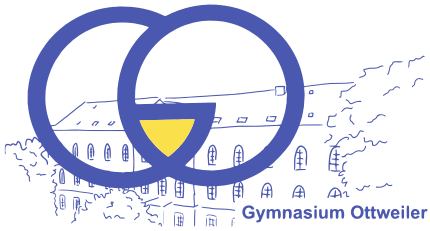
Kapitel 5

Automaten und formale Sprachen



Skript Informatik · Grundkurs

Sie geben im Onlineshop Ihre E-Mail-Adresse ein und vergessen das @. Noch bevor Sie auf „Bestellen“ klicken, färbt sich das Feld rot. Ob es die Adresse gibt, weiß die Seite nicht — aber dass `max.mustermann.de` keine sein *kann*, sieht sie sofort. So etwas passiert ständig um Sie herum: Das Drehkreuz im Freibad lässt Sie erst nach dem Chip durch, die Ampel weiß, dass nach Rot-Gelb Grün kommt, und Ihr Handy entsperrt sich erst nach der richtigen Ziffernfolge. Hinter all dem steckt dieselbe einfache Maschine. Um sie geht es in diesem Kapitel.

Auch ein Compiler liest Ihr Programm und weiß nach dem Bruchteil einer Sekunde, ob es regelgerecht aufgebaut ist. Er versteht dabei nicht, *was* Sie vorhaben — er prüft nur, ob die Zeichenfolge den Regeln der Sprache gehorcht. Genau diese Trennung macht ihn möglich: Form lässt sich mechanisch prüfen, Bedeutung nicht.

Dieses Kapitel baut das Werkzeug dafür auf. Erst die Frage, was eine Sprache formal überhaupt ist, dann Grammatiken als Regelwerk zum Erzeugen von Wörtern, dann endliche Automaten als Maschinen zum Erkennen von Wörtern. Am Ende steht ein Programm, das eine solche Maschine ausführt — und die Einsicht, wo diese einfachen Maschinen an ihre Grenze stoßen.

Lernziele

Am Ende dieses Kapitels können Sie

- natürliche und künstliche Sprachen unterscheiden und Syntax von Semantik trennen,
- die Arbeitsweise von Interpreter und Compiler beschreiben,
- die Begriffe Alphabet, Wort, A^* und formale Sprache sicher verwenden,
- eine Grammatik aus Terminalen, Nichtterminalen, Produktionen und Startsymbol angeben und Ableitungen samt Ableitungsbaum aufschreiben,
- reguläre Grammatiken erkennen und zu einer Sprache einen endlichen Automaten konstruieren — als Zustandsgraph und als Übergangstabelle,
- deterministische von nichtdeterministischen Automaten unterscheiden und einen NEA mit der Teilmengenkonstruktion in einen DEA umwandeln,
- eine reguläre Grammatik in einen endlichen Automaten und einen Automaten zurück in eine Grammatik übersetzen,
- einen endlichen Automaten programmieren,
- reguläre Ausdrücke lesen, schreiben und ihren Zusammenhang zu Automaten und Grammatiken erklären,
- Mealy- und Moore-Automaten unterscheiden und Übergangs- samt Ausgabetabelle angeben,
- begründen, warum endliche Automaten Klammerstrukturen nicht prüfen können,
- die vier Stufen der Chomsky-Hierarchie verbal charakterisieren und je ein Beispiel für eine nicht-reguläre und eine nicht-kontextfreie Sprache nennen.

5.1 Sprachen und ihre Regeln

Tippen Sie in Ihren Taschenrechner `3 + × 4`. Viele Modelle antworten mit *Syntax ERROR*: Die Zeichenfolge ist gar keine Rechnung, der Rechner kann sie nicht einmal lesen. Tippen Sie dagegen `5 ÷ 0`, kommt *Math ERROR*: Die Eingabe ist tadellos gebaut, nur ergibt sie keinen Sinn. Ihr Taschenrechner unterscheidet damit genau die zwei Ebenen, um die es in diesem Abschnitt geht.

Sprachen dienen der Verständigung. Man unterscheidet **natürliche Sprachen**, die zwischen Menschen gewachsen sind (Deutsch, Französisch), und **künstliche Sprachen**, die jemand bewusst festgelegt hat. Programmiersprachen gehören dazu, aber auch die Formelsprache der Mathematik.

Jede Sprache gehorcht Regeln auf zwei Ebenen:

- Die **Syntax** legt die erlaubte Form von Wörtern und Sätzen fest.
- Die **Semantik** bestimmt deren Bedeutung.

Der Unterschied lässt sich am Beispiel zeigen. Der Satz „Der Hund ist ein Insekt" ist syntaktisch einwandfrei und semantisch falsch. Die Zeile `x := "drei" + 5;` ist umgekehrt semantisch klar gemeint und syntaktisch bzw. typmäßig unzulässig. Ein Compiler prüft die Syntax vollständig, die Semantik nur zu einem kleinen Teil — deshalb erzeugt ein fehlerfrei übersetztes Programm noch lange nicht das gewünschte Ergebnis.

Dass natürliche Sprachen erheblich komplizierter sind als Programmiersprachen, zeigt schon ein einziger Satz:

> Die deutsche Umgangssprache ist, wie jeder, der sie — und das wird an diesem Satz hier, den ich der Erläuterung wegen einschiebe, deutlich — lernen muss, weiß, eine sehr komplizierte Sprache.

Ein Mensch versteht ihn nach zweimaligem Lesen. Eine Maschine bräuchte dafür eine Grammatik, die beliebig tief verschachtelte Einschübe erlaubt — genau die Art von Struktur, an der die Automaten dieses Kapitels scheitern werden (Abschnitt 5.10).

Arbeitsauftrag 1

1. Nennen Sie je zwei natürliche und zwei künstliche Sprachen und begründen Sie die Zuordnung.
2. Geben Sie ein Beispiel für einen deutschen Satz, der syntaktisch korrekt, aber semantisch sinnlos ist — und eines für den umgekehrten Fall.
3. Ihr Programm übersetzt fehlerfrei, liefert aber falsche Ergebnisse. Welche Ebene ist betroffen? Begründen Sie.

5.2 Vom Quelltext zur Maschine

Bei einer Pressekonferenz überträgt eine Dolmetscherin jeden Satz, während der Gast noch spricht — wiederholt er sich, übersetzt sie eben noch einmal. Ein Roman dagegen wird einmal vollständig übersetzt, gedruckt und dann tausendfach gelesen, ohne dass der Übersetzer noch dabei ist. Genau diese beiden Wege gibt es auch, um ein Programm auf den Prozessor zu bringen.

Ein Prozessor versteht nur seine eigene Maschinensprache. Der Weg dorthin führt über eines von zwei Verfahren.

Ein **Interpreter** bekommt Programm und Eingabedaten gleichzeitig und führt jede Anweisung sofort aus. Jede Zeile ist ihm auch beim hundertsten Schleifendurchlauf wieder neu — das kostet Zeit, erlaubt aber sofortiges Ausprobieren. Python arbeitet im Kern so.

Ein **Compiler** (von *to compile*, zusammenstellen) trennt beides zeitlich: Zuerst wird das Programm unabhängig von allen Eingabedaten analysiert und in die Zielsprache übersetzt, danach läuft das Ergebnis beliebig oft. Delphi und C++ arbeiten so.

```
Quellsprache          →   Compiler   →   Zielsprache
(Pascal, Delphi, C)                     (Maschinen-, Assemblersprache)
```

Die Übersetzung zerfällt in zwei Phasen:

- **Analyse.** Die **lexikalische Analyse** zerlegt den Zeichenstrom in *Token* — Schlüsselwörter, Bezeichner, Zahlen, Operatoren. Die **syntaktische Analyse** prüft, ob diese Token nach den Regeln der Grammatik zusammenpassen, und baut daraus einen Ableitungsbaum. Fehlermeldungen entstehen hier.
- **Synthese.** Aus dem Baum wird der Code der Zielsprache erzeugt.

