Einführung in die Computerlinguistik und Sprachtechnologie

Marcus Kracht
Fakultät LiLi
Universität Bielefeld
Postfach 10 01 31
33501 Bielefeld
marcus.kracht@uni-bielefeld.de

Inhaltsverzeichnis

1 Vorbemerkung 3

2 Praktische Bemerkungen zu OCaml 5

3 Willkommen im getypten Universum 9

4 Wie man Funktionen definiert 19

5 Module 24

6 Mengen und Funktoren 28

7 Kombinatoren 32

8 Grundbegriffe der Semantik 40

9 Objekte und Methoden 46

10 Buchstaben und Zeichenketten 49

11 Reguläre Ausdrücke 57

12 Reguläre Ausdrücke in OCaml 65

13 Endliche Automaten 73

14 Konstruktion eines Automaten aus einem Regulären Term 86

15 Minimale Automaten 91

16 Ein Wenig über Komplexität 98

17 Endliche Transduktoren 105

18 Finite State Morphologie 114

19 Transduktoren im Einsatz 119

20 Kontextfreie Grammatiken 125

21 Ambiguität 132

22 Parsing 136

23 Kategorialgrammatik 142

24 Semantik der Kategorialgrammatiken 149

25 Kombinatorische Kategorialgrammatik 154

B Symbole 160

C Index 160

Literaturverzeichnis 160

1 Vorbemerkung

Diese Vorlesung ist eine Einführung in die Computerlinguistik. Als solche ist sie auch eine Einführung in den Gebrauch des Computers. Das bedeutet, dass der Kurs nicht nur theoretische Methoden vorstellt sondern auch praktische, nämlich in Form von Programmieraufgaben. Im Prinzip setzt der Kurs keinerlei Programmierkenntnisse oder mathematisches Wissen voraus, aber gewisse Erfahrung im formalen Arbeiten und Argumentieren.

Nach einiger Überlegung habe ich mich entschlossen, die Programmierbeispiele in OCaml abzufassen, und zwar aus dem Grund, dass es streng getypt ist und seine Syntax sehr transparent. Python wäre auch gut gewesen, aber Python ist nicht ganz so mächtig, wir müssten also einen längeren Vorlauf hinnehmen, bevor wir an die höheren Aufgaben gehen können.

OCaml kann man hier bekommen (einfach auf das Link klicken):

```
http://caml.inria.fr/
```

Auf der dann erscheinenden Seite sollte man unter “Objective Caml” zu “Releases” gehen und kann dann die richtige Version laden. Viele Linux Distributionen bieten OCaml als Paket an (zum Beispiel SuSE und Ubuntu). Ein Tip für Versierte: wer zunächst Tcl/Tk installiert (nicht die zugehörigen dev-Pakete vergessen!) und dann erst OCaml, der bekommt auch einige Extras geliefert: dazu gehört der nützliche ocamlbrowser, mit dem man schnell die Befehle und ihre Syntax erfragen kann.

Wer ein etwas komfortableres Toplevel haben möchte (das einem die Befehle sogar anzeigt), sollte auch UTOP installieren. Dazu benötigt man als Erstes OPAM (OCaml Package Manager). Anschließend kann man UTOP mittels OPAM installieren (Befehl: `opam install utop`). Sofern alles glatt geht, hat man jetzt ein neue Toplevel System. Einfach `utop` eingeben, und es wird gestartet.

Man kann dazu ein Handbuch herunterladen (oder auf den hier gezeigten Link klicken). Das Handbuch ist zwar recht trocken (und zum Selbststudium nicht geeignet), aber es enthält eine Synopse aller internen Funktionen sowie der standardmäßig geladenen Module. Das ist recht praktisch zum Nachschlagen. Wer es geschafft hat, ocamlbrowser zu bekommen, hat damit auch ein Werkzeug in der Hand, um schnell mal während des Programmierens etwas nachschlagen zu

können. Ocamlbrowser gibt einem allerdings nur die Typen bekannt, während im Handbuch auch in wenigen Worten die Funktionsweise erklärt wird.

Dazu gibt es auch ein sehr schönes Buch in Englisch (und Französisch). Es läasst sich auf dieser Webseite finden, welche eine Liste von Büchern unter anderem in Englisch, Französisch und Deutsch enthält. Die aktuelle Version ist 4.02. OCaml kann man unter allen gängigen Plattformen installieren. Es ist leicht zu starten, und wir benötigen keinerlei zusätzliches Material oder Module. Der vorliegende Text enthält mehr, als in der Vorlesung selbst erwähnt werden kann, sollte deswegen allerdings auch selbsterklärend sein. Ich werde mich bemühen, sämtliche Programmierkonzepte zu erläutern. Ich weise auch darauf hin, dass Objekte streng genommen nicht nötig sind, man kann alles auch mit Modulen programmieren, insofern wäre `Caml` (ohne 'O') ausreichend.

Zum Thema Computerlinguistik gibt es ein paar Lehrbücher. Hier sei etwa das deutschsprachige [1] erwähnt. Dies hat für uns allerdings den Nachteil, dass die Dinge zwar schön beschrieben werden, aber leider keine Beweise geliefert werden. Das bedeutet, dass man nur begrenzt nachvollziehen kann, warum die Dinge so sind, wie sie sind. Außerdem ist die Darstellung an vielen Stellen knapp gehalten. Ansonsten ist auch die Homepage vom Natural Language Toolkit sehr anschaulich. Das ist eine Einführung in das NLTK. Das NLTK ist in Python geschrieben und enthält jede Menge Werkzeuge, um mit Texten zu hantieren. Ganz nebenbei wird allerdings auch viel an Erklärung geboten. Wer lieber mit Python als mit OCaml arbeiten will, sei darauf verwiesen.

2 Praktische Bemerkungen zu OCaml

Sobald Ocaml installiert ist, kann es wie folgt aufgerufen werden.

① Unter Windows, indem man das Ikon für das Programm anklickt. Dies öffnet ein interaktives Fenster wie das xterminal in `Unix`.

② Unter Unix oder Linux, indem man in der Konsole einfach `ocaml` tippt.

Danach meldet sich Ocaml mit einer schlichten Begrüßung, die nur die Versionsnummer enthält.

(1)
```
Objective Caml version 4.02.0
```

Dann ist es im interaktiven Modus. Das Prompt ist `#`. Wenn es erscheint, ist OCaml bereit, eine neue Eingabe entgegenzunehmen. Man kann dann ein Kommando eingeben, zum Bespiel

(2)
```
# 4+5;;
```

Dazu gleich ein paar wichtige Bemerkungen.

① Das # ist nicht vom Benutzer einzugeben.

② Am Ende einer Zeile muss man die Return-Taste drücken. (Dies werde ich nicht weiter erwähnen.)

③ Das doppelte Semikolon ist wichtig. Ocaml denkt sonst, das Kommando geht über mehrere Zeilen. Das Prompt, mit dem es sicht dann meldet, ist anders.

④ In vielen Fällen ist es nicht wichtig, ob man ein Leerzeichen setzt, oder ob man eine neue Zeile beginnt. Ocaml ist in dieser Hinsicht tolerant.

⑤ Aussteigen kann man mit `#quit;;`, wobei hier das # vom Benutzer selbst kommt. Es gibt eine Handvoll sogenannter toplevel-Befehle, die man nicht einbetten kann, und diese beginnen mit dem Doppelkreuz. (Und wie immer Return nicht vergessen.)

Man beherzige bitte Folgendes.

> Ich verwende folgende Konvention: alles, was der Nutzer eingibt, ist in blau geschrieben, und was OCaml antwortet in rot. Schreibmaschinenschrift sind echte Buchstaben, das heißt, Buchstaben, die eingegeben werden, oder die auf dem Bildschirm erscheinen. Alles andere sind nur Platzhalter (wie etwa <Zahl> ein Platzhalter für einen Zahlausdruck ist, das heißt, eine Folge von Ziffern).

Doch nun zurück zu unserer Sitzung (2). OCaml führt die Operation aus und meldet sich wie folgt zurück.

(3)
```
- : int = 9
#
```

Es gibt uns nicht nur Antwort, indem es uns sagt, dass das Ergebnis 9 ist, sondern es sagt uns auch, was für einen Typ der Wert hat, nämlich `int`. Das steht für **integer**, also ganze Zahl. Man bemerke, dass OCaml seine Zeile nicht mit Semikolon beendet; das Semikolon ist ja nur für uns da, damit wir wissen, wo eine Programmzeile beginnt. Die Bereitschaft zu neuer Eingabe wird von OCaml mit dem Prompt gezeigt.

Hier ist noch ein Beispiel.

(4)
```
# let a = 'q';;
```

Dies sagt OCaml, dass der Variablen `a` der Wert `q` zugewiesen. Hier ist die Antwort von OCaml.

(5)
```
val a : char = 'q'
```

(Und danach kommt der Prompt, was ich in Zukunft nicht weiter erwähnen werde.) OCaml sagt uns, dass dies ein Buchstabe ist (**character**). Dass es uns so versteht, liegt an den Anführungszeichen. Sind sie einfach, wie hier, so schließen sie einen Buchstaben ein; sind sie doppelt, so schließen sie eine Zeichenkette ein; doch davon später. Wir tippen nun

(6)
```
# Char.code 'q';;
```

OCaml antwortet uns wie folgt.

(7)
```
- : int = 113
#
```

Und wieder bekommen wir den Prompt. Das interaktive Arbeiten ist gut zum ersten Kennenlernen, aber mit der Zeit will man sich das dauernde Eintippen von stets demselben ersparen. Dann wird man die Eingabe in einer Datei vorbereiten und OCaml insgesamt übergeben wollen.

Dazu benötigt man einen Editor. Und zwar einen sehr einfachen, also kein Programm zum Briefeschreiben, sondern etwa Emacs oder Gvim. Diese laufen auf allen Plattformen. Windows liefert als einzig brauchbaren Editor Notepad, aber es ist unproblematisch, emacs oder gvim zu installieren. (Allerdings mag gvim für Neulinge etwas gewöhnungsbedürftig sein weil es zwischen Eingabe- und Kommandomodus unterscheidet. Erst mit der Zeit lernt man seine Vorzüge schätzen. Gvim jedenfalls hat eine Windows-ähnliche graphische Oberfläche.) Üblicherweise liefern solche Editoren auch Syntaxüberprüfung mit, was sehr nützlich ist.

Mit Hilfe des Editors müssen Sie eine Datei öffnen. Diese soll den Namen <meinedatei>`.ml` tragen, mit dem Suffix `.ml`. Hierbei ist, wie oben schon gesagt, <meinedatei> nicht etwa die Buchstabenfolge selbst (dann wäre sie in Schreibmaschinenschrift gesetzt), sondern ein Platzhalter dafür. Der Name sollte mit einem Kleinbuchstaben beginnen; das ist zwar in den meisten Fällen unerheblich, aber ich will es lieber gleich gesagt haben. Dann bekommt man später keinen Ärger. Die Datei enthält nun das Programm (wobei man sich hier das Markieren des Zeilenendes durch das doppelte Semikolon in der Regel sparen kann. OCaml weiß ja, von wem das alles stammt ...) Nun starten wir OCaml und sagen

(8)
```
# #use "<meinedatei>.ml";;
```

Also, wir haben jetzt zwei #, eines ist das OCaml Prompt, und das andere ist das Doppelkreuz, das zum Toplevelkommando gehört. Und weil dies zum Kommando gehört, darf kein Leerzeichen zwischen # und dem Wort `use` stehen. Wenn man alles richtig macht, wird OCaml die Datei einlesen und alles ausführen. Anders als beim Interaktiven Programm werden wir nicht mit Zwischenergebnissen beliefert. OCaml gibt uns lediglich eine Erklärung darüber, welche Funktionen oder Objekte jetzt definiert sind, und welchen Typen sie haben.

Allerdings wird es gerade am Anfang gar nicht so weit kommen. OCaml geht mehrmals durch die Datei durch (das macht es auch im interaktiven Modus). Das

erste Mal prüft es lediglich die Syntax. Da man die Syntax erst einmal lernen muss, ist es klar, dass man Anfang viele Syntaxfehler macht. Dann bleibt man bei der ersten Hürde stecken. Mit der Zeit gibt sich das. Beim zweiten Mal prüft OCaml durch, ob alle Typen korrekt und eindeutig sind. Wenn nicht, dann bekommen wir eine Fehlermeldung, die in etwa Folgendes sagt: dieser Ausdruck hier hat den Typ soundso, ich habe aber den Typ dasunddas erwartet. Dabei hört OCaml immer beim ersten Fehler auf. Das kann lästig sein, ist aber nicht zu vermeiden, weil das Programm ja nicht weiß, was anstelle dessen, was dasteht, hätte stehen sollen. Es ist deswegen praktisch, wenn man die Datei stets geöffnet hält und den Fehler ausbessert und gleich neu einliest. Es ist wichtig zu wissen, dass manche (nicht alle) Editoren anzeigen, an welcher Stelle der Cursor steht. Gvim zum Beispiel zeigt die Position wie folgt: `258,51`. (Allerdings ist Gvim unter Mac nicht so komfortablel.) Dies steht rechts unten im Fenster und bedeutet, dass der Cursor in Zeile 258 und an Position 51 (von links) sich befindet. OCaml gibt seinerseits bei der Fehlermeldung an, in welcher Zeile und an welcher Stelle der Fehler zu finden ist. Das erleichtert die Suche. (Gvim erlaubt im Kommandomodus mit der Eingabe `:`<Zahl><Return> das Springen auf die Zeile mit der Nummer <Zahl>.)

Die Programme kann man auf viele Dateien verteilen, allerdings behandelt OCaml jede Datei als Modul und eröffnet einen Namespace. Da wir in dieser Vorlesung keine großen Programme schreiben werden, wird uns das nicht weiter kümmern. Und schließlich noch ein wichtiger Hinweis.

> In einer Datei kann man jede Menge Kommentare unterbringen. **Kommentar** ist Text, der vom Programm nicht als Programmtext interpretiert wird. Damit das geschieht, muss der Kommentar als solcher markiert werden. Dies geschieht bei OCaml, indem man ihn wie folgt einschließt: der Beginn des Kommentars wird mit `(*` markiert, das Ende mit `*)`. Dazwischen darf jeder beliebige Text stehen (man sollte aber nach Möglichkeit `(*` und `*)` im Kommentar vermeiden, auch wenn das nicht verboten ist). Man darf auch Zeilenumbrüche unterbringen!

3 Willkommen im getypten Universum

In OCaml hat jeder Ausdruck einen Typ. Der Typ bestimmt, wie man das Objekt verwenden kann. Er kann nicht verändert werden; man kann nur neue Objekte anderen Typs von einem Objekt erstellen. Dies nennt man auch *statische Typzuweisung*. Aber das werden wir noch kennenlernen. Typen kann man selber definieren; es gibt voreingestelle Typen, die gängigsten davon sind

(9)

Zeichen	character	`char`
Zeichenkette	string	`string`
ganze Zahl	integer	`int`
reelle Zahl	float	`float`
Boolesch	boolean	`bool`

Es gibt Konventionen, wie man Objekte eines gewissen Typs dem Computer übergibt. Dies ist in jeder Programmiersprache so, und man muss halt diese Konventionen lernen und einhalten. Ein Zeichen (character) wird in einfache Anführungszeichen gesetzt: `'a'` bezeichnet zum Beispiel den Buchstaben `a`. Eine Zeichenkette muss in doppelte Anführungzeichen gesetzt werden: `"Kater"` bezeichnet die Zeichenkette `Kater`. Der Unterschied ist für OCaml durchaus wichtig. `"a"` ist eine Zeichenkette, die einen Buchstaben enthält, nämlich `a`. Obwohl auf dem Bildschirm normalerweise `'a'` und `"a"` gleich wiedergegeben werden, ist OCaml sehr penibel: die beiden sind nicht gleich, weil nicht gleichen Typs. Das ist, zumindest am Anfang, sehr lästig, aber auch sehr heilsam. So, wie wir zwischen der Ziffer `9` und der durch die Ziffer repräsentierten Zahl unterscheiden müssen (aber nicht immer tun), so müssen wir hier zwischen einem Buchstaben und seinem Vorkommen in einer Zeichenkette unterscheiden.

Der Typ `bool` hat nur zwei Werte, nämlich `true` und `false`. Zahlen werden wie üblich durch Ziffernfolgen repräsentiert. `10763` ist zum Beispiel eine Zahl (integer). Man beachte, dass `"10763"` für OCaml eine Zeichenkette ist, also vom Typ `string`, nicht `int`. Das ist nützlich. Zahlen vom Typ `int` sind zu unterscheiden von Zahlen vom Typ `float` (reelle Zahlen). Es gilt folgende Konvention: eine reelle Zahl hat stets irgendwo einen Punkt, also etwa `3.14`. Die Zahl 1 wird also als reelle Zahl so geschrieben: `3.0`. Die Null hinter dem Komma ist optional; `3.` tut es auch. Man muss man zwischen `1` (`int`) und `1.` (`float`) unterscheiden, nicht aber zwischen `1.` und `1.0`. OCaml hat dazu dieses zu sagen:

(10)
```
# 1.0 = 1;;
```

```
This expression has type int but is here used with type
    float
# 1. = 1.0;;
- : bool = true
```

Denn bei OCaml reicht es nicht aus, einfach nur materiell gleich zu sein; auch der Typ muss übereinstimmen. Die ganze Zahl 1 ist gleich groß wie die reelle Zahl 1, aber OCaml verlangt von uns eine eindeutige Typzuordnung. Manche Sprachen sind da etwas freizügiger und deklarieren eine ganze Zahl wenn nötig auch als reelle um und umgekehrt. Aber das hat seine eigenen Gefahren. Zum Beispiel ist die Wandlung einer reellen Zahl in eine ganze Zahl nicht eindeutig geregelt und sollte am besten nicht automatisch erfolgen.

OCaml unterstreicht hier das erste Vorkommen eines Ausdrucks, der problematisch ist. Bis zu `1.0 =` ist die Welt in Ordnung. Das Programm arbeitet sich dabei von links nach rechts vor, deswegen lässt es alle Information außer Acht, die eventuell noch kommen könnte. Das Gleichheitszeichen steht nach einer reellen Zahl, und OCaml erwartet jetzt eine reelle Zahl. Anstelle einer reellen Zahl kommt aber eine ganze Zahl, und OCaml beschwert sich bei uns. Ich weise gleich darauf hin, dass das Gleichheitszeichen wie man sagt **polymorph** ist, das heißt, mit vielen verschiedenen Typen gebraucht werden kann. Es darf zwischen zwei Ausdrücken beliebigen Typs stehen, vorausgesetzt, sie haben den gleichen Typ. Deswegen sind `true = false`, `"Katze" = "Hund"` und so weiter alle legal (aber diese beiden sind falsch; wahr dagegen sind `true = true`, `"Katze" = "Katze"`). Will man den Typ der Gleichheit angezeigt bekommen, so sage man Folgendes:

(11)
```
# (=) ;;
val f : 'a -> 'a -> bool = <fun>
```

Die Symbolik werde ich noch erklären. Für den Moment ist nur dies wichtig: “=” ist ein Infixsymbol, es steht *zwischen* seinen Argumenten. Infixsymbole müssen, wenn sie ohne Argumente gebraucht werden, in Klammern gesetzt werden, da OCaml normalerweise Präfixnotation benutzt. Dies gilt also auch für “+” und andere Infixsymbole.

Die Typen mögen lästig sein, aber die Hauptarbeit wird uns abgenommen. OCaml berechnet die Typen selbst. Angenommen, wir definieren eine Funktion

f, welche eine Zahl zu sich selbst addiert.

(12)
```
# let f x = x + x;;
val f : int -> int = <fun>
```

Das Zeichen + ist reserviert für die Addition von ganzen Zahlen. Das heißt, wir dürfen es nicht benutzen, um reelle Zahlen zu addieren. Deswegen sagt uns OCaml, dass die Funktion `f` eine Funktion ist, welche eine ganze Zahl benötigt und eine ganze Zahl ausgibt. Dies kann man aus dem Typ ablesen. Dieser ist `int -> int`. Dabei ist das Wort <fun> nur dazu da, um zu sagen, dass das Objekt eine Funktion ist.

Falls wir eine Funktion definieren wollen wie f, aber von reellen Zahlen nach reellen Zahlen, müssen wir ein anderes Additionszeichen nehmen, nämlich `+.`:

(13)
```
# let g x = x +. x;;
val g : float -> float = <fun>
```

Kommen wir noch einmal auf den Polymorphismus zurück. Nehmen wir einmal an, wir definieren die Identitätsfunktion.

(14)
```
# let iden x = x;;
val iden : 'a -> 'a = <fun>
```

OCaml erklärt uns, dies sei eine Funktion, und gibt den Typ mit `'a -> 'a` an. Dabei ist `'a` kein konkreter Typ sondern ein Platzhalter. Die Funktion ist somit einsetzbar für ein Argument gleich welchen Typs! Wir können das bei der Gleichheit auch vorführen:

(15)
```
# let gleich x y = (x = y);;
val gleich : 'a -> 'a -> bool = <fun>
```

Dies bedeutet soviel wie: das erste Argument (hier `x`) muss vom Typ `'a` (also beliebig) sein, das zweite Argument (`y`) vom Typ `'a` (also vom gleichen Typ wie `x`). Das Ergebnis ist wiederum vom Typ `bool` (also `true` oder `false`).

Im Folgenden werde ich ein wenig über die Gedanken hinter dem Typenuniversum sagen. Typen wurden ursprünglich von Russell eingeführt, um die Mengenabstraktion zu beschränken. Russell hatte gezeigt, dass man Dinge definieren kann, die es eigentlich gar nicht geben darf (die Menge aller Mengen, die sich

selbst nicht enthalten). Es entstand die Frage, wie man verhindern kann, dass solche Definitionen überhaupt zugelassen sind. Daraus entstand die Idee, Mengen einen Typ zuzuordnen. Falls man Mengen eines Typs α formen will, kann man dies nicht aus Mengen des Typs α tun, sondern muss dayzu Mengen niederen Typs nehmen. In Computersprachen liegt ein ähnlicher Gedanke zugrunde: man will verhindern, dass der Benutzer Sachen definiert (oder programmiert), die offenkundig nicht gehen können. Dazu wird jedem Objekt ein Typ zugeordnet, und diese Typen werden streng verwaltet.

Ein Typ ist ein sehr einfaches Objekt; es ist ein Term in einer beliebigen Signatur, die natürlich vorher festgelegt werden muss. Eine Signatur is ein Paar $\langle F, \Omega \rangle$, wo $\Omega : F \to \mathbb{N}$. Dabei ist F die Menge der Symbole, $\Omega(f)$ für jedes Symbol seine Stelligkeit. Ist $\Omega(f) = 0$, so ist f ein Grundsymbol oder **Konstante**. Alles anderen Symbole heißen **Typkonstruktoren**. Eine sehr beliebte Signatur, ursprünglich aus der Montague Grammatik, besteht (neben den Grundsymbolen e, s und t) aus den Typkonstruktoren $\to$ und $\bullet$. Es ist $\Omega(\to) = \Omega(\bullet) = 2$, dass heißt, beide Konstruktoren benötigen zwei Argumente. Wir spielen das Spiel mit dieser Konstellation einmal durch. Die abstrakte Situation mit beliebiger Signatur ist dann schnell gelernt.

Definition 3.1 (Typ) *Sei B eine Menge, der Menge der **Grundtypen**.* $\mathrm{Typ}_{\to,\bullet}(B)$*, die **Menge der Typen** über B, mit den zweistelligen Typkonstruktoren $\to$ und $\bullet$, ist die kleinste Menge derart, dass gilt*

- $B \subseteq \mathrm{Typ}_{\to,\bullet}(B)$*, und*
- *wenn* $\alpha, \beta \in \mathrm{Typ}_{\to,\bullet}(B)$ *dann auch* $\alpha \to \beta, \alpha \bullet \beta \in \mathrm{Typ}_{\to,\bullet}(B)$.

Jedem Typ α wird eine Interpretation M_α zugeordnet. Dies ist die Menge aller Objekte, die unter diesen Typ fallen. Wir verlangen, dass wenn α und β verschiedene Typen sind, so soll $M_\alpha \cap M_\beta = \varnothing$ sein, das heißt, kein Objekt darf zugleich vom Typ α als auch vom Typ β sein. Man mag glauben, dass dem so nicht ist. Zum Beispiel ist ja der Buchstabe k physisch gleich der Zeichenkette, welche aus dem einzigen Symbol k besteht. Aber man soll sich nicht täuschen; wir werden unten sehen, dass Zeichenketten sogenannte Arrays (oder Vektoren) sind, insofern ist es von der Implementierung fraglich, dass ein Vektor von Buchstaben gleich einem Buchstaben ist. Wir sehen an der Oberfläche keinen Unterschied, aber dass

heißt nicht, dass keiner existiert. Auf der anderen Seite ist die ganze Zahl 1 wertgleich der reellen Zahl 1, und doch macht OCaml einen Unterschied. Auch hier ist es so, dass die unterliegende Implementierung verschieden ist; außerdem gibt es auf den ganzen Zahlen andere Funktionen als auf den reellen. Insofern ist die Unterscheidung zumindest nützlich.

Seien nun M_α und M_β gegeben. Dann sei $M_{\alpha\to\beta}$ wie folgt definiert.

(16) $M_{\alpha\to\beta} = \{f : f \text{ ist eine Funktion von } M_\alpha \text{ nach } M_\beta\}$

Zum Beispiel können wir eine Funktion f durch die Vorschrift $f(x) = 2x + 3$ definieren. In OCaml tun wir dies wie folgt.

(17)
```
# let f x = (2 * x) + 3;;
val f : int -> int = <fun>
```

Jetzt verstehen wir besser, was OCaml uns mitteilt. Es sagt, dass der Wert von `f` den Typ `int -> int` hat. Denn `f` ist eine Funktion von ganzen Zahlen zu ganzen Zahlen. Man beachte, dass OCaml den Typ selber ausrechnet. Wir müssen dafür nichts tun. Aber wie berechnet OCaml den Typ eines Ausdrucks?

Dazu gibt es ein paar einfache Regeln. Falls x ein Objekt vom Typ $\alpha \to \beta$ ist und y ein Objekt vom Typ α, dann ist x eine Funktion, welche y als Wert nehmen kann. In dieser Situation ist die Kette, welche x gefolgt von y enthält (aber durch Leerzeichen getrennt), wohldefiniert und bezeichnet ein Objekt vom Typ β. OCaml erlaubt auch den Gebrauch von Klammern (welche manchmal natürlich notwendig sind). Aus diesem Grunde ist `f 43` ein wohlgeformter Ausdruck, wie auch `(f 43)` oder sogar `(f (43))`. OCaml sagt uns dazu Folgendes.

(18)
```
# f 43;;
- : int = 89
```

Das Ergebnis ist eine ganze Zahl, deren Wert 89 ist. Man darf der Funktion `f` auf jeden Ausdruck anwenden, der vom Typ `int` ist. Es kann Funktionszeichen enthalten, solange diese definiert sind. Die Symbole `*` und `+` sind vordefiniert. Im Handbuch steht, dass ihr Typ `int -> int -> int` ist, was bedeutet, dass sie Funktionen von ganzen Zahlen nach Funktionen von ganzen Zahlen nach ganzen Zahlen sind. Deswegen ist `(2 * x)` eine ganze Zahl, wenn `x` dies ist. Ebenso ist dann `(2 * x) + 3` vom Typ `int`.

Eine zweite Regel sagt, dass, wenn α und β Typen sind, so ist auch $\alpha \bullet \beta$ ein Typ, der **Produkttyp**. Dazu gehören alle Paare $\langle x, y\rangle$, bei denen x vom Typ α und y vom Typ β ist.

(19) $$M_{\alpha\bullet\beta} = M_\alpha \times M_\beta = \{\langle x, y\rangle : x \in M_\alpha, y \in M_\beta\}$$

Zur Paarbildung benutzt OCaml runde Klammern und in OCamls Schreibweise erscheint $*$ anstelle von $\bullet$, ansonsten ist alles wie beschrieben. Das Objekt `('a', 7)` ist wohldefiniert und vom Typ `char` $\bullet$ `int`. Bei OCaml sieht das so aus.

(20)
```
# ('a', 7);;
- : char * int = ('a', 7);;
```

Dies bedeutet, dass OCaml verstanden hat und dass das Objekt den Produkttyp aus character und ganze Zahl hat. (Die Klammern dürfen oft auch weggelassen werden. Ich rate jedoch davon ab.)

Man kann auch selber Typen definieren. Dazu verwendet man eine **Typendeklaration**.

(21)
```
# type prod = int * int;;
type prod = int * int
```

Diese Deklaration bindet den Identifikator `prod` and den Typ von Paaren von ganzen Zahlen, der von OCaml auch mit `int * int` bezeichnet wird. Da dazu nichts weiter zu sagen ist, wiederholt OCaml einfach zum Zeichen, dass es die Definition verstanden hat. Eine solche Deklaration macht natürlich wenig Sinn, wenn die Typen ohnehin von OCaml erkannt werden. Denn wenn wir anschließend das Objekt `(3, 4)` eingeben, dann sagt uns OCaml nicht etwas, dass es vom Typ `prod` ist, sondern sagt nur, es ist vom Typ `int * int`. Etwa so:

(22)
```
# let h = (3,4);;
val h : int * int = (3, 4)
```

Wenn wir es unbedingt so einrichten wollen, dass wir Objekte des neuen Typs bekommen, müssen wir unsere Typdeklaration anders gestalten. Wir benötigen dazu einen Typkonstruktor, den wir sogar selber definieren dürfen. Typkonstruktoren müssen mit einem Großbuchstaben beginnen. Wir definieren einen Konstruktor `Pr` wie folgt.

(23)
```
# type prod = Pr of int * int;;
type prod = Pr of int * int
```

Dies bedeutet, dass `Pr` eine Funktion ist, welche aus Objekten vom Typ `int * int` Objekte vom Typ `prod` macht.

(24)
```
# let h = Pr (3,4);;
val h : prod = Pr (3, 4)
```

Nunmehr haben wir tatsächlich ein Objekt vom Typ `prod`. Dies können wir daran sehen, dass Funktionen, die auf Paare anwendbar sind, nicht mehr zulässig sind für `h`.

Zum Beispiel gibt es die Funktionen `fst` und `snd`. Diese liefern zu einem Paar das erste bzw. zweite Element.

(25)
```
# fst (3,4);;
- : int = 3
# snd (3,4);;
- : int = 4
```

Die Funktionen `fst` und `snd` sind polymorph. Den Typ gibt uns OCaml mit `'a * 'b -> 'a` bzw. `'a * 'b -> 'b` an. Versuchen wir jetzt, eine dieser Funktionen auf eine Objekt vom Typ `prod` anzuwenden, beschwert sich OCaml bei uns. Das zeigt folgender Dialog im Anschluss an (23).

(26)
```
# fst (Pr (3,4));;
This expression has type prod but is
    used here with type 'a * 'b
```

Das ist gut so, denn `fst` und `snd` sind ja nur auf Paaren definiert, der Typ `prod` dient dazu, Paare zu verkapseln. Man kann sich (zum Glück) aber eine Funktion definieren, die dasselbe leistet:

(27)
```
# let erstes (Pr (x,y)) = x
val erstes : prod -> int = <fun>
```

Damit haben wir also eine Funktion `erstes` definiert, die einem `prod` das erste Element zuordnet. Dies lässt sich aber nun ihrerseits *nicht* auf Paare anwenden. Dafür ist nach wie vor `fst` zuständig.

Eine Alternative zum Paar ist das sogenannte **Record**. Am Besten sehen wir uns ein Beispiel an.

(28)
```
# type auto = {hersteller : string; jahr : int;
    gebraucht : bool};;
```

Dies definiert einen Record vom Typ `auto`, das drei Komponenten hat: einen *Hersteller*, ein *Herstellungsjahr*, sowie einen Hinweis, ob es gebraucht ist oder nicht. Diese Komponenten heißen **Felder**. Man beachte, dass die Felder Namen haben müssen, die mit Kleinbuchstaben anfangen. Ansonsten sind sie frei wählbar (man darf natürlich — wie immer — keinen Namen verwenden, der schon anderweitig vergeben ist). Ein Record kann beliebig viele Felder haben. Die Felder sind gleichberechtigt, und die Ordnung onter ihnen ist beliebig. Bei Paaren gibt es einen Unterschied zwischen `string * (int * bool)` oder `(string * int) * bool` oder sogar `bool * (string * int)`. Ein Objekt von diesem Typ bekommen wir wie folgt.

(29)
```
# let meinauto = {hersteller = "Honda"; jahr = 1977;
    gebraucht = false};;
val meinauto : {hersteller = "Honda"; jahr = 1977;
    gebraucht = false}
```

(Man beachte: der Herstellername ist eine Zeichenkette, also müssen die Anführungszeichen gesetzt werden. `false` ist eine Konstante, also keine Anführungszeichen.) Damit wird `meinauto` ein Element vom Typ `auto`. Um auf ein Feld zuzugreifen, bedient man sich der Notation <Record>.<Feld>. Also liefert zum Beispiel `meinauto.jahr` das (Herstellungs)jahr des Objektes `meinauto`. Ebenso liefert `meinauto.hersteller` den Namen des Herstellers. Wenn man ein Record definieren will, muss man alle Felder angeben. Die Reihenfolge ist dabei unerheblich.

(30)
```
# let seinauto = {hersteller = "BMW"};;
Some record field labels are undefined: jahr gebraucht
```

Zusätzlich wird der Text in Klammern unterstrichen, damit wir wissen, wo der Wurm drin ist. Das ist im Übrigen nicht nur eine Warnung, denn das Objekt

`seinauto` existiert nicht.

(31)
```
# seinauto.brand;;
Unbound value seinauto
```

Wenn man sich das Handbuch anschaut, wird man feststellen, dass OCaml eine Fülle von Konstruktoren hat, und die Beherrschung des vollen Mechanismus' erfordert Übung. Glücklicherweise benötigen wir nur einen Bruchteil davon. Die meisten Konstruktoren sind allerdings nicht im Kern von OCaml vorhanden, sondern ihrerseits mit Hilfe von OCaml selbst definiert.

Ein sehr wichtiger Konstruktor ist `list`. Dieser formt Listen von Objekten eines beliebigen Typs. Die Syntax ist einfach. Die Liste wird mit einer eckigen Klammer eröffnet und geschlossen; die Objekte einer Liste werden durch Semikolon getrennt.

(32)
```
# [2; 0; -7];;
- : int list = [2; 0; -7]
```

Aus jedem beliebigen Typ α lässt sich ein Typ der Listen von α-Objekten formen. So ist `['a'; 'u'; '0']` eine Liste von Characters, `[true; false]` eine Liste von Boolean, und so weiter. Ebenso gibt es Listen von Listen: `[[2; 0]; [7]]` ist ein Beispiel. Man beachte, dass die Liste `["Maus"]` verschieden ist von der Zeichenkette `"Maus"`. OCaml hat einen Typkonstruktor namens `list`; dies ist ein sogenannter **Postfixoperator**, weil er seinem Argument nachgestellt wird.

(33)
```
# ['a'; 'u'; '0'];;
- : char list = ['a'; 'u'; '0']
```

Dies bindet den Identifier `u` an den Typ von Listen von Zahlen.

Es gibt auch die Schreibweise mit dem doppelten Semikolon. a`::`ℓ bezeichnet die Liste, die aus der Liste ℓ durch voranstellen des Elements a entsteht. `[]` ist die leere Liste. Dann ist zum Beispiel `'a' :: []` nichts weiter als `['a']`, und `'b' :: ['a']` nichts weiter als `['b'; 'a']`. Konkatenation von Listen wird durch `@` bezeichnet. Dies ist ein sogenannter **Infixoperator**, weil er zwischen seinen Argumenten steht (in OCaml eher die Ausnahme). Man beachte, dass die Elemente einer Liste den gleichen Typ haben müssen.

(34)
```
# let h = [3; "st"; 'a'];;
This expression has type string but is here used with
    type int
```

Wiederum müssen wir verstehen, dass der Fehler aus der Sicht von OCaml beim zweiten Listenelement auftritt. Denn das erste ist eine Zahl, und Listen von Zahlen sind durchaus legale Objekte. Aber weil das nächste Element eine Zeichenkette ist, ist die Liste nicht wohlgeformt und OCaml beschwert sich bei uns.

4 Wie man Funktionen definiert

Funktionen kann man in OCaml auf verschiedene Weisen definieren. Die erste ist die allgemein übliche, in der man explizit sagt, was diese Funktion tun soll.

(35)
```
# let appen x = x^"a";;
val appen : string -> string = <fun>
```

Diese Funktion hängt an die Zeichenkette den Buchstaben `a` (genauer: sie verkettet sie mit der Zeichenkette, die aus dem einzigen Buchstaben `a` besteht). Man beachte, dass die Definition mit `let` begonnen wird. Das erste, was nach `let` kommt, ist der Name der Funktion. (Nachher werden wir auch die Definition `let rec` kennenlernen.) Im Anschluss an den Funktionsnamen folgen die Argumente der Funktion und dann ein Gleichheitszeichen. Das Gleichheitszeichen signalisiert, welche der Identifier Argumente der Funktion sind. Man muss sich klarmachen, dass OCaml keine Ahnung hat, was wir definieren wollen. Es weiß von der Funktion nicht, wie viele und welche Argument sie hat und erschließt das einzig aus der Syntax. Die Namen von Variablen sind ja überwiegend frei, solange sie nicht mit einem Großbuchstaben beginnen und anderweitig vergeben werden. Für OCaml ist zum Beispiel dieses eine legale Definition, obwohl die Variable `y` nicht verwendet wird.

(36)
```
# let appen x y = x^"a";;
val appen : string -> 'a -> string = <fun>
```

Es gibt auch noch eine andere Art, welche explizit sagt, wo die Argumente sind.

(37)
```
# let app = (fun x -> x^"a");;
val app : string -> string = <fun>
```

Die Variable vor dem Gleichheitszeichen ist nun verschwunden. Diese Methode hat vor allem den Vorzug, dass wir der Funktion keinen Namen geben müssen, um auf sie zurückzugreifen. Anstelle von `app` schreibt man einfach `(fun x -> x^"a")`. Ansonsten besteht kein Unterschied in der Funktionsweise.

Wie in allen andern Programmiersprachen auch, haben wir Verzweigungen (`if`$\cdots$`then`$\cdots$`else`). OCaml hat aber auch noch ein anderes sehr nützliches Instrument, nämlich `match`$\cdots$`with`. Es erlaubt, einfache Objekte, eine Liste, ein Paar, oder mehrere Objekte gleichzeitig abzufragen und je nach Beschaffenheit

der Objekte die Funktion zu definieren. Die Fälle in der Klausel werden durch einen senkrechten Strich abgetrennt.

(38)
```
# let f p = match p with
      (n,true) -> n | (n,false) -> -n;;
val f : int * bool -> int = <fun>
```

Dies definiert eine Funktion von Paaren (n, b), wo n eine Zahl und b ein Boolescher Wert ist, nach Zahlen. Und zwar ist der Wert n, wenn b wahr ist und sonst $-n$.

Ein fundamentales Werkzeug zur Definition von Funktion ist die sogenannte **Rekursion**. Rekursion ist ein Verfahren, bei dem die Funktion anscheinend durch sich selbst definiert wird. Nehmen wir zum Beispiel an, wir haben nur die Nachfolgerfunktion $x \mapsto x + 1$. Damit können wir die Addition von zwei natürlichen Zahlen wie folgt definieren.

(39) $$0 + y := y; (n + 1) + y := (n + y) + 1$$

Damit das weniger banal aussieht, schreibe ich $s(x)$ für den Nachfolger von x, und dann haben wir

(40) $$0 + y := y; s(n) + y := s(n + y)$$

Das können wir auch etwas anders formulieren:

(41) $$x + y := \begin{cases} y & \text{falls } x = 0 \\ s(n + y) & \text{falls } x = s(n) \end{cases}$$

Also ist die Addition wie folgt erklärt: ist $x = 0$, dann ist $x + y$ einfach gleich y. Ansonsten (weil x eine natürliche Zahl ist) ist x der Nachfolger einer Zahl, sagen wir n. Dann ist $x + y := s(n + y)$. Sehen wir uns mal an, warum das eigentlich funktioniert. Wir wollen $3 + 4$ ausrechnen. Anhand der Definition finden wir, dass dies $s(2 + 4)$ ist, denn 3 ist der Nachfolger von 2. Nun haben wir immer noch kein Ergebnis, aber vielleicht können wir ja $2 + 4$ ausrechnen. Da 2 der Nachfolger von 1 ist, bekommen wir $s(1 + 4)$. Wir wiederholen den Schritt und bekommen $1 + 4 = s(0 + 4)$. Jetzt sind wir am Ziel: $0 + 4 = 4$, und das Ergebnis ist $s(s(s(4))) = 7$.

(42)
$$\begin{aligned} 3 + 4 &= s(2 + 4) \\ &= s(s(1 + 4)) \\ &= s(s(s(0 + 4))) \\ &= s(s(s(4))) \\ &= 7 \end{aligned}$$

OCaml muss man speziell warnen, dass man eine rekursive Funktion definieren will. Dazu dient `let rec` anstelle von `let`. Hier ist der Programmkode:

(43)
```
let rec add x y =
    if (x = 0) then y
    else 1 + (add (x-1) y);;
```

Ich weise noch einmal darauf hin, dass die Einrückung nicht nötig ist, wir können darauf verzichten. Ebenso ist es unerheblich, wo die Zeile endet, solange wir kein Wort trennen.

Wir können Rekursion auch für Funktionen auf Zeichenketten verwenden. Hier ist ein Beispiel.

(44)
$$d(\vec{x}) := \begin{cases} \varepsilon & \text{falls } \vec{x} = \varepsilon \\ aad(\vec{y}) & \text{falls } \vec{x} = a\vec{y} \end{cases}$$

Die letzte Definition sieht etwas anders aus als die erste. ε bezeichnet das leere Wort. Also ist zunächst einmal $d(\varepsilon) = \varepsilon$. Die zweite Klausel sagt, dass, wenn $\vec{x} = a\vec{y}$ ist für einen Buchstaben a und eine Zeichenkette $\vec{y}$, dann ist $d(\vec{x}) = aad(\vec{y})$.

Der Trick hinter der Rekursion ist folgender. Wir definieren eine Funktion $f(x)$ dadurch, dass wir einen Wert $f(y)$ für ein anderes Objekt y ausrechnen und daraus $f(x)$ berechnen. Damit das überhaupt geht, müssen wir sicherstellen, dass y in einem gewissen Sinne einfacher ist als x. Und am Ende, wenn es keine einfacheres Objekt als x selbst gibt, nennen wir den Wert der Funktion. Dieses letzte Detail wird gerne vergessen: ohne eine explizite Definition des einfachen Fallsist die Rekursion nicht fundiert und läuft ins Leere. Es genügt keineswegs, einfach nur zu sagen, dass $s(x) + y = s(x + y)$ ist. Schauen wir mal an, was OCaml dazu sagt:

(45)
```
# let rec add x y = 1 + (add (x-1) y);;
val add : int -> 'a -> int = <fun>
# add 3 4;;
Stack overflow during evaluation (looping recursion?)
```

OCaml hat versucht, den Ausdruck zu berechnen und dabei zu viel Platz verbraucht; eine nachträgliche Analyse kommt zu dem Schluss, wir haben wohl eine Rekursion, die nicht terminiert. Das stimmt in diesem Fall sogar. Der Typ der Funktion wird übrigens mit `int -> 'a -> int` angegeben. Das liegt daran, dass die Grundklausel fehlt. OCaml weiß effektiv nicht, welchen Typ das zweite Argument hat. Nur das erste steht fest, weil der Ausdruck `x-1` einen Hinweis auf den

Typ gibt. Auch dass das Ergebnis eine ganze Zahl ist, kann OCaml erschließen. Nur bei dem zweiten Argument hat OCaml Probleme. Ein ähnlicher Fall ergibt sich, wenn wir die originale Definition (43) nehmen und Folgendes eingeben.

(46)
```
# let rec add x y =
    if (x = 0) then y
    else 1 + (add (x-1) y);;
val add : int -> int -> int = <fun>
# add (-3) 4;;
Stack overflow during evaluation (looping recursion?)
```

Das Problem ist, dass `add (-3) 4` durch `add (-4) 4` erklärt wird, dies wiederum durch `add (-5) 4`, und so weiter, und wir niemals an eine Ende kommen. (Im Gegensatz dazu ist `add 4 (-3)` unproblematisch. Können Sie entdecken, warum?) Das Problem ist hierbei, dass die Rekursion nur für eine natürliche Zahl als Parameter gemacht ist. Wir müssten also verhindern, dass sie für andere Zahlen gebraucht wird. Wir müssen also OCaml erlauben, die Reißleine zu ziehen. Da wir in diesem Fall kein Ergebnis wollen, muss ein besonderes Objekt her. Dies sind die sogenannten **Ausnahmen**. Diese erlauben, gezielt aus einer Berechnung auszusteigen. Hier ist ein Beispiel.

(47)
```
# exception Eingabe_negativ;;
exception Eingabe_negativ
```

Ausnahmen müssen mit einem Großbuchstaben beginnen. Ausnahmen können wir benutzen, um an beliebiger Stelle im Programm auszusteigen.

(48)
```
# let rec add x y =
    if (x < 0) then raise Eingabe_negativ
    else if (x = 0) then y
    else 1 + (add (x-1) y);;
```

Wenn wir jetzt eine negative Zahl eingeben, wirft uns OCaml wunschgemäß aus dem Programm und teilt uns auch mit, warum.

(49)
```
# add (-3) 4 ;;
Exception: Eingabe_negativ.
```

Ein sehr wichtiger Fall für Rekursion ist Rekursion über Datenstrukturen, insbesondere Listen. Nehmen wir mal an, wir haben eine Liste von ganzen Zahlen

und wollen jede dieser Zahlen mit sich selbst multiplizieren (quadrieren). Dazu können wir folgendes Programm schreiben.

(50)
```
# let rec quad l =
    match l with
    [] -> []
    | h :: t -> (h * h) :: (quad t) ;;
val quad : int list -> int list = <fun>
# quad [3; 5];;
- : int list = [9; 25]
```

Das ist eine sehr schöne Art, wie man Funktionen auf Elemente in einer Liste einzeln anwendet. Das Standardmäßig geladene Modul `List` hat sogar eine Funktion, die ganz abstrakt erlaubt, eine Funktion vom Typ `'a -> 'b` auf eine Liste von Elementen des Typs `'a` anzuwenden. Man bekommt dann eine Liste von Elementen des Typs `'b`. Diese Funktion heißt `List.map`. Wir können sie auch leicht selbst basteln.

(51)
```
# let rec listmap f l =
    match l with
    [] -> []
    | h :: t -> (f h) :: (listmap f t) ;;
val listmap : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

Bliebe uns noch eine letzte Frage, nämlich, warum wir nicht genauso auch mit Zahlen so umgehen können:

(52)
```
match x with 0 -> ... | (x-1) -> ...
```

Der Grund liegt in der Tatsache, dass das Matching nur über Strukturen erfolgt. Eine Zahl ist keine Struktur, also kann der Matcher nichts darüber wissen, was `x-1` ist. Denn der Matcher rechnet nicht. Ebenso kann man nicht über Zeichenketten matchen, aber über Records und Paare. Das ist eine Entscheidung, die getroffen wurde, um Fallunterscheidungen sehr schnell laufen zu lassen. Der Matcher soll schnell arbeiten, und das kann er nicht, wenn er auch noch rechnen soll.

5 Module

Module sind ein Weg, um Programmteile zu einem Paket zusammenzuschnüren. Wenn Programme sehr groß sind, lohnt es sich, es in **Module** zu teilen. Dies kann entweder implizit geschehen, indem man das Programm auf mehrere Dateien verteilt, oder explizit, indem man Module vereinbart. Die Syntax ist denkbar einfach:

(53)
```
module <Modulname> =
  struct
Folge von Definitionen
  end;;
```

Der Modulname muss mit einem Großbuchstaben beginnen. Ein Modul besteht aus einem eigenständigen Programm, insofern wird man sich fragen, wozu so ein Modul nützt. Die Antwort liegt in der Verwaltung von Namen, die damit zusammenhängt. Nehmen wir mal an, wir hätten ein Modul mit dem Namen `Seminar` vereinbart. Dieses enthalte irgendwelche Funktionen, zum Beispiel die Funktion `sprecher`. Innerhalb des Moduls können wir diese Funktion unter diesem Namen aufrufen; außerhalb des Moduls allerdings heißt sie `Seminar.sprecher`.

Nehmen wir nun an, wir wollen einen Programmteil vollständig separat halten, oder wir wollen ein schon einmal geschriebenes Programm benutzen. Dieses liegt in einer Datei mit Namen <Name>, etwa `seminar.ml`. Darin liegt das Programm, allerdings unverpackt (also ohne Einfassung in eine Modulstruktur). Dann können wir dieses Programm über den Befehl `#use "seminar.ml"` laden. (Das Doppelkreuz kündigt ein sogenanntes Toplevel-Kommando an. Diese darf man nicht in Funktionsdefinitionen einbetten. Es gibt nicht viele davon, die wichtigsten sind `#quit`, `#use` und `#load`, jeweils ohne Leerzeichen!) In diesem Fall steht es uns direkt zur Verfügung. Wir können es aber auch kompilieren, dann erzeugt OCaml unter anderem eine Datei namens `seminar.cma`. Dies ist eine Bibliotheksdatei. Diese können wir jetzt mit dem folgenden Befehl laden.

(54)
```
# #load "seminar.cma";;
#
```

Nun stehen die Funktionen ebenfalls zur Verfügung; jedoch hat OCaml die Bibliothek als eigenes Modul verpackt. Die Funktion `sprecher` muss jetzt als `Seminar.sprecher` aufgerufen werden. Der Modulname leitet sich dabei aus dem Dateinamen ab. Hätte die Datei `logik.ml` geheißen, so wäre der Name des Moduls nunmehr

`Logik` und die Funktion nunmehr `Logik.sprecher`. Jetzt wird auch klar, warum die Dateinamen in OCaml am besten mit einem Kleinbuchstaben beginnen: OCaml kommt aus irgendeinem Grund durcheinander, weil die Dati und der Modul den gleichen Namen tragen anstelle dass die Datei mit einem Kleinbuchstaben beginnt, der Modul aber mit einem Großbuchstaben.

Sehen wir uns das einmal an. Auf meinem System finden wir die Bibliotheken von OCaml in folgendem Verzeichnis.

```
usr/lib/ocaml/
```

Dort gibt es eine Datei namens `list.ml` und eine Datei namens `list.mli`. Die Datei `list.ml` enthält Funktionen des Moduls `List`, die im Handbuch erklärt sind (weil das Modul standardmäßig dazugehört). Dieses Modul ist in OCaml programmiert und kann (im Gegensatz zu den kompillierten Dateien, etwa `list.cmo`, `list.cmi`, `list.cma`, und einige mehr), im Klartext gelesen werden. Das Typeninterface heißt `list.mli`, und man kann es selbst erstellen, wenn man will.

Die erste Funktion, die man dort findet ist `length`.

