Video

(Grundlagen der) Theoretische(n) Informatik

Kernziele:
- Bedeutung von Formalisierung erkennen
- informelle Inhalte formal im passenden Modell darstellen
- Grenzen der Programmierbarkeit ausloten

Ich

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Stephan Kleuker, geboren 1967, verheiratet, 2 Kinder
- seit 1.9.09 an der HS, Professur für Software-Entwicklung
- vorher 4 Jahre FH Wiesbaden
- davor 3 Jahre an der privaten FH Nordakademie in Elmshorn
- davor 4 ½ Jahre tätig als Systemanalytiker und Systemberater in Wilhelmshaven
- davor 6 Jahre wissenschaftlicher Mitarbeiter mit Promotion im Bereich Theoretische Informatik an der Carl-von-Ossietzky-Universität Oldenburg

- s.kleuker@hs-osnabrueck.de, Zoom-Termine kurzfristig per E-Mail vereinbar

Ablauf

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- eigentlich 4h Vorlesung = 5 CP, genauer: 2h Vorlesung + praktikumsartige Diskussionsrunden (2h)
- Praktikumsaufgaben freiwillig, bereiten die Klausur vor bzw. zeigen praktische Umsetzung als Programmieraufgaben in Java
- wieso 4h? Sie haben während der Vorlesungszeit und der gesamten Praktikumszeit die Möglichkeit Fragen zu stellen
- Prüfung durch Klausur nach VL-Zeit
- Folienveranstaltungen sind schnell, bremsen Sie mit Fragen, nutzen Sie bei Videos die Stopp-Taste zum Nachdenken und Fragen zu formulieren
- von Studierenden wird hoher Anteil an Eigenarbeit erwartet
- falls Wiederholung der Prüfung notwendig, dringend empfohlen an gesamter Veranstaltung von Prof. Dr. Morisse teilzunehmen (überlappender Inhalt, aber andere didaktische Aufbereitung)

Literatur

- K. Morisse, Einführung in die Theoretische Informatik, Vorlesungsskript Hochschule Osnabrück, aktuelle Version
- (Ergänzung zu einzelnen Teilthemen) S. Kleuker, Formale Modelle der Softwareentwicklung, Vieweg+Teubner, Wiesbaden, 2009

[nur Formale Sprachen und Komplexitätstheorie]

- C. Wagenknecht, M. Hielscher, Formale Sprachen, abstrakte Automaten und Compiler, 3. Auflage, Springer Vieweg, Wiesbaden, 2022 (auch FLACI; nächste Folie)
- A. Schulz, Grundlagen Theoretischer Informatik, Springer Vieweg, Wiesbaden, 2022
- J. Hromkovič, Theoretische Informatik, 5. Auflage , Springer Vieweg, Wiesbaden, 2014
- L. Priese, K. Erk, Theoretische Informatik, 4. Auflage , Springer Vieweg, Wiesbaden, 2018 [Aussagenlogik]
- (weitere Angaben teilweise direkt vor aktuellem Thema)

Software

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- zur Theoretischen Informatik ist parallel zum Aufkommen von Web-Seiten eine bemerkenswerte Menge an Material zu Teilthemen der theoretischen Informatik entstanden
- oft Simulatoren für Formalismen (Automaten, Turing-Maschinen) und ergänzendes Lernmaterial
- gute Nachricht: die Ergebnisse sind nicht veraltet und fachlich meist zumindest gut
- schlechte Nachricht: gibt nicht die eine Sammlung, weit verteilt
- sehr gutes Beispiel: FLACI (Formale Sprachen, abstrakte Automaten, Compiler und Interpreter, https://flaci.com/home/), s. Skript K. Morisse
- Java Formal Languages and Automata Package http://www.jflap.org/
- fehlt (????), Software zur Eingabe eigener Maschinen zur Aufgabenlösung, Prüfung mit Ergebnis („ist falsch, könnte richtig sein“); hierzu wird prototypische Software zur Verfügung gestellt

Was sind Grundlagen

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Theoretische Informatik ist ein aktueller großer weitverzweigter Forschungsbereich
- Beispiele für wesentliche Zweige sind
  - Formale Sprachen und Compilerbau („Beschreibung verarbeitbarer Sprachen“, z. B. Syntax von Programmiersprachen)
  - Entscheidbarkeit (gibt es Grenzen des Programmierbaren)
  - Komplexitätstheorie (u. a. wie schnell mit wieviel Speicherverbrauch)
  - Logik(en) (u. a. wie kann ich Aussagen formalisieren und verknüpfen)
  - Verifikation (u. a. was kann ich wie formal als korrekt beweisen)
- Klassische (veraltete ?) Grundlagenliteratur ist etwa 1985 stehen geblieben, Schwerpunkt auf ersten drei Themen
- aktuelle Modulbeschreibungen zeigen, dass davon abgewichen wird; diese Veranstaltung weicht ebenfalls davon ab

Ist Theoretische Informatik nicht Mathematik

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- im ursprünglichen Kern: ja.
- was soll das dann im Informatik-Studium?
- da Informatik auch einen ingenieurwissenschaftlichen Anteil hat, macht sie sich Ergebnisse der Mathematik zu nutze (und muss sie nicht im Detail verstehen, da der Mathematik vertraut wird)
- d. h. wir lernen wichtige Aussagen und Ansätze kennen, die mathematisch fundiert sind und praktische Relevanz haben (FH)
- d. h. wir führen keine mathematischen Beweise; aber lernen Beweisideen kennen, die uns bei der Nutzung der Kenntnisse helfen (in der Theoretischen Informatik basieren viele Beweise auf konstruktiven Algorithmen, die wir verstehen und umsetzen wollen)

(fast) keine Beweise und weniger Formalismus

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- genannte Literatur hat zwei Punkte gemeinsam: didaktisch sehr gut aufbereitet und nutzen viele formale Notationen
- in jede Notation muss eine Einarbeitung stattfinden
- Frage: muss der ganze Formalismus sein?
- Antwort 1: teilweise ja, um Begriffe zu präzisieren
- Antwort 2: oft nein, da nur Abkürzungen eingeführt werden, die Texte mathematisch exakter und kompakter erscheinen lassen, aber eine Nutzung nebenbei extrem erschweren
- Fazit für diese Veranstaltung: auf unnötige Abkürzungen wird meist verzichtet, (Alphabet, Zustände, Endzustände, Überführungsfunktion, Startzustand) statt z. B. $(\Sigma, S, \delta, s_0, F)$
- klare Bezeichner verbessern Lesbarkeit von Programmen
- eher Bequemlichkeit: statt $z_0 \in$ Zustände oft nur z0 $\in$ Zustände

Die Geschichte (die erzählt werden soll) der Vorlesung

Jede Software-entwickelnde Person ist von dem Wunsch der perfekten Software getrieben. Dazu muss zunächst klar sein, was gemacht werden muss, um dann nachzuweisen, dass die Anforderungen erfüllt sind. Wir stellen aber zunächst fest, dass es so einen Nachweis allgemein nicht geben kann. Das führt zur präziseren Frage, wann so ein Nachweis überhaupt möglich ist. Dazu müssen Programmiersprachen exakt mit ihrer Syntax beschrieben werden, um dann ihnen eine eindeutige Bedeutung (Semantik) zuzuordnen, um dann ein Nachweissystem aufzubauen mit dem wir zumindest von Hand die Korrektheit von Programmen beweisen können.

Wir klären dann einen interessanten Teilbereich für den diese Verifikation nicht nur von Hand sondern vollständig automatisch durchführbar ist.

Abschließend ordnen wir einigen theoretischen Beifang konkreten Themen der Theoretischen Informatik zu.

Inhaltsverzeichnis

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

0. Grundlagen
1. Turing-Maschinen und Entscheidbarkeit
2. Kontextfreie Grammatiken
3. Semantik und Programmverifikation
4. Endliche Automaten und reguläre Ausdrücke
5. Sprachklassen (genauer)
6. Komplexität
7. Auszug weiterer Theorie-Inhalte

Wir stellen aber zunächst fest, dass es so einen Nachweis allgemein nicht geben kann.

Dazu müssen Programmiersprachen exakt mit ihrer Syntax beschrieben werden

um dann ihnen eine eindeutige Bedeutung (Semantik) zuzuordnen, um dann ein Nachweissystem aufzubauen mit dem wir zumindest von Hand die Korrektheit von Programmen beweisen können.

Wir klären dann einen interessanten Teilbereich für den diese Verifikation nicht nur von Hand sondern vollständig automatisch durchführbar ist.

Abschließend ordnen wir einigen theoretischen Beifang konkreten Themen der Theoretischen Informatik zu.

0. Grundlagen

zentrale Inhalte:

- Begriffsklärung Syntax und Semantik
- mathematische Grundlagen
- Axiomensysteme
- Abzählbarkeit und Überabzählbarkeit

Glossar

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- abzählbare Menge
- Axiomensystem
- Cantorsches Diagonalverfahren 1. Version
- Cantorsches Diagonalverfahren 2. Version
- Differenz (Mengen)
- endliche Menge
- gerichteter Graph
- Graph
- Kante
- Knoten
- Komplement
- Mächtigkeit
- Menge
- NatürlicheZahlen
- Schnitt
- Semantik
- Syntax
- unendliche Menge
- ungerichteter Graph
- Vereinigung
- $\in$

Syntax und Semantik (1/2)

- Syntax definiert, wie etwas aussehen soll, damit es zur Menge der betrachteten Elemente gehört
- Syntax legt für Programmiersprache fest, wie Befehle aussehen und welche Befehle zum Programm gehören
- nur wenn Regeln der Syntax eingehalten, dann kann Programm genutzt (kompiliert, interpretiert, …) werden
- Syntax kann z. B. definiert werden über Grammatiken (Backus-Naur-Formen) und Syntaxdiagramme
- Beispiel : 7 + 3 (infix) + 7 3 (präfix) 7 3 + (postfix) ; kann alles die gleiche Semantik haben

Syntax und Semantik (2/2)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Semantik beschreibt die Bedeutung von Elementen
- Grundidee: jedem syntaktisch korrekten Element wird eine Bedeutung zugewiesen
- aus dem Zeichen „+“ wird die aus der Mathematik bekannte Addition
- Semantik klärt für alle Fälle eindeutig, was die Bedeutung ist
- Semantik hängt immer mit von den umgebenden Elementen ab, z. B. kann + auch für die Konkatenation von Strings stehen
- nicht jedem syntaktisch korrekten Element kann eine semantisch sinnvolle Bedeutung zugeordnet werden, z. B. 0 / 0; aber auch die Zuordnung von Fehlern ist teil der Semantik
- präzisere Zusammenhänge, z. B. Compilerbau

Erinnerung: mathematische Grundlagen – Mengen 1/2

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- eine *Menge* ist eine Zusammenfassung von Objekten aus einem vorgegebenen Grundraum, der *Grundmenge*. Ein Objekt kann entweder zur Menge gehören (geschrieben: Objekt $\in$ Menge) oder nicht zur Menge gehören (geschrieben: Objekt $\notin$ Menge).
- Mengen haben keine Reihenfolge, wird in einer Beschreibung ein Element mehrfach angegeben, ist das zu ignorieren
  - {1,2,3} = {3,2,1} = {3,3,1,3,2,1}
- kann die Anzahl der Elemente einer Menge angegeben werden, also ist sie einfach hinschreibbar, heißt die Menge *endlich*, sonst *unendlich*
- In diesen Folien beschreibt NatürlicheZahlen = {0, 1, 2, …} die unendliche Menge der natürlichen Zahlen (einschließlich 0)
- die leere Menge, also Menge ohne Elemente wird {} geschrieben (in Literatur auch $\emptyset$)

Erinnerung: mathematische Grundlagen – Mengen 2/2

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Der *Schnitt* von zwei Mengen enthält alle Elemente, die in beiden Mengen vorkommen, formaler:

  $$\text{Menge1} \cap \text{Menge2} = \{e \mid e \in \text{Menge1 und } e \in \text{Menge2}\}$$

- Die *Vereinigung* von zwei Mengen enthält alle Elemente, die in mindestens einer der beiden Mengen vorkommen, formaler:

  $$\text{Menge1} \cup \text{Menge2} = \{e \mid e \in \text{Menge1 oder } e \in \text{Menge2}\}$$

- Die *Differenz* von zwei Mengen enthält alle Elemente, die in der ersten und nicht in der zweiten Menge vorkommen, formaler:

  $$\text{Menge1} - \text{Menge2} = \{e \mid e \in \text{Menge1 und } e \notin \text{Menge2}\}$$

- Das *Komplement* einer Menge enthält genau die Elemente der Grundmenge, die in der Menge nicht vorkommen, formaler:

  $$\text{Komplement(Menge)} = \{e \mid e \notin \text{Menge und } e \in \text{Grundmenge}\}$$

- weitere Grundlagen (zu Mengen) im Skript von K. Morisse [Mor]

Axiomensysteme (1/2)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Um Aussagen beweisen zu können, müssen zunächst die Fundamente (Grundlagen, Axiome) definiert werden, die als wahr angenommen werden

Beispiel: Peano-Axiome für natürliche Zahlen

1. 0 ist eine natürliche Zahl.
2. Zu jeder natürlichen Zahl n gibt es eine natürliche Zahl n', die Nachfolger von n ist.
3. Zwei verschiedene natürliche Zahlen haben verschiedene Nachfolger.
4. 0 ist nicht Nachfolger einer natürlichen Zahl.
5. Enthält eine Menge natürlicher Zahlen die 0 und mit jeder natürlichen Zahl auch deren Nachfolger, so enthält sie alle natürlichen Zahlen.

https://www.spektrum.de/lexikon/mathematik/peano-axiome/7735

Operatoren wie + werden auf der Basis der Axiome definiert und dann mit Hilfe der Axiome z. B. bewiesen a + b = b + a

Axiomensysteme (2/2)

Zermelo-Fraenkel-Mengenlehre beschreibt axiomatische Mengenlehre und ist Grundlage fast aller Mathematik-Ansätze; Auswahl der 9 Axiome:

- Axiom der leeren Menge: Es gibt mindestens eine Menge, die keine Elemente enthält.
- Extensionalitätsaxiom: Wenn zwei Mengen die gleichen Elemente enthalten, sind sie gleich. (folgt: leere Menge ist eindeutig)
- Paarmengenaxiom: Für zwei Mengen gibt es genau eine Menge, diese zwei Mengen enthält (aus Ext: {x,y} = {y,x})
- Vereinigungsaxiom: Zu jeder Menge gibt es eine Menge, die alle Elemente der Elemente der Grundmenge enthält
- Unendlichkeitsaxiom: Es gibt eine nicht-leere Menge, die die leere Menge enthält
- Potenzmengenaxiom: Zu jeder Menge existiert eine andere Menge (Potenzmenge), die alle Teilmengen der Ursprungsmenge enthält

u. a. https://www.henked.de/begriffe/menge.htm#ZFC

Abzählbarkeit (1/4)

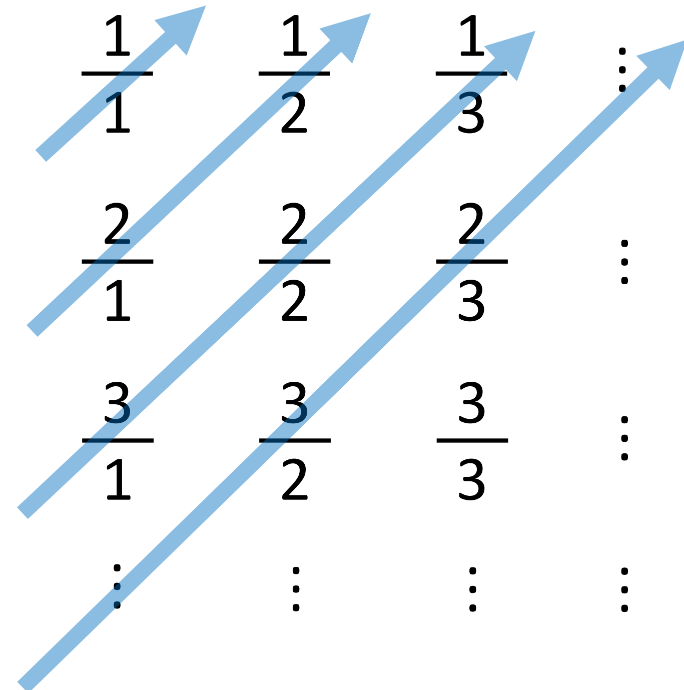
HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Auch bei unendlichen Mengen macht es Sinn ihre Größe zu vergleichen, genauer ins Verhältnis zu setzen
- Definition: Eine Menge heißt *abzählbar*, wenn sie entweder endlich ist oder es eine surjektive Funktion NatürlicheZahlen -> Menge gibt, sonst heißt die Menge überabzählbar
- *surjektiv* bedeutet, dass es für jedes Element der Menge eine natürliche Zahl gibt, die die Funktion auf dieses Element abbildet (nicht unbedingt bijektiv; es dürfen auch mehrere natürliche Zahlen auf das gleiche Element der Menge abgebildet werden)
  - Die Menge der natürlichen Zahlen ist abzählbar
    f: NatürlicheZahlen -> NatürlicheZahlen mit $f(x) = x$
  - Die Menge der natürlichen Zahlen erweitert um das Element blubb ist abzählbar
    f: NatürlicheZahlen -> NatürlicheZahlen $\cup$ {blubb}
    mit $f(0)$ = blubb und $f(x) = x-1$ für $x > 0$

Abzählbarkeit (2/4) - abzählbar

- Sind Menge1 und Menge2 abzählbar, so haben sie die gleiche *Mächtigkeit*
- Die Menge der ganzen Zahlen ist abzählbar mit
  f(x) = x div 2, wenn x gerade
  f(x) = -((x+1) div 2), wenn x ungerade, anschaulich 0,-1,1,-2,2, ...
- Satz: Die Menge der rationalen Zahlen (Brüche) ist (mit Hilfe des 1. Cantorschen Diagonalverfahrens) abzählbar

$$\frac{1}{1}\ \frac{2}{1}\ \frac{1}{2}\ \frac{3}{1}\ \frac{2}{2}\ \frac{1}{3}\ \frac{4}{1} \ldots$$



Abzählbarkeit (3/4) – überabzählbar (1/2)

- Satz: Die reellen Zahlen sind überabzählbar
- Annahme: Die reellen Zahlen sind abzählbar mit einer Funktion f, dann kann man die reellen Zahlen nacheinander hinschreiben
- konstruiere jetzt eine neue Zahl z, die sich an der n-ten Nachkommastelle von der Zahl f(n) unterscheidet, eine Möglichkeit im Beispiel ist 0,9802….
- da nach Annahme die reellen Zahlen abzählbar sind, muss diese konstruierte Zahl an einer Stelle p in der Tabelle auftauchen, also f(p) = z
- da aber sich die Zahl f(p) an der p-ten Stelle von z unterscheidet, gilt $f(p) \neq z$, Widerspruch, damit Annahme nicht zutreffend
- (2. Cantorsche Diagonalverfahren)

x	f(x)
0	0,000000000…
1	0,800000000…
2	0.333333333…
3	0,314592653…
…	….

Abzählbarkeit (4/4) – überabzählbar (2/2)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

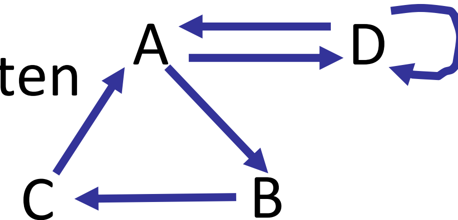
- Sei M = {m0, m1, ...} eine abzählbar unendliche Menge, dann ist die Potenzmenge Pot(M) (auch 2^M geschrieben) überabzählbar
- Beweisidee: Jedes Element P∈Pot(M) lässt sich durch eine Folge von 0 und 1 darstellen; eine 0 an der i-ten Position bedeutet mi∉P, eine 1 mi∈P
- Annahme: Pot(M) ist abzählbar mit einer Funktion f, dann kann man die Elemente von Pot(M) nacheinander hinschreiben.
- Konstruiere jetzt Eintrag inv, der an der i-ten Stelle den Eintrag von f(i) invertiert, dann ist inv auch eine Darstellung eines Elements aus Pot(M), damit muss es eine Position pos geben mit f(pos) = inv, da aber nach Konstruktion von inv sich der Wert an der Stelle pos von f(pos) unterscheidet, folgt daraus f(pos) ≠ inv, also gilt die Annahme nicht

x	f(x)
0	1000000...
1	0110011...
2	1101000...
3	0010101...
...	...
inv	0011...

Graph

- Definition: Ein *Graph* wird definiert durch eine endliche Menge von Knoten und eine Menge von Kanten, die aus Paaren von Knoten bestehen, genauer

  Graph = (Knoten, Kanten), Kanten $\subseteq$ Knoten $\times$ Knoten



- Graphen können visualisiert werden

  Graph = ({A,B,C,D}, {(A,B), (B,C), (C,A), (A,D), (D,A), (D,D)})

- Ein Graph heißt *ungerichtet*, wenn für jede Kante

  (knoten1,knoten2) $\in$ Kanten auch (knoten2,knoten1) $\in$ Kanten gilt, sonst heißt er *gerichtet*.

- der Beispielgraph ist gerichtet
- Anmerkungen: Knoten = vertices, Kanten = edges, Variante zur Definition eines ungerichteten Graphen,

  Kanten $\subseteq$ {{k1,k2} : k1, k2 $\in$ Knoten}

Video

1. Turing-Maschinen und Entscheidbarkeit

zentrale Inhalte:

- Formalisierung des Programmbegriffs
- Turing-Maschinen und Sprachakzeptanz
- Aufzählbarkeit
- Entscheidbarkeit

Glossar

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- akzeptierbar
- akzeptierte Sprache
- Alphabet
- Alphabet*
- Alphabet+
- aufzählbar
- berechenbar
- Berechnung
- Buchstaben
- Busy Beaver
- charakteristische Funktion
- Church‘sche These
- divergieren
- entscheidbar
- Funktionskomposition
- Gödelisierung
- hängen
- halten
- Halteproblemsprache
- Konfiguration
- Konkatenation
- leeres Wort
- Postsches Korespondenzproblem
- Rechnung
- Reduktion
- Register-Maschine
- rekursiv aufzählbar
- semi-entscheidbar
- Sprache
- Startkonfiguration
- terminierende Berechnung
- Turing-Maschine
- turing-berechenbar
- Überführungsfunktion
- universelle Turing-Maschine
- WHILE-Programm
- Wort
- Wortlänge
- Zeichen
- Zeichenanzahl
- Zustandsdiagramm
- Σ
- $\circ$
- *

Vorgehen

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

Wir stellen aber zunächst fest, dass es so einen Nachweis allgemein nicht geben kann.

- Definition formale Sprachen, z. B. zur Beschreibung von Programmiersprachen (Alphabet, Wort, Sprache)
- Präzisierung des Programmbegriffs mit Turing-Maschinen; Ziel Turing-Maschine (und Simulator dafür) selbst programmieren
- wichtiges Hilfsmittel: Merken des aktuell erreichbaren vollständigen Zustands einer Turing-Maschine in Ausführung (Konfiguration)
- Akzeptanz von Wörtern, Aufgabe: Entscheidbarkeit von Sprachen
- Halteproblem, Abzählbarkeit
- Nachweis von Unentscheidbarkeit durch Reduktion

Wort (1/2)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Definition: Ein *Alphabet* ist eine beliebige endliche, nicht-leere Menge. Die Elemente des Alphabets werden *Zeichen* (oder auch *Buchstaben*) genannt. Als Kurzschreibweise wird gerne Σ (großes Sigma) verwandt
  - Alphabet1 = {a,b,c,d,e} Alphabet2={if, while, for} Alphabet3 = $\{\theta, \Leftrightarrow, \vee, \forall\}$
- Ein *Wort* über einem Alphabet ist eine endliche, damit eventuell auch leere Folge aus Zeichen aus dem Alphabet. Seien a1,...,an Zeichen aus dem Alphabet, dann kann ein Wort als (a1,...,an) geschrieben werden. Die übliche Kurzform ist a1...an.
- Die leere Folge hat kein Zeichen, um sie zu erkennen wird sie als ε (kleines Epsilon) geschrieben und heißt *leeres Wort* (alternativ: λ).
  - Worte über Alphabet1: a, aa, abcde, ee, ε
  - Worte über Alphabet2: if, ifif, whileforwhile, ε
  - Worte über Alphabet3: $\theta\Leftrightarrow\theta$, , $\Leftrightarrow\Leftrightarrow$, $\vee\forall\vee\forall$, ε

Wort (2/2)

- *Alphabet** bezeichnet die (unendliche) Menge von Wörtern über dem Alphabet. Mit *Alphabet$^+$* wird die Menge der Worte über dem Alphabet beschrieben, die mindestens ein Zeichen hat, d.h. Alphabet$^+$ = Alphabet* - {ε}
  - Alphabet1* = {ε, a, b, c, d, e, aa, ab, ac, ...}
  - Alphabet1$^+$ = {a, b, c, d, e, aa, ab, ac, ...}
- Für ein Wort w bezeichnet |w| die *Länge des Wortes*, also die Anzahl der Zeichen von w. Für ein Zeichen a aus dem Alphabet bezeichnet *anzahl(a,w)* die Anzahl der Vorkommen von a in w.
- formaler: anzahl: Alphabet $\times$ Alphabet* -> NatürlicheZahlen
  - $|\varepsilon| = 0$ |a| = 1 |aaa| = 3 | whileforwhile | = 3
  - anzahl(a,ε) = 0 anzahl(a, a) = 1 anzahl(a, abba) = 2

Konkatenation

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- konkatenieren $\approx$ aneinanderhängen
- Definition: Seien wort1 und wort2 zwei Worte über einem Alphabet, dann entsteht die *Konkatenation*, geschrieben $\circ$, der Wörter durch ein Anhängen von wort2 an wort1 , genauer

  $\circ$: Alphabet* $\times$ Alphabet* -> Alphabet*
- wort1= a1…an wort2 = b1…bn dann wort1 $\circ$ wort2 = a1…anb1…bn
- ab $\circ$ ba = abba a $\circ$ bba = abba a $\circ$ ε = ε $\circ$ a = a
- Randnotiz: Da jedes Zeichen eines Alphabets auch ein Wort der Länge 1 ist, besteht jedes Wort aus der Konkatenation seiner Zeichen
- Wenn es klar ist, dass es um eine Konkatenation geht, wird das Konkatenationszeichen einfach weggelassen , also statt

  wort1 $\circ$ wort2 dann wort1wort2

weitere Operatoren auf Wörtern

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Für ein Wort lässt sich die *i-te Iteration* induktiv wie folgt definieren:

  $w^0 = \varepsilon$

  $w^{n+1} = ww^n$

  - $\varepsilon^0 = \varepsilon$ $\varepsilon^{42} = \varepsilon$ $(ab)^0 = \varepsilon$ $(ab)^3 = ababab$

- Die *Umkehrung* (auch Reverse) eines Wortes lässt sich induktiv über den Aufbau des Wortes definieren
- ist Wort = ε ist $w^R = \varepsilon^R = \varepsilon$
- ist Wort = va, und a ein Zeichen des Alphabets, dann ist $(va)^R = a(v^R)$
  - $(abba)^R = abba$ $(bba)^R = abb$

Sprache

- Definition: Eine *Sprache* über einem Alphabet ist eine Teilmenge von Alphabet*
  - $\{\varepsilon\}$, $\{\varepsilon, a, aa\}$, $\{a, aa, bbb\}$, $\{a^n: n \in$ NatürlicheZahlen$\}$, Alphabet1* sind Beispielsprachen über Alphabet1
- Die Operatoren auf Wörtern werden wie folgt auf Sprachen erweitert:

  Sprache1 $\circ$ Sprache2 = $\{uv \mid u \in$ Sprache1, $v \in$ Sprache2$\}$, kürzer geschrieben als Sprache1Sprache2

  $\{\varepsilon\} \circ \{\varepsilon, a, aa\} = \{\varepsilon, a, aa\}$ $\{\} \circ \{\varepsilon, a, aa\} = \{\}$

  $\{a^n: n \in$ NatürlicheZahlen$\} \circ \{a\} = \{a^n: n \in$ NatürlicheZahlen, $n > 0\}$

  Sprache$^0 = \{\varepsilon\}$ Sprache^{n+1} = SpracheSprache^n , d.h. auch $\{\}^0 = \{\varepsilon\}$

  Der *Kleene-Stern-Abschluss* eine Sprache ist definiert als Vereinigung aller Iterationen der Sprache

  Sprache* = Sprache$^0 \cup \ldots \cup$ Sprache^n, $n \in$ NatürlicheZahlen (unendliche Vereinigung)

Abzählbarkeit

Video

- Satz: Für jedes Alphabet ist Alphabet* abzählbar.
- Konstruktion: Sei das Alphabet = {a1, ..., an}, die Wörter werden schrittweise der Länge nach durchgegangen und erhalten so ihre Position: ε , a1, a2, ..., an, a1a1, a1a2, ..., a1an, ..., ana1, ..., anan, a1a1a1, ...
- Beispiel: Für das Alphabet {a,b,c} beginnt die Funktion wie folgt:
  f(0) = ε f(1) = a f(2) = b f(3) = c f(4) = aa f(5) = ab f(6) = ac
  f(7) = ba f(12) = cc f(13) = aaa f(14) = aab ...
- Folgerung: Jede Sprache $\subseteq$ Alphabet* ist abzählbar (Elemente, die nicht dazu gehören auslassen)
- Folgerung: Da Sprache $\in$ Pot(Alphabet*), gilt die Menge aller Sprachen ist nicht abzählbar

Gödelisierung (nach Kurt Gödel)

- Satz: Für ein gegebenes Alphabet kann jedes Wort in eine eindeutige natürliche Zahl verwandelt werden, d. h.
- gödel: Alphabet* -> NatürlicheZahlen mit aus
  gödel(w) = gödel(v) folgt w = v
- Erinnerung Mathematik: für jede natürliche Zahl gibt es eine eindeutige Primzahlzerlegung ($42 = 2*3*7$, $36 = 2^2*3^2$, $37=37^1$)
- Verfahren: Nummeriere Elemente des Alphabets durch {a1, a2, ..., an}, nutze erste n Primzahlen 2, 3, 5, 7, 11, 13, 17,...

  Betrachte für ein Wort für jedes seiner Zeichen an der Position i die Nummer des Zeichens ni aus der Nummerierung, berechne dann gödel(z1...zk) = $2^{n1}*3^{n2}*5^{n3}*\ldots*pk^{nk}$ pk ist k-te Primzahl

  statt beliebigen Zahlen oft auch nur {0, 1} das Ziel; dabei 0 um Zahlenenden zu markieren, 42 = 01111011

Gödelisierung - Beispiel

- Alphabet = {a, b, c, + , *, (,)}
- Position: 1 2 3 4 5 6 7
- Wort w a * b
- Positionen 1 5 2
- gödel(w) = $2^1 * 3^5 * 5^2$ = 12150
- Wort w (a * (a + b))
- Positionen 6 1 5 6 1 4 2 7 7
- gödel(w) = $2^6 * 3^1 * 5^5 * 7^6 * 11^1 * 13^4 * 17^2 * 19^7 * 23^7$
  = 1950623654329909000993827637741380000
- nicht alle Zahlen gehören zu Elementen aus Alphabet* 256 = 2^8
- Anmerkung: Das Rechnen mit sehr großen ganzen Zahlen ist generell kein Problem, benötigt aber eigene Klassen, in Java z. B. BigInteger

Programmierbarkeit

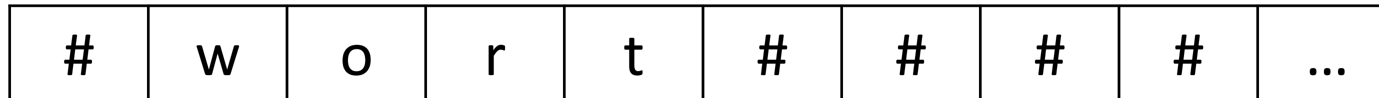
- Um die Frage zu klären, „was programmiert werden kann bzw. was nicht“, ist der Begriff „programmierbar“ zu klären
- dies wird durch sogenannte Turing-Maschinen erfolgen
- A. Turing, On computable numbers with an application to the Entscheidungsproblem”. In: Proc. London Math. Soc. 2.42 (1936), S. 230–265
- dazu gehört die Church‘sche These:

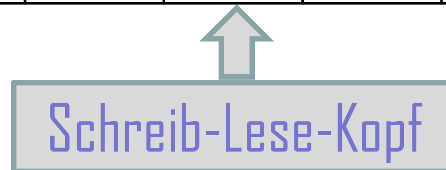
„Nach der Church-Turing-These wird der intuitive Begriff des Berechenbaren durch die formale, mathematisch exakte Definition des Turing-Maschinen-Berechenbaren exakt wiedergegeben. Da der Begriff des Berechenbaren, der nur intuitiv fassbar ist, kein exaktes mathematisches Konzept ist, lässt sich diese Aussage nicht beweisen. “ [Mor]

Rechenmodell einer Turingmaschine (1/2)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

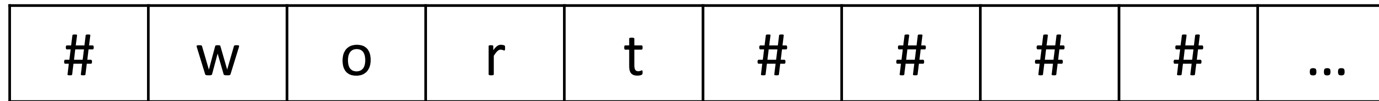
- es gibt ein unendliches Band von Speicherzellen, die jeweils ein Zeichen aufnehmen können, leere Zellen werden mit einem speziellen Zeichen gekennzeichnet. Am Anfang steht ein beliebiges Wort in den Anfangszellen des Bandes





- Es gibt einen Schreib-Lese-Kopf, der immer auf genau einer Zelle steht. In einem Arbeitsschritt wird der Inhalt der Zelle gelesen und dann eine der drei folgenden Aktionen ausgeführt, schreibe ein (nicht notwendigerweise neues) Zeichen und
  - gehe nach links
  - gehe nach rechts
  - stoppe (beende) die Ausführung

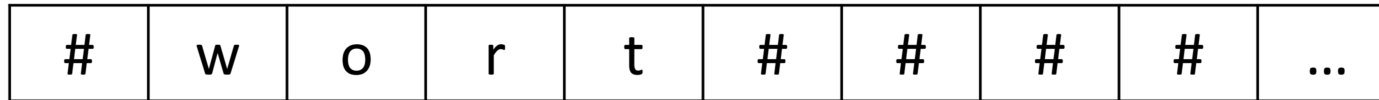
Rechenmodell einer Turingmaschine (2/2)



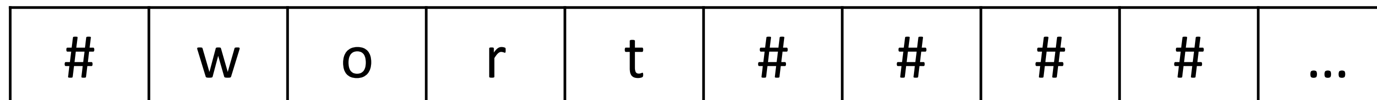
- Um nächsten Schritt zu berechnen hat die Turingmaschine einen aktuellen Zustand, in dem das Zeichen gelesen und dann eine der Aktionen ausgeführt wird, definiert in einer Überführungsfunktion
- Rahmenbedingungen:
  - es gibt einen Startzustand
  - der Schreib-Lesekopf steht am Anfang eine Zelle hinter dem Wort

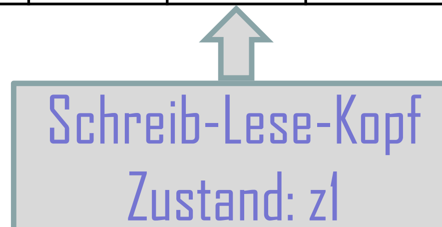
Hinweis: In der Literatur einige vergleichbare Varianten in der Definition

Überführungsfunktion genauer



- was muss beschrieben werden, um einen Schritt zu formalisieren?
- wo ist die Maschine? im Zustand Start, liest ein #
- was soll passieren? schreibe #, gehe in Zustand z1 und nach links





Turing-Maschine

Definition: Eine *Turing-Maschine* ist ein Tupel (Zustände, Alphabet, *Überführungsfunktion*, Start) mit

- Zustände ist eine endliche nicht leere Menge
- # $\in$ Alphabet (Leerzeichen, blank, b̸)
- Start $\in$ Zustände
- Überführungsfunktion über:
  Zustände $\times$ Alphabet -> Zustände $\times$ Alphabet $\times$ {LINKS, RECHTS, STOPP}

Eine *Konfiguration* einer Turing-Maschine beschreibt vollständig die aktuelle Situation der Maschine, sie ist gegeben durch (z, w1 a w2) mit
z $\in$ Zustände, w1 $\in$ Alphabet* dem Bandinhalt links des Kopfes,
a $\in$ Alphabet, dem Zeichen unter dem Lesekopf,
w2 $\in$ Alphabet* $\circ$ (Alphabet – {#}) $\cup$ {ε} dem Bandinhalt rechts des Kopfes (ohne das unendliche leere Band)

Konfiguration genauer

statt:

#	w	o	r	t	#	#	#	#	...

Schreib-Lese-Kopf
Zustand: Start

(Start, #wort # ε)   Kurzschreibweise: (Start, #wort<u>#</u>) , Unterstrich zeigt Position des Schreib-Lese-Kopfes

statt:

#	w	o	r	t	#	#	#	#	...

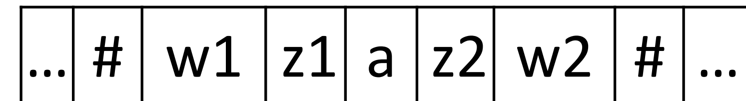
Schreib-Lese-Kopf
Zustand: z1

(z1, #wor t ε)   kurz (z1, #wor<u>t</u>)

Arbeitsweise einer Turing-Maschine

Gegeben sei eine Turing-Maschine (Zustände, Alphabet, Überführungsfunktion, Start) zu $\in$ Zustände, w1,w2 $\in$ Alphabet*, z1 $\in$ Alphabet, z2 $\in$ Alphabet $\cup$ {ε} in einer Konfiguration K1 = (zu, w1z1 a z2w2) und sei über(zu, a) = (zu2, b, X), dann berechnet sich die Nachfolgekonfiguration K2, geschrieben

K1 -> K2 wie folgt



- Sei X = STOPP : K2 =(zu2, w1z1 b z2w2)
- Sei X = LINKS : K2 = (zu2, w1 z1 bz2w2)
- Sei X = RECHTS : K2 = (zu2, w1z1b z2 w2), wenn z2 $\neq \varepsilon$

  K2 = (zu2, w1z1b # ε), z2 = ε

Gegeben sei ein Wort w ohne Leerzeichen, w $\in$ (Alphabet-{#})*, dann heißt K0 = (Start, #w # ε) (kurz: (Start, #w<u>#</u>)) *Startkonfiguration*; das Leerzeichen # am Anfang markiert den Wortanfang

Ergebnis einer Turing-Maschine

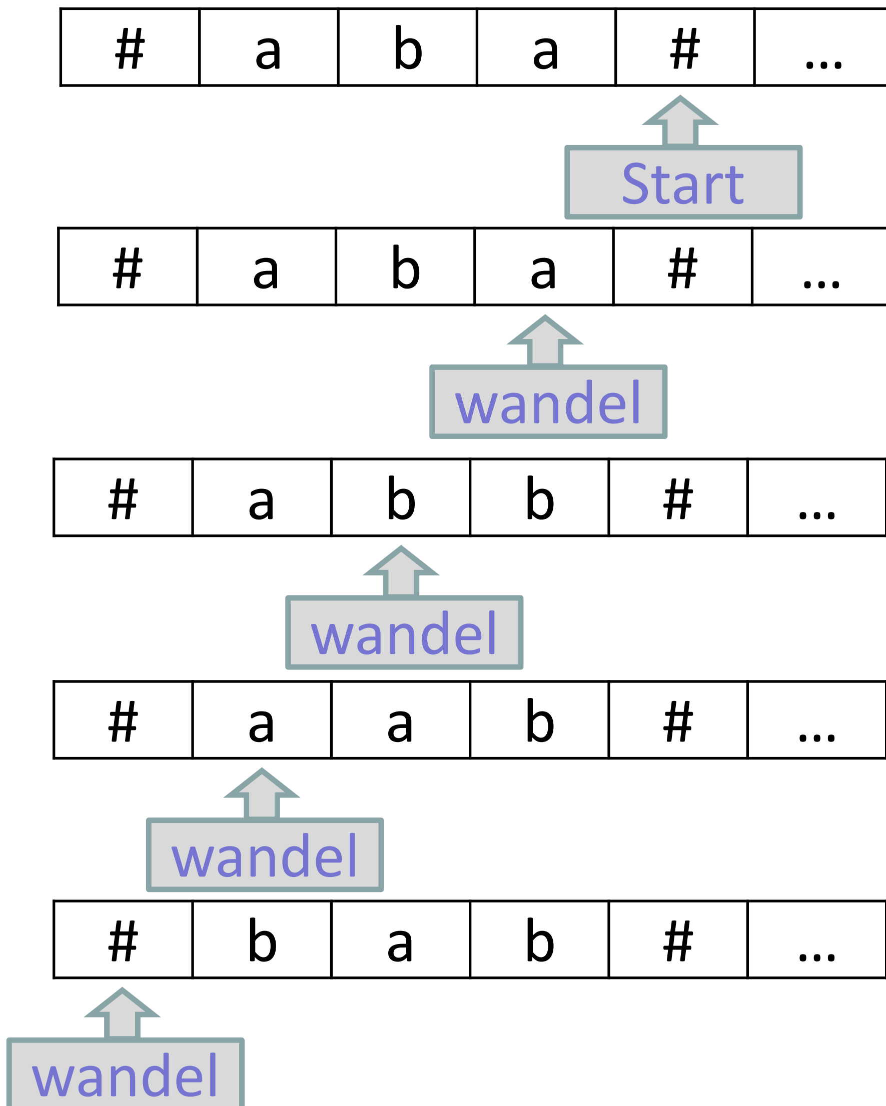
- Definition: Die Folge von Konfigurationen K0 -> K1 -> … -> Kn wird *Rechnung* (bzw. *Berechnung*) einer Turing-Maschine genannt und abkürzend K0 ->* Kn geschrieben, ergänzend gilt K0 ->* K0
- Eine Turing-Maschine *hält* in einer Konfiguration , wenn bei dem letzten Ergebnis der genutzten Überführungsfunktion ein STOPP (Stopp-Symbol) steht, die Berechnung heißt dann *terminierende Berechnung*; wird auch als „die Turing-Maschine hält angesetzt auf die Eingabe an" oder „die Turing-Maschine hält in der Konfiguration" beschrieben
- Gibt es eine unendliche Berechnung, *divergiert* die Turing-Maschine
- Eine Turing-Maschine *hängt* in einer Konfiguration (zu, w1 z w2), wenn w1 = ε und über(zu, z) = (. , . , LINKS) (Versuch über den linken Rand hinaus zu laufen, . für beliebig)
- Die *akzeptierte Sprache einer Turing-Maschine* TM ist definiert als Lang(TM) = { w $\in$ (Alphabet-{#})* | $\exists$K (Start, #w$\underline{\#}$) ->* K und TM hält in K}

Beispiel 1: Turing-Maschine (1/2)

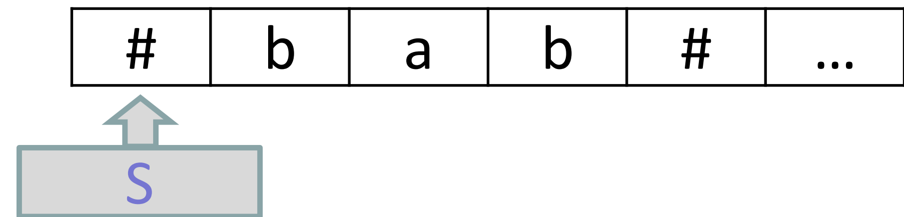
- Hinweis: nicht immer besteht Interesse an der akzeptierten Sprache; es kann auch um zu erreichende Band-Inhalte gehen
- Wandle alle a in b und alle b in a um:

  Wandere kontinuierlich nach links und ändere dabei das gelesene Zeichen um, falls dann # erreicht wird, ist der Anfang gefunden, ans Ende des Wortes gelaufen und die Arbeit beendet
- verkürzend L, R, S für LINKS, RECHTS, STOPP
- TM = ({Start, wandel, S}, {a, b, #}, über, Start) mit
- über(Start, #) = (wandel, #, L)
- über(wandel, a) = (wandel, b, L) // ändern und weiterlaufen
- über(wandel, b) = (wandel, a, L)
- über(wandel, #) = (S, #, S) // Zustand markiert Ende
- über(x,y) muss nicht total definiert sein (geht eventuell nicht weiter)

Beispiel 1: Turing-Maschine (2/2)



über(Start, #) = (wandel, #, L)
über(wandel, a) = (wandel, b, L)
über(wandel, b) = (wandel, a, L)
über(wandel, #) = (S, #, S)



es wäre formal zu zeigen, dass die Maschine für jede Eingabe das gewünschte Verhalten hat, gezeigter Ablauf entspricht nur Test

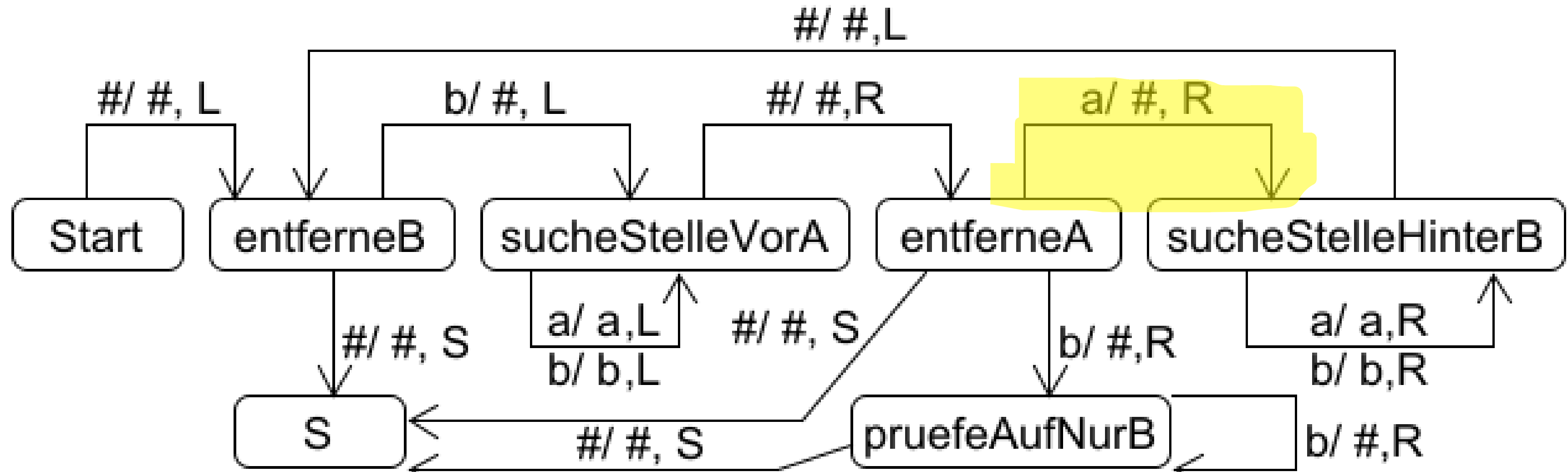
Beipiel 2: Turing-Maschine (1/3)

- konstruiere Turing-Maschine TM mit Lang(TM) = $\{a^n b^{n+m} \mid n, m \geq 0\}$
- wie immer am Anfang: überlege einen Algorithmus
- neue Zustände, die für bestimmte Situationen stehen
- neue Zeichen, die sich Informationen auf dem Band merken
- nehme b, streiche es, lauf ganz nach links und streiche a, laufe ganz nach rechts streiche b; wenn links am Ende kein a (und Rest nur b), dann gehe in Halte-Zustand
- TM = ({Start, entferneB, sucheStelleVorA, entferneA, sucheStelleHinterB, pruefeAufNurB, S}, {a, b, #}, über, Start) mit
- über(Start, #) = (entferneB, #, L)
- über(entferneB, #) = (S, #, S) // gleiche viele a und b (auch 0)
- über(entferneB, b) = (sucheStelleVorA, #, L)
- über(sucheStelleVorA, x) = (sucheStelleVorA, x, L), $x \in \{a, b\}$

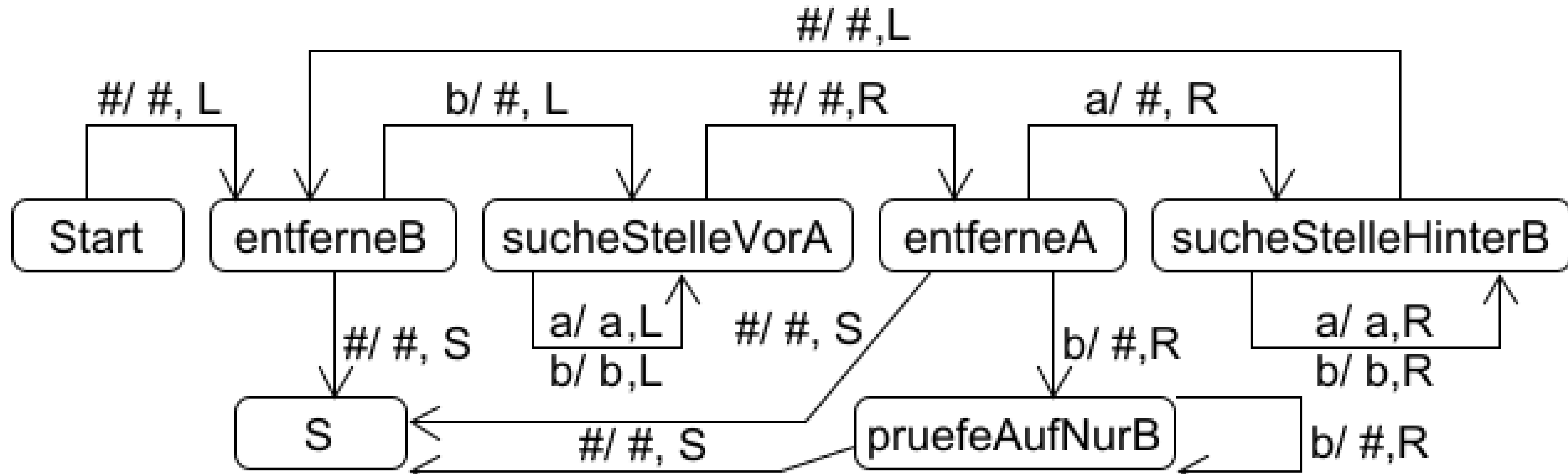
Beispiel 2: Turing-Maschine (2/3) - Zustandsdiagramm

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- über(sucheStelleVorA, #) = (entferneA, #, R)
- über(entferneA, #) = (S, #, S) // passiert, wenn ein b mehr als a
- über(entferneA, b) = (pruefeAufNurB, #, R)
- über(entferneA, a) = (sucheStelleHinterB, #, R)
- über(sucheStelleHinterB, x) = (sucheStelleHinterB, x, R), $x \in \{a, b\}$
- über(sucheStelleHinterB, #) = (entferneB, #, L)
- über(pruefeAufNurB, b) = (pruefeAufNurB, #, R)
- über(pruefeAufNurB, #) = (S, #, S)



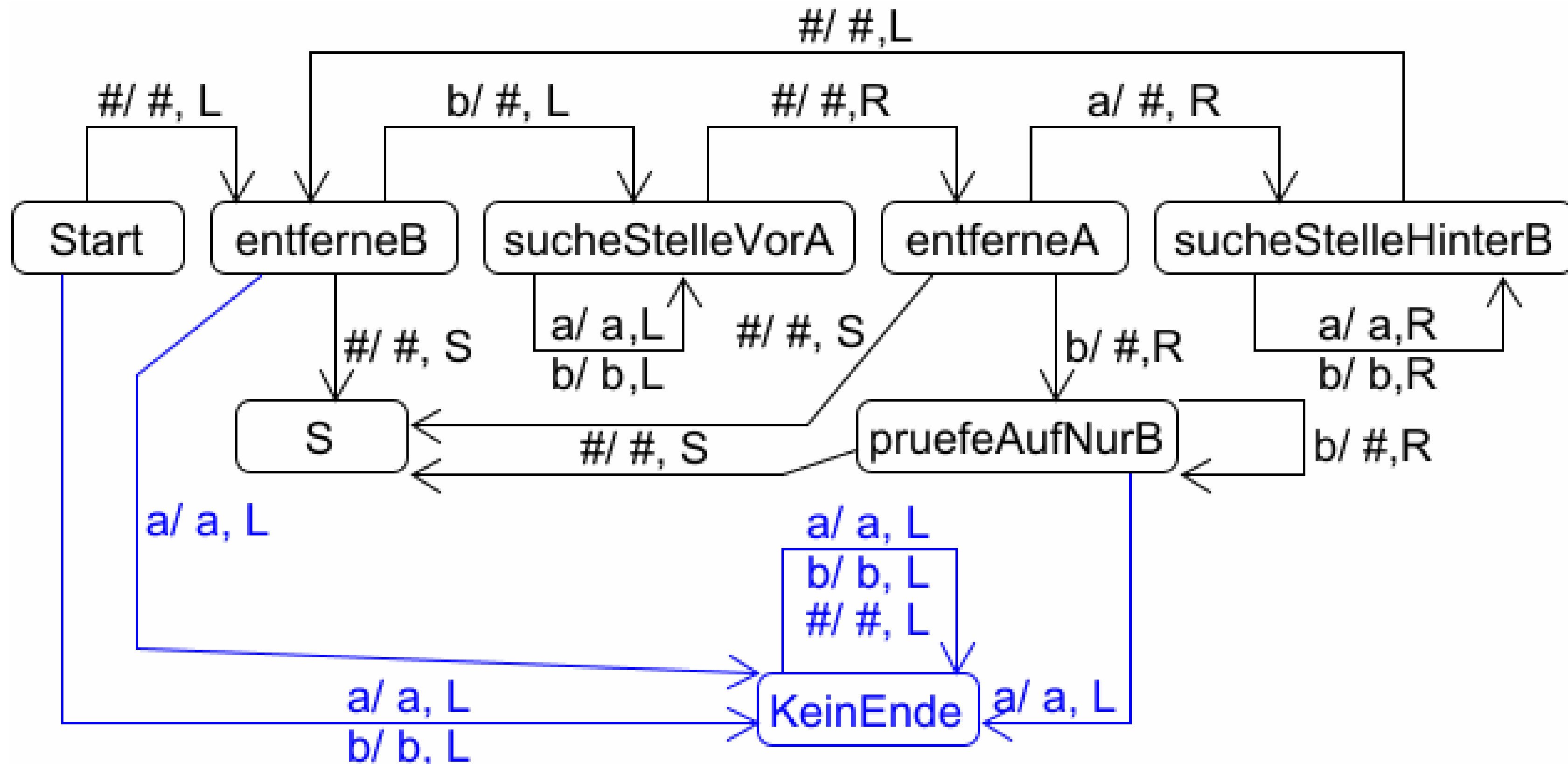
Beispiel 2: Turing-Maschine (3/3)



(Start, #aabbb#) -> (entferneB, #aabbb) -> (sucheStelleVorA, #aabb) -> (sucheStelleVorA, #aabb) -> (sucheStelleVorA, #aabb) -> (sucheStelleVorA, #aabb) -> (sucheStelleVorA, #aabb) -> (entferneA, #aabb) -> (sucheHinterB, ##abb) -> (sucheHinterB, ##abb) -> (sucheHinterB, ##abb) -> (sucheHinterB, ##abb#) -> (entferneB, ##abb) -> (sucheStelleVorA, ##ab) -> (sucheStelleVorA, ##ab) -> (sucheStelleVorA, ##ab) -> (entferneA, ##ab) -> (sucheHinterB, ###b) -> (sucheHinterB, ###b#) -> (entferneB, ###b) -> (sucheStelleVorA, ###) -> (entferneA, ####) -> (S, ####)

Vervollständigung der Überführungsfunktion

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES



- formal wird so Endlosablauf bei Nichterfolg spezifiziert
- Kanten in S fehlen (Variante mit S als Spezialzustand „halt“ s. [Mor])
- bleibt zu zeigen: Maschine akzeptiert nur Worte der Sprache, jedes Wort der Sprache wird akzeptiert

Beispiel 3: Turing- Maschine (1/4)

- Auf dem Band stehen am Anfang n-Striche (*unäre Darstellung* der Zahl n) , am Ende sollen 2n-Striche stehen
- Idee: ein neues Trennsymbol (%) zwischen Ein- und Ausgabe setzen, ersten nicht bearbeiteten Strich links suchen, diesen markieren (neues Zeichen ~), nach rechts gehen und zwei Striche anfügen, dann nach Links gehen und ersten nicht bearbeiteten Strich suchen; wenn nicht vorhanden, markierte Striche und Trennsymbol löschen und an das Ende des Ergebnisses laufen
- TM = ({Start, sucheStrich, entferneEingabe, maleStriche, zweiterStrich, zumStart, S}, {I, ~, %, #}, über, Start) mit
- über(Start, #) = (sucheStrich, % , L)   // Trennzeichen setzen
- über(sucheStrich, #) = (entferneEingabe, #, R)
- über(sucheStrich, ~) = (sucheStrich, ~, L)
- über(sucheStrich, I) = (maleStriche, ~, R) // Strich verarbeitet

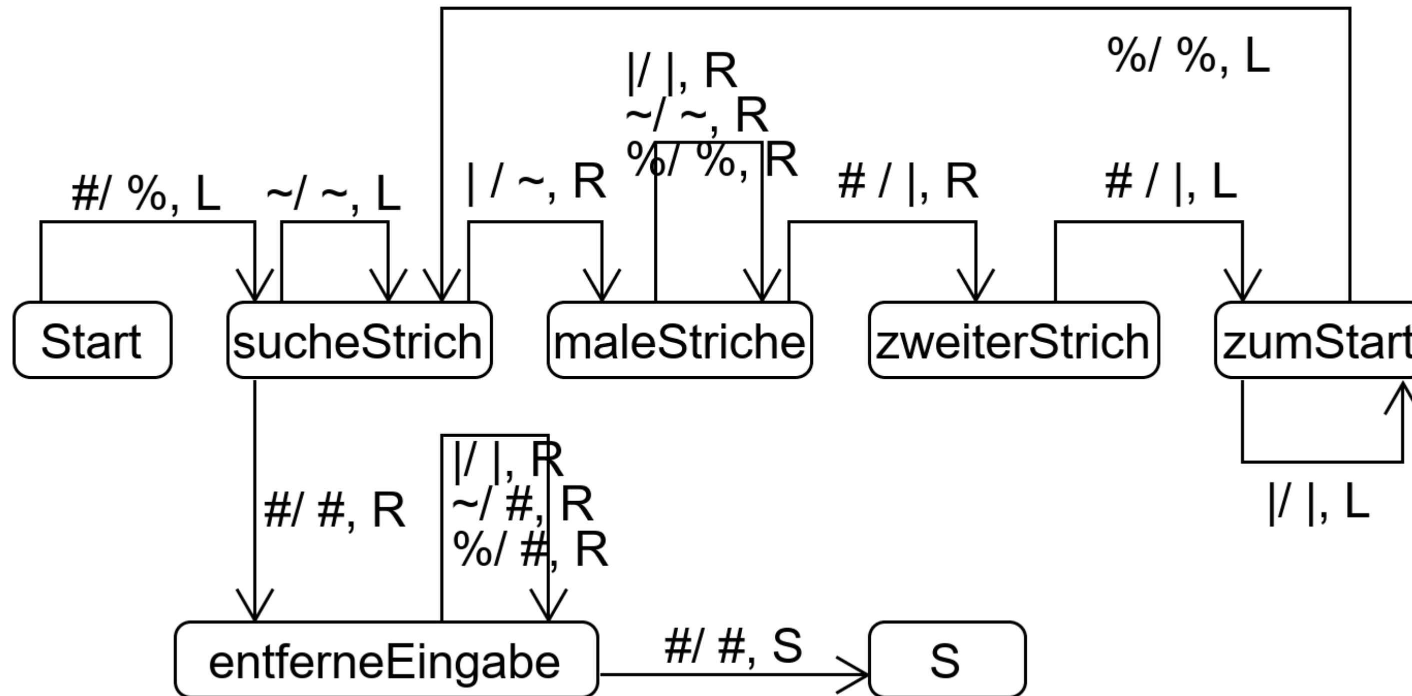
Beispiel 3: Turing- Maschine (2/4)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

