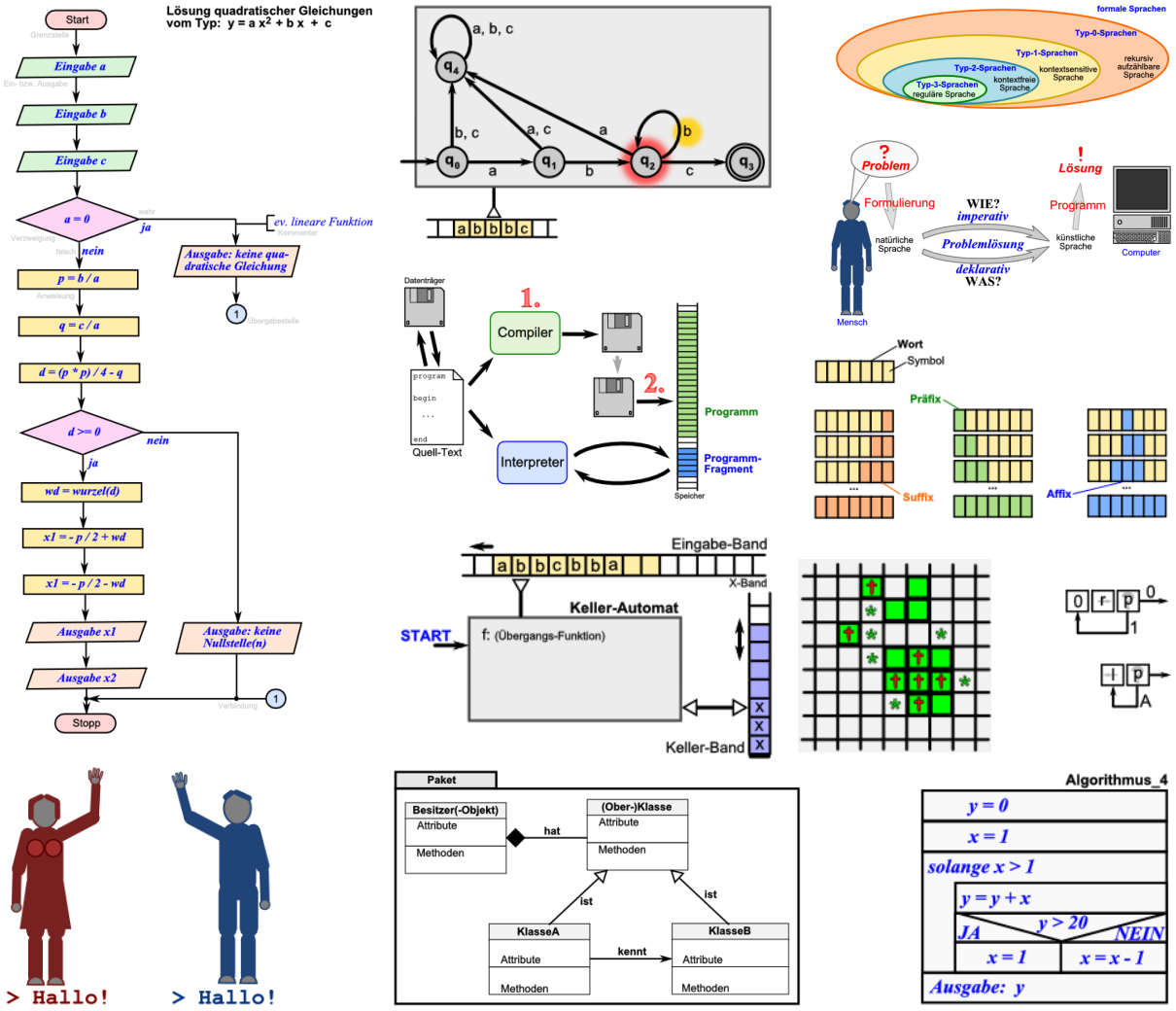
Informatik

für die Sekundarstufe II

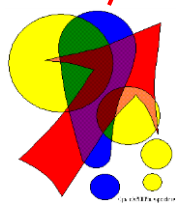
- Sprachen und Automaten -

Autor: L. Drews



teilredigierte Version 0.11c (2024)

Legende:
mit diesem Symbol werden zusätzliche Hinweise, Tips und weiterführende Ideen gekennzeichnet



Nutzungsbestimmungen / Bemerkungen zur Verwendung durch Dritte:

(1) Dieses Skript (Werk) ist zur freien Nutzung in der angebotenen Form durch den Anbieter (lern-soft-projekt) bereitgestellt. Es kann unter Angabe der Quelle und / oder des Verfassers gedruckt, vervielfältigt oder in elektronischer Form veröffentlicht werden.

(2) Das Weglassen von Abschnitten oder Teilen (z.B. Aufgaben und Lösungen) in Teildrucken ist möglich und sinnvoll (Konzentration auf die eigenen Unterrichtsziele, -inhalte und -methoden). Bei angemessen großen Auszügen gehört das vollständige Inhaltsverzeichnis und die Angabe einer Bezugsquelle für das Originalwerk zum Pflichtteil.

(3) Ein Verkauf in jedweder Form ist ausgeschlossen. Der Aufwand für Kopierleistungen, Datenträger oder den (einfachen) Download usw. ist davon unberührt.

(4) Änderungswünsche werden gerne entgegen genommen. Ergänzungen, Arbeitsblätter, Aufgaben und Lösungen mit eigener Autorenschaft sind möglich und werden bei konzeptioneller Passung eingearbeitet. Die Teile sind entsprechend der Autorenschaft zu kennzeichnen. Jedes Teil behält die Urheberrechte seiner Autorenschaft bei.

(5) Zusammenstellungen, die von diesem Skript - über Zitate hinausgehende - Bestandteile enthalten, müssen verpflichtend wieder gleichwertigen Nutzungsbestimmungen unterliegen.

(6) Diese Nutzungsbestimmungen gehören zu diesem Werk.

(7) Der Autor behält sich das Recht vor, diese Bestimmungen zu ändern.

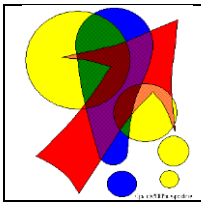
(8) Andere Urheberrechte bleiben von diesen Bestimmungen unberührt.

Rechte Anderer:

Viele der verwendeten Bilder unterliegen verschiedensten freien Lizenzen. Nach meinen Recherchen sollten alle genutzten Bilder zu einer der nachfolgenden freien Lizenzen gehören. Unabhängig von den Vorgaben der einzelnen Lizenzen sind zu jedem extern entstandenen Objekt die Quelle, und wenn bekannt, der Autor / Rechteinhaber angegeben.

public domain (pd)	Zum Gemeingut erklärte Graphiken oder Fotos (u.a.). Viele der verwendeten Bilder entstammen Webseiten / Quellen US-amerikanischer Einrichtungen, die im Regierungsauftrag mit öffentlichen Mitteln finanziert wurden und darüber rechtlich (USA) zum Gemeingut wurden. Andere kreative Leistungen wurden ohne Einschränkungen von den Urhebern freigegeben.
gnu free document licence (GFDL; gnu fdl)	
creative commens (cc) creative commons	BY: od. neu … Namensnennung … nichtkommerziell … in der gleichen Form … unter gleichen Bedingungen

Die meisten verwendeten Lizenzen schließen eine kommerzielle (Weiter-)Nutzung aus!



Bemerkungen zur Rechtschreibung:

Dieses Skript folgt nicht zwangsläufig der neuen **ODER** alten deutschen Rechtschreibung. Vielmehr wird vom Recht auf künstlerische Freiheit, der Freiheit der Sprache und von der Autokorrektur des Textverarbeitungsprogramms microsoft ® WORD ® Gebrauch gemacht.

Für Hinweise auf echte Fehler ist der Autor immer dankbar.

Inhaltsverzeichnis:

Seite

0. Vorwort

> Studium ohne Hingabe schadet dem Gehirn.
> Leonardo DA VINCI

Um dem unvorbereiteten Leser ein wenig Orientierung zu geben, was für ihn interessant bzw. notwendig ist, sind die Abschnitte mit Niveau-Kennungen versehen.
Entweder handelt es sich um Bereich von bis oder um Einzel-Festlegungen.
Die Niveau-Stufen sind von mir folgendermaßen gekennzeichnet:
Das Grundlagen-Niveau ist für alle Leser gedacht. Hier werden allgemeine Sachverhalte vorgestellt, die auch nicht immer zwangsläufig zur Informatik gehören.

Im Basis-Niveau betrachten wir die Dinge, die man einem Fach oder Grundkurs zuordnen würde. Ich orientiere mich dabei auch am Kern-Curriculum für Berlin, Brandenburg und Mecklenburg-Vorpommern. Ev. passt es genau so auch in anderen Bundesländern? Dabei können einzelne Themen mehr oder weniger ausgefüllt werden. Vieles liegt im Ermessen des Kurs-Leiters oder der Planung in der Schule.
Für die Leistungskurse oder das Hauptfach sind die Inhalte mit dem Fortgeschrittenen-Niveau. Auch hier gilt, dass ohne weiteres Gebiete aus dem untergeorneten Niveau schon die Grenze des Lehrplans sind, aber es könnten auch Themen des – von mir subjektiv eingestuften - Experten-Niveaus inhaltlich vorgeschrieben sein.

Der Kurs-Leiter sollte immer eine auf die Bedingungen und Anforderungen zugeschnittene Themen-Auswahl vornehmen. Einige deutlich weitergehende Themen sind als Experten-Niveau klassifiziert.
Wer das Skript als Wiederholung / Vorbereitung für das Studium benutzt, muss natürlich explizit auf die Anforderungen des Lehrstuhls achten. Sonst könnte es sein, dass man mit seinem Niveau deutlich unter den Forderungen liegt.
Experten-Themen eigenen sich auch für Projekte, Grob-orientierungen für Schüler-Vorträge usw. usf. Wen sie nicht interessieren, der überspringt sie einfach. Man muss nicht alles wissen, man muss nur wissen, wo es steht und wo man sich ev. wieder belesen kann.

interssante Links:
http://www.computerlexikon.com
http://flemming-universum.de

http://schuelerlabor.informatik.rwth-aachen.de (diverse Informatik-Projekte und -Module)

1. Algorithmik

Problem-Fragen für Selbstorganisiertes Lernen
Womit beschäftigt sich die Theoretische Infromatik? Was sind Algorithmen? Gibt es auch außerhalb der Computerwelt Algorithmen? Wenn JA, welche? Wie lassen sich Algorithmen in Programme umsetzen? Was sind Compiler und was Interpreter? Was unterscheidet sie und was haben sie gemeinsam? Welche Arten der Programmierung sind möglich? Welche Art ist die Beste? Kann man jedes Problem mit jeder Programmiersprache lösen? Welche Programmiersprache ist die Beste? Ist alles berechenbar? Wie aufwendig können Berechnungen werden? Kann man alles in absehbarer Zeit berechnen? Hört jeder Rechner / jeder Algorithmus irgendwann mal auf? Kann jeder Rechner jeden Algorithmus abarbeiten? Welche allgemeinen Daten-Strukturen gibt es? Mit welchen allgemeinen Methoden kann man auf Daten zugreifen oder sie verändern.

In der Informatik geht es genau so wenig um Computer, wie in der Astronomie um Teleskope.
Edsger Wybe DUKSTRA

1.0. Wissenschaft Informatik und ihre Teilgebiete

Definition(en): Informatik
Informatik ist die Wissenschaft, die sich mit der Verarbeitung, Darstellung, Speicherung und Übertragung von Daten beschäftigt.
Informatik ist die Wissenschaft, die sich mit den Fragen rund um die Verarbeitung von Daten beschäftigt.

wichtige Teilbereiche der Informatik

- **praktische Informatik** — Software-Entwicklung; Entwicklung von Programmierungs-Techniken, Testverfahren
- **technische Informatik** — Hardware-Entwurf, Maschinen-nahe Programmierung
- **angewandte Informatik** — Nutzung der Informatik in anderen Wissenschaften und in der Gesellschaft
- **theoretische Informatik** — mathematische und theoretische Grundlagen
- **…**

Definition(en): Theoretische Informatik
Die Theoretische Informatik ist der Teilbereich der Informatik, der sich mit den allgemeinen (theoretischen) – meist mathematischen und / oder logischen – Grundlagen beschäftigt.
Die Theoretische Informatik ist der der Teil der Informatik, der sich mit Abstraktionen, Modell-Bildung, allgemeinen Fragen der Informations-Verarbeitung befasst.

Arbeitsfelder der Theoretischen Informatik

- **Algorithmen-Theorie / Algorithmen-Analyse**
- **Berechenbarkeits-Theorie**
- **Automaten-Theorie**
- **Komplexitäts-Theorie**
- **formale Sprachen und Grammatiken**
- **formale Semantik**
- **Codierungs-Theorie**
- **Theorie der Programmierung**
  - **Compiler- und Interpreter-Bau**
  - **automatische Programm-Synthese / -Generierung**
- **Kryptologie**
  - **Chiffrierung / Dechiffrierung**
  - **Kryptoanalyse**
  - **Authentifizierung**
  - **Steganograpphie**
- **Künstliche Intelligenz**
  - **Bild-Erkennung / Bild-Bearbeitung / Computervision**
  - **Machinelles Lernen**
  - **Sprach-Erkennung / Sprach-generierung**
  - **Künstliche Neuronale Netze / Deep Learning**
  - **Wissens-Repräsentation / Wissens-Management**
  - **…**
- **…**

Bevor wir die Algorithmen-Theorie etwas näher beleuchten, sei kurz etwas zu den anderen Arbeitsfeldern gesagt.
Berechenbarkeits-Theorie: In diesem Arbeitsfeld geht es allgemein darum, ob alles in mathematische Modelle beschrieben und somit berechenbar ist. Im Speziellen geht dann auch darum, ob die verwendeten Modelle / Algorithmen immer Lösungen bringen. Die Frage nach der Terminiertheit von Programmen und Algorithmen ist eine der wichtigen Arbeitsbereiche.
Komplexitäts-Theorie: Wenn man weiss, dass ein Modell / Algorithmus berechenbar ist, dann ist damit noch lange nicht gesagt, dass man es auch praktisch. Viele Probleme – z.B. bestimmte Dechiffrierungen, Berechnungen sind so kompliziert (aufwendig), dass man sie garnicht mit der heutigen Technik lösen kann. Viele sind sogar so komplex, dass auch in Zukunft mit keiner praktischen Lösung gerechnet wird. Viel Hoffnung setzt man aber auf Quanten-Computer.
Codierungs-Theorie: Damit etwas mit mathematischen Modellen und informatischen Algorithmen berechnet werden kann, müssen die Sachverhalte in Zahlen – und letztendlich in Binär-Werte – umgesetzt werden. Mit diesem Thema beschäftigt sich die Codierungs-Theorie. Fälschlicherweise denken viele Laien bei Codierung an Verschlüsselung a'la Geheimhaltung. Das ist Thema der Kryptologie oder auf der gleichen Begriffs-Ebene – der Chiffrierung. Bei Codierung geht es um die Umschreibung von Daten in Zahlen / Binär-Werte nach festen und allgemein bekannten Regeln und Systemen.
Kryptologie: In diesem Arbeits-Bereich, der vielfach völlig eigenständig gesehen wird, geht es um die die geheime Umsetzung von Daten. Die Chiffrierung realisiert genau diese Verschlüsselung. Neben der reinen praktischen Arbeit geht es in der Kryptologie natürlich auch um das Knachen fremder / unbekannter Chiffren. Die Krypto-Analyse ist praktisch als "Wissenschaft" genausoalt, wie es Verschlüsselungen gibt.
Heute umfasst die Kryptologie auch Verfahren zur sicheren Authentifizierung von Kommunikanten in Netzen, in denen sich niemals alle Teilnehmer kennen können.
Kein Thema erhitzt so die Gemüter und Diskussionen, wie das Thema **Künstliche Intelligenz**. Besonders schwierig ist dieses Thema deshalb, weil es bis heute keine gültigen Definitionen dazu gibt was Intelligenz allgemein und was künstliche Intelligenz im Speziellen ist. Und nicht's ist in der Theoretischen Informatik so verpönt, wie fehlende oder nicht brauchbare Definitionen.

Die Künstliche Intelligenz ist ein echtes Mode-Thema, was sicher in den nächsten Jahren und Jahrzehnten eine weitere rasante Entwicklung nehmen wird. Besonders dem Maschinellen Lernen (ML) und der Wissens-Repräsentationen werden riesige Potentiale zugesprochen. Die KI ist mitlerweise ein so umfassendes Gebiet und mit so vielen spannenden Detail's gespickt, dass wir ein separates Skript dazu entwickeln (🕮 **Projekt: Deep Learning**).

Algorithmen-Theorie

Begriff Algorithmus leitet sich aus dem Namen eines berühmten arabischen Universalgelehrten Abu Dscha'far Muhammad ibn Musa al-CHWARIZMI.
Der Name wurde lange als Autoritäts-Beweis bzw. –Begründung für Rechenregeln und Lehrsätze verwendet: "Algorismi hat gesagt, …".
Im 13. und 14. Jahrhundert wurde der Begriff dann auch viel für "Schritte der Anweisung" in wissenschaftlichen und kirchlich-katechistischen Anwendungen benutzt.

Definition(en): Algorithmus
Ein Algorithmus ist eine Klasse präzise formulierter Handlungsanweisungen, die bei vorgegebenen Eingangs-Daten mit Hilfe üblicher / definierter Regeln zu einem Ergebnis (Ausgangs-Daten) führt.
Ein Algorithmus ist eine eindeutige Vorschrift zur Lösung eines Problems / einer Aufgabe.
Ein Algorithmus ist eine päzise, ausführbare Anweisungs-Folge endlicher Länge, die zur Lösung einer Aufgabe führt.
Ein Algorithmus ist eine in einer definierten Sprache formulierte Arbeitsvorschrift zur Bearbeitung einer Aufgabe.
Ein Algorithmus ist ein Lösungs-Verfahren für ein Problem.
Ein Algorithmus ist eine Vorschrift zur Lösung einer Klasse von Problemen / Aufgaben.
Ein Algorithmus ist ein Verfahren, das in einem endlichen Text notiert werden kann, effektiv von einem Menschen oder einer Maschine ausgeführt werden kann, aus einer begrenzten Menge von Elementar-Operationen besteht, Ein- und Ausgaben ermöglicht und zu jeder Eingabe ein reproduzierbares Ergebnis zurückliefert.

Weitere Definition werden wir im Abschnitt zu den Automaten besprechen (→ x. Automaten).
Sie gelten im informatischen Bereich als sehr gut gefasste Definition.
Wichtig ist es auch immer, im Hinterkopf zu behalten, das der Algorithmus für sich endlich sein mus, nicht die Ausführung. Ein klassisches Beispiel hierfür könnte eine Endlos-Schleife sein. Sie wird oft für die Aufrechterhaltung des Systembetriebes benötigt.

```
…
while True:
    #Anfang des ständig auszufühenden Blocks
    …
    #Ende des stämdig auszuführenden Blocks
…
```

Endlos-Schleife in Python

Ein Algorithmus ist eine Folge von Regeln zur Bildung komplizierter mathematischer Funktionen aus einfacheren mathematischen Funktionen. GÖDEL
Ein Algorithmus ist eine Menge von Anweisungen für eine einfache (TURING-)Maschine. TURING
Ein Algorithmus ist ein Verfahren, das bei geeigneter GÖDELisierung eine zugeordnete Funktion berechnet.

Stellt sich natürlich die Frage: "Gibt es zu allen (mathematisch) formulierbaren Problemen eine Algorithmus?"
Heute müssen wir leider sagen: NEIN. Eines dieser Probleme ist das sogenannte "Halte-Problem". Es ist ein Gegenstand der Theoretischen Informatik. Wir kommen auf dieses Problem später noch zurück (→).

Exkurs: historische u.a. Definitionen des Begriffes Algorithmus

Q:

Ein Algorithmus ist ein Verfahren, welches mit Hilfe einer entsprechenden Maschine realisiert werden kann.
J. R. SHOENFIELD (1967)

Unter einem Algorithmus versteht man eine genaue Vorschrift, nach der ein gewisses System von Operationen in einer bestimmten Reihenfolge auszuführen ist und mit der man alle Aufgaben eines gegebenen Typs lösen kann.
TRACHTENBROT (1977)

Ein Algorithmus ist eine mechanische Regel oder eine automatisierte Methode oder eine Programm für die Ausführung mathematischer Funktionen.
N. J. CUTLAND (1980)

Ein Algorithmus ist eine Folge von Regeln zur Erzeugung komplizierter mathematischer Funktionen aus einfachen mathematischen Funktionen.
GÖDEL (19??)

Alles was berechenbar ist lässt sich als Algorithmus darstellen
?? (19??)

Ein Algorithmus ist eine Menge von Anweisungen für einen einfachen Automaten (heute TURING-Maschine genannt).
TURING (19??)

Q:

CHURCH-TURING-These:
1. Alle vernünftigen Definitionen von "Algorithmus", soweit sie bekannt sind, sind gleichwertig und gleichbedeutend.
2. Jede vernünftige Definition von "Algorithmus". die jemals irgendwer aufgestellt hat, ist gleichwertig und gleichbedeutend zu denen, die wir kennen.

allgemeine Beispiele für Algorithmen:

- Vorschriften zum Addieren, Subtrahieren, Multiplizieren, Dividieren, Potenzieren, … von Zahlen
- (Zusammen-)Bau-Anleitung für ein Möbelstück
- Hochsteigen einer Treppe
- Erstellen eines Diagramms aus Daten
- Spielen von "Mensch ärgere dich nicht"
- Bedienung eines Handy's / Smartphone's
- Einstellen eines Bildes beim Mikroskopieren
- Musizieren / Spielen nach Noten
- Prozent-Rechnung
- Wechseln einer Glühlampe
- Berechnen des größten gemeinsamen Teilers zweier Zahlen
- Polynom-Division
- Übertragung eines Textes mittels MORSE-Zeichen
- Reinigen einer Waffe
- Lösen einer linearen Gleichung
- Login in ein Rechner-Netz
- Bedienungs-Vorschrift für einen Fernseher
- Wechseln eines Autorades
- Wechseln aller vier Räder (→ 4x "Wechseln eines Autorades")
- Lösen einer quadratischen Gleichung
- Festlegen der Suche-Ergebnisse in google (PAGE-Rang-Algorithmus)
- Verschlüsseln / Entschlüsseln eines Textes
- Bilden der Ableitung einer Funktion
- Berechnung des Volumens und der Oberfläche eines Quaders
- Basteln nach eine Anleitung
-

Struktur-Elemente eines Algorithmus (algorithmische Grund-Bausteine)

• **Folge** **Sequenz / Aneinanderreihung**	Aneinanderreihung von mindestens zwei Anweisungen oder Struktur-Elementen eines Algorithmus
• **Auswahl** **Verzweigung / Entscheidung** **Selektion / Bifurkation**	Auswahl einer Anweisung oder eines Strukur-Elementes eines Algorithmus in Abhängigkeit von einer Bedingung
• **Wiederholung** **Schleife / Schlaufe / Zyklus / Loop / Rückkopplung**	bestimmte oder unbestimmte Wiederholung von Algorithmen-Teilen

Definition(en): algorithmische Grund-Bausteine
Algorithmische Grund-Bausteine sind abstrakte Strukturen von Anweisungen. Zu den algorithmischen Grund-Bausteinen zählt man Folgen, Verzweigungen und Schleifen.

Merkmale eines Algorithmus / Anforderungen an einen Algorithmus

- **Endlichkeit (Finitheit)**

  ein Algorithmus muss aus einer endlichen Zahl von Schritten mit einer endlichen Länge bestehen
  dabei zählen nur die formulierten / angegebenen Schritte des Algorithmus selbst, nicht die Anzahl der Schritte während der Abarbeitung
  unter dynamischer Finitheit versteht man die Eigenschaft, dass der Algorithmus während der Abarbeitung nur eine endliche menge an Speicherplätzen in Anspruch nimmt

- **Eindeutigkeit (Determiniertheit, Wiederholbarkeit)**

  ein Algorithmus muss bei gleichen Eingangswerten und gleichen Bearbeitungsbedingungen immer die gleichen Ausgabewerte erzeugen.
  (Ein Algorithmus darf nicht der Willkür des Anwenders unterliegen.)
  - deterministische Algorithmen können zu jedem Zeitpunkt einer Unterbrechung auf die Determiniertheit überprüft werden und ergeben bei gleichen Eingaben und Bedingungen immer das gleiche Ergebnis
  - nicht-deterministische Algorithmen ist der weitere Ablauf nicht eindeutig vorhersehbar, dadurch sind die Ergebnisse nicht eindeutig vorhersagbar

- **Ausführbarkeit**

  ein Algorithmus darf nur Anweisungen enthalten, die vom ausführenden System auch verarbeitet / erledigt werden können.
  (Eine Schrittfolge ist nur dann für den Ausführenden ein Algorithmus, wenn er die Folge abarbeiten / verstehen kann.)

- **Terminiertheit**

  ein Algorithmus muss nach einer endlichen Zahl von Arbeitschritt zu einem Ende kommen (/ terminieren)
  hierbei geht nur um die Arbeitsschritte, die bei einer Abarbeitung des Algorithmus erledigt werden
  ein Algorithmus ist dann terminierend, wenn er bei jeder Art (zugelassener) Eingaben zu einem – wie auch gearteten – Ergebnis kommt
  für Eingaben, die keine Lösung (im Sinne des Algorithmus) ermöglichen, kann ein Algorithmus terminieren, musses er aber nicht

- **Bestimmheit**

  in einem Algorithmus muss zu jedem Zeitpunk / Arbeitstakt feststehen, welcher Schritt als nächstes erledigt werden muss.

- **Allgemeingültigkeit (Allgemeinheit, Abstraktion)**

  ein Algorithmus muss alle Aufgaben des gleichen Typs abarbeiten können. Die Lösbarkeit bzw. Nichtlösbarkeit einer Aufgabe muss eindeutig bestimmt sein.
  als Aufgaben eines Typ's werden dabei vorrangig solche mit veränderten Eingaben verstanden

- **Diskretheit**

  ein Algorithmus besteht aus einzelnen Schritten zusammehngesetzt
  die Schritte werden auch Elementar-Operationen genannt

- **Korrektheit**

  das Ergebnis eines Algorithmus muss immer exakt sein

weiterhin können die Merkmale:

- **Verständlichkeit (Klarheit)** — ein Algorithmus sollte so formuliert sein, dass er von mehreren informatischen System (einschließlich des Menschen) nachvollzogen (verstanden) werden kann
  dabei muss ein Algorithmus nicht direkt verstanden werden, es reicht die Übersetzbarkeit in z.B. von Menschen verstandene Ausdrücke
- **Effizienz** — ein Algorithmus sollte so gestaltet sein, dass er mit den Ressourcen (üblicherweise: Speicherplätze, Rechenzeit) bestmöglich umgeht
- **Implementierbarkeit (Exaktheit)** — meint die Umsetzbarkeit in eine Form, die z.B. von einer Maschine verstanden werden kann (Programm)
- **Verifizierbarkeit** — der Algorithmus sollte hinsichtlich seine Korrektheit (und der Terminierung) überprüfbar sein

hinzugezählt werden.
Leistungs-Merkmale sind:

- **Speicherplatz-Bedarf**
- **Rechenzeit(-Aufwand)**
- **Parallelisierbarkeit**

Für informatische Prozesse – zu denen eben Algorithmen gehören – gilt das EVA-Prinzip. Auf Algorithmen bezogen heißt das, es existieren Eingabe-Daten, die verarbeitet / umgewandelt werden. Als Ergebnis entstehen Ausgabe-Daten.

Prozess / Vorgang	Eingabe-Daten	Verarbeitung	Ausgabe-Daten
Multiplikation von zwei natürlichen Zahlen	2 natürliche Zahlen a und b	a wird b-mal zu 0 addiert	Produkt (a*b)
Bestimmen des Volumen's eines Quaders	Höhe, Breite, Länge	Berechnung Höhe*Breite*Länge	Produkt (h*b*l)
Ermitteln des Preises einer Ware an der Kasse	Strich-Code (Barcode) Händler-Warenwirtschafts-System (Datenbank)	Umwandeln des Barcodes in eine Zahl (→ EAN) Suche der EAN in Datenbank Ermitteln des Preises	Preis
Bezahlen mit der EC-Karte	EC-Karte mit IBAN-Kontonummer PIN	Berechnen der Berechtigung aus IBAN und PIN	Berechtigung (JA oder NEIN)
Erraten einer Zufalls-Zahl zwischen 1 und 1000	Zufalls-Zahl geratene Zahl	Prüfen auf Gleichheit	Übereinstimmung (JA oder NEIN)

Algorithmus oder nicht, das ist hier die Frage.

Aufgabe / Problem / …	Terminiertheit (endlich)	Determiniertheit (eindeutig)	Finitheit (ausführbar)	Abstraktion (allgemeingültig)	Bestimmtheit (bestimmt)		Schlussfolgerung
Berechnung der Summe zweier Zahlen	✓	✓	✓	✓	✓		→ Algorithmus
fortlaufende Multiplikation einer bestimmten Zahl mit 3	✗	✓	?	?	✓		
Berechnung aller Primzahlen	✗	✓	✓	?	✓		
Berechnung der Primzahlen im Intervall von a bis b	✓	✓	✓	✓	✓		→ Algorithmus
Test, ob eine Zahl durch x teilbar ist	✓	✓	✓	✓	✓		→ Algorithmus
Benotung einer Englisch-Klausur	✓	✗	✓	✗	?		
Ziehen eines Schoko-Riegel's an einem Snack-Automaten	✓	✓	✓	✓	✓		→ Algorithmus
Hausarbeit (putzen, saugen, Staub wischen, …)	?	✗	✓	✗	?		
Bestimmung der Nullstellen einer Funktion	✓	✓	✓	✓	✓		→ Algorithmus
Komponieren einen Top10-Hits	✓	✓	?	✗	?		
Kochen eines Frühstücks-Ei's	✓	✓	✓	✓	✓		→ Algorithmus
Konstruktion der Winkelhalbierenden eines vorgegebenen Winkels	✓	✓	✓	✓	✓		→ Algorithmus
Einschließen eines Koffer's in einem Gepäck-Schließfach	✓	✓	✓	✓	✓		→ Algorithmus
Subtraktion von zwei natürlichen Zahlen	✓	✓	✓	✓	✓		→ Algorithmus
Umwandeln einer römischen Zahl in eine arabische	✓	✓	✓	✓	✓		→ Algorithmus
Multiplikation von vier aufeinanderfolgenden ungeraden ganzen Zahlen	✓	✓	✓	✓	✓		→ Algorithmus
Einnehmen von Medikamenten an einem Tag	✓	✓	✓	✓	✓		→ Algorithmus

Aufgaben:

1. Prüfen Sie, ob die nachfolgenden Aufgaben / Probleme durch einen Algorithmus lösbar sind! Begründen Sie Ihe Meinung!

a) abwechselndes Addieren und Multiplizieren von zwei Zahlen
b) Zählen aller Vokale in einem Text
c) Auflisten aller Quadratzahlen
d) Ordnen von Zahlen der Größe nach absteigend
e) Schreiben eines Romans
f) Ausdrucken aller geraden Zahlen aus dem Bereich 0 bis 1'000'000
g) Führen einer Diskussion über quadratische Gleichungen
h) Schießen eines Tores in einem Fußballspiel
i) Differenzieren eines Polynoms
j) Ermitteln von a und b in einer linearen Gleichung vom Typ $ax + b = 0$
k) Schreiben eines Gedichtes
l) Lösen einer quadratischen Gleichung vom Typ $y = ax^2 + bx + c$
m) Verschlüsseln eines Textes
n) Erzeugen eines Strickmuster für einen Pullover

Definition(en): Anweisung
Eine Anweisung ist eine Aufforderung an ein verarbeitendes System eine bestimmte Funktion auszuführen / zu erfüllen.

Definition(en): Elementar-Anweisung
Eine Elementar-Anweisung ist eine – in einem bestimmten System – kleinste ausführbare Aufforderung. Elementar-Anweisungen lassen sich zu zusammengesetzten bzw. strukturierten Anweisungen kombinieren.

Definition(en): zusammengesetzte / strukturierte Anweisung
Eine zusammengesetzte oder strukturierte Anweisung ist eine Gruppe zusammengehörender, kombinierter Elementar-Anweisung, die eine komplexe Reaktion des System ermöglicht.

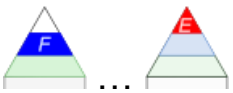
Maschinen – gemeint sind da natürlich vorrangig Computer – sollen Aufgaben für uns übernehmen. Das funktioniert nur, wenn es einen passenden Algorithmus gibt. Der Maschine muss gesagt werden, was sie wann, wie tuen soll. Die Formulierung von Algorithmen für Maschinen nennen wir Programmierung.
(Da wir Menschen im weiteren Sinne auch Maschinen sind (genetische Maschinen mit dem Hauptzweck genetisches Material zu vervielfachen), gilt das Prinzip der Programmieung auch für uns. Auch Verhaltens-Muster, Normen usw. werden durch die verschiedenen Gemeinschaften auf Personen programmiert. Die Betrachtung dieser Sachverhalte muss aber in den Gesellschaftswissenschaften (Psychologie, Sozialwissenschaft, Philosophie, Rechtswissenschaften , …) erfolgen.

Definition(en): Programmierung
Unter Programmierung versteht man den Teil der Software-Entwicklung, der sich mit dem Erstellen von (Computer-)Programmen beschäftigt.
Programmierung ist die Tätigkeit und das Verfahren zum Umsetzen von Algorithmen zu dessen Abarbeitung auf Computern.
Programmierung ist die Codierung von Algorithmen für Computer.
Programmierung ist die Umsetzung eines Algorithmus in eine Programmiersprache.
Programmierung ist die Festlegung von Operationen, die ein Computer in der vorgegebenen Art und Weise ausführen soll.
Programmierung ist die Erstellung von Software (Programme und Daten(strukturen)) für einen Computer.

Hier seien kurz einige moderne Tendenzen und Aspekte der Programmierung genannt.
Eine der herausragenden Tendenzen der letzten Jahre ist die agile Programmierung. Bei ihr steht der Mensch – als Programmierer und Nutzer – im Vordergrund.
Ziel agiler Techniken ist die Reduzierung bürokratischer Abläufe. Vor allem die vorlaufenden (theoretischen) Entwicklungs-Phase(n) sollen verkleinert und effektiviert werden.
Agile Programmierung steht für die frühzeitige Bereitstellung von Prototypen und / oder zumindestens teilweise funktionierenden Teil-Entwicklungen, um mit dem Kunden / Nutzer in Dialog treten zu können und die zukünftige Software-Entwicklung vor allem in Richtung Nutzer-Freundlichkeit zu leiten.
Agiles Arbeiten bietet sich bei komplexen, langfristigen Entwicklungen an, die sich an vielen aktuellen Veränderungen orientieren müssen.
Eine zweite moderne Richtung ist die Pair-Programmierung. Dabei arbeiten Programmierer praktisch immer in Zweier-Teams. Einer programmiert direkt am Computer und der andere kontrolliert sofort, hinterfragt Arbeits-Taktiken und Algorithmen und macht gleich Verbesserungs-Vorschläge. Es wird also das Vier-Augen-Prinzip angewandt.
Die Fehler-Zahl in Programmen, die mittels Pair–Programmierung entwickelt wurden, ist deutlich kleiner, als bei solchen die erst von einem Programmierer erstellt und dann von einem anderen kontrolliert oder überarbeitet wurden. Die Entwicklungs-Geschwindigkeit liegt in vergleichbarer Größe, als würden beide Programmierer einzeln arbeiten. Gut eingespielte Teams überschreiten mit Rollenwechseln auch die Einzel-Arbeits-Geschwindigkeiten.
Vorteil der Pair–Programmierung ist z.B. auch die Pool-Bildung von kompetenten, auskunftsfähigen Personen. Da jetzt immer zwei Personen über eine Software – oder eine spezielle Komponente Bescheid wissen, ist eine Wissen-Lücke bei Krankheit, Arbeitsplatz-Wechsel usw. usf. deutlich reduziert. Ausscheidende Personen werden schnellstmöglich ersetzt und

von den / der anderen angeleitet. Nicht selten werden Junior- und Senior-Programmierer kombiniert.
I.A. ist es nicht wichtig, welche Programmiersprache man lernt. Wer programmieren kann, de stellt sich schnell auf eine andere Programmiersprache um. Es ist kaum damit zu rechnen, dass die heute aktuellen – und vielleicht hochgelobten – Programmiersprachen in 10 oder 20 Jahren noch benutzt werden.
Wichtig ist im Zusammenhang mit Programmierung das systematische Lösung von Aufgaben (Problemen). Wir müssen durch passende Taktiken und Startegien komplexerer Zusammenhänge beherschen lernen.
Hauptziel ist deshalb das Erlernen eines Denken in Strukturen und Algorithmen. Genau diese Aspekte werden in den verschiedensten Lehrbüchern immer vorkommen, egal welche Programmiersprache sie propagieren. Natürlich gibt es auch immer selbstverliebte Kurse, Kursleiter, Programmier-Konzepte usw. Lassen wir ihnen ihre Freude und schauen hinter die Fassade, da sehen wir auch hier wieder die zentralen Konzepte: Algorithmen und Strukturen.
Einer der herausragende Ziele der Programmierung im schulischen Umfeld ist das Erlernen der systematischen Konstruktion von Programmen. Wie geht man vor, um ein (informatisches) Problem in eine nutzbare Software umzusetzten. Wichtig ist dabei auch immer die kritsche Analyse der eigenen Tätigkeiten und des erstellten Programm-Codes. Wer es früh lernt mit der eigenen Unzulänglichkeit zu leben, seine eigenen Fehler zu erkennen, sie offensiv und konstruktiv zu beheben, der wird in größeren Teams und bei komplexen Aufgaben ein wichtiger Leistungs-Träger sein.
In der schulischen Programmierung steht auch die Fähigkeit zum Lesen, Modifizieren und Erweitern von anderen Programmen hoch im Kurs. Vielfach wird das Erlernen mit dem Analysieren von einfachen Programmen begonnen. Besonders bei Programmiersprachen mit einem hohen Erklärungs-Bedarf bei der Quell-Text-Konstruktion bietet sich diese Arbeitsweise an.
Ein breiten Raum nimmt das umfangreiche (auch grenzüberschreitendes / Grenzenaustestendes) Prüfen der Programme ein. Hierbei lernen wir, wo die Probleme liegen, was andere von Programmen erwarten und welche Forderungen oft von Programmierern und Nutzern unterstellt werden. Hier ist das Kennenlernen beider Seiten sehr hilfreich.
Insgesamt ist es wichtig einen guten Programmier-Stil (Beachtung der üblichen Stil-Vorgaben, Gefühl für guten Stil entwickeln) aufzubauen.
Es ist dagegen zu Anfang zweitrangig, effiziente Programme zu schreiben. Zuerst einmal müssen sie funktionieren und die Forderungen (Aufgabenstellungen) erfüllen. Dann kommt der gute Programmier-Stil dazu. Erst hinterher sollte man sich darum kümmern, die Programme möglichst effektiv für die spezielle Hardware / Rechner-Situation zu erstellen. Wer die ersten beiden Stufen erfolgreich nimmt, wird die dritte auch erklimmen.
Zu vermeiden ist von Anfang an die Trick-Prammierung. Damit ist die umständliche oder hinterlistige Programmierung um ihrer selbst willen. Einige Freak's (Neerd's) müssen sich selbst irgendetwas beweisen. Sicher kann man damit auch ein guter – vielleicht sogar besonders erfolgreicher – Programmierer werden. Das wird aber nur (ganz) wenigen gelingen. Die Welt von heute arbeitet vernetzt und im Team.
Ist wirklich mal eine Trick-reiche Programmierung notwendig, dann ist sie sehr ausführlich zu kommentieren
Ein alleiniges Funktionieren eines Programms besagt noch nichts über die Software-Qualität aus.

1.0.1. außergewöhnliche Algorithmen - Algorithmen mit Pfiff

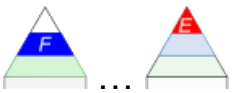


Hier finden sich einige Algorithmen, die mir beim Literatur-Studium oder bei Recherchen im Internet aufgefallen sind und die ungewöhnlich, überraschend oder manchmal auch unglaublich sind.

Berechnung / Ermittlung der Quadratwurzel einer natürlichen Zahl
Vorgabe: natürliche_Zahl
(1) Summe=1; Zahl=1; Anzahl=1
(2) FALLS Summe<natürliche_Zahl DANN (3) SONST (6)
(3) ERHÖHE Zahl um 2 (nächste ungerade Zahl)
(4) ERHÖHE Summe um Zahl
(5) ERHÖHE Anzahl um 1
(6) FALLS Summe=natürliche_Zahl DANN (8) SONST (7)
(7) FALLS Summe>natürliche_Zahl DANN (9) SONST (3)
(8) AUSGABE Anzahl (ist die Wurzel) STOPP
(9) AUSGABE "keine Wurzel im Bereich der natürlichen Zahlen!" STOPP

(Summiere die ungeraden Zahlen bis die Summe gleich der auszuwertenden Zahl ist; die Anzahl der Summanden ist die Wurzel (der (auszuwertenden) Zahl))

1.0.2. was bisher noch keinen Platz gefunden hat



deterministischer Algorithmus (Berechnung | a – b |)
(1) EINGABE a
(2) EINGABE b
(3) FALLS a>b DANN (4) SONST (5)
(4) d = a – b WEITER (6)
(5) d = b – a
(6) AUSGABE d
(7) STOPP

indeterministischer Algorithmus (Berechnung | a – b |
(1) EINGABE a
(2) EINGABE b
(3) WEITER (4) oder (5)
(4) d = a – b WEITER (6)
(5) d = b – a
(6) FALLS d<0 DANN (7) SONST (8)
(7) AUSGABE –d WEITER (9)
(8) AUSGABE d
(9) STOPP

Algorithmus führt auf verschiedenen – ev. zufällig ausgewählten – Wegen zum gleichen Ziel (er determiniert gleich!)

Umwandeln einer römischen Zahl in eine arabische:

Alphabet und Werte der römischen Zahlen:
Grundsymbole:
I → 1 X → 10 C → 100 M → 1000
dürfen maximal 3x hintereinander geschrieben werden
Hilfssymbole
V → 5 L → 50 D → 500
dürfen nur 1x geschrieben werden
Gesamtwert (arabische Zahl) → Addition der Werte der Symbole; steht ein Symbol mit einem kleineren Wert vor einem größeren, wird der Wert des kleinen Symbols subtrahiert

Zahlen-Systeme:

sedezimale Zaheln = hexadezimale Zahlen

vigesimale Zahlen sind die Zahlen zur Basis 20

sexagesimale Zahle sind Zahlen zur Basis 60

Algorithmus zur Umwandlung einer Dezimalzahl in ein beliebiges Zahlensystem

in JavaScript:

```
function decToAnyBase (dnum, b) {
// wandelt eine Dezimalzahl dnum in eine Zahl zur Basis b um
  var z = dnum;       // Integer
  var bpot = b;       // Integer
  var code = 0;       // Integer
  var ziffer = "";    // String
  var ergebnis = ""; // String

  if ((dnum > 0) && (b > 1)) {
    while (bpot <= z) bpot = bpot*b;

    do {
       bpot = bpot/b;
       code = Math.floor (z/bpot);
       codeplus55 = String.fromCharCode(code + 55);
       codeplus48 = String.fromCharCode(code + 48);
       ziffer = (code > 9) ? codeplus55 : codeplus48;
       ergebnis = ergebnis + ziffer;
       z = z - code*bpot;
    } while (bpot > 1);

    return ergebnis;
  }
  else return ("nix");
}
```

Q: http://www.henked.de/begriffe/algorithmus.htm

Sind Algorithmen frei von Diskreminierung / Rassismus / …?

Leider NEIN!

Algorithmic Bias (Algorithmische Vorurteile)

Biographie: Charles BABBAGE (1791 - 1871)

1.1. Problem-Lösen / Problem-Lösungsstrategien

Probleme im Sinne der Informatik

DRY-Prinzip

Versuch und Irrtum
Teile und herrsche

Das Fluss-Überquerungs-Problem

Ein Roboter soll 3 Objekte (Kohlkopf, Ziege und Wolf) über einen Fluss bringen. Solange der Roboter in der Nähe ist, dann er das übliche Fressen (Ziege den Kohl; Wolf die Ziege) verhindern.
Zur Überquerung des Flusses steht ein Boot mit begrenzter Transport-Kapazität zu Verfügung. Es kann neben dem Roboter, der rudern muss, immer nur 2 der 3 Objekte tragen.
Wie muss der Roboter nun die Überfahrten planen / durchführen, damit alle 3 Objekte unbeschadet auf der anderen Fluss-Seite ankommen?
Zuerst müssen wir uns ein verarbeitbares Modell des realen Problems erstellen, denn die Objekte bekommen wir ja nicht in den Computer rein und den Roboter ohne Vorüberlegungen einfach wirken zu lassen, wäre vielleicht auch nicht ganz optimal.
Schließlich braucht er nur einen Fehler zu machen und schon ist die Aufgabe nicht gelöst.
Man könnte nun z.B. für alle Objekte die Aufenthalts-Orte für alle denkbaren Situationen zusammentragen.
In der nebenstehenden Tabelle haben wir alle möglichen Situationen – die wir in der Informatik Zustände nennen – zusammengegestellt. Wir begnügen uns im weiteren Verlauf mit Zustands-Kennung aus den Groß-Buchstaben. Wobei für jedes Objekt immer der Ort seines Aufenthalts verschlüsselt ist. Das erste H oder D besagt etwas zum Auffenthalts-Ort des Roboter's.
Der zweite beschreibt den Aufenhalts-Ort des Kohlkopfes. Ziege und Wolf folgen entsprechend.
Weiterhin können wir einen **Start-Zustand** und einen **End-Zustand** festlegen. In unserem Fall gibt es ja von beiden nur jeweils einen. Alle anderen Zustände sind Übergangs-Zustände.

Roboter	Kohlkopf	Ziege	Wolf	Zustand
hier	hier	hier	hier	HHHH
hier	hier	hier	drüben	HHHD
hier	hier	drüben	hier	HHDH
hier	hier	drüben	drüben	HHDD
hier	drüben	hier	hier	HDHH
hier	drüben	hier	drüben	HDHD
hier	drüben	drüben	hier	HDDH
hier	drüben	drüben	drüben	HDDD
drüben	hier	hier	hier	DHHH
drüben	hier	hier	drüben	DHHD
drüben	hier	drüben	hier	DHDH
drüben	hier	drüben	drüben	DHDD
drüben	drüben	hier	hier	DDHH
drüben	drüben	hier	drüben	DDHD
drüben	drüben	drüben	hier	DDDH
drüben	drüben	drüben	drüben	DDDD

Roboter	Kohlkopf	Ziege	Wolf	Zustand
hier	**hier**	**hier**	**hier**	**HHHH**
hier	hier	hier	drüben	HHHD
hier	hier	drüben	hier	HHDH
hier	hier	drüben	drüben	HHDD
hier	drüben	hier	hier	HDHH
hier	drüben	hier	drüben	HDHD
hier	drüben	drüben	hier	HDDH
hier	drüben	drüben	drüben	HDDD
drüben	hier	hier	hier	DHHH
drüben	hier	hier	drüben	DHHD
drüben	hier	drüben	hier	DHDH
drüben	hier	drüben	drüben	DHDD
drüben	drüben	hier	hier	DDHH
drüben	drüben	hier	drüben	DDHD
drüben	drüben	drüben	hier	DDDH
drüben	**drüben**	**drüben**	**drüben**	**DDDD**

Als nächstes prüfen wir, welche Zustände nicht existieren dürfen, weil sie in einem Fressen ausarten.
So geht es nicht, das der Ziege und Wolf auf der einen Seite sind und der Roboter auf der anderen. Somit fallen die Zustände HHDD und DDHH weg.
Gleiches können wir für Kohl und Ziege aussagen. Damit gehen die Zustände HDDH und DHHD raus. Zuguterletzt fallen die Zustände raus, wo alle Objekte auf der einen Seite sind und der Roboter auf der anderen. Also fallen auch HDDD und DHHH weg.
Übrig bleibt ein Set aus 10 Zuständen.

Roboter	Kohlkopf	Ziege	Wolf	Zustand
hier	**hier**	**hier**	**hier**	**HHHH**
hier	hier	hier	drüben	HHHD
hier	hier	drüben	hier	HHDH
hier	hier	drüben	drüben	HHDD
hier	drüben	hier	hier	HDHH
hier	drüben	hier	drüben	HDHD
hier	drüben	drüben	hier	HDDH
hier	drüben	drüben	drüben	HDDD
drüben	hier	hier	hier	DHHH
drüben	hier	hier	drüben	DHHD
drüben	hier	drüben	hier	DHDH
drüben	hier	drüben	drüben	DHDD
drüben	drüben	hier	hier	DDHH
drüben	drüben	hier	drüben	DDHD
drüben	drüben	drüben	hier	DDDH
drüben	**drüben**	**drüben**	**drüben**	**DDDD**

Jetzt überlegen wir uns, wie wir von einem Zustand zu anderen kommen. Das Wechseln von einem Zustand zu einem anderen nennen wir Übergang.
Um z.B. vom **Start-Zustand HHHH** zum Zustand HHHD zu kommen, muss der W(olf) ins Boot. Da sich aber der Ort des Roboter nicht verändert, funktioniert das nicht.
Wir würden das so:

$$\text{HHHH} \xrightarrow{\text{W}} \text{Fehler}$$

notieren. Vor dem Pfeil steht der aktuelle Zustand. Über den Pfeil schreiben wir die Veränderungen (in der Praxis z.B. dann Eingaben).

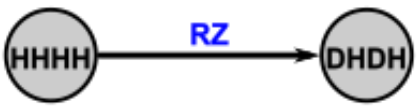
Roboter	Kohlkopf	Ziege	Wolf	Zustand
hier	**hier**	**hier**	**hier**	**HHHH**
hier	hier	hier	drüben	HHHD
hier	hier	drüben	hier	HHDH
hier	drüben	hier	hier	HDHH
hier	drüben	hier	drüben	HDHD
drüben	hier	drüben	hier	DHDH
drüben	hier	drüben	drüben	DHDD
drüben	drüben	hier	drüben	DDHD
drüben	drüben	drüben	hier	DDDH
drüben	**drüben**	**drüben**	**drüben**	**DDDD**

Der Pfeil deutet den eigentlichen Übergang an und es folgt der erreichte Zustand. Hier wäre es ein Fehler-Zustand. Dieser wird nicht immer mit betrachtet. Wir lassen ihn hier auch einfach weg. Schließlich suchen wir nur funktionierende Lösungen.
Nur der Roboter kann in unserem Modell rudern. Alle Übergänge ohne R(oboter) sind also hinfällig. Es müssen also immer Veränderungen (Eingaben) mit einem R sein.

Ein möglicher Übergang wäre:

$$\text{HHHH} \xrightarrow{\text{RZ}} \text{DHDH}$$

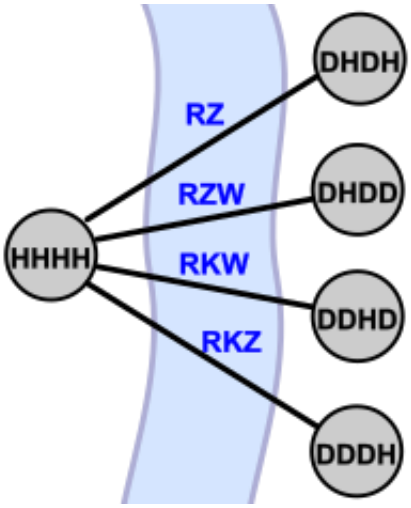
Dies bedeutet, dass man vom Zustand HHHH zum Zustand DHDH kommt, wenn man RZ durchführt (eingibt).

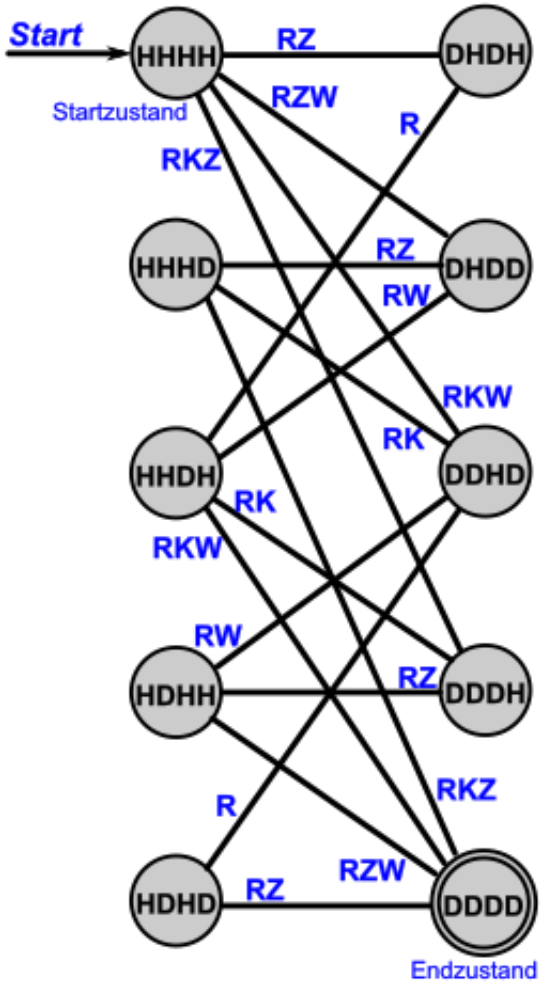


Das würde einem Boots-Transfer von Roboter mit der Ziege entsprechen. Da die Richtung für den Übergang eigentlich egal ist, der Boots-Transfer funktioniert ja gleichermaßen in beide Richtungen, geben wir beide Pfeile an oder verzichten auf die Pfeilspitzen.

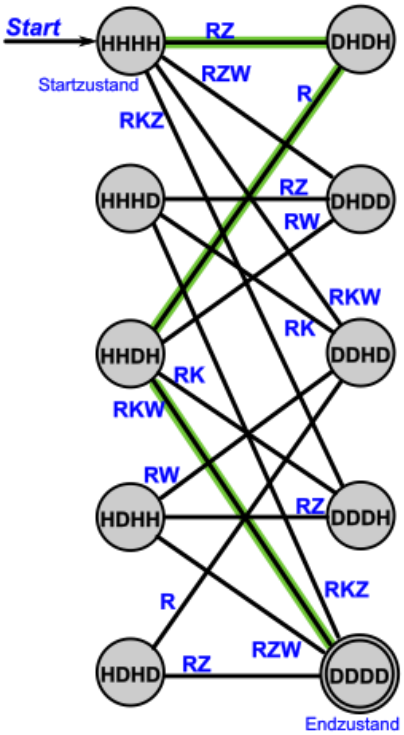
HHHH $\underset{RZ}{\overset{RZ}{\rightleftharpoons}}$ DHDH

Als nächste Schritte im Problemlöse-Prozess muss man nun alle möglichen Übergänge für den Zustand HHHH spezifizieren.
(Der Fluss ist nur zur Veranschaulichung hinterlegt und ist auch auf die Betrachtung durch den Roboter bezogen.)
Da wir den Endzustand DDDD noch nicht unter den Zuständen haben, müssen wir zwangsläufig weitermachen. Auch sonst sollte man in der Modellierung nicht stehen bleiben. Mit allen anderen Zuständen verfährt man deshalb genauso. Dadurch erhält man ein Schema, dass Graph genannt wird. Hier ist es der vollständige Graph zum Problem. Die Zustände sind die Knoten und die Übergänge sind die Kanten.





In dem Graphen suchen wir nun nach einer Verbindung zwischen HHHH und DDDD.
Ich habe mal oben angefangen und mit RZ den Zustand DHDH. Als nächstes wähle ich R (ist ja auch die einzige Alternative) und gelange zu HHDH. Beim systematischen Ausprobieren der Möglichkeiten von hier aus, stße ich auf den Übergang mit RKW zu DDDD – und bin am Ziel.



Das Protokoll für die Lösung des Problem's sieht also z.B. so aus:

HHHH $\xrightarrow{RZ}$ DHDH $\xrightarrow{R}$ HHDH $\xrightarrow{RKW}$ DDDD

Aufgaben:

1. *Finden Sie weitere Lösungen? Notieren Sie dann die Protolle!*
2. *Gibt es längere oder kürze Lösungen? Direkte Wiederholungen (also ein hin und her) sind ausgeschlossen.*
3. *Welches ist die längste Lösung ohne eine Schleife oder direkte Wiederholungen?*

Computer finden mit dem oben gezeigten Modell schnell eine Lösung. So könnte man das Problem z.B. mit PROLOG lösen lassen:

Eine echte Intelligenz müsste allerdings die Text-Aufgabe verstehen und selbst das Modell entwerfen. Das scheint doch sehr schwierig zu sein. Überlegen Sie sich einfach, wieviele Zusatz- und Hintergrund-Informationen Sie selber gebraucht und benutzt haben!

Bei Planungs-Problemen spricht man konzeptionell oft von Zustands-Raum, Übergängen und Kosten.

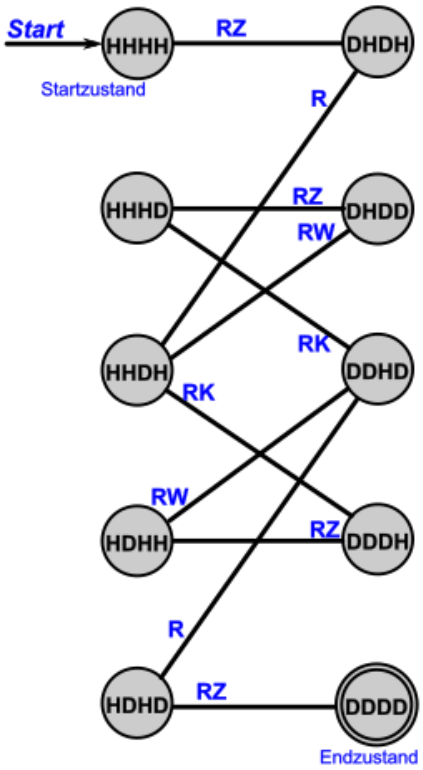
Definition(en): Zustand
Unter einem Zustand versteht man die (diskrete) Situation, in der sich ein System befindet.

Definition(en): Zustands-Raum
Der Zustands-Raum ist die menge der gültigen Zustände für ein System.

Definition(en): Übergang
Ein Übergang ist der Wechsel eines Systems von einem Zustand in den nächsten. Zumeist gibt es einen Auslöser (Eingabe, Bedingungswechsel, …) als Auslöser für den Wechsel.

Definition(en): Kosten
Kosten sind Konsequenzen aus den Übergängen. Man kann Kosten als Rechnungs-Posten für die Gesamt-Rechnung der Problemlösung betrachten.

Die ursprüngliche Version des Fluss-Überquerungs-Problem's geht davon aus, dass der Roboter nur immer jeweils ein Objekt rüberrudern kann. Wie sieht die Lösung hier aus? Gibt es überhaupt eine?
Für einen Lösungs-Versuch können wir die gleichen Zustände übernehmen. Nun sind allerdings nur noch ausgewählte Übergänge möglich.
Reichen diese aus, um einen Lösungs-Weg zu finden? Probieren Sie es selbst aus!



Aufgaben:

1. *Finden Sie eine Lösung für das ursprüngliche Fluss-Überquerungs-Problem mit immer nur einem zusätzlichen Objekt, dass der Roboter rüberrudern kann!*
2. *Gibt es mehrere Lösungen? Welche Lösung ist die kürzeste?*
3.

für die gehobenene Anspruchsebene:

4. *Überlegen Sie sich den Lösungs-Versuch für ein Überquerungs-Problem mit 4 Objekten in der Nahrungskette und jeweils maximal 2 zusätzlichen Objekten im Boot!*
5. *Gibt es eigentlich auch eine Lösung mit nur einem zusätzlich zu transportierenden Objekt? Begründen Sie Ihre Meinung anhand von Modellen od.ä.!*

Lösungen mit Spiel-Bäumen

Viele der bearbeiteten informatischen Probleme werden als Spiele verstanden. Die möglichen Spiel-Züge gleichen dabei einer Baum-Struktur. Daher der Begriff Spiel-Baum.

Null-Summen-Spiele
haben immer eine Lösung, bei der der eine Spieler (etwas) gewinnt und der andere Spieler (den gleichen Wert) verliert. Ein Remee ist möglich.

1.2. Darstellung von Algorithmen

Algorithmus muss vom Ersteller als auch vom Ausführer verstanden werden können
Menschen brauchen eine Dokumentations-Form für Algorithmen z.B. für Bücher usw.

beide müssen gemeinsame "Sprache" bzw. Darstellung finden
beides wird in der Informatik als Sprache verstanden

Definition(en): Sprache
Eine Sprache ist ein vereinbartes / definiertes System aus Zeichen / Signalen / Symbolen und den zugehörigen Regeln, das von den Kommunikanten (Ersteller u. Ausführer) zur Verständigung verwendet wird / wurde.

Später werden wir Sprache – vor allem für theoretische, informatische Zwecke noch genauer definieren.

> Die Mathematiker sind eine Art Franzosen;
> redet man mit ihnen,
> so übersetzen sie es in ihre Sprache,
> und dann ist es alsobald ganz etwas anderes.
> Johann Wolfgang VON GOETHE

Für die meisten Sprachen gibt es formale Notationen / Formulierung. Nur wenige Sprachen kennen keine Schrift oder Symbolik.

Jeder Algorithmus kann sicher in Text-Form aufgeschrieben werden. Wir sprechen von der **verbale**n **Formulierung / Notierung**. Selbst wenn man sich an bestimmte Sprach-Floskeln oder –Konventionen hält, ist ein Verständnis über die natürlichen Sprach-Barrieren hinweg meist sehr kompliziert. Wer noch nie einen kyrillischen Text gelesen hat, wird schon an den ungewöhnlichen Buchstaben scheitern. Noch komplizierter wird es, wenn der Text in asiatischen oder arabischen Sprachen abgefasst ist. Da sieht alles für uns Mittel-Europäer eher wie Krakeln aus.

Beispiele:

- Bedienungs-Anleitungen
- Rezepte (Speisen, Getränke)
- Spiel-Anleitungen (Spiel-Regeln)

Pseudo-Code

Der Pseudo-Code ist eine eingeschränkte natürliche Sprache. Meist ist sie schon an die favorisierte Programmiersprache angelehnt und mit entsprechenden Anweisungs-Wörtern aufgebaut. Guter Pseudo-Code benutzt aber allgemeine Formulierungen. Die Umsetzung eines Pseudo-Codes sollte in jede Programmiersprache (einer Sprach-Klasse ()) möglich sein.

es existiert eine Algorithm Definition Language (ADL)

übliche Schlüsselwörter:
falls … dann … [sonst …]
wiederhole …solange_bis …
weiter(_bei), springe_zu
start, ende, stop

Oft werden diese Schlüsselwörter in Großbuchstaben gesetzt und die anderen Zusatz-Informationen normal dazugeschrieben. Die Schlüsselwörter sind häufig relativ einfach in bestimmte Code-Strukturen umzusetzen. Dadurch hat der geübte Programmierer gute Orientierungs-Hilfen im Pseudo-Code.

Beispiele:
- Trainings-Programme
- Regie-Anweisungen
-

Pseudo-Code mit farblich hervorgehobenen Schlüssel-Wörtern (hier auch in Großbuchstaben):

Lösung (Berechnung Nullstelle(n)) für quadratische Gleichung		
(1)	**START**	Programm-Anfang
(2)	**LIES** a	Eingabe eines Wertes
(3)	**LIES** b	
(4)	**LIES** c	
(5)	**FALLS** a=0 **DANN** (6) **SONST** (8)	Alternative / Verzweigung
	Lösung des Fall: y = 0x² + bx +c = bx +c	**Bemerkung** also eigentlich nur lin. Fkt.
(6)	**DRUCKE** "**keine quadratische Gleichung**"	Ausgabe eines Textes
(7)	**WEITER** (18)	
	Lösung Normalfall	
(8)	p = b / a	Berechnung / Zuweisung
(9)	q = c / a	
(10)	d = p / 2 * p / 2 - q	
(11)	**FALLS** d>=0 **DANN** (12) **SONST** (17)	
	zwei Nullstellen	
(12)	wd = **wurzel**(d)	Benutzung einer Funktion
(13)	x1 = - p / 2 + wd	
(14)	x2 = - p / 2 - wd	
(15)	**DRUCKE** x1	
(16)	**DRUCKE** x2 WEITER (18)	
(17)	**DRUCKE** "**keine Nullstelle(n)**"	
(18)	**ENDE**	Programm-Ende

Pseudo-Code nur mit farblich hervorgehobenen Schlüssel-Wörtern:
(ein ähnlicher Algorithmus für das gleiche Problem)

Lösung (Berechnung Nullstelle(n)) für quadratische Gleichung		
(1)	**start**	
(2)	**lies** a, b, c	
(3)	**falls** a<>0 **dann** (6)	
(4)	**drucke** "**keine quadratische Gleichung**"	
(5)	**weiter** (13)	
(6)	p = b / a ; q = c / a	
(7)	d = p / 2 * p / 2 - q	
(8)	**falls** D<0 **dann** (12)	
(9)	wd = **wurzel**(d)	
(10)	x1 = - p / 2 + wd ; x2 = - p / 2 - wd	
(11)	**drucke** x1, x2 **weiter** (13)	
(12)	**drucke** "**keine Nullstelle(n)**"	
(13)	**ende**	

Quicksort(L: Liste):

WENN L eine leere Liste ist ODER nur ein Element enthält
DANN ist Liste sortiert → STOP
SONST

Divide (Herrsche):

Wähle (zufällig) ein Element Z aus der Liste L
Zerlege L Element-weise:

WENN Element E kleiner gleich Z
DANN hänge E an Liste L_1 an
SONST hänge E an die Liste L_2 an

Conquer (Teile):

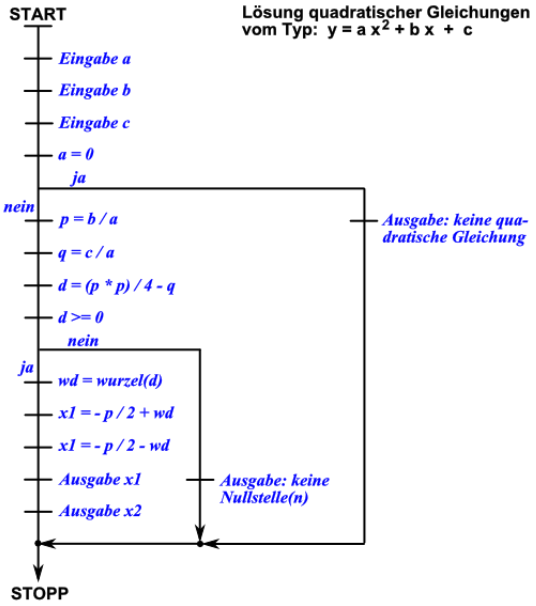
Quicksort(L1)
Quicksort(L2)

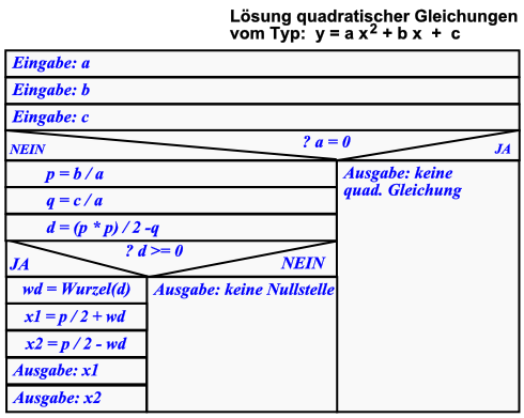
Merge (Mische):

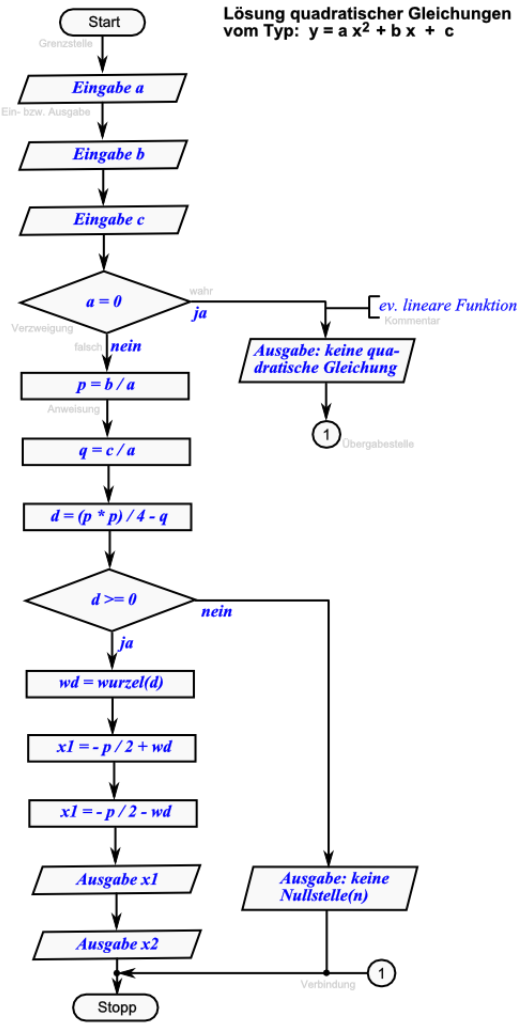
Hänge L1, Z und L2 so hintereinander als L oder Ergebnis-Liste E

Rückgabe L oder E

die hier nur abgebildeten Visualisierungen von Algorithmen werden genauer unter → 1.2.1. formale Visualisierung von Algorithmen besprochen







Programmiersprachen-Generationen

- **1. Generation** Maschinen-Sprachen; um 1950
- **2. Generation** Assembler-Sprachen ~ ab 1950
- **3. Generation** höhere Sprachen ~ ab 1954
- **4. Generation** Skript- und deklarative Sprachen; ~ ab 1960
- **5. Generation** Sprachen für "künstliche Intelligenz" ~ ab 1978

Übersetzungs-System / Ausführungs-Schemata

- **Interpreter-Sprachen** schrittweise Übersetzung (in Maschinen-Code) und Abarbeitung
- **Compiler-Sprachen** vollständige Übersetzung (in Maschinen-Code) und unabhängige Abarbeitung

dazu später mehr (→ x.y. Übersetzung von (höhreren) Programmiersprachen in Maschinencode)

Maschinensprache (Maschinen-Code, nativer Code, Binär-Code, Programm-Datei, EXE, …)

Das einzige was Computer wirklich verstehen sind Nullen und Einsen – genauer gesagt eigentlich die Signale "keine Spannung" und "Spannung". I.A. steht die Null eben für keine vorhandene Spannung und die Eins für eine bestimmte Spannung.
In manchen Systemen ist auch der Null eine bestimmte – natürlich andere – Spannung zugeordnet.
Aus dem Wirrwarr von Nullen und Einsen rechts wird wohl kaum jemand etwas sinnvolles herauslesen können. Wir brauchen eine Menschen-"würdigere" Darstellung der Daten.

01010111	
11010111	
10100000	
01111010	
01110100	
10010011	
10111111	
01010010	

Definition(en): Maschinensprache
Die Maschinensprache ist die Programmiersprache, in der die Anweisungen direkt an und für den Prozessor notiert und auch direkt ausgeführt werden.
Die Maschinensprache eines Prozessor(-Systems) ist die Gesamtheit seiner Binär-codierten Instruktionen. Programme zur Abarbeitung in einem System müssen immer in Form von Maschinen-Instruktionen (Binär-Code) vorliegen.

Eine etwas bessere Form ist die Darstellung der Bytes als Hexadezimal-Zahl. Bei ihr werden immer 4 Bit's zu einem Hexadezimal-Ziffer zusammengefasst.
Hexadezimal-Zahlen haben die Zahlen-Basis 16. Zur Symbolisierung der "Ziffern" werden die Ziffern 0 bis 9 und dann noch die Buchstaben A bis F verwendet.
Bei acht Bit's werden es dann entsprechend zwei Hexadezimal-Ziffern.
Die Inhalte der einzelnen Zellen sind jetzt deutlich besser lesbar, sie bleiben trotzdem kryptisch.

57	
D7	
A0	
7A	
74	
93	
BF	
52	

Einem "normalen" Nutzer kann man kaum zumuten zu wissen, was bei "57" (sprich fünf sieben, statt siebenundfünfzig, um eine Verwechslung mit den dezimalen Zahlen zu vermeiden) passiert.

Maschinen-(orientierte) Programmierung

- **Maschinensprachen**
- **Assemblersprachen** Assembler
- **einfache Programmiersprachen** C

Einzelnen oder mehreren Hexadezimal-Zahlen werden Menschen-lesbaren Befehlen zugeordnet. Jetzt lässt sich eine Art Programm-Code erkennen. Trotzdem ist ein "Normal"-Nutzer immer noch deutlich überfordert. Die Programmierung erfolgt quasi direkt auf dem Prozessor und ist damit auch sehr primitiv.

`ld A,00`	
`add A,01`	
`mov A,003A`	

Programm-Beispiel "Hello World" in Assembler

```
DATA SEGMENT                ;- Beginn des Datensegments
Meldung db "Hello World"    ;- Die Zeichenkette "Hello World"
        db "$"              ;- Zeichen, das INT 21h (s.u) als Ende der Zeichenkette erkennt
DATA ENDS                   ;- Ende des Datensegments
                            ;
CODE SEGMENT                ;- Beginn des Codesegments
ASSUME CS:CODE,DS:DATA      ;- dem Assembler die vorgesehenen Segmente und Segmentregister
                            ;  mitteilen
Anfang:                     ;- Einsprung-Label fuer den Anfang des Programms
   mov ax, DATA             ;- Adresse des Datensegments in das Register "AX" laden
   mov ds, ax               ;  In das Segmentregister "DS" uebertragen (in das DS Register
                            ;  kann nicht direkt geladen werden)
   mov dx, offset Meldung   ;- die zum Datensegment relative Adresse des Textes in das "DX"
                            ; Datenregister laden
                            ;  die vollstaendige Adresse von "Meldung" befindet sich nun im
                            ;  Registerpaar DS:DX
   mov ah, 09h              ;- die Unterfunktion 9 des Betriebssysteminterrupts 21h
                            ;  auswaehlen
   int 21h                  ;- den Betriebssysteminterrupt 21h aufrufen (hier erfolgt die
                            ;  Ausgabe des Textes am Schirm)
   mov ax, 4C00h            ;- die Unterfunktion 4Ch (Programmbeendigung) des
                            ;  Betriebssysteminterrupts 21h festlegen
   int 21h                  ;- diesen Befehl ausfuehren, damit wird die Kontrolle wieder an
                            ;  das Betriebssystem zurueckgegeben
CODE ENDS                   ;- Ende des Codesegments
                            ;
END Anfang                  ;- dem Assembler- und Linkprogramm den Programm-Einsprunglabel
                            ;  mitteilen
                            ;- dadurch erhaelt der Befehlszaehler "PC" beim Aufruf des
                            ;  Programmes diesen Wert
```

Q: https://de.wikipedia.org/wiki/Assemblersprache (ganz leicht geändert)

Die kleingeschriebenen Befehle, wie mov und int sind die eigentlichen Anweisungen für den Prozessor. Der Rest des Quell-Textes ist Beiwerk für den Menschen. Dadurch werden Daten und deren Strukturen abgebildet sowie die Lesbarkeit des Quell-Textes erhöht.

Definition(en): Programm
Ein Programm ist ein Algorithmus in einer Sprache, die eine Datenverarbeitungsanlage / ein Informations-verarbeitendes System direkt oder indirekt abarbeiten kann.
Ein Programm ist eine Folge von Anweisungen, die zu einer Programmier-Sprache gehören und eine Umschreibung eines Algorithmus darstellen.
Ein Programm ist eine Folge von Computer-verständlichen Befehlen, die eine bestimmte Aufgabe lösen.
Ein Programm ist eine Software, die auf einem Computer eine bestimmte Aufgabe erfüllt.
Ein Programm ist eine logisch gegliederte Einheit von Anweisungen, die in Form einer Programmiersprache – zur Lösung einer Aufgabe – für die Ausführung auf einem Computer formuliert wurden.
Ein Programm ist eine Folge von Befehlen, die einer Informations-verarbeitenden Maschine mitteilen, was sie wann und wie zu machen hat (, um eine bestimmte Aufgabe zu lösen).
Die Befehls-Folge nach der ein Computer arbeitet, nennt man ein Programm.

Anforderungs-Kriterien an ein Programm

- **Korrektheit** — die im Entwurf / Pflichtenheft / Aufgabenstellung beschriebenen Leistungen / Vorgaben müssen exakt erfüllt werden
  - Vermeidung syntaktischer Fehler
  - Vermeidung semantischer Fehler
  - **partielle Korrektheit** — falls der Algorithmus anhält, dann ist das Ergebnis eine gültige Lösung
  - **totale Korrektheit** — der Algorithmus hält immer an und liefert dann immer eine gültige Lösung
- **Robustheit** — Absicherung des Programms gegen variable Hard- und Software-Umgebungen und gegen (gewollte oder nicht gewollte) unreguläre Nutzer-Aktivitäten
- **Wartbarkeit** — Quelltext muss dokumentiert und später oder von anderen Personen lesbar und änderbar sein
- **Effizienz**
  Performanz — die begrenzten Ressourcen (Laufzeit, Speicher (Programmcode, Daten)) sollen nur soweit genutzt werden, wie es notwendig ist

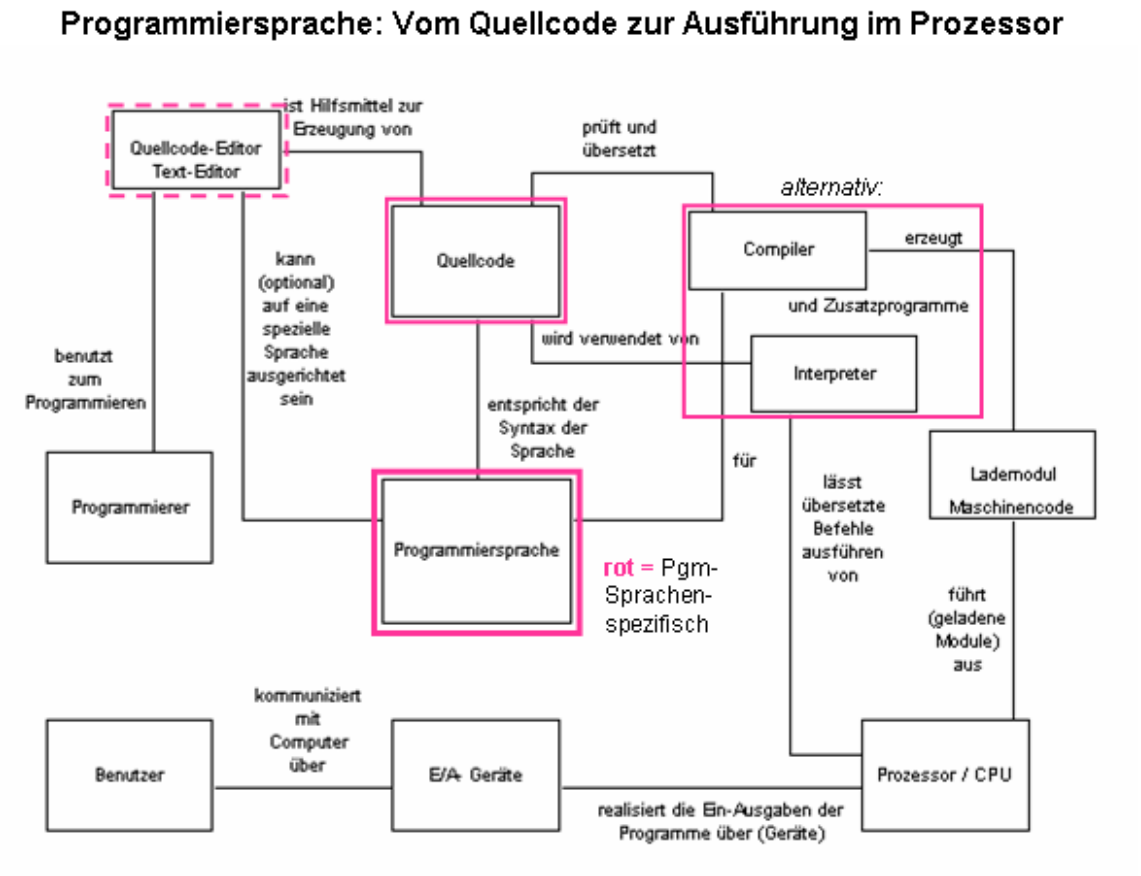
Test auf Robustheit ist i.A. der Zeit-aufwändigste

Bereich der Wartbarkeit wird vor allem von Anfängern unterschätzt
Typisch für Anfänger ist die Verwendung von x, y, a, b usw. als schnelle Variablen-Namen. Meist werden diese Variablen dann auch noch für die unterschiedlichsten Zwecke, z.T. im gleichen Programm benutzt. Für jemanden Außenstehenden ist es schwer die Intensionen eines solchen Programmierers nachzuvollziehen. Fehler finden ist dann extrem schwer.

Bei Erweiterungen oder der Übernahme der Programme werden die Variablen logischerweise weiter verwendet. Jetzt kann es zu Doppel-Belegungen und Widersprüchen kommen. Bei Verwendung sprechender Variablen fällt die Doppel-Nutzung meist deutlich schneller auf, da hier Namen viel seltener sind.

Definition(en): programmieren
Unter Programmieren versteht man die schöpferische oder automatische Tätigkeit der Erstellen von Programmen / Algorithmen für eine informatisches (oder technisches) System.

umgangssprachlich werden viele Begriffe gleichbedeutend verwendet
das folgende Schema stellt die Begriffe in geordneten Zusammenhängen dar



Begriffszusammenhänge zum Sachverhalt "Programmiersprache"
Q: de.wikipedia.org (VÖRBY)

Definition(en): Programmiersprache
Eine Programmiersprache ist eine künstliche (bzw. formale) Sprache, die zur Formulierung eines Algorithmus und der zugehörigen Daten dient.
Eine Programmiersprache ist eine formale Sprache, die dazu dient einen Algorithmus sowohl für einen Computer als auch für einen Programmierer / Anwender / Nutzer lesbar darzustellen.
Eine Programmiersprache ist eine (formale) Sprache, deren Syntax und Semantik exakt definiert sind.

Anforderungen an eine Programmiersprache

- **Universalität** — es muss die Formulierung beliebiger / einer großen Gruppe von Algorithmen möglich sein
- **automatische Analysierbarkeit** — mindestens ein Programm (Interpreter / Compiler) muss den Quell-Text in Maschinen-Code oder eine andere automatisch analysierbare Sprache umsetzen
- **Eindeutigkeit der Anweisungen**

Definition(en): Syntax
Der Syntax ist die Definition der zulässigen Symbole und Wörter sowie deren Anordnungen, die in einer Sprache genutzt werden können.

Definition(en): Semantik
Die Semantik ist die Definition der Bedeutungen der (zulässigen) Symbole und Wörter sowie deren Anordnungen.

		Semantik	
		semantisch richtig / wahr	**semantisch falsch**
Syntax	**syntaktisch richtig / wahr (exakt)**	1 + 2 = 3	1 + 2 = 2
	syntaktisch falsch	eins + two = 3	one + 2 = IIII

Es kommt auf Groß- und Kleinschreibung, Zeichensetzung und Betonung an!

Liebe Schwiegermutter, ich mag Dich Ungeheuer gern!
Liebe Schwiegermutter, ich mag Dich ungeheuer gern!

Ich komme, nicht erschießen!
Ich komme nicht, erschießen!

Ich komme, nicht hängen!
Ich komme nicht, hängen!

Hängt ihn nicht, laufen lassen!
Hängt ihn, nicht laufen lassen!

Was willst du schon wieder?
Was, willst du schon wieder?

Der Mann sagt, die Frau kann nicht Auto fahren.
Der Mann, sagt die Frau, kann nicht Auto fahren.

Sabine versprach ihrem Vater, einen Brief zu schreiben.
Sabine versprach, ihrem Vater einen Brief zu schreiben.

Herr Schmidt, der Pfarrer, und ich spielten Golf.
Herr Schmidt, der Pfarrer und ich spielten Golf.

Komm wir essen, Opa!
Komm, wir essen Opa!

Komm, wir essen Tante Erna.
Komm, wir essen, Tante Erna.

Kinder-Spezial: 2 Erwachsene essen 1 Kind gratis
Kinder-Spezial: 2 Erwachsene essen; 1 Kind gratis

Die Hochzeit ist geplatzt: Er wollte sie nicht.
Die Hochzeit ist geplatzt: Er wollte, sie nicht.

Der redliche Mensch denkt an sich selbst zuletzt.
Der redliche Mensch denkt an sich, selbst zuletzt.

Du hast den schönsten Hintern weit und breit.
Du hast den schönsten Hintern, weit und breit.

Schülern mit negativen Attributen helfen.

Q: u.a.: https://www.editionblaes.de/wie-ein-komma-den-sinn-eines-satzes-veraendert/

höhere Programmiersprachen

auch Hochsprachen genannt
meist Hardware-unabhängig, es kann aber für unterschiedliche Hardware auch unterschiedliche Umsetzungen (Implementierungen) geben; jede Umsetzung ist auch für sich unterschiedlich Fehler-behaftet
Problem / Algorithmus wird in einem eigenem Bedeutungs-Modell relativ abstrakt umgesetzt
meist für Mensch gut lesbar / verständlich

Definition(en): höhere Programmiersprache
Höhere Programmiersprachen sind solche Programmiersprachen, bei denen die Lesbarkeit und Verständlichkeit besonders für den Anwender / Nutzer / Programmierer im Vordergrund steht.

Programm-Beispiel "Hello World" in PASCAL

```
program hello;

const text='Hello World';

begin
  writeln(text);
end.
```

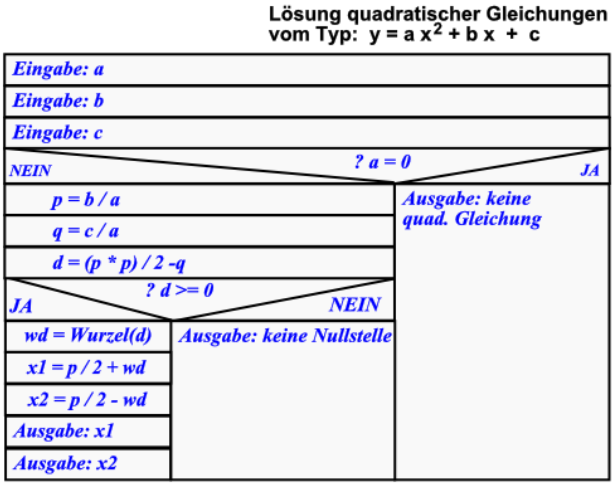
Programm-Beispiel "Hello World" in JAVA

```
class hello {

  public static void main(String[ ] args){
    String text = "Hello world";
    System.out.println(text);
  }
}
```

Es gibt auch noch einfachere Versionen. Um aber etwas mehr von der jeweiligen Programmier-Sprache aufzuzeigen, haben wir hier etwas längere Konstruktionen genutzt.

Problem-orientierte / höhere Programmierung

- **einfache / Maschinen-nahe Programmiersprachen** C
- **Problem-orientierte Sprachen**
- **Objekt-orientierte Sprachen** Java



Beispiel: Lösung einer quadratischen Gleichung mit Python

```
import math

print("Lösung von quadratischen Gleichungen vom Typ:  y = a x² + b x + c")
a=eval(input("Geben Sie den Faktor für x² ein (a = ): "))
b=eval(input("Geben Sie den Faktor für x ein  (b = ): "))
c=eval(input("Geben Sie die Konstante c ein   (c = ): "))
print()
if a==0:
    print("keine quadratische Gleichung!")
else:
    p=b/a
    q=c/a
    d=p/2*p/2-q
    if d>=0:
        wd=math.sqrt(d)
        x1=-p/2+wd
        x2=-p/2-wd
        print("1. Nullstelle bei (x = ): ",x1)
        print("2. Nullstelle bei (x = ): ",x2)
    else:
        print("keine Nullstelle vorhanden")
```

```
>>>
Lösung von quadratischen Gleichungen vom Typ:  y = a x² + b x + c
Geben Sie den Faktor für x² ein (a = ): 3
Geben Sie den Faktor für x ein  (b = ): 2
Geben Sie die Konstante c ein   (c = ): -5

1. Nullstelle bei (x = ):  1.0000000000000002
2. Nullstelle bei (x = ):  -1.6666666666666667
>>>
```

Quicksort in Haskell / Erlang:

```
quicksort :: Ord a => [a] -> [a]
quicksort [ ] = [ ]
quicksort (e:es) = quicksort [ x | x <- es, x<= e] ++ [e]
                   ++ quicksort [ x | x <- es, x > e]
```

Definition(en): informatische Modellierung
Unter informatischer Modellierung versteht man die Umsetzung von Sachverhalten / Objekten / Prozessen in ein Programm oder Programm-System.

Biographie: Grace Murray HOPPER (1906 - 1992)

machte sich für die Entwicklung verständlicher Programmiersprachen stark

Spitzname "Grandma COBOL"

> Vor dem Zweiten Weltkrieg war das Leben einfach.
> Danach hatten wir (Computer-)Systeme.

von ihr stammt auch der erste dokumentierte Eintrag in einem Logbuch über einen "Bug"
hier war es eine Motte, die für den Ausfall eines Relais verantwortlich war und eine beachtliche Störung des Rechner zur Folge hatte

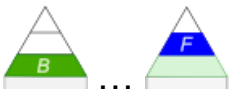
> Es ist immer einfacher,
> [hinterher] um eine Entschuldigung zu bitten,
> als [vorher dafür] eine Genehmigung zu bekommen.

sie ist auch "mitverantwortlich" für viele Probleme bezüglich des Millenium-Bug's
die Programmierer hatte – wegen des sehr teuren Arbeitsspeicher zu der Zeit – die Jahres-Zahlen auf die letzten zwei Ziffern beschränkt
sie hatte nicht erwartet, das ihre ursprünglichen Routinen usw. solange (bis über 1999 hinaus) Bestand haben würden

> Der gefährlichste Satz einer Sprache ist:
> "Das haben wir schon immer so gemacht."

> Im Zweifelsfall – Tu es.

1.2.1. formale Visualisierung von Algorithmen



Heute arbeiten an Programm-Entwicklungen meist recht viele Programmierer. In anderen Konstelationen arbeiten Programmierer aus mehreren Ländern zeitversetzt – quasi rund um die Uhr – an einem projekt. Kaum einer hat da zu jeder Zeit den vollen Überblick über alle Details und Änderungen.
Für die Software-Entwickler, aber auch für Projekt-Manager und Software-Kunden (z.B. die Tester) sind verschiedene Kommunikations- und Dokumentations-Mittel notwendig, um den Überblick zu behalten und Doppel-Entwicklungen zu vermeiden. Für die Entwickler sind das zuerst einmal grobe und feinere Pläne zu den Programmteilen und benutzten Algorithmen.
Derzeit ist die am Weitesten verbreitete Dokumentations-Art UML. Diese stellen wir weiter hinten bei der Objekt-orientierten Programmnierung (OOP) vor. Sie geht wesentlich über die einfache Algorithmen-Darstellung hinaus.
Bleiben wir zuerst bei der Darstellung der klassischen Algorithmen, wie sie aber auch wieder in den Methoden der OOP eine Rolle spielen.
Praktisch geht es um die graphisch orientierte Darstellung des Programmablaufs. In guten Visualisierungen nimmt die inhaltliche Unterlegung – also die Semantik – eine sehr große Rolle ein. Syntaktische Strukturen aus der später zu nutzenden Programmiersprache sollten nur sehr sparsam eingesetzt werden. Gute Visualisierungen benutzen nur die üblichen Programm-Strukturen, wie z.B. Sequenzen, Verzweigungen und Schleifen in sehr allgemeinen Formen. Eigentliches Ziel ist, den Algorithmus so darzustellen, dass er in jede beliebige – oder besser gesagt, jede geeignete – Programmiersprache umgesetzt werden kann.

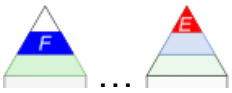
Möglichkeiten der Algorithmen-Visualisierung

- **Programm-Ablauf-Plan PAP** — relativ alte, Symbol-orientierte Form der Algorithmen-Darstellung
- **Programmlinien-Methode** — Programm-Ablauf ist ein einfacher Graph (Linie) mit semantischen Details
- **Struktogramme** — Block-orientierte Algorithmen-Darstellung, besonders für Bottom-up- und Top-down-Entwurfstechniken geeignet
- **UML-Diagramme** — Objekt- und Ablauf-orientierte Visualisierung mit der Möglichkeit zur automtischen Quell-Code-Generierung

Alle Visualisierungs-Möglichkeiten sind Sinnbild-Methoden. Für die Programmlinien-Methode gilt das praktisch nur noch für die Ablauflinie.
Besonders bei komplexen Programmen, Programmteilen oder Algorithmen sind die aufgezeigten Visualisierungen wichtige Programm-Entwurfs-Methoden.
Anfänger neigen dazu, die Algorithmen-Beschreibung außerhalb von Programmier-Systemen als lästig und ungeeignet anzusehen. Aus ihrer Sicht muss zuviel geschrieben und gezeichnet werden. Sie sehen die Visualisierung als doppelte Arbeit. Für das Arbeiten mit Algorithmen-Visualisierungen spricht aber, dass man irgendwann bei größeren Projekten nicht mehr ohne sie auskommt. Erst dann damit anzufangen ist schwierig. Da ist ein Üben mit kleinen, einfachen Algorithmen – wie wir sie in den schulischen Kursen verwenden – viel sinnvoller.
Die gedankliche Auseinandersetzung mit dem Algorithmus hilft auch viele unnötige Programmier-Arbeit zu vermeiden. Welche Schleife ist geeigneter, nehme ich zuerst den einen Teil in die Verzweigung oder den anderen? Kann ich Abläufe verkürzen? Lassen sich Algo-

rithmen-Teile auslagern? All solche Fragen sind besser vor einem wilden Eintippen zu klären.
Ein weiteres Argument für eine ordentliche Dokumentation ist die Wiederverwendbarkeit von Programmen. Leider spielt dieses in der Schule kaum eine Rolle. In der großen Software-Welt können so Millionen gespart werden. So manche Firma mit verteilten Einrichtungen, Teams oder Projekten musste das schon schmerzlich erleben. Erst später mit Dokumentationen anzufangen, ist dann ein sehr schwerer Weg für die Firma und die Mitarbeiter.
Wer schon mal einen älteren – nicht trivialen – Programm-Code ändern, erweitern oder auch nur verstehen sollte, der weiss wovon ich rede. Es dauert oft länger den undokumentierten Algorithmus zu verstehen, als das Problem neu zu programmieren. Im Ergebnis ist das Schade um die frühere Arbeit und in der praktischen Wirtschaft bleibt ein vermeidbarer Verlust übrig, den ein Kunde indirekt bezahlen muss, oder der ein Produkt unnötig teuerer macht.
Historisch betrachtet kam den Visualisierungen in den Kinderjahren der Datenverarbeitung eine sehr große Bedeutung zu. Eine programmierbare Rechen-Maschine zur erfolgreichen Arbeit zu bringen, war ohne guten Programm-Ablauf-Plan kaum möglich. Man hatte gar nicht die Ressourcen stundenlang zu probieren. Geräte waren teuer und somit sehr selten. Die Rechen-Zeiten wurde genau festgelegt und waren auch nur sehr begrenzt verfügbar.
Da war vorlaufende Planung alles. Dazu kam, dass man oft gar nicht selbst an die Maschine kam. Die dort arbeitenden "Programmierer" waren die alleinigen Herrscher. Der einfache Nutzer musste also in geeigneter Form mit dem Programmierer kommunizieren. Dazu wurden – mangels Programmiersprachen-Kenntnissen auf der Seite der Nutzer – Programm-Ablauf-Pläne benutzt.
Die neuen Möglichkeiten – vor allem Struktogramme und UML-Diagramme – gleich am Computer zu erstellen, eröffnen ganz neue Möglichkeiten. Es gibt immer mehr Programmier-Systeme, in denen gleich aus den Visualisierungen Quell-Texte – z.T. auch in verschiedenen Programmier-Sprachen – erzeugen lassen. Der moderne Programmierer kümmert sich dann mehr um die Fein- und Optimierungs-Arbeit.

1.2.1.1. Programm-Ablauf-Pläne (PAPs)



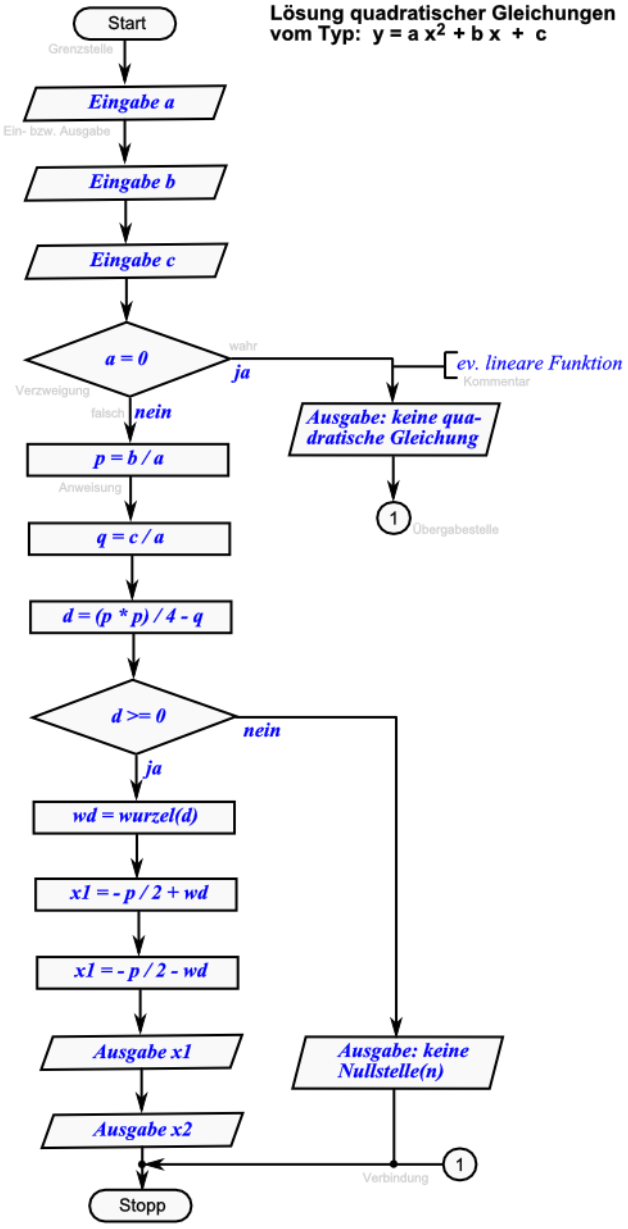
Programm-Ablauf-Pläne (PAP) prägten die ersten Jahre der Software-Entwicklung.

nach DIN 66001
außer Programm-Ablaufplänen (PAPs) noch Datenflusspläne spezifiziert

früher deutlich mehr Symbole
heute stark eingeschränkt und verallgemeinert

Vorteile
unabhängig von späterer Programmiersprache
guter Überblick; verständliche Veranschaulichung
zeitlicher / logische Ablauf ersichtlich
auch Sprünge / Sprung-Anweisungen darstellbar

Nachteile
sehr platzaufwendig
hoher Zeichen-Aufwand
ermöglichen Spagetti-Code's (viele (wilde) Sprünge)

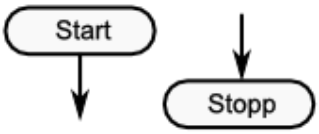


gut für Bottom-up-Entwicklungen geeignet
bei diesen beginnt man bei problematischen Einzel-Anweisungen und baut dann nach und nach ein immer größer werdendes Programm darum

vom Kleinen zum Großen

zugelassene Algorithmen-Strukturen (mit Sinnbildern nach DIN)

- **Grenzstelle**
  Anfang und Ende
  (terminal, interrupt)
  Rechteck mit abgerundeten Ecken und Start- bzw. Stopp-Beschriftung



- **(Wert-)Zuweisung**
  Berechnungen
  Operationen, …
  (process)
  Rechteck mit Berechnung, Beschreibung der Operation oder auch ganzer Aufgaben-Blöcke

  - **Eingabe**
    (input)
    Parallelogramm
    meist mit "Eingabe" beschriftet

  - **Ausgabe**
    (output)
    Parallelogramm
    meist mit "Ausgabe" beschriftet

- **Sequenz**
  ()
  Pfeil- bzw. Linien-verbundene Aneinanderreihung zugelassener Symbole

- **Verzweigung**
  ()
  Rhombus (Raute)
  Beschriftung mit Vergleich od.ä. die Rhombusspitzen sind die Ausgangspunkte für die Antwort-Linien (Fall-beschriftet)

- **Unterablauf**
  Unterprogramm
  Funktionen
  Prozeduren
  (predefined process)
  Rechteck mit doppelten Rahmenlinien rechts und links

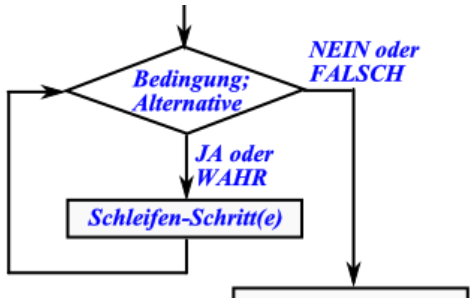
- **Anknüpfungspunkte**
  Übergangsstelle
  (connector)
  Kreise, z.B. mit Nummern

- **Bemerkung**
  Kommentar
  (comment, annotation)

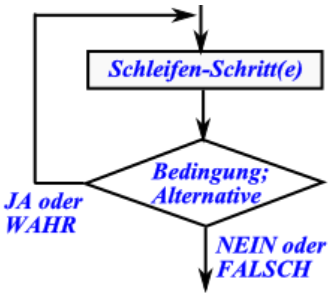
zusammengesetzte Algorithmen-Strukturen

- **Zyklus**
  Schleife

kein eigenständiges Symbol; wird durch Linienzüge dargestellt; am Anfang oder am Ende muss Verzweigung im Ablauf auftauchen (Schleifen-Abbruch)



Kopf-gesteuert; abweisend



Fuß-gesteuert; nicht abweisend

Arbeitsregeln für die Erstellung von Programmablaufplänen (PAPs)

- **Allgemeingültigkeit**
  - PAP sollte keine Programmiersprachen-spezifischen Befehle usw. enthalten
  - die dargestellten Abläufe und Logiken sollten leicht zu verfolgen und zu verstehen sein
  - das PAP sollte in jede beliebige (imperative) Programmiersprache umgesetzt werden können
- **Deklaration**
  - neben den Abläufen sind bestimmte Vorgaben / Festlegungungen (Zahlen-Typ, Text-Längen, …) zu machen
  - die Deklarationen sind vor den Verwendung zu notieren (optimalerweise in den ersten / dem ersten Block)
- **Exklusivität**
  - jede einzelne Schritt (Eingabe, Ausgaben, Anweisung, …) erhält ein eigenes Symbol lt. DIN / EN
  - es darf keine Zusammenfassung von mehreren (auch nicht bei gleichartigen) Anweisungen / Schritten erfolgen!
  - zu jedem PAP gehört ein Name

Programme zur Programmablaufplan-Erstellung:

- **PapDesigner** Lizenz: eigene; praktisch: CC-BY-NC-ND

  http://friedrich-folkmann.de/papdesigner/Hauptseite.html

- **Dia** Lizenz: opensource

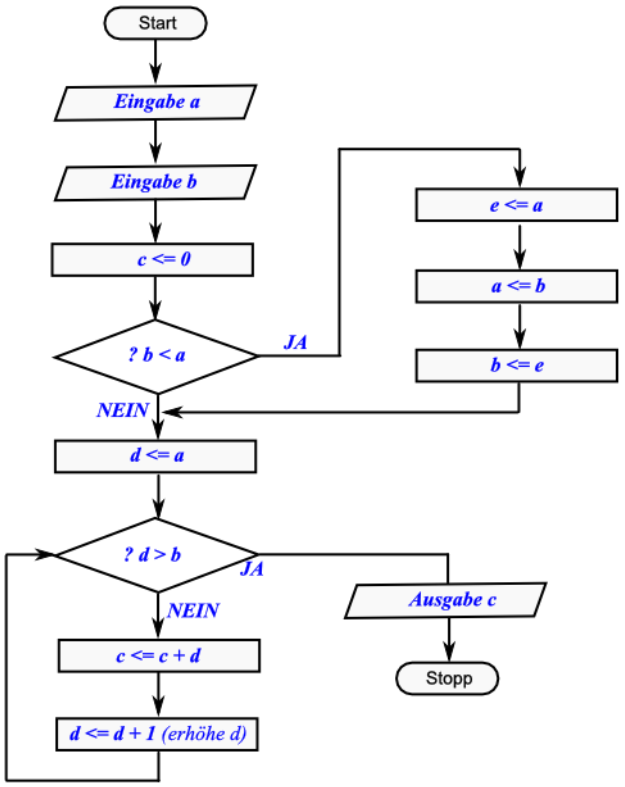
  auch als portable App verfügbar

  http://dia-installer.de/

-

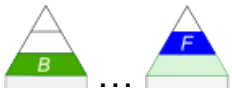
Aufgaben:

1. *Stellen Sie die Ermittlung des Minimums von zwei Zahlen als PAP dar!*
2. *Was macht der nebenstehende – als PAP – dargestellte Algorithmus? Erklären Sie die Abläufe genau! Benennen Sie den Algorithmus!*
3. *Variieren Sie den PAP so ab, dass die Variablen sprechend sind!*
4. *Erstellen Sie ein PAP, dess das Produkt einer Reihe von Zahlen von x bis y berechnet!*



5. *Erstellen Sie je ein PAP zu folgenden Algorithmen!*
   a) *Berechnung einer KELVIN- aus einer Grad-CELSIUS-Temperatur*
   b) *Erraten einer Farbe aus dem klassischen Regenbogen (rot, orange, gelb, grün, türkis, blau, violett), die sich der Nutzer ausgewählt hat*
   c) *Berechnung des Multiplikator (f) und den Rest (r) für die ganzzahlige Division (im Bereich der Natürlichen Zahlen) ($y = f x + r$)*
   d) *Prüfung, ob es sich bei einer natürlichen Zahl um eine Primzahl handelt*

1.2.1.2. Programmlinien-Methode



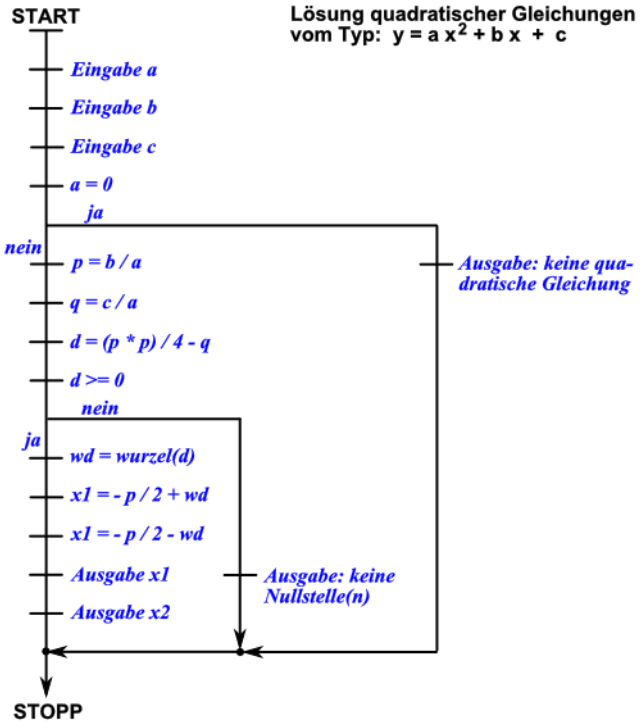
Eigentlich ist diese Methode einer der effektivsten. In der Praxis ist sie eher eine Skizzen-Methode für Programmierer. In der Schule hat die Besprechung diese Methode eher informativen Charakter.

auch Strukturlinien-Diagramme genannt, typische Ablauf-Diagramme

Ablauflinien als Pfeile
in regelmäßigen Abständigen die Arbeitsschritte abgetragen

praktisch Symbolfrei und deshalb auch recht übersichtlich und vom Verlauf leicht nachvollziehbar; sehr kompakt

weniger Verwirrung durch Symbole
mehr Ablauf- / Verlaufs-orientiert
dicht am Programmier / Algorithmus



Vorteile
unabhängig von späterer Programmiersprache
sehr einfache Symbolik
guter Überblick; verständliche Veranschaulichung
zeitlicher / logische Ablauf ersichtlich
auch Sprünge / Sprung-Anweisungen darstellbar
weniger Platz-aufwendig als PAP; kompakter
schneller und leichter zeichenbar (praktisch, wie Freihand-Skizze)
selbst Ergänzungen lassen sich als Aufweitungen gut einbauen

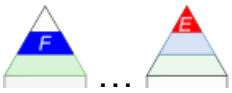
Nachteile
durch fehlende Symbole Struktur-Orientierung schwieriger
je nach Verwendungs-Zweck und Projekt-Fortschritt können kleine oder komplexe Arbeitschritte an die Führungs-Linien geschrieben werden. Selbst eine Mischung unterschiedlicher Detail-Tiefen ist gut möglich.
ermöglichen Spagetti-Code's (viele Sprünge möglich und nicht als nachteilig ersichtlich)
Programmstrukturen (Funktionen, Prozeduren, Unterprogramme) müssen separat abgeleitet werden

Aufgaben:

1. *Erstellen Sie ein Programmlinien-Diagramm für die Berechnung einer Kubikzahl!*
2. *Setzen Sie die folgenden Algorithmen in Programmlinien-Diagramme um!*
   a) *Berechnung einer KELVIN- aus einer Grad-CELSIUS-Temperatur*
   b) *Erraten einer Farbe aus dem klassischen Regenbogen (rot, orange, gelb, grün, türkis, blau, violett), die sich der Nutzer ausgewählt hat*
   c) *Berechnung des Multiplikator (f) und den Rest (r) für die ganzzahlige Division (im Bereich der Natürlichen Zahlen) ($y = f\,x + r$)*
   d) *Prüfung, ob es sich bei einer natürlichen Zahl um eine Primzahl handelt*

x.

1.2.1.3. Struktogramme



Struktogramme sind die ultimative Visualisierung von Algorithmen in der Schule. Dies ist auch durch ihre gute Eignung für Top-down-Entwicklungen bedingt. Aus einem "groben" Block lassen sich schnell viele "feinere" Blöcke erstellen, die dann wieder durch noch feinere bis auf Anweisungs-Niveau runtertransferiert werden können.
nach NASSI und SHNEIDERMAN

definiert / in DIN 66261 bzw. EN 28631

Lösung quadratischer Gleichungen vom Typ: $y = a\,x^2 + b\,x + c$

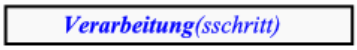
Eingabe: a
Eingabe: b
Eingabe: c
? a = 0
NEIN / *JA*
p = b / a
q = c / a
*d = (p * p) / 2 -q*
Ausgabe: keine quad. Gleichung
? d >= 0
JA / *NEIN*
wd = Wurzel(d)
x1 = p / 2 + wd
x2 = p / 2 - wd
Ausgabe: x1
Ausgabe: x2
Ausgabe: keine Nullstelle

Regeln / Empfehlungen zur Arbeit mit Struktogrammen
maximale Größe eines Struktogramms auf eine Seite (A4) beschränken
es gibt nur einen Eingang (Start) und einen Ausgang (Ende)
jedes Grundsymbol hat ebenfalls nur einen Eingang und einen Ausgang
Grundsymbole können aneinander / hintereinander (Sequenz) oder ineinander (Schachtelung)
schrittweise Verfeinerung nach dem Top-Down-Prinzip

zusätzlich:
für größere Algorithmen zusätzlich Struktur-Baum (hierarchisches Funktions-Diagramm) anlegen

zugelassene Algorithmen-Strukturen (mit Sinnbildern nach DIN)

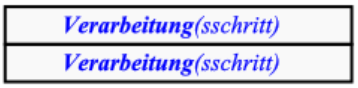
- **(Wert-)Zuweisung** — Rechteck mit Berechnung, Beschreibung der Operation oder auch ganzer Aufgaben-Blöcke



  - **Eingabe**

  - **Ausgabe**

- **Sequenz** — Aufeinanderfolge von Blöcken (Rechtecken) gleicher Breite



- **Verzweigung** — Rechteck mit Keil-Teilung (oben steht Vergleich od.ä.); unten links und rechts die möglichen Ausprägungen) (darunter geht es in geteilten Blöcken weiter; Nein/Falsch-Zweig kann auch leer sein)

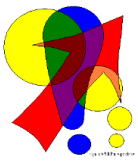
- **Zyklus** — Rechteck mit innerem (rechteckigem) Block; (Lage des inneren Blockes je nach Steuerungstyp unten oder oben)

Kopf-gesteuert; abweisend

Fuß-gesteuert; nicht abweisend

- **Unterprogramm**
  Prozedur
  Funktion — Rechteck mit doppelter Rahmenlinie rechts und links

- **Abbruch** — Rechteck mit rausweisendem Winkel / Pfeil



Hinweise:

Die nachfolgenden Struktogramm-Symboliken sind falsch und dürfen NICHT verwendet werden!

Blöcke dürfen nur vor- oder nachgelagert (**grünliche Positionen**) oder – wenn möglich – vollständig intern erweitert (**orange-farbene Positionen**) werden.
Ein Überlappen von Blöcken ist nicht zulässig!

