

# Grundbegriffe der Theoretischen Informatik

Sommersemester 2016 - Thomas Schwentick

## 0: Einleitung

Version von: 12. April 2016 (13:13)

# Inhalt

## ▷ **0.1 Was ist Theoretische Informatik?**

0.2 Themen der GTI-Vorlesung

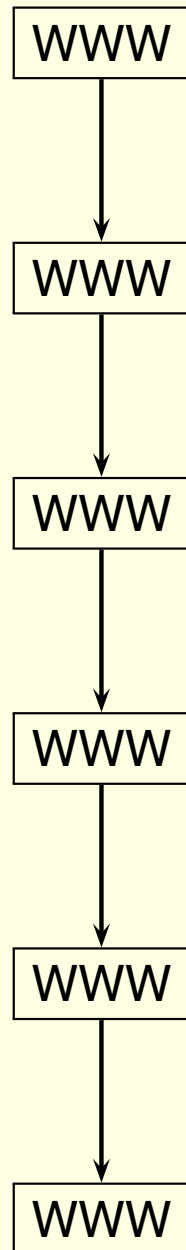
0.3 Literatur


0.4 Organisatorisches

# Warum Theorie der Informatik helfen kann

- 2013 in der Presse (<http://heise.de/-1803813>)
- Um auf die Kontaktliste eines gesperrten iPhones zuzugreifen, konnte man
  - es einschalten
  - den „Entsperren“-Slider nach rechts ziehen
  - auf „Notruf“ tippen
  - auf den Ausschalt-Knopf drücken, bis die Ausschalt-Option angezeigt wird
  - „Abbrechen“ klicken
  - eine Notrufnummer wählen, verbinden und sofort wieder auflegen
  - das Gerät aus- und einschalten
  - den „Entsperren“-Slider ziehen
  - den Ausschaltknopf gedrückt halten
  - kurz bevor die Ausschalt-Option angezeigt wird, auf „Notruf“ drücken
- Die Kontakte öffnen sich und bleiben geöffnet, solange der Ausschaltknopf gedrückt bleibt
- Besser wäre es, beweisen zu können, dass es eine solche Lücke nicht gibt...

# Wichtige Themen der Informatik – sehr grob



- Schritte auf dem Weg von der Programmidee zu Programm
    - ... und zugehörige Lehrveranstaltungen  ganz grob
  - Spezifikation:
    - Software-Technik, DAP I, Formale Methoden des System-Entwurfs
  - Wahl geeigneter Algorithmen:
    - DAP II, Effiziente Algorithmen, Komplexitätstheorie
  - Umsetzung der Algorithmen in ein Programm:
    - DAP I, Software-Technik, SoPra
  - Übersetzung des Programms:
    - Übersetzerbau
  - Ausführung des Programms:
    - Rechnerstrukturen, Betriebssysteme, HaPra
- 
- Die GTI bezieht sich auf mehrere dieser Schritte

# Wunschzettel der Informatiker

- Wünsche für die einfache Softwareerstellung
  - Zu jeder Spezifikation sollte es einen Algorithmus und ein Programm geben
  - Programme sollten aus der Spezifikation automatisch erzeugt werden
  - Es sollte automatisch überprüft werden können, ob ein Programm seiner Spezifikation entspricht
  - Programme sollten möglichst klein sein
  - Programme sollten möglichst schnell sein
  - Die Übersetzung von Programmen in Maschinenprogramme sollte automatisch erfolgen
- In dieser Vorlesung werden wir Grenzen für die Erfüllung dieser Wünsche kennen lernen
- Diese Grenzen waren unter den ersten Erkenntnissen der Theoretischen Informatik — bevor es mit der praktischen Informatik so richtig los ging
- Aber was ist überhaupt Theoretische Informatik?

# Was ist Theoretische Informatik? (1/2)

- Was ist Theoretische Informatik?
- Die Antwort auf diese Frage hat zwei Komponenten:
  - Die Methoden der Theoretischen Informatik
  - Die Inhalte der Theoretischen Informatik
- Die Theoretische Informatik verwendet die **mathematische Methode** für die **Grundlagen der Informatik**
- Ganz grob lässt sich das durch die folgende „Gleichung“ ausdrücken:

$$\frac{\text{Theoretische Informatik}}{\text{Informatik}} \approx \frac{\text{Mathematik}}{\text{Physik}}$$

# Methodik der Theoretischen Informatik

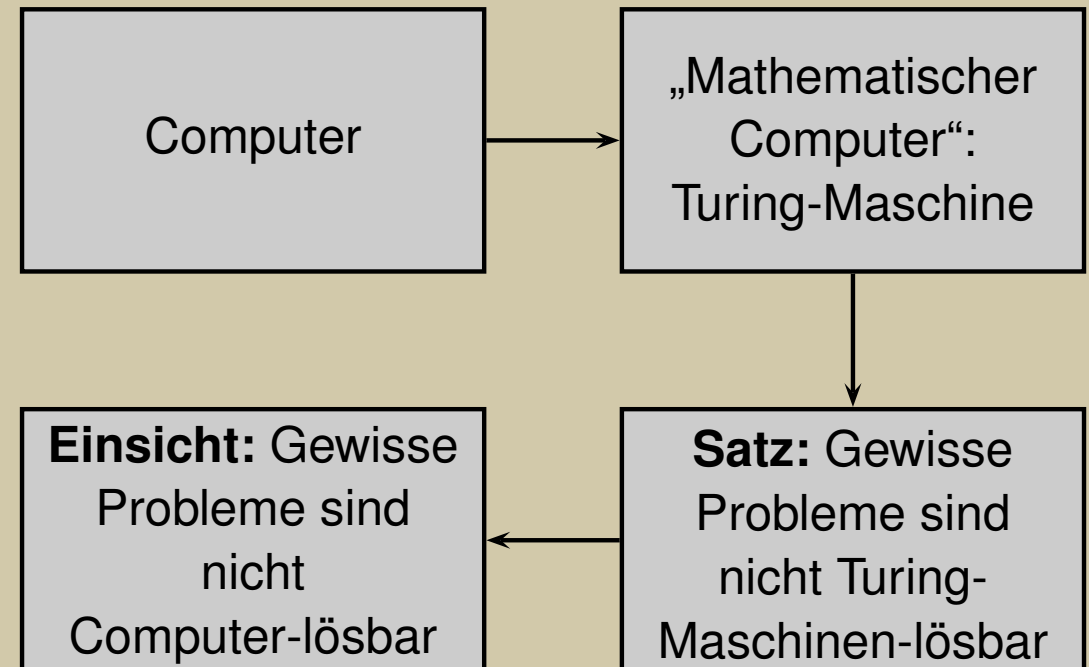
- **Typische Vorgehensweise:**

- Modellierung von im Zusammenhang mit Computern und Berechnungen auftretenden Phänomenen durch mathematische Objekte
- Formulierung von Aussagen über diese Objekte
- Beweis dieser Aussagen

- Warum ist der mathematisch präzise Ansatz (inklusive Beweisen) für die Informatik sinnvoll?

- Die Semantik von Spezifikationen sollte präzise definierbar und beispielsweise die Äquivalenz von Spezifikation und Modell beweisbar sein
- Auch die Semantik von Algorithmen/Programmen sollte präzise definierbar sein und Korrektheits- und Aufwandsaussagen sollten sich beweisen lassen

## Beispiel



# Was ist Theoretische Informatik? (2/2)

Artikel

Diskussion

Lesen

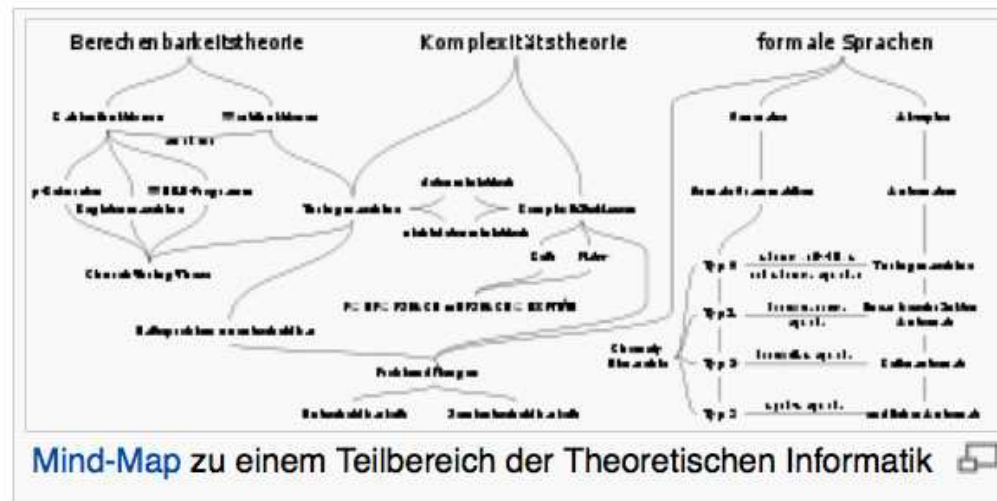
Bearbeiten

Suche



## Theoretische Informatik

Die **Theoretische Informatik** beschäftigt sich mit der Abstraktion, Modellbildung und grundlegenden Fragestellungen, die mit der Struktur, Verarbeitung, Übertragung und Wiedergabe von **Informationen** in



Mind-Map zu einem Teilbereich der Theoretischen Informatik

Zusammenhang stehen. Ihre Inhalte sind **Automatentheorie**, Theorie der **formalen Sprachen**, **Berechenbarkeits-** und **Komplexitätstheorie**, aber auch **Logik** und **formale Semantik** sowie die **Informations-**, **Algorithmen-** und **Datenbanktheorie**.

©Wikipedia

- Wir werfen nun einen Blick auf die Inhalte der Theoretischen Informatik



# Teilgebiete der Theoretischen Informatik (1/3)

- **Berechenbarkeit: „Was ist überhaupt möglich?“**
  - Was ist ein Algorithmus?
  - Welche Probleme lassen sich algorithmisch lösen?
  - Wie lassen sich die algorithmisch nicht lösbaren Probleme klassifizieren?
- **Komplexitätstheorie: „Was ist effizient möglich?“**
  - Klassifikation algorithmischer Probleme nach ihrer prinzipiellen Schwierigkeit
  - Vergleich der dabei entstehenden Klassen
  - Besondere Berechnungsmodi: zufallsgesteuert, parallel, nicht-deterministisch
  - **P** vs. **NP**
- **Effiziente Algorithmen: „Wie geht es am schnellsten?“**
  - Effiziente Algorithmen für konkrete Probleme
  - Datenstrukturen
  - Algorithmische Geometrie, Graph-Algorithmen, ...
  - Parallele Algorithmen, Online-Algorithmen, ...

# Teilgebiete der Theoretischen Informatik (2/3)

- **Formale Sprachen: „Wie sieht ein Programm aus?“**
  - Wie lässt sich die syntaktische Struktur von Computer-Programmen beschreiben?
  - Reguläre Sprachen
  - Kontextfreie Sprachen
  - Sprachen von unendlichen Zeichenketten, Bäumen etc.
- **Semantik: „Was bedeutet ein Programm?“**
  - Wie lässt sich die Bedeutung/Wirkung von Computer-Programmen so formalisieren, dass z.B. ihre Korrektheit bewiesen werden kann?
  - Semantik von Programmiersprachen
  - Funktionale und logische Programmiersprachen
  - Model Checking
- **Viele weitere Gebiete:**
  - Theorie verteilter Systeme
  - Datenbanktheorie
  - ...

## Teilgebiete der Theoretischen Informatik (3/3)

- Diese Aufzählung der Teilgebiete der Theoretischen Informatik ist nicht vollständig
- Die **Special Interest Group on Algorithms and Computation Theory (SIGACT)** der **Association for Computing Machinery (ACM)** schreibt dazu:
  - „TCS covers a wide variety of topics including algorithms, data structures, computational complexity, parallel and distributed computation, probabilistic computation, quantum computation, automata theory, information theory, cryptography, program semantics and verification, machine learning, computational biology, computational economics, computational geometry, and computational number theory and algebra“
- Die GTI-Vorlesung behandelt hiervon nur einen kleinen Ausschnitt
  - Allerdings ist die Stoffauswahl ziemlich kanonisch und unterscheidet sich von Uni zu Uni meist nur wenig
  - Darüber hinaus werden viele der hier behandelten Grundbegriffe auch in den anderen Bereichen der Theoretischen Informatik und in vielen angewandten Teilen der Informatik verwendet

# Inhalt

0.1 Was ist Theoretische Informatik?

▷ **0.2 Themen der GTI-Vorlesung**

0.3 Literatur

0.4 Organisatorisches

# Die 4 Themen der GTI-Vorlesung - A: Reguläre Sprachen

- Die **regulären Sprachen** bilden eine Klasse von einfachen Sprachen mit
  - vielen Anwendungen
  - vielen äquivalenten Charakterisierungen
  - weiteren angenehmen Eigenschaften

- **Beispiele für Anwendungen**

- Bezeichner in Programmiersprachen
- Verifikation zustandsbasierter Systeme
- Schemasprachen für XML
- Zeichenkettensuche (grep)
- Spezifikation zulässiger Eingaben für Web-Formulare

- **Viele Charakterisierungen:**

- Reguläre Ausdrücke
- Endliche Automaten
  - \* deterministisch oder nicht-deterministisch
  - \* 1-Weg oder 2-Weg
  - \* mit oder ohne  $\epsilon$ -Übergänge
- Grammatiken, logische Formeln, ...

- **Angenehme Eigenschaften:**

- einfache Spezifikationssprache:
  - reguläre Ausdrücke
- einfache Testalgorithmen
  - endliche Automaten
- Automatische Umwandlung von Spezifikationen in Testalgorithmen
- Optimaler Testalgorithmus konstruierbar
- Methode zum Testen, ob Spezifikationen und Programme äquivalent sind
- Methode zum Nachweis, dass eine Sprache nicht regulär ist

# Die 4 Themen der GTI-Vorlesung - B: Kontextfreie Sprachen

- Die **kontextfreien Sprachen** bilden eine weitere Klasse einfacher Sprachen:
  - ausdrucksstärker als die regulären Sprachen
  - auch viele Charakterisierungen
  - angenehme Eigenschaften, aber nicht mehr ganz so viele
- Hauptanwendung: Syntaktische Struktur von Programmiersprachen

- **Charakterisierungen**
  - Spezifikationssprache: Grammatiken, bzw. Syntaxdiagramme
  - Algorithmen: Kellerautomaten (richtig effizient nur für Teilklassen)
- **Angenehme Eigenschaften**
  - automatische Umwandlung von Spezifikation in Testalgorithmus
  - Methode zum Nachweis, dass eine Sprache nicht kontextfrei ist

# Die 4 Themen der GTI-Vorlesung - C: Berechenbarkeit

- **Hauptfragen:**

- Was heißt „berechenbar“?
- Gibt es algorithmische Probleme, die nicht berechenbar sind?
- Falls ja, wie sehen solche Probleme aus?

- **Was heißt berechenbar?**

- Es gibt viele Modelle, die den Begriff eines Algorithmus und damit Berechenbarkeit formalisieren
- Praktisch alle sind äquivalent!
- Church-Turing-These: Diese Modelle definieren Berechenbarkeit
- Wir werden kennen lernen:
  - \* Turing-Maschinen
  - \* WHILE-Programme
  - \* Rekursive Funktionen

- **Und was ist nicht berechenbar?**

- Wir werden sehen: es gibt viele, auch praktisch relevante Probleme, die sich algorithmisch nicht lösen lassen

# Die 4 Themen der GTI-Vorlesung - D: Komplexitätstheorie

- **Hauptfragen:**

- Was heißt „effizient berechenbar“?
- Gibt es algorithmische Probleme, die berechenbar, aber nicht effizient berechenbar sind?
- Falls ja, wie sehen solche Probleme aus?

- **Was heißt effizient berechenbar?**

- Weitgehender Konsens:
  - \* Rechenzeit wächst höchstens polynomiell in Größe der Eingabe
- Die genannten Berechnungsmodelle sind auch in dieser Hinsicht äquivalent
- Komplexitätsklasse: **P**

- **Wie sehen nicht effizient berechenbare Probleme aus?**

- Es gibt tausende von praktisch relevanten Problemen, für die
  - \* kein effizienter Algorithmus bekannt ist
  - \* kein Beweis, dass sie nicht effizient berechenbar sind, bekannt ist
- Die meisten davon sind im folgenden Sinne äquivalent:
  - \* Hat eines einen effizienten Algorithmus, so haben alle einen
  - \* Hat eines keinen effizienten Algorithmus, so hat keines einen
- **NP**-vollständige Probleme
- **P = NP**-Frage



# Sichere Systeme

- Das iPhone-Beispiel illustriert, dass sichere Systeme ein wichtiges Thema für die Informatik darstellen
  - Sie werden uns als Querschnittsthema in mehreren Teilen der Vorlesung beschäftigen
  - Meistens in der Form eines algorithmischen Problems dieser Art:
    - Gegeben: ein Modell  $S$  eines Systems und eine formale Beschreibung  $E$  einer Eigenschaft
    - Lässt sich automatisch überprüfen, ob  $S$  die Eigenschaft  $E$  hat?
- Zum Beispiel:
    - $S$  ein Smartphone-OS,  
 $E$ : „Kein Zugang ohne PIN möglich“
    - $S$  ein Programm,  
 $E$ : „ $S$  terminiert für jede Eingabe“
    - $S$  ein Kommunikationsprotokoll,  
 $E$ : „ $S$  führt niemals zu einer Verklemmung“
    - $S$  ein Schaltkreis,  
 $E$  eine Boolesche Funktion, die  $S$  berechnen soll
- Wir werden Fälle sehen, in denen eine solche automatische Überprüfung möglich ist, aber auch andere...

# Was sollen Sie in GTI lernen?

- **Erkenntnisse:**

- Grenzen der Möglichkeiten von Computern: nicht berechenbare Probleme
- Praktische Grenzen von Computern: nicht effizient lösbare Probleme
- ...

- **Fähigkeiten:**

- Abstraktion
- Formalisieren
- Analyse, z.B.: Erkennen von schwierigen Berechnungsproblemen
- ...

- **Grundwissen & Handwerkszeug:**

- Formale Grundlagen des Rechnens mit Computern
- Umgang mit regulären Sprachen, regulären Ausdrücken und Automaten
- Reguläre Sprachen / endliche Automaten „erkennen“
- Theoretische Grundlagen für Übersetzer
- ...

# Das Semester im Überblick

## 0: Einleitung

### A: Reguläre Sprachen

- 1: Reguläre Ausdrücke
- 2: Endliche Automaten
- 3: Äquivalenz der Modelle
- 4: Minimierung endlicher Automaten
- 5: Automatensynthese, Grenzen und Algorithmen
- 6: Anwendungen regulärer Sprachen

### B: Kontextfreie Sprachen

- 7: Kontextfreie Grammatiken
- 8: Normalformen und Erweiterungen
- 9: Kellerautomaten
- 10: Äquivalenz der Modelle
- 11: Pumping-Lemma, Algorithmen und Abschlusseigenschaften
- 12: Wortproblem und Syntaxanalyse

### C: Berechenbarkeit

- 13: Erste Erkenntnisse
- 14: Verschiedene Berechnungsmodelle
- 15: Turing-Maschinen
- 16: Die Church-Turing-These
- 17: Grenzen der Berechenbarkeit
- 18: Weitere unentscheidbare Probleme

### D: Komplexitätstheorie

- 19: Polynomielle Zeit
- 20: Schwierige algorithmische Probleme
- 21: Der Satz von Cook
- 22: Weitere **NP**-vollständige Probleme
- 23: **NP**: Weitere Erkenntnisse
- 24: Zufallsbasierte Algorithmen
- 25: Zufallsbasierte Komplexitätsklassen

- Die Kapitelstruktur wird sich voraussichtlich etwas ändern

# Inhalt

0.1 Was ist Theoretische Informatik?

0.2 Themen der GTI-Vorlesung

▷ **0.3 Literatur**

0.4 Organisatorisches

# Literatur

- Hopcroft, Motwani, Ullman. Einführung in die Automatentheorie, Formale Sprachen und Berechenbarkeit. Pearson.  
(ältere Auflagen: ... und Komplexitätstheorie)
    - Umfassend, viele Beispiele, gut aufgebaut, sehr verständlich
  - Wegener. Theoretische Informatik. Teubner.
    - gut aufgebaut, verständlich, mittlerer Umfang
  - Folienskript SoSe 2015, über die Homepage der Veranstaltung
    - Kurz, wenige Beispiele, kostenlos
  - G. Vossen, U. Witt: Grundlagen der Theoretischen Informatik mit Anwendungen. Vieweg.
    - Umfassend, viele Beispiele, gut aufgebaut, verständlich
  - Schöning. Theoretische Informatik kurz gefasst. Spektrum.
    - gut aufgebaut, sehr verständlich, kürzer als die anderen, nicht so umfassend
- 
- Das Erscheinungsjahr bzw. die Auflage ist jeweils nicht so wichtig (neuer ist natürlich besser...), aber beim erstgenannten Buch sollte unbedingt Herr Motwani einer der Autoren sein

# Inhalt

0.1 Was ist Theoretische Informatik?

0.2 Themen der GTI-Vorlesung

0.3 Literatur

▷ **0.4 Organisatorisches**

# Die Bestandteile der Veranstaltung

- Vorlesung
- Übungsaufgaben
- Übung
- Tutorien
- Prüfungen
- GTI im Web

# Vorlesung

- **Termine:**

- Dienstag, 10:15 – 11:50 Uhr, HG II, HS 3
- Donnerstag, 12:15 – 13:50 Uhr, HG II, HS 3
- (jeweils 5 Minuten Pause)

- Durchgängiger Einsatz von **Folien**
- Erläuterungen an der Tafel
- Die Folien können von der Webseite geladen werden

- Kapitel 1: 19.4.

- **Zweck der Vorlesung:**

- Vermittlung aller wesentlichen Inhalte

- **Gebrauchsanleitung für die Vorlesung:**

- Denken Sie mit
- Stellen Sie Fragen
- Schreiben Sie nur das Nötigste mit
- **Klappen Sie Ihr Notebook bitte zu!**

- **Nachbereitung:**

- Arbeiten Sie die Vorlesung nach
- Geben Sie sich dabei erst zufrieden, wenn Sie jedes Detail jeder Folie (mindestens einmal!) verstanden haben



# Übungsaufgaben (1/3)

- **Zweck der Übungsaufgaben:**

- Wiederholung der Inhalte der Vorlesung
- Erkennen, wo es mit dem Verständnis noch hapert
- Entwicklung der Fähigkeit, die in der Vorlesung gelernten Techniken **anzuwenden**
- Klausurvorbereitung

- **Anforderungen an die Lösungen**

- Lösungen sind nur vollständig, wenn sie begründet und erklärt werden
- Falls Beweise erwartet werden, wird dies in der Aufgabenstellung ausdrücklich erwähnt

- **Gruppenarbeit**


- Sie können die Übungsaufgaben zusammen mit anderen bearbeiten und gemeinsam Lösungswege suchen
- Das **Aufschreiben** der Lösungen erfolgt aber individuell

- **Sanktionen:**

- Wenn mehrere Personen offensichtlich voneinander abgeschriebene Lösungen abgeben, erhalten sie alle 0 Punkte
- Im Wiederholungsfall ist die Studienleistung verwirkt und wir behalten uns weitere Maßnahmen vor

# Übungsaufgaben (2/3)

## • Übungsaufgaben-Lebenszyklus:

- Ausgabe Übungsblatt:
  - \* dienstags, Woche  $n$  in der Vorlesung
- Abgabe Lösungen:
  - \* dienstags Woche  $n+1$  (23:59 Uhr)
-  aber beachten Sie die Öffnungszeiten!
- Besprechung:
  - \* Übung von Donnerstag Woche  $n+1$  bis Mittwoch Woche  $n+2$

## • Abgabe und Korrektur:

- Abgabe
  - \* in der Vorlesung
  - \* sonstige Möglichkeiten: siehe Übungsblatt
- Ihre Lösungen werden korrigiert
- Zu jedem Übungsblatt wird eine **Beispiel-Lösung** veröffentlicht  
(am Ende von Woche  $n+2$ )
- Wichtig: es kann mehrere Lösungswege geben, die Beispiel-Lösung repräsentiert meist nur **einen** davon

# Übungsaufgaben (3/3)

- **Anzahl und Art der Übungsblätter**

- Blatt 0 mit Präsenzaufgaben (erste Übungsstunde)
- Übungsblätter 1-12 zur Bearbeitung und Abgabe

- **Aufgabentypen:**

- Je Blatt ca. drei Aufgaben, deren Bearbeitung Punkte ergeben kann
- möglicherweise eine schwierigere Zusatzaufgabe, die bei Bearbeitung korrigiert und bepunktet wird, aber nicht besprochen wird

- **Übungsblatt 1**

- Blatt 1 bezieht sich auf den Stoff der Vorlesungen vom 19.4. (Dienstag) und 21.4. (Donnerstag)
- Da die Zeit von Donnerstag bis Dienstag etwas kürzer ist, sind die Aufgaben etwas weniger umfangreich und ergeben etwas weniger Punkte

# Übung

- In den Übungsstunden sollen erlernt werden:
  - mündliche Präsentation von Inhalten der Theoretischen Informatik
  - die Fähigkeit zur Reflexion von Inhalten der Theoretischen Informatik, auch im Gespräch mit anderen
- Um dieses Ziel zu erreichen
  - werden die Lösungen der Übungsaufgaben von den Studierenden vorgeführt
  - werden alternative Lösungswege und fehlerhafte Ansätze unter den Studierenden diskutiert
  - werden Präsenzaufgaben bearbeitet

- **Termine:** siehe Übungsseite

- **Beginn:**
  - Donnerstag, 21.4.
  - In der ersten Übungsstunde wird ein Ü-Blatt mit „Präsenzaufgaben“ besprochen

- **Anmeldung zu den Übungsgruppen:**
  - <http://ess.cs.tu-dortmund.de/ASSESS/>
  - Anmeldung ist schon möglich
  - bis Sonntag, 17.4.
  - Bekanntgabe der Gruppeneinteilung am Montag, 18.4.

# Tutorien

- **Zweck des Tutoriums:**

- Wiederholung des Stoffes und Prüfungsvorbereitung
- Besondere Unterstützung der „ThIfAI“-Hörer

- **Termine:**

(B2) Mo 14-16, OH12, 1.055 (ThIfAI)

(A1) Mi 14-16, OH12, E.003 (ThIfAI)

(A2) Do 10-12, OH14, E 23

(B1) Fr 12-14, OH12, 3.031

- Beginn: Mittwoch, 20.4. (A1)!
- ThIfAI-Hörer haben bei Raumüberfüllung an den gekennzeichneten Terminen Vorrang

- **Themen:**

- Werden jeweils auf der Übungsseite bekannt gegeben
  - \* In den beiden A-Tutorien und den beiden B-Tutorien jeweils das gleiche Thema
- Themenvorschläge sind willkommen (INPUD)

# Prüfung und Studienleistung (1/2)

- Klausurtermine:
  - 1. Klausur am 15.8.2016, 8:00-11:00 Uhr
  - 2. Klausur: 11.10.2016, 8:00-11:00 Uhr
- **Erlaubte Hilfsmittel:** zwei DIN A4 Blätter mit selbst erstellten handschriftlichen Notizen
  - Details dazu folgen noch
- Anmeldung zur Klausur im BOSS
  - **Vergewissern Sie sich bitte, dass die Anmeldung im BOSS geklappt hat!!!**
  - Studierende, für deren Studiengänge keine Anmeldung im BOSS möglich ist, melden sich direkt bei uns an (nähere Informationen folgen)

## Prüfung und Studienleistung (2/2)

- Voraussetzung zur Teilnahme an der Klausur ist das Erbringen der **Studienleistung**:
  - Erreichen von
    - \*  $\geq$  **29** Punkten aus 87 Punkten der Aufgabenblätter 1-6
    - \*  $\geq$  **30** Punkten aus 90 erreichbaren Punkten der Aufgabenblätter 7-12
- Zum Erwerb der genannten Kompetenzen und als Vorbereitung zur Klausur empfehlen wir dringend die aktive Teilnahme an den Übungsgruppen

- Die Studienleistung muss nicht erbracht werden von:
  - Studierenden des Diplomstudiengangs
  - Studierenden, die die Studienleistung im letzten SoSe erbracht haben
  - Studierenden, die die Studienleistung irgendwann erbracht haben **und** bereits eine Klausur geschrieben haben (laut Auskunft Prüfungsausschuss)
- Dringende Empfehlung für diese Gruppen: nehmen Sie aktiv am Übungsbetrieb teil

# GTI im Web

- Auf der **Vorlesungsseite** finden sich:
  - die Vorlesungsankündigung
  - die Übersicht der Übungstermine
  - aktuelle Informationen
  - die Vorlesungsfolien
  - Übungsblätter
  - Beispiel-Lösungen
  - Informationen zu den Klausuren
- Die Materialien sind aus dem Uninetz erreichbar (VPN) oder per User/Kennwort-Kombination GTI16 Turing36

- Die Vorlesungsfolien gibt es in drei Varianten:
  - Zum Anschauen am Bildschirm (jeder „Klick“ auf Extra-Seite)
  - Zum Ausdrucken
  - Zum Ausdrucken mit reduziertem Farbeinsatz
  - Öko-Tipp: 4-auf-1 ist noch gut lesbar
  - Hinweis: nicht alle Abbildungen/Fotos werden in den Foliensätzen zur Verfügung gestellt
- Im INPUD-Forum gibt es eine moderierte Diskussionsgruppe zu GTI
- Dort können Sie Fragen stellen, mit anderen über Vorlesung und Übung diskutieren, sich Lernpartner suchen usw.



Zu guter Letzt...

Viel Erfolg!

Und auch etwas Spaß!

# Grundbegriffe der Theoretischen Informatik

Sommersemester 2016 - Thomas Schwentick

## Teil A: Reguläre Sprachen

### 1: Reguläre Ausdrücke: Motivation, Definition, Beispiele

Version von: 19. April 2016 (12:10)

# Inhalt

- ▷ **1.1 Einleitung: Beschreibung von e-Mail-Adressen**
- 1.2 Alphabete, Wörter, Sprachen
- 1.3 Reguläre Ausdrücke: Syntax und Semantik
- 1.4 Reguläre Ausdrücke: Beispiele, Erweiterungen, Äquivalenzen

# Beispiel: HTML-Formular mit e-Mail-Adresse

## Hauptziele der nächsten vier Stunden

- Wir betrachten heute eine Methode, die es ermöglicht, **einfache** syntaktische Bedingungen für Zeichenketten **unzweideutig** zu beschreiben
  - Als Beispiel werden wir e-Mail-Adressen beschreiben
- Danach (am Donnerstag) werden wir eine einfache Klasse von Programmen kennen lernen, die solche Bedingungen überprüfen können
- Schließlich werden wir Methoden kennen lernen, mit denen wir Beschreibungen automatisch in Testprogramme übersetzen und diese minimieren können

# e-Mail-Adressen: informelle Beschreibung

- Typische e-Mail-Adressen:
  - President@  
whitehouse.gov
  - thomas.schwentick@  
cs.tu-dortmund.de
  - zu\_hause.hanna-  
mustermann82@  
irgendein.provider.de

- Die genauen **Regeln für e-Mail-Adressen** sind ziemlich **kompliziert**
- Wir betrachten deshalb nur eine **Annäherung** der korrekten e-Mail-Adressen

## „Definition“: e-Mail-Adressen

- E-Mail-Adressen haben die Form  
**„Lokaler-Name@Domain-Name“**
- Ein **„Label“** verwendet Zeichen aus: a-z, A-Z, 0-9, „-“, „\_“
- Am Anfang eines Labels steht immer ein Buchstabe
- Der **lokale Name** besteht aus beliebig vielen (aber mindestens einem) Labels, die durch „.“ getrennt werden
- Der **Domain-Name** besteht aus beliebig vielen (aber mindestens zwei) Labels, die durch „.“ getrennt werden
- Das letzte Label im Domain-Namen besteht aus zwei bis vier Buchstaben

## Zwischenziele für die heutige Vorlesung

- Formale Beschreibung aller Zeichenketten, die diesen Regeln entsprechen
- Da die Beschreibung (später) automatisch in ein Programm übersetzt werden soll, brauchen wir einen Beschreibungs-Formalismus mit einer klaren **Semantik**:  
**reguläre Ausdrücke**



Vorher benötigen wir noch ein paar Grundbegriffe

# Inhalt

1.1 Einleitung: Beschreibung von e-Mail-Adressen

▷ **1.2 Alphabete, Wörter, Sprachen**

1.3 Reguläre Ausdrücke: Syntax und Semantik

1.4 Reguläre Ausdrücke: Beispiele, Erweiterungen, Äquivalenzen

# Zeichenketten und Sprachen: informell

- Eine **Zeichenkette** (oder: ein **String**) ist eine (endliche) Folge von Zeichen
- Im Kontext von Computern spielen Zeichenketten und Mengen von Zeichenketten eine große Rolle

## Beispiel

- Jedes Computer-Programm

```
public class Hallo
{
    public static void main(String[] args)
    {
        System.out.println("Hallo Welt!");
    }
}
```

ist eine Zeichenkette:

```
public class Hallo { public static
void main(String[] args) { System.
out.println("Hallo Welt!"); } }
```

- Die Menge der erlaubten Zeichen nennen wir **Alphabet**
- Eine Menge von Zeichenketten heißt **(formale) Sprache**

## Beispiel

- Die Menge aller syntaktisch korrekten JAVA-Programme ist eine Sprache

## Beispiel

- Die Menge aller syntaktisch korrekten JAVA-Programme, die „Hallo Welt!“ ausgeben, ist eine Sprache

## Beispiel

- Die Menge der Befehlssequenzen einer Fernbedienung für einen Fernseher, nach deren Ausführung der Lautsprecher ausgeschaltet ist, ist eine Sprache über dem Alphabet der „Tasten“



Jetzt: formale Definitionen

# Alphabete, Wörter, Sprachen (1/3)

## Definition: Alphabet 1.1

- Ein Alphabet  $\Sigma$  ist eine endliche, nicht-leere Menge
- Die Elemente eines Alphabets heißen Zeichen oder Symbole
- Notation für Alphabete:
  - große, griechische Buchstaben, z.B.  $\Sigma$  („Sigma“),  $\Delta$  („Delta“),  $\Gamma$  („Gamma“)
- Notation für einzelne Zeichen:
  - kleine griechische Buchstaben, z.B.  $\sigma$  („sigma“),  $\tau$  („tau“), . . .

## Beispiel

- $\{A, \dots, Z\}$
- $\{0, 1\}$
- $\{\leftarrow, \uparrow, \downarrow, \rightarrow\}$

## Definition

- Ein Wort  $w$  über einem Alphabet  $\Sigma$  ist eine endliche Folge  $\sigma_1 \cdot \dots \cdot \sigma_n$  von Zeichen von  $\Sigma$
- Wir bezeichnen Wörter auch als **Strings** oder **Zeichenketten**
- Notation für Wörter:  $u, v, w, \dots$
- Die Länge  $|w|$  von  $w = \sigma_1 \cdot \dots \cdot \sigma_n$  ist die Anzahl  $n$  der Zeichen von  $w$
- $\epsilon$ : Wort der Länge 0 (leeres Wort)

## Beispiel

- $\Sigma = \{a, \dots, z\}$
- Wörter über  $\Sigma$ :
  - informatik
  - jdgsfsxnnffishwrafdhug
  - abba
  - j
  - $\epsilon$



## Alphabete, Wörter, Sprachen (2/3)

### Definition

- Eine Sprache über einem Alphabet  $\Sigma$  ist eine (endliche oder unendliche) Menge von Wörtern über  $\Sigma$

### Beispiel

- Menge aller syntaktisch korrekten e-Mail-Adressen
- Menge aller Strings über  $\{0, 1\}$ , die den Teilstring **010** enthalten
- Menge aller Strings über  $\{0, 1\}$ , die abwechselnd 0 und 1 enthalten

# Alphabete, Wörter, Sprachen (3/3)

## Beispiel

- Menge aller HTML-Tags:

$M_{\text{HTML}} \stackrel{\text{def}}{=} \{ \langle A \rangle, \langle ABBREV \rangle, \langle ACRONYM \rangle, \langle ADDRESS \rangle, \langle APPLET \rangle, \langle AREA \rangle, \langle AU \rangle, \langle AUTHOR \rangle, \langle B \rangle, \langle BANNER \rangle, \langle BASE \rangle, \langle BASEFONT \rangle, \langle BGSOUND \rangle, \langle BIG \rangle, \langle BLINK \rangle, \langle BLOCKQUOTE \rangle, \langle BQ \rangle, \langle BODY \rangle, \langle BR \rangle, \langle CAPTION \rangle, \langle CENTER \rangle, \langle CITE \rangle, \langle CODE \rangle, \langle COL \rangle, \langle COLGROUP \rangle, \langle CREDIT \rangle, \langle DEL \rangle, \langle DFN \rangle, \langle DIR \rangle, \langle DIV \rangle, \langle DL \rangle, \langle DT \rangle, \langle DD \rangle, \langle EM \rangle, \langle EMBED \rangle, \langle FIG \rangle, \langle FN \rangle, \langle FONT \rangle, \langle FORM \rangle, \langle FRAME \rangle, \langle FRAMESET \rangle, \langle H1 \rangle, \langle H2 \rangle, \langle H3 \rangle, \langle H4 \rangle, \langle H5 \rangle, \langle H6 \rangle, \langle HEAD \rangle, \langle HR \rangle, \langle HTML \rangle, \langle I \rangle, \langle IFRAME \rangle, \langle IMG \rangle, \langle INPUT \rangle, \langle INS \rangle, \langle ISINDEX \rangle, \langle KBD \rangle, \langle LANG \rangle, \langle LH \rangle, \langle LI \rangle, \langle LINK \rangle, \langle LISTING \rangle, \langle MAP \rangle, \langle MARQUEE \rangle, \langle MATH \rangle, \langle MENU \rangle, \langle META \rangle, \langle MULTICOL \rangle, \langle NOBR \rangle, \langle NOFRAMES \rangle, \langle NOTE \rangle, \langle OL \rangle, \langle OVERLAY \rangle, \langle P \rangle, \langle PARAM \rangle, \langle PERSON \rangle, \langle PLAINTEXT \rangle, \langle PRE \rangle, \langle Q \rangle, \langle RANGE \rangle, \langle SAMP \rangle, \langle SCRIPT \rangle, \langle SELECT \rangle, \langle SMALL \rangle, \langle SPACER \rangle, \langle SPOT \rangle, \langle STRIKE \rangle, \langle STRONG \rangle, \langle SUB \rangle, \langle SUP \rangle, \langle TAB \rangle, \langle TABLE \rangle, \langle TBODY \rangle, \langle TD \rangle, \langle TEXTAREA \rangle, \langle TEXTFLOW \rangle, \langle TFOOT \rangle, \langle TH \rangle, \langle THEAD \rangle, \langle TITLE \rangle, \langle TR \rangle, \langle TT \rangle, \langle U \rangle, \langle UL \rangle, \langle VAR \rangle, \langle WBR \rangle, \langle XMP \rangle \}$

- $M_{\text{HTML}}$  ist eine Sprache über dem Alphabet

$$\Sigma \stackrel{\text{def}}{=} \{ \langle, \rangle, /, A, \dots, Z, 1, \dots, 6 \}$$

- Da  $M_{\text{HTML}}$  endlich ist, ist es auch ein Alphabet

## Beispiel

```
<HTML>
  <HEAD>
    <TITLE>GTI</TITLE>
  </HEAD>
  <BODY>
    ...
  </BODY>
</HTML>
```

- ist ein Wort über dem Alphabet  $\Sigma$ ,
- kann aber auch als Wort über dem Alphabet  $M_{\text{HTML}} \cup M_{\text{/HTML}} \cup \{A, \dots, Z\}$  aufgefasst werden  
(  $M_{\text{/HTML}}$ : Menge der schließenden Tags)

# Konkatenation von Strings

## Definition

- Das Zeichen „ $\cdot$ “ bezeichnet die Operation der **Konkatenation** von Strings:
  - Sind  $u$  und  $v$  Wörter, so bezeichnet  $u \cdot v$  das Wort, das entsteht, wenn  $v$  hinter  $u$  geschrieben wird

## Beispiel

- $abb \cdot cda = abbcda$
- $a \cdot abc \cdot ba = aabcba$
- $abbc \cdot \epsilon = abbc$

## Definition

- $u^n$  bezeichnet die  $n$ -malige **Wiederholung** von  $u$ , also:
  - $u^0 = \epsilon$ ,  $u^1 = u$ ,  $u^2 = uu$ , ...
- Induktiv:  $u^0 = \epsilon$ ,  $u^{n+1} = u^n \cdot u$

## Beispiel

- $(ab)^2 = abab$
- $(aaa)^3 = aaaaaaaaaa$
- $(abc)^0 = \epsilon$
- $\epsilon^3 = \epsilon$

- Wir werden das Operationssymbol  $\cdot$  meistens weglassen und einfach  $uv$  statt  $u \cdot v$  schreiben
- Ist  $u = abc$  und  $v = bca$ , so ist also  $uv = abcbca$

# Inhalt

1.1 Einleitung: Beschreibung von e-Mail-Adressen

1.2 Alphabete, Wörter, Sprachen

▷ **1.3 Reguläre Ausdrücke: Syntax und Semantik**

1.4 Reguläre Ausdrücke: Beispiele, Erweiterungen, Äquivalenzen

# Beschreibungsformalismus: Anforderungen

- Unser Beschreibungsformalismus soll
  - möglichst einfach sein, aber
  - genügend ausdrucksstark, um z.B. die Syntax von (unserer Definition von) Mailadressen beschreiben zu können
- Um Mailadressen adäquat beschreiben zu können, benötigen wir zumindest drei Konstruktionselemente:
  - Es muss möglich sein, Teilsprachen zu **konkatenieren**:
    - \* Lokaler-Name@Domain-Name
  - Es muss möglich sein, aus mehreren Alternativen **auszuwählen**:
    - \* „**Label**“ verwenden Zeichen aus: a-z, A-Z, 0-9, ...“
  - Es muss möglich sein, Elemente zu **wiederholen**:
    - \* „Der lokale Name besteht aus beliebig vielen (aber mindestens einem) Labels... “

# Reguläre Ausdrücke: „Definition durch Beispiele“

- Die **Wiederholung** wird durch  $*$  ausgedrückt:
  - $b^*$   $\equiv$  „beliebig viele  $b$ “
  - Entspricht der Sprache  $\{\epsilon, b, bb, bbb, bbbb, \dots\}$
- Die **Konkatenation** wird wie ein Produkt geschrieben:
  - $ab^*$   $\equiv$  ein  $a$  gefolgt von beliebig vielen  $b$
  - Entspricht der Sprache  $\{a, ab, abb, abbb, \dots\}$
- Die **Auswahl** wird durch  $+$  ausgedrückt:
  - $b + c^*$   $\equiv$  „ein  $b$  oder beliebig viele  $c$ “
  - Entspricht der Sprache  $\{b, \epsilon, c, cc, ccc, \dots\}$
- ✎ Die Beispiele entsprechen nicht genau der folgenden Definition der Syntax regulär Ausdrücke: es fehlen Klammern
  - Das Weglassen von Klammern werden wir aber später erlauben...

# Reguläre Ausdrücke: Syntax

## Reguläre Ausdrücke: Syntax 1.2

- Sei  $\Sigma$  ein Alphabet
- Die folgenden Regeln definieren die Menge der regulären Ausdrücke über  $\Sigma$ :

- 1) a) Das Zeichen  $\emptyset$  ist ein regulärer Ausdruck  
b) Das Zeichen  $\epsilon$  ist ein regulärer Ausdruck  
c) Für jedes  $\sigma \in \Sigma$  ist  $\sigma$  ein regulärer Ausdruck
- 2) Sind  $\alpha$  und  $\beta$  reguläre Ausdrücke, so auch
  - a)  $(\alpha\beta)$ , und
  - b)  $(\alpha + \beta)$
- 3) Ist  $\alpha$  ein regulärer Ausdruck, so auch  $(\alpha^*)$

- ✎ Wir verwenden die Abkürzung **RE** für „regulärer Ausdruck“, da der englische Begriff **regular expression** lautet
  - Entsprechend als Mehrzahl: **REs** für **regular expressions**


## PINGO-Frage: pingo.upb.de

- Welche der folgenden Zeichenketten sind reguläre Ausdrücke?

- (A)  $(ab + c)^*$
- (B)  $((ab)^* + (b(a^*)))$
- (C)  $(a + +b)$
- (D)  $((b^*)^*)$

- ✎ Wir werden bald das Weglassen „überflüssiger“ Klammern erlauben
- ✎ Wir werden zur Bezeichnung regulärer Ausdrücke meist Buchstaben vom Anfang des griechischen Alphabets verwenden:  $\alpha, \beta, \gamma, \dots$

# Reguläre Ausdrücke: Semantik (1/3)

- Wir haben jetzt festgelegt, was reguläre Ausdrücke **sind**
  - Wir haben also die **Syntax** regulärer Ausdrücke definiert
  - Wir haben aber noch **nicht** definiert, was reguläre Ausdrücke **bedeuten**
- 
- Im nächsten Schritt definieren wir deshalb die **Semantik** regulärer Ausdrücke
    - Dazu definieren wir zunächst Operatoren auf Sprachen, die den Konstruktionselementen regulärer Ausdrücke entsprechen
-  Zur Definition der Semantik der Auswahl benötigen wir keinen neuen Operator, da sie der Vereinigung entspricht

## Definition

- Sind  $L_1$  und  $L_2$  Sprachen, so sei:
  - $\underline{L_1 \circ L_2} \stackrel{\text{def}}{=} \{uv \mid u \in L_1, v \in L_2\}$
- Ist  $L$  eine Sprache, so sei
  - $\underline{L^*} \stackrel{\text{def}}{=} \{u_1 \cdots u_n \mid u_1, \dots, u_n \in L, n \in \mathbb{N}_0\}$

1.3

## Beispiel

- $\{a\}^* = \{\epsilon, a, aa, aaa, \dots\}$
- $\Sigma^*$  bezeichnet die Menge aller Strings über dem Alphabet  $\Sigma$
- $\{a\}^* \circ \{b\}^* = \{\epsilon, a, b, aa, ab, bb, aaa, aab, abb, \dots\}$
- $\{a\}^* \cup \{b\}^* = \{\epsilon, a, b, aa, bb, aaa, bbb, \dots\}$



## Reguläre Ausdrücke: Semantik (2/3)

- Die folgende Definition der Semantik regulärer Ausdrücke ordnet jedem regulären Ausdruck  $\alpha$  eine Sprache  $L(\alpha)$  zu
- Zum Beispiel soll gelten:  
$$L((a^*)) = \{\epsilon, a, aa, aaa, \dots\}$$

### Reguläre Ausdrücke: Semantik

- Für jeden regulären Ausdruck  $\alpha$  sei  $\underline{L(\alpha)}$  wie folgt definiert:
  - $* L(\emptyset) \stackrel{\text{def}}{=} \emptyset$
  - $* L(\epsilon) \stackrel{\text{def}}{=} \{\epsilon\}$
  - $* L(\sigma) \stackrel{\text{def}}{=} \{\sigma\}$ , für jedes  $\sigma \in \Sigma$
  - Sind  $\alpha$  und  $\beta$  reguläre Ausdrücke so ist
    - $* L((\alpha\beta)) \stackrel{\text{def}}{=} L(\alpha) \circ L(\beta)$ ,
    - $* L((\alpha + \beta)) \stackrel{\text{def}}{=} L(\alpha) \cup L(\beta)$
  - Ist  $\alpha$  ein regulärer Ausdruck, so ist
$$L((\alpha^*)) \stackrel{\text{def}}{=} L(\alpha)^*$$

### Definition: Reguläre Sprache

- Eine Sprache  $L$  heißt regulär, falls es einen regulären Ausdruck  $\alpha$  gibt mit  $L = L(\alpha)$

# Reguläre Ausdrücke: Semantik (3/3)

## Reguläre Ausdrücke: Semantik (Wdh.)

- Für jeden regulären Ausdruck  $\alpha$  sei  $L(\alpha)$  wie folgt definiert:

- $\ast \quad L(\emptyset) \stackrel{\text{def}}{=} \emptyset$
- $\ast \quad L(\epsilon) \stackrel{\text{def}}{=} \{\epsilon\}$
- $\ast \quad L(\sigma) \stackrel{\text{def}}{=} \{\sigma\},$

für jedes  $\sigma \in \Sigma$

- Sind  $\alpha$  und  $\beta$  reguläre Ausdrücke so ist
  - $\ast \quad L((\alpha\beta)) \stackrel{\text{def}}{=} L(\alpha) \circ L(\beta),$
  - $\ast \quad L((\alpha + \beta)) \stackrel{\text{def}}{=} L(\alpha) \cup L(\beta)$
- Ist  $\alpha$  ein regulärer Ausdruck, so ist  $L((\alpha^*)) \stackrel{\text{def}}{=} L(\alpha)^*$

## Beispiel

$$\begin{aligned} & L((((a^*)b) + (a(b^*)))) \\ &= L(((a^*)b)) \cup L((a(b^*))) \\ &= (L((a^*)) \circ L(b)) \cup (L(a) \circ L((b^*))) \\ &= (\{\epsilon, a, aa, \dots\} \circ \{b\}) \cup (\{a\} \circ \{\epsilon, b, bb, \dots\}) \\ &= \{b, a, ab, aab, abb, aaab, abbb, \dots\} \end{aligned}$$

# Zwischenbemerkung

## Vorsicht

- Wir verwenden die Symbole  $\emptyset$  und  $\epsilon$  mit zwei Bedeutungen:
  - $\emptyset$  bezeichnet einerseits die leere Menge, kann aber auch als Zeichen in einem regulären Ausdruck vorkommen
  - $\epsilon$  bezeichnet einerseits den Leerstring, kann aber ebenfalls als Zeichen in einem regulären Ausdruck vorkommen
- Das ist so üblich
- Aus dem Kontext wird immer ersichtlich sein, was jeweils gemeint ist
- Desgleichen verwenden wir  $*$  sowohl als syntaktisches Element von regulären Ausdrücken als auch als Operator

# Inhalt

1.1 Einleitung: Beschreibung von e-Mail-Adressen

1.2 Alphabete, Wörter, Sprachen

1.3 Reguläre Ausdrücke: Syntax und Semantik

▷ **1.4 Reguläre Ausdrücke: Beispiele, Erweiterungen, Äquivalenzen**

# Reguläre Ausdrücke: Beispiele

- Die Beispiel-Ausdrücke sind schwer lesbar: zu viele Klammern
- Ohne Klammern ist die Semantik zunächst unklar:  $ab+cd^*$  könnte bedeuten:
  - $((ab)+((cd)^*))$
  - $((a(b+c))d)^*$
  - $((ab)+(c(d^*)))$
- Deshalb verwenden wir Präzedenzregeln:
  - Klammern binden am stärksten
  - Dann  $*$
  - Dann Konkatination
  - Dann  $+$
- Der Ausdruck  $ab + cd^*$  steht also für  $((ab) + (c(d^*)))$

## Beispiel

- Die Menge aller Strings über  $\{0, 1\}$ , die 010 als Teilstring enthalten:

## Beispiel

- Die Menge aller Strings über  $\{0, 1\}$ , die abwechselnd 0 und 1 enthalten:

oder

## Beispiel

- Die Menge aller Strings über  $\{0, 1\}$ , die gerade viele Einsen enthalten:

# Reguläre Ausdrücke: Syntaktischer Zucker

- Um praktisch relevante Beispiele ausdrücken zu können, sind reguläre Ausdrücke manchmal etwas umständlich
- Deshalb werden in der Praxis meist **erweiterte reguläre Ausdrücke** verwendet, die abkürzende Schreibweisen wie die folgenden erlauben:

## Definition

| Abkürzung ...      | ... steht für ...          | Erläuterung                                      |
|--------------------|----------------------------|--|
| $[a - z]$          | $a + \dots + z$            |  |
| $\alpha?$          | $(\alpha + \epsilon)$      | keinmal oder einmal $\alpha$                     |
| $\alpha^+$         | $\alpha\alpha^*$           | mindestens einmal $\alpha$                       |
| $\alpha^n$         | $\alpha \dots \alpha$      | $n$ mal $\alpha$                                 |
| $\alpha^{\{m,n\}}$ | $\alpha^m (\alpha?)^{n-m}$ | mindestens $m$ -mal, höchstens $n$ -mal $\alpha$ |

- Damit können wir jetzt die Syntax von e-Mail-Adressen beschreiben:  
 $([a-zA-Z][a-zA-Z0-9\_-]*.)^*[a-zA-Z][a-zA-Z0-9\_-]*@([a-zA-Z][a-zA-Z0-9\_-]*.)^+[a-zA-Z]\{2,4\}$
- Solange nicht ausdrücklich etwas anderes gesagt wird, werden wir uns aber im Folgenden auf „reine“ reguläre Ausdrücke beschränken!

# Reguläre Ausdrücke: Äquivalenzen (1/4)

## Satz 1.1

- Es gelten folgende Äquivalenzen für beliebige reguläre Ausdrücke  $\alpha, \beta, \gamma$ :

### Assoziativität bezüglich $+$ und $\circ$

- $\alpha + (\beta + \gamma) \equiv (\alpha + \beta) + \gamma$
- $\alpha(\beta\gamma) \equiv (\alpha\beta)\gamma$

### Kommutativität bezüglich $+$

- $\alpha + \beta \equiv \beta + \alpha$

### Neutrale Elemente bezüglich $+$ und $\circ$

- $\emptyset + \alpha \equiv \alpha \equiv \alpha + \emptyset$
- $\epsilon\alpha \equiv \alpha \equiv \alpha\epsilon$

### Distributivität

- $\alpha(\beta + \gamma) \equiv \alpha\beta + \alpha\gamma$
- $(\alpha + \beta)\gamma \equiv \alpha\gamma + \beta\gamma$

### Idempotenz des Stern-Operators

- $(\alpha^*)^* \equiv \alpha^*$

### Nullelement bezüglich $\circ$ und $*$

- $\emptyset\alpha \equiv \emptyset \equiv \alpha\emptyset$
- $\emptyset^* \equiv \epsilon$

### Leerstring bzgl. $*$


- $\epsilon^* \equiv \epsilon$


## Definition

- Für zwei reguläre Ausdrücke  $\alpha, \beta$  schreiben wir  $\alpha \equiv \beta$ , falls  $L(\alpha) = L(\beta)$

## Beispiel

$$\begin{aligned} & (1 + \epsilon)(01)^*(0 + \epsilon) \\ & \equiv 1(01)^*(0 + \epsilon) + (01)^*(0 + \epsilon) \\ & \equiv 1(01)^*0 + 1(01)^* + \\ & \qquad \qquad \qquad (01)^*0 + (01)^* \end{aligned}$$

 Der letzte Ausdruck lässt sich vereinfachen zu  $(10)^* + (10)^*1 + (01)^*0 + (01)^*$   
– Das lässt sich aber nicht mit den obigen Regeln herleiten

 Die Assoziativität bezüglich  $\circ$  erlaubt uns, in einer Folge von Konkationen alle Klammern wegzulassen:  
 $((ab)c)d)e \equiv abcde \equiv a(b(c(de)))$

## Reguläre Ausdrücke: Äquivalenzen (2/4)

- Wie lassen sich die Äquivalenzen aus Satz 1.1 **be-**  
**weisen**?

- Wir betrachten einen Beispielbeweis für die Äquivalenz  $\alpha(\beta + \gamma) \equiv \alpha\beta + \alpha\gamma$

- Wir müssen zeigen, dass

$$L(\alpha(\beta + \gamma)) = L(\alpha\beta + \alpha\gamma)$$

- Wir müssen also zeigen, dass zwei Sprachen gleich sind

- Sprachen sind Mengen von Strings...

- Gleichheit von zwei Mengen  $M_1, M_2$  lässt sich meist am besten in zwei Schritten zeigen:

- $M_1 \subseteq M_2$

- $M_2 \subseteq M_1$

- Also:

- für alle  $w \in M_1$  gilt  $w \in M_2$

- für alle  $w \in M_2$  gilt  $w \in M_1$



## Reguläre Ausdrücke: Äquivalenzen (3/4)

Beweis von  $\alpha(\beta + \gamma) \equiv \alpha\beta + \alpha\gamma$

- Seien  $\alpha, \beta, \gamma$  beliebig
- Wir wollen zuerst zeigen:  $L(\alpha(\beta + \gamma)) \subseteq L(\alpha\beta + \alpha\gamma)$
- Sei also  $w$  ein beliebiger String aus  $L(\alpha(\beta + \gamma))$ 
  - ➔ es gibt  $u \in L(\alpha)$  und  $v \in L(\beta + \gamma)$  mit  $w = uv$ 
    - ☞ Semantik REs: Konkatination
  - ➔  $v \in L(\beta)$  oder  $v \in L(\gamma)$ 
    - ☞ Semantik REs: Auswahl
- Wir unterscheiden zwei Fälle:
  - 1. Fall:  $v \in L(\beta)$ 
    - \* Da  $u \in L(\alpha)$  und  $v \in L(\beta)$ , ist  $w = uv \in L(\alpha\beta)$ 
      - ☞ Semantik REs: Konkatination
    - ➔  $w \in L(\alpha\beta + \alpha\gamma)$ 
      - ☞ Semantik REs: Auswahl
  - 2. Fall:  $v \in L(\gamma)$ 
    - ➔  $w = uv \in L(\alpha\gamma)$ 
      - ☞ Semantik REs: Konkatination
    - ➔  $w \in L(\alpha\beta + \alpha\gamma)$ 
      - ☞ Semantik REs: Auswahl
- Der Beweis von  $L(\alpha\beta + \alpha\gamma) \subseteq L(\alpha(\beta + \gamma))$  ist analog

# Reguläre Ausdrücke: Äquivalenzen (4/4)

- Wie lässt sich beweisen, dass eine vermutete oder behauptete Äquivalenz **nicht** gilt?
- Wir betrachten als Beispiel die **nicht gültige** Äquivalenz
$$\alpha + (\beta\gamma) \equiv (\alpha + \beta)(\alpha + \gamma)$$
- Wir formulieren zunächst eine mathematische Aussage, die die Ungültigkeit dieser Äquivalenz ausdrückt

## Proposition 1.2

- Es gibt reguläre Ausdrücke  $\alpha, \beta, \gamma$  mit
$$L(\alpha + (\beta\gamma)) \neq L((\alpha + \beta)(\alpha + \gamma))$$
- Diese Proposition lässt sich sehr einfach durch Angabe eines **Beispiels** beweisen
  - ✎ Das Beispiel belegt die Gültigkeit der Proposition, für die behauptete Äquivalenz ist es ein **Gegenbeispiel**

## Beweis

- Wir wählen:
  - $\alpha = a, \beta = b, \gamma = c$
- Dann gilt:
  - $L(\alpha + (\beta\gamma)) = \{a, bc\}$
  - $L((\alpha + \beta)(\alpha + \gamma)) = \{aa, ba, ac, bc\}$
- Insbesondere:
  - $aa \notin L(\alpha + (\beta\gamma))$
  - $aa \in L((\alpha + \beta)(\alpha + \gamma))$
- Also gilt im allgemeinen **nicht** die Aussage
$$L((\alpha + \beta)(\alpha + \gamma)) \subseteq L(\alpha + (\beta\gamma))$$
und deshalb **auch nicht**
$$L(\alpha + (\beta\gamma)) = L((\alpha + \beta)(\alpha + \gamma))$$
- ✎ Bei Beweisen durch Gegenbeispiel sollte das Gegenbeispiel jeweils so konkret wie möglich gewählt werden

# Reguläre Ausdrücke: Zusammenfassung

## Themen dieses Kapitels

- Warum reguläre Ausdrücke?
- Begriffe: Alphabete, Wörter, Sprachen
- Syntax regulärer Ausdrücke
- Semantik regulärer Ausdrücke
- Präzedenzregeln
- Abkürzende Schreibweisen für reguläre Ausdrücke
- Äquivalenzregeln für reguläre Ausdrücke
- Beweismethoden:
  - Mengengleichheit
  - Beweis durch Gegenbeispiel

## Kapitelfazit

- Was haben wir bisher erreicht?
  - Wir können syntaktisch korrekte e-Mail-Adressen jetzt sauber spezifizieren
  - Wir können sie aber noch nicht automatisch in Algorithmen/Programme übersetzen
- Damit werden wir uns im nächsten Kapitel beschäftigen

# Erläuterungen

## Bemerkung 1.1

- Zu definierende **Begriffe** sind durch Unterstreichung, Fettdruck und dunkles Türkis zu erkennen
- Auch Schreibweisen wie  $u^n$  werden so gekennzeichnet
- Später werden diese Schreibweisen dann aber ohne Unterstreichung etc. verwendet

## Bemerkung 1.3



### Zu beachten:

- $\mathbb{N}$  bezeichnet in dieser Vorlesung die Menge der natürlichen Zahlen **ohne 0**,
- $\mathbb{N}_0$  bezeichnet die natürlichen Zahlen mit 0
- $L_1^*$  enthält also immer auch den Leerstring

## Bemerkung 1.2

- Die Definition der Syntax regulärer Ausdrücke ist eine induktive Definition (Näheres dazu in der nächsten Vorlesung)
- Die Menge aller regulären Ausdrücke über  $\Sigma$  ist selbst auch wieder eine Sprache
  - Alphabet:  $\Sigma \cup \{\emptyset, \epsilon, +, *, ), (\}$

# Grundbegriffe der Theoretischen Informatik

Sommersemester 2016 - Thomas Schwentick

Teil A: Reguläre Sprachen

2: Endliche Automaten

Version von: 21. April 2016 (14:25)

# Testprogramme für reguläre Sprachen

- In der letzten Stunde hatten wir einen (erweiterten) regulären Ausdruck für e-Mail-Adressen konstruiert:  
$$([a-zA-Z][a-zA-Z0-9\-\_]*\.)^*[a-zA-Z][a-zA-Z0-9\-\_]*@([a-zA-Z][a-zA-Z0-9\-\_]*\.)^+[a-zA-Z]\{2,4\}$$
- Heute beschäftigen wir uns mit Testalgorithmen für reguläre Sprachen
  - Also: Algorithmen, die für eine gegebene Sprache  $L$  testen, ob ein Eingabewort  $w$  in  $L$  ist
- Im ersten Teil werden wir aus einem in Pseudocode geschriebenen Testalgorithmus ein Berechnungsmodell für reguläre Sprachen abstrahieren: **endliche Automaten**
- Im zweiten Teil werden wir eine flexiblere Variante endlicher Automaten betrachten, die die automatische Umwandlung von REs in endliche Automaten erleichtern wird

# Ein „Programm“ zur Erkennung von e-Mail-Adressen

## Alg. 2.1: Erkennung von e-Mail-Adressen


```
InLabel := false; Error := false;
Local := true; Letters := true
chars := 0; labs := 0;
while NOT EOF() do
    z := NextSymbol
    case z IN
        [a - zA - Z]:
            InLabel := true; chars := chars + 1
        [0-9\-\_]:
            if InLabel then
                chars := chars + 1; Letters := false
            else
                Error := true
        [.]:
            if InLabel then
                chars := 0; labs := labs + 1
                InLabel := false; Letters := false
            else
                Error := true
```



## Alg. 2.1 (Fortsetzung)

```
while ... do {Fortsetzung while-Schleife}
    case ...
        [@]:
            if Local then
                chars := 0; labs := 0;
                InLabel := false;
                Local := false; Letters := true
            else
                Error := true
    if NOT Error AND Letters AND NOT Local
    AND labs ≥ 1 AND 2 ≤ chars ≤ 4 then
        Print(„OK“)
    else
        Print(„Nicht OK“)
```

- Das ist „einfach so runter programmiert“
  - Ist das Programm korrekt? ☐
- Gibt es einen Weg, um einen regulären Ausdruck **automatisch** und **zuverlässig** in ein Programm zu übersetzen?

# Vom Programm zum Automaten

- Was ist besonders an Algorithmus 2.1?
  - Die Eingabe wird **Zeichen für Zeichen** gelesen und verarbeitet
  - Es gibt nur **begrenzt viele** (relevante) Kombinationen von **Werten** der Programm-Variablen:
    - \* InLabel  $\in \{\text{true}, \text{false}\}$
    - \* Error  $\in \{\text{true}, \text{false}\}$
    - \* Local  $\in \{\text{true}, \text{false}\}$
    - \* Letters  $\in \{\text{true}, \text{false}\}$
    - \* chars  $\in \{0, 1, 2, 3, 4, „\geq 5“\}$
    - \* labs  $\in \{0, „\geq 1“\}$
- Wir nennen jede mögliche **Kombination von Variablenwerten** einen **Zustand** 
- Beim Lesen eines Zeichens geht das Programm also von einem Zustand in einen anderen Zustand über

- Ein solches System aus (endlich vielen) Zuständen und Zustandsübergängen heißt **endliches Transitionssystem** oder **endlicher Automat**
  -  Die feinen Unterschiede zwischen diesen Begriffen werden wir später betrachten
- Wieviele Zustände hat der Automat für Algorithmus 2.1?
  - Bei naiver Vorgehensweise: 
- Wir werden sehen:  
das geht erheblich besser
- Jetzt betrachten wir aber zunächst mal ein kleineres Beispiel eines Automaten

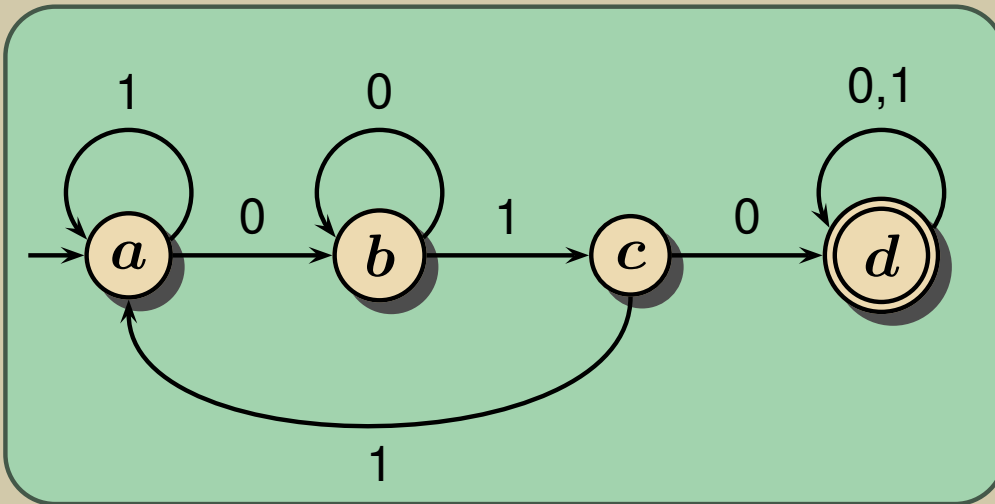


# Inhalt

- ▷ **2.1 Endliche Automaten: Definition und Konstruktion**
- 2.2 Nichtdeterministische endliche Automaten
- 2.3 Von regulären Ausdrücken zu nichtdeterministischen Automaten

# Endliche Automaten: Beispiel


## Beispiel



- Eingabe: 001101010
- Eingabe: 101100

- Der String 001101010 wird von dem Automaten akzeptiert
- Der String 101100 wird von dem Automaten nicht akzeptiert

- Dieser Automat akzeptiert einen String genau dann, wenn er den Teilstring 010 enthält, also wenn er in der Sprache  $L((0 + 1)^*010(0 + 1)^*)$  ist
- Wir sagen, dass der Automat die Sprache  $L((0 + 1)^*010(0 + 1)^*)$  **entscheidet**

-  Die graphische Darstellung von Automaten ist zwar anschaulich, aber wir benötigen präzise Definitionen, die unzweideutig festlegen,
- was ein endlicher Automat ist, und
  - wie ein endlicher Automat „funktioniert“

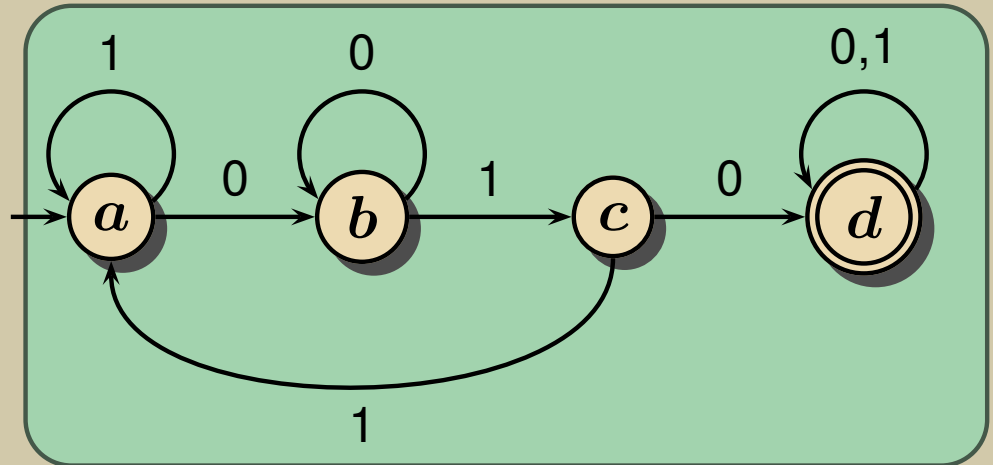
# Endliche Automaten: Definition

## Definition: Syntax endlicher Automaten 2.1

- Ein endlicher Automat  $\mathcal{A}$  besteht aus
  - einer endlichen Menge  $Q$  von **Zuständen**,
  - einem **Eingabealphabet**  $\Sigma$ ,
  - einer **Transitionsfunktion**  
 $\delta : Q \times \Sigma \rightarrow Q$ ,
  - einem **Startzustand**  $s \in Q$ , und
  - einer Menge  $F \subseteq Q$  **akzeptierenden Zuständen**
- Wir schreiben  $\mathcal{A} = (Q, \Sigma, \delta, s, F)$

- In der graphischen Darstellung von endlichen Automaten
  - wird der Startzustand durch eine „aus dem nichts kommende“ Kante markiert,
  - werden die akzeptierenden Zustände durch einen doppelten Rand markiert

## Beispiel: der 010-Automat formal



- $\mathcal{A}_{010} = (Q_{010}, \{0, 1\}, \delta_{010}, s_{010}, F_{010})$
- $Q_{010} = \{a, b, c, d\}$
- $s_{010} = a$
- $F_{010} = \{d\}$
- $\delta_{010}(a, 0) = b, \delta_{010}(a, 1) = a, \dots$



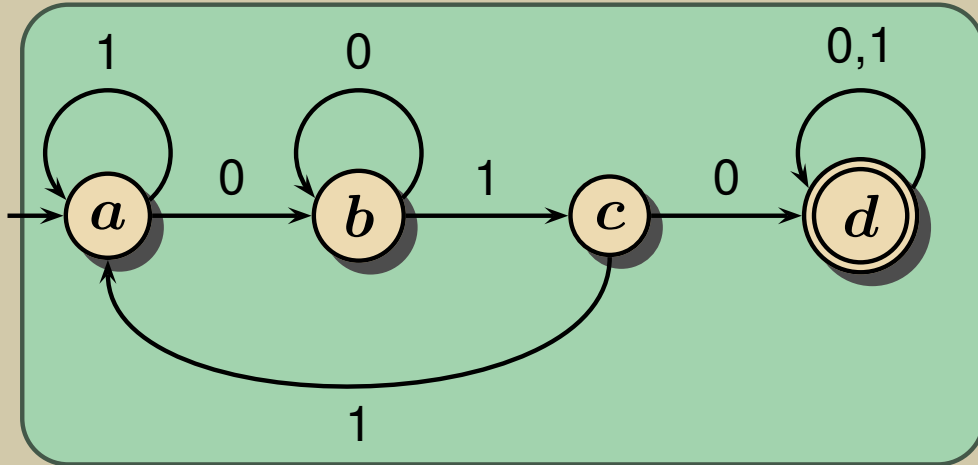
Abkürzung:

**DFA** für **deterministic finite automaton**

- Plural: **DFAs** für **deterministic finite automata**

# Endliche Automaten: Semantik (1/2)

## Beispiel



- **Informelle Semantik** endlicher Automaten:
  - Der Automat liest die Eingabe Zeichen für Zeichen, beginnend im Startzustand
  - Er geht dabei jeweils gemäß der Transitionsfunktion  $\delta$  in einen Zustand über
  - Er **akzeptiert** die Eingabe, falls er am Ende in einem Zustand aus  $F$  ist
- Um Aussagen über Automaten **beweisen** zu können, benötigen wir wieder eine **formale Semantik**

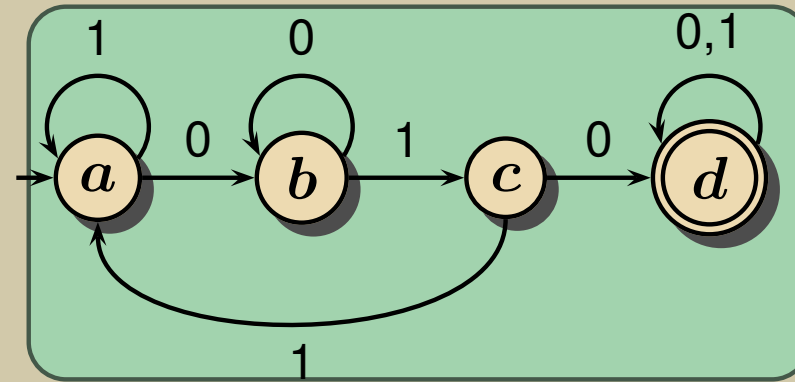
- Dazu definieren wir die **erweiterte Transitionsfunktion**  $\delta^*$ :
  - $\delta^*(q, w)$  soll der Zustand sein, den der Automat annimmt, wenn er vom Zustand  $q$  aus  $w$  liest

## Definition: Semantik endlicher Automaten

- Die **erweiterte Transitionsfunktion**  $\delta^* : Q \times \Sigma^* \rightarrow Q$  eines Automaten  $\mathcal{A} = (Q, \Sigma, \delta, s, F)$  ist wie folgt induktiv definiert:
  - $\delta^*(q, \epsilon) \stackrel{\text{def}}{=} q$ ,
  - $\delta^*(q, u\sigma) \stackrel{\text{def}}{=} \delta(\delta^*(q, u), \sigma)$   
für  $u \in \Sigma^*, \sigma \in \Sigma$
- $\mathcal{A}$  akzeptiert  $w$   $\stackrel{\text{def}}{\iff} \delta^*(s, w) \in F$
- Von  $\mathcal{A}$  entschiedene Sprache:
$$L(\mathcal{A}) \stackrel{\text{def}}{=} \{w \in \Sigma^* \mid \mathcal{A} \text{ akzeptiert } w\}$$

# Endliche Automaten: Semantik (2/2)

## Beispiel



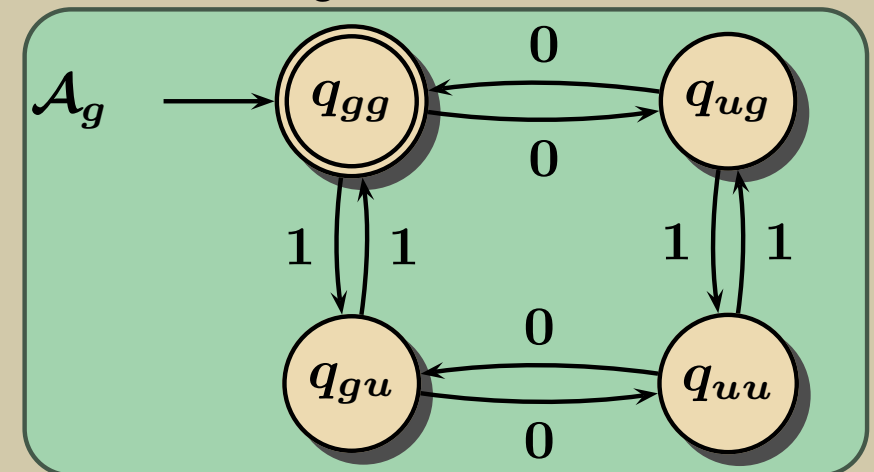
$$\begin{aligned} & \delta^*(a, 0110101) \\ &= \delta(\delta^*(a, 011010), 1) \\ &= \delta(\delta(\delta^*(a, 01101), 0), 1) \\ &= \delta(\delta(\delta(\delta^*(a, 0110), 1), 0), 1) \\ &= \delta(\delta(\delta(\delta(\delta^*(a, 011), 0), 1), 0), 1) \\ &= \delta(\delta(\delta(\delta(\delta(\delta^*(a, 01), 1), 0), 1), 0), 1) \\ &= \delta(\delta(\delta(\delta(\delta(\delta(\delta^*(a, 0), 1), 1), 0), 1), 0), 1) \\ &= \delta(\delta(\delta(\delta(\delta(\delta(\delta^*(a, \epsilon), 0), 1), 1), 0), 1), 0), 1) \\ &= \delta(\delta(\delta(\delta(\delta(\delta(a, 0), 1), 1), 0), 1), 0), 1) \\ &= \delta(\delta(\delta(\delta(b, 1), 1), 0), 1), 0), 1) \\ &= \delta(\delta(\delta(\delta(c, 1), 0), 1), 0), 1) \\ &= \delta(\delta(\delta(a, 0), 1), 0), 1) \\ &= \delta(\delta(b, 1), 0), 1) \\ &= \delta(c, 0), 1) \\ &= \delta(d, 1) \\ &= d \end{aligned}$$

# Konstruktion endlicher Automaten: Beispiel

- Wie lässt sich zu einer gegebenen Sprache ein Automat konstruieren?
- Es sind folgende Fragen zu klären:
  - Welche Informationen muss der Automat sich merken (Zustände)?
    - \* Welche Bedeutung haben dann die einzelnen Zustände?
  - Wie ändern sich diese Informationen durch das Lesen eines Zeichens (Transitionen)?
  - Wie ist die Initialisierung? (Startzustand)
  - Wie sollen die gemerkten Informationen das Akzeptieren der Eingabe beeinflussen?

## Beispiel

- Konstruktion eines DFA für die Menge  $L_g$  aller Zeichenketten über  $\{0, 1\}$  mit gerade vielen 0 und 1
- Informationen, die der Automat sich merkt:
  - Ist die Anzahl der bisher gelesenen 0-Zeichen gerade oder ungerade
  - Ist die Anzahl der bisher gelesenen 1-Zeichen gerade oder ungerade
- Das ergibt vier Zustände:  $q_{gg}$ ,  $q_{gu}$ ,  $q_{ug}$ ,  $q_{uu}$  mit offensichtlicher Bedeutung:



- Wie ändern sich die Zustände? Klar!
- Welches ist der Startzustand?  $q_{gg}$
- Welche Zustände sollen Akzeptieren bewirken?  $q_{gg}$

## Zwischenfrage

PINGO-Frage: pingo.upb.de

Wie viele Zustände benötigt ein DFA, der genau die Strings über  $\{a, b\}$  akzeptiert, die mit  $a$  beginnen und gerade viele  $b$ 's haben?

- (A) 3
- (B) 4
- (C) 5
- (D) 6

# Inhalt

2.1 Endliche Automaten: Definition und Konstruktion

▷ **2.2 Nichtdeterministische endliche Automaten**

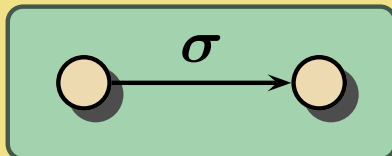
2.3 Von regulären Ausdrücken zu nichtdeterministischen Automaten



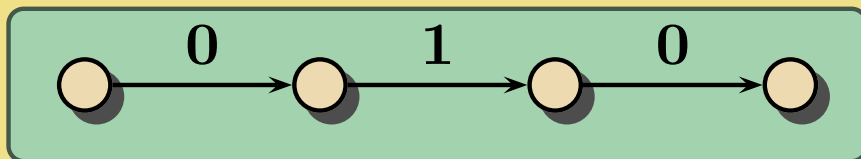
# Von REs zu DFAs: Grundideen

- **Unser Ziel:** Wir suchen eine Methode zur Umwandlung von REs in Automaten
- Die Grundideen dafür sind sehr einfach und werden hier zunächst anhand von Automatenfragmenten vorgestellt
- Im Prinzip sollen REs induktiv in DFAs umgewandelt werden

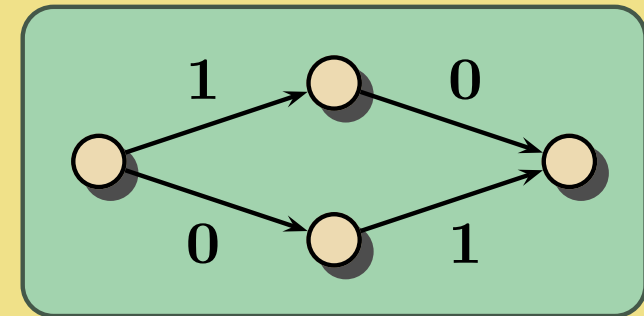
- Ein Zeichen  $\sigma$  eines regulären Ausdrucks wird in eine einzelne Transition übersetzt:



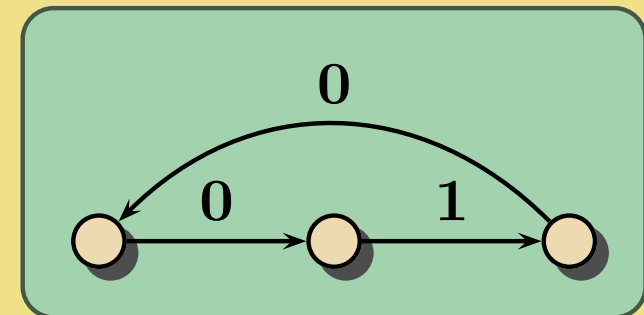
- Die **Konkatenation** von Zeichen entspricht der Hintereinanderausführung von Transitionen, z.B. für 010:



- Die **Auswahl** in REs entspricht einer Verzweigung im Automaten, z.B. ergibt sich für  $10 + 01$ :



- Die **Iteration** entspricht einer Schleife im Automaten: z.B. ergibt sich für  $(010)^+$ :



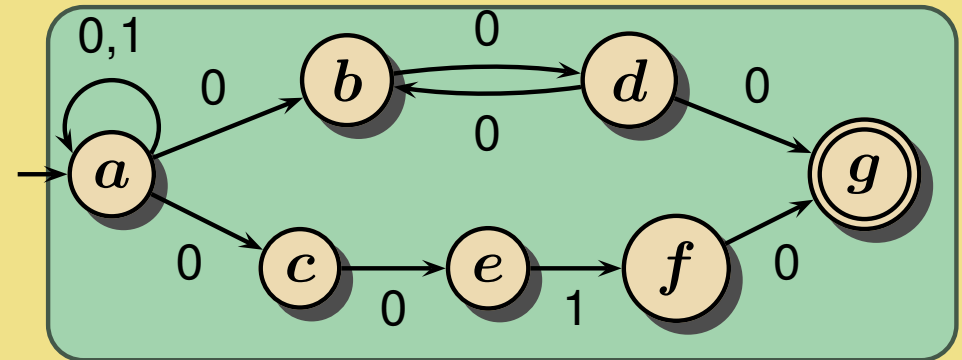
# Von REs zu DFAs: Schwierigkeiten und Lösungsansatz

## Schwierigkeiten

- Die Umsetzung der Auswahl  $10 + 01$  ist einfach, da die beiden Teilausdrücke mit verschiedenen Zeichen anfangen
- Aber wie soll eine **Auswahl** wie  $((00)^+ + 001)$  umgesetzt werden?
  - Soll eine gelesene 0 als erstes Zeichen von  $(00)^+$  oder als erstes Zeichen von  $001$  angesehen werden? **2.2**
- Und wie soll die **Konkatenation einer Iteration** mit einem einzelnen Zeichen wie  $(0 + 1)^*0$  umgesetzt werden?
  - Soll eine 0 als Zeichen von  $(0 + 1)^*$  oder als abschließende 0 betrachtet werden?
- Beide Schwierigkeiten kommen in dem Ausdruck  $(0 + 1)^*((00)^+ + 001)0$  kombiniert vor

## Lösungsansatz

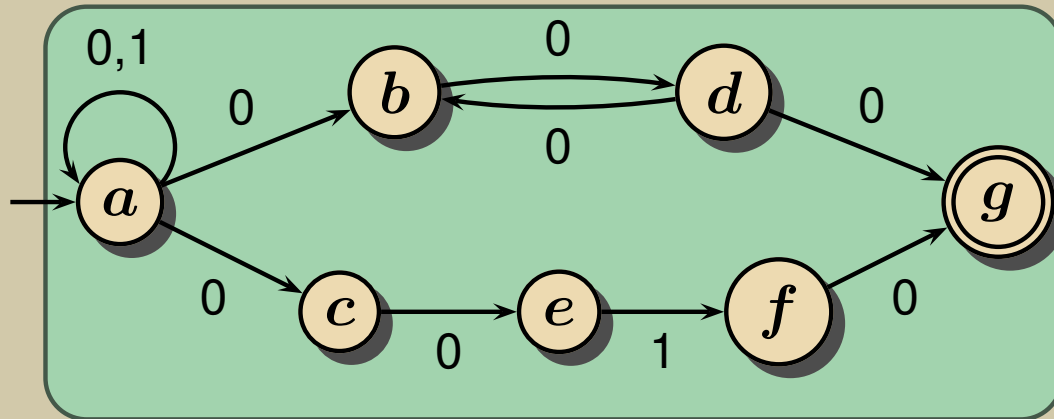
- Wir erweitern unser Automatenmodell und erlauben, dass ein Automat in einem Zustand mehrere Transitionen für das selbe gelesene Zeichen hat
- Der Ausdruck  $(0 + 1)^*((00)^+ + 001)0$  lässt sich dann übersetzen in:



- Ein solcher Automat kann für dieselbe Eingabe verschiedene Berechnungen haben
- Wie soll dann definiert sein, dass er eine Eingabe akzeptiert?

# Nichtdeterministische Endliche Automaten

## Beispiel



0010010

- Wir sagen, „**der Automat akzeptiert ein Wort  $w$** “, falls eine Berechnung **existiert**, in der  $w$  vollständig gelesen und dann ein akzeptierender Zustand erreicht wird  
→ **nichtdeterministisches Akzeptieren**

## Definition

- Ein nichtdeterministischer endlicher Automat  $\mathcal{A} = (Q, \Sigma, \delta, s, F)$  besteht aus
  - einer Zustandsmenge  $Q$ ,
  - einem Eingabealphabet  $\Sigma$ ,
  - einem Anfangszustand  $s \in Q$ ,
  - einer Menge  $F$  akzeptierender Zustände,
  - sowie einer **Transitionsrelation**

$$\delta \subseteq Q \times \Sigma \times Q$$

## Beispiel (Forts.)

- Im Beispiel enthält  $\delta$  unter anderem:
  - $(d, 0, b)$
  - $(d, 0, g)$
  - $(e, 1, f)$
  - $(a, 1, a)$aber kein Tupel der Art  $(g, \sigma, p)$  für ein  $\sigma \in \{0, 1\}$  und  $p \in Q$

# Nichtdeterministische Automaten: Notation

- **NFA**: Abkürzung für den Begriff „nichtdeterministischer Automat“  
(**non-deterministic finite automaton**)
- Statt  $(p, \sigma, q) \in \delta$  verwenden wir häufig die intuitivere Notation  $p \xrightarrow{\sigma} q$ 
  - Um zu betonen, dass es sich um eine Transition im Automaten  $\mathcal{A}$  handelt, schreiben wir manchmal auch  $p \xrightarrow{\sigma, \mathcal{A}} q$
- Zur Vorbereitung für die Definition der Semantik von NFAs formalisieren wir zunächst den informellen Begriff „Berechnung“ durch den formalen Begriff **Lauf**


# NFAs: Semantik (1/2)

## Definition: Lauf eines NFA

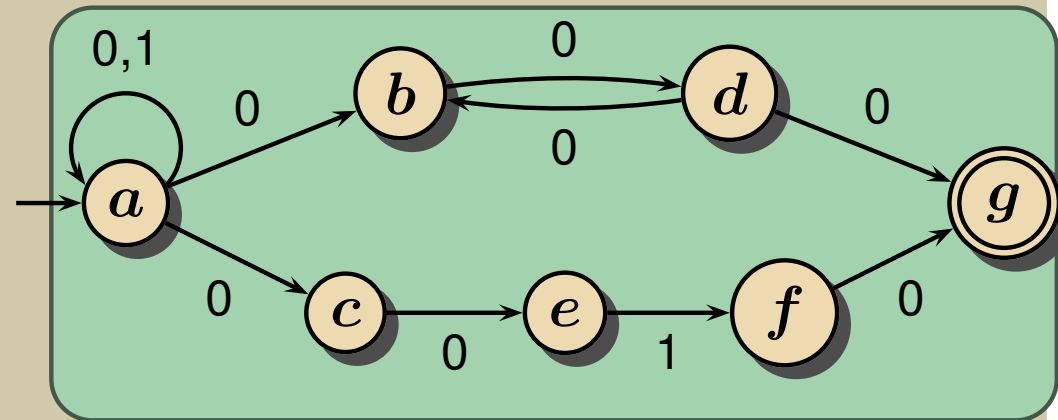
- Ein **Lauf**  $\rho$  eines NFAs  $\mathcal{A} = (Q, \Sigma, \delta, s, F)$  ist eine Folge der Art  $q_0, \sigma_1, q_1, \dots, \sigma_n, q_n$ , wobei
  - für alle  $i \in \{0, \dots, n\}$ :  $q_i \in Q$ ,
  - für alle  $i \in \{1, \dots, n\}$ :  $\sigma_i \in \Sigma$ , und
  - für alle  $i \in \{1, \dots, n\}$ :  $q_{i-1} \xrightarrow{\sigma_i} q_i$
- Wir sagen:  $\rho$  ist ein Lauf von  $q_0$  nach  $q_n$ , der den String  $w = \sigma_1 \dots \sigma_n$  liest

- Statt  $\rho = q_0, \sigma_1, q_1, \dots, \sigma_n, q_n$  schreiben wir meist  

$$\rho = q_0 \xrightarrow{\sigma_1} q_1 \dots q_{n-1} \xrightarrow{\sigma_n} q_n$$
- Abkürzende Schreibweise:
  - $\underline{p \xrightarrow{w} q} \stackrel{\text{def}}{\iff}$  es gibt einen Lauf von  $p$  nach  $q$ , der den String  $w$  liest

 Spezialfall  $n = 0$ :  $q$  ist ein Lauf für das leere Wort

## Beispiel



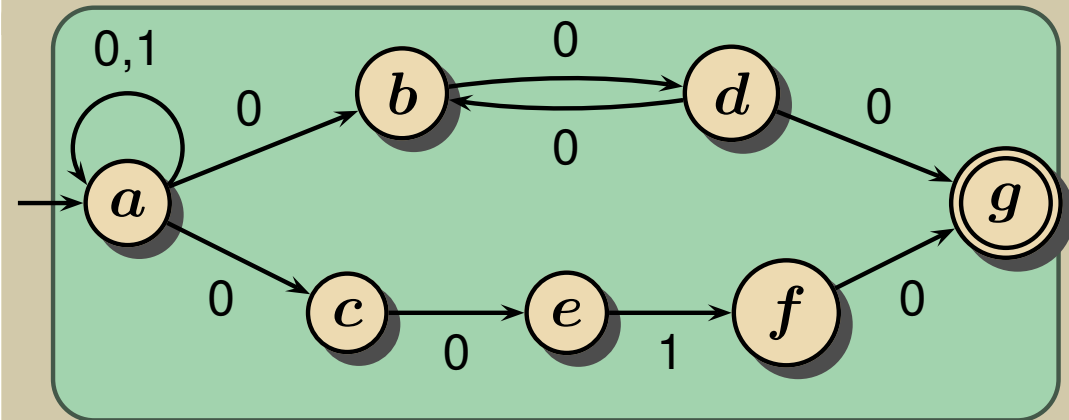
- $a, 0, b, 0, d, 0, b, 0, d, 0, g$  ist ein Lauf von  $a$  nach  $g$ , der das Wort **000000** liest
- $a, 1, a, 1, b, 0, d$  ist kein Lauf
- $b, 0, d, 0, g$  ist ein Lauf von  $b$  nach  $g$ , der das Wort **00** liest
- In der anderen Notation:
  - $a \xrightarrow{0} b \xrightarrow{0} d \xrightarrow{0} b \xrightarrow{0} d \xrightarrow{0} g$
  - $a \xrightarrow{000000} g$
  - $b \xrightarrow{0} d \xrightarrow{0} g$

## NFAs: Semantik (2/2)

### Definition: Semantik von NFAs

- Sei  $\mathcal{A} = (Q, \Sigma, \delta, s, F)$  ein NFA
- Ein Lauf von  $s$  zu einem Zustand aus  $F$  heißt akzeptierend
- $L(\mathcal{A})$  ist die Menge aller Strings, für die es einen akzeptierenden Lauf von  $\mathcal{A}$  gibt:  
$$L(\mathcal{A}) \stackrel{\text{def}}{=} \{w \in \Sigma^* \mid s \xrightarrow{w} q, q \in F\}$$

### Beispiel



- Sei  $\mathcal{A}$  der obige NFA
- $a \xrightarrow{0} a \xrightarrow{0} b \xrightarrow{0} d \xrightarrow{0} b \xrightarrow{0} d$  ist ein nicht akzeptierender Lauf für 00000
- $a \xrightarrow{0} b \xrightarrow{0} d \xrightarrow{0} b \xrightarrow{0} d \xrightarrow{0} g$  ist ein akzeptierender Lauf für 00000
- Da es einen akzeptierenden Lauf für 00000 gibt, ist  $00000 \in L(\mathcal{A})$
- Für 01001 gibt es keinen akzeptierenden Lauf, deshalb ist  $01001 \notin L(\mathcal{A})$

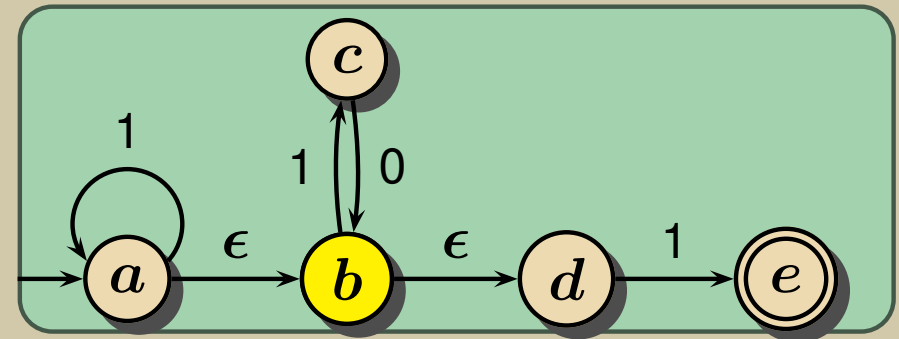
# Bemerkungen zu nichtdeterministischen Automaten

- Nichtdeterminismus ist zunächst gewöhnungsbedürftig und hat für viele etwas „Beunruhigendes“
- Zur Beruhigung:
  - Nichtdeterministische Automaten sind für uns zunächst nur Mittel zum Zweck:
    - \* Sie stellen einen Zwischenschritt zwischen regulären Ausdrücken und (deterministischen) endlichen Automaten dar
    - \* Die am Ende resultierenden Testprogramme sind also deterministisch
  - Nichtdeterminismus ist zur Modellierung von Systemen allerdings häufig sehr hilfreich

# NFA mit $\epsilon$ -Transitionen

- Um den Übergang von regulären Ausdrücken (REs) zu nichtdeterministischen endlichen Automaten (NFAs) zu erleichtern, betrachten wir die Erweiterung von NFAs um  **$\epsilon$ -Transitionen**
- Damit machen wir NFAs „noch nichtdeterministischer“:
  - Sie bekommen die Möglichkeit den Zustand zu wechseln, ohne ein Zeichen zu lesen
- Warum ist das praktisch?
  - Damit lassen sich Konkatenationen und Wiederholungen übersichtlicher im NFA repräsentieren
- Als Beispiel betrachten wir einen Automaten für den RE  $1^*(10)^*1$

## Beispiel

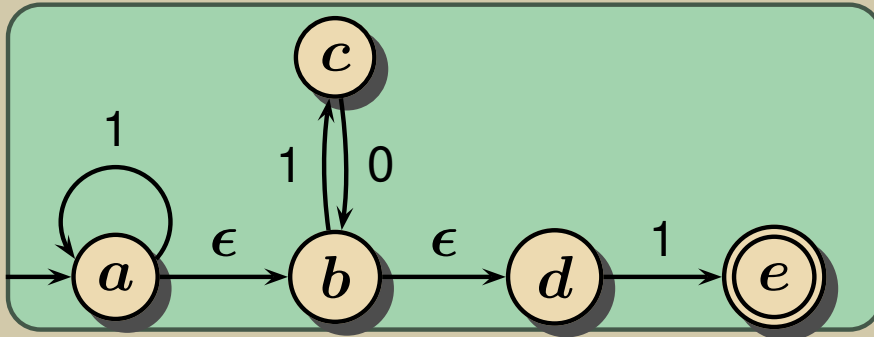


1|10101



# NFA mit $\epsilon$ -Transitionen: formal

## Beispiel



## Definition

- Ein nichtdeterministischer endlicher Automat  $\mathcal{A} = (Q, \Sigma, \delta, s, F)$  mit  $\epsilon$ -Transitionen ( $\epsilon$ -NFA) besteht aus

- einer Zustandsmenge  $Q$ ,
- einem Eingabealphabet  $\Sigma$ ,
- einem Anfangszustand  $s \in Q$ ,
- einer Menge  $F$  akzeptierender Zustände,
- sowie einer **Transitionsrelation**

$$\delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times Q$$

- Im Beispiel gilt  $a \xrightarrow{\epsilon} b$

- Wie lässt sich die Semantik von  $\epsilon$ -NFAs definieren?

- Im Prinzip wie zuvor: aber Läufe dürfen jetzt auch Schritte der Art  $p \xrightarrow{\epsilon} q$  enthalten, falls  $(p, \epsilon, q) \in \delta$

- Wir überladen unsere Notation etwas und schreiben  $p \xrightarrow{\epsilon} q$  auch, wenn es einen Lauf von  $p$  nach  $q$  gibt, der nur  $\epsilon$ -Transitionen verwendet

- Bei einem  $\epsilon$ -NFA bedeutet die Schreibweise  $p \xrightarrow{w} q$ , dass es einen Lauf von  $p$  nach  $q$  gibt, der  $w$  liest und zwischendurch möglicherweise auch noch  $\epsilon$ -Transitionen verwendet

# Inhalt

2.1 Endliche Automaten: Definition und Konstruktion

2.2 Nichtdeterministische endliche Automaten

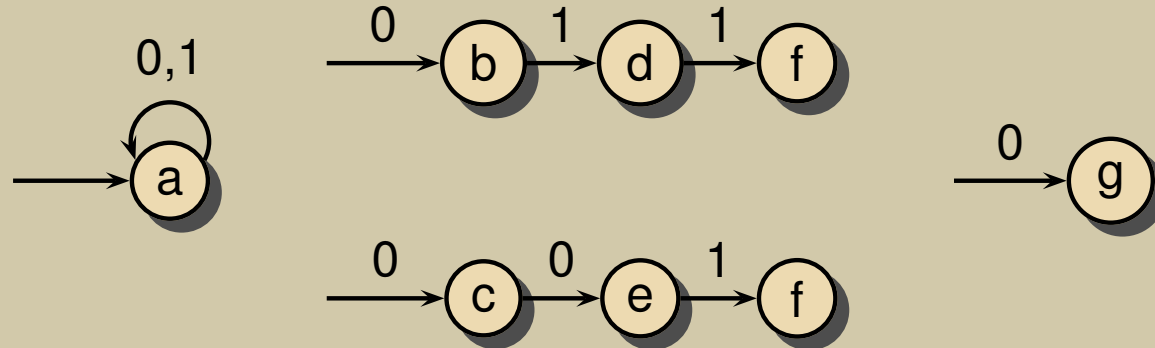
▷ **2.3 Von regulären Ausdrücken zu nichtdeterministischen Automaten**

## Umwandlung RE -> NFA: Beispiel

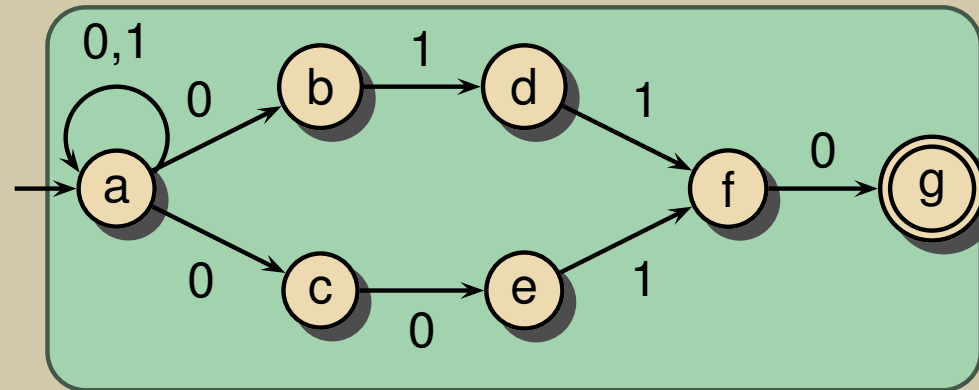
- Die Umwandlung von REs in NFAs lässt sich relativ einfach **induktiv** umsetzen (im Beispiel sogar ohne  $\epsilon$ -Transitionen)

### Beispiel

- Für  $(0 + 1)^*(011 + 001)0$  ergeben sich Teilautomaten:



- ... aus denen sich der Gesamtautomat zusammensetzen lässt:



# Vom RE zum $\epsilon$ -NFA (1/2)

## Proposition 2.2

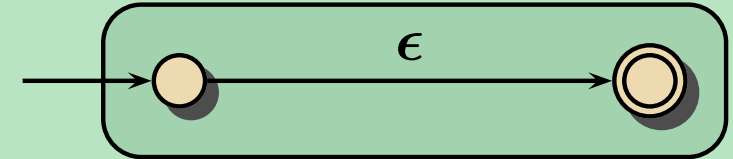
- Zu jedem RE  $\alpha$  gibt es einen  $\epsilon$ -NFA  $\mathcal{A}$ , so dass  $L(\alpha) = L(\mathcal{A})$  gilt

## Beweisskizze

- Wir zeigen etwas mehr:  $\mathcal{A}$  kann so konstruiert werden, dass
  - in den Startzustand keine Transitionen hineinführen, und
  - aus dem eindeutigen akzeptierenden Zustand keine Transitionen herausführen
- Also:  $\mathcal{A} = (Q, \Sigma, \delta, s, F)$  mit
  - wenn  $p \xrightarrow{\sigma} q$  oder  $p \xrightarrow{\epsilon} q$  dann  $p \notin F$  und  $q \neq s$
  - $|F| = 1$
- Wir konstruieren  $\mathcal{A}$  durch Induktion nach der Struktur von  $\alpha$
- Den Beweis, dass die Konstruktion korrekt ist, ersparen wir uns: er wird mit struktureller Induktion geführt

## Beweisskizze (Forts.)

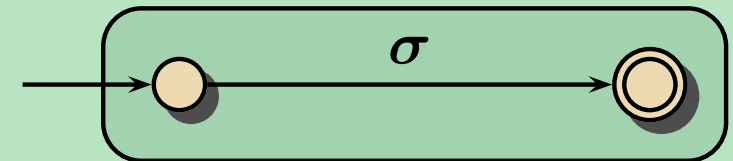
- $\alpha = \epsilon$ :



- $\alpha = \emptyset$ :



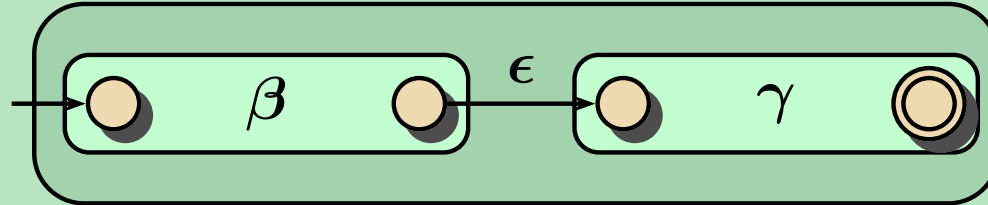
- $\alpha = \sigma$ :



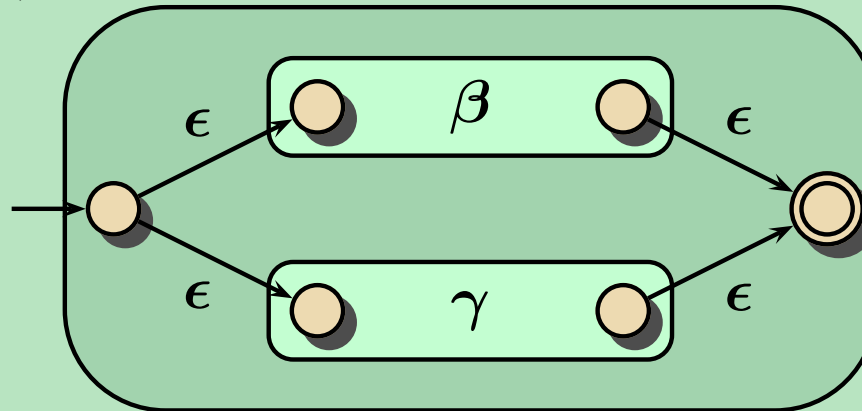
## Vom RE zum $\epsilon$ -NFA (2/2)

### Beweisskizze (Forts.)

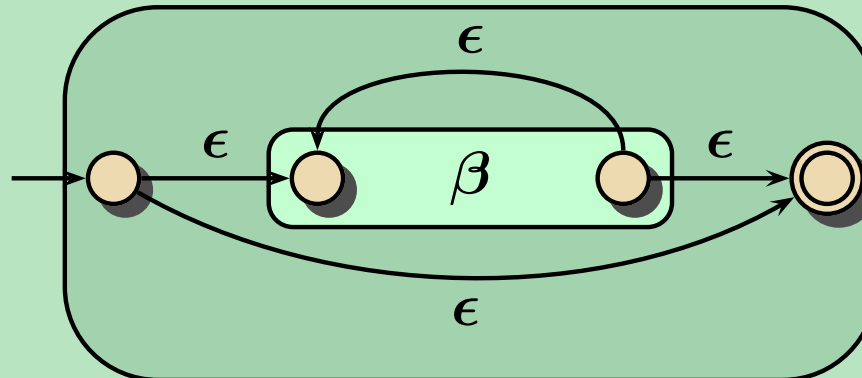
- $\alpha = \beta\gamma$ :



- $\alpha = \beta + \gamma$ :



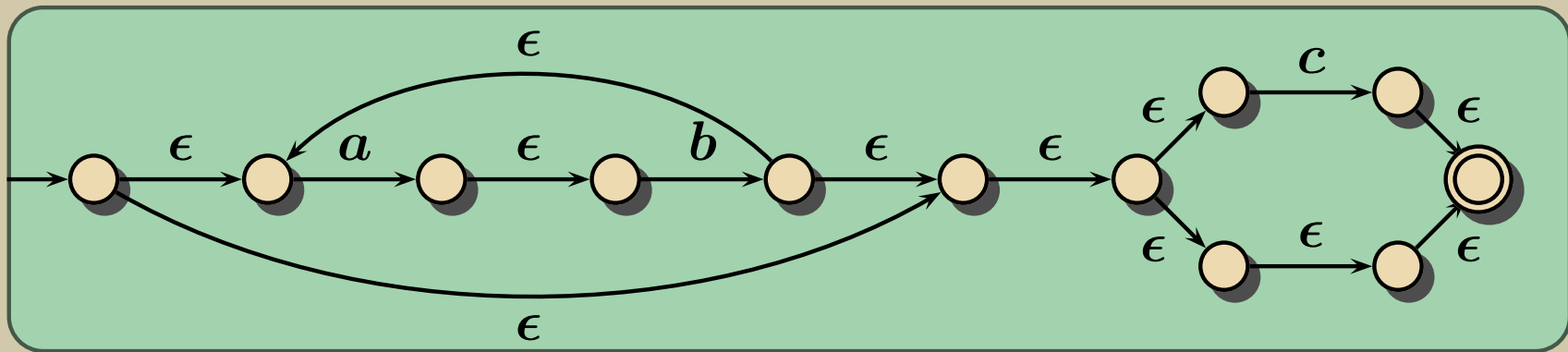
- $\alpha = \beta^*$ :



# Vom RE zum $\epsilon$ -NFA: Beispiel

## Beispiel

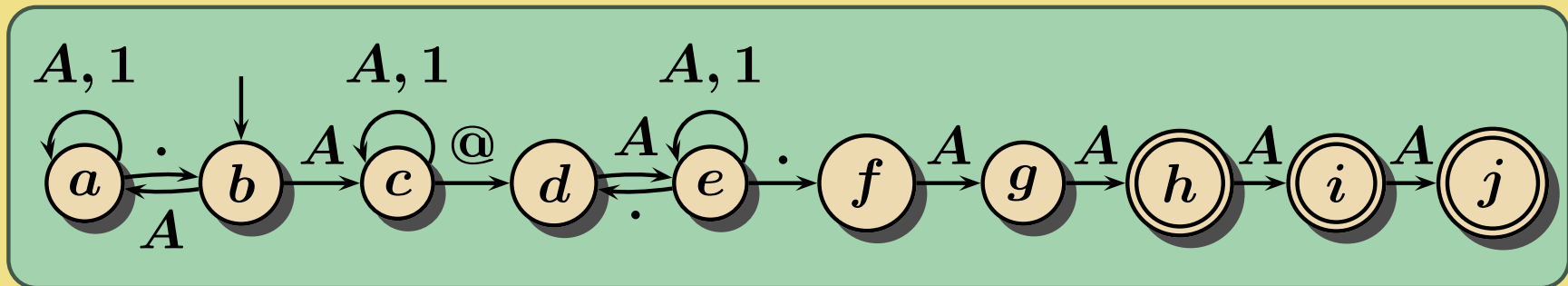
- Bei der Umwandlung des regulären Ausdrucks  $(ab)^*(c+\epsilon)$  nach der beschriebenen Methode erhalten wir:



# E-Mail-Adressen: Vom RE zum NFA

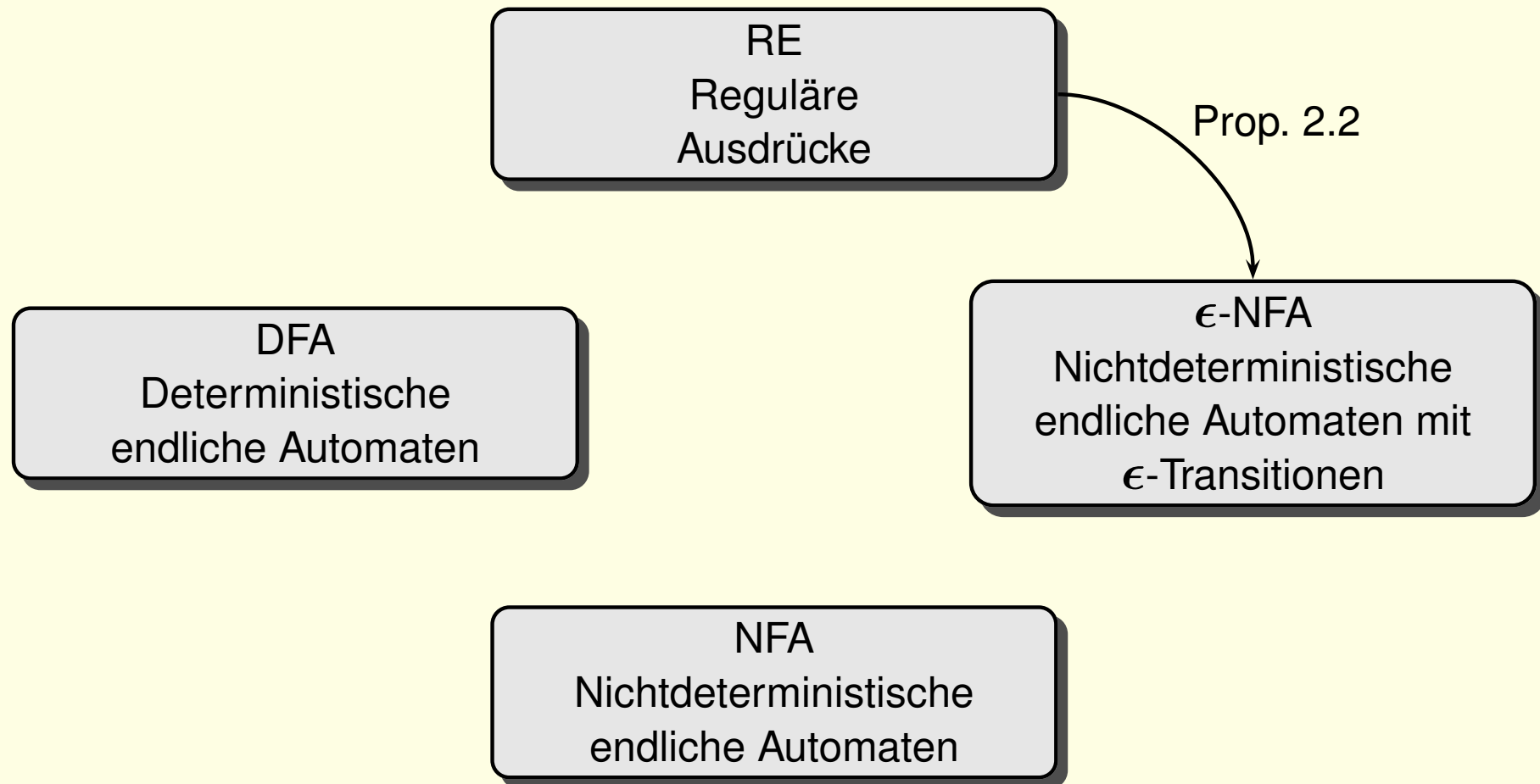
- Jetzt können wir also einen RE automatisch in einen äquivalenten  $\epsilon$ -NFA umwandeln
- Wir betrachten das Beispiel des e-Mail-Ausdrucks:  
$$([a-zA-Z][a-zA-Z0-9\-\_]*\cdot)^*[a-zA-Z][a-zA-Z0-9\-\_]*@([a-zA-Z][a-zA-Z0-9\-\_]*\cdot)^+[a-zA-Z]\{2,4\}$$
- Statt des automatisch erzeugten  $\epsilon$ -NFA betrachten wir aus Platzgründen allerdings einen etwas „optimierten“ NFA
- Wir verwenden im NFA einige Abkürzungen:
  - $A$  steht für  $a, \dots, z, A, \dots, Z$
  - $1$  steht für  $0, \dots, 9$  sowie  $\backslash -$  und  $\backslash \_$

- NFA:



- Jetzt fehlt nur noch der Schritt zum DFA...

# Die bisher betrachteten Modelle





# Erläuterungen

## Bemerkung 2.1

- Die akzeptierenden Zustände von DFAs werden häufig auch „Endzustände“ genannt
- Diese Begriffsbildung verwenden wir in dieser Veranstaltung nicht
- Denn:
  - DFAs halten an, wenn sie das Eingabewort gelesen haben, unabhängig davon, ob sie einen akzeptierenden Zustand erreicht haben
  - Den Begriff „Endzustand“ werden wir später noch für Zustände (für andere Berechnungsmodelle) verwenden, die zum sofortigen Anhalten führen
- Wir verwenden trotzdem den üblichen Buchstaben  $F$  für die Menge der akzeptierenden Zustände, auch wenn er sich von **final** herleitet

## Bemerkung 2.2

- Wir verwenden hier  $((00)^+)$  als Abkürzung für  $00(00)^*$

## $\delta$ : Funktion oder Relation?

- Wir bezeichnen mit  $\delta$  in DFAs eine Funktion, in NFAs eine Relation
  - In DFAs könnten wir  $\delta$  auch als Relation schreiben
    - \* aber Funktionen sind dort intuitiver, da es zu jedem  $p \in Q$  und  $\sigma \in \Sigma$  nur genau einen Zustand  $q$  mit  $(p, \sigma, q) \in \delta$  gibt
  - Umgekehrt ließe sich  $\delta$  in NFAs auch als die Funktion definieren, die  $p$  und  $\sigma$  auf  $\{q \mid (p, \sigma, q) \in \delta\}$  abbildet
    - \* Für NFAs hat aber die Relationsschreibweise Vorteile

## Mengen von Startzuständen

- In der Literatur werden NFAs manchmal auch mit einer Menge  $I$  von Startzuständen definiert

# Zusammenfassung

## Themen dieses Kapitels

- Definition von endlichen Automaten, deterministisch und nichtdeterministisch
- Semantik von endlichen Automaten
- Konstruktion endlicher Automaten

## Kapitelfazit

- Endliche Automaten sind die Abstraktion von Programmen einer sehr einfachen Struktur
- Um reguläre Ausdrücke in DFAs umzuwandeln, sind nichtdeterministische endliche Automaten ein hilfreicher Zwischenschritt
- REs lassen sich leicht in  $\epsilon$ -NFAs umwandeln

# Grundbegriffe der Theoretischen Informatik

Sommersemester 2016 - Thomas Schwentick

Teil A: Reguläre Sprachen

3: Äquivalenz der Modelle

Version von: 26. April 2016 (12:14)

# Einleitung

- Wir kennen schon:
    - Reguläre Ausdrücke (REs)
    - NFAs (auch mit  $\epsilon$ -Übergängen)
    - DFAs
  - Wir können REs in  $\epsilon$ -NFAs umwandeln
- 
- In diesem Kapitel werden wir sehen:
    - $\epsilon$ -NFAs (und NFAs) lassen sich in DFAs umwandeln
    - DFAs lassen sich auch in REs umwandeln
  - Alle vier Modelle sind gleich mächtig
- 
- Wir werden die Größen der dabei jeweils konstruierten Objekte vergleichen
  - Und mit dem Nachweis der Korrektheit von Automaten werden wir uns auch beschäftigen

# Inhalt

## ▷ 3.1 Vom NFA zum DFA

3.2 Vom DFA zum RE

3.3 Größenverhältnisse bei den Umwandlungen

3.4 Korrektheitsbeweise für DFAs

## Vom NFA zum DFA

- Wir verhalten sich NFAs zu DFAs?
- Gibt es Sprachen, die von einem NFA entschieden werden, aber von keinem DFA?
- Erstaunlicherweise und praktischerweise ist die Antwort **nein!**

### Satz 3.1

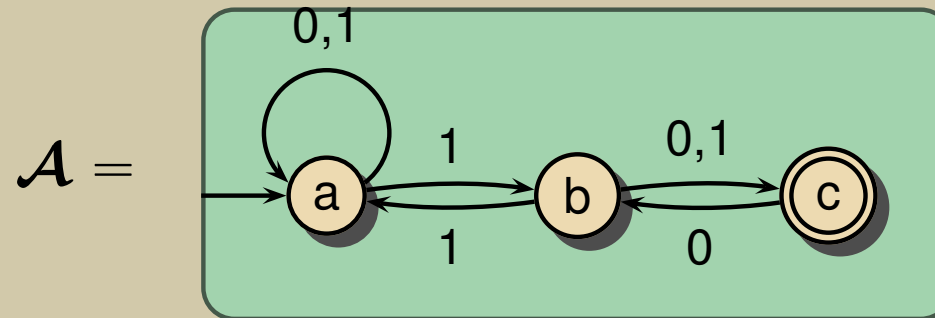
- Zu jedem NFA  $\mathcal{A}$  gibt es einen DFA  $\mathcal{A}_D$  mit  $L(\mathcal{A}_D) = L(\mathcal{A})$

### Beweisidee

- Als Zustände von  $\mathcal{A}_D$  werden die Teilmengen der Zustandsmenge von  $\mathcal{A}$  gewählt
  - Nach Lesen eines Wortes  $w$  soll der Zustand von  $\mathcal{A}_D$  die Menge aller Zustände von  $\mathcal{A}$  sein, zu denen es einen Lauf vom Startzustand gibt, der  $w$  liest
- **Potenzmengen-Automat**

# Potenzmengen-Automat: Beispiel (1/3)

Beispiel



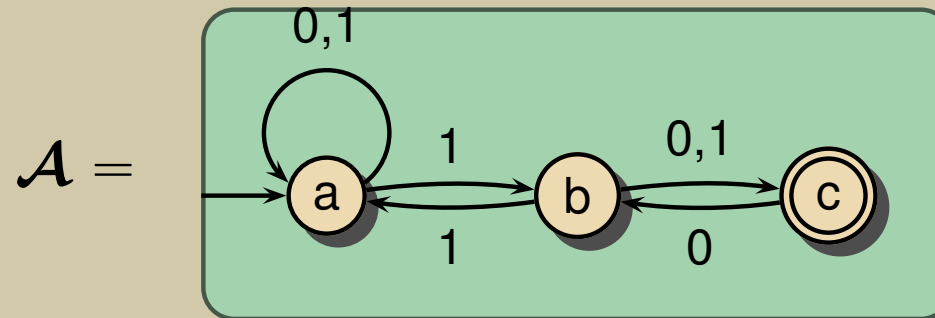
PINGO-Frage: [pingo.upb.de](http://pingo.upb.de)

Was ist die Menge aller Zustände von  $\mathcal{A}$ , zu denen es einen Lauf vom Startzustand gibt, der **1010** liest?