```
let rec length_aux len = function
    [] -> len
  | a::l -> length_aux (len + 1) l

let length l = length_aux 0 l
```
(55)

Um diese Funktion zu verwenden, muss man sie `List.length` nennen, denn ist standardmäßig als Modul `List` geladen. Die Rolle von `list.mli` ist dabei die, den anderen Modulen mitzuteilen, welche Funktionen das neugeschaffen Modul `List` zur Verfügung stellt. Dabei kann man Funktionen, die in der Datei nicht aufgeführt sind, nicht zugreifen, auch wenn sie definiert sind. Dies ermöglicht, gewisse Funktionsweisen geheimzuhalten, oder auch einfach, ein wenig Übersichtlichkeit zu schaffen. Die Funktion `length_aux` ist deswegen in `list.mli` nicht aufgeführt, weil sie niemand wirklich braucht.

Schauen wir uns die Definition von `length_aux` etwas genauer an. Diese ist eine Funktion von zwei Argumenten. Das erste, `len`, ist vor dem Gleichheitszeichen untergebracht, das zweite wird über das Wort `function` eingeführt. Dabei werden gleich zwei Fälle erörtert: das zusätzliche Argument ist die leere Liste oder von der Form `a :: l`. Im ersten Fall ist der Wert `len`. Im zweiten Fall ist das Ergebnis `length_aux (len+1) l`. Diese Definition ist recht trickreich. Wenn man

den Boden erreicht hat, das heißt, wenn man bei 0 ankommt, ist das Ergebnis bereits da. Die folgende Definition ist langsamer:

(56)
```
let rec length l =
    match l with
    [] -> 0
    | h :: t -> 1 + (length t)
```

Schauen wir uns das an. Links sehen wir die zweite Definition, rechts die in OCaml implementierte.

(57)
```
length [3; 4; 5]          length [3; 4; 5]
1 + length [4; 5]         length_aux 0 [3; 4; 5]
1 + (1+ length [5])       length_aux 1 [4; 5]
1 + (1+ (1 + length []))  length_aux 2 [5]
1 + (1+ (1 + 0))          length_aux 3 []
1 + (1+ 1))               3
1 + 2
3
```

Die linke ist halb so schnell, weil sie am Ende der Rekursion das Ergebnis schrittweise ausrechnen muss.

Was in der Dateistruktur die Datei `list.mli` ist, ist in der Modulstruktur die Signatur. Diese lässt sich auch direkt wie folgt spezifizieren.

(58)
```
module type <Signaturname> =
  sig
eine Folge von Typdefinitionen
  end;;
```

Man beachte, dass durch die Ansage `module type` OCaml darauf hingewiesen wird, dass es sich nicht um ein Modul sondern um eine Signatur handelt, die hier definiert wird. Die Signatur sagt lediglich, was für Funktionen welchen Typs das Modul bereitstellen soll.

OCaml besteht aus drei Teilen: dem **Kern**, der **Grundsprache** und den **Erweiterungen**. Alles, was nicht in der Grundsprache ist, wird in Form eines Moduls programmiert. Das Handbuch listet zwei bis drei Dutzend solcher Module auf, die standardmäßig geladen werden, wenn man OCaml aufruft. (Man kann Module hinzufügen, indem man die Datei `.ocamlinit` definiert, doch soll dies hier nicht behandelt werden.)

Zu guter Letzt noch ein Hinweis. Wenn man sich die langen Namen sparen will, zum Beispiel `List.length`, so kann man auch sagen

(59) `open` <Modulname>`;;`

Dann darf man anstelle der langen Namen auch die kurzen Versionen benutzen. Dies geschieht auf eigenes Risiko, denn es kann sein, dass man in zwei Modulen denselben Namen vergeben hat. Dann gilt nur die neuere Definition, die alte ist unzugänglich.

6 Mengen und Funktoren

Am Beispiel der Mengen kann man sehr anschaulich die Funktionsweise von Typkonstruktoren sehen. Nebenbei werden wir auch einen Einblick in Funktoren bekommen.

Nehmen wir also an, wir wollen genauso wie Listen auch Mengen von einem beliebigen Typ haben. Wir könnten uns vorstellen, dass es einen analogen Postfixkonstruktor `set` gibt, der aus einem Typ α den Typ der Mengen von α-Objekten macht. Das ist allerdings nicht so einfach. OCaml möchte nämlich, bevor es solch einen Typ bereitstellt, wissen, wie die Objekte des Typs α geordnet werden sollen. Denn intern gibt es keine Mengen, sondern diese werden als binär verzweigende Bäume implementiert. Für den Aufbau dieser Bäume ist eine lineare Ordnung auf den Elementen nötig. Diese Ordnung muss eine Funktion sein, welche für zwei Elemente *a* und *b* sagt, ob *a* kleiner als *b* ist, ob *a* gleich *b* ist oder ob *a* größer als *b* ist. Diese Funktion hat als Wert -1, 0, oder 1, hat also den Typ α `-> int`.

Wie nun kann OCaml eine solche Funktion geben? Zunächst einmal existiert eine Funktion `compare`, welche für jeden definierten Typ α eine solche Funktion darstellt. Dies kann man immer nehmen, aber man muss es nicht tun. Wenn uns also nichts Besseres einfällt, sagen wir Folgendes.

```
module OPStrings =
  struct
    type t = string * string
    let compare = compare
  end;;
```
(60)

Diese Definition klingt merkwürdig. Definieren wie nicht gerade `compare` durch sich selbst? Die Antwort ist: das tun wir nicht. Denn auf der linken Seite des Gleichheitszeichens definieren wir die Funktion genannt `OPStrings.compare` auf der rechten benutzen wir `compare`. Wir können auch, wenn wir Missverständnisse vermeiden wollen, die Bezeichnung `Pervasives.compare` benutzen. Es

steht uns frei, eine Funktion selbst zu definieren:

```
(61)  module OPStrings =
        struct
          type t = string * string
          let compare x y =
            if x = y then 0
            else if fst x > fst y || (fst x = fst y
                 && snd x > snd y) then 1
            else -1
        end;;
```

Darauf antwortet OCaml dieses:

```
(62)  module OPStrings :
      sig type t = string * string val compare :
          'a * 'b -> 'a * 'b -> int end
```

Dies besagt, dass nun ein Modul namens `OPStrings` zur Verfügung steht, dessen Grundtyp der Typ `string * string` (Paare von Zeichenketten) ist, und der eine Funktion `compare` von dem Typ `'a -> 'a -> int` ist. Wir haben nun einen neuen Typ `OPStrings.t`, welcher gleich `string * string` ist. OCaml sagt wiederum nicht, was die Funktionen tun, sondern nur, welchen Typ sie haben. Dies ist die Signatur, welche in `sig`$\cdots$`end` eingeschlossen ist.

Aber wo sind jetzt die Mengen? Die sind noch nicht da; das Modul `OPStrings` wird lediglich gebraucht, damit daraus ein neues Modul `PStringSet` entsteht. Dies entsteht jetzt aber nicht dadurch, dass wir explizit alles durchprogrammieren. Sondern es entsteht aus `OPStrings` durch Anwenden eines *Funktors*. Dieser Funktor wird im Modul `Set` bereitgestellt. Er heißt (im Modul) `Make`. Deswegen müssen wir ihn unter dem Namen `Set.Make` aufrufen.

```
(63)  module PStringSet = Set.Make(OPStrings);;
```

Wenn wir dies tun, antwortet mit einer langen Liste von Typdefinitionen. Diese können wir im Handbuch unter dem Modul `List` nachlesen. Der Funktor leistet hier ganze Arbeit. So stellt er die Funktion `add` bereit, welche man jetzt unter dem Name `PStringSet.add` verfügbar hat. Dazu gibt es die leere Menge, genannt `empty`. Tippen wir zum Beispiel

```
(64)  # st = PStringSet.empty;;
```

so bezeichnet `st` von nun an die leere Menge von Paaren von Zeichenketten. Mit dem folgenden Befehl bekommen wir die Menge {(`cat`, `mouse`)}:

(65)
```
# PStringSet.add ("cat", "mouse") st;;
```

Eine Alternative dazu ist

(66)
```
# PStringSet.singleton ("cat", "mouse");;
val st : PStringSet.t = <abstr>
```

Das Handbuch gibt Auskunft zu allen Funktionen.

Wie oben bereits gesehen, kann man diese Mengen eigentlich gar nicht so direkt anschauen. Der Typ `PStringSet.t` ist abstrakt (OCaml schreibt `<abstr>`). Das bedeutet, dass uns OCaml nicht zeigt, wie die Mengen aussehen. Wir werden, anders als bei Basistypen, Paaren oder Listen nicht weiter aufgeklärt, was für eine Objekt wir da haben. Das ist kein böser Wille. OCaml hat schlicht keine Ahnung, wie es sich uns mitteilen soll. Damit wir also eine Menge "sehen" können, gibt es zwei Wege. Erstens wir programmieren explizit ein Verfahren, das uns die Menge so zeigt, wie wir sie sehen wollen. Das andere arbeitet mit der Funktion `elements`. Diese macht aus einer Menge eine Liste, bei der die Elemente aufsteigend geordnet sind.

(67)
```
# let h = PStringSet.elements st;;
- : PStringSet.elt list = [("cat", "mouse")]
```

Ich weise noch darauf hin, dass man sich die Definition eines Moduls namens `OPStrings` wie folgt sparen kann.

(68)
```
module PStringSet =
  Set.Make(struct
    type t = string * string
    let compare = compare
  end);;
```

Dies bewirkt dasselbe, nur ist `OPStrings` nicht definiert.

Ähnlich wie Mengen werden auch Hashtabellen definiert. Hashtabellen sind nichts weiter als schnelle Implementierungen von Funktionen, die auf endlich vielen Werten definiert sind. Eine Hashtabelle wird zwischen zwei Typen definiert. Das Argument der Funktion heißt auch Schlüssel (**key**). Wenn man eine

Hashtabelle in OCaml definieren will, benutzt man wiederum einen Funktor. Dazu müssen wir eine Signatur liefern, die aus zwei Funktionen besteht. Die eine, `equal`, sagt, wann zwei Elemente gleich sind. Die zweite, `hash`, rechnet zu jedem Schlüsselement eine Zahl aus.

(69)
```
module HashedTrans =
  struct
    type t = int * char
    let equal x y = (((fst x = fst y)
    && (snd x = snd y)))
    let hash x = ((256 * (fst x))
    + (int_of_char (snd x)))
  end;;

module HTrans = Hashtbl.Make(HashedTrans);;
```

Wir haben hier eine Hashtabelle mit Schlüssel Paeren von Zahlen und Buchstaben definiert. Die Hashfunktion ist $(n, c) \mapsto 256n + \mathit{int\text{-}of}(c)$.

Wiederum stehen Defaultfunktionen zur Verfügung. Diese heißen `Hashtbl.hash` sowie `(=)`. Die Klammern sind deswegen notwendig, weil das Gleichheitszeichen ein Infixsymbol ist (siehe Seite 54). Jetzt steht der Typ von Hashtabellen zur Verfügung. Man beachte, dass der Typ des Werts dieser Tabellen nicht festgelegt ist. Man kann ihn frei wählen. Wenn man allerdings das Objekt definiert, muss der Typ feststehen (was meist ohnehin der Fall ist).

7 Kombinatoren

Kombinatoren sind Ausdrücke, die aus nichts anderem bestehen, als aus geschachtelten Funktionsaufrufen. Kombinatoren sind deswegen für uns wichtig, weil die Notation von OCaml direkt auf kombinatorischer Logik beruht. Außerdem können wir sehr schön die Wirkungsweise des Typmatchers beobachten. In OCaml können Funktionen technisch immer nur auf ein Argument angewendet werden. Ist f eine Funktion und x ein Objekt, so ist fx oder (fx) das Ergebnis der Anwendung von f auf x. fx kann auch wiederum eine Funktion sein, die dann wieder auf etwas angewendet werden kann, und so weiter. Ebenso kann das Argument einer Funktion seinerseits eine Funktion sein. Wendet man fx auf y an, so dürfen wir einfach fxy schreiben, lediglich mit dem Leerzeichen als Trennzeichen. Diese Ausdrücke denken wir uns links assoziativ geklammert, in diesem Fall also $((fx)y)$. Ebenso ist $fxyz$ nichts weiter als $(((fx)y)z)$, und so weiter.

Die gebräuchlichsten Kombinatoren sind I, K und S. Sie sind wie folgt definiert.

$$\begin{aligned} \mathsf{I}x &:= x \\ \mathsf{K}xy &:= x \\ \mathsf{S}xyz &:= xz(yz) \end{aligned} \tag{70}$$

In OCaml können wir diese Definition fast wörtlich übernehmen.

```
# let i x = x;;
val i : 'a -> 'a = <fun>
# let k x y = x;;
val k : 'a -> 'b -> 'a = <fun>
# let s x y z = x z (y z)
val s : ('a -> 'b -> 'c) -> ('a -> 'b) -> 'a -> 'c
    = <fun>
```
(71)

Wir wollen genau unter die Lupe nehmen, wie OCaml eigentlich auf seine Antworten kommt. Natürlich gibt OCaml jedem Kombinator einen Typ. Dabei verwendet es Platzhalter (Variable) über Typen. Die Kombinatoren sind also polymorph oder typenabstrakt. Man beachte auch, dass OCaml Klammern sparsam verwendet. Dabei gilt, anders als bei Funktionen, die rechtsassoziative Schreibweise. Also ist $x \to y \to z$ kurz für $x \to (y \to z)$, $v \to x \to y \to z$ kurz für $v \to (x \to (y \to z))$. Dies liegt daran, dass die Typen spiegelverkehrt zu den

Funktionsausdrücken aufgeschrieben werden (das erste Argument ist rechts von der Funktion, erscheint als Typ aber links vom Zieltyp). Der letzte Typ is zum Beispiel

(72) `(('a -> ('b -> 'c)) -> (('a -> 'b) -> ('a -> 'c)))`

Dies bedeutet, dass das erste Argument den Typ `('a -> ('b -> 'c))` haben, das zweite den Typ `'a -> 'b` und das dritte den Typ `'a`.

Hier noch ein paar Worte zur Syntax von OCaml. Alles, was man definiert und kein konkretes Objekt ist (also eine Zeichenkette oder ein Buchstabe), wird ohne syntaktischen Zucker hingeschrieben (keine Anführungszeichen) und heißt **Identifier** (**Identifikator**). Es heißt so, weil es ein Objekt bezeichnet. Identifikatoren dürfen nicht beliebig sein. Das liegt daran, dass OCaml relativ gnädig ist, wenn es um Klammern geht. Wir dürfen zum Beispiel das Leerzeichen vor einer Klammer weglassen, weil die Klammer nicht zum Identifikator gerechnet wird. Das bedeutet für uns natürlich, dass wir keine Identifikatornamen nehmen dürfen, welche Klammern enthalten. Sie sollten auch nicht nur aus Ziffern bestehen, weil sie dann ja als Zahl gelesen werden. In jedem Fall müssen sie mit einem Buchstaben beginnen und sollten ansonsten nur Buchstaben, Ziffern, Bindestrich und Unterstrich enthalten. Dabei gibt es eine klare Regelung, welche Identifikatoren mit einem Großbuchstaben beginnen müssen und welche mit einem Kleinbuchstaben. Variable und Typen beginnen mit Kleinbuchstaben, Typkonstruktoren und Modulnamen mit Großbuchstaben. Das ist der Grund, warum die Kombinatoren in OCaml jetzt `i`, `k` und `s` heißen und nicht `I`, `K`, `S`. Die genaue Syntax wird im Handbuch erklärt, ist aber nur etwas für Leute, die es ganz genau wissen wollen. Die Syntax ist in Backus-Naur-Form abgefasst, ohne weitere Erklärungen.

Identifikatoren müssen eindeutig sein. Das bedeutet, dass ein und derselbe Identifikator nicht zweimal vergeben werden kann. OCaml löst das Problem so, dass stets die neueste Definition gilt. Die alte ist dann allerdings verloren. Weil wir das nicht wollen, müssen wir aufpassen, nicht denselben Identifikator zweimal zu vergeben.

Wir wollen nun mit unseren Kombinatoren ein wenig experimentieren.

(73)
```
# k i 'a' 'b';;
- : char = 'b'
```

Aufgrund der Konvention wird der Ausdruck als linksgeklammert betrachtet, das heißt, ist äquivalent zu `((k i) 'a') 'b'`. Nun ist Kla = I, nach Definition von

K, und es ist lb = b. Eine andere Klammerung liefert ein anderes Ergebnis.

(74)
```
# k (i 'a') 'b';;
- : char = 'a'
```

Ein anderer Kombinator ist P. Dieser ist die Projektion auf das zweite Argument, das heißt P$xy := y$. Hier ist eine Definition mit Hilfe von I, K.

(75)
```
# let p x y = k i x y;;
val p = 'a -> 'b -> 'b = <fun>
```

Wie können wir feststellen, dass dies die richtige Definition ist? Was wir nicht tun können, ist einfach `p x y` abzufragen. Denn `x` und `y` werden von OCaml als Identifikatoren gelesen und müssen in diesem Moment definiert oder gebunden sein. Das ist nicht der Fall. Wir können aber die Zeichenketten `"x"` und `"y"` füttern. Das funktioniert, weil OCaml keinerlei Zeichenkettenfunktionen anwendet, das heißt, die Variablennamen werden nicht verändert.

(76)
```
# p x y ;;
Unbound value x
# p "x" "y" ;;
- : string = "y"
```

Wenden wir uns noch kurz dem Typenmatcher zu. Betrachen wir folgenden Dialog.

(77)
```
# s i i;;
This expression has type ('a -> 'b') -> 'a -> 'b but
    is used here with type ('a -> 'b) -> 'a
```

Wie kommt OCaml dazu, dass dieser Ausdruck nicht wohlgeformt ist? Dies geht folgendermaßen. OCaml betrachtet den Ausdruck `s i` als einen Teilterm und wertet ihn aus. Dazu werden zunächst die Typen bestimmt. Da `i` das Argument von `s` ist (die Funktion steht immer vor ihrem Argument), muss sein Typ gleich dem des ersten Arguments von `s` sein. Das erste Argument von `s` hat den Typ `'a -> 'b -> 'c`, und `i` hat den Typ `'a -> 'a`. Damit diese gleich sind, muss `'a = 'b -> 'c` sein. Damit können wir `'a` eliminieren, bekommen für `i` den Typ `('b -> 'c) -> ('b -> 'c)`, und für `s` den Typ

(78)
```
(('b -> 'c) -> 'b -> 'c)
-> (('b -> 'c) -> 'b) -> ('b -> 'c) -> 'c
```

Wir wenden nun `s` auf `i` an und erhalten

(79) `(('b -> 'c) -> 'b) -> ('b -> 'c) -> 'c`

Nun wollen wir diesen Ausdruck erneut auf `i` anwenden. Das bedeutet, wir müssen die folgende Gleichung lösen:

(80) `('b -> 'c) -> 'b = 'a -> 'a`

Dies bedeutet, dass wir für `'a` den Ausdruck `'b -> 'c` einsetzen müssen. Wenn wir dies tun, bekommen wir allerdings dieses Ergebnis:

(81) `('b -> 'c) -> 'b = ('b -> 'c) -> 'b -> 'c`

An diesem Punkt stockt OCaml. Denn diese Typen sind nicht gleich, denn es folgt unmittelbar, dass `'b = 'b -> 'c` sein muss, und das geht nicht. Es teilt uns nun Folgendes mit: falls man die Funktion anwenden will, so muss das Argument den Typ `('b -> 'c) -> 'b` haben, aber es hat (nach einiger Rechnung, wie wir gerade gesehen haben) den Typ `('b -> 'c) -> 'b -> 'c`. Aber das ist nicht haargenau, was wir mitgeteilt bekommen. Der Unterschied liegt hier aber nur darin, dass OCaml die Variablen umbenannt hat; es schreibt `'a` anstelle von `'b` und `'b` anstelle von `'c`. Das ist aber völlig in Ordnung (Variable darf man umbenennen). OCaml versucht, mit Name immer sparsam zu sein.

Ich beende das Kapitel mit einem allgemeinen Algorithmus, wie man den Typ eines Ausdrucks bestimmt bzw. nachrechnet, dass das Objekt nicht existiert. Ich wähle dazu die Definition des Kombinators W:

(82) $\mathsf{W}xy := xyy$

Dazu bestimmen wir sämtliche Teilterme aller Ausdrücke. Die Teilterme sind:

(83) $x, y, xy, xyy, \mathsf{W}, \mathsf{W}x, \mathrm{W}xy.$

Jeder Teilterm hat einen Typ. Dieser ist im Moment noch unbekannt, also eine Variable.

(84) $x : \alpha, y : \beta, xy : \gamma, xyy : \delta, \mathsf{W} : \varepsilon, \mathsf{W}x : \eta, \mathrm{W}xy : \theta.$

Da $\mathsf{W}xy = xyy$, muss $\theta = \delta$ sein, und so können wir θ eliminieren:

(85) $x : \alpha, y : \beta, xy : \gamma, xyy : \delta, \mathsf{W} : \varepsilon, \mathsf{W}x : \eta, \mathrm{W}xy : \delta.$

Ab jetzt gehen wir wie folgt vor. Der Typenkalkül besitzt eine einzige Funktionalgleichung.

Applikation. Ist MN ein Term vom Typ γ, wo M den Typ α und N den Typ β hat, so ist $\alpha = \beta \to \gamma$.

Dies gibt uns folgende Gleichungen.

$$\begin{array}{l} \alpha{=}\beta \to \gamma \\ \gamma{=}\beta \to \delta \\ \varepsilon{=}\alpha \to \eta \\ \eta{=}\beta \to \delta \end{array} \tag{86}$$

Gesucht ist ε. Die letzte Gleichung liefert einen Wert für η:

$$\begin{array}{l} \alpha{=} \quad \beta \to \gamma \\ \gamma{=} \quad \beta \to \delta \\ \varepsilon{=}\alpha \to \beta \to \delta \end{array} \tag{87}$$

Die erste Gleichung gibt uns den Wert für α:

$$\begin{array}{l} \gamma{=} \qquad\quad \beta \to \delta \\ \varepsilon{=}(\beta \to \gamma) \to \beta \to \delta \end{array} \tag{88}$$

Zu guter Letzt setzen wir für γ ein.

$$\varepsilon = (\beta \to \beta \to \delta) \to \beta \to \delta \tag{89}$$

Um ein Gleichungssystem mit Typen im allgemeinen Fall zu lösen, gibt es zwei Regeln. Die erste haben wir schon angewendet:

Regel A. Es enthalte das Gleichungssystem die Gleichung $\alpha = M$, wo α eine Variable ist und M ein Term. Es gibt drei Fälle.

1. M ist identisch mit α (das bedeutet, wir haben effektiv die Gleichung $\alpha = \alpha$). Dann kann man diese Gleichung ersatzlos streichen.
2. M enthält die Variable α nicht. Dann ersetze man in allen anderen Gleichungen α durch M und streiche die Gleichung.
3. M enthält die Variable α (aber $M \neq \alpha$). Dann ist das Gleichungssystem nicht lösbar.

Der letzte Fall verdient Beachtung. Denn in diesem Fall hat M die Form $K \to L$. Die Gleichung ist deswegen nicht lösbar, weil für Typausdrücke ohne Variable $M = N$ nur dann gelten kann, wenn sie syntaktisch gleich sind. Das bedeutet im Einzelnen Folgendes.

> **Regel B.** Es sei $M = K \to L$ eine Gleichung. Diese ist nur dann erfüllbar, wenn M die Form $A \to B$ hat, und in diesem Fall ist die Gleichung gleichwertig mit der Kombination der *zwei* Gleichungen $A = K$ und $B = L$.

Wir haben bei $M = K \to L$ zwei Fälle: der erste ist, dass M eine Variable ist; diesen Fall haben wir schon betrachtet. Der zweite Fall ist jetzt auch abgehandelt.

Betrachten wir noch einmal den Term SII. Wir wollen sehen, ob er einen Typ hat. Wir haben

(90) $\mathsf{S} : (\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to (\alpha \to \gamma)$

I hat den Typ $\delta \to \delta$; der erste komplexe Term, SI, habe den Typ ε. Wir bekommen mit der Applikationsregel daraus diese Gleichung:

(91) $(\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to (\alpha \to \gamma) = (\delta \to \delta) \to \varepsilon$

Wenden wir die Regel B an, bekommen wir

(92)
$$\begin{array}{ll} \alpha \to \beta \to \gamma & = \delta \to \delta \\ \varepsilon & = (\alpha \to \beta) \to (\alpha \to \gamma) \end{array}$$

Daraus bekommen wir, erneut mit Regel B,

(93)
$$\begin{array}{ll} \alpha & = \delta \\ \beta \to \gamma & = \delta \\ \varepsilon & = (\alpha \to \beta) \to (\alpha \to \gamma) \end{array}$$

Die erste Gleichung kann eliminiert werden, indem δ durch α ersetzt wird.

(94)
$$\begin{array}{ll} \beta \to \gamma & = \alpha \\ \varepsilon & = (\alpha \to \beta) \to (\alpha \to \gamma) \end{array}$$

Und schließlich ersetzen wir auch α:

(95) $\varepsilon = ((\beta \to \gamma) \to \beta) \to ((\beta \to \gamma) \to \gamma)$

Anschließend wird der Term SI auf I angewendet. Letzteres hat den Typ $\zeta \to \zeta$. Der Zieltyp (der Typ von SII) ist η. Hierzu ist Folgendes zu sagen: verschiedene Vorkommen einer Variablen haben immer denselben Typ. Variable sind also nicht polymorph. Hingegen sind die Kombinatoren polymorph, sodass verschiedene Vorkommen desselben Kombinators durchaus veschiedenen Typ haben können, weswegen wir $\zeta \to \zeta$ ansetzen und nicht einfach $\delta \to \delta$.

$$\begin{array}{l} \varepsilon=((\beta \to \gamma) \to \beta) \to ((\beta \to \gamma) \to \gamma) \\ \varepsilon=(\zeta \to \zeta) \to \eta \end{array} \tag{96}$$

Daraus erhalten wir eine einzige Gleichung

$$((\beta \to \gamma) \to \beta) \to ((\beta \to \gamma) \to \gamma) = (\zeta \to \zeta) \to \eta \tag{97}$$

Diese wiederum zerfällt in zwei Gleichungen:

$$\begin{array}{l} (\beta \to \gamma) \to \beta=\zeta \to \zeta \\ (\beta \to \gamma) \to \gamma=\eta \end{array} \tag{98}$$

Die erste Gleichung zerfällt wiederum in zwei Gleichungen.

$$\begin{array}{ll} \beta \to \gamma & =\zeta \\ \beta & =\zeta \\ (\beta \to \gamma) \to \gamma & =\eta \end{array} \tag{99}$$

Die ersten beiden Gleichungen ergeben

$$\beta \to \gamma = \beta \tag{100}$$

Dies ist nicht erfüllbar. Auch dies muss man streng genommen zeigen, da ja β und γ variable sind. Angenommen, wir setzen für β den (variablenfreien) Typ M und für γ den Typ N. Dann bekommen wir

$$M \to N = M \tag{101}$$

Es enthalte M m Vorkommen des Pfeils, N hingegen n Vorkommen des Pfeils. Dann enthält der linke Term $m + n + 1$ Vorkommen des Pfeils, der rechte aber nur m. Die Terme müssen aber syntaktisch gleich sein. Das ist nicht möglich.

Ein letzter Hinweis. Die Typen bilden eine sogenannte absolut freie Algebra. Gleichheit ist in einer solchen Algebra immer identisch mit rein syntaktischer

Gleichheit. In solchen Algebren gibt es also, wie wir gerade gesehen haben, sogenannte allgemeinste Lösungen für Gleichungssysteme. Das soll bedeuten, dass ein Gleichungssystem mit Variablen derart gelöst werden kann, dass es eine Substitution in die Variablen dergestalt gibt, dass sämtliche Lösungen Substitutionsinstanzen dieser einen Substitution sind. Das Resultat lässt sich also wiederum abstrakt formulieren. Dies erklärt, wie OCaml typenabstrakt arbeiten kann. Entweder der Ausdruck besitzt keinen Typ, dann wird er abgelehnt, oder er besitzt einen abstrakten Typ der Form, dass sämtliche Substitutionsinstanzen auch tatsächlich valide Instanzen sind.

8 Grundbegriffe der Semantik

Um zu verstehen, wofür Kombinatoren überhaupt gebraucht werden, eignet sich die Semantik. Eines der Grundprobleme der Semantik ist das Folgende: man errechne die Bedeutung eines komplexen Ausdrucks aus der Bedeutung seiner Teile. Dazu eine einfache Überlegung. Ein einfacher, intransitiver Satz besteht aus einem Subjekt und einem Prädikat:

(102) `Johann singt.`

In der Semantik bestimmt man die Bedeutung eines Namens als einen Gegenstand, während die Bedeutung eines Satzes ein Wahrheitswert ist. Was aber ist die Bedeutung eines intransitiven Verbs? Dies ist schlicht eine Funktion, die jedem Gegenstand einen Wahrheitswert zuordnet. In der Sprache der Typen gesprochen haben Eigennamen den Typ e (“entities”) und Sätze den Typ t (“truth value”), und somit haben intransitive Prädikate den Typ $e \to t$. Wie aber kommt die Bedeutung des Satzes zustande? Hier sagt man, dass die Bedeutung des Satzes die Anwendung der Bedeutung des Prädikats auf den vom Subjekt bezeichneten Gegenstand ist. Wir machen dies wie folgt kenntlich:

(103)
`Johann`	:e	$::j$
`singt`	:e $\to$ t	$::f$
`Johann singt.`	:t	$::\mathsf{R}jf$

Hierbei ist

(104) $\mathsf{R}xy := yx$

R ist ein Kombinator, er beschreibt die “Rückwärtsanwendung” einer Funktion auf sein Argument. Wir haben also $Rjf = f(j)$, wie wir das wollten. In einer Sprache, in der das Verb vor dem Subjekt steht (Gaelisch), tritt an die Stelle von R der Kombinator A, definiert durch

(105) $\mathsf{A}xy := xy$

Nehmen wir ein konkretes Beispiel. Der Bereich der Individuen kann im Prinzip alles sein, deswegen nehmen wir der Einfachheit halber die ganzen Zahlen (`int`). /`Johann`/ ordnen wir die Zahl 0 zu und /`singt`/ diejenige Funktion `f` von `int` nach

bool, die 0 den Wert true zuordnet und jeder anderen Zahl den Wert false. Das bedeutet, dass in diesem Fall der Satz /Johann singt./ wahr ist.

(106)
```
# let johann = 0;;
val johann : int = 0
# let singt x = if x = 0 then true else false;;
val singt : int -> bool = <fun>
# let r x y = y x;;
val r : 'a -> ('a -> 'b) -> 'b = <fun>
# r johann singt;;
- : bool = true
# r 1 singt;;
- : bool = false
```

Damit haben wir unser erster semantisches Modell gebaut, zusammen mit einem Lexikon aus zwei syntaktischen Objekten und fertiger Semantik. Die Bedeutung einer Konstituente wurde in diesem Fall dadurch errechnet, dass die Bedeutung der zweiten Konstituente auf die der ersten angewendet wurde; kurz, es wurde der Kombinator R angewendet.

Haben wir zwei Sätze, so können wir sie unter anderem durch das Wort /und/ verbinden. Dieses Wort steht zwischen seinen Argumenten. Wie bei OCaml auch hat es folgende Semantik, wobei $\mathsf{und}'xy := 0$ genau dann, wenn $x = 0$ oder $y = 0$, und $\mathsf{und}'11 := 1$. Ich werde es wie in den Büchern üblich so halten, dass die Bedeutungen der Worte dadurch bezeichnet werden, dass das Wort in sans-serife Schrifttyp gesetzt wird und ein Apostroph drangehängt wird. Das erspart mir die langwierige Erklärung, um welche Funktion es sich jetzt handelt. Es ist zwar dann immer noch nicht wirklich klar, welche Funktion dies konkret ist, aber in den hier geschilderten Fällen lassen sich die Details mühelos einfügen.

(107) $\mathsf{und} : \mathsf{t} \to \mathsf{t} \to \mathsf{t} :: \mathsf{und}'$

Wir erweitern unser Lexikon wie folgt:

(108)
```
# let peter = 1;;
val peter : int = 1
# let rennt x = if x = 1 then true else false;;
val rennt : int -> bool = <fun>
# let a x y = x y;;
val a : ('a -> 'b) -> 'a -> 'b = <fun>
# let und x y = (x && y);;
val und : bool -> bool -> bool = <fun>
# (r (r johann singt) (a und (r peter rennt)));;
- : bool = true
```

Soweit, so gut. Die Semantik eines Satzes bekommen wir also, indem wir eine Konstituentenstruktur "raten" und anschließend mit Hilfe von Vorwärts- oder Rückwärtsanwendung die Bedeutungen zu Funktion-Argumentpaaren verschrauben, bis am Ende ein Wahrheitswert herauskommt. Dies war in der Tat die ursprüngliche Vorstellung. Transitive Verben lassen sich damit zum Beispiel zwanglos einordnen.

Ein Problem entsteht allerdings in folgender Konstruktion.

(109) `Johann rennt und singt.`

Das Problem ist, dass das Wort /und/ nicht zwei Wahrheitswerte verbindet sondern nur Funktionen in Wahrheitswerte. Dafür ist die ursprüngliche Semantik also nicht gemacht. Allerdings tut es das so, dass am Ende auch wieder eine Konjunktion dasteht, so, als hätte anstellen von (109) das Folgende dagestanden:

(110) `Johann rennt und Johann singt.`

Das Wort /Johann/ steht aber nur einmal da, obwohl es zweimal verwendet wird. Außerdem haben die Sätze verschiedene Konstituenten:

(111) [Johann [rennt [und singt]]]

(112) [[Johann rennt] [und [Johann singt]]]

Um dies hinzubekommen, verändern wir nicht etwa die Semantik von /und/ sondern erlauben vielmehr in diesem Fall den (optionalen) Gebrauch des folgenden Kombinators:

(113) $\mathsf{S}_1 wxyz = w(xz)(yz)$

Die Bedeutung von /und/ wird jetzt sowohl und' wie auch $(\mathsf{S}_1\ \mathsf{und}')$ sein. Die Bedeutung des Satzes oben wird dann

(114) $(\mathsf{R}\,\mathsf{johann}'(\mathsf{R}\,\mathsf{rennt}'\,(\mathsf{A}\,(\mathsf{S}_1\ \mathsf{und}')\,\mathsf{singt}')))$

Sehen wir nun nach, ob wir bekommen, was wir wollten.

$$
\begin{aligned}
& (\mathsf{R}\,\mathsf{johann}'(\mathsf{R}\,\mathsf{rennt}'\,(\mathsf{A}\,(\mathsf{S}_1\ \mathsf{und}')\,\mathsf{singt}'))) \\
= \ & (\mathsf{R}\,\mathsf{rennt}'\,(\mathsf{A}\,(\mathsf{S}_1\ \mathsf{und}')\,\mathsf{singt}')))(\mathsf{johann}') \\
(115) \quad = \ & (\mathsf{A}\,(\mathsf{S}_1\ \mathsf{und}')\,\mathsf{singt}')))(\mathsf{rennt}')(\mathsf{johann}') \\
= \ & (\mathsf{S}_1\ \mathsf{und}')(\mathsf{singt}')(\mathsf{rennt}')(\mathsf{johann}') \\
= \ & \mathsf{und}'(\mathsf{singt}'(\mathsf{johann}'))(\mathsf{rennt}'(\mathsf{johann}'))
\end{aligned}
$$

Wenn man weiter nachdenkt, wird man sehen, dass auch transitive Verben mittels /und/ verbunden werden können. Auch dafür kann man einen Kombinator finden:

(116) $\mathsf{S}_2 vwxyz = v(wyz)(xyz)$

Dies erscheint recht merkwürdig: offenkundig "zaubern" wir immer neue Bedeutungen für das Wort /und/ hervor. Woher kommt dafür die Rechtfertigung?

Die Rechtfertigung kommt im Wesentlichen daher, dass Kombinatoren nichts hinzuerfinden. Sie kommen ohne konkrete Funktionen aus. Man kann also keinen Kombinator erfinden, der zu einer Zahl 1 hinzuzählt. Alles, was Kombinatoren tun, ist, das gegenseitige Spiel von Funktion und Argument zu regeln. Wortbedeutungen erhalten so eine Flexibilität, die wir uns für die natürliche Sprache wünschen.

Hier noch ein weiteres Beispiel. Der Quantor /jeder/ hat den Typ $(e \to t) \to (e \to t) \to t$. Dies ersieht man daraus, dass er mit einem Nomen kombiniert und anschließend mit einem intransitiven Verb. Die Semantik eines allgemeinen Nomens ist dieselbe wie die eines intransitiven Verbs: eine Funktion von Gegenständen nach Wahrheitswerte (Typ $e \to t$).

(117) `Jeder Student lernt.`

Hier ist dazu das Lexikon.

	`jeder`	$:(e \to t) \to (e \to t) \to t$	$::\mathsf{jeder}'$
(118)	`Student`	$:e \to t$	$::\mathsf{student}'$
	`lernt`	$:e \to t$	$::\mathsf{lernt}'$

Hierbei ist $\mathsf{jeder}' PQ = 1$ genau dann, wenn für jedes x aus $P(x)$ auch $Q(x)$ folgt.

(119) $(\mathsf{A}\,(\mathsf{A}\,\mathsf{jeder}'\,\mathsf{student}')\,\mathsf{lernt}')$

Es hat also /`jeder Student`/ den Typ $(e \to t) \to t$. Nun können wir es aber auch mit /`Johann`/ verbinden:

(120) `Johann und jeder Student`

Da Konjunktion nur dann funktionieren darf, wenn die Konjunktionsglieder den gleichen Typ haben, stehen wir vor einem Rätsel. Auch hier gibt es Abhilfe in Form des Anhebungskombinators T:

(121) $\mathsf{T}xy = yx$

Dies ist im Prinzip genau der rückwärtsgerichtete Applikationskombinator (also R), den wir schon verwendet haben. Hier aber benutzen wir ihn als einstellige Funktion, wie wir das gewohnt sind: wir haben in $\mathsf{R}xy$ eine Konstituente $\mathsf{R}x$. Da der Wert (und damit der Typ) von $\mathsf{R}xy$ auch noch von y abhängt, ist $\mathsf{R}x$ typabstrakt. Der Typ des Kombinators ist $\alpha \to (\alpha \to \beta) \to \beta$. Da der Typ von x gegeben ist, errechnet sich daraus ein abstrakter Typ, der nur noch von β abhängt. Setzen wir zum Beispiel $\alpha := e$, so bekommen wir $e \to (e \to \beta) \to \beta$, und damit ist der Typ des Ausdrucks $(\mathsf{T}\,\mathsf{johann}') = (\mathsf{R}\,\mathsf{johanna}')$ genau $(e \to \beta) \to \beta$. Damit hat es (mit $\beta := t$) schon die richtige Form, um koordiniert werden zu können. (Der Grund, warum ich hier T und nicht R benutze, ist rein historischer Natur.) Wir müssen nur noch die Bedeutung von /`und`/ richtig anheben, dann sind wir fertig.

(122)
$$\begin{aligned}
& (\mathsf{A}(\mathsf{A}\,(\mathsf{R}\,\mathsf{johann}')\,(\mathsf{A}\,(\mathsf{S}_1\,\mathsf{und}')\,(\mathsf{A}\,\mathsf{jeder}'\mathsf{student}')))\,\mathsf{lernt}') \\
= \; & (\mathsf{A}\,(\mathsf{R}\,\mathsf{johann}')\,(\mathsf{A}\,(\mathsf{S}_1\,\mathsf{und}')\,(\mathsf{A}\,\mathsf{jeder}'\mathsf{student}')))(\mathsf{lernt}') \\
= \; & (\mathsf{A}\,(\mathsf{R}\,\mathsf{johann}')\,(\mathsf{A}\,(\mathsf{S}_1\,\mathsf{und}')\,(\mathsf{jeder}'\,\mathsf{student}')))(\mathsf{lernt}') \\
= \; & (\mathsf{A}\,(\mathsf{R}\,\mathsf{johann}')\,(\mathsf{S}_1\,\mathsf{und}'\,(\mathsf{jeder}'\,\mathsf{student}')))(\mathsf{lernt}') \\
= \; & (\mathsf{A}\,(\mathsf{R}\,\mathsf{johann}')\,(\mathsf{S}_1\,\mathsf{und}'\,(\mathsf{jeder}'\,\mathsf{student}')))(\mathsf{lernt}') \\
= \; & (\mathsf{S}_1\,\mathsf{und}'\,(\mathsf{jeder}'\,\mathsf{student}')(\mathsf{R}\,\mathsf{johann}')\,\mathsf{lernt}' \\
= \; & \mathsf{und}'(\mathsf{jeder}'\,\mathsf{student}'\,\mathsf{lernt}')(\mathsf{R}\,\mathsf{johann}'\,\mathsf{lernt}') \\
= \; & \mathsf{und}'(\mathsf{jeder}'\,\mathsf{student}'\,\mathsf{lernt}')(\mathsf{lernt}'\,\mathsf{johann}')
\end{aligned}$$

Man beachte, wie die beiden Konstituenten letztlich getrennte Wege gehen. Während /`jeder Student`/ das Verb als sein Argument nimmt, ist es bei /`Johann`/ letztlich umgekehrt. Das Wichtigste aber ist: sämtliche Kombinatoren sind verschwunden, wir haben nur noch die Basisfunktionen, verbunden durch Applikation. Nun kann man also den Wert dieses Ausdrucks ausrechnen, und man bekommt einen Wahrheitswert.

Diese letzte Eigenschaft ist wichtig, wenn man verstehen will, was der Typenkalkül leistet: wir können anhand des Typs abschätzen, welcher Art das Objekt ist, welches wir bekommen. Ferner können wir, sofern der Typ einer Konstituente einfach ist, sämtliche Kombinatoren eliminieren.

9 Objekte und Methoden

Ein wichtiger Bestandteil vieler moderner Computersprachen sind die **Objekte**. OCaml steht für Objektorientiertes Caml, das bedeutet, OCaml lässt auch Objekte zu. Objekte kann man frei definieren und verhalten sich ähnlich wie Module (sodass man in Caml eigentlich auch ohne Objekte auskommen könnte). Objekte sind Datenstrukturen, die man abspeichern oder weitergeben kann. Sie können Variable enthalten und Funktionen, um diese zu verändern. Ein Beispiel hilft vielleicht, das Konzept besser zu verstehen.