Merke

Der Lexer benutzt endliche Automaten (Abschnitt 5.6), der Parser eine Grammatik (Abschnitt 5.4). Dieses Kapitel baut also genau die beiden Werkzeuge, mit denen jeder Compiler beginnt.

Arbeitsauftrag 2

4. Nennen Sie je einen Vorteil und einen Nachteil von Interpreter und Compiler.
5. Zerlegen Sie die Zeile `zaehler := zaehler + 1;` von Hand in Token und benennen Sie deren Art.
6. Warum meldet der Compiler einen Tippfehler in einem Schlüsselwort, nicht aber eine verwechselte Variable?

5.3 Alphabet, Wort und Sprache

Ihre Handy-PIN besteht aus vier Ziffern, und 0000 zählt genauso wie 4711. Erlaubt sind die zehn Zeichen 0 bis 9, jede PIN ist eine Folge aus genau vier davon. In der Sprache dieses Abschnitts: Das Ziffernfeld ist das Alphabet, jede PIN ein Wort der Länge 4, und die PINs, die Ihr Handy entsperren, bilden eine sehr kleine Sprache. Wie viele PINs es gibt, rechnen Sie mit dem Satz weiter unten in einer Zeile aus.

Definition (Alphabet). Ein **Alphabet** A ist eine endliche, geordnete Menge von Zeichen.

Beispiele: $A = \{0, 1\}$ für Dualzahlen, $A = \{a, \ldots, z, 0, \ldots, 9\}$ für Bezeichner, für Pascal die Menge aller erlaubten Zeichen samt Schlüsselwörtern.

Definition (Wort). Ein **Wort** über A ist eine endliche Folge von Zeichen aus A. Das **leere Wort** wird mit ε bezeichnet und hat die Länge 0.

Für $A = \{a, b, c\}$ sind a, ab, cba, $aaaa$ und ε Wörter.

Bezeichnung. Die Menge aller endlich langen Wörter über A heißt A^*, die Menge aller Wörter der Länge n heißt A^n.

$$A^0 = \{\varepsilon\} \qquad A^1 = \{a, b, c\} \qquad A^2 = \{aa, ab, ac, ba, bb, bc, ca, cb, cc\}$$

Satz. Ist A ein Alphabet mit k Zeichen, so hat A^n genau k^n Elemente. A^* hat unendlich viele Elemente — aber abzählbar unendlich viele, denn man kann die Wörter der Länge nach und innerhalb gleicher Länge alphabetisch aufzählen.

Definition (formale Sprache). Jede Teilmenge $L \subseteq A^*$ heißt **formale Sprache** über A.

Diese Definition ist großzügig: Jede beliebige Auswahl von Wörtern ist eine Sprache. Interessant werden nur die Sprachen, die sich *endlich beschreiben* lassen — durch eine Regel, eine Grammatik oder einen Automaten.

Arbeitsauftrag 3

7. $A = \{0, 1\}$. Geben Sie A^2 und A^3 vollständig an. Wie viele Elemente hat A^8?
8. $A = \{x, y, z\}$. Notieren Sie A^1 und die ersten zehn Wörter von A^3.
9. $A = \{a, b, c\}$. Welche der folgenden Mengen sind formale Sprachen über A? $L_4 = \{\}$, $L_5 = \{\varepsilon\}$, $L_6 = \{x\}$, $L_7 = \{a\}$. Begründen Sie.
10. Beschreiben Sie in Worten: $L_{10} = \{b, ab, aab, aaab, \ldots\}$ und $L_{11} = \{b, ba, baa, baaa, \ldots\}$.
11. $A = \{a, b, c\}$. Geben Sie die Sprache aller Wörter der Länge 3 an, deren mittleres Zeichen ein c ist. Wie viele Wörter sind das?

5.4 Grammatiken

Wählen Sie aus jeder Spalte ein Teil: *Mein Hund* | *Mein kleiner Bruder* | *Der Drucker* — *hat* — *meine Hausaufgaben* | *mein Referat* — *gefressen* | *gelöscht* | *versteckt*. Das ergibt $3 \cdot 2 \cdot 3 = 18$ Ausreden, ohne dass Sie eine einzige auswendig lernen. So arbeitet eine Grammatik: wenige Regeln, viele Sätze. Ob die Ausrede glaubwürdig ist, entscheidet sie nicht — das ist Semantik.

Nicht jede Folge von Wörtern ergibt einen Satz:

- Hund alle sind.
- Der Hund ist ein Insekt.
- `while do end x < 3 begin`

Die erste und die dritte Zeile verstoßen gegen die Form, die zweite nur gegen den Inhalt. Für die Form zuständig ist die **Grammatik**. Sie hat zwei Aufgaben: die zulässigen Zeichenfolgen zu beschreiben und die Regeln dafür darzustellen. Die Semantik spielt dabei keine Rolle.

Definition: Grammatik

Eine **Grammatik** ist ein **4-Tupel** $G = (V_N, V_T, P, S)$ mit

- der Menge V_N der **Nichtterminalzeichen** (syntaktische Variablen),
- der Menge V_T der **Terminalzeichen** (auch Token — die Zeichen, die im fertigen Wort stehen dürfen), wobei $V_N \cap V_T = \emptyset$,
- der Menge P der **Produktionen** (Regeln der Form „links $\rightarrow$ rechts") und
- dem **Startsymbol** $S \in V_N$.

Die Angabe als Tupel ist die übliche Schreibweise und wird im Abitur erwartet. Sie zwingt dazu, nichts zu vergessen: vier Klammern, vier Bestandteile.

Eine **Ableitung** ist eine Folge von Ersetzungen, beginnend beim Startsymbol. Die Menge aller ableitbaren Wörter über V_T heißt die von G **erzeugte Sprache** $L(G)$.

5.4.1 Ein erstes Beispiel

$$V_T = \{\text{Katzen}, \text{Hunde}, \text{schnurren}, \text{fressen}\}, \quad V_N = \{S, \text{Subjekt}, \text{Prädikat}\}$$

$$\begin{aligned}
(P_1) &\quad S \rightarrow \text{Subjekt Prädikat} \\
(P_2) &\quad \text{Subjekt} \rightarrow \text{Katzen} \mid \text{Hunde} \\
(P_3) &\quad \text{Prädikat} \rightarrow \text{schnurren} \mid \text{fressen}
\end{aligned}$$

Der Pfeil bedeutet: Das Nichtterminal links darf durch die rechte Seite ersetzt werden, der Strich | trennt Alternativen. Ableitung von „Hunde schnurren":

$$S \Rightarrow \text{Subjekt Prädikat} \Rightarrow \text{Hunde Prädikat} \Rightarrow \text{Hunde schnurren}$$

Der Satz gehört also zu $L(G)$ — dass Hunde nicht schnurren, ist eine Frage der Semantik und geht die Grammatik nichts an. Grafisch notiert man die Ableitung als **Ableitungsbaum**:

```
          S
         / \
   Subjekt  Prädikat
      |        |
    Hunde   schnurren
```

5.4.2 Zweites Beispiel: Klammerterme

Gesucht ist eine Grammatik für korrekt geschachtelte Klammern wie `(())` oder `((()())())`.

$$V_T = \{(,\ )\}, \qquad V_N = \{S\}, \qquad P = \{\, S \to (S) \mid SS \mid \varepsilon \,\}$$

Ableitung von `(())()`:

$$S \Rightarrow SS \Rightarrow (S)S \Rightarrow ((S))S \Rightarrow (())S \Rightarrow (())(S) \Rightarrow (())()$$

Beachten Sie die Regel $S \to (S)$: Das Nichtterminal steht *zwischen* Terminalzeichen. Diese Bauart wird in Abschnitt 5.10 entscheidend.