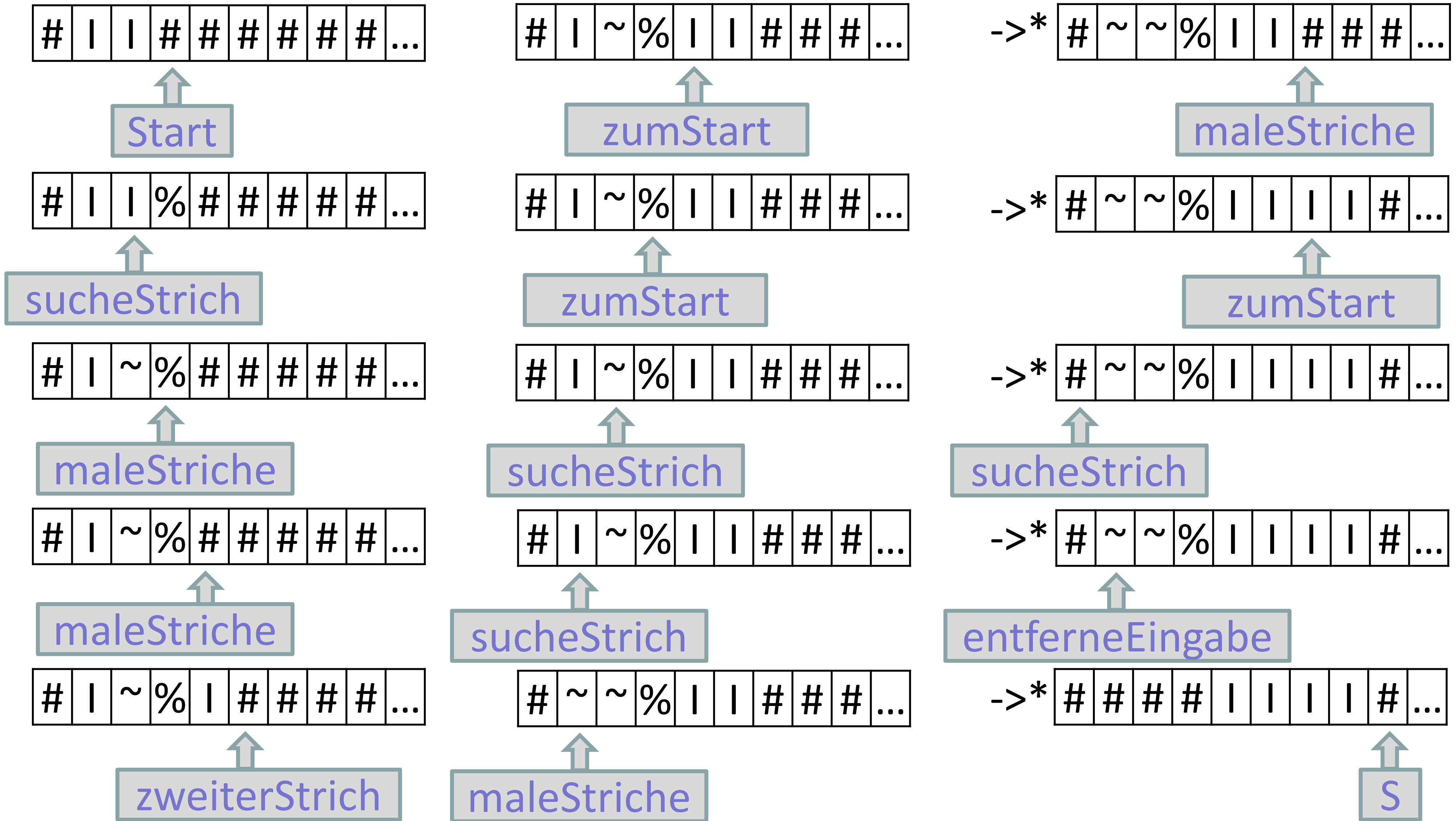
- über(maleStriche, ~) = (maleStriche, ~, R)
- über(maleStriche, %) = (maleStriche, %, R)
- über(maleStriche, I) = (maleStriche, I, R)
- über(maleStriche, #) = (zweiterStrich, I, R)
- über(zweiterStrich, #) = (zumStart, I, L)
- über(zumStart, I) = (zumStart, I, L)
- über(zumStart, %) = (sucheStrich, % ,L)
- über(entferneEingabe, ~) = (entferneEingabe, #, R)
- über(entferneEingabe, %) = (entferneEingabe, #, R)
- über(entferneEingabe, I) = (entferneEingabe, I, R)
- über(entferneEingabe, #) = (S, #, S)

Beispiel 3: Turing- Maschine (3/4)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES



Beispiel 3: Turing- Maschine (4/4)

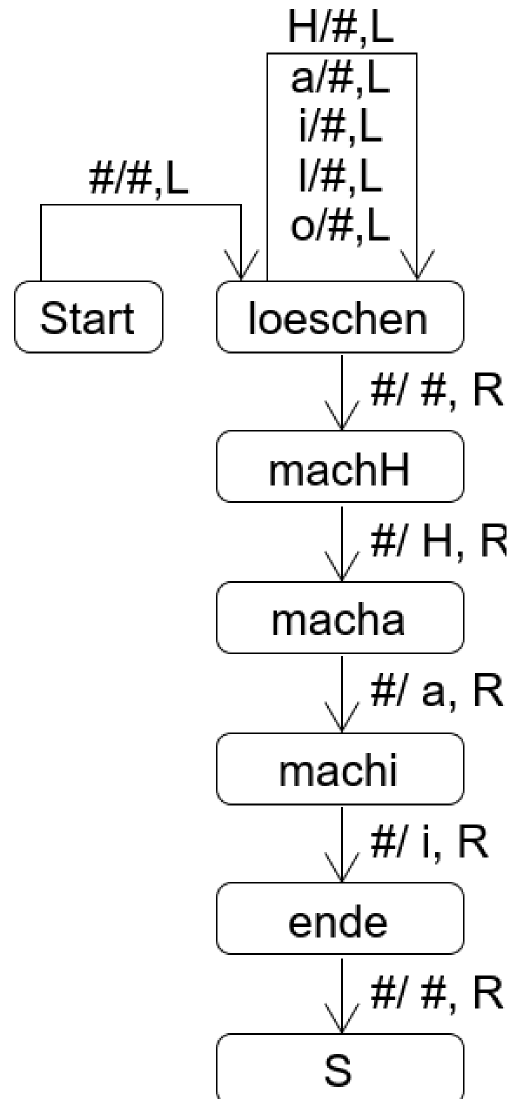


Beispiel 4: Turing- Maschine (1/2)

Schreibe eine Turing-Maschine, die ihre Eingabe löscht und „Hai“ auf das Band schreibt; Alphabet {H, a, i, l, o}

```
tm.stringAlsTuringMaschine("""
  % Zustaende
  Z: Start loeschen machH macha machi ende S
  % Alphabet, Leerzeichen # automatisch dabei
  A: H a i l o
  % Start
  S: Start
  % Ueberfuehrungsfunktion
  % alt  lesen neu  schreiben Richtung
  Start    #   loeschen  #       L
  loeschen H   loeschen  #       L
  loeschen a   loeschen  #       L
  loeschen i   loeschen  #       L
  loeschen l   loeschen  #       L
  loeschen o   loeschen  #       L
  loeschen #   machH     #       R
  machH    #   macha     H       R
  macha    #   machi     a       R
  machi    #   ende      i       R
  ende     #   S         #       S
  """);
```

Beispiel 4: Turing- Maschine (2/2)




```
Start: (Start, #Hallo#)
(loeschen, #Hallo#)
(loeschen, #Hall##)
(loeschen, #Hal###)
(loeschen, #Ha####)
(loeschen, #H#####)
(loeschen, #######)
(machH, #######)
(macha, #H#####)
(machi, #Ha####)
(ende, #Hai###)
(S, #Hai###)
```

Informelle Analyse

Beispiele suggerieren Verwandtschaft zur Programmierung:

- Alternativen spezifizierbar
- Schleifen spezifizierbar
- Informationen (Parameter) erkennbar und lesbar
- Variablen (z. B. Anzahl Striche) variierbar
- neue Variablen und Speicherelemente (neue Zeichen) spezifizierbar

-> Church‘sche These könnte wahrscheinlich stimmen

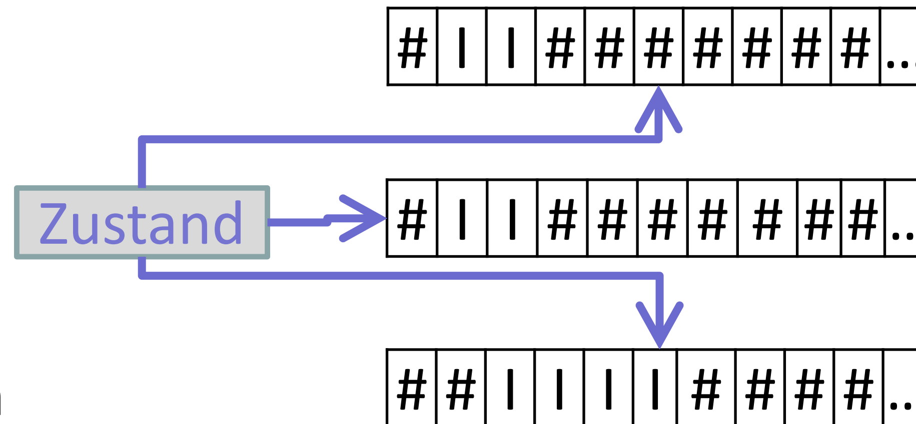
(was wäre wenn nicht; völlig neue Gedankenwelt; allerdings werden dann eine oder mehrere Thesen ähnlicher Art benötigt (Axiome) um neue Schlüsse abzuleiten)

Turing-Maschinen-Varianten (1/3)

n Band

n S-Leseköpfe

1 Zustand



Überführungsfunktion

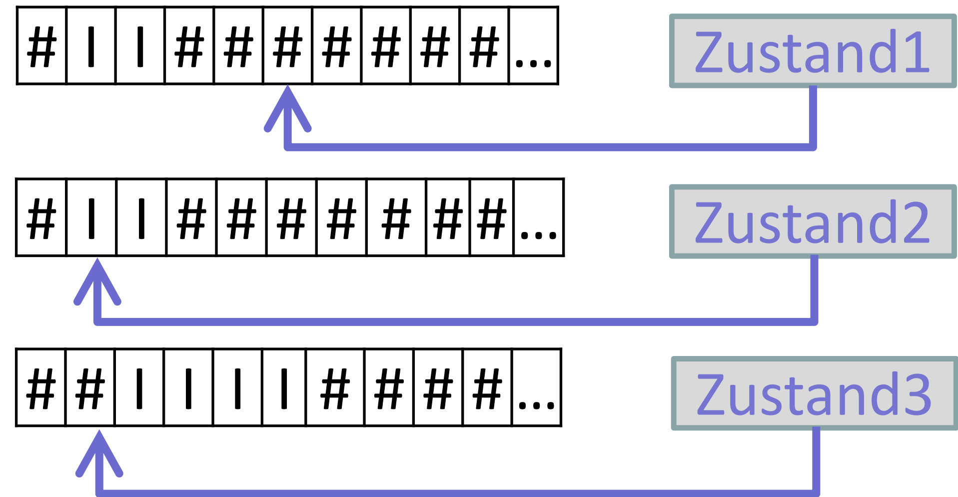
- über(Zustand, Zeichen1, ..., Zeichenn) = (Folgezustand, NeuesZeichen1, ..., NeuesZeichenn, Richtung1, ... , Richtungn)
- zentraler Zustand, der mehrere Zeichen gleichzeitig lesen und ändern kann
- sinnvolle Richtung „stehen bleiben", wenn z. B. auf anderen Bändern was gesucht wird
- „leicht" in unsere Ein-Band-Turing-Maschine überführbar, Bandinhalte mit Trennsymbol getrennt hintereinander, dann immer ein Schritt pro Band (viel wandern und Lücken schaffen)

Turing-Maschinen-Varianten (2/3)

n Bänder

n S-Leseköpfe

n Zustände



Überführungsfunktion

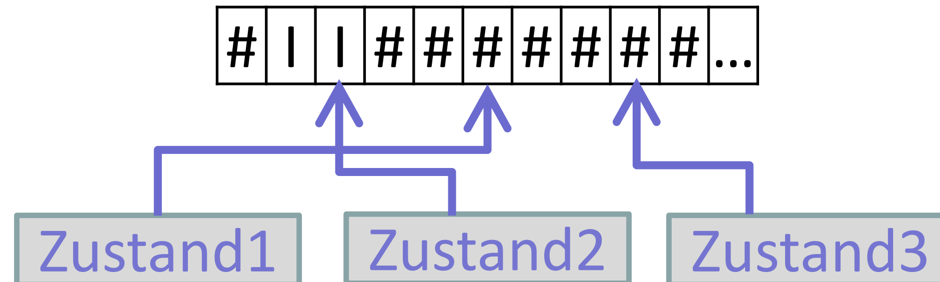
- über(Zustand1, …, Zustandn, Zeichen1, …, Zeichenn) = (Folgezustand1, …, Folgezustandn, NeuesZeichen1, …, NeuesZeichenn, Richtung1, … , Richtungn)
- anschaulich: n parallel laufende Maschinen
- „leicht" in unsere Ein-Band-Turing-Maschine überführbar, Bandinhalte mit Trennsymbol getrennt hintereinander, Zustand ergibt sich aus kartesischem Produkt, der anderen Zustandsräume, Zeichen werden einzeln abgearbeitet

Turing-Maschinen-Varianten (3/3)

1 Band

n Zustände

n S-Leseköfe



Überführungsfunktion

- über(Zustand1, ..., Zustandn, Zeichen1, ..., Zeichenn) = (Folgezustand1, ..., Folgezustandn, NeuesZeichen1, ..., NeuesZeichenn, Richtung1, ... , Richtungn)
- anschaulich: n auf einem Band parallel laufende Maschinen
- „leicht“ in unsere Ein-Band-Turing-Maschine überführbar, Zustand ergibt sich aus kartesischem Produkt, der anderen Zustandsräume, Zeichen werden einzeln abgearbeitet
- Variante mit einem Zustand denkbar (aus kartesischem Produkt)

Berechenbare Funktion

Definition: Sei ein Alphabet ohne #-Zeichen gegeben. Eine Funktion

$$f: (Alphabet^*)^m \text{ -> } (Alphabet^*)^n$$

heißt *Turing-Maschinen-berechenbar*, wenn es eine Turing-Maschine TM=(Zustände, Alphabet‘, Überführungsfunktion, Start) mit # $\in$ Alphabet‘, Alphabet $\subseteq$ Alphabet‘ gibt, so dass für alle w1,…,wm, u1,…,un $\in$ Alphabet* gilt:

- f(w1,…,wm) = (u1,…,un) genau dann wenn es eine terminierende Berechnung Start, #w1#w2#...#wm$\underline{\#}$ ->* z, #u1#u2#...#un$\underline{\#}$ gibt
- f(w1,…,wm) ist undefiniert genau dann wenn TM in #w1#w2#...#wm$\underline{\#}$ startet und die Rechnung hängt oder nicht terminiert

Beispiel 3 zeigt, dass f: Alphabet* -> Alphabet * mit $f(I^n) = I^{2n}$ und sonst undefiniert, berechenbar ist (genauer Ergebnis noch nach links schieben)

Wieso mal ohne mal mit

- In der vorherigen Definition werden Funktionen auf Worten ohne # betrachtet
- Hintergrund ist die Startkonfiguration: (Start, #w#) , das linke Leerzeichen dient zur Erkennung, dass die Maschine vor Eingabe steht; würde w Leerzeichen enthalten, wäre es nicht möglich den Anfang zu erkennen
- bei #w1#w2#...#wm# klar, m-tes Leerzeichen ist beim Durchlauf nach rechts erstes Zeichen vor der Eingabe (mit m Zuständen einfach erreichbar)
- Während der Berechnung können beliebig viele # an beliebigen Stellen stehen
- Sollten in w Leerzeichen stehen sollen, ist dies einfach konstruierbar, nutze dazu sonst nicht genutztes Zeichen z und wandele beim ersten Durchlauf von rechts nach links alle z in # um

Komposition von Funktionen

Definition: Gegeben f: $(\text{Alphabet}^*)^m$ -> $(\text{Alphabet}^*)^n$
g: $(\text{Alphabet}^*)^n$ -> $(\text{Alphabet}^*)^p$

dann ist die *Komposition von Funktionen* $g \circ f$ ist definiert als g(f(w))

- Die Komposition von berechenbaren Funktionen ist berechenbar
- Ansatz: Gegeben die Maschinen für f und g, beachte, dass Zustandsmengen disjunkt sind (sonst umbenennen), Starte mit Maschine für f, wenn diese stoppen würde, gehe zum Anfangszustand der Maschine von g über (evtl. zwei weitere neue Hilfszustände) und lasse diese laufen

- Beispiel: $f(I^n) = I^{2n}$ berechenbar, dann auch $f \circ f = I^{4n}$ berechenbar

Wunschwelt

Video

- es gibt eine eindeutige Beschreibung, was eine Software machen soll -> Anforderungsanalyse
- es entsteht ein Programm
- es wird automatisch nachgewiesen, dass das Programm die Anforderungen erfüllt (Verifikation), bzw. solange es dies nicht tut, wird entwickelt

- Realität: Verifikation findet nicht statt, nur Validierung durch Tests (oder andere Ansätze zur Umwandlung von Anforderungen in Programmcode)
- warum? generell kann es diesen automatischen Nachweise nicht geben (nächste Folien)
- Achtung: gibt (kleine) Teilaufgaben wo das machbar ist

Entscheidbarkeit

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

Definition: Gegeben sei eine Sprache $\subseteq$ Alphabet*, aus formalen Gründen mit {#, N, Y} $\subseteq$ Alphabet. Gegeben Sei eine Turing-Maschine (Zustände, Alphabet‘, Überführungsfunktion, Start) mit Alphabet $\subseteq$ Alphabet‘.

- Die Turing-Maschine *entscheidet die Sprache*, falls für alle Wörter w $\in$ Alphabet* eine terminierende Berechnung existiert mit:

$$\text{Start}, \#w\underline{\#} \rightarrow^* \begin{cases} z, \#Y\underline{\#} & \text{falls } w \in \text{Sprache} \\ z, \#N\underline{\#} & \text{sonst} \end{cases}$$

- anschaulich, die Turing-Maschine hält garantiert und berechnet ob das eingegebene Wort zur Sprache gehört oder nicht

- Ein Sprache heißt *entscheidbar* (oder rekursiv), falls eine Turing-Maschine existiert, die die Sprache entscheidet

Akzeptanz

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

Definition: Gegeben sei eine Turing-Maschine (Zustände, Alphabet, Überführungsfunktion, Start)

- Die Turing-Maschine *akzeptiert ein Wort* ∈ Alphabet* , falls die Turing-Maschine bei Eingabe des Wortes eine terminierende Berechnung hat (also anhält)
- Die Turing-Maschine *akzeptiert eine Sprache*, falls für alle Wörter w ∈ Alphabet* gilt:

  die Turing-Maschine akzeptiert w gdw. w ∈ Sprache
- Eine Sprache heißt *akzeptierbar*, wenn es eine Turing-Maschine gibt, die die Sprache akzeptiert

- für Akzeptanz reicht es aus, dass die Turing-Maschine bei zur Sprache gehörenden Worten irgendwann anhält; es ist aber unklar, ob sie dies wirklich macht (wäre zu zeigen)

Zusammenhang Entscheidbarkeit - Akzeptanz

- Satz: Ist eine Sprache entscheidbar, dann ist sie auch akzeptierbar
- Idee: Nutzung Turing-Maschine für Entscheidbarkeit, wenn die Maschine „N“ berechnen will, jage sie in eine Endlosschleife

- Die andere Richtung gilt nicht, da unklar ist, ob die Maschine für Akzeptanz irgendwann anhält

- Satz: Ist eine Sprache und Ihr Komplement akzeptierbar, dann ist die Sprache entscheidbar
- Idee: Simuliere an zwei getrennten Stellen auf dem Band die beiden Turing-Maschinen für die Akzeptanz, der Zustandsraum besteht aus dem kartesischen Produkt der beiden Turing-Maschinen; wenn eine anhält lösche alles und gebe Y oder N aus

Aufzählbarkeit

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

Definition: Eine Sprache $\subseteq$ Alphabet* heißt *rekursiv aufzählbar*, wenn Sprache = {} oder es eine Turing-Maschine (Zustände, {I, #} $\cup$ Alphabet, Überführungsfunktion, Start) gibt mit Sprache = { w | es gibt ein $n \in$ NatürlicheZahlen, einen Zustand z und eine terminierende Berechnung mit Start, $\#I^n\underline{\#}$ ->* z, $\#w\underline{\#}$ }

- werden nacheinander I^n Zeichen in die Turing-Maschine eingegeben, wird letztendlich jedes Wort der Sprache als Ergebnis vorkommen (Ergebnisse dürften sogar mehrfach vorkommen)
- „jedes Wort kommt irgendwann mal vor"
- Worte, die nicht zur Sprache gehören, werden nicht als Ergebnis vorkommen, haben z. B. eine divergierende Berechnung
- Randnotiz: für unendlich viele Wörter werden unterschiedliche Berechnungen benötigt, da Zustandsmenge endlich, gibt es mindestens einen Zustand der sich wiederholt

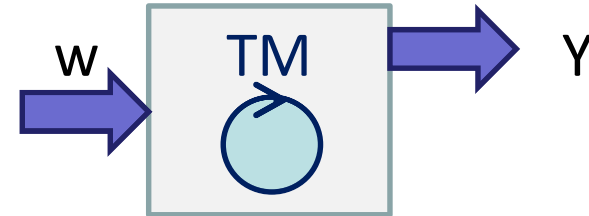
Aufzählbarkeit – alternative Definition

Definition: Sei Sprache $\subseteq$ Alphabet*, eine Turing-Maschine (Zustände, Alphabet', Überführungsfunktion, Start) mit Alphabet $\subseteq$ Alphabet', und Zustand q $\in$ Zustand *zählt die Sprache auf*, wenn Sprache = { w | es gibt u $\in$ Alphabet'* und eine Berechnung mit Start, $\underline{\#}$ ->* q, #w$\underline{\#}$u }

Eine Sprache $\subseteq$ Alphabet* heißt *rekursiv aufzählbar*, wenn sie die leere Menge ist oder es eine Turing-Maschine gibt, die sie aufzählt

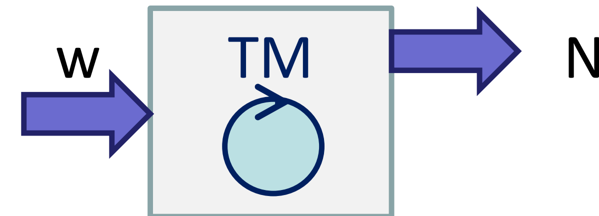
- Idee ist, dass alle Worte wieder schrittweise aufgezählt werden, dabei wird immer mal wieder der Zustand q besucht (Blink-Zustand), der angibt, dass das nächste Wort gefunden wurde; u ist Rest der z. B. zum Weiterrechnen benötigt wird
- generell gibt es oft Varianten für Definitionen; betreffen die den gleichen Begriff muss die Äquivalenz der Begriffsdefinitionen gezeigt werden (aus dem einen folgt das andere und andersherum oder durch Äquivalenzumformungen)
- Idee hier: siehe Randnotiz letzte Folie

Entscheidbarkeit - Aufzählbarkeit

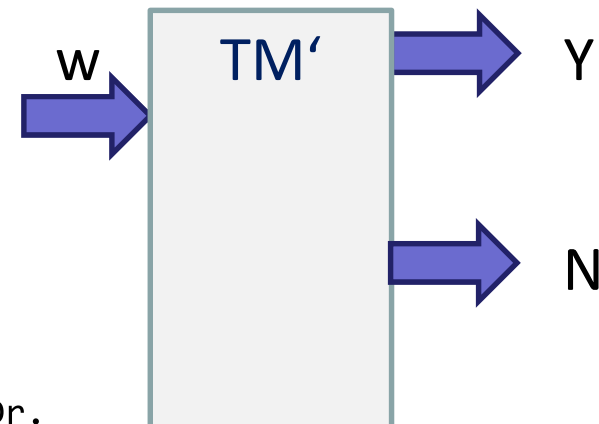


Komplement

falls Komplement aufzählbar



falls Sprache und Komplement aufzählbar



Begriffszusammenhänge

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Satz: Sprache rekursiv aufzählbar genau dann wenn Sprache akzeptierbar
- Idee: Umbau der jeweils gegebenen Turing-Maschinen

- Satz: Sprache und ihr Komplement aufzählbar genau dann wenn Sprache entscheidbar
- Idee: Umbau der jeweils gegebenen Turing-Maschinen

- Statt rekursiv aufzählbar wird oft auch *semi-entscheidbar* geschrieben

- Satz: Die Menge aller Turing-Maschinen ist rekursiv aufzählbar
- Idee: konstruiere schrittweise alle möglichen Wörter (lexikographische Ordnung) und prüfe, ob es sich um eine syntaktisch korrekte Turing-Maschine handelt

Abzählbarkeit der Turing-Maschinen

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

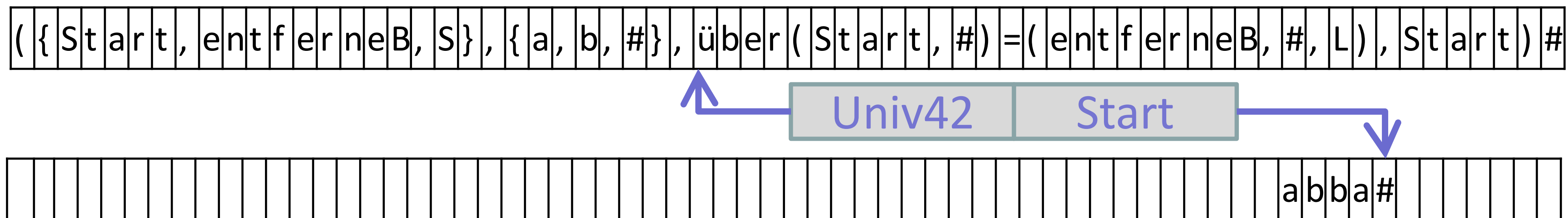
- Jede Turing-Maschine kann selbst als ein Wort aufgefasst werden, z. B. ein Wort über Alphabet* , mit Alphabet = UTF-8; sollte Zeichen außerhalb des Alphabets verwendet werden, sind diese leicht als Zeichen des Alphabets darstellbar
- Einschub: Wir wissen, dass das Alphabet {0,1} ausreicht, beliebige endliche Texte (also auch Beschreibungen von Turing-Maschinen) zu kodieren (geht nur um den Text, nicht die Ausführung)
- damit ist die Menge aller Turing-Maschinen abzählbar
- da die Menge aller Sprachen aus Alphabet* überabzählbar ist (Erinnerung „2. Cantor“) folgt

Es gibt Sprachen, die nicht von Turing-Maschinen akzeptiert werden können

- Zwischenfrage: klingt nicht gut, aber haben solche Sprachen praktische Relevanz? (Spoiler:ja)

Universelle Turing-Maschine

- bisher immer eine konkrete Turing-Maschine, die auf Eingaben losgelassen wird; eine universelle Turing-Maschine bekommt eine Turing-Maschine und ein Wort übergeben und führt die übergebene Turing-Maschine auf dem Wort aus
- Existenz durch Konstruktion nachweisbar
- Idee: Z. B. 2-Band-Turing-Maschine mit 2 Köpfen: auf dem ersten Band steht die Turing-Maschine (normale Textform), auf dem zweiten Band das abzuarbeitende Wort und dann werden immer die passenden Übergänge in der Textform der Turingmaschine gesucht und ausgeführt



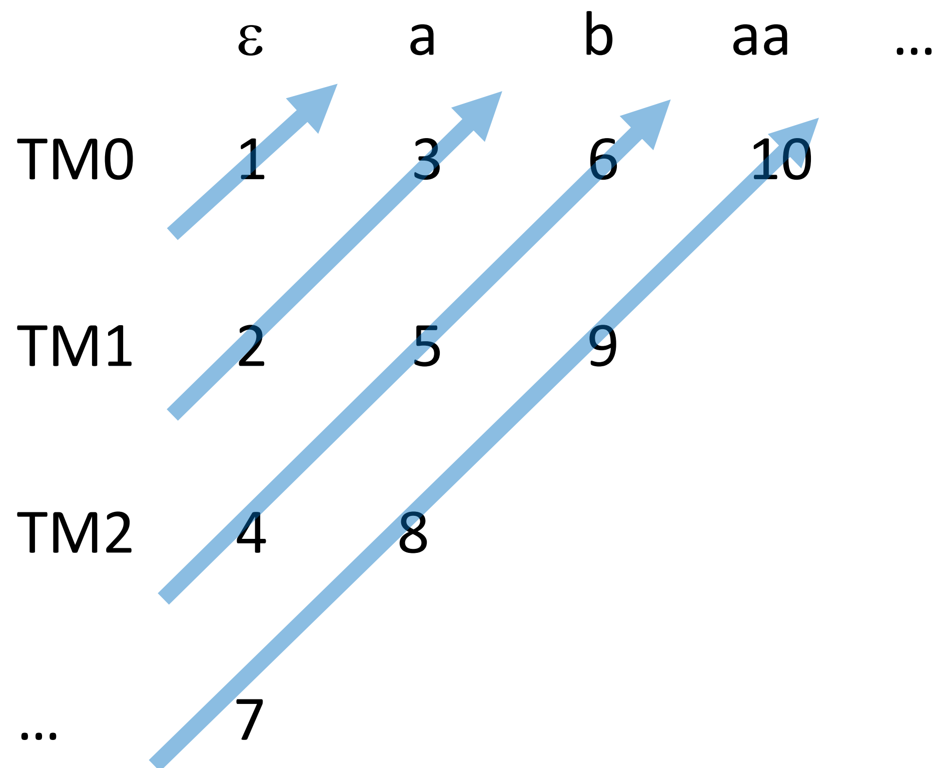
Halteproblem

als Sprache

- Halteproblemsprache = { T#w | w$\in$Alphabet*, T ist eine syntaktisch korrekte Turingmaschine und T angesetzt auf w hält an (d.h. w $\in$ Lang(T))}
- Halteproblem: gehört ein Wort zur Halteproblemsprache?
- anschaulicher: gibt es einen Algorithmus, der für eine beliebige Turing-Maschine und ein beliebiges Wort entscheidet, ob die Turing-Maschine angesetzt auf das Wort anhält oder nicht
- konkreter: Schreibe ein Java-Programm, das als Eingabe ein Java-Programm und eine Sequenz von Eingaben erhält und das die Frage beantwortet, ob das Java-Programm dann terminiert oder nicht
- Satz: Halteproblemsprache ist aufzählbar und nicht entscheidbar

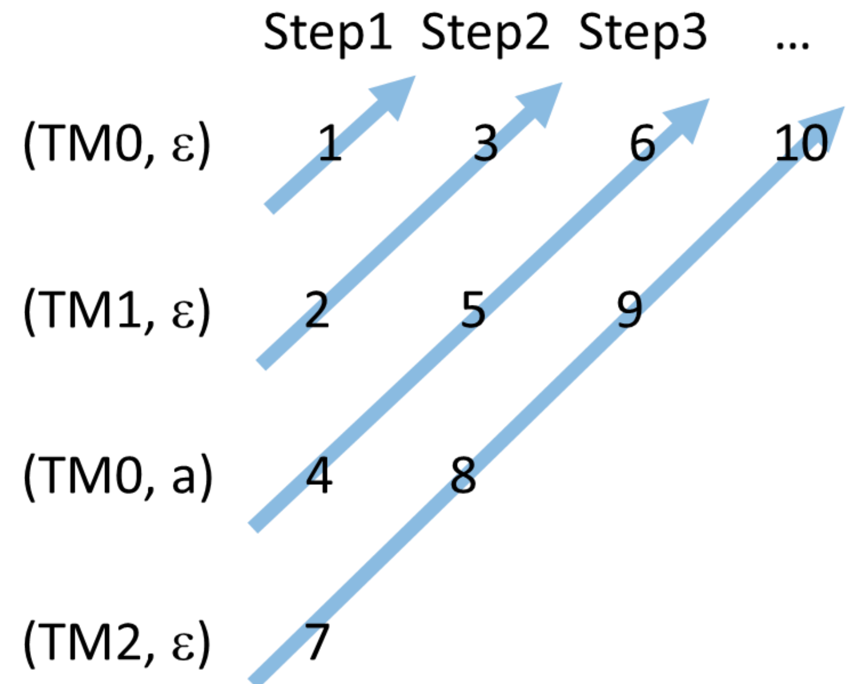
Halteproblemsprache aufzählbar (1/2)

- Erinnerung: Die Menge aller Turing-Maschinen AlleTM ist aufzählbar, die Menge Alphabet* ist aufzählbar, dann ist auch AlleTM $\times$ Alphabet* aufzählbar



Halteproblemsprache aufzählbar (2/2)

- Halteproblemsprache aufzählbar, durch: nehme erstes Paar (Turing-Maschine, Wort) und mache den ersten Schritt der Turing-Maschine auf dem Wort
- nehme zweites Paar und mache den ersten Schritt der Turing-Maschine auf dem Wort
- mache den zweiten Schritt des ersten Paares
- mache ersten Schritt des dritten, dann den zweiten des zweiten, dann den dritten des ersten Paares …
- wenn Turing-Maschine anhält gib das Paar aus



- so werden alle Elemente, Turing-Maschinen mit haltenden Eingaben, letztendlich ausgegeben

Halteproblemsprache nicht entscheidbar (1/2)

- Vorüberlegung: Jede Turing-Maschine selbst ist Element von Alphabet*
- Angenommen Problem ist entscheidbar, dann ist eine Turing-Maschine *Basis* konstruierbar, die für eine gegebene Turing-Maschine und ein Wort 0 ausgibt, wenn die Turing-Maschine auf dem Wort nicht anhält und sonst 1, kurz
- *Basis*(TMi, w) = 1 TMi hält auf w
- *Basis*(TMi, w) = 0 TMi hält nicht auf w
- Konstruiere daraus Turing-Maschine Inv, die für die i-te Turing-Maschine und das i-te Wort 0 ausgibt, wenn *Basis* 1 ausgibt und 1, wenn Basis 0 ausgibt
- *Inv*(wi) = 1 – *Basis*(TMi, wi)

	ε	a	b	aa	ab	ba	bb	…
TM0	1	0	0	0	0	0	0	…
TM1	0	1	1	0	0	1	1	…
TM2	1	1	0	1	0	0	0	…
TM3	0	0	1	0	1	0	1	…
…	…							
Inv	0	0	1	1	…			

Halteproblemsprache nicht entscheidbar (2/2)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Da *Inv* Turing-Maschine, gibt es ein i mit TMi = *Inv*.
- schauen wir jetzt die i-te-Stelle an
- Annahme: *Basis*(*Inv*, *Inv*) = 1, also *Inv* angesetzt auf *Inv* hält an, das bedeutet, dass *Inv*(*Inv*) = 0 ist, also an der gleichen Stelle eine 1 und eine 0 stehen muss
- Annahme: *Basis*(*Inv*, *Inv*) = 0, also *Inv* angesetzt auf *Inv* hält nicht an, das bedeutet, dass *Inv*(*Inv*) = 1 ist, also gilt *Basis*(*Inv*, *Inv*) = 1, also an der gleichen Stelle eine 0 und eine 1 stehen muss

	ε	a	b	aa	ab	ba	bb	…	Inv
TM0	1	0	0	0	0	0	0	…	
TM1	0	1	1	0	0	1	1	…	
TM2	1	1	0	1	0	0	0	…	
TM3	0	0	1	0	1	0	1	…	
…	…								
Inv	0	0	1	1	…				?

Konsequenz des Halteproblems

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

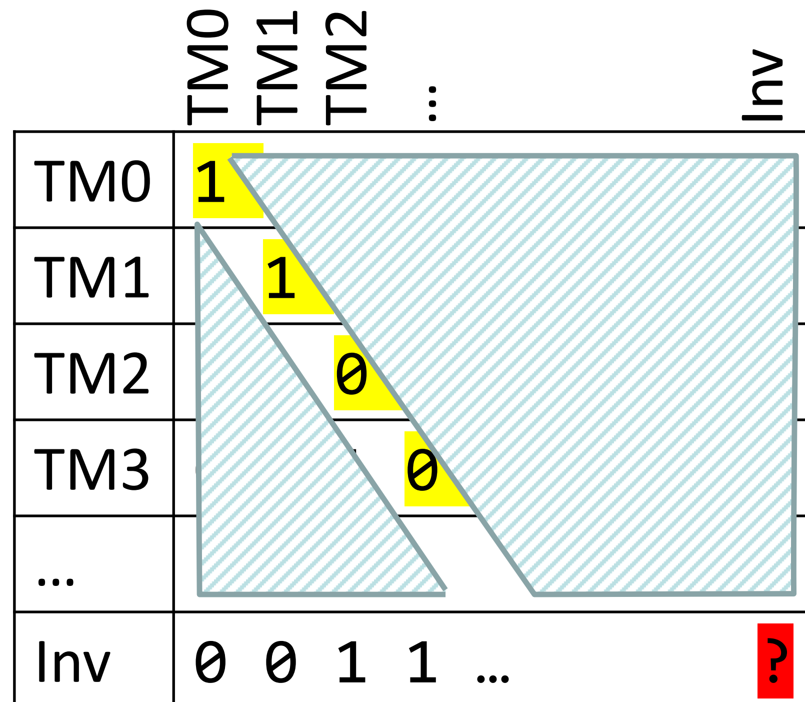
- Sie können kein immer terminierendes Programm schreiben, dass überprüft ob ein anderes Programm terminiert oder nicht
- die allgemeine Nicht-Überprüfbarkeit gilt für viele Anforderungen

- dies bedeutet nicht, dass man keine Programmeigenschaften beweisen kann; das ist aber aufwändig oder/und die Möglichkeiten des Programms sind sehr eingeschränkt
- Beispiel: Turing-Maschine läuft immer nur nach links und hält bei einem Leerzeichen an; dann ist Terminierung nachweisbar

- Frage: Wenn nicht allgemein machbar, gibt es Unterstützung bei der Beweisführung? Ja, s. Programmverifikation später in der Veranstaltung

Beweis genauer angesehen

- bei genauerer Betrachtung des Beweises haben wir mehr gezeigt; genauer ist die SpezielleHalteproblemsprache = { T | T ist eine syntaktisch korrekte Turingmaschine und T angesetzt auf sich selbst hält an (d.h. T ∈ Lang(T))} nicht entscheidbar
- Beweis wie vorher, genauer, die schraffierten Flächen waren für den Beweis irrelevant



abzählbar aber nicht aufzählbar

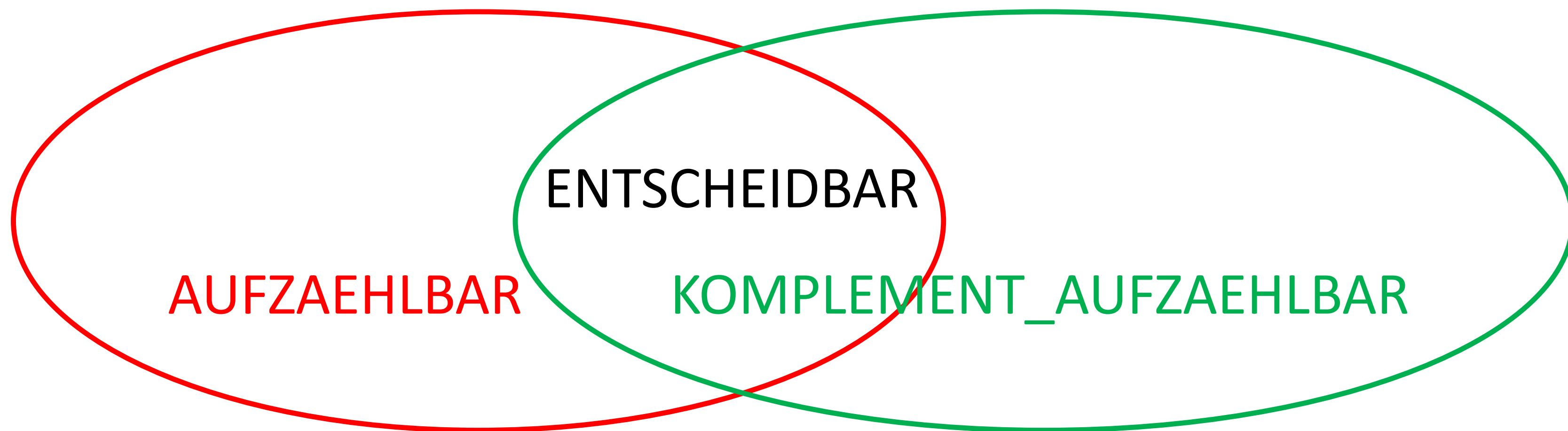
- Bekannt: Halteproblemsprache aufzählbar, aber nicht entscheidbar
- folgt: Komplement der Halteproblemsprache, also
  Alphabet* - Halteproblemsprache, ist nicht aufzählbar
- (wäre es aufzählbar, wäre Halteproblemsprache entscheidbar)

- Bekannt: Alphabet* ist abzählbar
- Bekannt: Jede Teilmenge einer abzählbaren Menge ist abzählbar (lasse Elemente, die nicht dazugehören, beim Abzählen aus)

- Da (Alphabet* - Halteproblemsprache) $\subseteq$ Alphabet* ist diese Menge nicht aufzählbar, aber abzählbar
- anschaulicher Unterschied: Das Verfahren zum Abzählen muss nicht berechenbar sein, was zum Aufzählen benötigt wird

Zusammenhang

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES



- ENTSCHEIDBAR = Menge aller entscheidbaren Sprachen
- AUFZAEHLBAR = Menge aller aufzählbaren Sprachen
- KOMPLEMENT_AUFZAEHLBAR = Menge aller Sprachen, deren Komplement aufzählbar ist

Reduktion

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Um Nicht-Entscheidbarkeit zu zeigen , muss da immer so ein Diagonal-Beweis konstruiert werden?
- Nein, Idee ist, wenn man für ein Problem zeigen kann, dass, wenn es lösbar ist, dann auch das Halteproblem lösbar ist, dann ist (auch) das Ursprungsproblem nicht entscheidbar
- Definition: Gegeben seien Sprache1 und Sprache2 nicht notwendigerweise über den gleichen Alphabeten Alphabet1 und Alphabet2 sowie eine total berechenbare Funktion f: Alphabet1* -> Alphabet2*; gelte dann für alle w $\in$ Alphabet1*

  w $\in$ Sprache1 genau dann wenn f(w) $\in$ Sprache2,

  dann heißt Sprache1 *reduzierbar* auf Sprache2 (geschrieben Sprache1 $\leq_f$ Sprache2; f kann weggelassen werden)

Reduktionsverhalten

Satz: Sei Sprache1 $\leq_f$ Sprache2, dann gilt:

- ist Sprache2 entscheidbar, dann auch Sprache 1

  (um zu prüfen, ob ein Wort in Sprache1 liegt, wird die Funktion f angewandt und für das Ergebnis der Entscheidungsalgorithmus (die zugehörige Turing-Maschine) für Sprache 2 angewandt

- ist Sprache1 nicht entscheidbar, dann auch Sprache2

  $\equiv$ Sprache2 entscheidbar -> Sprache1 entscheidbar

  $\equiv \neg$ ($\neg$ ($\neg$ Sprache 2 entscheidbar $\vee$ Sprache1 entscheidbar))

  $\equiv \neg$ (Sprache 2 entscheidbar $\wedge$ Sprache1 nicht entscheidbar)

  $\equiv$ (Sprache 2 nicht entscheidbar $\vee$ $\neg$ Sprache1 nicht entscheidbar)

  $\equiv$ ($\neg$ Sprache1 nicht entscheidbar $\vee$ Sprache 2 nicht entscheidbar)

  $\equiv$ Sprache1 nicht entscheidbar -> Sprache2 nicht entscheidbar

Reduktion anschaulicher

- OO-Adapter-Pattern: Wir kennen keine Lösung für unser Problem X, haben aber eine Lösung für ein Problem Y. Ansatz: Schreibe Programm, das X in ein Problem vom Typ Y verwandelt und nutze das bekannte Verfahren (ist genau „nur" Transformation)
- Wir wollen beliebige Objekte einer Klasse K sortieren. Java-Klassenbibliothek kann nur Objekte sortieren, die das Interface Comparable implementieren. Lösung: K implements Comparable
- Forderungen für Reduktionsfunktion
  - total: für alle Werte definiert
  - berechenbar: gibt z. B. Turing-Maschine
  - präzise: w $\in$ Sprache1 folgt f(w) $\in$ Sprache2 und
    w $\notin$ Sprache1 folgt f(w) $\notin$ Sprache2
- Beispiele oben nur Transformation, letzte Zeile von präzise irrelevant

Reduktionsfunktion (1/2) – muss nicht existieren

- total berechenbare Funktion f: Alphabet1* -> Alphabet2*; gelte dann für alle w $\in$ Alphabet1* w $\in$ Sprache1 genau dann wenn f(w) $\in$ Sprache2
- Sprache1 sei Halteproblemsprache, Sprache2 = {abba} (also endlich)
- Versuch f(w) = abba (konstante Funktion); es gilt

  w1 $\in$ Sprache1 -> f(w1) $\in$ Sprache2, da aber

  wenn w2 $\notin$ Sprache1, also

  w2 $\in$ Alphabet* - Halteproblemsprache -> f(w2) $\in$ Sprache2

  keine Reduktion, da w2 $\notin$ Sprache1
- anschaulich: f muss die „Mächtigkeit“ beibehalten, Sprache1 und Sprache2 als Bauchgefühl ähnlich groß

Reduktionsfunktion (2/2)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

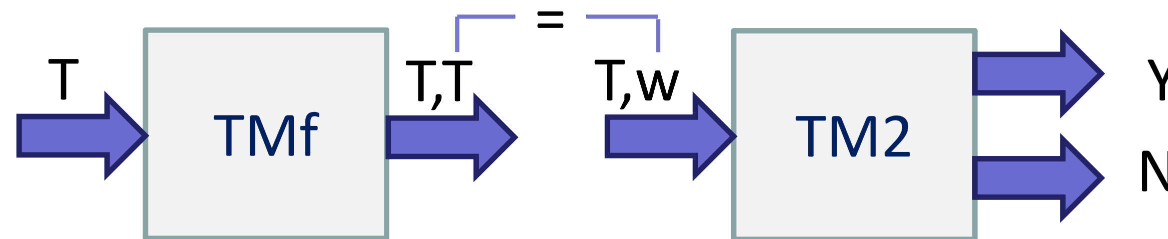
- total berechenbare Funktion f: Alphabet1* -> Alphabet2*; gelte dann für alle w $\in$ Alphabet1* w $\in$ Sprache1 genau dann wenn f(w) $\in$ Sprache2
- Sprache1 sei SpezielleHalteproblemsprache
- Sprache2 sei jetzt Halteproblemsprache (für Wort v)
- Konstruiere zu einer beliebigen Turing-Maschine T eine Maschine namens AufSichSelbst(T) , die für eine Eingabe w dann w löscht und stattdessen T auf das Band schreibt und dann T darauf ausführt
- Es gilt T $\in$ SpezielleHalteproblemsprache genau dann wenn
  AufSichSelbst(T) $\in$ Halteproblemsprache
- d.h. da die SpezielleHalteproblemsprache unentscheidbar ist auch das Halteproblem unentscheidbar (ok, wussten wir vorher)

Reduktion visualisiert

- Die Sprache AllgemeineHalteproblemsprache(w) = { T | die Turing-Maschine T hält auf w angesetzt an} ist unentscheidbar
- Annahme entscheidbar,

  dann existiert TM:

dann wäre die SpezielleHalteproblemsprache auch entscheidbar



TMf berechnet die Reduktionsfunktion für

SpezielleHalteproblemsprache $\leq_f$ AllgemeineHalteproblemsprache(w)

genereller Ansatz: nehme an, dass ein Problem entscheidbar, zeige dann, dass ein nicht entscheidbares Problem dadurch entscheidbar

Weitere unentscheidbare Probleme (für Turing-Maschinen)

- Leerheitsproblem: Menge aller Turing-Maschinen, die nie anhalten

- Totalitätsproblem: Menge aller Turing-Maschinen, die bei jeder Eingabe anhalten

- Gleichheitsproblem: Menge von Turing-Maschinen-Paaren, die die gleiche Sprache akzeptieren

Weitere unentscheidbare Probleme (Beispiele)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Gültigkeit von prädikatenlogischen Formeln, z. B.
  $(\forall a\ P(a,a)) \wedge Q(b) \wedge (\exists c\ \neg P(x,c))$

- Postsches Korrespondenz Problem

  Gegeben eine endliche Menge von Paaren PKP = {(p1, q1), ..., (pn, qn) } $\subseteq$ Alphabet* $\times$ Alphabet*. Gibt es eine endliche Folge i1,...,im mit $1 \leq ij \leq n$, so dass pi1.pi2. ... .pim = qi1.qi2. ... .qim

  z. B PKP = {(a,ba), (b ,aa), (aa,a)} Lösung aa aa b a

  = a a aa ba

  Aussage gilt, wenn Alphabet mindestens 2 Zeichen hat

Charakteristische Funktion einer Menge

Definition: Sei M $\subseteq$ Basis eine Menge. Die *charakteristische Funktion einer Menge* charakteristisch(M): Basis -> {0,1} ist definiert als

charakteristisch(M)(x) = 0, wenn x $\notin$ M

charakteristisch(M)(x) = 1, wenn x $\in$ M

Satz: Eine Sprache ist genau dann entscheidbar, wenn ihre charakteristische Funktion turing-berechenbar ist

(anschaulich, statt Y und N steht 1 und 0 auf dem Band)

Satz: Es gibt nicht turing-berechenbare Funktionen

(dies gilt für alle charakteristischen Funktionen nicht-entscheidbarer Sprachen)

also gibt es Funktionen, die nicht programmierbar sind

Busy Beaver

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- eigentlich nette Aufgabe: Gegeben seien n Zustände, schreibe eine Turing-Maschine, die mit leerem Band startet (Eingabe ε) und die terminiert und möglichst viele I auf das Band schreibt (Endzustand oder ähnliches nicht eingerechnet)

- definiere Funktion biber: NatuerlicheZahlen -> {I}* mit
  biber(n) = maximale Anzahl an Strichen, die mit n Zuständen erzeugbar
- Funktion wächst enorm schnell, genaue Werte ab n=5 nicht bekannt
- biber(5) ≥ 4098
- biber(6) ≥ $1{,}29 * 10^{865}$
- Varianten mit n-Band m-Kopf etc. denkbar

Busy Beaver - Anfang

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

biber(2) = 4

```
z0 #   z1 1 R
z0 1   z1 1 L
z1 #   z0 1 L
z1 1   S  1 S
```

```
(z0,   ###)
(z1,   #1#)
(z0,   #11)
(z1,   #11)
(z0,  #111)
(z1,  1111)
(S ,  1111)
```

biber(3) = 6

```
z0 #   z1 1 L
z0 1   z2 1 R
z1 #   z0 1 R
z1 1   z1 1 L
z2 #   z1 1 R
z2 1   S  1 S
```

```
(z0,   ###)
(z1    #1#)
(z0,   11#)
(z2,   11#)
(z1,   111#)
(z0,   1111#)
(z1,   11111)
(z1,   11111)
(z1,   11111)
(z1,   11111)
(z1,  #11111)
(z0,  111111)
(z2,  111111)
(S,   111111)
```

biber(4) = 13

```
z0 #   z1 1 R
z0 1   z2 # R
z1 #   z0 1 L
z1 1   z0 1 R
z2 #   S  1 S
z2 1   z3 1 R
z3 #   z3 1 L
z3 1   z1 # L
(S, 111111111111#1)
```

biber(5) = 4098 (?)

```
z0 #   z1 1 L
z0 1   z2 1 R
z1 #   z2 1 L
z1 1   z1 1 L
z2 #   z3 1 L
z2 1   z4 # R
z3 #   z0 1 R
z3 1   z3 1 R
z4 #   S  1 S
z4 1   z0 # R
```

https://www.thi.uni-hannover.de/fileadmin/thi/abschlussarbeiten/2020/ba_john.pdf
(sehr untypische Bachelor-Arbeit)

biber nicht berechenbar

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Satz: Die Funktion biber ist nicht turing-berechenbar
- angenommen es gibt eine Turing-Maschine, die biber berechnet, sie habe z Zustände.
- es wird folgende neue Turing-Maschine konstruiert: Sie schreibt m Striche auf das Band, benötigt dazu maximal m/2-Zustände und führt dann die biber-Turing-Maschine aus
- die neue Maschine hat maximal $z + m/2 + 42$ Zustände (42 willkürlich, abhängig von Turing-Maschinen-Variante, z. B. um korrekte Startposition zu setzen)
- wähle m jetzt so, dass $m > z + m/2 + 42$ (z und 42 sind Konstanten, also kein Problem)
- dann schreibt die neue Maschine biber(m) Striche auf das Band, aber mit weniger als m Zuständen. Widerspruch.

Variationen – Turing-Maschine als Bandinhalt

- wir haben die Turing-Maschine als Zeichenfolge auf das Band geschrieben; bleibt die Frage wieviele verschiedene Zeichen neben dem Leerzeichen werden eigentlich benötigt?
- Es reichen 2 Zeichen
- Binärcodierung in {0,1} klassisch kein Problem, wichtig nur, feste Bitanzahl pro Zeichen, um Zeichen voneinander trennen zu können
- Es reicht ein Zeichen
- Unär-Codierung, zu jedem Wort, also auch jeder Turing-Maschine gehört eine Gödel-Zahl, diese kann mit Strichen auf das Band geschrieben werden (da Dekodierung der Gödelzahl berechenbar, können alle Schritte basierend auf Gödel-Zahlen erfolgen)
- deshalb berechenbare Funktionen verallgemeinert als f: NatürlicheZahlen -> NatürlicheZahlen dargestellt (nutzt Gödelisierung)

Alternative Rechenmodelle

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

Video

Zur Validierung der Church'schen These wurden viele verschiedene Berechnungsmodelle formalisiert; es konnte (zumindest bisher) immer die Äquivalenz zum Turing-Maschinen-Modell gezeigt werden

- Register-Maschine
- LOOP-Programme
- my-rekursive Funktionen
- Ersetzungssysteme
- praktisch jede Programmiersprache
- ...

nicht

- SQL-92
- reguläre Ausdrücke
- ...

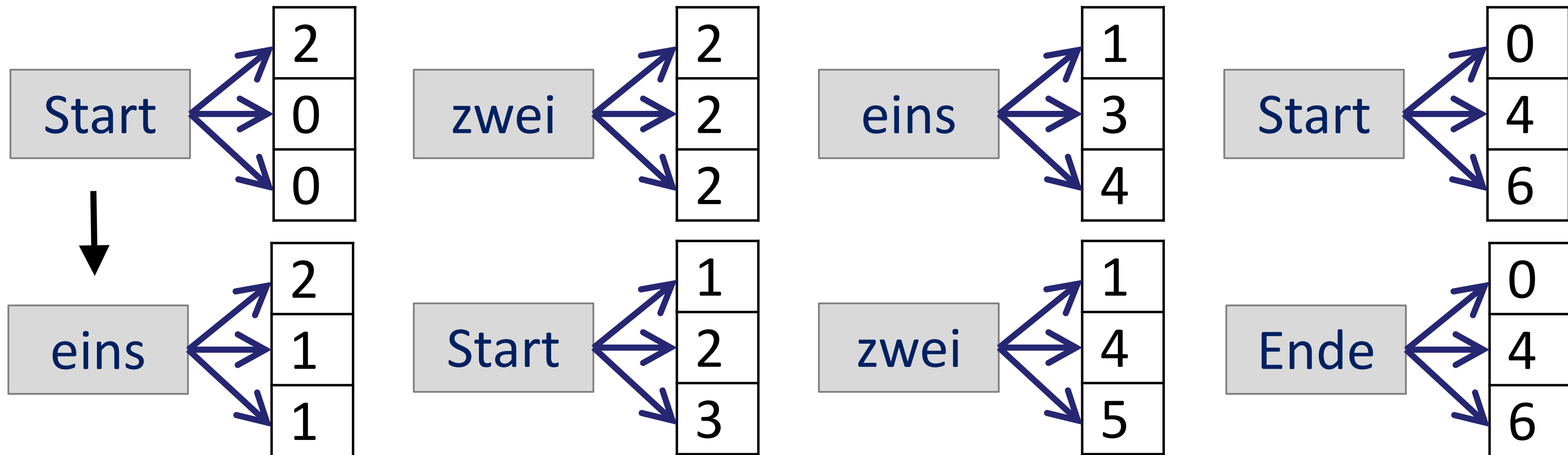
Register-Maschine (1/2)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

Definition: Eine k-Registermaschine ($k \in$ NatürlicheZahlen) hat k Register, also Speicher in denen jeweils eine natürliche Zahl steht und ist ein 4-Tupel (Zustände, Überführungsfunktion, Start, Ende) mit

- Zustände endliche Menge, enthält mindestens Start, Ende
- Überführungsfunktion: Zustände $\times$ $\{0,1\}^k$ -> Zustände $\times$ $\{-1,0,1\}^k$ dabei steht {0,1} für die Prüfung ob der Wert im n-ten Register 0 ist oder nicht (1), dabei steht {-1,0,1} für die jeweilige Wertänderung die auf diesem Register ausgeführt wird
- Eine Konfiguration ist Element von Zustände $\times$ NatürlicheZahlen^k
- Am Anfang erhalten die ersten m Register die Eingabewerte, alle anderen Register sind 0, die Berechnung endet, wenn der Zustand Ende erreicht wird, das Ergebnis steht in den ersten n Registern, berechnet Funktionen NatürlicheZahlen^m -> NatürlicheZahlen^n

Register-Maschine (2/2)



ueber(Start,(1,0,0)) = (eins, (0,1,1))

ueber(Start,(1,1,1)) = (eins, (0,1,1))

ueber(Start,(0,0,0)) = (Ende, (0,0,0))

ueber(Start,(1,1,1)) = (Ende, (0,0,0))

ueber(eins,(1,0,0)) = (zwei, (0,1,1))

ueber(eins,(1,1,1)) = (zwei, (0,1,1))

ueber(zwei,(1,1,1)) = (Start, (-1,0,1))

Register-Maschine-Variante

- hat beliebige Anzahl von Registern, die natürliche Zahlen aufnehmen können, in den ersten n Registern steht am Ende das Ergebnis
- Programm ist Folge von Befehlen, dazu gibt es Befehlszähler, mit dem nächster Befehl ausgewählt wird (Sprünge möglich)
- typische Befehle:

  Befehlsnummer: COPY RegisterX TO RegisterY

  Befehlsnummer: ADD Konstante TO RegisterX

  Befehlsnummer: SUB Konstante FROM RegisterX

  Befehlsnummer: COPY INDIREKT(RegisterX) TO RegisterY

  Befehlsnummer: GOTO Befehlsnummer

  Befehlsnummer: IF RegisterX IS 0 GOTO Befehlsnummer

  Befehlsnummer: END
- das Bauchgefühl „Assembler“ passt
- gibt Varianten, viele Mengen von Befehlsätzen sind turing-mächtig

WHILE-Programme (1/2)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Variablen: x0, x1, ... Konstanten 0, 1, 2, ... Operatoren: +, -
  Trennsymbole: ; , := Schlüsselworte: LOOP DO END
- Befehle: Variable := Variable + Konstante
  Variable := Variable – Konstante // minimal 0
  LOOP Variable DO Befehle END
  WHILE Variable $\neq$ 0 DO Befehle END
  Befehle1; Befehle2
- x0 := x1 + 0;
  LOOP x2 DO x0 := x0 + 1 END
- berechnet Funktionen NatürlicheZahlen^n -> NatürlicheZahlen
  Eingabewerte w1,...,wn stehen in x1, ..., xn ,
  xn+1, ... weitere Variablen haben am Anfang den Wert 0
  Ergebnis am Ende in x0

WHILE-Programme (2/2)

- geht eigentlich auch IF, sowas wie IF x42 == 0 THEN A ELSE B END
- ja, ergänze neue Variablen macha, machb, tmp, die nicht im Ursprungsprogramm vorkommen

```
machb := 0;    // automatisch
tmp := 0;      // automatisch
macha := machb + 1;
LOOP x42 DO
  machb := tmp + 1 ;
  macha := tmp + 0;
END;
LOOP macha DO A END;
LOOP machb DO B END
```

Ausblick: Satz von Rice

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

Satz: Sei TuringMaschinenFunktionen die Klasse aller turing-berechenbaren Funktionen und S eine echte, nicht leere Teilmenge von TuringMaschinenFunktionen. Dann ist

$$L(S) := \{TM \mid TM \text{ berechnet eine Funktion aus } S\}$$

unentscheidbar.

Die Beweisidee basiert wieder auf der Diagonalisierung.

vereinfacht veranschaulicht: jede nicht triviale Anforderung ist für Turing-Maschinen nicht automatisch nachweisbar

praktisch: Jede ernsthafte Anforderung kann für ein Programm nicht automatisch überprüft werden

Ausblick: Gödels Unvollständigkeitssatz

gibt genauere mehrere Unvollständigkeitssätze, zentral:

1. Es gibt kein widerspruchfreies rekursiv aufzählbares formales System mit dem alle Aussagen über die natürlichen Zahlen mit den Operatoren + und * beweisen oder widerlegen kann.
2. Sollte es solch ein System geben, könnte es die eigene Widerspruchsfreiheit nicht beweisen.

