHILE-Programmen eine *Print*-Anweisung spendieren, könnten wir alle Elemente von U durch ein Programm ausdrucken lassen:

`n := 0 ; While true do { Print( f(n)) ; n := n + 1 }.`

Es ist nicht verlangt, dass die Elemente von U in einer bestimmten Reihenfolge erzeugt werden, auch darf ein Element mehrfach erscheinen.

Wir könnten dieses Programm auch benutzen, um festzustellen, *ob* ein bestimmtes Element x zu U gehört. Wir produzieren der Reihe nach $f(0), f(1), f(2), \ldots$ und schreiben sie in die Ausgabe. Sobald x in der Ausgabe auftritt, stoppen wir und geben als Antwort auf die Frage „Ist $x \in U$?" die Antwort 1 aus. Falls x nicht in U ist, werden wir ewig warten. Dieser Algorithmus berechnet daher die partielle Funktion

$$\delta_U(x) = \begin{cases} 1 & x \in U \\ \bot & \text{sonst.} \end{cases} \tag{5.7.1}$$

Sie gibt uns eine positive Antwort, falls $x \in U$ ist. Falls $x \notin U$, erhalten wir nie eine Antwort, da der Algorithmus dann nicht terminiert. Wir nehmen dies als Anlass für folgende Definition:

Definition 5.7.2. Eine Menge $U \subseteq \Sigma^*$ heißt *erkennbar* (oder *semi-entscheidbar*), falls die partielle Funktion δ_U aus Gleichung 5.7.1 berechenbar ist.

In der Praxis hat man oft Verfahren ν_U, die nicht nur erfolgreich feststellen, ob ein Element x in der Menge U ist, sondern auch im Falle, dass $x \notin U$ ist, dies oft nach endlicher Zeit korrekt feststellen, also $\nu_U(x) = 1$ gdw. $x \in U$ und $\nu_U(x) = 0$ oder $\nu_U(x) = \perp$ sonst. Aus einem derartigen ν_U kann man selbstverständlich leicht die in Definition 5.7.2 geforderte Funktion δ_U erzeugen, denn $\delta_U(x) = ite(\nu_U(x) = 1, 1, \Omega(x))$, wobei Ω die überall nicht definierte partielle Funktion ist.

Nach dem bereits oben gesagten ist jede aufzählbare Menge erkennbar. Es gilt aber auch die Umkehrung.

Satz 5.7.3. *Eine Menge ist $U \subseteq \Sigma^*$ ist genau dann aufzählbar, wenn sie erkennbar ist.*

Beweis. Sei U erkennbar, dann gibt es einen Algorithmus $D_U(x)$, der $\delta_U(x)$ berechnet. Wir können D_U als While-Programm in Kleene'scher Normalform (5.5.1) aufschreiben:

$D_U(x) =$

```
I;
while B do S
```

wobei I, B und S Loop-Programme sind, in denen der Input-Parameter x vorkommen kann. Nun modifizieren wir D_U zu einem LOOP-Programm $D_U(n, x)$, das immer dann mit $D_U(x)$ übereinstimmt, wenn die Berechnung von $D_U(x)$ höchstens n Schleifendurchläufe benötigt:

$D_U(n, x) =$

```
I;
for k:=0 to n do
    if B then S;
if not(B) then write(x)
```

□

Schließlich lassen wir die Schranke n wachsen und erhalten ein Programm, das alle Elemente x aus U aufzählt:

```
n:= 0 ;
while true do
    for k := 0 to n do D_U(n, k);
    n := n + 1
```

Die Aufzählung ist offensichtlich nicht injektiv. Jedes Element von U wird sogar unendlich oft ausgegeben. Diesen Schönheitsfleck könnten wir beheben, wenn wir vor der Ausgabe von x kontrollieren würden, ob das Element schon früher einmal gefiunden wurde. Was wir jedoch prinzipiell nicht beheben können, ist der Mangel, dass die Aufzählung der Elemente von U nicht in der Reihenfolge geschieht, in der sie ursprünglich erzeugt wurden, z.B. nicht der Größe nach. Die Reihenfolge in der die

Elemente von U gefunden werden, hängt von der Anzahl der Schritte ab, die für die Berechnung von $\delta_U(x)$ benötigt werden.

Definition 5.7.4. $U \subseteq \Sigma^\star$ heißt *entscheidbar*, wenn die charakteristische Funktion (Seite 169) χ_U mit

$$\chi_U(x) = \begin{cases} 1 & x \in U \\ 0 & \text{sonst} \end{cases}$$

berechenbar ist.

Ist Uentscheidbar, dann ist offensichtlich auch $\Sigma^\star \setminus U$ entscheidbar, denn $\chi_{\Sigma^\star \setminus U} = not \circ \chi_U$. Außerdem ist U auch erkennbar, denn

$$\delta_U(x) = \text{if } \chi_U(n) = 1 \text{ then } 1 \text{ else } \Omega(n)$$

ist auf jeden Fall berechenbar. Folglich ist auch $\Sigma^\star \setminus U$ erkennbar. Der zu *erkennbar* synonyme Begriff „*semi-entscheidbar*“ wird jetzt durch den folgenden Satz gerechtfertigt:

Satz 5.7.5. *Eine Menge U ist genau dann entscheidbar, wenn sowohl U als auch ihr Komplement $\bar{U} = \Sigma^\star - U$ semi-entscheidbar sind.*

Beweis. In den Semi-Entscheidungsalgorithmen $D_U(x) = I_U$; while B_U do S_U für U und $D_{\bar{U}}(x) = I_{\bar{U}}$; while $B_{\bar{U}}$ do $S_{\bar{U}}$ für $\bar{U}$ beschränken wir ähnlich wie vorhin die Anzahl der Schleifendurchläufe durch n und erhalten die Algorithmen $D_U(x, n)$ und $D_{\bar{U}}(x, n)$. Der folgende Algorithmus lässt $D_U(x)$ und $D_{\bar{U}}(x)$ jeweils n mal ihre Schleifenkörper ausführen. Falls $D_U(x, n)$ dann beendet ist, wird das Ergebnis 1 ausgegeben, falls $D_{\bar{U}}(x, n)$ dann beendet ist, wird 0 ausgegeben:

```
n := 0;
while true do
  D_U(x, n) ; if not(B_U) return 1
  D_Ū(x, n) ; if not(B_Ū) return 0
```

□

Beispiel 5.7.6. Die folgenden Mengen sind aufzählbar:
- Jede endliche Menge $U \subseteq \mathbb{N}$.
- Die Menge aller Primzahlen.
- Die Menge aller Teilworte in der Dezimalentwicklung von π.
- Die Menge $C = \{n \mid collatz(n) = 1\}$ (siehe Seite 168).

In den ersten beiden Fällen ist die Aufzählbarkeit anhand der Definition unmittelbar ersichtlich. Die Aufzählung aller Teilworte von π ist nicht so einfach, da wir alle Teilworte jeder Länge sehen wollen. Der offensichtliche Test, der $n = (d_1 \ldots d_k)_{10}$ als Teilwort in π sucht, liefert im Erfolgsfall die richtige Antwort. Falls n kein Teilwort von π

ist, verfängt er sich in einer Endlosschleife. Satz 5.7.3 zeigt uns, dass wir uns dennoch einen Aufzählungsalgorithmus bauen können.

Für das Collatz-Problem des letzten Beispiels starten wir einfach die Berechnung von $collatz(n)$. Falls $n \in C$ ist, erscheint irgendwann das Ergebnis 1 ansonsten terminiert unsere Rechnung nicht, wir berechnen also exakt $\delta_C(n)$. Möglicherweise ist δ_C sogar die konstante Funktion mit Wert 1, aber das ist bisher noch unbekannt.

5.7.2 Gödelisierungen

Ein interessantes Beispiel einer entscheidbaren Menge ist die Menge aller partiell berechenbaren Funktionen selber. Wir können irgendeines der besprochenen Algorithmenkonzepte wählen, beispielsweise die Registermaschinen. Zu jedem gegebenen Text $w \in ASCII^*$ können wir entscheiden, ob es sich bei w um ein syntaktisch korrektes RM-Programm handelt oder nicht. Damit ist die Menge aller RM-Programme entscheidbar, insbesondere aufzählbar, wir können also eine Auflistung $P_1, P_2, \ldots, P_n, \ldots$ aller RM-Programme herstellen. Für beliebiges k können wir jedes der Programme P_i als Codierung einer berechenbaren Funktion $\varphi_i^{(k)} : \mathbb{N}^k \to \mathbb{N}$ deuten:

Vor Beginn der Ausführung legen wir die k Argumente $x_1, \ldots, x_k$ in den Registern $c[1], \ldots, c[k]$ ab. Dann starten wir P_i. Falls das Programm hält, lesen wir das Ergebnis aus Register $c[1]$, also $\varphi_i^{(k)}(x_1, \ldots, x_k) = c[1]$, falls es nicht hält schreiben wir: $\varphi_i^{(k)}(x_1, \ldots, x_k) = \perp$.

Jede partiell berechenbare Funktion $f : \mathbb{N}^k \to \mathbb{N}$ wird durch mindestens ein P_i realisiert. i heißt dann auch *Gödelnummer* der Funktion f. Da man f auf unendlich viele verschiedene Weisen programmieren kann, hat jede partielle berechenbare Funktion mehrere, sogar unendlich viele, Gödelnummern. Insbesondere gilt:

Satz 5.7.7. *Die partiellen berechenbaren (k-stelligen) Funktionen sind aufzählbar.*

Eigentlich wären wir mehr an den totalen berechenbaren Funktionen interessiert. Doch einen entsprechenden Satz kann man für diese nicht beweisen. Es gilt sogar:

Satz 5.7.8. *Die totalen berechenbaren (k-stelligen) Funktionen sind nicht aufzählbar.*

Beweis. Wir benutzen wieder Cantors Diagonaltrick: Angenommen, $\psi_1, \psi_2, \psi_3, \ldots$ sei eine Aufzählung der totalen berechenbaren (1-stelligen) Funktionen. Betrachte jetzt die Funktion

$$\Delta(n) := \psi_n(n) + 1.$$

Δ ist offensichtlich total und berechenbar: Um $\Delta(n)$ zu berechnen, starten wir die hypothetische Aufzählung der totalen berechenbaren Funktionen, bzw. deren Algorithmen. Sobald wir zum Algorithmus P_n gekommen sind, starten wir P_n mit Input n. Wenn P_n fertig ist, addieren wir zum Ergebnis noch 1. Da Δ total und berechenbar ist,

muss es ein i geben mit $\psi_i = \Delta$. Dies liefert aber sofort den Widerspruch

$$\psi_i(i) = \Delta(i) = \psi_i(i) + 1.$$

Dieses Ergebnis ist verheerend. Man kann kein vernünftiges Algorithmenkonzept definieren, das genau die totalen berechenbaren Funktionen liefert. Die „primitiv rekursiven Funktionen“ stellen zwar ein vernünftiges Algorithmenkonzept dar, sie erfassen aber nicht alle totalen berechenbaren Funktionen, wie wir am Beispiel der Ackermannfunktion gesehen haben. □

Aufgabe 5.7.9. Begründen Sie, warum der Diagonaltrick nicht mit den partiellen berechenbaren Funktionen funktioniert.

5.7.3 Das Halteproblem

Wir haben mehrfach betont, dass schon aus Anzahlgründen Funktionen $f : \mathbb{N} \to \mathbb{N}$ existieren müssen, die nicht berechenbar sind. Jetzt wollen wir eine konkrete nicht berechenbare Funktion $h : \mathbb{N} \to \mathbb{N}$ kennenlernen. Wir werden die Funktion aus einer konkreten Problemstellung gewinnen.

Das Problem, zu bestimmen, ob ein Algorithmus für einen bestimmten Input halten wird, oder nicht, ist durchaus praktisch relevant. Wenn Sie ein Programm in einer höheren Programmiersprache fertiggestellt haben, kann der Compiler feststellen, ob Ihr Programm syntaktisch korrekt ist, oder nicht. Das bedeutet, dass die Menge aller syntaktisch korrekten Java/C/Python/While-Programme entscheidbar ist. Das ist einer der Gründe, warum man sich bei der Spezifikation der Syntax einer Programmiersprache üblicherweise auf kontextfreie Sprachen beschränkt (siehe Satz 3.6.3).

Nicht entscheidbar, und daher niemals als feature eines Compilers erhältlich, wird eine Auskunft sein, ob ein Programm halten wird. Da Programme üblicherweise einen Input verarbeiten, formulieren wir das Halteproblem etwas allgemeiner.

Halteproblem: *Gegeben ein Algorithmus A und ein Input n. Wird A mit Input n terminieren oder nicht?*

Selbstverständlich könnten wir den Algorithmus A mit Input n starten und warten, was passiert. Terminiert die Ausführung, dann werden wir irgendwann wissen, dass $A(n)$ hält. Hält das Programm nach einer bestimmten Zeit (einer Sekunde, einem Tag, einem Jahr, ...) noch nicht, dann wissen wir nicht, ob es daran liegt, dass wir immer noch nicht lange genug gewartet haben, oder ob das Programm in einer Endlosschleife gefangen ist.

Jeder Computerbenutzer kennt die Situation, dass der Rechner nicht mehr reagiert. Dann stellt sich immer die Frage: Soll ich noch etwas warten, vielleicht geht es dann weiter, oder ist es sinnlos und ich muss den Rechner oder das Programm neu starten. Das ist ein praktischer Aspekt des Halteproblems.

Aber warum soll das Problem nicht lösbar sein? Wenn wir die Unlösbarkeit beweisen wollen, müssen wir zeigen, dass es keinen Algorithmus gibt, der dieses Problem löst. Das ist eine Aussage über *alle möglichen Algorithmen*. Hier zahlt es sich aus, dass wir den Begriff des Algorithmus präzise definiert haben, sonst könnten wir eine solche Aussage gar nicht treffen.