5.4.3 Drittes Beispiel: ein Satzmuster

$$V_T = \{\text{der, den, Hund, Briefträger, beißt, schlägt}\}$$
$$V_N = \{S, \text{Subjekt, Prädikat, Objekt, Nomen}\}$$
$$P = \{\, S \to \text{Subjekt Prädikat Objekt},\ \text{Subjekt} \to \text{der Nomen},\ \text{Objekt} \to \text{den Nomen},$$
$$\text{Prädikat} \to \text{beißt} \mid \text{schlägt},\ \text{Nomen} \to \text{Hund} \mid \text{Briefträger} \,\}$$

Arbeitsauftrag 4

12. Leiten Sie mit der letzten Grammatik die Sätze „der Briefträger beißt den Hund" und „der Briefträger schlägt den Briefträger" ab und zeichnen Sie je einen Ableitungsbaum.
13. Wie viele verschiedene Sätze erzeugt diese Grammatik insgesamt? Begründen Sie mit einer Rechnung.
14. Leiten Sie `((()))` und `()()()` mit der Klammergrammatik ab.
15. Geben Sie eine Grammatik an, die alle Wörter aus a und b erzeugt, die mit a beginnen.
16. Erweitern Sie die Satzgrammatik um ein Adjektiv vor dem Nomen.

5.5 Reguläre Grammatiken

Wer ein Kennzeichen wie WND-AB 123 vorliest, geht Zeichen für Zeichen von links nach rechts. Merken muss er sich dabei nur eines: ob er gerade im Kreiskürzel, bei den Buchstaben oder bei den Ziffern ist. Er blättert nie zurück und muss nichts Späteres vorwegnehmen. Grammatiken, die so gebaut sind — ein Zeichen anhängen, dann weiter —, heißen regulär.

Definition. Eine Grammatik heißt **regulär**, wenn jede Produktion die Form

$$A \to xB \qquad \text{oder} \qquad A \to x$$

hat, mit $A, B \in V_N$ und $x \in V_T$. Rechts steht also **genau ein Terminalzeichen** und dahinter höchstens *ein* Nichtterminal — und wenn, dann *ganz rechts*. Die erzeugte Sprache heißt **reguläre Sprache**.

Stolperstein

Die Definition ist enger, als sie beim Querlesen wirkt. Nicht erlaubt sind $A \to xyB$ (zwei Terminalzeichen), $A \to Bx$ (Nichtterminal links) und $A \to \varepsilon$ (leeres Wort). Umgekehrt gilt: Eine Grammatik, die diese Form *nicht* hat, kann durchaus eine reguläre Sprache erzeugen — man kann sie dann in die reguläre Form umschreiben. „Regulär" ist also eine Eigenschaft der *Grammatik*; „reguläre Sprache" eine Eigenschaft der *Sprache*. Beides auseinanderzuhalten lohnt sich.

Die Klammergrammatik aus 5.4 ist nach dieser Definition nicht regulär — bei $S \to (S)$ steht das Nichtterminal in der Mitte. Das ist kein Zufall, wie sich zeigen wird.

Beispiel: gerade Dualzahlen. Gesucht ist eine Grammatik für alle Dualzahlen ohne führende Null, die gerade sind — die also mit 1 beginnen und auf 0 enden.

$$V_T = \{0, 1\}, \quad V_N = \{S, A\}, \quad P = \{\, S \to 1A, \quad A \to 0A \mid 1A \mid 0 \,\}$$

Ableitung von 1010: $S \Rightarrow 1A \Rightarrow 10A \Rightarrow 101A \Rightarrow 1010$

> In der Webfassung steht hier ein Bedienfeld zum Ausprobieren: **Ableitungswerkzeug**.

Geben Sie im Bedienfeld auch 101 ein — eine ungerade Zahl. Es gibt keine Ableitung, und das Werkzeug sagt es.

Beispiel: Bezeichner. In vielen Sprachen gilt: Ein Bezeichner beginnt mit einem Buchstaben, danach folgen beliebig viele Buchstaben oder Ziffern. Mit $V_T = \{bu, zi\}$ (Buchstabe, Ziffer):

$$V_N = \{S, R\}, \qquad P = \{\, S \to bu \mid bu\,R, \quad R \to bu\,R \mid zi\,R \mid bu \mid zi \,\}$$

Arbeitsauftrag 5

17. Prüfen Sie, ob die Grammatik $P = \{S \to aS \mid bS \mid aaA,\ A \to aA \mid bA \mid \varepsilon\}$ die obige Definition erfüllt — prüfen Sie jede Produktion einzeln. Beschreiben Sie anschließend $L(G)$ in Worten und geben Sie eine Grammatik an, die dieselbe Sprache erzeugt und die Definition erfüllt.
18. Geben Sie eine reguläre Grammatik für alle ungeraden Dualzahlen an und leiten Sie 1011 und 1001 ab.
19. Gibt es eine reguläre Grammatik für $L = \{10, 100, 110, 1000, 1010, 1100, 1110, \ldots\}$? Beschreiben Sie die Sprache zuerst in Worten.
20. Begründen Sie, warum $S \to aSb$ nicht regulär ist.

5.6 Endliche Automaten

Das Drehkreuz kennt genau zwei Zustände: *gesperrt* und *frei*. Halten Sie im Zustand gesperrt Ihre Karte an den Leser, wechselt es nach frei; gehen Sie hindurch, ist es wieder gesperrt. Drücken Sie gegen ein gesperrtes Drehkreuz, bleibt es gesperrt. Mehr weiß es nicht — weder, wie viele Leute heute schon durch sind, noch, wer Sie sind. Ein Türschloss mit Tastenfeld, das aufgeht, sobald die letzten drei Tasten 7, 3, 3 waren, ist übrigens genauso gebaut wie der Automat für „endet auf `abb`“ weiter unten.

> In der Webfassung steht hier ein Bedienfeld zum Ausprobieren: **Zustandsmaschine**.

Eine Grammatik *erzeugt* Wörter. Für den Compiler brauchen wir das Gegenstück: eine Maschine, die ein vorgelegtes Wort *prüft*. Das leistet der endliche Automat.

Definition: Endlicher Automat

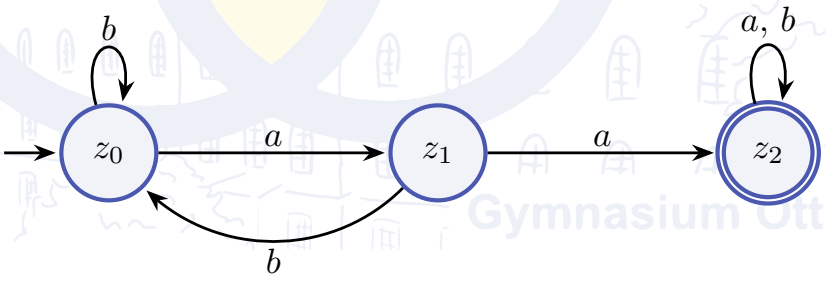
Ein **endlicher akzeptierender Automat** (Akzeptor) ist ein **5-Tupel** $M = (Z, A, f, z_0, F)$ mit

- der endlichen **Zustandsmenge** Z,
- dem **Eingabealphabet** A,
- der **Übergangsfunktion** $f : Z \times A \to Z$,
- dem **Anfangszustand** $z_0 \in Z$,
- der **Finalmenge** $F \subseteq Z$ der **Endzustände**.

Der Automat akzeptiert ein Wort, wenn er nach dem Lesen des letzten Zeichens in einem Endzustand steht.