- vereinfacht: man kann mit Peano oder Zermelo-Fränkel nicht alle Aussagen über natürliche Zahlen herleiten; würde das gehen würde es Aussagen geben, die als wahr und als falsch beweisbar sein könnten
- die Forderung nach Vollständigkeit (alles was geht ist beweisbar) und Widerspruchsfreiheit (keine Aussage gleichzeitig wahr und falsch) ist also hier nicht erreichbar.
- sehr detaillierte (78:48), unendlich viel bessere Erklärung in „Gödel (miss)verstehen - Was sagt der Unvollständigkeitssatz wirklich aus? “ https://www.youtube.com/watch?v=qSiLjXlFlYE

Fazit (Zwischenstand)

- Wir wissen jetzt, dass es viele interessante Probleme gibt, die nicht durch einen Algorithmus gelöst werden können
- Wir haben festgestellt, dass es wichtig ist, die formale Semantik, hier Konfigurationen und Übergänge anzugeben, um die Bedeutung zu präzisieren

- Wir wollen trotzdem die Korrektheit von Programmeigenschaften für konkrete Fälle zeigen können
- dazu muss die Semantik von Programmen definiert werden (Schritt 2)
- dazu muss der Aufbau von Programmen definiert (Syntax) und überprüft werden können (nächster Schritt)

Video

2. Kontextfreie Grammatik

zentrale Inhalte:

- erzeugte Sprache
- Wortproblem
- Normalformen
- Ableitungsbaum

Glossar

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Ableitung
- Ableitungsbaum
- (sprach)-äquivalent
- Chomsky-Normalform
- CYK-Algorithmus
- ε -Regel
- eindeutige Sprache
- erzeugte Sprache
- Grammatik-Transformation
- Ketten-Regel
- Kleene-Stern
- kontextfreie Grammatik
- Linksableitung
- mehrdeutige Grammatik
- mehrdeutige Sprache
- Mengengleichheit
- Nichtterminale
- Normalform
- Produktionen
- Regeln
- Sprache
- Terminale
- Wortproblem

Motivation von Grammatiken

- Programmiersprachen haben immer Syntax-Regeln, die einzuhalten sind
- keine vollständigen Gemeinsamkeiten der Sprachen
  - z. B. ob Typen explizit anzugeben sind
  - z. B. wo stehen Variablennamen oder Rückgabetypen

```
func minMax(val1, val2 int) (int, int) {
```

- Syntax wird eindeutig durch Regeln definiert, oft ein Standard, z. B. https://docs.oracle.com/javase/specs/jls/se19/html/jls-19.html
- Wunsch: schnell überprüfbar, ob Syntax eingehalten wird (und nebenbei z. B. Variablen einsammeln)
- zentrale Ablaufstrukturen und Typdefinitionen werden durch kontextfreie Grammatiken beschrieben; weitere Infos während der Analyse eingesammelt (Variablen mit Sichtbarkeiten) und ausgewertet

Kontextfreie Grammatik

Definition: Eine *kontextfreie Grammatik* ist ein Viertupel Gram (Nichtterminale, Terminale, Regeln, Startsymbol). Sie besteht aus

- einer endlichen Menge von *Nichtterminal*en (auch *Variable*n genannt)
- einer endlichen Menge von *Terminal*en, die disjunkt von den Nichtterminalen ist, also Nichtterminale $\cap$ Terminale = {}
- einer endlichen Menge von *Regel*n (auch *Produktion*en genannt) aus Nichtterminale $\times$ (Nichtterminale $\cup$ Terminale)* , die typischerweise in der Form P -> Q geschrieben werden
- einem Startsymbol $\in$ Nichtterminale

- wenn nicht anders angegeben, beginnen Nichtterminale mit einem großen Buchstaben und Terminale sind kleine Buchstaben oder Ziffern

Beispiel Grammatik

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

Nichtterminale = {Start, A, B} Terminale = {a, b}

Regeln = { Start -> A Start B | ε
A -> Aa | a
B -> b | ε}

Gram = (Nichtterminale, Terminale, Regeln, Start)

- A -> Aa | a ist abkürzende Schreibweise für A -> Aa und A -> a
- Idee: Nichtterminale werden ersetzt durch die rechte Seite, beginnend mit dem Startsymbol, bis am Ende ein Wort aus Terminalzeichen steht
- Start -> A Start B -> AB -> AaB -> AaaB -> Aaab -> aaab

Ableitung

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Definition: Gegeben sei eine Grammatik Gram = (Nichtterminale, Terminale, Regeln, Start) und es gibt ein Wort w = uNv mit N $\in$ Nichtterminale und u,v $\in$ (Nichtterminale $\cup$ Terminale)* und eine Regel N -> A $\in$ Regeln. Dann kann das Wort uAv in einem Schritt aus w *abgeleitet* werden. Hierfür wird (auch) w -> uAv geschrieben.
- Für Wörter mit w0 -> w1 -> … -> wn wird verkürzend auch w0 ->* wn geschrieben, dabei gilt auch w0 ->* w0 (0 Schritte). Die Folge w0,…,wn heißt Ableitung (oder Rechnung) von w0 nach wn in Gram der Länge n.

Erzeugte Sprache

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Definition: Die von einer Grammatik *erzeugte Sprache* Lang(Grammatik) ist die Menge aller ausschließlich aus Terminalzeichen bestehender Worte, die aus dem Startsymbol abgeleitet werden, formaler

  Lang(Grammatik) = { w $\in$ Terminale* | Start ->* w }

- Zwei Grammatiken Gram1 und Gram2 heißen *äquivalent* (genauer sprachäquivalent), wenn gilt Lang(Gram1) = Lang(Gram2)
- Start -> A Start B | ε A -> Aa | a B -> b | ε

  Lang(Gram) = $\{a^m b^n \mid m \geq n\}$

- Gram2 : Start -> a Start b | A A -> Aa | ε

  mit erster Regel gleich viele a wie b, dann auf A „umschalten" und noch beliebig viele a in der Mitte erzeugen

  Lang(Gram2) = Lang(Gram)

Validierung

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Start -> A Start B | ε A -> Aa | a B -> b | ε

  Lang(Gram) = $\{a^m b^n \mid m \geq n\}$
- Die Angabe der Sprache ist zunächst nur eine Behauptung, diese ist zu beweisen (formal), zumindest zu validieren (semi-formal)
- Gleichheit von Mengen: A = B üblich zeige: $A \subseteq B$ und $B \subseteq A$
- Lang(Gram) $\subseteq \{a^m b^n \mid m \geq n\}$; jedes mit Gram erzeugte Wort liegt in dieser Menge: erste und zweite Regel garantiert gleich viele A und B, fünfte Regel wandelt nur B in b, vierte Regel wandelt A in a (damit m=n) und ermöglicht mit dritter Regel weitere a, damit $m \geq n$
- $\{a^m b^n \mid m \geq n\} \subseteq$ Lang(Gram) ; jedes Wort der Menge erzeugbar; gegeben $m \geq n$, d. h. m = n + p, wende erste Regel n-mal an, dann zweite Regel, Zwischenergebnis $A^n B^n$, dann wende p-mal dritte Regel an mit Zwischenergebnis $A^{n+p} B^n$, dann nur A und B in a b umwandeln, B eventuell löschen, wenn Wort nur aus a besteht

Wie werden Grammatiken erstellt – 1. Beispiel

HOCHSCHULE OSNABRÜCK UNIVERSITY OF APPLIED SCIENCES

$\{a^n b^n c^m d^m \mid m > 0, n > 0\}$

- nur grobes Kochrezept, kein Automatismus möglich
- schauen an welchen Stellen sich Wortfragmente wiederholen, bzw. Abhängigkeiten bestehen (für n und unabhängig davon für m), dafür dann Regeln aufstellen

  A -> aAb | ab   C -> cCd | cd

- dann Regeln mit neuen Nichtterminalen in der gewünschten Reihenfolge zusammensetzen

  Start -> AC

- dann über Optimierungen nachdenken; hier keine, aber Varianten

  Start -> aAbcCd   A -> aAb | ε   C -> cCd | ε

Wie werden Grammatiken erstellt – 2. Beispiel

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

$\{a^n b^m c^n d^p \mid m \geq 0, n \geq 0, p \geq 0\}$

- schauen an welchen Stellen sich Wortfragmente wiederholen, dafür dann Regeln aufstellen

  gleich viele a und c (auch 0) mit „was dazwischen“ :

  A -> aAc | X

  beliebig viele b: B -> Bb | ε beliebig viele d: D -> Dd | ε

- dann Regeln mit neuen Nichtterminalen in der gewünschten Reihenfolge zusammensetzen

  Start -> AD X -> B

- dann über Optimierungen nachdenken; u. a. X durch B ersetzen

  Start -> AD A -> aAc | B B -> Bb | ε D -> Dd | ε

Wie werden Grammatiken erstellt – 3. Beispiel

$\{a^n b^m \mid n < m\}$

- schauen an welchen Stellen sich Wortfragmente wiederholen, dafür dann Regeln aufstellen

  zuerst gleich viele a und b (auch 0) :

  A -> aAb   dann Übergang zu neuem Nichtterminal

  A -> B

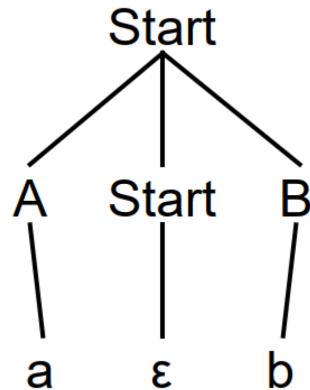
- dann mindestens ein und beliebig viele b

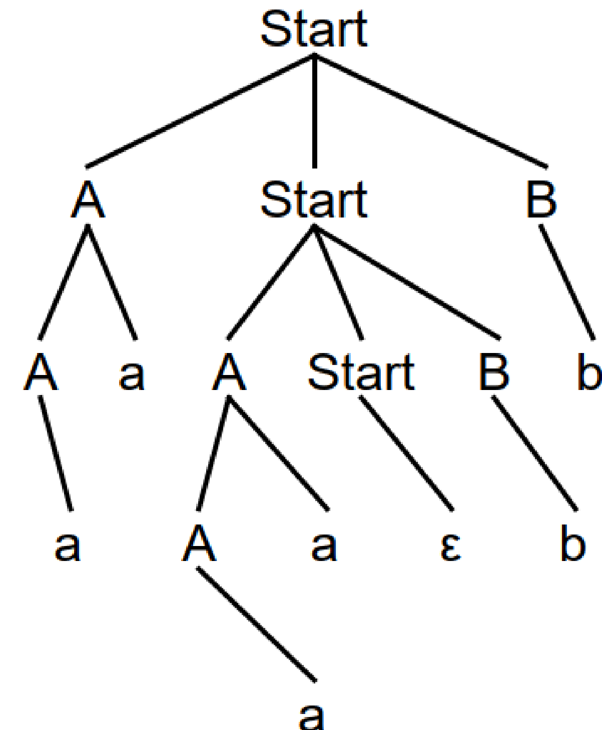
  B -> b | bB

- wähle A als Startsymbol (oder Start -> A)

Ableitungsvisualisierung: Ableitungsbaum

- Start -> A Start B | ε A -> Aa | a B -> b | ε
- Wörter : ab aaaabb





- Wurzel ist Startsymbol
- Nichtterminale als innere Knoten
- Terminale bzw. ε als Blätter
- Kanten gerichtet von oben nach unten
- man könnte genutzte Regeln annotieren
- (könnte als Spezialfall eines Graphen formalisiert werden)

Qualitäten von Grammatiken

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

Nichtterminale = {Start, A, B, X, Y}   Terminale = {a, b, x}

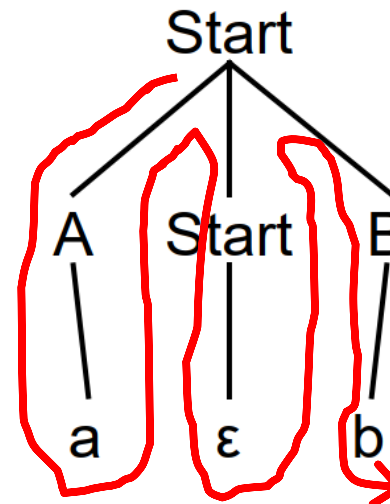
Regeln = { Start -> A Start B | ε ,   A -> Aa | a | B,
B -> A | B | b ,   X -> b}

Gram = (Nichtterminale, Terminale, Regeln, Start)

- obige Grammatik ist syntaktisch korrekt, enthält aber mehrere „schlechte“ Eigenschaften
- nicht benutztes Nichtterminal, nicht benutztes Terminal, nicht nutzbare Regel, offensichtlich unnötige Regel (B -> B)
- nicht ganz offensichtlich unnötige zyklische Ableitung (A -> B -> A)
- Ziele: Grammatiken ohne unnötigen Kram, eventuell weitere Spezialformen, mit denen einfacher gearbeitet werden kann

Linksableitung

- Definition: Eine Ableitung mit w0 -> w1 -> … -> wn heißt *Linksableitung*, wenn in jedem Schritt das am weitesten links stehende Nichtterminalzeichen ersetzt wird
- Start -> A Start B | ε   A -> Aa | a   B -> b | ε
- Start -> A Start B -> AB -> aB -> ab ist keine Linksableitung
- Start -> A Start B -> a Start B -> aB -> ab ist Linksableitung
- Linksableitung entspricht einem Linksdurchlauf durch Ableitungsbaum



- (analog Rechtsableitung)

Mehrdeutige Grammatik

- Definition: Eine Grammatik Gram heißt *mehrdeutig*, wenn es ein wort $\in$ Lang(Gram) gibt, für das es zwei verschiedene Linksableitungen gibt.

- Beispiel: S -> AB | ab A -> a B -> b mit

  Ableitungen: S -> AB -> aB -> ab und S -> ab

- Definition: Eine nicht mehrdeutige Grammatik heißt *eindeutig.*

- im obigen Beispiel gibt es auch eine eindeutige Grammatik, einfach S -> ab streichen

Mehrdeutige Sprachen

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Definition: Eine kontextfreie Sprache L heißt *eindeutig*, wenn es eine eindeutige Grammatik Gram mit Lang(Gram) = L gibt.

- Beispiel: Alle endlichen Sprachen sind eindeutig.

- Eine kontextfreie Sprache L heißt *inhärent mehrdeutig*, wenn es keine eindeutige Grammatik Gram mit Lang(Gram) = L gibt.

- Der Nachweis der inhärenten Mehrdeutigkeit ist meist sehr aufwändig (lassen wir), Beispielsprache:

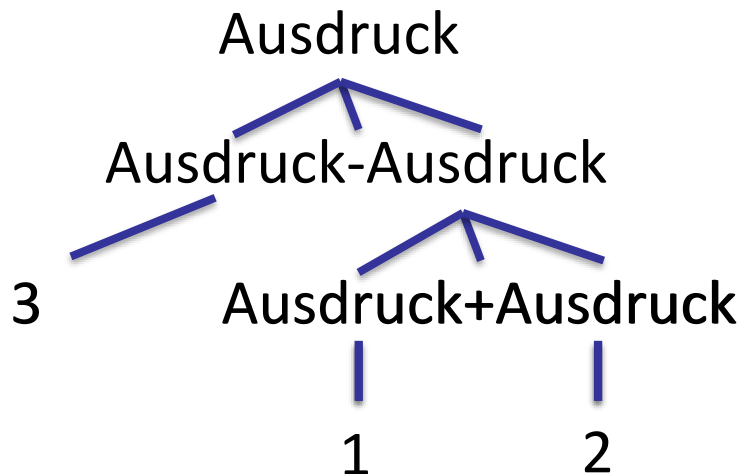
$$\{a^n b^n c^m \mid n,m \in Nat\} \cup \{a^m b^n c^n \mid n,m \in Nat\}$$

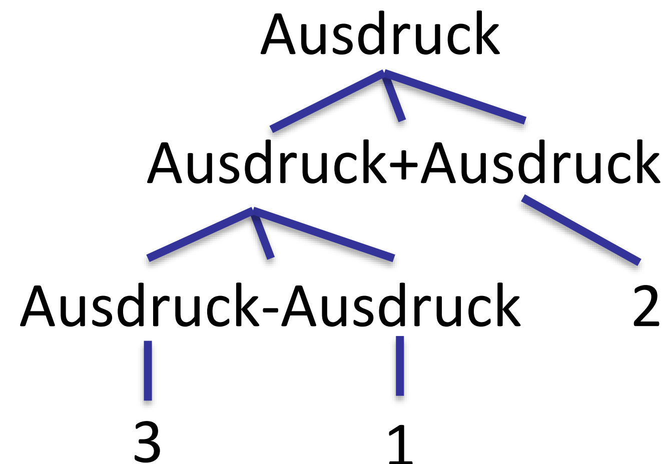
Beispiel: Mehrdeutige Grammatik

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Ausdruck -> 1
- Ausdruck -> 2
- Ausdruck -> 3
- Ausdruck -> Ausdruck+Ausdruck
- Ausdruck -> Ausdruck-Ausdruck

- Wort: 3-1+2
- basiert eine Berechnung auf der Ableitung, ergibt links: 0 und rechts: 4





Ziel: eindeutige Sprachen mit eindeutigen Grammatiken

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

Video

- warum: zu jedem Wort genau eine eindeutige Ableitung, die z.B. Basis der Festlegung der Semantik wird
- wichtig, solche Grammatiken muss es nicht geben, wenn Sprache mehrdeutig

- Ansätze Mehrdeutigkeit zu vermeiden:
- die linke Seite einer Regel soll sich nur maximal einmal auf der rechten Seite befinden, also nicht Ausdruck -> Ausdruck+Ausdruck
- bei notwendigen Wiederholungen dies mit neuer Hilfsvariable markieren, dass in diesem Zweig keine Wiederholung ist, z. B. Ausdruck -> ITerm+Ausdruck (z. B. mit ITerm kein + mehr ableitbar)
- keine Kettenregeln der Form A -> B
- ε nur einmal in Regeln und dann nur als Start -> ε

Beispiel: eine Miniprogrammiersprache (1/4)

beliebige Ziffernfolge (ohne Vorzeichen); jeweils eindeutig

- Zahl -> Zahl 0 | Zahl 1 | Zahl 2 | Zahl 3 | Zahl 4 | Zahl 5 | Zahl 6 | Zahl 7 | Zahl 8 | Zahl 9 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Variablenname, vereinfacht nur u v _ x y z, optional gefolgt von Zahl

- Variable -> u Zahl | v Zahl | _ Zahl | x Zahl | y Zahl | z Zahl |u | v | _ | x | y | z

mathematische Ausdrücke

- Ausdruck -> ITerm+Ausdruck | ITerm-Ausdruck | (Ausdruck) | ITerm
- ITerm -> Zahl
- ITerm -> Variable

Beispiel: eine Miniprogrammiersprache (2/4)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

Boolesche Bedingungen

- Bedingung -> BTerm and Bedingung
- Bedingung -> BTerm or Bedingung
- Bedingung -> (Bedingung)
- Bedingung -> BTerm
- BTerm -> not(Bedingung)
- BTerm -> Ausdruck < Ausdruck
- BTerm -> Ausdruck > Ausdruck
- BTerm -> Ausdruck == Ausdruck
- BTerm -> true
- BTerm -> false

Beispiel: eine Miniprogrammiersprache (3/4)

Programmierbefehle (Startsymbol ist Sequenz)

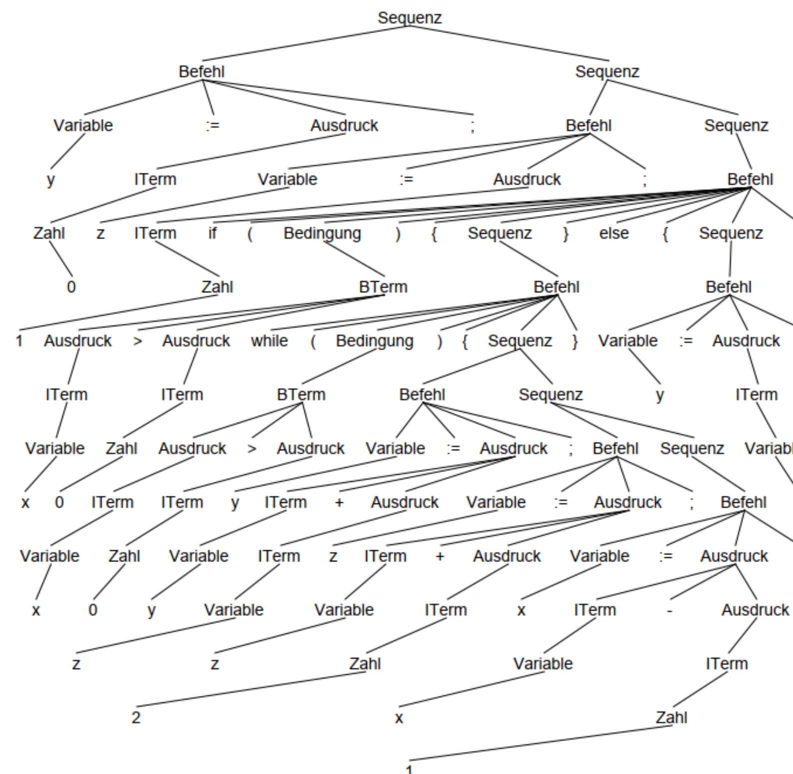
- Sequenz -> Befehl Sequenz
- Sequenz -> Befehl
- Befehl -> Variable := Ausdruck;
- Befehl -> if (Bedingung) {Sequenz} else {Sequenz}
- Befehl -> while (Bedingung) {Sequenz}

- ist nicht genau das, was wir von Java, C++, C# kennen
- ; nur am Ende von Zuweisungen, die mit := geschehen
- kein if ohne else, immer geschweifte Klammern notwendig
- kein {}, aber Trick möglich: else{x:=x;}
- Hinweis: Umgang mit Leerzeichen zu klären (hier nur Lesbarkeit)

Beispiel: eine Miniprogrammiersprache (4/4)

- Terminale = {or, (,), false, +, -, 0, 1, 2, 3, 4, 5, u, 6, v, 7, 8, x, 9, y, z, ;, {, true, <, }, >, :=, while, not, and, else, if, _, ==}
- Nichtterminale = {Zahl, Variable, BTerm, Ausdruck, Sequenz, ITerm, Bedingung, Befehl} , Start = Sequenz

```
y:=0;
z:=1;
if(x>0){
  while(x>0){
    y:=y+z;
    z:=z+2;
    x:=x-1;
  }
} else{
  y:=y;
}
```



Analyse-Miniprogrammiersprache

- generell Syntax von Programmiersprachen so darstellbar
- while als ein Zeichen schneller; while als 5 Zeichen geht genauso
- es muss aber nicht kontextfreie Anteile geben, z. B. in Ausdrücken dürfen nur Variablen vorkommen, die vorher deklariert wurden (hier z. B. als linke Seite einer Zuweisung)
- über die eigentlicher Bedeutung (Semantik) eines Programms wird keine Aussage gemacht (hier wird z. B. mit passender Semantik x quadriert mit Ergebnis in y)

- zentrales ToDo: finde zu gegebenem Programm zugehörige Ableitung, insofern Syntax-Regeln eingehalten wurden (sogenanntes *Wort-Problem*, gegeben eine Grammatik und ein Wort, prüfe ob Wort von der Grammatik erzeugt werden kann)

Transformation von Grammatiken

- bekannt: zwei Grammatiken heißen (sprach-)äquivalent, wenn Sie die gleiche Sprache erzeugen
- Ansatz zum Wortproblem für beliebige Grammatik Gram: Transformiere die gegebene Grammatik in eine äquivalente Grammatik TransGram, für die das Wortproblem einfacher zu lösen ist. Zeige Lang(Gram) = Lang(TransGram), typischerweise als zwei Inklusionen
- Ansatz hier: mehrere Transformationen
  - entferne Regeln mit ε auf der rechten Seite
  - entferne Kettenregeln der Form A -> B
  - wandle Grammatik in sogenannte Chomsky-Normalform
  - zeige Lösung des Wortproblems für die Chomsky-Normalformen
  - Berechne rückwärts zugehörigen Ableitungsbaum für Gram

Entfernung von ε-Regeln (1/3)

- 1. Schritt: Berechne alle Nichtterminalzeichen, die nach ε ableitbar sind.
- Fixpunktberechnung auf Nichtterminalzeichen

```
Vorher = {}
NachLeer = { N ∈ Nichtterminale |  N -> ε ∈ Regeln}
while(Vorher ≠ NachLeer) {
  Vorher = NachLeer
  forall  P -> Q ∈ Regeln {
     if (Q = N1...Nn  && (für 1 ≤ i ≤ n : Ni ∈ Vorher)) {
      NachLeer = NachLeer ∪ {P}
    }
  }
}
```

Entfernung von ε-Regeln (2/3)

- 2. Schritt: Konstruiere neue Grammatik, wenn ein Nichtterminal ursprünglich nach ε ableitbar, lösche es in Regeln auf der rechten Seite (falls ε zur Sprache gehört, fliegt es hier erstmal raus)

```
AlteRegeln = {}
while (AlteRegeln != Regeln) {
  AlteRegeln = Regeln
  forall  reg = P -> Z1...Zn ∈ Regeln {
    forall(i: 1...n){
      if (Zi ∈ NachLeer && n > 1) {
        Regeln = Regeln ∪ {P -> Z1...Zi-1 Zi+1...Zn}
      }
    }
    if n == 0 { // leere rechte Seite
      Regeln = Regeln – {reg}
    }
  }
}
```

Entfernung von ε-Regeln (3/3) - Beispiel

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

A -> AcB | A | BB | a   B -> b | ε   mit Start = A

1.Schritt

- 1. Iteration NachLeer = {B}
- 2. Iteration NachLeer = {B, A} merken A -> ε: A -> BB -> B -> ε

2. Schritt

- Regeln = {A -> AcB | A | BB | a , B -> b | ε } // Ausgangssituation
- Regeln = {A -> AcB | A | BB | a | cB | Ac | B , B -> b}
- Regeln = {A -> AcB | A | BB | a | cB | Ac | B | c , B -> b}
- für ursprüngliche Ableitung merken:

  A -> cB: A -> AcB -> cB   A -> Ac: A -> AcB -> Ac

  A -> c: A -> AcB -> cB -> c   A -> B: A -> BB -> B

Entfernung von Ketten-Regeln (1/3)

- 1. Schritt: Berechne alle Paare von Nichtterminalzeichen (A, B), für die gilt: A ->* B
- Fixpunktberechnung auf Paaren von Nichtterminalzeichen

```
Alt = {}
Kette = { (N, N) | N ∈ Nichtterminale  }
while(Alt ≠ Kette) {
  Alt = Kette
  forall (N1, N2) ∈ Kette
    forall  P -> Q ∈ Regeln {
      if (P == N2 && Q ∈ Nichtterminale) {
        Kette = Kette ∪ {(N1, Q)}
      }
    }
  }
}
```

Entfernung von Ketten-Regeln (2/3)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- 2. Schritt: Konstruiere neue Grammatik mit neuer Regelmenge, wenn ein Nichtterminal N1 in ein Nichtterminal N2 ableitbar und es Regel N2 -> Z (Z kein Nichtterminal) gibt, dann füge N1 -> Z in neue Regelmenge

```
NeueRegeln = {}
forall (N1, N2) ∈ Kette {  // (1)
  forall  P -> Q ∈ Regeln {
    if (P == N2 && !(Q ∈ Nichtterminale)) { // (2)
      NeueRegeln = NeueRegeln ∪ {N1 -> Q}
    }
  }
}
```

- (1) da (Ni, Ni) ∈ `Kette` werden alle alten Regeln übernommen, außer
- (2) garantiert, dass keine Kettenregeln in NeueRegeln sind

Entfernung von Ketten-Regeln (3/3) - Beispiel

A -> a | B   B -> b | bb | C   C -> c | A   mit Start = A

1.Schritt

- 0. Iteration Kette = {(A,A), (B,B), (C,C)}
- 1. Iteration Kette = {(A,A), (B,B), (C,C), (A,B), (B,C), (C,A)}
- 2. Iteration Kette = {(A,A), (B,B), (C,C), (A,B), (B,C), (C,A), (A,C), (B,A), (C,B)}

2. Schritt

- aus A -> a | B   wird   A -> a,   C -> a,   B -> a
- aus B -> b | bb | C wird B -> b | bb, A -> b | bb, C -> b | bb
- aus C -> c | A wird C -> c, B -> c, A -> c
- für ursprüngliche Ableitung immer Herleitung merken, z. B.

  A -> c: A -> B -> C -> c   A -> b: A -> B -> b   A -> bb: A -> B -> bb
- Im konkreten Fall reicht : A -> a |b | bb | c

wozu Normalformen

- Normalformen generell: Dadurch dass „ein Ding“ in eine „äquivalente“ Form gebracht wird, wird es einfacher bearbeitbar, bzw. sind Eigenschaften einfacher identifizierbar

- z. B. konjunktive oder disjunktive Normalform von Booleschen Formeln
- Beispiel KNF: $(a \vee b \vee c) \wedge (a \vee \neg b \vee \neg c) \wedge \neg a$
- zeigt auch, dass es mehrere Arten von zugehörigen Normalformen (passend zur Aufgabenstellung) geben kann

- z. B. obere Dreiecksmatrix bei Matrizen
- einfach Lösungsraum bestimmen
- Rang bestimmen

$$\begin{pmatrix} 1 & -1 & -1 \\ 0 & 1 & -1 \\ 0 & 0 & 3 \end{pmatrix}$$

Chomsky-Normalform

- Idee: (relativ) einfache Möglichkeit zu erkennen, ob ein Wort ableitbar ist, dazu nur Regeln die auf der rechten Seite entweder nur Terminalzeichen oder nur (mind. 2) Nichtterminalzeichen haben.
- Definition: Eine Grammatik Gram = (Nichtterminalzeichen, Terminalzeichen, Regeln, Start) ist in *Chomsky-Normalform*, wenn jede Regel eine der beiden folgenden Formen hat:
  - A -> BC mit A, B, C $\in$ Nichtterminalzeichen
  - A -> a mit A $\in$ Nichtterminalzeichen, a $\in$ Terminalzeichen
  - soll ε zur Sprache gehören gibt es die Regel Start -> ε
- Satz: Zu jeder kontextfreien Grammatik Gram gibt es ein sprachäquivalente Grammatik Gram‘ in Chomsky-Normalform.
  Lang(Gram) = Lang(Gram‘)

Konstruktion der Chomsky-Normalform

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

(1) entferne alle ε- und alle Kettenregeln

(2) für alle Regeln, die der Chomsky-Normalform widersprechen

- (a) ergänze für jedes Terminalzeichen t ein neues Nichtterminalzeichen Nt und die Regel Nt -> t
- (b) ersetze auf den rechten Seiten der ursprünglichen Regeln, die nicht nur aus genau einem Terminalzeichen bestehen, jedes Terminalzeichen durch das vorher ergänzte zugehörige Nichtterminalzeichen
- Zwischenstand: nur kritische Regeln $A \rightarrow N_1N_2...N_n$ mit $n > 1$
- (c) ergänze für jede kritische Regel mit $n > 2$, $n-2$ neue Nichtterminalzeichen $Nr_1, ..., Nr_{n-2}$, streiche kritische Regel und ergänze die Regeln $A \rightarrow N_1Nr_1$ $\quad Nr_1 \rightarrow N_2Nr_2$ $\quad Nr_{n-2} \rightarrow N_{n-1}N_n$
- (d) ergänze Start -> ε, falls Start $\in$ NachLeer
- wieder Herleitung merken, jede neue Regel basiert auf einer alten Regel

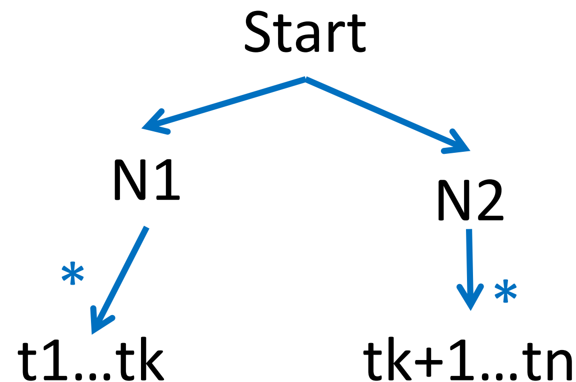
Konstruktion Chomsky-Normalform

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- A -> ABB | a   B -> bBbb | bb
- (a): ergänze N1 -> a   N2 -> b
- (b): ersetze: A -> ABB | a   B -> N2 B N2 N2 | N2 N2
- (c) gesamt:
  - A -> A M1
  - M1 -> B B
  - A -> a
  - B -> N2 Q1
  - Q1 -> B Q2
  - Q2 -> N2 N2
  - B -> N2 N2
  - N1 -> a (unnötig, aber nicht falsch)
  - N2 -> b

rekursive Lösung des Wortproblems

- gegeben Grammatik Gram in Chomsky-Normalform und Wort t1...tn, wenn n=1 oder n=0 trivial entscheidbar
- wenn n > 2 muss es zwei Nichtterminale N1 und N2 geben, so dass Start -> N1 N2 , N1 ->* t1...tk , N2 -> tk+1...tn (0 < k < n)



- Algorithmus prüft, ob es eine Zerlegung des Wortes gibt, so dass beide Teile abgeleitet werden können

rekursive Lösung des Wortproblems - formaler

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
boolean ableitbar(X Nichtterminalzeichen, t1...tn Wort) {
  if (n == 1) {
    return (X -> t1 ∈ Regeln)?
  }
  for(k = 1; k < n; k++){
    if( X -> N1 N2 ∈ Regeln && ableitbar(N1, t1...tk)
                              && ableitbar(N2, tk+1...tn) {
      return true;
    }
  }
  return false;
}

// zur Prüfung wird als erster Parameter Start genutzt
```

Grundidee: iterative Lösung des Wortproblems

- gegeben Wort t0 t1 t2 t3 t4 (nur Terminalzeichen)
- bestimme Nichtterminalzeichen, die jeweils zum obigen Zeichen abgeleitet werden (existiert immer, können mehr sein, wenn so eine Regel schon in der Ausgangsgrammatik enthalten)

{N0} {N1} {N2} {N3} {N4,X}

- bestimme Nichtterminalzeichen, die Paare ableiten, z.B. M0 -> N0N1, die Zeichen muss es nicht geben (Menge steht an linker Position)

{M0} {M1} {} {M2,Y}

- bestimme Nichtterminalzeichen, die drei hintereinander folgende Zeichen ableiten für t0 t1 t2, z. B. Q0 -> M0 N2 oder Q1 -> N0 M1

{Q0,Q1} {} {}

- schrittweise weiter, bis Wortlänge erreicht, wenn in der einen Menge das Startsymbol, ist das Wort ableitbar (und Ableitung konstruierbar)

Cocke, Younger, Kasami – Algorithmus (CYK)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

Video

```
boolean ableitbar(X Nichtterminalzeichen, t0...tn-1 Wort) {
  Set[][] n = Set<Nichtterminale>[n][n]
  //n[Zeichen im Wort, i-ter Schritt]
  for(int i = 0; i < n; i++) {
    forall  P -> ti ∊ Regeln {
      n[i][0] = n[i][0] ∪ {P}
    }
  }
  for(int j = 1; j < n; j++) { // Schrittzaehler
    for(int i = 0; i < n - j; i++) {  // Startposition Teilwort
      for(int k = 0; k < j; k++) { // Zerlegungspunkt
        forall N1 ∊ n[i][k] {
          forall N2 ∊ n[i+k+1][j-k-1] {
            forall  P -> N1 N2 ∊ Regeln {
              n[i][j] = n[i][j] ∪ {P}
  } } } } } }
  return (X ∊ n[0][n-1])?
```

CYK – Beispiel (1/5)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- 1. Schritt:

n	i=0	i=1	i=2	i=3	i=4
Wort w	a	b	a	a	b
j=0	{A,C}	{B}	{A,C}	{A,C}	{B}

S -> AB | BC
A -> BA | a
B -> CC | b
C -> AB | a

(für 1 Zeichen) für n[2][0] trage alle Nichtterminalzeichen ein, die direkt auf das Zeichen an 2.Position, also a, ableitbar sind (A ->a, C->a)

CYK – Beispiel (2/5)

- 2. Schritt:

n	i=0	i=1	i=2	i=3	i=4
Wort w	a	b	a	a	b
j=0	{A,C}	{B}	{A,C}	{A,C}	{B}
j=1	{C,S}	{A,S}	{B}	{C,S}	

S -> AB | BC
A -> BA | a
B -> CC | b
C -> AB | a

(für 2 Zeichen) für n[2][1] prüfe
ob n[2][0] kombiniert mit n[3][0] erreichbar, prüfe für Kombinationen AA, AC, CA, CC, da (nur) B ->CC trage B in das Feld ein;
man weiß, dass aus B Teilwort aa ab der Position 2 ableitbar ist

CYK – Beispiel (3/5)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- 3. Schritt:

n	i=0	i=1	i=2	i=3	i=4
Wort w	a	b	a	a	b
j=0	{A,C}	{B}	{A,C}	{A,C}	{B}
j=1	{C,S}	{A,S}	{B}	{C,S}	
j=2	{B}	{}	{B}		

S -> AB | BC
A -> BA | a
B -> CC | b
C -> AB | a

(für 3 Zeichen) für n[0][2] prüfe
ob n[0][0] kombiniert mit n[1][1] erreichbar (nein, nicht möglich AA, AS, CA, CS)
ob n[0][1] kombiniert mit n[2][0] kombinierbar, ja B ->CC (nichts für CA, SA,SC)

CYK – Beispiel (4/5)

- 4. Schritt:

n	i=0	i=1	i=2	i=3	i=4
Wort w	a	b	a	a	b
j=0	{A,C}	{B}	{A,C}	{A,C}	{B}
j=1	{C,S}	{A,S}	{B}	{C,S}	
j=2	{B}	{}	{B}		
j=3	{A,S}	{}			

S -> AB | BC
A -> BA | a
B -> CC | b
C -> AB | a

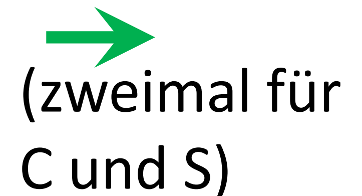
CYK – Beispiel (5/5)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- 5. Schritt:

n	i=0	i=1	i=2	i=3	i=4
Wort w	a	b	a	a	b
j=0	{A,C}	{B}	{A,C}	{A,C}	{B}
j=1	{C,S}	{A,S}	{B}	{C,S}	
j=2	{B}	{}	{B}		
j=3	{A,S}	{}			
j=4	{C,S}				

S -> AB | BC
A -> BA | a
B -> CC | b
C -> AB | a



(zweimal für C und S)

da S in n[0][4], ist Wort ableitbar

Fazit CYK

- CYK löst das Wortproblem (insofern korrekt umgesetzt)
- Ableitungsmatrix n ist als zentrales Prüfelement eindeutig
- Ableitungsmatrix kann benutzt werden um Ableitungsbaum für Grammatik und Wort zu konstruieren
- Ableitungsmatrix ermöglicht durch Prüfung ob verschiedene Ableitungen existieren, dies kann Mehrdeutigkeit zeigen
- vorgestelltes Verfahren von beliebiger Grammatik zur CYK-Nutzung ist grundsätzlich nutzbar, aber nicht sehr laufzeiteffizient
- Praxis für beliebige Grammatiken: Earley Parser (J. Earley, An efficient context-free parsing algorithm, Communications of the ACM, 13 (2): 94–102, 1970)
- für Programmiersprachen werden eingeschränkte kontextfreie Grammatiken genutzt, die noch effizienter parsen können (siehe Vorlesung Compilerbau)

Rekonstruktion der Ableitung

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

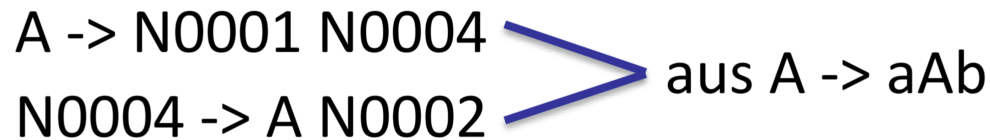
- CYK löst das Wortproblem für Grammatiken in Chomsky-Normalform
- aus der Matrix ist leicht eine/alle möglichen Linksableitungen berechenbar
- eine Rekonstruktion einer Ableitung für die Ausgangsgrammatik ist möglich, wenn alle Umwandlungsschritte dokumentiert sind
- bei dieser wird zu jeder in CYK angewandten Regel zunächst die Ausgangsregel bestimmt, dann zu dieser Regel die Ausgangsregel vor der Ketten-Regel-Entfernung, dann zu dieser Regel die Regel vor der Entfernung von ε-Regeln („rückwärts rechnen“)
- Grundidee: zu jeder neuen Regel die Liste der Regeln merken, mit denen diese Regel hergeleitet wurde

Beispiel zur Bestimmung der Ableitung (1/3)

- Ausgangsgrammatik: A -> aAb | B | ε B -> c
- ohne ε-Regeln : A -> aAb | B | ab B -> c

  A -> ab : A -> aAb -> ab
- ohne Ketten-Regeln: A -> aAb | ab | c B -> c

  A -> c : A -> B -> c
- in Chomsky-Normalform



A -> N0001 N0002 aus A -> ab

A -> c

B -> c

N0001 -> a

N0002 -> b

N0003 -> c

Beispiel zur Bestimmung der Ableitung (2/3)

A -> N0001 N0004

N0004 -> A N0002

A -> N0001 N0002

A -> c

B -> c

N0001 -> a

N0002 -> b

N0003 -> c

Wort: aabb

```
[N0001] [N0001] [N0002] [N0002]
[]      [A]     []
[]      [N0004]
[A]
```

gefundene Linksableitung:

0: A -> N0001N0004

1: N0001 -> a

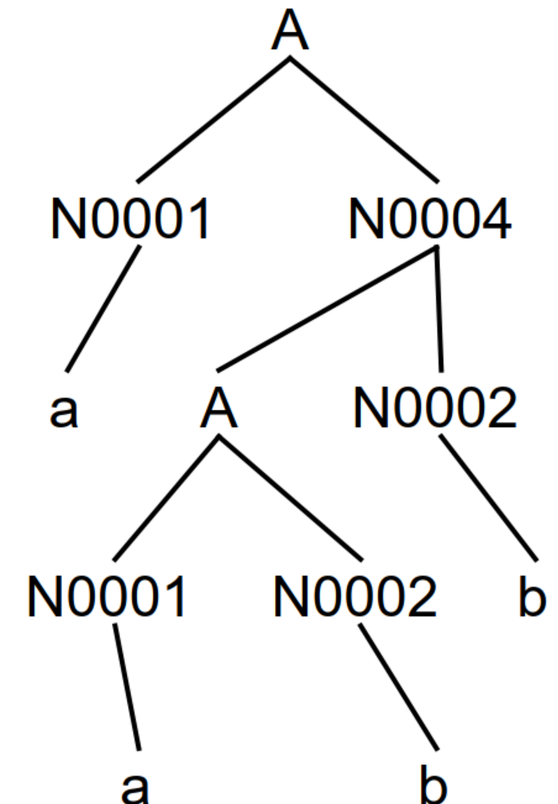
2: N0004 -> AN0002

3: A -> N0001N0002

4: N0001 -> a

5: N0002 -> b

6: N0002 -> b



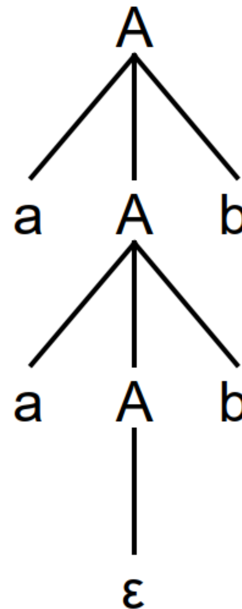
Beispiel zur Bestimmung der Ableitung (3/3)

gefundene Linksableitung:

0: A -> N0001N0004
1: N0001 -> a
2: N0004 -> AN0002
3: A -> N0001N0002
4: N0001 -> a
5: N0002 -> b
6: N0002 -> b

aus A -> aAb (0–2)

aus A -> ab (3)



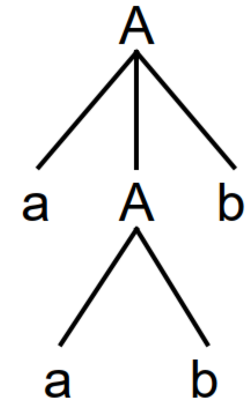
gefundene Linksableitung:

0: A -> aAb
1: A -> aAb
2: A -> ε

gefundene Linksableitung:

0: A -> aAb
1: A -> ab

aus A -> aAb -> ab



Zwischenstand

- Wir wussten schon, dass es viele interessante Probleme gibt, die nicht durch einen Algorithmus gelöst werden können
- Wir haben gesehen, dass die Syntax von Programmiersprachen im Wesentlichen mit kontextfreien Grammatiken festgelegt werden kann
- Wir können prüfen ob ein Programm syntaktisch korrekt ist und können eine zugehörige Ableitung rekonstruieren
- Die Syntax lässt (fast) keine Aussagen über die Korrektheit von Programmen zu
- dazu muss die Semantik von Programmen definiert werden (nächster Schritt)
- und nach Verallgemeinerungen zum Beweis von Programmeigenschaften gesucht werden (nächster Schritt, Teil 2)

Video

3. Semantik und Programmverifikation

zentrale Inhalte:

- formale Bedeutung von Programmen festlegen
- formaler Beweis
- Beweissystem
- Verifikation von Programmen

Glossar

Ausdruck

Bedingung

Beweisskizze

Beweissystem

Beweissystem partielle Korrektheit

Beweissystem totale Korrektheit

Divergenz

Hoare-Tripel

Korrektheitsformel

partielle Korrektheit

partielle Semantik

Schleifeninvariante

strukturierte operationelle Semantik

Substitution

Terminierungsfunktion

Testen

totale Korrektheit

totale Semantik

Übergangsrelation

Variable

Wertebereich

Zusicherung

Zustand eines Programms

Zustandsänderung

Literatur

[A094] K. Apt, E.-R. Olderog, Programmverifikation, Springer, Berlin, 2013

[Kle09] S. Kleuker, Formale Modelle der Softwareentwicklung, Kapitel 5, Vieweg+Teubner, Wiesbaden, 2009

Idee von Semantik

- Semantik soll präzise und eindeutig die Bedeutung von „Dingen" festlegen, haben wir bereits gemacht
- Syntax: Turing-Maschine
  -> Semantik: akzeptierte Sprache
- Syntax: kontextfreie Grammatik
  -> Semantik: erzeugte Sprache
- formal: Semantik ist mathematische Funktion,
  z. B. Lang(Gram)

um dann ihnen eine eindeutige Bedeutung (Semantik) zuzuordnen, um dann ein Nachweissystem aufzubauen mit dem wir zumindest von Hand die Korrektheit von Programmen beweisen können.

- Achtung: es muss nicht nur eine Semantik geben, wichtig ist nur, dass alle, die das „Ding" nutzen, die gleiche Semantik im Kopf haben, was u. a. in kritischen Fällen sichergestellt werden muss
- x/0 was ist das Ergebnis? Semantik in IEEE 754 – Standard festgelegt (mathematische Verfahren für Gleitkommazahlen)
- was uns fehlt: wie kann die Semantik von Programmiersprachen festgelegt werden (wichtig, auch nur ein Ansatz)

Kern imperativer Programmiersprachen

Ansatz: Zeige für einfache Sprache, wie Semantik definiert wird, Rest durch Erweiterungen. Es gibt drei zentrale Sprachkonstrukte:

- Sequenz oder Hintereinanderausführung, nachdem ein Befehl abgearbeitet wurde, folgt die Bearbeitung des nächsten Befehls
- Alternative, abhängig von einer auszuwertenden Bedingung wird entweder der eine oder der andere Folgebefehl ausgeführt
- Schleife, ein Befehl wird solange wiederholt ausgeführt, bis eine Abbruchbedingung erfüllt ist
- dazu kommen Variablen um Zwischenergebnisse zu speichern (Zuweisung)
- damit können alle Arten von Algorithmen beschrieben werden; ist Turing-mächtig

- Anmerkung: Es gibt andere Programmiersprachentypen: funktional (Lisp, Haskell, ML), logikorientiert (Prolog)

Erinnerung: Syntax der Beispielsprache

Sequenz -> Befehl Sequenz

Sequenz -> Befehl

Befehl -> Variable `:=` Ausdruck`;`

Befehl -> **`if`** `(`Bedingung`)` `{`Sequenz`}` **`else`** `{`Sequenz`}`

Befehl -> **`while`** `(`Bedingung`)` `{`Sequenz`}`

- Anmerkung: Leerzeichen und Einrückungen nur für Lesbarkeit, kein Teil der Syntax
- Anmerkung: Variablentypen werden nicht im Detail betrachtet, können aber wie andere Sprachkonstrukte ergänzt werden

Beispielprogramm


```
y := 0;
z := 1;
if(x > 0){
  while(x > 0){
    y := y + z;
    z := z + 2;
    x := x - 1;
  }
} else{
  y := y;
}
```

- da x keinen Wert bekommt kann man sich x als Eingabe denken (oder freie Variable)
- Auffällig ist der else-Zweig, da es kein if ohne else-Zweig gibt
- weiterhin muss im else-Zweig ein Befehl stehen
- Ansatz: Semantik für diese Kernsprache definieren, dann syntaktische Vereinfachungen

Variablen und Wertebereiche

Definition (Variablen eines Programms): Sei Prog ein Programm unserer Programmiersprache, dann bezeichnet Var(Prog) die Menge aller Variablen, die im Programm vorkommen. Für eine Variable x ϵ Var(Prog) (x Element aus Prog) sei Typ(x) der Typ oder Wertebereich [im Buch Wert(x)] (oder die Domäne) von Werten, die x annehmen kann.

- Var(Prog)={x, y, z}
- z. B. Typ(x) = Typ(y) = Typ(z) = int

```
y := 0;
z := 1;
if(x > 0){
  while(x > 0){
    y := y + z;
    z := z + 2;
    x := x - 1;
  }
} else{
  y := y;
}
```

Zustände

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

Definition (Zustände eines Programms): Sei Prog ein Programm unserer Programmiersprache mit $Var(Prog)=\{x_1,\ldots,x_n\}$, Sei $Var=\{x_1,\ldots,x_n,x_{n+1},\ldots,x_m\}$ eine Obermenge von Var(Prog), also $Var(Prog) \subseteq Var$, dann ist eine Abbildung

$$zust: Var \rightarrow Typ(x_1) \cup \ldots \cup Typ(x_m)$$

mit $zust(x_i) \in Typ(x_i)$ für i=1,..m, ein *Zustand* des Programms Prog. Die *Menge aller Zustände* von Prog wird mit Zust(Prog) bezeichnet.

Anschaulich ordnet ein Zustand jeder möglichen Variablen einen konkreten Wert aus ihrem Wertebereich zu

- Ein Zustand des Programms auf der vorherigen Folie ist z.B. zust(x)=3, zust(add)=1 und zust(erg)=0.

Semantik von Ausdrücken

- induktiv über den Aufbau, für Zustand zust

Ausdruck -> ITerm+Ausdruck | ITerm-Ausdruck | (Ausdruck) | ITerm

ITerm -> Zahl

ITerm -> Variable

- Sem(ITerm+Ausdruck , zust) = Sem(ITerm, zust) $+_{Sem}$ Sem(Ausdruck, zust)
- wieso $+_{Sem}$, formal muss jedem Syntax-Ausdruck eine semantische Bedeutung zugeordnet werden, ist hier das „übliche" + von int
- Sem((Ausdruck), zust) = (Sem(Ausdruck, zust)) // genauer $(_{Sem}$
- Sem(Zahl 2, zust) = Sem(Zahl, zust) * 10 + 2 // zust bei Konstante egal
- Sem(4, zust) = 4
- Sem(Variable, zust) = zust(Variable)
- weitere Regeln analog, im folgenden meist auf verzichtet, da hier keine Spezialfälle bearbeitet werden (z. B. x $+_{Sem}$y als (x+y)%4)

Beispiel: Semantik von Ausdrücken

- Zustand: zust(x)=3 und zust(y)=4
- Ausdruck: aus = (x+y) * 6
- Sem(aus, zust) = Sem((x+y)), zust) * Sem(6, zust)

  = (Sem(x+y, zust)) * 6 = (Sem(x, zust) + Sem(y, zust)) * 6

  = (zust(x) + zust(y)) * 6 = 7 * 6 = 42
- Generell gilt für Variablen x: Sem(x, zust)=zust(x).
- Semantik bildet Menge von Ausdrücken (AUSDRUECKE) zusammen mit Menge von Zuständen (Zust) auf Wertebereich des Ausdrucks ab
- hier ganzzahlige Ausdrücke:

  Sem: AUSDRUECKE x Zust $\rightarrow$ Integer

Semantik von Bedingungen (1/3, analog)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

Bedingung -> BTerm and Bedingung | BTerm or Bedingung |
(Bedingung) | BTerm

BTerm -> Ausdruck < Ausdruck | Ausdruck > Ausdruck | not(Bedingung) | Ausdruck == Ausdruck | true | false

- Sem(BTerm and Bedingung, zust)
  = Sem(BTerm, zust) und Sem(Bedingung, zust)
- Sem(not(Bedingung), zust) = nicht Sem(Bedingung, zust)
- Sem(Ausdruck1 < Ausdruck2, zust)
  = Sem(Ausdruck1 ,zust) $<_{Sem}$ Sem(Ausdruck2, zust)
- Sem(true, zust) = wahr
- Sem(false, zust) = falsch
- Rest analog

Semantik von Bedingungen (2/3)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Die Semantik benutzt zunächst die anschauliche Interpretation von „und“, „oder“ und „nicht“, wird (wie bekannt) formalisiert durch

X	Y	nicht X	X und Y	X oder Y
wahr	wahr	falsch	wahr	wahr
wahr	falsch	falsch	falsch	wahr
falsch	wahr	wahr	falsch	wahr
falsch	falsch	wahr	falsch	falsch

- weiterhin (wahr) ist wahr und (falsch) ist falsch
- hier Boolesche Ausdrücke:

  Sem: BOOLEANAUSDRUECKE x Zust $\rightarrow$ Boolean

- Generell bauen Semantiken meist auf den Semantiken der darunter liegenden Konstrukte auf

Semantik von Bedingungen (3/3)

bed = `(y > x) or ((x-5) < y)`, Zustand zust(x)=42 und zust(y)=40

Sem(bed, zust)= *Sem*(`(y > x)`, zust) oder *Sem*(`((x-5) < y)`, zust)
= (*Sem*(`y > x` , zust)) oder (*Sem*(`(x-5) < y`, zust));
= (*Sem*(y, zust) $>_{Sem}$ *Sem*(x, zust))
oder (*Sem*(`(x-5)`, zust) $<_{Sem}$ *Sem*(y, zust)) [*]
= (zust(y) > zust(x)) oder ((*Sem*(`x-5` , zust)) < zust(y))
= (40 > 42) oder ((*Sem*(x, zust) - 5) < 40)
= (falsch) oder ((zust(x) - 5) < 40)
= falsch oder ((42-5) < 40)
= falsch oder ((37) < 40)
= falsch oder (37 < 40)
= falsch oder (wahr)
= falsch oder wahr
= wahr

[*] hier findet wieder der Übergang vom Syntax-Zeichen > zum semantischen Zeichen >, genauer $>_{Sem}$ statt, wobei letztes Zeichen eine Semantik für Zahlenpaare hat

Semantik einer Zuweisung

Video

Definition (*Zustandsänderung*): Gegeben sei ein Programm P, ein Zustand zust$\in$Zust(P), eine Variable x$\in$Var(P) und ein Wert w$\in$Typ(x). Eine Zustandsänderung in der Variablen x auf den Wert w, geschrieben zust[x:=w], ist für y$\in$Var(P) definiert als:

$$\text{zust}[x:=w](y) = \begin{cases} \text{zust}(y), \text{ falls } y \neq x \\ w, \text{ falls } y = x \end{cases}$$

- w kann auch ein Ausdruck sein, dann *Sem*(w, zust) rechts
- Anschaulich wird durch eine Zuweisung der Wert der Variablen auf der linken Seite verändert, so dass ein neuer Zustand resultiert
- Gilt zust(x)=3 und zust(y)=5 dann ist

  zust[y:=x](x)=3 und zust[y:=x](y)=3

Herausforderung: Semantik von Programmen

- es wird nur eine Funktion Sem genutzt, da durch Parameter eindeutig ist, welche Variante (bis hierher Integer oder Boolean) genutzt werden muss
- bisherige Semantik-Betrachtung relativ einfach, da „nur“ eine Auswertung stattfindet
- Anschaulich wird ein Programm schrittweise abgearbeitet
- jeder Befehl findet in einem bestimmten Zustand statt; der Zustand wird zur Auswertung der Ausdrücke genutzt
- nach einer Befehlsausführung befindet sich das Programm in einem evtl. neuen Zustand
- Eine Programmausführung ist damit eine Folge von Zuständen

Semantik von Programmausführungen (SOS)

- Idee der *strukturierten operationellen Semantik* (SOS) nach Plotkin:
  - Definiere Semantik für einen Schritt, mit dem ein Programm in einem Zustand in ein neues Programm (das Restprogramm) mit einem neuen Zustand überführt wird
  - Dieser Übergang wird als Transition beschrieben: ein noch auszuführendes Programm Prog1 im Zustand zust1 führt einen Schritt aus und es gibt ein neues auszuführendes Programm Prog2 im Folgezustand zust2

    <Prog1, zust1> → <Prog2, zust2>
- <Prog, zust> wird auch *Konfiguration* genannt

Semantik: Abarbeitung der Programmiersprache (1/2)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Definition (Semantik von Programmen): Bezeichne Ende ein Programm ohne Anweisungen, wie es z. B. nach der vollständigen Abarbeitung eines Programms erreicht wird (oder leeres Programm). Dabei soll Ende Prog und Prog Ende jeweils als Prog interpretiert werden.

  Ein Programm Prog unserer Programmiersprache mit zust$\in$Zust(Prog) wird durch folgende Transitionen der *Übergangsrelation* schrittweise abgearbeitet:

(1) <**x := Ausdruck;** , zust> $\rightarrow$ <Ende, zust[x := Sem(Ausdruck, zust)]>

(2) Wenn <Prog1, zust1> $\rightarrow$ <Prog2, zust2>,
dann <Prog1 Prog, zust1> $\rightarrow$ <Prog2 Prog, zust2>

(schrittweise Definition über den Aufbau als Folge von Befehlen)

Semantik: Abarbeitung der Programmiersprache (2/2)

(3) Wenn Sem(Bedingung, zust) = wahr, dann

<**if(**Bedingung**)**{Prog1} **else** {Prog2}, zust> → <Prog1, zust>

(4) Wenn Sem(Bedingung, zust) = falsch, dann

<**if(**Bedingung**)**{Prog1} **else** {Prog2}, zust> → <Prog2, zust>

(5) Wenn Sem(Bedingung, zust) = wahr, dann

<**while(**Bedingung**)**{Prog}, zust>
→ <Prog **while(**Bedingung**)**{Prog}, zust>

(6) Wenn Sem(Bedingung, zust) = falsch, dann

<**while(**Bedingung**)**{Prog}, zust> → <Ende, zust>

(passender zu den Regeln könnte statt Prog auch das Nichtterminal Sequenz stehen (s. Regeln))

Möglichkeiten zur Syntax- und Semantikerweiterung

- Grundsätzlich kann man neue Sprachkonstrukte einführen und ihnen weitere Transitionsregeln zuordnen
- Befehl -> **`if`** **`(`**Bedingung**`)`** **`{`**Sequenz**`}`**

  (7) Wenn *Sem*(<Bedingung>,zust)=wahr, dann

  <**`if(`**Bedingung**`){`**Prog**`}`**, zust> → <Prog, zust>

  (8) Wenn *Sem*(<Bedingung>,zust)=falsch, dann

  <**`if(`**Bedingung**`){`**Prog**`}`**, zust> → <Ende, zust>

- Alternativ kann man eine Abbildung der neuen Sprachkonstrukte auf die alten Sprachkonstrukte definieren, wodurch die Semantik sofort festgelegt ist
- Obige Bedingung ist eine Abkürzung für:

```
if(<Bedingung>){<Befehlssequenz>} else {x := x;}
```

Werden zwei Semantiken angegeben, muss wenn möglich nachgewiesen werden, dass sie das gleiche Verhalten beschreiben

Reflexive transitive Hülle

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

Definition (Erreichbare Konfigurationen): Zu einem Programm Prog unserer Programmiersprache mit zust $\in$ Zust(Prog) wird folgende Transitionsrelation $\rightarrow^*$ als *erweiterte Übergangsrelation* definiert:

(a) <Prog, zust> $\rightarrow^*$ <Prog, zust> (reflexiv)

(b) Wenn <Prog1, zust1> $\rightarrow^*$ <Prog2, zust2>
und <Prog2, zust2> $\rightarrow$ <Prog3, zust3>
dann auch <Prog1, zust1> $\rightarrow^*$ <Prog3, zust3> (transitiv)

- Anmerkung: Die Relation kann auch zur Definition von Semantik-Regeln in Form der SOS-Semantik genutzt werden (wird teilweise sogar benötigt)

Semantik der do-while-Schleife

- Die do-while Schleife sei wie folgt definiert:

  Befehl -> **do** {Sequenz} **while** (<Bedingung>)

- eine mögliche Semantikdefinition durch Übersetzung:

  Prog **while(**<Bedingung>**)**{Prog}

- alternativ: Angabe von Semantikregeln für die erweiterte Übergangsrelation

  Wenn <Prog, zust1> $\rightarrow^*$ <Ende, zust2>
  und *Sem*(Bedingung, zust2) = wahr, dann

  <**do** {Prog} **while(**Bedingung**)** , zust1>
  $\rightarrow^*$ < **do** {Prog} **while(**Bedingung**)** , zust2>

  Wenn <Prog, zust1> $\rightarrow^*$ <Ende, zust2>
  und *Sem*(Bedingung, zust2) = falsch, dann

  <**do** {Prog} **while(**Bedingung**)** ,zust1> $\rightarrow^*$ <Ende, zust2>

Beispiel: Programmabarbeitung (1/2)

<Prog,zust>

mit (1),(2) → < `while(x > 0){`

```
erg := erg * 2;
x := x - 1;
```

`}` , zust1> mit zust1(x)=2 und zust1(erg)=1

mit (5) → <

```
erg := erg * 2;
x := x - 1;
while(x > 0){
  erg := erg * 2;
  x := x - 1;
}
```

, zust1>

mit (1),(2) → <

```
x := x - 1;
while(x > 0){
  erg := erg * 2;
  x := x - 1;
}
```

, zust2> mit zust2(x)=2 und zust2(erg)=2

Prog ≡

```
erg := 1;
while(x > 0){
  erg := erg * 2;
  x := x - 1;
}
```

zust(x)=2

zust(erg)=0

Beispiel: Programmabarbeitung (2/2)