Die mit dem Halteproblem assoziierte mahematische Funktion $hält(m, n)$ können wir mathematisch folgendermaßen definieren:

$$hält(m, n) := \begin{cases} 0 & \varphi_m(n) = \bot \\ 1 & \text{sonst} \end{cases}.$$

Satz 5.7.10. *Die Funktion* $hält : \mathbb{N}^2 \to \mathbb{N}$ *ist eine totale Funktion, die nicht berechenbar ist.*

Beweis. Wieder verwenden wir Cantors Trick. Wenn $hält$ berechenbar wäre, dann sicher auch die folgende Funktion Δ:

$$\Delta(n) := \text{if } hält(n, n) = 1 \text{ then } \Omega(n) \text{ else return } 1.$$

Dann müsste Δ eine Gödelnummer besitzen, zum Beispiel k. Die k-te partielle berechenbare Funktion P_k würde also Δ berechnen. Wir wenden Δ auf k an und erhalten den Widerspruch:

$$\begin{aligned} \Delta(k) \neq \bot &\iff P_k(k) \neq \bot \\ &\iff \varphi_k(k) \neq \bot \\ &\iff hält(k, k) = 1 \\ &\iff \Delta(k) = \Omega(k) \\ &\iff \Delta(k) = \bot. \end{aligned}$$

□

Die Funktion $g(n) = hält(n, n)$, mit der der obige Beweis arbeitet, nennt man auch das „spezielle Halteproblem“. Sie beantwortet die Frage, ob die n-te berechenbare Funktion angewendet auf ihre eigene Gödelnummer stoppt oder nicht.

Mit der „konkreten“ nicht berechenbaren Funktion $hält$ gewinnen wir jetzt auch entsprechend konkrete Beispiele nicht entscheidbarer und nicht aufzählbarer Mengen.

Korollar 5.7.11. *Die Menge* $H = \{n \in \mathbb{N} \mid hält(n, n) = 1\}$ *ist nicht entscheidbar.*

Beweis. Wäre H entscheidbar, so wäre δ_H berechenbar. Aber

$$\delta_H(n) = \begin{cases} 1 & hält(n, n) = 1 \\ 0 & \text{sonst} \end{cases} = hält(n, n)$$

ist nicht berechenbar. □

Da H nicht entscheidbar ist, folgt aus Satz 5.7.5, dass entweder H oder $\bar{H} := \mathbb{N} \setminus H$ nicht aufzählbar sein kann. Offensichtlich ist aber H erkennbar, also semi-entscheidbar. Es folgt für das Komplement:

Korollar 5.7.12. *$\bar{H}$ ist nicht aufzählbar.*

5.7.4 Der Satz von Rice

Das Halteproblem ist die bekannteste und wichtigste Frage, die man algorithmisch nicht lösen kann. Die Unentscheidbarkeit dieses Problems wurde 1936 von Alan Turing bewiesen. In der Folge wurden eine Reihe anderer Probleme identifiziert, die ebenfalls algorithmisch nicht lösbar sind. Der übliche Beweisgang war die Rückführung des Problems P auf das Halteproblem: Aus einem hypothetischen Algorithmus für P konstruierte man einen neuen Algorithmus, der das Halteproblem lösen würde. Beispielsweise kann man für Algorithmen bzw. berechenbare Funktionen nicht entscheiden

- ob sie eine Ausgabe erzeugen
- ob sie bei mindestens einem Input halten
- ob sie eine konstante Funktion berechen
- ob sie mit einer gegebenen Funktion übereinstimmen.

1951 gelang es Henry G. Rice zu zeigen, dass nicht nur die obigen, sondern alle interessanten nichttrivialen Eigenschaften von Programmen unentscheidbar sind. Genauer definierte er:

Definition 5.7.13. Eine *semantische Eigenschaft* von Programmen ist eine Eigenschaft E, die nicht von der Darstellung der Programme abhängt, sondern nur von der von ihnen berechneten Funktion: falls zwei Programme P_i und P_j die gleiche Funktion berechnen, soll gelten: $E(P_i) \iff E(P_j)$.

Eine Eigenschaft E heißt trivial, falls sie für jeden oder für keinen Algorithmus zutrifft. Alle oben erwähnten Eigenschaften, wie auch das Halteproblem sind nichttriviale semantische Eigenschaften. Somit werden sie von folgendem Satz erschlagen:

Satz 5.7.14 (Satz von Rice)**.** *Jede nichttriviale semantische Eigenschaft E von Algorithmen ist unentscheidbar.*

Beweis. Sei A_Ω der Algorithmus, der die überall undefinierte partielle Funktion Ω berechnet. Wir können annehmen, dass A_Ω die Eigenschaft E nicht erfüllt, ansonsten würden wir die komplementäre Eigenschaft $\neg E$ betrachten. Da E nichttrivial ist, gibt es einen Algorithmus B, der E erfüllt. Angenommen, E wäre entscheidbar, dann könnten wir das Halteproblem $hält(m, n)$ folgendermaßen lösen:

Wir erzeugen (algorithmisch) den folgenden Algorithmus $A(m, n)$, den wir als While-Programm formuliert haben.

$$x_1 := n;\ P_m\ ;\ x_1 := n\ ;\ B$$

Dabei nutzen wir aus, dass die berechenbaren Funktionen aufzählbar sind: Aus m kann uns die Aufzählung den Programmtext P_m erzeugen, aus m und n können wir also automatisch den Algorithmus $A(m, n)$ gewinnen.

Nun gilt: Wenn P_m mit Input n hält, ist der Algorithmus semantisch äquivalent zu B, er erfüllt also E.

Wenn P_m mit Input n nicht hält, ist der Algorithmus äquivalent zu A_Ω, er erfüllt also nicht E. Insgesamt erfüllt er also genau dann E, wenn $P_m(n)$ hält. Könnten wir E entscheiden, so könnten wir auf diese Weise auch $hält(m, n)$ berechnen. □

Wir beenden dieses Kapitel mit einer Übersicht der wichtigsten Funktionenklassen, die wir besprochen haben.

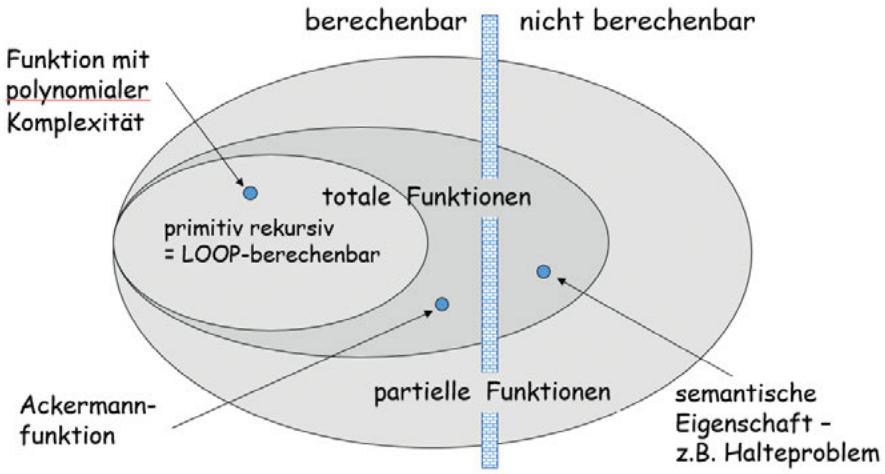


Abb. 5.7.1: Klassen berechenbarer Funktionen

Kapitel 6

Komplexität

6.1 Probleme und Sprachen

In den letzten beiden Kapiteln haben wir viele Entscheidungsprobleme diskutiert. Einige davon waren unlösbar. Dazu gehört beispielweise das Problem festzustellen, ob P_n, das Programm mit Gödelnummer n mit Input n halten wird oder nicht, oder auch die Frage, ob P_i und P_j die gleiche Funktion berechnen.

Andere Probleme sind lösbar, etwa das Problem festzustellen, ob ein deterministischer oder nichtdeterministischer Automat A ein Wort w erkennt, ob also $w \in L(A)$ ist. Auch für eine kontextfreie Grammatik G war das Problem, zu entscheiden, ob $w \in L(G)$ oder $w \notin L(G)$, lösbar.

Gegen Ende des letzten Kapitels sind wir aber auch auf Probleme gestoßen, die zwar theoretisch lösbar sind, praktisch aber nur für sehr kleine Inputwerte handhabbar sind. Dazu gehörte beispielsweise die Berechnung der Ackermannfunktion, die bereits für einen kleinen Input wie $(m, n) = (4, 2)$ eine Zahl mit ca. 20000 Dezimalstellen liefert.

Wir waren bisher dennoch damit zufrieden, festzustellen, *ob* ein Problem lösbar ist, oder nicht. Die Ackermannfunktion ist für beliebige positive Eingaben zwar theoretisch lösbar, praktisch ist das aber nicht der Fall. Daher wollen wir uns in diesem Kapitel damit beschäftigen eine Grenze zwischen theoretischer und praktischer Lösbarkeit einzuziehen. Was ist also theoretisch lösbar, in der Praxis aber nur für kleine Eingaben.

Die Theorie der Berechenbarkeit befasst sich mit der Frage, welche Funktionen prinzipiell berechenbar sind, unabhängig vom Aufwand, der dafür zu treiben ist. In diesem Kapitel wollen wir untersuchen, welche Funktionen *effizient* berechenbar sind.

Im Allgemeinen wird der zeitliche Aufwand für die Ausführung eines Algorithmus von dessen Input abhängen. Wenn der Berechnungsaufwand exponentiell mit der Größe der Argumente ansteigt, dann werden wir den Algorithmus als *nicht effizi-*

https://doi.org/10.1515/9783110442397-007

ent einstufen. Hier geht es also nicht um Halten oder nicht Nichthalten sondern allein um Effizienz. Daher werden wir hier nur totale Funktionen betrachten.

6.1.1 Generate and Test

Triviale Algorithmen beinhalten oft ein „systematisches Probieren“. Sie bestehen aus zwei Phasen, die parallel oder zeitlich verschränkt ablaufen, und die man mit *Generate and Test* charakterisieren kann. Abstrakt formuliert man einen solchen Algorithmus als Suche nach einer Lösung zu einem gegebenen Problem. Die Lösungsmethode könnte abstrakt so dargestellt werden:
Generate: erzeuge Lösungskandidaten
Test: prüfe, ob einer der Kandidaten eine Lösung ist.
Dies erscheint eine überaus naive Vorgehensweise zu sein, und obwohl sie für theoretische Zwecke oft genügt, ist sie für praktische Zwecke meist nicht zu gebrauchen. In diesem Kapitel werden wir aber sehen, dass es eine Klasse von Problemen gibt, für die vermutlich kein besserer Lösungsalgorithmus existiert. Die Einschränkung „vermutlich“ weist darauf hin, dass dahinter ein bis dato ungelöstes mathematisches Problem steht. Die Vermutung wird aber durch die Gewissheit genährt, dass die effiziente Lösung eines dieser Probleme zu einer ebenso effizienten Lösung von allen dieser Probleme führen würden. Kurz: könnte man eines der Probleme effizient lösen, so könnte man alle effizient lösen.

Ein wichtiges Beispiel ist eine Variante des sogenannten *Bin-Packing* Problems. Es geht darum, Pakete verschiedener Größe in einem großen Karton unterzubringen. Viele Menschen würden folgendermaßen vorgehen: Probiere eine Reihenfolge, in der die Pakete in den Karton gepackt werden. Teste, ob er noch schließt. Selbstverständlich gibt es eine Reihe von Daumenregeln, vornehm gesprochen *Heuristiken* für dieses Problem. Beispielsweise kann man mit den größten Paketen beginnen und versuchen, die verbliebenen Zwischenräume mit kleineren Paketen zu füllen. Wie alle Heuristiken hat auch diese den Nachteil, nicht immer zu funktionieren.

6.1.2 Das SAT-Problem

Ein Dreh- und Angelpunkt für die erwähnte Problemklasse ist das sogenannte SAT-Problem. Dieses besteht darin, zu einer aussagenlogischen Formel, wie beispielsweise

$$x_1 \vee (\neg x_2 \implies (x_3 \wedge x_4)) \wedge ((x_2 \wedge \neg x_3 \wedge \neg x_4) \implies x_1) \implies \neg x_1 \vee \neg(x_3 \vee \neg x_4)$$

zu entscheiden, ob diese Formel *erfüllbar* (engl. *satisfiable*) ist, ob es also eine Möglichkeit gibt, den Variablen $x_1, x_{2,} x_3, x_{4,} \ldots$ Wahrheitswerte $true$ bzw. $false$ zuzuordnen, so dass der resultierende Term insgesamt zu *true* evaluiert.

Im Beispiel haben wir eine Formel mit $n = 4$ Variablen und wir suchen eine Kombination von Werten $x_1, x_2, x_3, x_4 \in \{true, false\}$, die die Formel insgesamt wahr macht. Statt $true$ und $false$ benutzen wir der Kürze halber in Zukunft die Binärzahlen

0 und 1. Wir werden diese und andere Probleme in diesem Kapitel prägnant mit den Schlagworten „Gegeben/Gefragt“ beschreiben. Im Falle des SAT-Problems:

Gegeben: Eine Formel $F(x_1, \ldots, x_n)$ der Aussagenlogik mit Variablen $x_1, \ldots, x_n$ und den logischen Operatoren $\wedge, \vee, \neg, \rightarrow$.