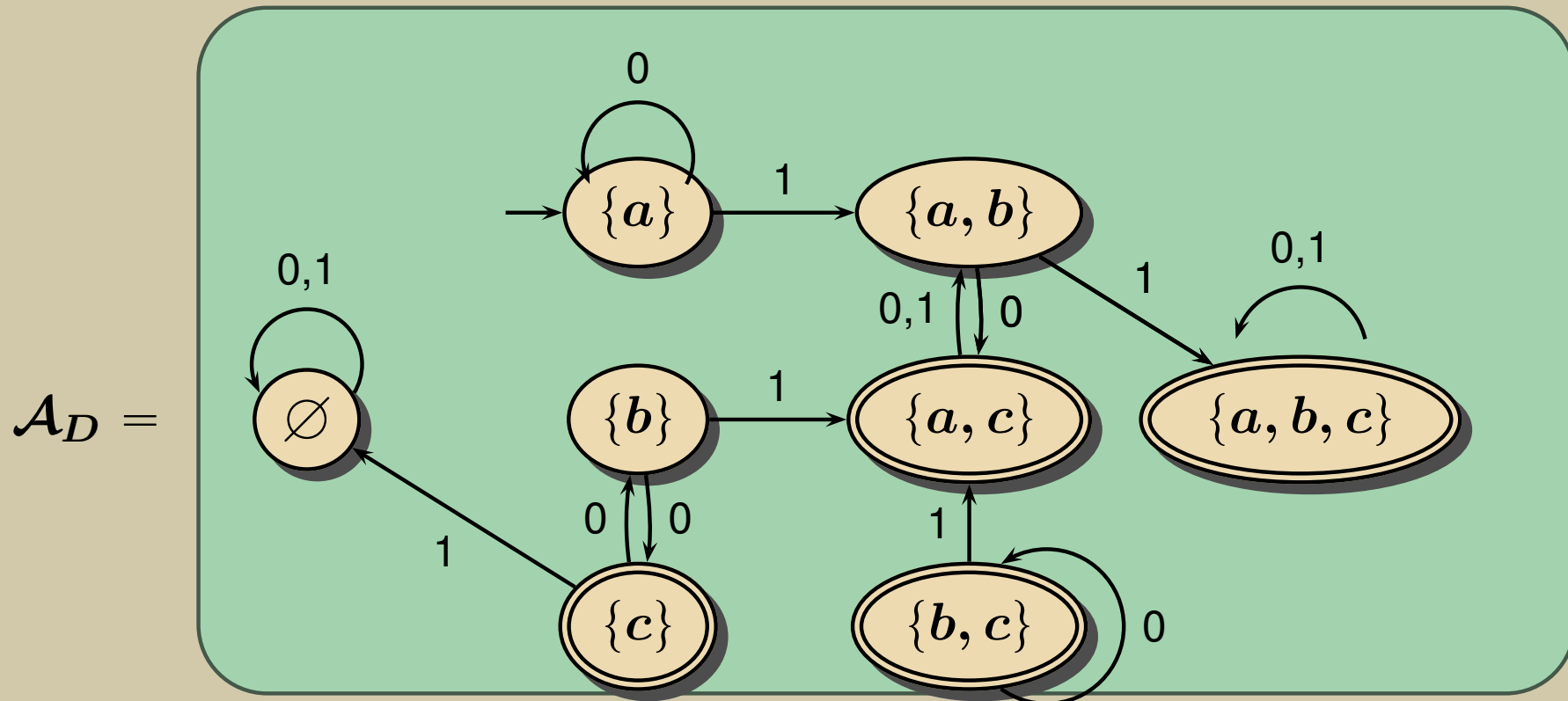
- (A)  $\{a, b, c\}$
- (B)  $\emptyset$
- (C)  $\{a, c\}$
- (D)  $\{a, b\}$

# Potenzmengen-Automat: Beispiel (2/3)

Beispiel



1010



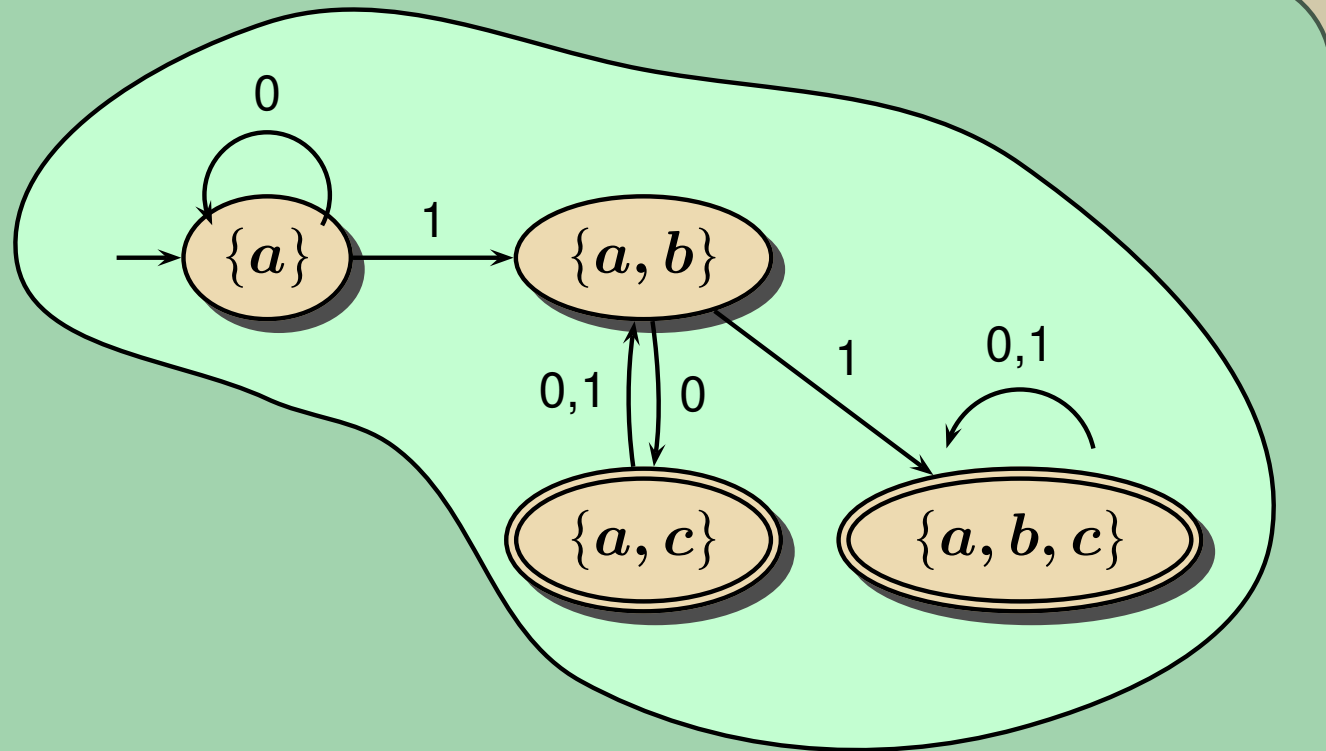


## Potenzmengen-Automat: Beispiel (3/3)

- Es genügt, die von  $\{a\}$  aus erreichbaren Zustände in  $\mathcal{A}_D$  aufzunehmen

Beispiel

$\mathcal{A}_D =$



# Einschub: Strukturelle Induktion

- Wir beweisen die Korrektheit von  $\mathcal{A}_D$  mit **struktureller Induktion**

- Die Definition von regulären Ausdrücken ist ein Beispiel für eine **induktive Definition** einer Menge:

- Zuerst werden gewisse Grundelemente der Menge definiert

\*  $\emptyset$ ,  $\epsilon$  und  $\sigma$ , für alle  $\sigma \in \Sigma$

- Dann wird beschrieben, wie aus gegebenen Elementen der Menge neue Elemente gewonnen werden:

\* Konkatenation, Auswahl, Wiederholung

- Die Menge besteht dann aus allen so (in endlich vielen Schritten) konstruierbaren Elementen, und keinen anderen

- Ein (hoffentlich) bekanntes Beispiel einer induktiven Definition:

–  $0 \in \mathbb{N}_0$

–  $n \in \mathbb{N}_0 \Rightarrow n + 1 \in \mathbb{N}_0$

- Induktive Definitionen ermöglichen:

- induktive Definitionen von Funktionen auf den Elementen der Menge
- induktive Beweise von Eigenschaften aller Elemente der Menge und

- Auch die Menge  $\Sigma^*$  aller Strings über  $\Sigma$  lässt sich induktiv definieren:

–  $\epsilon \in \Sigma^*$

– Ist  $w \in \Sigma^*$  und  $\sigma \in \Sigma$ , so ist  $w \cdot \sigma \in \Sigma^*$

- Induktive Definition einer Funktion über  $\Sigma^*$ :

–  $\delta^*(q, \epsilon) = q$ ,

–  $\delta^*(q, u\sigma) = \delta(\delta^*(q, u), \sigma)$   
für  $u \in \Sigma^*, \sigma \in \Sigma$

- Beweise mit **struktureller Induktion** beweisen die Aussage zuerst für die Grundelemente und dann für „zusammengesetzte Elemente“

# Beweis von Satz 3.1

## Satz 3.1

- Zu jedem NFA  $\mathcal{A}$  gibt es einen DFA  $\mathcal{A}_D$  mit  $L(\mathcal{A}_D) = L(\mathcal{A})$

## Beweis

- Sei  $\mathcal{A} = (Q, \Sigma, \delta, s, F)$
- Wir definieren  $\mathcal{A}_D \stackrel{\text{def}}{=} (\mathcal{P}(Q), \Sigma, \delta_D, \{s\}, F_D)$  durch
  - $\delta_D(S, \sigma) \stackrel{\text{def}}{=} \{q \mid \exists p \in S : p \xrightarrow{\sigma} q\}$ ,  
für alle  $S \subseteq Q$  und  $\sigma \in \Sigma$ , und
  - $F_D \stackrel{\text{def}}{=} \{S \subseteq Q \mid S \cap F \neq \emptyset\}$
- Für jeden String  $w \in \Sigma^*$  sei  $R(w) \stackrel{\text{def}}{=} \{q \mid s \xrightarrow{w} q\}$

## Beweis (Forts.)

- Durch Induktion nach  $w$  zeigen wir:

$$\delta_D^*(\{s\}, w) = R(w) \quad (*)$$

- $w = \epsilon$ :  $\delta_D^*(\{s\}, \epsilon) = \{s\} = R(\epsilon)$

- $w = u\sigma$ :

$$\begin{aligned} \delta_D^*(\{s\}, w) &= \delta_D(\delta_D^*(\{s\}, u), \sigma) && \text{Def } \delta_D^* \\ &= \delta_D(R(u), \sigma) && \text{Induktion} \\ &= \{q \mid \exists p \in R(u) : p \xrightarrow{\sigma} q\} && \text{Def } \delta_D \\ &= \{q \mid \exists p : s \xrightarrow{u} p \xrightarrow{\sigma} q\} && \text{Def } R \\ &= R(u\sigma) = R(w) \end{aligned}$$

- Also:

$$\begin{aligned} &\mathcal{A}_D \text{ akzeptiert } w \\ &\iff \delta_D^*(\{s\}, w) \in F_D && \text{Def „DFA akzeptiert“} \\ &\iff \delta_D^*(\{s\}, w) \cap F \neq \emptyset && \text{Def } F_D \\ &\iff R(w) \cap F \neq \emptyset && (*) \\ &\iff \mathcal{A} \text{ akzeptiert } w && \text{Def „NFA akzeptiert“} \end{aligned}$$

## Vom $\epsilon$ -NFA zum DFA (1/2)

- Wir haben REs nicht in NFAs sondern in  $\epsilon$ -NFAs umgewandelt
- $\epsilon$ -NFAs müssen auch noch in DFAs umgewandelt werden

### Proposition 3.2

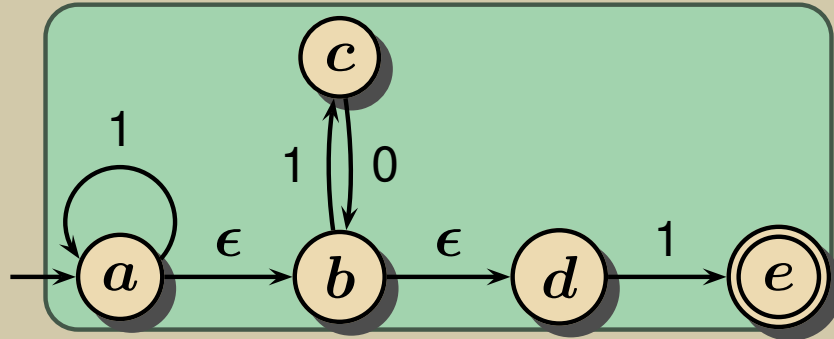
- Zu jedem  $\epsilon$ -NFA  $\mathcal{A}$  gibt es einen DFA  $\mathcal{A}_D$  mit  $L(\mathcal{A}_D) = L(\mathcal{A})$

### Beweisskizze

- Sehr ähnlich zur Umwandlung von NFAs in DFAs
- Wir verwenden einen neuen Begriff:
  - $\epsilon$ -closure( $p$ )  $\stackrel{\text{def}}{=} \{q \mid p \xrightarrow{\epsilon} q\}$ 
    - \* (Menge aller von  $p$  aus ohne Lesen eines Symbols erreichbaren Zustände)
  - Für  $S \subseteq Q$ :
$$\underline{\epsilon\text{-closure}(S)} \stackrel{\text{def}}{=} \bigcup_{q \in S} \epsilon\text{-closure}(q)$$
- Gegenüber dem Beweis von Satz 3.1 zu ändern:
  - Startzustand:  $\epsilon\text{-closure}(s)$  statt  $\{s\}$
  - $\delta_D(S, \sigma) \stackrel{\text{def}}{=} \epsilon\text{-closure}(\{q \mid \exists p \in S : p \xrightarrow{\sigma} q\})$

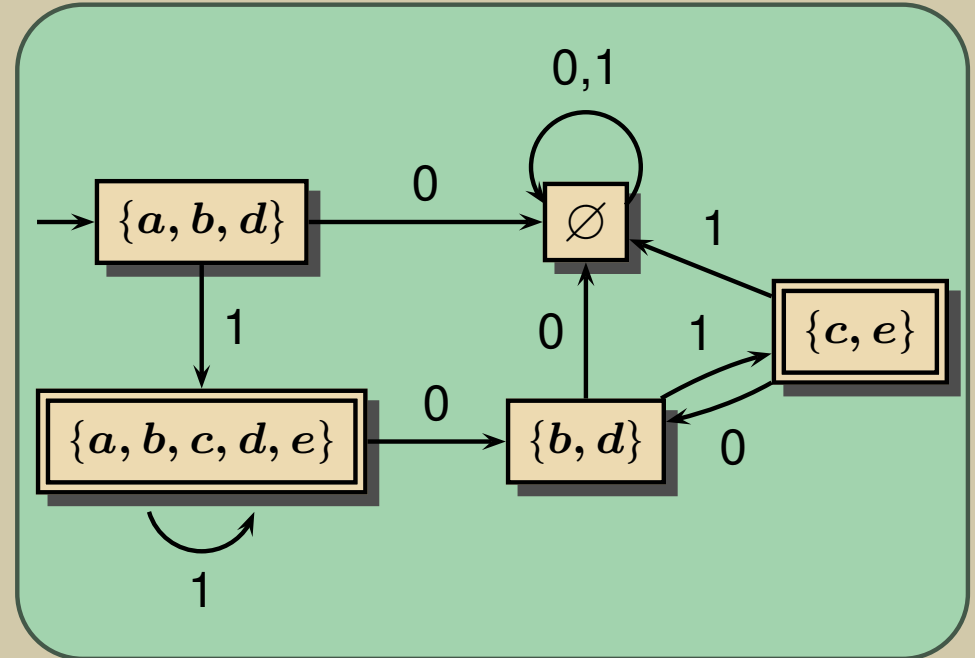
## Vom $\epsilon$ -NFA zum DFA (2/2)

Beispiel:  $\epsilon$ -NFA

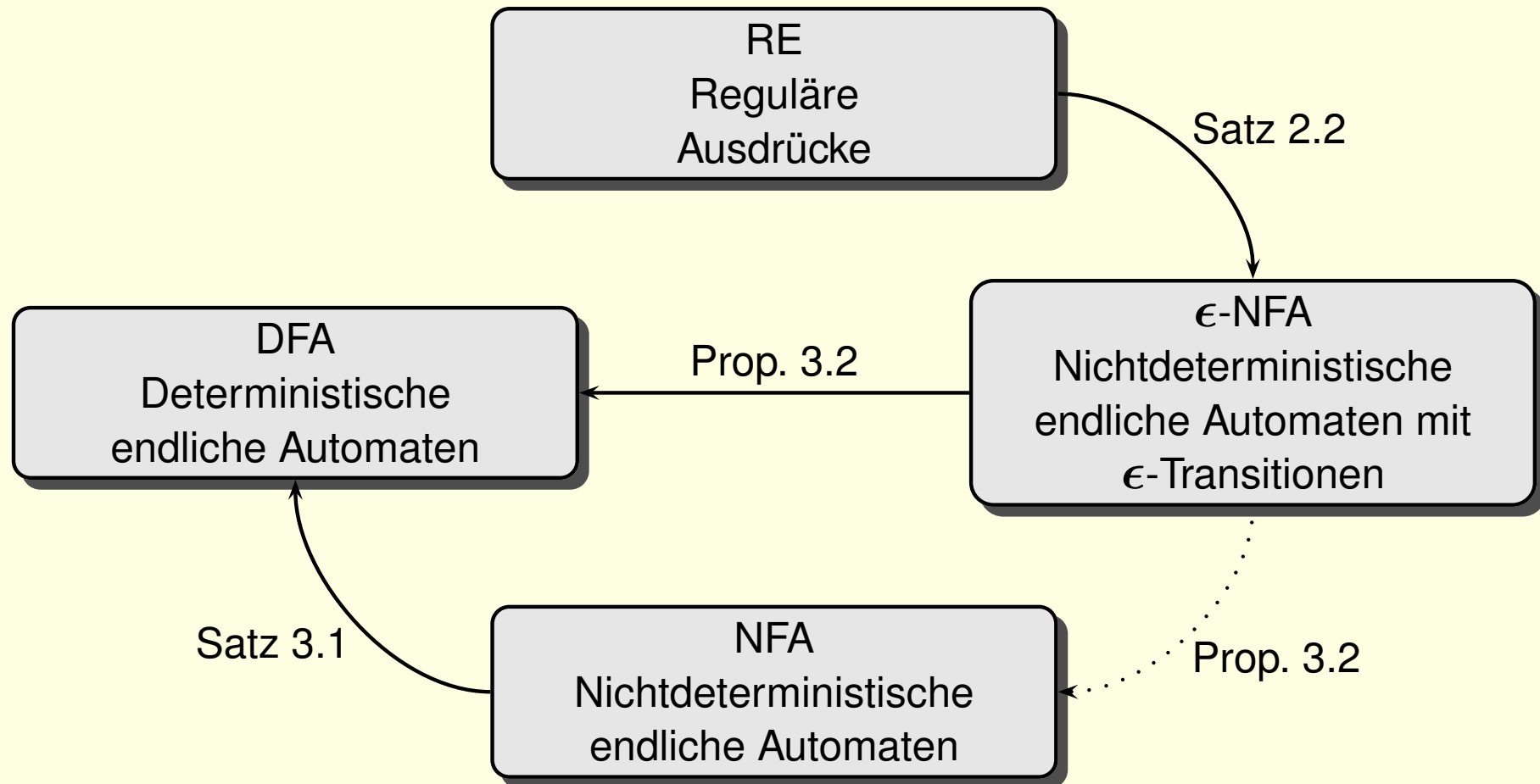


- $\epsilon$ -closure( $a$ ) =  $\{a, b, d\}$
- $\epsilon$ -closure( $b$ ) =  $\{b, d\}$

Beispiel: äquivalenter DFA

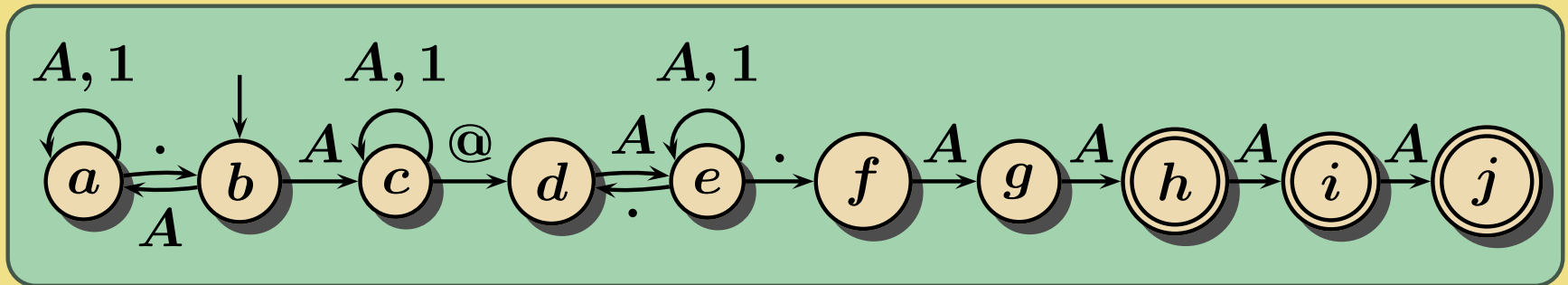


# Die Äquivalenz der Modelle (Forts.)



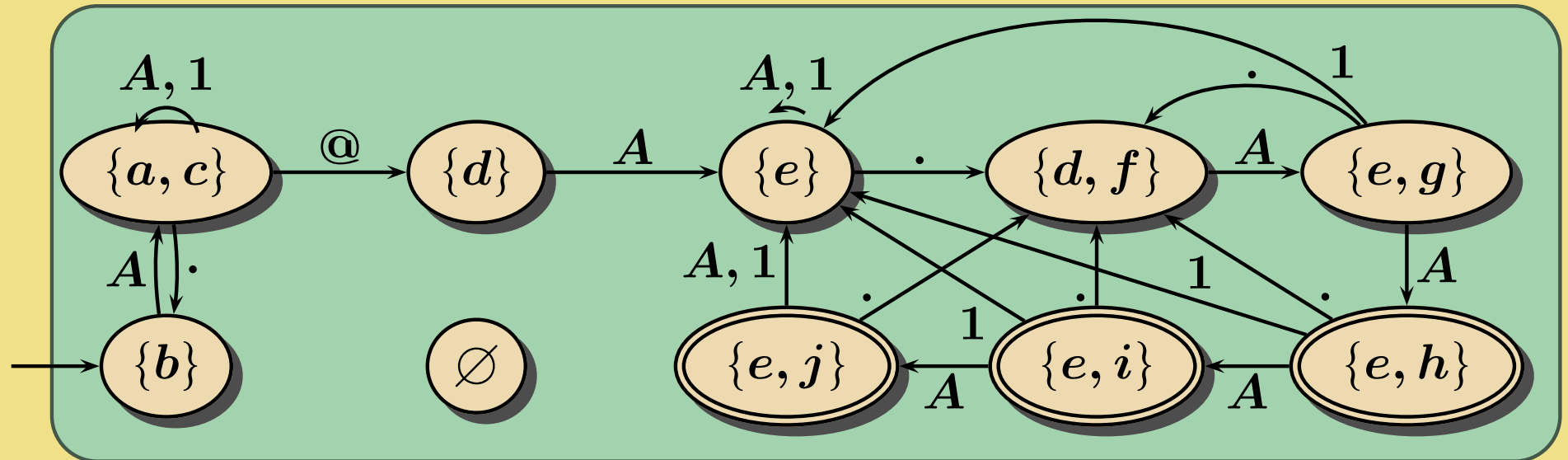
## Vom RE zum DFA

- Im letzten Kapitel hatten wir aus dem erweiterten regulären Ausdruck für Mail-Adressen bereits einen NFA konstruiert:



✎ Zur Erinnerung:  $A$  steht für  $a \dots, z, A \dots, Z$  und  $1$  für  $0, \dots, 9, -, _$

- Dieser lässt sich in den folgenden DFA umwandeln:



✎ Alle übrigen Übergänge führen in den Zustand  $\emptyset$

# Inhalt

3.1 Vom NFA zum DFA

▷ **3.2 Vom DFA zum RE**

3.3 Größenverhältnisse bei den Umwandlungen

3.4 Korrektheitsbeweise für DFAs



# Endliche Automaten vs. reguläre Ausdrücke

- Um den Nachweis der Äquivalenz der betrachteten Modelle abzuschließen, zeigen wir folgendes Resultat

Proposition 3.3 [McNaughton, Yamada 60]

- Zu jedem DFA  $\mathcal{A}$  gibt es einen RE  $\alpha$  mit
$$L(\alpha) = L(\mathcal{A})$$

- Für den Beweis von Proposition 3.3 betrachten wir zuerst einen konstruktiven und anschaulichen Weg, um von  $\mathcal{A}$  zu  $\alpha$  zu kommen:

## Zustandselimination

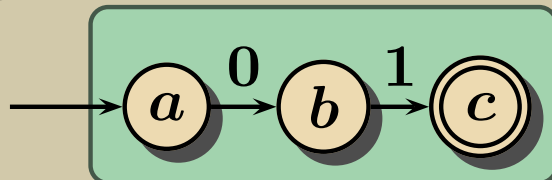
- Der Nachweis, dass diese Konstruktion korrekt ist, ist aber technisch mühsam
- Deshalb führen wir den formalen Beweis für Proposition 3.3 dann auf eine etwas weniger anschauliche, aber leicht hinzuschreibende Weise

# Vom DFA zum RE: Anschauliche Vorgehensweise (1/4)

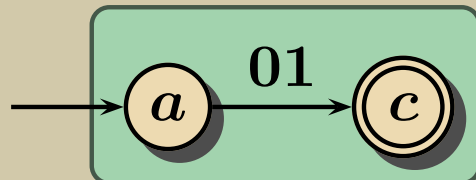
- Grundidee der anschaulichen Vorgehensweise
  - Wir verwenden ein **hybrides Automatenmodell**, dessen Transitionen mit regulären Ausdrücken (statt einzelnen Zeichen) beschriftet sind
  - Durch **sukzessives Entfernen von Zuständen** wird schließlich ein einzelner regulärer Ausdruck erreicht

## Ein einfaches Beispiel

- Wandle



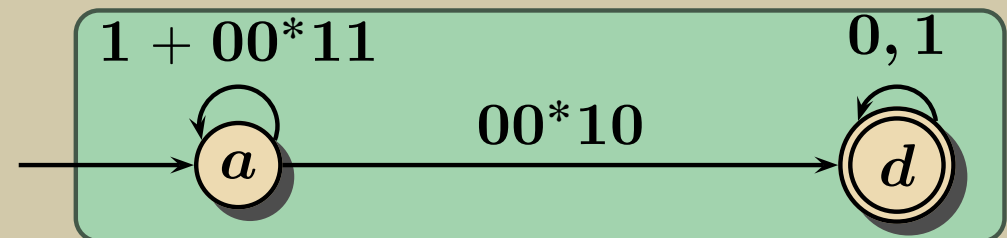
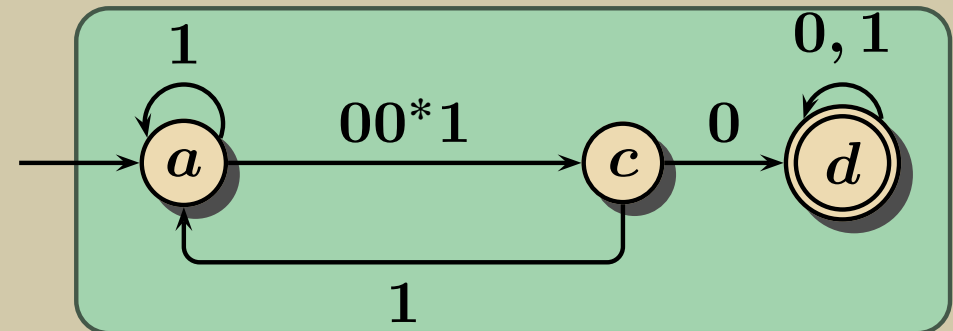
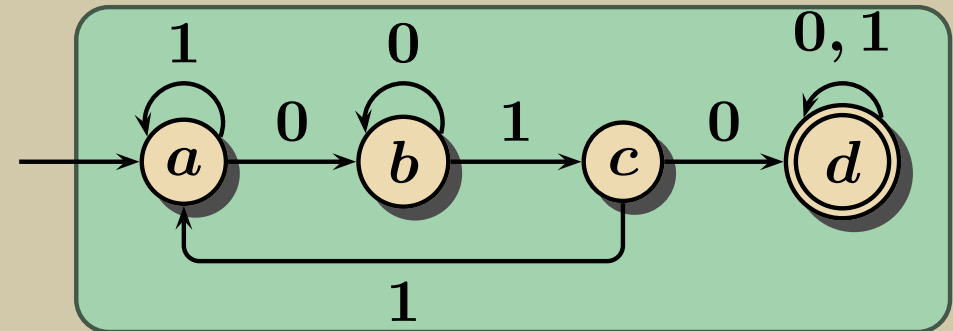
um in



und erhalte den regulären Ausdruck **01**

## Ein komplizierteres Beispiel

- Umwandlung von

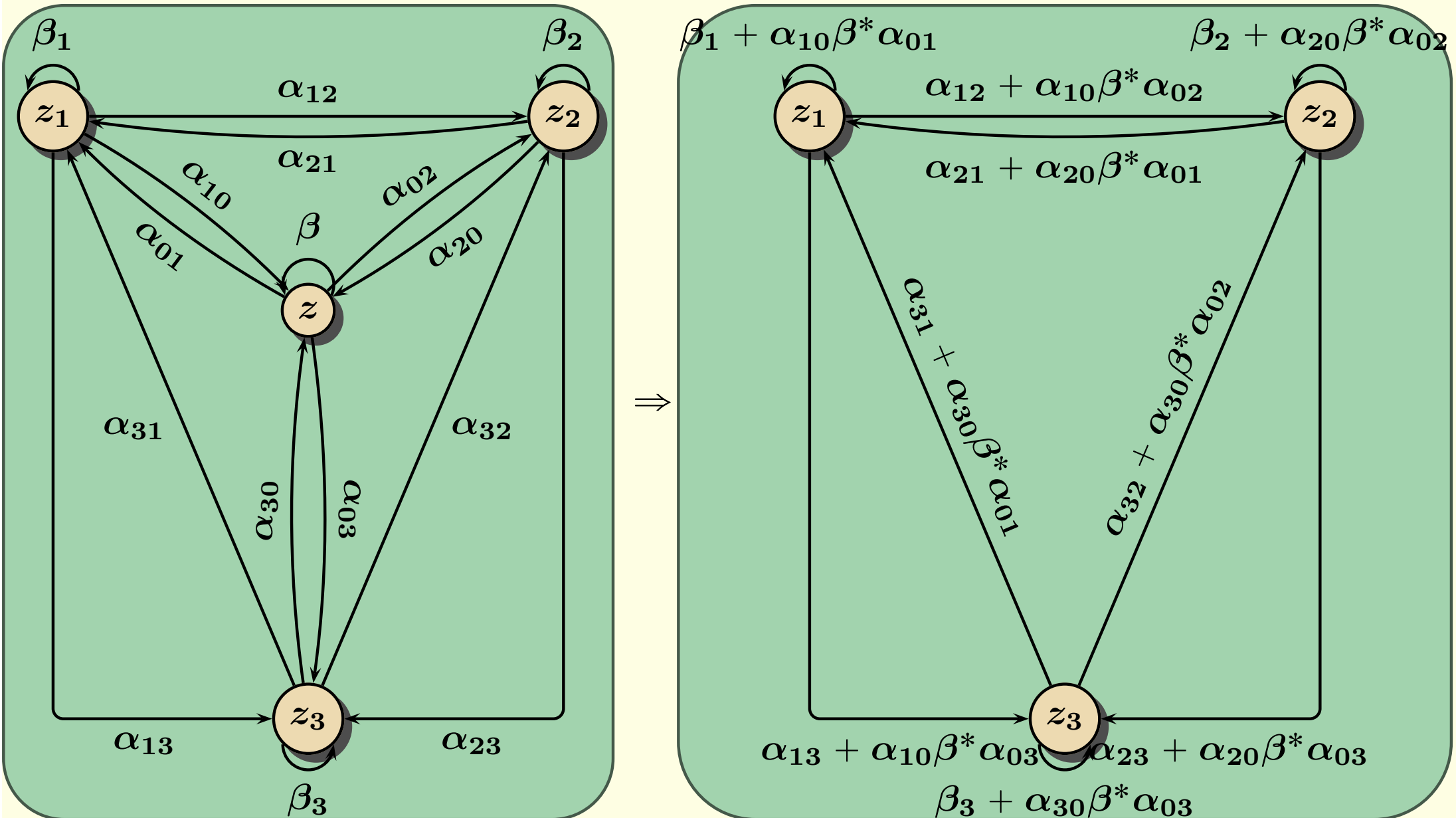


ergibt den regulären Ausdruck

$$(1 + 00^*11)^*00^*10(0 + 1)^*$$

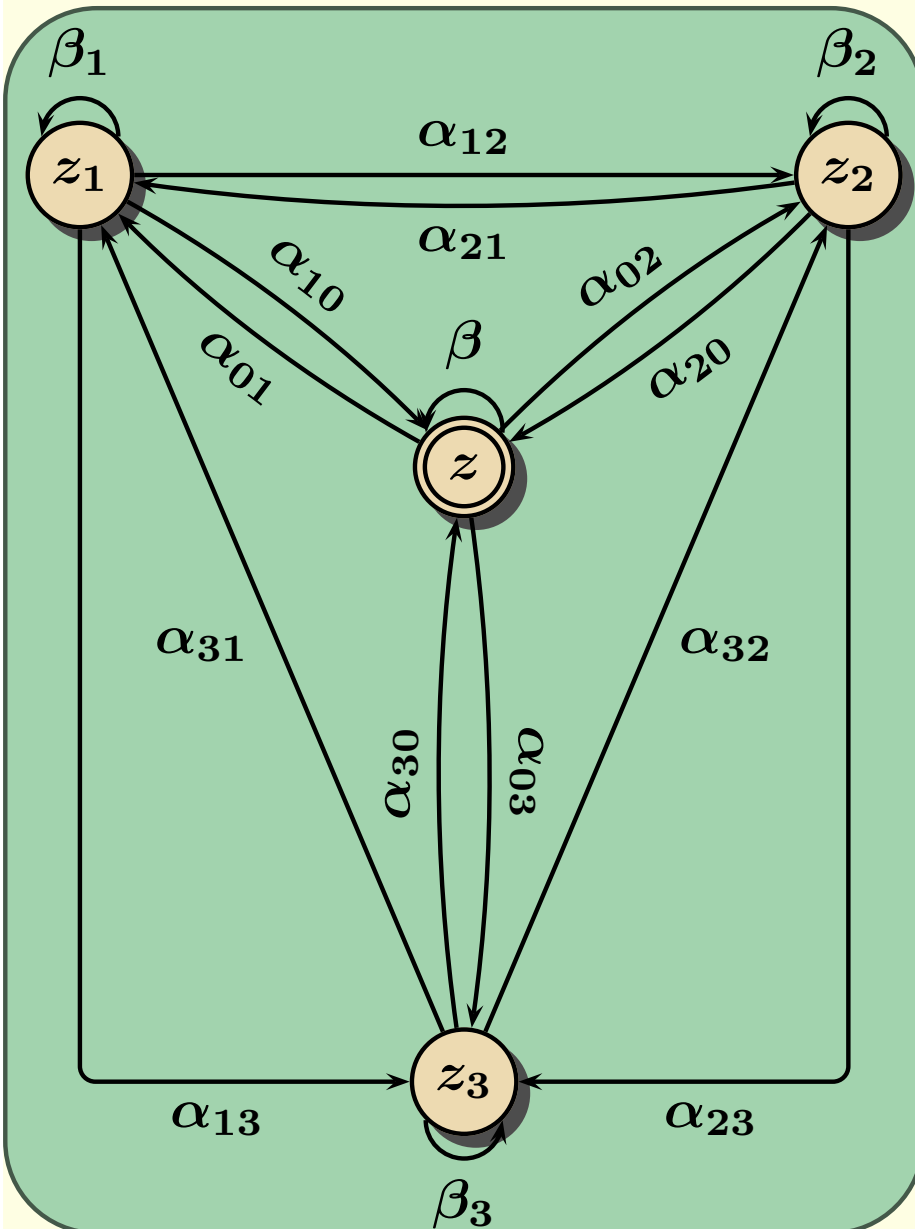
## Vom DFA zum RE: Anschauliche Vorgehensweise (2/4)

- Im Allgemeinen wird ein **nicht akzeptierender Zustand**  $z$  wie folgt entfernt:

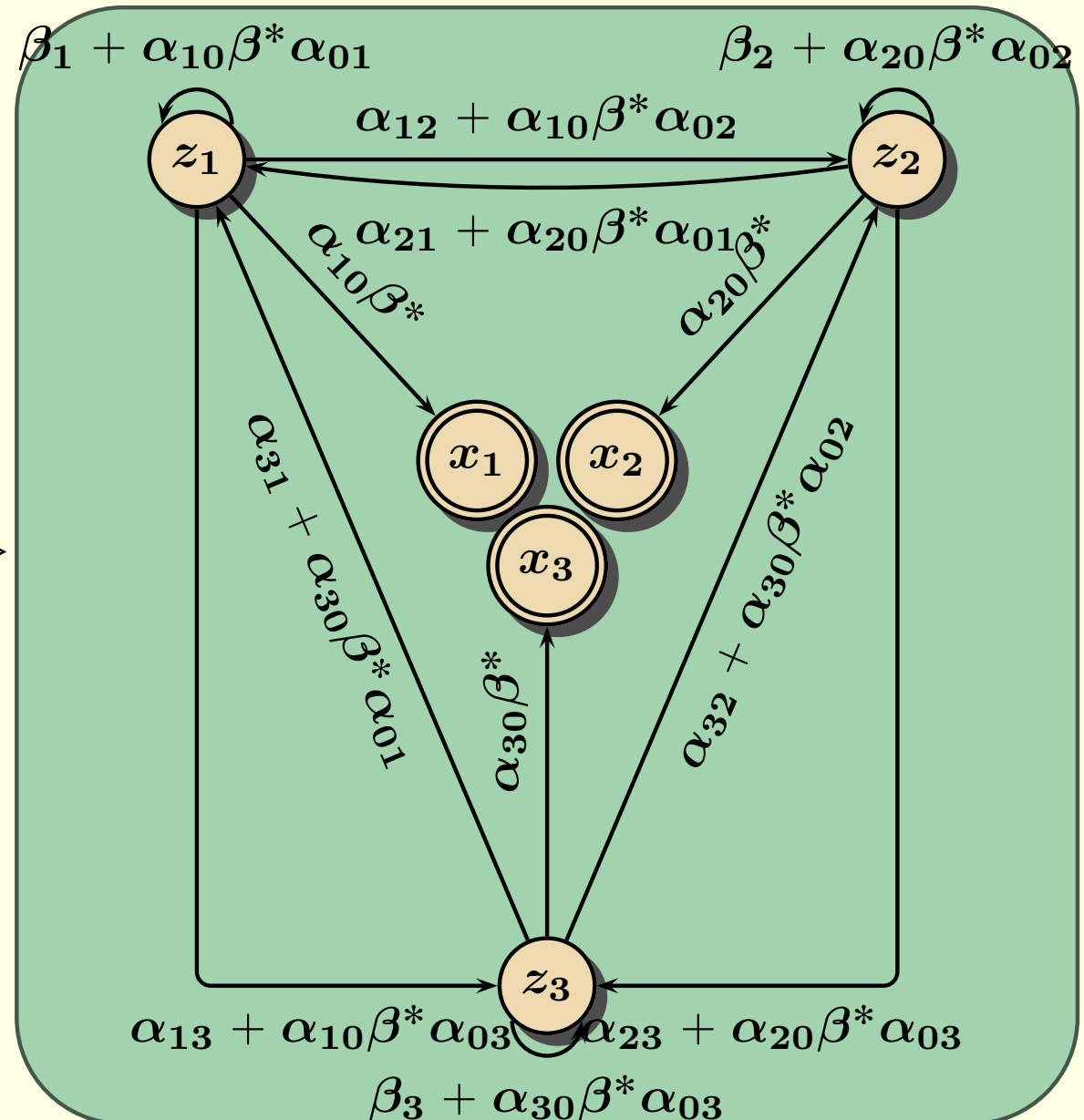


# Vom DFA zum RE: Anschauliche Vorgehensweise (3/4)

- Im Allgemeinen wird ein **akzeptierender Zustand**  $z$  wie folgt behandelt:



$\Rightarrow$

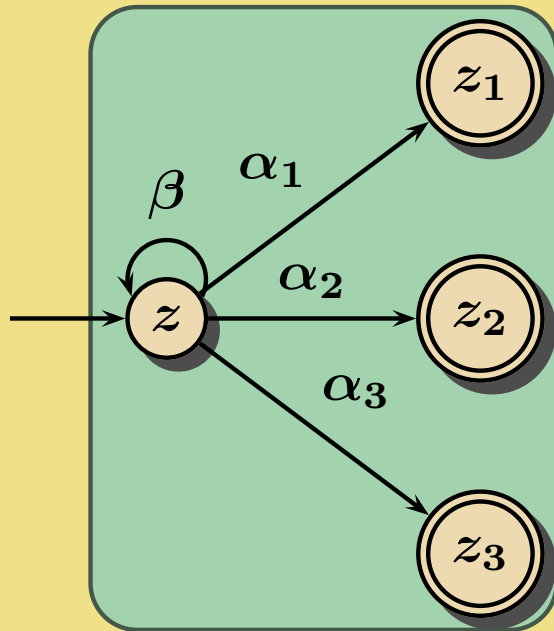


## Vom DFA zum RE: Anschauliche Vorgehensweise (4/4)

- Am Ende erhalten wir einen hybriden Automaten in einer von zwei Formen:

- 1. Fall:

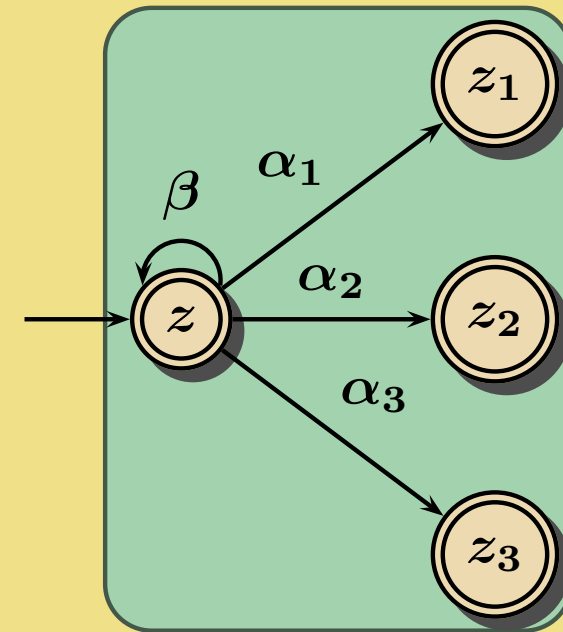
Startzustand ist **nicht akzeptierend**:



Der zugehörige reguläre Ausdruck ist dann  $\beta^*(\alpha_1 + \alpha_2 + \alpha_3)$

- Der Fall, dass es in  $z$  keine Schleife gibt, entspricht  $\beta = \emptyset$  und liefert  $\emptyset^*(\alpha_1 + \alpha_2 + \alpha_3) \equiv (\alpha_1 + \alpha_2 + \alpha_3)$

- 2. Fall: Startzustand ist **akzeptierend**:



Der zugehörige reguläre Ausdruck ist dann  $\beta^*(\alpha_1 + \alpha_2 + \alpha_3 + \epsilon)$

- Wie gesagt: der Beweis der Korrektheit dieser Vorgehensweise ist etwas mühsam
- Deshalb betrachten wir jetzt einen Beweis, der weniger anschaulich ist, sich aber leichter aufschreiben lässt

# Vom DFA zum RE: Beweis (1/4)

Proposition 3.3 [McNaughton, Yamada 60]

- Zu jedem DFA  $\mathcal{A}$  gibt es einen regulären Ausdruck  $\alpha$  mit  $L(\alpha) = L(\mathcal{A})$

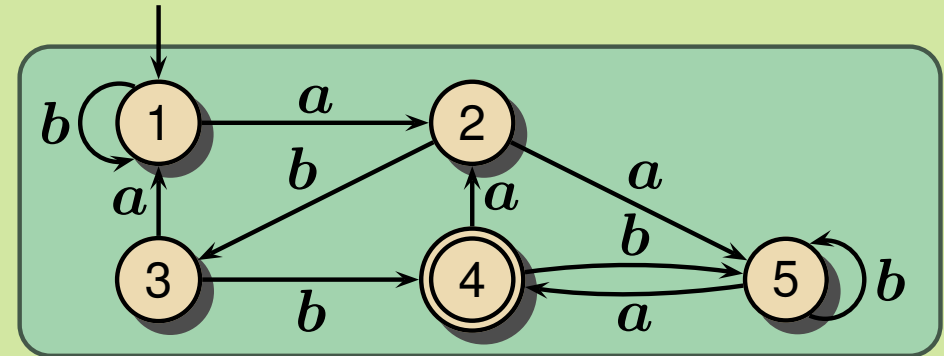
## Beweisskizze

- Sei  $\mathcal{A} = (Q, \Sigma, \delta, s, F)$
- Seien oBdA  $Q = \{1, \dots, n\}$  und  $s = 1$
- Für jedes  $i, j \in \{1, \dots, n\}$  und  $k \in \{0, \dots, n\}$  soll  $\underline{L_{i,j}^k}$  die Menge aller Strings  $w \in \Sigma^*$  sein, für die der Automat
  - vom Zustand  $i$  in den Zustand  $j$  übergeht,
  - und zwischendurch nur Zustände aus  $\{1, \dots, k\}$  annimmt
- $L_{i,j}^k \stackrel{\text{def}}{=} \text{Menge aller Strings } w \text{ mit:}$ 
  - $\delta^*(i, w) = j$  und
  - für alle echten Präfixe  $v \neq \epsilon$  von  $w$  ist  $\delta^*(i, v) \leq k$

3.1

PINGO-Frage: pingo.upb.de

Ein regulärer Ausdruck für die Menge  $L_{1,5}^3$  zum Automaten



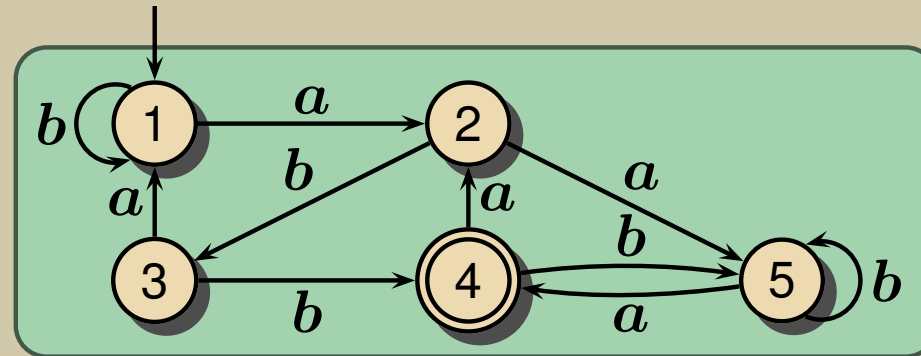
ist:

- (A)  $aa$
- (B)  $aab^*$
- (C)  $b^*a(bab^*a)^*a$
- (D)  $b^*a(bab^*a)^*ab^*$

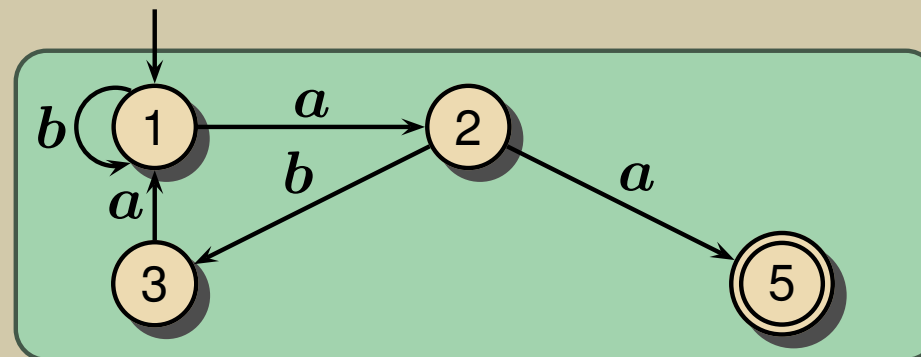
# Vom DFA zum RE: Beweis (2/4)

## Beispiel

- Für den Automaten




entspricht die Menge  $L_{1,5}^3$   
dem (partiellen) Teil-DFA



## Vom DFA zum RE: Beweis (3/4)

### Beweisskizze (Forts.)

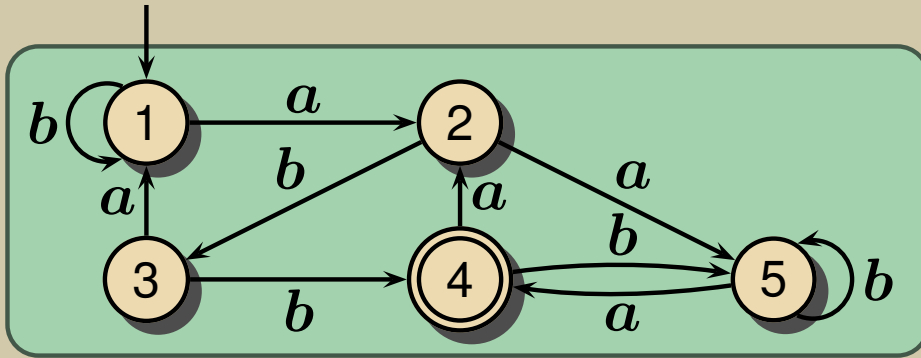
- Behauptung: für jede Menge  $L_{i,j}^k$  gibt es einen regulären Ausdruck  $\alpha_{i,j}^k$  mit  $L(\alpha_{i,j}^k) = L_{i,j}^k$
- Beweis durch Induktion nach  $k$
- $k = 0$ :
  - Für  $i \neq j$  ist
$$L_{i,j}^0 = \{\sigma \in \Sigma \mid \delta(i, \sigma) = j\}$$

 Hier sind nur direkte Übergänge erlaubt!  
Keine Zwischenschritte!
  - Analog:
$$L_{i,i}^0 = \{\epsilon\} \cup \{\sigma \in \Sigma \mid \delta(i, \sigma) = i\}$$
- $L_{i,j}^0$  und  $L_{i,i}^0$  sind endlich und können deshalb durch reguläre Ausdrücke beschrieben werden



# Vom DFA zum RE: Beweis (4/4)

## Beispiel



$\underbrace{ab}_{\in L_{1,3}^2} \underbrace{abbbab}_{\in L_{3,3}^2} \underbrace{abab}_{\in L_{3,3}^2} \underbrace{abaa}_{\in L_{3,5}^2} \in L_{1,5}^3 (!)$

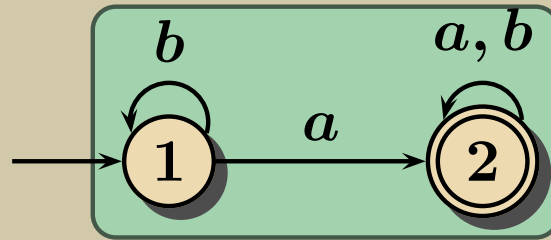
## Beweisskizze (Forts.)

- $k > 0$ :  

$$L_{i,j}^k = L_{i,j}^{k-1} \cup L_{i,k}^{k-1} (L_{k,k}^{k-1})^* L_{k,j}^{k-1}$$
- Nach Induktion gibt es für jede auf der rechten Seite vorkommende Sprache einen regulären Ausdruck, also auch für  $L_{i,j}^k$
- Der RE  $\alpha$  ist dann  $\sum_{i \in F} \alpha_{1,i}^n$
- Die Konstruktion funktioniert natürlich genauso auch für NFAs

# Vom DFA zum RE: Beispiel

## Beispiel



- $k = 0$ :
  - $\alpha_{1,1}^0 = b + \epsilon$ ,  $\alpha_{1,2}^0 = a$ ,  $\alpha_{2,1}^0 = \emptyset$ ,  $\alpha_{2,2}^0 = a + b + \epsilon$
- $k = 1$ :
  - $\alpha_{1,1}^1 = b + \epsilon + (b + \epsilon)(b + \epsilon)^*(b + \epsilon) \equiv b^*$
  - $\alpha_{1,2}^1 = a + (b + \epsilon)(b + \epsilon)^*a \equiv b^*a$
  - $\alpha_{2,1}^1 = \emptyset + \emptyset(b + \epsilon)^*(b + \epsilon) \equiv \emptyset$
  - $\alpha_{2,2}^1 = (a + b + \epsilon) + \emptyset(b + \epsilon)^*a \equiv a + b + \epsilon$
- $k = 2$ :
  - $\alpha_{1,1}^2 = b^* + b^*a(a + b + \epsilon)^*\emptyset \equiv b^*$
  - $\alpha_{1,2}^2 = b^*a + b^*a(a + b + \epsilon)^*(a + b + \epsilon) \equiv b^*a(a + b)^*$
  - $\alpha_{2,1}^2 = \emptyset + (a + b + \epsilon)(a + b + \epsilon)^*\emptyset \equiv \emptyset$
  - $\alpha_{2,2}^2 = (a + b + \epsilon) + (a + b + \epsilon)(a + b + \epsilon)^*(a + b + \epsilon) \equiv (a + b)^*$
- $\alpha = b^*a(a + b)^*$  (mit  $k = 2$  hier nur  $\alpha_{1,2}^2$  nötig)

# Reguläre Sprachen: Äquivalenz

- Insgesamt haben wir den folgenden Satz bewiesen:

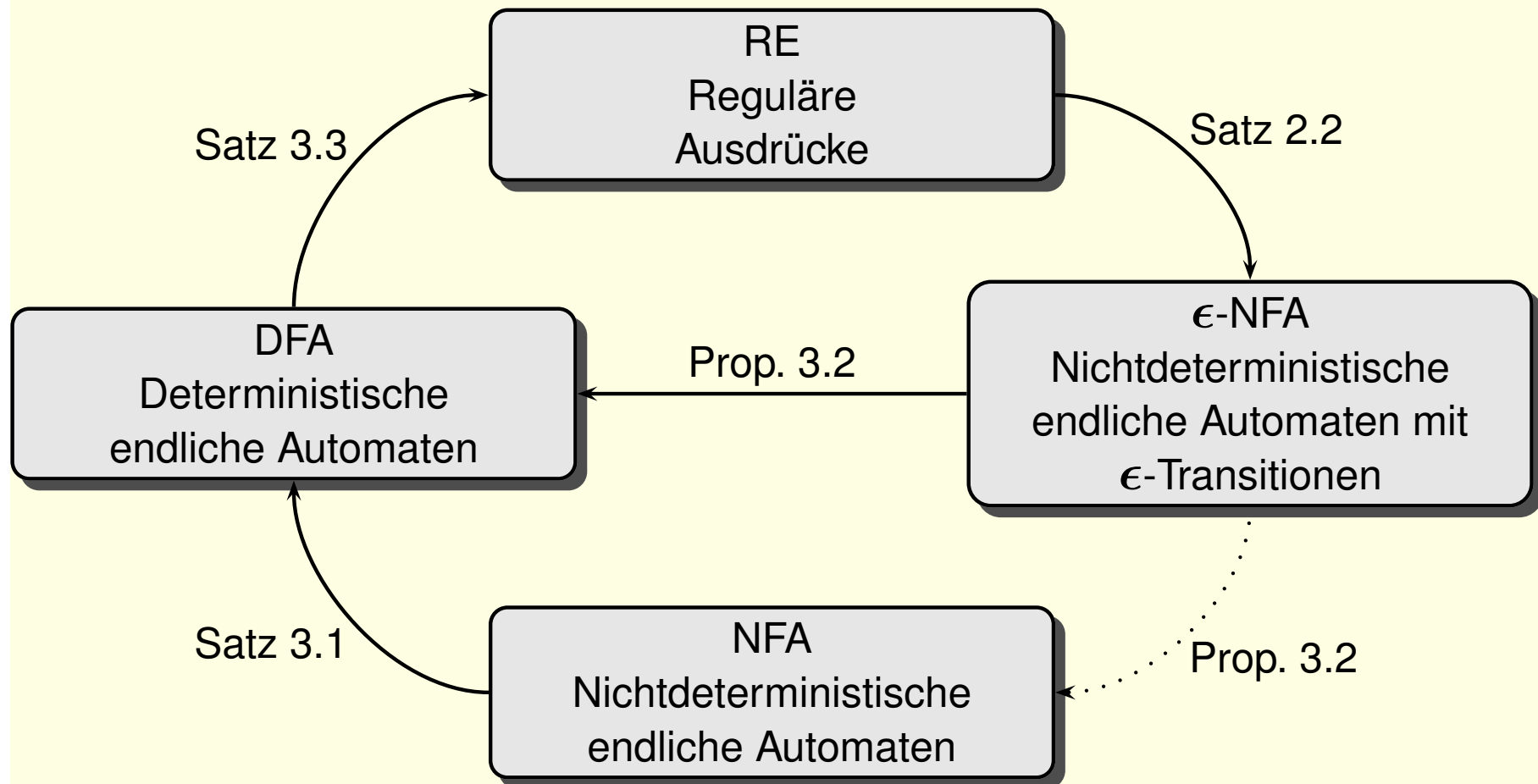
## Satz 3.4

Für eine Sprache  $L \subseteq \Sigma^*$  sind äquivalent:

- (a)  $L = L(\alpha)$  für einen RE  $\alpha$
- (b)  $L = L(\mathcal{A})$  für einen DFA  $\mathcal{A}$
- (c)  $L = L(\mathcal{A})$  für einen NFA  $\mathcal{A}$
- (d)  $L = L(\mathcal{A})$  für einen  $\epsilon$ -NFA  $\mathcal{A}$

- Die regulären Sprachen bilden also eine sehr robuste Klasse von Sprachen

# Die Äquivalenz der Modelle (Forts.)



# Inhalt

3.1 Vom NFA zum DFA

3.2 Vom DFA zum RE

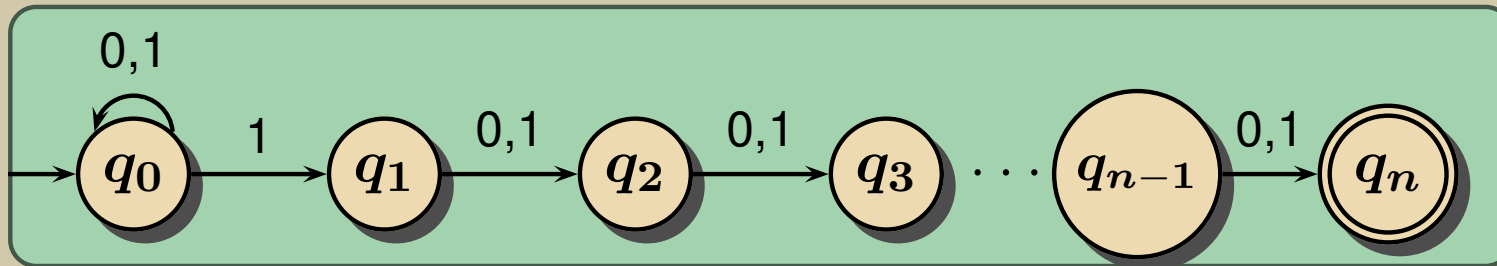
▷ **3.3 Größenverhältnisse bei den Umwandlungen**

3.4 Korrektheitsbeweise für DFAs

# Vom NFA zum DFA: Größe des Potenzmengenautomaten

- Wie groß kann der Potenzmengenautomat  $\mathcal{A}_D$  im Verhältnis zu  $\mathcal{A}$  im Beweis von Satz 3.1 werden?
- Klar: maximal  $2^{|Q|}$  Zustände (= Anzahl der Teilmengen von  $Q$ )
- Aber: kann es wirklich passieren, dass alle möglichen Teilmengen von  $Q$  erreichbar sind?

## Beispiel

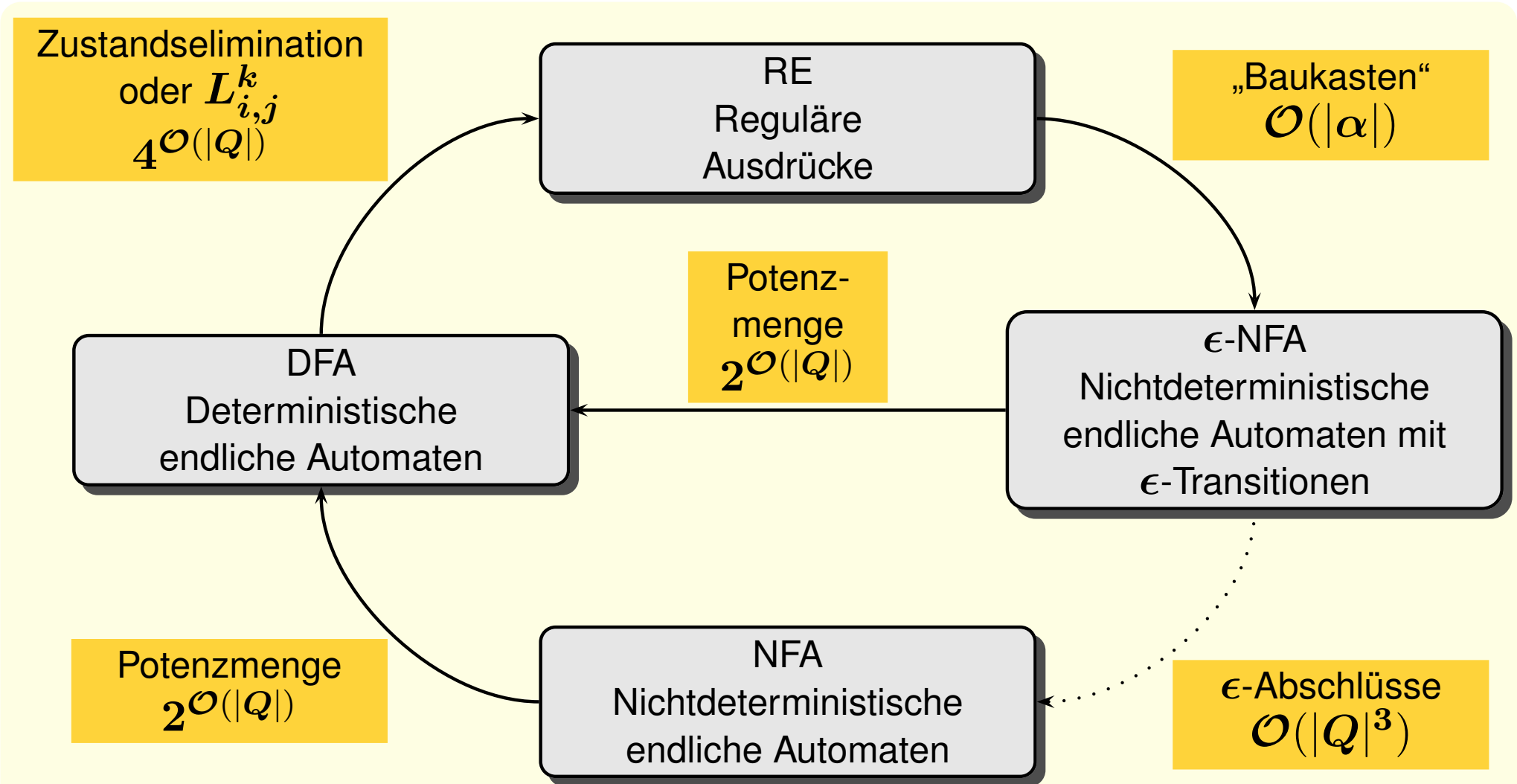


- Dieser Automat akzeptiert, falls das  $n$ -te Zeichen von rechts eine 1 ist
- Wir werden in Kapitel 5 zeigen: es gibt keinen DFA mit  $< 2^{|Q|}-1$  Zuständen für diese Sprache

## Vom DFA zum RE: Größe des REs

- Wie groß wird der RE  $\alpha$ , der bei der Umwandlung vom DFA  $\mathcal{A}$  nach dem Beweis von Proposition 3.3 entsteht?
- Sei  $g(k)$  die maximale Länge eines Ausdrucks für  $L_{i,j}^k$
- Dann gilt:
  - $g(0) = \mathcal{O}(|\Sigma|)$
  - $g(k) \leq 4g(k-1) + \mathcal{O}(1)$
- Also:  $g(n) = 4^{\mathcal{O}(n)} |\Sigma|$

# Die Äquivalenz der Modelle: Größenverhältnisse





# Inhalt

3.1 Vom NFA zum DFA

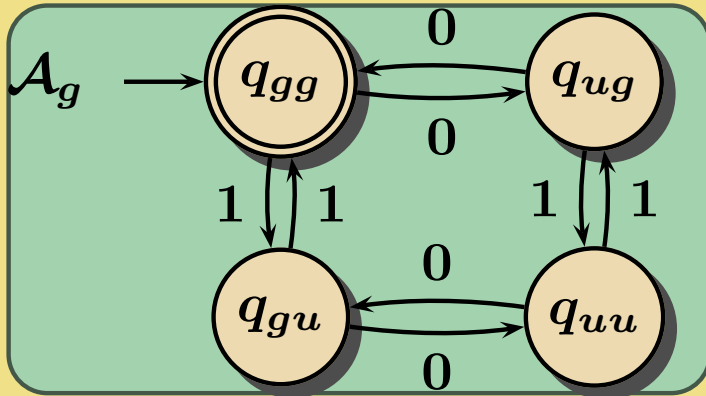
3.2 Vom DFA zum RE

3.3 Größenverhältnisse bei den Umwandlungen

▷ **3.4 Korrektheitsbeweise für DFAs**

# Korrektheitsbeweis für Automaten (1/2)

- Zur Erinnerung:  $L_g$  ist die Menge aller Strings über  $\{0, 1\}$  mit gerade vielen Einsen und Nullen



- Es erscheint offensichtlich, dass  $\mathcal{A}_g$  die Sprache  $L_g$  entscheidet
- Können wir das auch **beweisen**?

- Wir benötigen für den Beweis ein wenig Notation:
  - $\#_{\tau}(w) \stackrel{\text{def}}{=} \text{Häufigkeit des Vorkommens des Zeichens } \tau \text{ im String } w$   
 \* z.B.:  $\#_a(baaba) = 3$
  - $n \equiv_k m \stackrel{\text{def}}{\Leftrightarrow} n \text{ und } m \text{ haben bei Division durch } k \text{ denselben Rest (für } k \in \mathbb{N})$   
 \* z.B.:  $5 \equiv_3 2$

- Mit dieser Notation definieren wir formal:
  - $L_g \stackrel{\text{def}}{=} \{w \in \{0, 1\}^* \mid \#_0(w) \equiv_2 0, \#_1(w) \equiv_2 0\}$

## Proposition 3.5

- $L(\mathcal{A}_g) = L_g$
- Korrektheitsbeweise für Automaten zeigen meisten durch Induktion nach  $w$ , dass die intuitive Bedeutung der Zustände mit der tatsächlichen Bedeutung übereinstimmt

# Korrektheitsbeweis für Automaten (2/2)

## Proposition 3.5

- $L(\mathcal{A}_g) = L_g$

## Beweisskizze

- Wir zeigen durch Induktion nach  $|w|$ , dass für alle  $w \in \Sigma^*$  gilt:
  - $\delta^*(q_{gg}, w) = q_{gg} \iff \#_0(w) \equiv_2 0, \#_1(w) \equiv_2 0$
  - $\delta^*(q_{gg}, w) = q_{ug} \iff \#_0(w) \equiv_2 1, \#_1(w) \equiv_2 0$
  - $\delta^*(q_{gg}, w) = q_{gu} \iff \#_0(w) \equiv_2 0, \#_1(w) \equiv_2 1$
  - $\delta^*(q_{gg}, w) = q_{uu} \iff \#_0(w) \equiv_2 1, \#_1(w) \equiv_2 1$
- Daraus folgt dann die Proposition wegen  $F = \{q_{gg}\}$

## Beweisskizze (Forts.)

- $w = \epsilon: \checkmark$
- $w = v\sigma$  für ein  $v \in \Sigma^*$  und ein  $\sigma \in \Sigma$ :
  - Nach Induktion gilt für  $v$  die Induktionsbehauptung
  - Wir unterscheiden 8 Fälle, je nach  $\sigma$  und  $\delta^*(q_{gg}, v)$ :
    - \* Beispielfall:
$$\sigma = 1, \delta^*(q_{gg}, v) = q_{ug}$$
$$\Rightarrow \#_0(v) \equiv_2 1, \#_1(v) \equiv_2 0$$

☞ Induktion

$$\Rightarrow \#_0(w) \equiv_2 1, \#_1(w) \equiv_2 1$$

☞ da  $\sigma = 1$

$$\Rightarrow \text{Behauptung} \quad \text{☞ da } \delta(q_{ug}, 1) = q_{uu}$$
  - Die anderen sieben Fälle sind analog

# Zusammenfassung

## Themen dieser Vorlesung

- Umwandlung von NFAs und  $\epsilon$ -NFAs in DFAs (Potenzmengenkonstruktion)
- Umwandlung von DFAs in REs
- Korrektheitsbeweise für endliche Automaten

## Kapitelfazit

- Alle betrachteten Modelle beschreiben reguläre Sprachen
- Einige Umwandlungen zwischen den Modellen können exponentiell große Objekte erzeugen

# Erläuterungen: Präfixe, Suffixe und Teilstrings

## Bemerkung 3.1

- Sind  $x, y, z$  Wörter und ist  $w = xyz$ , so heißt
  - $x$  ein Präfix von  $w$ ,
  - $y$  ein Teilstring von  $w$  und
  - $z$  ein Suffix von  $w$
- Dabei können  $x, y$  oder  $z$  auch leer sein.
- Ein **echtes Präfix**  $x$  von  $w$  ist ein Präfix mit  $x \neq w$

## Beispiel

- Der String *abab* hat die
  - Präfixe  $\epsilon, a, ab, aba, abab$
  - Suffixe  $\epsilon, b, ab, bab, abab$
  - Teilstrings,  $\epsilon, a, b, ab, ba, aba, bab, abab$

## Literaturhinweise

**Umwandlung DFA → RE:** R. McNaughton and H. Yamada. Regular Expressions and State Graphs for Automata. *IEEE Transactions on Electronic Computers*, EC-9:39–47, 1960

# Grundbegriffe der Theoretischen Informatik

Sommersemester 2016 - Thomas Schwentick

Teil A: Reguläre Sprachen

4: Minimierung von Automaten

Version von: 28. April 2016 (18:34)

## Der Borussia-Newsticker-Automat (1/3)

- Ich habe einen Bekannten, der ein ziemlich großer Fan von Borussia ist
- Er sammelt auch Fanartikel, die mit Borussia zu tun haben und verfolgt einen Newsticker, der ihn über Auktionen informiert
- Dabei möchte er auch Auktionen finden, in deren Beschreibung der Name „Borussia“ falsch geschrieben wurde (borussia, borusssia, borusia, borussia, brussia, borissia)
- Können wir ihm dabei helfen?

### Definition: MultiSearch

**Gegeben:** Menge  $M = \{w_1, \dots, w_n\}$  von Zeichenketten, String  $v$

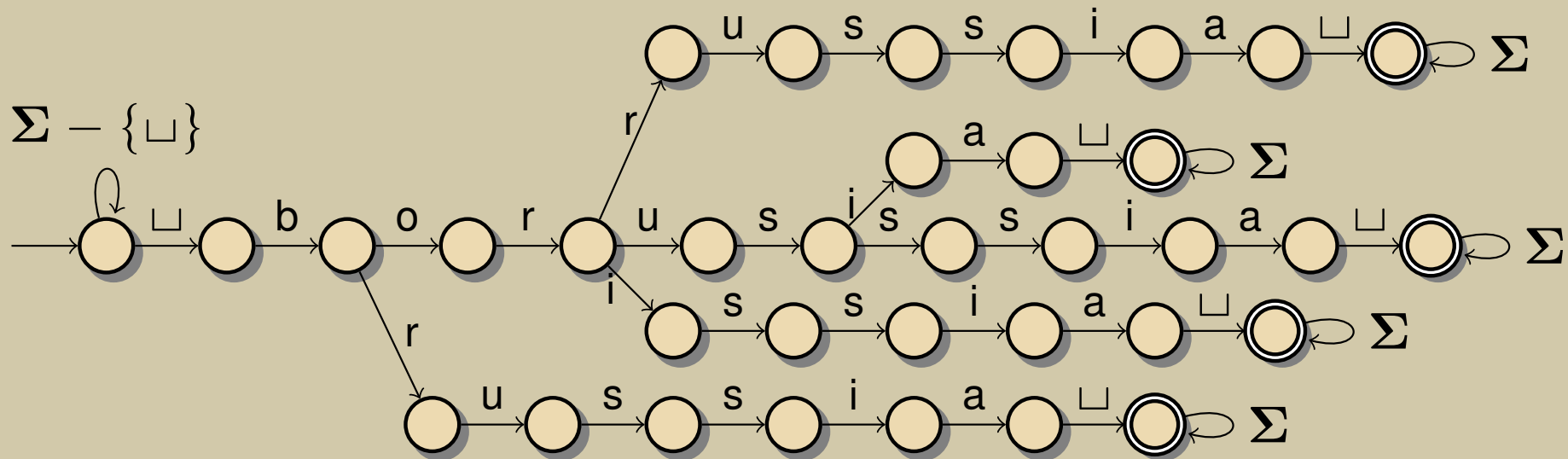
**Frage:** Kommt einer der Strings  $w_1, \dots, w_n$  in  $v$  vor?


- $v$  entspricht also dem Newsticker
- $w_1, \dots, w_n$  entsprechen den möglichen (richtigen und falschen) Schreibweisen von „Borussia“



# Der Borussia-Newsticker-Automat (2/3)

## Beispiel: Umsetzung als DFA



- $\Sigma = \{a, \dots, z, \sqcup\}$    $\sqcup$  steht für das Leerzeichen
- Jeder Zustand hat ausgehende Transitionen für alle Symbole aus  $\Sigma$ 
  - Nicht angezeigte Transitionen, bei denen Blanks gelesen werden, führen in den zweiten Zustand
  - Alle anderen nicht angezeigten Transitionen führen in den Startzustand
- Ist diese Lösung optimal?
- Oder gibt es einen kleineren Automaten für die „Borussia-Sprache“?

Idee: Wim Martens

# Minimierung endlicher Automaten: Fragen & Antworten

- Gibt es zu jedem DFA einen kleinsten äquivalenten DFA? **Natürlich!**

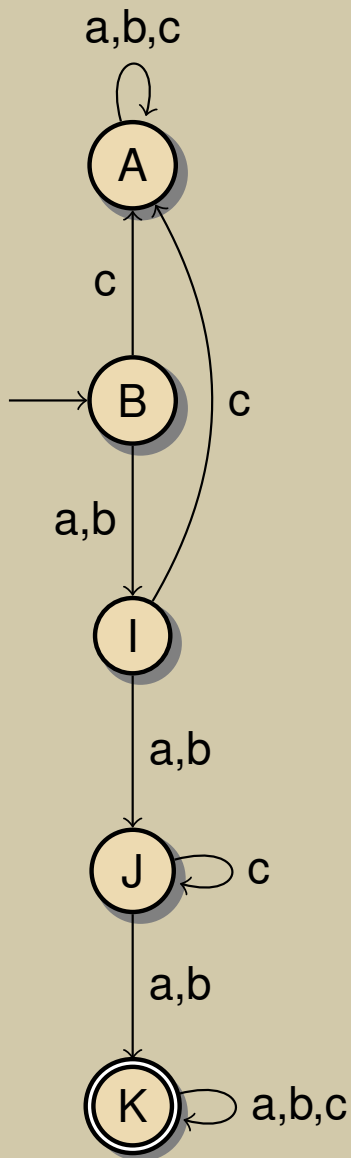
- Wieviele verschiedene kleinste äquivalente DFAs kann es geben? **Nur einen, bis auf Isomorphie**

- Kann man zu jedem DFA effizient den kleinsten äquivalenten DFA konstruieren?  
**Ja, das ist sogar ziemlich einfach**

- Zur genaueren Beantwortung dieser Fragen müssen wir die Struktur regulärer Sprachen etwas besser verstehen
- Dabei hilft uns eine etwas „mathematischere“ Sicht auf reguläre Sprachen

# Minimierung: Grundidee

## Beispiel



- $p, q$   $F$ -äquivalent:  $p \in F \iff q \in F$

## Beispiel

- Unerreichbare Zustände können gelöscht werden:  $G$  und  $H$
- $F$ -äquivalente Zustände, für die alle Übergänge in dieselben Zustände führen, können verschmolzen werden:  $E$  und  $K$
- Senkenzustände können verschmolzen werden:  $L$  und  $A$
- Allgemein:  $F$ -äquivalente Zustände, von denen aus das Akzeptierverhalten für alle nachfolgenden Eingabesequenzen gleich ist, können verschmolzen werden:
  - $D, F$ , und  $J$
  - $C$  und  $I$

- Was soll „das Akzeptierverhalten für alle nachfolgenden Eingabesequenzen ist gleich“ genau bedeuten?
- Wie lassen sich die verschmelzbaren Zustände berechnen?

# Inhalt

- ▷ **4.1 Satz von Myhill und Nerode**
- 4.2 Minimierungsalgorithmus für DFAs

# Nerode-Relation: Definition und Beispiel

- Die folgende Definition präzisiert den vagen Begriff „das Akzeptieverhalten für alle möglichen nachfolgenden Eingabesequenzen ist gleich“ durch eine Äquivalenzrelation für Strings

## Definition

- Sei  $L \subseteq \Sigma^*$  eine Sprache
- Die **Nerode-Relation**  $\sim_L$  auf  $\Sigma^*$  ist auf  $\Sigma^*$  definiert durch:
  - $x \sim_L y \stackrel{\text{def}}{\iff}$   
für alle  $z \in \Sigma^*$  gilt:  
 $xz \in L \iff yz \in L$
- Wir werden sehen: ein Automat ist minimal für  $L$ , wenn jeweils alle Strings, die bezüglich  $\sim_L$  äquivalent sind, den Automaten in den selben Zustand bringen

## Beispiel

- Sei  $L = L_g$  (gerade vielen Einsen und Nullen)
- Wann sind zwei Strings  $x$  und  $y$  bezüglich  $\sim_L$  äquivalent?

– Gilt  $011 \sim_L 01$ ?

- \* Nein: denn die Wahl von  $z = 0$  ergibt:
  - $(xz =) 0110 \in L$  aber
  - $(yz =) 010 \notin L$

– Gilt  $10 \sim_L 010$ ?

- \* Nein: denn die Wahl von  $z = 1$  ergibt:
  - $(xz =) 101 \notin L$  aber
  - $(yz =) 0101 \in L$

– Gilt  $100 \sim_L 10110$ ?

- \* Ja: Beide haben ungerade viele Einsen und gerade viele Nullen und erreichen  $L$  durch Anhängen von Strings mit ungerade vielen Einsen und gerade vielen Nullen

- Beobachtung:  $\sim_L$  hat vier Äquivalenzklassen

## Exkurs: Äquivalenzrelationen (1/3)

- Sei  $A$  eine Menge
- $A^n \stackrel{\text{def}}{=} \text{Menge der } n\text{-Tupel mit Einträgen aus } A$
- Eine Menge  $R \subseteq A^n$  heißt  **$n$ -stellige Relation über  $A$**

### Beispiel

- **Gleiches-Semester-Relation:**
  - $A$ : Menge aller Studierenden
  - $(x, y) \in R$ , falls  $x$  und  $y$  im selben Semester sind
- **Gleicher-Rest-Relation modulo  $k$ :**
  - $A$  : Menge  $\mathbb{N}$  der natürlichen Zahlen
  - $(x, y) \in R$  falls  $x$  und  $y$  bei Division durch  $k$  den selben Rest haben
  - \* Schreibweise:  $x \equiv_k y$

## Exkurs: Äquivalenzrelationen (2/3)

### Definition

- Eine 2-stellige (binäre) Relation  $R$  über  $A$  heißt
  - reflexiv  $\stackrel{\text{def}}{\iff}$   
für alle  $x \in A$  gilt:  $(x, x) \in R$
  - symmetrisch  $\stackrel{\text{def}}{\iff}$   
für alle  $x, y \in A$  gilt:  
 $(x, y) \in R \Rightarrow (y, x) \in R$
  - transitiv  $\stackrel{\text{def}}{\iff}$   
für alle  $x, y, z \in A$  gilt:  
 $(x, y) \in R, (y, z) \in R \Rightarrow (x, z) \in R$

- Eine 2-stellige reflexive, symmetrische, transitive Relation heißt Äquivalenzrelation

### Beobachtung

- $\sim_L$  ist eine **Äquivalenzrelation**

- Äquivalenzrelationen werden oft mit  $\sim$  statt  $R$  bezeichnet
- Infix-Notation:  $x \sim y$  statt  $(x, y) \in \sim$
- Beispiele: Gleiches-Semester-Relation, Gleicher-Rest-Relation

- Äquivalenzklasse: maximale Menge  $K$  von Elementen, so dass für alle  $x, y \in K$ :  $x \sim y$
- $[x] \stackrel{\text{def}}{=} \text{Äquivalenzklasse von } x$ , also die Menge aller  $y$  mit  $x \sim y$
- Wird eine Äquivalenzklasse in der Form  $[x]$  benannt, so wird  $x$  oft als Repräsentant dieser Klasse bezeichnet

- Es gilt:  $y \in [x] \iff [y] = [x]$

### Beispiel

- Bezüglich der  $\equiv_6$ -Relation gilt:  
 $[2] = \{2, 8, 14, 20, \dots\}$

# Satz von Myhill und Nerode (1/4)

- Zur Erinnerung:

$$x \sim_L y \stackrel{\text{def}}{\iff} \text{für alle } z \in \Sigma^* \text{ gilt} \\ (xz \in L \iff yz \in L)$$

## Satz 4.1

- Eine Sprache  $L$  ist genau dann regulär, wenn  $\sim_L$  endlich viele Äquivalenzklassen hat

## Beweis

- Wir zeigen zuerst:  
 $\sim_L$  hat endlich viele Klassen  $\implies L$  ist regulär
- Wir definieren den **Äquivalenzklassenautomaten**  $\mathcal{A}_L \stackrel{\text{def}}{=} (Q, \Sigma, \delta, s, F)$  für  $L$  wie folgt:  $\boxplus$ 
  - \*  $Q$  ist die Menge der Äquivalenzklassen von  $\sim_L$
  - \*  $s \stackrel{\text{def}}{=} [\epsilon]$
  - \*  $F \stackrel{\text{def}}{=} \{[x] \mid x \in L\}$
  - \* für alle  $x \in \Sigma^*, \sigma \in \Sigma$ :  
 $\delta([x], \sigma) \stackrel{\text{def}}{=} [x\sigma]$

## Beweis (Forts.)

- Vorsicht:  $\delta$  ist mit Hilfe von Repräsentanten der Klassen definiert  
→ wir müssen zeigen, dass die Definitionen des Funktionswertes **nicht** von der Wahl des Repräsentanten **abhängen**
  - Also: wenn wir zwei verschiedene Strings  $x, y$  aus einer Äquivalenzklasse von  $\sim_L$  für die Definition von  $\delta$  verwenden, erhalten wir jeweils das selbe Ergebnis
- Behauptungen:
  - (1)  $\delta$  ist wohldefiniert:  
$$x \sim_L y \implies x\sigma \sim_L y\sigma$$
  - (2)  $F$  ist sinnvoll definiert:  
$$[x] \in F \iff x \in L$$
  - (3)  $L(\mathcal{A}_L) = L$



## Satz von Myhill und Nerode: Beweis (2/4)

### Beweis (Forts.)

(1) Zu zeigen: falls  $x \sim_L y$ , so gilt für alle  $\sigma \in \Sigma$ :

- $x\sigma \sim_L y\sigma$

- Sei also  $x \sim_L y$  und  $\sigma \in \Sigma$

- Sei  $z \in \Sigma^*$  beliebig:

$$(x\sigma)z \in L \iff x(\sigma z) \in L$$

$$\iff y(\sigma z) \in L \quad \text{☞ wegen } x \sim_L y$$

$$\iff (y\sigma)z \in L$$

(2) Zu zeigen:  $[x] \in F \iff x \in L$

- $[x] \in F \Rightarrow$  es gibt  $y$  mit  $x \sim_L y$  und  $y \in L$

➡ Mit  $z = \epsilon$  ergibt sich  $x \in L \iff y \in L$ , also:  $x \in L$

- Umgekehrt folgt aus  $x \in L$  auch  $[x] \in F$  nach Definition von  $F$

# Satz von Myhill und Nerode: Beweis (3/4)

## Beweis (Forts.)

(3) Wir zeigen zunächst durch Induktion, dass für alle  $w \in \Sigma^*$  gilt:

$$\delta^*(s, w) = [w] \quad (\#)$$

–  $w = \epsilon$  ✓

–  $w = u\sigma$ :

$$\begin{aligned} \delta^*(s, u\sigma) &= \delta(\delta^*(s, u), \sigma) && \text{☞ Def. } \delta^* \\ &= \delta([u], \sigma) && \text{☞ Ind.} \\ &= [u\sigma] && \text{☞ Def. } \delta \end{aligned}$$

– Also:

$$\begin{aligned} w \in L(\mathcal{A}_L) &\iff \delta^*(s, w) \in F && \text{☞ Def } L(\mathcal{A}_L) \\ &\iff [w] \in F && \text{☞ } (\#) \\ &\iff w \in L && \text{☞ (2)} \end{aligned}$$

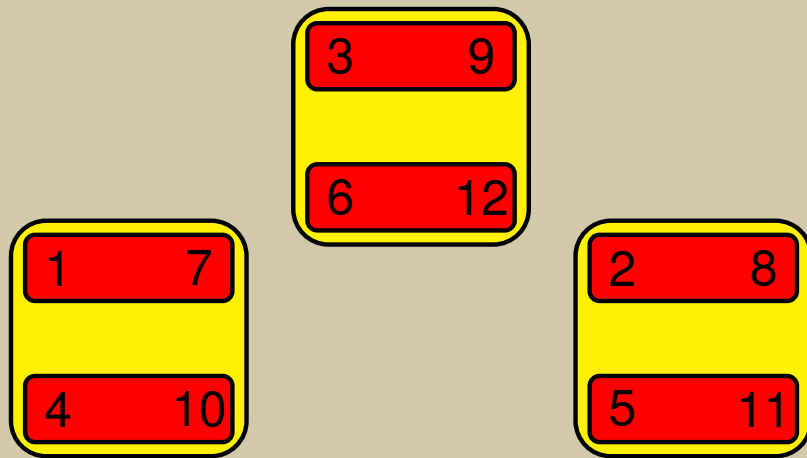
• Aus (1)-(3) folgt, dass  $\mathcal{A}_L$  ein Automat für  $L$  ist

➡  $L$  ist regulär

## Exkurs: Äquivalenzrelationen (3/3)

- Für die „Rückrichtung“ des Beweises benötigen wir den Begriff der **Verfeinerung einer Äquivalenzrelation**

### Beispiel



Klassen modulo 3

Klassen modulo 6

- Die Äquivalenzrelation  $\equiv_6$  ist eine Verfeinerung der Äquivalenzrelation  $\equiv_3$

### Definition

- Seien  $\sim_1, \sim_2$  Äquivalenzrelationen über derselben Grundmenge
- $\sim_1$  heißt **Verfeinerung von**  $\sim_2$ , wenn für alle  $x, y$  gilt:  $x \sim_1 y \Rightarrow x \sim_2 y$

- Falls  $\sim_1$  Verfeinerung von  $\sim_2$  ist, gilt:  
Anzahl Klassen von  $\sim_1 \geq$   
Anzahl Klassen von  $\sim_2$

- Weiteres Beispiel: die Gleiches-Semester- und-gleicher-Studiengang-Relation ist eine Verfeinerung der Gleiches-Semester-Relation

## Satz von Myhill und Nerode: Beweis (4/4)


### Beweis (Forts.)

- Jetzt zeigen wir:  
 **$L$  regulär  $\Rightarrow$**   
 **$\sim_L$  hat endlich viele Klassen**
- Sei  $\mathcal{A} = (Q, \Sigma, \delta, s, F)$  ein DFA für  $L$
- Wir definieren eine Äquivalenzrelation  $\sim_{\mathcal{A}}$  mit  $|Q|$  Klassen und zeigen:  
 $\sim_{\mathcal{A}}$  ist eine Verfeinerung von  $\sim_L$
- Dann folgt:
  - Anzahl Klassen von  $\sim_L$   
 $\leq$  Anzahl Klassen von  $\sim_{\mathcal{A}}$   
 $= |Q| < \infty$

### Beweis (Forts.)

- Wir definieren  $\sim_{\mathcal{A}}$  durch:  
$$\underline{x \sim_{\mathcal{A}} y} \stackrel{\text{def}}{\Leftrightarrow} \delta^*(s, x) = \delta^*(s, y)$$
- **Behauptung:**  $\sim_{\mathcal{A}}$  ist eine Verfeinerung von  $\sim_L$ , also:  
für alle  $x, y$  gilt:  $x \sim_{\mathcal{A}} y \Rightarrow x \sim_L y$
- Seien also  $x, y \in \Sigma^*$  mit  $x \sim_{\mathcal{A}} y$ 
  - ➔  $\delta^*(s, x) = \delta^*(s, y)$
  - ➔ für alle  $z \in \Sigma^*$  gilt:  
$$\delta^*(s, xz) = \delta^*(s, yz)$$
  - ➔ für alle  $z \in \Sigma^*$  gilt:  
$$xz \in L \iff yz \in L$$
  - ➔  $x \sim_L y$
- Damit ist der Beweis des Satzes von Myhill und Nerode vollständig

## Satz von Myhill und Nerode: Anwendung (1/2)

- Mit dem Satz von Myhill und Nerode lässt sich also herausfinden, ob eine gegebene Sprache regulär ist:
  - Wir haben gesehen, dass die Relation  $\sim_{L_g}$  vier Klassen hat
  - $L_g$  ist also regulär
- Der in Kapitel 2 konstruierte Automat  $\mathcal{A}_g$  ist sogar im Wesentlichen der Äquivalenzklassenautomat zu  $L_g$   
 (bis auf Isomorphie, siehe später)
- Der Satz ist aber auch für den Nachweis, dass eine gegebene Sprache **nicht** regulär ist, nützlich

## Satz von Myhill und Nerode: Anwendung (2/2)

### Beispiel

- Wir berechnen die Äquivalenzklassen von  $L_{ab} = \{a^n b^n \mid n \geq 0\}$
- Es gilt z.B.:  

$$a^4 b \sim_{L_{ab}} a^5 b^2 \sim_{L_{ab}} a^6 b^3 \dots$$
- $\sim_{L_{ab}}$  hat die Klassen:
  - $B_k \stackrel{\text{def}}{=} \{a^{i+k} b^i \mid i \geq 1\}$ ,  
für jedes  $k \geq 0$ ,
  - $A_k \stackrel{\text{def}}{=} \{a^k\}$ , für jedes  $k \geq 0$ ,
  - $C \stackrel{\text{def}}{=} \{a^i b^j \mid i < j\} \cup \overline{L(a^* b^*)}$ ,  
die Klasse aller Strings, für die es überhaupt keine Verlängerung gibt, die in  $L_{ab}$  liegt:

- Notation:
  - Sei  $L$  eine Sprache über  $\Sigma$  und  $v \in \Sigma^*$
  - $\underline{L/v} \stackrel{\text{def}}{=} \{z \in \Sigma^* \mid vz \in L\}$

### Beispiel (Forts.)

- Um nachzuweisen, dass dies die Äquivalenzklassen von  $\sim_{L_{ab}}$  sind, ist zu zeigen:
  - (1) Jeder String kommt in einer Klasse vor ✓
  - (2) Für alle Strings  $u, v$  in derselben Klasse gilt:  

$$u \sim_{L_{ab}} v$$
  - (3) Für Strings  $u, v$  aus verschiedenen Klassen gilt:  

$$u \not\sim_{L_{ab}} v$$

- Dazu genügt es zu zeigen, dass
  - (2') für alle Strings  $v$  einer Klasse die Menge  $L_{ab}/v$  gleich ist
  - (3') für verschiedene Klassen die Mengen  $L_{ab}/v$  verschieden sind

- Für jedes  $k$  gilt:
  - Für  $v \in B_k$  ist  $L_{ab}/v = \{b^k\}$
  - Für  $v \in A_k$  ist  $L_{ab}/v = \{a^i b^{i+k} \mid i \geq 0\}$
- Für  $v \in C$  ist  $L_{ab}/v = \emptyset$

- ➡ Die Äquivalenzklassen sind korrekt angegeben
- ➡ unendlich viele Klassen  $\Rightarrow L_{ab}$  nicht regulär

# Minimaler Automat: Eindeutigkeit (1/3)


- Was bringt uns der Satz von Myhill und Nerode für die Minimierung von Automaten?
- Aus dem Beweis können wir direkt schließen:

## Lemma 4.2

- Ist  $L$  eine reguläre Sprache und  $\mathcal{A} = (Q, \Sigma, \delta, s, F)$  ein DFA für  $L$ , dann gilt:  
$$|Q| \geq \text{Anzahl Klassen von } \sim_L$$

- Also: Jeder Automat für  $L$  hat mindestens so viele Zustände wie der Äquivalenzklassenautomat  $\mathcal{A}_L$
- $\mathcal{A}_L$  ist also **ein** minimaler Automat für  $L$

- Wir zeigen gleich:
  - In einem gewissen Sinne ist  $\mathcal{A}_L$  sogar in jedem Automaten für  $L$  enthalten
  - Und: falls  $|Q|$  gleich der Anzahl der Klassen von  $\sim_L$  ist, sind  $\mathcal{A}$  und  $\mathcal{A}_L$  „praktisch identisch“
- $\mathcal{A}_L$  ist also **der** minimale Automat für  $L$

 Wir betrachten zuerst die Formalisierung von „praktisch identisch“: Isomorphie von DFAs

## Minimaler Automat: Eindeutigkeit (2/3)

### Definition

- Seien  $\mathcal{A}_1 = (Q_1, \Sigma, \delta_1, s_1, F_1)$  und  $\mathcal{A}_2 = (Q_2, \Sigma, \delta_2, s_2, F_2)$  DFAs mit dem selben Eingabe-Alphabet
- $\mathcal{A}_1$  und  $\mathcal{A}_2$  sind **isomorph**, falls es eine Bijektion  $\pi : Q_1 \rightarrow Q_2$  gibt mit:
  - (1)  $\pi(s_1) = s_2$ ,
  - (2) für alle  $q \in Q_1$  gilt:
$$q \in F_1 \iff \pi(q) \in F_2, \text{ und}$$
  - (3) für alle  $q \in Q_1$  und  $\sigma \in \Sigma$  gilt:
$$\pi(\delta_1(q, \sigma)) = \delta_2(\pi(q), \sigma)$$
- Notation:  $\mathcal{A}_1 \cong \mathcal{A}_2$
- Informell bedeutet  $\mathcal{A}_1 \cong \mathcal{A}_2$ : ⊕
  - Die DFAs unterscheiden sich nur hinsichtlich der Namen der Zustände:
    - \* Wenn in  $\mathcal{A}_1$  die Zustände gemäß  $\pi$  umbenannt werden, ergibt sich  $\mathcal{A}_2$



## Minimaler Automat: Eindeutigkeit (3/3)

### Lemma 4.3

- Ist  $\mathcal{A}$  ein Automat für eine Sprache  $L$ , der die selbe Anzahl von Zuständen wie  $\mathcal{A}_L$  hat, so gilt:  
 $\mathcal{A} \cong \mathcal{A}_L$
- Der Beweis findet sich im Anhang

# Minimalautomat

- Insgesamt haben wir bisher gezeigt:

## Satz 4.4

- Für jede reguläre Sprache  $L$  ist  $\mathcal{A}_L$  der bis auf Isomorphie eindeutig bestimmte minimale Automat für  $L$

## Beweisskizze

- Nach Lemma 4.2 hat jeder Automat für  $L$  mindestens so viele Zustände wie  $\mathcal{A}_L$
- Nach 4.3 ist jeder Automat für  $L$ , der genau so viele Zustände wie  $\mathcal{A}_L$  hat, isomorph zu  $\mathcal{A}_L$
- Wir betrachten jetzt, wie sich  $\mathcal{A}_L$  aus einem gegebenen Automaten für  $L$  berechnen lässt

# Inhalt

4.1 Satz von Myhill und Nerode

▷ **4.2 Minimierungsalgorithmus für DFAs**

# Minimaler Automat: Berechnung

- Wie lässt sich  $\mathcal{A}_L$  konstruieren?

- Auch hierfür liefert Satz 4.1 einen Hinweis:

- Ist  $\mathcal{A}$  ein Automat für  $L$ , so ist  $\sim_{\mathcal{A}}$  eine Verfeinerung von  $\sim_L$

➡  $\mathcal{A}_L$  kann durch Zusammenlegen von Zuständen (Äquivalenzklassen) aus  $\mathcal{A}$  erzeugt werden

- Genauer: zwei Zustände  $p, q$  von  $\mathcal{A}$  können zusammengelegt werden, wenn sie im folgenden Sinne **äquivalent** sind:

(\*\*) für alle  $z \in \Sigma^*$  gilt:

$$\delta^*(p, z) \in F \iff \delta^*(q, z) \in F$$

➡ Um den minimalen Automaten zu konstruieren, genügt es also, zu berechnen, welche Zustände zusammen gelegt werden können

- Es ist allerdings algorithmisch einfacher, zunächst zu berechnen, welche Zustände **nicht** zusammen gelegt werden können

- Deshalb betrachten wir jetzt einen Algorithmus, der die Menge  $N(\mathcal{A})$  der **nicht äquivalenten Paare** berechnet:

$$\underline{N(\mathcal{A})} \stackrel{\text{def}}{=} \{(p, q) \mid p, q \in Q, \\ \exists w : \delta^*(p, w) \in F \not\iff \delta^*(q, w) \in F\}$$

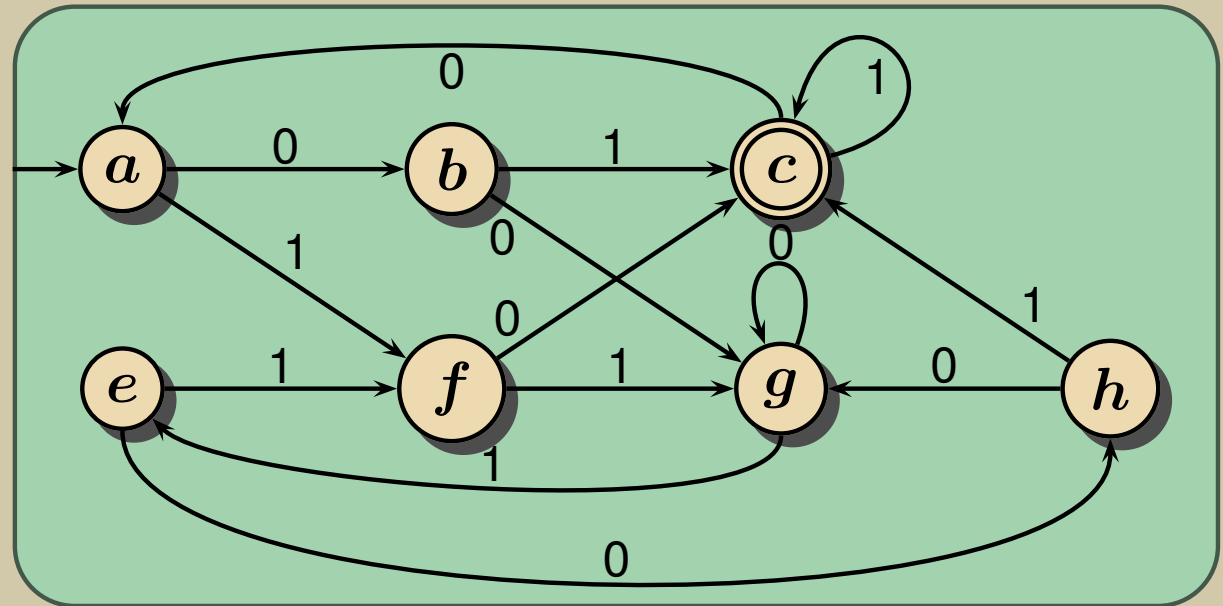
# Der Markierungsalgorithmus

## Markierungsalgorithmus

- Eingabe:  
 $\mathcal{A} = (Q, \Sigma, \delta, s, F)$
- Ausgabe: Relation  $N(\mathcal{A})$

- $M := \{(p, q), (q, p) \mid p \in F, q \notin F\}$
- $M' := \{(p, q) \notin M \mid \exists \sigma \in \Sigma : (\delta(p, \sigma), \delta(q, \sigma)) \in M\}$
- $M := M \cup M'$
- Falls  $M' \neq \emptyset$ , weiter mit 2.
- Ausgabe  $M$

## Beispiel



|          | <i>a</i> | <i>b</i>       | <i>c</i>       | <i>e</i>       | <i>f</i>       | <i>g</i>       | <i>h</i>       |
|----------|----------|----------------|----------------|----------------|----------------|----------------|----------------|
| <i>a</i> |          | x <sup>1</sup> | x <sup>0</sup> |                | x <sup>1</sup> | x <sup>2</sup> | x <sup>1</sup> |
| <i>b</i> |          |                | x <sup>0</sup> | x <sup>1</sup> | x <sup>1</sup> | x <sup>1</sup> |                |
| <i>c</i> |          |                |                | x <sup>0</sup> | x <sup>0</sup> | x <sup>0</sup> | x <sup>0</sup> |
| <i>e</i> |          |                |                |                | x <sup>1</sup> | x <sup>2</sup> | x <sup>1</sup> |
| <i>f</i> |          |                |                |                |                | x <sup>1</sup> | x <sup>1</sup> |
| <i>g</i> |          |                |                |                |                |                | x <sup>1</sup> |
| <i>h</i> |          |                |                |                |                |                |                |

Nicht markiert:  $(a, e), (b, h)$

# Markierungsalgorithmus: Korrektheit (1/2)

## Lemma 4.5

- Der Markierungsalgorithmus berechnet  $N(\mathcal{A})$

## Beweisskizze

- Zur Erinnerung:  
$$N(\mathcal{A}) = \{(p, q) \mid p, q \in Q, \exists w : \delta^*(p, w) \in F \not\equiv \delta^*(q, w) \in F\}$$
- Wir zeigen:
  - (a) Wenn  $(p, q)$  im  $k$ -ten Durchlauf (von 2.) markiert wird,  
dann gibt es einen String  $w$  der Länge  $k$  mit
$$\delta^*(p, w) \in F \not\equiv \delta^*(q, w) \in F$$
  - (b) Wenn  $(p, q)$  durch den Algorithmus nicht markiert wird,  
dann gilt für alle Strings  $w \in \Sigma^*$ :
$$\delta^*(p, w) \in F \iff \delta^*(q, w) \in F$$

## Beweisskizze für (a)

- Beweis durch Induktion nach  $k$ 
    - $k = 0 \checkmark$
    - Von  $k - 1$  zu  $k$ :
      - Zu jedem Paar  $(p, q)$ , das im  $k$ -ten Durchlauf markiert wird, gibt es ein Paar  $(p', q')$ , das im  $(k - 1)$ -ten Durchlauf markiert wird mit
$$\delta(p, \sigma) = p', \delta(q, \sigma) = q'$$
      - Nach Induktion gibt es also einen String  $v$  der Länge  $k - 1$  mit
$$\delta^*(p', v) \in F \not\equiv \delta^*(q', v) \in F$$
- $\Rightarrow \delta^*(p, \sigma v) \in F \not\equiv \delta^*(q, \sigma v) \in F$

# Markierungsalgorithmus: Korrektheit (2/2)

## Lemma 4.5

- Der Markierungsalgorithmus berechnet  $N(\mathcal{A})$

## Beweisskizze


- Zur Erinnerung:  
$$N(\mathcal{A}) = \{(p, q) \mid p, q \in Q, \exists w : \delta^*(p, w) \in F \not\leftrightarrow \delta^*(q, w) \in F\}$$
- Wir zeigen:
  - (a) **Wenn**  $(p, q)$  im  $k$ -ten Durchlauf (von 2.) markiert wird,  
**dann** gibt es einen String  $w$  der Länge  $k$  mit
$$\delta^*(p, w) \in F \not\leftrightarrow \delta^*(q, w) \in F$$
  - (b) **Wenn**  $(p, q)$  durch den Algorithmus nicht markiert wird,  
**dann** gilt für alle Strings  $w \in \Sigma^*$ :
$$\delta^*(p, w) \in F \iff \delta^*(q, w) \in F$$

## Beweisskizze für (b)

- Beweis durch Widerspruch:
  - Angenommen, es gibt ein Gegenbeispiel  $(p, q, w)$ , so dass
    - \* der Algorithmus  $(p, q)$  nicht markiert, aber
    - \*  $\delta^*(p, w) \in F \not\leftrightarrow \delta^*(q, w) \in F$
  - \* Sei  $(p, q, w)$  ein Gegenbeispiel mit dem kürzest möglichen  $w$
  - \* Klar:  $w \neq \epsilon$  wegen Schritt (1) des Alg.
  - \* Seien  $v \in \Sigma^*$ ,  $\sigma \in \Sigma$  mit  $w = \sigma v$
  - \* Da  $(p, q)$  unmarkiert ist, ist auch  $(\delta(p, \sigma), \delta(q, \sigma))$  unmarkiert
  - ➡  $(\delta(p, \sigma), \delta(q, \sigma), v)$  ist auch ein Gegenbeispiel, aber  $v$  ist kürzer als  $w$
  - \* **Widerspruch zur Wahl von**  $(p, q, w)$

# Wiederholung: $\mathcal{O}$ -Notation

## Definition

- Seien  $f, g : \mathbb{N} \rightarrow \mathbb{R}$  zwei Funktionen
- $f = \mathcal{O}(g)$ : es gibt  $c, d$ , so dass für alle  $n \geq d$  gilt:  $f(n) \leq cg(n)$  

## Beispiel

- $12n \log n = \mathcal{O}(n^2)$
- $5n^3 + 12n^2 + 17n + 100 = \mathcal{O}(n^3)$

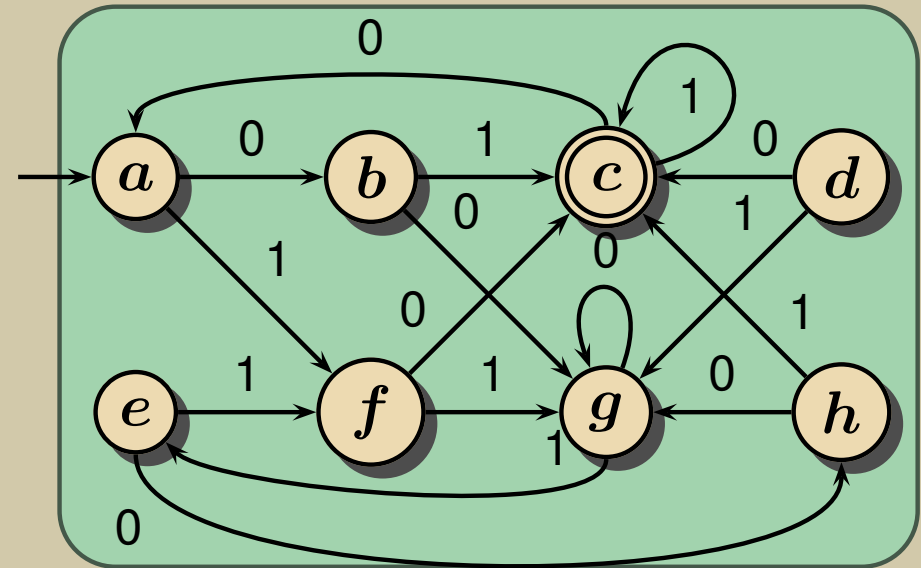


# Minimierungsalgorithmus

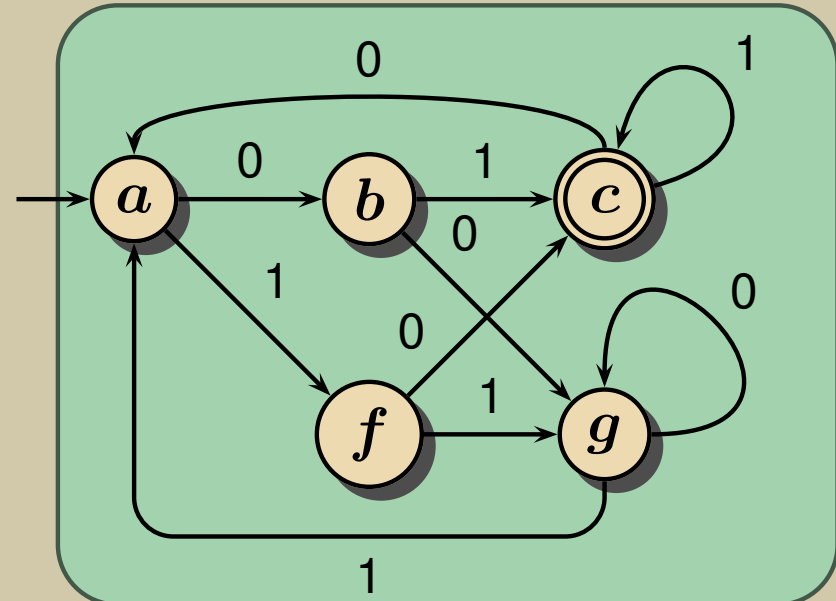
## Minimierungsalgorithmus für DFA

- Eingabe:  $\mathcal{A} = (Q, \Sigma, \delta, s, F)$
  - Ausgabe: minimaler Automat  $\mathcal{A}'$  mit  $L(\mathcal{A}') = L(\mathcal{A})$
1. Entferne alle Zustände von  $\mathcal{A}$ , die von  $s$  aus nicht erreichbar sind.
  2. Berechne die Relation  $N(\mathcal{A})$  mit dem Markierungs-Algorithmus
  3. Verschmelze sukzessive alle nicht markierten Zustandspaare zu jeweils einem Zustand.
- Laufzeit des Minimierungsalgorithmus:
    1.  $\mathcal{O}(|\delta|) = \mathcal{O}(|Q|^2|\Sigma|)$   
👉 nächstes Kapitel
    2.  $\mathcal{O}(|Q|^2|\Sigma|)$  bei geschickter Implementierung
    3.  $\mathcal{O}(|Q|^2|\Sigma|)$Zusammen:  $\mathcal{O}(|Q|^2|\Sigma|)$

## Beispiel



## Minimaler Automat:



## Vom RE zum DFA: vollständig

- Damit kennen wir nun alle Teilschritte von der Spezifikation einer regulären Sprache bis zur Berechnung eines möglichst kleinen endlichen Automaten

1. Spezifiziere die Sprache durch einen regulären Ausdruck  $\alpha$
2. Wandle  $\alpha$  in einen  $\epsilon$ -NFA  $\mathcal{A}_1$  um
3. Wandle  $\mathcal{A}_1$  in einen DFA  $\mathcal{A}_2$  um
4. Wandle  $\mathcal{A}_2$  in einen minimalen DFA  $\mathcal{A}_3$  um

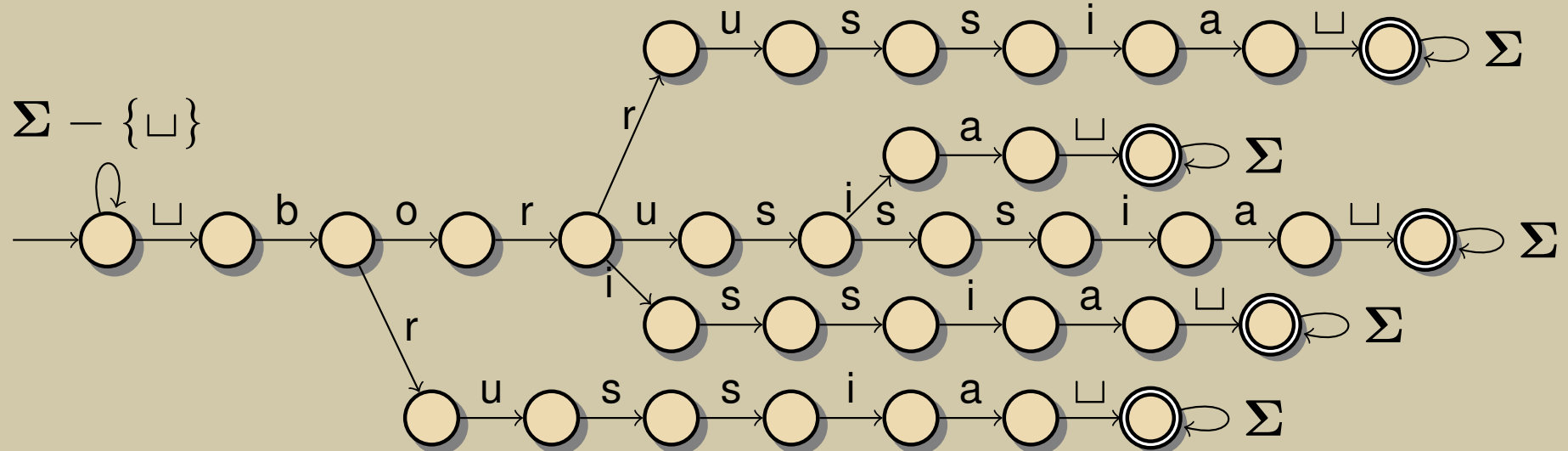


Der e-Mail-Adressen-DFA ist übrigens schon minimal...

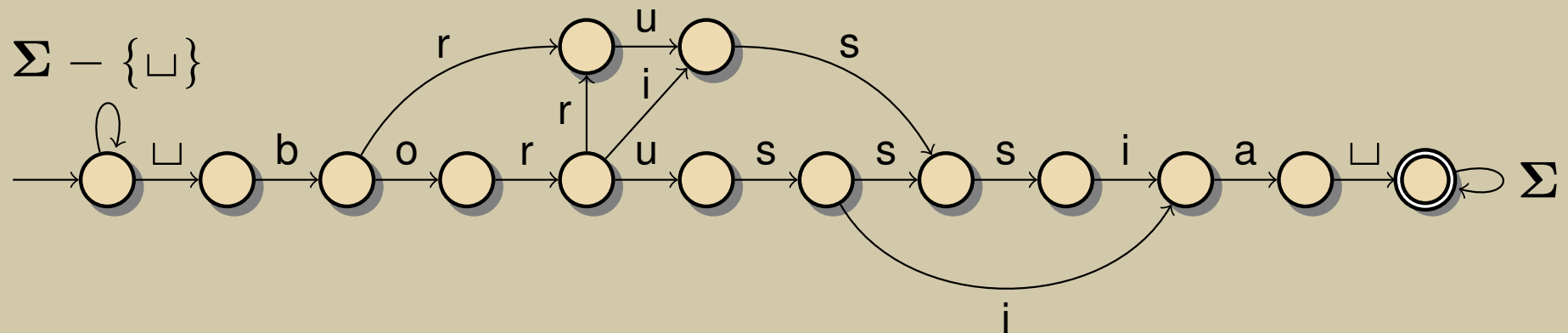
# Der Borussia-Newsticker-Automat (3/3)

## Beispiel

- Ist der Borussia-Automat minimal?



- Nein, dies ist der minimale DFA:

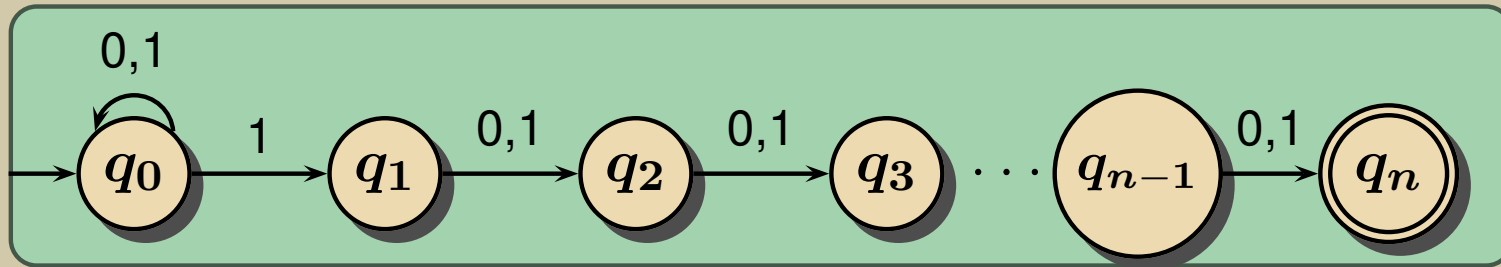


# Satz von Myhill und Nerode: weitere Anwendung

- Der Satz von Myhill und Nerode liefert auch eine Methode um die Größe des Minimalautomaten für eine reguläre Sprache zu berechnen:
  - Zähle die Klassen von  $\sim_L$

## Beispiel

- Wir betrachten wieder die Sprache  $L_n$  aller 0-1-Strings, deren  $n$ -tes Zeichen von rechts eine 1 ist:



- Es ist leicht zu zeigen, dass zwei Strings  $x, y$  genau dann in derselben Äquivalenzklasse von  $\sim_{L_n}$  sind, wenn sie dasselbe Suffix der Länge  $n$  haben

 Dabei werden bei Strings der Länge  $< n$  führende Nullen „hinzugedacht“

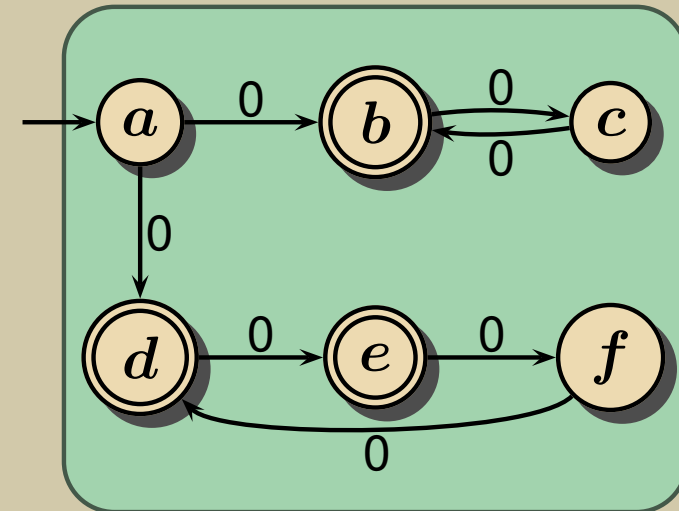
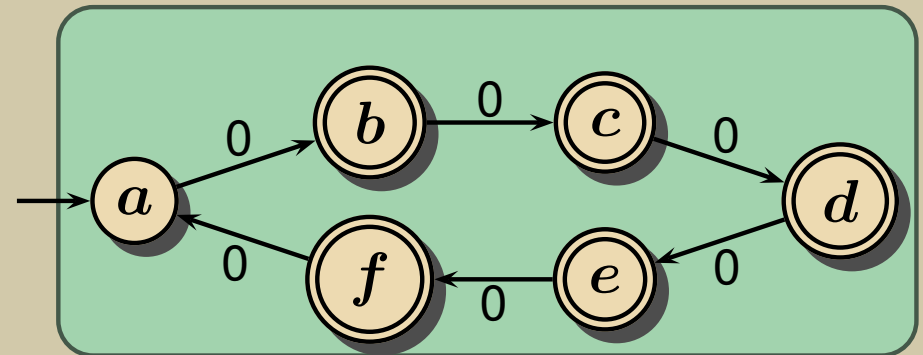
- ➡ Es gibt soviele Klassen in  $\sim_L$  wie es 0-1-Strings der Länge  $n$  gibt
- ➡  $\sim_{L_n}$  hat  $2^n$  Klassen
- ➡ Jeder Automat für  $L_n$  hat mindestens  $2^n$  Zustände

# Minimale NFAs

- Es gibt zwar auch zu jedem NFA  $\mathcal{A}$  einen kleinsten NFA  $\mathcal{A}'$  mit  $L(\mathcal{A}') = L(\mathcal{A})$
- Aber der kleinste NFA ist im Allgemeinen nicht bis auf Isomorphie eindeutig

## Beispiel

- Die Menge aller Strings der Form  $0^n$ , für die 6 kein Teiler von  $n$  ist, hat zwei kleinste NFAs:



- Idee: 6 ist genau dann kein Teiler von  $n$  wenn 2 oder 3 kein Teiler von  $n$  ist

## Rot, Gelb, Grün

- Minimale Automaten haben höchstens einen Zustand, von dem aus kein akzeptierender Zustand mehr erreichbar ist
  - Dieser Zustand entspricht der roten Ampel im Webformular-Beispiel
  - Akzeptierende Zustände entsprechen der grünen Ampel
  - Alle übrigen Zustände entsprechen der gelben Ampel
- Wir nennen diesen Zustand manchmal Fehlerzustand (im Englischen: *sink state*)
- Dass ein solche Zustand nicht immer existiert, ist schon am ersten Beispielautomaten von Kapitel 2 zu sehen.

# Zusammenfassung

- Zu einem gegebenen DFA ist der minimale äquivalente DFA bis auf Isomorphie eindeutig bestimmt und kann mit Hilfe des Markierungsalgorithmus in Zeit  $\mathcal{O}(|Q|^2|\Sigma|)$  berechnet werden
- **Literatur:**
  - John R. Myhill. Finite automata and the representation of events. Technical Report WADC TR-57-624, Wright-Paterson Air Force Base, 1957
  - A. Nerode. Linear automaton transformations. *Proc. Amer. Math. Soc.*, 9:541–544, 1958

# Erläuterungen

## Bemerkung 4.1

- Die Notation  $f = \mathcal{O}(g)$  kann leicht zu Missverständnissen führen:
  - Denn: das Gleichheitszeichen wird hier, anders als sonst üblich, in einem nicht symmetrischen Sinn verwendet
  - Es gilt z.B.
    - \*  $n^2 = \mathcal{O}(n^3) = \mathcal{O}(n^4)$ ,
    - \* aber nicht  $n^2 = \mathcal{O}(n^4) = \mathcal{O}(n^3)$
- Sauberer ist es, (wie in Mafl)  $\mathcal{O}(g)$  als Notation für eine Menge von Funktionen (oder Folgen) zu betrachten:
  - $\mathcal{O}(g) \stackrel{\text{def}}{=} \{h \mid \exists c, d \ \forall n \geq d : h(n) \leq cg(n)\}$
  - Schreibweise dann:  $f \in \mathcal{O}(g)$
- Aber: Wir folgen hier der Tradition...



# Minimaler Automat: Eindeutigkeit (3/3)

## Lemma 4.3

- Ist  $\mathcal{A}$  ein Automat für eine Sprache  $L$ , der die selbe Anzahl von Zuständen wie  $\mathcal{A}_L$  hat, so gilt:  $\mathcal{A} \cong \mathcal{A}_L$

## Beweisidee

- Sei  $\mathcal{A} = (Q, \Sigma, \delta, s, F)$  ein solcher Automat
- $\mathcal{A}$  minimal  $\Rightarrow$   
in  $\mathcal{A}$  sind alle Zustände erreichbar
- $\Rightarrow$  für jeden Zustand  $q$  von  $\mathcal{A}$  gibt es einen String  $w_q$  mit  $\delta^*(s, w_q) = q$
- Wir definieren eine Abbildung  $\pi$  durch:  $\pi(q) \stackrel{\text{def}}{=} [w_q]$
- Behauptung:**  
 $\pi$  ist ein Isomorphismus von  $\mathcal{A}$  auf  $\mathcal{A}_L$

## Beweisdetails

- (1)  $\pi(s) = [\epsilon] \checkmark$
- (2)  $q \in F \iff w_q \in L \iff \pi(q) = [w_q]$  ist akzeptierender Zustand von  $\mathcal{A}_L$
- (3) Für  $q \in Q, \sigma \in \Sigma$  gilt:
$$\begin{aligned}\pi(\delta(q, \sigma)) &= \pi(\delta(\delta^*(s, w_q), \sigma)) \\ &= \pi(\delta^*(s, w_q \sigma)) \\ &= [w_q \sigma] \\ &= \delta'([w_q], \sigma) \\ &= \delta'(\pi(q), \sigma)\end{aligned}$$

  $\delta'$  bezeichnet die Überföhrungsfunktion von  $\mathcal{A}_L$

- $\pi$  ist bijektiv, denn:
  - Aus dem Beweis von Satz 4.1 folgt:  
 $\sim_{\mathcal{A}}$  ist eine Verfeinerung von  $\sim_L$
  - Da  $\sim_{\mathcal{A}}$  und  $\sim_L$  gleich viele Klassen haben gilt also:  
 $\sim_{\mathcal{A}} = \sim_L$
- $\Rightarrow \pi$  ist eine Bijektion

- Also ist  $\pi$  ein Isomorphismus und es folgt:  $\mathcal{A} \cong \mathcal{A}_L$

# Grundbegriffe der Theoretischen Informatik

Sommersemester 2016 - Thomas Schwentick

Teil A: Reguläre Sprachen

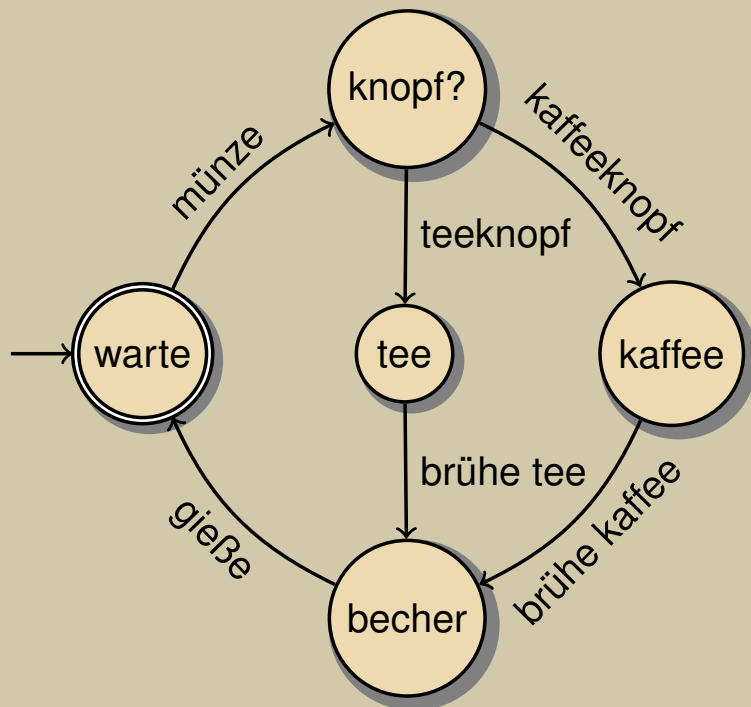
5: Abschlusseigenschaften, Grenzen und Algorithmen

Version von: 3. Mai 2016 (13:03)

## Einleitung (1/2)

- Das Verhalten von vielen praktischen Systemen kann durch DFAs oder NFAs abstrahiert/beschrieben werden

### Kaffeemaschine als DFA



- Oft möchten wir Eigenschaften solcher Systeme **automatisch überprüfen**

### Beispiel

- Die Maschine soll nur Kaffee ausgießen, wenn (seit dem letzten Kaffee) eine Münze eingeworfen wurde
- Diese Eigenschaft lässt sich durch einen RE ausdrücken:

$$R \stackrel{\text{def}}{=} (S^* \text{ münze } S^* \text{ gieße } S^*)^* \\ \text{(Mit } S = \Sigma - \{\text{münze, gieße}\})$$

- Lässt sich automatisch überprüfen, ob der Automat die Eigenschaft  $R$  erfüllt?
- Formal führt das zur Frage: Ist  $L(A) \subseteq L(R)$ ?
  - Äquivalent: Ist  $L(A) \cap (\Sigma^* - L(R)) = \emptyset$ ?
- Es wäre also praktisch, eine Toolbox für Automaten zu haben
  - Schnitt, Vereinigung, Komplement, etc.
  - Testalgorithmen...

## Einleitung (2/2)

- Wir setzen in diesem Kapitel die Untersuchung der Klasse der regulären Sprachen fort
- Wir betrachten
  - algorithmische Methoden, mit denen sich reguläre Sprachen kombinieren und modifizieren lassen
  - eine weitere, einfache Methode zum Nachweis, dass eine Sprache nicht regulär ist
  - Algorithmen zum Testen von Eigenschaften einer durch einen Automaten gegebenen Sprache

# Inhalt


- ▷ **5.1 Abschlusseigenschaften und Synthese von Automaten**
- 5.2 Grenzen der Regulären Sprachen
- 5.3 Weitere Algorithmen für Automaten

# Größenmaße für Automaten

- Der Aufwand der folgenden Algorithmen wird in Abhängigkeit von der Größe der Eingabe beschrieben
- Wenn Die Eingabe aus Automaten besteht, stellt sich also die Frage:
  - **Wie „groß“ ist ein endlicher Automat  $\mathcal{A} = (Q, \Sigma, \delta, s, F)$  ?**
- Es können verschiedene Größenmaße definiert werden:
  - Anzahl der Zustände:  $|Q|$
  - Anzahl der Transitionen:  $|\delta|$
  - Größe der Kodierung des Automaten als Bitstring
- Wir verwenden hier nur die ersten beiden Maße und geben das verwendete Maß jeweils explizit an

- Für reguläre Ausdrücke  $\alpha$  bezeichnet  $|\alpha|$  einfach die Länge des Strings

– Also:  $|(ab)^*c(d + \epsilon)| = 11$


 Ob Konkatinationssymbole und Klammern für die Größe eines RE gezählt werden, ist für das Folgende nicht so wichtig

- Wir interessieren uns nur für asymptotische Abschätzungen
- $|\alpha|$  (nach unserer Definition) ist linear in der Anzahl der Symbolvorkommen und der Vorkommen des \*-Operators (sofern doppelte Klammerpaare (...)) nicht vorkommen)

# Synthese endlicher Automaten: Boolesche Operationen (1/3)

- Wir betrachten jetzt, auf welche Weisen aus regulären Sprachen neue reguläre Sprachen gewonnen werden können

- Uns interessiert also:
  - Unter welchen Operationen ist die Klasse der regulären Sprachen abgeschlossen ist?
  - Mit welchen Algorithmen lassen sich solche Operationen ausführen?

 Wichtig: es geht im Folgenden **nicht** um Abschlusseigenschaften von einzelnen Sprachen sondern um Abschlusseigenschaften der Klasse aller regulären Sprachen

- Wir beginnen mit Booleschen Operationen

- Reguläre Ausdrücke haben einen Operator für die Vereinigung

➡ Die **Vereinigung** zweier regulärer Sprachen ist regulär

- Um reguläre Sprachen algorithmisch gut „verarbeiten“ zu können, ist es wichtig, dass auch der **Durchschnitt** zweier regulärer Sprachen und das **Komplement** einer regulären Sprache wieder regulär sind

- Außerdem sollten diese Booleschen Operationen auf der Ebene endlicher Automaten möglichst effizient ausgeführt werden können

# Synthese endlicher Automaten: Boolesche Operationen (2/3)

## Satz 5.1

- Seien  $\mathcal{A}_1 = (Q_1, \Sigma, \delta_1, s_1, F_1)$  und  $\mathcal{A}_2 = (Q_2, \Sigma, \delta_2, s_2, F_2)$  DFAs
- Dann lassen sich Automaten für die folgenden Sprachen konstruieren:
  - (a) für  $L(\mathcal{A}_1) \cap L(\mathcal{A}_2)$   
mit  $|Q_1||Q_2|$  Zuständen in Zeit  $\mathcal{O}(|Q_1||Q_2||\Sigma|)$
  - (b) für  $L(\mathcal{A}_1) \cup L(\mathcal{A}_2)$   
mit  $|Q_1||Q_2|$  Zuständen in Zeit  $\mathcal{O}(|Q_1||Q_2||\Sigma|)$
  - (c) für  $\Sigma^* - L(\mathcal{A}_1)$  mit  $|Q_1|$  Zuständen in Zeit  $\mathcal{O}(|Q_1|)$

## Folgerung 5.2

- Die regulären Sprachen sind unter Durchschnitt, Vereinigung und Komplementbildung abgeschlossen

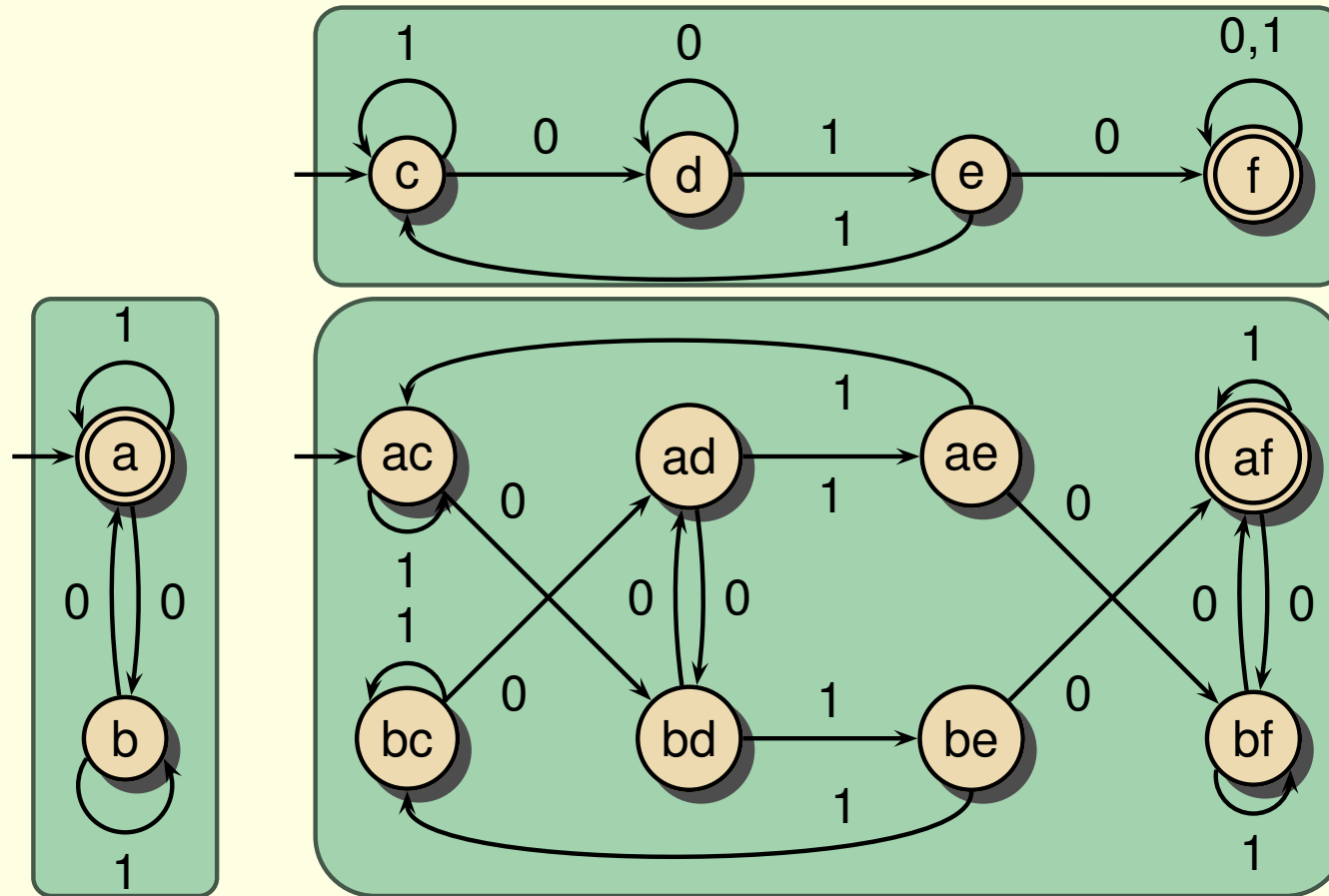
 (a) und (b) gelten auch für NFAs

- Für den Beweis von (a) und (b) verwenden wir das Konzept des **Produktautomaten**



## Produktautomat: Beispiel

- Ein Automat für die Menge aller Strings, die 010 als Teilstring enthalten **und** gerade viele Nullen haben:




0110100

- Um einen Automaten für die oben genannte Sprache zu erhalten, muss af **als akzeptierender Zustand** gewählt werden

# Synthese endlicher Automaten: Boolesche Operationen (3/3)

## Definition

- Seien  $\mathcal{A}_1 = (Q_1, \Sigma, \delta_1, s_1, F_1)$ ,  $\mathcal{A}_2 = (Q_2, \Sigma, \delta_2, s_2, F_2)$  DFAs
- Sei  $F \subseteq Q_1 \times Q_2$
- Der **Produktautomat zu  $\mathcal{A}_1$  und  $\mathcal{A}_2$**  mit akzeptierender Menge  $F$  ist der Automat  $\mathcal{B} \stackrel{\text{def}}{=} (Q_1 \times Q_2, \Sigma, \delta_{\mathcal{B}}, (s_1, s_2), F)$ , wobei  $\delta_{\mathcal{B}}$  komponentenweise definiert ist, d.h.:
  - Für  $q_1 \in Q_1$  und  $q_2 \in Q_2$  sei
$$\delta_{\mathcal{B}}((q_1, q_2), \sigma) \stackrel{\text{def}}{=} (\delta_1(q_1, \sigma), \delta_2(q_2, \sigma))$$

 Wir schreiben manchmal  $\mathcal{A}_1 \times \mathcal{A}_2$  für den Produktautomaten, ohne eine akzeptierende Zustandsmenge zu spezifizieren

## Beweis von Satz 5.1

- Wir beweisen zunächst Teil (a): Durchschnitt
- Sei  $\mathcal{B}$  der Produktautomat zu  $\mathcal{A}_1$  und  $\mathcal{A}_2$  mit akzeptierender Menge  $F_1 \times F_2$
- Durch Induktion lässt sich leicht zeigen, dass für alle  $w \in \Sigma^*$  gilt:
$$\delta_{\mathcal{B}}^*((s_1, s_2), w) = (\delta_1^*(s_1, w), \delta_2^*(s_2, w))$$
- Es folgt:
$$\begin{aligned} w \in L(\mathcal{A}_1) \cap L(\mathcal{A}_2) &\iff \delta_1^*(s_1, w) \in F_1 \text{ und } \delta_2^*(s_2, w) \in F_2 \\ &\iff (\delta_1^*(s_1, w), \delta_2^*(s_2, w)) \in F_1 \times F_2 \\ &\iff \delta_{\mathcal{B}}^*((s_1, s_2), w) \in F_1 \times F_2 \\ &\iff w \in L(\mathcal{B}) \end{aligned}$$
- Teil (b) kann analog bewiesen werden, mit akzeptierender Menge  $F_1 \times Q_2 \cup Q_1 \times F_2$
- Teil (c) ist noch einfacher: Wähle  $(Q_1, \Sigma, \delta_1, s_1, Q_1 - F_1)$  als DFA

# Zum Verständnis des Produktautomaten

PINGO-Frage: [pingo.upb.de](http://pingo.upb.de)

- Wie muss die akzeptierende Menge  $F$  des Produktautomaten gewählt werden, damit er die Menge aller Strings akzeptiert, die von einem der Automaten  $\mathcal{A}_1$  und  $\mathcal{A}_2$  akzeptiert wird, aber nicht vom anderen?