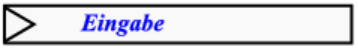
Aufgaben:

1. *Überlegen Sie sich, an welchen Stellen eine Verzweigungs-Block eweitert werden darf! Verwenden Sie die gleiche Farb-Codierung, wie oben!*
2.

zusätzliche / übliche Strukturen (mit Sinnbildern; z.T. in der DIN aufgeführt)

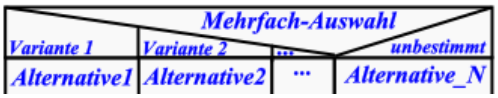
- **Block** — zwei ineinander geschachtelte Rechtecke

- **Anweisung**
  - **Eingabe** — veraltet



  - **Ausgabe** — veraltet

- **Mehrfach-Verzweigung**



abgeleitet aus den Regeln für die PAPs ergeben sich fast identische Regeln und Vorschriften für Struktogramme:

Arbeitsregeln für die Erstellung von Struktogrammen

- **Allgemeingültigkeit**
  - Struktogramme sollten keine Programmiersprachen-spezifischen Befehle usw. enthalten
  - die dargestellten Abläufe und Logiken sollten leicht zu verfolgen und zu verstehen sein
  - ein Struktogramm sollte in jede beliebige (imperative) Programmiersprache umgesetzt werden können
- **Deklaration**
  - neben den Abläufen sind bestimmte Vorgaben / Festlegungungen (Zahlen-Typ, Text-Längen, …) zu machen
  - die Deklarationen sind vor den Verwendung zu notieren (optimalerweise in den ersten / dem ersten Block)
- **Exklusivität**
  - jede einzelne Schritt (Eingabe, Ausgaben, Anweisung, …) erhält ein eigenes Symbol lt. DIN / EN
  - es darf keine Zusammenfassung von mehreren (auch nicht bei gleichartigen) Anweisungen / Schritten erfolgen!
  - Gruppen von (später zu verfeinernden oder schon vorhandenen) Block-Squenzen (z.B. Detail-Algorithmen) können zu einem Block zusammengefasst / durch einen Block ersetzt werden
  - zu jedem Struktogramm gehört ein Name

Programme zur Struktogramm-Erstellung:

- **NSD-Editor** Lizenz: GNU General Public Licence

  http://diuf.unifr.ch/drupal/sites/diuf.unifr.ch.drupal.softeng/files/teaching/studentprojects/kalt/ftp-nsd.html

- **Struktorizer** zum Erstellen von NASSI-SHNEIDERMANN-Diagrammen
  JAVA-Laufzeitumgebung notwendig
  auch als App im AbiIOStick; als portable App nutzbar

  Virengeprüfter Download:
  http://www.chip.de/downloads/Structorizer_64884440.html

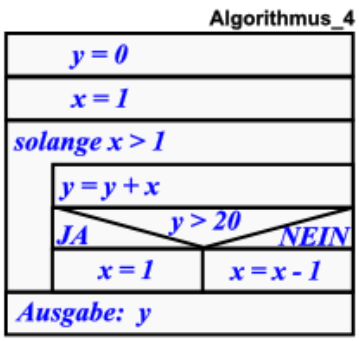
-

Biographie: Ada LOVELACE (1815 - 1852)

Namensgebend für einer der großen Programmiersprachen: "Ada". Heute nicht mehr so bedeutsam.

Aufgaben:

1. *Von einem Auto ist die Motor-Leistung in PS bekannt (Eingabe). Erstellen Sie ein Struktogramm, anhand dessen eine Umrechnung in die gültigen SI-Einheit erfolgt! (Die ordnungsgemäße Ausgabe gehört dazu!))*
2. *Erstellen Sie ein Struktogramm, für die Umrechnung von einer Grad-CELSIUS- in eine KELVIN-Temperatur mit Erkennung von Eingabefehlern (nur bezüglich der Zahlen)!*
3. *Geben Sie einen Algorithmus für die folgenden Aufgaben in Form jeweils eines Struktogramms an!*
   a) *Eine einzugebende (natürliche) Zahl soll, wenn sie gerade ist mit 2 multipliziert werden, wenn sie eine ungerade Zahl ist soll eine 3 addiert werden.*
   b) *Für ein einzugebendes Jahr soll geprüft werden, ob es sich um ein Schaltjahr handelt.*
   c) *Ein Programm zur Unterstützung eines Trainers soll aufgrund des gemessenen Pulses empfehlen, ob der Sportler weiter eine Runde läuft (bis Puls 180) oder die Runde schnell gehen soll. Nach der 12. Runde soll das Trainings-Programm für den Sportler beendet werden.*
   d) *Für einen Statistik-Test soll 1000x gewürfelt werden. Die Würfe mit einer 5 sollen gezählt werden. Am Ende des Würfelns soll geprüft werden, inwieweit der Versuchswert (prozentual) vom Erwartungswert abweicht!*
   e) *In einem weiteren Statistik-Versuch wird mit einem Dodekaeder gewürfelt. Die Seiten sind mit Zahlen beschriftet. Dieses Mal sollen alle Würfe ausgewertet werden.*
4. *Überlegen Sie sich, was das nebenstehende Struktogramm beschreibt und welches Ergebnis am Ende ausgegeben wird!*

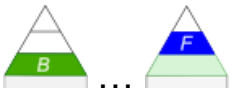


5. *Die Summe aller (natürlichen) Zahlen zwischen 1 und 243 sollen mit einer Fuß- und einer Kopf-gesteuerten Schleife berechne. werden! Entwickeln Sie dazu passende Struktogramme!*
6. *Jeder wählt sich ein kleines Problem und entwickelt dazu Struktogramm! (auf sprechende Variablen wird mit Absicht verzichtet!))*
   Die Struktogramme werden innerhalb des Kurses ausgetauscht (z.B. 3 Plätze weitergegeben). Was macht das nun vorliegende Struktogramm und welcher Algorithmus steckt dahinter?

für die gehobene Anspruchsebene:

7. *Wählen Sie einen Algorithmus / eine Berechnung (z.B. aus der Mathematik, Wirtschaft, Physik, …). Erstellen Sie dazu ein Struktogramm! Versuchen Sie nur allgemeine Abfragen / Angaben zu machen (also z.B. statt "Eingabe Temperatur" nur "Eingabe "Wert1" usw. usf.). Hängen Sie das Struktogramm für alle öffentlich aus! Alle Kursteilnehmer sind nun dazu aufgefordert, die verschiedenen Algorithmen zu erkennen. Wer erkennt die meisten?*

1.3. Übersetzung von (höheren) Programmiersprachen in Maschinencode

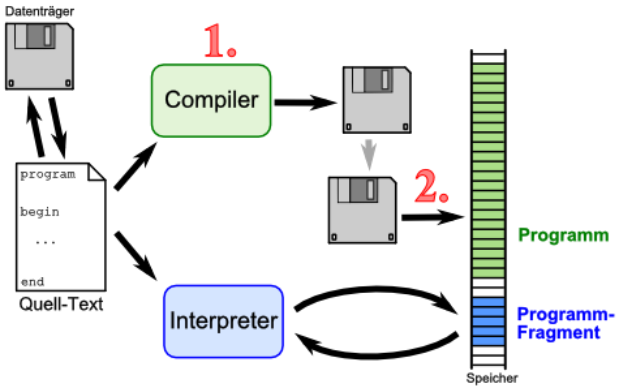


Als es noch keine Computer gab,
gab es auch das Programmieren als Problem nicht.
Als es dann ein paar leistungsschwache Computer gab,
wurde das Programmieren zu einem kleinen Problem
und nun, wo wir leistungstarke Computer haben,
ist auch das Programmieren zu einem riesigen Problem angewachsen.
In diesem Sinne hat die elektronische Industrie
kein einziges Problem gelöst, sondern nur neue geschaffen.
Edsger Wybe DIJKSTRA (1972)

Definition(en): Interpreter
Interpreter sind Übersetzungsprogramme, die einen Quelltext (**einer bestimmten Programmier-Sprache**) zur Laufzeit Stück-weise in Maschinen-Code überführt.
Ein Interpreter ist ein Programm, das den Quelltext zur Laufzeit einliest, analysiert und ausführt. Der Interpreter wird bei jeder Abarbeitung des Programms (Quelltextes) gebraucht.

Definition(en): Compiler
Compiler sind Übersezungsprogramme, die einen Quelltext (aus einer höheren Programmiersprache) als Gesamtheit in Maschinen-Code überführt.
Ein Compiler ist ein Programm, das den Quelltext in ein – für sich – ausführbares Maschinen-Programm übersetzt. Der Compiler wird zur Abarbeitung des Programms nicht mehr gebraucht.

für viele Programmiersprachen existieren sowohl Interpreter als auch Compiler



Beim Compiler ist die Übersetzung und Abarbeitung zeitlich und meist auch räumlich voeinander getrennt. Der Compiler erzeugt aus dem Quelltext eine ausführbare Datei (in DOS-WINDOWS-Systemen eine EXE). Diese Exe-Datei kann dann woanders und zu jedem beliebigen Zeitpunkt – völlig unabhängig vom Compiler – ausgeführt / gestartet werden.
Interpreter arbeiten praktisch inline. Soll das Programm genutzt werden, muss der Quelltext neu übersetzt werden. Dabei entstehen kleine Programm-Abschnitte, die dann im Speicher ausgeführt werden.
Für den nächsten Programm-Abschnitt aus dem Quelltext wird dann die Übersetzung im nächsten Schritt getätigt. Für Interpreter-Sprachen werden also immer der Quelltext und das Interpreter-Programm (bzw. eine Runtime-Version davon) benötigt.

	Interpreter	**Compiler**
Vorteile	• kleine Programme (nur Quelltext) • meist nur kleiner Teil des Interpreters muss an die Besonderheiten des Betriebssystems etc. angepasst werden (Kernel, Kern) • Laufzeitfehler können interaktiv korrigiert werden •	• Übersetzung nur einmalig notwendig • EXE kann frei (unabhängig vom Compiler) verteilt werden • auf Anwendersystem nur Speicherplatz für EXE und die Daten benötigt •
Nachteile	• Übersetzungsvorgang muss bei jedem Programmlauf erneut erfolgen • Anwender eines Programms braucht immer auch den Interpreter auf seinem System (Speicherplatzbedarf!) •	• relativ aufwändige Programme (enthalten viele Bibliotheken und Module) • für jedes Betriebssystem etc. ist ein spezieller Compiler notwendig • Laufzeitfehler u.U. schwer zu lokalisieren •

Der wohl erste Compiler für eine Programmiersprache wurde für FORTRAN entwickelt. Der damalige Aufwand betrug 18 Personen-Jahre. Heute – basierend auf den Erkenntnissen der Theoretischen Informatik – ist es zur Praktikums-Aufgabe für Studenten geworden. Dazu später mehr. Besonders die Abschnitte zu den Sprachen, Grammatiken und Automaten sind hier relevant.

Interpreter-Prinzip:

SourceCode → Interpreter → Programm → Ausgabe ***Laufzeit (Runtime)***
Daten ↗

praktischer Simultan-Übersetzer

Compiler-Prinzip:

SourceCode → Compiler → Programm ***Entwicklungs-Zeit (Compiletime)***

Daten → Programm → Ausgabe ***Laufzeit (Runtime)***

bei JAVA und Python Kombination aus Compiler und Interpreter

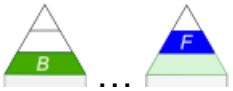
SourceCode → Compiler → ByteCode ***Entwicklungs-Zeit (Compiletime)***

Laufzeit (Runtime)

ByteCode → ByteCode-Interpreter → Programm → Ausgaben
Daten ↗

V: relativ schnell; Plattform-unabhängig

1.4. Einteilung der Programmiersprachen



besser eigentlich Einteilung nach den vorherrschenden Paradigmen (Konzepte; Denkschemata) der Programmiersprache
meist sind mehrere Paradigmen / Struktur-Systeme in den konkreten Programmiersprachen vereint, um sie breit einsetzbar zu machen und methodische Schwierigkeiten / Eingeschränktheiten zu umschiffen
schließlich steht nicht das Prinzip im Vordergrund sondern ein lauffähiges Programm zur Problem-Lösung

	Programmiersprachen / Paradigmen		
Problem	**imperative / prozedurale**	**deklarative / prädikative**	
Beschreibung Kreis	LEGE Mittelpunkt fest SETZE in einer Richtung Punkt in bestimmten Abstand (= Radius) VERÄNDERE Richtung schrittweise WIEDERHOLE (ab SETZE) bis Richtungsänderung mindestens 360° beträgt	Menge aller Punkte zu einem vorgegebenem (Mittel-)Punkt, die alle den gleichen Abstand (Radius) zu diesem haben	
Merkmale / Eigenschaften			
Vorteile			
Nachteile			
geeignet für …			
eher ungeeignet für …			

Programmier-Paradigmen

• **strukturierte Programmierung**	Programme bestehen aus Teil-Programmen, die im Wesentlichen auf den Strukturen Sequenz, Selektion und Interation bestehen Teil-Programme können wiederum aus Teil-Programmen (z.B. Prozeduren, Funktionen, …) bestehen
• **generative Programmierung**	automatische Code-Erzeugung durch einen Generator der Generator basiert auf dem EVA-Prinzip
• **Objekt-orientierte Programmierung OOP**	z.B.: Java, Delphi (ObjectPascal)
• **Aspekt-orientierte Programmierung AOP**	Erweiterung der Objekt-orientierten Programmierung über Klassen hinweg Trennung von Programm-Logik und Geschäfts-Logik konsequente Polymorphie z.B.: AspectJ
• **Modell-getriebene Software-Entwicklung**	vom Modell zum automatisch erzeugten Quell-Code nicht eindeutig unabhängig von den anderen Paradigmen, hat eher etwas verbindenes, verallgemeinerndes z.B. UML

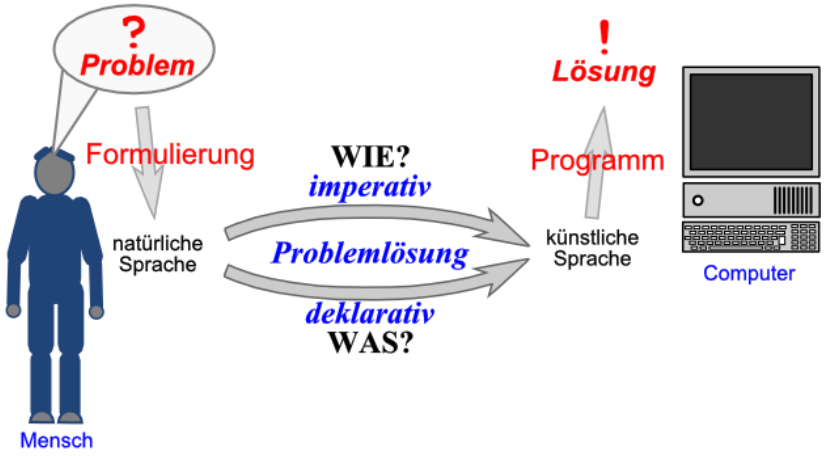
weitere theoretische Programmier-Paradigmen

• **intensionalle Programmierung**	(noch ohne offizielle Implementierung) Abrückung von klassischen Quell-Text Orientierung auch an anderen Sprachen und Notationen (Notenschrift, chem. Zeichensprache, Schaltungs-Symbole, …) Erfassen mehrerer Aspekte und Modelle des Ziel-Objektes
• **Subjekt-orientierte Programmierung**	basiert auf dem Objekt-orientiertem Ansatz → jetzt Subjekte Subjekte haben Status und eine Objekt-Identifikation keine Interna zu den Objekten bekannt Wirkung im Programm wird von Aktivierungs-Reihenfolge der Subjekte bestimmt (in der OOP vom Verhalten der Klasse) kooperativer Ansatz
•	

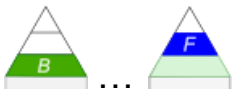
	Programmier-Paradigmen / Programmier-Modell										
Programmier-sprache	**imperativ**	**struktoriert**	**prozedural**	**Objekt-orientiert**		**deklarativ**	**funktional**	**logisch**		**nebenläufig**	
Ada	✓			✓						✓	
Basic											
C	✓										
Haskel	+/-					✓				+/-	
Pascal											
Prolog						✓	✓	✓			
Python											
Scala	+/-			✓		+/-	✓			✓	
Scheme	✓			+/-		✓	✓			+/-	

1.4.1. strukturierte Programmiersprachen

Programme sind in kleinere Untereinheiten (Module, Unterprogramme, Funktionen, Prozeduren, …) zerlegt
alle Teile basieren auf den Grund-Struktur-Elementen (Folge (Sequenz), Verzweigung (Selektion), Wiederholung (Interation))
mehr allgemeines Kriterium
kaum durch spezielle Programmiersprachen unterlegt, die nur dem Kriterium der Strukturierung folgen



1.4.2. imperative Programmiersprachen



auch: **Befehls-orientierte Programmiersprachen**
Anweisungen und Befehle beschreiben die genaue Arbeit / die genauen Arbeitsschritte, die der Rechner zur Lösung der Aufgabe abzuarbeiten hat
Beschreibung des Lösungsweges
Programm beschreibt **WIE** ein Problem gelöst werden soll
das Programm enthält also direkte Arbeits-Anweisungen für den Computer: MACHE die; TUE jenes; BERECHNE a+b; GEBE_AUS Zeichen "A"
eingebautes Variablen-Konzept

Folge aus nacheinander auszuführenden Anweisungen

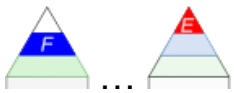
Beispiele für imperative Programmiersprachen

- **Assembler** (niedere /) Maschinen-nahe Programmiersprache
- **Ada**
- **ALGOL (ALGOrithmic Language)** Block-Struktur; lexikalische Variablen-Bindung
- **BASIC** eine sehr einfache Programmiersprache, in die man sich extrem schnell einarbeiten kann
  mittlerweile veraltete Einsteiger-Programmiersprache
- **C / C++ / C#** Klasse von Programmiersprachen, die relativ Maschinen-nah ist
  neuer Varianten (C++, C#) sind auch Objekt-orientiert
- **COBOL**
- **FORTRAN**
- **Java** derzeit die am breitesten verwendete Programmiersprache
- **Modula**
- **PASCAL** einfache Sprache, die ursprünglich für akademische und Ausbildungs-Zwecke entwickelt wurde
- **Perl**

Beispiele für imperative Programmiersprachen (Fortsetzung)

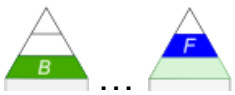
- **PL/1**
- **Simula**
- **Smalltalk**
- **Python** beliebte Sprache; klein aber effektiv
  sehr flexibel
- **Ruby** sehr kompakte und hoch effektive Sprache mit einem eigenen (etwas anderen) Syntax-System
-

1.4.2.1. prozedurale Programmiersprachen



beinhalten Variablen, Zuweisungen und Kontroll-Strukturen
es gibt ev. Unterprogramme, Prozeduren und Funktionen
beim Aufruf einer Funktion erfolgt i.A. die Erfüllung einer speziellen Aktion oder es wird ein (einzelner) Rückgabe-Wert erzeugt, der an den Funktions-Namen gebunden ist
klassische Beispiele sind solche Funktionen, wie Wurzel, Sinus oder Tangens
Prozeduren haben keinen an den Proezedur-Namen gebundenen Rückgabe-Wert.
Hier erfolgt ein Daten-Austausch (auch Rückgabe ist möglich) über Parameter. Diese müssen mit einer besonderen Kennung versehen werden, damit sie nicht am Ende der Prozedur gelöscht werden
typisch hierfür ist das klassische BASIC
moderne BASIC-Varianten (VisualBASIC) sind auch Objekt-orientiert angelegt (→ x.y.z.a. Objekt-orientierte Programmiersprachen)

1.4.2.2. Objekt-orientierte Programmiersprachen



im Vordergrund steht ein Objekt bzw. der Typ eines Objektes (Klassen) zu dem Daten (Attribute) und Methoden festgelegt werden
Attribute sind die Eigenschaften und Merkmale der Objekte. Sie charakterisieren die einzelnen Objekte. Objekte werden aus Klassen – quasi Objekttyp-Beschreibungen – abgeleitet. Klassen können ebenfalls Attribute besitzen, um z.B. alle Konten einer Bank zählen zu können.
Methoden sind die Fähigkeiten, die Objekte haben. Sie können bestimmte Aktionen durchführen, Werte berechnen usw. usf. Die Definition der Methoden erfolgt bei den Klassen. Je-

des Objekt erhält bei seiner Erstellung den gesamten Klassen-Satz an Methoden. Dazu kann ein Objekt noch individuelle Methoden dazubekommen oder die der Klasse überschreiben.

Objekte sind gekapselte Bausteine, die intern Zustände und Prozeduren enthalten und mit der Außenwelt nur über Nachrichten kommunizieren. Ein direkter Eingriff von außen auf die Zustände oder die Prozeduren ist nicht möglich / nicht erwünscht

beinhalten Objekte und Klassen
realisieren abstrakte Datentypen (ADT's) und Vererbung

in rein Objekt-orientierten Programmiersprachen ist alles über Objekte und ihren Zuständen und Prozeduren realisiert
eher selten
Beispiele: Common Lisp, Smalltalk

partiell Objekt-orientierte Programmiersprachen enthalten auch andere Programmier-Modelle vielfach wurde den "klassischen" Programmiersprachen irgendwann das Objekt-Modell hinzugefügt

Beispiele:	PASCAL	→	ObjectPASCAL		
	BASIC	→	VisualBASIC		
	C	→	C++	→	C#

Paradigmen

- **Polymorphismus** — Fähigkeit eines Bezeichners oder einer Funktion, mit verschieden Datentypen benutzt zu werden
- **Datenkapselung** — Arbeiten nach dem Black-Box-Prinzip; für äußere Elemente ist die innere Realisierung nicht sichtbar
  (intern könnte z.B. ein Objekt in einer anderen Programmiersprache beschrieben werden, als das Objekt-aufrufende / -benutzende verwendet
- **Vererbung** — bedeutet, dass abgeleitete Klassen / Objekte die Daten-Strukturen und Methoden der übergeordneten / vererbenden Klasse mitbekommt (erbt)

fast alle neueren Programmiersprachen oder die neueren Versionen alter / bewährter Programmiersprachen sind Objekt-orientiert ausgelegt / um die Objekt-orientierung erweitert

Beispiele für Objekt-orientierte Programmiersprachen

imperativ …

- **Smalltalk**
- **Java**
- **Eifel**
- **C++ / C#**
- **Object PASCAL**
-
-

funktional …

- **XLisp**
- **Haskell**
-
-
-
-
-

prädikativ … / logisch …

- **Prolog**
-
-

1.4.2.3. moderne ikonisch-symbolische Programmierung (Block-Programmierung)

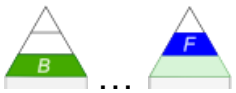
statt mit Texten zu arbeiten, werden Blöcke (quasi nach dem Baustein- bzw. LEGO-Prinzip)mit einander kombiniert

Snap!
Scratch
Blockly
MakeBlock
…

Snap4Arduino (→ http://snap4arduino.rocks/; online Version: http://snap4arduino.rocks/run/) od. S4A (→ http://s4a.cat/) od. ModKit (→ http://www.modkit.com/)

ähnliche Systeme werden in der produktiven Praxis ebenfalls genutzt, dann allerdings nicht ganz so aufgebläht und schrill gefärbt
einen Eindruck wie so etwas aussehen könnte, kann man bei den modernen Versionen von BlueJ und Greenfood sehen. Hier ist der Block-Charakter allerdings nur unterschwellig realisiert.

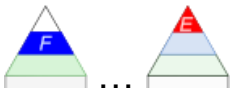
1.4.3. deklarative Programmiersprachen



Programmierer gibt neben der Daten-Basis und Regeln / Funktionen noch das Ziel oder die Problem-Frage vor, der Rechner sucht den Lösungsweg alleine (der Weg ist das Ziel)
Programm beschreibt das Wissen zur Lösung des Problems / der Aufgabe
oft interessiert nur, ob es eine Lösung gibt oder nicht
die exakten Lösungen sind dann Beiwerk

Spezifikation dessen, **WAS** berechnet werden soll
Man kann sich das als Fragestellen an das System vorstellen: ? Gibt es einen Zusammenhang zwischen Sachverhat A und B, egal über wieviele Ecken?
? Gibt eine mathematische Lösung eines Problem's, unabhängig von einem genauen / konkreten Algorithmus?
Compiler / Interpreter legt fest, wie Berechnung erfolgt
realiseren effektive Such-Verfahren, können Rücksprünge (Backtracking) machen, wenn es keine Lösung mehr innerhalb eines Versuchs-Zweiges gibt.
beherschen z.B. auch Versuch-und-Irrtum-Verfahren (trial and error) oder Teile-und-herrsche-Verfahren (dividere and conquer).

1.4.3.1. logische Programmiersprachen



auch: **prädikative Programmiersprachen**
Wissen wird aus Fakten (Wissensbasis, Prädikaten, Definitionen) und Regeln (Verhältnisse / Beziehungen zwischen den Fakten) zusammengestellt (meist völlig ungeordnet)
Programm kann neue Fakten ableiten
Regeln zur Definition von Relationen (Beziehungen zwischen Fakten)

Programm zieht Schlussfolgerungen; stellt logische Verbindungen (Schlüsse) zwischen akten auf
zur Lösung eines Problems werden Operationen der (mathematischen) Logik verwendet
bauen auf Prädikaten-Logik auf
Programm beschreibt **WAS** das Problem ist; die Lösung sucht der Computer (durch systematischen Durchsuchen des Wissens) und unter verwendung allgemeingültiger Problem-Löse-Algorithmen

Beispiele für logische Programmiersprachen

- **Prolog** relativ einfache, universelle und weit verbreitete logische Programmiersprache
- **Goedel**
- **Escher**
- **Mercury**
-
-

logische Programmiersprachen arbeiten oft nach der REPL-Arbeitsweise
Read – Evaluate – Print – Loop
Lesen der Eingabe; Auswerten der Eingabe(n); Ausdruck von Ergebnis oder Fehler-Meldung; Beginn von vorne

1.4.3.2. funktionale Programmiersprachen

auch: **applikative Programmiersprachen**
zur Lösung des Problems werden (definierte) Funktionen und / oder deren Verknüpfungen verwendet
Programm ist Folge / Schachtelung von aufeinander bezogenen, einfachen Funktionen
Funktionen hier im klassischen Sinne als eindeutige Abbildung einer Eingabe-Menge auf eine Ausgabe-Menge verstanden
Programm ist auch wieder (nur) eine Funktion; kann sich selbst aufrufen (Rekursion)
Paradigma schwer konsequent durchhaltbar, deshalb nur rel. wenig wirklich durchgehend funktional funktionierende Programmiersprachen verfügbar
Determinierung des Programms durch zugewiesene Parameter (Funktions-Argumente)
meist mit

keine Seiten-Effekte
Rekursion ist dominierendes Lösungs-Prinzip

Beispiele für funktionale Programmiersprachen

- **Lisp (LISt Processor)** — einfacher (???) Syntax; flexibel; erweiterbar; interaktiv; dynamische Prüfung der Datentypen; automatische Speicher-Verwaltung; rekursives Modell
- **Logo**
- **Haskell**
- **Hope**
- **Scheme**
- **Sisal**

Programm-Beispiele für Lösungen in funktionellen Programmiersprachen:

HASKELL: Summe und Produkt einer Liste von Zahlen:

```
sumList :: Num a => List a -> a
sumList Nil = 0
sumList (Cons x xs) = x + sumList xs

productList :: Num a => List a -> a
productList Nil = 1
productList (Cons x xs) = x * productList xs
```

SCHEME: Summe und Produkt einer Reihe von Zahlen:

```
(define (summe n)
  (if (= 0 n)
    0
    (+ n (summe (- n 1)))))

(define (produkt n)
  (if (= 1 n)
    1
    (* n (produkt (- n 1)))))
```

LISP: Summe und Produkt einer Reihe von Zahlen:

```
(defun summe (n)
  (if (eq n 0)
    nil
    (cons n (summe (- n 1)))
  )
)

(defun produkt (n)
  (if (eq n 1)
    nil
    (cons n (produkt (- n 1)))
  )
)
```

1.5. komplexe Programmier-Aufgaben:

Wählen Sie eine geeignete oder Ihre präferierte Programmiersprache zur Lösung der nachfolgenden Aufgaben aus!
Überlegen Sie sich bzw. vergleichen mit anderen, ob die von Ihnen präferierte programmiersprache gut geeignet ist das gewählte Problem zu lösen!
Zahlen-Eigenschaften nach: www.zahlen.mathematic.de

Aufgaben:

1. ***Berechnen Sie die Summe und das Produkt einer Reihe von einzugebener Zahlen sowie Summe und Produkt der reziproken Werte!***
2. ***Erstellen Sie ein Programm, dass im Zahlen-Raum bis zur einer einzugebenen (größeren) natürlichen Zahl, die Kombination von drei aufeinanderfolgenden Primzahlen findet, deren Produkt möglichst dicht an der Zahlen-Grenze liegt!***
3. ***Prüfen Sie ob eine als Zeichen-String vorgegebene Zahl (ohne Leer- und Vorzeichen bzw. Nachkommastellen) im auszuwählenden Zahlensystem gültig ist! (Die Ziffern werden als ASCII-Zeichen notiert. Gültige und unterscheidbare Zeichen sind: 0 .. 1 A .. Z a .. z → das sollte auch bis zum Sexagesimal-System reichen! Doppeldeutung A = a muss nicht beachtet werden!)***
4. ***Lassen Sie durch eine Erweiterung des Programms von 3. prüfen, ob es sich bei der eingegeben Zahl um eine normale Zahl handelt! Normale Zahlen enthalten alle Ziffern ihres Alphabetes mit der gleichen Häufigkeit.***
5. ***Erstellen Sie das Programm "Zahlen-Charakterisierer"! Das Programm soll eine einzugebene ganze Zahl (ev. zuerst nur für natürliche Zahlen) Charakter-Eigenschaften prüfen bzw. bestimmen und ausgeben, ob die Zahl die Eigenschaft hat oder nicht bzw. den berechneten Wert. Das Programm sollte später um weitere Zahlen-Eigenschaften ergänzt werden können und passend kommentiert sein! Auf die (spätere) Nutzbarkeit von Unterprogrammen ist zu achten! Wählen Sie sich mindestens 12 Eigenschaften aus! Die Reihenfolge kann frei geändert werden!***
   a) ***männliche Zahl (Zahl ist ungerade und größer als 1)***
   b) ***Quersumme (ist die Summe der einzelnen Ziffern der Zahl (ohne deren Potenzwert))***
   c) ***titanische Zahl (ist eine Primzahl mit mindestens 1000 Stellen***
   d) ***weibliche Zahl (Zahl ist eine positive gerade Zahl)***
   e) ***Totient od. Indikator (ist die Anzahl der Primzahlen, die kleiner als die (gegebene) Zahl ist)***
   f) ***zusammengesetzte (od. zerlegbare od. teilbare) Zahl (ist eine Zahl, die mehr als zwei positive Teiler hat ODER eine gerade Zahl, die größer als 1 ist)***
   g) ***abundante Zahl (wenn echte Teilersumme (Summe aller Teiler (ohne Rest), außer die Zahl selbst) größer als die Zahl selbst ist)***
   h) ***arme od. defizierte od. mangelhafte Zahl (wenn echte Teilersumme kleiner als das doppelte der Zahl ist)***

i) vollkommene od. perfekte Zahl (wenn die echte Teilersumme gleich der Zahl selbst ist)

j) Sophie-GERMAIN-Primzahl (sind Primzahlen, bei denen der Term 2 p +1 wieder eine Primzahl ist)

k) reiche od. überschießende od. übervollständige Zahl (wenn die echte Teilersumme größer als das Doppelte der Zahl selbst ist)

l) SMITH-Zahl (wenn die Quersumme der Zahl gleich der Quersummen ihrer Primfaktoren ist; außer Primzahlen!)

m) erhabene Zahl (wenn Zahl und deren echte Teilersumme vollkommene Zahlen sind)

n) palindrome Zahl (wenn die Zahl und die umgedrehte Ziffernfolge gleich (groß) sind)

o) palindrome Primzahl (wenn Zahl eine Primzahl ist und die Zahl und deren umgedrehte Ziffernfolge gleich sind)

p) SIERPINSKI-Zahl (ist eine ungerade, natürliche Zahl n, bei der der Term $n\,2^x + 1$ immer eine zusammengesetzte Zahl ergibt (x ist eine beliebige natürliche Zahl))

q) RIESEL-Zahl (sind ungerade, natürliche Zahlen, bei denen der Term $n\,2^x - 1$ immer eine zusammengesetzte Zahl ergibt (x ist eine beliebige natürliche Zahl))

r) strobogrammatische Zahl (ist eine Zahl, die um 180° gedreht wieder die gleiche Zahl ergibt (hier gelten 1, 2 mit 5, 6 mit 9, 8 und 0 als drehbare oder strobogrammatische Ziffern))

s) strobogrammatische Primzahl (ist eine Primzahl, die auch strobogrammatisch ist)

6. Gesucht wird ein modulares Programm, dass für zwei natürliche Zahlen prüft, ob es sich um ein Paar mit den folgenden Eigenschaften handelt!

a) befreundete Zahlen (wenn die echten Teilersummen beider Zahlen gleich sind))

b) Primzahlen-Zwilling (wenn zwei aufeinanderfolgende Primzahlen eine Differenz von 2 aufweisen)

c) Teiler-fremde (od. inkommensurable) Zahlen (ganze Zahlen, die außer -1 und 1 keine gemeinsamen Teiler besitzen)

7. Gesucht wird ein modulares Programm, dass für drei natürliche Zahlen prüft, ob es sich um ein Tripel mit den folgenden Eigenschaften handelt!

a) pythagoreische Zahlen (Tripel erfüllt die diophantische Gleichung 2. Grades ($a^2 + b^2 = c^2$))

b) Primzahlen-Drilling (wenn drei aufeinanderfolgende Zahlen die Reihe p, p+2, p+6 bilden ODER wenn innerhalb einer Dekade (also 10 aufeinanderfolgenden Zahlen) drei Primzahlen vorkommen)

8.

1.6. evolutionäre Algorithmen

Besonders nicht-lineare Gleichungs-Systeme – und die sind die Basis der meisten komplexen Modelle, die z.B. von einem Algorithmus bearbeitet werden sollen – sind so vielgestaltig und eben nicht vorausberechenbar, dass eine systematische Suche ewig dauern würde. Ob das Ziel wirklich erreicht wird, hängt dann schon vom Such-Raster ab. Ist es zu groß übersieht man die optimalen Lösungen, ist es zu klein sucht man sich (zeitlich) tod. Genetische Startegien passen die Parameter an. Kommt man einem Ziel näher, wird eben feiner gesucht. Eine Gefahr besteht allerdings bei evolutionären Strategien – das wirklich beste Ziel findet man ev. nicht, weil man auf einem Nebenschauplatz (/ um ein lokales Optimum) kämpft. Da helfen aber meist mehrfache Suchläufe.
Solange das Ziel klar ist, scheint Programmierung ein relativ kleines Problem zu sein. Was aber, wenn genau das Ziel nicht so deutlich umrissen werden kann?
Wie es die Natur jeden Tag tut, so werden in der Praxis (Industrie, Handel, …) optimale Lösungen gesucht. Bei einer systematischen Suche muss man u.U. riesige Datenbereiche ausprobieren. Solche Such-Vorgänge sind oft extrem Zeit-aufwändig. Also sucht man effektivere Techniken. Dazu eignen sich ev. stochastische Strategien. Man variiert z.B. ein Detail mehr oder weniger zufällig und prüft dann die Eignung. Hat man eine Lösung gefunden, setzt man die stochastische Suche beim nächsten Faktor fort. Das Ziel hat man erreicht, wenn durch die Veränderungen ein Produkt entsteht, dass den Anforderungen entspricht. Noch effektiver ist das Erwürfeln vieler Faktoren und das vergleichende Prüfen der Lösungen gegeneinander. Nun kann man z.B. im Nahbereich um eine Zwischen-Lösungen wiederum zufällig nach noch optimaleren Lösungen suchen. Das Verfahren kann dann immer weiter verfeinert werden.
Ein anderes Verfahren geht von einem zufälligen Ausgangs-Punkt aus. Nun wird über einzelne Faktoren "mutiert" (leicht geändert). Die sich ergebenen Lösungen werden gegen die Vor-Lösung bewertet. Ist eine Lösung besser, wird sie der neue Ausgangs-Punkt und man setzt solange so fort bis man keine Verbesserung mehr findet. Gelangs man von mehreren Ausgangs-Situationen immer zur gleichen optimalen Lösung, dann kann man mit großer Wahrscheinlichkeit davon ausgehen, auch die wirklich beste Lösung gefunden zu haben. (Dafür gibt es aber auch bei sehr vielen Versuchen keine Garantie!)
Natürlich kann man solche Strategien auch auf Algorithmen oder Quell-Texte anwenden. Ungünstige Veränderungen bringen schechtere Ergebnisse hervor, günstige Veränderungen machen den Algorithmus vielleicht besser oder zumindestens gleichwertig. Die ungeeigneten Algorithmen werden verworfen und mit den guten weiter fortgefahren.
Das hört sich zuerst sehr aufwendig an, in vielen hoch-komplexen und z.T. auch sehr variablen Algorithmen ist es aber eine wesentlich effektivere Möglichkeit zu besseren Algorithmen(-Varianten) zu kommen.
Vielleicht hilft ein Vergleich bzw. Analogon dabei, das Prinzip besser zu verstehen.
Wir haben schon Primzahlen gesucht bzw. geprüft. Bei sehr großen Zahlen kann die Suche sehr rechen-aufwendig werden. Prüft man jeden möglichen Teiler z.B. aufsteigend alle ungeraden Zahlen, dann kann die Anzahl der Prüfungen schon recht groß werden.
Bei stochastischen Prüfungen geht man einfach davon aus, dass es auch ausreicht, nur einige wenige – aber zufällig ausgewählte – Teiler zu prüfen.
Schon bei wenigen Test's bekommt man eine recht hohe Wahrscheinlichkeit, dass es sich bei der zu prüfenden Zahl um eine Primzahl handelt oder eben nicht. Weitere Prüfungen verbessern das Ergebnis nur unwesentlich.

Aufgaben:

1. *Warum finden evolutionäre Algorithmen nicht garantiert die beste Lösung? Erläutere ausführlich!*
2. *Recherchiere, in welchen Bereichen heute mit evolutionären Algorithmen gearbeitet wird!*
3.

Arten evolutionärer Algorithmen

- **genetische Algorithmen (GA)** — Unterscheidung noch Geno- und Phänotyp (Genotyp z.B. Bit- od. Zeichen-Folge, die irgendwelche Phänotypen (z.B. Farben, Formen) codieren); Mutationen / Veränderungen auf dem Genotyp; Bewertung der resultierenden Phänotypen (ev. Auslese)
- **genetische Programme (PG)**
- **generalisierte genetische Programmierung (GGP)**
- **evolutionäre Strategien (ES)** — Nachkommen / neue Varianten haben leicht veränderte Details / mutierte Merkmale, die an den Anforderungen bewertet (ausgelesen) werden (nur die Geeignetesten kommen weiter)
- **evolutionäre Programmierung (EP)**
- **Neuronale Netze (NN, KNN)**
- **Memetische Algorithmen**

Ablauf-Schema eines genetischen Algorithmus

1. **Initialisierung**
2. **Ausgangs-Situation bewerten**
3. **ev. Mutation / Veränderung**
    - **ev. + Crossing over**
    - **ev. + Hinzufügen / Entfernen**
4. **Selektion (Bewertung) und Replikation (Vermehrung)**
5. **Erzeugung von Nachkommen durch Auswahl von Komponenten / Mutationen / …**
6. **Wiederholung ab Schritt 3 bis Abbruch-Kriterium erreicht wird**

Ablauf-Schema eines memetischen Algorithmus

1. **Erzeugung einer zufälligen Ausgangs-Population**
2. **lokale Suche jedes Individuum (nach optimalster Situation / Lage / Bedingungen / …)**
3. **Interaktion mit anderen Induviduen**
    a. **ev. + Kooperation / Crossing over**
    b. **ev. + Wettkampf / Selektion**
4. **Wiederholung ab Schritt 3 bis Abbruch-Kriterium erreicht wird**

1.7. Software-Ergonomie

Definition(en): Software-Ergonomie

DIN EN ISO 9241 Teil 110

Gestaltungs-Empfehlungen der Software-Ergonomie

Aufgaben-Angemessenheit

Funktionalitäten und Dialoge sollen der typischen Arbeits-Aufgabe / dem typischen Arbeits-Ablauf entsprechen

Selbstbeschreibungs-Fähigkeit

Nutzer soll zu jeder Zeit wissen, an welcher Stelle der Arbeit er gerade ist, welche weiteren Handlungen möglich sind und wie diese ausgeführt werden können

Erwartungs-Konformität

Passung der Software – deren Dialoge und Funktionen – zu den typischen / allgemein anerkannten – Standard's und Verabredungen

z.B. Hinweis auf noch nicht gespeicherte Daten beim Verlassen des Programm's
das x-Symbol als Zeichen zum Schließen des Programm's oder des Fenster's

Lernförderlichkeit

Programm soll den Nutzer durch den Ablauf führen und ihn unterstützen

Steuerbarkeit

Programm ist so gestaltet, dass der Anfang der Aufgaben-Bearbeitung klar ist, alle notwendigen Zwischen-Entscheidungen angemessen umgesetzt wurden und dass das Ende / Ergebnis klar ersichtlich wird

Fehler-Toleranz

trotz kleinerer – für die Software erkennbare – Fehler oder Fehl-Einstellungen soll – ev. mit minimalen Korrekturen – trotzdem das Ziel / Ergebnis erreicht werden (können)

Individualisierbarkeit

wenn der Nutzer seine Mensch-Maschine-Interaktion (z.B. Maus- oder Tastatur-Bedienung) sowie die Darstellung von Informationen jederzeit ändern kann

2. Objekt-orientierte Software-Entwicklung

moderne Art der Programmierung

Erfassen von Objekten und Objekt-Klassen, die auf dem Computer verarbeitet werden sollen

Objekte sind Dinge der Realität
bei der Klassen-Bildung werden wesentliche Eigenschaften herausgearbeitet

	"Max"			
	"King"			
	"Luis"			
	"Bello"		→	**Hunde**
	"Prinz"			
	"Rex"			
	"Stromer"			
Objekt(e)	*Bezeich-nung*	*charakeristische Merkmale*	*Beziehung*	*Objekt-Klassen-Bezeichnung*

vorher ließe sich ev. auch noch eine Objekt-Klasse vereinbaren, die alle "Schäferhunde" beinhaltet

schon Kleinkinder nehmen Verallgemeinerung vor, auch wenn sie für die Klasse noch andere Namen haben (z.B. "Wau-wau")

Modell-Bildung (vom Objekt zur Klasse)

Prozess der Umsetzung einer Miniwelt in ein Computer-/Programmier-Modell wird **O**bjekt-**o**rientierte **M**odellierung genannt (OOM)

OOM beginnt mit **O**bjekt-**o**rientierter **A**nalyse (OOA)
Beschreibung der Miniwelt mit ihren Objekten, deren Eigenschaften und Funktionen
entspricht der konzeptionellen Ebene aus dem Datenbank-Design (z.B. Erstellung eines Entity-Relationship-Diagramms (ERM))

Darstellung als (UML-)Klassen-Diagramm

Definition(en): Klasse
Eine Klasse ist eine Gruppe von Objekten, für die (charkteristische) Eigenschaften (Attribute, Merkmale), Operationen (Methoden, Funktionen, Fähigkeiten) und deren Semantik allgemeingültig festgelegt ist.

nach OOA folgt im Objekt-orientiertem Design (OOD) die Anpassung des Modells an die technischen Gegebenheiten und Möglichkeiten (z.B. Datenspeicher, Netzwerk-Verbindung, Benutzer-Oberfläche (GUI), …)
Festlegen von Gruppen und Hierrachien zur Klassifizierung der Objekte in funktionelle Einheiten

im Datenbank-Design entspricht dies der logischen Ebene, d.h. das Entity-Relationship-Diagramm wird in eine geeignete Datenbank umgesetzt (meist als relationale Datenbank)
dazu gehört das Planen von Zugriffs-Rechten; Sichten, Abfragen und Formularen (ev. auch Berichte und / oder Modulen (Makros oder Unterprogramme))

Ziel von OOA und OOD sind Aussagen darüber, welche Klassen man definieren möchte, wie sie im Rang zueinander stehen (Hierarchie), welche Eigenschaften sie haben und was sie für Aufgaben / Funktionen / Dienste sie erfüllen sollen

Attribute (Eigenschaften, Merkmale, …) dienen der Beschreibung von Objekten
mindestens ein Name (/ eine ID)
dazu Merkmale anhand derer die einzelnen - zur Klasse gehörenden – Objekte unterschieden und / oder charkterisiert werden können
für bestimmte Merkmale müssen gleich bei der Erzeugung / Ersterfassung des Objektes initiale Werte vorbelegt werden
in der Informatik ist auch die Sichtbarkeit von Attributen von außen ein wichtiges Thema (Black-Box-Prinzip)

Definition(en): Attribute
Attribute sind Daten-Elemente, die zur Charakterisierung der (Klasse von) Objekte(n) benutzt werden und durch individuelle (Objekt-abhängige) Werte belegt sind.

Definition(en): Methoden
Methoden sind Leistungen / Fähigkeiten (Dienstleistungen, Services, …) die von einem / für ein Objekt aufgerufen werden können.

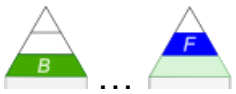
Methoden haben Signatur (Name; Parameter; Rückgabewerte)
häufig werden Methoden als Implementation einer Operation in einer Programmiersprache / einem Algorithmus / Problemlöse-Strategie betrachtet

heute ist ist UML (Unified Modeling Language) das favorisierte graphisches Mittel
seit 1997 standardisiert
Werkzeug für den Entwurf und der Dokumentation
moderne Systeme bieten Mehrwert: Roh-Quelltexte (geht bis hin zur automatischen Programm-Erstellung großer Software-Teile); Simulation von Modellen (Vollständigkeit, Leistungsfähigkeit, Widersprüche, Dopplungen, …)
in Industrie recht weit verbreiteter Standard (vor allem bei komplexen Systemen)
zunehmend auch in Schulen, Universitäten etc.

im Datenbank-Design folgt nun die Implementierung des Modells in ein konkretes Datenbank-Management-System (DBMS) einschließlich der notwendigen Sichten, Abfragen, Formulare, Berichte und Module
in der Objekt-orientierten Modellierung heisst dieser Teil Objekt-orientierte Programmierung (OOP)
Erstellen von Programmen auf der Basis von Klassen (mit Attributen und Methoden) und der Nutzung der Klassen-Definitionen zur Erzeugung und Nutzung von Instanzen (entsprechen den Objekten der Miniwelt)
Realisierung der Kommunikation(s-Möglichkeiten) zwischen den Klassen

Definition(en): Instanzen
Instanzen sind aus einer Klasse abgeleitete (Modell-)Objekte, die konkreten (Miniwelt-)Objekten zugeordnet werden.

2.1. UML



UML steht für Unified Modeling Language (vereinheitlichte Entwicklungs-Sprache)
seit 1997 Standard

in der praktischen Informatik hauptsächlich Werkzeug zur graphischen Darstellung von Objekt-Klassen und deren Beziehungen untereinander
Version 1:

derzeit aktuell Version 2:
seit 2015 Version 2.5
sehr breites Anwendungs-Feld

14 verschiedene Diagramm-Arten definiert

Diagramm-Arten im UML

- **Verhaltens-Diagramme (Diagram)** — beschreiben Abläufe bzw. Prozesse (*Wer und Was soll behandelt werden?*)
- **Use Cases Anwendungsfall-Diagramm (Use Case Diagram)** — definieren der Funktionen (*Wer macht Was?*)
- **Aktivitäts-Diagramme (Activity Diagram)** — definieren der Abläufe (*Was wird Wann gemacht?*)
- **Klassen-Diagramme (Class Diagram)** — beschreiben der Objekt-Klassen (Modell-Objekte) und deren Zusammenspiel (Beziehungen) (*Was wird bearbeitet und Wie hängen sie zusammen?*)
  typisch für die Objekt-orientierte Programmierung (→ 2. Objekt-orientierte Software-Entwicklung)
- **Zustands-Diagramm (Statechart Diagram)** — wichtige Anwendung in der Automaten-Theorie (→ 3.2. Automaten) als ein Standard-Darstellungsmittel (→ 3.2.3. Zustands-Diagramme → 3.2.3.2. UML-Zustands-Diagramme)
- **Sequenz-Diagramm (Sequenz Diagram)**
- **Kollaborations-Diagramm (Collaboration Diagram)**
- **Komponenten-Diagramm (Component Diagram)**
- **Verteilungs-Diagramm (Deployment Diagram)**

Definition(en): Unified Modeling Language (UML)
UML ist eine graphische Modellierungs-Sprache für die Spezifikation, Konstruktion und Dokumentation von Software.

2.1.1. UML-Klassendiagramm

Klassen sind Zusammensetzungen von gleichartigen Objekten
Gleichartigkeit kann sich auf die Attribute und / oder die Methoden beziehen

abstrakte Klassen als Grundlage für weitere Unter- und / oder Ober-Klassen

drei-geteiltes Rechteck mit Klassen-Name, zweitens den Attributen (Eigenschaften) der Klasse und drittens den Methoden (Aufgaben, Leistungen, Dienste, …)

Hund		Klassen-Name
Name Rasse Steuernummer Alter Masse Wurmbehandlung		Attribute
gib_Laut läuft_nach hat _Geburtstag bekommt_Steuernummer setze_Wurmbehandlung zeige_Wurmbehandlung		Methoden

diese Rohversion wird noch durch wichtige technische Detail erweitert
wird nur ein enzelnes (konkretes) Objekt im Klassendiagramm dargestellt, dann wird der Objekt-Name unterstrichen

Bello		Objekt-Name
		Attribute

Zu den Attributen (Zustands-Variablen) gehören noch Datentypen, die im Programm benutzt werden

Hund		Klassen-Name
Name: String Rasse: String Steuernummer: Integer Alter: Integer Masse: Float Wurmbehandlung: Boolean		Attribute Datentypen: String … aphanumerische Zeichen-Ketten Integer … ganze Zahlen Float … Fließkommazahlen Boolean … Wahrheitswert
		Methoden

häufig benutzt man weit verbreite Standard-Namen für die Datentypen. Das liegt daran, dass die fertigen UML's später in beliebige Programmiersprachen umgesetzt werden können. In vielen Fällen muss man aber auch an den Konventionen der UML-Software orientieren

neben dem Datentyp ist es bei Variablen wichtig zu wissen und festzulegen, wer alles auf eine Variable zugreifen darf
Programmierer sprechen auch von der Sichtbarkeit und Verfügbarkeit
klassisch sind viele Variablen nur innerhalb einer Klasse sichtbar
für einen Programm-Nutzer (des fertigen Anwender-Programms) ist es nicht wichtig, in welche Form z.B. die Masse gespeichert wird, das könnten z.B. kg oder lbs. sein
unabhängig davon wird die Masse bei Erfordernis in der Landes-typischen Form ausgegeben
die üblichen Alternativen sind die private oder globale Nutzung / Freigabe der Variable
private (interne) Variablen erhalten ein Minus-Zeichen (**-**)
mit einem Plus-Zeichen (**+**) kennzeichnet man globale (public) Variablen
in UML-Diagrammen heißen die Charakterisierungs-Zeichen **Modifizierer** (Modifier) und werden vor dem Bezeichner (Namen) des Attributes (der Variable) notiert

Hund		Klassen-Name
+ Name: String **+** Rasse: String **+** Steuernummer: Integer **+** Alter: Integer **+** Masse: Float **-** Wurmbehandlung: Boolean		Attribute **Datentypen:** String … alphanumerische Zeichen-Ketten Integer … ganze Zahlen Float … Fließkommazahlen Boolean … Wahrheitswert
		Methoden

Syntax-Diagramm für Attribut: Modifizierer Bezeichner : Datentyp

als nächstes werden die Methoden genauer spezifiziert
einige Methoden brauchen zusätzliche Werte
diese werden üblicherweise Argumente genannt
z.B. beim Lautgeben könnte die Anzahl übergeben werden
die Notierung im Klassen-Diagramm erfolgt mit einem Variablen-Namen (der üblicherweise innerhalb der Methode privat ist) und dem Datentyp zu dieser Variable

Hund		Klassen-Name
…		**Attribute**
gib_Laut(anzahl: Integer) …		Methoden

wenn in einem anderen System (Programm) z.B. die Methode **gib_Laut()** ohne Argumente beschrieben wird, dann könnte das (standardmäßig) für ein einmaliges Bellen stehen.

Hund		Klassen-Name
…		**Attribute**
gib_Laut(anzahl: Integer) …		Methoden

Natürlich muss sich das aufrufende Programm daran später orientieren.
bei den Methoden ergibt sich das gleiche Sichtbarkeits-Problem, wie bei den Attributen
die Kennzeichnung erfolgt analog zu den Attributen
üblicherweise sind Methoden öffentlich (public)
typischerweise sind nur einige wenige – nur intern benutzte – Methoden privat
weiterhin kennzeichnet man im Modifizierer, ob die Methode einen fragenden / auslesenden Charakter (Anfrage) hat und einen oder mehrere Werte zurückliefert Das Zeichen dafür ist ein Fragezeichen (**?**).
bei Methoden mit setzenden / schreibenden Charakter (Auftrag) ohne einen Rückgabe-Wert (an das aufrufende Programm) wird ein Ausrufe-Zeichen (**!**) benutzt

Hund		Klassen-Name
+ Name: String **+** Rasse: String **+** Steuernummer: Integer **+** Alter: Integer **+** Masse: Float **-** Wurmbehandlung: Boolean		Attribute
+! gib_Laut(anzahl: Integer) **+!** läuft_nach(xpos, ypos: Float) **+!** hat _Geburtstag() **+!** bekommt_Steuernummer(Nummer: Integer) **+!** setze_Wurmbehandlung(): Boolean **+?** zeige_Wurmbehandlung(): String		Methoden

Das obige Beispiel ist nicht vollständig. Es sollen nur die Prinzipien aufgezeigt werden. In Programmen brauchen wir noch Programm-Funktionen, die ein konkrete Realisierung des Objektes "Bello" ermöglichen. In der Programmierung heißen die informatischen Abbilder eines Objektes Instanzen. Instanzen sind immer einer Klasse zugeordnet. Vor allem wenn mit mehreren Objekten in einem Programm gearbeitet werden soll, dann ist zum Einen einen Funktion notwendig, die einen neuen Hund – eben z.B. "Bello" – erzeugt. Solche Funktionen werden Konstruktor (Constracter) genannt. Sie beinhalten alle die Attributs-Festlegungen, die für ein neues Objekt gemacht werden müssen.
Zum Anderen müssen Instanzen auch mal wieder sauber aus dem System entfernt werden können. Dazu dienen Destruktoren.
Für Konstruktoren wird statt dem Ausrufezeichen ein "C" notiert. Somit entsteht die folgende (erste) Klassen-Definition für die Objekt-Klasse "Hund".

Hund		Klassen-Name
+ Name: String + Rasse: String + Steuernummer: Integer + Alter: Integer + Masse: Float - Wurmbehandlung: Boolean		Attribute
+C neu(Name: String, Steuernummer: Integer) +! gib_Laut(anzahl: Integer) +! läuft_nach(xpos, ypos: Float) +! hat _Geburtstag() +! bekommt_Steuernummer(Nummer: Integer) +! setze_Wurmbehandlung(): Boolean +? zeige_Wurmbehandlung(): String		Methoden

Die Methode "neu" wird ohne Datentyp verzeichnet. Das liegt daran, dass das Ergebnis diesen Methoden-Aufrufs keine Daten eines oder mehreren Types sind, sondern es wird eine Instanz von "Hund" zurückgeliefert. Natürlich können – und müssen wahrscheinlich auch – diverse Initialisierungs-Attribute mitgegeben werden.
In anderen UML-Notierungen wird statt dem Schlüssel-Wörtchen "neu" die Name der Klasse als Konstruktor geschrieben. Meist verzichtet man dann auch auf das "C" bei der Klassifizierung der Methode. Für unser Hund-Beispiel sähe das dann so aus:

Hund		Klassen-Name
…		Attribute
+ **Hund**(Name: String, Steuernummer: Integer) … …		Methoden

Die Umsetzung der Klassen-Diagramme ist eine klar definierte und stupide Tätigkeit, die wir dem Computer überlassen können. Viele UML-Werkzeuge leisten das auch.
Schauen wir uns eine mögliche Umsetzungen des Attributs-Teils in Python und Java an.

Python	Java

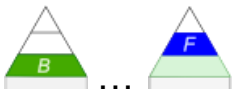
class Hund:
  def __init:
  Name=""
  Rasse=""
  Steuernummer=0
  Alter=-1
  Masse=0
  Wurmbehandlung=False
 	
…	

Mit den Methoden verfährt das UML-System genauso. Nach Schema F werden die Rahmen für die Funktionen erstellt:

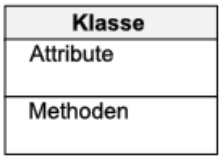
Python	Java

class Hund:
  def __init:
  Name=""
  Rasse=""
  Steuernummer=0
  Alter=-1
  Masse=0
  Wurmbehandlung=False
 	
def gib_Laut(anzahl):
  pass
 	
def laeuft_nach(xpos, ypos):
  pass
 	
def hat_Geburtstag:
  pass
 	
def bekommt_Steuernummer(nummer):
  pass
 	
def setze_Wurmbehandlung(code):
  self.Wurmbehandlung=code
 	
def zeige_Wurmbehandlung:
  return self.Wurmbehandlung	

2.1.2. UML-Klassen-Beziehungen



vereinfachtes Klassen-Symbol



Definition(en): Assoziation
Eine Assoziation ist eine Beziehung zwischen zwei Objekten.

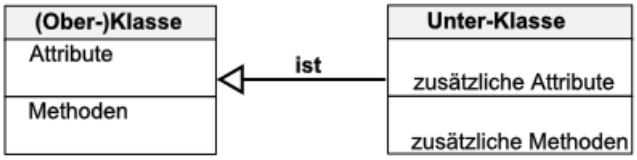
Definition(en): Multiplizität
Die Multiplizität beschreibt den Bereich der erlaubten Kardinalitäten zwischen zwei Objekten.

Definition(en): Kardinalität
Die Kardinalität beschreibt die Anzahl der zulässigen Objekte (z.B. in einer Assoziation).

2.1.2.1. IST-Beziehung / Vererbung / Generalisierung

Generalisierung

auch **"ist_ein(e)"** oder **"ist_zugeordnet_zu"** Verbindung / Beziehung



Beispiele:

Fahrzeug	⟵	PKW	*PKW*	***ist***	*(ein)*	*Fahrzeug*
Fahrzeug	⟵	Dampflok	*Dampflok*	***ist***	*(ein)*	*Fahrzeug*
Fahrzeug	⟵	Bus	*Bus*	***ist***	*(ein)*	*Fahrzeug*
Tier	⟵	Vogel	*Vogel*	***ist***	*(ein)*	*Tier*
Tier	⟵	Affe	*Affe*	***ist***	*(ein)*	*Tier*

intuitiv verständlicher ist die umgekehrte Pfeil-Darstellung:

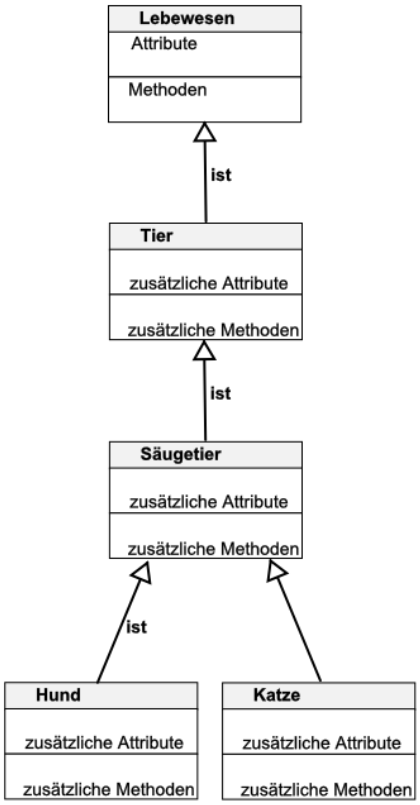
PKW	⟶	Fahrzeug
Dampflok	⟶	Fahrzeug
Bus	⟶	Fahrzeug
Vogel	⟶	Tier
Affe	⟶	Tier

Generalisierungen können sich auch mehrstufig / über mehrere Ebenen ergeben:

Mensch ⟶ Menschenaffe ⟶ Affe ⟶ Säugetier ⟶ Tier
Dampflok ⟶ Lok ⟶ Schienenfahrzeug ⟶ Fahrzeug

in einem UML-Diagramm ergeben sich dann Ketten von Klassen

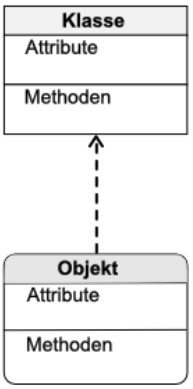
da auch mehrere Klassen wieder zu einer Klasse gehören können treten in vielen Beziehungs-Darstellungen dann auch Verzweigungen auf.



böse Fragen zwischendurch:

1. *Könnnen in UML-Diagrammen eigentlich auch Ringe (Schleifen, Wiederholungen, …) auftreten?*
2. *Könnte eigentlich auch eine direkte IST-Beziehung von "Katze" nach "Lebewesen" aufgemacht werden?*

Bisher sprachen wir nur über Klassen, also Gruppen von Objekten. Irgendwann kommen wir bei dem Aufstellen von Beziehungen dann aber auch in der Ebene der Objekte an.



Realisierung / Instanzierung
mit gestrichelter Verbindungs-Linie

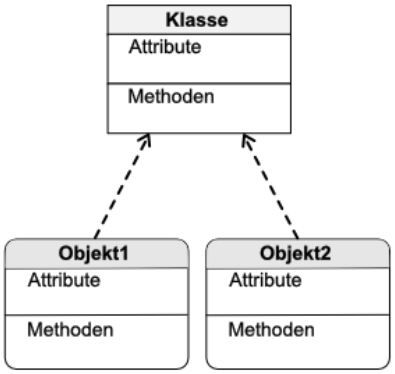
in einigen Darstellungssystemen bekommen Objekte bei ihren Rechtecken runde Ecken
sachlich reicht ein (eckiges) Rechteck mit einem gestrichelten Pfeil zur Objekt-Kennzeichnung aus.

Beispiele:

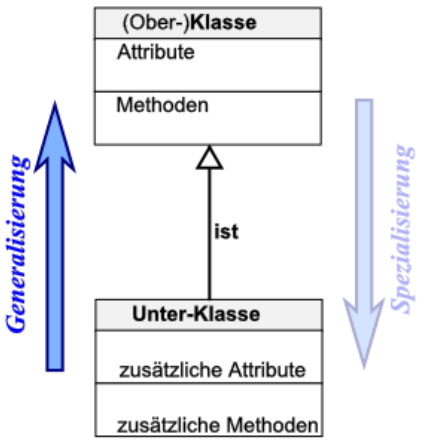
Elephant	⊲----- "Dumbo"	"Dumbo"	***ist** (ein spezieller)*	*Elephant*
Bär	⊲----- "Puh"	"Puh"	***ist** (ein spezieller)*	*Bär*
Beruf	⊲----- "Maurer"	"Maurer"	***ist** (ein spezieller)*	*Beruf*
Beruf	⊲----- "Politiker"	"Politiker"	***ist** (ein spezieller)*	*Beruf*
geom. Figur	⊲----- "Rechteck"	"Rechteck"	***ist** (ein spezielle)*	*geom. Figur*

Wir sehen hier schon, dass die Individualisierung Grenzen hat. Die Frage, ob "Maurer" als ein Objekt oder doch als Klasse zu betrachten ist, bestimmt das Modell unserer Miniwelt. Je nach Verwendungs-Zweck können zwei Dinge in verschiedenen Modellen unterschiedliche , aber eben auch gleiche Sachverhalte oder Hierarchie-Ebenen beschreiben.

von einer Klasse lassen sich üblicherweise mehrere Objekte ableiten / instanzieren oder mit anderen Worten: Mehrere (vergleichbare und zusammengehörende) Objekte gehören zu einer Klasse.



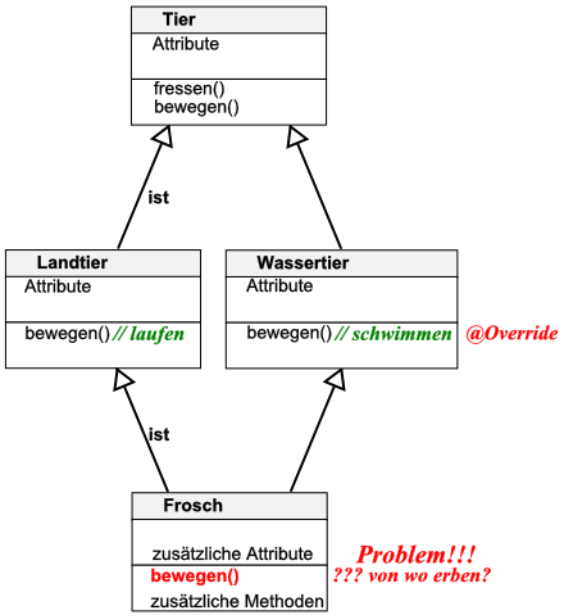
Sowohl bei der Generalisierung von Klassen als auch bei der Instanzierung von Objekten, bekommen die untergeordneten Klassen oder Objekte immer Eigenschaften (Attribute) und Fähigkeiten (Methoden) übertragen. Das sind ja genau die, die sie zu einer Klasse zugehörig machen.
Je tiefer die Ebenen, umso individualisierende Attribute und Methoden haben die Klassen oder Objekte.
In der Informatik sagen wir, die (untergeodneten) Klassen oder Objekte erben von ihren übergeordneten Klassen.



Definition(en): Vererbung
Unter Vererbung versteht (in der Informatik) die Umsetzungs-Mechanismen für Beziehungen (Relationen) zwischen Klassen verschiedener Hierrachie-Ebenen (Ober- und Unter-Klassen). Dabei werden i.A. Attribute und Methoden der Ober-Klasse auch den Unter-Klassen verfügbar gemacht.
Vererbung ist ein grundlegendes Prinzip der Objekt-orientierten Programmierung, bei dem aus Basis-Klassen (übergeordneten / allgemeineren Klassen) abgeleitete (untergeordnete / speziellere) Klassen erstellt werden.

Mehrfach-Vererbungen sind i.A. problematisch, da es zu Konflikten hinsichtlich der geerbten Attribute und Methoden aus den Oberklassen kommen kann

hier müssen die problematischen (/ doppelten) geerbten Methoden und Attribute auf der untergeordneten Ebene neu festgelegt werden. Die Vererbung wird unterbrochen. Wir sagen die Attribute bzw. Methoden werden überschrieben – praktisch neu definiert.



Aufgaben:

1. ***Prüfen Sie, ob die nachfolgenden Beziehungen als Klassen- bzw. Objekt-Beziehungen korrekt sind! Begründen Sie Ihre Entscheidung!***

   a) Lebewesen ⟵ Pflanze ⟵ Samenpflanzen ⟵ Tulpen
   b) chem.Element ⟶ Reinstoff ⟶ Stoff
   c) Sonne ⟶ Sonnensystem ⟵ Planeten ⟵ Gasplaneten
   d) Fleischfresser ⟶ Wölfe ⟵ Säugetiere
   e) Stoff ⟶ Körper ⟶ Material
   f) Quadrat ⟶ Paralleogramm ⟶ Rechteck ⟶ geom.Figur ⟶ Vieleck
   g) Fleischfresser ⟶ Wölfe ⟵ Säugetiere
   h) Hunde ⟵ Landtiere ⟶ Säugetiere ⟵ Lebewesen ⟶ Materie

2.

2.1.2.2. KENNT-Beziehung

beide Seiten der Verbindung können mit Rollen-Namen (Rollen) versehen werden

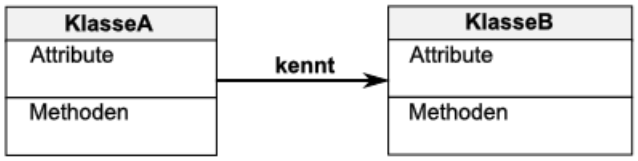
Beispiele:

Warenkorb	⟶	Artikel	*Warenkorb*	***kennt***	*Artikel*
Schüler	⟶	Lehrer	*Schüler*	***kennt***	*Lehrer*
Lehrer	⟶	Schüler	*Lehrer*	***kennt***	*Schüler*
Kapitän	⟶	Schiff	*Kapitän*	***kennt***	*Schiff*
Offizier	⟶	Schiff	*Offizier*	***kennt***	*Schiff*
Offizier	⟶	Waffe	*Offizier*	***kennt***	*Waffe*
Polizist	⟶	Waffe	*Polizist*	***kennt***	*Waffe*
Krimineller	⟶	Waffe	*Krimineller*	***kennt***	*Waffe*

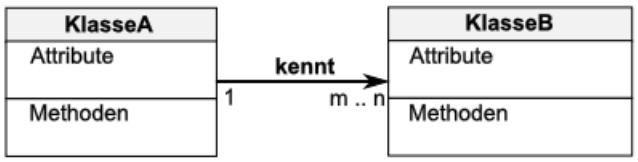
Rolle	**kennt**	**Klasse / Objekt**
Funktion	**kennt**	**Klasse / Objekt**
Klasse	**kennt**	**Klasse / Objekt**
Objekt	**kennt**	**Klasse**
Objekt	**kennt**	**Objekt**
Verwendungs-zweck	**kennt**	**Klasse / Objekt**

vor allem für den Nachrichten-Austausch wichtig
Artikel teilt dem Warenkorb mit, dass nur noch drei auf Lager sind

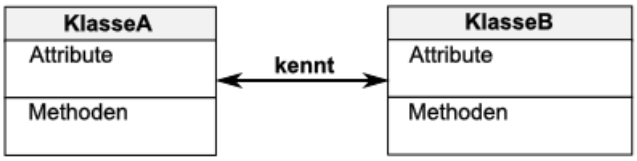
nur in einer Richtung navigierbar (benutzbar)
Andere Formen der Navigabilität sind die Unbestimmtheit, das Navigations-Verbot und die beidseitige Navigierbarkeit / Kenntnis.



mit Multiplizität / Kardinalität
Anzahl bis Bandbreite (Minimum .. Maximum)
ist das Minimum 0, dann ist die Verbindung optional

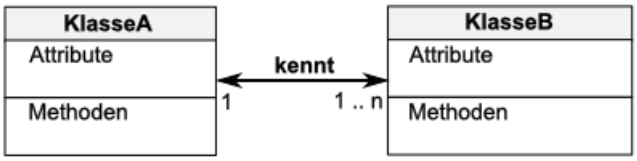


Kapitän	⟶	Schiff	*Kapitän*	***kennt***	*Schiff*
Offizier	⟶	Schiff	*Offizier*	***kennt***	*Schiff*



Schüler	⟶	Lehrer	*Schüler*	***kennt***	*Lehrer*
Lehrer	⟶	Schüler	*Lehrer*	***kennt***	*Schüler*

mit Multiplizität / Cardinalität



Assoziation

Konto ------- Kunde

Varianten der Assoziation sind die Aggregation und die Komposition

Definition(en): Assoziation
Assoziationen sind Beziehungen (Menge von Objekt-Verbindungen) zwischen Klassen mit gemeinsamer Struktur und Semantik.

Navigation beschreibt Erreichbarkeit / Sichtbarkeit / Kenntnisgrad einer Beziehung

unbestimmte Navigation
es wird keine Aussage zur Navigierbarkeit gemacht

wenn eine Instanz des einen Objektes oder die Klasse existiert, dann kann auch eine Beziehung zur anderen Klasse oder einer Instanz eines anderesartigen Objektes bestehen
beide Objekte / Klassen entstammen i.A. eigenständigen Klassen-Hierarchien

Beispiele:
Konto ------- Kunde

erlaubte Navigation
von dem einen (instanzierten) Objekt oder einer Klasse darf eine Beziehung zur anderen Klasse oder einem entsprechenden Objekt aufgebaut und genutzt werden

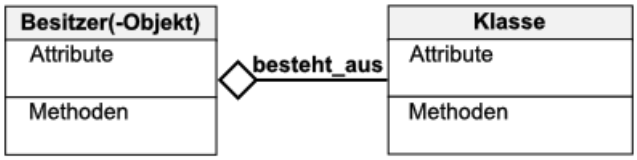
Beispiele:
Fahrer ------> Auto

verbotene Navigation
es existieren zwar Beziehungen, aber eine Nachrichten-Übertragung, Sichtbarkeit usw. ist nicht gegeben

Beispiele:
Abteilung —⊬— Angestellter

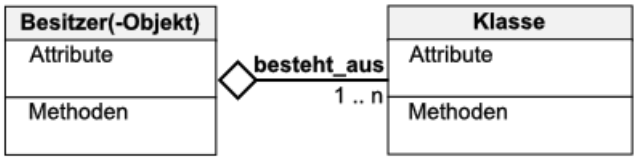
2.1.2.3. BESTEHT_AUS-Beziehung / Aggregation

"besteht_aus" deutet schon das Zusammenstellen aus Teilen hin. Diese Teile sind i.A. eigenständig und existieren auch, wenn das übergeordnete Teil (als Zusammenstellung) nicht mehr existiert. Deshalb spricht man auch von einer (schwachen) Aggregation.



Teile werden zu einem Ganzen zusammengefasst / zusammengestellt

z.B.:
Auto-Teile eines Auto's
Personen einer Familie
Mobiliar / Ausstattung eines Hotel's
Personal einer Gaststätte
Schüler in einer Schulklasse



Definition(en): Aggregation
Eine Aggregation ist eine Beziehung zwischen Objekten, bei die untergeodneten Objekte auch unabhängig von den übergeordneten Objekten existieren können.
Aggregationen sind Assoziationen, dernen Klassen Ganzes-Kleines-Hierarchien darstellen.

Aggregationen sind die normalen Assoziationen in JAVA

stärker als Assoziation
assoziert Besitz

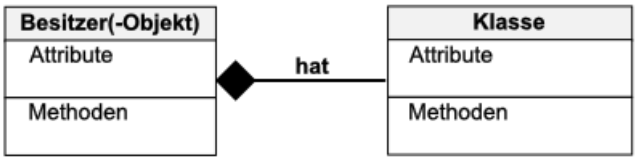
Beispiele:

Auto	◇——	Fahrer	(*Auto besitzt einen Fahrer*)
Restaurant	◇——	Gast	(*Restaurant besitzt einen Gast*)
Mannschaft	◇——	Spieler	(*eine Manschaft ist zusammengesetzt aus Spieler(n)*)
Team	◇——	Mitglieder	(*ein Team hat Mitglieder*)

i.A. ist die Unterscheidung zwischen Assoziation und Aggregation eher schwierig

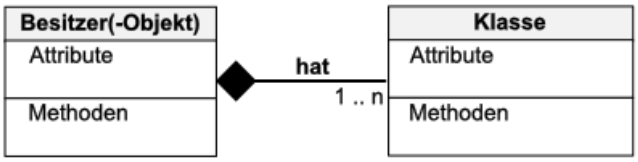
2.1.2.4. HAT-Beziehung

"Hat", "hat_ein(e)" "hat_eine_Beziehung_zu" ist eine starke Aggregation – besser als Komposition bezeichnet. Die Zusammenstellung hat nur einen Zweck, wenn



Entfernt man die (übergeordnete) Zusammenstellung, dann hat das untergeordnete Teil keinen Sinn mehr. Eigenständig kann es nicht existieren.

z.B.:
Räume eines Gebäudes



Beispiele:

Gebäude	◆——	Raum	(*ein Gebäude enthält Räume*)
Mensch	◆——	Herz	(*zum Mensch gehört ein Herz*)
Arm	◆——	Muskel	(*der Arm ist aus Muskel(n) aufgebaut*)
Buch	◆——	Kapitel	(*ein Buch beinhaltet ein Kapitel*)
Regierung	◆——	Minister	(*die Regierung hat Minister*)

Definition(en): Komposition
Eine Komposition ist eine Beziehung zwischen Objekten, bei der das untergeordnete Objekt nicht ohne das übergeordnete Objekt existieren kann. Im Falle des Löschens des übergeordneten Objektes werden die an der Komposition beteiligten (Unter-)Objekte / Bestandteile mit gelöscht.
Eine Komposition ist eine Zusammenstellung von Objekten (Teilen), deren Existenz vom übergeordneten Objekt (/dem Ganzen) abhängig ist.

Komposition ist strenge Form der Aggregation
Kardinalität auf der Seite des übergordneten Objektes (Aggregat-Seite; Kompositions-Objekt) ist immer 1
die untergeordneten Objekte (Teile) sind immer genau von einem (1) übergeordneten Objekt (Kompositions-Seite)
die Existenz / Lebensdauer der Komponenten / Einzelteile ist dem Ganzen / der Komposition (als Objekt) untergeordnet

eine Komposition kann in der Programmiersprache JAVA nicht direkt abgebildet werden

2.1.2.5. Annotation / Paketierung

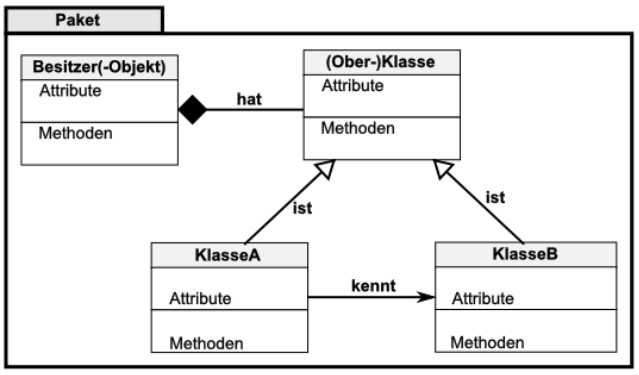
Annotationen oder Notizen dienen der Erläuterung

gestrichelte Verbindungslinie zu einem "Zettel" mit "Eselsecke"

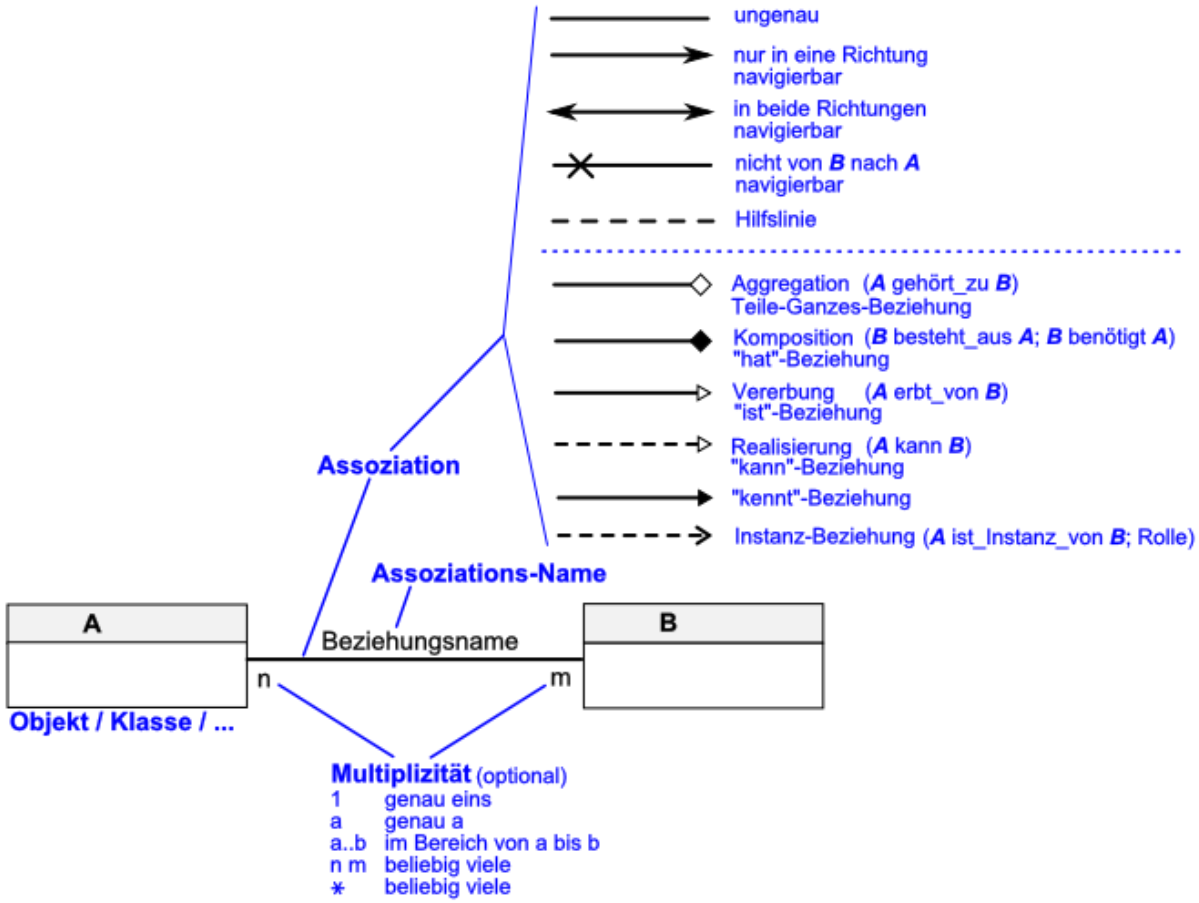
keine semantische Bedeutung

mehrere Klassen werden zu Paketen zusammengefasst
Strukturierung
Verringerung der konkreten / aktuellen (Ebenen-)Komplexität
Steigerung der Übersichtlichkeit
Modularität (Austauschbarkeit)

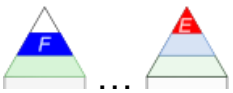
Pakete bekommen umschreibenden Rechteck-Rahmen mit einem Reiter (links oben), der die Bezeichnung des Paketes enthält



Übersicht / Legende zu UML-Diagrammen:



2.1.3. UML-Anwendungsfall-Diagramme - Use Case



stellt aus der Sicht des Anwenders das erkennbare, äußere Systemverhalten
Beschreibung der Zusammenhänge zwischen den Anwendungsfällen und zwischen den Akteuren und dem Anwendungsfall
Akteure als Strichmännchen; die Anwendungsfälle als Ovale
zeigt Akteure (Handlungs-Objekte), Struktur und zusammenhänge zwischen verschieden Geschäfts- bzw. System-Vorfällen und den resultierenden notwendigen Reaktionen (Verfahren)

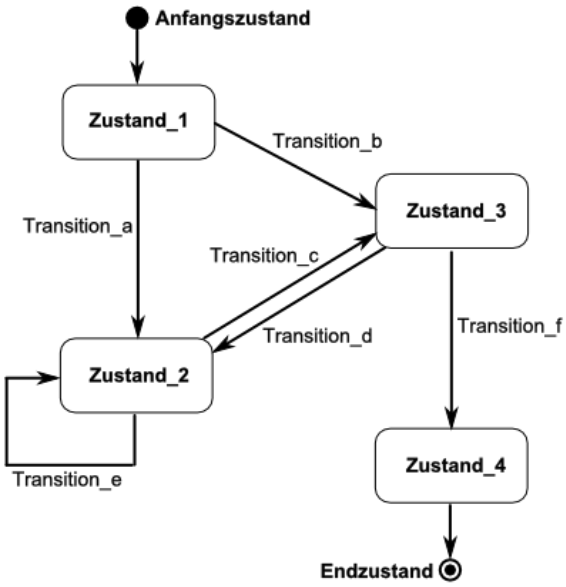
Definition(en): Anwendungsfall-Diagramm
Anwendungsfall-Doagramme stellen zusammengehörende Mengen von Anwendungsfällen und den Beziehungen zwischen den Akteuren und / oder Anwendungsfällen dar.

2.1.4. UML-Sequenz-Diagramme

Definition(en): Sequenz-Diagramm
Sequenz-Diagramme stellen die zeitliche Abfolge / Reihenfolge von Interaktionen zwischen Objekten dar.

2.1.5. UML-Zustands-Diagramme

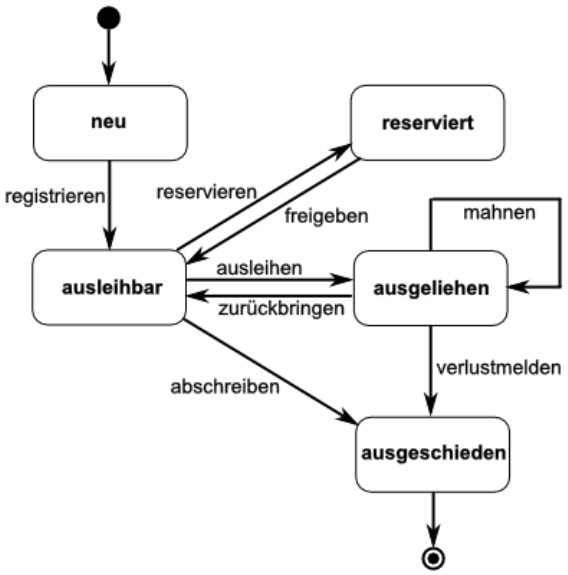
betrachtet wird eine hypothetische Maschine, die sich über eine definierte Menge von Zustanden beschreiben lässt
Maschine hat zu jeder Zeite einen der möglichen Zustände
es gibt einen Anfangs- bzw. Ausgangs-Zustand
die Menge der Ereignisse ist definiert
es gibt einen oder mehrere End-Zustände



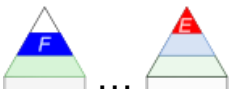
Zustände werden als abgerundete Rechtecke gezeichnet
Pfeile geben die möglichen Aktivitäten (Transitionen) an
es gibt einen Anfangszustand und einen oder mehrere Endzustände

Definition(en): Zustands-Diagramm
Zustands-Diagramme visualisieren die Zustände eines Objektes und deren Übergänge (Zustandsänderungs-Funktionen) mit ev. notwendigen Eingaben (Bedingungen, Ereignissen) und Ausgaben.

mehr zu den Zustands-Diagrammen besprechen wir bei den Automaten (→ 3.2.3.2. UML-Zustands-Diagramme), da hier ein wichtiges Einsatzgebiet liegt



2.1.6. UML-Aktivitäts-Diagramme



Aktivitäten wie im Zustands-Diagramm als abgerundete Rechtecke
an die Übergänge werden die Bedingungen oder Eingaben etc. in eckigen Klammern notiert
zusätzliche Verzeigungen / Prüfungen werden durch Rhomben im Verlauf einer splittenden Aktivität mit mind. 2 zusätzlichen Bedingungen angegeben

mehrere (zusammengehörende) Aktivitäten können wiederum in einem abgerundeten Rechteck gruppiert werden
Objekt-Flüsse werden als Pin's (kleine Quadrate am Anfang und Ende der Pfeile (Kontroll-Flüsse) notiert
es gibt einen Start-Knoten (Anfangs-Kontrollknoten)

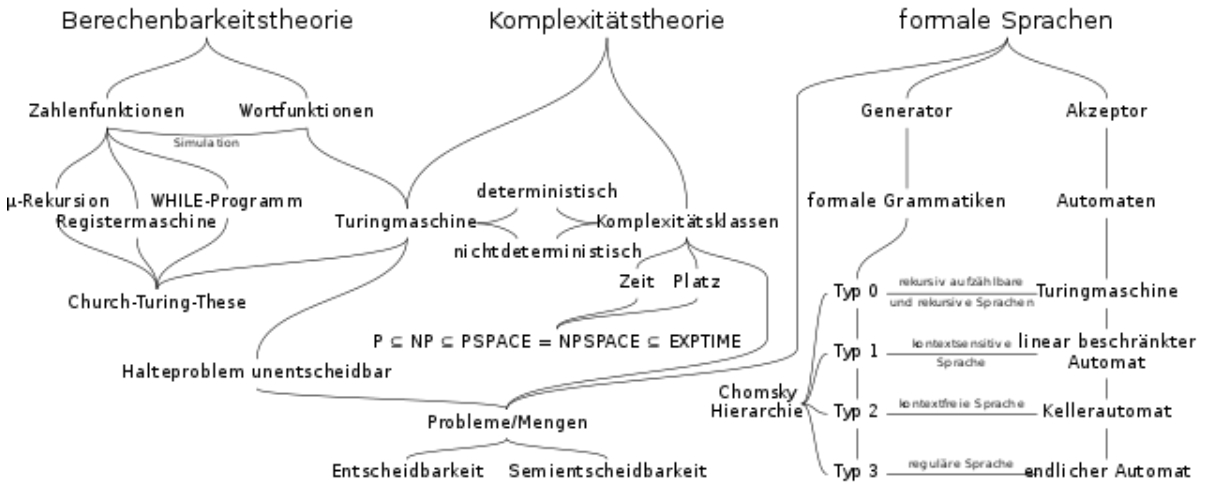
Definition(en): Aktivitäts-Diagramm
Aktivitäts-Diagramme stellen die Aktivitäten und deren Bedingungen für die Übergänge zu den nächsten Aktivitäten dar.

3. formale Sprachen und Automaten

Problem-Fragen für Selbstorganisiertes Lernen
Was ist Automat? Mit welchen Automaten beschäftigt sich die Theoretische Informatik? Was sind Sprachen? Welche Arten von Sprachen gibt es? Wie kann man Sprachen sinnvoll einteilen? Mit welchen Sprachen beschäftigt sich die Theoretische Informatik? Was sind CHOMSKY-Sprachen und welchen Typ haben sie? Was ist eine Grammatik? Haben Grammatiken und Automaten Beziehungen innerhalb der Theoretischen Informatik? Welche Arten von Automaten (in der Theoretischen Informatik) gibt es? Wie lassen sich Algorithmen beschreiben? Was haben Compiler und Interpreter mit formalen Sprachen und Automaten zu tun? Ist alles berechenbar? Was tun und können MEALY-, MORRE- und TURING-Automaten? Wie komplex werden Berechnungen? Kann man alles in absehbarer Zeit berechnen? Hört jeder Rechner / jeder Algorithmus irgendwann mal auf? Welche allgemeinen Daten-Strukturen gibt es? Mit welchen allgemeinen Methoden kann man auf Daten zugreifen oder sie verändern.

Theorie der (formalen) Sprachen und Automaten-Theorie

Bestandteil der Theoretischen Informatik



Übersicht über wichtige Inhalte der Theoretischen Informatik
Q: de.wikipedia.org (Paeng)

Links:
Hypertext-System zu Themen der Theoretischen Informatik (Sprachen, Grammatiken, Automaten):
→ http://ddi.cs.uni-potsdam.de/Forschung/SIMBA/export/mod-lklass/einleitung2.htm

Definition(en): natürliche Sprache(n)
Natürliche Sprachen sind die von Menschen – historisch gewachsenen – gesprochenen, geschriebenen oder gebärdeten Sprachen, mit einem offenem Regelwerk.
Natürliche Sprachen sind Sprachen, die sich aus variablen und diffenzierten Wort-Schätzen, und unterschiedlich interpretierbaren Wort-Konstrukten zusammensätzen. In natürlichen Sprachen sind auch nicht-regelkonforme Sätze oder Wort-Konstrukte erlaubt.

Bestandteile / Aspekte einer natürlichen Sprache

• **Semantik**	Bedeutung, Sinngehalt	***sehr variabel***
• **Phonetik**	Ausdruck	***vielgestaltig***
• **Syntax**	Grammatik, Rechtschreibung	***Orientierungs-rahmen***
• **Kontext**	Umwelt, Sinnzusammenhang, Situation	***sehr variabel***
• **Pragmatik**	Handlungs-Bezug, Aufträge, Anweisungen	***sehr variabel***

Beispiel:
Auto mit Frau als Fahrerin und ihr Mann als Beifahrer stehen an einer roten Ampel. Als es grün wird, sagt der Mann: "Schatz, es ist grün."

Aufgaben:

1. *Welche Interpretationen, … des obigen Satzes kennen Sie? Versuchen Sie die Stimmung, die Intonation und die Situation möglichst gut wiederzugeben!*
2. *Beobachten Sie innerhalb von 24 Stunden den Sprachgebrauch in Ihrer Umgebung! Notieren Sie sich Aussprüche usw., die unterschiedlich interpretiert, … wurden oder es hätten sein können!*
3.

Bedeutung / Interpretation / …

- **informativ** — einfache Mitteilung
- **informativ, überheblich** — Mitteilung mit Unterton: Hast Du das schon wieder nicht gesehen?
- **informativ, auffordernd** — Mitteilung mit Aufforderung: "Fährst Du jetzt endlich los!"
-
- **…** — …

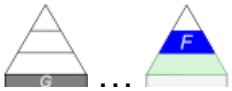
Veränderungen stehen vor der Tür.
Lasse sie ruhig zu.
(aus einem chinesischen Glückskeks)

Change is happening in your life.
So go with the flow!
(aus einem chinesischen Glückskeks)

Semantik eines Wortes oder einer kleinen Wort-Kombination ist schwer (für Computer-Systeme) fassbar
der Teelöffel und einTeelöffel Salz

Definition(en): künstliche oder konstruierte Sprachen
Eine künstliche Sprache ist eine Sprache, die sich durch ein eindeutiges, geschlossenes Regelsystem (Grammatik) auszeichnet.
Eine künstliche Sprache ist ein von Menschen erstelltes Konstrukt aus Inhalten und Regeln, um eine Kommunikation in einem bestimmten Bereich mit bestimmten Zielen zu gestalten.

3.0. Grundbegriffe und ein Einführungs-Beispiel



Problem-Fragen für Selbstorganisiertes Lernen
Was sind Sprachen? Was charakterisiert Sprachen? Wozu dienen Sprachen? Welche Arten von Sprachen gibt es? Wie kann man Sprachen sinnvoll einteilen? Mit welchen Sprachen beschäftigt sich die Theoretische Informatik? Was sind CHOMSKY-Sprachen und welchen Typ haben sie? Was ist eine Grammatik? Haben Grammatiken und Automaten Beziehungen innerhalb der Theoretischen Informatik? Welche Arten von Automaten (in der Theoretischen Informatik) gibt es? Kann man alles mit Sprachen mitteilen? Gibt es Sprachen, die nicht gesprochen werden?

Sprach-Bestandteile

- **Syntax**
  - **Rechtschreibung** — exakte Notierung (od.ä.) der Symbole und deren Kombination (zu Worten)
  - **Grammatik** — exakte – die Semantik unterstützende – Notierung der Symbole und Worte zu in der Kommunikation nutzbaren Ausdrücken (Sätzen)
- **Pragmatik** — Handlungs-Hinweise / -Aufträge
Problem- / Frage-Stellung
- **Semantik** — eigentlicher (ev. auch mehrdeutiger) Inhalt
  - **Lexik** — Wortsammlung mit zugeordneter/n Bedeutung(en)

Syntax, Semantik und Pragmatik sind drei Haupt-Elemente der Semiotik (Lehre von den Zeichen-Systemen)

Zeichen bilden Wörter. Wörter bilden Sätze.
"H" + "a" + "u" + "s" → "Haus"
"Ein Haus hat ein Dach."

Sätze können syntaktisch richtig sein, aber semantisch keinen Sinn ergeben.
"Ein Haus sieht einen Koffer."
"Veganer essen täglich Fleisch."

Die Semantik ist schwer zu formulieren / zu umschreiben und ist häufig mehrdeutig; schwingt häufig in der Kommunikation mit.

Analyse-Möglichkeiten

• **syntaktische Analyse**	recht gut möglich, da Rechtschreibung und Grammatik oft bekannt sind und entweder feststehend sind oder als Orientierungsrahmen bekannt sind
• **lexikalische Analyse**	weitesgehend klar; nur Wort-Neuschöpfungen oder neue Bedeutungs-Zuordnungen müssen geklärt werden
• **grammatikalische Analyse**	meist sehr gut möglich, da die Grammatik sehr stark die klare Kommunikation bestimmt Grammatik-Regeln in Sprachen meist sehr domininierend
• **semantische Analyse**	schwierig, da häufig pragmatische, situative und phonetische Informationen fehlen oder nicht eindeutig sind

Grammatik einer ((sehr, sehr) einfachen, deutschen) Sprache / Kleinkinder-Sprache

Regeln:

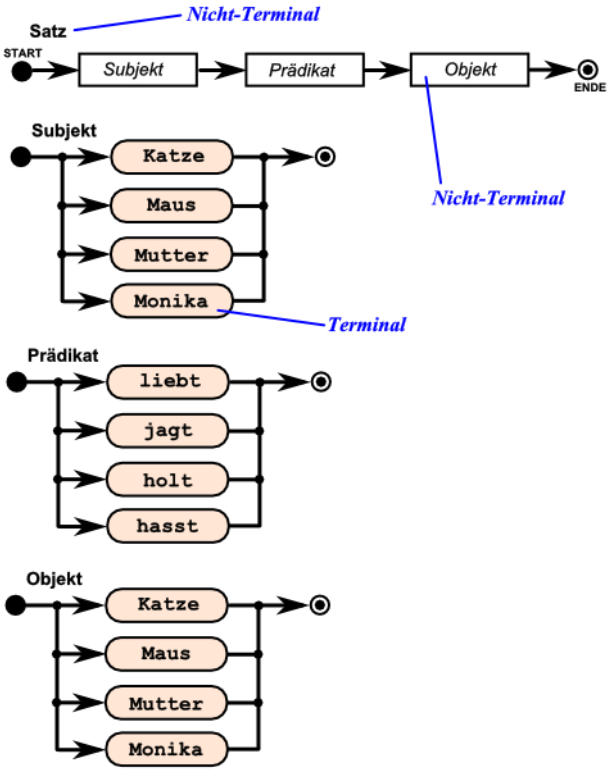
```
<Satz>          →  <Subjekt>  <Prädikat>  <Objekt>
```

Basiselemente / Atome:

```
<Subjekt>       →  "Maus"  |  "Mutter"  |  "Monika"  |  "Katze"
<Prädikat>      →  "liebt"  |  "jagt"  |  "holt"  |  "hasst"
<Objekt>        →  "Maus"  |  "Mutter"  |  "Monika"  |  "Katze"
```

Eine Regel beschreibt den Aufbau eines Sprach-Elementes. In unserem Beispiel sind es die Satzglieder **Subjekt**, **Prädikat** und **Objekt**. Die einfache Hintereinanderschreibung (Aufreihung) besagt, dass die Elemente in dieser Reihenfolge notiert werden müssen. Somit ist auch festgelegt, dass jeder der Sätze in der "sehr, sehr einfachen deutschen Sprache" aus genau drei Worten bestehen muss, eben einem **Subjekt**, einem **Prädikat** und einem **Objekt**. Wie genau diese Elemente aussehen, wird in weiteren Regeln beschrieben. Das können weitere Regeln zum Zusammenstellen eines Satzes sein oder eben Regeln mit gültigen Subjekten, Prädikaten bzw. Objekten. Solche elementaren Basis-Teile werden oft auch Atome genannt. Gibt es mehrere, dann können sie i.A. alternativ ausgewählt werden. In den obigen Regeln nutzt man den senkrechten Strich als Kennzeichen für Alternativen. Im Sprach-Beispiel sind das z.B. die Subjekte **Maus**, **Mutter**, **Monika** und **Katze**, die alternativ ausgewählt werden können.

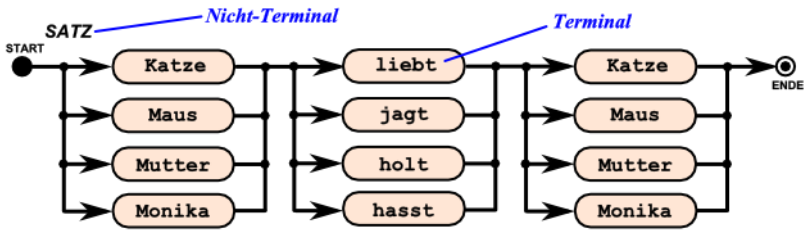
Während die grammatikalischen Elemente (**Subjekt**, **Prädikat** und **Objekt**) nur in der Grammatik selbst vorkommen, sind die Atome die Elemente, die wirklich im Satz vorkommen. Sie sind quasi das letzte – abschließende – Glied der Regeln und werden deshalb als Terminale bezeichnet.
Die beschreibenden (grammatikalischen) Elemente werden Nicht-Terminale genannt.
Da die textlichen Regeln recht schnell unübersichtlich werden, nutzt man gerne sogenannte Syntax-Diagramme (siehe Abb. rechts), um eine Sprache (oder deren Elemente) zu definieren und darzustellen. Die Symbolik wird später noch genauer erklärt und auf den heutigen Standard reduziert.
Die Strukturierung und Definition der Nicht-Terminale folgt eigentlich vorrangig den Anforderungen. In unserem Fall könnte man Subjekt und Objekt als ein Nicht-Terminal ansehen und es z.B. Substantiv nennen. Die Regel für den Satz würde dann lauten:




```
<Satz>          →  <Substantiv>  <Prädikat>  <Substantiv>
```

Natürlich lassen sich auch immer weniger oder mehr Terminale in die Definitionen einbauen. Minimal ist allerdings jeweils eins notwendig.

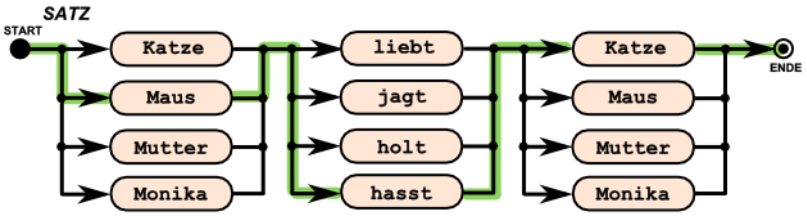
Grundsätzlich lässt sich unsere Grammatik auf ein Nicht-Terminal reduzieren. Hier ist das auch noch ausreichend übersichtlich. Bei großen Sprach-Definitionen, wie z.B. eine Programmier-Sprache, würde das aber zu unübersichtlich werden.



Kleine Nicht-Terminale lassen sich auch später besser verwalten, programmieren (beim Compiler- oder Interpreter-Bau), testen, verändern und dokumentieren.

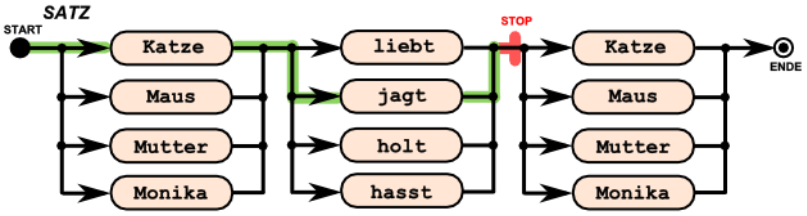
Die Syntax-Diagramme lassen auch die Prüfung der Gültigkeit eines vorgegebenen Satzes zu.

Nehmen wir an, für den Satz "**Maus hasst Katze**" soll geprüft werden, ob er Teil der "sehr sehr einfachen deutschen Sprache" ist. Dafür ist es nur notwendig zu kontrollieren, ob es einen Weg (entlang der Pfeile) vom START zum ENDE gibt.



Wie man schnell erkennt, existiert ein solcher Weg. Damit ist der Satz gültig.

Anders sieht das im Fall von "Katze jagt Hund" aus. Hier bleibt der Test für das Objekt (bzw. das 2. Substantiv) stecken. Es gibt kein Terminal "Hund". Somit ist der Satz kein gültiger Bestandteil unserer "sehr, sehr einfachen deutschen Sprache".



Es ändert sich auch nicht, wenn man die offensichtlich wahre Semantik des Satzes mit einbezieht. Die spielt in unseren Betrachtungen zuerst einmal keine Rolle.

mögliche Regel-Erweiterungen:

`<Satz>` → `<Subjekt>` `<Prädikat>`

Für Nutzer und Freunde der Programmiersprache PROLOG sei hier eine einfach Umsetzung angegeben:

	Quellcode	Kommentar(e)
1 2 3	`% Autor: lsp: dre` `% Datum: 09.2015`	%-Zeichen sagt: es folgt ein Kommentar; hier Autor und in der nächsten Zeile das Erstellungsdatum
4	`satz(X,Y,Z):-subjekt(X), praedikat(Y), objekt(Z).`	ist eine Regel mit dem Namen "satz", die besagt, ein Satz besteht aus drei Teilen (X,Y,Z) und z.B. soll X ein Subjekt sein
5 6 7	`subjekt(X):-substantiv(X).` `objekt(X):-substantiv(X).`	subjekt ist ein Substantiv objekt ist ein Substantiv

8
9
10
11
12
13
14
15
	`praedikat(liebt).`
	`praedikat(jagt).`
	`praedikat(folgt).`
	`praedikat(hasst).`
	`substantiv(maus).`
	`substantiv(mutter).`
	`substantiv(monika).`
	`substantiv(katze).`
		hier erfolgt die Definition der verfügbaren Worte der Sprache

Wie Sie sicher bemerkt haben, wird hier eine der besprochenen Vereinfachungen (Umstrukturierungen) benutzt.

Aufgaben:

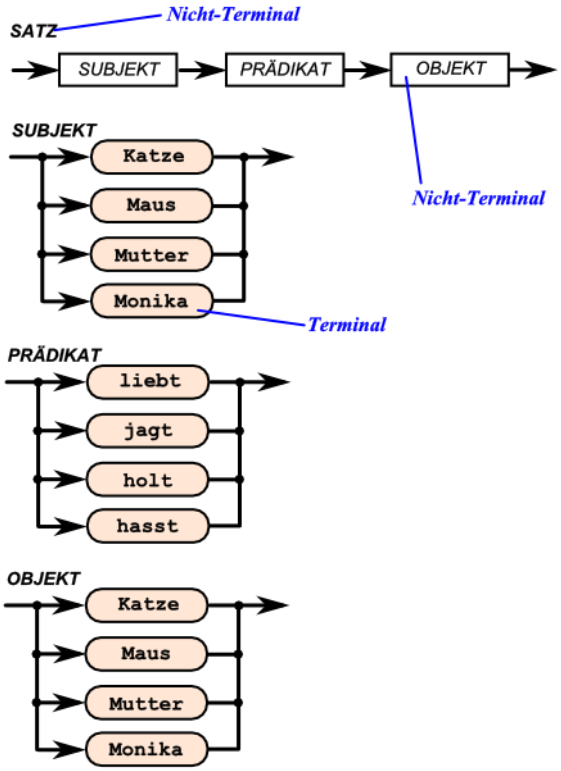
1. ***Geben Sie das obige Programm in ein PROLOG-System ein (z.B. SWI-Prolog vom IoStick)!***
2. ***Überlegen Sie sich, ob die folgenden einfachen deutschen "Sätze" durch die Prüfung (durch unser Programm) kommen könne!***
3. ***Testen Sie die "Sätze" im PROLOG-System!***
4. ***Erweitern Sie die Defintionen (Atome) um weitere Worte! Testen Sie jeweils neu mögliche Sätze! Testen Sie auch immer nicht-zulässige Sätze!***

für die gehobene Anspruchsebene:

5. ***Erweitern Sie den Regel-Teil um weitere Konstrukte (zur Verbesserung der Sprache)! Geeignet sind z.B. die Objekte "Artikel" und / oder "Adjektiv". Ihrer "Kreativität" sind dabei nur die Regeln der echten deutschen Grammatik als Grenzen gesetzt.***

BACKUS-NAUR-Schreibweise (BACKUS-NAUR-Notation, BNF)

Nichtterminal: möglichst in Großbuchstaben und kursiv gesetzt ➔ *ZIFFER*
Terminale: wie üblich geschrieben (ev. in Anführungszeichen); normal gesetzt ➔ 1 "2"



Produktionen: Nichtterminal gefolgt von einem Rechtspfeil und einem oder mehreren verknüpften Nichtterminal(en) oder Nichtterminal gefolgt von einem Rechtspfeil und einem Terminal
übliche Verknüpfung ist die Aneinanderreihung (Anhängen, Verketten, Hintereinanderschreiben)
senkrechte Strichen zeigen Alternativen auf (sind praktisch Verkürzungen von zwei oder mehr Regeln zu einer kompakteren)

Terminale werden in der weit verbreitenden Darstellung als BACKUS-NAUR-Diagramm (auch Syntax-Diagramme genannt) in Kreisen, Ovalen oder abgerundeten Rechtecken notiert

Abkürzung für Terminal-Symbole (TS)

Nichtterminale werden im BN-Diagramm (BN-Syntax-Diagramm) als Rechtecke gezeichnet

Nicht-Terminal-Symbole wurden von CHOMSKY zur Strukturierung einer Sprache eingeführt.
Nichtterminale natürlicher Sprachen sind z.B. Satz, Subjket, Prädikat, Verb, Objekt, …
In Programmiersprachen sind Variable, Anweisung, Prozedur, Bedingung die typischen Nichtterminale.

Abkürzung für Nicht-Terminal-Symbole (NTS)

Definition(en): Terminale
Terminale sind gültige Elemente aus dem Zeichenvorrat einer Sprache.

Definition(en): Nicht-Terminale
Nicht-Terminale sind grammatikalische Elemente / Variablen einer Sprache.

die Nicht-Terminal(-Symbol)e müssen schrittweise durch andere Nicht-Terminale oder dann Terminale ersetzt werden, um einen gültigen Ausdruck der Sprache zu erzeugen

Die oben umrissene einfache deutsche Sprache ließe sich dann wie folgt definieren:

allgemein	**einfache, deutsche Sprache**
G = (T, N, S, P)	
T	T = {Maus, Mutter, Monika, Katze, liebt, jagt, holt, hasst}
N	N = {<Satz>, <Subjekt>, <Prädikat>, <Objekt>}
S	S = <Satz>
P	P = {<Satz> → <Subjekt> <Prädikat> <Objekt>, <Subjekt> → Maus, <Subjekt> → Mutter, <Subjekt> → Monika, <Subjekt> → Katze, <Prädikat> → liebt, <Prädikat> → jagt, <Prädikat> → holt, <Prädikat> → hasst, <Objekt> → Maus, <Objekt> → Mutter, <Objekt> → Monika, <Objekt> → Katze }

Die Regel-Formulierung entspricht der BACKUS-NAUR-Form (BNF). In der erweiterten BACKUS-NAUR-Form (EBNF) wird die Formulierung etwas effektiver:

allgemein	**einfache, deutsche Sprache**
G = (T, N, S, P)	
T	T = {Maus, Mutter, Monika, Katze, liebt, jagt, holt, hasst}
N	N = {<Satz>, <Subjekt>, <Prädikat>, <Objekt>}
S	S = <Satz>
P	P = {<Satz> → <Subjekt> <Prädikat> <Objekt>, <Subjekt> → Maus | Mutter | Monika | Katze, <Prädikat> → liebt | jagt | holt | hasst, <Objekt> → Maus | Mutter | Monika | Katze }

Näheres und eine qualifizierte Einordnung erfolgt später (→ 3.0.2. Darstellungen / Visualisierungen von Grammatiken).
Auch die Definition einer Sprache greifen wir später wieder auf (→).

Nutzt man eine verallgemeinerte Definition von Sprache, dann lassen sich z.B. die folgenden Sprachen konstruierten / aufzeigen / definieren:

allgemein	**zu definierende Sprache**
G = (T, N, S, P)	

Beispiele für Grammatiken und deren Sprachen:

allgemein	**einfache, deutsche Sprache**
G = (T, N, S, P)	
T	T = **{a, b, c, d}**
N	N = **{S, X}**
S	S = **S**
P	P = **{S, X}, S, {S → aXbc, X → aXb, aXb → d}**

G = (T, N, S, P) = ({a, b, c, d}, {S, X}, S, {S → aXbc, X → aXb, aXb → d})
P_1 P_2 P_3

w_1:	P_1	**S → aXbc**	
	P_2	a**X**bc → a**aXb**bc	
	P_2	aa**X**bbc → aa**aXb**bbc	
	P_2	aaa**X**bbbc → aaa**aXb**bbbc	
	P_3	aaa**aXb**bbbc → aaa**d**bbbc	willkürliche Regelanwendung
w_2:	P_1	S → aXbc	
	P_3	aXbc → dc	kleinstmögliche Regelnutzung scheinbar einfachstes Wort
w_3:	P_1	S → aXbc	
	P_2	aXbc → aaXbbc	
	P_3	aaXbbc → adbc	kleinstmögliche Nutzung aller Regeln
w_4:	P_1	S → aXbc	
	P_2	aXbc → aaXbbc	
	P_2	aaXbbc → aaaXbbbc	
	P_2	aaaXbbbc → aaaaXbbbbc	
	P_2	aaaaXbbbbc → aaaaaXbbbbbc	
	P_2	aaaaaXbbbbbc → aaaaaaXbbbbbbc	
	P_3	aaaaaaXbbbbbbc → aaaaadbbbbbc	wieder willkürl. Regelnutzung

…

L(G) = {dc, adbc, aadbbc, aaadbbbc, …, aaaaadbbbbbc, …}
L(G) = {a^0db^0c, a^1db^1c, a^2db^2c, … , a^5db^5c, …}

→ **L(G) = { w ϵ T* | w = a^ndb^nc; n ≥ 0}**

Definition(en): Syntax
Der Syntax (einer Sprache) ist das Regelwerk für den Aufbau von Sprach-Teilen (z.B. Sätzen).
Der Syntax ist die Summe der grammatikalischen regeln einer Sprache.

besitzen auch Semantik
aber keine weiteren Sprach-Charakteristika

3.0.2. Darstellungen / Visualisierungen von Grammatiken

3.0.2.1. BACKUS-NAUR-Form (BNF)

- Nicht-Terminale werden in spitze Klammern **<** … **>** eingeschlossen
- das Meta-Symbol **::=** bedeutet "definiert als" ("ergibt sich aus" / "produziert durch")

und zusätzlich eingeführt

- das Meta-Symbol **|** dient als Trenn-Zeichen zwischen Alternativen (vor allem, um unedliche viel Zeilen für die Definition eines größeren Alphabetes zu sparen)

in der rudimentärsten Form wird auf das Meta-Symbol "|" verzichtet. Es dient im wesentlichen dazu unendliche Folgen von Regeln z.B. für die Definiton der Ziffern zu verhindern.

