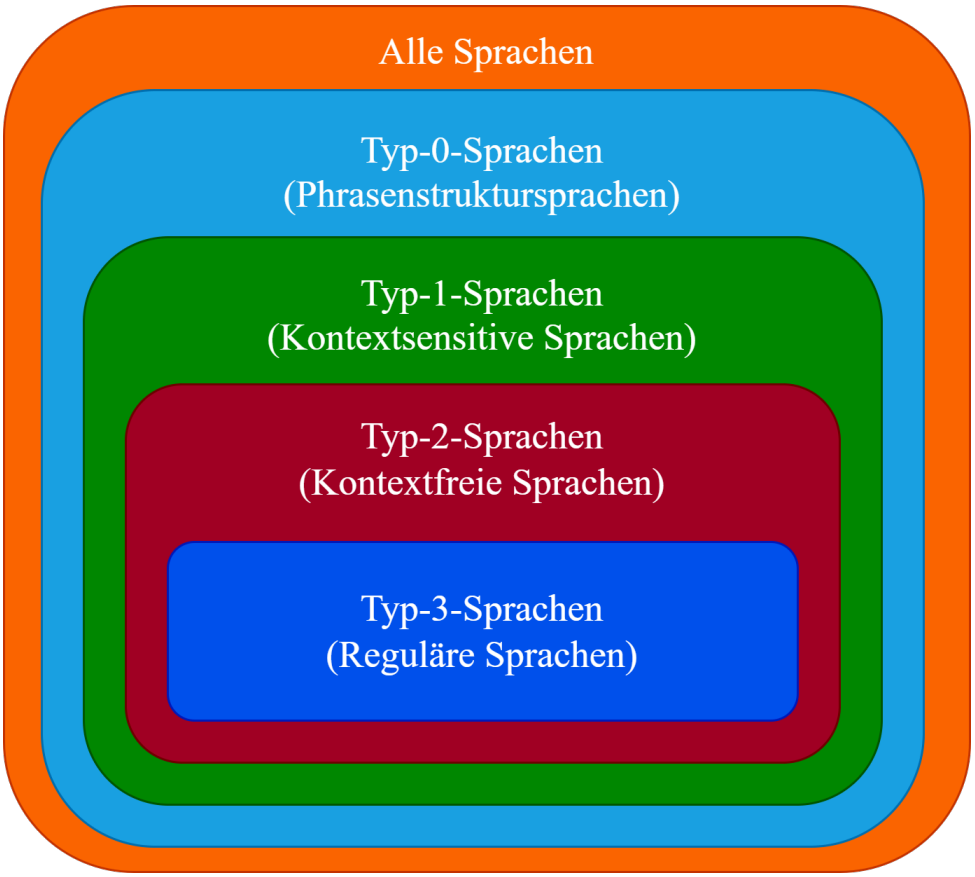
Formale Sprachen



Inhaltsverzeichnis

1. Sprachen

1.1. Einleitung

Auf den ersten Blick erscheint es etwas rätselhaft, warum sich die Informatik mit der Sprachlehre beschäftigen sollte. Auf den zweiten Blick wird dies aber schon klarer, da es nicht umsonst um **Programmiersprachen** genannt wird.

Jede Sprache hat ihre eigene **Syntax** und es steckt eine gewisse **Semantik** hinter den Begriffen, beide Begriffe werden im Folgenden noch ausführlich besprochen.

Die formalen Sprachen sind ein wichtiges Teilgebiet der **theoretischen Informatik**, d.h. in diesem Schuljahr stehen weniger die praktischen Dinge im Vordergrund, sondern der theoretische Unterbau.

Die Sprachentheorie kommt in vielen Bereichen der Informatik zum Einsatz, insbesondere aber in einem der zentralsten Bereiche, dem **Compilerbau**. Einer Programmiersprache liegt eine sogenannte **Grammatik** zugrunde, die letztendlich festlegt was ein "gültiges Programm" (d.h. syntaktisch korrekt!) darstellt. Der Compiler muss also einerseits diese Struktur überprüfen, andererseits dann eine Übersetzung in Maschinensprache oder eine andere Programmiersprache vornehmen.

Ein (sehr grober) Überblick über die Funktionsweise eines Compilers wäre also:

1. **Syntaxanalyse**: Der Compiler nutzt die formalen Regeln der Sprache, um den Quellcode syntaktisch zu analysieren, d.h. es wird geklärt: "Handelt es sich um ein korrektes Programm".
2. **Semantische Analyse**: Es existieren in der Regel auch weitere formale Regeln, die die **Bedeutung** (Semantik) des Programms überprüfen sollen. Das geht natürlich nicht vollständig, aber beispielsweise können die zulässigen Bereiche von Variablen oder die korrekte Verwendung von Typen ("starke Typisierung von Java" z.B.!) überprüft werden.
3. **Übersetzung und Optimierung**: Viele Compiler optimieren den geschriebenen Code und erstellen eine *Zwischendarstellung*, um die Performance zu steigern.
4. **Codegenerierung**: Zuletzt muss die optimierte Darstellung noch in den tatsächlich von der Maschine ausführbaren Code überführt werden. (oder alternativ in die Darstellung der Ziel-Programmiersprache)

> ***Hinweis***
>
> Da es sich bei diesem Thema um ein sehr schwieriges handelt - es gibt mehrere Vorlesungen, die sich ausschließlich damit im Studium beschäftigen - können wir in der Schule nur eine sehr abgespeckte Variante dieser Theorien behandeln, die sich in der Regel nicht auf Programmiersprachen, sondern auf andere Ausdrücke bezieht, die nach bestimmten Regeln aufgebaut werden, z.B. Email-Adressen, ISBN-Nummern, Autokennzeichen, etc.

Das folgende Unterkapitel klärt zunächst grundlegende Begriffe etwas genauer.

1.2. Grundlagen

Zunächst muss definiert werden, was eine formale Sprache eigentlich sein soll:

Definition

Eine formale Sprache L besteht aus einer Menge von **Zeichenketten (Wörtern)** eines **Alphabets Σ**, das alle benutzbaren **Zeichen (Token)** beinhaltet.

Kurz: $L \subseteq \Sigma^*$

Der Ausdruck Σ^* ist eine Kurzschreibweise für alle möglichen Wörter, die aus einem Alphabet gebildet werden können. Besteht das Alphabet z.B. nur aus einem einzigen Buchstaben $\Sigma = \{a\}$, so könnten die folgenden Wörter damit gebildet werden:

$$\Sigma^* = \{\varepsilon, a, aa, aaa, \cdots\}$$

Dabei steht ε für das "leere Wort" (kein Leerzeichen, sondern einfach "kein Wort", so wie in Java "" einem leeren String entspricht - es ist offiziell zwar eine Zeichenkette, enthält aber kein Zeichen!). in Σ^* stecken also **alle** möglichen Wörter. Unsere formale Sprache muss aber nicht alle enthalten. Wir könnten z.B. die Sprache definieren, die nur eine geradzahlige Anzahl an a's enthält, also:

$$L = \{\varepsilon, aa, aaaa, \cdots\}$$

Und offensichtlich gilt $L \subseteq \Sigma^*$

Weitere Beispiele:

1. **Natürliche Sprachen**: Am Beispiel Deutsch. Unser Alphabet besteht aus den folgenden Zeichen:

   $$\Sigma = \{a; \cdots z; A; \cdots Z; \text{ä}; \text{Ä}; \text{ö}; \text{Ö}; \text{ü}; \text{Ü}; \text{ß}; !; "; ";; ,; .; :; \text{Leerzeichen}\}$$

   Σ^* ist dann die Menge aller möglichen Zeichenketten, allerdings gehören nicht alle zu unserer Sprache!

Gehören zur Sprache	Gehören nicht zur Sprache
Informatik	Infom
Pizza	Pazzizo
Das ist ein deutscher Satz!	sjg aksjdfkajs aksjkkii!

   Die Analyse natürlicher Sprachen beschäftigt Linguisten schon lange, bisher entziehen sich diese aber einer strengen formalen Klassifikation - d.h. die Regeln für die Bildung deutscher, englischer, französischer und anderer Sprachen sind nicht eindeutig bzw. nicht vollständig bekannt (wenig überraschend!). Wir werden uns mit schöneren (d.h. eindeutigeren) **künstlichen** Sprachen beschäftigen.
2. **ISBN-Nummern**: Buchnummern bestehen aus fünf-Zahlengruppen, z.B.:

   $$978 - 3 - 86680 - 192 - 9$$

   Dabei gibt es natürlich auch gewisse Regeln, die dritte fünfstellige Nummer bezeichnet den Verlag, d.h. nicht jede fünfstellige Zahl existiert als Verlagsnummer.

An diesem Beispiel lässt sich der Unterschied zwischen **syntaktisch** und **semantisch** gut herausarbeiten:

1. Das "Wort" 999 ist nicht Teil der Sprache, weil es **syntaktisch** falsch ist, es ist keine dreizehnstellige Zahl.
2. Das Wort $978-3-99999-192-9$ ist nicht Teil der Sprache, weil es die entsprechende Verlagsnummer nicht gibt. Es erfüllt **semantisch** nicht die Voraussetzungen, d.h. aufgrund der **Bedeutung**.

3. **Autokennzeichen**: z.B.: NM-AB 34. Offensichtlich ist nicht jede Zeichenkette auch ein existierendes (oder mögliches) Autokennzeichen.

> ***Hinweis***
>
> Ersteres ist natürlich wieder nichts, was man einfach formalisieren kann. Die Behandlung von syntaktischen Fehlern ist natürlich wesentlich einfacher als die Behandlung von semantischen Fehlern - anders gesagt: es ist vergleichsweise einfach zu entscheiden, ob ein Programm korrekt geschrieben ist (Syntax), aber weniger leicht zu entscheiden, ob es tut was es soll (Semantik).
>
> **Für Profis**: Streng genommen können beide Probleme natürlich nicht entscheidbar sein, siehe letztes Kapitel dieses Schuljahr und Satz von Rice.

4. **Mathematische Terme**: Selbsterklärend, Terme werden in der 7. Klasse im Wesentlichen sogar als "sinnvolle Aneinanderreihung von Zahlen, Variablen und Operationen" definiert. D.h. $2x + 3 + 7$ gehört zur Sprache, 2+++5 dagegen nicht.
5. **Viele weitere Beispiele**: Chemische Reaktionsgleichungen, Adressen, Nuklidschreibweise, Notation von Schachbewegungen, Geografische Koordinaten, Chat-Ausdrücke/Smileys, Barcodes, Steno, Raumnummern, Musiknoten, Telefonnummern, IP-Adressen, Morsecode, Militärische Handzeichen, Programmiersprachen etc.

Im Folgenden beschäftigen wir uns mit der Systematik, wie die **Regeln** für Sprachen beschrieben werden können, d.h.: wie können wir beschreiben, welche Wörter zu einer Sprache gehören und welche nicht - und das führt direkt zum Begriff der **Grammatik**.

2. Grammatiken

2.1. Grundlagen

Auch bei natürlichen Sprachen werden die Regeln zur Bildung von Sätzen durch Grammatiken festgelegt. Bei "künstlichen" Sprachen ist dies ähnlich, jedoch etwas systematischer als die "gewachsenen" natürlichen Sprachen.

Ein offensichtliches Problem der Beschreibung von allgemeinen Sprachen ist, dass eine Sprache unendlich viele korrekte Zeichenketten haben kann, trotz eines endlichen Alphabets (z.B. die Sprache aller Zeichenketten, die nur eine geradzahlige Anzahl an a's haben!).

Die Lösung sind **(endlich) viele Produktionsregeln**, die die Bildung dieser möglichen Worte beschrieben. Die Produktionsregeln zusammen mit dem Alphabet bilden dann die **Grammatik** einer Sprache.

Das zu einfache Beispiel:

Wir wollen die Sprache aller einstelligen Zahlen, sprich die Sprache der Ziffern definieren. Das zugrunde liegende Alphabet ist also $\Sigma = \{0, 1, \cdots, 9\}$. Die zugehörige Produktionsregel wird wie folgt notiert:

$$(1)\ \text{Ziffer} \rightarrow \text{'0'} \mid \text{'1'} \mid \text{'2'} \mid \text{'3'} \mid \text{'4'} \mid \text{'5'} \mid \text{'6'} \mid \text{'7'} \mid \text{'8'} \mid \text{'9'}$$

Diese Regel besteht aus:

1. einer **linken Seite**, hier steht ein **Nichtterminalsymbol**, d.h. ein Symbol, das nicht zum Alphabet gehört, sondern nur eine "Beschreibung" darstellt (das wird beim nächsten Beispiel deutlicher).
2. Zeichen des Alphabets (sogenannte **Terminalsymbole**) in Apostrophen - das sind die Zeichen des Alphabets, die wir mit dieser Regel "**produzieren**" können.
3. senkrechten Strichen |, die ein "oder" darstellen.