(A)  $(F_1 - F_2) \cup (F_2 - F_1)$

(B)  $(Q_1 \times Q_2) - (F_1 \times F_2)$

(C)  $(Q_1 \times F_2) \cup (F_1 \times Q_2)$

(D)  $(Q_1 \times (Q_2 - F_2)) \cup ((Q_1 - F_1) \times Q_2)$

(E)  $(F_1 \times (Q_2 - F_2)) \cup ((Q_1 - F_1) \times F_2)$

# Synthese endlicher Automaten: Konkatenation und Iteration

- Die Definition regulärer Ausdrücke garantiert auch den Abschluss unter Konkatenation und Stern

## Satz 5.3

- Seien  $\mathcal{A}_1$  und  $\mathcal{A}_2$  DFAs (oder NFAs) für Sprachen  $L_1$  und  $L_2$
- Dann lassen sich NFAs (oder DFAs) für  $L_1 \circ L_2$  und  $L_1^*$  konstruieren

## Beweisidee

- Ein DFA für  $L_1 \circ L_2$  kann in zwei Schritten gewonnen werden:
  1. Verknüpfung von  $\mathcal{A}_1$  und  $\mathcal{A}_2$  durch  $\epsilon$ -Übergänge von der akzeptierenden Zuständen von  $\mathcal{A}_1$  zum Startzustand von  $\mathcal{A}_2$ 
    - Wie bei der Umwandlung von REs in  $\epsilon$ -NFAs
  2. Determinisierung des entstandenen  $\epsilon$ -NFAs
- $L_1^*$ : analog

# Abschlusseigenschaften: Homomorphismen (1/3)

- Wir betrachten nun weitere Abschlusseigenschaften der Klasse der regulären Sprachen, die vor allem für theoretische Zwecke hilfreich sind:
  - Abschluss unter Homomorphismen
  - Abschluss unter inversen Homomorphismen

## Definition

- Eine Funktion  $h : \Sigma^* \rightarrow \Gamma^*$  ist ein **Homomorphismus**, wenn für alle Strings  $u, v \in \Sigma^*$  gilt:  $h(uv) = h(u)h(v)$
- Aus der Definition folgt:  $h(\epsilon) = \epsilon$
- Zur Definition eines Homomorphismus von  $\Sigma^*$  nach  $\Gamma^*$  genügt es,  $h(\sigma)$  für alle  $\sigma \in \Sigma$  festzulegen
- Dadurch ist  $h(w)$  auch für beliebige Strings  $w = \sigma_1 \cdots \sigma_n$  eindeutig festgelegt:  
$$h(w) = h(\sigma_1) \cdots h(\sigma_n)$$

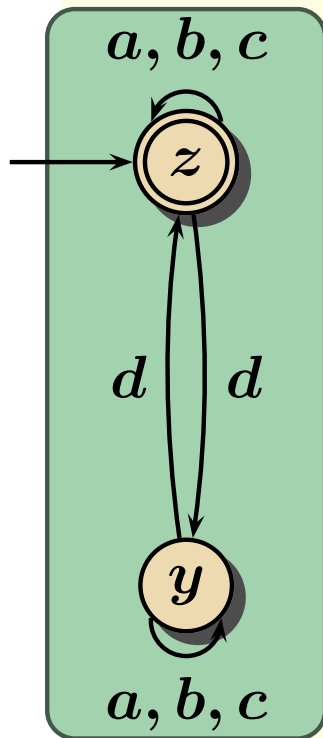
## Beispiel

- $h : \{a, b, c, d\}^* \rightarrow \{0, 1\}^*$
- Definiert durch:
  - $h(a) \stackrel{\text{def}}{=} 00$
  - $h(b) \stackrel{\text{def}}{=} 1$
  - $h(c) \stackrel{\text{def}}{=} \epsilon$
  - $h(d) \stackrel{\text{def}}{=} 0110$
- Dann:  $h(abdc) = 0010110$
- Für  $L \subseteq \Sigma^*$  sei  
$$\underline{h(L)} \stackrel{\text{def}}{=} \{h(w) \mid w \in L\}$$
- Für  $L \subseteq \Gamma^*$  sei  
$$\underline{h^{-1}(L)} \stackrel{\text{def}}{=} \{w \mid h(w) \in L\}$$
- Wir werden sehen: aus einem DFA für  $L$  lassen sich leicht konstruieren:
  - ein DFA für  $h^{-1}(L)$  und
  - ein NFA für  $h(L)$
- ➡  $h(L)$  und  $h^{-1}(L)$  regulär

# Abschlusseigenschaften: Homomorphismen (2/3)

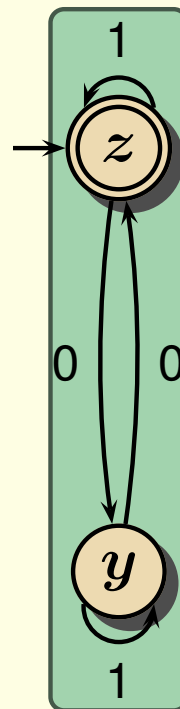
- $h_1$ :  
 $a \mapsto 00$   
 $b \mapsto 1$   
 $c \mapsto \epsilon$   
 $d \mapsto 0100$

- $h_2$ :  
 $0 \rightarrow abc$   
 $1 \rightarrow aa$



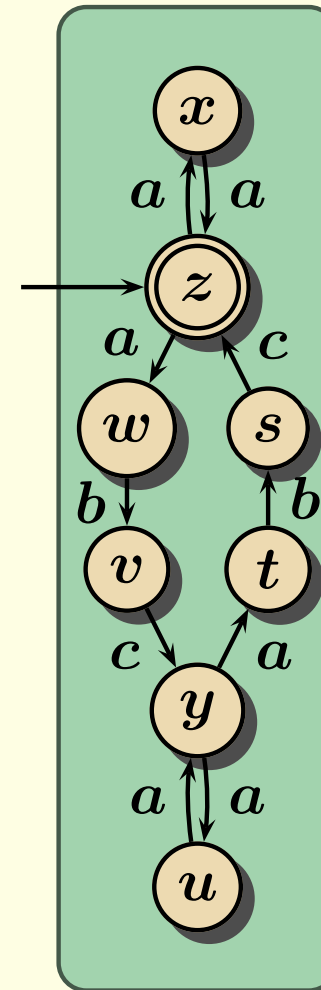
Automat für  $h_1^{-1}(L)$

$h_1^{-1}$   
 $\Leftarrow$



Automat für  $L$

$h_2$   
 $\Rightarrow$



Automat für  $h_2(L)$

# Abschlusseigenschaften: Homomorphismen (3/3)

## Satz 5.4

- Ist  $L$  eine reguläre Sprache über  $\Sigma$  und ist  $\Gamma$  ein Alphabet, so sind die folgenden Sprachen regulär:
  - (a)  $h(L)$ , für jeden Homomorphismus  $h : \Sigma^* \rightarrow \Gamma^*$ ,
  - (b)  $h^{-1}(L)$ , für jeden Homomorphismus  $h : \Gamma^* \rightarrow \Sigma^*$ ,

## Beweisidee

- (b) Sei  $\mathcal{A} = (Q, \Sigma, \delta, s, F)$  DFA für  $L$ 
  - Wir definieren  $\mathcal{A}' \stackrel{\text{def}}{=} (Q, \Gamma, \delta', s, F)$  durch:
$$\delta'(q, \sigma) \stackrel{\text{def}}{=} \delta^*(q, h(\sigma))$$
  - Dann gilt:  $\delta'^*(s, w) = \delta^*(s, h(w))$
  - Also:  $w \in L(\mathcal{A}') \iff h(w) \in L(\mathcal{A})$
- (a) Idee:
  - Ersetze die Transition  $\delta(q, \sigma)$  durch eine Folge von Transitionen für  $h(\sigma)$
  - neue Zustände einfügen

# Synthese endlicher Automaten: Größe

- Die folgende Tabelle gibt eine Übersicht über die Größe der Zielautomaten (Anzahl Zustände) für die betrachteten Operationen
- Dabei spielt es eine Rolle, ob die gegebenen Automaten und der Zielautomat DFAs oder NFAs sind
- $Q_1$  und  $Q_2$  bezeichnen jeweils die Zustandsmengen für Automaten für  $L_1$  und  $L_2$
- $|h|$  und  $|S|$  bezeichnen jeweils die Größe der Repräsentation von  $h$  und  $S$

|                 | DFA $\rightarrow$ DFA                 | DFA $\rightarrow$ NFA             | NFA $\rightarrow$ NFA                 |
|-----------------|---------------------------------------|-----------------------------------|---------------------------------------|
| $L_1 \cap L_2$  | $\mathcal{O}( Q_1  \times  Q_2 )$     | $\mathcal{O}( Q_1  \times  Q_2 )$ | $\mathcal{O}( Q_1  \times  Q_2 )$     |
| $L_1 \cup L_2$  | $\mathcal{O}( Q_1  \times  Q_2 )$     | $\mathcal{O}( Q_1  +  Q_2 )$      | $\mathcal{O}( Q_1  +  Q_2 )$          |
| $L_1 - L_2$     | $\mathcal{O}( Q_1  \times  Q_2 )$     | $\mathcal{O}( Q_1  \times  Q_2 )$ | $ Q_1  \times 2^{\mathcal{O}( Q_2 )}$ |
| $L_1 \circ L_2$ | $ Q_1  \times 2^{\mathcal{O}( Q_2 )}$ | $\mathcal{O}( Q_1  +  Q_2 )$      | $\mathcal{O}( Q_1  +  Q_2 )$          |
| $L_1^*$         | $2^{\mathcal{O}( Q_1 )}$              | $\mathcal{O}( Q_1 )$              | $\mathcal{O}( Q_1 )$                  |
| $h(L_1)$        | $2^{\mathcal{O}( Q_1  +  h )}$        | $\mathcal{O}( Q_1  +  h )$        | $\mathcal{O}( Q_1  +  h )$            |
| $h^{-1}(L_1)$   | $\mathcal{O}( Q_1 )$                  | $\mathcal{O}( Q_1 )$              | $\mathcal{O}( Q_1 )$                  |



# Inhalt

5.1 Abschlusseigenschaften und Synthese von Automaten

▷ **5.2 Grenzen der Regulären Sprachen**

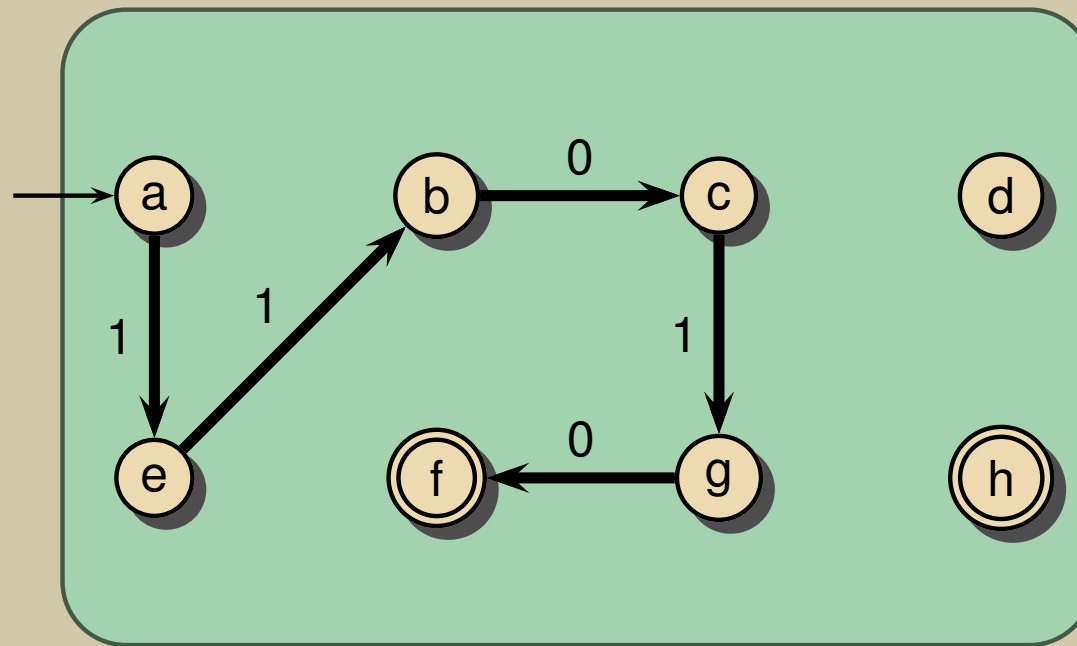
5.3 Weitere Algorithmen für Automaten

# Pumping-Lemma: Einleitung

- Wir haben mit dem Satz von Myhill und Nerode bereits eine Methode kennen gelernt, mit der wir überprüfen können, ob eine Sprache regulär ist
- Damit haben wir gezeigt, dass die Sprache  $L_{ab} = \{a^m b^m \mid m \geq 0\}$  **nicht** regulär ist
- Wir werden jetzt mit dem **Pumping-Lemma** eine weitere Methode kennen lernen, mit der sich nachweisen lässt, dass eine gegebene Sprache nicht regulär ist
- **Vorteile:**
  - Recht einfach und anschaulich
  - Lässt sich verallgemeinern für kontextfreie Sprachen
  - Liefert interessante Einsicht über reguläre Sprachen
- **Nachteile:**
  - Funktioniert nicht immer
  - Lässt sich nicht zum Nachweis von Regularität verwenden

# Pumping-Lemma: Grundidee (1/2)

## Beispiel



110 010 10

$x y z$

110 010 010 10

$x yy z$

110 010 010 010 10

$x yyy z$

$\vdots$

$x y^k z$

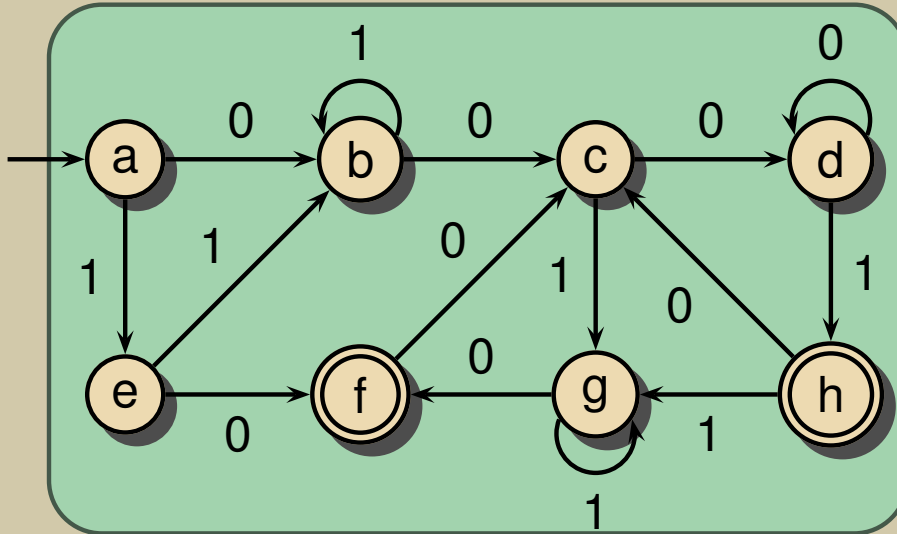
110 10

$x z$

- **Beobachtung:** Ein „Kreis“ in einer akzeptierenden Berechnung lässt sich beliebig oft wiederholen
- Der akzeptierte String wird dabei „aufgepumpt“ (oder „abgepumpt“)

## Pumping-Lemma: Grundidee (2/2)

### Beispiel



- Etwas formaler:

- Wenn  $|w| \geq |Q|$  gilt, muss es einen Zustand geben, den der DFA beim Lesen von  $w$  (mindestens) zweimal besucht
- Wenn der DFA beim Lesen eines Strings  $w \in L(\mathcal{A})$  einen Zustand zweimal besucht, lässt  $w$  sich so in  $xyz$  zerlegen, dass gelten:
  - \*  $y \neq \epsilon$  und
  - \* für alle  $k \geq 0$  ist  $xy^kz \in L(\mathcal{A})$

➡ Wenn  $L$  regulär ist, gibt es ein  $n$ , so dass sich jedes  $w \in L$  mit  $|w| \geq n$  in  $xyz$  zerlegen lässt, so dass gelten:

- $y \neq \epsilon$  und
- für alle  $k \geq 0$  ist  $xy^kz \in L(\mathcal{A})$

✎  $n$  ergibt sich hier als Größe  $|Q|$  der Zustandsmenge eines DFAs für  $L$

# Pumping-Lemma: Aussage und Beweis

## Satz 5.5

- Sei  $L$  regulär
- Dann gibt es ein  $n$ , so dass jeder String  $w \in L$  mit  $|w| \geq n$  auf mindestens eine Weise als  $w = xyz$  geschrieben werden kann, so dass die folgenden Aussagen gelten:
  - (1)  $y \neq \epsilon$
  - (2)  $|xy| \leq n$
  - (3) für alle  $k \geq 0$  ist  $xy^kz \in L$



Die Aussage des Pumping-Lemmas gilt auch in der Form:

$$w \notin L \dots \Rightarrow \dots xy^kz \notin L$$

## Beweisskizze

- $L$  regulär  $\Rightarrow$   
 $L = L(\mathcal{A})$  für einen DFA  $\mathcal{A}$
- Sei  $n$  die Anzahl der Zustände von  $\mathcal{A}$
- Sei  $w \in L$  mit  $|w| \geq n$ 
  - ➔ Beim Lesen der ersten  $n$  Zeichen von  $w$  muss sich ein Zustand wiederholen
  - ➔  $\delta^*(s, x) = \delta^*(s, xy)$  für gewisse  $x, y, z$  mit  $w = xyz$  und (1) und (2)
  - Sei  $q \stackrel{\text{def}}{=} \delta^*(s, x)$
  - ➔  $\delta^*(q, y) = q$
  - ➔  $\delta^*(s, xy^kz) = \delta^*(s, xyz) \in F$ , für alle  $k \geq 0$
  - ➔ (3)

# Pumping-Lemma: Anwendung (1/2)

- Für den Nachweis, dass eine gegebene Sprache nicht regulär ist, ist die folgende äquivalente Formulierung des Pumping-Lemmas besser geeignet

## Korollar 5.6

- Sei  $L$  eine Sprache
  - Angenommen, für jedes  $n > 0$  gibt es einen String  $w \in L$  mit  $|w| \geq n$ , so dass für jede Zerlegung  $w = xyz$  mit
    - (1)  $y \neq \epsilon$  und
    - (2)  $|xy| \leq n$ein  $k \geq 0$  existiert, so dass  $xy^kz \notin L$
  - Dann ist  $L$  nicht regulär
- Da Korollar 5.6 die Kontraposition von Satz 5.5 ist, folgt es direkt aus Satz 5.5

## Beispiel

- Sei wieder  $L_{ab} \stackrel{\text{def}}{=} \{a^m b^m \mid m \geq 0\}$
- Sei  $n$  beliebig
- Wir wählen  $w = a^n b^n \in L_{ab}$   
(wichtig:  $w$  hängt von  $n$  ab!)
- $|w| = 2n \geq n$
- Seien nun  $x, y, z$  beliebige Strings, die  $w = xyz$  und die folgenden Bedingungen erfüllen:
  - (1)  $y \neq \epsilon$
  - (2)  $|xy| \leq n$
- Wegen (2) enthält  $y$  kein  $b$ , wegen (1) enthält es mindestens ein  $a$   
➔  $xz$  hat mehr  $b$  als  $a$
- Wähle  $k = 0$ :  
➔  $xy^0z = xz \notin L_{ab}$
- ➔  $L_{ab}$  ist nicht regulär

## Pumping-Lemma: Anwendung (2/2)

### Beispiel

- Sei wieder  
 $L_{ab} \stackrel{\text{def}}{=} \{a^m b^m \mid m \geq 0\}$
- Sei  $n$  beliebig
- Wir wählen  $w = a^n b^n \in L_{ab}$   
(wichtig:  $w$  hängt von  $n$  ab!)
- $|w| = 2n \geq n$
- Seien nun  $x, y, z$  beliebige Strings, die  $w = xyz$  und die folgenden Bedingungen erfüllen:
  - (1)  $y \neq \epsilon$
  - (2)  $|xy| \leq n$
- Wegen (2) enthält  $y$  kein  $b$ , wegen (1) enthält es mindestens ein  $a$   
➡  $xz$  hat mehr  $b$  als  $a$
- Wähle  $k = 0$ :  
➡  $xy^0z = xz \notin L_{ab}$
- ➡  $L_{ab}$  ist nicht regulär

### • Bemerkungen zur Anwendung des Pumping Lemmas

- $n$  dürfen Sie **nicht** wählen
  - \* Der Beweis muss für beliebiges  $n$  funktionieren
- $w \in L$  dürfen Sie selbst (geschickt) wählen
  - \* Es muss in Abhängigkeit von  $n$  gewählt werden ( $|w| \geq n$ )
    - Dabei ist  $n$  für den Beweis eine Variable
- $x, y, z$  dürfen Sie **nicht** wählen
  - \* Wir wissen aber (und verwenden), dass (1) und (2) gelten
- Zuletzt muss ein  $k$  gefunden werden, für das  $xy^kz \notin L$  gilt
  - \* Sehr oft ist hier eine Fallunterscheidung nötig:
    - nach den Möglichkeiten, wie  $x, y, z$  den String  $w$  unterteilen

# Das Pumping-Lemma als Spiel

- Das Pumping-Lemma ist zwar im Kern recht anschaulich, der Wechsel zwischen Existenz- und Allquantoren kann jedoch durchaus zu Verwirrung führen
- Es kann deshalb hilfreich sein, die Aussage des Pumping-Lemmas in ein 2-Personen-Spiel zu fassen

- **Spiel für Sprache  $L$ :**

- Person 1 wählt  $n$
- Person 2 wählt ein  $w \in L$  mit  $|w| \geq n$
- Person 1 wählt  $x, y, z$  mit
$$w = xyz, y \neq \epsilon, |xy| \leq n$$
- Person 2 wählt  $k$

- Falls  $xy^kz \notin L$ , hat Person 2 gewonnen, andernfalls Person 1

- Es gilt: **falls Person 2 eine Gewinnstrategie hat, ist  $L$  nicht regulär**
- Wenn Sie nachweisen wollen, dass  $L$  nicht regulär ist, **sind Sie Person 2**



# Grenzen des Pumping-Lemmas

- Sei  $L$  die Sprache  $\{a^m b^n c^n \mid m, n \geq 1\} \cup \{b^m c^n \mid m, n \geq 0\}$
- Klar:  $L$  ist nicht regulär
  - Das lässt sich durch eine leichte Abwandlung des Beweises aus Kapitel 4 für  $L_{ab}$  zeigen
- Aber:  $L$  erfüllt die Aussage des Pumping-Lemmas:
  - Jeder String  $w$ , lässt sich als  $xyz$  zerlegen, mit  $x = \epsilon$  und  $|y| = 1$
  - ➔ dann lässt sich  $y$  beliebig wiederholen:
    - \* falls  $y = a$ : klar, dann ist das Wort in der ersten Menge und es dürfen beliebig viele  $a$ 's kommen
    - \* falls  $y \neq a$ : klar, dann ist das Wort in der zweiten Menge und das Zeichen darf beliebig wiederholt werden

- Es gilt aber die folgende Verallgemeinerung des Pumping-Lemmas

## Satz 5.7 [Jaffe, 78]

- Eine Sprache  $L$  ist **genau dann** regulär **wenn** es ein  $n$  gibt, so dass jeder String  $w \in L$  mit  $|w| \geq n$  auf mindestens eine Weise als  $w = xyz$  geschrieben werden kann, so dass die folgenden Aussagen gelten:
  - (1)  $y \neq \epsilon$
  - (2)  $|xy| \leq n$
  - (3') für alle  $k \geq 0$  gilt:  $xy^k z \sim_L xyz$

# Reguläre Sprachen: Grenzen

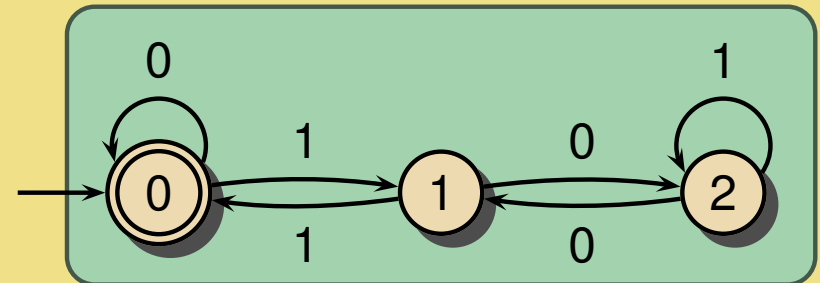
- Woran lässt sich erkennen, ob eine Sprache regulär ist?
- Intuitiv: wenn es genügt, sich beim Lesen eines Eingabewortes nur konstant viel Information zu merken, **unabhängig von der Eingabelänge**
- Beispiel:  $L_{ab} = \{a^m b^m \mid m \geq 0\}$  ist nicht regulär, da nach Lesen von  $a^i$  „das  $i$  gemerkt sein muss“
- Insbesondere darf der Wertebereich, in dem gezählt wird, nicht mit der Eingabelänge größer werden
- Aber: es gibt auch reguläre Sprachen, die mit Zahlen zu tun haben, zum Beispiel:  $L_{\text{dreier}} \stackrel{\text{def}}{=} \{w \mid w \text{ ist die Binärdarstellung einer Zahl, die durch drei teilbar ist}\}$

# Reguläre Sprachen: Zählen modulo ...

- Ziel: Automat für  $L_{\text{drei}} = \{w \mid w \text{ ist die Binärdarstellung einer Zahl, die durch drei teilbar ist}\}$
- Ansatz: was passiert, wenn an eine Binärzahl eine 0 oder 1 angehängt wird?
- Notation:
  - Für  $w \in \{0, 1\}^*$  sei  $B(w)$  die Zahl, die von  $w$  repräsentiert wird
  - Also:  $B(1100) = 12$

- Es gelten:
  - $B(u0) = 2B(u)$
  - $B(u1) = 2B(u) + 1$
- Also: wenn  $B(u) \equiv_3 0$ , dann  $B(u0) \equiv_3 0$  und  $B(u1) \equiv_3 1$

→ Grundidee des Automaten: der Zustand (0,1, oder 2) gibt den Rest der bisher gelesenen Binärzahl bei Division durch 3 an



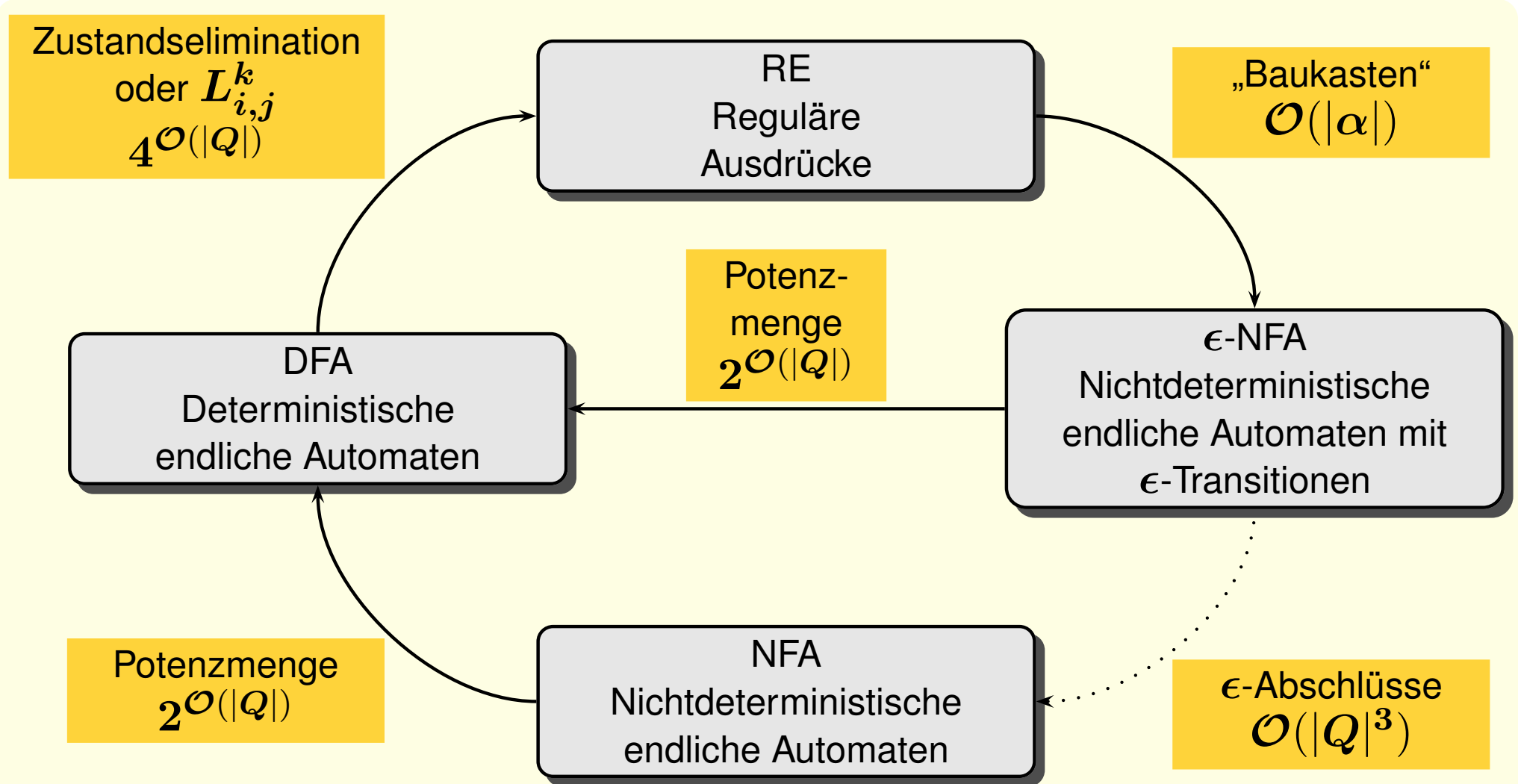
# Inhalt

5.1 Abschlusseigenschaften und Synthese von Automaten

5.2 Grenzen der Regulären Sprachen

▷ **5.3 Weitere Algorithmen für Automaten**

# Umwandlungsalgorithmen (Wdh.)



- Die Laufzeiten der Umwandlungsalgorithmen entsprechen im Wesentlichen den Größen der Zielobjekte

# Algorithmen: Leerheitstest

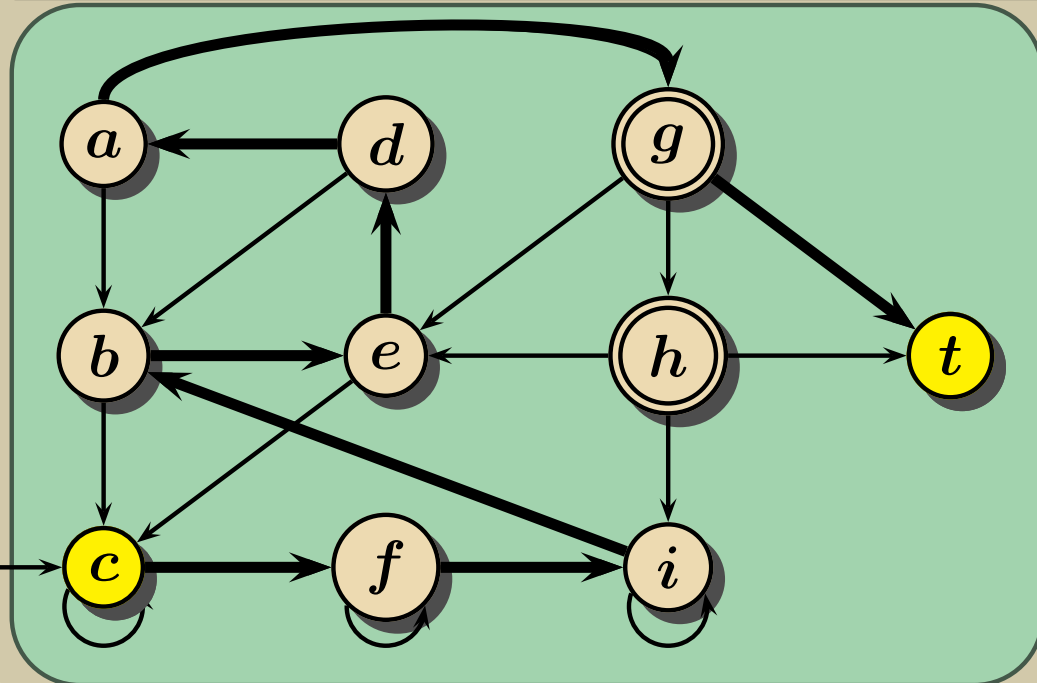
- Für Anwendungen (nicht nur) im Bereich des Model Checking ist das folgende Problem wichtig:

Definition: Leerheits-Problem für DFAs

**Gegeben:** DFA  $\mathcal{A}$

**Frage:** Ist  $L(\mathcal{A}) \neq \emptyset$ ?

Beispiel



Algorithmus:

- Vergiss die Kantenmarkierungen
- Füge einen Zielknoten  $t$  ein und Kanten von allen akzeptierenden Knoten zu  $t$
- Teste, ob es einen Weg von  $s$  nach  $t$  gibt
- Falls ja: Ausgabe „ $L(\mathcal{A}) \neq \emptyset$ “

- Das Leerheitsproblem für DFAs lässt sich also auf das Erreichbarkeitsproblem in gerichteten Graphen zurückführen
  - Aufwand:**  $\mathcal{O}(|\delta|)$
- Der Algorithmus funktioniert auch für NFAs
- Der Leerheitstest für reguläre Ausdrücke  $\alpha$  ist (fast) trivial:
  - falls kein  $\emptyset$  vorkommt, ist  $L(\alpha) \neq \emptyset$
  - Ansonsten lässt sich  $\alpha$  gemäß der Regeln aus Kapitel 2 vereinfachen
  - $L(\alpha) \neq \emptyset \iff$   
am Schluss bleibt **nicht**  $\emptyset$  übrig

# Algorithmen: Wortproblem

## Definition: Wortproblem für Reguläre Sprachen

**Gegeben:** Wort  $w \in \Sigma^*$ , reguläre Sprache  $L$ , repräsentiert durch DFA  $\mathcal{A}$ , NFA  $\mathcal{A}'$  oder RE  $\alpha$

**Frage:** Ist  $w \in L$ ?

- Der Algorithmus für das Wortproblem und damit der Aufwand hängen von der Repräsentation der Sprache ab:

– **DFA:** Simuliere den Automaten  
Aufwand:  $\mathcal{O}(|w| + |\delta|)$

– **NFA:** Simuliere den Potenzmengenautomaten, ohne ihn explizit zu konstruieren  
\* Speichere dabei immer nur die aktuell erreichte Zustandsmenge  
Aufwand:  $\mathcal{O}(|w| \times |\delta|)$

– **RE:** Wandle den RE in einen  $\epsilon$ -NFA um und simuliere dann den Potenzmengenautomaten  
Aufwand:  $\mathcal{O}(|w| \times |\alpha|)$

## Erläuterungen:

- Für die Simulation wird jeweils zuerst die Transitionsfunktion  $\delta$  in ein Array geschrieben
- Aufwand  $\mathcal{O}(|\delta|)$
- Die einzelnen Übergänge können dann durch Nachschauen in der Tabelle ausgeführt werden
- Bei der Simulation von NFAs sind die Tabelleneinträge jeweils Mengen von Zuständen

# Algorithmen: Äquivalenztest für DFAs (1/3)

Definition: Äquivalenzproblem für DFAs

**Gegeben:** DFAs  $\mathcal{A}_1$  und  $\mathcal{A}_2$

**Frage:** Ist  $L(\mathcal{A}_1) = L(\mathcal{A}_2)$ ?

- Wir betrachten zwei Lösungsmethoden:

- (1) Mit Minimalautomaten
- (2) Mit dem Produktautomaten

## (1) Mit Minimalautomaten:

- Konstruiere die Minimal-Automaten:  $\mathcal{A}'_1$  und  $\mathcal{A}'_2$  zu  $\mathcal{A}_1$  und  $\mathcal{A}_2$
- Teste, ob  $\mathcal{A}'_1$  und  $\mathcal{A}'_2$  isomorph sind:
  - \* Konstruiere dazu schrittweise eine Bijektion  $\pi$  von (den Zuständen von)  $\mathcal{A}'_1$  auf  $\mathcal{A}'_2$
  - \* Initialisierung:  $\pi$  bildet Startzustand auf Startzustand ab
  - \* Dann: Setze  $\pi$  gemäß der Transitionen von  $\mathcal{A}'_1$  und  $\mathcal{A}'_2$  fort
- Aufwand:  $\mathcal{O}(|\Sigma|(|Q_1|^2 + |Q_2|^2))$

## (2) Mit dem Produktautomaten:

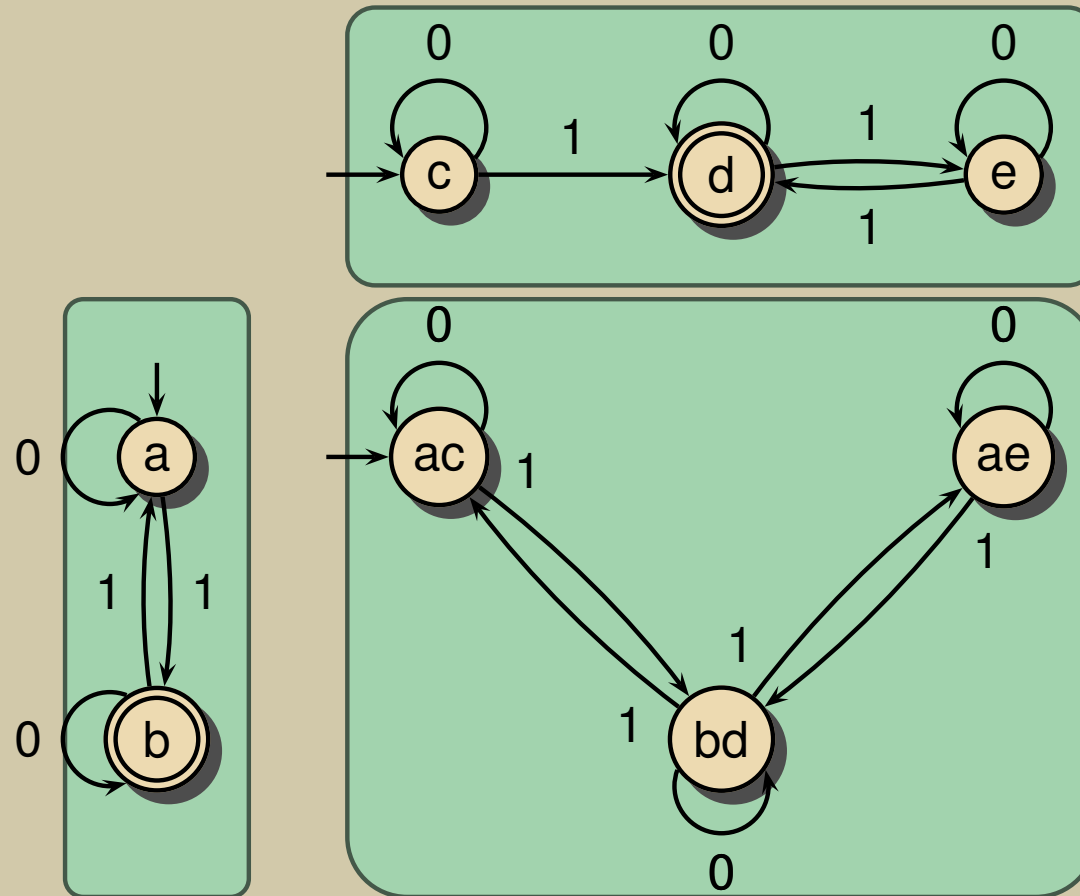
- Konstruiere den Produktautomaten  $\mathcal{A} = \mathcal{A}_1 \times \mathcal{A}_2$  mit der akzeptierende Menge  $F \stackrel{\text{def}}{=} \{(p_1, p_2) \mid (p_1 \in F_1, p_2 \notin F_2) \text{ oder } (p_1 \notin F_1, p_2 \in F_2)\}$
- ➔  $\delta_{\mathcal{A}}^*((s_1, s_2), w) \in F$  genau dann, wenn  $w$  genau von einem der beiden Automaten akzeptiert wird
- Also:  $L(\mathcal{A}_1) = L(\mathcal{A}_2)$  genau dann, wenn  $L(\mathcal{A}) = \emptyset$
- Die Frage „Ist  $L(\mathcal{A}_1) = L(\mathcal{A}_2)$ ?“ kann dann also durch den Leerheitstest für  $\mathcal{A}$  entschieden werden
- ➔ Aufwand:  
 $\mathcal{O}(|Q_1| \times |Q_2| \times |\Sigma|)$



# Algorithmen: Äquivalenztest für DFAs (2/3)

## Beispiel

- Sind diese beiden DFAs äquivalent?

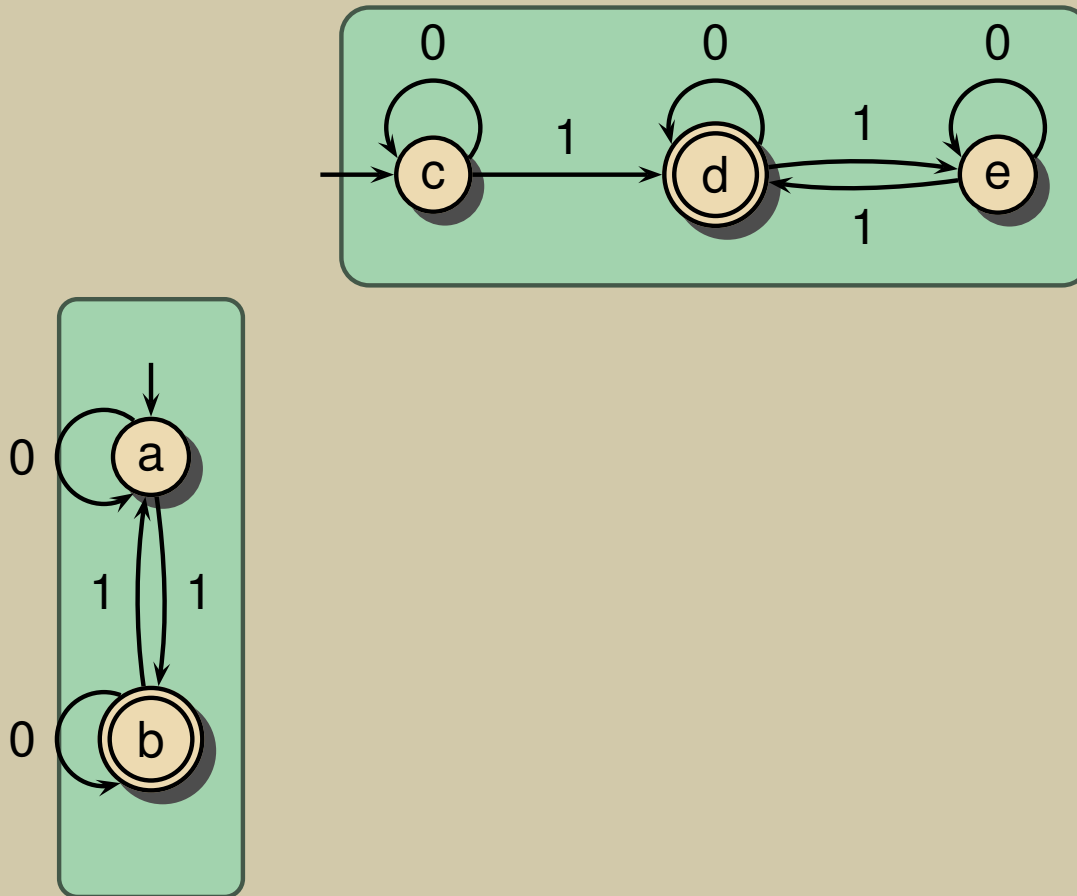


- Berechnung des Produktautomaten
- Nicht erreichbare Zustände entfernen
- Die Sprache des Automaten ist leer
- ➡ Die Automaten sind äquivalent

# Algorithmen: Äquivalenztest für DFAs (3/3)

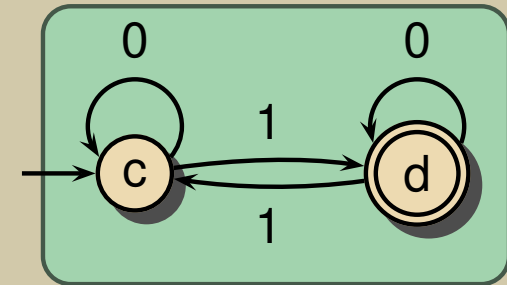
## Beispiel

- Sind diese beiden DFAs äquivalent?



## Beispiel (Forts.)

- Minimierung des oberen Automaten liefert:



- Beide Automaten sind nun isomorph gemäß

- $a \mapsto c$
- $b \mapsto d$

- Noch eine weitere Methode:
  - Führe den Markierungsalgorithmus auf „ $\mathcal{A}_1 \cup \mathcal{A}_2$ “ aus und überprüfe, ob  $(s_1, s_2)$  markiert wird

# Algorithmen: Äquivalenztest für NFAs und REs

- Äquivalenztests für NFAs und REs sind zwar auch automatisierbar, aber die Komplexität ist erheblich größer
- Genauer: Zu testen, ob zwei reguläre Ausdrücke (oder zwei NFAs) äquivalent sind ist vollständig für die Komplexitätsklasse **PSPACE**
  - Das gilt sogar, wenn einer der REs gleich  $\Sigma^*$  ist
  - Intuitiver Grund: Die REs müssen zuerst in DFAs umgewandelt werden
- Was „vollständig für **PSPACE**“ bedeutet, werden wir im letzten Teil der Vorlesung sehen
- Hier lässt sich schon sagen: das Problem ist (wohl) noch schwieriger als **NP**-vollständige Probleme wie das Traveling Salesman Problem

# Algorithmen: Endlichkeitstest

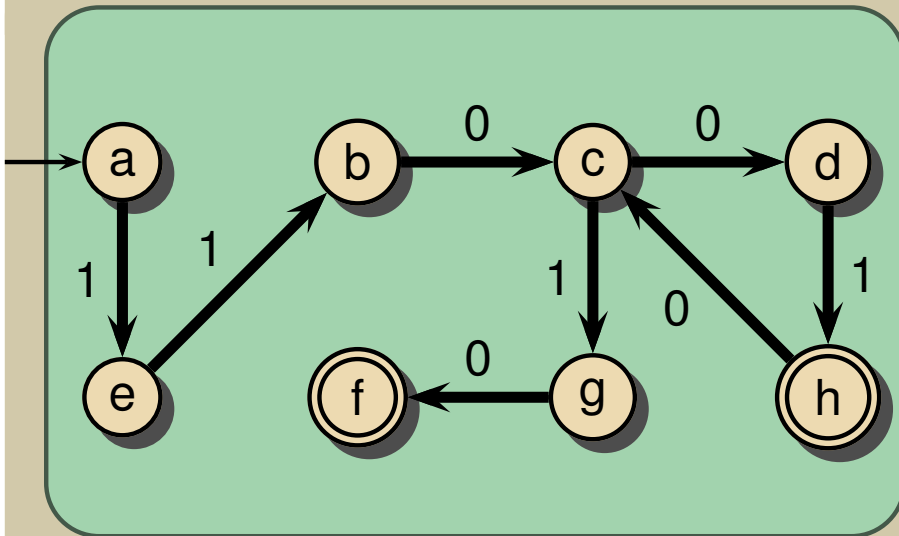
Definition: Endlichkeitsproblem für DFAs

**Gegeben:** DFA  $\mathcal{A}$

**Frage:** Ist  $L(\mathcal{A})$  endlich?

- Hier hilft uns die Grundidee des Pumping-Lemmas weiter:

Beispiel



Satz 5.8

- Die Sprache eines DFA  $\mathcal{A}$  ist genau dann unendlich, wenn  $\mathcal{A}$  einen Zustand  $q$  mit den folgenden Eigenschaften hat:
  - (a)  $q$  ist von  $s$  aus erreichbar  
d.h.:  $\exists x \in \Sigma^* : \delta^*(s, x) = q$
  - (b)  $q$  liegt auf einem Kreis  
d.h.:  $\exists y \in \Sigma^* : y \neq \epsilon$  und  $\delta^*(q, y) = q$
  - (c) Von  $q$  aus ist ein akzeptierender Zustand erreichbar  
d.h.  $\exists z \in \Sigma^* : \delta^*(q, z) \in F$

Beweisidee

„ $\Leftarrow$ “ Dann werden die unendlich vielen Strings  $xy^0z, xy^1z, xy^2z, \dots$  von dem DFA akzeptiert

„ $\Rightarrow$ “ Die Existenz eines solchen Zustandes  $q$  folgt wie im Beweis des Pumping Lemmas

- Aufwand:  $\mathcal{O}(|\delta|)$  (Doppelte DFS-Suche)
- Funktioniert auch für NFAs

# Zusammenfassung

- Um Automaten und reguläre Ausdrücke anwenden zu können (zum Beispiel im Model Checking), benötigen wir Algorithmen für die Synthese und zum Testen von Eigenschaften regulärer Sprachen
- Synthese:
  - Die regulären Sprachen sind **unter vielen Operationen abgeschlossen**
  - In vielen Fällen lassen sich die entsprechenden Zielautomaten effizient berechnen
  - Für Boolesche Operationen spielen **Produktautomaten** eine wichtige Rolle
- Test von Eigenschaften:
  - **Leerheit** und **Endlichkeit** der Sprache eines NFA können effizient getestet werden - dabei wird im Wesentlichen ein Erreichbarkeitsproblem für gerichtete Graphen gelöst
  - **Äquivalenz zweier DFAs** kann ebenfalls effizient getestet werden
  - **Äquivalenz von NFAs und REs** ist im allgemeinen (wohl) erheblich schwieriger zu testen
- Das Pumping-Lemma liefert ein weiteres Verfahren zum Nachweis, dass eine Sprache nicht regulär ist

# Grundbegriffe der Theoretischen Informatik

Sommersemester 2016 - Thomas Schwentick

Teil A: Reguläre Sprachen

6: Anwendungen und Erweiterungen

Version von: 10. Mai 2016 (12:10)

## 6.1 Anwendungen regulärer Sprachen

### ▷ 6.1.1 Lexikalische Analyse

6.1.2 Zeichenkettensuche

6.1.3 Model Checking

6.1.4 UML

6.1.5 Automatic Planning

6.1.6 XML

## 6.2 Erweiterungen der endlichen Automaten

## Anwendung: Lexikalische Analyse (1/3)

- Phasen eines Compilers (schematisch):

- Lexikalische Analyse
- Syntaktische Analyse
- Semantische Analyse
- Zwischencode-Erzeugung
- Zwischencode-Optimierung
- Code-Erzeugung

- Die **lexikalische Analyse** wird von einem **Scanner** durchgeführt

- Dafür ist die Ausdruckskraft regulärer Sprachen ausreichend

- Bei der **Syntaxanalyse** wird die Struktur eines Programmes überprüft und in Form eines Syntax-Baumes repräsentiert
- Sie wird vom **Parser** durchgeführt
- Dazu werden **kontextfreie Sprachen** verwendet, die wir in Teil B kennenlernen werden




## Anwendung: Lexikalische Analyse (2/3)

- **Was macht ein Scanner?**
- Der Scanner fasst zusammengehörige Zeichen des Programmtextes zu **Token** zusammen und ordnet sie **Token-Klassen** zu

### Beispiel

Zaehler := Zaehler + 3 \* Runde;

| Token   | Klasse                   |
|---------|--------------------------|
| Zaehler | Identifikator            |
| :=      | Zuweisungs-Operator      |
| Zaehler | Identifikator            |
| +       | Additions-Operator       |
| 3       | Zahl                     |
| *       | Multiplikations-Operator |
| Runde   | Identifikator            |
| ;       | Semikolon                |

- Tokenklassen lassen sich durch reguläre Ausdrücke beschreiben (hier in UNIX-Schreibweise):
  - Identifikator:  
 $[A - Za - z]$   
 $[A - Za - z0 - 9]^*$
  - Zahl:  $[0 - 9]^+$
  - Multiplikations-Operator:  $*$
  - Zuweisungs-Operator:  $:=$
  - usw.
- Aus diesen regulären Ausdrücken lässt sich ein **endlicher Automat mit Ausgabe** konstruieren, der die Zerlegung und Zuordnung vornimmt
  -  Automaten mit Ausgabe betrachten wir in diesem Kapitel auch noch

## Anwendung: Lexikalische Analyse (3/3)

- Scanner müssen nicht eigenhändig von Grund auf programmiert werden
- Es gibt **Scanner-Generatoren**, denen nur die regulären Ausdrücke und die entsprechenden Aktionen mitgeteilt werden müssen

- Zum Beispiel: `lex`:

|   |  |
|---|--|
| <code>if</code>                             | <code>{return (if)}</code>   |
| <code>[A - Za - z][A - Za - z0 - 9]*</code> | <code>{Code, um den zugehörigen Identifikator in der <b>Symboltabelle</b> zu finden;<br/>return (ID)}</code> |
| <code>[0 - 9]*</code>                       | <code>{Code, um Wert der Zahl zu berechnen;<br/>return (val)}</code>   |
| <code>:=</code>                             | <code>{return (assignment)}</code>   |
| <code>...</code>                            | <code>...</code>   |

## 6.1 Anwendungen regulärer Sprachen

6.1.1 Lexikalische Analyse

▷ **6.1.2 Zeichenkettensuche**

6.1.3 Model Checking

6.1.4 UML

6.1.5 Automatic Planning

6.1.6 XML

6.2 Erweiterungen der endlichen Automaten

# Anwendung: Zeichenkettensuche

- Endliche Automaten können auch für die Suche nach Zeichenketten in Texten verwendet werden
- Wir betrachten dazu das Beispiel der Suche nach mehreren gegebenen Wörtern in einem (typischerweise großen) Text

## Definition: MULTISEARCH

**Gegeben:** Menge  $M = \{w_1, \dots, w_n\}$  von nicht leeren Zeichenketten, String  $v$

**Frage:** Kommt einer der Strings  $w_1, \dots, w_n$  in  $v$  vor?

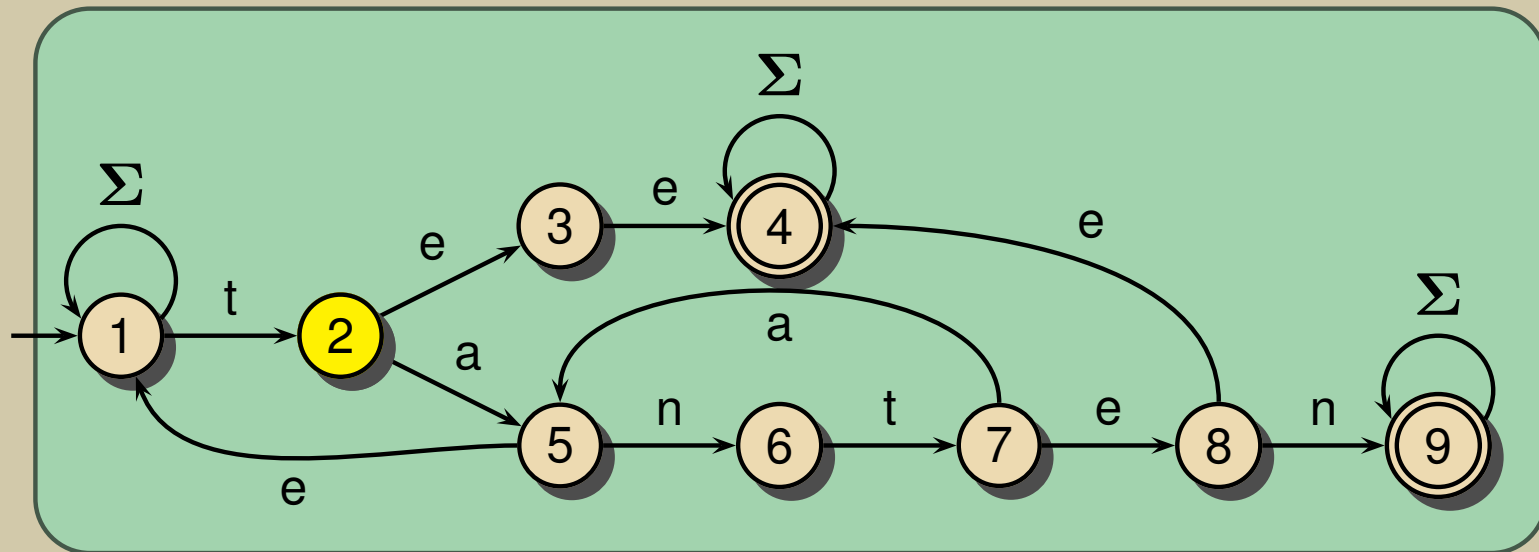
- $v$  repräsentiert also den Text, in dem gesucht wird
- In der Praxis ist  $v$  **viel** länger als die Strings in  $M$
- Wir gehen im Folgenden davon aus, dass kein String  $w_i$  Teilstring eines anderen Strings  $w_j$  ist
  - ✎ Sonst kann  $w_j$  einfach weggelassen werden

## Zeichenkettensuche: Beispiel (1/2)

- Zur Suchmenge  $M$  lässt sich einfach ein NFA konstruieren, der einen Text genau dann akzeptiert, wenn er ein Wort aus  $M$  enthält

### Beispiel

- $M = \{\text{tee, tanten}\}$

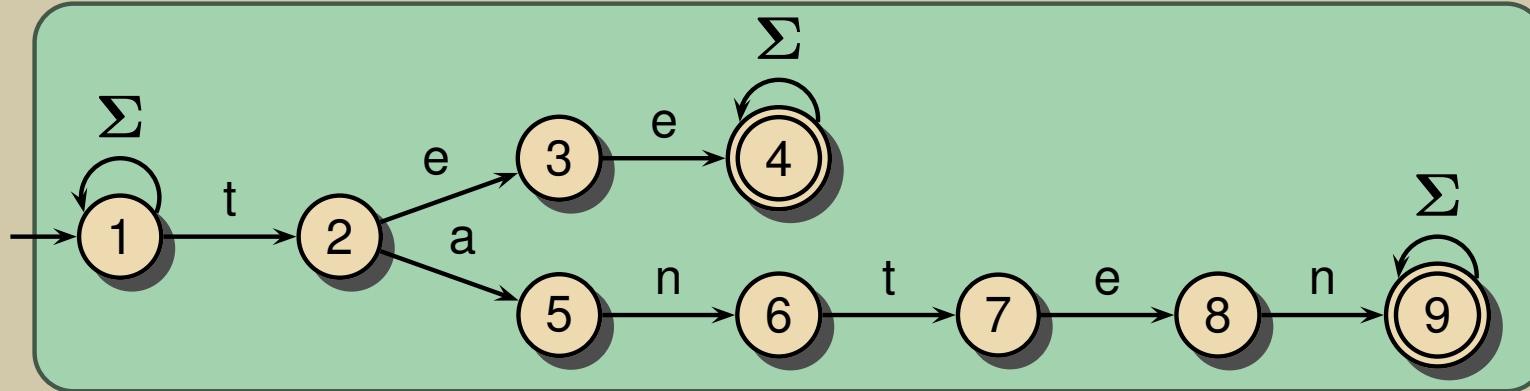


- Wie lässt sich dieser Automat deterministisch machen?
  - Klar: mit der Potenzmengenkonstruktion, aber geht es vielleicht direkter?
- Dazu betrachten wir die Eingabe: **taet|antanteeta**

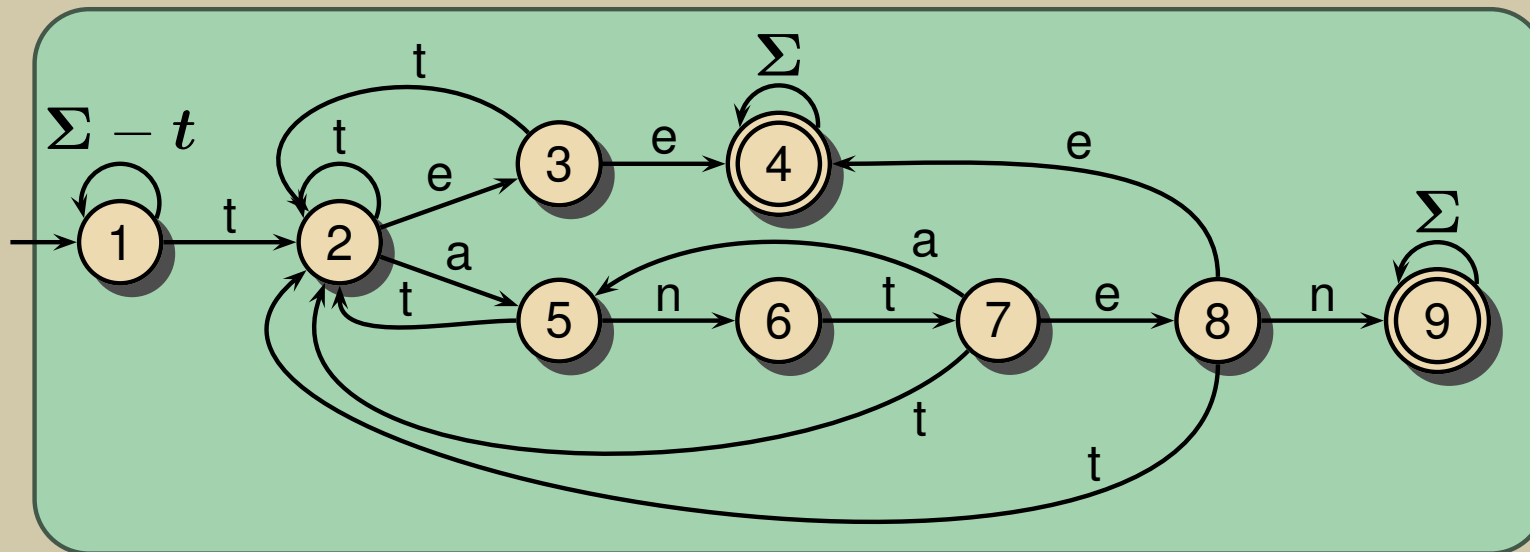
## Zeichenkettensuche: Beispiel (2/2)

- Insgesamt erhalten wir zu dem NFA den unten angegebenen DFA  
(alle nicht gezeigten Übergänge des DFA führen in Zustand 1)

Beispiel: NFA für  $M = \{\text{tee, tanten}\}$



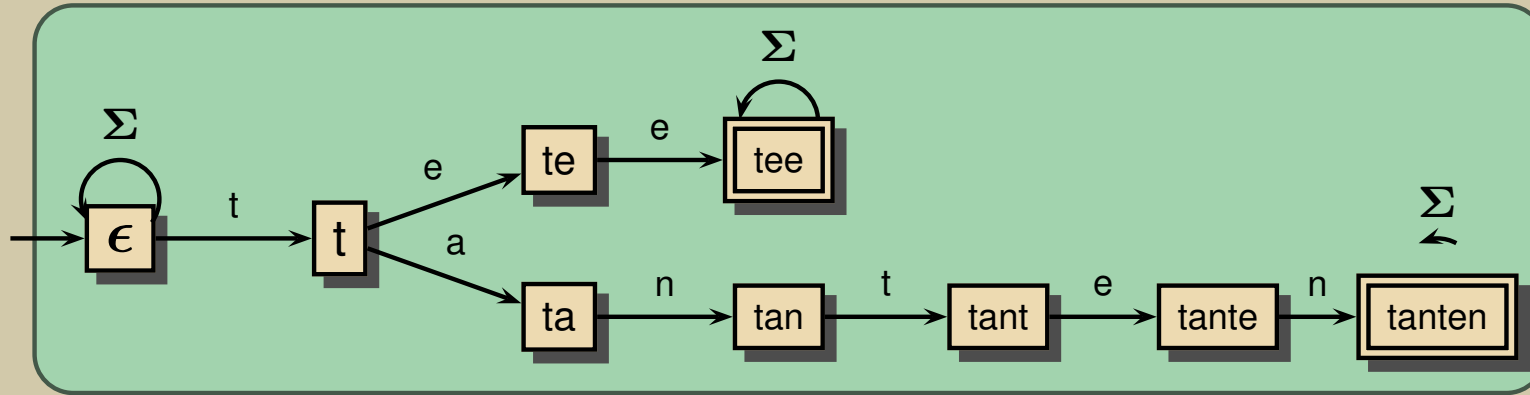
Beispiel: DFA für  $M = \{\text{tee, tanten}\}$



# NFA zur Zeichenkettensuche: allgemein

- Wie können wir allgemein zu einer gegebenen Menge  $M = \{w_1, \dots, w_n\}$  von Zeichenketten den NFA und den DFA formal definieren?

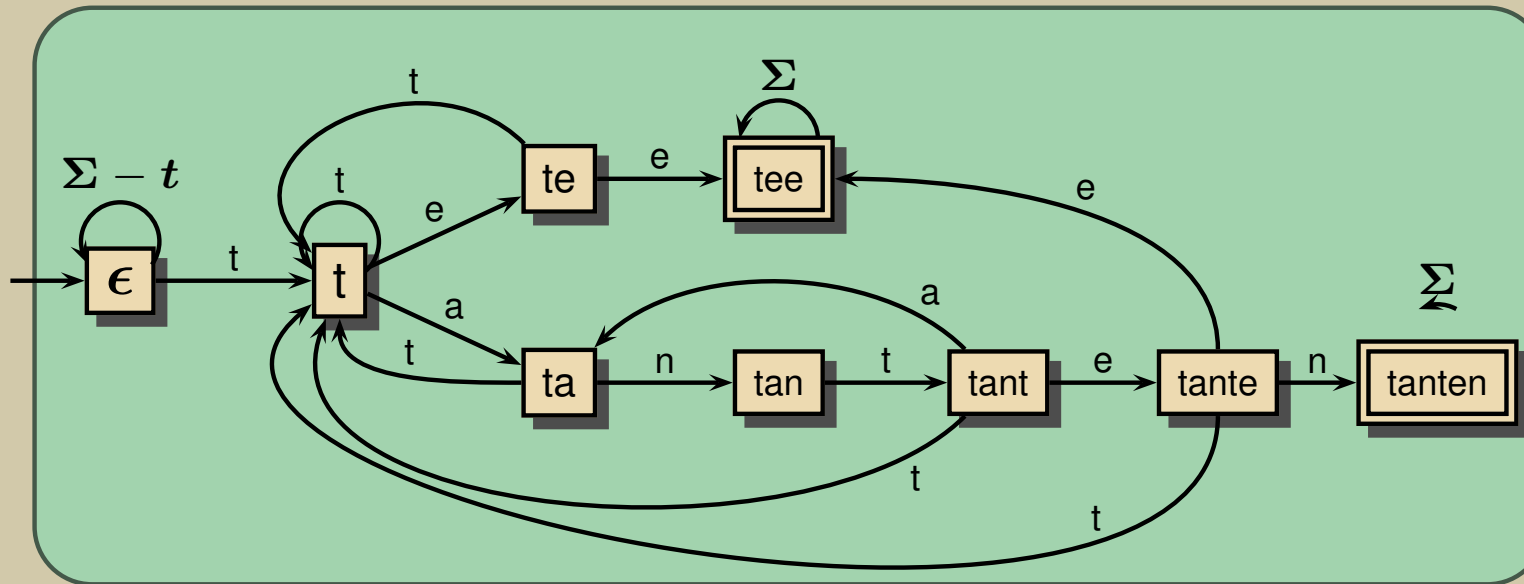
Beispiel: NFA für  $M = \{\text{tee}, \text{tanten}\}$



- NFA  $\mathcal{A}_M \stackrel{\text{def}}{=} (Q, \Sigma, \delta, s, F)$  mit
  - $Q \stackrel{\text{def}}{=} \{u \in \Sigma^* \mid u \text{ ist Präfix von } w_i, \text{ für ein } i \leq n\}$
  - $s \stackrel{\text{def}}{=} \epsilon, F \stackrel{\text{def}}{=} M$
  - $\delta$  enthält die folgenden Transitionen für  $u \in Q$  und  $\sigma \in \Sigma$ :
    - \*  $(\epsilon, \sigma, \epsilon)$
    - \*  $(u, \sigma, u\sigma)$ , falls  $u\sigma \in Q$
    - \*  $(u, \sigma, u)$ , falls  $u \in M$

# DFA zur Zeichenkettensuche: allgemein

Beispiel: DFA für  $M = \{\text{tee}, \text{tanten}\}$



- Der DFA  $\mathcal{A}'_M \stackrel{\text{def}}{=} (Q, \Sigma, \delta', s, F)$  hat die selbe Zustandsmenge, den selben Startzustand, und die selbe akzeptierende Menge wie der NFA  $\mathcal{A}_M = (Q, \Sigma, \delta, s, F)$
- Beispielsweise gilt  $\delta'(\text{tant}, a) = \text{ta}$ , weil „ta“ das längste gelesene Suffix von „tanta“ ist, das auch Präfix eines Suchstrings ist
- Allgemein definieren wir:

$$\delta'(u, \sigma) \stackrel{\text{def}}{=} \begin{cases} \text{maximales Suffix } y \text{ von } u\sigma \text{ mit } y \in Q & \text{falls } u \notin F \\ u & \text{falls } u \in F \end{cases}$$

- Auf den Nachweis der Korrektheit verzichten wir: Induktion



## Anwendung: Zeichenkettensuche (Forts.)

- Bei der Zeichenkettensuche bringt also die Umwandlung des NFA in einen DFA keinerlei Größenzuwachs mit sich
  - Wie das Beispiel des „Borussia-Automaten“ schon gezeigt hat, kann es durchaus sein, dass sich gegenüber der hier angegebenen DFA-Konstruktion durch Minimierung sogar noch Zustände einsparen lassen
- Der Automat  $\mathcal{A}'_M$  lässt sich leicht in einen **Algorithmus** umwandeln: wenn der Automat konstruiert ist, lässt sich dann in **linearer Zeit** testen, ob ein Text einen String aus  $M$  enthält
  - Durch eine einfache Modifikation lassen sich auch alle passenden Textstellen ausgeben
- In vielen Fällen ist es praktisch, zur Spezifikation des Suchmusters die Flexibilität regulärer Ausdrücke zur Verfügung zu haben
- Der reguläre Ausdruck wird dann in einen endlichen Automaten (mit Ausgabe) umgewandelt, der die Stellen des Textes, an denen das Suchmusters passt, ausgibt
- Beispiel:
  - **grep** (Global search for a **R**egular **E**xpression and **P**rint out matched lines)
  - Varianten: egrep, fgrep
  - „**egrep** ***"m(e|a)(i|y)e?r"*** **adressen.txt**“ gibt alle Zeilen der Datei adressen.txt aus, die einen **Meier-artigen** String enthalten
- Um Zeichenketten in riesigen Datenmengen zu finden (Suchmaschinen), sind natürlich erheblich ausgeklügeltere Datenstrukturen und Algorithmen nötig

## 6.1 Anwendungen regulärer Sprachen

6.1.1 Lexikalische Analyse

6.1.2 Zeichenkettensuche

▷ **6.1.3 Model Checking**

6.1.4 UML

6.1.5 Automatic Planning

6.1.6 XML

## 6.2 Erweiterungen der endlichen Automaten

# Anwendung: Automaten-basiertes Model Checking (1/3)

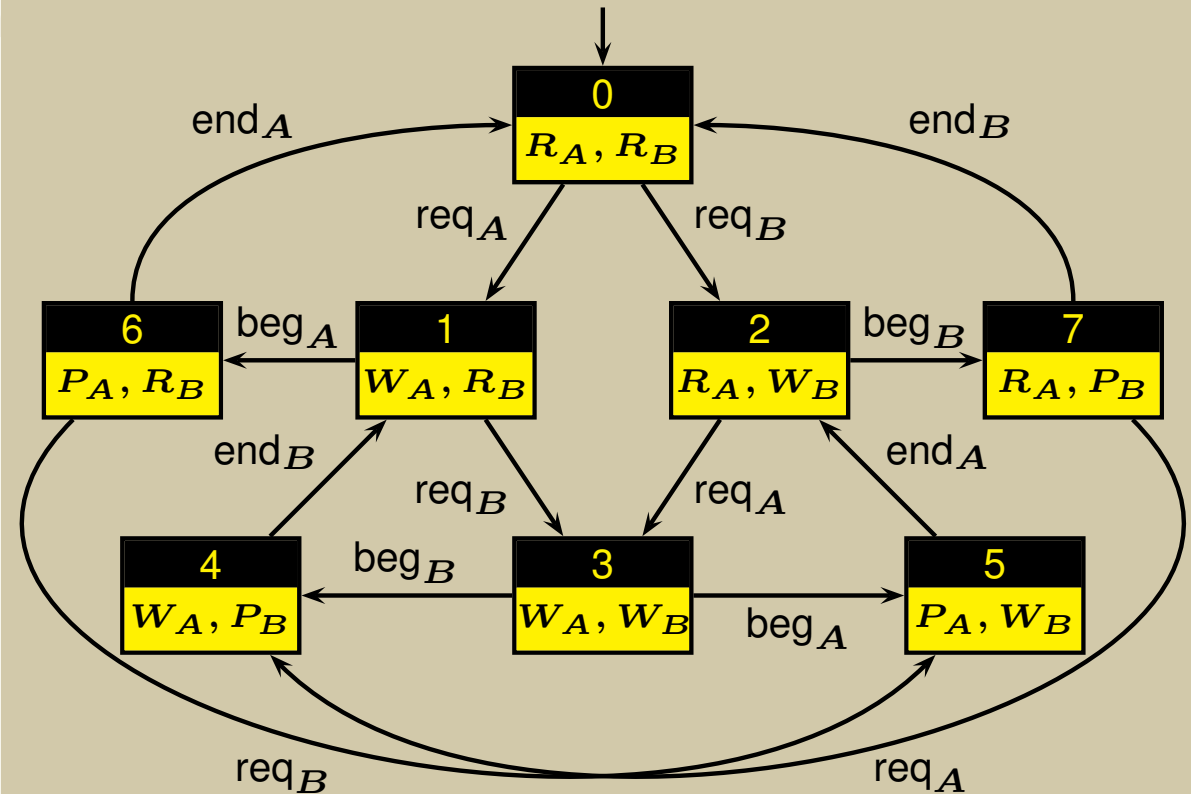
## Beispiel: Drucker-Server

- 2 Benutzer (A,B), 1 Drucker
- Jeder Benutzer kann
  - auf Drucker warten (W)
  - drucken (P)
  - nichts tun (R)
- Die Aktionen des Druckers:
  - $\text{req}_X$ : Benutzer  $X$  startet Druckauftrag
  - $\text{beg}_X$ : Beginn des Drucks für Benutzer  $X$
  - $\text{end}_X$ : Ende des Drucks für Benutzer  $X$

- Systemzustände entsprechen Zuständen eines NFA
- Transitionen entsprechen Übergängen eines NFA

→ **endliches Transitionssystem**

## Beispiel

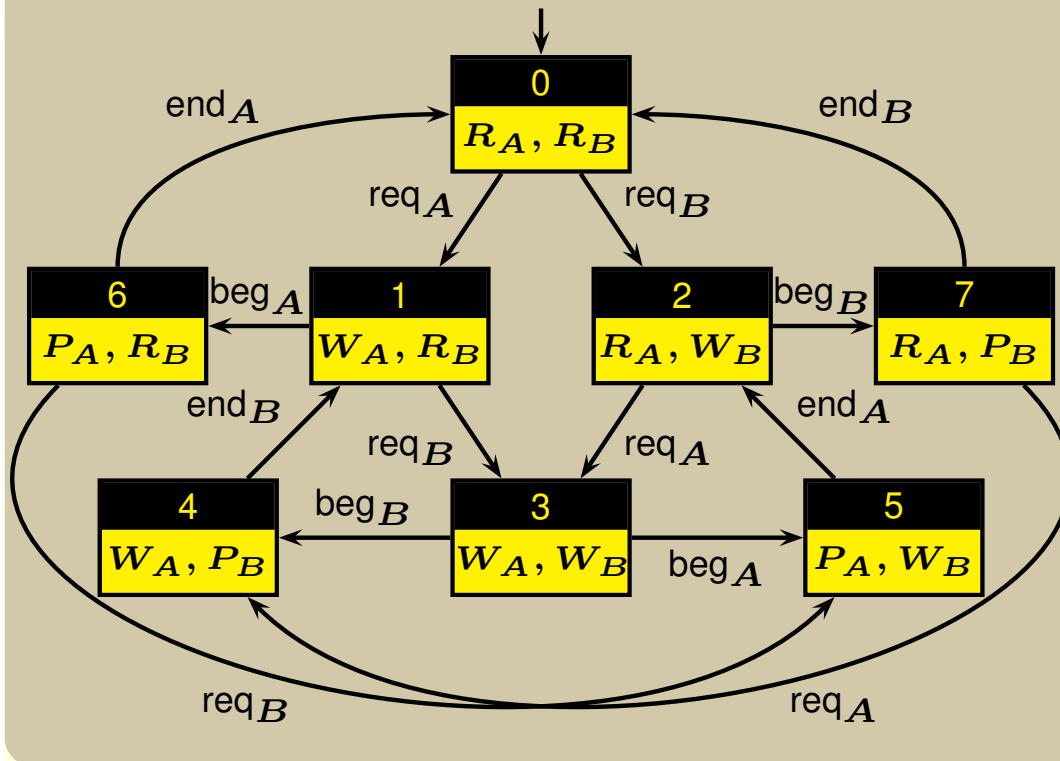


- Wie in Kripkestrukturen haben die Zustände hier jeweils noch eine Menge von Propositionen (aus:  $\{W_A, P_A, R_A, W_B, P_B, R_B\}$ )

✎ Wir werden uns hier aber auf Eigenschaften beschränken, die sich auf Aktionen beziehen

## Anwendung: Automaten-basiertes Model Checking (2/3)

### Beispiel

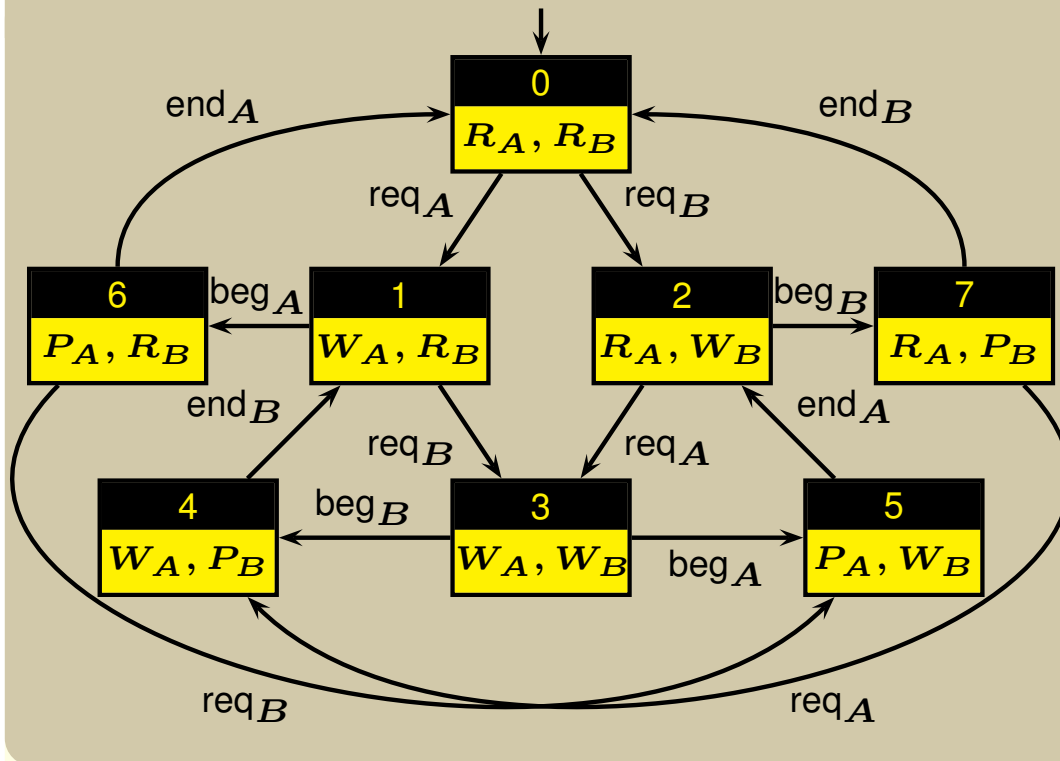


- Wir betrachten an Hand dieses sehr (!) einfachen Beispiels die automatische Verifikation von System-Eigenschaften
- Wir wollen testen, ob die Menge der möglichen Läufe des Transitionssystems eine bestimmte Eigenschaft hat

- Beispiel-Eigenschaft:
  - „Wer zuerst kommt, druckt zuerst“
  - Präziser (und eingeschränkter) sei  $P$  die Eigenschaft:  
 $P$ : Falls das erste  $req_A$  vor dem ersten  $req_B$  vorkommt, so soll das erste  $beg_A$  vor dem ersten  $beg_B$  vorkommen

# Anwendung: Automaten-basiertes Model Checking (3/3)

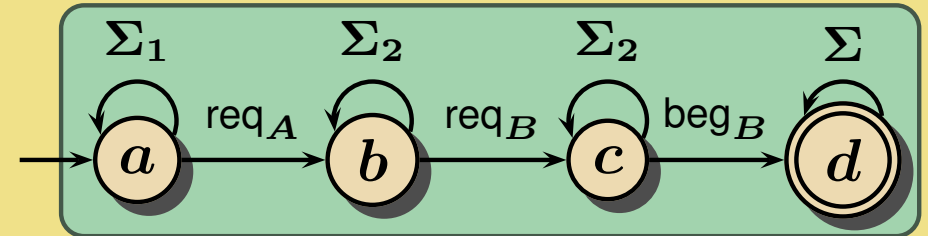
## Beispiel



$P$ : Falls das erste  $\text{req}_A$  vor dem ersten  $\text{req}_B$  vorkommt, so soll das erste  $\text{beg}_A$  vor dem ersten  $\text{beg}_B$  vorkommen

- Sei  $\mathcal{A}_T$  der NFA, der aus  $\mathcal{T}$  entsteht, indem alle Zustände als akzeptierend gewählt werden

- Wir konstruieren einen NFA  $\mathcal{A}_{\neg P}$ , der alle Strings akzeptiert, die  $P$  nicht erfüllen:



wobei  $\Sigma_1 \stackrel{\text{def}}{=} \Sigma - \{\text{req}_A, \text{req}_B\}$  und  $\Sigma_2 \stackrel{\text{def}}{=} \Sigma - \{\text{beg}_A\}$  sei

- $P$  gilt in  $\mathcal{T} \iff L(\mathcal{A}_T) \cap L(\mathcal{A}_{\neg P}) = \emptyset$

- Das können wir in zwei Schritten testen:
  - Wir konstruieren den NFA  $\mathcal{B}$  für  $L(\mathcal{A}_T) \cap L(\mathcal{A}_{\neg P})$
  - Wir prüfen, ob  $L(\mathcal{B}) = \emptyset$

- Aber: meistens werden Eigenschaften **unendlicher Berechnungen** spezifiziert  
 → Es werden Automaten für unendliche Strings verwendet:  **$\omega$ -Automaten**

## 6.1 Anwendungen regulärer Sprachen

6.1.1 Lexikalische Analyse

6.1.2 Zeichenkettensuche

6.1.3 Model Checking

▷ **6.1.4 UML**

6.1.5 Automatic Planning

6.1.6 XML

6.2 Erweiterungen der endlichen Automaten

## Anwendung: UML (1/2)

- **UML** (Unified Modeling Language):
  - Beschreibungssprache für die System-Analyse und System-Entwurf
- UML verwendet verschiedene Diagrammtypen:
  - Klassendiagramm
  - Use-Case-Diagramm
  - Aktivitätsdiagramm
  - Sequenzdiagramm
  - **Zustandsdiagramm**
  - Kollaborationsdiagramm
  - Komponentendiagramm
  - Verteilungsdiagramm

## 6.1 Anwendungen regulärer Sprachen

6.1.1 Lexikalische Analyse

6.1.2 Zeichenkettensuche

6.1.3 Model Checking

6.1.4 UML

▷ **6.1.5 Automatic Planning**

6.1.6 XML

6.2 Erweiterungen der endlichen Automaten



# Automatic Planning

- Beim **Planen** geht es darum, eine Schrittfolge zu finden, die eine gegebene Situation in eine angestrebte Zielsituation überführt
- Beim **automatischen Planen** soll eine solche Schrittfolge automatisch gefunden werden
- Situationen können dabei durch **Zustände** eines Transitionssystems modelliert werden
- Entsprechend werden einzelne Schritte durch Transitionen modelliert
- Ein **klassisches Planungs-Szenario** besteht aus:
  - einer Menge  $S$  von Situationen,
  - einer Anfangssituation  $i$ ,
  - einer Menge  $G$  von Zielsituationen,
  - einer Menge  $A$  von Aktionen,
  - einer Menge  $A(s) \subseteq A$  von möglichen Aktionen für jede Situation  $s$
  - einer Transition  $\delta(s, a)$  für jede mögliche Aktion  $a \in A(s)$
  - einer Kostenfunktion  $c : A^* \rightarrow \mathbb{R}_+$
- Herausforderung: Zustandsmenge wird riesig, deshalb Leerheitstest schwierig

## 6.1 Anwendungen regulärer Sprachen

6.1.1 Lexikalische Analyse

6.1.2 Zeichenkettensuche

6.1.3 Model Checking

6.1.4 UML

6.1.5 Automatic Planning

▷ **6.1.6 XML**

6.2 Erweiterungen der endlichen Automaten

# XML & Document Type Declarations (DTDs)

## Beispiel: DTD

```
<!DOCTYPE Composers [  
  <!ELEMENT Composers (Composer*)>  
  <!ELEMENT Composer (Name, Vita, Piece*)>  
  <!ELEMENT Vita (Born, Married*, Died?)>  
  <!ELEMENT Born (When, Where)>  
  <!ELEMENT Married (When, Whom)>  
  <!ELEMENT Died (When, Where)>  
  <!ELEMENT Piece (PTitle, PYear,  
    Instruments, Movements)>  

```

## Beispiel: XML-Dokument

```
<Composer>  
  <Name>Claude Debussy</Name>  
  <Vita>  
    <Born> <When>August 22, 1862</When><Where>Paris</Where></Born>  
    <Married><When>October 1899</When><Whom>Rosalie</Whom></Married>  
    <Married><When>January 1908</When><Whom>Emma</Whom></Married>  
    <Died><When>March 25, 1918</When><Where>Paris</Where></Died>  
  </Vita>  
  <Piece>  
    <PTitle>La Mer</PTitle>  
    <PYear>1905</PYear>  
    <Instruments>Large orchestra</Instruments>  
    <Movements>3</Movements> ...  
  </Piece>...  
</Composer>...
```

# Eine Einschränkung von DTDs ...

- Alles, was mit XML zu tun hat, wird vom **World Wide Web Consortium (W3C)** normiert ([www.w3.org](http://www.w3.org))

- Zum Thema **reguläre Ausdrücke in DTDs** sagt das W3C:

- „The content of an element matches a content model if and only if it is possible to trace out a path through the content model, obeying the sequence, choice, and repetition operators and matching each element in the content against an element type in the content model.“

 Also: reguläre Ausdrücke

- „For compatibility, it is an error if the content model allows an element to match more than one occurrence of an element type in the content model“

 Also: spezielle reguläre Ausdrücke

- „For more information, see **E Deterministic Content Models**“

Und in E steht dann:

- „As noted in 3.2.1 Element Content, it is required that content models in element type declarations be deterministic. This requirement is for compatibility with SGML (which calls deterministic content models "unambiguous"); “

- „For example, the content model  $((b, c)|(b, d))$  is non-deterministic, because given an initial  $b$  the XML processor cannot know which  $b$  in the model is being matched without looking ahead to see which element follows the  $b$ . In this case, the two references to  $b$  can be collapsed into a single reference, making the model read  $(b, (c|d))$ . An initial  $b$  now clearly matches only a single name in the content model. The processor doesn't need to look ahead to see what follows; either  $c$  or  $d$  would be accepted“

## ...und was das W3C über die Umsetzung verrät

- Und bei der genauen Erklärung kommen dann Automaten ins Spiel:
  - „More formally: a finite state automaton may be constructed from the content model using the standard algorithms, e.g. algorithm 3.5 in section 3.9 of Aho, Sethi, and Ullman [Aho/Ullman].“
  - „In many such algorithms, a follow set is constructed for each position in the regular expression (i.e., each leaf node in the syntax tree for the regular expression); if any position has a follow set in which more than one following position is labeled with the same element type name, then the content model is in error and may be reported as an error.“
  - „Algorithms exist which allow many but not all non-deterministic content models to be reduced automatically to equivalent deterministic models; see Brüggemann-Klein 1991 [Brüggemann-Klein]“

### Fragen

- Was ist eine sinnvolle Definition von **deterministischen regulären Ausdrücken (DREs)** ?
- Wie lassen sich DREs erkennen?
- Kann jede reguläre Sprache von einem DRE beschrieben werden?
- Wenn nicht, wie lässt sich für eine Sprache herausfinden, ob es geht?

### Beispiel

- $(a + b)^* a$  ist kein DRE, es gibt aber einen dazu äquivalenten DRE:  
$$b^* a (b^* a)^*$$
- $(a + b)^* a (a + b)$  ist kein DRE, es gibt aber keinen dazu äquivalenten DRE

# Deterministische reguläre Ausdrücke: Definition

- Die folgende Definition von DREs soll garantieren, dass beim Scannen eines Wortes, jedes Zeichen des Wortes immer in eindeutiger Weise **einem** Zeichen des regulären Ausdrucks entspricht

- Also:
  - $bc + bd$  ist kein DRE
  - $b(c + d)$  ist ein DRE

- Für ordnen jedem regulären Ausdruck  $\alpha$  einen **nummerierten regulären Ausdruck**  $\alpha^\sharp$  zu:
  - Dazu nummerieren wir die Positionen von  $\alpha$ , die Zeichen aus  $\Sigma$  tragen, von links nach rechts durch, beginnend mit 1:
    - \*  $(bc + bd)^\sharp = b_1c_2 + b_3d_4$
    - \*  $(b(c + d))^\sharp = b_1(c_2 + d_3)$

 Solche REs beschreiben also Strings über Alphabeten mit nummerierten Symbolen

- Ist  $x$  ein String mit nummerierten Zeichen, so bezeichnet  $x^\natural$  den String der durch Entfernen der Nummern entsteht:  
 $(b_1d_3)^\natural = bd$

- Ein RE  $\alpha$  heißt **deterministisch** wenn es keine erweiterten Strings  $x\sigma y$  und  $x\tau z$  in  $L(\alpha^\sharp)$  gibt mit:
  - $\sigma \neq \tau$
  - $\sigma^\natural = \tau^\natural$

- Also:
  - $b_1c_2 = \epsilon \cdot b_1 \cdot c_2 = x\sigma y$  und
  - $b_3d_4 = \epsilon \cdot b_3 \cdot d_4 = x\sigma z$zeigen, dass  $bc + bd$  nicht deterministisch ist

# Pingo

PINGO-Frage: [pingo.upb.de](http://pingo.upb.de)

- Welche der folgenden regulären Ausdrücke sind deterministisch?

(A)  $b(ba + ca)^*a + ac$

(B)  $b(ab + bc)^*a + ac$

(C)  $b(ab + ac)^*b + ac$

(D)  $b[(ba + ca)^*a + ac]$

# Erkennen von DREs

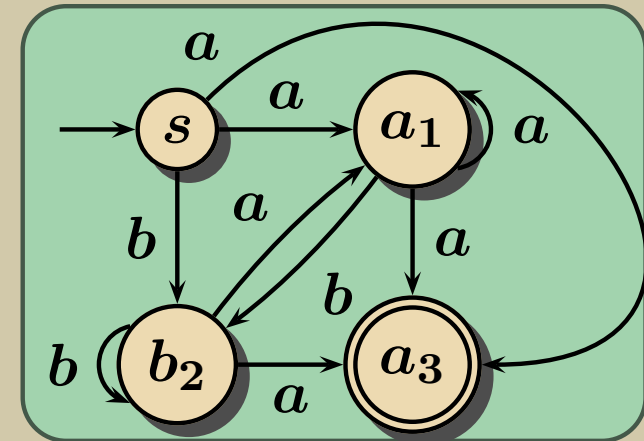
- Um zu erkennen, ob ein RE  $\alpha$  deterministisch ist, wandeln wir ihn auf eine neue Weise in einen NFA  $\mathcal{A}_\alpha$  (**Glushkov-Automat**) um:
  - Die Zustände des NFAs sind die Zeichen von  $\alpha^\#$  plus ein Startzustand  $s$
  - $\delta(s, \sigma) = \{\sigma_i \mid \sigma_i x \in L(\alpha^\#), \text{ für einen erweiterten String } x\}$
  - $\delta(\tau_i, \sigma) = \{\sigma_j \mid x \tau_i \sigma_j y \in L(\alpha^\#), \text{ für erweiterte Strings } x, y\}$
  - $F = \{\sigma_i \mid x \sigma_i \in L(\alpha^\#), \text{ für einen erweiterten String } x\}$

## Proposition 6.1

- Ein RE  $\alpha$  ist genau dann deterministisch, wenn  $\mathcal{A}_\alpha$  deterministisch ist

## Beispiel

- Beispielausdruck:
  - $\alpha = (a + b)^* a$
- Glushkov-Automat zu  $\alpha$ :



➡  $\alpha$  ist nicht deterministisch

- Eine reguläre Sprache hat genau dann einen DRE, wenn die Zusammenhangskomponenten ihres Minimalautomaten eine (ziemlich komplizierte) Bedingung erfüllen [Brüggemann-Klein, Wood 97]



# Inhalt

6.1 Anwendungen regulärer Sprachen


## 6.2 Erweiterungen der endlichen Automaten

▷ 6.2.1 Automaten mit Ausgabe

6.2.2 Zelluläre Automaten

6.2.3 Message Sequence Charts

## Automaten mit Ausgabe (1/3)

- Bisher haben wir nur Automaten betrachtet, die Sprachen definieren
  - Diese Automaten haben nur zwei mögliche „Ausgaben“: Akzeptieren oder Ablehnen
- Für den Einsatz in einem Compiler und für viele andere Zwecke ist es nützlich, Automaten um eine reichhaltigere Ausgabe-Komponente zu erweitern
  -  Dafür werden wir als erstes ein Beispiel betrachten

## Automaten mit Ausgabe (2/3)

### Beispiel

- Als Beispiel betrachten wir einen Automaten mit Ausgabe für die Addition zweier Binärzahlen  $x$  und  $y$
- Idee: der Automat liest jeweils ein Bit beider Zahlen und gibt ein Ergebnisbit aus

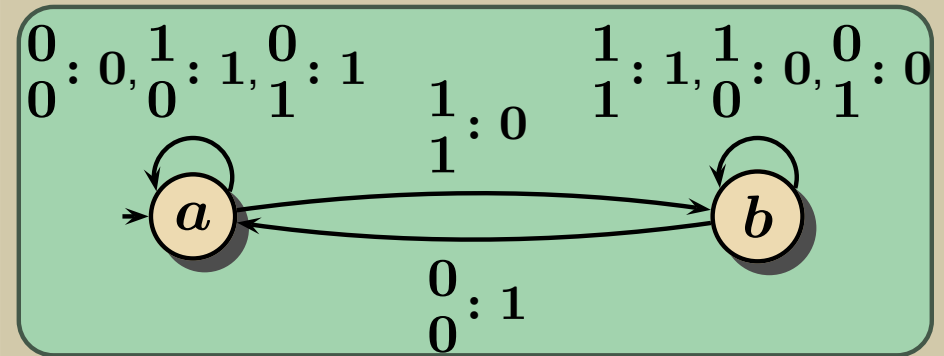
- Dazu werden  $x$  und  $y$  zusammen als String über  $\Sigma = \{0011\}$  kodiert
- Jedes Zeichen kodiert ein Bit von  $x$  und  $y$
- Da die Addition mit den niederwertigen Stellen beginnt, wird der Eingabe-String für den Automaten umgekehrt
- Damit die Eingabe wohlgeformt ist, werden  $x$  und  $y$  mit führenden Nullen aufgefüllt
- Um Platz für einen Übertrag zu haben, werden beide Zahlen um eine führende Null erweitert

- Für  $x = 26 = 11010_2$  und  $y = 79 = 1001111_2$  ergibt sich also der Eingabestring:  

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 |

### Beispiel

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 |



|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 |
|---|---|---|---|---|---|---|---|

- Dies ist ein Beispiel für einen **Mealy-Automaten**
- Er arbeitet deterministisch
- Für jedes Eingabezeichen gibt er ein Zeichen aus

## Automaten mit Ausgabe (3/3)

- Ein **Mealy-Automat** ist definiert wie ein DFA mit den folgenden Modifikationen:
  - Es gibt keine akzeptierende Menge  $F$
  - Es gibt ein **Ausgabe-Alphabet**  $\Gamma$  und eine Funktion  $\lambda : Q \times \Sigma \rightarrow \Gamma$ , die die auszugebenden Zeichen berechnet
- Mealy-Automaten können also nur String-Funktionen  $f$  berechnen, die, für alle Strings  $w$ , die Bedingung  $|f(w)| = |w|$  erfüllen
- Beispiel:  
Homomorphismen  $h$  mit  $h : \Sigma \rightarrow \Gamma$
- Ein alternatives Modell sind **Moore-Automaten**:
  - Bei ihnen hängt die Ausgabe nur vom jeweiligen Zustand ab:  $\lambda : Q \rightarrow \Gamma$
  - Bei Moore-Automaten ist
$$|f(w)| = |w| + 1$$
- Es gibt viele weitere Automatenmodelle zur Definition von Stringfunktionen
- Ein sehr allgemeines Modell stellen **String-Transducer** dar:
  - Sie arbeiten nichtdeterministisch
  - Sie können in jedem Schritt einen String ausgeben (also:  $\lambda(q, \sigma) \subseteq \Gamma^*$ )
- Um einen Scanner zu modellieren wäre also ein deterministischer String-Transducer ein geeignetes Modell

# Inhalt

6.1 Anwendungen regulärer Sprachen

## **6.2 Erweiterungen der endlichen Automaten**

6.2.1 Automaten mit Ausgabe

▷ **6.2.2 Zelluläre Automaten**

6.2.3 Message Sequence Charts

## Weitere Automatenmodelle: Zelluläre Automaten

- Neben einer Vielzahl von Varianten endlicher Automaten werden für viele Modellierungsaufgaben auch Modelle vernetzter Automaten verwendet
- Durch Kommunikation zwischen den Einzelautomaten können dabei äußerst komplizierte Berechnungsmodelle entstehen

- Als erstes Beispiel eines solchen Automatennetzes betrachten wir **zelluläre Automaten**

- In einem Gitter (typischerweise:  $\mathbb{Z} \times \mathbb{Z}$ ) sind „Automaten“ verteilt
- Der Zustand jedes Automaten zum nächsten Zeitpunkt hängt von seinem aktuellen Zustand sowie von den Zuständen seiner Nachbarautomaten ab

- Einfachstes Beispiel: **Conways Spiel des Lebens** (siehe: Wikipedia):
  - Jeder Automat kann zwei Zustände annehmen:  $\square$  oder  $\blacksquare$
  - Ein Automat nimmt den Zustand  $\blacksquare$  genau dann an, wenn zuvor genau drei seiner Nachbarn im Zustand  $\blacksquare$  waren, andernfalls geht er in den Zustand  $\square$

- Zelluläre Automaten eignen sich zur Simulation natürlicher und künstlicher Prozesse wie Verhalten von Gasen, Verkehrssimulation, Wachstumsprozesse,...

# Inhalt

6.1 Anwendungen regulärer Sprachen

## **6.2 Erweiterungen der endlichen Automaten**

6.2.1 Automaten mit Ausgabe

6.2.2 Zelluläre Automaten

▷ **6.2.3 Message Sequence Charts**

## Weitere Automatenmodelle: Message Sequence Charts

- **Message Sequence Charts (MSCs)** bestehen aus endlichen Automaten, die sich über FIFO-Kommunikationskanäle Nachrichten senden können
  - Damit können verteilte Systeme simuliert werden, wie zum Beispiel Telekommunikationssysteme oder auch Web Services
  - Von internen Berechnungen der beteiligten Prozessoren wird üblicherweise abstrahiert
- Statt Zeichen werden Aktionen „konsumierte“, genauer:
    - Es gibt zwei Arten von Aktionen:
      - \* Senden einer Nachricht: „!a“
      - \* Empfangen einer Nachricht: „?a“
    - Nachrichten werden in den Kanälen zwischengespeichert
  - Ein Beispiel findet sich im Buch von Vossen und Witt, Kapitel 4, siehe auch: [http://dbms.uni-muenster.de/publications/books/Grundkurs\\_Theoretische\\_Informatik/](http://dbms.uni-muenster.de/publications/books/Grundkurs_Theoretische_Informatik/)
  - Gewisse Eigenschaften solcher Systeme lassen sich dann automatisch testen (zumindest unter einschränkenden Annahmen)



# Zusammenfassung

- Reguläre Sprachen und insbesondere endliche Automaten haben unzählige Anwendungen
- Es gibt viele Varianten endlicher Automaten
- Es gibt viele weitere Automatenmodelle, die das Grundprinzip endlicher Automaten in verschiedener Hinsicht verallgemeinern

## Literatur für dieses Kapitel

- Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and Francois Yergeau. Extensible markup language (XML) 1.0 (fifth edition). Technical report, W3C, 2008. [www.w3.org/TR/2008/REC-xml-20081126](http://www.w3.org/TR/2008/REC-xml-20081126)
- A. Brüggemann-Klein and Wood D. Deterministic regular languages. Technical Report Bericht 38, Universität Freiburg, Institut für Informatik, 1991
- A. Brüggemann-Klein and D. Wood. One-unambiguous regular languages. *Information and Computation*, 142(2):182–206, 1998
- Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: Theory and Practice*. Morgan Kaufmann, San Francisco, CA, 2004
- G. Vossen and U. Witt. *Grundlagen der Theoretischen Informatik mit Anwendungen*. Vieweg, 2000

# Grundbegriffe der Theoretischen Informatik

Sommersemester 2016 - Thomas Schwentick

Teil B: Kontextfreie Sprachen

7: Kontextfreie Grammatiken

Version von: 12. Mai 2016 (13:49)

# Inhalt

## ▷ 7.1 Kontextfreie Grammatiken: Beispiele und Definition

7.2 Ableitungen und Ableitungsbäume

7.3 Mehrdeutigkeit

7.4 Konstruktion von Grammatiken

7.5 Die Chomsky-Hierarchie

7.6 Erweiterte kontextfreie Grammatiken

# Motivation

- Ziel: Methode zur Beschreibung der Syntax von Programmiersprachen
- Wir haben im ersten Teil der Vorlesung gesehen, dass sich Bezeichner in Programmentexten durch reguläre Sprachen beschreiben lassen

- Wie ist es mit anderen Konstrukten, die in Programmen vorkommen?

## Beispiel

- In Programmen kommen häufig arithmetische Ausdrücke wie z.B.  $((a + b) \times (a + a)) \times b$  vor
- Die Menge  $M$  der wohlgeformten arithmetischen Ausdrücke über einem Alphabet wie  $\{a, b, +, \times, „(“, „)“\}$  sollte von einer Methode zur Spezifikation der Syntax einer Programmiersprache beschreibbar sein
- Ist  $M$  regulär?

## „Proposition“

- Die Menge  $M$  der wohlgeformten arithmetischen Ausdrücke über  $\{a, b, +, \times, „(“, „)“\}$  ist nicht regulär

## Beweisidee

- Dazu genügt es, die „Zukunftssprachen“  $M/v_n$  der folgenden Strings  $v_n$ , für  $n \geq 1$  zu betrachten:
  - $v_n \stackrel{\text{def}}{=} ({}^n a$
- Für jedes  $n$  enthält  $M/v_n$  den String  $[+a]^n$ , aber keinen String der Art  $[+a]^m$  für  $m \neq n$ 
  - ✎ Hier sind „(“ und „)“ Symbole des Alphabets und „[“ und „]“ sind Meta-Symbole
- Also hat  $M$  unendlich viele Nerode-Klassen...
- Wie können wir Sprachen wie  $M$  beschreiben?

# Nicht-reguläre Sprachen

- Die folgenden Sprachen sind nicht regulär:

- (a)  $L_{\text{pali}} = \{w \in \{a, b\}^* \mid w^R = w\}$ 
  - $w^R \stackrel{\text{def}}{\Leftrightarrow}$  Umkehrung von  $w$ , also  $(abdc)^R = cdba$
- (b)  $L_{ab} = \{a^n b^n \mid n \geq 0\}$
- (c)  $L_{\text{doppel}} = \{ww \mid w \in \{a, b\}^*\}$
- (d)  $L_{\text{quad}} = \{a^{n^2} \mid n > 0\}$
- (e)  $L_{\text{prim}} = \{a^p \mid p \text{ ist Primzahl}\}$
- (f)  $L_{()} =$  Menge aller wohlgeformten Klammerausdrücke

- Für einige dieser Sprachen bieten kontextfreie Grammatiken eine „einfache“ Beschreibungsmöglichkeit

## Beispiel

- Wir betrachten zunächst die Sprache  $L_{\text{pali}}$  der Palindrome über  $\{a, b\}$ :

- Palindrome über  $\{a, b\}$  lassen sich leicht induktiv definieren:

- $\epsilon, a, b$  sind Palindrome
- Ist  $w$  ein Palindrom, so auch  $awa$
- Ist  $w$  ein Palindrom, so auch  $bwb$

- Lässt sich das kompakter aufschreiben?

- Idee: Schreibe  $P \rightarrow w$  statt „ $w$  ist Palindrom“

- Dann lassen sich die genannten Regeln wie folgt zusammenfassen:

$$P \rightarrow \epsilon \quad (1)$$

$$P \rightarrow a \quad (2)$$

$$P \rightarrow b \quad (3)$$

$$P \rightarrow aPa \quad (4)$$

$$P \rightarrow bPb \quad (5)$$

# Palindrome erzeugen

- Die Regeln für Palindrome,

$$P \rightarrow \epsilon \quad (1)$$

$$P \rightarrow a \quad (2)$$

$$P \rightarrow b \quad (3)$$

$$P \rightarrow aPa \quad (4)$$

$$P \rightarrow bPb \quad (5)$$

lassen sich auf verschiedene Weisen interpretieren:

- Wir können Regeln der Art  $P \rightarrow w$  als Rezepte zum Erzeugen von Palindromen „bottom-up“ auffassen:

$b$  ist Palindrom (3)

$\Rightarrow aba$  ist Palindrom (4)

$\Rightarrow babab$  ist Palindrom (5)

$\Rightarrow bbababb$  ist Palindrom (5)

- Wir können Regeln der Art  $P \rightarrow w$  auch als Anleitung zum Erzeugen von Palindromen „top-down“ auffassen:

$$P \xRightarrow{(5)} bPb$$

$$\xRightarrow{(5)} bbPbb$$

$$\xRightarrow{(4)} bbaPabb$$

$$\xRightarrow{(3)} bbababb$$

## Ein weiteres Beispiel

- Wir beschreiben jetzt **arithmetische Ausdrücke mit Bezeichnern**:

- Operationssymbole:  $+$ ,  $\times$
- Ein **Bezeichner** sei ein String der Form  $(a + b)(a + b + 0 + 1)^*$ 
  - Zum Beispiel:  $bb1$
- Dazu kommen noch Klammern
- Das Alphabet für unsere arithmetischen Ausdrücke ist also:
 
$$\Sigma = \{a, b, 0, 1, (, ), +, \times\}$$
- Ein Beispiel-Ausdruck:
 
$$(a + b0) \times bb1 + a0$$

- Induktive „Definition“ arithmetischer Ausdrücke:
  - Bezeichner, oder
  - Ausdruck  $+$  Ausdruck, oder
  - Ausdruck  $\times$  Ausdruck, oder
  - $(\text{Ausdruck})$

- In „Regel-Schreibweise“:

$$A \rightarrow B$$

$$A \rightarrow A + A$$

$$A \rightarrow A \times A$$

$$A \rightarrow (A)$$

$$B \rightarrow a$$

$$B \rightarrow b$$

$$B \rightarrow Ba$$

$$B \rightarrow Bb$$

$$B \rightarrow B0$$

$$B \rightarrow B1$$

- Den Beispiel-Ausdruck erhalten wir dann so:

$$A \Rightarrow A + A$$

$$\Rightarrow A \times A + A$$

$$\Rightarrow (A) \times A + A$$

$$\Rightarrow (A + A) \times A + A$$

$$\Rightarrow (B + A) \times A + A$$

$$\Rightarrow (a + A) \times A + A$$

$$\Rightarrow (a + B) \times A + A$$

$$\Rightarrow (a + B0) \times A + A$$

$$\Rightarrow (a + b0) \times A + A$$

$$\Rightarrow (a + b0) \times B + A$$

$$\Rightarrow (a + b0) \times B1 + A$$

$$\Rightarrow (a + b0) \times Bb1 + A$$

$$\Rightarrow (a + b0) \times bb1 + A$$

$$\Rightarrow (a + b0) \times bb1 + B$$

$$\Rightarrow (a + b0) \times bb1 + B0$$

$$\Rightarrow (a + b0) \times bb1 + a0$$



# Kontextfreie Grammatiken: Definition

## Definition

- Eine **kontextfreie Grammatik**  $G = (V, \Sigma, S, P)$  besteht aus
  - einer endlichen Menge  $V$  von **Variablen**
  - einem Alphabet  $\Sigma$
  - einem **Startsymbol**  $S \in V$ ,
  - einer endlichen Menge  $P$  von **Regeln**:
- Dabei muss gelten:  $\Sigma \cap V = \emptyset$

 Statt  $(A, BC) \in P$  schreiben wir  $A \rightarrow BC$

## Beispiel

- Die Regeln für arithmetische Ausdrücke sind also Regeln einer kontextfreien Grammatik
- Formal lässt sich diese Grammatik wie folgt aufschreiben:  
 $(\{A, B\}, \{a, b, 0, 1, (, ), +, \times\}, A, \{(B, a), (B, b), (B, Ba), \dots, (A, (A))\})$

- Übliche Bezeichnungen:
  - Elemente von  $V \cup \Sigma$ : **Symbole**
  - Elemente von  $\Sigma$ : **Terminalsymbole**
  - Elemente von  $V$ : Variablen oder **Nicht-Terminalsymbole**
  - Regeln aus  $P$ : **Produktionen**

- Mit  $|G|$  bezeichnen wir die Größe einer Grammatik:

$$\underline{|G|} \stackrel{\text{def}}{=} |V| + |\Sigma| + \sum_{(X, \alpha) \in P} (|\alpha| + 1)$$

- Die Grammatik für arithmetische Ausdrücke hat die Größe 40

# Kontextfreie Grammatiken: Kompakte Schreibweise

- Alle Regeln mit derselben linken Seite werden üblicherweise zusammengefasst:

– Statt

$$* X \rightarrow \alpha_1$$

$$* X \rightarrow \alpha_2$$

$$* \dots$$

$$* X \rightarrow \alpha_k$$

schreiben wir also:

$$X \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_k$$

## Beispiel

$$A \rightarrow B \mid A + A \mid A \times A \mid (A)$$

$$B \rightarrow a \mid b \mid Ba \mid Bb \mid B0 \mid B1$$

- Meistens werden Grammatiken einfach durch die Angabe ihrer zusammengefassten Regeln beschrieben
- Dabei gelten folgende Konventionen:
  - Alle Symbole, die links in einer Regel vorkommen, sind **Variablen**
  - Das **Startsymbol** ist die Variable der linken Seite der ersten Regel

# Kontextfreie Grammatiken: Semantik

- **Informell:** in einem Ableitungsschritt wird eine Variable  $X$  durch eine rechte Seite  $\gamma$  einer Regel  $X \rightarrow \gamma$  ersetzt, z.B.:
  - $bPb \Rightarrow_G baPab$

## Definition

- Sei  $G = (V, \Sigma, S, P)$  eine kontextfreie Grammatik
- Eine **Satzform** ist ein String aus  $(V \cup \Sigma)^*$
- Eine Satzform  $v$  geht aus einer Satzform  $u$  in einem Ableitungsschritt hervor, wenn es
  - Satzformen  $\alpha, \beta, \gamma$ ,
  - eine Variable  $X$  und
  - eine Regel  $X \rightarrow \gamma$  in  $P$gibt, so dass
  - $u = \alpha X \beta$  und
  - $v = \alpha \gamma \beta$
- Schreibweise:  $\underline{u \Rightarrow_G v}$

- **Informell:** eine Ableitung ist eine Folge von Ableitungsschritten

## Definition

- Sei  $G = (V, \Sigma, S, P)$  eine kontextfreie Grammatik
- Eine Folge  $u_0, u_1, \dots, u_n$  heißt **Ableitung**, falls für jedes  $i \in \{1, \dots, n\}$  gilt:  $u_{i-1} \Rightarrow_G u_i$
- Schreibweise:  $\underline{u_0 \Rightarrow_G^n u_n}$ 
  - oder  $\underline{u_0 \Rightarrow_G^* u_n}$ , wenn es auf die Zahl der Schritte nicht ankommt
- Wir sagen auch:  $u_n$  ist aus  $u_0$  (in  $n$  Schritten) ableitbar
- $\underline{L(G)} \stackrel{\text{def}}{=} \{w \in \Sigma^* \mid S \Rightarrow_G^* w\}$   
ist die **von  $G$  erzeugte Sprache**
- Eine Sprache  $L$  heißt **kontextfrei**, falls  $L = L(G)$  für eine kontextfreie Grammatik  $G$

# Ableitung: Beispiel

## Beispiel

$$\begin{aligned} A &\Rightarrow A + A \\ &\Rightarrow A \times A + A \\ &\Rightarrow (A) \times A + A \\ &\Rightarrow (A + A) \times A + A \\ &\Rightarrow (B + A) \times A + A \\ &\Rightarrow (a + A) \times A + A \\ &\Rightarrow (a + B) \times A + A \\ &\Rightarrow (a + B0) \times A + A \\ &\Rightarrow (a + b0) \times A + A \\ &\Rightarrow (a + b0) \times B + A \\ &\Rightarrow (a + b0) \times B1 + A \\ &\Rightarrow (a + b0) \times Bb1 + A \\ &\Rightarrow (a + b0) \times bb1 + A \\ &\Rightarrow (a + b0) \times bb1 + B \\ &\Rightarrow (a + b0) \times bb1 + B0 \\ &\Rightarrow (a + b0) \times bb1 + a0 \end{aligned}$$

# Inhalt

7.1 Kontextfreie Grammatiken: Beispiele und Definition

▷ **7.2 Ableitungen und Ableitungsbäume**

7.3 Mehrdeutigkeit

7.4 Konstruktion von Grammatiken

7.5 Die Chomsky-Hierarchie

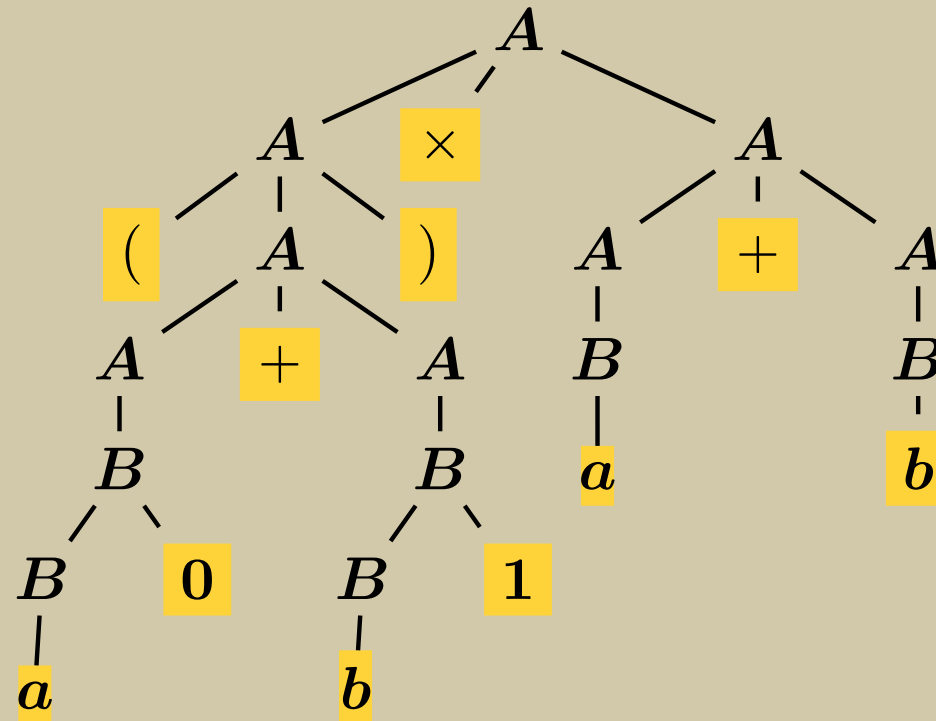
7.6 Erweiterte kontextfreie Grammatiken

# Ableitungsbäume

- Ableitungen lassen sich durch **Ableitungsbäume** visualisieren

## Beispiel

$$\begin{aligned} A &\rightarrow B \mid A + A \mid A \times A \mid (A) \\ B &\rightarrow a \mid b \mid Ba \mid Bb \mid B0 \mid B1 \end{aligned}$$



- Dieser Ableitungsbaum hat den **Blattstring**  $(a0 + b1) \times a + b$

- $S \Rightarrow_G^* w \iff$  es gibt einen Ableitungsbaum  
mit Wurzel  $S$  und Blattstring  $w$

# Ableitungsbäume und Ableitungen: Definitionen (1/2)

## Definition

- Ein **Ableitungsbaum** zu einer kontextfreien Grammatik  $G = (V, \Sigma, S, P)$  ist ein geordneter Baum  $T$  mit Wurzel, der die folgenden Eigenschaften hat
  - Die **Blätter** sind mit Terminalsymbolen oder mit  $\epsilon$  markiert
  - Die **inneren Knoten** sind mit Variablen aus  $V$  markiert
  - Die **Wurzel** ist mit  $S$  markiert
  - Für jeden inneren Knoten  $v$  gibt es eine Regel  $X \rightarrow \alpha$  aus  $P$ , so dass
    - $v$  mit  $X$  markiert ist und
    - die Kinder von  $v$  von links nach rechts mit den Zeichen aus  $\alpha$  markiert sind
- Der **Blattstring** eines Ableitungsbaumes besteht aus den Symbolen der Blätter, die nicht mit  $\epsilon$  markiert sind, von links nach rechts gelesen
- Ist  $T$  ein Ableitungsbaum mit Blattstring  $w$ , so nennen wir  $T$  **Ableitungsbaum für  $w$**

# Ableitungsbäume und Ableitungen: Definitionen (2/2)

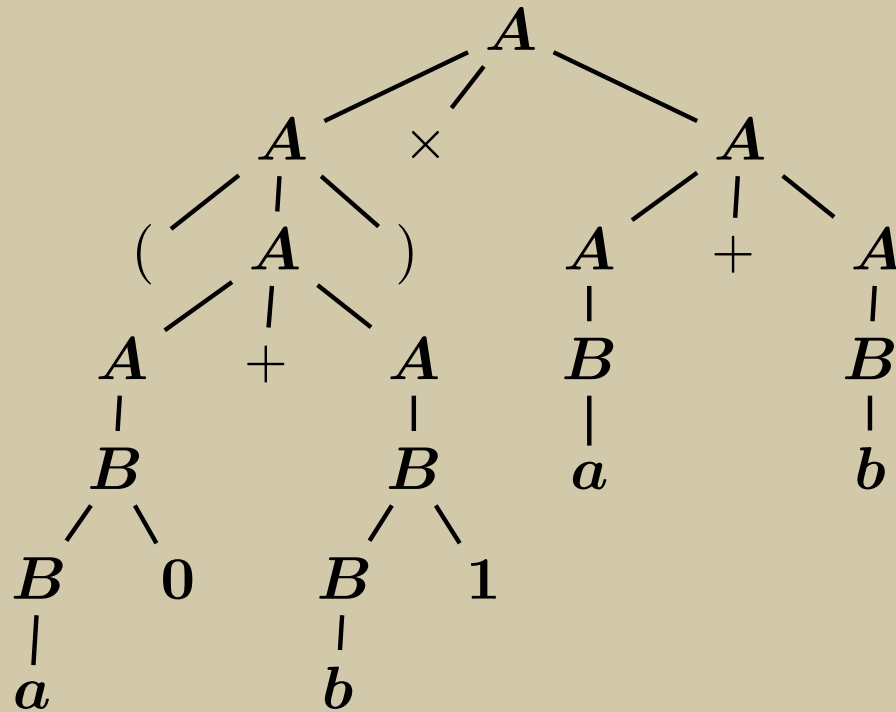
## Definition

- Zwei spezielle Arten von Ableitungen:
  - Linksableitung: Ableitung, in der in jedem Schritt die am weitesten links stehende Variable ersetzt wird
    - \* Schreibweise:  $S \Rightarrow_l^* w$  bzw.  $S \Rightarrow_{G,l}^* w$
  - Rechtsableitung: analog
    - \* Schreibweise:  $S \Rightarrow_r^* w$  bzw.  $S \Rightarrow_{G,r}^* w$
- Zu jedem Ableitungsbaum gibt es je eine Links- und eine Rechtsableitung



# Eine Linksableitung

## Beispiel

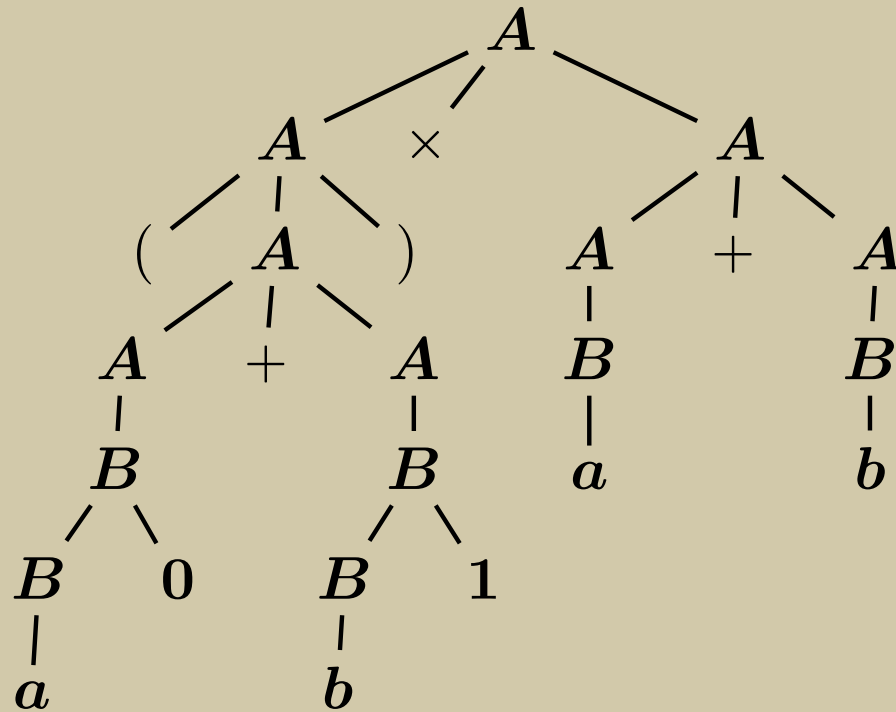


## Beispiel

$$\begin{aligned}
 A &\Rightarrow_l A \times A \\
 &\Rightarrow_l (A) \times A \\
 &\Rightarrow_l (A + A) \times A \\
 &\Rightarrow_l (B + A) \times A \\
 &\Rightarrow_l (B0 + A) \times A \\
 &\Rightarrow_l (a0 + A) \times A \\
 &\Rightarrow_l (a0 + B) \times A \\
 &\Rightarrow_l (a0 + B1) \times A \\
 &\Rightarrow_l (a0 + b1) \times A \\
 &\Rightarrow_l (a0 + b1) \times A + A \\
 &\Rightarrow_l (a0 + b1) \times B + A \\
 &\Rightarrow_l (a0 + b1) \times a + A \\
 &\Rightarrow_l (a0 + b1) \times a + B \\
 &\Rightarrow_l (a0 + b1) \times a + b
 \end{aligned}$$

# Eine Rechtsableitung

## Beispiel



## Beispiel

$$\begin{aligned}
 A &\Rightarrow_r A \times A \\
 &\Rightarrow_r A \times A + A \\
 &\Rightarrow_r A \times A + B \\
 &\Rightarrow_r A \times A + b \\
 &\Rightarrow_r A \times B + b \\
 &\Rightarrow_r A \times a + b \\
 &\Rightarrow_r (A) \times a + b \\
 &\Rightarrow_r (A + A) \times a + b \\
 &\Rightarrow_r (A + B) \times a + b \\
 &\Rightarrow_r (A + B1) \times a + b \\
 &\Rightarrow_r (A + b1) \times a + b \\
 &\Rightarrow_r (B + b1) \times a + b \\
 &\Rightarrow_r (B0 + b1) \times a + b \\
 &\Rightarrow_r (a0 + b1) \times a + b
 \end{aligned}$$

# Inhalt

7.1 Kontextfreie Grammatiken: Beispiele und Definition

7.2 Ableitungen und Ableitungsbäume

▷ **7.3 Mehrdeutigkeit**

7.4 Konstruktion von Grammatiken

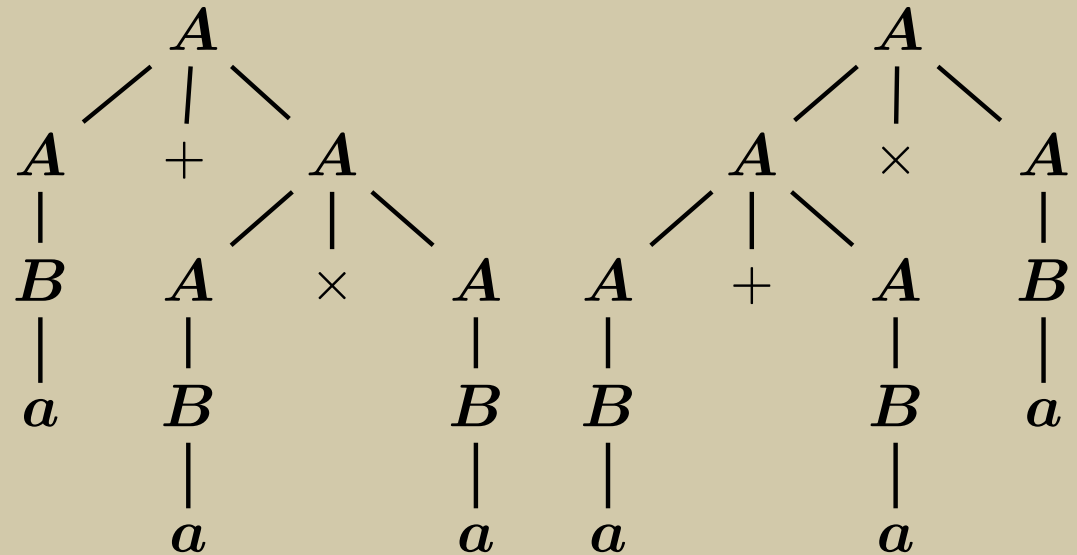
7.5 Die Chomsky-Hierarchie

7.6 Erweiterte kontextfreie Grammatiken

# Eindeutige vs. mehrdeutige Grammatiken (1/2)

- Wir haben gesehen: im Allgemeinen kann es zu einem Ableitungsbaum verschiedene Ableitungen geben
- Also kann es zu einem String mehrere Ableitungen haben
- Kann derselbe String verschiedene Ableitungsbäume haben?

## Beispiel



- Für Compiler ist es ungünstig, wenn der Ableitungsbaum nicht eindeutig ist:
  - Denn der Ableitungsbaum soll die Auswertungsreihenfolge eines Ausdrucks festlegen

## Beispiel

- Der linke Baum entspricht der Auswertung  $a + (a \times a)$
- Der rechte Baum entspricht der Auswertung  $(a + a) \times a$

## Eindeutige vs. mehrdeutige Grammatiken (2/2)

### Definition

- Eine kontextfreie Grammatik  $G$  heißt **mehrdeutig**, falls es einen String  $w$  gibt, der zwei verschiedene Ableitungsbäume bezüglich  $G$  hat
  - Andernfalls heißt  $G$  **eindeutig**

### Beispiel

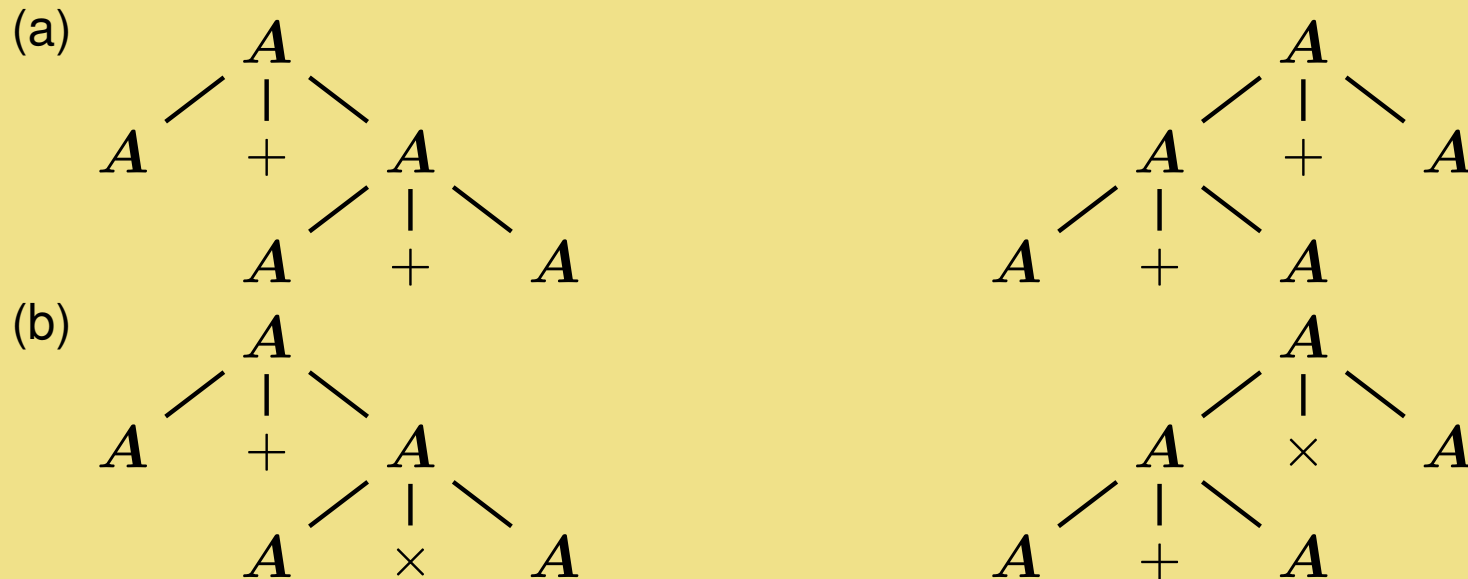
- Die Grammatik für arithmetische Ausdrücke ist mehrdeutig
- Wie sehen gleich: die Sprache der arithmetischen Ausdrücke hat auch eine eindeutige Grammatik

- Wie schwierig ist es zu testen, ob eine Grammatik eindeutig ist?
  - Mehr als schwierig:  
es gibt kein allgemeines Verfahren dafür
- Teil C der Vorlesung

# Arithmetische Ausdrücke

$$A \rightarrow B \mid A + A \mid A \times A \mid (A)$$
$$B \rightarrow a \mid b \mid Ba \mid Bb \mid B0 \mid B1$$

- Die obige Grammatik ist auf zweifache Weise mehrdeutig:



- Die Mehrdeutigkeit (a) lässt sich auf einfache Weise beheben:
  - Da die Operation  $+$  assoziativ ist, genügt es, immer die rechte Struktur zu erzeugen:  $A \rightarrow A + B \mid B$
- Die Mehrdeutigkeit (b) hängt mit der Bindungsstärke der Operatoren zusammen („Punkt vor Strich“)
  - Aber sie lässt sich ebenfalls beheben...

# Eine eindeutige Grammatik für arithmetische Ausdrücke

- **Ziel:** eindeutige Grammatik für arithmetische Ausdrücke
- **Idee:** Operatoren mit geringer Bindung werden später ausgewertet und sollten im Baum deshalb weit oben sein
  - ➔ Die Regeln für  $+$  müssen in der Grammatik in „einer höheren Ebene“ vorkommen als die Regeln für  $\times$

- Modifizierte Grammatik:

$$A \rightarrow A + T \mid T$$

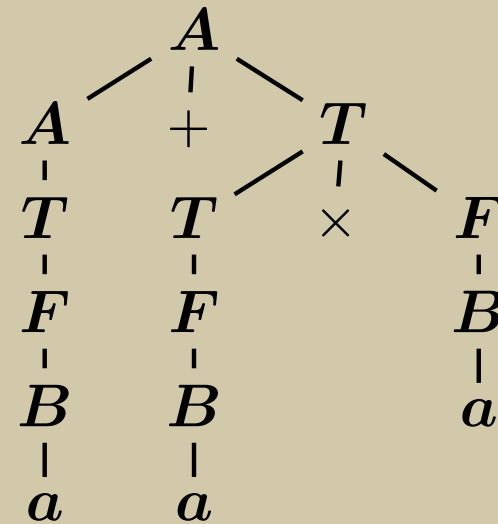
$$T \rightarrow T \times F \mid F$$

$$F \rightarrow (A) \mid B$$

$$B \rightarrow a \mid b \mid Ba \mid Bb \mid B0 \mid B1$$

- Bezüglich dieser Grammatik hat  $a + a \times a$  eine eindeutige Ableitung

## Beispiel



## Fakt

- Die modifizierte Grammatik für arithmetische Ausdrücke ist eindeutig

## Beweisidee

- Zeige für alle Variablen der Grammatik, dass jede aus ihr ableitbare Satzform einen eindeutigen Ableitungsbaum hat
- Dies lässt sich durch Induktion nach der Höhe des minimalen Ableitungsbaums beweisen

# Inhärent mehrdeutige kontextfreie Sprachen

## Definition

- Eine kontextfreie Sprache  $L$  heißt **eindeutig**, falls sie eine eindeutige Grammatik hat
  - Andernfalls heißt  $L$  **inhärent mehrdeutig**

## Beispiel

$$S \rightarrow AX_{bc} \mid X_{ab}C \mid \epsilon$$

$$C \rightarrow Cc \mid \epsilon$$

$$A \rightarrow Aa \mid \epsilon$$

$$X_{ab} \rightarrow aX_{ab}b \mid \epsilon$$

$$X_{bc} \rightarrow bX_{bc}c \mid \epsilon$$

ist eine mehrdeutige kontextfreie Grammatik für die Sprache  $L_{abc} \stackrel{\text{def}}{=} \{a^i b^j c^k \mid i = j \text{ oder } j = k\}$

- $L_{abc}$  ist inhärent mehrdeutig
- Intuitiver Grund: Strings der Form  $a^n b^n c^n$  erfüllen beide Bedingungen „ $i = j$ “ und „ $j = k$ “ und haben deshalb zwei Ableitungsbäume
  - Aber: der Beweis dafür ist ziemlich kompliziert
- Es gibt auch kein allgemeines Verfahren, das entscheidet, ob die Sprache einer gegebenen kontextfreien Grammatik eindeutig ist



# Inhalt

7.1 Kontextfreie Grammatiken: Beispiele und Definition

7.2 Ableitungen und Ableitungsbäume

7.3 Mehrdeutigkeit

▷ **7.4 Konstruktion von Grammatiken**

7.5 Die Chomsky-Hierarchie

7.6 Erweiterte kontextfreie Grammatiken

# Konstruktion einer kontextfreien Grammatik

## Beispiel

- Wir konstruieren eine kontextfreie Grammatik für  $L_{ab} = \{a^n b^n \mid n \geq 0\}$
- Immer, wenn sie vorne ein  $a$  erzeugt, soll sie hinten ein  $b$  erzeugen:  $S \rightarrow aSb$
- Irgendwann soll sie damit aufhören:  $S \rightarrow \epsilon$
- Insgesamt also:  $S \rightarrow aSb \mid \epsilon$
- Beispielableitung:

$$\begin{aligned} S &\Rightarrow aSb \\ &\Rightarrow aaSbb \\ &\Rightarrow aaaSbbb \\ &\Rightarrow aaabbbb \end{aligned}$$

# Eine etwas kompliziertere kontextfreie Grammatik (1/2)

- Wir betrachten jetzt ein komplizierteres Beispiel einer kontextfreien Grammatik
- Es illustriert, dass eine rekursive Herangehensweise bei der Konstruktion kontextfreier Grammatiken helfen kann

## Beispiel

- Sei  $L_{a=b} \stackrel{\text{def}}{=} \{w \in \{a, b\}^* \mid \#_a(w) = \#_b(w)\}$ 
  - Zur Erinnerung:  $\#_a(w)$  ist die Anzahl der Positionen in  $w$ , an denen  $a$  steht
- Wie lassen sich die Strings dieser Sprache erzeugen?

## Beispiel

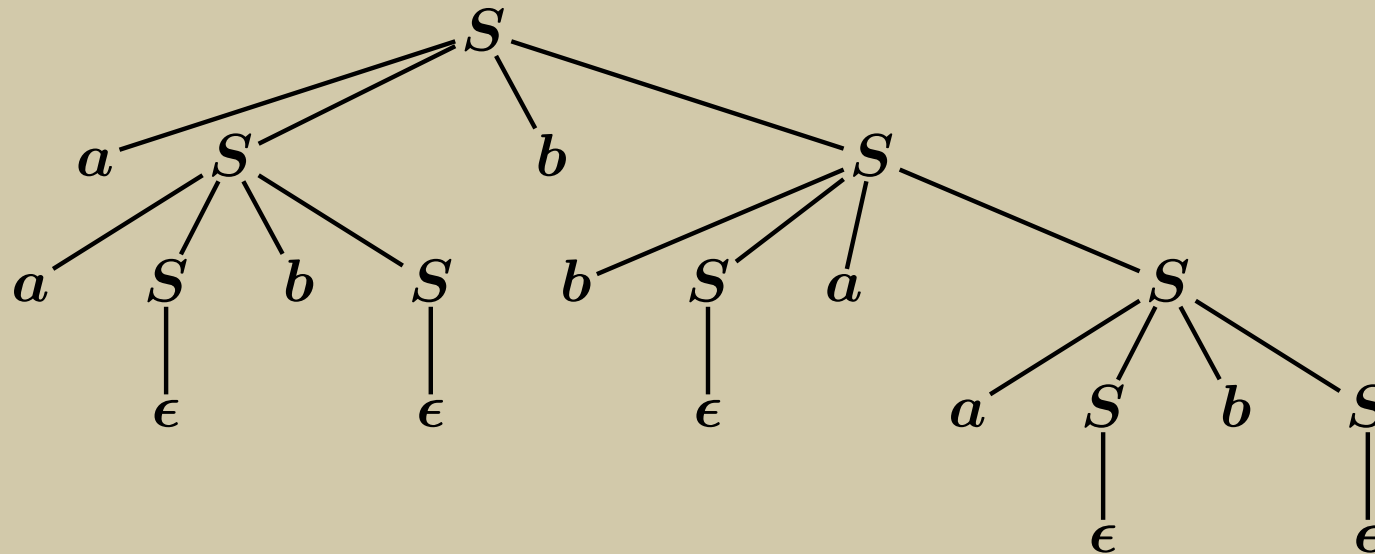
- **Idee:** Strings aus  $L_{a=b}$  lassen sich schreiben
  - in der Form  $aubv$  oder
  - in der Form  $buav$ ,wobei sowohl in  $u$  als auch in  $v$  gleich viele  $a$ 's wie  $b$ 's vorkommen
  - $u$  ist dabei der kürzeste String hinter  $a$  (bzw.  $b$ ), für den  $aub$  (bzw.  $bua$ ) gleich viele  $a$ 's wie  $b$ 's hat

- Eine Grammatik für  $L_{a=b}$  könnte also die Regeln  $S \rightarrow aSbS$  und  $S \rightarrow bSaS$  verwenden
- Der Leerstring ist natürlich auch noch in  $L_{a=b}$
- Insgesamt ergibt sich also die folgende Grammatik  $G_{a=b}$ :
$$S \rightarrow aSbS \mid bSaS \mid \epsilon$$

## Eine etwas kompliziertere kontextfreie Grammatik (2/2)

### Beispiel

- Wir betrachten einen Ableitungsbaum für den String  $aabbbaab$
- Zur Erinnerung:  $G_{a=b}$  ist  $S \rightarrow aSbS \mid bSaS \mid \epsilon$



- Dass  $G_{a=b}$  wirklich genau die Strings der Sprache  $L_{a=b}$  erzeugt, zeigen wir in Kapitel 12

# Inhalt

7.1 Kontextfreie Grammatiken: Beispiele und Definition

7.2 Ableitungen und Ableitungsbäume

7.3 Mehrdeutigkeit

7.4 Konstruktion von Grammatiken

▷ **7.5 Die Chomsky-Hierarchie**

7.6 Erweiterte kontextfreie Grammatiken

# Chomsky-Grammatiken: Definition

- Kontextfreie Grammatiken sind der (mit Abstand bedeutendste) Spezialfall eines allgemeineren Konzeptes
- **Chomsky-Grammatiken** wurden in den 50er Jahren von dem Linguisten Noam Chomsky im Zusammenhang der Analyse natürlicher Sprachen eingeführt
- Sie erlauben auf der linken Seite einer Regel nicht nur Variablen sondern Satzformen, z.B.:
  - $aBC \rightarrow De$

## Definition

- Eine **Chomsky-Grammatik** ist ein 4-Tupel  $(V, \Sigma, P, S)$  mit  $V, \Sigma, S$  wie zuvor und  $P \subseteq (V \cup \Sigma)^* V (V \cup \Sigma)^* \times (V \cup \Sigma)^*$
- Auf der linken Seite jeder Regel ist also immer ein String über  $V \cup \Sigma$  mit mindestens einer Variablen
- Ableitungsschritt:  $\alpha\beta\gamma \Rightarrow \alpha\delta\gamma$ ,  
falls  $\beta \rightarrow \delta$  Regel von  $P$  ist

# Noam Chomsky

## Kurz-Bio: Noam Chomsky

- Geboren: 7.12.1928 in Philadelphia
- Studium der Philosophie und Linguistik an der University of Pennsylvania und in Harvard
- Promotion 1955: University of Pennsylvania
- Er lehrt seit 1955 am MIT
- Grundlegende Studien zur Beschreibung natürlicher Sprachen mit formalen Grammatiken

(Quellen: Wikipedia)

# Chomsky-Grammatiken: Beispiel

## Beispiel-Grammatik

$$\begin{aligned} S &\rightarrow SABC \mid \epsilon \\ AB &\rightarrow BA \\ BA &\rightarrow AB \\ AC &\rightarrow CA \\ CA &\rightarrow AC \\ BC &\rightarrow CB \\ CB &\rightarrow BC \\ A &\rightarrow a \\ B &\rightarrow b \\ C &\rightarrow c \end{aligned}$$

## Beispiel-Ableitung

$$\begin{aligned} S &\Rightarrow SABC \\ &\Rightarrow SABCABC \\ &\Rightarrow ABCABC \\ &\Rightarrow BACABC \\ &\Rightarrow BAACBC \\ &\Rightarrow bAACBC \\ &\vdots \\ &\Rightarrow baacbc \end{aligned}$$

- Diese Grammatik erzeugt die Sprache

$$- L_{abc} \stackrel{\text{def}}{=} \{w \mid \#_a(w) = \#_b(w) = \#_c(w)\}$$

aller Strings über  $\{a, b, c\}$ , bei denen die Anzahl der a, b und c gleich ist



# Die Chomsky-Hierarchie

- Die **Chomsky-Hierarchie** umfasst 4 Klassen von Sprachen

| Typ | Name            | Regel-Einschränkung<br>$\alpha \rightarrow \beta$    |
|-----|-----------------|--|
| 0   | Typ 0           | keine  |
| 1   | kontextsensitiv | $ \alpha  \leq  \beta $                              |
| 2   | kontextfrei     | $X \rightarrow \beta$                                |
| 3   | regulär         | $X \rightarrow \sigma$ oder $X \rightarrow \sigma Y$ |

- Bei den Typen 1 und 3 ist jeweils auch die Regel  $S \rightarrow \epsilon$  erlaubt, falls  $S$  auf keiner rechten Seite vorkommt
  - ✎ Bei den Typen 0 und 2 sind  $\epsilon$ -Regeln sowieso erlaubt
- Grammatiken, die nur Regeln der Formen  $X \rightarrow \sigma$  und  $X \rightarrow \sigma Y$  haben, heißen **rechtslinear**
  - Auch die (analog definierten) linkslinearen Grammatiken erzeugen genau die regulären Sprachen

# Inhalt

7.1 Kontextfreie Grammatiken: Beispiele und Definition

7.2 Ableitungen und Ableitungsbäume

7.3 Mehrdeutigkeit

7.4 Konstruktion von Grammatiken

7.5 Die Chomsky-Hierarchie

▷ **7.6 Erweiterte kontextfreie Grammatiken**

# Erweiterte kontextfreie Grammatiken

- Rechte Seiten der kompakten Notation für kontextfreie Grammatiken erinnern an reguläre Ausdrücke:  $\alpha_1 \mid \cdots \mid \alpha_k$  entspricht  $\alpha_1 + \cdots + \alpha_k$
- Warum nicht reguläre Ausdrücke erlauben?