Aus rudimentärer BNF zur Beschreibung von Ziffer:

```
<Ziffer> ::= 0
<Ziffer> ::= 1
<Ziffer> ::= 2
<Ziffer> ::= 3
<Ziffer> ::= 4
<Ziffer> ::= 5
<Ziffer> ::= 6
<Ziffer> ::= 7
<Ziffer> ::= 8
<Ziffer> ::= 9
```

wird dann:

```
<Ziffer> ::= 0 | 1 | 2| 3 | 4 | 5 | 6 | 7 | 8 | 9
```

eignet sich zur Darstellung aller Grammatiken für kontextfreie Sprachen (nach CHOMSKY: Typ-2-Sprache)
hiezu zählen z.B. alle gängigen Programmiersprachen mit ihrem definiertem Syntax

Definition(en): BNF – BACKUS-NAUR-Form
Die BACKUS-NAUR-Form ist eine Schreibweise für die Darstellung der Grammatik (und des Syntax) einer (Kontext-freien) Sprache.
Die BACKUS-NAUR-Form ist eine formale Metasprache zur Beschreibung von Grammatiken (Typ-2-Grammatiken nach CHOMSKY).

Sprache mit Worten, die aus beliebig vielen a-Symbolen bestehen

```
<Wort>   ::= <Symbol>
<Symbol> ::= a | <Symbol> <Symbol>
```

Sprache aus Worten, die aus bieliebig vielen 1 bestehen und am Ende zweimal die Sequenz 01 enthalten

```
<Wort>       ::= <Vorlauf> <Ende>
<Vorlauf>    ::= 1 | <Vorlauf> <Vorlauf>
<Ende>       ::= <Endsymbol> <Endsymbol>
<Endsymbol>  ::= 01
```

Aufgaben:

1. ***Geben Sie die Terminale sowie die Nicht-Terminale der obigen Grammatik als Mengen an!***
2. ***Gesucht ist die BNF für die Grammatik zur Beschreibung einer Sprache, deren Worte immer abwechselnd aus "+" und "-"-Zeichen bestehen!***
3. ***Erstellen Sie eine BNF-Grammatik, mit der sich 24-Stunden-Uhrzeiten konstruieren lassen.***
4. ***Notieren Sie die Grammatik der IP-Adresse als EBF! Zahlen dürfen zuerst einmal als Bereich angegeben werden! Z.B.: 0 \| 1 \| .. \| 10***
5. ***Geben Sie die Grammatik an, mit der die Worte der Sprache der MORSE-Zeichen (mit ein bis vier Zeichen) bilden lassen.***
6. ***Prüfen Sie, ob die folgenden Worte in der angegeben EBF-Grammatik***

```
<Wort>     ::= <Sequenz> | <Sequenz> <Sequenz>
<Sequenz>  ::= <SymbolA> <SymbolB> <SymbolC>
<SymbolA>  ::= #
<SymbolB>  ::= * | <SymbolC>
<SymbolC>  ::= *
```

a)	#**	b)	**#**	c)	#**#**#**#**#**
d)	#**+**#**#**#**	e)	#*#*#	f)	**#**#

für die gehobene Anspruchsebene:

7. ***Notieren Sie die Grammatik der IP-Adresse als EBF! Das Alphabet besteht nur aus den Ziffern und dem Punkt.***
8. ***Geben Sie eine Grammatik an, mit der sich die angelsächsiche Uhrzeit-Darstellung aufbauen lässt!***

3.0.2.2. Erweiterte BACKUS-NAUR-Form (EBNF)

- identisch zu BNF
- das Meta-Symbol (… **|** …) besagt, dass genau eine Alternative gewählt werden muss
- die Meta-Symbole **[** … **]** kennzeichnet Optionen (können, müssen aber nicht gewählt werden)
- die Meta-Symbole **{** … **}** kennzeichnen eine Wiederholung, wobei der Inhalt mindestens einmal gewählt werden muss

EBNF ist genauso Leistungs-fähig und Ausdrucks-stark wie Synthax-Diagramme
eine automatische Umwandlung ist möglich
z.B. http://bottlecaps.de/rr/ui (Railroad Diagram Generator)

Vorteile der Synthax-Diagramme liegt in deren Anschaulichkeit und dem möglichen händischen (Finger-)Arbeiten
meist kompakter, da in einem Synthax-Diagramme üblicherweise gleich mehrere EBNF-Regeln zusammengefasst werden

Vorteile der EBNF liegt in der einfacher Konstruktion (Niederschrift) und Änderung
meist direkte Übertragung in Übersetzer-Programme (Automaten) möglich
klare und gleichbleibende Struktur

Definition(en): EBNF – Erweiterte BACKUS-NAUR-(Normal-)Form
Die Erweiterte BACKUS-NAUR-Form ist eine Schreibweise für die Darstellung der Grammatik (und des Syntax) einer Sprache unter Verwendung von Struktur-bestimmenden Meta-Symbolen.
Die Erweiterte BACKUS-NAUR-Form ist eine formale, kompakte Metasprache zur Beschreibung von Grammatiken (Typ-2-Grammatiken nach CHOMSKY).

Terminale werden als einfache Zeichen oder gegebenenfalls in Anführungszeichen notiert
Nicht-terminale werden in spitzen Klammern geschrieben
eine Produktion besteht aus einem (zu beschreibenen) Nicht-Terminal, dem Produktions- bzw. Regel-Zeichen "::=" und der Regel-Beschreibung (Nicht-Terminale und / oder Terminale
die Nacheinander-Schreibung bedeutet Sequenz
eine Trennung mit einem sekrechten Strich "|" steht für eine Alternative

einfaches Beispiel: "natürliche Zahlen"

```
<natürlicheZahlen>   ::= <NichtnullZiffer> | <NichtnullZiffer> <Ziffer>
<NichtnullZiffer>    ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<Ziffer>             ::= 0 | <NichtnullZiffer> | <Ziffer> <Ziffer>
```

die erweiterte BNF enthält noch zusätzliche Metasprach-Elemente, die z.B. Wiederholungen einfach beschreiben

verwendet wird dazu die geschweifte Klammer, was in der Klammer steht wird mindestens einmal durchlaufen
unser Beispiel mit den "natürlichen Zahlen" lässt sich somit etwas vereinfachen:

```
<natürlicheZahlen>   ::= <NichtnullZiffer> | <NichtnullZiffer> { <Ziffer> }
<NichtnullZiffer>    ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<Ziffer>             ::= 0 | <NichtnullZiffer>
```

Aufgaben:

1. *Erweitern Sie die einfache deutsche Grammatik um ein Satzzeichen!*
2. *Erweitern Sie die BNF für die natürlichen Zahlen auf die Sprache der ganzen Zahlen!*
3. *Gesucht ist die BNF für die hexadezimalen Zahlen! (hier seien mehrere führende Nullen zulässig!)*
4.

für die gehobene Anspruchsebene:

5. *Erstellen Sie eine BNF für die Sprache der 16-bit-Adressen in Hexadimal-Form!*
6. *Erstellen Sie die BNF für reele Zahlen!*

Beispiel: "ganze Zahlen" (BNF)

```
<ganzeZahlen>        ::= <VorzeichenZahl> | <VorzeichenloseZahl>
<VorzeichenZahl>     ::= <Vorzeichen> <VorzeichenZahl>
<Vorzeichen>         ::= + | -
<VorzeichenloseZahl> ::= <NichtnullZiffer> | <NichtnullZiffer> <Ziffern>
<Ziffern>            ::= <Ziffer> | <Ziffer> <Ziffern>
<NichtnullZiffer>    ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<Ziffer>             ::= 0 | <NichtnullZiffer>
```

Beispiel: "ganze Zahl" (EBNF)

```
<ganzeZahlen>     ::= [<Vorzeichen>] <VorzeichenloseZahl> <NichtnullZiffer>
                      {<Ziffer>}
<Vorzeichen>      ::= + | -
<NichtnullZiffer> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<Ziffer>          ::= 0 | <NichtnullZiffer>
```

Definition der modifizierten EBF als EBF

```
<ModifizierteEBF>   ::= '' | <Regel> <ModifizierteEBF> .
<Regel>             ::= <NichtTerminal> ':' ':' '=' <Liste> '.' .
<Liste>             ::= '' | <Element> <Liste> .
<Element>           ::= <Terminal> | <NichtTerminal> .
<Terminal>          ::= '"' <Symbol> '"' .
<NichtTerminal>     ::= '<' <Bezeichner> '>' .
<Bezeichner>        ::= <GrossBuchstabe> <BezeichnerZeichen> |
                        <GrossBuchstabe> .
<BezeichnerZeichen> ::= <BezeichnerZeichen> <BezeichnerZeichen> |
                        <KleinBuchstabe> | <GrossBuchstabe> |
                        <Ziffer> | '_' .
<Symbol>            ::= '!' | '$' | '%' | ... |
                        <KleinBuchstabe> | <GrossBuchstabe> | <Ziffer> .
<KleinBuchstabe>    ::= 'a' | 'b' | 'c' | ... | 'z' .
<GrossBuchstabe>    ::= 'A' | 'B' | 'C' | ... | 'Z' .
<Ziffer>            ::= '1' | '2' | '3' | ... | '0' .
```

Achtung! Hier sind die Terminal-Symbole in einfache Hochkommata (') eingefasst, weil das Anführungszeichen (") selbst als Terminal-Zeichen gebraucht wird.

Aufgaben:

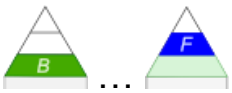
1.

2. Prüfen Sie, ob die nachfolgenden Ausdrücke gültige Regeln einer modifizierten EBF sind!

a) `<Buchstabe> ::= <KleinBuchstabe> | <GrossBuchstabe>`
b) `<SpezialBuchstabe> | <Ziffer> := <Zeichen>`
c) `<TShirt> ::= <Aermel> <Vorderseite> <Aermel> <Rueckseite>`
d) `"A" ::= '"' 'A' '"'`
e) `<Kette> ::= '' | <Kette> <Element>`
f)
g)
h)

3.

3.0.3. Prüfung einer Grammatik – Wie machen es die Interpreter und Compiler?



Grammatik einer ((noch) einfache(re)n, deutschen) Sprache

Regeln:

```
<Satz>        → <Subjekt> <Prädikat> <Objekt>
<Subjekt>     → <Artikel> <Attribut> <Substantiv>
<Attribut>    → ""  |  <Adjektiv>  |  <Adjektiv>  <Attribut>
<Objekt>      → <Artikel> <Attribut> <Substantiv>
```

Basiselemente:

```
<Prädikat>    → "liebt"  |  "jagt"  |  "holt"  |  "hasst"
<Artikel>     → ""  |  "die"
<Substantiv>  → "Maus"  |  "Mutter"  |  "Monika"  |  "Katze"
<Adjektiv>    → "kleine"  |  "graue"  |  "große"  |  "schnelle"
```

Testen eines Satzes mit der Grammatik:

Wie geht ein Computer vor?
Verstehen der prinzipiellen Arbeit eines Compiler's oder Interpreter's.
ähnlich zu verstehen sind Parser,

ähnlich zur folgenden Seite!

Bilden eines Satzes mit der Grammatik:

für **<Satz>** existiert eine Regel:

<Satz> → <Subjekt> <Prädikat> <Objekt>

erstes zu bildendes Element: **<Objekt>**

<Satz> → <Subjekt> <Prädikat> **<Objekt>**

für **<Objekt>** existiert eine Regel:

<Objekt> → <Artikel> <Attribut> <Substantiv>

erstes zu bildendes Element: **<Substantiv>**

<Objekt> → <Artikel> <Attribut> **<Substantiv>**

für **<Substantiv>** existiert eine Regel mit Auswahl-Elementen:

<Substantiv> → "Maus" | "Mutter" | "Monika" | "Katze"

ein Element auswählen:

<Substantiv> → "Maus" | "Mutter" | **"Monika"** | "Katze"

FERTIG mit **<Substantiv>** zurück zu Element: **<Objekt>**

!!!ab hier jetzt verkürzt dargestellt:

nächstes zu bildendes Element: **<Attribut>**

<Objekt> → <Artikel> **<Attribut>** **"Monika"**

für **<Attribut>** existiert eine Regel mit Auswahl-Elementen:
ein Element auswählen:

<Attribut> → **""** | <Adjektiv> | <Adjektiv> <Attribut>

FERTIG mit **<Attribut>** zurück zu Element: **<Objekt>**

nächstes zu bildendes Element: **<Artikel>**

<Objekt> → **<Artikel>** **""** **"Monika"**

für **<Artikel>** existiert eine Regel mit Auswahl-Elementen:
ein Element auswählen:

<Artikel> → "" | **"die"**

FERTIG mit **<Artikel>** zurück zu Element: **<Objekt>**

<Objekt> ist fertig:

<Objekt> → **"die"** **""** **"Monika"**

FERTIG mit **<Objekt>** zurück zu Element: **<Satz>**

nächstes zu bildendes Element mit Auswahl-Elementen: **<Prädikat>**

<Satz> → <Subjekt> **<Prädikat>** **"die"** **""** **"Monika"**

für **<Prädikat>** existiert eine Regel:
ein Element auswählen:

<Prädikat> → **"liebt"** | "jagt" | "holt" | "hasst"

FERTIG mit **<Prädikat>** zurück zu Element: **<Satz>**

nächstes zu bildendes Element mit Auswahl-Elementen: **<Prädikat>**

<Satz> → **<Subjekt>** **"liebt"** **"die"** **""** **"Monika"**

für **<Subjekt>** existiert eine Regel:

<Subjekt> → <Artikel> <Attribut> <Substantiv>

erstes zu bildendes Element: **<Substantiv>**

<Subjekt> → <Artikel> <Attribut> **<Substantiv>**

für **<Substantiv>** existiert eine Regel mit Auswahl-Elementen:
ein Element auswählen:

<Substantiv> → "Maus" | "Mutter" | "Monika" | "Katze"
FERTIG mit <Substantiv> zurück zu Element: <Subjekt>

nächstes zu bildendes Element: <Attribut>
<Objekt> → <Artikel> <Attribut> "Mutter"

für <Attribut> existiert eine Regel mit Auswahl-Elementen:
ein Element auswählen:
<Attribut> → "" | <Adjektiv> | <Adjektiv> <Attribut>

für <Adjektiv> existiert eine Regel mit Auswahl-Elementen:
ein Element auswählen:
<Adjektiv> → "kleine" | "graue" | "große" | "schnelle"
FERTIG mit <Adjektiv> zurück zu Element: <Attribut>

FERTIG mit <Attribut> zurück zu Element: <Objekt>

nächstes zu bildendes Element: <Artikel>
<Objekt> → <Artikel> "große" "Mutter"

für <Artikel> existiert eine Regel mit Auswahl-Elementen:
ein Element auswählen:
<Artikel> → "" | "die"
FERTIG mit <Artikel> zurück zu Element: <Objekt>

<Objekt> ist fertig:
<Objekt> → "die" "große" "Mutter"
FERTIG mit <Objekt> zurück zu Element: <Satz>

<Satz> → "die" "große" "Mutter" "liebt" "die" "" "Monika"

Satz-Regel erfüllt: Ergebnis lautet:
<Satz> → "die" "große" "Mutter" "liebt" "die" "" "Monika"

also : "Die große Mutter liebt die Monika." – Ein Satz der zumindestens im Imbiss-Deutsch eine Chance hätte. Ein Deutsch-Lehrer wäre damit nicht zufrieden – der müsste dann erst mal aber einer bessere Grammatik anbieten.
Was auf den ersten Blick sehr kompliziert aussieht, ist sture Abarbeitung immer der gleichen Schemata und da ist der Computer die perfekte Hilfe. Er arbeitet solche Probleme (Algorithmen) perfekt ab.

Grammatik = (N,T,P,S)
N = { Nominalphrase,Verbalphrase,Eigenname,Artikel,Substantiv,Verb }
T = { Marie, Bernd, Hund, Maus,Ball,Kasten,der,die,das,frisst, liest,jagt,liebt,. }
P = { Satz → Nominalphrase Verbalphrase .
Nominalphrase → Eigenname | Artikel Substantiv
Verbalphrase → Verb | Verb Nominalphrase
Eigenname → Marie | Bernd
Substantiv → Hund | Maus | Ball | Kasten
Artikel → der | die | das
Verb → frisst | liest | jagt | liebt
S = Satz

3.0.3.1. alternative Realisierung in PROLOG

Eine ähnliche, aber anders in Prolog realisierte, Umsetzung der einfachen deutschen Sprache (→ Grammatik einer ((sehr, sehr) einfachen, deutschen) Sprache / Kleinkinder-Sprache) sieht so aus:

	Quellcode	Kommentar(e)
1	`% Autor:  T. Hempel`	%-Zeichen sagt: es folgt ein Kommentar; hier Autor und in der nächsten Zeile das Erstellungsdatum
2	`% Quelle: Zimmermann: "Die Implementation kontextfreier Grammatiken in PROLOG".`	
3	`%         In LOG IN 21 (2001) Heft 5/6 S. 68ff.`	
4	`% Datum: 11.10.2010`	
5	`% 02 Artikel`	
6		
7	`% Symbol   Interpretation`	diese Kommentare beschreiben das PRO-LOG-Aufbau-Konzept und den Umgang mit dem Programm
8	`% -->      besteht aus`	
9	`% ,        und`	
10	`% |        oder`	
11	`% []       Wort/Terminal, besteht immer aus Kleinbuchstaben`	
12	`% ''       Nichtterminal (Variable), beginnt immer mit einem Grossbuchstaben`	
13		
14	`% Arbeitsweise:`	
15	`% Quelltext schreiben/ergaenzen/korrigieren`	
16	`% Quelltext konsultieren (Taste F9)`	
17	`% bei Fehlermeldungen im unteren Fensterbereich -> Korrektur und erneutes Konsultieren`	
18	`% bei Fehlerfreiheit im unteren Fensterbereich`	
19		
20	`% Aufruf: ?- phrase('Satz',[kuehe, fressen, graeser]).`	so muss die Grammatik getestet werden!
21	`%         ?- phrase('Satz', X).`	
22		
23	`'Satz'       --> 'Subjekt', 'Praedikat', 'Objekt'.`	hier erfolgt die Definition der verfügbaren Regeln der Sprache sowie die Angabe von gültigen Worten
24	`'Subjekt'    --> 'Substantiv'.`	
25	`'Objekt'     --> []|'Substantiv'.`	
26	`'Substantiv' --> [kuehe]|[graeser].`	
27	`'Praedikat'  --> [fressen].`	

Q: Weiterbildung HILF!2018 Rostock

Ob diese Umsetzung nun besser oder schlechter als die vorne beschriebene Version, erschließt sich mir nicht. Vielleicht ist sie etwas intuitiver und weniger PROLOG.

Da später auch noch andere Implementierungen diesem Prinzip folgen, habe ich diese Variante hier mit aufgenommen. Jeder nutze die Version, die ihm persönlich mehr liegt.
Zu beachten ist aber, dass die hier vorgestellte Version mit dem Plural (Mehrzahl) arbeitet, was bei einigen Erweiterungen (z.B. Adjektive) von Vorteil sein kann.

Aufgaben:

1. Testen Sie die folgenden Ausdrücke! Überlegen Sie sich vorher, wie das Prolog-System reagieren müsste!

a) phrase('Satz', [kuehe, fressen, graeser]).
b) phrase('Praedikat', [fressen]).
c) phrase('Satz', [graeser, fressen, kuehe]).
d) phrase('Satz', [alle, katzen, jagen, die, maeuse]).
e) phrase('Subjekt', [graeser]).
f) phrase('objekt', [alle Katzen])
g) phrase('Satz', [alle, gelben, großen, katzen, jagen, die, klugen, maeuse]).

2. Erweitern Sie die Grammatik zuerst einmal um weitere passende Wörter (Terminale)! Die nachfolgen Wörter sollten dabei sein!)

Mäuse, Katzen, lieben, hassen, Radieschen, sehen, erschrecken, Kinder, Blumen, brauchen

3. Setzen Sie die folgenden Probleme in passende Anfragen in der erweiterten Grammatik um! (Satzzeichen werden nicht beachtet!)

a) Handelt es sich bei: "" um einen Satz?
b) Ist dies ein gültiger Ausdruck für einen Satz, ein Substantiv und ein Prädikat?: Die Katzen jagen.
c) Handelt es sich bei: "" um einen Satz, Artikel und / oder ein Substantiv?
d) Handelt es sich bei: "Die Katzen fressen die schönen Kühe." um keinen Satz?

4. Erweitern Sie nun die Grammatik um weitere Regeln für "Artikel" und " Adjektive"! Geben Sie auch passende Terminale ein! (Die nachfolgen Wörter sollten dabei sein!)

alle, die, schönen, gelben, großen, klugen,

5. Ergänzen Sie die Grammatik um Konstrukte der Form: Artikel (Adjektiv) Verb (Adverb)! Der Satz: "der große Hund bellte laut" sollte dann passen.

6. Testen Sie Ihre Grammatik mit passenden und unpassenden Ausdrücken! (Testen Sie auch die Ausdrücke von Aufgabe 1!

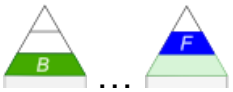
7. Übergeben Sie Ihre Grammatik- / Prolog-Datei einem anderen Kursteilnehmer und lassen Sie diesen Ihre Grammatik testen!

8. Testen Sie die Grammatik eines anderen Kursteilnehmers und dokumentieren Sie die Tests!

für das gehobene Anspruchsniveau:

9. Erweitern Sie die Grammatik um einen Teil der Fragen zulässt und testen Sie diese! (Auf das Fragezeichen dieses Mal nicht verzichten!)

3.1. Sprachen und Grammatiken



Problem-Fragen für Selbstorganisiertes Lernen
Welche Arten von Sprachen gibt es? Mit welchen Sprachen beschäftigt sich die Theoretische Informatik (TI)? Hat jede Sprache eine Grammatik? Was haben Sprachen noch außer einer Grammatik? Welche Elemente haben Sprachen? Zu welcher Sprach-Gruppe gehören Programmiersprachen? Warum müssen Programmiersprachen so formal sein? Kann man mit einer Grammatik beweisen, dass ein Computer-Programm richtig arbeitet? Was hat es mit dem Sprachen-Pabst Noah CHOMSKY auf sich? Was hat er besonderes geleistet? Wo geht der KLEENE-Stern auf? Gibt es leere Worte? Wie lang ist ein leeres Wort? Ist codieren und chiffrieren das Gleiche? Was hat das mit de TI zu tun? Haben auch Geheimsprachen eine Grammatik? Können Computer die menschliche Sprache verstehen?

Zweck von Sprachen:

Sprachen sind im Allgemeinen durch Regeln charakterisiert. Anhand der Regeln kann eine Sprache oder deren Teile – z.B. Worte, Sätze, … - kontruiert werden. Desweiteren kann man eine Sprachen oder deren Teile anhand der Regeln auf Korrektheit prüfen. Das Konstrukt von Regeln zu einer Sprache nennt man allgemein Grammatik.
Zum klaren Verständnis muss hier noch mal darauf hingewiesen werden, dass es nicht darum geht einen sinnvollen Satz oder ein wirklich existierendes Wort zu erstellen oder zu prüfen. Grammatiken sind nur für die ordnungsgemäße Kombination der gültigen Zeichen / Symbole einer Sprache zuständig. Viele regionale Sprach-Gewohnheiten oder Dialekte vergegenwärtigen uns diesen Sachverhalt jeden Tag wieder.

In der Informatik sind Sprachen z.B. bei der Konstruktion einer Programmiersprache und dann bei Compiler- bzw. Interpreter-Bau wichtig.
Die fertigen Compiler bzw. Interpreter müssen dann die gleiche Sprache verstehen können. Sie führen zuerst immer eine Syntax-Analyse durch, bevor dann letztendlich die Übersetzung in Maschine-Code oder Zwischen-Codes vorgenommen wird.

In der Gegenwart nimmt auch die Verarbeitung von natürlicher Sprache eine immer größere bedeutung ein. Da ist zum Einen die Analyse von Texten und zum Anderen die Sprach-Erkennung. Siri und Cotana sind da nur die bekanntesten Vertreter.
Auch die Erzeugung von Sprache aus Text gehört mit in den Arbeits-Bereich der Computer-Linguistik.

Die Vielzahl möglicher Sprachen muss für sinnvolle Betrachtungen in Gruppen oder Klassen eingeteilt werden.

Nutzung der Grammatik für die …

- **Erstellung von Sprach-Elementen (Transduktion)**
- **Prüfung eines Elements auf Zugehörigkeit zur Sprache (Akzeptanz)**

Sprachen niederen Typ's sind Erzeugungs-mächtiger als die höheren Sprach-Typen
die Art (/ der Typ) der Grammatik bestimmt den Typ der Sprache

Das bedeutet, das aus dem gleichen Alphabet – bezogen auf die gleiche Wort-Länge – mehr Wörter / Sätze gebildet werden können.

allgemein Alphabet einer formalen Sprache **Σ** oder **X**
i.A. ist die resultierende Sprache – also die Menge aller gültigen Wörter - ist meist unendlich groß
mit **Σ*** gekennzeichnet; man spricht auch von der KLEENEschen Hülle (KLEENE sprich: *klie ni*)

(die) einfachste Sprache(n) enthält / enthalten keine Wörter
vielfach wird ein Ausdruck in der Sprache – quasi ein Satz / eine Eingabe – als Wort bezeichnet
Sprachen mit wenigen Symbolen und Regeln haben oft auch endlichen Mengen an gültigen Wörtern

Die CHOMSKY-Hierarchie (1956; auch: CHOMSKY-SCHÜTZENBERGER-Hierarchie) ist eine Unterteilung formaler Sprachen in der Theoretischen Informatik.

Sprachen-Typ nach CHOMSKY

- **Typ 0** uneingeschränkte Grammatik; beliebige formale Grammatik
  hierzu gehören praktisch alle Grammatiken
  mit TURING-Automaten prüfbar
  entscheidbar

- **Typ 1** Kontext-sensitive Grammatik
  mit linear Platz-beschränkter nicht-deterministem TURING-Automaten bzw. linear beschränktem Automaten bearbeitbar
  Einschränkung gegenüber Typ 0: für alle Regeln **u** → **v** gilt, dass die Worte auf der rechten Seite der Regel genausolang sind, wie die auf der linken, z.B.:
  α **A** $\beta \rightarrow \alpha \gamma \beta$

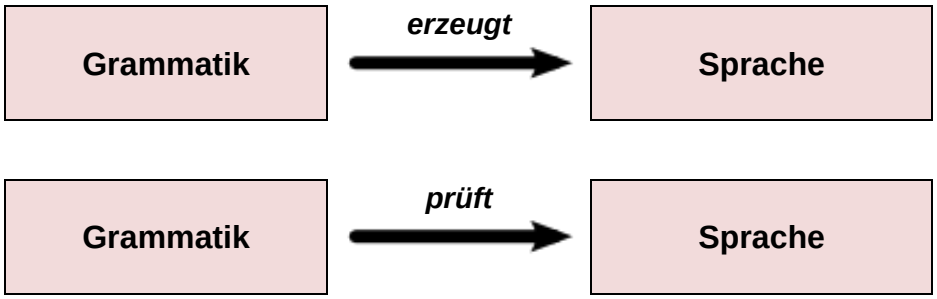
- **Typ 2** Kontext-freie Grammatik
  mit nicht-deterministischen Keller-Automaten bearbeitbar
  Einschränkung gegenüber Typ 1:
  A → α

- **Typ 3** reguläre Grammatik
  mit endlichem Automaten bearbeitbar
  Einschränkung gegenüber Typ 2:
  A → α **|** α **B**

A, B … Nicht-Terminale, Variablen; **a, b** … Terminale; α, β, γ … Symbol-Folgen

Definition(en): formale Grammatik
Eine formale Grammatik G ist das Bildungs- und / oder Erkennungs-Regelwerk für eine formale / künstliche Sprache.

formale Grammatiken heißen auch CHOMSKY-Grammatiken



abstrakte Definition(en): formale Grammatik

Eine formale Grammatik G ist ein 4-Tupel aus:
einer endlichen Menge von Terminalen T (entspricht dem Alphabet Σ der Sprache),
einer endlichen Menge von Nichtterminalen N,
einer endlichen Menge von Regel / Projektionen P und
einem Startsymbol s (aus der Menge der Nichtterminalen).

$G = (N, T, P, s)$; $N \cap T = \varnothing$; $p_1 \rightarrow p_2$; $s \in N$
oder $G = (N, \Sigma, P, s)$; $N \cap A = \varnothing$; $p_1 \rightarrow p_2$; $s \in N$

Eine Grammatik G ist ein System aus Variablen (einschließlich einer Start-Variable s), einem Alphabet Σ (od. den Terminale T) und einem Konstrukt von Produktions-Regeln P (od. Ersetzungs-Regeln).

Eine Grammatik ist eine Zusammenstellung von Nichtterminal-Symbolen, Terminal-Symbolen, mindestens einem Start-Symbol und eine Menge von Produktions-Regeln (,die sich auf die genannten Symbole beziehen).

In der TI benutzen wir z.B. sehr häufig die folgenden Alphabete:

Alphabet Σ (ohne leeres Zeichen)	"Alphabet" mit leerem Zeichen
{0,1}	{0,1,ε}
{a}	{a,ε}
{a,b}	{a,b,ε}
{0,1,2,3,4,5,6,7,8,9}	{0,1,2,3,4,5,6,7,8,9,ε}
{a,b,c,…,x,y,z}	{a,b,c,…,x,y,z,ε}
{(,)}	{(,),ε}
→ ASCII-Code	

Definition(en): Alphabet

Ein Alphabet Σ ist eine endliche, nicht-leere Menge an Symbolen bzw. Terminalen T (z.B. Buchstaben, Ziffern und Zeichen).

Das Alphabet Σ einer formalen Sprache L ist die Menge derer terminaler Symbole T.

Ein Alphabet ist eine endliche, total geordnete Menge unterschiedlicher Symbole / Zeichen.

ASCII-Code-Tabelle (Basis-Version)

HEX	MSD	0	1	2	3	4	5	6	7
LSD	**bits**	**000**	**001**	**010**	**011**	**100**	**101**	**110**	**111**
0	**0000**	NUL	DLE	SPACE	0	@	P	-	p
1	**0001**	SOH	DC1	!	1	A	Q	a	q
2	**0010**	STX	DC2	"	2	B	R	b	r
3	**0011**	ETX	DC3	#	3	C	S	c	s
4	**0100**	EOT	DC4	$	4	D	T	d	t
5	**0101**	ENQ	NAK	%	5	E	U	e	u
6	**0110**	ACK	SYN	&	6	F	V	f	v
7	**0111**	BEL	ETB	'	7	G	W	g	w
8	**1000**	BS	CAN	(	8	H	X	h	x
9	**1001**	HAT	EM	)	9	I	Y	i	y
A	**1010**	LF	SUB	*	:	J	Z	j	z
B	**1011**	VT	ESC	+	,	K	[	k	{
C	**1100**	FF	FS	,	<	L	\	l	--
D	**1101**	CR	GS	-	=	M	]	m	}
E	**1110**	SO	RS	.	>	O	^	n	~
F	**1111**	SI	US	/	?	N	←	o	DEL

wichtige Codes

NUL	NULL	leeres Datenfeld / Byte	
SOH	Start of Heading		
STX	Start of Text		
ETX	End of Text		
EOT	End of Transmission		
ENQ	Enquiry		
ACK	Acknowlegde		
BEL	bell	Signalausgabe	
BS	Backspace		[←]
HT	Horizontal Tabulation	Horizontaler Tabulator	[Tab]
LF	Line Feed	Zeilenvorschub	
VT	Vertcal Tabulation	vertikaler Tabulator (Seitenvorschub)	
CR	Carriage Return	Wagen Rückfahrt / Zeilenrücksprung	[Enter] → LF + CR
SO	Shift Out		
SI	Shift IN		
DLE	Data Link Escape		
DC	Device Control		
NAK	Negative Acknowledge		
SYN	Synchronous Idle		
ETB	End of Transmission Block		
CAN	Cancel	Abbruch	
EM	End of Medium	Papierende	
SUB	Substitute		
ESC	Escape		[Esc]
FS	File Seperator		
GS	Group Seperator		
RS	Record Seperator		
US	Unit Seperator		
SPACE / SP	Space / Blank	Leerzeichen	[ ]
DEL	Delete	Löschen	

Aus einem Alphabet lassen sich die Wörter einer Sprache ableiten. Schon mit wenigen Symbolen im Alphabet können sich sehr viele Wörter ergeben.
Z.B. gibt es für das Alphabet Σ = {o,i} die folgende Wort-Menge:

{o, i, oo, oi, io, ii, ooo, ooi, oio, oii, ioo, ioi, iio, iii, …)

Die Menge ist unendlich groß, wie man schon aus dem Anfang der Menge erkennen kann.
Kommen weitere Symbole zur Sprache hinzu, wächst die Variablilität der Wörter sehr schnell. Nehmen wir z.B. m und x dazu, dann sähe der Anfang der Menge schon so aus:

{o, i, m, x, oo, oi, om, ox, io, ii, im, ix, mo, mi, mm, mx, xo, xi, xm, xx, ooo, ooi, oom, oox, oio, oii, oim, oix, omo, omi, omm, omx, oxo, oxi, oxm, oxx, ioo, ioi, iom, iox, iio, iii, iim, iix, imo, imi, imm, imx, ixo, ixi, ixm, ixx, moo, moi, mom, mox, mio, mii, mim, mix, mmo, mmi, mmm, mmx, mxo, mxi, mxm, mxx, xoo, xoi, xom, xox, xio, xii, xim, xix, xmo, xmi, xmm, xmx, xxo, xxi, xxm, xxx, …}

Mit mehr Symbolen steigt nur die Anzahl der Wörter einer bestimmten Länge. Man spricht auch von einer Wort-Ebene.
Dieses gewaltige Ansteigen der Wort-Menge in Abhängigkeit von der Anzahl der Symbole und der Wort-Länge wird uns später beim Wort-Problem begegnen (/ auf die Füße fallen!)

Definition(en): Wort (über ein Alphabet)
Ein Wort (über ein Alphabet Σ^*) ist eine endliche – ev. auch leere – Folge von Symbolen eines Alphabetes.

Ein Wort kann auch aus einer leeren Menge von Symbolen bestehen.
λ ist das leere Wort / leere Folge von Symbolen
Das leere Wort λ ist niemal in einer Sprache enthalten, vielmehr handelt es sich um das Wort, was ohne ein Symbol (bzw. das leere Symbol ε) gebildet wird / werden kann.
A* ist die Menge aller Wörter eines Alphabetes (einschließlich des leeren Wortes)

Definition(en): formale Sprache
Eine formale Sprache L ist eine Teilmenge aus den Wörtern Σ^* zu einem Alphabet Σ. $L \subseteq \Sigma^*$
Eine formale (Automaten-)Sprache ist eine Sprache, die ein Automat erkennen kann.
Eine formale Sprache L(G) ist eine Sprache L, die von einer formalen Grammatik G vollständig beschrieben wird.
Eine formale Sprache ist eine künstliche Sprache mit einer abgeschlossenen Grammatik (→ Syntax) und einer zugeordenten und definierten Bedeutung der Wörter (→ Semantik).

formale Sprache ist Teilmenge von Σ^*

Beispiele:
$\Sigma = \{0,1\}$
$\Sigma^* = \{\lambda, 0, 1, 00, 01, 10, 11, 000, 001, \ldots\}$
$L = \{0, 1, 10, 11, 101, 110, 111, \} \subseteq \Sigma^*$

$\Sigma = \{\}$

jede formale Sprache besitzt entweder eine Grammatik mit deren Hilfe die Sprache erzeugt werden kann oder es existiert ein Automat, der die Sprache erkennen kann

Grammatik G ist ein Vier-Tupel aus der (endlichen) Menge der Nichtterminalsymbole N, der (endlichen) Menge der Terminalsymbole T, der (endlichen) Menge der Produktions-Regeln P und dem Start-Symbol S.
die Mengen N und T sind disjunkt, d.h. ein Element, das in der einen Menge enthalten ist, kann nicht in der anderen auftreten. $N \cap T = \varnothing$
Regel x → y bedeutet, dass das Teilwort x durch das Teilwort y ersetzt werden kann

allgemein gilt für die Produktions-Regeln der Aufbau:

linke_Seite → *rechte_Seite*

G = (N, T, P, S)

zwei Grammatiken sind äquivalent, wenn sie die gleiche Sprache erzeugen

Kontext-freie Grammatik
Produktions-Regel: A → y
links steht genau ein Nichtterminal-Symbol

wichtig für die Beschreibung / Definition von Programmiersprachen
notwenig für den Bau der Interpreter / Compiler zum Übersetzen der Sprache in Maschinen-Code
Interpreter / Compiler erkennen die formale (Kontext-freie) Sprache

Notation üblich in
Syntax-Diagrammen
Nichtterminal-Symbole in Rechtecken; Terminal-Symbole in abgerundeten Rechtecken oder Ovalen / Kreisen

Extended BACKUS-NAUR-Form (EBNF, erweiterte BACKUS-NAUR-Form)
(s.a. → 3.0.2. Darstellungen / Visualisierungen von Grammatiken)
" " .. definiert das eingeschlossene Terminal-Symbol
= .. definiert als
| .. Alternative
[ ] .. optionale Folge (Inhalt der Klammer kann folgen oder nicht)
{ } .. beliebige Folge (Inhalt kann beliebig oft wiederholt werden)

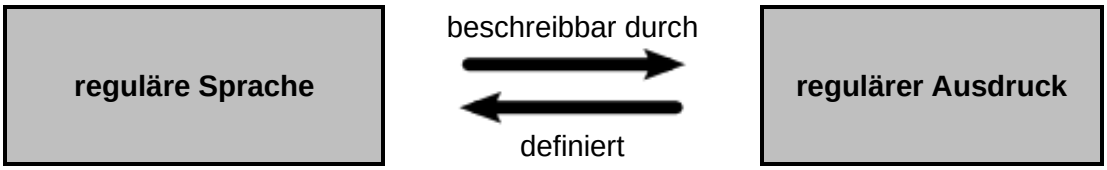
Sachlich gehören die formalen zu den regulären Spachen – also den Typ-3-Sprachen nach CHOMSKY.

Definition(en): reguläre Ausdrücke
Reguläre Ausdrücke sind (Teil-)Mengen / ist die Menge von gültigen Worten einer Sprache.
Reguläre Ausdrücke (induktive Definition) 1. ist ein regulärer Ausdruck für $L(\varnothing) = \varnothing$ 2. ist ein regulärer Ausdruck für $L(\lambda) = \{ \lambda \}$ 3. für jedes Zeichen a des Alphabet's Σ und $a \in \Sigma$ ist $\underline{a}$ ein regulärer Ausdruck für $L(\underline{a}) = \{a\}$ 4. sind α und β reguläre Ausdrücke, so ist es auch $(\alpha + \beta)$ mit $L(\alpha + \beta) = L(\alpha) \cup L(\beta)$ und auch $(\alpha \bullet \beta)$ mit $L(\alpha\beta) = L(\alpha) \circ L(\beta)$ und auch (α^*) mit $L(a^*) = L(\alpha)^*$
Reguläre Ausdrücke sind Notationen für (formale) Sprachen, die durch Automaten erkannt werden können.

λ … leeres Wort; L … Sprache ; $\underline{a}$ … regulärer Ausdruck; Σ … Alphabet ; a … Wort ;
α … (Grammatik-)Regel ; * … Wiederholung; * … KLEENE-Stern ()

Bedeutung regulärer Ausdrücke:

- in Suchfunktionen (z.B. grep in der Shell des Betriebssystems Linux, "Suchen …" in Textverarbeitungssystemen, Datenbanksystemen)
- Texteditoren / Programmier-Systeme (z.B. Syntax-Erkennung / Syntax-Highlighting)
- Textverarbeitungssysteme
- Suchmaschinen
- Mustererkennung



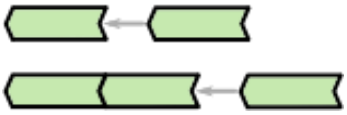
Ein Wort lässt sich aus einer Aneinanderreihung von Teilworten zusammensetzen
$w = w_1 w_2 \ldots w_n$

alternativ lässt sich schreiben:
$w = w_1 * w_2 * \ldots * w_n$

auch wenn ???

da die Wörter aus einzelnen Zeichen a zusammengesetzt werden können, gilt auch:
$w = a_1 a_2 \ldots a_n$

Konkatenation (von catena = lat.: Kette)
darunter versteht man das Regel-gerechte Aneinanderhängen / Verketten von Symbolen



Definition(en): Verkettung / Konkatenation
Die Verkettung oder Konkatenation ist eine binäre / zweistellige Operation (Operator-Symbol: ○), bei der die Operanden (meist Symbole, Zeichenketten und / oder Grammatik-Regeln) in der angegebenen Folge (Seitigkeit (von links nach rechts)) hintereinander notiert werden.

Verkettungen können, wie besprochen auf Symbole oder Worte angewendet werden. In der TI ist aber auch die Konkatenation von Mengen erlaubt. Dabei werden die Elemente beider Mengen Paar-weise aneinander gehängt. Die Ergebnis-Menge kann dabei bis zum Produkt der Elemente-Anzahlen der Ausgangs-Mengen groß werden.

Aufgaben:

1.

x. Zu einer Sprache $L = \Sigma^*$ *ist das Alphabet* Σ = {a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,z} *gegeben. Welche der nachfolgenden Wörter gehören nicht zur Sprache L? Begründen Sie!*

a)	farbig	b)	jxuqrlly	c)	geändert
d)	einzig	e)	simple	f)	sehr gut
g)	hört	h)	neuartisch	i)	Sprache

x. Welche Wörter gehören zur Sprache Σ_b^* *über dem binären Alphabet* Σ_b = {0,1}*? Zählen Sie mindestens 10 auf! Nennen Sie auch drei Worte, die nicht zur Sprache gehören! Kennzeichnen Sie mit einem andersfarbigen Stift die erste Fehlerstelle in jedem Wort!*

x.

für die gehobene Anspruchsebene:

x.

M1 **M2** **M1 ∪ M2** **M**

Definition(en): Vereinigung
Die Vereinigung ist die Zusammenführung zweier oder mehr Mengen (z.B. M1, M2) zu einer Ergebnis-Menge M. $M = M1 \cup M2$

Werden ev. mehrere oder entsprechend mächtige Mengen konkateniert, dann spricht man auch von Potentierung.

Definition(en): Potenzierung
Die Potenzierung ist die mehrfache Konkatenation von Mengen. $M3 = M1 \cup M2$ $M = M3 \cup M4$

Definition(en): Länge einer Zeichenkette
Die Länge **|k|** einer Zeichenkette **k** (/ eines Wortes der Sprache) ist die Anzahl der Alphabetszeichens (dieser Sprache). Für das leere Wort λ gilt **|k| = 0**.

Stephen Cole KLEENE (1909 – 1994) (KLEENE sprich: *klie'ni*)
untersuchte um 1956 Mengen von Zeichenketten, die von Automaten akzeptiert werden
führte den Begriff der regulären Menge ein
Die Vereinigung aller Potenzierungen ergibt den sogenannten KLEENE-Stern.

Definition(en): KLEENE-Stern
Der KLEENE-Stern ist eine einstellige Operation für Mengen M. Es gilt $M^* = M^0 \cup M^1 \cup M^2 \cup \ldots \cup M^n$

Ein wichtiges Ergebnis der Arbeiten von KLEENE (1956) war das folgende Theorem:

Theorem:	Eine Sprache ist dann regulär, wenn sie von einem Automaten erkannt werden kann.

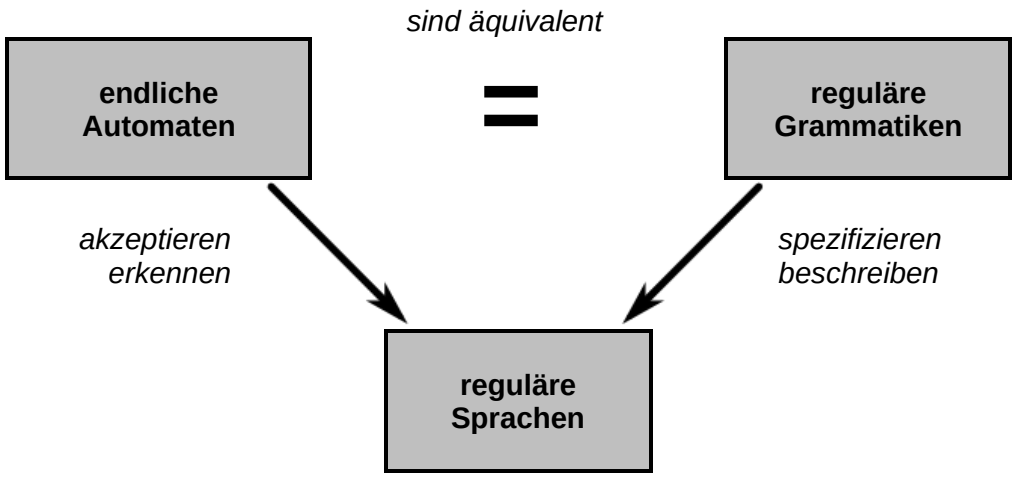
Ein Automat in diesem Sinne ist eine theoretische, mathematische Konstruktion zur Lösung von algorithmischen Aufgaben (→ 3.2. Automaten).
In zwei weiteren – quasi Unter-Theoremen – spezifizierte er die Erstellung und Prüfung von regulären Sprachen:

Theorem:	Eine reguläre Sprache kann von einem endlichen Automaten dargestellt / erzeugt werden.

Meint also, dass es einen endlichen Automaten gibt, der für diese Sprache Wörter erzeugen / synthetisieren kann. Der gemeinte endliche Automat ist durch eine begrenzte Anzahl von Zuständen charakterisiert (→ 3.2.3. endliche Automaten). Dieses (Unter-)Theorem wird deshalb auch als Synthese-Theorem bezeichnet.

Theorem:	Eine Sprache ist dann regulär, wenn sie von einem endlichen Automaten erkannt werden kann.

Dieses Theorem betrachtet die Akzeptanz / Analyse von Sprachelementen und wird auch Analyse-Theorem genannt. Für jede reguläre Sprache gibt es (muss es geben) einen endlichen Automaten, der die Prüfung von Wörtern auf Zugehörigkeit zur Sprache ermöglicht.
Im Umkehrschluss heißt das auch: Jede von einem endlichen Automaten dargestellte Sprache ist regulär.



Grammtiken sind u.U. dazu da, um Wörter einer Sprache zu erzeugen:

- Wörter werden schrittweise aus Symbolen (Terminale) und / oder Hilfs-Symbolen (Nicht-Terminale) zusammengesetzt / zusammengestellt / konkateniert
- alle Hilfs-Symbole (Nícht-Terminale) müssen am Ende ersetzt / umgewandelt / entfernt sein
- bei rechts- oder links-linearen Grammatiken wird immer nur ein Symbol (Terminal) pro Schritt erzeugt; ev. wird noch ein einzelnes Hilfs-Symbol (Nicht-Terminal) mitbenutzt
- nicht-lineare Grammatiken erzeugen mehrere Symbole gleichzeitig

Wiederholung Symbolik:
direkte Folge von Zeichen (quasi (fehlendes) Mal-Zeichen zwischen Symbolen) → Konkatenation
+ → Alternative; dieses oder das andere
* → KLEENE-Stern; beliebig viele (auch garkeine Aufrufe der Symbol(e)-Gruppen) Wiederholungen

Beispiele für Sprachen:

reguläre Sprache	**Umschreibung**	**Beispiel-Wörter**	**Erläuterungen**
(ab + aab)*	akzepiert Folge: ab oder aab und diese auch immer wieder hintereinander folgend	ε ab abab aab abababaab aabab	 weil 2. Menge auch ε enthält diverse gemischte Konkatenationen
		aa**ab**b **a** **b** abab**b**abaab	wäre nur mit einschiebender Regel möglich keine gültigen Wörter keine Regel zur Folge b auf b vorhanden

Zusammenfassung:
Eine Sprache ist eine Menge von Wörtern. Wörter sind Folgen von Buchstaben bzw. Symbolen aus einem Alphabet. Ein Alphabet ist die Menge der Buchstaben bzw. Symbole. Die Buchstaben bzw. Symbole können – müssen aber nicht – nach bestimmten Regeln kombiniert (aneinandergereiht) werden. Die Art der Einschränkungen in der Buchstaben- bzw. Symbol-Kombination / Die Beschränkung durch die Regeln bestimmt den Sprach-Typ (Hierarchie nach CHOMSKY).
Aus praktischen Gründen werden den Sprachen häufig noch das zusätzliche leere Wort λ (griech. Buchstabe: lambda) zugeordnet.
Die Länge eines Wortes wird symbolisch mit Betragsstrichen gekennzeichnet.

In der Theoretischen Informatik spielen auch die folgenden – definierten – Sprachen eine Rolle:

- die leere Sprache L = {} (ist eine Sprache!)
- eine Sprache nur mit dem leeren Wort ε , also L = {λ} (ist eine Sprache! (mit einem leeren Wort λ))
- die Universal-Sprache L = Σ^* , also die Menge aller endlichen Folgen von Symbolen / Zeichen aus einem Alphabet (, ist eine Sprache!)

Eine wirkliche Trennung von ε und λ ist nicht möglich. Beide sind äquivalent. Oft wird nur ε benutzt. Die Länge von ε bzw. λ ist 0.

Operations-Gruppen Operationen	**mathematische Notierung**	**Bemerkungen / …**
Mengen-theoretische Operationen		
Vereinigung	$L_1 \cup L_2$	reguläre Operation
Durchschnitt	$L_1 \cap L_2$	
Differenz	$L_1 \setminus L_2$	
Komplement	$\neg L = T^* \setminus L$	
Verknüpfungen		
Konkatenation / Verkettung	$L_1L_2 = \{w_1w_2 : w_1 \in L_1, w_2 \in L_2\}$	reguläre Operation
Kombination		
Interation	$L^* = \bigcup_{i=1}^{\infty} L^i$	reguläre Operation

Grammatik	Regeln	Sprachen	Entscheidbarkeit	Automaten	Abgeschlossenheit[1]	Zeitabschätzung
Typ-0 Beliebige formale Grammatik	$\alpha \rightarrow \beta$ $\alpha \in (\Sigma \cup N)^* \setminus \Sigma^*, \beta \in (\Sigma \cup N)^*$	rekursiv aufzählbar (nicht „nur" rekursiv, die wären entscheidbar!)	–	Turingmaschine (egal ob deterministisch oder nicht-deterministisch)	$\circ, \cap, \cup, *$	n.m.
Typ-1 Kontextsensitive Grammatik	$\alpha A \beta \rightarrow \alpha \gamma \beta$ $A \in N, \alpha, \beta \in (\Sigma \cup N)^*, \gamma \in (\Sigma \cup N)^+$ $S \rightarrow \varepsilon$ ist erlaubt, wenn es keine Regel $\alpha \rightarrow \beta S \gamma$ in P gibt.	kontextsensitiv	Wortproblem	linear platzbeschränkte nichtdeterministische Turingmaschine	$\complement, \circ, \cap, \cup, *$	$2^{O(n)}$
Typ-2 Kontextfreie Grammatik	$A \rightarrow \gamma$ $A \in N, \gamma \in (\Sigma \cup N)^*$	kontextfrei	Wortproblem, Leerheitsproblem, Endlichkeitsproblem	nichtdeterministischer Kellerautomat	$\circ, \cup, *$	$O(n^3)$
Typ-3 Reguläre Grammatik	$A \rightarrow aB$ (rechtsregulär) oder $A \rightarrow Ba$ (linksregulär) $A \rightarrow a$ $A \rightarrow \varepsilon$ $A, B \in N, a \in \Sigma$ Nur links- oder nur rechtsreguläre Produktionen	regulär	Wortproblem, Leerheitsproblem, Endlichkeitsproblem, Äquivalenzproblem	Endlicher Automat (egal ob deterministisch oder nicht-deterministisch)	$\complement, \circ, \cap, \cup, *$	$O(n)$

Legende für die Spalte *Regeln*

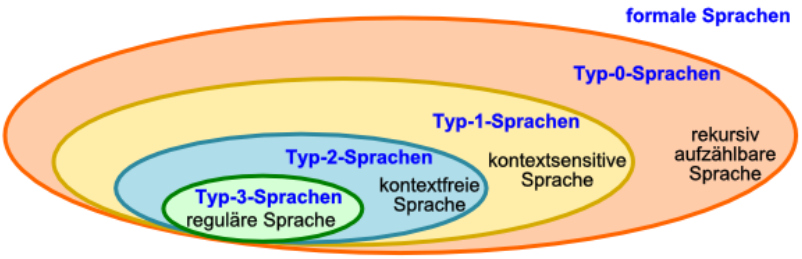
- Σ endliches *Alphabet* (Menge der Terminalsymbole)
- N Menge der Nichtterminalsymbole
- $S \in N$ *Startsymbol*
- ε leeres Wort
- P Menge von Produktionsregeln
- $\setminus$ ... Mengen-Differenzbildung
- $*$ ... Kleenescher Abschluss
- $\gamma \in (\Sigma \cup N)^+$ ... γ muss mindestens ein Symbol (also ein Terminal oder ein Nichtterminal) enthalten

Legende für die Spalte *Abgeschlossenheit*

- $\complement$ ... Komplementbildung
- $\circ$ ... Konkatenation
- $\cap$ ... Schnittmenge
- $\cup$ ... Vereinigungsmenge
- $*$ ... Kleenescher Abschluss

In der obigen Tabelle werden somit mit deutschen/lateinischen Großbuchstaben Nichtterminalsymbole dargestellt, $A, B \in N$ mit deutschen/lateinischen Kleinbuchstaben Terminalsymbole $a \in \Sigma$ und griechische Kleinbuchstaben werden verwendet, wenn es sich sowohl um Nichtterminal als auch um Terminalsymbole handeln kann. (Achtung: $*$ und $^+$: ein griechischer Kleinbuchstaben kann somit für mehrere Terminal- oder Nichtterminalsymbole stehen!)

Q: https://de.wikipedia.org/wiki/Chomsky-Hierarchie



bisher ist weder Noam CHOMSKY (1928 -) noch einem anderen Forscher oder Forscher-Team eine vollständige, mathematische Beschreibung einer natürlichen Sprache gelungen Hauptgrund sind auch die Mehrdeutigkeiten in den Sprachen selbst

Auch wenn es sich auf den ersten Blick so ähnlich anhört, das sogenannte "Wort-Problem" hat nichts mit der Mehrdeutigkeit von Wörtern in natürlichen Sprachen zu tun. Beim "Wort-Problem" stellen wir uns oder einem Automaten die Frage, ob irgend ein Wort zur betrachteten Sprache gehört. Das hört sich für uns Menschen, die jeden Tag mit Worten und Sprache leben recht einfach an, aber der Teufel liegt im Detail. Denken Sie z.B. an das Wort "trekzyck". Ist das ein Wort der menschlichen Sprache? Für einige Sprachen existieren "DUDEN" oder Wörterbücher. Hier könnte man formal prüfen, ob das Wort drinsteht. Aber natürliche Sprachen sind nicht auf die "DUDEN" usw. beschränkt. Natürlich können neue Wörter hinzukommen oder in anderen Sprachen / Slang's usw. existieren. Wir können die Frage zwar beantworten, wenn wir das Wort in einem der Bücher finden, aber was wenn nicht? Trotzdem kann das Wort ein gültiges Sprach-Element sein. Es könnte gerade erfunden / definiert worden sein od. einen wenig bekannten Gegenstand beschreiben.

Definition(en): Wort-Problem
Das Wort-Problem ist das Entscheidungs-Verfahren, ob ein gegebenes Wort zur betrachteten Sprache gehört. Das Wort-Problem ist **entscheidbar**, wenn es eine Funktion gibt, nach der das Wort berechnet / abgeleitet werden kann.
Das Wort-Problem ist die Frage, ob für ein Wort aus einem Alphabet ($w \in \Sigma^*$) die Zugehörigkeit zur von einer Grammatik erzeugten Sprache ($w \in L(G)$) besteht.

Zum Wort-Problem zu den einzelnen Sprach-Typen nach CHOMSKY gibt es dabei folgende Aussagen:

- Für Sprachen vom Typ 0 ist das Wort-Problem rekursiv aufzählbar und nicht entscheidbar (unentscheidbar).
- Das Wort-Problem für Sprachen vom Typ 1 ist entscheidbar. Dabei ist der Zeitaufwand höchstens expotentiell und der Speicherbedarf exakt linear zur Wortlänge.
- Für Sprachen vom Typ 2 gibt Algorithmen, die das Wort-Problem entscheiden können, wobei der Zeitbedarf höchsten kubisch und der Speicherbedarf höchtens quadratisch von der Wortlänge abhängig.
- Das Wort-Problem für Sprachen vom Typ 3 ist durch deterministische endliche Automaten (DEA) entscheidbar. Von der Wortlänge ergibt ein linear steigender Zeitaufwand bei konstantem Speicherbedarf.

Die Assagen sind an sich für uns interessant. Beweisen werden wir soetwas hier nicht – wir glauben einfach dem großen CHOMSKY. Im Detail sind die Aussagen sehr spezifisch und auch nicht so einfach abzuleiten.

Für die Sprachen vom Typ 1 kann mit folgendem Algorithmus geprüft werden, ob ein Wort w in der Sprache L enthalten ist:

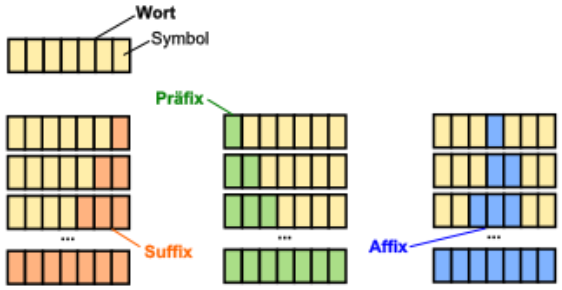
1. berechne die Wortlänge n = |w|
2. setze m = 0
3. setze (kleinste Sprach-Ebene) T(0,n) = Startsymbol
4. wiederhole
   - 4.1. berechne T(m+1,n)
   - 4.2. inkrementiere m

   solange_bis Wort_enthalten (w in T(m,n)) oder Wortmenge_bleibt_gleich (T(m+1,n) = T(m,n))

???Zuordnung?:

Für Untersuchungen von Wörtern sind neben der Zugehörigkeit zu einer Sprache häufig auch bestimmte Charakteristika interessant. So braucht man z.B. manchlal die Aussage, ob eine Wort mit einem bestimmten Teil / einer bestimmten Zeichen-Folge beginnt. Wir suchen also einen Präfix. Präfixe kennen wir in unserer natürlichen Sprache zu genüge. Typische Präfixe sind "un", "a", um", ver".

Beispiele:
ungesund; **un**sensibel; **umver**teilen; **a**biotisch; …



Suffixe sind entsprechende Anhänge / Wort-Enden. In der deutschen Sprache sind das z.B. "ung", keit" und "ig".

Beispiele:
fließ**ig**; Gemütlich**keit**; Vernalisier**ung**; …

Mit Affixen sind Wortstämme / Teilwörter / innere Wort-Teile gemeint.
In Worten sind häufig alle drei Teile (Präfix, Affix, Suffix) vorhanden. Präfixe und Suffixe sind dabei nicht immer vorhanden. Oft ist nur einer von beiden im Wort einer natürlichen Sprache enthalten.
In der Theoretischen Informatik wird allerding die Abgrenzung der Wort-Teile weit offener gesehen. Hier gibt keinen inhaltlichen (semantischen) Bezug. Ob wir mit dem Wort "unmöglich" arbeiten oder mit "unlmögich" ist der Wort-Analyse völlig egal. Suchen wir die – typisch deutschen Prä- bzw. Suffixe "un" und "lich", dann ergibt sich für beide Worte das gleiche Ergebnis – sie sind enthalten. Selbst der Affix "mög" ist in beiden Worten zu erkennen.
Aus der Sicht der Theoretischen Informatik ist aber auch "unm" oder "unmö" ein möglicher Präfix. Natürlich nun nicht mit gleichen Test-Ergebnissen für beide Worte. Was im Deutsch-Unterricht gar nicht geht, ist in der TI ein ständiges Thema. Jede beliebige Zeichen-Kombination kann z.B. als Präfix aufgefasst werden. Somit wäre "unmöglich" auch ein Präfix von "unmöglich".

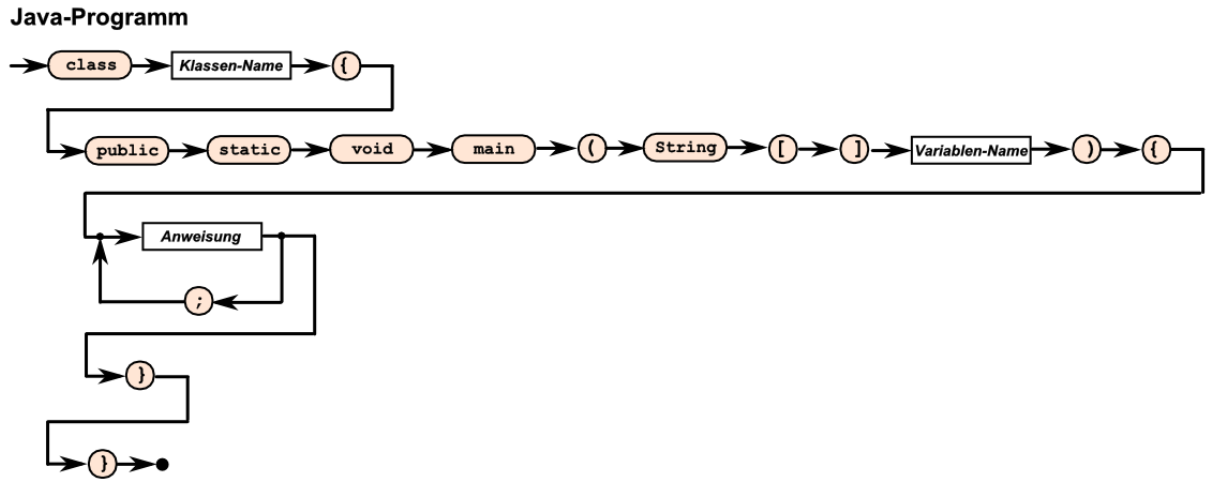
böse Frage zwischendurch:
Ist das nicht Quatsch? Erläutern Sie Ihre Position dazu!

Aber es ist auch Affix und Suffix. In der TI interessieren nur die reinen Zeichen und ihre Kombinationen. Jedweder Bezug zur deutschen oder einer anderen Sprache ist eher zufällig und häufig auch als Mittel der Veranschaulichung, aber auch zur Verwirrung / Verkomplizierung gedacht (z.B. in LK's). Die Theoretische Informatik ist eben eine abstrakte Wissenschaft und möchte entsprechend bedient werden.
Die Krönung solcher TI-Betrachtungen ist die Frage, wieoft ist das leere Wort ε als Präfix, Affix und Suffix in einem Wort vorkommt (wenn ε in der Sprache zugelassen ist)? Die vielleicht überraschende Antwort ist: beliebig / unendlich oft.

Aufgaben:

1. ***Geben Sie drei weitere typisch deutsche Präfixe an! Nennen Sie jeweils zwei Beispiele passender (deutscher) Wörter!***
2. ***In der TI kann man aus dem Wort "Gebrauchsanweisung" auch einen oder mehrere Suffixe ableiten. Nennen Sie alle Suffixe!***
3.

Syntax-Diagramm für ein JAVA-Programm (Grund-Struktur):



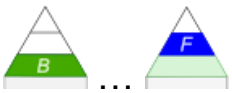
Die Linienführung im obigen Syntax-Diagramm wurde in Anlehnung an die Schreibung von Java-Programmen in den verschiedensten Editoren so ausgeführt. Sachlich hätte auch alles eine Kette (mit der kleinen Schleife bei Anweisung) sein können.

```
class ??? {public static void main(String[] args) {
    ???;
    }
}
```

	Grammatik				
Sprache	**Al-pha-bet**	**Produktionen Regeln**			**Bemerkungen**
alle					
formale					
Typ 0 rekursiv ab-zählbare, nicht-kontext-sensitiv allg. Regel-Spr.		*unbeschränkt*		TURING-Maschine	*Teilmenge von "alle Sprachen"*
entscheidbare					
Typ 1 Kontext-sensitive, nicht-regulär Längen-monotone S.		a <X> b → a <Y> b *Länge der Zeichenketten auf beiden Seiten gleich groß!* a <X> b → a c b <S> → ε		linear be-schränkte TURING-Maschine	*Teilmenge von "Typ-0-Sprachen"* *a und b sind der Kontext (welcher notwendig ist)*
expansive		a → b <S> → ε			
Typ 2 Kontext-freie, nicht-regulär		<X> → a <Y> b		Keller-Automat	*Teilmenge von "Typ-1-Sprachen"*
linear		<X> → ε <X> → a <Y> b			
Typ 3 reguläre		rechts-linear: <X> → a | a <Y> <X> → ε ***alternativ (für alle Regeln!):*** links-linear: <X> → a | <Y> a <X> → ε		endlicher Automat	*Teilmenge von "Typ-2-Sprachen"*

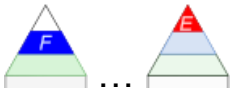
→ kann auch durch ::= ersetzt werden

3.1.z. Simulation von Grammatiken am PC



Neben einigen frei verfügbaren Programmen mit dem primären Thema "Grammatik" im Sinne der Theoretischen Informatik, sticht besonders die logische Programmiersprache PROLOG (z.B. das freie SWI-PROLOG) als Simulator für Grammatiken hervor.
Es existieren weiterhin viele Spezial-Programme zur Bearbeitung von Grammatiken und Automaten (die ja später noch folgen).
Auf der Abitur-Version des Io-Stick's sind die Programme SWI-PROLOG, JFLAP, AutoEdit und Grammatik-Editor quasi als portable Apps verfügbar gemacht.
Persönlich würde ich für viele Zwecke – trotz der nur im englischen verfügbaren Sprach-Version – JFLAP empfehlen. Es bietet unter einer Oberfläche viele Funktionen, die uns im Unterricht interessieren.
Schauen Sie sich die Möglichkeiten der Programme einfach an und entscheiden Sie dann nach Ihren Bedürfnissen.
Als JAVA-Programm sollte JFLAP auch auf den verschiedensten Betriebssystemen laufen.
Als Daten-Dateien benutzt JFLAP sogenannte JFF-Dateien. Die JFF-Dateien sind praktisch XML-Dateien. Sie lassen sich auch in einem Text-Editor erstellen und verändern.

3.1.z.1. Grammatiken bearbeiten / testen mit PROLOG



Eine "einfache deutsche Grammatik" haben wir ja schon weiter vorn mit PROLOG getestet.

Die in der Grammatik benötigten Terminale heißen in PROLOG Atome und sind einfache Konstrukte. Die atomaren Ausdrücke müssen in Kleinbuchstaben geschrieben werden.

```
% Sprache, die beliebig viele a und b enthält; n>0

terminal(a).
terminal(b).

%Regeln
wort(X):-terminal(Y).
wort(X):-wort,terminal(Y).

%Start-Symbol
start(X):-wort(X).
```

In einigen folgenden PROLOG-Programmen wird mit einem selbst-definierten Ableitungs-Operator gearbeitet.

Die Definition steht am Anfang des Programm's. In ihr wird die Priorität, die Anordnung des Operators zu Operanten und das eigentliche Symbol "--->" definiert. Neugierige informieren sich in PROLOG-Büchern über das genaue Funktionieren.

Der Rest des Programm's ist klassische Definition einer Grammatik eben mit dem neuen Ableitungs-Operator.

```
% Programm zur Erzeugung einer Grammatik S → 0S1 | 01
% Definition eines Pfeil-Operators
:- op(900,xfx,--->).

%
varsym(V).

% Defintion des Startsymbols
startsymbol(s).

% Defintion der Grammatik-Regeln
[s] ---> [0,1].
[s] ---> [0,s,1].
```

Q: /2, S. 68/; leicht geänd.: dre

Für eine Palindrom-Sprache könnte die Programmierung den folgenden Quelltext ergeben:

```
% Programm zur Erzeugung eines Wortes
erzeuge(W):- startsymbol(S), [S] ==>* W,
             (member(X,W), varsym(X))

V ==> W:- L ---> R, haenge_an(Links, L, Rechts, V),
          haenge_an(Links, R, Rechts, W).

haenge_an(Links, Mitte, Rechts, A):- append(X,Rechts,A),
                                     append(Links, Mitte, X).
```

Q: /2, S. 68/; leicht geänd.: dre

Eine Testung erfolgt immer in der nebenstehenden Form.

In vielen höheren Programmiersprachen gibt es effektive Funktionen zum Zerlegen einer textzeile in einzelne Zeichen.

Beispiele für Palindrome:
AHA, BOB, EBBE, EGGE, EHE, GAG, KAJAK, LAGER, LAGERREGAL, LEVEL, MARKTKRAM, NEBEN, NEFFEN, NEOZOEN, NUN, OTTO, POP, RADAR, REITTIER, RENNER, RENTNER, ROTOR, STETS,TOT, TUT, UHU

Beispiele für Satz-Palindrome: *(**Achtung!** Leerzeichen werden überlesen.)*
Bau ab!
Bei Liese sei lieb!
Dreh mal am Herd!
Eine Hure ruhe nie.
Er hortet Rohre.
Es eilt Liese!
Nie solo sein!
Sei fein, nie fies.
Sei fies!
Tu erfreut!
Und nu?

FORSYTH-EDWARDS-Notation (FEN) für die Speicherung von Spielständen beim Schach:
besteht aus den 6 Elementen:

Figurenstellung, Am_Zug, Rochade, en_passant, Halbzüge und Zugnummer hintereinander mit Leerzeichen getrennt

Figurenstellung:
acht Zeichenfolgen, die Schrägstriche / getrennt sind hintereinander geschrieben
Zeichengruppe max. 8 Zeichen lang
Kleinbuchstaben kennzeichen schwarze Figuren, Großbuchstaben die weißen
leere Felder werden als Gruppe in Ziffern-Form notiert
Figuren-Codes: p,P … Bauer (); q,Q … Dame (Queen); k,K … König (King)
b,B … Läufer (); n,N … Springer (); r,R … Turm ();
Am_Zug ist entweder w (weiß, white) oder b (schwarz, black)
Rochade entweder König- oder Dame-Zeichen oder bei ausgeführter Rochade ein Strich –
k,K steht für die noch mögliche kurze (Königs-seitige) Rochade
q,Q steht für die noch mögliche lange (Dame-seitige) Rochade
en_passant … wenn im letzten Zug ein Bauer zwei Felder vorgesetzt wird, wird hinter dem Spalten-Buchstaben die übersprungene Spalte - also entweder 3 oder 6 - angegeben
Halbzüge … ist die Anzahl der Halbzüge seit dem letzten Bauern-Zug bzw. dem Schlagen einer Figur
Zugnummer … Nummer des nächsten Zuges beginnend mit 1; nach jedem Zug von Schwarz wird um 1 erhöht

Start-Position eines Schachspiels wäre somit:
`rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq – 0 1`

ipigisi-Sprache für Mathe-Aufgaben der 1. und 2. Klasse
nach: www.inf-schule.de

```
ipisisisisigisisisisisi
```
wird zerlegt in:
```
i   p   isisisi   g   isisisisi
1   +     4       =     5
```

isisisisisi **p** isi **g** isisisi
6 **–** 2 **=** 4

zu prüfende Ausdrücke:

Aufgaben:

1. Setzen Sie die folgenden Aufgaben in die ipigisi-Sprache um!

a)	4 + 3 = 7	b)	7 +1 = 8	c)	6 – 3 = 3
d)	7 + 2 = 5 + 4	e)	5 – 3 = 7 – 5	f)	+3 = 1 + 2
g)	8 + 2 – 4 = –3	h)	5 + 1 = 7 – 3	i)	3 – 2 + 6 = 4 – 5 + 8

2. Prüfen Sie die Gültigkeit der folgenden Ausdrücke in der ipigisi-Sprache!

a)	isisisisisipisisigisisisisisisisisi	b)	isisisisisisisisimisisisisisisigisi
c)	igisisisisisisisisimisisisisisisi	d)	isisisisisisisisisigisisisisisipisisi
e)	isisisigisisisisisisimisigisisipi	f)	isisisisimisisisipisisimisigisisisisi

3.

ASCII-Emoticons
z.B.:
```
:-D    ;o0    :[    ;-)    :'C    x)    =)    ;-P    :-b    …
```

Hier noch einige schöne Beispiele für Akzeptoren in PROLOG:

Akzeptor für Double-Zahlen (entsprechend dem JAVA-Syntax)

	Quellcode	Kommentar(e)
1	`% Autor:  T. Hempel`	%-Zeichen sagt: es folgt ein Kommentar; hier Autor, Quelle und das Erstellungsdatum
2	`% Quelle: Zimmermann, H.-U.: "Die Implementation kontextfreier Grammatiken in PROLOG".`	
3	`%         In LOG IN 21 (2001) Heft 5/6 S. 68ff.`	
4	`% Datum: 11.10.2010`	
5		
6	`% Symbol   Interpretation`	kurze Darstellung der verwendeten Symbole und der Struktur der Regeln
7	`% -->      besteht aus`	
8	`% ,        und`	
9	`% |        oder`	
10	`% []       Markierung eines Wortes, das immer aus Kleinbuchstaben besteht`	
11	`% ''       Markierung einer Variable, die immer mit einem Grossbuchstaben beginnt`	
12		
13	`% Aufruf: ?- phrase('Double',X).`	so wird getestet
14		
15	`'Double'` `--> 'Vorzeichen','Zahl','Nachkomma','Exponent'.`	Definition der Regeln und der Daten-Basis
16	`'Vorzeichen'` `--> []|['+']|['-'].`	
17	`'Ziffer'` `--> [0]|[1]|[2]|[3]|[4]|[5]|[6]|[7]|[8]|[9].`	
18	`'Zahl'` `--> 'Ziffer'|'Ziffer','Zahl'.`	
19	`'Nachkomma'` `--> []|['.'],'Zahl'.`	
20	`'Exponent'` `--> []|'Exponentenzeichen','Zahl'.`	
21	`'Exponentenzeichen'` `--> [e].`	

Die nachfolgenden Aufgaben sollen vor allem den Umgang mit dem Prolog-System und dem Quell-Text schulen. Unter Verwendung der Beipiele sollte es dann später auch gelingen, gleichartige Akzeptoren zu "programmieren".

Aufgaben:

1. Testen Sie die nachfolgenden Ausdrücke auf ihre Gültigkeit als Double-Zahl!

a)	-23.75e3	b)	23.952E-38	c)	-8,3E4.3	d)	0.0e0.0	e)	-.3e7
f)	0.0000000003938	g)	eE	h)	2+e7	i)	6^2	j)	0000000629

2. Verändern Sie den Akzeptor so, dass er nun (nur noch) Ganze Zahlen erkennt! In einer zweiten Stufe verbinden Sie die beiden Akzeptoren so, dass ein neuer Akzeptor sowohl Double- als auch Longint-Zahlen (ohne Grenzen) erkennt!

3. Konstruieren Sie einen Akzeptor für Natürliche (Dezimal-)Zahlen!

4. Gesucht wird ein Apzeptor für Oktal-Zahlen ohne Komma-Stellen!

5. Erstellen Sie einen Akzeptor für Hexadezimal-Zahlen (ohne Nachkomma-Stellen)!

6. Testen Sie den Hexadezimal-Akzeptors mit den folgenden Ausdrücken: (Verbessern Sie den Akzeptor bei offensichtlichen Fehlern!)

a)	8A74	b)	3963	c)	0EE0EE	d)	E5EG	e)	FF07,04
f)	06 9F f5 a5	g)	eE	h)	abcfde	i)	88:7E:9A:61:45:fE	j)	192.168.20.67

für die gehobene Anspruchsebene:

7. Erstellen Sie ein Regel-System in Prolog, dass es gestattet imaginäre Zahlen (der Form: -2,3+4,1i) zu erkennen! (Wenn Ihnen imaginäre Zahlen nichts sagen, dann können Sie auch kurz im Internet oder in Mathe-Lexika informieren, Für die Lösungs der Aufgabe ist das aber nicht notwendig.)

Prüfer für Lacher (Hahah!-Sprache)

	Quellcode	Kommentar(e)
1	`% Autor:  L. Drews`	%-Zeichen sagt: es folgt ein Kommentar; hier Autor, Quelle und das Erstellungsdatum
2	`% Quelle: abgeleitet aus Grammatik-Akzeptor und Lachautomat von T. Hempel`	
3	`% Datum: Sep. 2018`	
4		
5	`% Symbol   Interpretation`	kurze Darstellung der verwendeten Symbole und der Struktur der Regeln
6	`% -->      besteht aus`	
7	`% ,        und`	
8	`% |        oder`	
9	`% []       Markierung einer Lachsilbe, immer aus Kleinbuchstaben bestehend`	
10	`% ''       Markierung eines Zeichens / einer Zeichenkette`	
11		
12	`% Aufruf: ?- phrase('Lacher',X).`	so wird getestet
13	`%         ?- phrase('Lacher',['ha','ha','hah','!').`	
14		
15	`'Lacher'     --> 'Vorlacher','Nachlacher','Ende'.`	Definition der Regeln und der Daten-Basis
16	`'Vorlacher'  --> []|[Vorlacher].`	
17	`'Nachlacher' --> 'hah'.`	
18	`'Ende'       --> '!'.`	

Q: für Grammatik-Akzeptor und Lachautomat: T.Hempel; Weiterbildung HILF!2018

Einen passenden akzepierenden Automaten (Was auch immer das ist?) stellen wir weiter hinten – ebenfalls in Prolog umgesetzt – vor (→).

Aufgaben:

1. *Machen Sie aus dem Lacher-Prüfer einen, der den Weihnachtsmann erkennt (Hohoh!-Sprache)! Das kürzeste Wort, welches der Weihnachtsmann normalerweise sagt, ist: "Hohohoh!".)*
2.
3. *Erstellen Sie einen Kuh-Prüfer, der eine Kuh am mehr oder weniger langgezogenem "muuuh" erkennt!*

Symbolisches Rechnen - Differenzieren

```
d(X,X,1):- !.
d(C,X,0):- atomic(C),C\=X,!.
d(~X,X,~DX):- d(X,X,DX).
d(C,X,0):- number(C).
d(pot(X,N),X,N*pot(X,N1)):- N>1, N1 is N - 1.
d(sin(Y),X,cos(Y)*DY):- d(Y,X,DY).
d(cos(Y),X,~sin(Y)*DY):- d(Y,X,DY).
d(exp(X),X,exp(X)).
d(log(X),X,1/X).
d(pot(A/N),X,N*pot(A,N1)*DA):- N>1, N1 is N - 1, d(A,X,DA).

d(F+G,X,DF+DG):- d(F,X,DF), d(G,X,DG).
d(F-G,X,DF-DG):- d(F,X,DF), d(G,X,DG).
d(F*G,X,DF*DG):- d(F,X,DF), d(G,X,DG).
d(F/G,X,((DF*G) - (F*DG))/(G*G)):- d(F,X,DF), d(G,X,DG).

d(X+C,X,DX):- atomic(C), C\=X, d(X,X,DX), !.
d(C+X,X,DX):- atomic(C), C\=X, d(X,X,DX), !.
d(X-C,X,DX):- atomic(C), C\=X, d(X,X,DX), !.
d(C-X,X,DX):- atomic(C), C\=X, d(X,X,DX), !.

vereinfadd(0+X,X).
vereinfadd(X+0,X).
vereinfsub(X - 0,X).
vereinfsub(0 - X,-X).
vereinfmult(X*1,X).
vereinfmult(1*X,X).
vereinfmult(0*X,0).
vereinfmult(X*0,0).
vereinfdiv(X/1,X).
vereinfpot(pot(X,1),X).

vereinfachen(A,A):- atomic(A), !.
vereinfachen(X,Z):- vereinfadd(X,Z),!.
vereinfachen(X,Z):- vereinfsub(X,Z),!.
vereinfachen(X,Z):- vereinfmult(X,Z),!.
vereinfachen(X,Z):- vereinfdiv(X,Z),!.
vereinfachen(X,Z):- vereinfpot(X,Z),!.
vereinfachen(X+Y,X1+Y1):- vereinfachen(X,X1), vereinfachen(Y,Y1).
vereinfachen(X*Y,X1*Y1):- vereinfachen(X,X1), vereinfachen(Y,Y1).
vereinfachen(X - Y,X1 - Y1):- vereinfachen(X,X1), vereinfachen(Y,Y1).
vereinfachen(X/Y,X1/Y1):- vereinfachen(X,X1), vereinfachen(Y,Y1).
vereinfachen(pot(X,Y),pot(X1,Y1)):- vereinfachen(X,X1), vereinfachen(Y,Y1).
diff(Y,X,Z):- d(Y,X,X1), vereinfachen(X1,Z).
```

Aufruf z.B.:

```
diff(sin(x^2),x,X1).
```

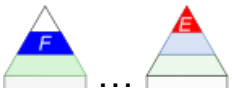
```
/* Listing 1
   Symbolisches Differenzieren in PROLOG
   nach Clocksin/Mellish
   ~: Negation
   ^: Exponentiation; manche Interpreter
      gestatten **
   Aufruf: d(ln(x)*sin(x^2+3),x,K).
   modifiziert von Alex Pretschner       */

d(X,X,1) :- 1.          /* x'=1         */
d(C,X,0) :- atomic(C).  /* Konstante    */
```

```
d(~U,X,~A) :- d(U,X,A). /* Negation     */
/* Summenregel */
d(U+V,X,A+B) :- d(U,X,A), d(U.X,B).
d(U-V,X,A-B) :- d(U,X,A), d(U,X,B).
/* konst. Paktor */
d(C*U,X,C*A) :- atomic(C), C\= X, d(U,X,A), !.
/* Produktregel */
d(U*V,X,B*V+A*U) :- d(U,X,B), d(V,X,A).
d(U/v,X,A) :- d(U*VA(~1),X,A). /* Quotient */
/* Funktionsableitungen */
d(ln(U),X,A*U^(~1)) :- d(U,X,A).
d(sin(U),X,A*cos(U)) :- d(U,X,A). /* usw. */
/* Potenzen */
d(U^V,X,B*U^V*ln(U)+V*U^V*A*U^(~1)) :- d(U,X,A), d(V,X,B).
```

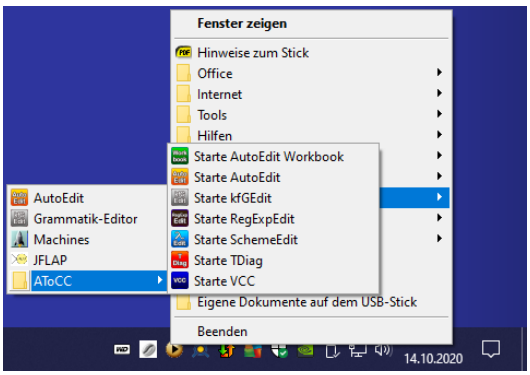
Q: PRETSCHNER: "symbolisches Differenzieren: Man macht's prozedural.-ST-Magazin 05/1994
→ http://www.stcarchiv.de/stc1994/05/symbolisches-differenzieren

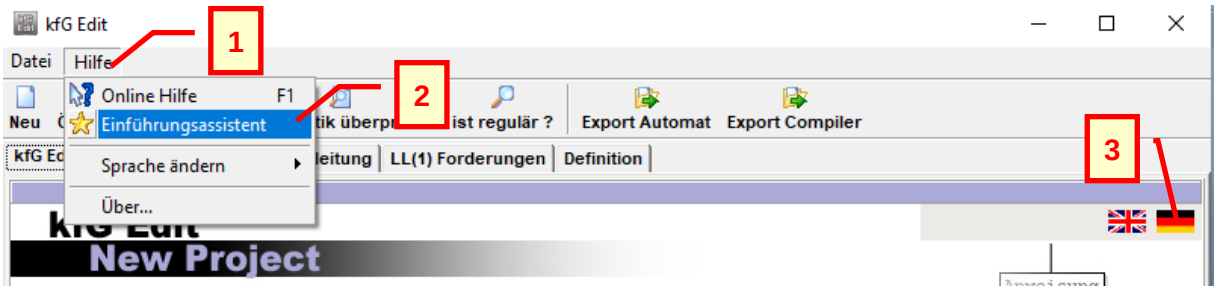
3.1.z.2. Grammatiken bearbeiten / testen mit JFLAP

3.1.z.3. Grammatiken bearbeiten / testen mit dem kfG-Editor aus AtoCC



Die Programm-Sammlung / Lern-Umgebung **AtoCC** (**A**utomata **to** **C**ompiler **C**onstruction (neuerdings auch: "from automaton to compiler construction")) von Genesis-X7 Software enthält diverse Programme (App's) für den Umgang mit Sprachen, Grammatiken und Automaten. Dazu gibt es einen Leistungs-fähigen Editor für die Erstellung und Bearbeitung von Aufgaben(-Blättern) z.B. für Schüler und Studenten.
Für die Bearbeitung von Sprachen und / oder Grammatiken nutzen wir das Programm "**kfGEdit**" (Editor für kontextfreie Sprachen).
In der nachfolgenden Besprechunggehen wir davon aus, dass die Programm-Sammlung vom IoStick (→ https://www.tinohempel.de/info/info/IoStick/index.html) genutzt wird.
Nach dem Start des IoStick's sehen wir in der Task-Leiste das eingefärbte Dreiecks-Symbol. (Die Farbe kann anders sein!)
Klickt man auf dieses Symbol, dann bekommt man das Menü des IoStick's angezeigt. Unter dem Menü-Eintrag "Sprachen und Automaten" befindet sich das Programm "Grammatik Editor". Eigentlich ist ein Link auf das Programm "kfGEdit" aus der Programm-Sammlung "AToCC".
Ein guter Start vor dem eigentlichen Nutzen ist der "Einführungsassistent" (**2**) unter dem Menü-punkt "Hilfe" (**1**).

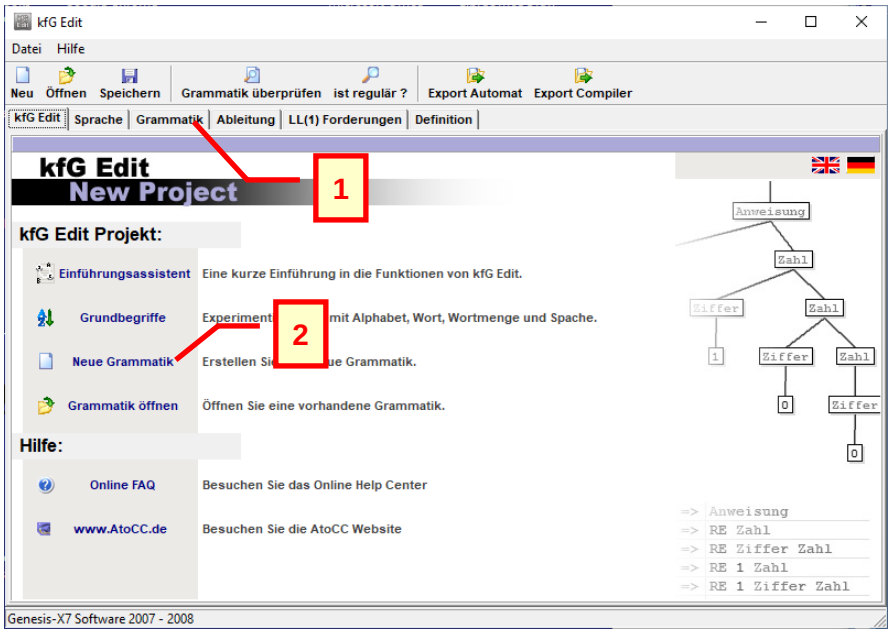




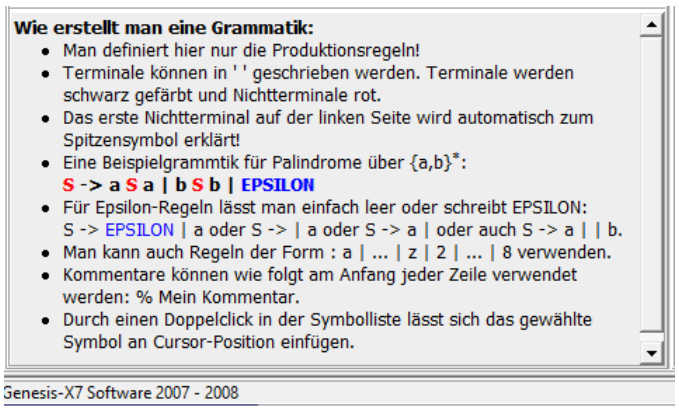
Der Einführungsassistent erklärt kurz die Funktionsweise und das Arbeiten mit dem Programm.
Ev. sollte man schon vorher die Programm-Sprache einstellen (**3**). Dann muss man sich ev. nicht mit englisch-sprachigen Ausdrücken herumschlagen.
Ebenfalls gut als Einstieg eignet sich ein Video-Tutorial (→ http://www.atocc.de/cgi-bin/atocc/site.cgi?lang=de&site=tutorials) auf der Webseite des Projekts (→ http://www.atocc.de).

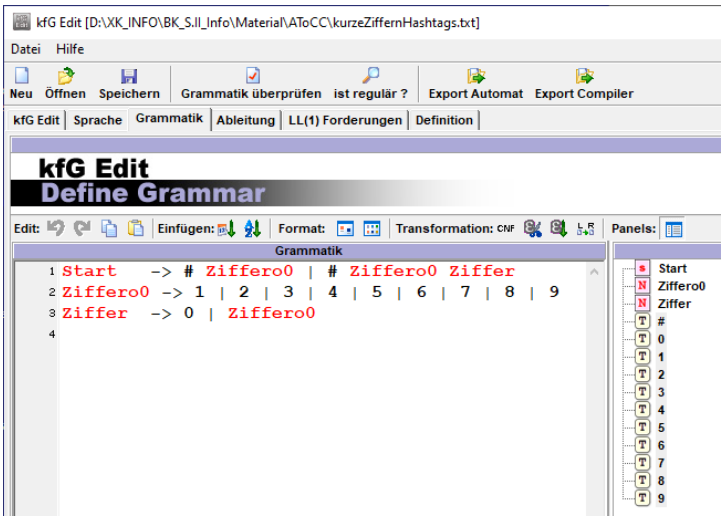
Erstellen einer neuen Grammatik

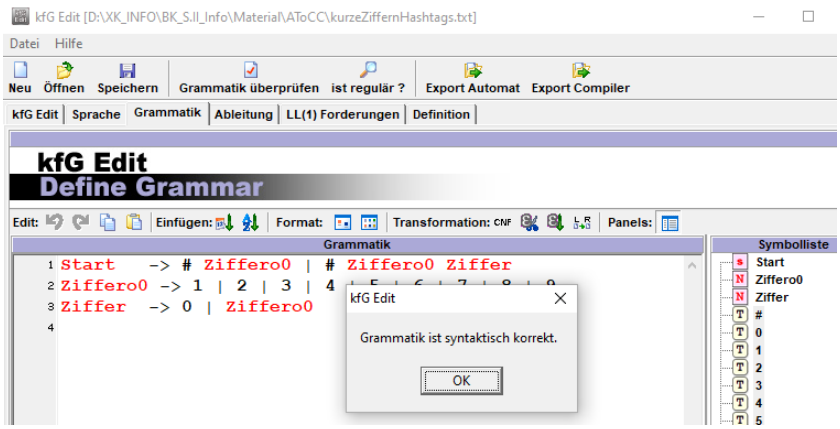
Die Projekt-Start-Seite von kfGEdit bietet die wichtigsten Start-Möglichkeiten an. Bei "Neue Grammatik" gelange wir in den "Grammatik"-Eingabe-Bereich.
Gleiches erreicht man direkt über den Reiter "Grammatik".
Hier gibt unten eine Kurz-Beschreibung.
Es empfiehlt sich einfach das Programm-Fenster zu vergrößern (ev. maximieren), um sich dann diesen Beschreibungs-Teil vollumfänglich anzeigen zu lassen.

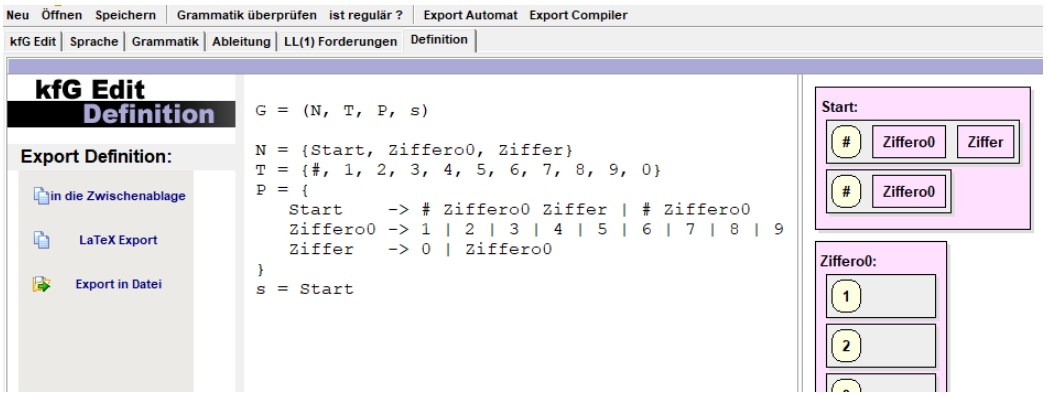


Die Bereiche lassen sich an den Grenzen verkleinern oder vergrößern.
Wird ein Begriff auf der linken Seite eines Pfeil's (->) geschrieben, dann ist es automatisch ein Nicht-Terminal. Es reicht z.B. ein Buchstabe. Die Nicht-Terminale können auch sprechend sein – also z.B. ganze Worte. Vor allem bei größeren Grammatiken macht das Sinn. Alle anderen Einträge sind dann Terminale. Will man das besonders deutlich machen, wie z.B. bei Leerzeichen oder mehrzeichigen Gruppen, dann kann man einfache Hochkommata zum Abgrenzen verwenden.



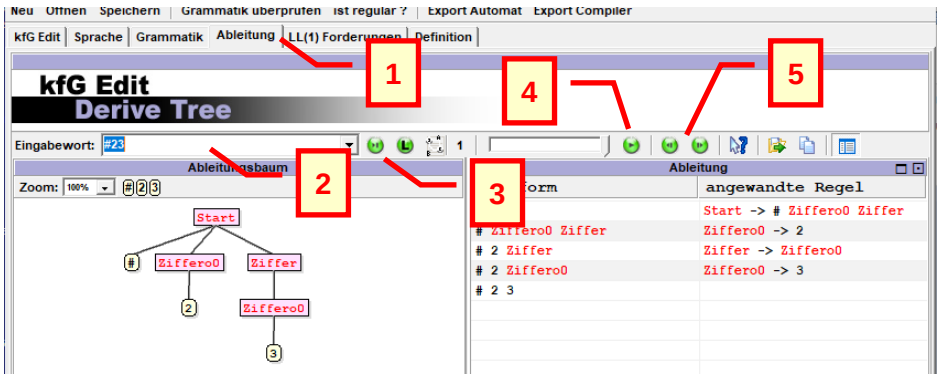






Die Blöcke rechts stellen eine Art Syntax-Diagramm dar.

Arbeiten mit einer Grammatik



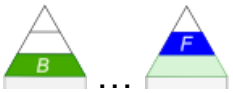
(**2**) Eingabe und Liste der Eingabe-Worte / Test-Worte / zu prüfenden Zeichenketten
(**3**) schneller Test auf Ableitbarkeit (? Ist Wort Element der Sprache?)
(**4**) animierte Ableitung
(**5**) Schritt-Steuerung der Ableitung (Vor und Zurück)

(**L**) bzw. (**R**) bestimmt die Art- und Weise der Ableitung
L bedeutet **Links-Ableitung**, d.h. bei einer Regel wird die rechte Seite jeweils von links Element für Element aufgelöst. Es wird probiert, ob ein Terminal passt. Im Falle eines Nicht-

Terminal's wird mit diesem in gleicher Weise fortgeführt, bis das Nicht-Terminal vollständig in Terminale zerlegt wurde. Dann wird mit dem nächsten (rechts stehenden) Element weitergemacht.
Bei einer **Rechts-Ableitung** wird die rechte Seite der Regel auch von rechts zerlegt.

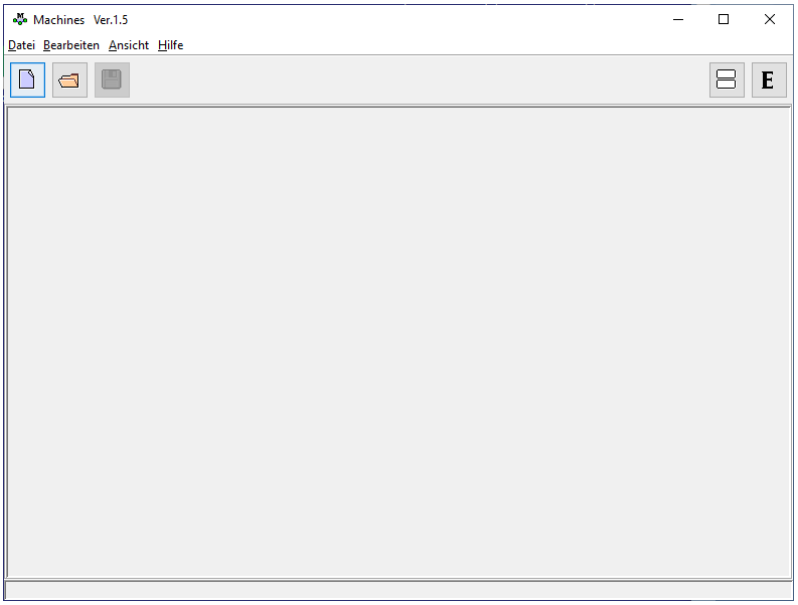
weiterführende Links:
http://www.atocc.de (Projekt-Webseite)
http://www.atocc.de/cgi-bin/atocc/site.cgi?lang=de&site=tutorials (Tutorials)
https://flaci.com/home/ (neue online-Version(en) der AtoCC-Programme; (!!! Wahrscheinlich nicht für Prüfungen oder Klausuren zugelassen!)

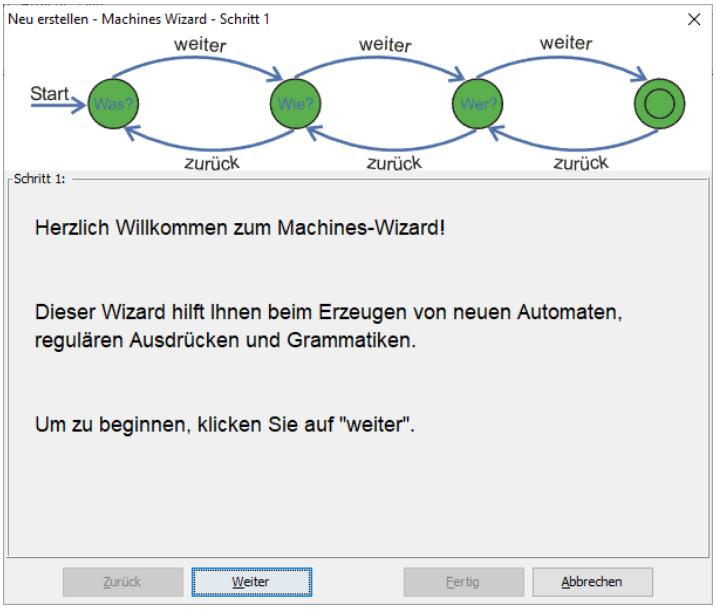
3.1.z.4. Grammatiken bearbeiten / testen mit Machines

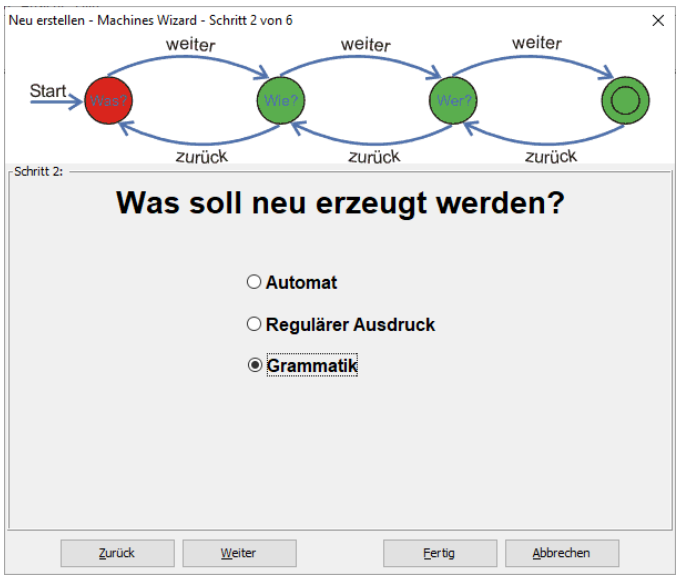


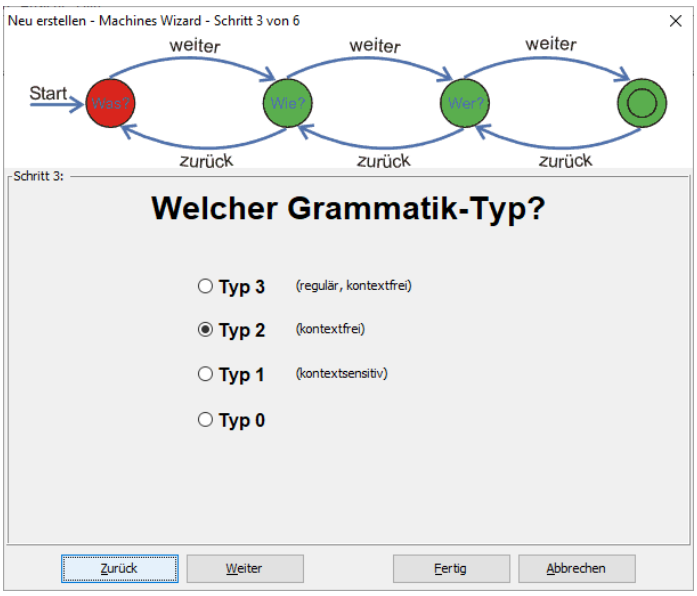
benötigt Java-Umgebung
auch für Linux verfügbar
auf dem IoStick vorhanden und lauffähig

einfache Bedienung; kaum Einarbeitungs-Zeit notwendig
auch für die Bearbeitung von regulären Ausdrücken (→) und diversen Automaten (→ 3.1.z.4. Grammatiken bearbeiten / testen mit Machines) nutzbar
didaktisch orientiert
Konzentration auf das Wesentliche
übersichtlich
wirkt manchmal einwenig altbacken



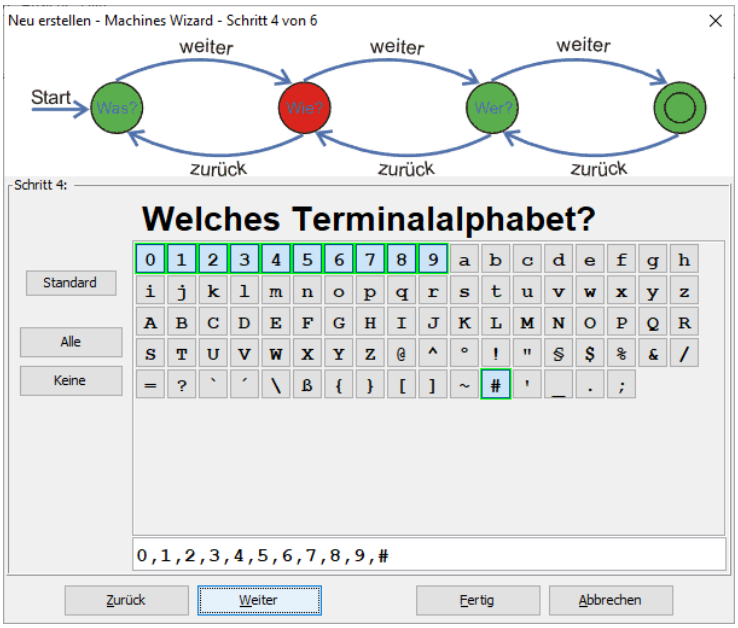

Neu erstellen - Machines Wizard - Schritt 1
weiter
weiter
weiter
Start
Was?
Wie?
Wer?
zurück
zurück
zurück
Schritt 1:
Herzlich Willkommen zum Machines-Wizard!
Dieser Wizard hilft Ihnen beim Erzeugen von neuen Automaten, regulären Ausdrücken und Grammatiken.
Um zu beginnen, klicken Sie auf "weiter".
Zurück
Weiter
Fertig
Abbrechen

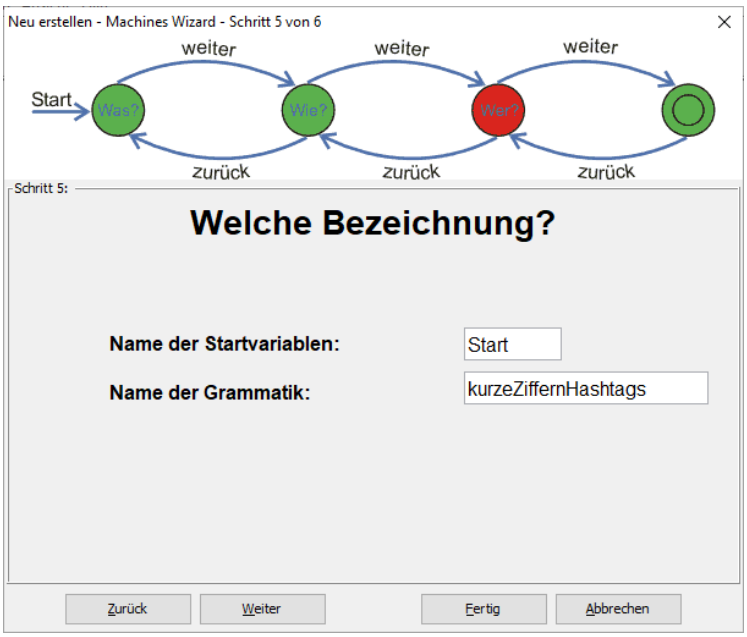

Neu erstellen - Machines Wizard - Schritt 2 von 6
weiter
weiter
weiter
Start
Was?
Wie?
Wer?
zurück
zurück
zurück
Schritt 2:
Was soll neu erzeugt werden?
Automat
Regulärer Ausdruck
Grammatik
Zurück
Weiter
Fertig
Abbrechen

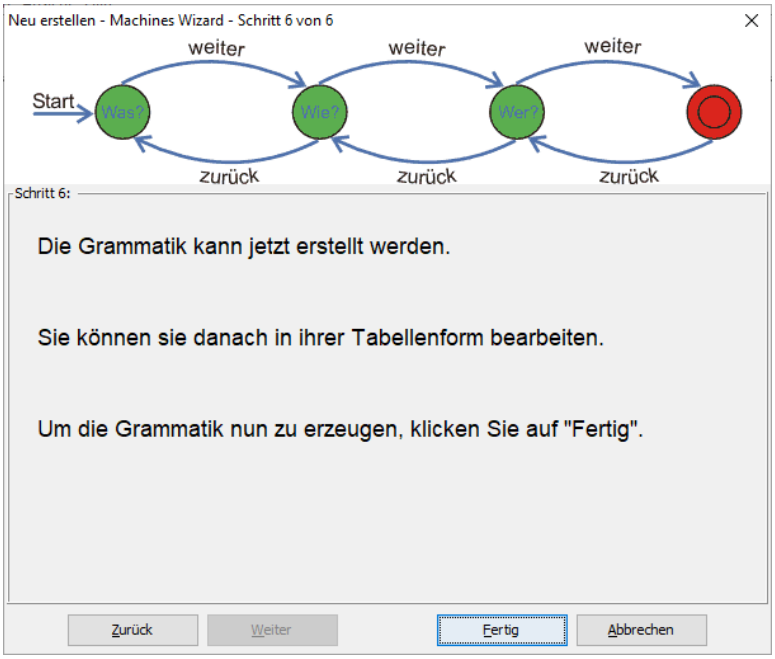

Neu erstellen - Machines Wizard - Schritt 3 von 6
weiter
weiter
weiter
Start
Was?
Wie?
Wer?
zurück
zurück
zurück
Schritt 3:
Welcher Grammatik-Typ?
Typ 3 (regulär, kontextfrei)
Typ 2 (kontextfrei)
Typ 1 (kontextsensitiv)
Typ 0
Zurück
Weiter
Fertig
Abbrechen

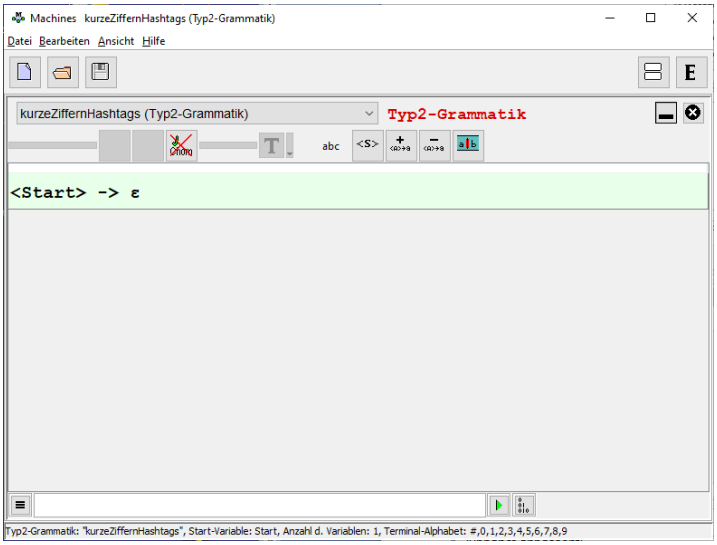
hier genau Arbeiten

später scheint die Änderung des Alphabet's schwierig zu sein

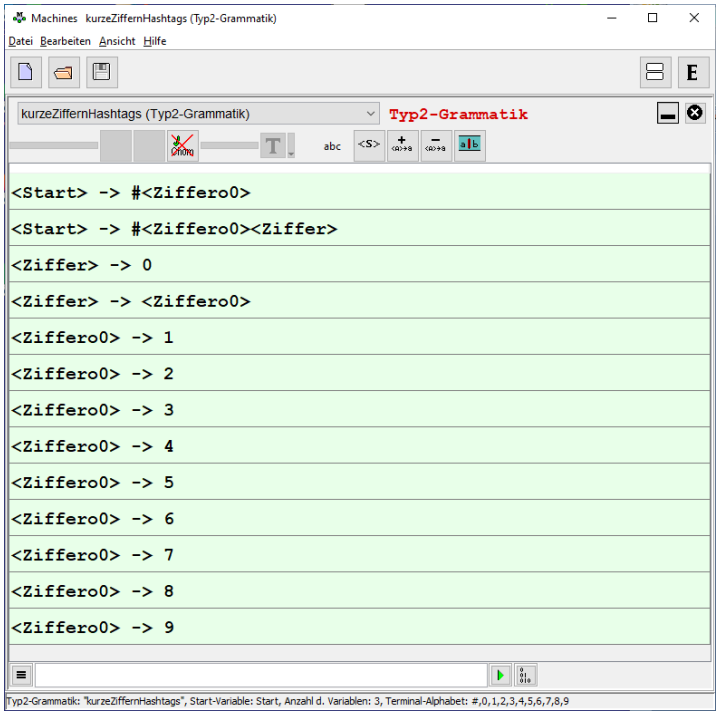




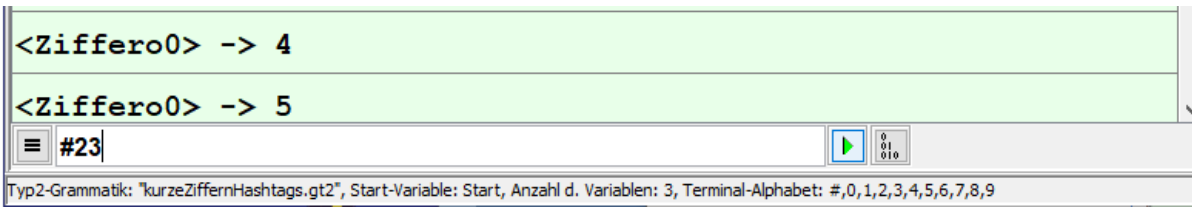

Neu erstellen - Machines Wizard - Schritt 6 von 6
weiter
weiter
weiter
Start
Was?
Wie?
Wer?
zurück
zurück
zurück
Schritt 6:
Die Grammatik kann jetzt erstellt werden.
Sie können sie danach in ihrer Tabellenform bearbeiten.
Um die Grammatik nun zu erzeugen, klicken Sie auf "Fertig".
Zurück
Weiter
Fertig
Abbrechen

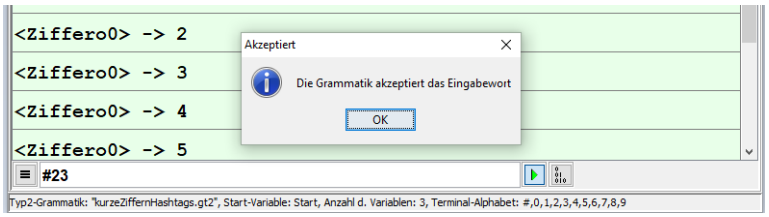

Machines kurzeZiffernHashtags (Typ2-Grammatik)
Datei Bearbeiten Ansicht Hilfe
kurzeZiffernHashtags (Typ2-Grammatik)
Typ2-Grammatik
<Start> -> ε
Typ2-Grammatik: "kurzeZiffernHashtags", Start-Variable: Start, Anzahl d. Variablen: 1, Terminal-Alphabet: #,0,1,2,3,4,5,6,7,8,9

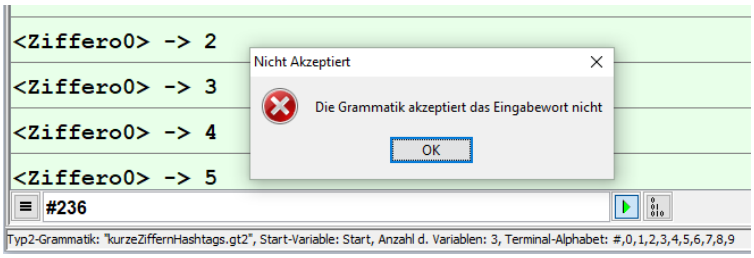

Produktion editieren
#
0
1
2
3
4
5
6
7
8
ENTF
<>
->
|
ε
<Start>
<Ziffero0>
:art> -> #<Ziffero0> | #<Ziffero0> <Ziffer
Übernehmen (OK)
Verwerfen (Abbrechen)



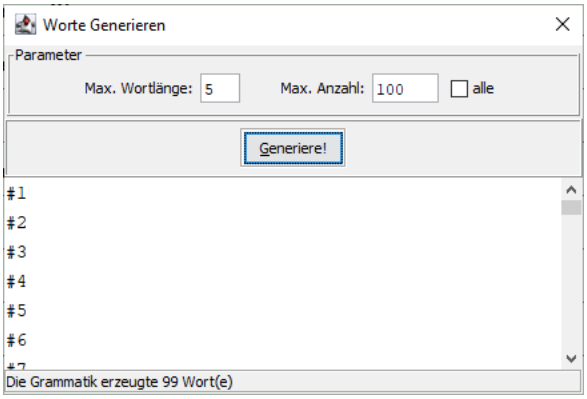
Testen / Ableiten



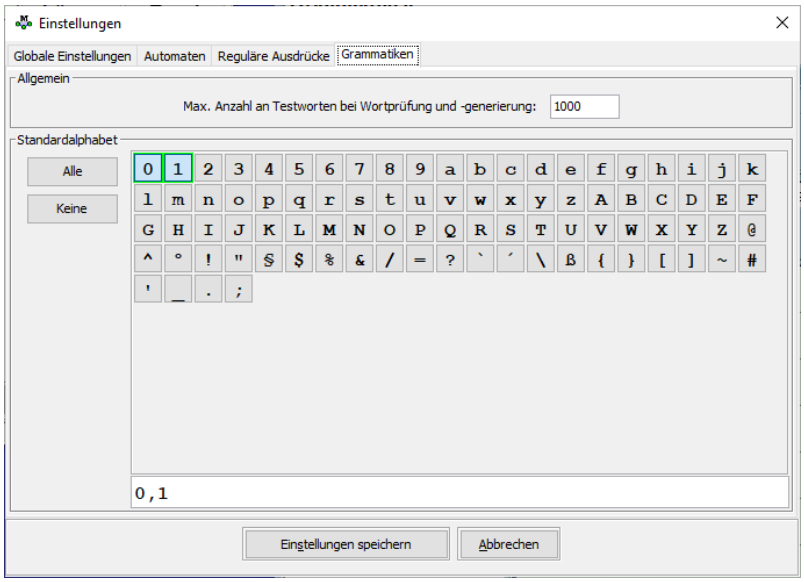




Worte erzeugen / generieren



1. Alphabet anpassen
"Bearbeiten" "Einstellungen"



3.1.z. Kurz-Vorstellung der Sprach-Typen nach CHOMSKY

von CHOMSKY selbst wurden die Grammatiken noch Phrasenstruktur-Grammatiken genannt

immer charakterisiert durch ein Quad-Tupel (4-Tupel) der folgenden Elemente:

G = (T, N, P, S)

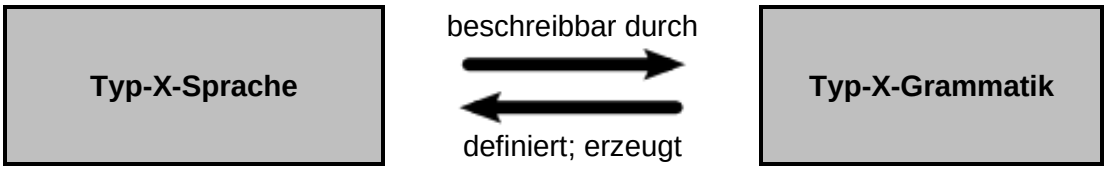
Grammatik-Elemente nach CHOMSKY

• **Menge an Terminal-Symbolen T**	sind die eigentlichen Zeichen / Buchstaben / Symbole der Sprache (aus ihnen werden die Wörter (der Sprache) zusammengesetzt
• **Menge an Nicht-Terminal-Symbolen N**	Hilfs-Wörter / Meta-Symbole einer Grammatik, die nur in den Regeln, aber nicht mehr in der produzierten Sprache vorkommen dürfen
• **Menge an (Produktions-)Regel P**	Ableitungen / regeln der Grammatik Ersetzung von Nicht-Terminalen durch Terminale und / oder Nicht-Terminale
• **Start-Symbol / Anfangszeichen S**	erstes abzuleitendes (Nicht-Terminal-)Zeichen Start-Nicht-Teminal

Wiederholung von → 3.1. Sprachen und Grammatiken

abstrakte Definition(en): formale Grammatik
Eine formale Grammatik G ist ein 4-Tupel aus: einer endlichen Menge von Terminalen T (entspricht dem Alphabet Σ der Sprache), einer endlichen Menge von Nichtterminalen N, einer endlichen Menge von Regel / Projektionen P und einem Startsymbol s (aus der Menge der Nichtterminalen). G = (N, T, P, s) ; $N \cap T = \varnothing$; $p_1 \rightarrow p_2$; $s \in N$ oder G = (N, Σ, P, s) ; $N \cap A = \varnothing$; $p_1 \rightarrow p_2$; $s \in N$
Eine Grammatik G ist ein System aus Variablen (einschließlich einer Start-Variable s), einem Alphabet Σ (od. den Terminale T) und einem Konstrukt von Produktions-Regeln P (od. Ersetzungs-Regeln).
Eine Grammatik ist eine Zusammenstellung von Nichtterminal-Symbolen, Terminal-Symbolen, mindestens einem Start-Symbol und eine Menge von Produktions-Regeln (,die sich auf die genannten Symbole beziehen).