Die obige Regel kann also so interpretiert werden:

Das Nichtterminal "Ziffer" kann mit irgendeiner Ziffer von 0 bis 9 ersetzt werden.

Das sieht bisher noch nicht besonders nützlich aus, deswegen:

Ein etwas komplexeres Beispiel:

Als nächstes sollen alle dreistelligen Zahlen gebildet werden können. Man kann also eine ähnliche Regel wie oben benutzen, muss aber sicherstellen, dass die erste Ziffer keine 0 ist, da es sonst keine dreistellige Zahl wäre:

$$(1)\ \text{Zahl} \rightarrow \text{ErsteZiffer Ziffer Ziffer}$$
$$(2)\ \text{ErsteZiffer} \rightarrow \text{'1'} \mid \text{'2'} \mid \text{'3'} \mid \text{'4'} \mid \text{'5'} \mid \text{'6'} \mid \text{'7'} \mid \text{'8'} \mid \text{'9'}$$
$$(3)\ \text{Ziffer} \rightarrow \text{ErsteZiffer} \mid \text{'0'}$$

Die erste Zeile besteht dieses Mal nur aus Nichtterminalsymbolen und beschreibt die Struktur der Zahlen komplett, die übrigen Zeilen legen dann fest, welche Terminale, sprich welche Ziffern, in den einzelnen Schritten ersetzt werden dürfen. In der dritten Regel greifen wir dabei wieder auf die zweite Regel zurück, um Schreibarbeit zu sparen.

> ***Hinweis***
>
> Insbesondere in schriftlichen Prüfungen muss eindeutig sein, ob eine Zeichenkette **ein** oder **zwei** Nichtterminale darstellt. Im späteren Verlauf ist es üblich die Nichtterminalsymbole abzukürzen, man würde hier z.B. **E** für die ErsteZiffer nehmen und **Z** für die Ziffer. Der Ausdruck **EZ** und **E Z** kann dann aber durchaus unterschiedliche Bedeutung haben, wenn es noch eine weitere Regel mit $EZ \to \cdots$ gibt.
>
> **Quintessenz**: leserlich und eindeutig schreiben, deutliche Leerzeichen einbauen!

Das noch komplexere Beispiel:

Als letztes soll die Sprache aller dreistelligen Zahlen gebildet werden, die mit einer geraden Zahl starten und mit einer ungeraden Zahl enden, das zugehörige Alphabet sind wieder die Ziffern. Das können wir beispielsweise mit folgenden Regeln darstellen:

(1) Zahl → gerade Ziffer ungerade

(2) gerade → '2' | '4' | '6' | '8'

(3) ungerade → ''1 | '3' | '5' | '7' | '9'

(4) Ziffer → gerade | ungerade

Für ein bestimmtes Wort der Sprache können wir eine **Ableitung** (hat nichts mit der mathematischen Funktion zu tun!) angeben, d.h. eine Folge an Produktionsregeln, die aus dem Startsymbol (hier Zahl) das gesuchte Wort durch Ersetzungen bildet, z.B. wird die Zahl 211 wie folgt gebildet:

$$\text{Zahl} \overset{(1)}{\to} \text{gerade Ziffer ungerade} \overset{(2)}{\to} 2 \text{ Ziffer ungerade}$$
$$\overset{(4)}{\to} 2 \text{ ungerade ungerade} \overset{(3)}{\to} 21 \text{ ungerade} \overset{(3)}{\to} 211$$

Ist eine Zeichenfolge, bzw. hier Zahl nicht in der Sprache, so können wir keine entsprechende Reihenfolge finden.

Eine alternative Darstellung ist der sogenannte **Ableitungsbaum** (oder auch **Syntaxbaum**, für das obige Beispiel:

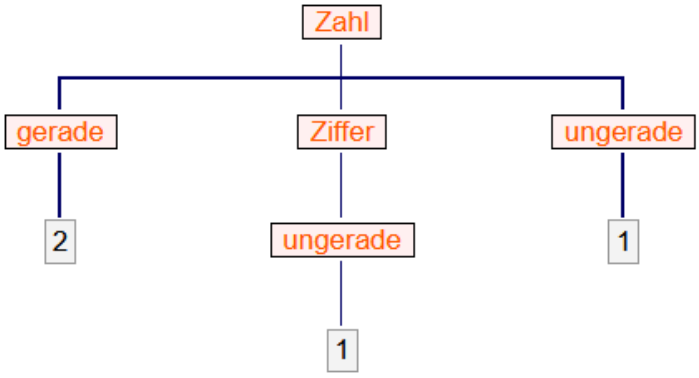


Figure 1: Erstellt mit Flaci

Die Wurzel des Baums ist das **Startsymbol**. Die jeweiligen Kinder des Knotens ergeben sich durch Anwendung einer Regel.

Es gilt außerdem:

- jedes Blatt muss ein **Terminalsymbol** sein.

- jeder innere Knoten muss ein **Nichtterminalsymbol** sein.

Zusammengefasst:

Definition

Eine **Grammatik** einer formalen Sprache wird definiert durch vier verschiende Eigenschaften bzw. Objekte:

1. **Vokabular V**: endliche Menge von Nichtterminalsymbolen.
2. **Alphabet Σ**: eine Menge von Zeichen/Terminalsymbolen.
3. **Startsymbol S**: das Startsymbol der Ableitung.
4. **Produktionsregeln P**: Regeln der Form:

   $$<\text{Nr.}>: <\text{Nichtterminal}> \rightarrow <\text{wird ersetzt durch}>$$

Kurz: G = (V, Σ, P, S)

Hinweise

1. Unsere Notation der Produktionsregeln orientiert sich an dem, was oft in Schulbüchern angegeben ist. Eine alternative ist die sogenannte erweiterte Backus-Naur-Form, kurz **EBNF**, siehe z.B. hier. Dazu später mehr.
2. Streng genommen können mit der angegebenen Form an Produktionsregeln nicht alle formalen Sprachen angegeben werden, dazu mehr im Exkurs zu Chomsky.

Das noch komplexere Beispiel von oben würde also vollständig so angegeben werden:

$$
\begin{aligned}
G = (&\{\text{Zahl}, \text{gerade}, \text{Ziffer}, \text{ungerade}\} \\
&\{0,1,2,3,4,5,6,7,8,9\}, \\
&(1)\ \text{Zahl} \rightarrow \text{gerade Ziffer ungerade} \\
&(2)\ \text{gerade} \rightarrow \text{'0'} \mid \text{'2'} \mid \text{'4'} \mid \text{'6'} \mid \text{'8'} \\
&(3)\ \text{ungerade} \rightarrow \text{''1} \mid \text{'3'} \mid \text{'5'} \mid \text{'7'} \mid \text{'9'} \\
&(4)\ \text{Ziffer} \rightarrow \text{gerade} \mid \text{ungerade} \\
&\\
&\text{Zahl})
\end{aligned}
$$

In aller Regel werden bei Aufgaben zu Grammatiken nur die Produktionsregeln verlangt, da die übrigen Mengen bzw. das Startsymbol aus diesen oder dem Kontext ersichtlich werden. Ist eine **vollständige** Beschreibung der Grammatik explizit gefragt, so muss dennoch die obige Form angegeben werden, siehe Aufgabe 3 auf der nächsten Seite.

Aufgaben

1. Definieren Sie die Produktionsregeln einer Grammatik, die die folgenden umgangssprachlichen Ausdrücke erzeugt. Geben Sie außerdem jeweils einen Ableitungsbaum für die Worte in Klammern an:
   - Alle zweistelligen natürlichen Zahlen.
   - Alle zweistelligen ganzen Zahlen.
   - Alle höchstens zweistelligen ganzen Zahlen.
   - Alle natürlichen Zahlen.
   - Alle Wörter mit 4 Buchstaben, die mit “e” beginnen und mit “a” enden

   > ***Hinweis***
   >
   > Als Alphabet sind nur die Kleinbuchstaben zugelassen, es müssen keine “sinn-vollen” Wörter im Sinne einer natürlichen Sprache sein.

Zur Lösung

2. Beschreiben Sie die Wörter der Sprache, die durch folgende Produktionsregeln erzeugt werden, umgangssprachlich:

   (1) Zahl → '5' Ziffer Ende | ε
   (2) Ziffer → '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'
   (3) Ende → '12' | '22'

Zur Lösung

3. Geben Sie eine **vollständige** Grammatik für die folgenden Sprachen an:

   > ***Hinweis***
   >
   > Sie können Flaci zur Überprüfung nutzen

   - Symmetrische a-b-Ketten nach folgendem Muster:

     aba, aabaa, aaabaaa, aaaabaaaa,$\cdots$
   - Geradzahlige Zahlen in Binärdarstellung:

     $0, 10, 100, 110, 1000, 1010, 1100, 1110, \cdots$
   - Natürliche Zahlen mit Tausenderpunkten:

     $1, 50, 375, 1.451, 100.105, 205.111.305 \cdots$

Zur Lösung

Lösungen:

Zweistellige natürliche Zahlen:

(1) NatZwei → ErsteZiffer Ziffer
(2) ErsteZiffer → '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
(3) Ziffer → '0' | ErsteZiffer

Zweistellige ganze Zahlen:

(1) GanzZwei → ErsteZiffer Ziffer | '−' ErsteZiffer Ziffer
(2) ErsteZiffer → '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
(3) Ziffer → '0' | ErsteZiffer

Höchstens zweistellige natürliche Zahlen:

(1) NatHöchstensZwei → ErsteZiffer | ErsteZiffer Ziffer
(2) ErsteZiffer → '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
(3) Ziffer → '0' | ErsteZiffer

oder auch:

(1) NatHöchstensZwei → ErsteZiffer Ziffer
(2) ErsteZiffer → '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
(3) Ziffer → '0' | ErsteZiffer | ε

Natürliche Zahlen:

1. Variante (Aufbau der Zeichenkette von vorne)

(1) Nat → ErsteZiffer | ErsteZiffer kette
(2) kette → Ziffer kette | Ziffer (Selbstreferenz!)
(3) ErsteZiffer → '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
(4) Ziffer → '0' | ErsteZiffer

2. Variante (Aufbau der Zeichenkette von hinten)

(1) Nat → ErsteZiffer | Nat Ziffer (Selbstreferenz!)
(2) ErsteZiffer → '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
(3) Ziffer → '0' | ErsteZiffer

Die sich selbst referenzierenden Regeln machen es möglich, dass unendlich viele Zeichen entstehen können. (Natürlich wäre es z.B. auch möglich, dass sich zwei Regeln gegenseitig referenzieren und so unendlich viele Zeichen ermöglichen)

Alle Wörter mit 4 Buchstaben, die mit "e" beginnen und mit "a" enden:

(1) Wort → 'e' Buchstabe Buchstabe 'a'
(2) Buchstabe → 'a' | 'b' | ··· | 'z'

Zurück zur Aufgabe

Beschreibungeder Sprache: Alle vierstelligen Zahlen, die mit 5 beginnen und auf 12 oder 22 enden **oder** keine Zahl, also das leere Wort! Zurück zur Aufgabe

Symmetrische a-b-Ketten:

$$G_1 = \{\{\text{Anzahl}\}, \{a, b\},$$
$$\{(1)\ \text{Anzahl} \rightarrow \text{'a' 'b' 'a'} \mid \text{'a' Zahl 'a'}\},$$
$$\text{Anzahl}\}$$

Wir müssen hier in einer Regel symmetrisch vor und hinter unserem Vervielfacher arbeiten, da ansonsten eine ungleiche Anzahl an a's möglich wäre! Also eine Regel der Form

$$\text{Wort} \rightarrow \text{X 'b' X}$$

(X bildet beliebig viele a's) ist nicht möglich

Geradzahlige Zahlen in Binärdarstellung

Zuerst muss man die Regelmäßigkeit erkennen:

Es handelt sich um alle Ketten von Nullen und Einsen, die auf eine 0 enden und mit einer 1 beginnen. Einzige Ausnahme hier ist die 0, da diese ebenfalls schon geradzahlig ist.

$$(1)\ \text{Zahl} \rightarrow \text{'0'} \mid \text{'1'}\ X\ \text{'0'}$$
$$(2)\ X \rightarrow XX \mid \text{'1'} \mid \text{'0'} \mid \varepsilon$$

Natürliche Zahlen mit Tausenderpunkten

Wir müssen sicherstellen, dass am Anfang keine Null steht und das alle drei Zahlen ein Punkt folgt.

$$(1)\ \text{Zahl} \rightarrow \text{Präfix} \mid \text{Zahl}$$
$$(2)\ \text{Präfix} \rightarrow E \mid EZ \mid EZZ$$
$$(3)\ \text{Dreier} \rightarrow \text{'.'}\ ZZZ$$
$$(4)\ E \rightarrow \text{'1'} \mid \text{'2'} \mid \text{'3'} \mid \text{'4'} \mid \text{'5'} \mid \text{'6'} \mid \text{'7'} \mid \text{'8'} \mid \text{'9'}$$
$$(5)\ Z \rightarrow E \mid \text{'0'}$$

Die erste Regel stellt dabei sicher, dass zu einer bestimmten Anzahl an Dreierpaketen an Zahlen (die dann mit Punkten versehen werden) genau ein Präfix hinzukommt, dass aus einer bis drei Zahlen bestehen kann (insbesondere auch notwendig, um ein- bis dreistellige Zahlen zu basteln!)