## Definition

- Eine **erweiterte kontextfreie Grammatik**  $G = (V, \Sigma, S, P)$  besteht aus
  - einer Menge  $V$  von **Variablen**
  - einem Alphabet  $\Sigma$
  - einem **Startsymbol**  $S \in V$ ,
  - einer Menge  $P$ , die für jede Variable  $X \in V$  genau eine **Regel**  $X \rightarrow \alpha_X$  enthält, wobei  $\alpha_X$  ein regulärer Ausdruck über  $V \cup \Sigma$  ist
- In einem Ableitungsschritt kann dann immer eine Variable  $X$  durch einen String  $\beta \in L(\alpha_X)$  ersetzt werden
- der Knotengrad in Ableitungsbäumen kann beliebig groß werden

## Beispiel

- Zur Erinnerung: Grammatik für arithmetische Ausdrücke:
$$\begin{aligned} A &\rightarrow A + T \mid T \\ T &\rightarrow T \times F \mid F \\ F &\rightarrow (A) \mid B \\ B &\rightarrow a \mid b \mid Ba \mid Bb \mid B0 \mid B1 \end{aligned}$$
- Erweiterte kontextfreie Grammatik für die selbe Sprache:
$$\begin{aligned} A &\rightarrow T[+T]^* \\ T &\rightarrow F[\times F]^* \\ F &\rightarrow (A) \mid B \\ B &\rightarrow [a \mid b][a \mid b \mid 0 \mid 1]^* \end{aligned}$$
- ✎ Dabei sind  $[$  und  $]$  Meta-Symbole zum Klammern und  $|$  ist ein Meta-Symbol für die Vereinigung (anstelle des üblichen „+“)

# Erweiterte kontextfreie Grammatiken: Ableitungsbaum

Beispiel: Erweiterte Grammatik

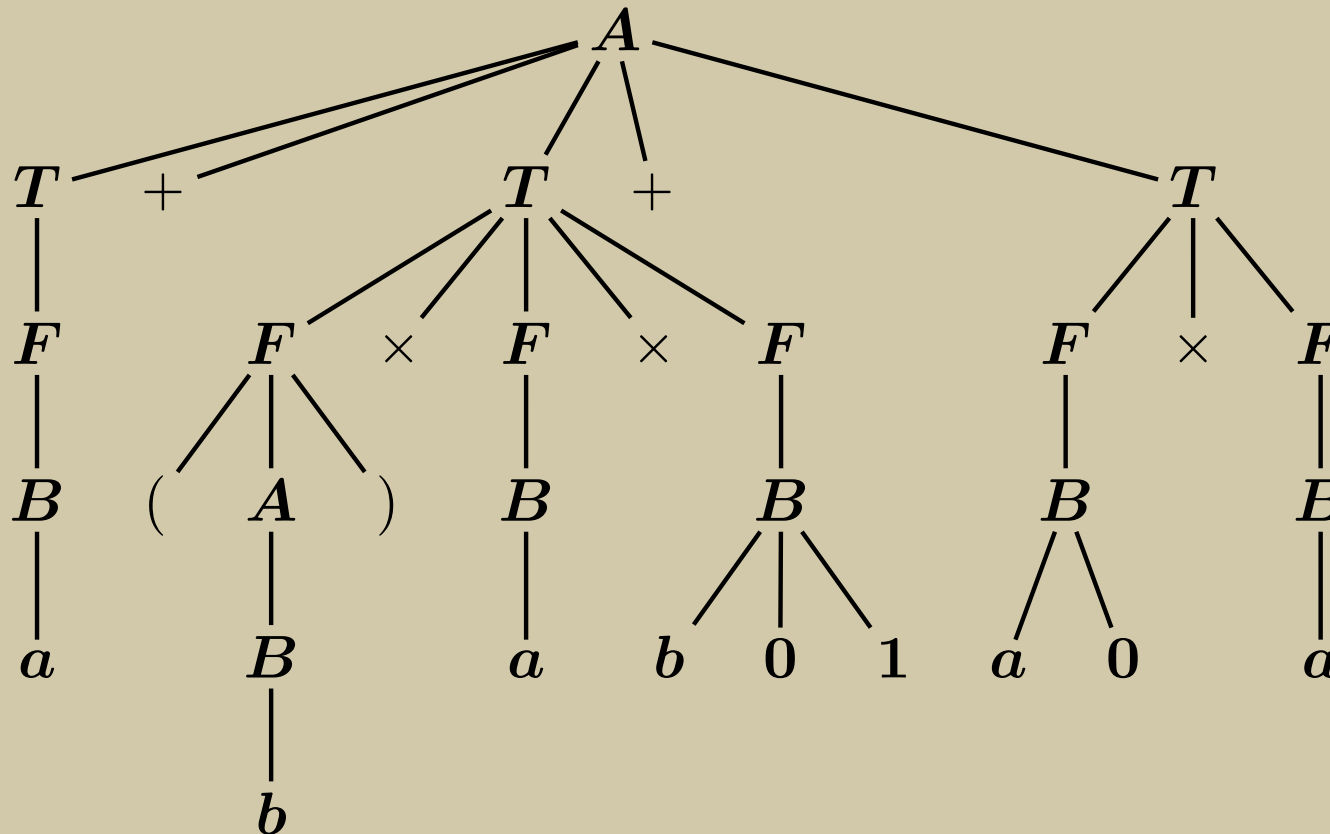
$$A \rightarrow T[+T]^*$$

$$T \rightarrow F[\times F]^*$$

$$F \rightarrow (A) \mid B$$

$$B \rightarrow [a \mid b][a \mid b \mid 0 \mid 1]^*$$

Beispiel: Ableitungsbaum



# Ausdrucksstärke erweiterter kontextfreier Grammatiken

## Satz 7.1

- Sei  $L$  eine Sprache
- Dann sind äquivalent:
  - (a)  $L = L(G)$  für eine kontextfreie Grammatik  $G$
  - (b)  $L = L(G')$  für eine erweiterte kontextfreie Grammatik  $G'$
- Die Beweisidee findet sich im Anhang

# BNF: Backus-Naur Form

- Die **Backus-Naur-Form** ist eine alternative Notation für kontextfreie Grammatiken:

|                         |     |  |
|-------------------------|-----|--|
| <Programm>              | ::= | "PROGRAM" <Bezeichner> "BEGIN" <Satzfolge> "END" . |
| <Bezeichner>            | ::= | <Buchstabe> <Restbezeichner>                       |
| <Restbezeichner>        | ::= | <Buchstabe oder Ziffer> <Restbezeichner>           |
| <Buchstabe oder Ziffer> | ::= | <Buchstabe>   <Ziffer>                             |
| <Buchstabe>             | ::= | A   B   C   D   ...   Z   a   b   ...   z          |
| <Ziffer>                | ::= | 0   1   2   3   4   5   6   7   8   9              |
| <Satzfolge>             | ::= | ...  |
| ...                     |     |  |

(aus: Wikipedia)

- Also:
  - statt  $\rightarrow$  wird  $::=$  verwendet
  - Variablen in Klammern <...>
- Außerdem können optionale Elemente in Klammern [...] gesetzt werden  
(entsprechend (...) in RAs)

# EBNF: Erweiterte Backus-Naur Form

- BNF entspricht kontextfreien Grammatiken
- EBNF entspricht erweiterten kontextfreien Grammatiken
- Zusätzliche Möglichkeiten:
  - Konkatenation: durch Komma angedeutet
  - Wiederholung: durch geschweifte Klammern { und }
  - Terminalzeichen werden in Anführungszeichen gesetzt, deshalb für Variablen keine spitzen Klammern mehr nötig
  - ; als Zeilenendesymbol

- Beispiel:

|             |   |  |
|-------------|---|--|
| Programm    | = | "PROGRAM" Bezeichner "BEGIN" { Zuweisung ["," ] } "END" "." ;  |
| Bezeichner  | = | Buchstabe { ( Buchstabe   Ziffer ) } ;   |
| Zahl        | = | [ "-" ] Ziffer { Ziffer } ;  |
| String      | = | "" { AlleZeichen - "" } "" ;   |
| Zuweisung   | = | Bezeichner ":( Zahl   Bezeichner   String ) ;  |
| Buchstabe   | = | "A"   "B"   "C"   "D"   "E"   "F"   "G"   "H"   "I"   "J"   "K"   "L"   "M"   "N"<br>  "O"   "P"   "Q"   "R"   "S"   "T"   "U"   "V"   "W"   "X"   "Y"   "Z" ; |
| Ziffer      | = | "0"   "1"   "2"   "3"   "4"   "5"   "6"   "7"   "8"   "9" ;  |
| AlleZeichen | = | ? alle sichtbaren Zeichen ? ;  |

(aus: Wikipedia)

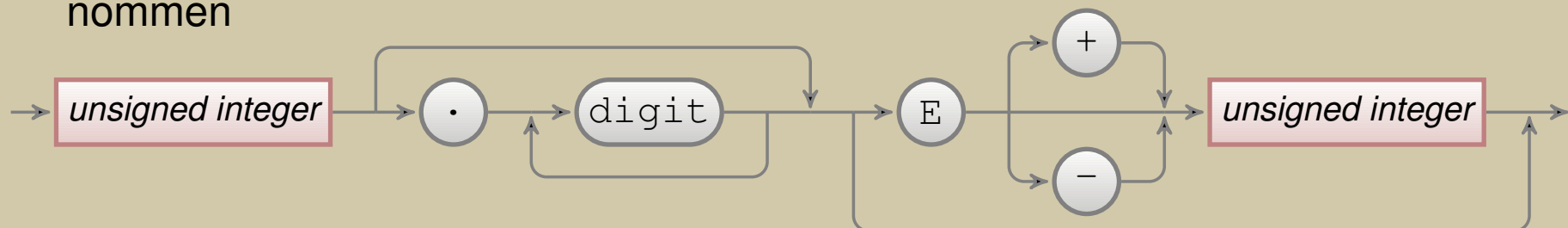
# Syntaxdiagramme

- **Syntaxdiagramme** sind eine weitere, sehr intuitive Notation für erweiterte kontextfreie Grammatiken
- Grammatiken lassen sich wie folgt übersetzen:

| Regel                    | Diagramm |
|--------------------------|----------|
| $A \rightarrow BC$       |          |
| $A \rightarrow B \mid C$ |          |
| $A \rightarrow B^*$      |          |

## Beispiel

- Das folgende Beispiel eines Syntaxdiagramms ist dem TikZ/PGF-Handbuch entnommen





# Zusammenfassung

- Mit Hilfe kontextfreier Grammatiken lassen sich einige nicht reguläre Sprachen wie die Menge aller Palindrome und die Menge (gewisser) arithmetischer Ausdrücke beschreiben
- Ableitungen kontextfreier Grammatiken lassen sich anschaulich durch Ableitungsbäume darstellen
- Für viele Zwecke ist es wünschenswert, dass jeder String der Sprache einen eindeutigen Ableitungsbaum hat, das ist jedoch nicht immer möglich
- Erweiterte kontextfreie Grammatiken sind genauso ausdrucksstark wie kontextfreie Grammatiken
- Syntaxdiagramme und EBNF bieten eine alternative Syntax
- Kontextfreie Grammatiken sind eine eingeschränkte Form von Chomsky-Grammatiken

# Beweisidee für Satz 7.1

## Beweisidee

- „ $(a) \Rightarrow (b)$ “: ✓
- „ $(b) \Rightarrow (a)$ “:
  - Sei  $G'$  eine erweiterte kontextfreie Grammatik für  $L$  mit Startsymbol  $S$
  - Sei  $X \rightarrow \alpha_X$  eine Regel von  $G'$
  - ➡  $\alpha_X$  ist ein regulärer Ausdruck
  - ➡  $L(\alpha_X)$  ist eine reguläre Sprache über  $V \cup \Sigma$
  - ➡  $L(\alpha_X)$  ist kontextfrei ☞ Kapitel 13
  - Sei, für jedes  $X$ ,  $G_X$  eine Grammatik für  $L(\alpha_X)$  mit Startsymbol  $X$
  - Sei  $G$  die Grammatik, die durch Vereinigung der Grammatiken  $G_X$  entsteht, mit Startsymbol  $S$
  - **Behauptung:**  $L(G) = L(G')$
  - Bemerkungen:
    - ✎ Die Grammatiken  $G_X$  haben  $V \cup \Sigma$  als Terminalalphabet und Regeln der Form  $X \rightarrow \sigma$  oder  $X \rightarrow \sigma Y$  ☞ rechtslinear
- \* Ihre Variablenmengen müssen disjunkt gewählt werden

# Grundbegriffe der Theoretischen Informatik

Sommersemester 2016 - Thomas Schwentick

Teil B: Kontextfreie Sprachen

8: Normalformen und Korrektheitsbeweise

Version von: 17. Mai 2016 (12:21)


# Normalformen: Einleitung

## Erinnerung an die Logik-Vorlesung

- Für aussagenlogische Formeln gibt es verschiedene Normalformen:
  - konjunktive Normalform
  - disjunktive Normalform
  - Negationsnormalform
- Zu jeder aussagenlogischen Formel gibt es also eine äquivalente Formel mit einer gewissen syntaktischen Struktur
- Das ist für viele Zwecke nützlich:
  - Um den Resolutionskalkül anzuwenden, muss die Formel in KNF vorliegen
  - Um den Tableauekalkül anzuwenden, muss die Formel in NNF vorliegen

- Jetzt betrachten wir Normalformresultate für kontextfreie Grammatiken
- Sie besagen, dass es zu jeder kontextfreien Sprache eine Grammatik einer eingeschränkten Form gibt

- **Chomsky-Normalform:** Nur Regeln der Art
  - $X \rightarrow YZ$  und
  - $X \rightarrow \sigma$
- **Greibach-Normalform:** Nur Regeln der Art
  - $X \rightarrow \sigma X_1 \cdots X_k, k \geq 0$

 In beiden Fällen wird bei Bedarf zusätzlich eine Regel  $S \rightarrow \epsilon$  hinzu genommen

- Nutzen dieser Normalformen:
  - Die automatische Verarbeitung von Grammatiken wird erleichtert
    - weniger mögliche Fälle zu implementieren
  - Beweise werden übersichtlicher
    - weniger mögliche Fälle zu untersuchen

## ▷ 8.1 Chomsky-Normalform

8.2 Greibach-Normalform


8.3 Korrektheitsbeweise für kontextfreie Grammatiken

8.4 Anhang: Beweisdetails

# Chomsky-Normalform: Definition

## Definition

- Eine Variable  $X$  einer Grammatik  $G = (V, \Sigma, S, P)$  heißt ⊕
  - **nützlich**, falls es eine Ableitung  $S \Rightarrow_G^* \alpha X \beta \Rightarrow_G^* w$  mit  $w \in \Sigma^*$  gibt

 Eine Variable ist also nützlich, wenn sie in mindestens einer Ableitung eines Wortes vorkommt

## Definition

- $G = (V, \Sigma, S, P)$  ist in **Chomsky-Normalform (CNF)** wenn
  - $G$  nur nützliche Variablen enthält und
  - alle Regeln von  $G$  die Form
    - \*  $X \rightarrow YZ$  oder
    - \*  $X \rightarrow \sigma$haben mit  $X, Y, Z \in V, \sigma \in \Sigma$
- Falls  $S$  in keiner rechten Regelseite vorkommt, ist zusätzlich die Regel  $S \rightarrow \epsilon$  erlaubt

• Wir werden jetzt beweisen, dass es zu jeder kontextfreien Grammatik eine äquivalente Grammatik in CNF gibt

• Der Beweis liefert auch einen Algorithmus für die Umwandlung in CNF

# Chomsky-Normalform: Ausblick

- Grammatiken werden durch die Umwandlung in CNF meist größer, aber homogener, wie sich an der Beispiel-Grammatik zeigen wird

## Beispiel-Grammatik: $G_0$

$$\begin{aligned} S &\rightarrow bDD \mid Ca \mid bc \\ A &\rightarrow B \mid aCC \mid baD \\ B &\rightarrow cBD \mid \epsilon \mid AC \\ C &\rightarrow bD \mid aBA \\ D &\rightarrow CD \mid a \mid EF \\ E &\rightarrow Eb \\ F &\rightarrow a \end{aligned}$$

## Beispiel-Grammatik in CNF

$$\begin{aligned} S &\rightarrow W_b S_1 \mid C W_a \mid W_b W_c \\ S_1 &\rightarrow DD \\ A &\rightarrow W_a A_1 \mid W_b A_2 \mid W_c B_1 \mid AC \mid \\ &\quad W_b D \mid W_a C_1 \mid a \\ A_1 &\rightarrow CC \\ A_2 &\rightarrow W_a D \\ B &\rightarrow W_c B_1 \mid AC \mid W_b D \mid W_a C_1 \mid a \\ B_1 &\rightarrow BD \mid CD \mid a \\ C &\rightarrow W_b D \mid W_a C_1 \mid a \\ C_1 &\rightarrow BA \mid W_a A_1 \mid W_b A_2 \mid W_c B_1 \mid \\ &\quad AC \mid W_b D \mid W_a C_1 \mid a \\ D &\rightarrow CD \mid a \\ W_a &\rightarrow a \\ W_b &\rightarrow b \\ W_c &\rightarrow c \end{aligned}$$

# Chomsky-Normalform: Satz

## Satz 8.1 [Chomsky 59]

- Für jede kontextfreie Sprache  $L$  gibt es eine Grammatik  $G$  in Chomsky-Normalform mit  $L(G) = L$

### Beweisskizze

- Sei  $G_0 = (V, \Sigma, S, P)$  eine Grammatik für  $L$
- Wir entfernen in  $G_0$  nach und nach die Merkmale, die der CNF im Weg stehen und konstruieren dafür Grammatiken mit folgenden Eigenschaften:

( $G_1$ ) nur nützliche Variablen

( $G_2$ ) rechte Seiten enthalten genau ein Terminalsymbol oder nur Variablen

( $G_3$ ) ohne Regeln  $X \rightarrow \beta$  mit  $|\beta| > 2$

( $G_4$ ) ohne  $\epsilon$ -Regeln

( $G_5$ ) ohne Regeln der Art  $X \rightarrow Y$

- Es gilt dann  $L(G_5) = L - \{\epsilon\}$ , deshalb muss  $G_5$  evtl. um  $S \rightarrow \epsilon$  ergänzt werden



# CNF: (1) Entfernen von nutzlosen Variablen (1/4)

## Definition

- Eine Variable  $X$  einer Grammatik  $G = (V, \Sigma, S, P)$  heißt ⊕
  - **erreichbar**, falls es eine Ableitung  $S \Rightarrow_G^* \alpha X \beta$  gibt
  - **erzeugend**, falls es eine Ableitung  $X \Rightarrow_G^* w$  mit  $w \in \Sigma^*$  gibt

- Klar: nützlich  $\Rightarrow$  erreichbar und erzeugend


## Beispiel

- In der Grammatik
$$\begin{array}{l} S \rightarrow AB \mid a \\ A \rightarrow b \\ C \rightarrow ab \end{array}$$
ist
  - $S$  nützlich
  - $C$  erzeugend, aber nicht erreichbar
  - $B$  erreichbar, aber nicht erzeugend
  - $A$  erreichbar und erzeugend, aber nicht nützlich

- Wie die Variable  $A$  im Beispiel zeigt, ist eine erreichbare und erzeugende Variable nicht immer nützlich
- Es genügt aber, zuerst die nicht erzeugenden und dann die (dann) nicht erreichbaren Variablen zu entfernen

## Algorithmus CNF1


- 1:  $G'_1 := G_0$  ohne die nicht erzeugenden Variablen
- 2:  $G_1 := G'_1$  ohne die nicht erreichbaren Variablen

 Dabei werden jeweils außer den Variablen auch alle Regeln entfernt, in denen sie (links oder rechts) vorkommen

## CNF: (1) Entfernen von nutzlosen Variablen (2/4)

- Berechnung der Menge  $V_e$  der **erzeugenden Variablen**:

- Initialisiere  $V_e$  durch die Menge aller Variablen  $X$ , für die es eine Regel gibt, deren rechte Seite nur aus Terminalsymbolen besteht

 Also eine Regel  $X \rightarrow \alpha$  mit  $\alpha \in \Sigma^*$

- Solange möglich, füge Variablen  $X$  zu  $V_e$  hinzu, wenn sie eine Regel haben, die auf der rechten Seite nur Terminalsymbole und Variablen aus  $V_e$  enthält

- Variablen, die am Ende nicht in  $V_e$  sind, sind nicht erzeugend

### Beispiel-Grammatik: $G_0$

$$\begin{aligned} S &\rightarrow bDD \mid Ca \mid bc \\ A &\rightarrow B \mid aCC \mid baD \\ B &\rightarrow cBD \mid \epsilon \mid AC \\ C &\rightarrow bD \mid aBA \\ D &\rightarrow CD \mid a \mid EF \\ E &\rightarrow Eb \\ F &\rightarrow a \end{aligned}$$

- $E$  ist nicht erzeugend und wird daher nicht in  $G'_1$  aufgenommen

### $G'_1$

$$\begin{aligned} S &\rightarrow bDD \mid Ca \mid bc \\ A &\rightarrow B \mid aCC \mid baD \\ B &\rightarrow cBD \mid \epsilon \mid AC \\ C &\rightarrow bD \mid aBA \\ D &\rightarrow CD \mid a \\ F &\rightarrow a \end{aligned}$$

## CNF: (1) Entfernen von nutzlosen Variablen (3/4)

- Berechnung der **nicht erreichbaren Variablen**:
- Konstruiere einen Graphen  $H(G'_1)$  zu  $G'_1$ :
  - Knotenmenge: Variablen von  $G'_1$
  - Kante von  $X$  nach  $Y$ , wenn  $Y$  auf der rechten Seite einer Regel zu  $X$  vorkommt
- Berechne alle in diesem Graphen von  $S$  aus erreichbaren Knoten
- Die übrigen Knoten sind nicht erreichbar

$G'_1$

$$\begin{aligned} S &\rightarrow bDD \mid Ca \mid bc \\ A &\rightarrow B \mid aCC \mid baD \\ B &\rightarrow cBD \mid \epsilon \mid AC \\ C &\rightarrow bD \mid aBA \\ D &\rightarrow CD \mid a \\ F &\rightarrow a \end{aligned}$$

- $F$  ist in  $G'_1$  nicht erreichbar

$G_1$

$$\begin{aligned} S &\rightarrow bDD \mid Ca \mid bc \\ A &\rightarrow B \mid aCC \mid baD \\ B &\rightarrow cBD \mid \epsilon \mid AC \\ C &\rightarrow bD \mid aBA \\ D &\rightarrow CD \mid a \end{aligned}$$

# CNF: (1) Entfernen von nutzlosen Variablen (4/4)

## Lemma 8.2

- Sei  $G_1$  durch Algorithmus CNF1 berechnet
- Dann gelten:
  - (a)  $L(G_1) = L(G_0)$
  - (b)  $G_1$  enthält nur nützliche Variablen

## Beweisidee

- (a) –  $L(G_1) \subseteq L(G_0)$ :
  - \* weil alle Regeln aus  $G_1$  auch in  $G_0$  vorkommen
- $L(G_0) \subseteq L(G_1)$ :
  - \* weil alle in einer Ableitung  $S \Rightarrow_{G_0}^* w$  vorkommenden Variablen und Regeln bei der Umwandlung in  $G_1$  erhalten bleiben
- (b) – Sei  $X$  eine Variable von  $G_1$ 
  - ➡  $X$  ist erreichbar in  $G'_1$
  - ➡  $S \Rightarrow_{G'_1}^* \alpha X \beta$  für gewisse  $\alpha, \beta$ 
    - Wegen Schritt (1) des Algorithmus sind  $X$  und alle Symbole aus  $\alpha, \beta$  erzeugend in  $G'_1$   
(und damit auch in  $G_1$ )
  - ➡  $X$  ist nützlich

## CNF: (2) Variablen und Terminalsymbole trennen

### Algorithmus CNF2

- 1: **for** jedes Terminalsymbol  $\sigma \in \Sigma$  **do**
- 2:   Füge eine neue Variable  $W_\sigma$  hinzu
- 3:   Ersetze in allen rechten Regelseiten  $\sigma$  durch  $W_\sigma$
- 4:   Füge eine neue Regel  $W_\sigma \rightarrow \sigma$  hinzu

### Lemma 8.3

- Sei  $G_2$  durch Algorithmus CNF2 aus  $G_1$  berechnet
- Dann gelten:
  - (a)  $L(G_2) = L(G_1)$
  - (b) Jede rechte Regelseite von  $G_2$  ist von einem der folgenden drei Typen:
    - ein Terminalsymbol
    - $\epsilon$
    - ein oder mehrere Variablen

## CNF: (2) Variablen und Terminalsymbole trennen: Beispiel

$G_1$

$$\begin{aligned} S &\rightarrow bDD \mid Ca \mid bc \\ A &\rightarrow B \mid aCC \mid baD \\ B &\rightarrow cBD \mid \epsilon \mid AC \\ C &\rightarrow bD \mid aBA \\ D &\rightarrow CD \mid a \end{aligned}$$

$G_2$

$$\begin{aligned} S &\rightarrow W_bDD \mid CW_a \mid W_bW_c \\ A &\rightarrow B \mid W_aCC \mid W_bW_aD \\ B &\rightarrow W_cBD \mid \epsilon \mid AC \\ C &\rightarrow W_bD \mid W_aBA \\ D &\rightarrow CD \mid W_a \\ W_a &\rightarrow a \\ W_b &\rightarrow b \\ W_c &\rightarrow c \end{aligned}$$

## CNF: (3) Rechte Seiten verkürzen

### Algorithmus CNF3

1: Entferne jede Regel der Art

$$X \rightarrow Y_1 \cdots Y_k, \text{ mit } k > 2$$

2: Füge dafür folgende Regeln hinzu:

$$X \rightarrow Y_1 Z_1$$

$$Z_1 \rightarrow Y_2 Z_2$$

$$\vdots \rightarrow \vdots$$

$$Z_{k-2} \rightarrow Y_{k-1} Y_k$$

- Dabei sind die  $Z_i$  neue (und für jede ersetzte Regel andere) Variablen, für alle  $i \in \{1, \dots, k-2\}$

### Lemma 8.4

- Sei  $G_3$  durch Algorithmus CNF3 aus  $G_2$  berechnet
- Dann gelten:
  - (a)  $L(G_3) = L(G_2)$
  - (b) Jede rechte Regelseite von  $G_3$  ist von einem der folgenden Typen:
    - ein Terminalsymbol
    - $\epsilon$
    - eine Variable
    - zwei Variablen

## CNF: (3) Rechte Seiten verkürzen: Beispiel

$G_2$

$$S \rightarrow W_b D D \mid C W_a \mid W_b W_c$$

$$A \rightarrow B \mid W_a C C \mid W_b W_a D$$

$$B \rightarrow W_c B D \mid \epsilon \mid A C$$

$$C \rightarrow W_b D \mid W_a B A$$

$$D \rightarrow C D \mid W_a$$

$$W_a \rightarrow a$$

$$W_b \rightarrow b$$

$$W_c \rightarrow c$$

$G_3$

$$S \rightarrow W_b S_1 \mid C W_a \mid W_b W_c$$

$$S_1 \rightarrow D D$$

$$A \rightarrow B \mid W_a A_1 \mid W_b A_2$$

$$A_1 \rightarrow C C$$

$$A_2 \rightarrow W_a D$$

$$B \rightarrow W_c B_1 \mid \epsilon \mid A C$$

$$B_1 \rightarrow B D$$

$$C \rightarrow W_b D \mid W_a C_1$$

$$C_1 \rightarrow B A$$

$$D \rightarrow C D \mid W_a$$

$$W_a \rightarrow a$$

$$W_b \rightarrow b$$

$$W_c \rightarrow c$$



## CNF: (4) $\epsilon$ -Regeln entfernen (1/3)

### Beispiel

- Was ist zu beachten, wenn eine Grammatik (unter anderem) die Regeln

$$A \rightarrow BC$$

$$B \rightarrow EF$$

$$C \rightarrow DE$$

$$D \rightarrow E$$

$$E \rightarrow \epsilon$$

enthält und wir die  $\epsilon$ -Regel  $E \rightarrow \epsilon$  entfernen wollen?

- Um weiterhin den Effekt der (Teil-)Ableitung  $B \Rightarrow EF \Rightarrow F$  zu ermöglichen, sollte die Regel  $B \rightarrow F$  hinzugefügt werden
- Das Entfernen von  $E \rightarrow \epsilon$  bewirkt aber auch, dass  $\epsilon$  nicht mehr von  $D$  und  $C$  abgeleitet werden kann
- Deshalb müssen auch für Vorkommen dieser Variablen auf rechten Regelseiten Ergänzungen vorgenommen werden:  
 $\rightarrow$  z.B. neue Regel:  $A \rightarrow B$

- Der folgende Algorithmus CNF4 berechnet deshalb zunächst die Menge  $V'$  aller Variablen, von denen der Leerstring abgeleitet werden kann
- Für jedes Vorkommen dieser Variablen in rechten Regelseiten fügt er dann neue Regeln ein
- Alle  $\epsilon$ -Regeln werden dann entfernt
- Es ist klar, dass der Leerstring danach nicht mehr erzeugt werden kann

## CNF: (4) $\epsilon$ -Regeln entfernen (2/3)

### Algorithmus CNF4

- 1:  $V' := \{X \mid X \Rightarrow^* \epsilon\}$
- 2: **for**  $X \rightarrow YZ$  in  $P$  **do**
- 3:   **if**  $Z \in V'$  **then**
- 4:     Füge Regel  $X \rightarrow Y$  hinzu
- 5:   **if**  $Y \in V'$  **then**
- 6:     Füge Regel  $X \rightarrow Z$  hinzu
- 7: Entferne alle Regeln der Art  $X \rightarrow \epsilon$
- 8: Entferne nutzlos gewordene Variablen mit CNF1

### Lemma 8.5

- Sei  $G_4$  durch Algorithmus CNF4 aus  $G_3$  berechnet
- Dann gelten:
  - (a)  $L(G_4) = L(G_3) - \{\epsilon\}$
  - (b) Jede rechte Regelseite von  $G_4$  ist von einem der folgenden 3 Typen:
    - ein Terminalsymbol
    - eine Variable
    - zwei Variablen

## CNF: (4) $\epsilon$ -Regeln entfernen: Beispiel

$G_3$

$$\begin{aligned} S &\rightarrow W_b S_1 \mid C W_a \mid W_b W_c \\ S_1 &\rightarrow D D \\ A &\rightarrow B \mid W_a A_1 \mid W_b A_2 \\ A_1 &\rightarrow C C \\ A_2 &\rightarrow W_a D \\ B &\rightarrow W_c B_1 \mid \epsilon \mid A C \\ B_1 &\rightarrow B D \\ C &\rightarrow W_b D \mid W_a C_1 \\ C_1 &\rightarrow B A \\ D &\rightarrow C D \mid W_a \\ W_a &\rightarrow a \\ W_b &\rightarrow b \\ W_c &\rightarrow c \end{aligned}$$

- $V' = \{A, B, C_1\}$

$G_4$

$$\begin{aligned} S &\rightarrow W_b S_1 \mid C W_a \mid W_b W_c \\ S_1 &\rightarrow D D \\ A &\rightarrow B \mid W_a A_1 \mid W_b A_2 \\ A_1 &\rightarrow C C \\ A_2 &\rightarrow W_a D \\ B &\rightarrow W_c B_1 \mid A C \mid C \\ B_1 &\rightarrow B D \mid D \\ C &\rightarrow W_b D \mid W_a C_1 \mid W_a \\ C_1 &\rightarrow B A \mid A \mid B \\ D &\rightarrow C D \mid W_a \\ W_a &\rightarrow a \\ W_b &\rightarrow b \\ W_c &\rightarrow c \end{aligned}$$

## CNF: (4) $\epsilon$ -Regeln entfernen (3/3)

### Algorithmus CNF4

- 1:  $V' := \{X \mid X \Rightarrow^* \epsilon\}$
- 2: **for**  $X \rightarrow YZ$  in  $P$  **do**
- 3:   **if**  $Z \in V'$  **then**
- 4:     Füge Regel  $X \rightarrow Y$  hinzu
- 5:   **if**  $Y \in V'$  **then**
- 6:     Füge Regel  $X \rightarrow Z$  hinzu
- 7: Entferne alle Regeln der Art  $X \rightarrow \epsilon$
- 8: Entferne nutzlos gewordene Variablen (CNF1)

### Beweisidee für Lemma 8.5

- Da der Algorithmus sowohl Regeln hinzufügt als auch Regeln entfernt, ist der Nachweis, dass die neue Grammatik äquivalent ist (bis auf  $\epsilon$ ), etwas kompliziert
- Es wird bewiesen, dass für jede Variable  $X$  von  $G_3$  und jeden String  $w \in \Sigma^* - \{\epsilon\}$  äquivalent sind:
  - (1)  $X \Rightarrow_{G_4}^* w$
  - (2)  $X \Rightarrow_{G_3}^* w$
- Für „(1)  $\Rightarrow$  (2)“ lässt sich für jede neue Regel  $X \rightarrow Y$  in  $G_4$  zeigen:
$$X \Rightarrow_{G_3}^* Y$$
- Für „(2)  $\Rightarrow$  (1)“ lässt sich durch Induktion nach  $n$  zeigen:
$$X \Rightarrow_{G_3}^n w \Rightarrow X \Rightarrow_{G_4}^* w$$
- Weitere Details im Anhang

## CNF: (5) Einzel-Variablen der rechten Seite entfernen (1/2)

- Jetzt müssen nur noch die **Einheits-Regeln** entfernt werden, also Regeln der Form  $X \rightarrow Y$

### Beispiel

- Was ist beim Entfernen der Einheits-Regeln aus einer Grammatik, die folgende Regeln enthält, zu beachten?

$$A \rightarrow BC$$

$$B \rightarrow D$$

$$C \rightarrow DF$$

$$D \rightarrow E$$

$$E \rightarrow e$$

- Um den Effekt der Ableitung  $C \Rightarrow DF \Rightarrow EF \Rightarrow eF$  zu erhalten, nehmen wir die Regel  $D \rightarrow e$  auf:

$$C \Rightarrow DF \Rightarrow eF$$

- Um den Effekt der Ableitung

$$A \Rightarrow BC \Rightarrow DC \Rightarrow EC \Rightarrow eC$$

zu erhalten, nehmen wir auch noch  $B \rightarrow e$  auf:

$$A \Rightarrow BC \Rightarrow eC$$

- Der folgende Algorithmus CNF5 berechnet zunächst alle Variablenpaare  $X \neq Y$ , für die  $X \Rightarrow^* Y$  gilt

- Dann fügt er für jedes solche Paar zu jeder binären Regel  $Y \rightarrow \alpha$  eine neue Regel  $X \rightarrow \alpha$  hinzu

## CNF: (5) Einzel-Variablen der rechten Seite entfernen (2/2)


### Algorithmus CNF5

- 1:  $U := \{(X, Y) \mid X \Rightarrow_{G_4}^* Y, X \neq Y\}$
- 2: **for**  $(X, Y)$  in  $U$  und jede Nicht-Einheitsregel  $Y \rightarrow \alpha$  **do**
- 3:   Füge neue Regel  $X \rightarrow \alpha$  ein
- 4: Entferne alle Einheitsregeln
- 5: Entferne alle nicht erreichbaren Variablen (und ihre Regeln)

### Lemma 8.6

- Sei  $G_5$  durch Algorithmus CNF5 aus  $G_4$  berechnet
- Dann gelten:
  - (a)  $L(G_5) = L(G_4)$
  - (b)  $G_5$  ist in Chomsky-Normalform

### Beweisidee

- „ $L(G_5) \subseteq L(G_4)$ “:
  - Für jede neue Regel  $X \rightarrow \alpha$  von  $G_5$  gibt es in  $G_4$  eine Variable  $Y$ , so dass gilt:
    - \*  $X \Rightarrow_{G_4}^* Y$
    - \*  $Y \rightarrow \alpha$  ist Regel in  $G_4$
  - Also gibt es für jede neue Regel  $X \rightarrow \alpha$  von  $G_5$  in  $G_4$  eine Ableitung  $X \Rightarrow_{G_4}^* \alpha$
- „ $L(G_4) \subseteq L(G_5)$ “:
  - Jede Folge  $X \Rightarrow_{G_4} \dots \Rightarrow_{G_4} Y \Rightarrow_{G_4} \alpha$  kann in  $G_5$  durch Anwendung von  $X \rightarrow \alpha$  ersetzt werden
    -  Dabei ist  $\alpha$  keine Variable!
- Weitere Details im Anhang

## CNF: (5) Einzel-Variablen entfernen (Beispiel)

$G_4$

$$\begin{aligned}
 S &\rightarrow W_b S_1 \mid C W_a \mid W_b W_c \\
 S_1 &\rightarrow D D \\
 A &\rightarrow B \mid W_a A_1 \mid W_b A_2 \\
 A_1 &\rightarrow C C \\
 A_2 &\rightarrow W_a D \\
 B &\rightarrow W_c B_1 \mid A C \mid C \\
 B_1 &\rightarrow B D \mid D \\
 C &\rightarrow W_b D \mid W_a C_1 \mid W_a \\
 C_1 &\rightarrow B A \mid A \mid B \\
 D &\rightarrow C D \mid W_a \\
 W_a &\rightarrow a \\
 W_b &\rightarrow b \\
 W_c &\rightarrow c
 \end{aligned}$$

$$U = \{(C_1, A), (A, B), (B, C), \\
 (C, W_a), (B_1, D), (D, W_a), \\
 (C_1, B), (C_1, C), (C_1, W_a), \\
 (A, C), (A, W_a), (B, W_a), (B_1, W_a)\}$$

$G_5$

$$\begin{aligned}
 S &\rightarrow W_b S_1 \mid C W_a \mid W_b W_c \\
 S_1 &\rightarrow D D \\
 A &\rightarrow W_a A_1 \mid W_b A_2 \mid W_c B_1 \mid A C \mid \\
 &\quad W_b D \mid W_a C_1 \mid a \\
 A_1 &\rightarrow C C \\
 A_2 &\rightarrow W_a D \\
 B &\rightarrow W_c B_1 \mid A C \mid W_b D \mid W_a C_1 \mid a \\
 B_1 &\rightarrow B D \mid C D \mid a \\
 C &\rightarrow W_b D \mid W_a C_1 \mid a \\
 C_1 &\rightarrow B A \mid W_a A_1 \mid W_b A_2 \mid W_c B_1 \mid \\
 &\quad A C \mid W_b D \mid W_a C_1 \mid a \\
 D &\rightarrow C D \mid a \\
 W_a &\rightarrow a \\
 W_b &\rightarrow b \\
 W_c &\rightarrow c
 \end{aligned}$$

# Chomsky-Normalform: Abschluss der Beweisskizze

- Falls die Sprache  $L$  den String  $\epsilon$  enthält, ergänzen wir noch einen weiteren Schritt

## Algorithmus CNF6

- 1: **if**  $\epsilon \in L$  und  $S$  kommt in keiner rechten Seite einer Regel vor **then**
- 2:   Füge neue Regel  $S \rightarrow \epsilon$  ein
- 3: **else**
- 4:   Füge ein neues Startsymbol  $S'$  und die Regel  $S' \rightarrow \epsilon$  hinzu
- 5:   Füge für jede Regel  $S \rightarrow \alpha$  die Regel  $S' \rightarrow \alpha$  hinzu

## Satz 8.1 [Chomsky 59]

- Für jede kontextfreie Sprache  $L$  gibt es eine Grammatik  $G$  in Chomsky-Normalform mit  $L(G) = L$

## Beweisskizze

- Sei  $G_0 = (V, \Sigma, S, P)$  eine Grammatik für  $L$
- Für die wie beschrieben konstruierten Grammatiken  $G_1, \dots, G_5$  gilt:

$$\begin{aligned} L(G_5) &= L(G_4) && \text{Lemma 8.6} \\ &= L(G_3) - \{\epsilon\} && \text{Lemma 8.5} \\ &= L(G_2) - \{\epsilon\} && \text{Lemma 8.4} \\ &= L(G_1) - \{\epsilon\} && \text{Lemma 8.3} \\ &= L(G_0) - \{\epsilon\} && \text{Lemma 8.2} \end{aligned}$$

- Falls  $\epsilon \notin L(G_0)$ , gilt also  $L(G_5) = L(G_0)$
- Falls  $\epsilon \in L(G_0)$ , gilt für die in CNF6 konstruierte Grammatik:  $L(G_6) = L(G_0)$



## CNF: Größe

- Wie groß wird die durch den Algorithmus insgesamt konstruierte CNF-Grammatik im Vergleich zur Ausgangsgrammatik?
- Sei  $m = |\Sigma|$  und, für jedes  $i$  jeweils
  - $n_i$  die Anzahl der Variablen von  $G_i$ ,
  - $k_i$  die Anzahl der Produktionen von  $G_i$
  - $l_i$  die maximale Länge einer rechten Seite von  $G_i$
- Dann gilt für die einzelnen Teilschritte:
  1. Alle Parameter können allenfalls kleiner werden
  2.  $n_2 = n_1 + m, k_2 = k_1 + m$
  3.  $n_3 = \mathcal{O}(k_2 l_2), k_3 = \mathcal{O}(l_2 k_2), l_3 = 2$
  4.  $k_4 \leq 3k_3$
  5.  $k_5 = \mathcal{O}(k_4 n_4)$
- Insgesamt also:
  - $k_5 = \mathcal{O}(l_0^2 (m + k_0)^2) = \mathcal{O}(|G_0|^4)$
  - $n_5 = \mathcal{O}(l_0 (k_0 + m))$

# Inhalt

8.1 Chomsky-Normalform

▷ **8.2 Greibach-Normalform**

8.3 Korrektheitsbeweise für kontextfreie Grammatiken

8.4 Anhang: Beweisdetails

# Greibach-Normalform (1/2)

## Definition

- Eine kontextfreie Grammatik  $G = (V, \Sigma, S, P)$  ist in **Greibach-Normalform** (GNF) wenn
  - $V$  nur nützliche Variablen enthält und
  - alle Regeln von  $G$  von der Form
    - \*  $X \rightarrow \sigma Y_1 \cdots Y_k$   
mit  $X, Y_1, \dots, Y_k \in V, \sigma \in \Sigma$   
sind
- Falls  $S$  in keiner rechten Regelseite vorkommt, ist zusätzlich die Regel  $S \rightarrow \epsilon$  erlaubt

## Satz 8.7 [Greibach 65]

- Ist  $L$  kontextfrei, so gibt es eine Grammatik  $G$  in Greibach-Normalform, so dass  $L(G) = L$

## Kurz-Bio: Sheila Greibach

- Geboren: 1939
- Studium am Radcliffe College, Cambridge, Massachusetts
- Professorin an der University of California, Los Angeles
- Arbeitsgebiete: Formale Sprachen, Compilerbau, Theoretische Informatik

(Quellen: Wikipedia)

## Greibach-Normalform (2/2)

- Es ist in der Greibach-Normalform sogar möglich, die Anzahl der Variablen in rechten Regelseiten auf maximal zwei zu beschränken
- Es gibt dann also nur Regeln der Art:
  - $X \rightarrow \sigma$
  - $X \rightarrow \sigma Y$
  - $X \rightarrow \sigma Y Z$

### Beispiel

- Die Grammatik  $G'_{a=b}$ :
$$\begin{aligned} S &\rightarrow aB \mid bA \mid \epsilon \\ S' &\rightarrow aB \mid bA \\ A &\rightarrow aS' \mid bAA \mid a \\ B &\rightarrow bS' \mid aBB \mid b \end{aligned}$$
ist in Greibach-Normalform

### Bemerkungen

- Ist  $G$  in GNF, so haben Strings der Länge  $n$  Ableitungen der Länge  $n$
- Der Beweis von Satz 8.7 ist (noch) aufwändiger als der Beweis von Satz 8.1  
→ Buch von Ingo Wegener  
(siehe auch: [Blum, Koch 99])

# Inhalt

8.1 Chomsky-Normalform

8.2 Greibach-Normalform

▷ **8.3 Korrektheitsbeweise für kontextfreie Grammatiken**

8.4 Anhang: Beweisdetails

# Korrektheitsbeweise für Grammatiken: allgemein

- Sei  $G$  eine Grammatik und  $L$  eine Sprache
- Wie lässt sich beweisen, dass  $L(G) = L$  gilt?
- Wie üblich: Zeige  $L(G) \subseteq L$  und  $L \subseteq L(G)$
- $L(G) \subseteq L$  bedeutet, dass die Grammatik *nur* Strings aus  $L$  erzeugt ☞ Korrektheit
  - Der Korrektheitsbeweis ist meist durch Induktion nach der Ableitungslänge möglich
- $L \subseteq L(G)$  bedeutet, dass die Grammatik *alle* Strings aus  $L$  erzeugt ☞ Vollständigkeit
  - Meistens durch Induktion nach der Wortlänge
  - Der Induktionsschritt verwendet meist eine geeignete Fallunterscheidung
  - Hier ist es oft eine weitere, nicht-induktive Definition von  $L$  nötig
- Hat eine Grammatik mehrere Variablen, so wird meist für jede Variable  $X$  gezeigt, was die Sprache  $L(X)$  der von  $X$  aus ableitbaren Strings ist

- ☞ Faustregel:
- Zum Beweis der Korrektheit wird im Ableitungsbaum „von oben nach unten“ argumentiert
  - Zum Beweis der Vollständigkeit von unten nach oben

- Wir betrachten im Folgenden drei Korrektheitsbeweise:
  - für eine sehr einfache Grammatik: Palindrome
  - für eine etwas trickreichere Grammatik:  $G_{a=b}$
  - für eine ziemlich trickreiche Grammatik:  $G_{\text{diff}}$

# Korrektheitsbeweise: Palindrome (1/2)

- Zur Erinnerung:

$$L_{\text{pali}} = \{w \in \{a, b\}^* \mid w^R = w\}$$

- Mit  $G_{\text{pali}}$  bezeichnen wir die Grammatik

$$P \rightarrow \epsilon \mid a \mid b \mid aPa \mid bPb$$

- Nicht-induktive Definition von Palindromen:

- Ein String  $w$  ist genau dann ein Palindrom, wenn für alle  $i \in \{1, \dots, |w|\}$  gilt:

$$w[i] = w[|w| - i + 1]$$

## Proposition 8.8

- $L(G_{\text{pali}}) = L_{\text{pali}}$

## Beweisskizze

- Korrektheit „ $L(G_{\text{pali}}) \subseteq L_{\text{pali}}$ “:

- Sei  $w \in L(G_{\text{pali}})$

$$\Rightarrow P \Rightarrow^k w \text{ für ein } k \geq 1$$

- Wir zeigen  $w \in L$  durch Induktion nach der Ableitungslänge  $k$

- $k = 1$ :

- \*  $\epsilon, a, b$  sind Palindrome ✓

- $k > 1$ :

- \* Fallunterscheidung nach dem ersten Ableitungsschritt

- \* 1. Fall:  $P \Rightarrow aPa \Rightarrow^{k-1} ava = w$ ,  
für ein  $v \in \{a, b\}^*$

$$\Rightarrow P \Rightarrow^{k-1} v$$

$$\Rightarrow v^R = v$$

Induktion

$$\Rightarrow w^R = av^Ra = ava = w$$

- \* 2. Fall:  $P \Rightarrow bPb \Rightarrow^{k-1} bvb = w$ ,  
für ein  $v \in \{a, b\}^*$

→ analog

# Korrektheitsbeweise: Palindrome (2/2)

## Beweisskizze (Forts.)

- **Vollständigkeit** „ $L_{\text{pali}} \subseteq L(G_{\text{pali}})$ “:

- Sei  $w \in L_{\text{pali}}$  und  $n \stackrel{\text{def}}{=} |w|$
- Wir zeigen  $w \in L(G_{\text{pali}})$  durch Induktion nach  $n$

- $n \in \{0, 1\}$ :

➡  $w \in \{\epsilon, a, b\}$  ist ableitbar ✓

- $n \geq 2$ :

\* 1. Fall:  $w[1] = a$

➡  $w[n] = a$

➡  $w = ava$  für einen String  $v$  der Länge  $n - 2$

• Da  $w^R = w$  gilt auch  $v^R = v$ , denn, für alle  $i \leq n - 2$ :

$$v[i] = w[i + 1]$$

☞ Def von  $v$

$$= w[n - (i + 1) + 1]$$

☞  $w$  Palindrom

$$= v[n - (i + 1)]$$

☞ Def von  $v$

$$= v[(n - 2) - i + 1]$$

• Induktion:  $P \Rightarrow^* v$

➡  $P \Rightarrow aPa \Rightarrow^* ava = w$

\* 2. Fall:  $w[1] = b$  analog



# Korrektheitsbeweise: $L_{a=b}$ (1/2)

- Zur Erinnerung:

$$- L_{a=b} = \{w \in \{a, b\}^* \mid \#_a(w) = \#_b(w)\}$$

- $G_{a=b}$  ist  
 $S \rightarrow aSbS \mid bSaS \mid \epsilon$

## Proposition 8.9

$$\bullet L(G_{a=b}) = L_{a=b}$$

## Beweisskizze: Korrektheit

- Zu zeigen:  $L(G_{a=b}) \subseteq L_{a=b}$
- Sei  $w \in L(G_{a=b})$
- Induktion nach der Ableitungslänge  $k$ :

$$\bullet k = 1: S \Rightarrow \epsilon \checkmark$$

- $k > 1$ : Fallunterscheidung nach dem ersten Ableitungsschritt

$$- 1. \text{ Fall: } S \Rightarrow aSbS \Rightarrow^{k-1} aubv = w$$

$$\begin{aligned} * \text{ Induktion: } \#_a(u) &= \#_b(u) \text{ und} \\ \#_a(v) &= \#_b(v) \end{aligned}$$



$$\begin{aligned} \#_a(w) &= \#_a(u) + \#_a(v) + 1 \\ &= \#_b(u) + \#_b(v) + 1 \\ &= \#_b(w) \end{aligned}$$

$$- 2. \text{ Fall: } S \Rightarrow bSaS \Rightarrow^* buav = w \text{ analog}$$

## Korrektheitsbeweise: $L_{a=b}$ (2/2)

- Sei für jeden String  $w \in \{a, b\}^*$ :  

$$\underline{d(w)} \stackrel{\text{def}}{=} \#_b(w) - \#_a(w)$$
- Also:  $L_{a=b} = \{w \in \{a, b\}^* \mid d(w) = 0\}$

### Beweisskizze: Vollständigkeit


- Zu zeigen:  $L_{a=b} \subseteq L(G_{a=b})$
- Sei  $w \in L_{a=b}$
- Induktion nach  $n \stackrel{\text{def}}{=} |w|$
- $n = 0$ :  $S \Rightarrow \epsilon \checkmark$

### Beweisskizze (Forts.)

- $n > 0$ :
  - 1. Fall:  $w = au$  für ein  $u \in \{a, b\}^*$  mit  $|u| = n - 1$  ⊕
    - Da  $d(w) = 0$  gilt  $d(u) = 1$
    - Sei  $i \geq 1$  die kleinste Zahl mit  $d(u[1, i]) = 1$
    - ➡  $d(u[1, i - 1]) = 0$  und  $u[i] = b$ 
      - Außerdem:  $d(u[i + 1, n - 1]) = d(w) - d(u[1, i]) - 1 = 0$
      - Induktion:  $S \Rightarrow^* u[1, i - 1]$  und  $S \Rightarrow^* u[i + 1, n - 1]$
      - ➡  $S \Rightarrow aSbS \Rightarrow^* au[1, i - 1]bu[i + 1, n - 1] = au = w$
  - Der Fall  $w = bv$  ist analog

## Korrektheitsbeweise: $L_{\text{diff}}$ (1/3)

- Für einen String  $w$  gerader Länge bezeichne  $1^{\text{st}}(w)$  seine erste Hälfte und  $2^{\text{nd}}(w)$  seine zweite Hälfte


 Für Strings ungerader Länge seien beide Funktionen undefiniert

- Sei  $L_{\text{diff}} \stackrel{\text{def}}{=} \{w \in \{a, b\}^* \mid 1^{\text{st}}(w) \neq 2^{\text{nd}}(w)\}$

- Idee für die Konstruktion einer Grammatik:

- Sorge dafür, dass  $a$  in  $1^{\text{st}}(w)$  an derselben Stelle steht wie  $b$  in  $2^{\text{nd}}(w)$

\* Erlaube ansonsten beliebige Zeichen

 Oder umgekehrt mit  $b$  in  $1^{\text{st}}(w)$  und  $a$  in  $2^{\text{nd}}(w)$

- Setze dazu  $w = w_1 w_2$  aus Teilstrings  $w_1 = u_1 a v_1$  und  $w_2 = u_2 b v_2$  zusammen mit  $|u_1| = |v_1|$  und  $|u_2| = |v_2|$

- Setze  $i \stackrel{\text{def}}{=} |u_1|$  und  $j \stackrel{\text{def}}{=} |u_2|$   $\boxplus$

$$\Rightarrow |w| = 2i + 1 + 2j + 1 = 2(i + j + 1)$$

- Das mittlere  $a$  von  $w_1$  ist an Position  $i + 1$  von  $1^{\text{st}}(w)$

- Das mittlere  $b$  von  $w_2$  ist in  $w$  an Position  $2i + 1 + j + 1 = (i + j + 1) + (i + 1)$

$\Rightarrow$  es ist in  $2^{\text{nd}}(w)$  an Position  $i + 1$

- Sei  $G_{\text{diff}}$  also die Grammatik

$$S \rightarrow AB \mid BA$$

$$A \rightarrow aAa \mid bAb \mid aAb \mid bAa \mid a$$

$$B \rightarrow aBa \mid bBb \mid aBb \mid bBa \mid b$$

Satz 8.10

- $L(G_{\text{diff}}) = L_{\text{diff}}$

## Korrektheitsbeweise: $L_{\text{diff}}$ (2/3)

- Zur Illustration betrachten wir eine Ableitung des Strings  $w = \mathbf{abaaabba}$
- Aus  $A$  ist  $\mathbf{abaaa}$  ableitbar (mit  $\mathbf{a}$  in der Mitte)
- Aus  $B$  ist  $\mathbf{bba}$  ableitbar (mit  $\mathbf{b}$  in der Mitte)
- Also ist  $w$  ableitbar durch


$$S \Rightarrow AB \Rightarrow^* \mathbf{abaaa}B \Rightarrow^* \mathbf{abaaabba}$$

- Zu beachten:  $A$  erzeugt *nicht* die erste Hälfte von  $w$

## Korrektheitsbeweise: $L_{\text{diff}}$ (3/3)

### Beweisskizze

- Durch Induktion ist sehr leicht zu zeigen:
  - $L(A) = \{uav \mid |u| = |v|, u, v \in \{a, b\}^*\}$
  - $L(B) = \{ubv \mid |u| = |v|, u, v \in \{a, b\}^*\}$

 Hier bezeichnet  $L(A)$  die Menge der von der Variablen  $A$  ableitbaren Terminalstrings

$$\begin{aligned} \Rightarrow L(G_{\text{diff}}) = L(S) = \\ \{u_1av_1u_2bv_2 \mid |u_1|=|v_1|, |u_2|=|v_2|, \\ u_1, u_2, v_1, v_2 \in \{a, b\}^*\} \cup \\ \{u_1bv_1u_2av_2 \mid |u_1|=|v_1|, |u_2|=|v_2|, \\ u_1, u_2, v_1, v_2 \in \{a, b\}^*\} \end{aligned}$$

- Damit lässt sich  $L(G_{\text{diff}}) = L_{\text{diff}}$  recht einfach zeigen

 Anhang

# Zusammenfassung

- Jede kontextfreie Sprache hat eine Grammatik in Chomsky-Normalform, also nur mit Regeln der Art  $X \rightarrow YZ$  und  $X \rightarrow a$
- Jede kontextfreie Sprache hat eine Grammatik in Greibach-Normalform, also nur mit Regeln der Art  $X \rightarrow aY_1 \cdots Y_k$
- Außerdem kann jeweils noch eine Regel  $S \rightarrow \epsilon$  hinzukommen
  - dann darf  $S$  aber auf keiner rechten Seite vorkommen

# Literatur für dieses Kapitel

- **Chomsky-Normalform:**

- Noam Chomsky. On certain formal properties of grammars. *Information and Control*, 2(2):137–167, 1959

- **Greibach-Normalform:**

- Sheila A. Greibach. A new normal-form theorem for context-free phrase structure grammars. *J. ACM*, 12(1):42–52, 1965
- Norbert Blum and Robert Koch. Greibach normal form transformation revisited. *Inf. Comput.*, 150(1):112–118, 1999

# Inhalt

8.1 Chomsky-Normalform

8.2 Greibach-Normalform

8.3 Korrektheitsbeweise für kontextfreie Grammatiken

▷ **8.4 Anhang: Beweisdetails**



## CNF: (4) $\epsilon$ -Regeln entfernen

### Beweisskizze (Forts.)

- „ $X \Rightarrow_{G_4}^* w \Rightarrow X \Rightarrow_{G_3}^* w$ “:  
Wir zeigen:  

$$X \rightarrow_{G_4} Y \Rightarrow X \Rightarrow_{G_3}^* Y$$

☞ für jede neue Regel  $X \rightarrow Y$

  - Denn dann gibt es zu jeder Ableitung  $S \Rightarrow_{G_4}^* w$  eine Ableitung  $S \Rightarrow_{G_3}^* w$
- Sei also  $X \rightarrow Y$  neu in  $G_4$ 
  - 1. Fall:** In  $G_3$  gibt es eine Regel  $X \rightarrow YZ$  und  $Z \Rightarrow_{G_3}^* \epsilon$ 
    - Dann gilt:  

$$X \Rightarrow_{G_3} YZ \Rightarrow_{G_3}^* Y$$
  - 2. Fall:** In  $G_3$  gibt es eine Regel  $X \rightarrow ZY$  und  $Z \Rightarrow_{G_3}^* \epsilon$ 
    - Dann gilt:  

$$X \Rightarrow_{G_3} ZY \Rightarrow_{G_3}^* Y$$

### Beweisskizze (Forts.)

- „ $X \Rightarrow_{G_3}^* w \Rightarrow X \Rightarrow_{G_4}^* w$ “:
- Wir zeigen durch Induktion nach  $n$ :
  - $X \Rightarrow_{G_3}^n w \Rightarrow X \Rightarrow_{G_4}^* w$
- Es gelte also  $X \Rightarrow_{G_3}^n w$
- $n = 1$ :  $w = \sigma$  und  $X \rightarrow \sigma$  ist Regel von  $G_3$ , also auch von  $G_4$
- $n > 1$ :
  - **1. Fall:**  

$$X \Rightarrow_{G_3} YZ \Rightarrow_{G_3}^{n-1} w_1 w_2 = w$$

mit  $Y \Rightarrow_{G_3}^i w_1$ ,  $Z \Rightarrow_{G_3}^j w_2$  und  $i + j = n - 1$

    - \* Falls  $w_1 \neq \epsilon \neq w_2$  folgt die Behauptung per Induktion
    - \* Falls  $w_1 = \epsilon$ ,  $w_2 \neq \epsilon$ , so ist  $Y \in V'$   
 $\Rightarrow X \Rightarrow_{G_4} Z \Rightarrow_{G_4}^* w_2 = w$
    - \* (Analog:  $w_1 \neq \epsilon$ ,  $w_2 = \epsilon$ )
  - **2. Fall:**  $X \Rightarrow_{G_3} Y \Rightarrow_{G_3}^* w$ 
    - \* Induktion liefert:  $X \Rightarrow_{G_4} Y \Rightarrow_{G_4}^* w$

## CNF: (5) Einzel-Variablen der rechten Seite entfernen (3/3)

### Beweisskizze für Lemma 8.6

(b) klar

(a) „ $L(G_5) \subseteq L(G_4)$ “:

- Sei  $X \rightarrow \alpha$  eine gegenüber  $G_4$  neue Regel von  $G_5$
- ➔ In  $G_4$  gibt es dann eine Regel  $Y \rightarrow \alpha$ , für ein  $Y$ , und es gilt  $X \Rightarrow_{G_4}^* Y$
- ➔  $X \Rightarrow_{G_4}^* \alpha$
- ➔ Zu jeder Ableitung  $S \Rightarrow_{G_5}^* w$  gibt es also eine Ableitung  $S \Rightarrow_{G_4}^* w$

### Beweisskizze (Forts.)


- „ $L(G_4) \subseteq L(G_5)$ “:
  - Sei  $w \in L(G_4)$
  - Dann gibt es eine Linksableitung für  $w$  in  $G_4$
  - Wird in einer Linksableitung eine Einheitsregel  $X \rightarrow Y$  verwendet, so muss  $Y$  im nächsten Schritt wieder ersetzt werden
  - Nach endlich vielen Schritten muss dann zum ersten Mal eine Regel  $Z \rightarrow \alpha$  verwendet werden, die keine Einheitsregel ist
  - ➔  $X \Rightarrow_{G_4}^* Z$
  - ➔ Nach Konstruktion enthält  $G_5$  dann die Regel  $X \rightarrow \alpha$
  - ➔ Zu jeder  $G_4$ -Ableitung gibt es eine äquivalente  $G_5$ -Ableitung

# Details des Korrektheitsbeweises für $L_{\text{diff}}$

Beweisskizze: „ $L(G_{\text{diff}}) \subseteq L_{\text{diff}}$ “

- Sei  $w \in L(G_{\text{diff}})$
- 1. Fall:  $w = u_1 a v_1 u_2 b v_2$  für gewisse  $u_1, u_2, v_1, v_2 \in \{a, b\}^*$  mit  $|u_1| = |v_1|$  und  $|u_2| = |v_2|$ 
  - Sei  $i \stackrel{\text{def}}{=} |u_1|$  und  $j \stackrel{\text{def}}{=} |u_2|$
  - ➔  $|w| = 2i + 2j + 2$  und
    - \*  $w[i + 1] = a$
    - \*  $w[2i + j + 2] = b$
  - Wie schon berechnet ist dann
    - \*  $1^{\text{st}}(w) = a$
    - \*  $2^{\text{nd}}(w) = b$
  - ➔  $w \in L_{\text{diff}}$
- 2. Fall:  $w = u_1 b v_1 u_2 a v_2 \dots$ 
  - analog

Beweisskizze: „ $L_{\text{diff}} \subseteq L(G_{\text{diff}})$ “

- Sei  $w \in L_{\text{diff}}$  und  $n \stackrel{\text{def}}{=} |1^{\text{st}}(w)|$    $|w| = 2n$
- OBdA: gibt es ein  $i \in \{0, \dots, n - 1\}$ , so dass  $1^{\text{st}}(w)[i + 1] = a$  und  $2^{\text{nd}}(w)[i + 1] = b$
- Wir setzen:
  - $u_1 \stackrel{\text{def}}{=} w[1, i]$
  - $v_1 \stackrel{\text{def}}{=} w[i + 2, 2i + 1]$
  - $u_2 \stackrel{\text{def}}{=} w[2i + 2, n + i]$
  - $v_2 \stackrel{\text{def}}{=} w[n + i + 2, 2n]$
- ➔  $w = u_1 a v_1 u_2 b v_2$  und
  - $|u_1| = i = |v_1|$  und  $|u_2| = n + i - (2i + 2) + 1 = n - i - 1 = 2n - (n + i + 2) + 1 = |v_2|$
- ➔  $w \in L(G_{\text{diff}})$

# Grundbegriffe der Theoretischen Informatik

Sommersemester 2016 - Thomas Schwentick

Teil B: Kontextfreie Sprachen

9: Kellerautomaten

Version von: 2. Juni 2016 (14:10)

# Inhalt

## ▷ 9.1 Kellerautomaten: Definitionen

9.2 Leerer Keller vs. akzeptierende Zustände

9.3 Grammatiken vs. Kellerautomaten

9.4 Kellerautomaten: Korrektheitsbeweise

9.5 Anhang: Beweisdetails

## Ein Beispiel: Klammersausdrücke

- Sei  $L_{\langle 2 \rangle}$  die Sprache der korrekt geklammerten „Tag-Ausdrücke“ mit zwei Tag-Paaren  $\langle b \rangle \langle /b \rangle$  und  $\langle a \rangle \langle /a \rangle$ 
  - Also über dem Alphabet  $\Sigma = \{\langle a \rangle, \langle /a \rangle, \langle b \rangle \langle /b \rangle\}$

- $\langle a \rangle \langle a \rangle \langle b \rangle \langle /b \rangle \langle a \rangle \langle /a \rangle \langle /a \rangle \langle b \rangle \langle /b \rangle \langle /a \rangle$  ist korrekt
- $\langle a \rangle \langle a \rangle \langle b \rangle \langle /b \rangle \langle a \rangle \langle /b \rangle \langle /a \rangle \langle b \rangle \langle /b \rangle \langle /a \rangle$  ist nicht korrekt

- $L_{\langle 2 \rangle}$  ist kontextfrei und wird von der folgenden Grammatik erzeugt:

$$K \rightarrow KK \mid \langle b \rangle K \langle /b \rangle \mid \langle a \rangle K \langle /a \rangle \mid \epsilon$$

- Klar:  $L_{\langle 2 \rangle}$  ist nicht regulär:
  - die Strings  $\langle a \rangle^n$ ,  $n \geq 0$ , sind paarweise nicht äquivalent bezüglich  $\sim_{L_{\langle 2 \rangle}}$
- Wie lässt sich algorithmisch testen, ob ein gegebener Klammerausdruck korrekt ist?
- Idee:
  - Versuche zusammengehörige Klammern zu finden
  - Geeignete Datenstruktur:
    - \* Keller („Last In First Out“)

# Erkennen von Klammerausdrücken mit Hilfe eines Kellers

## Beispiel



$\langle b \rangle \langle a \rangle \langle b \rangle \langle a \rangle \langle a \rangle \langle /a \rangle \langle b \rangle \langle /b \rangle \langle /a \rangle \langle /b \rangle \langle a \rangle$   
 $\langle /a \rangle \langle /a \rangle \langle b \rangle \langle /b \rangle \langle /b \rangle$

- $\langle a \rangle$  und  $\langle b \rangle$  werden jeweils auf den Keller gelegt
- Wenn ein  $\langle /a \rangle$  gelesen wird, muss ein  $\langle a \rangle$  auf dem Keller sein  
(und wird gelöscht)
- Wenn ein  $\langle /b \rangle$  gelesen wird, muss ein  $\langle b \rangle$  auf dem Keller sein  
(und wird gelöscht)
- Am Schluss muss der Keller leer sein

# Kellerautomaten: informell

- Die Vorgehensweise dieses Algorithmus ähnelt einem endlichen Automaten:
  - Zeichenweises Lesen der Eingabe von links nach rechts
  - Am Ende Entscheidung, ob die Eingabe akzeptiert wird
- Allerdings verwendet der Algorithmus zusätzlich einen Keller
- Solche Algorithmen modellieren wir im Folgenden durch **Kellerautomaten**
- Wir werden sehen, dass Kellerautomaten genau die kontextfreien Sprachen entscheiden können
- Das eben betrachtete Beispiel war nur ein sehr einfacher Kellerautomat
- Im Allgemeinen erlauben wir zusätzlich:
  - Zustände (endlich viele)
  - Nichtdeterminismus
  - $\epsilon$ -Übergänge
  - Zwei mögliche Arten von Akzeptierungsbedingungen:
    - \* leerer Keller
    - \* akzeptierende Zustände
  - Zusätzliche Symbolmenge für Keller
  - Zusätzliches unterstes Kellersymbol
  - Schreiben mehrerer Kellersymbole in einem Schritt




# Kellerautomaten: Definition

## Definition

- Ein **Kellerautomat**  $\mathcal{A}$  besteht aus
  - einer Zustandsmenge  $Q$ ,
  - einem Eingabealphabet  $\Sigma$ ,
  - einem **Kellularphabet**  $\Gamma$   
(nicht notwendigerweise disjunkt zu  $\Sigma$ ),
  - einer endlichen **Transitionsrelation**  $\delta \subseteq (Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma) \times (Q \times \Gamma^*)$ ,
  - einem Startzustand  $s$ ,
  - einem **untersten Kellersymbol**  $\tau_0 \in \Gamma$ , und
  - einer Menge  $F$  akzeptierender Zustände

- Also:  $\mathcal{A} = (Q, \Sigma, \Gamma, \delta, s, \tau_0, F)$



 Englische Bezeichnung:  
*pushdown automaton*

- Deshalb Abkürzung: PDA

- Warum ist  $\delta$  so kompliziert?

- Das Verhalten des Automaten im nächsten Schritt darf abhängen von:
  - dem aktuellen Zustand  $p$
  - dem nächsten Eingabesymbol  $\sigma$
  - dem obersten Kellersymbol  $\tau$

- In einem Schritt:

- kann sich der Zustand ändern   $q$
- kann ein Eingabesymbol gelesen werden  muss aber nicht:  $\epsilon$
- kann sich der Kellerinhalt verändern:
  - \*  $\tau$  kann durch (möglicherweise) mehrere Symbole ersetzt werden

 String  $z$


- Transitionen sind deshalb von der Form
  - $(p, \sigma, \tau, q, z)$  mit  $\sigma \in \Sigma$  oder
  - $(p, \epsilon, \tau, q, z)$

# Kellerautomat für $L_{\langle 2 \rangle}$ : formal

## Beispiel

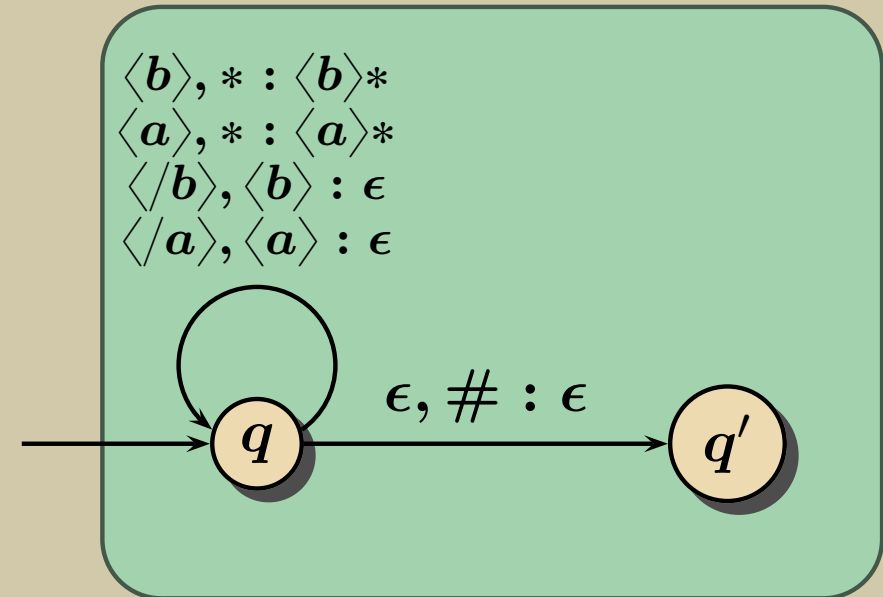
- Ein PDA  $\mathcal{A}_{\langle 2 \rangle}$  für die Sprache  $L_{\langle 2 \rangle}$ , der dem vorgestellten Algorithmus entspricht, lässt sich wie folgt definieren:  
 $(\{q, q'\}, \{\langle b \rangle, \langle a \rangle, \langle /a \rangle, \langle /b \rangle\}, \{\langle b \rangle, \langle a \rangle, \#\}, \delta, q, \#, \emptyset)$ ,  
wobei  $\delta$  die folgenden Transitionen enthält:
  - $(q, \langle a \rangle, \tau, q, \langle a \rangle \tau)$ , für alle  $\tau \in \Gamma$
  - $(q, \langle b \rangle, \tau, q, \langle b \rangle \tau)$ , für alle  $\tau \in \Gamma$
  - $(q, \langle /a \rangle, \langle a \rangle, q, \epsilon)$
  - $(q, \langle /b \rangle, \langle b \rangle, q, \epsilon)$
  - $(q, \epsilon, \#, q', \epsilon)$

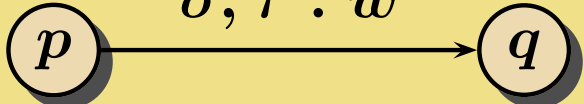
- Dabei ist  $\#$  das unterste Kellersymbol, das zu Beginn der Berechnung schon im Keller liegt und am Ende der Berechnung „anzeigt“, ob alle Klammern wieder vom Keller gelöscht wurden

 Im Unterschied zur informellen Darstellung im Beispiel

## Beispiel

- $\mathcal{A}_{\langle 2 \rangle}$  als Diagramm:



- 

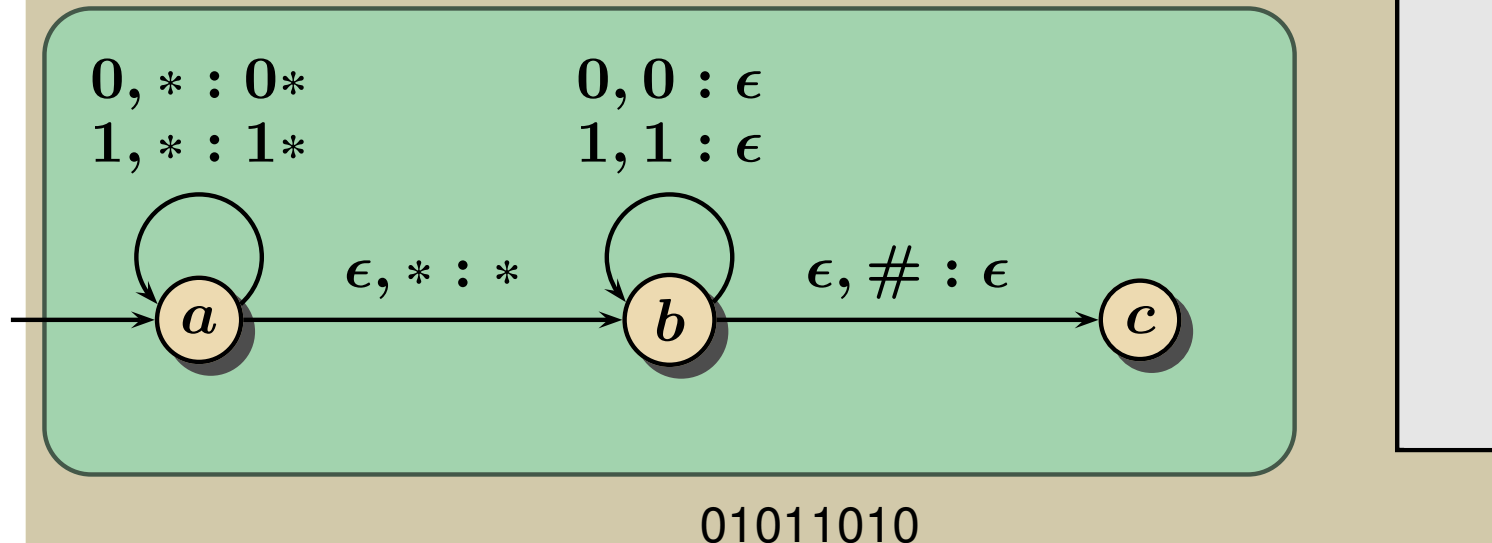
steht für  $(p, \sigma, \tau, q, w) \in \delta$

- Die abkürzende Schreibweise  $\langle b \rangle, * : \langle b \rangle^*$  bedeutet, dass alle Übergänge der Art  $\langle b \rangle, \tau : \langle b \rangle \tau$ , mit  $\tau \in \Gamma$  möglich sind

# Kellerautomaten: 2. Beispiel

## Beispiel

- Kellerautomat  $\mathcal{A}_{\text{rev}}$  für  $L_{\text{rev}} \stackrel{\text{def}}{=} \{ww^R \mid w \in \{0, 1\}^*\}$
- **Konstruktionsidee:**
  - „Rate“ die Stelle, an der  $w$  zu Ende ist
  - Kopiere bis zu dieser Stelle alles auf den Keller
  - Nach dieser Stelle vergleiche immer das nächste Eingabesymbol mit dem obersten Kellersymbol (und lösche dieses)



# Kellerautomaten: Konfigurationen


- Das zukünftige Verhalten eines endlichen Automaten hängt jeweils ab von:
  - dem aktuellen Zustand,
  - den noch zu lesenden Eingabezeichen

- Das zukünftige Verhalten eines Kellerautomaten hängt jeweils ab von:
  - dem aktuellen Zustand,
  - den noch zu lesenden Eingabezeichen,
  - **dem Kellerinhalt**

→ der Kellerinhalt muss für die Definition der Semantik von Kellerautomaten berücksichtigt werden

- Läufe (Berechnungen) bestehen bei PDAs also nicht nur aus Folgen von Zuständen und gelesenen Zeichen
- Stattdessen werden wir Folgen von **Konfigurationen** betrachten, die jeweils die aktuelle „Situation“ beschreiben

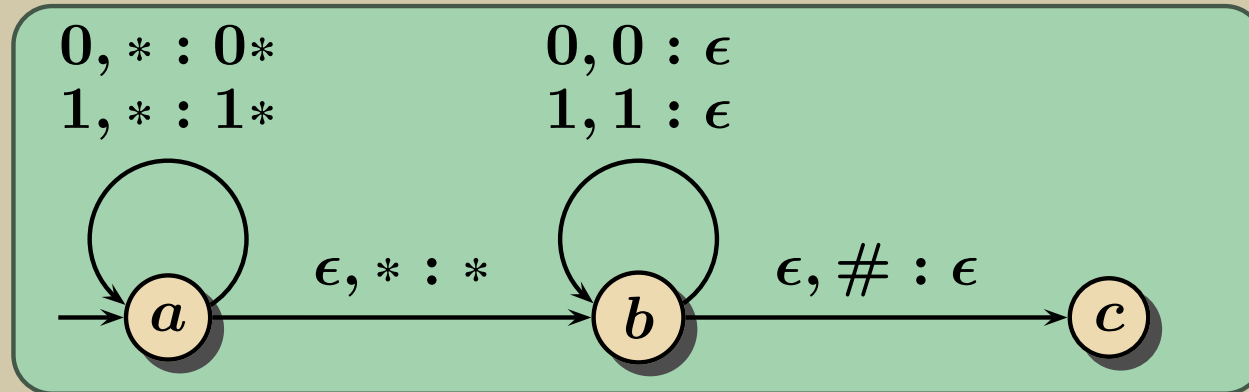
## Definition

- Sei  $\mathcal{A} = (Q, \Sigma, \Gamma, \delta, s, \tau_0, F)$  ein Kellerautomat
- Eine **Konfiguration**  $(q, u, v)$  von  $\mathcal{A}$  besteht aus:
  - einem Zustand  $q \in Q$
  - der noch zu lesenden Eingabe  $u \in \Sigma^*$
  - dem Kellerinhalt  $v \in \Gamma^*$ 
    -  Das erste Zeichen von  $v$  ist das oberste Kellerzeichen!

- Startkonfiguration bei Eingabe  $w$ :  
 $(s, w, \tau_0)$

# Konfigurationen: Beispiel

Beispiel




01011010

$(a, 01011010, \#) \vdash (a, 1011010, 0\#)$   
 $\vdash (a, 011010, 10\#)$   
 $\vdash (a, 11010, 010\#)$   
 $\vdash (a, 1010, 1010\#)$   
 $\vdash (b, 1010, 1010\#)$   
 $\vdash (b, 010, 010\#)$   
 $\vdash (b, 10, 10\#)$   
 $\vdash (b, 0, 0\#)$   
 $\vdash (b, \epsilon, \#)$   
 $\vdash (c, \epsilon, \epsilon)$

# Kellerautomaten: Konfigurationen und Berechnungen

## Definition

- Sei  $\mathcal{A} = (Q, \Sigma, \Gamma, \delta, s, \tau_0, F)$  ein PDA
- Die Nachfolgekongurationsrelation  $\vdash_{\mathcal{A}}$  ist wie folgt definiert
- Für alle  $p, q \in Q, \sigma \in \Sigma, \tau \in \Gamma, u \in \Sigma^*, z, v \in \Gamma^*$  gilt:
- $(p, \sigma u, \tau v) \vdash_{\mathcal{A}} (q, u, zv)$ , falls  $(p, \sigma, \tau, q, z) \in \delta$
- $(p, u, \tau v) \vdash_{\mathcal{A}} (q, u, zv)$ , falls  $(p, \epsilon, \tau, q, z) \in \delta$
- Wenn  $\underline{K \vdash_{\mathcal{A}} K'}$  gilt heißt  $K'$  (eine) Nachfolgekonfiguration von  $K$

 Wenn der Automat  $\mathcal{A}$  durch den Kontext klar ist, lassen wir das Subskript  $\mathcal{A}$  meist weg

## Definition

- Eine Berechnung (oder: ein Lauf) eines PDA  $\mathcal{A}$  ist eine Folge  $K_1, \dots, K_n$  von Konfigurationen mit
  - $K_i \vdash K_{i+1}$ , für alle  $i \in \{1, \dots, n-1\}$
- Schreibweise:  $K_1 \vdash_{\mathcal{A}}^* K_n$



Zu beachten:

- Wenn die Eingabe schon vollständig gelesen wurde, ist es immer noch möglich,  $\epsilon$ -Übergänge auszuführen,
  - \* aber: bei leerem Keller gibt es keine Nachfolgekonfiguration!

# Kellerautomaten: Akzeptieren

- Wann akzeptiert  $\mathcal{A}$  die Eingabe?
- Bei den bisherigen Beispielen galten am Ende der Berechnung die beiden folgenden Aussagen:
  - der Keller ist leer
  - der Automat ist in einem „speziellen“ Zustand
- Wir definieren zwei Varianten von PDAs, deren Akzeptieren jeweils auf **einer** dieser beiden Bedingungen basiert
- Denn: mal ist das eine praktischer, mal das andere
- Dann werden wir sehen:
  - Beide Modelle sind äquivalent

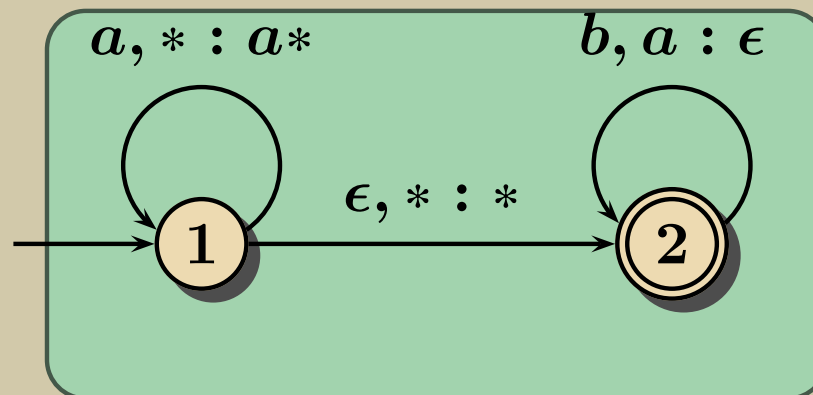
## Definition

- Sei  $\mathcal{A} = (Q, \Sigma, \Gamma, \delta, s, \tau_0, F)$  ein Kellerautomat
- $\mathcal{A}$  akzeptiert einen String  $w \in \Sigma^*$ , falls
  - $F = \emptyset$  und  $(s, w, \tau_0) \vdash^* (q, \epsilon, \epsilon)$  für ein  $q \in Q$   
☞ „Akzeptieren mit leerem Keller“
  - oder
  - $F \neq \emptyset$  und  $(s, w, \tau_0) \vdash^* (q, \epsilon, u)$  für ein  $u \in \Gamma^*$  und  $q \in F$   
☞ „Akzeptieren mit akzeptierenden Zuständen“
- $\underline{L(\mathcal{A})} \stackrel{\text{def}}{=} \{w \mid \mathcal{A} \text{ akzeptiert } w\}$
- Wir sagen, dass  $\mathcal{A}$  die Sprache  $L(\mathcal{A})$  entscheidet

# Akzeptierende Zustände: Beispielautomat

## Beispiel

- $L = \{a^i b^j \mid i \geq j\}$
- Idee:
  - 1.Phase: Lege jedes  $a$  auf den Keller
  - 2.Phase: Lösche für jedes  $b$  ein  $a$  vom Keller
  - Falls auf diese Weise der String ganz gelesen wird, akzeptiere (auch wenn noch etwas im Keller steht)



aaabb wird akzeptiert  
aaabbbb wird nicht akzeptiert

#



# Inhalt

9.1 Kellerautomaten: Definitionen

▷ **9.2 Leerer Keller vs. akzeptierende Zustände**

9.3 Grammatiken vs. Kellerautomaten

9.4 Kellerautomaten: Korrektheitsbeweise

9.5 Anhang: Beweisdetails

# Leerer Keller vs. akzeptierende Zustände

## Satz 9.1

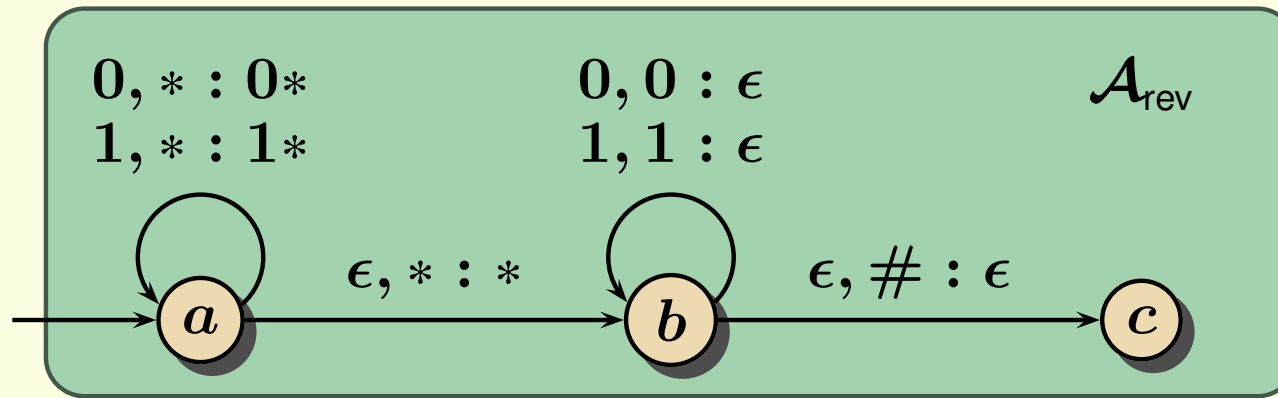
- (a) Für jeden Kellerautomaten  $\mathcal{A}$ , der mit leerem Keller akzeptiert, gibt es es einen Kellerautomaten  $\mathcal{B}$ , der mit akzeptierenden Zuständen akzeptiert und  $L(\mathcal{A}) = L(\mathcal{B})$  erfüllt
- (b) Für jeden Kellerautomaten  $\mathcal{A}$ , der mit akzeptierenden Zuständen akzeptiert, gibt es es einen Kellerautomaten  $\mathcal{B}$ , der mit leerem Keller akzeptiert und  $L(\mathcal{A}) = L(\mathcal{B})$  erfüllt
- Beide Beweise verwenden die Methode der **Simulation**:
  - Der Automat  $\mathcal{B}$  ahmt jeweils das Verhalten von  $\mathcal{A}$  nach
  - Kurz: „ $\mathcal{B}$  simuliert  $\mathcal{A}$ “

## Leerer Keller $\rightarrow$ akzeptierende Zustände: Idee

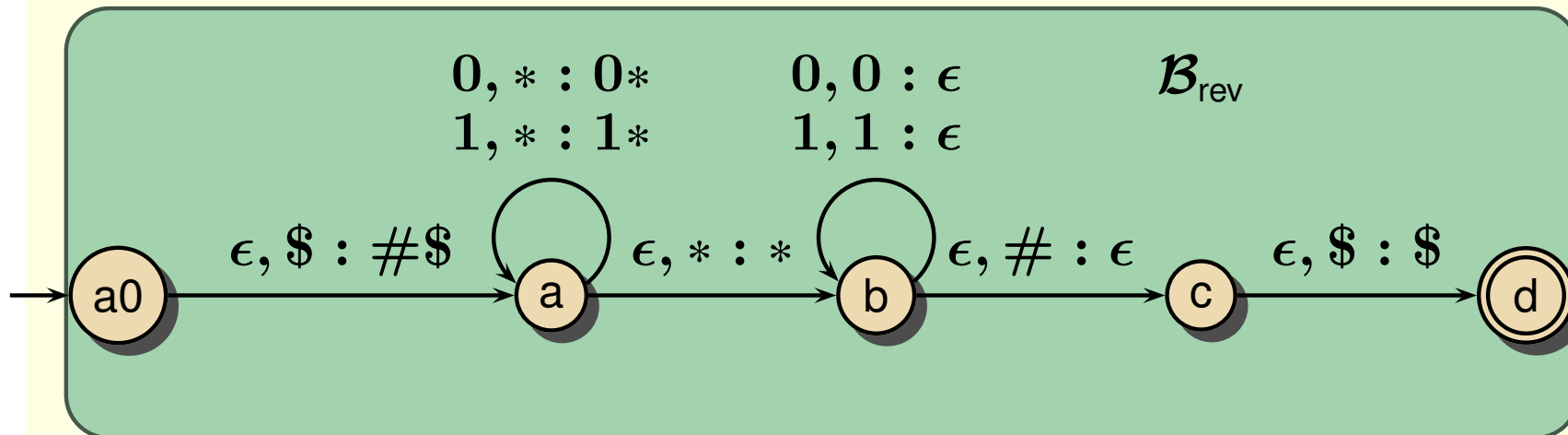
### Beweisidee zu Satz 9.1 (a)

- „Leerer Keller  $\rightarrow$  akzeptierende Zustände“
- **Herausforderung:** wenn in  $\mathcal{A}$  der Keller leer wird, ist keine weitere Transition in einen akzeptierenden Zustand möglich
- **Idee:**
  - $\mathcal{B}$  simuliert  $\mathcal{A}$
  - $\mathcal{B}$  verwendet gegenüber  $\mathcal{A}$  ein neues unterstes Kellersymbol  $\$$ , das zu Beginn der Simulation unter das unterste Kellersymbol  $\tau_0$  von  $\mathcal{A}$  gelegt wird
  - Wenn bei der Simulation in  $\mathcal{B}$  das Zeichen  $\$$  „sichtbar“ wird, wäre in  $\mathcal{A}$  der Keller leer
  - Falls bei der Simulation von  $\mathcal{A}$  das Symbol  $\$$  auf dem Keller zum Vorschein kommt, kann  $\mathcal{B}$  deshalb in den akzeptierenden Zustand übergehen

# Leerer Keller $\rightarrow$ akzeptierende Zustände: Beispiel



01011010

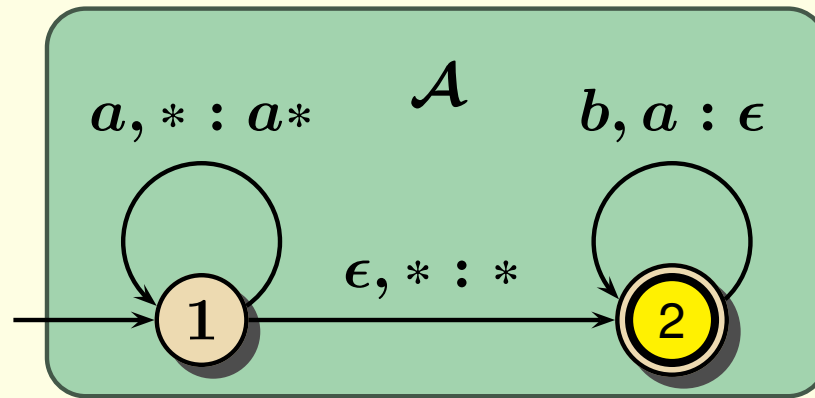


# Akzeptierende Zustände → Leerer Keller: Idee

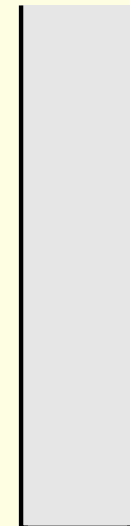
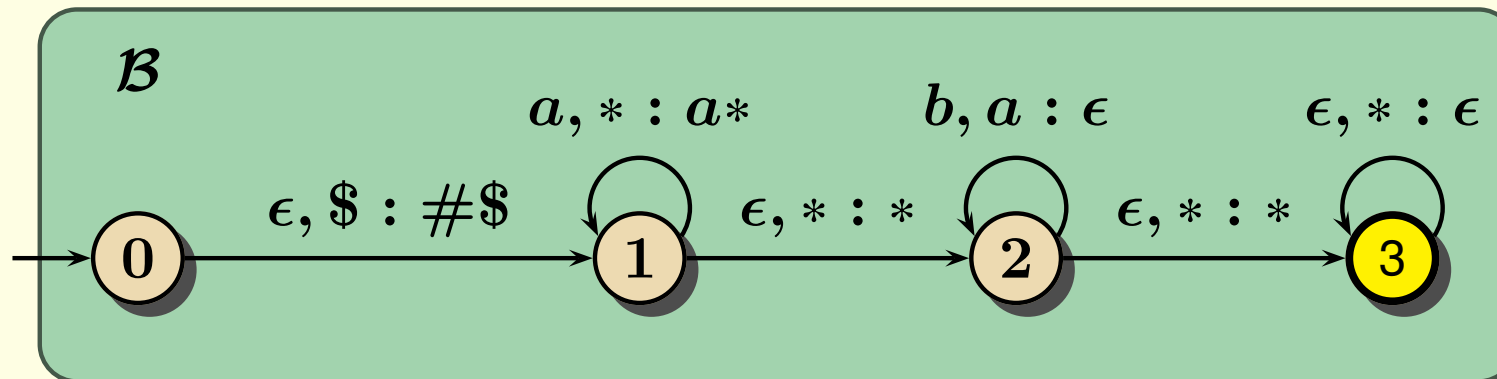
## Beweisidee zu Satz 9.1 (b)

- „Akzeptierende Zustände → leerer Keller“
- **Herausforderungen:**
  - Wenn  $\mathcal{A}$  am Ende in einem akzeptierenden Zustand landet, muss  $\mathcal{B}$  den Keller noch leeren
  - Wenn es eine Berechnung von  $\mathcal{A}$  gibt, die am Ende einen leeren Keller hat, aber keinen akzeptierenden Zustand, so soll diese Berechnung in  $\mathcal{B}$  nicht den Keller leeren
- **Idee:**
  - $\mathcal{B}$  simuliert  $\mathcal{A}$
  - Von jedem Zustand in  $F$  aus kann  $\mathcal{B}$  in den „Aufräumzustand“  $q_a$  übergehen und dann den Keller mit Hilfe von  $\epsilon$ -Übergängen leeren
  - Wenn die Eingabe vollständig gelesen war, führt das zum Akzeptieren mit leerem Keller
  - Damit keine Berechnung von  $\mathcal{A}$  fälschlich zum Akzeptieren von  $\mathcal{B}$  führt, indem  $\mathcal{A}$  den Keller selbst leert (ohne in einen akzeptierenden Zustand zu gehen), verwendet  $\mathcal{B}$  wieder ein neues unterstes Kellersymbol  $\$$

# Akzeptierende Zustände $\rightarrow$ Leerer Keller: Beispiel



aaabb|



# Inhalt

9.1 Kellerautomaten: Definitionen

9.2 Leerer Keller vs. akzeptierende Zustände

▷ **9.3 Grammatiken vs. Kellerautomaten**

9.4 Kellerautomaten: Korrektheitsbeweise

9.5 Anhang: Beweisdetails

# Äquivalenz von Grammatiken und Kellerautomaten

- Ziel: Nachweis, dass Kellerautomaten genau die kontextfreien Sprachen entscheiden

## Satz 9.2

- Zu jeder kontextfreien Grammatik  $G$  gibt es einen Kellerautomaten  $\mathcal{A}$  mit  $L(\mathcal{A}) = L(G)$
- Der Beweis von Satz 9.2 ist nicht sehr schwierig und folgt einer einfachen Idee:
  - Der Kellerautomat versucht eine Linksableitung zu finden

## Satz 9.3

- Zu jedem Kellerautomaten  $\mathcal{A}$  gibt es eine kontextfreie Grammatik  $G$  mit  $L(G) = L(\mathcal{A})$
- Der Beweis von Satz 9.3 ist deutlich komplizierter



# Grammatik $\rightarrow$ Kellerautomat: Idee

## Satz 9.2


- Zu jeder kontextfreien Grammatik  $G$  gibt es einen Kellerautomaten  $\mathcal{A}$  mit
$$L(\mathcal{A}) = L(G)$$

## Beweisidee

- Sei  $G = (V, \Sigma, S, P)$
- **Idee:**
  - Der Kellerautomat  $\mathcal{A}$  erzeugt bei Eingabe  $w$  eine Linksableitung für ein Wort  $v$  und testet, dass  $w = v$  gilt
  - Erzeugen und Testen sind dabei ineinander verschränkt:
    - \* Wenn die aktuelle Satzform mit einem Terminalsymbol anfängt, wird dieses gleich mit  $w$  verglichen

## Beweisidee (Forts.)

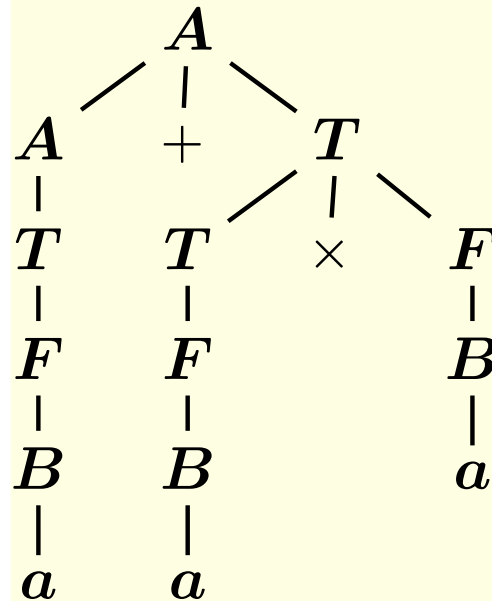
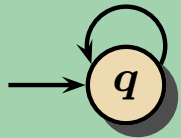
- Ein einzelner Schritt einer Linksableitung
  - ersetzt in einer Satzform der Art  $uX\alpha$
  - die Variable  $X$  durch einen String  $\beta$
  - gemäß einer Regel  $X \rightarrow \beta$

 mit  $u \in \Sigma^*$ ,  $X \in V$ ,  
 $\alpha \in (\Sigma \cup V)^*$
- In  $\mathcal{A}$  soll dies der folgenden Situation entsprechen:
  - $u$  ist schon gelesen,  $X\alpha$  ist der Kellerinhalt
  - Zur Umsetzung des Ableitungsschrittes geht  $\mathcal{A}$  wie folgt vor:
    1. „Rate“ Regel  $X \rightarrow \beta \in P$
    2. Ersetze auf dem Keller  $X$  durch  $\beta$
    3. Vergleiche die führenden Terminalsymbole von  $\beta\alpha$  mit den nächsten Zeichen der Eingabe und reduziere sie (= lösche sie vom Keller)

# Grammatik → Kellerautomat: Beispiel

$$\begin{aligned}
 A &\rightarrow A + T \mid T \\
 T &\rightarrow T \times F \mid F \\
 F &\rightarrow (A) \mid B \\
 B &\rightarrow a \mid b \mid Ba \mid \\
 &\quad Bb \mid B0 \mid B1
 \end{aligned}$$

$a, a : \epsilon$   
 $b, b : \epsilon$   
 $0, 0 : \epsilon$   
 $1, 1 : \epsilon$   
 $\times, \times : \epsilon$   
 $+, + : \epsilon$   
 $(, ( : \epsilon$   
 $), ) : \epsilon$   
 $\epsilon, A : A + T$   
 $\epsilon, A : T$   
 $\epsilon, T : T \times F$   
 $\epsilon, T : F$   
 $\epsilon, F : (A)$   
 $\epsilon, F : B$   
 $\epsilon, B : a$   
 $\epsilon, B : b$   
 $\epsilon, B : Ba$   
 $\epsilon, B : Bb$   
 $\epsilon, B : B0$   
 $\epsilon, B : B1$



$a + a \times a$

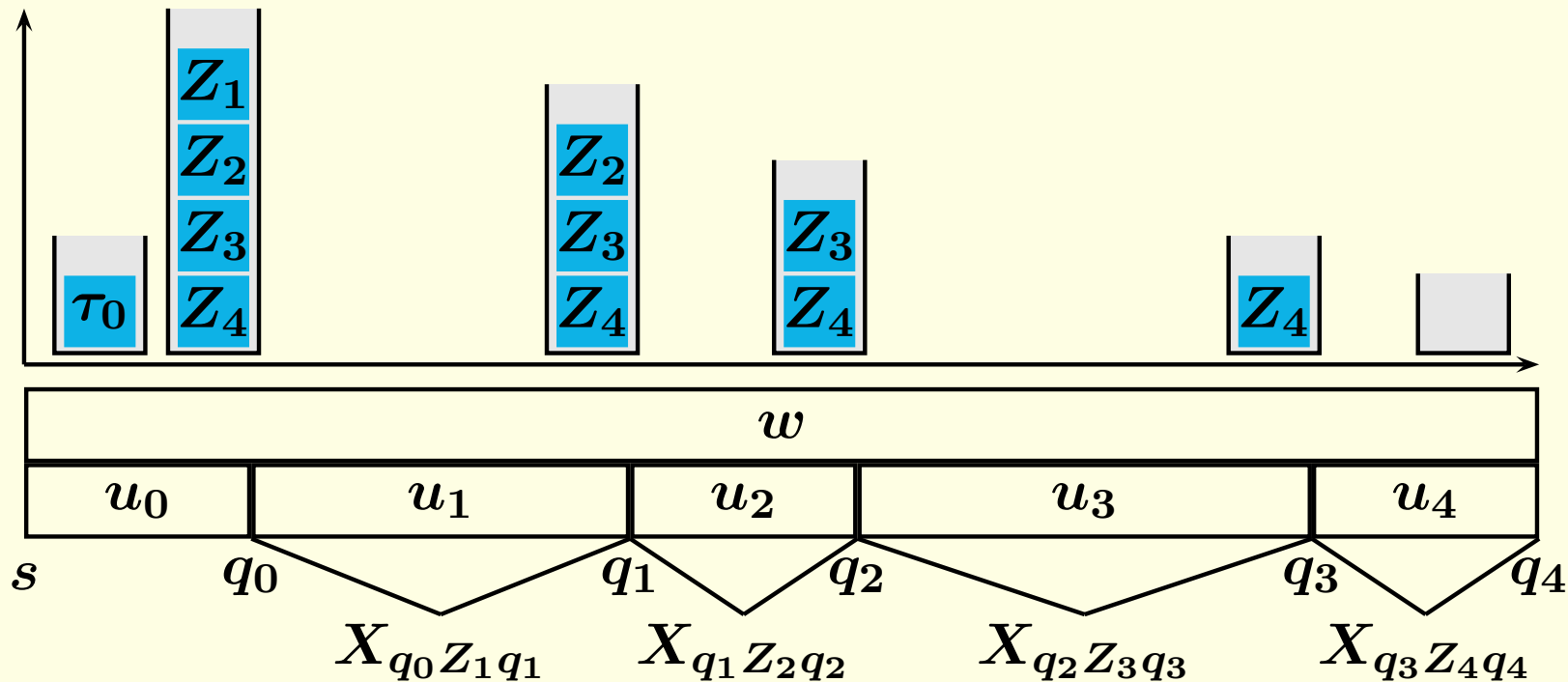
$(q, a + a \times a, A)$   
 $\vdash (q, a + a \times a, A + T)$   
 $\vdash (q, a + a \times a, T + T)$   
 $\vdash (q, a + a \times a, F + T)$   
 $\vdash (q, a + a \times a, B + T)$   
 $\vdash (q, a + a \times a, a + T)$   
 $\vdash (q, +a \times a, +T)$   
 $\vdash (q, a \times a, T)$   
 $\vdash (q, a \times a, T \times F)$   
 $\vdash (q, a \times a, F \times F)$   
 $\vdash (q, a \times a, B \times F)$   
 $\vdash (q, a \times a, a \times F)$   
 $\vdash (q, \times a, \times F)$   
 $\vdash (q, a, F)$   
 $\vdash (q, a, B)$   
 $\vdash (q, a, a)$   
 $\vdash (q, \epsilon, \epsilon)$

# Grammatik $\rightarrow$ Kellerautomat: Konstruktion

## Beweis von Satz 9.2

- Sei  $G = (V, \Sigma, S, P)$
- $\mathcal{A} \stackrel{\text{def}}{=} (\{q\}, \Sigma, V \cup \Sigma, \delta, q, S, \emptyset)$ ,
  - $\delta \stackrel{\text{def}}{=} \{(q, \sigma, \sigma, q, \epsilon) \mid \sigma \in \Sigma\} \cup \{(q, \epsilon, X, q, \alpha) \mid X \rightarrow \alpha \in P\}$
- Eine detaillierte Beweisskizze findet sich im Anhang

# Kellerautomat → Grammatik: Idee

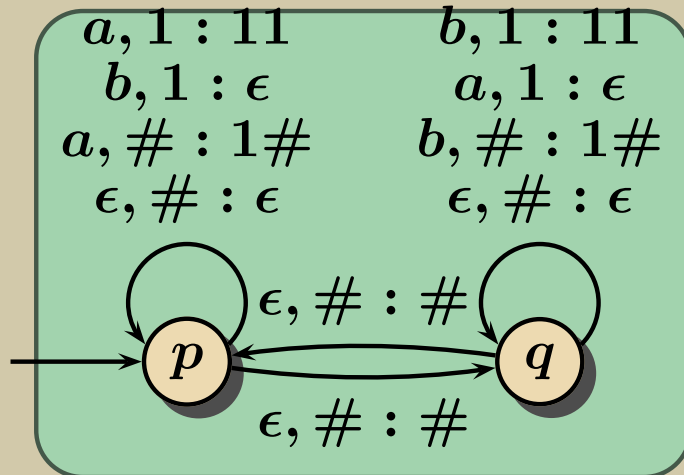


- Im ersten Schritt ersetzt  $\mathcal{A}$  das unterste Kellersymbol durch einen String  $Z_1 \cdots Z_k$  und liest ein Präfix  $u_0$  der Eingabe  $w$  ( $u_0 \in \Sigma \cup \{\epsilon\}$ )
- Die Zeichen  $Z_1, \dots, Z_k$  werden dann im Rest der Berechnung nach und nach wieder vom Keller gelöscht
- Dabei werden Teilstrings  $u_1, \dots, u_k$  der Eingabe gelesen
- Idee für die Grammatik:
  - Für jede Kombination  $p, p' \in Q, \tau \in \Gamma$  enthält  $G$  eine Variable  $X_{p, \tau, p'}$ , die alle Strings erzeugt, für die  $\mathcal{A}$  eine Teilberechnung von Zustand  $p$  in Zustand  $p'$  hat, die insgesamt das Zeichen  $\tau$  vom Keller löscht

# Kellerautomat → Grammatik: Beispiel

## Beispiel

- Kellerautomat für  $L_{a=b} = \{w \in \{a, b\}^* \mid \#_a(w) = \#_b(w)\}$ :



- Zustand  $p$ :
  - \* Mindestens so viele  $a$  wie  $b$  gelesen
  - \* Anzahl der Einsen auf dem Keller entspricht Überschuss an  $a$ 's
- Zustand  $q$ : analog umgekehrt

- Die daraus entstehende Grammatik:

$$S \rightarrow X_{p\#p} \mid X_{p\#q}$$

$$X_{p\#p} \rightarrow aX_{p1p}X_{p\#p} \mid aX_{p1q}X_{q\#p} \mid \epsilon \mid X_{q\#p}$$

$$X_{p\#q} \rightarrow aX_{p1p}X_{p\#q} \mid aX_{p1q}X_{q\#q} \mid X_{q\#q}$$

$$X_{p1p} \rightarrow aX_{p1p}X_{p1p} \mid aX_{p1q}X_{q1p} \mid b$$

$$X_{p1q} \rightarrow aX_{p1p}X_{p1q} \mid aX_{p1q}X_{q1q}$$

$$X_{q\#q} \rightarrow bX_{q1q}X_{q\#q} \mid bX_{q1p}X_{p\#q} \mid \epsilon \mid X_{p\#q}$$

$$X_{q\#p} \rightarrow bX_{q1q}X_{q\#p} \mid bX_{q1p}X_{p\#p} \mid X_{p\#p}$$

$$X_{q1q} \rightarrow bX_{q1q}X_{q1q} \mid bX_{q1p}X_{p1q} \mid a$$

$$X_{q1p} \rightarrow bX_{q1q}X_{q1p} \mid bX_{q1p}X_{p1p}$$

# Kellerautomat $\rightarrow$ Grammatik: Beweis (1/2)

## Satz 9.3

- Zu jedem Kellerautomaten  $\mathcal{A}$  gibt es eine kontextfreie Grammatik  $G$  mit  $L(G) = L(\mathcal{A})$

## Beweisidee

- Sei  $\mathcal{A} = (Q, \Sigma, \Gamma, \delta, s, \tau_0, \emptyset)$ 
  - (oBdA:  $\mathcal{A}$  akzeptiert mit leerem Keller)
- Weitere Annahme (oBdA):  $\mathcal{A}$  legt in einem Schritt maximal zwei Zeichen auf den Keller
- Wir konstruieren eine Grammatik  $G_{\mathcal{A}}$  mit Variablen  $X_{p,\tau,p'}$ , für alle  $p, p' \in Q$  und  $\tau \in \Gamma$  mit der folgenden Intention:
  - $X_{p,\tau,p'} \Rightarrow^* w$  soll gelten, falls  $\mathcal{A}$  durch Lesen von  $w$  vom Zustand  $p$  in den Zustand  $p'$  kommen und dabei insgesamt  $\tau$  vom Keller löschen kann
  - Formal soll also gelten:
$$X_{p,\tau,p'} \Rightarrow^* w \iff (p, w, \tau) \vdash_{\mathcal{A}}^* (p', \epsilon, \epsilon)$$

# Kellerautomat → Grammatik: Beweis (2/2)

## Beweis von Satz 9.3

- $P$  enthält pro Tupel in  $\delta$  eine oder mehrere Regeln, jeweils für alle möglichen Zustände  $p_1, p_2$ :

| $\delta$                              | $P$   |
|---------------------------------------|---|
| $(p, \alpha, \tau, q, \epsilon)$      | $X_{p,\tau,q} \rightarrow \alpha$                                       |
| $(p, \alpha, \tau, q, \tau_1)$        | $X_{p,\tau,p_1} \rightarrow \alpha X_{q,\tau_1,p_1}$                    |
| $(p, \alpha, \tau, q, \tau_1 \tau_2)$ | $X_{p,\tau,p_2} \rightarrow \alpha X_{q,\tau_1,p_1} X_{p_1,\tau_2,p_2}$ |

- Dabei ist  $\alpha \in \Sigma \cup \{\epsilon\}$  (also: Zeichen oder Leerstring)
- Zusätzlich hat  $G_{\mathcal{A}}$  das Startsymbol  $S$  und Regeln  $S \rightarrow X_{s,\tau_0,q}$ , für jedes  $q \in Q$
- **Behauptung:**

$$X_{p,\tau,p'} \Rightarrow^* w \iff (p, w, \tau) \vdash_{\mathcal{A}}^* (p', \epsilon, \epsilon)$$
- „ $\Leftarrow$ “: Induktion nach der Anzahl der Berechnungsschritte
- „ $\Rightarrow$ “: Induktion nach der Anzahl der Ableitungsschritte
- Die Details finden sich im Anhang

# Kellerautomaten und Grammatiken: Fazit

## Satz 9.4

- Für eine Sprache  $L$  sind äquivalent:
  - $L$  ist kontextfrei
  - $L$  wird von einem Kellerautomaten mit akzeptierenden Zuständen entschieden
  - $L$  wird von einem Kellerautomaten mit leerem Keller entschieden

- Wie groß werden die bei der Umwandlung konstruierten Objekte?
  - Grammatik  $\rightarrow$  Kellerautomat:  $\mathcal{O}(n)$
  - Kellerautomat  $\rightarrow$  Grammatik:  $\mathcal{O}(n^4)$
  - Zwischen Kellerautomaten:  $\mathcal{O}(n)$

- Aus Satz 9.3 und dem Beweis von Satz 9.2 folgt außerdem folgende Normalform für Kellerautomaten:

## Folgerung

- Zu jedem Kellerautomaten gibt es einen äquivalenten Kellerautomaten mit nur einem Zustand



# Inhalt

9.1 Kellerautomaten: Definitionen

9.2 Leerer Keller vs. akzeptierende Zustände

9.3 Grammatiken vs. Kellerautomaten

▷ **9.4 Kellerautomaten: Korrektheitsbeweise**

9.5 Anhang: Beweisdetails

# Intervall-Notation für Teilstrings

- Wir verwenden zukünftig die folgende Notation, um über Teilstrings und einzelne Zeichen von Strings zu sprechen
- Sei  $w = \sigma_1 \cdots \sigma_n$  ein String  
(der Länge  $n$ )
- Dann sei, für alle  $i, j \in \{1, \dots, n\}$  mit  $i \leq j$ :
  - $\underline{w[i]} \stackrel{\text{def}}{=} \sigma_i$
  - $\underline{w[i, j]} \stackrel{\text{def}}{=} \sigma_i \cdots \sigma_j$
  - $\underline{w[i, j)} \stackrel{\text{def}}{=} \sigma_i \cdots \sigma_{j-1}$
  - $\underline{w(i, j]} \stackrel{\text{def}}{=} \sigma_{i+1} \cdots \sigma_j$
  - $\underline{w(i, j)} \stackrel{\text{def}}{=} \sigma_{i+1} \cdots \sigma_{j-1}$   
(nur für  $i < j$ )
  - $\underline{w[*, j]} \stackrel{\text{def}}{=} \sigma_1 \cdots \sigma_j$
  - $\underline{w[i, *]} \stackrel{\text{def}}{=} \sigma_i \cdots \sigma_n$
- Für  $i > j$  sei  $w[i, j] \stackrel{\text{def}}{=} \epsilon$

## Beispiel

- Sei  $w = acbbcabba$
- Dann ist
  - $w[3] = b$
  - $w[4, 6] = bca$
  - $w[4, 6) = bc$
  - $w(4, 6] = ca$
  - $w(4, 6) = c$
  - $w(4, 5) = \epsilon$
  - $w[*, 3] = acb$
  - $w[5, *] = cabba$

# Kellerautomaten: Korrektheitsbeweise (1/2)

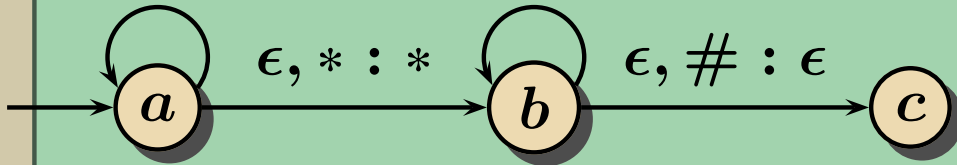
## Beispiel

0, \* : 0\*

0, 0 :  $\epsilon$

1, \* : 1\*

1, 1 :  $\epsilon$



## Proposition 9.5

$$L(\mathcal{A}_{\text{rev}}) = L_{\text{rev}}$$

- Zur Erinnerung:

$$L_{\text{rev}} = \{ww^R \mid w \in \{0, 1\}^*\}$$

☞ Palindrome gerader Länge

## Beweisskizze

- Wir beweisen:

- $L_{\text{rev}} \subseteq L(\mathcal{A}_{\text{rev}})$

☞ Vollständigkeit

- $L(\mathcal{A}_{\text{rev}}) \subseteq L_{\text{rev}}$

☞ Korrektheit

## Beweisskizze für „ $L_{\text{rev}} \subseteq L(\mathcal{A}_{\text{rev}})$ “

- Wir zeigen, dass für beliebige  $w \in \Sigma^*$  der String  $ww^R$  von  $\mathcal{A}_{\text{rev}}$  akzeptiert wird
- Dazu lässt sich durch Induktion nach der Länge von  $w$  beweisen:

(a)  $(a, ww^R, \#) \vdash^* (a, w^R, w^R\#)$   
und

(b)  $(b, w^R, w^R\#) \vdash^* (b, \epsilon, \#)$

- Dann folgt:

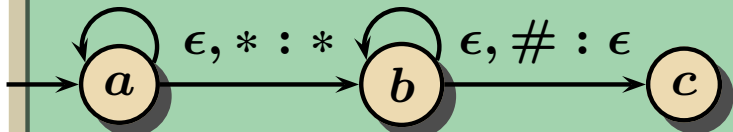
$$\begin{aligned} (a, ww^R, \#) &\vdash^* (a, w^R, w^R\#) \\ &\vdash (b, w^R, w^R\#) \\ &\vdash^* (b, \epsilon, \#) \\ &\vdash (c, \epsilon, \epsilon) \end{aligned}$$

➡  $ww^R \in L(\mathcal{A}_{\text{rev}})$

# Kellerautomaten: Korrektheitsbeweise (2/2)

## Beispiel

$0, * : 0*$        $0, 0 : \epsilon$   
 $1, * : 1*$        $1, 1 : \epsilon$



## Beweisskizze für „ $L(\mathcal{A}_{\text{rev}}) \subseteq L_{\text{rev}}$ “

- Klar: ein String  $v$  ist genau dann in  $L_{\text{rev}}$ , wenn
  - er gerade Länge  $n = 2k$  hat und
  - für jedes  $i \leq k$  gilt:  

$$v[i] = v[n - i + 1]$$

## Beweisskizze (Forts.)

- Sei  $v$  ein von  $\mathcal{A}_{\text{rev}}$  akzeptierter String mit  $n$  Zeichen  
 $\Rightarrow (a, v, \#) \vdash^* (b, \epsilon, \#) \vdash (c, \epsilon, \epsilon)$
- Nach Konstruktion von  $\mathcal{A}$  verläuft diese Berechnung in drei Phasen
  1.  $\mathcal{A}_{\text{rev}}$  liest ein Präfix  $v[1, k]$  der Eingabe (für ein  $k \leq n$ ) und schreibt es zeichenweise auf den Keller,
  2. dann geht  $\mathcal{A}_{\text{rev}}$  in den Zustand  $b$  über
  3. schließlich liest  $\mathcal{A}_{\text{rev}}$  die restliche Eingabe  $v[k + 1, n]$  und vergleicht sie mit den zuvor auf den Keller geschriebenen Zeichen
- Durch Induktion lässt sich zeigen:
  - Die Konfiguration nach Phase 1 ist  

$$(a, v[k + 1, n], v[1, k]^R \#)$$
  - Damit Phase 3 erfolgreich ist, muss gelten:
    - \*  $n - k = k \Rightarrow n = 2k$
    - \* für jedes  $i \leq k$  ist  $v[i] = v[n - i + 1]$
- $\Rightarrow v \in L_{\text{rev}}$

# Zusammenfassung

- Kellerautomaten entstehen durch Erweiterung von  $\epsilon$ -NFAs um einen Keller (LIFO)
- Kellerautomaten, die durch leeren Keller akzeptieren, sind genauso mächtig wie Kellerautomaten, die mit akzeptierenden Zuständen akzeptieren
- Mit Kellerautomaten und kontextfreien Grammatiken lassen sich genau dieselben Sprachen beschreiben: die kontextfreien Sprachen
- Der Kellerautomat zu einer Grammatik versucht eine Linksableitung zu finden
- Die Konstruktion der Grammatik zu einem Kellerautomaten ist erheblich komplizierter

# Inhalt

9.1 Kellerautomaten: Definitionen

9.2 Leerer Keller vs. akzeptierende Zustände

9.3 Grammatiken vs. Kellerautomaten

9.4 Kellerautomaten: Korrektheitsbeweise

▷ **9.5 Anhang: Beweisdetails**

# Leerer Keller vs. akzeptierende Zustände: Beweisideen (1/4)

- Der Beweis der Äquivalenz der beiden Akzeptiermethoden von PDAs verwendet mehrfach die folgende einfache Erkenntnis:
  - Eine Berechnung wird nur von den wirklich gelesenen Zeichen der Eingabe und des Kellers beeinflusst:
- (a) Deshalb können die Schritte einer Berechnung immer noch ausgeführt werden, wenn hinter der Eingabe und unter dem Keller etwas hinzugefügt wird
- (b) Andererseits können Zeichen der Eingabe, die während einer (partiellen) Berechnung (noch) nicht gelesen wurden und Zeichen des Kellers, die niemals sichtbar werden, entfernt werden, ohne die Berechnung zu beeinflussen

• Notation:  $\underline{K \vdash_{(\gamma)}^* K'} \stackrel{\text{def}}{\iff}$

es gibt eine Berechnung  $K \vdash \dots \vdash K'$ , in der jede Konfiguration **vor**  $K'$  einen String  $u\gamma$  mit  $u \neq \epsilon$  im Keller stehen hat

 In  $K'$  kann  $\gamma$  „alleine“ im Keller stehen

## Lemma 9.6



- Sei  $\mathcal{A} = (Q, \Sigma, \Gamma, \delta, s, \tau_0, F)$  ein Kellerautomat
- Seien
  - $x, y, w \in \Sigma^*$ ,
  - $\alpha \in \Gamma^+, \beta, \gamma \in \Gamma^*$ ,
  - $p, q \in Q$
- Dann sind äquivalent:
  - (a)  $(p, x, \alpha) \vdash^* (q, y, \beta)$
  - (b)  $(p, xw, \alpha\gamma) \vdash_{(\gamma)}^* (q, yw, \beta\gamma)$
- Der Beweis kann leicht durch Induktion nach der Berechnungslänge geführt werden

# Leerer Keller vs. akzeptierende Zustände: Beweisideen (2/4)

## Beweisidee zu Satz 9.1 (a)

- „Leerer Keller  $\rightarrow$  akzeptierende Zustände“
- **Idee:**
  - $\mathcal{B}$  simuliert  $\mathcal{A}$
  - $\mathcal{B}$  verwendet gegenüber  $\mathcal{A}$  ein neues unterstes Kellersymbol  $\$$ , das zu Beginn der Simulation unter das unterste Kellersymbol  $\tau_0$  von  $\mathcal{A}$  gelegt wird
  - Wenn bei der Simulation in  $\mathcal{B}$  das Zeichen  $\$$  „sichtbar“ wird, wäre in  $\mathcal{A}$  der Keller leer
  - Falls bei der Simulation von  $\mathcal{A}$  das Symbol  $\$$  auf dem Keller zum Vorschein kommt, kann  $\mathcal{B}$  deshalb in den akzeptierenden Zustand übergehen

## Beweisansatz zu Satz 9.1 (a)

- Sei  $\mathcal{A} = (Q, \Sigma, \Gamma, \delta, s, \tau_0, \emptyset)$
- Sei  $\mathcal{B} \stackrel{\text{def}}{=} (Q \cup \{q_0, q_a\}, \Sigma, \Gamma \cup \{\$\}, \delta', q_0, \$, \{q_a\})$
- Dabei sind:
  - $q_0, q_a \notin Q$  neue Zustände, und
  - $\$ \notin \Gamma$  ein neues Kellersymbol
- $\delta'$  enthält:
  - alle Transitionen von  $\delta$
  - $(q_0, \epsilon, \$, s, \tau_0 \$)$   Initialisierung
  - $(q, \epsilon, \$, q_a, \$)$ , für alle  $q \in Q$   
 entspricht leerem Keller in  $\mathcal{A}$



# Leerer Keller vs. akzeptierende Zustände: Beweiseideen (3/4)

## Beweisdetails zu Satz 9.1 (a)

- Zur Erinnerung:
  - $\mathcal{A} = (Q, \Sigma, \Gamma, \delta, s, \tau_0, \emptyset)$
  - $\mathcal{B} \stackrel{\text{def}}{=} (Q \cup \{q_0, q_a\}, \Sigma, \Gamma \cup \{\$, \delta', q_0, \$, \{q_a\})$
  - $\delta'$  enthält:
    - \* alle Transitionen von  $\delta$
    - \*  $(q_0, \epsilon, \$, s, \tau_0 \$)$
    - \*  $(q, \epsilon, \$, q_a, \$)$ ,
 für alle  $q \in Q$
- **Behauptung:**  $L(\mathcal{B}) = L(\mathcal{A})$
- Ausnahmsweise zeigen wir nicht zwei Inklusionen sondern direkt, dass für alle Strings  $w \in \Sigma^*$  gilt:
 
$$w \in L(\mathcal{B}) \iff w \in L(\mathcal{A})$$
- Für den Beweis sei  $w \in \Sigma^*$  beliebig

## Beweisdetails zu Satz 9.1 (a) (Forts.)



- Der erste Schritt von  $\mathcal{B}$  bei Eingabe  $w$  ist auf jeden Fall  $(q_0, w, \$) \vdash_{\mathcal{B}} (s, w, \tau_0 \$)$
- Da  $\mathcal{B}$  genau dann in  $q_a$  übergeht, wenn  $\$$  oberstes Kellersymbols ist, gilt für alle  $u \in \Gamma^*$ :
 
$$(s, w, \tau_0 \$) \vdash_{\mathcal{B}}^* (q_a, \epsilon, u) \Rightarrow u = \$$$
 Und:  $(s, w, \tau_0 \$) \vdash_{\mathcal{B}}^* (q_a, \epsilon, \$) \Rightarrow$   
 es gibt ein  $q: (s, w, \tau_0 \$) \vdash_{\mathcal{B},(\$)}^* (q, \epsilon, \$)$
- Wegen Lemma 9.6 gilt, für alle  $q$ :
 
$$(s, w, \tau_0 \$) \vdash_{\mathcal{B},(\$)}^* (q, \epsilon, \$) \iff (s, w, \tau_0) \vdash_{\mathcal{B}}^* (q, \epsilon, \epsilon)$$
- Da  $\mathcal{A}$  und  $\mathcal{B}$  identisch arbeiten, solange  $\$$  nicht zu sehen ist, gilt, für alle  $q$ :
 
$$(s, w, \tau_0) \vdash_{\mathcal{B}}^* (q, \epsilon, \epsilon) \iff (s, w, \tau_0) \vdash_{\mathcal{A}}^* (q, \epsilon, \epsilon)$$
- Insgesamt haben wir also:
 
$$(s, w, \tau_0 \$) \vdash_{\mathcal{B}}^* (q_a, \epsilon, u) \iff \text{es gibt ein } q: (s, w, \tau_0) \vdash_{\mathcal{A}}^* (q, \epsilon, \epsilon)$$
- ➔  $w \in L(\mathcal{B}) \iff w \in L(\mathcal{A})$

# Leerer Keller vs. akzeptierende Zustände: Beweisideen (4/4)

## Beweisidee zu Satz 9.1 (b)

- „Akzeptierende Zustände  $\rightarrow$  leerer Keller“
- **Idee:**
  - $\mathcal{B}$  simuliert  $\mathcal{A}$
  - Von jedem Zustand in  $F$  aus kann  $\mathcal{B}$  in den „Aufräumzustand“  $q_a$  übergehen und dann den Keller mit Hilfe von  $\epsilon$ -Übergängen leeren
  - Wenn die Eingabe vollständig gelesen war, führt das zum Akzeptieren mit leerem Keller
  - Damit keine Berechnung von  $\mathcal{A}$  fälschlich zum Akzeptieren von  $\mathcal{B}$  führt, indem  $\mathcal{A}$  den Keller selbst leert (ohne in einen akzeptierenden Zustand zu gehen), verwendet  $\mathcal{B}$  wieder ein neues unterstes Kellersymbol  $\$$

## Beweisansatz zu Satz 9.1 (b) (Forts.)

- Sei  $\mathcal{A} = (Q, \Sigma, \Gamma, \delta, s, \tau_0, F)$
- Sei  $\mathcal{B} \stackrel{\text{def}}{=} (Q \cup \{q_0, q_a\}, \Sigma, \Gamma \cup \{\$, \}, \delta', q_0, \$, \emptyset)$ 
  - $q_0, q_a \notin Q, \$ \notin \Gamma$  (wie zuvor)
- $\delta'$  enthält:
  - alle Transitionen aus  $\delta$
  - $(q_0, \epsilon, \$, s, \tau_0 \$)$   Initialisierung
  - $(q, \epsilon, \tau, q_a, \tau)$ , für alle  $q \in F$ 
    - \* Aus akzeptierenden Zuständen ist ein Übergang nach  $q_a$  möglich
  - $(q_a, \epsilon, \tau, q_a, \epsilon)$   zum Leeren des Kellers
- **Behauptung:**  $L(\mathcal{B}) = L(\mathcal{A})$   
(ohne Beweis)

# Grammatik → Kellerautomat: Beweisdetails (1/3)

## Beweis von Satz 9.2

- Sei  $G = (V, \Sigma, S, P)$
- $\mathcal{A} \stackrel{\text{def}}{=} (\{q\}, \Sigma, V \cup \Sigma, \delta, q, S, \emptyset)$ ,
  - $\delta \stackrel{\text{def}}{=} \{(q, \sigma, \sigma, q, \epsilon) \mid \sigma \in \Sigma\} \cup \{(q, \epsilon, X, q, \alpha) \mid X \rightarrow \alpha \in P\}$

- **Behauptung:**  $L(\mathcal{A}) = L(G)$ 
  - Wir führen den Beweis nur für Grammatiken in Chomsky-Normalform

- **Wir zeigen zuerst:**  $L(G) \subseteq L(\mathcal{A})$ 
  - Sei  $w \in L(G)$  und sei
 
$$S \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \dots \Rightarrow \gamma_n = w$$
 eine Linksableitung für  $w$  mit:
    - \*  $\gamma_i = u_i X_i \alpha_i$
    - \*  $z_i \stackrel{\text{def}}{=} \text{Suffix von } w \text{ mit } w = u_i z_i$
  - Dabei sind:
    - \*  $u_i, z_i \in \Sigma^*, X_i \in V, \alpha_i \in V^*$ ,  
für  $i < n$
    - \*  $u_n = w, X_n \alpha_n = \epsilon, z_n = \epsilon$

## Beweis (Forts.)

- $X_i$  ist also die am weitesten links stehende Variable der  $i$ -ten Satzform
- $u_i$  ist der String aus Terminalzeichen links davon
- $\alpha_i$  ist der String rechts davon, der nur aus Variablen besteht, da dies eine Linksableitung zu einer CNF-Grammatik ist

- Die im  $(i + 1)$ -ten Schritt angewendete Regel sei
  - $X_i \rightarrow Y_i Z_i$  oder
  - $X_i \rightarrow \sigma_i$
- Dabei sind:
  - $\sigma_i \in \Sigma$
  - $Y_i, Z_i \in V$

- Wir zeigen durch Induktion nach  $i$ , dass für alle  $i \geq 0$  gilt:
 
$$(q, w, S) \vdash_{\mathcal{A}}^* (q, z_i, X_i \alpha_i)$$

# Grammatik $\rightarrow$ Kellerautomat: Beweisdetails (2/3)

## Beweisdetails für $L(G) \subseteq L(\mathcal{A})$

- Ind.-Beh.:  $(q, w, S) \vdash_{\mathcal{A}}^* (q, z_i, X_i \alpha_i)$
  - $i = 0: \checkmark$  ( $z_0 = w, \alpha_0 = \epsilon$ )
  - Von  $i$  zu  $i + 1$ :
    - Nach Induktion gilt:
 
$$(q, w, S) \vdash_{\mathcal{A}}^* (q, z_i, X_i \alpha_i)$$
    - Wir unterscheiden nach der Art der im  $(i + 1)$ -ten Schritt verwendeten Regel
    - 1. Fall:  $X_i \rightarrow Y_i Z_i$ 
      - \* Dann gelten:
        - $z_{i+1} = z_i$
        - $X_{i+1} = Y_i$  und
        - $\alpha_{i+1} = Z_i \alpha_i$
      - \* Nach Definition von  $\mathcal{A}$  gilt dann:
 
$$(q, z_i, X_i \alpha_i) \vdash (q, z_i, Y_i Z_i \alpha_i) = (q, z_{i+1}, X_{i+1} \alpha_{i+1})$$
- ➡ Induktionsbehauptung

## Beweisdetails (Forts.)

- 2. Fall:  $X_i \rightarrow \sigma_i$ 
  - Da wir eine Linksableitung einer CNF-Grammatik haben, ist das erste Symbol von  $\alpha_i$  eine Variable, also  $X_{i+1}$  in der von uns gewählten Notation
  - Es gilt:  $\alpha_i = X_{i+1} \alpha_{i+1}$
  - Es folgt:
 
$$(q, z_i, X_i \alpha_i) \vdash (q, z_i, \sigma_i X_{i+1} \alpha_{i+1})$$
  - Da die Ableitung insgesamt  $w$  erzeugt, ist  $\sigma_i$  das erste Zeichen von  $z_i$  und es gilt  $z_i = \sigma_i z_{i+1}$
  - Dann folgt:  $(q, z_i, \sigma_i X_{i+1} \alpha_{i+1}) \vdash (q, z_{i+1}, X_{i+1} \alpha_{i+1})$
- ➡ Induktionsbehauptung
- Der 2. Fall findet insbesondere im letzten Ableitungsschritt Anwendung und führt damit zur Konfiguration  $(q, \epsilon, \epsilon)$
- ➡  $w \in L(\mathcal{A})$

# Grammatik → Kellerautomat: Beweisdetails (3/3)

## Beweis (Forts.)

- Zu zeigen:  $L(\mathcal{A}) \subseteq L(G)$
- Wir beweisen durch Induktion nach der Berechnungslänge  $n$ :
  - Für alle  $X \in V$  und  $w \in \Sigma^*$ :  
 wenn  $(q, w, X) \vdash^n (q, \epsilon, \epsilon)$ ,  
 dann  $X \Rightarrow^* w$
- $n = 1$ :
  - $(q, w, X) \vdash (q, \epsilon, \epsilon)$
  - ➔  $X = S$  und  $w = \epsilon$  und es gibt die Regel  $S \rightarrow \epsilon$  in  $G$   
 ☞ wegen CNF
  - ➔  $X \Rightarrow^* w$
- $n = 2$ :
  - $(q, w, X) \vdash (q, w, \sigma)$   
 $\vdash (q, \epsilon, \epsilon)$
  - ➔  $w = \sigma$  und es gibt die Regel  $X \rightarrow \sigma$  in  $G$
  - ➔  $X \Rightarrow^* w$

## Beweis (Forts.)

- $n + 1$ :  $(q, w, X) \vdash^{n+1} (q, \epsilon, \epsilon)$ 
  - Sei  $(q, w, X) \vdash (q, w, Z_1 Z_2)$  der erste Schritt der Berechnung, für gewisse  $Z_1, Z_2 \in V$
  - ➔  $(q, w, Z_1 Z_2) \vdash^n (q, \epsilon, \epsilon)$  und in dieser Berechnung werden  $Z_1$  und  $Z_2$  nach und nach vom Keller entfernt
  - ➔ Es gibt eine Zerlegung  $w = u_1 u_2$ , so dass  
 $(q, u_1 u_2, Z_1 Z_2) \vdash_{(Z_2)}^{m_1} (q, u_2, Z_2)$   
 $\vdash^{m_2} (q, \epsilon, \epsilon)$
  - Backstage-Lemma:
    - \*  $(q, u_1, Z_1) \vdash^{m_1} (q, \epsilon, \epsilon)$
  - Induktion:  $Z_1 \Rightarrow^* u_1$  und  $Z_2 \Rightarrow^* u_2$   
 ☞  $m_1, m_2 \leq n$
  - ➔  $X \Rightarrow Z_1 Z_2 \Rightarrow^* u_1 u_2 = w$
- Die Anwendung auf  $X = S$  liefert dann  $L(\mathcal{A}) \subseteq L(G)$

# Kellerautomat → Grammatik: Beweisdetails (1/2)

## Beweis von Satz 9.3 (Forts.)

- Wir zeigen zuerst durch Induktion nach  $n$ :
  - falls  $(p, w, \tau) \vdash_{\mathcal{A}}^n (p', \epsilon, \epsilon)$
  - so gilt:  $X_{p,\tau,p'} \Rightarrow^* w$
- $n = 1$ :
  - Dann gilt:
    - \*  $w = \epsilon$  und  $(p, \epsilon, \tau, p', \epsilon) \in \delta$  oder
    - \*  $w = \sigma$  und  $(p, \sigma, \tau, p', \epsilon) \in \delta$
  - Im ersten Fall enthält  $P$  die Regel  $X_{p,\tau,p'} \rightarrow \epsilon$  im zweiten Fall  $X_{p,\tau,p'} \rightarrow \sigma$
- $n > 1$ : Wir betrachten zuerst den Fall, dass der erste Schritt der Berechnung ein Zeichen  $\sigma$  liest, also:
 
$$(p, w, \tau) \vdash (q, u, \tau_1 \tau_2)$$

mit  $w = \sigma u$
- Dann gilt:  $(p, \sigma, \tau, q, \tau_1 \tau_2) \in \delta$
- Nach Konstruktion von  $G$  gibt es also für alle  $p_1$  und  $p'$  eine Regel
 
$$X_{p,\tau,p'} \rightarrow \sigma X_{q,\tau_1,p_1} X_{p_1,\tau_2,p'}$$

## Beweis (Forts.)

- Sei  $p_1$  der Zustand nach dem Entfernen von  $\tau_1$  vom Keller in der Berechnung
 
$$(q, u, \tau_1 \tau_2) \vdash^{n-1} (p', \epsilon, \epsilon)$$
- Seien  $u = u_1 u_2$ , so dass  $u_1$  bis zum Entfernen von  $\tau_1$  gelesen wird
- ➔ Es gibt  $i, j$  mit  $i + j = n - 1$ , so dass:
 
$$(p, w, \tau) \vdash (q, u_1 u_2, \tau_1 \tau_2) \vdash_{(\tau_2)}^i (p_1, u_2, \tau_2) \vdash^j (p', \epsilon, \epsilon)$$
- Mit Lemma 10.3 gelten also:
  - $(q, u_1, \tau_1) \vdash^i (p_1, \epsilon, \epsilon)$  und
  - $(p_1, u_2, \tau_2) \vdash^j (p', \epsilon, \epsilon)$
- Nach Induktion folgt:
  - $X_{q,\tau_1,p_1} \Rightarrow^* u_1$
  - $X_{p_1,\tau_2,p'} \Rightarrow^* u_2$
- ➔ 
$$X_{p,\tau,p'} \Rightarrow \sigma X_{q,\tau_1,p_1} X_{p_1,\tau_2,p'} \Rightarrow^* \sigma u_1 u_2 = w$$
- Der Fall, dass der erste Schritt ein  $\epsilon$ -Übergang ist, lässt sich analog beweisen

# Kellerautomat → Grammatik: Beweisdetails (2/2)

## Beweis von Satz 9.3 (Forts.)

- Wir zeigen jetzt durch Induktion nach der Ableitungslänge  $n$ :
  - falls  $X_{p,\tau,p'} \Rightarrow^n w$
  - so gilt:  $(p, w, \tau) \vdash_{\mathcal{A}}^* (p', \epsilon, \epsilon)$
- $n = 1$ : Die einzigen Regeln von  $G$ , die keine Variablen erzeugen, sind von der Form
  - $X_{p,\tau,p'} \rightarrow \alpha$ , mit  
 $(p, \alpha, \tau, p', \epsilon) \in \delta$   
 mit  $\alpha \in \Sigma \cup \{\epsilon\}$
- Die Behauptung folgt direkt
- $n > 1$ : In diesem Fall gibt es zwei verschiedene Typen des ersten Ableitungsschrittes

## Beweis (Forts.)

- Wir betrachten den ersten Fall:
 
$$X_{p,\tau,p'} \Rightarrow \alpha X_{q,\tau_1,p_1} X_{p_1,\tau_2,p'}$$
- Es gilt dann:
 
$$\alpha X_{q,\tau_1,p_1} X_{p_1,\tau_2,p'} \Rightarrow^{n-1} w$$
- Also gibt es  $u_1, u_2$  mit  $w = \alpha u_1 u_2$  und  $i, j$  mit  $i + j = n - 1$ , so dass gilt:
  - $X_{q,\tau_1,p_1} \Rightarrow^i u_1$
  - $X_{p_1,\tau_2,p'} \Rightarrow^j u_2$
- Nach Induktion folgt:
  - $(q, u_1, \tau_1) \vdash^* (p_1, \epsilon, \epsilon)$
  - $(p_1, u_2, \tau_2) \vdash^* (p', \epsilon, \epsilon)$
- Mit Lemma 10.3 gelten dann auch:
  - $(q, u_1 u_2, \tau_1 \tau_2) \vdash^* (p_1, u_2, \tau_2)$
  - $(p_1, u_2, \tau_2) \vdash^* (p', \epsilon, \epsilon)$
- Zusammen ergibt sich
 
$$(p, w, \tau) \vdash (q, u_1 u_2, \tau_1 \tau_2) \vdash_{(\tau_2)}^* (p', \epsilon, \epsilon)$$
- Die anderen Fällen sind analog

# Grundbegriffe der Theoretischen Informatik

Sommersemester 2016 - Thomas Schwentick

## Teil B: Kontextfreie Sprachen

### 10: Pumping-Lemma, Algorithmen und Abschlusseigenschaften

Version von: 24. Mai 2016 (12:02)



# Einleitung

- In diesem Kapitel werden wir sehen, welche der angenehmen Eigenschaften der regulären Sprachen auch für die kontextfreien Sprachen gelten, und welche nicht
- Methoden zum Nachweis, dass eine Sprache nicht kontextfrei ist:
  - Es gibt ein Pumping-Lemma für kontextfreie Sprachen, das nur wenig komplizierter als das für reguläre Sprachen ist
  - Zum Satz von Nerode korrespondierende Resultate haben wir aber nicht
- Algorithmen:
  - Einige algorithmische Probleme für kontextfreie Sprachen lassen sich effizient lösen
  - Andere gar (!) nicht
- Abschlusseigenschaften:
  - Die Klasse der kontextfreien Sprachen ist unter weniger Operationen abgeschlossen als die Klasse der regulären Sprachen
  - Die Klasse der deterministisch kontextfreien Sprachen hat andere Abschlusseigenschaften als beide Klassen

# Inhalt

## ▷ 10.1 Das Pumping-Lemma für kontextfreie Sprachen

10.2 Algorithmen für kontextfreie Sprachen

10.3 Abschlusseigenschaften der kontextfreien Sprachen

10.4 Deterministische Kellerautomaten

# Pumping-Lemma für kontextfreie Sprachen

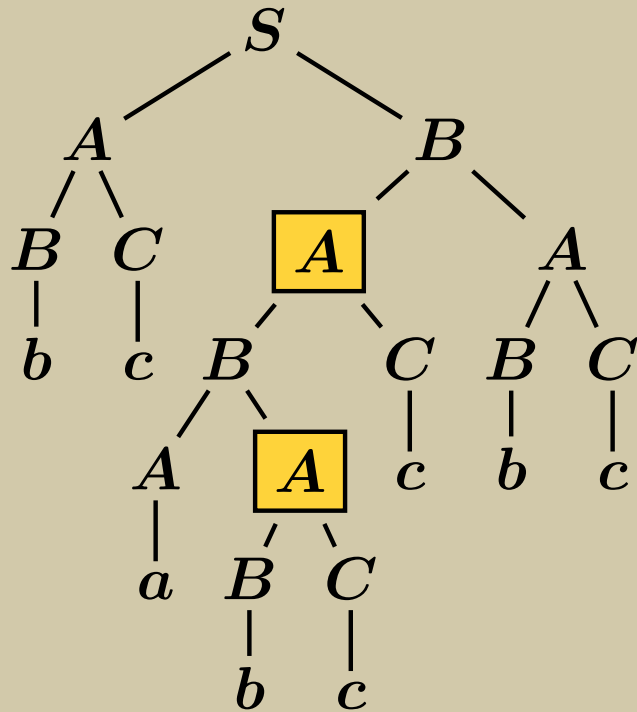
- Zur Erinnerung:
  - Das Pumping-Lemma für reguläre Sprachen beschreibt eine Abschlusseigenschaft, die jede reguläre Sprache hat
  - Es wird benutzt, um zu beweisen, dass eine gegebene Sprache **nicht** regulär ist
- Jetzt betrachten wir eine ähnliche Aussage für kontextfreie Sprachen
  - Der Beweis beruht darauf, dass „gleichartige Teile“ eines Ableitungsbaumes beliebig oft wiederholt werden können
  - Das ist ähnlich wie beim Pumping-Lemma für reguläre Sprachen, nur etwas komplizierter

# Pumping-Lemma: Vorbereitendes Beispiel

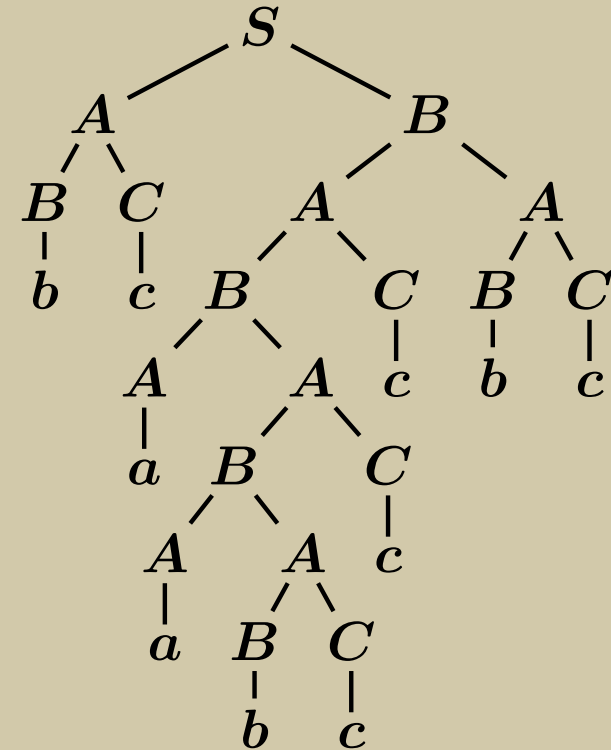
Beispiel: Grammatik

$$S \rightarrow AB \mid a$$
$$A \rightarrow BC \mid a$$
$$B \rightarrow AA \mid b$$
$$C \rightarrow CB \mid c$$

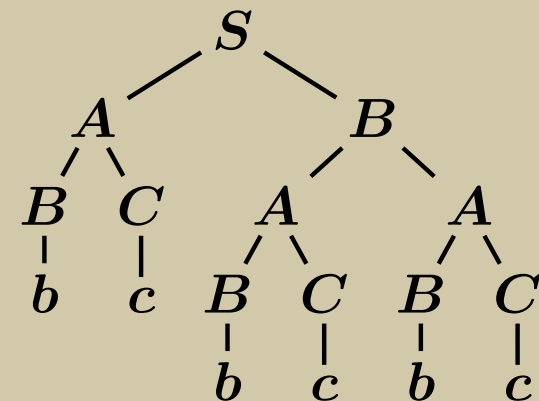
Beispiel: Ableitung



Beispiel: „Mittelteil“ wiederholen

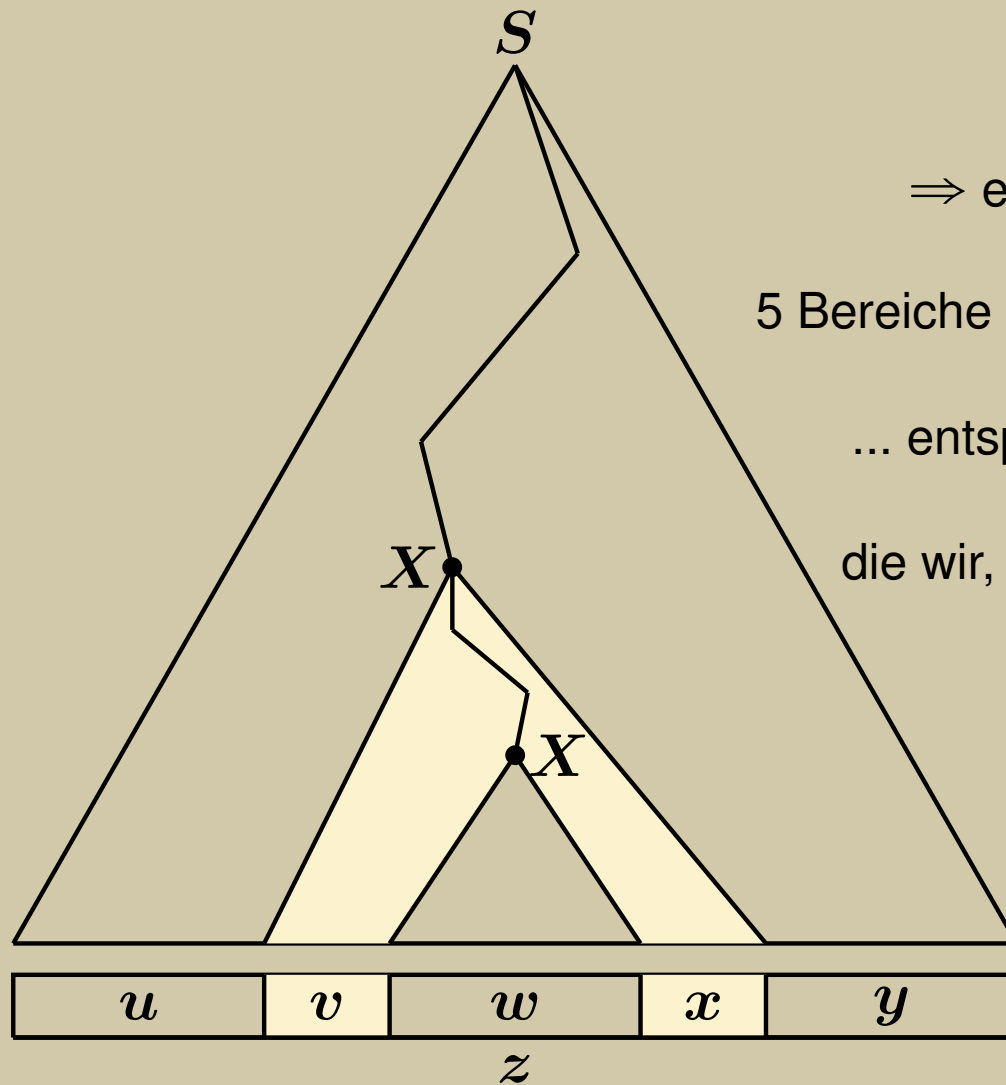


Beispiel: „Mittelteil“ löschen



# Pumping-Lemma: Illustration (1/2)

Beispiel: Ableitungsbaum  $T$  für  $z$



Weg mit Länge  $> |V|$

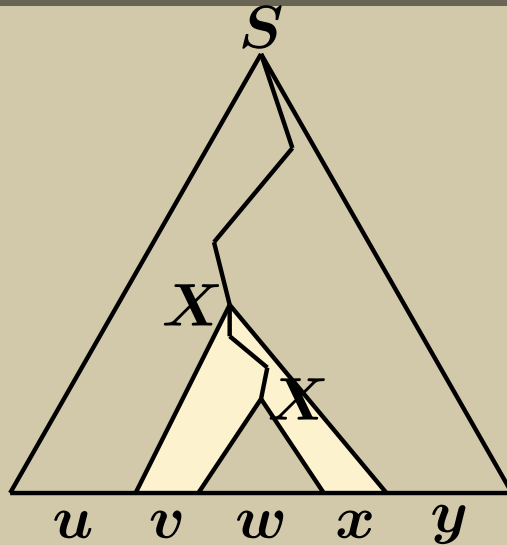
$\Rightarrow$  eine Variable  $X$  doppelt