Gefragt: Gibt es eine Belegung der Variablen $x_1 = b_1, \ldots, x_n = b_n$ mit Wahrheitswerten $b_i \in \{0, 1\}$, so daß $F(b_1, \ldots, b_n) = 1$ ist.

Eine naive Lösung des SAT-Problems, formuliert als Generate/Test-Schema ist:

Generate: Erzeuge alle potientiellen Kandidaten, d.h. alle $(b_1, \ldots, b_n) \in \{0, 1\}^n$.

Test: Prüfe durch Einsetzen und Auswerten, ob ein Kandidat eine Lösung ist.

Offensichtlich wächst die Anzahl der Kandidaten exponentiell mit n, der Anzahl der Variablen, während jeder Test, ob es sich bei einem Kandidaten um eine Lösung handelt, nur von der Größe der Formel abhängig ist. Man muss nur jede Variable x_i durch den entsprechenden Wert b_i des Lösungskandidaten ersetzen und den Wahrheitswert des so entstandenen booleschen Ausdrucks ausrechnen.

SAT-Solver, die heute standardmäßig Fragestellungen aus der industriellen Praxis bearbeiten, kommen mit hunderttausenden von Variablen zurecht. Dafür sind gute Heuristiken essentiell. Andererseits gibt es auch schlecht konditionierte Probleme mit nur wenigen hundert Variablen, an denen sich heutige Systeme noch die Zähne ausbeißen.

6.1.3 CLIQUE:

Definition 6.1.1. Ein (gerichteter) Graph $G = (V, E)$, besteht bekanntlich aus einer Grundmenge V und einer Relation $E \subseteq V \times V$.

Die Elemente aus V heißen *Knoten* (engl. *vertex*), und die Elemente $(x, y) \in E$ heißen Kanten (engl.: *edge*), dabei ist x der Startpunkt und y der Zielpunkt der Kante $k = (x, y)$.

Ein Graph G heißt *ungerichtet*, falls mit $(x, y) \in E$ immer auch $(y, x) \in E$ ist.

Graphen sind anschauliche Gebilde, weil man die Knoten $x, y, \ldots$ als kleine Kreise darstellen kann und die Kanten (x, y) als Pfeile von x nach y. Für einen ungerichteten Graphen stellt man die Kanten durch Linien zwischen zwei Punkten dar.

Beispiele für die Anwendung von Graphen in der Informatik gibt es viele und mit der Einführung der sozialen Netze ist auch die Analyse riesiger Graphen mit Millionen von Knoten interessant geworden. Mit Graphen kann man z.B. soziale Netzwerke modellieren. Die Knoten entsprechen den Teilnehmern und die Kanten können beispielsweise die Relation „kennt“ oder „hat Kontakt mit“ beschreiben. Interessant sind dabei u.a. Fragen zu Gruppenbildungen, zum Abstand ausgewählter Personen, welche Personen besonders viele Kontakte haben, etc..

Eine *Clique* ist allgemein eine Teilmenge C der Knoten des Graphen G, so dass je zwei Elemente aus C mit einer Kante verbunden sind. Kurz: $C \subseteq V$ und $\forall x, y \in$

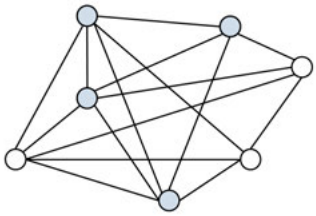


Abb. 6.1.1: Ungerichteter Graph mit 4-Clique

$C.(x, y) \in E$. Eine n-Clique ist eine Clique C mit n Elementen. Abbildung 6.1.1 zeigt einen Graphen mit einer 4–Clique, deren Knoten grau hervorgehoben sind.

Die in der Figur dargestellte 4-Clique ist nicht die einzige, es gibt eine weitere 4-Clique, und ob es eine 5-Clique gibt, ist nicht so leicht zu beantworten. Das Cliquenproblem ist genau die Frage, ob es in einem einem Graphen G zu vorgegebenem k eine k- Clique gibt, kurz:

Gegeben: Ein ungerichteter Graph G mit n Elementen und eine Zahl $k \in \mathbb{N}$
Gefragt: Besitzt der Graph G eine k-Clique?

Auch hier liegt eine naive Lösung auf der Hand:
Generate: Erzeuge alle k-elementigen Knotenmengen $X \subseteq V$.
Test: Sind je zwei Knoten in X verbunden?

Diesmal gibt es für einen Graphen mit n Elementen $\binom{n}{k} = \frac{n!}{k!(n-k)!}$ viele k-elementige Teilmengen, die als Lösungskandidaten in Frage kommen, während auch hier wieder jeder Test vergleichsweise schnell ist: Es ist jedesmal das Vorhandensein von $k(k-1)$ Kanten zu überprüfen.

6.1.4 Travelling Salesman

Die Geschichte hinter dem Problem erzählt von einem Handlungsreisenden, der alle Städte in einem Gebiet besuchen muss (siehe auch Band 1, ab Seite 430). Dazu fährt er von seiner Heimatstadt los, besucht jede andere Stadt und kehrt am Abend wieder nach Hause zurück. Er hat also eine Rundreise unternommen. Der Handlungsreisende besitzt eine Karte mit den Straßenverbindungen zwischen je zwei Städten und deren jeweilige Entfernungen. Gefragt ist, ob es eine Rundreise gibt, bei der alle Städte besucht werden, und die eine maximale Gesamtstrecke der Länge k Kilometer nicht überschreitet.

Für die Modellierung führen wir den Begriff des bewerteten Graphen $G = (V, d)$ ein. Auch hier ist V eine Menge von Knoten, aber

$$d : V \times V \to \mathbb{N}^{\infty}$$

ist eine Abbildung in die Menge $\mathbb{N} \cup \{\infty\}$.

Dabei soll $d(x, y)$ üblicherweise den kürzesten direkten Abstand zwischen zwei Knoten x und y darstellen. Falls es keine Kante zwischen x und y gibt, modelliert man dies mit der Bewertung $d(x, y) = \infty$. Falls $d(x, y) = d(y, x)$ für alle Knoten x, y gilt, nennen wir den bewerteten Graphen ungerichtet. Die Länge einer Rundreise berechnet man als Summe der Distancen zwischen zwei nacheinander besuchten Knoten.

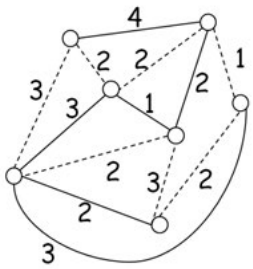


Abb. 6.1.2: Travelling Salesman Problem mit Rundreise der Länge 15

Das Travelling Salesman Problem (TSP) können wir ähnlich wie die vorigen Probleme so zusammenfassen:

Gegeben: Ein bewerteter Graph und ein Wert $n \in \mathbb{N}$
Gefragt: Gibt es eine Rundreise mit Länge höchstens n?

Auch hier hat man die triviale Generate/Test-Lösung

Generate: Alle möglichen Rundreisen,
Test: Ist die Länge einer Rundreise kleiner oder gleich n?

6.1.5 Minimierungsprobleme und Lösungssuche

Während das SAT-Problem ein reines ja/nein-Problem ist — gibt es eine Belegung oder gibt es keine – werden das Bin-Packing Problem wie auch das TSP oft als Optimierungsproblem formuliert: man soll Pakete so in den Container packen, dass das transportierte Gesamtvolumen maximal wird, bzw man soll eine Rundreise minimaler Länge finden. Betrachten wir beispielsweise den letzteren Fall, so können wir eine beliebige Rundreise auswählen und deren Länge m ermitteln. Die Länge einer kürzesten Rundreise muss also zwischen dem unteren Wert $u = 0$ und dem oberen Wert $v = m$ liegen. Sei $(u+v)/2$ die Mitte des Intervalls, dann können wir das ja/nein-Problem mit $n = (u+v)/2$ formulieren. Als Ergebnis erhalten wir die Information, ob der minimale Wert in der linken oder der rechten Intervallhälfte liegt. Nach spätestens $log_2 n$ vielen Aufrufen des ja/nein-Problems ist dann auch das Minimierungsproblem gelöst. Insofern können wir uns auf im Folgenden auf Entscheidungsprobleme konzentrieren.

Analog können wir auch die Größe m einer maximalen Clique eines Graphen $G = (V, E)$ bestimmen. Hier liegt der Wert zwischen den Schranken $u = 0$ und $v = |V|$ und wie zuvor formulieren wir das Entscheidungsproblem, ob es eine Clique der Größe $k = (u + v)/2$ gibt. Die Antwort „ja“ setzt $u := k$, die Antwort „nein“ setzt $v := k - 1$. Auch hier ist man nach $log_2|V|$ vielen Aufrufen des Entscheidungsproblems am Ziel.

Will man für das SAT-Problem nicht nur die Existenz einer Lösung, sondern auch eine konkrete Lösung für das Erfüllbarkeitsproblem $F(x_1, \ldots, x_n)$ in der Hand halten, so kann der Reihe nach die erfüllenden Werte $b_1, \ldots, b_n$ der Variablen $x_1, \ldots, x_n$ finden. Ist nämlich $F(0, x_2, \ldots, x_n)$ erfüllbar, so setzen wir b_1 :=0, ansonsten $b_1 := 1$. Mit dem so gefundenen Wert b_1 fahren wir fort und fragen nach der Erfüllbarkeit von $F(b_1, x_2, \ldots, x_n)$, etc. .

6.1.6 Probleme sind Sprachen

Ja/Nein-Probleme sind Entscheidungsprobleme für geeignete Sprachen. Für eine Klasse von Problemen legt man eine Codierung durch ein Alphabet fest, so dass jedes einzelne Problem durch ein Wort beschrieben wird. Die Menge aller lösbaren Probleme ist dann eine Teilmenge, also eine Sprache. Die Frage, ob ein bestimmtes Problem lösbar ist, ist äquivalent zu der Frage, ob das zugehörige Wort in der Sprache aller lösbaren Probleme liegt.

SAT-Probleme kann man beispielsweise als Wörter im Alphabet

$$\Sigma = \{ x, \vee, \wedge, \neg, (,) \}$$

kodieren. Für die potentiell unbegrenzt vielen Variablen können wir x, xx, xxx, ... verwenden. Die kontextfreie Sprache aller syntaktisch korrekten Formeln zerfällt dann in die Teilsprache derer Formeln, die erfüllbar sind und derer die nicht erfüllbar sind.

Analog kann man das Cliquen-Problem als Wort über dem Alphabet

$$\Sigma = \{x, 0, 1, (,), ”, ”\}$$

kodieren. Die Knoten v_i des Graphen repräsentiert man wie oben durch x,xx, xxx, ... und die Kanten durch Paare „(v_i, v_j)” von Knoten. Stellt man diesem noch die binär codierte Zahl k voran, so hat man das konkrete Cliquenproblem als Wort w in der Sprache aller Worte der Form

$$(k, (v_{i_1}, v_{j_1}), \ldots, (v_{i_n}, v_{j_n}))$$

kodiert.

6.2 Maschinenmodelle und Komplexitätsmaße

Wir betrachten in diesem Kapitel nur algorithmisch lösbare Probleme, d.h. entscheidbare Sprachen $L \subseteq \Sigma^*$ über einem Alphabet Σ. Die charakteristische Funktion von L

ist also auf jeden Fall total und berechenbar. Somit existiert auf jeden Fall eine Turing-Maschine T, die L entscheidet. Die Frage ist somit nur, wie effizient dies möglich sein soll.

Definition 6.2.1. Sei T eine Turingmaschine mit Bandalphabet Σ, die $L \subseteq \Sigma^*$ entscheidet.
- Für jedes Wort $w \in \Sigma^*$ sei $time_T(w)$ die Anzahl der Schritte, die Turingmaschine T mit Input w benötigt, um "$w \in L$" zu entscheiden.
- Für eine beliebige Zahl $n \in \mathbb{N}$ setzen wir:

$time_T(n) := max\{time_T(w) \mid w \in \Sigma^*, |w| \leq n\}$.
- Die Funktion $time_T : \mathbb{N} \to \mathbb{N}$ heißt *Kostenfunktion* der Turingmaschine T.

Beispiel 6.2.2. Wir betrachten eine Turingmaschine, die feststellen soll, ob ein Wort ein Palindrom ist, ob es also von vorne und von hinten gelesen das gleiche Wort ergibt, wie etwa „otto" oder „rentner". Das Problem definiert die Sprache $PALINDROM := \{w \in \Sigma^* \mid w = w^R\}$ wobei w^R das zu w reverse Wort bezeichnet. Zur Lösung können wir eine Turingmaschine bauen, die jeweils den ersten Buchstaben liest, diesen löscht und an das Ende des Wortes läuft. Findet sie dort den gleichen Buchstaben, so löscht sie diesen und macht mit dem verkürzten Wort weiter. Es ist leicht zu sehen, dass für ein Wort der Länge n höchstens

$$n + (n-1) + \ldots + 1 = n * (n+1)/2$$

Schritte auszuführen sind.

Es ist in einem solchen Falle noch nicht ausgeschlossen, dass das Problem, Palindrome zu erkennen, sogar mit einem geringeren Zeitaufwand lösbar wäre. Dazu müssten wir aber Turingmaschinen konstruieren, die das Problem mit entsprechend wenigen Schritten lösen, was im vorliegenden Falle kaum gelingen wird.

Für While-Programme könnten wir entsprechend die Anzahl der Zuweisungen plus die Anzahl der zu überprüfende Bedingungen zählen. Bei Goto-Programmen wären die Anzahl der Zuweisungen und der Sprünge ein gutes Maß für den Berechnungsaufwand. Jeder Algorithmus definiert also eine Kostenfunktion. Diese Funktion werden wir nie exakt ermitteln, sondern nur abschätzen.