```
class konto =
  object
  val mutable x = 0.0
    method stand = x
    method einzahlen y = x <- x +. y
    method auszahlen y = x <- x -. y
end;;
```
(123)

Zunächst nur die Syntax: die Struktur ist `class` <Objektname> `= object` ··· `end`. <Objektname> ist dazu da, um die Art des Objekts zu bezeichnen. Zwischen `object` und `end` erfolgen Definitionen, die entweder die Form `val` oder `method` haben. Falls wir `val x = 0.0` schreiben, können wir `x` nicht verändern. Damit dies möglich wird, müssen wir schreiben `val mutable x = 0.0`. Damit wird `x` zu Beginn der Wert 0 zugewiesen. Mit `method` wird eine Funktion definiert. Nach dem Wort `method` kommt der Name der Funktion, anschließend die Variable, die es benötigt, dann das Gleichheitszeichen. Achtung: da der Name `x` vergeben ist, sollte man die Variablen der Funktion nicht mehr `x` nennen. Man beachte auch, dass `x` auf der rechten Seite auftritt. Die Schreibweise `x <- x +. y` bedeutet, dass der Variable `x` nunmehr der alte Wert von `x` vermehrt um den Wert von `y` zugewiesen wird. (Eine Anmerkung: `+.` ist die Addition von `float` (reelle Zahlen), `-.` die Subtraktion. Die reelle Zahl 0 kann wahlweise `0.0` oder auch `0.`, ohne folgende Nullen, geschrieben werden.)

Wenn wir das obenstehende eingeben, bekommen wir folgende Antwort.

(124)
```
class konto :
  object
    val mutable x : float
    method stand : <unit>
    method einzahlen : float -> <unit>
    method auszahlen : float -> <unit>
end
```

Man beachte den Typ <unit>. Dieser Typ hat kein Objekt, sondern bezeichnet nur eine Aktion. Es dauert ein wenig, bis man verstanden hat, wozu das nützlich ist. Die Methode einzahlen, zum Beispiel, benötigt eine reelle Zahl als Eingabe und dann wird das Objekt verändert. Das heißt, wir bekommen kein Ergebnis, sondern es tut sich einfach nur etwas, nämlich dass die Variable x einen neuen Wert hat.

Nun, da wir die Klasse definiert haben, können wir Objekte dieser Klasse schaffen, und zwar so viele, wie wir wollen.

(125)
```
# let m = new konto;;
val m : konto = <obj>
# m#stand;;
- : float = 0.
```

Die erste Zeile bindet den Identifikator m an eine Objekt der Klasse konto. OCaml wiederholt dies für uns. this. In der dritten Zeile wenden wir die Methode stand auf das Objekt an. Man beachte, dass der Wert von x nicht einfach durch das Tippen von m geholt werden kann. Das hat mehrere Gründe. Der Wichtigste ist, dass in einem Objekt mehrere Größen definiert sein können und wir daher klar sagen müssen, welche von diesen wir haben wollen. Zweitens kann es sein, dass wir in dem Objekt auch Größen haben wollen, die wir nicht nach außen preisgeben wollen. Aus diesen Gründen ist es nicht automatisch gestattet, eine Variable zu lesen. Dazu muss man erst eine Methode definieren!

Wir sehen im Übrigen auch, dass der Befehl m#stand sowohl den Namen des Objekts wie auch die Methode enthält. Das ist wichtig, weil wir zum Beispiel auch noch ein anderes Objekt diesen Typs definieren können, sagen wir mal pi, und dann mit Hilfe pi#stand den Wert der Variable x in diesem Objekt abfragen

können.

(126)
```
# let q = new konto;;
val q : konto = <obj>
# q#auszahlen 5.;;
- : unit = ()
# q#stand;;
- : float = -5.
```

Der Wert von `x` ist jeweils lokal in dem Objekt vorhanden. Jedes Objekt hat seine eigene Variable, und diese werden unabhängig voneinander verändert. Deswegen hat `p#stand` nichts mit `q#stand` zu tun!

10 Buchstaben und Zeichenketten

Nach dieser Vorbereitung in die Benutzung von OCaml wenden wir uns jetzt der Sprachverarbeitung zu. Wir beginnen damit, uns ausführlich mit Buchstaben und Zeichenketten zu beschäftigen. Eine sehr schöne (allerdings auch umfangreiche) Darstellung zum Thema Alphabete und Unicode findet man in [6].

Buchstaben (oder *characters*) sind Einzelsymbole. Zeichenketten werden aus Buchstaben aufgebaut. Bevor wir uns mit Zeichenketten befassen können, müssen wir uns erst einmal mit **Buchstaben** vertraut machen, denn diese sind in Tat komplizierter, als man denkt. Das **Alphabet** ist die Menge der Buchstaben, die uns zur Verfügung steht. Je nach Anwendung kann das Alphabet kleiner oder größer sein. OCaml benutzt standardmäßig die Tabelle ISO Latin 1 (auch bekannt als ISO 8859-1). Man kann also nur Buchstaben aus diesem Zeichensatz verwenden. Das lässt sich auch ändern; dazu gibt es ein spezielles Modul (zum Beispiel Camomile), um vollen Unicode verwenden zu können, allerdings sinnvollerweise nur bei Zeichenketten. Das ist deswegen so, weil Programme ja durch die Welt gehen und praktisch in jeder Tastatur manipulierbar sein müssen. Und am praktischsten ist es, wenn man nur diejenigen Symbole verwendet, die auf jeder Tastatur zu haben sind. Ich habe im Übrigen die Probe gemacht und ein Programm geschrieben, das in einer Datei alle Zeichen ablegt zusammen mit ihrem Code. Diese Datei heißt `iso_latin_1.txt` und kann man einem Editor angeschaut werden. (Gvim macht hier eine sehr gute Figur. Sie müssen schon einen Editor nehmen, das Terminal tut es nicht, weil es einige Zeichen als Steuerzeichen interpretiert.)

Das Wort “Buchstabe” für Character ist ein bisschen unangemessen. Viele der erlaubten Zeichen kann man nämlich auf dem Bildschirm gar nicht sehen, wie etwa <return>. Andere werden für spezielle Zwecke gebraucht, etwa sogenannte **Delimiter**. Diese strukturieren den Text (wie Klammern) oder verpacken ihn zu einem speziellen Typ (Anführungszeichen). Man hat wie überall das Problem, dass man gelegentlich auch ein spezielles Symbol in seinem nichttechnischen Sinne in einem Text verwenden will, etwa die doppelten Anführungszeichen in einer Zeichenkette. Denn eine Zeichenkette ist für OCaml alles, was zwischen zwei doppelten Anführungszeichen steht. Wie kann man also eine Zeichenkette definieren, die ein oder mehrere solcher Anführungszeichen enthält? Die Lösung besteht in einer speziellen Konvention, mit der wir diese Zeichen “zitieren” können. Damit uns OCaml also nicht falsch versteht, gibt es eine Konvention, wie man dieses Zeichen in der direkten Kommunikation vermeiden kann; man nennt

das auch “maskieren”. Diese ist bei OCaml, dass wir einen beliebigen Buchstaben durch die Kombination \ gefolgt von 3 Ziffern ersetzen dürfen. Wer ins Handbuch geht, kann dort Funktionen namens `int_of_char` und `char_of_int` finden. Diese sind invers zueinander. Es gibt 256 Buchstaben, und diesen entsprechen die Zahlen von 0 bis 255.

(127)
```
# int_of_char 'L';;
- : int = 76
# char_of_int 76;;
- : char = 'L'
```

Sehen wir uns nun an, wie man das verwenden kann.

(128)
```
# "\076ehrer";;
- : string = "Lehrer"
```

Was uns OCaml zurückgibt, ist nicht das, was wir eingetippt hatten. Aber das war so beabsichtigt. Denn jetzt wissen wir, wie wir im Grunde genommen jedes Symbol eingeben können, zum Beispiel doppelte Anführungszeichen.

(129)
```
# int_of_char '"';;
- : int = 34
# "\034Lehrer\034";;
- : string = "\"Lehrer\""
```

OCaml teilt uns das Ergebnis mit, indem es allerdings (zur Sicherheit) einen Schrägstrich vor die Anführungszeichen setzt. Dies soll uns sagen, dass das folgende Vorkommen auch tatsächlich ein Anführungszeichen ist. Diese Methode steht uns im Übrigen auch frei:

(130)
```
# "\"Lehrer\"";;
- : string = "\"Lehrer\""
```

Allerdings ist diese Methode der Eingabe nicht so flexibel (das entsprechende Symbol muss zum Beispiel auf der Tastatur verfügbar sein).

OCaml hat ein Standardmodul namens `Char`, das ein paar Funktionen enthält. Die Funktion `code` (also dann `Char.code`, solange wir den Modul nicht geöffnet haben) ist identisch mit `int_of_string`. Die Funktion `escaped` gibt uns eine Sequenz (= Zeichenkette), die den Buchstaben ersetzt.

Außer OCaml selbst stellt uns der Computer auch Hürden auf. Solange wir englische Texte mit englischer Tastatur, französische Text mit französischer Tastatur usf. schreiben, ist die Welt in Ordnung. Aber sobald wir französische Texte auf einer englischen (oder deutschen) Tastatur schreiben wollen, wird die Lage kompliziert. Auf das Problem gibt es im Wesentlichen zwei Antworten: die eine ist graphisch. Wir klicken auf der Befehlsfläche, bis wir ein Fenster bekommen mit vielen Kästchen, von denen jedes ein Symbol enthält, welches wir anklicken können, um es in den Text eingesetzt zu bekommen. Die andere Lösung ist tastaturbasiert. Es gibt eine besondere Tastenfolge, die dieses Symbol eindeutig repräsentiert. (In gvim kann man inzwischen jedes Unicode Zeichen bekommen, indem man <Strg>V gefolgt von dem numerischen Code eingibt oder <Strg>K gefolgt von einem zwei-Tasten-Code. Wenn Sie die Optionen sehen wollen, sagen Sie einfach `:digraphs` or simply `:dig` im Kommando-Modus, und dann sagt gvim, was es anbietet.)

Die Umstellung auf Unicode schreitet langsam voran. Bis vor Kurzem war es noch ein Problem, Sonderzeichen in HTML zu bekommen. Den Buchstaben /ä/ bekam man nur, indem man die Sequenz /`&auml;`/ in den Quelltext schrieb. (Die Schrägstriche /·/ sind meine eigene Art, Zeichen und Zeichenketten zu zitieren, damit sie als Zitat sichtbar sind.) Inzwischen ist es vielfach möglich, /ä/ direkt einzugeben. Trotzdem gibt es auch bei HTML Grenzen. Da das Kaufmanns-Und /&/ und das Semikolon diese Sequenzen einrahmen, darf man jetzt natürlich nicht zulassen, dass der Nutzer eines dieser Symbole tippt, wenn er es auf dem Bildschirm haben will. Es muss also umgangen werden. Das Semikolon ist dabei unkritisch; lediglich das Kaufmanns-Und muss man stets durch /`&`/ eingeben, wenn es denn erscheinen soll. Ähnliche Probleme werden wir bei regulären Ausdrücken kennenlernen. Dort wird man der Sache dann durch eine wahre Orgie von Schrägstrichen Herr.

An dieser Stelle soll noch erwähnt werden, dass vom Standpunkt der Textverarbeitung Alphabete relativ komplexe Gebilde sind. Es gibt zunächst einmal mehrere Sorten Zeichen. Wer sich einen Überblick verschaffen will, kann ja mal unter Unix /`man regex`/ aufrufen. Dazu gehören Buchstaben des Alphabets, Ziffern, Interpunktionszeichen, Trennzeichen, graphische Symbole. Zusätzlich muss man beachten, dass es Alternatischreibungen gibt (heutzutage meist verursacht durch fehlende Zeichen in Tastaturen). Im Deutschen ist es üblich, /ß/ durch /ss/, /ä/ durch /ae/ zu ersetzen. Wer mit Google sucht, wird feststellen, dass man auch dann Treffer bekommt, wenn man eine solche Ersetzung vornimmt (etwa

/Uebersee/ anstelle von /Übersee/). Im Deutschen ist die Klein- und Großschreibung wichtig, auf sie wird nicht verzichtet. Dennoch gibt es zahlreiche Umstände, wo der Unterschied nicht gemacht wird. Aus praktischen Erwägungen ignorieren Suchmaschinen den Unterschied. Die Ordnung in Alphabeten ist auch verschieden von Sprache zu Sprache. Im Deutschen steht /ä/ vor /ae/ aber hinter /ad/. Im Finnischen dagegen, welches fast den gleichen Buchstabenvorrat hat wie das Deutsche, kommen /ä/, /ö/ hinter /z/ (/ü/ gibt es nicht, dafür schreibt man schlicht /y/, und dies steht vor /z/). Manche **Digraphen** (Doppelbuchstaben) werden auch als ein Symbol behandelt, wie etwa /cs/ im Ungarischen (entspricht lautlich [tʃ]), das nach /c/ und vor /d/ sortiert wird. Deswegen steht /cukor/ 'Zucker' vor /csirke/ 'Huhn' im Wörterbuch, obwohl die Reihenfolge der Buchstaben ansonsten wie im Deutschen ist. Dazu muss man wissen, dass die Kombination /cs/ in diesem Fall als ein Einzelzeichen behandelt wird. Die ungarischen Wörterbücher haben denn auch nach der Sektion /c/ eine Sektion /cs/, auf der die Sektion /d/ folgt. Ganz anders im Deutschen, wo /sch/ ([ʃ]) zwar lautlich ein einziges Phonem ist aber im Wörterbuch als Kombination aus drei Buchstaben aufgefasst wird (also eigentlich Trigraph heißen müsste). Dies zeigt sich unter darin, dass man diese Buchstabenkette nicht trennen darf. Digraphen sind recht unberechenbar. Zunächst einmal ist nicht jede Kombination, die wie ein Digraph aussieht, auch wirklich einer. Im Deutschen spricht man das /sch/ in /Mäuschen/ nicht [ʃ] und eine Trennung in /Mäus/ und /chen/ ist gestattet. Analog ist im Ungarischen nicht jede Kombination aus /c/ und /s/ schon der Digraph [tʃ], nämlich wenn dazwischen eine Wortgrenze liegt. Außerdem wird die Verdopplung von /cs/ nicht /cscs/ geschrieben sondern /ccs/. Dies ist konsequenterweise ein Trigraph, kein Digraph.

OCaml hat auch ein Modul für Zeichenketten genannt `String`. Darin findet man eine Reihe nützlicher Funktionen. Außerdem werden uns noch `Buffer` und `Stream` interessieren. Doch davon später. Mathematisch gesehen ist eine Zeichenkette über einem Alphabet A eine Funktion von einem endlichen Anfangsstück der natürlichen Zahlen, also der Menge $\{0, 1, 2, \ldots, n-1\}$ in A. So ist zum Beispiel die Kette `"Katze"` die Funktion f von $\{0, 1, 2, 3, 4\}$ in das Alphabet derart, dass $f(0) =$ `K`, $f(1) =$ `a`, $f(2) =$ `t`, $f(3) =$ `z` und $f(4) =$ `e`. Implementiert wird eine Zeichenkette als Vektor (English `array`) über dem Alphabet. Schauen wir uns das

bei OCaml an.

(131)
```
# let u = "Katze" ;;
val u : string = "Katze"
# u.[0];;
- : char = 'K'
# u.[4];;
- : char = 'e'
# u.[5];;
Exception: Invalid_argument "index out of bounds".
```

Ganz wichtig: OCaml fängt bei 0 mit Zählen an. Der erste Buchstabe ist `u.[0]`, der zweite `u.[1]` und so weiter. `u.[5]` ist in diesem Fall aber nicht zugelassen; die Zeichenkette hat die Länge 5, deswegen ist bei 4 Schluss. Man beachte, dass es eine Zeichenkette der Länge 0 gibt. Das ist die leere Zeichenkette. Diese können wir mit "" bezeichnen. Ohne die Anführungszeichen wäre für uns leider nichts zu sehen und OCaml ginge das nicht anders.

Definition 10.1 (Zeichenkette) *Es sei A eine beliebige Menge, das sogenannte **Alphabet**. Eine **Zeichenkette** über A der Länge n ist eine Funktion*

$$\vec{x} : \{0, 1, 2, \cdots, n-1\} \to A$$

*Es bezeichnet $|\vec{u}|$ die Länge von $\vec{x}$. Wir schreiben auch $\vec{x}(i)$ für den Buchstaben an der Stelle i. Ist $n = 0$, so ist $\vec{x} : \varnothing \to A$ die sogenannte **leere Zeichenkette** (oder auch das **leere Wort**).*

Definition 10.2 (Verkettung) *Seien $\vec{x}$ und $\vec{y}$ Zeichenketten der Länge m und n. Dann ist $\vec{x}^\frown\vec{y}$ diejenige Zeichenkette der Länge $m + n$, für die Folgendes gilt.*

(132)
$$(\vec{x}^\frown\vec{y})(j) = \begin{cases} \vec{x}(j) & \text{falls } j < m, \\ \vec{y}(j-m) & \text{sonst.} \end{cases}$$

*Wir nennen $\vec{x}^\frown\vec{y}$ die **Verkettung** von $\vec{x}$ mit $\vec{y}$. $\vec{x}$ ist ein **Präfix** von $\vec{y}$, falls es ein $\vec{w}$ gibt mit $\vec{y} = \vec{x}^\frown\vec{w}$; und es ist ein **Postfix**, wenn es ein $\vec{w}$ gibt mit $\vec{y} = \vec{w}^\frown\vec{x}$. $\vec{x}$ ist **Teilwort**, falls es $\vec{w}$ und $\vec{v}$ gibt mit $\vec{y} = \vec{w}^\frown\vec{x}^\frown\vec{v}$.*

OCaml hat eine Funktion `String.length` welche die Länge einer Zeichenkette ausgibt. So gibt zum Beispiel `String.length "Katze"` den Wert 5. Wir haben

schon gesehen, dass nach unserer Zählkonvention die Zeichenkette auf den Zahlen von 0 bis 4 definiert ist. Das Folgende zeigt einen beliebten Fehler. Man will den letzten Buchstaben und gibt daher als Argument `String.length` <Zeichenkette> ein.

(133)
```
# "Katze".[2];;
- : char = 't'
# "Katze".[String.length "Katze"];;
Exception: Invalid_argument "index out of bounds".
```

Man soll immer daran denken, dass das letzte Symbol durch `String.length` (<Zeichenkette> `-1`) gegeben ist.

In OCaml ist `^` ein Infixoperator, der die Konkatenation bezeichnet. Also ist `"Raub"^ "Katze"` gleich `"RaubKatze"`. An dieser Stelle sei darauf hingewiesen, dass OCaml aus Gründen der Benutzerfreundlichkeit auch Infixoperatoren zulässt. Ansonsten wäre nämlich vieles unlesbar. Das hat aber auch seinen Preis. Denn wir können einen Infixoperator nicht einfach als Funktionsobjekt verwenden. Das liegt diesmal nicht an seinem Typ (der ist richtig), sondern an seiner Syntax. Es gibt folgende Konvention.

> Ist <Operator> ein Infixoperator, so ist (<Operator>) der zugehörige Postfixoperator. Der Ausdruck (<Operator>) <Arg1> <Arg2> ist gleich dem Ausdruck <Arg1> <Operator> <Arg2>.

Doch nun zu Zeichenketten im allgemeinen. Wir verwenden folgende Abkürzung. $\vec{x}^n$ bezeichnet die Konkatenation von n Kopien von $\vec{x}$. Eine rekursive Definition ist wie folgt.

(134) $$\vec{x}^0 = \varepsilon$$

(135) $$\vec{x}^{n+1} = \vec{x}^n \hat{} \vec{x}$$

Zum Beispiel ist $(\mathtt{vux})^3 = \mathtt{vuxvuxvux}$. Die Klammern sind notwendig, da der Exponent stärker bindet; deswegen ist $\mathtt{vux} = \mathtt{vuxxx}$.

Eine **Sprache** über A ist eine Menge von Zeichenketten über A. Dazu gehören solche Objekte wie die Menge aller Zeichenketten über A (mit A^* bezeichnet), die leere Menge oder die Menge $\{\varepsilon\}$. Abgesehen davon sind auch jede Menge

sinnvoller Objekte dabei. Auf Sprachen definieren wir folgende Operationen.

(136a) $L \cdot M := \{\vec{x}\vec{y} : \vec{x} \in L, \vec{y} \in M\}$

(136b) $L/M := \{\vec{x} : \text{für alle } \vec{y} \in M \text{ ist } \vec{x}\vec{y} \in L\}$

(136c) $M\backslash L := \{\vec{x} : \text{für alle } \vec{y} \in M \text{ ist } \vec{y}\vec{x} \in L\}$

$L \cdot M$ ist die Sprache aller Kombinationen, die man aus Ketten von L mit Ketten von M bekommen kann. Sei zum Beispiel

(137) $L := \{\texttt{Alfred␣}, \texttt{Mein Nachbar␣}, \dots\}$

(mit ␣ das Leerzeichen) die Menge der Nominalphrasen und

(138) $M := \{\texttt{singt}, \texttt{rettet␣die␣Königin}, \dots\}$

die Menge der Verbalphrasen. Dann ist $L \cdot M$ die folgende Menge.

(139)
$$\begin{aligned} L \cdot M = \{\ & \texttt{Alfred␣singt}, \texttt{Alfred␣rettet␣die␣ Königin}, \\ & \texttt{Mein␣Nachbar␣singt}, \\ & \texttt{Mein␣Nachbar␣rettet␣die␣Königin}, \dots\} \end{aligned}$$

Dies entspricht der Idee der Regel S → NP VP: welches $\vec{x}$ immer eine NP ist und welches $\vec{y}$ eine VP, die Verkettung $\vec{x}^\frown\vec{y}$ ist ein S. (Da es noch mehr Regeln geben kann, muss hier nicht Gleichheit herrschen.)

Die Sprache L/M bezeichnet die Menge der Zeichenketten, die zu L-Ketten werden, wenn man eine beliebige M-Kette anhängt. Es gilt also sowohl

(140) $L/M \cdot M \subseteq L$

als auch

(141) $(L \cdot M)/M = L$

Sei zum Beispiel $L = \{\texttt{Stühle}, \texttt{Pferde}, \texttt{Mäuse}, \dots\}$ und $M = \{\texttt{e}\}$. Dann ist

(142) $L/M = \{\texttt{Stühl}, \texttt{Pferd}, \texttt{Mäus}, \dots\}$

Falls L kein Wort enthält, das mit e aufhört, dann ist L/M leer.

(143) $\{\texttt{Milch}, \texttt{Väter}\}/M = \varnothing$

Zum Schluss komme ich noch einmal auf die Module `Buffer` und `Stream` zurück. **Buffers** sind eine schnellere Implementierung von `string`. Die Verkettung von Buffern ist wesentlich schneller als die von Zeichenketten, wie man im Handbuch nachlesen kann. Um einen Buffer zu bekommen, sagt man zum Beispiel

(144)
```
# let b = Buffer.create 10 ;;
val b : Buffer.t = <abstr>
# Buffer.add_string b "Panther";;
- : unit = ()
# Buffer.contents b ;;
- val : string = "Panther"
```

Ein **Stream** ist dazu da, sehr schnell durch eine Zeichenkette oder Datei zu gehen. Im Zusammenhang mit Suchen oder der Manipulation von großen Textmengen ist das Hantieren mit Zeichenketten extrem langsam. Hier bieten sich Streams an. Ein Stream ist so etwas wie eine Warteschlange. Die Objekte werden aufgereiht, und wir können sie eins nach dem anderen anschauen (mit `next`). Haben wir sie angeschaut, werden sie von der Warteschlange entfernt. Es gibt keine Möglichkeit, ein Objekt wieder in die Schlange zu tun. Es gibt aber die Möglichkeit, das Ende der Warteschlange zu sehen, ohne die Objekte zu entfernen (siehe dazu die Funktionen `peek` oder `npeek`). In der Informatim sind Parser sehr beliebt, die mit begrenztem Sichtfenster (sogenanntes **lookahead**) arbeiten. Sie treffen eine unwiderrufliche Entscheidung für eine Aktion anhand der Symbole, die sie im Sichtfenster sehen. OCaml tut dies auch. Beim ersten Fehler steigt es aus und beschwert sich dann beim Nutzer.

11 Reguläre Ausdrücke

Reguläre Ausdrücke sind Ausdrücke, die eine reguläre Sprache bezeichnen. In den nächsten Kapiteln werden wir diese Sprachen ausführlicher studieren und auch noch andere Arten finden, wie man reguläre Sprachen beschreiben kann. Eine sehr schöne Darstellung von regulären Ausdrücken findet sich in [3].

Ich beginne zunächst noch mit ein paar allgemeinen Definitionen. Zunächst sei für eine Sprache P die Exponentiation P^n rekursiv definiert durch

$$P^n := \begin{cases} \{\varepsilon\} & \text{falls } n = 0 \\ P \cdot P^{n-1} & \text{falls } n > 0 \end{cases} \tag{145}$$

Dann ist, wie man leicht sieht, $P^1 = \{\varepsilon\} \cdot P = P$.

Definition 11.1 *Es sei $P \subseteq A^*$ eine Sprache über A. Dann ist P^* definiert als*

$$P^* := \bigcup_{n \in \mathbb{N}} P^n = \{\varepsilon\} \cup P \cup P \cdot P \cup P \cdot P \cdot P \cup \cdots \tag{146}$$

* *heißt der* ***Kleene-Stern****.*

Jetzt wird auch klar, warum die Menge der Zeichenketten über A mit A^* bezeichnet wurde: dies ist einfach ein Spezialfall der obigen Notation.

Ein **regulärer Ausdruck** wird aus den Buchstaben des Alphabets mit Hilfe der Symbole 0, ε, $\cdot$, $\cup$, und * gebildet. Die Buchstaben des Alphabets sowie die Symbole 0 und ε sind Konstanten. $\cdot$ und $\cup$ sind binäre Operatoren, * ist einstellig. Einem regulären Ausdruck ordnen wir eine Sprache wie folgt zu.

$$\begin{aligned} L(0) &:= \varnothing \\ L(\varepsilon) &:= \{\varepsilon\} \\ L(s \cdot t) &:= L(s) \cdot L(t) \\ L(s \cup t) &:= L(s) \cup L(t) \\ L(s^*) &:= L(s)^* \end{aligned} \tag{147}$$

$\cdot$ wird oft weggelassen. st ist also dasselbe wie $s \cdot t$. Ebenso ist /`Weg`/ dasselbe wie `W` $\cdot$ `e` $\cdot$ `g`.

Definition 11.2 *Eine Sprache S heißt **regulär**, falls sie die Form $S = L(t)$ hat für einen regulären Term t.*

Daraus lässt sich leicht sehen, dass jede Menge bestehend aus einer einzigen Zeichenkette regulär ist. Und daraus lässt sich wiederum ableiten, dass jede endliche Menge regulär ist. Aber das ist nicht alles.

Ich weise darauf hin, dass $L(t)$ eine Sprache ist, das heißt, eine Menge von Zeichenketten, während t ein Term ist. Dabei sagen wir, $\vec{x}$ **fällt unter den Term** t, wenn $\vec{x} \in L(t)$. Im Umgangsgebrauch wird nicht immer sorgfältig unterschieden. So bezeichnet der Buchstabe /a/ sowohl die Zeichenkette /a/ wie auch den Term /a/. Dessen Sprache ist {a} ist, also verschieden von /a/ selbst. Bei der Schreibung von Termen benutzen wir wie üblich Klammern. Wir haben nun

$$\begin{array}{ll} L(\mathtt{a} \cdot (\mathtt{b} \cup \mathtt{c})) & = \{\mathtt{ab}, \mathtt{ac}\} \\ L((\mathtt{cb})^*\mathtt{a}) & = \{\mathtt{a}, \mathtt{cba}, \mathtt{cbcba}, \ldots\} \end{array} \tag{148}$$

Die folgenden Abkürzungen sind gebräuchlich.

$$\begin{array}{ll} s? & := \varepsilon \cup s \\ s^+ & := s \cdot s^* \\ s^n & := s \cdot s \cdot \cdots \cdot s \quad (n\text{-mal}) \end{array} \tag{149}$$

Wir schreiben $s \subseteq t$, wenn $L(s) \subseteq L(t)$. Dies ist gleichbedeutend mit $L(s \cup t) = L(t)$. Wir schreiben auch $s = t$, falls $L(s) = L(t)$.

Reguläre Ausdrücke sind aus dem Computer nicht wegzudenken. Wenn wir zum Beispiel nach einer Zeichenkette suchen, etwa /radfahren/, so genügt es nicht, nur diese Zeichenkette anzuschauen. Am Satzanfang wird daraus nämlich /Radfahren/. Also müssen wir mindestens nach zwei Zeichenketten suchen. Oder wir suchen eine Datei, deren Name mit /.html/ aufhört, oder eine Kombination von zwei Wörtern, die durch eine beliebige Zahl von Leerzeichen (oder Zeilenumbruch!) voneinander getrennt sein können. Was wir dann tun (oder der Computer für uns tut) ist, dass wir einen regulären Ausdruck formulieren, der alle diese Fälle genau umfasst, und diesen dann der Suchmaschine übergibt. (Wie diese aussieht, werden wir später sehen.) Das Unix Kommando `egrep` ist dafür gut.

Wir haben schon gesehen, dass es Terme gibt, die zwar verschieden sind, aber dieselbe Sprache definieren. Dazu können wir einige allgemeine Gesetze formu-

lieren, die einfach aufgrund der Definitionen gelten.

$$
\begin{array}{rl@{\qquad}rl}
s \cdot (t \cdot u) & = (s \cdot t) \cdot u & & \\
s \cdot \varepsilon & = s & \varepsilon \cdot s & = s \\
s \cdot 0 & = 0 & 0 \cdot s & = 0 \\
s \cup (t \cup u) & = (s \cup t) \cup u & & \\
s \cup t & = t \cup s & & \\
s \cup s & = s & & \\
s \cup 0 & = s & 0 \cup s & = s \\
s \cdot (t \cup u) & = (s \cdot t) \cup (s \cdot u) & & \\
(s \cup t) \cdot u & = (s \cdot u) \cup (t \cdot u) & & \\
s^* & = \varepsilon \cup s \cdot s^* & &
\end{array}
\tag{150}
$$

Diese Gleichungen sind einfach nachzuprüfen.

Für unseren Gebrauch ist die folgende Tatsache von zentraler Bedeutung. Es sei folgende Gleichung gegeben ($\cdot$ bindet stärker als $\cup$).

$$X = \mathsf{a} \cup \mathsf{b} \cdot X \tag{151}$$

Hier bezeichnet X eine Sprache. Wir fragen: welches ist die kleinste Lösung dieser Gleichung? Laut Gleichung ist dies eine Menge M, welche zumindest einmal /a/ enthält. Zusätzlich enthält sie mit jeder Zeichenkette $\vec{v}$ auch noch die Zeichenkette $\mathsf{b}\hat{}\vec{v}$. Da nun /a/ in M ist, ist folglich auch /ba/ in M und deswegen /bba/, /bbba/, und so weiter. Wie man leicht sieht, ist die Sprache, die hier entsteht, durch den folgenden Term beschrieben.

$$X = \mathsf{b}^*\mathsf{a} = \{\mathsf{a}, \mathsf{ba}, \mathsf{bba}, \mathsf{bbba}, \cdots\} \tag{152}$$

(Wir unterscheiden ab jetzt auch nicht mehr zwischen Sprachen und Termen, die diese Sprachen beschreiben!) Setzen wir nun X derart, dann gilt die Gleichung in der Tat. Denn

$$\mathsf{b}X = \{\mathsf{ba}, \mathsf{bba}, \mathsf{bbba}, \cdots\} \tag{153}$$

und so ist $\mathsf{a} \cup \mathsf{b}X = X$.

Satz 11.3 *Es seien s und t reguläre Ausdrücke. Dann hat die Gleichung $X = t \cup s \cdot X$ eine eindeutig bestimmte kleinste Lösung. Diese ist $s^* \cdot t$.*

Beweis. Ich definiere folgende Funktion f:

$$f(L) := t \cup s \cdot L \tag{154}$$

Die Gleichung formuliert sich dann ganz einfach wie folgt:

$$X = f(X) \tag{155}$$

Man sagt, X sei **Fixpunkt** der Funktion f. Es gilt nun im Allgemeinen nicht $f(L) = L$, insofern ist nicht jede Sprache ein Fixpunkt. Den kleinsten Fixpunkt bekommen wir wie folgt. Wir definieren

$$X_0 := \varnothing \qquad X_{n+1} := f(X_n) \tag{156}$$

Es gilt dann $X_1 = f(X_0) = t \cup s \cdot \varnothing = t$ und deswegen ist $X_0 \subseteq X_1$. Die Funktion f ist monoton, das heißt, ist $L \subseteq M$, so ist $f(L) \subseteq f(M)$. Also folgt aus $X_0 \subseteq X_1$ unmittelbar $X_1 \subseteq X_2$, da $X_1 = f(X_0)$ und $X_2 = f(X_1)$. Daraus folgt:

$$X_0 \subseteq X_1 \subseteq X_2 \subseteq \cdots \tag{157}$$

Wir setzen also

$$X_* := \bigcup_{n \in \mathbb{N}} X_n \tag{158}$$

Dann ist

$$\begin{aligned} f(X_*) &= f\left(\textstyle\bigcup_{n\in\mathbb{N}} X_n\right) \\ &= \textstyle\bigcup_{n\in\mathbb{N}} f(X_n) \\ &= \textstyle\bigcup_{n\in\mathbb{N}} X_{n+1} \\ &= X_* \end{aligned} \tag{159}$$

Der letzte Schritt ist gültig, weil $X_0 \subseteq X_1$, insofern die Menge X_0 auch schon in der großen Vereinigung vorhanden ist. Damit müssen wir jetzt nur noch wissen, wie X_* aussieht. Dazu bestimmen wir X_n. Ich behaupte: $X_n = \bigcup_{i<n} s^i t$ für $n > 0$. Dies zeigt man induktiv. Ist $n = 1$, so lautet die Behauptung $X_1 = s^0 t = t$, was wir oben schon gesehen haben. Jetzt sei $X_n = \bigcup_{i<n} s^i t$ schon gezeigt. Dann ist

$$\begin{aligned} X_{n+1} &= f(X_n) \\ &= t \cup s\left(\textstyle\bigcup_{i<n} s^i\right) t \\ &= t \cup \textstyle\bigcup_{0<i<n+1} s^i t \\ &= \left(\varepsilon \cup \textstyle\bigcup_{i<n+1} s^i\right) t \\ &= \left(\textstyle\bigcup_{i<n+1} s^i\right) t \end{aligned} \tag{160}$$

Damit ist die Behauptung fast gezeigt. Fehlt nur noch die Bemerkung, dass X_* die Vereinigung aller dieser Mengen ist, also $(\bigcup_{n\in\mathbb{N}} s^n)\,t = s^*t$, nach Definition des Sterns. ⊣

Dabei gibt es ein paar Sonderfälle, die ich gleich anmerke. Sie sind alle in Satz 11.3 enthalten. Der erste ist $t = 0$. Dann ist $X = 0$ offenkundig eine Lösung. In der Tat ist dann auch $s^*t = 0$. Ein anderer Sonderfall ist $\{\varepsilon\} = L(s)$, Dann hat man die Gleichung $X = t \cup \varepsilon X$. Diese hat t als kleinste Lösung. In der Tat ist $\varepsilon^*t = t$.

Wenn man nun von einigen dieser Sonderfälle absieht, gibt es sogar meist eine einzige Lösung. Dann kann also der Zusatz "kleinste" entfallen.

Satz 11.4 *Es seien s und t reguläre Ausdrücke und $\varepsilon \notin L(s)$. Dann hat die Gleichung $X = t \cup s \cdot X$ eine eindeutig bestimmte Lösung. Diese ist $s^* \cdot t$.*

Beweis. Sei M eine zweite Lösung, also $M = t \cup sM$ und $M \neq L(s^*t)$. Dann ist $L(s^*t) \subsetneq M$. Es gibt also ein $\vec{x} \notin L(s^*t)$. Wir wählen $\vec{x}$ von minimaler Länge. Dann ist $\vec{x} \notin L(t)$, und so muss $\vec{x} \in L(s)M$ sein. Also hat $\vec{x}$ die Form $\vec{x} = \vec{u}\vec{x}_1$ wo $\vec{u} \in L(s)$ und $\vec{x}_1 \in M$. Da jetzt $\varepsilon \notin L(s)$, so ist $\vec{u} \neq \varepsilon$, und $\vec{x}_1$ hat kleinere Länge als $\vec{x}$. Es ist aber $\vec{x}_1 \notin L(s^*t)$, da sonst $\vec{x} = \vec{u}\vec{x}_1 \in L(s^*t)$. Wir haben nun einen Widerspruch. M kann somit nicht existieren. ⊣

Ich mache darauf aufmerksam, dass dieser Satz ganz allgemein gilt.

Satz 11.5 *Es seien P und Q Sprachen. Dann besitzt die Gleichung $X = Q \cup P \cdot X$ eine eindeutig bestimmte kleinste Lösung. Diese ist $P^* \cdot Q$. Ist $\varepsilon \notin P$ so ist die Lösung sogar eindeutig.*

$s \cdot t$ ist die einzige Lösung für X des Gleichungssystems aus $Y = t$ und $X = s \cdot Y$; $s \cup t$ ist die einzige Lösung für X des Gleichungsystems bestehend aus $X = Y \cup Z$, $Y = s$, $Z = t$. Mit diesen eher banalen Umformungen können wir eine reguläre Sprache als kleinste Lösung eines bestimmten Gleichungssystems darstellen.

Es seien X_i, $i < n$, Variable für Sprachen. Eine **reguläre Gleichung** hat die Form

$$X_i = s_0 \cdot X_0 \cup s_1 \cdot X_1 \ldots \cup s_{n-1} \cdot X_{n-1} \tag{161}$$

(Falls $s_i = 0$, so kann der Term $s_i \cdot X_i$ weggelassen werden.) Eine solche Gleichung ist **einfach**, wenn für alle $i < n$, entweder $s_i = 0$ oder $s_i = a$ für einen Buchstaben a. Nehmen wir nun an, wir haben eine Gleichung $X = t$, wo t ein regulärer Ausdruck ist. Dann treten mehrere Fälle auf; $t = s \cdot u$, $t = s \cup u$, $t = s^*$, $t = a$, $t = 0$ oder $t = \varepsilon$. In den letzten drei Fällen ist die Gleichung einfach. In den anderen Fällen habe ich eine Methode genannt, wie die Gleichung in ein bis drei Gleichungen umgeformt werden kann, die t nicht mehr enthalten, nur noch s bzw. u. Dies ermöglicht also, eine reguläre Sprache als die kleinste Lösung eines Systems aus einfachen Gleichungen zu bekommen. Genauso kann man ein reguläres Gleichungssystem in ein einfaches reguläres System umformen.

Satz 11.6 *Es sei E ein System von regulären Gleichungen in den Variablen X_i, $i < n$, so dass X_i genau einmal auf der linken Seite einer Gleichung ist. Dann existieren reguläre Terme s_i, $i < n$, derart, dass die kleinste Lösung von E gerade $X_i = L(s_i)$, $i < n$, ist.*

Dieses Theorem verlangt nicht, dass die Gleichungen einfach sind.

Der Beweis ist eine Induktion über n. Wir nehmen an, die Behauptung gilt für jedes System aus höchstens $n - 1$ Variablen. Nehmen wir nun an, die erste Gleichung hat die Form

$$X_0 = s_0 \cdot X_0 \cup s_1 \cdot X_1 \ldots \cup s_{n-1} \cdot X_{n-1} \tag{162}$$

Zwei Fälle sind möglich. Fall (1). $s_0 = 0$. Dann sagt uns die Gleichung, was X_0 ist, sofern wir die X_i, $0 < i < n$, haben. Denn dann hat (162) die Form

$$X_0 = s_1 \cdot X_1 \cup s_1 \cdots X_1 \cup \ldots \cup s_{n-1} \cdot X_{n-1} \tag{163}$$

und dies ist eine explizite Darstellung von X_0. Wir ersetzen in den anderen Gleichungen die Vorkommen von X_0 durch die rechte Seite von (163) und formen sie um. Dann erhalten wir ein Gleichungssystem, das X_0 nicht mehr enthält (mit Ausnahme der Gleichung (163)). Dies lösen wir. Nach Induktionsannahme sind die Lösungen regulär, und durch Terme t_i, $0 < i < n$, gegeben. Dann ist

$$X_0 = s_1 \cdot t_1 \cup s_2 \cdot t_2 \cup \ldots \cup s_{n-1} \cdot t_{n-1} \tag{164}$$

Dies ist auch regulär. Fall (2). $s_0 \neq 0$. Dann benutzen wir Theorem 11.3 und lösen die Gleichung:

$$\begin{aligned} X_0 &= s_0^*(s_1 \cdot X_1 \ldots \cup s_{n-1} \cdot X_{n-1}) \\ &= s_0^* s_1 \cdot X_1 \cup s_0^* s_2 \cdot X_2 \cup \ldots \cup s_0^* s_{n-1} \cdot X_{n-1} \end{aligned}$$

Nun gehen wir wie in Fall (1) vor.

Am Besten versteht man dies mit einem Beispiel. Tabelle 1 zeigt ein Gleichungssystem mit drei Gleichungen für die Variable X_0, X_1 und X_2. (Das ist das System (I).) Wir wollen die kleinste Lösung finden. Zunächst lösen wir die zweite Gleichung mit Hilfe von Satz 11.3 für X_1 und bekommen (II). Die Vereinfachung ist, dass X_1 durch X_0 direkt definiert ist. Die Rekursion wurde eliminiert. Als nächstes setzen wir die Lösung für X_1 in die anderen Gleichungen ein. Das ergibt (III). Anschließend setzen wir in die erste Gleichung die Lösung für X_2 ein. Das ist (IV). Wiederum benutzen wir Satz 11.3 und bekommen (V). Jetzt ist X_0 explizit gegeben. Dies setzen wir ein und bekommen explizite Lösungen für X_1 und X_2 (VI).

Tabelle 1: Lösung eines Gleichungssystems

(I)	$X_0 =$	ε			$\cup \mathsf{d}X_2$
	$X_1 =$		$\mathsf{b}X_0$	$\cup \mathsf{d}X_1$	
	$X_2 =$			$\mathsf{c}X_1$	
(II)	$X_0 =$	ε			$\cup \mathsf{d}X_2$
	$X_1 =$		$\mathsf{d}^*\mathsf{b}X_0$		
	$X_2 =$			$\mathsf{c}X_1$	
(III)	$X_0 =$	ε			$\cup \mathsf{d}X_2$
	$X_1 =$		$\mathsf{d}^*\mathsf{b}X_0$		
	$X_2 =$		$\mathsf{cd}^*\mathsf{b}X_0$		
(IV)	$X_0 =$	ε			$\cup \mathsf{dcd}^*\mathsf{b}X_0$
	$X_1 =$				$\mathsf{d}^*\mathsf{b}X_0$
	$X_2 =$				$\mathsf{cd}^*\mathsf{b}X_0$
(V)	$X_0 =$	$(\mathsf{dcd}^*\mathsf{b})^*$			
	$X_1 =$		$\mathsf{d}^*\mathsf{b}X_0$		
	$X_2 =$		$\mathsf{cd}^*\mathsf{b}X_0$		
(VI)	$X_0 =$				$(\mathsf{dcd}^*\mathsf{b})^*$
	$X_1 =$				$\mathsf{d}^*\mathsf{b}(\mathsf{dcd}^*\mathsf{b})^*$
	$X_2 =$				$\mathsf{cd}^*\mathsf{b}(\mathsf{dcd}^*\mathsf{b})^*$

12 Reguläre Ausdrücke in OCaml

Wir werden uns jetzt etwas genauer ansehen, wie reguläre Ausdrücke in OCaml gehandhabt werden. In anderen Sprachen ist die Behandlung nicht genau gleich aber dennoch recht ähnlich, weil die auftretenden Probleme stets ähnliche Lösungen verlangen. OCaml hat ein Modul namens `Str`. Damit kann man unter anderem reguläre Ausdrücke definieren. Dies Modul ist nicht standardmäßig dabei, deswegen muss man es nach dem Start laden:

(165)
```
# #load "str.cma";;
```

Eine Alternative ist, genau die blaue Zeile in eine Datei namens `.ocamlinit` zu schreiben. Das hat den folgenden Effekt. Sobald man OCaml startet, liest es die Datei `.ocamlinit` ein, als wäre sie Zeile für Zeile eine Eingabe. Damit kann man sich gewisse Anfangsroutinen ersparen.

In dem Modul gibt es einen Typ namens `regexp` und zahlreiche Funktionen dafür. Das Modul ist allerdings etwas anders geartet als die meisten. Die Datei `str.ml` sucht man vergeblich. Sie existiert nicht, weil OCaml das Hantieren mit regulären Ausdrücken nicht selbst macht. Das wird ausgelagert. OCaml übergibt das Problem einfach der im System bereits kodierten Suchmaschine. Dies hat leider den Effekt, dass man nicht garantieren kann, dass sich zwei Programme genau gleich verhalten. Außerdem ist die Syntax der regulären Ausdrücke abhängig von der Plattform. Wie wir gleich sehen werden, ist es denn auch so, dass der reguläre Ausdruck zunächst als Zeichenkette hingeschrieben wird. Dann geht die Funktion `Str.regexp` darüber und analysiert ihn. Das Ergebnis wird der Plattform übergeben, die ihrerseits eine Syntax für reguläre Ausdrücke hat, die man (nach dem Auspacken) eingehalten haben muss. Im Folgenden beschränke ich mich auf das Verhalten von Unix (welches auf Gnu Emacs basiert). Die Abweichungen halten sich sehr in Grenzen.

Wie schon gesagt, wird der reguläre Ausdruck über eine Zeichenkette definiert (wie sie ja auch auf dem Papier erscheint). Damit allerdings aus der Zeichenkette ein regulärer Ausdruck wird, muss man die Funktion `Str.regexp` aufrufen.

(166)
```
# Str.regexp "Tiger";;
- : Str.regexp = <abstr>
```

Man merke, dass ein Wort direkt als regulärer Ausdruck aufgefasst wird (dieses definiert dann auch die Sprache, die dieses Wort enthält). Dies bedeutet, dass die

Buchstaben des Alphabets für sich selbst stehen, wie auch die Ziffern. Wir haben dies oben auch so gehandhabt. Allerdings gibt es ein paar Sonderzeichen. Die Buchstaben `$^.*+?\[ ]` sind vergeben und stehen nicht für sich selbst. Darauf gehe ich unten noch näher ein. Zunächst bemerke ich Folgendes.

- `.` (= Punkt) ist kurz für "jeder Buchstabe außer Zeilenumbruch".
- `*` (postfix) steht für * (beliebige Wiederholung).
- `+` (postfix) steht für $^+$ (beliebige, mindestens einmalige Wiederholung).
- `?` (postfix) steht für ? (höchstens einmal).
- `^` steht für den Zeilenanfang.
- `$` steht für das Zeilenende.
- `\\(` linke (öffnende) Klammer.
- `\\)` rechte (schließende) Klammer.
- `\\|` steht für $\cup$ (Disjunktion).

Das, was wir im laufenden Text als /(a$\cup$b)$^+$d?/ wiedergeben, muss in OCaml jetzt wie folgt geschrieben werden: /\\(a\\|b\\)+d?/. Man beachte, dass OCaml erlaubt, ein Vorkommen eines Zeichens zu maskieren, indem man den Backslash (\) gefolgt von dem 3-stelligen Code des Zeichens (bzw. in einigen Fällen gefolgt von dem Zeichen selbst) eingibt. Dies darf und muss man hier auch verwenden. Die generelle Regel ist diese: die oben genannten Sonderzeichen können nicht als Buchstaben verwendet werden und müssen deswegen maskiert werden. Dafür gibt es auf dem ersten Blick mehrere Möglichkeiten, weil OCaml erlaubt, die Zeichen verschieden zu maskieren. Auf der anderen Seite erlaubt die Plattform ebenfalls, Zeichen zu maskieren. Um aber die Wirkungsweise genau zu verstehen, müssen wir uns vergegenwärtigen, dass es eine Syntax auf der Plattform gibt, die wir über den Umweg der Eingabe durch OCaml erzielen müssen.

So ist zum Beispiel 43 der Code von +. Sollten wir also einmal, sagen wir, den Buchstaben `i` gefolgt von einer echte Folge von Pluszeichen suchen wollen (also /i+/, /i++/, /i+++/ und so weiter), so gibt es offensichtlich mehrere Möglichkeiten. So können wir in OCaml einen regulären Ausdruck aus der folgenden Zeichenkette formen: /i\043+/. (Wir dürfen aber nicht /i\43+/ schreiben,

denn die Null darf nicht weggelassen werden.) Dieser Ausdruck funktioniert aber nicht richtig, er ist äquivalent zu /i+/ (er matcht /i/, was nicht sein soll). Dies liegt daran, dass wir OCaml /+/ maskiert gegeben haben, aber OCaml reicht es an die Plattform als echtes Pluszeichen weiter. Die Plattform erwartet den Ausdruck /i\++/, wo das erste Pluszeichen maskiert ist, das zweite nicht. Diesen Schrägstrich müssen wir unsererseits also eingeben, was wir aber jetzt nicht direkt tun sondern via OCaml. Wir dürfen also auch nicht /i++/ schreiben, weil sonst das erste Plus als Sonderzeichen gelesen wird. Der Ausdruck ist legal, bezeichnet aber eine echte Folge von echten Folgen von /i/, ist also äquivalent mit /i+/. Wie bekommen wir nun OCaml dazu, den Schrägstrich an die Plattform weiterzugeben? Indem wir ihn unsererseits maskieren! Eine Möglichkeit ist diese: /i\092++/. Damit wird OCaml bedeutet: der Schrägstrich ist echt, er dient nicht zur Maskierung des Pluszeichens. Also löst OCaml dies als die Folge /i\++/ auf und gibt dies an die Plattform weiter, wie gewünscht. Eine zweite Möglichkeit ist, anstelle von /\092/ die Folge /\\/ zu setzen. Wir erhalten so /\\++/. Auf diese Weise müssen wir jedes oben genannte Sonderzeichen behandeln. (Ich füge hinzu, dass ich, um Ihnen die Lage zu erklären, Zeichenketten innerhalb von Schrägstrichen schreibe. Dies ist gewissermaßen die dritte Ebene. Diese müssen Sie ihrerseits weglassen, wenn Sie diese Zeichenketten wie angegeben benutzen wollen.)

Man beachte, dass die Klammern wie auch der senkrechte Strich keine Sonderzeichen sind. Auf der anderen Seite brauchen wir die Klammern, und Terme zu strukturieren. Deswegen müssen wir die Klammern nicht dann maskieren, wenn wir sie als Klammern brauchen sondern wenn wir daraus Termstrukturierungssymbole machen wollen. Unix benötigt dazu einen Schrägstrich davor (das ist der generelle Maskierer). Da OCaml nun seinerseits den Schrägstrich zum Maskieren benutzt, müssen wir OCaml sagen, dass der Schrägstrich, den wir setzen, als solcher weitergegeben werden soll. Deswegen müssen wir wie oben gezeigt zwei Schrägstriche setzen (oder /\092/). OCaml übersetzt die zwei Schrägstriche als einen, der dann an das System weitergegeben wird. Hätten wir nur /\(/ hingeschrieben, würde OCaml versuchen, das dadurch bezeichnete Symbol herauszusuchen, welches aber nicht existiert. OCaml meckert daher nur an dem Ausdruck herum. Wenn wir dagegen die Sequenz /\\(/ hinschreiben, so liest OCaml die Sequenz als “maskierter Schrägstrich” + “Klammer” und gibt sie an die Plattform als /\(/ weiter.

Außer den oben erwähnten Konstrukten gibt es noch viel mehr, weil das Schreiben von regulären Ausdrücken ansonsten eine sehr mühselige Arbeit wird. Es

kommt zum Beispiel häufiger vor, dass man vorgeben will, dass ein Zeichen aus einer ganz bestimmten Liste von Zeichen stammen soll. Dazu kann man die eckigen Klammern verwenden. /[/ und /]/ fassen eine Ansammlung von Buchstaben zusammen. Der Ausdruck /[abx09]/ steht zum Beispiel für eines von /a/, /b/, /x/, /0/ oder /9/. Falls wir nun festlegen wollen, dass wir einen Kleinbuchstaben haben wollen, können wir zum Beispiel schreiben

(167) `[abcdefghijklmnopqrstuvwxyz]`

(Die Reihenfolge ist allerdings unerheblich.) Da dies allerdings auch zu umständlich ist, darf man gewisse Gruppen auch noch wie folgt abkürzen. /[0-9]/ bezeichnet die Ziffern, /[a-f]/ die Kleinbuchstaben von 'a' bis 'f', /[A-F]/ die Großbuchstaben von 'A' bis 'F'; und /[a-fA-F]/ ist für ein Einzelzeichen, das ein Klein- oder Großbuchstabe ist, von 'a' bis 'f' (bzw. von 'A' bis 'F'). Wenn die öffnende Klammer von /^/ gefolgt wird, wird die Menge invertiert. Jetzt bezeichnet es alle Symbole, die *nicht* zu den erwähnten gehören; zum Beispiel /[^a-z]/ meint: alles außer Kleinbuchstaben. Diese Konvention hat nun wiederum den Effekt, dass der Bindestrich sowie der Hut zu Sonderzeichen werden. Aber da sie nur an bestimmten Stellen auftreten können, hat man Folgendes vereinbart: ein Bindestrich am Anfang oder am Ende bezeichnet sich selbst: /[-az]/ wie auch /[az-]/ trifft genau auf die Symbole /-/, /a/ und /z/ zu, aber zum Beispiel nicht auf /b/. Ein Hut am Ende bezeichnet sich ebenfalls selbst.

Jetzt wissen wir ungefähr Bescheid, wie man reguläre Ausdrücke schreibt. Wie aber lassen sie sich verwenden? Zum Beispiel wollen wir einen regulären Ausdruck finden. Was aber heißt das, und wie macht man das? Zunächst einmal will ich sagen, was es heißt, nach einem regulären Ausdruck *t* zu suchen. Das bedeutet, in einer Zeichenkette ein Teilwort zu finden, das unter *t* fällt. Das ist aber nicht so eindeutig, wie es sein müsste. Denn es kann viele Vorkommen geben. Die Konvention ist die: wir suchen in der Zeichenkette dasjenige Vorkommen, das früher oder genauso früh wie alle anderen beginnt. Falls wir in /Banane/ zum Beispiel das Wort /an/ suchen, so bekommen wir das erste (bestehend aus den Positionen 1 und 2) und nicht das zweite (bestehend aus den Position 3 und 4) geliefert. Dies ist aber immer noch nicht eindeutig. Nehmen wir an, wir suchen nach /a/ oder /an/, dann haben wir zwei Vorkommen, die bei Position 1 beginnen. Das erste ist von der Länge 1 (bis einschließlich Position 1), das zweite von der Länge 2 (bis einschließlich Position 2). In diesem Fall unterscheidet man zwischen einem **gierigen Matcher**, welches stets das längste Wort ausgibt, und einem **faulen Matcher**, der stets das kürzeste Wort ausgibt. In Unix sind Matcher standardmä-

ßig gierig.

OCaml hat nun eine Funktion namens `search_forward`. Diese braucht einen regulären Ausdruck, eine Zeichenkette und eine Zahl. Die Zahl gibt die Position an, ab der gesucht wird. Das Ergebnis ist wieder eine Zahl, welche uns die erste Position angibt, wo ein Wort gefunden wurde. Falls der Matcher nicht fündig wird, gibt er eine Ausnahme aus: `Not_found`. Wer sich die Ausnahme ersparen will, kann mit `string_match` einen Vortest machen. Dies kostet allerdings unnötig Zeit. OCaml hat einen generellen Mechanismus, um mit Ausnahmen umzugehen. Dies ist das Konstrukt `try ··· with ···`.

(168) `try Str.search_forward ··· with Not_found ···`

Dies bedeutet: versuche die Aufgabe zu lösen (hier: ein Teilwort zu suchen); falls Du die genannte Ausnahme triffst (hier: `Not_found`), dann tue etwas anderes. Siehe dazu das Handbuch.

Wenn man eine Instanz gefunden hat, ruft man `match_beginning` bzw. `match_end` auf, und sie liefern uns Beginn und Ende des Teilworts. Diese brauchen allerdings `unit` als Input, also muss man sie wie folgt aufrufen: /`Str.match_beginning ()`/.

Schauen wir uns das mal am Beispiel des Unterschieds zwischen einem faulen und einem gierigen Matcher an. Wir wollen a^* in der Zeichenkette /`bcagaa`/ finden.

(169)
```
# let ast = Str.regexp "a*";;
val ast : String.regexp = <abstr>
# Str.search_forward ast "bcagaa" 0;;
- : int = 2
# Str.match_end ();;
- : int = 3
# Str.match_beginning ();;
- : int = 2
# Str.search_forward ast "bcagaa" 3;;
- : int = 4
# Str.match_end ();;
- : int = 6
# Str.match_beginning ();;
- : int = 4
```

Man beachte, dass OCaml uns das Ende als diejenige Position mitteilt, die nicht mehr in der Zeichenkette ist. So bekommen wir die Antwort 6, obwohl es eine solche Position gar nicht gibt. Das bedeutet schlicht, dass wir am Ende der Zeichenkette sind. Man mache sich klar, dass diese Konvention notwendig ist. Denn die leere Zeichenkette muss ja verschieden sein von einem Einzelzeichen. Bei der leeren Zeichenkette sind Beginn und Ende gleich, bei einem Einzelzeichen ist die Endposition um eins größer.

Ein Vorteil des gierigen Matchers gegenüber dem faulen ist, dass wir sehr leicht prüfen können, ob die gesamte Zeichenkette unter einen regulären Ausdruck fällt. Dazu versuchen wir einfach, ab Position 0 zu matchen. Wenn dies gelingt und die Endposition gerade der Länge der Zeichenkette entspricht, dann ist die Zeichenkette eine Instanz.

(170)
```
let exact_match r s =
    if Str.string_match r s 0
    then ((Str.match_beginning () = 0)
        && (Str.match_end () = (String.length s)))
    else false;;
```

Ein ganz anderes Thema, aber nicht weniger wichtig, sind Ersetzung. Zeichenkettenersetzung arbeitet in zwei Phasen. Die erste ist die des Suchens. Die zweite ist die eigentliche Ersetzung. Um zu wissen, was am Ende herauskommt, muss man sehr gut über den Suchalgorithmus Bescheid wissen. Angenommen, wir wollen in der Zeichenkette $\vec{x}$ ein Vorkommen des Ausdrucks s durch die Zeichenkette $\vec{y}$ ersetzen. Dann sucht zunächst der Matcher ein Vorkommen von s. Hat er es gefunden, wird es herausgenommen und $\vec{y}$ eingefügt. Man beachte: wir suchen mittels eines regulären Ausdrucks, aber das, was wir einsetzen, ist eine Zeichenkette.

Hier ist ein Beispiel. Eine IP Adresse ist eine Folge der Form $\vec{c}_0.\vec{c}_1.\vec{c}_2.\vec{c}_3$, wo $\vec{c}_i$ eine Zeichenkette ist, genauer eine Zahl von 0 to 255. Führende Nullen dürfen weggelassen werden, aber $\vec{c}_i$ darf nicht leer sein. Beispiele sind `192.168.0.1`, `145.146.29.243`, oder `127.0.0.1`. Der folgende Ausdruck beschreibt die legalen $\vec{c}_i$.

(171)
```
let tm = "[0-1][0-9][0-9]\\|2[0-4][0-9]\\|25[0-5]
                    \\|[0-9][0-9]\\|[0-9]"
```

(Der Zeilenumbruch darf nicht in dem Programm selbst vorkommen.) Ich habe dabei verwendet, dass die Verkettung stärker bindet als die Disjunktion. Es ist

also /ab\\|cd/ die Disjunktion aus /ab/ und /cd/, nicht etwa aus /abd/ und /acd/. Der Ausdruck kann etwas ökonomischer gestaltet werden, aber ich will darauf jetzt nicht eingehen.

Diese Zeichenkette wird mit `Str.regexp` in einen regulären Ausdruck verwandelt, den wir `tm` nennen wollen.

(172)
```
let m = Str.regexp ( "\\("^tm^"\\)\\.\\("^tm^
    "\\)\\.\\("^tm^"\\)\\.\\("^tm^"\\)")
```

(Dieser Zeilenumbruch hingegen darf in dem Programm vorkommen.) Dieser Ausdruck liefert eine genaue Maske für IP Adressen. Wozu aber sind die Klammern gut? Die Ausdrücke in Klammern werden im Laufe eines Matches ebenfalls gematcht. Im Gegensatz zu anderen Teiltermen ist das Ergebnis eines solchen Matchings nachher abrufbar. Die Klammerausdrücke werden intern abgezählt (von 1 bis 9) und wir können für jede Zahl abrufen, wo das entsprechende Teilwort gefunden wurde. Hier sei unsere IP Adresse.

(173)
```
...129.23.145.110...
```

Wir nehmen an, `m` matche mit der Anfangsposition 1230. Die erste Klammer hat die Nummer 1. Sie spannt in dieser Situation ein Wort der Länge 3 ein, nämlich die Zeichenkette /129/. Dies können wir mit der Funktion `matched_group` abfragen. Das erste Argument ist eine Zahl, dann kommt die Zeichenkette, und dann bekommen wir die Zeichenkette, die an dieser Stelle gefunden wurde. Wir können auch die Positionen abfragen, wenn wir dies wünschen (mit `Str.group_beginning` und `Str.group_end`). Nach erfolgter Suche können wir in dem String genannt `u` folgendes aufrufen:

```
let s = "Der erste Teil der IP Adresse ist
    "^(Str.matched_group 1 u) ^"."
```

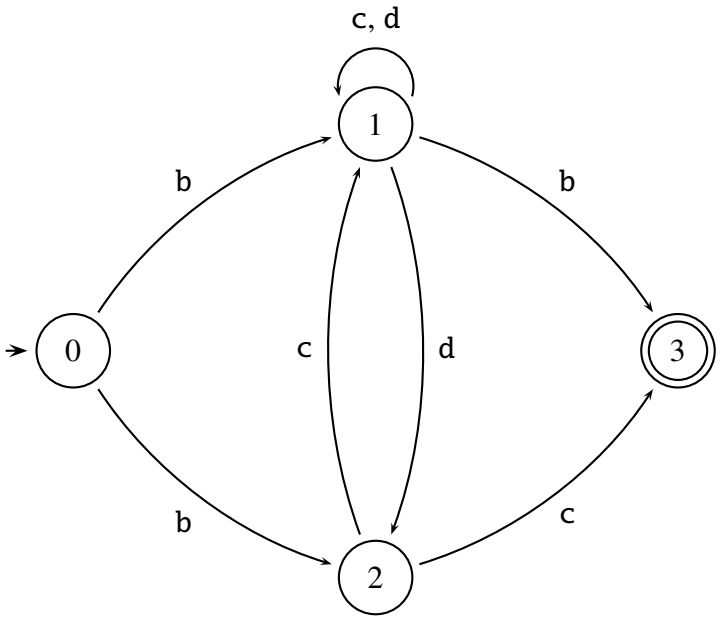
Wir bekommen dann den folgenden Wert für `s`:

(174)
```
"Der erste Teil der IP Adresse ist 129.
```

(Der letzte Punkt ist dabei nicht Teil der Adresse!) Um dies nun dazu zu benutzen, in einer Zeichenkette Ersetzungen vorzunehmen, gibt es die Variablen `\\0`, `\\1`, `\\2`,..., `\\9`. Nach einer erfolgreichen Suche können wird die gesamte Zeichenkette der Variablen `\\0` zugewiesen, der Inhalt der ersten Klammer der Variable

`\\1`, der Inhalt der zweiten Klammer der Variable `\\2` tund so fort. Eine **Schablone** ist eine Zeichenkette, die außer Buchstaben auch noch die Variable `\\`n enthält. Die Funktion `global_replace` nimmt als Eingabe einen regulären Ausdruck a, eine Zeichenkette $\vec{t}$ und eine Zeichenkette $\vec{u}$. Als Ausgabe bekommen wir das Ergebnis der globalen Ersetzung der Vorkommen von a in $\vec{u}$ durch $\vec{t}$. Hierbei ist $\vec{t}$ zwar eine Zeichenkette, agiert aber in Wirklichkeit wie eine Schablone. Denn bei der Ersetzung von dem Vorkommen werden die Variablen instantiiert. Nehmen wir zum Beispiel an, wir wollen von der IP Adresse nur den ersten Teil im Text behalten. Dann können wir dafür die Schablone /`\\1`/ benutzen. Wenn wir die IP Adresse durch den ersten Teil gefolgt von /`.0.1`/ ersetzen wollen, dann können wir die Schablone /`\\1.0.1`/ nehmen. Man beachte dass in der Schablone nur der Schrägstrich ein Sonderzeichen ist. Ein Punkt ist keines, muss also nicht maskiert werden.

Abbildung 1: Ein endlicher Automat



13 Endliche Automaten

Ein **endlicher Automat** ist ein Quintupel

(175) $\mathfrak{A} = \langle A, Q, i_0, F, \delta \rangle$

derart, dass A eine endliche Menge ist, das sogenannte **Alphabet**, Q eine endliche Menge, die Menge der **Zustände**, $i_0 \in Q$ der **Startzustand**, $F \subseteq Q$ die Menge der **akzeptierenden Zustände** und $\delta \subseteq Q \times A \times Q$ die **Übergangsrelation**. Abbildung 1 zeigt einen endlichen Automaten über dem Alphabet $A = \{\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d}\}$. Die Zustände werden durch Kreise symbolisiert, wobei doppelt eingerandete Kreise akzeptierende Zustände bezeichnen (hier 3). Ein kleiner Pfeil zeigt auf den initialen Zustand (hier 0). Die Übergangsrelation wird durch die Pfeile gegeben. Der Automat hat die folgenden Übergänge.

(176) $\{\langle 0, \mathsf{b}, 1\rangle, \langle 0, \mathsf{b}, 2\rangle, \langle 1, \mathsf{b}, 2\rangle, \langle 1, \mathsf{c}, 1\rangle, \langle 1, \mathsf{d}, 1\rangle, \langle 2, \mathsf{c}, 1\rangle, \langle 2, \mathsf{c}, 3\rangle\}$

Das Alphabet geht aus der Zeichung nicht hervor: man sieht keine Übergänge für a, obwohl dies Buchstabe des Alphabets ist. Allerdings wird man nur in seltenen Fällen Automaten bauen wollen, die überhaupt keinen Übergang für einen bestimmten Buchstaben vorsehen. (Und auch dann könnte man zum Beispiel noch einen Zustand, sagen wir 4, hinzunehmen, und lediglich einen einzigen neuen Übergang, nämlich $\langle 4, \mathsf{a}, 4\rangle$, und wir hätten a im Bild verankert, ohne dass er je zum Einsatz kommen könnte.)

Ich schreibe $x \xrightarrow{a}_{\mathfrak{A}} y$ oder einfach $x \xrightarrow{a} y$ falls $\langle x, a, y\rangle \in \delta$. Diese Schreibweise wird auf reguläre Ausdrücke wie folgt erweitert.

$$
\begin{aligned}
x \xrightarrow{0} y &:\Leftrightarrow \text{ falsch}\\
x \xrightarrow{\varepsilon} y &:\Leftrightarrow x = y\\
x \xrightarrow{s\cup t} y &:\Leftrightarrow x \xrightarrow{s} y \text{ oder } x \xrightarrow{t} y\\
x \xrightarrow{st} y &:\Leftrightarrow \text{es existiert ein } z \text{ mit } x \xrightarrow{s} z \text{ und } z \xrightarrow{t} y\\
x \xrightarrow{s^*} y &:\Leftrightarrow \text{es existiert ein } n \text{ mit } x \xrightarrow{s^n} y
\end{aligned}
\tag{177}
$$

Dies erlaubt uns, ganz kurz und bündig die von einem Automaten **akzeptierte Sprache** zu definieren.

$$L(\mathfrak{A}) := \{\vec{x} \in A^* : \text{es gibt } q \in F \text{ mit } i_0 \xrightarrow{\vec{x}} q\} \tag{178}$$

Es gibt noch einen etwas allgemeinere Begriff eines endlichen Automaten, den ich **endlichen ε-Automat** nennen will. Die Definition ist analog zu der eines endlichen Automaten, nur dass $\delta \subseteq Q \times (A \cup \{\varepsilon\}) \times Q$. Das bedeutet, dass nunmehr $q \xrightarrow{\varepsilon} q'$ auch dann der Fall sein kann, wenn $q \neq q'$. Ein ε-Automat erlaubt nämlich leere Übergänge, dh Übergänge mit dem leeren Wort. Ist $\mathfrak{E}$ ein ε-Automat, so schreiben wir wie oben $q \xrightarrow{\vec{x}} q'$. Dies ist definiert wie in (177) mit Ausnahme von

$$x \xrightarrow{\varepsilon} y \Leftrightarrow x = y \text{ oder } \langle x, \varepsilon, y\rangle \in \delta \tag{179}$$

Satz 13.1 *Zu jedem ε-Automaten $\mathfrak{E} = \langle A, Q, i_0, F, \delta\rangle$ gibt es einen endlichen Automaten $\mathfrak{A}$ derart, dass $L(\mathfrak{A}) = L(\mathfrak{E})$.*

Beweis. Zunächst sei vorausgesetzt, dass $\varepsilon \notin L(\mathfrak{E})$. Sei $\mathfrak{E} = \langle A, Q, i_0, F, \delta\rangle$. Wir setzen $\mathfrak{A} := \langle A, Q, i_0, F, \delta'\rangle$ mit

$$\delta' := \{\langle x, a, y\rangle : x \xrightarrow{a}_{\mathfrak{E}} y\} \tag{180}$$

Nun zeigen wir für alle $\vec{x} \neq \varepsilon$:

$$q \xrightarrow{\vec{x}}_{\mathfrak{A}} q' \text{gdw. } q \xrightarrow{\vec{x}}_{\mathfrak{E}} q' \tag{181}$$

Wir zeigen nun (181) durch Induktion über die Länge von $\vec{x}$. Es genügt offensichtlich, den Fall $\vec{x} = a$ zu zeigen. Es sei $q \xrightarrow{a}_{\mathfrak{E}} q'$. (Das ist der Induktionsanfang; der Intuktionsschritt ist leicht.) Das bedeutet, es gibt $\mathfrak{E}$-Übergänge

$$q \xrightarrow{\varepsilon} q_1 \xrightarrow{\varepsilon} q_2 \cdots \xrightarrow{\varepsilon} q_n \xrightarrow{a} r_m \xrightarrow{\varepsilon} r_{m-1} \cdots \xrightarrow{\varepsilon} r_2 \xrightarrow{\varepsilon} r_1 \xrightarrow{\varepsilon} q' \tag{182}$$

Nach Definition ist dann aber $q \xrightarrow{a} q'$, weil $\langle q, a, q'\rangle \in \delta'$. Die Umkehrung gilt ebenso.

Nun zu dem Fall $\varepsilon \in L(\mathfrak{E})$. Wähle ein $i \notin Q$. Wir setzen $\mathfrak{A} := \langle A, Q \cup \{i\}, i, F \cup \{i\}, \delta'\rangle$, wobei für alle $a \in A$

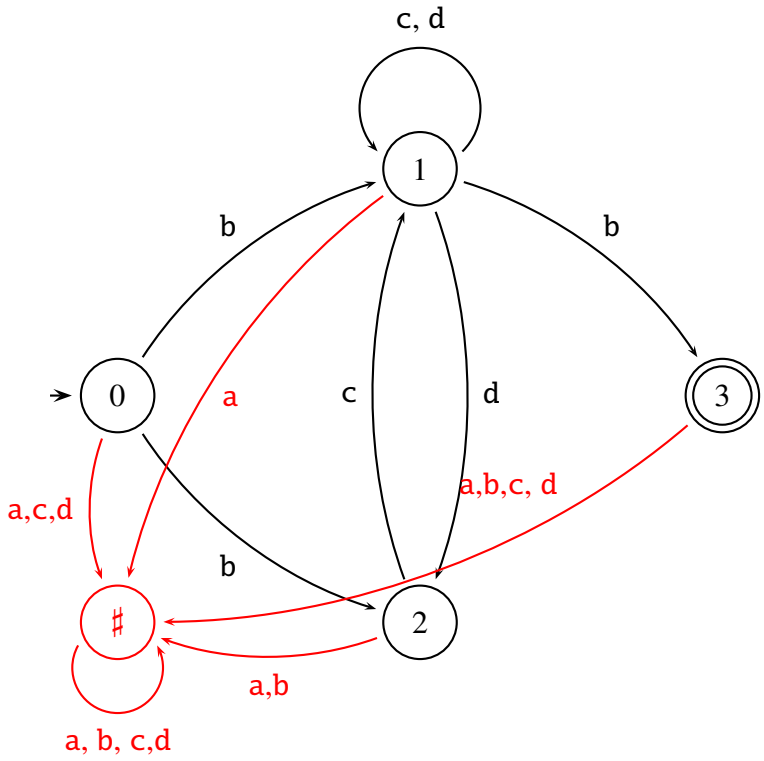
$$\begin{aligned} \delta' := \quad & \{\langle x, a, y\rangle : x, y \in Q \text{ und } x \xrightarrow{a}_{\mathfrak{E}} y\} \\ \cup \quad & \{\langle i, a, x\rangle : x \in Q \text{ und } i_0 \xrightarrow{a}_{\mathfrak{E}} x\} \end{aligned} \tag{183}$$

Man beachte, dass $\varepsilon \in L(\mathfrak{A})$. Insofern müssen wir für unsere Behauptung (181) nur für nichtleere Worte zeigen. Ferner gilt: ist $\vec{x} = a\vec{y}$, und ist $i \xrightarrow{a} q \xrightarrow{\vec{y}} r$, dann ist $q \in Q$ und jeder $\mathfrak{A}$-Pfad verläuft in Q, sodass wir den obigen Beweis wieder verwenden können. ⊣

Ein Automat ist **partiell**, falls es für ein q und $a \in A$ kein q' gibt mit $q \xrightarrow{a} q'$. Ist $\mathfrak{A}$ nicht partiell, so ist er **total**. Es ist nicht schwer, einen Automat total zu machen (bei gleicher Sprache). Man füge nur einen neuen Zustand $q_\sharp$ hinzu samt allen Übergängen der Form $\langle q, a, q_\sharp\rangle$, wo kein Übergang von q mit a zu einem anderen Zustand existiert. Schließlich füge man noch die Übergänge $\langle q_\sharp, a, q_\sharp\rangle$ hinzu für jedes $a \in A$. $q_\sharp$ ist nicht akzeptierend.

Proposition 13.2 *Zu jedem Automaten $\mathfrak{A}$ existiert ein totaler Automat $\mathfrak{A}^\sharp$ mit $L(\mathfrak{A}^\sharp) = L(\mathfrak{A})$.*

Abbildung 2: Totalisierung eines Automaten



Die Datei `fstate.ml` enthält ein Modul namens `FSA`. Darin wird ein Objekttyp `automaton` definiert. Wer sich damit vertraut macht, kann mit Automaten in OCaml arbeiten.

(184)
```
class automaton =
  object
    val mutable i = 0
    val mutable x = StateSet.empty
    val mutable y = StateSet.empty
    val mutable z = CharSet.empty
    val mutable t = Transitions.empty
    method get_initial = i
    method get_states = x
    method get_astates = y
    method get_alph = z
    method get_transitions = t
    method initialize_alph = z <- CharSet.empty
...
    method list_transitions = Transitions.elements t
  end;;
```

Das Programm ist nicht nach dem Gesichtspunkt der Effizienz geschrieben. Sondern es kopiert die Definition relativ genau. Zum Beispiel ist das Alphabet eine *Menge* von Buchstaben und keine Liste (was einigen Aufwand erspart hätte). Desgleichen für `StateSet`. Dann gibt es den Typ "transition", mit drei Feldern, `ein`, `symbol` und `aus`. Diese legen den Zustand vor dem Übergang, das Symbol und den Endzustand fest. Das Attribut `mutable` sagt, dass die Werte im Objekt verändert werden können. Der Wert nach dem Gleichheitszeichen zeigt den Wert, der beim Erschaffen des Objekts zugewiesen wird. Man muss den Wert also nie selber setzen, aber es ist ratsam, dies zu tun, um Fehler zu vermeiden. Wie schon besprochen, braucht man bei einem Objekt eine Methode selbst um den Wert einer Variable anzuzeigen. Man lade also zunächst `fstate.ml` und öffne `FSA`.

(185)
```
# let a = new automaton;;
val a : FSA.automaton = <obj>
```

Dieser Automat hat den Startzustand 0, alle Mengen sind leer, nur die Zustandsmenge ist {0}. Mit dem folgenden Befehl addiert man den Buchstaben `z` zum

Alphabet.

(186)
```
# a#add_alph 'z';;
- : unit = ()
```

Oder hier ist der Befehl, einen Übergang hinzuzufügen.

(187)
```
# a#add_transitions {ein = 0; aus = 2; symbol = 'z'};;
```

Den Automaten kann man also Schritt für Schritt definieren.

Satz 13.3 *Es sei $\mathfrak{A}$ ein endlicher Automat. Dann ist $L(\mathfrak{A})$ regulär.*

Beweis. Wir stellen den Automaten durch ein Gleichungssystem dar. Dazu sei für einen Zustand q folgende Sprache definiert

$$T_q := \{\vec{x} : \text{ es existiert } r \in F : q \xrightarrow{\vec{x}} r\}. \tag{188}$$

Die T_q sind Sprachen. Falls nun $\langle q, a, r\rangle$ so ist $a \cdot T_r \subseteq T_q$. Wir können die T_q miteinander in Beziehung setzen. (**Fall 1**) $q \notin F$.

$$T_q = \bigcup_{\langle q,a,r\rangle\in\delta} a \cdot T_r \tag{189}$$

Sei nämlich $\vec{x} \in T_q$. Dann ist $q \xrightarrow{\vec{x}} s \in F$ für ein $s \in F$. Da $q \notin F$, so ist $s \neq q$ und deswegen $\vec{x} \neq \varepsilon$. Also existiert ein a mit $\vec{x} = a\vec{y}$ und ein r mit $q \xrightarrow{a} r \xrightarrow{\vec{y}} s$. Dies bedeutet $\vec{y} \in T_r$ und $\vec{x} = a\vec{y} \in a \cdot T_r$. Diese Argumentation ist umkehrbar, und so sind die Mengen gleich.

(**Fall 2**) $q \in F$. Dann haben wir

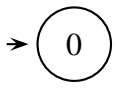
$$T_q = \varepsilon \cup \bigcup_{\langle q,a,r\rangle\in\delta} a \cdot T_r \tag{190}$$

(190) zeigt man ähnlich (da $q \in F$, darf jetzt $\vec{x}$ zusätzlich das leere Wort sein). Wir haben also ein reguläres Gleichungssystem, und für jedes q gibt es einen regulären Term t_q mit $T_q = L(t_q)$. Es ist dann

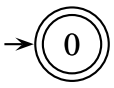
$$L(\mathfrak{A}) = T_{i_0} \tag{191}$$

Abbildung 3: Endliche Automaten für elementare Terme

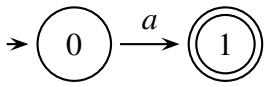
(a) 0



(b) ε



(c) a



Dies ist regulär. ⊣

Die Umkehrung gilt nun ebenso: zu jeder regulären Sprache S existiert ein endlicher Automat $\mathfrak{A}$ mit $L(\mathfrak{A}) = S$. Wir werden einen solchen Automaten konstruieren.

Beginnen wir mit dem Term $s = 0$. Man nehme einen beliebigen Automaten und setze $F := \varnothing$. Dieser Automat akzeptiert keine Zeichenkette. Nun sei $s = \varepsilon$. Man nehme zwei Zustände, 0 und 1. 0 sei der Anfangszustand und akzeptierend. 1 sei nicht akzeptierend. $\delta := \{\langle 0, a, 1\rangle, \langle 1, a, 1\rangle : a \in A\}$. Nun sei $s = a$. Dann setze $Q := \{0, 1\}$, 0 initial, $F := \{1\}$; $\delta := \{\langle 0, a, 1\rangle\}$. Dies sichert den Induktionsanfang (siehe Abbildung 3). Nun gehen wir die komplexen Ausdrücke durch. Zum Beispiel st.

Lemma 13.4 *Ist $L = L(\mathfrak{A})$ und $M = L(\mathfrak{B})$, so existiert ein endlicher Automat $\mathfrak{C}$ mit $L \cdot M = L(\mathfrak{C})$.*

Beweis. Sei $\mathfrak{A} = \langle A, Q, i_0, F, \delta\rangle$ und $\langle B, R, j_0, G, \eta\rangle$. Es sei $Q \cap R = \varnothing$. Dann konstruieren wir $\mathfrak{C} = \langle A \cup B, Q \cup R, i_0, G, \theta\rangle$, wobei

(192) $\theta = \delta \cup \eta \cup \{\langle q, \varepsilon, j_0\rangle : q \in F\}$

$\mathfrak{C}$ ist ein ε-Automat. Laut Satz 13.1 können wir einen gleichwertigen endlichen Automaten daraus konstruieren. Zu zeigen ist $L(\mathfrak{C}) = L \cdot M$. Sei dazu $\vec{x} \in L(\mathfrak{C})$. Ein beliebiger Lauf $i_0 \xrightarrow{\vec{x}} q \in G$ muss durch j_0 gehen. Denn die einzige Weise, von den Zuständen aus $\mathfrak{A}$ zu denen aus $\mathfrak{B}$ zu gelangen, ist via einem Übergang $r \xrightarrow{\varepsilon} j_0$, wo $r \in F$. Das wiederum heißt, es lässt sich der Lauf und $\vec{x}$ zerlegen in $i_0 \xrightarrow{\vec{y}} r \xrightarrow{\varepsilon} j_0 \xrightarrow{\vec{z}} q$, wo $r \in F$ und $q \in G$. Das bedeutet $\vec{x} = \vec{y}\vec{z}$ mit $\vec{y} \in L(\mathfrak{A})$ und $\vec{z} \in L(\mathfrak{B})$. Die Umkehrung ist ähnlich. ⊣

Nun zu s^*.

Lemma 13.5 *Es sei $L = L(\mathfrak{A})$. Dann existiert ein $\mathfrak{B}$ mit $L(\mathfrak{B}) = L^*$.*

Beweis. Sei $\mathfrak{A} = \langle A, Q, i_0, F, \delta\rangle$. Definiere

$$\delta' := \delta \cup \{\langle q, \varepsilon, i_0\rangle : q \in F\} \tag{193}$$

Denn ist

$$\mathfrak{B} := \langle A, Q, i_0, \{i_0\}, \delta'\rangle \tag{194}$$

Man beachte, dass, wenn $q \xrightarrow{\vec{x}}_{\mathfrak{B}} q' \in F$, so auch $q \xrightarrow{\vec{x}}_{\mathfrak{B}} i_0$, sodass wir tatsächlich uns auf die Menge $\{i_0\}$ einschränken können. Nun also zur Behauptung, dass $L(\mathfrak{B}) = L^*$. Es sei $\vec{x} \in L(\mathfrak{B})$. Dann existiert ein akzeptierender Lauf und eine Zerlegung dieses Laufes wie folgt:

$$i_0 \xrightarrow{\vec{x}_0}_{\mathfrak{B}} q_0 \xrightarrow{\varepsilon}_{\mathfrak{B}} i_0 \xrightarrow{\vec{x}_0}_{\mathfrak{B}} q_1 \xrightarrow{\varepsilon}_{\mathfrak{B}} i_0 \cdots i_0 \xrightarrow{\vec{x}_0}_{\mathfrak{B}} q_{n-1} \xrightarrow{\varepsilon}_{\mathfrak{B}} i_0 \tag{195}$$

wobei keiner der Teilläufe für die $\vec{x}_i$ einen ε-Übergang enthält. Damit ist dann $\vec{x}_i \in L(\mathfrak{A})$, da jeder Lauf in $\mathfrak{A}$ ist. Und so ist $\vec{x} \in L(\mathfrak{A})^*$. Ist $\vec{x} \in L(\mathfrak{A})^*$, so besitzt $\vec{x}$ eine Zerlegung $\vec{x} = \vec{x}_0\vec{x}_1 \cdots \vec{x}_{n-1}$ mit $\vec{x}_i \in L(\mathfrak{A})$ für jedes $i < n$. Und so existiert ein Lauf wie in (195). ⊣

Schließlich wenden wir uns $s \cup t$ zu. Wir konstruieren den folgenden Automaten.

$$\mathfrak{A} \oplus \mathfrak{B} = \langle A, Q^{\mathfrak{A}} \times Q^{\mathfrak{B}}, \langle i_0^{\mathfrak{A}}, i_0^{\mathfrak{B}}\rangle, (F^{\mathfrak{A}} \times Q^{\mathfrak{B}}) \cup (Q^{\mathfrak{A}} \times F^{\mathfrak{B}}), \delta^{\mathfrak{A}} \times \delta^{\mathfrak{B}}\rangle \tag{196}$$

$$\delta^{\mathfrak{A}} \times \delta^{\mathfrak{B}} = \{\langle x_0, x_1\rangle \xrightarrow{a} \langle y_0, y_1\rangle : \langle x_0, a, y_0\rangle \in \delta^{\mathfrak{A}}, \langle x_1, a, y_1\rangle \in \delta^{\mathfrak{B}}\} \tag{197}$$

Lemma 13.6 *Für jede Zeichenkette* $\vec{u}$

$$\langle x_0, x_1\rangle \xrightarrow{\vec{u}}_{\mathfrak{A}\oplus\mathfrak{B}} \langle y_0, y_1\rangle \quad \Leftrightarrow \quad x_0 \xrightarrow{\vec{u}}_{\mathfrak{A}} y_0 \text{ und } y_0 \xrightarrow{\vec{u}}_{\mathfrak{B}} y_1 \tag{198}$$

Der Beweis erfolgt durch Induktion über die Länge von $\vec{u}$. Es ist damit leicht zu sehen, dass $L(\mathfrak{A}\oplus\mathfrak{B}) = L(\mathfrak{A}) \cup L(\mathfrak{B})$. Denn $\vec{u} \in L(\mathfrak{A}\oplus\mathfrak{B})$ genau dann, wenn $\langle i_0^{\mathfrak{A}}, i_0^{\mathfrak{B}}\rangle \xrightarrow{\vec{u}} \langle x_0, x_1\rangle$, für ein $\langle x_0, x_1\rangle \in (F^{\mathfrak{A}}\times Q^{\mathfrak{B}})\cup(Q^{\mathfrak{A}}\times F^{\mathfrak{B}})$. Letzteres ist äquivalent zu $i_0^{\mathfrak{A}} \xrightarrow{\vec{u}} x_0$ und $i_0^{\mathfrak{B}} \xrightarrow{\vec{u}} x_1$, wo $x_0 \in F^{\mathfrak{A}}$ oder $x_1 \in F^{\mathfrak{B}}$. Das ist wiederum nichts anderes als $\vec{u} \in L(\mathfrak{A})$ oder $\vec{u} \in L(\mathfrak{B})$.

Satz 13.7 *Genau dann ist L regulär, wenn es einen endlichen Automaten* $\mathfrak{A}$ *gibt mit* $L = L(\mathfrak{A})$.

Dieses Resultat können wir verwenden, um noch mehr über Abschlusseigenschaften von regulären Sprachen zu erfahren. Es gibt nämlich Konstruktionen auf Automaten, die sich auf regulären Ausdrücken nicht so ohne weiteres replizieren lassen. Ein Beispiel ist der Schnitt von regulären Sprachen.

Satz 13.8 *Mit L und M ist auch* $L \cap M$ *regulär.*

Wir basteln einen Automaten $\mathfrak{A}\times\mathfrak{B}$ mit dem Unterschied, dass die akzeptierenden Zustände etwas anders definiert werden.

$$\mathfrak{A}\times\mathfrak{B} := \langle A, Q^{\mathfrak{A}}\times Q^{\mathfrak{B}}, \langle i_0^{\mathfrak{A}}, i_0^{\mathfrak{B}}\rangle, F^{\mathfrak{A}}\times F^{\mathfrak{B}}, \delta^{\mathfrak{A}}\times\delta^{\mathfrak{B}}\rangle \tag{199}$$

Man zeigt ganz leicht, dass $L(\mathfrak{A}\times\mathfrak{B}) = L(\mathfrak{A}) \cap L(\mathfrak{B})$.

Definition 13.9 *Ein Automat heißt* ***deterministisch****, falls für jeden Buchstaben a und Zustände p, q, q′ gilt: ist* $p \xrightarrow{a} q, q'$*, so ist* $q = q'$.

Es ist leicht zu sehen, dass ein Automat genau dann deterministisch ist, wenn aus $p \xrightarrow{\vec{x}} q, q'$ folgt $q = q'$.

Proposition 13.10 *Es sei* $\mathfrak{A} = \langle A, Q, i_0, F, \delta\rangle$ *ein deterministischer Automat. Sei* $\mathfrak{A}^n := \langle A, Q, i_0, Q - F, \delta\rangle$. *Dann ist* $L(\mathfrak{A}^n) = A^* - L(\mathfrak{A})$.

Zu jedem Automaten existiert nun tatsächlich ein deterministischer Automat mit gleicher Sprache. Dazu sei $\mathfrak{A}^d$ wie folgt definiert.

$$
\begin{array}{ll}
Q^d & := \wp(Q) \\
i^d & := \{i_0\} \\
F^d & := \{U : U \cap F \neq \varnothing\} \\
\delta^d & := \{\langle U, a, V\rangle : V = \{q : \text{es existiert } p \in U \text{ mit } p \stackrel{a}{\to} q\}\} \\
\mathfrak{A}^d & := \langle A, Q^d, i^d, F^d, \delta^d\rangle
\end{array} \tag{200}
$$

Satz 13.11 *$\mathfrak{A}^d$ ist deterministisch. $L(\mathfrak{A}^d) = L(\mathfrak{A})$. Ferner ist $U \stackrel{\vec{x}}{\to} V$ genau dann, wenn V die Menge aller q ist, für die ein $p \in U$ existiert mit $p \stackrel{\vec{x}}{\to} q$ in $\mathfrak{A}$.*

$\mathfrak{A}^d$ ist sicher deterministisch. Die zweite Behauptung folgt unmittelbar aus der dritten; denn ist $\vec{x} \in L(\mathfrak{A}^d)$, so ist $\{i_0\} \stackrel{\vec{x}}{\to} V$ für ein $V \in F^d$, das heißt für ein V mit $V \cap F \cap \varnothing$. Also existiert ein $q \in F \cap V$, und damit ist mit der dritten Behauptung $p \stackrel{\vec{x}}{\to} q$ in $\mathfrak{A}$, was nichts anderes als $\vec{x} \in L(\mathfrak{A})$ ist. Diese Argumentation ist umkehrbar. Die dritte Behauptung ergibt sich leicht durch Induktion über die Länge von $\vec{x}$. Die Fälle $|\vec{x}| \leq 1$ sind durch die Definition erledigt. Nun sei $\vec{x} = \vec{y}a$. Dann ist $U \stackrel{\vec{x}}{\to} W$ und sei V die Menge der q, für die $p \in U$ existiert mit $p \stackrel{\vec{x}}{\to} q$. Dann ist $U \stackrel{\vec{x}}{\to} W$ genau dann, wenn $U \stackrel{\vec{y}a}{\to} W$ ist für ein W, das heißt aber $U \stackrel{\vec{y}}{\to} V' \stackrel{a}{\to} W$. Weil der Automat deterministisch ist, ist $V' = V$. Nun ist W die Menge der q derart, dass ein q' in V existiert mit $q' \stackrel{a}{\to} q$. Nach Definition von V existiert ein $q \in U$ mit $q \stackrel{\vec{x}}{\to} q'$. Insgesamt existiert also für jedes $q \in W$ ein $p \in U$ mit $p \stackrel{\vec{x}a}{\to} q$. Dies zeigt die dritte Behauptung.

Ich stelle hier noch eine andere Methode vor. Diese baut den Automaten aus dem Term selbst.

Definition 13.12 *Es sei $L \subseteq A^*$ eine Sprache. Dann sei*

$$L^p = \{\vec{x} : \textit{es gibt } \vec{y} \textit{ mit } \vec{x}\frown\vec{y} \in L\} \tag{201}$$

*Dies ist die sogenannte **Präfixhülle** von L.*

Wenn wir eine Zeichenkette nach Vorkommen von Worten in L an der Stelle n durchgehen, beginnen wir an dieser Position zunächst ein Präfix der Länge 1 zu matchen. Ist dieses schon in L, so sind wir fertig. Wenn nicht, gehen wir eine Position weiter und versuchen, nunmehr ein Präfix der Länge 2 zu matchen. Ist dieses in L, so sind wir fertig, ansonsten geht es weiter.

Dies ist nun nicht ganz so so einfach, wie es sich anhört, aber daraus lässt sich ein sehr schöner Algorithmus (in Form eines Automaten) basteln. Denn wenn L regulär ist, ist auch L^p regulär (das kann man mit Hilfe von Automaten sehr schön sehen). Wir können auch den dazugehörigen regulären Ausdruck berechnen. Sei s ein regulärer Ausdruck. Wir definieren die Präfixhülle von s, $s^\dagger$, wie folgt.

$$
\begin{array}{ll}
0^\dagger & := \varnothing \\
a^\dagger & := \{\varepsilon, a\} \\
\varepsilon^\dagger & := \{\varepsilon\} \\
(s \cup t)^\dagger & := s^\dagger \cup t^\dagger \\
(st)^\dagger & := \{su : u \in t^\dagger\} \cup s^\dagger \\
(s^*)^\dagger & := s^\dagger \cup \{s^* u : u \in s^\dagger\}
\end{array} \tag{202}
$$

Schließlich setzen wir

$$
s^p := \bigcup_{t \in s^\dagger} t \tag{203}
$$

Lemma 13.13 *Genau dann ist $\vec{x}$ ein Präfix einer Zeichenkette von s, wenn $\vec{x} \in L(t)$ für ein $t \in s^\dagger$. Also ist*

$$
L(s)^p = L(s^p) \quad (= \bigcup_{t \in s^\dagger} L(t)) \tag{204}
$$

Beweis. Der Beweis ist über Induktion über den Aufbau von s. Die Fälle $s = \varnothing$ und $s = \varepsilon$ sind leicht. Nun sei $s = a$ mit $a \in A$. Dann ist $t = a$ oder $t = \varepsilon$ und die Behauptung ist leicht zu sehen.

Nun sei $s = s_1 \cup s_2$.

$$\begin{aligned} L(s)^p &= L(s_1 \cup s_2)^p \\ &= L(s_1)^p \cup L(s_2)^p \\ &= L(s_1^p) \cup L(s_2^p) \\ &= \textstyle\bigcup_{t \in s_1^\dagger} L(t) \cup \bigcup_{t \in s_2^\dagger} L(t) \\ &= \textstyle\bigcup_{t \in s_1^\dagger \cup s_2^\dagger} L(t) \\ &= \textstyle\bigcup_{t \in s^\dagger} L(t) \\ &= L(s^p) \end{aligned} \tag{205}$$

Nun sei $s = s_1 \cdot s_2$. Zunächst gilt $L(s)^p = L(s_1)^p \cup L(s_1)L(s_2)^p$.

$$\begin{aligned} L(s)^p &= L(s_1)^p \cup L(s_1)L(s_2)^p \\ &= L(s_1^p) \cup L(s_1)L(s_2^p) \\ &= \textstyle\bigcup_{t \in s_1^\dagger} L(t) \cup L(s_1) \left(\bigcup_{t \in s_2^\dagger} L(t) \right) \\ &= \textstyle\bigcup_{t \in s_1^\dagger} L(t) \cup \left(\bigcup_{t \in s_1 s_2^\dagger} L(t) \right) \\ &= \textstyle\bigcup_{t \in s_1^\dagger \cup s_1 s_2^\dagger} L(t) \\ &= \textstyle\bigcup_{t \in s^\dagger} L(t) \\ &= L(s^p) \end{aligned} \tag{206}$$

Schließlich sei $s = s_1^*$. Wir benützen die Gleichung $L(s_1^*)^p = L(s_1)^p \cup L(s_1^*)L(s_1)^p$.

$$\begin{aligned} L(s)^p &= L(s_1^*)^p \cup L(s_1)L(s_1^*)^p \\ &= L(s_1^p) \cup L(s_1^*)L(s_1)^p \\ &= \textstyle\bigcup_{t \in s_1^\dagger} L(t) \cup L(s_1^*) \left(\bigcup_{t \in s_1^\dagger} L(t) \right) \\ &= \textstyle\bigcup_{t \in s_1^\dagger} L(t) \cup \left(\bigcup_{t \in s_1^* s_1^\dagger} L(t) \right) \\ &= \textstyle\bigcup_{t \in s_1^\dagger \cup s_1^* s_2^\dagger} L(t) \\ &= \textstyle\bigcup_{t \in s^\dagger} L(t) \\ &= L(s^p) \end{aligned} \tag{207}$$

Dies beendet den Beweis. ⊣

Es sei hier kurz erwähnt, dass für alle $s \neq \varnothing$ gilt $\varepsilon \in s^\dagger$; ebenso ist natürlich $s \in s^\dagger$. Beides lässt sich leicht durch Induktion zeigen.

Sei nun s ein regulärer Ausdruck. Wir definieren einen Automaten $\mathfrak{A}(s)$ wie folgt. Die Zustände seien die Terme aus $s^\dagger$. Zwischen ihnen definieren wir folgende Übergänge: $t \xrightarrow{a} u$ genau dann, wenn $ta \subseteq u$. (Dies bedeutet, dass $L(ta) \subseteq L(u)$,

welches wir auch mit $ta \subseteq u$ bezeichnen.) Das Startsymbol ist ε, und es gibt einen einzigen akzeptierenden Zustand, s.

$$
\begin{aligned}
A(s) &:= \{a \in A : a \in s^\dagger\} \\
\mathfrak{A}(s) &:= \langle A(s), s^\dagger, \varepsilon, \{s\}, \to\rangle
\end{aligned}
\tag{208}
$$

Dies nennen wir den **kanonischen Atomaten** von s. Wir werden nun zeigen, dass dieser genau s erkennt. Dies folgt aus einem allgemeineren Sachverhalt, der sogar leichter zu eigen ist.

Lemma 13.14 *In $\mathfrak{A}(s)$ gilt $\varepsilon \stackrel{\vec{x}}{\to} t$ genau dann, wenn $\vec{x} \in L(t)$. Das heißt, dass die Sprache, die im Zustand t akzeptiert wird, genau $L(t)$ ist.*

Beweis. Zunächst zeigen wir, dass wenn $\varepsilon \stackrel{\vec{x}}{\to} u$ dann $\vec{x} \in L(u)$. Der Beweis ist über Induktion nach der Länge von $\vec{x}$. Ist $\vec{x} = \varepsilon$, so ist die Behauptung gewiss richtig. Nun sei $\vec{x} = \vec{y}a$ für ein a und ein $\vec{y}$. Sei $\varepsilon \stackrel{\vec{x}}{\to} u$. Dann gibt es ein t mit $\varepsilon \stackrel{\vec{y}}{\to} t \stackrel{a}{\to} u$. Nach Induktionsannahme ist $\vec{y} \in L(t)$, und nach Definition von $\to$ $L(t)a \subseteq L(u)$. Daraus folgt die Behauptung. Die Umkehrung zeigen auch durch Induktion nach der Länge von $\vec{x}$. Angenommen, $\vec{x} \in L(u)$. Dann zeigen wir $\varepsilon \stackrel{\vec{x}}{\to} u$. Ist $\vec{x} = \varepsilon$, so sind wir fertig. Nun sei $\vec{x} = \vec{y}a$ für ein $a \in A$ und $\vec{y}$. Wr müssen zeigen, dass es ein $t \in s^\dagger$ gibt mit $ta \subseteq u$. Denn dann gilt nach Induktionsannahme $\varepsilon \stackrel{\vec{y}}{\to} t$, und so $\varepsilon \stackrel{\vec{y}a}{\to} u$, gemäß Definition von $\to$.

Nun also zu der fehlenden Behauptung, dass, falls $\vec{y}a \in L(u)$, so gibt es ein $t \in s^\dagger$ derart, dass $\vec{y} \in L(t)$ und $L(ta) \subseteq L(u)$. Diesmal machen wir Induktion über u. Man beachte, dass $u^\dagger \subseteq s^\dagger$. Das wird noch nützlich werden. Fall 1. $u = b$ für ein $b \in A$. Dann ist $\varepsilon \in s^\dagger$, und $t := \varepsilon$ tut das Gewünschte. Fall 2. $u = u_1 \cup u_2$. Dann sind u_1 und u_2 beide in $s^\dagger$. Nun sei $\vec{y}a \in L(u_1)$. Nach Induktionsannahme gibt es ein t mit $L(ta) \subseteq L(u_1) \subseteq L(u)$, und daraus folgt die Behauptung. Ähnlich im Fall $\vec{y}a \in u_2$. Fall 3. $u = u_1u_2$. Unterfall 3a. $\vec{y} = \vec{y}_1\vec{y}_2$, mit $\vec{y}_1 \in L(u_1)$ und $\vec{y}_2 \in L(u_2)$, $\vec{y}_2 \neq \varepsilon$. Nach Induktionsannahme gibt es t_2 mit $L(t_2a) \subseteq L(u_2)$. Dann ist $t := u_1t_2$ der gewünschte Term. Da $t \in u^\dagger$, so ist auch $t \in s^\dagger$. Und $L(ta) = L(u_1t_2a) \subseteq L(u_1u_2) = L(u)$. Unterfall 3b. $\vec{y} \in L(u_1)$ (weil $\varepsilon \in L(u_2)$). Diesen Fall muss man nicht weiter betrachten; er ist schon durch die Induktionsannahme gedeckt. Fall 4. $u = u_1^*$. Sei $\vec{y}a \in L(u)$. Dann hat $\vec{y}$ eine Zerlegung $\vec{z}_0\vec{z}_1 \cdots \vec{z}_{n-1}\vec{v}$ derart, dass $\vec{z}_i \in L(u_1)$ für alle $i < n$, und $\vec{v}a \in L(u_1)$. Nach Induktionsannahme gibt es ein t_1

mit $L(t_1 a) \subseteq L(u_1)$ und $\vec{v} \in L(t_1)$. Ferner ist $\vec{z}_0 \vec{z}_1 \cdots \vec{z}_{n-1} \in L(u)$. Setze $t := ut_1$. Dies hat die gewünschet Eigenschaft. ⊣

Proposition 13.15 *Sei L regulär. Dann ist auch L^p regulär.*

Zum Beweis sei der Automat $\langle A(s), s^\dagger, \varepsilon, s^\dagger, \to \rangle$ genommen. Dieser akzeptiert alle Zeichenketten $\vec{x}$ für die ein $t \in s^\dagger$ existiert mit $\vec{x} \in L(t)$. Nach dem eben Gesagten sind das alle Zeichenketten aus $L(s)^p$.

14 Konstruktion eines Automaten aus einem Regulären Term

Man kann auch mittels eines regulären Terms direkt eine Zeichenkette verarbeiten. Ich mache das mit einem Beispiel vor. Für die Zwecke dieses Kapitels benutze ich eine etwas vereinfachte Syntax, indem ich auf Escape-Sequenzen verzichte. Klammern und Hilfszeichen werden jetzt von Buchstaben getrennt. Das vereinfacht die Schreibung und verändert nichts am Prinzip. Ich habe das Verfahren implementiert. Dabei benutze ich `ocamllex` und `ocamlyacc` zum Einlesen. Das Programm besteht deswegen aus mehreren Teilen, siehe die Kurswebseite zur Installation.

Es sei der Term `(a|(b|c)*)b` gegeben. Wir stellen uns vor, dass wir uns durch den Term von links nach rechts durchhangeln. Wir arbeiten den Term von links nach rechts ab, indem wir Symbol nach Symbol einlesen. Dazu erlaube ich, einen Buchstaben zu unterstreichen. Die Unterstreichung soll bedeuten, dass wir uns gerade an dieser Stelle im Term befinden. Unterstrichen werden nur Buchstaben, keine Sonderzeichen. Ich symbolisiere Unterstreichen hier mit Rot. Am Anfang befinden wir uns noch nicht im Term, dann unterstreichen wir nichts. Das gibt uns folgende gefärbten Terme:

(209) `(a|(b|c)*)b`, `(a|(b|c)*)b`, `(a|(b|c)*)b`, `(a|(b|c)*)b`, `(a|(b|c)*)b`

Definition 14.1 (Unifarbterm) *Ein **Unifarbterm** ist ein Ausdruck der folgenden Gestalt:*

① *a, wo a ein Buchstabe ist.*

② s^**, wo s ein Unifarbterm ist.*

③ *st oder* $s \cup t$*, wo entweder s ein Unifarbterm und t ein regulärer Term ist oder s ein regulärer Term und t ein Unifarbterm.*

Wir beginnen mit dem ersten Term. Wir lesen ein Symbol ein. Ist es `a`, so springen wir zu `(a|(b|c)*)b`. Wie kommt es dazu? Das unterstrichene Symbol ist

dasjenige, welches wir gerade gelesen haben. Zusätzlich ist es in der Sprache des Terms auch tatsächlich ein Buchstabe, der ganz am Anfang auftreten kann, nämlich in `ab`. Lesen wir anschließend ein `b` ein, dann gehen wir von diesem Term über zu `(a|(b|c)*)b`. Wir akzeptieren diese Kette, weil der gefärbte Term final ist.

Definition 14.2 (Finaler Unifarbterm) *Ein Unifarbterm t heißt **final**, wenn Folgendes gilt:*

① *$t = a$, wo a ein Buchstabe ist.*

② *$t = uv$ und u ist ein finaler Unifarbterm, v ein regulärer Term mit $\varepsilon \in L(v)$.*

③ *$t = uv$ und u ist ein regulärer Term, v ein finaler Unifarbterm.*

④ *$t = u \cup v$ und u oder v ist ein finaler Unifarbterm.*

⑤ *$t = u^*$ und u ist finaler Unifarbterm.*

Es gibt noch eine Extraklausel. Wir akzeptieren das leere Wort, wenn der Term das Wort ε enthält. Dies lässt sich rein syntaktisch feststellen. Zunächst einmal lässt sich 0 aus jedem Term eliminieren, falls der Term nicht zu 0 reduziert. ($0 \cup s = s \cup 0 = s$, $s0 = 0s = 0$, $0^* = 0$.)

Proposition 14.3 *Sei $L(t) \neq 0$. Genau dann ist $\varepsilon \in L(s)$, wenn*

① *$s = \varepsilon$; oder*

② *$s = t^*$; oder*

③ *$s = t \cup u$ und (a) $\varepsilon \in L(t)$ oder (b) $\varepsilon \in L(u)$; oder*

④ *$s = tu$ und $\varepsilon \in L(t)$ und $\varepsilon \in L(u)$.*

Schauen wir uns nun an, was passiert, wenn der erste Buchstabe `b` ist. Dann gehen wir zu `(a|(b|c)*)b`, weil ja eine Zeichenkette auch mit `b` anfangen kann. Andererseits könnten wir auch zu `(a|(b|c)*)b` gehen. Da wir dies nicht entscheiden können, erlauben wir nunmehr das Unterstreichen aller Buchstaben gleichzeitig. Gleichzeitig sind Buchstaben unterstrichen, wenn wir nicht wissen, wo wir sind,

oder genauer, wenn wir sowohl an der einen wie an der anderen Stelle sein könnten. Es steht jetzt `(a|(b|c)*)b` kurz für die Menge {`(a|(b|c)*)b`, `(a|(b|c)*)b`}. Da wir immer nur ein Symbol verarbeiten, werden immer nur Vorkommen ein und desselben Buchstabens unterstrichen sein.

Kommen wir nun zu der Frage, wie wir von einem Zustand zum nächsten kommen. Dazu die folgende Definition.

Definition 14.4 (Initialer Unifarbterm) *v ist a-**initial** falls eine der folgenden Bedingungen zutrifft.*

① $v = a$,

② $v = ww'$, *und w ist a-initial.*

③ $v = ww'$, $\varepsilon \in L(w)$ *und w' ist a-initial.*

④ $v = w \cup w'$, *w ist gefärbt und w ist a-initial.*

⑤ $v = w \cup w'$, *w' ist gefärbt und w' ist a-initial.*

⑥ $v = w^*$, *w ist a-initial.*

Es seien s und s' Färbungen desselben Terms u. Es existiert ein a-Übergang von s nach s', falls gilt:

① $s = vw$, $s' = v'w$, *v und v' sind Unifarbterme, und es existiert ein a-Übergang von v nach v'.*

② $s = vw$, $s' = vw'$, *w und w' sind Unifarbterme, und es existiert ein a-Übergang von w nach w'.*

③ $s = vw$, $s' = v'w'$, *v, w' sind Unifarbterme, v ist final und w' ist a-initial.*

④ $s = v^*$, $s' = v'^*$, *und*

- *es existiert ein a-Übergang von v nach v' oder*
- *v ist final und v' ist a-initial.*

Damit bekommen wir folgende Übergänge:

(210)
$$
\begin{aligned}
\texttt{(a|(b|c)*)b} &\xrightarrow{\texttt{a}} \texttt{(a|(b|c)*)b}\\
\texttt{(a|(b|c)*)b} &\xrightarrow{\texttt{b}} \texttt{(a|(b|c)*)b}\\
\texttt{(a|(b|c)*)b} &\xrightarrow{\texttt{b}} \texttt{(a|(b|c)*)b}\\
\texttt{(a|(b|c)*)b} &\xrightarrow{\texttt{b}} \texttt{(a|(b|c)*)b}\\
\texttt{(a|(b|c)*)b} &\xrightarrow{\texttt{b}} \texttt{(a|(b|c)*)b}\\
\texttt{(a|(b|c)*)b} &\xrightarrow{\texttt{c}} \texttt{(a|(b|c)*)b}\\
\texttt{(a|(b|c)*)b} &\xrightarrow{\texttt{b}} \texttt{(a|(b|c)*)b}\\
\texttt{(a|(b|c)*)b} &\xrightarrow{\texttt{b}} \texttt{(a|(b|c)*)b}\\
\texttt{(a|(b|c)*)b} &\xrightarrow{\texttt{c}} \texttt{(a|(b|c)*)b}\\
\texttt{(a|(b|c)*)b} &\xrightarrow{\texttt{b}} \texttt{(a|(b|c)*)b}
\end{aligned}
$$

Wir sehen, dass dieser Automat nichtdeterministisch ist. Wie oben schon angedeutet, erlauben wir jetzt das Färben belieger Buchstaben. Wir nennen so etwas einen **Multifarbterm**. Ein Multifarbterm steht wie angedeutet für die Menge aller Unifarbterme, die durch Weglassen von Färbungen aus ihm entstehen. Dabei müssen wir beachten, dass wir auch gar keinen Buchstaben färben können. Das entspricht dann der leeren Menge. Der Startzustand ist hierbei extra zu nehmen. Ich benutze ein Extrasymbol, sagen wir ♡. Wir haben jetzt von den in (209) genannten Unifarbtermen 2^4 Multifarbterme und noch den Zustand ♡. Dabei verrate ich schon jetzt, dass man Multifarbterme mit ungleichen markierten Buchstaben nicht erreichen kann, sodass nur noch folgende Liste übrigbleibt.

(211) ♡, `(a|(b|c)*)b`, `(a|(b|c)*)b`, `(a|(b|c)*)b`,
`(a|(b|c)*)b`, `(a|(b|c)*)b`, `(a|(b|c)*)b`

Wir basteln jetzt eine deterministische Übergangsrelation, die in Tabelle 4 gezeigt ist. Als Letztes gilt es noch zu klären, welche Zustände akzeptierend sind. Dies sind

(212) `(a|(b|c)*)b`, `(a|(b|c)*)b`

Denn nur das letzte Vorkommen von `b` ist am Ende des Terms, und es muss folglich in einem akzeptierenden Zustand markiert sein.

Abbildung 4: Die Übergangsrelation zwischen Multifarbtermen

♡ $\xrightarrow{a}$ (**a**|(b|c)*)b
♡ $\xrightarrow{b}$ (a|(**b**|c)*)**b**
♡ $\xrightarrow{c}$ (a|(b|**c**)*)b
(**a**|(b|c)*)b $\xrightarrow{a}$ (a|(b|c)*)b
(**a**|(b|c)*)b $\xrightarrow{b}$ (a|(b|c)*)**b**
(**a**|(b|c)*)b $\xrightarrow{c}$ (a|(b|c)*)b
(a|(**b**|c)*)b $\xrightarrow{a}$ (a|(b|c)*)b
(a|(**b**|c)*)b $\xrightarrow{b}$ (a|(**b**|c)*)**b**
(a|(**b**|c)*)b $\xrightarrow{c}$ (a|(b|**c**)*)b
(a|(b|c)*)**b** $\xrightarrow{a}$ (a|(b|c)*)b
(a|(b|c)*)**b** $\xrightarrow{b}$ (a|(b|c)*)b
(a|(b|c)*)**b** $\xrightarrow{c}$ (a|(b|c)*)b
(a|(**b**|c)*)**b** $\xrightarrow{a}$ (a|(b|c)*)b
(a|(**b**|c)*)**b** $\xrightarrow{b}$ (a|(**b**|c)*)**b**
(a|(**b**|c)*)**b** $\xrightarrow{c}$ (a|(b|**c**)*)b
(a|(b|**c**)*)b $\xrightarrow{a}$ (a|(b|c)*)b
(a|(b|**c**)*)b $\xrightarrow{b}$ (a|(**b**|c)*)**b**
(a|(b|**c**)*)b $\xrightarrow{c}$ (a|(b|**c**)*)b

15 Minimale Automaten

In diesem Abschnitt wollen wir uns einmal mit dem Problem auseinandersetzen, wie man mit einem Automaten nun erkennen kann, ob eine Zeichenkette akzeptiert ist oder nicht. Dabei soll es auch darum gehen, wie man durch geschickte Arbeit überflüssige Schritte eliminieren kann. Denn wir werden sehen, dass selbst der schnellste Computer mit einfachen Problemen lahmgelegt werden kann, wenn nur der Algorithmus umständlich genug ist.

Sei also ein Automat $\mathfrak{A}$ und eine Zeichenkette $\vec{x} = x_0 x_1 \dots x_{n-1}$ gegeben. Wie lange dauert es, bis wir wissen, ob $\mathfrak{A}$ akzeptiert oder nicht, das heißt, ob $\vec{x} \in L(\mathfrak{A})$? Die Antwort wird naturgemäß sowohl von $\mathfrak{A}$ als auch $\vec{x}$ abhängen. Nach Definition haben wir $\vec{x} \in L(\mathfrak{A})$ dann und nur dann, wenn es Zustände q_i, $i < n + 1$, gibt mit

$$i_0 = q_0 \xrightarrow{x_0} q_1 \xrightarrow{x_1} q_2 \xrightarrow{x_2} q_2 \dots \xrightarrow{x_{n-1}} q_n \in F \tag{213}$$

Zu entscheiden, ob nun $q_i \xrightarrow{x_i} q_{i+1}$ oder nicht, ist ein einzelner Schritt und kostet konstant viel Zeit. Wir schauen einfach nach, ob $\langle q_i, x_i, q_{i+1} \rangle \in \delta$.