Für jede Sprache vom CHOMSKY-Typ X und die dazuhörige Grammatik vom gleichen Typ gilt immer der nachfolgende Zusammenhang:



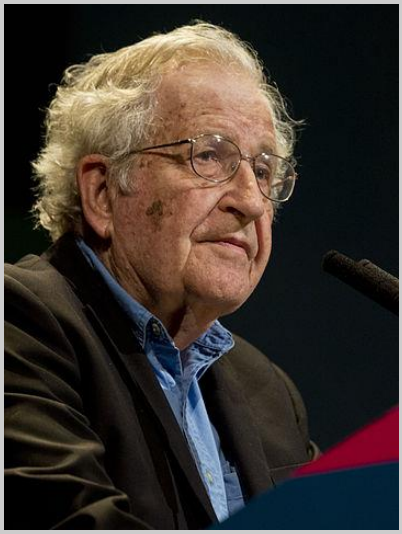
je höher der Typ, umso stärker sind die Einschränkungen durch die Grammatik
anders herum kann man sagen, dass die Sprachen / Grammatiken mit steigender Typ-Höhe immer Struktur-ärmer werden

Einige Sprachen können von Grammatiken verschiedener Typ-Ebenen erzeugt werden. In diesen Fällen wird die Sprache immer dem "höchsten" Typ zugeordnet

Biographie: Avram Noam CHOMSKY (1928 -)

Noam CHOMSKY zählt zu den weltweit bekanntesten Intellektuellen. Mit einer linken Grundposition gilt er als einer der einflußreichsten und prominensten Kritiker der amerikanischen Politik. Seine Arbeiten im Bereich der Linguistik gelten als bahnbrechend.

Geboren wurde Avram Noam CHOMSKY am 7. Dezember 1928 in Philadelphia (Pennnsylvania (USA)) als Kind jüdischer Eltern.



Q: de.wikipedia.org (→ https://www.flickr.com/photos/culturaargentina)

Ab 1945 studierte er zuerst an der University of Pennsylvania, dann an der Harvard University in Cambridge (Massachusettes (USA)).
Seine Promotion begann er 1950 wieder in Pennsylvania im Bereich Linguistik. Schon seine Dissertation

Biographie: Joseph WEIZENBAUM (1923 - 2008)

deutsch-amerikanischer Informatiker
kritischer Geist, der sich viel mit der Verantwortung der Informatik auseinandergesetzt hat
ist aber auch philosophisch aktiv gewesen und beschäftigte sich auch mit vielen aktuellen Wissenschafts- und Gesellschafts-Problemen / -Themen
bezeichnete sich gerne als Dissidenten und Ketzer der Informatik

> Computer sind wie alle Instrumente nicht wertfrei, sondern erben ihre Werte von der Gesellschaft, in der sie eingebettet sind. In einer vernünftigen

geboren als 2. Kind der Ehe von Henriette und Jechiel WEIZENBAUM in Berlin
einer seiner Brüder (Heinrich WEIZENBAUM (1923 - 2005)) war ebenfalls ein bedeutender Informatiker; bekannt unter dem Namen Henry F. SHERWOOD

Luisenstädtisches Realgymnasium Berlin
dann als Jude an die Jüdische Knabenschule verwiesen

1936 Emigration in die USA

studerte ab 1941 Mathematik an der Wayne State University (Detroit, Michigan, USA)

im II. Weltkrieg zwischenzeitlich Dienst in der metreologischen Abteilung der Air Force

1946 Fortsetzung des Studium's
1950 Master
danach Assistent im Team, dass einen Großrechner für die mathematische Fakultät seine Universität entwarf, baute und den Betrieb betreute

1952 – 1963 System-Ingenieur im Computer Development Laboratory der General Electric Corporation (GE)

1963 assoiierte Professur am MIT (Massachusetts Institute of Technology)
ab 1970 eigene Professur für Computer-Wissenschaften

war ab der Mitte der 60er Jahre am Aufbau des Arpanet (wissenschaftlicher und militärischer Vorläufer des heutigen Internet)

schrieb (1966) Programm "ELIZA"
quasi eine Version eines TURING-Test's
sehr einfaches Programm (damals in BASIC geschrieben)
simmuliert einen Gesprächspartner / Psychologen
Programm reagiert natürlich-sprachlich (in Englisch) auf die Eingaben des Nutzer's
viele Nutzer nahmen das Programm als menschlichen Therapeuten wahr, es wurden selbst intimste Detail's mitgeteilt und häufig stark emotional reagiert
das Programm zählt heute zu den Meilensteinen der

KI-Forschung

J. WEIZENBAUM; 2005
Q: de.wikipedia.org (Ulrich Hansen)

verheiratet mit Ruth
hatte vier Töchter

gestorben 2008 in Berlin

2001 Großes Bundesverdienstkreuz
verschiedene Preise von Organisationen aus dem Bereich Informatik
diverse Ehren-Doktoren-Titel

sein Buch "Die Macht der Computer und die Ohnmacht der Vernunft" (1976)

3.1.z.1. rekursiv aufzählbare / uneingeschränkte / beliebig formale Sprachen (CHOMSKY-Typ 0)

Zum Typ 0 der Grammatiken zählen alle formalen Grammatiken. Die Produktions-Regeln können beliebig mit den Terminalen und Nicht-Terminalen hantieren.

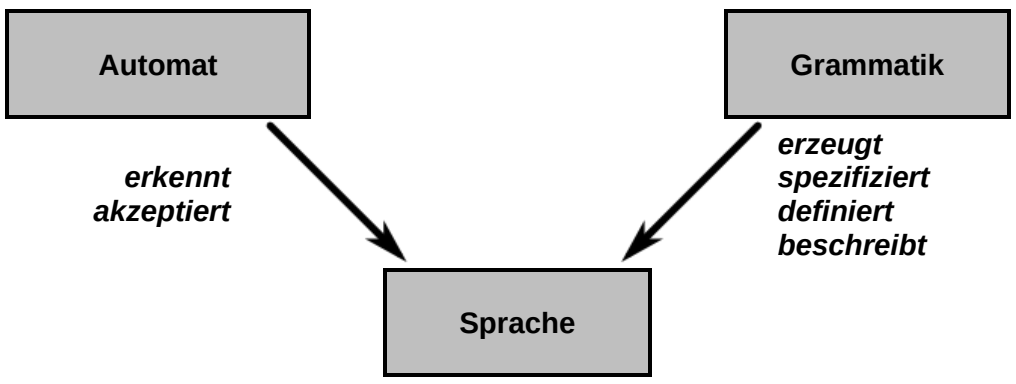
Definition(en): Typ-0-Grammatik
Eine Grammatik vom Typ 0 nach CHOMSKY ist jede mögliche Grammatik ohne Einschränkung der Regeln.

Definition(en): Typ-0-Sprache
Eine Sprache vom Typ 0 nach CHOMSKY ist eine Sprache die auf einer Typ-0-Grammatik basiert.

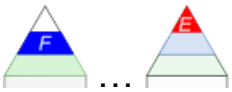
Da bei Typ-0-Sprachen keine grammatikalischen Einschränkungen bestehen, ist auch ein leeres Wort ε kein Problem.
Anders sieht das bei den nachfolgenden Sprachen von Typ-1 bis 3 aus. Deshalb gibt es hier die Sonder-Regelung, dass ε ein mögliches Wort der betrachteten Sprache ist.
Die Sonder-Regelung ist dabei so vereinbart, dass das leere Wort ε nur aus dem Startsymbol s abgeleitet werden darf und das Start-Symbol selbst darf niemals auf der rechten Seite einer Regel auftauchen.

L_0 – also Sprachen vom Typ 0 – werden auch TURING-erkennbare Sprachen genannt, da es für sie jeweils immer (mindestens) eine TURING-Maschine (einen speziellen abstrakten Erkennungs-Automaten) gibt, die genau diese Sprache erkennt. D.h. der Automat wird mit einer Zeichenfolge der Sprache gefüttert und prüft diese. Am Ende kann die TURING-Maschine dann anzeigen, ob das Wort zur Sprache gehört oder nicht.

Es gibt also auch einen Zusammenhang zwischen Sprachen, Grammatiken und Automaten.



3.1.z.2. Kontext-sensitive Sprachen (CHOMSKY-Typ 1)



Vielfach ist in einer Sprach bzw. Grammatik entscheidend, welche Zeichen, Wörter oder Platzhalter neben einem Zeichen stehen. In so einem Fall ist die Grammatik vom Kontext (– also der Umgebung –) abhängig. Wir sprechen auch von Kontext-sensitiven Sprachen. Sie werden von passenden – Kontext-sensitiven – Grammatiken (context-sensitive languages, CSL) erzeugt oder können an ihnen geprüft werden.

Nach CHOMSKY sind Typ-1-Grammatiken dazu geeignet, um Wörter einer Sprache zu generieren und sie ebenfalls mit ihr (auf Zugehörigkeit) zu testen.
Die Einschränkung gegenüber der Typ-0-Grammatiken besteht darin, dass bei allen Produktions-Regeln die linke Seite höchstens so lang sein darf, wie die rechte.

Grammatiken vom Typ 1 sind automatisch auch vom Typ 0. Für sie gibt aber als Einschränkung die Bedingung, dass für alle Produktions-Regeln $p_1 \rightarrow p_2$ gilt $|p_1| \leq |p_2|$
Zur Erinnerung sei hier noch mal kurz darauf hingewiesen, dass die senkrechten Striche (i.A. als Absolut-Zeichen bezeichnet) die Wortlänge meinen.

z:B.: **Grammatik1**

S → ab	hier rechte Seite länger als linke
S → aSb	deshalb mögliche Typ-1-Grammatik

z:B.: **Grammatik2**

S → ABABAB	
S → SAB	
BA → a	ab hier längere linke Seite
Aa → ε	deshalb **keine** Typ-1-Grammatik
aB → ε	

mögliche Nicht-Terminale stehen zwischen Terminalen
??? prüfen: das Start-Symbol kommt in keiner Regel bzw. Produktion auf der rechten Seite vor

Definition(en): Typ-1-Grammatik
Eine Grammatik vom Typ 1 nach CHOMSKY ist eine Grammatik, bei denen jede Regel ein Nicht-Terminal in einem Kontext / einer Umgebung in eine nicht-leere Folge von Nicht-Terminal und / oder Terminal-Symbolen ersetzt.

Definition(en): Typ-1-Sprache
Eine Sprache vom Typ 1 nach CHOMSKY ist eine Sprache die auf einer Typ-1-Grammatik basiert.

Betrachten wir noch einmal die Grammatik G = ({a,b}, {S}, P,S) (obiges Beispiel: Grammatik1) mit den Produktionen P:

S → ab
S → aSb

Erstellt man einen **Wort-Baum**, in dem alle Worte bis zu einer bestimmten (Wort-)Länge mit ihren Ableitungen enthalten sind, dann kann man in ihm auch gut prüfen, ob ein beliebiges Wort zu einer Grammatik gehört. D.h. im Fachjargon, dass das Wort-Problem für diese Grammatik lösbar ist.
Weil eben mehrere Worte einer Sprache hier enthalten sind, handelt es sich **nicht** um einen Ableitungs-Baum für ein Wort.
Im Wortbaum werden die möglichen Regel-Anwendungen als neue Zweige hinzugefügt. Da wir in der betrachteten Sprache nur zwei Regeln haben, ist unser Wort-Baum auch nur dichotom.
Praktisch ließe sich das Wort zu immer neuen Worten verlängernen. Da aber die Wortlänge eines einzuordnenden Wortes bekannt / ermittelbar ist, kann man ab einer bestimmten Ebene aufhören. Per Definition können Ableitungen bei Typ-1-Grammatiken niemals kürzer werden, wie es noch bei Typ-0 möglich war.

Hat man ein Wort der Sprache / Grammatik zugeordnet, dann kann man sich auch den eigentlichen Ableitungs-Baum ansehen. Er enthält immer nur das gesuchte Wort und seine Entwicklungs-Schritte. Diese sind immer durch noch enthaltene Nicht-Terminale gekennzeichnet.
Jede Ebene ist durch eine genutzte Regel hinzugekommen. Für ein einzelnes Wort muss der Baum auch immer endlich gross sein.
Im nebenstehenden **Ableitungs-Baum** ist das resultierende Wort von den Zweig-Ende zu einem Wort zusammenzusetzen. In der Abbildung fängt man dabei links oben an und arbeitet sich dann entgegen dem Uhrzeigersinn nach rechts ober durch.
Das erzeugte Wort des nebenstehenden Ableitungs-Baum's ist **aaaabbbb**. Beim Lesen muss hier links oben begonnen werden und dann im entgesetzten Uhrzeigersinn weiter vervollständigt werden.

Benötigt man für die Sprache auch das (einzelne) leere Wort ε, dann müssen die Produktions-Regeln u.U. angepasst werden, damit die ε-Sonder-Regelung eingehalten wird. Im Allgemeinen lassen sich die Regeln aber umschreiben. Wie das geht, beschreibe ich am Ende des Abschnitts zu den Typ-2-Sprachen, weil das Verfahren auch für sie zutrifft.

Beispiel:

$L = \{a^n b^n c^n;\ n \geq 1\}$

G = ({a,b,c}, {S,B,C}, {S → aSBC, S → aBC, CB → BC , aB → ab, aB → bb, bC → bc, cC → cc}, S)

Beispiel:

L = geklammerte Ausdrücke von Additionen und Multiplikationen

G = ({(,a,+,*,)}, {S,A,B},{S → A, S → S + A, A → B, A → A * B, B → a, B → (S)}, S)

lassen sich in die KURODA-Normalform überführen
in dieser Form liegen alle Regeln in der einer der folgenden Regelformen vor:

$A \rightarrow a$
$A \rightarrow B$
$A \rightarrow BC$
$AB \rightarrow CD$
…

Groß-Buchstaben stehen hier allgemein für Nicht-Terminale; Klein-Buchstaben allgemein für Terminale

für jede CHOMSKY-Typ-1-Grammatik gibt es eine Grammatik in der KURODA-Normalform

Algorithmus zum Lösen des Wort-Problems für Typ-1-Sprachen:

```
LESE G        (Grammatik-Regeln)
LESE w        (Wort-Länge)
T := {S}
WIEDERHOLE
        T' := T
        T := T'  U  {u  |  |u|  ≤  |w|   UND   u' ϵ T'}
BIS  (w IN T)  ODER  (T = T')
RÜCKGABE  WAHR
```

(Prüfen! Algorithmus gibt kein FALSCH für den Fall zurück, dass die Wortlänge überschritten wurde!)

3.1.z.3. Kontext-freie Sprachen (CHOMSKY-Typ 2)

Bei Kontext-freien Sprachen können Nicht-Terminale ohne Berücksichtigung des Kontextes (also der Zeichen-Umgebung) ersetzt werden. Üblich ist für sie die Abkürzung CFG für **c**ontext-**f**ree **g**rammar, also den Kontext-freien Grammatiken, durch die sie erzeugt werden.
Kontext-freie Sprachen sind nicht-reguär. Das heißt, sie enthalten als Untermenge zwar die regulären Sprachen, sind aber praktisch größer – also umfassender.
Die Kontext-freien Sprachen werden dann wieder von den Kontext-sensitiven Sprachen umschlossen.

Definition(en): Typ-2-Grammatik
Eine Grammatik vom Typ 2 nach CHOMSKY ist eine (formale) Grammatik, bei der nur solche Ersetzungs-Regeln enthalten sind, die aus einem Nicht-Terminal-Symbol beliebig lange Folgen von Terminal und / oder Nicht-Terminal-Symbolen ableiten.

Definition(en): Typ-2-Sprache
Eine Sprache vom Typ 2 nach CHOMSKY ist eine Sprache die auf einer Typ-2-Grammatik basiert.

Die Regeln einer Typ-2-Grammatik haben dabei immer die Form **A → X**, wobei **X** wieder aus Terminal- und Nicht-Terminal-Symbolen bestehen kann.
Da auf der linken Seite der Regel(n) nur **A** als Nicht-Terminal-Symbol vorkommt, hängt alles davon ab, dass genau dieses Nicht-Terminal-Symbol in der produzierten Zeichenfolge (hier: **X**) vorkommt. Da das nicht vom Kontext abhängig ist, spricht man von einer Kontext-freien Grammatik. Mit andeen Worten, es handelt sich um ein Kontext-freies Regelwerk.Solche Regeln könnten z.B sein: **X → x** oder **X → YY** , …
Mit anderen Worten gilt, dass eine Nicht-Terminal-Symbol auf der linken Seite einer Regel niemals von Symbolen umgeben ist – also keinen Kontext hat.
Jede Regel besteht zudem auf der linken Seite nur aus einem Nicht-Terminal.

Für Kontext-freie Sprachen gibt es keinen allgemeinen erkennenden Endlichen Automaten (→ 3.2.3. endliche Automaten). Die Kontext-freien Sprachen benötigen eine nicht-deterministischen Keller-Automaten (→ 3.2.8. nicht-deterministische Keller-Automaten)zur Erkennung.
Das Problem liegt darin, dass (einfache) endliche Automaten (gemeint sind Akzeptoren → 3.2.6. Automaten ohne Ausgabe) zwar die reinen Regeln und Symboliken prüfen kann, aber z.B. nicht die Anzahl öffnender und abschließender Klammern kontrollieren kann. Und neben der Anzahl muss auch die Ineinander-Schachtelung stimmen, egal was zwischen ihnen steht.
Es existieren auch keine rechts- oder linkslinearen Grammatiken für Sprachen vom CHOMSKY-Typ 2.

Zu den Kontext-freien Sprachen gehören z.B. Palindrome oder Klammer-Systeme

Grammatik einer Palindrom-Sprache (liefert alle Palindrome zum Alphabet {0,1})

G = ({S}, {0,1}, {S → ε, S → 0, S → 1, S → 0S0, S → 1S1}, S)

diese Grammatik ist eindeutig, da es für jedes Palindrom nur eine mögliche Ableitung gibt

gibt es mehrere mögliche Ableitungen eines Wortes über die Grammatik(-Regeln), dann sprechen wir von einer mehr-deutigen Grammatik

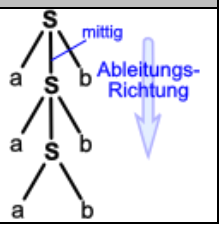
eine mehr-deutige Grammatik ist z.B.:

G = ({S, X}, {0,1}, {S → X, X → ε, X → XX, X → 1X0}, S)

betrachten wir zum Nachweis der Mehrdeutigkeit nur das Beispiel-Wort ***10***, dann gibt es z.B. die folgenden Ableitungs-Möglichkeiten:

I) S(X(1,X(ε),0))
II) S(X(X(ε),X(1, X(ε),0)))
III) S(X(X(1, X(ε),0), X(ε)))
…

Beispiel: nicht-reguläre Sprache L = { $a^n b^n$; n ≥ 0 }

Grammatik	**Ableitung von: aaabbb**	
G = ({a, b}, {S}, S, P) T N P = { S → ab, P_1 S → aSb } P_2	P_2 **S → aSb** P_2 **aSb → aaSbb** P_1 **aaSbb → aaabbb**	

Aufgaben:

1. *Erstellen Sie die Grammatik für das Klammer-System der geschweiften Klammern! Leiten Sie das Wort: {{{{}}}} aus der zugehörigen Sprache $L_{\{\}}$ über die Regeln und als Ableitungs-Schema ab?*
2.

Benötigt man für die Sprache auch das leere Wort ε, dann müssen die Produktions-Regeln u.U. angepasst werden, damit die ε-Sonder-Regelung eingehalten wird. Im Allgemeinen lassen sich die Regeln aber umschrieben. Das folgende Verfahren gilt genauso für Typ-1-Sprachen.

1. **falls** s auf der rechten Seite einer Regel vorkommt **dann** ersetze s durch ε **und** ergänze die geänderten Regel als zusätzliche Regel s'
2. s' wird das neue Start-Symbol **und** es wird eine Regel s' → s hinzugefügt
3. **falls** die Regel s → ε vorhanden ist **dann** wird diese in s' → ε umbenannt **oder** (neu) hinzugefügt

nur für Sprachen vom Typ 2 gilt zusätzlich:

4. falls Regeln der Form N → ε vorhanden sind, dann müssen diese ensprechend den Schritten 1. bis 3. für N statt s angewendet werden

Beispiel1:

s → ε | 0s1 | asb wird: s' → ε | s
s → 0s1 | asb | 01 | ab

aus /d, S. 17/

Beispiel2:

Gleitkommazahlen: **Σ** = {+;-;0;1;2;3;4;5;6;7;8;9;,;e;E} (hier Semikolon als Trenner, da Komma für die Zeichenkette gebraucht wird!)

Z = (0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9)
(+ | - | e) (ZZ* | ZZ*,Z* | Z*,ZZ*)

(! noch nicht vollständig)!

G_{GKZ} = (**Σ**, {}, {},)

Wir sehen schon, dass die Grammatiken in der Tupel-Definition einer Grammatik recht unübersichtlich werden. Die Informatiker John W. BACKUS und Peter NAUR beschäftigten sich u.a. mit der Formulierung von Grammatiken und entwickelten eine vereinheitlichte, recht übersichtliche Darstellungs-Form. Sie wird allgemein BACKUS-NAUR-Form oder kurz BNF genannt. Auf der linken Seite wird der abzuleitende Ausdruck notiert. Die rechte Seite stellt die Regel-Details dar. Beide Seiten sind durch das Regel-Zeichen **::=** getrennt. Terminale werden als einfache Zeichen notiert. Zeichen-Gruppen sollten ev. durch umschließende Hoch-Kommata oder Anführungs-Zeichen begrenzt werden. Nicht-Terminale schreibt man in eckigen Klammern: **< >**.
Eine Folge von Nicht-Terminal- und / oder Terminal-Zeichen auf der rechten Seite der Regel wird durch direktes Hintereinanderschreiben definiert. Zur besseren Lesbarkeit wird manchmal noch ein Leerzeichen benutzt.

```
<Nicht-Terminal> ::= Terminal                                  einfache Regel
<Nicht-Terminal> ::= <Nicht-Terminal>                          einfache Regel
<Nicht-Terminal> ::= <Nicht-Terminal> <Nicht-Terminal>         (direkte) Folge
<Nicht-Terminal> ::= Terminal <Nicht-Terminal>                 (direkte) Folge
<Nicht-Terminal> ::= <Nicht-Terminal> Terminal                 (direkte) Folge
```

BNF erfüllt genau die Forderung an eine kontext-freie Grammatik dahingehend, dass auf der linken Seite immer nur genau ein Nicht-Terminal steht.

häufig wird hierfür die kompaktere Erweiterte BACKUS-NAUR-Fom (EBNF) notiert. Sie verfügt über weitere Meta-Zeichen, die nur Darstellung von Wiederholungen dienen. Sie stehen praktisch für die Formulierung aller geeigneten Grammatiken zur Verfügung.

Bestandteil der EBNF sind noch ein Meta-Zeichen, das der kompakteren / zusammengefassten Darstellung von mehreren Regeln dient. Mit dem senkrechten Strich **|** wird eine Alternative charakterisiert.

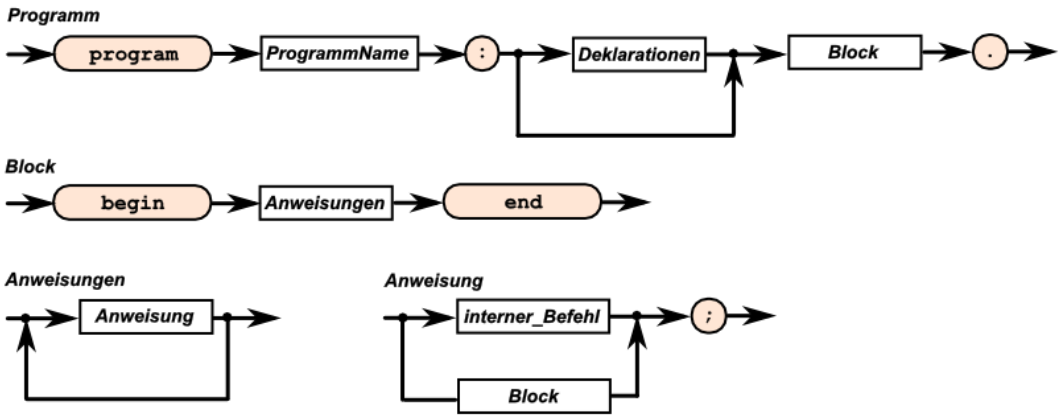
```
<Nicht-Terminal> ::= Terminal                                  einfache Regel
<Nicht-Terminal> ::= <Nicht-Terminal>                          einfache Regel
<Nicht-Terminal> ::= <Nicht-Terminal>   |   Terminal          Alternative
<Nicht-Terminal> ::= <Nicht-Terminal> <Nicht-Terminal>         (direkte) Folge
```

zu den Typ-2-Sprachen gehören auch die meisten Programmier-Sprachen; typisch PASCAL und MODULA

Beispiel: Ausschnitt aus der Grammatik einer Programmiersprache (Variable)

```
<einfacheVariable>    ::= <VariablenName>
<VariablenName>       ::= <Variable>
<Variable>            ::= <Buchstabe> | <Variable><Buchstabe> |
                          <Variable><Ziffer>
<Buchstabe>           ::= <Großbuchstabe> | <Kleinbuchstabe>
<Großbuchstabe>       ::= A | B | C | D | E | F | G | H | I | J | K | L |
                          M | N | O | P | Q | R | S | T | U | V | W | X |
                          Y | Z
<Kleinbuchstabe>      ::= a | b | c | d | e | f | g | h | i | j | k | l |
                          m | n | o | p | q | r | s | t | u | v | w | x |
                          y | z
<natürlicheZahlen>    ::= <Null> | <NichtnullZiffer> |
                          <NichtnullZiffer> <Ziffer>
<NichtnullZiffer>     ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<Ziffer>              ::= <Null> | <NichtnullZiffer> | <Ziffer>
<Null>                ::= 0
```

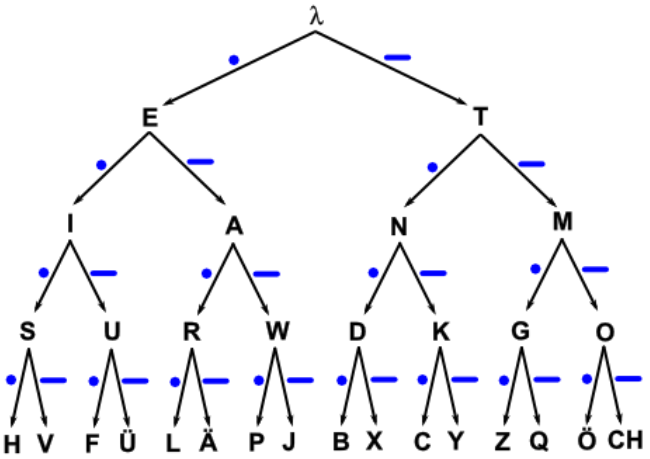
Alternativ finden sich auch für viele Programmier-Sprachen Syntax-Diagramme, die den gleichen Sachverhalt mehr graphisch orientiert anzeigen:



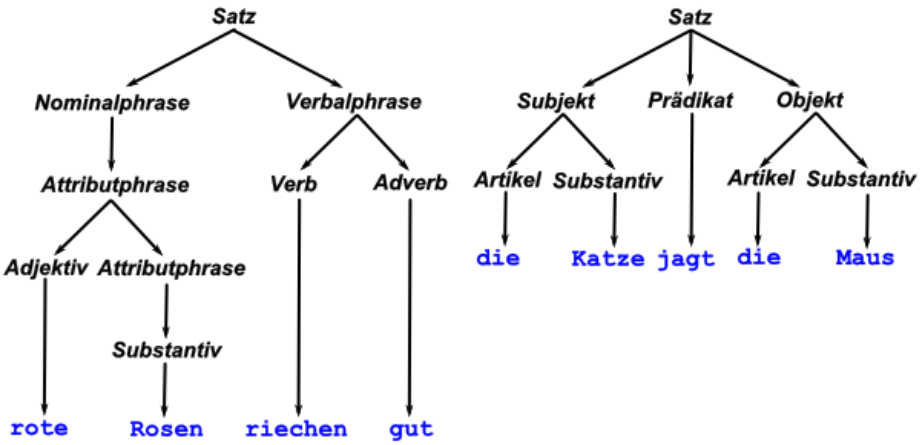
Ausschnitt aus einem (möglichen) Syntax-Diagramm
zur Programmier-Sprache PASCAL

nachteilig ist der große Platz-Bedarf

weitere Möglichkeit zur Darstellung und Analyse der Sprache / Grammatik ist die Angabe eines **Ableitungsbaum**s. Das funktioniert besonders bei geschlossenen / endlichen Sprachen



Das Erzeugen eines Ableitungs-Baums bzw. das Testen, ob ein Ausdruck zur Sprache gehört, nennt man parsen.
Programme, die Texte in reguläre (erkennbare) Ausdrücke auflösen / zerlegen, heißen Parser. Dazu gleich mehr (→ 3.1.z.4. reguläre Sprachen (CHOMSKY-Typ 3)).
Wörter (Sätze) regulärer Sprachen können durch Automaten zerlegt werden und dann letztendlich auch akzeptiert werden.



arithmetrische Ausdrücke (Grundsystem)

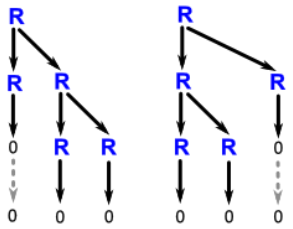
```
<Ausdruck>          ::= <Term>
<Ausdruck>          ::= ( <Ausdruck> )
<Ausdruck>          ::= <Ausdruck> <Operator> <Term>
<Term>              ::= <Operant>
<Term>              ::= <Term> <Operator> <Operant>
<Operant>           ::= <Zahl> | <Variable>
<Operator>          ::= + | - | * | /
<Zahl>              ::= → Def. "Zahl"
<Variable>          ::= → Def. "Variable"
```

man spricht von einer Links-Ableitung, wenn bei jedem Ableitungs-Schritt / jeder Regel-Anwendung immer bzw. zuerst das am weistesten links stehende Nicht-Terminal – bei Vorhandensein einer Regel – ersetzt wird.

Beispiel:
Grammatik G = ({R},{0},R,{R → 0, R → RR}), abzuleitendes Wort w = 000

mögliche Ableitungen: R ⇨ RR ⇨ 0R ⇨ 0RR ⇨ 00R ⇨ 000
R ⇨ RR ⇨ RRR ⇨ 0RR ⇨ 00R ⇨ 000

mögliche Ableitungsbäume:



(→ diese Grammatik ist also mehrdeutig)

Aufgaben:

1. *Überlegen Sie sich, wie Rechts-Ableitungen funktionieren! Stellen Sie Ihre Position dem Kurs vor!*
2. *Leiten Sie für obige Sprache (Null-Sprache) die folgenden Worte ab! Geben Sie, wenn es möglich ist, mindestens zwei Ableitungen (zur Kennzeichnung der Mehrdeutigkeit) an!*

   a) 00 b) 0000 c) 000000
3. *Gegeben ist die Grammatik G = ({a,b}, {S}, {S → a, S → b, S → SS}, S). Geben Sie für die folgenden Worte jeweils eine Links-Ableitung an!*

   a) aaa b) abaaa c) aaabaa
4. *Gegeben ist die Grammatik G = ({8,0,3}, {S}, {S → 8, S → 0, S → 3, S → SS}, S). Geben Sie für die folgenden Worte jeweils eine Links- und eine Rechts-Ableitung an, wenn dies möglich ist!*

   a) 8 b) 33 c) 030
   d) 8888 e) 30883 f) 803030308
5. *

Nur als Hinweis für weiterführende Studien sei hier erwähnt:
Die Kontext-freien Sprachen sind gegenüber den regulären Sprachen hinsichtlich der Operationen (Vereinigung, Spiegelung, Verkettung, Homomorphismen-Anwendung und KLEENE-schen Hüllenbildung (Interation)) abgeschlossen. Sie sind nicht abgeschlossen hinsichtlich Durchschnitten, dem Komplement (Negation), logarithmischer Platz-beschränkter Reduktion und der symmetrischen Differenz.

weitere Beispiele für Kontext-freie Grammatiken:

```
<deutschePostadresse>   ::=  <Adressat> <NZ> <Strasse> <NZ> <Ort> <NZ>
<Adressat>              ::=  <PersonenAdressat> | <InstitutionsAdressat>
<PersonenAdressat>      ::=  <Anrede> <Titel> <Name> | <Anrede> <Name>
<InstitutionsAdressat>  ::=  <InstitutionsName> <NL> <PersonenAdressat> |
                             <InstitutionsName>
<Titel>                 ::=  Prof. | Dr. | Prof. Dr. | Dipl.-Ing. | RA |
                             Dipl. Ing. Dr. → weitere
<Name>                  ::=  <VorName> <NachName> | <NachName>
<InstitutionsName>      ::=  → Text
<NachName>              ::=  → Text
<VorName>               ::=  → Text
<Strasse>               ::=  <StrassenName> | <StrassenName> <StrNummer> |
                             <StrNummer>
<StrassenName>          ::=  → Text
<StrNummer>             ::=  → Zahl <Buchstabe> | → Text
<Buchstabe>             ::=  → Buchstaben
```

Aufgaben:

1. *Erweitern Sie die Grammatik der deutschen Postadresse um die Definitionen für die angedeuteten Nicht-Terminale Zahl und Buchstabe!*
2. *Prüfen Sie Ihre eigene Adresse anhand der Grammatik auf Akzeptanz!*
3. *Erstellen Sie eine Grammatik, die eMail-Adressen erkennt / erzeugt!*
4.

für die gehobene Anspruchsebene:

x. *Erweitern Sie die Grammatik der deutschen Postadresse um die Definitionen für das angedeuteten Nicht-Terminale Text! (Es werden hier nur Einzelworte erwartet!)*

x. *Definieren Sie eine Grammatik zu einer Geheimsprache, die aus dem MORSE-Alphabet abgeleitet ist und bei der Folgen von Punkten durch Buchstaben (entsprechend ihrer Position im Alphabet) und die Striche als Ziffer codiert werden! Beispiel: V → – • – – → 1a2 od. V → • • • – → c1*

komplexe und übergreifende Aufgaben:

1.

x. Erstellen Sie für jeden der folgenden Ausdrücke den Ableitungsbaum! Es gilt die Grammatik G = ({•,O}, {S}, {S → •, S → O, S → SS}, S).

a) OOO b) O•• c) O•O••••O

x. Prüfen Sie, ob das Parsen der folgenden Texte als "arithmetrischen Ausdruck" positiv verlaufen würde! Begründen Sie immer Ihr Prüf-Ergebnis!

a)	a + b	b)	4 – 9	c)	3 * a + (c – b)
d)	3 a : 4	e)	x / y	f)	a (c + a)
g)	(a + b) / (3 * x – 4)	h)	(a + b) / 3 * x – 4	i)	x^2 + 7
j)	7 + 3 !	k)	a + - b	l)	- 4 + 6 * 3 5

x. Überlegen Sie sich, welche Grammatik eine Sprache definiert, die eine Sprache aus beliebig vielen v , ^und – besteht!

x.

x. Erweitern Sie die BNF für die arithmetischen Ausdrücke um ein vollständiges Klammer-System auch mit eckigen und geschweiften Klammern!

x. Geben Sie die Typen der nachfolgenden Grammatik-Regeln an!

a)
S → 01
S → 0S1

b)
R →][
R →]R[

c)
S → ( ) | [ ] | { }
S → (S) | [S] | { S }

d)
R → x
R → +
R → #
R → RR

e)
S → A | B | ε
A → CD | C | D
B → EF | E | F
C → 0C1 | 01
D → 2D | 2
E → 0E | 0
F → 1F2 | 12

f)

x. In die arithmetrischen Ausdücke sollen die beiden Konstanten (Kreiszahl) p und (EULERsche Zahl) e– auch als Objekt "Konstante" – in die BNF eingebaut werden. Machen Sie einen Vorschlag zur Erweiterung der BNF! Die Konstanten erhalten die speziellen Zeichen "PI" und "E".

x. Prüfen Sie, ob die nachfolgend angegebene Sprach-Darstellung (Ausschnitt) in der BACKUS-NAUR-Form vorliegt! Ist sie exakt? Kann sie vereinfacht werden?

x. Zeichnen Sie für die nachfolgende BNF ein Syntax-Diagramm!

für die gehobene Anspruchsebene:

x. Erweitern Sie die obige BNF ("arithmetrische Ausdrücke") um die Möglichkeit solche Ausdrücke wie: 4 x , ½ a und x b (statt z.B. x·b) mit zu verarbeiten!

3.1.z.4. reguläre Sprachen (CHOMSKY-Typ 3)

auch erkennbare Sprachen genannt
oder als reguläre Mengen bzw. Ausdrücke bezeichnet

Für die Produktions-Regeln der regulären Sprachen (also Sprachen vom Typ 3) gilt, dass sie mindestens vom Typ 2 sein müssen und zusätzlich immer gilt, dass auf der rechten Seite der Regeln immer Terminal(e) und Variablen in gleicher Reihenfolge stehen.

Die Regeln enstammen dem folgenden Grund-Schemata:

A → xB
A → x
A → ε *Standard-mäßig nicht enthalten, bei vielen Systemen aber aus praktischen Gründen integriert*

Alle Regeln sind einseitig linear.
Somit unterscheidet man je nach Aufbau-Prinzip der Grammatik rechts- bzw. links-reguläre Grammatiken.
(besser: rechts- / links-linear)

rechts-lineare Grammatik für ganze Zahlen
Nicht-Terminal-Zeichen stehen immer nur rechts

```
S --> + Z
S --> - Z
S --> 0..9
S --> 1..9 R
R --> 0..9
R --> 0..9 R
Z --> 1..9
Z --> 1..9 R
```

rechts-lineare Wörter entstehen somit von links nach rechts

spricht man allgemein nur von einer regulärer Grammatik, dann ist zumeist eine rechts-lineare (rechts-reguläre) gemeint

Für die Sprache L = { aba^n ; n ∈ ℕ} kann z.B. die rechts-lineare Grammatik definiert werden.

```
S --> a b A
A --> a A
A --> a
```

Eine äquivalente links-lineare Grammatik hat dementsprechend nur Regeln, bei denen eventuell vorkommende Nicht-Terminale in den Regeln immer links (also vor einem / den Terminal(en)) stehen.

```
S --> S a
S --> A b
A --> a
```

links-lineare Grammatik für ganze Zahlen
hier stehen die Nicht-Terminal-Symbole immer nur links (gefolgt von Terminalen oder Nicht-Terminal-Symbolen

```
???
```

aus einer links-linearen Grammatik produzierten Wörter werden also immer von rechts nach links erstellt

auch einseitig lineare Grammatiken genannt

Definition(en): Typ-3-Grammatik
Eine Grammatik vom Typ 3 nach CHOMSKY ist eine Grammatik vom Typ 2, wobei die rechte seite der Regeln so aufgebaut sein müssen, dass Terminale und Nicht-Terminale immer in der .gleichen Reihenfolge stehen.

Definition(en): Typ-3-Sprache
Eine reguläre Sprache / eine Sprache vom Typ 3 nach CHOMSKY ist eine Sprache die auf einer Typ-3-Grammatik basiert.
Eine Sprache vom Typ 3 nach CHOMSKY ist eine Sprache die von einem endlichen Automaten (egal ob deterministisch od. nicht-deterministisch) akzeptiert wird.

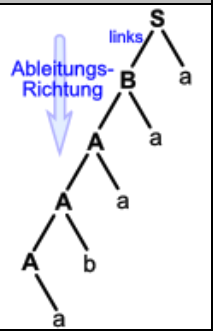
Wird eine Sprache von einer rechts- oder links-linearen Grammatik erzeugt, dann ist es eine reguläre Sprache (Sprache vom CHOMSKY-Typ 3).

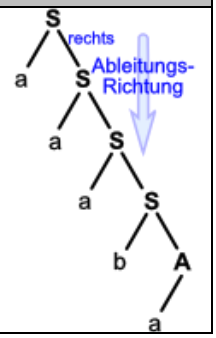
Benötigt man für eine Typ-3-Sprache auch das leere Wort ε, dann müssen die Produktions-Regeln u.U. angepasst werden, damit die ε-Sonder-Regelung eingehalten wird. Im Allgemeinen lassen sich die Regeln aber umschrieben.

1. **falls** s auf der rechten Seite einer Regel vorkommt **dann** ersetze s durch ε **und** ergänze die geänderten Regel als zusätzliche Regel s'
2. jedes s auf der rechten Seite einer Regel wird durch s' ersetzt
3. alle Regeln der Form s → … werden kopiert und das s auf der linken Seite durch s' ersetzt
4. hinzufügen von s → ε **UND falls** vorhanden wird s' → ε entfernt
5. falls Regeln der Form N → e vorhanden sind, dann müssen diese ensprechend den Schritten 1. bis 4. für N statt s angewendet werden

Sprachen vom Typ 3 nach CHOMSKY sind abgeschlossen gegenüber Vereinigung. Komplement und Durchschnitt.

Beispiel: reguläre Sprache L = { $a^n ba$; n ≥ 0 }

links-lineare Grammatik	Ableitung von: aaaba	
G = ({a, b}, {S, A, B}, S, P) T N P = { S → Ba, P_1 A → a, P_2 A → Aa, P_3 B → b, P_4 B → Ab } P_5	P_1 **S → Ba** P_5 **Ba → Aba** P_3 **Aba → Aaba** P_3 **Aaba → Aaaba** P_2 **Aaaba → aaaba**	

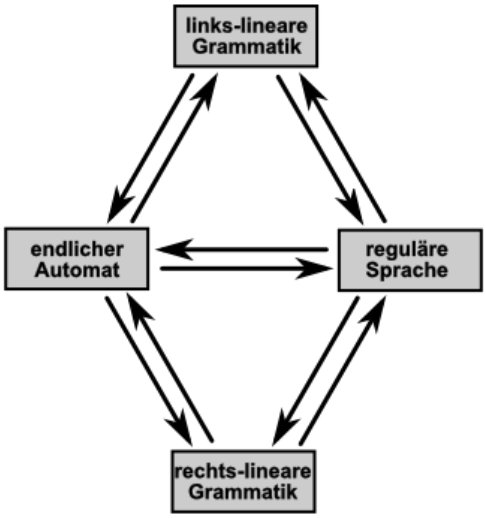
rechts-lineare Grammatik	Ableitung von: aaaba	
G = ({a, b}, {S, A}, S, P) T N P = { S → aS, P_1 S → bA, P_2 A → a } P_3	P_1 **S → aS** P_1 **aS → aaS** P_1 **aaS → aaaS** P_2 **aaaS → aaabA** P_3 **aaabA → aaaba**	

Für die regulären Sprachen gibt es einige Entscheidungs-Probleme, die Inhalt der Theoretischen Informatik sind:

Entscheidungs-Probleme (regulärer Sprachen)

- **Wort-Problem** — gehört ein Wort w zur Sprache L ? ; w ∈ Σ*
- **Leerheits-Problem** — ist die Sprache L eine leee Menge?
- **Endlichkeits-Problem** — besteht die Sprache L aus einer endlichen Menge an Wörtern?
- **Äquivalenz-Problem** — sind die Sprachen L_1 und L_2 gleich
- **Inklusions-Problem** — ist die Sprache L_2 ind der Sprache L_1 enthalten? umschließt die Sprache L_1 die Sprache L_2 ?

Alle diese Probleme sind für reguläre Sprachen entscheidbar.

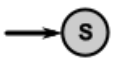
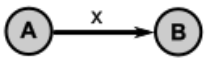
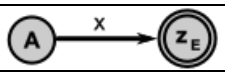


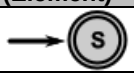
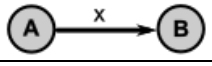
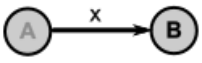
Somit besteht eine Äquivalenz zwischen den regulären Sprachen auf der einen Seite und endlichen Automaten auf der anderen. Beide sind wiederum über links- oder rechts-lineare Grammatiken – oder allgemein einer regulären Grammatik – miteinander verbunden. Alle sind somit äquivalent zueinander.

Bedingt durch die übergreifende Äquivalenz lassen sich die Elemente von Sprache, Grammatik und Automat meist ganz logisch (formal) ineinander überführen. Das nutzt man zu gegenseitigen Konstruktion.

rechts-lineare Grammatik	endlicher Automat	Bemerkungen Hinweise
T	**X**	X = T
N	**Z**	
S	**z_0**	
P	**f**	

Verwendet man als Automaten-Modell das der MOORE-Automaten, dann ergeben sich die folgenden Übertragungs-Möglichkeiten.
MOORE-Automaten sind Automaten mit Ausgabe (Transduktoren), bei denen eine Eingabe den nächsten (erreichbaren) Zustand bestimmt. Eine Ausgabe von Symbolen erfolgt immer beim Erreichen des Zustand's. Genaueres dann später unter → 3.2.5.2. MOORE-Automaten.

rechts-lineare Grammatik Regel / Element	**Zustands-Diagramm (Element)**	
S		
A → xB		
A → x		

links-lineare Grammatik Regel / Element	**Zustands-Diagramm (Element)**	
S		
B → Ax		
B → x		

In der Praxis sind aber Transduktoren – also Automaten mit Ausgabe etwas überdimmensioniert für die Prüfungs-Aufgaben zu einer Grammatik. Meist will man nur wissen, ob eine Ausdruck ein gültiger Ausdruck ist, oder eben nicht. Es zählt also nur das End-Ergebnis der Prüfung. Hierfür sind eigentlich Akzeptoren die bessere Wahl. Sie haben keine Ausgabe. Das End-Ergebnis wird durch das Erreichen eines definierten End-Zustandes des Automaten signalisiert.

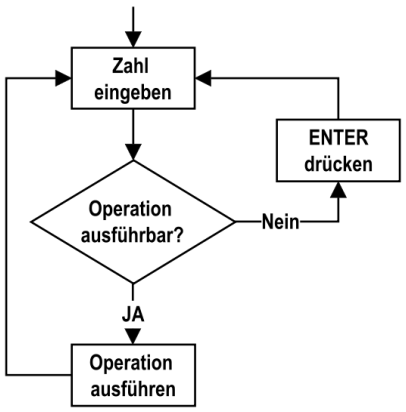
rechts-lineare Grammatik	endlicher Automat	Bemerkungen Hinweise
T	**X**	X = T
N	**$Z \setminus \{z_0\}$**	
S	**z_E ; $\lvert z_E \rvert = 1$**	
P	**f**	

Der reguläre Ausdruck (0 + 10*10*1)* über dem Alphabet {0,1} beschreibt alle Wörter, die eine durch 3 teilbare Anzahl von Einsen enthält.

3.1.z. Grammatik-Beispiel: umgekehrte polnische Notation

Wir alle kennen aus der Mathematik die algebraische Notation von Operationen / Termen. Sie sind allgegenwärtig und auf den ersten Blick scheinen sie auch die beste Möglichkeit für die Notation von Berechnungen zu sein. Das stimmt aber nicht! Und dann sind da noch die Klammern, wie schnell vergißt man sie oder es fehlt eine der beiden oder sie sind einfach ander falschen Stelle. Geht es vielleicht auch ohne Klammern?
Nehmen wir uns ein einfaches Beispiel vor:

Wollen wir **(4 + 3) * 5** berechnen, dann ist die Klammersetzung zwingend notwendig, da 4 + 3 * 5 aufgrund der höheren Priorität der Punkt-Operatoren ein gänzlich anderes Ergebnis ergeben würde. Der Nachteil dieser mathematischen Notation (Infix-Notation) liegt in einem hohen Prüfungs-Aufwand für die Rechen-Maschine. Was muss zuerst gerechnet werden? Werden die Klammer-regeln eingehalten?
Die umgekehrte polnische Notation (UPN) ist eine Postfix-Notation von mathematischen Ausdrücken. Hier werden immer zuerst die Operanden angegeben und dann nachfolgend der Operator.
Aus **(4 + 3) * 5** wird nun **4 3 + 5 *** - ein vollkommen Klammer-freier Ausdruck. Auch für Maschinen ist die Abarbeitung von links nach recht sehr einfach zu realisieren. Das Fluss-Diagramm ist deshalb auch extrem einfach.
Nachteilig ist hier aber die schlechte Lesbarkeit für uns Menschen, weshalb sich die Infix-Notation praktisch vollständig durchgesetzt hat. Es existieren nur noch wenige Taschen-Rechner, die das Prinzip der UPN (auch reverse polnische Notation genannt) benutzen.
Entwickelt wurd die umgekehrte polnische Notation vom Polen (wenn überrascht's) Jan LUKASIEWICZ (1920).



Fluss-Diagramm eines UPN-Systems
Q: de.wikipedia.org ()

Aufgaben:

1. *Erstellen Sie ein äquivalentes Fluss-Diagramm für ein Infix-System!*
2. *Übertragen Sie die folgenden Ausdrücke in die umgekehrte polnische Notation!*

a) 5 * (6 – 8) b) (4 – 2) / (3 + 7) c)
d) e) $(((4 + 5) * (2 + 3) + 6) / (8 + 7))^9$

3. *Machen Sie aus den folgenden UPN-Ausdücken die üblichen Infix-Ausdrücke!*

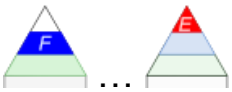
a) 7 4 – 4 * b) 3 * 4 5 + 7 * c) 6 7 – 6 * *
d) e) 4 5 + 2 3 + * 6 + 8 7 + / 9 y^x

	"normale" Notation Infix-Notation	umgekehrte polnische Notation Postfix-Notation
Tasten auf Taschenrechner	• Gleichheits-Zeichen • Klammern (außer bei sehr einfachen Modellen mit begrenzten Rechenmöglichkeiten) • ev. verzögerte Anzeige von Zwischen-Ergebnissen, wegen der Beachtung der Operator-Prioritäten	• keine Klammern und kein Gleichheits-Zeichen • dafür "Enter"-Taste zum Trennen von Operanden und Operatoren • nach Eingabe eines Operators erfolgt sofort Berechnung des aktuellen (Zwischen-)Ergebnisses
Beispiel-Aufgabe	$\frac{(4+3)\cdot 5+1}{(6-2)} = 9$	
Tasten-Sequenz für passenden Taschen-Rechner	[(] [4] [+] [3] [)] [*] [5] [+] [1] [/] [(] [6] [-] [2] [)] [=]	[4] [Enter] [3] [+] [5] [*] [1] [+] [6] [Enter] [2] [-] [/]
Statistik	• 16 Tastendrücke •	• 13 Tastendrücke •
Vorteile	• übersichtlich • für "Normalo's" klar strukturiert •	• kommt völlig ohne Klammern aus • gut programmierbar (Anlegen von Makro's) • Rechensystem rund 10x schneller • entspricht eher dem wirklichem Rechnenweg •
Nachteile	• mehr Tasten-Drücke für Klammern notwendig • Rechensystem relativ langsam	• unübersichtlich • für "Normalo's" schwerer zu erfassen
Realisierung in Programmier-sprachen	praktisch alle anderen Programmiersprachen	FORTH Reverse Polish LISP
Realisierung in Geräten	heute übliche Taschenrechner	diverse wissenschaftliche und finanz-ökonomische Taschenrechner zwischen 1970 und 2005 (bis heute Modelle verfügbar)
Realisierung / Emulation auf Computern	"Taschenrechner" unter Windows	"Yacal" in Verbindung mit der Programmiersprache YaBASIC

Aufgaben:

1. *Überlegen Sie sich im Team, ob es auch eine Prefix-Notation geben könnte (ohne weitere Literatur und Internet!)! Stellen Sie die Team-Arbeit als Schüler-Vortrag vor!*
2.

Veranschaulichung der verschiedenen Rechentypen an einem Koch-Rezept



Q: http://www.linux-community.de/ausgaben/LinuxUser/2004/03/Wissenschaftliche-Taschenrechner-mit-umgekehrt-polnischer-Notation/

Herstellen von Butter-Creme

um eine qualitativ hochwertige Creme zu erhalten sind neben der Einhaltung der Zutaten-Mengen auch die Reihenfolge der Herstellungs-Schritte wichtig.
Die Frage, die wir uns hier stellen ist, wie "programmieren" wir einen Küchen-Roboter, damit er eine optimale Butter-Creme produziert. Dabei sollte er die einzelnen Komponenten in der Arbeits-Reihenfolge aus dem Schrank holen und mit möglichst wenigen Wegstell- und Wechsel-Operationen auskommen.

algorithmische Notation

Wir nehmen 250g Butter und fügen hinzu (475ml Milch, welcher zugegeben und aufgekocht wurde (25ml Milch verrührt mit (1 Tüte Puddingpulver vermischt mit 6 EL Zucker))).

Q: s. oben; leicht geändert: dre

umgekehrte polnische Notation

Wir nehmen 250 Gramm Butter, 425 ml Milch, 25 ml Milch, eine Tüte Puddingpulver und 6 Esslöffel Zucker. Den Zucker mit dem Puddingpulver mischen; mit der kleinen Milchmenge verrühren; dieses in die grosse Milch geben und aufkochen; diesen Pudding zur Butter hinzufügen.

Q: s. oben; leicht geändert: dre

Beim HP besteht der Stapel aus 4 Speicher-Zellen – auch Register genannt. Sie heißen X, Y, Z und T. Die Eingaben landen im Register X. Durch die Taste [Enter] wird die Eingabe eingekellert in das Register Y.

logische Küchen-Notation

Das Pudding-Pulver wird mit 6 Eßlöffeln Zucker gemischt (z.B. in Misch-Becher). Dann nehme man 500 ml Milch und nehme davon ungefähr 25 ml (in den Misch-Becher) ab.
Pulver und Milch werden nun gut durchmischt. Die große Milchmenge wird erhitzt und dann das angerührte Pudding-Pulver dazugegeben. Die Mischung wird weiter erhitzt bis sie dickflüssig wird.
Die Butter wird zerteilt in eine Schüssel gegeben.
Nach einem leichten Abkühlen der Pudding-Masse gibt man diese vorsichtig und in kleinen Portionen zur Butter. Die Creme muss immer gut umgerührt werden.

"Roboter"-Algorithmus:
Misch-Becher nehmen
wiederhole 6x
    Eßlöffel Zucker einfüllen
Pudding-Pulver einfüllen
mischen
wenig Milch einfüllen
mischen
Milch in Topf füllen
Topf erwärmen
Misch-Becher in Topf einfüllen
Topf erwärmen
Butter in Schale füllen
Butter zerkleinern
Topf-Inhalt in kleinen Portionen in Schale einfüllen
rühren
abkühlen lassen

3.1.z. Grammatik-Beispiel: einfache Turtle-Graphik (LOGO)

einfache Turtle-Graphik (LOGO):
kleine Auswahl an Anweisungen und Meta-Strukturen (weitere unter: http://www.calormen.com/jslogo/#)

fd … vorwärts, es folgt Anzahl Schritte
bk … rückwärts, es folgt Anzahl Schritte
rt … nach rechts drehen, es folgt die Gradzahl
lt … nach links drehen, es folgt die Gradzahl
st … zeige Turtle (zeichnen!)
ht … verstecke Turtle (nicht zeichnen!)

Meta-Strukturen:
repeat … […] … wiederhole Anzahl [Schleifeninhalt]
if … … … […] … Verzweigung Bedingung (Op1 Zeichen Op2) und Klammern der Wahr-Teil
clearscreen … Löschen des Bildschirms
Variablennamen beginnen mit Doppelpunkt
Funktion beginnen mit TO und enden mit END ; hinter TO folgt der Name und dann die Liste der übergebenen Variablen
Rückgabe mit Funktions-Name gefolgt vom Wert oder einer Variable

zu Testen z.B. mit dem LOGO-Interpreter www.calormen.com/jslogo

Aufgaben:

1. ***Geben Sie die Anweisungen für das Zeichnen eines Quadrates mit einer Kanten-Länge von 20 Schritten an!***
2. ***Entwickeln Sie einen Algorithmus, der das Haus vom Nikolaus mit einer Haus-Breite von 100 Schritten zeichnet! Verwenden Sie solche Anweisungen, die später auch mit anderen Schrittlänge funktionieren können!***
3.

spezielle Realisierungen für Turtle-Grafiken sind möglich mithilfe von OpenOffice als DRAW-Ausgabe-Objekt oder mit GeoGebra
Die Lern-Umgebungen KARA nutzen ebenfalls eine LOGO-System.

Links:
https://archive.geogebra.org/de/wiki/index.php/Turtlegrafik (GeoGebra-Umsetzung einer Turtle-Grafik))

SIERPINSKI-Dreiecke (rekursiv)

```
TO sd :s :l
   CS
   sdrec :s :l
END

TO machA :l
   SETHEADING 90
   FD :l
END

TO machB :l
   SETHEADING 330
   FD :l
END

TO machC :l
   SETHEADING 210
   FD :l
END

TO sdrec :s :l
   IF :s = 0 [machA :l machB :l machC :l]
   IF :s > 0 [sdrec :s-1 :l/2 machA :l/2
              sdrec :s-1 :l/2 machB :l/2
              sdrec :s-1 :l/2 machC :l/2]
END
```

Q: https://docplayer.org/47624357-Turtlegrafik-in-logo.html

→ Erweiterung: SIERPINSKI-Teppich

Q: https://docplayer.org/47624357-Turtlegrafik-in-logo.html

KOCH-Kurve zeichnen (rekursiv)

```
TO kk :s :l
   CS
   RT 90
   kkrec :s :l
END

TO kkrec :s :l
   IF :s = 0 [FD :l]
   IF :s > 0 [kkrec :s-1 :l/3 LT 60
              kkrec :s-1 :l/3 LT 120
              kkrec :s-1 :l/3 LT 60]
END
```

Q: https://docplayer.org/47624357-Turtlegrafik-in-logo.html

Simulation von Sprachen und Grammatiken:

hier kontextfrei; jeweils nur eine Ableitungsregel je Symbol (D0L)

Q: https://docplayer.org/47624357-Turtlegrafik-in-logo.html

Eigenschaften formaler Sprachen

- Jede endliche Sprache lässt sich durch eine CHOMSKY-Typ-3-Grammatik darstellen.
- Es gibt unendlich viele Sprachen, die durch eine CHOMSKY-Typ-3-Grammatik erstellt werden können.
- Jede CHOMSKY-Typ-3-Grammatik ist auch eine CHOMSKY-Typ-2-Grammatik. (Die CHOMSKY-Typ-2-Grammatiken schließen den CHOMSKY-Typ-3-Grammatik ein.)
- Es gibt Sprachen, die durch eine CHOMSKY-Typ-2-Grammatik, nicht aber durch eine CHOMSKY-Typ-3-Grammatik beschrieben werden können.
- Jede ε-freie CHOMSKY-Typ-2-Grammatik (d.h. die keine Regel der Form N → ε besitzt) ist auch vom CHOMSKY-Typ-1.
- Es gibt Sprachen, die durch eine CHOMSKY-Typ-1-Grammatik, nicht aber durch eine CHOMSKY-Typ-2-Grammatik beschrieben werden können.
- Jede Grammatik vom CHOMSKY-Typ 1 ist auch immer eine Grammatik vom Typ 0.
- Es gibt Sprachen, die durch eine CHOMSKY-Typ-0-Grammatik, nicht aber durch eine CHOMSKY-Typ-1-Grammatik beschrieben werden können.

(komplexe) Aufgaben (zu Sprachen und Grammatiken)

x.

x. Geben Sie eine Grammatik für übliche ("normale") eMail-Adressen an!

x. Prüfen Sie ob die folgende EBNF für eine deutsches Ziffern-Datum exakt ist! Wenn JA, dann geben Sie eine Test-Menge an, mit der sowohl gültige und ungültige Daten geprüft werden / wurden! Wenn NEIN, geben Sie an, für welche Fälle ein Datum nicht gültig ist!

deutsche Datum-Format (nur Zahlen)

```
<Datum>       ::= <Tag>.<Monat>.<Jahr>
<Datum>       ::= <Tag>.<Monat>.
<Jahr>        ::= <zweistelligesJahr>
<Jahr>        ::= <vierstelligesJahr>
<zweistelligesJahr> ::= <Ziffer><Ziffer>
<vierstelligesJahr> ::= <Jahrtausend><Jahrhundert><zweistelligesJahr>
<Jahrtausend>       ::= "0"  |  "1"  |  "2"
<Jahrhundert>       ::= <Ziffer>
<Monat>             ::= [ "0" ] <NichtNullZiffer>
<Monat>             ::= "1" ( "0"  |  "1"  |  "2" )
<Tag>               ::= [ "0" ] <NichtNullZiffer>
<Tag>               ::= ( "1"  |  "2" ) <Ziffer>
<Tag>               ::= "3" ( "0"  |  "1" )

<Ziffer>            ::= → "0" | <NichtNullZiffer>
<NichtNullZiffer>   ::= → "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8"
                          | "9"
```

x. Gesucht ist die Grammatik für Neue römische Zahlen (Hier sind vier gleiche aufeinanderfolgende Symbole z.B. IIII möglich; damit sind Rückschritte nicht mehr notwendig)! (für 9 wird statt: IX jetzt VIIII geschrieben!)

x.

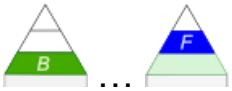
für die gehobene Anspruchsebene:

x.

x. Gesucht ist die Grammatik für die Römischen Zahlen!

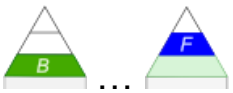
x.

3.2. Automaten



Problem-Fragen für Selbstorganisiertes Lernen
Was sind Automaten? Wie lassen sich Automaten definieren? Wieso gehören Automaten zur Theoretischen Informatik (TI)? Welche Arten von Automaten gibt es? Was ist der Unterschied zwischen Automaten und Maschinen in der Theoretischen Informatik? Warum beschäftigt sich die Theoretische Informatik mit (primitiven / einfachen / abstrakten) Automaten? Lässt sich alles mit den Automaten der TI erledigen / berechnen? Was haben Automaten mit Grammatiken und Sprachen zu tun? Welchen Bezug haben die theoretischen Automaten zu den Computern / zur praktischen Informatik? Was sind Transduktoren und Akzeptoren? Was leisten Transduktoren und Akzeptoren? Gibt es Automaten, die alles können? Warum kommen endliche Automaten nicht immer zu einem Ende?

3.2.0. Geschichtliches



Automaten gab es wohl schon vor der Antike. Die Bewässerungssysteme der Babylonier und der Römer sind vielleicht als die ersten "Automaten" anzusehen. Aus der Antike ist vielen von uns das Dampf-getriebene Tor-Öffnungssystem von HERON VON ALEXANDRIA (100 v.Chr.) durch den Geschichts-Unterricht bekannt.
Das Mittelalter war dann durch Hochleistungen im Puppen- und Mechanismen-Bau gekennzeichnet. Hierzu gehören ganz bestimmt auch die vielen klassischen (analogen) Uhren. Hochpräzise Chronographen ermöglichten die exakte Positions-Bestimmung auf hoher See. Viele der Kirchturm-Uhren funktionieren schon über Jahrhundert hinweg ziemlich genau.
Ein Meisterwerk des "Automaten"-Handwerks ist sicher die "Astronomische Uhr" der Rostocker Marienkirche. Sie wurde 1472 in Gang gesetzt und zeigt diverse astronomische Informationen an. 2018 musste allerdings eine neue Kalenderscheibe eingesetzt werden. Der Mechanismus funktioniert aber immer weiter.

Aufgaben:

1. ***Recherchieren Sie, welche Angaben von der "Astronomischen Uhr" der Rostocker Marienkirche genau angezeigt werden!***
2. ***Warum musste 2018 eine neue Kalenderscheibe eingebaut werden?***
3. ***Zeiger-Uhren werden oft als analoge Uhren deklariert. Ist diese Aussage korrekt? Begründen Sie Ihre Meinung!***

Einfache Automaten sind praktisch Steuerungs-System / einfache Programm-Abarbeitungs-Geräte; aus Eingaben oder Vor-Situationen ergeben sich immer gleiche Nachfolge-Situationen. Sie können praktisch nicht / kaum rechnen (und damit eigentlich auch nicht regulieren): Schon einfache Störungen bringen die Systeme aus dem "Konzept". Ein Speichern von Information ist auch nur selten möglich.

Der Begriff des Automaten geht auf das lat. ***automatus = freiwillig, aus eigenem Antrieb*** zurück. Auch die alten Griechen kannten ***automatos = von selbst geschehend***.
"Automatisch" ist als Wort tief in unsere Sprache eingebaut. Es fällt uns schwer, für das Adjektiv *automatisch* eine passendes Synonym zu finden, dass immer noch alle Aspekte der Eigenschaft beinhaltet. Dazu gehören z.B. die Charakterisierungen, wie: selbstständig, schematisch, mechanisch und reproduzierbar. Aber auch solche Charakterisierungen wie autonom, gleichförmig, widerspruchslos, unabhängig und maschinell sind oft zu gebrauchen.

Mit der industriellen Revolution – deren Anfang viele in Dampfmaschine und mechanischem Webstuhl sehen – begann dann eine Explosions-artige Entwicklung und Verbreitung von Automaten.

Klassische Automaten – häufig auch gleich als solche benannt – sind z.B.:

- Spiel-Automat
- Sicherungs-Automat (Leistungs-Schutzschalter)
- Fahrschein-Automat
- Foto-Automat
- Tauch-Automat (Luft-Regler, Atem-Regler)
- Automatic-Uhr
- (automatische) Waschmaschine (Wasch-Automat) (ich kenne noch Zeiten, da war eine "halbautomatische" Waschmaschine eine große Anschaffung im Haushalt; meine Oma hat noch an einem Tag in der Woche alle Wäsche hintereinander in einem großen Bottich gewaschen / gekocht und dann alles 5x gespült)
- Park-Automat
- Spiel-Uhr
- Vergaser (für Verbrennungsmotoren)
- Verbrennungsmotor
- Geld-Automat
- Automatic-Waffe
- Snack- / Getränke-Automat
- Sebstbedienungs-Kasse
- Rechen-Maschine
- Spül-Einrichtung beim WC
- Piep-Ei
- automatische Schranke (bei Parkplätzen, Grenzkontrollen oder an Bahnlinien)
- Computer
- Blenden- und Belichtungs-Automat (in einer Kamera)
- Tempomat
-

Unsere heutige Welt mit Milliarden von Computern und anderen automatischen Geräten ist sicher nicht der Endpunkt der Entwicklung.

Die modernen Systeme der IoT ("**I**nternet **o**f **T**hings"-Technologie, Web 4.0) eröffnen gerade eine nächste "Explosion" im Bereich der Automaten.

Ein genialer Erfinder und Automatenbauer war Leonardo TORRES Y QUEVEDO (1852 - 1936). Er entwickelte eine universeller Rechenmaschinen mit folgenden Eigenschaften:

- gekurbelter Rechenschieber zur Lösung algebaischer Aufgaben
- elektrische Schaltung zur Berechnung von $a*(y-z)^2$

Weiterhin konstruierte er Maschinen die folgende Probleme lösen konnten:

- El Ajedrestia – der Schachspieler (1912; 1920 verbessert) (setzte mit Turm und König den gegnerischen König matt; Ergebnis wurde von Schallplatte eingespielt)
- 1920 elektromagnetische Vier-Spezies-Rechenmaschine mit Schreibmaschine als Eingabe-Einrichtung; elektromagnetischem Speicher und Druckwerk

Von TORRES Y QUEVEDO stammen auch erste erste Ideen zum Thema Rechner-Netze (!!!).

Definition(en): Automat
Ein Automat ist eine Maschine, die vorbestimmte Abläufe / Handlungen / Prozesse gesteuert oder geregelt ausführt / ablaufen lässt.
Ein Automat ist ein technisches Gerät oder ein Modell, das die ausgehend von einer Ausgangs-Situation eine bestimmte andersartige End-Situation immer wieder (automatisch) reproduziert.
Ein Automat ist eine Zusammenstellung von Komponenten, die ein Ziel-orientiertes Ablaufen von Vorgängen weitgehend ohne ständige Einflußnahme durch den Menschen realisieren.
Ein Automat ist eine System, das selbst in der Lage ist sein Verhalten ohne das unmittelbare Einwirken des Menschen (oder eines anderen externen Systems) zu steuern.

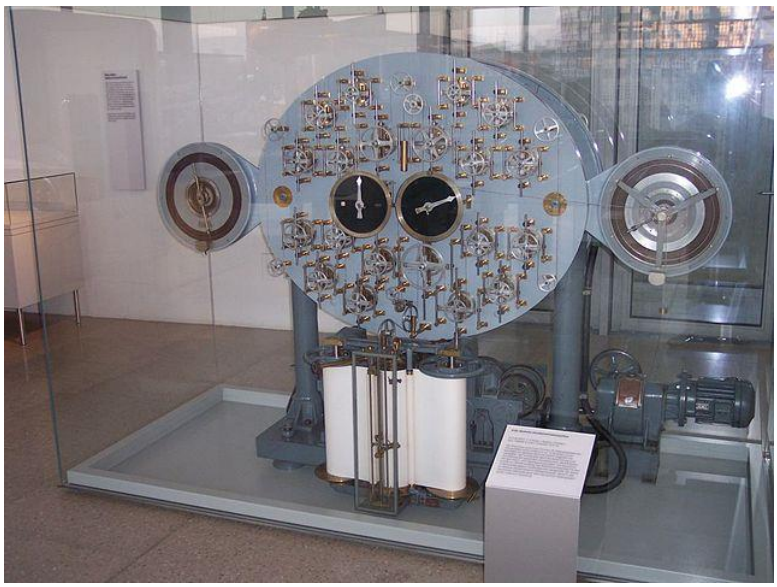
Automaten-Begriffe in der Gesellschaft

- **"Duden"** "In der Informatik bezeichnet man als Automaten vorwiegend mathematische Modelle von Geräten, die Zeichenfolgen verarbeiten und dabei Antworten geben, wobei die Eingabe oft nur gelesen, aber nicht verändert werden darf."
- **einfache Person** Ein Automat ist ein technisches Gerät, das nach einem Knopf-Druck einen Befehl ausführt.
- **Grundschul-Lehrkraft** Ein Automat ist eine Maschine oder ein Gerät, das eine Funktion z.B. nach einem auslösenden Signal, ausführt.
- **Medien-Pädagoge** Ein Automat generiert auf eine vorbestimmte, immer gleiche Weise, aus einer Eingabe (Input) eine erwartbare / vorhersehbare Ausgabe / Reaktion (Output).
- **Bildungs-Politiker** Ein Automat ist eine Maschine, die auf Knopfdruck reagiert.
-
-
-

nach/aus: LogIN 189/190 2018 S. 54

der Gezeiten-Rechner im Museum in Bremerhaven

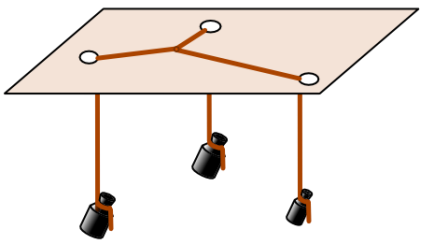
→ https://www.sat1regional.de/computergeschichte-in-bremerhaven-erster-deutscher-gezeitenrechner-laeuft-wieder/



Gezeiten-Rechenmaschine von 1915
im Deutschen Schiffahrts-Museum Bremerhaven
Q: de.wikipedia.org (Stahlkocher)

elektro-mechanischer Analog-Rechner

Prinzip eines "Analog-Rechner's" für Verteilungs-Probleme



gebaut in den 50er Jahren

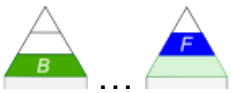
bis zum Ende der 60er Jahre im Betrieb

Gezeiten-Rechner der DDR
im Deutschen Schiffahrts-Museum Bremerhaven
Q: de.wikipedia.org (Dr. Karl-Heinz Hochhaus)

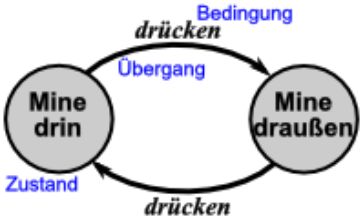
gebaut Ende der 30er Jahre

Welt-größte Gezeiten-Rechenmaschine
im Deutschen Museum München
Q: de.wikipedia.org ()

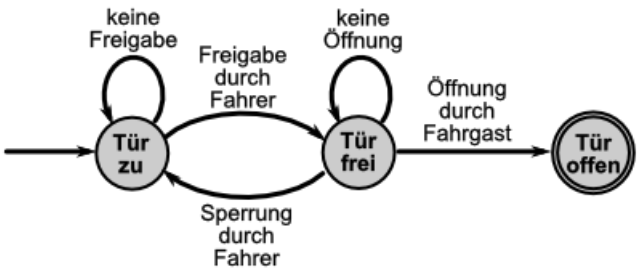
3.2.1. Einstieg



Einen kleinen Automaten haben wir fast alle täglich mit in der Schule – den Kugelschreiber. Es gibt zwei verschiedene Zustände, in denen sich der Automat Kugelschreiber befinden kann. Entweder ist die Mine drin oder draußen. Durch ein Drücken auf den oberen Knopf wechselt der Automat seinen aktuellen Zustand in den anderen.



In der Theoretischen Informatik beschäftigen wir uns besonders mit Automaten, die genau durch solche definierten Zustände charakterisiert sind.
Für die Zustände nutzt man in graphischen Darstellungen Kreise (selten Ovale). Die Übergänge (Transitionen, Zustandswechsel) zwischen den Zuständen werden durch Pfeile dargestellt. Dabei wird immer nur eine Richtung benutzt. An die Pfeile schreibt man die Bedingung ((das auslösende) Ereignis, Ursache) für den Übergang.
Die Abbildungen entsprechen schon sehr stark den klassischen Graphen.
Für irgendwelche (theoretischen) Betrachtungen bietet dieser Automat sicher nicht viel, aber wir wollen uns ja auch vorsichtig in die Problematik hineinarbeiten.
Nehmen wir deshalb ein etwas komplizierteres zweites Beispiel.
Die Tür-Mechanismen bei Nahverkehrs-Bussen oder –Zügen bzw. der Straßenbahnen kennen wir wohl auch alle.
Während der Fahrt lassen sich die Türen nicht öffnen, da sie vom Fahrer gesperrt sind. Drückt dieser den Freigabe-Knopf, dann leuchten offtmals die Tür-Schalter auf. Der Fahrgast kann nun den Taster an der Tür betätigen und die Tür öffnet sich.
Dieser Mechanismus gehört sicher zu den einfacheren Steuerungs- und Regelungs-Systemen. Das System ist dadurch gekennzeichnet, dass es mehrere definierte Zustände besitzt. Das sind z.B. die Zustände "Tür gesperrt" oder "Tür zu", "Tür frei" und "Tür offen".
Zu einem Zeitpunkt kann das System immer nur einen dieser Zustände (im Modell / Schema: Kreise) einnehmen. Für die Übergänge zwischen den Zuständen sind immer bestimmte Aktivitäten / Übergänge (Pfeile) notwendig.
Damit das System z.B. von seinem Ausgangs-Zustand – zu erkennen am einleitenden Pfeil – in den Zustand "Tür frei" wechseln kann, muss der Fahrer den Freigabe-Taster betätigen.



Übergangs-Graph / Modell eines "Tür-Öffner-Systems"

Erfolgt jetzt keine Öffnungs-Anforderung durch einen Fahrgast, bleibt die Tür im "Frei"-Zustand. Sperrt sie der Fahrer durch eine andere Taste, wechselt das System in den "Tür zu" ("Gesperrt")-Zustand. Nur wenn ein Fahrgast die Tür öffnen lässt, wechselt das System in den Endzustand (doppelter Kreis) "Tür offen".

Definition(en): Zustands-Automat
Ein Zustands-Automat ist ein Steuerungs-Konzept, bei dem Takt-getrieben der betrieb über definierte Zustände / System-Situationen erfolgt.
Ein Zustands-Automat ist ein Automat oder ein Modell, bei dem sich die Form / Konstitution, in denen sich der Automat gerade befindet, durch definierte Eigenschaften / Situationen wiederholbar beschreiben lässt. Die Situationen / Eigenschaften nennen wir Zustände.
Ein Zustands-Automat ist ein Automat, der über eine abzählbar große Menge von System-Zuständen und zwischen diesen Takt-orientiert nach festen Regeln gewechselt wird.
Eine Zustandsmaschine ein Gerät / System, das den Status von Irgendetwas zu einem bestimmten Zustand speichert und auf der Grundlage von Eingaben den Zustand wechselt und ev. eine bestimmte Aktion / Ausgabe dabei realisiert.
Zustands-Maschine ist ein Konzept zum Entwerfen von digitaler Logik / theoretischen Modellen / Computer-Programmen.
Eine Zustands-Maschine ist ein Verhalts-Modell, bei dem ein Systemüber eine endliche Anzahl von Zuständen zwischen denen das System definiert wechselt.
Eine Zustands-Maschine ist ein Berechnungs-Modell, dass in Hard- und / oder Software implementiert werden kann und zur Simulation von sequentieller Logik und einigen Computerprogrammen verwendet werden können.

Zustands-Automaten werden i.A. auch als endliche Automaten (Abk. EA) oder – etwas seltener – als Zustands-Maschinen bezeichnet. Was es damit auf sich hat und was da endlich ist, klären wir etwas später ab (→ x.y.z. endliche Automaten).
Im englisch-sprachigen Bereich werden Zustands-Automaten abgekürzt als FSM für finite state machine bezeichnet.
Leider gibt ein Problem mit den Begrifflichkeiten rund um Automaten und Maschinen. Mal wird von einem Zustands-Automaten und dann wieder von einer Zustands-Maschine gesprochen. Gemeint wird immer das gleiche Modell. Allgemein werden Automaten als spezielle Maschinen verstanden. In der Theoretischen Informatik ist eine Maschine aber wieder eine spzieller Automat. Hier sind Maschinen solche Automaten, die eine Ausgabe haben. Auch andere Reaktionen / Aktionen sind möglich. Im derzeitigen Gebrauch der Begriffe gibt es eigentlich keine saubere Klassifizierung. Vielmehr wird der Wortklang und / oder die Zugehörigkeit zu einer (Wissenschafts-)Schule als Maßstab angesetzt. Wir benutzen hier beide Begriff äquivalent. Oft entscheidet der Wortklang.

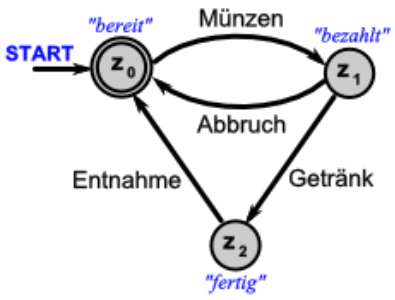
Aufgaben:

1. *Erweitern Sie das obige Zustands-Modell durch die "Haltewunsch"-Anforderung auf Fahrgast-Seite!*
2. *Erstellen Sie ein Zustands-Modell für die Tür-Schließung durch den Fahrer und die Kontrolle durch eine Lichtschranke!*
3. *Suchen Sie sich ein weiteres einfaches System mit klar definierten Zuständen aus Ihrer Lebenswelt heraus! Stellen Sie dazu ein Zustands-Modell auf!*

Viele Automaten aus der Praxis Automaten lassen sich in Gruppen oder Klassen einteilen. So gibt es Automaten, die bestimmte Aktivitäten akzeptieren und auf diese reagieren, z.B. Tür-Öffner, Park- oder Geld- Automaten. Andere sortieren Abfall oder Pakete. Bei wieder anderen halten sich die von außen sichtbaren System-Zustände sehr im Verborgenen.
Die verschiedenen praktischen Automaten lassen sich zudem meist in abstrakte Muster pressen. Das dient vor allem der einheitlichen Bearbeitbarkeit und einer möglichen allgemeinen theortischen Betrachtbarkeit. Gerade um die geht es auch in der Theoretischen Informatik. Hier werden sehr allgemeine Automaten-Modell benutzt, die mathematisch konzipiert sind. Solche Modelle sind aber die Voraussetzung dafür, um später z.B. zu erforschen, ob alles "berechenbar" ist oder ob jeder Automaten-Typ auch jedes Problem lösen kann. Dafür sind z.B. Sprachen und Grammatiken gern genutzte Arbeitsfelder. Schließlich wollen wir ja hundertprozentig sicher unsere Daten übertragen, Bankgeschäfte erledigen und ein smartes Zuhause steuern.
Zum Glück gibt es für die nicht so Mathematik-affinien unter uns auch die graphischen Modelle, die den Einstieg zumindestens deutlich vereinfachen.

Definition(en): (abstrakter) Automat
Ein (abstrakter) Automat (in der Theoretischen Informatik) ist eine Modell-Vorstellung von einer Informations-verarbeitenden Maschine.
Ein Automat der Theoretischen Informatik ist eine ideele Maschine zur Umsetzung eines Algorithmus, von Regelwerken (Grammatiken), von Steuerungs-Aufgaben oder Berechnungs-Modellen.
Ein abstrakter Automat ist ein mathematisches Modell für einfache Maschinen / Programme zur Lösung von bestimmten / definierten Problemen.

Gehen wir schon mal zu einem abstrakten Automaten über, der für eine klassische Anwendung in der Informatik steht.
Jeder kennt Parkschein- und Getränke-Automaten. Sie sind gern gewählte Einstiegs-Beispiele in der Automaten-Theorie.
Wir wählen hier mal einen Getränke-Automaten.
Wir haben schon festgelegt, dass Automaten in der Theoretischen Informatik (TI) durch Zustände gekennzeichnet sind. Für die graphische Darstellung wählt man Kreise, auch das haben wir ja schon besprochen. Man könnte den Zuständen eigene – sprechende / umgangssprachliche – Namen geben, so wie es im Modell blau in Anführungszeichen gemacht wurde.

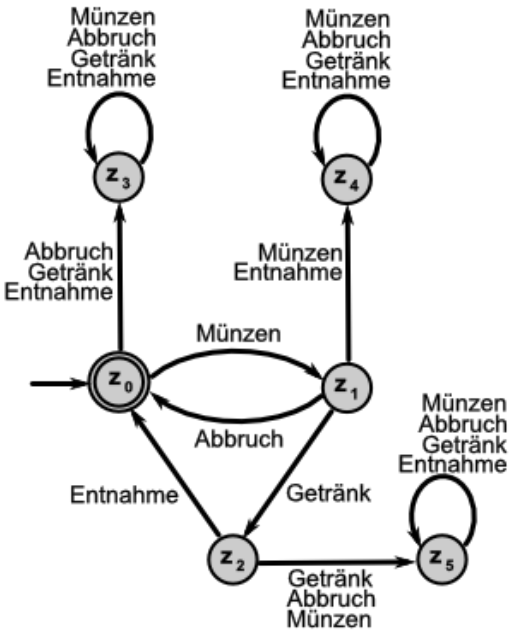


Für wissenschaftliche Betrachtungen sind diese Namen eher unwichtig (weil international kaum verständlich) und werden nur selten betrachtet.
Die in den Kreisen stehenden Namen sind die wissenschaftlichen Benennungen. Sie stehen für die nummerierten Zustände – hier z_0 bis z_2. Häufig findet man auch die Benennung der Zustände mit q_x oder nur mit Zahlen. Hier werden wir zumeist die Zustände mit z bezeichnen.
In einem Zustand kann sich der Automat beliebig lange befinden. Erst bei einer bestimmten Eingabe / Interaktion / einem Ereignis / einer Bedingung geht der Automat in den nächsten Zustand über. Diese Übergänge werden durch Pfeile dargestellt. An die Pfeile schreibt man die notwendige Eingabe / Interaktion für einen Zustandswechsel.
Der Endzustand – quasi ein besonders stabiler Zustand – wird doppelt eingekreist.

Aus der obigen Darstellung geht nicht hervor, was passiert, wenn der Automat irgendwie fehlbedient wird. Solche Betrachtungen sind aber in der Theoretischen Informatik auch wichtig.
Schließlich soll eine Überweisung auch erst dann abgeschlossen sein, wenn dass abgehobene (überwiesende) Geld auch wirklich auf dem Ziel-Konto angekommen ist.

Man kann jetzt die Modell-Darstellung um weitere Zustände und Übergänge erweitern, so dass in jeder Situation (- in jedem Zustand -) klar ist, was der Automat bei einer beliebigen Eingabe macht. Wir sprechen dann von einem vollständigen Automaten bzw. einem Automaten mit totaler Übergangs-Funktion.

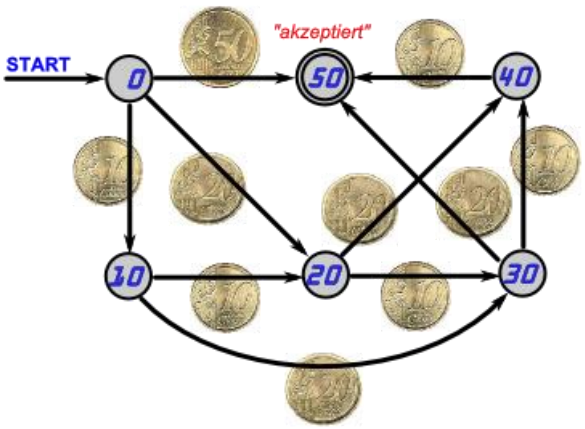
Die Bezeichnung totale Funktion ist mathematisch nicht ganz exakt, weil ja keine eindeutigen Hin- und Rück-Beziehungen bestehen. In der Theoretischen Informatik betrachtet man die Funktionen allgemein immer einseitig (meist von links nach rechts).



Die nebenstehende Graphik zeigt den vollständigen Automaten. Da für jeden Zustand bei einer beliebigen Eingabe immer ein bestimmter Folge-Zustand definiert ist, sprechen wir von einem deterministischen Automaten. Die gängige Abkürzung dafür ist **DA** oder **DEA** für **deterministischer endlicher Automat**.

Viele der gebräuchlichen Automaten haben in der Theoretischen Informatik solche Zwei- oder Drei-Buchstaben-Abkürzungen. U.U. werden auch die englischen Namen und Abkürzungen benutzt. Das wäre für einen deterministischen endlichen Automaten die Abkürzung für DEA bzw. DFA (deterministic finite automaton) oder DFM (deterministic finite state machine).

Der Bereich Münzen-Annahme könnte von uns nun als kleiner interner Automat gefasst werden.
Der Einfachheit halber betrachten wir einen Automaten ohne Rückgeld und ohne Überzahlung. Auch nimmt unser Automat nur die goldenen Cent-Münzen.
Die Zustände sind nach der Summe der bisher eingegebenen Geldwerte benannt und der typischen LED-Anzeige nachgeahmt.
Etwas übersichtlicher wird der Automat, wenn wir die eingegebenen Geldwerte an die Zustands-Übergänge notieren.



Aufgaben:

1. *Schreiben / Zeichnen Sie den Münz-Automat in eine TI-konforme Form um!*
2. *Handelt es sich um einen deterministischen Automaten? Begründen Sie Ihre Meinung!*

Was versteht nun unser 50-Cent-Münz-Automat? Er akzeptiert die folgenden Münz-Folgen:

- 50
- 20 20 10
- 20 10 20
- 10 20 20
- 10 20 10 10
- 10 10 20 10
- 10 10 10 20
- 10 10 10 10 10

Diese Münz-Folgen stellen praktisch die Sprache unseres Münz-Automaten dar. Automaten und Sprachen und damit auch Grammatiken bilden eine feste Einheit in der Theoretischen Informatik. Manchmal sind die einzelnen Teile kaum noch voneinander zu trennen. Es kommt da mehr auf die Art der Modell-Darstellung als auf den Namen an.
Die Münzen stellen das Alphabet unseres Automaten dar. Somit lässt sich das Alphabet mit:

$X = \{10, 20, 50\}$ → **endliche Menge der Eingabe-Symbole**

Weiterhin ist der Automat duch die Zustände gekennzeichnet – die als Digital-Anzeige – irgendwie für den Benutzer des Automaten sichtbar ist. Würden wir die Zustände aus der Graphik ableiten hätten wir so ähnliche Bezeichnungen, wie die Zeichen des Münz-Alphabet's. Deshalb benennen wir sie hier einfach in z_{10}, z_{20} usw. usf um.
Damit können wir ein weiteres Detail unseres Automaten zusammentragen:

$Z = \{z_0, z_{10}, z_{20}, z_{30}, z_{40}, z_{50}\}$ → **endliche Menge der Zustände**

Zwei der Zustände sind besonders wichtig. Das ist zum Einen:

$S = z_0$ → **der Start-Zustand**

Und zum Anderen ist das der Zustand, der den vollen zu zahlenden Geldwert repräsentiert. Wie wir später noch sehen werden kann es auch mehrere solchen End-Zustände geben. Hier ist es nur ein einzelner, der aber als mögliche größere Gruppe gefasst wird:

$Z_E = \{z_{50}\}$ → **Menge der akzeptierten End-Zustände**

Bleiben noch die Übergänge zwischen den Zuständen. Sie sind praktisch das Regelwerk (die Grammatik) unseres Automaten:

$f = \{z_0 \times 10 \rightarrow z_{10},$ **Überführungsfunktion**
$z_{10} \times 10 \rightarrow z_{20},$
$z_{20} \times 10 \rightarrow z_{30},$
$z_{30} \times 10 \rightarrow z_{40},$
$z_{10} \times 20 \rightarrow z_{30},$
$z_{20} \times 20 \rightarrow z_{40},$
$z_{30} \times 20 \rightarrow z_{50},$
$z_{40} \times 10 \rightarrow z_{50}\}$ → **Menge der Überführungen**

Die Überführungs-Funktion ist also die Gesamtheit aller Übergänge.
Damit haben wir alles zusammen, was unseren Automaten charakterisiert:

$A = (\{10, 20, 50\}, \{z_0, z_{10}, z_{20}, z_{30}, z_{40}, z_{50}\}, z_0, f, \{z_{50}\})$

Das lässt sich so verallgemeinern:

A = (X, Z, S, f, ZE)

Genau solche Tupel (hier ein Quint-Tupel / 5-Tupel) werden wir später zur exakten Definition der speziellen Automaten benutzen.

Definition(en): Tupel
Ein Tupel ist ein (gegliederte / geordnete / strukturierte) abgezählte Liste, dass eine Gruppe vom mathematischen Objekten zusammenfasst. Häufig wird die Anzahl der enthaltenen mathematischen Listen-Elemente zur Kurz-Charakterisierung der Tupel benutzt. (z.B. 4-Tupel oder Quadtupel)

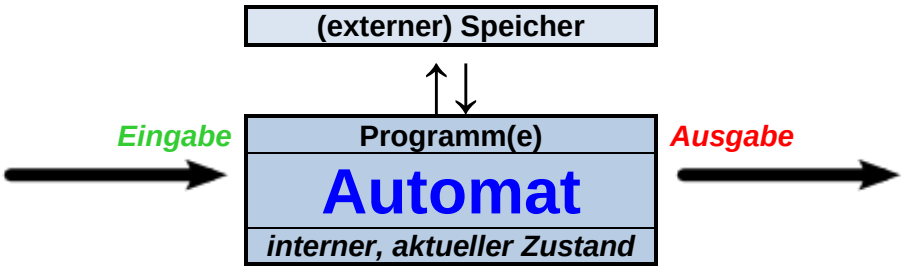
Aufgaben:

1. *Machen Sie aus dem obigen Münz-Automaten einen DA!*
2. *Erstellen Sie einen endlichen Automaten, der genau 70 Cent akzeptiert! (Es sind nur die Gold-farbenen Münzen zugelassen!)*
3. *Geben Sie die Sprache des 70-Cent-Münz-Automaten an!*

für die gehobene Anspruchsebene:

4. *Geben Sie eine allgemeine Tupel-basierte Definition eines 1€-Automaten an! (Akzeptiert werden nur die goldenen Cent-Münzen!)*

Deterministische Automaten können als Bestandteile von Informations-verarbeitenden Maschinen betrachtet werden. Als wesentlicher neuer Bestandteil – im Vergleich zu einem einfachen Automaten – kommt Speicher hinzu.



Wie wir später sehen werden sind viele der betrachteten Automaten der Theoretischen Informatik auch schon richtige Informations-verarbeitende Maschinen. Z.B. verfügen Keller-Automaten oder TURING-Maschinen schon über Speicher-Möglichkeiten.
Die Benennung in Automaten (also ohne Speicher) und Maschinen (mit eigenem Speicher) ist historisch bedingt und wird aber leider nicht immer konsequent verfolgt. Häufig wird der besser klingende Name oder eine im Sprachraum geprägte Form benutzt.

Exkurs: Graphen

Graphen G bestehen aus Knoten und Kanten. Knoten sind die Start- bzw. Ziel-Punkte für die Kanten.
Betrachten wir als Beispiel eine Landkarte mit ein paar Orten. Diese Orte können abstrakt als Punkte – und damit als Knoten – betrachten.
Ein Kante (engl.: edge) verbindet immer genau zwei Knoten (engl.: node(s)). Im Besipiel wären die Straßen zwischen den Orten die Kanten (Verbindungen).
Somit können wir auch schreiben:
G = (N,E), wobei N für die Menge der Knoten steht und E für die Menge der Kanten.
Jede Kante n kann über die beiden Knoten definiert werden, die sie verbindet n = (e_1,e_2). Solche Paare von Objekten – hier Knoten - werden Tupel genannt.

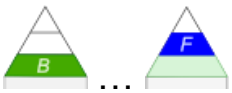
Die Kanten können eine Orientierung bzw. eine Richtung haben. Das hieße. z.B. es gibt nur einen Weg von Knoten 1 zu Knoten 2 und genau in diese Richtung. Dann ist dies eine gerichtete Kante. In den graphischen Darstellungen benutzt man einfach einen Pfeil. In unserem Karten-Beispiel wäe das eine Einbahnstraße von dem einem Ort zum anderen.
Sind beide Richtungen möglich, dann ist die Kante bidirektional. Man schreibt beide Pfeile oder aber man verzichtet auf die Pfeile.

Darstellung von Graphen in Computern z.B. auch als Adjazenz-Matrix.

Adjazenz-Listen

Liste von Adjazenz-Listen

3.2.2. Einteilungen



In Allgemeinen werden Zustands-Automaten in der TI mit den endlichen Automaten gleichgesetzt. Das ist auch für unsere Zwecke völlig ausreichend.

ein Zustand ist ebenfalls ein Modell-Objekt
kann z.B. durch Spannungen / Potentiale oder bestimmte Signale repräsentiert werden
ein einzelnes Bit z.B. in einem Speicher kann als Zustand (eben 0 oder 1) verstanden werden
in Speichern sind aber mittlerweile Millionen / Milliarden von Bit's vorhanden
das schraubt die Zustands-Menge extrem in die Höhe ($O(2^n)$) → "Kombinatorische Explosion"
hier macht es nicht immer Sinn jedes Bit einzeln zu betrachten; Zustände können also auch im Komplex betrachtet werden

Definition(en): Zustand
Ein Zustand ist die Situation eines Systems zu einem bestimmten Zeit-Punkt oder Arbeits-Schritt.

??? bessere Position suchen
hier wäre die Betrachtung als Byte schon effektiver

Definition(en): Zustands-Automat
Zustands-Automaten sind Automaten (im Sinne der Theoretischen Informatik), die durch diskrete Zeitpunkte (oder Abfolgen) und zugehörige (innere) Zustände gekennzeichnet sind.

Zunächst können Automaten sicher nach ihrem Verwendungs-Zweck oder den Anwendungen unterschieden werden:

nach praktischen / technischen Verwendungs-Gebieten

- **Steuerungs-Automaten** z.B.: Waschmaschine, Ampel, Computer, Piep-Ei, …
- **Lexikalische Analyse** z.B.: Compiler-Bau, Suche von Variablen-Bezeichnern od. Tag's, …
- **Text-Suche** z.B.: Suche von Web-Seiten oder Dokumenten anhand von Schlüssel-Begriffen, …
- **Software und Protokolle** z.B.: Geräte-Kommunikation, sicherer Datenaustausch, …
- **Entwurf und Testung digitaler Schaltungen** z.B.: Entwicklung von Chip's, …
-

Für die Theoretische Informatik (TI) hat diese Einteilung aber kaum eine Bedeutung. Hier sind die meisten Automaten zuerst einmal theoretische Modelle für vorrangig akademische Zwecke. Bei einigen Automaten aus dieser Kategorie kommt dann aber auch ab und zu ein Praxis-Bezug und eine praktische Anwendung durch. Dazu gehören ganz bestimmt die Keller-Automaten (→ 3.2.7. (deterministische) Keller-Automaten) und die Register-Maschinen (→ 3.2.10. Register-Automaten).
Bei den nachfolgenden Kategorisierungen beschränken wir uns auch fast ausschließlich auf die theoretischen Automaten der TI. Automaten, die zum Vergleich interessant sind, aber in diesem Skript keine Rolle spielen, werden ev. aber auch gleich mitdefiniert. Eine weitere Besprechung erfolgt dann aber nicht.
Ev. ist die Einteilung an dieser Stelle noch zu abstrakt. Sie greift auf die diversen Theorien und Betrachtungens-Richtungen der TI zurück. Wem das hier zu hoch erscheint, der möge sich erst einmal durch die anderen Kapitel arbeiten, um dann hier zusammenfassend zurückzukehren. Ansonsten werden hier jetzt einige bedeutende Kriterien für die Einteilung von Automaten kurz vorgestellt. Dann hat man eine Stelle, an der man später noch mal kurz zum Nachlesen oder Nachorientieren zurückspringen kann.
Eine der klassischen Systematisierungen von Automaten der TI beginnt bei der Frage-Stellung nach vorhandenen oder eben nicht vorhandenen Zuständen (innere Automaten-Situationen).

nach: Anzahl und Vorhandensein von Zuständen

- **endliche Automaten (Zustands-Automaten)** haben eine endliche / abzählbare Menge von Zuständen
  → 3.2.3. endliche Automaten
- **Zustands-lose Automaten** haben keine Zustände
  (z.T.: → PETRI-Netze)
-
- **autonome Automaten** auton. A. besitzen keine Eingänge, agieren unabhängig von Umwelt-Bedingungen; durchlaufen – meist zyklisch – immer die gleichen Zustände

Eigentlich müsste man die autonomen Automaten von solchen mit Eingängen (nicht-autonome A.) unterscheiden. Aber eine solche Unterscheidung nach der Umwelt-Abhängigkeit ist nicht üblich.
Auch wenn Automaten der Anschein von Perfektionismus umgibt, nicht alle sind so eindeutig und klar berechenbar in ihrer Arbeitsweise. Fehler und Probleme sind da ev. vorprogrammiert. Mittlerweile wissen wir ja auch, dass Maschinen – auch Computer – wirklich auch Fehler machen. Und damit sind nicht nur die gemeint, die der Mensch bei ihrer Konstruktion, Produktion oder Inbetriebnahme gemacht hat oder billigend in Kauf nimmt.

nach: Eindeutigkeit des Ablaufs

- **deterministischer A.** — für jede Eingabe und jeden Zustand ist die weitere Funktion des Automaten klar definiert; somit ist auch nur ein Start-Zustand möglich
  Automat hat mindestens einen definierten Endzustand
  → 3.2.6.1. deterministische endliche Automaten
- **nicht-deterministischer A.** — es gibt mehrere mögliche Arbeits-Wege, d.h. es sind mehrere verschiedene Zustandswechsel zulässig, die sich später auch als nicht geeignet herausstellen können und dann durch andere Wege ersetzt werden können
  n.-det. A. können mehrere Start-Zustände haben und bedürfen nicht zwangsläufig eines Endzustandes
  → 3.2.6.2. nicht-deterministische endliche Automaten
-

In einigen Automaten-Kategorien sind mehrere charakterisierende Merkmale vereint, so dass eine kurze Bezeichnung – wie Fachleute sie lieben – nur schwerlich möglich ist. Oft sind sie auch mit einem speziellen Wissenschaftler / Forscher / … historisch verbunden. Um deren herausragende Idee zu würdigen, werden auch einige Automaten nach ihren Erfindern bzw. Erstbeschreibern benannt:

nach: Erfinder oder Erstbeschreiber

- **MOORE-Automat** — umsetzender / übersetzender Automat, bei die Ausgabe im Zustand erfolgt
- **MEALY-Maschine**
- **TURING-Maschine** — universeller Automat mit einem Speicher-Band
- **MEDWEDJEW-Automat** (Semi-Automat) — Folge-Zustände ergeben sich aus dem jeweils aktuellen Zustand und einem aktuellen (variablen) Eingangs-Signal; besitzt keine Ausgabe-Funktion; nur die erreichten Zustände sind entscheidend
- **PETRI-Netze**
-

Bei wieder anderen Automaten stehen ganz bestimmte Bau- oder Konstruktions-Konzepte im Vordergrund. Diese waren dann Namens-gebend.

nach: Bau-, Speicher- bzw. Arbeits-Pinzip

- **Keller-Automat** — besitzen einen Keller-Speicher (Stack) nach dem LIFO-Speicher-Prinzip
- **Register-Automat** — verfügen über mehrere Speicher-Zellen (Register genannt), die einzeln und unabhängig angesprochen werden können
- **Band-Maschine** — Speicher-Medium ist ein (potentiell) unendliches Speicher-Band über das ein Lese-(Schreib-)Kopf bewegt wird
-
- **Semi-Automat** (MEDWEDJEW-Automat) — Folge-Zustände ergeben sich aus dem jeweils aktuellen Zustand und einem aktuellen (variablen) Eingangs-Signal; besitzt keine Ausgabe-Funktion; nur die erreichten Zustände sind entscheidend

nach: Zusammenstellungs- und Kombinations-Techniken von Automaten

- **Künstliche neuronale Netze (KNN)** — kontinuierliche Informations-Verarbeitung in Gruppen von Automaten (, die Neuronen nachbilden)
- **Zelluläre Automaten** — synchrone, getakte Arbeit von parallel angeordneten Automaten, die mit einem Speicher-Band arbeiten
- **Fleißige Biber** — (ev. auch komplexe) Zusammenschaltung von definierten TURING-Maschinen

Eine ebenfalls klassische Unterscheidung der Automaten kategorisiert nach der Art der Kommunikation des Automaten mit seiner Umwelt, wobei vorrangig die Ergebnis-Mitteilung gemeint ist.

nach: der Kommunikation des Ergebnisses / Arbeits-Verlaufs

- **Automaten mit Ausgabe (Transduktoren, Maschinen)** = umsetzende / übersetzende Automaten
- **Automaten ohne Ausgabe (Transitions-Systeme; Akzeptoren)** = erkennende / akzeptierende Automaten
-

Die Größe bzw. der Umfang des Eingabe-Bereichs / -Alphabetes gilt es ebenfalls zu beachten.

nach: der verarbeiteten Signal-Art (nach der Art der Signal-Verarbeitung / nach dem Eingabe-Alphabet)

- **binär** es gibt nur die Eingangs-Signale 0 bzw. 1
- **diskret** es gibt eine begrenzte / abzählbar große Menge an unterschiedlichen Eingangs-Signalen
- **analog** es gibt unendliche viele Niveaus der Eingangs-Signale und es lassen sich auch wieder immer beliebige Zwischenwerte ermitteln
  spielt in der klassischen TI eine kleinere Rolle; heute mit sogenannter FUZZY-Logik und bei Neuronalen Netzwerken von Bedeutung

Auf den ersten Blick würde man vielleicht denken, dass in der TI nur die binären Automaten betrachtet werden. Das stimmt aber nicht. Automaten, die nur mit 0 und 1 arbeiten sind meist nur Sonderfälle von diskreten Automaten. In der praktischen Umsetzung von Automaten in Schaltungen usw. spielen sie aber logischerweise die herausragende Rolle.
Für die Schule sind nur diskrete Automaten interessant. In die diskrete Signal-Art ist natürlich das binäre Signal (0 oder 1 bzw. O oder L bzw. AN oder AUS usw. usf.) eingeschlossen. Praktisch ist auch ein Alphabet der Form {a,b} binär. In jedem Fall ist es diskret.
Nur wenige Einteilungen erfassen alle möglichen oder denkbaren Automaten. Daneben sind aber auch noch weitere Einteilungen denkbar (z.B. nach Farbe oder Größe). Für informatische Zwecke sind sie wohl nicht wirklich sinnig.
Der Speicher – seine Größe und auch der Aufbau – dient als Grundlage für eine weitere Einteilungs-Möglichkeit:

nach: Art und Größe des verfügbaren (Daten-)Speichers

- **endlicher Automat (kein Speicher)** Automaten ohne Speicher, arbeiten also praktisch immer online
- **Keller-Automat (Keller-Speicher, Stack, LIFO-Speicher)** besitzen einen Last-In-First-Out-Speicher (zuletzt Rein – zuerst Raus) – als Keller-Prinzip bekannt
- **TURING-Maschine (unendlicher/s Speicher(-Band))** TURING-M. besitzen ein Speicher-Band; meist mit unendlich vielen Speicher-Stellen
- **linear beschränkter (TURING-)Automat (begrenzter/s Speicher(-Band))** lin.-beschr. TA besitzen ein auf die Eingabe begrenztes Speicherband
  (quasi: Ausgabe-Speicher = Eingabe-Speicher)
-

Eine weitere Einteilung könnte nach der Taktung der Systeme unternommen werden:

nach: der Rhythmerisierung

- **synchron (getaktet** (diskontinuierlich)**)** — besitzen eine Takt-Geber; führen zu jeder Zeit / zu jedem Takt etwas aus (und wenn es eine spezielle Nicht-Operation (NOP) ist) aus
  Arbeit des Automaten ist von den Eingaben und der Zeit (dem Takt) abhängig
- **asynchron (kontinuierlich)** — keine äußere Taktung; Arbeit von der Änderung von Umgebungs-Bedingungen / Eingaben abhängig
-

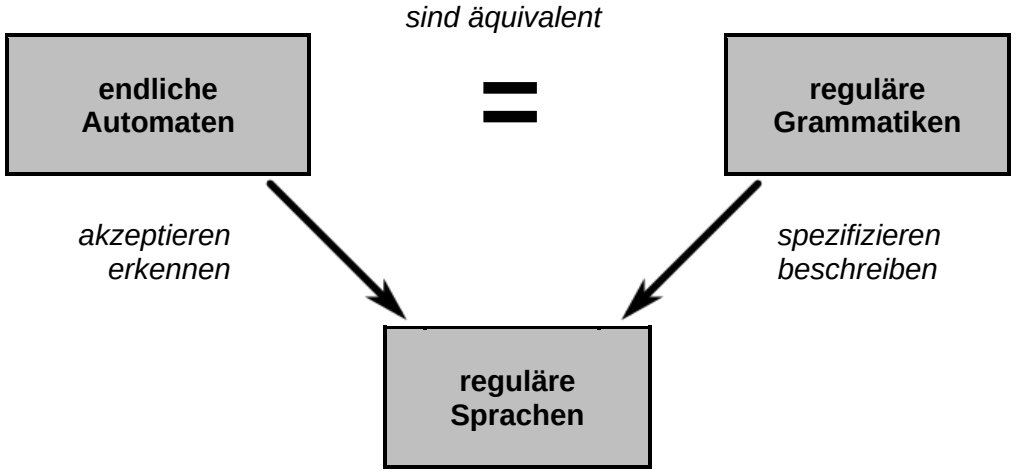
Nimmt man die Defintion der asynchronen Automaten ganz genau, dann sind viele einfache Automaten asynchron. Erst die komplexeren / höheren Automaten / Maschinen sind dann mehr Takt-betont. Letztendlich sind unsere Informations-verarbeitenden Maschinen (Computer) immer Synchron- / Takt-Maschinen.

Definition(en): asynchrone Automat
Asynchrone Automaten sind Automaten (im Sinne der Theoretischen Informatik), die durch eine (kontinuierliche) Feedback-Funktionen und (gleichzeitig) nicht durch äußere Taktung / Zeitgeber bestimmt werden.