5 Bereiche des Ableitungsbaums...

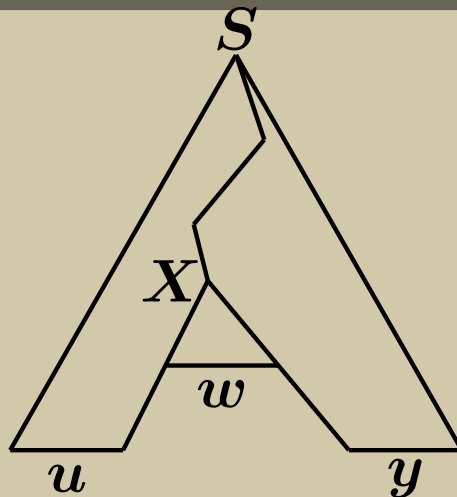
... entsprechen 5 Teilen von  $z$ ,  
die wir,  $u, v, w, x, y$  nennen

## Pumping-Lemma: Illustration (2/2)

Beispiel:  $T$

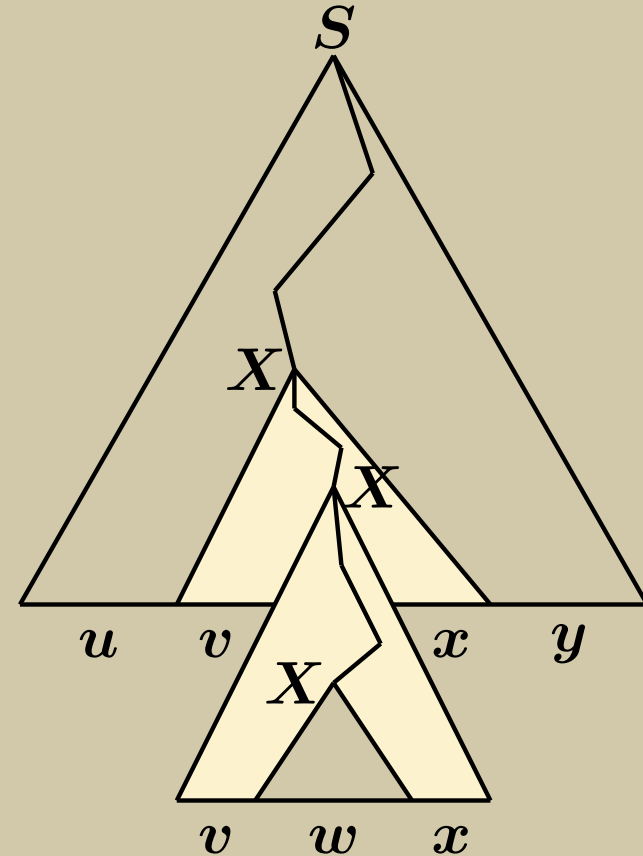


Beispiel:  $v$  und  $x$  entfernen



- Weglassen des „Mittelteils“ ergibt einen Ableitungsbaum für  $uwy$

Beispiel:  $v$  und  $x$  wiederholen



- Wiederholen des „Mittelteils“ ergibt einen Ableitungsbaum für  $uvvwxxy$

# Pumping-Lemma für kontextfreie Sprachen (1/2)

## Satz 10.1 (Pumping-Lemma)

- Ist  $L$  kontextfrei, so gibt es ein  $n \in \mathbb{N}$ , so dass sich jeder String  $z \in L$  mit  $|z| \geq n$  so in  $z = uvwxy$  zerlegen lässt, dass gelten:
  - (1)  $vx \neq \epsilon$ ,
  - (2)  $|vwx| \leq n$ ,
  - (3)  $uv^kwx^ky \in L$ , für alle  $k \geq 0$

## Beweisskizze

- Sei  $L$  eine kontextfreie Sprache
  - Sei  $G = (V, \Sigma, S, P)$  eine Grammatik für  $L$  in Chomsky-Normalform
  - Sei  $m \stackrel{\text{def}}{=} |V|$  die Anzahl der Variablen von  $G$
- Wir setzen  $n \stackrel{\text{def}}{=} 2^{m+1}$
- Sei  $z \in L$  beliebig mit  $|z| \geq n$
- Sei  $T$  ein Ableitungsbaum für  $z$

# Pumping-Lemma für kontextfreie Sprachen (2/2)

## Beweisskizze (Forts.)

- Da  $G$  in CNF ist, hat jeder innere Knoten von  $T$  höchstens zwei Kinder
  - ➔ Die Tiefe von  $T$  ist  $\geq m + 1$
- Sei  $W$  ein Weg maximaler Länge von der Wurzel von  $T$  zu einem Blatt von  $T$ 
  - ➔  $W$  enthält mindestens  $m + 1$  mit Variablen markierte Knoten
  - ➔ Unter den letzten  $m + 1$  dieser Knoten muss eine Variable  $X \in V$  doppelt vorkommen
  - ➔ Das aus dem oberen  $X$  abgeleitete Teilwort hat eine Länge  $\leq 2^{m+1}$ 
    - Und: Der obere  $X$ -Knoten hat 2 Kinder und erzeugt deshalb einen echt größeren String als der untere  $X$ -Knoten

- Also:  $S \Rightarrow^* uXy \Rightarrow^* uvXxy \Rightarrow^* uvwxy = z$  mit
  - $u, v, w, x, y \in \Sigma^*$
  - $v \neq \epsilon$  oder  $x \neq \epsilon$
  - $|vwx| \leq 2^{m+1} = n$

## Beweisskizze (Forts.)

- Idee: der am oberen  $X$  hängende Teilbaum kann an der Stelle des unteren  $X$  eingefügt werden und umgekehrt
  - Das Einfügen des oberen Teilbaums am unteren  $X$  kann wiederholt ausgeführt werden
- Für den formalen Beweis nutzen wir aus, dass gelten:
  - $X \Rightarrow^* vXx$  und
  - $X \Rightarrow^* w$
- Also gilt auch:
  - $S \Rightarrow^* uXy \Rightarrow^* uwy$  und
  - für jedes  $k \geq 1$ :  

$$S \Rightarrow^* uXy \Rightarrow^* uvXxy \Rightarrow^* \dots \Rightarrow^* uv^kXx^ky \Rightarrow^* uv^kwx^ky$$
- ➔  $uv^kwx^ky \in L$



# Pumping-Lemma: Beispiel

## Beispiel: Grammatik

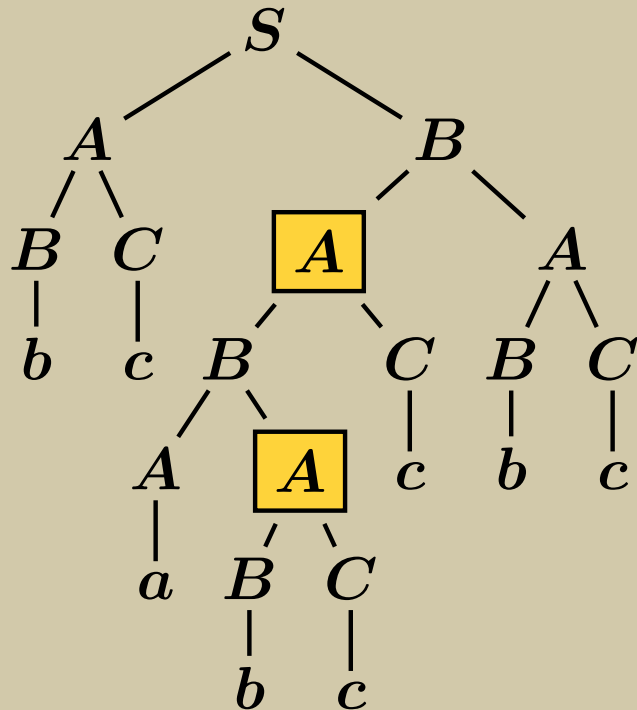
$$S \rightarrow AB \mid a$$

$$A \rightarrow BC \mid a$$

$$B \rightarrow AA \mid b$$

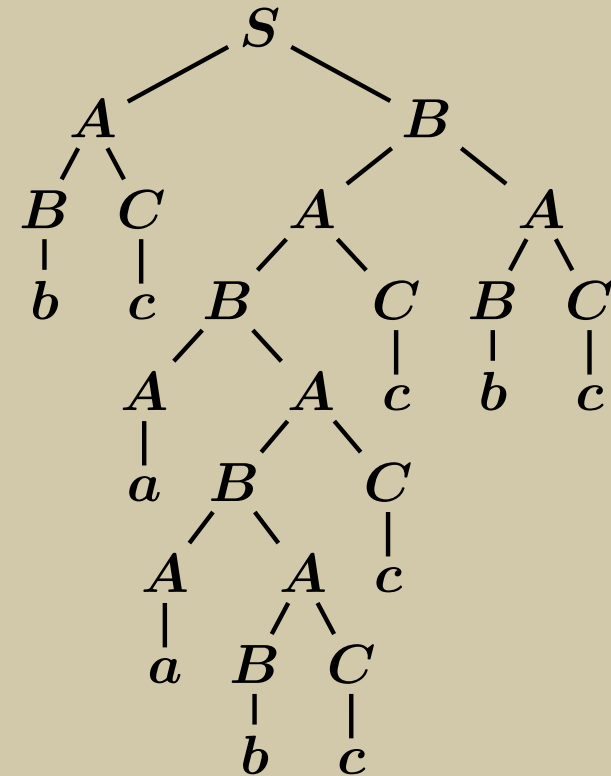
$$C \rightarrow CB \mid c$$

## Beispiel: Ableitung

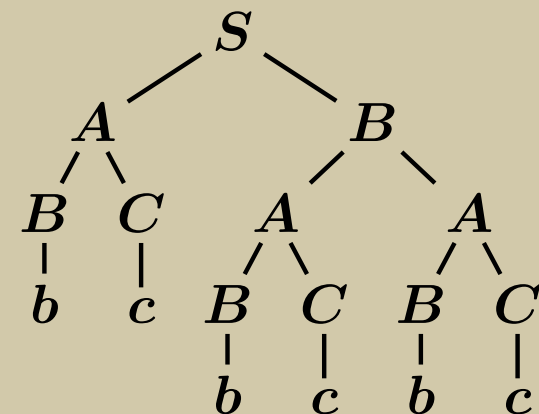


- $u = bc, w = bc, y = bc$
- $v = a, x = c$

## Beispiel: $v$ und $x$ wiederholen



## Beispiel: $v$ und $x$ löschen



# Pumping-Lemma: Anwendung (1/6)

- Wie beim regulären Pumping-Lemma betrachten wir eine Formulierung des Pumping-Lemmas, die sich besser zur Anwendung eignet
- Sie entsteht wieder einfach durch Kontraposition

## Korollar 10.2

- Sei  $L$  eine Sprache
- Angenommen, für jedes  $n > 0$  gibt es einen String  $z \in L$  mit  $|z| \geq n$  so dass für jede Zerlegung  $z = uvwxy$  mit
  - (1)  $vx \neq \epsilon$ ,
  - (2)  $|vwx| \leq n$ ,ein  $k \geq 0$  existiert mit  $uv^kwx^ky \notin L$
- Dann ist  $L$  nicht kontextfrei

- Bei der Anwendung des Pumping-Lemmas muss wieder darauf geachtet werden, an welchen Stellen im Beweis eine Wahl besteht und an welchen Stellen nicht:
  - $n$ : keine Wahl, das folgende Argument muss für beliebige  $n$  funktionieren
  - $z$  kann in Abhängigkeit von  $n$  frei in  $L$  gewählt werden
  - Zerlegung  $z$  in  $uvwxy$ : hier besteht keine Wahl, das Argument muss für beliebige Zerlegungen gelten, die (1) und (2) erfüllen
  - $k$  kann frei gewählt werden
- Zu beachten:
  - Beim regulären Pumping-Lemma war  $uv$  immer ein Präfix des Strings  $w$
  - Beim kontextfreien Pumping-Lemma kann sich  $vwx$  irgendwo in  $z$  befinden

# Pumping-Lemma: Anwendung (2/6)

## Korollar 10.2

- Sei  $L$  eine Sprache
- Angenommen, für jedes  $n > 0$  gibt es einen String  $z \in L$  mit  $|z| \geq n$  so dass für jede Zerlegung  $z = uvwxy$  mit
  - (1)  $vx \neq \epsilon$ ,
  - (2)  $|vwx| \leq n$ ,ein  $k \geq 0$  existiert mit  $uv^kwx^ky \notin L$
- Dann ist  $L$  nicht kontextfrei

## Proposition 10.3

- Die beiden folgenden Sprachen sind nicht kontextfrei:
  - (a)  $L_{abc} = \{a^m b^m c^m \mid m \geq 1\}$
  - (b)  $L_{\text{doppel}} = \{ww \mid w \in \{a, b\}^*\}$

## Beweis für Proposition 10.3 (a)

- Sei  $n$  beliebig
- Wähle  $z = a^n b^n c^n$
- Sei  $uvwxy$  eine Zerlegung von  $z$  mit  $u, v, w, x, y \in \{a, b, c\}^*$ , die (1) und (2) erfüllt
- Wegen (2) kann  $vx$  nicht sowohl  $a$  als auch  $c$  enthalten
  - ➔  $\#_a(uwy) = \#_a(z)$  oder  $\#_c(uwy) = \#_c(z)$
  - ➔  $uwy \notin L_{abc}$ , da in  $uwy$  zumindest ein Zeichen weniger als  $n$  mal vorkommt, aber  $a$  oder  $c$  noch  $n$  mal vorkommen
  - ➔  $L_{abc}$  nicht kontextfrei

# Pumping-Lemma: Anwendung (3/6)


## Korollar 10.2

- Sei  $L$  eine Sprache
- Angenommen, für jedes  $n > 0$  gibt es einen String  $z \in L$  mit  $|z| \geq n$  so dass für jede Zerlegung  $z = uvwxy$  mit
  - (1)  $vx \neq \epsilon$ ,
  - (2)  $|vwx| \leq n$ ,ein  $k \geq 0$  existiert mit
$$uv^kwx^ky \notin L$$
- Dann ist  $L$  nicht kontextfrei

## Proposition 10.3

- Die beiden folgenden Sprachen sind nicht kontextfrei:
  - (a)  $L_{abc} = \{a^m b^m c^m \mid m \geq 1\}$
  - (b)  $L_{\text{doppel}} = \{ww \mid w \in \{a, b\}^*\}$

## Beweis für Proposition 10.3 (b)

- Sei  $n$  beliebig
  - Wähle  $z = a^n b^n a^n b^n$  (4 „Blöcke“)
  - Sei  $uvwxy$  eine Zerlegung von  $z$  mit  $u, v, w, x, y \in \{a, b\}^*$ , die (1) und (2) erfüllt
  - Klar: falls  $|vx|$  ungerade ist, ist  $|uwy|$  auch ungerade, und deshalb  $uwy \notin L_{\text{doppel}}$
-  Im Rest des Beweises sei  $|vx|$  also gerade

## Pumping-Lemma: Anwendung (4/6)

### Korollar 10.2

- Sei  $L$  eine Sprache
- Angenommen, für jedes  $n > 0$  gibt es einen String  $z \in L$  mit  $|z| \geq n$  so dass für jede Zerlegung  $z = uvwxy$  mit
  - (1)  $vx \neq \epsilon$ ,
  - (2)  $|vwx| \leq n$ ,  
ein  $k \geq 0$  existiert mit
$$uv^kwx^ky \notin L$$
- Dann ist  $L$  nicht kontextfrei

### Proposition 10.3

- Die beiden folgenden Sprachen sind nicht kontextfrei:
  - (a)  $L_{abc} = \{a^m b^m c^m \mid m \geq 1\}$
  - (b)  $L_{\text{doppel}} = \{ww \mid w \in \{a, b\}^*\}$

### Beweis für Proposition 10.3 (b) (Forts.)

- Wir unterscheiden vier Fälle:
  - (1)  $vx$  enthält  $a$ 's aus dem ersten Block  
☞ möglicherweise aber auch andere Zeichen
  - (2)  $vx$  enthält  $b$ 's aus dem zweiten Block, aber keine  $a$ 's aus dem ersten Block  
☞ möglicherweise aber noch andere Zeichen
  - (3)  $vx$  enthält  $a$ 's aus dem dritten Block aber keine Zeichen aus den ersten zwei Blöcken  
☞ vielleicht aber Zeichen aus dem vierten Block
  - (4)  $vx$  enthält nur  $b$ 's aus dem vierten Block aber keine Zeichen aus anderen Blöcken

- In allen vier Fällen zeigen wir:
$$1^{\text{st}}(uwy) \neq 2^{\text{nd}}(uwy)$$
und damit  $uwy \notin L_{\text{doppel}}$   
✎ Zur Erinnerung:  $1^{\text{st}}(w)$  bezeichnet die erste Hälfte eines Strings  $w$  und  $2^{\text{nd}}(w)$  bezeichnet die zweite Hälfte

# Pumping-Lemma: Anwendung (5/6)

## Korollar 10.2

- Sei  $L$  eine Sprache
- Angenommen, für jedes  $n > 0$  gibt es einen String  $z \in L$  mit  $|z| \geq n$  so dass für jede Zerlegung  $z = uvwxy$  mit
  - (1)  $vx \neq \epsilon$ ,
  - (2)  $|vwx| \leq n$ ,ein  $k \geq 0$  existiert mit
$$uv^kwx^ky \notin L$$
- Dann ist  $L$  nicht kontextfrei

## Proposition 10.3

- Die beiden folgenden Sprachen sind nicht kontextfrei:
  - (a)  $L_{abc} = \{a^m b^m c^m \mid m \geq 1\}$
  - (b)  $L_{\text{doppel}} = \{ww \mid w \in \{a, b\}^*\}$

## Beweis für Proposition 10.3 (b)

- Zur Erinnerung:  $z = a^n b^n a^n b^n$  (4 „Blöcke“)


- (1)  $vx$  enthält  $a$ 's aus dem ersten Block
  - Möglicherweise enthält  $vx$  auch Zeichen aus dem zweiten Block
  - Da  $|vwx| \leq n$  enthält  $vx$  keine Zeichen aus den letzten beiden Blöcken
  - ➔  $uwy$  ist von der Form  $a^i b^j a^n b^n$  mit
    - \*  $i < n$  und  $j \leq n$
  - Da  $|vx| \leq n$  gilt:  $3n \leq |uwy| < 4n$
  - ➔ Das letzte Zeichen von  $1^{\text{st}}(uwy)$  ist ein  $a$  (aus dem dritten Block), aber das letzte Zeichen von  $2^{\text{nd}}(uwy)$  ein  $b$
  - ➔  $uwy \notin L_{\text{doppel}}$

## Pumping-Lemma: Anwendung (6/6)

### Beweis für Proposition 10.3 (b) (Forts.)

- Zur Erinnerung:  $z = a^n b^n a^n b^n$  (4 „Blöcke“)

(2)  $vx$  enthält  $b$ 's aus dem zweiten Block, aber keine  $a$ 's aus dem ersten Block

- ➔ Da  $|vwx| \leq n$  enthält  $vx$  keine  $b$ 's aus dem vierten Block
- ➔  $uwy$  ist von der Form  $a^n b^i a^j b^n$  mit
  - \*  $i < n$  und  $j \leq n$  und
  - \*  $i + j \geq n$    $|vwx| \leq n$
- ➔  $2^{\text{nd}}(uwy)$  endet mit einem Block der Form  $b^n$ , aber  $1^{\text{st}}(uwy)$  enthält weniger als  $n$   $b$ 's
- ➔  $uwy \notin L_{\text{doppel}}$

### Beweis für Proposition 10.3 (b) (Forts.)

(3)  $vx$  enthält  $a$ 's aus dem dritten Block aber keine Zeichen aus den ersten zwei Blöcken

- ➔  $uwy = a^n b^n a^i b^j$
- ➔  $a^n$ -Block in  $1^{\text{st}}(uwy)$ , aber nicht in  $2^{\text{nd}}(uwy)$
- ➔  $uwy \notin L_{\text{doppel}}$

(4)  $vx$  enthält nur  $b$ 's aus dem vierten Block aber keine Zeichen aus anderen Blöcken

- ➔  $uwy = a^n b^n a^n b^i$
- ➔  $1^{\text{st}}(uwy)$  beginnt mit  $a$ ,  $2^{\text{nd}}(uwy)$  mit  $b$
- ➔  $uwy \notin L_{\text{doppel}}$

# Inhalt

10.1 Das Pumping-Lemma für kontextfreie Sprachen

## **10.2 Algorithmen für kontextfreie Sprachen**

### ▷ **10.2.1 Umwandlungsalgorithmen**

10.2.2 Analyse-Algorithmen für kontextfreie Sprachen

10.3 Abschlusseigenschaften der kontextfreien Sprachen

10.4 Deterministische Kellerautomaten



# Kontextfreie Sprachen: Umwandlungsalgorithmen

- Die folgenden Umwandlungen sind in linearer Zeit möglich und erzeugen Objekte linearer Größe:
  - kontextfreie Grammatik  $\rightarrow$  Kellerautomat
  - Kellerautomat mit leerem Keller  $\rightarrow$  Kellerautomat mit akzeptierenden Zuständen
  - Kellerautomat mit akzeptierenden Zuständen  $\rightarrow$  Kellerautomat mit leerem Keller
- Die Umwandlung eines Kellerautomaten  $\mathcal{A}$  in eine Grammatik ist in Zeit  $\mathcal{O}(|\mathcal{A}|^4)$  möglich
  - Dabei bezeichnet  $|\mathcal{A}|$  die Größe der Transitionsfunktion, wobei jede Transition  $1 +$  Länge des neuen Kellerstrings beiträgt

- Die Umwandlung einer kontextfreien Grammatik in eine CNF-Grammatik ist in Zeit  $\mathcal{O}(|G|^2)$  möglich  
(wir hatten nur:  $\mathcal{O}(|G|^4)$ )
- Bei der Umwandlung in Greibach-Normalform ist Vorsicht geboten:
  - Der Original-Algorithmus von Greibach kann im schlimmsten Fall eine Grammatik exponentieller Größe erzeugen
  - Es gibt Algorithmen, die immer eine Grammatik in GNF der Größe  $\mathcal{O}(|G|^3)$  erzeugen  
[Rosenkrantz 67; Blum, Koch 98]

# Inhalt

10.1 Das Pumping-Lemma für kontextfreie Sprachen

## **10.2 Algorithmen für kontextfreie Sprachen**

10.2.1 Umwandlungsalgorithmen

### ▷ **10.2.2 Analyse-Algorithmen für kontextfreie Sprachen**

10.3 Abschlusseigenschaften der kontextfreien Sprachen

10.4 Deterministische Kellerautomaten

# Leerheitstests für kontextfreie Sprachen

Def.: Leerheitsproblem für kontextfreie Grammatiken

**Gegeben:** Grammatik  $G$

**Frage:** Ist  $L(G) \neq \emptyset$ ?

Def.: Leerheitsproblem für Kellerautomaten

**Gegeben:** Kellerautomat  $\mathcal{A}$

**Frage:** Ist  $L(\mathcal{A}) \neq \emptyset$ ?

## Satz 10.4

- Zu einer gegebenen kontextfreien Grammatik  $G$  lässt sich in linearer Zeit  $\mathcal{O}(|G|)$  entscheiden, ob  $L(G) \neq \emptyset$  gilt

## Beweisidee

- $L(G) \neq \emptyset$  gilt genau dann, wenn das Startsymbol  $S$  erzeugend ist
- Das lässt sich bei geschickter Implementierung in Zeit  $\mathcal{O}(|G|)$  testen

[Beeri, Bernstein 79]

- Leerheitstest für Kellerautomaten:
  - Wandle Kellerautomat in Grammatik um, dann Leerheitstest für Grammatik
  - Ein effizienterer „direkter“ Algorithmus ist (mir) nicht bekannt

# Endlichkeitstest für kontextfreie Sprachen

Def.: Endlichkeitsproblem für KFGs

**Gegeben:** Grammatik  $G$

**Frage:** Ist  $|L(G)| < \infty$ ?

Satz 10.5

- Das Endlichkeitsproblem für kontextfreie Grammatiken in CNF kann in linearer Zeit entschieden werden

Beweis

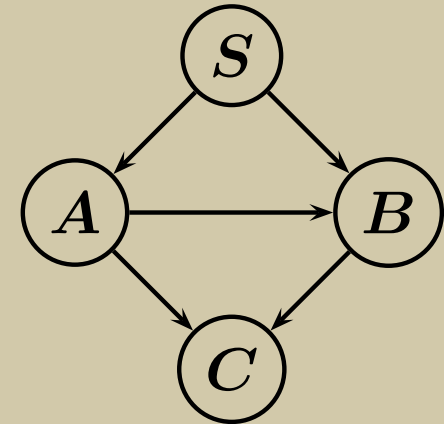
- Da  $G$  in CNF ist, hat  $G$  insbesondere keine nutzlosen Symbole
- Wir definieren zu  $G$  einen gerichteten Graphen  $H(G)$  (wie in CNF1):
  - Die Knoten von  $H(G)$  sind die Variablen von  $G$
  - Ist  $X \rightarrow YZ$  eine Regel von  $G$ , so hat  $H(G)$  die Kanten  $(X, Y)$  und  $(X, Z)$
- Behauptung:**  $L(G)$  ist genau dann unendlich, wenn  $H(G)$  einen gerichteten Kreis enthält

Beispiel

$G_1$ :

$$\begin{array}{l|l} S \rightarrow AB & \\ A \rightarrow BC & a \\ B \rightarrow CC & b \\ C \rightarrow a & \end{array}$$

$H(G_1)$ :

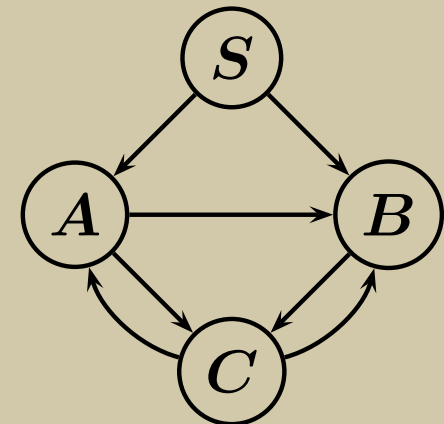


Beispiel

$G_2$ :


$$\begin{array}{l|l} S \rightarrow AB & \\ A \rightarrow BC & a \\ B \rightarrow CC & b \\ C \rightarrow AB & a \end{array}$$


$H(G_2)$ :



 Beweisdetails im Anhang

# Andere Tests für kontextfreie Sprachen

- Es gibt natürlich noch weitere Algorithmen für kontextfreie Grammatiken
- Es existieren aber auch Probleme, für die es keine Algorithmen gibt!
  - Die folgenden algorithmischen Probleme für kontextfreie Grammatiken können nicht von Algorithmen gelöst werden:
    - (1) Ist  $L(G)$  eindeutig oder inhärent mehrdeutig?
    - (2) Ist  $G$  eindeutig?
    - (3) Ist  $L(G)$  deterministisch kontextfrei?  siehe 10.3
    - (4) Ist  $L(G)$  regulär?
    - (5) Ist  $L(G_1) \cap L(G_2) = \emptyset$ ?
    - (6) Ist  $L(G_1) \cap L(G_2)$  kontextfrei?
    - (7) Ist  $L(G_1) \subseteq L(G_2)$ ?
    - (8) Ist  $L(G_1) = L(G_2)$ ?
    - (9) Ist  $L(G) = \Sigma^*$ ?

 Die genaue Bedeutung von „können nicht von Algorithmen gelöst werden“ werden wir in Teil C der Vorlesung kennen lernen

- Dort werden die Aussagen dann auch bewiesen

# Inhalt

10.1 Das Pumping-Lemma für kontextfreie Sprachen

10.2 Algorithmen für kontextfreie Sprachen

▷ **10.3 Abschlusseigenschaften der kontextfreien Sprachen**

10.4 Deterministische Kellerautomaten

# Abschlusseigenschaften der kontextfreien Sprachen

- Die Klasse der kontextfreien Sprachen ist unter vielen Operationen abgeschlossen
  - ...aber nicht unter ganz so vielen wie die regulären Sprachen

## Satz 10.6

- Die Klasse der kontextfreien Sprachen ist abgeschlossen unter
  - (a) Vereinigung,
  - (b) Konkatenation,
  - (c) dem  $*$ -Operator und
  - (d) dem  $+$ -Operator,

## Beweis

- Seien  $L_1$  und  $L_2$  kontextfreie Sprachen mit Grammatiken
  - $G_1 = (V_1, \Sigma, S_1, P_1)$  und
  - $G_2 = (V_2, \Sigma, S_2, P_2)$mit  $V_1 \cap V_2 = \emptyset$
- Dann können wir Grammatiken konstruieren für:
  - (a)  $L_1 \cup L_2$ : durch Vereinigung der beiden Grammatiken und Hinzunahme einer neuen Startvariablen  $S$  mit  $S \rightarrow S_1 \mid S_2$
  - (b)  $L_1 L_2$ : durch Vereinigung der beiden Grammatiken und Hinzunahme einer neuen Startvariablen  $S$  mit  $S \rightarrow S_1 S_2$
  - (c)  $L_1^*$ : durch Hinzunahme einer neuen Startvariablen  $S$  mit  $S \rightarrow S_1 S \mid \epsilon$
  - (d)  $L_1^+$ : durch Hinzunahme einer neuen Startvariablen  $S$  mit  $S \rightarrow S_1 S \mid S_1$

# Weitere Abschlusseigenschaften

## Satz 10.7

- (a) Ist  $L$  kontextfrei, so auch  $\{w^R \mid w \in L\}$
- (b) Ist  $L \subseteq \Sigma^*$  kontextfrei,  $h : \Sigma^* \rightarrow \Gamma^*$  ein Homomorphismus, so ist auch  $h(L)$  kontextfrei
- (c) Ist  $L \subseteq \Sigma^*$  kontextfrei,  $h : \Gamma^* \rightarrow \Sigma^*$  ein Homomorphismus, so ist auch  $h^{-1}(L)$  kontextfrei

## Beweisidee

- (a) Idee: Drehe die Regeln um
  - Sei  $G$  CNF-Grammatik für  $L$
  - Ersetze  $X \rightarrow YZ$  jeweils durch
$$X \rightarrow ZY$$
- (b) Ersetze in der Grammatik für  $L$  jedes Vorkommen eines Terminalsymbols  $\sigma \in \Sigma$  durch  $h(\sigma)$
- (c) Konstruktion eines Kellerautomaten  $\mathcal{B}$  für  $h^{-1}(L)$ :
  - ✎ analog zum Beweis für endliche Automaten
  - Sei  $\mathcal{A}$  Kellerautomat für  $L$
  - Wenn  $\mathcal{B}$  das Zeichen  $\sigma$  liest, simuliert er das Verhalten von  $\mathcal{A}$  bei Eingabe  $h(\sigma)$

→ Neue Zustände,  $\epsilon$ -Übergänge müssen beachtet werden



# Fehlende Abschlusseigenschaften

## Satz 10.8

- Die Klasse der kontextfreien Sprachen ist **nicht** abgeschlossen unter
  - (a) Durchschnitt,
  - (b) Komplement,
  - (c) Mengendifferenz

## Beweis

- (a) •  $L_1 \stackrel{\text{def}}{=} \{a^n b^n c^m \mid n, m \geq 1\}$   
•  $L_2 \stackrel{\text{def}}{=} \{a^m b^n c^n \mid n, m \geq 1\}$   
•  $L_1$  und  $L_2$  sind kontextfrei  
•  $L_1 \cap L_2 = L_{abc} = \{a^n b^n c^n \mid n \geq 1\}$   
ist **nicht** kontextfrei  
(Proposition 12.3 (a))
- (b) Sonst ließe sich der Durchschnitt durch Kombination von Vereinigung und Komplement ausdrücken
- (c) Sonst ließe sich das Komplement darstellen als:  $\Sigma^* - L$

- Unter Durchschnittsbildung sind die kontextfreien Sprachen also nicht abgeschlossen
  - Insbesondere gibt es also keine Produktautomatenkonstruktion für zwei Kellerautomaten

👉 Warum eigentlich?

- Es gilt aber eine schwächere Abschlusseigenschaft:

## Satz 10.9

- Ist  $L_1$  kontextfrei und  $L_2$  regulär, so ist  $L_1 \cap L_2$  kontextfrei

## Beweisidee

- Sei  $\mathcal{A}_1$  ein Kellerautomat für  $L_1$
- Sei  $\mathcal{A}_2$  ein DFA für  $L_2$
- $\mathcal{B}$  sei der „Produktautomat“ von  $\mathcal{A}_1$  und  $\mathcal{A}_2$
- $\mathcal{A}_2$  wirkt sich nur auf die Zustände aus, nicht auf den Kellerinhalt

# Inhalt


10.1 Das Pumping-Lemma für kontextfreie Sprachen

10.2 Algorithmen für kontextfreie Sprachen

10.3 Abschlusseigenschaften der kontextfreien Sprachen

▷ **10.4 Deterministische Kellerautomaten**

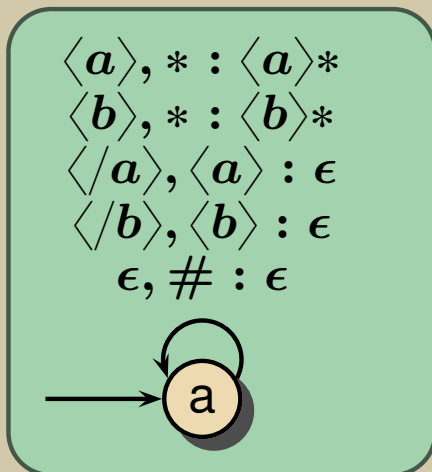
# Deterministische Kellerautomaten: Motivation

- Ein Nachteil von PDAs:
  - Algorithmisch zu entscheiden, ob ein gegebener PDA  $\mathcal{A}$  für einen gegebenen String  $w$  eine akzeptierende Berechnung hat, ist nicht so leicht
  - Es kann exponentiell viele verschiedene Berechnungen geben...
  - Der nahe liegende Algorithmus verwendet Backtracking und kann zu exponentiellem Aufwand führen  Kapitel 11
- **Frage:** Gibt es zu jedem PDA einen äquivalenten deterministischen Kellerautomaten (DPDA)?
  - Dazu müssen wir zuerst definieren, wann Kellerautomaten deterministisch sind
- PDAs haben zwei Quellen für Nichtdeterminismus:
  - Für einen Zustand  $p$ , ein gelesenes Eingabezeichen  $\sigma$ , und ein Kellersymbol  $\tau$  kann es in  $\delta$  mehrere Transitionen geben:
    - $(p, \sigma, \tau, q_1, w_1)$
    - $(p, \sigma, \tau, q_2, w_2)$
  - ➡ Klar: in deterministischen Kellerautomaten darf es für jede Kombination von  $p, \sigma, \tau$  nur eine Transition  $(p, \sigma, \tau, q, w)$  in  $\delta$  geben
- PDAs können außerdem  $\epsilon$ -Übergänge haben
  - Wir erlauben  $\epsilon$ -Übergänge auch in DPDAs
  - Aber: es darf keine zwei verschiedenen Transitionen
    - \*  $(p, \epsilon, \tau, q_1, w_1)$
    - \*  $(p, \epsilon, \tau, q_2, w_2)$geben
  - Und: wenn es eine Transition  $(p, \epsilon, \tau, q, w)$  gibt, so darf es für kein  $\sigma$  eine Transition  $(p, \sigma, \tau, q', w')$  geben

# Deterministische Kellerautomaten: Beispiele

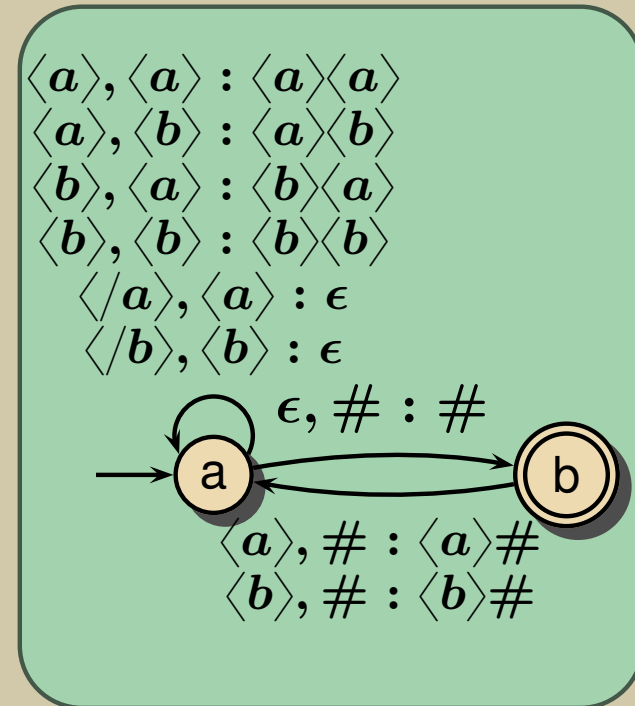
## Beispiel

- Zur Erinnerung: der folgende Kellerautomat entscheidet die Sprache der wohlgeformten Klammerausdrücke über  $\{\langle a \rangle, \langle /a \rangle, \langle b \rangle, \langle /b \rangle\}$ :



- Der Automat akzeptiert mit leerem Keller
- Aber der Automat ist nicht deterministisch:
  - $(a, \langle b \rangle, \#, a, \langle b \rangle \#) \in \delta$  und
  - $(a, \epsilon, \#, a, \epsilon) \in \delta$
- Ein kritischer String:
 
$$\langle b \rangle \langle a \rangle \langle /a \rangle \langle /b \rangle \langle b \rangle \langle /b \rangle$$
  - Wie soll es nach Lesen von  $\langle b \rangle \langle a \rangle \langle /a \rangle \langle /b \rangle$  weitergehen?

## Beispiel



- Dies ist ein deterministischer Kellerautomat mit akzeptierenden Zuständen für dieselbe Sprache
- Nach Lesen von  $\langle b \rangle \langle a \rangle \langle /a \rangle \langle /b \rangle$  geht er in einen akzeptierenden Zustand
- Wenn der String noch nicht zu Ende ist, kann er dann aber auch noch weitere Zeichen lesen

# Deterministische Kellerautomaten: Definition

- Wir verwenden in der Definition von DPDAs die folgende Notation:

$$- \underline{\delta(p, \sigma, \tau)} \stackrel{\text{def}}{=} \{(q, z) \mid (p, \sigma, \tau, q, z) \in \delta\}$$

$$- \underline{\delta(p, \epsilon, \tau)} \stackrel{\text{def}}{=} \{(q, z) \mid (p, \epsilon, \tau, q, z) \in \delta\}$$

## Definition

- Ein Kellerautomat  $\mathcal{A} = (Q, \Sigma, \Gamma, \delta, s, \tau_0, F)$  heißt deterministisch, falls für alle  $p \in Q, \sigma \in \Sigma, \tau \in \Gamma$  gilt:
  - $|\delta(p, \sigma, \tau)| + |\delta(p, \epsilon, \tau)| \leq 1$
- Eine Sprache heißt deterministisch kontextfrei, wenn sie von einem deterministischen Kellerautomaten entschieden wird

# Deterministisch kontextfreie Sprachen: Akzeptiermechanismen

## Satz 10.10

- (a) Zu jedem DPDA  $\mathcal{A}$ , der mit leerem Keller akzeptiert, gibt es es einen DPDA  $\mathcal{B}$ , der mit akzeptierenden Zuständen akzeptiert und  $L(\mathcal{B}) = L(\mathcal{A})$  erfüllt
- (b) Die Umkehrung gilt nicht

## Beweisidee

- (a) Die Konstruktion aus Satz 9.1 (a) erzeugt aus einem DPDA wieder einen DPDA
  - (b) Von einem DPDA mit leerem Keller akzeptierte Sprachen sind **präfixfrei**:
    - Sie enthalten keine Strings  $u$  und  $v$ , für die  $u$  echtes Präfix von  $v$  ist
      - Denn: nach dem Lesen von  $u$  ist der Keller leer, der Automat kann den Rest von  $v$  also gar nicht mehr lesen
- 
- DPDAs mit leerem Keller gibt es also nicht einmal für jede reguläre Sprache
  - Es gilt aber:
    - wenn  $L$  präfixfrei ist und einen DPDA mit akzeptierenden Zuständen hat
    - dann hat  $L$  auch einen DPDA mit leerem Keller

# Det. kontextfreie Sprachen: Komplementabschluss (1/2)

## Satz 10.11

- Die Klasse der deterministisch kontextfreien Sprachen ist abgeschlossen unter Komplementbildung

## Beweisidee

- Sei  $\mathcal{A}$  ein deterministischer Kellerautomat für Sprache  $L$
- Grundidee: Der Automat  $\mathcal{B}$  für  $\overline{L}$  entstehe durch Vertauschung akzeptierender und ablehnender Zustände in  $\mathcal{A}$


## Beweisidee (Forts.)

### • Komplikationen:


- (1)  $\mathcal{A}$  könnte anhalten, ohne die Eingabe ganz zu lesen,
  - (1a) weil  $\mathcal{A}$  in einer Konfiguration keine Transition hat,
  - (1b) weil vorzeitig eine Konfiguration mit leerem Keller erreicht wird, oder
  - (1c) weil nur noch eine (unendliche) Folge von  $\epsilon$ -Übergängen möglich ist➡ In all diesen Fällen muss  $\mathcal{B}$  die Eingabe akzeptieren
- (2)  $\mathcal{A}$  könnte akzeptieren mit einer Berechnung der Art:
$$(s, w, \tau_0) \vdash_{\mathcal{A}}^* (q_1, \epsilon, \alpha) \vdash_{\mathcal{A}}^* (q_2, \epsilon, \beta)$$
mit  $q_1 \in F$  und  $q_2 \notin F$  (oder umgekehrt)  
➡ In diesem Fall darf  $\mathcal{B}$  **nicht** akzeptieren, obwohl am Ende der Zustand  $q_2 \notin F$  erreicht wird!

# Det. kontextfreie Sprachen: Komplementabschluss (2/2)

## Beweisidee (Forts.)

- (1) Für das Problem von Berechnungen, die nicht die ganze Eingabe lesen, hat  $\mathcal{B}$  einen zusätzlichen Zustand  $q_+$ , in dem der Rest der Eingabe gelesen und dann akzeptiert wird
  - Die Frage ist nur: wie erkennt  $\mathcal{B}$ , dass er in den Zustand  $q_+$  übergehen muss?
- (1a) Hat  $\mathcal{A}$  für Zustand  $p$ , Eingabezeichen  $\sigma$  und Kellersymbol  $\tau$  keine Transition, so geht  $\mathcal{B}$  in den Zustand  $q_+$  über
- (1b) Um Konfigurationen mit leerem Keller zu vermeiden (und zu erkennen), verwendet  $\mathcal{B}$  ein neues unterstes Kellersymbol
  -  Ähnlich der Umwandlung von Kellerautomaten in Satz 9.1 (a) und 9.1 (b)
- (1c) Die Menge der Paare  $(p, \tau)$ , die zu unendlichen Folgen von  $\epsilon$ -Transitionen führen, lässt sich berechnen
  - \*  $\mathcal{B}$  geht für diese Paare in  $q_+$  über

## Beweisidee (Forts.)

- (2)  $\mathcal{B}$  merkt sich immer, ob seit der letzten Nicht- $\epsilon$ -Transition schon ein akzeptierender Zustand von  $\mathcal{A}$  gesehen wurde
- Mit diesen Ideen lässt sich ein korrekter Automat für das Komplement von  $L(\mathcal{A})$  konstruieren
-  Details finden sich beispielsweise im Buch von Ingo Wegener



# Det. kontextfreie Sprachen: Vereinigung und Durchschnitt

## Satz 10.12

- Die Klasse der deterministisch kontextfreien Sprachen ist **nicht** abgeschlossen unter Vereinigung und Durchschnitt

## Beweis

- Seien wieder
  - $L_1 := \{a^n b^n c^m \mid n, m \geq 1\}$  und
  - $L_2 := \{a^m b^n c^n \mid n, m \geq 1\}$
- $L_1$  und  $L_2$  sind sogar deterministisch kontextfrei
- Aber:  $L_1 \cap L_2$  ist noch nicht einmal kontextfrei
- ➡ Die deterministisch kontextfreien Sprachen sind nicht unter Durchschnitt abgeschlossen
- ➡ Die deterministisch kontextfreien Sprachen sind nicht unter Vereinigung abgeschlossen
- ✎ Sonst wären sie wegen De Morgan auch unter Durchschnitt abgeschlossen


# Nicht alle kontextfreien Sprachen haben einen DPDA

- Die Klasse der kontextfreien Sprachen hat also andere Abschlusseigenschaften als die Klasse der deterministisch kontextfreien Sprachen
  - ➡ die beiden Klassen sind verschieden
  - ➡ die deterministisch kontextfreien Sprachen bilden eine echte Teilklasse der Klasse der kontextfreien Sprachen
- Wir betrachten nun ein Beispiel einer kontextfreien Sprache, die nicht deterministisch kontextfrei ist:
  - Sei  $L_{\text{doppel}}$  die Sprache  $\{ww \mid w \in \{0,1\}^*\}$
  - Sei  $L_{\text{undoppel}}$  das Komplement von  $L_{\text{doppel}}$
  - Also:  $L_{\text{undoppel}}$  enthält alle Strings ungerader Länge sowie alle Strings aus  $L_{\text{diff}}$

## Proposition 10.13

- $L_{\text{undoppel}}$  ist kontextfrei aber nicht deterministisch kontextfrei

## Beweisskizze

- Dass  $L_{\text{doppel}}$  nicht kontextfrei ist, haben wir in Proposition 12.3 (b) schon bewiesen
- Also kann  $L_{\text{undoppel}}$  nicht deterministisch kontextfrei sein, sonst wäre es ja auch  $L_{\text{doppel}}$   Komplementabschluss
- Andererseits ist  $L_{\text{undoppel}}$  die Vereinigung
  - einer regulären Sprache (Strings ungerader Länge) und
  - der kontextfreien Sprache  $L_{\text{diff}}$und damit kontextfrei

# Verhältnis zu anderen Klassen

## Satz 10.14

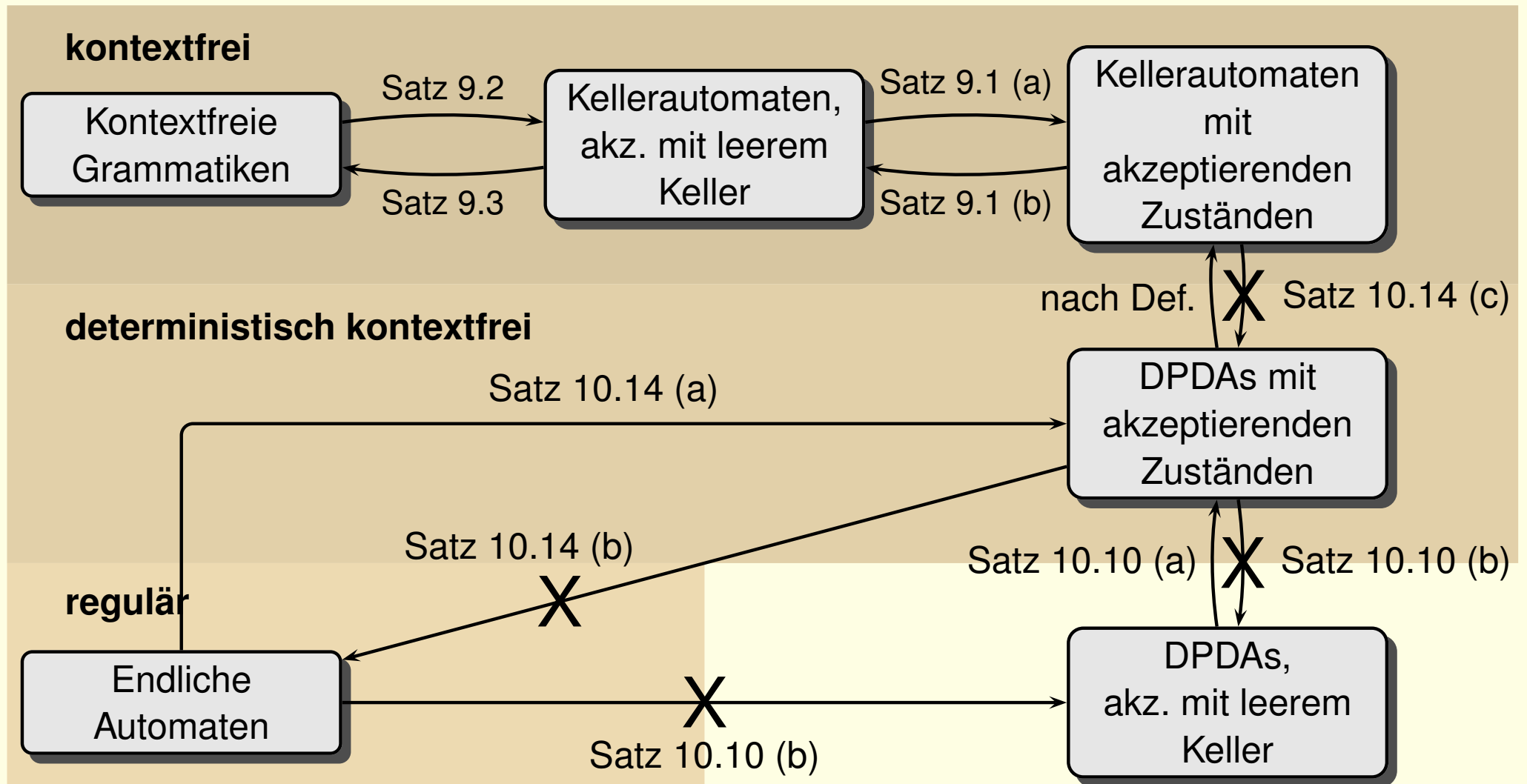
- (a) Jede reguläre Sprache ist deterministisch kontextfrei
- (b) Es gibt deterministisch kontextfreie Sprachen, die nicht regulär sind
- (c) Es gibt kontextfreie Sprachen, die nicht deterministisch kontextfrei sind

## Beweisidee

- (a) Jeder DFA kann als deterministischer Kellerautomat mit akzeptierenden Zuständen interpretiert werden, der seinen Keller nicht verwendet
- (b) Beispiel:  $\{a^n b^n \mid n \geq 0\}$
- (c) Beispiele:
  - $L_{\text{undoppel}}$
  - $L_{\text{rev}}$  (Beweis etwas komplizierter)

👉 Proposition 10.13

# Verhältnis der Modelle



# Zusammenfassung

- Das Pumping-Lemma für kontextfreie Sprachen ist ein Hilfsmittel um nachzuweisen, dass eine gegebene Sprache nicht kontextfrei ist
- Es gibt einige stärkere Versionen des Pumping-Lemmas, wie zum Beispiel **Ogden's Lemma**  
(siehe Buch von Ingo Wegener)
- Die kontextfreien Sprachen haben etwas weniger günstige algorithmische und Abschlusseigenschaften als die regulären Sprachen
- Insbesondere gibt es Fragen, die sich algorithmisch gar nicht lösen lassen
- Deterministische Kellerautomaten sind echt schwächer als Kellerautomaten und echt stärker als endliche Automaten

# Literatur für dieses Kapitel

- **Effiziente Umwandlung in GNF:**
  - Daniel J. Rosenkrantz. Matrix equations and normal forms for context-free grammars. *J. ACM*, 14(3):501–507, 1967
  - Norbert Blum and Robert Koch. Greibach normal form transformation revisited. *Inf. Comput.*, 150(1):112–118, 1999
- **Leerheitstest für kontextfreie Grammatiken:**
  - Catriel Beeri and Philip A. Bernstein. Computational problems related to the design of normal form relational schemas. *ACM Trans. Database Syst.*, 4(1):30–59, 1979
- **Pumping-Lemma:**
  - Y. Bar-Hillel, M. Perles, and E. Shamir. On formal properties of simple phrase structure grammars. *Z. Phonetik, Sprachwiss. Kommunikationsforsch.*, 14:143–172, 1961

# Endlichkeitstest für kontextfreie Sprachen: Beweisdetails

## Beweis (Forts.)

- Zu zeigen:  
 $L(G)$  unendlich  $\iff H(G)$  hat Kreis
  - Sei  $G = (V, \Sigma, S, P)$
  - Wir zeigen zuerst: „ $\Leftarrow$ “
  - Da  $H(G)$  einen Kreis hat, gibt es eine Variable  $X$  mit  $X \Rightarrow_G^* vXx$ , für gewisse  $v, x \in \Sigma^*$  mit  $vx \neq \epsilon$ 
    - \* Denn: die Anwendung der Regeln, die die Kanten des Kreises ergeben haben, liefert  $X \Rightarrow_G^* \alpha X \beta$  für gewisse  $\alpha, \beta \in V^*$
    - \* Alle in  $\alpha, \beta$  vorkommenden Variablen lassen sich zu nichtleeren Strings ableiten
  - Da  $X$  nützlich ist, gilt
    - \*  $X \Rightarrow_G^* w$  für ein  $w \in \Sigma^*$
    - \*  $S \Rightarrow_G^* uXy$ , für gewisse  $u, y \in \Sigma^*$
- ➡ alle (unendlich vielen) Strings der Form  $uv^kwx^ky$ ,  $k \geq 0$  sind in  $L$

## Beweis (Forts.)

- Wir zeigen jetzt: „ $\Rightarrow$ “
- Sei  $m \stackrel{\text{def}}{=} |V|$
- Wenn  $L(G)$  unendlich viele Strings enthält, enthält  $L(G)$  insbesondere einen String  $w$  mit  $|w| > 2^{m+1}$
- Wie im Beweis des Pumping-Lemmas muss deshalb auf einem Weg eines Blattes des Ableitungsbaumes zu  $w$  eine Variable  $X$  mehrfach vorkommen
- ➡  $X$  liegt in  $H(G)$  auf einem Kreis
- ➡ Behauptung

# Grundbegriffe der Theoretischen Informatik

Sommersemester 2016 - Thomas Schwentick

Teil B: Kontextfreie Sprachen

11: Wortproblem und Syntaxanalyse

Version von: 10. August 2016 (20:37)



# Wortproblem und Syntaxanalyse für kontextfreie Sprachen

- Phasen eines Compilers (schematisch):
  - Lexikalische Analyse
  - Syntaktische Analyse
  - Semantische Analyse
  - Zwischencode-Erzeugung
  - Zwischencode-Optimierung
  - Code-Erzeugung
- Bei der **Syntaxanalyse** wird die Struktur eines Programmes überprüft und in Form eines Syntax-Baumes repräsentiert
- Sie wird vom **Parser** durchgeführt
- Hierbei spielen kontextfreie Sprachen eine wichtige Rolle

- Die Syntaxanalyse liefert die Information, ob das gegebene Programm  $w$  syntaktisch korrekt ist
- Dies entspricht folgendem algorithmischen Problem:

Def.: Wortproblem für kontextfreie Sprachen

**Gegeben:** Wort  $w \in \Sigma^*$ , Grammatik  $G$

**Frage:** Ist  $w \in L(G)$ ?

- Wenn das Programm syntaktisch korrekt ist, soll die Syntaxanalyse auch einen Ableitungsbaum liefern, da dieser das Rückgrat für die Codeerzeugung darstellt
- Außerdem sollte bei syntaktisch inkorrekten Programmen  $w$  eine Begründung geliefert werden, warum  $w \notin L(G)$  ☞ Das betrachten wir nicht

Def.: Syntaxanalyse-Problem für kfr. Sprachen

**Gegeben:** Wort  $w \in \Sigma^*$ , Grammatik  $G$

**Gesucht:** Falls  $w \in L(G)$ : Ableitungsbaum

# Übersicht

- Das Wortproblem für reguläre Sprachen kann für jede feste reguläre Sprache in linearer Zeit gelöst werden
  - durch Auswertung eines DFA
- Auch deterministische Kellerautomaten können in linearer Zeit ausgewertet werden
  - aber leider gibt es nicht für jede kontextfreie Sprache einen deterministischen Kellerautomaten
- Wir werden in diesem Kapitel sehen:
- Die naive Auswertung von PDAs mit Backtracking kann zu exponentieller Laufzeit führen kann
- Es gibt einen Algorithmus, der das Syntaxanalyse-Problem für kontextfreie Sprachen in polynomieller Zeit löst:
  - Der **CYK-Algorithmus** basiert auf dynamischer Programmierung und hat Laufzeit  $\mathcal{O}(|G||w|^3)$
- Da kubische Laufzeit für viele Zwecke nicht akzeptabel ist, betrachten wir danach zwei eingeschränkte Grammatiktypen, die eine Syntaxanalyse in **linearer Zeit** erlauben:
  - LL(**1**)-Grammatiken: recht einfach zu definieren
  - LR(**1**)-Grammatiken: komplizierter zu definieren, aber gleichmächtig zu DPDAs

# Syntaxanalyse: Herangehensweisen

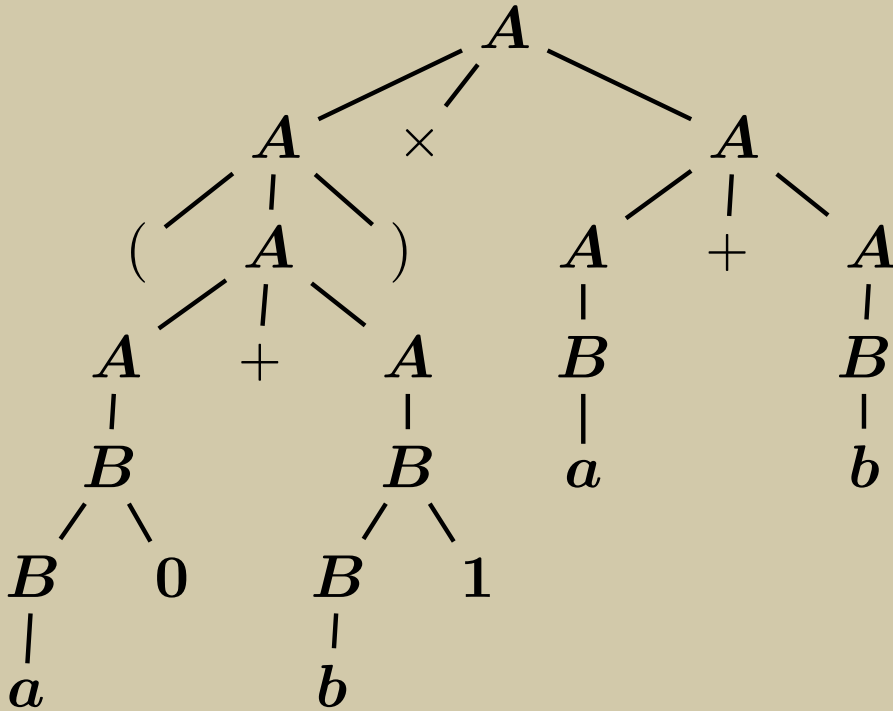
- Wir betrachten zwei Arten von Algorithmen für das Wortproblem für kontextfreie Sprachen
- Algorithmen, die versuchen beim Lesen des Eingabestrings von links nach rechts eine Ableitung zu erzeugen
  - Backtracking (Linksableitung, top-down)
  - $LL(k)$  (Linksableitung, top-down)
  - $LR(k)$  (Rechtsableitung, bottom-up)
- Algorithmen, die den Eingabestring „als Ganzes“ analysieren
  - CYK-Algorithmus
- Bevor wir uns dem Backtracking-Algorithmus zuwenden, werfen wir zunächst einen Blick auf den Ansatz der Top-down Syntaxanalyse
  - Bottom-up Syntaxanalyse werden wir gegen Ende des Kapitels betrachten

# Top-down Syntaxanalyse (1/3)

# Beispiel-Grammatik

$$\begin{array}{c} A \rightarrow B \mid A + A \mid A \times A \mid (A) \\ B \rightarrow a \mid b \mid Ba \mid Bb \mid B0 \mid B1 \end{array}$$

# Beispiel-Ableitungsbaum



- Bei der Top-down Syntaxanalyse wird der Ableitungsbaum von oben nach unten und (üblicherweise) von links nach rechts konstruiert
- Dieses Vorgehen ergibt eine Linksableitung

## Beispiel-Ableitung: Top-down

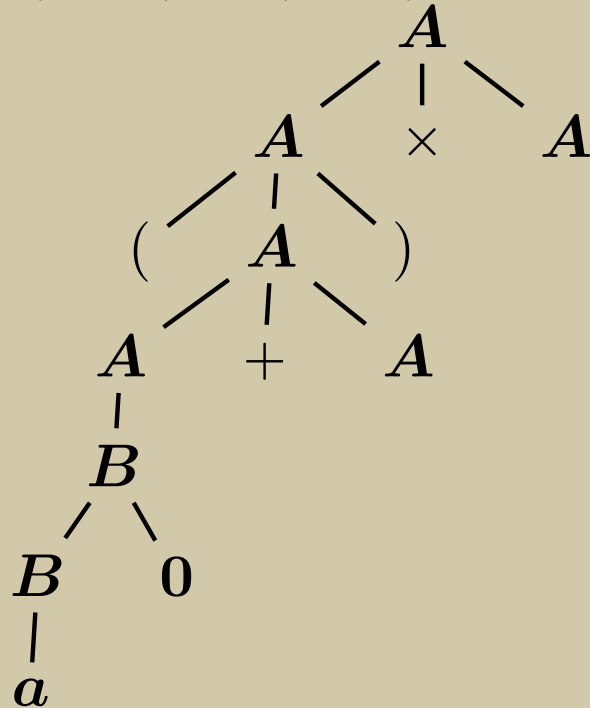
$$\begin{aligned}
A &\Rightarrow A \times A \\
&\Rightarrow (A) \times A \\
&\Rightarrow (A + A) \times A \\
&\Rightarrow (B + A) \times A \\
&\Rightarrow (B0 + A) \times A \\
&\Rightarrow (a0 + A) \times A \\
&\Rightarrow (a0 + B) \times A \\
&\Rightarrow (a0 + B1) \times A \\
&\Rightarrow (a0 + b1) \times A \\
&\Rightarrow (a0 + b1) \times A + A \\
&\Rightarrow (a0 + b1) \times B + A \\
&\Rightarrow (a0 + b1) \times a + A \\
&\Rightarrow (a0 + b1) \times a + B \\
&\Rightarrow (a0 + b1) \times a + b
\end{aligned}$$

## Top-down Syntaxanalyse (2/3)

### Beispiel

$A \rightarrow B \mid A + A \mid A \times A \mid (A)$

$B \rightarrow a \mid b \mid Ba \mid Bb \mid B0 \mid B1$



- Eingabe:  $(a0 + b1) \times a + b$

### Beispiel

- Der angegebene unvollständige Ableitungsbaum stellt eine Zwischensituation bei der Erzeugung einer Linksableitung für  $(a0 + b1) \times a + b$  dar
- Der linke Teil der Blätter des Baumes stimmt mit dem Anfang der Eingabe überein:  $(a0 +$
- Der unvollständige Baum entspricht der Satzform:  $(a0 + A) \times A$
- Als nächstes muss also die Variable  $A$  ersetzt werden
- Die restliche Satzform ist:  $) \times A$

## Top-down Syntaxanalyse (3/3)

- Die allgemeine Situation bei der Bestimmung des nächsten Schrittes einer Linksableitung für einen Eingabestring  $w$  ist wie folgt:

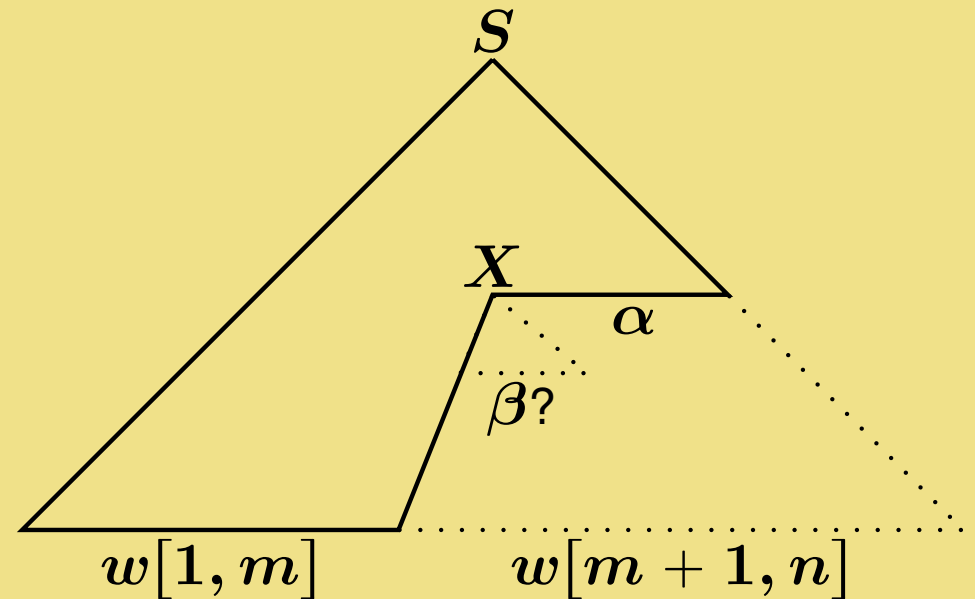
- Es ist schon eine Satzform abgeleitet
- Ihre erste Variable bezeichnen wir mit  $X$
- Die davor stehenden Zeichen aus dem Alphabet  $\Sigma$  müssen mit dem Anfang der Eingabe übereinstimmen

☞  $m \stackrel{\text{def}}{=} \text{Anzahl dieser Zeichen}$

- Den Rest der Satzform bezeichnen wir mit  $\alpha$

- Wir haben also:  $S \Rightarrow_l^* w[1, m]X\alpha$
- Damit insgesamt  $w$  erzeugt wird, muss also aus  $X\alpha$  der restliche String  $w[m + 1, n]$  erzeugt werden

- Der nächste Ableitungsschritt ist gesucht:  
 $w[1, m]X\alpha \Rightarrow_l w[1, m]\beta\alpha$



## 11.1 Algorithmen für beliebige kontextfreie Sprachen

### ▷ 11.1.1 Backtracking

#### 11.1.2 Der CYK-Algorithmus

## 11.2 Effiziente Syntaxanalyse

## Backtracking-Algorithmus: Idee

- Der Backtrackingalgorithmus versucht systematisch eine Linksableitung zu erzeugen
  - Er probiert dazu nach Ableitung von  $S \Rightarrow_l^* w[1, m] X \alpha$  alle Regeln der Form  $X \rightarrow \beta$  nacheinander aus
  - Wenn die entstehende Satzform nicht zur Eingabe passt oder zu lang wird, wählt er beim nächsten Mal die nächste Regel
  - Dabei kann es nötig sein, Schritte wieder rückgängig zu machen
- Die Laufzeit des Backtracking-Algorithmus kann exponentiell werden, wie das folgende Beispiel zeigt



# Backtracking-Algorithmus: Beispiel (1/2)

## • Backtracking-Algorithmus:

- Versuche, Linksableitung zu erzeugen
- Wähle immer jeweils die erste passende Regel
- Falls nicht erfolgreich:  
\* zurücksetzen und nächste Regel wählen

## Beispiel

### • Grammatik:

$$S \rightarrow aA0 \quad (1)$$

$$S \rightarrow aB1 \quad (2)$$

$$A \rightarrow aA0 \quad (3)$$

$$A \rightarrow aB1 \quad (4)$$

$$A \rightarrow c \quad (5)$$

$$B \rightarrow aA0 \quad (6)$$

$$B \rightarrow aB1 \quad (7)$$

$$B \rightarrow c \quad (8)$$

### • Eingabe: aaac111

## Lauf des Backtracking-Algorithmus

| Eingabe  | Satzform   | Regeln    | Letzte Aktion       |
|----------|------------|-----------|---------------------|
| aaac111  | S          |           |                     |
| a aac111 | aA0        | 1         | Regel 1             |
| a aac111 | a A0       | 1         | Vergleich: ok       |
| a aac111 | a aA00     | 1 3       | Regel 3             |
| aa ac111 | aa A00     | 1 3       | Vergleich: ok       |
| aa ac111 | aa aA000   | 1 3 3     | Regel 3             |
| aaa c111 | aaa A000   | 1 3 3     | Vergleich: ok       |
| aaa c111 | aaa aA0000 | 1 3 3 3   | Regel 3             |
| aaac 111 | aaaa A0000 | 1 3 3 3   | Vergleich: nicht ok |
| aaa c111 | aaa A000   | 1 3 3 (3) | zurück              |
| aaa c111 | aaa aB1000 | 1 3 3 4   | Regel 4             |
| aaac 111 | aaaa B1000 | 1 3 3 4   | Vergleich: nicht ok |
| aaa c111 | aaa A000   | 1 3 3 (4) | zurück              |
| aaa c111 | aaa c000   | 1 3 3 5   | Regel 5             |
| aaac 111 | aaac 000   | 1 3 3 5   | Vergleich: ok       |
| aaac1 11 | aaac0 00   | 1 3 3 5   | Vergleich: nicht ok |
| aaa c111 | aaa A000   | 1 3 3 (5) | zurück              |
| aa ac111 | aa A00     | 1 3 (3)   | zurück              |
| aa ac111 | aa aB100   | 1 3 4     | Regel 4             |
| aaa c111 | aaa B100   | 1 3 4     | Vergleich: ok       |
| aaa c111 | aaa aA0100 | 1 3 4 6   | Regel 6             |
| aaac 111 | aaaa A0100 | 1 3 4 6   | Vergleich: nicht ok |

# Backtracking-Algorithmus: Beispiel (2/2)

## • Backtracking-Algorithmus:

- Versuche, Linksableitung zu erzeugen
- Wähle immer jeweils die erste passende Regel
- Falls nicht erfolgreich:  
\* zurücksetzen und nächste Regel wählen

## Beispiel

### • Grammatik:

- $S \rightarrow aA0$  (1)
- $S \rightarrow aB1$  (2)
- $A \rightarrow aA0$  (3)
- $A \rightarrow aB1$  (4)
- $A \rightarrow c$  (5)
- $B \rightarrow aA0$  (6)
- $B \rightarrow aB1$  (7)
- $B \rightarrow c$  (8)

### • Eingabe: aaac111

## Lauf des Backtracking-Algorithmus (Forts.)

| Eingabe  | Satzform   | Regeln    | Letzte Aktion       |
|----------|------------|-----------|---------------------|
| aaac 111 | aaaa A0100 | 1 3 4 6   | Vergleich: nicht ok |
| aaa c111 | aaa B100   | 1 3 4 (6) | zurück              |
| aaa c111 | aaa aB1100 | 1 3 4 7   | Regel 7             |
| aaac 111 | aaaa B1100 | 1 3 4 (7) | Vergleich: nicht ok |
| aaa c111 | aaa B100   | 1 3 4 (7) | zurück              |
| aaa c111 | aaa c100   | 1 3 4 8   | Regel 8             |
| aaac 111 | aaac 100   | 1 3 4 8   | Vergleich: ok       |
| aaac1 11 | aaac1 00   | 1 3 4 8   | Vergleich: ok       |
| aaac11 1 | aaac10 0   | 1 3 4 8   | Vergleich: nicht ok |
| aaa c111 | aaa B100   | 1 3 4 (8) | zurück              |
| aa ac111 | aa A00     | 1 3 (4)   | zurück              |
| aa ac111 | aa c00     | 1 3 5     | Regel 5             |
| aaa c111 | aac 00     | 1 3 5     | Vergleich: nicht ok |
| aa ac111 | aa A00     | 1 3 (5)   | zurück              |
| a aac111 | a A0       | 1 (3)     | zurück              |
| a aac111 | a aB10     | 1 4       | Regel 4             |
| ...      | ...        | ...       | ...                 |
| aaac111  | aaac111    | 2 7 7 8   | Vergleich: ok       |

- Beobachtung: Bei Eingabe  $a^n c 1^n$  kommen alle  $n$ -stelligen Binärzahlen  $x$  in einer Satzform  $a^n c x$  vor  
➔ exponentiell viele Schritte

## 11.1 Algorithmen für beliebige kontextfreie Sprachen

11.1.1 Backtracking

▷ **11.1.2 Der CYK-Algorithmus**

11.2 Effiziente Syntaxanalyse

# Der CYK-Algorithmus

- Exponentieller Aufwand ist bei der Syntaxanalyse natürlich inakzeptabel
- Wir betrachten jetzt einen Algorithmus, der das Wortproblem für **beliebige** kontextfreie Grammatiken in polynomieller Zeit löst  
(für CNF in Zeit  $\mathcal{O}(|G||w|^3)$ )
- Der **CYK-Algorithmus** wurde von Cocke, Younger und Kasami (unabhängig voneinander) entwickelt  
(um 1965)
  - „Richtig“ veröffentlicht wurde er nur von Younger  
[\[Younger 67\]](#)
- Der CYK-Algorithmus verwendet **dynamische Programmierung**
- Er nutzt aus, dass die Grammatik  $G$  für  $L$  in CNF ist, lässt sich aber für kontextfreie Grammatiken, die nicht in CNF sind, anpassen

# Der CYK-Algorithmus: Grundidee

- Sei  $G$  in CNF gegeben und  $w$  ein String der Länge  $n$
- Zunächst wird das Problem in Teilprobleme zerlegt, die durch Parameter repräsentiert werden  
✎ Das Problem wird also „parametrisiert“

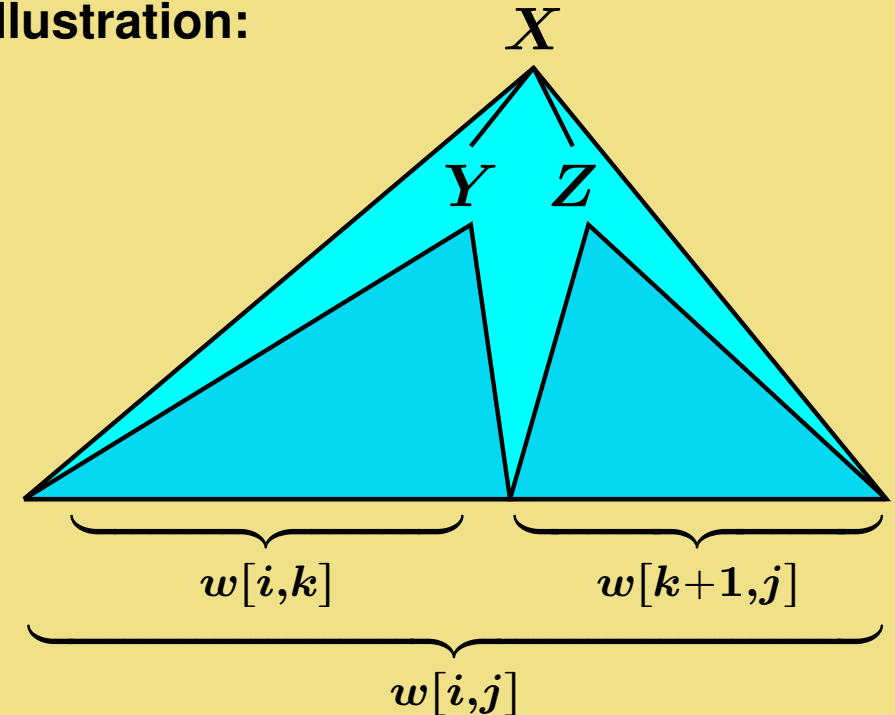
- Für jede Wahl von  $i, j \in \{1, \dots, n\}$  mit  $i \leq j$  sei

$$\underline{V_{i,j}} \stackrel{\text{def}}{=} \{X \in V \mid X \Rightarrow^* w[i, j]\}$$

- ✎  $V_{ij}$  ist also die Menge aller Variablen, aus denen  $w[i, j]$  abgeleitet werden kann

- Klar:  $w \in L(G) \iff S \in V_{1,n}$
- Der CYK-Algorithmus berechnet die Mengen  $V_{i,j}$  **bottom-up**

- **Illustration:**



- Er nutzt aus, dass bei einer CNF-Grammatik für  $X \in V$ ,  $1 \leq i < j \leq n$  äquivalent sind:
  - $X \in V_{ij}$
  - es gibt  $Y, Z \in V$  und  $k \in \{i, \dots, j-1\}$  mit:
    - \*  $X \rightarrow YZ$  ist Regel von  $G$ ,
    - \*  $Y \in V_{i,k}$  und
    - \*  $Z \in V_{k+1,j}$

# Der CYK-Algorithmus

## Algorithmus 11.1 (CYK-Algorithmus)

**Eingabe:**  $w \in \Sigma^*$ ,  $G = (V, \Sigma, S, P)$  in CNF

**Ausgabe:** „ja“, falls  $w \in L(G)$

```
1: for  $i := 1$  TO  $n$  do
2:    $V_{i,i} := \{X \in V \mid X \rightarrow w[i, i] \text{ in } P\}$ 
3: for  $\ell := 1$  TO  $n - 1$  do
4:   for  $i := 1$  TO  $n - \ell$  do
5:      $V_{i,i+\ell} := \emptyset$  {Teilstrings der Länge  $\ell + 1$ }
6:     for  $k := i$  TO  $i + \ell - 1$  do
7:        $V_{i,i+\ell} := V_{i,i+\ell} \cup \{X \mid$ 
          $X \rightarrow YZ \text{ in } P, Y \in V_{i,k}, Z \in V_{k+1,i+\ell}\}$ 
8: if  $S \in V_{1,n}$  then
9:   Akzeptieren
10: else
11:   Ablehnen
```

- Anweisung 7 wird durch eine Schleife über alle Regeln von  $G$  implementiert

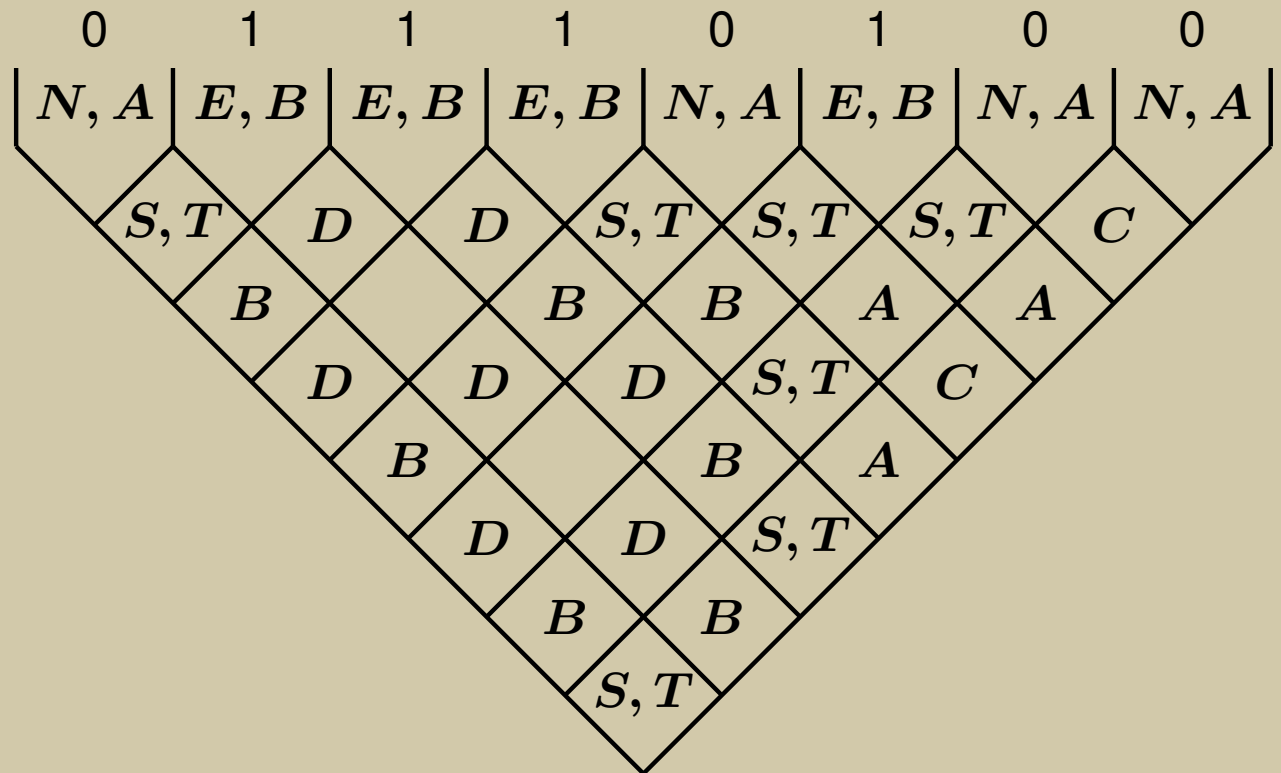
- Dass der Aufwand  $\mathcal{O}(n^3|G|)$  ist (für  $n = |w|$ ), lässt sich an den verschachtelten Schleifen des Algorithmus direkt ablesen

# CYK-Algorithmus: Beispiel

## Beispiel-Grammatik

$S \rightarrow NB \mid EA \mid \epsilon$   
 $T \rightarrow NB \mid EA$   
 $N \rightarrow 0$   
 $E \rightarrow 1$   
 $A \rightarrow 0 \mid NT \mid EC$   
 $B \rightarrow 1 \mid ET \mid ND$   
 $C \rightarrow AA$   
 $D \rightarrow BB$

## Verlauf der Bearbeitung



- Ergebnis:  $S \in V_{1,8}$  deshalb:  $01110100 \in L(G)$
- Wie lässt sich nun ein Ableitungsbaum für **01110100** gewinnen?
  - Durch eine kleine Erweiterung des CYK-Algorithmus: er merkt sich jeweils nicht nur  $X$  sondern auch das zugehörige  $k$


# Der erweiterte CYK-Algorithmus

## Algorithmus 11.2 Erweiterter CYK-Algorithmus

**Eingabe:**  $w \in \Sigma^*$ ,  $G = (V, \Sigma, S, P)$  in CNF

**Ausgabe:** Ableitungsbaum, falls  $w \in L(G)$

- 1: **for**  $i := 1$  **TO**  $n$  **do**
- 2:    $V_{i,i} := \{(X, i) \mid X \rightarrow w[i, i] \text{ in } P\}$
- 3: **for**  $\ell := 1$  **TO**  $n - 1$  **do**
- 4:   **for**  $i := 1$  **TO**  $n - \ell$  **do**
- 5:      $V_{i,i+\ell} := \emptyset$  {Teilstrings der Länge  $\ell + 1$ }
- 6:     **for**  $k := i$  **TO**  $i + \ell - 1$  **do**
- 7:        $V_{i,i+\ell} := V_{i,i+\ell} \cup \{(X, k) \mid$   
           $X \rightarrow YZ \text{ in } P, Y \in V_{i,k}, Z \in V_{k+1,i+\ell}\}$
- 8:   **if** es gibt kein  $k$  mit  $(S, k) \in V_{1,n}$  **then**
- 9:     Ablehnen
- 10: Konstruiere Ableitungsbaum rekursiv durch Aufruf von  $\text{Tree}(S, 1, n)$

 Dabei ist „ $Y \in V_{i,k}$ “ eine Abkürzung für:  
„es gibt ein  $m$  mit  $(Y, m) \in V_{i,k}$ “

## Algorithmus Tree

**Eingabe:**  $X, i, j$

**Ausgabe:** Ableitungsbaum für  
 $w[i, j]$  aus  $X$

- 1: **if**  $i = j$  **then**
- 2:   RETURN Blatt  $\sigma_i$
- 3: Wähle ein  $k$  mit  $(X, k) \in V_{i,j}$
- 4: Wähle  $Y, Z$ , so dass
  - $X \rightarrow YZ$ ,
  - $Y \in V_{i,k}$  und
  - $Z \in V_{k+1,j}$
- 5: RETURN Baum mit Wurzel  $X$ , linkem Teilbaum  $\text{Tree}(Y, i, k)$  und rechtem Teilbaum  $\text{Tree}(Z, k + 1, j)$

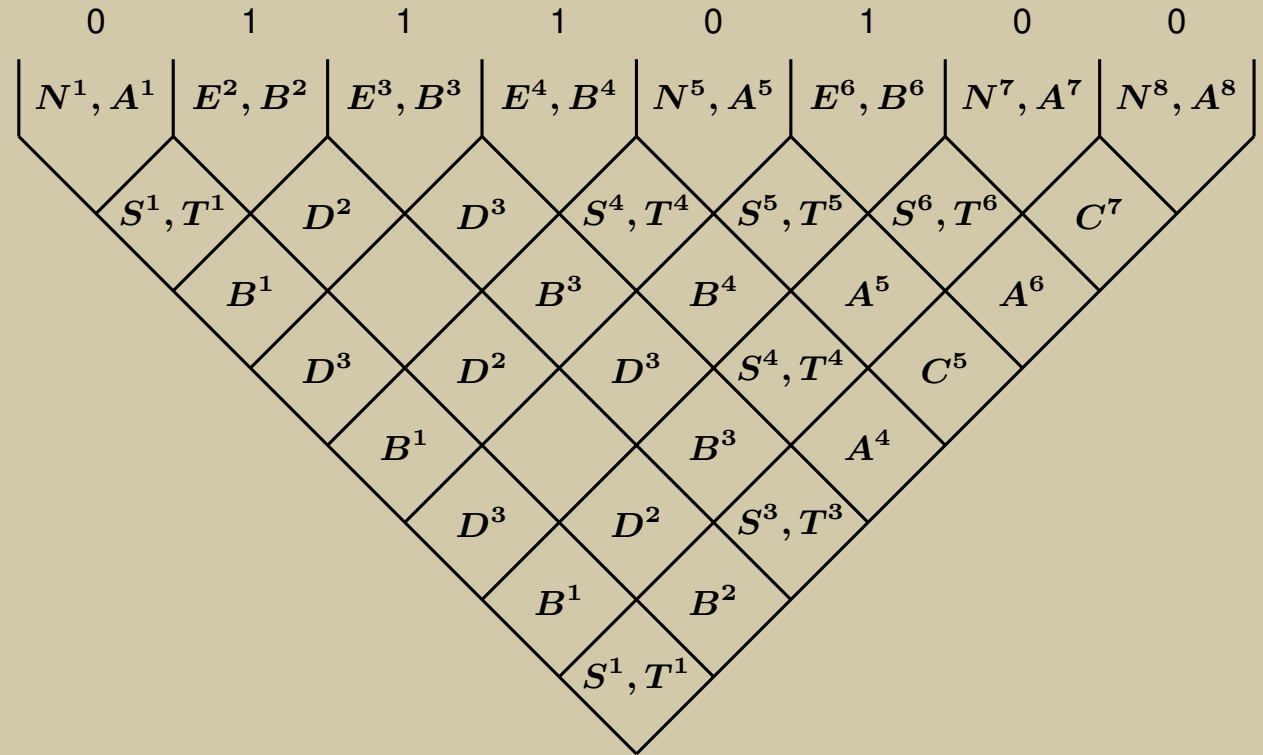


# Erweiterter CYK-Algorithmus: Beispiel

## Beispiel-Grammatik

$S \rightarrow NB \mid EA \mid \epsilon$   
 $T \rightarrow NB \mid EA$   
 $N \rightarrow 0$   
 $E \rightarrow 1$   
 $A \rightarrow 0 \mid NT \mid EC$   
 $B \rightarrow 1 \mid ET \mid ND$   
 $C \rightarrow AA$   
 $D \rightarrow BB$

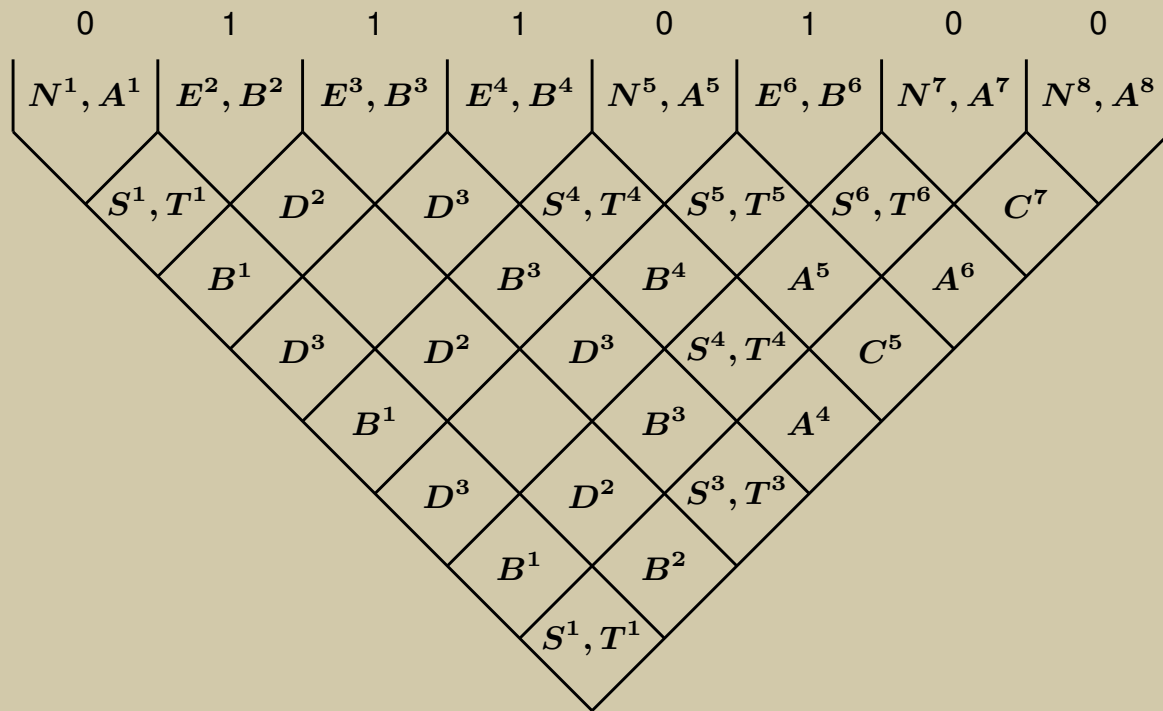
## Beispiel



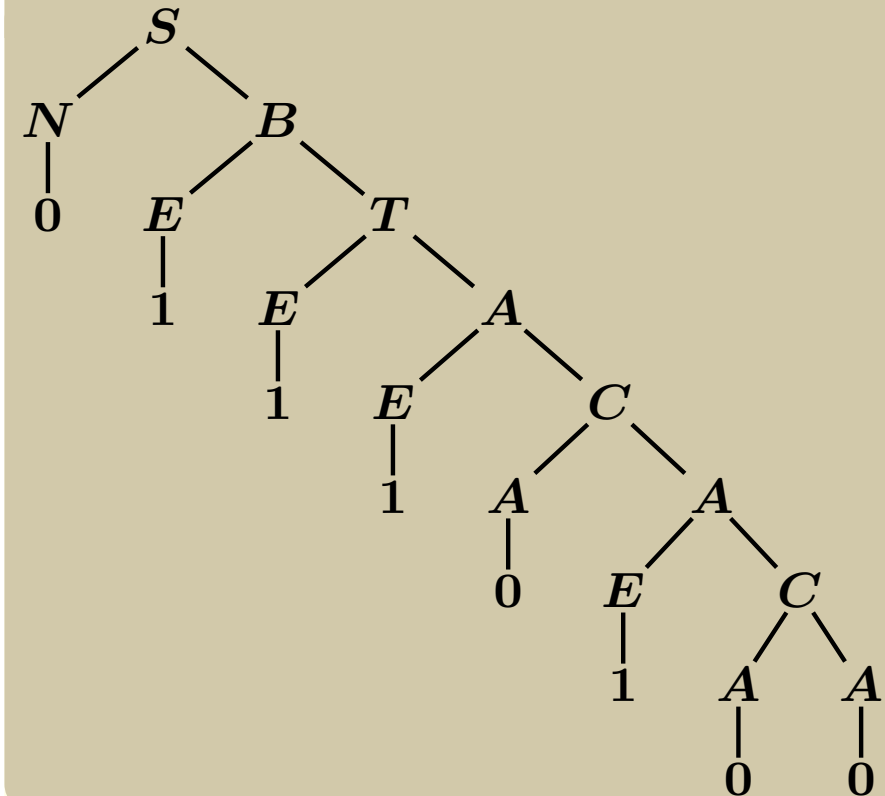
- Aus Platzgründen steht hier statt  $(X, k)$  jeweils  $X^k$
- Außerdem ist nur jeweils höchstens **ein**  $k$  mit  $(X, k) \in V_{i,j}$  angegeben

# Erweiterter CYK-Algorithmus: Beispiel-Ableitung

Beispiel



Beispiel



11.1 Algorithmen für beliebige kontextfreie Sprachen

▷ **11.2 Effiziente Syntaxanalyse**

11.2.1 Top-down-Syntaxanalyse

11.2.2 Bottom-up-Syntaxanalyse

# Effizientere Syntaxanalyse

- Syntaxanalyse von Programmtexten sollte möglichst in linearer Zeit erfolgen
  - Die beiden bisher betrachteten Algorithmen für das Syntaxanalyse-Problem sind also nicht effizient genug
- Für allgemeine kontextfreie Grammatiken sind aber leider keine Linearzeit-Algorithmen für das Syntaxanalyse-Problem bekannt
- Ein möglicher Ausweg ist, Grammatiken so einzuschränken, dass die Syntaxanalyse in linearer Zeit möglich wird
  - Das Ziel ist dabei, möglichst viele Sprachen mit den eingeschränkten Grammatiken beschreiben zu können

- Wir werden zwei Einschränkungen von Grammatiken kennen lernen
  - Bei beiden wird die Eingabe von links nach rechts gelesen
  - Bei beiden hängt die nächste Regelanwendung nur vom nächsten Zeichen der Eingabe ab
    - \* Damit kann uferloses Backtracking vermieden werden
- Bei  $LL(1)$ -Grammatiken wird, ausgehend vom Startsymbol, eine **Linksableitung** für  $w$  erzeugt
  - Top-down-Syntaxanalyse
- Bei  $LR(1)$ -Grammatiken wird, ausgehend von  $w$ , durch „Rückwärtsanwendung“ von Regeln eine **Rechtsableitung** erzeugt
  - Bottom-up-Syntaxanalyse
- Beide Grammatik-Typen gibt es auch mit Abhängigkeit von den nächsten  $k$  Zeichen

👉  $LL(k), LR(k)$

# Inhalt

11.1 Algorithmen für beliebige kontextfreie Sprachen

## **11.2 Effiziente Syntaxanalyse**

### ▷ **11.2.1 Top-down-Syntaxanalyse**

11.2.2 Bottom-up-Syntaxanalyse

# Effiziente Top-down-Syntaxanalyse: Vorüberlegungen (1/5)

- Im Folgenden betrachten wir nur Grammatiken  $G$ , die nicht **linksrekursiv** sind
  - ☞ d.h.:  $X \Rightarrow_G^* X\alpha$  mit  $\alpha \neq \epsilon$  ist verboten
    - Sonst bestünde die Gefahr von Endlosschleifen
- **Problem:** welche Regel soll angewendet werden, wenn mehrere Anwendungen möglich sind?
  - Wir wissen: Backtracking ist zu ineffizient
- **Idee** zur Vermeidung exponentiellen Aufwandes:
  - Wir schränken  $G$  so ein, dass immer direkt erkennbar ist, welche Regel angewendet werden muss
  - „Direkt erkennbar“ heißt dabei, dass für die Entscheidung nur das nächste Zeichen der Eingabe angeschaut werden muss

# Effiziente Top-down-Syntaxanalyse: Vorüberlegungen (2/5)

## Beispiel

- Wir betrachten die Grammatik für Palindrome:

$$P \rightarrow aPa \mid bPb \mid a \mid b \mid \epsilon$$

und versuchen eine Linksableitung für den String *abba* zu finden:

- Der Versuch scheitert schon im ersten Schritt, da aus der Kenntnis des ersten Zeichens *a* nicht hervorgeht, ob wir die Regel  $P \rightarrow a$  oder  $P \rightarrow aPa$  anwenden sollen
- Wenn wir korrekt mit  $P \Rightarrow aPa$  beginnen, stellt sich im zweiten Schritt wieder das Problem:

$$P \rightarrow b \text{ oder } P \rightarrow bPb$$

- Nach dem korrekten zweiten Schritt  $aPa \Rightarrow abPba$  gibt es wieder zwei Möglichkeiten:  $P \Rightarrow bPb$  oder  $P \Rightarrow \epsilon$

- Diese Grammatik ist also sicher nicht für die Top-down-Analyse mit nur einem „Vorschau-Zeichen“ geeignet
- Ein offensichtlicher Grund hierfür ist, dass es mehrere Regeln derselben Variablen gibt, deren rechte Seite mit demselben Terminalsymbol beginnt  
→ das verbieten wir in effizienten Top-down-Grammatiken

# Effiziente Top-down-Syntaxanalyse: Vorüberlegungen (3/5)

## Beispiel

- Wir betrachten die Grammatik

$$S \rightarrow aA \mid BC$$

$$A \rightarrow bA \mid c$$

$$B \rightarrow bC \mid Ca$$

$$C \rightarrow ba \mid ab$$

und suchen eine Linksableitung für *ababa*:

$$S \Rightarrow aA$$

$$\Rightarrow abA$$

$$\Rightarrow ?$$

Wir stellen fest, dass wir im ersten Schritt schon einen Fehler gemacht haben

- Eine Ableitung ergäbe sich durch:

$$S \Rightarrow BC$$

$$\Rightarrow CaC$$

$$\Rightarrow abaC$$

$$\Rightarrow ababa$$

- Was ist hier schiefgelaufen?

- Zwar haben die rechten Regelseiten hier jeweils verschiedene erste Terminalsymbole

- Aber aus *B* lässt sich der String *aba* ableiten

- Damit stehen die beiden rechten Regelseiten *aA* und *BC* miteinander in Konkurrenz, obwohl sie nicht mit dem selben Terminalsymbol beginnen

- Wir müssen also auch berücksichtigen, welche ersten Zeichen sich durch weitere Ableitung aus der ersten Variablen einer rechten Regelseite ergeben können



# Effiziente Top-down-Syntaxanalyse: Vorüberlegungen (4/5)

## Beispiel

- Wir betrachten die Grammatik

$$S \rightarrow aA \mid BC$$

$$A \rightarrow bA \mid c$$

$$B \rightarrow c \mid \epsilon$$

$$C \rightarrow ab$$

und suchen eine Linksableitung für  $ab$ :

$$S \Rightarrow aA$$

$$\Rightarrow abA$$

$$\Rightarrow ?$$

Wir stellen fest, dass wir wieder schon im ersten Schritt einen Fehler gemacht haben

- Eine Ableitung ergäbe sich durch:

$$S \Rightarrow BC$$

$$\Rightarrow C$$

$$\Rightarrow ab$$

- Was ist hier schiefgelaufen?

- Die rechten Regelseiten haben hier jeweils verschiedene erste Terminalsymbole, auch bei Berücksichtigung der möglichen ersten Terminalsymbole, die sich aus ersten Variablen rechter Regelseiten ableiten lassen

- Aber  $B$  lässt sich zu  $\epsilon$  ableiten und  $C$  zu  $ab$

- Deshalb stehen die beiden rechten Regelseiten  $aA$  und  $BC$  miteinander in Konkurrenz, obwohl aus  $B$  nicht als erstes Terminalsymbol  $a$  ableitbar ist

- Wir müssen also auch berücksichtigen, welche ersten Zeichen sich durch Ableitung aus den gesamten rechten Regelseiten ergeben können und dabei insbesondere  $\epsilon$ -Ableitungen berücksichtigen

# Effiziente Top-down-Syntaxanalyse: Vorüberlegungen (5/5)

## Beispiel

- Wir betrachten die Grammatik

$$S \rightarrow AB$$

$$A \rightarrow \epsilon \mid aB$$

$$B \rightarrow aaB \mid b$$

und suchen eine Linksableitung für  $aab$ :

$$S \Rightarrow AB$$

$$\Rightarrow aBB$$

$$\Rightarrow ?$$

Wir stellen fest, dass wir im zweiten Schritt schon einen Fehler gemacht haben

- Eine Ableitung ergäbe sich durch:

$$S \Rightarrow AB$$

$$\Rightarrow B$$

$$\Rightarrow aaB$$

$$\Rightarrow aab$$

- Was ist hier schiefgelaufen?

- Zwar haben die rechten Regelseiten jeweils verschiedene erste Terminalsymbole, auch bei Berücksichtigung der ableitbaren Strings
- Aber in  $AB$  kann das nächste Zeichen  $a$  sowohl aus  $A$  entstehen als auch (nach Anwendung von  $A \rightarrow \epsilon$ ) aus  $B$
- Dies führt dazu, dass bei der Satzform  $AB$  und nächstem Symbol  $a$  nicht klar ist, ob als nächstes  $A \rightarrow \epsilon$  oder  $A \rightarrow aB$  angewendet werden muss
- Wir müssen deshalb in einem solchen Fall ( $A \Rightarrow^* \epsilon$ ) auch darauf achten, welche Zeichen **hinter**  $A$  erzeugt werden können

- Diese Beobachtungen führen uns zur Definition von  $LL(1)$ -Grammatiken

# LL(1)-Grammatiken: Definition

- Zur Definition von LL(1)-Grammatiken verwenden wir die folgenden beiden Operatoren:

- Für eine Satzform  $\alpha$  sei  
 $\text{FIRST}(\alpha) \stackrel{\text{def}}{=} \{\sigma \in \Sigma \mid \alpha \Rightarrow^* \sigma v, v \in \Sigma^*\} \cup \{\epsilon \mid \alpha \Rightarrow^* \epsilon\}$

- Für eine Variable  $X$  sei  
 $\text{FOLLOW}(X) \stackrel{\text{def}}{=} \{\sigma \in \Sigma \mid S \Rightarrow^* uX\sigma v, u, v \in \Sigma^*\}$

- $\text{FIRST}(\alpha)$  enthält also alle ersten Terminalzeichen von Strings, die aus  $\alpha$  abgeleitet werden können (und evtl.  $\epsilon$ , wenn dieses aus  $\alpha$  abgeleitet werden kann)

- $\text{FOLLOW}(X)$  enthält alle Terminalzeichen, die in irgendeiner aus  $S$  ableitbaren Satzform unmittelbar hinter  $X$  vorkommen können

## Definition

- Sei  $G$  eine kontextfreie Grammatik  $G$  ohne nutzlose Variablen und ohne Linksrekursion
- $G$  ist eine **LL(1)-Grammatik**, wenn für alle Variablen  $X$  und alle Regeln  $X \rightarrow \alpha$  und  $X \rightarrow \beta$  mit  $\alpha \neq \beta$  die beiden folgenden Bedingungen gelten:
  - (i)  $\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \emptyset$
  - (ii) Falls  $\alpha \Rightarrow^* \epsilon$  so ist  
 $\text{FOLLOW}(X) \cap \text{FIRST}(\beta) = \emptyset$

- $\text{FIRST}(\alpha)$  und  $\text{FOLLOW}(X)$  können effizient berechnet werden
- LL(1)-Grammatiken lassen sich sehr einfach rekursiv implementieren
- Die Erzeugung von Code lässt sich dabei oft sehr leicht integrieren: **rekursiver Abstieg**

# LL(1)-Grammatiken: Beispiele

## Beispiel

$$\begin{aligned} S &\rightarrow aB \mid bA \mid c \\ A &\rightarrow aS \mid bAA \\ B &\rightarrow bS \mid aBB \end{aligned}$$

- Offensichtlich erfüllt die Grammatik alle Bedingungen der Art (i)
- Da keine  $\epsilon$ -Regeln vorkommen, erfüllt sie auch (ii)
- Also ist es eine LL(1)-Grammatik

## Beispiel

$$\begin{aligned} S &\rightarrow AB \\ A &\rightarrow \epsilon \mid aB \\ B &\rightarrow aA \mid b \end{aligned}$$


- FOLLOW( $A$ ) = { $a, b$ }
  - FIRST( $aB$ ) = { $a$ }
  - Da es eine Regel  $A \rightarrow \epsilon$  gibt, müssten diese beiden Mengen disjunkt sein
- keine LL(1)-Grammatik

## Beispiel

$$\begin{aligned} A &\rightarrow BC \mid ab \\ B &\rightarrow cAA \mid bc \mid \epsilon \\ C &\rightarrow cC \mid \epsilon \end{aligned}$$

- Es gilt:
    - FIRST( $A$ ) = { $a, b, c, \epsilon$ }
    - FIRST( $B$ ) = { $b, c, \epsilon$ }
    - FIRST( $C$ ) = { $c, \epsilon$ }
    - FOLLOW( $A$ ) = { $a, b, c$ }
    - FOLLOW( $B$ ) = { $a, b, c$ }
    - FOLLOW( $C$ ) = { $a, b, c$ }
    - $BC \Rightarrow^* \epsilon$
  - Also haben wir:
    - $A \rightarrow BC$  und  $A \rightarrow ab$ ,
    - $BC \Rightarrow^* \epsilon$
    - FOLLOW( $A$ )  $\cap$  FIRST( $ab$ ) = { $a$ }  $\neq \emptyset$
- ➔ Dies ist keine LL(1)-Grammatik

# LL(1)-Grammatiken: Parsing

- Parsing-Algorithmen für LL(1)-Grammatiken verwenden eine Tabelle, die dem Compiler in jeder Situation (Variable und nächstes Zeichen) sagt, welche Regel anzuwenden ist
- Die Berechnung dieser Tabelle wird hier nicht betrachtet
- Stattdessen schauen wir ein Beispiel an  
 Die Beispieldtabelle vernachlässigt das Wortende-Symbol
- LL( $k$ )-Grammatiken (für  $k \geq 2$ ) erlauben Parsing-Algorithmen, die die nächsten  $k$  Zeichen der Eingabe berücksichtigen, und verallgemeinern LL(1)-Grammatiken

## Beispiel

- Sei  $G$  die LL(1)-Grammatik

$$S \rightarrow AB \mid (S)S$$

$$A \rightarrow CA \mid \epsilon$$

$$B \rightarrow ba$$

$$C \rightarrow ca$$

- $\text{FIRST}(B) = \{b\}$ ,  $\text{FIRST}(C) = \{c\}$
- $\text{FIRST}(A) = \{c, \epsilon\}$
- $\text{FIRST}(S) = \{b, c, (\}$
- $\text{FOLLOW}(A) = \{b\}$

- Die zugehörige Tabelle ist:

|     | $a$ | $b$        | $c$  | $($    | $)$ |
|-----|-----|------------|------|--------|-----|
| $S$ |     | $AB$       | $AB$ | $(S)S$ |     |
| $A$ |     | $\epsilon$ | $CA$ |        |     |
| $C$ |     |            | $ca$ |        |     |
| $B$ |     | $ba$       |      |        |     |

# Inhalt

11.1 Algorithmen für beliebige kontextfreie Sprachen

## **11.2 Effiziente Syntaxanalyse**

11.2.1 Top-down-Syntaxanalyse

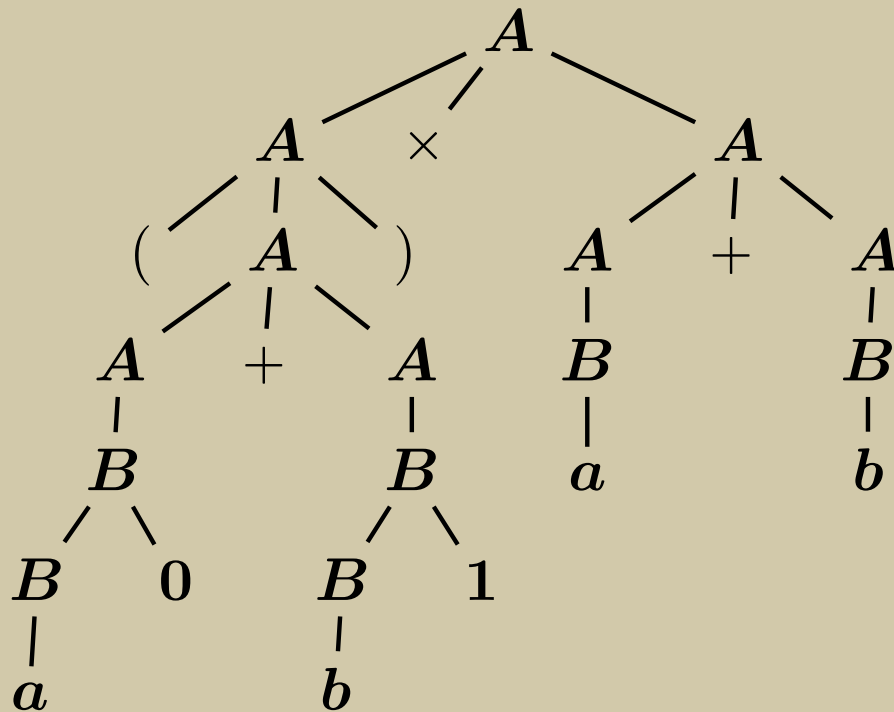
▷ **11.2.2 Bottom-up-Syntaxanalyse**

# Bottom-up Syntaxanalyse (1/3)

## Beispiel-Grammatik

$A \rightarrow B \mid A + A \mid A \times A \mid (A)$   
 $B \rightarrow a \mid b \mid Ba \mid Bb \mid B0 \mid B1$

## Beispiel-Ableitungsbaum



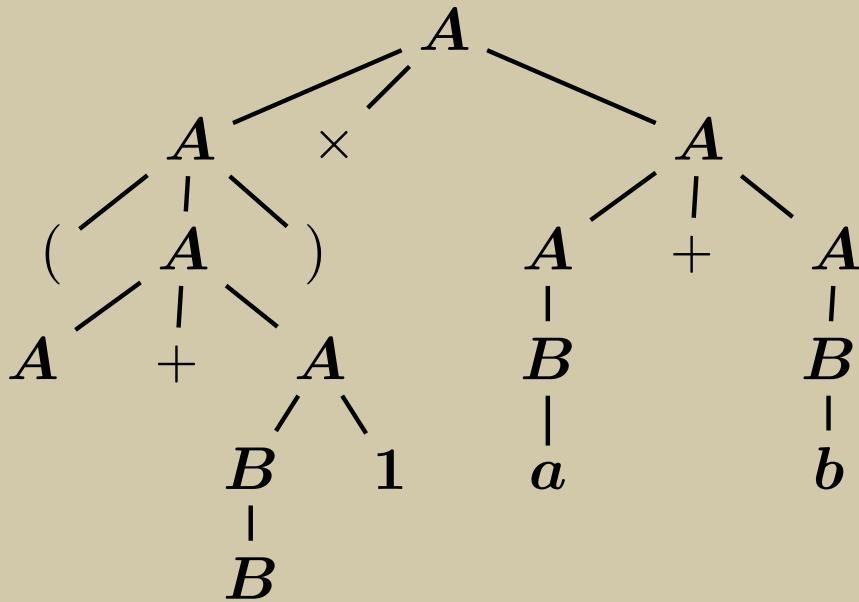
- Bei der Bottom-up Syntaxanalyse wird der Ableitungsbaum von unten nach oben und (üblicherweise) von links nach rechts konstruiert
- Dieses Vorgehen ergibt eine Rechtsableitung

## Beispiel-Ableitung: Bottom-up (rückwärts)

$(a0 + b1) \times a + b \Leftarrow$   
 $(B0 + b1) \times a + b \Leftarrow$   
 $(B + b1) \times a + b \Leftarrow$   
 $(A + b1) \times a + b \Leftarrow$   
 $(A + B1) \times a + b \Leftarrow$   
 $(A + B) \times a + b \Leftarrow$   
 $(A + A) \times a + b \Leftarrow$   
 $(A) \times a + b \Leftarrow$   
 $A \times a + b \Leftarrow$   
 $A \times B + b \Leftarrow$   
 $A \times A + b \Leftarrow$   
 $A \times A + B \Leftarrow$   
 $A \times A + A \Leftarrow$   
 $A \times A \Leftarrow A$

## Bottom-up Syntaxanalyse (2/3)

### Beispiel



- Eingabe:  $(a0 + b1) \times a + b$

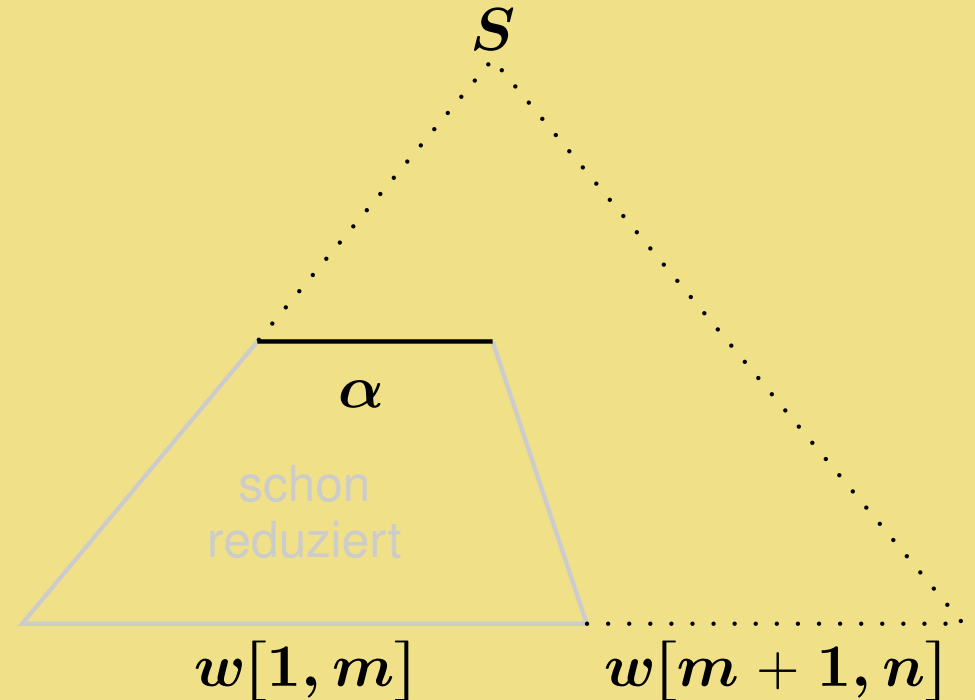
### Beispiel

- Der abgebildete unvollständige Ableitungsbaum stellt eine Zwischensituation bei der Erzeugung einer Rechtsableitung für  $(a0 + b1) \times a + b$  dar
- Das Anfangsstück  $(a0 + b1$  der Eingabe wurde schon gelesen und auf  $(A + B1$  reduziert
- Der unvollständige Baum entspricht der Satzform::  $(A + B1) \times a + b$
- Die restliche Eingabe  $) \times a + b$  muss noch gelesen werden
- Im nächsten Schritt muss wieder eine Regel rückwärts angewendet werden



## Bottom-up Syntaxanalyse (3/3)

- Die allgemeine Situation bei der Bestimmung des nächsten Schrittes einer Rechtsableitung für einen Eingabestring  $w$  ist wie folgt:
  - Für ein Anfangsstück  $w[1, m]$  des Strings wurden schon Regeln rückwärts angewendet
  - Die daraus entstandene Satzform  $\alpha$  liegt auf dem Keller
- Wir haben also:  $\alpha \Rightarrow_r^* w[1, m]$
- Damit insgesamt  $S$  erreicht wird, muss also  $\alpha w[m + 1, n]$  auf  $S$  „zurück abgeleitet werden“
- Im nächsten Schritt muss eine passende rechte Regelseite  $\beta$  und eine Regel  $X \rightarrow \beta$  identifiziert und dann (rückwärts) angewendet werden



# Bottom-up Syntaxanalyse: Vorüberlegungen

## Beispiel: Grammatik

$$S \rightarrow aABe$$

$$A \rightarrow Abc \mid b$$

$$B \rightarrow d$$

- Wir betrachten die Bottom-up Syntaxanalyse für den String *abbcede*
- Eine Rechtsableitung für diesen String:

## Beispiel: Rechtsableitung

$$S \Rightarrow aABe$$

$$\Rightarrow aAde$$

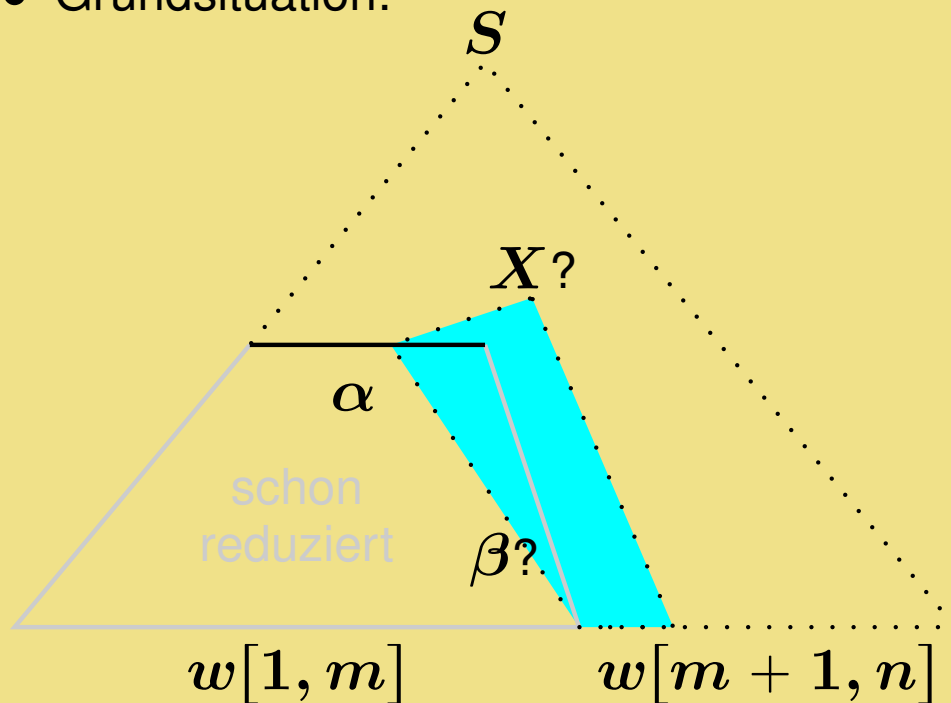
$$\Rightarrow aAbcde$$

$$\Rightarrow abbcde$$

- Bottom-up-Syntaxanalyse für *abbcede*
- Ziel: durch „umgekehrte“ Regelanwendung mit Rechtsableitung zu *S* kommen
- 1. Ableitungsschritt:
  - Wir suchen zuerst rechte Seiten von Produktionen in *abbcede*
  - Kandidaten: *b* und *d*
  - Wir entscheiden uns für  $A \rightarrow b$  und erhalten die Satzform *aAbcde*
- 2. Ableitungsschritt:
  - Kandidaten (in *aAbcde*): *Abc*, *b*, *d*
  - Welche Produktion sollen wir anwenden?
- Die Grammatik soll garantieren, dass das Finden der „richtigen rechten Seite“ immer „einfach“ ist
- Wir nennen diese „richtige rechte Seite“ den **Schlüssel** (engl.: **handle**)
  - Der Schlüssel von *aAbcde* ist *Abc*

# Bottom-up Sytaxanalyse: Prinzip (1/2)

- Grundsituation:



- $w[1, m]$  ist schon reduziert auf  $\alpha$
- $w[m + 1, n]$  ist noch zu lesen
- Nächster Schritt: Passenden Schlüssel  $\beta$  und Regel  $X \rightarrow \beta$  identifizieren und anwenden

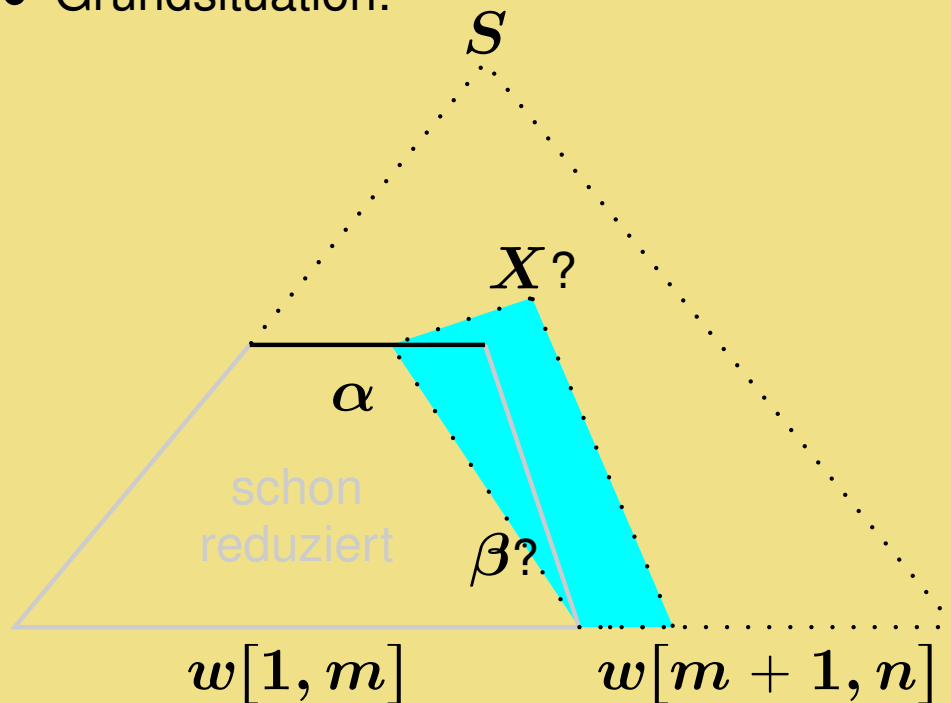
- $\beta$  kann
  - (a) ein Suffix von  $\alpha$  sein,
  - (b) ein Teilstring von  $w[m + 1, n]$  sein, oder
  - (c) aus einem Suffix von  $\alpha$  und einem Präfix von  $w[m + 1, n]$  bestehen

- Die jeweils auszuführende Aktion:
  - (a) Ersetze auf dem Keller  $\beta$  durch eine passende linke Regelseite  $X$  (**Reduce**)
  - (b,c) oder lege (zunächst)  $w[m + 1]$  auf den Keller (**Shift**)

→ **Shift-Reduce-Parsing**

## Bottom-up Sytaxanalyse: Prinzip (2/2)

- Grundsituation:




- Damit Shift-Reduce-Parsing ohne Backtracking möglich ist, muss die Grammatik folgende Bedingungen erfüllen:
  - (1) Der Parser muss den Schlüssel  $\beta$  identifizieren können, um zu entscheiden, ob er einen Reduce-Schritt ausführen kann
  - (2) Er muss erkennen können, bezüglich welcher Variablen  $X$  ein Reduktionsschritt mit Regel  $X \rightarrow \beta$  angewandt wird
- **Ziel bei LR(1)-Grammatiken:** um diese Entscheidungen zu treffen, muss nur das nächste Zeichen hinter dem (kürzesten möglichen) Schlüssel gelesen werden

# LR(1)-Grammatiken: Definition

## Definition

- Eine Grammatik  $G$  heißt **LR(1)-Grammatik**, falls für alle  $X \in V$ ,  $x, y \in \Sigma^*$ , und alle Satzformen  $\alpha, \beta, \gamma$  gelten:
  - (1) Falls für ein  $\sigma \in \Sigma$   
 $S \Rightarrow_r^* \alpha X \sigma x \Rightarrow_r \alpha \beta \sigma x$   
und  
 $\gamma \Rightarrow_r \alpha \beta \sigma y$   
gelten mit  $\gamma \neq \alpha X \sigma y$ , dann ist  $\gamma$  nicht von  $S$  aus ableitbar
  - (2) Falls  
 $S \Rightarrow_r^* \alpha X \Rightarrow_r \alpha \beta$   
und  
 $\gamma \Rightarrow_r \alpha \beta$   
gelten mit  $\gamma \neq \alpha X$ , dann ist  $\gamma$  nicht von  $S$  aus ableitbar

 Bedingung (1) sagt also, dass es mit aktueller Satzform  $\alpha\beta$  und nächstem Zeichen  $\sigma$  keine Alternative zur Rückwärtsanwendung von  $X \rightarrow \beta$  gibt

- Die Definition garantiert also gerade die Gültigkeit der Bedingungen (1) & (2) der vorherigen Folie
- Nach Lesen von  $\sigma$  (oder am Ende der Eingabe) und mit  $\alpha\beta$  auf dem Keller „weiß“ der Algorithmus, dass er  $X \rightarrow \beta$  anwenden muss
- Um zu einer LR(1)-Grammatik einen Shift-Reduce-Algorithmus zu gewinnen, ist eine genauere Analyse der Grammatik nötig
  - Dann können die anzuwendenden Regeln jeweils aus einer Tabelle abgelesen werden
  - Diese Analyse geht aber über den Rahmen dieser Vorlesung hinaus
- LR(1)-Grammatiken lassen sich verallgemeinern für eine weitere Vorausschau (*look-ahead*) von  $k$  Zeichen statt einem Zeichen

# LR( $k$ )-Grammatiken: Beispiele (1/2)

## Beispiel

$$S \rightarrow CD$$

$$C \rightarrow a$$

$$D \rightarrow EF \mid aG$$

$$E \rightarrow ab$$

$$F \rightarrow bb$$

$$G \rightarrow bba$$

- Die Grammatik hat nur zwei Rechtsableitungen:
  - $S \Rightarrow_r CD \Rightarrow_r CEF \Rightarrow_r CEbb \Rightarrow_r Cabbb$
  - $S \Rightarrow_r CD \Rightarrow_r CaG \Rightarrow_r Cabba$
- ➡ Sie erfüllt **nicht** die LR(1)-Bedingung:
  - $\sigma = b$
  - $\alpha = C, X = E, x = b, \beta = ab$
  - $\gamma = CaG, y = a$
  - Aber:  $\gamma = CaG \neq CEbb = \alpha X \sigma x$
- Sie ist aber eine LR(2)-Grammatik

# LR( $k$ )-Grammatiken

## Definition

- Für jedes  $k \geq 0$  heißt eine Grammatik  $G$  **LR( $k$ )-Grammatik**, falls für alle  $X \in V$ ,  $x, y \in \Sigma^*$ , und alle Satzformen  $\alpha, \beta, \gamma$  gelten:

- (1) Falls für ein  $z \in \Sigma^k$

$$S \Rightarrow_r^* \alpha X z x \Rightarrow_r \alpha \beta z x$$

und

$$\gamma \Rightarrow_r \alpha \beta z y$$

gelten mit  $\gamma \neq \alpha X z y$ , dann ist  $\gamma$  nicht von  $S$  aus ableitbar

- (2) Falls für ein  $z \in \Sigma^{<k}$

$$S \Rightarrow_r^* \alpha X z \Rightarrow_r \alpha \beta z$$

und

$$\gamma \Rightarrow_r \alpha \beta z$$

gelten mit  $\gamma \neq \alpha X z$ , dann ist  $\gamma$  nicht von  $S$  aus ableitbar

## LR( $k$ )-Grammatiken: Beispiele (2/2)

### Beispiel

$$S \rightarrow Cc \mid Dd$$

$$C \rightarrow Ca \mid \epsilon$$

$$D \rightarrow Da \mid \epsilon$$

- Die Grammatik erzeugt Strings der Form  $a^n c$  und  $a^n d$
- Ableitungen:
  - $S \Rightarrow_r Cc \Rightarrow_r^* Ca^n c \Rightarrow_r a^n c$
  - $S \Rightarrow_r Dd \Rightarrow_r^* Da^n d \Rightarrow_r a^n d$
- Der Parser müsste zuerst  $a^n c$  oder  $a^n d$  lesen, um zu wissen, ob als letzter Ableitungsschritt der Rechtsableitung die Regel  $C \rightarrow \epsilon$  oder  $D \rightarrow \epsilon$  angewendet werden muss
- ➡ Die Grammatik erfüllt für kein  $k$  die LR( $k$ )-Bedingung



# LL( $k$ )- und LR( $k$ )-Grammatiken: Eigenschaften

## Satz 11.3

- Für jedes  $k \geq 1$  lassen sich durch LL( $k+1$ )-Grammatiken mehr Sprachen beschreiben als durch LL( $k$ )-Grammatiken

## Satz 11.4

- Für jedes  $k \geq 1$  sind äquivalent:
  - $L$  ist deterministisch kontextfrei
  - $L = L(G)$  für eine LR( $k$ )-Grammatik  $G$

## Folgerung 11.5

- (a) Für jedes  $k \geq 1$  sind LR( $k$ )-Grammatiken und LR(1)-Grammatiken gleich ausdrucksstark
- (b) Das Syntaxanalyseproblem für LR( $k$ )-Grammatiken lässt sich in linearer Zeit lösen

- Außerdem gilt:

- LR(0)-Grammatiken entsprechen gerade den deterministischen Kellerautomaten, die mit leerem Keller akzeptieren
- Jede LL( $k$ )-Grammatik ist auch eine LR( $k$ )-Grammatik
- Aber: nicht zu jeder LR( $k$ )-Grammatik gibt es eine äquivalente LL( $k$ )-Grammatik
- LR( $k$ )-Grammatiken sind eindeutig

## Folgerung 11.6

- Jede deterministisch kontextfreie Sprache hat eine eindeutige Grammatik

- Weitere Informationen zur Syntaxanalyse mit LR( $k$ )-Grammatiken finden sich im Buch von Ingo Wegener
- Es gibt eine Vielzahl weiterer eingeschränkter kontextfreier Grammatiktypen, die für die Konstruktion von Compilern von Bedeutung sind
- Näheres (und natürlich sehr viel mehr) können Sie in der Vorlesung **Übersetzerbau** erfahren

# Zusammenfassung

- Das Wortproblem für kontextfreie Sprachen lässt sich mit dem CYK-Algorithmus in Zeit  $O(|G||w|^3)$  lösen
- Um die effiziente Syntaxanalyse von Programm-Code zu gewährleisten, ist es nötig, eingeschränkte Grammatiken zu verwenden
- Dabei gibt es zwei wichtige Ansätze:
  - Top-down: LL(**1**)-Grammatiken
  - Bottom-up: LR(**1**)-Grammatiken
- LL(**1**)-Grammatiken sind konzeptionell einfacher, aber LR(**1**)-Grammatiken sind ausdrucksstärker
- LR(**1**)-Grammatiken können genau die deterministisch kontextfreien Sprachen beschreiben
- Parser lassen sich zum Beispiel mit **yacc** automatisch aus kontextfreien Grammatiken erzeugen
- Dabei lässt sich im Falle eines (einfachen) Compilers sogar die Codeerzeugung integrieren
  - Zusammenspiel mit **lex**

# Literatur

- **CYK-Algorithmus:**

- Daniel H. Younger. Recognition and parsing of context-free languages in time  $n^3$ . *Information and Control*, 10(2):189–208, 1967

# Änderungslog

- 31.5.16:
  - Folie 29 korrigiert
- 10.8.16:
  - Folie 29 nochmals korrigiert ...

# Grundbegriffe der Theoretischen Informatik

Sommersemester 2016 - Thomas Schwentick

Teil C: Berechenbarkeit und Entscheidbarkeit

12: Verschiedene Berechnungsmodelle

Version von: 9. Juni 2016 (07:47)

# Inhalt

## ▷ **12.1 Einleitung in Teil C**

12.2 WHILE-Programme

12.3 GOTO-Programme

12.4 Turingmaschinen

# Ein einfaches algorithmisches Puzzle-Problem

- Wir betrachten zwei Varianten eines „Puzzle-Problems“
  - Eine wird sich als deutlich schwieriger als die andere herausstellen
- Einfache Variante des Puzzlespiels:
  - Gegeben: schwarze und gelbe Spielsteintypen mit Strings
  - Von jedem Steintyp stehen beliebig viele Steine zur Verfügung
  - Lässt sich das selbe Wort aus schwarzen wie aus gelben Spielsteinen legen?

## Beispiel

- Schwarze Steintypen: **01**, **10**, **011**
- Gelbe Steintypen: **101**, **00**, **11**
- Lösungswort: 1010011
  - ... in schwarz: **10** **10** **011**
  - ...und in gelb: **101** **00** **11**

## Def.: Pseudo-PCP

**Gegeben:** eine Folge  $(u_1, v_1), \dots, (u_k, v_k)$  von Paaren nicht-leerer Strings

**Frage:** Gibt es Indexfolgen  $i_1, \dots, i_n$  und  $j_1, \dots, j_m$  mit  $n \geq 1$ , so dass

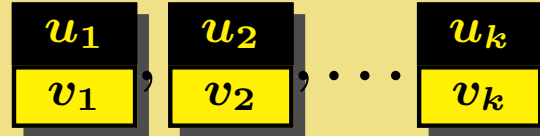
$$u_{i_1} u_{i_2} \cdots u_{i_n} = v_{j_1} v_{j_2} \cdots v_{j_m}?$$

- Die Frage ist also: Gibt es einen String  $w$ , der sowohl aus  $u_i$ s als auch aus  $v_j$ s zusammengesetzt werden kann
- Pseudo-PCP lässt sich mit Hilfe von Automaten in polynomieller Zeit entscheiden:
  - Konstruiere einen Automaten  $\mathcal{A}$ , der nichtleere Strings akzeptiert, die aus den  $u_i$  zusammengesetzt sind
  - Konstruiere einen Automaten  $\mathcal{B}$ , der nichtleere Strings akzeptiert, die aus den  $v_i$  zusammengesetzt sind
  - Teste ob  $L(\mathcal{A}) \cap L(\mathcal{B}) \neq \emptyset$

# Ein schwieriges algorithmisches Puzzle-Problem

- Jetzt betrachten wir die schwierigere Variante

- Gegeben eine Menge von Spielsteintypen



- Von jedem Typ stehen beliebig viele Steine zur Verfügung
- Lassen sich die Steine so in einer Reihe auslegen, dass das schwarze (obere) Wort gleich dem gelben (unteren) Wort ist?

## Beispiel

- Steintypen:
- Mögliche Lösung:

## Beispiel

- hat keine Lösung

Def.: Postsches Korrespondenzproblem (PCP)

**Gegeben:** eine Folge

$(u_1, v_1), \dots, (u_k, v_k)$  von  
Paaren nicht-leerer Strings

**Frage:** Gibt es eine Indexfolge  $i_1, \dots, i_n$   
mit  $n \geq 1$ , so dass

$$u_{i_1} u_{i_2} \cdots u_{i_n} = v_{i_1} v_{i_2} \cdots v_{i_n}?$$

- Die Frage ist also: Gibt es einen String  $w$ , der sowohl aus  $u_i$ s als auch aus  $v_i$ s zusammen gesetzt werden kann, und zwar **mit derselben nicht-leeren Indexfolge?**

- Wir nennen eine solche Indexfolge  $\vec{i} = i_1, \dots, i_n$  eine **Lösung** und den String  $u_{i_1} u_{i_2} \cdots u_{i_n}$  einen **Lösungsstring**



# Ein komplizierteres Beispiel

## Drittes Beispiel

- Steintypen: 

|     |
|-----|
| 001 |
| 0   |

, 

|     |
|-----|
| 01  |
| 011 |

, 

|     |
|-----|
| 01  |
| 101 |

, 

|     |
|-----|
| 10  |
| 001 |
- Kleinste Lösung:

|     |     |     |     |     |     |     |     |     |     |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 01  | 10  | 01  | 10  | 10  | 01  | 001 | 01  | 10  | 01  |
| 011 | 001 | 101 | 001 | 001 | 011 | 0   | 011 | 001 | 101 |

|     |     |     |     |     |     |     |     |     |     |     |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 10  | 01  | 10  | 10  | 01  | 10  | 10  | 01  | 001 | 10  | 10  |
| 001 | 101 | 001 | 001 | 101 | 001 | 001 | 011 | 0   | 001 | 001 |

|     |     |     |     |     |     |     |     |     |     |     |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 01  | 001 | 01  | 10  | 001 | 001 | 01  | 10  | 10  | 10  | 01  |
| 011 | 0   | 101 | 001 | 0   | 0   | 101 | 001 | 001 | 001 | 011 |

|     |     |     |     |     |     |     |     |     |     |     |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 001 | 01  | 001 | 001 | 001 | 01  | 10  | 01  | 10  | 001 | 01  |
| 0   | 011 | 0   | 0   | 0   | 101 | 001 | 101 | 001 | 0   | 011 |

|     |     |     |     |     |     |     |     |     |     |     |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 001 | 10  | 10  | 01  | 001 | 10  | 001 | 001 | 01  | 10  | 001 |
| 0   | 001 | 001 | 011 | 0   | 001 | 0   | 0   | 101 | 001 | 0   |

|     |     |     |     |     |     |     |     |     |     |     |     |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 001 | 01  | 001 | 001 | 01  | 001 | 01  | 001 | 10  | 001 | 001 | 01  |
| 0   | 101 | 0   | 0   | 101 | 0   | 011 | 0   | 001 | 0   | 0   | 101 |

# PCP ist algorithmisch nicht lösbar

- Wir werden in den nächsten Kapiteln den folgenden Satz beweisen

## Satz

- PCP ist nicht entscheidbar
- Dazu benötigen wir einige Vorbereitung
- Zunächst müssen wir den Begriff „entscheidbar“ definieren
- Informell soll ein algorithmisches Problem entscheidbar sein, wenn es einen Algorithmus gibt, der bei jeder Eingabe anhält und immer die richtige Antwort gibt
- Um dies zu formalisieren benötigen wir eine Definition von „Algorithmus“
- Um zu definieren, was ein Algorithmus ist, benötigen wir ein allgemeines „Berechnungsmodell“
- Damit unsere Definition nicht zu modellspezifisch wird, ziehen wir lieber mehrere Berechnungsmodelle in Betracht

# Inhalt

12.1 Einleitung in Teil C

▷ **12.2 WHILE-Programme**

12.3 GOTO-Programme

12.4 Turingmaschinen

# Übersicht

- Wir suchen Antworten auf folgende Fragen
  - Was ist ein Algorithmus?
  - Wann ist eine Funktion berechenbar?
- Wir betrachten dazu verschiedene Berechnungsmodelle
- Zwei Modelle, die von Programmiersprachen inspiriert sind:
  - WHILE-Programme
  - GOTO-Programme
- Ein Modell, das als mächtige Erweiterung der endlichen Automaten aufgefasst werden kann, ursprünglich aber als mathematische Formalisierung des „Rechnens mit Papier und Bleistift“ gedacht war:
  - Turingmaschinen
- Später betrachten wir noch ein Modell, das durch rekursive Definitionen inspiriert ist:
  - $\mu$ -rekursive Funktionen
- Im nächsten Kapitel werden wir die Mächtigkeit dieser Berechnungsmodelle miteinander vergleichen

# Partielle Funktionen

- Wir werden jetzt häufig partielle Funktionen über den natürlichen Zahlen verwenden
- Bei **partiellen Funktionen**  $f$  muss der Funktionswert nicht für alle Elemente der Grundmenge definiert sein
- Notation:
  - $f : \mathbb{N}_0 \rightarrow \mathbb{N}_0$
  - Um auszudrücken, dass  $f(n)$  undefiniert ist, schreiben wir:  $f(n) = \perp$

## Beispiel

- Sei sqrt die partielle Funktion  $\mathbb{N}_0 \rightarrow \mathbb{N}_0$ , die jeder natürlichen Zahl  $n$  die natürliche Zahl  $m$  mit  $m^2 = n$  zuordnet, wenn ein solches  $m$  existiert
- Es gilt also z.B.:
  - $\text{sqrt}(9) = 3$
  - $\text{sqrt}(10) = \perp$

## Definition

- Sei  $f : \mathbb{N}_0 \rightarrow \mathbb{N}_0$  eine partielle Funktion
  - Der **Definitionsbereich**  $D(f)$  einer partiellen Funktion  $f$  ist
$$\{n \in \mathbb{N}_0 \mid f(n) \neq \perp\}$$
  - Der **Wertebereich**  $W(f)$  einer partiellen Funktion  $f$  ist
$$\{n \in \mathbb{N}_0 \mid \exists m \in \mathbb{N}_0 : f(m) = n\}$$

- Eine **totale** Funktion  $\mathbb{N}_0 \rightarrow \mathbb{N}_0$  ist eine Funktion, die für alle natürlichen Zahlen definiert ist



Zu beachten:

- Der Begriff „partielle Funktion“ erzwingt nicht, dass es Zahlen gibt, für die kein Funktionswert existiert
- Jede totale Funktion ist also auch eine partielle Funktion



Wir werden auch partielle Funktionen über Strings betrachten

# WHILE-Programme: Beispiele

## Beispiel

```
 $x_1 := x_2;$   
WHILE  $x_3 \neq 0$  DO  
     $x_1 := x_1 + 1;$   
     $x_3 := x_3 \div 1$   
END
```

- Variablen nehmen in WHILE-Programmen nur Werte aus  $\mathbb{N}_0$  an
- Der Effekt des Programmes ist also:

$x_1 := x_2 + x_3$   
☞ Und:  $x_3 := 0$

## Beispiel

```
 $x_1 := 0;$   
WHILE  $x_3 \neq 0$  DO  
     $x_4 := x_2;$   
    WHILE  $x_4 \neq 0$  DO  
         $x_1 := x_1 + 1;$   
         $x_4 := x_4 \div 1$   
    END;  
     $x_3 := x_3 \div 1$   
END
```

- Der (wesentliche) Effekt des Programmes ist:  $x_1 := x_2 \times x_3$

## Beispiel

```
 $x_1 := 1;$   
WHILE  $x_1 \neq 0$  DO  
     $x_3 := x_2 + 2$   
END
```

- Dieses Programm hält nie an...