Gezeichnet wird er als **Zustandsgraph**: Zustände sind Kreise, Übergänge beschriftete Pfeile, der Startzustand hat einen Pfeil von außen, Endzustände einen doppelten Rand.

Beispiel: Wörter über $\{a, b\}$, die `aa` enthalten.



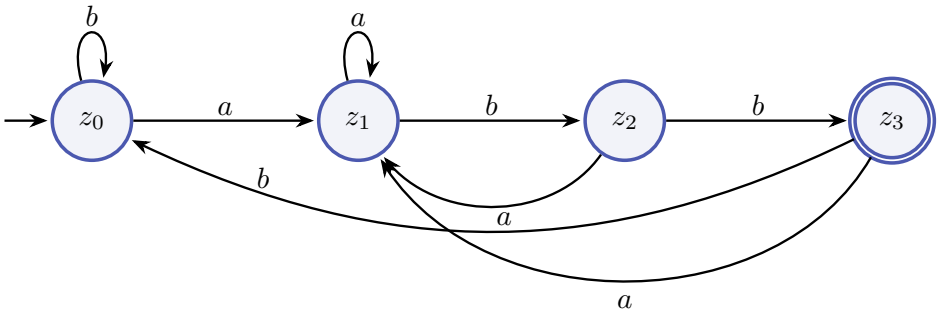
Zustandsgraph: akzeptiert alle Wörter über $\{a, b\}$, die `aa` enthalten.

In Worten: z_0 heißt „das zuletzt gelesene Zeichen war kein a“, z_1 heißt „das zuletzt gelesene Zeichen war ein a, aber noch kein aa”, und z_2 heißt „aa ist aufgetreten” — dieser Zustand wird nie wieder verlassen. Ein b in z_1 wirft zurück nach z_0. Beachten Sie: In z_0 kann der Automat schon beliebig viele a gelesen haben, nur eben nicht als letztes Zeichen. Der Zustand merkt sich nicht die Anzahl, sondern allein, wie weit das gesuchte aa gerade gediehen ist.

Dieselbe Information als **Übergangstabelle** — die Form, die man zum Programmieren braucht:

Zustand	bei a	bei b	Endzustand
z_0	z_1	z_0	nein
z_1	z_2	z_0	nein
z_2	z_2	z_2	**ja**

Beispiel: Wörter über $\{a, b\}$, die auf abb enden.



Jeder Zustand merkt sich, wie viel vom gesuchten abb schon gelesen ist.

Zustand	bei a	bei b	Bedeutung	Endzustand
z_0	z_1	z_0	nichts Passendes gesehen	nein
z_1	z_1	z_2	zuletzt a	nein
z_2	z_1	z_3	zuletzt ab	nein
z_3	z_1	z_0	zuletzt abb	**ja**

Der Zustand merkt sich also gerade so viel von der Vergangenheit, wie für die Entscheidung nötig ist — und keinen Buchstaben mehr. Das ist die zentrale Idee: **Der Zustand ist das Gedächtnis des Automaten, und dieses Gedächtnis ist endlich.**

> In der Webfassung steht hier ein Bedienfeld zum Ausprobieren: **Automatensimulator**.

Dasselbe Muster an einer Tür: Das Tastenschloss merkt sich nur, wie viel vom Code zuletzt getippt wurde. Alle Ziffern außer 7 und 3 stehen am Pfeil als „sonst“.

> In der Webfassung steht hier ein Bedienfeld zum Ausprobieren: **Zustandsmaschine**.

Arbeitsauftrag 6

21. Zeichnen Sie den Zustandsgraphen zur zweiten Tabelle und prüfen Sie die Wörter abb, aabb, abba, babb.
22. Konstruieren Sie einen DEA für alle geraden Dualzahlen und einen für alle ungeraden.
23. Konstruieren Sie einen DEA für Bezeichner nach der Regel aus 5.5.
24. Konstruieren Sie einen DEA über $\{a, b\}$, der genau die Wörter mit einer geraden Anzahl von a akzeptiert.
25. Geben Sie zu Ihrem Automaten aus Aufgabe 24 die Übergangstabelle an.
26. Welche Sprache akzeptiert der Automat mit $Z = \{z_0, z_1\}$, $F = \{z_1\}$ und den Übergängen $f(z_0, 0) = z_0$, $f(z_0, 1) = z_1$, $f(z_1, 0) = z_1$, $f(z_1, 1) = z_0$? (Tipp: Zählen Sie.)

5.6.1 Deterministisch oder nichtdeterministisch

Sie suchen in einem fremden Einkaufszentrum einen bestimmten Laden. Allein müssten Sie an jeder Gabelung raten. Mit genug Freunden geht es ohne Raten: An jeder Gabelung geht jemand in jede Richtung, und wer ankommt, ruft die anderen. Ein nichtdeterministischer Automat rechnet genau so — er verfolgt alle Wege gleichzeitig und ist zufrieden, wenn irgendeiner ankommt.

- Bei einem **deterministischen** Automaten (DEA) ist f eindeutig: Zu jedem Paar aus Zustand und Zeichen gibt es genau einen Folgezustand.
- Bei einem **nichtdeterministischen** Automaten (NEA) dürfen es mehrere sein. Der Automat müsste gewissermaßen raten, welchen Weg er nimmt, und akzeptiert ein Wort, wenn *irgendein* Weg in einen Endzustand führt.

Beim NEA ist f also keine Funktion in den Zuständen, sondern eine in die **Teilmengen** von Z: $f : Z \times A \to \mathcal{P}(Z)$. Der Automat ist gewissermaßen in mehreren Zuständen gleichzeitig.

NEAs sind oft schneller hingeschrieben — bauen lässt sich aber nur ein DEA, denn eine Maschine kann nicht raten. Der Trost ist ein Satz:

Merke

Zu **jedem** NEA gibt es einen DEA, der dieselbe Sprache akzeptiert. Nichtdeterminismus bringt keine zusätzliche Mächtigkeit, nur kürzere Diagramme.

5.6.2 Die Teilmengenkonstruktion

Der Beweis des Satzes ist zugleich ein Bauplan. Die Idee: Wenn der NEA in mehreren Zuständen gleichzeitig sein kann, macht man **jede mögliche Zustandsmenge zu einem einzigen DEA-Zustand**.

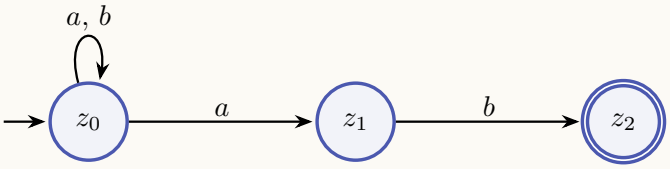
Definition: Teilmengenkonstruktion

Gegeben ein NEA mit Zustandsmenge Z. Der zugehörige DEA entsteht so:

1. **Startzustand** des DEA ist die Menge $\{z_0\}$.
2. Für jede bereits erzeugte Menge M und jedes Zeichen c bildet man die Menge aller Zustände, die der NEA von *irgendeinem* Zustand aus M mit c erreichen kann. Das ist der Folgezustand.
3. Neu entstandene Mengen werden genauso behandelt, bis keine neue mehr auftritt.
4. **Endzustände** des DEA sind genau die Mengen, die mindestens einen Endzustand des NEA enthalten.

Beispiel: Wörter über {a, b}, die auf ab enden

Der NEA rät, wann das letzte `ab` beginnt:



Der NEA. In z_0 hat er bei a zwei Möglichkeiten — das ist der Nichtdeterminismus.

	a	b
z_0	$\{z_0, z_1\}$	$\{z_0\}$
z_1	$\emptyset$	$\{z_2\}$
z_2	$\emptyset$	$\emptyset$

Endzustand ist z_2. In z_0 ist der Automat nichtdeterministisch: Bei a kann er dort bleiben oder nach z_1 wechseln.