```
mit (1),(2) → < while(x > 0){
                    erg := erg * 2;
                    x := x - 1;
                  } , zust3>  mit zust3(x)=1 und zust3(erg)=2
mit (5) → < erg := erg * 2;
              x := x - 1;
              while(x > 0){
               erg := erg * 2;
               x := x - 1;
              } , zust3>
mit (1),(2) → < x := x - 1;
                  while(x > 0){
                    erg := erg * 2;
                    x := x - 1;
                  } , zust4> mit zust4(x)=1 und zust4(erg)=4
```

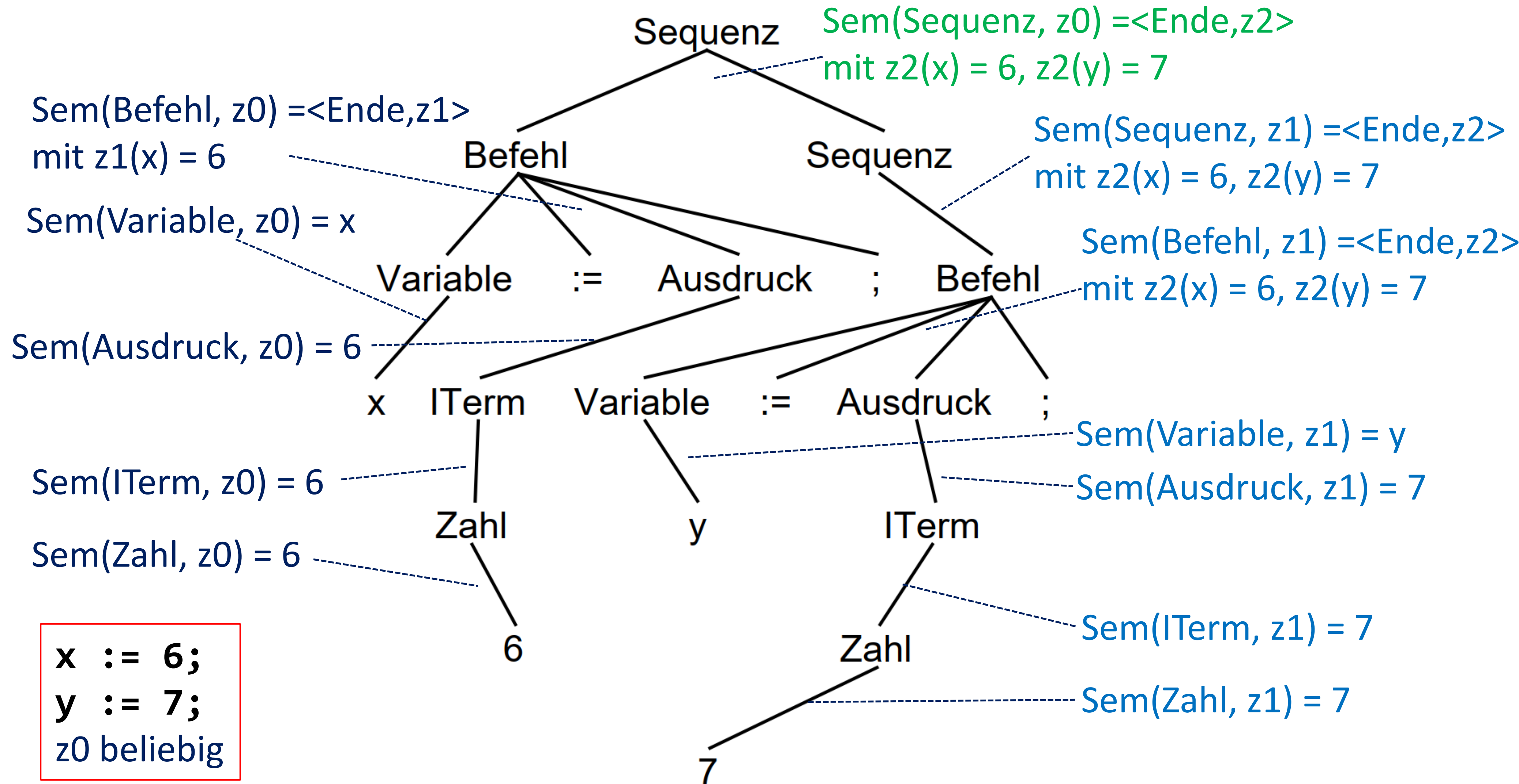
```
mit (1),(2) → < while( x > 0){
                    erg := erg * 2;
                    x := x - 1;
                  } , zust5>
mit zust5(x)=0 und zust5(erg)=4

mit (6) → <Ende, zust5>
```

Grammatiken, Ableitungen und Semantik

- Alle semantischen Betrachtungen orientieren sich an den Regeln der Grammatik
- konsequenter Ansatz: Ordne Semantik den einzelnen Ableitungen formal zu, jeweils mit einer konkreten Information zur Änderung des Zustands; formal werden Hilfsvariablen (zumindest eine) für Ausdrücke benötigt
- genauer werden Ausdrücke einfach ausgewertet und für jeden Befehl der Übergang von der Ausgangskonfiguration zur Zielkonfiguration beschrieben
- Anmerkung: Formale Semantik eines Programm wird etwas später genau definiert

Ableitungsbaum mit zugeordneter Semantik



Zusicherung

- Bisher wurden einzelne Zustände durch ihre Paare Variable und Wert einzeln angegeben
- Man kann Zustandsmengen aber auch durch Formeln (Zusicherungen) beschreiben, die verwandt mit Bedingungen sind

Definition (Zusicherung): Jede Bedingung die aus

BedingungZ -> Ausdruck Op Ausdruck

Op -> == | < | <= | > | >= | !=

konstruiert wurde, ist eine *einfache Zusicherung*. Weiterhin werden *Zusicherung*en induktiv definiert. Seien p und q Zusicherungen, dann sind

(1) $\neg p$ (nicht p)

(2) $p \wedge q$ (p und q)

(3) $p \vee q$ (p oder q)

(4) (p) auch Zusicherungen

Zusicherung und Zustandsmengen

informelle Definition (Semantik von Zusicherungen): Jede Zusicherung p kann für einen gegebenen Zustand z, der Werte für die vorkommenden Variablen enthält, nach wahr oder falsch ausgewertet werden

Zur Berechnung wird eine Semantik-Funktion *Sem* für Zusicherungen definiert. Es kann also für eine Zusicherung p und einen von den Variablen passenden Zustand zust immer *Sem*(p, zust) nach wahr oder falsch ausgewertet werden.

Zusicherungen werden später zur Semantik-Definition von Programmen und für Korrektheitsbeweise genutzt

Semantik von Zusicherungen (1/2)

Definition (Semantik von Zusicherungen): Sei p $\in$ ZUSICHERUNGEN eine Zusicherung aus einem Programm Prog, wobei ZUSICHERUNGEN für die Menge aller Zusicherungen steht. Sei zust $\in$ Zust(Prog). Die Semantik von Zusicherungen ist generell eine Abbildung einer Zusicherung und eines Zustandes auf einen der Werte „wahr" oder „falsch", also

- *Sem*: ZUSICHERUNGEN $\times$ Zust(Prog) $\rightarrow$ {wahr,falsch}. Dabei ist die Semantik von p im Zustand zust wie folgt definiert:
- falls p die Form Ausdruck1 Op Ausdruck2 hat, gilt

  Sem(p, zust) = *Sem*(Ausdruck1, zust) Op_{Sem} *Sem*(Ausdruck2, zust)
- falls p die Form Zusicherung1 $\wedge$ Zusicherung2 hat, gilt

  Sem(p, zust) = *Sem*(Zusicherung1, zust) und *Sem*(Zusicherung2, zust)

Semantik von Zusicherungen (2/2)

- falls p die Form Zusicherung1 $\vee$ Zusicherung2 hat, gilt

  Sem(p, zust) = *Sem*(Zusicherung1, zust) oder *Sem*(Zusicherung2, zust)
- falls p die Form $\neg$ Zusicherung hat, gilt

  Sem(p, zust) = nicht *Sem*(Zusicherung, zust)
- falls p die Form (Zusicherung) hat, gilt

  Sem(p, zust) = (*Sem*(Zusicherung, zust))

Beispiel: Zusicherung

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- $p \rightarrow q$ ist durch $\neg p \vee q$ definiert (logische Folgerung)
- $p \leftrightarrow q$ ist durch $(p \rightarrow q) \wedge (q \rightarrow p)$ definiert (logische Äquivalenz)

- $p \equiv x > 2 \rightarrow y > 4$

  zust(x) = 3 und zust(y) = 5, es gilt

  Sem(p, zust)= *Sem*($\neg$ x>2 $\vee$ y>4, zust)   // Negation bindet stärker

  = *Sem*($\neg$ x>2, zust) oder *Sem*(y>4, zust)

  = nicht *Sem*(x>2, zust) oder Sem(y, zust) > Sem(4, zust)

  = nicht Sem(x, zust) > Sem(2, zust) oder zust(y) > 4

  = nicht zust(x) > 2 oder 5 > 4

  = nicht 3 > 2 oder wahr

  = nicht wahr oder wahr = falsch oder wahr = wahr

Zusicherungen als Zustandsmengen

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

Definition (Zusicherungen als Zustandsmengen): Jede Zusicherung p steht anschaulich für alle Zustände, die diese Zusicherung erfüllen. Die Menge wird mit *Sem*(p) bezeichnet und ist durch

Sem(p) = { zust | *Sem*(p, zust) ist wahr}

also die Menge aller Zustände zust, mit denen p nach wahr ausgewertet wird, definiert.

- *Sem*(x>2 $\rightarrow$ y>4) = {zust| zust(x) $\leq$ 2} $\cup$ {zust | zust(y)>4}
  // andere Variable jeweils egal
- Statt p wird im Zustand zust nach wahr ausgewertet, schreiben wir auch vereinfachend zust *erfüllt* p.
- Um einige Klammern einzusparen, wird vereinbart, dass folgende Prioritäten genutzt werden. Die Prioritäten sind von hoch nach niedrig: $\neg$ $\wedge$ $\vee$ $\rightarrow$ $\leftrightarrow$

  Dies bedeutet, dass ein Operator mit hoher Priorität vor einem Operator mit niedriger Priorität ausgewertet wird.

Substitution

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

Definition (Substitution): Sei p eine Zusicherung, dann ist eine *Substitution* p[x:=w] dadurch definiert, dass alle Vorkommen von x in p durch w ersetzt werden.

Beispiele: Sei p die Zusicherung $x>2 \land y<4$. Dann kann man folgende Berechnungen anstellen

p[x:=z+2] ergibt $z+2>2 \land y<4$, was äquivalent zu $z>0 \land y<4$ ist

p[x:=y-3] ergibt $y-3>2 \land y<4$, was äquivalent zu $y>5 \land y<4$, was äquivalent zu false ist

p[y:=3] ergibt $x>2 \land 3<4$, was äquivalent zu $x>2$ ist.

Dabei bedeutet „äquivalent zu“, dass die Zusicherungen p und q die gleiche Semantik, also die gleichen Zustände die Zusicherungen erfüllen, haben. Dies wird auch $p \equiv q$ geschrieben.

Informelle Semantik von Programmen

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Anschaulich startet ein Programm mit einem Zustand zust1 und endet (falls es terminiert) im Zustand zust2
- Genauer kann man für jeden Programmschritt beschreiben, wie ein Eingangszustand in einen Folgezustand transformiert wird, genauer: Programm im Zustand zust1 macht einen Schritt in (Rest)-Programm im Zustand zust2
- Idee auf Zusicherungen übertragbar: {p} Prog {q}: Wenn ein Zustand die Zusicherung p erfüllt und Prog ausgeführt wird, landet man in einem Zustand, der q erfüllt
- kann auch als

  {Vorbedingung} Programm {Nachbedingung}

  geschrieben werden und wird *Hoare-Tripel* genannt

Divergenz

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
erg := 1;
while(x > 0){
  erg := erg * 2;
  x := x + 1;
}
```

- Programm terminiert nicht, z.B. für zust(x)=1
- „nicht terminierend“ heißt *divergierend* (Nomen: Divergenz)

- Divergenz kann Fehler bedeuten, muss es nicht, z. B. Steuerungssystem
- Programmsemantiken können Divergenz berücksichtigen

Partielle Semantik eines Programms

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

Definition (partielle Semantik): Gegeben sei ein Programm Prog und ein Zustand zust$\in$Zust(Prog). Dann ist die *partielle Semantik* definiert als

SemPart(Prog, zust) = { zz | <Prog, zust> $\to$* <Ende, zz> }

also zz wird erreicht, wenn Prog terminiert

Sei weiterhin eine Zusicherung p gegeben, dann kann die partielle Semantik auch auf p erweitert werden:

SemPart(Prog, p) = { zz | $\exists$ zust1 $\in$ Zust(Prog)

Sem(p, zust1) und <P, zust1> $\to$* <Ende, zz> }

also es gibt einen Zustand zust1, der p erfüllt und zz wird erreicht, wenn Prog gestartet mit zust1 terminiert

- Da Zustände maximal einen Nachfolger haben, ist die resultierende Menge SemPart(Prog, zust) leer oder enthält ein Element

Was heißt Korrektheit?

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Forderung „Programm soll korrekt sein“ macht wenig Sinn, evtl.:
  - Programm soll terminieren
  - Programm soll Rechner nicht zerstören
  - Programm soll private Daten schützen

- Für Korrektheit wird präzise Anforderung verlangt

- Anforderungen sollten die Form haben:

  Unter der folgenden Bedingungen wird erwartet, dass folgendes passiert und das Ergebnis die folgende Form hat

Korrektheit genauer

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- beschreibe Vorbedingung als Zusicherung p
- beschreibe Nachbedingung als Zusicherung q
- Programm Prog ist korrekt, wenn jeder Zustand, der p erfüllt, durch Ausführung von Prog in einen Zustand am Ende transformiert wird, der q erfüllt.

Beispiel:

- Zusicherung x ist positiv: $p \equiv x>0$
- Nachbedingung erg enthält am Ende x^2: $q \equiv erg=x^2$
- Ein Programm Prog, das x nicht verändert (wieso, s. später!) und am Ende in erg den quadrierten Wert enthält ist korrekt bezüglich der Vorbedingung und Nachbedingung
- Beispiel: Wenn Prog startet mit zust1(x)=2 und endet mit zust2(x)=2 und zust2(erg)=4, *kann* Prog korrekt sein
- späteres Ziel: Beweise, dass Prog korrekt bezüglich aller Zustände ist, die die Vorbedingung erfüllen

Partielle Korrektheit

Definition (partielle Korrektheit): Ein Programm Prog heißt *partiell korrekt* bezüglich einer als Zusicherung p formulierten Vorbedingung und einem als Zusicherung q formulierten Nachbedingung als gewünschtem Ergebnis, geschrieben als *Korrektheitsformel* {p} Prog {q}, wenn die Programmausführung jeden Zustand der p erfüllt in einen Zustand transformiert, der q erfüllt.

$\forall$ zz $\in$ SemPart(Prog, p): Sem(q, zz) = wahr

In {p} Prog {q} wird p *Vorbedingung* und q *Nachbedingung* genannt. Man sagt auch, dass dann die Korrektheitsformel {p} Prog {q} (für partielle Korrektheit) gilt, wird geschrieben als

$\models_{part}$ {p} Prog {q}

Anforderungen als Korrektheitsformeln

Video

- Programm Prog soll x quadrieren
  - $\{x>0\}$ Prog $\{erg=x^2\}$
  - problematisch, wenn x in Prog verändert wird
  - $\{x>0\}$ **`x:=0; erg:=0;`** $\{erg=x^2\}$ ist Korrektheitsformel
  - besser

    $\{y=x \wedge x>0\}$ Prog $\{erg=y^2\}$ und $y \notin Var(Prog)$

- Nachweis mit Semantik möglich, aber sehr aufwändig

Totale Semantik eines Programms

Definition (totale Semantik): Gegeben sei ein Programm Prog und ein Zustand zust$\in$Zust(Prog). Dann ist die *totale Semantik* definiert als

$$\mathrm{Sem}(\mathrm{Prog}, \mathrm{zust}) = \{ \mathrm{zz} \mid \langle \mathrm{Prog}, \mathrm{zust} \rangle \to^* \langle \mathrm{Ende}, \mathrm{zz} \rangle \}$$

$$\cup \{ \mathrm{diver} \mid \text{Prog divergiert von z aus} \}$$

Sei weiterhin eine Zusicherung p gegeben, dann kann die totale Semantik auch auf p erweitert werden:

$$\mathrm{Sem}(\mathrm{Prog}, p) = \{ \mathrm{zz} \mid \exists\, \mathrm{zust1} \in \mathrm{Zust}(\mathrm{Prog})\ \mathrm{Sem}(p, \mathrm{zust1}) \text{ und } \langle \mathrm{Prog}, \mathrm{zust1} \rangle \to^* \langle \mathrm{Ende}, \mathrm{zz} \rangle \}$$

$$\cup \{ \mathrm{diver} \mid \exists\, \mathrm{zust1} \in \mathrm{Zust}(\mathrm{Prog})\ \mathrm{Sem}(p, \mathrm{zust1}) \text{ und Prog divergiert von zust1 aus} \}$$

- spezielles Element diver (auch $\perp$ geschrieben, das so genannte Bottom-Symbol), das in keiner Menge sonst vorkommt, genutzt falls Prog divergieren kann

Totale Korrektheit

Definition (totale Korrektheit): Ein Programm Prog heißt *total korrekt* bezüglich einer als Zusicherung p formulierten Vorbedingung und einem als Zusicherung q formulierten gewünschten Ergebnis, geschrieben als *Korrektheitsformel* {p} Prog {q}, wenn die Programmausführung jeden Zustand, der p erfüllt, in einen Zustand transformiert, der q erfüllt. Formal muss gelten:

$$\forall\ zz \in \text{Sem(Prog, p): Sem(q, zz) = wahr}$$

In {p} Prog {q} wird p *Vorbedingung* und q *Nachbedingung* genannt. Man sagt auch, dass dann die Korrektheitsformel {p} Prog {q} (für totale Korrektheit) gilt, geschrieben: $\models$ {p} Prog {q}

- Totale Korrektheit: Prog terminiert garantiert
- Man beachte: zz kann diver sein, da aber Sem(q, diver) undefiniert (nicht wahr) ist, fordert Korrektheitsformel die Terminierung

Beispiel: Korrektheitsbetrachtung


```
erg := 0;  // Programm Prog
while(not(x == 0)){
  erg := erg + 2;
  x:= x - 2;
}
```

- totale Korrektheit:
  - $\{x=2 \lor x=4\}$ Prog $\{erg=2 \lor erg=4\}$
  - $\{(x=2 \lor x=4) \land y=x\}$

    Prog

    $\{(y=2 \land erg=2) \lor (y=4 \land erg=4)\}$
- nur partielle Korrektheit
  - $\{y=x\}$ Prog $\{erg=y\}$

- Forderung nach Terminierung unter Vorbedingung p:

  totale Korrektheit von $\{p\}$ Prog $\{true\}$

Idee: Beweissysteme

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Beweise in der Semantik zu rechnen ist sehr aufwändig, viele Schritte wiederholen sich
- Frage: Gibt es Beweisstrukturen, von denen man einmal zeigt, dass sie korrekt sind, die man immer wieder anwenden kann
- Lösung: *Beweissystem*
- typische Form einer Beweisregel:
  wenn P bewiesen, dann auch Q bewiesen $\frac{P}{Q}$
- Beispiel: Logik (Aussagenlogik), Komma für und

$$\frac{A \rightarrow B,\ B \rightarrow C}{A \rightarrow C} \qquad \frac{A}{A \vee B} \qquad \frac{A\ ,\ B}{A \wedge B} \qquad \frac{A \wedge B}{B \wedge A} \qquad \frac{A \vee B}{B \vee A}$$

$$\frac{A \rightarrow B,\ A}{B} \qquad \frac{A \rightarrow B,\ \neg B}{\neg A} \qquad \frac{A \vee B,\ \neg B}{A}$$

Beispiel: Anwendung des Beweissystems

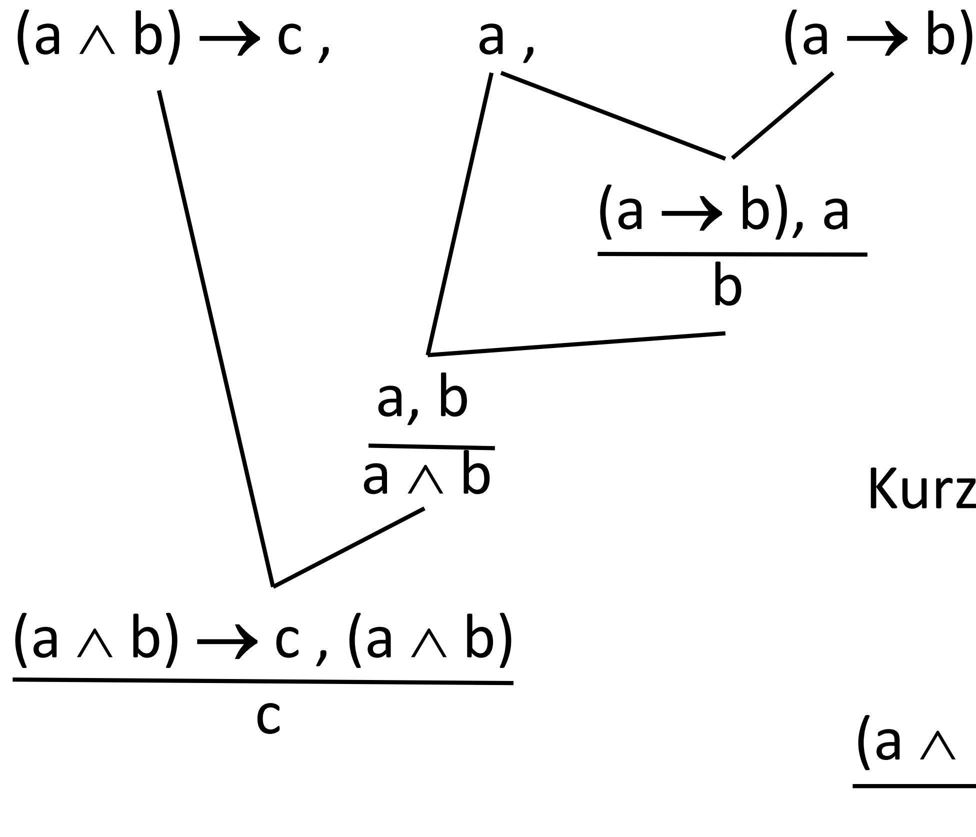
- ohne Beweissystem / Rechenregeln müssten Überprüfungen über Wertetabellen erfolgen (n Variablen, 2^n mögliche Belegungen)
- beweise: $(((a \land b) \to c) \land a \land (a \to b)) \to c$

a	b	c	$a \land b$	P1: $(a \land b) \to c$	P2: $a \to b$	$P1 \land a \land P2$
0	0	0	0	1	1	0
0	0	1	0	1	1	0
0	1	0	0	1	1	0
0	1	1	0	1	1	0
1	0	0	0	1	0	0
1	0	1	0	1	0	0
1	1	0	1	0	1	0
1	1	1	1	1	1	1

- gezeigt: $\models (((a \land b) \to c) \land a \land (a \to b)) \to c$

Beispiel: Anwendung Beweissystem

- ist im Wesentlichen „nur“ andere Notation



Kurzschreibweise

$$\dfrac{(a \wedge b) \rightarrow c, \quad \dfrac{a, \quad \dfrac{(a \rightarrow b), a}{b}}{(a \wedge b)}}{c}$$

gezeigt: $\vdash (((a \wedge b) \rightarrow c) \wedge a \wedge (a \rightarrow b)) \rightarrow c$

$\vdash$ bedeutet: die Formel ist im Beweissystem bewiesen (gültig)

Korrektheit und Vollständigkeit

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- $\models$: das direkte Rechnen mit der Semantik kann sehr aufwändig sein; oft werden auch „argumentative“ (nicht wirklich) formale Beweise geführt
- $\vdash$: die Berechnung mit Beweissystemen kann oft einfacher sein
- es gilt die Forderung nach Korrektheit des Beweissystems

  aus $\vdash$ Korrektheitsformel folgt $\models$ Korrektheitsformel
- es gilt der Wunsch nach Vollständigkeit des Beweissystems

  aus $\models$ Korrektheitsformel folgt $\vdash$ Korrektheitsformel
- Satz: Das vorgestellte Beweissystem für die Aussagenlogik ist korrekt und vollständig
- genauer betrachtet ist es nicht minimal, es können auch andere Regelsammlungen für „korrekt und vollständig“ genutzt werden

Beweissystem für partielle Korrektheit

(1) Zuweisung: $\{p[x{:=}y]\}$ `x := y;` $\{p\}$

(2) Komposition:

$$\frac{\{p\}\ \text{Prog1}\ \{q\},\ \{q\}\ \text{Prog2}\ \{r\}}{\{p\}\ \text{Prog1 Prog2}\ \{r\}}$$

(3) Alternative:

$$\frac{\{p \land B\}\ \text{Prog1}\ \{q\},\ \{p \land \neg B\}\ \text{Prog2}\ \{q\}}{\{p\}\ \texttt{if (B) \{Prog1\} else \{Prog2\}}\ \{q\}}$$

(4) Schleife:

$$\frac{\{p \land B\}\ \text{Prog}\ \{p\}}{\{p\}\ \texttt{while (B) \{Prog\}}\ \{p \land \neg B\}}$$

(5) Konsequenz:

$$\frac{p \rightarrow p1,\ \{p1\}\ \text{Prog}\ \{q1\},\ q1 \rightarrow q}{\{p\}\ \text{Prog}\ \{q\}}$$

mit Beweissystem gezeigter Beweis geschrieben: $\vdash_{part} \{p\}\ \text{Prog}\ \{q\}$

HOCHSCHULE OSNABRÜCK UNIVERSITY OF APPLIED SCIENCES

Beispielhafte Analyse der Zuweisungsregel

- zu zeigen: { y=4 } `x := 3;` { x=3 $\wedge$ y=4 }
  - (x=3 $\wedge$ y=4) [x:=3] $\equiv$ (3=3 $\wedge$ y=4) $\equiv$ y=4 (Zuweisungsregel und logische Umformung)
  - es gilt auch { x=3 $\wedge$ y=4 } `x := 3;` { x=3 $\wedge$ y=4 }, da (x=3 $\wedge$ y=4) $\rightarrow$ (y=4) folgt Aussage mit Konsequenz-Regel („Verstärkung“ vorne)
- auch { (x=5 $\wedge$ y=4) [x:=3] } `x := 3;` {x=5 $\wedge$ y=4}, umgeformt
  { false } x=3 ; {x=5 $\wedge$ y=4}

Beweis einer Zuweisungskette

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- zu zeigen: {x=y} `x := x * 2; y := y / 2;` {x=4y}

- Aus (x=4y)[y:=y/2] wird x=4(y/2) und damit x=2y
- Es gilt: {x=2y} y=y/2; {x=4y}

- (x=2y)[x:=x*2] wird zu x*2=2y und damit x=y
- Es gilt: {x=y} x=x*2; {x=2y}

- Mit Kompositionsregel folgt Behauptung

Beweis für `if`

zeige:

$\{x>5 \land x<10\}$

```
if(x>8) {x := x - 5;} else {x := x + 3;}
```

$\{x>3 \land x\leq 11\}$

Ansatz mit Zuweisungsregel:

$\{x>3 \land x \leq 11\}[x:=x-5] \equiv \{x>8 \land x \leq 16\}$

Verstärkung vorne: $(x>5 \land x<10 \land x>8) \rightarrow (x>8 \land x \leq 16)$

$\{x>3 \land x\leq 11\}[x:=x+3] \equiv \{x>0 \land x\leq 8\}$

$(x>5 \land x<10 \land x\leq 8) \rightarrow (x>0 \land x\leq 8)$

Beweis einer Schleife (1/2)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
erg := 1;  // Programm Prog
while(not (x == 0)){
  erg := erg * 2;
  x := x - 1;
}
```

- zu zeigen: $\{y=x \wedge x>0\}$ Prog $\{erg=2^y\}$
- generell schwierig: Finden einer *Schleifeninvariante*

  {p} <Schleifenrumpf> {p}

  (genauer gesucht: $\{p \wedge B\}$ <Schleifenrumpf> {p})
- Hier weiß man was rauskommen soll, $\{erg=2^y\}$ nicht als Invariante geeignet
- genereller Ansatz: Schleifen haben meist Schleifenzähler, dieser muss in das gewünschte Ergebnis eingebaut werden hier: $\{erg=2^{y-x}\}$

Beweis einer Schleife (2/2)

- $erg=2^{y-x}$ [x:=x-1] $\equiv$ $erg=2^{y-(x-1)}$ $\equiv$ $erg=2^{1+y-x}$
- $erg=2^{1+y-x}$ [erg:= erg*2] $\equiv$ $erg*2=2^{1+y-x}$ $\equiv$ $erg=2^{y-x}$
- mit Kompositionsregel:
  {$erg=2^{y-x}$} `erg := erg * 2; x := x - 1;` {$erg=2^{y-x}$}
- mit Verstärkung {$erg=2^{y-x} \wedge x \neq 0$} `erg := erg * 2; x := x - 1;` {$erg=2^{y-x}$}
  da : $(erg=2^{y-x} \wedge x \neq 0) \rightarrow erg=2^{y-x}$ (benötigt für Schleifenregel)
- mit Schleifenregel:
  {$erg=2^{y-x}$}
  ```
  while(x != 0){erg := erg * 2; x := x - 1;}
  ```
  {$erg=2^{y-x} \wedge \neg(x!=0)$}
- mit Konsequenzregel wird aus {$erg=2^{y-x} \wedge \neg(x!=0)$} die Zusicherung {$erg=2^{y}$}
- vor der Schleife: Aus ($erg=2^{y-x}$)[erg:=1] wird Zusicherung $1=2^{y-x}$ mit mathematischen Kenntnissen und Verstärkung folgt $y=x \wedge x>0$, der Rest mit Kompositionsregel

Bessere Abbruchbedingung

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Coding-Guideline: Nicht auf Gleichheit bzw. Ungleichheit bei Abbruch testen, hier besser:

```
while (x > 0)
```

- Invariante reicht dann nicht aus, muss zu

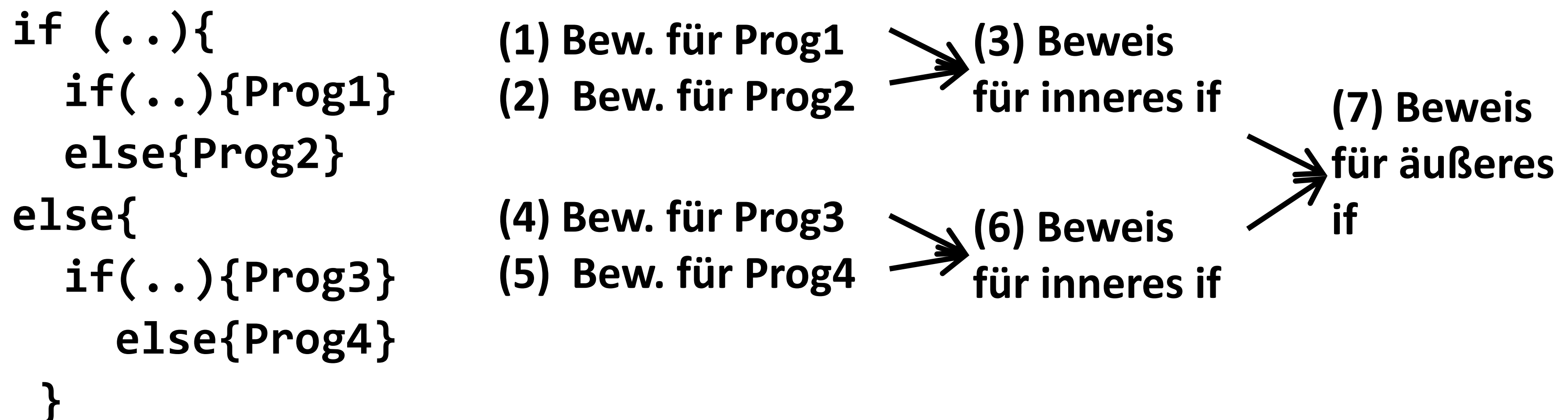
  $erg=2^{y-x} \wedge x>-1$ geändert werden, weiterhin muss x als ganzzahlig bekannt sein

  dann:

  aus $\neg(x>0) \wedge x>-1$ folgt dann $x \leq 0 \wedge x>-1$ und da x ganzzahlig ist $x=0$.

Konzept zur Beweisfindung

- Es gibt kein eindeutiges Kochrezept zur Beweiskonstruktion, aber typisch
  - bei Zuweisungsketten rückwärts rechnen
  - von innen nach außen arbeiten
  - Beweisfindung ist iterativer Prozess, mit Ansatz, Teilerfolg nur für einen Schritt, notwendiger Korrektur, Teilerfolg für größeres Programmstück, notwendige Korrektur, ...
- Beispiel:



Geschachtelte Alternative (Ansatz)

Zeige: $\{x>10 \land x<35\}$ Prog $\{x>12 \land x<25\}$

a1) $\{x>10 \land x<35 \land x>18 \land x<30\}$

```
x := x - 5;
```

$\{x>12 \land x<25\}$

b1) $\{x>10 \land x<35 \land x>18 \land \neg(x<30)\}$

```
x := x - 10;
```

$\{x>12 \land x<25\}$

a2) $\{x>10 \land x<35 \land x>18\}$

```
if(x < 30) {x := x - 5;} else {x := x - 10;}
```

$\{x>12 \land x<25\}$

b2) $\{x>10 \land x<35 \land \neg(x>18)\}$

```
x := x + 4;
```

$\{x>12 \land x<25\}$

```
if(x > 18){
  if(x < 30){
    x := x - 5;
  }
  else{
    x := x - 10;
  }
}
else{
  x := x + 4;
}
```

Rechentricks zur Alternative

zeigt man {p} Prog1 {q1} und {p} Prog2 {q2}

- weiß man, dass nach Prog1 oder Prog2 unter der Vorbedingung p immer $q1 \vee q2$ gilt,

  da $(p \wedge B) \rightarrow p$ und $(p \wedge \neg B) \rightarrow p$ sowie

  $q1 \rightarrow (q1 \vee q2)$ und $q2 \rightarrow (q1 \vee q2)$ gilt

  {p} **if (B) {**Prog1**} else {**Prog2**}** $\{q1 \vee q2\}$

- wenn weiterhin in den Programmen Prog1 und Prog2 die zum Booleschen Ausdruck B gehörenden Variablen nicht verändert werden, gilt

  {p} **if (B) {**Prog1**} else {**Prog2**}** $\{(B \rightarrow q1) \wedge (\neg B \rightarrow q2)\}$

  statt $(B \rightarrow q1) \wedge (\neg B \rightarrow q2)$ kann auch der logisch äquivalente Ausdruck $(B \wedge q1) \vee (\neg B \wedge q2)$ stehen

Überlegungen zur Terminierung

Video

- Betrachten wir folgendes Programm mit der Zusicherung {x>0} (x ganzzahlig); warum terminiert es

```
while(not(x == 0)){
  erg := erg * 2;
  x := x - 1;
}
```

- Anschaulich: x wird immer kleiner, so dass es irgendwann Null werden muss
- Formaler Ansatz: Es gibt eine Funktion t,
  - die einen ganzzahligen Wert liefert,
  - die von den Programmvariablen abhängt (z. B. x),
  - die bei jedem Schleifendurchlauf echt kleiner wird
  - z. B. t=x
- Ganzzahlig, da x auch bei x=x/2 immer echt kleiner wird
- t muss vor der Schleifenausführung einen positiven Wert haben und darf nie negativ werden

Beweissystem für totale Korrektheit (1/2) - übernommen

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

(1) Zuweisung: $\{p[x:=y]\}$ **`x := y;`** $\{p\}$

(2) Komposition:

$$\frac{\{p\}\ \text{Prog1}\ \{q\},\ \{q\}\ \text{Prog2}\ \{r\}}{\{p\}\ \text{Prog1 Prog2}\ \{r\}}$$

(3) Alternative:

$$\frac{\{p \land B\}\ \text{Prog1}\ \{q\},\ \{p \land \neg B\}\ \text{Prog2}\ \{q\}}{\{p\}\ \textbf{if (B) \{}\text{Prog1}\textbf{\} else \{}\text{Prog2}\textbf{\}}\ \{q\}}$$

(5) Konsequenz:

$$\frac{p \rightarrow p1,\ \{p1\}\ \text{Prog}\ \{q1\},\ q1 \rightarrow q}{\{p\}\ \text{Prog}\ \{q\}}$$

Beweissystem für totale Korrektheit (2/2)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

$$\frac{\{p \land B\}\ \text{Prog}\ \{p\} \qquad \{p \land B \land t=z\}\ \text{Prog}\ \{t<z\} \qquad p \rightarrow t \geq 0}{\{p\}\ \texttt{while (B) \{Prog\}}\ \{p \land \neg B\}}$$

dabei ist z eine ganzzahlige Variable, die in Prog nicht vorkommt und t ein beliebiger Ausdruck, der zu einer ganzen Zahl ausgewertet wird

- Regel ersetzt Regel (4) für partielle Korrektheit der Schleife
- t heißt *Terminierungsfunktion*; in jedem Schleifendurchlauf wird der Wert von t mindestens um eins kleiner und ist am Anfang mindestens 0
- mit Beweissystem gezeigter Beweis geschrieben: $\vdash \{p\}\ \text{Prog}\ \{q\}$

Korrektheit und Vollständigkeit

- Satz: Das Beweissystem für die totale Korrektheit ist korrekt und vollständig, wenn es um die Menge aller gültigen Zusicherungen ergänzt wird, etwas formaler:

  aus $\vdash \{p\}\ Prog\ \{q\}$ folgt $\models \{p\}\ Prog\ \{q\}$

  aus $\models \{p\}\ Prog\ \{q\}$ folgt $\vdash \{p\}\ Prog\ \{q\}$

- Erinnerung: wir nutzen mindestens den Typen int und benötigen u. a. in der Konsequenzregel Formeln der Form $p \rightarrow p1$, damit spielt Gödels Unvollständigkeitssatz eine Rolle, deshalb alle Zusicherungen hinzugenommen
- genauer betrachtet reicht die aktuelle Syntax nicht aus, um alle benötigten Terminierungsfunktionen zu beschreiben
- obiger Satz ist auch für das Beweissystem für partielle Korrektheit formulierbar

Anmerkungen zur Terminierungsfunktion

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Wichtig ist, dass sie immer runterzählt und dass sie größer 0 ist, dies muss auch Invariante garantieren
- Programm `while(x > 0){ ...; x := x - 1;}`

  wenn am Anfang x>=0 dann t=x wählen

  wenn x<0 am Anfang y=x (y kommt nicht vor und t= |y|+x wählen
  (| Betrag |)
- Es gibt kein automatisches Verfahren zur Bestimmung einer sinnvollen Invarianten und Terminierungsfunktion (wg. Unentscheidbarkeit)
- Man kann Heuristiken aufstellen, wie man anhand der Schleifenstruktur Invarianten und Terminierungsfunktionen finden kann

- Potenzrechenbeispiel: Invariante erweitern zu

  $\{erg=2^{y-x} \wedge x\geq 0\}$, dann t=x wählbar, wegen

  $\{x=z\}$ `erg := erg * 2; x := x - 1;` $\{x+1=z\}$ und $x+1=z \rightarrow x<z$

formale Beweise

- ein Beweis ist nur formal korrekt, wenn er in einem als korrekt bewiesenen Beweissystem (oder durch Umformungen) entstanden ist
- Umformungen sind formal : z. B. Äquivalenzumformungen für die Aussagenlogik, Berechnungen in Gleichungen
- kritisch: wie wird Beweissystem formal korrekt bewiesen (evtl. in sich selbst, aber was ergibt Fehler auf sich selbst angesetzt)
- gibt Ansätze Beweissysteme als Computer-Programme nutzbar zu machen mit Theorem-Beweisern, Nutzung sehr sehr sehr aufwändig
- auch Theorem-Beweiser können nicht vollständig automatisch arbeiten (müssten sonst Halte-Problem lösen können)

- fast alle „mathematischen Beweise“ in Ihrem Studium basieren auf Pseudo-Formalismen [in dieser VL sonst auch] und sind nicht formal korrekt (Beweisideen)

Reicht Testen nicht aus?

Testüberdeckungen: Gesucht Ansatz Qualität von Tests bewerten

viele Varianten:

- sichere, dass alle Anweisungen ausgeführt worden sind
- sichere, dass alle Alternativen ausgeführt worden sind, bei `if(x>42){…}` muss es Test mit x<=42 und x>42 geben
- sichere, dass in Booleschen Ausdrücken jeder Teilterm t relevant ist (t=true hat anderen Effekt als t=false)
- …

gibt für obere Ansätze Werkzeuge, die während der (Test-)Ausführung automatisch messen

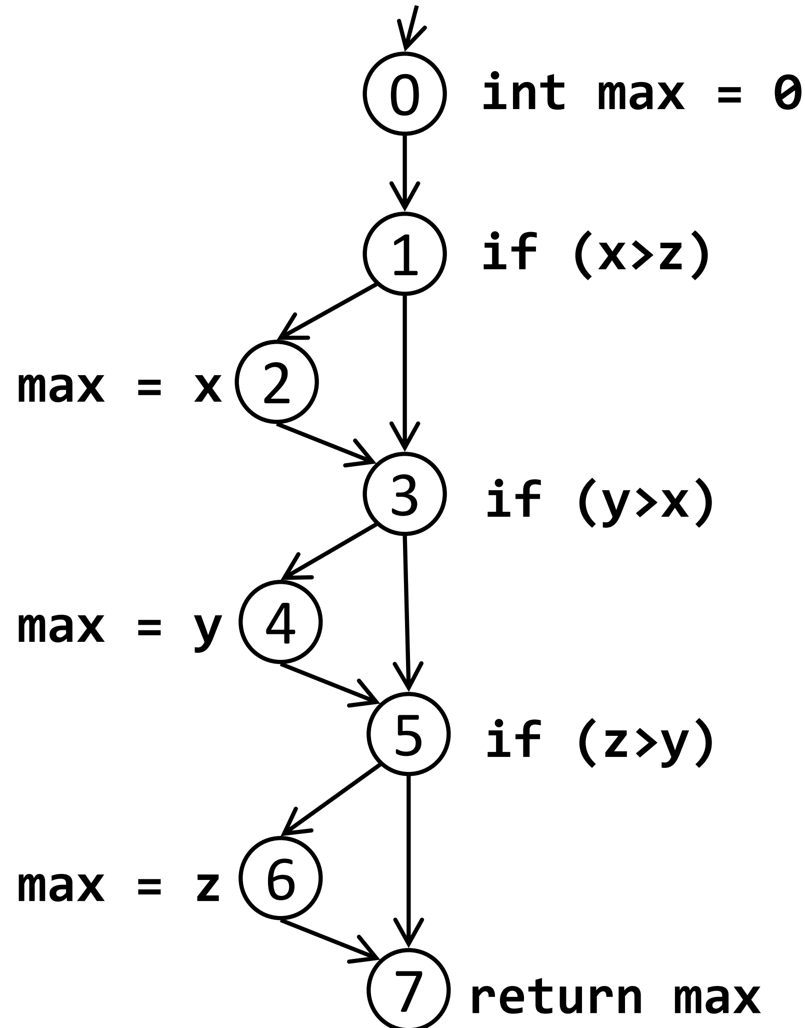
wichtig ist, dass präzise Zusicherungen genutzt werden

Ansatz ist in großen Projekten notwendig, aber nicht hinreichend

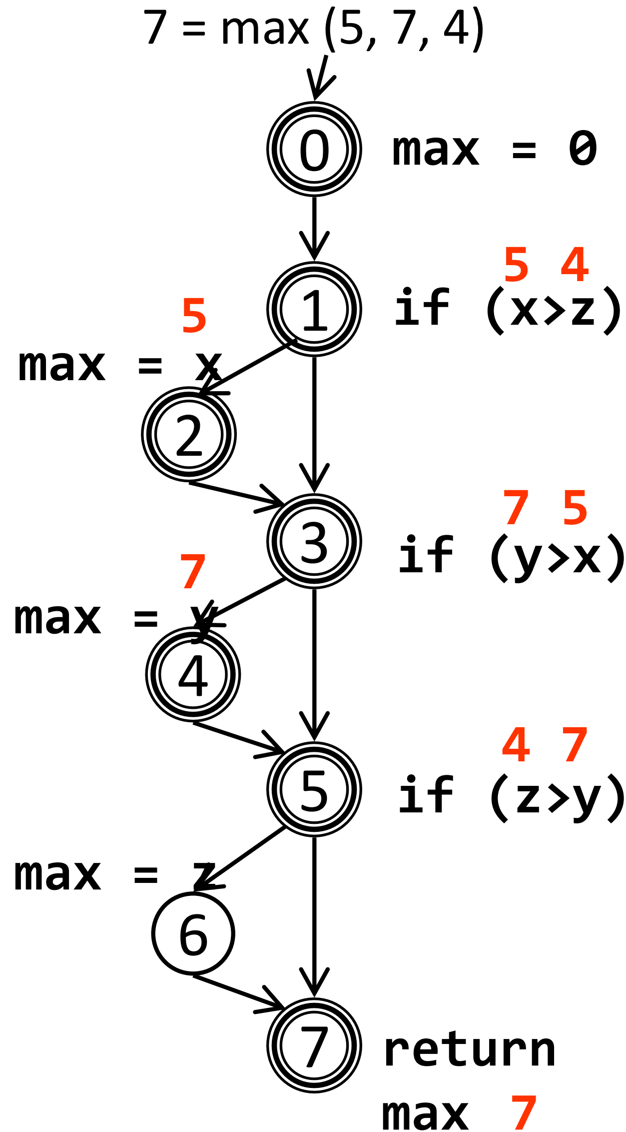
Beispiel Überdeckungen zur Messung der Testqualität

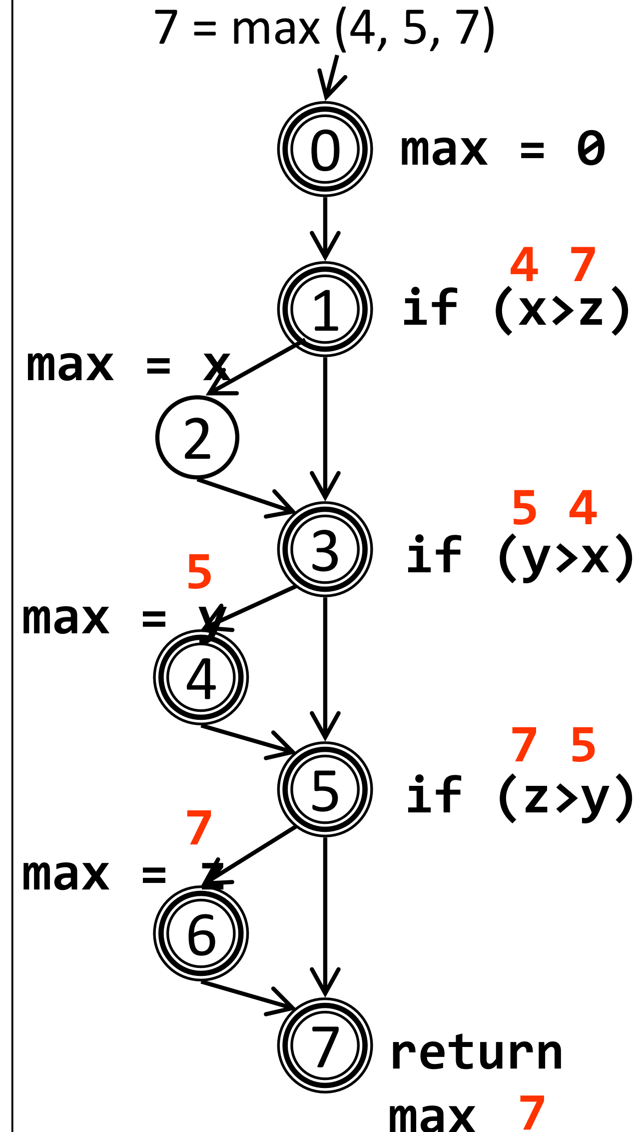
- Suche Maximum von drei ganzen Zahlen

```
public int max(int x,
               int y,
               int z) {
  int max = 0;
  if (x>z) {
    max = x;
  }
  if (y>x) {
    max = y;
  }
  if (z>y) {
    max = z;
  }
  return max;
}
```

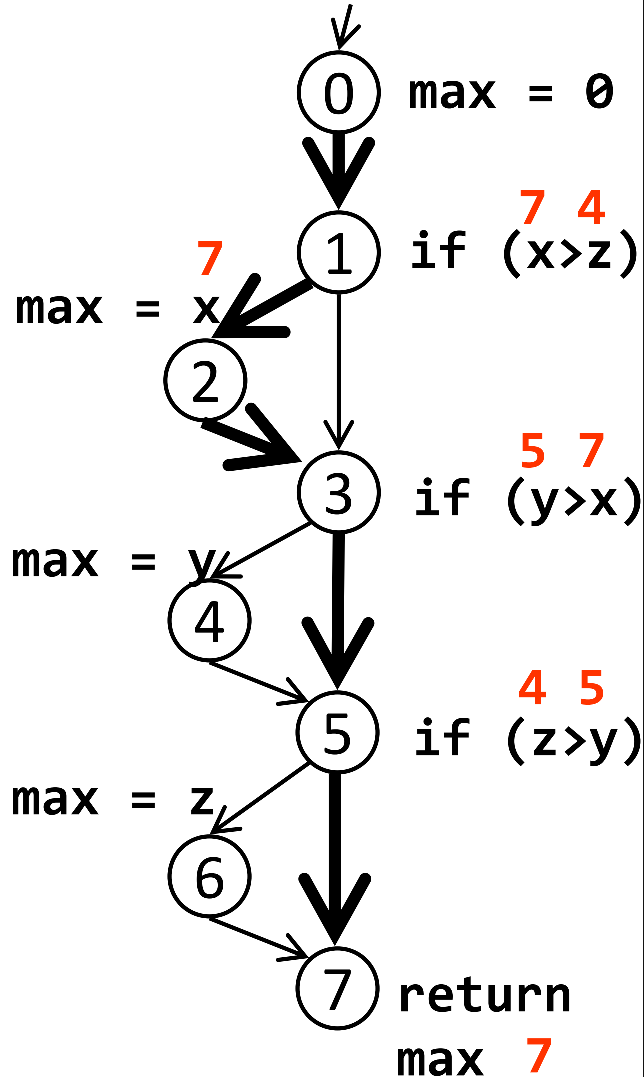


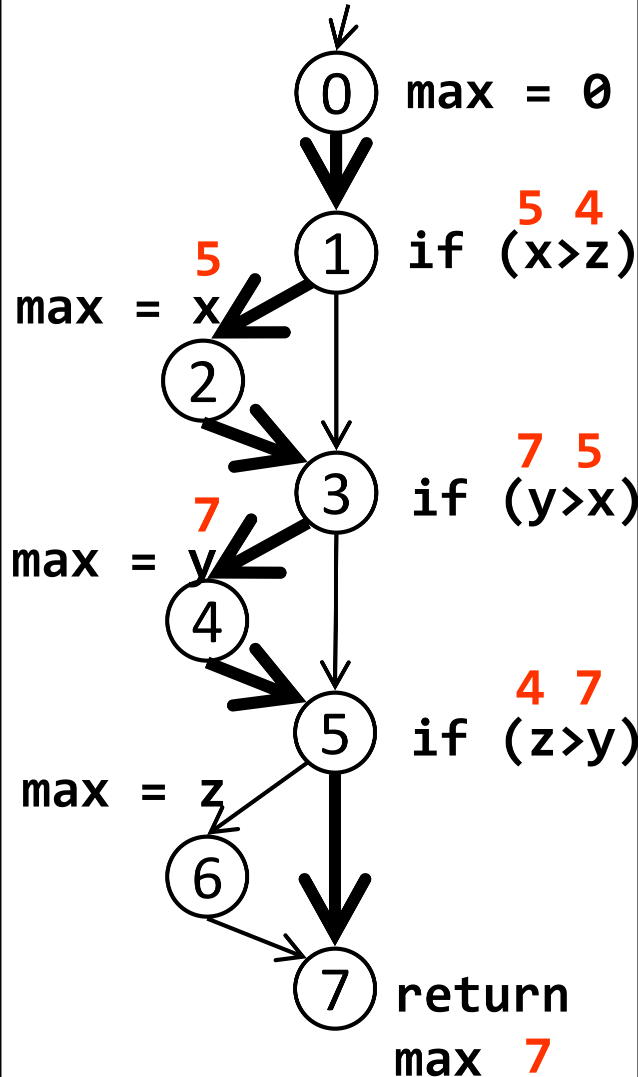
Anweisungsüberdeckung - jeder Knoten einmal

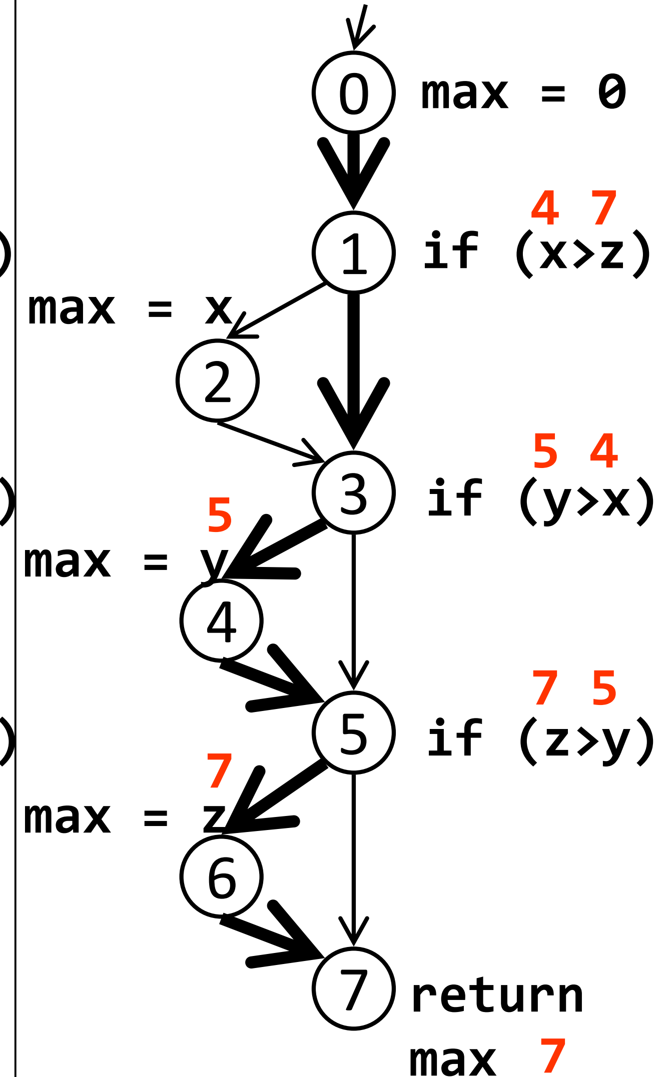




Zweigüberdeckung - jede Kante einmal







Beweisskizze

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

$\{p_0\}$
$Prog_1$
$\{q_1\}$
$\{p_1\}$
$Prog_2$
$\{q_2\}$
⋮
$\{q_{n-1}\}$
$\{p_{n-1}\}$
$Prog_n$
$\{q_n\}$

typisches Beweisergebnis für Programm Prog1;Prog2;...;Progn auf rechter Seite, es gilt

- $\{p_{i-1}\}$ $Prog_i$ $\{q_i\}$ korrekte Beweisskizze
- $q_i \rightarrow p_i$

- vereinfachte Darstellung links nennt sich Beweisskizze
- für Schleifen werden
  - {inv: <Zusicherung>} Invariante
  - {bd : <Ausdruck>} Terminierungsfunktion

angegeben

$\{p_0\}$
$Prog_1$
$\{p_1\}$
$Prog_2$
$\{p_2\}$
⋮
$\{p_{n-1}\}$
$Prog_n$
$\{p_n\}$

$(p_n=q_n)$

Beispiel: Beweisskizze (sequenzielles Programm)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

$\{y=x \wedge x>0\}$

`erg := 1;`

$\{inv: x\geq 0 \wedge erg=2^{y-x}\}$ $\{bd:x\}$

`while(x != 0){`

$\{x\geq 0 \wedge x\neq 0 \wedge erg=2^{y-x}\}$

`erg:= erg * 2;`

$\{x\geq 1 \wedge erg=2^{1+y-x}\}$

`x := x - 1;`

$\{x\geq 0 \wedge erg=2^{y-x}\}$

`}`

$\{erg=2^{y}\}$

Programm mit Beweisskizze verknüpft

```
public int zweierPotenz(int x){
  int y = x; // neue Hilfsvariable
  int z; // Hilfsvariable fuer Terminierungsfunktion
  int erg;
  assert(y == x && x > 0);
  erg = 1;
  assert(x >= 0 && erg == (int)Math.pow(2, y - x));
  while(x != 0){
    z = x; // Terminierungsfunktion initialisieren
    assert(x >= 0 && x != 0 && erg == (int)Math.pow(2, y - x));
    erg = erg * 2;
    assert(x >= 1 && erg == (int)Math.pow(2, 1 + y - x));
    x = x - 1;
    assert(x >= 0 && erg == (int)Math.pow(2, y - x));
    assert(x < z); // für Terminierungsfunktion
  }
  assert(erg == (int)Math.pow(2, y));
  return erg;
}
```

Beweisskizzen und Testen

- Die Ergänzung von Zusicherungen ermöglicht es, das laufende Programm zu testen
- Wichtig wäre, dass alle Zusicherungen in unterschiedlichen Situationen durchlaufen werden; dies ist mit zufälligen Tests kaum möglich

- Ansatz zur systematischen Nutzung von Zusicherungen:
  - Äquivalenzklassenansatz zur systematischen Bestimmung sinnvoller Eingaben
  - Nutzung von Grenzwerten um kritische Werte
  - Überdeckungsmessung zur Feststellung, ob alle Programmteile ausgeführt wurden

Gezielte Suche nach Fehlern

- Ziel des Testens ist das Finden von Fehlern und nicht die Aussage, dass die Software keine Fehler enthält
- Schlussrichtung: korrekt bewiesen => keine Fehler
- aber es gilt nicht der Umkehrschluss:
  keine Fehler gefunden => korrekt

- Testen bedeutet die Suche nach Fehlern, ein Testfall ist erfolgreich, wenn er einen Fehler gefunden hat
- Verifikation bedeutet den Nachweis der Fehlerfreiheit

- Die Fragen: Wer testet den Tester, wer verifiziert den Verifizierer bleiben im Raum

Praxisthemen für Verifikation

- Einführung von Datentypen
- Einführung von Funktionen, Methoden, Objektorientierung
- Einführung von Sichtbarkeiten, lokale und globale Variablen
- Wertübergabe: Call by Value und Call by Reference
- Polymorphie bei Vererbung (Typbestimmung zur Laufzeit)
- Kommunikation im System
- …

kann alles in Semantik eingebaut werden, wobei eventuell nicht mehr der operationelle Ansatz gewählt wird

Programmiersprachbesonderheiten

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
public static void main(String[] args){
   int imax=Integer.MAX_VALUE;
   int imax1=imax+1;
   double dmax=Double.MAX_VALUE;
   double dmax1=dmax*100.;
   double dmin1=-dmax1;
   double aha1=dmax1/dmax1;
   double aha2= 3./0.;
   System.out.println("imax: "+imax);
   System.out.println("imax1: "+imax1);
   System.out.println("dmax: "+dmax);
   System.out.println("dmax1: "+dmax1);
   System.out.println("dmin1: "+dmin1);
   System.out.println("aha1: "+aha1);
   System.out.println("aha2: "+aha2);
}
```

```
imax: 2147483647
imax1: -2147483648
dmax: 1.7976931348623157E308
dmax1: Infinity
dmin1: -Infinity
aha1: NaN
aha2: Infinity
```

Programmverifikation in der Praxis

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Automatische Überprüfung, ob Zusicherungen korrekt gewählt wurden, ist nicht möglich, da i.a. unentscheidbar
- Korrekte Regelanwendung kann durch Werkzeug (*Theorembeweiser*) kontrolliert werden

falls nicht verifiziert wird:

- Vor- und Nachbedingungen sind für jede Methode festzulegen und zu prüfen, C, C++, C# und Java haben dazu Befehl **`assert`**
- assert(<Boolesche Bedingung>) bricht Programm ab, wenn Bedingung nicht erfüllt ist
- sauberer Programmierstil: einfache kurze Methoden, z. B. niedrige McCabe-Zahl (Anzahl von Verzweigungen in der Methode)
- komplexe Algorithmen sind zu dokumentieren, damit ein Anderer sie verstehen (prüfen) kann
- jede Anweisung beim Testen mindestens einmal ausführen
- bei Testerstellung über Grenzfälle genau nachdenken

Fazit

- für jede Programmiersprache mit eindeutiger Grammatik ist eine eindeutige formale Semantik definierbar
- Schritte eines Programms führen zu Zustandsänderungen
- auf Basis der Semantik können Programmeigenschaften, also die Erfüllung von Anforderungen formal bewiesen werden (sehr aufwändig, aber machbar)
- Beweissysteme können formale Beweise einfacher machen und deutlich abkürzen
- Tests können (praktisch) nie Fehlerfreiheit garantieren
- Beweisinformationen können in systematische Testentwicklung einfließen
- fehlt: man kann verschiedene sinnvolle Semantiken definieren
- fehlt: wenn alle Typen endlich sind, ist ein endliches System (theoretisch) vollständig überprüfbar

Video

4. Endliche Automaten und reguläre Ausdrücke

zentrale Inhalte:

- Varianten von endlichen Automaten und akzeptierte Sprachen
- interne Übergänge (ε-Übergänge)
- reguläre Ausdrücke

Glossar

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

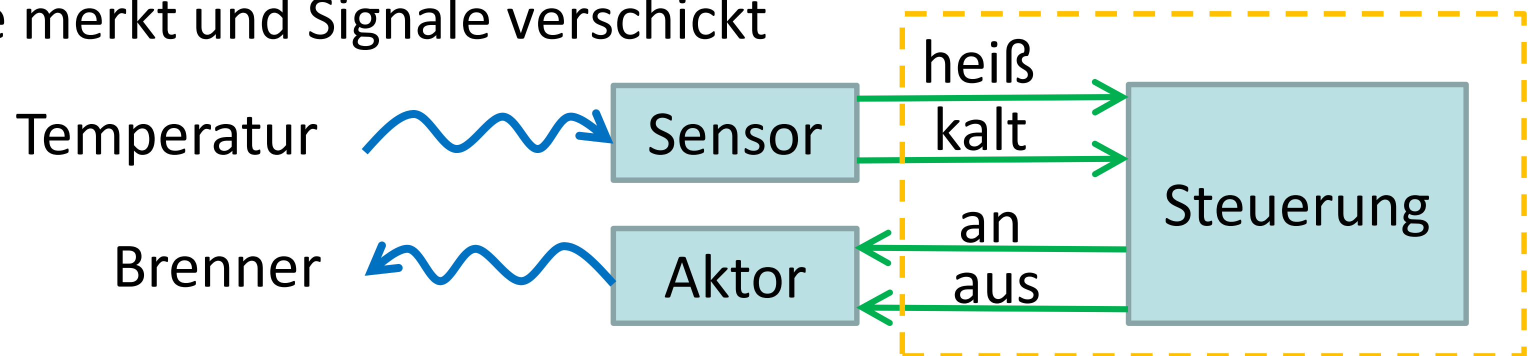
- äquivalente Automaten (sprachliche)
- äquivalente Zustände
- akzeptierte Sprache eines Automaten
- Automat in rechtslineare Grammatik
- Automat in regulären Ausdruck
- deterministischer Automat
- ε-Abschluss
- ε-Überführungsfunktion
- ε-Übergänge entfernen
- erweiterte Überführungsfunktion
- Generierung von Automaten
- Minimierung von Automaten
- Nichtdeterminismus entfernen
- nichtdeterministischer Automat
- nichtdeterministischer Automat mit ε-Übergängen
- Pumping-Lemma für reguläre Sprachen
- Rechtslineare Grammatik
- Rechtslineare Grammatik in Automaten
- reguläre Ausdrücke
- reguläre Ausdrücke in Automaten
- reguläre Ausdrücke in Java
- Statecharts
- Überführungsfunktion
- Zustandsdiagramm

Motivation

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Viele Steuerungssysteme bestehen aus
  - Sensoren, die Werte der Umgebung (oder Eingaben) erkennen, die Signale an eine Steuerung schicken
  - Aktoren, die die Umgebung verändern können und durch Signale gesteuert werden
  - Steuerung, die Signale annimmt, sich intern Zustände merkt und Signale verschickt

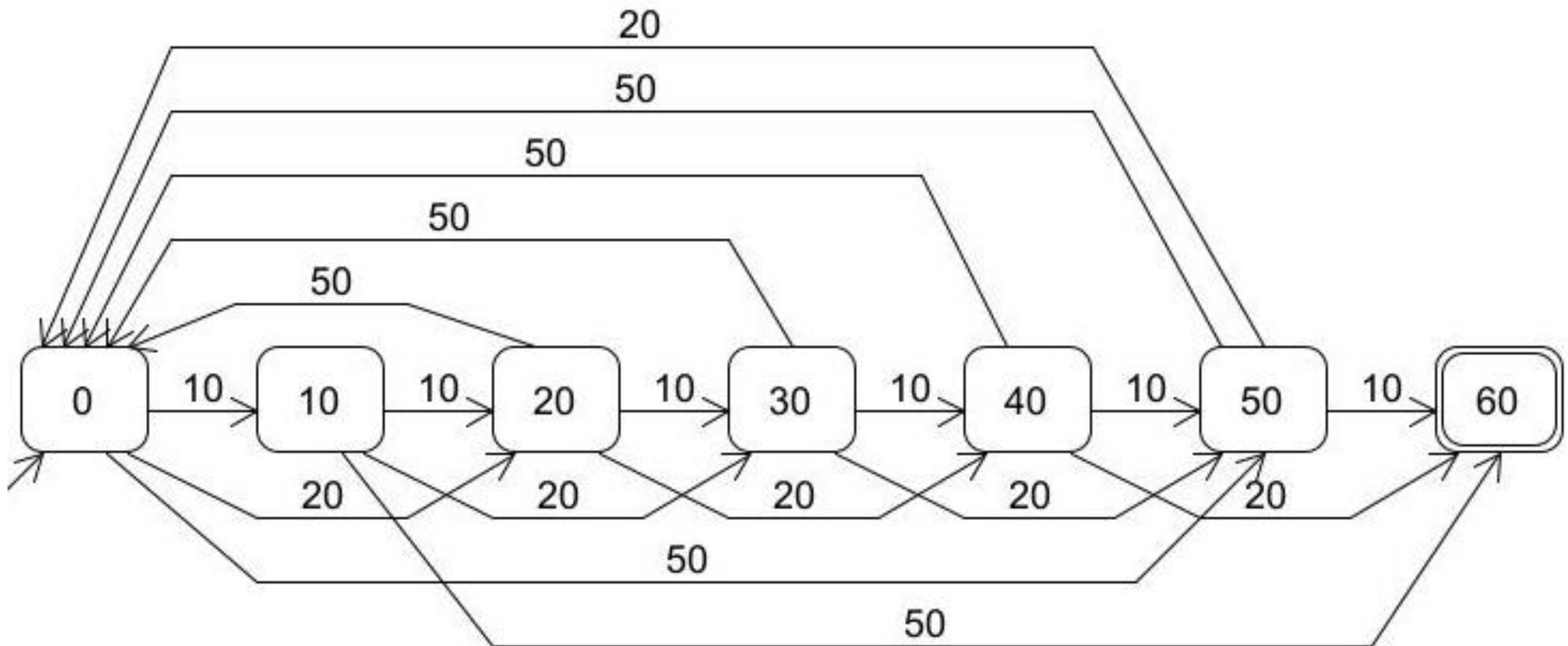
Wir klären dann einen interessanten Teilbereich für den diese Verifikation nicht nur von Hand sondern vollständig automatisch durchführbar ist.