Leider ist die Kostenfunktion für eine konkrete Turingmaschine T und eine konkrete Sprache $L \subseteq \Sigma^*$ in vielen interessanten Fällen nur sehr schwierig exakt zu ermitteln. Wir begnügen uns daher mit Schätzfunktionen, die eine obere Schranke für die tatsächliche Kostenfunktion liefern sollen. Eine Schätzfunktion ist damit einfach nur irgendeine Funktion $t : \mathbb{N} \to \mathbb{N}$.

6.2.1 Komplexitätsklassen, O-Notation

Wir wollen qualitative Aussagen über das Wachstum von Schätzfunktionen treffen. Dabei interessiert uns vor allem, wie die Funktionen wachsen, wenn wir ihre Eingabe vergrößern.

Wir wollen zum Beispiel eine *lineare Funktion* wie $l(n) = 42 * n$ von einer *quadratischen Funktion* $q(n) = 3 * n^2$ unterscheiden, nicht aber die lineare Funktion $l_1(n) = 5 * n$ von der ebenfalls linearen Funktion $l(n) = 42 * n$, oder die quadratische Funktion $q(n) = 3 * n^2$ von der ebenfalls quadratischen Funktion $q_1(n) = n^2$.

Für lineare Funktionen wissen wir nämlich, dass beispielsweise 10-fach größere Eingaben zu 10–fach größeren Ergebnissen führen – im Falle von Kostenfunktionen zu 10-fach größeren Kosten.

Für quadratische Funktionen führt jedoch eine 10-fach größere Eingabe zu einer Kostensteigerung um den Faktor 100. Dies sind substantielle Unterschiede zwischen linearen und quadratischen Funktionen, die Vorfaktoren 5, 42, 3, 1 spielen dabei kaum eine Rolle. Daher werden wir Funktionen $f(x)$ und $c * f(x)$ mit $c > 0$ in die gleiche Komplexitätsklasse einordnen wollen.

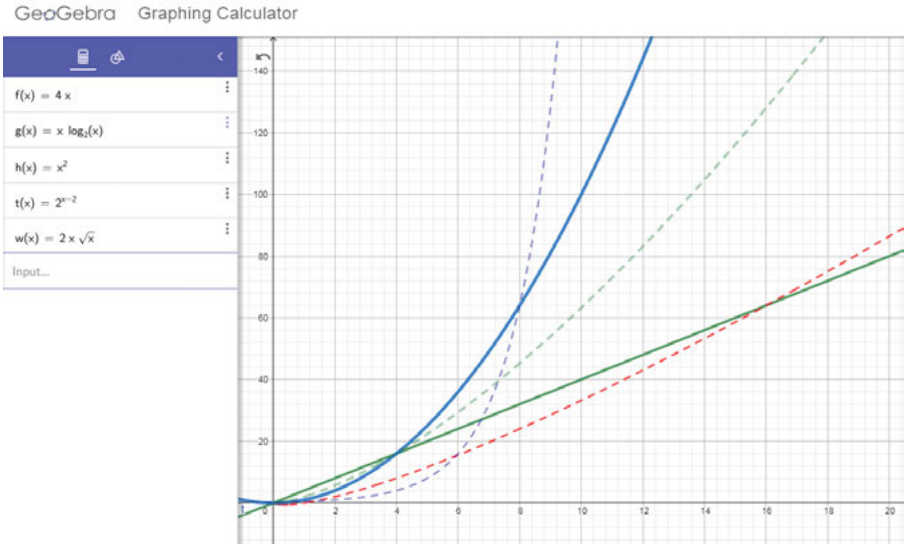


Abb. 6.2.1: Verlauf verschiedener Kostenfunktionen

Wenn wir mit $O(f(x))$ die Komplexitätsklasse von f bezeichnen, dann soll also $O(f(x)) = O(c * f(x))$ sein. Zudem sieht man an Figur 6.2.1 deutlich, dass die verschiedenen Funktionen ihr Wachstum erst ab einem bestimmten Punkt ausspielen.

So wächst die (gestrichelt dargestellte) Funktion $g(x) = x \cdot log_2 x$ erst ab $x = 16$ über die lineare Funktion $f(x) = 4 \cdot x$ hinaus.

Ebenso wächst die (gestrichelt dargestellte) exponentielle Funktion $t(x) = 2^{x-2}$ erst ab ca. $x = 7$ über die Funktion $w(x) = 2x \cdot \sqrt{x}$ hinaus und ab $x = 8$ über die (durchgezogen gezeichnete) quadratische Funktion $h(x) = x^2$. Letzteres könnte man als logische Formel so ausdrücken:

$$\forall x \geq 7 . h(x) \leq t(x).$$

Wen es uns gleichgültig ist, ab welchem konkreten Punkt $t(x) \geq h(x)$ ist, Hauptsache, es gibt so eine Stelle, würden wir einfach sagen:

$$\exists x_0 . \forall x \geq x_0 . h(x) \leq t(x).$$

Wenn wir schließlich noch Funktionen f und g, die sich nur um einen festen Vorfaktor voneinander unterscheiden, in die gleiche Komplexitätsklasse einordnen wollen, so definieren wir:

Definition 6.2.3. Seien $f, g : \mathbb{N} \to \mathbb{R}^+$ Funktionen, wobei $\mathbb{R}^+ = \{r \geq 0 \mid r \in \mathbb{R}\}$. Wir definieren:

$$f \in O(g) : \iff \exists c > 0 . \exists n_0 . \forall n \geq n_0 . f(n) \leq c \cdot g(n)). \tag{6.2.1}$$

Falls $f \in O(g)$ sagen wir, dass f in der Komplexitätsklasse von g ist, bzw, dass f höchstens so komplex ist wie g. Mit dieser Definition rechnet man leicht nach:

Lemma 6.2.4. *Für beliebige Funktionen $f, g, h : \mathbb{N} \to \mathbb{N}$ und Konstante $c \in \mathbb{R}^+$ gilt:*

1. $f \in O(f)$
2. $f \in O(g), g \in O(h) \implies f \in O(h)$
3. $f \in O(g) \implies c \cdot f \in O(g)$
4. $f_1, f_2 \in O(g) \implies f_1 + f_2 \in O(g)$ *sowie* $f_1 \cdot f_2 \in O(g^2)$.
5. $O(f) = O(g) \iff f \in O(g) \wedge g \in O(f)$

Beweis. Trivialerweise gilt, $\forall n \geq 0 . f(n) \leq f(n)$, somit ist die Formel 6.2.1 mit $c = 1$ und $n_0 = 0$ erfüllt.

Zu 2.: Aus den Voraussetzungen haben wir c_1, n_1 und c_2, n_2 mit

$$\forall n \geq n_1 . f(n) \leq c_1 \cdot g(n)$$

sowie

$$\forall n \geq n_2 . g(n) \leq c_2 \cdot h(n).$$

Für $n \geq max(n_1, n_2)$ gilt somit sowohl $f(n) \leq c_1 \cdot g(n)$ als auch $g(n) \leq c_2 \cdot h(n)$. Da $c_2 \geq 0$ vorausgesetzt war, ist

$$f(n) \leq c_1 \cdot g(n) \leq c_1 \cdot c_2 \cdot h(n).$$

Somit ist 6.2.1 erfüllt mit $c_0 = c_1 \cdot c_2$ und $n_0 := max(n_1, n_2)$.

Teile 3.und 4. überlassen wir dem Leser. Für die fünfte Behauptung folgt die Richtung " $\Longrightarrow$ " aus Teil 1. Für „$\Longleftarrow$ " müssen wir zeigen, dass $O(f) \subseteq O(g)$ und $O(g) \subseteq O(f)$. Sei also h irgendeine Funktion mit $h \in O(f)$. Wegen $f \in O(g)$ folgt mit Teil 2., dass $h \in O(g)$. Die andere Richtung verläuft analog. □

Bemerkung 6.2.5. In der Literatur hat sich leider die Notation "$f = O(g)$" eingebürgert, wenn es nach unserer obigen Definition $f \in O(g)$ heißen müsste. Wir werden diese missverständliche Notation nicht benutzen, es gibt, außer der Tradition, auch keinen guten Grund dafür.

In diesem Kapitel werden wir einen Algorithmus als *effizient* einstufen, wenn wir seine Kostenfunktion durch ein Polynom abschätzen können. Die *Komplexitätsklasse Pol* definieren wir als

$$Pol := \cup_{n\in\mathbb{N}} O(x^n).$$

Offensichtlich gilt für jedes Polynom $p(x)$, dass $p \in Pol$.

6.2.2 Die Sprachklasse P

Definition 6.2.6. Sei $L \subseteq \Sigma^*$ eine entscheidbare Sprache und $t : \mathbb{N} \to \mathbb{N}$ eine (Kosten)funktion. Wir sagen: L hat Komplexität $O(t)$, falls es eine Turingmaschine T gibt, die L entscheidet mit $time_T \in O(t)$.

L hat *polynomiale Komplexität*, falls es ein Polynom p gibt, so dass L Komplexität $O(p)$ hat. Die *Sprachklasse P* besteht aus allen Sprachen $L \subseteq \Sigma^*$ mit polynomialer Komplexität.

Beispiel 6.2.7. Für die Sprache PALINDROM gilt offensichtlich $PALINDROM \in P$. Für die Sprachen $CLIQUE$, SAT und TSP sind keine Entscheidungsalgorithmen polynomialer Komplexität bekannt. Die allgemeine Vermutung, dass solche polynomiellen Algorithmen auch nicht existieren können wird durch zahlreiche Argumente untermauert, aber eben zur Zeit noch nicht durch einen mathematischen Beweis. Eines der Argumente ist, dass wir zeigen können:

> Wäre $SAT \in P$, dann würde dies auch für $CLIQUE$, TSP und für eine Unzahl von bisher als schwer geltenden Problemen gelten.

Aber auch umgekehrt könnte man aus einer effizienten Lösung für $CLIQUE$ oder TSP eine ebenso effiziente Lösung von SAT gewinnen. Insofern glaubt die überwiegende Mehrheit der Informatiker, dass SAT, $CLIQUE$, TSP etc. nicht in P liegen, es wird also nie gelingen einen polynomialen Algorithmus zu finden. Diese Probleme scheinen eine Klasse für sich zu bilden, die wir später als NP bezeichnen werden.

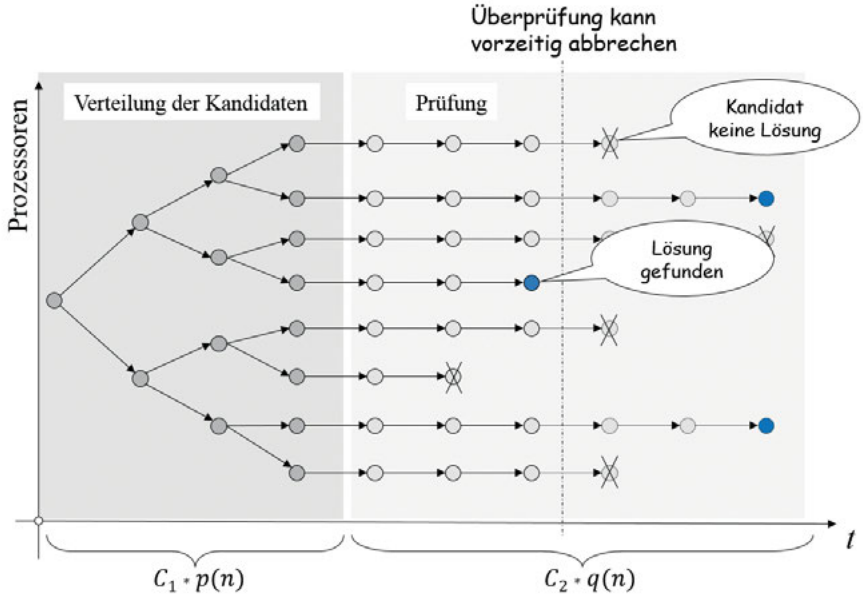


Abb. 6.2.2: Unbegrenzter Parallelismus

6.2.3 Parallelität

Wenn eine Turingmaschine das Problem SAT-Problem nicht effizient lösen kann, bestünde vielleicht die Möglichkeit, das Problem parallel arbeitenden Turing-Maschinen vorzulegen. Die Problemstruktur Generate/Test legt die ausnutzbare Parallelität nahe. Ein Rechner erzeugt alle Lösungskandidaten. Diese werden an parallel arbeitende Rechner verteilt von denen jeder dann seinen Kandidaten testet. Sobald einer der beteiligten Rechner seinen Kandidaten positiv getestet hat, kann er die frohe Botschaft an alle verkünden und der Algorithmus stoppt.

Da jeder einzelne Test effizient ausgeführt werden kann, sollte das Problem insgesamt effizient gelöst werden können.

Hierbei machen wir einige vereinfachende Annahmen. Zunächst gehen wir von unbegrenzter Parallelität aus. Wir haben also unbegrenzt viele Rechner zur Verfügung. Zudem ignorieren wir die Verwaltungskosten, die damit verbunden sind, die Kandidaten zu verteilen. Dennoch gelingt es uns bei unbegrenzter Parallelität die in diesem Kapitel besprochenen Probleme SAT, $CLIQUE$, TSP und viele andere in polynomieller Zeit zu lösen. Abbildung 6.2.2 illustriert den Ablauf und den notwendigen Aufwand.

In der ersten Phase werden die Lösungskandidaten erzeugt. Dies lässt sich bereits schrittweise parallelisieren. Selbst wenn jeder Prozess sich nur in zwei Prozesse aufspaltet bis jeder Kandidat seinen Prozess hat, können nach n Zeitschritten bereits 2^n

Lösungskandidaten erzeugt sein. In jedem Fall kann man (zumindest für die exemplarisch betrachteten Probleme) die Zeit bis zur Erzeugung aller Lösungskandidaten durch ein Polynom $c_1 \cdot p(n)$ begrenzen. In der zweiten Phase prüft jeder Prozess parallel seinen Lösungskandidaten.