Im Simulator sehen Sie die Zustands**menge** wandern — genau das, was die Teilmengenkonstruktion gleich zu einzelnen Zuständen zusammenfasst:

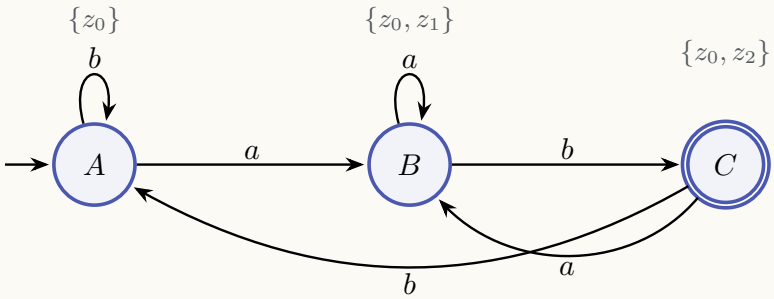
> In der Webfassung steht hier ein Bedienfeld zum Ausprobieren: **Automatensimulator**.

Durchführung. Start ist $A = \{z_0\}$.

Menge	mit a	mit b
$A = \{z_0\}$	$\{z_0, z_1\} = B$	$\{z_0\} = A$
$B = \{z_0, z_1\}$	$\{z_0, z_1\} = B$	$\{z_0, z_2\} = C$
$C = \{z_0, z_2\}$	$\{z_0, z_1\} = B$	$\{z_0\} = A$

Keine neue Menge mehr — fertig. C enthält z_2 und ist damit der einzige Endzustand.

Der fertige DEA:



Der konstruierte DEA. Über jedem Zustand steht die Menge von NEA-Zuständen, aus der er entstanden ist.

Übergangstabelle des DEA:

	a	b
$\to A$	B	A
B	B	C
C^*	B	A

Probe: bab läuft $A \to A \to B \to C$ — akzeptiert. aba läuft $A \to B \to C \to B$ — abgelehnt, richtig, denn aba endet nicht auf ab.

Merke

Hat der NEA n Zustände, hat der DEA höchstens 2^n — so viele Teilmengen gibt es. Meist sind es weit weniger, weil die meisten Mengen nie erreicht werden. Im Beispiel waren es 3 statt der möglichen 8.

Arbeitsauftrag 7

27. Konstruieren Sie mit der Teilmengenkonstruktion einen DEA zu dem NEA über $\{0,1\}$ mit $f(z_0,0)=\{z_0,z_1\}$, $f(z_0,1)=\{z_0\}$, $f(z_1,1)=\{z_2\}$, $F=\{z_2\}$. Welche Sprache akzeptiert er?
28. Geben Sie zum DEA aus dem Beispiel das 5-Tupel vollständig an.
29. Zeichnen Sie einen NEA, der alle Wörter über $\{a,b\}$ akzeptiert, die `aa` **oder** `bb` enthalten, und wandeln Sie ihn um. Wie viele Zustände hat der DEA?

Gymnasium Ottweiler

5.7 Automaten mit Ausgabe

LK

LK

Der GK-Lehrplan nennt ausdrücklich „endliche Automaten **ohne** Ausgabe". Mealy- und Moore-Automaten samt Ausgabetabelle stehen nur im LK-Lehrplan — dort sogar mit genau dem Beispiel, das hier durchgerechnet wird: Erzeuger und Prüfer eines Paritätsbits.

Bisher konnte unser Automat nur eines: am Ende ja oder nein sagen. Ein Getränkeautomat tut mehr — er gibt Wechselgeld heraus, schaltet eine Lampe ein, wirft eine Flasche aus. Solche Maschinen liefern **während** des Lesens Ausgaben.

Der Akzeptor ist dafür der Sonderfall: ein Automat, dessen einzige Ausgabe die Antwort am Schluss ist. Die allgemeine Form hat ein **Ausgabealphabet** dazu.

Definition: Mealy- und Moore-Automat

Ein **endlicher Automat mit Ausgabe** ist ein **6-Tupel** $M = (Z, A, B, f, g, z_0)$: Zu Zustandsmenge, Eingabealphabet, Übergangsfunktion und Anfangszustand kommen ein **Ausgabealphabet** B und eine **Ausgabefunktion** g hinzu. Eine Finalmenge F gibt es nicht — die Antwort steckt in der Ausgabe, nicht im Zustand am Ende. Zwei Bauarten unterscheiden sich darin, *woran* die Ausgabe hängt:

- **Mealy-Automat**: $g : Z \times A \to B$ — die Ausgabe hängt an **Zustand und Eingabe**, steht also am Übergang. Geschrieben wird sie an den Pfeil als `Eingabe / Ausgabe`.
- **Moore-Automat**: $g : Z \to B$ — die Ausgabe hängt **nur am Zustand**. Sie steht im Kreis, die Pfeile tragen nur die Eingabe.

Beide sind gleich mächtig: Zu jedem Mealy-Automaten gibt es einen Moore-Automaten mit derselben Wirkung und umgekehrt. Der Mealy-Automat kommt dabei meist mit weniger Zuständen aus, der Moore-Automat ist leichter zu lesen.

Eine Ampel ist ein Moore-Automat: Welche Lampen leuchten, hängt allein an ihrem Zustand, und der Takt schaltet sie von Phase zu Phase. Der Getränkeautomat verhält sich wie ein Mealy-Automat: Die Flasche fällt in dem Moment, in dem die letzte Münze eingeworfen wird — die Ausgabe gehört zum Übergang, nicht zum Zustand.

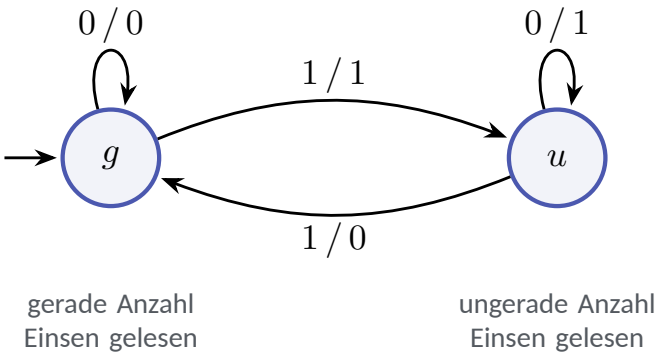
In der Webfassung steht hier ein Bedienfeld zum Ausprobieren: **Zustandsmaschine**.

In der Webfassung steht hier ein Bedienfeld zum Ausprobieren: **Zustandsmaschine**.

5.7.1 Beispiel: ein Paritätsbit erzeugen

Beim Übertragen von Daten hängt man oft ein **Paritätsbit** an: ein zusätzliches Bit, das so gewählt wird, dass die Gesamtzahl der Einsen **gerade** ist. Kippt unterwegs ein einzelnes Bit, stimmt die Rechnung nicht mehr und der Fehler fällt auf.

Der Erzeuger braucht dafür genau zwei Zustände — mehr muss er sich nicht merken:



Mealy-Automat zur Erzeugung des Paritätsbits. Am Pfeil steht `Eingabe / Ausgabe`.

Die Ausgabe ist jeweils die Parität **nach** dem gelesenen Zeichen. Das zuletzt ausgegebene Bit ist damit das gesuchte Paritätsbit.