# WHILE-Programme: Syntax

- **WHILE-Programme** verwenden die folgenden syntaktischen Grundelemente:
  - Variablen:  $x_1, x_2, x_3, \dots$
  - Konstanten:  $0, 1, 2, \dots$
  - Trennsymbole:  $;$   $:=$
  - Operationszeichen:  $+$   $\div$
  - Schlüsselwörter: WHILE, DO, END

## Definition

- Die **Syntax von WHILE-Programmen** ist wie folgt definiert:

### Wertzuweisung:

- $x_i := c$
- $x_i := x_j$
- $x_i := x_j + c$
- $x_i := x_j \div c$

sind WHILE-Programme

(für jede Konstante  $c$  und  $i, j \geq 1$ )

**Reihung:** Falls  $P_1$  und  $P_2$  WHILE-Programme sind, so auch  $P_1; P_2$

### (Bedingte Wiederholung)

Ist  $P$  ein WHILE-Programm, so auch

WHILE  $x_i \neq 0$  DO  $P$  END

# WHILE-Programme: Semantik (1/2)

- Wir definieren die Semantik von WHILE-Programmen durch ihre Wirkung auf Speicherinhalte
- Dabei modellieren wir Speicherinhalte durch Funktionen  $X$ , die die Werte der Variablen  $x_1, x_2, \dots$  repräsentieren
  - $X[i]$  repräsentiert den Wert von  $x_i$

## Definition

- Ein Speicherinhalt  $X$  ist eine Funktion  $\mathbb{N} \rightarrow \mathbb{N}_0$ , für die  $X[i] \neq 0$  nur für endlich viele  $i \in \mathbb{N}$  gilt
- Der initiale Speicherinhalt  $X_{\text{init}}^b$  bei Eingabe  $b \in \mathbb{N}_0$  ist definiert durch:

$$X_{\text{init}}^b[i] \stackrel{\text{def}}{=} \begin{cases} b & \text{für } i = 1 \\ 0 & \text{sonst} \end{cases}$$

- ✎ Wir schreiben hier  $X[i]$  statt  $X(i)$  um später die Unterscheidung zu anderen (runden) Klammern zu erleichtern

- ✎ Manchmal beschreiben wir Speicherinhalte als Folge von Zahlen  $X[1], X[2], X[3], \dots$ 
  - In dieser Sichtweise wird der initiale Speicherinhalt  $X_{\text{init}}^b$  bei Eingabe  $b$  durch die Folge  $b, 0, 0, \dots$  repräsentiert



# WHILE-Programme: Semantik (2/2)

## Definition

- Ist  $X$  ein Speicherinhalt und  $P$  ein WHILE-Programm, so bezeichne  $P(X)$  den Speicherinhalt nach Bearbeitung von  $P$
- $P(X)$  ist induktiv wie folgt definiert
- Falls  $P$  von der Form  $x_i := x_j + c$  ist:  

$$P(X)[k] \stackrel{\text{def}}{=} \begin{cases} X[j] + c & \text{für } k = i \\ X[k] & \text{sonst} \end{cases}$$
  - Analog für  $x_i := c$  und  $x_i := x_j$
- Falls  $P$  von der Form  $x_i := x_j \div c$  ist:  

$$P(X)[k] \stackrel{\text{def}}{=} \begin{cases} \max(X[j] - c, 0) & \text{für } k = i \\ X[k] & \text{sonst} \end{cases}$$
- Falls  $P$  von der Form  $P_1; P_2$  ist:  


$$P(X) \stackrel{\text{def}}{=} P_2(P_1(X))$$

## Definition (Forts.)

- Ist  $P$  von der Form  

$$\text{WHILE } x_i \neq 0 \text{ DO } P_1 \text{ END}$$
und  $X$  ein Speicherinhalt, so sei  

$$P(X) \stackrel{\text{def}}{=} \begin{cases} X & \text{falls } X[i] = 0 \\ P(P_1(X)) & \text{sonst} \end{cases}$$
  - Die durch ein WHILE-Programm  $P$  berechnete Funktion  $f_P$  ist wie folgt definiert:
    - Für jedes  $b \in \mathbb{N}_0$  ist  

$$f_P(b) \stackrel{\text{def}}{=} P(X_{\text{Init}}^b)[1]$$
  - Eine Funktion  $f : \mathbb{N}_0 \rightarrow \mathbb{N}_0$  heißt WHILE-berechenbar, falls  $f = f_P$  für ein WHILE-Programm  $P$
-  Der durch das Programm berechnete Funktionswert entspricht also dem Inhalt von  $x_1$  nach Ende der Berechnung

# WHILE-Programme: Bemerkungen

- Intuitive Bedeutung der Semantik der bedingten Wiederholung:
  - Solange  $X[i] \neq 0$  gilt, wird  $P_1$  ausgeführt
- Zu beachten:
  - Die Semantik-Definition ist rekursiv
  - $P(X)$  ist nicht für alle  $P$  und  $X$  definiert

- Auch mehrstellige Funktionen lassen sich durch WHILE-Programme berechnen:
  - Für jedes  $k$ -Tupel  $\vec{a} = (a_1, \dots, a_k)$  sei  $f_{P,k}(\vec{a})$  der Wert von  $X'(1)$ , wobei
    - \*  $X' = P(X_{\text{Init}}^{\vec{a}})$
    - \* und  $X_{\text{Init}}^{\vec{a}}$  der Folge  $a_1, \dots, a_k, 0, 0, \dots$  entspricht

- Wir haben schon gesehen, dass WHILE-Programme verschiedene Konstrukte simulieren können:
  - Addition zweier Variablen
  - Produkt zweier Variablen
- Es ist nicht schwer zu sehen, dass mit WHILE-Programmen auch bedingte Anweisungen der Art
$$\text{IF } x_1 = c \text{ THEN } P \text{ END}$$
simuliert werden können

- Wir werden im Folgenden solche Anweisungen als „syntaktischen Zucker“ erlauben
- WHILE-Programme im Sinne der formalen Definition nennen wir im Folgenden „einfache WHILE-Programme“
- Der Begriff „WHILE-berechenbar“ bezieht sich auf einfache WHILE-Programme

# Inhalt

12.1 Einleitung in Teil C

12.2 WHILE-Programme

▷ **12.3 GOTO-Programme**

12.4 Turingmaschinen

# GOTO-Programme: Syntax


## Definition

- Ein GOTO-Programm besteht aus einer Folge
  - $M_1 : A_1;$
  - $M_2 : A_2;$
  - $\vdots$
  - $M_k : A_k$von Anweisungen  $A_i$  mit Sprungmarken  $M_i$
- Mögliche Anweisungen:
  - **Wertzuweisung:**
    - \*  $x_i := c$
    - \*  $x_i := x_j$
    - \*  $x_i := x_j + c$
    - \*  $x_i := x_j \div c$
  - **Bedingter Sprung:**  
IF  $x_i = c$  THEN GOTO  $M_j$
  - **Stopanweisung:** HALT


## Beispiel

```
1 :  $x_4 := 1;$ 
2 :  $x_1 := x_2;$ 
3 : IF  $x_3 = 0$  THEN GOTO 7;
4 :  $x_1 := x_1 + 1;$ 
5 :  $x_3 := x_3 \div 1;$ 
6 : IF  $x_4 = 1$  THEN GOTO 3;
7 : HALT
```

- Intuitiv hat dieses Programm den Effekt:  
 $x_1 := x_2 + x_3$   
(und Seiteneffekte für  $x_3$  und  $x_4$ )

 Das Beispiel illustriert, dass sich unbedingte Sprünge durch bedingte Sprünge simulieren lassen

- Wir erlauben im Folgenden deshalb auch unbedingte Sprünge GOTO  $M_j$  als syntaktischen Zucker

 Sprungmarken, die nicht als Sprungadresse dienen, lassen wir oft weg

# GOTO-Programme: Semantik

## Definition

- Eine Konfiguration eines GOTO-Programmes  $P$  ist ein Paar  $(M; X)$ , wobei  $M$  eine Sprungmarke von  $P$  und  $X$  ein Speicherinhalt ist
- Start-Konfiguration bei Eingabe  $b$ :  

$$(M_1; X_{\text{init}}^b)$$
- Ist  $M$  eine Sprungmarke eines GOTO-Programms, so bezeichnet  $M + 1$  die Sprungmarke der folgenden Zeile
- Die Nachfolge-Konfiguration  $(M'; X')$  einer Konfiguration  $(M_\ell; X)$  ist wie folgt definiert:
  - Ist  $A_\ell$  eine Wertzuweisung, so ist  $M' \stackrel{\text{def}}{=} M_\ell + 1$  und  $X'$  ist definiert wie bei WHILE-Programmen
  - Falls  $A_\ell$  ein bedingter Sprung  

$$\text{IF } x_i = c \text{ THEN GOTO } M_j$$
ist, so ist  $X' \stackrel{\text{def}}{=} X$  und
$$M' \stackrel{\text{def}}{=} \begin{cases} M_j & \text{falls } X[i] = c \\ M_\ell + 1 & \text{sonst} \end{cases}$$

## Definition (Forts.)

- Halte-Konfiguration:  $(M_k + 1; X)$  oder  $(M_\ell; X)$  und  $A_\ell$  ist HALT
- Die Berechnung von  $P$  bei Eingabe  $b$  ist die eindeutig bestimmte Folge  $K_1, K_2, \dots$  von Konfigurationen mit:
  - $K_1 = (M_1, X_{\text{init}}^b)$  und
  - jede Konfiguration  $K_i$  ist die Nachfolgekonfiguration von  $K_{i-1}$  (für  $i > 1$ )
- Falls die Berechnung von  $P$  bei Eingabe  $b$  endlich ist, so ist die letzte Konfiguration eine Halte-Konfiguration  $(M; X)$
- Dann sei wieder:  $f_P(b) \stackrel{\text{def}}{=} X[1]$
- Falls die Berechnung von  $P$  bei Eingabe  $b$  unendlich ist, so ist  $f_P(b) \stackrel{\text{def}}{=} \perp$ ,
- Eine Funktion  $f : \mathbb{N}_0 \rightarrow \mathbb{N}_0$  heißt GO-TO-berechenbar, falls  $f = f_P$  für ein GOTO-Programm  $P$

# Inhalt

12.1 Einleitung in Teil C

12.2 WHILE-Programme

12.3 GOTO-Programme

▷ **12.4 Turingmaschinen**

# Warum Turingmaschinen?

- Bisher haben wir Berechnungsmodelle betrachtet, die sich an Programmiersprachen anlehnen:  
WHILE- und GOTO-Programme

- Jetzt betrachten wir ein Berechnungsmodell, das „menschliche Rechner“ zum Vorbild nimmt
- Dieses Modell wurde 1936 von Alan Turing „erfunden“

- Warum hat jemand
  - etliche Jahre vor dem Bau des ersten Computers und
  - Jahrzehnte vor der Entwicklung „richtiger Programmiersprachen“ein abstraktes Berechnungsmodell erfunden?

- Turing war Mathematiker und wollte beweisen, dass es kein automatisches Verfahren gibt, das zu jeder mathematischen Aussage entscheidet, ob sie wahr oder falsch ist

- Etwas anders formuliert, wollte er zeigen, dass es keinen Algorithmus für das folgende algorithmische Problem gibt

Definition: Allgemeingültigkeitsproblem

**Gegeben:** Prädikatenlogische Formel  $\varphi$

**Frage:** Gilt für alle passenden Modelle  $\mathcal{M}$ :

$$\mathcal{M} \models \varphi?$$

## Turings Ideen zur Berechenbarkeit (1/4)

- Wie gesagt: zu Turings Zeit gab es noch keine künstlichen programmierbaren Rechner
- Wenn er von einem **Computer** sprach, meinte er einen Menschen, der nach einem festgelegten Verfahren etwas berechnet
- Seine Vorstellungen, wie ein solcher *Computer* arbeitet, hat er in der folgenden Arbeit beschrieben:
  - A. M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proc. London Math. Soc.*, 2(42):230–265, 1936
- Die Abstraktion des Computers, die Turing in dieser Arbeit definierte, wird heute **Turingmaschine** genannt
- Schauen wir mal, welche Gedanken sich Alan Turing 1936 so gemacht hat...



## Turings Ideen zur Berechenbarkeit (2/4)

- Computing is normally done by writing certain symbols on paper
- We may suppose this paper is divided into squares like a child's arithmetic book
- In elementary arithmetic the two-dimensional character of the paper is sometimes used
- But such a use is always avoidable, and I think that it will be agreed that the two-dimensional character of paper is no essential of computation
- I assume then that the computation is carried out on one-dimensional paper, i.e. on a tape divided into squares
- I shall also suppose that the number of symbols which may be printed is finite
  - If we were to allow an infinity of symbols, then there would be symbols differing to an arbitrarily small extent

## Turings Ideen zur Berechenbarkeit (3/4)

- The **behaviour** of the computer at any moment is **determined by the symbols which he is observing and his "state of mind"** at that moment
- We may suppose that there is a **bound  $B$**  to the **number of symbols** or squares **which the computer can observe** at one moment
- If he wishes to observe more, he must use successive observations
- We will also suppose that the **number of states** of mind which need be taken into account **is finite**
- Let us imagine the operations performed by the computer to be split up into "**simple operations**" which are so elementary that it is not easy to imagine them further divided
- Every such operation consists of some change of the physical system consisting of the computer and his tape
- We know the state of the system if we know the sequence of symbols on the tape, which of these are observed by the computer (possibly with a special order), and the state of mind of the computer
- We may suppose that **in a simple operation not more than one symbol is altered**
  - Any other changes can be set up into simple changes of this kind
- The situation in regard to the squares whose symbols may be altered in this way is the same as in regard to the observed squares
- We may, therefore, without loss of generality, **assume that the squares whose symbols are changed are always "observed" squares**

## Turings Ideen zur Berechenbarkeit (4/4)


- Besides these changes of symbols, the simple operations must include changes of distribution of observed squares
  - The new observed squares must be immediately recognisable by the computer
  - I think it is reasonable to suppose that they can only be squares whose distance from the closest of the immediately previously observed squares does not exceed a certain fixed amount
  - Let us say that each of the new observed squares is within  $L$  squares of an immediately previously observed square
- The most general single operation must therefore be taken to be one of the following:
    - (A) A possible change (a) of symbol together with a possible change of state of mind
    - (B) A possible change (b) of observed squares, together with a possible change of state of mind
  - The operation actually performed is determined, as has been suggested on p.250, by the state of mind of the computer and the observed symbols
  - In particular, they determine the state of mind of the computer after the operation is carried out

# Vom Automaten zur Turingmaschine

- Turingmaschinen können als Erweiterung von endlichen Automaten in drei Stufen aufgefasst werden:

## (1) Mehr Bewegung:

- Der Kopf darf sich nach rechts **und** nach links bewegen

 Endliche Automaten mit dieser Erweiterung können nur reguläre Sprachen entscheiden

## (2) Schreiben:

- Die Symbole der Eingabe können verändert werden — jeweils an der Position des Kopfes

- Das Berechnungsmodell, das endliche Automaten um (1) und (2) erweitert, wird **linear beschränkte Automaten** genannt
  - Sie entscheiden genau die kontextsensitiven Sprachen

## (3) Mehr Platz:

- Der Arbeitsbereich kann über die Eingabe hinaus erweitert werden (nach rechts)
- Links von der Eingabe steht ein Symbol ( $\triangleright$ ), das den linken Rand markiert, der nicht verschoben werden kann
- Berechnungen enden nicht mehr durch Verlassen der Eingabe sondern durch Erreichen spezieller **Endzustände**

- Wir betrachten nun zunächst Beispiele von Turingmaschinen

# Turingmaschinen: 1. Beispiel

## Beispiel

Turingmaschine zum Test, ob die Eingabe von der Form  $ww^R$  ist

**Idee:** Vergleiche jeweils das erste mit dem letzten Symbol und lösche beide (durch Überschreiben mit  $\#$  bzw.  $\sqcup$ )

**a :** 0 und 1 überlesen, nach links — falls  $\triangleright$  oder  $\#$  nach rechts in **b**

**b:** Falls 0 nach rechts in **c** — falls 1 nach rechts in **d** (dabei 0/1 durch  $\#$  überschreiben) — falls  $\sqcup$  akz.

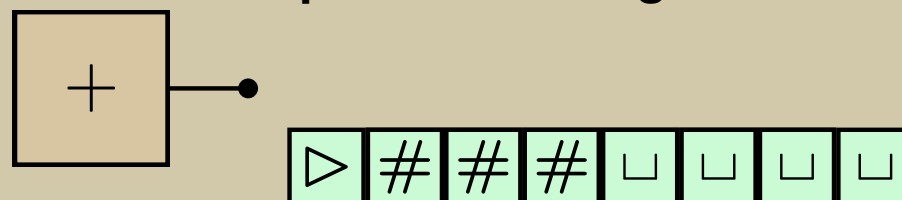
**c:** 0 und 1 überlesen nach rechts bis  $\sqcup$ , dann nach links in **e**

**d:** 0 und 1 überlesen nach rechts bis  $\sqcup$ , dann nach links in **f**

**e:** Falls 0 durch  $\sqcup$  ersetzen nach links in **a** — falls 1 oder  $\#$  ablehnen

**f:** Falls 1 durch  $\sqcup$  ersetzen nach links in **a** — falls 0 oder  $\#$  ablehnen

### 1. Beispielberechnung:



# Turingmaschinen: 1. Beispiel, 2. Berechnung

## Beispiel

Turingmaschine zum Test, ob die Eingabe von der Form  $ww^R$  ist

**a** : 0 und 1 überlesen, nach links — falls  $\triangleright$  oder  $\#$  nach rechts in **b**

**b**: Falls 0 nach rechts in **c** — falls 1 nach rechts in **d** (dabei 0/1 durch  $\#$  überschreiben) — falls  $\sqcup$  akz.

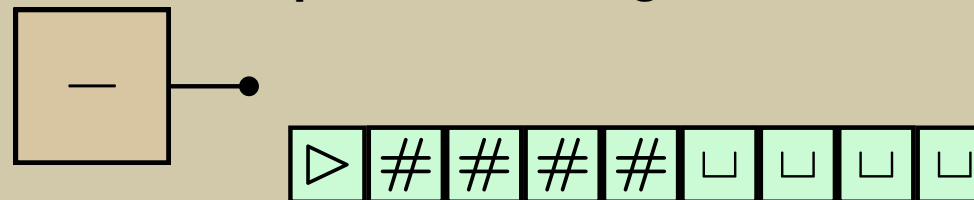
**c**: 0 und 1 überlesen nach rechts bis  $\sqcup$ , dann nach links in **e**

**d**: 0 und 1 überlesen nach rechts bis  $\sqcup$ , dann nach links in **f**

**e**: Falls 0 durch  $\sqcup$  ersetzen nach links in **a** — falls 1 oder  $\#$  ablehnen

**f**: Falls 1 durch  $\sqcup$  ersetzen nach links in **a** — falls 0 oder  $\#$  ablehnen

## 2. Beispielberechnung:



# Turingmaschinen: Definition

## Definition

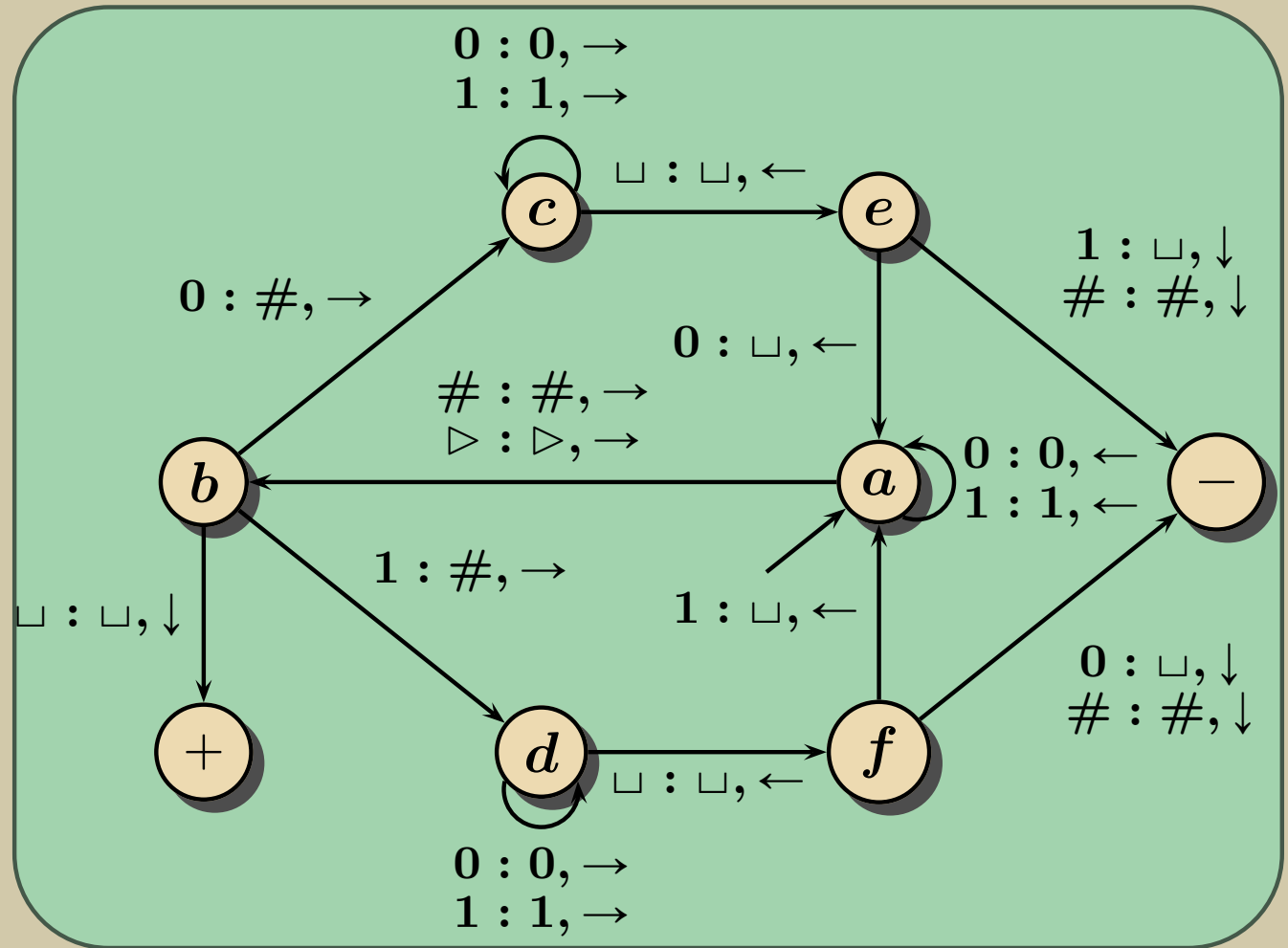
- Eine Turingmaschine  $M = (Q, \Gamma, \delta, s)$  besteht aus
  - einer endliche Menge  $Q$ , (Zustandsmenge)
  - einem Alphabet  $\Gamma$ , mit  $\sqcup, \triangleright \in \Gamma$ , (Arbeitsalphabet)
  - einer Funktion
$$\delta : Q \times \Gamma \rightarrow (Q \cup \{\text{ja, nein}, h\}) \times \Gamma \times \{\leftarrow, \downarrow, \rightarrow\}$$
(Transitionsfunktion),
  - einem ausgezeichneten Zustand  $s \in Q$  (Startzustand)
- Dabei seien  $Q, \Gamma, \{h, \text{ja, nein}\}$  und  $\{\leftarrow, \downarrow, \rightarrow\}$  paarweise disjunkt
- Turingmaschinen müssen außerdem die folgenden Bedingungen erfüllen:
  - Das Symbol  $\triangleright$  für den linken Rand darf nicht überschrieben werden
  - von  $\triangleright$  darf sich der Kopf nicht nach links bewegen
- Das lässt sich dadurch erreichen, dass  $\delta(q, \triangleright)$  immer von der Form  $(q', \triangleright, d)$  mit  $d \in \{\downarrow, \rightarrow\}$  ist

# Turingmaschinen: Diagramm-Darstellung

## Zustände der Beispiel-TM

- a** : 0 und 1 überlesen, nach links — falls  $\triangleright$  oder  $\#$  nach rechts in **b**
- b**: Falls 0 nach rechts in **c** — falls 1 nach rechts in **d** (da bei 0/1 durch  $\#$  überschreiben) — falls  $\sqcup$  akz.
- c**: 0 und 1 überlesen nach rechts bis  $\sqcup$ , dann nach links in **e**
- d**: 0 und 1 überlesen nach rechts bis  $\sqcup$ , dann nach links in **f**
- e**: Falls 0 durch  $\sqcup$  ersetzen nach links in **a** — falls 1 oder  $\#$  ablehnen
- f**: Falls 1 durch  $\sqcup$  ersetzen nach links in **a** — falls 0 oder  $\#$  ablehnen

## Beispiel-Turingmaschine als Diagramm



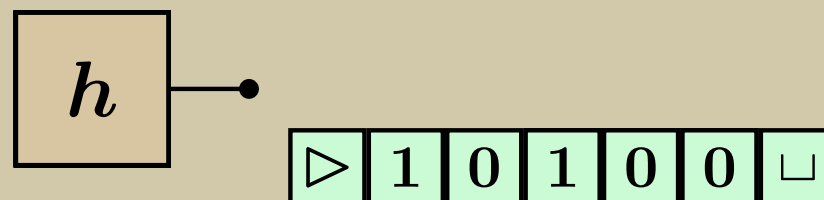
- Konvention: ist für ein Paar  $(q, \sigma) \in Q \times \Gamma$  kein Übergang eingezeichnet, so sei  $\delta(q, \sigma) \stackrel{\text{def}}{=} (\text{nein}, \sigma, \downarrow)$



# Turingmaschinen: 2. Beispiel

## 2. Beispiel-TM: Inkrementieren einer Binärzahl

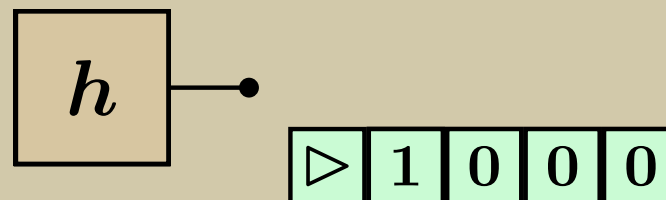
- Beschreibung der Zustände:
  - a**: Die TM läuft, ohne etwas zu verändern, nach rechts bis zum ersten Leerzeichen, und dann einen Schritt nach links in den Zustand **b**.
  - b**: Das maximale Suffix der Form  $1^i$  wird durch  $0^i$  ersetzt.
    - Ist davor eine **0**, so wird sie durch **1** ersetzt und **M** geht in den Zustand **c**
      - ✎ Das Suffix der Eingabe ist von der Form  $01^i$  und wird durch  $10^i$  ersetzt
    - Ist davor ein  $\triangleright$ , so geht **M** in Zustand **c**
      - ✎ Die Eingabe war von der Form  $1^i$ , wurde in  $0^i$  geändert, und muss noch in  $10^i$  umgewandelt werden
  - c**: Ersetzt die erste **0** durch eine **1**, und geht in den Zustand **d**
  - d**: Fügt eine **0** hinten an und geht in den Zustand **e**
  - e**: läuft zum linken Rand und geht in den Endzustand **h**



# Turingmaschinen: 2. Beispiel, 2. Berechnung

## 2. Beispiel-TM: Inkrementieren einer Binärzahl (Forts.)

- Turingmaschine zum Inkrementieren einer Binärzahl:
- Beschreibung der Zustände:
  - a***: Die TM läuft, ohne etwas zu verändern, nach rechts bis zum ersten Leerzeichen, und dann einen Schritt nach links in den Zustand ***b***.
  - b***: Das maximale Suffix der Form  $1^i$  wird durch  $0^i$  ersetzt.
    - Ist davor eine **0**, so wird sie durch **1** ersetzt und ***M*** geht in den Zustand ***e***
    - Ist davor ein  $\triangleright$ , so geht ***M*** in Zustand ***c***
  - c***: Ersetzt die erste **0** durch eine **1**, und geht in den Zustand ***d***
  - d***: Fügt eine **0** hinten an und geht in den Zustand ***e***
  - e***: läuft zum linken Rand und geht in den Endzustand ***h***



## Turingmaschinen: 2. Beispiel als Diagramm

### Beispiel: Zustände der 2. TM

**a:** Laufe nach rechts bis zum ersten  $\sqcup$ , dann einen Schritt nach links in Zustand **b**.

**b:** Gehe nach links bis zur nächsten **0** — ersetze dabei jede **1** durch **0**

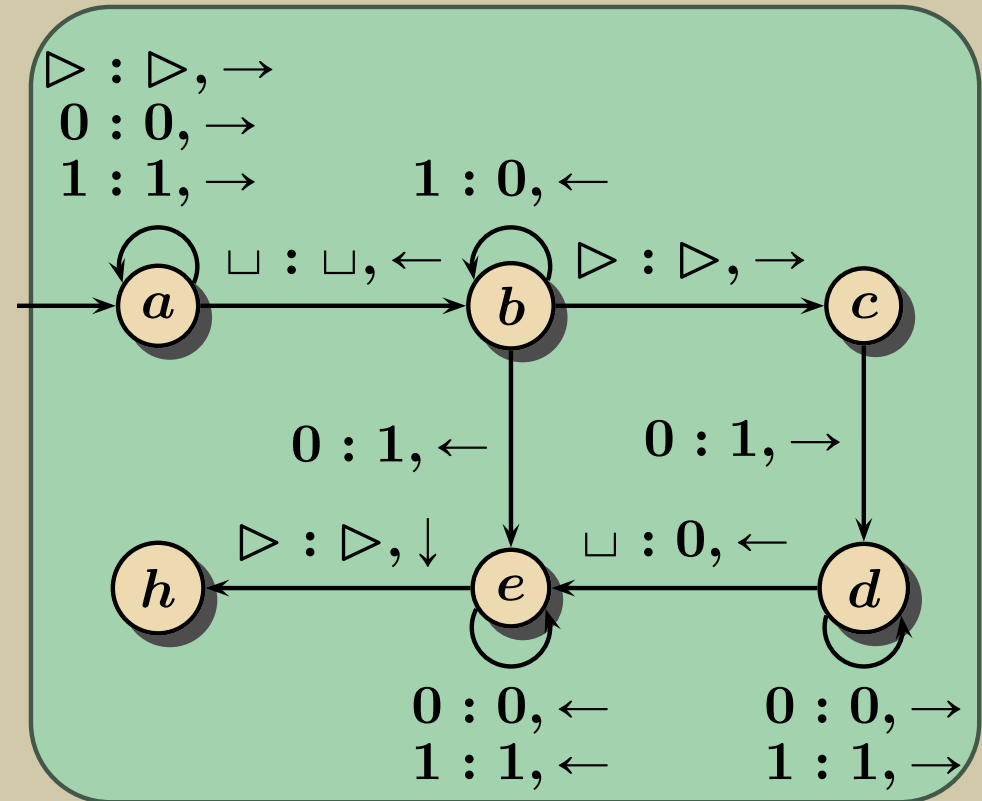
- Ist davor **0**, ersetze durch **1** und gehe in Zustand **e**
- Ist davor  $\triangleright$ , gehe in Zustand **c**

**c:** Ersetze die erste **0** durch eine **1**, und gehe in den Zustand **d**

**d:** Fügt eine **0** hinten an und gehe in den Zustand **e**

**e:** Laufe zum linken Rand und gehe in den Endzustand **h**

### Diagramm zur 2. TM



# Turingmaschinen: Konfigurationen (1/2)

- Um die aktuelle Situation einer TM zu beschreiben verwenden wir Konfigurationen, bestehend aus einem Zustand und einer String-Zeigerbeschreibung

## Definition

- Eine String-Zeigerbeschreibung  $(w, z)$  besteht aus
  - einem String  $w \in \Gamma^*$
  - und einer Zeigerposition  $z \in \mathbb{N}_0, z \leq |w|$

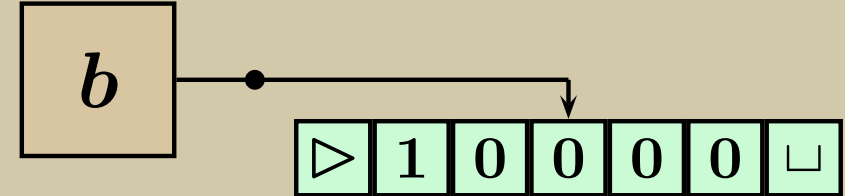
📌 Der linke Rand wird in  $w$  nicht repräsentiert!

- Wir vereinbaren:  $w[0] \stackrel{\text{def}}{=} \triangleright$

- Manchmal verwenden wir eine andere Notation und schreiben  $(u, \sigma, v)$  statt  $(w, z)$ , falls
  - $w = u\sigma v$ ,
  - $|u\sigma| = z$
- $\sigma$  ist dann das Zeichen, auf das der Zeiger zeigt,  $v$  ist der String rechts vom Zeiger,  $u$  ist der String links vom Zeiger (ohne  $u[0] = \triangleright$ )
- Falls  $z = 0$ :  $(\epsilon, \epsilon, w)$

## Beispiel

Die String-Zeigerbeschreibung zu



ist

- $(10000\sqcup, 3)$  oder
- $(10, 0, 00\sqcup)$

## Definition

- Eine Konfiguration von  $M$  ist ein Tupel  $(q, (w, z))$  mit
  - $q \in Q \cup \{\text{ja, nein, } h\}$   
(aktueller Zustand)
  - $w$  der aktuelle String
  - $z$  die Position des Zeigers der TM (linker Rand ist 0)

# Turingmaschinen: Konfigurationen (2/2)

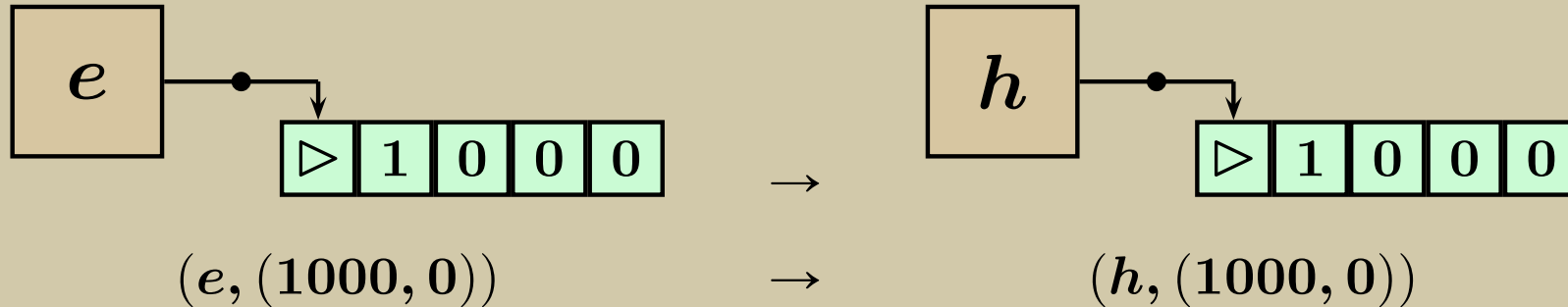
## Definition

- Sei  $K = (q, (w, z))$  eine Konfiguration mit  $w[z] = \sigma$  und sei  $\delta(q, \sigma) = (q', \tau, d)$  mit  $\tau \in \Gamma, d \in \{\leftarrow, \downarrow, \rightarrow\}$
- Dann ist die Nachfolgekonfiguration  $K' = (q', (w', z'))$  von  $K$  wie folgt definiert: (Schreibweise:  $K \vdash_M K'$ )
  - $z' = z + 1$ , falls  $d = \rightarrow$ ,
  - $z' = z$ , falls  $d = \downarrow$ ,
  - $z' = z - 1$ , falls  $d = \leftarrow$ ,
  - $w' = w$ , falls  $z = 0$
  - $w' = w[z/\tau] \sqcup$ , falls  $z = |w|$  und  $d = \rightarrow$ ,
  - $w' = w[z/\tau]$ , andernfalls
- Dabei bezeichnet  $w[z/\tau]$  den String, der aus  $w$  entsteht, indem das Zeichen an Position  $z$  durch  $\tau$  ersetzt wird
- Wir schreiben  $K \vdash_M^* K'$ , falls es Konfigurationen  $K_1, \dots, K_m$  gibt mit  $K \vdash_M K_1 \vdash_M \dots \vdash_M K_m \vdash_M K'$

# Turingmaschinen: Illustration der Nachfolgekonfiguration (1/2)

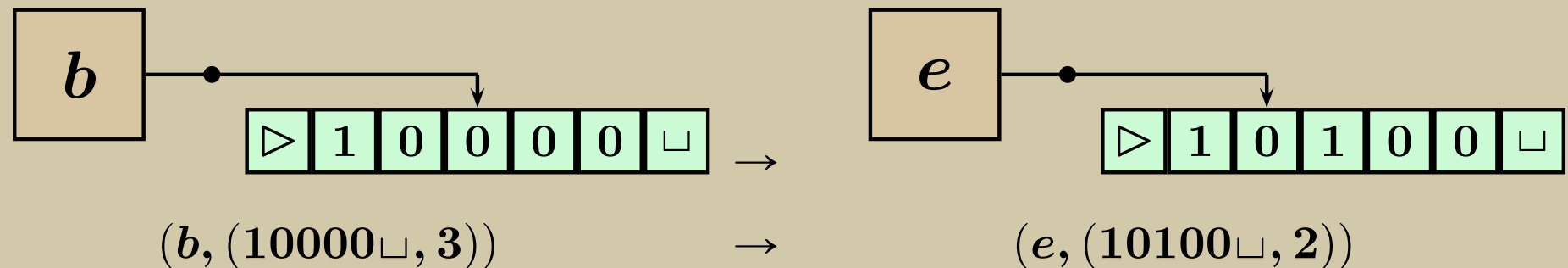
## Beispiel: Schritt ohne Kopf-Bewegung

- $\delta(e, \triangleright) = (h, \triangleright, \downarrow)$



## Beispiel: Linksschritt

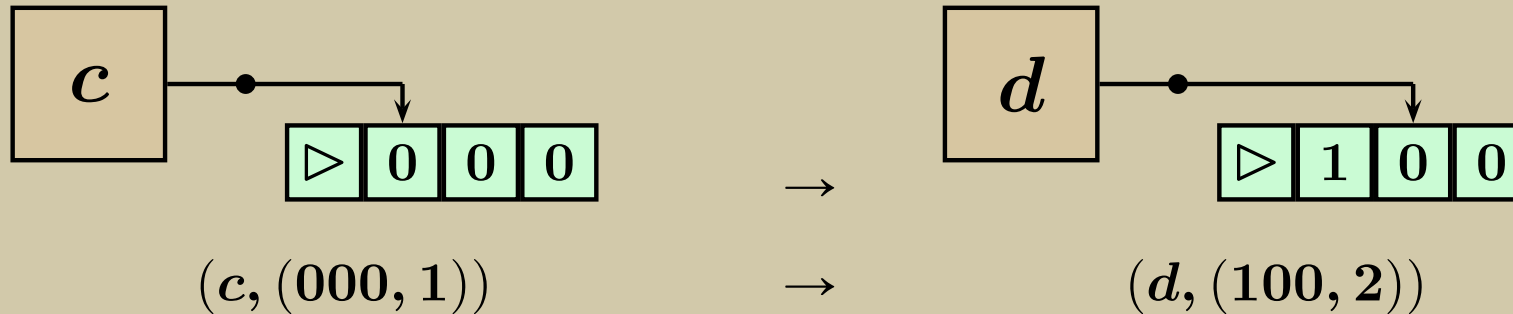
- $\delta(b, 0) = (e, 1, \leftarrow)$



# Turingmaschinen: Illustration der Nachfolgekonfiguration (2/2)

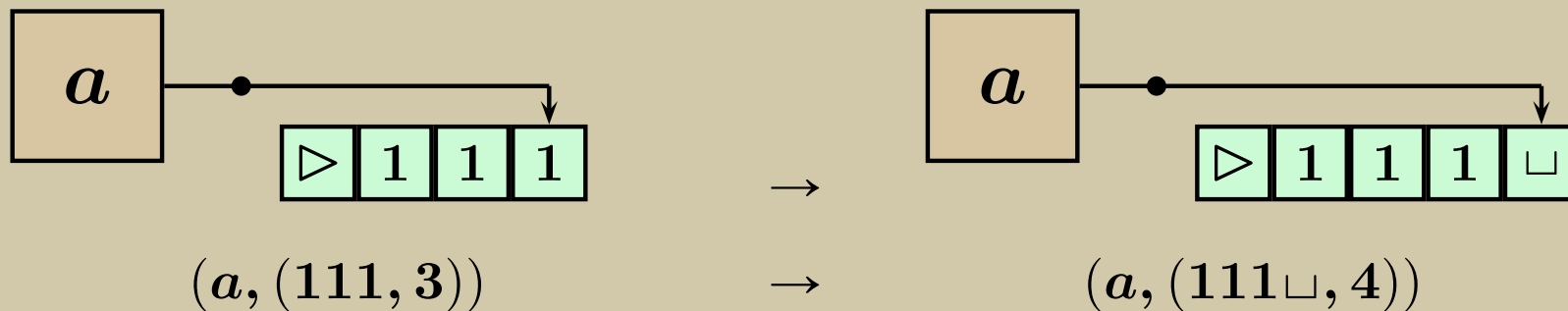
## Beispiel: Rechtsschritt

- $\delta(c, 0) = (d, 1, \rightarrow)$




## Beispiel: Rechtsschritt mit neuem Blank

- $\delta(a, 1) = (a, 1, \rightarrow)$



# Turingmaschinen: Semantik (1/2)

## Definition

- Sei  $\Sigma \subseteq \Gamma - \{\triangleright, \sqcup\}$  (**Ein-/Ausgabe-Alphabet**)
- Die Startkonfiguration von  $M$  bei Eingabe  $u \in \Sigma^*$  ist  $\underline{K_0(u)} \stackrel{\text{def}}{=} (s, (u, 0))$
- $(q, (w, z))$  heißt Haltekonfiguration, falls  $q \in \{h, \text{ja}, \text{nein}\}$
- $K_0, K_1, \dots, K_t$  heißt endliche Berechnung von  $M$  bei Eingabe  $u$ , falls
  - $K_0 = K_0(u)$ ,
  - $K_i \vdash_M K_{i+1}$  für alle  $i < t$ , und
  - $K_t$  eine Haltekonfiguration ist
- $M$  akzeptiert  $u$ , falls  $K_0(u) \vdash_M^* (\text{ja}, (w, z))$
- $M$  lehnt  $u$  ab, falls  $K_0(u) \vdash_M^* (\text{nein}, (w, z))$ 
  - (für gewisse  $w \in \Sigma^*, z \leq |w|$ )
- $M(u)$   $\stackrel{\text{def}}{=}$  die (endliche oder unendliche) Berechnung von  $M$  bei Eingabe  $u$    $M(u)$  ist nicht die Ausgabe!



# Turingmaschinen: Beispiel einer Konfigurationsfolge

## Beispiel

$h$

▷ 1 0 0 0

$(a, (\epsilon, \epsilon, 111)) \vdash_M (a, (\epsilon, 1, 11))$   
 $\vdash_M (a, (1, 1, 1))$   
 $\vdash_M (a, (11, 1, \epsilon))$   
 $\vdash_M (a, (111, \sqcup, \epsilon))$   
 $\vdash_M (b, (11, 1, \sqcup))$   
 $\vdash_M (b, (1, 1, 0\sqcup))$   
 $\vdash_M (b, (\epsilon, 1, 00\sqcup))$   
 $\vdash_M (b, (\epsilon, \epsilon, 000\sqcup))$   
 $\vdash_M (c, (\epsilon, 0, 00\sqcup))$   
 $\vdash_M (d, (1, 0, 0\sqcup))$   
 $\vdash_M (d, (10, 0, \sqcup))$   
 $\vdash_M (d, (100, \sqcup, \epsilon))$   
 $\vdash_M (e, (10, 0, 0))$   
 $\vdash_M (e, (1, 0, 00))$   
 $\vdash_M (e, (\epsilon, 1, 000))$   
 $\vdash_M (e, (\epsilon, \epsilon, 1000))$   
 $\vdash_M (h, (\epsilon, \epsilon, 1000))$

## Turingmaschinen: Semantik (2/2)

### Definition

- Eine TM  $M$  entscheidet eine Sprache  $L$ , falls für jedes  $u \in \Sigma^*$  gilt:
  - $u \in L \Rightarrow M$  akzeptiert  $u$
  - $u \notin L \Rightarrow M$  lehnt  $u$  ab
- $L(M) \stackrel{\text{def}}{=} \text{Menge aller von } M \text{ akzeptierten Wörter}$

- Zu beachten:
  - $L(M)$  ist immer definiert
  - Aber  $M$  entscheidet  $L(M)$  nicht immer!
    - \* Es könnte sein, dass  $M$  für gewisse Eingabewörter, die nicht in  $L(M)$  sind, nicht anhält
  - $M$  entscheidet  $L(M)$  genau dann, wenn  $M$  für jedes Eingabewort anhält

- Turingmaschinen können auch Funktionen berechnen:

### Definition

- $f_M(u) \stackrel{\text{def}}{=} v \in \Sigma^*$ , falls
  - $K_0(u) \vdash_M^* (h, (v, 0))$  oder
  - $K_0(u) \vdash_M^* (h, (v\tau w, 0))$   
für ein  $\tau \in \Gamma - \Sigma, w \in \Gamma^*$

- ✎  $f_M(u)$  ist nur dann definiert, wenn der Zeiger von  $M$  im Haltezustand ganz links steht
  - $f_M(u)$  ist dann die maximale Folge von Zeichen aus  $\Sigma$ , die direkt rechts vom Zeiger stehen
- Im Allgemeinen ist  $f_M$  also eine partielle Funktion  $\Sigma^* \rightarrow \Sigma^*$

### Definition

- Eine partielle Funktion  $f : \Sigma^* \rightarrow \Sigma^*$  heißt Turing-berechenbar, falls  $f = f_M$ , für eine Turingmaschine  $M$

# Literaturangaben

- Die Darstellung in diesem Kapitel richtet sich weitgehend nach
  - Uwe Schöning. *Theoretische Informatik - kurzgefaßt* (3. Aufl.). Hochschultaschenbuch. Spektrum Akademischer Verlag, 1997

# Grundbegriffe der Theoretischen Informatik

Sommersemester 2016 - Thomas Schwentick

Teil C: Berechenbarkeit und Entscheidbarkeit

13: Die Church-Turing-These

Version von: 9. Juni 2016 (14:00)

# Plan

- Alle im letzten Kapitel betrachteten Berechnungsmodelle sind gleichmächtig:
  - WHILE-Programme
  - GOTO-Programme
  - Turingmaschinen
- Die Church-Turing-These besagt, dass diese (und andere) Modelle gerade die intuitiv berechenbaren Funktionen erfassen
- Außerdem:
  - Turingmaschinen mit mehreren Strings können durch 1-String-Turingmaschinen simuliert werden

# Inhalt

## ▷ **13.1 WHILE vs. GOTO**

13.2 Mehrstring-Turingmaschinen

13.3 Turingmaschinen und WHILE/GOTO-Programme

13.4 Die Church-Turing-These

13.5 Entscheidbarkeit und Berechenbarkeit: Definition

# GOTO $\rightarrow$ WHILE

## Satz 13.1

- Jede GOTO-berechenbare Funktion ist auch WHILE-berechenbar

### Beweisidee

$M_1 : A_1;$   
 $M_2 : A_2;$   
 $\vdots$   
 $M_k : A_k$

$\rightarrow$

```

 $x_z := 1;$ 
WHILE  $x_z \neq 0$  DO
    IF  $x_z = 1$  THEN  $A'_1$  END;
    IF  $x_z = 2$  THEN  $A'_2$  END;
     $\vdots$ 
    IF  $x_z = k$  THEN  $A'_k$  END;
    IF  $x_z = k + 1$  THEN  $x_z := 0$ 
END
    
```

$$\bullet A_n = x_i := x_j + c \Rightarrow A'_n = x_i := x_j + c; x_z := x_z + 1$$

$$\bullet A_n = x_i := x_j \div c \Rightarrow A'_n = x_i := x_j \div c; x_z := x_z + 1$$

$$\bullet A_n = \text{IF } x_i = c \text{ THEN GOTO } M_j \Rightarrow A'_n = x_z := x_z + 1; \text{ IF } x_i = c \text{ THEN } x_z := j \text{ END}$$

$$\bullet A_n = \text{HALT} \Rightarrow A'_n = x_z := 0$$

# WHILE $\rightarrow$ GOTO

## Satz 13.2

- Jede WHILE-berechenbare Funktion ist auch GOTO-berechenbar

## Beweisidee

- Ein Teilprogramm

```
WHILE  $x_i \neq 0$  DO  $P$  END
```

kann durch

```
 $M_1$  : IF  $x_i = 0$  THEN GOTO  $M_2$ ;  
       $P'$ ; GOTO  $M_1$ ;  
 $M_2$  :  $x_i := x_i$ 
```

simuliert werden

- Kleines Fazit:
  - Die Klasse der WHILE-berechenbaren Funktionen ist also gleich der Klasse der GOTO-berechenbaren Funktionen
- Es gilt sogar:
  - Jede WHILE-berechenbare Funktion ist durch ein WHILE-Programm mit nur *einer* WHILE-Schleife (aber mehreren IF-Anweisungen) berechenbar



# Inhalt

13.1 WHILE vs. GOTO

▷ **13.2 Mehrstring-Turingmaschinen**

13.3 Turingmaschinen und WHILE/GOTO-Programme

13.4 Die Church-Turing-These

13.5 Entscheidbarkeit und Berechenbarkeit: Definition

# Mehrstring-Turingmaschinen: Beispiel

- Um die Umwandlung von WHILE-Programmen in Turingmaschinen zu erleichtern, gönnen wir uns etwas mehr Komfort:
  - Turingmaschinen mit mehreren Strings
- Wichtig: Bei jeder *einzelnen* Turingmaschine ist die Anzahl der Strings fest

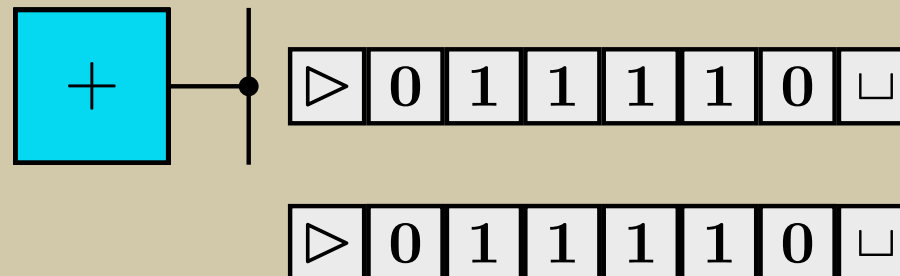
## Beispiel

2-String-Turingmaschine zum Test, ob die Eingabe von der Form  $ww^R$  ist:

**a:** Kopiere Eingabewort vom ersten auf den zweiten String (bis  $\sqcup$ )

**b/c:** Bewege Kopf 2 zurück an Anfang, teste dabei, ob die Anzahl der Zeichen gerade oder ungerade ist

**d:** Bewege Kopf 1 nach links, Kopf 2 nach rechts, vergleiche jeweils die gelesenen Zeichen

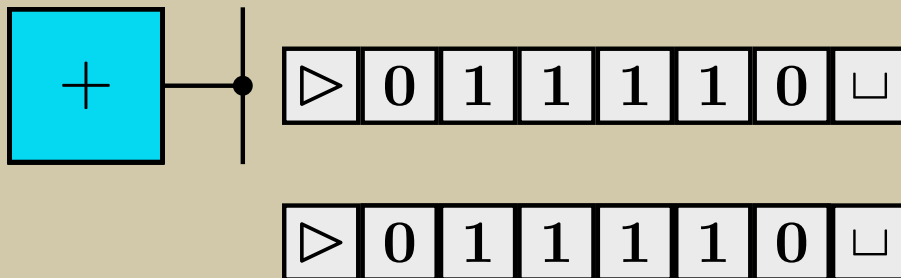


# Mehrstring-Turingmaschinen: Transitionsfunktion

## Beispiel

2-String-Turingmaschine zum Test, ob die Eingabe von der Form  $ww^R$  ist:

- a:** Kopiere Eingabewort vom ersten auf den zweiten String (bis  $\sqcup$ )
- b/c:** Bewege Kopf 2 zurück an Anfang, teste dabei, ob die Anzahl der Zeichen gerade oder ungerade ist
- d:** Bewege Kopf 1 nach links, Kopf 2 nach rechts, vergleiche jeweils die gelesenen Zeichen



## Beispiel

| Vorher |                  |                  | Nachher |                  |                  |               |               |
|--------|------------------|------------------|---------|------------------|------------------|---------------|---------------|
| $q$    | $\gamma_1$       | $\gamma_2$       | $q$     | $\gamma_1$       | $\gamma_2$       | $d_1$         | $d_2$         |
| $a$    | $\triangleright$ | $\triangleright$ | $a$     | $\triangleright$ | $\triangleright$ | $\rightarrow$ | $\rightarrow$ |
| $a$    | 0                | $\sqcup$         | $a$     | 0                | 0                | $\rightarrow$ | $\rightarrow$ |
| $a$    | 1                | $\sqcup$         | $a$     | 1                | 1                | $\rightarrow$ | $\rightarrow$ |
| $a$    | $\sqcup$         | $\sqcup$         | $b$     | $\sqcup$         | $\sqcup$         | $\downarrow$  | $\leftarrow$  |
| $b$    | $\sqcup$         | 0                | $c$     | $\sqcup$         | 0                | $\downarrow$  | $\leftarrow$  |
| $c$    | $\sqcup$         | 1                | $b$     | $\sqcup$         | 1                | $\downarrow$  | $\leftarrow$  |
| $b$    | $\sqcup$         | 1                | $c$     | $\sqcup$         | 1                | $\downarrow$  | $\leftarrow$  |
| $c$    | $\sqcup$         | 0                | $b$     | $\sqcup$         | 0                | $\downarrow$  | $\leftarrow$  |
| $b$    | $\sqcup$         | $\triangleright$ | $d$     | $\sqcup$         | $\triangleright$ | $\leftarrow$  | $\rightarrow$ |
| $d$    | 0                | 0                | $d$     | 0                | 0                | $\leftarrow$  | $\rightarrow$ |
| $d$    | 1                | 1                | $d$     | 1                | 1                | $\leftarrow$  | $\rightarrow$ |
| $d$    | $\triangleright$ | $\sqcup$         | +       | $\triangleright$ | $\sqcup$         | $\downarrow$  | $\downarrow$  |

# Mehrstring-Turingmaschinen: Definition (1/2)

## Definition: Mehrstring-TM (Syntax)

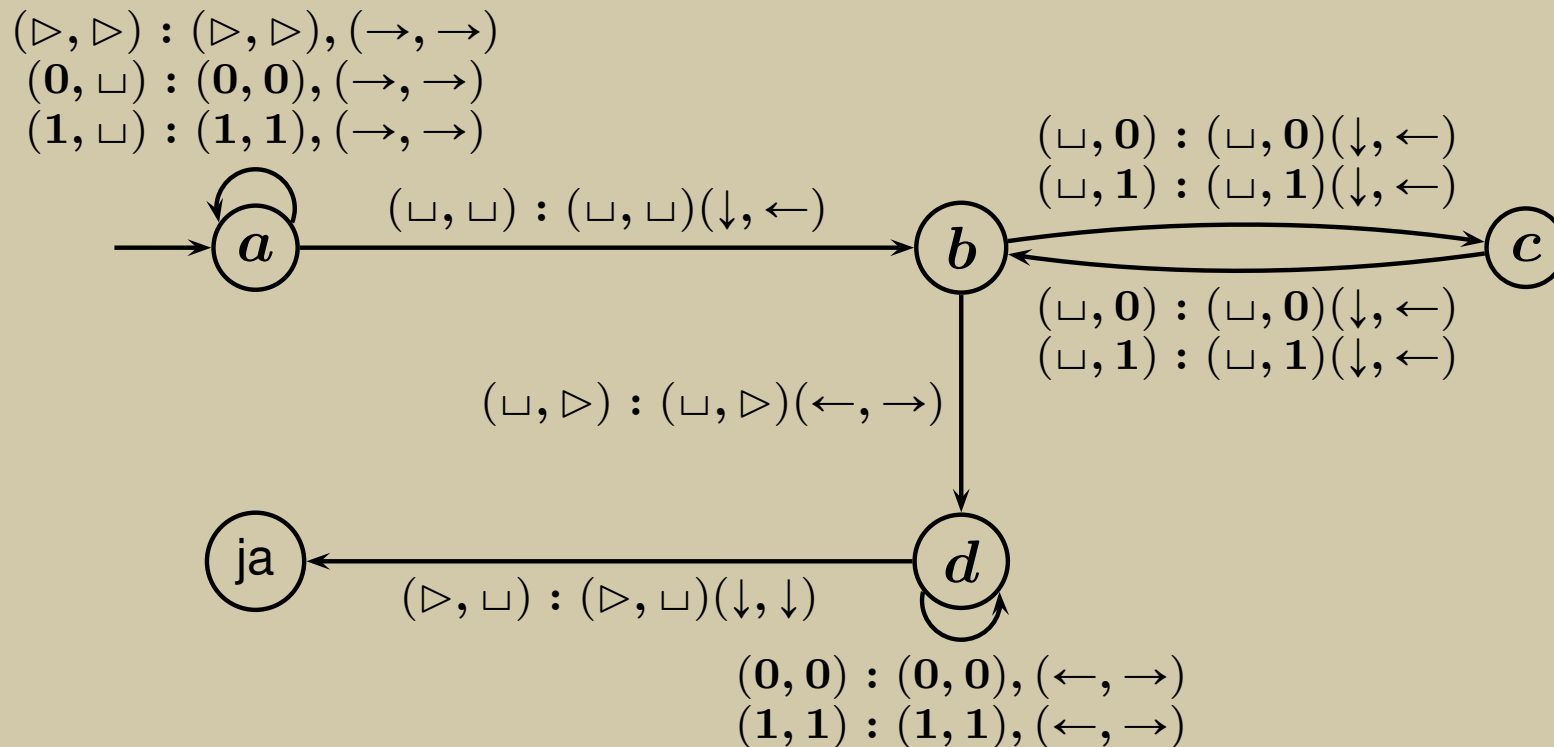
- Sei  $k \geq 1$
- Eine  $k$ -String-Turingmaschine  $M = (Q, \Gamma, \delta, s)$  besteht aus
  - einer Menge  $Q$  von **Zuständen**,
  - einem **Bandalphabet**  $\Gamma$  mit  $\sqcup \in \Gamma$  und  $\triangleright \in \Gamma$   
(wir nennen  $\sqcup$  "Blank" und  $\triangleright$  "linker Rand"),
  - einem **Anfangszustand**  $s \in Q$ , und
  - einer **Transitionssfunktion**  
$$\delta : Q \times \Gamma^k \rightarrow (Q \cup \{h, \text{ja}, \text{nein}\}) \times \Gamma^k \times \{\leftarrow, \downarrow, \rightarrow\}^k$$
- Dabei seien  $Q, \Gamma, \{h, \text{ja}, \text{nein}\}$  und  $\{\leftarrow, \downarrow, \rightarrow\}$  paarweise disjunkt

## Bemerkungen

- Die Anzahl  $k$  der Strings ist implizit durch  $\delta$  gegeben
- Wenn es auf das genaue  $k$  nicht ankommt, sagen wir auch Mehrstring-Turingmaschine statt  $k$ -String-Turingmaschine

# Mehrstring-Turingmaschinen: Diagrammdarstellung

## Beispiel-TM in Diagramm-Darstellung



- In diesem Beispiel gilt die Konvention:
  - Ist für  $(q, \sigma_1, \dots, \sigma_k) \in Q \times \Gamma^k$  kein Übergang eingezeichnet, so sei  $\delta(q, \sigma_1, \dots, \sigma_k) \stackrel{\text{def}}{=} (\text{nein}, \sigma_1, \dots, \sigma_k, \downarrow, \dots, \downarrow)$

# Mehrstring-Turingmaschinen: Definition (2/2)

## Definition: Mehrstring-TM (Semantik)

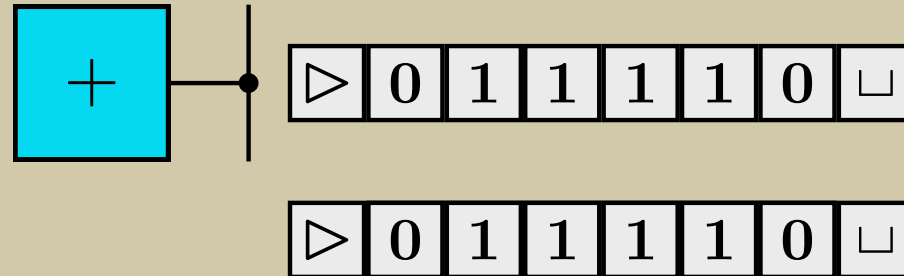
- Sei  $k \geq 1$  und  $M = (Q, \Gamma, \delta, s)$  eine  $k$ -string-TM
- **Ein-/Ausgabealphabet:**  
 $\Sigma \subseteq \Gamma - \{\sqcup, \triangleright\},$
- **Konfiguration** von  $M$ :  $k + 1$ -Tupel  $(q, s_1, \dots, s_k)$ , wobei
  - $q \in Q$
  - $s_i$  String-Zeiger-Beschreibung für  $i$ -ten String
- **Startkonfiguration**  $K_0(u)$  von  $M$  bei Eingabe  $u \in \Sigma^*$ :  
 $(s, (u, 0), (\epsilon, 0), \dots, (\epsilon, 0))$
- $(q, s_1, \dots, s_k)$  ist **Haltekonfiguration**, falls  $q \in \{h, \text{ja}, \text{nein}\}$

## Definition: Mehrstring-TM (Semantik) (Forts.)

- Sei  $K = (q, (u_1, z_1), \dots, (u_k, z_k))$  eine Konfiguration von  $M$  und sei, für jedes  $i$ ,  $\sigma_i \stackrel{\text{def}}{=} w[i]$
- Ist  $\delta(q, \sigma_1, \dots, \sigma_k) = (q', \tau_1, \dots, \tau_k, d_1, \dots, d_k)$ , so ist  $K' = (q', (u'_1, z'_1), \dots, (u'_k, z'_k))$  die **Nachfolgekonfiguration von  $K$** , wenn für alle  $i$  gilt:
  - $z'_i = z_i + 1$ , falls  $d_i = \rightarrow$ ,
  - $z'_i = z_i$ , falls  $d_i = \downarrow$ ,
  - $z'_i = z_i - 1$ , falls  $d_i = \leftarrow$ ,
  - $u'_i = u_i[z_i/\tau_i]\sqcup$ , falls  $z_i = |u'_i|$  und  $d_i = \rightarrow$ ,
  - $u'_i = u'_i[z_i/\tau_i]$ , andernfalls
- Schreibweise:  $K \vdash_M K'$ 
  - Sprechweise:  $M$  erreicht  $K'$  von  $K$  aus in einem Schritt
- Die übrigen Begriffe wie Berechnungen, Akzeptieren, Ablehnen,  $\vdash_M^*$ ,  $L(M)$  sind definiert wie bei (1-String-Turingmaschinen)

# Semantik von Mehrstring-TM: Beispiel

## Beispiel



$$\begin{aligned}
 &(a, (\epsilon, \epsilon, 011110), (\epsilon, \epsilon, \epsilon)) \vdash_M (a, (\epsilon, 0, 11110), (\epsilon, \sqcup, \epsilon)) \vdash_M \\
 &\quad (a, (0, 1, 1110), (0, \sqcup, \epsilon)) \vdash_M (a, (01, 1, 110), (01, \sqcup, \epsilon)) \vdash_M \\
 &\quad (a, (011, 1, 10), (011, \sqcup, \epsilon)) \vdash_M (a, (0111, 1, 0), (0111, \sqcup, \epsilon)) \vdash_M \\
 &\quad (a, (01111, 0, \epsilon), (01111, \sqcup, \epsilon)) \vdash_M (a, (011110, \sqcup, \epsilon), (011110, \sqcup, \epsilon)) \vdash_M \\
 &(b, (011110, \sqcup, \epsilon), (01111, 0, \sqcup)) \vdash_M (c, (011110, \sqcup, \epsilon), (0111, 1, 0\sqcup)) \vdash_M \\
 &(b, (011110, \sqcup, \epsilon), (011, 1, 10\sqcup)) \vdash_M (c, (011110, \sqcup, \epsilon), (01, 1, 110\sqcup)) \vdash_M \\
 &(b, (011110, \sqcup, \epsilon), (0, 1, 1110\sqcup)) \vdash_M (c, (011110, \sqcup, \epsilon), (\epsilon, 0, 11110\sqcup)) \vdash_M \\
 &(b, (011110, \sqcup, \epsilon), (\epsilon, \epsilon, 011110\sqcup)) \vdash_M (d, (01111, 0, \sqcup), (\epsilon, 0, 11110\sqcup)) \vdash_M \\
 &\quad (d, (0111, 1, 0\sqcup), (0, 1, 1110\sqcup)) \vdash_M (d, (011, 1, 10\sqcup), (01, 1, 110\sqcup)) \vdash_M \\
 &\quad (d, (01, 1, 110\sqcup), (011, 1, 10\sqcup)) \vdash_M (d, (0, 1, 1110\sqcup), (0111, 1, 0\sqcup)) \vdash_M \\
 &\quad (d, (\epsilon, 0, 11110\sqcup), (01111, 0, \sqcup)) \vdash_M (d, (\epsilon, \epsilon, 011110\sqcup), (011110, \sqcup, \epsilon)) \vdash_M \\
 & („ja“, (\epsilon, \epsilon, 011110\sqcup), (011110, \sqcup, \epsilon))
 \end{aligned}$$

# Turingmaschinen: Robustheit

## Satz 13.3

- Zu jeder Mehrstring-TM  $M = (Q, \Gamma, \delta, s)$  gibt es eine 1-String-TM  $M'$  mit  $L(M') = L(M)$

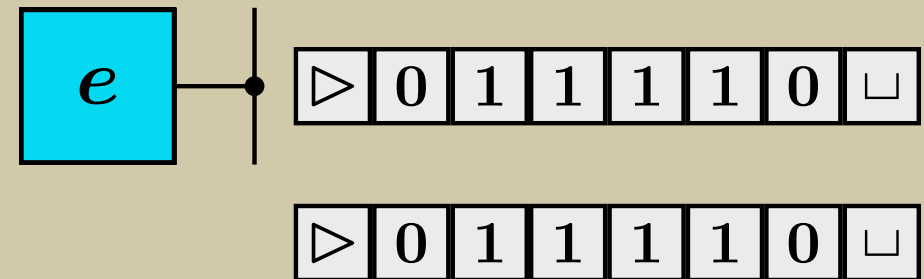
## Beweisidee

- Idee: „**Spurentechnik**“,  $M'$  verwendet Symbole aus  $(\Gamma \times \{-, +\})^k$ :
  - Die  $i$ -te Komponente  $(\gamma_i, p_i)$  bedeutet:
    - \* Zeichen  $\gamma_i$  im  $i$ -ten String
    - \*  $i$ -ter Kopf an dieser Position, falls  $p_i = +$
- Für einen Schritt von  $M$ 
  - läuft  $M'$  zum rechten Rand und liest alle Kopfsymbole von  $M$ ,
  - läuft  $M'$  zum linken Rand zurück und macht dabei alle nötigen Änderungen,
  - geht  $M'$  in den neuen Zustand über

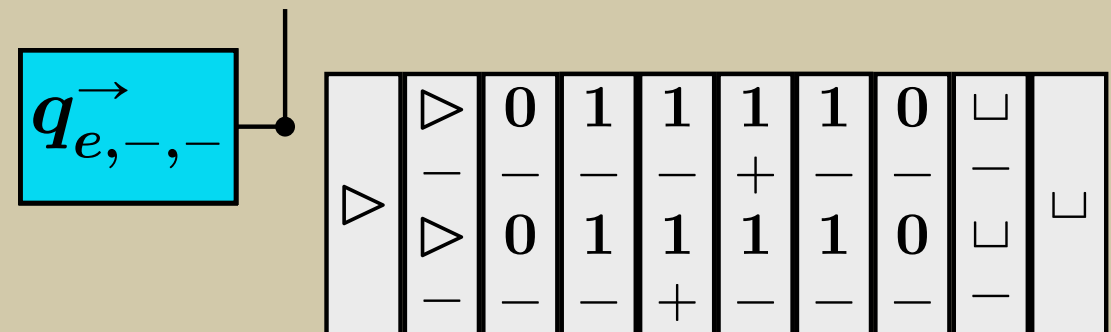
## Beispiel

- Simulation eines Schrittes einer 2-String-TM durch eine 1-String-TM
  - Hier:  $\delta(d, 1, 1) = (e, 1, 1, \leftarrow, \rightarrow)$

- 2-String-TM:



- 1-String-TM:



- Analog kann auch  $f_{M'} = f_M$  erreicht werden



# Inhalt

13.1 WHILE vs. GOTO

13.2 Mehrstring-Turingmaschinen

▷ **13.3 Turingmaschinen und WHILE/GOTO-Programme**

13.4 Die Church-Turing-These

13.5 Entscheidbarkeit und Berechenbarkeit: Definition

# Strings vs. Zahlen

- Turingmaschinen  $M$  berechnen partielle Funktionen  $f_M : \Sigma^* \rightarrow \Sigma^*$  (OBdA:  $\Sigma = \{0, 1\}$ )
- WHILE-Programme  $P$  berechnen partielle Funktionen  $f_P : \mathbb{N}_0 \rightarrow \mathbb{N}_0$
- Um die beiden Modelle miteinander zu vergleichen, müssen wir Strings und Zahlen ineinander umwandeln können
- Wir verwenden dazu die beiden wie folgt definierten Umwandlungsfunktionen:
  - Str2N bildet jeden Binärstring auf die durch ihn kodierte Zahl ab, also z.B.:
    - \*  $\text{Str2N}(110) = 6$
    - \*  $\text{Str2N}(00110) = 6$
    - \*  $\text{Str2N}(00000) = 0$
    - \*  $\text{Str2N}(\epsilon) = 0$
  - N2Str bildet jede natürliche Zahl auf ihren Binärstring ohne führende Nullen ab, also z.B.:
    - \*  $\text{N2Str}(6) = 110$
    - \*  $\text{N2Str}(0) = \epsilon$

# WHILE-Programme → Turingmaschinen (1/2)

## Satz 13.4

- Jede WHILE-berechenbare Funktion ist Turing-berechenbar
- Genauer: Für jede WHILE-berechenbare Funktion  $f : \mathbb{N}_0 \rightarrow \mathbb{N}_0$  gibt es eine Mehrstring-Turingmaschine  $M$ , so dass für alle  $n \in \mathbb{N}_0$  gilt:  
$$f(n) = \text{Str2N}(f_M(\text{N2Str}(n)))$$

## Beweisskizze


- Sei  $P$  ein WHILE-Programm für  $f$ 
  - ➔ es gibt ein  $k > 0$ , so dass  $P$  keine anderen Variablen als  $x_1, \dots, x_k$  benutzt
- Idee:  $P$  wird durch eine  $k$ -String Turingmaschine  $M$  simuliert
  - Jeder String von  $M$  repräsentiert dabei den Wert einer Variablen  $x_i$
  - Zu Beginn steht in String 1 der String  $\text{N2Str}(n)$  und auf den anderen Strings der Leerstring (entspricht 0)
  - Am Ende der Simulation steht auf String 1 die Binärkodierung des Ergebnisses
    - ✎ ... und muss dann noch in den  $k$ -ten String kopiert werden
  - Jedes Teilprogramm  $P'$  von  $P$  wird durch eine TM  $M_{P'}$  simuliert
    - \*  $M_{P'}$  ist dabei induktiv definiert

# WHILE-Programme → Turingmaschinen (2/2)

Beweisskizze für Satz 13.4 (Forts.)

| $P'$                            | $M_{P'}$  |
|---------------------------------|---|
| $x_j := x_i + c$                | <ul style="list-style-type: none"> <li>Falls <math>j \neq i</math> <ul style="list-style-type: none"> <li>String <math>j</math> mit <math>\sqcup</math> überschreiben</li> <li>String <math>i</math> nach String <math>j</math> kopieren</li> </ul> </li> <li><math>c</math>-mal 1 zu String <math>j</math> addieren</li> </ul> |
| $x_j := x_i \div c$             | analog  |
| $x_j := c$                      |   |
| $x_j := x_i$                    |   |
| $P_1; P_2$                      | Führe zuerst $M_{P_1}$ aus, dann $M_{P_2}$  |
| WHILE $x_i \neq 0$ DO $P_1$ END | <ul style="list-style-type: none"> <li>(a) Wenn <math>i</math>-ter String Leerstring ist: fertig</li> <li>(b) Andernfalls <math>M_{P_1}</math> ausführen, dann weiter mit (a)</li> </ul>  |

# Turingmaschinen → GOTO-Programme (1/5)

- Die Simulation von Turingmaschinen durch GOTO-Programme wirft ein Problem auf:
    - Turingmaschinen können, abhängig von der Eingabe, beliebig viele Positionen benutzen
    - Jedes GOTO- (oder WHILE-) Programm hat aber nur eine feste Zahl von Variablen
      - \* Wir können also leider **nicht** für jede Position des Turingmaschinen-Strings eine Variable verwenden
-  mit indirekter Adressierung ginge das...

- Wir werden deshalb String-Zeigerbeschreibungen, durch je drei Zahlen kodieren, damit sie in drei Variablen gespeichert werden können

- Strings über dem Arbeitsalphabet  $\Gamma = \{\sigma_1, \dots, \sigma_\ell\}$  interpretieren wir dazu als Zahlen in  $(\ell+1)$ -adischer Darstellung gemäß  $\sigma_i \mapsto i$ , für jedes  $i$

## Beispiel

- Für  $\Gamma = \{\triangleright, \sqcup, 0, 1\}$  ergibt sich
  - $\triangleright \mapsto 1$
  - $\sqcup \mapsto 2$
  - $0 \mapsto 3$
  - $1 \mapsto 4$
- Z.B.:  $\text{Str2N}_\Gamma(100) = 4 \times 5^2 + 3 \times 5 + 3 = 118$

- $\text{Str2N}_\Gamma(w)$  ist induktiv definiert durch:
  - $\text{Str2N}_\Gamma(\epsilon) \stackrel{\text{def}}{=} 0$  und
  - $\text{Str2N}_\Gamma(u\sigma_i) \stackrel{\text{def}}{=} (\ell + 1) \times \text{Str2N}_\Gamma(u) + i$

- Es gelten:
  - $\text{Str2N}_\Gamma(u\sigma) \div (\ell + 1) = \text{Str2N}_\Gamma(u)$
  - $\text{Str2N}_\Gamma(u\sigma) \bmod (\ell + 1) = \text{Str2N}_\Gamma(\sigma)$

## Turingmaschinen → GOTO-Programme (2/5)


- Für die Simulation von Turingmaschinen durch GOTO-Programme verwenden wir die Notation  $(u, \sigma, v)$  für String-Zeigerbeschreibungen
- Konfigurationen der TM  $M$  werden durch Speicherinhalte der Variablen  $x_1, x_2, x_3, x_4$  repräsentiert:
  - $S_M(q_i, (u, \sigma, v)) \stackrel{\text{def}}{=} i, \text{Str2N}_\Gamma(u), \text{Str2N}_\Gamma(\sigma), \text{Str2N}_\Gamma(v^R), \dots$

 Warum  $v^R$ ?

- \* Damit das erste Zeichen von  $v$  durch  $x_4 \bmod (\ell + 1)$  gegeben ist

### Beispiel

- Die Startkonfiguration  $(q_1, (\epsilon, \epsilon, 001))$  entspricht also dem Speicherinhalt  $1, 0, 0, 118, \dots$

 Zu beachten:  $\text{Str2N}_\Gamma(001^R) = \text{Str2N}_\Gamma(100) = 118$

- Die Umkehrabbildung  $\text{N2Str}_\Gamma : \mathbb{N} \rightarrow \Gamma^*$  von  $\text{Str2N}_\Gamma$  sei wie folgt definiert:
  - $\text{N2Str}_\Gamma(n) \stackrel{\text{def}}{=} \begin{cases} w & \text{falls } \text{Str2N}_\Gamma(w) = n, \text{ für ein } w \in \Gamma^* \\ \perp & \text{andernfalls} \end{cases}$

# Turingmaschinen → GOTO-Programme (3/5)

## Satz 13.5

- Jede Turing-berechenbare Funktion ist auch GOTO-berechenbar
- Genauer: für jede Turing-berechenbare Funktion  $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$  gibt es ein GOTO-Programm  $P$ , so dass für alle  $w \in \Sigma^*$  gilt:  

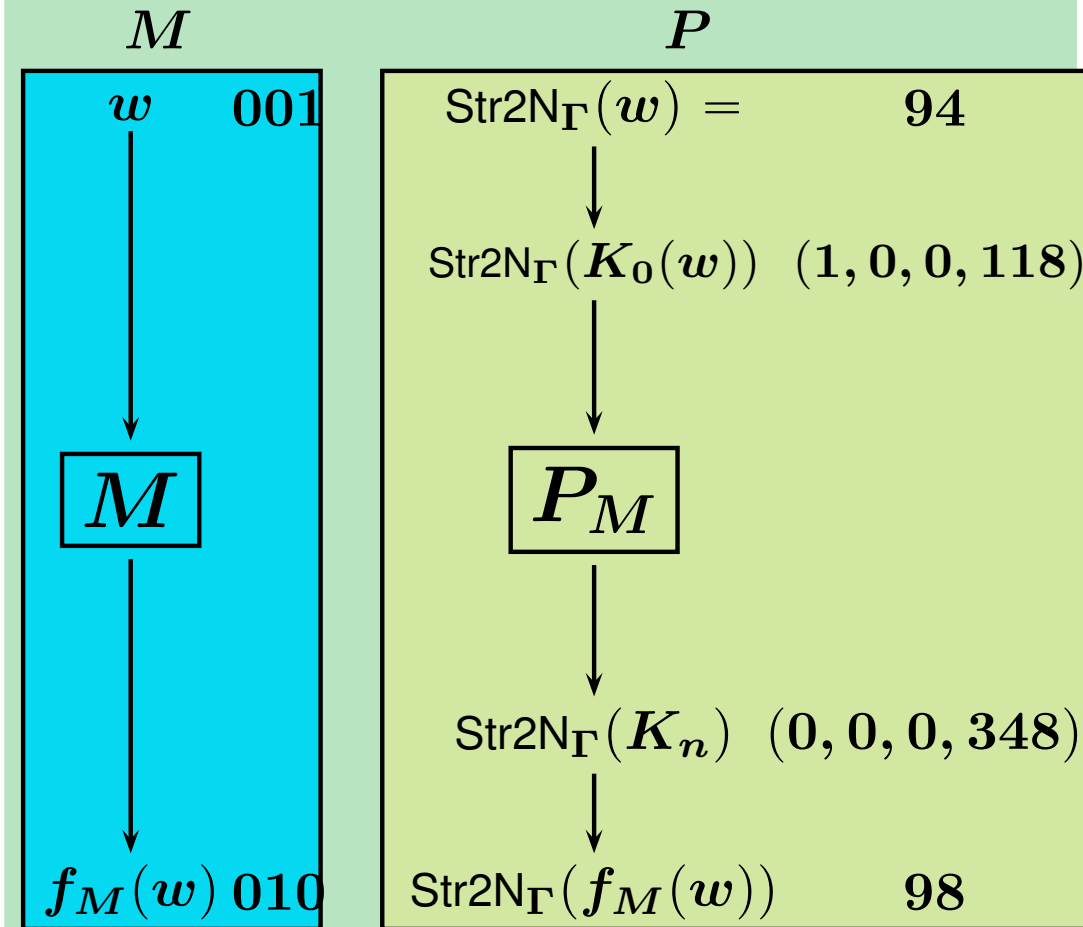
$$f(w) = \text{N2Str}_\Gamma(f_P(\text{Str2N}_\Gamma(w)))$$

## Beweisskizze

- Sei  $M = (Q, \Gamma, \delta, q_1)$  eine TM mit Zuständen  $q_1, \dots, q_k$ , die  $f$  berechnet (Proviso:  $q_0 = h$ )
- Wir repräsentieren Konfigurationen wie beschrieben durch die Variablen  $x_1, \dots, x_4$
- $P$  simuliert  $M$  in drei Phasen:
  1. Variablen initialisieren
  2.  $M$  schrittweise simulieren (Teilprogramm:  $P_M$ )
  3. Funktionswert aus  $x_2, x_3, x_4$  umkodieren

## Beweisskizze (Forts.)

- Simulation bei Eingabe **001** und Ausgabe **010**



- **13.1** ➡

# Turingmaschinen → GOTO-Programme (4/5)

## Beweisskizze (Forts.)

- $P$  simuliert die Berechnung von  $M$  Schritt für Schritt durch:

$M_1$ : IF ( $x_1 = 1$ ) AND ( $x_3 = 1$ )  
      THEN GOTO  $M_{11}$   
      IF ( $x_1 = 1$ ) AND ( $x_3 = 2$ )  
      THEN GOTO  $M_{12}$

⋮

IF ( $x_1 = k$ ) AND ( $x_3 = \ell$ )  
      THEN GOTO  $M_{k\ell}$   
IF ( $x_1 = 0$ ) THEN HALT

$M_{11}$ :  $P_{11}$   
      GOTO  $M_1$

$M_{12}$ :  $P_{12}$   
      GOTO  $M_1$

⋮

$M_{k\ell}$ :  $P_{k\ell}$   
      GOTO  $M_1$

- Zur Erinnerung:  $x_1$  speichert die Nummer des Zustandes,  $x_3$  die Kodierung des aktuellen Zeichens (und  $x_3 = 0$ , falls der Zeiger am linken Rand ist)



# Turingmaschinen → GOTO-Programme (5/5)

## Beweisskizze (Forts.)

- Beispiel für die Konstruktion der Teilprogramme  $P_{5,6}$ :

– Ist  $\delta(q_5, \sigma_6) = (q_8, \sigma_9, \rightarrow)$ ,  
dann ist  $P_{ij}$ :

$M_{5,6}: x_1 := 8;$   
IF  $x_3 = 0$  THEN GOTO  $M'_{5,6};$   
 $x_2 := (\ell + 1) \times x_2 + 9;$   
 $M'_{5,6}: x_3 := 2;$   
IF  $x_4 = 0$  THEN GOTO  $M''_{5,6};$   
 $x_3 := x_4 \bmod (\ell + 1);$   
 $M''_{5,6}: x_4 := x_4 \div (\ell + 1);$

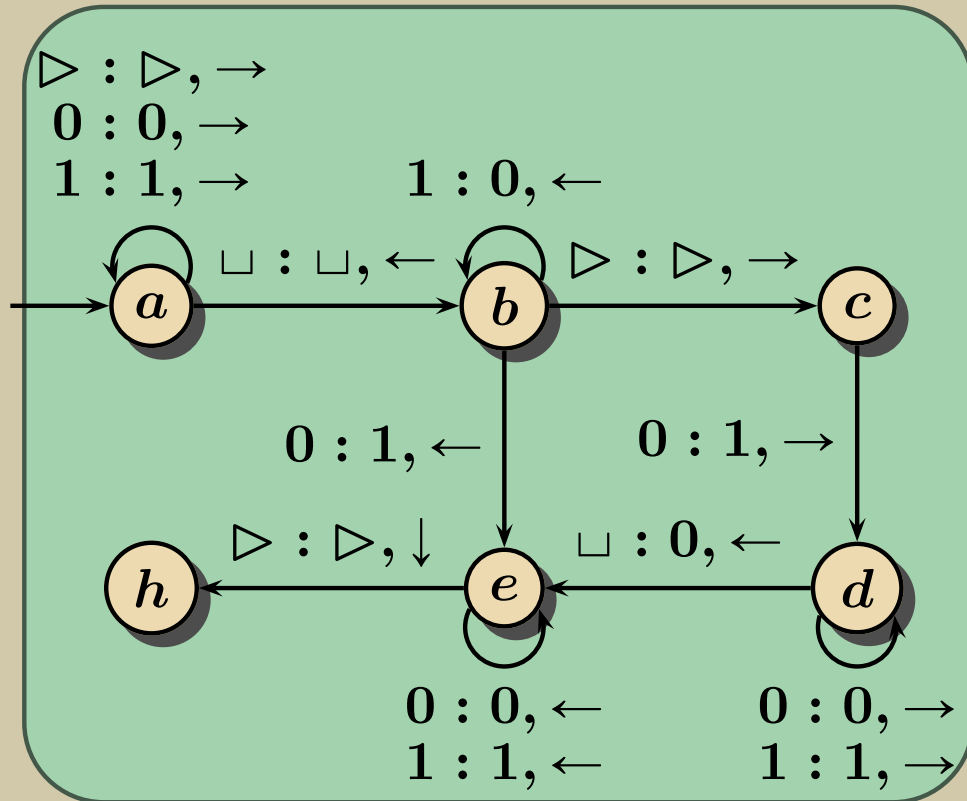
## Beweisskizze (Forts.)

- Erläuterungen:

- $x_1 := 8$ : Neuer Zustand  $q_8$
- IF  $x_3 = 0$  THEN GOTO  $M'_{5,6}$   
Falls Kopf auf dem linken Rand ist, wird  $x_2$  nicht verändert ( $x_2 = 0$ )
- $x_2 := (\ell + 1) \times x_2 + 9$   
Kodierung des neuen Strings links vom Kopf
- Die drei Zeilen ab  $M'_{5,6}$  bewirken, dass
  - \* im Falle einer Rechtsbewegung zu einer Position, die kein Eingabesymbol enthält und noch nicht besucht wurde, das aktuelle Zeichen zu einem Leerzeichen wird (= 2),
  - \* andernfalls das neue aktuelle Zeichen das erste Zeichen des bisherigen, durch  $x_4$  kodierten Strings rechts vom Kopf, wird
- $x_4 := x_4 \div (\ell + 1)$   
Der neue String rechts vom Kopf (auch im Falle, dass dieser leer ist, weil gerade erst ein Blank erzeugt wurde)

# Turingmaschinen → GOTO-Programme: Beispiel

Diagramm zur 2. TM



- Wir betrachten die Simulation dieser TM bei Eingabe **001**
- $\sigma_1 = \triangleright, \sigma_2 = \square, \sigma_3 = 0, \sigma_4 = 1$
- $q_0 = h, q_1 = a, q_2 = b, \dots$

Beispiel

- Statt der Eingabe  $w = 001$  erhält  $P$  die Zahl  $\text{Str2N}_\Gamma(w) = \text{Str2N}_\Gamma(001) = 94$
- Daraus berechnet  $P$  die Kodierung der Startkonfiguration  $(a, (\epsilon, \epsilon, 001))$  von  $M$ 
  - Es ergibt sich die Speicherbelegung **1, 0, 0, 118, ...**
- $M$  bewegt nun den Kopf nach rechts und bleibt im Zustand  $a$ 
  - Die neuen Werte für  $x_2, x_3, x_4$  ergeben sich durch:
    - \*  $x_2$  bleibt unverändert gleich  $\epsilon$ , da der Kopf am linken Rand steht
    - \*  $x_3 := x_4 \bmod 5 = 118 \bmod 5 = 3$ 
      - entsprechend dem Zeichen **0**
    - \*  $x_4 := x_4 \div 5 = 23$ 
      - entsprechend dem restlichen String **((01)<sup>R</sup> = 10)**
  - Die Konfiguration nach dem ersten Schritt entspricht also der Speicherbelegung **1, 0, 3, 23, ...**

# Inhalt

13.1 WHILE vs. GOTO

13.2 Mehrstring-Turingmaschinen

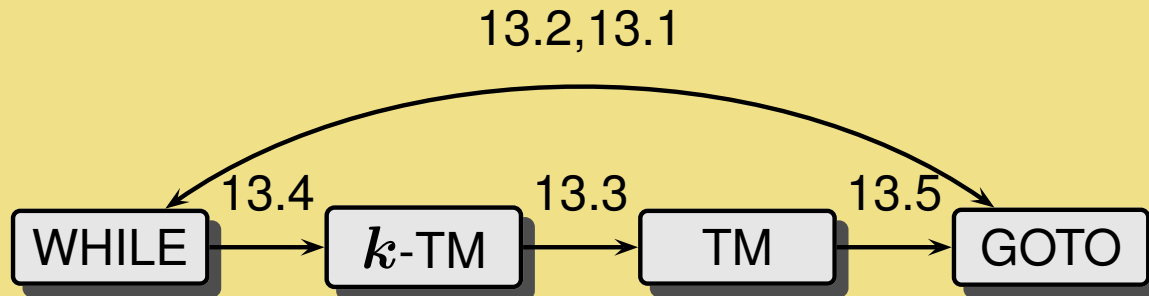
13.3 Turingmaschinen und WHILE/GOTO-Programme

▷ **13.4 Die Church-Turing-These**

13.5 Entscheidbarkeit und Berechenbarkeit: Definition

# Die Church-Turing-These

- Wir haben gesehen, dass alle bisher betrachteten Berechnungsmodelle äquivalent sind:



- Es gibt viel weitere Ansätze zur Formalisierung des Begriffes Algorithmus, die hinsichtlich ihrer Berechnungsstärke äquivalent sind, zum Beispiel:
  - 2-Kellerautomaten
  - Markov-Algorithmen
  - Typ-0-Grammatiken
  - $\lambda$ -Kalkül [Kleene 35, Church 36]
  - Register-Maschinen
- Aus diesem Grunde wird die Klasse der durch Turingmaschinen und die anderen genannten Modelle berechenbaren Funktionen als die „richtige“ Formalisierung des Algorithmus-Begriffs angesehen

- Es gibt wohl keine stärkeren „realistischen“ Berechnungsmodelle

- **Church-Turing-These:**

- Die Klasse der durch Turingmaschinen (WHILE-Programme,...) berechenbaren Funktionen umfasst alle intuitiv berechenbaren Funktionen

- Die Church-Turing-These wurde explizit erstmals von Kleene 1943 formuliert, aber dort schon auf Church und Turing zurückgeführt
- Sie ist nicht beweisbar
- Sie wäre aber im Prinzip widerlegbar: durch den Bau von Computern, die Funktionen berechnen, die nicht Turing-berechenbar sind

# Inhalt

13.1 WHILE vs. GOTO

13.2 Mehrstring-Turingmaschinen

13.3 Turingmaschinen und WHILE/GOTO-Programme

13.4 Die Church-Turing-These

▷ **13.5 Entscheidbarkeit und Berechenbarkeit: Definition**

# Entscheidbar und berechenbar

## Definition

- Eine Menge  $L \subseteq \Sigma^*$  heißt entscheidbar, falls es eine TM  $M$  gibt, die  $L$  entscheidet
- Zu beachten:
  - Bei einer entscheidbaren Menge muss die TM für *alle* Eingaben anhalten
- ✎ Statt „nicht entscheidbar“ sagen wir oft auch „unentscheidbar“
- Klar: Wenn  $L$  entscheidbar ist, dann auch das Komplement  $\overline{L}$  von  $L$

## Definition

- Eine partielle Funktion  $f : \Sigma^* \rightarrow \Sigma^*$  heißt berechenbar, falls es eine TM  $M$  gibt, die
  - für alle  $w \in D(f)$  mit Ausgabe  $f(w)$  anhält und
  - für alle  $w \notin D(f)$  nicht anhält
- $\mathcal{R} \stackrel{\text{def}}{=} \text{Menge der berechenbaren partiellen Funktionen}$
- ✎ Zur Erinnerung: auch totale Funktionen sind partielle Funktionen
  - $f$  kann also auch überall definiert sein...

# Algorithmische Probleme vs. Sprachen und Funktionen (1/4)

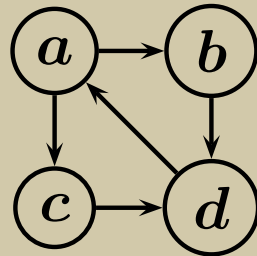
- Unsere bisherigen Berechnungsmodelle beziehen sich nur auf
  - Sprachen und Stringfunktionen bzw.
  - Mengen natürlicher Zahlen und Zahlenfunktionen
- Die soeben definierten Begriffe „entscheidbar“ und „berechenbar“ sind auch für Sprachen und Stringfunktionen definiert
- Wie hängt dies denn mit „richtigen“ algorithmischen Problemen zusammen?

## Algorithmische Probleme vs. Sprachen und Funktionen (2/4)

- Informatikerinnen und Informatikern wissen: alle Arten von Strukturen lassen sich durch 0-1-Strings kodieren
- Graphen können z.B. wie folgt durch Strings kodiert werden

### Beispiel

- Der Graph  $G =$



kann durch die Adjazenzmatrix

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

und diese dann durch den String  
 $\text{enc}(G) = 0110000100011000$  kodiert werden

- Solche Kodierungen ermöglichen uns, die Lücke zu schließen, die besteht zwischen
  - algorithmischen Problemen mit „komplizierteren“ Eingaben wie Graphen, Automaten etc., deren Lösbarkeit wir eigentlich untersuchen wollen, und
  - Sprachen und Funktionen auf Strings, die wir mit Turingmaschinen entscheiden bzw. berechnen können



Die Frage der Eindeutigkeit der Kodierung werden wir hier ignorieren

- Wichtig ist, dass jedem syntaktisch korrekten String eine (bis auf Isomorphie eindeutige) Eingabe zugeordnet werden kann



# Algorithmische Probleme vs. Sprachen und Funktionen (3/4)

- Algorithmische Entscheidungsprobleme entsprechen also Sprachen

## Definition: REACH

**Gegeben:** (Gerichteter) Graph  $G$ , Knoten  $s$  und  $t$

**Frage:** Gibt es in  $G$  einen Weg von  $s$  nach  $t$ ?


- Das algorithmische Entscheidungsproblem REACH entspricht der Sprache  $L_{\text{REACH}}$  aller 0-1-Strings, die einen gerichteten Graphen  $G$  und zwei Knoten  $s$  und  $t$  kodieren, in dem es einen Weg von  $s$  nach  $t$  gibt
- Besteht die Eingabe zu einem algorithmischen Problem aus mehreren Komponenten, so trennen wir diese in der Kodierung als Strings durch  $\#$
- $L_{\text{REACH}} = \{ \text{enc}(G) \# \text{enc}(s) \# \text{enc}(t) \mid G \text{ hat Weg von } s \text{ nach } t \}$   
für geeignete Kodierungsfunktionen  $\text{enc}$  für Graphen und Knoten

- Algorithmische Berechnungsprobleme entsprechen also Funktionen auf Strings

## Definition: MINGRAPHCOL

**Gegeben:** Ungerichteter Graph  $G$

**Gesucht:** Kleinstmögliche Anzahl von Farben, mit denen der Graph zulässig gefärbt werden kann

- Die zugehörige Funktion  $f_{\text{MINGRAPHCOL}}$  ordnet jedem String, der einen ungerichteten Graphen  $G$  kodiert, die (Kodierung der) kleinsten Zahl  $k$ , für die  $G$  eine  $k$ -Färbung hat, zu  
 Graphfärbungen werden wir in Teil D der Vorlesung noch genauer definieren


# Algorithmische Probleme vs. Sprachen und Funktionen (4/4)

- Wir werden die Begriffe „entscheidbar“ und „berechenbar“ auch für die entsprechenden algorithmischen Probleme verwenden
- Also: ist  $A$  ein algorithmisches Entscheidungsproblem und  $L_A$  entscheidbar, so nennen wir auch  $A$  entscheidbar
- Außerdem werden wir uns häufig die Church-Turing-These zunutze machen:
  - Statt eine TM für  $L_A$  zu konstruieren genügt es, einen Algorithmus für  $A$  anzugeben, um zu zeigen, dass  $A$  entscheidbar ist
- Ein Entscheidungsalgorithmus für  $A$  ist also künftig ein Algorithmus, der für jede Eingabe anhält und korrekt angibt, ob sie eine „Ja-Eingabe“ ist

# Entscheidbar und berechenbar: Beispiele

## Beispiel

- REACH ist entscheidbar
  - Der Tiefensuche-Algorithmus terminiert immer und gibt immer die richtige Antwort
- Das Wortproblem für kontextfreie Sprachen ist entscheidbar
  - Gegeben eine Grammatik  $G$  und ein Wort  $w$  kann mit dem CYK-Algorithmus überprüft werden, ob  $w \in L(G)$  ist
  - Der CYK-Algorithmus terminiert bei jeder Eingabe und gibt immer die richtige Antwort

 Bei den Grammatik-Beispielen nehmen wir der Einfachheit halber an, dass die Grammatiken in CNF sind

- Wenn nicht, können sie in eine CNF-Grammatik umgewandelt werden

## Beispiel

- Die Funktion, die jedem endliche Automaten  $\mathcal{A}$  die Anzahl der Zustände seines Minimalautomaten zuordnet, ist berechenbar und total
- Die Funktion, die jedem Paar  $(G_1, G_2)$  kontextfreier Grammatiken den lexikographisch kleinsten String  $w \in L(G_1) \cap L(G_2)$  zuordnet, ist berechenbar, aber nicht total
  - Sie ist undefiniert für Paare  $(G_1, G_2)$  mit  $L(G_1) \cap L(G_2) = \emptyset$
- Die Funktion, die jeder TM  $M$  und jeder Eingabe  $x$  den Wert  $f_M(x)$  zuordnet, ist berechenbar, aber nicht total

# Zusammenfassung

- Die verschiedenen Varianten des Turingmaschinen-Modells sind hinsichtlich ihrer Berechnungsstärke äquivalent
- Sie sind hinsichtlich ihrer Berechnungsstärke ebenfalls äquivalent zu WHILE-Programmen und GOTO-Programmen
- **Church-Turing-These:** Turing-Maschinen und die dazu äquivalenten Modelle sind die richtige Formalisierung des informellen Begriffes von **Algorithmus**
- Es gibt „universelle Turingmaschinen“
- Algorithmische Probleme können durch Sprachen und Funktionen auf Strings repräsentiert werden

# Erläuterungen

## Bemerkung 13.1

- Die Zahl 348 kodiert den String der TM am Ende der Berechnung:
  - $N2Str_{\Gamma}(348) = 010\sqcup$ , da
$$Str2N_{\Gamma}((010\sqcup)^R) = Str2N_{\Gamma}(\sqcup 010) = 2 \times 125 + 3 \times 25 + 4 \times 5 + 3 = 348$$
- $P$  berechnet daraus die Zahl 98, die den Ergebnisstring **010** kodiert

# Literatur

- Turing-Maschinen:
  - A. M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proc. London Math. Soc.*, 2(42):230–265, 1936
- $\lambda$ -Kalkül:
  - S. C. Kleene. A theory of positive integers in formal logic. *American Journal of Mathematics*, 57, 1935
  - Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2):345–363, 1936

# Grundbegriffe der Theoretischen Informatik

Sommersemester 2016 - Thomas Schwentick

Teil C: Berechenbarkeit und Entscheidbarkeit

14: Unentscheidbare Probleme 1

Version von: 16. Juni 2016 (11:16)

# Einleitung

- Wir beschäftigen uns in diesem Kapitel mit
  - algorithmischen Problemen, die nicht entscheidbar sind, und
  - mit dem Beweis dieser Tatsache
- Dabei lernen wir zwei Beweismethoden kennen:
  - Diagonalisierung
  - Reduktion
- Wir illustrieren das Prinzip zuerst an einem informellen Beispiel, bevor wir uns den „richtigen“ Sätzen und Beweisen zuwenden



# Inhalt

## ▷ 14.1 Hello, world!-Programme

14.2 Ein erstes unentscheidbares Problem

14.3 Reduktionen und weitere unentscheidbare Probleme

# „hello, world“-Programme: Einleitung (1/3)

## Beispiel: „hello, world“-Programm in Java

```
class HelloWorld {  
    static public void main( String args[ ] ) {  
        System.out.println( „Hello World!“ );  
    }  
}
```

- „**hello, world**“-Programme werden oft als erstes Beispiel beim Lehren einer Programmiersprache verwendet

- „hello, world“-Programme in Hunderten von Programmiersprachen finden sich auf **[www.roesler-ac.de/wolfram/hello.htm](http://www.roesler-ac.de/wolfram/hello.htm)**

## Beispiel: „hello, world“-Programm in C++

```
#include <iostream.h>  
main()  
{  
    cout << „Hello World!“ << endl;  
    return 0;  
}
```

## Beispiel: „hello, world“-Programm in Oz

```
functor  
import  
    System  
    Application  
define  
    {System.showInfo „Hello World!“}  
    {Application.exit 0}  
end
```

## „hello, world“-Programme: Einleitung (2/3)

- Für unsere Zwecke sind die syntaktischen Details konkreter Programmiersprachen nicht so wichtig
- Wir beschreiben Programme deshalb in Pseudocode

Beispiel: „hello, world“-Programm in Pseudocode

```
BEGIN
    PRINT(„hello, world“)
END
```

### Definition


- Ein **„hello, world“-Programm** sei ein Programm, das keine Eingabe erwartet und als erstes „hello, world“ ausgibt
- **Frage:** Wie schwierig ist es, einem Programm anzusehen, ob es ein „hello, world“-Programm ist?
- **Was könnte daran schwierig sein???**

# „hello, world“-Programme: Einleitung (3/3)

## Beispiel: „hello, world“-Programm?


```
1:  $m := 3$ 
2: while TRUE do
3:   for  $n := 3$  TO  $m$  do
4:     for  $x := 1$  TO  $m$  do
5:       for  $y := 1$  TO  $m$  do
6:         for  $z := 1$  TO  $m$  do
7:           if  $x^n + y^n = z^n$  then
8:             PRINT(„hello, world“)
9:    $m := m + 1$ 
```

- Dieses Programm sucht systematisch natürliche Zahlen  $n, x, y, z$  mit
  - $n \geq 3$  und  $x^n + y^n = z^n$
- Wenn es solche Zahlen gibt, wird irgendwann „hello, world“ ausgegeben

 Zur Erinnerung: Natürliche Zahlen in dieser Vorlesung:  $1, 2, 3, \dots$

## Satz von Fermat

- Es gibt keine natürlichen Zahlen  $x, y, z \in \mathbb{N}$  und  $n \geq 3$  mit
$$x^n + y^n = z^n$$

 Der Beweis dieses Satzes hat 350 Jahre gedauert...

## Korollar

- Das Beispielprogramm ist kein „hello, world“-Programm
- Warum ist es so schwierig herauszufinden, ob dieses Programm ein „hello, world“-Programm ist?
- **Intuitive Schwierigkeit:** Im Beispiel-Programm gibt es unendlich viele Wertekombinationen für  $x, y, z, n$
- Diese können nicht in endlicher Zeit ausprobiert werden

# „hello, world“-Tester: Definition

- Herauszufinden, ob ein gegebenes Programm ein „hello, world“-Programm ist, ist also nicht ganz so leicht
  - Aber wir haben ja Computer!
- 
- Programme sind Zeichenketten (Strings) und können von anderen Programmen als Eingabe eingelesen werden
  - Schreiben wir also einfach ein Programm, das automatisch testet, ob ein gegebenes Programm ein „hello, world“-Programm ist

Definition: „hello, world“-Problem

**Gegeben:** Programm  $P$

**Frage:** Ist  $P$  ein „hello, world“-Programm?

- Wir nennen ein Programm für das „hello, world“-Problem einen „hello, world“-Tester
  - Ein „hello, world“-Tester gibt also bei Eingabe eines Programmes  $P$  die Antwort
    - \* „ja“, falls  $P$  ein „hello, world“-Programm ist
    - \* „nein“, falls  $P$  kein „hello, world“-Programm ist
- Ein „hello, world“-Tester würde also herausfinden, dass das zweite Beispiel-Programm kein „hello, world“-Programm ist
  - ➡ „hello, world“-Tester müssen ziemlich clever programmiert sein
- **Gibt es überhaupt „hello, world“-Tester?**
- Falls es keine „hello, world“-Tester gibt, **lässt sich das beweisen?**

# „hello, world“-Tester: Theorem (1/5)

## Theorem

- Es gibt keine „hello, world“-Tester
- Wir beweisen zuerst, dass es keine Tester für das folgende (scheinbar etwas schwierigere) Problem für Programme *mit Eingaben* gibt
- Danach zeigen wir, dass es dann auch keine „hello, world“-Tester gibt

## Definition: hw-Problem mit Eingabe

**Gegeben:** Programm  $P$ , Eingabe  $I$

**Frage:** Gibt  $P$  bei Eingabe  $I$  „hello, world“ aus?

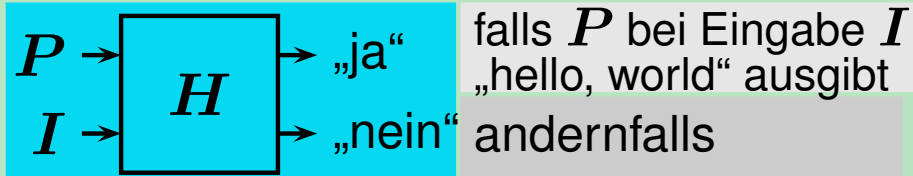
- Vereinbarung:
  - Programme lesen ihre Eingabe mit Anweisungen der Art „ $s := \text{READ}$ “
  - Jede solche Anweisung liest den jeweils nächsten String der Eingabe

- Wir beweisen also jetzt zuerst, dass es keinen Tester für das „hello, world“-Problem mit Eingabe gibt
- Wir führen einen Beweis durch Widerspruch
  - Wir nehmen an, es gäbe einen solchen Tester
  - Wir zeigen, dass sich daraus ein Widerspruch ergibt
  - Wir schließen daraus, dass die Annahme, es gäbe einen solchen Tester, falsch ist

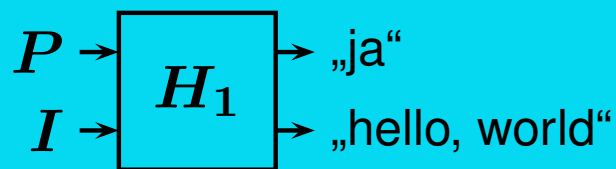
## „hello, world“-Tester: Theorem (2/5)

### „Beweis“

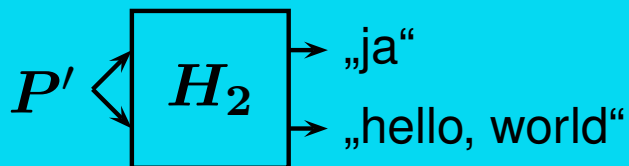
- Annahme: es gibt einen Tester  $H$  für das „hello, world“-Problem mit Eingabe:



- Wir können  $H$  in ein Programm  $H_1$  ändern, das wie  $H$  arbeitet, aber „hello, world“ anstelle von „nein“ ausgibt:

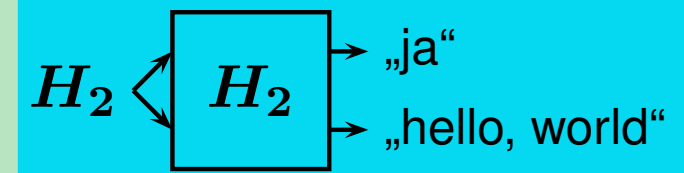


- Wir können  $H_1$  in ein Programm  $H_2$  ändern, das sich bei Eingabe eines Programmes  $P'$  so verhält wie  $H_1$  bei Eingabe  $P'$  (für  $P$ ) und  $P'$  (für  $I$ ):



### „Beweis“ (Forts.)

- Wie verhält sich  $H_2$  bei Eingabe  $H_2$ ?



Notation:  $H(P, I) \stackrel{\text{def}}{=} \text{Ausgabe von } H \text{ bei Eingabe } P \text{ und } I$

- 1. Fall:  $H_2(H_2) = \text{„ja“}$ 
  - $\Rightarrow H_1(H_2, H_2) = \text{„ja“}$
  - $\Rightarrow H(H_2, H_2) = \text{„ja“}$

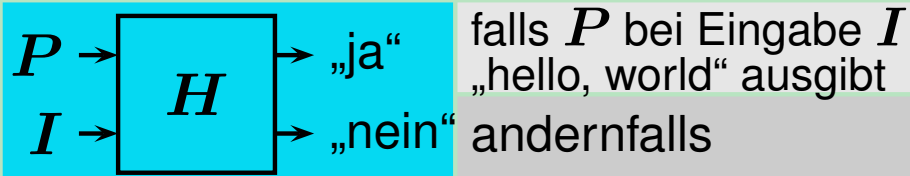
- $\Rightarrow H(H_2, H_2)$  ist falsch, denn:
  - \*  $H_2$  gibt bei Eingabe  $H_2$  nicht „hello, world“ aus
- $\Rightarrow$  **Widerspruch** zur Annahme, dass  $H$  ein Tester für das „hello, world“-Problem mit Eingabe ist

- Der erste Fall kann also nicht eintreten

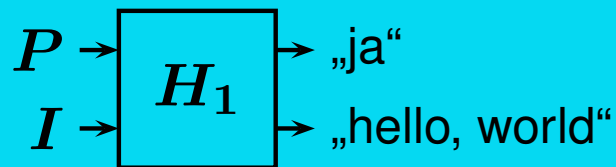
## „hello, world“-Tester: Theorem (3/5)

### „Beweis“ (Forts.)

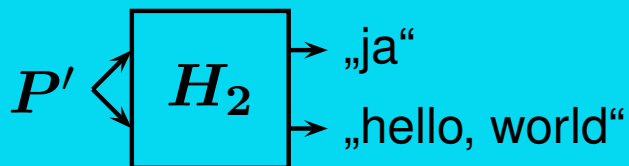
- Annahme: es gibt einen Tester  $H$  für das „hello, world“-Problem mit Eingabe:



- Wir können  $H$  in ein Programm  $H_1$  ändern, das wie  $H$  arbeitet, aber „hello, world“ anstelle von „nein“ ausgibt:

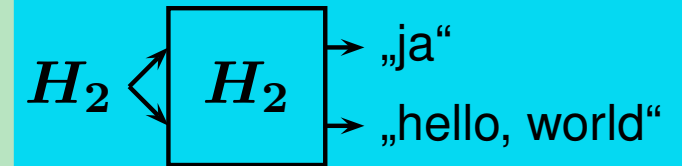


- Wir können  $H_1$  in ein Programm  $H_2$  ändern, das sich bei Eingabe eines Programmes  $P'$  so verhält wie  $H_1$  bei Eingabe  $P'$  (für  $P$ ) und  $P'$  (für  $I$ ):



### „Beweis“ (Forts.)

- Wie verhält sich  $H_2$  bei Eingabe  $H_2$ ?



- 2. Fall:  $H_2(H_2) = \text{„hello, world“}$ 
  - ➔  $H_1(H_2, H_2) = \text{„hello, world“}$
  - ➔  $H(H_2, H_2) = \text{„nein“}$

- ➔  $H(H_2, H_2)$  ist falsch, denn:
  - \*  $H_2$  gibt bei Eingabe  $H_2$  „hello, world“ aus

- ➔ **Widerspruch** zur Annahme, dass  $H$  ein Tester  $H$  für das „hello, world“-Problem mit Eingabe ist

- Der zweite Fall kann also auch nicht eintreten

- Die Annahme der Existenz eines Testers  $H$  führt also zu einem **Widerspruch**
  - ➔ Ein solcher Tester existiert nicht



## „hello, world“-Tester: Theorem (4/5)

### „Beweis“ (Forts.)

- Es gibt also keine Tester für hw-Programme mit Eingabe
- Dass es auch keine „hello, world“-Tester (für Programme ohne Eingabe) gibt, beweisen wir durch eine *Reduktion*
- Wir zeigen:
  - Wenn es einen „hello, world“-Tester  $H'$  (für Programme ohne Eingabe) gäbe, dann auch einen Tester  $H$  für das „hello, world“-Problem mit Eingabe

### „Beweis“ (Forts.)

- Denn um zu testen, ob ein Programm  $P$  mit Eingabe  $I$  „hello, world“ ausgibt, könnte  $H$  wie folgt vorgehen
- Konstruiere aus  $P$  ein Programm  $P_I$  ohne Eingabe:
  - Ersetze dazu die Anweisung „ $s := \text{READ}$ “ durch „ $s := I$ “
- Teste mit Hilfe von  $H'$ , ob  $P_I$  „hello, world“ ausgibt
- Falls „ja“: Ausgabe „ja“
- Falls „nein“: Ausgabe „nein“
- Da wir aber schon bewiesen haben, dass es keinen Tester für das „hello, world“-Problem mit Eingabe gibt, gibt es auch keinen Tester für das „hello, world“-Problem

## „hello, world“-Tester: „Theorem“ (5/5)

- Die Begriffe „Theorem“ und „beweisen“ stehen auf den vorhergehenden Folien in Anführungszeichen:
  - Um aus den Überlegungen der beiden letzten Folien wirklich ein Theorem und einen Beweis zu erhalten, müssten die verwendeten Begriffe präzise mathematische Definitionen haben
- Die Beweisidee lässt sich jedoch auf unsere formal definierten Berechnungsmodelle übertragen
- Denn der Beweis verwendet im Wesentlichen, dass Programme sich auf einfache Weisen modifizieren lassen
- Z.B.:
  - Modifikation der Ausgabe
  - Initialisierung des Programms mit einer Eingabe (statt Lesen der Eingabe)
- Wir werden nun zeigen, dass ein konkretes algorithmisches Problem, das auf Turingmaschinen basiert, unentscheidbar ist, und danach mit Hilfe von Reduktionen die Unentscheidbarkeit (vieler) anderer Probleme nachweisen

# Inhalt

14.1 Hello, world!-Programme

▷ **14.2 Ein erstes unentscheidbares Problem**

14.3 Reduktionen und weitere unentscheidbare Probleme

## Die „Diagonalsprache“ TM-DIAG (1/2)

- Wir beweisen jetzt für ein erstes konkretes Problem, dass es unentscheidbar ist
- Der Beweis verläuft ähnlich wie der informelle Beweis, dass es kein Programm zur Lösung des „hello, world“-Problems gibt
- Statt für Programme mit Eingabe zu fragen, ob sie „hello, world“ ausgeben, werden wir für Turingmaschinen  $M$  fragen, ob sie ihre eigene Kodierung durch einen String akzeptieren
- Im Folgenden betrachten wir Turingmaschinen ausschließlich über dem Ein-/Ausgabealphabet  $\Sigma = \{0, 1\}$ 
  - Die Resultate gelten aber entsprechend auch für jedes andere feste Alphabet

## Die „Diagonalsprache“ TM-DIAG (2/2)

- Wir nehmen im Folgenden an, dass wir eine Kodierung von Turingmaschinen zur Verfügung haben, die die folgenden Eigenschaften hat:
  - Für jede TM  $M$  gibt es einen String  $\text{enc}(M)$ , der sie kodiert
  - Jeder String  $w$  kodiert eine TM  $M_w$ 
    - ✎ Syntaktisch sinnlose Strings kodieren die TM, die immer sofort anhält und ablehnt
- Wie eine solche Kodierung konkret aussehen kann, betrachten wir in Kapitel 16

### Definition: TM-DIAG

**Gegeben:** Turingmaschine  $M$

**Frage:** Akzeptiert  $M$  die Eingabe  $\text{enc}(M)$ ?

### Satz 14.1

- TM-DIAG ist nicht entscheidbar
- Der Beweis verwendet die Methode der **Diagonalisierung**
- TM-DIAG scheint kein besonders interessantes algorithmisches Problem zu sein
  - Warum sollte es uns interessieren, ob eine TM „sich selbst“ akzeptiert?
- Das Resultat, dass TM-DIAG unentscheidbar ist, ist nur Mittel zum Zweck:
  - Wir werden alle weiteren Unentscheidbarkeitsresultate auf die Unentscheidbarkeit von TM-DIAG zurückführen

## TM-DIAG ist unentscheidbar (1/3)

- Im Beweis, dass TM-DIAG unentscheidbar ist, verwenden wir die folgende Aufzählung aller Strings über  $\Sigma^*$ 
    - $v_1 = \epsilon, v_2 = 0, v_3 = 1, v_4 = 00, \dots$
  - Statt  $M_{v_i}$  schreiben wir  $M_i$ 
    - $M_1, M_2, M_3, \dots$  ist also eine Aufzählung aller Turingmaschinen und für jedes  $i$  mit  $M_i \neq M_-$  gilt:  $\text{enc}(M_i) = v_i$
- ➡  $L_{\text{TM-DIAG}} = \{v_i \mid M_i \text{ akzeptiert die Eingabe } v_i\}$

# TM-DIAG ist unentscheidbar (2/3)

## Illustration der Beweisidee

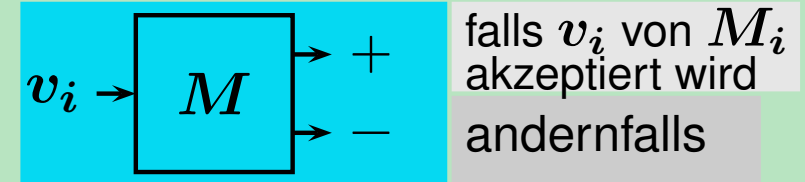
- Wir betrachten das Akzeptier- und Terminations-Verhalten von  $M_i$  bei Eingabe  $v_j$  für alle Kombinationen von  $i$  und  $j$ :

|          | $v_1$    | $v_2$    | $v_3$    | $v_4$    | $v_5$    | $\dots$  |
|----------|----------|----------|----------|----------|----------|----------|
| $M'$     | −        | +        | −        | +        | −        | $\dots$  |
| $M$      | +        | −        | +        | −        | +        | $\dots$  |
| $M_1$    | +        | −        | ⊥        | +        | −        | $\dots$  |
| $M_2$    | +        | −        | +        | ⊥        | −        | $\dots$  |
| $M_3$    | ⊥        | −        | +        | −        | +        | $\dots$  |
| $M_4$    | −        | +        | +        | ⊥        | −        | $\dots$  |
| $M_5$    | +        | −        | +        | −        | +        | $\dots$  |
| $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\ddots$ |

- +:  $M_i$  akzeptiert  $v_j$
- −:  $M_i$  lehnt  $v_j$  ab
- ⊥:  $M_i$  läuft bei Eingabe  $v_j$  endlos

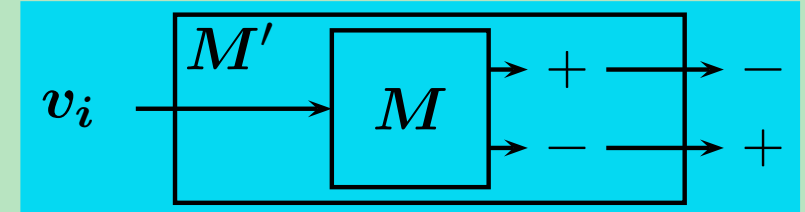
## Illustration der Beweisidee (Forts.)

- Annahme: es gibt eine TM  $M$  für TM-DIAG:



- $M$  hält immer an und akzeptiert  $v_i$  genau dann, wenn  $v_i$  von  $M_i$  akzeptiert wird

- Wir modifizieren  $M$  zu  $M'$  durch Umkehr des Akzeptierverhaltens:



- Dann gibt es  $\ell$  mit  $M_\ell = M'$  ( $\neq M_-$ )

➡ Dann sind äquivalent:

- $M_\ell$  akzeptiert  $v_\ell$
- $M'$  akzeptiert  $v_\ell$
- $M$  akzeptiert  $v_\ell$  nicht

$$\Rightarrow M' = M_\ell$$

➡ Widerspruch

➡ TM-DIAG ist nicht entscheidbar!

# TM-DIAG ist unentscheidbar (3/3)

- Wir beschreiben den Beweis nun noch einmal etwas ausführlicher

## Beweisskizze zu „TM-DIAG nicht entscheidbar“

- Um einen Widerspruch zu erreichen, nehmen wir an,  $M$  wäre eine Turing-Maschine, die TM-DIAG entscheidet
  - Zur Erinnerung:  $M$  müsste für alle Eingaben  $w$  anhalten und die richtige Antwort geben
- Sei  $M'$  die Turing-Maschine, die bei Eingabe  $w$  zuerst  $M$  bei Eingabe  $w$  simuliert und dann
  - akzeptiert, falls  $M$  ablehnt, aber
  - ablehnt, falls  $M$  akzeptiert
- Da  $M$  für jede Eingabe anhält (und akzeptiert oder ablehnt), gilt dies auch für  $M'$

## Beweisskizze (Forts.)

- 1. Fall:  $M' \in \text{TM-DIAG}$ 
  - ➡  $M$  akzeptiert  $\text{enc}(M')$ 
    - ☞ nach Annahme über  $M$
  - ➡  $M'$  lehnt  $\text{enc}(M')$  ab
    - ☞ nach Konstruktion von  $M'$
  - ➡  $M' \notin \text{TM-DIAG}$ 
    - ☞ nach Definition von TM-DIAG
  - ➡ Widerspruch

- 2. Fall:  $M' \notin \text{TM-DIAG}$ 
  - ➡  $M$  akzeptiert  $\text{enc}(M')$  nicht
    - ☞ nach Annahme über  $M$
  - ➡  $M'$  akzeptiert  $\text{enc}(M')$ 
    - ☞ nach Konstruktion von  $M'$
  - ➡  $M' \in \text{TM-DIAG}$ 
    - ☞ nach Definition von TM-DIAG
  - ➡ Widerspruch

- In beiden Fällen ergibt sich ein Widerspruch
  - ➡ TM-DIAG ist nicht entscheidbar



# Bedeutung des Begriffs Unentscheidbarkeit

- Wichtiger Hinweis:
  - Dass TM-DIAG unentscheidbar ist, bedeutet nur, dass es kein *allgemeines Verfahren* gibt, das für *alle* Eingaben  $M$  terminiert und entscheidet, ob  $M$  die Eingabe  $\text{enc}(M)$  akzeptiert
  - Für viele Turingmaschinen  $M$  lässt es sich durchaus herausfinden, ob sie „sich selbst akzeptieren“

# Inhalt

14.1 Hello, world!-Programme

14.2 Ein erstes unentscheidbares Problem

▷ **14.3 Reduktionen und weitere unentscheidbare Probleme**

## Weiteres Vorgehen

- Wie gesagt: die Unentscheidbarkeit von TM-DIAG ist erst der Anfang
- Unser Ziel ist jetzt, für interessantere Probleme zu zeigen, dass sie unentscheidbar sind
- Dafür werden wir als Zwischenschritt zunächst für zwei zu TM-DIAG ähnliche Probleme zeigen, dass sie unentscheidbar sind:
  - das Halteproblem für Turingmaschinen und
  - das Halteproblem für Turingmaschinen mit leerer Eingabe

### Definition: TM-HALT

**Gegeben:** Turingmaschine  $M$ , Eingabe  $x$  für  $M$

**Frage:** Hält  $M$  bei Eingabe  $x$  an?

### Definition: TM-E-HALT

**Gegeben:** Turingmaschine  $M$

**Frage:** Hält  $M$  bei Eingabe  $\epsilon$  an?

- Wir verwenden zum Nachweis der Unentscheidbarkeit zukünftig eine einfachere Methode als die „direkte Diagonalisierung“:  
**Reduktionen**

- Die Grundidee von Reduktionen ist, die Entscheidbarkeit eines Problems  $A$  auf die Entscheidbarkeit eines anderen Problems  $A'$  zurückzuführen

- Sie sollen uns Aussagen der folgenden Art ermöglichen:
  - wenn  $A'$  entscheidbar ist, ist dann auch  $A$  entscheidbar
- Daraus können wir dann folgern:
  - wenn  $A$  **nicht** entscheidbar ist, dann ist auch  $A'$  **nicht** entscheidbar

## Reduktionen

- Wir geben die formale Definition von Reduktionen für Sprachen
  - und erlauben uns dann, sie auch auf andere algorithmische Entscheidungsprobleme zu übertragen

### Definition

- Seien  $L, L' \subseteq \Sigma^*$
- Eine totale, berechenbare Funktion  $f : \Sigma^* \rightarrow \Sigma^*$  heißt Reduktion von  $L$  auf  $L'$ , wenn sie die folgende Reduktionseigenschaft hat:
  - für alle  $x \in \Sigma^*$  gilt:  $x \in L \iff f(x) \in L'$

- $L$  heißt auf  $L'$  reduzierbar, falls es eine Reduktion von  $L$  auf  $L'$  gibt
  - Notation:  $L \leq L'$

- Die Eigenschaft  $x \in L \iff f(x) \in L'$  lässt sich auch anders (aber äquivalent) formulieren:
  - Wenn  $x \in L$  dann  $f(x) \in L'$  und
  - wenn  $x \notin L$  dann  $f(x) \notin L'$

# Reduktionen: Erstes Beispiel (1/2)

- Wie gesagt: wir werden Reduktionen auch auf der Ebene algorithmischer Entscheidungsprobleme verwenden:
  - Sind  $\mathcal{A}, \mathcal{A}'$  zwei solche Probleme, so schreiben wir  $\mathcal{A} \leq \mathcal{A}'$ , falls  $L_{\mathcal{A}} \leq L_{\mathcal{A}'}$
- In Teil A der Vorlesung haben wir gesehen, dass sich das Nichtleerheitsproblem für endliche Automaten im Grunde wie das Erreichbarkeitsproblem für Graphen lösen lässt
- Diesen Zusammenhang präzisieren wir jetzt, indem wir zeigen, dass das Nichtleerheitsproblem auf das Erreichbarkeitsproblem reduzierbar ist

## Definition: DFA-NONEMPTY

**Gegeben:** DFA  $\mathcal{A}$

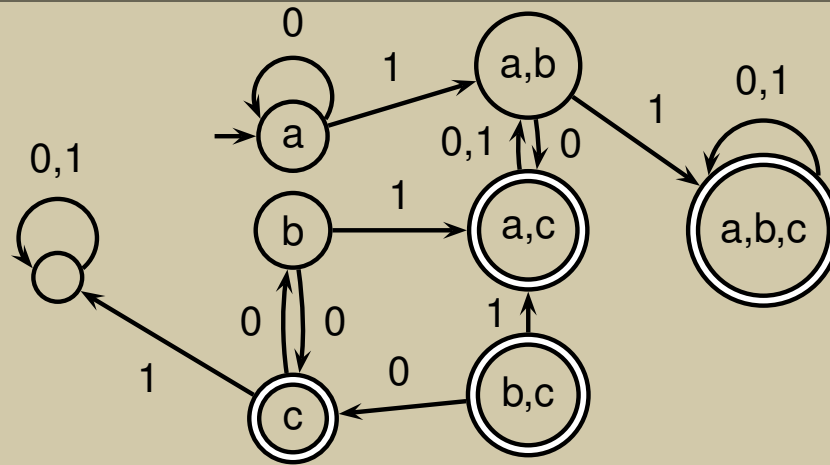
**Frage:** Ist  $L(\mathcal{A}) \neq \emptyset$ ?

## Beispiel

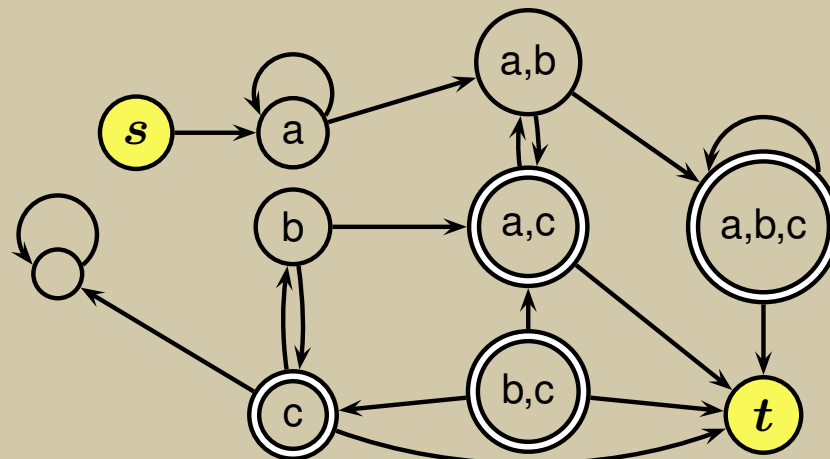
- Wir definieren eine Reduktionsfunktion um zu zeigen, dass  $\text{DFA-NONEMPTY} \leq \text{REACH}$  gilt:
  - Für  $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$  sei  $f(\mathcal{A}) \stackrel{\text{def}}{=} (G_{\mathcal{A}}, s, t)$ , wobei:
    - \*  $G_{\mathcal{A}} \stackrel{\text{def}}{=} (V_{\mathcal{A}}, E_{\mathcal{A}})$
    - \*  $V_{\mathcal{A}} \stackrel{\text{def}}{=} Q \cup \{s, t\}$
    - \*  $E_{\mathcal{A}} \stackrel{\text{def}}{=} \{(s, q_0)\} \cup \{(q, t) \mid q \in F\} \cup \{(q, q') \mid \delta(q, \sigma) = q', \sigma \in \Sigma\}$
- Dann gilt:
$$\mathcal{A} \in \text{DFA-NONEMPTY} \iff f(\mathcal{A}) \in \text{REACH}$$
- Und natürlich ist  $f$  berechenbar

# Reduktionen: Erstes Beispiel (2/2)

Beispiel:  $L_1$  (Automaten)



Beispiel:  $L_2$  (Graphen)



# Reduktionen: Zweites Beispiel (1/2)

## Satz 14.2

- $\text{PCP} \leq \text{CFG-SCHNITT}$

## Beweisskizze

- Sei  $(u_1, v_1), \dots, (u_k, v_k)$  eine Eingabe für PCP
  - (OBdA:  $\$ \notin \Sigma$ )
- Idee: Wir konstruieren Grammatiken  $G_1$  und  $G_2$  so, dass gilt:
  - $L(G_1)$  enthält alle Strings der Form
$$u_{i_1} \cdots u_{i_n} \$ i_n \cdots i_1, \text{ mit } n \geq 1$$
  - $L(G_2)$  enthält alle Strings der Form
$$v_{i_1} \cdots v_{i_n} \$ i_n \cdots i_1, \text{ mit } n \geq 1$$
- $G_1: S_1 \rightarrow u_1 S_1 1 \mid \cdots \mid u_k S_1 k \mid u_1 \$ 1 \mid \cdots \mid u_k \$ k$
- $G_2: S_2 \rightarrow v_1 S_2 1 \mid \cdots \mid v_k S_2 k \mid v_1 \$ 1 \mid \cdots \mid v_k \$ k$
- Dann sind äquivalent:
  - $(u_1, v_1), \dots, (u_k, v_k)$  hat eine PCP-Lösung
  - $L(G_1) \cap L(G_2) \neq \emptyset$

## Reduktionen: Zweites Beispiel (2/2)

### Beispiel

- Steintypen: 

|            |
|------------|
| <i>a</i>   |
| <i>aba</i> |

, 

|           |
|-----------|
| <i>ab</i> |
| <i>bb</i> |

, 

|            |
|------------|
| <i>baa</i> |
| <i>aa</i>  |
- $G_1$ :  
–  $S_1 \rightarrow aS_11 \mid abS_12 \mid baaS_13 \mid$   
 $a\$1 \mid ab\$2 \mid baa\$3$
- $G_2$ :  
–  $S_2 \rightarrow abaS_21 \mid bbS_22 \mid aaS_23 \mid$   
 $aba\$1 \mid bb\$2 \mid aa\$3$
- Mögliche Lösung: 

|            |
|------------|
| <i>a</i>   |
| <i>aba</i> |

|            |
|------------|
| <i>baa</i> |
| <i>aa</i>  |

|           |
|-----------|
| <i>ab</i> |
| <i>bb</i> |

|            |
|------------|
| <i>baa</i> |
| <i>aa</i>  |
- Zugehöriger String in  $L(G_1) \cap L(G_2)$ :  
 $abaaabbaa\$3231$

### Bemerkung

- Bei beiden Beispielen ist  $f$  formal nur für Strings definiert, die „vernünftige“ Eingaben für DFA-NONEMPTY bzw. PCP kodieren der Form  $\text{enc}(M)\#x$  definiert
- Wir können es aber zu einer totalen Funktion erweitern 14.1




# Reduktionen und unentscheidbare Probleme

- Informelle Interpretation von Reduktionen:
  - Aus  $A \leq A'$  folgt:
    - \* Falls es ein „Unterprogramm“ für  $A'$  gibt, so auch ein Programm für  $A$
  - Falls  $A \leq A'$  ist also in einem gewissen Sinne  $A$  nicht schwieriger als  $A'$

## Lemma 14.3

- Sind  $L, L'$  Sprachen mit  $L \leq L'$ , so gilt:
  - (a) Ist  $L'$  entscheidbar, dann auch  $L$
  - (b) Ist  $L$  unentscheidbar, dann auch  $L'$

- Um zu beweisen, dass ein Entscheidungsproblem  $A'$  unentscheidbar ist, genügt es also für ein schon als unentscheidbar bekanntes Problem  $A$  zu zeigen:  $A \leq A'$

 Vorsicht, sprachliche Fehlerquelle: wir führen die Unentscheidbarkeit von  $A'$  auf die Unentscheidbarkeit von  $A$  zurück, indem wir zeigen, dass  $A$  auf  $A'$  reduzierbar ist!