- auch in reiner Software, z. B. Events einer Android-App
- endliches Modell mit relativ einfachen Schleifen; fast alle Fragen sind entscheidbar; wir können automatisch Anforderungen beweisen

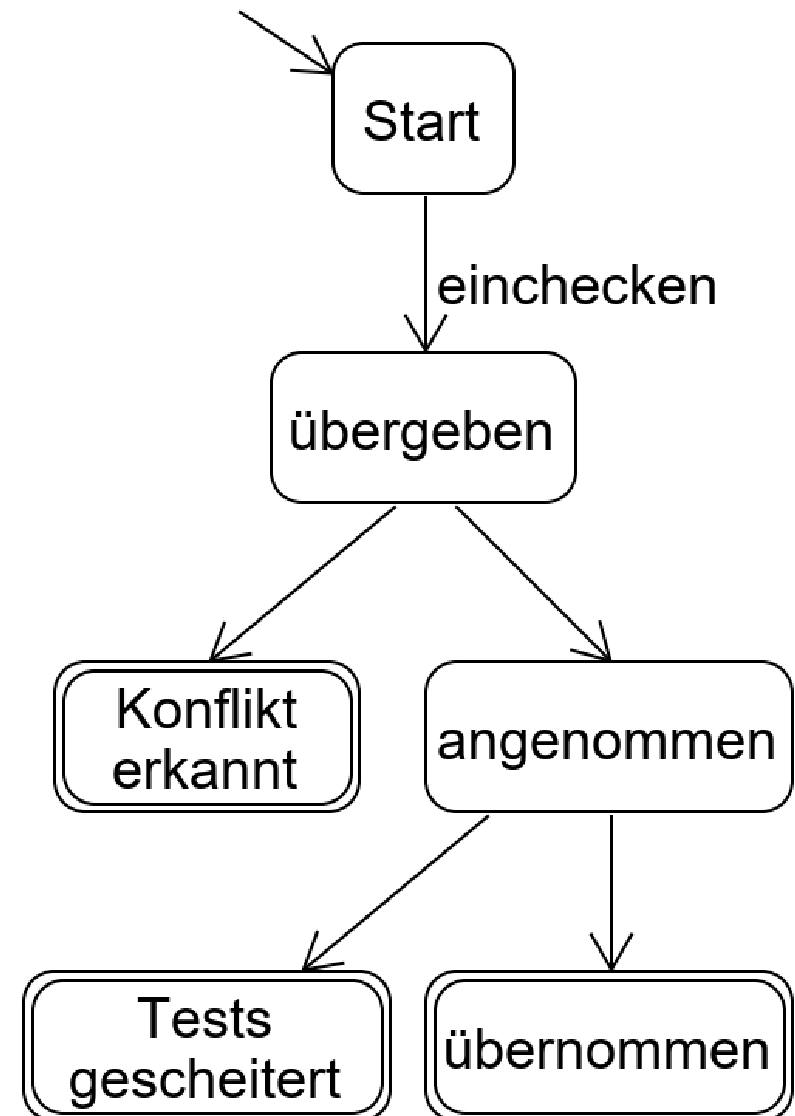
Beispiel: Verkaufsautomat für 60 Cent

- Eingabe-Signale: 10, 20, 50
- Ausgabe nicht sichtbar, aber einbaubar: Rückgabe bei Rücksprung auf 0, Produktausgabe am Ende



interne Übergänge

- Verwendung Versionsmanagement
- ein Steuersignal: einchecken
- dann Übergang „übergeben“
- weitere Übergänge hängen nicht von Nutzung ab; sind aus Nutzungssicht interne Signale
- bei Nutzung werden erreichte Zustände mitgeteilt
- in Endzuständen kann weitere Nutzung stattfinden
- typisch für reine Client-Sicht



Nichtdeterministischer Automat mit ε-Übergängen

Definition: Ein *nichtdeterministischer Automat mit ε-Übergängen* ist ein 6 -Tupel (Alphabet, Zustände, Endzustände, *Überführungsfunktion*, *ε-Überführungsfunktion*, Startzustand) mit

- Alphabet, wie bekannt, eine endliche nicht leere Menge von Zeichen
- Zustände, eine endliche nicht leere Menge von Zuständen
- Endzustände, Endzustände $\subseteq$ Zustände
- Überführungsfunktion: Zustände $\times$ Alphabet -> Pot(Zustände)
- ε-Überführungsfunktion: Zustände -> Pot(Zustände)
- Startzustand $\in$ Zustände

- Anmerkung: statt 2 Überführungsfunktionen kann äquivalent auch nur eine genutzt werden: Zustände $\times$ (Alphabet $\cup$ {ε}) -> Pot(Zustände)

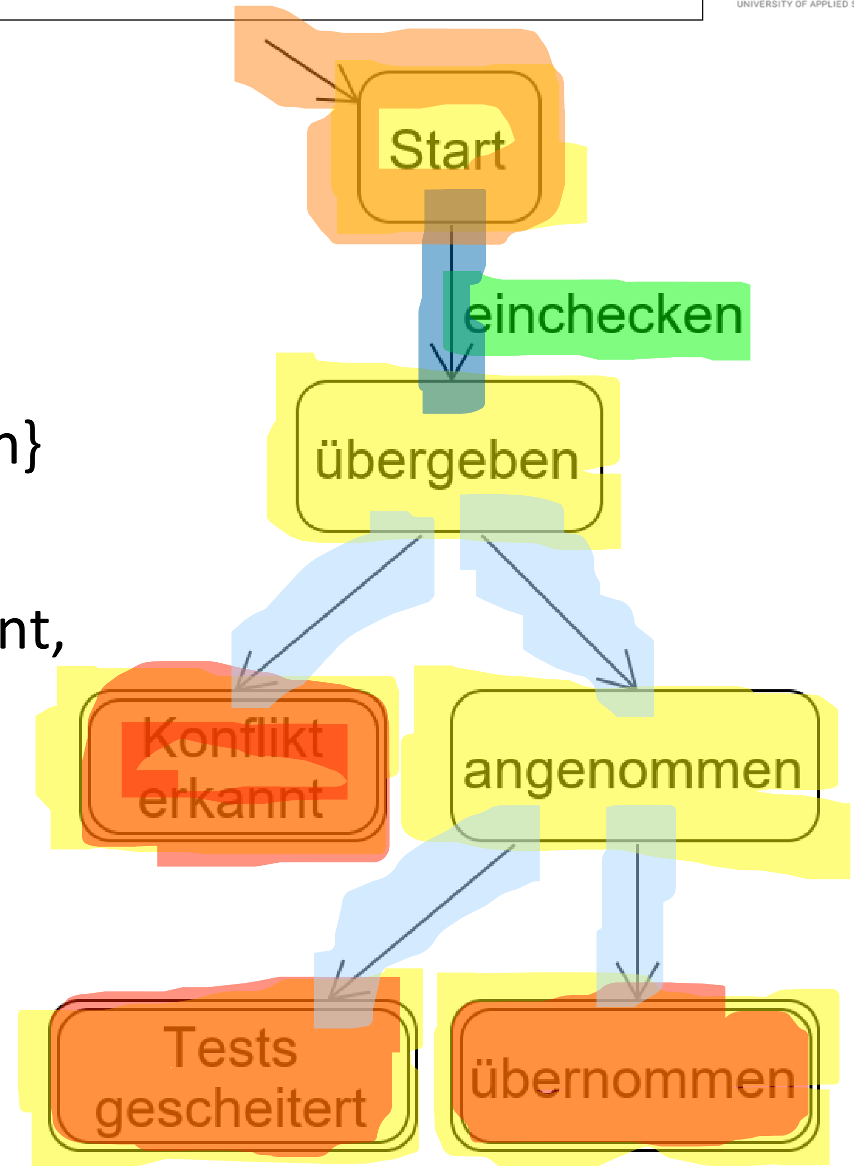
Visualisierung von Automaten

- Alphabet
- Zustände
- Endzustände
- Überführungsfunktion

  über(Start, einchecken) = {übergeben}
- ε-Überführungsfunktion

  ε- über(übergeben) = {Konflikt erkannt, angenommen}
- Startzustand

- könnte formal als Graph formalisiert werden

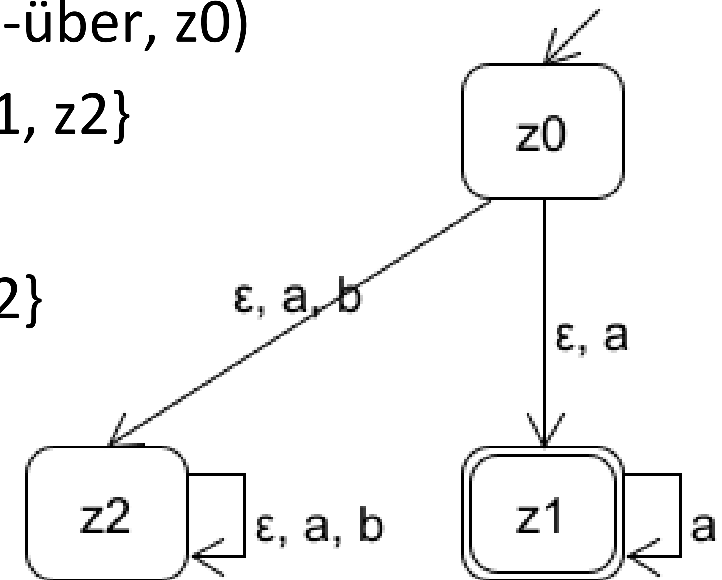


abstraktes Einstiegsbeispiel

- Automat A = ({a,b}, {z0,z1,z2}, {z1}, über, ε-über, z0)

über(z0, a) = {z1, z2}
über(z0, b) = {z2}
über(z1, a) = {z1}
über(z1, b) = {}
über(z2, a) = {z2}
über(z2, b) = {z2}

ε-über(z0) = {z1, z2}
ε-über(z1) = {}
ε-über(z2) = {z2}

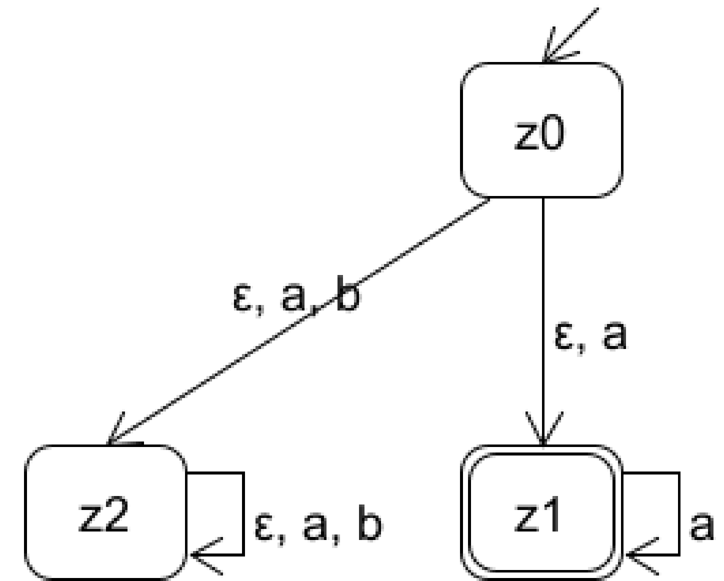


- graphische Darstellung zeigt markierten Start- und Endzustände sowie Überführungsfunktionen als *Zustandsdiagramm*
- Automaten akzeptieren Wörter. Ein Wort wird akzeptiert, wenn es einen Weg vom Start-zum Endzustand gibt, der mit diesem Wort (genauer den Zeichen des Wortes) markiert ist
- Beispielautomat akzeptiert u. a.: ε , a, aa, ..

genauere Analyse von Akzeptanz

Wort a kann in unendlich vielen Varianten geschrieben werden:

- a
- ε a
- a ε
- ε ε a ε ε
- ...



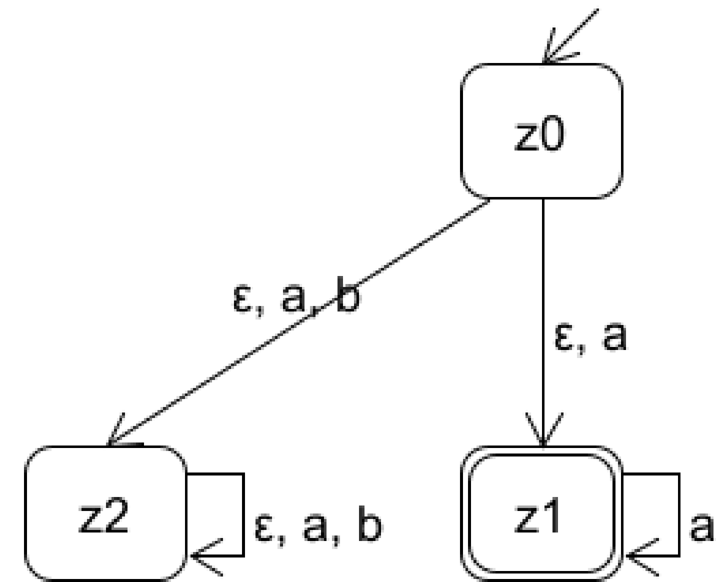
man betrachtet alle so erreichbaren Zustände

- a {z1, z2}
- ε a {z1, z2}
- a ε {z1, z2}
- ε ε a ε ε {z1, z2}

Wort wird akzeptiert, da in mindestens einer der erreichbaren Mengen ein Endzustand vorkommt

Nichtdeterminismus und Unvollständigkeit

- Nichtdeterminismus: es gibt mehrere Alternativen, was der Automat machen kann
- bereits zum Start kann der Automat in z0 oder (für eine beobachtende Person unsichtbar) bereits in z2 oder z1 sein
- beim Simulieren (Experimentieren) wählen wir einen Weg aus
- bei der Akzeptanz betrachten wir immer alle möglichen Wege



- beide Überführungsfunktionen können unvollständig sein; in z1 kein Übergang mit b und ε mit definiert

Erweiterung der Überführungsfunktionen

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

Definition: Der *ε-Abschluss*: Zustände -> Pot(Zustände)
der Funktion ε-über ist wie folgt definiert

ε-Abschluss(z) = { ze | es gibt n ∈ NatürlicheZahlen und z1,…,zn ∈ Zustände mit zi ∈ ε-über(zi-1) für 2 ≤ i ≤ n und z = z1 und ze = zn}

- Anschaulich: Welche Zustände sind in n ε-Schritten von z aus erreichbar; für n = 1 ist ze = z, damit z selbst enthalten

Definition: Die *erweiterte Überführungsfunktion*

über*: Zustände × Alphabet* -> Pot(Zustände)

wird induktiv wie folgt definiert

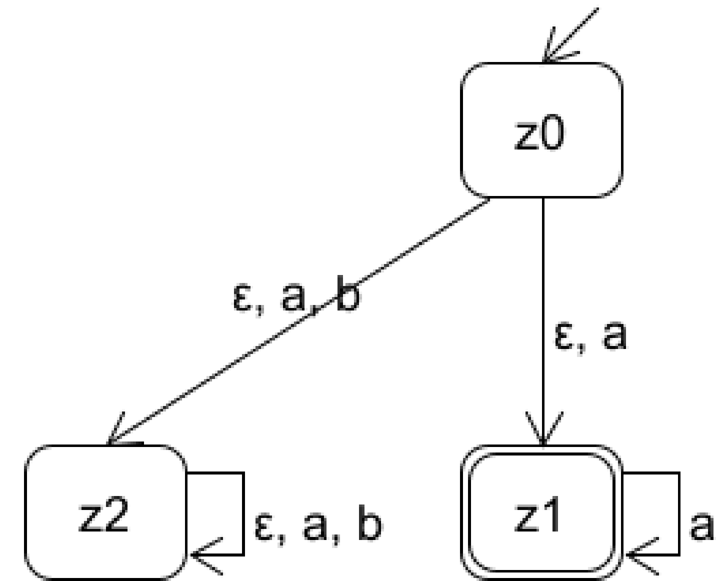
- über*(z, ε) = ε-Abschluss(z)
- über*(z, w1…wn) = {ze | es gibt zi ∈ über*(w1..wn-1)
  und zj ∈ über(zi, wn) und ze ∈ ε-Abschluss(zj)}

Beispiel

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- ε-Abschluss(z0) = {z0, z1, z2}
- ε-Abschluss(z1) = {z1}
- ε-Abschluss(z2) = {z2}

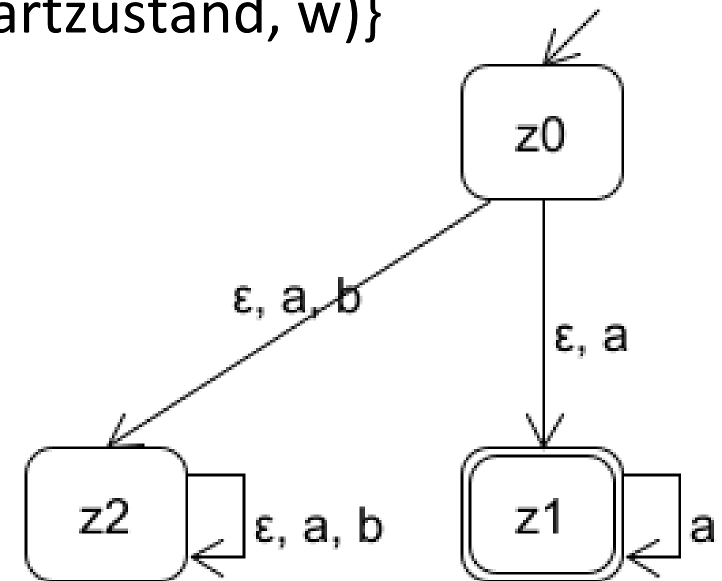
- über*(z0, ε) = {z0, z1, z2}
- über*(z0, a) = „betrachte alle Zustände aus über*(z0, ε), berechne die daraus mit a erreichbaren Zustände und davon jeweils den ε-Abschluss“ = {z1, z2}
- über*(z0, aa) = {z1, z2}
- über*(z0, ab) = {z2}



Akzeptanz

Definition :Die von einem Automaten A *akzeptierte Sprache* ist definiert als

Sprache(A) = {w ∈ Alphabet* | es gibt ze ∈ Endzustände und
ze ∈ über*(Startzustand, w)}



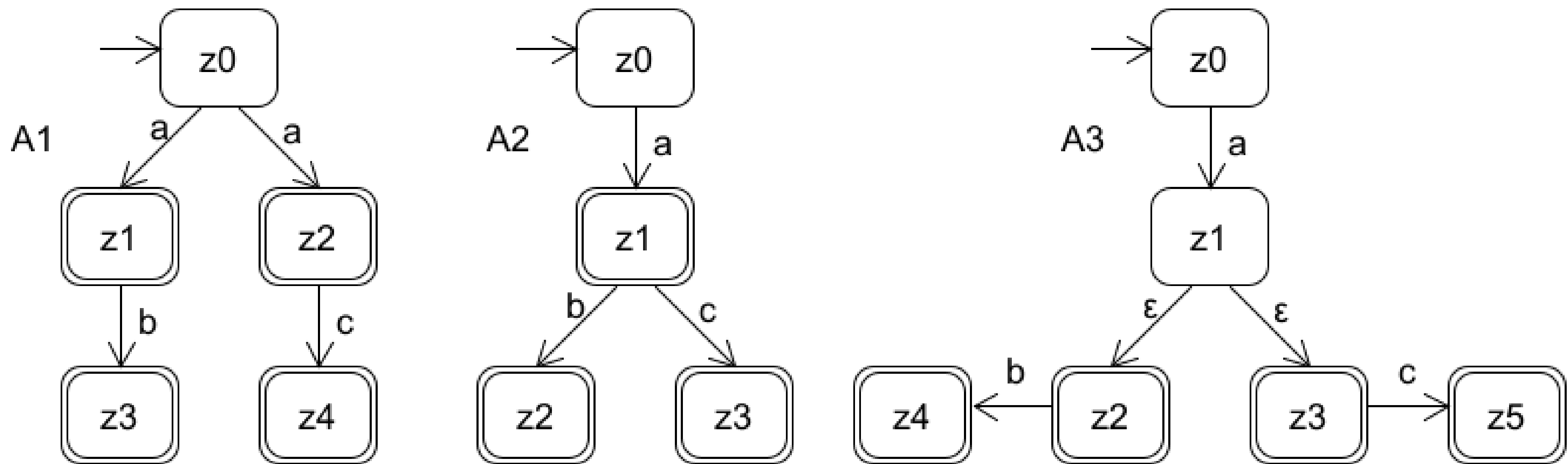
- Lang(A) = { a^n | n ∈ NatürlicheZahlen}
- ist zu beweisen, zunächst muss Lang(A) akzeptiert werden
- n = 0 , wähle ε-Übergang von z0 nach z1
- n > 0, wie n = 0, bleibe dann in z1
- weiterhin darf kein weiteres Wort akzeptiert werden; nur mit „Handwaving" ohne weiteren Formalismus erklärbar; nach einem b ist z1 nicht mehr erreichbar

Äquivalenz

Video

Definition: Zwei Automaten A1 und A2 heißen *(sprach-) äquivalent*, wenn Sprache(A1) = Sprache(A2) gilt.

- Anmerkung: es sind durchaus andere Äquivalenzbegriffe sinnvoll definierbar: folgende Automaten akzeptieren alle {a, ab, ac}



- in Variante unterscheidbar, welche nächste Zeichen nach einem Wort möglich sind, in A1 nur (a, {b}) und (a, {c}) in A2 nur (a, {b,c}) und in A3 (a, {b}), (a, {c}), (a, {b,c}) (sogenannte Readiness-Semantik)

Nichtdeterministischer Automat ~~mit ε-Übergängen~~

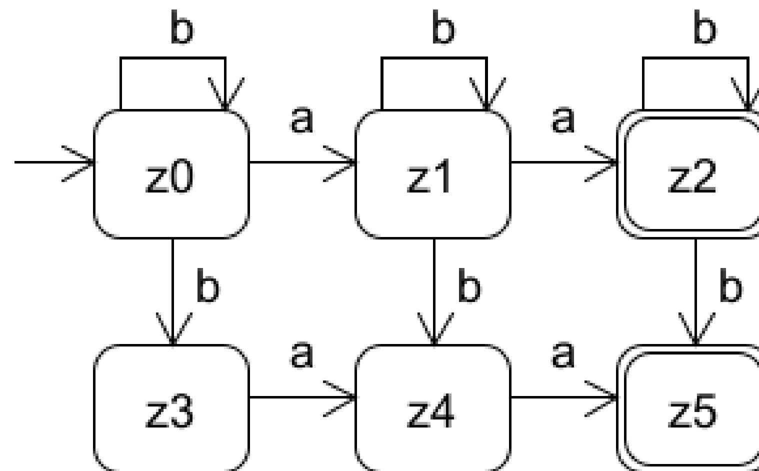
Definition: Ein *nichtdeterministischer Automat* ~~mit ε-Übergängen~~ ist ein ~~6~~ 5 -Tupel (Alphabet, Zustände, Endzustände, Überführungsfunktion, ~~ε-Überführungsfunktion~~, Startzustand) mit

- Alphabet, wie bekannt, eine endlich nicht leere Menge von Zeichen
- Zustände, eine endliche nicht leere Menge von Zuständen
- Endzustände, Endzustände $\subseteq$ Zustände
- Überführungsfunktion: Zustände $\times$ Alphabet -> Pot(Zustände)
- ~~ε-Überführungsfunktion: Zustände -> Pot(Zustände)~~
- Startzustand $\in$ Zustände

- Frage: Verlieren wir was, wenn es keine ε-Übergänge gibt
- Antwort: da wir (nur) an der (Sprach-)Äquivalenz interessiert sind, nein

Akzeptierte Sprache

- alle Definitionen sind 1:1 übernehmbar, können vereinfacht werden
- ε-Abschluss(z) fällt einfach weg, bzw. wird durch z ersetzt
- über*(z, ε) = ~~ε-Abschluss(z)~~ {z}
- über*(z, w1…wn) = {ze | es gibt zi $\in$ über*(w1..wn-1)
  und ze ~~j~~ $\in$ über(zi, wn) ~~und ze $\in$ ε-Abschluss(zj)~~}
- Sprache(A) = {w $\in$ Alphabet* | es gibt ze $\in$ Endzustände und
  ze $\in$ über*(Startzustand, w)} [genau wie vorher]
- Beispiel: Sprache(A) = {w | w enthält 2 a} = $\{b^k a b^l a b^m \mid k, l, m \geq 0\}$



Entfernung von ε-Übergängen (1/2)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

Satz: Zu jedem nichtdeterministischen Automaten mit ε-Übergängen A1 gibt es einen nichtdeterministischen Automaten (ohne ε-Übergänge) A2 mit Sprache(A1) = Sprache(A2) und umgekehrt

Beweisidee: „und umgekehrt" ist trivial, da jeder nichtdeterministische Automat auch ein nichtdeterministischer Automat mit ε-Übergängen ist, die ε-Überführung liefert immer die leere Menge

Konstruktionsschritte:

1. Berechne aus Zustand mit ε-Übergängen erreichbare Zustände

   ε-erreichbar(z) = über*(z, ε) [über Fixpunktberechnung]

für alle Zustände z: falls ε-erreichbar(z) $\cap$ Endzustände $\neq$ {}, dann ist z Endzustand im neuen Automaten

Entfernung von ε-Übergängen (2/2)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

2. Berechne mögliche Folgezustände für einen Zustand z und ein Zeichen des Alphabets a: (erst ε-Schritte, dann a, dann ε-Schritte)

```
möglicheFolgezustände(z,a) = {}
forall(vorher ∈ ε-erreichbar(z)) {
  möglicheFolgezustände(z,a) = möglicheFolgezustände(z,a)
       ∪ {nachEps | tmp ∈ über(vorher, a)
                       und nachEps ∈ ε-erreichbar(tmp)}
}
```

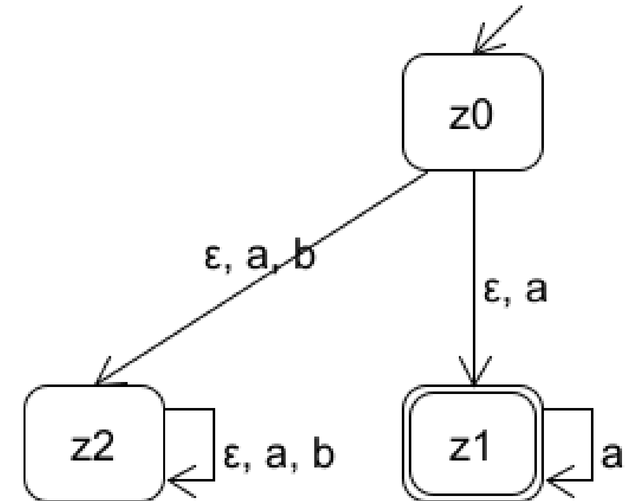
3. Berechne neue Überführungsfunktion überOhneEps (Rest vom Automaten wird übernommen)

```
forall(z ∈ Zustände){
  forall(a ∈ Alphabet) {
    überohneEps(z,a) = möglicheFolgeZustände(z, a)
  }
}
```

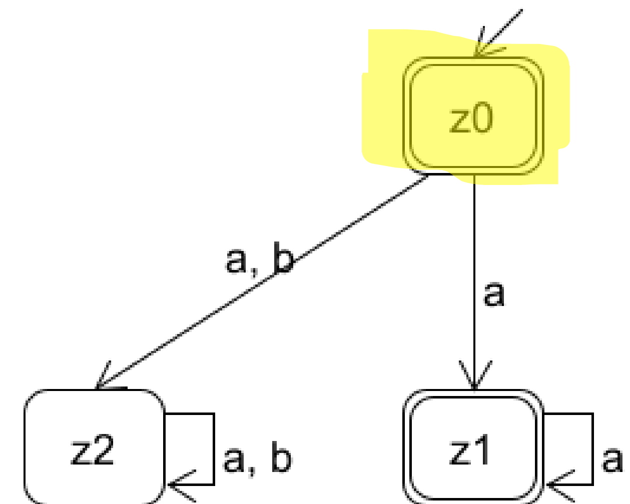
Beispiel 1 – Entfernung von ε-Übergängen

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- ε-erreichbar(z0) = {z0, z1, z2}
- ε-erreichbar(z1) = {z1}
- ε-erreichbar(z2) = {z2}



- möglicheFolgezustände(z0,a) = {z1, z2}
- möglicheFolgezustände(z0,b) = {z2}
- möglicheFolgezustände(z1,a) = {z1}
- möglicheFolgezustände(z1,b) = {}
- möglicheFolgezustände(z2,a) = {z2}
- möglicheFolgezustände(z2,b) = {z2}



Beispiel 2 – Entfernung von ε-Übergängen

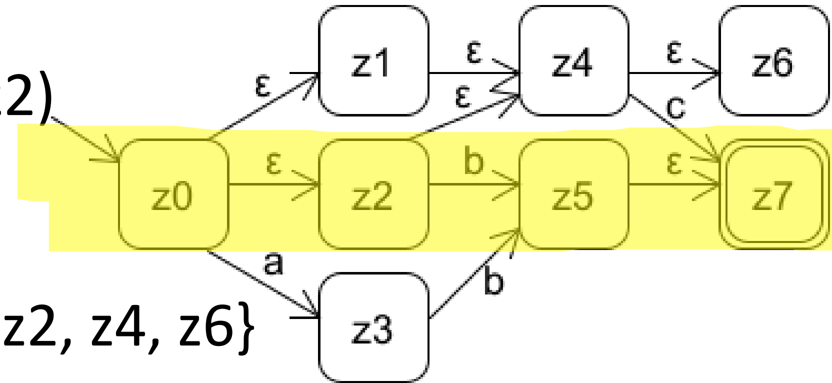
ε-erreichbar(z0) = {z0, z1, z2, z4, z6}

ε-erreichbar(z1) = {z1, z4, z6} = ε-erreichbar(z2)

ε-erreichbar(zi) = {zi}, für i = 3, 6, 7

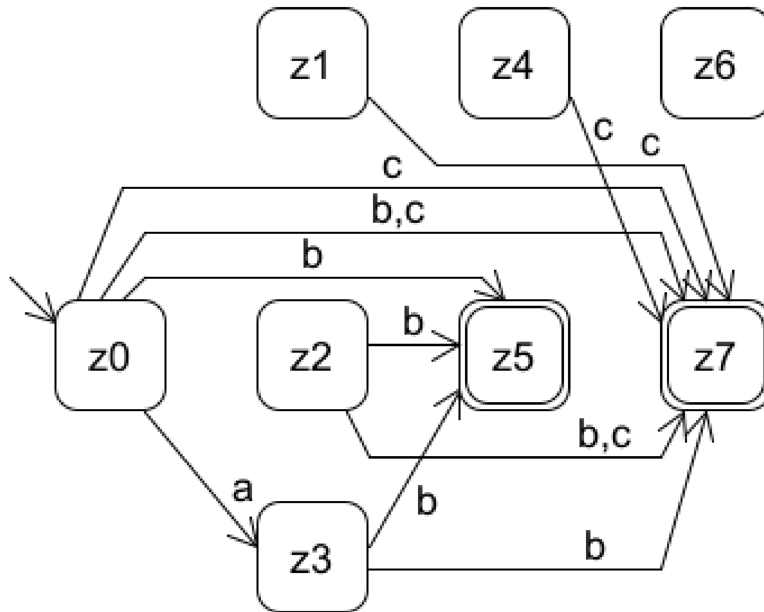
ε-erreichbar(z4) = {z4, z6}

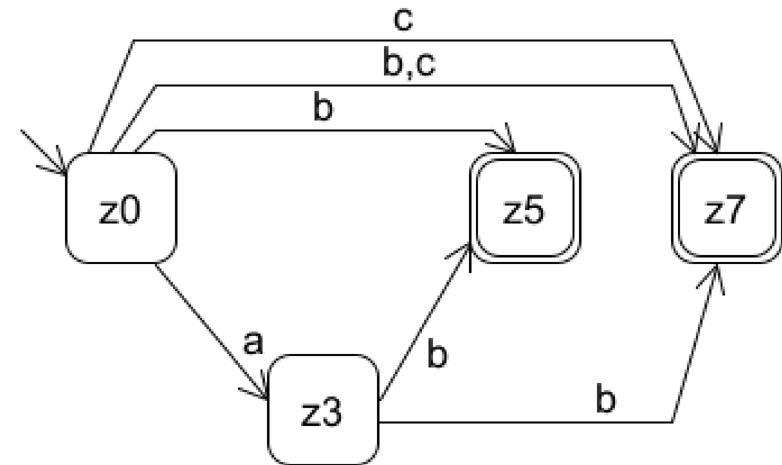
ε-erreichbar(z5) = {z5, z7} ε-erreichbar(z2) = {z2, z4, z6}



überOhneEps (Rest leere Menge):

z0	a	z3
z0	b	z5, z7
z0	c	z7
z1	c	z7
z2	b	z5, z7
z2	c	z7
z3	b	z5, z7
z4	c	z7



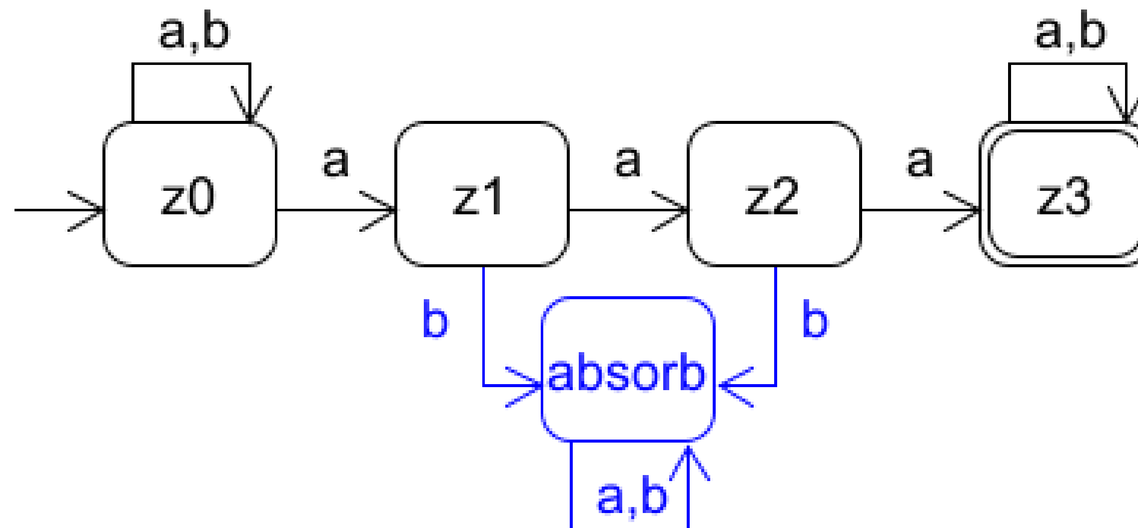


Vervollständigung von Automaten

- für nachfolgende Konstruktionen wird benötigt, dass es für jeden Zustand und jedes Zeichen einen Folgezustand gibt

  $\forall$ze $\in$Alphabet, zu $\in$ Zustände: über(zu, ze) $\neq${}

- Ansatz: ergänze neuen Zustand und ergänze für fehlende Kanten einen Übergang zu diesem neuen Zustand; für jedes Zeichen bleibt der neue Zustand in sich selbst (absorbierender Zustand)
- Satz: Der vervollständigte Automat ist äquivalent zum ursprünglichen Automaten



~~Nicht~~deterministischer Automat ~~mit ε-Übergängen~~

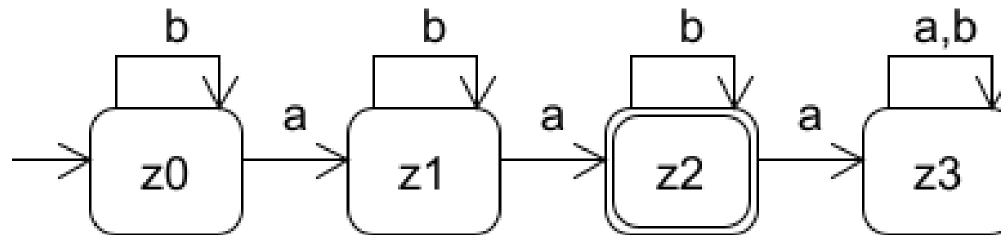
Definition: Ein ~~nicht~~deterministischer Automat ~~mit ε-Übergängen~~ ist ein ~~6~~ 5 -Tupel (Alphabet, Zustände, Endzustände, Überführungsfunktion, ~~ε-Überführungsfunktion~~, Startzustand) mit

- Alphabet, wie bekannt, eine endlich nicht leere Menge von Zeichen
- Zustände, eine endliche nicht leere Menge von Zuständen
- Endzustände, Endzustände $\subseteq$ Zustände
- Überführungsfunktion: Zustände $\times$ Alphabet -> ~~Pot(Zustände)~~ Zustände
  Achtung: die Funktion ist total, also für alle Paare definiert
- ~~ε-Überführungsfunktion: Zustände -> Pot(Zustände)~~
- Startzustand $\in$ Zustände

- Frage: Verlieren wir was, wenn es keinen Nichtdeterminismus gibt
- Antwort: da wir (nur) an der (Sprach-)Äquivalenz interessiert sind, nein

Akzeptierte Sprache

- alle Definitionen wären 1:1 übernehmbar, können aber vereinfacht werden, da Überführungsfunktion immer genau einen Zustand liefert
- über*: Zustände × Alphabet* -> ~~Pot(Zustände)~~ Zustände
- ε-Abschluss(z) fällt einfach weg, bzw. wird durch z ersetzt
- über*(z, ε) = ~~ε-Abschluss(z)~~ ~~{z}~~ z
- über*(z, w1…wn) = ze   mit es gibt ~~zi ∈~~ über*(w1..wn-1) = zi
  und ~~ze ∈~~ über(zi, wn) = ze ~~und ze ∈ ε-Abschluss(zj)~~
- Sprache(A) = {w ∈ Alphabet* | ~~es gibt ze ∈ Endzustände und ze ∈~~ über*(Startzustand, w) ∈ Endzustände }
- Beispiel: Sprache(A) = {w | w enthält 2 a} = $\{b^k a b^l a b^m \mid k, l, m \geq 0\}$



Entfernung von Nichtdeterminismus (1/4)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

Satz: Zu jedem nichtdeterministischen Automaten A1 gibt es einen deterministischen Automaten A2 mit Sprache(A1) = Sprache(A2) und umgekehrt

Beweisidee: „und umgekehrt“ ist trivial, da jeder deterministische Automat auch ein nichtdeterministischer Automat ist; formal muss die Überführungsfunktion auf Mengen umgeschrieben werden

Konstruktionsschritte anschaulich:

Betrachte für einen Zustand und ein Zeichen welche Zustände erreicht werden können, nehme dies als EINEN neuen Zustand, für den dann für jedes Zeichen einzeln berechnet wird, welche Menge von Zuständen dann erreichbar ist …

Formaler: Zustandsmenge von A2 ist Potenzmenge der Zustände von A1

Entfernung von Nichtdeterminismus (2/4) – etwas formaler

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

Berechne schrittweise Folgezustände für

A2 = (Alphabet_A1,

Zustände_A2 $\subseteq$ Pot(Zustände_A1),

{ze | ze $\in$ Zustände_A2 $\cap$ Endzustände_A1 } // jeder Zustand, der mindestens einen Endzustand enthält ist Endzustand,

über_A2 ,

{Startzustand_A1} // Zustand als einelementige Menge

)

Starte Berechnung von über_A2 mit {Startzustand} und berechne Folgemenge für jedes Zeichen, wird dabei ein neuer Zustand entdeckt, führe die Berechnung für diesen Zustand und jedes Zeichen fort; so wird Zustände_A2 berechnet

Entfernung von Nichtdeterminismus (3/4)

Starte mit {z0}

mit a erreichbar: {z1, z2} // neuer Zustand

mit b erreichbar: {z2} // neuer Zustand

betrachte {z2}

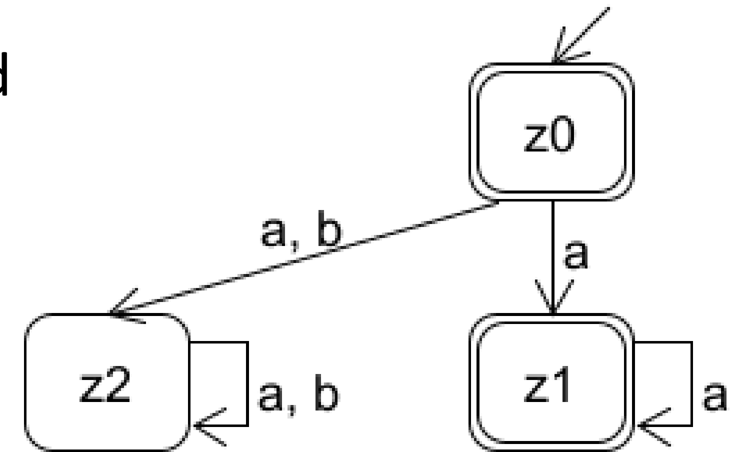
mit a erreichbar: {z2}

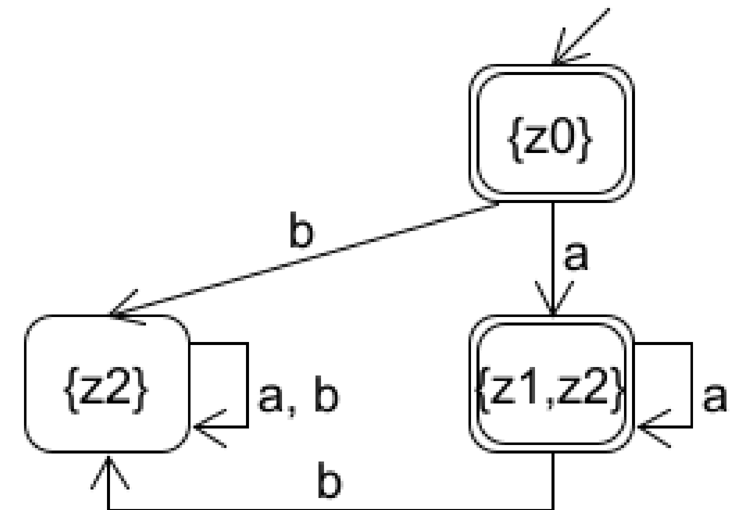
mit b erreichbar: {z2}

betrachte {z1, z2}

mit a erreichbar: {z1} ∪ {z2}

mit b erreichbar: { } ∪ {z2}





Entfernung von Nichtdeterminismus (4/4)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

Algorithmus für Zustandsberechnung und neue Überführungsfunktion:

```
start = {Startzustand_A1} // Zustandsmenge
anzahlAlt = 0
zustände = {start} // Menge von Zustandsmengen
while(anzahlAlt <> zustände.size()) { // Fixpunktberechnung
  anzahlAlt = zustände.size()
  for (Zustandsmenge zm: zustände) {
    for(Zeichen ze: Alphabet_A1) {
       folgezustand = {}
       for(Zustand zst: zm) {
         folgezustand = folgezustand ∪ ueber_A1(zst, ze)
       }
       zustände.add(folgezustand)
       ueber_A2.add(zm, ze, folgezustand)
    }
  }
}
```

Beispiel 2: Entfernung des Nichtdeterminismus

{z0} –a-> {z1,z2} (neu)

{z0} –b-> {z2} (neu)

{z1,z2} –a-> {z0,z1,z2} (neu)

{z1,z2} –b-> {z0,z1} (neu)

{z0,z1,z2} –a-> {z0,z1,z2}

{z0,z1,z2} –b-> {z0,z1,z2}

{z0,z1} –a-> {z1,z2}

{z0,z1} –b-> {z0,z2} (neu)

{z0,z2} –a-> {z0,z1,z2}

{z0,z2} –b-> {z1,z2}

{z2} –a-> {z0,z1,z2}

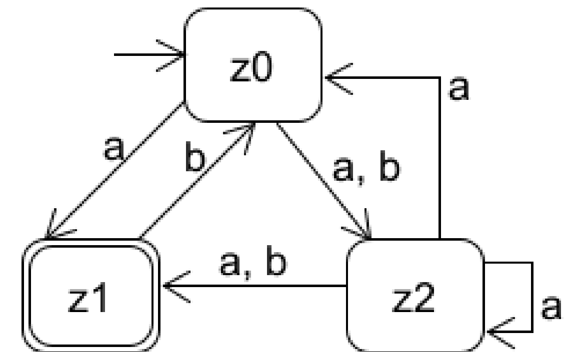
{z2} –b-> {z1} (neu)

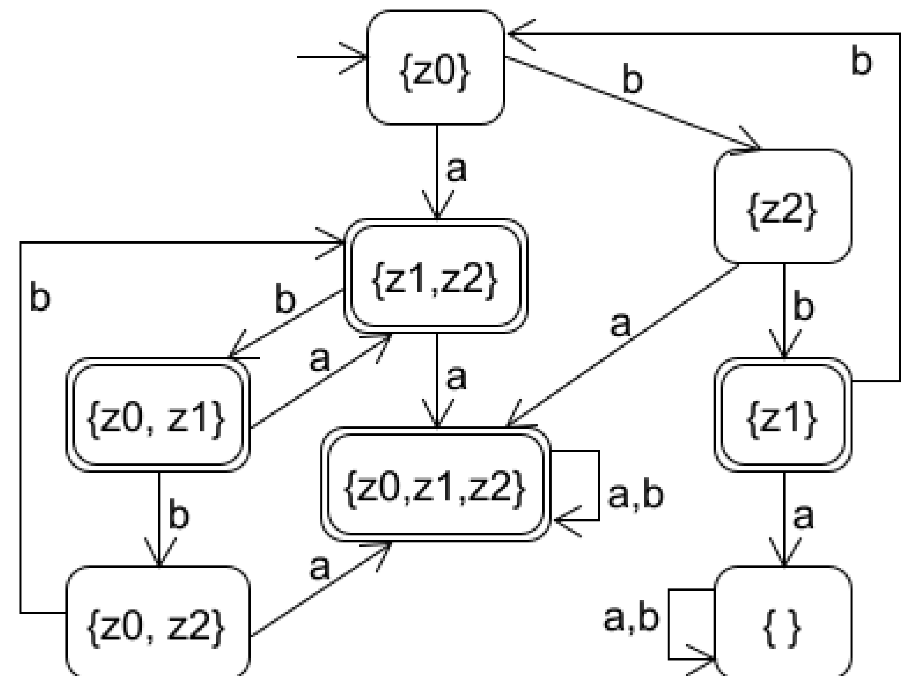
{z1} –a-> { } (neu)

{z1} –b-> {z0}

{ } –a-> { }

{ } –b-> { }

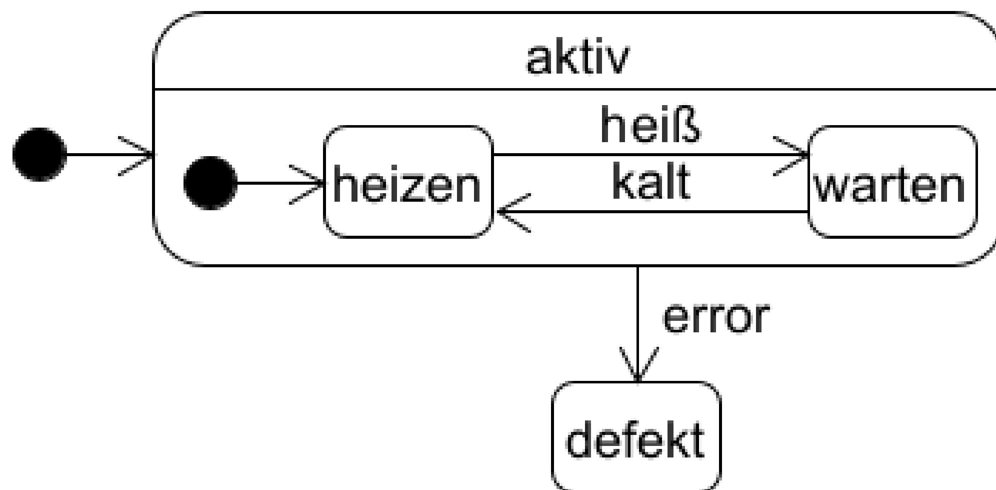




Exponentielle Zustandsanzahl

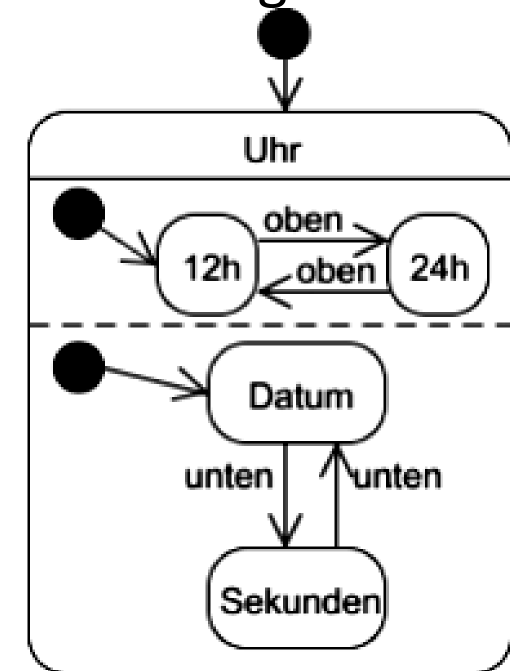
HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- vorheriges Beispiel zeigt, dass die exponentielle Anzahl an Zuständen erreicht werden kann
- Ansatz auf n Zustände übertragbar: mit b wird immer zum nächsten Zustand geschaltet, es gibt immer einen Zustand aus dem mit einem Zeichen zu i aus n Zuständen verzweigt wird ($0 \leq i \leq n$)
- Idee führte zu syntaktischen (und semantischen) Erweiterungen durch David Harel mit Statecharts



<- hierarchisch

parallel ->



s. z. B.: [Kle18] S. Kleuker, Grundkurs Software-Engineering mit UML, 4. aktualisierte Auflage, Springer Vieweg, Wiesbaden, 2018

Nichtdeterminismus - Anwendung

- theoretisch: für jeden möglichen nicht-deterministischen Schritt wird ein weiterer Überprüfungsprozess angestoßen, so kann in n-Schritten für ein Wort der Länge n überprüft werden, ob Wort akzeptiert wird
- praktisch: theoretischer Ansatz ist leicht im Programm umsetzbar, wenn es n > 1 Alternativen gibt, starte n-1 neue Prozesse, die auf dem gleichen Automaten die Überprüfung durchführen

```
global boolean ergebnis = false;

void akzeptiert(Zustand z, Wort w) {
  if (w == ε) {
    if (z ∈ Endzustände) {
      ergebnis = true;
    }
    return;
  }
  for( Zustand nächster: über(z, w.get(0))) {
    go akzeptiert(nächster, w.abZeichen(1)) // neuer Prozess
  }
}
```

Beispiel: Parallelisierung

(vereinfacht: nicht n sondern n-1 neue Prozesse)

Schritt 0: Prozess 0: check(z0, abaa)

Schritt 1: Prozess 0: check(z1, baa)

Schritt 2: Prozess 0: check(z2, aa)

Prozess 1: check(z3, aa)

Schritt 3: Prozess 0: check(z4, a)

Prozess 2: check(z5, a)

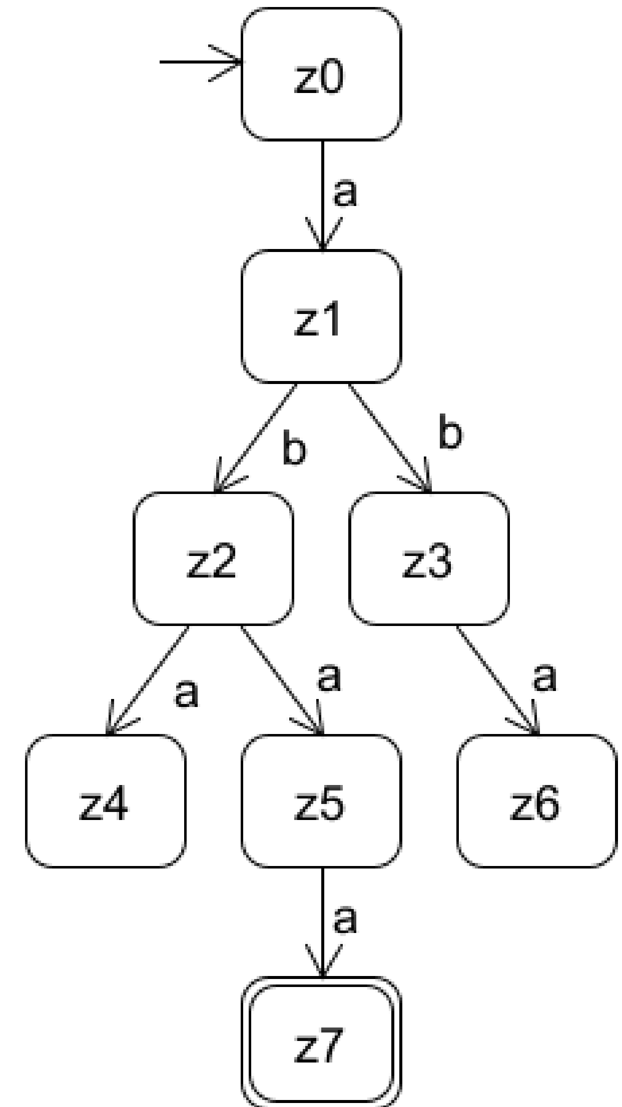
Prozess 1: check(z6, a)

Schritt 4: Prozess 0 terminiert ohne Effekt

Prozess 2: check(z7, ε)

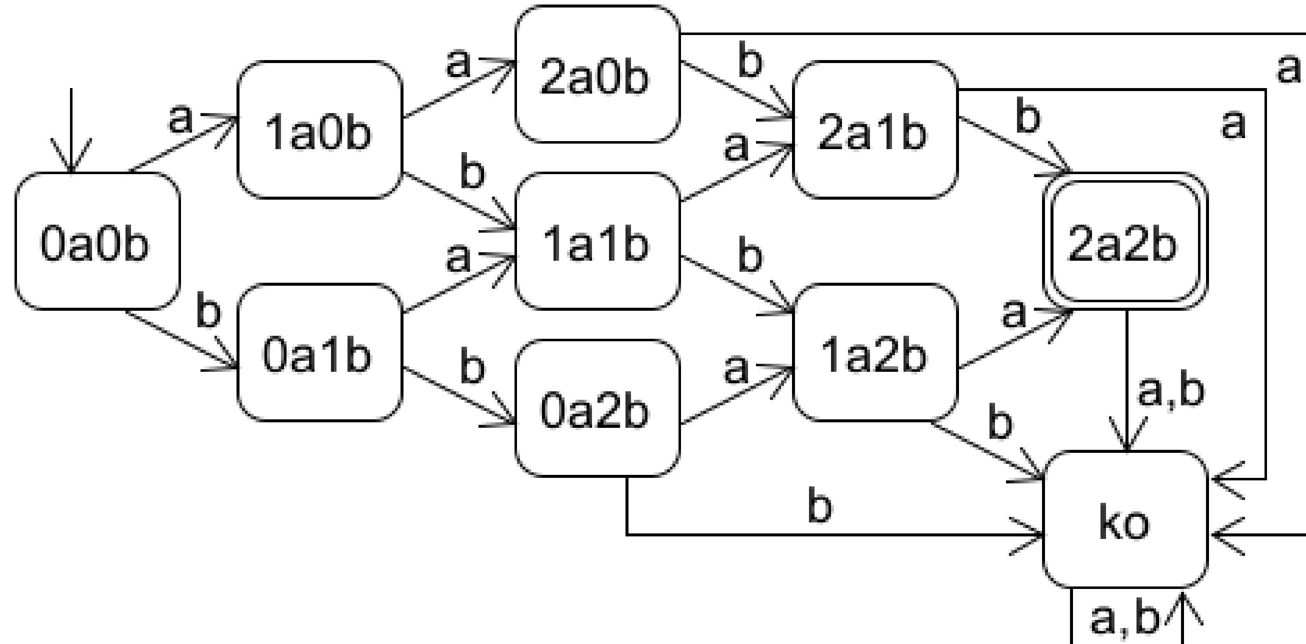
Prozess 1 terminiert ohne Effekt

Schritt 5: Prozess 2: ergebnis = true

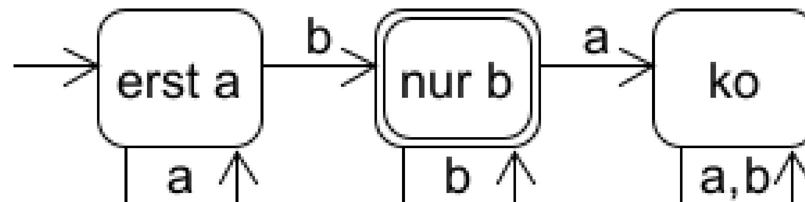


Modellierungsmöglichkeiten mit endlichen Automaten

- in Zuständen bis zu einem Wert n zählen, z. B. Wort aus 2 a und 2 b



- Reihenfolgen garantieren: erst nur a, dann nur mindestens ein b



was ist nicht modellierbar – Pumping-Lemma 1 (1/2)

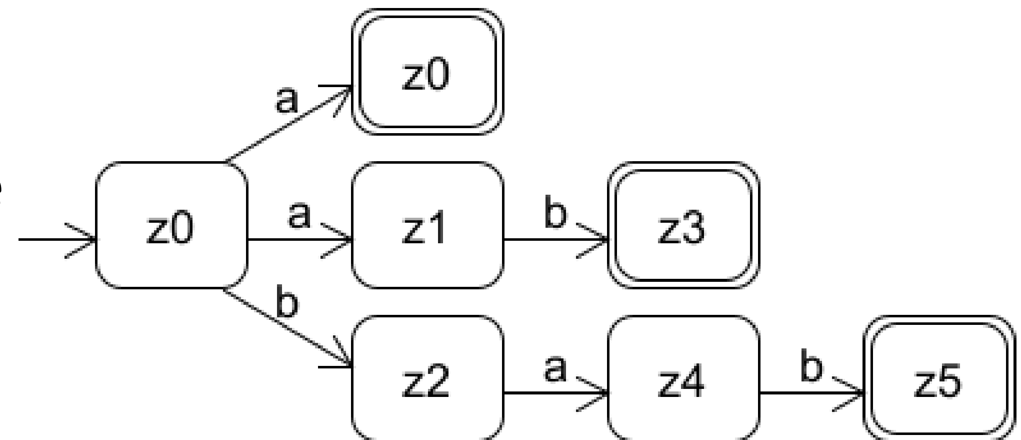
HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- erst a, dann b und es gibt weniger a als b (benötigte unendlich viele Zustände); formaler:

Satz (Pumping-Lemma für endliche Automaten [reguläre Sprachen]): Wenn eine Sprache von einem endlichen Automaten akzeptierbar ist, dann gibt es ein n, so dass für alle Worte w der Sprache, die **länger als n** sind, es eine Aufteilung von w gibt mit w = xyz, so dass auch xy^jz zur Sprache gehört, für $0 \leq j$ und x, y, z $\in$ Alphabet *, $y \neq \varepsilon$ und die Länge von xy ist $|xy| \leq n$.

erste Überlegung: endliche Sprachen sind akzeptierbar, z. B.