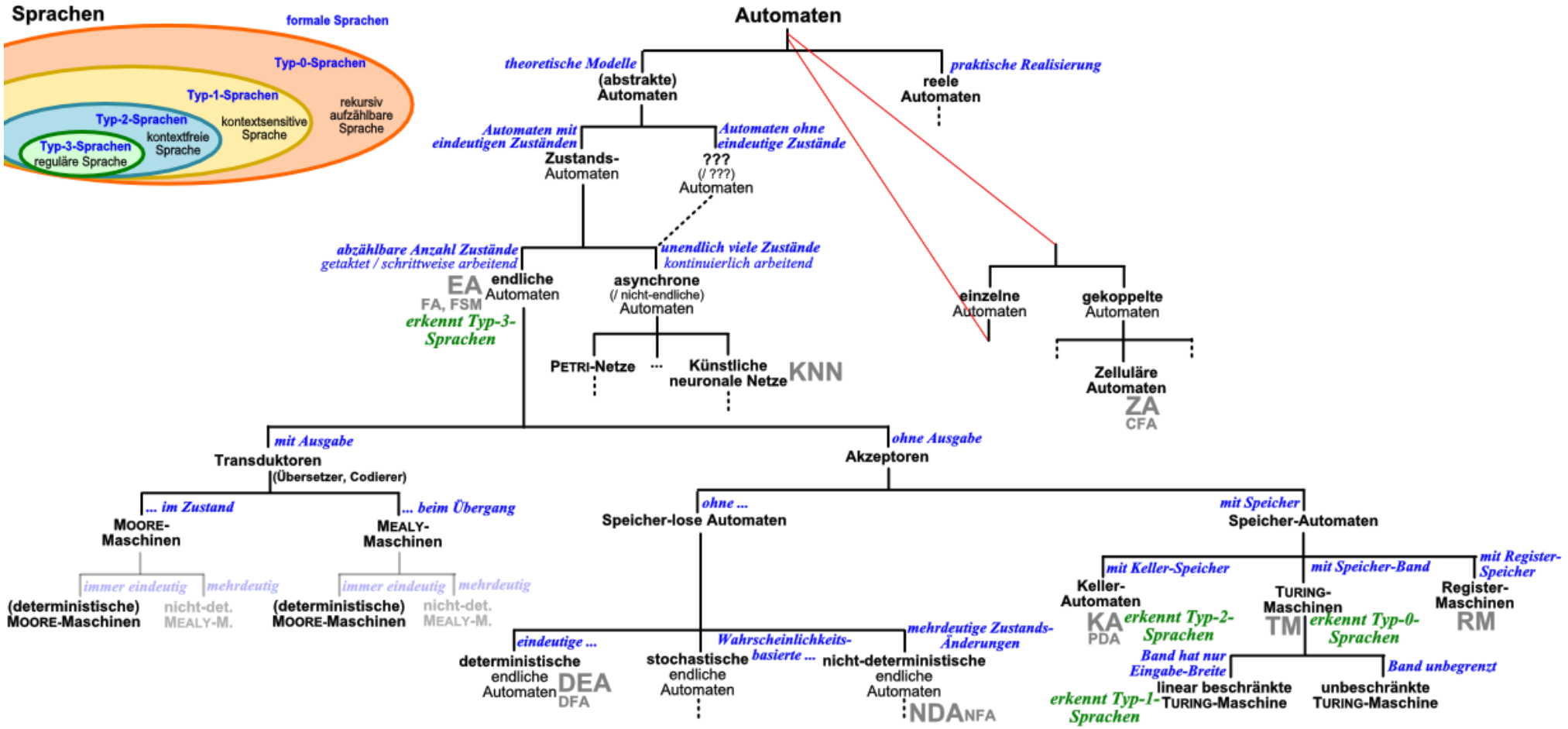
In den nachfolgenden Abschnitten werden vorrangig die Schul-relevanten Automaten vorgestellt. Der eine oder andere Automaten-Typ interessiert auch nur informativ oder könnte das Thema eines extra Vortrag's im Unterricht werden.

??? Einordnung: Zeit-Automaten → MevS_3.pdf

ev. noch anpassen!



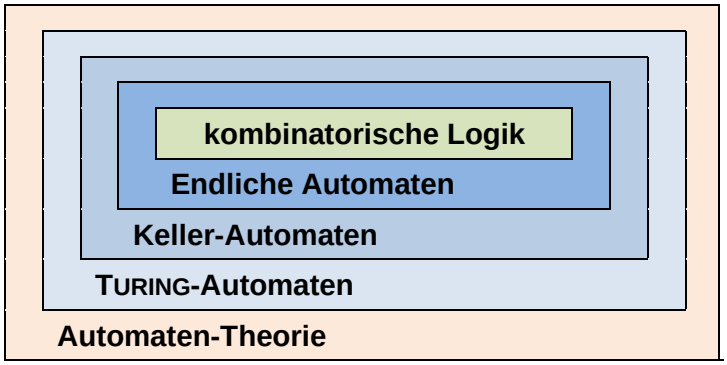
Vorschlag einer Systematisierung von Automaten (mit Beziehungen zu Sprachen) für den Schul-Gebrauch:

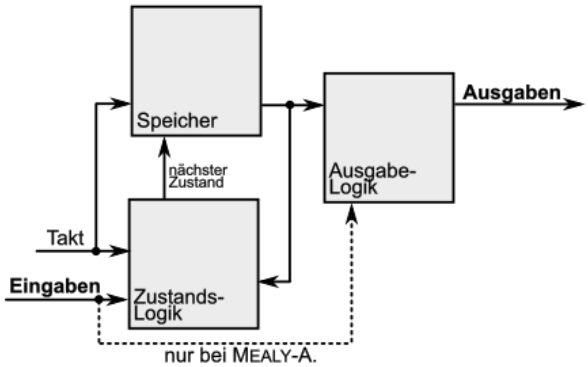


weitere Aspekte zur Einteilung von Automaten

Merkmal	MEALY-A.	MOORE-A.	HAREL-A.	UML	
Zustände und Übergänge	+	+	+	+	
Übergang erzeugt Ausgabe	+		+	+	
Zustand erzeugt Ausgabe		+	+	+	
Tiefe (Hierarchien, zusammengesetzte Zustände)			+	+	
Orthogonalität (parallele, untergeordnete Zustands-Maschinen)			+	+	
Rundruf-basierte Kommunikation			+	+	
Geschichte / Historie (Speicher); Aktionen; Verzögerungen; Auszeiten; Bedingungen			+	+	

Automaten-Klassen





3.2.3. Zustands-Diagramme

Einzelne Diagramme haben wir schon benutzt. Diese waren meist sehr einfach gestrickt. Für wissenschaftliche Zwecke und praktische Anwendungen – vor allem im internationalen Umfeld – sind aber standardisierte Diagramm-Formen notwendig.
Für die wissenschaftliche Nutzung sind HAREL-Diagramme eine gute Wahl. Praxisorientierter sind die UML-Diagramme, die i.A. eine leichte technische / praktische Umsetzung ermöglichen.

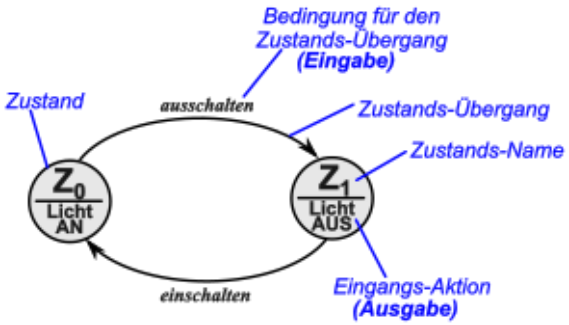
Allgemein sind Zustands-Diagramme Graphen mit den üblichen Knoten und Kanten. Letztere entsprechen den Verbindungen / Übergängen (Transitionen). Zustände werden als Knoten umgesetzt.

3.2.3.1. Diagramme nach HAREL

1987 von David HAREL veröffentlicht
gute Standardisierung, die vieles mit den klassischen Zustands-Diagrammen gemeinsam hat
klassische Diagramme sind meist leicht in die HAREL-Form zu überführen

Ist ein Zustand ein End-Zustand, dann wird dieser doppelt umkreist. Ein Pfeil ohne einen Ausgangs-Zustand benutzt man zum kennzeichnen des Start-Zustand's.

Die Bezeichnung kann frei gewählt werden. Das können sprechende Namen sein oder einfach nur Nummern bzw. Buchstaben.
Üblich ist eine Kennzeichnung als **Z** mit indizierter, fortlaufender Nummer.



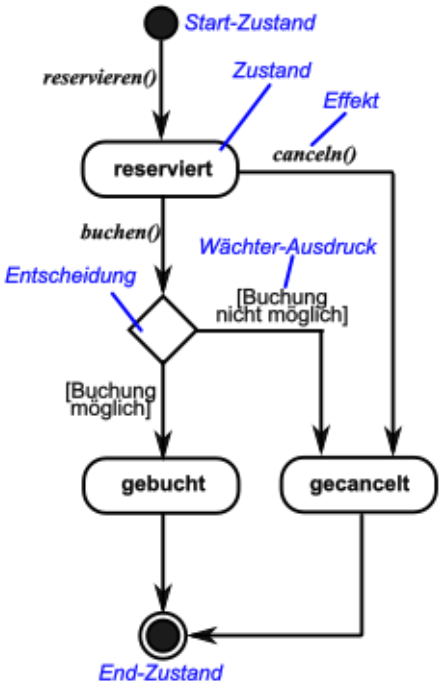
3.2.3.2. UML-Zustands-Diagramme

Zustände in Rechtecken mit abgerundeten Ecken geschrieben

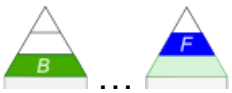
ähnlich, wie bei Objekten dürfen unter einer abgegrenzten Überschrift im Rechteck auch eine Liste der innerenen Aktivitäten / Methoden geführt werden
mögliche Aktivitäten können beim Eintritt in bzw. beim Austritt aus den Zustand ausgelöst werden
weiterhin ist eine Aktivität (do bzw. doActivity) die erledigt wird, solange der Zustand aktiv ist (sich das System in diesem Zustand befindet)
für die verschiedenen möglichen Ereignisse (Eingaben, …) können untergeordnete Routinen / Funktionen definiert werden

in alternativen Darstellungs-Formen ist auch die Notierung des Zustands-Namens als Reiter über dem Rechteck-Symbol zugelassen

Ein schwarzer Punkt dient als Start-Zustand. Von hier beginnen die Zustands-Änderung. Über eine Effekt kann von einen Zustand in den nächsten gewechselt werden. Dabei helfen die Effekte, also solche Methoden, Funktionen, …, die das System von einem Zustand in den nächsten übergehen lassen.
An bestimmten Stellen sind Verzweigungen notwendig. Über sogenannte Wächter-Ausdrücke (praktisch "Bedingungen") werden – je nach Entscheidungs-Kriterien – unterschiedliche Effekt möglich.
Am Ende sollte das System zu einem Endzustand finden. Diesen erkennt man am schwarzen, umkreisten Punkt. Ein X-förmiges Kreuz ist ebenfals als Symbol für ein End-Zustand möglich.
Es gibt weitere Symbole für spezielle Zustände und Strukuren. Diese benötigen wir aber nicht.
In der Theoretischen Informatik werden aber nur selten Automaten über UML-Diagramme dargestellt. Hier haben sich die HAREL-Diagramme mehr bewährt.



3.2.4. endliche Automaten



übliche Abk.: EA
i.A. auch: Zustands-Maschine, Zustands-Automat; engl. finite state machine (FSM) bzw. finite state automat (FSA)

Die wissenschaftliche Grundlagen und Inspirationen für die Automaten-Theorie stammten aus den Bereichen Biologie (MCCOLLOUGH, PITTS: Modellierung des Gehirns; 1943) der Elektrotechnik (MEALY: Schaltkreis-Entwurf; 1955) und der aufstrebenden Linguistik (CHOMSKY: Grammatiken; 1956).

Darstellung als Zustands-Graphen
Knoten (Node)
Verbindungen (Kanten; Edges)
Notierung der Übergangs-Funktion an die Kanten
ist keine Funktion notiert, dann erfolgt im nächsten Takt eine zwangsläufige Änderung in den nächsten Zustand

endliche Automaten sind als abstrakte Automaten relativ gut zu definieren, analysieren und theoretisch zu bearbeiten
alle ihre Berechnungen / Funktionen führen nach einer begrenzten / abzählbaren Anzahl von Schritten / Zustands-Veränderungen zu einem Ende (unabhängig vom Resultat)

Weiter vorne haben wir schon dargestellt, dass es zum Einen Automaten gibt, die Worte akzeptieren – also prüfen, ob diese zur Sprache des Automaten gehören. Zum Anderen können Automaten Sprachen generieren. Dabei geht es um die Suche nach allen möglichen Ausgaben / akzeptierbaren Wörtern eines Automaten / Suche nach den erkennbaren Sprachen des Automaten.
Die Sprachen endlicher Automaten sind i.A. einfach. Das bedeutet sie folgen einfachen Grammatiken. Sachlich werden von endlichen Automaten die Sprachen vom Typ 3 nach CHOMSKY erkannt bzw. produziert. Diese Sprachen heißen auch reguläre Sprachen.

Beispiel: MEDWEDJEW-Automat → sdigi05.pdf

Unter dem Verhalten eines (Zustands-)Automaten versteht man die Menge der aufeinanderfolgenden Zustände. Sie wird maßgeblich durch deren Anzahl im Zusammenhang mit der inneren Struktur des Automaten bestimmt.

Aktionen sind die Ausgaben eines Automaten

Aktions-Typen eines endlichen Automaten

- **Eingangs-Aktionen** Ausgabe wird beim Eintreten in den Zustand generiert
- **Ausgangs-Aktionen** Ausgabe wird beim Verlassen des Zustands generiert
- **Eingabe-Aktionen** Ausgabe wird abhängig vom aktuellen Zustand und der Eingabe generiert
- **Übergangs-Aktionen** Ausgabe wird abhängig von der Zustands-Änderung generiert

Wie wir schon gezeigt haben, unterscheidet man zwei Typen von Automaten, solche mit Ausgabe und solche ohne. **Automaten mit Ausgabe** werden **Transduktoren** genannt. Die **Automaten ohne Ausgabe** heißen **Akzeptoren**. Sie verarbeiten die Eingabe und wechseln dabei nur ihre Zustände. Trotzdem ist von außen immer sichtbar, in welchem Zustand sich der Automat befindet und ob es sich um einen End-Zustand handelt.

Funktionsweise eines endlichen Automaten:

- der Automat startet immer mit dem Start-Zustand
- durch eine Eingabe (meist eines Zeichens) wird in den nächsten Zustand gewechselt (Übergang ist durch den Pfeil gekennzeichnet; die Eingabe steht über dem Pfeil)
- als akzeptiert gilt eine Zeichen-Folge (Folge von Eingaben), wenn sich der Automat am Ende in einem End-Zustand befindet
- es erfolgt keine Speicherung der Wege / Zustands-Folgen (es kann aber ein Protokoll über den Arbeits-Verlauf aufgenommen werden)

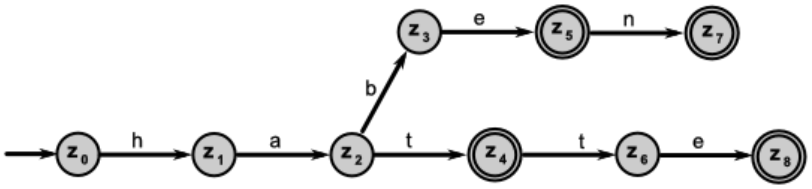
Definition(en): endlicher Automat
Ein endlicher Automat ist ein Automat mit einer endlichen (abzählbaren) Menge von (möglichen) Zuständen.
Ein endlicher Automat ist eine alternative Modell-Vorstellung zu / Darstellungs-Form einer regulären Grammatik.
Ein Automat ist endlich, wenn seine Anzahl möglicher Zustände endlich groß ist.
Ein endlicher Automat ist ein erkennendes System für eine (formale) Sprache.
Ein endlicher Automat ist ein Akzeptor für eine (formale) Sprache.

Endliche Automaten vom Typ **Akzeptor** zerlegen Wörter. Ziel ist die Prüfung, ob eine Zeichen- / Symbol-Folge zur Sprache des Automaten gehört – nicht mehr und nicht weniger!

- ein Wort bildet die Eingabe
- Wort wird Symbol-weise zerlegt
- die Symbole (Eingaben) bestimmen darüber, ob der Zustand beibehalten oder gewechselt wird
- die Übergänge zwischen den Zutänden wird nur durch die Symbole (Eingaben) bestimmt

Der nebenstehende endliche Automat kann z.B. die folgenden Wörter akzeptieren:

hat
haben
hatte



Dagegen werden die Worte: ***du, informatik, ha, hattest*** und ***hab*** nicht akzeptiert.

Definition(en): Akzeptor
Ein Akzeptor ist ein endlicher Automat, der ohne Ausgabe auskommt.
Akzeptoren sind Beschreibungsmittel für formale Sprachen.

Aufgaben:

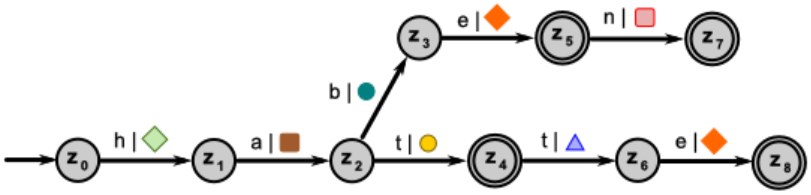
1. *Entscheiden Sie, ob die folgenden Worte (Eingaben) vom Automaten akzeptiert werden!*

a)	hit	b)	habe	c)	ha ben
d)	12345	e)	erhaben	f)	en

2. *Erstellen Sie den Automaten in einem geeigneten Simulations-Programm (Empfehlung: JFLAP → endlicher Automat (Finite Automaton))*
3. *Prüfen Sie die bei Augabe 1 entschiedenen Worte*
4. *Finden Sie einen Automaten, der eine möglichst große Menge von deutschen Wörtern akzeptiert! Gesucht ist der Automat mit den meisten akzeptierten Wörtern und der kleinsten Anzahl von Zuständen! (Die Maximalzahl von Zuständen wird auf 15 festgelegt!)*
5.

Eine weitaus umfangreichere Darstellung von Akzeptoren erfolgt im Abschnitt (→ 3.2.7. Automaten ohne Ausgabe – Akzeptoren).
Zuerst schauen wir uns aber die Transduktoren an
Transduktoren erzeugen Ausgaben (→ 3.2.6. Automaten mit Ausgaben - Transduktoren). Die Ausgaben sind von den Zuständen und / oder den Eingaben abhängig. Hierzu erfolgt später noch eine genauere Betrachtung (→ x.y.z.3. MOORE-Automaten bzw. x.y.z.4. MEALY-Automaten).

Der nebenstehend angegebene Transduktor arbeitet z.B. mit den gleichen Eingaben, wie der gerade besprochene Akzeptor. Die Ausgabe erfolgt immer bei Eingabe eines Zeichens und wird immer hinter einem senkrechten Strich angegeben.



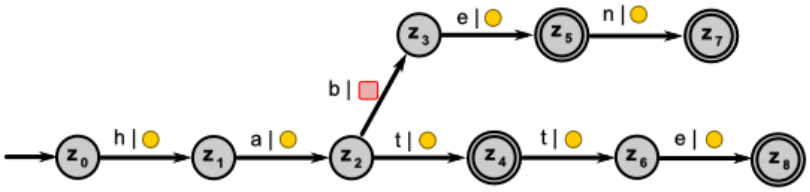
Wird also die Zeichen-Folge ***haben*** eingegeben, dann gibt der Automat ◇ ■ ● ◆ □ aus.
Auf den ersten Blick scheint der Automat auch fehlerhaft zu sein. Einmal gibt er bei Eingabe eines t ein ○ aus, im anderen Fall ein △. Dabei muss man aber beachten, dass die Ausgaben aus völlig anderen Zuständen heraus erfolgen. Der Automat kann also so korrekt sein. Das blaue Dreieck könnte ja auch als Buchstaben-Wiederholungs-Zeichen verstanden werden. Das kommt einfach auf die definierte Sprache an.

Aufgaben:

1. Prüfen Sie die folgenden Worte am angegebenen Automaten!

a)	hit	b)	habe	c)	ha ben
d)	12345	e)	erhaben	f)	en
g)	◇ ■ ● ◆ □	h)	◇ ■ hat	i)	● □ hatte ◇ ■ ●

2. Von einem Schüler wurde der nachfolgende Automat vorgeschlagen. Ist dieser korrekt? Begründen Sie Ihre Meinung!



3. WENN Sie sich bei Aufgabe 2 nicht für einen gültigen Automaten entschieden haben, DANN entwickeln Sie einen gültigen Automaten ANSONSTEN (Automat ist also gültig) geben Sie die Sprache des Automaten als Wort-Menge an!

Definition(en): Transduktor / Transducer
Ein Transduktor ist ein endlicher Automat, bei dem auch Ausgaben erfolgen.
Transduktoren sind endliche Automaten, die sowohl Lesen und Schreiben können.
Transduktoren sind Erstellungsmittel für formale Sprachen.

Als Gegen-Entwurf zu den endlichen Automaten könnte man die nicht-endlichen Automaten (→ x.y.z.1. Alternative: nicht-endliche Automaten?) machen. In dieser Form werden sie aber kaum betrachtet. Die hier gesammelten Automaten sind sehr vielgestaltig und auch durch sehr unterschiedliche Konzepte gekennzeichnet. Wir können aber Künstliche Neuronale Netze (KNN; → Künstliche Neuronale Netze) und PETRI-Netze (→ PETRI-Netze) dazu zählen. Sie nicht die primären Objekte der Schul-Informatik.

Definition(en): endliche Automatensprache
Eine endliche Automatensprache ist eine formale / künstliche Sprache, die ein endlicher Automat erkennen kann.

Aufgaben:

1.

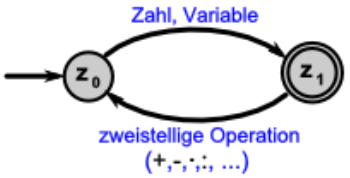
2.

x. Erstellen Sie den Zustands-Graphen für einen EA, der genau die folgenden Wörter mit dem Wortstamm "hör" erkennt! (Statt Buchstaben als Eingabe-Zeichen soll auf Morpheme (Silben u. Buchstaben-Gruppen) gesetzt werden!)

hören	zuhören	gehört	aufgehört	aufzuhören	aufhört	abhört
hört	zugehört	zuhört	aufhören	zuzuhören	abhören	abzuhören

x. Suchen Sie sich einen Wortstamm (auch für Substantive od.ä.) heraus und erstellen Sie eine Liste zu akzeptierender Wörter! Tauschen Sie mit einem Kurs-Teilnehmer Ihre Liste und erstellen Sie dazu einen passenden Endlichen Automaten!

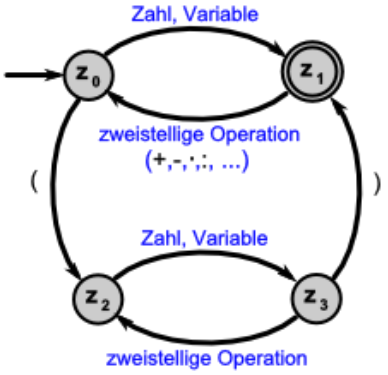
Endliche Automaten sind beschränkt, oder mit anderen Worten: sie haben Grenzen.
Schon an einem sehr einfachen Beispiel lässt sich zeigen, dass ein EA nur bestimmte (Größen-)Klassen von Sprachen bearbeiten kann.
Nehmen wir einen Automaten, der einfache Terme akzeptieren soll.
Den Automaten oben rechts (Automat K0) können wir mit den klassischen Termen wie:



Automat K0

49
a + 3
4 * b - 1

füttern und er akzeptiert diese.
Um Klammern mit auszuwerten, bedarf es schon eines erweiterten Automaten. Diesen sehen wir direkt rechts.



Automat K1

Aufgaben:

1. *Geben Sie drei mathematisch exakte Terme an, die der Automat K0 nicht akzeptiert!*
2. *Prüfen Sie, ob die folgenden Ausdrücke vom Automaten K1 akzeptiert werden!*

a)	23 * 7 + + 12	b)	4 * (3 + 5)	c)	5 (2 – 1)
d)	((23 – 6 : 3) : 7) + 5	e)	(x) – (a) * (b + 4)	f)	x * 4 = 3

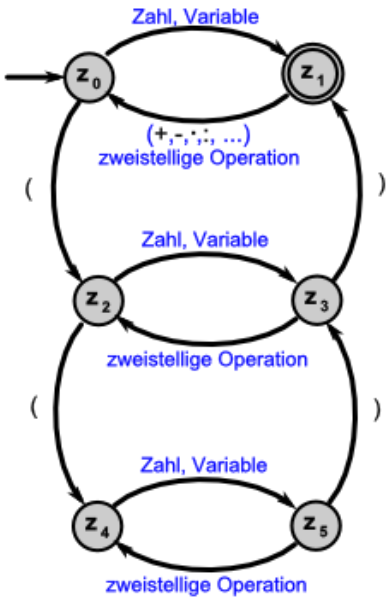
für die gehobene Anspruchsebene:

3. *Erweitern Sie den Automaten K1 so, dass er ein Gleichheitszeichen zwischen einem linken und einem rechten Term mit verarbeiten kann!*

Beim genaueren Betrachten / Ausprobieren fällt allerdings auf, dass er nur genau ein Klammer-Paar verarbeiten kann. Wollen wir nun ein zweites ermöglichen, wächst unser Automat um eine Ebene.

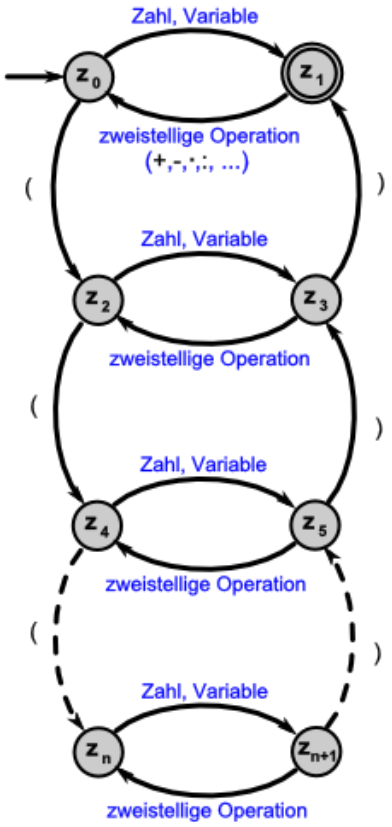
Aufgaben:

1. *Prüfen Sie die Ausdrücke von Aufgabe 2 des letzten Aufgaben-Blocks am Automaten K2!*
2. *Geben Sie drei mathematisch exakte Terme an, die der Automat K2 nicht akzeptiert!*



Automat K2

Um nun mehr Klammern in die Terme einzubauen, bräuchte man quasi einen unendlich tief gestaffelten Automaten. Dieser wäre aber bei den klassischen Aufgabenstellunegn mit meist nicht mehr als 4 bis 5 Klammer-Paaren deutlich überfordert.



Automat Kn

Aufgaben:

1. *Prüfen Sie ob Ihr Taschenrechner und die Taschenrechner-App Ihres Smartphone's oder Tablett's beliebig viele Klammer-Paare bearbeiten kann! Welchen Grenzen gibt es ev.?*
2. *Erstellen Sie einen Automaten, der Worte akzeptiert, die immer abwechselnd aus einen Buchstaben und einer Ziffer bestehen! Entscheiden Sie, womit angefangen werden soll und wie der Automat enden soll!*
3. *Gibt es eine Automat, der die gleiche Aufgabe – wie 2. – erfüllt, aber sowohl mal mit einer Ziffer und mal mit einem Buchstaben beginnen kann und auch auf beides enden kann? Begründen Sie Ihre Meinung!*

Man merkt hier schon, dass das Thema "Automaten" sehr differenziert betrachtet werden kann. Viele Automaten erscheinen oder sind sehr komplex und unübersichtlich. Es bietet sich also an, nach technischen Hilfsmitteln und Simulatoren zu fragen. Diese wollen wir hier auch vor der eigentlichen tieferen Besprechung der Automaten-Typen (Akzeptoren / Transduktoren) vorstellen (→ 3.2.5. Simulation endlicher Automaten). Sie können i.A. mit mehreren Automaten-Typen arbeiten. Ich möchte hier auch nicht unbedingt eine Software vorziehen. Alle haben Vor- und Nachteile. Eine Diskussion darüber ist hier nicht Ziel-führend. Beim Überblättern der nächsten Seiten werden Sie entweder die Ihnen zur verfügung gestellt Software wiederfinden oder bei freier Auswahl eine vorziehen. Sie selbst wissen meist am besten darüber Bescheid, was Ihnen liegt und was nicht.
Der nächste Abschnitt kann vom "Normal-Leser" beruhigt übersprungen werden (dann weiter bei: → 3.2.5. Simulation endlicher Automaten). Er ist mehr aus systematischen Gründen aufgenommen und soll einwenig die Leerstellen vieler klassischer Lehrbücher füllen – zumindestens ansatzweise.

3.2.4.1. Alternative: nicht-endliche Automaten?

Wenn es endliche Automaten gibt, gibt es dann auch nicht-endliche oder unendliche Automaten?
Das Kriterium für Endlichkeit wurde ja von uns über die Anzahl der Zustände festgelegt. Somit müssen nicht-endliche oder unendliche Automaten unendlich viele oder zumindestens nicht mehr abzählbar viele Zustände besitzen. Anders betrachtet, das Konzept der Zustände, so wie wir sie bisher betrachtet haben müsste aufgelöst sein. Zu solchen Zustandsunbestimmten Automaten kann man die PETRI-Netze (→ PETRI-Netze) und die Künstlichen Neuronalen Netze zählen. Bei den Künstlichen Neuronalen Netzen (KNN → Künstliche Neuronale Netze) wird das Fehlen von definierten Zustände recht deutlich. Hier können bestimmte Teile des "Automaten" beliebig viele – sprich analoge – Zustände einnehmen.
PETRI-Netze haben eine andere Modell-Struktur, die Zustände im Sinne der klassischen TI nicht enthalten. Als Äquivalent gibt es hier Plätze bzw. Stellen.

Definition(en): nicht-endlicher Automat
Ein nicht-endlicher / unendlicher Automat ist ein Automat mit einer nicht genau definierbaren / abzählberen Menge an Zuständen.

PETRI-Netze

Carl Adam PETRI beschäftige sich akademisch mit der Kommunikation von Automaten. Dabei setzte er sich verstärkt mit verteilten und nebenläufigen Prozessen auseinander. Im Rahmen dieser Forschung entstand das Konzept der später nach ihm benannten PETRI-Netze.

In der graphischen Darstellung von PETRI-Netzen findet man Plätze bzw. Stellen (mit Kreisen gekennzeichnet) und Transitionen (Rechtecke). Plätze stellen dabei die Objekte / Gegenstände (Substantive) dar, mit den Transtionen sind Methoden oder Tätigkeiten (Verben).

Soll die Methode nicht näher benannt werden, dann kann man das rechteck auch auf einen senkrechten Strich reduzieren.

Die Plätze stellen das Äquivalent zu Zuständen in Zustands-Automaten dar. Wir werden sehen einige der Aspekte von PETRI-Plätzen ähneln sehr stark den Zuständen.

Bestimmte Plätze können gelabelt (durch Punkt (Token (Marken)) gekennzeichnet) sein. Das bedeutet, dass sie quasi einen Anfangs-Platz bzw. einen / den aktuellen Platz darstellen. Ein Platz kann auch mehrere – auch unabhängige – Token besitzen.

Die Verbindung (mit einem Pfeil; Kante des Netzes) von einem Platz zu einer Transition wird als Konsumierung (auch: Konsum(p)tion) des Platzes durch die Transition verstanden. Eine Produktion (eines Platzes) ist dagegen die Verbindung einer Transition mit eben dem (zu produzierenden) Platz.

Die Kanten / Verbindungen eines PETRI-Netzes werden auch Fluss-Relationen genannt.

Ein PETRI-Netz kann damit über die drei Komponenten:

- (nicht-leere) Menge der Plätze P
- (nicht-leere)Menge der Transitionen T

und die

- (nicht-leere)Menge der Fluss-Relationen (Kanten) F

definiert werden. Somit ist ein PETRI-Netz N=(P,T,F).

Definition(en): PETRI-Netz

Ein PETRI-Netz N ist ein Tripel (3-Tupel) ***N = (P, T, F)*** mit:

P… nicht-leere Menge der Plätze
T… nicht-leere Menge der Transitionen
F… nicht-leere Menge der Kanten / Fluss-Relationen $F \subseteq (P \times T) \cup (T \times P)$

wobei die Mengen P und T disjunkt sind (überschneiden sich nicht).

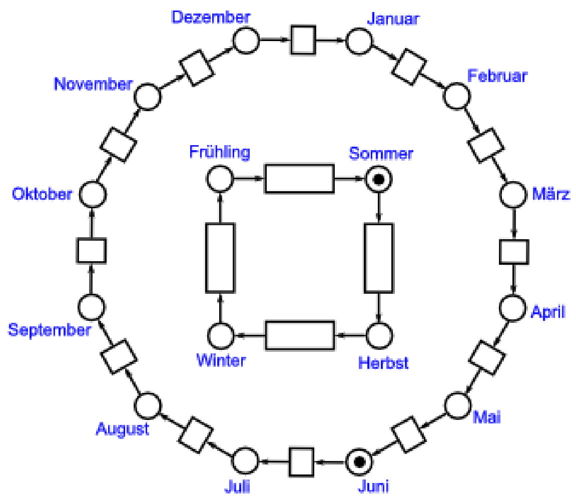
Es gibt auch Definitionen als Quad-Tupel N = (A, P, f, M_0). Sie beinhalten die Elemente

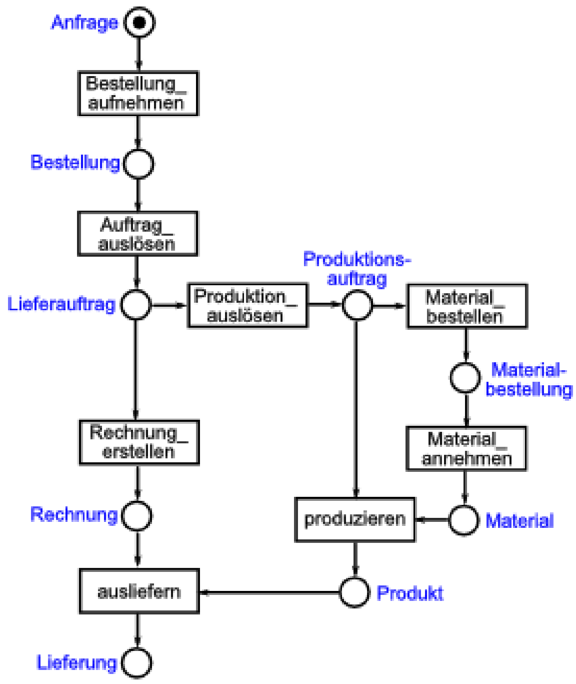
- das nicht-leere Kommunikations-Alphabet A
- eine nicht-leere Menge von Plätzen P
- die Transitions-Relation f

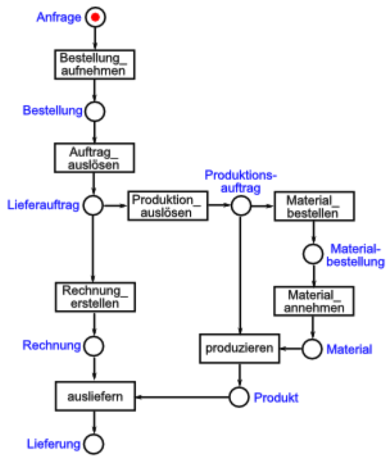
und

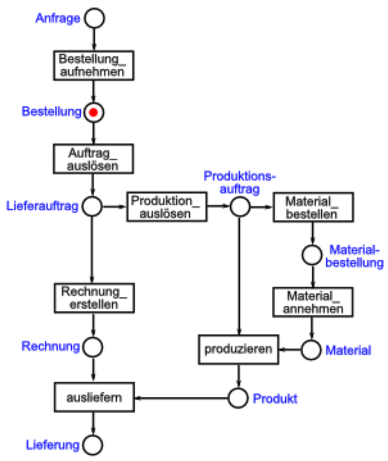
- die Anfangs-Markierung M_0

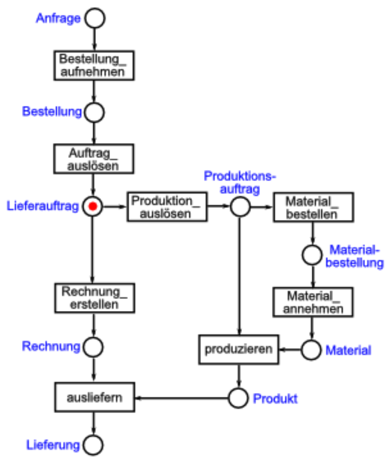
Ein schönes Beispiel für Nebenläufigkeit sind die Zyklen für die Jahreszeiten und die Monate eines Jahres.
Es gibt zwar bestimmte Synchronisationen bzw. Abhängigkeiten, aber die sehen auf der Südhalbkugel schon wieder ganz anders aus.
Aus der Wirtschaft kommt ein anderes anschauliches Beispiel. Der Prozess von einem Bedarf zu einer gelieferten Ware kann schon ein recht komplexes Netzwerk ergeben.

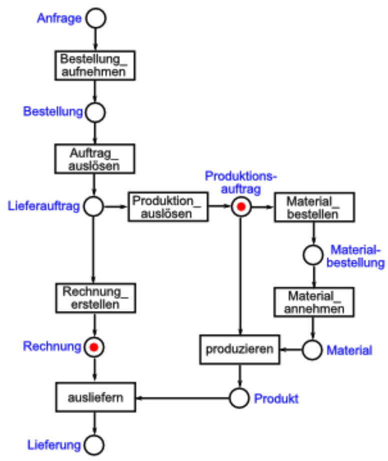


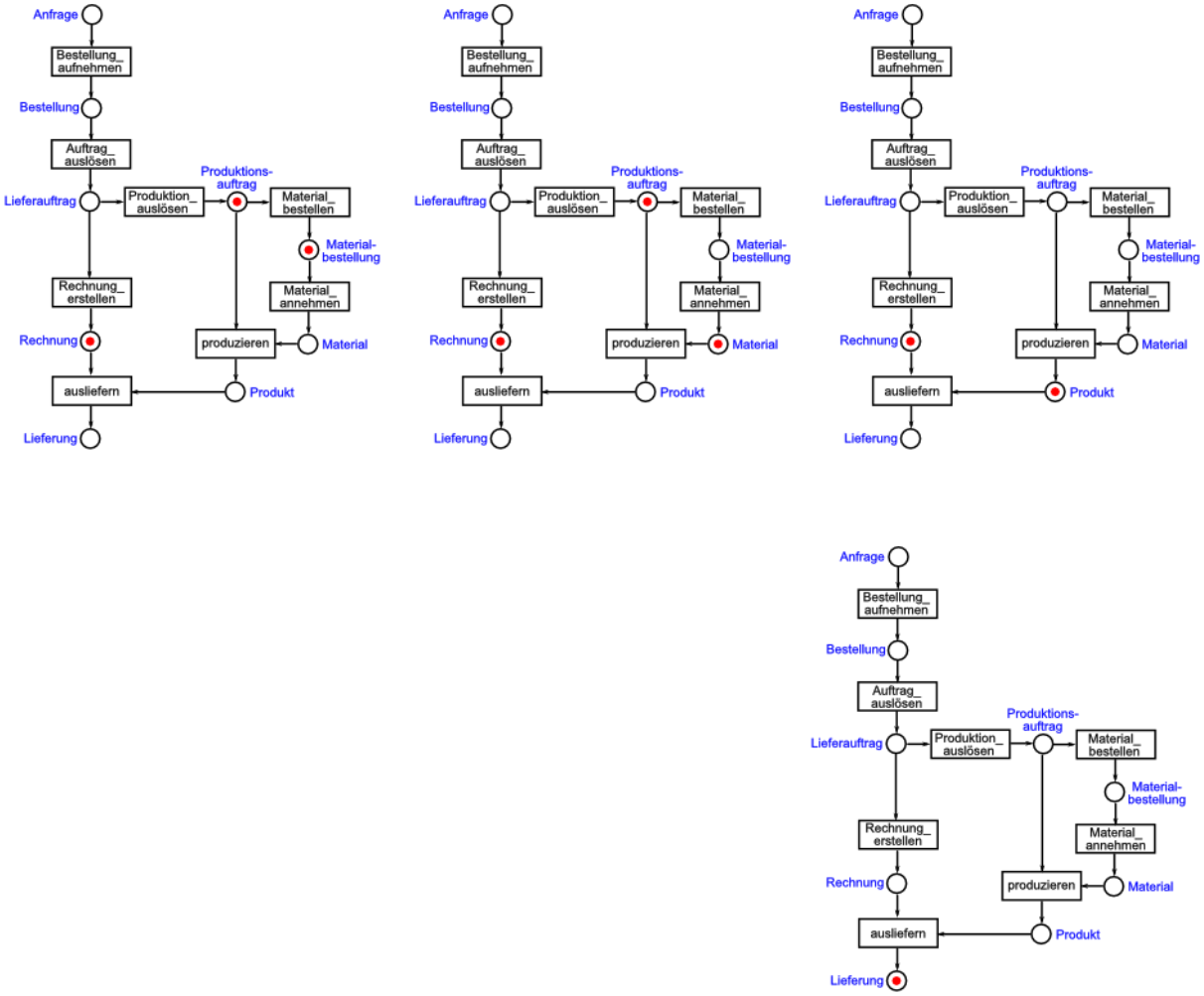






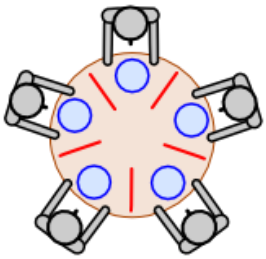




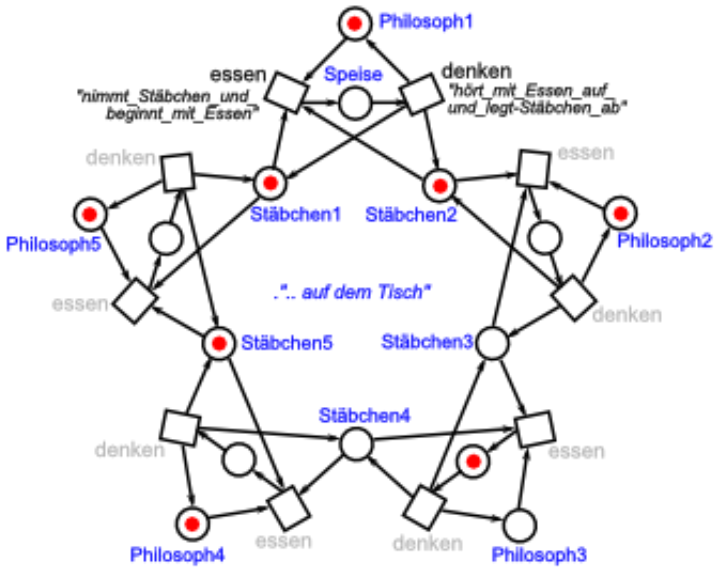


PETRI-Netze und Automaten ohne definierte Zustände sind insgesamt auch als Rekursive Transitions-Netzwerke (RTN) zu fassen. Sie reagieren kontinuierlich auf die Umgebung. Transitions-Netze können auch als gerichtete Graphen dargestellt werden, wobei Kanten und Knoten unterschiedliche Bedeutungen haben können.

PETRI-Netz zum Problem der fünf speisenden Philosophen



An einem runden Tisch sitzen 5 Philosophen. Vor ihnen steht jeweils ein Schüsselchen. Dazwischen liegt immer jeweils ein Stäbchen. Zum Essen braucht ein Philosoph – wie jeder andere Mensch – genau zwei Stäbchen (sein "Eigenes" und eins vom Nachbarn). Wie können sich die Philosophen das essen organisieren, damit alle möglichst gleich satt werden können? (Von Hygenie haben die Philosophen damals noch nicht verstanden!)
Da bei fünf Stäbchen immer nur maximal zwei Philosophen essen können, bleibt für die anderen Zeit zum Denken.
Als Strategie könnten sich die Philosophen verabreden, dass jeder zuerst nur die Stäbchen rechts nimmt und wenn die anderen (links) daliegen, dann dürfen diese benutzt werden.
Nach dem Essen werden die Stäbchen wieder hingelegt und der nächste kann sich bedienen oder auch der Philosoph selbst kann ev. wieder zugreifen (wenn er denn zwei Stäbchen greifen kann).
Führen alle fünf Philosophen die obige Startegie gleichzeitig aus, dann kann keiner speisen. Es würde zu einer Verklemmung kommen. Da es aber praktisch nie ein echtes Gleichzeitig gibt, wird der eine oder andere Philosoph (schnell) zwei Stäbchen greifen können.
Als PETRI-Netz dargestellt, könnte das Problem so aussehen.
Es wandern mehrere Token durch das Netz. Um zu Schalten (/ eine Methode auszuführen), müssen immer alle Eingänge einen Token haben.
Ist das z.B. für einen Philosophen so , dann kann die Transition "essen" schalten und "Speise" erhält den Token (Abb. 1 (unten)).
Da "Speise" der einzige Eingang von "denken" ist, kann diese Transition nun schalten und der Token von "Speise" verteilt sich auf die "Stäbchen" und den "Philosoph"en (Abb. 2).
Es erhalten alle Ausgänge einen Token, wenn eine Methode ausgeführt / geschaltet wurde.
Ein einzelner Philosoph hätte also gar keine Probleme, er würde immer abwechselnd "essen" und "denken".



nach Q: https://kogs-www.informatik.uni-hamburg.de/~neumann/P3-WS-2000/P3-Teil3.pdf (leicht geändert)

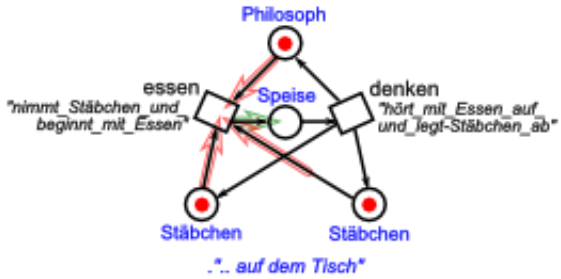


Abb. 1: Schalten von "essen"

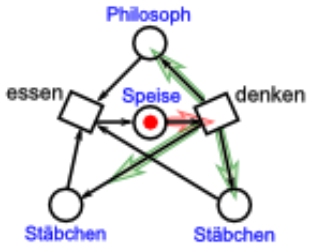


Abb. 2: Schalten von "denken"

Sitzen nun mehrere Philosophen am Tisch, dann gibt es auch andere Situationen im Netzwerk. Die folgenden verhindern ein Schalten von "essen".

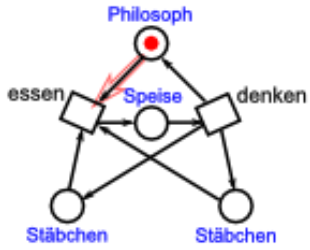


Abb. 3: kein Stäbchen

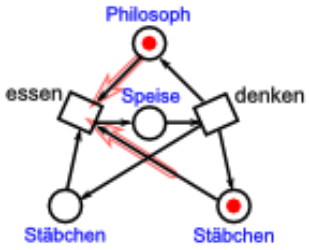


Abb. 4: nur 1 Stäbchen (rechts)

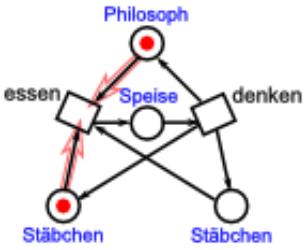
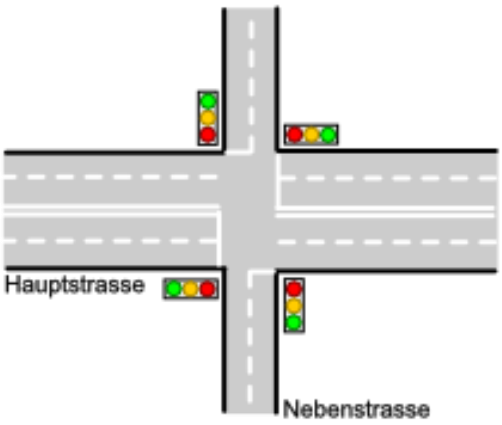


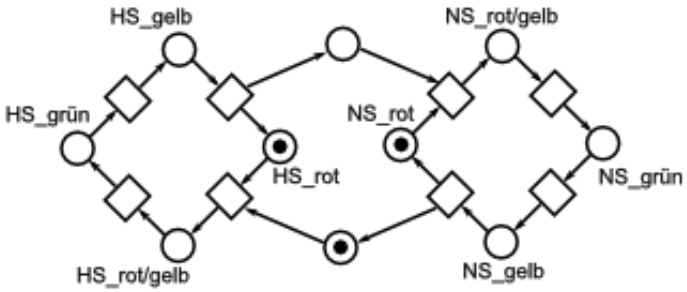
Abb. 5: nur 1 Stäbchen (links)

kleines (unvollständiges) Beispiel: Ampel-Steuerung:

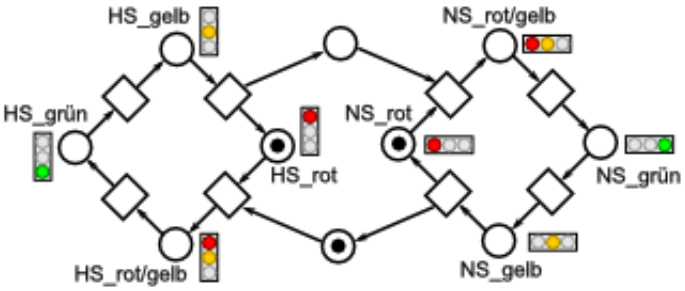
Zwei sich kreuzende Straßen werden von einer Ampel reguliert. Natürlich dürfen die beiden Ampel-Richtungen (Hauptstraße und Nebenstraße) nicht gleichzeitig grün zeigen.
Jede Ampel-Richtung ist durch einen definierten Phasen-Wechsel bestimmt. Ausgehend von "rot" folgt "rot/gelb" und dann die Freigabe mit "grün". Die Vorbereitung zur Sperrung erfolgt mit "gelb" und dann beginnt wieder alles von vorne. Die Dauer der Phasen werden z.B. durch die Transitionen bestimmt.
Für die zweite Straße gibt es ein äquivalentes System. Dieses muss nur Phasen-verschoben arbeiten bzw. zu bestimmten Phasen (Stellen) des anderen Systems blockieren.



Das PETRI-Netz für eine – sich gegenseitig blockierende – Steuerung könnte etwa so aussehen:
Jede Ampel-Richtung wird durch kleines PETRI-Netz veranschaulicht. Im nebenstehenden Graphen sind das die rhomischen Strukturen rechts und links. Das rechte Netz steht für die Hauptstraße (HS), das andere für die Nebenstrasse (NS).



Um einen gleich-phasigen Lauf zu verhindern, müssen die einzelnen Ampel-Richtungen, die durch jeweils ein kleines PETRI-Netz gesteuert sind, über spezielle Plätze miteinander gekoppelt werden. Das sind die zwei (Brücken-)Plätze zwischen den Rhomben.
Das Beispiel zeigt auch sehr schön die Begriffs-Beziehung zu "Netzen" – schon bei diesem einfachen Beispiel kommt es zu einer hohen Vermaschung.
Für ein besseres Verständnis sind im nebenstehenden PETRI-Netz die jeweiligen Ampel-Phasen auch graphisch dargestellt. Man kann dann auch gleich erkennen, wann die eine Steuerung auf die andere einwirkt.



In erweiterten Versionen könnten dann auch noch Induktions-Schleifen im Straßenbelag oder Bewegungsmelder ausgewertet werden. Aus Übersichtsgründen habe ich auf die Bezeichnung der Transitionen und einiger Plätze verzichtet. Das PETRI-Netz müsste also noch ein wenig erweitert werden.

Aufgaben (für die gehobene Anspruchsebene)

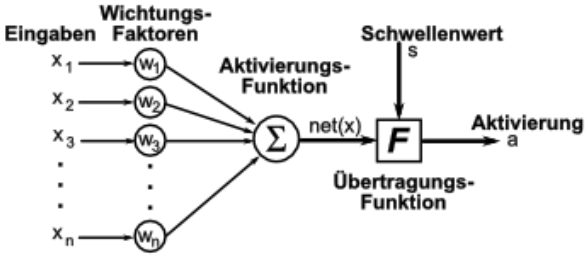
1. ***Verändern Sie das obige PETRI-Netz so, dass statt einer Nebenstraße eine Fussgänger-Ampel gesteuert wird!***
2. ***Erstellen Sie ein PETRI-Netz für eine Ampel-Steuerung, bei der ein Bedarf für die Nebenstraße durch dort eingelassene Induktions-Schleifen angemeldet wird!***

Künstliche Neuronale Netze

Bei den Künstlichen Neuronalen Netzen (Abk.: KNN; eng.: artificial neural network (ANN)) handelt es sich um ganze Klasse unterschiedlichster Netz-artiger Strukturen bzw. logischer Maschinen mit einer starken Ausrichtung in Richtung Fuzzy-Logik.
Künstliche Neuronale Netze stellen eine eigene Maschinen- bzw. Automaten-Welt. Der Begriff Universum würde es wahrscheinlich besser umschreiben.
Bei den KNN's handelt es sich quasi um ein Bionik-Projekt, bei dem die Funktionen von Nerven-Zellen und höheren neuronalen Strukturen (wie z.B. Rückemark, Gehirn) nachgebildet werden sollen. Zum Einen möchte man natürlich verstehen lernen, wie Nerven-Zellen und Gehirn solche herausragenden Leistungen vollbringen. Und zum Anderen geht es natürlich um die Schaffung von Denk-Maschinen, die eben genau solche Leistungen hervorbringen können.
Heute werden KNN z.B. in der Muster-Erkennung (Schrift- und Sprach-Erkennung, in Viren- und Spam-Scannern, Frühwarnsysteme, Müllsortierung, Data-Mining (Big Data), Überwachung von Aktien-Kursen), für die Roboter-Steuerung, die Erstellung virtueller Agenten usw. eingesetzt.

Definition(en): Künstliche Neuronale Netze (KNN)
Künstliche Neuronale Netze sind verknüpfte Mengen / Gruppen von Informations-Verarbeitungs-Einheiten (künstliche Neuronen / Neuronen-Modelle).
Künstliche Neuronale Netze sind selbstständig lernende Computer-Programme, die große Daten-Mengen mit (versteckten) Zusammenhängen analysieren und bewerten können.
Künstliche Neuronale Netzwerke sind Netze aus künstlichen Neuronen.

Ein wichtiger Grund-Baustein in den KNN ist das **Perceptron** (Perzeptron; Muster-Erkenner). Es entspricht modellhaft einer Nerven-Zelle.
Über die Dendriten werden die Eingaben von anderen Nerven-Zellen oder Sinnes-Zellen übernommen. Häufige Verwendung und positive Rückkopplungen erzeugen festere Verbindungen zwischen den Dendriten und den Informations-Quellen. Im Perceptron wird dieser Effekt durch die Wichtungs-Faktoren nachempfunden.



Gehen (bei einer Nerven-Zelle) genug Eingangs-Informationen ein und übersteigen diese insgesamt einen Schwellenwert, dann löst die Zelle ein Aktions-Potential aus, welches an nachfolgenden Nerven-Zellen und / oder andere wirkende Systeme (Muskeln, Drüsen) übertragen wird. Beim Peceptron haben wir dann eine Aktivierung zu verzeichnen. Sie ergibt sich aus der Summe der Eingänge, deren Wichtungs-Faktoren und der Aktivierungs-Funktion.
Die Charakterisierung der Natürlichen und Künstlichen Neuronalen Netze erfolgt auch über die hochgradig parallel arbeitenden Komponenten (z.B. eben die verschiedenen Neuronen).
Die Ableitungs- / Überführungs-Funktionen von Künstlichen Neuronen und ihren Netzwerken sind i.A. nicht-linear.

Mit KNN ist auch das Anlernen des System verbunden. Die einzelnen Wichtungs-Faktoren müssen so eingestellt werden, dass das gewünschte Ergebnis – also z.B. die Erkennung des Buchstaben "A" möglichst Fehler-arm funktioniert.
In erweiterten KNN kommen dann automatische Rückkopplungs-Strukturen usw. mit dazu, die dem System ein eigenständiges Lernen ermöglichen soll.
Nach eine großen euphorischen Phase hat sich nun etwas Normalität in der Welt der KNN eingestellt. Die Technologie hat große Potentiale, aber die ersten Erfolge waren wohl Grund für völlig übertriebene Erwartungen.
Ein große Bedeutung kommen den KNN bei der Lösung mathematischer Aufgaben (Primzahlen-Bestimmung, Faktorisierung von Zahlen, Vorhersage von (Pseudo-)Zufalls-Zahlen) und bei der Dechiffrierung verschlüselter Nachrichten zu.
Die klassische Verwendung von KNN findet man bei bei Klassifikations-, Optimierungs- und Prognose-Problemen. Heute werden sich auch immer häufiger im Betriebs- und Volkswirtschaftlichem Kontext verwendet. Dort z.B. zur Prognostizierung von Aktionkursen oder der Manipulation, Absatz- und Umsatz-Prognosen, Analyse und Prognose von Maßnahmen (Zins-Veränderungen, Steuern, Stützen, …).
Die Prüfung der Kreditwürdigkeit oder die Vorhersage von kriminellen Handlungen an bestimmten Orten zu irgendwelchen Zeiten gehören ebenfalls dazu. Etwas neuer ist die Anwendung ind der Produktions- und Personal-Planung.

Kleine Anekdote aus der Welt der Künstlichen Neuronalen Netze:
Eine Armee will ein Panzer-Erkennungs-System für den Einsatz im Kampfgebiet haben. Dazu sollten KNN eigentlich optimal geeignet sein. Also zeigt man einem teuer entwickelten KNN (Werbe-)Fotos mit Panzern in allen Lagen und solche ohne Panzer.
Beim ersten Praxiseinsatz des System auf einem Truppenübungsplatz bei schönem Wetter schlägt das System plötzlich immerzu Alarm, obwohl gar keine Panzer da sind.
Was war passiert? Das System bekam neben den tollen Panzer-Fotos als Gegenstück Fotos von Geländen bei trüben Tagen zum Lernen. Letztendlich hatte das teure KNN nur gelernt, schönes Wetter von solchem mit mieser Stimmung zu unterscheiden.

ausführlich im Skript "Projekt Deep Learning"

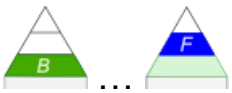
kleines (unvollständiges) Beispiel: Buchstaben-Erkennung:

100-Schritt-Regel:

Diese Regel soll die Leistungs-Fähigkeit von Neuronalen Netzen belegen.

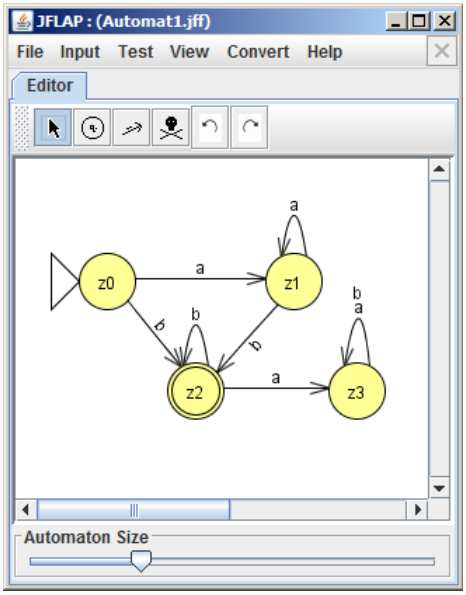
Ein menschliches Gehin kann ein bekanntes Bild / einen bekannten Gegenstand / eine bekannte Person in ungefähr 0,1 s erkennen. Dazu sind rund 100 sequentielle Verarbeits-Schritte notwendig.
In der Digital-Technik (moderne Computer) führen 100 Arbeits-Schritte (Assembler-Befehle) zu praktisch garnichts.

3.2.5. Simulation endlicher Automaten



Das Internet bietet mehrere verschiedene Simulations-Programme für den Bereich Automaten-Theorie. Uns interessiert hier vorrangig das Angebot an freier Software. Einige der Programme stelle ich nachfolgend vor.
Auf der Abitur-Version des Io-Stick's sind die Programme **JFLAP**, **AutoEdit** und **Grammatik-Editor** quasi als portable Apps verfügbar gemacht.
Persönlich würde ich – trotz der nur im englischen verfügbaren Sprach-Version – JFLAP favorisieren. Es bietet unter einer Oberfläche viele Funktionen, die uns im Unterricht interessieren und es kann auch mit allen üblichen Automaten umgehen.
Schauen Sie sich die Möglichkeiten der Programme einfach an und entscheiden Sie dann nach Ihren Bedürfnissen.
Als JAVA-Programm sollte JFLAP auch auf den verschiedensten Betriebssystemen laufen.
Die JFF-Dateien sind praktisch XML-Dateien. Sie lassen sich auch in einem Text-Editor erstellen und verändern.

3.2.5.1. Simulation endlicher Automaten mit JFLAP

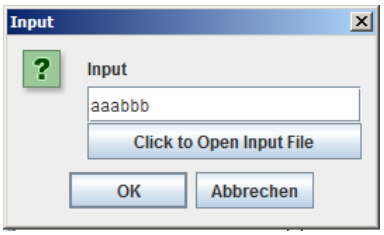


JFLAP (**J**ava **F**ormal **L**anguages **A**utomata **P**ackage) ist ein Editier- und Simulations-Programm für viele Automaten-Modelle. Die aktuelle Programm-Version kann unter jflap.org runtergeladen werden. Die Io-Stick-Nutzer finden das Programm in der Abi-Version.
Im Editor lassen sich Zustände (2. Editor-Schaltfläche) definieren.
Vorher wählt man die Art des Automaten aus (File" "New …"). Die meisten Automaten bzw. Modelle sind namentlich gut erkennbar. Ansonsten hilft die weiter unten folgende Tabelle weiter.
Die Datei sollte natürlich an dieser Stelle – wie üblich – schon mal angelegt und vorgespeichert ("File" "Save As …") werden.
Es empfiehlt sich auch das übliche Speichern ("File" "Save") vor jedem neuen Schritt bzw. nach abgeschlosenen Arbeits-Schritten.
Die Umbenennung von Zuständen – auf z.B. die in diesem Skript benutzten z-Buchstaben-Namen – lassen sich über das Kontext-Menü ändern. Zusätzlich sind Labels ("Beschreibungen") möglich.
Wenn bestimmte Kontext-Elemente nicht erreichbar sind, oder ein Doppel-Klick nicht das angeklickte Objekt erweitert, dann sollte man den aktuellen Editor-Modus prüfen. In den mesiten Fällen bringt das Einschalten des Zeige-Modus (Schaltfläche [↖]) den gewollten Effekt.
Einstellung von Start- ("Initial") und End-Zuständen ("Final") lassen sich hier ebenfalls erledigen. Die Übergänge zwischen den Zuständen werden ganz intuitiv durch Ziehen vom Ausgangs-Zustand zum Ziel-Zustand realisiert. Notwendige Angaben – z.B. die Eingabe(n) – sollten immer gleich ordentlich erfolgen. Benötigt man zwei Eingabe-Symbole an einem Übergang, dann wiederholt man einfach die Erstellung des Übergangs. Das zweite Symbol wird dann oberhalb notiert (siehe z.B. Übergang von z3 nach z3).

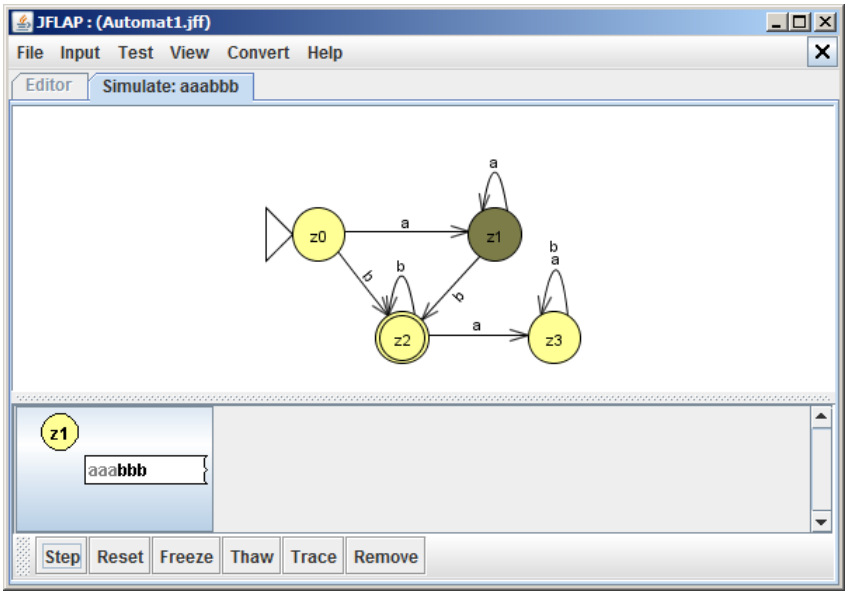
Für manche Automaten werden Mehrzeichen-Eingaben gebraucht. Damit dies später reibungslos funktioniert sollten die Eingaben auch schon bei den Übergängen in Anführungs-Zeichen ("…") gesetzt werden.

Automat / Modell in JFLAP	deutscher Name	Hinweise / Bemerkungen
Finite Automaton	endlicher Automat	
Mealy Machine	MEALY-Automat	werden als deterministisch betrachtet
Moore Machine	MOORE-Automat	
Pushdown Automaton	Keller-Automat	
Turing Machine	TURING-Maschine	
Multi-Tape Turing Machine	Mehr-Band-TURING-Maschine	
Grammar	Grammatik	
L-System	lambda-System	
Regular Expression	reguläre Ausdrücke	
Regular Pumping Lemma	reguläres Pumping-Lemma	
Context-Free Pumping Lemma	Kontext-freies Pumping-Lemma	

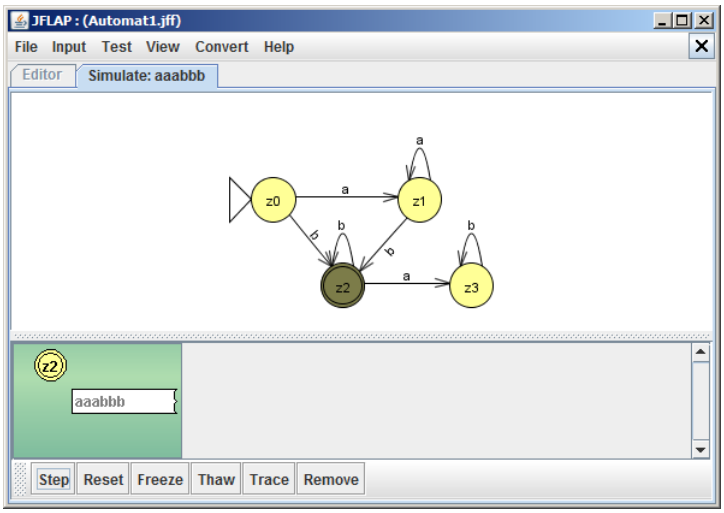
Nach Rückschritten (vorletzte Schaltfläche) muss einmal auf die Editor-Fläche geklickt werden, damit die Änderung angezeigt wird.
Bleibt nur noch der Totenkopf als Schaltflächen-Symbol. Mit dieser Schaltfläche wird einfach nur der Lösch-Modus aktiviert.
Mit einem fertigen Automaten lassen sich nun verschiedenste Operationen ausführen. Zu den typischen Dingen gehören sich die "Eingabe"-Tests. Im "Input"-Menü kann man nun auf verschiedene Art und Weise Eingaben testen.

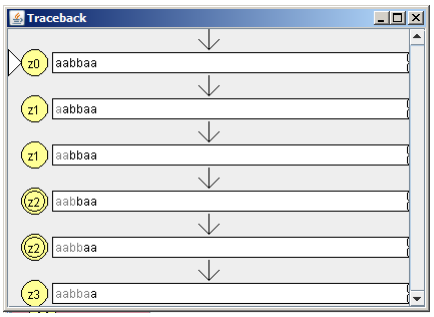


Bei manchen Automaten wird vor dem Testen auf Nicht-Determinismen geprüft. Das ist z.B. bei MEALY- und MOORE-Automaten so, obwohl diese zwangsläufig gar nicht immer deterministisch sein müssen. Im Programm JFLAP kann man jedenfalls nur mit den deterministischen Versionen arbeiten. Anders herum ist es natürlich eine gute Kontrolle, ob der Automat zumindestens hinsichtlich dieses Aspektes in Ordnung ist.
Eine explizite Testung auf Nicht-Determinismus ist unter "Test" "Highlight Nondeterminism" möglich. Dabei sollte man darauf achten, dass kein Zustand ausgewählt ist, sonst wird dieser auch weiterhin markiert angezeigt. Die problematischen ("nicht-deterministischen") Zustände werden ebenfalls markiert.
Unter **"Step with Closure …"** wird eine Eingabe (Text oder Datei) schrittweise auf Abschluss bzw. Geschlossenheit geprüft.
Mit "Step" geht es immer schrittweise durch die Eingabe-Sequenz. Was von dieser schon abgearbeitet wurde, wird grau angezeigt.
Schon bei der Eingabe muss man beachten, dass die Eingaben ohne Komma, Leerzeichen od.ä. eingegeben werden müssen. Auch Mehr-Symbol-Eingaben – wie z.B. eine "20" und eine "10" für einen Geld-Automaten müssen als"20""10" notiert werden!



Mit "Reset" kann man wieder von vorne beginnen.
Wird die Eingabe-Sequenz akzeptiert, dann erscheint die Band-Anzeige mit grünlichem Hintergrund (folgende Abb. links). Hierbei handelt es sich um ein Arbeits-Protokoll des Automaten.





Protokoll-Ansicht

Die rechte obere Abbildung zeigt die Verfolgung der Übergänge im "Trace"-Modus. Dazu muss aber in der unteren Band-Anzeige eine Situation ausgewählt werden.

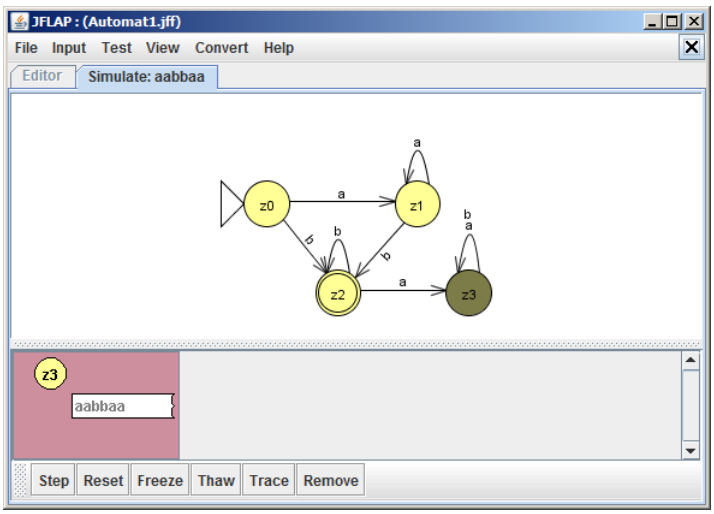
Nicht akzeptierte Eingaben werden durch einen rötlichen Hintergrund angezeigt.

Der aktuelle Simulations-Lauf lässt sich immer über die "Schließen"-Schaltfläche in der Menü-Leiste beenden (das untere (etwas größere) "Schließen"-Symbol).

Eine weitere Möglichkeit der Simulations-Kontrolle ist der Modus "Step by State …".

Der Eingabe-Test mit "Fast Run …" erzeugt sofort ein vollständiges Protokoll a'la "Trace".

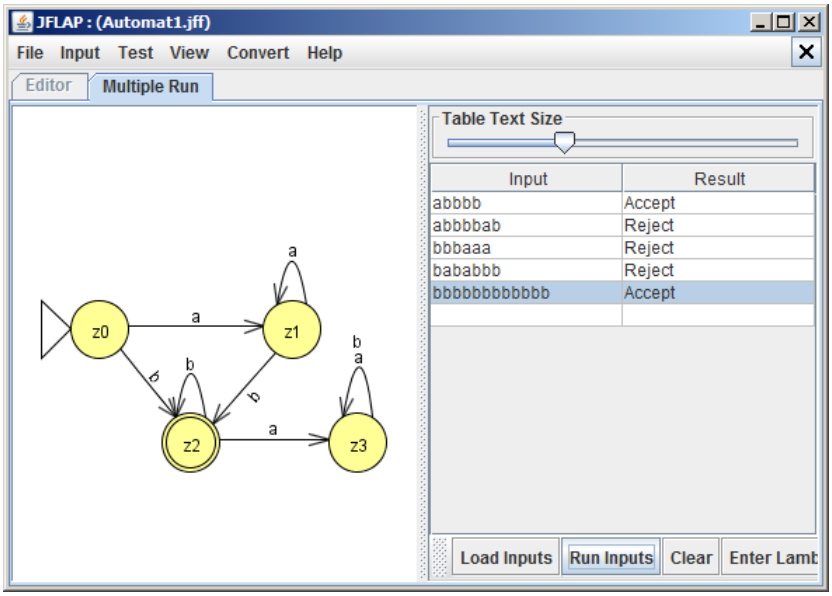
Im "input"-Bereich von "Multiple Run" lassen sich mehrere Texte / Eingaben gleichzeitig in einem Ritt prüfen.



Dazu füllt man zuerst die Input-Felder der rechten Tabellen-Spalte aus. Fehlende Zeile werden mittels [Tab]-Taste ergänzt. Danach können alle Eingaben mit "Rub Inputs" auf Akzeptanz geprüft werden.

Für die Tests kann man sich immer Dateien mit Eingabe-Sequenzen erstellen und diese dann ohne großes Eintippen verwenden.

Besonders bei den "Mehrfach-Läufen" oder sehr langen Eingaben ist



das praktisch.
Die Datei-basierten Mehrfach-Tests eignen sich auch sehr gut für die Prüfung von verschiedenen Maschinen / Automaten auf äquivalentes Verhalten.
Weitere "Test"s beziehen sich auf die Modelle selbst. Interessant ist für uns der Test von Nicht-Determinismen ("Test" "Highlight Nondeterminism") und der Äquivalenz-Vergleich ("Test" "Compare Equivalence") mit einem anderen geöffneten / geladenen Automaten.
In vielen Sprachen und Automaten sind leere Worte als Eingabe nicht gewollt oder zulässig. Mit der Programm-Funktion "Highlight λ-Transitions" im "Test"-Menü kann man sich die "leeren" Transitionen anzeigen lassen.
Zwischen λ und ε als leeres Zeichen kann man in den Einstellungen umschalten.
Ein Automat kann von JFLAP auch in eine Grammatik umgesetzt werden. Das lässt sich gut nutzen, wenn man ursprünglich eine Grammatik für die Konstruktion des Automaten verwendet hat. Man kann dann die vom Programm erzeugte Grammatik mit der ursprünglichen vergleichen.
Auf der Web-Seite zu JFLAP befindet sich – quasi als Hilfe – ein ausführliches – gut verständliches – Tutorial. Leider ist es in englischer Sprache, aber man kann es auch mit geringen Sprach-Kenntnissen gut nachvollziehen.

Links:
http://jflap.org (Webseite zum Programm)
http://tinohempel.de/info/info/IoStick/index.html (Webseite zum Io-Stick)

Aufgaben:

1. ***Bilden Sie mit JFLAP den 50-Cent-Cola-Automaten nach! Verändern Sie das Eingabe-Alphabet so, dass nur noch die erste Ziffer des Geldstücks als Eingabe verwendet wird (JFLAP kann in der normalen Notierung nur einzelne Zeichen als Eingabe auswerten!)! Testen Sie verschiedene Eingaben am angepassten Automaten (über "Input" + "Step by State …")!***
   (Alternativ können Sie alle Eingaben an den Übergängen in Anführungszeichen setzen. Auch bei den Tests sind die Zeichen-Gruppen jeweils in Anführungszeichen zu setzen. Leerzeichen zwischen Eingaben sind nicht zulässig!)
2. ***Dokumentieren Sie mit einfachen Bildschirm-Kopien den Graphen des Automaten und erzeugte Zustands-Protokolle (mittels "Input" + "Fast run …") für die nachfolgenden Eingaben!***

a)	5	b)	222	c)	1211
d)	32	e)	11111	f)	25
g)	1211	h)	55	i)	1112

3. ***Erstellen Sie mit JFLAP den 50-Cent-Cola-Automaten als Deterministischen Endlichen Automaten (DEA)!***
4. ***Entwickeln Sie mit JFLAP einen NEA, der Wörter aus einer Sprache akzeptiert, die mit mindestens 3x "a" beginnen und dann mindestens 2x "b" und 2x "c" folgen! Dokumentieren Sie mit einfachen Bildschirm-Kopien den Automaten und mindestens 6 Test's mit akzeptablen und nicht-akzeptablen Wörtern!***
5. ***Gesucht ist ein DEA, der Einsen im Bereich von 0 bis 9 zählt. Das "Zählwort" beginnt immer mit "1010" gefolgt von 0 bis 9 Einsen.***
   Beispiel: 101011111 für das Zähl-Ergebnis "5"
   Dokumentieren Sie Automat und Test's!
6. ***Erstellen Sie einen Automaten, der die Wörter mit "hör" oder eine entsprechende von Ihnen definierte Wörtergruppe (/ Wortstamm) akzeptiert (s.a. Aufgabe unter → 3.2.4.)!***

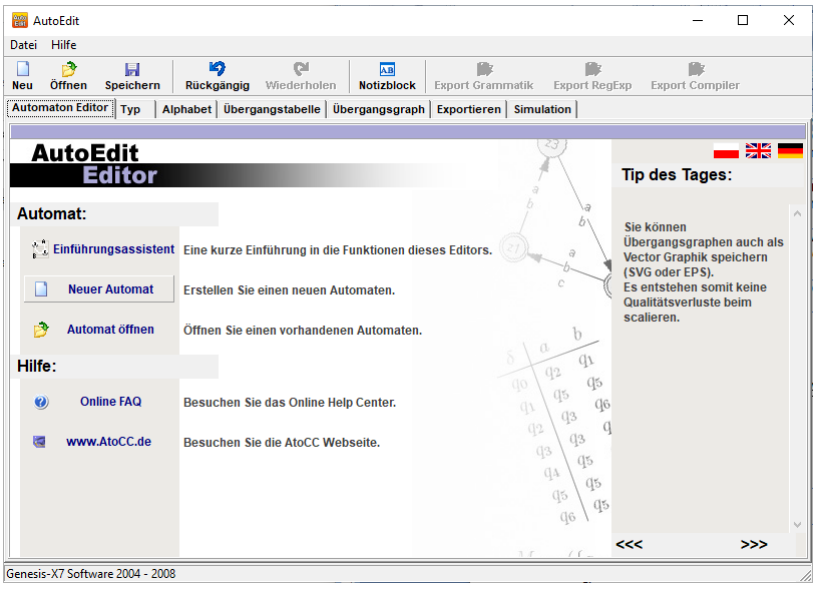
3.2.5.2. Simulationen von EA's mit der TI-Suite AtoCC

AtoCC (from **A**utomaton **to** **C**ompiler **C**onstraction) kommt der Eier-legenden Woll-Milch-Sau für den Unterricht zu Automaten und Grammatiken schon sehr nahe. Natürlich erhebt das Programm nicht diesen Anspruch, aber der Leistungs-Umfang der Suite ist schon umwerfend. Die Mächtigkeit erschwert aber auch einen einfachen, schnellen Einstieg als Nebenbei-Werkzeug.
Für unsere Zwecke ist zuerst einmal der Automaten-Editor **AutoEdit** interessant.
Weitere Komponenten der Suite sind:

- **AutoEdit Workbook** — Autoren-System für Aufgaben zu Automaten, Grammatiken, … mittels AutoEdit
- **kfG Edit** — Editor für die Erstellung und Weiterarbeit mit Kontext-freien Grammatiken
- **TDiag** — Programm zur Erstellungen und Bearbeitung von T-Diagrammen im Compiler-Bau
- **VCC** — der Visual Compiler Compiler dient der Erstellung und Testung eigener Compiler
- **SchemeEdit** — Editor für erzeugte Petite Chez Scheme-Dateien

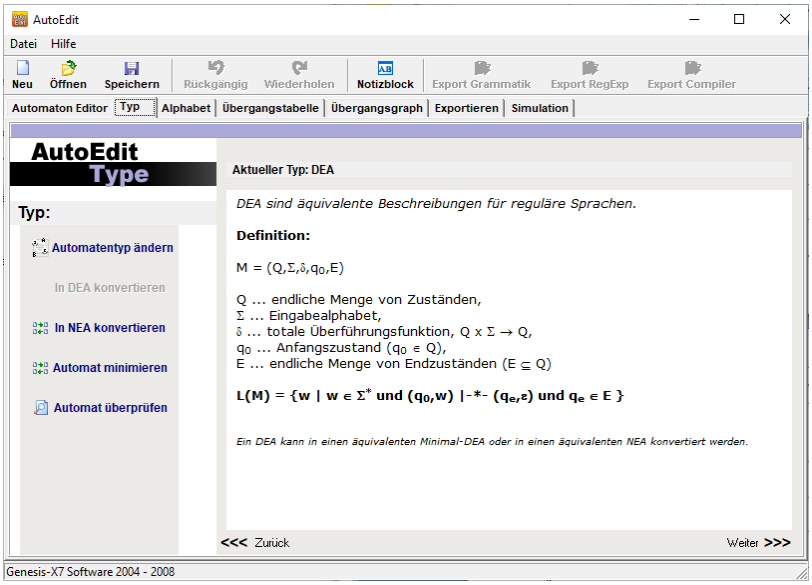
3.2.5.2.1. Arbeiten mit dem Modul AutoEdit

Das Programm AtoCC – im Konkreten "AutoEdit" – startet mit einem Assistenten zur Definition und Nutzung von Automaten.
AutoEdit steht für Automaten-Editor.
Vorhandene Automaten können über "Automat öffnen" geladen werden.
Zum Erstellen eines neuen Automaten wählen wir die Schaltfläche "Neuer Automat".
Über die "vor"- (>>>) und "zurück"-Schaltflächen (<<<) kann man sich jederzeit durch den Assistenten bewegen.

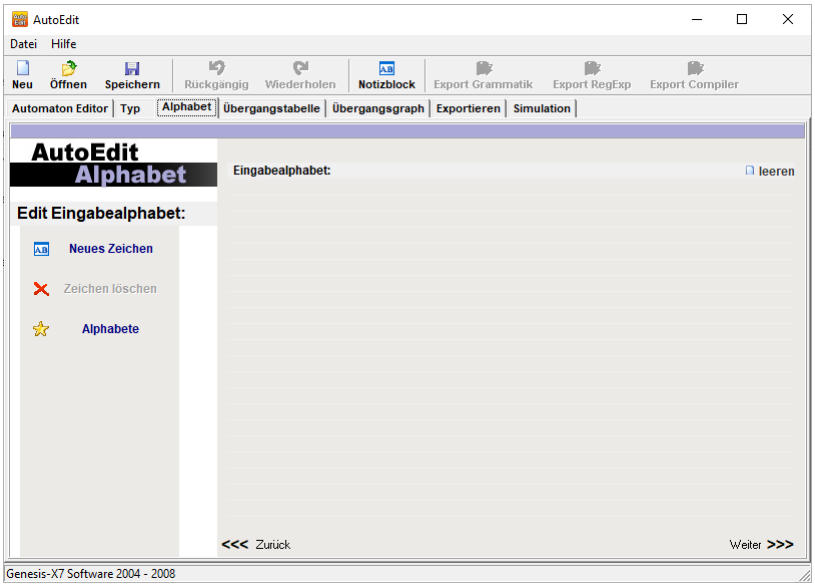


Gleiches funktioniert über die Reiter über dem Assistenten-Fenster.

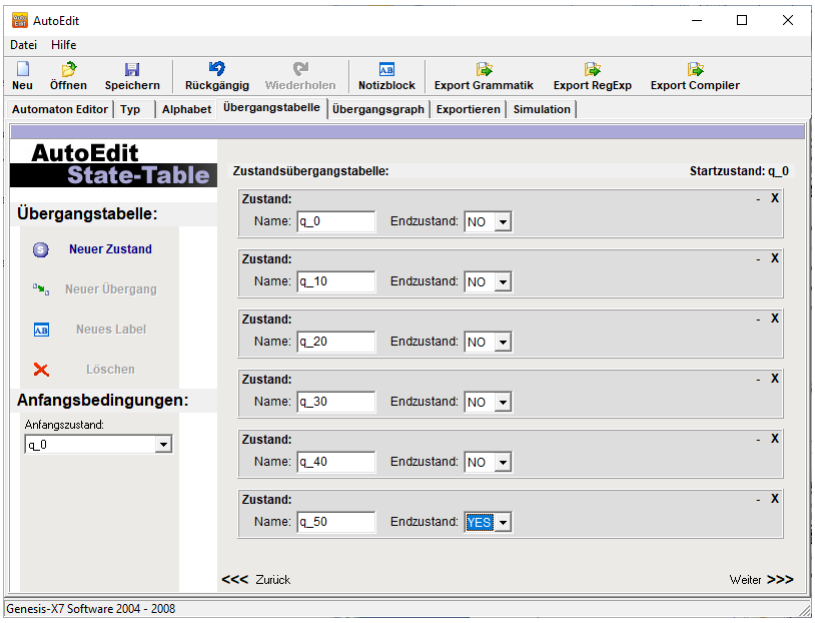
Die nächste Aufgabe innerhalb des Assistensten ist es den Automaten-Typ festzulegen. In gewissen Grenzen ist eine Änderung nachträglich auch noch möglich.



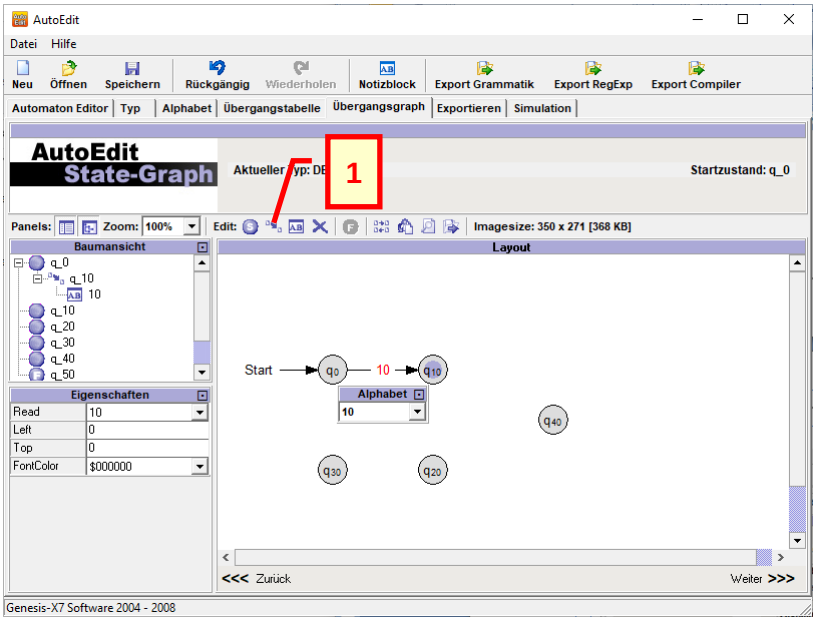
Die Definition des Eingabe-Alphabet's ist der nächste Schritt.
Es lassen sich fertige "Alphabete" auswählen oder eben ein eigenes definieren. In jede Zeile wird ein Zeichen oder eine Zeichenfolge als Symbol des Eingabe-Alphabet's eingegeben.



Es folgt die Definition der (Menge der) Zustände. Der Name ist frei wählbar und sollte systematisch erfolgen. Für jeden einzelnen Zustand muss entschieden werden, ob er als Endzustand taugt.

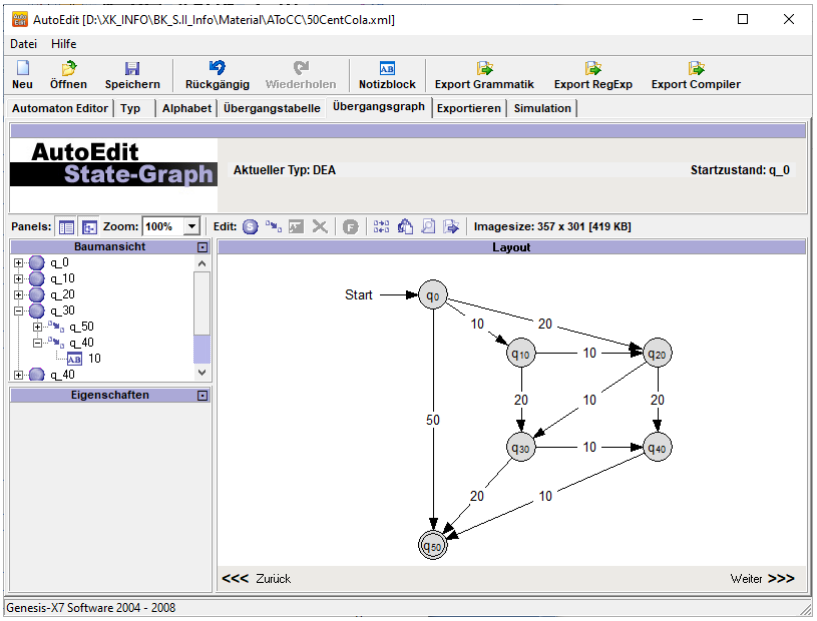


Nach dem Wechsel in den nächsten Assistenten-Schritt "Übergangsgraph" müssen die Übergänge definiert werden.
Über die etwas unscheinbare Schaltfläche "Neuer Übergang (t)" (**1**) wird eine neue Kante im graph erzeugt. Es erscheint auch eine Auswahl-Box, aus der die zugehörige Eingabe augwählt werden muss.
Vor jedem neuen Übergang muss immer erst die Schaltfläche (**1**) angeklickt werden, ansonsten ist man im Editier-Modus und kann die Knoten verschieben.

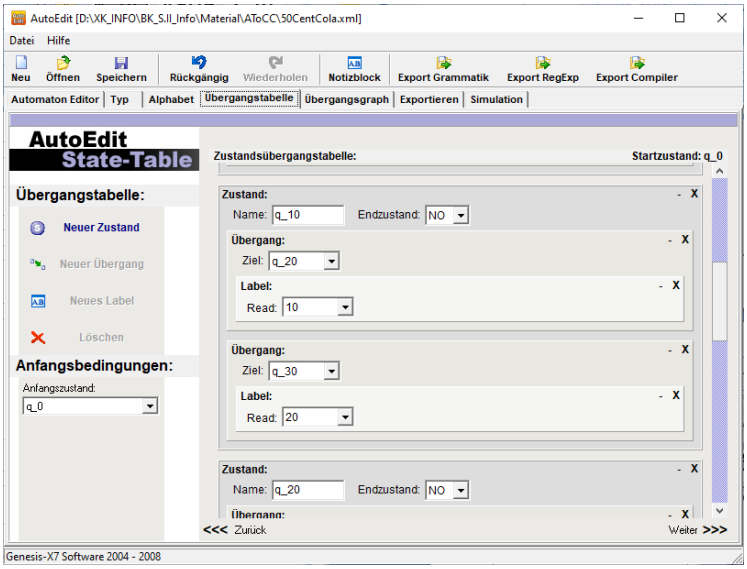


Sollen mehrere Eingaben zum gleichen Folge-Zustand führen, dann legt man einfach einen neuen Übergang an. Die Eingaben werden dann zusammen aufgelistet.

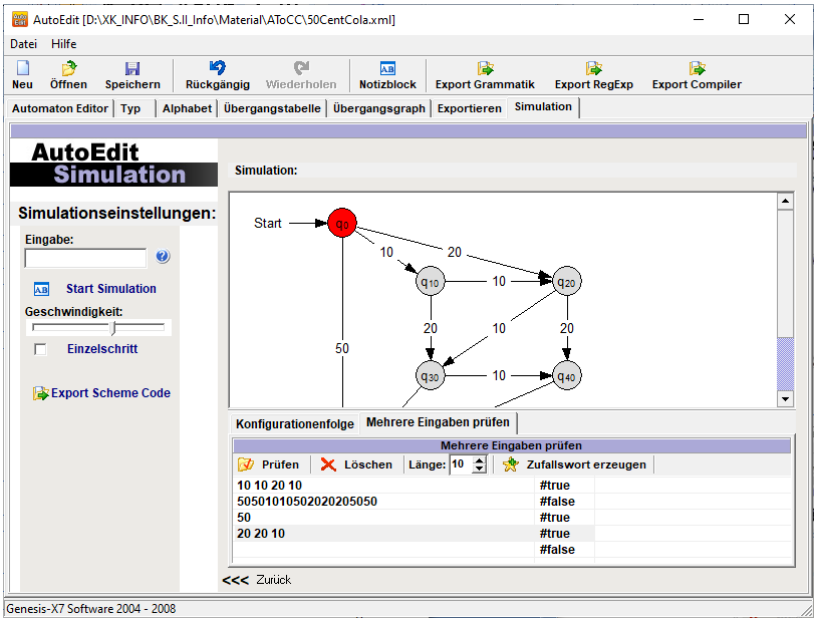
Spätestens hier ist das Abspeichern der Automaten-Definition sinnvoll.



Die "Übergangstabelle" ist etwas unübersichtlich. Hier können aber alle Detail's nochmals geändert werden. Ein Zustand mit seinen Übergängen ist immer durch ein entsprechend großes Rechteck begrenzt.
Ein echte Tabelle, wie wir sie gemeinhin fordern ist dies nicht.

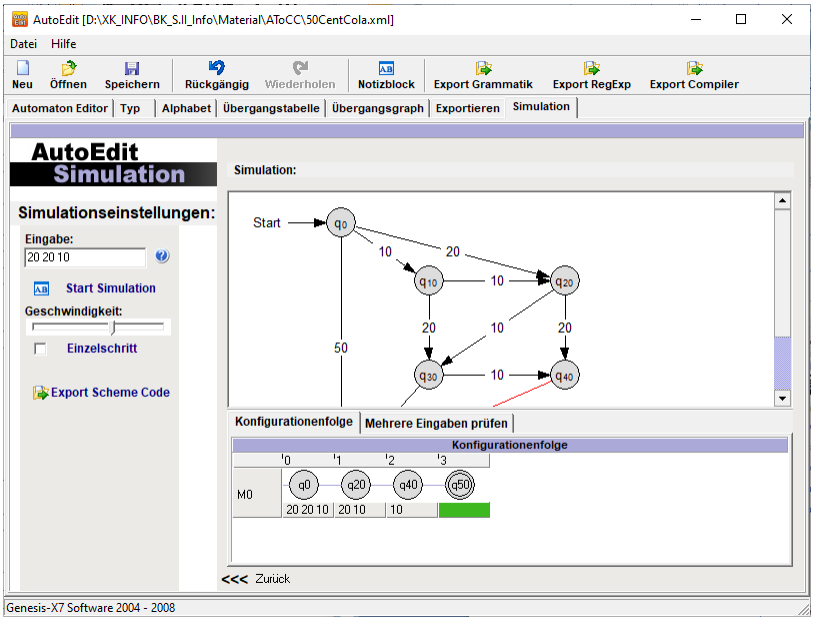


Ob nun mit einem gerade "geöffneten" oder "erstellten" Automaten, irgendwann wollen wir in nutzen / testen / simulieren.
Unter dem Graphen finden wir eine Tabelle, in der wir die zu testenden Eingabe-Worte vorgeben können.
Zufällige Eingabe-Folgen lassen sich genauso erzeugen wie eigene Sequenzen. Die Eingaben können mit und ohne Leerzeichen notiert werden.



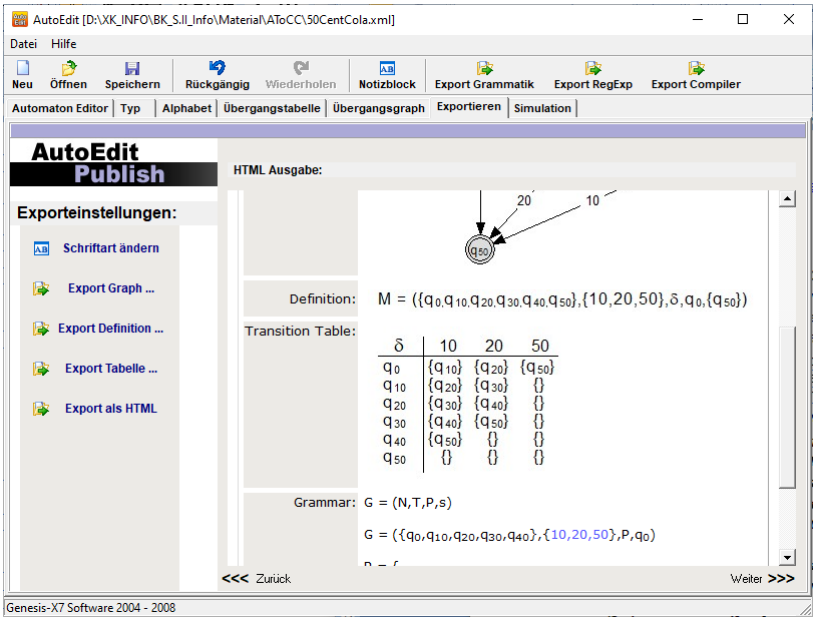
Über die Schaltfläche "Prüfen" werden alle Eingabeworte in einem Ritt getestet.

Einzelne Eingaben lassen sich auch über den linken Bereich simulieren.
Wählt man dann unten den Reiter "Konfigurationsfolge" an, dann erhält man nach einem "Start Simulation" die Zustands-Folge.
Ist das Eingabe-Band am Ende grün gefärbt, dann wird das "Eingabe"-Wort akzeptiert, sonst eben nicht.



Mit längeren Eingaben, wie sie bei NEA's eigentlich zugelassen sind, geht AutoEdit etwas eigenwillig um und akzeptiert diese ev. nicht.

Unter dem Reiter "Exportieren" erhalten wir eine schöne Zusammenfassung zu unserem Automaten. Hier ist dann auch eine übersichtliche Zustands-Übergangs-Tabelle angegeben.



Links:
http://atocc.de (Programm-Webseite)
http://atocc.de/cgi-bin/atocc/site.cgi?lang=de&site=tutorials (div. online-Tutorials)

Aufgaben:

1. *Bilden Sie mit AtoCC den 50-Cent-Cola-Automaten nach!*
2. *Dokumentieren Sie mit einfachen Bildschirm-Kopien die Konfigurationsfolgen des Automaten für die nachfolgenden Eingaben!*

a)	50	b)	202020	c)	10201010
d)	3020	e)	1010101010	f)	2050
g)	10201010	h)	5050	i)	10101020

3. *Erstellen Sie mit AtoCC den 50-Cent-Cola-Automaten als Deterministischen Endlichen Automaten (DEA)!*
4. *Entwickeln Sie mit AtoCC einen NEA, der Wörter aus einer Sprache akzeptiert, die mit mindestens 3x "a" beginnen und dann mindestens 2x "b" und 2x "c" folgen! Dokumentieren Sie mit einfachen Bildschirm-Kopien den Automaten und mindestens 6 Test's mit akzeptablen und nicht-akzeptablen Wörtern!*
5. *Gesucht ist ein DEA, der Einsen im Bereich von 0 bis 9 zählt. Das "Zählwort" beginnt immer mit "1010" gefolgt von 0 bis 9 Einsen.*
   Beispiel: 101011111 für das Zähl-Ergebnis "5"
   Dokumentieren Sie Automat und Test's!
6. *Erstellen Sie einen Automaten, der die Wörter mit "hör" oder eine entsprechende von Ihnen definierte Wörtergruppe (/ Wortstamm) akzeptiert (s.a. Aufgabe unter → 3.2.4.)!*

3.2.5.3 Simulation von endlichen Automaten mit AuDeS

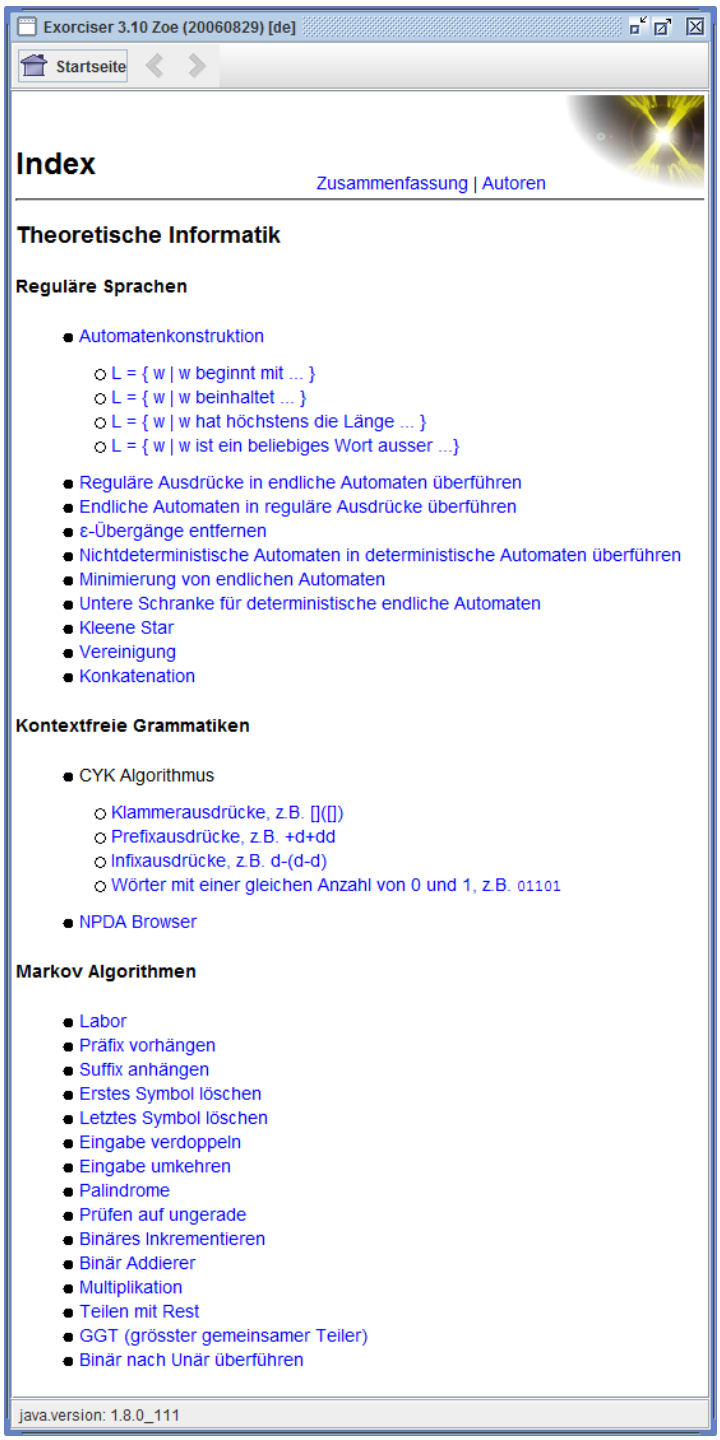
AuDeS steht für "Automaten Designer für Schüler". Das Programm ist als portables Programm (portable App) einsetzbar. Für Anpassungen von Pfaden usw. ist eine im Programm-Verzeichnis liegende Konfigurations-Datei (AuDeS.exe.config) zu editieren.
Für die nicht so im XML-Bewanderten hier einige kurze Hinweise zum Konfigurieren.
Vor dem Editieren zuerst einmal die alte Datei sichern oder in ein anderes Verzeichnis kopieren. Die Datei lässt sich recht gut mit dem Editor "Notepad++" ändern. Dieser Editor unterstützt das Higlighting für XML. Dadurch werden zusammenghörende Stellen und die änderbaren Einträge leichter erkennbar. In den meisten Fällen wird es wohl nur auf die Änderung der Einträge mit den Schlüsseln ("key") für das "AbgabeVerzeichnis" und den Installations-Ordner für BlueJ ("BlueJPath") hinauslaufen. Die änderen dürfen nur im Bereich "value" zwischen den nach dem Gleichheitszeichen folgenden Anführungszeichen erfolgen. Wenn man also z.B. seinen Abgabe-Ordner aus dem eigenen lokalen Netzwerk "X:\Abgabe\Informatik" nutzen möchte, dann muss ganau dieser Pfad bei value eingetragen werden. Der XML-Tag würde dann so aussehen:

```
<add key="AbgabeVerzeichnis" value="x:\Abgabe\Informatik" />
```

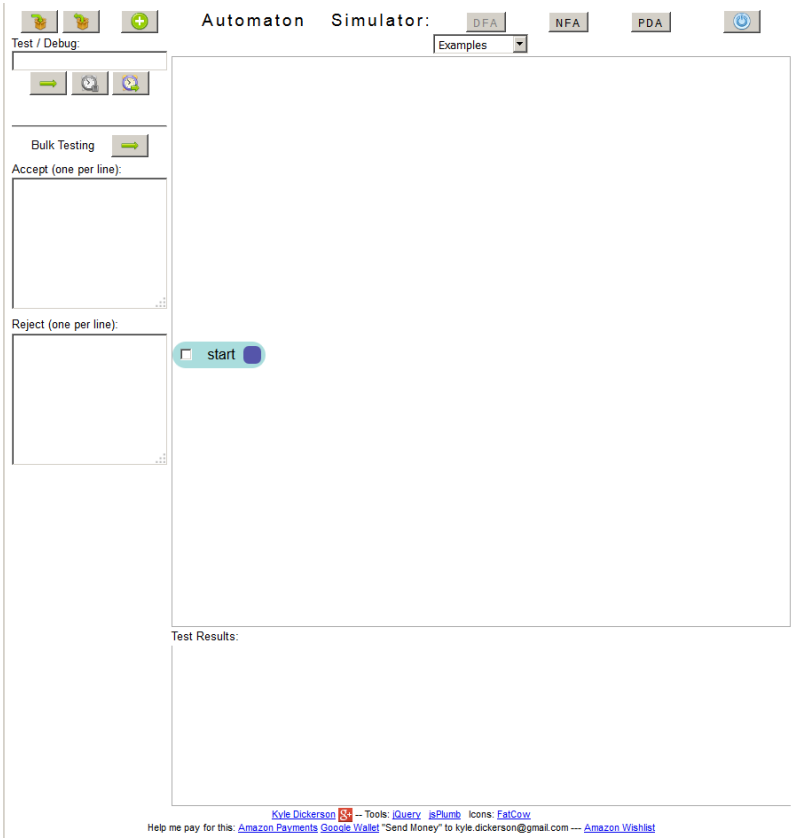
Gleichermaßen verfährt man mit dem Pfad zu "BlueJ", je nach dem wo das Programm installiert ist.
Der letzte Pfad ist besonders wichtig, wenn aus dem Automaten-Diagramm (im Programm unter "Design" angelegt) später ein BlueJ-UML-Objekt mit passendem Code erzeugt werden soll. Der reine Java-Quellcode ist aus dem Programm selbst auch schon erzeugbar.

3.2.5.4. Arbeiten mit dem Programm Exorciser

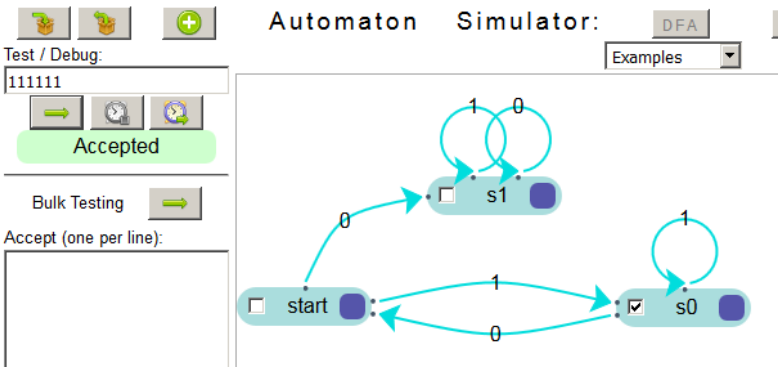
Exorciser ist eher als gesamtheitliches Übungs- und Lern-Programm angelegt. Es enthält Aspekte des "programmierten Lernen". Das Programm orientiert sich auch mehr am Studenten der Theoretischen Informatik, was aber nicht seine Verwendbarkeit in der Sek. II schmälert. Nur werden wir hier nicht alle Punkte / Inhalte betrachten können und wollen(!).



3.2.5.5. online-Simulator "AutomatonSimulator.com"



Wegen der großflächigen Anlage der Applikation werden im Folgenden immer nur die wichtigen Ausschnitte gezeigt.
Die Applikation ist denkbar einfach zu benutzen. Die ausschließlich englischsprachige Oberfläche stört da kaum, da eigentlich alle Schaltflächen sinnig mit Pictogrammen versehen sind und auch kleine Hilfs-Textchen beim Überfahren mit der Maus angezeigt werden.
Beim Start der Applikation ist DFA – also ein DEA (deterministischer endlicher Automat) – voreingestellt. Etwas verwirrend ist vielleicht, dass der aktive Automaten-Typ angegraut dargestellt wird. Die anderen beiden Automaten-Typen sind nichtdeterministische endliche Automaten (NFA) und der Keller-Automat PDA (Push-down Automaton).
Aber Achtung! Beim Umschalten wird der alte Automat gelöscht.
Zum Ausprobieren sind drei Automaten als Beispiele unter "Examples" – einschließlich Tests – zu finden.



Über die Schaltfläche mit dem grünen Plus fügt man sich neue Zustände hinzu. Die Bezeichnung erfolgt automatisch. Der Start-Zustand ist immer gesetzt. Über das Options-Kästchen setzt man den Zustand als End-Zustand.
Die Verbindungs-Linien werden ausgehend von den dunkelblauen Bereichen zum Ziel-Zustand aufgebaut. Die Angabe der Eingabe ist obligatorisch.
Natürlich lassen sich auch Verbindungen zu sich selbst ziehen.
Der fertige Automat kann über das ganz linke Symbol ("Pfeil in Paket") im Browser-Speicher erfolgen. Alternativ kann man sich den Automaten als "Plaintext" anzeigen lassen und in einen Editor kopieren. Dort lässt er sich dann formatieren, drucken und auch lokal als Text-Dokument speichern.
Bei "Test / Debug:" kann eine zu prüfende Zeichenkette eingegeben werden. Mit dem darunter positionierten grünen Pfeil wird der test gestartet. Mit der ganz rechten Schaltfläche (Uhr

mit Pfeil) kann der Prüfvorgang Zeichen-weise – bzw. besser gesagt Zustand-weise geprüft werden.
Ein größere Liste von Wörtern kann bei "Bulk Test" geprüft werden. Unter "Accept" kommen die Wörter, die der Automat akzeptieren sollte und die anderen werden bei "Reject" aufgelistet. Der Test lässt sich dann mit der "Pfeil"-Schaltfläche neben "Bulk Test" als Gruppe starten.

http://automatonsimulator.com/

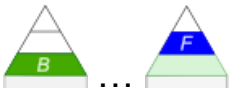
3.2.5.6. Arbeiten mit dem "FSM Creator"

FSM steht für "Finite State Machine". Das Programm ist eine JAVA-Applikation und in englischer Sprache. Somit muss also eine JAVA-Umgebung auf dem ausführenden Rechner vorhanden sein. Das sollte unabhängig von der Betriebssystems-Basis auf fast jedem Rechner heute möglich sein.