Zurück zur Aufgabe

2.2. Grammatik einer Programmiersprache

Ziel dieses kurzen Kapitels ist es, die Grammatik der einfachen Programmiersprache von "Robot Karol" aufzustellen. (Falls sich jemand an dieser Stelle fragt, warum wir keine andere Sprache nehmen, der sei z.B. auf die Python-Dokumentation hingewiesen :)

Es gab in Robot Karol (Ursprungsversion, verkürzter Befehlssatz) die folgenden Möglichkeiten:

1. **Befehle**: Schritt, LinksDrehen, RechtsDrehen, Hinlegen, Aufheben
2. **Bedingungen**: IstWand, NichtIstWand, IstZiegel, NichtIstZiegel
3. **Kontrollstrukturen**:
   - wiederhole n mal *wiederhole
   - wiederhole solange *wiederhole
   - wenn dann *wenn
   - wenn dann *wenn

Aufgabe

Entwickeln Sie die Produktionsregeln einer Grammatik für die obenstehend beschriebene Programmiersprache. Testen Sie Ihre Grammatik mit Flaci und lassen Sie sich den entsprechenden Ableitungsbaum anzeigen.

Die Lösung folgt direkt auf der nächsten Seite!

Die Produktionsregeln der Grammatik von Robot Karol

(1) Programm → *A* | *A* Programm

(2) *A* → Befehl | Wiederholung | Entscheidung

(3) Befehl → 'Schritt' | 'LinksDrehen' | 'Rechtsdrehen' | 'Hinlegen' | 'Aufheben'

(4) Entscheidung → 'wenn' Bedingung 'dann' Programm '*wenn' |
'wenn' Bedingung 'dann' Programm 'sonst' Programm '*wenn'

(5) Wiederholung → 'wiederhole' *N* 'mal' Programm '*wiederhole' |
'wiederhole solange Bedingung Programm '*wiederhole'

(6) Bedingung → 'IstWand' | 'NichtIstWand' | 'IstZiegel' | 'NichtIstZiegel'

(7) *N* → *E* | *E* Rest

(8) Rest → *Z* | *Z* Rest

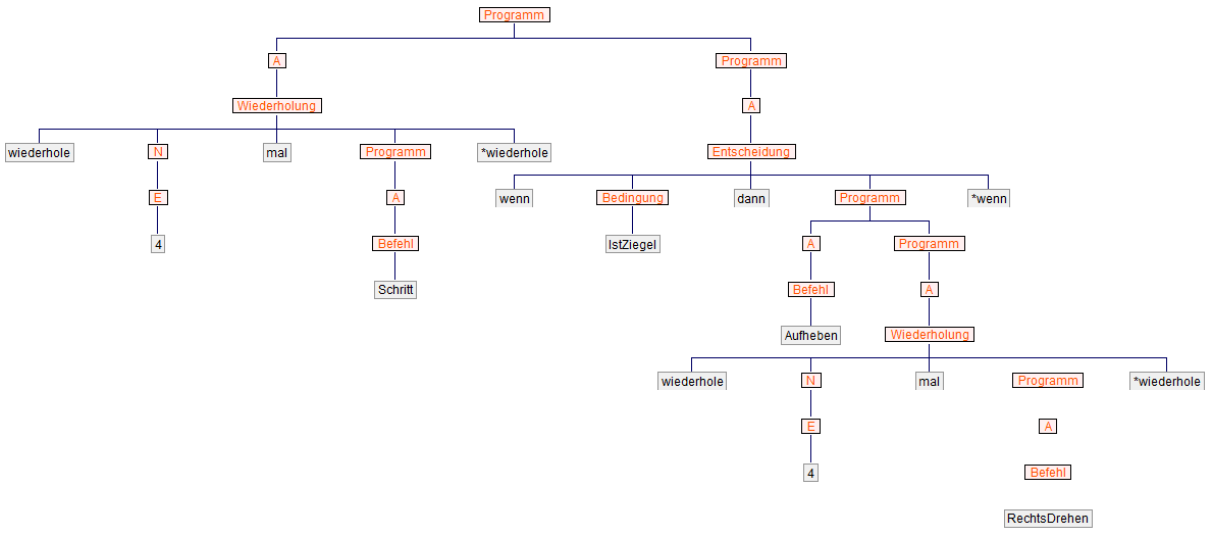
(9) *E* → '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

(10) *Z* → *E* | '0'

An dieser Stelle empfiehlt sich wirklich die Eingabe der Grammatik in Flaci und die Überprüfung eines klassischen Programms:

```
wiederhole 4 mal
    Schritt
*wiederhole
wenn IstZiegel
    dann Aufheben
    wiederhole 4 mal
        RechtsDrehen
    *wiederhole
*wenn
```

Es ergibt sich der folgende Ableitungsbaum (reinzoomen nötig!):



2.3. Grammatik von E-Mail-Adressen

Für Emails gelten bei uns die folgenden (vereinfachten) Regeln:

1. Eine **Email-Adresse** besteht aus einer Benutzerkennung, dem @-Zeichen und dem Namen einer Domäne.
2. Für die **Benutzerkennung** dürfen Kleinbuchstaben und Ziffern verwendet werden. Sie darf beliebig, mindestens jedoch ein zeichen lang sein.
3. Die **Domäne** setzt sich zusammen aus einer oder mehrerer Unterdomänen, gefolgt von einer Hauptdomäne (Top-Level-Domain). Die Bestandteile der Domäne sind jeweils durch einen Punkt getrennt.
4. Die **Top-Level-Domain** darf nur aus Kleinbuchstaben zusammengesetzt sein und ist entweder zwei oder drei Zeichen lang.
5. Jede **Unterdomäne** darf Kleinbuchstaben und Zahlen enthalten. Sie darf beliebig, mindestens jedoch zwei Zeichen lang sein.

Aufgabe

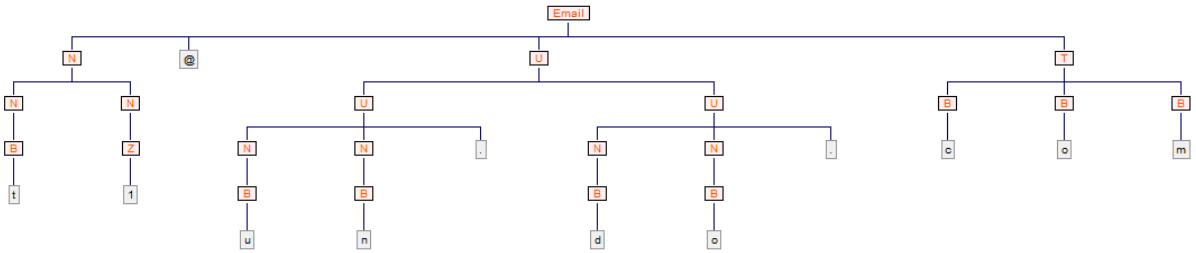
Entwickeln Sie die Produktionsregeln einer Grammatik für die oben beschriebenen Regeln für Email-Adressen. Testen Sie Ihre Grammatik mit Flaci und lassen Sie sich den entsprechenden Ableitungsbaum anzeigen.

Die Lösung findet sich auf der nächsten Seite!

In der folgenden Lösung steht N für die Nutzerkennung (auch verwendet als beliebig lange Zeichenkette, aber mindestens ein Zeichen lang), U für eine Unterdomäne und T für eine Top-Level-Domain:

$$
\begin{aligned}
&(1)\ \text{Email} \to N\ \text{'@'}\ UT\\
&(2)\ N \to B \mid Z \mid NN\\
&(3)\ B \to \text{'a'} \mid \cdots \mid \text{'z'}\\
&(4)\ Z \to \text{'0'} \mid \cdots \mid \text{'9'}\\
&(5)\ U \to NN\ \text{'.'} \mid UU\\
&(6)\ T \to BB \mid BBB
\end{aligned}
$$

Damit Flaci nicht allzuviel zu tun hat verwenden wir zum Testen nur die Emailadresse “t1@un.do.-com”. Es ergibt sich der folgende Ableitungsbaum (wieder reinzoomen nötig!):



2.4. EBNF

Wie bereits in einem Hinweis erwähnt gibt es eine weitere übliche Notationsform, die **erweiterte Backus-Naur-Form**. Es gelten die folgenden Regeln (Auszug der für uns Wichtigsten):

1. Alle Regeln haben die Form <Nichtterminal>= · · · ;
2. Aneinanderreihungen mit , (kann weggelassen werden)
3. Oder-Symbol |
4. Terminalsymbole werden in Anführungszeichen oder Apostrophe gesetzt.
5. **Optionen** werden in eckige Klammern gesetzt [· · ·]
6. **Gruppierung von Elementen**: (· · ·)
7. **Wiederholung** (auch Null mal): {· · ·}
8. **Mehrfachausführung**: 3*X
9. **Ausschluss**: · · · −X

Die Apostrophe bei Zeichen sind insbesondere deswegen wichtig, um Zeichen der EBNF-Notation von der Verwendung innerhalb der Zeichenkette zu unterscheiden, so hat das "="-Zeichen untenstehend zwei verschiedene Bedeutungen - der Beginn der Regel und die Verwendung als tatsächliches Zeichen

Beispiel = Anfang '+' Mitte '=' Ende ;

> ***Hinweis***
>
> Nimmt man die EBNF-Regeln ernst, müsste z.B. nach Anfang ein , stehen. Diese Kommata dürfen aber weggelassen werden (siehe oben)

Im Folgenden finden sich einige Beispiele, um die EBNF-Regeln zu veranschaulichen:

Regel	**Möglicher Output**
Anfang = ['A' | 'B'] 'C';	**AC** oder **BC** oder **C**
Mitte = 'A' ('X' | 'Y' | Var);	**AX** oder **AY** oder **A**Var
Ende= 3* Baum 'Z';	**Baum Baum Baum Z**
Var = 'A'{'B' 'C' } 'D';	**AD** oder **ABCD** oder **ABCBCD** oder · · ·
Text = 'X' {Var} − Var ;	**X** oder **X Var Var** oder **X Var Var Var**

Aufgaben

Schreiben Sie die Produktionsregeln in EBNF- Form für die folgenden umgangssprachlich beschriebenen Zeichenketten (jeweils ggf. mit den bisher gemachten Vereinfachungen)

1. Natürliche Zahlen
2. alle Vielfachen von 5 (also 5,10,15,· · ·)
3. Ganze Zahlen
4. Dezimalzahlen
5. geradzahlige Binärzahlen
6. Die Sprache, die nur die Wörter **x**, **xz** und **xyz** enthält (ohne "Oder"-Zeichen)
7. Natürliche Zahlen mit Tausenderpunkten
8. Emails
9. "Korrekte" Summen und Differenzen, also z.B.: $(15 + (-3)) - 500 + ((33 + 11) - (-5))$

Die Lösungen finden sich ab der nächsten Seite.