Dieselbe Information als **Übergangs- und Ausgabetabelle**:

Zustand	bei 0	Ausgabe	bei 1	Ausgabe
g (gerade)	g	0	u	1
u (ungerade)	u	1	g	0

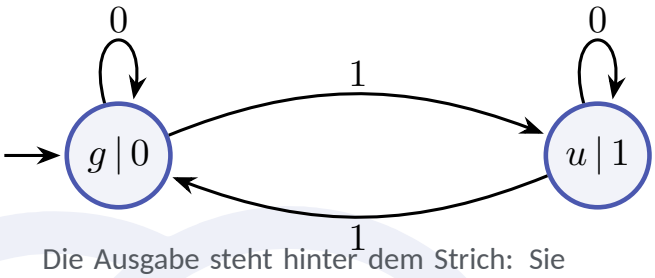
Gymnasium Ottweiler

Beispiel: Das Datenwort 1011

gelesen		**1**	**0**	**1**	**1**
Zustand	g	u	u	g	u
Ausgabe		1	1	0	1

Das letzte Ausgabebit ist 1 — gesendet wird also `10111`. Die Probe: `10111` enthält vier Einsen, und vier ist gerade.

Derselbe Automat als **Moore-Maschine**. Die Ausgabe wandert vom Pfeil in den Zustand:



Die Ausgabe steht hinter dem Strich: Sie gehört zum Zustand, nicht zum Übergang. Die Pfeile tragen nur noch die Eingabe.

Moore-Automat für dieselbe Aufgabe.

Die Tabelle wird dadurch schmaler — die Ausgabespalte gehört jetzt zum Zustand, nicht zum Übergang:

Zustand	**Ausgabe**	**bei 0**	**bei 1**
g (gerade)	0	g	u
u (ungerade)	1	u	g

Gymnasium Ottweiler

Stolperstein

Zählen Sie die Ausgaben nach: Der Mealy-Automat liefert zu `1011` die vier Bits $1, 1, 0, 1$ — eines pro gelesenem Zeichen. Der Moore-Automat liefert fünf, nämlich $0, 1, 1, 0, 1$: Er gibt schon im Anfangszustand aus, bevor überhaupt ein Zeichen gelesen ist. Das letzte Bit ist in beiden Fällen dasselbe und damit das gesuchte Paritätsbit; bei einem Wort der Länge n hat ein Mealy-Automat aber n Ausgaben und ein Moore-Automat $n + 1$. „Gleich mächtig" heißt also nicht „Ausgabe für Ausgabe gleich".

5.7.2 Beispiel: das Paritätsbit prüfen

Auf der Empfängerseite genügt wieder ein Akzeptor — hier interessiert nur ja oder nein. Er liest Datenwort *und* Paritätsbit und akzeptiert, wenn die Zahl der Einsen gerade ist:

$$M = (\{g, u\},\ \{0, 1\},\ f,\ g,\ \{g\}) \quad \text{mit} \quad f(g,0) = g,\ f(g,1) = u,\ f(u,0) = u,\ f(u,1) = g$$

Vergleichen Sie ihn mit dem Automaten aus Aufgabe 26: Dieselben Übergänge, aber der jeweils andere Zustand ist Endzustand. Zwei Automaten, die sich nur in der Finalmenge unterscheiden, akzeptieren genau die zueinander komplementären Sprachen.

Merke

Erzeuger und Prüfer haben denselben Zustandsgraphen. Der Unterschied liegt allein darin, was man von ihm will: Der **Erzeuger** gibt bei jedem Schritt etwas aus, der **Prüfer** sagt am Ende nur, ob er in einem Endzustand steht.

Stolperstein

Ein Paritätsbit erkennt **einen** gekippten Bitfehler sicher. Kippen **zwei** Bits, ist die Parität wieder in Ordnung und der Fehler bleibt unentdeckt. Und korrigieren kann es gar nichts — es sagt nur, dass etwas falsch ist, nicht was.

Arbeitsauftrag 8

30. Lassen Sie den Erzeuger auf `1101` und `0000` laufen. Notieren Sie Zustände und Ausgaben wie in der Tabelle oben. Welches Paritätsbit wird angehängt?
31. Prüfen Sie mit dem Prüfer die Wörter `10111`, `10011` und `11011`. Welche sind unterwegs verfälscht worden?
32. Zeichnen Sie einen Mealy-Automaten für einen Getränkeautomaten, der 50-Cent- und 1-Euro-Stücke annimmt und bei 1,50 € eine Flasche ausgibt. Was gibt er bei 2 € aus?
33. Ein Moore-Automat gibt bei jedem gelesenen Zeichen aus, ob die bisher gelesene Zeichenzahl gerade ist. Zeichnen Sie ihn.
34. Geben Sie denselben Automaten als Mealy-Automat an. Wie viele Ausgaben liefern die beiden zu einem Wort der Länge 6?
35. Begründen Sie, warum der Akzeptor ein Sonderfall des Automaten mit Ausgabe ist.

5.8 Grammatik und Automat sind zwei Seiten derselben Sache

Eine Bauanleitung für ein Klemmbausteinmodell sagt, wie man es Schritt für Schritt *aufbaut*. Wer ein fertiges Modell vor sich hat, kann umgekehrt Stein für Stein *prüfen*, ob es nach dieser Anleitung gebaut ist. Beides beschreibt dasselbe Modell, einmal erzeugend, einmal prüfend. Für reguläre Sprachen sind Grammatik und Automat genau dieses Paar.

Zwischen regulären Grammatiken und endlichen Automaten lässt sich mechanisch hin- und herübersetzen.

Von der Grammatik zum Automaten. Jedes Nichtterminal wird ein Zustand, das Startsymbol der Startzustand. Aus $A \to xB$ wird der Übergang $f(A, x) = B$. Aus $A \to x$ wird ein Übergang in einen zusätzlichen Endzustand E.

Beispiel mit $P = \{S \to 1A,\ A \to 0A \mid 1A \mid 0\}$ (gerade Dualzahlen). Die Regeln $A \to 0A$ und $A \to 0$ beginnen beide mit einer 0 — der Zustand A hat bei 0 also **zwei** Nachfolger:

Zustand	bei 0	bei 1	Endzustand
S	–	$\{A\}$	nein
A	$\{A, E\}$	$\{A\}$	nein
E	–	–	**ja**

Stolperstein

Die Konstruktion liefert im Allgemeinen einen **NEA**, keinen DEA — immer dann, wenn zu einem Nichtterminal zwei Regeln mit demselben Terminalzeichen gehören. Das ist kein Mangel: Mit der Teilmengenkonstruktion aus 5.6.2 wird daraus ein DEA.

Hier von Hand nachgebaut: Aus $\{A, E\}$ wird ein DEA-Zustand, den wir E' nennen. Er ist Endzustand, weil er E enthält, und verhält sich sonst wie A:

Zustand	bei 0	bei 1	Endzustand
S	–	A	nein
A	E'	A	nein
E'	E'	A	**ja**

Probe: 1010 läuft $S \to A \to E' \to A \to E'$ — akzeptiert. 1011 endet in A und wird abgelehnt, richtig, denn 1011 ist ungerade.

Stolperstein

In der Tabelle steht bei S und der 0 ein Strich: Dieser Übergang kommt in der Grammatik nicht vor, weil keine Dualzahl mit 0 beginnt. Streng genommen verlangt die Definition aus 5.6 aber eine **auf ganz** $Z \times A$ **definierte** Übergangsfunktion. Man ergänzt deshalb einen zusätzlichen **Fehlerzustand** z_F, der nie Endzustand ist und alle Zeichen auf sich selbst abbildet; jeder Strich führt dorthin. Gezeichnet wird er meist weggelassen, weil er das Bild unübersichtlich macht. Im Programm später entspricht ihm der Zweig „Übergang nicht in der Tabelle → abweisen".

Vom Automaten zur Grammatik. Umgekehrt wird jeder Zustand ein Nichtterminal, der Startzustand das Startsymbol. Aus $f(Z_1, x) = Z_2$ wird die Produktion $Z_1 \to xZ_2$ — und ist Z_2 ein **Endzustand**, so kommt zusätzlich die abschließende Regel $Z_1 \to x$ hinzu.