## Beweisidee

- (a) Sei  $f$  eine Reduktion von  $L$  auf  $L'$ 
  - Entscheidungs-Algorithmus für  $L$ :
    - \* Bei Eingabe  $w$ , berechne  $f(w)$
    - \* Teste  $f(w) \in L'$  mit Hilfe eines Entscheidungsalgorithmus für  $L'$
    - \* Akzeptiere, falls ja, lehne ab, falls nein

(b) Kontraposition von (a)

- Wir werden (unter anderem) zeigen:
  - $\text{TM-DIAG} \leq \text{TM-HALT} \leq \text{TM-E-HALT} \leq \text{PCP}$
- Durch mehrfache Anwendung von Lemma 14.3 und mit  $\text{PCP} \leq \text{CFG-SCHNITT}$  folgt dann, dass CFG-SCHNITT unentscheidbar ist

## Weitere unentscheidbare Probleme (1/2)

### Satz 14.4

- TM-HALT ist nicht entscheidbar

### Beweisskizze

- Wir zeigen:  
 $\text{TM-DIAG} \leq \text{TM-HALT}$ 
  - Dann folgt die Behauptung mit Lemma 14.3
- Prinzipielle Idee:  
 $M \mapsto (M, \text{enc}(M))$
- Komplikation:  $M$  könnte bei Eingabe  $\text{enc}(M)$  anhalten und *ablehnen*
- Dann wäre  $M \notin \text{TM-DIAG}$  aber  $(M, \text{enc}(M)) \in \text{TM-HALT}$
- Deshalb modifizieren wir die TM  $M$  so, dass sie nie anhält und ablehnt

### Beweisskizze (Forts.)

- Für eine TM  $M$  sei  $M'$  die TM, in der alle Transitionen  $\delta(q, \sigma) = (\text{nein}, d, \tau)$  durch Transitionen  $\delta(q, \sigma) = (q, \downarrow, \sigma)$  ersetzt werden
- Dadurch wird erreicht, dass
  - $M'$  anhält und akzeptiert, falls  $M$  akzeptiert, und
  - $M'$  nicht anhält, falls  $M$  ablehnt oder nicht anhält
- Wir definieren die Funktion  $f$  durch:  
$$f(M) \stackrel{\text{def}}{=} (M', \text{enc}(M))$$
- Dann gilt:

$$M \in \text{TM-DIAG}$$

$$\iff M \text{ akzeptiert } \text{enc}(M)$$

$$\iff M' \text{ hält bei Eingabe } \text{enc}(M) \text{ an}$$

$$\iff f(M) \in \text{TM-HALT}$$

## Weitere unentscheidbare Probleme (2/2)

### Satz 14.5

- TM-E-HALT ist nicht entscheidbar

### Beweisskizze

- Wir zeigen:  $\text{TM-HALT} \leq \text{TM-E-HALT}$
- Für jede TM  $M$  und jeden String  $x \in \Sigma^*$  sei  $M_{M,x}$  die TM, die
  - ihre eigentliche Eingabe löscht,
  - stattdessen  $x$  auf ihren String schreibt,
  - und dann  $M$  bei Eingabe  $x$  simuliert
- $f$  sei definiert durch:
$$f((M, x)) \stackrel{\text{def}}{=} M_{M,x}$$

### Beweisskizze (Forts.)

- $f$  ist eine Reduktion von TM-HALT auf TM-E-HALT:
  - $f$  ist total und berechenbar ✓
  - Es gilt:
    - $(M, x) \in \text{TM-HALT}$
    - $\iff M$  hält bei Eingabe  $x$
    - $\iff M_{M,x}$  hält bei Eingabe  $\epsilon$
    - $\iff M_{M,x} \in \text{TM-E-HALT}$
    - $\iff f((M, x)) \in \text{TM-E-HALT}$

# Zusammenfassung

- Definition der Begriffe **entscheidbar** und **unentscheidbar**
- Auf ähnliche Weise, wie wir uns von der algorithmischen Unlösbarkeit des „hello, world“-Problems überzeugt haben, lässt sich zeigen, dass das **Halte-Problem für Turing-Maschinen unentscheidbar** ist
- Für viele andere Probleme lässt sich die Unentscheidbarkeit mit Hilfe von **Reduktionen** beweisen

# Erläuterungen

## Bemerkung 14.1

- Wenn wir eine Reduktionsfunktion  $f$  von einem algorithmischen Problem  $A$  auf ein Problem  $A'$  angeben, geben wir  $f(x)$  nur für syntaktisch korrekte Eingaben  $x$  für  $A$  an
  - Wenn  $A$  einen Graphen als Eingabe „erwartet“, werden wir  $f(G)$  also nur für Graphen  $G$  definieren
- Daraus können wir dann wie folgt eine totale Reduktionsfunktion  $f' : \Sigma^* \rightarrow \Sigma^*$  gewinnen:
  - Für syntaktisch korrekte Eingaben  $w = \text{enc}(G)$  ergibt sich dann  $f'(w) \stackrel{\text{def}}{=} \text{enc}(f(G))$
  - Für Strings  $w$ , die keinen Graphen kodieren, setzen wir  $f(w) \stackrel{\text{def}}{=} y$  für ein festes  $y \notin L_{A'}$

# Grundbegriffe der Theoretischen Informatik

Sommersemester 2016 - Thomas Schwentick

Teil C: Berechenbarkeit und Entscheidbarkeit

15: Unentscheidbare Probleme 2

Version von: 16. Juni 2016 (13:56)

# Einleitung

- In Teil A der Vorlesung haben wir gesehen, dass es möglich ist, automatisch zu überprüfen, ob die Implementierung einer regulären Sprache (durch einen DFA) korrekt ist bezüglich der Spezifikation (durch einen regulären Ausdruck)
- Es ist auch möglich,
  - eine „optimale“ Implementierung aus einer Spezifikation zu konstruieren
  - zu testen, ob zwei Spezifikationen äquivalent sind
  - zu testen, ob zwei Implementierungen äquivalent sind
- Natürlich wäre es schön, wenn dies alles auch allgemeiner möglich wäre:
  - für Programme/Algorithmen statt DFAs und
  - für allgemeine semantische Spezifikationen statt reguläre Ausdrücke
- In diesem Kapitel werden wir sehen:
  - Für allgemeine Programme und Spezifikationen ist **nichts** von dem möglich
  - Semantische Eigenschaften von Programmen sind nicht automatisch überprüfbar
    - \* Das ist unterm Strich die Aussage des Satzes von Rice
- Wir werden sogar zeigen, dass die Unmöglichkeit von Äquivalenztests schon für kontextfreie Sprachen und ihre Beschreibungsformen gilt

# Inhalt

## ▷ 15.1 Der Satz von Rice

15.2 Das Postsche Korrespondenzproblem

15.3 Unentscheidbare Grammatikprobleme



## Satz von Rice (1/3)

- Wir zeigen jetzt, dass jede nicht-triviale semantische Eigenschaft von Algorithmen (Programmen, Turingmaschinen) unentscheidbar ist
- Wir betrachten dazu zunächst Turingmaschinen, die Funktionen berechnen
- Semantische Eigenschaften formalisieren wir durch Mengen von Funktionen
- Für jede Menge  $S \subseteq \mathcal{R}$  berechenbarer Funktionen definieren wir das folgende algorithmische Problem

Definition:  $\text{TM-FUNC}(S)$

**Gegeben:** Turingmaschine  $M$

**Frage:** Ist  $f_M \in S$ ?

### Satz 15.1 [Rice 53]

- Sei  $S$  eine Menge berechenbarer partieller Funktionen mit  $\emptyset \neq S \neq \mathcal{R}$
- Dann ist  $\text{TM-FUNC}(S)$  unentscheidbar
- Wählt man beispielsweise  $S$  als Menge aller totalen, berechenbaren Funktionen, so folgt mit dem Satz von Rice, dass es unentscheidbar ist, ob eine gegebene TM eine totale Funktion berechnet
  - Das ist natürlich nicht sehr überraschend, aber wir werden gleich noch weitere Anwendungen betrachten

# Satz von Rice: Anwendungen

- Aus dem Satz von Rice lassen sich viele Unentscheidbarkeitsresultate folgern, beispielsweise:
- Für jede feste berechenbare Funktion  $f$  ist es unentscheidbar, ob ein gegebenes Programm  $f$  berechnet:
  - Wähle  $S = \{f\}$
- Für beliebige feste Strings  $u, w$  ist es unentscheidbar, ob ein gegebenes Programm bei Eingabe  $u$  die Ausgabe  $w$  hat:
  - Wähle  $S = \{f \mid f(u) = w\}$
- Es ist unentscheidbar, ob zwei gegebene Programme äquivalent sind:
  - Das folgt direkt aus der Unentscheidbarkeit des ersten Problems
  - Dieses lässt sich nämlich auf die Äquivalenz zweier Programme reduzieren

- Auf ähnliche Weise folgt die Unentscheidbarkeit der in der Einleitung genannten Probleme
- Dass wir hier „Programm“ statt „TM“ schreiben, ist durch die (empirische Gültigkeit der) Church-Turing-These gerechtfertigt

## Satz von Rice (2/3)

### Beweisskizze

- Sei  $f_{\perp}(w) \stackrel{\text{def}}{=} \perp$ , für alle  $w \in \Sigma^*$
- 1. Fall:  $f_{\perp} \notin S$ :
  - Sei  $f \in S$  beliebig,  $M_f$  TM für  $f$
- Wir zeigen:  $\text{TM-E-HALT} \leq \text{TM-FUNC}(S)$
- Also: bilde  $M$  so auf  $M'$  ab, dass gilt:
  - $M(\epsilon)$  terminiert  $\Rightarrow M'$  berechnet  $f$
  - $M(\epsilon)$  terminiert nicht  $\Rightarrow M'$  berechnet  $f_{\perp}$
- Für jede TM  $M$  sei dazu  $M'$  die TM, die bei Eingabe  $x$ 
  - (1) zuerst  $M$  bei Eingabe  $\epsilon$  simuliert (allerdings „hinter  $x$ “ und ohne  $x$  zu verändern)
  - (2) und falls diese Berechnung terminiert, die TM  $M_f$  bei Eingabe  $x$  simuliert
- Ein solches  $M'$  kann auch mit Satz 13.3 aus einer geeigneten 2-TM gewonnen werden

### Beweisskizze (Forts.)

- Wir zeigen, dass  $M \mapsto M'$  eine Reduktion von  $\text{TM-E-HALT}$  auf  $\text{TM-FUNC}(S)$  ist
- Sei  $M \in \text{TM-E-HALT}$ 
  - ➔  $M(\epsilon)$  terminiert
  - ➔ Phase (1) von  $M'$  terminiert
  - ➔ Phase (2) simuliert das Verhalten von  $M_f$  bei Eingabe  $x$  und gibt also  $f(x)$  aus (falls das definiert ist)
  - ➔  $M'$  berechnet  $f$
  - ➔  $M' \in \text{TM-FUNC}(S)$
- Sei nun  $M \notin \text{TM-E-HALT}$ 
  - ➔  $M(\epsilon)$  terminiert nicht
  - ➔  $M'$  berechnet  $f_{\perp}$
  - ➔  $M' \notin \text{TM-FUNC}(S)$

## Satz von Rice (3/3)

### Beweisskizze (Forts.)

- Im zweiten Fall ( $f_{\perp} \in S$ ) lässt sich analog zeigen, dass TM-E-HALT auf das Komplement  $\overline{\text{TM-FUNC}(S)}$  reduzierbar ist
  - ➡  $\overline{\text{TM-FUNC}(S)}$  ist unentscheidbar
  - ➡  $\text{TM-FUNC}(S)$  ist unentscheidbar

# Inhalt

15.1 Der Satz von Rice

▷ **15.2 Das Postsche Korrespondenzproblem**

15.3 Unentscheidbare Grammatikprobleme

# PCP ist unentscheidbar (1/7)

- Zur Erinnerung:

Def.: Postsches Korrespondenzproblem (PCP)

**Gegeben:** eine Folge

$(u_1, v_1), \dots, (u_k, v_k)$  von Paaren nicht-leerer Strings

**Frage:** Gibt es eine Indexfolge  $i_1, \dots, i_n$  mit  $n \geq 1$ , so dass

$$u_{i_1} u_{i_2} \cdots u_{i_n} = v_{i_1} v_{i_2} \cdots v_{i_n}?$$

Satz 15.2

- PCP ist unentscheidbar

Beweisidee

- Wir definieren ein „Zwischen-Problem“ SPCP und zeigen:
  - $\text{TM-HALT} \leq \text{SPCP}$  und
  - $\text{SPCP} \leq \text{PCP}$
- Daraus folgt mit Lemma 14.3, dass SPCP und dann auch PCP unentscheidbar sind

Def.: PCP-Problem mit Startpaar (SPCP)

**Gegeben:** eine Folge

$(u_1, v_1), \dots, (u_k, v_k)$  von Paaren von Strings

**Frage:** Gibt es eine Indexfolge  $i_1, \dots, i_n$  mit  $n \geq 1$ , so dass


- $u_{i_1} u_{i_2} \cdots u_{i_n} = v_{i_1} v_{i_2} \cdots v_{i_n}$
- und  $i_1 = 1$ ?

# Die erste Beispiel-TM (leicht modifiziert)

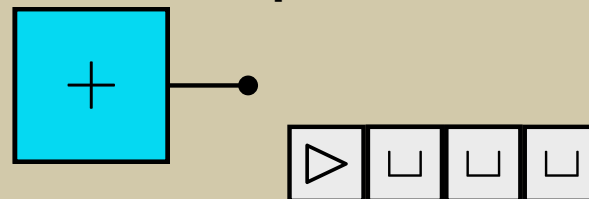
## Beispiel

Turing-Maschine zum Test, ob Eingabe von der Form  $ww^R$  ist

- a:** **0** und **1** überlesen, nach links — falls  $\triangleright$  oder  $\sqcup$  nach rechts in b
- b:** Falls **0** nach rechts in c — falls **1** nach rechts in d (**0/1** durch  $\sqcup$  überschreiben) — falls  $\sqcup$  akz.
- c:** **0** und **1** überlesen nach rechts bis  $\sqcup$ , dann nach links in e
- d:** **0** und **1** überlesen nach rechts bis  $\sqcup$ , dann nach links in f
- e:** Falls **0** durch  $\sqcup$  ersetzen nach links in a — falls **1** oder  $\sqcup$  ablehnen
- f:** Falls **1** durch  $\sqcup$  ersetzen nach links in a — falls **0** oder  $\sqcup$  ablehnen

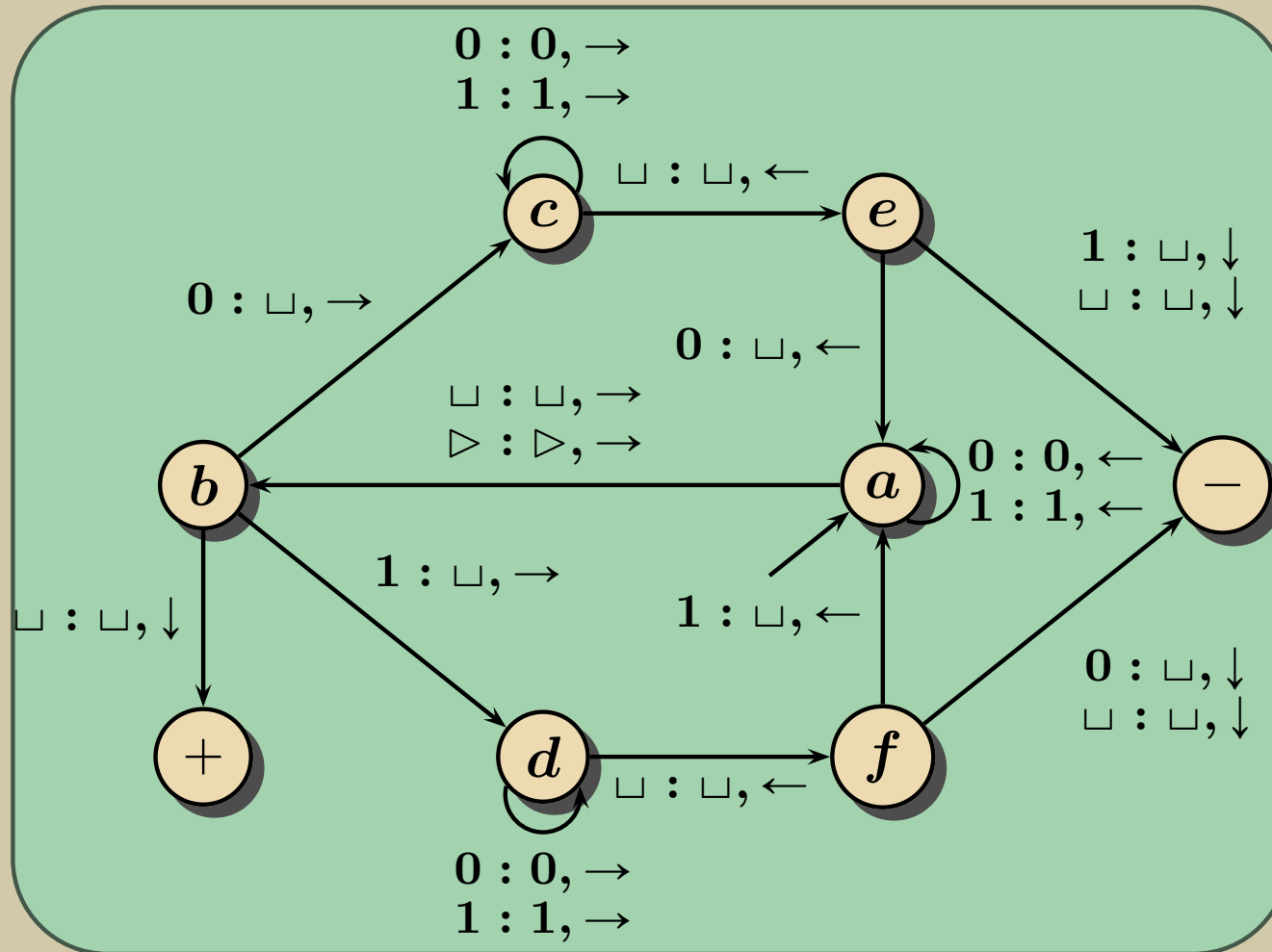
 Die TM nutzt  $\sqcup$  statt  $\#$ , da wir  $\#$  für die Konkatenation verwenden wollen

### Beispielberechnung:



# Beispiel-TM als Diagramm

## Beispiel-Turingmaschine als Diagramm

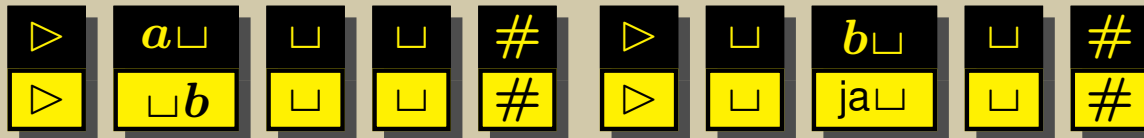
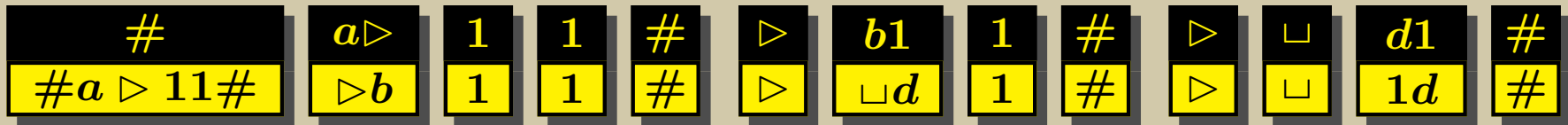




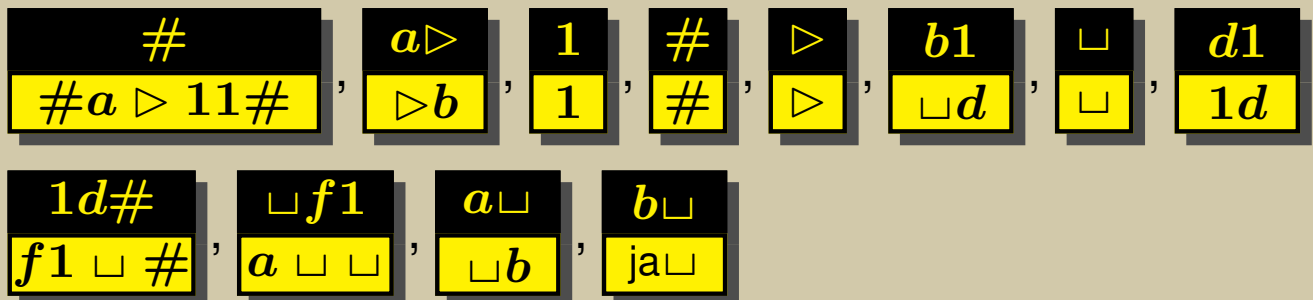
# Beispielberechnung als Präfix einer Lösung

## Beispiel (Forts.)

- Terminierende Berechnung von  $M$  bei Eingabe **11** als Präfix einer SPCP-Lösung
- Präfix des Lösungsstrings:



- Verwendete Steintypen:



# PCP ist unentscheidbar (2/7)

## Beweisskizze (Forts.)

- Wir konstruieren eine Reduktionsfunktion  $f$  mit:  
 $(M, x) \in \text{TM-HALT} \iff f((M, x)) \in \text{SPCP}$
- Lösungen von  $f((M, x))$  entsprechen also terminierenden Berechnungen  $M(x)$

- Wir kodieren dazu Konfigurationen  $K = (q, (y, \sigma, z))$  von  $M$  durch Strings  $\hat{K} \stackrel{\text{def}}{=} \triangleright yq\sigma z$

- Idee der Reduktion:
- Sei  $M(x) = K_0 \vdash_M \dots \vdash_M K_t$   
 $(K_0 \stackrel{\text{def}}{=} K_0(x))$
- Lösungsstrings für  $f((M, x))$  beginnen mit der Konkatination der Konfigurationen dieser Berechnung:

$$\hat{K}_0 \# \hat{K}_1 \# \dots \# \hat{K}_t$$

☞ über das Ende reden wir später

## Beweisskizze (Forts.)

- Für jedes  $\ell < t$ 
  - ist  $\hat{K}_0 \# \hat{K}_1 \# \dots \# \hat{K}_\ell$  ein Präfix  $u_{i_1} \dots u_{i_k}$  des Lösungsstrings
  - und  $v_{i_1} \dots v_{i_k}$  ist  $\hat{K}_0 \# \hat{K}_1 \# \dots \# \hat{K}_{\ell+1}$

- Also übereinander geschrieben:

$$\begin{array}{ccccccc} \# & \hat{K}_0 \# & \hat{K}_1 \# & \dots & \hat{K}_\ell \# \\ \# \hat{K}_0 \# & \hat{K}_1 \# & \hat{K}_2 \# & \dots & \hat{K}_{\ell+1} \# \end{array}$$

- Es gilt dann für jedes  $i$ :  
wenn  $u_i$  eine Position in einer Konfiguration  $K_j$  repräsentiert, dann repräsentiert  $v_i$  dieselbe Position in  $K_{j+1}$
- Dieses „Übereinanderlegen“ aufeinanderfolgender Konfigurationen ermöglicht es, die Konfigurationsfolge auf Korrektheit zu testen

# PCP ist unentscheidbar (3/7)

## Beweisskizze (Forts.)

- Sei  $M = (Q, \Gamma, \delta, s)$  TM,  $x \in \Sigma^*$ 
  - OBdA:  $\# \notin \Gamma$
- Die zugehörige SPCP-Eingabe hat das Alphabet  $Q \cup \Gamma \cup \{\#\}$  und die folgenden Regeln:

(1) Start: 

|         |
|---------|
| #       |
| #s ▷ x# |

(2) Kopierregeln: 

|          |
|----------|
| $\sigma$ |
| $\sigma$ |

  
für jedes  $\sigma \in \Gamma \cup \{\#\}$

(3)  $\delta$ -Regeln: für alle  $q, q' \in Q$   
und alle  $\sigma, \sigma', \tau \in \Gamma \cup \{\#\}$ :

- falls  $\delta(q, \sigma) = (q', \sigma', \downarrow)$ : 

|             |
|-------------|
| $q\sigma$   |
| $q'\sigma'$ |
- falls  $\delta(q, \sigma) = (q', \sigma', \rightarrow)$ : 

|             |
|-------------|
| $q\sigma$   |
| $\sigma'q'$ |
- falls  $\delta(q, \sigma) = (q', \sigma', \leftarrow)$ :  

|                 |
|-----------------|
| $\tau q\sigma$  |
| $q'\tau\sigma'$ |

 für alle  $\tau \in \Gamma$ ,

## Beweisskizze (Forts.)

(4)  $\delta$ -Regeln für den „rechten Rand“:

Für alle  $q, q' \in Q$  und alle  $\sigma', \tau \in \Gamma \cup \{\#\}$ :

- falls  $\delta(q, \sqcup) = (q', \sigma', \downarrow)$ : 

|               |
|---------------|
| $q\#$         |
| $q'\sigma'\#$ |
- falls  $\delta(q, \sqcup) = (q', \sigma', \rightarrow)$ : 

|               |
|---------------|
| $q\#$         |
| $\sigma'q'\#$ |
- falls  $\delta(q, \sqcup) = (q', \sigma', \leftarrow)$ :  

|                   |
|-------------------|
| $\tau q\#$        |
| $q'\tau\sigma'\#$ |



 für alle  $\tau \in \Gamma$ ,

## PCP ist unentscheidbar (4/7)

- Wie gesagt: der erste Teil des Lösungsstrings ist die Konkatenation der Kodierungen aller Konfigurationen der terminierenden Berechnung:

$$\begin{array}{ccccccc} \# & \hat{K}_0\# & \hat{K}_1\# & \cdots & \hat{K}_{t-1}\# \\ \# \hat{K}_0\# & \hat{K}_1\# & \hat{K}_2\# & \cdots & \hat{K}_t\# \end{array}$$

- Dies ist aber noch kein Lösungsstring, da der „ $u$ -String“ nur ein Präfix des  $v$ -Strings ist
  - Der  $v$ -String ist genau um  $\hat{K}_t\#$  länger
  - Da die Konfiguration  $K_t$  (im Gegensatz zu  $K_0$ ) nicht bekannt ist, kann diese Lücke nicht durch eine einzelne Regel der SPCP-Eingabe überbrückt werden

- Stattdessen verwenden wir zusätzliche Löschregeln der Arten  und , durch deren Anwendung immer kürzere Teilstrings von  $\hat{K}_t$  entstehen, bis der  $v$ -String nur noch zwei Zeichen länger ist als der  $u$ -String

- Sei dazu  $C_0 \stackrel{\text{def}}{=} \hat{K}_t$  und entstehe  $C_{i+1}$  aus  $C_i$  jeweils durch Löschen *eines* Nachbarzeichen des Zustandssymbolen (ja, nein, oder  $h$ ), und sei  $C_c \in \{\text{ja, nein, } h\}$

- Insgesamt ergibt sich dann folgende Korrespondenz:

$$\begin{array}{cccccccccccccccc} \# & \hat{K}_0\# & \hat{K}_1\# & \cdots & \hat{K}_{t-1}\# & \hat{K}_t\# & C_1\# & \cdots & C_{c-1}\# & C_c\#\# & = \\ \# \hat{K}_0\# & \hat{K}_1\# & \hat{K}_2\# & \cdots & \hat{K}_t\# & C_1\# & C_2\# & \cdots & C_c\# & \# \end{array}$$

# PCP-Lösungsstring für die Beispielberechnung

## Beispiel (Forts.)

- Terminierende Berechnung von  $M$  bei Eingabe **11** als SPCP-Lösung
- Lösungsstring:

|        |    |   |   |   |   |    |   |   |   |   |    |   |
|--------|----|---|---|---|---|----|---|---|---|---|----|---|
| #      | a▷ | 1 | 1 | # | ▷ | b1 | 1 | # | ▷ | □ | d1 | # |
| #a▷11# | ▷b | 1 | 1 | # | ▷ | □d | 1 | # | ▷ | □ | 1d | # |

|   |   |   |      |   |     |   |   |
|---|---|---|------|---|-----|---|---|
| ▷ | □ | 1 | 1d#  | ▷ | □f1 | □ | # |
| ▷ | □ | 1 | f1□# | ▷ | a□□ | □ | # |

|   |    |   |   |   |   |   |     |   |   |
|---|----|---|---|---|---|---|-----|---|---|
| ▷ | a□ | □ | □ | # | ▷ | □ | b□  | □ | # |
| ▷ | □b | □ | □ | # | ▷ | □ | ja□ | □ | # |

|   |     |   |   |   |     |   |   |   |
|---|-----|---|---|---|-----|---|---|---|
| ▷ | □ja | □ | □ | # | ▷ja | □ | □ | # |
| ▷ | ja  | □ | □ | # | ja  | □ | □ | # |

|     |   |   |     |   |      |
|-----|---|---|-----|---|------|
| ja□ | □ | # | ja□ | # | ja## |
| ja  | □ | # | ja  | # | #    |

- Verwendete Steintypen:

|        |    |   |   |   |    |   |    |
|--------|----|---|---|---|----|---|----|
| #      | a▷ | 1 | # | ▷ | b1 | □ | d1 |
| #a▷11# | ▷b | 1 | # | ▷ | □d | □ | 1d |

|      |     |    |     |     |     |     |      |
|------|-----|----|-----|-----|-----|-----|------|
| 1d#  | □f1 | a□ | b□  | □ja | ▷ja | ja□ | ja## |
| f1□# | a□□ | □b | ja□ | ja  | ja  | ja  | #    |

# PCP ist unentscheidbar (5/7)

## Beweisskizze (Forts.)

- Sei  $M = (Q, \Gamma, \delta, s)$  TM,  $x \in \Sigma^*$ 
  - OBdA:  $\# \notin \Gamma$
- Die zugehörige SPCP-Eingabe hat das Alphabet  $Q \cup \Gamma \cup \{\#\}$  und die folgenden Regeln:

(1) Start: 

|         |
|---------|
| #       |
| #s ▷ x# |

(2) Kopierregeln: 

|          |
|----------|
| $\sigma$ |
| $\sigma$ |

  
für jedes  $\sigma \in \Gamma \cup \{\#\}$

(3)  $\delta$ -Regeln: für alle  $q, q' \in Q$   
und alle  $\sigma, \sigma', \tau \in \Gamma \cup \{\#\}$ :

- falls  $\delta(q, \sigma) = (q', \sigma', \downarrow)$ : 

|             |
|-------------|
| $q\sigma$   |
| $q'\sigma'$ |
- falls  $\delta(q, \sigma) = (q', \sigma', \rightarrow)$ : 

|             |
|-------------|
| $q\sigma$   |
| $\sigma'q'$ |
- falls  $\delta(q, \sigma) = (q', \sigma', \leftarrow)$ : 

|                 |
|-----------------|
| $\tau q\sigma$  |
| $q'\tau\sigma'$ |

 für alle  $\tau \in \Gamma$ ,

## Beweisskizze (Forts.)

(4)  $\delta$ -Regeln für den „rechten Rand“:

Für alle  $q, q' \in Q$  und alle  $\sigma', \tau \in \Gamma \cup \{\#\}$ :

- falls  $\delta(q, \sqcup) = (q', \sigma', \downarrow)$ : 

|               |
|---------------|
| $q\#$         |
| $q'\sigma'\#$ |
- falls  $\delta(q, \sqcup) = (q', \sigma', \rightarrow)$ : 

|               |
|---------------|
| $q\#$         |
| $\sigma'q'\#$ |
- falls  $\delta(q, \sqcup) = (q', \sigma', \leftarrow)$ : 

|                   |
|-------------------|
| $\tau q\#$        |
| $q'\tau\sigma'\#$ |

 für alle  $\tau \in \Gamma$ ,

(5) Löschregeln:

|             |
|-------------|
| $\sigma ja$ |
| ja          |

, 

|             |
|-------------|
| ja $\sigma$ |
| ja          |

, 

|               |
|---------------|
| $\sigma nein$ |
| nein          |

, 

|               |
|---------------|
| nein $\sigma$ |
| nein          |

, 

|            |
|------------|
| $\sigma h$ |
| h          |

, 

|            |
|------------|
| h $\sigma$ |
| h          |

  
für alle  $\sigma \in \Gamma \cup \{\#\}$

(6) Abschlussregeln:

|      |
|------|
| ja## |
| #    |

, 

|        |
|--------|
| nein## |
| #      |

, 

|     |
|-----|
| h## |
| #   |

# PCP ist unentscheidbar (6/7)

## Beweisskizze (Forts.)


- Falls  $M(x)$  anhält, gibt es  $\hat{K}_0, \dots, \hat{K}_t, C_0, \dots, C_c$ , so dass für alle  $i$  gilt:
  - $\hat{K}_0$  kodiert  $K_0(x) = (s, (x, 0))$ ,
  - $K_i \vdash_M K_{i+1}$ ,
  - $C_0 = \hat{K}_t$ ,
  - $C_{i+1}$  entsteht aus  $C_i$  durch Löschen eines Nachbarzeichens von ja, nein,  $h$ , und
  - $C_c = \text{ja (oder nein oder } h)$

- Lösungswort:  
 $\# \hat{K}_0 \# \hat{K}_1 \# \dots \hat{K}_t \# C_1 \# \dots$   
 $\dots C_{c-1} \# C_c \# \#$

## Beweisskizze (Forts.)

- Umgekehrt gibt es eine Lösung (mit  $i_1 = 1$ ) nur, falls  $M(x)$  anhält
- Denn:
  - Die Regeln der Typen (2), (3) und (4) erzwingen, dass der Anfang des Lösungsstrings das Anfangsstück einer Berechnung von  $M(x)$  kodiert
  - Die Regeln (1)-(4) bewirken, dass der  $v$ -String zunächst immer länger als der  $u$ -String ist
  - Ein Längenausgleich zwischen dem  $v$ -String und dem  $u$ -String ist nur durch Anwendung der Regeln aus (5) und (6) möglich
  - Diese können jedoch nur angewendet werden, wenn die Berechnungsfolge eine Konfiguration mit einem Endzustand erreicht
- ➔  $M(x)$  terminiert

- Insgesamt haben wir also:  $\text{TM-HALT} \leq \text{SPCP}$

 Der vollständige, formale Beweis, dass  $f$  eine Reduktion ist, ist natürlich etwas komplizierter

# SPCP $\leq$ PCP: Beispiel

## Beispiel

- Die PCP-Eingabe 

|    |
|----|
| 0  |
| 01 |

, 

|    |
|----|
| 1  |
| 11 |

, 

|    |
|----|
| 10 |
| 0  |
- $z$  hat die PCP-Lösungsstrings
  - |    |
|----|
| 0  |
| 01 |

|    |
|----|
| 10 |
| 0  |

 mit Indexfolge 1, 3 und
  - |    |
|----|
| 1  |
| 11 |

|    |
|----|
| 10 |
| 0  |

 mit Indexfolge 2, 3

- Wir bilden diese PCP-Eingabe ab auf

$$f(z) = \begin{array}{|c|c|c|c|c|} \hline @0@ & 0@ & 1@ & 1@0@ & \$ \\ \hline @0@1 & @0@1 & @1@1 & @0 & @$ \\ \hline \end{array}$$

- Die Indexfolge 2, 3 führt dann nicht mehr zu einer Lösung
- Die Indexfolge 1, 3 führt jetzt zur Lösung 1, 4, 5 mit dem Lösungsstring

$$\begin{array}{|c|c|c|} \hline @0@ & 1@0@ & \$ \\ \hline @0@1 & @0 & @$ \\ \hline \end{array}$$

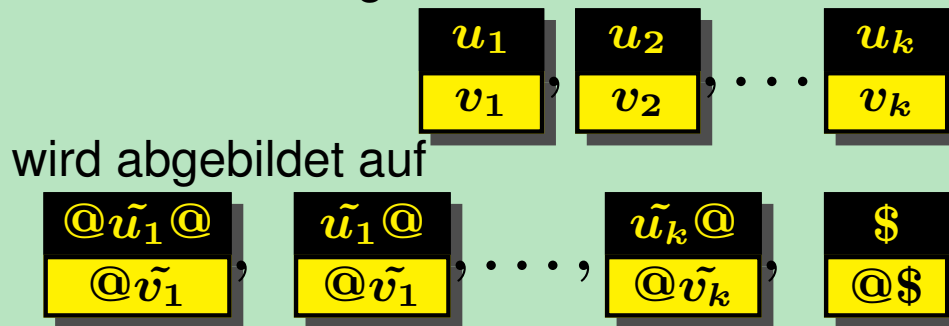
- Allgemein gilt:  $f(z)$  hat genau dann eine Lösung, wenn  $z$  eine Lösung mit  $i_1 = 1$  hat



# PCP ist unentscheidbar (7/7)

## Beweisskizze (Forts.)

- Es bleibt zu zeigen:  $\text{SPCP} \leq \text{PCP}$
- Wir nehmen dazu an, dass die Zeichen \$ und @ nicht in den Eingaben für SPCP vorkommen
- Für jeden String  $w = a_1 \cdot \dots \cdot a_m$ , mit  $a_i \in \Sigma$  sei  $\tilde{w}$  der String  $a_1 @ a_2 @ \dots @ a_m$
- Wir definieren die Reduktion  $f$  wie folgt:
  - Eine SPCP-Eingabe



## Beweisskizze (Forts.)

- Zu zeigen:  $f$  ist total und berechenbar und es gilt für alle SPCP-Eingaben  $z$ :
  - $z$  hat genau dann eine Lösung mit  $i_1 = 1$ , wenn  $f(z)$  überhaupt eine Lösung hat
- Denn:
- Sei  $(1, i_2, \dots, i_n)$  eine Lösung für  $z$
- ➔  $(1, i_2 + 1, \dots, i_n + 1, k + 2)$  ist eine Lösung für  $f(z)$
- Sei umgekehrt  $(i_1, \dots, i_n)$  eine Lösung minimaler Länge für  $f(z)$
- ➔
  - $i_1 = 1$
  - $i_n = k + 2$  sowie
  - $i_j \in \{2, \dots, k + 1\}$ , für  $j \in \{2, \dots, n - 1\}$
- ➔  $(1, i_2 - 1, \dots, i_{n-1} - 1)$  ist Lösung für  $z$

# Inhalt

15.1 Der Satz von Rice

15.2 Das Postsche Korrespondenzproblem

▷ **15.3 Unentscheidbare Grammatikprobleme**

# Unentscheidbare Grammatik-Probleme (1/3)

## Satz 15.3

- CFG-SCHNITT ist unentscheidbar

## Beweis

- Dies folgt sofort aus der Unentscheidbarkeit von PCP und  $\text{PCP} \leq \text{CFG-SCHNITT}$   
☞ Satz 14.2, Lemma 14.3

## Definition: CFG-UNAMB

**Gegeben:** Kontextfreie Grammatik  $G$   
**Frage:** Ist  $G$  eindeutig?

- Eindeutige Grammatiken heißen auf Englisch *unambiguous*

## Definition: CFG-EQUI

**Gegeben:** Kontextfreie Grammatiken  $G_1, G_2$   
**Frage:** Ist  $L(G_1) = L(G_2)$ ?

## Definition: CFG-REG-EQUI

**Gegeben:** Kontextfreie Grammatik  $G$ , RE  $\alpha$   
**Frage:** Ist  $L(G) = L(\alpha)$ ?

## Definition: CFG-ALL

**Gegeben:** Kontextfreie Grammatik  $G$   
**Frage:** Ist  $L(G) = \Sigma^*$ ?

## Definition: CFG-CONT

**Gegeben:** Kontextfreie Grammatiken  $G_1, G_2$   
**Frage:** Ist  $L(G_1) \subseteq L(G_2)$ ?


## Satz 15.4

- Die folgenden Entscheidungsprobleme sind unentscheidbar:
  - CFG-UNAMB
  - CFG-EQUI
  - CFG-REG-EQUI
  - CFG-ALL
  - CFG-CONT

# Weitere unentscheidbare Grammatik-Probleme (2/3)

## Beweisskizze: CFG-UNAMB

- Beweis durch Reduktion:  

$$\text{PCP} \leq \overline{\text{CFG-UNAMB}}$$
- Sei  $z = (u_1, v_1), \dots, (u_k, v_k)$  eine Eingabe für PCP
- Wir definieren wieder:
  - $G_1: S_1 \rightarrow u_1 S_1 1 \mid \dots \mid u_k S_1 k \mid u_1 \$1 \mid \dots \mid u_k \$k$
  - $G_2: S_2 \rightarrow v_1 S_2 1 \mid \dots \mid v_k S_2 k \mid v_1 \$1 \mid \dots \mid v_k \$k$
- Klar:  $G_1$  und  $G_2$  sind jeweils eindeutig
  - Die Zahlenfolge am Ende des Strings bestimmt eindeutig den Ableitungsbaum  sogar die Ableitung
- Sei  $G$  nun die durch Vereinigung der Regeln aus  $G_1$  und  $G_2$  entstehende Grammatik, ergänzt um  $S \rightarrow S_1 \mid S_2$

## Beweisskizze (Forts.)

- Es gilt:
  - Ist  $\vec{i}$  eine Lösung für  $z$  mit Lösungswort  $w$ , so hat der String  $w\$ \overleftarrow{i}$  zwei Ableitungsbäume
    - \* Dabei steht  $\overleftarrow{i}$  für die umgekehrte Konkatenation der Zahlen aus  $\vec{i}$
  - Also:  $z$  hat Lösung  $\Rightarrow G$  ist nicht eindeutig
- Andererseits:
  - Hat ein Wort  $w\$ \overleftarrow{i}$  zwei Ableitungsbäume, so verwendet einer  $S_1$ , der andere  $S_2$ , also:
 
$$w\$ \overleftarrow{i} \in L(G_1) \cap L(G_2)$$
  - ➔  $\vec{i}$  ist Lösung von  $z$ 
    - Also:  $G$  ist nicht eindeutig  $\Rightarrow z$  hat Lösung
- Insgesamt:
 
$$z \in \text{PCP} \iff G \text{ nicht eindeutig}$$

# Weitere unentscheidbare Grammatik-Probleme (3/3)

## Beweisskizze (Forts.)

- Die Unentscheidbarkeit von CFGALL folgt aus dem Beweis der Unentscheidbarkeit von CFG-SCHNITT
- Zu den Grammatiken  $G_1, G_2$  aus diesem Beweis lassen sich kontextfreie Grammatiken  $H_1, H_2$  für die Komplemente von  $L(G_1)$  und  $L(G_2)$  finden
  - Denn:  $L(G_1)^R$  und  $L(G_2)^R$  sind deterministisch kontextfrei
  - ➡ Die Komplemente von  $L(G_1)$  und  $L(G_2)$  sind kontextfrei!
- ➡  $L(G_1) \cap L(G_2) = \emptyset \iff L(H_1) \cup L(H_2) = \Sigma^*$
- Sei nun  $H$  die durch Vereinigung der Regeln von  $H_1$  und  $H_2$  und Hinzufügung einer neuen Startvariablen  $S$  und zusätzlichen Regeln  $S \rightarrow S_1 \mid S_2$  entstehende Grammatik
- ➡  $L(G_1) \cap L(G_2) = \emptyset \iff L(H) = \Sigma^*$
- Wir erhalten insgesamt:  $\text{PCP} \leq \overline{\text{CFGALL}}$
- Daraus folgt die Unentscheidbarkeit von CFGALL
- Die Unentscheidbarkeit der anderen Probleme folgt leicht durch Reduktion von  $\overline{\text{CFGALL}}$  und damit auch von CFGALL

# Zusammenfassung

- Der Satz von Rice sagt aus, dass semantische Aussagen über Programme nicht algorithmisch getestet werden können
- Das **Postsche Korrespondenzproblem** ist unentscheidbar
- Aus diesem Resultat lässt sich die Unentscheidbarkeit einiger Probleme nachweisen, die mit kontextfreien Sprachen zu tun haben

# Literatur

- PCP:
  - Emil L. Post. A variant of a recursively unsolvable problem.  
*Bulletin of the American Mathematical Society*, 52:264–269,  
1946

# Grundbegriffe der Theoretischen Informatik

Sommersemester 2016 - Thomas Schwentick

Teil C: Berechenbarkeit und Entscheidbarkeit

16: Varianten, Einschränkungen und Erweiterungen

Version von: 21. Juni 2016 (12:10)



## And the winner is...

- In 2013 fand in Großbritannien eine Online-Umfrage statt unter dem Motto **Great British Innovations**
- Es sollte die wichtigste Britische Erfindung der vergangenen 100 Jahre gewählt werden:
  - <http://www.topbritishinnovations.org>
- Zur Wahl standen unter anderem:
  - 3D-Displays ohne Glas
  - das WWW
  - Flüssigkristall (-Anzeigen)
  - Die Doppelhelix
  - DNA Sequenzierung
  - der Mini Cooper
- Gewonnen hat mit 24% der Stimmen:  
**die universelle Turingmaschine**
- Was das ist, schauen wir uns jetzt an

# Inhalt


## ▷ 16.1 Universelle Turingmaschinen

16.2 Semi-Entscheidbarkeit

16.3  $\mu$ -rekursive und primitiv rekursive Funktionen

16.4 Weitere unentscheidbare Probleme

# Universelle Turingmaschinen: Vorüberlegungen

- Ein wichtiges Merkmal der von Neumann-Architektur von Rechnern ist die prinzipielle Gleichbehandlung von Programmen und Daten:
    - Programme sind nicht „fest verdrahtet“
    - Vielmehr kann der selbe Rechner viele verschiedene Programme ausführen, die im Speicher wie „normale“ Daten repräsentiert werden
  - Wir werden jetzt sehen, dass sich dieses Prinzip auch bei Turingmaschinen anwenden lässt
    - Wir konstruieren dazu jetzt eine feste Turingmaschine  $U$ , die als Eingabe eine (Kodierung einer) Turingmaschine  $M$  und einen String  $x$  erhält und dann  $f_M(x)$  berechnet
      - \*  $U$  wird als „Interpretierer“ arbeiten
  - Genau genommen hat von Neumann dieses Prinzip von Turingmaschinen übernommen
-  Im Folgenden betrachten wir Turingmaschinen ausschließlich über dem Ein-/Ausgabealphabet  $\Sigma = \{0, 1\}$
- Die Resultate gelten aber entsprechend auch für jedes andere feste Alphabet
  - Wir gehen außerdem im Folgenden davon aus, dass die Menge der Zustände und das Arbeitsalphabet einer TM geordnet sind

# Kodierung von Turingmaschinen (1/2)

- Bei der Konstruktion einer universellen Turingmaschine  $U$  ergibt sich eine Komplikation:
  - $U$  wird ein festes Alphabet und eine feste Zustandsmenge haben
  - Die Turingmaschinen, die  $U$  als Eingabe bekommt, können aber verschiedene und beliebig große Arbeitsalphabete und Zustandsmengen haben
- ➔ Damit  $U$  beliebige TMs verarbeiten kann, müssen diese **kodiert** werden

- Wir kodieren Turingmaschinen über dem Alphabet  $\Sigma = \{0, 1\}$ :
  - Für eine gegebene TM  $M$  sei zunächst num eine Funktion 16.1, die jedem Zustand, jedem Zeichen und jeder Richtung eine Nummer zuordnet:
    - \*  $\text{num} : Q \rightarrow \mathbb{N}$
    - \*  $\text{num} : \Gamma \rightarrow \mathbb{N}$
    - \*  $\text{num} : \{\leftarrow, \downarrow, \rightarrow\} \rightarrow \mathbb{N}$

- Dabei soll immer gelten:

| $x$           | $\text{num}(x)$ |
|---------------|-----------------|
| $\leftarrow$  | 1               |
| $\downarrow$  | 2               |
| $\rightarrow$ | 3               |

| $x$              | $\text{num}(x)$ |
|------------------|-----------------|
| $\triangleright$ | 1               |
| $\sqcup$         | 2               |
| 0                | 3               |
| 1                | 4               |

| $x$  | $\text{num}(x)$ |
|------|-----------------|
| $s$  | 1               |
| ja   | 2               |
| nein | 3               |
| $h$  | 4               |

- Wir kodieren dann Zustände, Zeichen und Richtungen durch 0-1-Strings gemäß:

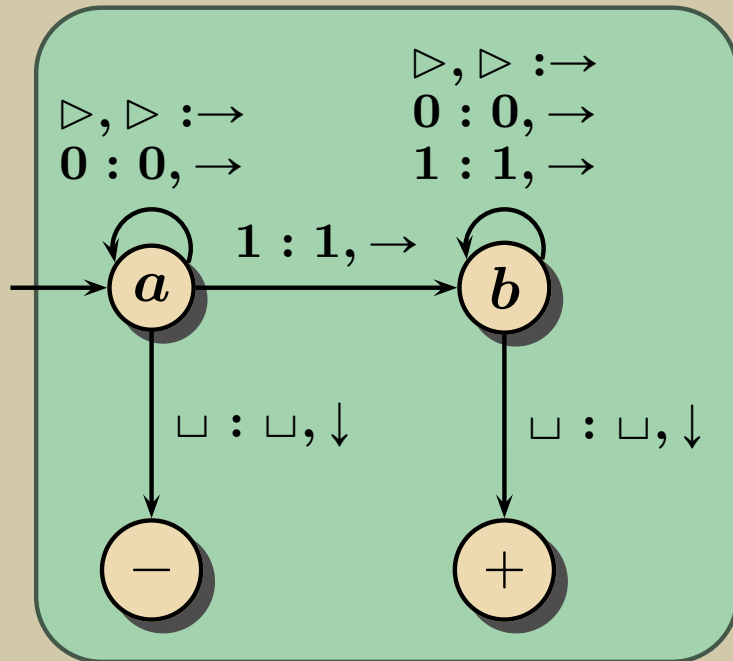
- $\underline{\text{enc}(q)} \stackrel{\text{def}}{=} 0^{\text{num}(q)} 1$
- $\underline{\text{enc}(\sigma)} \stackrel{\text{def}}{=} 0^{\text{num}(\sigma)} 1$
- $\underline{\text{enc}(d)} \stackrel{\text{def}}{=} 0^{\text{num}(d)} 1$

- $\delta(q, \sigma) = (q', \sigma', d)$  kodieren wir durch den String  $\underline{\text{enc}(q, \sigma)} \stackrel{\text{def}}{=} 1 \text{enc}(q) \text{enc}(\sigma) \text{enc}(q') \text{enc}(\sigma') \text{enc}(d)$

- Die Kodierung einer TM  $M$ :
  - $\underline{\text{enc}(M)} \stackrel{\text{def}}{=} \text{Konkatenation der Strings } \text{enc}(q, \sigma) \text{ in lexikographischer Ordnung nach } \text{num}(q), \text{num}(\sigma)$

# Kodierung von Turingmaschinen: Beispiel

## Beispiel-Turingmaschine



- Zur Erinnerung:

| $x$           | $\text{num}(x)$ | $x$              | $\text{num}(x)$ | $x$  | $\text{num}(x)$ |
|---------------|-----------------|------------------|-----------------|------|-----------------|
| $\leftarrow$  | 1               | $\triangleright$ | 1               | $s$  | 1               |
| $\downarrow$  | 2               | $\sqcup$         | 2               | ja   | 2               |
| $\rightarrow$ | 3               | 0                | 3               | nein | 3               |
|               |                 | 1                | 4               | $h$  | 4               |

## Beispiel-TM als String

- Da  $s = a$  muss nur noch die Kodierung für  $b$  ergänzt werden:

| $x$ | $\text{num}(x)$ |
|-----|-----------------|
| $a$ | 1               |
| $b$ | 5               |




- Also:

| $q, \sigma$         | $\text{enc}(q, \sigma)$     |
|---------------------|-----------------------------|
| $a, \triangleright$ | 1010101010001               |
| $a, \sqcup$         | 1010010001001001            |
| $a, 0$              | 10100010100010001           |
| $a, 1$              | 10100001000001000010001     |
| $b, \triangleright$ | 10000101000001010001        |
| $b, \sqcup$         | 1000001001001001001         |
| $b, 0$              | 1000001000100000100010001   |
| $b, 1$              | 100000100001000001000010001 |

- Die Kodierung der TM ist dann:

10101010100011010010001001001101  
 00010100010001101000010000010000  
 10001100001010000010100011000001  
 00100100100110000010001000001000  
 10001100000100001000001000010001

## Kodierung von Turingmaschinen (2/2)

- Strings, die Turingmaschinen kodieren, müssen (bisher) eine spezielle Form haben
  - Zum Beispiel müssen sie mindestens von der Form  $(10^+ 10^+ 10^+ 10^+ 10^+ 1)^*$  sein
- Wir würden aber gerne *jedem* 0-1-String  $w$  eine TM  $M_w$  zuordnen
- Deshalb definieren wir:
  - $\underline{M_w} \stackrel{\text{def}}{=}$ 
    - \* die TM  $M$  mit  $\text{enc}(M) = w$ , falls eine solche TM  $M$  existiert
    - \* die TM  $M_-$ , die bei Lesen des linken Rand-symbols  $\triangleright$  sofort in den ablehnenden Zustand übergeht, andernfalls
-  Ob es zu einem String  $w$  eine TM  $M$  mit  $\text{enc}(M) = w$  gibt, lässt sich von einer TM überprüfen
-   $M$  ist eindeutig  bis auf die Namen der Zustände
  - Die Details ersparen wir uns

# Existenz einer universellen Turingmaschine

## Satz 16.1

- Es gibt eine **universelle Turingmaschine**, d.h. eine Turingmaschine  $U$ , die für jede TM  $M$  und jeden 0-1-String  $x$  die folgenden Bedingungen erfüllt:
  - Falls  $M$  die Eingabe  $x$  akzeptiert, so akzeptiert  $U$  die Eingabe  $\text{enc}(M)\#x$
  - Falls  $M$  die Eingabe  $x$  ablehnt, so lehnt  $U$  die Eingabe  $\text{enc}(M)\#x$  ab
  - Falls  $M$  bei Eingabe  $x$  nicht terminiert, so terminiert  $U$  bei Eingabe  $\text{enc}(M)\#x$  auch nicht

## Beweisskizze

- Wir konstruieren  $U$  als 4-String TM:
  - String 1:  
Inhalt des Arbeits-Strings von  $M$
  - String 2:  $\text{enc}(M)$
  - String 3: Zustand von  $M$
  - String 4: Hilfsstring für Kopieroperationen

## Beweisskizze (Forts.)

- Bei Eingabe  $\text{enc}(M)\#x$  geht  $U$  wie folgt vor:
  - Kopiere  $\text{enc}(M)$  auf String 2
  - Ersetze  $\#$  durch  $\triangleright$  auf String 1 und ersetze  $x$  durch
$$\text{enc}(x) \stackrel{\text{def}}{=} \text{enc}(x_1) \cdot \dots \cdot \text{enc}(x_{|x|})$$
  - Schreibe 01 auf String 3
  - Simuliere  $M$  Schritt für Schritt mit Hilfe von  $\text{enc}(M)$  auf String 2

☞ Startzustand

## Bemerkungen

- $U$  kann auch überprüfen, ob die Eingabe überhaupt von der Form  $\text{enc}(M)\#x$  ist und ablehnen, falls dies nicht der Fall ist
- Die Konstruktion kann auch für Turingmaschinen, die Funktionen berechnen, angepasst werden
- Wie in Satz 13.3 kann  $U$  auch als 1-String-TM konstruiert werden

# Inhalt

16.1 Universelle Turingmaschinen

▷ **16.2 Semi-Entscheidbarkeit**

16.3  $\mu$ -rekursive und primitiv rekursive Funktionen

16.4 Weitere unentscheidbare Probleme



# Semientscheidbare Sprachen

## Definition

- Eine Menge  $L \subseteq \Sigma^*$  heißt semi-entscheidbar, falls es eine TM  $M$  mit  $L = L(M)$  gibt
- **Zu beachten:**
  - Bei einer semi-entscheidbaren Menge ist erlaubt, dass die TM für manche Eingaben  $x \notin L$  nicht terminiert
- Klar: jede entscheidbare Sprache ist auch semi-entscheidbar
- Ein „Semi-Entscheidungsalgorithmus“ für  $A$  ist ein Algorithmus, der
  - für alle „Ja-Eingaben“ anhält und akzeptiert und
  - für alle anderen Eingaben ablehnt oder nicht anhält

## Beispiel

- CFG-SCHNITT ist semi-entscheidbar
  - Ein Algorithmus für CFG-SCHNITT kann alle Strings  $w$  über dem Terminalalphabet der Länge nach erzeugen und mit dem CYK-Algorithmus jeweils testen, ob  $w \in L(G_1)$  und  $w \in L(G_2)$ 
    - \* Wenn er ein solches  $w$  findet, akzeptiert er
    - \* Wenn er kein solches  $w$  findet, terminiert er nicht
- Weitere semientscheidbare Probleme:
  - TM-DIAG, TM-HALT, PCP,...
- Dass TM-HALT semientscheidbar aber nicht entscheidbar ist lässt sich wie folgt interpretieren:
  - es gibt keine einfachere, allgemeine Methode, etwas über das Verhalten einer TM herauszufinden als sie zu simulieren

# Entscheidbar vs. Semi-entscheidbar

- Die Begriffe „entscheidbar“ und „semi-entscheidbar“ sind eng miteinander verknüpft, wie das folgende Lemma zeigt

## Lemma 16.2

- Sei  $L \subseteq \Sigma^*$  eine Sprache
- Dann gilt:  
 $L$  entscheidbar  $\iff$   
 $L$  und  $\bar{L}$  semi-entscheidbar
- Dabei bezeichnet  $\bar{L}$  wieder das Komplement  $\Sigma^* - L$  von  $L$

## Beweisidee

- „ $\Rightarrow$ “:
  - Sei  $M$  eine TM, die  $L$  entscheidet
  - ➔  $L = L(M)$ , also ist  $L$  semi-entscheidbar
  - Vertauschen von ja und nein in  $M$  ergibt eine immer terminierende TM  $M'$  mit  $L(M') = \bar{L}$
  - ➔  $\bar{L}$  ist auch semi-entscheidbar
- „ $\Leftarrow$ “:
  - Seien  $M_1, M_2$  Turingmaschinen mit  $L = L(M_1)$  und  $\bar{L} = L(M_2)$
  - Sei  $M$  die TM, die, bei Eingabe  $w$ , beide Turing-Maschinen mit Eingabe  $w$  simultan simuliert
    - \* Falls  $M_1$  akzeptieren würde, so akzeptiert  $M$
    - \* Falls  $M_2$  akzeptieren würde, so lehnt  $M$  ab
  - Da einer der beiden Fälle irgendwann zutreffen muss, terminiert  $M$  nach endlich vielen Schritten
- Zu beachten: es ist wichtig, dass die beiden Simulationen **simultan** und nicht nacheinander stattfinden

# Nicht semi-entscheidbare Probleme

- Lemma 16.2 liefert eine einfache Methode zum Nachweis, dass eine Sprache nicht semi-entscheidbar ist

## Lemma 16.3

- Das Komplement von TM-HALT ist nicht semi-entscheidbar

## Beweis

- TM-HALT ist semi-entscheidbar
  - Wäre das Komplement von TM-HALT auch semi-entscheidbar, so wäre gemäß Lemma 16.2 TM-HALT entscheidbar
- ➡ Widerspruch

# Semi-entscheidbar vs. rekursiv aufzählbar (1/2)

- Die semi-entscheidbaren Sprachen werden häufig auch „rekursiv aufzählbar“ genannt
- Warum dies so ist, betrachten wir als nächstes

## Definition

- Eine Sprache  $L \subseteq \Sigma^*$ , heißt **rekursiv aufzählbar**, falls es eine (2-String-) TM  $M$  gibt, die nach und nach alle Elemente von  $L$  auf ihren zweiten String schreibt
  - Die Strings aus  $L$  werden dabei durch  $\#$  getrennt
  - Der Zeiger des zweiten Strings bewegt sich niemals nach links
- Wir sagen, dass  $M$  die Strings aus  $L$  nach und nach „ausgibt“
- Zu beachten:  $M$  hält bei unendlichen Sprachen  $L$  nicht an

# Semi-entscheidbar vs. rekursiv aufzählbar (2/2)

## Lemma 16.4

- Sei  $L$  eine Sprache
- Dann gilt:  
 $L$  rekursiv aufzählbar  $\iff$   
 $L$  semi-entscheidbar

## Beweisskizze

- „ $\Rightarrow$ “:
  - Sei  $L$  rekursiv aufzählbar
  - Sei  $M$  eine 2-String-TM, die die Strings aus  $L$  nach und nach auf ihrem zweiten String erzeugt
  - Eine TM  $M'$ , die  $L$  „semi-entscheidet“ kann wie folgt konstruiert werden:
    - \* Bei Eingabe  $x$  simuliert  $M'$  die TM  $M$
    - \* Falls diese den String  $x$  auf dem zweiten String ausgeben würde, akzeptiert  $M'$
    - \* Falls dies nicht passiert, hält  $M'$  nicht an
- „ $\Leftarrow$ “:
  - Sei  $M$  eine TM mit  $L(M) = L$
  - Eine „Aufzählungs-TM“  $M'$  für  $L$  kann wie folgt vorgehen:
    - \* Für jedes  $n > 0$  simuliert sie für alle Strings  $w$  der Länge  $\leq n$  die TM  $M$  bei Eingabe  $w$  für  $n$  **Schritte**
    - \* Falls  $M(w)$  bei einer solchen Simulation akzeptieren würde, gibt sie  $w$  auf String 2 aus

## Entscheidbar vs. Berechenbar (1/2)

- Der Zusammenhang zwischen berechenbaren Funktionen und entscheidbaren Mengen lässt sich mit Hilfe des Begriffs der **charakteristischen Funktionen** präzise formulieren

### Definition

- Sei  $L \subseteq \Sigma^*$  eine Sprache
- Die **charakteristische Funktion** von  $L$ ,  $\chi_L: \Sigma^* \rightarrow \{0, 1\}$ , ist definiert durch:
$$\chi_L(w) \stackrel{\text{def}}{=} \begin{cases} 1 & w \in L \\ 0 & w \notin L \end{cases}$$
- Die **partielle charakteristische Funktion** von  $L$ ,  $\chi'_L: \Sigma^* \rightarrow \{0, 1\}$ , ist definiert durch:
$$\chi'_L(w) \stackrel{\text{def}}{=} \begin{cases} 1 & w \in L \\ \perp & w \notin L \end{cases}$$

## Entscheidbar vs. Berechenbar (2/2)

- Aus Sicht der zur Berechnung verwendeten Turingmaschine ist der Unterschied zwischen dem Entscheiden einer Sprache  $L$  und der Berechnung ihrer charakteristischen Funktion  $\chi_L$  rein formal:
  - Beim Entscheiden von  $L$  geht sie für Strings  $w \in L$  am Ende in den Zustand ja
  - Beim Berechnen von  $\chi_L$  gibt sie für Strings  $w \in L$  den Wert 1 aus
- Das ist also im Grunde nur eine Frage des *User Interfaces*

### Lemma 16.5

- Sei  $L \subseteq \Sigma^*$  eine Sprache
- Dann gelten:
  - (a)  $L$  entscheidbar  $\iff \chi_L$  berechenbar
  - (b)  $L$  semi-entscheidbar  $\iff \chi'_L$  berechenbar

### Beweis

- Wir betrachten den Beweis von (a) „ $\Rightarrow$ “
  - Die anderen Beweise sind analog
- Sei  $L$  entscheidbar
- ➡ Dann gibt es eine TM  $M$ , die  $L$  entscheidet
- Aus  $M$  lässt sich eine TM  $M'$  konstruieren, die, wenn  $M$  in den Zustand „ja“ gehen würde, im letzten String  $1\square$  neben  $\triangleright$  schreibt, den Zeiger an den linken Rand bewegt und anhält
  - (und analog für Zustand „nein“ mit  $0\square$ )
- ➡  $f_{M'} = \chi_L$
- ➡  $\chi_L$  berechenbar

# Inhalt

16.1 Universelle Turingmaschinen

16.2 Semi-Entscheidbarkeit

▷ **16.3  $\mu$ -rekursive und primitiv rekursive Funktionen**

16.4 Weitere unentscheidbare Probleme



# Rekursive Funktionen: Vorüberlegungen

## Beispiel

- Die Addition natürlicher Zahlen lässt sich induktiv mit Hilfe der „+1“-Funktion definieren:

$$- m + 0 \stackrel{\text{def}}{=} m$$


$$- m + (n + 1) \stackrel{\text{def}}{=} (m + n) + 1$$

- Analog lässt sich die Multiplikation induktiv mit Hilfe der Addition definieren:

$$- m \times 0 \stackrel{\text{def}}{=} 0$$

$$- m \times (n + 1) \stackrel{\text{def}}{=} (m \times n) + m$$

- Wir werden jetzt genauer untersuchen, welche Funktionen sich mit solchen induktiven Definitionen beschreiben lassen

 Wir verwenden für arithmetische Operationen Funktionsnotation anstelle der Infixnotation

- Mit  $s : \mathbb{N}_0 \rightarrow \mathbb{N}_0$  bezeichnen wir die Funktion  $n \mapsto n + 1$

## Beispiel

- Formal definieren wir die 2-stellige Funktion  $\text{add} : \mathbb{N}_0^2 \rightarrow \mathbb{N}_0$  induktiv wie folgt:

$$- \text{add}(0, m) \stackrel{\text{def}}{=} m, \text{ für alle } m \in \mathbb{N}_0$$

$$- \text{add}(n + 1, m) \stackrel{\text{def}}{=} s(\text{add}(n, m)), \text{ für alle } n, m \in \mathbb{N}_0$$

- Analog definieren wir  $\text{mult} : \mathbb{N}_0^2 \rightarrow \mathbb{N}_0$  durch:

$$- \text{mult}(0, m) \stackrel{\text{def}}{=} 0$$

$$- \text{mult}(n + 1, m) \stackrel{\text{def}}{=} \text{add}(\text{mult}(n, m), m)$$




- Wir verwenden hier also
  - gewisse Basisfunktionen ( $s, 0$ ),
  - Komposition von Funktionen, und
  - rekursive Definitionen

- Dies führt uns zur Definition der **primitiv rekursiven Funktionen**

# Primitiv rekursive Funktionen: Definition (1/2)

- Die primitiv rekursiven Funktionen sind induktiv wie folgt definiert

## Definition: Basisfunktionen

- Alle Funktionen  $c^k : \mathbb{N}_0^k \rightarrow \mathbb{N}_0$  mit  $c \in \mathbb{N}_0$ , definiert durch
  - $c^k(x_1, \dots, x_k) \stackrel{\text{def}}{=} c$sind primitiv rekursiv  konstante Funktionen
- Alle Funktionen  $\pi_i^k : \mathbb{N}_0^k \rightarrow \mathbb{N}_0$ , definiert durch
  - $\pi_i^k(x_1, \dots, x_k) \stackrel{\text{def}}{=} x_i$sind primitiv rekursiv  Projektionen
- Die Funktion  $s : \mathbb{N}_0 \rightarrow \mathbb{N}_0$ , definiert durch
  - $s(x_1) \stackrel{\text{def}}{=} x_1 + 1$ist primitiv rekursiv  Nachfolgerfunktion

## Primitiv rekursive Funktionen: Definition (2/2)

### Definition (Forts.): Zusammengesetzte Funktionen

- Sind die Funktionen

- $h : \mathbb{N}_0^\ell \rightarrow \mathbb{N}_0$  und

- $g_1, \dots, g_\ell : \mathbb{N}_0^k \rightarrow \mathbb{N}_0$

primitiv rekursiv, so auch die durch

- $f(x_1, \dots, x_k) \stackrel{\text{def}}{=} h(g_1(x_1, \dots, x_k), \dots, g_\ell(x_1, \dots, x_k))$

definierte Funktion  $f : \mathbb{N}_0^k \rightarrow \mathbb{N}_0$

 Komposition

- Sind die Funktionen

- $g : \mathbb{N}_0^{k-1} \rightarrow \mathbb{N}_0$  und

- $h : \mathbb{N}_0^{k+1} \rightarrow \mathbb{N}_0$

primitiv rekursiv, so auch die durch

- $f(0, x_2, \dots, x_k) \stackrel{\text{def}}{=} g(x_2, \dots, x_k)$

- $f(x+1, x_2, \dots, x_k) \stackrel{\text{def}}{=} h(f(x, x_2, \dots, x_k), x, x_2, \dots, x_k)$

definierte Funktion  $f : \mathbb{N}_0^k \rightarrow \mathbb{N}_0$

 Primitive Rekursion

# Primitiv rekursive Funktionen: Beispiele (1/2)

- Wenn wir die Definition der primitiv rekursiven Funktionen ganz genau befolgen, erhalten wir sehr schlecht lesbare Funktionsdefinitionen
- Deshalb verwenden wir eine leicht abkürzende Notation für die Konstanten und Projektionen und erlauben Variablen wie  $x, y, z$

## Beispiel: Addition

- $\text{add}(0, x_2) \stackrel{\text{def}}{=} \pi_1^1(x_2)$ 
  - Abkürzende Notation:  $\text{add}(0, y) \stackrel{\text{def}}{=} y$
- $\text{add}(x + 1, x_2) \stackrel{\text{def}}{=} s(\pi_1^3(\text{add}(x, x_2), x, x_2))$ 
  - 🖋 Streng genommen müsste die Komposition  $s \circ \pi_1^3$  sogar zuerst als neue Funktion definiert werden...
  - Abkürzende Notation:  
 $\text{add}(x + 1, y) \stackrel{\text{def}}{=} s(\text{add}(x, y))$

## Beispiel: Multiplikation

- Multiplikation in abkürzender Notation:
  - Abkürzende Notation:  $\text{mult}(0, y) \stackrel{\text{def}}{=} 0$
  - $\text{mult}(x + 1, y) \stackrel{\text{def}}{=} \text{add}(\text{mult}(x, y), y)$

## Beispiel: Signum

- Sei  $\sigma : \mathbb{N}_0 \rightarrow \mathbb{N}_0$  die „Sigma-Funktion“ mit der „üblichen Definition“

$$\sigma(x) \stackrel{\text{def}}{=} \begin{cases} 1, & \text{falls } x > 0 \\ 0, & \text{falls } x = 0 \end{cases}$$

- $\sigma$  ist primitiv rekursiv:
  - $\sigma(0) \stackrel{\text{def}}{=} 0$
  - $\sigma(x + 1) \stackrel{\text{def}}{=} 1$

- Sei  $\tau : \mathbb{N}_0 \rightarrow \mathbb{N}_0$  definiert durch
  - $\tau(0) \stackrel{\text{def}}{=} 1$
  - $\tau(x + 1) \stackrel{\text{def}}{=} 0$die Funktion  $1 - \sigma$

## Primitiv rekursive Funktionen: Beispiele (2/2)

- Wir definieren einen „kleinste Nullstelle“-Operator mit „beschränktem Suchraum“

### Beispiel

- Sei  $f(x, y) = (x^2 \dot{-} y) + (y \dot{-} x^2)$
- Die kleinste Nullstelle von  $f$  bezüglich  $y = 25$  ist 5
- Allgemein ist die „kleinste Nullstelle“ von  $f$  bezüglich eines festen Wertes  $b$  für  $y$  die kleinste Zahl  $a \in \mathbb{N}_0$  mit  $f(a, b) = 0$
- Für eine Funktion  $f : \mathbb{N}_0^k \rightarrow \mathbb{N}_0$  sei  $\text{MIN}_f : \mathbb{N}_0^k \rightarrow \mathbb{N}_0$  die Funktion:  
 $\text{MIN}_f(x, x_2, \dots, x_k) \stackrel{\text{def}}{=} \begin{cases} \min\{n \leq x \mid f(n, x_2, \dots, x_k) = 0\}, & \text{falls so ein } n \text{ existiert} \\ 0, & \text{andernfalls} \end{cases}$

- $\text{MIN}_f(x, \dots) = 0$  kann bedeuten,
  - dass es keine Nullstelle  $\leq x$  gibt, oder
  - dass die kleinste Nullstelle 0 ist

### Beispiel: beschränktes Minimum

- Ist  $f$  primitiv rekursiv, so auch  $\text{MIN}_f$ :
  - $\text{MIN}_f(0, x_2, \dots, x_k) \stackrel{\text{def}}{=} 0$
  - $\text{MIN}_f(x + 1, x_2, \dots, x_k) \stackrel{\text{def}}{=}$

$$\begin{aligned} &\text{MIN}_f(x, x_2, \dots, x_k) + \\ &\quad \left[ (x + 1) \times \right. \\ &\quad \left. \tau(\text{MIN}_f(x, x_2, \dots, x_k)) \times \right. \\ &\quad \left. \sigma(f(0, x_2, \dots, x_k)) \times \right. \\ &\quad \left. \tau(f(x + 1, x_2, \dots, x_k)) \right] \end{aligned}$$

- Erläuterung:
  - $\text{MIN}_f(x + 1, x_2, \dots, x_k) = \text{MIN}_f(x, x_2, \dots, x_k)$   
falls  $\text{MIN}_f(x, x_2, \dots, x_k) \neq 0$   
oder  $f(0, x_2, \dots, x_k) = 0$
  - Andernfalls ist es  $x + 1$ ,  
falls  $f(x + 1, x_2, \dots, x_k) = 0$
  - Sonst: 0

# $\mu$ -rekursive Funktionen

- Wir betrachten jetzt einen Operator, der „**global**“ nach der kleinsten Nullstelle sucht

## Definition

- Ist  $f : \mathbb{N}_0^{k+1} \rightarrow \mathbb{N}_0$  eine partielle Funktion, so sei die partielle Funktion  $\mu f : \mathbb{N}_0^k \rightarrow \mathbb{N}_0$ , definiert durch:
- $\mu f(x_1, \dots, x_k) \stackrel{\text{def}}{=} \begin{cases} \text{kleinstes } n \in \mathbb{N}_0, \text{ für das gilt:} \\ * f(n, x_1, \dots, x_k) = 0 \text{ und} \\ * f(m, x_1, \dots, x_k) \neq \perp, \text{ für alle } m < n \\ \perp, \text{ wenn kein solches } n \text{ existiert} \end{cases}$
- Die  **$\mu$ -rekursiven Funktionen** sind wie die primitiv rekursiven Funktion definiert mit der zusätzlichen Regel:
  - Ist  $f$   $\mu$ -rekursiv, so auch  $\mu f$

## Beispiel

- Ist  $f(x, y) = (x^2 \dot{-} y) + (y \dot{-} x^2)$ , so ist also  $\mu f(y)$  die nicht negative ganzzahlige Quadratwurzel von  $y$ , falls diese existiert

# $\mu$ -rekursiv vs. WHILE

## Satz 16.6

- Eine partielle Funktion  $f : \mathbb{N}_0^k \rightarrow \mathbb{N}_0$  ist genau dann  $\mu$ -rekursiv, wenn sie durch ein WHILE-Programm berechnet werden kann

## Beweisidee

- „WHILE  $\rightarrow \mu$ -rekursiv“: kompliziert
- „ $\mu$ -rekursiv  $\rightarrow$  WHILE“: sehr einfach
  - Sei  $f : \mathbb{N}_0^{k+1} \rightarrow \mathbb{N}_0$  WHILE-berechenbar
  - Das folgende WHILE-Programm berechnet  $\mu f$ :  
☞ Wesentlicher Induktionsschritt

```

$$\begin{aligned} x_{k+1} &:= 0; \\ x_{k+2} &:= f(0, x_1, \dots, x_k); \\ \text{WHILE } x_{k+2} \neq 0 \text{ DO} \\ &\quad x_{k+1} := x_{k+1} + 1; \\ &\quad x_{k+2} := f(x_{k+1}, x_1, \dots, x_k); \\ \text{END;} \\ x_1 &:= x_{k+1} \end{aligned}$$

```

- Dabei ist  $x_{k+2} := f(x_{k+1}, x_1, \dots, x_k)$  eine abkürzende Schreibweise für den Einschub eines WHILE-Programmes zur Berechnung von  $f$
- Zu beachten: es ist möglich, dass das Programm für  $f$  nicht terminiert
  - \* In diesem Fall ist der Wert für  $\mu f$  undefiniert

# LOOP-Programme (1/2)

- LOOP-Programme sind eine Einschränkung von WHILE-Programmen
- Ihre Syntax ergibt sich aus der Syntax von WHILE-Programmen wie folgt:

- Keine bedingte Wiederholung
- Weiteres Schlüsselwort: LOOP
- **(Unbedingte Wiederholung)**  
Falls  $P$  ein LOOP-Programm ist, so auch  
LOOP  $x_i$  DO  $P$  END


## LOOP-Programm 1

```
 $x_1 := x_2;$   
LOOP  $x_3$  DO  $x_1 := x_1 + 1$   
END
```

- Effekt:  $x_1 := x_2 + x_3$

## LOOP-Programm 2

```
 $x_2 := 1;$   
LOOP  $x_1$  DO  $x_2 := 0$  END;  
LOOP  $x_2$  DO  $P$  END
```

 Dabei ist  $P$  ein beliebiges LOOP-Programm

- Effekt: IF  $x_1 = 0$  THEN  $P$  END



## LOOP-Programme (2/2)

### Definition

- **Semantik von LOOP-Schleifen:**

Ist  $P$  von der Form

LOOP  $x_i$  DO  $P_1$  END


ist, so ist  $P(X) \stackrel{\text{def}}{=} P_1^{X(i)}(X)$

- Dabei bezeichnet  $P_1^{X(i)}$  die  $X(i)$ -malige Wiederholung von  $P_1$

- Die Funktion  $f_P$  ist dann wie bei WHILE-Programmen definiert
- $f : \mathbb{N}_0 \rightarrow \mathbb{N}_0$  heißt LOOP-berechenbar, falls  $f = f_P$  für ein LOOP-Programm  $P$

### Proposition 16.7

- $f$  LOOP-berechenbar  $\Rightarrow$   
 $f$  WHILE-berechenbar
- Die Umkehrung gilt jedoch nicht

 Wir verwenden die gleiche Konvention im Hinblick auf syntaktischen Zucker wie bei LOOP-Programmen

# LOOP-Programme vs. primitive Rekursion

- Erinnerung: LOOP-Programme können beliebig-stellige Funktionen berechnen

## Satz 16.8

- Eine Funktion  $f : \mathbb{N}_0^k \rightarrow \mathbb{N}_0$  ist genau dann primitiv rekursiv, wenn sie durch ein LOOP-Programm berechnet werden kann

## Beweisskizze

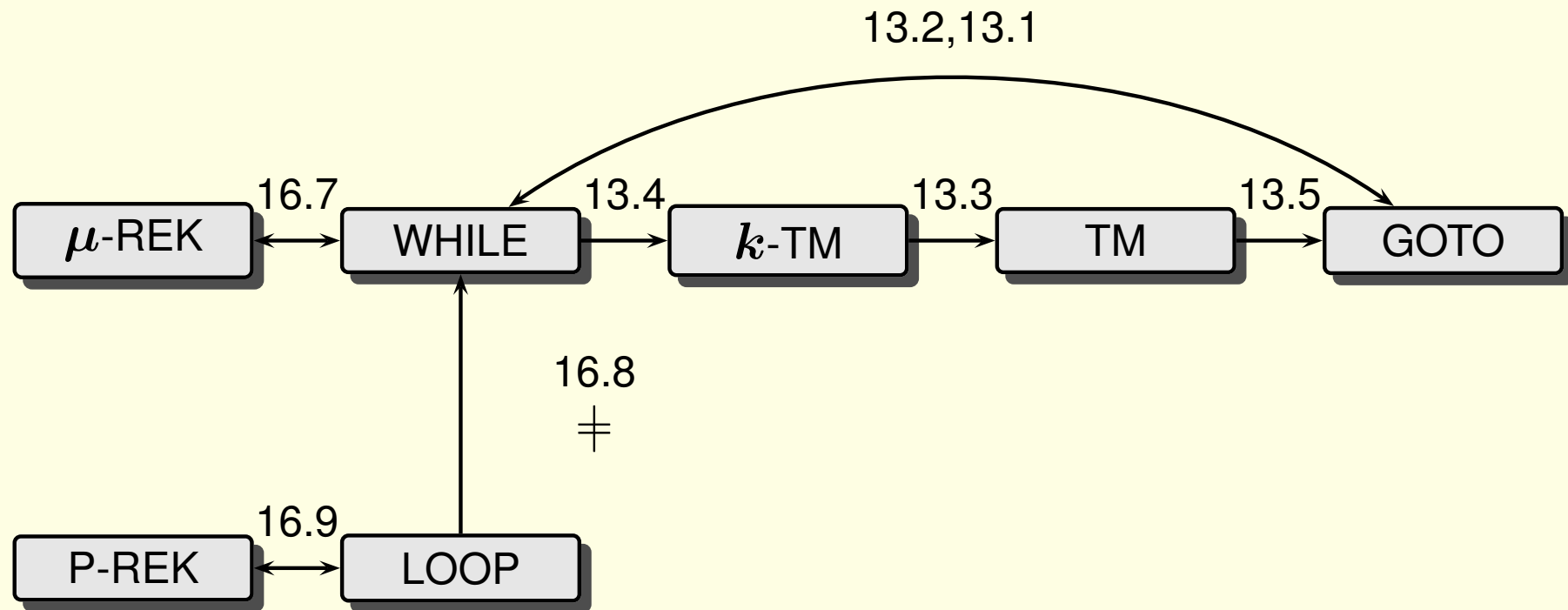
- „**LOOP**  $\rightarrow$  **primitiv rekursiv**“:  
etwas technischer Beweis, findet sich im Buch von Schöning
- „**primitiv rekursiv**  $\rightarrow$  **LOOP**“:  
– Wesentlicher Schritt: Ist  $f$  definiert durch:

$$\begin{aligned} f(0, x_2, \dots, x_k) &\stackrel{\text{def}}{=} g(x_2, \dots, x_k) \\ f(x+1, x_2, \dots, x_k) &\stackrel{\text{def}}{=} h(f(x, x_2, \dots, x_k), x, x_2, \dots, x_k), \end{aligned}$$

so ist  $f(x_1, \dots, x_k)$  berechenbar durch:

```
 $x_{k+1} := 0; x_{k+2} := g(x_2, \dots, x_k);$ 
LOOP  $x_1$  DO
     $x_{k+2} := h(x_{k+2}, x_{k+1}, x_2, \dots, x_k);$ 
     $x_{k+1} := x_{k+1} + 1$ 
END;
 $x_1 := x_{k+2}$ 
```

# Verhältnis der Berechnungsmodelle



# Charakterisierungen der semi-entscheidbaren Sprachen

- Der folgende Satz fasst verschiedene Charakterisierungen der semi-entscheidbaren Sprachen zusammen

## Satz 16.9

- Für jede Sprache  $L \subseteq \Sigma^*$  sind die folgenden Aussagen äquivalent:
  - (a)  $L$  ist semi-entscheidbar
  - (b)  $L$  ist rekursiv aufzählbar
  - (c)  $\chi'_L$  ist (Turing-, GOTO-, WHILE-) berechenbar
  - (d)  $\chi'_L$  ist  $\mu$ -rekursiv
  - (e)  $L = D(f)$  für ein berechenbares  $f$
  - (f)  $L = W(f)$  für ein berechenbares  $f$

# Inhalt

16.1 Universelle Turingmaschinen

16.2 Semi-Entscheidbarkeit

16.3  $\mu$ -rekursive und primitiv rekursive Funktionen

▷ **16.4 Weitere unentscheidbare Probleme**

## Logik und Berechenbarkeit (1/3)

- Wie in Kapitel 12 erwähnt, wurden Turingmaschinen zunächst mit dem Ziel entwickelt, zu zeigen, dass das „Entscheidungsproblem“ keine Lösung hat
- Wir betrachten jetzt kurz einige der erzielten Ergebnisse
- Für die Definition prädikatenlogischer Formeln wird auf die Logik-Vorlesung verwiesen

### Definition: FO-SAT

**Gegeben:** prädikatenlogische Formel  $\varphi$

**Frage:** Ist  $\varphi$  erfüllbar?

### Definition: FOTaut

**Gegeben:** prädikatenlogische Formel  $\varphi$

**Frage:** Ist  $\varphi$  allgemein gültig?

### Definition: FO-FINSAT

**Gegeben:** prädikatenlogische Formel  $\varphi$

**Frage:** Hat  $\varphi$  ein endliches Modell?

## Logik und Berechenbarkeit (2/3)

### Satz 16.10

- (a)  $\overline{\text{FO-SAT}}$ ,  $\text{FO-TAUT}$ , und  $\text{FO-FINSAT}$  sind semientscheidbar, aber nicht entscheidbar
- (b)  $\text{FO-SAT}$  ist nicht semientscheidbar

- Zur Erinnerung: für jede prädikatenlogische Formel  $\varphi$  gilt:
  - $\varphi$  ist Tautologie  $\iff \neg\varphi$  ist unerfüllbar
- Also gelten:
  - $\text{FO-TAUT} \leq \overline{\text{FO-SAT}}$  und
  - $\overline{\text{FO-SAT}} \leq \text{FO-TAUT}$

### Beweisidee

- Dass  $\text{FO-TAUT}$  (und damit auch  $\overline{\text{FO-SAT}}$ ) semientscheidbar ist, folgt aus dem Resolutionssatz:
  - Ist eine Formel eine Tautologie, so gibt es dafür einen (endlichen) Resolutionsbeweis, der in endlich vielen Schritten gefunden werden kann
- Die Semientscheidbarkeit von  $\text{FO-FINSAT}$  folgt, da die endlichen Strukturen systematisch aufgezählt werden können, bis ein endliches Modell gefunden ist
- Die Unentscheidbarkeit von  $\overline{\text{FO-SAT}}$ ,  $\text{FO-TAUT}$ , und  $\text{FO-FINSAT}$  ergibt sich durch Reduktion von  $\text{TM-E-HALT}$ :
  - Zu jeder TM  $M$  lässt sich eine Formel  $\varphi_M$  konstruieren, die genau dann ein Modell hat, wenn  $M(\epsilon)$  terminiert
    - \* (und das Modell enthält dann eine Kodierung dieser Berechnung)
- Dass  $\text{FO-SAT}$  nicht semientscheidbar ist, folgt mit Lemma 16.2

# Logik und Berechenbarkeit (3/3)

- Auch der automatische Beweisbarkeit von Aussagen über die Arithmetik sind enge Grenzen gesetzt:

## Definition: FO-NAT-MC

**Gegeben:** prädikatenlogische Formel  $\varphi$

**Frage:** Gilt  
 $(\mathbb{N}_0, 0, 1, +, \times) \models \varphi$ ?

## Satz 16.11

- FO-NAT-MC ist nicht semientscheidbar

## Beweisidee

- Der Beweis ist durch Reduktion von  $\overline{\text{TM-E-HALT}}$
- Die Idee ist, zu jeder TM  $M$  eine arithmetische Formel  $\psi_M$  zu konstruieren, so dass gilt:  
 $(\mathbb{N}_0, 0, 1, +, \times) \models \psi_M \iff M(\epsilon) \text{ terminiert nicht}$
- Dabei wird das Konzept der **Gödelisierung** verwendet:
  - Wie wir bei der Simulation von TMs durch GOTO-Programme gesehen haben, lassen sich Konfigurationen einer TM in vier Zahlen kodieren
  - Es lässt sich zeigen, dass beliebig lange (endliche) Konfigurationsfolgen auch durch einzelne Zahlen kodiert werden können
  - Außerdem lässt sich eine Formel  $\varphi_M(x)$  konstruieren, die ausdrückt, dass eine gegebene Zahl  $x$  eine terminierende Berechnung von  $M$  bei leerer Eingabe kodiert
  - Dann sei  $\psi_M \stackrel{\text{def}}{=} \neg \exists x \varphi_M(x)$



# Hilberts zehntes Problem

- David Hilbert hat in seiner Rede beim internationalen Mathematikerkongress im Jahre 1900 für das neue Jahrhundert 23 zu lösende Probleme formuliert
- Das zehnte Problem war, ein Verfahren zu finden, das für eine beliebige diophantische Gleichung entscheidet, ob sie lösbar ist
- Etwas umformuliert geht es um das folgende algorithmische Problem:

## Definition: HILBERT10

**Gegeben:** Ganzzahliges Polynom  $f(x_1, \dots, x_k)$

**Frage:** Gibt es  $n_1, \dots, n_k \in \mathbb{Z}$  mit  
$$f(n_1, \dots, n_k) = 0?$$

- Matyasevich hat 1970 bewiesen, dass dieses Problem nicht entscheidbar ist

# Abstufungen unentscheidbarer Probleme (1/2)

- Die nicht entscheidbaren Probleme lassen sich unterteilen in:
  - semientscheidbare Probleme und
  - nicht semientscheidbare Probleme
- Es gibt noch erheblich weitergehende Unterteilungen

- Es lässt sich zeigen, dass eine Sprache  $L$  genau dann semientscheidbar ist, wenn es eine entscheidbare Sprache  $L'$  (von Tupeln von Strings) gibt, so dass für alle Strings  $w \in \Sigma^*$  gilt:  
$$w \in L \iff \exists x_1, \dots, x_\ell \in \Sigma^* : (w, x_1, \dots, x_\ell) \in L'$$

## • Arithmetische Hierarchie:

- $\Sigma_1^0 \stackrel{\text{def}}{=} \text{semientscheidbare Sprachen}$
- $\Sigma_2^0 \stackrel{\text{def}}{=} \text{Sprachen } L, \text{ für es eine entscheidbare Sprache } L' \text{ gibt, so dass gilt:}$   
$$w \in L \iff \exists x_1, \dots, x_\ell \in \Sigma^* \forall y_1, \dots, \forall y_m \in \Sigma^* : (w, x_1, \dots, x_\ell, y_1, \dots, y_m) \in L'$$
- $\Sigma_k^0 \stackrel{\text{def}}{=} \text{analog mit } k \text{ Quantorenblöcken, beginnend mit einem Block von Existenzquantoren}$
- $\Pi_k^0 \stackrel{\text{def}}{=} \text{analog zu } \Sigma_k^0, \text{ beginnend mit einem Block von Allquantoren}$

- Wir betrachten im Folgenden wieder algorithmische Probleme statt Sprachen
  - Die Quantifizierung kann dann auch über andere abzählbare Mengen erfolgen (z.B.:  $\mathbb{N}$ )

## Abstufungen unentscheidbarer Probleme (2/2)

- Eine Sprache  $L$  heißt **vollständig** für eine Klasse  $\mathcal{C}$ , wenn
  - $L \in \mathcal{C}$  und
  - für jede Sprache  $L' \in \mathcal{C}$  gilt:  
 $L' \leq L$

### Beispiel

- TM-DIAG, TM-HALT, und TM-E-HALT sind vollständig für  $\Sigma_1^0$

- Hält eine gegebene TM  $M$  nur für endlich viele Strings?
  - $\exists m \in \mathbb{N} \forall z \in \Sigma^* \forall n \in \mathbb{N}$ :
    - \* falls  $|z| > m$  läuft  $M(z)$  mindestens  $n$  Schritte ohne zu akzeptieren
  - Dieses Problem ist vollständig für  $\Sigma_2^0$

- Ist die von  $M$  berechnete Funktion  $f_M$  total?
  - $\forall y \in \Sigma^* \exists n \in \mathbb{N}$ :
    - \*  $M(y)$  hält nach spätestens  $n$  Schritten an und erzeugt eine Ausgabe
  - Dieses Problem ist vollständig für  $\Pi_2^0$

- Ist  $W(f_M)$  entscheidbar?
  - $\exists \text{TM } M' \forall y_1, y_2 \in \Sigma^* \forall m \in \mathbb{N}$   
 $\exists z \in \Sigma^* \exists n_1, n_2 \in \mathbb{N}$ :
    - \*  $M'(y_1)$  terminiert nach spätestens  $n_1$  Schritten
    - \* falls  $M(y_2)$  nach  $m$  Schritten die Ausgabe  $y_1$  hat, akzeptiert  $M'(y_1)$ , und
    - \* falls  $M'(y_1)$  akzeptiert, hat  $M(z)$  nach  $n_2$  Schritten die Ausgabe  $y_1$
  - Dieses Problem ist vollständig für  $\Sigma_3^0$

- Es gibt noch die **analytische Hierarchie...**

# Zusammenfassung

- Es gibt universelle Turingmaschinen
- Die  $\mu$ -rekursiven Funktionen sind genau die berechenbaren Funktionen
- Die primitiv rekursiven Funktionen sind genau die LOOP-berechenbaren Funktionen
- Aus der Unentscheidbarkeit des Halteproblems lässt sich folgern, dass die Erfüllbarkeit prädikatenlogischer Formeln und einige verwandte Probleme unentscheidbar sind
- Die unentscheidbaren Probleme lassen sich weiter klassifizieren, z.B., durch die Klassen der Arithmetischen Hierarchie

# Literatur

- Arithmetische Hierarchie:
  - Heribert Vollmer: Berechenbarkeit und Logik: Vorlesungsskript, Uni Hannover, 2005

# Erläuterungen

## Bemerkung 16.1

- Wir verwenden den Namen num hier für formal verschiedene Definitionsbereiche:  $Q, \Gamma, \{\leftarrow, \downarrow, \rightarrow\}$

# Grundbegriffe der Theoretischen Informatik

Sommersemester 2016 - Thomas Schwentick

Teil D: Komplexitätstheorie

17: Polynomielle Zeit

Version von: 23. Juni 2016 (14:08)

# Inhalt

## ▷ **17.1 Zwei algorithmische Probleme**

17.2 Berechnungsaufwand und Komplexitätstheorie

17.3 Laufzeit und erweiterte Church-Turing-These

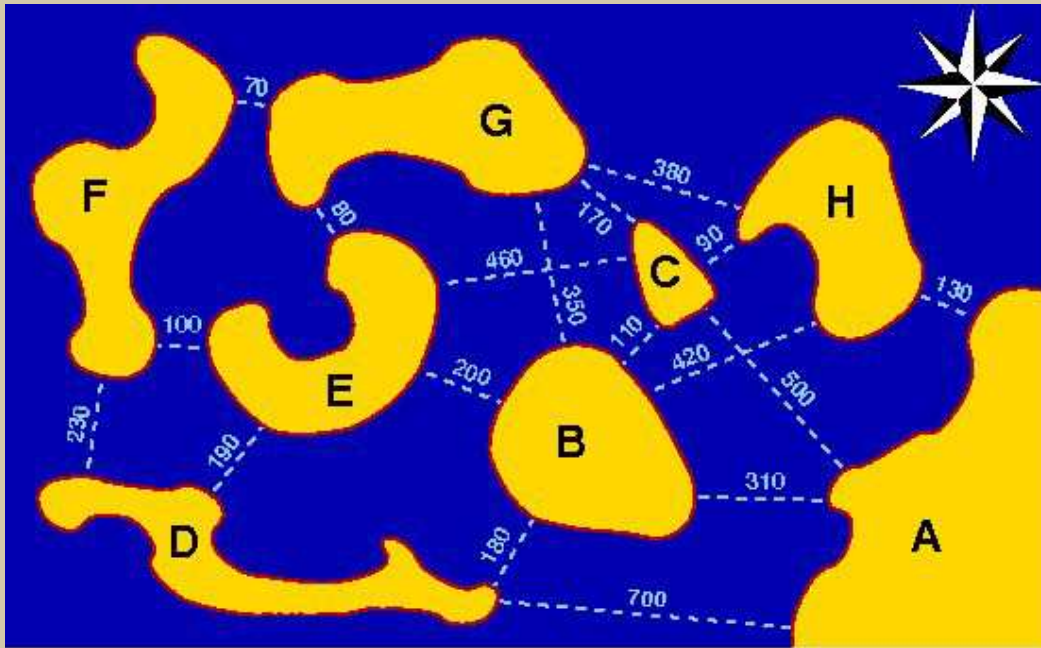
17.4 Effizient lösbare Entscheidungsprobleme

17.5 Optimierungs- vs. Entscheidungsprobleme




# Sparsamer Brückenbau

## Beispiel



- Einst lebten in einem fernen Inselreich die Algolaner
- Sie wohnten verstreut auf allen sieben Inseln
- Zwischen den sieben Inseln und dem Festland verkehrten mehrere Fähren, die gegenseitige Besuche und Ausflüge auf das Festland ermöglichten
- Die Fährverbindungen sind in der Karte gestrichelt eingezeichnet

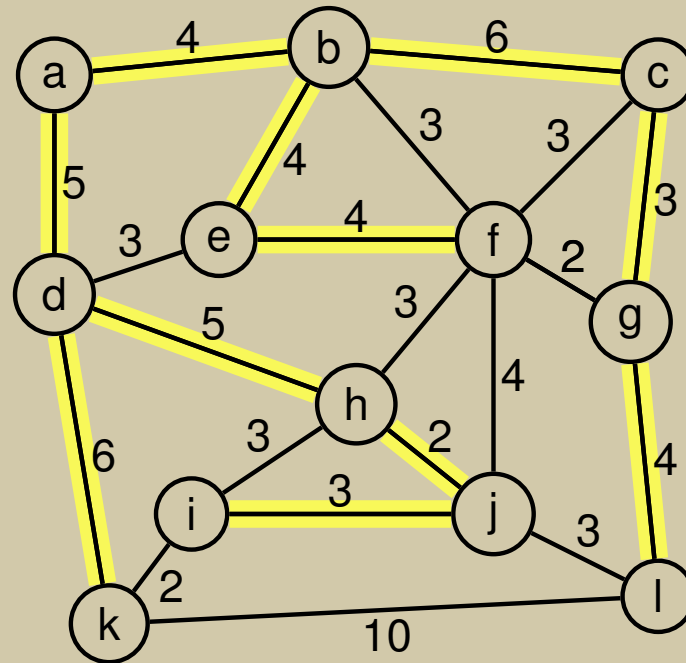
 Die Zahlen geben die Länge der Fährverbindungen in Metern an

## Beispiel (Forts.)

- Bei stürmischem Wetter kam es regelmäßig vor, dass eine Fähre kenterte
  - Deshalb beschlossen die Algolaner, einige Fährverbindungen durch Brücken zu ersetzen
  - Natürlich sollte der Bauaufwand dafür möglichst gering sein
- 
- Das Beispiel führt uns zu dem Problem der **Minimalen Spannbäume**
  - Das Beispiel stammt von Katharina Langkau und Martin Skutella (TU Berlin)
  - Die Quelle zu diesem Beispiel und viel mehr über Algorithmen finden Sie unter „**Algorithmus der Woche**“

# Minimale Spannbäume

Beispiel: ein Spannbaum



Gewicht: 46

Definition: MINSPANNINGTREEO

**Gegeben:** Graph  $G = (V, E)$   
(ungerichtet, zusammenhängend)  
Gewichtsfunktion  $\ell : E \rightarrow \mathbb{N}$

**Gesucht:** Aufspannender Baum  $T \subseteq E$  von  $G$   
mit minimalem Gesamtgewicht  $\sum_{e \in T} \ell(e)$

# Minimale Spannbäume: Algorithmus

## Algorithmus von Prim

**Eingabe:** Graph  $G = (V, E)$ ,  
Gewichtsfunktion  $\ell$

**Ausgabe:** Spannbaum  $(V, T)$ ,  $T \subseteq E$ ,  
minimalen Gewichts

$r :=$  beliebiger Knoten aus  $V$

$R := \{r\}; Q := V - \{r\}$

$T := \emptyset$

WHILE  $Q \neq \emptyset$  DO

$(u, v) :=$  Kante minimalen Gewichts  
        mit  $u \in R, v \in Q$

$T := T \cup \{(u, v)\}$

$R := R \cup \{v\}$

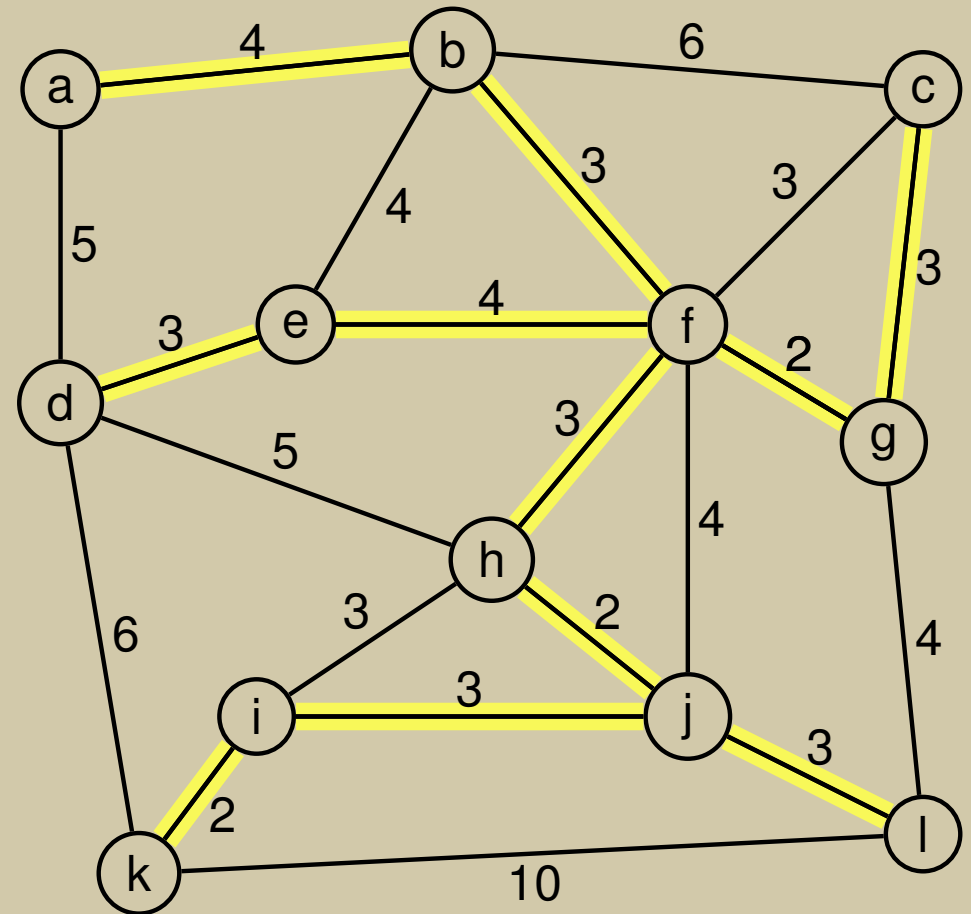
$Q := Q - \{v\}$

END

Ausgabe  $T$

- Aufwand, bei geschickter Implementierung:  
 $\mathcal{O}(|V| \log(|V|) + |E|)$  Schritte

## Beispiel



Gewicht: 32

# Komfortabler Brückenbau

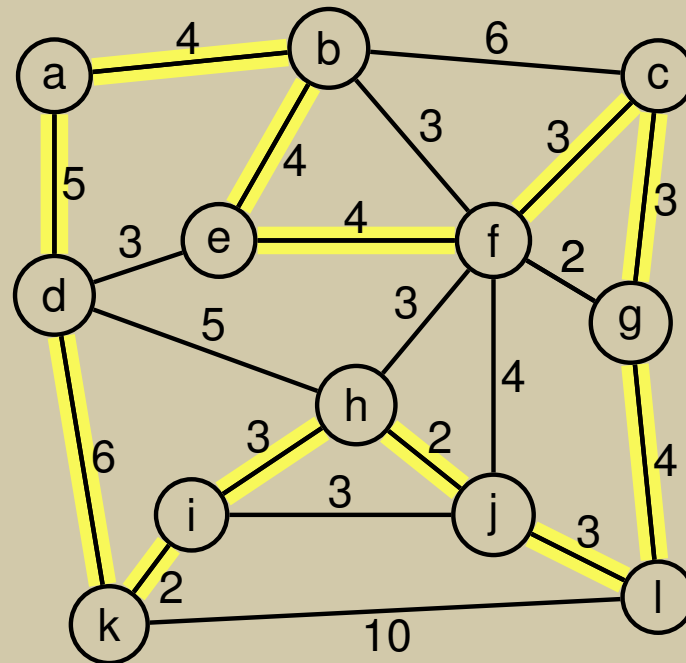
## Beispiel

WWW

- Nach reiflicher Überlegung beschloss das Oberhaupt des Inselstaates, für den Brückenbau ein anderes Optimierungskriterium zu verwenden:
  - Die Brücken sollten so konstruiert werden, dass er auf seiner wöchentlichen Rundreise durch sein Reich einen möglichst kurzen Brückensamtweg zurücklegen muss

# Minimale Kreise

## Beispiel



Gesamtstrecke: 41

## Definition: MINCYCLEO

**Gegeben:** Graph  $G = (V, E)$   
(ungerichtet, zusammenhängend)

Entfernungsfunktion  $\ell : E \rightarrow \mathbb{N}$

**Gesucht:** Kreis  $K \subseteq E$  durch alle Knoten mit minimalem Gesamtgewicht  $\sum_{e \in K} \ell(e)$  (oder  $\perp$ )

# Minimale Kreise: Algorithmus

- **Algorithmus für MINCYCLEO:**

- Zähle alle Folgen  $v_{i_1}, \dots, v_{i_n}$  von Knoten von  $G$  auf, die jeden Knoten genau einmal enthalten
- Teste jeweils, ob  $(v_{i_1}, v_{i_2}) \dots, (v_{i_n}, v_{i_1})$  ein Kreis ist
- Wähle den Kreis mit minimalem Kantengewicht aus

- Aufwand im schlimmsten Fall:

$$\text{mindestens } n! \sim \left(\frac{n}{e}\right)^n \sqrt{2\pi n}$$

- Ein wesentlich besserer Algorithmus ist nicht bekannt

- MINCYCLEO ist eine Variante des Problems des Handlungsreisenden („Traveling Salesperson“):

- Ein Handlungsreisender soll eine gegebene Menge von Städten auf einer möglichst kurzen Rundreise besuchen
- Viele Anwendungen:
  - \* Transport- und Logistikprobleme (Paketdienst, Schulbus, Versand,...)
  - \* Maschinensteuerung

# Effizient lösbare algorithmische Probleme: Vorbemerkungen

- In Teil C der Vorlesung ging es um die Frage, welche algorithmischen Probleme überhaupt mit Computern gelöst werden können
  - Dabei haben wir uns dann vor allem mit Problemen beschäftigt, die **nicht** lösbar sind
- Wenn ein Problem wirklich mit Computern gelöst werden soll, genügt es aber nicht, dass es prinzipiell lösbar ist
  - Es sollte auch einigermaßen „effizient“ lösbar sein
- ✎ Wenn das Ergebnis einer Berechnung erst nach einigen Menschengenerationen vorliegt, könnte es sein, dass die Frage schon in Vergessenheit geraten ist: **42**
- In Teil D der Vorlesung geht es um die Grenze zwischen algorithmischen Problemen, die **effizient** mit Computern gelöst werden können, und solchen, die (scheinbar) **nicht** effizient lösbar sind
- In diesem Kapitel beschäftigen wir uns mit folgenden Fragen
- Wie wird der Berechnungsaufwand eines Algorithmus bzw. eines algorithmischen Problems definiert?
  - Welche Rolle spielt dabei die Kodierung der Eingabe?
  - Welche Rolle spielt das zugrunde gelegte Berechnungsmodell?
- Welche Arten von Ressourcenbeschränkungen von Berechnungen werden (üblicherweise) betrachtet?
- Wann wird ein algorithmisches Problem als effizient lösbar betrachtet?

# Inhalt

17.1 Zwei algorithmische Probleme

▷ **17.2 Berechnungsaufwand und Komplexitätstheorie**

17.3 Laufzeit und erweiterte Church-Turing-These

17.4 Effizient lösbare Entscheidungsprobleme

17.5 Optimierungs- vs. Entscheidungsprobleme



# Ressourcen

- Der Aufwand einer Berechnung lässt sich hinsichtlich verschiedener Ressourcen messen:
  - Laufzeit
  - Benötigter Speicherplatz
  - Energieverbrauch
  - Anzahl Prozessoren
  - ...
- Die mit Abstand am meisten betrachteten Ressourcen sind dabei Laufzeit und Speicherplatz
- Speicherplatz kann eine wesentlich kritischere Resource sein als Laufzeit, da er nicht alleine durch Geduld vergrößert werden kann
- In dieser Vorlesung werden wir uns aber fast ausschließlich mit Laufzeit beschäftigen

# Laufzeit: asymptotischer Worst-Case-Aufwand

- Wie schon aus DAP 2 bekannt ist, wird meist das **asymptotische** Laufzeitverhalten von Algorithmen betrachtet:
  - Der Aufwand wird als Funktion in der Größe der Eingabe für wachsende Eingabegrößen untersucht
- Zumeist wird der **Worst-Case**-Aufwand betrachtet:
  - Wenn ein Algorithmus eine Worst-Case-Laufzeit  $\mathcal{O}(n^2)$  hat, so gibt es Konstanten  $c$  und  $n_0$ , so dass der Algorithmus für **jede Eingabe** der Größe  $n > n_0$  maximal  $cn^2$  Schritte benötigt
- Worst-Case Aufwand bietet eine Garantie, dass der Algorithmus in jedem Fall nach einer bestimmten Schrittzahl zum Ende kommt (abhängig von der Eingabegröße)
- Eine Alternative wäre beispielsweise die Betrachtung der durchschnittlichen Laufzeit
  - Sie führt jedoch zu einer Reihe von Schwierigkeiten (siehe später in diesem Kapitel)

# Komplexitätstheorie: Vorbemerkungen (1/2)

- Die Komplexitätstheorie will nicht nur die Laufzeit einzelner *Algorithmen* untersuchen
  - Sie interessiert sich vielmehr für die prinzipielle algorithmische Schwierigkeit von *algorithmischen Problemen*
    - Gibt es für ein bestimmtes Problem überhaupt einen Algorithmus mit einem bestimmten Worst-Case Aufwand?
  - Das Ziel sind dabei nicht möglichst präzise Schranken ( $\mathcal{O}(n \log n)$  vs.  $\mathcal{O}(n^2)$ ) sondern eine grobe Kategorisierung der Probleme nach ihrer algorithmischen Schwierigkeit
    - z.B.: „effizient lösbar“ vs. „nicht effizient lösbar“
  - Die Komplexitätstheorie fasst Probleme mit ähnlichem Ressourcenverbrauch in **Komplexitätsklassen** zusammen
  - Komplexitätsklassen werden üblicherweise durch drei Komponenten beschrieben:
    - **Modus der Berechnung:**
      - \* z.B., deterministisch, nicht-deterministisch, probabilistisch, parallel
    - **Art der betrachteten Ressource:**
      - \* z.B.: Laufzeit, Speicherbedarf, Anzahl Zufallsbits, Prozessorenzahl
    - **Wachstumsverhalten bzgl. der betrachteten Ressource:**
      - \* z.B.: logarithmisch, polynomiell, exponentiell
  - Bekannteste offene Frage: **Ist  $P \neq NP$ ?**
    - D.h.: können in polynomieller Zeit mehr Probleme nichtdeterministisch (**NP**) als deterministisch (**P**) gelöst werden?
- Hauptthema in diesem Teil der Vorlesung

## Komplexitätstheorie: Vorbemerkungen (2/2)

- Verhältnis zwischen verschiedenen Teilgebieten der **Algorithmentheorie**:
  - **Berechenbarkeitstheorie**: Klassifikation von Problemen nach entscheidbar und (verschiedenen Graden von) unentscheidbar
  - **Komplexitätstheorie**: Klassifikation von Problemen nach algorithmischer Schwierigkeit
  - **Effiziente Algorithmen**: Konstruktion möglichst effizienter Algorithmen

# Inhalt

17.1 Zwei algorithmische Probleme

17.2 Berechnungsaufwand und Komplexitätstheorie

▷ **17.3 Laufzeit und erweiterte Church-Turing-These**

17.4 Effizient lösbare Entscheidungsprobleme

17.5 Optimierungs- vs. Entscheidungsprobleme

## Laufzeit: Vorbemerkungen

- Wir beschäftigen uns jetzt mit der Laufzeit von Algorithmen und insbesondere mit den schon genannten Fragen:
  - Welche Rolle spielt die Kodierung der Eingabe?
  - Welche Rolle spielt das zugrunde gelegte Berechnungsmodell?

## Laufzeit: Definitionen (1/2)

- Wir definieren jetzt formal die Laufzeit von Algorithmen
- Es stellt sich die Frage, inwieweit die Laufzeit eines Algorithmus vom zugrunde liegenden Berechnungsmodell abhängt
- Wir betrachten zunächst die Laufzeit von Turingmaschinen und GOTO-Programmen
- Wir müssen uns jeweils überlegen, was wir als einzelnen Schritt einer Berechnung zählen wollen
- Bei der formalen Definition der Laufzeit gehen wir davon aus, dass die Eingabe kodiert als String (bei TMs) oder Zahl (bei GOTO-Programmen) vorliegt

### Definition


#### • Turingmaschinen:

- Ist  $K_0(x) \vdash_M K_1 \vdash_M K_2 \vdash \cdots \vdash_M K_m$  eine Berechnung einer TM  $M$  bei Eingabe  $x$  und ist  $K_m$  Halte-Konfiguration, so definieren wir
  - \*  $\underline{t_M(x)} \stackrel{\text{def}}{=} m$  (Laufzeit von  $M$  bei Eingabe  $x$ )
- Falls keine solche Folge existiert:
  - \*  $\underline{t_M(x)} \stackrel{\text{def}}{=} \perp$

#### • GOTO-Programme:

- Analog wie bei TMs definieren wir für GOTO-Programme  $P$ :
  - \*  $\underline{t_P(n)} \stackrel{\text{def}}{=} m-1$  (Laufzeit von  $P$  bei Eingabe  $n$ ) falls  $(M_1, X_1) \vdash_P \cdots \vdash_P (M_m, X_m)$  mit Haltekonfiguration  $(M_m, X_m)$  und  $X_1 \stackrel{\text{def}}{=} X_{\text{Init}}^n$

  $\vdash_P$  bezeichnet die Nachfolgekonnfigurationsrelation

-  Bei LOOP-/WHILE-Programmen verzichten wir auf die (etwas kompliziertere) formale Definition des Aufwandes
- Intuitiv: je Zuweisung oder Schleifen-Test 1 Schritt

## Laufzeit: Definitionen (2/2)

- Die genaue Schrittzahl einzelner Berechnungen ist für unsere zukünftigen Untersuchungen meistens nicht interessant
- Stattdessen interessieren uns **asymptotische obere Schranken** des Aufwandes bei größer werdenden Eingaben
- Deshalb definieren wir zunächst die Größe einer Eingabe

### Definition: Eingabegröße

**Turingmaschinen:** Die Eingabegröße ist die Länge des Eingabestrings

### LOOP-/WHILE-/GOTO-Programme:

Die Eingabegröße ist die Länge der Kodierung der Eingabezahl, also  $|\text{N2Str}(n)| = \lfloor \log(n + 1) \rfloor$  bei Eingabe  $n$

### Definition: Zeitschranke

**Turingmaschinen:** Eine Funktion


$T : \mathbb{N} \rightarrow \mathbb{R}$  heißt **Zeitschranke** für eine TM  $M$ , falls es ein  $n_0 \in \mathbb{N}$  gibt, so dass für alle  $x \in \Sigma^*$  mit  $|x| > n_0$  gilt:

$$t_M(x) \leq T(|x|)$$

**LOOP-/WHILE-/GOTO-Programme:** Eine

Funktion  $T : \mathbb{N} \rightarrow \mathbb{R}$  heißt **Zeitschranke** für ein LOOP-, WHILE- oder GOTO-Programm  $P$ , falls es ein  $n_0 \in \mathbb{N}$  gibt, so dass für alle  $n \in \mathbb{N}$  mit  $n > n_0$  gilt:

$$t_P(n) \leq T(\lfloor \log(n + 1) \rfloor)$$

 Wir gehen hier davon aus, dass GOTO- und WHILE-Programme Anweisungen der Art  $x_i := x_j + x_k$  verwenden dürfen

- Die „Zeitschranken-Eigenschaft“ muss nur für genügend große Eingaben gelten
- Wir betrachten hier nur TMs und Programme, die immer terminieren

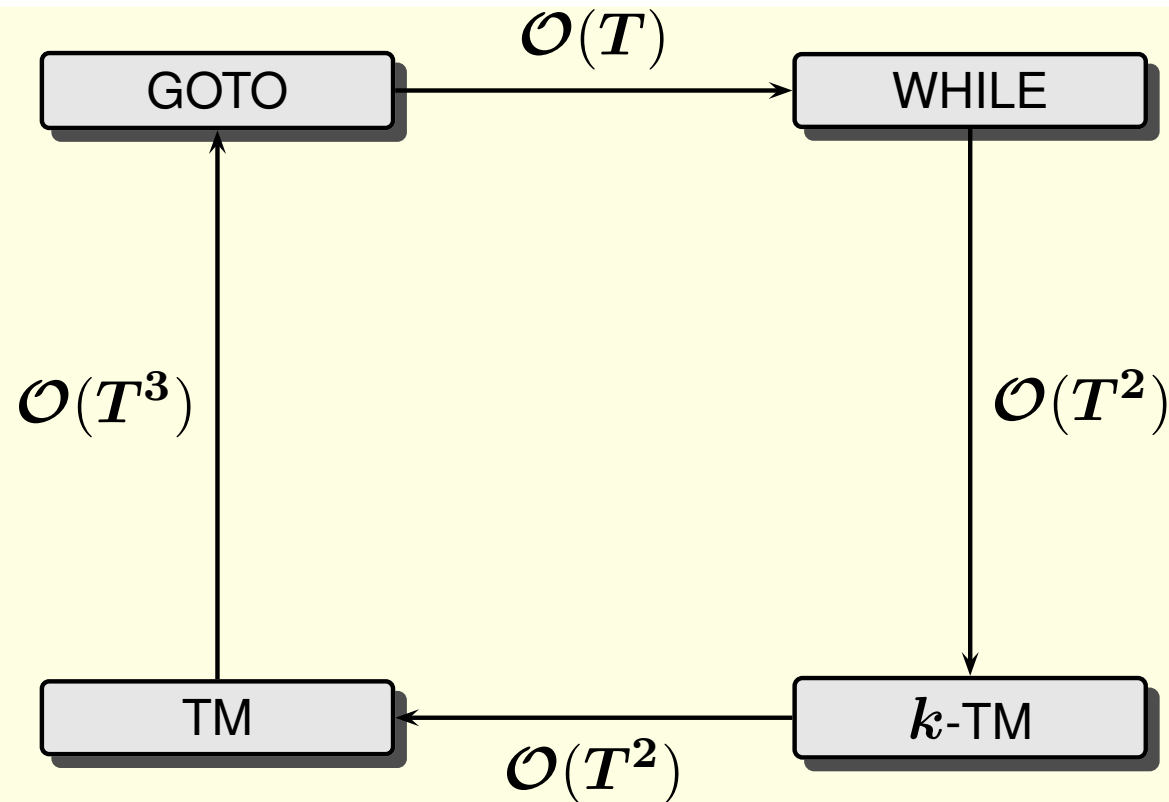


# Erweiterte Church-Turing-These (1/2)

## Lemma 17.1

- Seien  $f : \Sigma^* \rightarrow \Sigma^*$ ,  $g : \mathbb{N} \rightarrow \mathbb{N}$  und sei  $T : \mathbb{N} \rightarrow \mathbb{N}$ , wobei für alle  $n \in \mathbb{N}$  gilt:  $T(n) \geq n$
- Dann gelten die folgenden Aussagen:
  - (a) Ist  $P$  ein GOTO-Programm, das  $g$  mit Zeitschranke  $T$  berechnet, so gibt es ein WHILE-Programm  $P'$ , das  $g$  mit Zeitschranke  $\mathcal{O}(T)$  berechnet
  - (b) Ist  $P$  ein WHILE-Programm, das  $g$  mit Zeitschranke  $T$  berechnet, so gibt es eine  $k$ -String-TM  $M$ , die  $g$  im Sinne von Satz 13.4 mit Zeitschranke  $\mathcal{O}(T^2)$  berechnet
  - (c) Ist  $M$  eine  $k$ -String-TM, die  $f$  mit Zeitschranke  $T$  berechnet, so gibt es eine 1-String-TM  $M'$ , die  $f$  mit Zeitschranke  $\mathcal{O}(T^2)$  berechnet
  - (d) Ist  $M$  eine 1-String-TM, die  $f$  mit Zeitschranke  $T$  berechnet, so gibt es ein GOTO-Programm  $P$ , das  $g$  im Sinne von Satz 13.5 mit Zeitschranke  $\mathcal{O}(T^3)$  berechnet
- Also: alle hier genannten Berechnungsmodelle sind bezüglich des Zeitaufwandes „polynomiell äquivalent“
- Beweis durch genaue Analyse der jeweiligen Simulationen

## Erweiterte Church-Turing-These (2/2)



- Die Äquivalenz aus Lemma 17.1 lässt sich auf alle in Kapitel 13 genannten Berechnungsmodelle ausdehnen
- Die **erweiterte Church-Turing-These** besagt, dass sie sich auf **alle „vernünftigen“** Berechnungsmodelle erweitern lässt:
  - Also: „vernünftige“ Berechnungsmodelle unterscheiden sich hinsichtlich ihrer Laufzeit nur um polynomielle Faktoren

## Zwei Sichtweisen: Formal vs. informell

- Bei der Analyse der Komplexität algorithmischer Probleme werden wir, je nach Kontext, eine formale oder eine informelle Sichtweise einnehmen:

### Formale Sichtweise:

- Algorithmische Probleme sind Sprachen oder Funktionen über Binärstrings
- Algorithmen sind Turingmaschinen
- Diese Sichtweise ist geeignet um Aussagen zu *beweisen*
  - \* insbesondere für untere Schranken
- Wir werden sie nur anwenden, wenn nötig

### Informelle Sichtweise:

- Algorithmische Probleme haben komplexe Eingaben
- Algorithmen werden in Pseudocode oder noch informeller beschrieben
- Aufwandanalyse ist grob und „handwaving“
- Wir werden diese Sichtweise einnehmen, wann immer es möglich ist
  - \* insbesondere für obere Schranken

- Bei der informellen Sichtweise stellt sich die Frage, wie die Größe der Eingabe „gemessen“ wird

- Hier werden wir recht flexibel sein
  - Bei Graphen mit  $n$  Knoten wäre die Länge der Kodierung als String beispielsweise  $n^2$ , wir werden aber als Eingabegröße die Anzahl der Knoten (also:  $n$ ) nehmen

- Wichtig ist nur, dass
  - die „formale Eingabegröße“ höchstens polynomiell größer ist als die „informelle Eingabegröße“ (im Beispiel: quadratisch)
  - und umgekehrt (in der Regel ist die „informelle Eingabegröße“ aber sowieso kleiner als die „formale Eingabegröße“)

# Inhalt

17.1 Zwei algorithmische Probleme

17.2 Berechnungsaufwand und Komplexitätstheorie

17.3 Laufzeit und erweiterte Church-Turing-These

▷ **17.4 Effizient lösbare Entscheidungsprobleme**

17.5 Optimierungs- vs. Entscheidungsprobleme

## Effizient lösbare Probleme

- Nach diesen Vorbereitungen können wir uns nun endlich der Frage zuwenden, wann wir ein algorithmisches Problem als effizient lösbar ansehen wollen
- Wir werfen dazu nochmals einen Blick auf das Laufzeitverhalten der beiden Algorithmen aus der Einleitung:

| Eingabegröße | Prims Algorithmus | MINCYCLEO-Alg. |
|--------------|-------------------|----------------|
| 10           | 0,01 Sekunden     | 0,03 Sekunden  |
| 15           | 0,02 Sekunden     | 20 Minuten     |
| 20           | 0,04 Sekunden     | >1000 Jahre    |
| 30           | 0,09 Sekunden     |                |
| 40           | 0,16 Sekunden     |                |
| 100          | 1 Sekunden        |                |
| 1000         | 1,6 Minuten       |                |

- Dabei nehmen wir an, dass die genaue Laufzeit der Implementierungen wie folgt ist:
  - Für Prim:  $(1/10.000) n^2$  Sekunden
  - Für MINCYCLEO:  $(1/1.000.000.000) n!$  Sekunden

# Polynomielle vs. exponentielle Laufzeit

- Der Prim-Algorithmus hat **polynomielle Laufzeit**, da er ein Polynom ( $cn^2$ ) als Laufzeitschranke hat
- Der naive Algorithmus für MINCYCLEO hat hingegen **exponentielle Laufzeit**
- Der **prinzipielle Unterschied** zwischen polynomieller und exponentieller Laufzeit lässt sich wie folgt beschreiben:
  - Wenn Eingaben der Größe  $n$  bisher in Zeit  $t$  bearbeitet werden können,
  - dann können mit einem doppelt so schnellen Rechner in der selben Zeit bearbeitet werden:
    - \* bei **polynomiellem Aufwand**:  
Eingaben der Größe  $cn$ , für ein  $c > 1$
    - \* bei **exponentiellem Aufwand**:  
Eingaben der Größe  $n + c$ , für ein  $c > 0$
- ✎ Die obigen Aussagen beziehen sich auf die Laufzeit von Algorithmen, nicht auf die allgemeine Komplexität der betrachteten Probleme


# Zeitbasierte Komplexitätsklassen

- Es besteht ein weitgehender Konsens darüber, dass ein Problem nur dann als effizient lösbar bezeichnet werden kann, wenn es in polynomieller Zeit lösbar ist
  - Die Frage, ob die Umkehrung auch gilt, diskutieren wir gleich
- Die erweiterte Church-Turing-These rechtfertigt nun die folgende Definition der Komplexitätsklasse **P**
  - Denn es ist egal, ob wir für die Definition von **P** Turingmaschinen oder ein anderes Berechnungsmodell zugrunde legen

## Definition

(a) Für  $T : \mathbb{N} \rightarrow \mathbb{R}$  sei **TIME**( $T$ ) die Menge aller Sprachen  $L$ , für die es eine  $k$ -String TM  $M$  mit Zeitschranke  $T$  gibt, so dass  $L = L(M)$

(b)  $\mathbf{P} \stackrel{\text{def}}{=} \bigcup_{p \text{ Polynom}} \mathbf{TIME}(p)$

 Die Komplexitätsklasse **P** wurde übrigens erst in den 60er Jahren definiert — lange Zeit nach den ersten Untersuchungen der entscheidbaren Sprachen...

## **P** $\equiv$ effizient lösbar Probleme? (1/3)

- Wir gehen in den folgenden Kapiteln davon aus, dass die Komplexitätsklasse **P** eine vernünftige Formalisierung des informellen Begriffes der „effizient lösbar Probleme“ ist
- Diese Sichtweise ist sehr weit verbreitet, aber durchaus nicht unumstritten
- Einige Einwände und mögliche Er widerungen darauf betrachten wir auf den nächsten beiden Folien



## P $\equiv$ effizient lösbare Probleme? (2/3)

- **Einwand:** Polynome mit großen Exponenten (z.B.:  $n^{1000}$ ) haben mit „effizient“ nichts zu tun

- **Aber:**

- Wenn für ein „natürliches“ Problem überhaupt ein polynomieller Algorithmus gefunden wird, gibt es (für relevante algorithmische Probleme) meistens auch einen mit kleinem Exponenten (z.B.: 2 oder 3)
- Außerdem ist es vorteilhaft, dass die Klasse der Polynome unter Komposition abgeschlossen ist
  - Programme und Unterprogramme

- **Einwand:** Worst-Case-Komplexität ist ungeeignet, der Durchschnittsfall wäre interessanter

- **Aber:**

- In vielen Fällen ist eine Laufzeit-Garantie wichtig
- Durchschnittskomplexität ist viel komplizierter zu handhaben:
  - \* Es müsste zum Beispiel die Frage beantwortet werden, wie die Wahrscheinlichkeitsverteilung der Eingaben ist
  - \* Es ist viel schwieriger die Durchschnittskomplexität eines Problems zu analysieren

## **P** $\equiv$ effizient lösbare Probleme? (3/3)

- **Einwand:** Entscheidungsprobleme (Sprachen) sind zu eingeschränkt

- **Aber:**

- Im nächsten Abschnitt werden wir sehen, dass sich die Frage nach der effizienten Lösbarkeit von Optimierungsproblemen auf die Frage nach der effizienten Lösbarkeit von Entscheidungsproblemen zurückführen lässt

- **Einwand:** Die Definition von **P** hängt von der Wahl des Berechnungsmodells ab (hier: TMs)

- **Aber:**

- Die erweiterte Church-Turing-These sagt, dass alle „vernünftigen“ Modelle sich nur polynomiell bzgl. Zeitaufwand unterscheiden

# Nicht in polynomieller Zeit lösbare Probleme

## Definition

- **EXPTIME**  $\stackrel{\text{def}}{=} \bigcup_{p \text{ Polynom}} \text{TIME}(2^p)$

- Die Klasse **EXPTIME** ist eine echte Oberklasse von **P**, d.h., sie umfasst **P**, enthält aber auch Probleme, die sich nicht in polynomieller Zeit lösen lassen
  - Und das lässt sich auch beweisen
- Beispiel: Das Problem zu entscheiden, ob beim Brettspiel GO auf einem  $n \times n$ -Brett der erste Spieler eine Gewinnstrategie hat, ist in **EXPTIME** aber nicht in **P**
  - Dies gilt für die Ko-Regel, die Stellungswiederholungen verbietet

## Polynomielle Zeitschranken: $n^k$

- Wir nutzen die folgende **Beobachtung**, um uns den Umgang mit komplizierten Polynomen als Zeitschranken zu ersparen:
  - Wenn eine Turingmaschine  $M$  eine polynomielle Zeitschranke hat, dann hat sie auch eine Zeitschranke der Form  $n^k$ , für ein geeignetes  $k$

- Denn:

- Sei  $p(n) = \sum_{i=0}^{\ell} c_i n^i$  ein Polynom, das Zeitschranke für  $M$  ist
  - \* Es gibt also ein  $n_0$ , so dass für alle Strings  $x$  mit  $|x| > n_0$  gilt:  
 $t_M(x) \leq p(|x|)$
- Wir wählen
  - \*  $n'_0 \stackrel{\text{def}}{=} \max\{n_0, c_0, \dots, c_\ell\}$  und
  - \*  $k \stackrel{\text{def}}{=} \ell + 2$
- Dann gilt für alle  $n > n'_0$ :  
$$p(n) \leq n'_0 \sum_{i=0}^{\ell} n^i \leq n'_0 n^{\ell+1} \leq n^{\ell+2}$$

→ Wir werden also bei Problemen aus **P** zukünftig davon ausgehen, dass sie eine Zeitschranke der Form  $n^k$  haben

# Inhalt

17.1 Zwei algorithmische Probleme

17.2 Berechnungsaufwand und Komplexitätstheorie

17.3 Laufzeit und erweiterte Church-Turing-These

17.4 Effizient lösbare Entscheidungsprobleme

▷ **17.5 Optimierungs- vs. Entscheidungsprobleme**

# Optimierungsprobleme vs. Entscheidungsprobleme (1/6)

- Einige algorithmische Probleme haben „ja“ oder „nein“ als Antwort, (**Entscheidungsprobleme**)  
andere suchen eine optimale Lösung  
(**Optimierungsprobleme**)
- Eine dritte Variante ist die Berechnung des **Wertes einer optimalen Lösung**
- Wir werden sehen: hinsichtlich polynomieller Lösbarkeit können wir uns auf Entscheidungsprobleme beschränken
- Wir betrachten die drei Varianten am Beispiel des Traveling-Salesperson-Problems

## Optimierungsprobleme vs. Entscheidungsprobleme (2/6)

- Das TSP-Problem (*Traveling Salesperson*) sucht nach der kürzesten Rundreise durch eine gegebene Menge von Städten, die jede Stadt genau einmal besucht
- Formal besteht die Eingabe zum TSP-Problem aus einer Folge  $s_1, \dots, s_n$  von Städten und einer Entfernungsfunktion  $d$ 
  - $d(s_i, s_j)$  ist die Entfernung von  $s_i$  nach  $s_j$
- Wir betrachten hier nur den symmetrischen Fall: für alle  $i, j$  ist  $d(s_i, s_j) = d(s_j, s_i)$
- Formal ist eine TSP-Reise eine bijektive Funktion  $f : \{1, \dots, n\} \rightarrow \{s_1, \dots, s_n\}$ 
  - $f(i)$  ist die  $i$ -te besuchte Stadt
- Die Gesamtstrecke  $d(f)$  einer solchen TSP-Reise  $f$  ist
$$d(f) \stackrel{\text{def}}{=} d(f(n), f(1)) + \sum_{i=1}^{n-1} d(f(i), f(i+1))$$

### Definition: TSP

**Gegeben:** Entfernungsfunktion  $d$ , Zielwert  $k \in \mathbb{N}$

**Frage:** Gibt es eine TSP-Reise  $f$  zu  $d$  mit  $d(f) \leq k$ ?

### Definition: TSPO


**Gegeben:** Entfernungsfunktion  $d$

**Gesucht:** TSP-Reise  $f$  zu  $d$  mit minimaler Gesamtstrecke

### Definition: TSPV

**Gegeben:** Entfernungsfunktion  $d$

**Gesucht:** Minimale Gesamtstrecke  $d(f)$  einer TSP-Reise zu  $d$

 Da die Entfernungsfunktion  $d$  implizit auch die Städte repräsentiert, geben wir sie nicht explizit als Eingabe der drei obigen Probleme an

# Optimierungsprobleme vs. Entscheidungsprobleme (3/6)

## Lemma 17.2

- (a) Falls TSP in polynomieller Zeit lösbar ist, dann auch TSPV
- (b) Falls TSPV in polynomieller Zeit lösbar ist, dann auch TSPO

## Beweisskizze für (a)

- Idee: **Binäre Suche**
- Annahme:  $A$  ist ein Algorithmus für TSP mit polynomieller Laufzeit
- Sei  $d$  eine Entfernungsfunktion für TSPV mit  $n$  Städten
- Sei  $m$  der maximale in  $d$  vorkommende Funktionswert
  - ➔ Die optimale Lösung hat höchstens den Wert  $N \stackrel{\text{def}}{=} nm$
- Zu beachten: die Kodierung von  $d$  als Eingabestring benötigt nur  $\mathcal{O}(n^2 \log m)$  Bits

## Beweisskizze (Forts.)

- Der Algorithmus arbeitet wie folgt:
  - 1:  $i := 0; j := N$
  - 2: **repeat**
  - 3:    $k := \left\lfloor \frac{i+j}{2} \right\rfloor$
  - 4:   **if**  $A$  sagt, dass Lösung  $\leq k$  existiert **then**
  - 5:      $j := k$
  - 6:   **else**
  - 7:      $i := k + 1$
  - 8: **until**  $i = j$
  - 9: Ausgabe  $j$  (oder  $\perp$ , wenn  $j = N + 1$ )
- Korrektheit: Durch Induktion nach der Anzahl der Schleifendurchläufe ist leicht zu zeigen, dass der optimale Wert immer in  $[i, j]$  liegt
- Laufzeit:
  - In jedem Durchlauf wird  $j - i$  ungefähr halbiert
  - ➔  $\mathcal{O}(\log N)$  Schleifendurchläufe
  - Jeder Durchlauf benötigt nur polynomielle Zeit
  - ➔ Insgesamt polynomielle Laufzeit



# Optimierungsprobleme vs. Entscheidungsprobleme (4/6)

## Beweisskizze für (b)

- Sei  $d$  eine TSPO-Eingabe mit  $n$  Städten und sei  $m$  wieder der maximal vorkommende Funktionswert von  $d$
- Für zwei Indizes  $k, \ell$  bezeichne  $d_{(k,\ell)}$  die Entfernungsfunktion definiert durch:
$$d_{(k,\ell)}(s_i, s_j) \stackrel{\text{def}}{=} \begin{cases} m + 1 & \text{falls } i = k \text{ und } j = \ell \text{ (oder umgekehrt)} \\ d(s_i, s_j) & \text{sonst} \end{cases}$$
- Die beiden folgenden Aussagen sind für jedes Paar  $k, \ell$  mit  $k \neq \ell$  äquivalent:
  - die minimale Gesamtstrecke zu  $d_{(k,\ell)}$  ist gleich der minimalen Gesamtstrecke zu  $d$
  - es gibt zu  $d$  eine minimale TSP-Reise  $f$ , die **nicht** direkt von  $s_k$  nach  $s_\ell$  (oder umgekehrt) geht

## Optimierungsprobleme vs. Entscheidungsprobleme (5/6)

## Beweisskizze (Forts.)


- Algorithmus:
  - 1:  $m :=$  maximaler Funktionswert von  $d$
  - 2:  $d' := d$
  - 3: **for** jedes Index-Paar  $k \neq \ell$  **do**
  - 4:   **if** optimaler Wert für  $d'_{(k,\ell)} =$   
                                optimaler Wert für  $d$  **then**
  - 5:          $d' := d'_{(k,\ell)}$
- Behauptung: nach Ende der Berechnung induzieren die Paare  $s_i, s_j$  mit  $d'(s_i, s_j) \leq m$  eine TSP-Reise zu  $d$  mit minimaler Gesamtentfernung
- Dazu lässt sich durch Induktion nach der Anzahl der Schleifendurchläufe zeigen
  - Der Wert der optimalen Lösung für  $d'$  ändert sich nicht
- Und: am Ende ist nur noch eine TSP-Reise übrig

# Optimierungsprobleme vs. Entscheidungsprobleme (6/6)

- Das TSP-Optimierungsproblem kann also (in polynomieller Zeit) auf das TSP-Entscheidungsproblem zurückgeführt (reduziert) werden

- Eine Lemma 17.2 entsprechende Aussage gilt für die meisten uns interessierenden Optimierungsprobleme:

- (a) funktioniert immer wie bei TSP

 Wenn es nur polynomiell viele Lösungswerte gibt, ist binäre Suche nicht nötig

- Für (b) muss jeweils ein individueller Ansatz gefunden werden

\* Das ist in der Regel ähnlich wie bei TSPO möglich

→ Wir beschränken uns im Folgenden deshalb der Einfachheit halber auf Entscheidungsprobleme

- **Notation:** Optimierungsprobleme haben am Ende Ihres Namens ein O, die zugehörigen Entscheidungsprobleme nicht

# Zusammenfassung

- Wir konzentrieren uns bei der Betrachtung effizient lösbarer algorithmischer Probleme auf die asymptotische Laufzeit im *worst case*
- Die in Teil C betrachteten Berechnungsmodelle ergeben eine robuste Definition der Klasse der in polynomieller Zeit lösbaren Entscheidungsprobleme
  - Formal basieren unsere Definitionen auf Berechnungen von Turingmaschinen
- Wir betrachten die Begriffe „effizient lösbar“ und „in polynomieller Zeit lösbar“ im Folgenden als gleichbedeutend
- Es gibt auch Probleme, die sich nicht in polynomieller sondern nur in (mindestens) exponentieller Zeit lösen lassen

# Grundbegriffe der Theoretischen Informatik

Sommersemester 2016 - Thomas Schwentick

Teil D: Komplexitätstheorie

18: **NP** und **NP**-Vollständigkeit

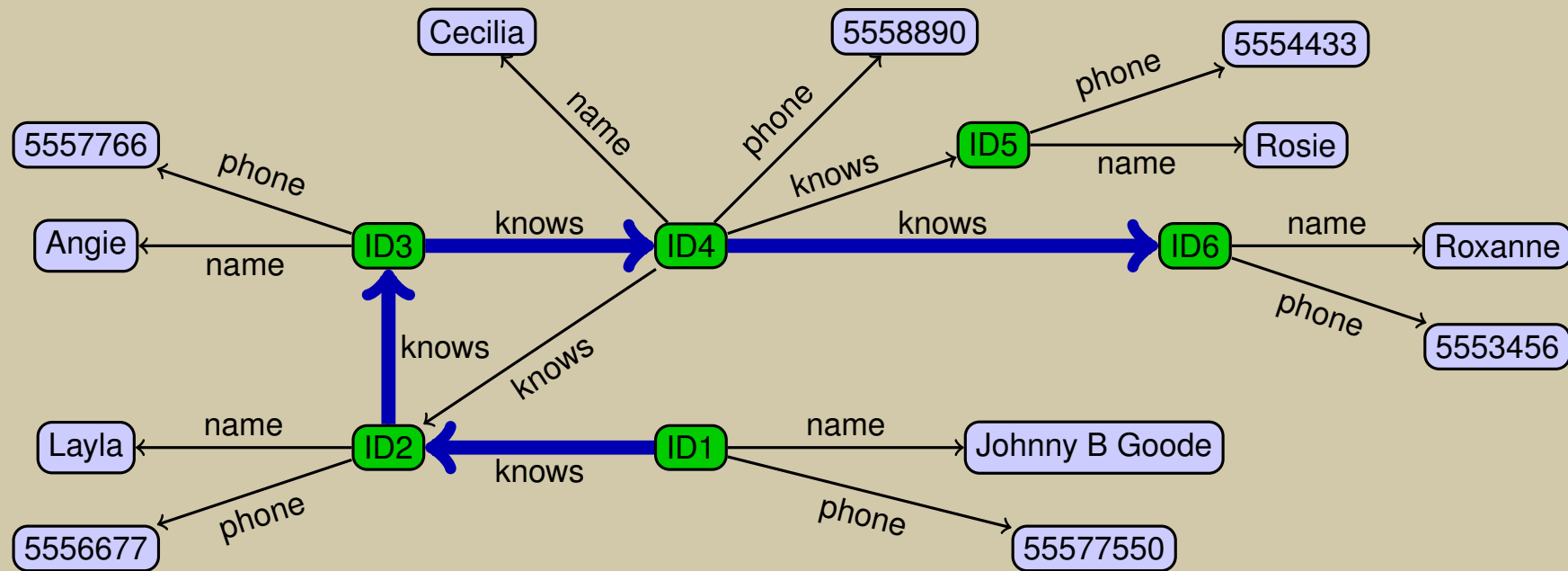
Version von: 30. Juni 2016 (14:29)

## NP-Vollständigkeit: Einleitung (1/6)

- Eine kleine Geschichte
  - ✎ nach Garey&Johnson, abgewandelt von Wim Martens
- Stellen Sie sich vor...
- Sie sollen ein Tool entwickeln, das superschnell Anfragen auf graphstrukturierten Daten auswertet
- Die Anfragen sind reguläre Ausdrücke (aka SPAR-QL)
- Bei Eingabe eines regulären Ausdrucks  $\alpha$  soll das Tool in der Datenbank Knotenpaare  $(a, b)$  finden, die durch einen Weg verbunden sind, dessen *Label-Folge* in  $L(\alpha)$  liegt
- ...aber der Weg muss wirklich ein Weg sein
  - (kein Knoten doppelt)

## NP-Vollständigkeit: Einleitung (2/6)

### Beispiel: Graph-DB und Anfrage



- **Anfrage:** Gibt es einen Weg von ID1 zu ID6, der zum regulären Ausdruck  $\alpha = (\text{knows})^*$  passt?

## NP-Vollständigkeit: Einleitung (3/6)

- Sie versuchen einen Algorithmus zu entwerfen, merken aber, dass Sie kein schnelles Verfahren für dieses Problem finden
- Sie haben schon Effizienzprobleme beim regulären Ausdruck (knows knows)\*
- Sie können das Problem kaum besser lösen, als durch Ausprobieren aller Wege, und das sind sehr, sehr viele...
- Heute ist der Termin für die Vorstellung Ihrer Ergebnisse
- **Was tun?**



## NP-Vollständigkeit: Einleitung (4/6)

- Option 1:

„Ich kann keine effiziente Methode finden. Ich glaube, ich bin einfach unfähig.“

- Wäre das klug?

## NP-Vollständigkeit: Einleitung (5/6)

- **Option 2:**  
„Ich kann keine effiziente Methode finden, weil es keine solche Methode gibt.“
- Das wäre ideal. Damit hätten Sie auch gleich eine Million Dollar verdient...