Diese prinzipielle Vorgehensweise wird übrigens auch von sogenannten Grid-Computing Projekten eingesetzt. Eines der bekanntesten ist das Projekt *SETI@Home*, wobei *SETI* ein Akronym für „*Search for ExtraTerrestrian Intelligence*“ ist. Hier wird mit Radioteleskopen nach „schmalbandigen“ Funksignalen aus dem Weltraum gesucht, von denen man vermutet, dass sie nicht natürlichen Ursprungs sind, sondern auf technologisch erzeugte Quellen schließen lassen. Diesem und ähnlichen Projekten kann jeder beitreten und seinen Rechner in Zeiten in denen er ansonsten nicht genutzt würde, in den Denst der Wissenschaft zu stellen. Auch Betrüger machen sich heute solche Grid-Technologien zunutze. Allerdings kapern sie ohne Zustimmung der Besitzer deren Rechner, schließen diese zu einem Verbund zusammen, um Bitcoins zu schürfen.

– Im Falle des SAT-Problems mit n Inputvariablen gibt es 2^n Lösungskandidaten. Diese kann man in der Zeit $O(log_2(2^n)) = O(n)$ auf parallele Prozessoren verteilen. Jeder Lösungskandidat kann in linearer Zeit überprüft werden. Dabei kann man sich die kurze Evaluierung boolescher Ausdrücke zunutze machen ($0 \wedge x = 0$, $1 \vee x = 1$, $0 \to x = 1 = x \to 1$).

– Für das CLIQUE-Problem haben wir einen ungerichteten Graph mit n Knoten. Dieser besitzt maximal $n(n-1)/2$ Kanten und $\binom{n}{k} \leq 2^n$ viele k-elementige Teilmengen. Die erste Phase, die Verteilphase, geschieht in Zeit $O(log_2(2^n)) = O(n)$ wie oben, und die Überprüfung, ob ein Kandidat, also eine k-elementige Teilmenge, eine Lösung darstellt, benötigt höchsten die Zeit $O(k^2) \leq O(n^2)$ ist also auch polynomial in der Inputgröße.

Mit dem skizzierten Parallelitätsmodell (bei unbeschränkter Parallelität) lassen sich beide Probleme dann auch in polynomialer Zeit lösen.

6.2.4 Nichtdeterministische Turingmaschinen.

In der Klasse P haben wir die Probleme zusammengefasst, die effizient, also in polynomieller Zeit lösbar sind. Allgemeiner verstehen wir unter der Sprachklasse P alle diejenigen Sprachen L für die es einen Entscheidungsalgorithmus gibt, der die Frage "$w \in L$?" in polynomieller Zeit $c_1 \cdot p(n)$, gemessen an der Länge $|w| = n$, entscheiden kann. In der Klasse NP wollen wir diejenigen Probleme versammeln, die mit dem obigen Modell der unbeschränkten Paralelität noch in polynomieller Zeit lösbar sind. In der Literatur benutzt man allerdings zur formalen Definition den Begriff der Nichtdeterministischen Turingmaschine. Als Verallgemeinerung der „normalen“ Turingma-

schine ist die Auswahl des nächsten Berechnungsschrittes nichtdeterministisch. Befindet sich die Turingmaschine in Zustand q und ist unter dem Lese-Schreibkopf das Zeichen e zu sehen, dann muss $\delta(q, e)$ nicht genau ein Tripel (q', e', d) sein, das den neuen Zustand q', das neue Zeichen e' und die Kopfbewegung $d \in \{L, R\}$ spezifiziert, sondern eine endliche Menge

$$\delta(q, e) = \{(q_1, e_1, d_1), \ldots, (q_n, e_n, d_n)\},$$

aus der die Nachfolgekonfiguration nichtdeterministisch ausgewählt werden darf. Formal beschreibt die Übergangsfunktion δ einer *nichtdeterministischen Turingmaschine* daher eine Funktion

$$\delta : Q \times \Sigma \to \mathbb{P}(Q \times \Sigma \times \{L, R\}),$$

wobei $\mathbb{P}$ wie üblich den Potenzmengenoperator darstellen soll. Q, Σ, L und R haben die übliche Bedeutung als endliche Mengen von Zuständen, Zeichen und den Richtungen *L*inks und *R*echts. Eine solche nichtdeterministische Turingmaschine wählt sich in jeder Konfiguration ein Tripel (q_i, e_i, d_i) und führt den entsprechenden Übergang aus. Sie hält, falls sie in einen Endzustand gelangt ist, oder falls $\delta(q, e) = \{\}$ ist.

Offensichtlich kann man jede deterministische Turingmaschine auch als spezielle nichtdeterministische Turingmaschine auffassen, bei der jede Menge $\delta(q, e)$ einelementig ist und somit keine echte Auswahl zulässt.

Ist $M = (Q, \Sigma, \delta)$ eine nichtdeterministische Turingmaschine, so besteht die Sprache $L(M)$ aus allen Worten w in Σ^* für die eine akzeptierende Berechnung der Maschine M mit Input w existiert. Mit anderen Worten ist $w \in L(M)$, falls die Maschine, gestartet mit dem Input w auf dem Band und dem Kopf vor dem ersten Zeichen von w in jedem Zustand (q, e) eine Nachfolgesituation

$$(q', e', d') \in \delta(q, e)$$

auswählen könnte, so dass sie schließlich in einen akzeptierenden Zustand gelangt oder hält.

Mit $nTime_T(w)$ bezeichnet man die minimale Anzahl von Schritten, die die Nichtdeterministische Turingmaschine T benötigt, um das Wort $w \in L(M)$ zu akzeptieren. Dies ist ein optimistischer Ansatz: man geht davon aus, dass zu jedem Zeitpunkt eine optimale Auswahl aus $\delta(q, e)$ getroffen wird, so dass ein Haltezustand in kürzestmöglicher Zeit erreicht wird.

Da es keine Vorschrift und keinen Anhaltspunkt gibt, welche Nachfolgekonfiguration aus den durch

$$\delta(q, e) \subseteq Q \times \Sigma \times \{L, R\}$$

eingeschränkten Möglichkeiten jeweils gewählt werden sollte, kann man eine solche Maschine nur durch parallele Ausführung vollständig analysieren.

Zur Lösung des SAT–Problems können wir uns eine NTM vorstellen, die nichtdeterministisch mal eine 0 und mal eine 1 auf das Band schreibt. Die erzeugte 0 – 1-Folge wird als Belegung interpretiert und es wird getestet, ob sie die Formel erfüllt. Im

Erfolgsfall könnten wir dann nach polynomial vielen Schritten fertig sein. Die – mit viel Glück – nichtdeterministisch gewählte Belegung entspricht der Belegung die der Glückspilz in dem Rechner-Grid getestet hat, der zuerst eine Lösung vermeldete. Daher ist die optimistische Vorstellung der NTM, die nur richtige Entscheidungen trifft, nicht ganz so abwegig.

6.2.5 Guess and Check.

Die Vorgehensweise wird auch mit „Guess and Check“ umschrieben. Zunächst rät man einen Lösungskandidaten. Anschließend wird überprüft, ob der geratene Kandidat tatsächlich eine Lösung für das Problem ist. Das Raten (engl.: *to guess*) entspricht der nichtdeterministischen Auswahl der Turingmaschine, bzw. den parallel arbeitenden Agenten. Die Check-Phase überprüft den geratenen Lösungsvorschlag, um zu sehen, ob er tatsächlich eine Lösung darstellt. Diese Aufgabe sollte in polynomialer Zeit zu erledigen sein.

6.3 Die Sprachklasse NP

Sei nun $L \subseteq \Sigma^*$ eine entscheidbare Sprache und $t : \mathbb{N} \to \mathbb{N}$ eine Schätzfunktion. Wir sagen $nTime(L) = O(t)$, falls es eine nichtdeterministische Turingmaschine T gibt, die genau L akzeptiert wobei für jedes $w \in \Sigma^*$ gilt

$$nTime_T(|w|) = O(t(|w|).$$

Definition 6.3.1. Die Abkürzung *NP* steht für *nichtdeterministisch polynomial*. Sie beschreibt die Klasse aller Sprachen, die durch eine nichtdeterministische Turingmaschine in polynomialer Zeit erkannt werden können, kurz

$$NP = \{L \mid nTime(L) = O(p), p \text{ ein Polynom}\}.$$

Da wir Worte über verschiedenen Alphabeten leicht (jedenfalls in polynomialer Zeit) ineinander überführen können, z.B. in das binäre Alphabet, wie wir es von Rechnern gewohnt sind, ist die Definition unabhängig von dem gewählten Alphabet, in dem das Problem codiert ist.

Den Zusammenhang zwischen Nichtdeterminismus und und dem als Parallelismus verkleideten *Guess and Check* Paradigma kann man so beschreiben: Der optimistische Nichtdeterminismus steuert die Turingmaschine so, dass nur der richtige Kandidat erzeugt und geprüft wird. Umgekehrt kann die unbeschränkte Parallelität sämtlichen Verzweigungen die die nichtdeterministische Auswahl bietet, gleichzeitig folgen. Um zu zeigen, dass ein Problem in der Klasse *NP* liegt, reicht es daher, eine Guess and Check Lösung anzugeben, bei der in polynomialer Zeit alle Lösungskandidaten parallel erzeugt werden.

6.3.1 Reduzierbarkeit

Eine gute Strategie, ein neues Problem anzugehen, besteht darin, es auf ein anderes, bereits gelöstes Problem zurückzuführen. Diese Vorgehensweise ist auch mit Komplexitätsbetrachtungen verträglich, sofern der Aufwand des Übersetzens nicht ins Gewicht fällt.

Da jedes ja/nein-Problem als Frage „Ist ein Wort in der Sprache “ kodiert werden kann, betrachten wir zwei Sprachen $L \subseteq \Sigma^*$ und $M \subseteq \Gamma^*$ und eine berechenbare Abbildung, die ein Problem w aus L in ein entsprechendes Problem $f(w)$ aus M übersetzen soll. Aus einer Lösung für $f(w)$ wollen wir dann auf die Lösung von w schließen. Die Übersetzungsfunktion f sollte möglichst nicht schwieriger zu berechnen sein, als die Lösung des ursprünglichen Probleme. Daher definieren wir:

Definition 6.3.2. Seien $L \subseteq \Sigma^*$ und $M \subseteq \Gamma^*$ Sprachen und $f : \Sigma^* \to \Gamma^*$ eine berechenbare Funktion.

- L heißt *f-reduzierbar* auf M, falls gilt:
$$w \in L \iff f(w) \in M.$$
- L heißt polynomial reduzierbar auf M falls es ein f gibt, so dass L auf M f-reduzierbar ist und $time(f(w)) \in O(p(|w|)$ für ein Polynom p. In diesem Falle schreiben wir
$$L \leq_p M.$$

Aus der Definition folgen unmittelbar zwei wichtige Eigenschaften: Falls L auf M reduzierbar ist und M entscheidbar, dann ist auch L entscheidbar. Ohnehin interessieren wir uns in diesem Kapitel nur für entscheidbare Sprachen. Wichtiger ist aber, dass aus $L \leq_p M$ und $m \in NP$ auch folgt, dass $L \in NP$.

Das folgende Lemma ist nicht schwer, aber auch nicht ganz so offensichtlich, wie es die Notation nahelegt. Daher werden wir es beweisen.

Lemma 6.3.3. *Aus $L_1 \leq_p L_2$ und $L_2 \leq_p L_3$ folgt $L_1 \leq_p L_3$.*

Beweis. Seien $L_i \subseteq \Sigma_i^*$, für $i = 1, 2, 3$ Sprachen und $f_1 : \Sigma_1^* \to \Sigma_2^*$, $f_2 : \Sigma_2^* \to \Sigma_3^*$ von polynomieller Komplexität, so dass L_1 mit f_1 auf L_2 reduzierbar ist und L_2 mit f_2 auf L_3. Somit ist L_1 mit $f_2 \circ f_1$ auf L_3 reduzierbar.

Zudem sind f_1 und f_2 in polynomialer Zeit berechenbar, es gibt für $k = 1, 2$ also Turingmaschinen T_k, die $f_k(w)$ in $c_k \cdot p_k(|w|)$ Schritten berechnen.

Für die Berechnung von $f_1(w)$ benötigt T_1 maximal

$$c_1 \cdot p_1(|w|)$$

Schritte. Da in jedem Schritt höchstens ein Zeichen geschrieben werden kann, folgt aber auch:

$$|f_1(w)| \leq c_1 \cdot p_1(|w|).$$

Für die Berechnung von $f_2(f_1(w))$ benötigt T_2 jetzt allerhöchstens

$$c_2 \cdot p_2(|f_1(w)|) \leq c_2 \cdot p_2(c_1 \cdot p_1(|w|))$$

Schritte. Für das Polynom $q(n) = c_2 \cdot p_2(c_1 \cdot p_1(n))$ gilt also

$$time((f_2 \circ f_1)(w)) = time(f_2(f_1(w)) \leq q(|w|).$$

□

6.3.2 SAT und 3-SAT

Eine Variante des SAT-Problems geht davon aus, dass die Formel bereits in konjunktiver Normalform (KNF) vorliegt. Dies ist eine Konjunktion von Disjunktionen von Literalen, wobei ein Literal entweder eine Variable oder eine negierte Variable ist. Jede Formel kann in eine äquivalente Formel in konjunktiver Normalform umgewandelt werden. Allerdings ist diese Umwandlung im Allgemeinen sehr aufwendig, im schlimmsten Fall sogar von exponentieller Komplexität. Im Rest des Kapitels benutzen wir folgende Begriffe:

- ein *Literal* ist eine Variable oder die Negation einer Variablen.
- eine *Klausel* ist eine Disjunktion von Variablen
- eine *Konjunktive Normalform* (KNF) ist eine Konjunktion von Klauseln.