Da nun $\vec{x}$ gegeben ist, haben wir auch seine Länge n. Also können wir ja einfach alle möglichen Folgen von q_i durchgehen und (213) überprüfen. Hat man m Zustände, würden dies immerhin m^n Folgen sein. Dieser Algorithmus ist sehr langsam! (Genauer: statt m steht unten die maximale Zahl p derart, dass ein Zustand mit einem Buchstaben p verschiedene Übergänge erlaubt.) Außerdem müssten wir die Zeichenkette immer und immer wieder durchgehen. Wir hätten gerne einen Algorithmus, bei dem wir nur einmal durch die Zeichenkette gehen und dann am Ende ein Ergebnis haben.

Das ist in der Tat möglich. Dazu tun wir Folgendes. Wir beginnen am Anfang der Zeichenkette und notieren als Ausgangsmenge die Menge $\{i_0\}$. Im ersten Schritt sammeln wir alle Zustände, die man von i_0 aus mit x_0 erreichen kann. Diese Menge heiße H_1. Im zweiten Schritt suchen wir alle Zustände, die man von H_1 aus mit x_1 erreichen kann. Diese Menge heiße H_2. Im Schritt 3 sammeln wir alle Zustände, die man von H_1 mit x_2 erreichen kann, und so weiter. Wir haben dann

$$i_0 \xrightarrow{x_0 x_1 \dots x_{j-1}} q_j \text{ gdw. } q_j \in H_j \tag{214}$$

Also ist $\vec{x} \in L(\mathfrak{A})$ genau dann, wenn $H_n \cap F \neq \emptyset$. Denn dann existiert ein akzeptierender Zustand in H_n.

Hier ein Beispiel. Der Automat sei $\langle\{\mathsf{a},\mathsf{b},\mathsf{c}\},\{0,1,2\},0,\{2\},\delta\rangle$, wo

$$\begin{aligned}\delta = \{\ & \langle 0,\mathsf{a},0\rangle,\langle 0,\mathsf{b},1\rangle,\langle 0,\mathsf{b},2\rangle,\langle 1,\mathsf{b},1\rangle,\\ & \langle 1,\mathsf{c},2\rangle,\langle 2,\mathsf{c},0\rangle\}\end{aligned} \tag{215}$$

Gegeben sei die Zeichenkette abbc. Wir beginnen mit $H_0 := \{0\}$. Das erste Symbol ist a. Mittels a kommen wir von 0 aus lediglich nach 0, denn ist $\langle 0,\mathsf{a},q\rangle \in \delta$, so ist $q = 0$. Also ist $H_1 = \{0\}$. Das zweite Symbol ist b. Mittels b kommen wir von 0 sowohl nach 1 wie auch nach 2. Also ist $H_2 = \{1,2\}$. Das dritte Symbol ist b. Es gibt keinen Übergang von 2 mit b, wohl aber einen Übergang von 1 mit b, und zwar nach 1. Also ist $H_3 = \{1\}$. Das nächste Symbol ist c. Damit kommt man von 1 nach 2. Der Automat akzeptiert die Zeichenkette.

Schauen wir uns an, wieviel Arbeit man damit hat, wenn $\mathfrak{A}$ m Zustände besitzt. Falls H irgendeine Menge von Zuständen ist, so hat H höchstens m Elemente. Die Nachfolgermenge zu bestimmen, kostet uns höchstens $m \times m \times |A|$ Schritte (man überprüfe einfach alle Übergänge $x \xrightarrow{a} y$ daraufhin, ob $x \in H$, und tue dann y in H'). Dieser Aufwand ist beschränkt; er hängt nicht von $\vec{x}$ ab. In der Tat können wir jetzt unsere Zeichenkette Schritt für Schritt durchgehen. Wir konstruieren H_j einfach, nachdem wir das Symbol x_{j-1} gelesen haben. Auf der anderen Seite ist es mühselig, diese Arbeit stets aufs Neue zu machen. Denn es gibt nur begrenzt viele Zustandsmengen, und wir müssten ja nur für jede einzelne Menge und jeden Buchstaben wissen, welches der nächste Zustand ist. Dies kann man tabellieren (dazu eignet sich eine Hashtabelle). Wenn wir dies tun, haben wie schon den Schritt zur Konstruktion von $\mathfrak{A}^d$ vollzogen. Wenn wir uns noch einmal die Definition von $\mathfrak{A}^d$ anschauen, stellen wir fest, dass dessen Übergangsrelation genau der oben berechneten Relation entspricht (welche im Übrigen eine Funktion ist, weil der Automat ja deterministisch ist). In der Tat ist das exponentielle Verhalten, das oben beschrieben wird, bei einem deterministischen Automaten problemlos, den jeder Zustand hat nur einen Nachfolger (in Abhängigkeit vom Buchstaben, der gerade gelesen wird).

Das Rezept ist also wie folgt: man nehme anstelle eines Automaten $\mathfrak{A}$ immer einen deterministischen Automaten (zum Beispiel $\mathfrak{A}^d$). In diesem Fall ist die Berechnung von '$\vec{x} \in L(\mathfrak{A})$?' in Echtzeit zu machen.

Nun ist die Größe des Automaten in der Regel kein Problem, und in der Komplexitätsberechnung unterschlägt man normalerweise Konstanten, aber praktisch gesehen ist die Rechnung langsamer, wenn der Automat größer ist. Es ist keine Kunst, riesige Automaten zu basteln; die Kunst ist, möglichst wenig Zustände

zu haben. Dazu, wie man mit wenig Zuständen auskommen kann, jetzt folgende Definition.

Definition 15.1 *Sei L eine Sprache. Gegegeben eine Zeichenkette $\vec{x}$, sei $[\vec{x}]_L = \{\vec{y} : \vec{x}\hat{}\vec{y} \in L\}$. Wir schreiben auch $\vec{x} \sim_L \vec{y}$, falls $[\vec{x}]_L = [\vec{y}]_L$. Der **Index** von L ist die Zahl der verschiedenen Mengen $[\vec{x}]_L$.*

Hier ist ein Beispiel. Die Sprache $L = \{\mathsf{ab}, \mathsf{ac}, \mathsf{bc}\}$ hat die folgenden Indexmengen:

$$
\begin{array}{lcl}
[\varepsilon]_L & = & \{\mathsf{ab}, \mathsf{ac}, \mathsf{bc}\} \\
[\mathsf{a}]_L & = & \{\mathsf{b}, \mathsf{c}\} \\
[\mathsf{b}]_L & = & \{\mathsf{c}\} \\
[\mathsf{c}]_L & = & \varnothing \\
[\mathsf{ab}]_L & = & \{\varepsilon\}
\end{array} \tag{216}
$$

Nehmen wir eine leicht andere Sprache, $M = \{\mathsf{ab}, \mathsf{ac}, \mathsf{bc}, \mathsf{bb}\}$.

$$
\begin{array}{lcl}
[\varepsilon]_M & = & \{\mathsf{ab}, \mathsf{ac}, \mathsf{bb}, \mathsf{bc}\} \\
[\mathsf{a}]_M & = & \{\mathsf{b}, \mathsf{c}\} \\
[\mathsf{c}]_M & = & \varnothing \\
[\mathsf{ab}]_M & = & \{\varepsilon\}
\end{array} \tag{217}
$$

Man rechnet leicht nach, dass $[\mathsf{b}]_M = [\mathsf{a}]_M$. Wir werden sehen, dass dieser Unterschied bedeutet, dass es einen Automaten gibt, der die Zugehörigkeit zu M mit weniger Zuständen prüfen kann wie die Zugehörigkeit zu L.

Seien nun I und J zwei Indexmengen. Dann setze $I \xrightarrow{a} J$ genau dann, wenn $J = a\backslash I$. Dies ist wohldefiniert. Denn sei $I = [\vec{x}]_L$ und seit $\vec{x}a$ das Präfix einer Zeichenkette von L. Dann ist $[\vec{x}a]_L = \{\vec{y} : \vec{x}a\vec{y} \in L\} = \{\vec{y} : a\vec{y} \in I\} = a\backslash I$. Dies definiert einen deterministischen Automaten mit Anfangszustand L $(= [\varepsilon]_L)$. Die akzeptierenden Zustände sind diejenigen Indexmengen, die ε enthalten. Wir nennen diesen Automaten den **Indexautomaten** und bezeichnen ihn mit $\mathfrak{I}(L)$. (Dieser wird auch der **Myhill-Nerode Automat** genannt.)

Satz 15.2 (Myhill-Nerode) *$L(\mathfrak{I}(L)) = L$.*

Beweis. Durch Induktion über die Länge von $\vec{x}$ zeigen wir, dass $L \xrightarrow{\vec{x}} I$ genau dann, wenn $[\vec{x}]_L = I$. Falls $\vec{x} = \varepsilon$, so ist die Behauptung gerade: $L = L$ genau

dann, wenn $[\varepsilon]_L = L$. Aber $[\varepsilon]_L = L$, und so ist dies richtig. Nun sei $\vec{x} = \vec{y}a$. Nach Induktionsannahme ist $L \xrightarrow{\vec{y}} J$ genau dann, wenn $[\vec{y}]_L = J$. Es ist $J \xrightarrow{a} I$ genau dann, wenn $I = a\backslash J = a\backslash[\vec{y}a]_L = [\vec{x}]_L$, wie versprochen.

$\vec{x}$ wird genau dann von $\mathfrak{I}(L)$ akzeptiert, wenn es eine Berechnung von L zu $[\vec{y}]_L$ gibt, welche ε enthält. Nach dem oben Gesagten ist dies äquivalent zu $\varepsilon \in [\vec{x}]_L$, und das ist genau $\vec{x} \in L$. ⊣

Der Indexautomat ist tatsächlich der kleinste Automat, der eine gegebene Sprache erkennt. Sei nämlich $\mathfrak{A}$ ein Automat. Dann setzen wir für einen Zustand q

$$[q] := \{\vec{x} : \text{ es gibt } q' \in F \colon q \xrightarrow{\vec{x}} q'\} \tag{218}$$

Es ist leicht zu sehen, dass für jeden Zustand q eine Zeichenkette $\vec{x}$ existieren muss mit $[q] \subseteq [\vec{x}]_L$. Sei nämlich $\vec{x}$ derart, dass $i_0 \xrightarrow{\vec{x}} q$. Dann gilt für alle $\vec{y} \in [q]$, dass $\vec{x}\vec{y} \in L(\mathfrak{A})$, nach Definition von $[q]$. Also ist $[q] \subseteq [\vec{x}]_L$. Umgekehrt muss es für jedes $[\vec{x}]_L$ einen Zustand q geben derart, dass $[q] \subseteq [\vec{x}]_L$. Wiederum finden wir q als einen Zustand derart, dass $i_0 \xrightarrow{\vec{x}} q$. Es sei nun $\mathfrak{A}$ deterministisch und total. Dann gibt es für jede Zeichenkette $\vec{x}$ genau einen Zustand $[q]$ mit $[q] \subseteq [\vec{x}]_L$. Offensichtlich ist dann $[q] = [\vec{x}]_L$. Denn falls $\vec{y} \in [\vec{x}]_L$, so ist $\vec{x}\vec{y} \in L$, woher $i_0 \xrightarrow{\vec{x}\vec{y}} q' \in F$ für ein q'. Da der Automat deterministisch ist, ist $i_0 \xrightarrow{\vec{x}} q \xrightarrow{\vec{y}} q'$, und so $\vec{y} \in [q]$.

Daraus folgt die Behauptung, dass der Indexautomat der kleinste deterministische undd totale Automat ist. Wir haben nun schon gesehen, wie der Automat definiert ist. Aber dies ist keine Konstruktion, die wir wirklich durchführen können. Deswegen sei hier ein Konstruktionsverfahren beschrieben. Das eine startet mit einem Automaten und konstruiert einen kleineren Automaten, das andere beginnt beim regulären Ausdruck.

Sei also $\mathfrak{A}$ ein Automat. Eine Relation $\sim$ heiße ein **Netz**, falls gilt:

❶ aus $q \sim q'$ und $q \xrightarrow{a} r$, $q' \xrightarrow{a} r'$ folgt $q' \sim r'$, und

❷ wenn $q \in F$ und $q \sim q'$, dann auch $q' \in F$.

Ein Netz induziert eine Partition der Zustandsmenge in Mengen der Form $[q]_\sim = \{q' : q' \sim q\}$. Im allgemeinen heißt **Partition** einer Menge Q eine Menge Π von

nichtleeren Teilmengen von Q derart, dass jedes Element von Q in genau einer Menge von Π liegt.

Gegeben ein Netz $\sim$ sei

$$
\begin{aligned}
[q]_\sim &:= \{q' : q \sim q'\} \\
Q/\sim &:= \{[q]_\sim : q \in Q\} \\
F/\sim &:= \{[q]_\sim : q \in F\} \\
\delta/\sim &:= \{\langle [q']_\sim, a, [q]_\sim\rangle : \langle q', a, q\rangle \in \delta\} \\
\mathfrak{A}/\sim &:= \langle A, Q/\sim, [i_0]_\sim, F/\sim, \delta/\sim\rangle
\end{aligned} \tag{219}
$$

Lemma 15.3 $L(\mathfrak{A}/\sim) = L(\mathfrak{A})$.

Beweis. Durch Induktion über die Länge von $\vec{x}$ kann man Folgendes zeigen: wenn $q \sim q'$ und $q \xrightarrow{\vec{x}} r$, dann gibt es ein $r' \sim r$ derart, dass $q' \xrightarrow{\vec{x}} r'$. Nun sei $\vec{x} \in L(\mathfrak{A}/\sim)$. Dies bedeutet, dass es ein $[q]_\sim \in F/\sim$ gibt mit $[i_0]_\sim \xrightarrow{\vec{x}} [q]_\sim$. Dies wiederum bedeutet, dass es $q' \sim q$ gibt mit $i_0 \xrightarrow{\vec{x}} q'$. Da nun $q \in F$ so ist auch $q' \in F$, nach Definition von Netzen. Also ist $\vec{x} \in L(\mathfrak{A})$. Umgekehrt, sei $\vec{x} \in L(\mathfrak{A})$. Dann ist $i_0 \xrightarrow{\vec{x}} q$ für ein $q \in F$. Also ist $[i_0]_\sim \xrightarrow{\vec{x}} [q]_\sim$, wie eine leichte Induktion zeigt. Nach Definition von $\mathfrak{A}/\sim$ ist $[q]_\sim$ ein akzeptierender Zustand. Also ist $\vec{x} \in L(\mathfrak{A}/\sim)$. $\dashv$

Alles, was wir tun müssen, ist, die Zustände in einer Partitionsmenge zusammenzufassen. Also müssen wir nur das gröbste Netz auf $\mathfrak{A}$ finden. Dieses finden wir so. Im ersten Schritt setzen wir $q \sim_0 q'$ genau dann, wenn $q, q' \in F$ oder $q, q' \in Q - F$. Dies muss noch kein Netz sein. Induktiv definieren wir wie folgt:

$$
\begin{aligned}
q \sim_{i+1} q' :\Leftrightarrow\ & q \sim_i q' \text{ und für alle } a \in A, \text{ und alle } r \in Q: \\
& \text{falls } q \xrightarrow{a} r, \text{ so existiert ein } r' \sim_i r \text{ mit } q' \xrightarrow{a} r' \\
& \text{falls } q' \xrightarrow{a} r, \text{ so existiert ein } r' \sim_i r \text{ mit } q' \xrightarrow{a} r
\end{aligned} \tag{220}
$$

Es ist klar, dass $\sim_{i+1} \subseteq \sim_i$. Auch ist, falls $q \sim_{i+1} q'$ und $q \in F$ auch $q' \in F$, weil dies schon für $\sim_0$ gilt. Endlich, wenn $\sim_{i+1} = \sim_i$, so ist $\sim_i$ ein Netz. Dies legt folgendes Verfahren nahe. Wir beginnen mit $\sim_0$ und konstruieren $\sim_i$ Schritt für Schritt. Falls $\sim_{i+1} = \sim_i$, so ist die Konstruktion fertig. Offensichtlich ist dies nach endlich vielen Schritten erreicht.

Definition 15.4 *Sei $\mathfrak{A}$ ein endlicher Automat. $\mathfrak{A}$ heißt **verfeinert** wenn nur die Identität ein Netz auf $\mathfrak{A}$ ist.*

Satz 15.5 *Es seien $\mathfrak{A}$ und $\mathfrak{B}$ deterministische, totale und verfeinerte Automaten derart, dass jeder Zustand erreichbar ist. Ist dann $L(\mathfrak{A}) = L(\mathfrak{B})$, so sind die Automaten isomorph.*

Beweis. Es habe $\mathfrak{A}$ die Zustandsmenge $Q^{\mathfrak{A}}$ und $\mathfrak{B}$ die Zustandsmenge $Q^{\mathfrak{B}}$. Für $q \in Q^{\mathfrak{A}}$ sei $I(q) := \{\vec{x} : q \xrightarrow{\vec{x}} r \in F\}$, und ähnlich für $q \in Q^{\mathfrak{B}}$. Sicher haben wir nach Annahme $I(i_0^{\mathfrak{A}}) = I(i_0^{\mathfrak{B}})$. Sei nun $q \in Q^{\mathfrak{A}}$ und $q \xrightarrow{a} r$. Dann ist $I(r) = a\backslash I(q)$. Deswegen ist, falls $q' \in Q^{\mathfrak{B}}$, $q' \xrightarrow{a} r'$ und $I(q) = I(q')$ auch $I(r) = I(r')$. Nun konstruieren wir eine Abbildung $h : Q^{\mathfrak{A}} \to Q^{\mathfrak{B}}$ wie folgt. $h(i_0^{\mathfrak{A}}) := i_0^{\mathfrak{B}}$. Falls $h(q) = q'$, $q \xrightarrow{a} r$ und $q' \xrightarrow{a} r'$ dann $h(r) := r'$. Da alle Zustände von $\mathfrak{A}$ erreichbar sind, ist h auf allen Zuständen definiert. Diese Abbildung ist injektiv, denn $I(h(q)) = I(q)$ und $\mathfrak{A}$ ist verfeinert. (Jeder Homomorphismus induziert ein Netz, sodass, wenn die Identität das einzige Netz ist, so ist h injektiv.) h ist surjektiv, da alle Zustände von $\mathfrak{B}$ erreichbar sind und $\mathfrak{B}$ verfeinert. ⊣

Das eine Rezept ist also wie folgt: man besorge sich einen deterministischen, totalen Automaten L und verfeinere ihn. Dies liefert einen Automaten, der bis auf Umbenennung der Zustände eindeutig ist.

Das andere Rezept ist dual zu Lemma 13.13. Setze

$$\begin{aligned} 0^\ddagger &:= \varnothing \\ a^\ddagger &:= \{\varepsilon, a\} \\ \varepsilon^\ddagger &:= \{\varepsilon\} \\ (s \cup t)^\ddagger &:= s^\ddagger \cup t^\ddagger \\ (st)^\ddagger &:= \{ut : u \in s^\ddagger\} \cup t^\ddagger \\ (s^*)^\ddagger &:= s^\ddagger \cup \{us^* : u \in s^\ddagger\} \end{aligned} \tag{221}$$

Es sei ein regulärer Ausdruck s gegeben. Dann können wir effektiv die Mengen $[\vec{x}]_L$ berechnen. Diese sind entweder $\varnothing$ oder von der Form $L(t)$ für ein $t \in s^\ddagger$. Der Startzustand ist s, und die akzeptierenden Zustände haben die Form $t \in s^\ddagger$, wo $\varepsilon \in t$. Man beachte aber, dass es a priori nicht klar ist, ob zwei gegebene Ausdrücke t, u die gleiche Sprache haben, ds heißt, ob $L(t) = L(u)$. Also müssen wir wissen, ob wir effektiv entscheiden können, ob $t = u$. Hier ist ein Rezept. Konstruiere einen Automaten $\mathfrak{A}$ derart, dass $L(\mathfrak{A}) = L(t)$ und einen Automaten $\mathfrak{B}$

mit $L(\mathfrak{B}) = A^* - L(u)$. Wir können annehmen, dass diese deterministisch sind. $L(\mathfrak{A} \times \mathfrak{B}) = L(t) \cap (A^* - L(u))$. Dies ist genau dann leer, wenn $L(t) \subseteq L(u)$. Dual dazu können wir einen Automaten konstruieren, der $L(u) \cap (A^* - L(t))$ erkennt, was genau dann leer ist, wenn $L(u) \subseteq L(t)$. Also müssen wir letztlich nur entscheiden könne, ob für einen Automaten $\mathfrak{A}$ $L(\mathfrak{A})$ leer ist. Dies aber ist wir folgt entscheidbar. Dazu definieren wir folgende Mengen. $M_0 := F$, $M_{n+1} := M_0 \cup \{q :$ existiert $q' \in M_n$ und a mit $q \xrightarrow{a} q'\}$. Dann ist $M_0 \subseteq M_1 \subseteq M_2 \subseteq \cdots$. Genau dann ist $L(\mathfrak{A}) \neq \emptyset$, wenn es ein n gibt mit $i_0 \in M_n$.

Satz 15.6 *Die folgenden Probleme sind entscheidbar für gegebene reguläre Ausdrücke t und u: 'L(t) = ∅', 'L(t) ⊆ L(u)' und 'L(t) = L(u)'.* ⊣

Es folgt nun daraus, dass die Indexmaschine effektiv konstruiert werden kann. Dies geht wie folgt. Von s aus konstruieren wir einen Automaten für die $t \in s^\ddagger$. Als nächstes berechnen wir für gegebenes t', ob die Äquivalenz $L(t) = L(t')$ für ein $t \neq t'$ gilt. Dann lassen wit t' weg und behalten nur t. Setze $s \xrightarrow{a} t$ falls $L(t) = a\backslash L(s)$.

Satz 15.7 *Genau dann L ist regulär, wenn es nur endlich viele Indexmengen hat.*

16 Ein Wenig über Komplexität

Ich werde in diesem Abschnitt ein paar Dinge zu Komplexität sagen. Dabei soll es lediglich um Zeitkomplexität gehen; das bedeutet, wir werden fragen, wie lange es dauert, eine gewisse Funktion zu berechnen, etwa, eine Liste zu sortieren oder eine Datenabfrage zu beantworten. Ist $f : M \to N$ die zu berechnende Funktion, so ordnet man den Elementen von M eine Größe zu und berechnet den Zeitaufwand in Abhängigkeit von dieser Größe. Ist etwa $M = A^*$, so können wir als Größe von $\vec{x}$ die Länge von $\vec{x}$ nehmen. Meist ist die zentrale Frage, wie schnell ein Algorithmus im schlechtesten Fall rechnet. Gegeben ein Algorithmus Γ lässt sich die Frage relativ leicht beantworten, wieviel Zeit Γ für einen gegebenen Input x benötigt. Sei dies $\zeta_\Gamma(x)$. Ist γ die Größenfunktion auf M, so ist die Zeitkomplexität von Γ

$$f_{\Gamma;\gamma}(n) := \max\{\zeta_\Gamma(x) : \gamma(x) = n\} \tag{222}$$

Desweiteren möchte man natürlich wissen, ob sich das Problem vielleicht schneller berechnen lässt, das heißt, ob es ein Δ gibt mit $f_{\Delta;\gamma}(n) < f_{\Gamma;\gamma}(n)$ für fast alle n. (Dies bedeutet, dass es nur endlich viele Ausnahmen gibt.)

Dass das Berechnen einer Funktion auch von der Größe abhängt, kann man anhand der natürlichen Zahlen illustrieren. Nimmt man die Turing-Kodierung, so wird einer Zahl n eine Zeichenkette der Länge $n + 1$ zugeordnet. Nimmt man die übliche Binärkodierung, so bekommt eine Zahl n eine Zeichenkette der Länge $\log_2 n$. Ein Algorithmus, der zwei Zahlen addiert, muss im Falle der Turingmaschine mindestens eine der Zahlen abschreiten, wird also Zeit linear in der Länge der Zeichenkette benötigen. Bei der Turing-Kodierung ist diese Zeit linear in der Größe der Zahl, bei der Binärkodierung ist sie logarithmisch. In praktischen Anwendung ist es daher von entscheidender Bedeutung, in welcher Weise das Problem gegeben, das soll heißen kodiert ist. Die Kodierung entscheidet über die Größe der Eingabe und daher über den Zeitbedarf.

Ein spezieller Fall ist $N = \{0, 1\}$. In diesem Fall sprechen wir anstelle von einer Funktion von einem **Problem**. Anschaulich bedeutet nämlich $f(x) = 1$, dass x wahr ist und $f(x) = 0$, dass x falsch ist. Ich schreibe Probleme mit Anführungszeichen. Zum Beispiel ist '$\vec{x} \in L(\mathfrak{A})$?' ein Problem. Zu berechnen ist, wie gesagt, ob der Wert 1 ist oder 0. Umgangssprachlich ist es das Problem, gegeben eine Zeichenkette $\vec{x}$ und ein Automat $\mathfrak{A}$, zu entscheiden, ob $\vec{x} \in L(\mathfrak{A})$ ist. Die Antwort darauf soll uns ein Algorithmus geben. Dieser Algorithmus ist eine effektive Rechenvorschrift, welche, gegeben in diesem Fall $\vec{x}$ und $\mathfrak{A}$, die richtige Antwort zu

dem Problem berechnet. Jeder Algorithmus hat eine Komplexität; darunter verstehen wir die Zeit und den Platz, den dieser Algorithmus braucht, um seine Antwort zu berechnen. Im Folgenden werden wir nur fragen, wie lange ein Algorithmus braucht. Dabei ist es aber nicht so, dass wir die Zeit in Sekunden angeben können, denn wir wissen ja nicht, wie lange ein Einzelschritt braucht. Wir können nur sagen: der Algorithmus braucht soundsoviel Elementarschritte. (Dabei ist auch nicht immer klar, was eine Elementarschritt ist, aber das Problem lässt sich lösen.) Wir können auch sagen: dieser oder jener Algorithmus ist schneller, braucht weniger Elementarschritte, oder weniger Platz. Ich weise auch darauf hin, dass die Anzahl der Elementarschritte (oder der Platzverbrauch) von den Eingabeparametern abhängt, hier also $\vec{x}$ und $\mathfrak{A}$. Dabei wäre es wenig nützlich, wenn die Abhängigkeit von $\vec{x}$ oder $\mathfrak{A}$ zu konkret wäre. In der Regel berechnet man den Zeitverbrauch in abhängigkeit von der Länge von $\vec{x}$ und der Anzahl der Zustände von $\mathfrak{A}$. Das genügt in aller Regel.

Der naive Algorithmus, der entscheidet, ob ein Automat eine Zeichenkette akzeptiert oder nicht, braucht exponentiell viel Zeit. Ebenso der naive Algorithmus, ob die von einem Automaten erkannte Sprache leer ist. Dieser wäre wie folgt: wir nehmen alle Zeichenketten, deren Länge nicht die Anzahl ζ der Zustände des Automaten überschreitet, und probieren diese der Reihe nach durch. Zunächst einmal müssen wir uns überzeugen, dass wir nur diese Zeichenketten betrachten müssen. Wenn der Automat überhaupt eine Zeichenkette akzeptiert, dann schon eine der Länge $\leq \zeta$. Denn sei der folgende Lauf gegeben.

$$i_0 = q_0 \overset{x_0}{\to} q_1 \overset{x_1}{\to} q_2 \overset{x_2}{\to} q_3 \cdots \overset{x_{n-1}}{\to} q_n \tag{223}$$

Wenn $q_i = q_{i+k}$ $(k > 0)$, dann können wir das Segment $q_i \overset{x_i}{\to} x_{i+1} \cdots \overset{x_{i+k-2}}{\to} x_{i+k-1}$ aus dem Lauf entfernen und bekommen wieder einen Lauf. Ist der erste Lauf akzeptierend, so auch der zweite. Es gibt α^n Zeichenketten der Länge n, wo $\alpha = |A|$, die Größe des Alphabets ist und

$$1 + \alpha + \alpha^2 + \ldots + \alpha^n = (\alpha^{n+1} - 1)/(\alpha - 1) \tag{224}$$

Zeichenketten der Länge $\leq n$. Nehmen wir der Einfachheit mal $\alpha = 2$. Dann ist $(2^{n+1} - 1)/(2 - 1) = 2^{n+1} - 1$. Wir vereinfachen noch einmal. Sei die Anzahl der

benötigten Schritte 2^n. Hier ist tabelliert, wie schnell diese Zahl wächst.

(225)

n	2^n	n	2^n
1	2	11	2048
2	4	12	4096
3	8	13	8192
4	16	14	16384
5	32	15	32768
6	64	16	65536
7	128	17	131072
8	256	18	262144
9	512	19	524288
10	1024	20	1048576

Wir nehmen mal an, der Computer sei in der Lage, 1 Million Schritte in einer Sekunde zu berechnen. Dann braucht er für $n = 10$ etwa eine Millisekunde (1024/1000000). Bei $n = 20$ braucht er schon 1 Sekunde. Jedesmal, wenn die Größe um 10 zunimmt, multipliziert sich die Zeit mit mehr als 1000! Also brauchen wir bei $n = 30$ schon 18 Minuten und bereits einige Monate für $n = 40$. Endliche Automaten haben aber mehrere Hundert, gar Tausend Zustände. Ein solcher Algorithmus ist also nicht praktikabel. Und wir haben die wahre Schrittzahl noch unterschätzt. Bei ζ Zuständen brauchen wir rund α^ζ Schritte!

Man wünscht sich also ein Verfahren, das schneller geht. Denn es ist ja nicht ausgemacht, dass der Algorithmus, denn wir gewählt haben, nicht doch umständlicher ist als nötig. In der Tat kann man das Ganze sehr viel schneller haben. Wir nennen einen Zustand n-**erreichbar**, wenn es ein Wort $\vec{x}$ der Länge n gibt derart, dass $i_0 \xrightarrow{\vec{x}} q$. Und es sei q **erreichbar**, wenn es n-erreichbar ist für ein $n \in \mathbb{N}$. Es bezeichnet nun R_n die Menge der Zustände, die k-erreichbar sind für ein $k \leq n$. Dann ist $R_0 = \{i_0\}$. Der Algorithmus ist nun wie folgt. In jedem Schritt berechnen wir zu der bisherigen Menge R_n nun die Menge aller Zustände, die von einem Zustand aus R_n aus in einem Schritt erreicht werden können und fügen diese hinzu. Das ergibt die Menge R_{n+1}. Wir wiederholen diesen Schritt solange, bis keine Zustände mehr hinzukommen. Dann ist $R_{n+1} = R_n$. Es ist klar, dass wir höchstens $\zeta - 1$ Schritte brauchen, dass heißt, dass $n < \zeta$. Denn bei jedem Schritt muss ja ein Zustand hinzukommen, sonst hört das Verfahren auf.

Wir schauen uns nun an, wie lange dieser Algorithmus benötigt. Sei R_n gegeben. Dann nehmen wir für jedes $q \in R_n$ und jeden Buchstaben a jedes q' mit

$q \xrightarrow{a} q'$ hinzu. Dies macht $|R_n| \times |A| \times \zeta$ Elementarschritte. Nun ist $|R_n| \leq \zeta$, also können wir dies mit $\alpha\zeta^2$ nach oben abschätzen. Wir wiederholen dies ζ mal, so kommen wir dann auf $\alpha\zeta^3$ Elementarschritte.

Aber auch hier gilt: es geht noch besser. Der Algorithmus arbeitet immer noch unnötig viel. Denn wenn wir genau hinschauen, berechnet er die Nachfolger eines Zustandes immer wieder. Das lässt sich vermeiden. Hier ist eine noch bessere Methode. Wir beginnen mit $R_{-1} := \varnothing$ und $S_0 = \{i_0\}$. S_1 ist dann die Menge der Nachfolger von S_0 abzüglich S_0 selbst, und $R_0 := S_0$. In dem $i$ten Schritt berechnen wir die Mengen S_i und R_{i-1}. R_{i-1} ist die Menge der höchstens $i-1$-erreichbaren Zustände und S_i ist die Menge der Zustände, die in i Schritten erreichbar sind aber nicht in $i-1$ Schritten. Diese sind in einem Schritt von den in $i-1$ aber nicht in $i-2$ Schritten erreichbaren Zuständen erreichbar (also von den Zuständen in S_{i-1}) und sind nicht in R_{i-1}. Der Algorithmus ist also wie folgt.

$$\begin{aligned} R_i &:= S_i \cup R_{i-1} \\ S_{i+1} &:= \{q' : \text{ es existiert } a \in A, q \in S_i \text{ mit } q \xrightarrow{a} q'\} - R_i \end{aligned} \tag{226}$$

Dieser Algorithmus ist viel schneller, denn für jedes $q \in Q$ werden nur einmal die Nachfolger berechnet. Der vorherige Algorithmus berechnet hingegen Nachfolger in jedem Schritt neu. Trotzdem fällt bei der Komplexitätsberechnung der Gewinn nicht ins Auge. Denn in jedem Schritt fallen $|A||S_i||Q| = \alpha|S_i|\zeta$ an, und wir müssen ζ viele Schritte machen. Überschlagsmäßig käme dabei wieder $\alpha\zeta^2$ heraus, aber wenn wir genau rechnen, dann bekommen wir $\alpha\zeta$ in jedem Schritt, also $\alpha\zeta^2$ insgesamt. Ist der Automat deterministisch, so verbessert sich dies sogar zu $\alpha\zeta$.

Ich tabelliere n^2:

(227)

n	n^2	n	n^2
1	1	11	121
2	4	12	144
3	9	13	169
4	16	14	196
5	25	15	225
6	36	16	256
7	49	17	289
8	64	18	324
9	81	19	361
10	100	20	400

Für $n = 10$ benötigt der Algorithmus 100 Schritte, oder eine zehntel Millisekunde. Für $n = 20$ sind es viermal so viel, für $n = 30$ etwa eine Millisekunde. Bei tausend Zuständen dauert es immerhin eine Sekunde, es sind nämlich eine Million Schritte nötig. Diesen Algorithmus kann man nicht mehr wesentlich verbessern. Denn der Automat besitzt bis zu $\alpha\zeta^2$ Übergänge, und wir müssen im schlechtesten Fall jeden einzelnen von ihnen einmal verwenden.

Nun gibt es Probleme, die man nicht einfacher machen kann. Ein solches Problem ist das sogenannte **3SAT**, ein Problem, ob eine boolesche Formel gewissen Typs, erfüllbar ist oder nicht. Dieses Problem hat die Stufe **NP**, welches besagt, dass das Nachprüfen einer Lösung polynomiell viel Zeit benötigt. Allerdings muss man die Lösung raten, und wenn man nicht raten will, so muss man viel mehr Zeit aufbringen. Wieviel mehr ist immer noch ungelöst. Bisher ist nicht gezeigt, dass nicht doch **NP** = **P**, das heißt, dass es einen polynomiellen Algorithmus gibt, bei dem man nicht raten muss.

Ich beschließe dieses Kapitel mit einem Algorithmus, der sehr bekannt ist und wieder einmal zeigt, wie man durch geschicktes Rechnen eine bessere Leistung bekommt. Das Problem ist denkbar einfach: man bekommt eine Liste und soll die Elemente der Größe nach ordnen. Der naive Algorithmus ist wie folgt: wir gehen die Liste durch und suchen das kleinste Element, nehmen es aus der Liste und stellen es in einer neue Liste an den Anfang. Nun suchen wir in der Liste das kleinste Element, nehmen es heraus und stellen es der neuen Liste hinter das bereits vorhandene Element. Wie sucht man nach dem kleinsten Element? Wir benötigen ein Gedächtnis von zwei Zahlen, I und K. Wir setzen $I := 0$ und K das erste Element der Liste. Wir gehen die Liste der Reihe nach durch; jedesmal, wenn wir einen Schritt weitergehen, tun wir Folgendes: ist das gerade gelesene Listenelement kleiner als K, dann wird K nunmehr auf das neue Listenelement gesetzt und I auf die Ordnungsnummer der Liste (damit wir wissen, wo das Element gefunden wurde). Andernfalls bleiben I und K wie sie sind. K sagt uns also, wie hoch der Wert des bisher kleinsten Elements ist und I, wo es zu finden ist. Wir müssen ganz durch die Liste gehen. Sind wir am Ende angekommen, so haben wir mit I die Nummer, wo das Element zu finden ist (K brauchen wir streng genommen nicht mehr). Im $i$ten Schritt ist die Liste $n - i + 1$ lang, wenn n die Länge der Ausgangsliste ist; also benötigen wir $n - i + 1$ viele Schritte zum Suchen.

$$n + (n-1) + (n-2) + \ldots + 2 + 1 = n \cdot (n+1)/2 \tag{228}$$

Das ist nicht schlecht, aber wenn die Listen lang sind, wächst diese Zahl beträchtlich. Immerhin geht es schneller.

Man teile die Liste L in zwei gleich große Teile L_1 und L_2. Es ist also L die Konkatenation von L_1 mit L_2. Falls das nicht genau geht, sei L_1 um ein Element größer als L_2. Wir ordnen diese nach unserem Algorithmus. Wenn dieser fertig ist, haben wir Listen M_1 und M_2, beide aufsteigend geordnet. Die Frage ist nun, wie man solche Listen zu einer aufsteigenden Liste M zusammenschweißt. Sei zum Beispiel $M = [1, 3, 5]$ und $N = [2, 4, 5]$. Wir berechnen L. Wir vergleichen die ersten Listenelemente. Das kleinere wird L gegeben, und aus seiner Liste gestrichen. Wir bekommen nun

(229) $M = [3, 5], N = [2, 4, 5], L = [1]$

Jetzt vergleichen wir wieder die ersten Elemente von M und N und schlagen L das kleinere der beiden Element zu.

(230) $M = [3, 5], N = [4, 5], L = [1, 2]$

Und so machen wir weiter.

(231) $M = [5], N = [4, 5], L = [1, 2, 3]$

(232) $M = [5], N = [5], L = [1, 2, 3, 4]$

(233) $M = [], N = [5], L = [1, 2, 3, 4, 5]$

(234) $M = [], N = [], L = [1, 2, 3, 4, 5, 5]$

Jeder Schritt ist ein einfacher Größenvergleich. Wir müssen so viele Vergleiche anstellen, wie Elemente in M und N zusammen vorhanden sind.

Zurück zu unserem Sortieralgorithmus. Die Listen L_1 und L_2 haben zusammen n Elemente, ebenso M_1 und M_2. Also benötigen wir n Schritte, um diese zusammenzuschweißen. Die Länge von L_1 und L_2 ist $n/2$. Wir müssen nun aber die Listen L_1 und L_2 auch noch ordnen. Dazu benutzen wir die gleiche Methode: wir halbieren die Listen und ordnen die Stücke. Alle zusammen verlangen n Schritte (wir bekommen vier Teile der Größe $n/4$). Wie oft also muss man teilen? Die Antwort ist: so oft, bis $2^k = n$ ist. Dann aber ist $k = \log_2 n$ (dies geht in der Regel nicht exakt auf). Wie insgesamt benötigte Zeit ist also $n \log_2 n$. Ist $n = 1024$, so ist $k = 10$, und wir benötigen nun 10 mal 1024 Schritte, also 10240. Der naive Algorithmus hingegen verlangt 1024×1024 Schritte, also rund 100mal so viel! Ich tabelliere die Laufzeit für den naiven und den neuen Algorithmus.

(235)

	2^0	2^4	2^8	2^{12}	2^{16}
n	1	16	256	4096	65536
$n \log_2 n$	0	64	2048	59152	1048576
$n^2/2$	1/2	128	32768	8388608	2147483648

In der Komplexitätstheorie befasst man sich nicht in aller Einzelheit damit, wieviel Schritte ein Algorithmus benötigt. Das Problem ist vielfach, dass die Rechnung viel zu kompliziert wird und wenig bringt. Zum Beispiel ist unklar, was genau ein Einzelschritt ist und wie viel er effektiv dauert (das hängt natürlich auch von der Hardware ab). Deswegen sieht man oft von Konstanten ab. Man interessiert sich nicht dafür, ob ein Algorithmus $3n$ oder $5n$ oder $126n$ Schritte benötigt, sondern sagt: er ist linear, das heißt, ein Vielfaches von n. Man schreibt dann einfach $O(n)$. Dies bedeutet aber auch noch mehr, nämlich dass das lineare Verhalten erst ab einer Minimallänge gilt. Man sagt, die Abschätzung gelte für fast alle n. Ein Algorithmus, der $42n + 107$ Schritte braucht, ist somit gleichrangig mit einem Algorithmus, der n Schritte braucht. Dabei heißt gleichrangig, dass $O(42n + 107) = O(n)$, und dies bedeutet, dass es ein n_0 gibt und eine Konstante C_0, derart, dass $42n + 107 \leq Cn$ für alle $n \geq n_0$, und ein n_1 und eine Konstante C_1 derart, dass $n \leq C_1(42n+107)$ für alle $n \geq n_1$. Wir schreiben $O(n) \leq O(42n+107)$, und meinen damit, dass es C_0 und n_0 wie angegeben gibt. Entsprechend schreiben wir $O(42n+107) \leq O(n)$ für den zweiten Sachverhalt. Es ist $O(n) = O(42n+107)$, womit wir sagen wollen, das beides zusammen gilt. Sehen wir uns an, warum das richtig ist. Wählen wir $C_1 = 1$ und $n_1 := 0$. In der Tat ist $n \leq 42n + 107$ für alle n, also haben wir die zweite Ungleichung. Wir wählen nun $C_0 := 43$ und $n_0 := 108$. Für $n \geq 107$ ist

$$43n = 42n + n \geq 42n + 107 \tag{236}$$

wie zu zeigen war.

Ist die Laufzeit eines Algorithmus $cn^2 + bn + a$, so ist dieser gleichrangig mit n^2, und so weiter. Beim Polynom zählt also nur der Grad. Allerdings gibt es Zwischenstufen zwischen $O(n)$ und $O(n^2)$. So ist $O(n) < O(n \log_2 n) < O(n^2)$.

17 Endliche Transduktoren

Endliche Transduktoren sind sehr ähnlich zu endlichen Automaten. Die einfachste Art, sich einen Transduktor vorzustellen, ist als einen endlichen Automaten, der auf einem Ausgabeband eine Spur hinterlässt. Die populäre Variante ist die einer Übersetzungsmaschine mit endlichem Gedächtnis. Ein **endlicher Transduktor** ist ein Sextupel

$$\mathfrak{T} = \langle A, B, Q, i_0, F, \delta \rangle \tag{237}$$

wo A und B Alphabete sind, A das **Eingabealphabet** und B das **Ausgabealphabet**, Q eine endliche Menge (die Menge der inneren **Zustände**), i_0 der **Startzustand**, F die Menge der **akzeptierenden Zustände** und

$$\delta \subseteq Q \times A_\varepsilon \times Q \times B_\varepsilon \tag{238}$$

(Hier ist $A_\varepsilon := A \cup \{\varepsilon\}$, und $B_\varepsilon := B \cup \{\varepsilon\}$.) Wir schreiben $q \xrightarrow{a:b} q'$ falls $\delta(a, q, b, q') \in \delta$. Wir sagen in diesem Fall, dass $\mathfrak{T}$ einen Übergang von q nach q' macht bei Eingabe a, und dass er dabei b ausgibt.

Wir erweitern diese Notation auf Zeichenketten. Falls $q \xrightarrow{\vec{u}:\vec{v}} q'$ und $q' \xrightarrow{\vec{x}:\vec{y}} q''$ so schreiben wir $q \xrightarrow{\vec{u}\vec{x}:\vec{v}\vec{y}} q''$.

Wir werden im Verlaufe noch einige Transduktoren konstruieren. Für jetzt soll ein einfaches Beispiel genügen. (Vrgl. die Abbildung 5.) Das Eingabealphabet ist $A := \{\mathtt{0}, \mathtt{1}\}$, das Ausgabealphabet $B := \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}$. Es soll folgende `00` duch `a` ersetzt werden, `01` durch `b`, `10` durch `c` und `11` durch `d`. Wir benutzen dazu drei Zustände, 0, 1 und 2, 0 sei initial und als einziger akzeptierend. Die Übergänge sind wie folgt.

$$\begin{aligned}\langle 0, \mathtt{0}, 1, \mathtt{a}\rangle, \langle 0, \mathtt{0}, 2, \mathtt{b}\rangle, \langle 0, \mathtt{1}, 1, \mathtt{c}\rangle, \langle 0, \mathtt{1}, 2, \mathtt{d}\rangle,\\ \langle 1, \mathtt{0}, 0, \varepsilon\rangle, \langle 2, 1, 0, \varepsilon\rangle\end{aligned} \tag{239}$$

Nehmen wir an, der Automat bekommt die Eingabe `011011`. Dann passiert Folgendes.

- Im Zustand 0 liest der Automat `0` ein und geht in den Zustand 1 über; Ausgabe ε.

- Im Zustand 1 liest der Automat 1 ein und geht in den Zustand 0 über; Ausgabe b.
- Im Zustand 0 liest der Automat 1 ein und geht in den Zustand 2 über; Ausgabe ε.
- Im Zustand 2 liest der Automat 0 ein und geht in den Zustand 0 über; Ausgabe c.
- Im Zustand 0 liest der Automat 1 ein und geht in den Zustand 2 über; Ausgabe ε.
- Im Zustand 2 liest der Automat 1 ein und geht in den Zustand 0 über; Ausgabe d.

In etwas komprimierter Form stellt man das so dar:

$$0 \xrightarrow{0:\varepsilon} 1 \xrightarrow{1:b} 0 \xrightarrow{1:\varepsilon} 2 \xrightarrow{0:c} 0 \xrightarrow{1:\varepsilon} 2 \xrightarrow{1:d} 0 \tag{240}$$

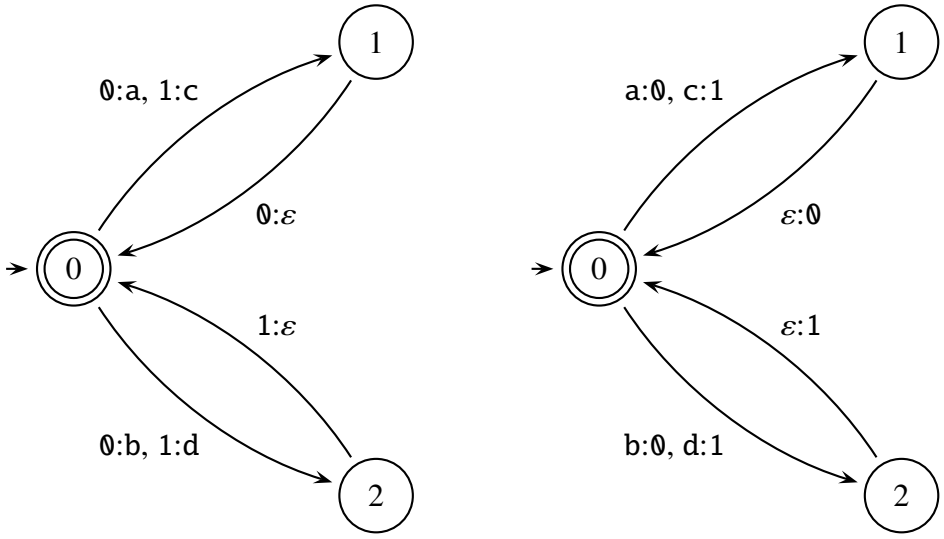
Die Eingabe ist akzeptiert, wenn der Automat in einen akzeptierenden Zustand läuft (genauer: wenn es einen Lauf gibt derart, dass der Automat bei dieser Eingabe in einen akzeptierenden Zustand läuft). Im vorliegenden Fall wird eine Eingabe nur dann akzeptiert, wenn sie gerade Länge hat.

Das Eingabealphabet sei $A := \{a, b, c, d\}$, das Ausgabealphabet $B := \{0, 1\}$. Wir wollen Worte über A in Worte über B übersetzen. Dabei soll a durch 00, b durch 01, c durch 10 und d durch 11 wiedergegeben werden. Dazu sei die Zustandsmenge $\{0, 1, 2\}$, der Startzustand 0 und 0 sei auch der einzige akzeptierende Zustand. Die Übergänge sind

$$\langle 0, a, 1, 0\rangle, \langle 0, b, 2, 0\rangle, \langle 0, c, 1, 1\rangle, \langle 0, d, 2, 1\rangle, \langle 1, \varepsilon, 0, 0\rangle, \langle 2, \varepsilon, 0, 1\rangle \tag{241}$$

Dabei schreiben wir $\langle 0, a, 1, 0\rangle$ alternativ als $0 \xrightarrow{a:0} 1$, und es bedeutet, dass die Maschine im Zustand 0 durch Lesen des Buchstabens a in den Zustand 1 übergeht und dabei den Buchstaben 0 ausgibt. Man beachte, dass wir auch Übergänge der Form $1 \xrightarrow{\varepsilon:0} 0$ haben, das bedeutet, dass die Maschine im Zustand 1 keinen Buchstaben einliest, dafür aber 0 ausgibt und in den Zustand 0 übergeht. Diese Übergänge sind nötig, weil ja die Eingabe und die Ausgabe nicht in der Länge

Abbildung 5: Ein endlicher Transduktor



übereinstimmen müssen. So müssen wir gelegentlich ein Symbol einlesen ohne zu schreiben oder umgekehrt.

Wir bezeichnen mit $R_{\mathfrak{T}}$ die folgende Relation.

$$R_{\mathfrak{T}} := \{\langle \vec{x}, \vec{y}\rangle : i_0 \xrightarrow{\vec{x}:\vec{y}} q \in F\} \tag{242}$$

Für eine Zeichenkette $\vec{x}$ setze

$$R_{\mathfrak{T}}(\vec{x}) := \{\vec{y} : \vec{x} R_{\mathfrak{T}} \vec{y}\} \tag{243}$$

Für eine Sprache $S \subseteq A^*$ sei schließlich

$$R_{\mathfrak{T}}[S\,] := \{\vec{y} : \text{exists } \vec{x} \in S \ : \ \vec{x} R_{\mathfrak{T}} \vec{y}\} \tag{244}$$