Direkt-Link (zum Programm-Download):
http://wvsg.schulen2.regensburg.de/joomla/images/Faecher/Informatik/Informatik_12/Programme_Loesungen/1_3_Endliche_Automaten/FSMCreator.jar

Link:
http://wvsg.schulen2.regensburg.de/joomla/images/Faecher/Informatik/Informatik_12/informatik_12_1_3_Endliche_Automaten.html

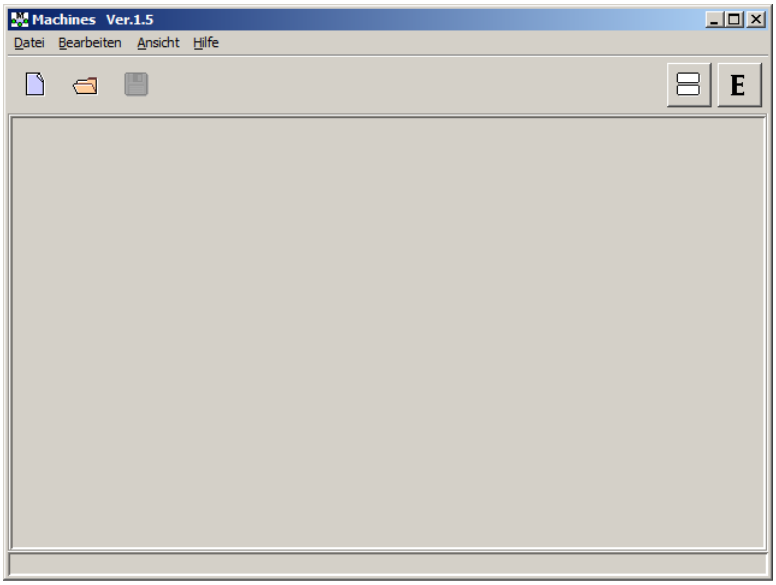
3.2.5.7. Arbeiten mit Machines



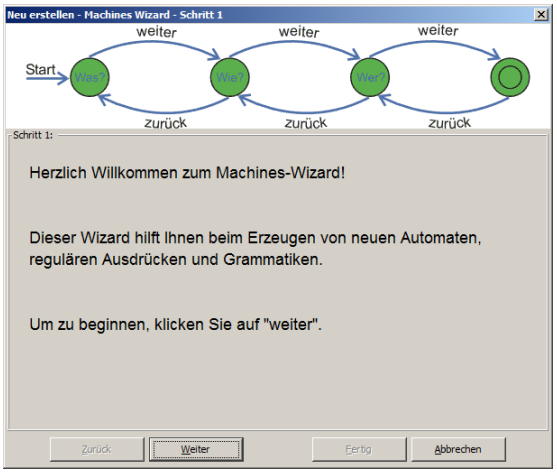
benötigt Java-Umgebung
auch für Linux verfügbar
auf dem IoStick vorhanden und lauffähig

einfache Bedienung; kaum Einarbeitungs-Zeit notwendig
auch für die Bearbeitung von regulären Ausdrücken (→) und Grammatiken (→ x.y.z.4. Grammatiken bearbeiten / testen mit Machines) nutzbar
didaktisch orientiert
Konzentration auf das Wesentliche
übersichtlich
wirkt manchmal einwenig altbacken

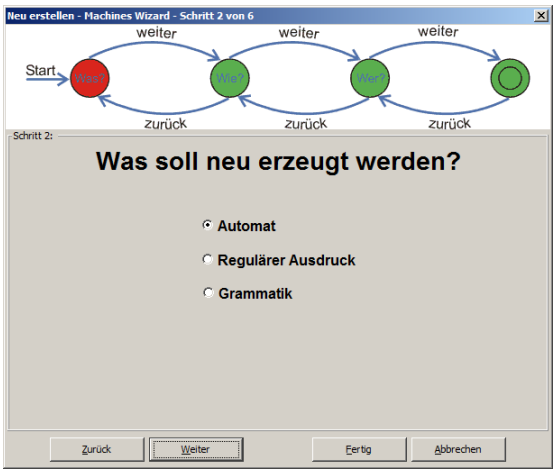
Übersicht / Struktur / Bedienteile

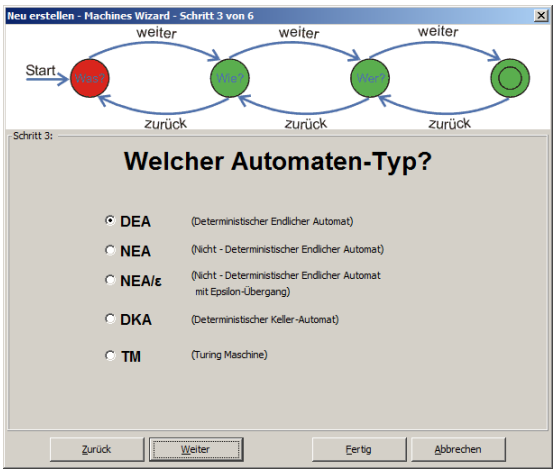


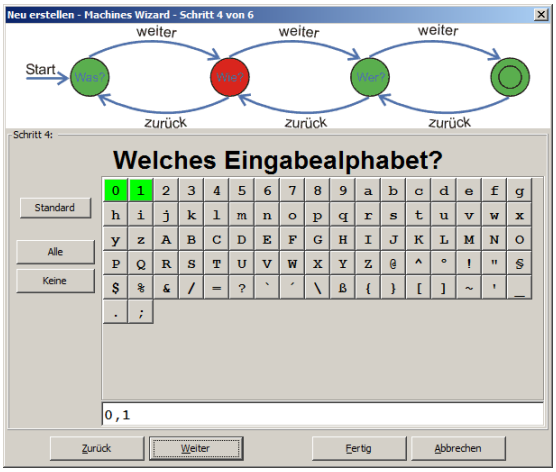
es gibt einen Automaten-Wizard, der bei der Erstellung eines Automaten hilft
wird automatisch beim Erstellen eines neuen Automaten aufgerufen, lässt sich aber abkürzen (wenn bestimmte "Standards" benutzt werden sollen)

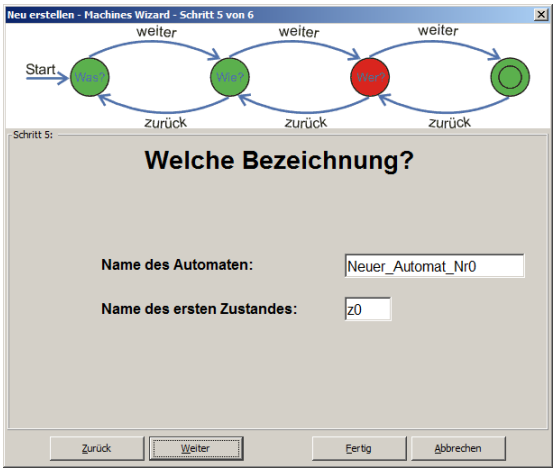


ist so eindeutig und intuitiv, dass wir hier nur die Schritte abbilden



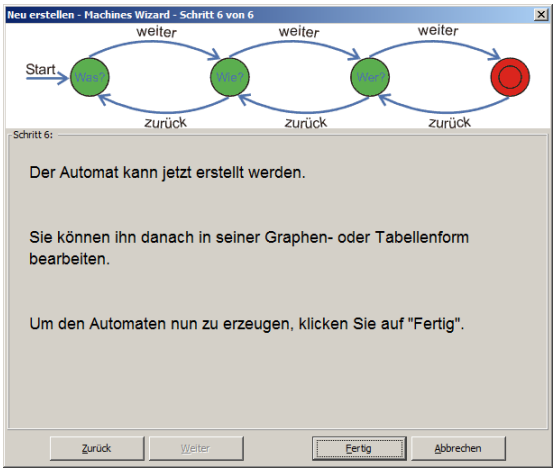




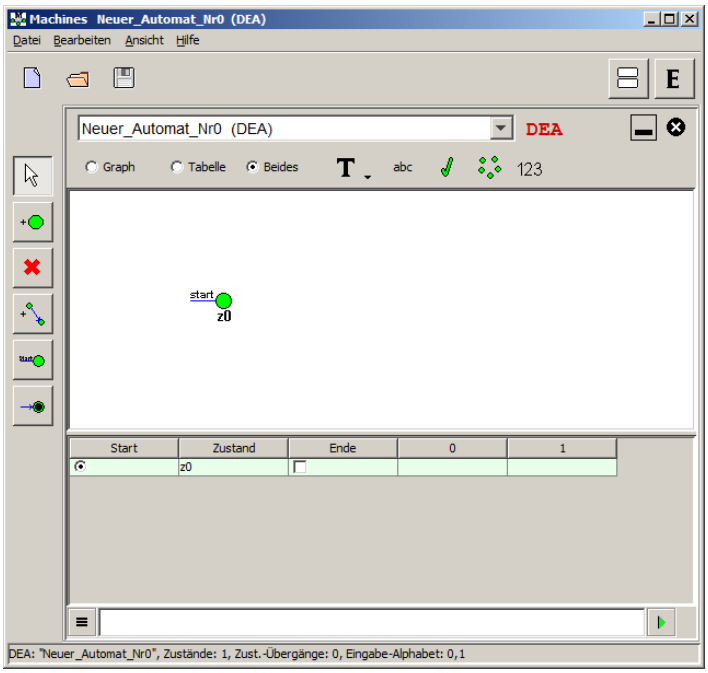


Alphabet ist nach Einstellungen nicht mehr zu verändern

Automaten-Dateien haben Dateityp-Endung je nach Automaten-Typ
ein DEA ist eben nur als DEA zu nutzen



zum Arbeiten und Erkunden von Automaten ist die parallele Ansicht von Graph und Tabelle (Automaten-Tafel) sehr interessant



Automaten lassen sich mit Eingabe-Worten testen
schrittweise mit definierbarer Pausen-Zeit (delay) oder in einem Ritt
aktuelles Arbeits-Ergebnis wird in rot/grün angezeigt

Automaten-Typen in Machines

- **DEA (deterministischer endlicher Automat)**
  näher ausgeführt:
  → x.y.z.1. deterministische endliche Automaten

- **NEA (nicht-deterministischer endlicher Automat)**
  näher ausgeführt:
  → x.y.z. nicht-deterministische endliche Automaten

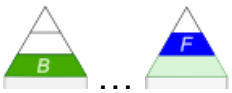
- **NEA/ε (nicht-deterministischer endlicher Automat mit ε-Übergang)**
  näher ausgeführt:
  → x.y.z. nicht-deterministische endliche Automaten

- **DKA (deterministischer Keller-Automat)**
  näher ausgeführt:
  → x.y.z. (deterministische) Keller-Automaten

- **TM (TURING-Maschine)**
  näher ausgeführt: → x.y.z. TURING-Maschinen

Links: (da der Autor / Verantwortliche ein "Reisender" ist, können einzelne Links ungültig sein! Links aus der Hilfe / Über sind ungültig!)
(als gute Such-Kombination haben sich: "Socher Machines " erwiesen)
http://zeus.fh-brandenburg.de/~socher/tgi/ (Download's und Handbuch)

3.2.6. Automaten mit Ausgaben - Transduktoren

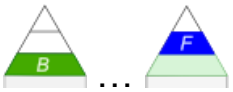


Bei der Einteilung der Automaten haben wir eine Gruppe dadurch gekennzeichnet, dass diese Ausgaben produziert. Natürlich kommunizieren auch die anderen Automaten mit der Umwelt, bei den **Transduktoren** – wie die **Automaten mit Ausgabe** auch heißen – erfolgt aber immer eine direkte Ausgabe. Die Akzeptoren – also die Automaten ohne Ausgabe – erzeugen nur eine indirekte Ausgabe zum Gesamt-Ergebnis der Automaten-Tätigkeit. I.A. ist das dann nur ein "akzeptiert" oder "nicht-akzeptiert".
Unsere – auch endliche Transduktoren genannten – Automaten arbeiten in den meisten Fällen als kontinuierliche Maschinen, d.h. sie sind immer online. Wenn eine Eingabe kommt, wird diese verarbeitet und die Ausgabe getätigt, kommt keine Eingabe, dann wird einfach gewartet.
Die im Folgenden besprochenen Automaten sind solche ohne Speicher-Möglichkeit. Die Automaten kennen auch keine früher durchlaufenen Zustände. Wir als äußerer Beobachter können natürlich Protokoll über die vergangenen Schritte führen.
Für die Transduktoren mit Speicher-Möglichkeiten machen wir extra Kapitel (→ 3.2.10. TURING-Maschinen + 3.2.12. Register-Maschinen) auf. Das ist zwar nicht ganz systematisch, wird aber der Komplexität und der Bedeutung dieser Automaten gerecht.

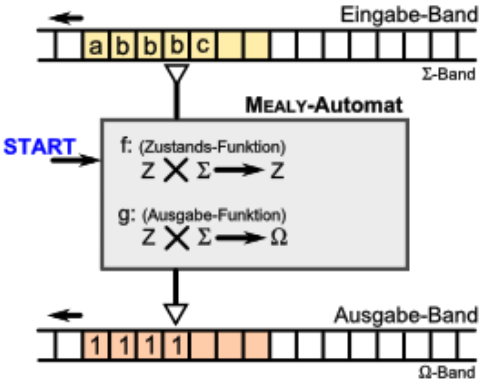
Definition(en): Transduktor
Ein Transduktor ist ein endlicher Automat, der aus einer Seuenz von Eingabe-Symbolen eine Sequenz von Ausgabe-Symbolen erzeugt.
Transduktoren sind endliche Automaten, die eine Ausgabe-Funktion enthalten.
Transduktoren sind übersetzende Automaten. Sie wandeln Worte der einen Sprache (Eingabe-Symbole) in Worte einer anderen Sprache um (Ausgabe-Symbole).

Neben den MEALY- und MOORE-Automaten, die wir im Folgenden besprechen werden, gibt es in der Klasse der Transduktoren auch noch die HAREL-Automaten (→ 3.2.6.4. weitere Zustands-Automaten mit Anzeige).
HAREL-Automaten (→ 3.2.6.4.1. HAREL-Automat) besitzen hierarchisch organisierte (Super-)Zustände und Ereignis-abhängige Wächter-Funktionen. Als Ausgabe erfolgt eine Anzeige der erreichten Zustände.

3.2.6.1. MEALY-Automaten



In der Fachliteratur auch häufig als MEALY-Maschine bezeichnet. Als Maschinen werden in der Theoretischen Informatik Automaten mit Ausgaben betrachtet. Somit müsste ein MEALY-Automat also wirklich exakter-weise als MEALY-Maschine benannt werden. Dies wird aber in den wenigsten Büchern so getan.
Diese Begriffs-Bestimmung ist aber nicht eindeutig und durchgängig. Wir sprechen hier – wie in den meisten Schul-Büchern oder Schul-/Abitur-Aufgaben von "Automaten mit Ausgaben" oder Transduktoren.
Namensgebend ist der Mathematiker George H. MEALY (1927 – 2010), der sich in Amerika mit der Automaten-Theorie beschäftige.
Das Konzept eines Automaten mit Ausgabe veröffentlichte MEALY 1955.
bei MEALY-Maschinen wird jeder Übergangs-Funktion (also jeder Kante im Zustands-Diagramm) eine Ausgabe zugeordnet
eine Ausgabe erfolgt also in direkter Abhängigkeit von der Eingabe während des Zustands-Wechsel

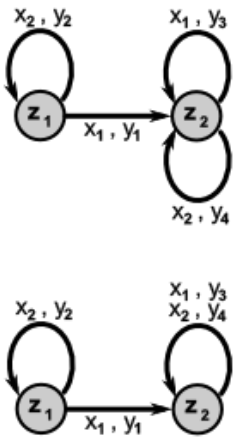


MEALY-Automat
als Transduktor (Übersetzer)

von STARKE () dann 1969 auch als deterministischer MEALY-Automat eingeordnet

kann auch nicht-deterministisch sein, wird aber selten betrachtet

Die Ausgaben des Automaten müssen dabei nicht zwangsläufig etwas mit den einzelnen Zuständen zu tun haben. So kann z.B. ein Automat mit 100 verschiedenen Zuständen trotzdem nur zwei, drei verschiedene Zeichen ausgeben.
Um die graphische Darstellung etwas übersichtlicher zu machen, dürfen Überführungen auch zusammengefasst werden. Im nebenstehenden Beispiel ist – ausgehend von Zustand z_2 jeweils eine Überführung nach Eingabe von x_1 bzw. x_2 möglich. In beiden Fällen wird wieder z_2 als Folge-Zustand erreicht. Beide Graphen werden zu einem zusammengefasst. Die einzelnen Überführungs-Funktionen werden Tabellen-artig an den Graphen geschrieben. Bei sehr großen Eingabe-Alphabeten dürfen auch weitere Zusammenfassungen bzw. Verallgemeinerungen vorgenommen werden. Ist z.B. jede Ziffer ein Eingabe-Zeichen, dann könnte man für den gleichen Folge-Zustand und gleiche Ausgabe y_x, wie folgt zusammenfassen: 0 .. 9, y_x
Ein so umgewandelter – besser umgeschriebener / umgezeichneter – Automat hat sich nicht verändert. Die echte Vereinfachung – auch Reduktion genannt – ist ein anderes Vorgehen. Dazu später mehr.



Automaten-Tafel

Abbildung der Folge-Zustände (Überführungs-Funktion)

(aktueller) Zustand	Eingabe-Symbole				
	x_0	x_1	x_2	…	x_n
z_0	z_1	z_4	…	…	…
z_1	z_1	…	z_2	…	…
z_2	…	z_n	…	…	…
…	…	…	…	…	…
z_n	…	…	…	…	…

Abbildung der Ausgaben (Ausgabe- / Ergebnis-Funktion)

(aktueller) Zustand	Eingabe-Symbole				
	x_0	x_1	x_2	…	x_n
z_0	y_3	y_3	-…	…	…
z_1	y_2	…	y_1	…	…
z_2	…	y_n	…	…	…
…	…	…	…	…	…
z_n	…	…	…	…	…

zusammengefasste / kombinierte Automaten-Tafel

Abbildung der Folge-Zustände und Ausgaben (kombinierte Ausgabe- / Ergebnis-Funktion)

(aktueller) Zustand	Eingabe-Symbole				
	x_0	x_1	x_2	…	x_n
z_0	z_1, y_3	z_4, y_3	…, …	…	…
z_1	z_1, y_2	…, …	z_2, y_1	…	…
z_2	…, …	z_1, y_n	…, …	…	…
…	…	…	…	…	…
z_n	…	…	…	…	…

Definition(en): MEALY-Automat

Ein endlicher Automat ist ein Sept-Tupel (7-Tupel) $A = (Z, \Sigma, \Omega, f, g, z_0, Z_E)$ mit:

Z … endliche Menge der Zustände; $|Z| < \infty$
Σ … Eingabe-Alphabet; $|\Sigma| < \infty$, $Z \cap \Sigma = \varnothing$
Ω … Ausgabe-Alphabet; $|\Omega| < \infty$
f … Überführungs-Funktion (totale Funktion); $Z \times \Sigma \rightarrow Z$
g … Ausgabe-Funktion; $Z \times \Sigma \rightarrow \Omega$
z_0 … Start-Zustand; $z_0 \in Z$
Z_E … endliche Menge der akzeptierten Zustände (End-Zustände); $Z_E \subseteq Z$

wird MEALY-Automat genannt.
Die Menge Z_E kann entfallen (wenn z.B. das Akzeptieren keine Rolle spielt / die reguläre Sprache des Automaten nicht interessiert), dann lässt sich der MEALY-Automat **auch als Hex-Tupel (6-Tupel):** $A = (Z, \Sigma, \Omega, f, g, z_0)$ defnieren.

Ein MEALY-Automat ist ein endlicher Automat mit (zusätzlichen) Ausgabe(n) (- also ein Transduktor), wobei die Ausgaben während der Zustands-Änderung erfolgt.

Ein MEALY-Automat ist ein endlicher Automat, bei dem die Ausgaben ausschließlich vom jeweiligen Zustands-Übergang abhängig sind.

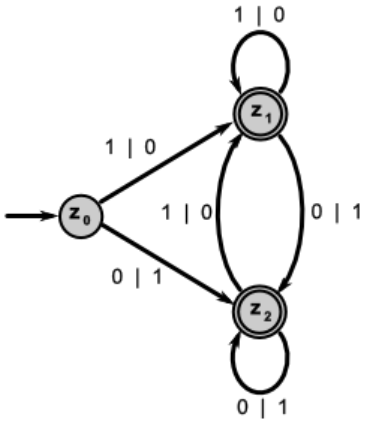
Ein deterministischer abstrakter MEALY-Automat ist ein Quad-Tupel (4-Tupel) $A = (\Sigma, Z, \Omega, h)$ mit:

Σ … Eingabe-Alphabet; endliche Menge von Symbolen / Zeichen
Z … endliche Menge der Zustände
Ω … Ausgabe-Alphabet; endliche Menge von Symbolen / Zeichen
h … Überführungs-Funktion; $h(x,y) = [f(z,x),g(z,x)]$; wenn $x_0 \in \Sigma$ und $y \in \Omega$
f … Zustands-Überführungs-Funktion
g … Ausgabe-Funktion

Der Start-Zustand wird standard-mäßig mit z_0 festgelegt.

3.2.6.1.1. Arbeit eines MEALY-Automaten

Wir wählen hier als Beispiel einen relativ einfachen, übersichtlichen Automaten, damit wir nicht unendlich viel zu Schreiben haben. Der Automat soll aus einer eingegeben 1 eine 0 machen und umgekehrt. Wir sprechen auch vom Invertieren.
Zum Vergleich werden wir die gleiche Funktion auch später noch bei anderen Automaten besprechen und dort deren spezielle Arbeitsweisen aufzeigen.
Die Definition unseres MEALY-Automaten könnte in etwa so so aussehen:



$A = (\{z_0,z_1,z_2\}, \{0,1\}, \{\varepsilon,0,1\}, f, g , z_0, \{z_1,z_2\})$

Die Funktionen **f** und **g** sind in separaten Tabellen gegeben. Das erleichtert uns das Heraussuchen der passende Werte für die Notierung im Protokoll.

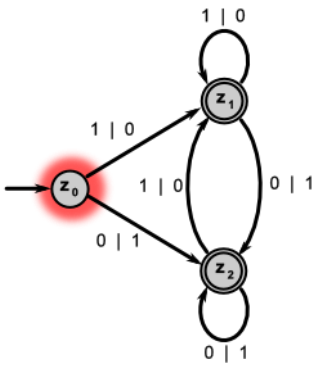
f:	Σ	
Zust.	**0**	**1**
z_0	z_2	z_1
z_1	z_2	z_1
z_2	z_2	z_1

g:	Σ	
Zust.	**0**	**1**
z_0	1	0
z_1	1	0
z_2	1	0

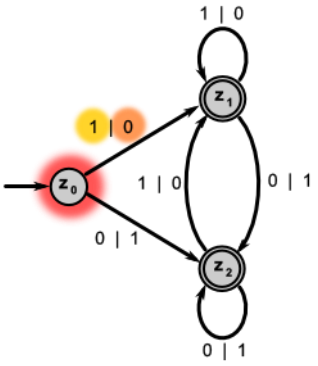
Aufgaben:

1. *Erläutern Sie die Definition des obigen Automaten!*
2. *Notieren Sie die Definition des obigen Automaten in reiner Text-Form (also ohne Tabelle(n))!*

Zum Nachverfolgen sei auf dem Eingabe-Band die Zeichen-Kette 01101 gegeben.
(Die farbliche Untermalung für Eingabe- und Ausgabe-Band wird hier nur zu besseren Unterscheidung gemacht.)

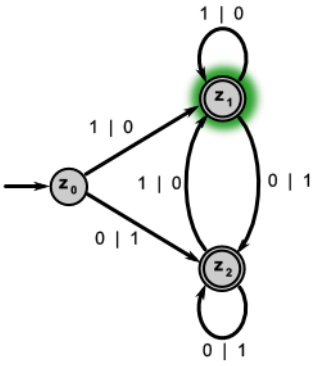


1
Gestartet wird mit Zustand z0. Dieser ist kein Endzustand.

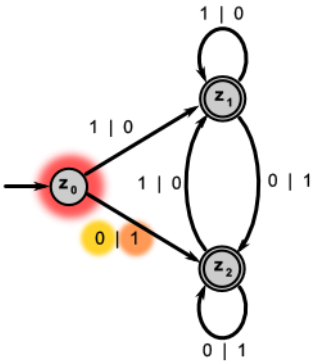


2
Mit dem Lesen einer 1 vom Eingabe-Band würde der Automat eine 0 ausgeben.

z_0 x 1 → z_1 (f)
z_0 x 1 → 0 (g)



3
Es würde dann der Zustandswechsel nach Zustand z1 erfolgen.

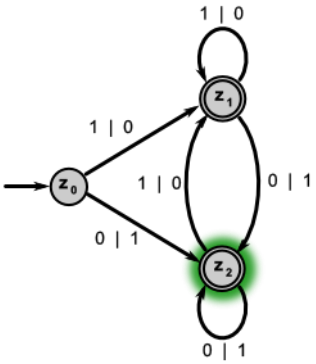


4

Aber das erste Zeichen auf dem Σ-Band (**0**1101) ist die 0. Damit wird der Wechsel zu z_2 ausgelöst und im Übergang eine 1 ausgegeben.

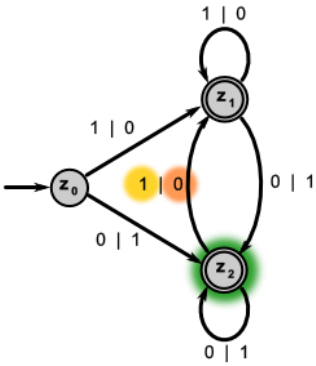
z_0 x 0 → z_2 (f)
z_0 x 0 → 1 (g)

Das aktuelle Ω-Band sähe dann so aus: **1**.



5

Im Zustand z_2 wartet der Automat nun auf die nächste Eingabe. Das ist auf unserem Eingabe-Band (0**1**101) eine 1.

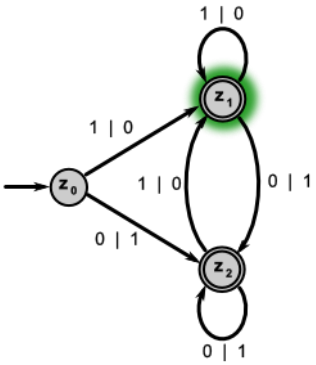


6

Mit der Eingabe der 1 kommt es zum Zustands-Übergang zu z_1 verbunden mit der Ausgabe einer 0.

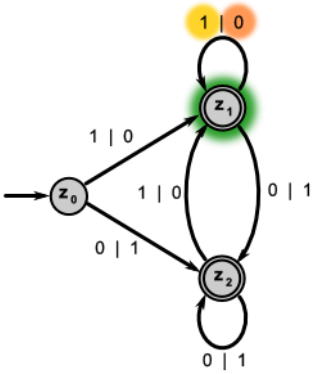
z_2 x 1 → z_1 (f)
z_2 x 1 → 0 (g)

Das Ausgabe-Band verlängert sich somit auf: 1**0**



7

Somit ist der Automat wieder im Zustand z_1.

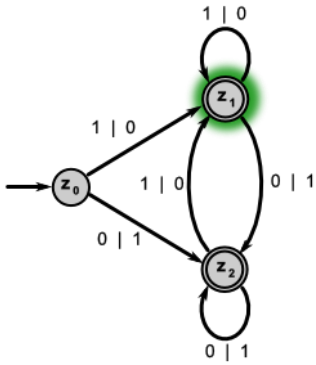


8

Durch eine Eingabe von einer 1 vom Σ-Band (01**1**01)ergibt sich:

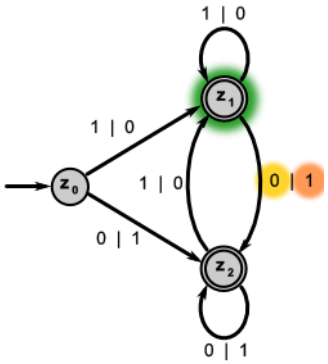
z_1 x 1 → z_1 (f)
z_1 x 1 → 0 (g)

ein Verbleib in z_1 und eine damit verbundene Ausgabe von 0 auf das Ausgabe-Band (10**0**).



9

Die Situation unterscheidet sich sachlich nicht von der in Schritt 7. Hier können wir gut sehen, dass es für den Automaten keine Möglichkeit gibt, zu erkennen aus welcher ursprünglichen Situation heraus er diesen Zustand erreicht hat. Der Automat hat keinen Speicher und auch kein internes Protokoll.

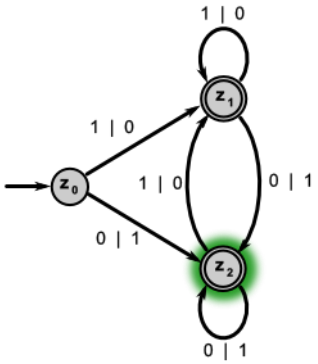


10

Das Eingabe-Band (0110**1**) liefert nun eine 0. Damit verbunden sind der Zustandswechsel nach z_2 du die Ausgabe einer 1:

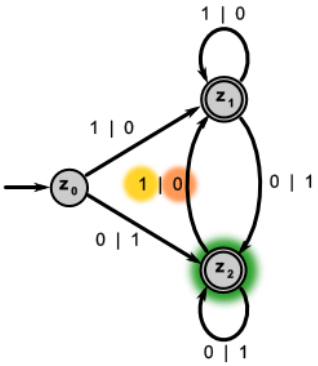
z_2 x 1 → z_1 (f)
z_2 x 1 → 0 (g)

Das Ω-Band zeigt uns nun die Folge: 100**1** an.



11

Der Automat ist nun wieder im Zustand z2.



12

Als letztes Zeichen befindet sich wiederum eine 1 auf dem Σ-Band (0110**1**). Die wirksamen Funktionen:

z_2 x 1 → z_1 (f)
z_2 x 1 → 0 (g)

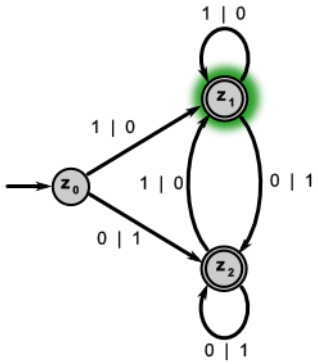
kennen wir schon. Mit dem letzten Eingabe-Zeichen ergibt sich auch ein quasi-Abschluss für das Ω-Band 1001**0**.

Alle ausgeführten Überführungs- und Ausgabe-Funktionen kann man sich in eine übersichtliche Tabelle zusammenfassen:

Schritt	f	g
Start	z_0	
1	z_0 x 0 → z_2	z_0 x 0 → 1
2	z_2 x 1 → z_1	z_2 x 1 → 0
3	z_1 x 1 → z_1	z_1 x 1 → 0
4	z_1 x 0 → z_2	z_1 x 0 → 1
5	z_2 x 1 → z_1	z_2 x 1 → 0

Um die Arbeit des Automaten zu dokumentieren, benötigt man drei Angaben. Das ist zum Einen der aktuelle Zustand und zum Zweiten das zu lesende Symbol auf dem Eingabe-Band. Die dritte Angabe ergibt sich aus der resultierenden Ausgabe des Automaten. So ein 3-Tupel (Tri-Tupel) wird Konfiguration k eines Automaten genannt und in eckigen Klammern **[ ]** notiert.
Beim oberen Automaten und der ersten Arbeits-Situation erhalten wir also die Konfiguration [z_0, 0, 1].
Bis zum Arbeits-Ende folgt eine Konfiguration auf die vorherige.
Man spricht dann von der **Konfigurations-Folge K**.

K = ($k_0 \vdash k_1 \vdash .. \vdash k_n$ **)**



13

Da das Ω-Band jetzt leer ist, verbleibt der Automat im Zustand z_1. Dieser ist im angegebenen deterministischen Modell ein End-Zustand, so dass auch eine Akzeptanz des Eingabe-Wortes unterstellt werden kann.

Das Zeichen " ⊢ " steht dabei für ***"führt zu"*** bzw. ***"es folgt"***.
Insgesamt ergibt sich also die Konfigurations-Folge:

K_A = ([z_0, 0, 1] ⊢ [z_2, 1, 0] ⊢ [z_1, 1, 0] ⊢ [z_1, 0, 1] ⊢ [z_2, 1, 0])

In einer erweiterten Tabellen-Form sähe die Konfigurations-Folge etwa so aus:
Der wesentliche Teil ist in der Tabelle bläulich gekennzeichnet.

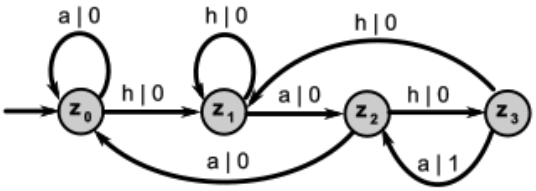
Konfigurationen des Automaten

Arbeits-Schritt	aktueller Zustand	(nächstes) Eingabe-Symbol	Ausgabe-Symbol	Folge-Zustand	Rest-Wort
1	**z_0**	**0**	**1**	z_2	**1101**
2	**z_2**	**1**	**0**	z_1	**101**
3	**z_1**	**1**	**0**	z_1	**01**
4	**z_1**	**0**	**1**	z_2	**1**
5	**z_2**	**1**	**0**	--	-

Aufgaben:

3-bit binär	GRAY-Code
000	000
001	001
010	011
011	010
100	110
101	111
110	101
111	100

1. ***Informieren Sie sich, wozu der GRAY-Code verwendet wird! Realisieren Sie einen MEALY-Automaten in Form eines Graphen, der einen 3-bit-Binär-Code in GRAY-Code übersetzt!***
2. ***Notieren Sie die Definition des Automaten von Aufgabe 1!***
3. ***Geben Sie die Konfigurations-Folgen für die folgenden Eingaben an!***
   a) 011 b) 110 c) 111 d) 000
4. ***Gegeben ist der folgende Automat.***
   a) Klassifizieren Sie den Automaten nach mindestens 3 Kriterien (zur Einteilung von Automaten)!
   b) Geben Sie eine Definition für den Automaten an! Die Überführungs-Funktionen sind als eigenständige Tabellen anzugeben!
   c) Geben Sie die Konfigurations-Folgen für die folgenden Eingaben an!
   ca) aha cb) haaahaaa cc) ahaaahhah cd) aaahhhaaa



für die gehobene Anspruchsebene:

5. ***Realisieren Sie einen MEALY-Automaten, der einen 4-bit-Binär-Code in GRAY-Code übersetzt!***

später noch Aufgaben vom MOORE übernehmen (wenn dort fertig!)

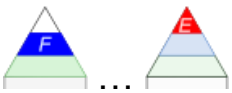
3.2.6.1.2. Simulation eines MEALY-Automaten

MEALY-Automat

AnfangsZustand setzen Z_0
Lesekopf auf 1. Position (X) i=1
Solange x_i existiert
x_i lesen
y_i bestimmen g: $Z \times X \rightarrow Y$
z_i bestimmen f: $Z \times X \rightarrow Z$
i inkrementieren
X- und Y-Band positionieren

Struktogramm für MEALY-Automaten

3.2.6.1.3. Äquivalenz von Zuständen und Automaten



zwei Zustände eines MEALY-Automaten sind äquivalent, wenn sie das Gleiche leisten, d.h. die erweiterte Ausgabe-Funktion beider Zustände gleich ist
die erweiterte Ausgabe-Funktion eines Zustandes umfasst alle (einfachen) Ausgabe-Funktionen für jede mögliche Eingabe dieses Zustandes

zwei MEALY-Automaten mit gleichem Ein- und Ausgabe-Aphabet sind äquivalent, wenn deren Mengen an Zuständen äquivalent sind

3.2.6.1.4. Reduktion / Vereinfachung eines MEALY-Automaten

Komplizierte Automaten oder Maschinen können zumeist reduziert / vereinfacht werden. Dafür gibt es zuverlässige Algorithmen.

Ziel sind einfache und damit unkompliziertere und "billigere" Automaten zu erhalten
ein MEALY-Automat ist reduziert, wenn es in ihm keine zwei Zustände mehr gibt, die äquivalent sind

häufig lassen sich MEALY-Automaten reduzieren, dazu kann der folgende Algorithmus genutzt werden

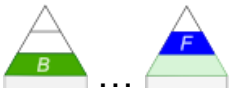
Reduktions-Algorithmus

1. **Sortieren der Zustände nach Ausgabe**
2. **Gruppieren der Zustände nach der Lage der Folge-Zustände**
   (innerhalb gleicher Klassen)
3. **Zusammenfassen und Umbenennen von äquivalenten Zuständen**
4. **Wiederholen ab 2. bis alle Klassen ein-elementig sind**

Aufgaben:

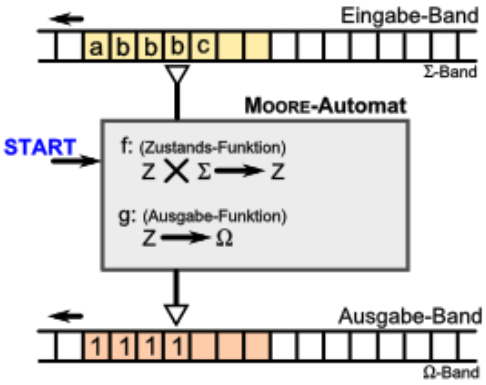
1.
2. *Erstellen Sie einen MEALY-Automaten, der die Zeichen-Folge des Eingabe-Bandes um ein Zeichen versetzt ausgibt. Als erstes Zeichen soll eine Raute (#) ausgegeben werden! Das letzte Zeichen des Eingabe-Bandes geht verloren.*
3.

3.2.6.2. MOORE-Automaten



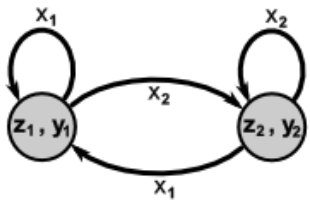
Üblich ist es eigentlich, Automaten mit Ausgaben als Maschinen zu bezeichnen. Leider wird das in der Literatur nicht immer konsequent durchgezogen.
Im englischen Sprachraum wird häufiger von MOORE-Machine's gesprochen und geschrieben.

Als Argument für die Benennung doch als Automat wird oft angebracht, dass die Ausgabe z.B. beim MOORE-Automaten quasi mit dem Zustand verbunden ist. Aus der abgehobenen theoretischen Sicht kommt praktisch nicht wirklich etwas Neues hinzu. Es ist und bleibt ein Zustand, der eben so nebenbei mit einer Ausgabe verbunden ist.
Die endlichen Automaten, bei denen eine Ausgabe nur durch die Zustände bestimmt wird, werden nach ihrem Entwickler MOORE-Automaten genannt.
Edward Forrest MOORE (1925 – 2003) war Professor für Mathematik und Informatik in Amerika. Er gilt als einer der Mitbegründer der Automaten-Theorie.

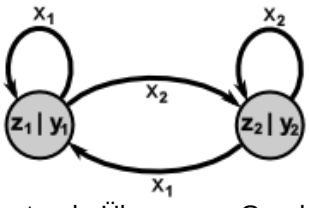


MOORE-Automat
als Transduktor (Übersetzer)

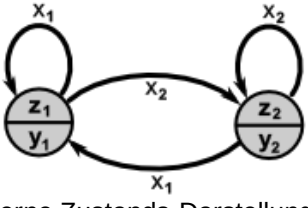
I.A. werden die MOORE-Automaten auch deteministisch ausgeführt, was heißt, dass zu jedem Zustand und für jede mögliche Eingabe einen Folge-Zustand definiert ist. Ein deterministischer MOORE-Automat verfügt also über vollständige Automaten-Tafeln.
Fehlen einzelne Übergänge, dann handelt es sich erst einmal um einen nicht-deterministischen Automaten.In Abhängigkeit von der jeweiligen Eingabe kommt es zum Zustands-Wechsel. Dem neuen Zustand ist jeweils eine Ausgabe zugeordnet. Diese Ausgabe wird im Graphen hinter einem Komma im Zustands-Symbol notiert. Manche Autoren benutzen auch den senkrechten Strich "|" als Trenner.
Die Variablen z bzw. der linke Teil der Zustands-Beschriftung steht für den Zustand. Der rechte Teil – hier die Variablen mit y - steht dagegen für die Ausgabe (des Zustands). Die Übergänge sind immer von den Eingaben abhängig, die hier mit den x-Variablen gekennzeichnet werden.
Ganz moderne Graphen benutzen einen waagerecht geteilten Kreis als Zustands-Symbol. In der oberen Hälfte wird der Zustands-Name – bei uns hier vorrangig z_0 bis z_n notiert. Die untere Hälfte ist der Ausgabe vorbehalten.
Da sich häufig die Bennenungen von Zuständen und die Ausgaben-Symbole stark ähneln, kann diese Sicht etwas unübersichtlich werden. Wir bleiben hier bewusst bei der Schul-üblichen / klassischen Darstellung.
Man kann sich sicher schon denken, dass die Definition eines MOORE-Automaten nicht so wesentlich anders ist, als die eines abstrakten Automaten (→ x.y.z. endliche Automaten).
Hinzukommen muss natürlich das Ausgabe-Alphabet und eine Funktion, welche die Erzeugung der Ausgaben abbildet.



Zustands-Übergangs-Graph
(Komma-Schreibweise,
Ausschnitt)



Zustands-Übergangs-Graph
(Strich-Schreibweise,
Ausschnitt)

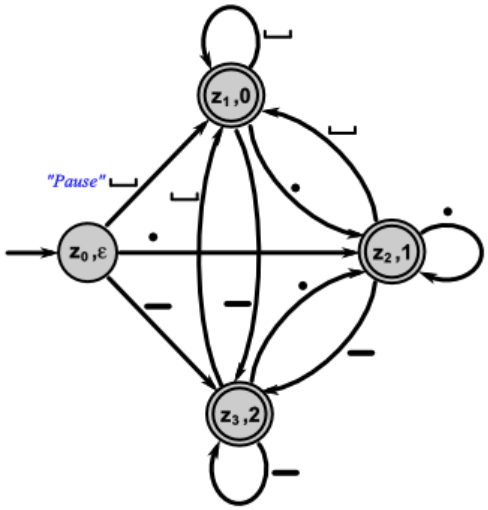


moderne Zustands-Darstellung

Da die Ausgabe-Funktion nicht direkt von der Eingabe abhängig ist, sondern nur vom erreichten Zustand, ist die Ausgabe-Funktion auch denkbar einfach.

Definition(en): MOORE-Automat
Ein MOORE-Automat ist ein endlicher Automat, bei dem die Ausgaben ausschließlich vom jeweiligen Zustand abhängig sind.
Ein MOORE-Automat ist ein Sept-Tupel (7-Tupel) $A = (Z, \Sigma, \Omega, f, g, z_0, Z_E)$ mit: Z … endliche Menge der Zustände; $\|Z\| < \infty$ Σ … Eingabe-Alphabet; $\|\Sigma\| < \infty$, $Z \cap \Sigma = \varnothing$ Ω … Ausgabe-Alphabet; $\|\Omega\| < \infty$ f … Überführungs-Funktion (totale Funktion); $Z \times \Sigma \rightarrow Z$ g … Ausgabe-Funktion; $Z \rightarrow \Omega$ z_0 … Start-Zustand; $z_0 \in Z$ Z_E … endliche Menge der akzeptierten Zustände (End-Zustände); $Z_E \subseteq Z$ Die Menge Z_E kann entfallen (wenn z.B. das Akzeptieren keine Rolle spielt / die reguläre Sprache des Automaten nicht interessiert), dann lässt sich der MOORE-Automat auch als Hex-Tupel (6-Tupel): $A = (Z, \Sigma, \Omega, f, g, z_0)$ defnieren.
Ein MOORE-Automat ist ein endlicher Automat mit (zusätzlichen) Ausgabe(n) (- also ein Transduktor), wobei die Ausgaben durch den Zustand bestimmt sind.

Eine Automaten-Tafel ist auch für jeden MOORE-Automaten erstellbar. Sie ist und bleibt eine der effektivsten Formen der Darstellung der Automaten-Funktionen. Die übliche Zustands-Tabelle wird dazu um eine Spalte erweitert, in welche die Ausgabe zu dem Zustand eingebracht wird.
Sachlich ist es eigentlich eine extra Tabelle für die Ausgabe-Funktion g. Aus Effektivitäts-Gründen werden beide Funktionen gemeinsam notiert. Manche Autoren fassen dies auch in der Funktions-Beschreibung zusammen: aus Z x Σ → Z und Z → Y wird dann Z x Σ → Z,Y.
Betrachten wir einen kleinen Umkodier-Automaten. Dieser soll eine MORSE-Nachricht in eine Computer-lesbare Form übersetzen. Aus dem Punkt "" soll eine "1" für die interne Darstellung werden.



Der Strich "-" wird zur "2" und die Pause "_" als "0" codiert.
Damit haben wir schon wesentliche Teile der Automaten-Definition zusammen:

A = ({z_0, z_1, z_2, z_3}, {_, •, ▬}, {0, 1, 2}, f, g, z_0, {z_1, z_2, z_3}**)**
A = (Z, Σ, Ω, f, g, z_0, Z_E)

Es fehlen (aber) die Konkretisierungen der Funktionen **f** und **g**. Das sind zum Einen die Überführungs-Funktion und zum Anderen die Ausgabe-Funktion. Beide Funktionen lassen sich in Textform oder als Tabellen angeben:

Darstellung aller Übergänge aus dem Graphen:

f = { z_0 X _ → z_1, — *Kante vom Zustand z_0 nach z_1 nach Eingabe von _*
z_0 X • → z_2, — *Kante vom Zustand z_0 nach z_2 nach Eingabe von •*
z_0 X — → z_3, — *Kante vom Zustand z_0 nach z_2 nach Eingabe von —*
z_1 X _ → z_1, — *…*
z_1 X • → z_2,
z_1 X — → z_3,
z_2 X _ → z_1,
z_2 X • → z_2,
z_2 X — → z_3,
z_3 X _ → z_1,
z_3 X • → z_2,
z_3 X — → z_3 }

Für die Ausgabe-Funktion ist die Beschreibung deutlich einfacher, den die Ausgabe ist nur vom erreichten Zustand abhängig:

Darstellung aller Ausgabe aus dem Graphen:

f = { z_0 → e, — *im Zustand z_0 wird das leeres Wort / Zeichen ausgegeben*
z_1 → 0, — *im Zustand z_0 wird eine 0 ausgegeben*
z_2 → 1, — *im Zustand z_0 wird eine 1 ausgegeben*
z_3 → 2 } — *…*

In der ausführlichen Tabellen-Form – also den zwei Automaten-Tafeln würde man die nebenstehende Darstellung erhalten.
Man sieht schnell, dass die zweite Tabelle viele Elemente der ersten Tabelle enthält, die nur Schreibaufwand darstellen. Zusammengefasst ergibt sich die untere Tabelle mit beiden Funktionen.
Mit dieser gemeinsamen Tabelle wird bei den meisten Funktions-Beschreibungen auch gearbeitet.
Da die Tabellen-Form deutlich übersichtlicher sind, nutzt man sie auch in Kombination mit der obigen Tupel-Angabe zur Definition eines MOORE-Automaten.

f:

Zust.	_	•	—
z_0	z_1	z_2	z_3
z_1	z_1	z_2	z_3
z_2	z_1	z_2	z_3
z_3	z_1	z_2	z_3

g:

Zust.	Ausg.
z_0	ε
z_1	0
z_2	1
z_3	2

f,g:

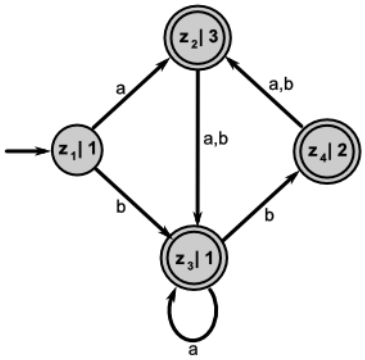
	_	•	—	Ausg.
z_0	z_1	z_2	z_3	ε
z_1	z_1	z_2	z_3	0
z_2	z_1	z_2	z_3	1
z_3	z_1	z_2	z_3	2

MEALY- und MOORE-Automaten sind leicht zu verwechseln. Als kleine Eselsbrücke bietet sich an, zu merken, dass beim MOORE-Automaten **mehr** (***more***) **im Zustand** steht.
Eine andere Eselsbrücke könnte sein, dass sich in MOOR**E** das **E** hinten befindet, also erst im Zustand auftaucht. Dagegen sind bei M**EA**LY das **E** und **A** – für **E**ingaben und **A**usgaben – zusammen, also was auf die Kante kommt.
Für die etwas freakigeren Leser bietet sich auch eine Brücke über die Ausgabe-Funktion an. Beim **einsilbig**en MOORE ist die Ausgabe-Funktion nur von **ein**em **Argument** – den Zuständen – abhängig (Z → Y) . Dagegen braucht man für den **zweisilbig**en MEALY(-Automaten) auch **zwei Argumente** – die Zustände und die Eingaben (Z x E → Y).

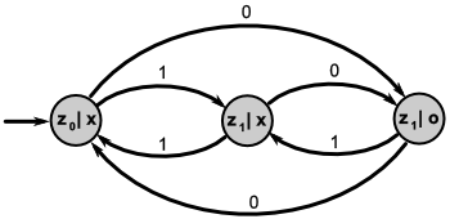
??? prüfen
Einen Sonderfall des MOORE-Automaten stellt der MEDWEDJEW-Automat dar. Bei ihm ist der aktuelle Zustand gleichzeitig der Ausgabe-Wert des Automaten.

Aufgaben:

1. *Gegeben ist der nebenstehende Graph eines MOORE-Automaten. Geben Sie die Tupel-Definition des Automaten an!*



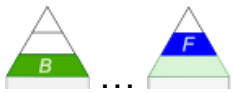
2. *Zum folgenden Automaten ist eine Definition anzugeben!*



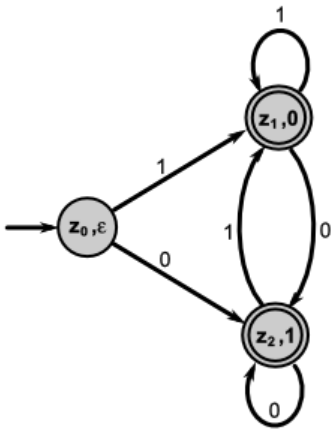
3. *Erstellen Sie aus der folgenden Definition eines MOORE-Automaten einen Graphen!*

A = ({z_0, z_1, z_2, z_3}, {0, 1}, {0, 1}, {z_0 X 0 → z_2, z_0 X 1 → z_1, z_1 X 0 → z_3,
z_1 X 1 → z_0, z_2 X 0 → z_0, z_2 X 1 → z_3, z_3 X 0 → z_3, z_3 X 1 → z_3, },
{ z_0 → 0, z_0 → 1, z_0 → 1, z_0 → 1}, z_0**)**

3.2.6.2.1. Arbeit eines MOORE-Automaten



Wir wählen hier als Beispiel einen relativ einfachen, übersichtlichen Automaten, damit wir nicht unendlich viel zu Schreiben haben. Der Automat soll aus einer eingegeben 1 eine 0 machen und umgekehrt. Wir sprechen auch vom Invertieren. Zum Vergleich haben wir die gleiche Funktion auch schon vorne (→ x.y.z.1.1. Arbeit eines MEALY-Automaten) durch einen MEALY-Automaten realisiert und dessen Arbeits-Ablauf besprochen.
Die Definition unseres MOORE-Automaten könnte in etwa so so aussehen:



$A = (\{z_0,z_1,z_2\}, \{0,1\}, \{\varepsilon,0,1\}, f, g, z_0, \{z_1,z_2\})$
A = (Z , S , W , f , g , z0 , Z)

Interessant ist, dass sich bis hierhin die Grob-Definitionen von MEALY- und MOORE-Automat gleichen.
Die Funktionen **f** und **g** sind in separaten Tabellen gegeben. Das erleichtert uns das Heraussuchen der passende Werte für die Notierung im Protokoll.

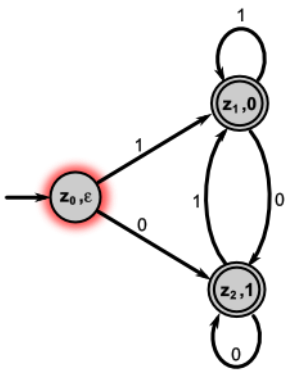
f:	Σ	
Zust.	**0**	**1**
z_0	z_2	z_1
z_1	z_2	z_1
z_2	z_2	z_1

g:	
Zust.	**Ausg.**
z_0	ε
z_1	0
z_2	1

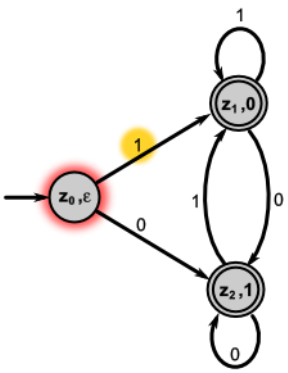
Aufgaben:

1. *Erläutern Sie die Definition des obigen Automaten!*
2. *Schreiben Sie die Funktionen in eine gemeinsame Tabelle! Ist das überhaupt zulässig?*
3. *Notieren Sie die Definition des obigen Automaten in reiner Text-Form (also ohne Tabelle(n))!*

Zum Nachverfolgen sei auf dem Eingabe-Band die Zeichen-Kette 01101 gegeben.
(Die farbliche Untermalung für Eingabe- und Ausgabe-Band wird hier nur zu besseren Unterscheidung gemacht.)

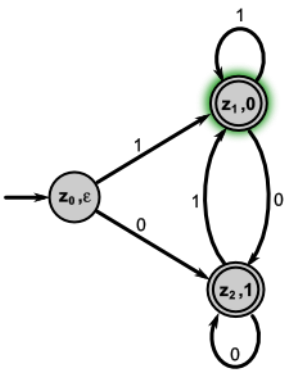


1



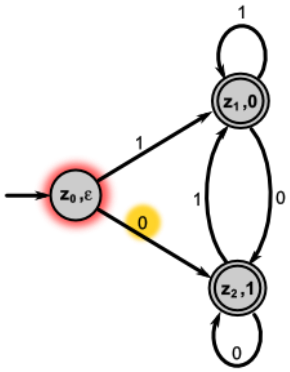
2

Wäre das erste Zeichen auf dem Eingabe-Band eine 1, dann würde der obere Übergang zu z_1 genutzt werden.



3

In dem Fall ergäbe sich die – zu z_1 gehörende – Ausgabe 0.

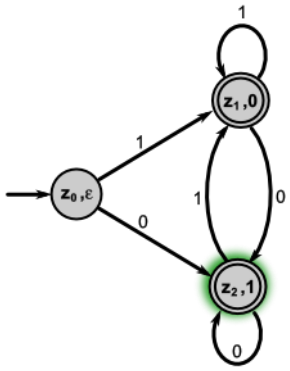


4

Das erste Zeichen ist aber eine 0 (**0**1101), so zum Zustand z_2 gewechselt wird. Für das Protokoll notieren wir (noch für z_0):

Start → z_0 ; z_0 → ε (f;g)

Auf das Ausgabe-Band wird ein ε – also ein leeres Zeichen – geschrieben.

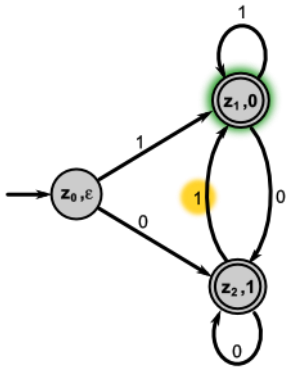


5

Mit dem Erreichen von z_2 kommt es zur Ausgabe von 1 auf dem Ausgabe-Band (**1**). Jetzt kommen die Übergänge und Anzeigen:

z_0 x 0 → z_2 (f)
z_2 → 1 (g)

ins Protokoll. Jetzt ist auch das erste echte Zeichen auf dem Ausgabe-Band zu sehen: ε**1**

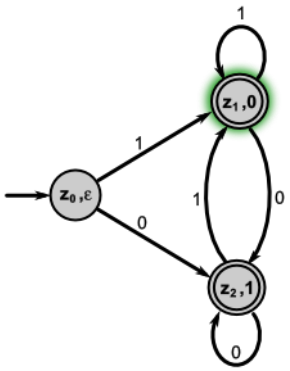


6

Es folgt als nächste Zeichen im Beispiel eine 1 (0**1**101). Der Automat wechselt also zum Zustand z_1 verbunden mit der Ausgabe einer 0.

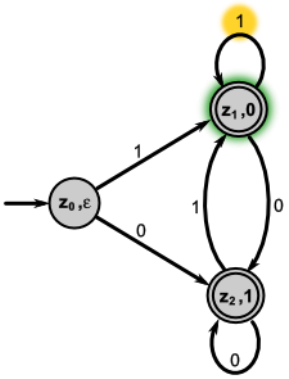
z_2 x 1 → z_1 (f)
z_1 → 0 (g)

Damit steht auf dem Ausgabe-Band nun die Zeichenkette: ε10



7

Der Automat verharrt nun im Zustand z_1, bis eine neue Eingabe vom Eingabe-Band kommt.

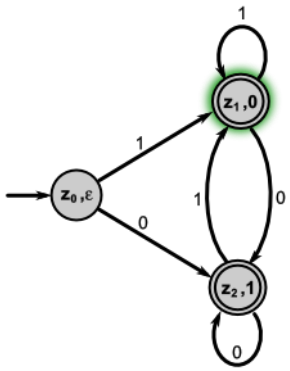


8

Wenn dass, wie im Beispiel, eine 1 (01101) ist, dann ergeben sind die Funktionen:

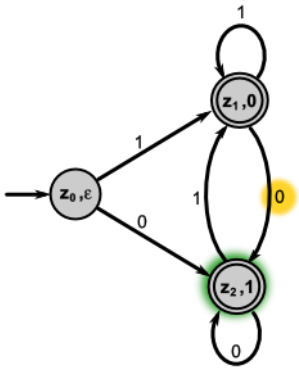
z_1 x 1 → z_1 (f)
z_1 → 0 (g)

und der Automat verbleibt also im Zustand z_1. Trotzdem wird erneut eine 0 (ε100) ausgegeben.



9

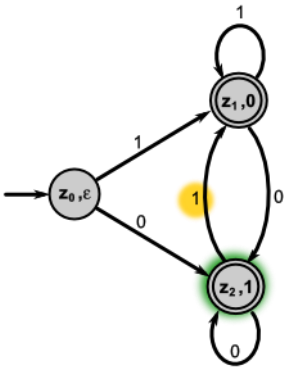
Und wieder ist der im Automat im Zustand z_1. Hier kann man schön sehen, dass aus der Situation heraus nicht zuerkennen ist, wie dieser Zustand erreicht wurde. Der Automat hat kein Gedächtnis oder eine eingebaute Verlaufs-Speicherung.



10

Mit der Eingabe einer 0 (0110**0**1) ergibt sich wieder ein Übergang zu z_1 und der Ausgabe einer 1 (ε1001**1**)

z_1 x 0 → z_1 (f)
z_2 → 1 (g)

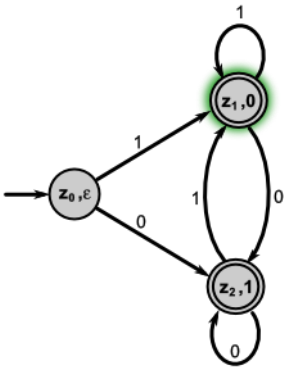


11

Als letztes Zeichen haben wir noch eine 1 auf dem Σ-Band. Das bewirkt:

z_2 x 1 → z_1 (f)
z_1 → 0 (g)

Die Zeichenkette auf dem Ω-Band ist nun: ε10010 zu sehen. Sie stellt wirklich die Invertierung von 01101 dar.



12

Damit haben wir den letzten Zustand erreicht. Da dieses ein Endzustand ist, hat der Automat die Eingabe-Zeichen-Folge auch akzeptiert, was aber bei Transduktoren weniger interessiert.
Der Automat hätte auch ohne die beiden Endzustände z_1 und z_2 die gleiche Übersetzungs-Leistung gebracht.

Zusammengezogen ergibt sich für die Eingabe-Zeichenfolge: 01101 das nebenstehende Ablauf- oder Automaten-Protokoll.
Um die Arbeit des Automaten zu dokumentieren benötigt man die Konfiguration des Automaten. Diese ist bei einem MOORE-Automaten von einem Zustand (mit verbundener Ausgabe) und der folgenden Eingabe charakterisiert.
Auf die Tabellen-Form und die Schritt-Nummerierung kann auch noch verzichtet werden, so dass man das Protokoll (Konfigurations-Folge) auch als reinen Text:

	f	g
Start	z_0	z_0 → ε
1	z_0 x 0 → z_2	z_2 → 1
2	z_2 x 1 → z_1	z_1 → 0
3	z_1 x 1 → z_1	z_1 → 0
4	z_1 x 0 → z_2	z_2 → 1
5	z_2 x 1 → z_1	z_1 → 0

K = ($[(z_0,\varepsilon), 0] \vdash [(z_2,1), 1] \vdash [(z_1,0), 1] \vdash [(z_1,0), 0] \vdash [(z_2,1), 1] \vdash [(z_1,1), -]$)

schreiben kann.

Aufgaben:

1. ***Erstellen Sie das Protokoll für die Eingabe von 1110010!***
2. ***Gegeben ist das nochfolgende Protokoll. Rekonstruieren Sie daraus sowohl das Eingabe- als auch das endgültige Ausgabe-Band!***

   $K = ([(z_0,\varepsilon), 0] \vdash [(z_2,1), 1] \vdash [(z_1,0), 1] \vdash [(z_1,0), 0] \vdash [(z_2,1), 1] \vdash [(z_2,1), 1] \vdash$
   $[(z_1,0), 1] \vdash [(z_1,0), 0] \vdash [(z_2,1), 1] \vdash [(z_2,1), 1] \vdash [(z_1,1), -])$

3. ***Kann man aus dem nun nachfolgend gegebene Protokoll den zugehörigen (deterministischen) MEALY-Automaten ableiten? Wenn JA, dann machen Sie das, ansonsten geben Sie an, warum das nicht bzw. bis wo das geht!***

für die gehobene Anspruchsebene:

3. ***Könnte die Eingabe-Zeichenfolge eigentlich auch $\varepsilon 101\varepsilon 00$ lauten? Begründen Sie Ihre Bewertung!***

3.2.6.2.2. Simulation eines MOORE-Automaten

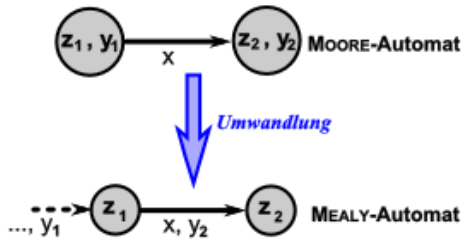
MOORE-Automat

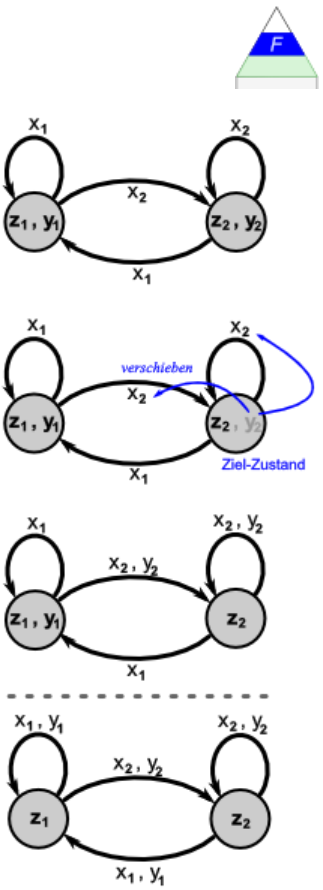
AnfangsZustand setzen Z_0
Zeichen für Z (y) auf Y-Band schreiben
Lesekopf auf 1. Position (X) i=1
Solange x_i existiert
x_i lesen
z_i bestimmen f: $Z \times X \rightarrow Z$
Y-Band positionieren
y_i bestimmen g: $Z \rightarrow Z$
X-Band positionieren
i inkrementieren

Struktogramm zur Simulation eines MOORE-Automaten

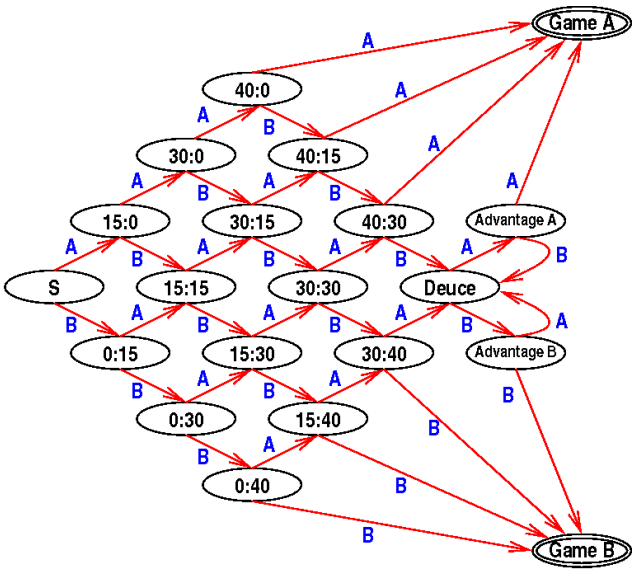
3.2.6.2.3. Überführung eines MOORE- in einen MEALY-Automaten

Jeder MOORE-Automat kann in einen Automaten überführt werden, der die Ausgabe während des Zustands-Übergangs (/ der Transition) macht. Diese Automaten heißen nach ihrem Erfinder MEALY-Automat.
Die Überführung ist sehr leicht. Man **"verschiebt"** die Ausgabe des Folge-Zustangs an die Transition. Es ist ja offensichtlich auch egal, ob die Ausgabe erst erfolgt, wenn der Zustand erreicht ist, oder wie jetzt angestrebt schon während des Zustands-Übergangs.
Wurden alle Transitionen dahingehend angepasst, dann kann die Ausgabe im Zustand (- also das MOORE-Relikt -) gelöscht werden.
MOORE-Automaten können mittels digitaler Schaltungs-Technik realsiert werden. Sie sind dann häufig Minimal-Bauelemente für komplexere Schaltungen.





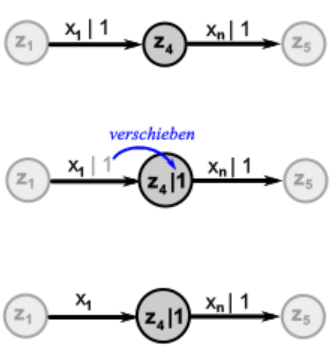
Überführung eines MOORE-Automaten(-Auschnitts) in einen MEALY-Automaten(-Ausschnitt)
oben: Schritte für einen Zustand (z_2,y_2 ⇨ z_2)



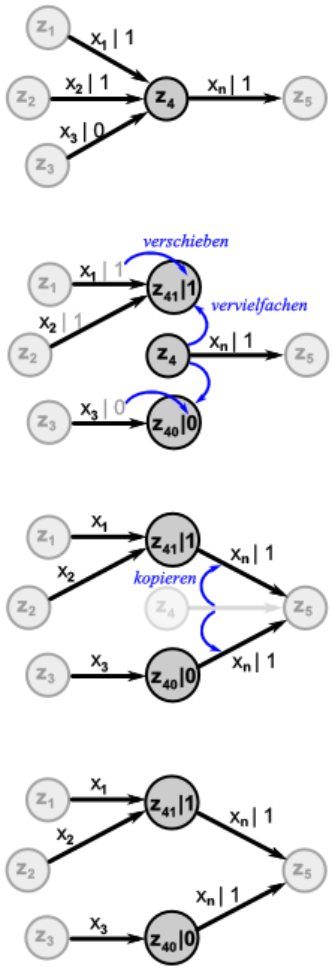
Ein Spiel Tennis als MOORE-Automat
Q: http://www.mathematik.uni-ulm.de/sai/ws04/prog/slides/automata-19.html

3.2.6.4.4. Überführung / Umwandlung eines MEALY- in einen MOORE-Automaten

MEALY- und MOORE-Automaten lassen sich ineinander überführen und sind damit gleichwertig.
Während die Umwandlung eines MOORE- in einen MEALY-Automaten sehr leicht ist, kann die entgegengesetzte Überführung etwas komplizierter werden.
Die Ausgaben werden in die Folge-Zustände (hier z_4) verschoben. Der Folge-Zustand erhält also die Ausgabe und die Transition verliert ihre. Wenn man sich die Arbeit des Automaten(-Stück's) vorstellt, dann ist ja auch praktisch egal, ob die Ausgabe während des Übergangs oder erst im erreichten Folge-Zustand erledigt wird.

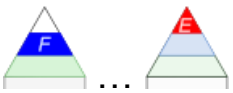


Wenn allerdings unterschiedliche Ausgaben zu einem Folge-Zustand (hier: z_4) hin auftauchen – z.B. von zwei unterschiedlichen Transitionen (Kanten, Übergängen) – dann muss der Folge-Zustand aufgespalten werden (hier z_{41} u. z_{40}).
Jeder neue Folge-Zustand erhält die Kanten, welche die ausgewählte Ausgabe des neuen Folge-Zustands haben. Man sagt auch: "Die Transitionen werden umgehängt." Wenn dies für alle Ausgaben erledigt ist, dann sollte der alte Folge-Zustand (z_4) keine zuführenden Kanten mehr haben.
Alle abgehenden Kanten des alten Folge-Zustands werden auf den neuen Folge-Zustand kopiert. Am Schluß kann der alte Folge-Zustand (und die als Kopier-Vorlagen dienen abgehenden Kanten) entfernt werden. Der alte Folge-Zustand (z_4) ist ja jetzt praktisch nicht mehr erreichbar!
Im nächsten Schritt wird dann mit dem nächsten Zustand weitergemacht – z.B. mit dem Zustand z_5 und seinen Zugängen.

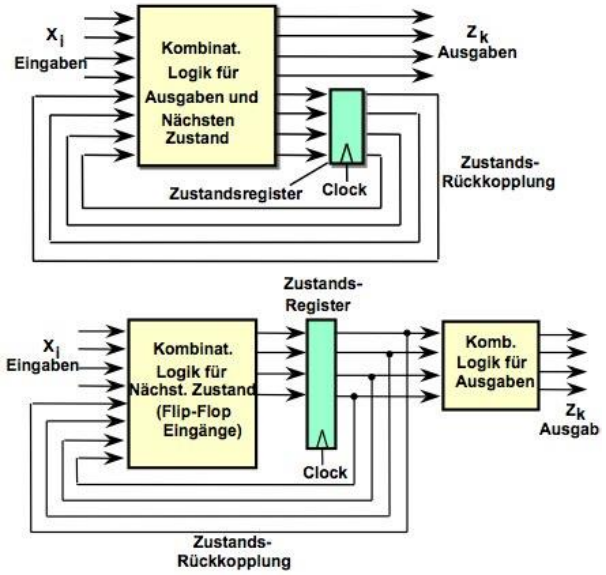


Ein MEALY- und ein MOORE-Automat mit jeweils gleichen Ein- und Ausgabe-Alphabeten sind dann gleichwertig (haben die gleiche Leistung), wenn die Mengen der Leistungen ihrer Zustände gleich sind.
Zu jedem MEALY-Automaten mit n Zuständen und m Eingabe-Symbolen gibt es einen MOORE-Automaten mit n(m+1) Zuständen.

3.2.6.4.5. MEALY- und MOORE-Automaten in der Logik-Entwicklung



MEALY- und MOORE-Automaten bilden die Grundlage für die Entwicklung von Logik-Schaltungen bis hin zu Mikropozessoren.
Hier sieht die eher technische Betrachtung etwas anders aus.
Beim genaueren Hinsehen erkennen wir aber die Charakteristika wieder. Der MEALY-Automat produziert aus seine Verarbeitungs-Logik heraus. Er erzeugt seine Ausgaben ja praktisch zwischen den Zuständen / mit dem Zustandswechsel.
Der MOORE-Automat erzeugt die Ausgaben erst nach seiner internen Verarbeitung der Informationen (Eingänge und internen Zustände). Seine Ausgaben-Logik ist deshalb auch separat ausgeführt und praktisch nur von den Zuständen abhängig. Eine direkte Abhängigkeit von den Eingaben besteht nicht.



Q: https://www.fachschaft.informatik.tu-darmstadt.de/forum/viewtopic.php?t=11779

3.2.6.3. Erstellen eines Automaten (mit Anzeige)

aus: http://www.iwi.hs-karlsruhe.de/~lino0001/skripte/Automatisierungsprojekte/FolienPDF/VorlesungAuto2_4.pdf (geändert: dre)

Problem- / Aufgaben-Stellung

Gegeben ist eine Digital-Uhr, die über zwei Druck-Knöpfe eingestellt werden kann. Der linke Knopf dient zum Wechseln zwischen dem Stunden-, Minuten- und Sekunden-Einstellen (Ziffern blinken dann). Mit dem rechten Knopf wird die blinkende Zahl immer um 1 erhöht, bis der linke Knopf gedrückt wird.
Ist die letzte Einstellung beendet, dann geht die Uhr in die normale Zeitanzeige über.

Festlegen der Modell-Objekte / Modellwelt

Analyse / Festlegen von Zuständen:

- Zustand0 (z_0): Normalzeit-Anzeige (z.B. nach Einlegen der Batterie)
- Zustand1 (z_1): Stunden-Stellen (Stunden-Ziffern blinken)
- Zustand2 (z_2): Minuten-Stellen (Minuten-Ziffern blinken)
- Zustand3 (z_3): Sekunden-Stellen (Sekunden-Ziffern blinken)

Analyse / Festlegung möglicher Eingaben:

- Ereignis1 (S): Start-Signal (z.B. nach Einlegen der Batterie)
- Ereignis2 (R): Knopf links gedrückt
- Ereignis3 (L): Knopf rechts gedrückt

Analyse / Festlegung der Anzeigen:

- Anzeige1 (UZ): Uhrzeit-Anzeige (z.B. nach Einlegen der Batterie: 00:00:00)
- Anzeige2 (StB): Stunden-Ziffern blinken
- Anzeige3 (St+): Stunden um 1 erhöhen
- Anzeige4 (MiB): Minuten-Ziffern blinken
- Anzeige5 (Mi+): Minuten um 1 erhöhen
- Anzeige6 (SeB): Sekunden-Ziffern blinken
- Anzeige7 (Se0): Sekunden auf 00 setzen

Analyse / Festlegung der Übergänge

Da die Knöpfe quasi mehrfach belegt sind muss zur eindeutigen Beschreibung des Übergangs und der passenden Anzeige auch die aktulle Situation (Zustand) beachtet werden.

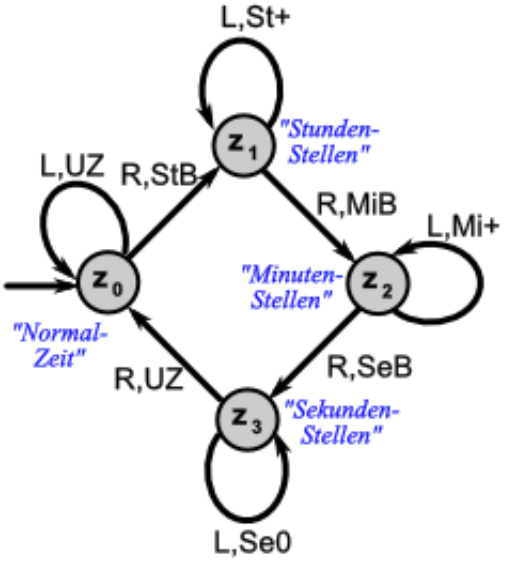
akt. Situation				Übergänge / Anzeigen
z0				• Wenn Batterie eingelegt wird (Ereignis1 → S), dann Normalzeit-Anzeige (UZ, hier: 00:00:00) (Zustand0 → z_0)
				○ Wenn Knopf rechts gedrückt wird (Ereignis2 → R), dann Stellen der Stunden (Blinken der Stunden-Ziffern) (Anzeige2 → StB)
	z1			▪ Wenn Knopf links gedrückt wird (Ereignis3 → L), dann Stunden-Zahl um 1 erhöhen (Anzeige3 → St+)
				▪ Wenn Knopf rechts gedrückt wird (Ereignis2 → R), dann Stellen der Minuten (Blinken der Minuten-Ziffern) (Anzeige4 → MiB)
		z2		• Wenn Knopf links gedrückt wird (Ereignis3 → L), dann Minuten-Zahl um 1 erhöhen (Anzeige5 → Mi+)
				• Wenn Knopf rechts gedrückt wird (Ereignis2 → R), dann Stellen der Sekunden (Blinken der Sekunden-Ziffern) (Anzeige6 → SeB)
			z3	○ Wenn Knopf links gedrückt wird (Ereignis3 → L), dann Sekunden-Anzeigen nullen (Anzeige7 → Se0)
				○ Wenn Knopf rechts gedrückt wird (Ereignis2 → R), dann Anzeige der Normal-Zeit (jetzt: gestellte Zeit) (Anzeige1 → UZ)
				○ Wenn Knopf links gedrückt wird (Ereignis3 → L), dann verbleibt die Normalzeit-Anzeige (jetzt: aktuelle Zeit)

Erstellen der Übergangs- und Anzeige-Funktionen

Zustand	**Überführungs-Funktion nach Eingabe von …**		**Ausgabe-Funktion nach Eingabe von …**		
	Knopf **L**	Knopf **R**	Knopf **L**	Knopf **R**	
z_0 "Normal-Zeit"	z_0	z_1	UZ	StB	
z_1 "Stunden-Stellen"	z_1	z_2	St+	MiB	
z_2 "Minuten-Stellen"	z_2	z_3	Mi+	SeB	
z_3 "Sekunden-Stellen"	z_3	z_0	Se0	UZ	

Erstellen eines Graphen

Ob man die Tabellen oder den Graphen zuerst zeichnet ist egal, da beide gleichwertig sind.
Sie beschreiben die selben Übergänge und Ausgaben.
Da die Ausgaben (Anzeigen) von den aktuellen Zuständen und den Eingaben abhängen, handelt es sich also um eine MEALY-Automaten.



Aufgaben:

1. *Ein Programmierer moniert die gemeinsame Darstellung der Überführungs- und Anzeige-Funktion in einer Tabelle. Er braucht zwei unabhängige Tabellen für seinen programmierten Automaten. Helfen Sie ihm!*
2. *Wandeln Sie den MEALY-Automaten ("Digital-Uhr") in einen MOORE-Automaten! (Geht das überhaupt? Begründen Sie!)*
3. *Geben Sie die Definition des MEALY-Automaten ("Digital-Uhr") in reiner Text-Form an!*

3.2.6.4. weitere Zustands-Automaten mit Anzeige

3.2.6.4.1. HAREL-Automat

Hybrid aus MEALY- und MOORE-Automat
Zustands-Übergänge abhängig von Eingabe und Bedingungen / einer Wächter-Funktion
Zustände können / sind geschachtet (hierarchisch) gruppiert
die Zustände haben ein Gedächtnis
es gibt nebenläufige Prozesse (es existieren u.U. Ober-Zustände, die sich gleichzeit in einem oder mehreren Unter-Zuständen befinden können

Zeit-(Takt-)Steuerung ist weitgehend aufgelöst, Automat reagiert unmittelbar / sofort

HAREL hat zusätzliche Strukturen und Notations-Elemente in sein Automaten-Verständnis eingebracht
dazu gehören:

- Hierarchien mit Unterzustands-Automaten
- Komposition zur Darstellung von parallelen Zustands-Automaten
- Inter-Level-Transitionen (Übergänge über Hierarchie-Ebenen hinaus)
- History-Konnektor (Zustands-Speicher)
- temporale Logik (für Transitition mit Timeout's)
- Entry-, Exit- und Throughout- bzw. Do-Aktionen

mit ihren Möglichkeiten sind HAREL-Automaten als sehr universell anzusehen

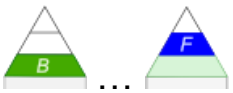
3.2.7. Automaten ohne Ausgabe – Akzeptoren

Als Quasi-Gegenstück zu den Automaten mit Ausgabe kann man solche ohne eine solche sehen. Wir sprechen von Akzeptoren, da sie lediglich eine Eingabe prüfen und dann letztendlich akzeptieren oder ablehnen.

Definition(en): Akzeptoren
Ein Akzeptor ist ein endlicher Automat der eine Sequenz von Eingabe-Symbolen akzeptiert oder eben nicht..
Akzeptoren sind abstrakte Automaten ohne eine Ausgabe-Funktion.

Die Akzeptanz bzw. eben die Nicht-Akzeptanz wird nicht als Anzeige verstanden, sondern mehr als Außen-Wirkung des Gesamtsystems. Dabei ist es von außen nicht möglich, festzustellen, welcher Zustand für die Akzeptanz bzw. Nicht-Akzeptanz "verantwortlich" ist. Bei Betrachtungen der Abläufe im Automaten sehen wir das natürlich genau.
Grundsätzlich werden zwei große Klassen von Automaten bei den Akzeptoren unterschieden. Die Klassifizierung ist klar gegeben. Die eine Klasse sind die deterministischen Automaten (→ x.y.z.1. deterministische endliche Automaten) und die andere die nicht-deterministischen (→ x.y.z. nicht-deterministische endliche Automaten). Was sich allerdings für die hierrarchische Ordnung beider Klassen zueinander ergibt, ist nicht so absolut festzumachen. Dazu gleich (→ x.y.z.1. deterministische endliche Automaten) bzw. später (→ x.y.z. nicht-deterministische endliche Automaten) mehr.

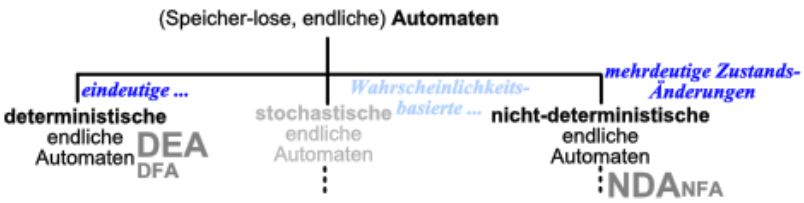
3.2.7.1. deterministische endliche Automaten



Mit der Gruppe der deterministischen Automaten greifen wir uns eine recht umfangreiche Klasse aus der Unzahl von Automaten heraus. Wie wir schon erläutert haben, sind sie als endliche Automaten durch eine begrenzte Anzahl (abgezählte Menge) von Zuständen charakterisiert. Bei den deterministischen Automaten darf immer nur ein Start-Zustand definiert sein und es muss für jeden Zustand und für jedes Eingabe-Symbol eindeutig geklärt sein, welcher Zustand als nächstes folgt. Mit anderen Worten darf es in der Überführungs-Funktion (z.B. in der Tabellen-Form) keine Lücken geben.
In der Praxis haben sich die Abkürzungen DEA aber auch DA eingebürgert. Die englische Bezeichnung lautet: deterministic finite automaton (DFA).

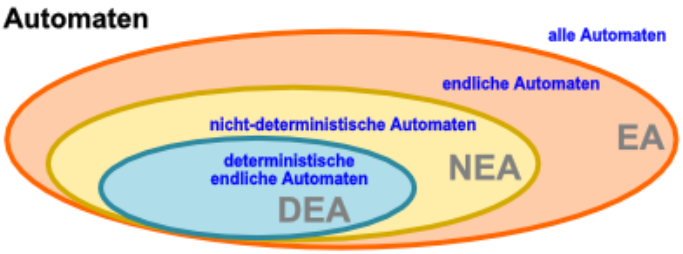
ein DEA besitzt immer genau einen Start-Zustand und mindestens einen End-Zustand
Automat ist immer nur in genau einem Zustand
es treten nur definierte und reproduzierbare Folgezustände ein
ausgehend von einem bestimmten Zustand tritt nach einer definierten Eingabe immer ein klar eingegrenzter Folgezustand ein
bei DEA's ist die Abbildung $Z \times X$ in Z ist und Z_A ein-elementig

Die DEA's können als Spezialfall der nicht-deterministischen endlichen Automaten (NEA) (→ x.y.z. nicht-deterministische endliche Automaten) gefasst werden. Bei einer anderen Herangehensweise stellen sie auch eine Alternative zu den nicht-deterministischen endlichen Automaten dar.

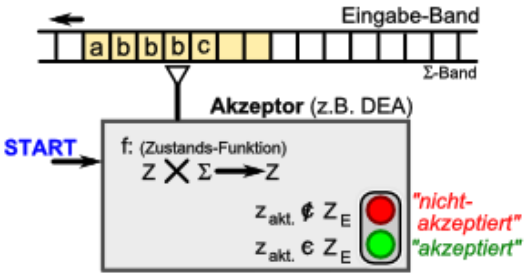


s.a. große Einteilung (→ x.y.z. Einteilungen)

Je nachdem, wie man das Unterscheidungs-Merkmal festlegt. Geht man Mengen-bezogen vor, dann stellen die DEA's eine eingeschlossene Menge in den NEA's dar.
Das dies nur die halbe Wahrheit ist, werden wir später bei den Nicht-deterministischen Automaten noch genau sehen.



Deterministische endliche Automaten lassen sich leicht aus MOORE-Automaten ableiten. Dabei entfällt die Ausgabe, genauso wie die dann nicht mehr benötigte Ausgabe-Funktion **g**.

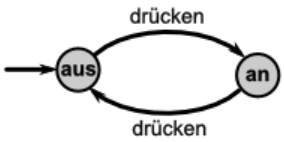


(deterministischer) endlicher Automat
als Akzeptor (Überprüfer)

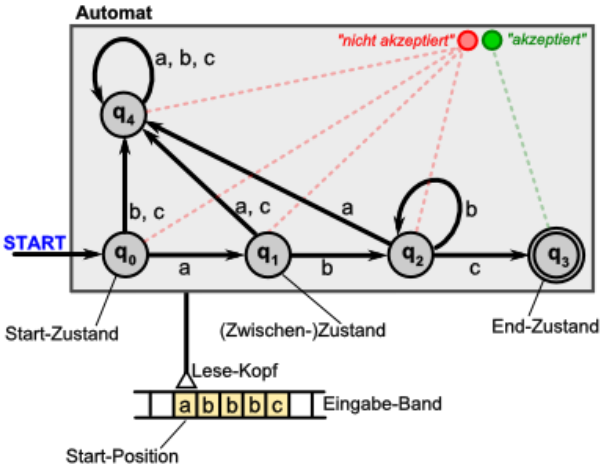
Endzustände werden mit Doppel-Kreisen gekennzeichnet stellt quasi eine innere Anzeige dar
und jeder DEA muss auch mindestens einen End-Zustand besitzen

die akzeptierte Sprache besteht aus allen Wörtern bei denen der Automaten (mindestens) einen (der möglichen) End-Zustand einnimmt

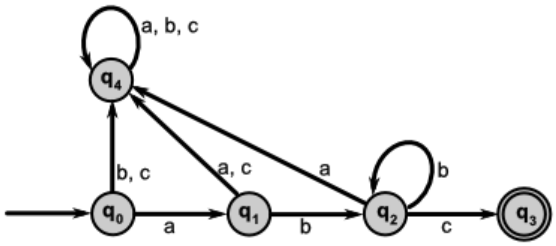
jede Eingabe führt zu einem definierten Folge-Zustand; ε-Übergänge sind nicht gestattet
somit muss für jeden Zustand auch für jede Eingabe eine zu einem Zustand führende Übergangs-Funktion definiert sein



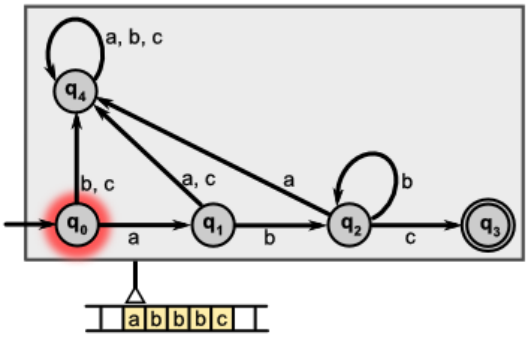
Modell-Umsetzung eines endlichen Automaten besteht aus einem Eingabe-Band mit einzelnen Speicherfeldern für die Symbole der Sprache
der Automat hat einen Lese-Kopf, der auf der Start-Position des Bandes liegt und das erste Symbol (des Eingabe-Wortes) lesen kann
der Automat selbst befindet sich im vordefiniertem Ausgangs-Zustand



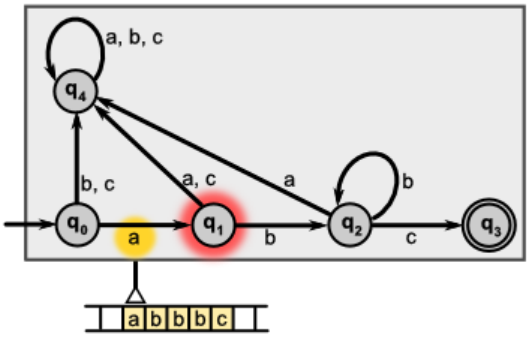
in der Praxis findet man meist nur die abstrakte Version des Automaten, indem nur noch die Zustände und die Überführungs-Funktionen notiert sind
Man nennt die Darstellung allgemein auch **Zustands-Diagramm**, welches durch Zustände und Kanten gekennzeichnet ist.



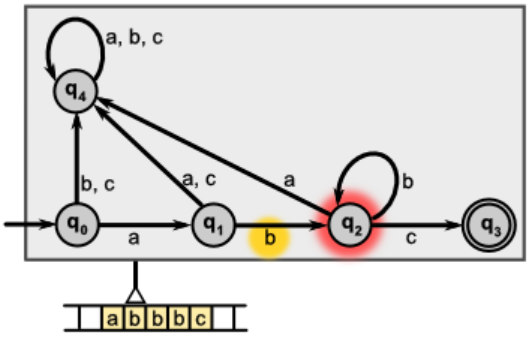
Am Anfang befindet sich der Automat im Ausgangs-Zustand. Dieser ist immer gut dadurch erkennbar, das er einen hinweisenden Pfeil ohne vorherigen Zustand hat und auch ohne Überführungs-Funktion auskommt.
In den folgenden Abbildungen wird der jeweils aktive (momentane) Zustand durch einen umgebenden Schein gekennzeichnet.

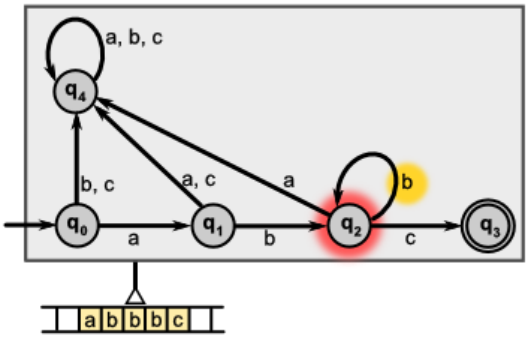


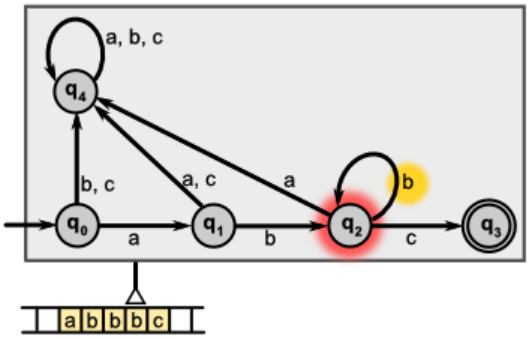
Ausgehend von einem Zustand wird jetzt immer die aktuelle Position des Eingabe-Bandes abgelesen und der passende Übergang gewählt. Es ergibt sich hier der Zustand "q1", der einen nicht-akzeptierenden Zustand darstellt.
Wäre in unserem Beispiel ein "b" oder ein "c" auf dem Band, dann hätte der Übergang zu Zustand "q4" stattgefunden
für alle möglichen Symbole muss es Überführungs-Funktionen geben, mit anderen Worten es muss von jedem Zustand einen definierten Folgezustand aufgrund des gelesenen Symbols geben
bei jedem weiteren Arbeitsschritt wird das Band eine Position nach rechts bewegt und es gibt wieder einen Zustandswechsel im Automaten in Abhängigkeit vom gelesenen



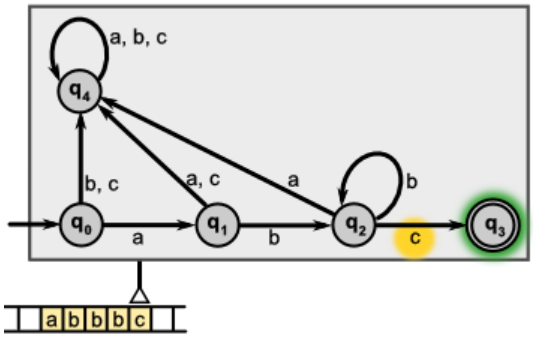
Ein Automat kann bei entsprechender Eingabe in seinem momentanen Zustand verharren. Das er arbeitet, wird aber zumindestens am Weiterwandern des Eingabe-Bandes sichtbar.







Mit der nächsten Lese-Operation bekommt der Automat ein "c" vorgesetzt. Damit ist ein Übergang zu Zustand "q3" möglich. Der Automat hat einen End-Zustand erreicht. Die Symbol-Kette vom Eingabe-Band "abbbc" wurde vom Automaten akzeptiert.



Überführungs-Funktionen werden auch gerne in Form von Tabellen dargestellt das unterstützt auch die Simulation in Computer-Programmen
üblich ist die Notation der Ausgangs-Zustände als Zeilen-Kopf ganz links (1. Spalte) und die Folge-Funktionen in Abhängigkeit von den Eingabe-Symbolen (Spalten-Köpfe). Die Folge-Funktion steht immer in der resultierenden Zelle.

Abbildung der Folge-Zustände (Überführungs-Funktion)

(aktueller) Zustand	Eingabe-Symbole				
	a_0	**a_1**	**a_2**	**…**	**a_n**
q_0	q_1	q_4	…	…	…
q_1	q_1	…	q_2	…	…
q_2	…	q_n	…	…	…
…	…	…	…	…	…
q_n	…	…	…	…	…

Für den oben gezeichneten Automaten würde die Überführungs-Tabelle dann so aussehen.
Aus dem aktuellen Zustand (oranger Hintergrund) und dem zu verarbeitenden Eingabe-Symbol (gelblicher Hintergrund) ergibt sich der nächste Zustand (weißer Hintergrund).
Da der Zustand q_3 ein finaler Zustand (grüner Hintergrund) ist, gibt es für ihn keine Folge-Zustände.
Betrachten wir nochmal die Abarbeitung des Eingabe-Wortes **abbbc**.

Abbildung der Folge-Zustände (Überführungs-Funktion)

(aktueller) Zustand	Eingabe-Symbole		
	a	**b**	**c**
q_0	q_1	q_4	q_4
q_1	q_4	q_2	q_4
q_2	q_4	q_2	q_3
q_3	-	-	-
q_4	q_4	q_4	q_4

Ein Situation bestehend aus aktuellem Zustand und dem zulesenden Eingabe-Symbol wird **Konfiguration k** genannt. Sie stellen ein Tupel (2-Tupel) dar.
Eine Konfiguration folgt auf die andere. Der resultiende Zustand der letzten Konfiguration tauch als neuer aktueller Zustand in der Folge-Konfiguration auf. Die **Konfigurations-Folge K** eines Automaten ist praktisch ein Protokoll über der Arbeits-Verlauf des Automaten.
In der Tabelle ist der wesentliche Teil dafür bläulich gekennzeichnet.

Konfigurationen des Automaten

Arbeits-Schritt	aktueller Zustand	nächstes Eingabe-Symbol	Folge-Zustand	Rest-Wort
0	q_0	**a**	q_1	**bbbc**
1	q_1	**b**	q_2	**bbc**
2	q_2	**b**	q_2	**bc**
2	q_2	**b**	q_2	**c**
3	q_2	**c**	q_3	-
4	q_3	-		

Eine andere Darstellung der Automaten-Konfigurations-Folge ist eine textliche Ausgabe in der Form:

K = ($[q_0, a] \vdash [q_1, b] \vdash [q_2, b] \vdash [q_2, b] \vdash [q_2, c] \vdash [q_3, -]$ **)**

wobei das Zeichen " ⊢ " für ***"führt zu"*** bzw. ***"es folgt"*** steht. Bei diese Form wir die Reihenfolge der Konfigurationen (Arbeits-Zustände) deutlicher.
WENN das Eingabe-Wort vollständig abgetastet / eingelesen wurde UND der Automat steht auf einem vorgegebenenm End-Zustand, DANN gilt das Wort als ***"akzeptiert"***; SONST ist es kein gültiges Wort der Sprache, die der Automat bearbeiten soll
im zweiten Fall wird das Eingabe-Wort ***"nicht akzeptiert"***.
Das Akzeptanz-Verhalten des endlichen Automaten beschreibt also die gültigen Ausdrücke (Worte) die zur Sprache des Automaten gezählt werden.

Definition(en): deterministischer endlicher Automat (DEA, FSM, DFA)

Ein determinstischer endlicher Automat wechselt bei allen Zuständen in Abhängigkeit von den Eingaben in (einzelne) eindeutige definierte Folge-Zustände.

Ein deterministischer endlicher Automat ist ein Quin-Tupel (5-Tupel) $M = (Z, \Sigma, f, z_0, Z_E)$ mit:

Z … endliche Menge der Zustände
Σ … Eingabe-Alphabet; $Z \cap \Sigma = \varnothing$
f … Überführungs-Funktion; $Z \times \Sigma \rightarrow Z$; (Funktion muss nicht ***total*** sein)
z_0 … Start-Zustand; $z_0 \in Z$
Z_E … endliche Menge der End-Zustände

Einen etwas anderen Ansatz verfolgt STARKE (1969) mit seiner Definition eines determinstischen abstrakten Automaten. Er hatte wohl mehr die Maschinen (in der heutigen Sicht Automaten mit Ausgabe) bzw. Transduktoren im Sinn.

Definition(en): deterministischer abstrakter Automat

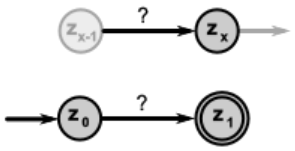
Ein deterministischer abstrakter Automat ist ein Quad-Tupel (4-Tupel) $M = (X, Z, Y, f)$ mit:

X … Eingabe-Alphabet; endliche Menge von Symbolen / Zeichen
Z … endliche Menge der Zustände
Y … Ausgabe-Alphabet; endliche Menge von Symbolen / Zeichen
f … Überführungs-Funktion; $Z \times \Sigma \rightarrow Z$

3.2.7.1. Konstruktion eines DEA

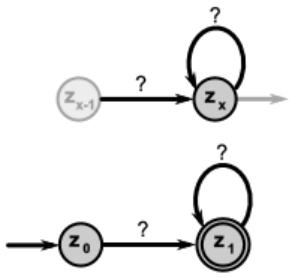
Für viele reguläre Ausdrucke sind sehr einfache Automaten ableitbar. Diese können dann als Bau-Elemente für komplexere Sprachen anwenden.

einzelne Zeichen
das Fragezeichen (?) steht für das entsprechende Zeichen. In unseren Schul-Automaten ist das oft 0,1,a oder b usf.usf.
Die obere Abbildung zeigt ein Element innerhalb eines Automaten, während in der unteren Abbildung ein akzeptierender Automat gezeichnet ist.

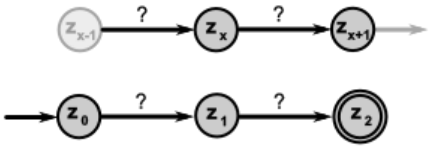


eine Folge von gleichen Zeichen

Stern-Operator über das Zeichen
da bei einem DEA keine ε-Übergänge möglich sind, wird mindestens die Eingabe eines Zeichens erwartet. Weitere Zeichen sind aber möglich.

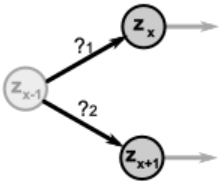


Konkatenation
bestimmte Anzahl gleicher Zeichen und / oder bestimmte Aneinanderreihung von Zeichen

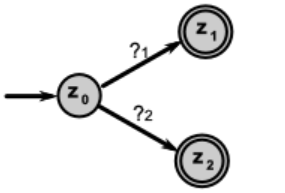


Vereinigung oder Alternative

Plus-Operator
hierbei sind mindestens zwei verschiedene Teilsprachen / Teil-Grammatiken in der definierten Sprache / Grammatik integriert. Der Automat geht entweder zu Zustand zx oder zu zx+1, jenachdem welche der beiden möglichen Eingaben vorliegt.

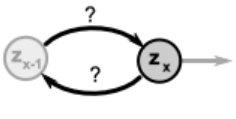


Wenn man ganz exakt konstruieren will, dann entsteht zuerst ein NEA mit ε-Übergängen. Diese können aber später auch wieder eleminiert werden. Für die Konstruktion eines DEA ist es natürlich eine Notwendigkeit.

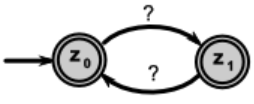


KLEENE-Konstruktion

Stern-Operator
kein-mal, ein-mal oder beliebig oft wiederholend



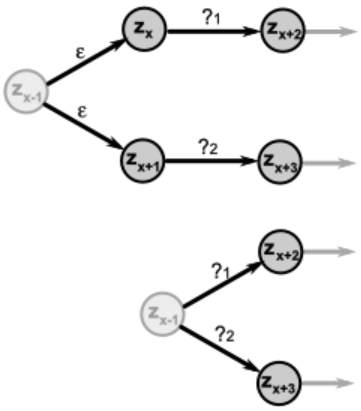
Stern-Operator



ODER

Entfernung von ε-Übergängen

ist für einen DEA notwedig, da dieser eben keine ε-Übergänge enthalten darf



Aufgaben:

1. *Entwickeln Sie einen DEA, der den Turmbau aus Zylindern (a'la Turm von Hanoi) mit den Durchmessern 5, 4, 3, 2 und 1 (cm) prüft! Es kann mit einer beliebigen Größe begonnen werden und immer nur ein kleiner Zylinder auf einen größeren gesetzt werden. Akzeptiert wird der Bau nur, wenn der kleinste Zylinder obenaufliegt. Ein Turm besteht aus mindestens zwei Zylindern.*
2. *Realisieren Sie Ihren Automaten (von Aufgabe 1.) mit JFLAP! Prüfen Sie dann die folgenden "Türme"!*

a)	5 4 3 2 1	b)	4 2 1	c)	4 3
d)	1	e)	3 5 2	f)	4 3 2
g)	5 5 3 3 3 1 1 1 1	h)	5 1	i)	7

3. *In einer erweiterten Version des Turmbaus sind auch mehrere Zylinder eines Durchmessers erlaubt, die bei gleicher Größe ebenfalls übereinander gelegt werden dürfen. Prüfen Sie die folgenden Eingaben!*

a)	5 3 3 3 2 2 1	b)	3 1 1 1 1	c)	5 2 3 3 3 1
d)	1 2 3 4 5	e)	5 5 5 5 3 3 3	f)	4 1
g)	5 5 5 5 5 5 5 5 5 1	h)	1 1	i)	4 3 2

4. *Entwickeln Sie einen DEA, der einen unendlich hohen Turm aus LEGO-Steinen prüft. Zur Verfügung stehen Steine der Breite 1, 2 und 4. Die Tiefe der Steine ist immer auf 2 (übliche Stein-Tiefe) festgelegt und kann vernachlässigt werden. Die Steine dürfen so kombiniert werden, dass ein unendlich hoher Turm der Breite 4 (und der Tiefe 2) entsteht! Immer wenn eine Breite 4 erreicht wurde, dann wird auf der nächsten Ebene / Schicht weitergebaut! Ist die Schicht breiter als 4, dann ist der Turmbau ab dort ungültig!*
5. *Realisieren Sie Ihren Automaten (von Aufgabe 4.) mit JFLAP! Prüfen Sie dann die folgenden "Türme"!*

a)	4 4 2 2 2 1 1	b)	1 1 1 2 4 2 2 4	c)	1 2 1 4 4 2 2 2 1 1
d)	1 2	e)	2 2 2 2 2 2 2 2 2 2	f)	2 2 2 2 2 2 2 2 2 2 2
g)	4 4 4 4 4 4 2 2 2 1 4	h)	2 4 1 1 4	i)	2 1 4

für die gehobene Anspruchsebene:

6. *Entwickeln Sie den "Zylinder-Turmbau-Akzeptierer so weiter, dass beliebige "kleinste" Zylinder oben auf liegen dürfen!*
7. *Der LEGO-Turm soll mit einer Breite von 6 aus Steinen mit zwei unterschiedlichen Farben gebaut werden! Innerhalb einer Schicht müssen sich die Farben immer abwechseln. Die nächste Schicht kann mit der gleichen Farbe beginnen.*

3.2.7.2. Reduktion eines DEA

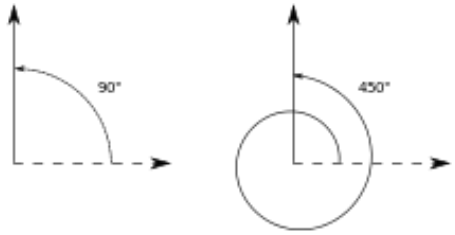
auch Minimierung / Vereinfachung genannt

Ziel sind kleine – genauer gesagt kleinste – Automaten, die immer noch die gleiche Leistung bringen
Vorteile: höhere Übersichtlichkeit, besseres Verständnis; schneller; billiger; kompakter
Bei vielen Algorithmen / Verfahren wird die Eleminierung der nicht-erreichbaren Zustände als erster Schritt gemacht. Wir schieben diesen Schritt hier vor die eigentlichen Reduktionen, um so die eigentlichen Schritte besser sichtbar zu machen.
Alle nicht-ereichbaren Zustände sind dadurch zu erkennen, dass zu ihnen keine Pfeile hinführen. Sind solche Zustände erst einmal rausgestrichen, dann kann es auch sein, dass auf einem Mal Automaten-Teile isoliert sind. Sie sind ebenfalls nicht erreichbar und können als Gruppe ebenfalls entfernt werden.

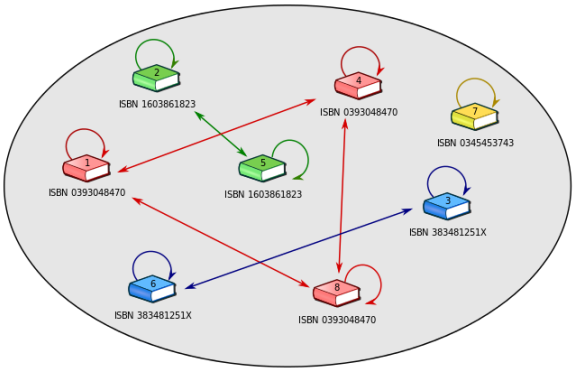
Die Verfahren reichen dabei von recht einfachen bis zu sehr komplexen Algorithmen. Welche Verfahren jeweils genutzt werden können, hängt auch von der Komplexität des Automaten ab. Die komplexeren Algorithmen funktionieren allgemein und terminieren auch.
Die einfachen Verfahren sind meist nur bei einigen Automaten mit übersichtlichen Überführungs-Funktionen möglich. Dabei kann auch nicht sichergestellt werden, dass der resultierende Automat wirklich minimal ist.
Als letztes Verfahren stelle ich hier ein verständliches, graphisch-orientiertes Vorgehen vor, dass ich im Internet fand.Für den schulischen gebrauch halte ich es für super optimal. Mir ist völlig unklar, warum es sich dort nicht schon lange etabliert hat?

Definition(en): Äquivalenz-Klasse

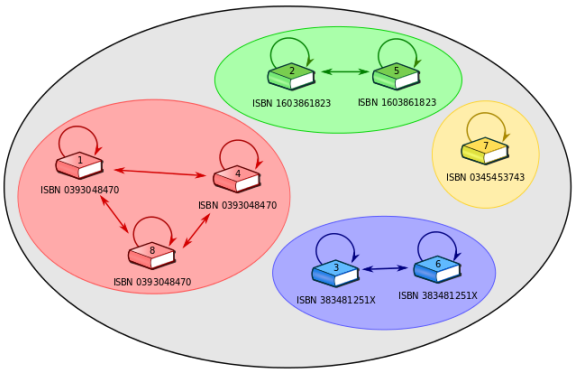
Eine Äquivalenz-Klasse ist die Teil-Menge an Elementen, deren Zugehörigkeit zu dieser durch eine ausgewählte Funktionalität bestimmt wird.



äquivalente Winkel
Q: de.wikibooks.org [Mathe für Nicht-Freaks] (Stephan Kulla)



äquivalente Bücher (gleiche ISBN) einer Bibliothek
Q: de.wikibooks.org [Mathe für Nicht-Freaks] (Stephan Kulla)



Äquivalenz-Klassen in der Bibliothek
Q: de.wikibooks.org [Mathe für Nicht-Freaks] (Stephan Kulla)

informatisch erinnert die Äquivalenz-Klassen-Bildung an die Modulo-Operation
mit ihr werden die Zahlen ja auch in Klassen zerlegt z.B. bei "mod 3" erhalten wir 3 Klassen

Zahl n	n mod 3		n	n mod 3		n	n mod 3
1	**1**		2	**2**		3	**0**
4	**1**		5	**2**		6	**0**
7	**1**		8	**2**		9	**0**
10	**1**		11	**2**		12	**0**
13	**1**		14	**2**		15	**0**
3 * x + 1	**1**		3 * x + 2	**2**		3 * x	**0**
Klasse1 (Rest 1)			**Klasse2 (Rest 2)**			**Klasse3 (Rest 0)**	

für x = 0,1,2, …

weitere Beispiele für Äquivalenz-Klassen

(alle) Menschen lassen sich (vereinfacht) in die Äquivalenz-Klassen Männer und Frauen zerlegen

die Schüler einer Schule lassen sich in Schulklassen einordnen

Im Falle der Verfahren um die Reduzierung von Automaten geht es um die Funktionalität den / einen End-Zustand zu erreichen.

Aufgaben:

1. *Geben Sie drei weitere mögliche Äquivalenz-Klassen-Bildungen hinsichtlich der Zahlen von 1 bis 20 an!*
2.
3.

Reduktion mittels Zustands-Tabelle

Entfernen aller Zustände, die nicht erreicht werden können (also kein fremder Übergang vorhanden ist)

ev. wiederholen, bis nur noch erreichbare Zustände vorhanden sind

Reduktions-Algorithmus

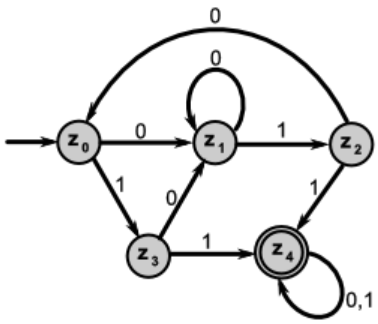
1. **Sortieren der Zustände nach Folge-Zuständen**
2. **Gruppieren der Zustände nach der Lage der Folge-Zustände** (innerhalb gleicher Klassen)
3. **Zusammenfassen und Umbenennen von äquivalenten Zuständen**
4. **Ändern der alten Folge-Zustände bei den anderen Zuständen in den neuen zusammengefassten Zustand**
5. **Wiederholen ab 2. bis alle Klassen ein-elementig sind**

Beispiel:

Der nebenstehende Automat A – ein DEA – kann binäre Zahlen, die mindestens einmal zwei aufeinander folgende Einsen enthalten, erkennen.

L = {0 | 1} 11 {0 | 1}

Es ist der gleiche, den wir für das andere Verfahren benutzt habe.



A = ({z0,z1,z2,z3,z4}, {0,1}, **f**, z0, {z4}), wobei **f** z.B. aus dem Graphen übernommen werden kann.

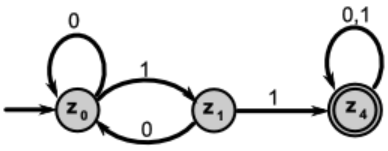
Nachdem die Überführungs-Funktion in Form einer Tabelle (Automaten-Tafel) vorliegt, werden die Zustände nach den Folge-Zuständen sortiert und Zustände mit gleichen Folge-Zuständen markiert.

DEA f	Eingabe	
Zustände	**0**	**1**
z_0	z_1	z_3
z_1	z_1	z_2
z_2	z_0	z_4
z_3	z_1	z_4
$^*z_4$	z_4	z_4

DEA	Eingabe	
Zustände	**0**	**1**
z_2	z_0	z_4
z_1	z_1	z_2
z_0	z_1	z_3
z_3	z_1	z_4
$^*z_4$	z_4	z_4

Dieses Verfahren führt nicht zwangsläufig zu einem minimalistischen – äquivalenten – Automaten. Es ist aber realtiv einfach und bringt schon akzeptable Ergebnisse.

Offensichtlich gibt es aber auch noch einen kleineren Automaten (DEA), der ebenfalls zwei Einsen in Folge akzeptiert.

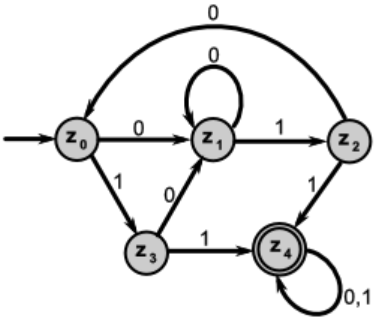


Reduktion mittels Äquivalenz-Tabelle

Reduktions-Algorithmus

0. **Entfernen nicht-erreichbarer Zustände** (gemeint sind Zustände oder Gruppen davon, die vom Start-Zustand mit keiner Eingabe-Folge erreicht werden können)
1. **Erstellen der Äquivalenz-Tabelle** (für jeden Zustand mit jedem anderen; unterer Teil der Tabelle reicht)
2. **Markieren aller Zustands-Paare, bei denen mind. ein Zustand ein End-Zustand ist**
3. **Markieren aller Zustands-Paare …** (Bedingung???)
4. **Testen der anderen Zustands-Paare, bis keine Veränderung mehr eintritt**
5. **Verschmelzen der nicht-markierten Zustands-Paare**

Beispiel:



Der nebenstehende Automat A – ein DEA – kann binäre Zahlen, die mindestens einmal zwei aufeinander folgende Einsen enthalten, erkennen.

L = {0 | 1} 11 {0 | 1}

Es ist der gleiche, den wir für das andere Verfahren benutzt habe.

A = ({z0,z1,z2,z3,z4}, {0,1}, **f**, z0, {z4}), wobei **f** z.B. aus dem Graphen übernommen werden kann.

Nachdem die Äquivalenz-Tabelle erstellt ist, werden alle Zustand-Kombinationen markiert, bei denen (genau) ein End-Zustand enthalten ist. In unserem Beispiel wären das die Kombinationen mit z4.

Wenn zwei End-Zustände auftauchen, dann könnten sie äquivalent sein.

	z_0	z_1	z_2	z_3	$*z_4$
z_0					X
z_1					X
z_2					X
z_3					X
$*z_4$	X	X	X	X	
	z_0	z_1	z_2	z_3	$*z_4$

Sinn der Tabelle ist es, die Zustände zu finden, die gleich sind – also das Gleiche leisten.

Die Übergänge sind – wie üblich von waagerecht nach senkrecht zu verstehen.

Das X bedeutet, dass die Zustände nicht miteinander äquivalent sind.

Man sieht schon, dass die obere Hälfte – also der rechte, obere Teil der Tabelle über der Diagonale – der unteren, linken Hälfte entspricht. Um nicht alles doppelt zu notieren, beschränkt man sich auf die eine Hälfte. Wir verzichten deshalb auch auf die obere Zustands-Liste und nutzen nur die untere.

Jetzt wird jede Zustands-Kombination geprüft, ob sie das Gleiche leistet – also äquivalent ist.

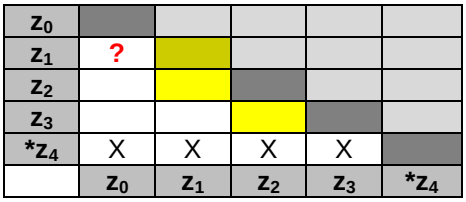
Für die Kombination z0 und z1 (grüner Hintergrund) ergeben sich folgende Übergänge:

z0: "0" → z1
z1: "0" → z1 somit die Kombination {z1,z1}.

z0: "1" → z3
z1: "1" → z2 hier somit die Kombination {z2,z3}.

z_0					
z_1					
z_2					
z_3					
$*z_4$	X	X	X	X	
	z_0	z_1	z_2	z_3	$*z_4$

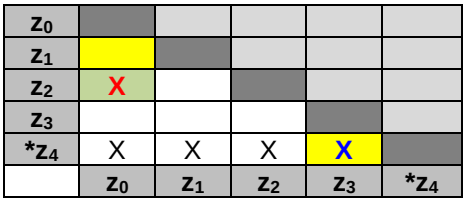
Nun wird für den gerade untersuchten Übergang (**rotes Fragezeichen**) geprüft, ob für die Kombinationen ({z1} und {z2,z3}) schon Markierungen (auf Nicht-Äquivalenz; gelber Hintergrund) in der Tabellen vorhanden sind. Dieses ist bisher nicht erfolgt, was bedeutet, es gibt noch keine Aussage über eine mögliche Äquivalenz.



Also muss die Kombination z0,z1 Übergang (**?**) erst einmal offen bleiben.
Wir prüfen nun die nächste Kombination, also z0,z2:
Für die Kombination z0 und z2 ergeben sich folgende Übergänge:

z0: "0" → z1
z2: "0" → z0 somit die Kombination {z0,z1}.

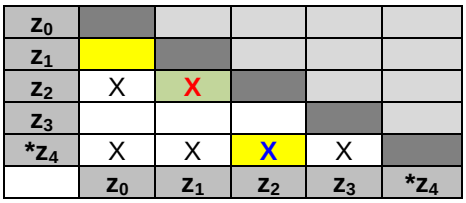
z0: "1" → z3
z2: "1" → z4 hier also {z3,z4}.



Jetzt finden wir in der Tabelle für die Kombination z3,z4 (**blaues Kreuz**) schon einen Eintrag für eine Nicht-Äquivalenz.
Damit kann die gerade bearbeitete Kombination auch rausgestrichen werden. Es reicht, wenn für eine der ermittelten Kombinationen die Nicht-Äquivalenz nachweisen können. Das kann man sich in den weiteren Schritten immer zunutze machen.
Die nächste zu prüfende Äquivalenz ist die für die Zustände z1 und z2:

z1: "0" → z1
z2: "0" → z0 somit die Kombination {z0,z1}.