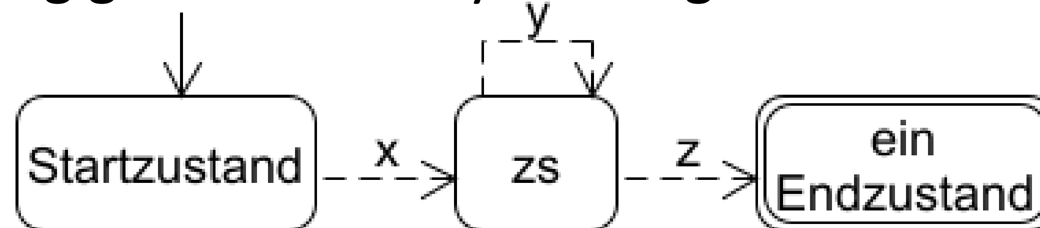
L = {a, ab, bab}, wähle n = 4 (Länge des längsten Wortes) und dann muss kein Wort aufgepumpt oder geschrumpft (j = 0) werden



was ist nicht modellierbar – Pumping-Lemma 1 (2/2)

Überlegung für unendliche Sprachen zu denen es einen Automaten gibt, zeige dass es für lange Worte w die Zerlegung in xyz geben muss

- der Automat hat k Zustände
- die Sprache hat Worte die länger als k sind (sonst nicht unendlich viele Worte in der Sprache), z. B. ein Wort w
- wenn der Automat das Wort w abarbeitet, wird pro Zeichen ein Zustand besucht, da laenge(w) > k, muss es einen Zustand zs geben, der sich in der Abarbeitung wiederholt
- genauer gibt es dann ein Teilwort y von w mit ueber*(zs, y) = zs
- dann gibt es einen Anfang von w mit über*(Startzustand, x) = zs
- dann gibt es ein Ende von w mit über*(zs, z) ∈ Endzustände
- damit Zerlegung gefunden und y beliebig oft wiederholbar (auch 0-mal)



Beispiel: Klausuraufgabenvariante (1/2)

Video

Erinnerung: Pumping Lemma für endliche Automaten

L von endlichem Automaten akzeptierbar $\Rightarrow$

$\exists$ n$\in$NatuerlicheZahlen $\forall$ w$\in$\{v | v$\in$L $\wedge$ länge(v)>n\} $\exists$ x,y,z (w=xyz
und y$\neq\varepsilon$ und länge(xy) $\leq$ n und $\forall$ i$\in$NatuerlicheZahlen $xy^iz\in$L)

Aufgabe: Begründen oder widerlegen Sie, dass $\{a^nb^m \mid n < m\}$ mit einem endlichen Automaten akzeptierbar ist.

Begründung wäre eine Angabe eines endlichen Automaten (geht nicht)

Widerlegen: Zeige, dass die Sprache das Pumping-Lemma nicht erfüllt.

$\forall$ n$\in$NatuerlicheZahlen $\exists$ w$\in$\{v | v$\in$L $\wedge$ länge(v)>n\} $\forall$ x,y,z (w$\neq$xyz
oder y=ε oder länge(xy) > n oder $\exists$ i$\in$NatuerlicheZahlen $xy^iz\notin$L)

es also für lange Worte die aufpumpbare Zerlegung w = xy^jz nicht gibt

Ansatz: Suche kritisches Wort (genauer Wortstruktur, da $\forall$n) , bestimme alle Zerlegungsmöglichkeiten und zeige, dass keine davon aufpumbar ist

Klausuraufgabenvariante (2/2) - Sprachmächtigkeit

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

Aufgabe: Begründen oder widerlegen Sie, dass $\{a^n b^m \mid n < m\}$ mit einem endlichen Automaten akzeptierbar ist.

- betrachte Wort(struktur) $a^k b^{k+1}$, gibt 3 Möglichkeiten zum Aufpumpen
- y besteht aus a, für i=2 Anzahl a größer-gleich b, Widerspruch
- y besteht aus ab, für i=2 wird erst a, dann b verletzt, Widerspruch
- y besteht aus b, für i=0 wird Anzahl b kleiner-gleich a, Widerspruch

- ähnlich wie bei der Entscheidbarkeit, kann man sich Fragen wie mächtig ist ein Konzept, wie das der endlichen Automaten
- $\{a^n b^m \mid n < m\}$ ist nicht mit endlichen Automaten akzeptierbar, aber trivial mit eine Turing-Maschine akzeptierbar und mit einer kontextfreien Grammatik erzeugbar
- Start -> a Start b | B B -> b | bB

Kann mit endlichen Automaten programmiert werden?

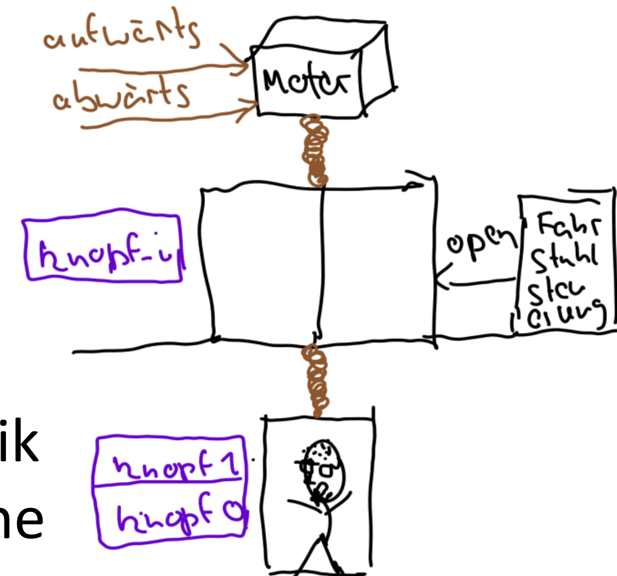
- Ja, die verwendeten Typen müssen nur endlich sein (sind eigentlich alle); genauer einen kleinen Wertebereich haben
- Zustände ergeben sich aus dem kartesischen Produkt der Wertebereiche der Variablen
- so „normale“ imperative Programme formulierbar
- Ansatz: Automatengeneratoren: Nutze Programmiersprache und generiere daraus einen (evtl. sehr großen) endlichen Automaten (z. B. eine Steuerung)
- Vorteil: für endliche Automaten praktisch alles Interessante entscheidbar; nur problematisch wenn es zu viele erreichbare Zustände gibt (Zustandsraum-Explosion)
- Ansatz wird auch zur Verifikation genutzt (Model-Checking)

Beispielskizze: Fahrstuhlsteuerung (1/5)

Ereignisse (Alphabet)

- > knopf_i : Anforderungsknopf im i-Stock oder Fahrstuhl gedrückt (Technik garantiert, dass der Knopf erst erneut gedrückt werden kann, wenn die Steuerung ihn freigibt)
- open_i >: Tür im i-ten Stock wird geöffnet (Technik garantiert, dass Tür erst nach bestimmter Zeit ohne Personen in der Tür schließt, Fahrstuhlsteuerung solange blockiert)
- aufwärts >: Fahrstuhl ein Stockwerk hochfahren
- abwärts >: Fahrstuhl ein Stockwerk herunterfahren

- > x: für eingehendes Signal der Steuerung
- x >: für ausgehendes Signal der Steuerung an Aktuator



Beispielskizze: Fahrstuhlsteuerung (2/5) – n Stockwerke

Variablen (eine mögliche Modellierung)

```
boolean[n] knopf = false // true, wenn i-ter Knopf gedrückt
-1..1 richtung = 0  // Richtung des Fahrstuhls, 0 = steht
0..n min = n - 1 // niedrigstes anzufahrendes Stockwerk
0..n max = 0     // höchstes anzufahrendes Stockwerk
0..n aktuell = 0 // Stockwerk in den sich Fahrstuhl befindet
0..n wartend = 0 // Anzahl der gerade gedrückten Knöpfe

>knopf[i]: if richtung == 0 && wartend == 0
           then wartend = 1
                if aktuell < i
                then richtung = 1, max = i
                if aktuell > i
                then richtung = -1, min = i
           else wartend = wartend + 1
           //… (nächste Folie)
```

Beispielskizze: Fahrstuhlsteuerung (3/5) – Hauptprogramm


```
          else wartend = wartend + 1
               if i > max then max = i
               if i < min then min = i

loop  // Hauptprogramm, läuft unendlich
if wartend > 0  && knopf[aktuell] // Stockwerk erreicht
   then öffnen[aktuell]>, knopf[aktuell] = false, wartend -= 1
       if wartend == 0
       then richtung = 0, min = n - 1, max = 0
       else if aktuell == max
            then richtung = -1, max = 0 //umkehren, da jemand wartet
            if aktuell == min
            then richtung = 1, min = n - 1
   else if richtung == 1 then aufwärts>, aktuell ++
        if richtung == -1 then abwärts >, aktuell --
end loop
```

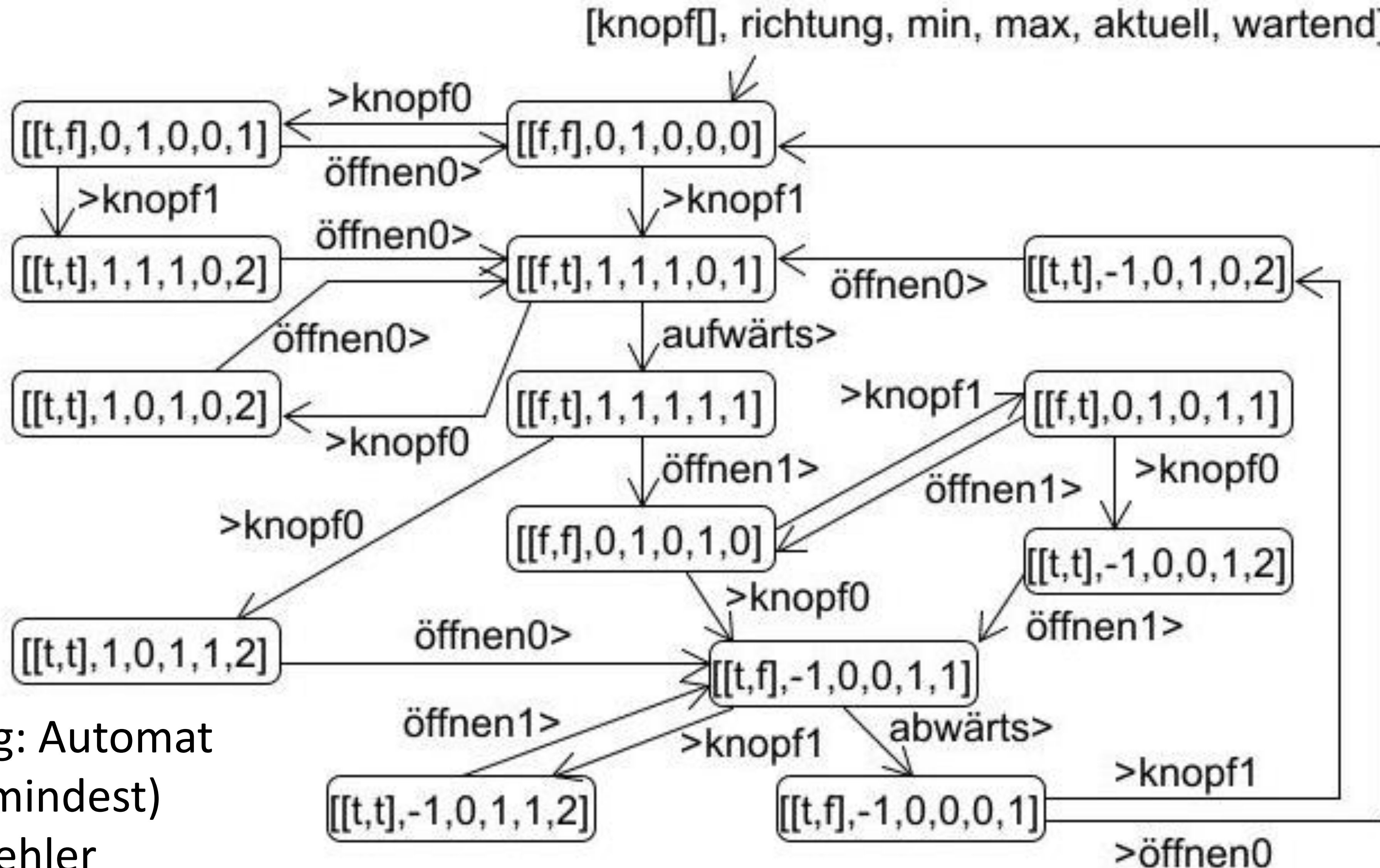
Beispielskizze: Fahrstuhlsteuerung (4/5)

- automatisch generierter Automat
- Zustand: [knopf[], richtung, min, max, aktuell, wartend]
- 2 Stockwerke: 2*2 *3 *2 *2 *2 *3 = 288 Zustände
- Berechnungsidee: [[f,f],0,1,0,0,0] - >knopf1 -> [[f,t],1,1,1,0,1]

```
knopf[i]: if richtung == 0
          then wartend = 1
               if aktuell < i
               then richtung = 1, max = i
               if aktuell > i
               else richtung = -1, min = i
```

- praktisch alle Überprüfungen entscheidbar, u. a. Wertebereiche werden eingehalten; öffnen_i nur wenn aktuell = i

Beispielskizze: Fahrstuhlsteuerung (5/5) – 2 Stockwerke



Achtung: Automat hat (zumindest) einen Fehler

Minimierung von endlichen Automaten

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

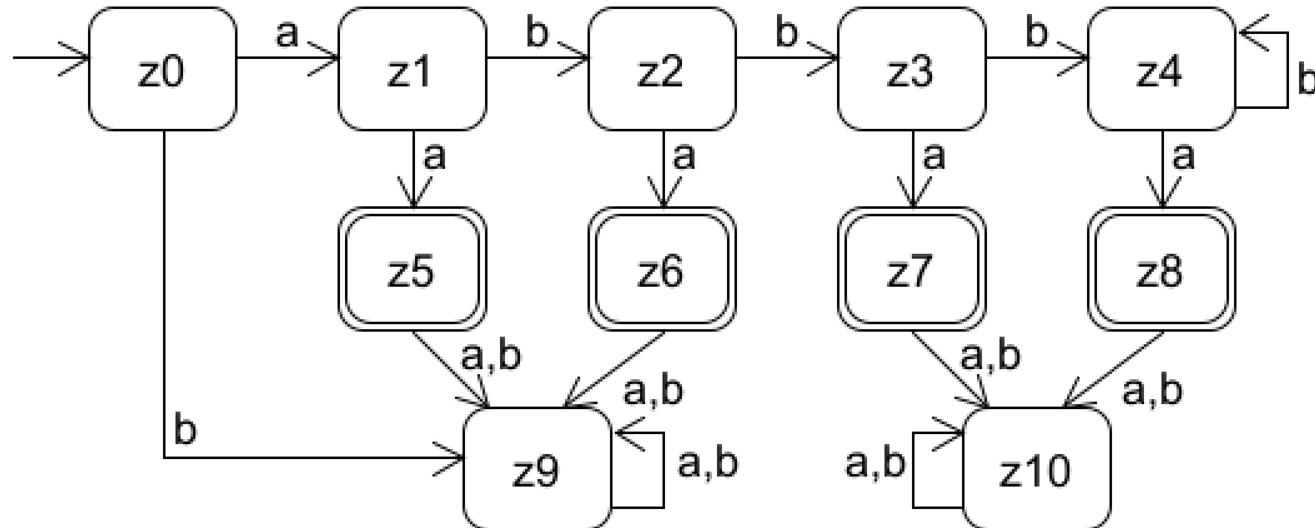
- bisher hatten Sie (hoffentlich) mehrfach das Gefühl, zu einem gegebenen Automaten fällt ihnen ein sprachäquivalenter Automat mit weniger Zuständen ein
- Ansatz: kläre zunächst, wann zwei Zustände eines Automaten äquivalent sind (sich gleich verhalten) und fasse dann alle miteinander äquivalenten Zustände zu einem Zustand zusammen

Definition: Zwei Zustände z1 und z2 eines deterministischen Automaten heißen *äquivalent*, wenn ausgehend von diesen Zuständen für jedes Wort jeweils immer ein oder kein Endzustand erreicht wird; formaler:

für alle Worte w $\in$Alphabet*:

über*(z1, w) $\in$Endzustände $\Leftrightarrow$ über*(z2, w) $\in$Endzustände

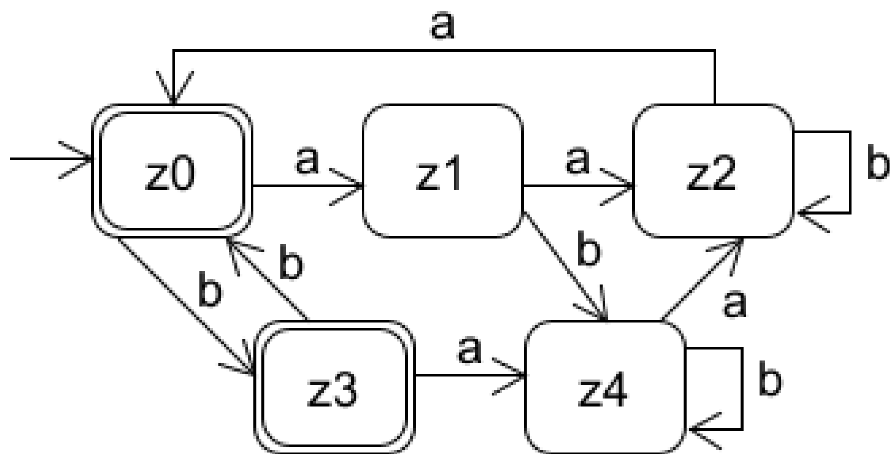
Beispiel mit äquivalenten Zuständen



- man „sieht“ leicht, dass z5 und z6 äquivalent sind, da sie sich nach einem Schritt gleich verhalten und nach null Schritten beide Endzustände sind
- man berechnet, dass z0 und z1 nicht äquivalent sind, obwohl sie nach null Schritten Nicht-Endzustände sind, aber der eine mit a einen Endzustand erreicht und der andere nicht
- allgemeine Idee: schrittweise Analyse der Äquivalenz, suche nach Widerspruch, erst nach 0, dann nach 1, dann 2 Schritten …

Schrittweise Berechnung der Äquivalenz (1/2)

- Matrix hält fest ob Zustände äquivalent sind, wenn nein ein x



0. Schritt: trenne End- und Nicht-Endzustände

	z1	z2	z3	z4
z0	X	X		X
z1			X	
z2			X	
z3				X

i-ter Schritt: wenn (zx, zy) äquivalent in Schritt i-1, prüfe ob über(zx, a) äquivalent mit über(zy, a) und über(zx, b) äquivalent mit über(zy, b)

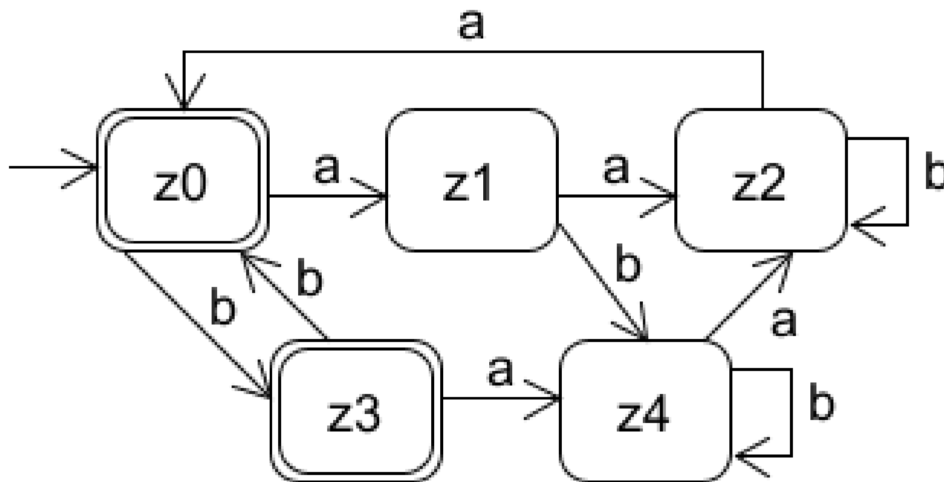
(z1, z2) –a-> (z2, z0)   (z2, z4) –a-> (z0, z2)

Ende, wenn kein neues X gesetzt

	z1	z2	z3	z4
z0	X	X		X
z1		X	X	
z2			X	X
z3				X

Schrittweise Berechnung der Äquivalenz (2/2)

- keine neuen Nicht-Äquivalenzen im nächsten Schritt



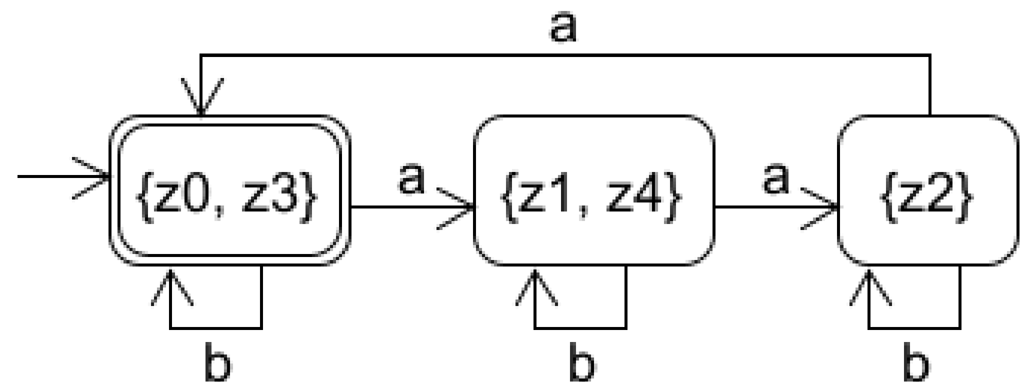
	z1	z2	z3	z4
z0	X	X		X
z1		X	X	
z2			X	X
z3				X

Ergebnis:

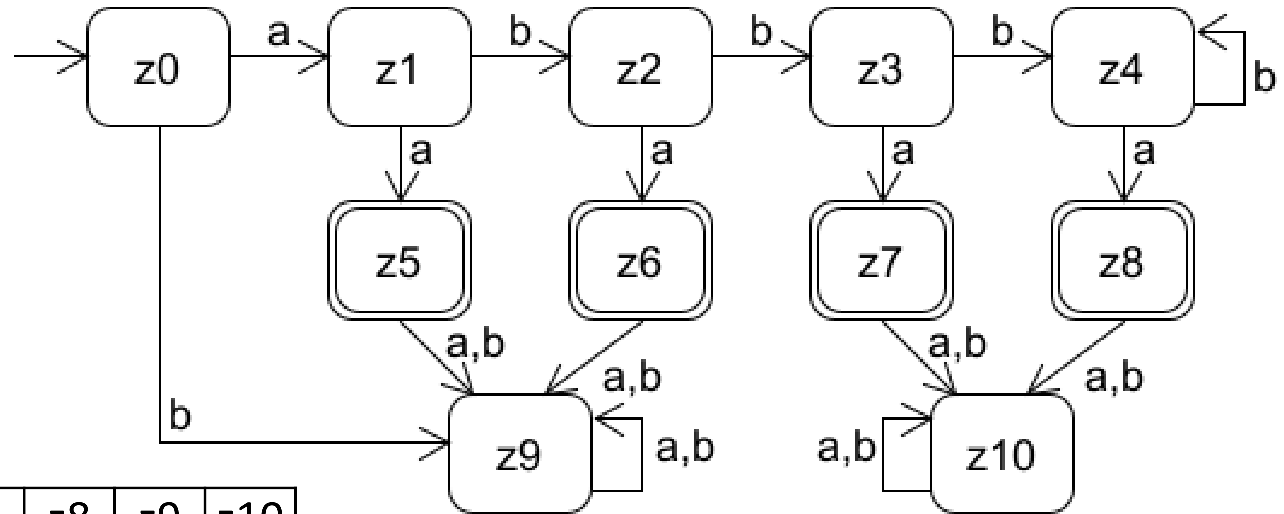
z0 äquivalent mit z3

z1 äquivalent mit z4

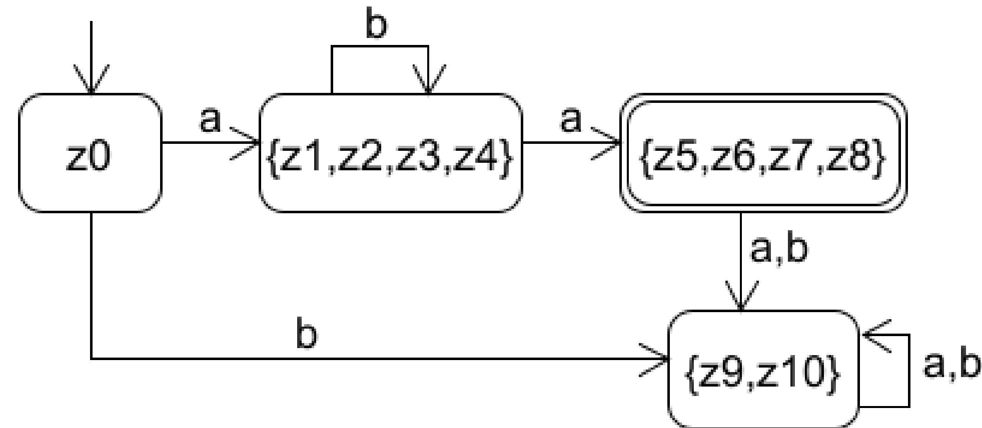
z2 ohne äquivalenten Zustand



Äquivalenz – 2. Beispiel



	z1	z2	z3	z4	z5	z6	z7	z8	z9	z10
z0	X	X	X	X	X	X	X	X	X	X
z1					X	X	X	X	X	X
z2					X	X	X	X	X	X
z3					X	X	X	X	X	X
z4					X	X	X	X	X	X
z5									X	X
z6									X	X
z7									X	X
z8									X	X
z9										



Minimierungsalgorithmus genauer (1/4)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

Satz: Zu jedem deterministischen endlichen Automaten gibt es einen bezüglich der Zustandsanzahl minimalen sprachäquivalenten Automaten (genauer: alle minimalen Automaten sind isomorph, d. h. Zustände können bei anderem Ergebnis einfach umbenannt werden)

```
// Zustände sind geordnete Menge
boolean äquivalent[Zustände][Zustände] // in Java Zuständen erst
                                        // int-Werte zuordnen
for(za: Zustände) {
  for (zb: Zustände) {  // Schritt 0
      äquivalent[za][zb]
          =    (za ∈ Endzustände && zb ∈ Endzustände)
             || (za ∉ Endzustände && zb ∉ Endzustände)
  }
}
```

Minimierungsalgorithmus genauer (2/4)

```
weiter = true
while (weiter) {  // Fixpunkt
  weiter = false;
  for(za: Zustände) {
    for (zb: Zustände) {
      for (zeich: Alphabet) {
        if (äquivalent[za][ab] &&
            !äquivalent[über(za,zeich)] [über(zb,zeich)]) {
          äquivalent[za][zb] = false
          weiter = true
        }
      }
    }
  }
}
```

Minimierungsalgorithmus genauer (3/4)

```
// Aufbau des neuen Automaten, Berechnung der neuen Zustände
bearbeiteteZustände = {}
zuständeMin = {}  // Ergebnisautomat
for (z1: Zustand) {
  if (z1 ∉ bearbeiteteZustände) {
    bearbeiteteZustände.add(z1)
    neuerZustand = {z1}
    for (z2: Zustand) {
      if (z2 ∉ bearbeiteteZustände  &&  äquivalent[z1][z2]){
        neuerZustand.add(z2)
        bearbeiteteZustände.add(z2)
      }
    }
    zuständeMin.add(neuerZustand)
  }
}
```

Minimierungsalgorithmus genauer (4/4)

```
// Aufbau des neuen Automaten, Berechnung der neuen
// Überführungsfunktion

for (z: zuständeMin) {
  for(zeich: Alphabet) {
     erreichterZustand = über(z.get(0), zeich)
     for(zcheck: zuständeMin) { // suche Äquivalenzklasse
       if(erreichterZustand ∈ zcheck){
         überMin(z, zeich) = zcheck
       }
     }
  }
}
```

Rechtslineare Grammatiken (analog Linkslineare Gra.)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

Definition: Eine ~~*kontextfreie*~~ *rechtslineare Grammatik* ist ein Viertupel (Nichtterminale, Terminale, Regeln, Startsymbol). Sie besteht aus

- einer endlichen Menge von *Nichtterminal*en (auch *Variabl*en genannt)
- einer endlichen Menge von *Terminal*en, die disjunkt von den Nichtterminalen ist, also Nichtterminale $\cap$ Terminale = {}
- einer endlichen Menge von *Regel*n (auch *Produktion*en genannt) aus

  Nichtterminale $\times$ ~~(Nichtterminale $\cup$ Terminale)*~~,

  Terminale* $\circ$ (Nichtterminale $\cup$ $\{\varepsilon\}$)

  die typischerweise in der Form P -> Q geschrieben werden
- einem Startsymbol $\in$ Nichtterminale

Streichungen und rote Teile machen deutlich, dass jede rechtslineare Grammatik auch eine kontextfreie Grammatik ist

Beispiel: rechtslineare Grammatik

Nichtterminale × (Terminale* ° (Nichtterminale ∪ {ε}))

L = { w | die Anzahl der a in w ist durch 3 teilbar, b beliebig}

Start -> ε | b Start | a A1 Terminale = {a, b}

A1 -> b A1 | a A2 Nichtterminale = {Start, A1, A2}

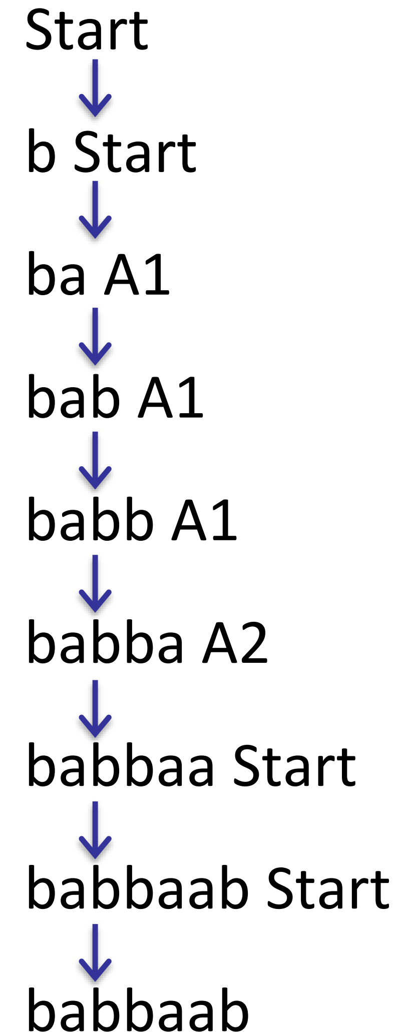
A2 -> b A2 | a Start

anschaulich: nur mit Start -> ε kann Regelnutzung beendet werden, da sonst Nichtterminal im abgeleiteten Wort

Start -> b Start erzeugt beliebig viele b

Start -> a A1 kann a erzeugt werden, durch das Nichtterminalzeichen wird gemerkt, dass es ein A gibt

Regeln für A1 und A2 analog, zurück zum Start nur, wenn es 3 a gibt



endlicher Automat – rechtslineare Grammatik (1/4)

Satz: Eine Sprache ist genau dann von einem endlichen Automaten akzeptierbar, wenn sie von einer rechtlinearen Grammatik erzeugt werden kann

Ansatz: zeige beide Richtungen konstruktiv.

(i) konstruiere zum deterministischen Automaten eine rechtslineare Grammatik

(ii) konstruiere zu rechtslineare Grammatik endlichen nichtdeterministischen Automaten mit ε-Übergängen

endlicher Automat – rechtslineare Grammatik (2/4)

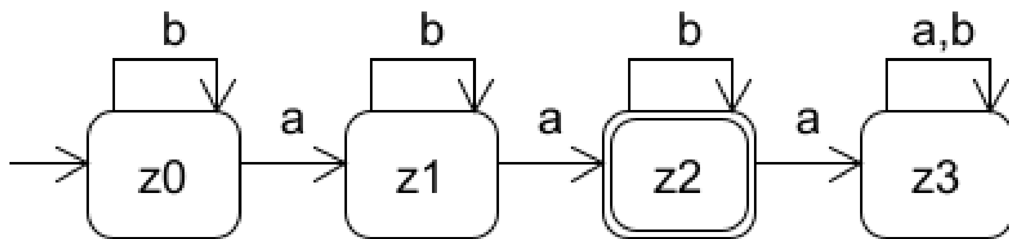
(i) konstruiere zum deterministischen Automaten eine rechtlineare Grammatik

Nichtterminalzeichen sind Zustände des Automaten

jeder Übergang über(z1, a) = z2 wird zu z1 -> a z2

für alle zE $\in$ Endzustände: zE -> ε

das Startsymbol ist der Anfangszustand



z0 -> b z0 | a z1
z1 -> b z1 | a z2
z2 -> b z2 | a z3 | ε
z3 -> b z3 | a z3
(letzte Zeile wegoptimierbar)

endlicher Automat – rechtslineare Grammatik (3/4)

(ii) konstruiere zu rechtslinearer Grammatik endlichen nichtdeterministischen Automaten mit ε-Übergängen

Nichtterminalzeichen werden zu Teilen der Zustandsmenge, Startsymbol wird zu Startzustand, betrachte verschiedene Regelarten

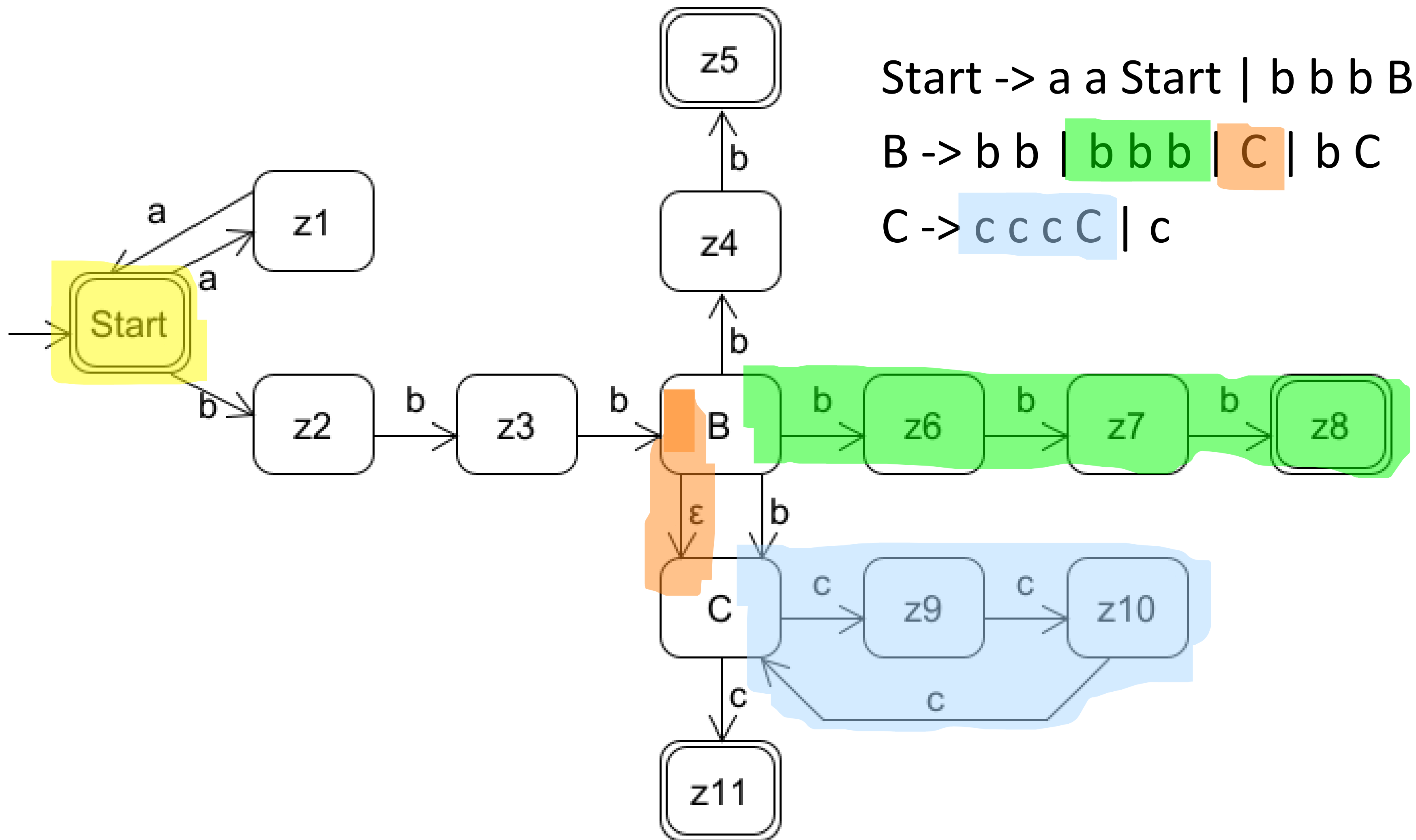
N1 -> a1 a2 ..an N2 (n > 0) : führe jeweils neue Zustände z1, …, zn-1 ein und setze z1 $\in$ über(N1, a1), z2 $\in$ über(z1, a2) , … , N2 $\in$ über(zn-1, an)

N1 -> a1 a2 ..an (n > 0) : führe jeweils neue Zustände z1, …, zn ein und setze z1 $\in$ über(N1, a1), z2 $\in$ über(z1, a2) , … zn $\in$ über(zn-1, an) und zn $\in$ Endzustände

N1 -> N2 : N2 $\in$ ε-über(N1)

N1 -> ε : N1 $\in$ Endzustände

endlicher Automat – rechtslineare Grammatik (4/4)



Reguläre Ausdrücke - Vorgehen

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- reguläre Ausdrücke werden in der Programmierung oft eingesetzt, um bestimmte Worte in einem Text zu finden oder einen Text auf syntaktische Korrektheit zu prüfen
- wir nutzen hier zuerst eine etwas modifizierte Syntax zur Beschreibung von regulären Ausdrücken
- analysieren dann die Ausdrucksmächtigkeit
- schauen uns dann die Syntax in Programmkonstrukten an
- und schauen dann auf Erweiterungen, der originalen regulären Ausdrücke, die in Programmiersprachen ergänzt werden

Reguläre Ausdrücke - Syntax

- *reguläre Ausdrücke* werden (hier) durch folgende Grammatik beschrieben, Reg ist das Startsymbol

  Reg -> (Reg)* | (Reg+Reg) | RegReg | (Reg) | {}

  Reg -> 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 | 0 | 1 | 2 | 3 | 4 |5 | 6 | 7 | 8 | 9
- * auch Kleene-Stern genannt für beliebig oft
- + für Alternative
- Konkatenation ohne Verknüpfungszeichen
- Klammern für Prioritäten, {} für leere Menge
- generell kann man Klammern reduzieren, u. a. wenn Prioritäten festgelegt werden, z. B. ab* ≠ (ab)* , bei uns a(b)* eindeutig, hier genutzte Form auch im Beispiel-Framework umgesetzt
- Praxis: beliebige UniCode-Zeichen erlaubt (Alphabet)

Reguläre Ausdrücke - Semantik

Definition: Sei RegulächeAusdrücke die Menge aller regulären Ausdrücke

Die Semantik von regulären Ausdrücken (ist eine Sprache) und wird durch folgende Abbildung festgelegt

Sprache: ReguläreAusdrücke -> Pot(Alphabet*)

- Sprache((Reg)*) = (Sprache(Reg))*
- Sprache((Reg1+Reg2)) = Sprache(Reg1) ∪ Sprache(Reg2)
- Sprache(Reg1Reg2) = Sprache(Reg1) ° Sprache(Reg2)
- Sprache((Reg)) = (Sprache(Reg))
- Sprache({}) = {}
- Sprache(x) = {x} für jedes Zeichen

Reguläre Ausdrücke - Beispiele

einmal sauber berechnet

Sprache((a + (bc)*)) = (Sprache(a + (bc)*)) = (Sprache(a) $\cup$ Sprache((bc)*))

= ({a} $\cup$ Sprache((bc))*) = ({a} $\cup$ (Sprache(bc))*)

= ({a} $\cup$ (Sprache(b) $\circ$ Sprache(c))*) = ({a} $\cup$ ({b} $\circ$ {c})*)

= ({a} $\cup$ ({bc})*) = {a} $\cup$ {ε, bc, bcbc, bcbcbc, ...}

Sprache(({})*) = {ε}

Sprache(((0 + 1))*) = {ε, 0, 1, 00, 01, 10, 11, 000, 001, ...}

Sprache(((x + y) + z) (0 + (1 + 2))) = {x0, x1, x2, y0, y1, y2, z0, z1, z2}

gibt viele Rechenregeln, z. B.:

x + x = x	a(x + y) = (ax + ay)	x{} = {}	(x + {}) = x
x({})* = x	(x + y)a = (xa + ya)	(x + y) = (y + x)	

((x + y) + z) = (x + (y + z))

Reguläre Ausdrücke und endliche Automaten (1/9)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

Video

Satz: Eine Sprache ist genau dann von einem endlichen Automaten akzeptierbar, wann sie die Semantik eines regulären Ausdrucks ist.

Beweisidee: zwei Richtungen,

(i) konstruiere zu einem regulären Ausdruck einen endlichen nichtdeterministischen Automaten mit ε-Übergängen

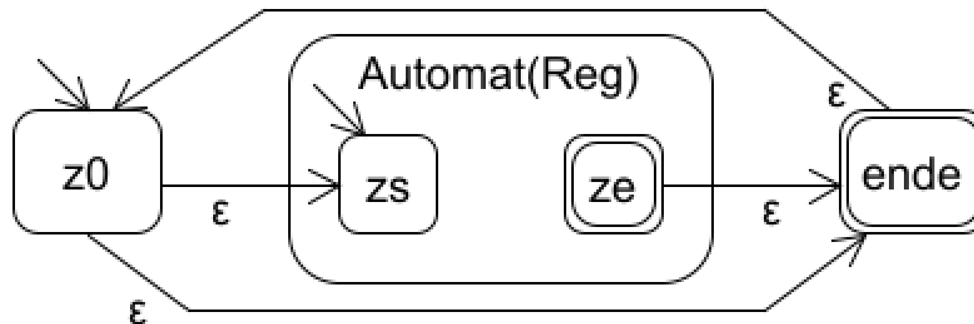
(ii) konstruiere zu einem deterministischen endlichen Automaten einen regulären Ausdruck

Reguläre Ausdrücke und endliche Automaten (2/9)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

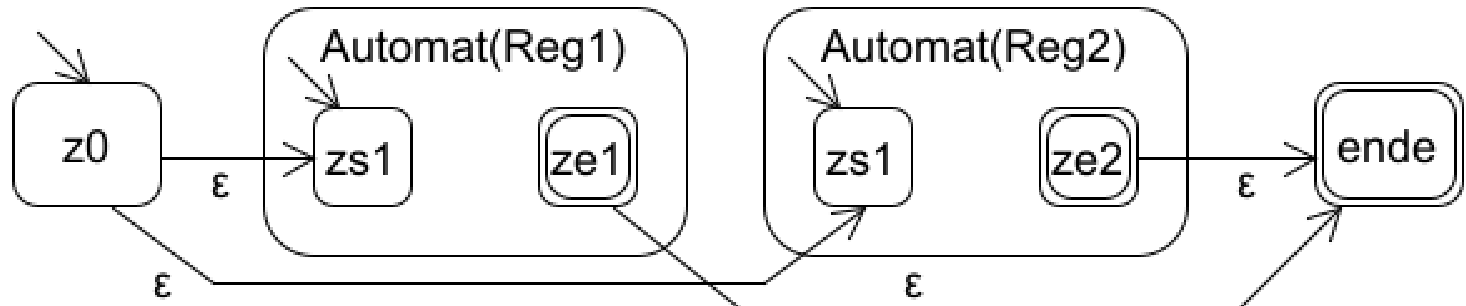
(i) konstruiere zu einem regulären Ausdruck einen endlichen nichtdeterministischen Automaten mit ε-Übergängen

- Ansatz: zu jeder Erstellungsregel für reguläre Ausdrücke gib einen passenden Automaten an
- Automat((Reg)*)



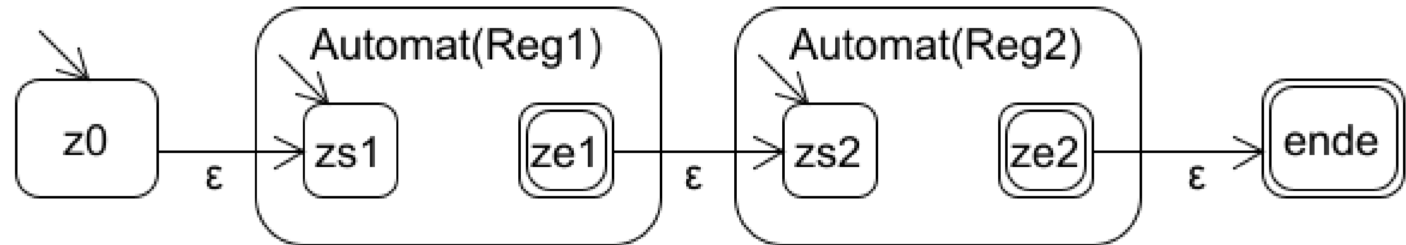
[interne Endzustände sind am Ende keine Endzustände]

- Automat((Reg1+Reg2))

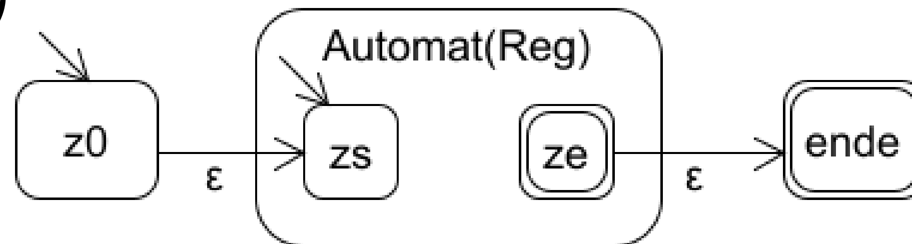


Reguläre Ausdrücke und endliche Automaten (3/9)

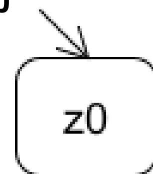
- Automat(Reg1Reg2)



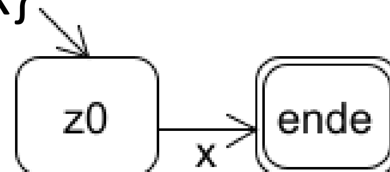
- Automat((Reg))



- Automat({}) = {}

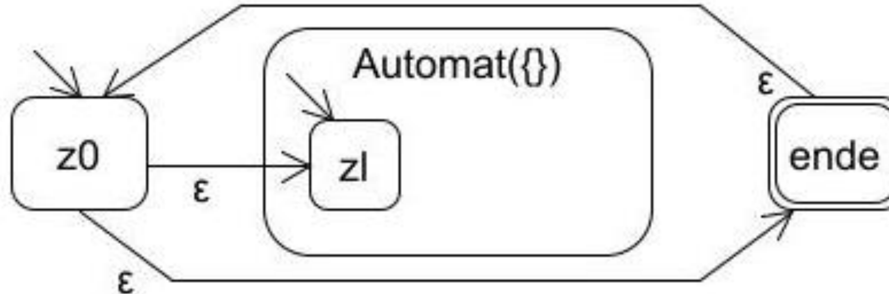


- Automat(x) = {x}



Reguläre Ausdrücke und endliche Automaten (4/9)

- {}*



- (a + (bc)*)

Automat(a)
z0
zs1
ze1
ende
ε
a
ε
ε
Automat((bc)*)
Automat(bc)
Automat(b)
Automat(c)
z4
zs4
zs2
ze2
zs3
ze3
ze4
z5
ε
ε
b
ε
c
ε
ε
ε
ε
ε

Reguläre Ausdrücke und endliche Automaten (5/9)

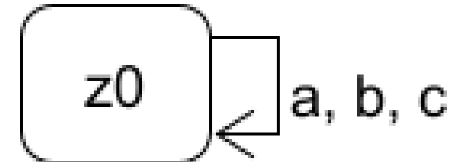
- (ii) konstruiere zu einem nichtdeterministischen endlichen Automaten mit ε–Übergängen einen regulären Ausdruck
- Ansatz: An Kanten des Automaten steht bereits ein regulärer Ausdruck (ein Zeichen oder {}*), verschmelze schrittweise Zustände und annotiere Kanten mit passenden regulären Ausdrücken, bis nur noch ein Anfangszustand und Endzustände existieren
- sinnvolle Vereinfachung, füge neuen Startzustand und Endzustand hinzu, die mit ursprünglichen mit ε–Kanten verbunden werden
- zentraler Algorithmus:

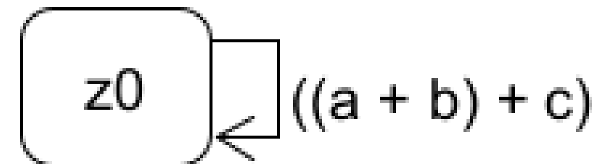
```
for(z: Zustand) {
  if (z ≠ Startzustand && z ∉ Endzustände) { //alle alten Zu.
    ersetzeZustand(z)
}
```

- Variante: mit Gleichungssystem (Arden)

```
entferne(Zustand z):
  looper = {} // Set von Zeichen bzw. reg. Ausdrücken
  for(a: Alphabet){
    if (ueber(z,a) == z) {
      looper.add(a)
      ueber.remove(z, a)
    }
  }
  if (looper ≠ {}) {
    ueber(z, alsAusdruck(looper)) = z
  }

  for(von: Zustände) { // pruefe ob von von über z nach nach
    for(nach: Zustände) {
    vonReg = null
    nachReg = null
```

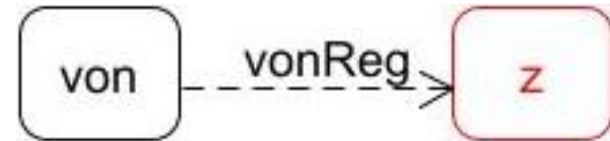




Reguläre Ausdrücke und endliche Automaten (7/9)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

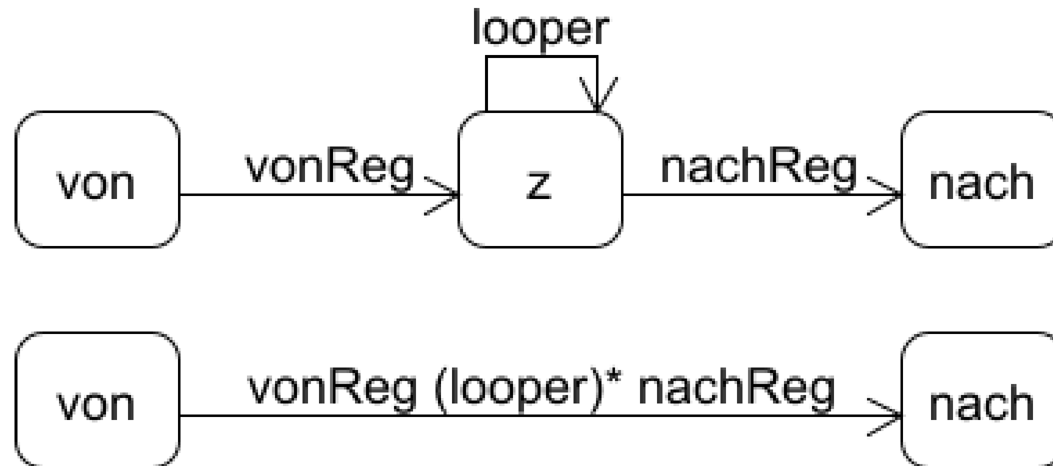
```
vonReg = null
nachReg = null
for(zust,reg : über.definitionsbereich()) {
  if (zust == von && z == über(zust, reg) && von≠ z){
    if (vonReg == null) {
      vonReg = reg
    } else {
      vonReg = (vonReg + reg) // regulärer Ausdruck
  }
  if (zust == z && nach == über(zust, reg) && nach≠ z){
    if (nachReg == null) {
      nachReg = reg
    } else {
      nachReg = (nachReg + reg) // regulärer Ausdruck
  }
}
```



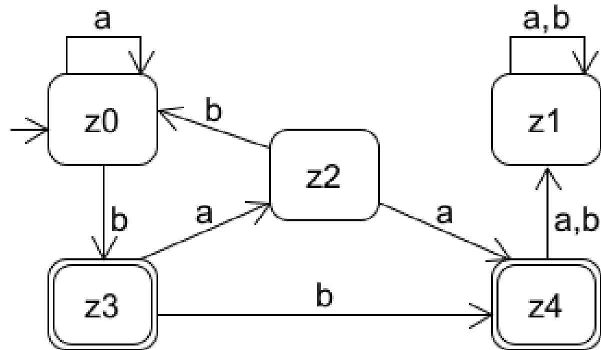
Reguläre Ausdrücke und endliche Automaten (8/9)

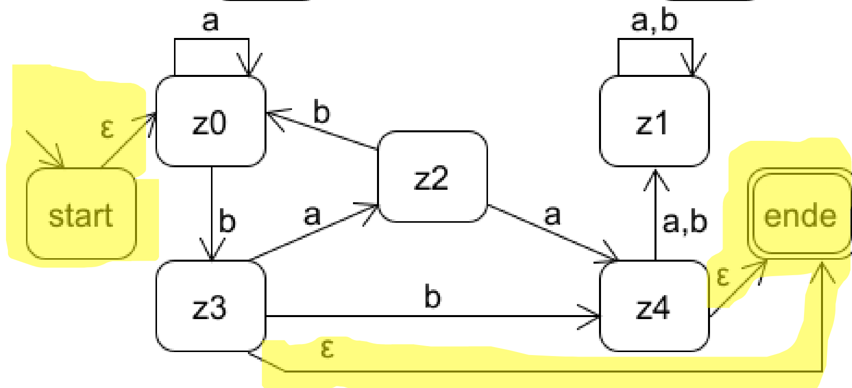
```
if (vonReg ≠ null && nachReg ≠ null) {
  if (looper ≠ null) {
    über(von, „vonReg (looper)* nachReg“) = nach
  } else {
    über(von, „vonReg nachReg“) = nach
  }
}
}
```

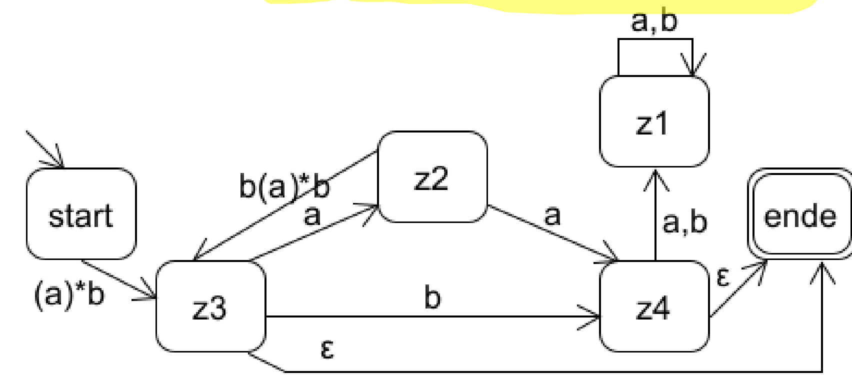
für if:

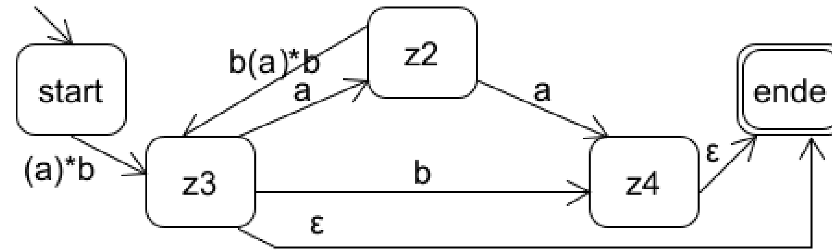


Reguläre Ausdrücke und endliche Automaten (9/9)

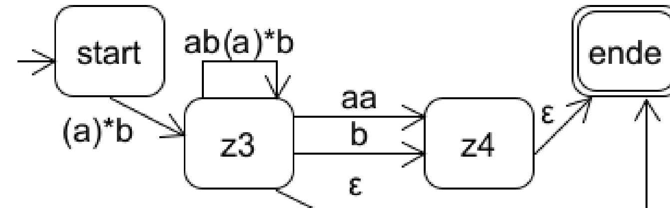


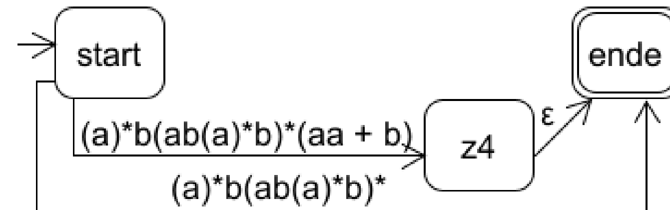


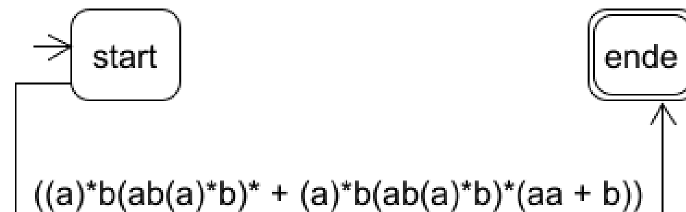




da z1 kein „nach“ fällt er automatisch weg







einige Varianten:
(((a + bab))*b + ((a + bab))*b(aa + b))
(((a + bab))*b + ((a + bab))*b(b + aa))
(((a + bab))*b(b + aa) + ((a + bab))*b)

Reguläre Ausdrücke in Java (1/5)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
public class Main {

  public static void pruefe(String regAusdruck, String s){
    System.out.println(Pattern.matches(reAusdruck, s));
  }

  public static void main(String[] arg){
    pruefe("q","q");                // true
    pruefe("qq","q");               // false
    pruefe("q*","q");               // true   // beliebig oft
    pruefe("q*","");                // true
    pruefe("q+","");                // false  // mindestens einmal
    pruefe("q+","q");               // true
    pruefe("q|r","q");              // true   // Alternative (oder)
    pruefe("q|r","r");              // true
    pruefe("q|r","s");              // false
    pruefe("(q|r)*","qqrrqq");  // true
```

Reguläre Ausdrücke in Java (2/5)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
pruefe("..","qr");            // true  // ein bel. Zeichen
pruefe("..","qrs");           // false
pruefe(".*","blubb");         // true
pruefe("q?","");              // true  // ein- oder keinmal
pruefe("q?","q");             // true
pruefe("q?","qq");            // false
pruefe("q{3}","qq");          // false // genau 3-mal
pruefe("q{3}","qqq");         // true
pruefe("q{3,}","qq");         // false // mindestens 3-mal
pruefe("q{3,}","qqq");        // true
pruefe("q{3,5}","qq");        // false // zwischen 3 u 5-mal
pruefe("q{3,5}","qqqqqq");    // false
pruefe("q\\*","qq");          // false // Fluchtsymbol
pruefe("q\\*","q*");          // true
pruefe("q(\\*)","q*");        // true  // Klammern erlaubt
pruefe("q\\\\","q\\");        // true  //   \ immer \\
```