Dies ist die Menge aller Zeichenketten $\vec{y}$ über B, für welche es eine Zeichenkette $\vec{x}$ über A gibt derart, dass $\vec{x} R_{\mathfrak{T}} \vec{y}$, was wiederum bedeutet, dass $i_0 \xrightarrow{\vec{x}:\vec{y}} q$ für ein $q \in F$.

Ich weise vorsorglich darauf hin, dass man Übergänge der Form $\langle q, \varepsilon, q', \varepsilon\rangle$ eliminieren kann, ohne dass sich wesentliches ändert. Sei nämlich δ gegeben, so sei setze

$$\begin{aligned} \delta^* := \quad & \delta - \{\langle q, \varepsilon, q', \varepsilon\rangle : q, q' \in Q\} \\ \cup \quad & \{\langle q, \varepsilon, q', a\rangle : q \xrightarrow{\varepsilon:a} q'\} \\ \cup \quad & \{\langle q, a, q', \varepsilon\rangle : q \xrightarrow{a:\varepsilon} q'\} \end{aligned} \tag{245}$$

Allerdings kann man Übergänge mit Leerzeichen nicht gänzlich vermeiden, sofern mindestens ein Zeichen konsumiert wird.

Definition 17.1 ***Reguläre Relationenterme (rRT)*** *sind wie folgt definiert.*

① $\varnothing$ *ist ein rRT.*

② *Sind* $x \in A_\varepsilon$ *und* $y \in B_\varepsilon$*, so ist* $x : y$ *ein rRT.*

③ *Sind* r *und* s *rRTe, so auch* r^**,* $r \cup s$ *und* $r \cdot s$*.*

Es sei nun

① $L(\varnothing) := \varnothing$.

② $L(x : y) := \langle x, y\rangle$.

③ $L(R \cdot S) := \{\langle \vec{x}\vec{u}, \vec{y}\vec{v}\rangle : \langle \vec{x}, \vec{y}\rangle \in L(R), \langle \vec{u}, \vec{v}\rangle \in L(S)\}$.

④ $L(R \cup S) := L(R) \cup L(S)$.

⑤ $L(R^*) := \bigcup_{i\in\mathbb{N}} L(R^n)$

*Eine Relation $R \subseteq A^* \times B^*$ heißt **regulär**, falls es einen regulären Relationenterm r gibt mit $R = L(r)$.*

Ich mache darauf aufmerksam, dass es nunmehr Terme für reguläre Sprachen gibt und Terme für reguläre Relationen. Die Symbole $\cdot$, $\cup$, sowie * können für beide benutzt werden. Wir verabreden noch, dass für beliebige Zeichenketten $\vec{x}$, $\vec{y}$ die Notation $\vec{x} : \vec{y}$ das Paar $\langle \vec{x}, \vec{y}\rangle$ bezeichnet, und dass schließlich für Terme r und s $r : s$ die Menge $L(r) \times L(s)$ bezeichne (also nicht $\langle L(r), L(s)\rangle$!).

Satz 17.2 *Es sei $R \subseteq A^* \times B^*$ ist genau dann regulär wenn es einen endlichen Transduktoren $\mathfrak{T}$ gibt mit $R = R_{\mathfrak{T}}$.*

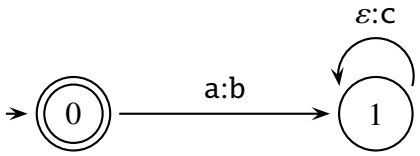
Der Beweis ist analog zu dem Beweis für Automaten. Ist R regulär, so definieren wir den Automaten induktiv über den Term, der R definiert. (Welchen wir dafür nehmen, ist dafür unerheblich.) Dazu müssen wir wie bei endlichen Automaten Konstrukte für Vereinigung, Konkatenation und Stern definieren. Dabei laufen die Beweise für Konkatenation und Stern wie im einfachen Fall. Bei der Vereinigung tun wir Folgendes. Gegeben seien zwei Automaten $\mathfrak{T} = \langle A, B, Q_1, i_1, F_1, \delta_1\rangle$ und $\mathfrak{T}_2 = \langle A, B, Q_2, i_2, F_2, \delta_2\rangle$, derart, dass

1. $L(\mathfrak{T}_1) = R_1$ und $L(\mathfrak{T}_2) = R_2$.
2. $Q_1 \cap Q_2$.

Dann sei $i_3 \notin Q_1 \cup Q_2$ ein neuer Zustand, und wir setzen $Q_3 := Q_1 \cup Q_2 \cup \{i_0\}$, $F_3 := F_1 \cup F_2$, sowie

(246) $\quad \delta_3 := \delta_1 \cup \delta_2 \cup \{\langle i_3, \varepsilon, i_1, \varepsilon\rangle, \langle i_3, \varepsilon, i_2, \varepsilon\rangle\}$

Abbildung 6: Endliche Eingabe, unendliche Ausgabe



Schließlich sei

(247) $\mathfrak{T}_3 := \langle A, B, Q_3, i_3, F_3, \delta_3 \rangle$

Dann ist $L(\mathfrak{T}_3) = L(\mathfrak{T}_1) \cup L(\mathfrak{T}_2)$. Denn ist $i_3 \xrightarrow{\vec{x}:\vec{y}} q$ für $q \neq i_3$, dann ist $q \in Q_1$ oder $q \in Q_2$, aber nicht beides zugleich. Im ersten Fall haben wir $i_3 \xrightarrow{\varepsilon:\varepsilon} i_1 \xrightarrow{\vec{x}:\vec{y}} q$, wobei der zweite Übergang ganz in $\mathfrak{T}_1$ läuft, im zweiten Fall haben wir $i_3 \xrightarrow{\varepsilon:\varepsilon} i_2 \xrightarrow{\vec{x}:\vec{y}} q$, wobei der zweite Übergang ganz in $\mathfrak{T}_2$ läuft. Im ersten Fall ist dann $q \in F_1$, also $\langle \vec{x}, \vec{y} \rangle \in L(\mathfrak{T}_1)$, im zweiten Fall $q \in F_2$ und so $\langle \vec{x}, \vec{y} \rangle \in L(\mathfrak{T}_2)$. Die Umkehrung ist ebenso leicht.

Für die andere Richtung schließlich haben wir einen Transduktor und definieren jetzt für zwei Zustände i und j die Sprachen $L(i, j) := \{\langle \vec{x}, \vec{y} \rangle : i \xrightarrow{\vec{x}:\vec{y}} j\}$ und stellen wiederum ein Gleichungssystem auf, das wir lösen.

Viele Relationen zwischen Worten sind regulär. Zum Beispiel ist die Morphologie in aller Regel mit regulären Relationen behandelbar. Doch davon später.

Der oben definierte Automat hatte die Eigenschaft, dass jedem Wort über A ein Wort über B zugeordnet wird, aber nicht jedem Wort über B ein Wort über A. Außerdem ist die Relation in beide Richtungen eindeutig, das heißt, jeder A-Zeichenkette enstpricht genau ein B-Zeichenkette, und jeder B-Zeichenkette entspricht höchstens eine A-Zeichenkette. Es kann aber eine reguläre Relation in beide Richtungen eine echte Relation sein. Hier ist zum Beispiel ein Transduktor, der das einzige Wort a in bc^* übersetzt.

(248) $A := \{\mathsf{a}\}, B := \{\mathsf{b}, \mathsf{c}\}, Q := \{0, 1\}, i_0 := 0, F := \{1\},$
$\delta := \{\langle 0, \mathsf{a}, 1, \mathsf{b}\rangle, \langle 1, \varepsilon, 1, \mathsf{c}\rangle\}$

Die einzige Eingabe, die dieser Automat akzeptiert, ist das Wort a. Dieses wird übersetzt zu b, bc, bcc und so weiter. So kann die Übersetzung hochgradig unterbestimmt sein. Man beachte auch das Folgende. Für eine Sprache $S \subseteq A^*$ gilt

(249) $$R_{\mathfrak{T}}[S] = \begin{cases} \mathsf{bc}^* & \text{falls } \mathsf{a} \in S \\ \varnothing & \text{sonst.} \end{cases}$$

Das ist deswegen so, weil nur die Zeichenkette a eine Übersetzng hat. Für alle anderen Zeichenketten $\vec{x} \neq \mathsf{a}$ ist $R_{\mathfrak{T}}(\vec{x}) = \varnothing$.

Einen Transduktor kann man auch umkehren. Dann übersetzt er von B^* nach A^*. Dazu sei $\langle A, B, Q, i_0, F, \delta\rangle$. Dann setze

(250) $$\mathfrak{T}^{\bowtie} := \langle B, A, i_0, Q, F, \delta^{\bowtie}\rangle$$

wobei

(251) $$\delta^{\bowtie} := \{\langle x, q, y, q'\rangle : \langle y, q, x, q'\rangle \in \delta\}$$

Ferner sei für eine Relation $R \subseteq M \times N$:

(252) $$R^{\smile} := \{\langle y, x\rangle : \langle x, y\rangle \in R\}$$

dies nennen wir die **konverse Relation**. Dann ist also

(253) $$R^{\smile}_{\mathfrak{T}}(\vec{y}) = \{\vec{x} : \vec{x} R_{\mathfrak{T}} \vec{y}\}$$

Ebenso ist dann

(254) $$R^{\smile}_{\mathfrak{T}}[S] := \{\vec{x} : \text{exists } \vec{y} \in S \ : \ \vec{x} R_{\mathfrak{T}} \vec{y}\}$$

Hier ist ein interessanter Satz über reguläre Relationen.

Satz 17.3 *Es sei $\mathfrak{T}$ ein Transduktor von A nach B und sei $S \subseteq A^*$ regulär. Dann ist $R_{\mathfrak{T}}[S]$ auch regulär.*

Beweis. Sei $\mathfrak{T} = \langle A, B, Q, i_0, F, \delta\rangle$. Sei ferner S regulär. Dann existiert ein endlicher Automat $\mathfrak{B} = \langle A, Q', j_0, F', \eta\rangle$ derart, dass $L(\mathfrak{B}) = S$. Wir konstruieren zunächst einen Transduktor $\mathfrak{U} := \langle A, B, Q' \times Q, \langle j_0, i_0\rangle, F' \times F, \theta\rangle$, wobei

$$\langle q, r\rangle \overset{x:y}{\to} \langle q', r'\rangle :\Leftrightarrow \langle q, x, q'\rangle \in \delta, \langle r, x, r', y\rangle \in \eta \tag{255}$$

(Falls $x = \varepsilon$, so sei $q' = q$.) Dies ist ein Transduktor, der nur die Sprache S akzeptiert. Denn $\langle j_0, i_0\rangle \overset{\vec{x}:\vec{y}}{\to} \langle q, r\rangle$ genau dann, wenn $j_0 \overset{\vec{x}}{\to} q$ in $\mathfrak{A}$ und $i_0 \overset{\vec{x}:\vec{y}}{\to} r$ in $\mathfrak{T}$. Und wenn $\mathfrak{U}$ das Paar $\vec{x} : \vec{y}$ akzeptiert, muss $q \in F$ sein, also $\vec{x} \in L(\mathfrak{A})$. Offensichtlich ist

$$R_{\mathfrak{U}}[A^*] = \{\vec{y} : \text{ es gibt } \vec{x} \in S \text{ mit } \langle \vec{x}, \vec{y}\rangle \in R_{\mathfrak{T}}\} \tag{256}$$

Dies ist unsere Sprache. Nun machen wir einen zweiten Schritt und eliminieren das Eingabealphabet:

$$\mathfrak{B} := \langle B, Q' \times Q, \langle j_0, i_0\rangle, F' \times F, \theta\rangle \tag{257}$$

Hierbei ist

$$\theta := \{\langle\langle q, r\rangle, y, \langle q', r'\rangle\rangle : \text{ es existiert } x \in A_\varepsilon \text{ mit } \langle\langle q, r\rangle, x, \langle q', r'\rangle, y\rangle\} \tag{258}$$

Hier ist nun $\langle q, r\rangle \overset{\vec{y}}{\to} \langle q', r'\rangle$ für beliebiges $\vec{y} \in B^*$, falls ein $\vec{x} \in A^*$ existiert mit $\langle q, r\rangle \overset{\vec{x}:\vec{y}}{\to} \langle q', r'\rangle$ in $\mathfrak{U}$. Und dies bedeutet, dass es ein $\vec{x} \in L(\mathfrak{A}) = S$ gibt mit $\vec{x} R_{\mathfrak{T}} \vec{y}$, was zu zeigen war. ⊣

Man beachte, dass der Transduktor dazu benutzt wird, um eine beliebige Sprache zu übersetzen. Das bedeutet, dass $R_{\mathfrak{T}}[S]$ auch dann definiert ist, wenn S nicht regulär ist. Es gilt aber zum Beispiel, dass $R_{\mathfrak{T}}[S]$ kontextfrei ist, wenn S kontextfrei ist.

Korollar 17.4 *Es sei $f : A \to B^*$ eine beliebige Funktion. Dann sei*

$$U_f(\vec{x}) := \begin{cases} \varepsilon & \text{falls } \vec{x} = \varepsilon \\ U_f(\vec{y}) f(a) & \text{falls } \vec{x} = \vec{y}a \end{cases} \tag{259}$$

Dann ist $U_f[S]$ regulär, wenn S regulär ist.

Zum Beweis müssen wir jetzt lediglich einen Transduktor konstruieren, dessen Relation $\{\langle \vec{x}, U_f(\vec{x})\rangle : \vec{x} \in A^*\}$ ist.

Ein sehr wichtiger Satz ist der folgende Kaskadensatz. Seien $U \subseteq D \times E$ und $V \subseteq E \times F$ beliebige Relationen. Dann bezeichnet

$$R \circ S := \{\langle x, z\rangle : \text{existiert ein } y \in E \text{ mit } \langle x, y\rangle \in R, \langle y, z\rangle \in S\} \tag{260}$$

die sogenannte **Verkettung** von R und S. Es ist $x\,R \circ S\,z$ genau dann, wenn ein y existiert mit $x\,R\,y\,S\,z$.

Satz 17.5 (Kaskadensatz) *Ist $R \subseteq A^*$ und $S \subseteq B^* \times C^*$ eine reguläre Relation, so auch $R \circ S$.*

Ich erwähne noch ein paar Spezialfälle. Zum Beispiel können wir zu jedem einfachen regulären Term t einen relationalen Term $t^{\bowtie}$ wie folgt definieren:

① $\varnothing^{\bowtie} := \varnothing$.

② $\varepsilon^{\bowtie} := \varepsilon : \varepsilon$.

③ $a^{\bowtie} := a : a$.

④ $(t \cup u)^{\bowtie} := t^{\bowtie} \cup u^{\bowtie}$.

⑤ $(t \cdot u)^{\bowtie} := t^{\bowtie} \cdot u^{\bowtie}$.

⑥ $(t^*)^{\bowtie} := (t^{\bowtie})^*$.

Dann gilt:

$$L(t^{\bowtie}) = \{\langle \vec{x}, \vec{x}\rangle : \vec{x} \in L(t)\} \tag{261}$$

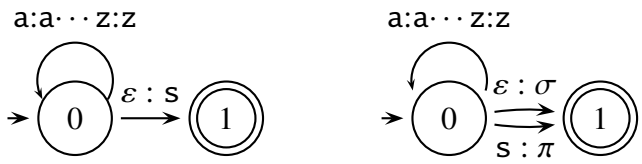
Bezeichne mit Δ_U die Sprache

$$\Delta_U := \{\langle \vec{x}, \vec{x}\rangle : \vec{x} \in U\} \tag{262}$$

Haben wir nun eine beliebige reguläre Relation $R \subseteq A^* \times B^*$ und eine reguläre Sprache $U \subseteq A^*$, so ist $\Delta_U \circ R = (U \times B^*) \cap R$. Dies ist die Sprache $\{\langle \vec{x}, \vec{y}\rangle : \vec{x} \in U \text{ und } \langle \vec{x}, \vec{y}\rangle \in R\}$. Nach dem Kaskadensatz ist diese auch regulär.

Abbildung 7: Die einfache Pluralmaschine fürs Englische

(a) Sg → Pl (b) Dekomposition



18 Finite State Morphologie

Ein sehr beliebtes Anwendungsfeld von Transduktoren ist in der Morphologie. Morphologie ist praktisch ausnahmslos mit endlichen Transduktoren behandelbar. Dies bedeutet etwa Folgendes. Nehmen wir die Relation zwischen Oberflächenmorphologie und Tiefenmorphologie, dh zwischen der Form eines Wortes auf dem Papier und seiner Analyse in Morpheme. Oder nehmen wir die Relation zwischen einem Nomen im Singular und seiner Pluralform. All diese Relationen sind regulär.

Nehmen wir zum Beispiel eine Maschine, die den Plural im Englischen bildet. Diese hat zwei Zustände, 0, und 1; 0 ist initial, 1 ist akzeptierend. Die Übergänge sind wie folgt. (Vgl. Abbildung 7(a). Ich benutze der Einfachheit halber nur Kleinbuchstaben.)

(263) $\langle 0, \mathsf{a}, 0, \mathsf{a}\rangle, \langle 0, \mathsf{b}, 0, \mathsf{b}\rangle, \ldots, \langle 0, \mathsf{z}, 0, \mathsf{z}\rangle, \langle 0, \varepsilon, 1, \mathsf{s}\rangle.$

Diese Maschine nimmt die Eingabe und hängt ein /s/ an. Das ist natürlich etwas primitiv; denn erstens schaut die Maschine gar nicht, ob das Wort ein Nomen das Englischen ist (was auch besser so ist, weil das Lexikon eigentlich offen ist, das heißt, ständigem Wandel unterliegt) und zweitens gibt bei der Pluralbildung sogar im Englischen eine ganze Menge Unregelmäßigkeiten.

Anstelle der Pluralbildung können wir auch eine Maschine bauen, die eine Tiefenrepräsentation, etwa /car + SG/ oder /car + PL/ nimmt und im ersten Fall

/car/ und im zweiten /cars/ ausgibt. Sei σ das Symbol für Singular und π das Symbol für Plural; dann bekommen wir folgende Übergänge (siehe auch die Abbildund 7(b)).

(264) $\langle 0, \mathtt{a}, 0, \mathtt{a}\rangle, \langle 0, \mathtt{b}, 0, \mathtt{b}\rangle, \ldots, \langle 0, \mathtt{z}, 0, \mathtt{z}\rangle, \langle 0, \sigma, 1, \varepsilon\rangle, \langle 0, \pi, 1, \mathtt{s}\rangle.$

Diese Maschine akezptiert eine Zeichenkette mit einem einzigen Vorkommen von /σ/ oder /π/ am Ende. Dieses wird in /ε/ bzw. /s/ umgeschrieben. Diese Maschine können wir auch umdrehen, und dann übersetzt sie Oberflächenformen in Tiefenformen. Sie übersetzt /arm/ in /armσ/ und /arms/ in /armπ/ und /armsσ/. Das mag stören, aber ist eigentlich völlig in Ordnung. Die Maschine hat nämlich gar keine Ahnung, welcher Art das englische Lexikon ist und geht davon aus, dass eine Form in /s/ ja auch ein Singular sein könnte. In der Tat gibt es solche Worte, /bus/ ist eines; oder /plus/. Die Maschine zur Übersetzung in Tiefenformen hat einen Vorteil. Wir können die Übersetzung von Singular nach Plural nun durch eine Kaskade erreichen: wir übersetzen die Oberflächenform in eine Tiefenform, in dieser ersetzen wir σ durch π (das ist auch regulär), und dann übersetzen wir das Ganze wieder in eine Oberflächenform.

(265) $\mathtt{car} \to \mathtt{car}\sigma \to \mathtt{car}\pi \to \mathtt{cars}$

Doch nun zurück zu unserem alten Plan, die Pluralisierungsmaschine. Es gibt da Ausnahmen zu beachten. So wird der Plural auf /es/ gebildet, wenn das Wort /sh/ endet: /bushes/, /splashes/. Zuzüglich wird, wenn das Wort in /s/ endet, das /s/ verdoppelt; dann ist die Endung effektiv /ses/. Das ist so bei /busses/ und /plusses/. Dazu lassen wir die Maschine in verschiedene Zustände gehen, je nachdem, was für eine Endung das Wort hat.

(266)
$$\begin{array}{llll}
\langle 0, \mathtt{a}, 0, \mathtt{a}\rangle, & \langle 0, \mathtt{b}, 0, \mathtt{b}\rangle, & \ldots, & \langle 0, \mathtt{z}, 0, \mathtt{z}\rangle, \\
\langle 0, \mathtt{a}, 4, \mathtt{a}\rangle, & \ldots, & \langle 0, \mathtt{r}, 4, \mathtt{r}\rangle, & \langle 0, \mathtt{t}, 4, \mathtt{t}\rangle, \\
\ldots, & \langle 0, \mathtt{z}, 4, \mathtt{z}\rangle, & \langle 0, \mathtt{s}, 2, \mathtt{s}\rangle, & \langle 2, \mathtt{a}, 4, \mathtt{a}\rangle, \\
\ldots, & \langle 2, \mathtt{g}, 4, \mathtt{g}\rangle, & \langle 2, \mathtt{h}, 3, \mathtt{h}\rangle, & \langle 2, \mathtt{i}, 4, \mathtt{i}\rangle, \\
\ldots, & \langle 2, \mathtt{z}, 4, \mathtt{z}\rangle, & \langle 2, \varepsilon, 3, \mathtt{s}\rangle, & \langle 3, \varepsilon, 4, \mathtt{e}\rangle, \\
\langle 4, \sigma, 1, \varepsilon\rangle, & \langle 4, \pi, 1, \mathtt{s}\rangle. & &
\end{array}$$

Diese Maschine stelle ich noch einmal in abstrakter Form dar. Der einzige akzeptierende Zustand ist 1. Der Initialzustand ist 0. Man beachte auch, dass das Ausgabealphabet nicht identisch mit dem Eingabealphabet ist, was sich als vorteilhaft erweist.

(267)
$$\begin{array}{ll}
 & \{\langle 0, c, 0, c\rangle : c \in \{\mathtt{a}, \cdots, \mathtt{z}\} \cup \{\langle 0, c, 4, c\rangle : c \in \{\mathtt{a}, \cdots, \mathtt{r}, \mathtt{t}, \cdots, \mathtt{z}\} \\
\cup & \{\langle 0, \mathtt{s}, 2, \mathtt{s}\rangle, \langle 2, \mathtt{h}, 3, \mathtt{h}\rangle\} \cup \{\langle\langle 2, c, 4, c\rangle : c \in \mathtt{a}, \cdots, \mathtt{g}, \mathtt{i}, \cdots, \mathtt{z}\} \\
\cup & \{\langle 2, \varepsilon, 3, \mathtt{s}\rangle, \langle 3, \varepsilon, 4, \mathtt{e}\rangle, \langle 4, \sigma, 1, \varepsilon\rangle, \langle 4, \pi, 1, \mathtt{s}\rangle\}
\end{array}$$

Abstrakt sieht diese Menge so aus. Dies erschöpft keinesfalls die Komplikationen. So gibt es noch die Alternation /y/ – /ies/ (/crony/ – /cronies/, ist jedoch unwirksam nach einem Vokal: /alloy/ – /alloys/), /f/ – /v/ (welche allerdings nicht hundertprozentig ist): /leaf/ – /leaves/ (aber /belief/ – /beliefs/). Dann gibt es noch die unregelmäßigen Formen (/man/ – /men/, /mouse/ – /mice/) sowie einige importierte Pluralformen (/formula/ – /formulae/, /index/ – /indices/, /tableau/ – /tableaux/). Ist dies alles berücksichtigt, hat der Automat ein paar hundert Zustände!

Hier noch anderes Beispiel. Im Ungarischen haben die Kasussuffixe verschiedene Formen. Das Dativsuffix ist /nak/ oder /nek/. Die Form hängt vom Vokalismus des Wortes ab. Falls die Wurzel einen Hintervokal enthält (a, o, u) dann ist das Suffix /nak/; andernfalls ist es /nek/. Das Suffix für den Komitativ/Instrumental ist ein besonderer Fall: wenn es hinzefügt wird, wird der Endkonsonant verdoppelt, und dann ist das Suffix /al/, wenn das Wort einen Hintervokal enthält, und /el/ andernfalls. Falls das Wort in einem Vokal endet, dann ist das Suffix /val/ oder /vel/. (/dob/ (‘die Trommel’) – /dobnak/ (‘der Trommel’) – /dobbal/ (‘mit der Trommel’); /szeg/ (‘der Nagel’) – /szegnek/ (‘dem Nagel’) – /szeggel/ (‘mit einem Nagel’); /tó/ (‘der See’) – /tónak/ – /tóval/ (‘mit dem See’).) Wir können auch sagen, das Suffix sei /val/ bzw. /vel/, und dass es sich an den vorhergehenden Konsonanten assimiliert. Auf der anderen Seite gibt es eine Handvoll Worte, die mit /z/ enden (/ez/ ‘dies’, /az/ ‘das’) bei denen sich der Endkonsonant an den Konsonant der folgenden Endung assimiliert. Wir bekommen also /ennek/ (‘diesem’); im Komitativ aber bekommen wir /ezzel/ ‘hiermit’ oder auch /evvel/.

Die Pluralbildung im Deutschen ist im Vergleich zu Englischen ein Dschungel. Zunächst gibt es verschiedene Suffixe: /e/, /s/, /(e)n/, /(e)r/, und das leere Suffix. Dann gibt es noch den Umlaut, dh. den Wandel von /a/ zu /ä/, von /o/ zu /ö/ und von /u/ zu /ü/. Die folgende Tabelle gibt Beispiele.

(268)

Singular	Plural
Brot	Brote
Auto	Autos
Bude	Buden
Kind	Kinder
Leiter	Leiter
Vater	Väter
Tochter	Töchter
Mutter	Mütter

Und dann gibt es auch noch Kombinationen vom Umlaut und Suffix; und die unausweichlichen Sonderformen (/Indizes/, /Lemmata/ und so weiter). Was die Sache aber noch erheblich verkompliziert ist die Tatsache, dass es keine phonologische Regel gibt, die voraussagen könnte, wie der Plural gebildet werden muss. Sondern dies ist abhängig von dem Grundwort. Eine Maschine, die dies leistet, muss Hunderte, wenn nicht Tausende von Zuständen haben.

Transduktoren sind auch nützlich, um die Unebenheiten der Rechtschreibung wenigstens teilweise auszubügeln. So habe ich oben schon darauf hingewiesen, dass im Englischen nach Konsonant /y/ im Plural zu /ie/ wird. Dies ist im Englischen ein allgemeineres Phänomen. Es gilt auch für andere Endungen: /happy/ : /happier/, /fly/ : /flies/, /cry/ : /cried/. Im Ungarischen wird [dj] durch /gy/ wiedergegeben. Wird dieses Phonem verdoppelt, so schreibt man /ggy/ und nicht, wie man vermuten würde, /gygy/. So lautet der Instrumental von /hegy/ ‘Hügel’ denn auch /heggyel/ und man schreibt nicht etwa */hegygyel/. (Beim Zeilenumbruch bekommen wir aber genau die graphische Verdopplung: /hegy-/<newline>/gyel/.) Deswegen ist es ratsam, gleich zu Anfang einen Transduktor zu verwenden, der /ggy/ durch /gygy/ wiedergibt, damit wir wenige Umstände bei dem Morphemkombination machen müssen.

Zum Abschluss noch ein recht trickreiches Beispiel. Das Arabische hat, was man **nichtkonkatenative Morphologie** nennt. Das bedeutet, dass ein Morphem nicht einfach eine Zeichenkette ist, die irgendwo angehägt word. Im Ägyptischen Arabisch (wie in den anderen Arabischen Sprache auch) bestehen Wuzeln in der Regel aus 3 Konsonanten. So ist zum Beispiel /ktb/ ‘schreiben’ eine Wurzel, wie auch /drs/ ‘lernen’. Worte werden daraus auf recht kunstvolle Weise geformt. Es gibt Präfixe, Suffixe, Infixe, und Transfixe.

(269)

[katab]	‘er schrieb’	[daras]	‘er lernte’
[ʔamal]	‘er tat’	[naʿal]	‘er kopierte’
[baktib]	‘ich schreibe’	[badris]	‘ich lerne’
[baʔmil]	‘ich tue’	[banʿil]	‘ich kopiere’
[iktib]	‘schreibe!’	[idris]	‘lerne!’
[iʔmil]	‘tue!’	[inʿil]	‘kopiere!’
[kaatib]	‘Schreiber’	[daaris]	‘Schüler’
[ʔaamil]	‘Macher’	[naaʿil]	‘Kopierer’
[maktuub]	‘geschrieben’	[madruus]	‘gelernt’
[maʔmuu]	‘getan’	[manʿuul]	‘kopiert’

Nehmen wir nun an, wir wollen /katab/ morphologisch analsysieren. Dann wird

es zum Beispiel zu der Folge /ktb/ plus 3. Person plus Singular plus Vergangenheit. Ähnlich die anderen Wurzeln. Und es würde /baktib/ in /ktb/ plus 1. Person, plus Singular plus Präsens übersetzen. Und so weiter.

19 Transduktoren im Einsatz

Transduktoren sind nichtdeterministisch, aber im Gegensatz zu endlichen Automaten ist dies nicht immer zu ändern. Hier ist ein Beispiel. Der Automat hat die Zustände 0 und 1, 0 ist initial und 1 akzeptierend. Die Übergänge sind $0 \xrightarrow{\mathsf{a}:\mathsf{b}} 1$ und $1 \xrightarrow{\varepsilon:\mathsf{a}} 1$. Diese Machine akzeptiert als einzige Eingabe die Zeichenkette a und gibt eine beliebige Zeichenkette der Form ba^n, $n \in \mathbb{N}$, aus. Was für Läufe hat diese Maschine? Bei Eingabe a gibt es offenkundig unendlich viele Läufe.

(270)
$$0 \xrightarrow{\mathsf{a}:\mathsf{b}} 1$$
$$0 \xrightarrow{\mathsf{a}:\mathsf{b}} 1 \xrightarrow{\varepsilon:\mathsf{b}} 1$$
$$0 \xrightarrow{\mathsf{a}:\mathsf{b}} 1 \xrightarrow{\varepsilon:\mathsf{b}} 1 \xrightarrow{\varepsilon:\mathsf{b}} 1$$
$$0 \xrightarrow{\mathsf{a}:\mathsf{b}} 1 \xrightarrow{\varepsilon:\mathsf{b}} 1 \xrightarrow{\varepsilon:\mathsf{b}} 1 \xrightarrow{\varepsilon:\mathsf{b}} 1$$

Außerdem kann es verschiedene Läufe bei gegebener Ausgabe geben. Hier ist ein Beispiel. Wieder gibt es zwei Zustände, 0 (initial) und 1 (akzeptierend). Und es gibt die folgenden Übergänge: $0 \xrightarrow{\mathsf{a}:\mathsf{b}} 1$ sowie $1 \xrightarrow{\varepsilon:\mathsf{b}} 1$ und $1 \xrightarrow{\mathsf{a}:\varepsilon} 1$. Für das Paar aaa : bbbb gibt es mehrere Läufe.

(271)
$$0 \xrightarrow{\mathsf{a}:\mathsf{b}} 1 \xrightarrow{\mathsf{a}:\varepsilon} 1 \xrightarrow{\mathsf{a}:\varepsilon} 1 \xrightarrow{\varepsilon:\mathsf{b}} 1 \xrightarrow{\varepsilon:\mathsf{b}} 1 \xrightarrow{\varepsilon:\mathsf{b}} 1$$
$$0 \xrightarrow{\mathsf{a}:\mathsf{b}} 1 \xrightarrow{\mathsf{a}:\varepsilon} 1 \xrightarrow{\varepsilon:\mathsf{b}} 1 \xrightarrow{\mathsf{a}:\varepsilon} 1 \xrightarrow{\varepsilon:\mathsf{b}} 1 \xrightarrow{\varepsilon:\mathsf{b}} 1$$
$$0 \xrightarrow{\mathsf{a}:\mathsf{b}} 1 \xrightarrow{\varepsilon:\mathsf{b}} 1 \xrightarrow{\mathsf{a}:\varepsilon} 1 \xrightarrow{\mathsf{a}:\varepsilon} 1 \xrightarrow{\varepsilon:\mathsf{b}} 1 \xrightarrow{\varepsilon:\mathsf{b}} 1$$
$$0 \xrightarrow{\mathsf{a}:\mathsf{b}} 1 \xrightarrow{\varepsilon:\mathsf{b}} 1 \xrightarrow{\mathsf{a}:\varepsilon} 1 \xrightarrow{\varepsilon:\mathsf{b}} 1 \xrightarrow{\mathsf{a}:\varepsilon} 1 \xrightarrow{\varepsilon:\mathsf{b}} 1$$

(Es gibt sogar noch mehr Läufe als hier gezeigt.) Im Fall der hier gezeigten Maschine wächst die Zahl der Läufe mit der Länge des Eingabe/Ausgabe Paares an. Es gibt keine Hoffnung, die Anzahl der Läufe nach oben zu begrenzen.

Trotzdem möchten wir einen Algorithmus haben, der Antwort auf die folgenden Fragen gibt:

- ☞ Gegeben ein Transduktor $\mathfrak{T}$ und ein Paar $\vec{x} : \vec{y}$, gibt es einen akzeptierenden Lauf von $\mathfrak{T}$ für dieses Paar. (Das ist das Problem '$\langle \vec{x}, \vec{y} \rangle \in R_{\mathfrak{T}}$?')
- ☞ Gegeben $\vec{x}$ und $\mathfrak{T}$, gibt es eine Zeichenkette $\vec{y}$ derart, dass $\langle \vec{x}, \vec{y} \rangle \in R_{\mathfrak{T}}$?

☞ Gegeben $\vec{x}$, gibt es einen Weg, $R_{\mathfrak{T}}(\vec{x})$ geschlossen hinzuschreiben?

Wir werden diese Fragen nacheinander behandeln. Man kann auf zwei Weisen mit Nichtdeterminismus fertigwerden. Entweder folgen wir einem Lauf und machen nach Bedarf Backtracking (sogenannte **depth first Suche** oder **Tiefensuche**). Oder wir berechnen zuerst alle möglichen nächsten Schritte, bevor wir einen davon gehen. Dies ist die **breadth first Suche** oder **Breitensuche**.

Schauen wir uns am Beispiel der zweiten Maschine an. Wir nummerieren die Übergänge wie folgt.

(272) $t(0) = \langle 0, \mathtt{a}, 1, \mathtt{b}\rangle$

(273) $t(1) = \langle 1, \mathtt{a}, 1, \varepsilon\rangle$

(274) $t(2) = \langle 1, \varepsilon, 1, \mathtt{b}\rangle$

Diese Nummerierung benutzen wir, um die Übergänge zu ordnen. Wir beginnen mit dem Initialzustand 0. Gegeben sie das Paar `aaa : bbbb`. Zunächst ist kein Buchstabe gelesen. Nun suchen wir Übergänge, die von unserem Zustand losgehen und mit den Buchstaben kompatibel sind, die wir gerade betrachten. Da gibt es nur eine Möglichkeit: den Übergang $t(0)$. Dies ergibt, dass wir nunmehr im Zustand 1 sind, und das Paar `aa : bb` bearbeiten. Eine **Konfiguration** ist die Konjunktion folgender Tatsachen

1. die Maschine ist in Zustand q,
2. die Maschine ist an Position i auf der linken Zeichenkette, und
3. die Maschine ist an Position j auf der rechten Zeichenkette.

Eine Position in einer Zeichenkette kann man durch einen vertikalen Strich in der Zeichenkette bezeichnen. Ist die Maschine auf der linken Zeichenkette an Position 1 und auf der rechten an Position 1, und ist sie in Zustand 1, so schreiben wir $1, \mathtt{a|aa} : \mathtt{b|bbb}$. Der Strich wird unmittelbar nach dem letzten Symbol gesetzt, das schon gelesen wurde. Wäre die Maschine an Position 2 auf der rechten Zeichenkette, so schrieben wir stattdessen $1, \mathtt{a|aa} : \mathtt{bb|bb}$.

Als nächstes haben wir eine Wahlmöglichkeit: wir können enweder $t(1)$ nehmen und `a` lesen, oder wir nehmen $t(2)$ und lesen `b`. Wir nehmen den Übergang mit der kleineren Ordnungsnummer, in diesem Fall also $t(1)$. Die Konfiguration

sieht nun so aus: $1, \mathsf{aa}|\mathsf{a} : \mathsf{b}|\mathsf{bbb}$. Nun haben wir noch einmal dieselbe Wahl zwischen $t(1)$ und $t(2)$. Wiederum entscheiden wir uns für den Übergang $t(1)$; das gibt uns $1, \mathsf{aaa}| : \mathsf{b}|\mathsf{bbb}$. Jetzt gibt es keine Wahl mehr, wie können nur mit $t(2)$ weitergehen, was uns $1, \mathsf{aaa}| : \mathsf{bbb}|\mathsf{b}$ gibt; und dann noch einmal mit $t(2)$, und wir sind fertig. Dies ist der erste Lauf. Um nun die anderen Läufe zu bekommen, machen wir sogenanntes **Backtracking**. Wir gehen zurück zu der letzten Stelle, an der wir eine Wahl hatten und entscheiden uns für die andere Möglichkeit. Dies war bei der Konfiguration $1, \mathsf{aa}|\mathsf{a} : \mathsf{b}|\mathsf{bbb}$. Wir haben $t(1)$ gewählt, nund wählen wir stattdessen $t(2)$. Wir bekommen $1, \mathsf{aa}|\mathsf{a} : \mathsf{bb}|\mathsf{bb}$. Von hier haben wir nun wiederum eine Wahl, und wir folgen wieder dem Übergang mit der kleinsten Ordnungsnummer. Dies gibt uns den dritten Lauf. Wir gehen wieder zurück zum letzten Gabelpunkt und wählen nunmehr $t(2)$.

(275) $$0 \xrightarrow{\mathsf{a}:\mathsf{b}} 1 \xrightarrow{\mathsf{a}:\varepsilon} 1 \xrightarrow{\varepsilon:\mathsf{b}} 1 \xrightarrow{\varepsilon:\mathsf{b}} 1 \xrightarrow{\mathsf{a}:\varepsilon} 1 \xrightarrow{\varepsilon:\mathsf{b}} 1$$

Und so geht die Aufzählung der Läufe weiter:

(276)
$$0 \xrightarrow{\mathsf{a}:\mathsf{b}} 1 \xrightarrow{\mathsf{a}:\varepsilon} 1 \xrightarrow{\varepsilon:\mathsf{b}} 1 \xrightarrow{\varepsilon:\mathsf{b}} 1 \xrightarrow{\mathsf{a}:\varepsilon} 1 \xrightarrow{\varepsilon:\mathsf{b}} 1$$
$$0 \xrightarrow{\mathsf{a}:\mathsf{b}} 1 \xrightarrow{\mathsf{a}:\varepsilon} 1 \xrightarrow{\varepsilon:\mathsf{b}} 1 \xrightarrow{\varepsilon:\mathsf{b}} 1 \xrightarrow{\varepsilon:\mathsf{b}} 1 \xrightarrow{\mathsf{a}:\varepsilon} 1$$
$$0 \xrightarrow{\mathsf{a}:\mathsf{b}} 1 \xrightarrow{\varepsilon:\mathsf{b}} 1 \xrightarrow{\mathsf{a}:\varepsilon} 1 \xrightarrow{\mathsf{a}:\varepsilon} 1 \xrightarrow{\varepsilon:\mathsf{b}} 1 \xrightarrow{\varepsilon:\mathsf{b}} 1$$
$$0 \xrightarrow{\mathsf{a}:\mathsf{b}} 1 \xrightarrow{\varepsilon:\mathsf{b}} 1 \xrightarrow{\mathsf{a}:\varepsilon} 1 \xrightarrow{\varepsilon:\mathsf{b}} 1 \xrightarrow{\mathsf{a}:\varepsilon} 1 \xrightarrow{\varepsilon:\mathsf{b}} 1$$
$$0 \xrightarrow{\mathsf{a}:\mathsf{b}} 1 \xrightarrow{\varepsilon:\mathsf{b}} 1 \xrightarrow{\mathsf{a}:\varepsilon} 1 \xrightarrow{\varepsilon:\mathsf{b}} 1 \xrightarrow{\varepsilon:\mathsf{b}} 1 \xrightarrow{\mathsf{a}:\varepsilon} 1$$
$$0 \xrightarrow{\mathsf{a}:\mathsf{b}} 1 \xrightarrow{\varepsilon:\mathsf{b}} 1 \xrightarrow{\varepsilon:\mathsf{b}} 1 \xrightarrow{\mathsf{a}:\varepsilon} 1 \xrightarrow{\varepsilon:\mathsf{b}} 1 \xrightarrow{\mathsf{a}:\varepsilon} 1$$
$$0 \xrightarrow{\mathsf{a}:\mathsf{b}} 1 \xrightarrow{\varepsilon:\mathsf{b}} 1 \xrightarrow{\varepsilon:\mathsf{b}} 1 \xrightarrow{\varepsilon:\mathsf{b}} 1 \xrightarrow{\mathsf{a}:\varepsilon} 1 \xrightarrow{\mathsf{a}:\varepsilon} 1$$

Das allgemeine Rezept ist wie folgt. Jedesmal, wenn wir eine Wahl haben, nehmen wir den Übergang mit der kleinsten Ordnungsnummer. Wenn wir wieder an diesen Punkt zurückkommen, versuchen wir es mit der nächstkleineren Ordnungsnummer. Und wenn wir anschließend wiederkommen, nehmen wir die nächsthöhere Nummer, so lange, bis es keine Möglichkeiten mehr gibt; in diesem Fall gehen wir zu dem davorliegenden Gabelpunkt zurück.

Beim Backtracking ist zu beachten, dass es uns die Lösungen nicht sofort liefert sondern nach und nach. Wir müssen nur den letzten Lauf speichern. Dies erlaubt uns, den nächsten Lauf zu finden. Und wir wissen auch, wann es keinen

weiteren mehr gibt. Dies ist der Fall, wenn wir in jedem Punkt den Übergang mit der größtmöglichen Ordnungsnummer gewählt haben. Denn dann sind wir beim Backtracking ganz am Anfang angekommen, und es gibt keinen Übergang mit höherer Ordnunsgnummer, als den, den wir gerade probiert hatten.

Nun schauen wir uns die andere Strategie an. Sie besteht in der Aufzählung der partiellen Läufe. Ein partieller Lauf mit einem Paar Zeichenketten ist einfach eine Folge von Übergängen, die möglicherweise nicht vollendet wurde. Wir verlangen nicht, dass der partielle Lauf vollendet werden kann, sondern wüssten wir ja bereits, wie es weitergeht. Wir beginnen mit dem leeren Lauf. Im nächsten Schritt zählen wir alle Übergänge auf und erweitern die Läufe um einen Schritt. Es gibt nur eine Möglichkeit, $t(0)$, also ist $t(0)$ bereits alle Läufe der Länge ≤ 1. Damit kommen wir zur Konfiguration $1, \mathsf{a|aa} : \mathsf{b|bbb}$. Nun können wir $t(1)$ oder $t(2)$ wählen.

(277) Schritt 2

$t(0), t(1) : 1, \mathsf{aa|a} : \mathsf{b|bbb}$

$t(0), t(2) : 1, \mathsf{a|aa} : \mathsf{bb|bb}$

In der nächsten Runde erweitern wir die Ergebnisse wieder um einen Schritt. Wir können beide Läufe um $t(1)$ oder $t(2)$ erweitern.

(278) Schritt 3

$t(0), t(1), t(1) : 1, \mathsf{aaa|} : \mathsf{b|bbb}$

$t(0), t(1), t(2) : 1, \mathsf{aa|a} : \mathsf{bb|bb}$

$t(0), t(2), t(1) : 1, \mathsf{aa|a} : \mathsf{bb|bb}$

$t(0), t(2), t(2) : 1, \mathsf{a|aa} : \mathsf{bbb|b}$

Im vierten Schritt haben wir keine Wahl in der ersten Zeile: die Eingabe ist verbraucht, wir müssen $t(2)$ nehmen. In allen anderen Läufen sind beide Wege offen.

(279) Step 4

$t(0), t(1), t(1), t(2) : 1, \mathsf{aaa|} : \mathsf{bb|bb}$

$t(0), t(1), t(2), t(1) : 1, \mathsf{aaa|} : \mathsf{bb|bb}$

$t(0), t(1), t(2), t(2) : 1, \mathsf{aaa|} : \mathsf{bbb|b}$

$t(0), t(2), t(1), t(1) : 1, \mathsf{aaa|} : \mathsf{bbb|b}$

$t(0), t(2), t(1), t(2) : 1, \mathsf{aa}|\mathsf{a} : \mathsf{bbb}|\mathsf{b}$
$t(0), t(2), t(2), t(1) : 1, \mathsf{aa}|\mathsf{a} : \mathsf{bbb}|\mathsf{b}$
$t(0), t(2), t(2), t(2) : 1, \mathsf{a}|\mathsf{aa} : \mathsf{bbbb}|$

Und so weiter.

Man beachte, dass es nicht nötig ist, sich jeden Lauf einzeln zu merken. Wenn zwei Läufe in derselben Konfiguration enden, dann können sie auf die gleiche Weise fortgesetzt werden. Während also die Zahl der Läufe durchaus exponentiell sein kann, ist die Zahl der Konfigurationen propotional zum Produkt der Längen der der beiden Zeichenketten. (Dies liegt daran, dass wir eigentlich in jedem Schritt nur wählen können, ob wir die linken oder die rechte Zeichenkette lesen wollen. Daher das exponentielle Wachstum.)

Wir können den Algorithmus also wesentlich verbessern, indem wir uns nicht den Lauf merken sondern nur die resultierende Konfiguration. In jedem Schritt kalkulieren wir einfach die Nachfolgerkonfigurationen, und merken uns, wie man dahin von den existierenden Konfigurationen kommt. Mit jedem Schritt rücken wir die Positionsmarken mindestens einen Schritt auf einer Zeichenkette nach vorne, sodass wir in jedem Schritt dem Ende näherrücken. Wie lang braucht dieser Algorithmus? Konfigurationen sind Tripel $\langle x, j, k\rangle$, wo x ein Zustand des Automaten ist, i eine Position auf der linken Zeichenkette und k eine Position auf der rechten Zeichenkette. Gegeben $\vec{x} : \vec{y}$ gibt es also $|Q| \cdot |\vec{x}| \cdot |\vec{y}|$ viele solche Tripel. Alles, was der Algorithmus berechnet ist, welche Konfigurationen von $\langle i_0, 0, 0\rangle$ aus erreichbar sind. Dann schaut es, welche die Form $\langle q, |\vec{x}|\vec{y}|\rangle$ haben, wo $q \in F$. Dieser Algorithmus benötigt also höchstens $|Q| \cdot |\vec{x}| \cdot |\vec{y}|$ Schritte, dh. $O(|\vec{x}||\vec{y}|)$. (Man erinnere sich hier an die Diskussion in Sektion 15. Was wir hier effektiv berechnen ist die Erreichbarkeitsrelation in dem Raum der Konfigurationen. Dies kann man in linear Zeit tun, Zeit linear in der Größe des Raumes.)

Man mag denken, dass der erste Algorithmus schneller ist, weil er in einen Lauf relativ schnell einsteigt. Aber man kann zeigen, dass er unter ungünstigen Bedingungen tatsächlich exponentiell viele Schritte braucht. Dieser Algorithmus is also im schlechtesten Fall sehr langsam, kann aber unter günstigen Bedingungen natürlich auch schnell sein.

Jetzt noch zu dem dritten Problem: es sei $\vec{x}$ gegeben, was sind die $\vec{y}$ mit $\vec{x} R_{\mathfrak{T}} \vec{y}$? Wir haben schon gesehen, dass es unendlich viele $\vec{y}$ geben kann, sodass man diese nicht einfach aufzählen kann. Die Menge ist auf der anderen Seite regulär, also

kann man sie durch einen regulären Ausdruck beschreiben. Wir wollen dies hier nicht tun, sondern ein einfacheres Problem betrachten, nämlich überhaupt eine Zeichenkette $\vec{y}$ zu finden. Dazu zuerst einmal ein künstliches Beispiel. Der Automat habe die folgenden Übergänge.

(280)
$$0 \xrightarrow{\text{a:c}} 1 \qquad 0 \xrightarrow{\text{a:d}} 2$$
$$1 \xrightarrow{\text{a:c}} 1 \qquad 2 \xrightarrow{\text{a:d}} 2$$
$$2 \xrightarrow{\text{b:}\varepsilon} 3$$

0 sei initial, 1 und 3 akzeptierend. Die akzeptierten Eingabeketten sind von der Form $\mathtt{a}^+\mathtt{b}?$. Die definierte Relation ist wie folgt.

(281) $\{\langle \mathtt{a}^n, \mathtt{c}^n\rangle : n > 0\} \cup \{\langle \mathtt{a}^n\mathtt{b}, \mathtt{d}\rangle : n > 0\}$

Die Übersetzung einer gegebenen Zeichenkette ist eindeutig. Aber sie hängt von dem letzten Buchstaben der Zeichenkette ab. Wir können also den Automaten nicht einfach als eine Maschine denken, die von links nach rechts durchgeht und eine Übersetzung angibt. Denn der Automat ist nichtdeterministisch (als Akzeptor des Eingabe), und wir wissen nicht, welcher der Stränge letztlich der richtige ist.

Wir betrachten jetzt ein analoges Problem aus der natürlichen Sprache. Angenommen, wir bauen einen Transduktor für Arabisch. Und angenommen, dass die Oberflächenform `katab` in die Tiefenform `ktb+3Pf` übersetzt werden soll, hingegen `baktib` in die Form `ktb+1Pr`. Gegeben: Eingabe `baktib`. Die Konfiguration ist $0, |\mathtt{baktib}$. Keine Ausgabe. Der erste Buchstabe ist `b`. Wir haben zwei Möglichkeiten: wir betrachten es als Teil der Wurzel (etwas `badan`, welches in `bdn+3Pf` übersetzt werden muss), oder wir betrachten es als ersten Buchstaben des Präfixes `ba`. Bis hierher können wir nichts tun, um eine Möglichkeit auszuschließen. Wir halten sie beide offen. Erst wenn wir am vierten Buchstaben, `t`, ankommen, wird die Lage entschieden. Wenn dies eine 3. Person Perfekt Form ist, müsste jetzt `a` stehen. Wir schlagen hier also folgenden Algorithmus vor. Man versuche, einen Lauf zu finden, der die Eingabekette matcht, und schaue dann nach der Ausgabekette, die dadurch definiert wird.