Die naheliegende Abkürzung $Z \to \varepsilon$ für jeden Endzustand wäre bequem, ist nach unserer Definition aus 5.5 aber nicht erlaubt: Dort darf rechts nur xB oder x stehen, niemals das leere Wort. Deshalb der Umweg über die zusätzliche Regel.

Merke

Reguläre Grammatik, endlicher Automat und regulärer Ausdruck (Abschnitt 5.11) beschreiben genau dieselbe Klasse von Sprachen — die regulären Sprachen. Drei Schreibweisen, ein Begriff.

Ein Sonderfall bleibt: Nach unserer Definition erzeugt jede Produktion mindestens ein Zeichen, das leere Wort ε ist also in keiner so erzeugten Sprache enthalten. Ein Automat kann es dagegen akzeptieren — nämlich dann, wenn sein Startzustand ein Endzustand ist. Für diesen einen Fall erlaubt man zusätzlich die Regel $S \to \varepsilon$ für das Startsymbol. Mit ihr stimmen die drei Beschreibungen dann ohne Ausnahme überein.

Gymnasium Ottweiler

5.9 Einen Automaten programmieren

Die Wache in einem Schleichspiel patrouilliert, bis sie Sie sieht. Dann verfolgt sie Sie; verliert sie Sie aus den Augen, sucht sie eine Weile und kehrt schließlich zur Patrouille zurück. Spieleentwickler programmieren solches Verhalten sehr oft als endlichen Automaten: eine Variable für den Zustand, eine Tabelle für die Übergänge. Wie wenig Code das ist, zeigt dieser Abschnitt.

> In der Webfassung steht hier ein Bedienfeld zum Ausprobieren: **Zustandsmaschine**.

Die Übergangstabelle lässt sich unmittelbar in Code übersetzen. Das Programm hält nur eine einzige Variable fest: den aktuellen Zustand.

Python

```python
# Automat: akzeptiert alle Woerter ueber {a, b}, die "aa" enthalten
UEBERGAENGE = {
    ("z0", "a"): "z1", ("z0", "b"): "z0",
    ("z1", "a"): "z2", ("z1", "b"): "z0",
    ("z2", "a"): "z2", ("z2", "b"): "z2",
}
START = "z0"
ENDZUSTAENDE = {"z2"}


def akzeptiert(wort):
    zustand = START
    for zeichen in wort:
        if (zustand, zeichen) not in UEBERGAENGE:
            return False                    # Zeichen nicht im Alphabet
        zustand = UEBERGAENGE[(zustand, zeichen)]
    return zustand in ENDZUSTAENDE


for wort in ["abab", "abba", "baab", "aa", ""]:
    print(wort, akzeptiert(wort))
```

Object Pascal

Gymnasium Ottweiler

```
program Automat;

type
  TZustand = (z0, z1, z2);

function Uebergang(zustand: TZustand; zeichen: Char): TZustand;
begin
  Result := zustand;
  case zustand of
    z0: if zeichen = 'a' then Result := z1 else Result := z0;
    z1: if zeichen = 'a' then Result := z2 else Result := z0;
    z2: Result := z2;                        { Endzustand bleibt erhalten }
  end;
end;

function Akzeptiert(const wort: string): Boolean;
var
  zustand: TZustand;
  i: Integer;
begin
  zustand := z0;
  for i := 1 to Length(wort) do
    zustand := Uebergang(zustand, wort[i]);
  Result := zustand = z2;
end;

begin
  WriteLn(Akzeptiert('abab'));               { False }
  WriteLn(Akzeptiert('baab'));               { True  }
end.
```

Beachten Sie den Unterschied im Aufbau: Python hält die Tabelle als Datenstruktur, Pascal als `case`-Verzweigung. Die Datenstruktur ist der bessere Weg — man kann einen anderen Automaten einsetzen, ohne den Code zu ändern.

Arbeitsauftrag 9

36. Erweitern Sie das Python-Programm so, dass es den durchlaufenen Zustandsweg ausgibt.
37. Ersetzen Sie die Tabelle durch die Ihres Automaten für „endet auf abb” und testen Sie.
38. Schreiben Sie die Pascal-Fassung so um, dass sie eine Übergangstabelle als Array `array[TZustand, 'a'..'b'] of TZustand` benutzt statt der `case`-Anweisung.
39. Schreiben Sie ein Programm, das einen Text zeichenweise liest und prüft, ob er ein gültiger Bezeichner ist.
40. **Projekt:** Ein Programm liest eine Übergangstabelle aus einer Textdatei und prüft dann beliebige Wörter. Damit haben Sie einen Automaten-Simulator geschrieben.

5.10 Wo endliche Automaten scheitern

LK

LK

Die Chomsky-Hierarchie und die Beispiele nicht-regulärer und nicht-kontextfreier Sprachen stehen nur im LK-Lehrplan. Im Grundkurs genügt die Einsicht aus dem ersten Absatz: Der Zustand eines endlichen Automaten ist sein ganzes Gedächtnis, und das ist endlich.

Ein Türsteher soll am Ende des Abends sagen, ob alle Gäste wieder draußen sind. Mit einer Strichliste ist das leicht. Darf er sich aber nur merken, ob „wenig“, „mittel“ oder „viel“ los ist, kann er es nicht — egal, wie aufmerksam er ist. Genau in dieser Lage ist ein endlicher Automat: Er hat nur endlich viele Zustände, die Gästeschlange ist aber beliebig lang.

Konstruieren Sie einen Automaten, der genau die Wörter $a^n b^n$ akzeptiert, also ab, $aabb$, $aaabbb$ und so weiter — gleich viele a wie b.

Es gelingt nicht, und der Grund ist grundsätzlich: Der Automat müsste sich merken, wie viele a er gelesen hat, bevor das erste b kommt. Diese Zahl ist unbeschränkt, seine Zustandsmenge aber endlich. Bei genügend langer Eingabe muss er zwangsläufig zweimal in denselben Zustand geraten, obwohl er unterschiedlich viele a gesehen hat — und dann kann er die beiden Fälle nicht mehr unterscheiden.

Dasselbe gilt für die Klammersprache aus 5.4: Verschachtelte Klammern erfordern ein unbeschränktes Gedächtnis. Genau deshalb war jene Grammatik nicht regulär.

Eine Maschine, die $a^n b^n$ sehr wohl erkennt, kennen Sie schon: die **Turingmaschine** aus Abschnitt 1.2. Sie streicht Paare durch und schreibt sich die Zwischenstände aufs Band. Der Unterschied ist genau dieser Schreibplatz — der endliche Automat hat keinen.

Merke

Endliche Automaten erkennen genau die regulären Sprachen. Alles, was Zählen oder Schachteln erfordert, liegt außerhalb.