Reguläre Ausdrücke in Java (3/5)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
pruefe("[qwe]","w");           // true  Alternative (oder)
pruefe("[^qwe]","w");          // false // keines der Zeichen
pruefe("[0-9][^a-f]","1A");    // true  // von - bis
pruefe("[0-9][^a-f]","1a");    // false
pruefe("\\d+","12345");        // true // \d Ziffer
pruefe("\\D+","12345");        // false  // \D keine Ziffer
pruefe("\\s*"," \n\t\f\r ");  // true // \s Weißraum
pruefe("\\S*","hall\noo");     // false // \S kein Weißraum
pruefe("\\w*","i_a");          // true // \w = [a-zA-Z_0-9]
pruefe("\\W*","  \n ");        // true  // nicht \w
pruefe("\\p{Lower}\\p{Upper}","aA"); // true  // klein groß
pruefe("\\p{Lower}\\p{Upper}","Aa"); //false  // gibt mehr
                                      // dieser Zeichenklassen
pruefe("(?i)aAa(?-i)Aa","aaaAa"); // true // (?i) Flag Case
                                         // insensitive
pruefe("(?i)aAa(?-i)Aa","aaaaa"); // false // (?-i) Flag
                                          // ausschalten
```

Reguläre Ausdrücke in Java (4/5)

```
pruefe(".*","bl\nub\nb");   // false  // . kein Zeilenumbruch
pruefe("(?s).*","bl\nub\nb");// true  // . auch für
                                      // Steuerzeichen
pruefe("^a.*","aaa");       // true // ^ markiert Zeilenanfang
pruefe("^a.*"," aaa");      // false
pruefe(".*a$","aaa");       // true // $ markiert Zeilenende
pruefe(".*a$","aaa ");      // false

Pattern pat = Pattern.compile("\\d+");
Matcher mat = pat.matcher("1. Die Antwort ist 42");
while(mat.find()){
  System.out.println(mat.group() + " : "
      + mat.start() + "-" + mat.end());
}
```

```
1 : 0-1
42 : 19-21
```

Reguläre Ausdrücke in Java (5/5)


```
    Pattern pat2 = Pattern.compile("a+");
    Matcher mat2 = pat2.matcher("aaaa");
    while(mat2.find()) { // greedy: Suche maximaler Wortlänge
      System.out.println(mat2.group() + " : "
          + mat2.start() + "-" + mat2.end());
    }

    Pattern pat3 = Pattern.compile("b+?");
    Matcher mat3 = pat3.matcher("bbbb");
    while(mat3.find()){ // mit ? nicht greedy
      System.out.println(mat3.group() + " : "
          + mat3.start() + "-" + mat3.end());
    }

    Pattern pat4 = Pattern.compile("a|o"); //[ao]
    String[] splits = pat4.split("Hallo Costa!");
    for(String s:splits){
      System.out.println(s);
    }
  }
}
```

```
aaaa : 0-4
```

```
b : 0-1
b : 1-2
b : 2-3
b : 3-4
```

```
H
ll
 C
st
!
```

Reguläre Ausdrücke in der Programmierung

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- reguläre Ausdrücke werden oft um neue Operatoren erweitert
- Erweiterungen oft aus Bequemlichkeit, die die Ausdrücke und ihre Nutzung kompakter machen, ohne die Sprachmächtigkeit zu ändern (rückübersetzbar, z. B. + für 1 oder mehr; ? 0 oder 1; [^...] nicht; Zugriff auf vorher verarbeitetes oder übernächstes Zeichen)
- Gibt echte Erweiterungen, wobei Sprachen verarbeitet werden, die nicht durch klassische reguläre Ausdrücke beschreibbar sind

```
pruefe("(a*)b\\1","aaaabaaaa");   // true
pruefe("(a*)b\\1","aaba");        // false
```

  Klammern definieren Gruppen, ab 1 nummeriert; mit \i kann auf i-te Gruppe zugegriffen werden, es wird gleiches Wort gefordert, Ausdruck beschreibt nicht reguläres $\{a^n b a^n \mid n \geq 0\}$

- A. Srivastava, Java 9 Regular Expressions (English Edition), Packt Publishing, Birmingham (UK), 2017
  https://docs.oracle.com/javase/tutorial/essential/regex/

Fazit

- mit Automaten viele unterschiedliche reale Systeme, bzw. viele Teile eines komplexen Systems gut spezifizierbar
- einige Spezifikationsvarianten, u. a. mit ε-Übergängen und Nichtdeterminismus
- Automatenarten ineinander wandelbar; aber Zustandsraumexplosion möglich
- Sprachanalyse mit regulären Ausdrücken sehr gut machbar; diese einfach in Automaten umwandelbar (und umgekehrt)
- viele reale Anforderungen automatisch überprüfbar

Video

5. Sprachklassen

zentrale Inhalte:

- Sprachhierarchie
- Wortproblem
- Abschlusseigenschaften

Glossar

Abschlusseigenschaften

Chomsky Hierarchie

kontextfreie Sprachen

kontextsensitive Sprachen

kontextsensitive Grammatik

Pumping Lemma Typ 2

reguläre Sprachen

rekursiv aufzählbare Sprachen

was wir über Sprachen wissen und wissen wollen

- eine Sprache ist eine Teilmenge von Alphabet*
- es gibt Möglichkeiten zum Erzeugen / Beschreiben (Grammatiken, reguläre Ausdrücke) und zum Akzeptieren / Erkennen (Turing-Maschinen, endliche Automaten)
- es gibt Modelle, die gleichmächtig sind (z. B. endliche Automaten und reguläre Ausdrücke) und die verschieden mächtig sind (z. B. endliche Automaten und kontextfreie Grammatiken; Erinnerung Pumping-Lemma)
- Frage 1: Ist das Wortproblem (Wort gehört zur Sprache?) entscheidbar?
- Frage 2: Können die so entstehenden Sprachklassen genauer gegeneinander separiert werden?
- Frage 3: Sind die Sprachklassen unter Mengenoperatoren (Vereinigung, Schnitt, Komplement) abgeschlossen
- Frage 4: [sehr viel mehr Interessantes]

reguläre Sprachen, Typ 3

- Modelle: rechtslineare Grammatik, endliche Automaten, reguläre Ausdrücke
- Wortproblem entscheidbar, z. B. Automat abarbeiten
- Pumping-Lemma für reguläre Sprachen zeigte $\{a^n b^m \mid n < m\}$ ist nicht regulär, aber mit kontextfreier Grammatik beschreibbar

  Start -> a Start b | B   B -> b | b B

Anmerkung zu Typ-Nummern (Chomsky[-Schützenberger]-Hierarchie)

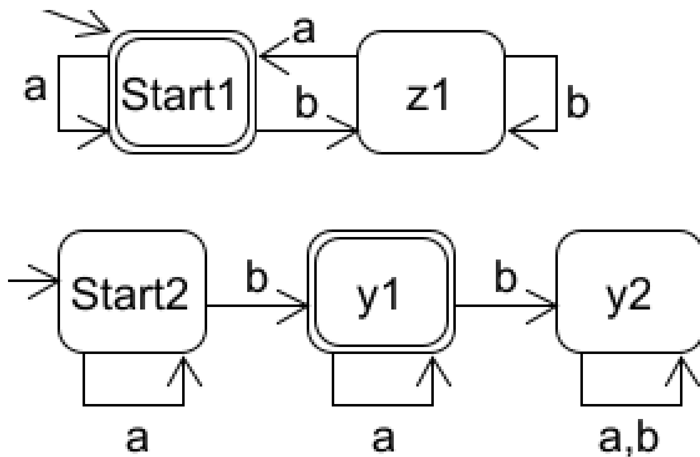
- sind am Ende des Kapitels halbwegs intuitiv (jede Sprachklasse ist echte Teilmenge der anderen Sprachklasse (Typ3 $\subseteq$ Typ2))
- aber, es gibt viele klar abgrenzbare Sprachklassen dazwischen, deshalb sind Nummern Willkür (die endlichen Sprachen, sind echte Teilmenge von Typ 3, haben aber keine Nummer)

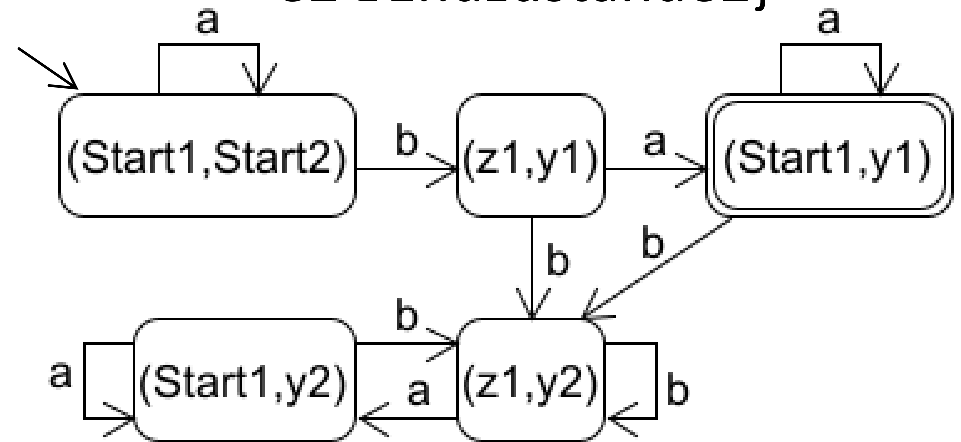
Abschlusseigenschaften regulärer Sprachen

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Vorüberlegung: zwei deterministische Automaten parallel nutzbar (Alphabet, Zustände1 $\times$ Zustände2, Endzustände12, über12, (Startzustand1,Startzustand2)) mit

  über12((z1,z2), a) = (über1(z1,a), über2(z2,a))
- für Schnitt: Endzustände12 = {(e1, e2) | e1 $\in$ Endzustände1 und e2 $\in$ Endzustände2}
- für Vereinigung: Endzustände12 = {(e1, e2) | e1 $\in$ Endzustände1 oder e2 $\in$ Endzustände2}





- für Komplement: EndzuständeKompl = Zustände - Endzustände

kontextfreie Sprachen, Typ 2

- erzeugt von kontextfreien Grammatiken, akzeptiert von nichtdeterministischen Kellermaschinen (passt leider nicht in die VL, grob endlicher Automat mit Stapel, der oben gelesen und geschrieben werden kann)
- bekannt: Wortproblem lösbar (unser Ansatz über Chomsky-Normalform)
- typisch Form: Sprache, die ausgehend von Nichtterminalzeichen „gleichmäßig“ nach links und rechts wächst, z. B. $\{a^n b^n c^m \mid n,m \geq 0\}$

  Start -> X C   X -> a X b | ε   C -> c C | ε
- (erfüllt nebenbei nicht das Pumping-Lemma für regulären Sprache, da ab einer bestimmten Länge auch Worte der Form $a^n b^n$ (m = 0) aufpumpbar sein müssen)

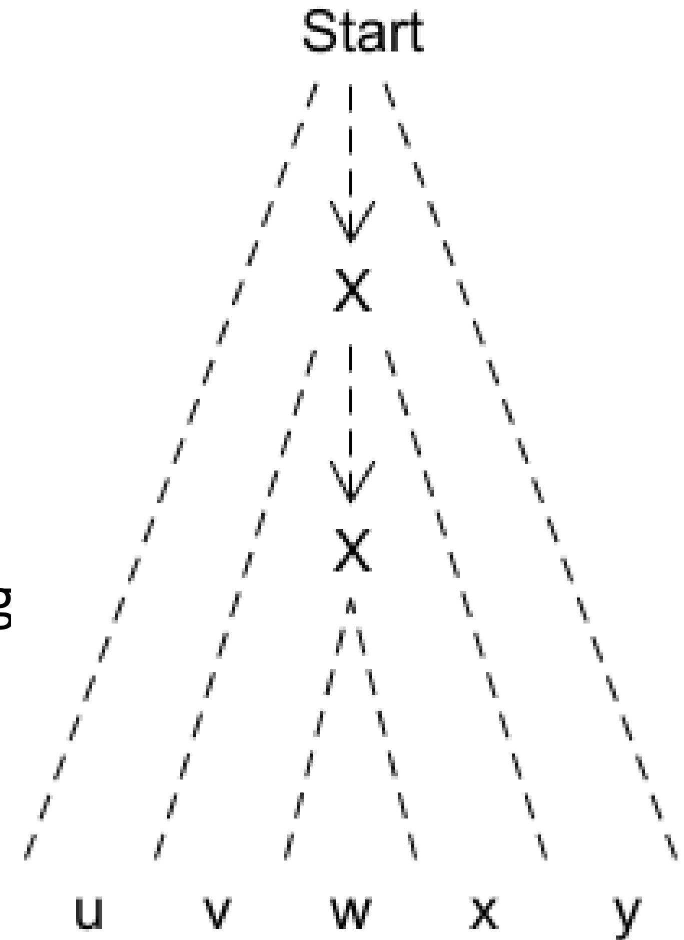
Pumping Lemma für kontextfreie Sprachen

- Erinnerung: Satz (Pumping-Lemma 1 für reguläre Sprachen): Wenn eine Sprache von einem endlichen Automaten akzeptierbar ist, dann gibt es ein n, so dass für alle Worte w der Sprache, die **länger als n** sind, es eine Aufteilung von w gibt mit w = xyz, so dass auch xy^jz zur Sprache gehört, für $0 \leq j$ und x, y, z $\in$ Alphabet *, $y \neq \varepsilon$ und die Länge von xy ist $|xy| \leq n$.

- Satz (Pumping-Lemma 2 für kontextfreie Sprachen): Wenn eine Sprache von einer kontextfreien Grammatik erzeugbar ist, dann gibt es ein n, so dass für alle Worte w der Sprache, die **länger als n** sind, es eine Aufteilung von w gibt mit w = uvxyz, so dass auch uv^jxy^jz zur Sprache gehört, für $0 \leq j$ und u,v, x, y, z $\in$ Alphabet* und $vy \neq \varepsilon$ (also zusammen v oder y dürfen das leere Wort sein) und die Länge von vxy ist $|vxy| \leq n$

Beweisidee für Pumping-Lemma 2

- Jede Grammatik kann in eine sprachäquivalente Grammatik ohne Kettenregeln X -> Y transformiert werden.
- Wenn es eine Grammatik gibt, dann hat sie i Nichtterminale und für die rechte Seite der Regeln gibt es eine maximale Länge von m.
- Für ein Wort der Sprache mit der Länge größer als i*m gilt, dass sich mindestens ein Nichtterminal, genauer eine Regelanwendung X -> W in der Ableitung wiederholen muss.
- Da W mindestens ein Nichtterminal enthält, ist das der Ursprung des aufpumpbaren Teilwortes.



Beispielanwendung des Pumping-Lemmas 2

Aufgabe: Begründen oder widerlegen Sie, dass $\{a^n b^n c^n \mid n \geq 0\}$ mit einer kontextfreien Grammatik erzeugbar ist.

Begründung wäre eine Angabe einer Grammatik (geht nicht)

Widerlegen: Zeige, dass die Sprache das Pumping-Lemma 2 nicht erfüllt; es also für lange Worte die Zerlegung $w = uv^jxy^jz$ nicht gibt

Bestimme Zerlegungsmöglichkeiten für $a^k b^k c^k$ (k sehr groß): damit Wort aufpumpbar und in der Sprache bleibt, gibt es nur die Möglichkeiten

(i) dass v oder y jeweils aus verschiedenen Zeichen bestehen, dann gehört das Wort mit j = 2 aber nicht zur Sprache

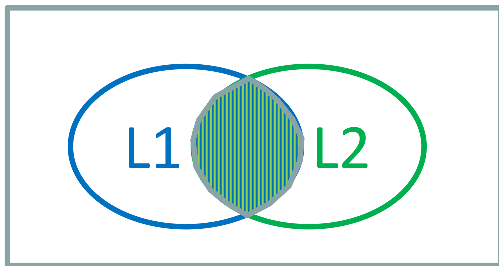
(ii) dass v und y aus den gleichen Zeichen bestehen

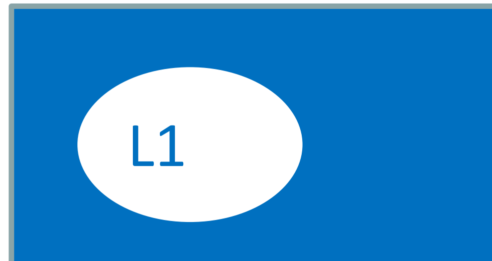
(iii) dass v aus einem und y aus einem anderen Zeichen besteht

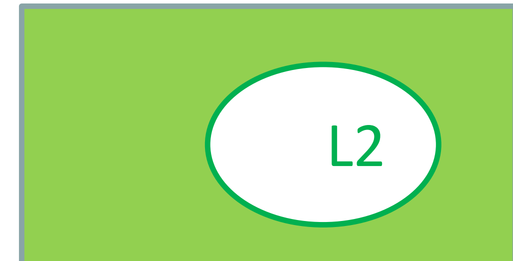
bei (ii) und (iii) aber Widerspruch u. a. mit j = 0, da Anzahl des dritten Zeichens nicht reduziert wird

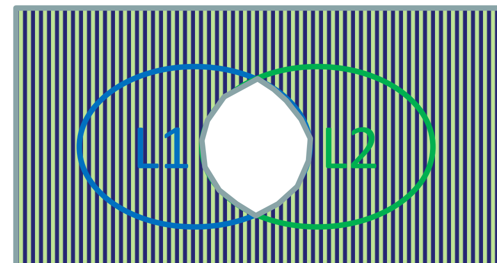
Abschlusseigenschaften kontextfreier Sprachen

- Vereinigung: ja,   NeuesStartsymbol -> Start1 | Start2
- Schnitt: nein, da $\{a^n b^n c^n \mid n \geq 0\} = \{a^n b^n c^m \mid n,m \geq 0\} \cap \{a^m b^n c^n \mid n,m \geq 0\}$
- Komplement: falls Ja, müsste Typ2 auch gegen Schnitt abgeschlossen sein, da
  $L1 \cap L2 = \text{Komplement}(\text{Komplement}(L1) \cup \text{Komplement}(L2))$









kontextsensitive Sprachen, Typ1 (nur viel zu kurz)

Video

durch (kontextsensitive) Grammatiken erzeugbar

- kontextsensitiv bedeutet, dass beim Einsatz von Regeln das Umfeld, also die umgebenden Zeichen, beachtet werden
- auf der linken Seite der Regeln können Nichtterminale gemischt mit Terminalen als Wort stehen, Alphabet = Nichtterminale $\cup$ Terminale
  Regeln: uXv -> uyv mit X Nichtterminale, $y \in Alphabet^+$,
  $u,v \in Alphabet^*$ (u und v können ε sein), Start -> ε nur erlaubt, wenn Start nicht auf der rechten Seite
- äquivalent: monotone Grammatiken: P -> Q, dann $|P| \leq |Q|$, Start -> ε
- ({B, C, S, Start}, {a, b, c}, Prod, Start) mit Start -> S | ε S -> aSBC | aBC
  CB -> BC aB -> ab bB -> bb bC -> bc cC -> cc
- Start -> aSBC -> aaBCBC -> aaBBCC -> aabBCC -> aabbCC -> aabbcC -> aabbcc .... damit $\{a^n b^n c^n \mid n \geq 0\}$ ist kontextsensitiv
- ist abgeschlossen für Vereinigung, Schnitt und Komplement, Wortproblem entscheidbar, hat Kuroda-Normalform

rekursiv aufzählbare Sprachen, Typ 0

- Sprachen, die von einer Turing-Maschine akzeptiert werden
- Sprachen, die von einer Grammatik mit folgenden erlaubten Regelarten akzeptiert werden Regeln: uXv -> y mit X Nichtterminale, u,v,y $\in$ Alphabet* (u und v können ε sein, Kontext kann verschwinden)
- bekannt: Wortproblem nicht entscheidbar
- abgeschlossen gegen Schnitt und Vereinigung (einfach zwei Turing-Maschinen mit zwei Bändern parallel als eine Turing-Maschine laufen lassen)
- nicht abgeschlossen gegen Komplement, da sonst Wortproblem entscheidbar (einfach zwei Turing-Maschinen, eine die die Sprache akzeptiert und eine die das Komplement akzeptiert mit zwei Bändern parallel als eine Turing-Maschine laufen lassen; nach Abarbeitung schauen, welche Teil-Turing-Maschine angehalten ist)

Mengenzusammenfassung

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

beliebige Sprachen

Typ-0: rekursiv aufzählbar: z. B. Turing-Maschine

Typ-1: kontextsensitiv: z. B. Grammatik

Typ-2: kontextfrei: z. B. Grammatik

Typ-3: regulär: z. B. Automat, regulärer Ausdruck, Grammatik

- eindeutige kontextfreie Sprachen, endliche Sprachen, ... fehlen

Abschlusseigenschaften

	Typ 0 rekursiv aufzählbar	Typ 1 kontext-sensitiv	Typ 2 kontextfrei	Typ 3 regulär	endlich
Vereinigung	X	X	X	X	X
Schnitt	X	X		X	X
Komplement		X		X	
Konkatenation	X	X	X	X	X
Kleene Abschluss	X	X	X	X	

- viele Varianten: Abschluss gegen Schnitt mit regulärer Sprache

Äquivalenz kontextfreier Sprachen nicht entscheidbar

- für zwei gegebene Grammatiken nicht automatisch entscheidbar, ob sie die gleiche Sprache erzeugen
- (natürlich gibt es spracherhaltende Transformationen)
- bedeutet z. B.: benutzen zwei Compiler-Hersteller für die gleiche Sprache eigene Grammatiken, ist nicht sichergestellt, dass sie wirklich die gleiche Sprache unterstützen
- Grammatiken auch woanders versteckt: z. B: XML-Schema, mit denen gültige XML-Dokumente definiert werden, z. B. JSON-Schema, wenn sie zur Definition gewünschter JSON-Dokumente definiert werden

6. Komplexität

zentrale Inhalte:

- Laufzeit
- P und NP
- NP vollständig (NP hart)
- Reduktion

Glossar

Landau-Notation

nichtdeterministische Turing-Maschine

NP

NP hart

NP vollständig

O-Notation

P

polynomielle Reduktion

Polynom

SAT

Komplexität - Einleitung

- Laufzeit: In Abhängigkeit von der Größe der Eingabe, wie lange braucht das Programm maximal (oder im Durchschnitt; hier nicht betrachtet)
- Speicherverbrauch: In Abhängigkeit von der Größe der Eingabe, wieviel Speicher braucht das Programm maximal (oder im Durchschnitt); Thema in dieser VL nicht betrachtet
- Average- und Worst-Case interessant, hier nur Worst (maximal)

- zur Laufzeit wird eine Programmiersprache oder Maschinen-Modell als Grundlage benötigt; hierzu werden Turing-Maschinen genutzt
- Lineare Faktoren werden typischerweise vernachlässigt

bekannt (?)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Laufzeit (und Speicherverbrauch) werden intensiv in „Algorithmen und Datenstrukturen“ betrachtet

Effizienz: O-Notation

- Definition **Wachstumsordnung** (*O*- oder *Landau*-Notation):

  Eine Funktion $f(n)$ ist in der Wachstumsordnung $O(g(n))$ mit einer Funktion $g: \mathbb{N} \rightarrow \mathbb{R}^{+}$, wenn es Konstanten $n_0 > 0$ und $c > 0$ mit $f(n) \leq c \cdot g(n)$ für $n \geq n_0$ gibt.
- $f(n)$ kann z.B. Laufzeitfunktion entsprechen $T(n)$
- Anschaulich: $f(n)$ wächst höchstens so stark wie $g(n)$

H. Eikerling, Folien zu AuD, HS OS, SoSe 21

Erinnerung: Von Turing-Maschine berechnete Funktion

Definition: Sei ein Alphabet ohne #-Zeichen gegeben. Eine Funktion

$$f: (Alphabet^*)^m \text{ -> } (Alphabet^*)^n$$

heißt *Turing-Maschinen-berechenbar*, wenn es eine Turing-Maschine TM=(Zustände, Alphabet‘, Überführungsfunktion, Start) mit # $\in$ Alphabet‘, Alphabet $\subseteq$ Alphabet‘ gibt, so dass für alle w1,…,wm, u1,…,um $\in$ Alphabet* gilt:

- f(w1,…,wm) = (u1,…,un) genau dann wenn es eine terminierende Berechnung Start, #w1#w2#...#wm$\underline{\#}$ ->* z, #u1#u2#...#un$\underline{\#}$ gibt
- f(w1,…,wm) ist undefiniert genau dann wenn TM in #w1#w2#...#wm$\underline{\#}$ startet und die Rechnung hängt oder nicht terminiert
- die Anzahl der Schritte, die eine terminierende Berechnung benötigt, wird Laufzeit(-Funktion) benannt
- vereinfacht wird einer Eingabe der Länge n (Summe der Längen der wi) eine Laufzeitfunktion zugeordnet

Erinnerung: Funktion - Entscheidung

Komplexitätsprobleme werden typischerweise als Entscheidungsprobleme formuliert, typisch wird geprüft, ob ein Wort zu einer Sprache gehört

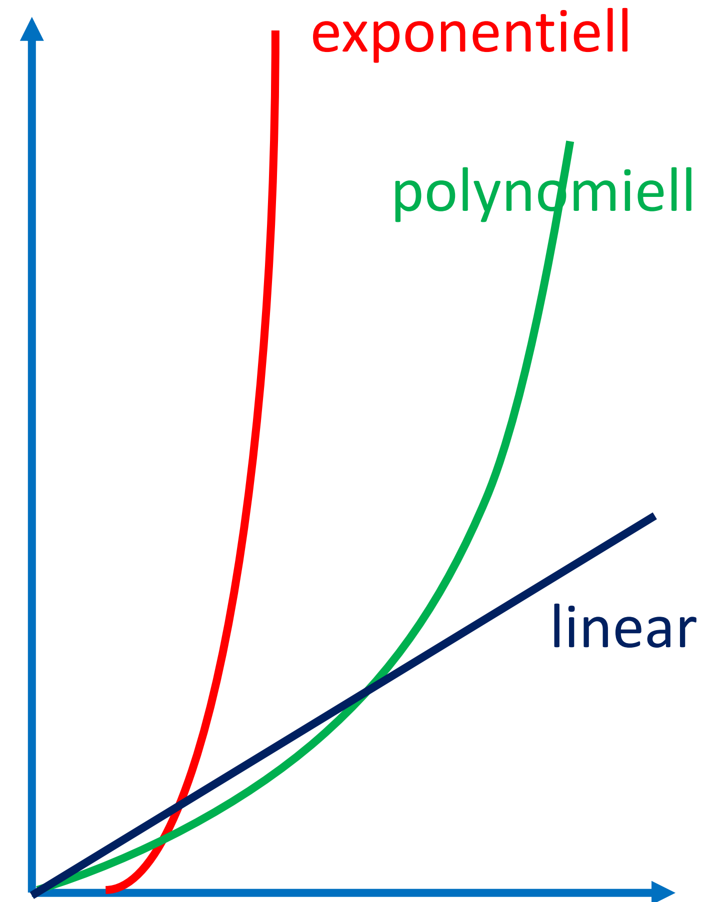
- Jede Sprachzugehörigkeit kann mit ihrer charakteristischen Funktion beschrieben werden, für eine Sprache, $f_{Sprache}$: Alphabet* -> 0,1
  $f_{Sprache}(x) = 0$, wenn $x \notin$ Sprache $f_{Sprache}(x) = 1$, wenn $x \in$ Sprache
- L = {} , dann $f_L(x) = 0$, für alle x
- Jede Funktion kann als Sprache interpretiert werden, genauer über Ihren Funktionsgraphen, z. B. f: M1 -> M2, dann
  Sprache(f) = {w1#w2 | mit f(w1) = w2}
- f(x, y) = x + y M(f) = {x#y#z | x+y = z}

Anmerkung: Bei Berechnungen darf statt an Turing-Maschinen an normale Programmiersprachen gedacht werden

Polynome

- Polynom-Funktion

  $f(x) = a1 + a2x + a3x^2 + a4x^3 + \ldots + amx^n$
- n heißt Grad des Polynoms
- Polynome abgeschlossen gegen Addition und Multiplikation
- obiges f liegt in $O(x^n)$, Konstanten und kleine Polynome egal
- generell: polynomielle Laufzeit meist noch akzeptabel (real Grad maximal 3)
- gibt noch viele andere Funktionsmengen, z. B. O(n log n)



P

Definition: Eine Turing-berechenbare Funktion liegt in der Menge der in polynomieller Laufzeit berechenbaren Funktionen P, wenn es eine die Funktion berechnende Turing-Maschine und ein Polynom poly gibt, so dass die Laufzeitfunktion der Turing-Maschine in O(poly) liegt.

- Wichtig: Es muss nur eine (geschickt geschriebene) Turing-Maschine mit der gewünschten Laufzeitfunktion geben
- Aus A&D: Sortieren gehört in die Klasse P

- Hinweis: Wir betrachten hier die Zeitkomplexität, sehr ähnlich kann auch die Bandkomplexität betrachtet werden. Dabei geht es um die maximal benötigte Anzahl von Bandplätzen.

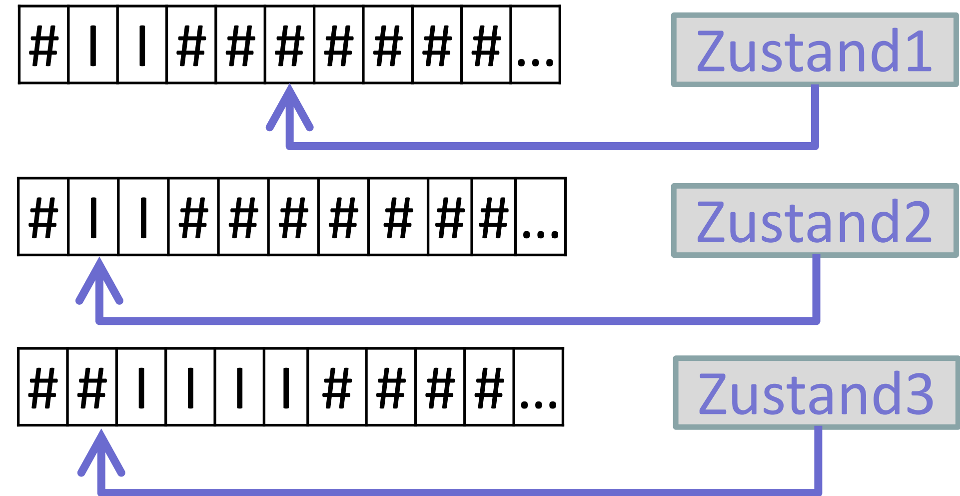
Turing-Maschinen-Varianten

Erinnerung: bisher bekannte Turing-Maschinen-Varianten:

m Bänder

m Zustände

m S-Leseköpfe



Satz: Die Definition von P ist unabhängig von der gewählten Turing-Maschinen-Variante.

Beweisidee: Jedes m kann zur konstanten Beschleunigung genutzt werden, grobe Beschleunigung um konstanten Faktor m^3; Umwandlung in Standard-Turing-Maschine interessante Bastelaufgabe

Anmerkung: natürlich ist Beschleunigung um Konstante praktisch relevant

Nichtdeterministische Turing-Maschine

- Ähnlich wie bei Automaten können auch Turing-Maschinen nichtdeterministisch definiert werden

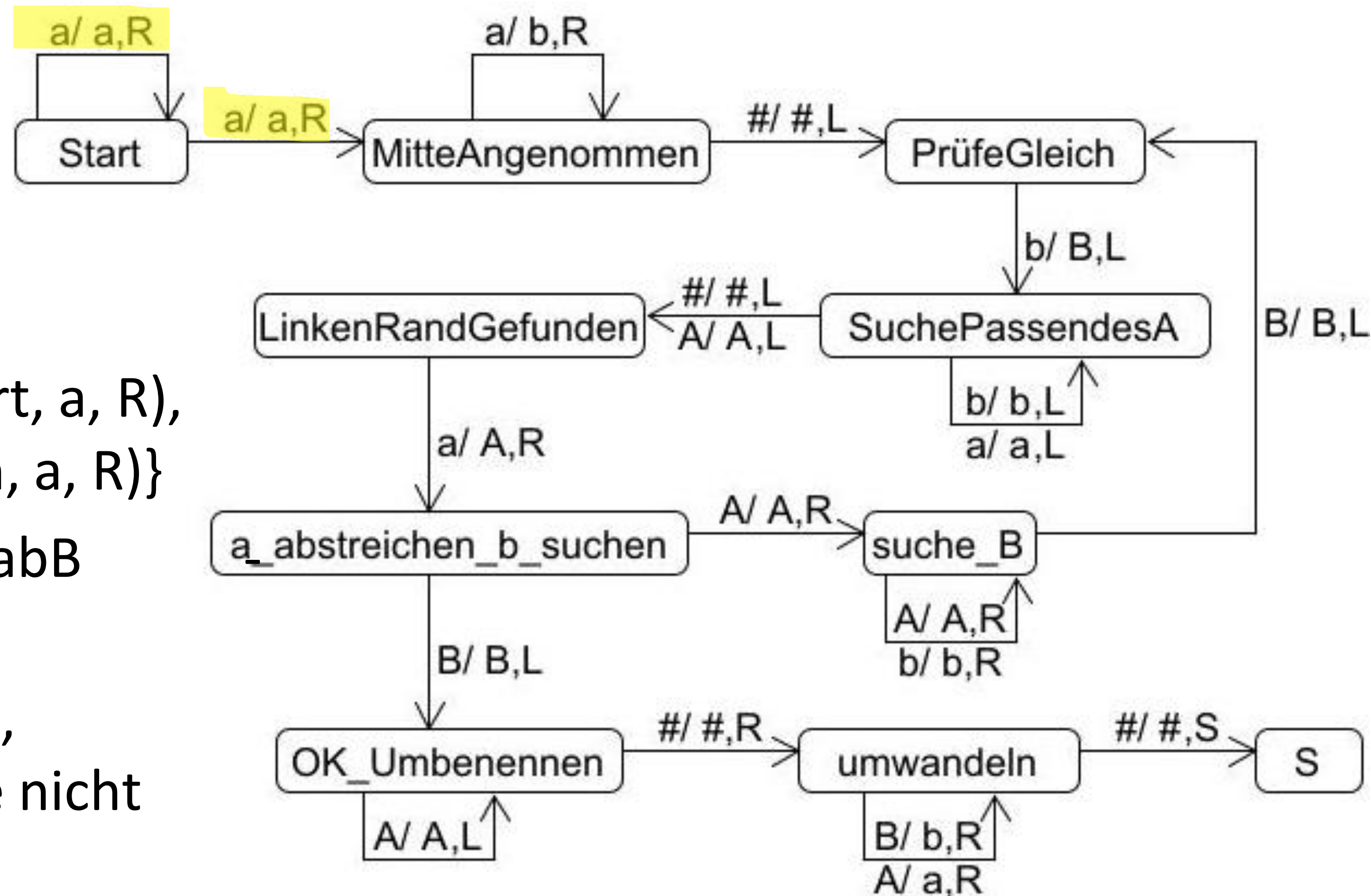
Definition: Eine nichtdeterministische *Turing-Maschine* ist ein Tupel (Zustände, Alphabet, *Überführungsfunktion*, Start) mit

- Zustände ist eine endliche Menge, mit halt $\in$ Zustände
- \# $\in$ Alphabet (Leerzeichen, blank, b)
- Start $\in$ Zustände
- Überführungsfunktion über:
  ~~Zustände × Alphabet -> Zustände × Alphabet × {LINKS, RECHTS, STOPP}~~
  Zustände × Alphabet
  -> Pot (Zustände × Alphabet × {LINKS, RECHTS, STOPP})
- Anschaulich: Abhängig vom Zustand und gelesenem Zeichen, kann es mehrere Varianten geben; für jede Konfiguration kann es eine Menge von Folgekonfigurationen geben; hält, wenn sie in einer der Folgekonfigurationen hält (es muss nur einen Weg geben)

Beispiel: Nichtdeterministische Turing-Maschine

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- berechne $f(a^{2n}) = a^n b^n$, n > 0; sonst undefiniert
- Nichtdeterministisch: Rate Mitte und prüfe dann Ergebnis



über(Start,a) = {(Start, a, R), (MitteAngenommen, a, R)}

aaaa ->* aabb ->* AabB >* AABB ->* aabb

wenn falsch geraten, terminiert Maschine nicht

Mächtigkeit nichtdeterministischer Turing-Maschinen

Satz: Mit deterministischen Turing-Maschinen können genau die gleichen Funktionen wie mit nichtdeterministischen berechnet werden

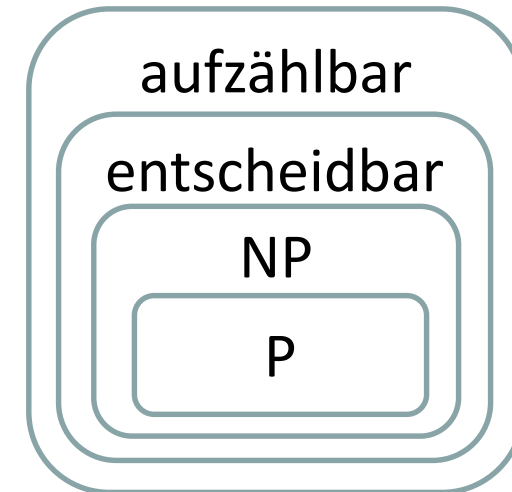
Beweisidee: kann vom Übergang von nichtdeterministischen zu deterministischen Automaten übernommen werden, gibt exponentiell viele Zustände

- bedeutet: egal ob mehrere Bänder, Köpfe oder Nichtdeterminismus, Berechenbarkeit bleibt gleich
- Achtung: der Übergang zum Exponentiellen hat Auswirkungen auf die Laufzeit, es kann z. B. direkt nicht garantiert werden, dass in P geblieben wird

NP

Definition: Eine Turing-berechenbare Funktion liegt in der Menge der nichtdeterministisch in polynomieller Laufzeit berechenbaren Funktionen NP, wenn es eine die Funktion berechnende nichtdeterministische Turing-Maschine und ein Polynom poly gibt, so dass die Laufzeitfunktion der Turing-Maschine in O(poly) liegt.

- Wichtig: Es muss nur eine (geschickt geschriebene) Turing-Maschine mit der gewünschten Laufzeitfunktion geben; in dieser kann geraten und muss das Ergebnis nur mit „bestem Raten“ in der Laufzeit gefunden werden
- $P \subseteq NP$, klar, deterministisch Spezialfall von nichtdeterministisch
- $NP \subseteq P$ unklar, Annahme Nein, aber unklar ob Beweis dafür überhaupt existiert bzw. existieren kann



P-NP-Vermutung

Polynomielle Reduktion

- nächstes Teilziel genauere Analyse von P und NP
- Erinnerung: Reduktion als Hilfsmittel bei Entscheidbarkeit

Definition: Gegeben seien Sprache1 und Sprache2 nicht notwendigerweise über den gleichen Alphabeten Alphabet1 und Alphabet2 sowie eine in polynomieller Zeit total berechenbare Funktion f: Alphabet1* -> Alphabet2*; gelte dann für alle w $\in$ Alphabet1*

w $\in$ Sprache1 genau dann wenn f(w) $\in$ Sprache2,

dann heißt Sprache1 polynomiell *reduzierbar* auf Sprache2 (geschrieben Sprache1 $\leq_f^{poly}$ Sprache2; f kann weggelassen werden)

Satz: Sei Sprache1 $\leq_f^{poly}$ Sprache2.

Wenn Sprache2 $\in$ NP, dann auch Sprache1 $\in$ NP.

Wenn Sprache2 $\in$ P, dann auch Sprache1 $\in$ P.

NP-hart und NP-vollständig

Definition: Eine Sprache heißt *NP-hart* (auch *NP-schwer*), wenn jede Sprache aus NP polynomiell reduzierbar auf diese Sprache ist, formaler

$$\forall \text{ Sprache'} \in \text{NP: Sprache'} \leq_f^{poly} \text{Sprache}$$

Definition : Eine Sprache heißt NP-vollständig , wenn sie in NP liegt (Sprache $\in$ NP) und NP-hart ist.

Folgerung: Kann für ein einziges NP-hartes Problem gezeigt werden, dass es auch in P liegt, dann gilt P = NP.

Wenn P $\neq$ NP gilt, dann kann kein NP-vollständiges Problem in polynomieller Zeit gelöst werden.

Wie wird NP-Vollständigkeit gezeigt

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

Um zu zeigen, dass eine Sprache NP-vollständig ist:

1. zeige, dass sie in NP liegt, es also einen nichtdeterministischen Algorithmus mit polynomieller Laufzeit gibt, der Sprachzugehörigkeit entscheidet
2. nutze eine bekannte NP-vollständige Sprache Spr
3. reduziere die Spr auf Sprache: Spr $\leq_f^{poly}$ Sprache

Erinnerung: Schritt 2 bedeutet:

$\forall$ Sprache‘ $\in$ NP: Sprache‘ $\leq_f^{poly}$ Spr ; da 3. Spr $\leq_f^{poly}$ Sprache

und Sprache‘ $\leq_f^{poly}$ Spr $\leq_f^{poly}$ Sprache gilt

$\forall$ Sprache‘ $\in$ NP: Sprache‘ $\leq_f^{poly}$ Sprache (also Sprache NP-vollständig)

SAT

Satz von Cook und Levin: SAT ist NP-vollständig.

SAT = { w | w ist eine erfüllbare aussagenlogische Formel}

(Alphabet [relativ egal] ob {0,1} oder „unsere Zeichen“)

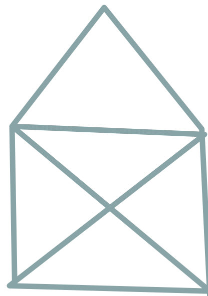
Beweisidee: (a) Zeige, dass SAT in NP. Für aussagenlogische Formeln existiert eine kontextfreie Grammatik, so dass für ein Wort in polynomieller Laufzeit entschieden werden kann, ob es eine syntaktisch korrekte Formel ist. Rate dann eine Belegung der Variablen und überprüfe (in polynomieller Zeit), ob Formel nach wahr ausgewertet werden kann.

(b) Zeige: $\forall$ Sprache‘$\in$ NP: Sprache‘ $\leq_f^{poly}$ SAT. Grober Ansatz: Es gibt nach Definition eine nichtdeterministische Turing-Maschine, die in polynomieller Laufzeit die Zugehörigkeit zu Sprache‘ entscheidet. Zeige dann, dass Turing-Maschinen als aussagenlogische Formel kodierbar sind.

Beispiel: Probleme in NP

- zentraler Ansatz immer SAT $\leq_f^{poly}$ neues Problem; also dass aussagenlogische in polynomieller Zeit in neues Problem wandelbar
- SAT $\leq_f^{poly}$ KNF-SAT   KNF-SAT = { w | w ist aussagenlogische Formel in konjunktiver Normalform}
- SAT $\leq_f^{poly}$ 3SAT   3SAT = { w | w $\in$ KNF-SAT, alle Teilterme haben 3 Variablen}

- ist ein Graph mit 3 Farben zu färben (so dass zwei benachbarte Knoten nicht die gleiche Farbe haben; 3SAT $\leq_f^{poly}$ 3Farben)
- hat ein Graph einen Hamilton-Zyklus (gibt es Weg entlang der Kanten des jeden Knoten genau einmal besucht und zum Ausgangsknoten zurückführt (Haus vom Nikolaus)



kleiner Ausblick

- für NP-vollständige Probleme, werden oft Heuristiken entwickelt; Algorithmen, die in „schneller“ Zeit zumindest gute Lösungen finden
- Beispiel: Graph, an jeder Kante stehen Kosten, suche Weg, der alle Knoten besucht mit minimalen Kosten (NP-vollständig)
- wieder Fragen für Sprachklassen stellbar: Abschluss gegen Komplement, Vereinigung, Differenz, Kleene-Stern, …
- viele weitere, praktisch relevante Fragen zum Verhältnis der Klassen formulierbar und teilweise nicht gelöst
- immer wieder neue Ergebnisse wie PRIME $\in$ P, PRIME = { w | w ist Primzahl}; dabei beachten, dass die Länge der Eingabe nicht w sondern log w ist [AKS04]

[AKS04] M.Agrawal, N. Kayal, N. Saxena, PRIMES is in P, Annals of Mathematics, Vol 160, Seiten 781–793, 2004, https://annals.math.princeton.edu/wp-content/uploads/annals-v160-n2-p12.pdf

Video

7. Auszug weiterer interessanter Theorie-Inhalte

zentrale Inhalte:

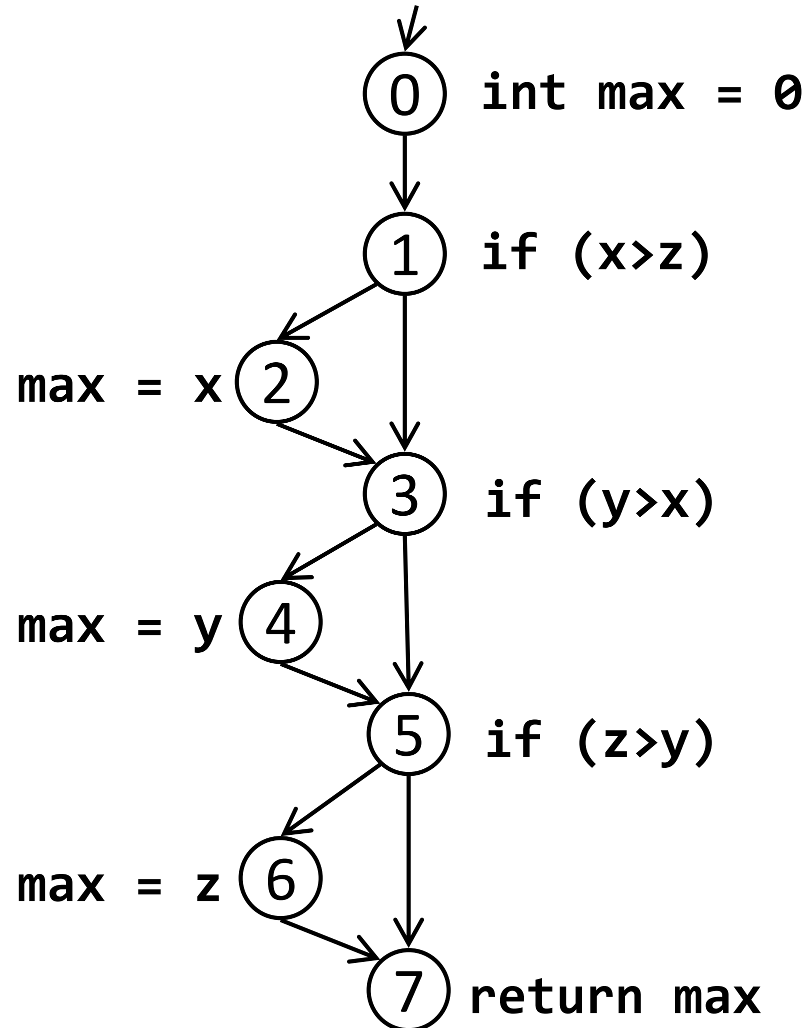
- Einblick Model-Checking
- Einblick Petri-Netze

Erinnerung

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Suche Maximum von drei ganzen Zahlen

```
public int max(int x,
               int y,
               int z) {
  int max = 0;
  if (x>z) {
    max = x;
  }
  if (y>x) {
    max = y;
  }
  if (z>y) {
    max = z;
  }
  return max;
}
```



Motivation von Model Checking

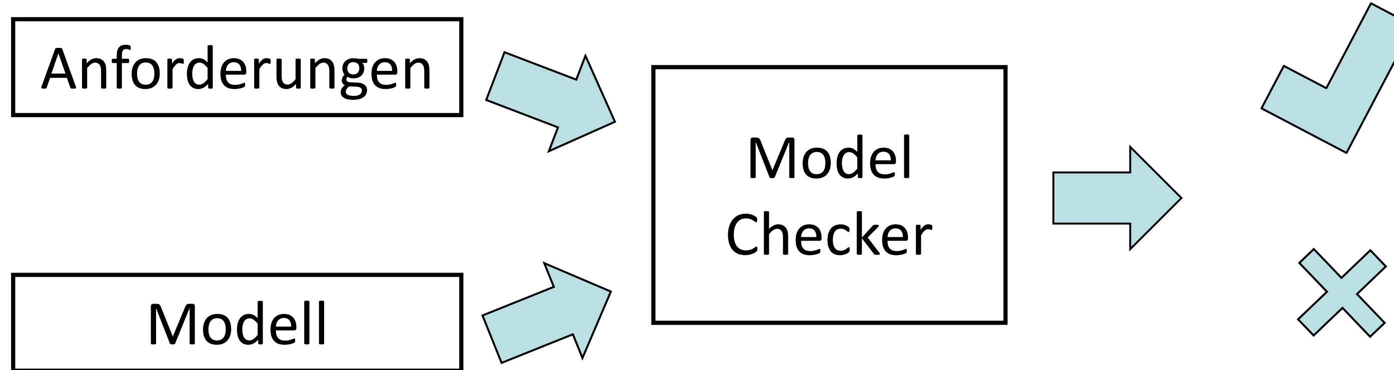
- klassische Testmethoden können keine Korrektheit garantieren
- Ansatz: Gegeben ist ein Modell (oder Spezifikation) und eine Anforderung, dann überprüft der Model-Checking-Algorithmus, ob das Modell die Anforderung erfüllt oder nicht. Falls das Modell die Anforderung nicht erfüllt, sollte der Algorithmus ein Gegenbeispiel liefern [hängt aber von Art der Anforderungsspezifikation ab].
- Beispiel Model Checker: SPIN entwickelt von Gerard Holzmann, zunächst als Simulator, dann Verifikationswerkzeug
- 2001 renommierten ACM Software System Award (z. B.: 1983 UNIX, 1987 Smalltalk, 1991 TCP/IP, 1995 World-Wide Web, 2002 Java)
- www.spinroot.com (frei verfügbar, seit 1991)
- Ansatz: Berechne alle erreichbaren Zustände und analysiere sie

[Hol03] G. J. Holzmann, The Spin Model Checker, Addison-Wesley, Boston, 2003

[Kle09] S. Kleuker, Formale Modelle der Softwareentwicklung, Vieweg+Teubner, Wiesbaden, 2009

Model Checking

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES



- Anforderungen
  - implizit: terminiert immer, kein Deadlock, …
  - explizit: Zusicherungen, formale Logik, …
- Modell: Modellierungssprache mit formaler Semantik, bietet Sprachkonstrukte an (Nichtdeterminismus, Zeit, Wahrscheinlichkeit, …)
- Model Checker: Prüft, ob Modell Anforderungen erfüllt, liefert (wenn sinnvoll) Gegenbeispiel

Verifikation deterministischer Programme

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Sortierverfahren, informelle Spezifikation:

Laufe mit dem Zähler i von 0 bis zur Arraygröße-1

Laufe mit dem Zähler j von i+1 bis zur Arraygröße

falls das j-te Element kleiner als das i-te Element ist, vertausche diese

(Ist Min-Sort, nach i-tem Durchlauf steht i-t-kleinster Wert an Position i)

Umsetzung in Java (1/3)


```
 6 public class Sortierer {
 7   public final static int N = 5;
 8   public void sortieren(byte[] array) {
 9     byte i = 0;
10     byte j;
11     byte tmp;
12
13     while (i < N - 1) {
14       j = (byte) (i + 1);
15       while( j < N - 1) {
16         if (array[i] > array[j]) {
17           tmp = array[i];
18           array[i] = array[j];
19           array[j] = tmp;
20         }
21         j++;
22       }
23       i++;
24     }
25   }
```

Umsetzung in Java (2/3)

```
public class SortiererTest {

  @Test
  void testVorherSortiert() {
    byte[] vorher = {-2, -2, -1, 0, 1};
    byte[] nachher = Arrays.copyOf(vorher, Sortierer.N);
    new Sortierer().sortieren(nachher);
    Assertions.assertTrue(istSortiert(vorher, nachher));
  }

  @Test
  void testSortiertFlexibel() {
    byte[] vorher = {0, -1, 1, 0, 2};
    byte[] nachher = Arrays.copyOf(vorher, Sortierer.N);
    new Sortierer().sortieren(nachher);
    Assertions.assertTrue(istSortiert(vorher, nachher));
  }
}
```

Umsetzung in Java (3/3)

```
  private int anzahlVonIn(byte wert, byte[] array) {
    int ergebnis = 0;
    for(int i = 0; i < Sortierer.N; i++) {
      if (array[i] == wert) { ergebnis++; }
    }
    return ergebnis;
  }

  private boolean istSortiert(byte[] vorher, byte[] nachher) {
    for(int i = 0; i + 1 < Sortierer.N; i++) {
      if (nachher[i] > nachher[i + 1]) { return false; }
    }
    for(int i = 0; i < Sortierer.N; i++) {
      if (anzahlVonIn(vorher[i], vorher) !=
          anzahlVonIn(vorher[i], nachher)) { return false; }
    }
    return true;
  }
}
```

Auszug aus Syntax von Promela

Video

- Spezifikation von Prozessen mit: **`proctype Initialize(){`**
- Start mit **`run Initialize();`**
- Nichtdeterminismus:

```
if
:: x < 10 -> x = x + 2
:: x < 15 -> x = x + 1
:: else -> skip
fi;
```

```
do
:: x < 10 -> x = x + 2
:: x < 15 -> x = x + 1
:: else -> skip
do;
```

- Es werden alle ausführbaren Alternativen (stehen nach `::`) bestimmt, dann nicht-deterministisch eine ausgewählt (Guarded-Command-Language; Dijkstra)
- Boolesche Bedingungen sind ausführbar, genau dann, wenn sie wahr sind, **`x>4; x<3; x==5;`** kann Spezifikationsteil sein (while(x <= 4) {waitALittleBit();})

Sortierverfahren (1/5)


```
#define N 5
#define MAX 3
byte array[N];
byte vorher[N];
bool initialized = false;
bool sorted = false;

init{
  run Initialize();
  run Sort();
  run Proof()
}
```

mögliche Ergebnisse für array:
0 0 0 0 0
0 0 0 0 1
…
3 3 3 3 3 [1024 Möglichkeiten]

```
proctype Initialize(){
  byte count = 0;
  byte rnd = 0;
  do
  :: count < N ->
       rnd = 0;
       do
       :: rnd < MAX -> rnd = rnd + 1
       :: true ->
            array[count] = rnd;
            vorher[count] =array[count];
            break
       od;
       count = count + 1;
  :: else -> break
  od;
  initialized = true
}
```

Sortierverfahren (2/5)

```
proctype Sort(){
  byte i = 0;
  byte j;
  byte tmp;
  initialized;
  do
  :: i < N - 1 ->
       j = i + 1;
       do
       :: j < N - 1 ->
          /*< muss <= sein */
          if
          :: array[i] > array[j] ->
               tmp = array[i];
               array[i] = array[j];
               array[j] = tmp
          :: else ->
               skip
          fi;
          j = j + 1
          :: else ->
             break
       od;
       i = i + 1;
  :: else ->
       break
  od;
  sorted = true
}
```

Sortierverfahren (3/5)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
proctype Proof(){
  byte count = 0;
  byte count2 = 0;
  byte anzahl1 = 0;
  byte anzahl2 = 0;
  sorted;
  /* pruefe ob array sortiert ist */
  do
  :: count < N - 1 ->
       assert(array[count] <= array[count + 1]);
       count = count + 1
  :: else ->
       break
  od;
```

Sortierverfahren (4/5)

```
  count = 0; /* pruefe auf gleiche Elemente */
  do
  :: count<=MAX ->
       anzahl1 = 0;    anzahl2 = 0;     count2 = 0;
       do
       :: count2 < N ->
            if
            :: vorher[count2] == count -> anzahl1 = anzahl1 + 1
            :: else ->
                 skip
            fi;
            if
            :: array[count2] == count -> anzahl2 = anzahl2 + 1
            :: else -> skip
            fi;
            count2 = count2 + 1
       :: else -> break
       od;
       assert(anzahl1 == anzahl2);
       count = count + 1
  :: else -> break
  od
}
```

Sortierverfahren (5/5)

- Verifikation (prüft alle möglichen Durchläufe von Initialize) und meldet Fehler mit Fehlerpfad
- Ausgabe der Simulation im Fehlerfall

```
array[0]   =  3
array[1]   =  3
array[2]   =  3
array[3]   =  3
array[4]   =  2
initialized   =  1
vorher[0]   =  3
vorher[1]   =  3
vorher[2]   =  3
vorher[3]   =  3
vorher[4]   =  2
sorted   =  1
```

Safety

- (•) safety
  - [x] + invalid endstates (deadlock)
  - [x] + assertion violations

Korrigierter Sortierer (j <= N-1)

```
Full statespace search for:
        never claim             - (not selected)
        assertion violations    +
        cycle checks            - (disabled by -DSAFETY)
        invalid end states      +

State-vector 48 byte, depth reached 240, errors: 0
   199108 states, stored
        0 states, matched
   199108 transitions (= stored+matched)
        0 atomic steps
hash conflicts:         1 (resolved)

unreached in init
        (0 of 4 states)
unreached in proctype Initialize
        (0 of 19 states)
unreached in proctype Sort
        (0 of 26 states)
unreached in proctype Proof
        (0 of 41 states)
No errors found -- did you verify all claims?
```

Funktionsweise von Petrinetzen

- Motivation für Petrinetze
- Schaltverhalten
- typische Netzeigenschaften
- Fairness

[Pet62] C. A. Petri: Kommunikation mit Automaten, Institut für Instrumentelle Mathematik, Bonn, Schriften des IMM Nr.2 (Dissertation), 1962

[Rei86] W. Reisig, Petrinetze, 2. Auflage, Springer, Berlin Heidelberg New York Tokio, 1986

[Kle09] S. Kleuker, Formale Modelle der Softwareentwicklung, Vieweg+Teubner, Wiesbaden, 2009

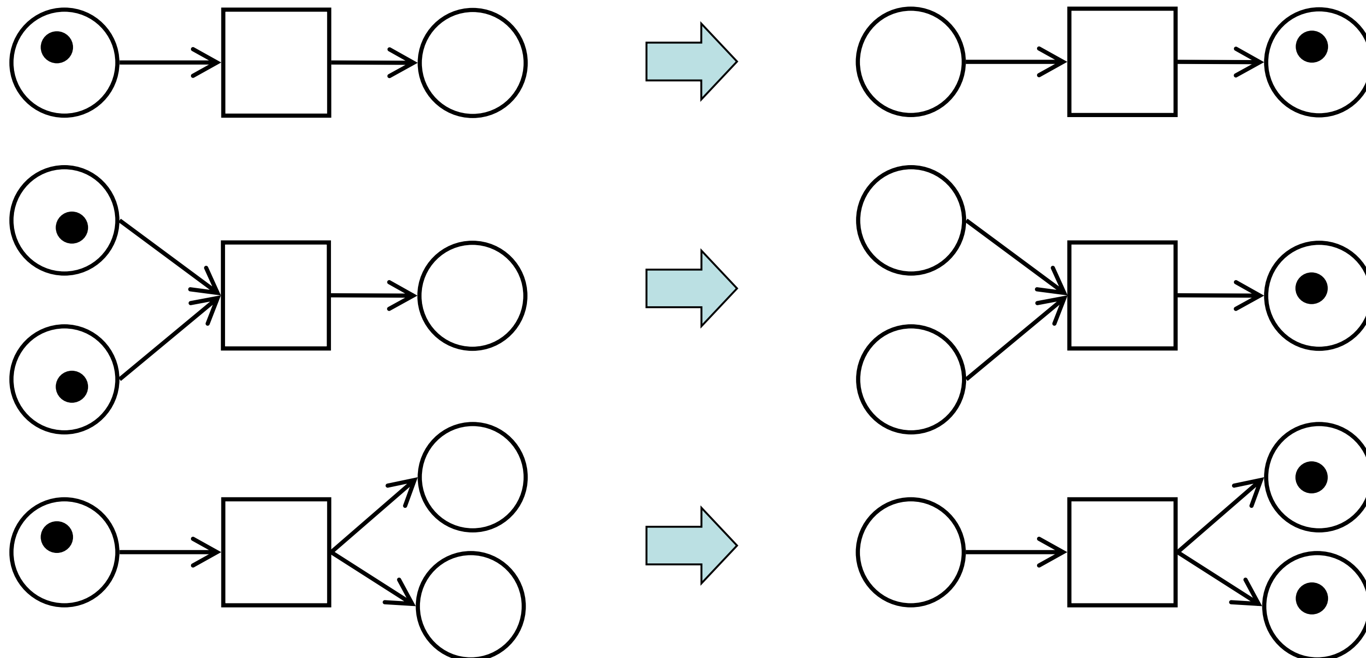
Motivation für Petrinetze

- (weitere) Spezifikationsmöglichkeit für verteilte Systeme
- Modelle sollen zur Verifikation möglichst einfach sein
- Generell gibt es zwei Arten von Informationen in Modellen
- Daten, die bearbeitet werden
- Aktionen (Transitionen), die Daten verarbeiten; aus Daten neue Daten berechnen

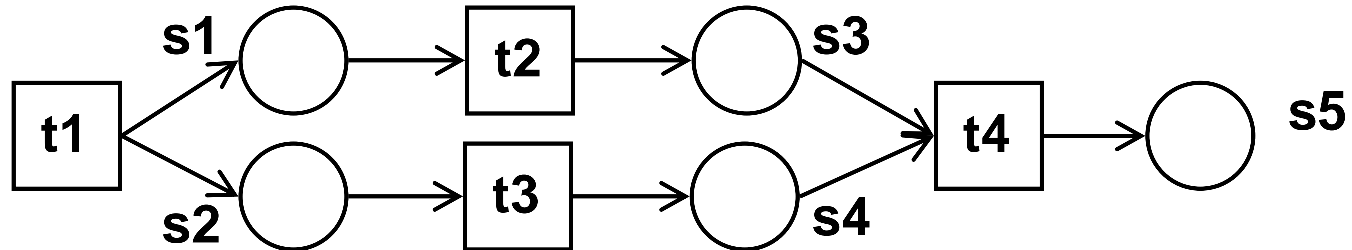
- Petrinetz-Ansatz nach C. A. Petri (1962)
- Stellen, die Daten aufnehmen können
- Daten als Token, die auf Stellen liegen
- Transitionen nehmen Token aus Stellen und legen neue Token auf evtl. andere Stellen

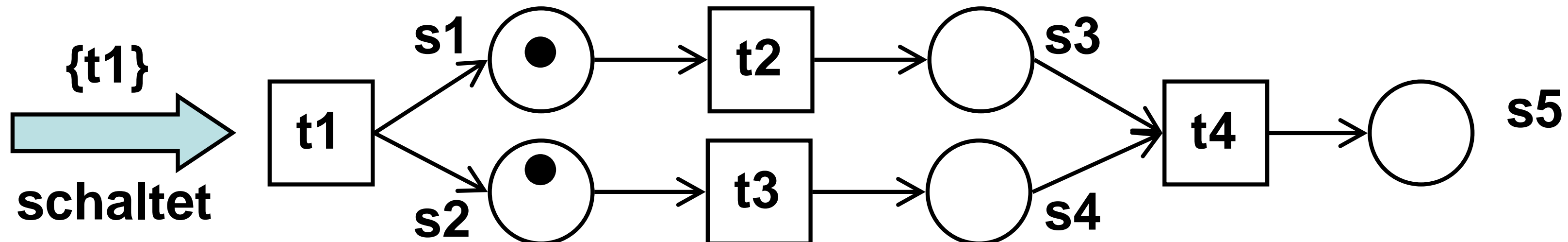
Schaltregel von Petrinetzen

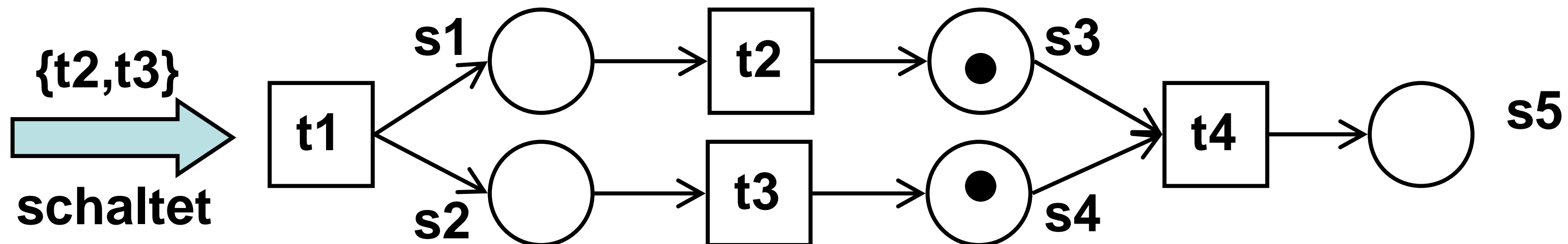
- Eine Transition (Kasten oder Strich) kann schalten, wenn auf jeder eingehenden Kante einer Stelle (Kreis) mindestens ein Token (gefüllter Punkt) liegt
- Beim Schalten wird ein Token von jeder Stelle einer eingehenden Kante weggenommen und ein Token auf jede Stelle einer ausgehenden Kante gelegt

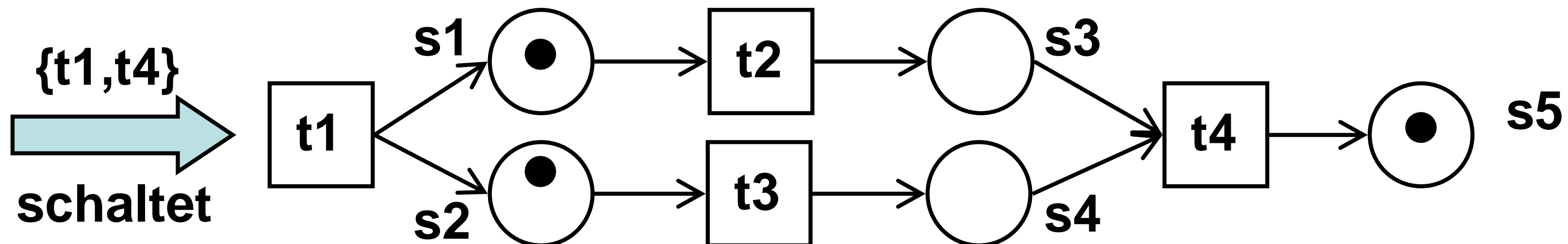


Beispiel für Schaltmöglichkeiten









Definition S/T-Netz (1/2)

- Definition (Petri-Netz): Ein Petri-Netz P =(S,T,G) besteht aus einer Menge von Stellen S, einer Menge von Transitionen T und einem gerichteten Verbindungsgraphen G, bei dem nur Stellen mit Transitionen und Transitionen mit Stellen verbunden sind,
  $G \subseteq (S \times T) \cup (T \times S)$.
- Definition (Vorbereich, Nachbereich): Sei $t \in T$ eine Transition eines Petri-Netzes, dann heißt
  $pre(t) = \{s \in S \mid (s,t) \in G\}$ Vorbereich von t
  $post(t) = \{s \in S \mid (t,s) \in G\}$ Nachbereich von t
- Definition (Markierung): Eine Markierung M eines Petrinetzes P=(S,T,G) ist eine Abbildung, die jeder Stelle des Netzes eine Anzahl von Token zuordnet, M: $S \rightarrow$ *NatürlicheZahlen*
- Definition (Schalten eines Netzes): Sei M eine Markierung eines Petri-Netzes P=(S,T,G), dann heißt eine Transition t aktiviert unter M, bzw. kann t schalten unter M, wenn auf allen Stellen des Vorbereichs von t mindestens ein Token liegt, d.h. für alle $s \in pre(t)$: $M(s) \geq 1$

Definition S/T-Netz (2/2)

- Eine Transitionsmenge $T' \subseteq T$ ist zusammen aktiviert, bzw. kann zusammen schalten, wenn genügend Marken auch in den gemeinsamen Vorbereichen liegen. Für alle s sei $s_{pre(T')}$ die Häufigkeit, mit der s in den Vorbereichen von T‘ vorkommt,

  $s_{pre(T')}$= anzahl($\{t \in T' \mid s \in pre(t)\}$), dann muss

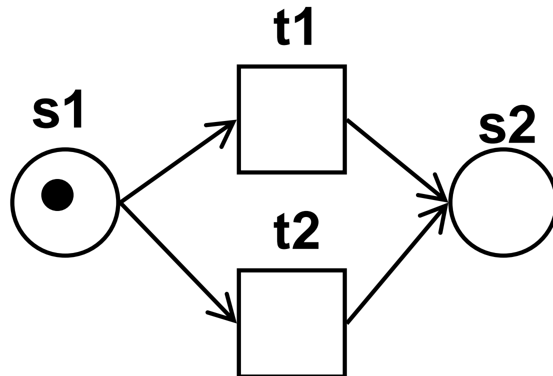
  für alle $s \in S$: $M(s) \geq s_{pre(T')}$

  Weiterhin sei $s_{post(T')}$= anzahl($\{t \in T' \mid s \in post(t)\}$).