Natürliche Zahlen:

(1) Nat = Z{Z | '0'};
(2) Z = '1'|$\cdots$ |'9';

Alle Vielfachen von 5:

(1) Zahl = '5' | Z {Z | '0'} ('0'|'5');
(2) Z = '1'|$\cdots$ |'9';

Ganze Zahlen:

(1) Ganz = '0' | ['-'] Z {Z | '0'};
(2) Z = '1'|$\cdots$ |'9';

Dezimalzahlen:

(1) Dezi = ['-'] ('0'| Z{Z|'0'})','{Z|'0'}Z;
(2) Z = '1'|$\cdots$ |'9';

Geradzahlige Binärzahlen:

(1) Zahl = '0'|'1' {'0'|'1'} '0';

oder

(1) Zahl = ['1' {'0'|'1'}] '0';

Die Sprache, die nur die Wörter x, xz und xyz enthält

(1) Wort = 'x'[['y']'z'];

Natürliche Zahlen mit Tausenderpunkten

(1) Zahl = E[Z][Z]{'.'3*Z};
(2) E = '1'|$\cdots$ |'9';
(3) Z = Z |'0';

Email-Adressen:

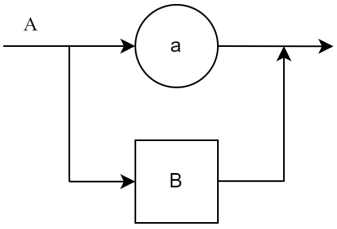
(1) Adresse = Benutzer '@' Domäne;
(2) Benutzer = (Z | B) {Z | B};
(3) Domäne = Unterdomäne {Unterdomäne} TLD;
(4) TLD = BB[B];
(5) Unterdomäne = 2*(Z | B) {Z | B}'.';
(6) Z '0'|$\cdots$ |'9';
(7) B 'a'|$\cdots$ | 'z';

Terme

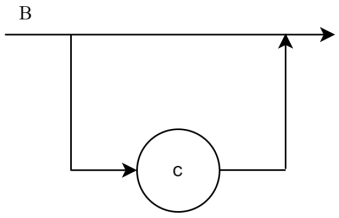
(1) Term = Summand ('+'|'-') Summand{('+'|'-') Summand};
(2) Summand = Zahl | '(' Term ')';
(3) Zahl = Nat | '0' | '(' '-' Nat')';
(4) Nat = Ziffer – '0' {Ziffer};
(5) Ziffer = '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9';

2.5. Syntaxdiagramme für EBNF

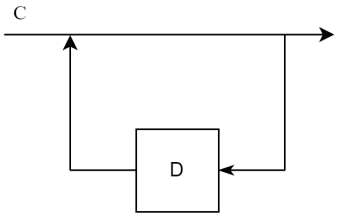
Statt eine Grammatik anzugeben, kann die EBNF-Notation auch in Diagrammform in sogenannten **Syntaxdiagrammen** dargestellt werden. Dabei gibt es für jede Produktionsregel ein eigenes Syntaxdiagramm.



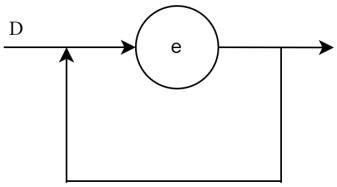
Auswahl: A = 'a' | B;



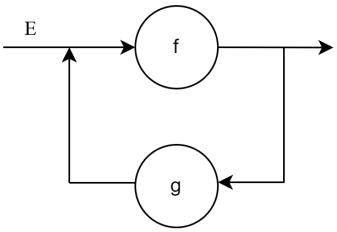
Option: B = ['c'];



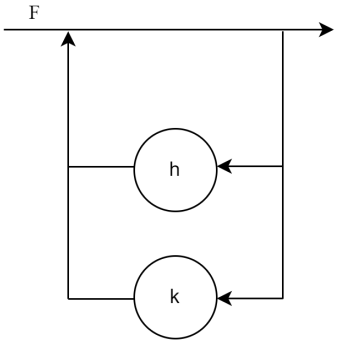
Wiederholung: C = { D };



Kombi 1: D = 'e' { 'e' };



Kombi 2: E = 'f' { 'g' 'f" };



Kombi 3: F = { 'h' | 'k' }

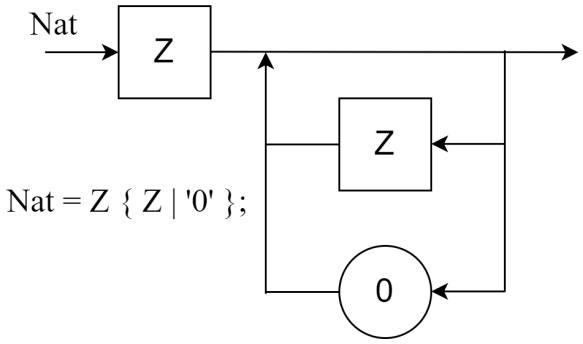
Aufgabe

Zeichnen Sie für die folgenden Regeln jeweils Syntaxdiagramme:

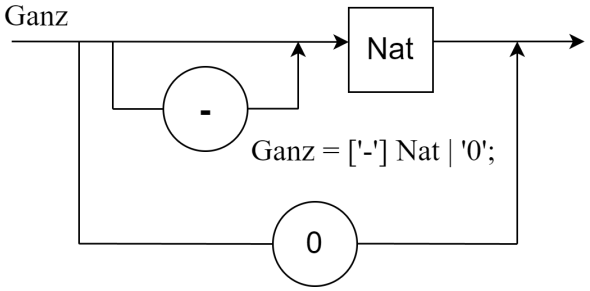
1. Nat = Z {Z | '0'};
2. Ganz = ['-'] Nat | '0';
3. Dez = ['-'] ('0' | Nat) ',' {Z|'0'} Z ;
4. Wort = 'X' {'x' | B A} ;
5. Wort2 = [B [C] ('x' | 'y' | 'z')] ;
6. Zahl = '5' | Z { Z | '0' } ('0' | '5');
7. ZahlT = E [Z] [Z] { '.' 3* Z };

Die Lösungen finden sich wieder ab der nächsten Seite:

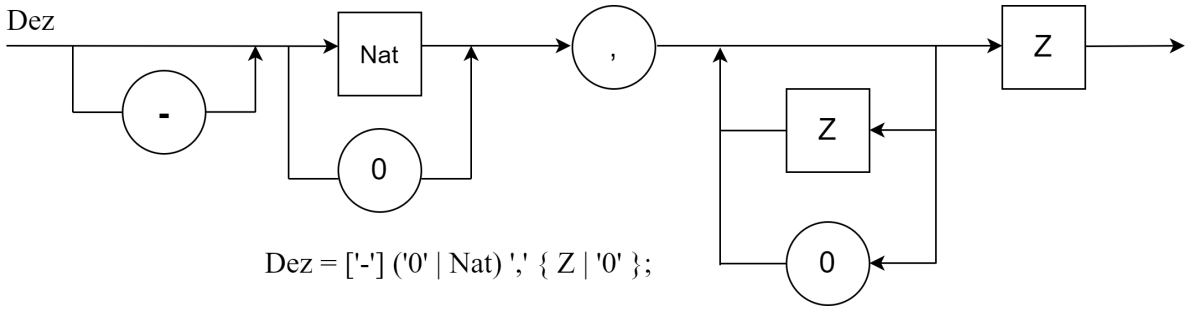
Natürliche Zahlen:



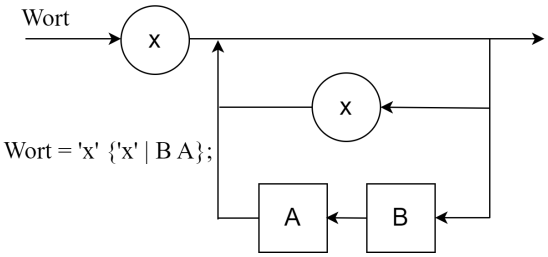
Ganze zahlen



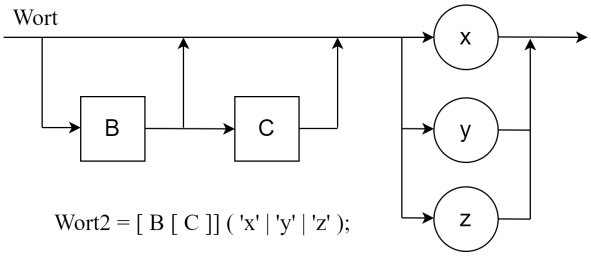
Dezimalzahlen:



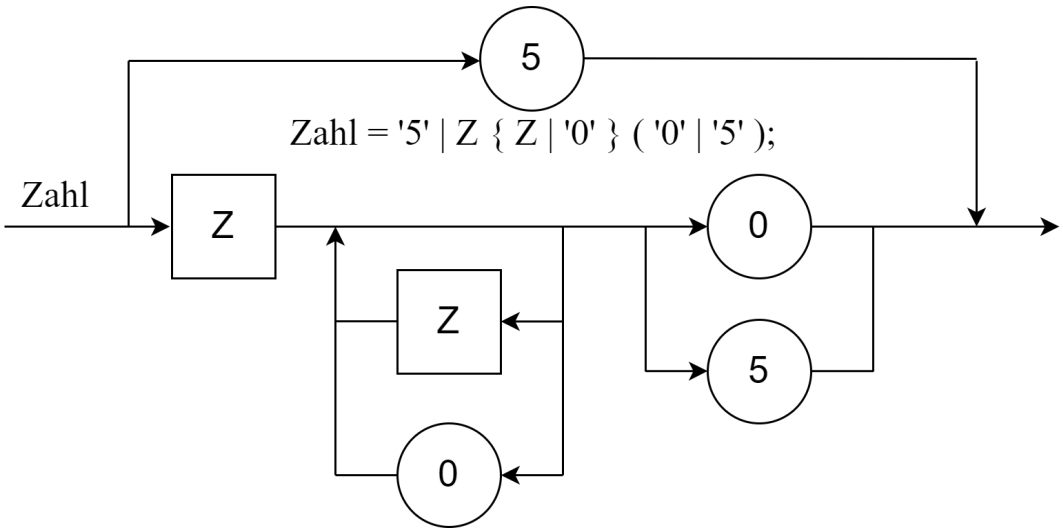
Wort:



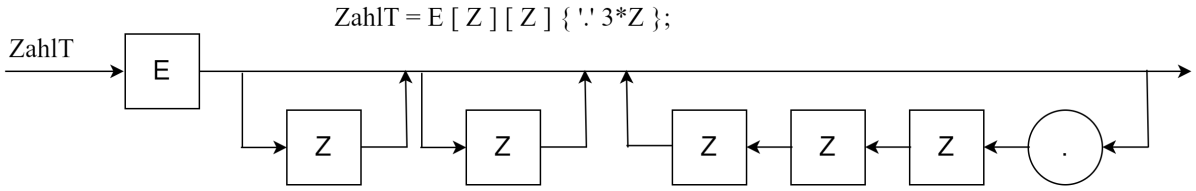
Wort 2



Vielfache von 5:

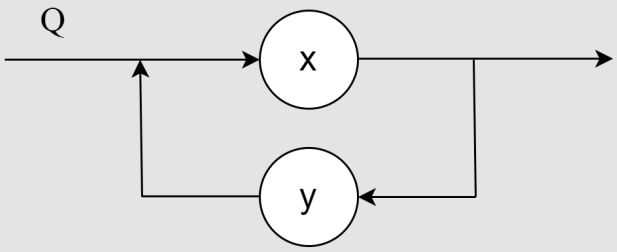


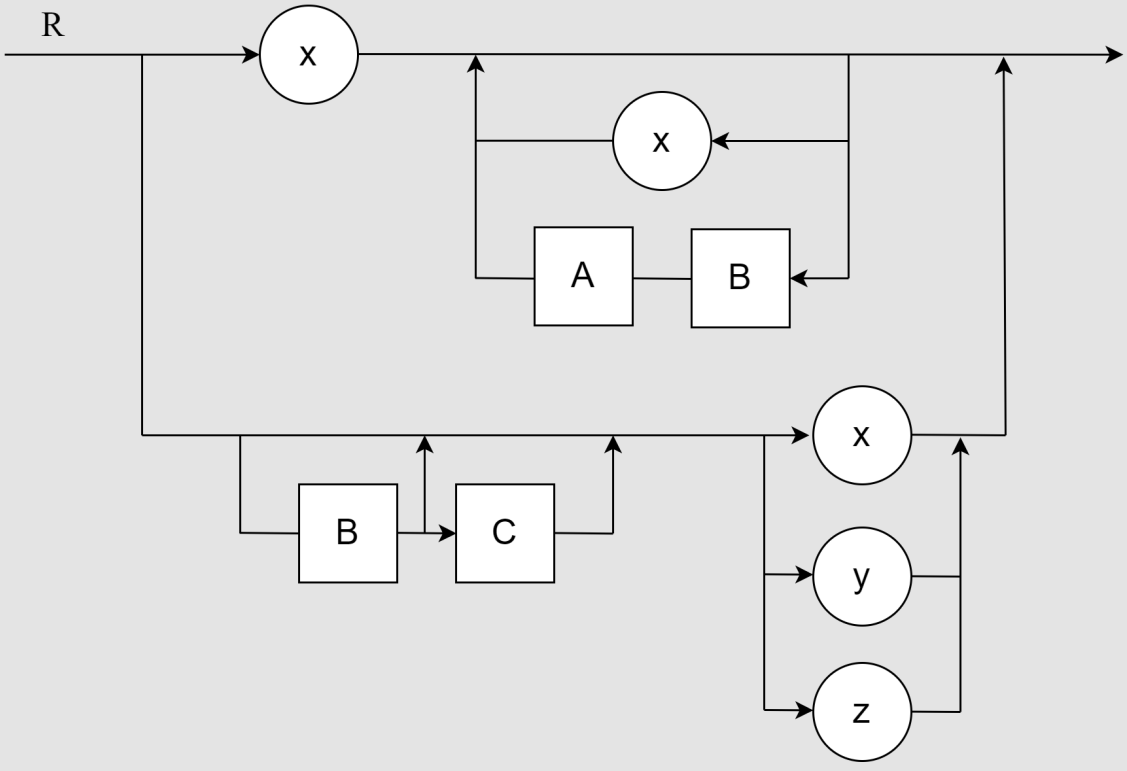
Natürliche Zahlen mit Tausenderpunkten:



Aufgabe

Geben Sie die Produktionsregeln in EBNF-Form an, die zu den folgenden Syntaxdiagrammen gehören:



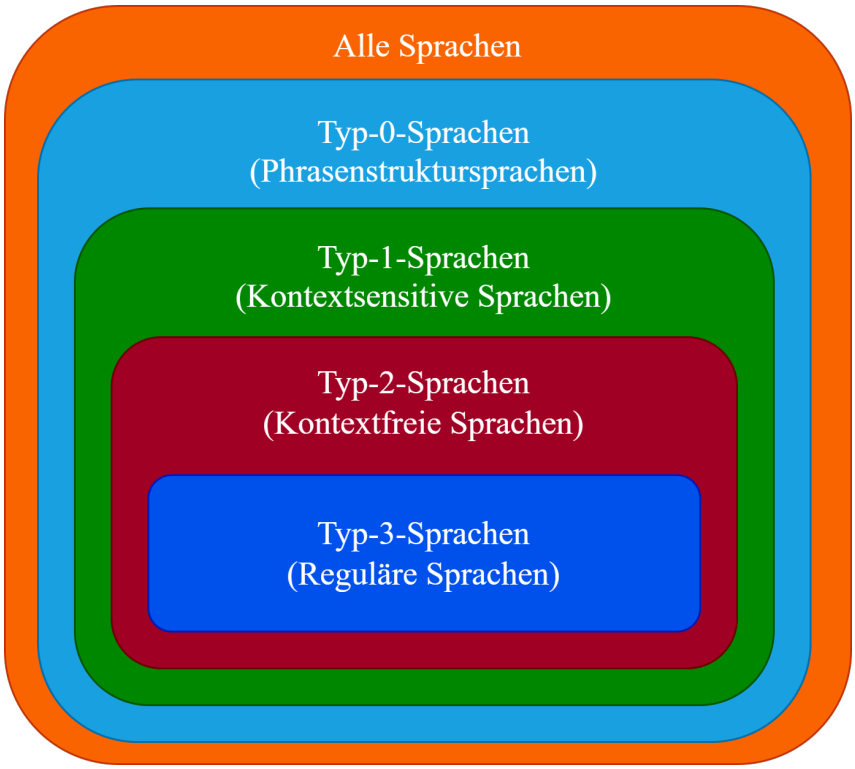


2.6. Exkurs: Sprachklassen nach Chomsky

Bisher haben wir nur an der Oberfläche der Sprachlehre gekratzt. Die bereits eingeführten künstlichen Sprachen können sich noch in **Klassen** einteilen lassen. Die Produktionsregeln der einzelnen Klassen unterliegen dabei verschiedenen Einschränkungen, d.h. manche Sprachen sind "**spezieller**".

Dabei gibt es einen direkten Zusammenhang zwischen den Eigenschaften der Produktionsregeln der **Grammatik** und den Eigenschaften der von ihr erzeugten Sprache **L(G)**.

Der amerikanische Sprachwissenschaftler Noam Chomsky hat ein Klassifikationssystem entwickelt, das vier Arten von formalen Grammatiken und damit von formalen Sprachen definiert ([1] S. 169).



Um die Unterschiede der Sprachen besser beleuchten zu können, ist eine allgemeinere Definition einer Grammatik notwendig, als die, die wir bisher verwendet haben.

Definition

Eine **Grammatik** einer formalen Sprache ist ein Viertupel (V, Σ, P, S) und besteht aus

1. der endlichen *Variablenmenge V (Nonterminale)*,
2. dem endlichen *Terminalalphabet* Σ mit $V \cap \Sigma = \emptyset$
3. der endlichen Menge P von *produktionen (Regeln)* und
4. der *Startvariablen* S mit $S \in V$.

Jede Produktion aus P hat die Form $l \to r$ mit $l \in (V \cup \Sigma)^+$ und $r \in (V \cup \Sigma)^*$ ([1], S. 164).

Der Wesentliche Unterschied liegt - neben der noch mathematischeren Notation - in den erlaubten Produktionsregeln. Die linke Seite einer Regel kann im Allgemeinen nicht nur ein Nichtterminalsymbol enthalten, sondern mindestens ein Terminal oder Nichtterminalsymbol. Das erweitert unsere Möglichkeiten beträchtlich!

Damit lässt sich die Klassifikation mit Leben füllen ([1], S. 169):

1. **Phrasenstrukturgrammatiken (Typ-0-Grammatiken)**
   Jede Grammatik, die die obere Definition erfüllt ist automatisch eine Typ-0-Grammatik. Es gibt insbesondere keine weiteren Einschränkungen für die Produktionsregeln.
2. **Kontextsensitive Grammatiken (Typ-1-Grammatiken)**
   Bei kontextsensitiven Grammatiken muss $|r| \geq |l|$ gelten, d.h. die rechte Seite muss mindestens so lang sein, wie die linke Seite der Produktionsregel (einzige Ausnahme: $S \to \varepsilon$, dann darf S aber in keiner rechten Seite vorkommen!)
3. **Kontextfreie Grammatiken (Typ-2-Grammatiken)**
   Zusätzlich zu der vorherigen Einschränkung darf jetzt auf der linken Seite nur noch ein einziges Nonterminal stehen (d.h. unsere Grammatikdefinition entspricht kontextfreien Sprachen).
4. **Reguläre Grammatiken (Typ-3-Grammatiken)**
   Die speziellsten Grammatiken, sie sind kontextfrei und besitzen zusätzlich die Eigenschaft, dass die rechte Seite einer Regel entweder ε ist oder ein Terminalsymbol gefolgt von einem Nichtterminalsymbol.

Beispiele ([1], S. 169ff.)

1. $L_3 := \{(ab)^n \mid n \in \mathbb{N}^+\}$ ist eine **reguläre Sprache**.

   > ***Hinweis***
   >
   > Der Exponent steht hier für eine Wiederholung, d.h. hier betrachten wir Wörter wie $ab, abab, \cdots$

   Mögliche Produktionsregeln sind:

   $$(1)\ S \to aB$$
   $$(2)\ B \to b$$
   $$(3)\ B \to bC$$
   $$(4)\ C \to aB$$

   Ableitung von $abab$:

   $$S \overset{(1)}{\to} aB \overset{(3)}{\to} abC \overset{(4)}{\to} abaB \overset{(2)}{\to} abab$$

   Das Wort wird also von links nach rechts "aufgebaut". Man spricht deswegen auch von einer "**rechts-linearen Grammatik**".
2. $L_2 := \{a^n b^n \mid n \in \mathbb{N}^+\}$ ist eine **kontextfreie Sprache**, aber nicht regulär. Die Sprache ist nicht mehr regulär, da wir in irgendeiner Form "zählen" müssten, wie viele a's es gibt, bevor die b's beginnen. Das ist mit den Regeln für reguläre Grammatiken nicht möglich.
   Eine mögliche Produktionsregel ist:

   $$(1)\ S \to aSb \mid ab$$
3. $L_1 := \{a^n b^n c^n \mid n \in \mathbb{N}^+\}$ ist eine **kontextsensitive Sprache**, aber nicht kontextfrei. Der Unterschied liegt darin, dass wir auch Terminalsymbole auf die linke Seite der Produktionsregeln packen können. Dadurch können wir die Regel abhängig von der **Umgebung** (also dem Kontext) des Nichtterminalsymbols machen, die Grammatik wird etwas länglich:

(1) $S \to SABC \mid abc$
(2) $CA \to AC$
(3) $CB \to BC$
(4) $BA \to AB$
(5) $cA \to Ac$
(6) $cB \to Bc$
(7) $bA \to Ab$
(8) $aA \to aa$
(9) $bB \to bb$
(10) $cC \to cc$

Eine Ableitung des Wortes *aaabbbccc* ist dem geneigten Lesy zur Übung überlassen :).

4. $L_0 := \{\omega \mid \omega \text{ codiert eine terminierende Turing-Maschine}\}$ ist eine **Typ-0-Sprache**, aber keine **kontextsensitive Sprache**. Eine explizite Grammatik für diese Sprache anzugeben ist schwer, deswegen wird an dieser Stelle darauf verzichtet.

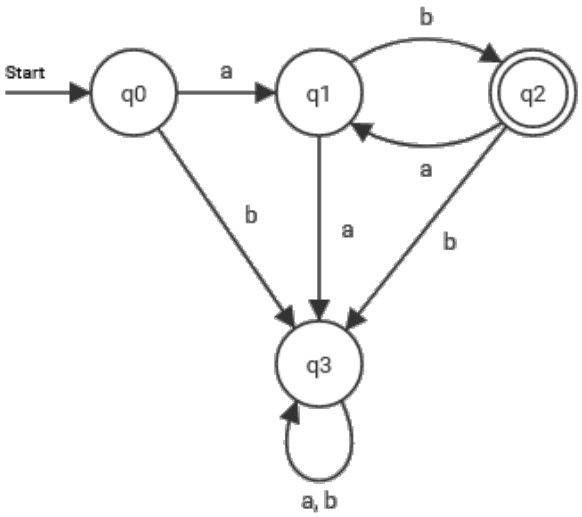
3. Endliche Automaten

3.1. Deterministische endliche Automaten

Die Überprüfung, ob ein bestimmtes Wort in einer Sprache liegt haben wir bisher über die direkte Anwendung der Regeln selbst händisch ausprobiert, bzw. anhand der Struktur der Wörter direkt erkannt.

Dieses Vorgehen ist im Allgemeinen - insbesondere bei komplexen Grammatiken bzw. Sprachen - natürlich fehleranfällig und sehr zeitintensiv. Es braucht also eine Möglichkeit das Ganze zu automatisieren. Das Werkzeug der Wahl ist dabei ein sogenannter **deterministischer endlicher Automat (DEA)**. Er prüft, ob eine eingegebene Zeichenkette ein Wort der entsprechenden Sprache ist.

Ein DEA wird häufig über ein sogenanntes **Zustandsdiagramm** visualisiert:



Dieser Automat akzeptiert alle Wörter der Sprache $L_3 := \{(ab)^n \mid n \in \mathbb{N}^+\}$, der Zustand $\boldsymbol{q_0}$ ist der **Startzustand**, der Zustand $\boldsymbol{q_2}$ ein Endzustand. Der Zustand $\boldsymbol{q_3}$ wird häufig auch **Fallenzsutand** genannt, da der Automat von dort nicht wieder wegkommt, dazu später mehr.

Formal definiert sich ein Automat wie folgt:

Definition

Ein deterministischer endlicher Automat lässt wird durch die folgenden Angaben festgelegt:

1. **Q**: eine endliche Menge von **Zuständen**
2. **Σ**: eine endliche Menge von Eingabesymbolen (unser **Alphabet**!)
3. $\boldsymbol{\delta}$ eine **Zustandsübergangsfunktion**, die angibt, in welchen Folgezustand $q' \in Q$ der Automat übergeht, wenn im aktuellen zustand $q \in Q$ das Symbol x aus Σ eingelesen wird, also $\delta(q, x) = q'$.
4. $\boldsymbol{q_0}$: ein **Anfangszustand**.
5. **F**: eine Menge von **Endzuständen** (final states)

Kurz: DEA = (Q, Σ, δ, q_0, F)

Beispiel $L_3 := \{(ab)^n \mid n \in \mathbb{N}^+\}$: $\mathbf{Q} = \{q_0; q_1; q_2; q_3\}$ $\mathbf{\Sigma} = \{a; b\}$ $\boldsymbol{\delta}$: siehe unten $\boldsymbol{q_0}$ $\mathbf{F} = \{q_2\}$

Die Zustandsübergangsfunktion wird entweder als Diagramm (wie oben) dargestellt, oder in einer Tabelle, in der für jedes Paar aus Zustand q und Alphabetzeichen x der neue Zustand steht.