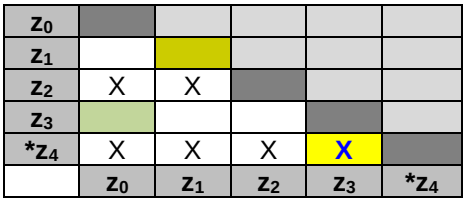
z1: "1" → z2
z2: "1" → z4 hier also {z2,z4}.



Wir erinnern uns noch, dass die z4-enthaltenen Kombinationen schon raussortiert wurden.
Also kann wieder die gerade bearbeitete Kombination markiert werden. Damit sind alle Übergänge zu z2 abgearbeitet.
Genau so gehen wir in der Zeile mit den Zustands-Kombinationen mit z3 vor.
Die nächste zu prüfende Äquivalenz ist die für die Zustände z0 und z3:

z0: "0" → z1
z3: "0" → z1 somit die Kombination {z1,z1}.

z0: "1" → z3
z3: "1" → z4 hier also {z3,z4}.

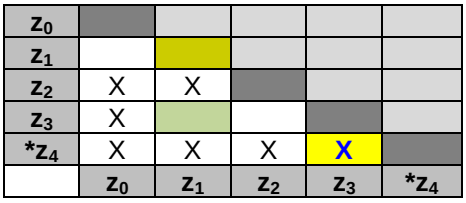


Die z4-enthaltene Kombination kann wieder raussortiert wurden.

Die nächste zu prüfende Äquivalenz ist die für die Zustände z1 und z3:

z1: "0" → z1
z3: "0" → z1 somit die Kombination {z1,z1}.

z1: "1" → z2
z3: "1" → z4 hier also {z3,z4}.



Die z4-enthaltene Kombination kann wieder raussortiert wurden.

Die letzte mit z3 zu prüfende Äquivalenz ist die für die Zustände z2 und z3:

z2: "0" → z0
z3: "0" → z1 somit die Kombination {z0,z1}.

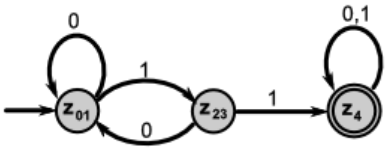
z2: "1" → z4
z3: "1" → z4 hier also {z4,z4}.

z_0					
z_1					
z_2	X	X			
z_3	X	X			
$*z_4$	X	X	X	X	
	z_0	z_1	z_2	z_3	$*z_4$

Beide Kombinationen sind nicht markiert, so dass hier auch ein nicht-äquivalenter Übergang vorliegt.
Übrig bleiben zwei Übergänge (Zustands-Kombinationen; roter Hintergrund), die jetzt als neue Zustände (z01 und z23) verschmolzen werden und neben z4 in den reduzierten Automaten eingehen.
Im folgenden werden dann die einzelnen Übergänge für die verschiedenen möglichen Eingaben konstruiert.

z_0					
z_1					
z_2	X	X			
z_3	X	X			
$*z_4$	X	X	X	X	
	z_0	z_1	z_2	z_3	$*z_4$

Es ergibt sich der nebenstehende reduzierte Automat mit den neuen Zuständen z01 und z23.
Einige praktische Umsetzungen dieses Vorgehens tragen in die Tabelle die Nummer der Durcharbeitungs-Runde ein. Da diese aber später keine weitere Verwendung findet, macht eine solche Differenzierung nur Sinn, wenn eventuelle Fehler gefunden werden sollen.



Wieder andere Umsetzungen tragen die Eingabe-Zeichen ein, für die eine definierte Aussage gemacht werden kann. Das macht deutlich mehr Arbeit und die Tabelle auch unübersichtlicher. Für unsere Zwecke bleiben wir bei der einfachen Verfahrensweise.

Aufgaben:

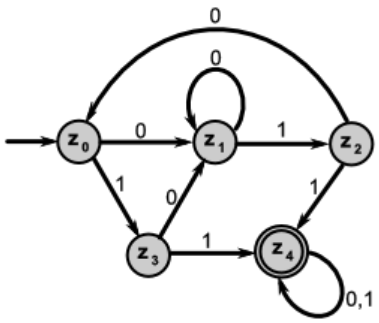
1. ***Prüfen Sie ob der nebenstehende Automat reduzierbar ist! Wenn JA, dann geben Sie den reduzierten Automaten an!***

2.
3.

Reduktion über Äquivalenz-Klassen (I)

Bei diesem Verfahren verwendet man Eingabe-Wörter mit steigender Länge, um grundsätzlich unterschiedliches Verhalten (oder eben gleichartiges) festzustellen. Dabei geht es im Wesentlichen darum die Nicht-Endzustände und die Endzustände gegenüber zu stellen.

Beispiel:



Der nebenstehende Automat A – ein DEA – kann binäre Zahlen, die mindestens einmal zwei aufeinander folgende Einsen enthalten, erkennen.

L = {0 | 1} 11 {0 | 1}

Es ist der gleiche, den wir für das andere Verfahren benutzt habe.

A = ({z0,z1,z2,z3,z4}, {0,1}, **f**, z0, {z4}), wobei **f** z.B. aus dem Graphen übernommen werden kann.

Starten wir mit der Wort-Länge 0, das entspricht nur dem leeren Wort. Hinsichtlich dieses Wortes können wir die Nicht-Endzustände und die Endzustände (hier nur einer) nicht unterscheiden. Somit erhalten wir zwei Äquivalenz-Klassen:

{z4} {z0,z1,z2,z3}

Für beide Klassen prüfen wir nun, ob sie sich hinsichtlich eines Wortes der Länge 1 unterscheiden lassen. Dafür eignet sich eine kleine Eingabe-Zustands-Tabelle. Diese ist praktisch eine nach links gedrehte Überführungs-Tabelle.
Viele Umsetzungen dieses Verfahrens arbeiten auch vertikal und unterscheiden die Klassen dann für die Zeilen.

Für jeden Zustand und jede Eingabe wird der Folge-Zustand eingetragen.
Für uns ist nun nur interessant, welche Verteilung von Nicht-Endzuständen und Endzuständen es gibt. Die Endzustände kennzeichen wir gelb.

Eingabe	z4	z0	z1	z2	z3
0	z4	z1	z1	z0	z1
1	z4	z3	z2	z4	z4

Aus der Tabelle kann man nun entnehmen, dass sich z4 hinsichtlich der eingegebenen Worte der Länge 1 keine Unterscheidung möglich ist. In der anderen Klasse ({z0,z1,z2,z3}) erhalten wir zwei neue Klassen, die siich hinsichtlich des Kriteriums "Erreichbarkeit eines Endzustandes" unterscheiden. Wir erhalten also die Äquivalenz-Klassen:

{z4} {z0,z1} {z2,z3}

Diese Klassen werden nun mit Worten der Länge 2 getestet:

Offensichtlich hat sich die Gruppierung (Klassen-Zuordnung) nicht geändert. Damit bricht das Verfahren hier ab und wir können aus den Klassen nun neue Zustände machen. Häufig bekommen die neuen "Äquivalenz"-Zustände nun kombinatorische Indizies.

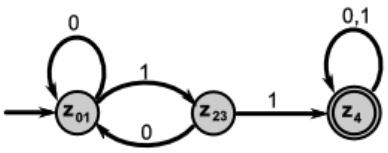
Eingabe	z4	z0	z1	z2	z3
00	z4	z1	z1	z1	z1
01	z4	z2	z2	z3	z2
10	z4	z1	z0	z4	z4
11	z4	z4	z4	z4	z4

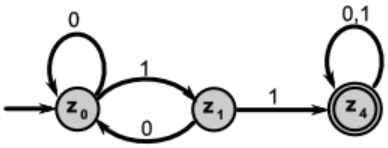
Die erste Klasse enthält nur den Zustand z4 – der bleibt uns erhalten. Die Zustände z0 und z1 werden zu einem neuen Zustand z01 zusammengeführt. Das Gleiche passiert bei z2 und z3. Hier heisst der neue Zustand z23.

Als nächstes wird jetzt die neue Automaten-Tafel aufgestellt.
Bei den kombinierten Zuständen werden beide ursprünglichen Zustands-Überführungen ausprobiert. Wenn wir oben keine Fehler gemacht haben, dann folgt jedesmal der gleiche Folge-Zustand.
Ausgehend von dieser Tabelle können wir am Schluß nun einen minimierten DEA zeichnen.

	Eingabe	
	0	1
z01	z01	z23
z23	z01	z4
*z4	z4	z4

Diese akzeptiert die gleiche Sprache, wie unser Ausgangs-Automat.
Eine weitere Minimierung ist nicht möglich.
Natürlich könnte man jetzt auch noch die (kombinierten) Zustände umbenennen, so dass man zu dem Automaten kommt, den wir vorne schon als sehr klein charakterisiert haben:





Will man aber irgendwelche Vergleiche usw. zwischen dem ursprünglichen und dem reduzierten Automaten anfangen, dann könnten die gleichnamigen Zustände für Verwirrung sorgen. In solchen Fällen bleibt man dann lieber bei den kombinierten Zustands-Namen.

Die einzige "Information", die von unseren Automaten nach draußen dringt ist die "Akzeptanz" bzw. "Nicht-Akzeptanz". Für die Kopplung von Automaten (→ x.y.z. gekoppelte Automaten) ist dies auch ausreichend.
In der Praxis sollen Informationen aber (eventuell) auch (zwischendurch) ausgegeben werden.
Man könnte nun dazu (einige) DEA's auch mit einer Ausgabe versehen. Für solche deterministische endliche Automaten mit Ausgabe (→ x.y.z. Automaten mit Ausgaben - Transduktoren) gibt es zwei grundsätzlich unterschiedliche Varianten – jenachdem an welcher Stelle die Ausgabe erfolgt:

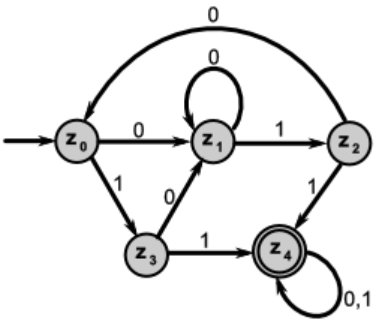
1. MOORE-Automaten (Ausgabe im Zustand) (→ x.y.z.2. MOORE-Automaten)
2. MEALY-Automaten (Ausgabe während des Zustandswechsel) (→ x.y.z.1. MEALY-Automaten)

Beide Automaten – bzw., wie wir sie heute auch betrachten – Maschinen, haben wir schon vorgestellt. Man kann sie quasi als spezielle DEA's betrachten. Es handelt sich i.A. um DEA's mit einer Ausgabe. Diese kann als nebenläufig betrachtet werden.
Wir sehen hier, dass bei den Automaten die Klassifikation recht differenziert ist und sich nicht immer gegenseitig ausschliesst.

Das Konzept der Äquivalenz-Klassen entspricht dem Konzept der verschmelzbaren Zustände.

Die vielleicht anschaulichste Verfahren-Führung für die Äquivalenz-Klassen-Konstruktion fand ich auf einer Webseite von Martin THOMA (https://martin-thoma.com/minimierung-eines-automaten-mittels-aquivalenzklassenkonstruktion/). Wir betrachten zum Vergleich immer noch den gleichen Automaten.

Beispiel:



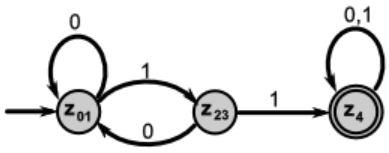
Der nebenstehende Automat A – ein DEA – kann binäre Zahlen, die mindestens einmal zwei aufeinander folgende Einsen enthalten, erkennen.

L = {0 | 1} 11 {0 | 1}

Es ist der gleiche, den wir für das andere Verfahren benutzt habe.

A = ({z0,z1,z2,z3,z4}, {0,1}, **f**, z0, {z4}), wobei **f** z.B. aus dem Graphen übernommen werden kann.

Das hier vorgestellte graphische Verfahren unterscheidet sich praktisch nicht von den anderen Äquivalenz-Algorithmen.
Somit beginnen wir auch hier mit der Notierung der Zustände.
Im 1. Schritt sortieren wir die End-Zustände heraus.
Nun (2. Schritt) prüfen wir wir immer für alle Eingabe-Symbole (hier ja nur 0 und 1), welche zu einem End-Zustand führen. Dazu schreibt man immer kleine Pfeile an die Zustände. Es wird also gekennzeichnet, von welchem Zustand man wohin gelangt. Beim Eingabe-Symbol 0 passiert nicht viel. Alle Zustands-Wechsel bleiben in der eigenen Gruppe (Äquivalenz-Klasse) – also kann hier keine neue Äquivalenz-Klasse erstellt werden.
Anders sieht das bei der 1 als Eingabe-Zeichen aus. Nun gehen zwei Übergänge zu z_4 – einem möglichen End-Zustand. Damit ergibt sich eine neue Äquivalenz-Klasse – nämlich die aus den Zuständen z_2 und z_3. Diese werden abgetrennt.
In der nächsten Runde (über das Eingabe-Alphabet) prüfen wir nun wieder die Übergänge – jetzt aber zur neuen Äquivalenz-Klasse. Bei der 0 ergibt sich wieder keine Klassen-Bildung.
Beim Prüfen der 1 als Eingabe-Symbol finden wir zwei Übergänge zur Äquivalenz-Klasse $\{z_2,z_3\}$ und damit einen bilden z0 und z1 ebenfalls eine Klasse für sich.
Da nun keine weitere Aufteilung mehr möglich ist, können wir hie mit dem Zerlegen enden.
Ansonsten hätten wir noch weiter über die Eingabe-Symbole getestet, bis keine Veränderungen mehr eintreten.
Aus den reduzierten Zuständen (eigentlich noch Äquivalenz-Klassen) erstellen wir nun den reduzierten Automaten.
Schöner und verständlicher geht es meiner Meinung nach garnicht. Da wundert man sich, dass diese Form der Äquivalenzklassen-Konstruktion so wenig publiziert wird.



Exkurs: Äquivalenz-Klassen am Beispiel von HTML-Tag's

Im XML sind die Tag's üblicherweise immer zwei-teilig. Es gibt also einen einleitenden / startenden Tag und einen abschließenden / beendenden Tag.

Einzel-Beispiele:
<html> </html>
<head> `Seitenname` </head>
<body> `Seitentext` </body>

werden mehrere Tag's benötigt / verwendet (z.B. für eine HTML-Seite), dann müssen sie als ganzer Block aneinander gehängt werden:

<head> `Seitenname` </head> <body> `Seitentext` </body>

oder sauber geschachtelt werden:

<html> <head> `Seitenname` </head> <body> `Seitentext` </body> </html>

Beim HTML ist es egal, ob der Gesamttext fortlaufend oder strukturiert geschrieben wird. Somit ist:

```
<html>
  <head>
    Seitenname
  </head>
  <body>
    Seitentext
  </body>
</html>
```

zur obigen Zeile gleichwertig.
Ein Überschneiden der Tag-Blöcke ist nicht zulässig!

Die folgenden HTML-Abschnitte gehören zu einer Äquivalenz-Klasse:

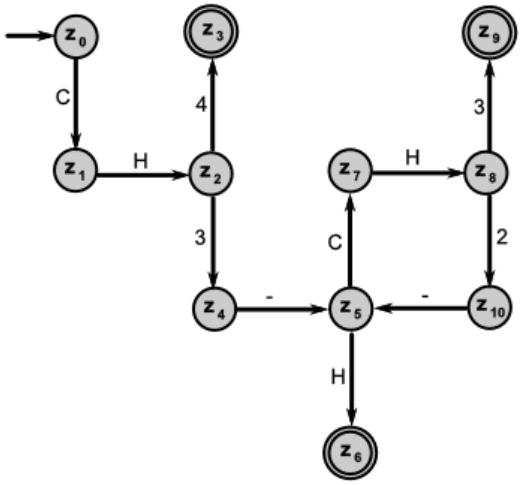
 `Absatz`
  fett
   fett und kursiv
   kursiv und fett
   kursiv und fett   … **fett**

Aufgaben:

1. *Erstellen Sie die Konfigurations-Folge für das Eingabe-Wort "abbbabc" auf dem Eingabe-Band! Wird die Eingabe akzeptiert oder nicht? Begründen Sie Ihre Meinung?*
2. *Prüfen Sie, ob die folgenden Symbol-Folgen auf dem Eingabe-Band zur Akzeptanz oder Ablehnung führen!*

a)	abc	b)	ac	c)	cccbbbaaa
d)	abbbbbbbbbcc	e)	acbbbc	f)	cababac
g)	bbbb	h)	aa	i)	ccccccccc

3. *Von einem Automaten, der die verkürzten Strukturformeln von Alkanen erkennen soll, ist der nebenstehende Entwurf eines DEA vorgeschlagen worden.*



   I) Prüfen Sie, ob es sich um einen DEA handelt! Wenn JA, dann erstellen Sie ihn in einem geeigneten Programm! Wenn NEIN, dann ergänzen Sie fehlende Teile oder korrigieren Sie den Überführungs-Graphen, übertragen Sie den Automaten in ein geeignetes Programm und prüfen Sie dann die folgenden "Wörter"!

akzeptierte "Wörter":

a)	CH3-H	b)	CH3-CH2-CH3	c)	CH4
d)	CH3-CH3	e)	CH3-CH2-H	f)	CH3-CH2-CH2-CH3

nicht akzeptierte "Wörter":

g)	CH2-H2	h)	CH2-CH2	i)	H-CH2-CH2-H
j)	CH3-CH-CH3	k)	CH-	l)	CH3-CH2CH3

   II) Stellen Sie die Tabelle mit den Überführungs-Funktionen auf!

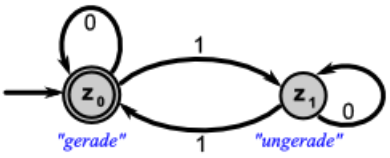
   III) Definieren Sie den Automaten!

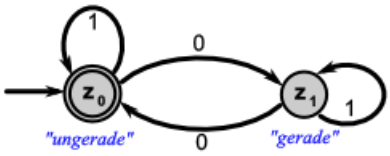
3. *Erstellen Sie einen DEA, der chemische Formeln (verkürzte Struktur-Formeln) für primäre, unverzweigte Alkane in der Form* CH_3-$(CH_2)_n$-OH *akzeptiert! Die Eingabe erfolgt z.B. so:* CH3-CH2-CH2-CH2-OH
4.

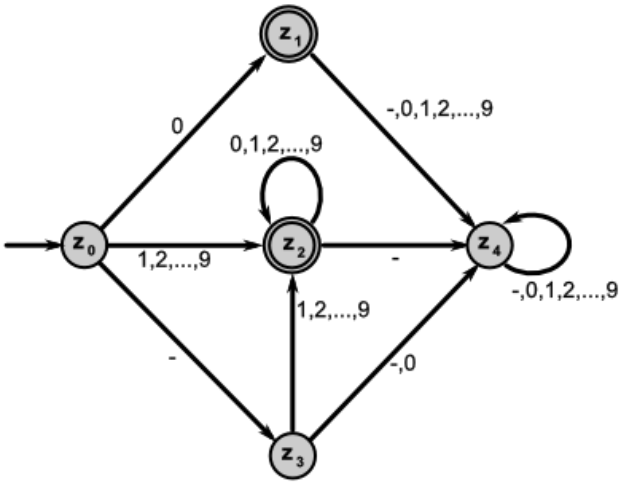
für die gehobene Anspruchsebene:

x. Welche Veränderungen im Zustands-Diagramm, in der Überführungs-Tabelle und im Akzeptanz-Verhalten (Beispiel-haft für die Fälle von Aufgabe 2) ergeben sich, wenn q_4 ein End-Zustand wäre?

x. Erstellen Sie einen DEA, der die verkürtzten Strukturformeln von unverzweigten Alkenen (nach obigen Muster) erkennt!



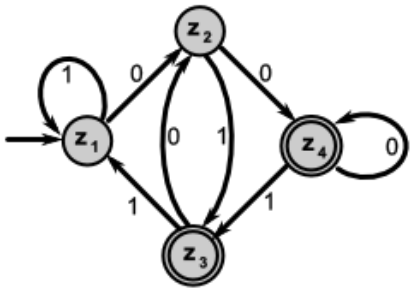


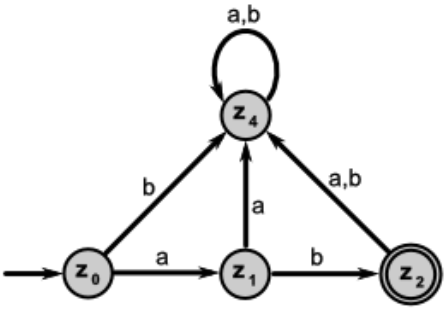


von der Grammatik zum Automaten (DEA)

1. Beschreibung der Sprache mit komplexen regulären Ausdrücken und / oder Grammatiken
2. Umwandlung in einfache reguläre Ausdrücke
3. Umwandlung in e-NEA
4. Umwandlung in e-freien DEA
5. Minimierung des DEA

endlicher Automat für die Sprache L = (0 ∪ 1)*0(0 ∪ 1)
vorletztes Zeichen (Dualzahl, binäres Alphabet) ist eine Null



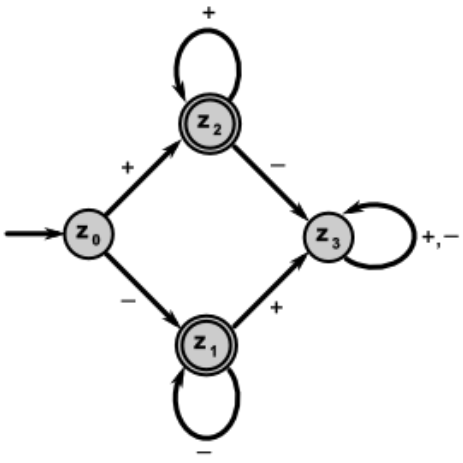


Aufgaben:

1.

2. Gegeben ist nebenstehender Automat:

A ($Z=\{z_0,z_1,z_2,z_3\}$, $X=\{+,-\}$, $Z_S=\{z_0\}$, $Z_E=\{z_1,z_2\}$)



a) Prüfen Sie einige ausgedachte Worte hinsichtlich der Akzeptanz durch den Automaten! Dokumentieren Sie die Worte und Akzeptanz-Ergebnisse (Endzustände) in einer Tabelle!

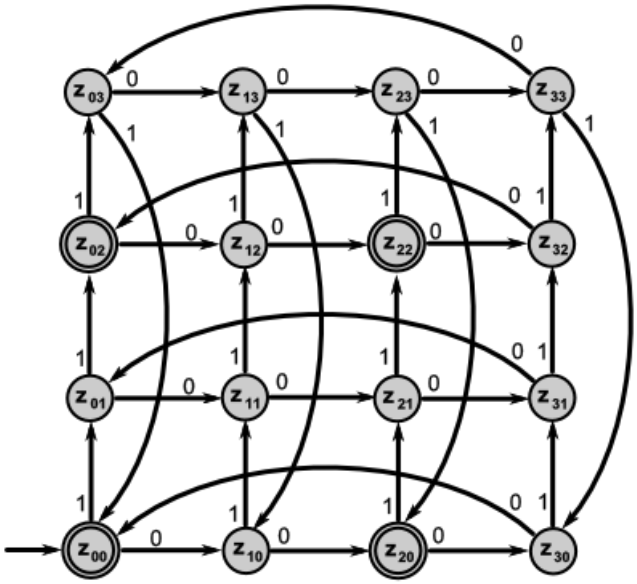
b) Erstellen Sie eine Tabelle mit den Überführungs-Funktionen!

c) Geben Sie die Sprache des Automaten an!

d) Vereinfachen Sie den Automaten, so dass nur noch ein End-Zustand vorkommt!

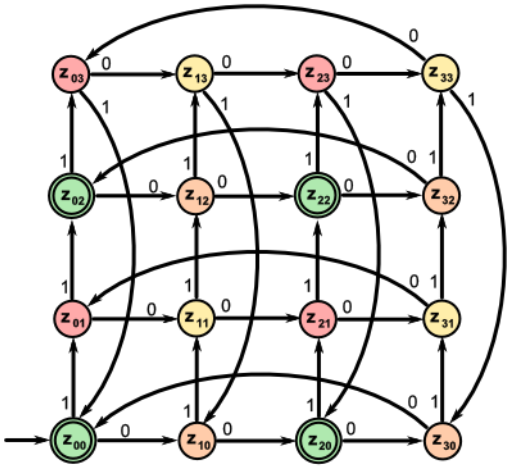
3.

Selbst ein DEA ohne überflüssige Zustände muss noch nicht minimal sein. Nebenstehend abgebildeter Automat akzeptiert alle Dual-Zahlen für die Modulo 10_2 eine Null ergibt.

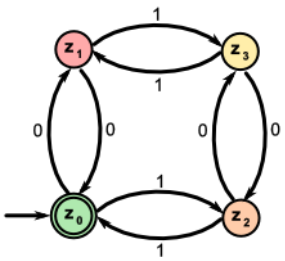


$L = \{ w \ \{0,1\}^* \ ; w \bmod 10_2 = 0\}$

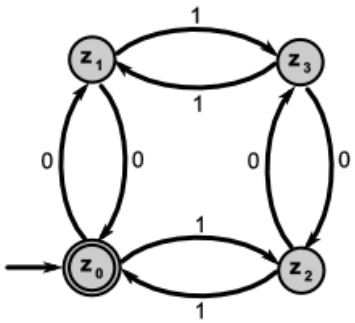
Beim genaueren Betrachten erkennt man, dass es scheinbar Zustände mit gleichem Akzeptanz-verhalte gibt. D.h. praktisch es ist egal von welchem der beiden man startet, man kommt mit gleichen Eingaben immer zu einem Endzustand oder eben nicht.
Färbt man die gleich-reagierenden Zustände gleichartig ein, dann lässt sich dieses Muster für die Vereinfachung nutzen.



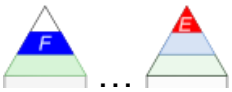
→



Der vereinfachte DEA leistet genauso viel, wie der ursprüngliche.



3.2.7.2. nicht-deterministische endliche Automaten



Abk.: NEA .. nicht-deterministischer Endlicher Automat; NFA .. nondeterministic Finite Automaton

Folge-Zustände eines nicht-deterministischen Systems sind nicht eindeutig
es ergeben sich mehrere Möglichkeiten bzw. Wege
quasi sind die nachfolgenden Zustände auch von noch folgenden Zuständen "abhängig"
ein nicht-deterministischer Automat könnte z.B. auch von Zufälligen Zustands-Änderungen geprägt sein

ein NDA hat einen oder mehrere Start-Zustände und mindestens einen End-Zustand.

nicht-detministische Automaten können sich gleichzeitig in mehreren Zuständen befinden

ausgehend von einem Zustand können trotz einer bestimmten Eingabe unterschiedliche Folge-Zustände oder Ausgaben erfolgen

Zufalls-Generatoren stellen in der Informatik häufig benötigte nicht-deterministische System (Algorithmen) dar
die meisten Systeme sind aber nur quasi- bzw. pseudo-zufällig
bei Kenntnis des Ausgangs-Zustandes und des Algorithmus lassen sich vielfach die ergebnisse deterministisch reproduzieren

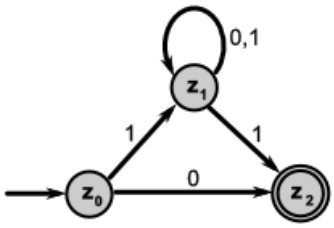
typischerweise sehr einfach
wegen der Nicht-Eindeutigkeit aber nur bedingt nutzbar
Problem lässt sich aber dadurch lösen, dass man einen NEA ind einen DEA umwandelt, was immer geht.

Die akzeptierte Sprache eines NEA ist die Menge der Wörter, die beim Abarbeiten von mindestens einem der Start-Zustände mindestens einen der End-Zustände erreichen.

Definition(en): nicht-deterministischer Automat (NEA, NdEA, NDEA)
Ein nicht-deterministischer Automat kann mindestens einmal von einem Zustand in mindestens zwei unterschiedliche Zustände (Zustands-Folgen) wechseln.
Ein deterministischer endlicher Automat ist ein Quin-Tupel (5-Tupel) $M = (Z, \Sigma, f, Z_0, Z_E)$ mit: Z … endliche Menge der Zustände Σ … Eingabe-Alphabet; $Z \cap \Sigma = \varnothing$ f … Überführungs-Funktion (totale Funktion); $Z \times \Sigma \rightarrow 2^Z$ Z_0 … Menge der Start-Zustände; $Z_0 \subset Z$ Z_E … endliche Menge der End-Zustände; $Z_E \subseteq Z$

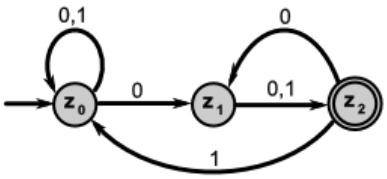
nicht-deterministische Automaten als Rate-Maschinen für den erfolgreichsten Weg
dabei muss der NEA eine bestimmte eingegebene Zeichen-Folge nur (mindestens) einmal akzeptieren

Gibt es keinen einzigen Weg durch den Automaten, dann gilt das Wort als "nicht akzeptiert".
NEA's können mehrere Start-Zustände haben.



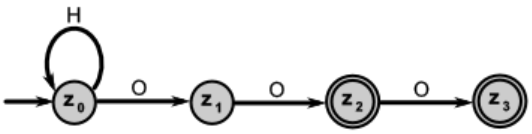
nicht-deterministischer Automat
Q: /2, S.30/

Nicht-deterministischer endlicher Automat (NEA) für die Sprache L = (0 ∪ 1)*0(0 ∪ 1)
vorletztes Zeichen (Dualzahl, binäres Alphabet) ist eine Null



Automat akzeptiert alle Worte aus dem Alphabet {H,O}, bei denen hinter beliebig viel H's zwei oder drei O's folgen.

w = {H} OO[O]



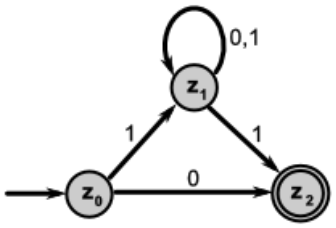
3.2.7.1. Arbeit eines Nicht-deterministischen endlichen Automaten

Die Unbestimmtheit des einzuschlagenen Weges durch den Zustands-Dschungel eines NEA macht die Erläuterung der Arbeit schon recht aufwändig. Prinzipiell arbeitet der NEA wie ein DEA. Die Wechsel der Zustände folgt dem gleichen Abläufen, wie wir es bei den DEA's besprochen haben.

Beim Betrachten können aber einige Sonderfälle auftreten, die bei der Arbeits-Besprechung beachtet werden müssen.
So kann es – ausgehend von einem Zustand – mehrere Folge-Zustände geben. D.h. es gibt für einen Zustand mehrer Überführungs-Funktionen zur gleichen Eingabe. Welcher der richtige Weg ist, kann sich u.U. erst mit dem letzten Eingabe-Zeichen ergeben.

Weiterhin kann es passieren, dass bestimmte Überführungen gar nicht definiert sind. Das bedeutet dann, dass – wenn es auch keinen anderen Weg mehr gibt – der Automat hier stehen bleibt und die Eingabe "nicht akzeptiert".

3.2.7.2. Simulation eines Nicht-deterministischen endlichen Automaten



nicht-deterministischer Automat
Q: /2, S.30/

Umsetzung des obigen NEA in ein PROLOG-Programm.

```
% Defintion StartZustand
startzustand(z0).
% Definition EndZustände
endzustand(z2).

% Definition ÜberführungsFunktionen
ueberfuehrung(z0,0,z2).
ueberfuehrung(z0,1,z1).
ueberfuehrung(z1,0,z1).
ueberfuehrung(z1,1,z1).
ueberfuehrung(z1,1,z2).

% Anwendungen für Automaten
% Akzeptor für Wort W
akzeptiert(W):-startzustand(z0),
               (z0:W) ==>* (Z:[]),
               endzustand(Z).

% AbleitungsFunktion
(Z:[A|W]) ==> (Z1:W):- ueberfuehrung(Z,A,Z1)
```

Q: /2, S.32/, leicht geänd.: dre

3.2.7.3. NEA oder DEA – Wer kann mehr? Welche Klasse ist mächtiger?

Satz:
Für jeden nicht-deterministischen endlichen Automaten (NEA) gibt es einen äquivalenten deterministischen endlichen Automaten (DEA).
NEA $\Rightarrow$ DEA

Ein nicht-deterministischer endlicher Automat (NEA) lässt sich in einen deterministischen (DEA) überführen.

Algorithmus zur Transformation von NEA ind DEA

1. ist der Übergang von einem Zustand zu einem anderen Zustand eindeutig, dann bleiben Folge-Zustand und Übergangs-Funktion erhalten
2. Existieren für einen Zustand Z_x und eine bestimmte Eingabe zwei Übergangs-Funktionen zu zwei Folge-Zuständen Z_y und Z_z, dann wird ein neuer Folge-Zustand Z_{yz} erstellt
3. Untersuchung der bisherigen Zustände: Entblättern des Automaten (es können Zustände entfallen, aber auch hinzukommen)
4. Wiederholung solange, bis keine Veränderungen mehr eintreten

Dabei ist allgemein eine Tendenz zu einer höheren Komplexität zu beobachten.

Satz:
Für jeden deterministischen endlichen Automaten (DEA) gibt es einen äquivalenten nicht-deterministischen endlichen Automaten (NEA).
DEA $\Rightarrow$ NEA

Beispiel für Umwandlung eines NEA in einen DEA:

Satz:
DEA $\Rightarrow$ NEA $\cup$ NEA $\Rightarrow$ DEA $\Rightarrow$ NEA $\equiv$ DEA

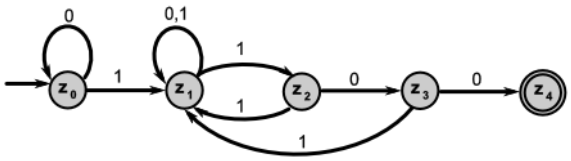
Verfahren zur Umwandlung eine NEA in einen DEA (alternative Beschreibung):

1. die Start-Zustände des NEA werden zu dem (einen) Start-Zustand des DEA zusammengefasst
2. zusammenfassen von Zuständen des NEA zu einem neuen Zustand im DEA
3. für jeden neuen Zustand im DEA und für jedes Zeichen werden die erreichbaren Zustände zu einem neuen Zustand im DEA zusammengefasst
4. ist ein Zustand des NEA ein End-Zustand, dann ist er auch End-Zustand des DEA
5. fehlende Übergänge führen zu einem Fehler-Zustand

Beispiel für Umwandlung eines NEA in einen DEA:

Lösung über die Zustands-Tabelle:

NEA und resultierender DEA als Graphen:
NEA akzeptiert alle Wörter aus "0" und "1", die mit einer "1" beginnen und auf "100" enden
L = 1 {0|1}*100



Umwandlung über die Überführungs-Funktion (Tabelle):

der NEA als Überführungs-Funktion als Zustands-Tabelle

NEA	Eingabe	
Zustand	0	1
z_0	z_0	z_1
z_1	z_1	$\{z_1,z_2\}$
z_2	z_3	z_1
z_3	z_4	z_1
$*z_4$	Ø	z_5

nach dem Überführungs-Algorithmus werden die nicht-eindeutigen Übergänge auf einen neuen Zustand gelegt
vom neuen Zustand werden Überführungen zu ??? konstruiert

der nicht definierte Übergang bei der Eingabe einer "1" im Endzustand wird auf einen Fehler-Zustand z_5 gelegt

	Eingabe	
Zustand	0	1
z_0	z_0	z_1
z_1	z_1	z_{12}
z_{12}	z_3	z_1
z_2	z_3	z_1
z_3	z_4	z_1
$*z_4$	Ø	**z_5**
z_5	**Ø**	**Ø**

!!! noch nicht fertig

	Eingabe	
Zustand	0	1
z_0	z_0	z_1
z_1	z_1	$\{z_1,z_2\}$
z_2	z_3	z_1
z_3	z_4	z_1
$*z_4$	Ø	z_5

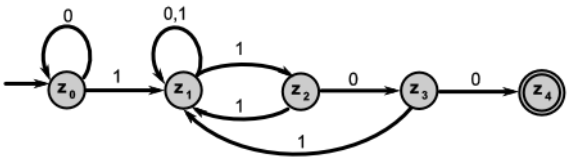
der Fehlerzustand führt immer zu sich selbst zurück – als hier auf z5

Entfernen der nicht-erreichbaren Zustände

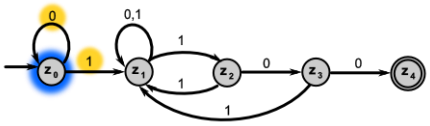
!!! noch kein DEA!

Lösung am Graphen:

NEA und resultierender DEA als Graphen:
NEA akzeptiert alle Wörter aus "0" und "1", die mit einer "1" beginnen und auf "100" enden
L = 1 {0|1}*100



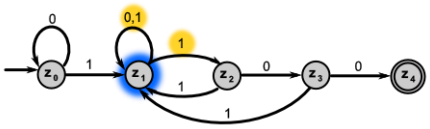
es wird bei z0 begonnen hier ist der Automat schon deterministisch



	Eingabe	
Zustand	0	1
z_0	z_0	z_1
z_1	z_1	$\{z_1,z_2\}$

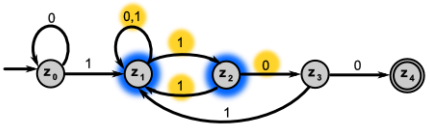
für z1 ist der Übergang bei 0 klar, für eine 1 gibt es zwei Möglichkeiten
diese notieren wir in einer Zustandsmenge {z1,z2}
die gefundene Zustandsmenge wird als neuer Zustand eingeführt
für diese betrachten wir alle möglichen Folgezustände
wir müssen also testen, wo man für 0 von z1 (→ z1) und wo von z2 (→ z3) hinkommt; damit zusammen also {z1,z3} erreichbar; das wird notiert
genauso verfahren wir für die 1:
z1 → z1,z2
z2 → z1
zusammen also {z1,z2}

	Eingabe	
Zustand	0	1
z_0	z_0	z_1
z_1	z_1	$\{z_1,z_2\}$
$\{z_1,z_2\}$	$\{z_1,z_3\}$	$\{z_1,z_2\}$

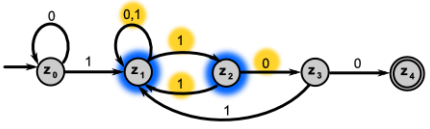


für die neue Zustands-Menge {z1,z3} verfahren wir wie oben weiter
0: z1 → z1; z3 → z4 ➔ {z1,z4}
1: z1 → z1,z2; z3 → z1 ➔ {z1,z2}

	Eingabe	
Zustand	0	1
z_0	z_0	z_1
z_1	z_1	$\{z_1,z_2\}$
$\{z_1,z_2\}$	$\{z_1,z_3\}$	$\{z_1,z_2\}$
$\{z_1,z_3\}$	$\{z_1,z_4\}$	$\{z_1,z_2\}$



es taucht wieder eine neue Zustands-Menge auf
für die neue Zustands-Menge {z1,z4} verfahren wir wie oben weiter



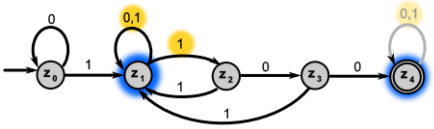
0: z1 → z1; z4 → - ➔ {z1,-}
1: z1 → z1,z2; z4 → - ➔ {z1,-}

	Eingabe	
Zustand	0	1
z_0	z_0	z_1
z_1	z_1	$\{z_1,z_2\}$
$\{z_1,z_2\}$	$\{z_1,z_3\}$	$\{z_1,z_2\}$
$\{z_1,z_3\}$	$\{z_1,z_4\}$	$\{z_1,z_2\}$
$\{z_1,z_4\}$	$\{z_1,-\}$	$\{z_1,z_2,-\}$

wenn wir in z4 sein würden, dann wäre jede Eingabe ein Fehler, das repräsentieren wir über einen neuen Fehler-Zustand z5 (in manchen Verfahren wird auch gerne zF benutzt, um den Zweck klar zu machen)
in deterministischen Automaten sind ja keine offenen Zustands-Übergänge möglich
es tauchen dadurch nochmals zwei neue Zustands-Mengen – mit einem Fehler-Zustand) – auf

die unklaren "Zustände" (-) werden also durch z5 ersetzt und die resultierenden Zustands-Mengen – hier jetzt zwei – wieder in die Tabelle eingetragen

auch für diese umbenannte Zustands-Mengen verfahren wir wie oben



	Eingabe	
Zustand	0	1
z_0	z_0	z_1
z_1	z_1	$\{z_1,z_2\}$
$\{z_1,z_2\}$	$\{z_1,z_3\}$	$\{z_1,z_2\}$
$\{z_1,z_3\}$	$\{z_1,z_4\}$	$\{z_1,z_2\}$
$\{z_1,z_4\}$	$\{z_1,z_5\}$	$\{z_1,z_2,z_5\}$
$\{z_1,z_5\}$	**$\{z_1,z_5\}$**	**$\{z_1,z_2,z_5\}$**
$\{z_1,z_2,z_5\}$	**$\{z_1,z_5\}$**	**$\{z_1,z_2,z_5\}$**

für {z1,z5} gilt:

0: z1 → z1; z5 → z5 ➔ {z1,z5}
1: z1 → z1,z2; z2 → z1; z5 → z5 ➔ {z1,z2,z5}

und für {z1,z2,z5} gilt ebenfalls:

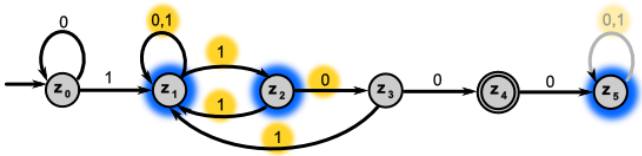
0: z1 → z1; z5 → z5 ➔ {z1,z5}
1: z1 → z1,z2; z2 → z1; z5 → z5 ➔ {z1,z2,z5}

da keine neuen Zustände oder Zustands-Mengen auftauchen sind wir mit diesem Teil des Umwandeln fertig

aus der nebenstehenden Tabelle mit Zustands-Mengen werden nun – durch Umbenennen – neue Zustände gemacht

man kann neu durchnummerieren oder vergibt Zustands-Namen, die an die ursprüngliche Zustands-Menge erinnern

	Eingabe	
Zustand	0	1
z_0	z_0	z_1
z_1	z_1	$\{z_1,z_2\}$
$\{z_1,z_2\}$	$\{z_1,z_3\}$	$\{z_1,z_2\}$
$\{z_1,z_3\}$	$\{z_1,z_4\}$	$\{z_1,z_2\}$
$\{z_1,z_4\}$	$\{z_1,z_5\}$	$\{z_1,z_5\}$
$\{z_1,z_5\}$	$\{z_1,z_5\}$	$\{z_1,z_2,z_5\}$
$\{z_1,z_2,z_5\}$	$\{z_1,z_5\}$	$\{z_1,z_2,z_5\}$

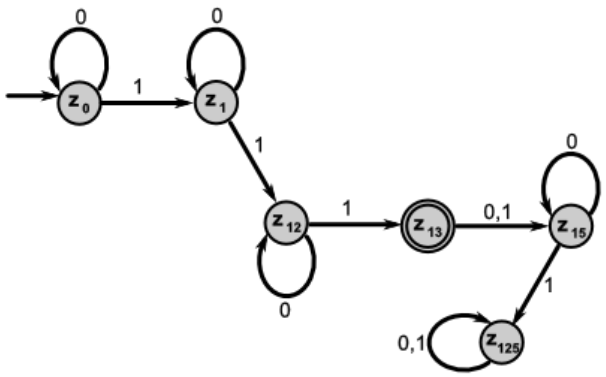


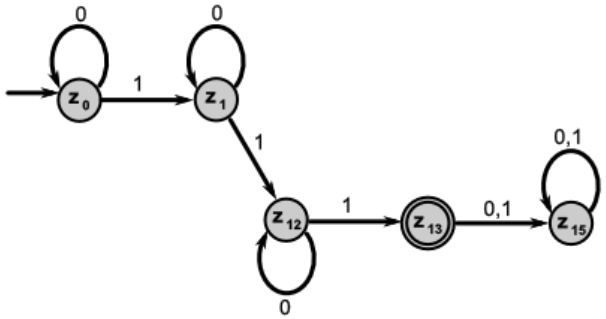
DEA	Eingabe	
Zustand	0	1
z_0	z_0	z_1
z_1	z_1	z_{12}
z_{12}	z_{13}	z_{12}
z_{13}	z_{14}	z_{12}
$*z_{14}$	z_{15}	z_{15}
z_{15}	z_{15}	z_{125}
z_{125}	z_{15}	z_{125}

in Graphen umgesetzter DEA

Men erkenn schnell, dass der Zustand

z125 nicht wirklich gebraucht wird. Es reicht die Schleife bei z15 für beide Eingaben zu nutzen, dann kann z125 einfach entfernt werden





genau ginge

erwartungsgemäß ist der DEA größer als der NEA

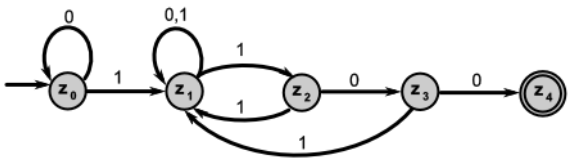
trotzdem ist eine Überprüfung auf eine Vereinfachung immer angebracht

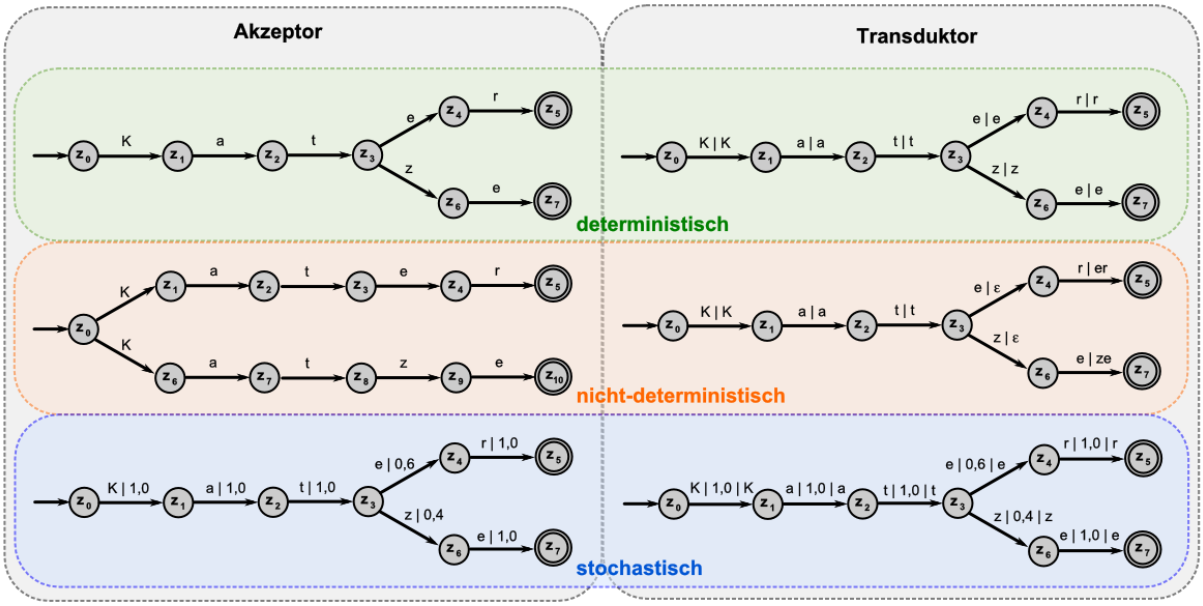
DEA	Eingabe	
Zustand	0	1
z_0	z_0	z_1
z_1	z_1	z_3
z_3	z_4	z_3
z_4	z_5	z_3
z_5	z_6	z_6
z_6	z_6	z_7
z_7	z_6	z_7

andere(s) Beispiel(e):
https://www.youtube.com/watch?v=cMOkwk45m7Y

Lösung über die Potenz-Mengen-Konstruktion:

NEA und resultierender DEA als Graphen:
NEA akzeptiert alle Wörter aus "0" und "1", die mit einer "1" beginnen und auf "100" enden
L = 1 {0|1}*100

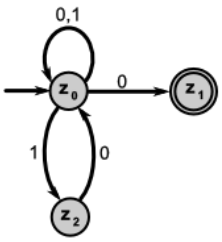




3.2.7.4. Reduktion eines NEA

nicht so klar, wie bei den DEA's

gut sichtbar an Beispiel rechts



oben: Beispiel-NEA
unten: reduzierter äquivalenter NEA

komplexe / Übungs-Aufgaben zu Automaten (ohne Speicher)

1.

2.

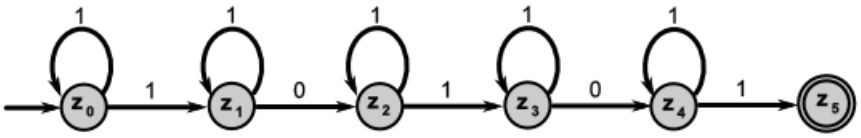
3. Erstellen Sie einen Automaten, der im Funkverkehr die Silben-Folgen von "Mayday! Mayday!" erkennt und diese akzeptiert (Alarm auslöst)!

4.

5.

6. Ein deterministischer Automat soll den internationalen MORSE-Funkkanal für Notrufe abhören und bei zweimal SOS hintereinander Alarm auslösen (akzeptiert)!

7. Der folgende Automat soll eine Sprache erkennen.

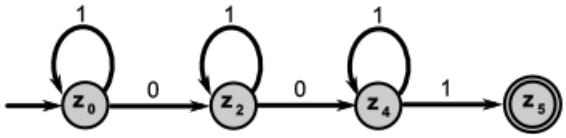


a) Um welche Art von Automat handelt es sich? Begründen Sie kurz!

b) Welche Sprache erkennt der Automat?

c) Definieren Sie den Automaten!

d) Als Vereinfachung wird der folgende Automat vorgeschlagen. Setzen Sie sich mit dem Vorschlag auseinander!



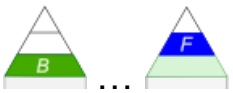
8. Geben Sie einen DEA an, der die Sprache: $0^n10^n1^n01^n$ erkennt!

9. Zwei Automaten senden immer abwechselnd im Takt ihre Stationsnummer ("1" bzw. "2"). Ein dritter Automat hört mit und analysiert die Signale. Er soll Alarm schlagen ("akzeptieren"), wenn einer der anderen Automaten ausfällt! Geben Sie einen Automaten an, der diese Aufgabe erfüllt!

10.

x. Ein Automat erkennt mittels Lichtschranke, ob ein Objekt sie unterbrochen hat. In dem Fall sendet dieser Automat über Funk eine "1" (ansonsten immer im Takt eine "0"). Ein anderer – zu realisierender Automat – soll das gesendete Signal abhören und beim Empfang einer "1" eine "2" als Quittung senden (sonst eine "3")!

3.2.8. (deterministische) Keller-Automaten



Will man bei einem großen Park, mehrgebäudigen Museum oder gar einem komplexen Labyrinth (oder Höhle) mit nur einem Ein-und Ausgang wissen, ob alle hineingegangenen Personen auch wieder herausgekommen sind, dann muss man sich die Zahl der Besucher irgendwo merken. Aber genau diese Eigenschaft – also eine Speicher – besitzen die bisher besprochenen Automaten nicht. Problematisch war auch die Beschränktheit der Automaten z.B. hinsichtlich der Klammer-Bearbeitung. Die exakte Anzahl zusammengehörender öffnender und schließender Klammern ist zumindestens universell mit den besprochenen Automaten nicht möglich.

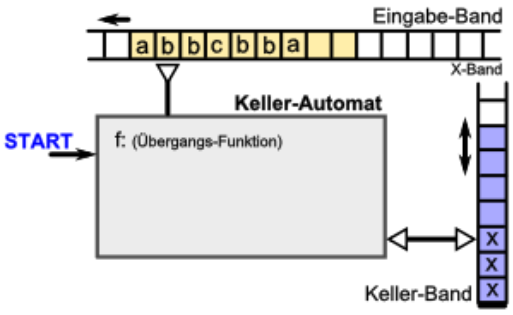
Eine Möglichkeit zur Lösung des Problems wäre natürlich ein Zähler, der z.B. für öffnende Klammer hochzählt und für schließende runter. Aber die Zahl der Klammern interessiert eigentlich gar nicht. Man müsste nur irgendwie organisieren, dass für jede öffnende Klammer auch irgendwann mal eine schließende kommt.

> Treffen sich zwei Mathematiker vor einem Kindergarten. Sie sehen drei Kinder rauskommen und fünf hineingehen.
> Darauf sagt der eine Mathematiker zu dem anderen: "Wenn jetzt noch zwei Kinder rauskommen, ist kein Kind mehr im Kindergarten."

(Vielleicht noch mal als Hinweis: Der Automat kann nicht prüfen, ob der Term mathematisch einen Sinn macht. Das ist Semantik und da hapert es derzeit noch bei der Computer-Aufarbeitung.)

Ein Automat für die Zähl-Aufgabe im Museum braucht garnicht wirklich zählen zu können, obwohl das vielleicht interessant wäre, es reicht z.B. sich für die eintretenden Personen immer einen Streichholz hinzulegen. Verlässt eine Person die Einrichtung, dann nimmt man wieder einen Streichholz weg. Am Ende des Tages sollten keine Streichhölzer übrig bleiben.

Ein Keller-Speicher ist genauso ein Stapel-Speicher. Jedesmal wenn eine Passung (Ein- und Ausgang, öffnende und schließende Klammer, …) kontrolliert werden soll, dann wird etwas auf den Stapel gelegt. Tritt das passende Ereignis / die Eingabe ein, dann wird das Merkzeichen vom Keller-Speicher entfernt.



Bei Keller-Speichern sprechen wir auch von LIFO-Speicher. Die Buchstaben stehen für **l**ast-**i**n-**f**irst-**o**ut.

Bei den Keller-Automaten (Abk.: KA) – also deterministischen endlichen Automaten mit einem Keller-Speicher – handelt es sich ebenfalls um Akzeptoren.

Es interessiert als nur, ob das Eingabe-Wort zur Sprache des Automaten passt oder nicht.

Sachlich gehören zu den deterministischen Keller-Automaten die kontextfreien Sprachen (CHOMSKY-Typ-2-Sprachen). Ein deterministischer KA ist somit mächtiger als ein EA, der Typ-3-Sprachen (reguläre Sprachen) erkennt. Mächtiger sind die TURING-Maschinen, die wir später besprechen (→ x.y.z. TURING-Maschinen).

Im englisch-sprachigen Bereich spricht man auch vom **p**ush**d**own **a**utomaton (PDA).

Definition(en): (deterministische) Keller-Automat (KA)
Ein deterministischer Keller-Automat ist ein deterministischer endlicher Automat mit einem LIFO-Speicher (Keller-Speicher).
Ein deterministischer Keller-Automat ist ein Sept-Tupel (7-Tupel) $M = (Z, \Sigma, \Gamma, f, z_0, \#, Z_E)$ mit: Z … endliche Menge der Zustände Σ … Eingabe-Alphabet; $Z \cap \Sigma = \varnothing$ Γ … Keller-Alphabet; $Z \cap \Sigma = \varnothing$ f … Überführungs-Funktion (totale Funktion); $Z \times (\Sigma \cup \varepsilon) \times \Gamma \rightarrow P(Z \times \Gamma^*)$ z_0 … der Start-Zustände; $z_0 \in Z$ # … das Anfangs-Symbol des Keller-Speichers Z_E … endliche Menge der End-Zustände; $Z_E \subseteq Z$

Neben der klassischen Definition über das Hept-Tupel (Sept- bzw. 7-Tupel) gibt es auch eine über Hex-Tupel (6-Tupel). In diesem Fall werden keine Endzustände (Z_E) definiert, sondern es reicht, wenn am "Ende" der Abarbeitung des Eingabe-Wortes der Keller-Speicher leer ist.

Überführungs-Funktion eines Keller-Automaten

Die Überführungs-Funktion beschreibt den Übergang von ein Konfiguration (→) aus aktuellem Zustand, dem gelesenen Zeichen vom Eingabe-Band sowie dem obersten Symbol auf dem Keller-Speicher (oberstes Stapel-Element) den nachfolgenden Zustand sowie eine Operation auf dem Keller.

aktueller Zustand	Eingabe-Symbol	aktueller Keller		neuer Zustand	neuer Keller
			→		
			→		
			→		
			→		

3.2.8.1. Arbeit eines Keller-Automaten

Keller-Automat ließt – so wie die anderen Automaten – das Eingabe-Band Zeichen für Zeichen. Gleichzeit wird auch immer die oberste Position des Keller-Speichers gelesen. Aus dem aktuellen Zustand, dem Eingabe-Symbol und dem Keller-(Top-)Symbol ergibt sich der Zustands-Wechsel und ein neues Symbol für den den Keller-Speicher.

Wenn das Eingabe-Band abgearbeitet ist und auch der Keller-Speicher leer ist, dann gilt das Eingabe-Wort als akzepiert bzw. zur Sprache des Automaten gehörend.
Beim Arbeiten der Keller-Automaten lässt man i.A. einige Ausnahmen / Sonder-Behandlungen zu. Dazu gehört, dass z.B. auch mal kein (echtes) Symbol vom Band gelesen wird. Das Band bleibt praktisch stehen und es wird ein leeres Symbol (ε) gelesen.
Bei nicht-deterministischen KA (→ x.y.z. nicht-deterministische Keller-Automaten) ist es zudem möglich, dass der Automat zwischen verschiedenen Übergängen "wählen" kann. Besser wäre wahrscheinlich die Aussage, dass er ausprobiert, welche ev. zum Erfolg (zur Akzeptanz) führt.

Keller-Operationen

• **push**	push(p, q) = pq	Einschreiben / Abspeichern eines Keller-Elementes Hinzufügen eines Keller-Elementes (obenauf)	p, K → pK
• **pop**	pop(py) = p	Löschen / Entfernen des obersten Keller-Elementes	pK → K K → ε
• **top**	top(py) = y	Lesen des obersten Keller-Elementes	K → K

Konfiguration und Konfigurations-Folge eines Keller-Automaten

Die Konfiguration (also die Arbeits-Situation) eines Keller-Automaten ist ein Tripel aus dem aktuellen Zustand, dem gelesenen Zeichen und einem Keller-Wort.

3.2.8.2. Simulation eines Keller-Automaten

Keller-Automat

AnfangsZustand setzen Z_0
Lesekopf auf 1. Position (X) i=1
Schreib-Lese-Kopf des Kellerspeicher auf 1. Position (leerer Keller)
Schreiben des Keller-Start-Zeichens in Keller
Eingabe-Zeichen x_1 lesen
Lesen der Keller-Spitze
falls kein STOP

? definierte (x , z ,α)	
JA	*NEIN*
ev. in Keller schreiben; Keller-Band neu positionieren (++1)	*STOP*
Folge-Zustand $z_{i,\alpha}$	
Eingabe-Band positionieren (++1)	
Eingabe-Zeichen x_i lesen	

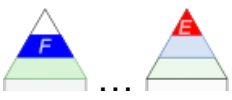
```
% Programm zur Simulation eines Keller-Automaten
% zur Palindrom-Testung
% Defintion des Startzustands
startzustand(z0).

% Defintion der Überführungs-Funktionen
ueberfuehrung(z0,X,A,z0,[A,X]).
ueberfuehrung(z0,A,A,z1,[]).
ueberfuehrung(z0,_,epsilon,z1,[]).
ueberfuehrung(z1,A,A,z1,[]).
ueberfuehrung(z0,[$],epsilon,z0,[]).
ueberfuehrung(z1,[$],epsilon,z1,[]).

% Ableitungs-Funktionen
Z: [X|K]:[A,W]) ==> (Z1:YK:W):- ueberfuehrung(Z, X, A, Z1, Y),
                                append(Y, K, YK).
Z: [X|K]:[A,W]) ==> (Z1:YK:W):- ueberfuehrung(Z, X, epsilon, Z1, Y),
                                append(Y, K, YK).

% Defintion des Akzeptor
akzeptiert(W):- startzustand(z0),
                (Z0:'$':W) ==>* (_Z:[]:[])).
```

Q: /2, S. 92 f/; leicht geänd.: dre

Zusatz-Info:
http://www.informatik.uni-rostock.de/~wid/vorl8 (durch Ändern der Nummer lassen sich weitere Kapitel aufrufen!)

Konfiguration muss nun auch die Keller-Funktion und den Keller-Stand enthalten.

3.2.8.3.Zwei-Keller-Automaten



Durch zwei Keller lässt sich ein Daten-Band simulieren. Der eine Keller speichert die Daten vor der Lese-Position und der andere Keller repräsentiert sowohl die Lese-Position (Top) und den nachdfolgenden Teil des Daten-Bandes.
Damit sind Zwei-Keller-Automaten einer TURING-Maschine gleichwertig. Anders ausgedrückt: Ein Keller-Automat mit zwei Keller-Speichern ist automatisch eine TURING-Maschine.

Aufgaben:

1.

2.

x. Der Keller-Automat K ist definiert durch:

KA = ($Z=\{z_0,z_1,z_2,z_3,z_4\}$, $\Sigma=\{0,1\}$, $\Gamma=\{N,E,k\}$, f, $Z_S=\{z_0\}$, k, $Z_E=\{z_4\}$)

f = …

$(0,z_0,k)$	=	(z_1,Nk)	$(1,z_0,k)$	=	(z_2,Ek)	(Λ,z_0,k)	=	(z_4,k)
$(0,z_1,k)$	=	(z_1,Nk)	$(1,z_1,k)$	=	(z_4,k)	(Λ,z_1,k)	=	(z_4,k)
$(0,z_1,N)$	=	(z_1,NN)	$(1,z_1,N)$	=	(z_1, Λ)	(Λ,z_1,N)	=	(z_1,NN)
$(0,z_2,E)$	=	(z_3,Λ)	$(1,z_2,E)$	=	(z_2,EE)	(Λ,z_2,E)	=	(z_2,EE)
$(0,z_3,E)$	=	(z_3,NN)	$(1,z_3,k)$	=	(z_3,Ek)	(Λ,z_3,k)	=	(z_4,k)
			$(1,z_3,E)$	=	(z_3,EE)			

a) Prüfen Sie durch Ableitungen / Berechnungen, ob K die folgenden Worte akzeptiert: 0011, 0101, 10101, 1100, 1010, 000, 111

b) Beschreiben / definieren Sie die Sprache, die K akzeptiert!

c) Erstellen Sie den Automaten in einem geeigneten Programm und prüfen Sie die oben angegebenen Eingabe-Worte!

d) In welchem Zustand befindet sich der Keller nach der Eingabe von $1^4 0^3$*?*

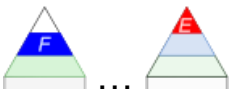
x.

für die gehobene Anspruchsebene:

x.

x.

3.2.9. nicht-deterministische Keller-Automaten



Definition(en): (nicht-deterministische) Keller-Automat (NKA)
Ein nicht-deterministischer Keller-Automat ist ein Sept-Tupel (7-Tupel) $M = (Z, \Sigma, \Gamma, f, z_0, \#, Z_E)$ mit: Z … endliche Menge der Zustände Σ … Eingabe-Alphabet; $Z \cap \Sigma = \varnothing$ Γ … Keller-Alphabet; $Z \cap \Sigma = \varnothing$ f … Überführungs-Funktion (partielle Funktion); $Z \times (\Sigma \cup \varepsilon) \times \Gamma \rightarrow P(Z \times \Gamma^*)$ z_0 … der Start-Zustände; $z_0 \in Z$ $\#$ … das Anfangs-Symbol des Keller-Speichers Z_E … endliche Menge der End-Zustände; $Z_E \subseteq Z$

Mit Hilfe eines nicht-deterministischen Keller-Automaten lassen sich Sprachen vom Typ 2 nach CHOMSKY erkennen.

Bei nicht-deterministischen Keller-Automaten sind mehrere alternative Übergänge für eine Konfiguration möglich. Der Automat muss die geeignete Überführung "erraten" / ausprobieren.

Zu jedem nicht-deterministischen Keller-Automaten gibt es einen äquivalenten nicht-deterministischen Keller-Automaten, der nur genau einen Zustand besitzt.

Aufgaben:

1.

x. Gesucht sind DKA's für die folgenden Sprachen! (Ersatzweise kann auch ein NKA angegeben werden!)

a) $\{a^n b^m ; n \leq m\}$
b) $\{a^n b^m ; n \geq m\}$
c) $\{x^a X^b x^a ; a,b \geq 0\}$
d) $\{0^n 1^n ; n \geq 1\} \cup \{0^n 1^{2n} ; n \geq 1\}$

x.

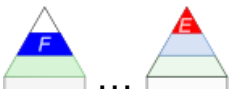
für die gehobene Anspruchsebene:

x.

Äquivalenz von deterministischen und nicht-deterministischen Keller-Automaten

Während deterministische endliche Automaten (DEA, DFA) und nicht-deterministische endliche Automaten (NEA, NFA) äquivalent sind, gilt dies nicht für Keller-Automaten.

3.2.10. TURING-Maschinen

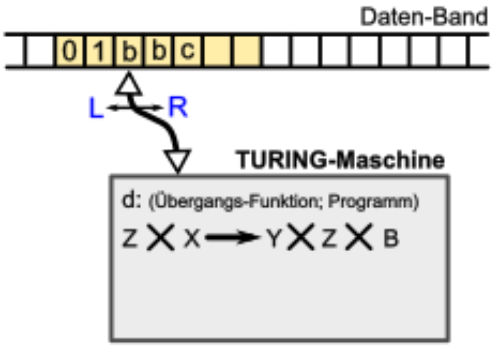


auch als TURING-Automat bezeichnet,
Maschinen-Bezeichnung besser, auch im Verständnis dessen, das es sich hierbei um einen Automaten mit Ausgabe handelt, deshalb auch die Ankürzung TM üblicher als TA

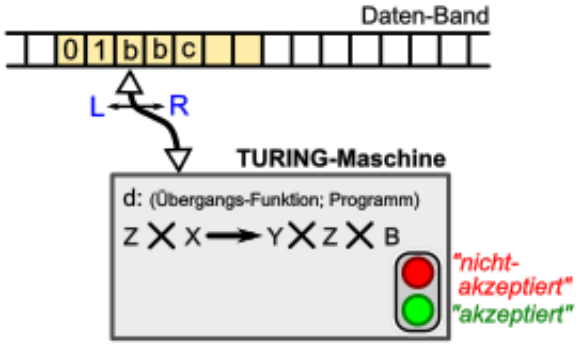
1936 – also lange vor den ersten echten Computern – durch Alan TURING () als abstraktes und universelles Modell vorgestellt
kann als eine der größten intellektuellen Entdeckungen eingestuft werden
als Mathematiker beschäftigte es sich mit Algorithmen und seine Maschine sollte eben genau solche Algorithmen verarbeiten können
ihm ging es darum aufzuzeigen, wie automatische Rechner funktionieren könnten und vor allem was sie alles berechnen können
TURING-Maschinen gehören heute zu den am Meisten benutzten Modellen in der Theoretischen Informatik

TURING's Weg zur Maschine
wenn Menschen eine systematische Berechnung durchführen, nutzen sie zumeist ein Rechenblatt zum Notieren der gegebenen Zahlen, der Zwischenwerte und des Ergebnisses
weiterhin wird mit einem Stift das Beschreiben des Blattes vorgenommen und ev. mit einem Radiergummi für die Möglichkeit der Korrektur gesorgt
insgesamt bearbeitet man nur endlich viele Zeichen, wie sie eben auf einem Rechenblatt Platz finden
für die Berechnungen selbst (z.B. die Addition zweier Zahler) gibt es eine endliche Berechnungs-Vorschrift

wir unterscheiden deterministische und nichtdeterministische TURING-Maschinen
Nicht- deterministische TURING-Maschinen werden oft mit der Abkürzung NTM oder NDTM. Sie benutzt statt einer vollständigen Funktion nur eine teiweise gefüllte (partielle) Übergangs-Relation (Übergangs-Tabelle). In der Praxis wird aber auch die Tabelle auch als Übergangs-Funktion geführt. Hier wird dann Wert auf das beschreibenden Wörtchen "partiell" gelegt.



verfügen über eine Steuer-Werk, in dem sich das Programm befindet
Takt-getrieben
System-eigener Takt, der auf das Ergebnis selbst keinen Einfluß hat



sind Automaten mit quasi unbegrenztem Speicher
im klassischen TURING-Automat nur ein Speicher(-Band), in erweiterten Modellen aber auch mehrere Speicher(-Bänder) zulässig / verwendet
besitzen einen Start-Zusand und auch mindestestens einen End-Zustand
Abhängigkeit des Folge-Zustands vom Vor-Zustand und dem Eingabe-Symbol (auf dem Eingabe-Band)
zur möglichen Ausgabe auf dem Speicher-Band kommt nun noch die Bewegung des Schreib-Lese-Kopfes (auf dem Speicher-Band) hinzu

Band ist praktisch unendlich lang, wird aber nur in einem endlichen Bereich genutzt
auf beiden Seiten vom eigentlichen Daten-Bereich befinden sich unendlich viele leere Zellen
in der Praxis betrachtet man der Einfachheit halber nur den mit Daten beschriebenen (endlichen) Band-Ausschnitt
leere Zellen werden entweder als Quadrat (□) oder als informatischer Leerzeichen-Unterstrich (␣) notiert, man bezeichnet diese Symbole auch als blank's
in einigen Definitionen ist das Band nur auf der rechten Seite unbegrenzt und auf der linken Seite durch das Band-Begrenzungs-Zeichen **$** abgeschlossen

Definition(en): TURING-Automat (TM, TA)

Eine TURING-Maschine ist ein endlicher Automat mit Speicher (/ einem Speicher-Band), bei dem über ein Programm (in Form einer tabellarischen Übergangs-Funktion) aus den gelesenen Band-Informationen neue Ausgabe-Daten für das Speicherband berechnet werden. Weiterhin sind im Programm der nächste Befehl und die Bewegung auf dem Speicherband festgelegt.

Eine TURING-Maschine ist ein Sept-Tupel (7-Tupel) $A = (Z, \Sigma, \Gamma, \delta, z_0, \square, Z_E)$ mit:

Z … endliche Menge der Zustände; $|Z| < \infty$
Σ … Eingabe-Alphabet; $|\Sigma| < \infty$, $Z \cap \Sigma = \varnothing$
Γ … Band-Alphabet; $|\Gamma| < \infty$; $\Sigma \subseteq \Gamma$
d, δ, Δ … Überführungs-Funktion; $Z \times \Gamma \rightarrow Z \times \Gamma \times B$ ***(deterministische TM)***
d, δ, Δ … partielle Überführungs-Funktion; $Z \times \Gamma \rightarrow \mathbb{P}(Z \times \Gamma \times B)$ ***(nicht-determ. TM)***
z_0 … Start-Zustand; $z_0 \in Z$
□ … Leer- bzw. Trenn-Zeichen □ $\in \Sigma$
Z_E … endliche Menge der akzeptierten Zustände (End-Zustände); $Z_E \subseteq Z$
B … fest definierte Menge der Kopf-Bewegungen ***{L,N,R}***

Einige Definitionen verzichten auf die Angabe des leeren Band-Symbol's. Dann reduziert sich die Definition auf ein Hex-Tupe (6-Tupel): $A = (Z, \Sigma, \Gamma, \delta, z_0, Z_E)$

Die partielle Überführungs-Funktion **d** oder δ (klein delta) liefert zu einem (aktuellen) Zustand und dem (aktuellen) gelesenen Band-Symbol den nächsten Zustand, eine Ausgabe auf das Speicher-Band sowie eine nun folgende Bewegung (des Bandes / des Lese-Kopfes) zurück. Für die Bewegung gibt es drei vordefinierte Möglichkeiten. Das Symbol L steht für eine Bewegung des Lese-Kopfes nach Links, so dass das ausgehend von der aktuellen Lese-/Schreib-Position das linke Symbol erreicht wird. Mit R entsprechend das rechte Band-Symbol. Betrachtet man das Band als beweglich, dann wird es bei L nach rechts und bei R eben nach links bewegt.
Eine Bewegung kann aber auch ausbleiben, d.h. das aktuelle (geschriebene) Symbol ist die Eingabe für den nun folgenden Zustand. Das Symbol hierfür ist emist ein N.

tabellelarische Dastellung der Überführungs-Funktion wird auch TURING-Tabelle genannt.
Diese enstspricht praktisch der nebenstehenden Struktur, nur dass man auf den Ableitungs-Pfeil verzichtet.

aktueller Zustand	Band-Symbol		neues Band-Symbol	neuer Zustand	Kopf-Richtung
		→			
		→			
		→			
		→			
		→			
		→			
		→			

kompaktere Tabelle

TURING-Programm

	Eingaben		
Zust.	**0**	**1**	**□**
z0	1,z0,R	0,z1,R	□,z0,L
z1			

die Überführungs-Funktion ist / kann deshalb partiell (also nicht vollständig) sein, weil z.B. für einen End-Zustand keine weitere Berechnung definiert sein muss, die TURING-Maschine hält (an). Hiermit ist ein wichtiges Problem der theoretischen Informatik assoziiert – das sogenannte Halte-Problem. Da aber später mehr (→).

Automat läßt sich auch als Graph darstellen. Die Darstellung der Zustände bleibt, wie bei den anderen Automaten-Graphen festgelegt. Die Kanten werden mit dem gelesenen Zeichen vom Band und dann nach einem senkrechten Strich abgetrennt das zu schreibende Symbol und die Kopf-Bewegung notiert.

gelesenes_Symbol* | *zuschreibendes_Symbol* , *Kopf-Bewegung

in der Praxis findet man auch die Notierung als Tripel:

(*gelesenes_Symbol* , *zuschreibendes_Symbol* , *Kopf-Bewegung*)

Beschreibung einer TURING-Maschine auch über die Komponenten:

- **externes Alphabet** endliche Menge von Zeichen für das Speicher-Band, wobei ein Zeichen ein Trenn- oder Leer-Zeichen sein muss
- **internes Alphabet** endliche Menge von (Automaten-)Zuständen, wobei ein Zustand als Start-Zustand und ein Zustand als Abbruch-Zustand dient
- **Übergangs-Tabelle** Relation mit den "Anweisungen", wie ausgehend vom aktuellen Zustand und dem gerade gelesenen Zeichen auf dem Speicher-Band weiter verfahren wird (die Anweisung bestimmt den nächsten Zustand, das auf das Speicher-Band zu schreibende Zeichen und eines Bewegung des Schreib-Lese-Kopfes auf dem Speicher-Band)

möglich

für den Endzustand / akzeptierenden Zustand bzw. dem STOP-Zustand wird oft eine Nummer 99 oder eben 999 usw. gegeben

Konfiguration einer TURING-Maschine

zusammengesetzt aus aktuellem Zustand, der Position des Lese-/Schreib-Kopfes (Band-Position) sowie die auf dem Band befindlichen / gespeicherten Symbole

Schritt / Takt	**Zustand**	**Band + Kopf-Position**

im Arbeits-Takt wird die Überführungs-Funktion berechnet; Ergebnis ist eine neue Konfiguration (wird diese als Start für eine neue äquivalente TURING-Maschine benutzt, dann ergibt sich am Schluß das gleiche Ergebnis, genau so wie es die Ausgangs-Maschine getan hätte)
gilt natürlich nur für die deterministische TURING-Maschine, nicht-deterministische Maschinen lassen ausgehend von einer Konfiguration mehrere neue Konfigurations-Folgen zu
die Überführungs-Funktion ermittelt den neuen Zustand, ein zu schreibendes Zeichen auf dem Band (akt. Position) und eine Bewegung des Schreib-Lese-Kopfes

kompakte Konfiguration auch als Wort **w** aus $\Sigma^* Q \Sigma^*$
damit ist ein w eine Folge aus dem besuchten und ev. auch schon geänderten Band-Abschnitt α (praktisch die bisherigen Ausgaben), dem aktuellen Zustand **z** und dem noch nicht bearbeiteten Band-Abschnitt β (praktisch die noch offenen Eingaben)

$$w = \alpha\, z\, \beta$$

Konfigurations-Folge $W = w_1 \vdash w_2 \vdash \ldots \vdash w_n$

Programm einer TURING-Maschine

Translations-Tabelle ist 5-Tupel:
(z, x, y, z', b)

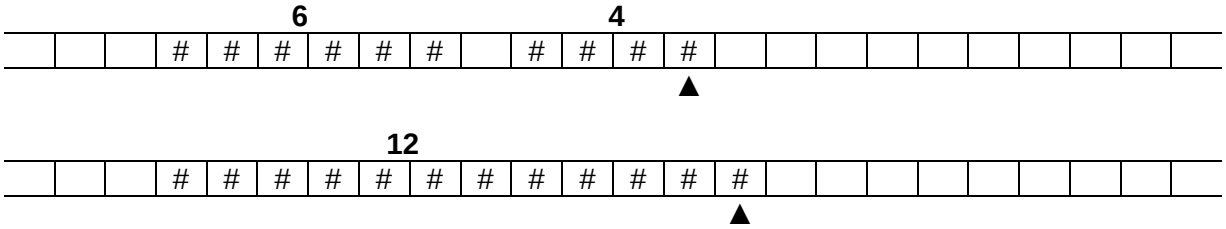
z … aktueller Zustand
x … gelesenes zeichen vom Band
y … zu schreibendes Zeichen auf das Band
z' … Folge-Zustand
b … Bewegung des Lese-Schreib-Kopfes (**L**inks, **R**echts, **N**oop)

Aufgaben:

1. *Leiten Sie aus der nebenstehenden Übergangs-Funktion den Graph einer TURING-Maschine TM ab! Geben Sie dann die vollständige TM an (Def.!)!*

aktueller Zustand	Band-Symbol		neues Band-Symbol	neuer Zustand	Kopf-Richtung
0 (add)	0	→	1	1	L
0 (add)	1	→	0	0	R
0 (add)	□	→	1	1	L
1 (back)	0	→	0	1	L
1 (back)	1	→	1	1	L
1 (back)	□	→	□	2	R
2 (stop)	?	→	-	-	-

2. *Vollziehen Sie die Funktion der TM nach! Verwenden Sie als Band-Belegung (1)1101! (hier soll das niederwertigste Bit der Binärzahl links stehen, Schreib-Lese-Kopf steht links auf dem ersten Symbol) (Ziel-Ergebnis: (0)0011)*
   Was leistet die Maschine?
3. *Simulieren Sie die Maschine in einem geeigneten Programm!*
4. *Entwickeln Sie eine TURING-Maschine, die eine normal notierte Binärzahl (niederwertigstes Bit rechts) genau so berechnet, wie die obige Maschine! Geben Sie die Definition dieser Maschine an!*
5. *Simulieren Sie Ihre TM mit einem geeigneten Programm (z.B. mit der Eingabe (1)0111! ((1)1000)*
6. *Auf dem Band einer TM steht der Lese-Schreib-Kopf rechts auf dem letzten Symbol einer Eingabe-Sequenz. Die Sequenz kann aus einer oder zwei Zählzahlen bestehen, die untereinander durch ein Trenn-Zeichen unterscheidbar sind. Erstellen Sie eine TM, die den Lese-Schreib-Kopf auf das ganz linke Symbol der Sequenz positioniert!*
   (Ein erstes Teilziel könnte eine TM sein, die nur eine Zählzahl behandelt (unteres Beispiel-Band).)



7. *Simulieren Sie Ihre TM mit einem geeigneten Programm und den angegebenen Beispiel-Eingaben! Sollte Ihr Simulations-Programm keine besondere Start-Position für den Schreib-Lese-Kopf zulassen, versuchen Sie ein anderes! Geht auch das nicht, dann tauschen Sie mit einem Kurs-Teilnehmer die Definitionen und simulieren per Hand!*

TURING-Berechenbarkeit

eine Funktion oder ein Algorithmus, der mit einer TURING-Maschine umsetzbar / berechenbar ist, wird als TURING-berechenbar verstanden

eine Funktion **f** gilt als "berechenbar", wenn es einen Algorithmus gibt, der nach endlich vielen Schritten **f** erfüllt / berechnet

alle berechenbaren Funktionen sind auch TURING-berechenbar, was auch anders herum gilt
da Algorithmen auch als Arbeits-Vorschriften für eine TURING-Maschine beschrieben werden können (→), folgt daraus, dass alle Algorithmen auf einer TURING-Maschine berechnet oder simuliert werden können

TURING-Maschinen können auch Sprachen vom Typ 0 (nach CHOMSKY) akzeptieren, da die Grammatiken für diese in TURING-berechenbare Algorithmen umgesetzt werden können

nicht-deterministische besitzen mehrere Aktionen () für eine Kombination aus aktuellem Zustand und gelesenem Band-Symbol
solche Maschinen können zufällig schnell eine Lösung des Algorithmus ableiten
nicht-deterministische TURING-Maschinen sind genauso Leistungs-fähig, wie deterministische

Definition(en): TURING-Berechenbarkeit
Eine Funktion ist TURING-berechenbar (bzw. rekursiv), wenn es eine TURING-Maschine gibt, die zu einer Eingabe den passenden Funktions-Wert berechnet.

Definition(en): rekursive Sprache / TURING-entscheidbare Sprache
Eine Sprache ist TM-entscheidbar / heißt rekursiv, wenn es eine TURING-Maschine gibt, die zu jeder Eingabe richtig (Wort gehört zur Sprache) terminiert.

Halte-Problem

eine TURING-Maschine akzeptiert ein Wort **w**, wenn sie sich am Ende in einem End-Zustand befindet **$w \in L(TM)$**
das Wort ist dann Element der Sprache der TURING-Maschine

es stelt sich die Frage, ob ein Algorithmus zu einem Ende kommt
für viele Algorithmen kann man das leicht beantworten (z.B. einfache Sequenz; Endlos-Schleife)
TURING konnte nachweisen, dass es keinen Algorithmus gibt, der die Frage für alle / beliebige Algorithmen beantworten kann
Beweis erfolgte mit seiner theoretischen Maschine

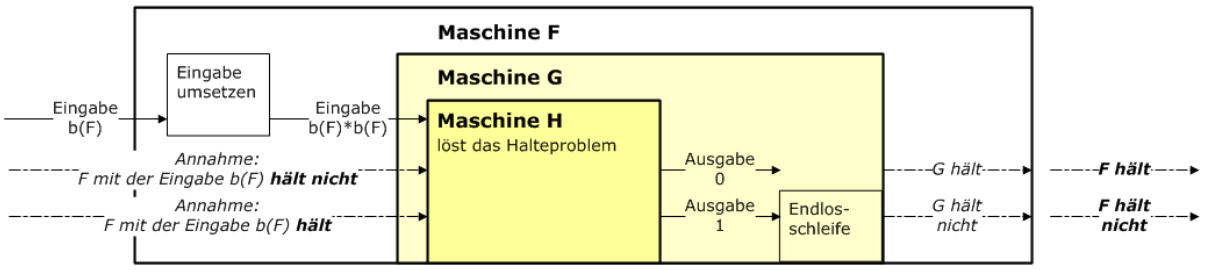
Beweis-Idee
Halte-Problem ist entscheidbar, wenn das Halte-Problem selbst und auch das Komplement semi-entscheidbar ist
semi-entscheidbar steht dafür, dass es eine TURING-Maschine (TM) gibt, welche die Worte einer Sprache akzeptiert, die anderen Worte aber nicht, die TM akzeptiert nur erkennbare Sprachen, nicht aber entscheidbare
daraus folgt, dass das Halte-Problem semi-entscheidbar sein muss
nimmt man nun eine universelle TURING-Maschine (uTM), die als Eingabe die Beschreibung einer TM bekommt, dann hält diese, wenn diese TM bei Eingabe eines Wortes w hält

die uTM kann also auch ihren eigenen Code zur Beurteilung vorgelegt bekommen

Diagonal-Argument

g(i,w)		**w = …**					
		1	**2**	**3**	**4**	**5**	**6**
i = …	**1**	1	ud	ud	1	ud	1
	2	ud	ud	ud	1	ud	ud
	3	ud	1	ud	1	ud	1
	4	1	ud	ud	1	ud	ud
	5	ud	ud	ud	1	1	1
	6	1	1	ud	ud	1	ud
g(i,i)		**1**	**ud**	**ud**	**1**	**1**	**ud**
f(i)		**1**	**ud**	**ud**	**1**	**1**	**ud**

ud … undefininiert



graphische / schematische Darstellung der Beweis-Kontruktion zum Halte-Problem
Q: de.wikipedia.org (Mussklprozz)

algorithmisch ist das Halte-Problem nicht entscheidbar
der Begriff "Halte-Problem" stammt nicht von TURING selbst, sondern wurde erst später von Martin DAVIS () geprägt

Halte-Problem erscheint abstrakt und völlig abgehoben oder theoretisch. Aber in der Praxis gibt es tatsächlich Programme / Algorithmen / Probleme, die als nicht berechenbar gelten – zumindestens in vertretbarer Zeit

Zerlegung einer natürlichen Zahl in Prim-Faktoren

Eigentlich ein einfaches / triviales Problem, dass prinzipiell schon Schüler in der Mitelstufe lösen können
Man nehme dazu beginnend mit 2 jede Primzahl und versuche die zu testende Zahl durch diese Primzahl zu teilen. gelingt das, dann ist die Primzahl ein Primfaktor und wird sich für das finale Produkt gemerkt.
Am Ende werden alle gemerkten Primzahlen als Produkt geschrieben.

zu testende Zahl: 46

1. Primzahl	2	46 / 2 = 23	→ teilbar	→ 2 merken
2. Primzahl	3	46 / 3 = 15 R 1	→ nicht teilbar	→ -
3. Primzahl	5	46 / 5 = 9 R 1	→ nicht teilbar	→ -
4. Primzahl	7	46 / 7 = 6 R 4	→ nicht teilbar	→ -
5. Primzahl	11	46 / 11 = 4 R 2	→ nicht teilbar	→ -
6. Primzahl	13	46 / 13 = 3 R 7	→ nicht teilbar	→ -
7. Primzahl	17	46 / 17 = 2 R 12	→ nicht teilbar	→ -
8. Primzahl	19	46 / 19 = 2 R 8	→ nicht teilbar	→ -
9. Primzahl	23	46 / 23 =2	→ teilbar	→ 23 merken

Ergebnis: 46 = 2 * 23

für große Zahlen wird der Test so lang, dass er praktisch nicht mehr berechnet werden kann, man bräuchte auch mit modernsten Computern irgendwann zu lange.
Allerdings steckt da noch ein anderes Problem dahinter. Die Aussagen, die wir eben getroffen haben, gelten für den obigen – traditionellen Algorithmus. Für einen anderen Algorithmus muss das wieder neu geprüft werden. Vielleicht kommt der mit ein paar Divisionen usw. usf aus und ist dann sofort fertig. Scheinbar gibt es aber derzeit keinen solchen Algorithmus. Wenn der existieren würde, dann wären kryptographische Verfahren, wie RSA, sofort hinfällig, weil dan die wichtigen Parameter der Verschlüsselung einfach zu berechnen wären..
Solche Probleme werden NP-vollständige Probleme genannt. NP steht dabei für nicht deterministisch polynomiell. Gemeint ist, dass eben die Berechnungs-Zeit nach bisherigem Stand polynomial steigt. Gesucht ist ein einziges Verfahren aus dieser Gruppe, dass schneller als polynomial lösbar ist. Hat man ein Verfahren, dann kann man alle NP-Probleme lösen, da sie alle ineinander überführbar sind. (Also hier gibt es noch ne FIELDS-Medaille (praktisch der NOBEL-Preis für Mathematik) zu gewinnen!)

interessante Links:
http://www.turingarchive.org/browse.php/B/12 (Original-Artikel von TURING (London Mathematical Society 1937))

CHURCH-TURING-These

die Klasse der intuitiv berechenbaren Funktionen stimmt mit den TURING-berechenbaren Funktionen überein

→ alles was überhaupt berechenbar ist, kann auch mit einer TURING-Maschien berechnet werden

→ Jede im intuitiven Sinn berechenbare Funktion ist TURING-Maschinen-brechenbar.

Definition(en): Algorithmus
Ein Algorithmus ist ein Verfahren, das von einer TURING-Maschine ausgeführt werden kann.

alle berechenbaren Funktionen können genau durch den Begriff Algorithmus beschrieben werden

gekoppelte TURING-Automaten sind der Problematik geschuldet, dass eine einfache Aufgabe schnell eines etwas komplexeren TURING-Automaten bedarf. Größere Probleme wären dann nur mit sehr komplexen Automaten realisierbar. Da man die Funktions-Fähigkeit des einfaches Systems schon geprüft / bewiesen hat, setzt man einfach auf die Kombination von einfacheren TURING-Automaten zu einem komplexeren Problem-Löser. Diese werden gekoppelte TURING-Automaten genannt.

Definiert man z.B. die folgenden TURING-Automaten:

Beispiele für "einfache" koppelbare TURING-Automaten

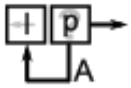
- **1** **Eins-Maschine** schreibt eine "1" auf das Band (aktuelle Position)
- **0** **Null-Maschine** schreibt eine "1" auf das Band (aktuelle Position)
- **A** **A-Maschine** schreibt eine "1" auf das Band (aktuelle Position)
- **B** **B-Maschine** schreibt eine "1" auf das Band (aktuelle Position)
- **r** **rechts-Maschine** bewegt das Band um eine Position nach rechts
- **l** **links-Maschine** bewegt das Band um eine Position nach links
- **p** **Prüf-Maschine** prüft, ob ein bestimmtes Symbol auf der Leseposition des Band's steht

dann kann man später einen gekoppelten Automaten durch Aneinanderreihung der Automaten-Symbole darstellen:

ArAr1l

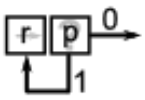
eine mögliche graphische Darstellung als Symbole mit dem Zeichen des "einfachen" TURING-Automaten

TURING-Maschine, die den linken Rand eines Blockes von A's auf dem Band sucht.

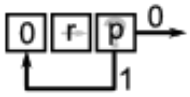


TURING-Maschine, die den rechten Rand eines Blockes von Einsen auf

dem Band sucht.



Maschine zum Löschen von Einsen in einem Block von links beginnend.




```
% Programm zur Simulation einer TURING-Maschine
% Defintion des Startzustands
startzustand(Z).

% Defintion des Endzustands
endzustand(Z).

% Defintion der Überführungs-Funktionen
ueberfuehrung(Z,A,Z1,B,X).

% Definition von Operatoren
:- op(700,xfx,==>).
:- op(700,xfx,==>*).

% Bildung der reflexiven transitiven Hülle
K1 ==>* K1
K1 ==>* K2 :- K1 ==>* K3, K3 ==>* K2.

quer([],['#']).
quer([A|W],[A|W]).

% Rechts-Bewegung
(V:Z:[A|W]) ==> (VB:Z1:WQ):- quer(W,WQ), ueberfuehrung(Z,A,Z1,B,r),
                             append(V,[B],VB).

% Links-Bewegung
(VC:Z:[A|W]) ==> (VQ:Z1:[C,B|W]):- ueberfuehrung(Z,A,Z1,B,r),
                                   append(V,[C],VC), quer(V,VQ).

% Defintion des Akzeptor
akzeptiert(W):- startzustand(Z0), ([]:Z0:W) ==>* (_:Z:_),
                endzustand(Z).
```

Q: /2, S. 107,193/; leicht geänd.: dre

kleine Automaten-Sammlung

TM: direkte Addition von Zählzahlen

Ausgangs-Situation:

			#	#	#	#	#	#		#	#	#	#										
			▲																				

direkte Addition meint hier, dass das Ergebnis direkt aus den Eingaben zusammengestellt wird
als Zählzahlen (hier 6 und 4) verstehe ich hier solche Zeichenmengen, die durch die Anzahl der Zeichen eine Zahl repräsentieren

Ergebnis:

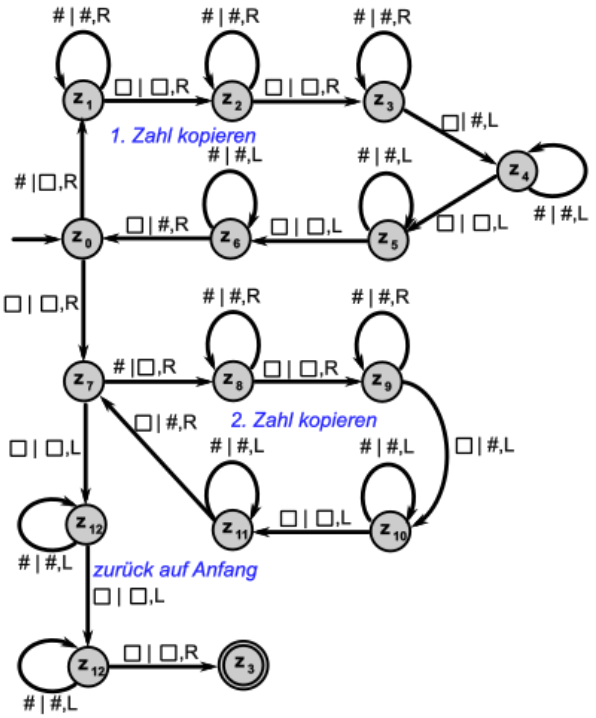
				#	#	#	#	#	#	#	#	#	#										
				▲																			

TM: indirekte Addition von Zählzahlen

Ausgangs-Situation:

			#	#	#	#		#	#	#													
			▲																				

bei der indirekten Addition werden die Eingaben (hier Zählzahlen) erhalten und das Ergebnis hinter diese auf das Band geschrieben
zwischen Eingaben und dem Ergebnis liegt ein Leerzeichen



Ergebnis:

			#	#	#	#		#	#	#		#	#	#	#	#	#	#					
			▲																				

indirekte Addition der beiden Zählzahlen (hier 4 und 3) zu einer neuen angehängten Zählzahl durch Hintereinander-Kopieren der beiden Zählzahlen

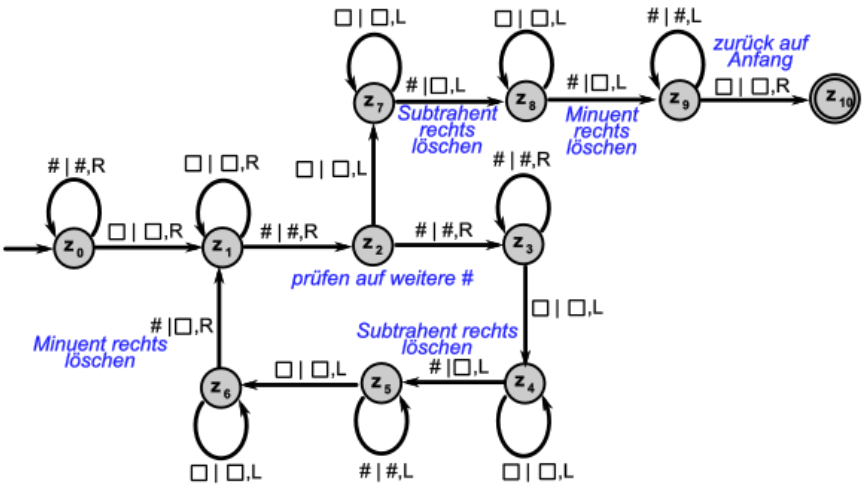
aktueller Zustand	Band-Symbol		neues Band-Symbol	neuer Zustand	Kopf-Richtung
0	☐	→	☐	7	R
0	#	→	☐	1	R
1	☐	→	☐	2	R
1	#	→	#	1	R
2	☐	→	☐	3	R
2	#	→	#	2	R
3	☐	→	#	4	L
3	#	→	#	3	R
4	☐	→	☐	5	L
4	#	→	#	4	L
5	☐	→	☐	6	L
5	#	→	#	5	L
6	☐	→	#	0	R
6	#	→	#	6	L
7	☐	→	☐	12	L
7	#	→	☐	8	R
8	☐	→	☐	9	R
8	#	→	#	8	R
9	☐	→	#	10	L
9	#	→	#	9	R
10	☐	→	☐	11	L
10	#	→	#	10	L
11	☐	→	#	7	R
11	#	→	#	11	L
12	☐	→	☐	13	L
12	#	→	#	12	L
13	☐	→	☐	14	R
13	#	→	#	13	L
14	☐	→	☐	- (*)	N
14	#	→	#	- (*)	N

TM: direkte Subtraktion von Zählzahlen

Ausgangs-Situation:

			#	#	#	#	#	#		#	#	#	#										
			▲																				

direkte Subtraktion, größere Zahl steht vorne (Anfangs-Position)



Ergebnis:

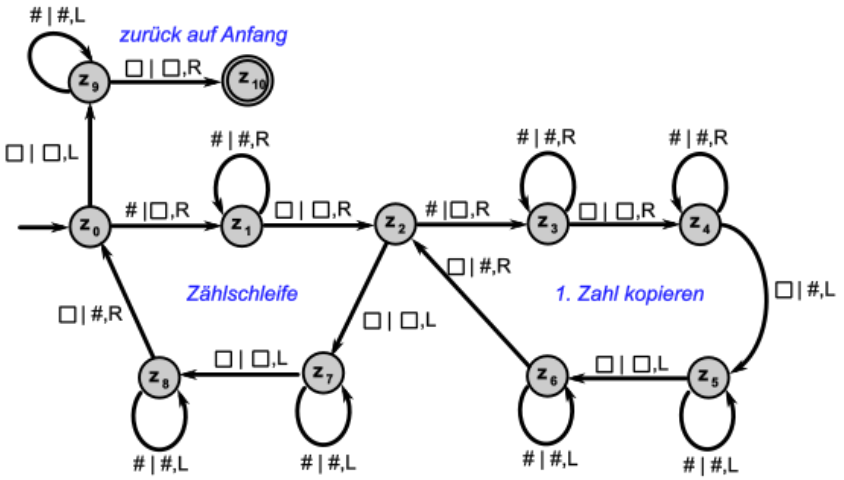
			#	#																			
			▲																				

TM: indirekte Multiplikation von Zählzahlen

Ausgangs-Situation:

			#	#		#	#	#	#	#													
			▲																				

indirekte Multiplikation



Ergebnis:

			#	#		#	#	#	#	#		#	#	#	#	#	#	#	#	#	#		
			▲																				

TM: indirekte Addition von Zählzahlen

Ausgangs-Situation:

					1	0	0	0	0	1													
					▲																		

Nullen-Verdopplungs-Maschine

Arbeitsweise:

aktueller Zustand	Band-Symbol		neues Band-Symbol	neuer Zustand	Kopf-Richtung
0	0	→	0	1	L
0	1	→	1	0	L
0	☐	→	☐	1	R
1	0	→	0	1	L
1	1	→	☐	2	R
1	☐	→			
2	0	→	☐	4	R
2	1	→	1	3	R
2	☐	→			
3	0	→	0	3	R
3	1	→	1	3	R
3	☐	→	1	7	N
4	0	→	0	4	R
4	1		1	4	R
4	☐		0	5	R
5	0				
5	1				
5	☐		0	6	L
6	0		0	6	L
6	1		1	6	L
6	☐		☐	2	R
7	?		-	-	-

Ergebnis:

			1	0	0	0	0	0	0	0	0	1											
			▲																				

TM: ???

Ausgangs-Situation:

			#	#	#	#	#	#	#	#	#		#	#	#	#							
			▲																				

Ergebnis:

													#	#	#	#		#	#				
			▲																				

TM: Erkennung der Sprache a^nb^n

Erkennen der Sprache a^nb^n

liest solange a's bis ein b gefunden wird
dann wird immer zwischen den a'a und b's hin und her gependelt und dabei immer ein b und ein a gelöscht
am Ende der b's (also Leerzeichen gefunden) prüft das Programm, ob noch ein a vorhanden ist

der mit **–** gekennzeichnete Zustand ist der reject-Zustand

aktueller Zustand	Band-Symbol		neues Band-Symbol	neuer Zustand	Kopf-Richtung
0	a	→		0	R
0	b	→	x	1	
0		→		-	
1	x	→		1	L
1	a	→	x	2	
1	$	→		-	R
2	x	→		2	R
2	b	→	x	1	
2	a	→		-	R
2		→		3	L
3	x	→		3	L
3	a	→		-	R
3	$	→		4*	R

er steht für eine definierte Nicht-Akzeptanz des Wortes

TM: ??? Erkennen von Buchstaben (beispielhaft von: a, b, c)

Erkennen von Buchstaben (hier a, b und c) und ersetzen durch eine 0

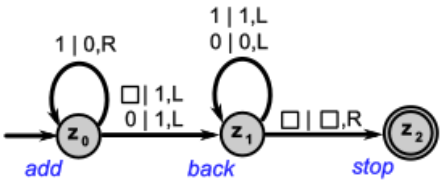
aktueller Zustand	Band-Symbol		neues Band-Symbol	neuer Zustand	Kopf-Richtung
0	0	→	0	0	R
0	1	→	1	0	R
0	a	→	0	0	R
0	b	→	0	0	R
0	c	→	0	0	R
0	☐	→	-	1	N
1	?	→	-	-	-
		→			

TM: Addition von 1 zu einer Binär-Zahl (umgedrehte Notierung)

gegeben ist eine Binär-Zahl auf dem Band, diese soll um Eins erhöht werden
zur Vereinfachung steht die Binärzahl von links nach rechts (kleinste zu größte Stelle) im Speicher-Band
die Maschine geht Zeichen-weise durch und schaut auf die Eingabe

aktueller Zustand	Band-Symbol		neues Band-Symbol	neuer Zustand	Kopf-Richtung
0 (add)	0	→	1	1	L
0 (add)	1	→	0	0	R
0 (add)	□	→	1	1	L
1 (back)	0	→	0	1	L
1 (back)	1	→	1	1	L
1 (back)	□	→	□	2	R
2 (stop)	?	→	-	-	-

bei einer 1 ergibt die Addition von 1 eine 0 an der Position und einen Übertrag auf die nachfolgende Binärstelle
ist eine 0 an der Position, dann ergibt sich eine 1 aus Ausgabe
in den Fall ist die Addition erledigt, weitere höhere Positionen müssen nicht mehr beachtet werden (für eine optimale Weiterarbeit sollte der lese-Kopf wieder an die Start-Position – also das niederste Bit zurückgefahren werden.



wird ein Leerzeichen gelesen, entspricht dies einer 0 und es wird eine 1 ausgegeben
das Zurückfahren erfolgt durch Lesen des Zeichen vom Band, vorhandene Nullen und Einsen werden wieder zurückgeschrieben, steht dort ein Leerzeichen ist man über das Ende der Binärzahl hinausgelangt und muss also eine Rechts-Bewegung machen
in dem Fall ist man dann auch fertig und die Maschine terminiert

TM: Addition von 1 zu einer Binär-Zahl (normale Notierung)

Algorithmus für Addition von 1 auf eine Binär-Zahl im Bandspeicher

0. Schreib-Leise-Kopf steht auf dem 1 Zeichen links (normal notierte Binärzahl)
1. Lese Zeichen, wenn Zeichen eine 0 oder eine 1, dann Schreibe 0 bzw. 1 auf's Band, Verbleibe im Schritt 1 und Gehe (Verschiebe Kopf) nach Rechts (weiter Ende suchen), wenn Zeichen ein Trennzeichen, dann schreibe Trennzeichen und Gehe nach Links (Ende gefunden und Schreib-Lese-Kopf auf Ende positionieren)
2. Lese Zeichen, wenn Zeichen eine 1, dann schreibe eine 0 und weiter mit Schritt 1 (Übertrag, wenn Zeichen eine 0, dann Schreibe eine 1 und weiter bei Schritt 3 (Addition fertig); Gehe nach Links
3. Lese Zeichen, wenn Zeichen eine 0 oder 1, dann Schreibe wieder 0 bzw. 1 und Bewege Kopf nach Links und mache weiter bei Schritt 3; Wenn Zeichen ein Trennzeichen ist, dann Schreibe Trennzeichen, bewege Kopf nach Rechts und weiter mit Schritt 4
4. STOP

TM: Addition von 1 zu einer Binär-Zahl (normale Notierung)

Ausgangs-Situation:

			0	1	1	1	1															
							▲															

für diese Maschine steht der Lese-Schreib-Lopf zuanfang rechts auf dem niederwertigsten Bit

aktueller Zustand	Band-Symbol		neues Band-Symbol	neuer Zustand	Kopf-Richtung
0	0	→	1	1	L
0	1	→	0	0	L
0	☐	→	1	2	N
1	0	→	0	1	L
1	1	→	1	1	L
1	☐	→	☐	2	R
2	0	→	0	2	N
2	1	→	1	2	N
2	☐	→	☐	2	N

Ergebnis:

			1	0	0	0	0															
			▲																			

kompaktere Tabelle zu obiger Übergangs-Funktion (TURING-Programm)

	Eingaben		
Zust.	**0**	**1**	**☐**
z0	1,z0,L	0,z1,L	1,z0,N
z1	0,z1,L	1,z1,L	☐,z2,R
z2	0,z2,N	1,z2,N	☐,z2,N

(alternative) Maschine (zur Addition von Binär-Zahlen) als "Quell-Text"

```
#     Zustaende
#     0     Startzustand
#     88    Fehler
#     99    Endzustand
#
0     1     1     N     12
0     0     0     N     12
0     *     *     N     88
#
# Dekrementieren
#
1     1     1     R     1
1     0     0     R     1
1     *     *     L     2
#
2     0     1     L     2
2     1     0     N     3
2     *     *     N     99
#
3     1     1     L     3
3     1     1     L     3
3     *     *     N     4
#
```

```
4     *     *     L     4
4     1     1     N     11
4     0     0     N     11
#
# Inkrementieren
#
11    1     0     L     11
11    *     1     R     12
11    0     1     R     12
#
12    1     1     R     12
12    0     0     R     12
12    *     *     N     13
#
13    *     *     R     13
13    0     0     N     1
13    1     1     N     1
```

Q: info_15_turing.pdf (BURGARD, Wolfram; STACHNISS, Cyrill) http://ais.informatik.uni-freiberg.de (Bewegungs-Symbole angepasst!)

TM: ??? Multiplikation einer Binär-Zahl mit ???

Multiplikation einer Binärzahl

aktueller Zustand	Band-Symbol		neues Band-Symbol	neuer Zustand	Kopf-Richtung
0	0	→			
0	1	→			
0	□	→			
		→			
		→			
		→			
		→			
		→			
		→			
		→			
		→			
		→			
		→			

TM: Multiplikation einer Binär-Zahl (normale Notierung) mit 16

Multiplikation einer Binärzahl mit 16 (also * 2 * 2 * 2 * 2)

hierfür reicht es das Ende der Binärzahl rechts zu finden (höchstes Bit steht links!)
an das Ende nun 4 Nullen anhängen

aktueller Zustand	Band-Symbol		neues Band-Symbol	neuer Zustand	Kopf-Richtung
		→			
		→			
		→			
		→			
		→			
		→			
		→			
		→			
		→			

		→			
		→			
		→			
		→			

TM: Berechnung des Einer-Komplement's einer Binär-Zahl (normale Notierung)

Berechnen der Einer-Komplement's
(höchstwertige Binärstelle links)

1. ev. nach ganz links gehen
2. wenn 0 oder 1 gelesen, dann Bit invertieren (also 1 od. 0 schreiben)
3. wenn Leer-Symbol, dann STOP

aktueller Zustand	Band-Symbol		neues Band-Symbol	neuer Zustand	Kopf-Richtung
		→			
		→			
		→			
		→			
		→			
		→			
		→			
		→			
		→			
		→			
		→			
		→			

TM: Inkrementieren einer Binärzahl

Algorithmus zum Inkrementieren einer Binär-Zahl im Bandspeicher (für menschl. Berechner)

0. Anfang bei höchster Ziffer ganz links (normal notierte Binärzahl)
1. Gehe zur letzten Ziffer der Zahl und starte mit Übertrag 1
2. ist aktuell gelesenes Zeichen eine 0, dann Übertrag 1, Schreibe eine 1, Setze Übertrag auf 0 und gehe nach Links
3. ist aktuell gelesenes Zeichen eine 1, dann ist Übertrag 1, Schreibe eine 0, Setze Übertrag auf 1 und gehe nach Links
4. ist aktuell gelesenes Zeichen ein Trennzeichen, dann ist Übertrag 1, Schreibe eine 1 und FERTIG
5. ist Übertrag 0 und gelesenes Zeichen eine 0 oder 1, dann schreibe auch wieder eine 0 bzw. 1 und gehe nach Links
6. ist Übertrag 0 und ist gelesenes Zeichen ein Trennzeichen, (dann schreibe wieder ein Trennzeichen,) und FERTIG

TM: ??? Akzeptieren der Sprachen $0^n1^m0^a1^b$

akzeptiert $0^n1^m0^a1^b$
mit n, b, a ≥ 0 und m ≥ 1

aktueller Zustand	Band-Symbol		neues Band-Symbol	neuer Zustand	Kopf-Richtung
0	0	→	0	0	R
0	1	→	1	0	R
0	□	→	□	4	N

? regulärer Übergang von z0 nach z1 (hier nur Übergang zu z4 def.???!!!)

1	0	→	0	1	R
1	1	→	1	1	R
1	□	→	□	2	L
2	0	→	1	2	L
2	1	→	0	3	L
2	□	→	0	4	N
3	0	→	0	3	L
3	1	→	1	3	L
3	□	→	□	4	R
4	?	→	-	-	-

Umschreibung der Leistungen der einzelnen Zustände:

im Zustand 0 den Kopf solange nach rechts bewegen, bis eine erste 1 gefunden wird, dann wechseln zu Zustand 1; in den Final-Zustand 4 gehen, wenn ein Leerzeichen gelesen wird
im Zustand 1 solange nach rechts gehen bis eine erstes Leerzeichen gefunden wird, dann wechseln nach Zustand 2
im Zustand 2 solange nach links bewegen, bis eine erste 1 gefunden wird, alle binären Zeichen werden dabei invertiert; in den Final-Zustand 4 wechseln, wenn ein Leerzeichen gefunden wird
in Zustand 3 solange nach links laufen, bis das erste Leerzeichen gefunden wird, dann in den Final-Zustand 4 wechseln, ansonsten alle Zeichen invertieren

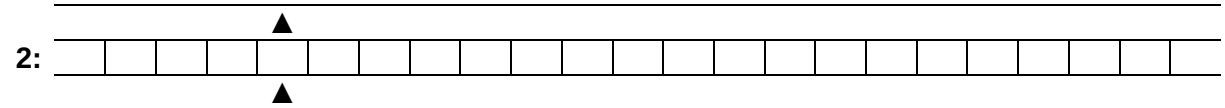
TM: ??? Akzeptieren der Sprachen $0^n1^m0^a1^b$

aktueller Zustand	**Band-Symbol**		**neues Band-Symbol**	**neuer Zustand**	**Kopf-Richtung**
		→			
		→			
	□	→			
		→			
		→			
		→			
		→			
		→			
		→			
		→			
		→			
		→			
		→			
		→			

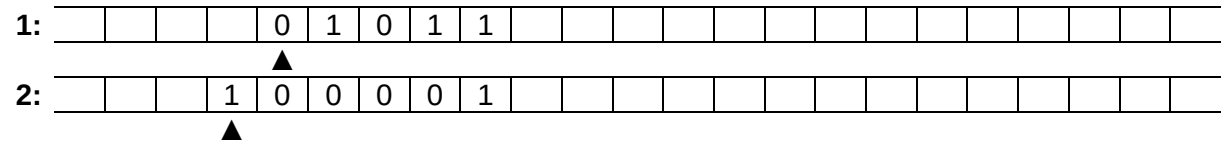
2-Band-TM: Multiplikation einer Binär-Zahl mit 3

Ausgangs-Situation:

1:					0	1	0	1	1													



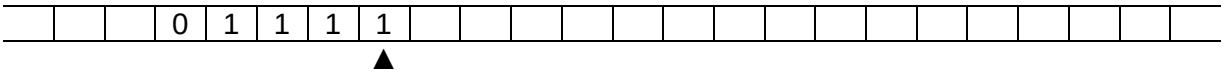
Ergebnis:



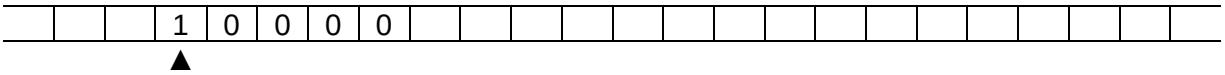
TM: ??? Akzeptieren der Sprachen $0^n1^m0^a1^b$

$\in \mathbb{N}$

Ausgangs-Situation:



Ergebnis:



	Eingaben		
Zust.	**0**	**1**	**□**
z0			
z1			
z2			

Aufgaben:

- ***x. Entwerfen Sie eine TURING-Maschine TM, die alle Worte der Sprache x^{2n} mit $n \in \mathbb{N}$ akzeptiert!***
- ***x. Erstellen Sie eine TURING-Maschine, welche die Worte der Sprache $a^{2n}b^{n}$ akzeptiert. Die Variable n beschreibt dabei alle natürlichen Zahlen.***
- ***x. Entwickeln Sie eine TM, die korrekte "Klammer"-Strukturen aus "Kleiner"- und "Größer"-Zeichen (z.B.: <<><<>>>) erkennt!***
- ***x. Entwickeln Sie eine TM, die korrekte Klammer-Strukturen (runde, eckige und geschweifte Klammern) erkennt!***

ev. nach hinten zu Berechenbarkeit verschieben od. hier nur exemplarisch

Halte-Problem für / mit Python

Biographie: Alan Mathison TURING (1912 - 1954)

Entschlüsselung Enigma mit der "Bombe"

Theoretische Informatik → TURING-Maschine

TURING-Test um künstliche Intelligenz zu bestimmen

erstes Schach-Programm "Turochamp"
da es noch keine ausreichend Leistungs-fähigen Rechner gab, berechnete TURING jeden Zug selbst, dafür brauchte er ungefähr 30 min
das einzige dokumentierte Spiel verlor er gegen einen Kollegen

arbeitete aber auch in der Bio-Informatik
beschäftigte sich mit Reaktions- Diffusions-Phänomenen, heute als TURING-Mechanismus bekannt
beschäftige sich auch mit der Struktur von Pflanzen und deren Zusammenhänge zu den FIBONACCHI-Zahlen

Q: de.wikipedia.org ()

Vor allem für theoretische Fragestellungen wurden und werden TURING-Maschinen mit bestimmten Abwandlungen, Beschränkungen usw. usf. definiert. Dabei geht es auch um die Frage, wie Leistungs-fähig die einzelnen Maschinen sind.
Insgesamt kann man sagen, dass alle veränderten TURING-Maschinen auch mit universellen TURING-Maschinen nachgebaut werden können.