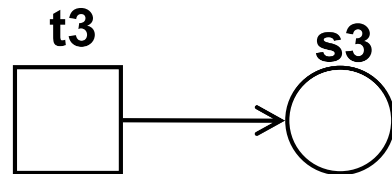
- Sei M eine Markierung und $T' \subseteq T$ aktiviert unter M, dann hat das Netz nach dem Schalten die Folgemarkierung M‘ (geschrieben: M[T‘>M‘), wobei für alle $s \in S$ gilt: $M'(s) = M(s)-s_{pre(T')}+ s_{post(T')}$. Für eine Transition $t \in T'$ heißt das:

$$M'(s) = \begin{cases} M(s), \text{ falls } s \notin pre(t) \cup post(t) \text{ oder } s \in pre(t) \cap post(t) \\ M(s)-1 \text{ falls } s \in pre(t) \text{ und } s \notin post(t) \\ M(s)+1 \text{ falls } s \in post(t) \text{ und } s \notin pre(t) \end{cases}$$

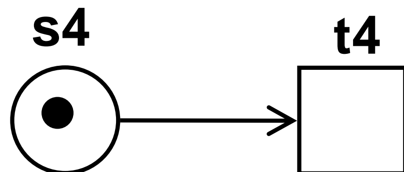
Analyse des Schaltverhaltens



t1 und t2 sind jeweils aktiviert, {t1,t2} ist nicht aktiviert, da nur ein Token auf s1



t3 immer aktiviert, kann beliebig viele Token erzeugen

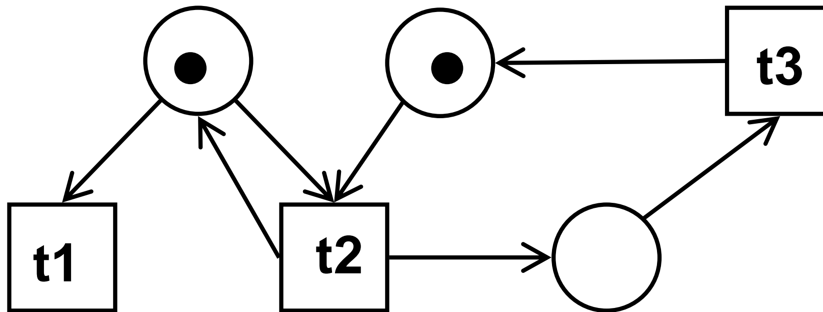


t4 aktiviert, kann beliebig viele Token (wenn vorhanden) von s4 löschen

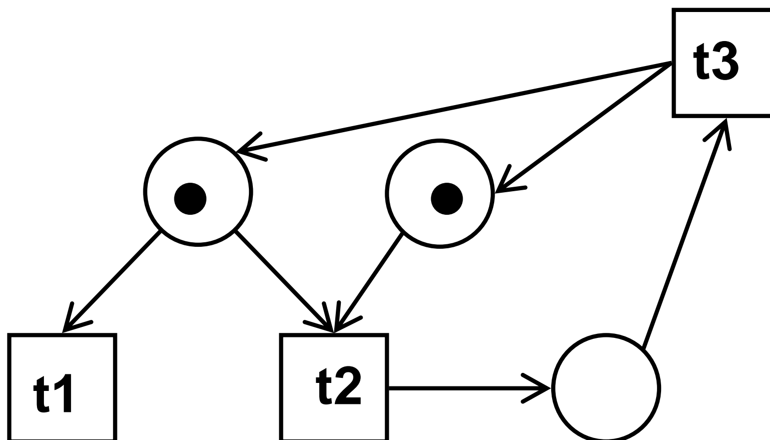
Petri-Netze als Prozesse

- Zu einem Petri-Netz wird immer eine Anfangsmarkierung M0 angegeben, dann können Fragen aus der Prozesswelt relevant werden:
- kann das Netz terminieren: wird eine Markierung erreicht, unter der keine Transition mehr aktiviert ist
- terminiert das Netz immer: da Netze nicht-deterministisch sind, können in verschiedenen Abläufen verschiedene Markierungen erreicht werden
- unerwünschte Terminierung = Deadlock
- wie bei Prozessen spielt bei Fragestellungen Fairness für einen Ausführungspfad M0[t1>M1[t2>M2[... eine Rolle
  - schwach fair: Transition, die unendlich oft aktiviert ist, schaltet auch
  - stark fair: Transition, die unendlich oft immer wieder aktiviert ist, schaltet auch

Fairness bei Netzen



unfair terminiert das Netz nicht, schwach fair terminiert es, da immer t1 aktiviert

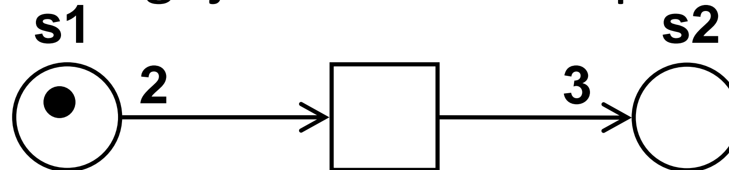


unfair und schwach fair terminiert das Netz nicht, stark fair terminiert es, da immer wieder t1 aktiviert

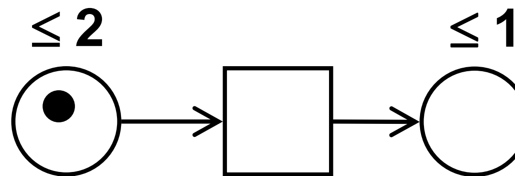
Spezielle Netze

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- In unserer Definition können beliebig viele Token auf einer Stelle liegen, weiterhin bewegt jede Transition pro Stelle nur ein Token



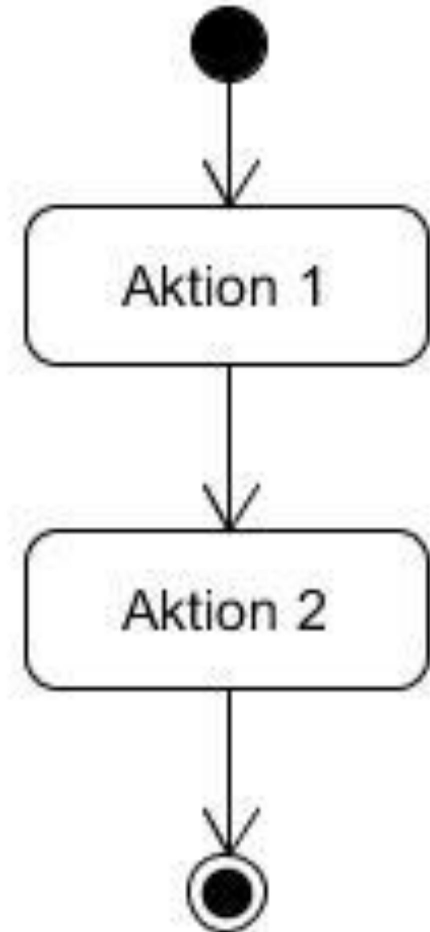
- Variante 1: Die Kanten des Netzes werden gewichtet, z,B. t1 zieht zwei Token von s1 ab (benötigt diese) und erzeugt drei Token auf s2



- Variante 2: Die Tokenanzahl pro Stelle wird begrenzt, es dürfen maximal max(s)-Token auf einer Stelle s liegen, d.h. zum Schalten muss sichergestellt sein, dass auf post(s) genügend Platz ist
- Variante 2.1: Es wird für alle Stellen s max(s)=1 gesetzt, entweder ein Token vorhanden oder nicht
- Variante 1 ist ausdrucksmächtig wie S/T-Netze
- Varianten 2 und 2.1 ausdrucksmächtig wie endliche Automaten

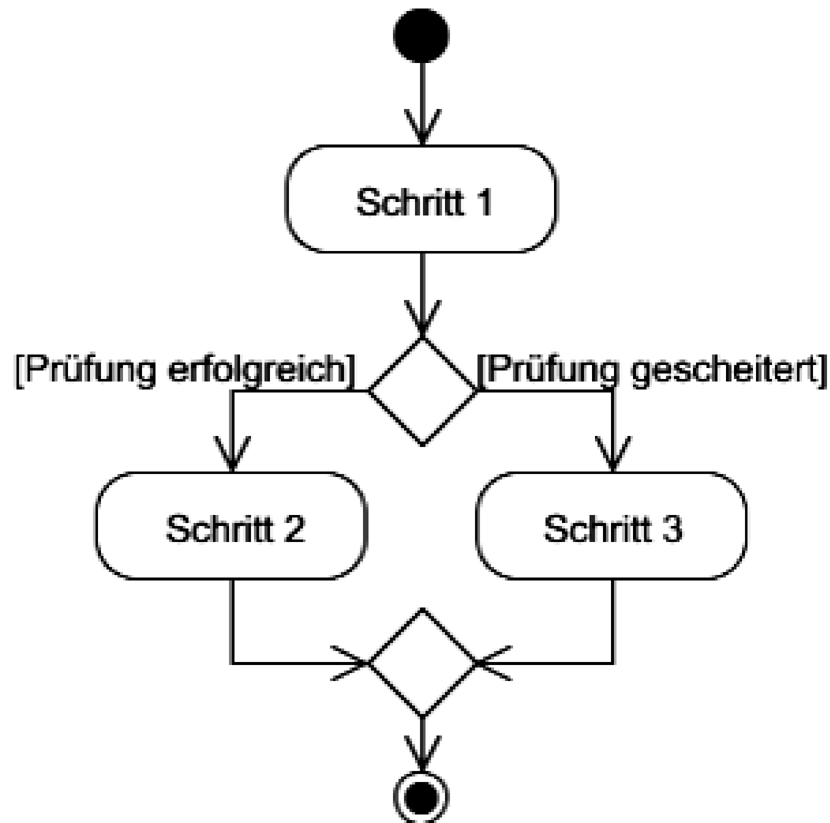
Beispielnutzung: Semantik von Aktivitätsdiagrammen

- Aktivitätsdiagramme sind zentrales Mittel der UML zur Beschreibung von Abläufen
- werden zur Modellierung von Abläufen in einer Software und zur Darstellung von Geschäftsprozessen genutzt
- haben seit der UML 2.0 eine formale Semantik basierend auf Petri-Netzen
- rechte Seite zeigt minimalen sequenziellen Ablauf



Prozessmodellierung mit Aktivitätsdiagrammen

Zur Beschreibung werden folgende elementare Elemente genutzt:



genau ein Startpunkt

einzelner Prozessschritt (Aktion)

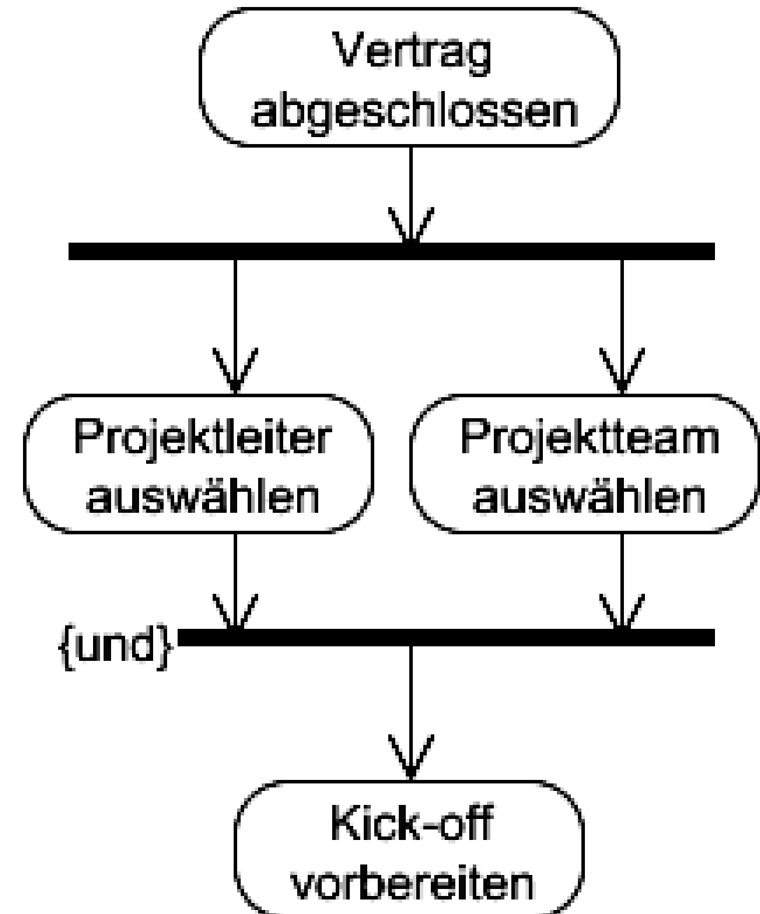
Kontrollknoten (Entscheidung)

Kontrollknoten (Zusammenführung)

Endpunkt (Terminierung)

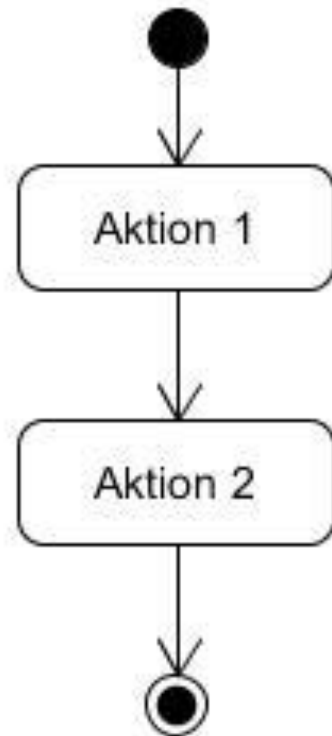
Parallelität in Prozessen

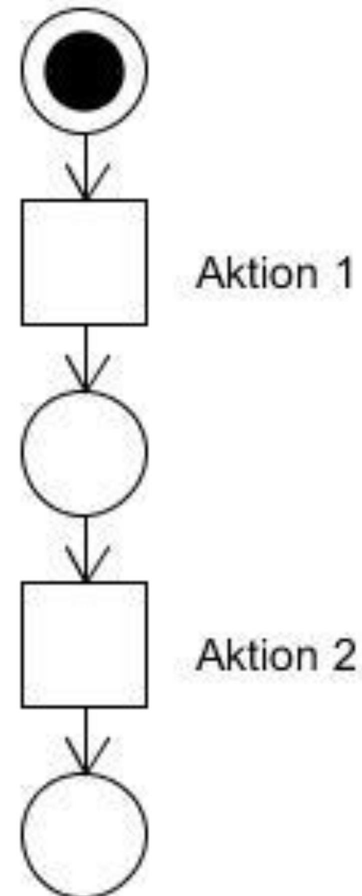
- Waagerechter oder senkrechter Strich steht für mögliche Prozessteilung (ein Pfeil rein, mehrere raus) oder Zusammenführung (mehrere Pfeile rein, ein Pfeil raus)
- Am zusammenführenden Strich steht Vereinigungsbedingung, z. B.
  - {und}: alle Aktionen abgeschlossen
  - {oder}: (mindestens) eine Aktion abgeschlossen
- UML 1.1 hatte andere Restriktionen



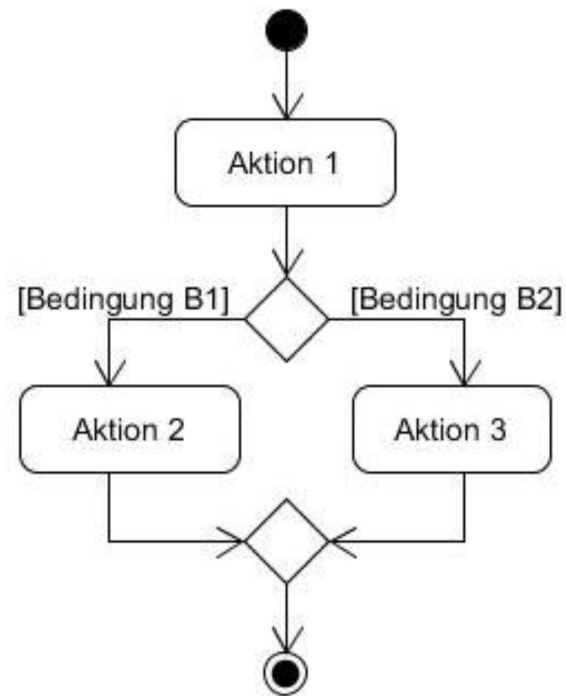
Umsetzung der Sprachelemente als Petri-Netz (1/3)

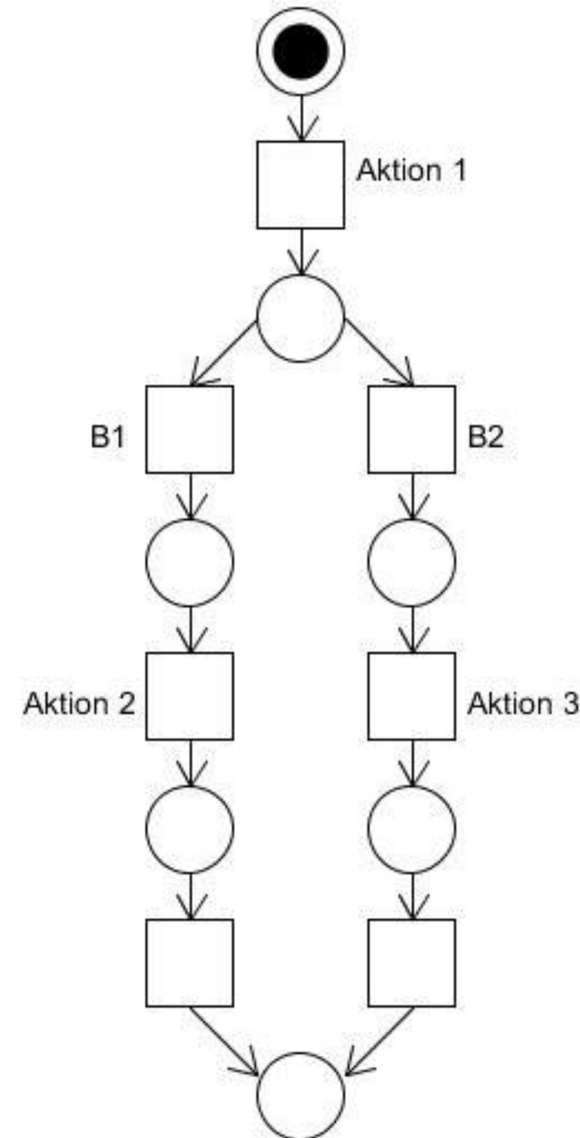
HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES



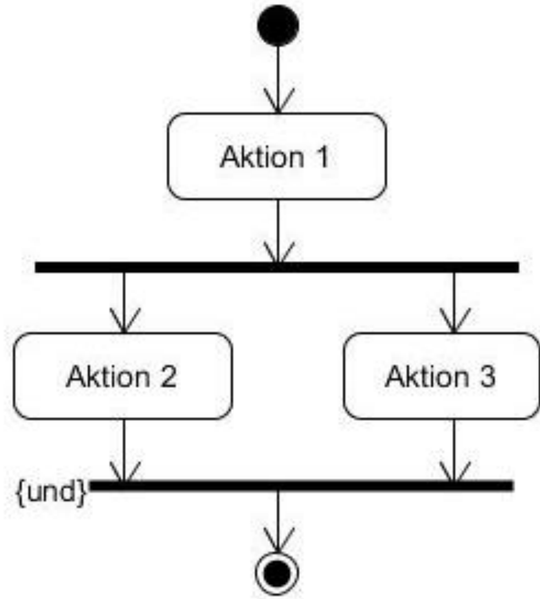


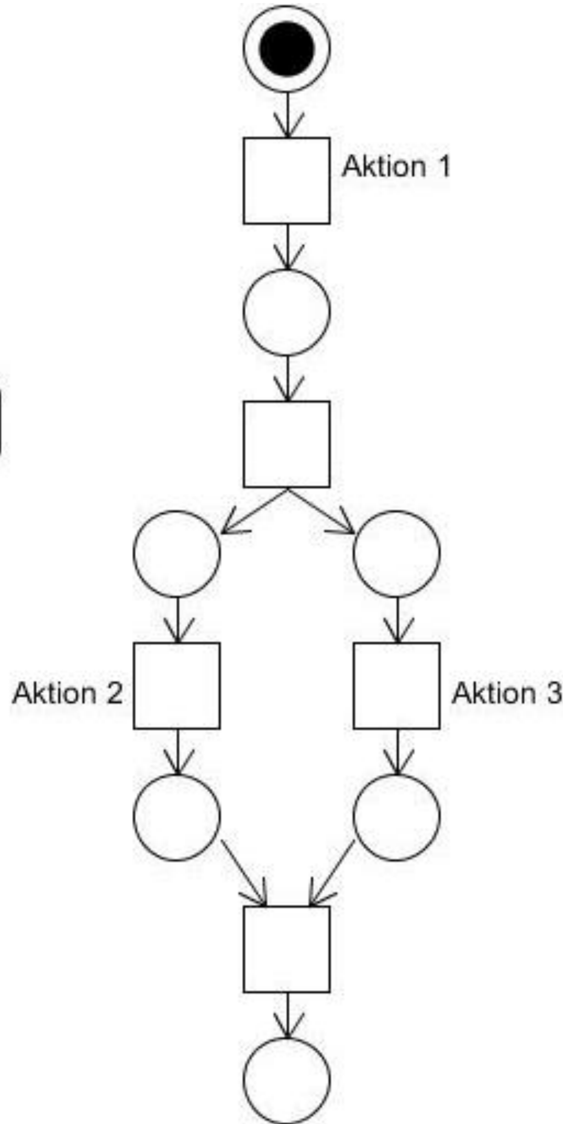
Umsetzung der Sprachelemente als Petri-Netz (2/3)

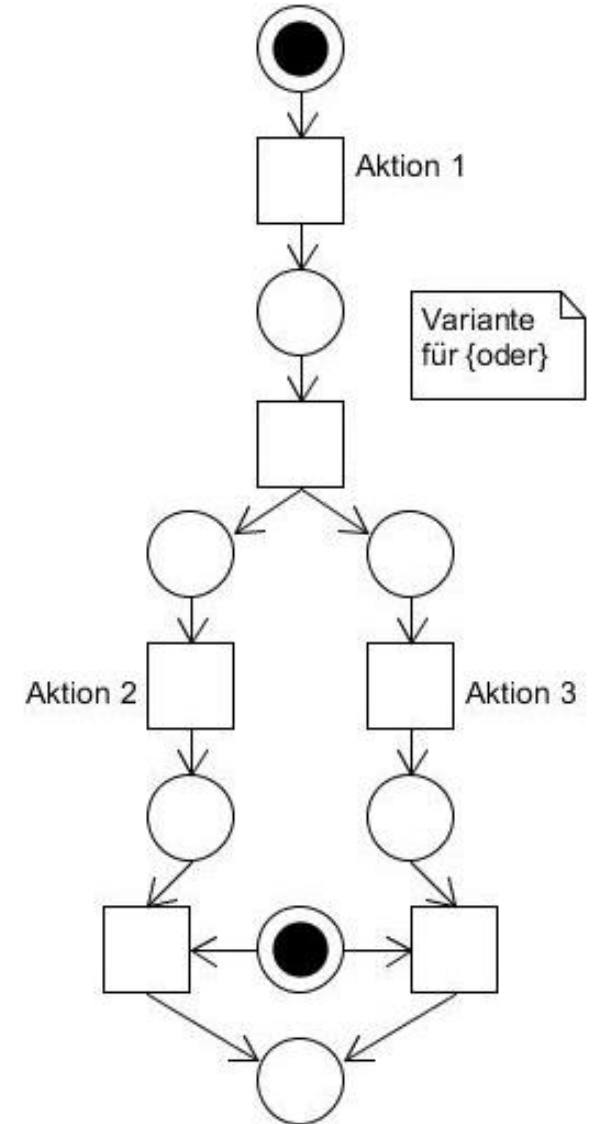




Umsetzung der Sprachelemente als Petri-Netz (3/3)

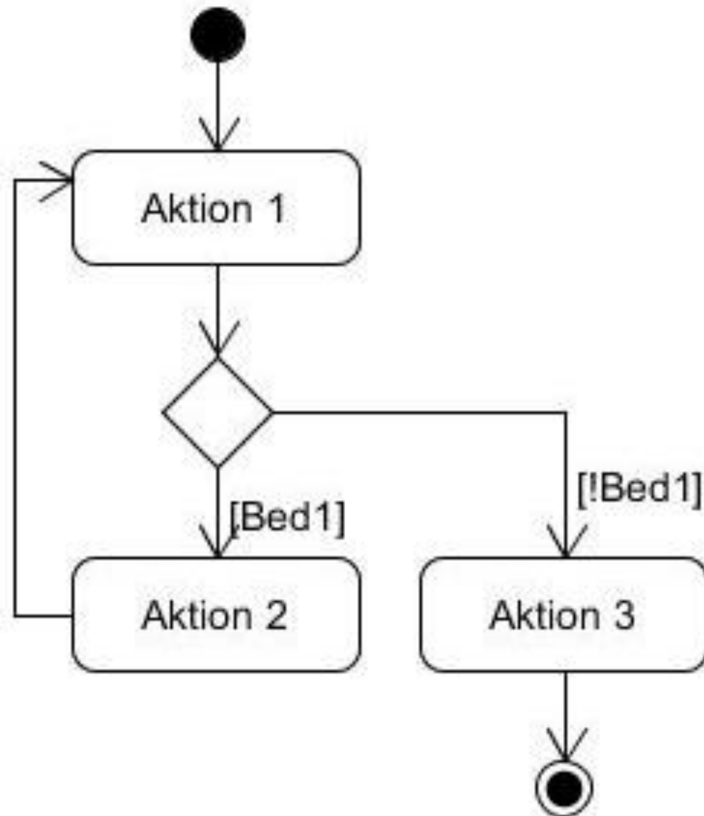


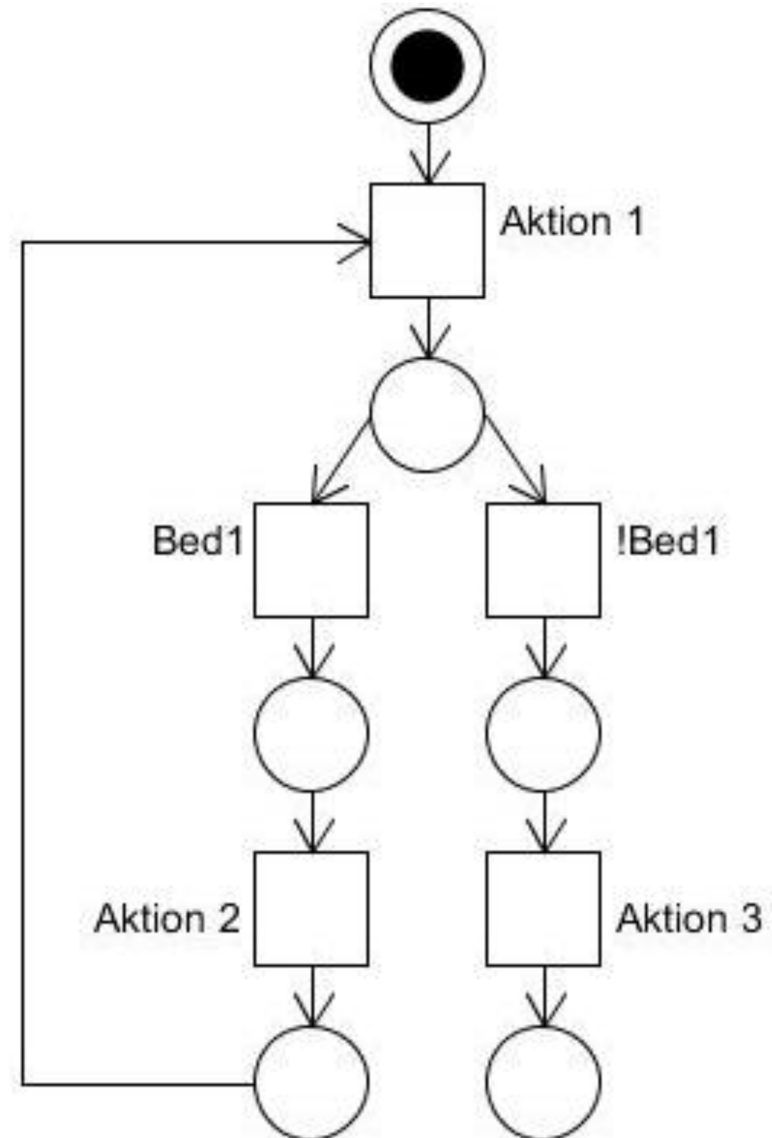




Hilfstoken müssen bei jedem neuen Durchlauf (auch Schleife) wieder hingelegt werden

Beispiel für Modellierungsfehler





Erreichbarkeits- und Überdeckungsgraphen

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Erreichbarkeit von Markierungen
- Erreichbarkeitsgraph
- Überdeckungsgraph

Erreichbare Markierungen

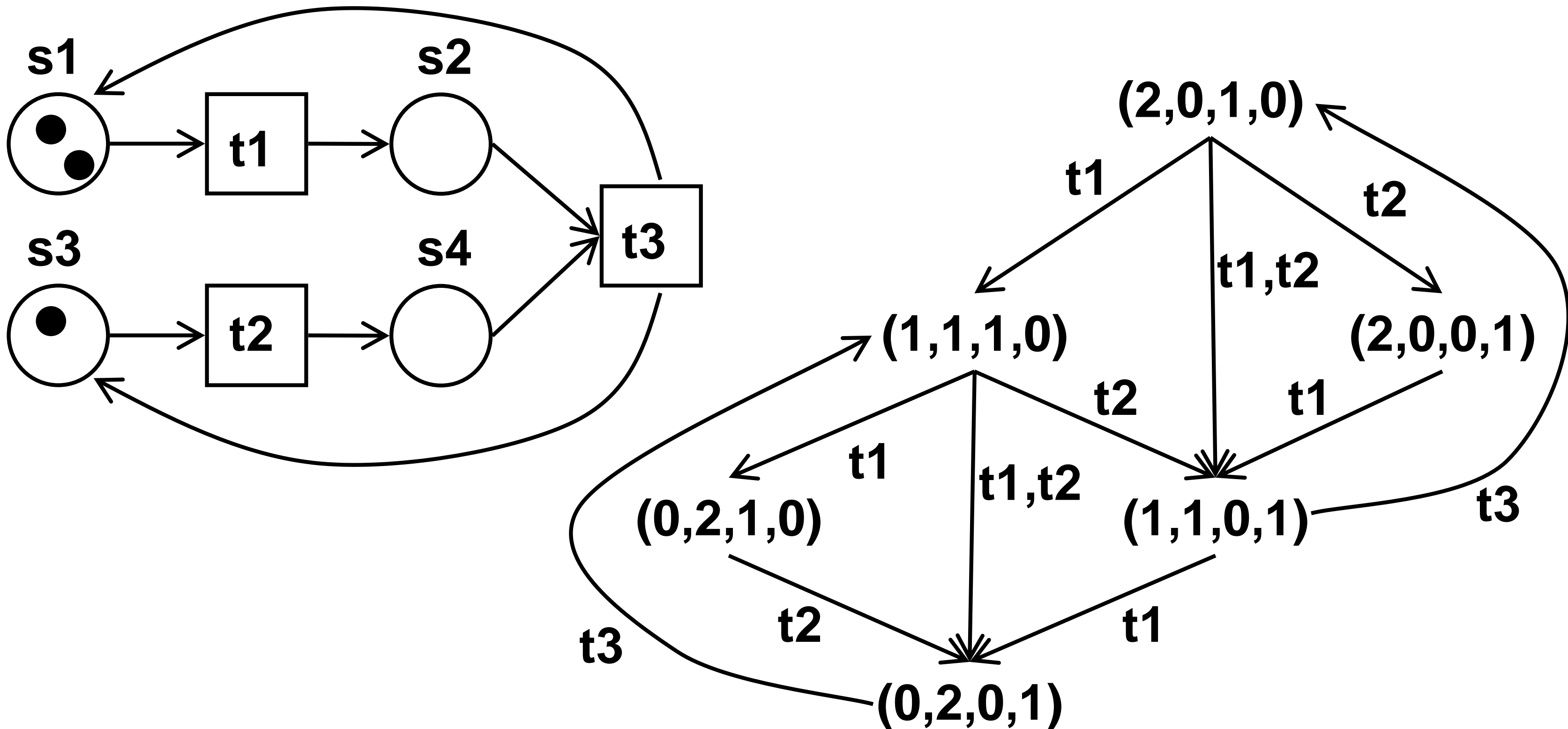
- Definition (Erreichbare Markierungen): Sei M eine Markierung eines Petri-Netzes P=(S,T,G), dann bezeichnet Erreichbar(P,M) die Menge aller von M aus erreichbaren Markierungen. Formal:

  Erreichbar(P,M) = { M' | es gibt ein i mit $0 \leq i \leq n$ und Transitionen ti$\in$T, sowie Markierungen Mi, so dass es eine Transitionsfolge M [t1> M1 [t2> M2 ... [tn> Mn=M' gibt}

- Abhängig vom Petrinetz kann Erreichbar(P,M) endlich oder unendlich sein.

Erreichbarkeitsgraph (auch: Fallgraph)

- Man kann alle erreichbaren Markierungen (Zustände) in einen Graphen eintragen und Kanten mit gefeuerten Transitionen beschriften:



- Problem: Erreichbarkeitsgraph nicht immer endlich (nur garantiert bei Stellenkapazitätsbeschränkung)

Formale Definition des Erreichbarkeitsgraphen

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Definition (Erreichbarkeitsgraph): Sei M eine Markierung eines Petri-Netzes P = (S,T,G), dann heißt ein Graph G =(Erreichbar(P,M), Tr), wobei Tr $\subseteq$ Erreichbar(P,M)$\times$Pot(T)$\times$Erreichbar(P,M) genau die Kanten (M,T',M') enthält, für die es eine Menge von Transitionen T'$\subseteq$T mit M [T'> M' gibt, Erreichbarkeitsgraph (oder auch Fallgraph) von P und M.

- Erinnerung: Pot(T) bezeichnet die Potenzmenge von T
- Ein Erreichbarkeitsgraph kann nur dann sinnvoll dargestellt werden, wenn Erreichbar(P,M) endlich ist.

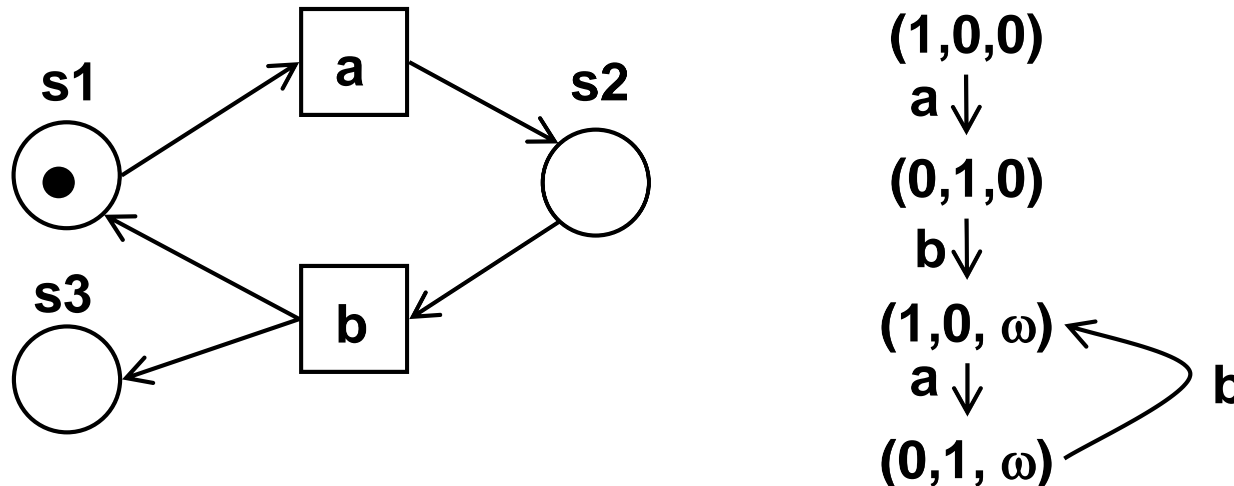
Ordnung auf Markierungen

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Möchte man einen endlichen Graphen zur Darstellung der Zustandsfolgen, ist die Idee, wenn die Werte einer erreichten Markierung M‘ echt größer als die Werte einer vorher erreichten Markierung M sind, dann kann die Tokenanzahl an den Stellen beliebig steigen, an denen die Tokenanzahl gestiegen ist.
- M‘ > M: für alle $s \in S$: $M'(s) \geq M(s)$ und es gibt ein $s \in S$: $M'(s) > M(s)$
- gilt M‘>M und für eine Stelle s damit M‘(s)>M(s), dann können auf s beliebig viele Token erzeugt werden.
- z. B. (0,4,1,1) > (0,3,0,1), d.h. (0,x,y,1) mit beliebig großen x und y erreichbar
- nicht jede x,y-Kombination erreichbar

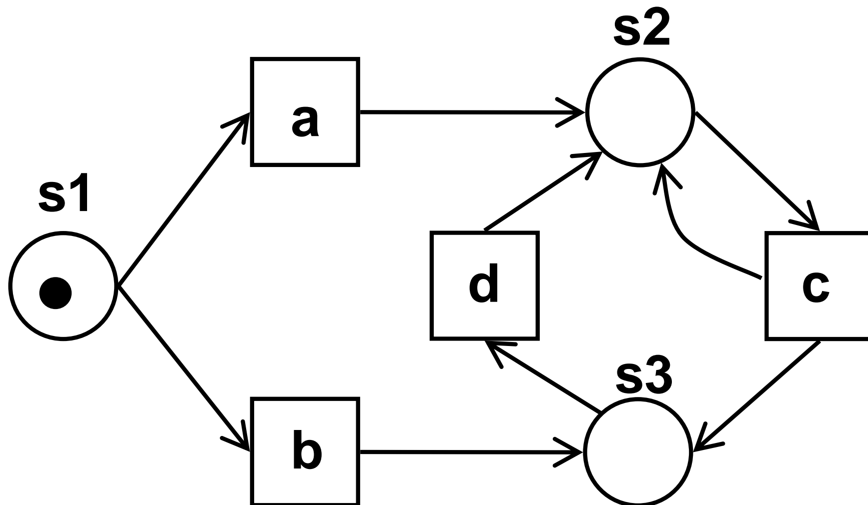
Überdeckungsgraph

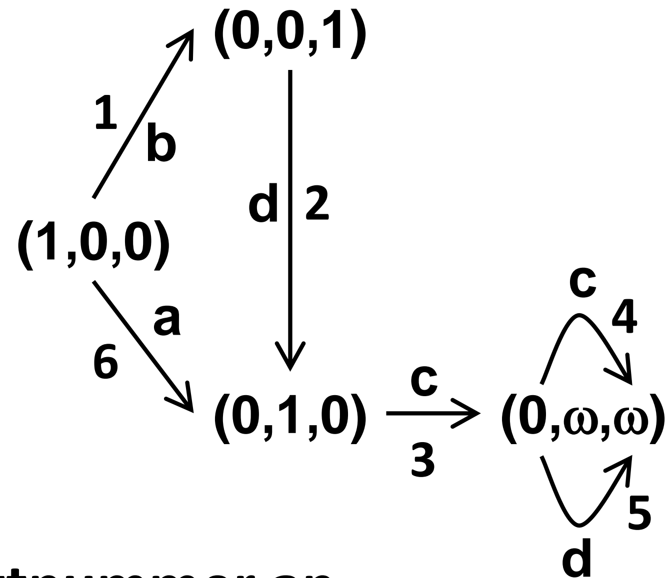
- Überdeckungsgraph wird wie Erreichbarkeitsgraph konstruiert, allerdings, wenn M‘ > M für vorher auf dem Pfad dahin berechnete Markierungen wird für alle Stellen s mit M‘(s)>M(s), wird statt M‘(s) der Wert ω für unendlich eingetragen (es werden alle bisher erreichten M betrachtet)



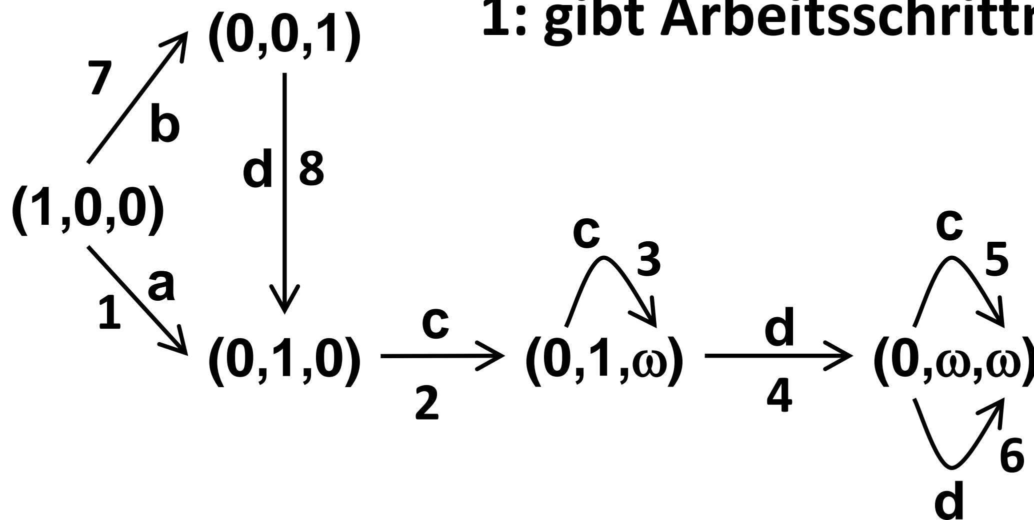
- Aus dem Überdeckungsgraphen lässt sich ablesen, welche Stellen beschränkt, bzw. unbeschränkt sind; bei unbeschränkten Stellen kann die Tokenanzahl beliebig wachsen

Überdeckungsgraph nicht eindeutig [Rei]





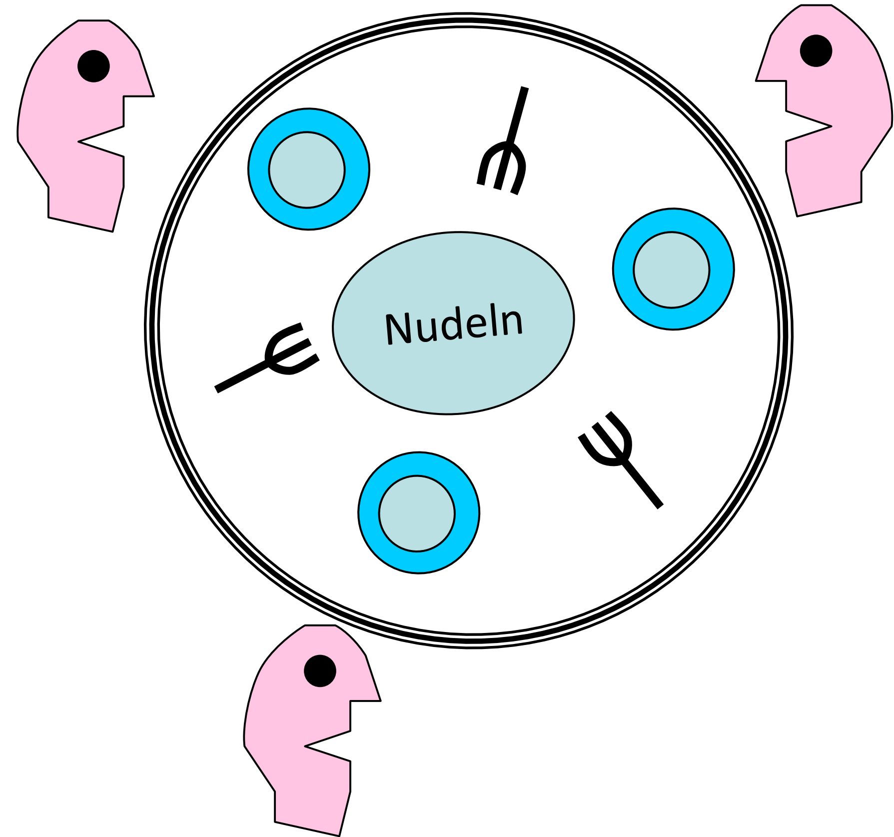
1: gibt Arbeitsschrittnummer an



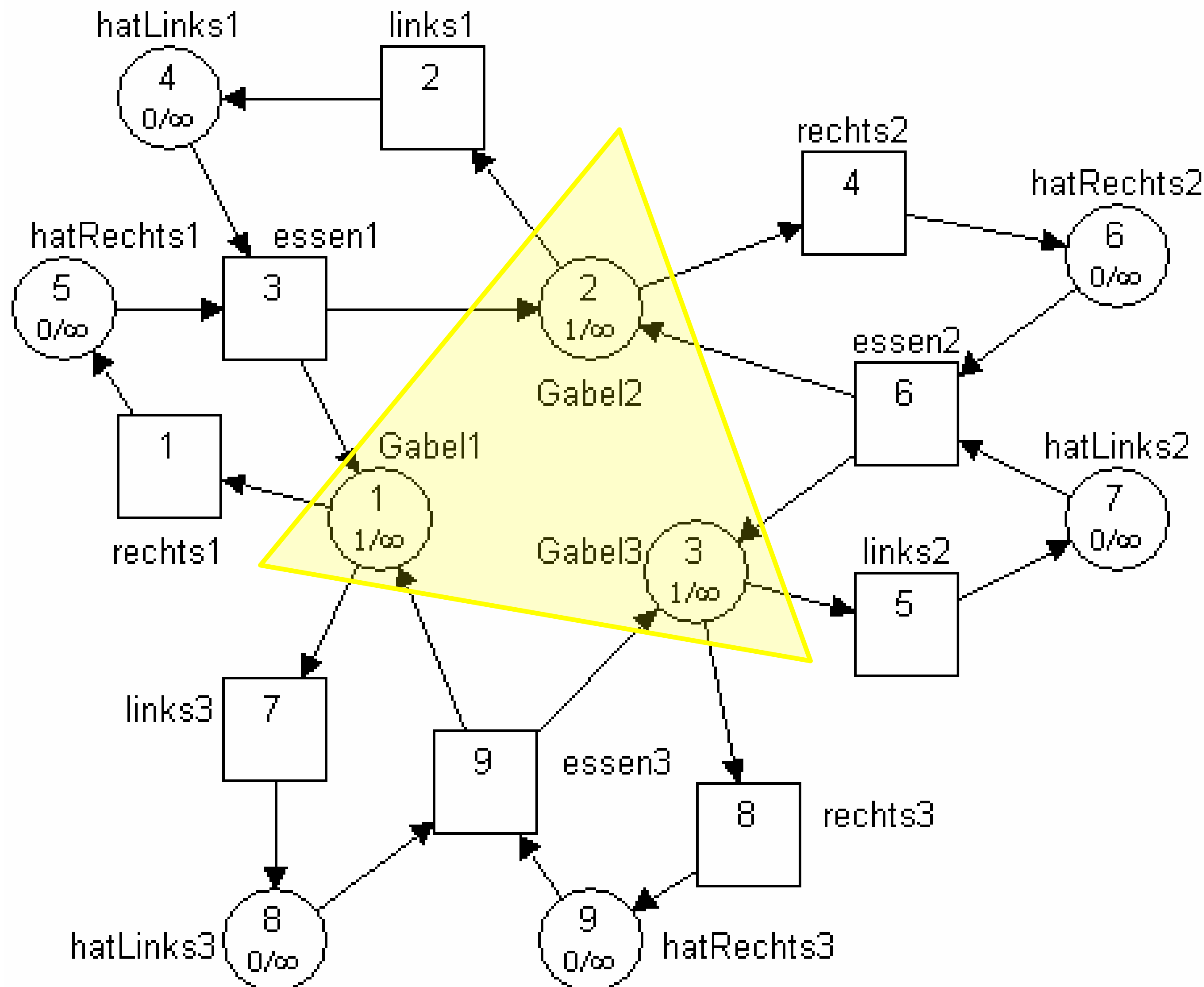
Anmerkung:
(0,ω,ω) muss nicht bedeuten, dass beliebige (0,x,y) erreicht werden können

Dinierende Philosophen (1/4)

Drei Philosophen haben sich zum Spaghetti-Essen getroffen. In der Mitte steht ein Topf mit Nudeln. Zwischen zwei Philsophen liegt jeweils eine Gabel. Wenn ein Philosoph Hunger hat, nimmt er die linke und die rechte Gabel, isst und legt die Gabeln wieder weg.



Dinierende Philosophen (2/4)



Dinierende Philosophen (3/4)

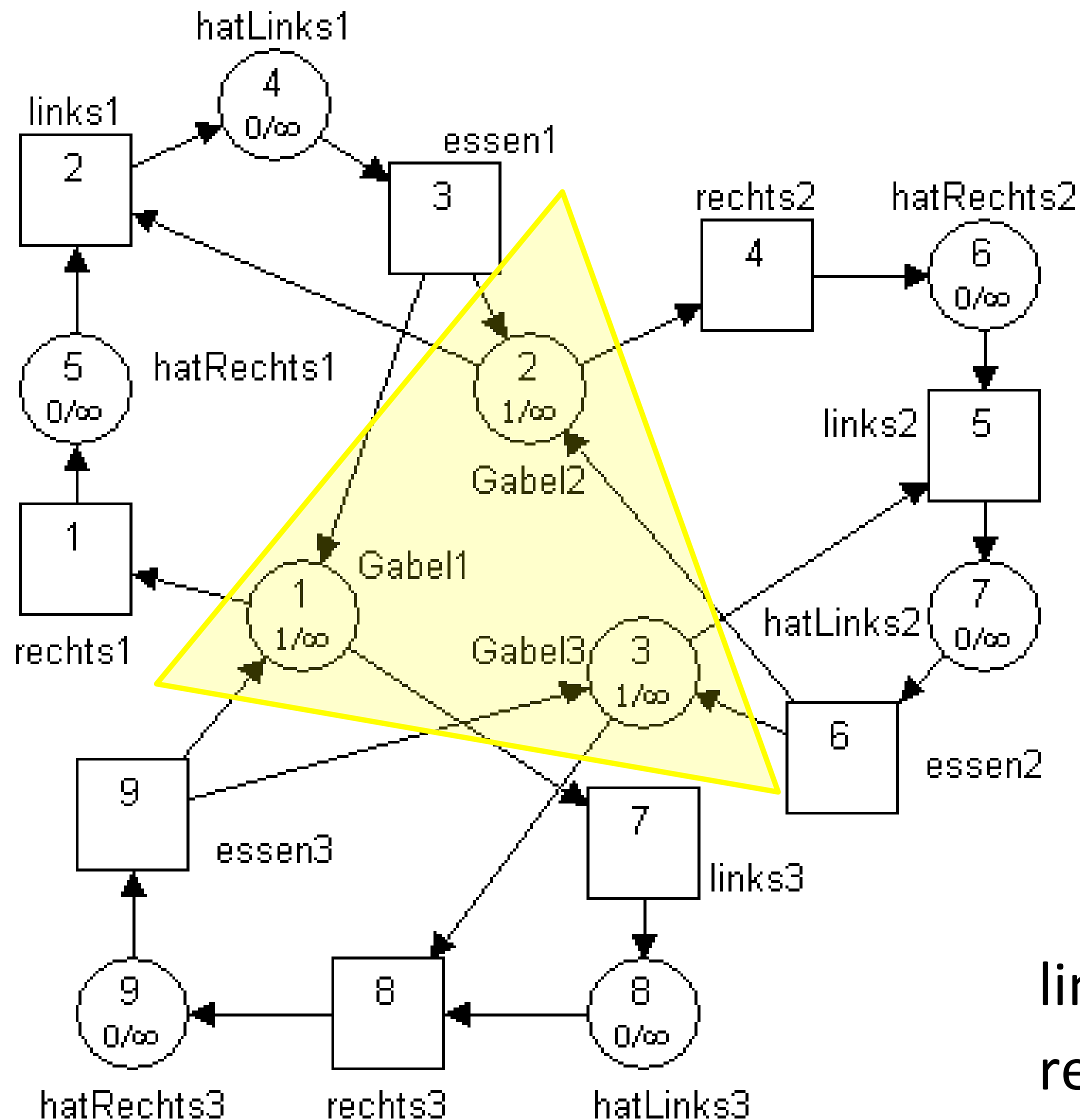

```
M020:   3 (  0  0  0  1  1  0  1  0  0) ---t3---> M005:   1 (  1  1  0  0  0  0  1  0  0)
M021:   3 (  0  0  0  1  1  0  0  0  1) ---t3---> M007:   1 (  1  1  0  0  0  0  0  0  1)
M022:   3 (  0  0  0  0  1  1  1  0  0) ---t6---> M002:   1 (  0  1  1  0  1  0  0  0  0)
M023:   3 (  0  0  0  0  1  1  0  0  1)
M024:   3 (  0  0  0  1  0  0  1  1  0)
M025:   3 (  0  0  0  1  0  0  0  1  1) ---t9---> M003:   1 (  1  0  1  1  0  0  0  0  0)
M026:   3 (  0  0  0  0  0  1  1  1  0) ---t6---> M006:   1 (  0  1  1  0  0  0  0  1  0)
M027:   3 (  0  0  0  0  0  1  0  1  1) ---t9---> M004:   1 (  1  0  1  0  0  1  0  0  0)
Graph ist komplett erstellt!
```

- Erreichbarkeitsgraph zeigt Deadlock
- Deadlock durch Regeln (z. B. wer nimmt in welcher Reihenfolge) vermeidbar

Dinierende Philosophen (4/4) – ohne Deadlock

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

rechts
links



rechts
links

links
rechts

Theoretische Aussagen zu Petri-Netzen

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Die Frage für ein gegebenes Petri-Netz, ob eine Markierung erreichbar ist, ist unendscheidbar

- Die Frage für ein gegebenes Petri-Netz, ob eine Markierung M überdeckt werden kann, ist entscheidbar (konstruiere Überdeckungsgraph, analysiere alle Markierungen M‘, falls M‘≥M, dann überdeckbar)

- Sind alle Stellen beschränkt, sind die genannten Probleme trivial entscheidbar

- Viele weitere Berechnungen möglich, z. B. Invarianten, die mit linearer Algebra berechnet werden