20 Kontextfreie Grammatiken

Definition 20.1 *Es sei A ein Alphabet. Sei N eine Menge von Symbolen, disjunkt zu A, und sei $\Sigma \subseteq N$. Eine **kontextfreie Regel** mit N als nichtterminalem Alphabet und und A als **terminalem Alphabet** ist ein Paar $\langle X, \vec{\alpha}\rangle$, wo $X \in N$ und $\vec{\alpha} \in (N \cup A)^*$. Wir schreiben auch $X \to \vec{\alpha}$. Eine **kontextfreie Grammatik** oder auch **KFG** ist ein Quadrupel $\langle A, N, \Sigma, R\rangle$, wo R eine endliche Menge von kontextfreien Regeln ist. Die Menge Σ ist die Menge der **Startsymbole**.*

Es wird oft angenommen, dass eine kontextfreie Grammatik nur ein einziges Startsymbol hat, aber in der Praxis wird das regelmäßig ignoriert. Man beachte, dass die kontextfreien Regeln auf der rechten Seite den gemischten Gebrauch von terminalen und nichtterminalen Zeichen erlauben. Die Konvention ist dabei wie folgt. Eine Kette von Terminalzeichen wird durch einen kleinen lateinischen Buchstaben mit Vektorpfeil gekennzeichnet ($\vec{x}$), eine gemischte Kette durch eine kleinen griechischen Buchstaben mit Vektorpfeil ($\vec{\alpha}$). Eine Kette von Nichtterminalsymbolen wird demgegenüber durch einen großen lateinischen Buchstaben mit Vektorpfeil bezeichnet. Fällt der Vektorpfeil aus, so handelt es sich um ein Einzelzeichen.

Hier ein paar Beispiele von kontextfreien Regeln. Man beachte die Schreibweise mit $<\cdots>$.

(282)
<Ziffer> → 0
<Ziffer> → 1
<Zahl> → <Ziffer>
<Zahl> → <Zahl><Ziffer>

Um die Darstellung etwas kompakter zu machen, bedient man sich einiger Abkürzungen. Der vertikale Strich '|' wird benutzt, um Regeln mit gleicher linker Seite zusammenzufassen. Falls wir also die Regeln $X \to \vec{\alpha}$ und $X \to \vec{\gamma}$ haben, so können wir stattdessen auch $X \to \vec{\alpha} \mid \vec{\gamma}$ schreiben. Man nennt das letztere auch eine Regel, aber es sind im technischen Sinne zwei Regeln.

(283) <Ziffer> → 0 | 1 | 2 | . . . | 9

Das obere steht immerhin für zehn Regeln. Es ist dabei klar, dass echte Symbol in der Reihenfolge erscheinen, wie sie gegeben werden.

Eine kontextreie Grammatik kann man als Darstellung der Satzstruktur auffassen oder als System zur Erzeugung von Sätzen. Wir beginnen mit der zweiten Auffassung. Wir schreiben $\vec{\alpha} \Rightarrow \vec{\gamma}$ und sagen, dass $\vec{\gamma}$ von $\vec{\alpha}$ **in einem Schritt abgeleitet wird**, falls es eine Regel $X \to \vec{\eta}$ gibt und ein Vorkommen von X in $\vec{\alpha}$ derart, dass $\vec{\gamma}$ durch Ersetzen dieses Vorkommens von X in $\vec{\alpha}$ durch $\vec{\eta}$ entsteht:

(284) $\vec{\alpha} = \vec{\sigma}^\frown X^\frown \vec{\tau}, \qquad \vec{\gamma} = \vec{\sigma}^\frown \vec{\eta}^\frown \vec{\tau}$

Ist zum Beispiel die Regel <Zahl> → <Zahl><Ziffer>, dann ergibt das Ergebnis der Ersetzung des Vorkommens von <Zahl> in der Zeichenkette /1<Ziffer><Zahl>0/ gerade /1<Ziffer><Zahl><Ziffer>0/. Man schreibe $\vec{\alpha} \Rightarrow^{n+1} \vec{\gamma}$, falls es ein $\vec{\rho}$ gibt derart, dass $\vec{\alpha} \Rightarrow^n \vec{\rho}$ und $\vec{\rho} \Rightarrow \vec{\gamma}$. Wir sagen in diesem Fall, dass $\vec{\gamma}$ in **in** $n + 1$ **Schritten aus** $\vec{\alpha}$ **ableitbar ist**. Mit der obenstehenden Grammatik haben wir:

(285) <Zahl> $\Rightarrow^4$ <Ziffer><Ziffer><Ziffer><Zahl>

Endlich sei $\vec{\alpha} \Rightarrow^* \vec{\gamma}$ ($\vec{\gamma}$ **ist aus** $\vec{\alpha}$ **ableitbar**), falls es ein n gibt mit $\vec{\alpha} \Rightarrow^n \vec{\gamma}$. Falls $X \Rightarrow^* \vec{\gamma}$, so sagen wir, dass $\vec{\gamma}$ eine Zeichenkette der **Kategorie** X sei. Die **Sprache** (erzeugt) von G, in Zeichen $L(G)$, besteht aus allen *Terminalzeichenketten*, deren Kategorie aus Σ ist. Äquivalent dazu ist

(286) $L(G) := \{\vec{x} \in A^* : \text{ es gibt } S \in \Sigma : S \Rightarrow^* \vec{x}\}$

Man verifiziert leicht, dass die obenstehende Grammatik den Ketten von Ziffern die Kategorie <Zahl> gibt. Falls dies nun das Startsymbol ist, so besteht die Sprache aus allen Ziffernfolgen. Wenn man allerdings das Symobl <Ziffer> als Startsymbol nimmt, so bekommt man lediglich die Ziffern (genauer: Ziffernfolgen der Länge 1). Das ist deswegen so, weil man aus <Ziffer> nicht das Symbol <Zahl> bekommen kann. Falls allerdings Σ beide Symbole enthält, dann bekommt man wiederum alle Ziffernfolgen. (Die vierte Möglichkeit ist $\Sigma = \varnothing$; aber dann ist die Sprache leer.)

Es gibt einen Trick, um die Startsymbole auf eines zu reduzieren. Man führe eine neues Symbol S^* ein zusammen mit allen Regeln der Form

(287) $S^* \to S$

wo $S \in \Sigma$. Dies ist eine andere Grammatik, aber sie erzeugt die gleichen terminalen Zeichenketten. Ich habe schon oben erwähnt, dass man in der Linguistik gerne mehrere Startsymbole hat. Es gibt zum Beispiel mehrere Typen von Sätzen

(Fragen, Befehle, Wünsche, und so weiter), und wenn man diese durch Nichtterminalsymbole unterscheiden will, muss man jedem den Status eines Startsymbols geben. Das ist in realen Grammatiken (HPSG, GPSG, LFG, generative Grammatik) der Fall. Man hat auch die Situation, dass in der Sprache eigentlich viele Konstituenten als volle Äußerungen fungieren können (zum Beispiel als Antwort auf eine Frage). Aber diese sind keine Sätze im eigentlichen Sinne (ihnen kann zum Beispiel das finite Verb fehlen, wie in /Achtung!/). Deswegen ergibt es sich, dass man besser diese Konstituente als Startsymbol aufnimmt.

Eine **Ableitung** ist eine volle Spezifikation der Art und Weise, wie eine Zeichenkette in einer Grammatik abgeleitet wurde. Hier ist eine Variante. Eine Ableitung ist eine Folge von Zeichenketten, bei denen mit Ausnahme der letzten jeweils ein Symbol untertrichen ist. Das unterstrichene Symbol wird beim Übergang auf die nächste Kette ersetzt. Sei die folgende Grammatik gegeben.

(288) $\mathtt{A} \to \mathtt{BA} \mid \mathtt{AA}, \mathtt{B} \to \mathtt{AB} \mid \mathtt{BB}, \mathtt{A} \to \mathtt{a}, \mathtt{B} \to \mathtt{b} \mid \mathtt{bc} \mid \mathtt{cb}$

Dann ist dieses hier eine Ableitung:

(289) $\underline{\mathtt{A}}$, $\mathtt{B}\underline{\mathtt{A}}$, $\mathtt{BB}\underline{\mathtt{A}}$, $\mathtt{BBB}\underline{\mathtt{A}}$, $\mathtt{B}\underline{\mathtt{B}}\mathtt{Ba}$, $\underline{\mathtt{B}}\mathtt{cbBa}$, $\mathtt{bcb}\underline{\mathtt{B}}\mathtt{a}$, bcbbca

Man beachte, dass das Unterstreichen wirklich notwendig ist. Im zweiten Schritt hätten wir nämlich auch die Regel $\mathtt{A} \to \mathtt{BA}$ anwenden können, und dasselbe Resultat erzielt. Mit Unterstreichen hätte das dann so ausgesehen.

(290) $\underline{\mathtt{A}}$, $\mathtt{B}\underline{\mathtt{A}}$, $\mathtt{BB}\underline{\mathtt{A}}$, $\mathtt{BBB}\underline{\mathtt{A}}$, $\mathtt{B}\underline{\mathtt{B}}\mathtt{Ba}$, $\underline{\mathtt{B}}\mathtt{cbBa}$, $\mathtt{bcb}\underline{\mathtt{B}}\mathtt{a}$, bcbbca

Ohne Unterstreichen gäbe es keinen Unterschied. (Ich äußere mich nicht weiter, wie man anhand der Unterstreichung die Regel finden kann, die angewendet wurde. Das ist eigentlich ganz leicht.)

An dieser Stelle sollte ich noch einmal auf die Frage nach Vorkommen zurückkommen. Dies wird besonders wichtig, wenn wir uns der Perspektive auf KFGs als Strukturbeschreibungen zuwenden.

Definition 20.2 *Seien $\vec{x}$ und $\vec{y}$ Zeichenketten. Ein **Vorkommen** von $\vec{y}$ in $\vec{x}$ ist ein Paar $\langle \vec{u}, \vec{v} \rangle$ von Zeichenketten derart, dass $\vec{x} = \vec{u}\vec{y}\vec{v}$. Wir nennen $\vec{u}$ den **linken Kontext von $\vec{y}$ in $\vec{x}$** und $\vec{v}$ den **rechten Kontext**.*

Zum Beispiel hat die Zeichenkette aa drei Vorkommen in aacaaa: $\langle \varepsilon, \mathtt{caaa} \rangle$, $\langle \mathtt{aac}, \mathtt{a} \rangle$ und $\langle \mathtt{aaca}, \varepsilon \rangle$. Es ist *sehr* wichtig, zwischen einem Teilwort und seinem

Vorkommen zu unterscheiden. Wir können ein Vorkommen durch Unterstreichen kennzeichnen, oder durch andere Mittel. Beim Parsen ist es gewöhnlich so, dass man den Vorkommen in der Zeichenkette Zahlenpaare zuweist. Dabei zählt man die Positionen zwischen den Buchstaben durch. Jeder Buchstabe spannt dann die Position j bis $j + 1$. Ein Vorkommen des leeren Worts spannt die Vorkommen $\langle j, j\rangle$. Im allgemeinen ist also ein Vorkommen ein Zahlenpaar $\langle i, j\rangle$, wo $i \leq j$. (Ist $i = j$ so haben wir, wie gesagt, ein Vorkommen des leeren Worts.) Hier ist ein Beispiel.

(291)
```
    E     d     d     y
0     1     2     3     4
```

Das Paar $\langle 2, 3\rangle$ repräsentiert das Vorkommen $\langle \mathtt{Ed}, \mathtt{y}\rangle$ des Buchstabens (genauer: der Zeichenkette) /d/. Das Paar $\langle 1, 2\rangle$ repräsentiert das Vorkommen $\langle \mathtt{E}, \mathtt{dy}\rangle$. Das Paar $\langle 1, 4\rangle$ repräsentiert das (eindeutige) Vorkommen der Zeichenkette /ddy/. Und so weiter.

Kommen wir auf die Ableitungen zurück. Eine Ableitung kann mit einer beliebigen Zeichenkette beginnen, aber normalerweise beginnen wir mit einem Nichtterminalsymbol, besser einem Startsymbol. Sei nun die abgeleitete Kette $\vec{\gamma}$ (diese kann auch Nichtterminalzeichen enthalten). Wir sehen uns die Ableitung (289) an. Die letzte Zeichenkette ist /bcbbca/. Der letzte Schritt ist die Ersetzung von /B/ in /bcbBa/ durch /bc/. Diese Ersetzung bedeutet, dass das Vorkommen $\langle \mathtt{bcb}, \mathtt{a}\rangle$ der Zeichenkette /bc/ eine Konstituente der Kategorie B ist.

(292) bcbbca

Der Schritt davor war die Ersetzung des ersten Vorkommens von /B/ durch /b/. Also ist dieses erste Vorkommen von /b/ eine Konstituente der Kategorie /B/. Der drittletzte Schritt ist die Ersetzung des zweiten Vorkommens von /B/ durch /cb/, welches anzeigt, dass das Vorkommen $\langle \mathtt{b}, \mathtt{bca}\rangle$ eine Konstituente der Kategorie B ist. Dann folgt die Ersetzung von /A/ durch /a/. Wir haben also die folgenden Konstituenten des ersten Niveaus: $\langle \varepsilon, \mathtt{b}\rangle$, $\langle \mathtt{b}, \mathtt{bca}\rangle$, $\langle \mathtt{bcb}, \mathtt{a}\rangle$ und $\langle \mathtt{bcbbc}, \varepsilon\rangle$. Jetzt gehen wir zu der Ersetzung von /B/ durch /BB/ über. Nun entspricht das Vorkommen von /BB/ in der Zeichenkette der Folge von Vorkommen $\langle \mathtt{b}, \mathtt{bca}\rangle$ von /cb/ gefolgt von dem Vorkommen $\langle \mathtt{bcb}, \mathtt{a}\rangle$ von /bc/. Die Verkettung ist das Vorkommen $\langle \mathtt{b}, \mathtt{a}\rangle$ von /cbbc/.

(293) bcbbca

Wir gehen einen Schritt zurück und finden, dass jetzt das erste Vorkommen von /B/ durch /BB/ ersetzt wird, und die neue Konstituente, die wir bekommen, ist

(294) bcbbca

welche die Kategorie B hat. Im letzte Schritt bekommen wir, dass die gesamte Zeichenkette die Kategorie A hat. Das gilt natürlich nur für diese Ableitung. Also fügt jeder Schritt in der Ableitung eine neue Konstituente hinzu. Die Struktur, die wir bekommen, ist wie folgt. Gegeben eine Zeichenkette $\vec{\alpha}$ und eine Menge Γ von Vorkommen von Teilworten von $\vec{\alpha}$ zusammen mit einer Funktion f, die jedem Mitglied von Γ ein Nichtterminalsymbol zuweist. Wir nennen die Elemente von Γ **Konstituenten** und $f(C)$ die **Kategorie** von C, $C \in \Gamma$.

Definition 20.3 (Markiertes Vorkommen) *Ein **markiertes Vorkommen** von $\vec{x}$ in $\vec{y}$ ist ein Paar $\langle \mu, C \rangle$, wo C ein Vorkommen von $\vec{x}$ in $\vec{y}$ ist. μ heißt auch die **Marke** des Vorkommens.*

Wir setzen einer Ableitung nun eine Menge von (markierten) Vorkommen von Konstituenten an die Seite.

Definition 20.4 *Es sei G eine Grammatik und Π eine G-Ableitung von $\vec{x}$. Dann definiert Π eindeutig eine Konstituenstruktur auf $\vec{x}$. $\Pi_0 = \langle \underline{A} \rangle$. Dann sei $\kappa(\Pi_0) := \{\langle A, \langle \varepsilon, \varepsilon \rangle\rangle\}$. $\Pi_{n+1} := \Pi; \vec{y}\underline{Y}\vec{z}, \vec{x}_n$. Sei $K := \kappa(\Pi; \vec{y}\underline{Y}\underline{z})$. Dann ist $\vec{x}_n = \vec{y}\vec{u}\vec{z}$. und $\kappa(\Pi_{n+1}) := \{\langle Y, \langle \vec{y}, \vec{z} \rangle\rangle\} \cup K'$, wo K' aus K hervorgeht, indem Vorkommen der Form $\langle X, \langle \vec{y}Y\vec{u}, \vec{v} \rangle\rangle$ durch $\langle X, \langle \vec{y}\vec{x}\vec{u}, \vec{v} \rangle\rangle$ und Vorkommen der Form $\langle X, \langle \vec{v}, \vec{u}Y\vec{x} \rangle\rangle$ durch Vorkommen der Form $\langle X, \langle \vec{u}, \vec{v}\vec{x}\vec{z} \rangle\rangle$ ersetzt werden.*

Der Übergang von einer Ableitung zu einer Menge von Konstituenten ist unkritisch, wenn wir von Fällen absehen, wo aus A B abgeleitet werden kann und aus B wiederum A. Ich werde auf die Konstituentenmengen im nächsten Kapitel noch zurückkommen.

Zum Vergleich: die Ableitung (290) ergibt eine andere Konstituentenstruktur auf der Zeichenkette. Der Unterschied ist, dass nicht (294) eine Konstituente ist, sondern

(295) bcbbca

Es ist aber nicht so, dass die Konstituentenstruktur die Ableitung vollständig bestimmen. Hier ist zum Beispiel eine Ableitung, die dieselbe Konstituentenstruktur ergibt wie (289).

(296) A, BA, BBA, BBBA, BBBa, BBbca, bBbca, bcbbca

Der Unterschied zwischen (289) und (296) wird aber vernachlässigt. Im Wesentlichen sind nur die Konstituentenstrukturen relevant, weil Bedeutungen von der Konstituentanalyse errechnet werden.

Konstituentenstrukturen werden mit Hilfe von Bäumen dargestellt. Ein **Baum** ist ein Paar $\langle T, < \rangle$, wo $<$ eine transitive Relation ist (was bedeutet, dass, wenn $x < y$ und $y < z$ so auch $x < z$), sodass es eine **Wurzel** gibt (das ist ein Element x derart, dass für alle $y \neq x$: $y < x$) und schließlich gilt, dass, wenn $x < y, z$ so entweder $y < z$, $y = z$ oder $y > z$. Wir sagen, dass x y **dominiert**, falls $x > y$; und dass es y **ummittelbar dominiert**, wenn es kein z gibt, das von x dominiert wird und y dominiert.

Kommen wir noch auf die Konstituentenstruktur zurück. Seien $C = \langle \vec{\gamma}_1, \vec{\gamma}_2 \rangle$ und $D = \langle \vec{\eta}_1, \vec{\eta}_2 \rangle$ zwei Vorkommen von Teilworten. Wir sagen, dass C von D **dominiert wird**, in Zeichen $C \prec D$, falls $C \neq D$ und (1) $\vec{\eta}_1$ ist ein Präfix von $\vec{\gamma}_1$ und (2) $\vec{\eta}_2$ ist ein Suffix von $\vec{\gamma}_2$. (Es kann sein, dass $\vec{\eta}_1 = \vec{\gamma}_1$ oder dass $\vec{\eta}_2 = \vec{\gamma}_2$, aber nicht beides zugleich.) Graphisch gesehen sagt diese Definition, dass, wenn man die Vorkommen unterstreicht, dann enthält die Linie von D echt die Linie von C. Zum Beispiel sei $C = \langle \mathtt{abc}, \mathtt{dx} \rangle$ und $D = \langle \mathtt{a}, \mathtt{x} \rangle$. Dann ist $C \prec D$, denn sie sind verschieden, und (1) und (2) sind erfüllt.

(297) abcddx

Der Baum wird nun wie folgt definiert. Die Knotenmenge ist die Menge Γ. Die Relation $<$ ist nichts anderes als $\prec$.

Die Bäume in der linguistischen Analyse sind in aller Regel geordnet. Die Ordnung ist dabei meist implizit repräsentiert über die Zeichenkette. Ansonsten wird sie so definiert. Sei $C = \langle \vec{\gamma}_1, \vec{\gamma}_2 \rangle$ ein Vorkommen von $\vec{\sigma}$ und $D = \langle \vec{\eta}_1, \vec{\eta}_2 \rangle$ ein Vorkommen von $\vec{\tau}$. Wir schreiben $C \sqsubset D$ und sagen, C **gehe** D **voran**, falls $\vec{\gamma}_1 \vec{\sigma}$ ein Präfix von $\vec{\eta}_1$ ist (das Präfix muss nicht echt sein). Falls man wieder C und D durch Linien repräsentiert, so ist gleichbedeutend damit, dass die Linie von C endet, bevor die Linie von D beginnt.

(298) abccddx

Hier ist $C = \langle \mathtt{a}, \mathtt{cddx} \rangle$ und $D = \langle \mathtt{abcc}, \mathtt{x} \rangle$. Haben wir also eine Ableitung, so bekommen wir einen geordneten Baum als Strukturbaum. Diese können wir auch mi Hilfe von Klammern symbolisieren. Nehmen wir etwa die Ableitung (296). Diese bestimmt den folgenden Baum über /bcbbca/:

(299) ((((b)(cb))(bc))(a))

Zu guter Letzt bemerken wir noch, dass jede Konstitutente auch noch ein Nichtterminalsymbol zugewiesen bekommt:

(300) $((((\mathtt{b})_{\mathtt{B}}(\mathtt{cb})_{\mathtt{B}})_{\mathtt{B}}(\mathtt{bc})_{\mathtt{B}})_{\mathtt{B}}(\mathtt{a})_{\mathtt{A}})_{\mathtt{A}}$

Die endgültige Struktur ist diese:

Definition 20.5 *Ein geordneter, markierter Baum ist ein Quadrupel $\langle T, <, \sqsubset, \ell \rangle$, wo $\langle T, <, \sqsubset \rangle$ ein geordneter Baum ist und $\ell : T \to M$ eine Markierungsfunktion.*

Die Regeln der Grammatik sagen uns, wie der Baum beschaffen sein muss, damit dervon der Grammatik zugelassen wird.

Proposition 20.6 *Sei $G = \langle A, N, \Sigma, R \rangle$ eine kontextfreie Grammatik und $\mathfrak{T} = \langle T, <, \sqsubset, \ell \rangle$ ein geordneter, markierter Baum. $\mathfrak{T}$ gehört genau dann zu einer G-Ableitung, wenn Folgendes gilt.*

1. *$\ell(x) \in A$, wenn x Blatt des Baumes ist und $\ell(x) \in N$ sonst;*
2. *ist r die Wurzel, so ist $\ell(r) \in \Sigma$;*
3. *ist x nicht Blatt, und sind $y_1 \sqsubset y_2 \sqsubset \cdots \sqsubset y_n$ die Töchter von x, so ist*

   (301) $\ell(x) \to \ell(y_1)^\frown \ell(y_2)^\frown \cdots^\frown \ell(y_n) \in R$

21 Ambiguität

Es sei G eine Grammatik und $\vec{x}$ eine Zeichenkette, so kann es vorkommen, dass $\vec{x}$ mehrere Ableitungen besitzt. Dies haben wir schon im vorigen Kapitel besprochen. Und zwar ist dies immer dann der Fall, wenn wir eine Zeichenkette haben, auf die wir im Prinzip zwei verschiedene Regeln anwenden können. In diesem Fall können wir uns für eine der beiden Möglichkeiten entscheiden und die andere zu einem späteren Zeitpunkt nachholen. Der Strukturbaum ist allerdings derselbe.

Es gibt aber auch Fälle, wo eine Grammatik ein und dieselbe Zeichenkette auf mehrere Arten ableiten so lässt, dass verschiedene Strukturen entstehen. Hier ist ein recht banales Beispiel. $G = \langle\{\mathtt{a}\}, \{\mathtt{S}\}, S, \{\mathtt{S} \to \mathtt{SS} \mid \mathtt{a}\}\rangle$. Die Zeichenkette /aaa/ besitzt (unter anderem) die folgenden Ableitungen:

(302) $\underline{\mathtt{S}}, \mathtt{S}\underline{\mathtt{S}}, \mathtt{SS}\underline{\mathtt{S}}, \mathtt{a}\underline{\mathtt{S}}\mathtt{S}, \mathtt{aa}\underline{\mathtt{S}}, \mathtt{aaa}$

(303) $\underline{\mathtt{S}}, \mathtt{S}\underline{\mathtt{S}}, \mathtt{SS}\underline{\mathtt{S}}, \mathtt{S}\underline{\mathtt{S}}\mathtt{a}, \underline{\mathtt{S}}\mathtt{aa}, \mathtt{aaa}$

Dazu gehören die folgenden Strukturen:

(304) $(((\mathtt{a})_\mathtt{S}(\mathtt{a})_\mathtt{S})_\mathtt{S}(\mathtt{a})_\mathtt{S})_\mathtt{S}$

(305) $((\mathtt{a})_\mathtt{S}((\mathtt{a})_\mathtt{S}(\mathtt{a})_\mathtt{S})_\mathtt{S}$

Definition 21.1 (Ambiguität) *Es sei G eine Grammatik. G heißt **ambig**, falls es eine Zeichenkette $\vec{x} \in L(G)$ gibt, die von G zwei verschiedene Strukturbäume zugewiesen bekommt. Eine Grammatik, die nicht ambig ist heißt **eindeutig**.*

Zwei Strukturbäume sind auch dann verschieden, wenn ein Knoten eine anderes Nichtterminalsymbol trägt, auch wenn ansonsten alles gleich bleibt. Hier ist ein Beispiel. $G = \langle\{\mathtt{a}\}, \{\mathtt{S}, \mathtt{A}\}, \mathtt{S}, \{\mathtt{S} \to \mathtt{aS} \mid \mathtt{aA} \mid \mathtt{a}, \mathtt{A} \to \mathtt{a}\}\rangle$. Diese Grammatik erlaubt folgende Ableitungen von /aa/:

(306) $\underline{\mathtt{S}}, \mathtt{a}\underline{\mathtt{A}}, \mathtt{aa}$

(307) $\underline{\mathtt{S}}, \mathtt{a}\underline{\mathtt{S}}, \mathtt{aa}$

Dies ergibt folgende Strukturen

(308) $(\underline{\mathtt{a}}(\mathtt{a})_\mathtt{A})_\mathtt{S}$

(309) $(\underline{\mathtt{a}}(\mathtt{a})_\mathtt{S})_\mathtt{S}$

Ein noch strengerer Begriff als die Eindeutigkeit ist die Transparenz.

Definition 21.2 (Transparenz) *Eine Grammatik heißt **transparent**, falls Folgendes gilt: besitzt $\vec{x}$ ein Konstituentenvorkommen von $\vec{z}$ mit der Kategorie A, und ist $\vec{y} \in L(G)$ eine Zeichenkette, die $\vec{z}$ als Teilwort enthält, so enthält $\vec{y}$ in jeder G-Ableitung $\vec{z}$ als Konstituente.*

Um das verständlicher zu machen, hier noch ein wenig Terminologie. Es sei $\vec{x}$ eine Zeichenkette, die unter einer gegebenen Ableitung eine A-Konstituente $\vec{y}$ enthält. Ein Vorkommen von $\vec{y}$ in einer Ableitung ist **akzidentiell**, wenn es kein Konstituentemvorkommen ist. Eine Grammatik heißt demnach transparent, wenn für eine Zeichenkette $\vec{y}$ stets gilt: hat $\vec{y}$ ein Vorkommen als A-Konstituente, so gibt es kein akzidentielles Vorkommen von $\vec{y}$ und kein Vorkommen als B-Konstituente mit $B \neq A$.

Ein Beispiel mag hier helfen. Die folgende Grammatik ist transparent.

(310) $\mathtt{F} \to \mathtt{p} \mid \neg\mathtt{F} \mid \wedge\mathtt{FF} \mid \vee\mathtt{FF} \mid \to\mathtt{FF}$

Diese Notation heißt übrigens Polnische Notation. Sie erlaubt es, Formeln eindeutig ohne Klammern aufzuschreiben. Jede Teilkette, die aussieht, wie eine Formel, ist auch eine Teilformel. Diese Grammatik ist transparent ebenso wie diese hier:

(311) $\mathtt{F} \to \mathtt{p} \mid (\neg\mathtt{F}) \mid (\mathtt{F}\wedge\mathtt{F}) \mid (\mathtt{F}\vee\mathtt{F}) \mid (\mathtt{F}\to\mathtt{F})$

Versuchen wir zu verstehen, warum das so ist. Es ist klar, dass, wenn die Grammatik nicht transparent ist, es ein akzidentielles Vorkommen einer Zeichenkette gibt. Wir wählen $\vec{y}$ mit minimaler Länge derart, dass es ein $\vec{x}$ gibt zusammen mit einer Ableitung D, in der $\vec{y}$ akzidentiell auftritt. Es gibt zwei Fälle:

(Fall A) $\vec{y}$ enthält in D eine nichttriviale Konstituente.

(Fall B) $\vec{y}$ enthält in D keine nichttriviale Konstituente.

Wir zeigen, dass (Fall A) nicht eintritt. Sei also (Fall A) eingetreten. Dann enthält $\vec{y}$ eine Konstituente $\vec{v}$ der Form $(\neg\mathtt{p})$, $(\mathtt{p}\wedge\mathtt{p})$, $(\mathtt{p}\vee\mathtt{p})$ oder $(\mathtt{p}\to\mathtt{p})$. Wenn $\vec{y}$ als Konstituente auftritt, so sind die bezeichneten Vorkommen ebenfalls Vorkommen von Konstituenten, andernfalls ist ja $\vec{y}$ nicht minimal. Betrachten wir nun das akzidentielle Vorkommen von $\vec{y}$. Darin finden wir $\vec{v}$. Dieses Vorkommen von $\vec{v}$ ist nicht akzidentiell ($\vec{y}$ ist ja minimal), und so ist es ein Konstitutentenvorkommen. Wir nehmen nun unsere Zeichenkette und ersetzen das bezeichnete Vorkommen

von $\vec{v}$ durch p. Auf diese Weise bekommen wir ein Beispiel, das kleiner ist als $\vec{y}$. Aber $\vec{y}$ war von minimaler Länge. Widerspruch.

Also ist (Fall B) eingetreten. Jetzt beginnen wir, $\vec{y}$ wiederum zu verkleinern. Wiederum treten 2 Fälle ein.

(Fall C) $\vec{y}$ ist die rechte Seite einer Regel.

(Fall D) $\vec{y}$ ist nicht die rechte Seite einer Regel.

Wir werden zeigen, dass, wenn $\vec{y}$ minimal ist für ein (nicht unbedingt terminales) $\vec{x}$, dann ist (Fall C) eingetreten.

Wir betrachten anstelle von $\vec{y}$ das Resultat der Ersetzung von $\vec{v}$ duch F. Dieses ist kein Vorkommen von $\vec{y}$ in der Zeichenkette, sondern wir müssen in $\vec{x}$ das ausgezeichnete Vorkommen von p ebenfalls durch F ersetzen. Da nun $\vec{y}$ eine Konstituente ist, so hat es im (Fall C) entweder die Form (¬F), (F∧F), (F∨F) oder (F→F), oder im (Fall D) enthält $\vec{y}$ eine echte Teilkonstituente dieser Form. Sei (Fall D) eingetreten. Wir ersetzen diese echte Teilkonstituente in $\vec{x}$ (und $\vec{y}$) durch F, und bekommen so ein akzidentielles Vorkommen einer kleineren Konstituente. Das ist dasselbe Verfahren wie im vorigen Paragraphen, nur dass wir jetzt auch nichtterminale Zeichenketten $\vec{x}$ betrachten. (Fall D) ist nicht eingetreten.

Damit gilt Folgendes: *Wenn es ein akzidentielles Vorkommen einer Konstituente gibt, so gibt es in einer (nicht notwendig terminalen) ableitbaren Zeichenkette ein akzidentielles Vorkommen einer rechten Seite einer Regel.*

So weit die allgemeine Reduktion. Jetzt zeigen wir, dass für unsere vorliegende Grammatik auch dies nicht sein kann. Konstituenten können offensichtlich, wenn sie nicht die Form p haben, nur mit einer schließenden Klammer enden. Wenn aber $\vec{y}$ akzidentiell ist, so beginnt es mit einem echten Endstück einer Konstuente oder endet mit einem echten Anfangsstück einer Konstituente. Beginnt $\vec{y}$ mit einem echten Endstück $\vec{u}$ einer Konstituente so enthält $\vec{u}$ mehr schließende als öffnende Klammern die Klammerbilanz wird also mitten in $\vec{y}$ negativ, was nicht der Fall ist; ist es ein echtes Anfangsstück, so enthält dieses mehr öffnende als schließende Klammern, was ebenfalls nicht vorliegt. Auch (Fall C) tritt nicht ein, wir sind fertig.

Ich füge noch hinzu, dass die folgende Grammatik *nicht* transparent ist.

$$\begin{array}{ll} & \mathtt{D} \ \to \mathtt{0} \mid \mathtt{1} \mid \cdots \mid \mathtt{9} \\ (312) & \mathtt{I} \ \to \mathtt{DI} \mid \varepsilon \\ & \mathtt{F} \ \to \mathtt{pI} \mid \mathtt{(\neg F)} \mid \mathtt{(F \wedge F)} \mid \mathtt{(F \vee F)} \mid \mathtt{(F \to F)} \end{array}$$

Es ist sogar so, dass die erzeigte Sprache überhaupt keine transparente Grammatik besitzt. Hier ist ein vereinfachter Fall.

Satz 21.3 *Die Sprache* $\mathtt{a}^*$ *besitzt keine transparente Grammatik.*

Zum Beweis überlege man, dass es m und n geben muss mit $1 < m < n$ derart, dass $\mathtt{a}^m$ als Konstituente in $\mathtt{a}^n$ vorkommt. Sei dies das Vorkommen $\langle \mathtt{a}^k, \mathtt{a}^{n-k-m} \rangle$. Gewiss ist $0 \leq k \leq n - m$, also $0 \leq n - m - k$. Fall 1. $n - m - k > 0$. Dann betrachte das Vorkommen $\langle \mathtt{a}^{k+1}, \mathtt{a}^{n-m-(k-1)} \rangle$ von $\mathtt{a}^m$. Diese überlappt mit dem ersten Vorkommen und muss deswegen akzidentiell sein. Fall 2. $n - m - k = 0$. Dann ist $k > 0$. Dann betrachte das Vorkommen $\langle \mathtt{a}^{k-1}, \mathtt{a}^{n-m-(k-1)} \rangle$ von $\mathtt{a}^m$. Dieses überlappt mit dem ursprünglichen Vorkommen und ist deswegen akzidentiell.

Satz 21.4 *Es gibt Sprachen, die kontextfrei sind und keine eindeutige Grammatik besitzen.*

Ein Beispiel ist die Sprache

$$(313) \qquad \{\mathtt{a}^m\mathtt{b}^m\mathtt{c}^n : m, n \in \mathbb{N}\} \cup \{\mathtt{a}^m\mathtt{b}^n\mathtt{c}^n : m, n \in \mathbb{N}\}$$

Die Idee ist, dass das Wort $\mathtt{a}^k\mathtt{b}^k\mathtt{c}^k$ für großes k einerseits in einer Ableitung eine Konstituente aus vielen $\mathtt{a}$ und $\mathtt{b}$ besitzten muss sowie in einer Ableitung eine Konstituente aus vielen $\mathtt{b}$ und vielen $\mathtt{c}$, aber dass das nicht gleichzeitig, das heißt in derselben Ableitung, sein kann.

22 Parsing

Es sei eine Grammatik G gegeben und eine Zeichenkette $\vec{x}$. Wir möchten zu $\vec{x}$ eine Ableitung finden (oder einen Baum). Die Erfüllung dieser Aufgabe nennt man **Parsing**.

Ich möchte hier zunächst einmal den sogenannten Shift-Reduce Parser vorstellen. Im Grunde genommen ist dies eine einfache Angelegenheit. Der Parser geht mit einem Lesekopf (symbolisiert duch $\vdash$) durch die Zeichenkette von links nach rechts durch; die gelesenen Symbole werden auf einem Stapel abgelegt. Falls das obere Segment des Stapels die rechte Seite einer Regel darstellt, so kann dieser Teil vom Stapel genommen werden und durch das linke Symbol ersetzt werden. Dies nennt man einen **Reduktionsschritt**. Hier ist ein Beispiel.

(314) $\mathtt{F} \to \mathtt{p} \mid \neg\mathtt{F} \mid \wedge\mathtt{FF} \mid \vee\mathtt{FF} \mid \to\mathtt{FF}$

Es sei die Zeichenkette $\to\vee\mathtt{pp}\neg\mathtt{p}$ gegeben.

(315)

0		$\vdash\to\vee\mathtt{pp}\neg\mathtt{p}$
1	$\to$	$\vdash\vee\mathtt{pp}\neg\mathtt{p}$
2	$\to\vee$	$\vdash\mathtt{pp}\neg\mathtt{p}$
3	$\to\vee\mathtt{p}$	$\vdash\mathtt{p}\neg\mathtt{p}$
4	$\to\vee\mathtt{F}$	$\vdash\mathtt{p}\neg\mathtt{p}$
5	$\to\vee\mathtt{Fp}$	$\vdash\neg\mathtt{p}$
6	$\to\vee\mathtt{FF}$	$\vdash\neg\mathtt{p}$
7	$\to\mathtt{F}$	$\vdash\neg\mathtt{p}$
8	$\to\mathtt{F}\neg$	$\vdash\mathtt{p}$
9	$\to\mathtt{F}\neg\mathtt{p}$	$\vdash$
10	$\to\mathtt{F}\neg\mathtt{F}$	$\vdash$
11	$\to\mathtt{FF}$	$\vdash$
12	$\mathtt{F}$	$\vdash$

Die Schritte 4, 6, 7, 10, 11 und 12 sind Reduktionsschritte; die anderen nennt man **Shifts** (oder **Schiebeschritte**). Sie bestehen darin, den Lesekopf weiterzurücken. Die Zeichenkette ist in $L(G)$, wenn der Algorithmus mit dem Startsymbol aufhört. In diesem Fall kann man die Sequenz in umkehrter Richtung auch als Ableitung lesen (wobei das Dreibein weggelassen wird und die Shift-Schritte ausgelassen werden) und man bekommt daraus wiederum einen Baum. Hier ist die zu (315)

korrespondierende Ableitung.

(316)

12	F
11	→FF
10	→F¬F
9	→F¬p
6	→∨FF¬p
5	→∨Fp¬p
0	→∨pp¬p

Dies glückt allerdings nicht für jede Ableitung. Ich gebe hier einen missglückten Versuch des Parsers wieder.

(317)

0		⊢→∨pp¬p
1	→	⊢∨pp¬p
2	→∨	⊢pp¬p
3	→∨p	⊢p¬p
4	→∨pp	⊢¬p
5	→∨pp¬	⊢p
6	→∨pp¬p	⊢
7	→∨pp¬F	⊢
8	→∨ppF	⊢

Hier geht es nicht weiter, weil es nicht erlaubt ist, die Zeichenkette zurückzuschieben. Während sich links von dem Dreibein (⊢) ein Stapel befindet, ist rechts davon ein Stream: einmal gelesen, wird das Zeichen weggeworfen. In einer vereinfachten Version ist der Stapel ist das einzige Gedächtnis, von dem der Parser aber nur einen begrenzten Teil einsehen kann; ferner kann er nicht Symbole wegnehmen, um sie nachher wieder auf den Stapel zu tun. Genauer gibt es eine Zahl p derart, dass der Parser die obersten p Symbole des Stapels sehen kann. Findet er dort den rechten Teil einer Regel, kann er diese reduzieren, was bedeutet, dass er diese rechte Seite auf dem Stapel löscht und die linke Seite auf den Stapel setzt (also ein einziges Symbol). Alternativ dazu wird der Parser als endlicher Automat beschrieben, der nur einen Stapel und einen Stream manipuliert, von denen er Symbole wegnehmen kann, aber nur auf den Stapel kann er Symbole schreiben. Man kann zeigen, dass das endlicher Gedächtnis erlaubt, stets sich den obersten Teil des Stapels merken zu können.

Wenn man sich das Problem näher anschaut, so sieht man, dass der Parser immer versuchen sollte, Konstituenten bei ihrem ersten Auftreten zu reduzieren.

Man nennt die korrespondierenden Ableitungen auch rechtsseitige Ableitungen, weil sie zuerst die rechten Knoten expandieren. Da der Parser umgekehrt arbeitet, muss er also die am weitesten links stehende Konstituente zuerst reduzieren. Das bedeutet, dass er eine Konstituente so früh wie möglich reduziert. Wir formulieren daher folgende Strategie:

> Der Parser soll mit einer Regel reduzieren, sobald die rechte Seite auf dem Stapel erscheint.

Leider funktioniert diese Strategie nicht immer. Denn was sie voraussetzt, ist, dass das erste Vorkommen einer rechten Seite einer Regel nicht akzidentiell ist. Dazu ein Beispiel. Die folgende Grammatik erzeugt eine etwas realistischere Syntax für Aussagenlogik.

(318)
$$\begin{aligned} \texttt{D} &\to \texttt{0} \mid \texttt{1} \mid \cdots \mid \texttt{9} \\ \texttt{I} &\to \texttt{DI} \mid \texttt{D} \\ \texttt{F} &\to \texttt{p} \mid \texttt{pI} \mid \texttt{(¬F)} \mid \texttt{(F∧F)} \mid \texttt{(F∨F)} \mid \texttt{(F→F)} \end{aligned}$$

Wir betrachten die Kette `(¬p1)`. Die oben beschriebene Strategie funktioniert hier nicht:

(319)
```
0       ⊢(¬p1)
1  (    ⊢¬p1)
2  (¬   ⊢p1)
3  (¬p  ⊢1)
4  (¬F  ⊢1)
5  (F   ⊢1)
6  (F1  ⊢)
7  (F1)⊢
```

Ab hier kann der Parser weder schieben noch reduzieren. Da er nicht beim Startsymbol angekommen ist, wird die Zeichenkette abgelehnt.

Das Problem ist schnell erkannt: die Regeln $\texttt{F} \to \texttt{p}$ und $\texttt{F} \to \texttt{pI}$ konkurrieren miteinander, denn /p/ ist ein Anfangsstück von /pI/. Mit der Strategie, immer zuerst zu reduzieren, gewinnt stets die erste Regel über die zweite. Um solcherlei Probleme zu verhindern, erlaubt man dem Parser, ein Stück weit in die zu lesende Zeichenkette hineinzuschauen. In unserem Fall genügt es, das erste Zeichen des Streams zu kennen. Wir sagen, der Parser habe ein **Lookahead von 1**. Für die vorliegende Grammatik wählen wir folgende Strategie.

Sofern das erste zu lesende Symbol keine Zahl ist, soll der Parser wann immer möglich reduzieren, ansonsten schieben.

(320)
```
0      ⊢(¬p1)
1  (   ⊢¬p1)
2  (¬  ⊢p1)
3  (¬p ⊢1)
4  (¬p1⊢)
5  (¬pD⊢)
6  (¬pI⊢)
7  (¬F ⊢)
8  (¬F)⊢
9  F   ⊢
```

Wichtig ist der Schritt von 3 nach 4: da rechts neben dem Dreibein eine Ziffer steht, nämlich /1/, ist der Parser gehalten, diesmal nicht zu reduzieren sondern zu schieben.

Hier nun eine formale Definition. Diese folgt [7] und ist recht verschieden von der in Wikipedia. Es sei

$$N^{(p)} = \bigcup_{i<p+1} N^i. \tag{321}$$

Definition 22.1 (Shift-Reduce Parser I) *Es sei $G = \langle S, N, A, R\rangle$ eine kontextfreie Grammatik. Ein **Shift-Reduce Parser für G mit Lookahead 1** ist eine Funktion $f : N^{(p)} \times A \to \{\texttt{shift}\} \cup \{\texttt{reduce}(\rho) : \rho \in R\}$, wo p eine natürliche Zahl ist. Dabei ist $f(\sigma, x) = \texttt{reduce}(\rho)$ nur dann, wenn $\rho = A \to \vec{u}$ und $\vec{u}^T$ ein Suffix von σ ist.*

Eine Konfiguration ist ein Paar $(\sigma, \vec{y})$, notiert $\sigma \vdash \vec{y}$, wo $S \in (N \cup A)^*$ der Stapel und $\vec{y} \in A^*$ eine Zeichenkette. Die Anfangskonfiguration ist $\varepsilon \vdash \vec{x}$. Sei der Zustand $\sigma \vdash \vec{y}$, mit $\vec{y} = y_0 \cdots y_{k-1}$, $k > 0$, gegeben. Es sei σ_p definiert wie folgt. Hat σ mindestens die Länge p, so besteht σ_p aus den p obersten Symbolen. Ist die Länge kleiner als p, so sei $\sigma_p := \sigma$. Dann ist die nächste Konfiguration wie folgt: $\sigma; y_0 \vdash y_1 \cdots y_{k-1}$, falls $f(\sigma_p, y_0) = \texttt{shift}$ und $\sigma^\rho \vdash \vec{y}$, falls $f(\sigma_p, y_0) = \texttt{reduce}(\rho)$. Dabei entsteht σ^ρ aus σ, indem die Symbole der rechten

Seite von ρ aus σ entfernt werden und stattdessen die linke Seite von ρ auf den Stapel gepackt wird. Dabei haben wir in der Definition gefordert, dass die Regel auch auf den Stapel angewendet werden kann. Sonst würde der Parser an dieser Stelle scheitern. Es soll aber immer möglich sein, durch die Zeichenkette zu gehen. Allerdings akzeptiert der Parser die Zeichenkette dann (und nur dann), wenn am Ende der Ableitung $S \vdash \varepsilon$ steht.

Eine Grammatik heißt **LR(k)-Grammatik**, wenn es gelingt, eine Strategie für einen Shift-Reduce Parser zu entwerfen, bei dem die Entscheidung für den nächsten Schritt allein mit Hilfe eines Lookaheads von k Symbolen getroffen werden kann. Dieser heißt dann auch LR(k)-Parser.

Definition 22.2 (Shift-Reduce Parser II) *Es sei $G = \langle S, N, A, R\rangle$ eine kontextfreie Grammatik und $k \geq 0$. Ein **Shift-Reduce Parser für G mit Lookahead k** ist eine Funktion $f : N^{(p)} \times A^{(k)} \to \{\texttt{shift}\} \cup \{\texttt{reduce}(\rho) : \rho \in R\}$, wo p eine natürliche Zahl ist. Dabei ist $f(\sigma, x) = \texttt{reduce}(\rho)$ nur dann, wenn $\rho = A \to \vec{u}$ und $\vec{u}^T$ ein Suffix von σ ist.*

Die wichtigste Klasse ist die der LR(1)-Grammatiken. Es lässt sich nämlich zeigen, dass es zu jeder LR(k)-Grammatik für eine Sprache und $k > 1$ eine LR($k-1$)-Grammatik für ebendiese Sprache gibt. Der Parser für OCamL ist ein LR(1)-Parser. Ein Shift-Reduce Parser macht stets nur einen Durchlauf durch eine Zeichenkette. Falls die betrachtete Sprache eine LR(k)-Grammatik besitzt, ist die Entscheidung, ob eine Zeichenkette in der Sprache ist, in linearer Zeit möglich. Eine kontextfreie Sprache, für die ein LR(k) Parser existiert, heißt auch **deterministisch**. Falls eine Sprache keine LR(k)-Grammatik besitzt, also nicht deterministische ist, müssen andere Techniken verwendet werden.

Als zweite Methode stelle ich deshalb den Chart-Parser vor. Dieser funktioniert ohne Vorbedingungen an die Grammatik und liefert immer eine Ableitung bzw. eine Struktur. Dafür benötigt er etwas mehr Zeit (kubische Zeit, also $O(n^3)$, welche man etwas verbessen kann, siehe [5]). Und das geht so. Es sei eine Zeichenkette $x_0 x_1 \cdots x_{n-1}$ gegeben. Wir bilden eine Matrix $M(i, j)$, wo $0 \leq i < j \leq n$. Dabei sei $m(i, j)$ die Menge aller Nichtterminalsymbole A derart, dass $A \to x_i x_{i+1} \cdots x_{j-1}$. Die Zeichenkette ist genau dann in $L(G)$, wenn $S \in m(0, n)$. Die Matrixelemente $m(i, j)$ werden nun aufsteigend für $d := j - i$ berechnet:

Für d = 1 bis n:

Für i = 0 bis n-d:
Berechne $m(i, d)$.