Die Formel

$$(\neg x_1 \vee x_2 \vee \neg x_3 \vee x_4) \wedge (x_1 \vee \neg x_2 \vee x_3 \vee \neg x_4) \wedge (x_1 \vee \neg x_3) \wedge (\neg x_1 \vee x_3)$$

liegt bereits als konjunktive Normalform vor. Sie besteht aus den 4 Klauseln

$$\{\{\neg x_1, x_2, \neg x_3, x_4\}, \{x_1, \neg x_2, x_3, \neg x_4\}, \{x_1, \neg x_3\}, \{\neg x_1 \vee x_3\}\}$$

mit insgesamt 8 Literalen: $\{\neg x_{1,} x_2, \neg x_3, x_{4,}, x_1, \neg x_2, x_3, \neg x_4\}$. Jede Klausel wie z.B. $(\neg x_1 \vee x_2 \vee \neg x_3 \vee x_4)$ können wir auch wieder als Menge von Literalen $\{x_1, x_2, \neg x_3, x_4\}$ darstellen, im vorliegenden Falle erhalten wir damit insgesamt die Darstellung

$$\{\{\neg x_1, x_2, \neg x_3, x_4\}, \{x_1, \neg x_2, x_3, \neg x_4\}, \{x_1, \neg x_3\}, \{\neg x_1 \vee x_3\}\}$$

als Menge von Mengen von Literalen, welche die ursprüngliche Formel repräsentiert. Eine erfüllende Belegung ist dann eine Abbildung der Variablen in die Wahrheitswerte $\{0, 1\}$ so dass *jede* Klausel wahr wird. Eine Klausel wird genau dann wahr, wenn *mindestens ein* Literal wahr ist. Ein Beispiel einer erfüllenden Belegung wäre im obigen Falle $x_1 = 0$, $x_2 = 0$, $x_3 = 0$ und x_4 beliebig.

In dieser Darstellung lautet das SAT-Problem:

6.3.3 SAT-Problem:

Gegeben: Eine Menge von Klauseln (Mengen von Mengen von Literalen) mit Variablen $x_1, \ldots, x_n$.

Gefragt: Gibt es eine Belegung der Variablen $x_1 = b_1, \ldots, x_n = b_n$ mit Wahrheitswerten $b_i \in \{0, 1\}$, so daß jede Klausel wahr wird.

Selbst wenn wir voraussetzen, dass unser SAT-Problem bereits in einer solchen Repräsentation als Menge von Mengen von Literalen vorliegt, wird es nicht einfacher. Was aber, wenn wir annehmen, dass jede Klausel „kurz“ ist, z.B. höchstens 3 Literale enthält?

6.3.4 Die Tseitin-Transformation und $3 - SAT$

1966 präsentierte G.S.Tseitin eine Repräsentation von logischen Formeln als Mengen von 3-Klauseln, also als Mengen von Klauseln mit jeweils höchstens 3 Variablen. Die Transformation überführt eine beliebige aussagenlogische Formel F in eine *erfüllungsäquivalente* Menge κ von 3-Klauseln. Dies bedeutet, dass die ursprüngliche Formel genau dann erfüllbar ist, wenn die gewonnene Menge von 3-Klauseln auch erfüllbar ist. Zudem hat die Transformation den Vorteil, nur einen Aufwand zu benötigen, der linear in der Größe der ursprünglichen Formel zu sein. Da muss es doch einen Haken geben?

Die gewonnene Menge von 3-Klauseln ist nicht mehr äquivalent zur Ausgangsformel, weil sie i.A. zusätzliche Variablen besitzt. Für unsere Zwecke ist das aber nicht relevant, denn sie ist *erfüllungsäquivalent*, was bedeutet, dass die ursprüngliche Formel genau dann erfüllbar ist, wenn die transformierte Menge von 3-Klauseln erfüllbar ist.

Die Idee ist ganz einfach und immer wieder in der Informatik anzutreffen. Ein kompliziertes Gebilde, wie hier den Baum, der die ursprüngliche Formel repräsentiert, zerschneidet man in einfachere Teile, wobei man jede Schnittstelle durch eine neue Variable markiert. Die einfacheren Teile kann man jetzt algebraisch behandeln. Anhand der Marken kann man aus den Teilen das ursprüngliche Gebilde rekonstruieren.

Im Falle einer aussagenlogischen Formel bringt man zunächst mit den deMorganschen Gesetzen die Negationen nach innen, bis sie nur höchstens noch vor den Variablen stehen. Jetzt hat die Formel nur $\vee$ und $\wedge$ als innere Knoten. An einem dieser Knoten, beispielsweise einem $\vee$-Knoten mit den Söhnen s_1 und s_2 wird sie jetzt zerschnitten und die Schnittstelle mit einer neuen Variablen z markiert. Die neue Variable z ersetzt einerseits den Operatorknoten, und andererseits wird die Äquivalenz

$$z \iff s_1 \vee s_2$$

für das abgeschnittene Bruchstück aufgenommen. Zum Schluss hat man nur Bruchstücke repräsentiert durch Äquivalenzen der Art $z \iff s_1 \vee s_2$ oder $z = s_1 \wedge s_2$. Offensichtlich ist die ursprüngliche Formel genau dann erfüllbar wenn die neuen Formeln mit den zusätzlichen Variablen erfüllbar sind.

Die folgende Figur zeigt die Tseitin Transformation anhand der Formel

$$x_1 \wedge \neg((x_2 \vee \neg x_3) \wedge \neg x_4).$$

Nachdem die Negationen nach innen gewandert sind, bleibt die Formel

$$x_1 \wedge (\neg x_2 \wedge x_3 \vee x_4)$$

übrig. Die beiden inneren Knoten Führen zu zwei Schnittstelen mit den neuen Variablen z_1 und z_2 und den drei Bruchstücken

$$\{x_1 \wedge z_1, z_1 \Leftrightarrow z_2 \vee x_4, z_2 \Leftrightarrow \neg x_2 \wedge x_3\}.$$

Jedes der Bruchstücke lässt sich äquivalent durch maximal drei Klauseln darstellen, diejeweils höchstens drei Literale beinhalten.

Aus $z \Leftrightarrow a \vee b$ werden die Klauseln $\{\neg z, a, b\}, \{\neg a, z, \}$ und $\{\neg b, z\}$ und aus einem Bruchstück der Art $z \Leftrightarrow a \wedge b$ die Klauseln $\neg z \vee a$, $\neg z \vee b$ und $\neg a \vee \neg b \vee z$. Insgesamt ist also die Formel

$$x_1 \wedge \neg((x_2 \vee \neg x_3) \wedge \neg x_4)$$

genau dann erfüllbar, wenn die Klauselmenge

$$\{x_1, z_1, \neg z_1 \vee z_2 \vee x_4, \neg z_2 \vee z_1, \neg x_4 \vee z_1, \neg z_2 \vee \neg x_2, \neg z_2 \vee x_3, x_2 \vee \neg x_3 \vee z_2\}$$

erfüllbar ist. Als Menge von Mengen von Literalen geschrieben ist also die Ursprungsformel erfüllungsäquivalent zu

$$\{\{x_1\}, \{z_1\}, \{\neg z_1, z_2, x_4\}, \{\neg z_2, z_1\}, \{\neg x_4 \vee z_1\}, \{\neg z_2, \neg x_2\}, \{\neg z_2, x_3\}, \{x_2, \neg x_3, z_2\}\}.$$

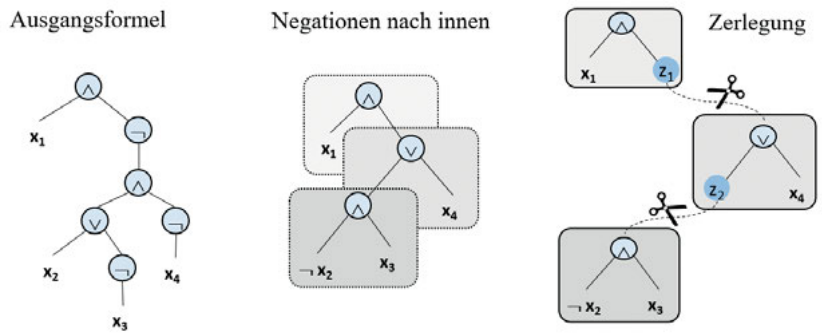


Abb. 6.3.1: Tseitin-Zerlegung

6.3.5 3-SAT-Problem:

Gegeben: Eine M Menge von 3-Klauseln

Gefragt: Ist M erfüllbar.

Das 3-SAT Problem ist also ein Spezialfall des SAT-Problems, so daß trivialerweise $3-SAT \leq_p SAT$ gilt. Die Tseitin Transformation belegt jetzt auch die Umkehrung, nämlich $SAT \leq_p 3-SAT$. Die Umwandlung eines konkreten SAT Problems in ein äquivalentes 3-SAT-Problem ist, wie wir gesehen haben, in einem Aufwand durchführbar, der linear in der Anzahl der Operatoren der ursprünglichen Formel ist. Die entstandene Formelmenge ist zwar nicht äquivalent, aber doch erfüllbarkeitsäquivalent, was für unsere Fragestellung mehr als genügt.

Liegt ein SAT-Problem schon als Menge von Klauseln vor, so kann man die Tseitin-Transformation viel einfacher gestalten. Jede Klausel $\{l_1, \ldots, l_k\}$ mit $k > 3$ Literalen können wir unter Zuhilfenahme einer neuen Variablen z in zwei kürzere Klauseln $\{l_1, l_2, z\}$ und $\{\neg z, l_3, \ldots, l_k\}$ aufspalten, von denen eine die Länge 3 hat und die zweite die Länge $k-1$. Aus

$$\{l_1, l_2, \ldots., l_{k-1}, l_k\}$$

wird iterativ schließlich die Menge

$$\{l_1, l_2, z_1\}, \{\neg z_1, l_3, z_2\}, \ldots, \{\neg z_{k-3} l_{k-1}, l_k\}.$$

6.4 NP-Vollständigkeit

Im Jahre 1971 erschien ein Artikel von Stephen A. Cook, der folgendes ankündigte:

„It is shown that any recognition problem solved by a polynomial time-bounded nondeterministic Turing machine can be "reduced" to the problem of determining whether a given propositional formula is a tautology."

In unserer Sprache ausgedrückt behauptete der Autor also, dass jedes Problem in der Klasse NP sich mit polynomiellem Aufwand sich auf ein SAT-Problem zurückführen lässt. Da SAT selber in der Klasse NP liegt, ist es, bis auf einen polynomiellen Übersetzungsaufwand das schwerste Problem in NP. Eine effiziente Lösung von SAT würde also eine effiziente Lösung aller Probleme in NP nach sich ziehen. In der Folge wurden eine Reihe anderer klassischer Probleme wie CLIQUE oder TSP als gleich schwer charakterisiert. Die Veröffentlichung des Buches von M.Garey und D.S.Johnson: "Computers and Intractability" im Jahre 1979 machte die Thematik der „NP-Vollständigkeit" einer breiten Öffentlichkeit bekannt, so dass in der Folge unzählige Veröffentlichungen und Doktorarbeiten neue Probleminstanzen als NP-vollständig, also gleich schwer wie SAT identifizierten.

Nach der Öffnung des eisernen Vorhangs im Jahre 1989 und dem damit einhergehenden freien Austausch russischer Mathematiker mit den Kollegen im Westen stellte

sich heraus, dass unabhängig von Stephen Cook auch Leonid Levin entsprechende Ergebnisse im Rahmen seiner Dissertation an der Lomonossow Universität in Moskau erzielt hatte, so dass man heute oft von dem Satz von Cook-Levin spricht.

6.4.1 Der Satz von Cook-Levin

Die Klasse *NP* der nichtdeterministisch polynomialen Probleme haben wir bereits kennengelernt und gesehen, dass sie unter anderem SAT, 3-SAT, CLIQUE und TSP enthalten. Wir definieren nun:

Definition 6.4.1. Ein Problem P ist *NP-hart*, wenn $Q \leq_p P$ für jedes Problem $Q \in NP$ gilt. Ist P selber in *NP* und auch *NP*-hart, so sagt man, P sei *NP-vollständig*.

Ein Problem P aus *NP* ist also NP-vollständig, wenn jedes andere Problem in *NP* in polynomialer Zeit auf P zurückgeführt werden kann. Damit gehört p zu den schwierigsten Problemen in *NP*. Mit dieser Begriffsbildung lautet der Satz von Cook-Levin:

Satz 6.4.2. *SAT ist NP-vollständig.*

Den Beweisgang wollen wir später erst erläutern. Die Bedeutung des Satzes ist aber immens. Jedes Problem, das von einer nichtdeterministischen Turingmaschine in polynomieller Zeit gelöst werden kann, können wir polynomial auf SAT zurückführen. Hätten wir einen effizienten Lösungsalgorithmus für SAT, so könnten wir – bis auf einen polynomiellen Übersetzungsvorgang – jedes Problem in NP effizient lösen. Um also den Satz von Cook-Levin zu zeigen, ist es notwendig darzustellen, wie man eine beliebige nichtdeterministische Turingmaschine und ihre erkennenden Läufe so in eine aussagenlogische Formel codieren kann, dass eine erfüllende Belegung dieser Formel einer Lösung des Problems durch die Turingmaschine entspricht.