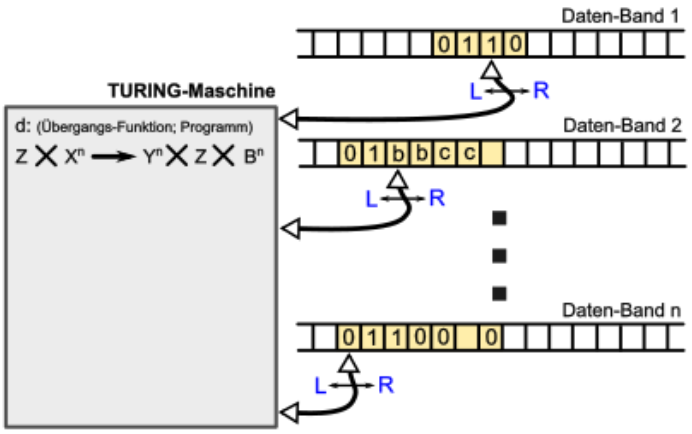
linear beschränkte TURING-Automaten

Linear beschränkte TURING-Automaten können Kontext-sensitive (also CHOMSKY-Typ-1) Sprachen erkennen.

Definition(en): linear beschränkter TURING-Automat (LBA)

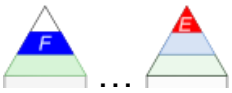
Ein linear beschränkter TURING-Automat ist eine Sonderform, bei der der Lese- und Schreib-Kopf niemals den Bereich der Eingabe verlässt.
(Das Band der TM ist also auch die Eingabe beschränkt.)

3.2.10.3. TURING-Maschinen mit mehreren Speicher-Bändern



Definition(en): Mehrband-TURING-Maschine (TA, TM)
Eine Mehrband-TURING-Maschine ist ein endlicher Automat mit mehreren unabhängigen Speichern (/ einem Speicher-Bändern), bei dem über ein Programm (in Form einer tabellarischen Übergangs-Funktion) aus den gelesenen Band-Informationen neue Ausgabe-Daten für die Speicherbänder berechnet werden. Weiterhin sind im Programm der nächste Befehl und die Bewegung auf den Speicherbändern festgelegt.
Eine Mehrband-TURING-Maschine ist ein Sept-Tupel (7-Tupel) $A = (Z, \Sigma, \Gamma, \delta, z_0, \square, Z_E)$ mit: Z … endliche Menge der Zustände; $\|Z\| < \infty$ Σ … Eingabe-Alphabet; $\|\Sigma\| < \infty$, $Z \cap \Sigma = \varnothing$ n … endliche Anzahl von Bändern; $n = 1, 2, 3, \ldots$ Γ … Band-Alphabet; $\|\Gamma\| < \infty$; $\Sigma \subseteq \Gamma$ d, δ, Δ … Überführungs-Funktion; $Z \times \Gamma^n \rightarrow Z \times \Gamma^n \times B^n$ ***(deterministische TM)*** d, δ, Δ … partielle Überführungs-Funktion; $Z \times \Gamma^n \rightarrow \mathbb{P}(Z \times \Gamma^n \times B^n)$ ***(n.-determ. TM)*** z_0 … Start-Zustand; $z_0 \in Z$ $\square$ … Leer- bzw. Trenn-Zeichen $\square \in \Sigma$ Z_E … endliche Menge der akzeptierten Zustände (End-Zustände); $Z_E \subseteq Z$ B … fest definierte Menge der Kopf-Bewegungen $\{L,N,R\}$ Einige Definitionen verzichten auf die Angabe des leeren Band-Symbol's. Dann reduziert sich die Definition auf ein Hex-Tupe (6-Tupel): $A = (Z, \Sigma, \Gamma, \delta, z_0, Z_E)$

3.2.10.4. Fleißige Biber



Für viele Fragen der Theoretischen Informatik ist die Frage nach der Effektivität und Leistungsfähigkeit einer TURING-Maschine interessant. Vom ungarischen Mathematiker Tibor RADO stammt das busy-beaver-Problem ("Fleißige-Biber-Problem"). In diesem ist eine deterministische TURING-Maschine (DTM) mit dem Eingabe-Alphabet {1,␣} gesucht, die auf ein leeres Speicher-Band maximal viele Zeichen schreiben kann.
Da die Maschine deterministisch sein soll, muss sie irgendwann halten!

Busy-Beaver der mit 2 Zuständen 4 Einsen schreibt.

Busy-Beaver Überführungs-Funktionen

z_0,␣	→	z_1,1,R
z_0,1	→	z_1,1,L
z_1,␣	→	z_0,1,L
z_1,1	→	z_e,1,R

Der gleiche Fleißige Biber – nur umgeschrieben. Die Überführungs-Funktionen hier als Zustands-Übergangs-Tabelle.

Busy-Beaver

	␣	1
z_0	z_1,1,R	z_1,1,L
z_1	z_0,1,L	z_e,1,R

Der nebenstehende Busy-Beaver schafft mit 3 Zuständen 6 Einsen zu schreiben.

Busy-Beaver

	␣	1
z_0	z_1,1,R	z_2,1,L
z_1	z_2,1,R	z_e,1,N
z_2	z_0,1,L	z_1, ␣,L

Ein wenig verändert, aber auch der nebenstehende Busy-Beaver schafft mit 3 Zuständen 6 Einsen zu schreiben.

Busy-Beaver

	␣	1
z_0	z_1,1,R	z_2,1,L
z_1	z_0,1,L	z_1,1,R
z_2	z_1,1,L	z_e, 1,N

Eine ähnliche Frage ist die nach einem Fleißigen Biber, der für eine bestimmte Anzahl von Zuständen möglichst viele Rechenschritte macht und am Schluss ein leeres Band hinterlässt.

Zustände	Anzahl Fleißige Biber	max. Anzahl Einsen	Anzahl Rechenschritte einer *Band leer*-Maschine	
1	169	1	1	
2	130'321	4	6	
3	≈2,4 * 10^{8}	6	21	
4	≈8,5 * 10^{11}	13	107	
5	≈4,8 * 10^{15}	≥4'098	≥47'176'870	
6	≈4,0 * 10^{19}	> 4,6 * 10^{1439}	>2,5 * 10^{2879}	

Daten-Q: /b, S.28/

Das wirkliche Problem ist nicht,
ob Maschinen denken,
sondern ob die Menschen es tun.
B. F. SKINNER

3.2.11. TURING-ähnliche Automaten

Wieviele Befehle / Symbole braucht man eigentlich, um einen Computer zu programmieren? Eine – mit einem gewissen Augenzwinkern versehene – Lösung bieten die Programmiersprachen Brainfuck und Ook!. Sie sind aber auch aus der Sicht der Sprachen und Automaten sehr interessant.

Aufgaben:

1. ***Recherchieren Sie zu Varianten / Abwandlungen von TURING-Maschinen! (Gemeint sind Typen, keine konkreten Maschinen (mit einem konkreten Programm)!) Erstellen Sie sich eine Tabelle, in der Sie den Namen der Variante und eine Kurz-Beschreibung notieren!***
2. ***Wählen Sie sich zwei Varianten aus und bereiten Sie eine Vorstellung der Maschinen vor dem Kurs vor! (Sprechen Sie sich im Kurs ab, damit nur wenige Dopplungen vorkommen!)***
3.

3.2.11.1. Brainfuck

Auf der Suche nach minimalen Programmier-Sprachen stößt man schnell auf Brainfuck. Diese eher esoterische Programmier-Sprache besteht aus gerade mal 8 Zeichen, die für jeweils einen Befehl stehen.
Brainfuck besteht aus den folgenden Befehlen:

Befehls-Zeichen	Semantik	C-Äquivalent			
>	inkrementiert den Zeiger	++ptr;			
<	dekrementiert den Zeiger	--ptr;			
+	inkrementiert den aktuellen Zellen-Wert	++*ptr;			
-	dekrementiert den aktuellen Zellen-Wert	--*ptr;			
.	gibt den aktuellen Zellen-Wert als ASCII-Zeichen aus	putchar(*ptr);			
,	ließt ein Zeichen ein und speichert den ASCII-Wert in die aktuelle Zelle	*ptr=getchar();			
[	springt, wenn der aktuelle Zellen-Wert 0 ist, soweit nach vorne, dass der Zeiger hinter dem zugehörigen] steht	while (*ptr) {			
]	springt, wenn der aktuelle zellen-Wert 0 ist, zu dem zugehörigen [zurück	}			

Mit minimalen Programmier-Sprachen möchte man vor allem die Grenzen der Programmierung aufzeigen, aber auch, dass man mit sehr wenigen Befehlen sehr komplexe Programme schreiben kann. Von eher akademischen Interesse ist dann auch, wie groß – oder besser wie klein - ein passendes Übersetzer-Programm (in Maschinen-Code) sein muss.
Für Brainfuck gibt es derzeit die folgenden Rekorde:

Betriebssystem / Rechner	**erstellt von**	**Größe**		
Commodore Amiga	Urban MÜLLER	240 Byte		
Linux x86	Brain MILLER	171 Byte		
ms-DOS	Bertram FELGENHAUER	96 Byte		

"Hello World!" in Braínfuck

```
++++++++++
[
>+++++++>++++++++++>+++>+<<<<-
]
>++.
>+.
+++++++.
.
+++.
>++.
<<+++++++++++++++.
>.
+++.
------.
--------.
>+.
>.
+++.
```

Datenfeld für 10 Zeichen

Festlegung von Einsprung-Punkten in die ASCII-Tabelle
Schleife für Textausgabe
Ausgabe von "H"
… "e"
… "l"
… "l"
… "o"
eines Leerzeichen's
… "W"
… "o"
… "r"
… "l"
… "d"
… "!"
Zeilenvorschub
Wagenrücklauf

Q: de.wikipedia.org

3.2.11.2. Ook!

Eine Steigerung ist die Ook-Sprache – quasi eine esoterische Programmiersprache für Orang Utans. Sie beschränkt sich auf drei Sprachelemente "Ook.". "Ook?" und "Ook!".
Das Ook wird dabei im Deutschen als "ugh" gesprochen – eben typisch orang-utanisch.
Zur Verbesserung der Übersichtlichkeit kann auf das "Ook" verzichtet werden, so dass nur noch die Symbole ".", "?" und "!" übrigbleiben.
Es werden immer zwei aufeinanderfolgend Symbole (2-Tupel) gemeinsam ausgewertet.

Befehls-Zeichen	Semantik	Brainfuck-Äquinalent	C-Äquivalent	Bemerkungen	
.?	inkrementiert den Zeiger	**+**	++ptr;	hier 2 Zellen gemeint	
?.	dekrementiert den Zeiger	**-**	--ptr;		
..	inkrementiert den aktuellen Zellen-Wert	**>**	++*ptr;		
!!	dekrementiert den aktuellen Zellen-Wert	**<**	--*ptr;		
!.	gibt den aktuellen Zellen-Wert als ASCII-Zeichen aus	**.**	putchar(*ptr);		
.!	ließt ein Zeichen ein und speichert den ASCII-Wert in die aktuelle Zelle	**,**	*ptr=getchar();		
!?	Schleifen-Anfang springt, wenn der aktuelle Zellen-Wert 0 ist, soweit nach vorne, dass der Zeiger hinter dem zugehörigen] steht	**[**	while (*ptr) {		
?!	Schleifen-Ende springt, wenn der aktuelle zellen-Wert 0 ist, zu dem zugehörigen [zurück	**]**	}		

Ook! wurde von David MORGAN-MAR entwickelt. Die Sprache ist TURING-vollständig.
Im Endeffekt beweist Ook! die Aussage des Fantasy-Schriftsteller's Terence David John PRATCHETT (1948 - 2015), dass man alles (Worte, Sätze, …) mit (einfachsten) Affen-Geräuschen ausdrücken könne.

"Hello World!" in Ook

```
.?.................!??........
.........?!!?!?.!..?............
..!??..........?!!?!?...!.......
........!.!.......!..?.?.?......
..........!??..........?!!?!?.!.
.?.?.?.....................!??...
..................?!!?!?.!!!!!!!.
?.?.?.?.!.......!.!!!!!!!!!!!!!.
!!!!!!!!!!!!!!!!..?.?..!.
```

Q: gc.de/gc

Um mal deutlich zu machen, wie groß / klein ein Übersetzungs-Programm sein kann / muss, hier ein Interpreter für Ook! von Øyvind GRØNNESBY

```
#!/usr/bin/env python
#
# an Ook! interpreter written in python
#
# you can wrap the memory pointer to the end of the the memory cells
# but you cannot do the same trick to get the first cell, since going
# further out would just initiate a new memory cell.
#
# 2001 (C) Øyvind Grønnesby <oyving@pvv.ntnu.no>
#
# 2003-02-06: Thanks to John Farrell for spotting a bug!

import sys, string, types

def massage(text):
    ret = []
    tok = []

    for line in text:
        if line[0] != ";" and line != "\n" and line != "":
            for token in line.split(" "):
                if token != "":
                    ret.append(token.strip())
    return ret

def sane(code):
    if len(code) % 2 == 0:
        return 1
    else:
        return 0

class OokInterpreter:

    memory = [0]
    memptr = 0
    file   = None
    code   = None
    len    = 0
    codei  = 0

    def __langinit(self):
        self.lang   = {'Ook. Ook?' : self.mvptrup,
                       'Ook? Ook.' : self.mvptrdn,
                       'Ook. Ook.' : self.incptr,
                       'Ook! Ook!' : self.decptr,
                       'Ook. Ook!' : self.readc,
                       'Ook! Ook.' : self.prntc,
                       'Ook! Ook?' : self.startp,
                       'Ook? Ook!' : self.endp}

    def mem(self):
        return self.memory[self.memptr]

    def __init__(self, file):
        self.__langinit()
        self.file = open(file)
        self.code = massage(self.file.readlines())
        self.file.close()
        if not sane(self.code):
            print self.code
```

```
            raise "OokSyntaxError", "Code not sane."
        else:
            self.cmds()

    def run(self):
        self.codei = 0
        self.len  = len(self.code)
        while self.codei < self.len:
            self.lang[self.code[self.codei]]()
            self.codei += 1

    def cmds(self):
        i = 0
        l = len(self.code)
        new = []
        while i < l:
            new.append(string.join((self.code[i], self.code[i+1]), " "))
            i += 2
        self.code = new

    def startp(self):
        ook = 0
        i   = self.codei
        if self.memory[self.memptr] != 0:
            return None
        while 1:
            i += 1
            if self.code[i] == 'Ook! Ook?':
                ook += 1
            if self.code[i] == 'Ook? Ook!':
                if ook == 0:
                    self.codei = i
                    break
                else:
                    ook -= 1
            if i >= self.len:
                raise 'OokSyntaxError', 'Unmatched "Ook! Ook?".'

    def endp(self):
        ook = 0
        i   = self.codei
        if self.memory[self.memptr] == 0:
            return None
        if i == 0:
            raise 'OokSyntaxError', 'Unmatched "Ook? Ook!".'
        while 1:
            i -= 1
            if self.code[i] == 'Ook? Ook!':
                ook += 1
            if self.code[i] == 'Ook! Ook?':
                if ook == 0:
                    self.codei = i
                    break
                else:
                    ook -= 1
            if i <= 0:
                raise 'OokSyntaxError', 'Unmatched "Ook? Ook!".'

    def incptr(self):
        self.memory[self.memptr] += 1

    def decptr(self):
        self.memory[self.memptr] -= 1
```

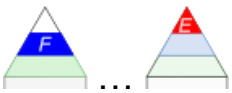
```
    def mvptrup(self):
        self.memptr += 1
        if len(self.memory) <= self.memptr:
            self.memory.append(0)

    def mvptrdn(self):
        if self.memptr == 0:
            self.memptr = len(self.memory) - 1
        else:
            self.memptr -= 1

    def readc(self):
        self.memory[self.memptr] = ord(sys.stdin.read(1))

    def prntc(self):
        sys.stdout.write(chr(self.mem()))


if __name__ == '__main__':
    o = OokInterpreter(sys.argv[1])
  o.run()
```

Q: https://github.com/oyving/pook/blob/master/pook.py

Links:

https://gc.de/gc (u.a. Brainfuck- und Ook!-online-Interpreter)

3.2.12. Register-Maschinen



vielfach auch Register-Automaten genannt, kurz RM

kann man schon als Primitiv-Version eines heutigen Computers verstehen
praktisch ist der Mikroprozessor eine erweiterte Realisierung einer Register-Maschine

erste Vorstellungen (1959) stammten von Heinz KAPHENGST (1932 -)
als Weiter-Entwicklung der TURING-Maschine verstanden

nannte die Register noch Fächer (Z_∞), Befehls-Zähler als besonderes Register (Z_0) interpretiert
unabhängig voneinander stellten dann John C. SHEPERDSON () und Howard E. STURGIS (1936 -) im Jahr 1963 eine sehr umfassende theoretische Betrachtung zu den RM's vor

Register-Maschinen bestehen aus:

- einem Eingabe-Wert x
- einem Programm P (aus durchnummerierten Befehlen, beginnend bei 1)
- einem Befehls-Zähler n
- einem (unendlich großen) Satz an (durchnummerierten) Registern R[i] (/ Speicherzellen)

und

- einem Ausgabe-Wert y

Die reinen technischen Definitionen – also der zur Verfügung stehende Satz an Arbeits-Befehlen – unterscheiden sich von Autor zu Autor.
Dabei kommen minimale Befehls-Sätze mit nur 3 Arbeits-Befehlen bis hin zu sehr umfangreichen und leistungsfähigen Befehls-Sätzen zum Einsatz.

eine mögliche Form der Weiter-Entwicklung der RM ist die RAM. RAM steht dabei für Random Access Machine und beschreibt eine Maschine, die ihre Register auch indirekt adressieren kann. Die wortwörtliche Übersetzung einer Beliebig-Zugriffs-Maschine ist sicher nicht Ziel-führend. Eine Benennung als Index-Register-Maschine trifft den Kern wohl besser.

Index-Register-Maschinen bestehen aus:

- **einem Eingabe-Wert x**
- einem Programm P (aus durchnummerierten Befehlen, beginnend bei 1)
- einem Befehls-Zähler n
- einem Akkumulator R_A (Register A, Haupt-Register)
- einem (unendlich großen) Satz an (durchnummerierten) Registern R[i] (/ Speicherzellen)

und

- **einem Ausgabe-Wert y**

Definition(en): Register-Maschine (Register-Automat)

möglicher Befehls-Satz einer RAM

Befehl	Wirkung beim Akkumulator A	Wirkung beim Befehlszähler n	Beschreibung / Bemerkungen
Register-Befehle / Speicher-Zugriff			
LOAD i	A ::= R[i]	n ::= inc(n)	Laden des Wertes aus dem Register mit dem Index i in den Akkumulator
CLOAD i	A ::= i	n ::= inc(n)	Laden eines Wertes in den Akkumulator
INDLOAD i	A ::= R[R[i]]	n ::= inc(n)	Laden des Wertes aus dem Register, dessen Nummer im Register mit dem Index i steht (in den Akkumulator)
STORE i	R[i] ::= A	n ::= inc(n)	Speichern des Akkumulator-Wertes in das Register mit dem Index i
INDSTORE i	R[R[i]] ::= A	n ::= inc(n)	Speichern des Akkumulator-Wertes in das Register, dessen Nummer in dem Register mit dem Index i steht
Verarbeitungs-Befehle / Rechen-Befehle			
ADD i	A ::= A + R[i]	n ::= inc(n)	Addieren des Wertes aus dem Register mit dem Index i zum Akkumulator
CADD i	A ::= A + i	n ::= inc(n)	Addieren des Wertes i zum Akkumulator
INDADD i	A ::= A + R[R[i]]	n ::= inc(n)	Addieren des Wertes aus dem Reg., dessen Nummer in dem Reg. mit dem Index i steht, zum Akkum.
SUB i	A ::= max(A - R[i], 0)	n ::= inc(n)	natürlichzahliges Subtrahieren (sonst Prinzip, wie bei Addition)
CSUB i	A ::= max(A - i, 0)	n ::= inc(n)	
INDSUB i	A ::= max(A – R[R[i]], 0)	n ::= inc(n)	
MUL i	A ::= A * R[i]	n ::= inc(n)	Multiplizieren (im Prinzip, wie bei Addition)
CMUL i	A ::= A * i	n ::= inc(n)	
INDMUL i	A ::= A * R[R[i]]	n ::= inc(n)	
DIV i	A ::= A / R[i]	n ::= inc(n)	Dividieren (im Prinzip, wie bei Addition)
CDIV i	A ::= A / i	n ::= inc(n)	
INDDIV i	A ::= A / R[R[i]]	n ::= inc(n)	
Steuer-Befehle			
GOTO i	**A ::= A**	n ::= i	
IF A **O** x GOTO i	**A ::= A**	wenn wahr dann n ::= i sonst n :: inc(n)	bedingter Sprung zum Befehl i; **O** ist ein Vergleich (<, >, =, ≠)
END	**A ::= A**	n ::= n	Programm beenden

3.2.12.x. der Know-How-Papier-Computer

1983 von Wolfgang BACK und Ulrich ROHDE für den WDR Computerclub entwickelt
damals kosteten Heimcomputer noch rund 4'000 DM (2'000 €) waren so Leistungs-fähig, wie heutige Taschenrechner

besteht aus Programm-Speicher, einem Programmzähler und 8 Registern
Programm-Speicher sind beginnend bei 0 unendlich viele Speicher-Zellen als Papierstreifen realisiert, jede Zelle hat eine fortlaufende Nummer (Adresse)
der Programmzähler wird durch einen Kugelschreiber dargestellt, der immer auf genau eine Speicher-Zelle des Programm-Speicher zeigt
die Register sind die Datenspeicher (einen RAM gibt es nicht!)
der Daten-Inhalt in den Registern wird durch Streichhölzer angezeigt

die Register sind zu Beginn willkürlich mit Daten (Streichhölzern) belegt

die Assembler-Sprache des Papier-Computers besteht aus 5 Befehlen

Befehl	Operanden-Typ	Beschreibung / Funktion	Beispiel
inc	Register	increment / Inkrement erhöht den Wert des angegebenen Registers um 1	
dec	Register	decrement / Dekrement verringert den Wert des angegebenen Registers um 1	
isz	Register	is_zero / Ist_null vergleicht den Inhalt des angegebenen Registers mit 0; ist er 0, dann wird der Programmzähler um 2 erhöht; wen nicht, dann nur um 1 (bei Null wird praktisch der nächste Befehl übersprungen)	
jmp	Adresse	jump_to / Sprung_nach	
stp	--	stop / Halt hält die Abarbeitung an	

weiterhin sind zusätzliche Assembler-Elemente zugelassen, die ein verbesserten Umgang mit dem Quelltext ermöglichen

Element	Beschreibung / Funktion	Beispiel
Leerzeile	Leerzeilen dienen nur der Strukturierung des Assembler-Textes (sie gehen nicht in den Programm-Code ein!)	
; ***Kommentar***	zusätzliche Informationen / Beschreibungen zum Code (diese Informationen gehen nicht in den Programm-Code ein!)	
***Label*:**	definiert eine symbolische Sprungadresse, die im umgesetzten Code einer Sprung-Adresse entspricht (dient vor allem dem freien Verschieben von Sprungadressen während der Entwicklung des Programm's)	

Programm-Beispiele für den Know-How-Papier-Computer (in Assembler)

Addition von Register A und B in Register C

```
; Im ersten Schritt wird das Zielregister C ausgenullt.
clearC:
isz C
jmp decC
jmp clearX
decC:
dec C
jmp clearC

; Dann wird X genullt.
clearX:
isz X
jmp decX
jmp processA
decX:
dec X
jmp clearX

; Wir kopieren den Wert aus A nach X und C
processA:
isz A
jmp inc
jmp restoreA
inc:
dec A
inc X
inc C
jmp processA

; Jetzt steht A in X und C, aber A=0
; Wir restaurieren den Wert in A aus X.
restoreA:
isz X
jmp incA
jmp processB
incA:
inc A
dec X
jmp restoreA

; Jetzt kopieren wir den Wert aus B nach X und C
; X ist 0, wird also wieder eine Sicherheitskopie.
; Und in C steht nachher die Summe.
processB:
isz B
jmp inc2
jmp restoreB
inc2:
dec B
inc X
inc C
jmp processB

; Wir restaurieren den Wert in B aus X.
restoreB:
isz X
jmp incB
jmp done
incB:
inc B
dec X
jmp restoreB

; Fertig.
done: stp;
```

Q: https://marian-aldenhoevel.de/papiercomputer/ (leicht geändert: dre)

Register A auf Null setzen

```
; Dieses Programm nullt das Register A

whileA: isz A ; Ist A=0?
jmp decA      ; Nein, weiter bei decA
jmp done      ; Ja, Sprung zum Ende
decA: dec A   ; A um eins verringern
jmp whileA    ; Sprung zum Anfang, wieder testen.

done:
stp           ; Fertig.
```

Q: https://marian-aldenhoevel.de/papiercomputer/ (leicht geändert: dre)

Kopieren von Register A nach Reg. B

Q: https://marian-aldenhoevel.de/papiercomputer/ (leicht geändert: dre)

Subtraktion von Register B vom Reg. A und Ablage des Ergebnisses in Reg. C

Q: https://marian-aldenhoevel.de/papiercomputer/ (leicht geändert: dre)

Multiplikation von Register A und Reg. B und Ablage des Ergebnisses in Reg. C

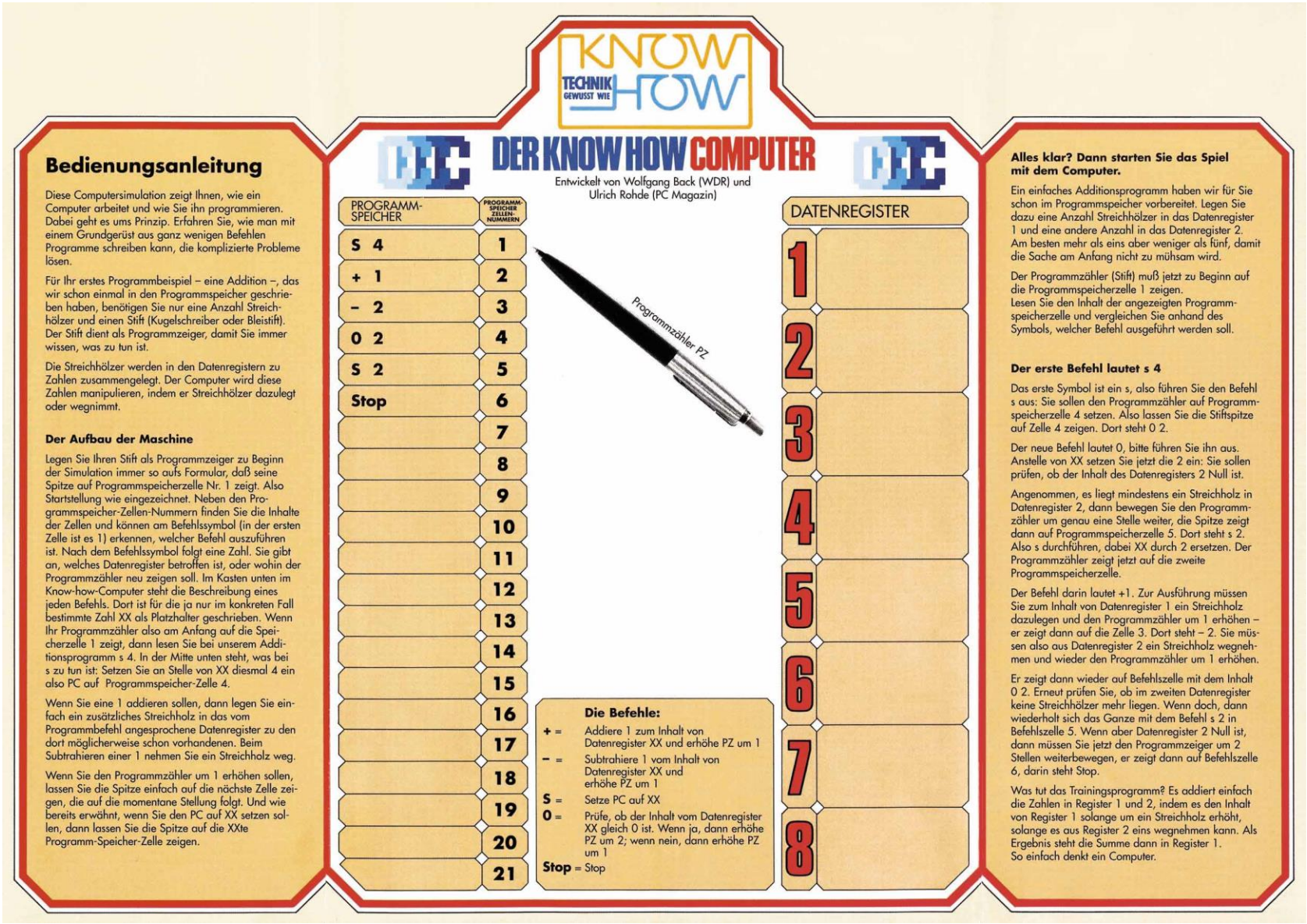
Q: https://marian-aldenhoevel.de/papiercomputer/ (leicht geändert: dre)

Division von Register A durch Reg. B und Ablage des Ergebnisses in Reg. C

```
; Dieses Programm berechnet die Division.
; Register C erhält den ganzzahligen Quotienten C = [A/B] und Register
; den Rest D = A mod B.
; Es verwendet die Register X, Y und Z für temporäre Daten und
; überschreibt die Daten darin.
; Ist B = 0 wird C = 0 und D = 0 ausgegeben.

; 0 => C
clearC:
isz C
jmp decC
jmp clearD
decC:
dec C
jmp clearC

; 0 => D
clearD:
isz D
jmp decD
jmp checkB
decD:
dec D
jmp clearD
checkB:
; Falls B=0 ist sind wir hier schon fertig.
isz B
jmp clearX
jmp done;

; 0 => X
```

```
clearX:
isz X
jmp decX
jmp loopCopyAX
decX:
dec X
jmp clearX
; A => X
loopCopyAX:
isz A
jmp doCopyAX
jmp clearY
doCopyAX:dec Ainc Xjmp loopCopyAX; 0 => YclearY:isz Yjmp decYjmp loopCopy-
BYZdecY:dec Yjmp clearY; B => YloopCopyBYZ:isz Bjmp doCopyBYZjmp loopDiv-
doCopyBYZ:inc Ydec Bjmp loopCopyBYZ; Jetzt haben wir C = 0 und so vorbe-
reitet für den Quotienten; Das Ergebnisregister D für den Rest ist auch 0.
A ist nach X; umgefüllt und B ist auch 0, sein Wert wurde nach Y umge-
füllt.; Y werden wir für die wiederholte Subtraktion verwenden.; Hier
steigen wir wieder ein, so lange X noch Streichhölzer hat,; also der Divi-
dend nicht ganz verbraucht ist.loopDiv:; Wir legen Streichhölzer aus Z
nach B und gleichzeitig aus X; zurück nach A.; Damit subtrahieren wir den
Quotienten einmal vom Ergebnis.; Falls uns die Streichhölzer in X reichen
erhöhen wir den Quotienten; einmal und legen sie für die nächste Iteration
aus B zurück nach; Z.; Gehen unterwegs die Streichhölzer in X aus hat B
jetzt den Rest. Wir; legen diese zurück nach Z und D damit am Ende der
Rest in D steht.isz Xjmp sub1jmp doneDivsub1:; Müssen wir noch Divisor ab-
ziehen?isz Yjmp doSub1jmp doneSub1doSub1:; Haben wir noch Dividend übrig
von dem wir abziehen können?isz Xjmp havejmp haveNothave:; Ein Streichholz
aus X nach Adec Xinc A; Ein Streichholz aus Y nach Bdec Yinc Bjmp sub1;
Einmal erfolgreich den ganzen Divisor abgezogen.doneSub1:; Quotient erhö-
heninc C; B zurück nach Y für die nächste Runde.loopRestoreY:isz Bjmp do-
RestoreYjmp loopDivdoRestoreY:dec Binc Yjmp loopRestoreYhaveNot:; Der Di-
vidend ist uns ausgegangen. Der Rest ist jetzt in B.; Wir füllen ihn von
dort ins Ergebnis nach D um und gleichzeitig
; zurück nach Y.
loopRestoreYD:
isz B
jmp doRestoreYD
jmp doneDiv
doRestoreYD:
dec B
inc Y
inc D
jmp loopRestoreYD

doneDiv:
; Wir sind fast fertig: Der ganze Dividend ist zurück nach A gewandert.
; Der Quotient ist im Ergebnisregister C, und der Rest in D. Es fehlt nur
; noch den Divisor von Y zurück nach B zu schieben.
loopRestoreB:
isz Y
jmp doRestoreB
jmp donedoRestoreB:
inc B
dec Y
jmp loopRestoreB

; Fertig.
done: stp;
```

Q: https://marian-aldenhoevel.de/papiercomputer/ (leicht geändert: dre)



Q: https://marian-aldenhoevel.de/papiercomputer/

kann alles berechnen, was ein Computer überhaupt berechnen kann
ist TURING-vollständig

Sprache	… Sprachen	erzeugende Grammatik		nicht-deterministi-scher Automat		deterministischer Automat (Akzeptor)	Sprach-Beispiel(e)	Vereinigung	Schnitt	Komplement	
endl. Mengen	- - -	- - -					{a,ab,abb}	+	+	-	
Typ 3	reguläre (REG)	reguläre G. rechts-lineare G.	=	(NFA)	=	endlicher Automat (DFA)	{a}*{b}*	+	+	+	
DKF	deterministisch kontextfrei	LR(k)-G.n k ≥ 1	=			nicht-deterministischer Keller-A.	$\{a^n b^n \mid n \geq 0\}$	-	-	+	
Typ 2	kontextfreie (CF)	kontextfreie G.	=	(PDA)		Keller-Automat (DPDA)	$a^n b^n$; $n \in \mathbb{N}$ od. $\{a^n b^n \mid n \geq 0\}$ $\{wcw^{rev}, w \in \{a,b\}^*\}$ Palindrome reguläre Klammer-Folgen	+	-	-	
Typ 1	kontextsensitive (CS)	kontextsensitive G. monotone G.	=	linear gebundene TURING-Maschine (LBA)	⊇	linear beschränkter Automat (DLBA)	$a^n b^n c^n$; $n \in \mathbb{N}$ od. $\{a^n b^n c^n ; n \geq 0\}$	+	+	+	
	- - -	- - -					L_e	+	+	+	
Typ 0	rekursiv auf-zählbare (RE)	Typ-0-G.	=	nicht-det. TURING-Maschine (NTM)	=	det. TURING-Automat (DTM)	H, $(G^* \setminus L_d)$	+	+	-	
abzählb. Mengen	- - -	- - -					L_d, $\mathbb{N}$				

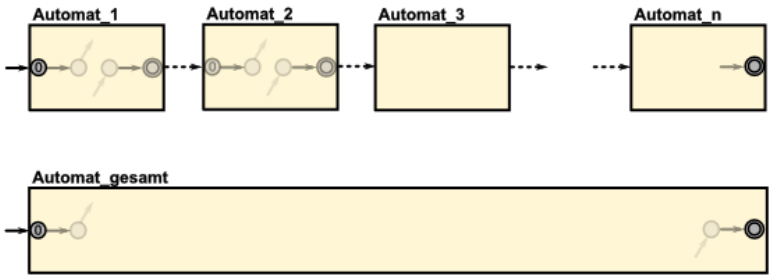
Sprache	Komple-ment	Durch-schnitt	Vereini-gung	Konkate-nation	KLEENE-Stern	
Typ 3	✓	✓	✓	✓	✓	
DKF	✓	–	–	–	–	
Typ 2	–	–	✓	✓	✓	
Typ 1	✓	✓	✓	✓	✓	
Typ 0	–	✓	✓	✓	✓	

Entscheidbarkeits-Resultate

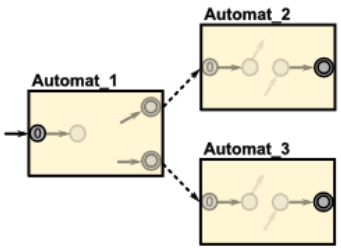
Sprache	Wort-Problem	Leerheits-Problem	Endlichkeits-Problem	Äquivalenz-Problem	Komplexität des Wort-Problems	
Typ 3	✓	✓	✓	✓	$\Theta(n)$	
DKF	✓	✓	✓	?		
Typ 2	✓	✓	✓	–	$O(n^3)$	
Typ 1	✓	–	–	–	$O(a^n)$ (expotentiell)	
Typ 0	–	–	–	–	halbentscheidbar	

3.2.13. Kombination von Automaten

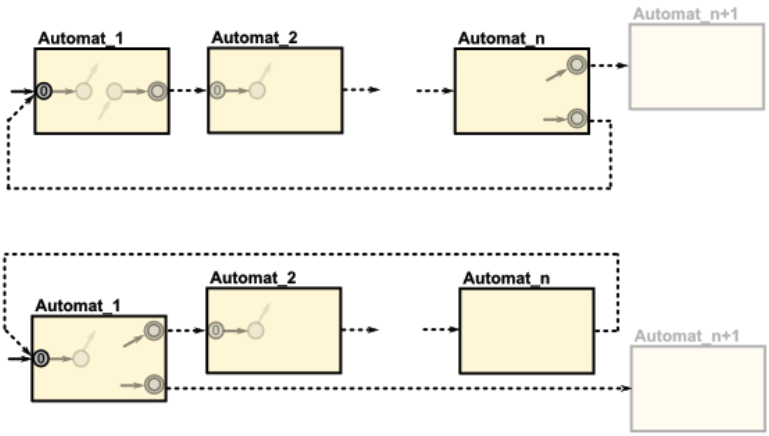
Die Nutzung der meisten von besprochenen Automaten ist auf Grund ihrer Primitivität vor allem auf die Theorie beschränkt. In der Technik gibt es sie natürlich auch, aber dort wohl nur als Bausteine für größere Systeme (Automaten).
Automaten lassen sich zu größeren Einheiten kombinieren. Prinzipiell haben wir die Möglichkeit viele Automaten parallel anzuordnen und arbeiten zu lassen. Als theoretisch-praktische Nutzung kommen da die Zellulären Automaten und Künstliche Neuronale Netze in Frage. In der Praxis kennen wir sie von Verarbeitungs-Einheiten auf Graphik-Karten. Sie berechnen tausende von Punkten (Pixel) für bestimmte Spiel-Szenen immer wieder neu, oder sie legen Bild-Muster auf bestimmte Flächen, damit der Eindruck einer bestimmten Oberflächen-Struktur entsteht.
Die andere Art der Anordnung von Automaten sind Aneinanderreihungen. Dabei ist der End-zustand des einen der Start-Zustand des nächsten Automaten.
Die einfachste Form der Aneinanderreihung ist die Sequenz. Zwei oder mehrere Automaten werden hintereinander abgearbeitet. Jeder einzelne Automat ist für eine bestimmte Aufgabe zuständig. Ist diese erfüllt, kann der nächste Automat mit seiner Aufgabe folgen.



Mit Automaten, die mindestens zwei End-Zustände oder eben sowas wie definierte Quasi-End-Zustände (z.B. Fehler-Zustände) besitzen, lassen sich auch Verzeigungen konstruieren.
Der eine Automat (z.B. Automat_2) kann dann als WAHR- oder THEN-Zweig verstanden werden, während der andere den FALSCH- bzw. ELSE-Zweig repräsentiert.
Hat der verteilende Automat noch mehr – von außen – nutzbare Zustände, dann sind auch Mehrfach-Verteilungen (CASE-Systeme) realisierbar.



Schleifen aus Automaten benötigen entweder am Anfang oder am Ende eine Verzeigung. Die sachliche Betonung liegt aber auf dem wiederholten Durchlauf durch eine Sequenz.
Jeder der Strukturen kann wiederrum gleich- oder anders-artige Kombinationen enthalten.

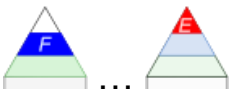


3.2.14. Projekt: Einfache Turtle-Graphik für Compiler-Bau-Anfänger

EBNF für eine einfache Turtle-Graphik:

```
<AnweisungsFolge>     ::= { <Anweisung> ";"}
<Anweisung>           ::= <Funktion> | <KontrollAnweisung>
<Funktion>            ::= <FunktionsName> <Wert>
<FunktionsName>       ::= "VOR" | "ZURUECK" | "HOCH" | "RUNTER" | "DREH"
                          | "AUF" | "AB"
<KontrollAnweisung>   ::= "MACHE" <Wert> "(" <AnweisungsFolge> ")"

<Wert>                ::= → <Zahl>
```

3.2.15. gekoppelte Automaten



Netze von Automaten – wie die vorne kurz betrachteten PETRI-Netze oder die Künstlichen Neuronalen Netze – können als Kopplungen von – z.T. verschiedenen – Automaten betrachtet werden. Das geht uns hier aber zuweit. Wir bleiben bei unseren einfachen – abstrakten – Automaten. I.A. werden bei einfachen Betrachtungen gekoppelter Automaten auch immer gleichartige Automaten miteinander verbunden.

Die Kopplung der Automaten ist praktisch als Parallelisierung gemeint. Viele gleichartige Automaten arbeiten gleichzeitig und gemeinsam. I.A. wird diese Parallelisierung bei Simulationen mangels so vieler gleichzeitig verfügbarer Maschinen durch eine sequentielle Abarbeitung umgangangen.

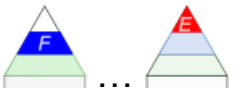
Wir kommen hier im Bereich der parallelen Informations-Verarbeitung an. Vor allem für Probleme, die von Einzel-Maschinen nur mit sehr großem Aufwand lösen können, ist das parallele Arbeiten ein mögliches Lösungs-Konzept.
Wenn z.B. ein einzelner Computer 1'000 Jahre für ein Problem bräuchte, dann sind 1'000 Computer theoretisch in der Lage, das Problem in einem Jahr zu lösen. Hat man dann 1'000'000 Computer, dann ist das Problem in ungefähr 8 Stunden gelöst.
Natürlich stellt sich die Frage ist der Aufwand sinnvoll? Für bestimmte Aufgaben schon (Dechiffrieren feindlicher Nachrichten). Nachteilig ist auch, dass wir heute Computer mit viel Ballast beschäftigen. Meist würden viel einfachere Computer ausreichen.
In unseren modernen Computern ist die Parallelisierung längst angekommen. Für Spiele-Szenen oder andere komplexe Bilder bzw. deren Verarbeitung müssen Millionen / Milliarden von einzelnen Pixeln in kürzester Zeit berechnet werden. Schließlich will keiner ruckelige Bilder. Die Graphikkarten von heute enthalten Prozessoren, die schon viele Operationen parallel abarbeiten können. Sie können somit als technische Realisierung gekoppelter Automaten verstanden werden.

In vielen anderen Wissenschaften interessieren sich die Forscher sehr stark für gekoppelte – z.T. sehr primitive – Systeme. Sie wollen z.B. die Schwarm-Intelligenz von Ameisen oder Bienen verstehen. Aber auch der Mensch unterliegt vielen Gruppen-Dynamiken, die man gerne verstehen und kontrollieren möchte.

Schon bei der Kopplung nur weniger Automaten stellt man fest, dass ein solches gekoppeltes System auf einmal viel mehr kann und leistet, als die Summe seiner (primitiven) Einzel-Automaten.

Erste Ideen zu gekoppelten Automaten stammen auch vom großen Informatiker John VON NEUMANN.

3.2.15.1. Zelluläre Automaten

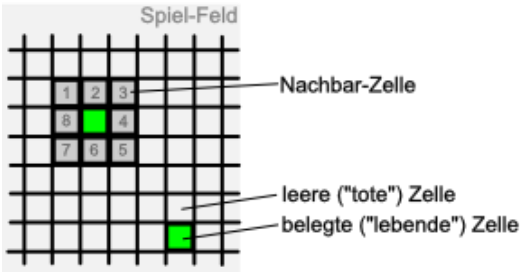


hier gemeint 2D Automaten
auch: cellular automata (CA), Zellular-Automaten, Poly-Automaten
Automaten-Theorie von Stanslaw Marcin ULAM ()

Nach Alvin Ray SMITH (1971) kann jede TURING-Maschine durch einen Zellulären Automaten ersetzt werden. Somit sind Zelluläre Automaten auch zur Lösung hoch-komplexer Probleme in der Lage.

der berühmteste Zellulärer Automat ist das Spiel "Life". Wobei der Begriff "Spiel" hier auch sehr weit und eher wissenschaftlich gefasst wird.

Alle zweidimensionalen Spiele oder Automaten basieren auf einem Zellen-Raster. In den klassischen Formen sind das Quadrate, die immer wieder von anderen Quadraten umgeben sind. Viele Spiel-Felder sind unendlich, d.h. wenn oben in der Darstellung Schluss ist, setzt sich das Spiel-Feld am unteren Rand des Spiel-Feldes fort. Man könnte sich das Spiel-Feld als Rolle in waagerechter Richtung vorstellen. Für die Sekrechte gilt das gleiche Prinzip.



Letztendlich ensteht quasi eine kugelige Welt. Wegen der etwas komplizierten Berechnung – vor allem an den Ecken des Spiel-Feldes – verwendet man auch häufig begrenzte Spiel-Felder quasi mit Rand.
Je Zelle – also ein einzelnes Quadrat – hat zu einem Zeitpunkt (besser Takt) immer einen bestimmten Zustand. Zum nächsten Takt hin werden nach bestimmten Übergangs-Regeln, die i.A. für alle Zellen gleich sind – der nächste Zustand abgeleitet. Als Eingaben fungieren die umliegenden Zellen. Jedes Quadrat hat insgesamt 8 davon.
Man nennt dies auch die MOORE-Nachbarschaft oder in Bezugnahme auf das Schach-Spiel Dame-Nachbarschaft. Vier Zellen haben nur über die Ecken Kontakt, die anderen vier liegen an den Kanten. Je nach Regel-System werden die Nachbarzellen gleichartig bewertet oder eben zwischen den diagonalen und lateralen (direkt angrenzenden) unterschieden. Werden nur die lateralen Nachbarn betrachtet heißt dies VON-NEUMANN-Nachbarschaft oder als Analogon zum Schach Turm-Nachbarschaft.
Jede Zelle kann als MEALY-Maschine betrachtet werden. (Da sich Maschine hier irgendwie komisch anhört, benutze ich im Weiteren den Begriff Automat. Das hat irgendwie was – zumindestens gefühlt – stationäres.)

Betrachten wir als einfaches Beispiel das Spiel "Life". Es wurde von John Horton CONWAY 1970 als "Spiel des Lebens" publiziert
Eine Zelle ist entweder "lebend" (gesetzt) oder "tod" (nicht gesetzt, leer).
Im Spiel gelten die folgenden Regeln:

1. eine tote Zelle mit 3 Nachbarn wird zur nächsten Generation hin neu geboren (Vermehrung)
2. lebende Zellen mit weniger als 2 lebenden Nachbarn sterben zur nächsten Generation hin (Einsamkeit)
3. lebende Zellen mit mehr als 3 lebenden Nachbarn sterben zur nächsten Generation hin (Konkurrenz, Überbevölkerung)
4. in anderen Situationen bleibt der Zustand unverändert

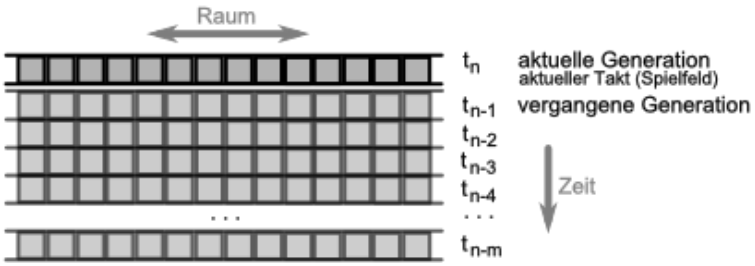
Beim Lesen der Regeln wird wohl schnell klar, warum das auch "Spiel des Lebens" heißt.
Heute werden Varianten (z.T. mehrdimensional) als sehr einfache und effektive Modelle für naturwissenschaftliche und wirtschaftlich-ökonomische Forschungen benutzt.
Besonders interessant ist hier aus biologischer Sicht, dass sich die einzelnen Zellen ziemlich typisch biologisch verhalten. Betrachtet man aber z.B. zufällige Muster ausreichender Dichte, dann treten auf einmal komplexe Objekte auf. Und was noch überraschender ist, diese Objekte bekommen völlig neuartige Eigenschaften. Einfache bleiben beständig und unbeweglich. Andere ossizilieren nur und wieder andere wandern durch die Spielfläche. Es gibt Objekte, die zeigen so etwas wie Generations-Wechsel. Sogar agressive "Kanonen" sind bekannt geworden. Und das alles nur durch Kombination von Zellen in bestimmer Anordnung. Alle Zellen agieren für sich nach den Regeln des Spiels, zusammen ergeben sie übergordnete Systeme.
Lange Zeit war die Suche nach besonderen Figuren mit solchen extravaganten Merkmale ein Massenphänomen. Heute ist etwas ruhiger um die zellulären Automaten geworden. Im Zuge der Forschung um Kooperation, Schwarm-Intelligenzen und Netzen werden die zellulären Automaten – in abgewandelter und spezialisierter Form – eine wieder grössere Bedeutung.
Zum Ausprobieren gibt es verschiedene fertige Programme oder Quell-Texte für die unterschiedlichsten Programmier-Sprachen. Wählen Sie einfach die Sprache, die Sie können, dann macht auch die Analyse des Quell-Textes Spass.
Suchen Sie auch mal nach "SmoothLife", wenn das nicht schon "Leben" ist, was dann (;-).
In fortgschrittenen Modell werden auch mehrere unterschiedliche Populationen betrachtet, die miteinander interagieren. So wird in "Wator" eine Räuber-Beute-Beziehung simuliert.
Heute experimentiert man mit den verschiedensten Regeln. Um in der Vielfalt die Übersicht zu behalten hat man einen eigenen Regel-Code entwickelt. Dazu wird eine durch einen Schrägstrich getrennte Zahlen-Kombination verwendet. Die erste Zahl(en-Reihe) gibt an, bei welcher Anzahl Nachbarn eine Zelle überlebt. Als zweite Zahl wird die Anzahl der notwendigen nachbarn für eine Neu-Geburt angegeben. Das Spiel Life nach CONWAY erhält dann also den Code 23/3 (sprich: zwei drei Strich drei). Mit solchen Code's ist eine internationale Kommunikation vereinfacht, da man nicht die landessprachliche Umschreibung der Regeln verstehen muss.

Kriterien für zelluläre Automaten (nach GERHARDT + SCHUSTER)

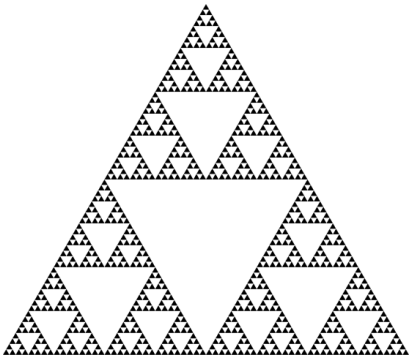
- Entwicklungen finden in Raum und Zeit statt
- der Raum ist eine diskrete Menge von Zellen
- jede Zelle kann eine begrenzte Anzahl von Zuständen einnehmen
- alle Zellen sind identisch und verhalten sich nach den gleichen Regeln (homogener zellulärer Automat)
- das Verhalten der Zellen hängt nur von ihrem aktuellen Zustand und den Zuständen der Nachbar-Zellen ab

Praktisch kann man die zelluären Automaten auch schon als Zeit-getakte Künstliche Neuronale Netze (→) auffassen. Damit schliesst sich der große Kreis der Automaten.

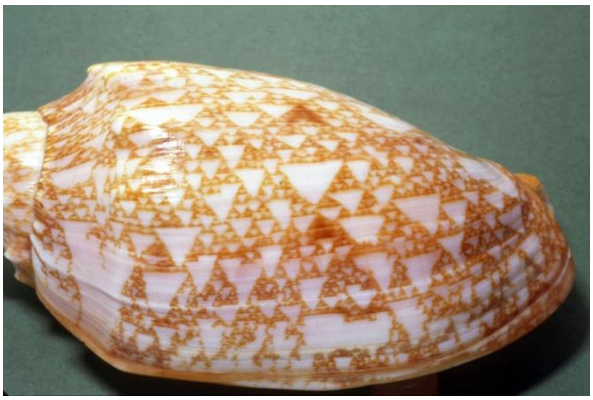
Es gibt Unmengen von Abwandlungen und geänderten System-Bedingungen. Recht bekannt sind auch eindimensionale Automaten. Ihre Welt besteht nur aus einem Band von Zellen. Meist ist das Band auf beiden Seiten miteinander verbunden – so dass ein Ring entsteht.



Bei solchen linearen zellulären Automaten nutzt man die zweite Dimension – meist nach unten – um vergangene Generationen zu dokumentieren. So entstehen häufig ganz spezifische Muster. Besonders bekannt sind die SIERPINSKI-Dreiecke.



SIERPINSKI-Dreiecke
Q: de.wikipedia.org (Cäsium137)



Gehäuse der Schnecke Cymbiola innexa
(Max-Planck-Institut für Entwicklungsbiologie + spiegel-online)
Q: http://www.spiegel.de/fotostrecke/muschelgehaeuse-muster-mit-formeln-erklaeren-fotostrecke-46503-8.html (10.02.2017)

(Lineare / Eindimensionale) Zelluläre Automaten lassen sich nach ihrem Verhalten in vier Klassen einteilen. Die Kriterien dazu stammen von Stephen WOLFRAM et.al.:

Definition(en): zellulärer Automat
Ein zellulärer Automat ist eine Kombination mehrerer gekoppelter Automaten, die i.A. einfachen Regeln folgen.
Zelluläre Automaten sind Netzwerke von meist gleichartigen Automaten.

Automaten-Klassen nach WOLFRAM

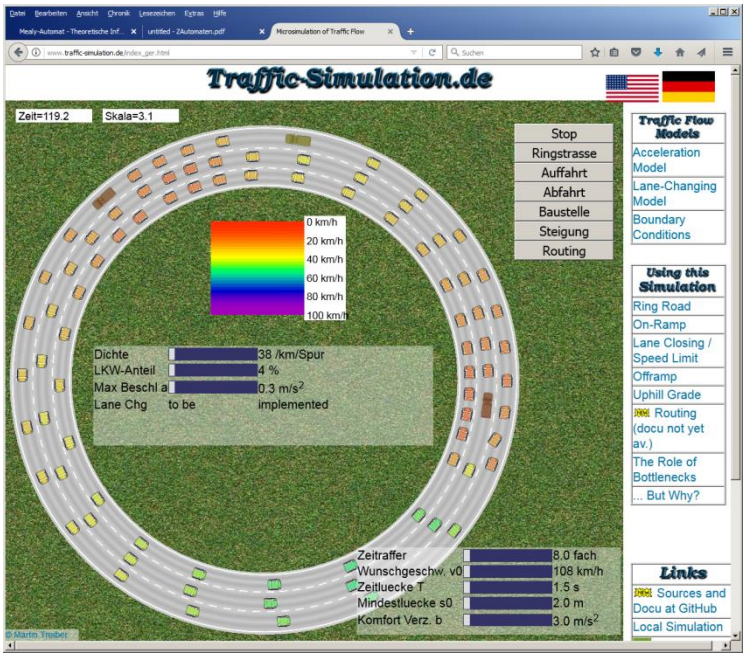
- **Klasse I** Automaten, die aus allen möglichen Ausgangs-Situationen unveränderliche Endzustände bilden (monotones Verhalten; räumlich homogener Fixpunkt)
  jeder Anfangs-Zustand konvergiert zu einem Fix-Punkt
  z.B.: totalistische Regel mit k=2, r=2 und Code 60; allg. Regel und Code 128

- **Klasse II** Automaten, die periodische Muster bilden (Es überleben häufig nur einzelne, mehr oder weniger breite Stränge. Es entstehen Stichcode-Muster. (endlose Zyklen derselben Zustände))
  jeder Anfangs-Zustand endet in einer Schleife von sich wiederholdenden Zuständen
  z.B.: totalistische Regel mit k=2, r=2 und Code 56; Code 4

- **Klasse III** Automaten zeigen chaotisches und unvorhersehbares Verhalten (mit ev. Struktur-Oasen); nehmen den gesamten Raum ein; größte Klasse
  unbegenztes Wachstum mit fester Wachstums-Rate
  es treten fraktale Muster und beliebige Perioden auf
  z.B.: totalistische Regel mit k=2, r=2 und Code 10; Code 22

- **Klasse IV** Automaten bilden komplizierte, ev. räumlich voneinander getrennte Strukturen mit isolierten, sich (in Raum und Zeit) bewegenden Teil-Strukturen; verkörpern den Rand zum Chaos; seltenste Klasse
  es kommt zu langlebigen, lokalen Mustern und Symmetrie-brechenden Konfigurationen
  z.B.: totalistische Regel mit k=2, r=2 und Code 20; Code 54

Lineare / Eindimensionale zelluläre Automaten können auch für einfache Verkehrs-Simulationen benutzt werden (NAGEL-SCHRECKENBERG-Modell). In solchen Fällen betrachtet man die Automaten in eine Richtung (meist von links nach rechts) hintereinander angeordnet – quasi wie eine Straße aus Abschnitten. Die Straße kann optimalerweise eine große Schleife darstellen. Die Auto's – also bestimmte Zell-Zustände – wechseln nach Berechnungen zur Geschwindigkeit (Beschleunigen, wenn frei Fahrt, Bremsen, wenn davor ein Auto fährt).
Die nebenstehend abgebildete online-Simulation ist etwas komplexer und realsitischer, aber zeigt keine wirklich anderen Effekte wie eine Simulation des NAGEL-SCHRECKENBERG-Modell auf einem eindimensionalen zellulären Automaten.



Stau's – scheinbar aus dem Nichts
Traffic-Simulation.de
Q: http://www.traffic-simulation.de/index_ger.html

Als Ergebnis entstehen Stau's, die zumeist auch noch weiter zu größeren Stau's aufschauckeln. Gleiche Ergebnisse erzielen auch deteministische Modelle, die auf Differenzial-Gleichungen basieren.

Aufgaben:

1. ***Rufen Sie die Wbseite möglichst zur gleichen zeit auf mehreren Rechnern auf! Beobachten Sie den Verkehrsfluß! wer ist der Verursacher eines oder mehrerer Stau's?***

Solche zellulären Automaten werden heute z.B. zur Simulation von Strömungen in Rohr-System usw. benutzt. Reine Berechnungs-Modelle sind extrem aufwendig und bringen im Vergleich zwar detailiertere aber keine sachlich besseren Ergebnisse.

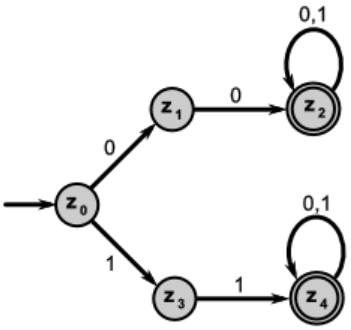
Links:
http://beltoforion.de/article.php?a=game_of_life&hl=de&p=running_the_simulation (Simulation in JAVA-script)
http://www.vlin.de/material/ZAutomaten.pdf (u.a. auch JAVA-Script Quelltexte)
http://www.mehr-davon.de/za/ (viele Zelluläre Automaten zum Ausprobieren)

weiterführende Literatur:
SCHOLZ, Daniel: Pixelspiele – Modellieren und Simulieren mit zellulären Automaten.-Berlin, Heidelberg: Springer Spektrum Verl., 2014
ISBN 978-3-642-45130-0 (eBook: 978-3-642-45131-7)

komplexe Aufgaben:

1.

2. Gegeben ist der Übergangs-Graph eines abstrakten Automaten.



a) Ordnen Sie den Automaten mindestens drei Klassen von Automaten zu! Begründen Sie die Zuordnung!

b) Welche Sprache akzeptiert der Automat?

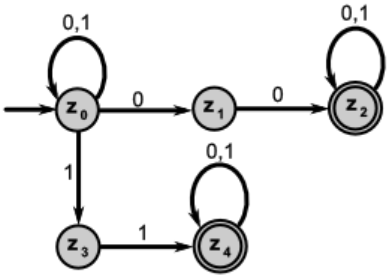
c) Definieren Sie den Automaten in Tupel-Form!

d) Geben Sie die Konfigurations-Folge für das Eingabe-Wort 000101 an!

3.

4.

5. Nebenstehender Zustands-Übergangs-Graph eines NEA soll eine Automaten bescheiben, der nur solche Eingabe-Worte erkennt, die einmal in der Mitte zwei Nullen enthalten..



a) Prüfen Sie die gemachte Leistungs-Aussage!

b) Welche Sprache akzeptiert der Automat?

c) Entwickeln Sie einen DEA, der die gleiche Sprache akzeptiert oder erklären Sie warum das nicht geht und es keinen pasenden DEA gibt!

d) Definieren Sie den DEA in Tupel-Form!

6.

weitere Übungs-Aufgaben (unsortiert, ohne Anspruchs-Differenzierung):

1. Erstellen Sie in einem geeigneten Simulation-Programm den folgenden DEA!
   A = ({z0,z1,z2,z3}, {"+","#"}, f, z0, {z3})
   Geben Sie jeweils 3 Wörter an, für die der Automat terminiert und nicht-terminiert!

f		
z	+	#
z0	z1	z3
z1	z2	z0
z2	z3	z1
z3	z0	z2

2. Entwickeln Sie einen NKA für die Sprache L = $\{+^n\#^{2n} \mid n>0\}$! (akzeptierte Wörter z.B.: +## +++###### ++++########
3. Entwickeln Sie einen DEA, der bei einer beliebig langen Eingabe von 0 oder 1 dann terminiert, wenn die letzten beiden Eingaben die Folge 10 sind!
4. Ein Automat soll Sequenzen von Einsen akzeptieren, wenn die Länge der Sequenz durch 3 teilbar ist,
   a) Planen Sie einen geeigneten Automaten M! Geben Sie diesen dann in der Tupel-Schreibweise an!
   b) Simulieren Sie den von Ihnen entwickelten Automaten mit einem geeigneten Programm!
   c) Prüfen Sie das exakte Funktionieren mit den folgenden Eingaben!

   a) 11 → nicht akzeptiert b) 111 → akzeptiert c) 111111 → akzeptiert
   d) → nicht akzeptiert e) 11111 → nicht akz. f) 111111111 → akz.
   g) 111111111111111 → akzeptiert h) 11111111111111111111 → nicht akz.
5. Erstellen Sie einen Automaten der Sequenzen von Einsen (also Wörter der Form 1^n) erkennt, deren Länge durch 3 oder 4 teilbar ist!
6. Gesucht wird ein NEA, der alle Wörter über das Alphabet {"#", "0"} akzeptiert, die mit "##" beginnen!
7. Erstellen Sie einen MOORE- und einen MEALY-Automaten, der die Negation einer binären Eingabe ausgibt! Geben Sie die Definitionen der Automaten an!
8. Erstellen Sie eine TURING-Maschine, die Gleichungen der Form: z.B. 1111=1111 oder 111111=111111 akzeptiert! (Die Anzahl der Einsen auf beiden Seiten der Gleichung soll übereinstimmen! Das Gleichheits-Zeichen wird als Symbol des Eingabe-Alphabet's verstanden.)
9. Gegeben ist die Sprache der Wörter über {"m","u"}*, die mehr u's enthalten als m's.
   a) Welche Art von Automaten müssen Sie für diese Aufgabe verwenden? Begründen Sie Ihre Meinung!
   b) Entwickeln Sie einen entsprechenden Automaten und realiseren Sie diesen in einem geeigneten Simulations-Programm!
   c) Testen Sie den Automaten mit jeweils 3 - von Ihnen vorüberlegten - Wörtern, die akzeptiert bzw. nicht-akzeptiert werden sollten!
10. Erstellen Sie einen NEA, der für Wörter aus dem Alphabet {"0","1"} prüft, ob diese mindestens 2x die Sequenz "10" enthalten!
11. Gesucht ist eine TURING-Maschine. welche die Sprachen L = $\{a^nb^nc^n \mid n>0\}$ akzeptiert!
12. Entwickeln Sie einen MOORE-Automaten, der das Einer-Kompliment einer Binär-Zahl berechnet! Geben Sie die Definition des Automaten an!

13. Entwickeln Sie einen Automaten A4, der Zahlen dahingehend untersucht, ob diese durch 4 teilbar ist (Hinweis: Teilbarkeits-Regel beachten / verwenden! Ziffern werden von links nach rechts eingegeben!)

14. Erstellen Sie in einem geeigneten Simulation-Programm den folgenden NEA!

f		
z	O	#
z0	{z0,z3}	{z0,z1}
z1	{}	{z2}
z2	{z2}	{z2}
z3	{z4}	{}
z4	{z4}	{z4}

A = ({z0,z1,z2,z3,z4}, {"O","#"}, f, z0, {z2,z4})

a) Dokumentieren Sie für die Eingabe "O#O##" die Konfigurations-Folge!

b) Zeichnen Sie einen Graphen für diesen Automaten!

c) Transformieren Sie den NEA in einen DEA!

15. Erstellen Sie einen Automaten (in Form eines Graphen), der für Zahlen prüft, ob diese durch 5 teilbar sind! (Die Zahlen werden Ziffern-weise von links nach rechts eingegeben!)

16. Planen Sie einen Endlichen Automaten, der eine Uhrzeit im Stunden-Minuten-Format akzeptiert!

17. Der folgende DKA soll die Sprache der exakt geklammerten arithmetischen Ausdrücke a realisieren.

A = ({S}, {"a","(",")","[",,"]"}, {S → a | SS | (S) | [S]}, S)

a) Zeichenen Sie einen Graphen für den Automaten!

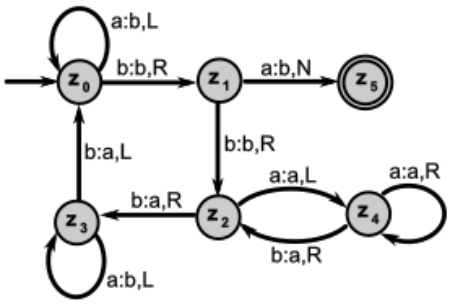
b) Setzen Sie den Automaten in einem geeigneten Programm um!

c) Prüfen Sie das exakte Funktionieren mit den folgenden Eingaben!

a) [] → nicht akzeptiert b) [(a)] → akzeptiert c) (((a (a)))) → akzeptiert
d) a) → nicht akzeptiert e) (([a))] → nicht akz. f) → akzeptiert
g) (a)((((a) a) a) a) → akzeptiert h) (((((x) x) x) (x) (x)))→ nicht akz.

18. Entwickeln Sie einen nicht-deterministischen Keller-Automaten, der die Sprache L = {0n1m2k | n,m,k> 0; n>m+k}! (Hinweis: Der Automat kann sich z.B. die Anzahl n merken und davon dann m und k abziehen!))

19. Realisieren Sie die abgebildete TM in einem geeigneten Simulations-Programm! Protokollieren Sie die Arbeit der TM bei Eingabe eines leeren Wortes!



20. Erstellen Sie eine Busy Beaver mit drei Zuständen, für den gilt: Σ = {0,1} und $ = 0!

21. Erstellen Sie einen MOORE-Automaten, der am Ende eine "1" ausgibt, wenn eine Buchstaben-Sequenz auf "a" endet! Eine "2" soll ausgegeben werden, wenn am Ende "aa" steht. In allen anderen Fällen soll eine "3" ausgegeben werden! (Es wird nur auf die letzte Ausgabe Wert gelegt!)

22. Entwickeln Sie einen EA, der ein deutsches (Kalender-)Datum (nur Tag und Monat) akzeptiert! (Alle Monate sind genau 30 Tage lang!)

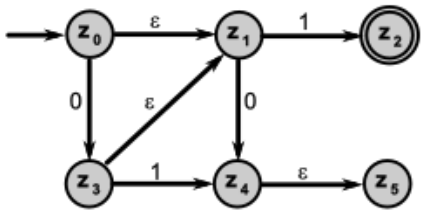
23. Planen Sie einen MEALY-Automaten, der ein Sandwich's ausgibt, wenn 3,50 € bezahlt wurden! Der Automat akzeptiert nur Euro-Geldstücke sowie 50-Cent-Münzen.

24. In einem Paket-Service sollen zugroße Pakete mit einem rotem Aufkleber versehen werden, wenn sie die Maße 3 x 5 x 4 überschreiten! (Einheit dm wird hier nicht betrachtet! Die Eingabe erfolgt immer in der gleichen Reihenfolge: Breite, Länge, Höhe (wie oben angegeben!))

25. Ist eigentlich ein Automat für den Paket-Service einfacher (zu bauen), wenn er die Maße immer von groß nach klein (also z.B. 5 x 4 x 3) bekommt? Überlegen Sie sich vorher eine Vermutung und erläutern Sie diese! Realisieren Sie dann einen entsprechenden Automaten!

26. Erweitern Sie den Automaten aus der vorherigen Aufgabe dahingehend, dass er zuviel gezahltes Geld zurückzahlt!

27. Nebenstehender NEA über das Alphabet $\Sigma = \{0,1\}$ ist gegeben.



a) Geben Sie die Definition dieses NEA an!

b) Entscheiden Sie, ob die Wörter: "00aa0", "000b0" und "0c0c0c0c" von dem NEA akzeptiert werden!

c) Wenden Sie ganz formal das Verfahren zu Entwicklung der zum Automaten gehörende reguläre Grammatik G an!

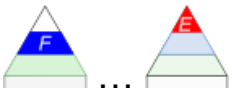
d) Testen Sie obige Test-Wörter auf G!

28. Erstellen Sie eine TURING-Maschine, die Gleichungen der Form: z.B. 1001=1001 oder 101110=101110 akzeptiert! (Die Binärzahlen auf beiden Seiten der Gleichung soll übereinstimmen! Das Gleichheits-Zeichen wird als Symbol des Eingabe-Alphabet's verstanden.)

3.2.16. CHOMSKY-Hierrarchie der Automaten

https://www.inf.hs-flensburg.de/lang/theor/chomsky-hierarchie-automat.htm

4. weitere Forschungs-Bereiche der Theoretischen Informatik



> Und doch verstehen die meisten Menschen nicht das Geringste von Computern. Wenn sie also nicht gerade über einen ausgeprägten Skeptizismus verfügen (etwa in der Art, wie man ihn einem Varietézauberer entgegenbringt), so können sie die intellektuellen Leistungen des Computers nur dadurch erklären, daß sie die einzige Analogie heranziehen, die ihnen zu Gebote steht, nämlich das Modell ihrer eigenen Denkfähigkeit. Dann nimmt es nicht wunder, daß sie über das Ziel hinausschießen; …
> Joseph WEIZENBAUM

Q: J. WEIZENBAUM: Die Macht der Computer und die Ohnmacht der Vernunft.-Suhrkamp Taschenbuch Wissenschaft (10. Aufl.; 2000).-S. 23

Problem-Fragen für Selbstorganisiertes Lernen

Können Computer alles berechnen?
Gibt es technische Grenzen der Berechenbarkeit?
Endet jeder Algorithmus mit einem Ergebnis?
Läßt sich die Berechenbarkeit mathematisch fassen und beweisen?
Läßt sich jede Funktion auch umgekehrt berechnen?
Gibt es eindeutig nicht berechenbare Funktionen? Kann man solche Funktionen für Verschlüsselungen usw. benutzen?
Was ist Entscheidbarkeit?
Was ist das Halte-Problem?

Was versteht man in der Theoretischen Informatik unter Komplexität?

4.1. Berechenbarkeit

Ist ein Problem berchenbar? Kann man beweisen, dass es berechenbar ist und kann man ermitteln, ob ein Computer das Problem mit den verfügbaren Ressourcen (Zeit und Speicher) überhaupt lösen kann? Solche und weitere Fragen sind Kern-Themen der Berechenbarkeits-Theorie.

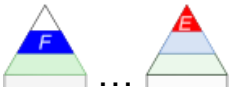
Berechenbarkeits-Begriffe

- **Semi-THUE-Systeme** THUE (1914)
- **Berechenbarkeit allgemein rekursiver Funktionen** HERBRAND, GÖDEL, KLEENE (1931 – 36)
- **Lambda-K-Berechenbarkeit** CHURCH, KLEENE (1936)
- **POSTsche kanonische Systeme** POST (1943)
- **MARKOV-Algorithmen** MARKOV (1954)
- **Register-Maschine** SHEPARDSON, STURGIS (1963)

Nach der CHURCH-TURING-These ist jeder Berechenbarkeits-Begriff gleichwertig. Etwas was nach dem einen Kriterien-System berechenbar ist, ist auch in den anderen Kriterien-System berechenbar. Nicht berechenbare Sachverhalte sind un bleiben "unberechenbar", egal welches Berechbarkeits-Kriterium man ansetzt.

Definition(en): Berechenbarkeit

4.1.1. Halte-Problem von TURING-Maschinen



Jeder Programmierer hat es wohl schon mal erlebt, dass ein Programm scheinbar unendlich rechnet (vielleicht wird es jeden Moment fertig) oder es steckt in einer Endlos-Schleife, die durch einen sematischen Fehler im Quellcode verursacht wird.
Es wäre wirklich toll, wenn unsere Quellcod-Übersetzer (Compiler oder Interpreter) neben den syntaktischen Fehlern auch bestimmte sematische Fehler finde würde. Gerade auf Fehler-bedingte Endlos-Schleifentesten zu können, wäre eine große Hilfe. Gesucht wir also ein Programm, dass genau solche Endlos-Schleifen finden kann und das auch bestimmen kann, ob unser neu programmiertes Programm regulär anhält.
Für sehr einfache Programme bekommen wir das ohne großen Aufwand oder intuitiv heraus. Gibt es nur eine Sequenz von Befehlen,:

```
Befehl1
Befehl2
Befehl3
```

dann ist ein Terminieren sehr wahrscheinlich. Aber eben auch nicht sicher. Schließlich kann eine der nach dem Übersetzen aufgerufenen Maschinen-Programme (z.B. des Betriebssystem's) – wegen eines dortigen Programmierfehlers – jetzt zufällig in eine Endlos-Schleife verfallen.
Kommt in einem Programm eine reinprogrammierte Endlos-Schleife a'la:

```
abbruch = False
while abbruch == False:
    # Scheifen-Inhalt
```

vor, dann kann man sich einer Endlos-Schleife schon wieder sehr sicher sein. Oder gibt es da irgendwo im Schleifen-Körper oder den aufgerufenen Funktionen eine unscheinbare, versteckte Programmzeile, die abbruch – wenn auch nur ganz, ganz selten – mal auf True setzt. Bei großen Programmen ist da intuitiv erst recht nichts mehr zu machen. Wir bräuchten unbedingt ein Programm, welches testet, ob ein Programm immer regulär endet. Dieses Programm sollte für alle Programme funktionieren, einschließlich sich selbst. Auch für das Überprüfungs-Programm wollen wir es schließlich wissen.
Aber programmieren wir uns zuerst einmal eine Test-Funktion, die haelt_an heißen soll und im Falle eines Anhalten's / Terminieren's ein WAHR zurückliefert. In den anderen Fällen soll ein FALSCH zurückgegeben werden.

```
FUNKTION haelt_an(Quellcode):
    # interne Kontrolle und Bewertung
    WENN terminiert:
        DANN rückgabe_wert = WAHR
        SONST rückgabe_wert = FALSCH
    RÜCKSPRUNG
```

Um die "Kleinigkeiten" – also die internen Kontrollen, Test's und die abschließende bewertung Bewertung in der Variable terminiert, erledigen wir später.
Zuerst schreiben wir noch schnell ein Haupt-Programm, das unserer Funktion **haelt_an()** irgendwelche Quellcode's übergibt und das Resultat dann ausgibt:

```
HAUPTPROGRAMM()
    WIEDERHOLE_SOLANGE haelt_an(HAUPTPROGRAMM()):
        DRUCKE("getestetes Programm terminiert")
    DRUCKE("getestetes Programm terminiert NICHT")
```

Super! Kontrollieren wir nun noch schnell die Programm-Logik, bevor wir uns um die oben erwähnten "Kleinigkeiten" kümmern.

Annahme: test() terminiert
dann liefert haelt() ein true zurück
damit läuft die while-Schleife in test() unendlich
somit terminiert test() nicht
→ Wiederspruch zur Annahme

Gegenannahme: test() termininiert nicht
dann liefert haelt() ein false zurück
damit würde die while-Schleife verlassen werden
somit terminiert test()
→ Wiederspruch zur Annahme

→ Annahme ist falsch: es gibt keine passende Funktion haelt()

ist nicht lösbar; allgemein nicht lösbar

```
funktion hält(programm,eingabe):
      wenn programm(eingabe) endet
      dann return 1
      sonst 0

unmöglich(eingabe):
      wenn ergebnis=hält(unmöglich,0)
      dann solange ergebnis=WAHR
            dann drucke("… das kann noch dauern …")
      sonst drucke("0")
```

Das Programm geht davon aus, dass die Funktion **hält** das Halte-Problem löst. D.h., dass bei Übergabe eines Programms die Funktion eine "1" zurückgibt, wenn das Programm terminiert. Ansonsten wird eine "0" zurückgegeben.
Für das Programm **unmöglich** ergibt sich nun ein Problem, wenn die Funktion **hält** berechnet, dass **unmöglich** terminiert, dann läuft **unmöglich** unendlich. Damit ergibt sich unlösbarer Widerspruch, der nur dadurch entsteht, dass wir davon ausgegangen sind (angenommen haben), dass die Funktion **hält** das Halte-Problem lösen kann.

Sprachen können dahingehend unterschieden werden, ob sie:

- entscheidbar oder rekursiv
- semi-entscheidbar oder rekursiv aufzählbar
- nicht rekursiv aufzählbar

sind.

Halte-Problem für / mit Perl

```
#!/usr/bin/perl
use strict;
use warnings;

sub haelt {
  my @quelltext = @_;
  my $rueckgabe;
  # hier unbekannter funktionierender Quelltext
  # wenn das Programm haelt, dann soll eine 1 zurückgegeben werden
  # wenn das Programm nicht hält, dann eine 0 zurückgegeben werden
  return $rueckgabe;
}


# wir lesen das Programm ein
my $datei = 'dateiname.pl';

open FILE, '<', $datei or die "konnte $datei nicht zum Lesen oeffnen.
$!\n";
my @zeilen = <FILE>;
close FILE;

if (&haelt(@zeilen) == 1) {
  my $x = 1;
  while ($x == 1) { };
}
else {
  print "haelt nicht!\n";
}

  END
```

Q: https://erasmus-reinhold-gymnasium.de/info/entscheidbarkeit/halteproblem.html

Bemerkungen
Gibt die Funktion "haelt" eine 1 zurück, so geht das Programm in eine Endlosschleife (im Quelltext: while ($x == 1) { };), hält also nicht.
Gibt die Funktion "haelt" eine 0 zurück, wird "hält nicht!" ausgegeben (im Quelltext: print "haelt nicht! ";) und das Programm hält.

wenn wir jetzt den eigenen Quelltext einlesen, führt dies zu folgenden Widersprüchen:
Kommt der Algorithmus zu dem Ergebnis, dass er hält, so geht das Programm hierdurch in eine Endlosschleife. -> **Widerspruch**
Umgekehrt, wird zurückgegeben, dass das Programm nicht hält, so hält es dadurch. -> **Widerspruch**

Fazit:
Das Halte-Problem ist nicht entscheidbar.
Q: https://erasmus-reinhold-gymnasium.de/info/entscheidbarkeit/halteproblem.html

Halte-Probleme für / mit JAVA

```
boolean haelt(String programm){
    if (haelt(uebersetzt(programm)))
        retrurn true;
    else
        return false;
}

void test(){
    while (haelt("test();"));
}
```

Annahme: test() terminiert
dann liefert haelt() ein true zurück
damit läuft die while-Schleife in test() unendlich
somit terminiert test() nicht
→ Wiederspruch zur Annahme

Gegenannahme: test() termininiert nicht
dann liefert haelt() ein false zurück
damit würde die while-Schleife verlassen werden
somit terminiert test()
→ Wiederspruch zur Annahme

→ Annahme ist falsch: es gibt keine passende Funktion haelt()

Unterscheidbarkeit von Sprachen hinsichtlich der Berechenbarkeit

- **entscheidbar** oder **rekursiv**
  es existiert eine deterministische TURING-Maschine, die bei jeder Eingabe terminiert, aber nur die Worte der Sprache akzeptiert
- **semi-entscheidbar** oder **rekursiv aufzählbar**
  es existiert eine deterministische TURING-Maschine, die bei die Eingabe eines gültigen Wortes terminiert und akzeptiert (hält); bei nicht Worten, die nicht zur Sprache gehören aber nicht hält
  → Beispiel-Sprache: Halte-Problem
- **nicht rekursiv aufzählbar**
  es existiert eine deterministische TURING-Maschine, die bei die Eingabe eines gültigen Wortes nicht hält, wohl aber bei nicht gültigen Worten
  → Beispiel-Sprache: komplement des Halte-Problem's

4.1.2. Berechenbarkeit von Funktionen / Algorithmen

4.2. Komplexität

Viele Dinge / Funktionen sind zwar berechenbar, aber oft ist der Speicher- oder Zeit-Aufwand so groß, dass das Ding / die Funktion eben praktisch nicht berechenbar ist. Es sind soviele Schritte notwendig, dass die Maschine / der Computer ewig rechnen würde oder einen unendlich großen Speicher bräuchte, der ebenfalls nicht verfügbar ist.

Komplexität ist ein Maß für die Verflechtung der Komponenten eines Systems.

Komplexität von Algorithmen ist heute oft das Einsatz-Kriterium z.B. in der Blockchain-Technologie
ein viel zittiertes Beispiel ist die Bitcoin-Krypto-Währung
Hier müssen Teilnehmer nachweisen, dass sie bestimmte Resourcen besitzen, um sich als einzelner Accout zu outen und zum Anderen einen Betrug zu verhindern, weil jeder jeden beliebigen (eben auch falschen Wert) in die Blockchain einbringen könnte.
Die Berechner (hier Miner genannt) müssen komplizierte Aufgaben lösen, für die es nach heutigen Stand keine einfachen Lösungen gibt.
Mit der Änderung der Rechen-Technologie z.B. auf Quanten-Rechner könnte diese Art der Absicherung allerdings zu Staub zerfallen.

Als Beispiel haben wir bei den "Fleißigen Bibern" schon die Zusammenhänge zwischen der Anzahl der Zustände und der Anzahl der möglichen TM bzw. die Anzahl der geschriebenen Einsen gesehen. Man nennt den Zusammenhang nach dem Erfinder der "Fleißigen Biber" RADO-Funktion.

Eine weitere extrem schnell wachsende Funktion ist die rekursive ACKERMANN-Funktion.

$$
\begin{aligned}
ack(a, b, 0) &= a + b \\
ack(a, 0, n+1) &= ack2(a, n) \\
ack(a, b+1, n+1) &= ack(a, ack(a, b, n+1), n)
\end{aligned}
$$

$$
ack2(a, n) = \begin{cases} 0, & \text{wenn } n=0 \\ 1, & \text{wenn } n=1 \\ a, & \text{wenn } n>1 \end{cases}
$$

durch PÈTER 1935 etwas einfacher definiert:

$$
\begin{aligned}
ack(0, m) &= m+1 \\
ack(n+1, 0) &= ack(n, 1) \\
ack(n+1, m+1) &= ack(n, ack(n+1, m))
\end{aligned}
$$

rekursiv (PYTHON):

```
def ackermann(n, m):
    if n==0:
```

```
        return m+1
    elseif  m==0:
        return ackermann(n-1, 1)
    else:
        return ackermann(n-1, ackermann(n, m-1)
```

teilweise interativ (PYTHON):

```
def ackermann(n, m):
    while n!=0:
       if m==0:
           m=1
       else:
           m=ackermann(n, m-1)
       n+=1
    return m+1
```

einfaches Durchsuchen einer Liste bedarf im Allgemeinen n Schritte, jedes Element muss einmal betrachtet werden; Programmierer würden hier sicher einfache FOR- oder WHILE- bzw. REPEAT-Schleifen benutzen
wächst die Liste, dann wächst auch die Anzahl der Schritte im gleichen Maß
somit linearer Zusammenhang zwischen Größe der bearbeiteten Daten oder der Größe des Daten-Objektes und dem Rechen- bzw. Zeit-Aufwand
selbst die Steigerung von n um 10er Potenzen hat nur eine gringe Wirkung

ein Argument beim Besprechen der Komplexität eines Algorithmus ist immer, ob man für etwas 2 oder 3 oder vielleicht 14 Schritte (Befehle) braucht
Praktisch würde eine solche Zahl als (ungefährer Faktor) vor dem n stehen. Bräuchte ein Algorithmus eben 3 Schritte würde das z.B. 3n bedeuten, bei 14 Schritten dann 14n. Bei linearen Zusammenhängen spielt der Faktor natürlich schon eine Rolle, aber er änder nichts am grundsätzlichen – eben linearen Zusammenhang. Solche Faktoren werden in der Betrachtung von Komplexitäten auch nicht beachtet, weil vielleicht in einer anderen Programmiersprache nur noch ein Schritt notwendig ist. In der Komplexitäts-Theorie geht es aber um grundsätzliche Zusammenhänge. Wie wir später noch sehen werden, kommt den Faktoren insgesamt auch wirklich nur eine untergeordnete Rolle zu.

weitere linear-komplexe Algorithmen:

- Prüfen, ob eine Liste sortiert ist
- Aufsummieren der Elemente einer Liste / eines Array's / …
- Bilden der Abweichungs-Quadrate in Datenreihen
- Berechnen des Mittelwertes einer Datenreihe
- …

Aufgaben:

1. ***Erstellen Sie in Ihrer favorisierten Programmiersprache ein Programm, dass ein Liste oder ein Array od.ä. mit einer vorgegebenen Länge erzeugt! Belegen Sie die Elemente mit einer einfachen Zufalls-Zahl! In einer separaten Funktion oder Schleife soll dann die Summe der Elemente gebildet werden! Lassen Sie sich die Rechnen-Zeit für die Zufalls-Belegung und die Summen-Bildung anzeigen!***
   Dokumentieren Sie die jeweiligen Zeiten für die Listen-Länge von 1, 10, 100, 1'000 und 10'000 Elementen! Stellen Sie den Zusammenhang graphisch dar!

2. *Erstellen Sie sich Kopien vom Programm aus Aufgabe 1! Variieren Sie nun die Berechnungs-Scheife:*
   a) *statt der Summe soll das Produkt gebildet werden*
   b) *statt der Summe soll der Mittelwert berechnet werden*
   c) *die Zufallszahl soll zuerst Verzehntfacht werden und dann die Wurzel berechnet werden, das Ergebnis geht in die Summen-Bildung ein*
   d) *gerade Listen-Werte werden verdoppelt und ungerade verdreifacht, die Ergebnisse gehen in die Summen-Bildung ein*
3.

Auch, wenn wir hier vorrangig auf die Rechen-Zeit oder die Algorithmen-Schritte geschaut haben, bei anderen Komplexitäts-Betrachtungen spielt z.B. der Speicher-Bedarf eine Rolle. Gerade bei rekursiven Algorithmen, kann es durch immerwährende Selbst-Aufrufe der Funktionen soviele Rücksprünge geben, dass der Speicher einfach nicht mehr ausreicht. Das Problem wird besonders dann akut, wenn komplexere Daten-Strukturen (z.B. mehrdimensionale Felder) in die rekursion einfließen. Ein Beispiel hierfür könnten Strömungs-Berechnungen von Flüssigkeiten in Rohr-Systemen sein.
Ob man nun den rechen-Aufwand oder den Speicher-Bedarf betrachtet, an den grundsätzlichen Zusammenhängen ändert sich dabei nicht.
Komplexität in der Informatik kann sich also auf verschiedene Merkmale beziehen. Der Speicher- und Rechenzeit-Bedarf sind dabei sicher die bedeutensten.

schnell würde man denken, schneller oder besser als n geht es nicht, aber der Eindruck täuscht
in Bäumen kann man noch effektiver auf Daten zugreifen
hier steigt der Zeit- oder Rechen-Aufwand nur mi der Wurzel aus n
oft als Logarithmus von n ausgedrückt

weitere logarithmisch-komplexe Algorithmen:

- Durchsuchen einer Liste (z.B. Telefonbuch, …) nach dem Teile-und-herrsche-Prinzip

Für viele Daten und / oder Algorithmen sind zwei oder mehrere ineinander geschachtelte Schleifen notwendig. Hier können wir schon abschätzen, dass der Aufwand mindestens n * m ist. Bei mehreren Schleifen-Ebenen kommen ev. auch noch mehrere Faktoren hinzu. Vereinfachen wir auf gleiche Längen, dann ergeben sich Rechen-Zeiten, die mit n^2, n^3 usw. usf. steigen.
Solche Algorithmen haben ein polynomïelle Komplexität. Ergäbe sich z.B. eine Berechnungs-Formel $a\ n^4 + b\ n^3 + n$, dann betrachtet man immer nur die höchste Potenz. In unserem Fall wäre also "nur" die 4. Potenz interessant

Aufgaben:

1. *Erstellen Sie in Ihrer favorisierten Programmiersprache ein Programm, dass ein Liste oder ein Array od.ä. mit einer vorgegebenen Länge erzeugt! Belegen Sie die Elemente mit einer einfachen Zufalls-Zahl! In einer separaten Funktion oder Schleife soll dann die Sortierung der Elemente vorgenommen werden! (Es reicht z.B. Bubble-Sort!)*

Lassen Sie sich die Rechnen-Zeit für die Zufalls-Belegung und die Summen-Bildung anzeigen!
Dokumentieren Sie die jeweiligen Zeiten für die Listen-Länge von 1, 10, 100, 1'000 und 10'000 Elementen! Stellen Sie den Zusammenhang graphisch dar!

2. *Wie verändert sich das Laufzeit-Verhalten, wenn Sie einen anderen Sortier-algorithmus verwenden! Geben Sie im Vorfeld Vermutungen ab! (In größeren Kursen kann Arbeits-teilig mehrere Algorithmen geprüft werden!)*
3. *Geben Sie zu allen bisher betrachteten Algorithmen die Komplexität an!*

Aufgaben:

4. *Welche Komplexitäts-Metriken gibt es? Wählen Sie drei aus und erläutern Sie diese kurz!*
5. *?.*

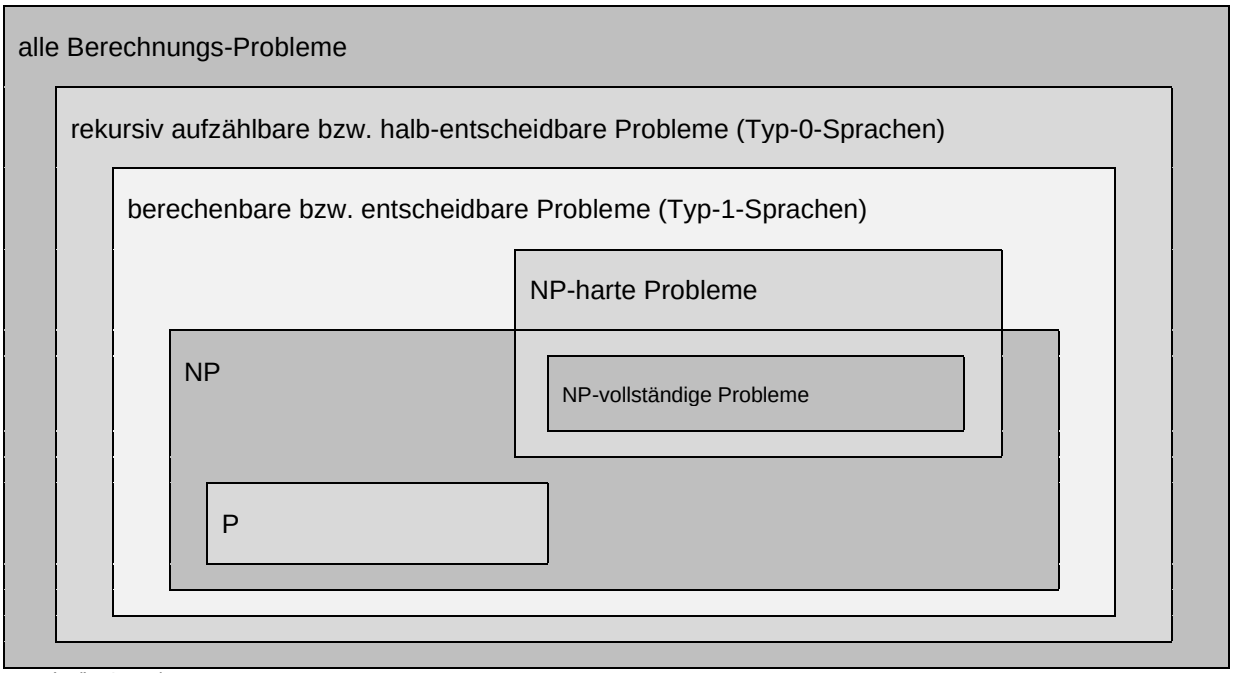
Der informatische "GAU" sind Algorithmen, die mit expotentieller Komplexität ausgestattet sind. Dazu gehören z.B.:

- Primfaktoren-Zerlegung
-
- Rundreise-Problem (jeder Knoten muss besucht werden, dabei wird der kürzeste Weg insgesamt gesucht; Start und Ende sind der gleiche Knoten) → HAMILTON-Kreis
- Berechnung eines Ausgangs-Dokumentes zu einem Hash-Wert, bzw. zu einem Hash-Wert mit besonderen Bedingungen (z.B. 6 Nullen am Anfang)
- Rucksack-Problem (Optimierung der Auswahl von Objekten (beim Rucksack-Problem: Objekte mit unterschiedlichen Werten und Gewichten bei einem insgesamt zulässigen Gesamtgewicht (für den Rucksack))
-

Komplexitäts-Klassen

• **Klasse P (polynomialer Aufwand)**	z.B.: rekursive Sortier-Algorithmen (O(n log n)), Such-Algorithmen (O(n)); GAUSS-Verfahren ($O(n^3)$)
• **Klasse NP (nicht polynomialer Aufwand)**	z.B.:
• **Klasse EXP (expotentieller Aufwand)**	praktisch nicht sinnvoll lösbare Probleme z.B.:

Definition(en): Komplexität
Die Komplexität eines Algorithmus (zu einem Problem) die Charakterisierung des Aufwandes (meist Speicher und / oder Zeit) des besten (optimalsten) funktionierenden Algorithmus, der genau das Problem löst.
Unter Komplexität versteht man den maximalen Ressourcen-Verbrauch / -Bedarf, den ein Algorithmus zur Lösung eines Problems benötigt / anfordert / verwendet.



nach: /b, S. 44/

P … polynomiale Komplexität
NP … nicht-polynominale (gemeint ist über-polynominale) Komplexität

Lösungen von Problemen können vorgegeben werden (z.B. von einem allwissenden Partner (Weisen)).
Kann man diese Lösungen mit normalem Aufwand prüfen, dann handelt es sich um eine Problem aus der NP-Klasse.

z.B.: Rundreise-Problem
(jeder Knoten eines Graphen wird genau einmal besucht, Anfang und Ende liegen auf dem gleichen Knoten)
Lösung ist z.B. eine Liste der Knoten (Orte der Rundreise, außer Ziel-Ort (weil Start-Ort))
Prüf-Verfahren (Verifikation für Rundreise-Problem)

- **Prüfung auf einmaliges Vorkommen der Knoten** — polynomielle Komplexität → linearer Zeitaufwand
- **Prüfung, ob zwischen zwei aufeinanderfolgenden Knoten eine Kante existiert** — polynomielle Komplexität → linearer Zeitaufwand

- **Start-Ende-Prüfung**
  (existiert eine Kante zwischen dem letzten Knoten der Liste und dem Start-Knoten)

ist ein Problem NP-vollständig und es existiert ein polynomieller Algorithmus hierfür, dann existiert auch für jedes anderes NP-vollständige Problem ein solcher (polynomieller) Algorithmus
jedes NP-vollständige Problem kann durch jedes andere NP-vollständige Problem gelöst werden
man sagt die Probleme sind aufeinander reduzierbar

Aufgaben:

1.
2.
3.

für die gehobene Anspruchsebene:

?. Recherchieren Sie was man unter einem transcomputationalen Problem versteht!
?.

Komplexitäts-Grad	**Formulierung $\Theta(x)$**	**Beispiel(e)**
konstant	c	Binär-Darstellung mit b Bit's → $\Theta(b)$
logarithmisch (organisch)	log n	schnelles Potentieren nach LÈGENDRE → $\Theta(\log n)$
linear	n 1+n n+log c n	normale Suche → n
		String-Suche in einem String der Länge k nach KNUTH-MORRIS-PRATT (KMP-Suche) → $\Theta(n+k)$
		String-Suche nach BOYER-MOORE → $\Theta(n+k)$
Ordnung n log n	n log n	Quicksort → $\Theta(n \log n)$
	log log n	Interpolations-Suche → $\Theta(\log \log n)$
	n log log n	Sieb des ERATHOSTHENES → $\Theta(n \log \log n)$
quadratisch	n^2	Bubble-Sort (mit 2 Schleifen) → $\Theta(n^2)$
kubisch	n^3 cn^3+d	
polynomial polynomisch	n^c $cn^2 + dn$	
exponentiell	c^n e^n c^{dn} c^n+n^3	FIBONACCHI-Folge → $\Theta(1{,}618^n)$
superexponentiell	n^n	Fakultäts-Funktion n! → $\Theta(n^{n+0,5})$

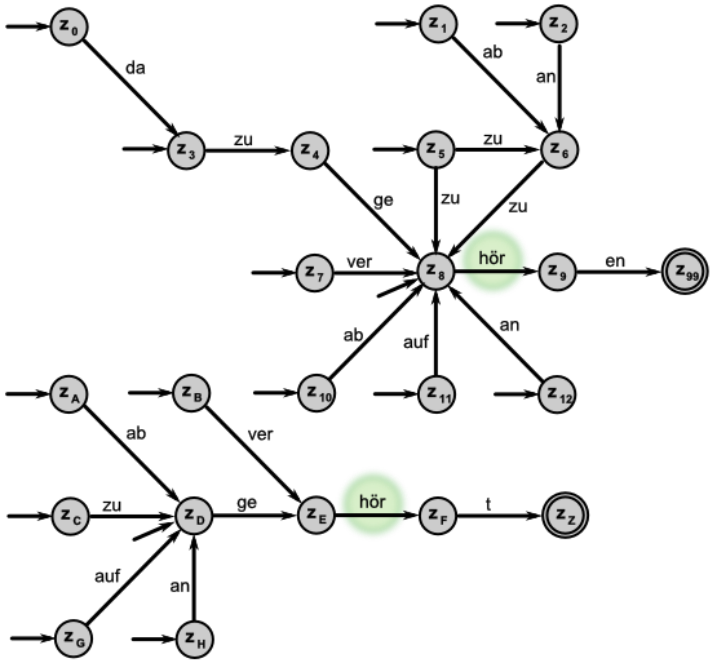
komplexe Aufgaben (zur Vorbereitung auf Klausuren / Prüfungen, …):

1.

x. Viele Programmiersprachen verwenden Variablen-Namen, die mit einem Buchstaben beginnen und dann beliebige Folgen von Buchstaben, Ziffern und Unterstrichen enthalten können. Entwicklen Sie einen Automaten, der solche Variablen-Namen akzeptiert!

x. In Python gibt es Variablen-Namen die mit einem bzw. zwei Unterstrichen beginnen und enden. Erstellen Sie einen oder mehrere Automaten, der / die diese Variablen erkennt / erkennen!

x. Gegeben sind die zwei Automaten, die Wörter um den Wortstamm "hör" herum erkennen sollen! Ordnen Sie die Automaten einem Automaten-Typ zu! Begründen Sie Ihre Wahl! Begründen Sie anhand der beiden Automaten, dass diese auch den jeweils übergeordneten Typen zugeordnet werden können! Belegen Sie für mögliche untergeordnete Automaten-Typen, dass diese nicht mehr zutreffen!



x. Geben Sie die vollständige Sprache der beiden Automaten an!

x. Gesucht ist ein Automat, der für Bit-Folgen deren gerade Parität ausgibt!

x. Ein Mikro-Computer-System soll mit einem kleinen PASCAL-ähnlichen Programmier-System ausgestattet werden. Dieses soll die nur die Variablen a, b, c, x, y und z benutzen können. Die genannten Variablen können als i (integer = Ganzzahl), r (real = Gleitkommazahl) oder t (text = Zeichenkette) deklariert werden. Das Schlüsselwort zur beginnenden Variablen-Deklaration heißt v. Zwischen dem Variablen-Namen und dem –Typ muss ein Doppelpunkt stehen. Einzelne Deklarationen werden durch ein Semikolen beendet.

a) Erstellen Sie ein Syntax-Diagramm für die Variablen-Deklaration des Mini-Programmier-Systems!

b) Erstellen Sie eine graphische Repräsentation eines Automaten, der die Variablen-Deklaration prüft!

c) Definieren Sie den Automaten in Text-Form! Für die Funktion(en) darf eine / dürfen geeignete Tabelle(n) (Automaten-Tafel) benutzt werden!

d) Geben Sie die Konfigurations-Folge Ihres Automaten für die folgende Variablen-Deklaration an!

v a:t; y:i;

x. Zu entwickeln sind zwei EA, von denen der eine prüfen soll, ob eine Dualzahl durch 2 teilbar ist. Der andere soll die Teilbarkeit durch 64 prüfen!

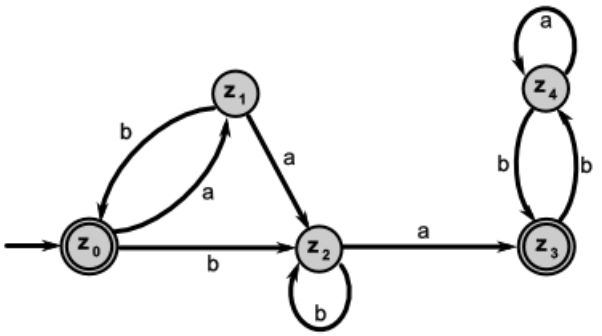
x. Gesucht ist ein Automat, der römische Zahlen bis 1000 (M) akzeptiert!

x. Erstellen Sie einen Automaten, der den Anfang von "Mensch-ärgere-Dich-nicht" (Heraussetzen der ersten Spiel-Figur) prüft!

x.

x. Gegeben sein soll der nebenstehende DEA.

Aufg. nach Q: /5, S. 32 (geänd.: dre)/



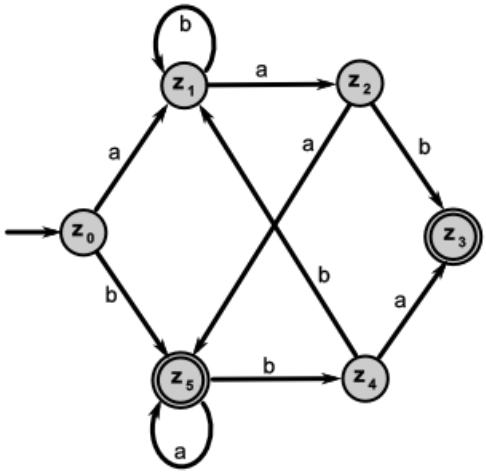
a) Prüfen Sie, ob es sich um einen DEA handelt! Begründen Sie Ihre Meinung!

b) Testen Sie die nachfolgenden Wörter auf Akzeptanz durch den Automaten!

b1) ab b2) aab b3) baa
b4) aabba b5) aabbab b6) baba
b7) bbba b8) bb b9) a b10) b b11) aaaabbaacaa b12) aabbaabbaab
b13) abababb

x. Gegeben sein soll der nebenstehende DEA.

Aufg. nach Q: /5, S. 32 (geänd.: dre)/



a) Prüfen Sie, ob es sich um einen DEA handelt! Begründen Sie Ihre Meinung!

b) Testen Sie die nachfolgenden Wörter auf Akzeptanz durch den Automaten!

b_1) ab b_2) aab b_3) baa
b_4) aabba b_5) aabbab b_6) baba
b_7) bbba b_8) bb b_9) a b_{10}) b
b_{11}) aaaabbaacaa b_{12}) aabbaabbaab
b_{13}) ababab b_{14}) abababa

c) Geben Sie eine vollständige Definition des Automaten an!

x. Von einem angeblichen NEA ist die nebenstehende Tabelle der Übergangs-Funktion übrig geblieben.

	Eingabe-Symbole	
Zustand	**0**	**1**
s	s,1	s
1	2	-
2	3	3
3	4	-
4	4	4

Aufg. nach Q: /5, S. 32 (geänd.: dre)/

a) Erstellen Sie eine Automaten-Definition! (Zustände, die vollständig zu sich selbst führen, gelten als final.)

b) Prüfen Sie, ob es sich tatsächlich um einen NEA handelt! Begründen Sie!

c) Erstellen Sie den Automaten in einer geeigneten Simulations-Umgebung!

d) Testen Sie die folgenden Worte auf Akzeptanz!

d_1) 00110 d_2) 001101 d_3) 1010 d_4) 1110 d_5) 11 d_6) 0 d_7) 1
d_8) 110001101000 d_9) 110011001000

x. Gegeben ist der nebenstehende Automat.

Aufg. nach Q: /5, S. 33 (geänd.: dre)/

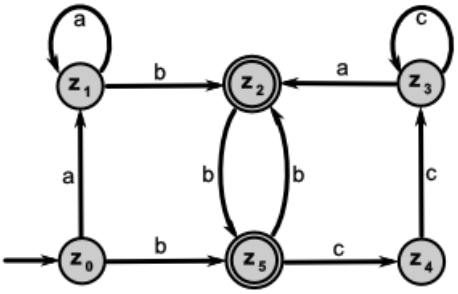
a) Gesucht wird die Sprache des Automa-

ten!

b) Erstellen Sie den Automaten in einer geeigneten Simulations-Umgebung!

c) Prüfen Sie die folgenden Worte auf Zugehörigkeit zur Sprache und auf Akzeptanz in der Automaten-Simulation!

d_1) abb d_2) b d_3) aaaabbb d_4) bad
d_5) abcabc d_6) bcccabbb d_7) z_0 z1 z2



x. Gegeben ist der nebenstehende Automat.

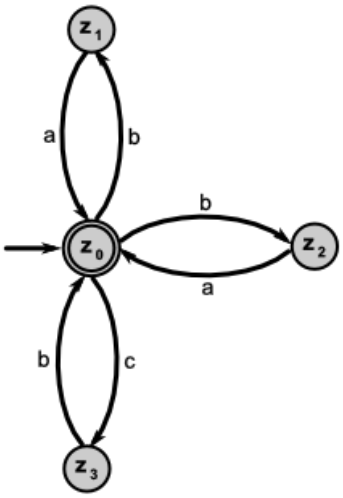
Aufg. nach Q: /5, S. 33 (geänd.: dre)/

a) Gesucht wird die Sprache des Automaten!

b) Erstellen Sie den Automaten in einer geeigneten Simulations-Umgebung!

c) Prüfen Sie die folgenden Worte auf Zugehörigkeit zur Sprache und auf Akzeptanz in der Automaten-Simulation!

c_1) abb c_2) b c_3) aaaabbb c_4) bad
c_5) abcabc c_6) bcccabbb c_7) z_0 z1 z2



komplexe Aufgaben (für die gehobene Anspruchsebene und Freaks):

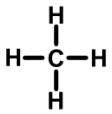
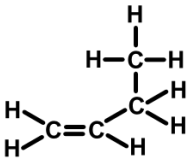
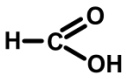
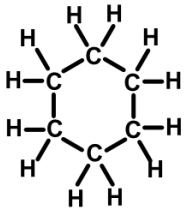
x. Zur Darstellung einer chemischen Struktur (gemeint sind vorrangig organische Stoffe) als Einzeiler verwendet man die canonische SMILES-Codierung (Simplified Molecular Input Line Entry Specification) mit den folgenden Regeln: (SYBYL Linearnotation)

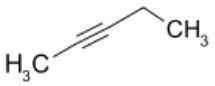
1. für Atome werden ihre Element-Symbole verwendet; Ionen-Ladungen notiert man in eckigen Klammern in der Reihenfolge: Art und Anzahl; z.B. [+2]
2. Wasserstoff-Atome müssen mitgeschrieben werden
3. benachbarte Atome werden nebeneinander notiert; ev. wird ein Punkt zur Trennung von Gruppen oder Ionen benutzt
4. Doppel- und Dreifach-Bindungen werden durch = bzw. # und aromatische Bildungen durch : dargestellt
5. Verzweigungen werden in runden Klammern notiert
6. Ringschlüsse werden durch Zahlen in eckigen Klammern zu den Ring-Atomen gekennzeichnet; mit @ wird der Ringschluß gekennzeichnet

(zusätzliche Vereinbarungen, die wir hier nicht beachten:

a. in eckigen Klammern dürfen an die Bindungen auch Bindungs-Attribute geschrieben werden; z.B.: [s=t] für eine trans-Doppel-Bindung
b. für Atom-Gruppen / Makro-Atome sind die üblichen Kurzschreibweisen zulässig; z.B.: Gln für die Aminosäure Glutamin)

Beispiele:

Methan	But-1-en	Methansäure	Cyclohexan
			
CH_4	CH_2=CH-CH_2-CH_3	HCOOH	C_6H_{12}
CH4	CH2=CHCH2CH3	CH3C(=O)OH	C[1]H2CH2CH2CH2CH2CH2@1

Pent-2-in	Benzen	Furan
		
CH_3-C≡C-CH_2-CH_3	C_6H_6	C_4H_4O
CH3C#CCH2CH3	C[1]H:CH:CH:CH:CH:CH: [1]	O[1]:CH:CH:CH:CH:@1

a) Konstruieren Sie einen akzeptierenden Automaten (nur für unverzweigte, verzweigte und cyclische Alkane sowie Aromaten)!

b) Erstellen Sie den Automaten in einer geeigneten Simulations-Umgebung!

Literatur und Quellen:

/1/

ISBN

/2/ SOCHER, Rolf:
Theoretische Grundlagen der Informatik.-München: Carl Hanser Verl.; 2008.-3. aktual. u. erw. Aufl.
ISBN 978-3-446-41260-6

/3/ MAGENHEIM, Johannes; u.a.:
Informatik macchiato – Cartoon-Informatikkurs für Schüler und Studenten.-München, …: Pearson Studium; 2009
ISBN 978-3-8273-7337-3

/4/ ENGELMANN, Lutz (Hrsg.):
Informatik – Gymnasiale Oberstufe.-Berlin, Frankfurt a.M.: DUDEN PAETEC Schulbuchverl.-2006.-1. Aufl.
ISBN 978-3-89818-622-3

/5/ BATTENFELD, G.; GRIMM, P; POLOCZEK, J.; SCHMIDT, E.; WAHA, R.:
Theoretische Informatik – Planung eines Kurses in der Jahrgangsstufe 13 I.-Entworfen im Rahmen eines Wochenlehrgangs "Didaktik der Informatik

/1/

ISBN

Online-Skripte, …:

/a/ WAGNER, Dorothea:
"Theoretische Grundlagen der Informatik – Endliche Automaten und reguläre Ausdrücke"
https://i11www.iti.uni-karlsruhe.de/_media/teaching/winter2011/tgi/vorlesung-1-2.pdf (01.02.2017)

/b/ ERTEL, Wolfgang:
"Theoretische Informatik" (2008)
http://www.hs-weingarten.de/~ertel/vorlesungen/thinf/skript.pdf (01.02.2017)

/c/ Theoretische Informatik / Das Prinzip des Automaten (2012)
https://de.wikibooks.org/wiki/Theoretische_Informatik/_Das_Prinzip_des_Automaten (01.02.2017)

/d/ LAUX, F.:
Vorlesung Theoretische Informatik – Automaten und Formale Sprachen (2010)
http://dblab.reutlingen-university.de/tl_files/TeachingResources/Theoretische%20Informatik/thi_02_automatenFL.pdf (05.02.2017)

/e/ Blitzkurs Theoretische Informatik (2007)
https://de.wikibooks.org/wiki/Blitzkurs_Theoretische_Informatik/_Druckversion
(07.0.2.2017)

/f/ BECKMANN, Hans-Georg:
Zelluläre Automaten (2003)
http://www.vlin.de/material/ZAutomaten.pdf (10.02.2017)

/g/

(2017)

/h/

(2017)

/A/ Wikipedia
http://de.wikipedia.org

https://de.wikibooks.org/wiki/Blitzkurs_Theoretische_Informatik/_Druckversion

Die originalen sowie detailliertere bibliographische Angaben zu den meisten Literaturquellen sind im Internet unter http://dnb.ddb.de zu finden.

Abbildungen und Skizzen entstammen den folgende ClipArt-Sammlungen:

/A/ 29.000 Mega ClipArts; NBG EDV Handels- und Verlags AG; 1997

/B/

andere Quellen sind direkt angegeben.

Alle anderen Abbildungen sind geistiges Eigentum:

/I/ lern-soft-projekt: drews (c,p) 1997 – 2024 lsp: dre

verwendete freie Software:

- **Inkscape** von: inkscape.org (www.inkscape.org)
- **CmapTools** von: Institute for Human and Maschine Cognition (www.ihmc.us)

⊞- (c,p) 2016 - 2024 lern-soft-projekt: drews -⊟
⊞- drews@lern-soft-projekt.de -⊟
⊞- http://www.lern-soft-projekt.de -⊟
⊞- 18069 Rostock; Luise-Otto-Peters-Ring 25 -⊟
⊞- Tel/AB (0381) 760 12 18 FAX 760 12 11 -⊟