$\boldsymbol{\delta}$	$\boldsymbol{a}$	$\boldsymbol{b}$
$\boldsymbol{q_0}$	q_1	q_3
$\boldsymbol{q_1}$	q_3	q_2
$\boldsymbol{q_2}$	q_1	q_3
$\boldsymbol{q_3}$	q_3	q_3

Überprüfung einer Zeichenkette:

Möchte man nun überprüfen, ob eine bestimmte Zeichenkette ein Wort der Sprache ist, so beginnt man im Startzustand. Danach werden alle Symbole der Zeichenkette nacheinander eingelesen. Nach jedem Symbol x geht der Automat gemäß der Übergangstabelle vom aktuellen q in den neuen Zustand q' über $\delta(q, x) = q'$.

Für das Wort $abab$ ergibt sich die folgende Reihenfolge an Zuständen:

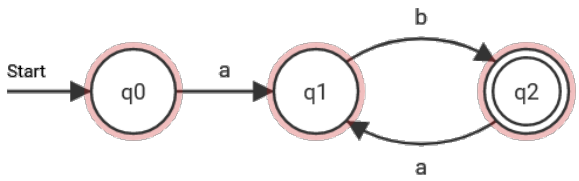
$$q_0 \overset{a}{\rightarrow} q_1 \overset{b}{\rightarrow} q_2 \overset{a}{\rightarrow} q_1 \overset{b}{\rightarrow} q_2$$

> ***Hinweis***
>
> Mit https://flaci.com kann diese Überprüfung auch automatisiert erfolgen!

Nach Abarbeitung der Zeichenkette gibt es drei Möglichkeiten:

1. Der Automat stoppt in einem Endzustand - daraus folgt, dass die Zeichenkette zur Sprachegehört.
2. Der Automat stoppt in einem Nicht-Endzustand - daraus folgt, dass die Zeichenkette nicht zur Sprache gehört.
3. Der Automat stoppt, da die Zeichenkette nicht komplett abgearbeitet werden kann, da es zu einer Kombination aus Zustand q und Symbol x keinen Folgezustand gibt - daraus folgt, dass die Zeichenkette nicht zur Sprache gehört.

Der dritte Fall kann nur auftreten, wenn der Automat **nicht vollständig** ist. Den obigen Automaten könnte man alternativ auch so darstellen:



$\boldsymbol{\delta}$	$\boldsymbol{a}$	$\boldsymbol{b}$
$\boldsymbol{q_0}$	q_1	
$\boldsymbol{q_1}$		q_2
$\boldsymbol{q_2}$	q_1	

In der ersten Variante diente der Zustand q_3 nur dazu, die "losen Enden" aufzusammeln, d.h. alle Eingaben, die zu einer garantiert ungültigen Kombination führen (wenn beispielsweise zwi a direkt nacheinander kommen). Wird so ein **Fallenzustand** benutzt und damit jede mögliche Kombination (q, x) kodiert, so spricht man von einem **vollständigen** Automaten.

Jeder nicht vollständige Automat kann also leicht zu einem vollständigen Automaten gemacht werden. Der Übersichtlichkeit halber ist aber häufig eine nicht vollständige Angabe zu bevorzugen.

> ***Hinweis***
>
> Ein DEA, der die Worte einer bestimmten Sprache akzeptiert, kann nur gefunden werden, wenn es sich um eine reguläre Sprache handelt!

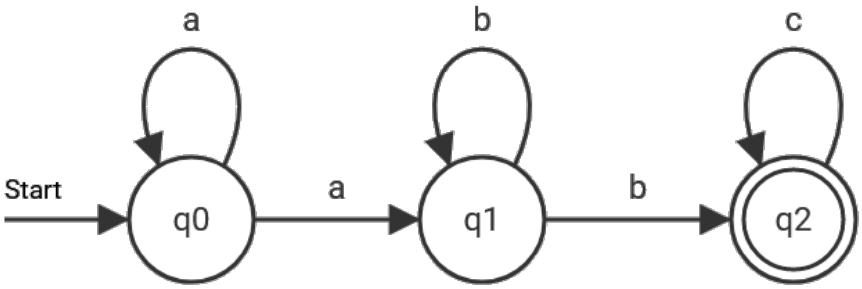
3.2. Exkurs: Nichtdeterministische endliche Automaten

Der obige Automatentyp war unter Anderen dadurch gekennzeichnet, dass es für jeden Zustand und eine Eingabe immer genau einen Folgezustand gibt. Das muss nicht notwendigerweise so sein. Lässt man mehrere ausgehende Kanten für eine Eingabe aus einem Zustand zu, so spricht man von einem **nichtdeterministischen endlichen Automaten** (NEA), da es nicht mehr exakt festgelegt (determiniert) ist, welcher Zustand als nächstes kommt. In der Übergangstabelle kann es also auch **Mehrfacheinträge** geben.

Ein Wort wird dann akzeptiert, wenn es **einen möglichen Weg** gibt, der zu einem Endzustand führt.

Überraschenderweise ist dieser Ansatz nicht "**mächtiger**" als die deterministischen Automaten, d.h. auch mit NEA's können nur reguläre Sprachen erkannt werden.

Beispiel: die Sprache, die mindestens ein a, dann mindestens ein b und am Ende beliebig viele c's enthält, kann mit folgenden Automaten beschrieben werden.



3.3. Übungen und Beispiele

Aufgabe

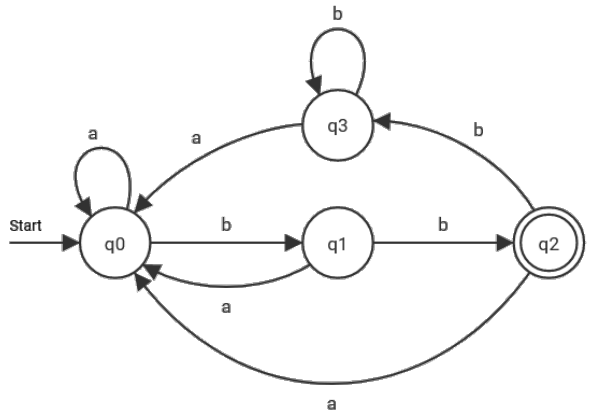
Geben Sie jeweils einen endlichen (nicht notwendigerweise vollständigen) Automaten an (Zustandsübergangsdiagramm), der die Wörter der Sprache erkennt:

1. Natürliche Zahlen
2. Ganze Zahlen
3. Natürliche, durch 10 teilbare Zahlen (als NEA und als DEA!)
4. a-b-Ketten, die
   - mit mindestens zwei “b” enden.
   - mit genau zwei “b” enden.
   - mindestens einmal “aaa” oder “bbb” enthalten.

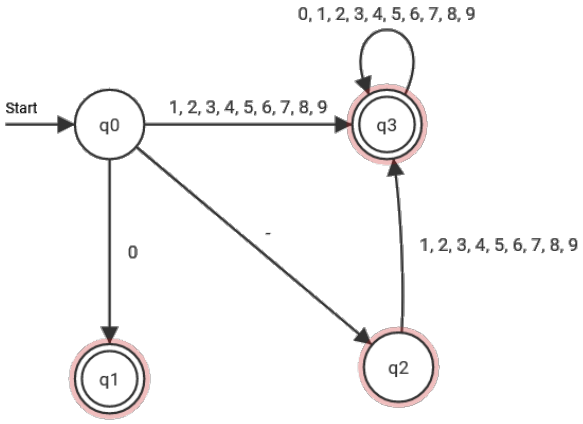
   > ***Hinweis***
   >
   > Eine a-b-Kette kann auch 0 a’s oder 0 b’s enthalten!
5. Natürliche Zahlen mit Tausenderpunkten

Wie immer beginnen die Lösungen auf der nächsten Seite

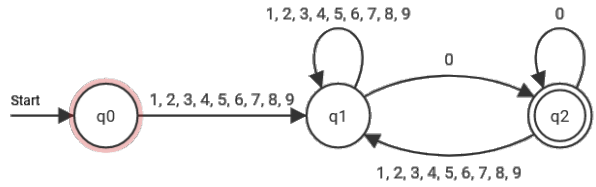
4b)



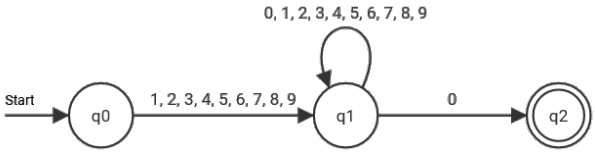
2)



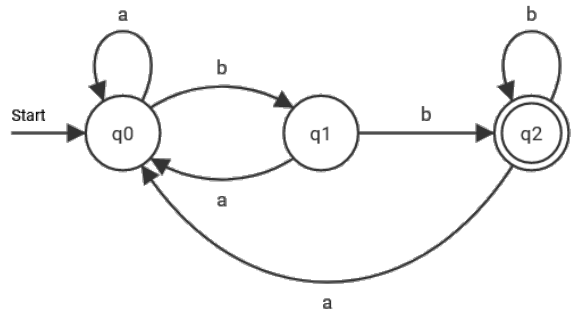
3-DEA



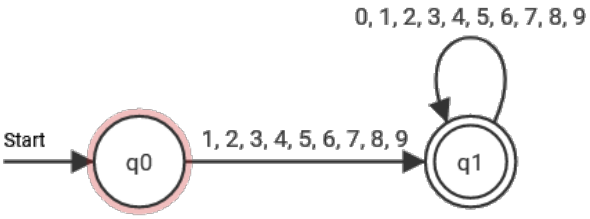
3-NEA



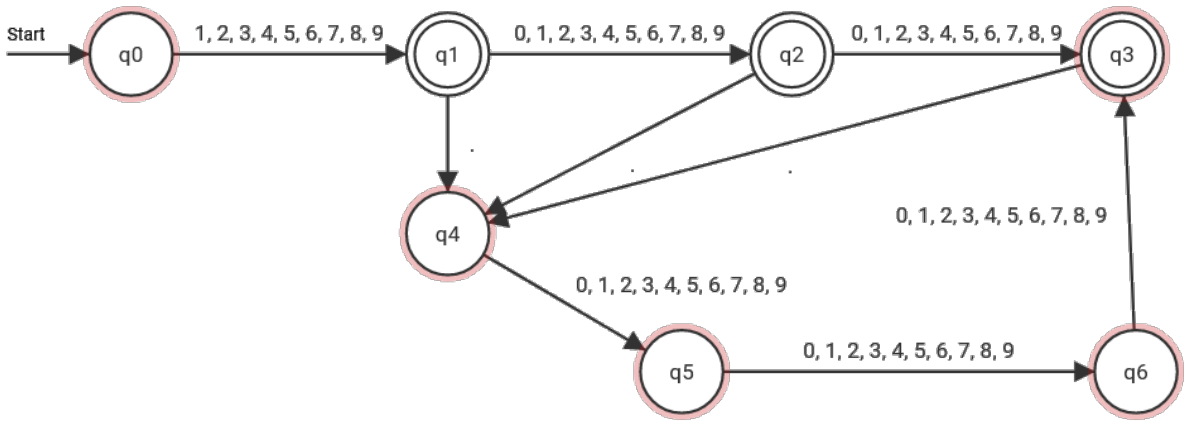
4a)



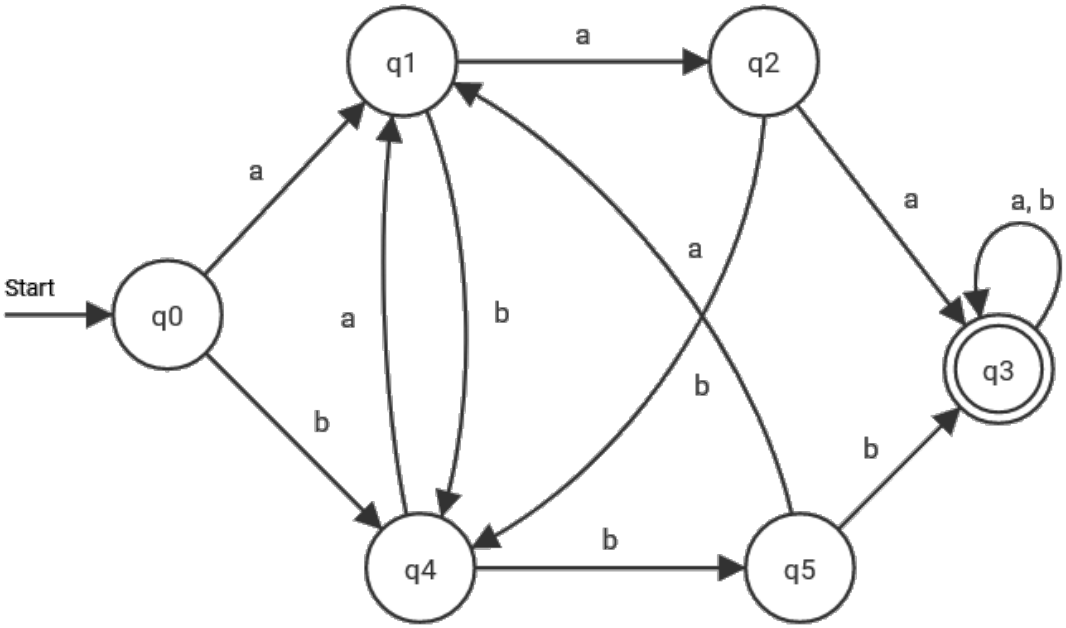
1)