Korollar 6.4.3. *3-SAT ist NP-vollständig.*

Beweis. Da SAT in NP ist, gilt dies erst recht für $3-SAT$. Mit der Tseitin-Transformation können wir wir in polynomialer Zeit jedes SAT-Problem in ein äquivalentes 3-SAT Problem umwandeln, so dass $SAT \leq_p 3-SAT$ ist. Sei Q nun ein beliebiges Problem in *NP*. Der Satz von Cook-Levin beinhaltet, dass $Q \leq_p SAT$ ist. Mit Lemma 6.3.3 erhalten wir $Q \leq_p 3-SAT$. □

Nachdem wir mit SAT und $3-SAT$ die ersten NP-vollständigen Probleme identifiziert haben, gestaltet es sich meist relativ einfach, weitere derartige Probleme zu finden. Wir gehen stets folgendermaßen vor:

Mit einer Guess-and-Check Methode zeigen wir zunächst, dass unser Problem Q in *NP* liegt. Anschließend führen wir SAT oder $3-SAT$ (oder irgendein anderes bereits als

NP-hart erkanntes Problem) mit einem polynomialen Algorithmus auf unser Problem Q zurück. Wir demonstrieren dies anhand des CLIQUE-Problems.

6.4.2 CLIQUE ist NP-vollständig

1. k-CLIQUE ist in NP:

Guess: Rate einen Kandidaten für eine k–Clique. Es sind $\binom{n}{k} \leq 2^n$ Kandidaten zu überprüfen, die in linearer Zeit auf parallele Prozesse verteilt werden können.

Check: Uberprüfe, ob ein Kandidat eine Lösung ist. Dies gelingt in polynomialer Zeit, da nur $k(k-1)/2$ viele Kanten zu prüfen sind.

2. k-CLIQUE ist NP-hart:

Zeige: 3-.SAT$\leq_p$ $k - CLIQUE$

Für den letzten Schritt müssen wir zeigen, wie wir ein beliebiges 3-SAT Problem in polynomialer Zeit in ein Cliquen-Problem überführen können.

Dazu sei eine Menge $\{\kappa_1, \ldots, \kappa_n\}$ von 3-Klauseln gegeben, von der wir wissen wollen, ob sie erfüllbar ist. Eine erfüllende Belegung muss in jeder Klausel mindestens ein Literal l auswählen das den Wert 1 erhält. Das geht natürlich nur, wenn unter den ausgewählten Literalen keine zwei widersprüchlich sind - die Belegung darf nicht etwa x aus der einen Klausel und $\neg x$ aus einer anderen Klausel auswählen.

Wir basteln nun einen Graphen, dessen Grundmenge

$$G = \kappa_1 \uplus \kappa_2 \uplus \ldots \uplus \kappa_n,$$

die disjunkte Vereinigung der Klauseln ist. Die Knoten des Graphen sind also Paare (i, l), wobei l ein Literal ist und i die Klausel, in der dieses vorkommt. Nun verbinden wir zwei Knoten (i, l) und (j, l') falls

$-i \neq j$ und
$-l \neq \neg l'$.

Zwei Literale werden also genau dann verbunden, wenn sie nicht komplementär zueinander sind und sich in verschiedenen Klauseln befinden.

Ist C eine n-Clique in diesem Graphen, so muss jeder Knoten $l \in C$ aus einer anderen Klausel stammen, und da wir genau n Klauseln haben, muss C sogar aus jeder Klausel exakt ein Literal enthalten. Weil je zwei Knoten in C verbunden sind, können diese nie komplementär sein. Somit wählt C aus jedem Knoten ein Literal aus, dem wir nun den Wahrheitswert 1 zuordnen können, um eine erfüllende Belegung zu erhalten.

Umgekehrt liefert jede erfüllende Belegung auch mindestens ein Literal aus jeder Klausel, welches den Wahrheitswert 1 erhält. Selbstverständlich muss die Belegung

konsistent sein, wenn sie in einer Klausel dem Literal l einen Wahrheitswert w zuweist, muss l auch in jeder anderen Klausel den Wahrheitswert w erhalten und $\neg l$ den Wahrheitswert $\neg w$. Damit ist eine erfüllende Belegung aber eine n–Clique.

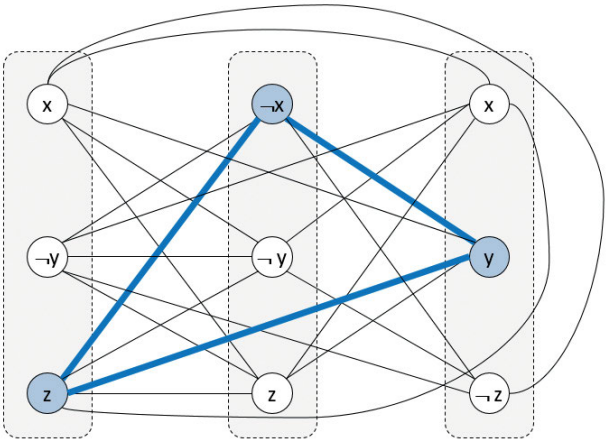


Abb. 6.4.1: Graph zu einer Klauselmenge, samt 3-Clique

Figur 6.4.1 zeigt den zu der Formel

$$F(x, y, z, u) = (x \vee \neg y \vee z) \wedge (\neg x \vee \neg y \vee z) \wedge (x \vee y \vee \neg z)$$

mit Klauselmenge

$$\{\{x, \neg y, z\}, \{\neg x, \neg y, z\}, \{x, y, \neg z\}\}$$

gehörenden Graphen. Hervorgehoben ist eine 3-Clique, die der Belegung

$$x = 0, y = 1, z = 1$$

entspricht.

6.4.3 Der Beweis des Satzes von Cook-Levin

Die Beweisidee des Satzes von Cook-Levin ist jetzt nicht mehr schwer nachzuvollziehen:

Jedes Problem P in NP ist durch eine Sprache L gegeben, und ein Wort w von dem eine nichtdeterministische Turingmaschine T in polynomieller Zeit entscheiden soll, ob $w \in L$ gilt oder nicht. Es gibt also ein Polynom p, so dass T für jedes w in $p(|w|)$ vielen Schritten feststellen kann, ob $w \in L$ gilt. Wir realisieren dies so, dass bestimmte Endzustände signalisieren, dass $w \in L$ ist, andere, dass $w \notin L$.

Die Idee ist jetzt, die Turingmaschine T zusammen mit dem Wort w in eine aussagenlogische Formel $F_{T,w}(x_1, \ldots, x_n)$ zu codieren, so dass gilt:

$$F_{T,w}(x_1, \ldots, x_n) \text{ erfüllbar} \iff w \in L.$$

Unsere Formel wird sehr viele Variablen besitzen, dennoch werden wir sie in polynomieller Zeit konstruieren können.

Sei $\Gamma = \Sigma \cup \{\#\}$ das Bandalphabet und Q die Menge der Zustände der Maschine mit Startzustand $s \in Q$.

Für ein gegebenes Wort w mit $|w| = n$ begrenzt $p(n)$ nicht nur die Anzahl der Schritte der Maschine, sondern auch den Bereich des Bandes, den der Lese-Schreibkopf im Laufe der Berechnung besuchen kann - vom Ausgangspunkt kann der Kopf höchstens $p(n)$ Schritte nach links oder nach rechts gehen.

Jetzt führen wir $O(p(n)^2)$ viele aussagenlogische Variablen ein, mit denen wir kodieren,

–was zum Zeitpunkt k in der i-ten der Bandzelle steht, für jedes $k \leq p(n)$ und $i \in Pos(n) = \{-p(n), \ldots, p(n)\}$

–wo sich der Kopf aufhält

–in welchem Zustand sich die Maschine befindet.

Da wir aber nur Boolesche Variablen zur Verfügung haben, benötigen wir für alle möglichen Kombinationen von Werten eine Boolesche Variable, die wahr ist, falls die Bedingung erfüllt ist, falsch sonst. Im Einzelnen sind dies:

$Q_{q,t}$	Ist die Maschine zum Zeitpunkt t in Zustand Q ?
$K_{i,t}$	Ist der Kopf zum Zeitpunkt t auf Bandposition i ?
$B_{i,e,t}$	Enthält Bandposition i zum Zeitpunkt t das Zeichen e?

Die Anzahl der Variablen ist zwar riesig: $|Q| \cdot p(n) + 2 \cdot p(n) \cdot p(n) + 2 \cdot p(n) \cdot |\Sigma| \cdot p(n)$, aber dennoch überschaubar, weil von der Größenordnung $O(p(n)^2)$.

Als nächstes benötigen wir sogenannte *Rahmenaxiome*, die die möglichen Ausprägungen der Variablen gemäß ihrer intendierten Bedeutung einschränken und die nichts mit der aktuellen Berechnung zu tun haben. Beispielsweise kann zu jedem Zeitpunkt die Maschine höchstens einen Zustand haben und der Kopf sich höchstens auf einer Position aufhalten und auf jeder Bandposition nur ein Zeichen stehen:

$\neg Q_{q,t} \vee \neg Q_{q',t}$ für alle $t \leq p(n)$ und alle $q \neq q' \in Q$
$\neg K_{i,t} \vee \neg K_{i',t}$ für alle $t \leq p(n)$ und alle $i \neq i' \in Pos(n)$
$\neg B_{i,e,t} \vee \neg B_{i,e',t}$ für alle $i \in Pos(n)$, $e \neq e' \in \Sigma$, $t \leq p(n)$.

Mindestens ein Zeichen muss aber an jeder Bandposition stehen, und dieses kann sich nur ändern, wenn der Kopf eben dort steht:

$\bigvee_{e \in \Sigma} B_{i,e,t}$ für alle $i \in Pos$ and alle $0 \leq t \leq p(n)$

$B_{i,e,t} \rightarrow B_{i,e,t+1} \vee K_{i,t}$ für alle $i \in Pos(n)$, $e \in \Sigma$, $t \leq p(n)$

Dass zu jedem Zeitpunkt t mindestens eine der $Q_{q,t}$ und eine der $K_{i,t}$ wahr sein muss, wird sich aus den folgenden Klauseln ergeben. Zunächst codieren wir die Ausgangskonfiguration durch die Klauseln, die beschreiben, dass die Maschine sich zum Zeitpunkt 0 im Anfangszustand befindet, der Lese-Schreibkopf auf Position 0 ist und auf den Bandpositionen das Wort $w = w_1 \ldots w_n$ und außerhalb das Leerzeichen '#' geschrieben steht:

$Q_{s,0}$	Am Anfang befindet sich die Maschine im Startzustand,
$K_{0,0}$	anfangs steht der Kopf auf Position 0 und
$B_{i,\#,0}$	für jedes $i \leq 0$ und jedes $i > n$
$B_{i,w_i,0}$	für $1 \leq i \leq n$.

Endlich kommen wir zu den Klauseln, die die möglichen Aktionen der Maschine kodieren. Für jedes $t \leq p(n)$, $i \in Pos$, $e \in \Sigma$ und $q \in Q$ benötigen wir

$$K_{i,t} \wedge Q_{q,t} \wedge B_{i,e,t} \rightarrow \bigvee_{(q',e',d)\in\delta(q,e)} (B_{i,e',t+1} \wedge K_{i+d,t+1} \wedge Q_{q',t+1})$$

Falls zum Zeitpunkt t der Kopf auf einer Position steht, die das Zeichen e enthält, dann wird zum folgenden Zeitpunkt für ein $(q', e', d) \in \delta(q, e)$ das Zeichen an Position i in e' geändert sein, der neue Zustand ist q' und der Kopf ist in Richtung d gewandert, wobei wir die Kopfbewegung mit $d \in \{-1, 1\}$ kodiert haben.

Schließlich muss die Maschine nach höchstens $p(n)$ Schritten in einem der Endzustände $F \subseteq Q$ angelangt sein:

$$\bigvee_{0\leq t\leq p(n)} \bigvee_{q\in F} Q_{f,t}.$$

Insgesamt haben wir jetzt $O(p(n)^3)$ viele Klauseln mit $O(p(n)^2)$ vielen verschiedenen Variablen erzeugt

Gibt es einen Lauf der Turingmaschine, der nach $p(n)$ vielen Schritten in einen Endzustand mündet, so liefert uns dieser eine Belegung der Variablen, die alle Klauseln wahr macht und umgekehrt liefert jede Belegung der Variablen, die alle Klauseln wahr macht, einen Lauf der Turingmaschine, der in einen Endzustand mündet.

P = NP ?

Die Probleme in der Klasse P zählt man zu den effizient lösbaren, während diejenigen in NP als schwierige oder harte Probleme gelten. Die schwierigsten Probleme in NP sind die NP-vollständigen, zu denen SAT, 3-SAT, CLIQUE und auch TSP gehören.

Es ist klar, dass $P \subseteq NP$ gilt. Erstaunlicherweise ist bis heute der Nachweis, dass $P \neq NP$ gilt, nicht gelungen. Es könnte daher immer noch möglich sein, dass die beiden Klassen zusammenfallen, dass also $P = NP$ gilt. Dazu müsste nur für ein einzi-

ges NP-vollständiges Problem ein deterministisch polynomieller Algorithmus gefunden werden, alle anderen Probleme aus NP wären dann automatisch auch in P.

Damit hätte man für alle der als schwer bekannten Probleme plötzlich effiziente Lösungen. Dies hält heute kaum jemand für möglich. Die Mehrzahl der Informatiker vermutet, dass $P \neq NP$ gilt, allerdings fehlt bis der mathematische Nachweis.

Die Frage, ob $P \neq NP$ ist, gilt als derzeit schwierigstes Problem der theoretischen Informatik und nicht umsonst zählt es zu den vom *Clay Mathematical Institute* bestimmten Millenium Problemen, für dessen Lösung zudem ein Preis von 1 Million US$ ausgelobt wurde.