Für solche Sprachen braucht man mächtigere Maschinen. Ein **Kellerautomat** besitzt zusätzlich einen Stapelspeicher: Bei jedem a legt er eine Marke ab, bei jedem b nimmt er eine weg — und akzeptiert, wenn der Stapel am Ende leer ist. Damit erkennt er die kontextfreien Sprachen, zu denen auch die Klammerstrukturen der Programmiersprachen gehören. Ganz oben steht die Turingmaschine aus Abschnitt 1.2. Diese Staffelung heißt **Chomsky-Hierarchie**:

Typ	Sprachklasse	Maschine	Form der Produktionen	Beispiel
3	regulär	endlicher Automat	$A \to xB$ oder $A \to x$	Bezeichner, Zahlen
2	kontextfrei	Kellerautomat	$A \to \alpha$ (links ein Nichtterminal)	$a^n b^n$, Klammern
1	kontextsensitiv	linear beschränkter Automat	$\alpha \to \beta$ mit $\lvert\alpha\rvert \le \lvert\beta\rvert$	$a^n b^n c^n$
0	rekursiv aufzählbar	Turingmaschine	beliebig	haltende Programme

In Worten, von der schwächsten zur mächtigsten Stufe:

- **Typ 3, regulär.** Das Gedächtnis ist endlich. Die Maschine merkt sich eine von endlich vielen Lagen, sonst nichts. Sie kann erkennen, *dass* etwas vorkam, aber nicht, *wie oft*.
- **Typ 2, kontextfrei.** Dazu kommt ein Stapel. Damit lässt sich zählen und schachteln, aber immer nur **eine** Sache zur Zeit — was heruntergenommen ist, ist weg.
- **Typ 1, kontextsensitiv.** Der Speicher wächst mit der Eingabe. Jede Regel darf die Satzform nur verlängern, nie verkürzen; deshalb bleibt die Wortfrage entscheidbar.
- **Typ 0, rekursiv aufzählbar.** Keine Beschränkung mehr. Das ist genau das, was eine Turingmaschine aufzählen kann — und hier wird die Wortfrage unentscheidbar: Gehört das Wort zur Sprache, hält die Maschine irgendwann und sagt ja; gehört es nicht dazu, läuft sie möglicherweise für immer. Das ist genau die *partielle* Entscheidbarkeit aus Abschnitt 4.4. „Rekursiv aufzählbar" ist deshalb **nicht** dasselbe wie „berechenbar".

Die vier Klassen sind echt ineinander geschachtelt — jede Stufe kann alles, was die vorige kann, und mehr:

$$\text{Typ 3} \subsetneq \text{Typ 2} \subsetneq \text{Typ 1} \subsetneq \text{Typ 0}$$

Zwei Beispiele markieren die beiden Sprünge, die in der Schule am häufigsten gebraucht werden:

Merke

$L_1 = \{a^n b^n \mid n \geq 1\}$ ist **nicht regulär**, aber kontextfrei. Der Grund steht oben: Ein endlicher Automat kann die a nicht zählen, ein Kellerautomat schon.

$L_2 = \{a^n b^n c^n \mid n \geq 1\}$ ist **nicht kontextfrei**, aber kontextsensitiv. Hier reicht auch der Stapel nicht: Beim Abarbeiten der b wird er geleert — für die c ist die Zahl der a dann verloren. Man müsste **zweimal** dasselbe zählen, und dafür braucht es ein Band statt eines Stapels.

Das erklärt nebenbei den Aufbau eines Compilers: Der Lexer erkennt Token — die sind regulär, ein endlicher Automat genügt. Der Parser prüft die Struktur — die ist kontextfrei und braucht einen Stapel.

Arbeitsauftrag 10

41. Versuchen Sie, einen Automaten für $a^n b^n$ mit höchstens fünf Zuständen zu bauen. Beschreiben Sie, woran es scheitert, und geben Sie ein Wort an, das Ihr Automat fälschlich akzeptiert.
42. Begründen Sie, warum ein Automat mit 100 Zuständen die Sprache $a^n b^n$ für $n \leq 40$ korrekt entscheiden kann — und warum das nichts nützt.
43. Ordnen Sie ein: Telefonnummern, korrekt geklammerte Terme, Datumsangaben im Format TT.MM.JJJJ, vollständige Pascal-Programme.

5.11 Reguläre Ausdrücke in der Praxis

ZUSATZ

> ZUSATZ
>
> Reguläre Ausdrücke stehen in keinem der beiden Lehrpläne. Sie sind hier aufgenommen, weil sie dieselbe Sprachklasse beschreiben wie Grammatik und Automat — und weil sie das einzige Werkzeug dieses Kapitels sind, das Ihnen im Alltag jede Woche begegnet.

Sie wollen in einem Chatverlauf alle Nachrichten finden, in denen eine Uhrzeit steht, egal ob 7:30 oder 18:05. Die normale Suche verlangt jede Uhrzeit einzeln. Ein regulärer Ausdruck beschreibt stattdessen das *Muster* einmal: `[0-9]+:[0-9][0-9]` — eine oder mehr Ziffern, ein Doppelpunkt, zwei Ziffern. Er benutzt nur Bausteine aus der Tabelle dieses Abschnitts.

Was in 5.5 als Grammatik und in 5.6 als Automat auftrat, schreibt man im Alltag als **regulären Ausdruck**. Editoren, Suchfunktionen, Formularprüfungen und Logauswertungen benutzen sie täglich.

Ausdruck	Bedeutung	passt auf
`abb`	genau diese Folge	`abb`
`a*`	beliebig viele a, auch keines	`ε`, `a`, `aaa`
`a+`	mindestens ein a	`a`, `aaa`
`a?`	höchstens ein a	`ε`, `a`
`[a-z]`	ein Zeichen aus dem Bereich	`k`
`(ab)+`	Wiederholung der Gruppe	`ab`, `ababab`
`a|b`	Alternative	`a` oder `b`

Ein Bezeichner ist damit `[A-Za-z][A-Za-z0-9]*`, eine Dualzahl ohne führende Null `1[01]*`, eine gerade Dualzahl `1[01]*0`.

```python
import re

muster = re.compile(r"[A-Za-z][A-Za-z0-9]*$")
for text in ["zaehler", "2fast", "x1", "mein wert"]:
    print(text, bool(muster.match(text)))
```

Unter der Haube übersetzt die Bibliothek den Ausdruck in einen endlichen Automaten — und läuft ihn genau so ab, wie Sie es in 5.9 programmiert haben.

Arbeitsauftrag 11

44. Schreiben Sie reguläre Ausdrücke für: Wörter über $\{a, b\}$ mit `aa`; ungerade Dualzahlen; deutsche Postleitzahlen; eine E-Mail-Adresse in grober Näherung.
45. Prüfen Sie mit dem Python-Beispiel Ihre Ausdrücke aus Aufgabe 44.
46. Begründen Sie, warum sich mit regulären Ausdrücken nicht prüfen lässt, ob in einem Term gleich viele öffnende wie schließende Klammern stehen.

5.12 Zusammenfassung

- Syntax beschreibt die Form, Semantik die Bedeutung. Maschinen prüfen zuverlässig nur die Form.
- Ein Alphabet A ist eine endliche Zeichenmenge, A^* die Menge aller Wörter darüber, eine formale Sprache eine Teilmenge von A^*.
- Eine Grammatik (V_N, V_T, P, S) erzeugt Wörter durch Ableitungen; sie ist regulär, wenn jede rechte Seite aus genau einem Terminalzeichen und höchstens einem Nichtterminal dahinter besteht.
- Ein endlicher Automat erkennt Wörter. Sein Zustand ist sein gesamtes Gedächtnis.
- Reguläre Grammatik, endlicher Automat und regulärer Ausdruck beschreiben dieselbe Sprachklasse und lassen sich ineinander überführen.
- Nichtdeterminismus verkürzt Diagramme, erweitert aber die Sprachklasse nicht.
- Zählen und Schachteln liegen außerhalb der regulären Sprachen; dafür braucht es Kellerautomaten und die höheren Stufen der Chomsky-Hierarchie.

Wiederholungsaufgaben

47. Erklären Sie die vier Bestandteile einer Grammatik an einem selbst gewählten Beispiel.
48. Geben Sie zur Sprache „alle Wörter über $\{0, 1\}$, die mit 0 beginnen und auf 1 enden" eine reguläre Grammatik, einen DEA und einen regulären Ausdruck an.
49. Wandeln Sie den Automaten aus Aufgabe 48 in eine Grammatik zurück und prüfen Sie, ob Sie dieselbe Sprache erhalten.
50. Erklären Sie einem Mitschüler in fünf Sätzen, warum ein endlicher Automat keine Klammern prüfen kann.
51. Welche Teile eines Compilers arbeiten mit endlichen Automaten, welche mit Grammatiken? Begründen Sie.

Gymnasium Ottweiler