5)

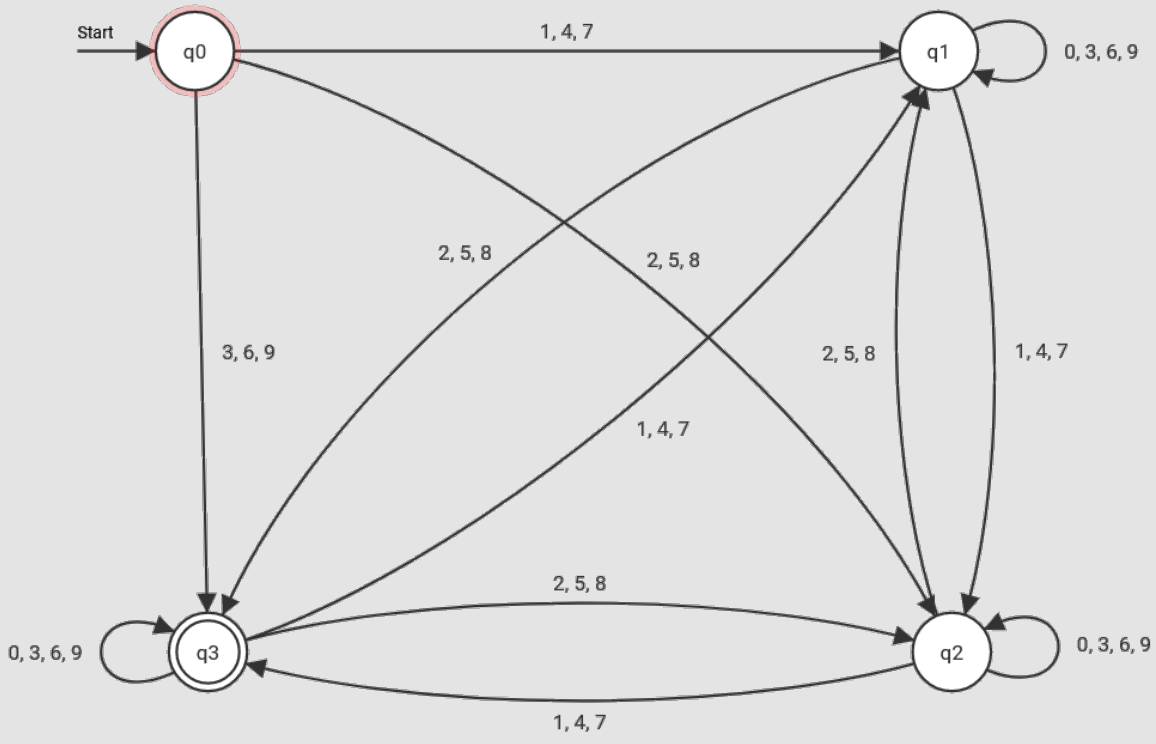


4c)



Aufgabe

Beschreiben Sie umgangssprachlich, welche Zeichenketten der folgende Automat erkennt:



Lösung: Der Automat akzeptiert alle durch drei teilbaren natürlichen Zahlen!

Aufgabe

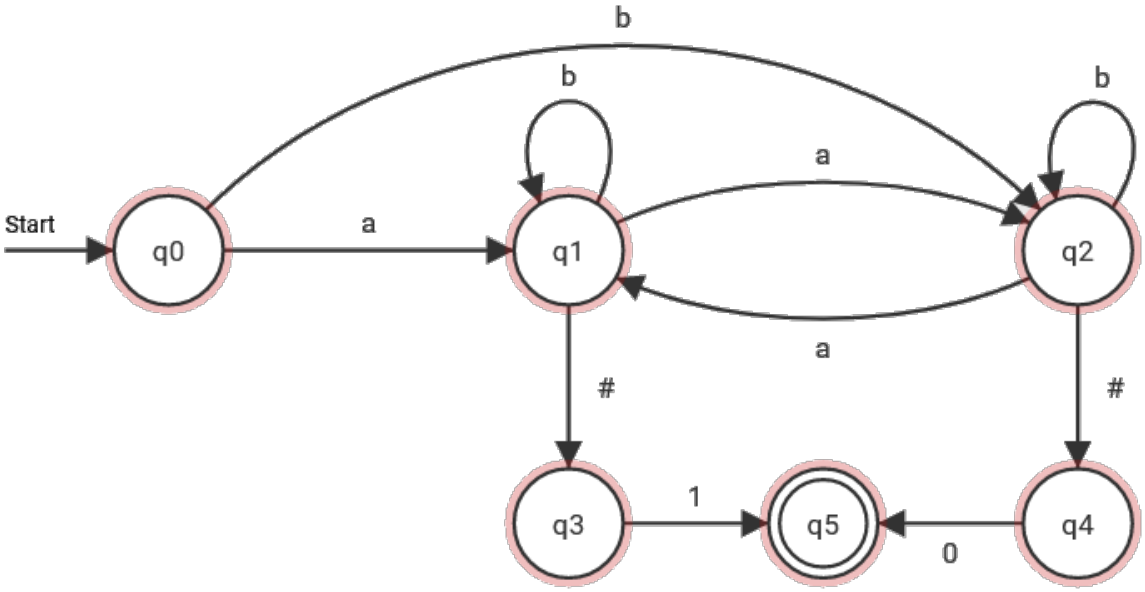
Geben Sie jeweils einen (nicht notwendigerweise vollständigen) endlichen Automaten an, der die folgende Form hat:

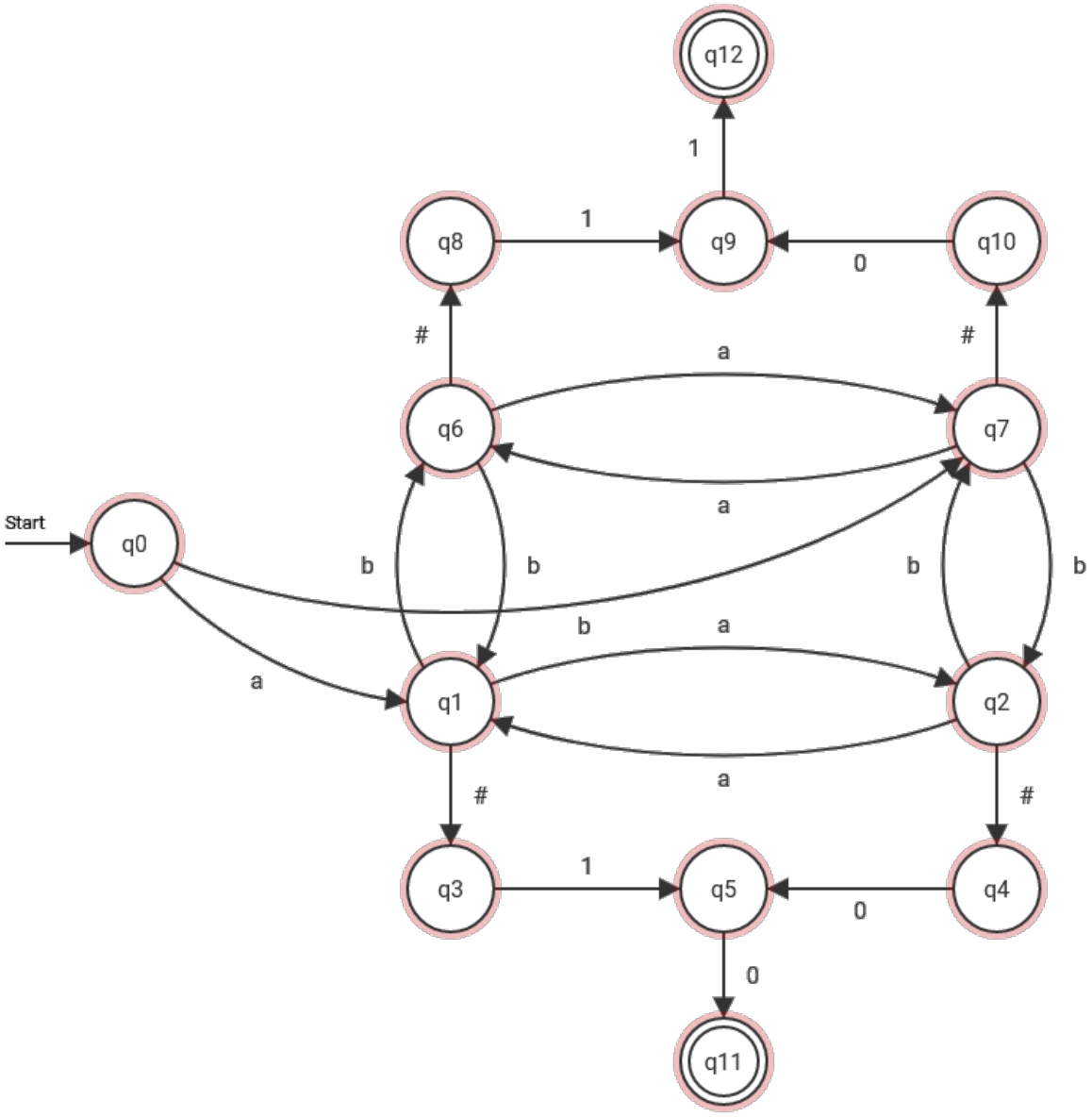
<kette>#<prüf>

wobei:

1. < kette> eine beliebig lange, aber nicht leere a-b-Kette ist und <prüf> eine 1, falls zuvor ungeradzahlig viele a's enthalten waren. Andernfalls eine 0.
2. < kette> eine beliebig lange, aber nicht leere a-b-Kette ist und <prüf> das Format $00, 01, 10$ oder 11 hat, wobei das zweite Zeichen von der Anzahl der b's abhängt (wiederum 1, falls ungeradzahlig).

Die Lösung wie immer auf der nächsten Seite!


Start
q0
q1
q2
q3
q4
q5
a
b
#
1
0


Start
q0
q1
q2
q3
q4
q5
q6
q7
q8
q9
q10
q11
q12
a
b
#
1
0

Abschlussaufgabe

Aufgabe

Zeichnen Sie das Zustandsdiagramm eines Automaten, der Datumsangaben der Form TT.MM.JJJJ erkennt, mit folgenden Einschränkungen:

1. Es sollen nur die **Jahreszahlen voon 1900 bis 2099** gültig sein und zudem sollen **alle Monate 31 Tage** haben.
2. Zusätzlich sollen jetzt die Tage den **Monaten angepasst** werden, jedoch hat der Februar immer **29 Tage**.
3. Zusätzlich soll der Februar jetzt **nur** 29 Tage in allen Jahren haben, die Vielfach von 4 sind.
4. Zusätzlich soll nun beachtet werden, dass **1900 kein Schaltjahr** ist!

3.4. Implementierung eines DEA

Ein Automat kann natürlich nicht nur gezeichnet werden, sondern auch implementiert werden. Wie auch in der 11. Jahrgangsstufe beginnen wir die Implementierung eher "naiv" und arbeiten uns dann voran.

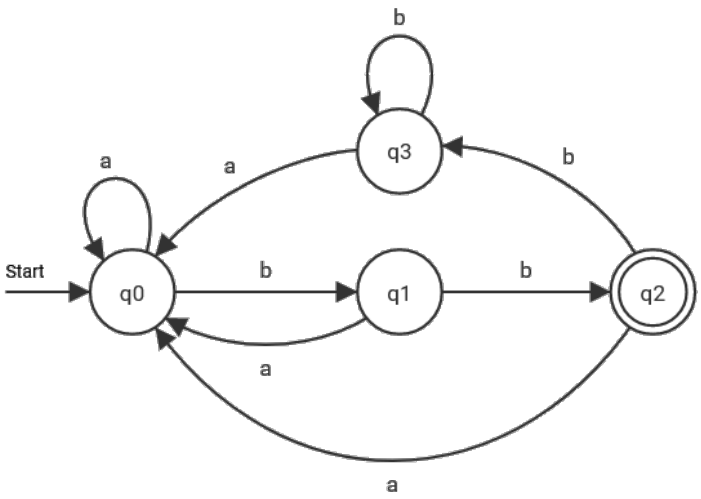
> ***Hinweis***
>
> Da der Anteil der Programmierung im 11.-Klassteil des Abiturs bereits recht hoch ist, sind konkrete Fragen zur Implementierung eines DEA nicht häufig, aber dennoch möglich.