6.5 Praktische Anwendung von SAT-Problemen

Das SAT-Problem klingt zunächst wie ein mathematischer Zeitvertreib. Es ist aber in der praktischen Informatik und in der Industrie von größter Bedeutung. Insbesondere die Verifikation endlicher Systeme, Algorithmen und Protokolle führt immer zu SAT-Problemen, die dann möglichst effizient gelöst werden müssen.

Allgemein kann man endliche Systeme (man spricht auch von Transitionssystemen) durch eine endliche Menge Q von Zuständen und eine Übergangsrelation $R \subseteq Q \times Q$ angeben. Statt $(s, s') \in R$ schreibt man

$$s \to_R s'$$

und sagt: „*von Zustand s kann das System in Zustand s' übergehen*".

Eine Folge $q, q_1, \ldots, q_n$ von Zuständen mit

$$q \to_R q_1 \to_R q_2 \ldots q_{n-1} \to_R p$$

nennt man einen *Pfad* von q nach p. Jeder Pfad beschreibt eine mögliche zeitliche Entwicklung des Systems. Meist hat man noch einen Anfangszustand $q_0 \in Q$ und eine oder mehrere Mengen $T \subseteq Q$ von Endzuständen.

In Anwendungen handelt es sich dabei oft um verbotene Zustände. Hat man zum Beispiel ein Bauteil, das eine Eisenbahn-Signalanlage steuert, so will man etwa garantieren, dass nie zwei sich kreuzende Fahrwege gleichzeitig freigegeben werden. Man will also verifizieren, dass gewisse verbotene Zustände nie erreicht werden können.

Endliche Transitionssysteme können mit Bitvektoren codiert werden. Mit Bitvektoren $(b_1, \ldots, b_n) \in \{0, 1\}^n$ der Länge n lassen sich bis zu 2^n Zustände repräsentieren. Einen Bitvektor $b = (b_1, \ldots, b_n)$ kann man auch als $\wedge -$ *Klausel* $K_b(x_1, \ldots, x_n)$ repräsentieren, so dass $K_b(x_1, \ldots, x_n) = 1$ genau dann gilt, wenn man $x_1 = b_1, \ldots, x_n = b_n$ einsetzt.

So repräsentiert beispielsweise die Klausel

$$K_b(x_1, x_2, x_3) = x_1 \wedge \neg x_2 \wedge x_3$$

den Bitvektor $(1, 0, 1)$.

Jeder Teilmenge von Zuständen $S \subseteq Q$ entspricht eine Menge von Bitvektoren. Jede Menge S von Bitvektoren kann man durch eine aussagenlogische Formel $F_S(x_1, \ldots, x_n)$ mit n Variablen $x_1, \ldots, x_n$ repräsentieren. Diese wählt man man so, dass

$$(b_1, \ldots, b_n) \in S \iff F_S(b_1, \ldots, b_n) = 1.$$

Praktisch kann man F_S als Disjunktion der Klauseln gewinnen, die den Bitvektoren aus S entsprechen. Ist beispielsweise

$$S = \{(1, 0, 1), (1, 1, 0), (0, 1, 1), (1, 1, 1)\},$$

so können wir F_S gewinnen, in dem wir die Disjunktion der Klauseln hinschreiben, die zu den Ellementen von S gehören. Hier also:

$$F_S(x_1, x_2, x_3) = (x_1 \wedge \neg x_2 \wedge x_3) \vee (x_1 \wedge x_2 \wedge \neg x_3) \vee (\neg x_1 \wedge x_2 \wedge x_3) \vee (x_1 \wedge x_2 \wedge x_3).$$

Oft kann man solche Formeln aber vereinfachen. Hier gilt z.B.

$$F_S(x_1, x_2, x_3) = (x_1 \wedge x_2) \vee (x_1 \wedge x_3) \vee (x_2 \wedge x_3).$$

Analog kann man jede Relation mit Bitvektoren der doppelten Länge $2n$ codieren. Dabei setzt man

$$R = \{(b_1, \ldots, b_n, b_1', \ldots, b_n') \mid (b_1, \ldots, b_n) \to_R (b_1', \ldots, b_n')\}.$$

Relationen kann man daher durch Formeln mit $2n$ Variablen $F_R(x_1, \ldots, x_n, y_1, \ldots, y_n)$ codieren. Hier gilt entsprechend:

$$(b_1, \ldots, b_n) \to_R (b_1', \ldots, b_n') \iff F_R(b_1, \ldots, b_n, b_1', \ldots, b_n') = 1$$

6.5.1 Transitionssystem und Darstellung als Schaltung

Als Beispiel wollen wir ein System mit 4 Zuständen 0, 1, 2, 3 entwerfen. Diese Zustände sollen zyklisch duchlaufen werden, so dass man nach 4 Schritten wieder am Ausgangszustand angelangt ist. Die Zustände Q können wir willkürlich durch die Bitvektoren $\{(0, 0), (0, 1), (1, 0), (1, 1)\}$ repräsentieren. Die Übergangsrelation

$$R = \{0 \to 1, 1 \to 2, 2 \to 3, 3 \to 0\}$$

wird dann repräsentiert durch

$$\{(0, 0, 0, 1), (0, 1, 1, 0), (1, 0, 1, 1), (1, 1, 0, 0)\}.$$

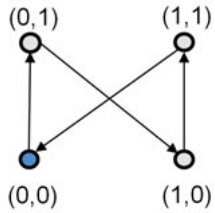


Abb. 6.5.1: Ringzähler

Als aussagenlogische Formel geschrieben erhalten wir

$$\begin{aligned} F_R \quad = \quad & (\neg x_1 \wedge \neg x_2 \wedge \neg y_1 \wedge y_2) \\ & \vee (\neg x_1 \wedge x_2 \wedge y_1 \wedge \neg y_2) \\ & \vee (x_1 \wedge \neg x_2 \wedge y_1 \wedge y_2) \\ & \vee (x_1 \wedge x_2 \wedge \neg y_1 \wedge \neg y_2) \end{aligned}$$

Diese Formel kann man vereinfachen zu:

$$F_R = (y_2 = \neg x_2) \wedge (y_1 = ((x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_2))$$

oder weiter zu

$$F_R = (y_2 = \neg x_2) \wedge (y_1 = x_1 \oplus x_2).$$

Diese Darstellung legt eine Implementierung als Booleschen Schaltkreis mit einem Negationsglied und einem XOR nahe:

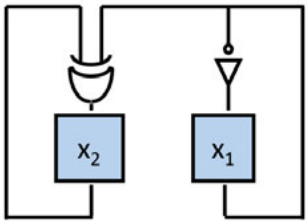


Abb. 6.5.2: Zähler mod 4

Jetzt könnten wir auch Fragen an dieses System stellen - etwa: „Gibt es einen geschlossenen Pfad der Länge 3?“ Dies wäre der Fall, wenn die folgende Formel erfüllbar ist:

$$F_R(x_1, x_2, y_1, y_2) \wedge F_R(y_1, y_2, z_1, z_2) \wedge F_R(z_1, z_2, x_1, x_2). \tag{6.5.1}$$

6.5.2 SAT-Solver

Dass die obige Formel nicht erfüllbar ist, kann man mit einem beliebigen *SAT-Solver* testen. Deren Inputsyntax ist sehr einfach – typischerweise werden SAT-Solver als Backend anderer Verifikationssysteme eingesetzt und in diesem Zusammenhang wird ihr Input automatisch erzeugt. Es existieren aber auch viele SAT-Solver, die man direkt über eine Web-Oberfläche testen kann. Wir haben das obige Beispiel mit einem auf *www.comp.nus.edu.sg/~gregory/sat/* bereitgestellten Solver getestet, der in Javascript geschrieben ist. Zuvor stellen wir die Formel in Konjunktiver Normalform (CNF) dar und erhalten:

$$\begin{aligned} F_R(x_1, x_2, y_1, y_2) \quad = \quad & (x_2 \vee y_2) \\ \wedge \quad & (\neg x_2 \vee \neg y_2) \\ \wedge \quad & (x_1 \vee x_2 \vee \neg y_1) \\ \wedge \quad & (\neg x_1 \vee \neg x_2 \vee \neg y_1) \\ \wedge \quad & (\neg x_1 \vee x_2 \vee y_1) \\ \wedge \quad & (x_1 \vee \neg x_2 \vee y_1). \end{aligned}$$

Aus Formel (6.5.1) ist also eine aussagenlogische (engl.: *propositional*) Formel in konjunktiver Normalform (*cnf*) mit 6 Variablen und 3*6=18 Klauseln geworden, was man in der Inputdatei des SAT-Solvers mit folgender Spezifikationszeile ankündigt:

```
p cnf 6 18
```

Es folgt die Angabe der Klauseln. Dabei werden die Variablen durch natürliche Zahlen repräsentiert, deren Negation durch die entsprechenden negativen Zahlen. Wir repräsentieren daher $x_1, x_2, y_1, y_2, z_1, z_2$ durch 1, 2, 3, 4, 5, 6. Mit 0 markiert man das Zeilenende und eine mit dem Buchstaben ´c´ beginnende Zeile ist ein Kommentar:

```
c    (x1,x2) --> (y1,y2) ist beschrieben durch:
 2  4  0
-2 -4  0
 1  2 -3 0
-1 -2 -3 0
-1  2  3 0
 1 -2  3 0
```

Diese 6 Klauseln repräsentieren also $F_R(x_1, x_2, y_1, y_2)$. Es folgen analog die Klauseln für $F_R(y_1, y_2, z_1, z_2)$ und $F_R(z_1, z_2, x_1, x_2)$. Übergeben wir den kompletten Text an den SAT-Solver (in dem obigen Solver fügen wir ihn mit cut/paste in das Inputfenster ein), so wird ohne erkennbare Verzögerung das Fenster rot, woran wir erkennen, dass die Formel nicht erfüllbar ist. Ersetzen wir die letzte Klausel durch die beiden Klau-

seln $F_R(z_1, z_2, u_1, u_2)$ und $F_R(u_1, u_2, x_1, x_2)$, fragen wir also nach der Existenz eines Pfades der Länge 4, so wird unser Fleiss durch ein grünes Fenster belohnt – es gibt also keinen Pfad der Länge 3, wohl aber einen der Länge 4.

6.5.3 Model Checking

Im Bereich des sogenannten „Model Checking“ wurden in den letzten 30 Jahren Werkzeuge entwickelt, mit denen man auf einfache Weise Systeme modellieren, spezifizieren und auf die Erhaltung von Eigenschaften prüfen kann. Eine dort angewandte Technik das „*Bounded Model Checking*“ macht im Prinzip nicht viel Anderes als das was wir soeben exerziert haben. Modelle und Fragestellungen werden in einer benutzerfreundlichen Sprache beschrieben und automatisiert in aussagenlogische Formeln compiliert und anschließend mit Hilfe eines SAT-Solvers nachprüft, ob diese erfüllbar sind. Dabei kann man mittlerweile mit Systemen umgehen, die Tausende von Booleschen Variablen beinhalten. Dies wird durch Fortschritte bei der praktischen Behandlung von SAT-Problemen möglich gemacht, die in den letzten Jahren erzielt worden sind.

Eine Kernidee beruht auf einer speziellen Darstellung boolescher Formeln durch „geordnete Entscheidungsbäume“, sogenannte „ordered binary decision diagrams“ kurz OBDDs.

Obwohl also SAT-Solving zu den schwersten und am wenigsten effizient lösbaren Problemstellungen der Informatik gehört, sollte dieses Ergebnis also nicht dazu verleiten, die Flinte ins Korn zu werfen. Mit Hilfe von heuristischen Methoden, die nicht immer, aber sehr oft gut funktionieren, hat man sehr gute Erfahrungen gemacht und nebenbei die Erkenntnis gewonnen, dass Problemstellungen, die in der Alltagspraxis entstehen, meist nicht so schlecht konditioniert sind wie die Probleme auf denen die Abschätzungen der Komplexitätstheorie basieren.

Literatur

Lehrbücher zur Theoretischen Informatik

Aho, Alfred; Lam, Monica; Sethi, Ravi; Ullman, Jeffrey: Compilers: Principles, Techniques, and Tools. Addison Wesley 2006.

Asteroth, Alexander; Baier, Christel: Theoretische Informatik. Eine Einführung in Berechenbarkeit, Komplexität und formale Sprachen. Pearson, München 2003.

Erk, Katrin; Priese, Lutz: Theoretische Informatik. Eine umfassende Einführung. 3. Auflage, Springer, Berlin 2008.

Hoffmann, Dirk W.: Theoretische Informatik. 3. Auflage, Carl-Hanser-Verlag 2015.

Linz, Peter: An Introduction to Formal Languages and Automata. Jones and Bartlett Publishers Inc., 5th ed., 2012.

Martin, John: Introduction to languages and the theory of computation. McGraw-Hill Inc., 1996.

Rich, Elaine: Automata, Computability, and Complexity. Prentice Hall, 2007.

Rodger, Susan; Finley, Thomas: JFLAP - An Interactive Formal Languages and Automata Package. Jones and Bartlett, 2006.

Schöning, Uwe: Theoretische Informatik - kurz gefasst. Springer Spektrum, 2008.

Vossen, Gottfried; Witt, Kurt-Ulrich: Grundkurs Theoretische Informatik. Springer, 2016.

Artikel

Davis M. (1978) What is a Computation? In: Steen L.A. (eds) Mathematics Today Twelve Informal Essays. Springer, New York, NY.

https://doi.org/10.1515/9783110442397-008

Stichwortverzeichnis

https://doi.org/10.1515/9783110442397-009