Fragen wir uns nun, wie man $m(i, d)$ berechnen kann. Dazu sehen wir uns eine Regel an, sagen wir $A \to UaC$. Diese besagt, dass wir eine A-Konstituente von Position i bis Position j haben, sofern es eine U-Konstituente von i bis j' gibt, $x_{j'} = a$, und eine C-Konstituente von $j' + 1$ bis j. Dass heißt: ist $U \in m(i, j')$ für ein j' mit $i \leq j' \leq j$ und $x_{j'} = a$ sowie $C \in m(j' + 1, j)$, so tue A in $m(i, j)$. Dies sieht schon fast nach einem Algorithmus aus. Damit dies auch wirklich ein Berechnungsverfahren wird, stellen wir eine Bedingung: keine Regel soll die Form $X \to \varepsilon$ haben (dieser Fall lässt sich leicht eliminieren, er stellt keine wirkliche Beschränkung dar). Das bedeutet, dass Konstituenten nicht leer sind. In diesem Fall ist $i < j' < j$, und $m(i, j)$ wird konstruiert mit Hilfe der $m(i, j')$, $m(j' + 1, j)$, welche echt kleinere Länge haben. Wir vereinfachen die Grammatik insgesamt wie folgt:

Definition 22.3 *Eine Grammatik ist in **Chomsky-Normalform**, wenn alle Regeln die Form $X \to a$ (a terminal) oder $X \to Y_1 Y_2$ (Y_1, Y_2 nichtterminal) haben.*

Satz 22.4 *Sei G eine kontextfreie Grammatik mit $\varepsilon \notin L(G)$. Dann existiert eine kontextfreie Grammatik H in Chomsky-Normalform mit $L(H) = L(G)$.*

Für eine Grammatik in Normalform ist der Algorithmus einfach.

Für i = 0 bis n-1:
Für jede Regel $A \to x_i$ addiere A zu $m(i, i + 1)$.

Für d = 1 bis n:

Für i = 0 bis n-d-1:

Für e = i+1 bis n-d-1:
Für jede Regel $X \to YZ$ und $Y \in m(i, e)$, $Z \in m(e, i + d)$ addiere X zu $m(i, i + d)$.

23 Kategorialgrammatik

In diesem Kapitel werde ich einen speziellen Typ von Grammatik vorstellen, die Kategorialgrammatiken, genauer die AB-Kategorialgrammatiken, benannt nach Kazimiercz Ajdukiewicz und Yehoshua Bar-Hillel. Diese sind in einem gewissen Sinne gleichwertig mit kontextfreien Grammatiken: jede kontextfreie Sprache besitzt auch eine Kategorialgrammatik. Der Vorzug von Kategorialgrammatiken ist ihre enge Bindung zur Semantik, die wir anschließend beleuchten werden.

Definition 23.1 *Die Menge der **Kategorien über** B ist die kleinste Menge* $\mathrm{Kat}(B)$*, für die die gilt:*

- $B \subseteq \mathrm{Kat}(B)$*,*
- *ist* $\alpha, \beta \in \mathrm{Kat}(B)$*, so auch* $\alpha/\beta \in \mathrm{Kat}(B)$*,*
- *ist* $\alpha, \beta \in \mathrm{Kat}(B)$*, so auch* $\alpha\backslash\beta \in \mathrm{Kat}(B)$*.*

Auf der Menge der Kategorien erkläre ich eine Operation, $\cdot$, mit den folgenden Eigenschaften:

(322) $\quad \alpha/\beta \cdot \beta = \alpha, \qquad \beta \cdot \beta\backslash\alpha = \alpha$

Ferner verabreden wir Folgendes:

Definition 23.2 *Es sei* $\langle B, \sqsubset, <, \ell\rangle$ *ein binär verzweigender markierter, geordneter Baum mit Marken in* $\mathrm{Kat}(B)$*. Dieser ist **wohlmarkiert**, falls gilt: sind* x_1 *und* x_2 *die Töchter von* y *und* $x_1 \sqsubset x_2$*, dann gilt*

(323) $\quad \ell(y) = \ell(x_1) \cdot \ell(x_2)$

Für die Zwecke der folgenden Definition ist $\wp_f(M)$ die Menge der endlichen Teilmengen von M.

Definition 23.3 *Eine **AB-Kategorialgrammatik** ist ein Quadrupel* $K = \langle A, B, S, t\rangle$*, wo*

- *A eine endliche Menge ist (das **Alphabet**),*

- *B eine endliche Menge (die **Basiskategorien**),*
- *$S \in B$ (die **Zielkategorie**) und*
- $t : A \to \wp_f(\mathrm{Kat}(B))$.

Wir schreiben $K \vdash \vec{x}$, wenn es einen wohlmarkierten Baum gibt mit Endkette $\vec{x}$, dessen Wurzel die Marke S trägt und bei dem die Marke des i-ten Blattes aus $t(x_i)$ ist. Es ist ferner

(324) $L(K) := \{\vec{x} : K \vdash \vec{x}\}$

Hier ein Beispiel. Es sei $B = \{\mathrm{N}, \mathrm{C}, \mathrm{D}\}$. Die Zielkategorie ist C. Das Alphabet ist {der, Hase, Igel, rennt, schnelle, langsame}.

(325)

	t
der	D/N
schnelle	N/N
langsame	N/N
Hase	N
Igel	N
rennt	D\C
schläft	D\C

Schauen wir uns die folgende Zeichenkette an.

(326) der schnelle Igel rennt

Wir betrachten dies als eine Folge der Länge 4 über dem Alphabet. Ich annotiere in einem ersten Schritt die Worte mit den Kategorien, die ihnen laut Grammatik zukommen:

(327) $\mathtt{der}_{\mathrm{D/N}}$ $\mathtt{schnelle}_{\mathrm{N/N}}$ $\mathtt{Igel}_{\mathrm{N}}$ $\mathtt{rennt}_{\mathrm{D\backslash C}}$

Da $\mathrm{N/N} \cdot \mathrm{N} = \mathrm{N}$, haben wir eine Konstituente der Kategorie N, deren Töchter die mittleren beiden Konstituenten sind:

(328) $\mathtt{der}_{\mathrm{D/N}}$ $(\mathtt{schnelle}_{\mathrm{N/N}}$ $\mathtt{Igel}_{\mathrm{N}})_{\mathrm{N}}$ $\mathtt{rennt}_{\mathrm{D\backslash C}}$

Ebenso finden wir, dass die ersten beiden Konstituenten zusammen eine Konstituente der Kategorie D bilden:

(329) $(\texttt{der}_{D/N} \quad (\texttt{schnelle}_{N/N} \quad \texttt{Igel}_N)_N)_D \quad \texttt{rennt}_{D\backslash C}$

Und schließlich bildet diese zusammen mit dem Verb einen Satz:

(330) $((\texttt{der}_{D/N} \quad (\texttt{schnelle}_{N/N} \quad \texttt{Igel}_N)_N)_D \quad \texttt{rennt}_{D\backslash C})_C$

Daher ist $K \vdash$ /der schnelle Igel rennt/. Eine andere Struktur lässt sich für diese Kette nicht finden.

Diese Grammatik erzeugt unendlich viele Sätze. Denn Adjektive können in unbegrenzter Anzahl eingefügt werden:

(331)
der Igel rennt
der schnelle Igel rennt
der schnelle schnelle Igel rennt

Gehen wir noch einen Schritt weiter und fügen ein weiteres Wort hinzu: /und/, mit Kategorie (C\C)/C. Aus der Kategorie lässt sich erschließen, dass /und/ sein erstes Argument rechts bekommen möchte und sein zweites Argument links.

(332) $((\texttt{der Igel rennt})_C \quad (\texttt{und}_{(C\backslash C)/C} \quad (\texttt{der Hase schläft})_C)_{C\backslash C})_C$

Satz 23.4 *Für jede AB-Kategorialgrammatik K ist L(K) kontextfrei.*

Beweis. Sei $K = \langle A, B, S, t\rangle$. Es sei Sub($\alpha$) wie folgt definiert. Ist $\alpha \in B$, so Sub(α) := $\{\alpha\}$. Ferner sei

(333)
$$\begin{aligned} &\text{Sub}(\alpha/\beta) := \text{Sub}(\alpha) \cup \text{Sub}(\beta) \cup \{\alpha/\beta\} \\ &\text{Sub}(\beta\backslash\alpha) := \text{Sub}(\alpha) \cup \text{Sub}(\beta) \cup \{\beta\backslash\alpha\} \end{aligned}$$

Anschaulich gesprochen ist Sub(α) die Menge der Subkategorien von α. Nun setzen wir

(334) $$N := \bigcup_{\alpha \in t(x), x \in A} \text{Sub}(\alpha)$$

Ferner setzen wir

$$\begin{aligned} R := \quad & \{\alpha \to \vec{x} : \alpha \in t(\vec{x})\} \\ \cup \quad & \{\alpha \to \beta \quad \beta\backslash\alpha : \alpha, \beta \in N\} \\ \cup \quad & \{\alpha \to \alpha/\beta \quad \beta : \alpha, \beta \in N\} \end{aligned} \tag{335}$$

Dann sei $G := \langle S, N, A, R\rangle$. Ich behaupte, dass $L(K) = L(G)$. Dies ist recht einfach. Es gilt sogar Folgendes: für jeden geordneten markierten Baum $\mathfrak{B}$ gilt: genau dann ist $\mathfrak{B}$ ein Baum für G, wenn er ein Baum für K ist. Dies zeigt man leicht durch Induktion über die Höhe des Baumes. ⊣

Die Umkehrung gilt auch. Der vollständige Beweis ist recht lang, weswegen ich ein paar Schritte auslasse. Wichtig hierbei ist die sogenannte Greibachsche Normalform.

Definition 23.5 *Eine kontextfreie Grammatik ist in* ***Greibachscher Normalform*** *(oder kurz* ***Greibach-Form****), wenn alle Regeln die Form $X \to a\vec{Y}$ haben, wo $X \in N$, $a \in A$ und $\vec{Y} \in N^*$.*

Man beachte, dass die Regeln $X \to a$ auch dazugehören. (Man setze einfach $\vec{Y} := \varepsilon$.) Ohne Beweis zitiere ich den folgenden Sachverhalt.

Satz 23.6 *Zu jeder kontextfreien Sprache L existiert eine kontextfreie Grammatik G in Greibach-Form mit $L = L(G)$.*

Aus einer Grammatik in Greibach-Form lässt sich recht schnell eine äquivalente AB-Kategorialgrammatik basteln. Sei nämlich eine Regel $\rho = X \to aY_0Y_1 \cdots Y_{n-1}$. Wir setzen

$$c(\rho) := (\cdots((X/Y_0)/Y_1)/\cdots Y_{n-1}) \tag{336}$$

Es ist dann $B := N$, und

$$t(a) := \{c(\rho) : \rho = X \to a\vec{Y}\} \tag{337}$$

Auch hier lässt sich einfach zeigen, dass die Grammatiken die gleichen Zeichenketten erzeugen.

Satz 23.7 *Zu jeder kontextfreien Sprache L existiert eine AB-Kategorialgrammatik K mit $L = L(K)$.*

Korollar 23.8 *Zu jeder kontextfreien Grammatik G existiert eine AB-Kategorialgrammatik K mit $L(G) = L(K)$.*

Es sei angemerkt, dass die von der AB-Kategorialgrammatik erzeugten Bäume nicht unbedingt diejenigen der ursprünglichen Grammatik sind. Einerseits ist dies klar, wenn *G* nicht in Chomsky-Form ist (weil dies auf die AB-Kategorialgrammatiken immer zutrifft). Andererseits läuft die Konstruktion über die Greibach-Form, sie zerstört also erst einmal die Struktur der Grammatik, um dann in einem zweiten Schritt die Chomsky-Form in der Kategorialgrammatik wiederherzustellen.

In der Sprachwissenschaft hat sich ein etwas anderes Modell herausgebildet, nämlich die sogenannte X-quer-Syntax. Die X-quer-Syntax unterscheidet zwischen **Köpfen** und **Phrase**. Köpfe sind jeweils lexikalische Elemente, während Phrasen es nicht sind, ausgenommen sie bestehen aus einem einzigen Kopf. Die Grundidee ist, dass der Kopf den Aufbau einer Phrase bestimmt. Ein Beispiel hilft, dies zu erläutern. Ein Verb erzeugt eine Verbalphrase. Die Kategorie des Verbs nennt man V, die der zugehörigen Phrase VP. Das Verb bestimmt nun Art und Anzahl der Argumente, die es zum Aufbau der Phrase benötigt. Transitive Verben benötigen zum Beispiel immer ein Akkusativobjekt (das ist definitionsgemäß so). Diese können auch Sätze sein (/`wissen`/). Intransitive Verben benötigen entweder nur ein Subjekt (/`laufen`/), ein Präpositionalobjekt (/`warten`/), oder ein Dativobjekt (/`helfen`/). Verben können gar keine Argumente benötigen (/`regnen`/), nur eines (/`laufen`/), zwei (/`zumuten`/) oder gar drei (/`wetten`/). Außer Argumenten gibt es noch **Adjunkte**. Diese sind der Art und Anzahl nach frei. Hierzu gehören Modifikatoren, also zum Beispiel Adjektive und Adverbien. Diese Syntax ist nicht unähnlich der Kategorialgrammatik. Auch in der Kategorialgrammatik bestimmt ein lexikalisches Element über seine Kategorie Art und Anzahl der Elemente, mit denen es kombinieren kann. Hat x die Kategorie X/Y, so kombiniert es mit einer Konstituente der Kategorie Y, um eine Konstituente der Kategorie X zu bilden. Es gibt allerdings auch Unterschiede, denn es ist nicht von vornherein klar, ob x nicht auch Argument sein kann (etwa eines Elements der Kategorie $(X/Y)\backslash Z$).

Ein anderes Phänomen ist dagegen nicht so einfach zu handhaben: die **Kongruenz**. Im Deutschen (wie in vielen anderen Sprachen) besteht Kongruenz, und zwar zwischen dem Artikel, dem Adjektiv und dem Nomen in Kasus, Numerus und Genus, während das Verb mit seinem Subjekt in Numerus und Person kongruiert. Zusätzlich gibt es noch innerhalb einer Nominalphrase unterschiedliche Formen des Adjektivs, je nachdem, welcher Artikel gewählt wird (/`ein schneller`

Igel/ aber /der schnelle Igel/). Ich betrachte im Folgenden nur Kongruenz bezüglich Kasus, Numerus und Person.

(338a) der schnelle Igel

(338b) *der schnellen Igel

(338c) *der schnelles Igel

(338d) Ich lese.

(338e) *Ich liest.

Die einfachste Art, Kongruenz in der Grammatik zu berücksichtigen, ist die Ausdifferenzierung der Kategorien. Anstelle von N werden wir jetzt Kategorien $N_{dat,sg,3}$ haben (was für ein Nomen der 3. Person im Dativ Singular steht). Ein Adjektiv, welches ein solches Nomen modifiziert, hat dann die Kategorie $N_{dat,sg,3}/N_{dat,sg,3}$.

(339)

	t
der	$D_{nom,sg,3}/N_{nom,sg,3}$
schnelle	$N_{nom,sg,3}/N_{nom,sg,3}$
langsame	$N_{nom,sg,3}/N_{nom,sg,3}$
Hase	$N_{nom,sg,3}$
Igel	$N_{nom,sg,3}$
rennt	$D_{nom,sg,3}\backslash C$
schläft	$D_{nom,sg,3}\backslash C$

Ich füge der Grammatik noch Pronomina hinzu:

(340)

	t
ich	$D_{nom,sg,1}$
du	$D_{nom,sg,2}$
er	$D_{nom,sg,3}$
sie	$D_{nom,sg,3}, D_{nom,pl,3}$
es	$D_{nom,sg,3}$
wir	$D_{nom,pl,1}$
ihr	$D_{nom,pl,2}$

Man beachte, dass das Pronomen /sie/ zwei Kategorien besitzt. Zusätzlich können wir die Formen der Verben jetzt aufnehmen.

(341)

	t
schlafe	$D_{nom,sg,1}\backslash C$
schläfst	$D_{nom,sg,2}\backslash C$
schläft	$D_{nom,sg,3}\backslash C$
schlafen	$D_{nom,pl,1}\backslash C, D_{nom,pl,3}\backslash C$
schlaft	$D_{nom,pl,2}\backslash C$

Schauen wir uns nun noch an, was passiert, wenn wir transitive Verben aufnehmen, etwa /sehen/. Es hat zwei Argumente, von denen das erste das transitive Objekt ist und das zweite das Nominativsubjekt. Das transitive Verb zeigt Kongruenz nur mit dem Subjekt. Deswegen hat es recht viele Kategorien. Ich zeige nur eine Verbform.

(342)

sehe	$(D_{nom,sg,1}\backslash C)/D_{akk,sg,1}, (D_{nom,sg,1}\backslash C)/D_{akk,sg,2},$ $(D_{nom,sg,1}\backslash C)/D_{akk,sg,3}, (D_{nom,sg,1}\backslash C)/D_{akk,pl,1},$ $(D_{nom,sg,1}\backslash C)/D_{akk,pl,2}, (D_{nom,sg,1}\backslash C)/D_{akk,pl,3}.$

Typischerwie behilft man sich hier mit einem Trick und führt statt der konkreten Kategorie abstrakte Kategorien ein, indem einige Subskripte durch Variable (hier ν und π) ersetzt werden.

(343)

sehe	$(D_{nom,sg,1}\backslash C)/D_{akk,\nu,\pi}$
siehst	$(D_{nom,sg,2}\backslash C)/D_{akk,\nu,\pi}$
sieht	$(D_{nom,sg,3}\backslash C)/D_{akk,\nu,\pi}$
sehen	$(D_{nom,pl,1}\backslash C)/D_{akk,\nu,\pi}, (D_{nom,pl,3}\backslash C)/D_{akk,\nu,\pi}$
seht	$(D_{nom,pl,3}\backslash C)/D_{akk,\nu,\pi}$

24 Semantik der Kategorialgrammatiken

Nachdem wir die Kategorien besprochen haben, wenden wir uns nun der Semantik zu. Man beobachtet, dass Elemente der gleichen syntaktischen Kategorie irgendwie auch semantisch ähnlich sind. Ein Nomen, zum Beispie, bezeichnet eine Eigenschaft von Individuen. Dazu gehören /Haus/, /Katze/ oder /Auto/. Diese bezeichnen die Eigenschaft, ein Haus, eine Katze bzw. ein Auto zu sein. Semantisch sind sie Funktionen von Individuen nach Wahrheitswerten (siehe Kapitel 8). Der semantische Typ ist aber nicht nur von seiner syntaktischen Klasse abhängig (Adjektiv, Verb, Artikel, und so weiter), sondern auch von der Anzahl und Art der Argumente, die es benötigt. So gibt es neben den obenstehenden Nomina auch noch sogenannte Relationalnomina (/Nachbar/, /Präsident/, /Besitzer/). Wir erkennen die zusätzliche semantische Komplikation an den PP-Komplementen (hier in Gestalt eines Genitivs oder einer PP, die mit /von/ eingeleitet wird).

(344a) der Nachbar von Johann

(344b) der Präsident von Frankreich

(344c) der Besitzer dieses Autos

Viele Autoren haben die enge Beziehung zwischen der syntaktischen Form und der Semantik erkannt. Montague hat diese in folgendes formales Prinzip gegossen. Die Abbildung τ von Kategorien zu Typen muss folgende Eigenschaft haben.

(345) $\tau(\alpha/\beta) = \tau(\beta\backslash\alpha) = \tau(\beta) \to \tau(\alpha)$

Dies sollte ich etwas qualifizieren. In Wirklichkeit verlangen wir $\tau(\alpha/\beta) = \tau(\beta\backslash\alpha) = f(\tau(\alpha), \tau(\beta))$ für eine zweistellige Funktion auf den Typen. So wird Montague folgend in [2] vorgeschlagen, dass $f(\gamma, \delta) = (s \to \delta) \to \gamma$, wobei s der Typ der möglichen Welten ist. Da wir aber ein extensionales Fragment haben, können wir diese Komplikation ignorieren.

(345) besagt, dass einzig die Zuordnung von Typen zu Basiskategorien frei ist; der einer komplexen Kategorie zugeordnete Typ ist dann festgelegt. Ich gebe die Zuordnung für die Grammatik des letzten Kapitels an:

(346)

α	$\tau(\alpha)$
D	e
N	$e \to t$
C	t

Daraus folgt

(347a) $\tau(N/N) = (e \to t) \to (e \to t)$

(347b) $\tau(D\backslash C) = e \to t$

(347c) $\tau(D/N) = (e \to t) \to e$

Dies wird noch zu modifizieren sein. Man beachte im Übrigen, dass das Bild einer Basiskategorie durchaus ein komplexer Typ sein darf.

Gehen wir noch einmal auf die Darstellung in Kapitel 8 zurück. Zu analysieren ist der Satz (102).

(348) `Johann singt.`

Dazu benutzen wir folgende Grammatik.

(349)

	t
`Johann`	D
`singt`	$D\backslash C$

Die Zielkategorie ist C. In der Tat ist, wie die Grammatik vorhersagt, (348) grammatisch, (350) aber nicht (was wir nicht aus den Typen ablesen können).

(350) *`Singt Johann.`

Die Bedeutung des Satzes wird nun wie folgt bestimmt.

> **(Vorwärtsapplikation)** Ist $\vec{x}$ eine Konstituente der Kategorie α/β und Bedeutung f, und ist $\vec{y}$ eine Konstituente der Kategorie β und Bedeutung x, so ist $\vec{x}\vec{y}$ eine Konstituente der Kategorie α und Bedeutung $f(x)$.

> **(Rückwärtsapplikation)** Ist $\vec{x}$ eine Konstituente der Kategorie $\beta\backslash\alpha$ und Bedeutung f, und ist $\vec{y}$ eine Konstituente der Kategorie β und Bedeutung x, so ist $\vec{y}\vec{x}$ eine Konstituente der Kategorie α und Bedeutung $f(x)$.

Wir vereinen dies in einen allgemeinen Kalkül, der Tripel $\langle \vec{x}, \alpha, M\rangle$ manipuliert. Zunächst ein paar Definitionen. Für Mengen M und N sei M^N nichts weiter als die Menge aller Funktionen von N nach M. Man vergleiche nun die folgende Definition mit (16).

Definition 24.1 *Es sei C eine Menge von Typen. Ein* ***Universum*** *über C ist eine Familie* $\mathcal{M} = \langle M_\alpha : \alpha \in \mathrm{Typ}_\rightarrow(C)\rangle$ *mit folgenden Eigenschaften.*

1. $M_\alpha \cap M_\beta = \emptyset$, *wenn* $\alpha \neq \beta$;
2. $M_{\alpha\rightarrow\beta} = M_\beta^{(M_\alpha)}$ *für alle* $\alpha, \beta \in \mathrm{Typ}_\rightarrow(C)$.

Ein Universum über den Basistypen aus C ist eindeutig durch die Mengen M_α für $\alpha \in C$ bestimmt.

Definition 24.2 *Eine* ***interpretierte AB-Kategorialgrammatik*** *ist ein Septupel*

(351) $\langle A, B, S, C, \mathcal{M}, \tau, L\rangle$,

wo A eine endliche Menge ist (das Alphabet), B eine endliche Menge (die Menge der Basiskategorien), $S \in B$, *C eine endliche Menge (die Menge der Basistypen),* $\mathcal{M}$ *ein Universum über* $\mathrm{Typ}_\rightarrow(C)$, τ *eine Typenabbildung und* $L \subseteq A \times \mathrm{Kat}(B) \times M$ *ebenfalls eine endliche Menge, das* ***Lexikon****. Dabei gilt für alle* $\langle \vec{x}, \alpha, m\rangle$, *dass* $m \in M_{\tau(\alpha)}$, *das heißt, dass der Typ von m genau* $\tau(\alpha)$ *ist.*

Man könnte in dieser Definition einiges implizit lassen. Wichtig sind hier vor allem L (das Lexikon) sowie S, die Zielkategorie. Mit Hilfe der beiden Regeln können wir wie folgt definieren.

Definition 24.3 *Ein* ***Zeichen*** *ist ein Tripel* $\langle \vec{x}, \alpha, m\rangle$, *wo* $\vec{x} \in A^*$, $\alpha \in \mathrm{Kat}(B)$ *und* $m \in M$. *Sei K eine interpretierte AB-Kategorialgrammatik. Es ist dann* $\Sigma(K)$ *die kleinste Menge von Zeichen, die Folgendes erfüllt.*

1. $L \subseteq \Sigma(K)$.
2. *Ist* $\langle \vec{x}, \alpha/\beta, m\rangle, \langle \vec{y}, \beta, n\rangle \in \Sigma(K)$, *so* $\langle \vec{x}\vec{y}, \alpha, m(n)\rangle \in \Sigma(K)$.
3. *Ist* $\langle \vec{x}, \beta\backslash\alpha, m\rangle, \langle \vec{y}, \beta, n\rangle \in \Sigma(K)$, *so* $\langle \vec{y}\vec{x}, \alpha, m(n)\rangle \in \Sigma(K)$.

Sei also M wie folgt. Sei $M_e = \{i, j, k, \ell\}$, $M_t = \{0, 1\}$. Dann stehen die Mengen M_α für komplexe Typen fest und wir haben ein Universum. Jetzt legen wir das Lexikon fest.

(352) $L := \{\langle \texttt{Johann}, D, i\rangle, \langle \texttt{singt}, D\backslash C, \{\langle i, 1\rangle, \langle j, 1\rangle, \langle k, 0\rangle, \langle \ell, 1\rangle\}\rangle\}$

Mit Hilfe der obenstehenden Regeln leiten wir ab:

(353) $\langle \texttt{Johann singt}, C, 1\rangle \in \Sigma(K)$

Ein Satz $\vec{x}$ ist wahr, wenn $\langle \vec{x}, C, 1\rangle \in \Sigma(K)$.

Betrachten wir nun noch einige andere Wortarten. Da wären zunächst die Nomina. Diese haben wie gesagt die Kategorie N, und ihr wird der Typ $e \to t$ zugeordnet, also Eigenschaften von Individuen. Sei also

(354a) $\langle \texttt{Igel}, N, \{\langle i, 0\rangle, \langle j, 1\rangle, \langle k, 1\rangle, \langle \ell, 0\rangle\}\rangle$,

(354b) $\langle \texttt{Hase}, N, \{\langle i, 0\rangle, \langle j, 0\rangle, \langle k, 0\rangle, \langle \ell, 1\rangle\}\rangle$,

neue Elemente des Lexikons. Ferner sei

(355) $\langle \texttt{der}, D/N, \iota\rangle \in L$

Hierbei ist $\iota(f) = a$, sofern a das einzige x ist mit $f(x) = 1$. Ansonsten sei $\iota(f) = i$. (Eigentlich lässt man $\iota(f)$ in diesem Fall undefiniert, aber die Definition des Universums erlaubt das nicht. Ich müsste ansonsten partielle Funktionen zulassen, was zumindest in OCaml problemlos geht.) Man überlege sich, dass damit

(356) $\langle \texttt{der Hase}, D, \ell\rangle, \langle \texttt{der Igel}, D, i\rangle \in \Sigma(K)$

Ferner bekommen wir

(357) $\langle \texttt{der Hase singt}, C, 1\rangle \in \Sigma(K)$

Adjektive haben die Kategorie N/N und so den Typ $(e \to t) \to (e \to t)$.

Wenden wir uns nun der Quantifikation zu. Der Satz

(358) `Jeder Hase singt.`

ist wahr genau dann, wenn jedes Individuum, welches ein Hase ist auch singt. Es hat sich herausgestellt, dass die Typzuweisung von e für D nicht geeignet ist. Stattdessen hätte man lieber die folgende Zuweisung.

(359)

α	$\tau(\alpha)$
D	$(e \to t) \to t$
N	$e \to t$
C	t

In diesem Fall aber kann das Verb sein Subjekt nicht als Argument nehmen. Wir bleiben also bei der alten Zuweisung und geben stattdessen den Determinatoren eine andere Kategorie. So haben wir zum Beispiel das folgende Element.

(360) $\langle \texttt{jeder}, (C/(D\backslash C))/N, \forall \rangle$

(361) $$\forall PQ = \begin{cases} 1 & \text{für alle } a\text{: } P(a) \rightarrow Q(a) \\ 0 & \text{für ein } a\text{: } P(a) \wedge \neg Q(a) \end{cases}$$

Setze $\mathsf{igel}' := \{\langle i, 0\rangle, \langle j, 1\rangle, \langle k, 1\rangle, \langle \ell, 0\rangle\}$ und $\mathsf{singt}' := \{\langle i, 1\rangle, \langle j, 1\rangle, \langle k, 0\rangle, \langle \ell, 1\rangle\}\rangle\}$. Wir leiten nun als Beispiel einen Satz ab.

(362)
1 $\langle \texttt{jeder}, (C/(D\backslash C))/N, \forall \rangle$
2 $\langle \texttt{Igel}, N, \mathsf{igel}' \rangle$
3 $\langle \texttt{jeder Igel}, C/(D\backslash C), \forall(\mathsf{igel}') \rangle$
4 $\langle \texttt{singt}, D\backslash C, \mathsf{singt}' \rangle$
5 $\langle \texttt{jeder Igel singt}, C, 0 \rangle$

Die letzte Zeile bekommen wir wie folgt. Es ist k ein Element mit $\mathsf{igel}'(k) = 1$ (ist also ein Igel) und $\mathsf{singt}'(k) = 0$ (singt nicht). Also ist $\forall(\mathsf{igel}')(\mathsf{singt}') = 0$. Auf diese Weise lassen sich weitere Quantoren (/ein/, /mehr als zwei/) und so weiter analysieren. Unser definiter Artikel hat jetzt die folgende Semantik.

(363) $\langle \texttt{der}, (C/(D\backslash C))/N, \lambda P.\lambda Q.Q(\iota P) \rangle$

Zu guter Letzt sei noch bemerkt, dass die Bedeutung von Sätzen keine Wahrheitswerte sein können. Der Satz /Johann singt./ ist mal wahr mal falsch, je nach Zeitpunkt der Äußerung. Da wir den Zeitpunkt der Äußerung nicht in der Semantik festlegen können, setzen wir fest, dass die Bedeutung eines Satzes eine Funktion von Zeitpunkten in Wahrheitswerte ist. Um dies zu erreichen, führen wir einen neuen Basistyp, z, ein. Die neue Typzuweisung ist wie folgt.

(364)

α	$\tau(\alpha)$
D	e
N	$e \rightarrow (z \rightarrow t)$
C	$(z \rightarrow t)$

25 Kombinatorische Kategorialgrammatik

Kommen wir nun auf die Frage zu sprechen, wie wir mit der Flexibilität bei den Typen zurechtkommen können, wie wir sie in dem Kapitel 8 gesehen haben. Dazu ein Beispiel. Der Ausdruck /Johann/ bezeichnet ein Individuum, während der Ausdruck /jeder Hase/ eine Funktion von Eigenschaften von Individuen nach Wahrheitswerten bezeichnet. Dies war insofern unproblematisch, als der Name die Kategorie D, der quantifizierte Ausdruck aber die Kategorie C/(C\D) hat. Die Typzuweisung kann damit eindeutig bleiben, obwohl Subjektsausdrücke verschiedene Bedeutungen besitzen. Es gibt aber Gründe, warum man dennoch darauf bestehen will, dass Subjektsausdrücke eine einheitliche Kategorie besitzen. Einer davon ist die Koordination.

(365) Johann und der Hase rennen.

(366) Jeder Hase und jeder Igel rennt.

(367) Johann und jeder Hase rennt.

(Ich lasse die Kongruenz hier mal unberücksichtigt. Die abgebildeten Sätze entsprechen meiner eigenen Intuition. Es dürfte nicht einfach sein, eine Regelmäßigkeit dahinter zu entdecken.) Offenkundig kann man sowohl quantifizierte DPs, definite DPs wie auch Eigennamen miteinander koordinieren. Da Koordination stets Gleichheit der Kategorien voraussetzt, möchte man in diesem Fall wenigstens eine einheitliche Kategorie für die beiden haben. Es gibt dazu zwei Wege. Der erste, von Montague vorgeschlagene, ist, grundsätzlich nur eine Kategorie zu vergeben, und dies wäre hier der der quantifizierten Phrase. Damit wird auch einem Namen die Kategorie C/(D\C) zugeordnet. Die Semantik ist wie folgt. Wir ordnen /Johann/ eine Funktion zu, die jedem Prädikat P den Wert 1 zuordnet, wenn P auf i (die ursprüngliche Denotation von /Johann/) zutrifft, ansonsten sei der Wert für P 0. Mit anderen Worten, wir ordnen /Johanna/ die Funktion $\lambda P.P(i)$ zu.

(368) $\langle \text{Johann}, \text{C}/(\text{D}\backslash\text{C}), \lambda P.P(i)\rangle$

In der Tat ist der Typ der Funktion $(e \to t) \to t$. Denn die Variable P muss den Typ $e \to t$ haben, da sie auf i angewendet wird und einen Wahrheitswert (der Typ, der mit C assoziiert wird) ausgeben soll. Dieses Manöver lässt sich stark verallgemeinern. Es sei ein Objekt x vom Typ α gegeben. Dieses kann als Argument einer Funktion f vom Typ $\alpha \to \beta$ dienen und liefert den Wert $f(x)$ vom Typ β.

Wir können aber auch x zunächst einmal wie folgt "anheben":

(369) $x^* := \lambda g.g(x)$

Hierbei ist g eine Variable vom Typ $\alpha \to \beta$. x^* hat jetzt den Typ $(\alpha \to \beta) \to \beta$. Und nun ist

(370) $x^*(f) = (\lambda g.g(x))(f) = f(x)$

Wir replizieren diese Anhebung jetzt für die Kategorien. Wir gehen aus von einem Zeichen $\langle \vec{x}, \kappa, m\rangle$. Dieses kann jetzt links von einem Zeichen $\langle \vec{y}, \kappa\backslash\lambda, n\rangle$ stehen, und dann bekommen wir ein Zeichen $\langle \vec{x}\vec{y}, \lambda, n(m)\rangle$. Oder aber wir "heben" das ursprüngliche Zeichen an auf

(371) $\langle \vec{x}, \lambda/(\kappa\backslash\lambda), \lambda g.g(m)\rangle$

Wiederum können wir diesen Zeichen mit $\langle \vec{y}, \kappa\backslash\lambda, n\rangle$ kombinieren und erhalten $\langle \vec{x}\vec{y}, \lambda, n(m)\rangle$.

In der Nachfolge von Montague hat sich durchgesetzt, den Typ einer Konstituente nicht ein für allemal festzulegen, sondern die Anhebung als einen neuen Typ von Abschlussregel zu formulieren:

> [**Typanhebung I**]
> Ist $\langle \vec{x}, \kappa, m\rangle \in \Sigma(K)$, so auch $\langle \vec{x}, \lambda/(\kappa\backslash\lambda), \lambda g.g(m)\rangle \in \Sigma(K)$, wo g eine Variable vom Typ $\tau(\kappa\backslash\lambda)$ ist.

Zu dieser Regel gibt es noch eine duale Formulierung, die den Fall behandelt, wo das Zeichen als Argument rechts von dem Funktor steht.

> [**Typanhebung II**]
> Ist $\langle \vec{x}, \kappa, m\rangle \in \Sigma(K)$, so auch $\langle \vec{x}, (\lambda/\kappa)\backslash\lambda, \lambda g.g(m)\rangle \in \Sigma(K)$, wo g eine Variable vom Typ $\tau(\lambda/\kappa)$ ist.

Verwenden wir diese Regeln, so erweitern wir die von der Grammatik ableitbaren Zeichen. Die Grammatik (also im Wesentlichen das Lexikon) bleibt dann gleich, jedoch bekommen wir jetzt Zeichen, die wir in der AB-Kategorialgrammatik nicht bekommen haben.

Es geht jetzt also darum, sich auf einen Vorrat an Schlussregeln zu einigen, mit denen man aus den Zeichen im Lexikon komplexe Zeichen ableiten kann. Da diese Regeln völlig allgemein und unbeschränkt arbeiten sollen, ist ihre Art und Zusammensetzung Gegenstand energischer Debatten. Hier geht es nämlich um das, was in der Transformationsgrammatik als Universalgrammatik bezeichnet wird, und dies ist im Grunde das, womit sich die Syntaxtheorie auseinandersetzen muss. Der allgemeinste Kalkül ist der sogenannte Lambek-Kalkül, den ich allerdings hier nicht vorstellen will. Stattdessen werde ich den kombinatorischen Zugang beschreiben.

In der **kombinatorischen Kategorialgrammatik** wird das Potential des Lexikons, Zeichen zu erzeugen, mit Hilfe von Kombinatoren bestimmt. Und das geht im Wesentlichen wie in Kapitel 8 beschrieben. Eine kombinatorische Regel macht aus einem, höchstens zwei Zeichen ein neues Zeichen. Die Verbindung auf der Zeichenkettenebene ist dabei stets und immer Verkettung. Auf der kategorialen und der semantischen Ebene hingegen werden verschiedene Operationen angewendet; auf der semantischen Ebene sind es Kominatorien. Diese Kombinatoren müssen allerdings zu den syntaktischen Kategorien passen.

Der einfachste Fall ist die Applikation. Es gibt zwei Formen, die Vorwärtsapplikation und die Rückwärtsapplikation. Welche wir wählen müssen, wird von den Kategorien entschieden. Die Vorwärtsapplikation ist dann zu wählen, wenn die linke Konstituente die Kategorie κ/λ hat, die rechte hingegen die Kategorie λ:

(372) $$\frac{\langle \vec{x}, \kappa/\lambda, m\rangle \quad \langle \vec{y}, \lambda, n\rangle}{\langle \vec{x}\,\vec{y}, \kappa, \mathsf{A}mn\rangle}$$

Hier nun ist die Rückwärtsapplikation:

(373) $$\frac{\langle \vec{x}, \lambda, m\rangle \quad \langle \vec{y}, \lambda\backslash\kappa, n\rangle}{\langle \vec{x}\,\vec{y}, \kappa, \mathsf{R}mn\rangle}$$

Die Typanhebung funktioniert wie folgt.

(374) $$\frac{\langle \vec{x}, \kappa, m\rangle}{\langle \vec{x}, \lambda/(\kappa\backslash\lambda), \mathsf{T}^{\tau(\lambda)}m\rangle} \qquad \frac{\langle \vec{x}, \kappa, m\rangle}{\langle \vec{x}, (\lambda/\kappa)\backslash\lambda, \mathsf{T}^{\tau(\lambda)}m\rangle}$$

Hierbei ist $\mathsf{T}^{\beta}x := \lambda f.f(x)$, wobei der Typ von f gerade $\alpha \to \beta$ ist, α der Typ von x.

Eine weitere, sehr wichtige Regel ist schon von Peter Geach vorgeschlagen worden, siehe [4]. Geach schlägt als allgemeine Regel die folgende vor: falls $[\kappa\,\lambda]$

eine Konstituente der Kategorie μ ist, so ist $[\kappa\ \lambda/\xi]$ eine Konstituente der Kategorie μ/ξ. Da bei Geach keinerlei Direktionalität kodiert ist und auch keine Semantik, ist seine Regel etwas schwer wiederzugeben. Ich betrachte daher einen Spezialfall, nämlich $\kappa = \zeta/\lambda$. Da ζ/λ und λ eine Konstituente der Kategorie ζ bilden, folgt jetzt, dass ζ/λ und λ/ξ eine Konstituente der Kategorie ζ/ξ bilden können.

Dies ist genau der Fall, in dem wir eine Konstituente $[\vec{x}\,[\vec{y}\,\vec{z}]]$ haben, die wir besser als $[[\vec{x}\,\vec{y}]\,\vec{z}]$ analysieren wollen. Wiederum ist Koordination ein wichtiges Argument.

(375) `das rote und das blaue Auto`

Wollen wir die Ketten /`das rote`/ sowie /`das blaue`/ koordinieren, so müssen sie Konstituenten sein. Das sind sie aber nicht, weil nur /`blaue Auto`/ eine Konstituente ist. Es ist aber so ziemlich unmöglich, eine saubere Analyse vorzulegen, wenn man nicht /`das rote`/ und /`das blaue`/ zu Konstituenten macht. Wie schon in Kapitel 8 erläutert, bedienen wir uns einer Paraphrase. Die Bedeutung von (375) soll identisch sein mit

(376) `das rote Auto und das blaue Auto`

Desweiteren soll jeweils die Bedeutung von /`das rote`/ so gestaltet sein, dass die Kombination mit /`Auto`/ dieselbe Bedeutung liefert wie als hätten wir die originale Konstituentenstruktur verwendet. Das Rezept dazu ist die Funktionskomposition. Betrachten wir nämlich die Kategorien: ist die Konstituentenstruktur $[\vec{x}\,[\vec{y}\,\vec{z}]]$, und hat $\vec{z}$ die Kategorie κ, so hat $\vec{y}$ die Kategorie λ/κ, und $\vec{x}$ die Kategorie μ/λ. (Es gibt noch andere Möglichkeiten!) In diesem Fall muss die Kategorie von $[\vec{x}\,\vec{y}]$ genau μ/κ sein. Der dazugehörende Kombinator ist

(377) $\mathsf{C}xy := \lambda z.x(yz)$

In der Tat ist

(378) $(\mathsf{C}xy)z = (\lambda z.x(yz))z = x(yz)$

Auch hier sehen wir, wie der Einsatz des Kombinators die Struktur der Applikationen ändert. Hier ist also die neue Kombinationsregel, genannt **harmonische Komposition**.

(379) $$\frac{\langle \vec{x}, \mu/\lambda, m\rangle \quad \langle \vec{y}, \lambda/\kappa, n\rangle}{\langle \vec{x}\,\vec{y}, \mu/\kappa, \mathsf{C}mn\rangle} \qquad \frac{\langle \vec{x}, \kappa\backslash\lambda, m\rangle \quad \langle \vec{y}, \lambda\backslash\mu, n\rangle}{\langle \vec{x}\,\vec{y}, \kappa\backslash\mu, \mathsf{O}mn\rangle}$$

Hierbei ist

(380) $\mathsf{O}xy := \lambda z.y(xz)$

Ich formuliere diese zum besseren Verständnis vollständig aus.

> [**Harmonische Komposition I**]
> Ist $\langle \vec{x}, \mu/\lambda, m\rangle \in \Sigma(K)$ und $\langle \vec{y}, \lambda/\kappa, n\rangle \in \Sigma(K)$, so ist auch $\langle \vec{x}\vec{y}, \mu/\kappa, \mathsf{C}mn\rangle \in \Sigma(K)$.

> [**Harmonische Komposition II**]
> Ist $\langle \vec{x}, \kappa\backslash\lambda, m\rangle \in \Sigma(K)$ und $\langle \vec{y}, \lambda\backslash\mu, n\rangle \in \Sigma(K)$, so ist auch $\langle \vec{x}\vec{y}, \kappa\backslash\mu, \mathsf{O}mn\rangle \in \Sigma(K)$.

Bei der harmonischen Komposition erwarten beide Konstituenten ihre Argumente in derselben Richtung. Das hat den Effekt, dass die Umklammerung keinerlei Wortstellungsvarianten erzeugt.

Es gibt aber auch die sogenannte disharmonische Komposition; diese ist in Sprachen nicht sonderlich beliebt, weil sie nämlich die Wortstellung verändert.

(381) $$\frac{\langle \vec{x}, \mu/\lambda, m\rangle \quad \langle \vec{y}, \kappa\backslash\lambda, n\rangle}{\langle \vec{x}\vec{y}, \kappa\backslash\mu, \mathsf{C}mn\rangle} \qquad \frac{\langle \vec{x}, \lambda/\kappa, m\rangle \quad \langle \vec{y}, \lambda\backslash\mu, n\rangle}{\langle \vec{x}\vec{y}, \mu/\kappa, \mathsf{O}mn\rangle}$$

Um zu sehen, warum diese Variante die Wortstellung verändert, nehmen wir für die erste Regel noch ein Element $\langle \vec{z}, \kappa, p\rangle$ hinzu. Ohne disharmonische Komposition bekommen wir eine Konstituentenstruktur $[\vec{x}[\vec{z}\vec{y}]]$, wie man leicht nachrechnet. Wenden wir die Regel an, so ist ferner auch noch diese Struktur möglich: $[\vec{z}[\vec{x}\vec{y}]]$. Wie man sieht, ist die Wortstellung jetzt auch noch verändert worden.

Zuletzt aber wollen wir uns der Koordination zuwenden. Wie wir gesehen hatten, muss man dem Wort /und/ sehr viele verschiedene Bedeutungen zuweisen, die aber alle aus einer einzigen entstehen. Um diese zu erhalten, schlagen wir nun eine allgemeine Regel vor.

(382) $$\frac{\langle /\mathsf{und}/, (\kappa\backslash\kappa)/\kappa, m\rangle}{\langle /\mathsf{und}/, ((\kappa/\lambda)\backslash(\kappa/\lambda))/(\kappa/\lambda), \mathsf{S}^{\tau(\lambda)}m\rangle}$$

Wobei

(383) $\mathsf{S}^{\gamma}xyzu = x(yu)(zu)$

und γ ist der Typ von u. Auch diese allgemeine Anhebungsregel geht übrigens auf Peter Geach zurück. Die Formulierung der Regel ist nicht ganz einfach. Zunächst einmal will man sie nur für logische Operationen anwenden. Dies kann man auf zwei Weisen implementieren. Entweder man benennt explizit die lexikalischen Elemente, auf die sie angewendet werden kann.

> [**Koordination**]
> Ist $\langle /\mathsf{und}/, (\kappa\backslash\kappa)/\kappa, m\rangle \in \Sigma(K)$, so ist auch $\langle /\mathsf{und}/, ((\kappa/\lambda)\backslash(\kappa/\lambda))/(\kappa/\lambda), \mathsf{S}^{\tau(\lambda)}m\rangle \in \Sigma(K)$.

Oder aber man fixiert den Typ bzw. die Kategorie des Ausdrucks. Wir sagen, ein Typ α sei **wahrheitsfunktional**, wenn entweder $\alpha = t$ (der Typ der Wahrheitswerte) oder $\alpha = \gamma \to \beta$, wo β wahrheitsfunktional ist. Mit dieser Definition können wir die Regel verallgemeinern.

> [**Koordination**]
> Es sei $\tau(\kappa)$ wahrheitsfunktional. Ist $\langle \vec{x}, (\kappa\backslash\kappa)/\kappa, m\rangle \in \Sigma(K)$, so ist auch $\langle \vec{x}, ((\kappa/\lambda)\backslash(\kappa/\lambda))/(\kappa/\lambda), \mathsf{S}^{\tau(\lambda)}m\rangle \in \Sigma(K)$.

Schließlich sei noch erwähnt, dass die Negation nicht unter diese Regel fällt, weil sie nur einstellig ist. Um dies zu ändern, müssten wir entweder noch eine weitere Koordinationsregel hinzufügen, oder aber ganz allgemein Kategorien zulassen, die nur aus einer einzigen Kategorie κ aufgebaut sind (wie etwa κ/κ oder $(\kappa/\kappa)\backslash\kappa$), wobei $\tau(\kappa)$ wahrheitsfunktional ist. Die Kombinatoren müssen dann entsprechend angepasst werden.

Literatur

[1] Carstensen, K.-U., Chr. Ebert, C. Endriss, S. Jekat, R. Klabunde und H. Langer: *Computerlinguistik und Sprachtechnologie. Eine Einführung*. Spektrum. Akademischer Verlag, Berlin, 2 Auflage, 2004.

[2] Dowty, David R., Robert E. Wall und Stanley Peters: *Introduction to Montague Semantics*. Nummer 11 in *Synthese Library*. Reidel, Dordrecht, 1981.

[3] Friedl, Jeffrey E. F.: *Reguläre Ausdrücke*. O'Reilly, 2003.

[4] Geach, Peter: *A Program for Syntax*. In: Davidson, Donald und Gilbert Harman (Herausgeber): *Semantics for Natural Language*, Nummer 40 in *Synthese Library*. Reidel, Dordrecht, 1972.

[5] Harrison, Michael A.: *Introduction to Formal Language Theory*. Addison Wesley, Reading (Mass.), 1978.

[6] Korpela, Jukka: *Unicode Explained*. O'Reilly, Sebastopol, CA, 2006.

[7] Kracht, Marcus: *Mathematics of Language*. Mouton de Gruyter, Berlin, 2003.

[8] Steedman, Mark: *The Syntactic Process*. MIT Press, Cambridge (Mass.), 2000.