3.4.1. Eine erste Implementierung

Grundsätzlich muss die Repräsentation des Automaten seinen aktuellen **Zustand speichern** und in irgendeiner Form die **Zustandsübergangstabelle** kennen und seinen **Zustand ändern** können, wenn ein Wort überprüft wird. Neben dem Konstruktor bieten sich also folgende Methoden an:

```
boolean testWord(String input)
void switchState(char c)
```

Um an einem konkreten Beispiel zu arbeiten verwenden wir den Automaten **4b)**, der **a-b-Ketten** erkennt, die mit genau 2 **b** enden. Zur Erinnerung:



Wir beginnen mit dem Grundgerüst:

```java
public class DEA {
    //Der Zustand kann der Einfachheit halber als Integer codiert werden:
    private int state;

    //In unserem Fall ist der Startzustand 0
    public DEA() {
        state = 0;
    }
}
```

Für die testWord()-Methode können wir die forEach-Struktur in Java verwenden:

```
public boolean testWord(String input) {
    // Der String muss zunächst in ein Array aus Charactern umgewandelt werden
    for(char c : input.toCharArray()) {
        //Diese Methode muss noch implementiert werden!
        switchState(c);
        //Eine Ausgabe-Methode - nicht zwingend nötig, aber nützlich!
        System.out.println("Zeichen " + c + " führt zu: " + state);
    }
    // Nach Abschluss wird überprüft, ob der Automat im Endzustand steht.
    if(state == 2) {
        //Falls ja wird er in den Anfangszustand versetzt...
        state = 0;
        //und zurückgegeben, dass das Wort in der Sprache enthalten ist
        return true;
    }  else {
        //andernfalls wird zurückgegeben, dass dem nicht so it.
        state = 0;
        return false;
    }
}
```

Die switchState()-Methode sorgt jetzt dafür, dass der Automat beim Lesen einer bestimmten Eingabe den Zustand (also das Attribut state) entsprechend des obigen Zustandsübergangsdiagramms wechselt. Die **Schwierigkeit** besteht dabei darin, dass die Änderung von zwei verschiedenen Parametern abhängt. Es gibt mehrere Ansätze, diesem Problem zu begegnen.

1. **Geschachtelte if-(else)-Bedingungen**: Beispielsweise kann zuerst abgefragt werden, in welchem Zustand sich der Automat befindet. Anschließend wird in jedem Zweig des if seperat ein weiteres if-else zur Abarbeitung des Zeichens.
2. **Geschachtelte switch-case-Strukturen**: Analog zu 1. kann statt einer if-else-Struktur auch die eingebaute switch-case-Syntax verwendet werden - insbesondere wenn es viele Zustände oder Zeichen gibt ist dies die bessere Variante.
3. **Mischung aus 1. und 2.**: Um beides zu veranschaulichen wird in der folgenden Methode für die Auswahl des Zustands switch-case verwendet und für die Entscheidung über das Zeichen if-else.

```
public void switchState(char c) {
    //Die Variable nach der unterschieden werden soll wird ausgewählt - hier state
    switch (state) {
        //Die einzelnen "Fälle" (cases) werden nacheinander abgearbeitet.
        case 0:
            /*Hier genügt ein if-else, da nur zwischen a und b als Eingabe
             unterschieden wird. Der Zustand verändert sich entsprechend.*/
            if(c == 'a') {
                state = 0;
            } else {
                state = 1;
            }
            /*break beendet die Ausführung - wird dies vergessen, so werden die
            übrigen Fälle auch betrachtet - häufiger Fehler!*/
            break;
        //Die restlichen Zustände verlaufen analog
```

```
        case 1:
            if(c == 'b') {
                state = 2;
            } else {
                state = 0;
            }
            break;
        case 2:
            if(c == 'a') {
                state = 0;
            } else {
                state = 3;
            }
            break;
        case 3:
            if(c == 'a') {
                state = 0;
            } else {
                state = 3;
            }
            break;
        /*Sollte ein Zustand nicht definiert sein, wird dieser default-case
        betreten. Hier kann beispielsweise eine Fehlerbehandlung erfolgen.*/
        default:
            break;
    }
```

Diese Implementierung reicht bereits aus! Ein kurzer Test zeigt, dass der Automat korrekte Ausgaben macht:

```
public static void main(String[] args) {
    DEA dea = new DEA();
    System.out.println( dea.testWord("aabb"));
    System.out.println(dea.testWord("abba"));
    System.out.println(dea.testWord("aaaabbbbcaa"));
}
```

liefert “true, false, false” (mit den weiteren Ausgaben von oben dazwischen).

Diese Implementierung hat allerdings offensichtliche Schwächen, z.B.:

1. Wenn in einem Wort “unerlaubte Zeichen” auftreten (das c im dritten Test), so wird dieses einfach übergangen.
2. Es wird nur ein einziger Automat definiert. Um eine andere Sprache erkennen zu können, müsste alles noch einmal neu geschrieben werden.

Die Zustände und die Übergänge sollten idealerweise also “von außen” definiert werden können. Dazu ist die switch-case Struktur nicht geeignet, deswegen verwenden wir stattdessen eine **HashMap**.

3.4.2. Exkurs - Implementierung mit Hilfe einer HashMap

Das Folgende geht wieder weit über den Schulstoff hinaus und ist nur als Anregung zu verstehen.

Um die obigen Probleme lösen zu können, darf der Zustand nicht direkt über den Code geändert werden - die Informationen über den Zustandswechsel und der Quellcode müssen separiert werden.

Es gibt selbstverständlich viele Ansätze, wie dies gelingen kann, eine (bzw. zwei) bieten sich aber besonders an und orientieren sich an der mathematischen Definition der **Zustandsübergangsfunktion δ**, zur Erinnerung:

$$\delta(q, c) = q'$$

wobei q und q' Zustände sind und c ein bestimmtes Zeichen (bei uns jetzt: char in Java!)

Es wird also eine Funktion genutzt, die **zwei** Eingabeparametern genau **ein** Ergebnis zuordnen. Es handelt sich hier allerdings um eine sogenannte **diskrete** Funktion (im Gegensatz zu den kontinuierlichen Funktionen, die üblicherweise in der Mathematik betrachtet werden, also z.B. $f : x \mapsto x^2$ mit $x \in \mathbb{R}$, die keine "Lücken" haben).

Wir kennen bereits zwei Datenstrukturen, die eine solche Zuordnung möglich machen würden, beide sind vernünftig einsetzbar:

1. Ein **2D-Array**: besonders effizient, wenn es ein vollständiger Automat ist.
2. Eine **HashMap**: die bessere Wahl, wenn der Automat nicht vollständig ist. Wenn es vorher nicht bekannt ist also im Schnitt also besser.

In beiden Fällen haben wir noch ein Problem: Java erlaubt es nicht, dass ein Tupel von Objekten einen Schlüssel für z.B. die HashMap oder einen Eintrag für ein Array bilden (Python z.B. erlaubt das!).

Um die Zuordnung also nutzen zu können, müssen wir noch eine "Dummy-Klasse" schreiben, die die beiden Informationen verknüpft (und nicht vergessen equals() und die hashCode()-Methode zu überschreiben, um Vergleiche und das Hashen für die HashMap möglich zu machen.)

Um sicherzustellen, dass bei den Zuständen keine falschen Zeichenketten auftauchen, kann wieder ein Enum verwendet werden, z.B.:

```
public enum State {
    Q0, Q1, Q2, Q3, Q4,
    Q5, Q6, Q7, Q8, Q9,
    Q10, Q11, Q12, Q13, Q14,
    Q15, Q16, Q17, Q18, Q19
}
```

Damit wird natürlich der Zustandsvorrat auf 20 Zustände limitiert, aber das ist aktuell eine nebensächliche Einschränkung. Zurück zur zweiten Hilfsklasse, Pair:

```
public class Pair {
    public State state;
    public char c;

    //Sowohl die Information zum Zustand als auch zum Character werden gespeichert
    public Pair(State state, Character c) {
        this.state = state;
        this.c = c;
    }
```

```
    //Die überschriebene equals-Methode wurde schon im Listenskript behandelt
    @Override
    public boolean equals(Object o) {
        if(this == o) {
            return true;
        }
        if(!(o instanceof Pair)) {
            return false;
        }
        Pair p = (Pair) o;
        if(p.state == this.state && p.c == this.c) {
            return true;
        }
        return false;
    }
    /*Die hashCode-Funktion muss überschrieben werden, um unser Pair als
    Schlüssel in der HashMap verwenden zu können. Wir verwenden einfach
    die Java-interne Hash-Funktion dafür.*/
    @Override
    public int hashCode() {
        return Objects.hash(state, c);
    }
}
```

Der Rohbau hat jetzt mehr Zustände, da sowohl der Startzustand, der aktuelle Zustand, die HashMap als auch die Endzustände gespeichert und gesetzt werden müssen:

```
public class DEA {
    private State state;
    private State[] finalStates;
    private HashMap<Pair, State> stateMap = new HashMap<Pair, State>();
    private State startState;

   public void reset(State startState, HashMap<Pair, State> map, State[] finalStates)
{
        this.startState = startState;
        state = startState;
        stateMap = map;
        this.finalStates = finalStates;
    }
}
```

Die *testWord()*-Methode sieht vom Grundaufbau her ähnlich aus, die *switchState()*-Methode entfällt jedoch, da dafür direkt die HashMap verwendet werden kann. Wir entnehmen mit der aktuellen Kombination aus Zustand und zu verarbeitendem Character den nächsten Zustand aus der HashMap und setzen damit den aktuellen Zustand neu.

Das Entnehmen liefert allerdings standardmäßig null zurück, sollte der Eintrag nicht in der HashMap sein, deswegen wird bei einem “falschen” Zeichen, das nicht im Alphabet enthalten ist, der aktuelle Zustand auf null gesetzt.

```
public boolean testWord(String input) {
    //Hilfs-Ausgabemethode
    System.out.println("Start bei Zustand " + state);
    //Analog zur ersten Implementierung
    for(char c : input.toCharArray()) {
        /*Der Zustand wird neu gesetzt, mit "get" erhält man
        den Eintrag in der HashMap*/
        state = stateMap.get(new Pair(state, c));
        //Falls das aktuelle Paar nicht bekannt ist, wird null zurückgegeben
        if(state == null) {
            return false;
        }
        System.out.println("Zeichen " + c + " führt zu: " + state);
    }
    if(Arrays.asList(finalStates).contains(state)) {
        state = startState;
        return true;
    } else {
        state = startState;
        return false;
    }
}
```

Damit ist die Implementierung schon abgeschlossen! Der Nachteil ist natürlich, dass damit die “Arbeit” des Automaten-Erstellens auf den Nutzer abgeschoben wird, die Testmethode ist dementsprechend länger:

```
public static void main(String[] args) {
    State startState = State.Q0;
    State[] finalStates = {State.Q2};
    HashMap<Pair,State> testMap = new HashMap<Pair, State>();
    //Es müssen alle Einträge der Zustandsübergangstabelle  angegeben werden!
    testMap.put(new Pair(State.Q0, 'a'), State.Q0);
    testMap.put(new Pair(State.Q0, 'b'), State.Q1);
    testMap.put(new Pair(State.Q1, 'a'), State.Q0);
    testMap.put(new Pair(State.Q1, 'b'), State.Q2);
    testMap.put(new Pair(State.Q2, 'a'), State.Q0);
    testMap.put(new Pair(State.Q2, 'b'), State.Q3);
    testMap.put(new Pair(State.Q3, 'a'), State.Q0);
    testMap.put(new Pair(State.Q3, 'b'), State.Q3);
    System.out.println(testMap.get(new Pair(State.Q0, 'b')));
    DEA dea = new DEA();
    dea.reset(startState, testMap, finalStates);
    System.out.println(dea.testWord("aabb"));
    System.out.println(dea.testWord("aabba"));
    System.out.println(dea.testWord("aabbcc"));
}
```

Der logische nächste Schritt wäre nun also, ein Format zu finden, in dem der Automat schneller angegeben werden kann und dann einen Konverter zu schreiben, der diesen Prozess oben automatisiert - das überlasse ich aber dem geneigten Lesy zur Übung. Flaci z.B. kann in ein JSON-Format exportieren.

Bibliography

[1] D. W. Hoffmann, *Theoretische Informatik*. München: Carl Hanser Verlag, 2018.