## NP-Vollständigkeit: Einleitung (6/6)

- Option 3:  
„Ich kann keine effiziente Methode finden, aber alle diese berühmten Informatiker können es auch nicht.“
- Besser als nichts. Damit das klappt, brauchen wir: **NP**-Vollständigkeit

# Übersicht

- In diesem Kapitel werden wir
  - eine Reihe schwieriger Berechnungsprobleme kennen lernen,
  - ihre Ähnlichkeit näher unter die Lupe nehmen,
  - einen Teil ihrer Ähnlichkeit durch die Definition der Komplexitätsklasse **NP** formalisieren, und
  - ein noch stärkeres Maß ihrer Ähnlichkeit durch den Begriff der **NP**-Vollständigkeit noch formalisieren

# Inhalt

## ▷ 18.1 Beispiele schwieriger Berechnungsprobleme

18.2 **NP**

18.3 **NP**-Vollständigkeit

# Schwierige Berechnungsprobleme: Rucksack

## Beispiel

- In zwei Monaten startet die nächste Rakete zur Raumstation
- Die Weltraumagentur ist etwas knapp bei Kasse und bietet deshalb kommerziellen Forschungsinstituten an, wissenschaftliche Experimente in der Raumstation durchzuführen
- Die Rakete kann noch maximal 645 kg zusätzliche Last für Experimente mitnehmen
- Die Agentur erhält von den Instituten verschiedene Angebote, in denen steht,
  - wieviel sie für Transport und Durchführung des Experiments zu zahlen bereit sind und
  - wie schwer die Geräte für ihr Experiment sind
- **Welche Experimente soll die Weltraumagentur auswählen, um den Gewinn zu maximieren?**

| Objekt-Nr.    | 1   | 2  | 3   | 4  | 5   | 6   | 7   | 8   |
|---------------|-----|----|-----|----|-----|-----|-----|-----|
| Gewicht       | 153 | 54 | 191 | 66 | 239 | 137 | 148 | 249 |
| Gewinn (1000) | 232 | 73 | 201 | 50 | 141 | 79  | 48  | 38  |

- (Beispiel von Rene Beier, Saarbrücken, Berthold Vöcking, Aachen)

- Dieses Beispiel führt zu einer Eingabe für das folgende **Rucksackproblem**:

## Definition: KNAPSACKO


**Gegeben:** Gewichtsschranke  $G$  und  $m$  Gegenstände repräsentiert durch

- Werte  $w_1, \dots, w_m$  und
- Gewichte  $g_1, \dots, g_m$ ,  
(alle Zahlen aus  $\mathbb{N}$ )

**Gesucht:**  $I \subseteq \{1, \dots, m\}$ , so dass  $\sum_{i \in I} w_i$  maximal ist und  $\sum_{i \in I} g_i \leq G$  gilt

- Informell: gesucht ist eine Menge von Gegenständen mit maximalem Gesamtwert und Gewicht  $\leq G$

# Schwierige Berechnungsprobleme: Graphfärbung

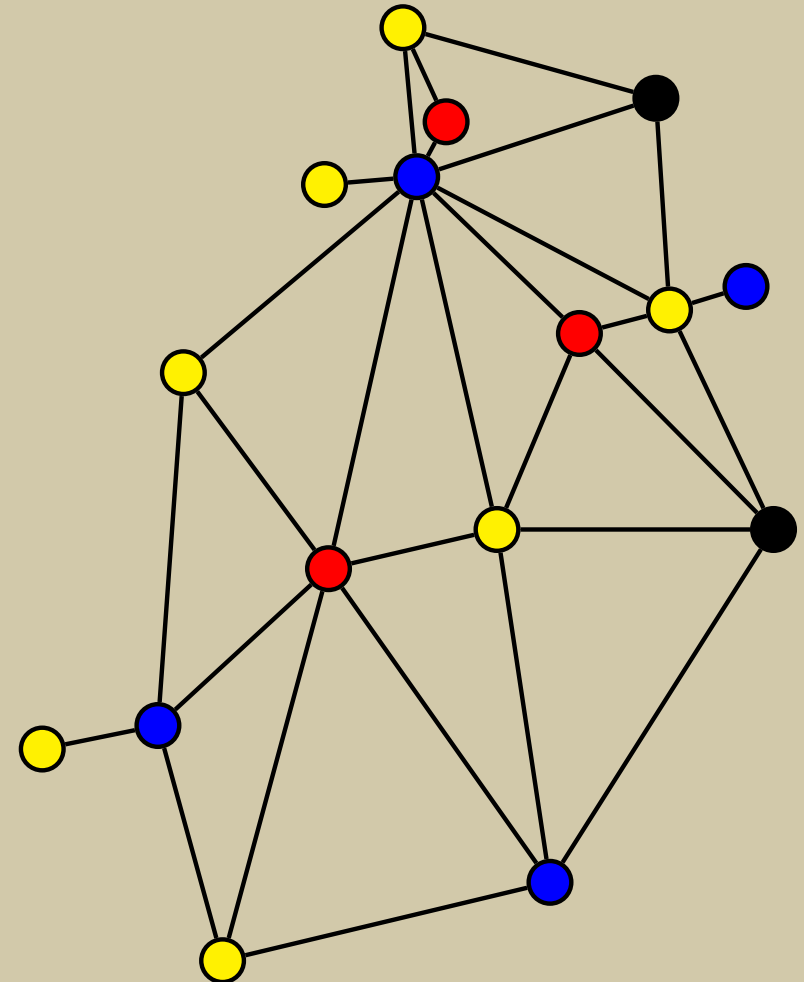
- **Landkartenfärbung:** Lassen sich die Länder einer gegebenen Landkarte mit einer gegebenen Anzahl von Farben so färben, dass benachbarte Länder verschiedene Farben haben?
- Beispiel: lässt sich die Karte der deutschen Bundesländer in dieser Art mit 3 Farben färben? **Nein!**
- 4 Farben genügen immer (wenn alle Länder zusammenhängen)  Vierfarbensatz
- Das Problem der Landkartenfärbung lässt sich zurückführen auf das allgemeinere Problem, die Knoten eines Graphen zu färben

## Definition: COL

**Gegeben:** Ungerichteter Graph  $G$ , Zahl  $k$

**Frage:** Lassen sich die Knoten von  $G$  mit  $k$  Farben **zulässig färben**, also so, dass durch Kanten verbundene Knoten verschiedene Farben haben?

## Beispiel



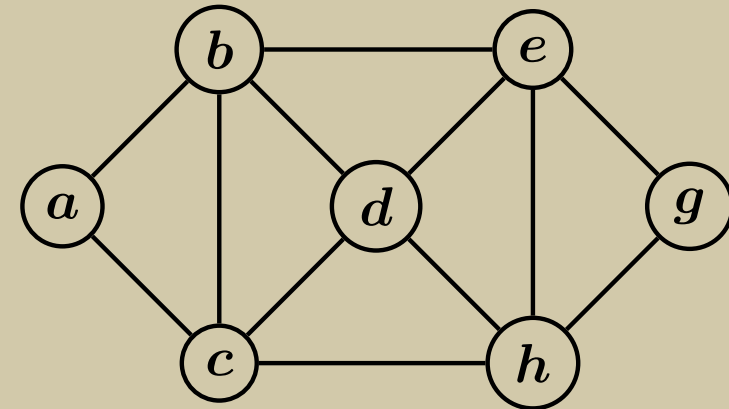
# Graphentheorie: Wiederholung

- Wir wiederholen sicherheitshalber einige Grundbegriffe aus der Graphentheorie:

## Definition

- Sei  $G = (V, E)$  ein Graph
- Sei  $v_0, \dots, v_n \in V$  eine Folge von Knoten von  $G$  mit der Eigenschaft, dass  $e_i \stackrel{\text{def}}{=} (v_{i-1}, v_i)$  für jedes  $i \in \{1, \dots, n\}$  eine Kante von  $G$  ist
- Dann heißt  $e_1, \dots, e_n$  eine Kantenfolge von  $G$
- Ist  $v_0 = v_n$ , so heißt die Kantenfolge geschlossen
- Eine Kantenfolge ist ein Weg oder Pfad, wenn die Knoten  $v_0, \dots, v_n$  paarweise verschieden sind  
(es darf allerdings  $v_0 = v_n$  gelten)
- Einen geschlossenen Weg nennen wir einen Kreis

## Beispiel

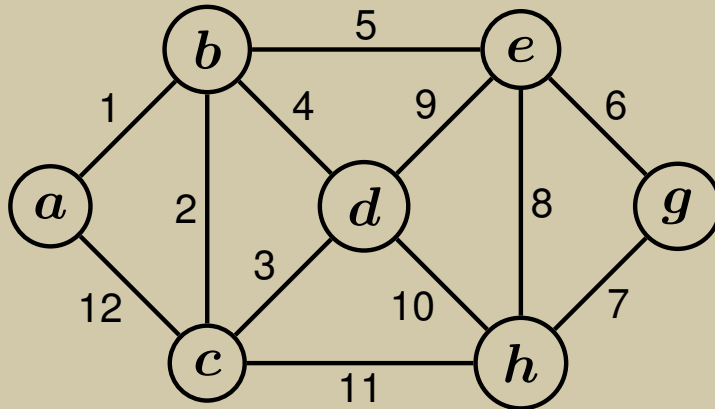


- $(a, b), (b, e), (e, d), (d, b), (b, c)$  ist eine Kantenfolge von  $G$
- $(a, b), (b, e), (e, d), (d, b), (b, c), (c, a)$  ist eine geschlossene Kantenfolge von  $G$
- $(a, b), (b, e), (e, d), (d, c)$  ist ein Weg von  $G$
- $(a, b), (b, e), (e, d), (d, c), (c, a)$  ist ein Kreis von  $G$

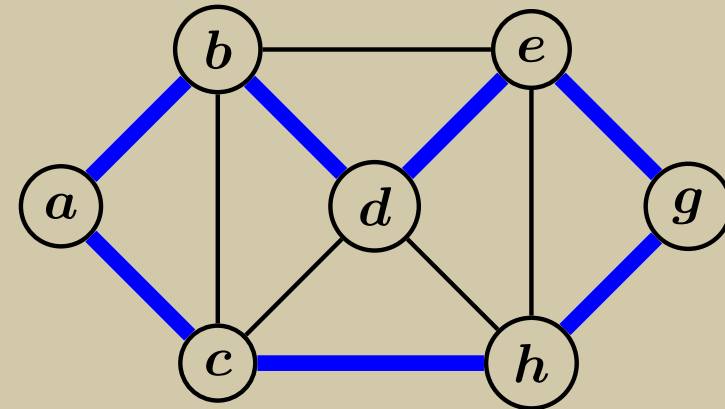


# Schwierige Berechnungsprobleme: Hamilton-Kreise

Beispiel



Beispiel



## Definition: EULERCYCLE

**Gegeben:** Ungerichteter Graph  $G$

**Frage:** Gibt es eine geschlossene Kantenfolge in  $G$ , die **jede Kante** genau einmal besucht?

## Fakt

- Ein zusammenhängender Graph  $G$  hat genau dann einen Euler-Kreis, wenn jeder Knoten geraden Grad hat

👉 gerade viele Nachbarknoten

→ Das ist in polynomieller Zeit testbar

## Definition: HAMILTONCYCLE

**Gegeben:** Ungerichteter Graph  $G$

**Frage:** Gibt es einen geschlossenen Weg in  $G$ , der **jeden Knoten** genau einmal besucht?

- Für HAMILTONCYCLE ist kein Algorithmus mit polynomieller Laufzeit bekannt
- ✍ Ein Euler-Kreis ist meistens kein Kreis sondern nur eine geschlossene Kantenfolge, die Bezeichnung Euler-Kreis ist aber allgemein üblich

# Schwierige Berechnungsprobleme: Das Cliques-Problem

- Zwei Knoten  $u, v$  eines ungerichteten Graphen  $G = (V, E)$  heißen benachbart, wenn  $(u, v) \in E$
- Eine  $k$ -Clique ist eine Menge  $C$  von  $k$  Knoten, die paarweise benachbart sind

- Das Cliques-Problem hat viele Anwendungen, z.B.
  - im Data Mining
  - in der Bioinformatik

## Definition: CLIQUEO

**Gegeben:** Graph  $G = (V, E)$

**Gesucht:** Maximale Clique in  $G$ ,  
d.h.:  
maximale Menge  $C$  von Knoten, die paarweise benachbart sind

## Definition: CLIQUE

**Gegeben:** Graph  $G = (V, E)$ ,  
Zahl  $k$

**Gesucht:** Gibt es in  $G$  eine Clique mit  $k$  Knoten?

# Schwierige Berechnungsprobleme: AL-Erfüllbarkeit

- **Aussagenlogische Formeln:**

- $x_1, x_2, x_3, \dots$  seien Variablen
- Jedes  $x_i$  ist eine aussagenlogische Formel
- Ist  $\varphi$  eine aussagenlogische Formel, so auch  $\neg\varphi$
- Sind  $\varphi_1, \varphi_2$  aussagenlogische Formeln, so auch  $\varphi_1 \wedge \varphi_2$  und  $\varphi_1 \vee \varphi_2$

- Eine Wahrheitsbelegung  $\alpha : \{x_1, x_2, \dots\} \rightarrow \{0, 1\}$  ordnet jeder Variablen einen Wert zu
- $\alpha \models \varphi$ : Die Formel  $\varphi$  erhält durch die Wahrheitsbelegung  $\alpha$  den Wert 1
- Eine Formel  $\varphi$  ist **erfüllbar**, wenn es ein  $\alpha$  gibt mit  $\alpha \models \varphi$

## Beispiel

- $(x_3 \vee x_1) \wedge (\neg x_2 \vee ((x_3 \wedge \neg x_1) \vee (x_2 \vee \neg x_1)))$

- Wir beschränken uns auf Formeln in **konjunktiver Normalform** (KNF):

$$(x_3 \vee x_1) \wedge (\neg x_2 \vee x_3 \vee x_2) \wedge (\neg x_2 \vee \neg x_3 \vee x_2) \wedge (\neg x_2 \vee x_3) \wedge (\neg x_2 \vee \neg x_3)$$

- Die  $x_i$  und  $\neg x_i$  heißen **Literale**
- Die disjunktiven Teilformeln heißen **Klauseln**
- Eine KNF-Formel ist in **3-KNF**, wenn jede Klausel genau drei Literale enthält
  - Das selbe Literal darf mehrfach in einer Klausel vorkommen

## Definition: SAT

**Gegeben:** Aussagenlogische Formel  $\varphi$  in KNF

**Frage:** Ist  $\varphi$  erfüllbar?

## Definition: 3-SAT

**Gegeben:** Aussagenlogische Formel  $\varphi$  in 3-KNF

**Frage:** Ist  $\varphi$  erfüllbar?

# Inhalt

18.1 Beispiele schwieriger Berechnungsprobleme

▷ **18.2 NP**

18.3 **NP**-Vollständigkeit

# Eigenschaften der betrachteten Probleme (1/2)

- Nur für zwei der in diesem und im letztem Kapitel erwähnten Probleme ist ein Algorithmus mit polynomieller Laufzeit bekannt:
  - EULERCYCLE
  - MINSPANNINGTREEO
- Für die folgenden Probleme ist kein solcher Algorithmus bekannt:
  - TSP
  - HAMILTONCYCLE
  - SAT
  - COL
  - KNAPSACK
  - 3-SAT
  - CLIQUE
- Wir werden jetzt untersuchen, wie „ähnlich“ diese Probleme zueinander sind

## Eigenschaften der betrachteten Probleme (2/2)

- Bei allen betrachteten Problemen gibt es für jede Eingabe eine Menge von **Lösungskandidaten** und es geht um die Frage, ob einer dieser Lösungskandidaten eine **Lösung** ist
- Lösungskandidaten sind:
  - TSP, HAMILTONCYCLE:  
alle Permutationen der Knotenmenge
  - SAT, 3-SAT: alle Wahrheitsbelegungen
  - COL: alle Färbungen mit  $k$  Farben
  - KNAPSACK:  
alle Teilmengen der Gegenstandsmenge
  - CLIQUE: alle Mengen mit  $k$  Knoten
- Lösungen sind:
  - SAT: Wahrheitsbelegungen, die alle Klauseln wahr machen
  - COL: Färbungen, die benachbarte Knoten verschieden färben
  - CLIQUE: Mengen von  $k$  Knoten, die alle paarweise miteinander durch Kanten verbunden sind

## Polynomielle Lösungskandidaten: NP (1/2)

- Die betrachteten Entscheidungsprobleme haben folgende Eigenschaften:
  - (1) Sie haben einen Suchraum von Lösungskandidaten
  - (2) Die Lösungskandidaten sind polynomiell groß in der Eingabe
  - (3) Jeder einzelne Lösungskandidat kann in polynomieller Zeit überprüft werden
- Verständnisfrage: Hat PCP auch die Eigenschaften (1)-(3)?
- Wir verwenden die Eigenschaften (1) - (3) zur Definition der Komplexitätsklasse **NP**

## Polynomielle Lösungskandidaten: NP (2/2)

- Wir betrachten im Folgenden Turingmaschinen, die Paare  $(w, y)$  von Strings über  $\Sigma$  als Eingabe verarbeiten
  - Zur Erinnerung: formal erhält die TM also eine Eingabe der Form  $w\#y$  mit  $w, y \in \{0, 1\}^*$
- Die Rechenzeit von  $M$  bei Eingabe  $(w, y)$  bezeichnen wir mit  $t_M(w, y)$

### Definition

- Sei  $\Sigma = \{0, 1\}$ ,  $T : \mathbb{N} \rightarrow \mathbb{R}$
- $\text{NTIME}(T)$   $\stackrel{\text{def}}{=}$  Klasse aller  $L \subseteq \Sigma^*$ , für die es eine TM  $M$  gibt, so dass gelten:
  - (i) für alle  $w \in \Sigma^*$  sind äquivalent:
    - $w \in L$
    - es gibt eine **Zusatzeingabe**  $y \in \Sigma^*$ , so dass  $(w, y) \in L(M)$
  - \* Wir sagen dann:  $M$  akzeptiert  $w$  nichtdeterministisch
- (ii) für alle  $w, y \in \Sigma^*$  gilt:
$$t_M(w, y) \leq T(|w|)$$
- Sprechweise:
  - $M$  **entscheidet** die Sprache  $L$  **nicht-deterministisch** mit Zeitschranke  $T$

- $\text{NP}$   $\stackrel{\text{def}}{=}$   $\bigcup_{p \text{ Polynom}} \text{NTIME}(p)$



## Bemerkungen: NP (1/2)

- Die in der Definition von **NP** verwendete Zusatzeingabe  $y$  entspricht gerade den **Lösungskandidaten** in den betrachteten schwierigen Entscheidungsproblemen
- Es gelten also:
  - SAT, 3-SAT  $\in$  **NP**
    - \* Zusatzeingabe: Wahrheitsbelegung der in der gegebenen Formel vorkommenden Variablen
  - 3-COL, COL  $\in$  **NP**
    - \* Zusatzeingabe: Knotenfärbung
  - HAMILTONCYCLE, TSP  $\in$  **NP**
    - \* Zusatzeingabe: Knotenfolge bzw. „Reisefunktion“
  - KNAPSACK  $\in$  **NP**
    - \* Zusatzeingabe: Menge von Gegenständen
  - CLIQUE  $\in$  **NP**
    - \* Zusatzeingabe: Menge von Knoten
- Zu beachten: bei der Definition von **NTIME**( $T$ ) hängt die Zeitschranke nur von der (Länge der) **Eingabe**  $w$ , aber **nicht von der Zusatzeingabe**  $y$  ab
  - ➡ Es genügt also, Zusatzeingaben der Länge  $\leq T(|w|)$  zu betrachten, da die TM mehr Zeichen der Zusatzeingabe gar nicht lesen kann

## Bemerkungen: NP (2/2)

- Oft wird zur Definition der Klasse **NP** das Berechnungsmodell **nichtdeterministischer Turingmaschinen** verwendet
- Nichtdeterministische TMs können, wie NFAs, in derselben Situation (Zustand, gelesenes Zeichen) mehrere Transitionen haben
- Die nichtdeterministische Vorgehensweise solcher NTMs lässt sich intuitiv als „Raten“ auffassen:
  - Wenn  $w \in L$ , dann gibt es eine Berechnung der NTM, die „richtig rät“ und akzeptiert
  - Wenn  $w \notin L$ , dann lehnt die NTM ab, unabhängig davon, was sie rät
- Die Zusatzeingabe in unserer Definition entspricht dem „Geratenen“ in der NTM-Definition
- Die hier gegebene Definition von **NP** betont stärker den Aspekt des Überprüfens von Lösungskandidaten polynomieller Größe und vermeidet „ratende Algorithmen“

## Bemerkungen: P vs. NP

- Lösungen finden:

|   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|
|   | 6 |   | 2 |   |   | 7 |   |   |
|   |   |   |   |   | 4 |   |   | 9 |
| 2 |   | 3 |   |   |   | 5 |   |   |
|   | 2 |   |   | 3 |   |   |   | 8 |
|   |   |   | 4 |   | 7 |   |   |   |
| 1 |   |   |   | 8 |   |   | 6 |   |
|   |   | 5 |   |   |   | 8 |   | 4 |
| 9 |   |   | 1 |   |   |   |   |   |
|   |   | 4 |   |   | 2 |   | 9 |   |

- Damit ein Problem in **P** ist, muss es einen Algorithmus geben, der effizient eine Lösung **findet**
- Zum Beispiel ist HORNSAT  $\in$  **P**: der Markierungsalgorithmus findet effizient eine erfüllende Belegung, wenn es eine gibt

- Lösungskandidaten überprüfen:

|   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|
| 4 | 6 | 8 | 2 | 9 | 5 | 7 | 3 | 1 |
| 7 | 5 | 1 | 3 | 6 | 4 | 2 | 8 | 9 |
| 2 | 9 | 3 | 7 | 1 | 8 | 5 | 4 | 6 |
| 5 | 2 | 9 | 6 | 3 | 1 | 4 | 7 | 8 |
| 8 | 3 | 6 | 4 | 2 | 7 | 9 | 1 | 5 |
| 1 | 4 | 7 | 5 | 8 | 9 | 3 | 6 | 2 |
| 3 | 1 | 5 | 9 | 7 | 6 | 8 | 2 | 4 |
| 9 | 8 | 2 | 1 | 4 | 3 | 6 | 5 | 7 |
| 6 | 7 | 4 | 8 | 5 | 2 | 1 | 9 | 3 |

- Damit ein Problem in **NP** ist, genügt ein Algorithmus, der effizient **überprüft**, ob ein Lösungskandidat eine Lösung ist
- SAT  $\in$  **NP**: es kann effizient getestet werden, ob eine gegebene Belegung die gegebene Formel wahr macht

# P vs NP vs. EXPTIME

## Proposition 18.1

$$\mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{EXPTIME}$$

### Beweisskizze

- $\mathbf{P} \subseteq \mathbf{NP}$ : die Zusatzeingabe kann einfach ignoriert werden...
  - $\mathbf{NP} \subseteq \mathbf{EXPTIME}$ :
    - Sei  $M$  eine TM, die  $L$  nichtdeterministisch mit polynomieller Zeitschranke  $p$  entscheidet
    - **EXPTIME**-Algorithmus:
      - \* Simuliere  $M$  für alle möglichen Zusatzeingaben  $y$  der Länge  $\leq p(|w|)$
- ➔ maximal  $2^{p(|w|)}$  Teilberechnungen zu je  $\leq p(|w|)$  Schritten

- Ob auch die Umkehrung ( $\mathbf{NP} \subseteq \mathbf{P}$ ) der ersten Aussage gilt, ist das größte offene Problem der Theoretischen Informatik
  - Falls sie gilt, lassen sich alle in diesem Kapitel betrachteten Probleme (und viele tausende weiterer Probleme) in polynomieller Zeit lösen

# Inhalt

18.1 Beispiele schwieriger Berechnungsprobleme

18.2 **NP**

▷ **18.3 NP-Vollständigkeit**

# Ähnlichkeit schwieriger Optimierungsprobleme

- Wir werden sehen: die genannten Probleme sind sich noch viel ähnlicher:
  - entweder lassen sie sich alle in polynomieller Zeit lösen oder keines
- Wir betrachten diesen Zusammenhang zunächst anhand von SAT und der folgenden Einschränkung von COL:

## Definition: 3-COL

**Gegeben:** Ungerichteter Graph  $G$

**Frage:** Lassen sich die Knoten von  $G$  mit 3 Farben zulässig färben?

## Proposition 18.2

- Die folgenden Aussagen sind äquivalent:
  - (a) SAT lässt sich in polynomieller Zeit lösen
  - (b) 3-COL lässt sich in polynomieller Zeit lösen

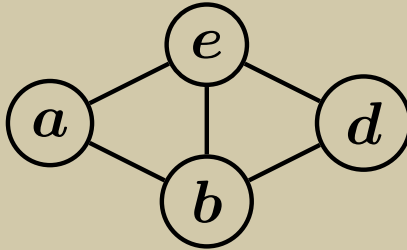
## Beweisskizze

- Wir zeigen hier nur: „(a)  $\Rightarrow$  (b)“
  - „(b)  $\Rightarrow$  (a)“ zeigen wir dann in Kapitel 19
- Wir nehmen an,  $A$  ist ein Algorithmus, der in Zeit  $|\varphi|^k$  entscheidet, ob eine gegebene KNF-Formel  $\varphi$  erfüllbar ist
- Wir beschreiben ein Verfahren, das unter Verwendung von  $A$  in polynomieller Zeit entscheidet, ob ein gegebener Graph  $G$  3-färbbar ist

## Beweis von Proposition 18.2: „(a) $\Rightarrow$ (b)“ (1/4)

### Illustration des Beweises

- Sei  $G$  der folgende Graph:



- $\varphi_G =$   
 $(x_{a1} \vee x_{a2} \vee x_{a3}) \wedge$   
 $(x_{b1} \vee x_{b2} \vee x_{b3}) \wedge$   
 $(x_{d1} \vee x_{d2} \vee x_{d3}) \wedge$   
 $(x_{e1} \vee x_{e2} \vee x_{e3}) \wedge$   
 $(\neg x_{a1} \vee \neg x_{b1}) \wedge (\neg x_{a2} \vee \neg x_{b2}) \wedge (\neg x_{a3} \vee \neg x_{b3}) \wedge$   
 $(\neg x_{a1} \vee \neg x_{e1}) \wedge (\neg x_{a2} \vee \neg x_{e2}) \wedge (\neg x_{a3} \vee \neg x_{e3}) \wedge$   
 $(\neg x_{b1} \vee \neg x_{d1}) \wedge (\neg x_{b2} \vee \neg x_{d2}) \wedge (\neg x_{b3} \vee \neg x_{d3}) \wedge$   
 $(\neg x_{b1} \vee \neg x_{e1}) \wedge (\neg x_{b2} \vee \neg x_{e2}) \wedge (\neg x_{b3} \vee \neg x_{e3}) \wedge$   
 $(\neg x_{d1} \vee \neg x_{e1}) \wedge (\neg x_{d2} \vee \neg x_{e2}) \wedge (\neg x_{d3} \vee \neg x_{e3})$

- Zulässige Färbung:  $c(a) = c(d) = 1, c(b) = 2, c(e) = 3$
- Korrespondierende Wahrheitsbelegung:
  - $\alpha(x_{a1}) = \alpha(x_{b2}) = \alpha(x_{d1}) = \alpha(x_{e3}) = 1$
  - $\alpha(x) = 0$  für alle übrigen Variablen  $x$


## Beweis von Proposition 18.2: „(a) $\Rightarrow$ (b)“ (2/4)

### Beweisskizze (Forts.)

- Sei also  $G = (V, E)$  eine Eingabe für 3-COL
- Wir konstruieren eine Formel  $\varphi_G$ , so dass gilt:  
 $\varphi_G$  erfüllbar  $\iff G$  3-färbbar  $(*)$
- $\varphi_G$  hat für jeden Knoten  $v \in V$  und jedes  $j \in \{1, 2, 3\}$  eine Variable  $x_{vj}$
- **Intention:**
  - Die Farben heißen 1, 2, 3
  - $\alpha(x_{vj}) = 1 \iff$   
 $\alpha$  repräsentiert eine Färbung, in der Knoten  $v$  die Farbe  $j$  hat

### Beweisskizze (Forts.)

- Wir verwenden folgende Teilformeln:
  - $\psi_v \stackrel{\text{def}}{=} (x_{v1} \vee x_{v2} \vee x_{v3})$   
 $\Rightarrow v$  hat (mindestens) eine Farbe
  - $\chi_{uv} \stackrel{\text{def}}{=} (\neg x_{u1} \vee \neg x_{v1}) \wedge (\neg x_{u2} \vee \neg x_{v2}) \wedge (\neg x_{u3} \vee \neg x_{v3})$   
 $\Rightarrow u$  und  $v$  haben verschiedene Farben
- Sei schließlich
 
$$\varphi_G \stackrel{\text{def}}{=} \bigwedge_{v \in V} \psi_v \wedge \bigwedge_{(u,v) \in E} \chi_{uv}$$

 Für jede ungerichtete Kante  $(u, v)$  verwenden wir nur die Formel  $\chi_{uv}$  oder die Formel  $\chi_{vu}$ , aber nicht beide



## Beweis von Proposition 18.2: „(a) $\Rightarrow$ (b)“ (3/4)

### Beweisskizze (Forts.)

- Zu zeigen:  $\varphi_G$  erfüllbar  $\iff G$  3-färbbar
- „ $\Leftarrow$ “:
  - Sei  $c : V \rightarrow \{1, 2, 3\}$  eine zulässige Färbung und
$$\alpha(x_{vi}) \stackrel{\text{def}}{=} \begin{cases} 1 & \text{falls } c(v) = i \\ 0 & \text{sonst} \end{cases}$$
  - Behauptung:  $\alpha \models \varphi_G$
  - Denn:
    - \*  $\alpha \models \psi_v$  für jeden Knoten  $v$  und
    - \*  $\alpha \models \chi_{uv}$  für jedes  $(u, v) \in E$ ,  
da  $c(u) \neq c(v)$

### Beweisskizze (Forts.)

- „ $\Rightarrow$ “:
  - Sei  $\alpha$  erfüllende Belegung von  $\varphi_G$
  - ➡ Für jedes  $v$  gilt  $\alpha \models \psi_v$
  - Sei  $c(v) \stackrel{\text{def}}{=} \text{kleinstes } j \text{ mit}$ 
$$\alpha(x_{vj}) = 1$$
  - Behauptung:  $c$  ist zulässige Färbung
  - Denn: wenn  $(u, v) \in E$ ,  
dann  $\alpha \models \chi_{uv}$
  - Also muss  $c(u) \neq c(v)$  sein

## Beweis von Proposition 18.2: „(a) $\Rightarrow$ (b)“ (4/4)

### Beweisskizze (Forts.)

- Zu beachten: Zu einem gegebenen  $G = (V, E)$  hat  $\varphi_G$  genau  $|V| + 3|E|$  Klauseln
- Ist  $n$  die Größe der Kodierung von  $G$  (beispielsweise durch Adjazenzlisten), so ist also  $|\varphi_G| \leq cn$ , für eine Konstante  $c$
- Offensichtlich kann  $\varphi_G$  in (polynomieller) Zeit  $n^\ell$ , für ein geeignetes  $\ell$ , berechnet werden
- Also kann 3-COL wie folgt gelöst werden:
  - Bei Eingabe  $G$  berechne  $\varphi_G$
  - Teste mit Hilfe von Algorithmus  $A$ , ob  $\varphi_G$  erfüllbar ist
- Die Gesamtlaufzeit ist  $\leq n^\ell + (cn)^k = \mathcal{O}(n^{\max(\ell, k)})$

➡ Behauptung

- Um Proposition 18.2 zu beweisen, hätte es genügt zu zeigen, dass gilt:  
 $\varphi_G$  erfüllbar  $\iff G$  3-färbbar
- Der Beweis zeigt mehr als das:
  - er konstruiert aus jeder erfüllenden Belegung von  $\varphi_G$  eine korrekte 3-Färbung von  $G$  und umgekehrt
- Das ist für diese Art von Beweisen typisch

# Polynomielle Reduktionen

- Schauen wir nochmal auf den Beweis von Proposition 18.2 „(a)  $\Rightarrow$  (b)“

- Sei  $f$  die Funktion  $G \mapsto \varphi_G$

- Dann gilt:

$$G \in 3\text{-COL} \iff f(G) \in \text{SAT}$$

- ➔  $f$  ist eine Reduktion von 3-COL auf SAT
  - Also:  $3\text{-COL} \leq \text{SAT}$

- Und:  $f$  ist in polynomieller Zeit berechenbar

## Definition

- Eine totale Funktion  $f$  heißt polynomielle Reduktion von  $L$  auf  $L'$ , falls
  - (1)  $f$  eine Reduktion ist, also für alle  $w \in \Sigma^*$  gilt:
$$w \in L \iff f(w) \in L', \text{ und}$$
  - (2)  $f$  in polynomieller Zeit berechenbar ist
- $L$  heißt polynomiell reduzierbar auf  $L'$ , falls es eine polynomielle Reduktion von  $L$  auf  $L'$  gibt
- Schreibweise:  $L \leq_p L'$

- Also:  $3\text{-COL} \leq_p \text{SAT}$

# Abschluss unter Reduktionen

## Proposition 18.3

- Seien  $L, L' \subseteq \Sigma^*$  und gelte:  $L \leq_p L'$
- Dann gelten:
  - (a) wenn  $L' \in \mathbf{P}$  dann  $L \in \mathbf{P}$
  - (b) wenn  $L' \in \mathbf{NP}$  dann  $L \in \mathbf{NP}$

## Beweisskizze für (a)

- Sei  $M'$  TM, die  $L'$  mit Zeitschranke  $n^k$  entscheidet
- Sei  $M_f$  eine TM, die mit Zeitschranke  $n^\ell$  eine Reduktion  $f$  berechnet, so dass gilt:
 
$$w \in L \iff f(w) \in L'$$

- Wir konstruieren eine TM  $M$ , die  $L$  entscheidet, indem sie bei Eingabe  $w$  wie folgt vorgeht:  $\oplus$ 
  - Berechne  $f(w)$   $\Rightarrow$  durch TM  $M_f$
  - Teste  $f(w) \in L'$   $\Rightarrow$  durch TM  $M'$
  - Falls ja, akzeptiere, falls nein, lehne ab
- Laufzeit:  $\leq |w|^\ell + |f(w)|^k \leq |w|^\ell + |w|^{k\ell}$
- ➡  $\Rightarrow L \in \mathbf{P}$

## Beweisskizze für (b)

- Der Beweis für (b) ist eine Erweiterung des Beweises von (a)
- Wir nehmen an, dass  $M'$  die Sprache  $L'$  nichtdeterministisch entscheidet
- Dann hat  $M$  ebenfalls eine Zusatzeingabe  $y$  und testet in der zweiten Phase, ob  $(f(w), y) \in L'$  ist
- Der Rest ist analog

## Folgerung 18.4

- Seien  $L, L' \subseteq \Sigma^*$  und gelte  $L \leq_p L'$
- ➡ Wenn  $L \notin \mathbf{P}$  dann  $L' \notin \mathbf{P}$

# NP-schwierige und NP-vollständige Probleme

## Definition

- Eine Sprache  $L$  heißt **NP-schwierig**, falls für alle  $L' \in \mathbf{NP}$  gilt:  $L' \leq_p L$

- Gibt es überhaupt **NP-schwierige** Probleme?

## Proposition 18.5

- TM-HALT ist **NP-schwierig**

## Beweisskizze

- Sei  $L \in \mathbf{NP}$
- ➔  $L \in \mathbf{EXPTIME}$
- Sei  $M$  TM, die  $L$  in exponentieller Zeit entscheidet
- Sei  $M'$  eine TM, die  $M$  bei Eingabe  $w$  simuliert und
  - akzeptiert, wenn  $M(w)$  akzeptiert, aber
  - endlos läuft, wenn  $M(w)$  ablehnt
- Sei  $f(w) \stackrel{\text{def}}{=} (M', w)$
- ➔  $f$  ist eine polynomielle Reduktion von  $L$  auf TM-HALT
- ➔ Behauptung

- Das zeigt zwar, dass es **NP-schwierige** Probleme gibt, aber das Halteproblem ist für unsere jetzigen Zwecke nicht interessant...
- Interessanter wären **NP-schwierige** Probleme **innerhalb von NP**

## Definition

- Eine Sprache  $L$  heißt **NP-vollständig**, falls  $L$  **NP-schwierig** ist und  $L \in \mathbf{NP}$  gilt
- Die **NP-vollständigen** Probleme sind also gewissermaßen die schwierigsten Probleme in **NP**


# NP-Vollständigkeit: Bedeutung

## Satz 18.6

- Sei  $L$  eine **NP**-vollständige Sprache
  - (a) Falls  $L \in \mathbf{P}$ , so ist  $\mathbf{P} = \mathbf{NP}$ 
    - insbesondere sind dann auch alle anderen **NP**-vollständigen Sprachen in  $\mathbf{P}$
  - (b) Falls  $L \notin \mathbf{P}$ , so ist  $\mathbf{P} \neq \mathbf{NP}$ 
    - und alle anderen **NP**-vollständigen Sprachen sind auch nicht in  $\mathbf{P}$

## Beweisskizze

- (a) • Sei  $L' \in \mathbf{NP}$ 
  - Da  $L$  **NP**-vollständig ist, gilt  $L' \leq_p L$
  - ➔  $L' \in \mathbf{P}$  (gemäß Proposition 18.3)
- (b) •  $L \notin \mathbf{P} \Rightarrow \mathbf{P} \neq \mathbf{NP}$  ist trivial
  - Dass dann alle anderen **NP**-vollständigen Sprachen auch nicht in  $\mathbf{P}$  sind, folgt aus (a)

- Die „Schicksal“ der „**P** vs. **NP**“-Frage hängt also an jedem einzelnen **NP**-vollständigen Problem
- Es stellt sich aber die Frage:
  - Gibt es überhaupt **NP**-vollständige Probleme?  Nächstes Kapitel

# Zusammenfassung

- Wir haben einige Berechnungsprobleme kennen gelernt, für die keine Algorithmen mit polynomieller Laufzeit bekannt sind
- **NP** ist die Klasse der Probleme, für die sich in polynomieller Zeit testen lässt, ob ein gegebener Lösungskandidat eine Lösung ist
- Die **NP**-vollständigen Probleme sind die schwierigsten Probleme in **NP**:
  - Jedes **NP**-Problem lässt sich polynomiell auf jedes **NP**-vollständige Problem reduzieren
- Ob  $P = NP$  ist, ist die größte offene Frage der Theoretischen Informatik

# Grundbegriffe der Theoretischen Informatik

Sommersemester 2016 - Thomas Schwentick

Teil D: Komplexitätstheorie

19: NP-vollständige Probleme

Version von: 5. Juli 2016 (12:11)



## Ein erstes NP-vollständiges Problem

- In Teil C der Vorlesung haben wir für die Sprache TM-DIAG die Unentscheidbarkeit durch Diagonalisierung bewiesen und dann alle weiteren unentscheidbaren Probleme durch (direkte oder indirekte) Reduktionen von TM-DIAG gewonnen
- Hinsichtlich **NP**-vollständiger Probleme gehen wir ähnlich vor:
  - Wir zeigen als nächstes, dass das Problem SAT **NP**-vollständig ist
    - \* Das ist die Aussage des *Satzes von Cook*
  - Für weitere Probleme weisen wir die **NP**-Vollständigkeit dann durch (direkte oder indirekte) polynomielle Reduktionen von SAT nach

# Inhalt

## ▷ 19.1 Der Satz von Cook

19.2 Polynomielle Reduktionen

19.3 3-SAT

19.4 Das Cliques-Problem

19.5 3-Färbbarkeit

19.6 Hamiltonkreise und TSP

19.7 Teilsummen und das Rucksack-Problem

# Satz von Cook (1/13)

Satz 19.1 [Cook 71]

SAT ist **NP**-vollständig

Beweisskizze

- SAT  $\in$  **NP**: ✓
- SAT ist **NP**-schwierig:
  - Wir zeigen, dass für jedes  $L \in \mathbf{NP}$  gilt:  $L \leq_p \text{SAT}$
- Sei  $L$  dazu eine beliebige Sprache aus **NP**
- Sei  $M = (Q, \Gamma, \delta, q_1)$  eine (1-String)-TM, die  $L$  mit polynomieller Zeitschranke  $n^k$  nichtdeterministisch entscheidet
- Sei  $w$  eine Eingabe für  $M$  und  $n$  die Länge von  $w$

Beweisskizze (Forts.)

- Wir zeigen, wie aus  $w$  eine KNF-Formel  $\varphi$  konstruiert werden kann, so dass gilt:  
 $M$  akzeptiert  $w$  nichtdeterministisch  $\iff \varphi$  ist erfüllbar
- Genauer: wir zeigen, dass es zu jeder Zusatzeingabe  $y$ , für die  $w$  von  $M$  akzeptiert wird, eine erfüllende Belegung  $\alpha$  für  $\varphi$  gibt, und umgekehrt
- **Wichtige Idee:**
  - Wir betrachten die Berechnung von  $M$  bei Eingabe  $w$  als eine **Berechnungstabelle**
    - \* Die Variablen von  $\varphi$  entsprechen den einzelnen Einträgen der Tabelle
    - \* Jede Variablenbelegung  $\alpha$  der Variablen von  $\varphi$  entspricht einer „ausgefüllten“ Tabelle
  - $\varphi$  soll genau dann wahr werden, wenn  $\alpha$  eine akzeptierende Berechnung repräsentiert

## Satz von Cook (2/13): Beispiel-TM

### Beispiel

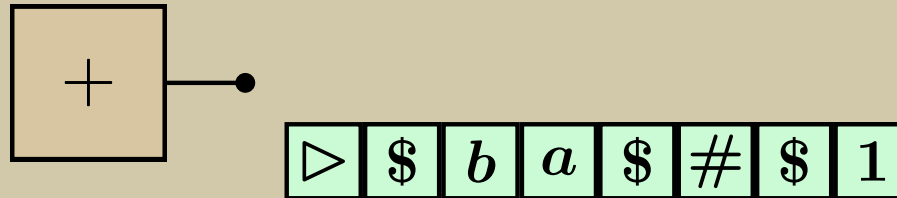
- Wie betrachten als Beispiel eine TM  $M$ , die die Sprache  $L$  aller Strings über  $\{a, b\}$ , die **kein Palindrom** sind, akzeptiert
- $M$  erwartet Eingaben der Art  $w\#y$ 
  - $w$  ist der zu überprüfende String  $\text{☞ } n \stackrel{\text{def}}{=} |w|$
  - $y$  ist eine Zusatzeingabe der Art  $0^k 1$
- $M$  akzeptiert genau dann, wenn die  $(k+1)$ -te Position von  $w$  von der  $(n-k)$ -ten Position verschieden ist

### Bemerkung

- $L$  ist in **NP**, aber natürlich auch in **P**
- Wir verwenden ein so einfaches Beispiel, weil es sich detailliert auf einer Folie unterbringen lässt

# Satz von Cook (3/13): Beispiel-Berechnung

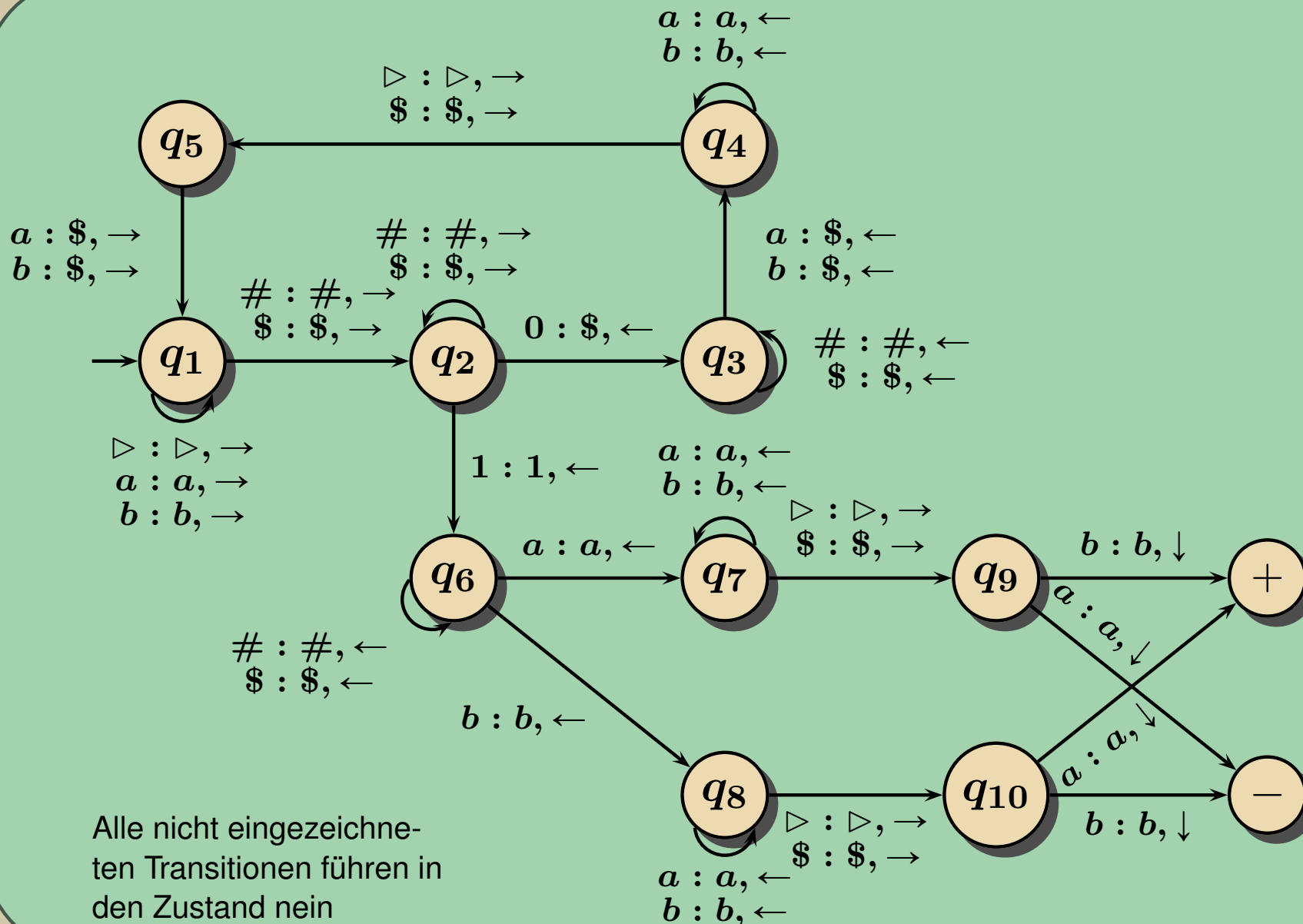
## Beispiel



- $M$  erwartet Eingaben der Art  $w\#y$
- $M$  bewegt den Kopf zuerst nach rechts auf das erste Zeichen von  $y$
- Falls dort eine  $0$  steht, wird sie durch  $\$$  überschrieben und  $M$  läuft nach links und überschreibt dabei das letzte und dann das erste Zeichen von  $w$  mit  $\$$
- Dann läuft  $M$  wieder nach rechts zum nächsten Zeichen von  $y$
- Wenn dort eine  $0$  steht, werden wieder das letzte und erste noch nicht veränderte Zeichen von  $w$  mit  $\$$  überschrieben
- Wenn dort eine  $1$  steht, werden das letzte und erste noch nicht veränderte Zeichen von  $w$  verglichen
- Sind diese beiden Zeichen verschieden, so akzeptiert  $M$ , andernfalls lehnt  $M$  ab

# Satz von Cook (4/13): Beispiel-TM als Diagramm

## Beispiel



# Satz von Cook (5/13): Beispiel einer Berechnungstabelle

Beispiel: Tabelle für 1. TM-Beispiel

| $t$ | $B$                         | $Z$   | $P$ |
|-----|-----------------------------|-------|-----|
| 0   | $\triangleright abaa\#01$   | $q_1$ | 0   |
| 1   | $\triangleright abaa\#01$   | $q_1$ | 1   |
| 2   | $\triangleright abaa\#01$   | $q_1$ | 2   |
| 3   | $\triangleright abaa\#01$   | $q_1$ | 3   |
| 4   | $\triangleright abaa\#01$   | $q_1$ | 4   |
| 5   | $\triangleright abaa\#01$   | $q_1$ | 5   |
| 6   | $\triangleright abaa\#01$   | $q_2$ | 6   |
| 7   | $\triangleright abaa\#\$1$  | $q_3$ | 5   |
| 8   | $\triangleright abaa\#\$1$  | $q_3$ | 4   |
| 9   | $\triangleright aba\$\#\$1$ | $q_4$ | 3   |
| 10  | $\triangleright aba\$\#\$1$ | $q_4$ | 2   |
| 11  | $\triangleright aba\$\#\$1$ | $q_4$ | 1   |
| 12  | $\triangleright aba\$\#\$1$ | $q_4$ | 0   |
| 13  | $\triangleright aba\$\#\$1$ | $q_5$ | 1   |

Beispiel (Forts.)

| $t$ | $B$                          | $Z$   | $P$ |
|-----|------------------------------|-------|-----|
| 14  | $\triangleright \$ba\$\#\$1$ | $q_1$ | 2   |
| 15  | $\triangleright \$ba\$\#\$1$ | $q_1$ | 3   |
| 16  | $\triangleright \$ba\$\#\$1$ | $q_1$ | 4   |
| 17  | $\triangleright \$ba\$\#\$1$ | $q_2$ | 5   |
| 18  | $\triangleright \$ba\$\#\$1$ | $q_2$ | 6   |
| 19  | $\triangleright \$ba\$\#\$1$ | $q_2$ | 7   |
| 20  | $\triangleright \$ba\$\#\$1$ | $q_6$ | 6   |
| 21  | $\triangleright \$ba\$\#\$1$ | $q_6$ | 5   |
| 22  | $\triangleright \$ba\$\#\$1$ | $q_6$ | 4   |
| 23  | $\triangleright \$ba\$\#\$1$ | $q_6$ | 3   |
| 24  | $\triangleright \$ba\$\#\$1$ | $q_7$ | 2   |
| 25  | $\triangleright \$ba\$\#\$1$ | $q_7$ | 1   |
| 26  | $\triangleright \$ba\$\#\$1$ | $q_9$ | 2   |
| 27  | $\triangleright \$ba\$\#\$1$ | ja    | 2   |

# Satz von Cook (6/13): Variablen von $\varphi$

## Beweisskizze (Forts.)

- Zur Vereinfachung nehmen wir oBdA an, dass die Zusatzeingabe genau die Länge  $n^k - n - 1$  hat und nur die Zeichen **0** und **1** verwendet
- Es gilt also:
  - $w\#y$  hat genau  $n^k$  Zeichen
  - Linker Rand: Position **0**
  - $w$ : Positionen  $1, \dots, n$
  - $\#$  an Position  $n + 1$
  - $y$ : Positionen  $n + 2, \dots, n^k$
- Klar: in  $n^k$  Schritten kann sich der Kopf der Turing-Maschine nicht über  $y$  hinaus bewegen
- Sei  $\Gamma = \{\sigma_1, \dots, \sigma_l\}$  das Arbeitssalphabet von  $M$
- Sei  $Q = \{q_1, \dots, q_m\}$  die Zustandsmenge von  $M$  inklusive ja, nein und  $h$

## Beweisskizze (Forts.)

- Die Formel  $\varphi$  verwendet die folgenden aussagenlogischen Variablen:

| Variablen Indizes |   | Intendierte Bedeutung   |
|-------------------|---|---|
| $Z_{t,q}$         | $t = 0, \dots, n^k$<br>$q \in Q$                                  | $\alpha(Z_{t,q}) = 1 \iff$<br>nach $t$ Schritten befindet sich $M$ im Zustand $q$                           |
| $P_{t,i}$         | $t = 0, \dots, n^k$<br>$i = 0, \dots, n^k$                        | $\alpha(P_{t,i}) = 1 \iff$<br>nach $t$ Schritten befindet sich der Kopf von $M$ auf Position $i$            |
| $B_{t,i,\sigma}$  | $t = 0, \dots, n^k$<br>$i = 0, \dots, n^k$<br>$\sigma \in \Gamma$ | $\alpha(B_{t,i,\sigma}) = 1 \iff$<br>nach $t$ Schritten befindet sich auf Position $i$ das Zeichen $\sigma$ |




# Satz von Cook (7/13): Belegung der Variablen

## Beispiel

- Wir werfen einen Blick auf die Variablenbelegung  $\alpha$ , die der Berechnungstabelle der Beispielberechnung entspricht
- Wir betrachten nur die Variablen, die die Konfiguration der TM nach 10 Schritten repräsentieren:
  - $t = 10, Z = q_4, P = 2$
  - $B = \triangleright aba\$ \# \$1$

## Bemerkung

-  Im Gegensatz zur Konstruktion im Beweis ist die Länge der Eingabe im Beispiel nicht gleich der Anzahl der Berechnungsschritte
- Deshalb werden hier, der allgemeinen Semantik von TMs entsprechend, hinter der Eingabe Blanks ergänzt

## Beispiel (Forts.)

- $\alpha(Z_{10,q_4}) = 1$
- $\alpha(Z_{10,p}) = 0$ , für  $p \neq q_4$
- $\alpha(P_{10,2}) = 1$
- $\alpha(P_{10,i}) = 0$ , für  $i \neq 2$
- $\alpha(B_{10,0,\triangleright}) = 1$
- $\alpha(B_{10,1,a}) = 1$
- $\alpha(B_{10,2,b}) = 1$
- $\alpha(B_{10,3,a}) = 1$
- $\alpha(B_{10,4,\$}) = 1$
- $\alpha(B_{10,5,\#}) = 1$
- $\alpha(B_{10,6,\$}) = 1$
- $\alpha(B_{10,7,1}) = 1$
- $\alpha(B_{10,i,\sqcup}) = 1$ , für alle  $i > 7$
- $\alpha(B_{10,i,\sigma}) = 0$ , für alle übrigen  $i, \sigma$

# Satz von Cook (8/13): Konsistenzbedingungen

## Beweisskizze (Forts.)

- $\varphi$  ist aus mehreren Teilformeln zusammengesetzt:

$$\varphi = \varphi_K \wedge \varphi_A \wedge \varphi_D \wedge \varphi_E$$

- $\varphi_K$  soll sicherstellen, dass  $\alpha$  überhaupt eine Tabelle repräsentiert, also jeder Eintrag der Tabelle genau einmal vorhanden ist

☞ Konsistenzbedingungen

- $\varphi_A$  drückt aus, dass die erste Zeile der Tabelle der Startkonfiguration entspricht

☞ Anfangsbedingungen

- $\varphi_D$  drückt aus, dass aufeinander folgende Zeilen der Tabelle der Transitionsfunktion entsprechen

☞ Transitionsbedingungen

- $\varphi_E$  drückt aus, dass die Berechnung akzeptiert

☞ Endbedingung

# Satz von Cook (9/13): Konsistenzbedingungen

## Beweisskizze (Forts.)

- $\varphi_K$  drückt für die von  $\alpha$  kodierte Tabelle aus:
  - Zu jedem Zeitpunkt ist der Zustand von  $M$  eindeutig
  - Zu jedem Zeitpunkt ist die Position des Kopfes von  $M$  eindeutig bestimmt
  - Zu jedem Zeitpunkt steht an jeder Stringposition genau ein Zeichen

## Beweisskizze (Forts.)

- Wir verwenden dabei die Hilfsformel

$$\psi_{\text{unique}}(x_1, \dots, x_s) \stackrel{\text{def}}{=} \left( \bigvee_{i=1}^s x_i \right) \wedge \left( \bigwedge_{i \neq j} (\neg x_i \vee \neg x_j) \right),$$

die wahr wird, wenn  $\alpha(x_i) = 1$ , für genau ein  $i$

- $\varphi_K \stackrel{\text{def}}{=} \bigwedge_t \psi_{\text{unique}}(Z_{t,q_1}, \dots, Z_{t,q_m}) \wedge \bigwedge_t \psi_{\text{unique}}(P_{t,0}, \dots, P_{t,n^k}) \wedge \bigwedge_{t,i} \psi_{\text{unique}}(B_{t,i,\sigma_1}, \dots, B_{t,i,\sigma_l})$

# Satz von Cook (10/13): Anfangsbedingungen

## Beweisskizze (Forts.)

- $\varphi_A$  beschreibt die Situation von  $M$  zum Zeitpunkt 0:

- $\varphi_A \stackrel{\text{def}}{=}$

$$Z_{0,q_1} \wedge P_{0,0} \wedge B_{0,0,\triangleright} \wedge \bigwedge_{i=1}^n B_{0,i,w[i]} \\ \wedge B_{0,n+1,\#} \wedge \bigwedge_{i=n+2}^{n^k} (B_{0,i,0} \vee B_{0,i,1})$$

- Zu beachten:
  - Die Formel drückt unter anderem aus, dass die Zusatzeingabe nur aus Nullen und Einsen besteht
  - Die Formel legt die Eingabe  $w$  fest
    - \* Dies ist die einzige Teilformel von  $\varphi$ , die wirklich von  $w$  abhängt
    - \* Die anderen Teilformeln hängen allenfalls von der Länge von  $w$  ab
  - Die Formel legt *nicht* die Zusatzeingabe  $y$  fest

# Satz von Cook (11/13): Transitionsbedingungen

## Beweisskizze (Forts.)

- $\varphi_D$  beschreibt die Beziehung zwischen den aufeinander folgenden Konfigurationen

- $\varphi_D \stackrel{\text{def}}{=} \varphi_{D_1} \wedge \varphi_{D_2}$ , wobei:
  - $\varphi_{D_1}$  beschreibt, was sich an der Stelle, an der sich der Kopf der Turing-Maschine befindet, ändert

- $\varphi_{D_1} \stackrel{\text{def}}{=} \bigwedge_{t,i,p,\sigma} [(Z_{t,p} \wedge P_{t,i} \wedge B_{t,i,\sigma}) \rightarrow (Z_{t+1,q} \wedge P_{t+1,i+d} \wedge B_{t+1,i,\tau})]$

- Dabei sind  $q, \tau, d$  jeweils durch  $\delta(p, \sigma) = (q, \tau, d)$  gegeben
  - $d$  wird hier als Zahl in  $\{-1, 0, 1\}$  interpretiert ( $\leftarrow \equiv -1, \downarrow \equiv 0, \rightarrow \equiv 1$ )

  $\varphi_{D_1}$  ist die einzige Teilformel von  $\varphi$ , die von der Transitionsfunktion  $\delta$  abhängt

## Beispiel

- Die Formel  $\varphi_{D_1}$  ist zu lang, um sie für das Beispiel ganz anzugeben
- Deshalb hier nur ein kleiner Ausschnitt für  $t = 8, i = 3, p = q_7$
- Es gilt in der TM
  - $\delta(q_7, a) = (q_7, a, \leftarrow)$
  - $\delta(q_7, b) = (q_7, b, \leftarrow)$
  - $\delta(q_7, \$) = (q_9, \$, \rightarrow)$

- Die entsprechende Teilformel lautet dann:
 
$$[(Z_{8,q_7} \wedge P_{8,3} \wedge B_{8,3,a}) \rightarrow (Z_{9,q_7} \wedge P_{9,2} \wedge B_{9,3,a})] \wedge [(Z_{8,q_7} \wedge P_{8,3} \wedge B_{8,3,b}) \rightarrow (Z_{9,q_7} \wedge P_{9,2} \wedge B_{9,3,b})] \wedge [(Z_{8,q_7} \wedge P_{8,3} \wedge B_{8,3,\$}) \rightarrow (Z_{9,q_9} \wedge P_{9,4} \wedge B_{9,3,\$})]$$

# Satz von Cook (12/13): Transitionsbedingungen (Forts.)

## Beweisskizze (Forts.)

- Die Teilformeln  

$$[(Z_{t,p} \wedge P_{t,i} \wedge B_{t,i,\sigma}) \rightarrow (Z_{t+1,q} \wedge P_{t+1,i+d} \wedge B_{t+1,i,\tau})]$$
 von  $\varphi_{D_1}$  sind noch nicht in KNF, lassen sich aber äquivalent umformen:
 
$$(\neg Z_{t,p} \vee \neg P_{t,i} \vee \neg B_{t,i,\sigma} \vee Z_{t+1,q}) \wedge (\neg Z_{t,p} \vee \neg P_{t,i} \vee \neg B_{t,i,\sigma} \vee P_{t+1,i+d}) \wedge (\neg Z_{t,p} \vee \neg P_{t,i} \vee \neg B_{t,i,\sigma} \vee B_{t+1,i,\tau})$$
- Ein technisches Detail: Was ist, wenn  $M$  weniger als  $n^k$  Schritte macht?
  - Dann wird in der Berechnungstabelle die Endkonfiguration ab der entsprechenden Zeile immer wiederholt
  - Die Formel wird analog gebildet (als wäre  $\delta(\text{ja}, \sigma) = (\text{ja}, \sigma, \downarrow)$ )

## Beweisskizze (Forts.)

- $\varphi_{D_2}$  drückt aus, dass sich der String an allen übrigen Positionen nicht verändert:

- $\varphi_{D_2} \stackrel{\text{def}}{=} \bigwedge_{t,i,\sigma} ((\neg P_{t,i} \wedge B_{t,i,\sigma}) \rightarrow B_{t+1,i,\sigma})$
- In konjunktiver Normalform:  

$$\varphi_{D_2} \stackrel{\text{def}}{=} \bigwedge_{t,i,\sigma} (P_{t,i} \vee \neg B_{t,i,\sigma} \vee B_{t+1,i,\sigma})$$

## Beispiel

- Für  $t = 8$  und  $i = 3$  ergibt sich also:

$$(P_{8,3} \vee \neg B_{8,3,a} \vee B_{9,3,a}) \wedge (P_{8,3} \vee \neg B_{8,3,b} \vee B_{9,3,b}) \wedge \dots \wedge (P_{8,3} \vee \neg B_{8,3,\$} \vee B_{9,3,\$}) \wedge (P_{8,3} \vee \neg B_{8,3,\triangleright} \vee B_{9,3,\triangleright})$$

# Satz von Cook (13/13): Abschluss des Beweises

## Beweisskizze (Forts.)

- Endbedingung:
  - $\varphi_E$  drückt aus, dass die letzte Konfiguration den ja-Zustand hat:

$$\varphi_E = Z_{n^k, \text{ja}}$$

- **Behauptung: Die Größe von  $\varphi$  ist polynomiell in  $n$ :**

- Größe der einzelnen Teilformeln:

|             |                       |
|-------------|-----------------------|
| $\varphi_K$ | $\mathcal{O}(n^{3k})$ |
| $\varphi_A$ | $\mathcal{O}(n^k)$    |
| $\varphi_D$ | $\mathcal{O}(n^{2k})$ |
| $\varphi_E$ | $\mathcal{O}(1)$      |

  $|\psi_{\text{unique}}(x_1, \dots, x_s)| = \mathcal{O}(s^2)$

- Zu beachten:  $m$  und  $l$  sind durch  $M$  bestimmt und damit konstant
- ➡ die Größe von  $\varphi$  ist polynomiell in  $n$

- Und: alle Teilformeln sind in konjunktiver Normalform, also auch ihre Konjunktion

## Beweisskizze (Forts.)

- Noch zu zeigen:  $w \in L \iff \varphi$  erfüllbar
- Also: Zu  $w$  gibt es genau dann eine Zusatzeingabe  $y$ , die  $M$  zum Akzeptieren bringt, wenn  $\varphi$  erfüllbar ist

- Falls es ein solches  $y$  gibt, können alle Variablen gemäß ihrer intendierten Bedeutung mit Wahrheitswerten belegt werden

➡  $\varphi$  wird wahr

- Umgekehrt: Zu jeder erfüllenden Belegung  $\alpha$  von  $\varphi$  lässt sich eine Zusatzeingabe  $y$  konstruieren und eine akzeptierende Berechnung von  $M$  bei Eingabe  $w$  und Zusatzeingabe  $y$  konstruieren

➡  $w \in L$

- Damit ist der Beweis des Satzes von Cook vollendet

# Inhalt

19.1 Der Satz von Cook

▷ **19.2 Polynomielle Reduktionen**

19.3 3-SAT

19.4 Das Cliques-Problem

19.5 3-Färbbarkeit

19.6 Hamiltonkreise und TSP

19.7 Teilsummen und das Rucksack-Problem



# Einleitung

- Wir wissen jetzt also: SAT ist **NP**-vollständig
- Ihre große Bedeutung hat die **NP**-Vollständigkeit erst durch den Nachweis erlangt, dass viele andere algorithmische Probleme **NP**-vollständig sind
- Die erste größere Menge solcher Probleme wurde von Karp 1972 vorgestellt
  - Die Arbeit enthält alle im Folgenden betrachteten Probleme
  - Die hier vorgestellten Beweise sind aber zum Teil anders

- Wie schon gesagt, werden wir ähnlich wie im Falle der unentscheidbaren Probleme in Teil C vorgehen:
  - Ausgehend von SAT zeigen wir die **NP**-Vollständigkeit der anderen Probleme jeweils mit Hilfe einer **einzelnen** polynomiellen Reduktion

- Zunächst vergewissern wir uns aber, dass dieser Ansatz wirklich funktioniert
- Dazu zeigen wir, dass wir wie folgt schließen können
  - Wenn alle **NP**-Probleme polynomiell reduzierbar auf eine Sprache  $L'$  sind
  - und  $L'$  polynomiell auf eine Sprache  $L$  reduzierbar ist
  - dann sind auch alle **NP**-Probleme polynomiell reduzierbar auf  $L$

- Wir müssen also zeigen, dass  $\leq_p$  eine transitive Relation ist

# Reduktionen und NP-Vollständigkeit (1/2)

## Lemma 19.2

- Seien  $L_1, L_2, L_3 \subseteq \Sigma^*$  Sprachen
- Falls  $L_1 \leq_p L_2$  und  $L_2 \leq_p L_3$ , so gilt auch  $L_1 \leq_p L_3$

## Beweisskizze

- Sei  $f_1$  eine Reduktion von  $L_1$  auf  $L_2$  und  $f_2$  eine Reduktion von  $L_2$  auf  $L_3$
- Behauptung: Die durch  $f(w) \stackrel{\text{def}}{=} f_2(f_1(w))$  definierte Funktion ist eine polynomielle Reduktion von  $L_1$  auf  $L_3$

## Beweisskizze (Forts.)

- $f$  ist eine Reduktion:
  - Für alle Strings  $w \in \Sigma^*$  gilt:
$$w \in L_1 \iff f_1(w) \in L_2 \iff f(w) = f_2(f_1(w)) \in L_3$$
- $f$  kann in polynomieller Zeit berechnet werden:
  - Seien  $n^i$  und  $n^j$  Zeitschranken für die Berechnung von  $f_1$  und  $f_2$
  - ➔ Dann kann  $f(w)$  in
$$|w|^i + |f_1(w)|^j \leq |w|^i + (|w|^i)^j = \mathcal{O}(|w|^{ij})$$
Schritten berechnet werden
- Die binäre Relation  $\leq_p$  zwischen Sprachen ist also transitiv

## Reduktionen und NP-Vollständigkeit (2/2)

- Die Möglichkeit des Nachweises der **NP**-Vollständigkeit durch eine einzelne Reduktion liefert nun das folgende Lemma

### Lemma 19.3

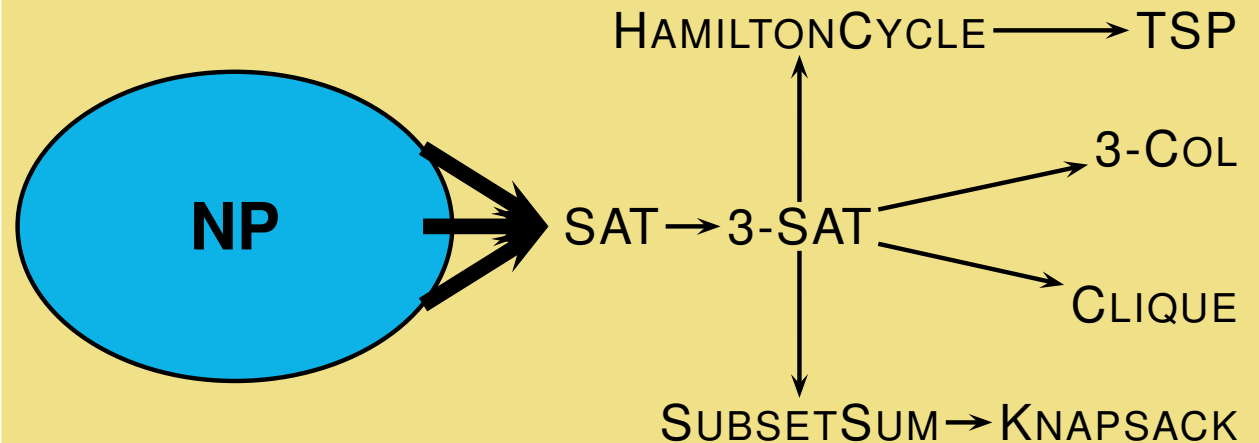
- Ist  $L'$  **NP**-schwierig und gilt  $L' \leq_p L$ , so ist auch  $L$  **NP**-schwierig

### Beweisskizze


- Sei  $L'' \in \mathbf{NP}$  beliebig
  - ➔  $L'' \leq_p L'$ 
    - ☞ da  $L'$  **NP**-schwierig
- Wegen  $L' \leq_p L$  folgt  $L'' \leq_p L$ 
  - ☞ gemäß Lemma 19.2
- ➔  $L$  **NP**-schwierig

- Um nachzuweisen, dass ein Problem  $L$  **NP**-vollständig ist, genügt es also zu zeigen:
  - $L \in \mathbf{NP}$
  - $L' \leq_p L$  für ein **NP**-vollständiges Problem  $L'$

- Die Reduktionen, die wir in diesem Kapitel betrachten werden, sind in der folgenden Abbildung zusammengefasst:



# Polynomielle Reduktionen: Rezept

- Die folgenden Beweise für Aussagen der Art  $L_1 \leq_p L_2$  verlaufen alle nach demselben Muster
  - Sie gehen in vier Schritten vor
- (1) Definiere eine Reduktionsfunktion  $f$ , die Eingaben für  $L_1$  auf Eingaben für  $L_2$  abbildet
  - (2) Zeige, dass  $f$  in polynomieller Zeit berechnet werden kann  
 Das ist meistens ziemlich offensichtlich
  - (3) Zeige: wenn  $w \in L_1$  dann ist auch  $f(w) \in L_2$ 
    - Beweise dazu, dass aus jeder Lösung  $y_1$  für  $w$  eine Lösung  $y_2$  für  $f(w)$  konstruiert werden kann
  - (4) Zeige: wenn  $f(w) \in L_2$  dann ist auch  $w \in L_1$ 
    - Beweise dazu, dass aus jeder Lösung  $y_2$  für  $f(w)$  eine Lösung  $y_1$  für  $w$  konstruiert werden kann
- Daraus folgt dann  $L_1 \leq_p L_2$

# Inhalt

19.1 Der Satz von Cook

19.2 Polynomielle Reduktionen

▷ **19.3 3-SAT**

19.4 Das Cliques-Problem

19.5 3-Färbbarkeit

19.6 Hamiltonkreise und TSP

19.7 Teilsummen und das Rucksack-Problem

# $SAT \leq_p 3\text{-SAT} (1/4)$

Proposition 19.4

$SAT \leq_p 3\text{-SAT}$

- Bei dieser Reduktion verwenden wir für lange Klauseln eine ähnliche Idee wie beim Entfernen langer rechter Seiten bei der Umwandlung kontextfreier Grammatiken in Chomsky-Normalform...

## Beispiel

- Für  $\varphi = (x_1 \vee \neg x_2) \wedge (x_2 \vee \neg x_4 \vee x_7 \vee \neg x_8 \vee x_1 \vee \neg x_5)$  sei:

$$\begin{aligned} f(\varphi) \stackrel{\text{def}}{=} & (x_1 \vee \neg x_2 \vee \neg x_2) \wedge \\ & (x_2 \vee \neg x_4 \vee y_1^2) \wedge \\ & (\neg y_1^2 \vee x_7 \vee y_2^2) \wedge \\ & (\neg y_2^2 \vee \neg x_8 \vee y_3^2) \wedge \\ & (\neg y_3^2 \vee x_1 \vee \neg x_5) \end{aligned}$$

- Die erfüllende Belegung  
 $\theta : x_1 \mapsto 0, x_2 \mapsto 0, x_4 \mapsto 1, x_7 \mapsto 0, x_8 \mapsto 0, x_5 \mapsto 1$   
kann zu einer erfüllenden Belegung von  $f(\varphi)$  erweitert werden durch:  
 $y_1^2 \mapsto 1, y_2^2 \mapsto 1, y_3^2 \mapsto 0$

# SAT $\leq_p$ 3-SAT (2/4)

## Beweisskizze

- Sei  $\varphi = K_1 \wedge \cdots \wedge K_m$  eine KNF-Formel

(1)  $f(\varphi) \stackrel{\text{def}}{=} \chi_1 \wedge \cdots \wedge \chi_m$ , wobei die 3-KNF-Formeln  $\chi_i$  wie folgt definiert sind

- Sei  $K_i = L_1 \vee \cdots \vee L_j$  (mit Literalen  $L_\ell$ )
- Wenn  $j = 1$ , dann  $\chi_i \stackrel{\text{def}}{=} L_1 \vee L_1 \vee L_1$
- Wenn  $j = 2$ , dann  $\chi_i \stackrel{\text{def}}{=} L_1 \vee L_2 \vee L_2$
- Wenn  $j = 3$ , dann  $\chi_i \stackrel{\text{def}}{=} K_i$
- Wenn  $j > 3$  verwenden wir  $j - 3$  neue Variablen  $y_1^i, \dots, y_{j-3}^i$  und definieren
$$\chi_i \stackrel{\text{def}}{=} (L_1 \vee L_2 \vee y_1^i) \wedge$$
$$(\neg y_1^i \vee L_3 \vee y_2^i) \wedge$$
$$\vdots$$
$$(\neg y_{j-4}^i \vee L_{j-2} \vee y_{j-3}^i) \wedge$$
$$(\neg y_{j-3}^i \vee L_{j-1} \vee L_j)$$

(2)  $f(\varphi)$  kann in quadratischer Zeit in  $|\varphi|$  berechnet werden

## SAT $\leq_p$ 3-SAT (3/4)

### Beweisskizze (Forts.)

(3)  $\varphi$  erfüllbar  $\Rightarrow f(\varphi)$  erfüllbar:

- Sei  $\theta$  eine Belegung mit  $\theta \models \varphi$
- ➔ für jedes  $i \leq m$  gilt:  $\theta \models K_i$
- Wir zeigen, dass wir  $\theta$  zu einer Belegung  $\theta'$  erweitern können (die auch für die neuen Variablen definiert ist), so dass, für jedes  $i$  gilt:  $\theta' \models \chi_i$
- Da die neuen Variablen jeweils nur in **einer** Teilformel  $\chi_i$  vorkommen, können wir die Erweiterung von  $\theta'$  für jedes  $i$  einzeln definieren

### Beweisskizze (Forts.)

- Sei also  $i \leq m$  und  $K_i = L_1 \vee \dots \vee L_j$
- Falls  $j \leq 3$ , muss  $\theta$  für  $\chi_i$  nicht erweitert werden
- Falls  $j > 3$ :
  - Wenn  $\theta$  eines der beiden ersten Literale von  $K_i$  wahr macht ( $\theta \models L_1$  oder  $\theta \models L_2$ ), können alle neuen Variablen mit 0 belegt werden:
    - \*  $\theta'(y_\ell^i) \stackrel{\text{def}}{=} 0$ , für alle  $\ell \leq j - 3$
  - Wenn nicht, aber  $\theta$  eines der beiden letzten Literale von  $K_i$  wahr macht ( $\theta \models L_{j-1}$  oder  $\theta \models L_j$ ), können alle neuen Variablen auf 1 gesetzt werden
    - \*  $\theta'(y_\ell^i) \stackrel{\text{def}}{=} 1$ , für alle  $\ell \leq j - 3$
  - Andernfalls sei  $p \leq j$  die kleinste Zahl mit  $\theta \models L_p$  und wir setzen
    - \*  $\theta'(y_\ell^i) \stackrel{\text{def}}{=} \begin{cases} 1 & \text{für alle } \ell \leq p - 2 \\ 0 & \text{für alle } \ell \geq p - 1 \end{cases}$
- In allen Fällen folgt dann:  $\theta' \models \chi_i$



# SAT $\leq_p$ 3-SAT (4/4)

## Beweisskizze (Forts.)

(4)  $f(\varphi)$  erfüllbar  $\Rightarrow \varphi$  erfüllbar:

- Sei  $\theta'$  eine erfüllende Belegung für  $f(\varphi)$
- ➔ für jedes  $i \leq m$  gilt:  $\theta' \models \chi_i$
- Wir definieren  $\theta(x_p) \stackrel{\text{def}}{=} \theta'(x_p)$ , für alle Variablen  $x_p$ , die in  $\varphi$  vorkommen
- Zu zeigen: für jedes  $i \leq m$  gilt:  $\theta \models K_i$
- Sei also  $K_i = L_1 \vee \dots \vee L_j$  eine Klausel von  $\varphi$
- Falls  $j \leq 3$ , ist  $K_i$  äquivalent zu  $\chi_i$  und deshalb folgt  $\theta \models K_i$  direkt aus  $\theta' \models \chi_i$
- Beobachtung: falls  $j > 3$  können nicht alle Klauseln von  $\chi_i$  durch Literale mit Variablen  $y_q^i$  wahr gemacht werden
  - denn: nach Konstruktion kann jede Variable nur eine Klausel wahr machen, es sind aber weniger neue Variablen als Klauseln
- ➔ für ein  $p \leq j$  muss gelten  $\theta' \models L_p$ , also auch:  $\theta \models K_i$

Folgerung 19.5

3-SAT ist **NP**-vollständig

# Inhalt

19.1 Der Satz von Cook

19.2 Polynomielle Reduktionen

19.3 3-SAT

▷ **19.4 Das Cliquen-Problem**

19.5 3-Färbbarkeit

19.6 Hamiltonkreise und TSP

19.7 Teilsummen und das Rucksack-Problem

## 3-SAT $\leq_p$ CLIQUE (1/5)

- Die Reduktionen  $3\text{-COL} \leq_p \text{SAT}$  und  $\text{SAT} \leq_p 3\text{-SAT}$  waren nicht allzu kompliziert
  - Dass sich die Korrektheit einer 3-Färbung eines Graphen in einer aussagenlogischen Formel „kodieren“ lässt, ist nicht allzu überraschend
- Wir wollen jetzt zeigen:  $3\text{-SAT} \leq_p \text{CLIQUE}$ 
  - Das ist schon weniger nahe liegend
  - Wie sollen Variablen, Wahrheitsbelegungen und Klauseln in einen Graphen kodiert werden?
  - Das ist deutlich komplizierter
- Grob gesagt, ist die Korrespondenz zwischen Formeln und Graphen in dieser Reduktion wie folgt:
  - Jedes **Vorkommen** eines Literals entspricht einem Knoten im Graphen
  - Kanten zwischen Knoten drücken aus, dass die entsprechenden Literale sich nicht widersprechen
  - Cliquen im Graphen entsprechen dann Mengen simultan erfüllbarer Literale

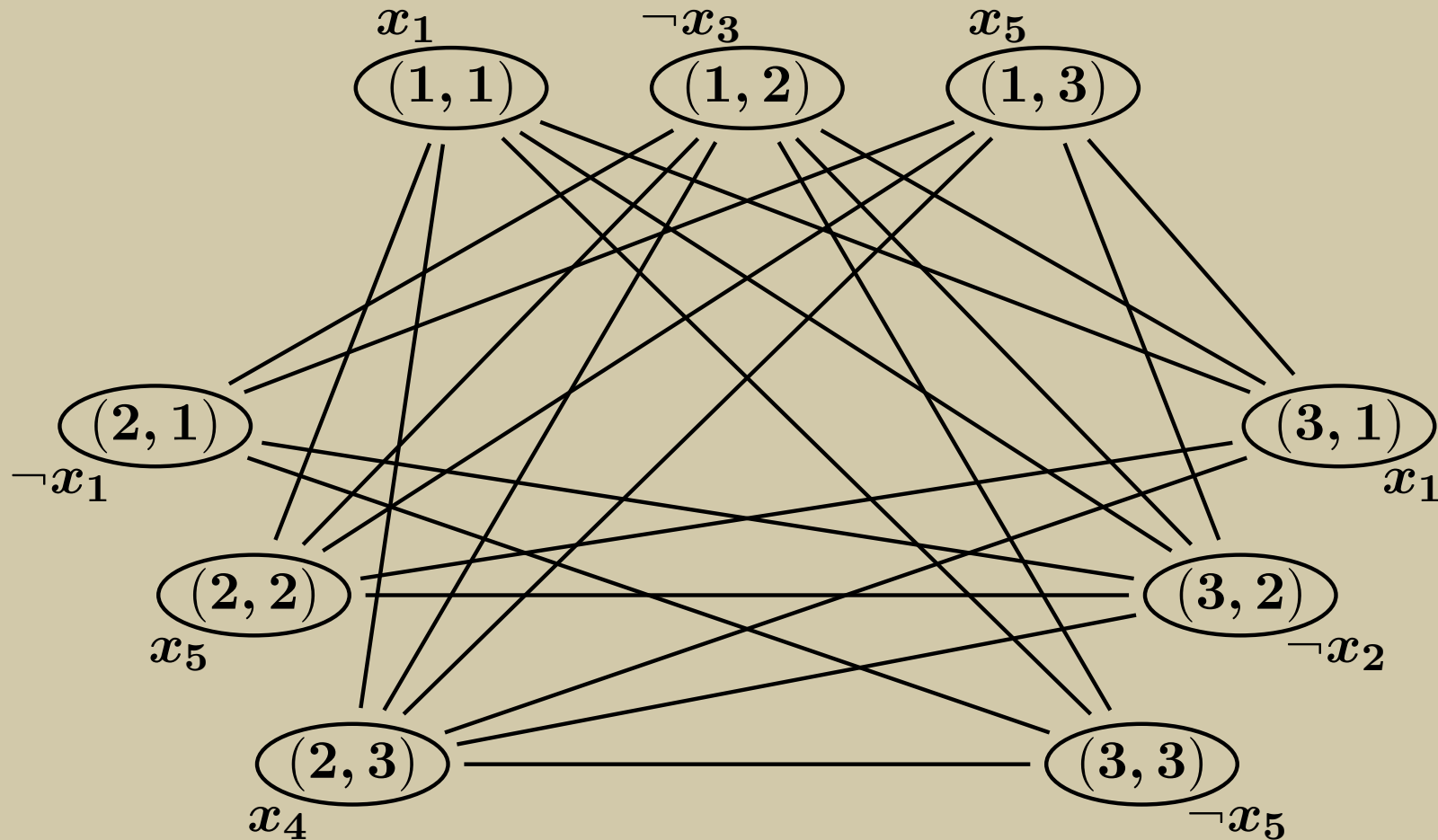
Proposition 19.6

$3\text{-SAT} \leq_p \text{CLIQUE}$

## 3-SAT $\leq_p$ CLIQUE (2/5)

### Illustration der Reduktion von 3-SAT auf CLIQUE

- Beispielformel  $\varphi = (x_1 \vee \neg x_3 \vee x_5) \wedge (\neg x_1 \vee x_5 \vee x_4) \wedge (x_1 \vee \neg x_2 \vee \neg x_5)$
- Beispielgraph  $G_\varphi$ :



## 3-SAT $\leq_p$ CLIQUE (3/5)

### Beweisskizze

- (1) Sei  $\varphi$  eine Formel in KNF mit  $m$  Klauseln zu je 3 Literalen
- Wir konstruieren einen Graphen  $G$  mit  $3m$  Knoten so dass gilt:  
 **$G$  hat eine Clique der Größe  $m \iff \varphi$  ist erfüllbar**
  - Sei  $\varphi = (L_{11} \vee L_{12} \vee L_{13}) \wedge \cdots \wedge (L_{m1} \vee L_{m2} \vee L_{m3})$
  - Sei  $G = (V, E)$  mit
    - $V = \{(i, j) \mid 1 \leq i \leq m, 1 \leq j \leq 3\}$
    - $E = \{((i, j), (k, l)) \mid i \neq k \text{ und } L_{ij} \neq \neg L_{kl}\}$
    - $G$  hat also einen Knoten für jedes Vorkommen eines Literals in einer Klausel von  $\varphi$
    - Zwei Knoten sind miteinander verbunden, wenn ihre Literale
      - \* nicht in derselben Klausel sind und
      - \* sich nicht direkt widersprechen, d.h. nicht einer Variablen  $x_i$  und ihrer Negation  $\neg x_i$  entsprechen
  - Wir definieren  $f(\varphi) \stackrel{\text{def}}{=} (G, m)$
- (2)  $f$  kann in quadratischer Zeit in  $|\varphi|$  berechnet werden

## 3-SAT $\leq_p$ CLIQUE (4/5)

### Beweisskizze (Forts.)

(3)  $\varphi$  erfüllbar  $\Rightarrow G$  hat eine  $m$ -Clique

- Sei  $\theta$  eine erfüllende Belegung für  $\varphi$
- Dann gibt es für jedes  $i \leq m$  ein  $j_i \in \{1, 2, 3\}$ , so dass  $\theta \models L_{ij_i}$
- ➔ Die Literale  $L_{1j_1}, \dots, L_{mj_m}$  sind in verschiedenen Klauseln und widersprechen sich nicht
- ➔ Ihre zugehörigen Knoten bilden eine Clique der Größe  $m$  in  $G$
- ➔ (3)

### Beweisskizze (Forts.)

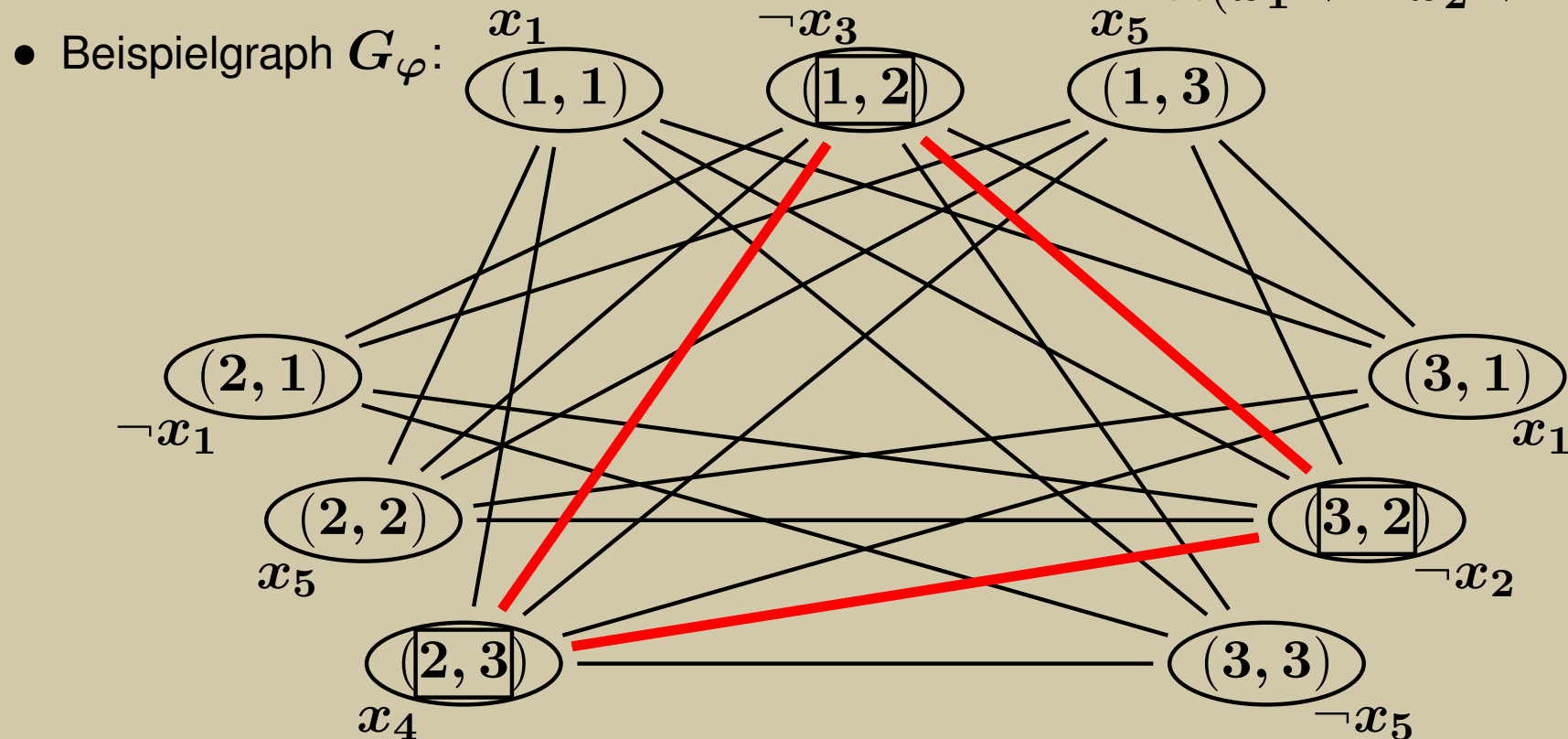
(4)  $G$  hat eine  $m$ -Clique  $\Rightarrow \varphi$  erfüllbar

- Sei  $C$  eine  $m$ -Clique von  $G$
- $C$  besteht aus Knoten zu  $m$  Literalen  $L_{1j_1}, \dots, L_{mj_m}$  aus verschiedenen Klauseln
- Da alle diese Literale miteinander verbunden sind, gibt es keine Variable  $x_p$ , für die sowohl  $x_p$  als auch  $\neg x_p$  in  $\{L_{1j_1}, \dots, L_{mj_m}\}$  vorkommt
- ➔  $\theta$  kann so definiert werden, dass alle Literale  $L_{1j_1}, \dots, L_{mj_m}$  wahr werden
- ➔  $\theta$  macht in jeder Klausel mindestens ein Literal wahr
- ➔ (4)

## 3-SAT $\leq_p$ CLIQUE (5/5)

### Illustration der Reduktion von 3-SAT auf CLIQUE

- Beispielformel  $\varphi = (x_1 \vee \neg x_3 \vee x_5) \wedge (\neg x_1 \vee x_5 \vee x_4) \wedge (x_1 \vee \neg x_2 \vee \neg x_5)$



- Die Knoten  $(1, 2)$ ,  $(2, 3)$  und  $(3, 2)$  bilden eine Clique in  $G_\varphi$
- ➡ Die Literale  $\neg x_3$ ,  $\neg x_2$  und  $x_4$  widersprechen sich nicht
- ➡ Sie induzieren deshalb eine erfüllende Belegung von  $\varphi$ :
  - $\theta(x_2) = 0, \theta(x_3) = 0, \theta(x_4) = 1$
  - Der Rest ist frei wählbar, z.B.:  $\theta(x_1) = 0, \theta(x_5) = 1$

# Inhalt

19.1 Der Satz von Cook

19.2 Polynomielle Reduktionen

19.3 3-SAT

19.4 Das Cliques-Problem

▷ **19.5 3-Färbbarkeit**

19.6 Hamiltonkreise und TSP

19.7 Teilsummen und das Rucksack-Problem



## 3-SAT $\leq_p$ 3-COL (1/9)

- Bei der nächsten Reduktion 3-SAT  $\leq_p$  3-COL wird wieder zu jeder 3KNF-Formel ein Graph (und eine Zahl) konstruiert
- Die Korrespondenz zwischen den Bestandteilen des Graphen und der Formel ist jedoch anders
- Wir assoziieren
  - jede aussagenlogische Variable  $x_i$  mit jeweils zwei Knoten  $x_i$  und  $\neg x_i$  des Graphen,
  - Wahrheitswerte mit Farben,
  - und Klauseln mit speziell konstruierten Teilgraphen, die nur dann korrekt gefärbt werden können, wenn die entsprechende Klausel durch die gegebene Wahrheitsbelegung wahr wird

### Proposition 19.7

- 3-SAT  $\leq_p$  3-COL

## 3-SAT $\leq_p$ 3-COL (2/9)

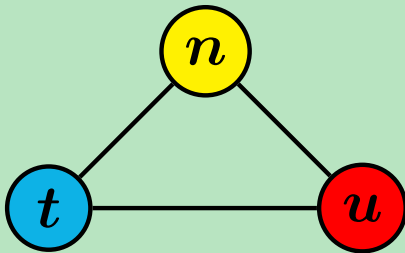
### Beweisskizze

- (1) Sei  $\varphi = K_1 \wedge \dots \wedge K_m$  eine 3KNF-Formel mit Variablen  $x_1, \dots, x_n$  und Klauseln  $K_1, \dots, K_m$
- Mit  $L_{i1}, L_{i2}, L_{i3}$  seien jeweils die drei Literale der Klausel  $K_i$  bezeichnet
  - Also:  $K_i = L_{i1} \vee L_{i2} \vee L_{i3}$
- Die Reduktion konstruiert einen Graphen  $G$  mit folgenden Knoten:
- Für jedes  $i \leq n$  je ein Knoten  $x_i$  und  $\neg x_i$   
(die **Literalknoten**)
  - Für jedes  $i \leq m$  fünf Knoten  $b_i, c_i, d_i, e_i, g_i$  pro Klausel  $K_i$   
(die **Klauselknoten**)
  - Drei Knoten  $t, u, n$
- Intention:  $G$  ist genau dann mit rot, blau und gelb zulässig färbbar, wenn  $\varphi$  erfüllbar ist
- Wir definieren dann  $f(\varphi) \stackrel{\text{def}}{=} G$

 Literale kommen im Folgenden in  $\varphi$  und als Knoten in  $G$  vor!

## Beweisskizze (Forts.)

- Unabhängig von der gegebenen Formel enthält  $G$  die Kanten  $(u, t), (u, n), (t, n)$ :

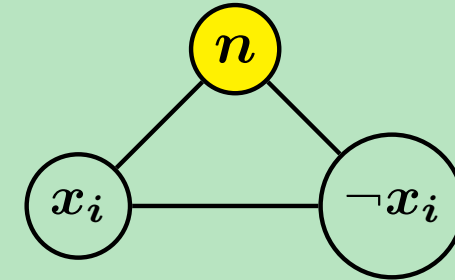


- Intention:**  $t, u, n$  müssen verschieden gefärbt werden

- Für den Nachweis der Reduktionseigenschaft werden wir im Folgenden oBdA davon ausgehen, dass zulässige Färbungen  $t$  blau,  $u$  rot und  $n$  gelb färben
- Intuitiv soll blau „wahr“ und rot „unwahr“ entsprechen, (gelb ist „neutral“)

## Beweisskizze (Forts.)

- Für jedes  $i$  enthält  $G$  die Kanten  $(n, x_i), (n, \neg x_i), (\neg x_i, x_i)$ :

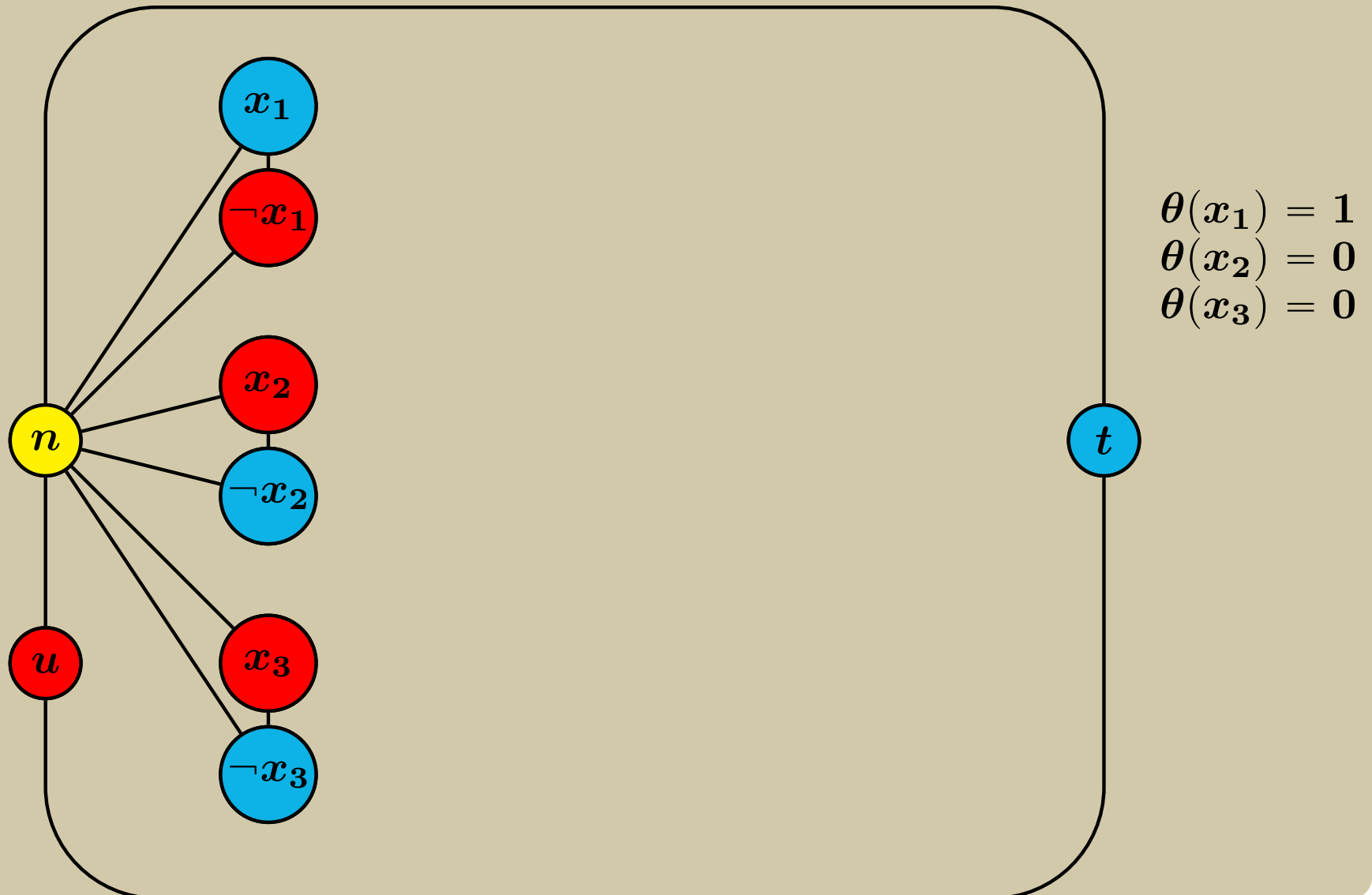


- Intention:**
  - Die Knoten  $x_i$  und  $\neg x_i$  müssen jeweils verschieden gefärbt werden klar
  - Da  $n$  gelb ist, müssen dafür die Farben rot und blau verwendet werden
- Wir erweitern unsere Intention entsprechend:
  - Falls  $x_i$  in  $G$  blau gefärbt wird, soll dies  $\theta(x_i) = 1$  entsprechen
  - Falls  $\neg x_i$  in  $G$  blau gefärbt wird, soll dies  $\theta(x_i) = 0$  entsprechen

# 3-SAT $\leq_p$ 3-COL (4/9)

## Beispiel (Forts.)

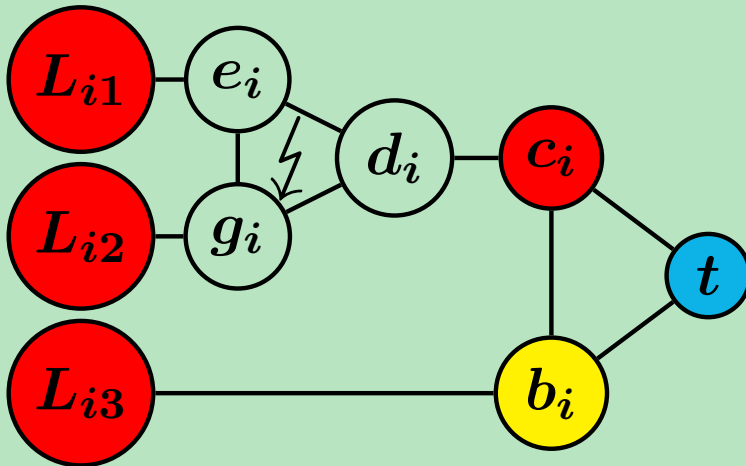
- Beispielformel  $\varphi = (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee x_2 \vee \neg x_3)$
- Beispielgraph  $G$  (im Aufbau):



# 3-SAT $\leq_p$ 3-COL (5/9)

## Beweisskizze (Forts.)

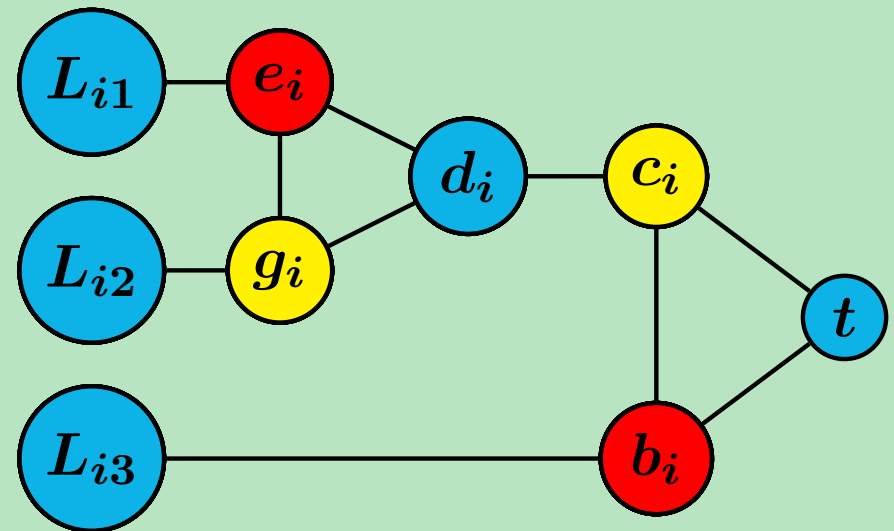
- Schließlich verwenden wir für jede Klausel  $K_i$  mit Literalen  $L_{i1}, L_{i2}, L_{i3}$  einen Teilgraphen  $H_i$  der folgenden Form



- Dabei sind  $b_i, c_i, d_i, e_i, g_i$  neue Knoten, die nur in  $H_i$  vorkommen
- Beobachtung: Sind alle drei Knoten  $L_{i1}, L_{i2}, L_{i3}$  rot, so kann dieser Teilgraph nicht zulässig gefärbt werden

## Beweisskizze (Forts.)

- Umgekehrt kann jede Färbung, die mindestens einen der Knoten  $L_i$  blau färbt, zu einer zulässigen Färbung des Teilgraphen erweitert werden:

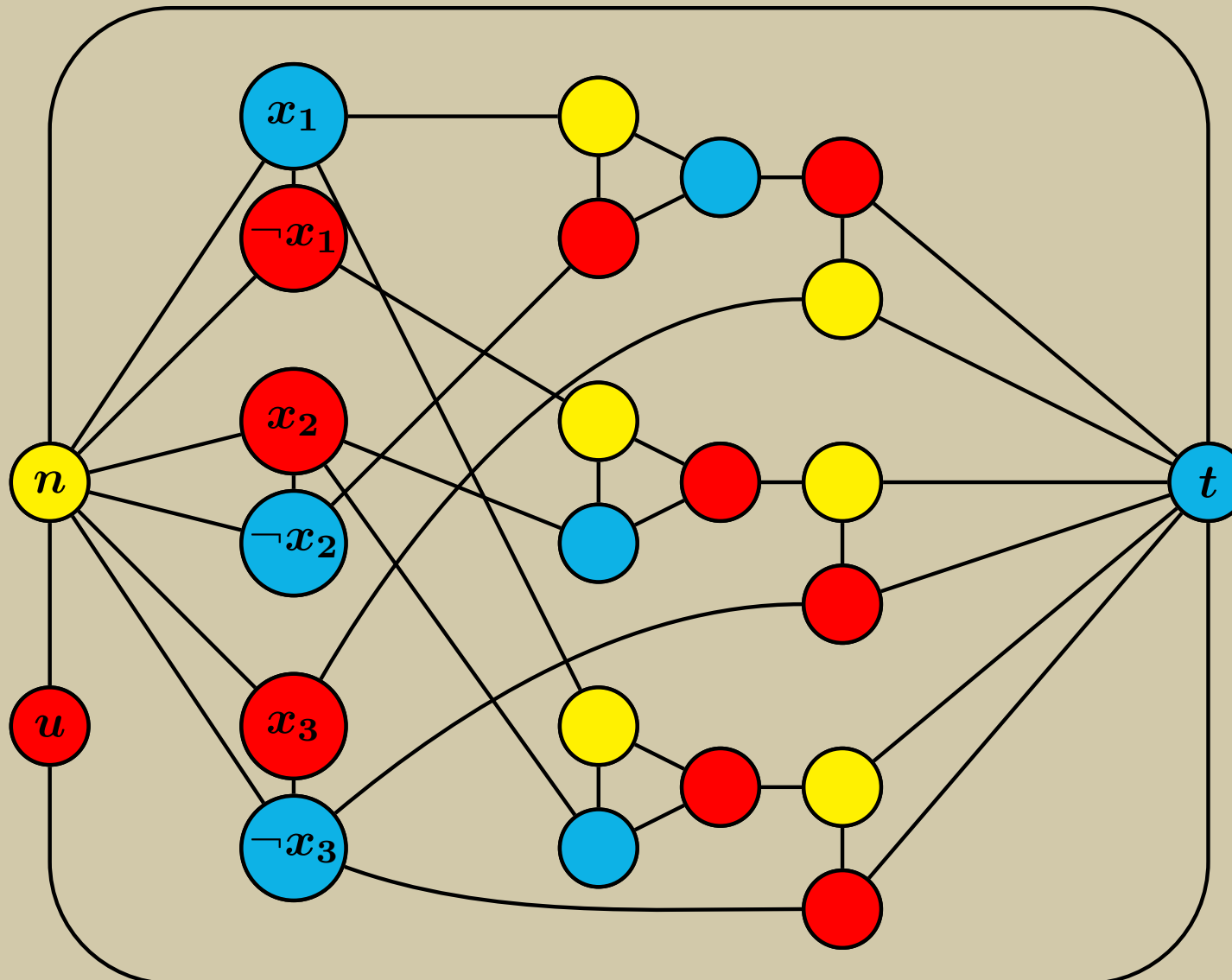


- Wir nennen den Teilgraphen  $H_i$  mit den Knoten  $b_i, c_i, d_i, e_i, g_i$ , den **Klausel-Teilgraphen** zu  $K_i$
- Die Knoten  $L_{i1}, L_{i2}, L_{i3}$  nennen wir die **mit  $H_i$  verbundenen Literalknoten**

# 3-SAT $\leq_p$ 3-COL (6/9)

## Beispiel (Forts.)

- Beispielformel  $\varphi = (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee x_2 \vee \neg x_3)$
- Beispielgraph  $G$ :



$$\begin{aligned}\theta(x_1) &= 1 \\ \theta(x_2) &= 0 \\ \theta(x_3) &= 0\end{aligned}$$

## Beweis von $3\text{-SAT} \leq_p 3\text{-COL}$ (7/9)

### Beweisskizze (Forts.)

- $G$  hat also insgesamt die folgenden Kanten:
  - $(u, t), (u, n), (t, n)$
  - Für jedes  $i \leq n$ :  $(n, x_i), (n, \neg x_i), (\neg x_i, x_i)$
  - Für jedes  $i \leq m$ :  
 $(L_{i1}, e_i), (L_{i2}, g_i), (L_{i3}, b_i), (e_i, g_i), (e_i, d_i),$   
 $(g_i, d_i), (d_i, c_i), (c_i, b_i), (c_i, t), (b_i, t)$

(2)  $f(\varphi) = G$  kann in polynomieller Zeit berechnet werden

## 3-SAT $\leq_p$ 3-COL (8/9)

### Beweisskizze (Forts.)

#### (3) $\varphi$ erfüllbar $\Rightarrow G$ 3-färbbar

- Sei  $\theta : \{x_1, \dots, x_n\} \rightarrow \{0, 1\}$  eine erfüllende Belegung für  $\varphi$
- Wir konstruieren eine zulässige Färbung  $c$ :
  - $c(t) \stackrel{\text{def}}{=} \text{blau}$ ,  $c(u) \stackrel{\text{def}}{=} \text{rot}$ ,  $c(n) \stackrel{\text{def}}{=} \text{gelb}$
  - $c(x_i) \stackrel{\text{def}}{=} \text{blau}$ , falls  $\theta(x_i) = 1$ , andernfalls rot
  - $c(\neg x_i) \stackrel{\text{def}}{=} \text{blau}$ , falls  $\theta(x_i) = 0$ , andernfalls rot
- Da  $\theta$  jede Klausel  $K_i$  wahr macht, hat jeder Teilgraph  $H_i$  mindestens einen blauen Literal-Knoten
- ➡ Jeder Teilgraph  $H_i$  kann zulässig gefärbt werden
- Bezüglich der übrigen Kanten ist die entstehende Färbung ebenfalls zulässig
- ➡  $f(\varphi)$  ist 3-färbbar
- ➡ (3)



## 3-SAT $\leq_p$ 3-COL (9/9)

### Beweis (Forts.)

#### (4) $G$ 3-färbbar $\Rightarrow \varphi$ erfüllbar

- Sei  $c$  eine zulässige Färbung von  $G$
- OBdA:  $c(t) = \text{blau}$ ,  $c(u) = \text{rot}$ ,  $c(n) = \text{gelb}$ 
  - Sonst benennen wir die Farben um

- ➡ Für jedes  $i \leq n$  gilt:
- $c(x_i) = \text{blau}$  und  $c(\neg x_i) = \text{rot}$  oder
  - $c(x_i) = \text{rot}$  und  $c(\neg x_i) = \text{blau}$

### Beweis (Forts.)

- Definiere  $\theta$  durch
  - $\theta(x_i) \stackrel{\text{def}}{=} \begin{cases} 1 & \text{falls } c(x_i) = \text{blau} \\ 0 & \text{falls } c(x_i) = \text{rot} \end{cases}$

- Für jedes Literal  $L$  gilt also:  
 $\theta \models L \iff c(L) = \text{blau}$

- Da jeder Klauselgraph  $H_i$  zulässig gefärbt ist, muss jeweils mindestens einer der Literalknoten  $L_{i1}, L_{i2}, L_{i3}$  blau sein
- ➡ für jede Klausel  $K_i$  wird mindestens eines der Literale  $L_{i1}, L_{i2}, L_{i3}$  wahr
- ➡  $\theta$  ist eine erfüllende Belegung von  $\varphi$
- ➡  $\varphi$  erfüllbar
- ➡ (4)

# Präsentation von Reduktionsbeweisen

- In der Darstellung von Reduktionen (z.B.: von 3-SAT auf 3-COL) vermischen sich meist mehrere Aspekte
- Die Beschreibung der Reduktionsfunktion
  - im Beispiel: der Funktion
$$f : \varphi \mapsto G_\varphi$$
- Die Beschreibung der Intention der Reduktion, also z.B.
  - die Korrespondenz zwischen Färbungen und Wahrheitsbelegungen („falls Knoten  $x_i$  blau wird, ist  $\theta(x_i) = 1$ “)
  - die Färbbarkeitseigenschaften der elementaren Dreiecke und der Klauselgraphen
- Diese Vermischung erscheint schwer vermeidlich, da andernfalls die Konstruktion kaum zu verstehen wäre

- Wichtig ist aber, dass Sie sich klarmachen, dass die Reduktionsfunktion  $f$  selbst *weder eine erfüllende Belegung noch eine zulässige Färbung konstruiert*
- Es besteht lediglich ein *Zusammenhang* zwischen erfüllenden Belegungen und zulässigen Färbungen
  - Mit Hilfe dieses Zusammenhangs beweisen wir dann in den Schritten (3) und (4), dass  $f$  eine Reduktion ist

# Inhalt

19.1 Der Satz von Cook

19.2 Polynomielle Reduktionen

19.3 3-SAT

19.4 Das Cliques-Problem

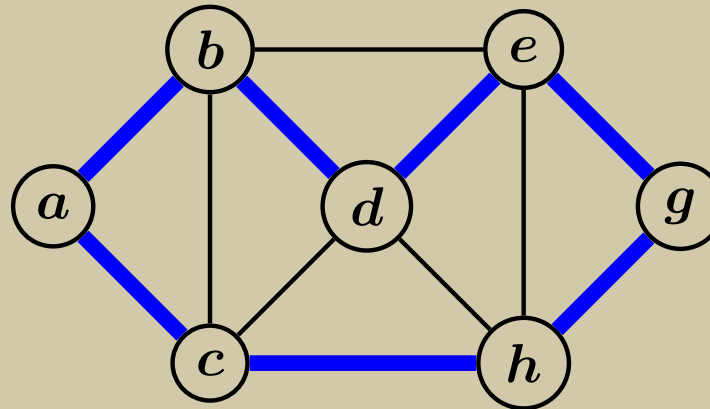
19.5 3-Färbbarkeit

▷ **19.6 Hamiltonkreise und TSP**

19.7 Teilsummen und das Rucksack-Problem

# Zur Erinnerung: Hamilton-Kreise

Beispiel



Definition: HAMILTONCYCLE

**Gegeben:** Ungerichteter Graph  $G$

**Frage:** Gibt es einen geschlossenen Weg in  $G$ , der jeden Knoten genau einmal besucht?

## 3-SAT $\leq_p$ HAMILTONCYCLE (1/9)

- Bei der folgenden Reduktion von 3-SAT auf HAMILTONCYCLE entspricht jede Variable  $x_i$  der Formel einem Knoten  $v_i$  des Graphen
- Jeder Knoten  $v_i$  hat zwei ausgehende Kanten, die zu den beiden Literalen  $x_i$  und  $\neg x_i$  korrespondieren
- Jede Wahrheitsbelegung entspricht dann der Auswahl einer Menge von Kanten für die Literale, die sie wahr macht
- Zu jeder Klausel der Formel gibt es einen Teilgraphen
  - Diese Klauselgraphen sorgen dafür, dass sich eine Menge von „Ausgangskanten“ genau dann zu einem Hamiltonkreis erweitern lässt, wenn die entsprechende Wahrheitsbelegung erfüllend ist

Proposition 19.8

3-SAT  $\leq_p$  HAMILTONCYCLE

## 3-SAT $\leq_p$ HAMILTONCYCLE (2/9)

### Beweisskizze

- Wir zeigen zuerst: 3-SAT  $\leq_p$  GHAMILTONCYCLE
  - GHAMILTONCYCLE: Gegeben ein **gerichteter** Graph  $G$ , hat  $G$  einen **gerichteten** Hamiltonkreis?

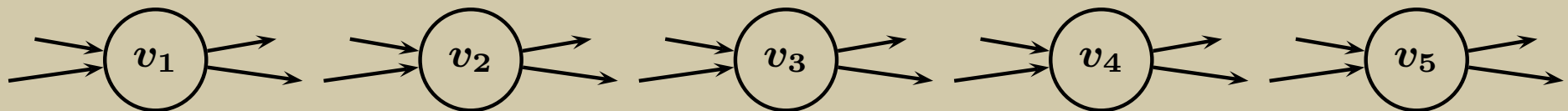
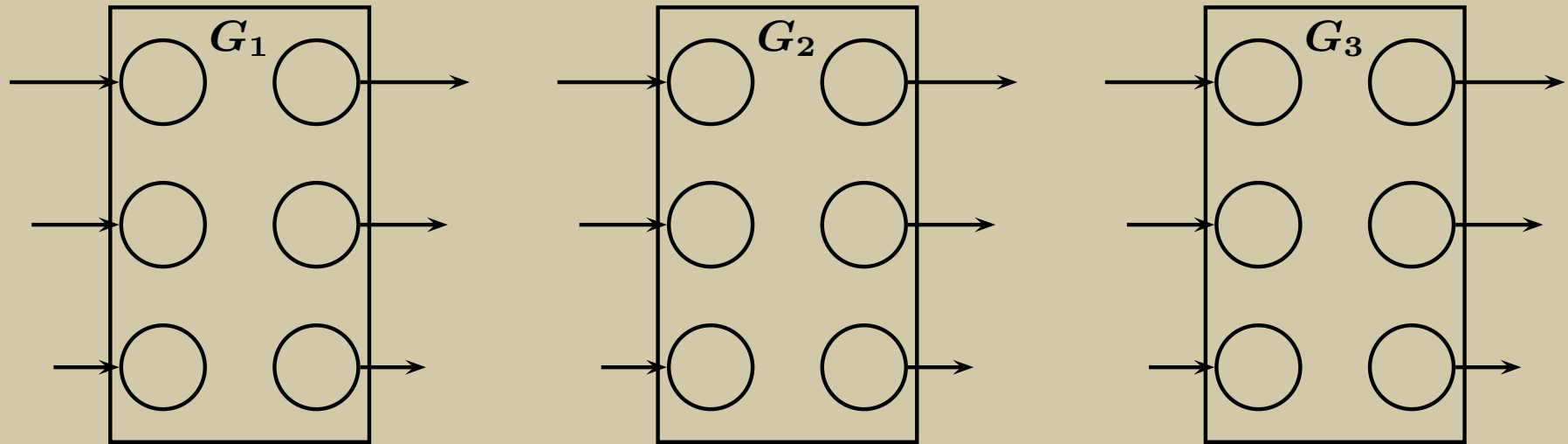
(1) Sei  $\varphi = (L_{11} \vee L_{12} \vee L_{13}) \wedge \cdots \wedge (L_{m1} \vee L_{m2} \vee L_{m3})$

- Seien  $x_1, \dots, x_n$  die Variablen von  $\varphi$
- Der Graph  $G$  hat  $n + 6m$  Knoten:
  - Die Knoten  $v_1, \dots, v_n$  repräsentieren die Variablen von  $\varphi$
  - Die Teilgraphen  $G_1, \dots, G_m$  mit Knoten  $u_{ij}$  und  $u'_{ij}$ ,  $1 \leq i \leq m, 1 \leq j \leq 3$ , repräsentieren die Klauseln von  $\varphi$
- Jeder Knoten  $v_i$  hat 2 eingehende und 2 ausgehende Kanten
  - Intention: die obere ausgehende Kante entspricht dem Wahrheitswert 1
- Die Teilgraphen  $G_j$  haben je 3 eingehende und 3 ausgehende Kanten

# 3-SAT $\leq_p$ HAMILTONCYCLE (3/9)

## Beispiel

- Beispiel-Formel:  $\varphi = (x_1 \vee \neg x_3 \vee x_5) \wedge (\neg x_1 \vee x_5 \vee x_4) \wedge (x_1 \vee \neg x_2 \vee \neg x_5)$
- Beispiel-Graph  $G$ :



## 3-SAT $\leq_p$ HAMILTONCYCLE (4/9)

### Beweisskizze (Forts.)

- Sei  $i$  fest und seien  $K_{j_1}, \dots, K_{j_l}$  die Klauseln, in denen  $x_i$  **positiv** vorkommt, und zwar als  $L_{j_1}b_1, \dots, L_{j_l}b_l$ 
  - D.h.:  $x_i$  kommt in Klausel  $K_{j_t}$  an der  $b_t$ -ten Stelle vor, für  $t \leq l$
- Dann hat  $G$  Kanten
  - vom ersten Ausgang von Knoten  $v_i$  zum  $b_1$ -ten Eingang von  $G_{j_1}$ ,
  - vom  $b_t$ -ten Ausgang von  $G_{j_t}$  zum  $b_{t+1}$ -ten Eingang von  $G_{j_{t+1}}$ , für alle  $t < l$ , und
  - vom  $b_l$ -ten Ausgang von  $G_{j_l}$  zum ersten Eingang von  $v_{i+1}$  (für  $i = n$  zum ersten Eingang von  $v_1$ ).
- Wir nennen diese Kantenmenge den **Literalweg** zu  $x_i$

### Beweisskizze (Forts.)

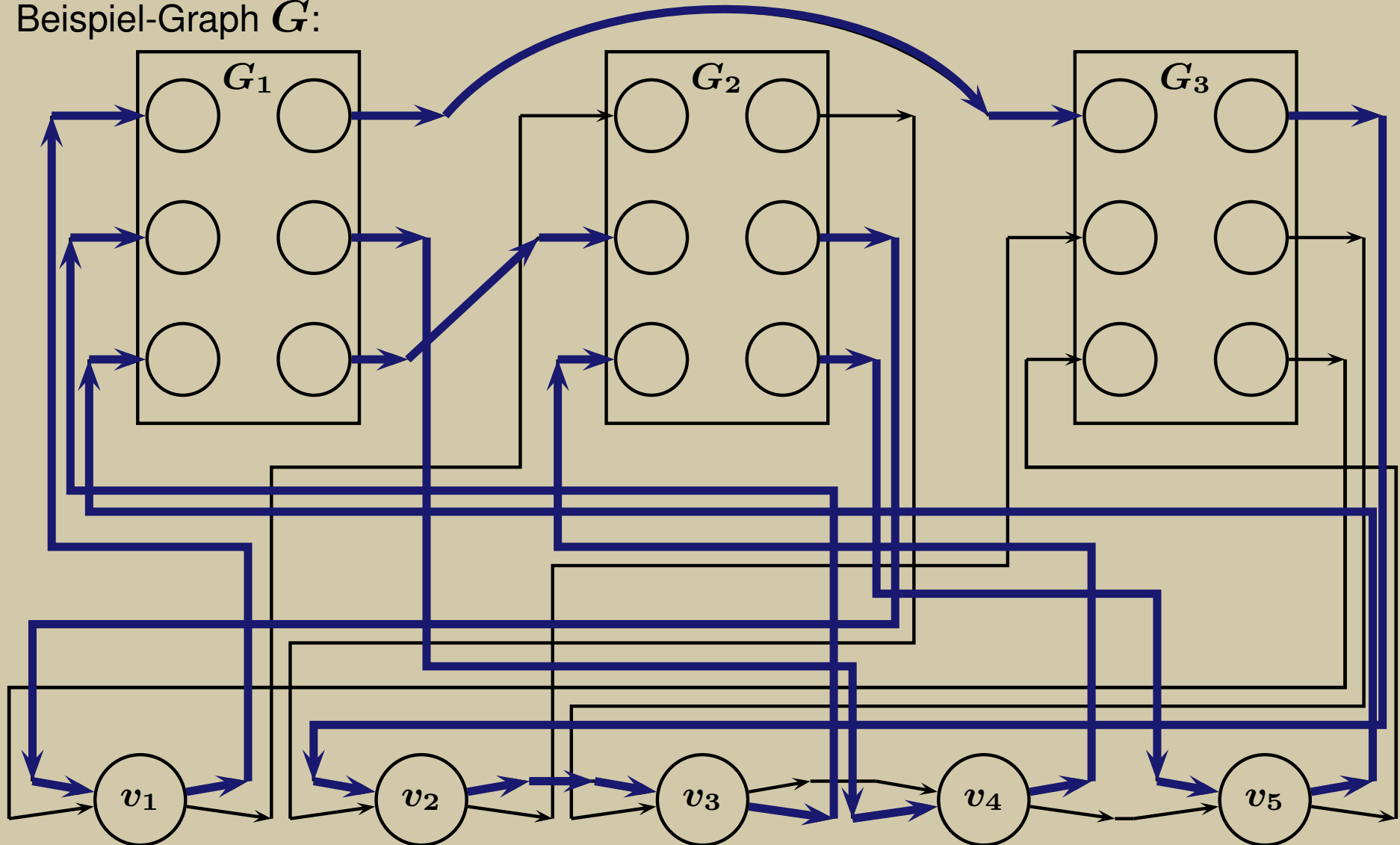
- Analog gibt es einen Literalweg zu  $\neg x_i$ , der im unteren Ausgang von  $v_i$  beginnt und alle Teilgraphen  $G_j$  mit **negativen** Vorkommen von  $x_i$  durchläuft
- Für jede Variable hat  $G$  also zwei Literalwege
- Intention: wenn eine Wahrheitsbelegung ein Literal wahr macht, dann besucht der Literalweg des Literals die Klauselgraphen aller Klauseln, die durch das Literal wahr werden



# 3-SAT $\leq_p$ HAMILTONCYCLE (5/9)

## Beispiel

- Beispiel-Formel:  $\varphi = (x_1 \vee \neg x_3 \vee x_5) \wedge (\neg x_1 \vee x_5 \vee x_4) \wedge (x_1 \vee \neg x_2 \vee \neg x_5)$   
Belegung:  $\theta(x_1) = 1, \theta(x_2) = 1, \theta(x_3) = 0, \theta(x_4) = 1, \theta(x_5) = 1$
- Beispiel-Graph  $G'$ :

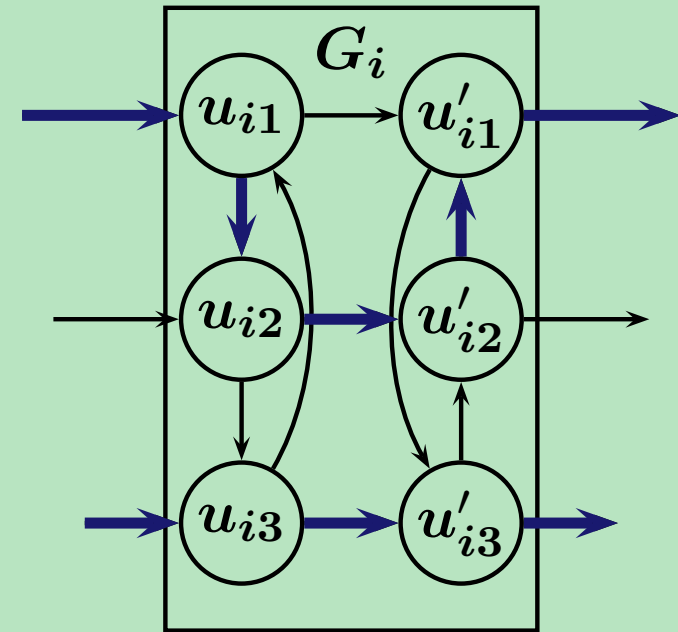


# 3-SAT $\leq_p$ HAMILTONCYCLE (6/9)

## Beweisskizze (Forts.)

- Konstruktion der Teilgraphen  $G_i$ :
- **Intention:** die Knoten von  $G_i$  lassen sich auf jede nicht-leere Menge von Literalwegen durch  $G_i$  verteilen (egal, ob es ein, zwei oder drei sind)
- $u_{i1}, u_{i2}, u_{i3}$ : Eingangsknoten, mit jeweils einer eingehenden Kante
- $u'_{i1}, u'_{i2}, u'_{i3}$ : Ausgangsknoten, mit jeweils einer ausgehenden Kante
- $G_i$  hat folgende innere Kanten:
  - $(u_{i1}, u'_{i1}), (u_{i2}, u'_{i2}), (u_{i3}, u'_{i3})$
  - $(u_{i1}, u_{i2}), (u_{i2}, u_{i3}), (u_{i3}, u_{i1})$
  - $(u'_{i2}, u'_{i1}), (u'_{i3}, u'_{i2}), (u'_{i1}, u'_{i3})$
  - Zu beachten ist, dass die Kanten zwischen den Knoten  $u'_{ij}$  gegenläufig zu den Kanten zwischen den Knoten  $u_{ij}$  sind

## Beweisskizze (Forts.)

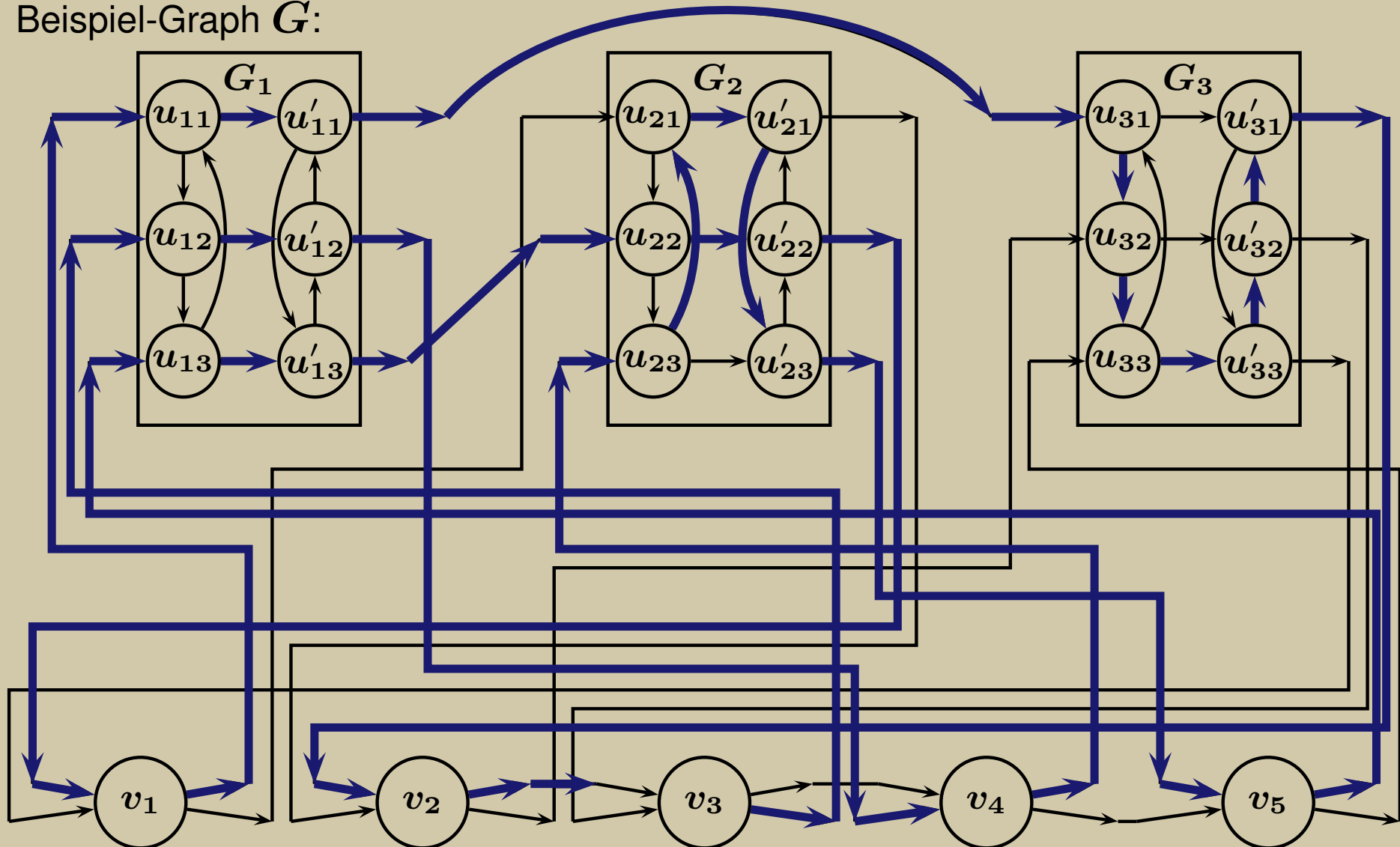


- Es gilt: für jede nicht-leere Menge  $S$  von Eingangskanten von  $G_i$  gibt es eine Menge von Wege innerhalb  $G_i$ n, die
  - alle Knoten aus  $G_i$  genau einmal besuchen,
  - $G_i$  in der Menge  $S$  betritt, und
  - $G_i$  in der zu  $S$  korrespondierenden Kantenmenge verlässt
- $f(\varphi) \stackrel{\text{def}}{=} G$

# 3-SAT $\leq_p$ HAMILTONCYCLE (7/9)

## Beispiel (Forts.)

- Beispiel-Formel:  $\varphi = (x_1 \vee \neg x_3 \vee x_5) \wedge (\neg x_1 \vee x_5 \vee x_4) \wedge (x_1 \vee \neg x_2 \vee \neg x_5)$   
Belegung:  $\theta(x_1) = 1, \theta(x_2) = 1, \theta(x_3) = 0, \theta(x_4) = 1, \theta(x_5) = 1$
- Beispiel-Graph  $G'$ :



## 3-SAT $\leq_p$ HAMILTONCYCLE (8/9)

### Beweisskizze (Forts.)

(2)  $f$  ist in polynomieller Zeit berechenbar

(3)  $\varphi$  erfüllbar  $\Rightarrow G$  hat gerichteten Hamilton-Kreis

- Sei  $\theta$  erfüllende Belegung für  $\varphi$
  - Zur Konstruktion des Hamiltonkreises  $H$  wird zunächst für jedes Literal  $L$  mit  $\theta(L) = 1$  der Literalweg zu  $L$  gewählt
  - Da  $\theta$  erfüllend ist, wird jedes  $G_i$  durch mindestens einen dieser Literalwege besucht
  - Nach Konstruktion der  $G_i$  können Innenkanten so gewählt werden, dass alle Knoten genau einmal erreicht werden (und der Weg ist zusammenhängend)
- ➡  $H$  ist Hamilton-Kreis

# 3-SAT $\leq_p$ HAMILTONCYCLE (9/9)

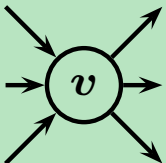
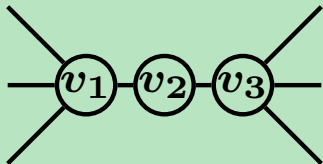
## Beweisskizze (Forts.)

(4)  $G$  hat gerichteten Hamilton-Kreis  $\Rightarrow \varphi$  ist erfüllbar

- Sei  $H$  ein Hamiltonkreis
- Falls die obere von  $v_i$  ausgehende Kante in  $H$  ist, setze  $\theta(x_i) \stackrel{\text{def}}{=} 1$  sonst  $\theta(x_i) \stackrel{\text{def}}{=} 0$
- Da jeder Klauselgraph durchlaufen wird, muss jede Klausel durch mindestens ein Literal wahr gemacht werden

➡  $\varphi$  erfüllbar

- Noch zu zeigen: GHAMILTONCYCLE  $\leq_p$  HAMILTONCYCLE

- Ersetze dazu  durch 

- Dann gilt:  
Der gerichtete Graph hat einen Hamilton-Kreis  $\iff$   
der ungerichtete Graph hat einen Hamilton-Kreis

# HAMILTONCYCLE $\leq_p$ TSP

## Satz 19.9

$$\text{HAMILTONCYCLE} \leq_p \text{TSP}$$

### Beweisskizze

(1) Sei  $G = (V, E)$  mit Knotenmenge  
$$V = \{v_1, \dots, v_n\}$$

- Sei  $f(G) \stackrel{\text{def}}{=} (s_1, \dots, s_n, d, n)$ , mit
  - $d(s_i, s_j) \stackrel{\text{def}}{=} \begin{cases} 1 & \text{für } (v_i, v_j) \in E \\ 2 & \text{für } (v_i, v_j) \notin E \end{cases}$

(2) ✓

- Es ist leicht zu sehen:
  - Jede TSP-Reise hat mindestens die Gesamtentfernung  $n$
  - Jeder Hamiltonkreis von  $G$  entspricht einer TSP-Reise der Gesamtstrecke  $n$  und umgekehrt

➡ (3), (4)

# Inhalt

19.1 Der Satz von Cook

19.2 Polynomielle Reduktionen

19.3 3-SAT

19.4 Das Cliquen-Problem

19.5 3-Färbbarkeit

19.6 Hamiltonkreise und TSP

▷ **19.7 Teilsummen und das Rucksack-Problem**

## 3-SAT $\leq_p$ SUBSETSUM $\leq_p$ KNAPSACK (1/6)

- Für den Nachweis, dass KNAPSACK **NP**-schwierig ist, verwenden wir das folgende Problem als Zwischenschritt

### Definition: SUBSETSUM

**Gegeben:** Menge  $S$  natürlicher Zahlen und eine Zielzahl  $k$

**Frage:** Gibt es  $T \subseteq S$  mit  $\sum_{a \in T} a = k$ ?

- Im Gegensatz zu den bisherigen Reduktionen wird in der folgenden Reduktion die Kodierung von Informationen in Zahlen eine Rolle spielen



# 3-SAT $\leq_p$ SUBSETSUM $\leq_p$ KNAPSACK (2/6)

Proposition 19.10

3-SAT  $\leq_p$  SUBSETSUM

Beweisskizze

(1) Sei  $\varphi = K_1 \wedge \dots \wedge K_m$ , mit Variablen aus  $\{x_1, \dots, x_n\}$

- Wir „kodieren“  $\varphi$  durch  $(m + n)$ -stellige Dezimalzahlen:

- Die ersten  $m$  Stellen entsprechen den  $m$  Klauseln

☞ vorderer Teil

- Die letzten  $n$  Stellen entsprechen den  $n$  Variablen

☞ hinterer Teil

- $k \stackrel{\text{def}}{=} \underbrace{44 \dots 44}_m \underbrace{11 \dots 11}_n$

- **Intention:**  $\varphi$  erfüllbar  $\iff$

es gibt eine Menge  $T$  mit  $\sum_{a \in T} a = k$

- Für jedes Literal gibt es in  $S$  eine Zahl:  $a_i$  für  $x_i$  und  $b_i$  für  $\neg x_i$
- Intention:  $\theta(x_i) = 1$  entspricht der Wahl von  $a_i$ ,  
 $\theta(x_i) = 0$  entspricht der Wahl von  $b_i$

# 3-SAT $\leq_p$ SUBSETSUM $\leq_p$ KNAPSACK (3/6)

## Beweisskizze (Forts.)

- $a_i, b_i$  haben eine 1 an der  $i$ -ten Position im hinteren Teil (und an allen anderen hinteren Positionen eine 0)
- An der  $j$ -ten Stelle des vorderen Teils hat  $a_i$  die Ziffer  $\ell \stackrel{\text{def}}{\Leftrightarrow} x_i$  kommt  $\ell$ -mal in  $K_j$  vor  
 $\Rightarrow$  analog für  $b_i$  und  $\neg x_i$

- Jede Wahrheitsbelegung  $\theta$  korrespondiert also zu einer Zahlenmenge, deren Summe
  - im hinteren Teil an jeder Position eine 1, und
  - im vorderen Teil je Klausel-Position die Anzahl ihrer wahren Literale hat

➡  $\theta \models \varphi \iff$  die Summe hat im vorderen Teil an jeder Position eine Zahl zwischen 1 und 3

- Damit im vorderen Teil überall eine 4 (und damit genau  $k$ ) erreicht werden kann, gibt es in  $S$  je Klausel von  $\varphi$  zwei weitere Zahlen:
  - $y_j$ : vorne an Position  $j$  eine 1, sonst 0
  - $z_j$ : vorne an Position  $j$  eine 2, sonst 0

## Beispiel

- Beispiel-Formel
 
$$\varphi = (x_1 \vee \neg x_3 \vee x_5) \wedge (\neg x_1 \vee x_5 \vee x_4) \wedge (x_1 \vee \neg x_2 \vee \neg x_5)$$
- $k = 444\ 11111$

|                    |                    |
|--------------------|--------------------|
| $a_1 = 101\ 10000$ | $b_1 = 010\ 10000$ |
| $a_2 = 000\ 01000$ | $b_2 = 001\ 01000$ |
| $a_3 = 000\ 00100$ | $b_3 = 100\ 00100$ |
| $a_4 = 010\ 00010$ | $b_4 = 000\ 00010$ |
| $a_5 = 110\ 00001$ | $b_5 = 001\ 00001$ |

|                    |
|--------------------|
| $y_1 = 100\ 00000$ |
| $y_2 = 010\ 00000$ |
| $y_3 = 001\ 00000$ |
| $z_1 = 200\ 00000$ |
| $z_2 = 020\ 00000$ |
| $z_3 = 002\ 00000$ |

## 3-SAT $\leq_p$ SUBSETSUM $\leq_p$ KNAPSACK (4/6)

### Vervollständigung des Beispiels

- Beispiel-Formel  $\varphi = (x_1 \vee \neg x_3 \vee x_5) \wedge (\neg x_1 \vee x_5 \vee x_4) \wedge (x_1 \vee \neg x_2 \vee \neg x_5)$

- Zahlen:

|                    |                    |                    |
|--------------------|--------------------|--------------------|
| $a_1 = 101\ 10000$ | $b_1 = 010\ 10000$ | $y_1 = 100\ 00000$ |
| $a_2 = 000\ 01000$ | $b_2 = 001\ 01000$ | $y_2 = 010\ 00000$ |
| $a_3 = 000\ 00100$ | $b_3 = 100\ 00100$ | $y_3 = 001\ 00000$ |
| $a_4 = 010\ 00010$ | $b_4 = 000\ 00010$ | $z_1 = 200\ 00000$ |
| $a_5 = 110\ 00001$ | $b_5 = 001\ 00001$ | $z_2 = 020\ 00000$ |
|                    |                    | $z_3 = 002\ 00000$ |

- Die Wahrheitsbelegung
  - $\theta(x_1) = \theta(x_2) = \theta(x_4) = \theta(x_5) = 1$
  - $\theta(x_3) = 0$

entspricht der Auswahl  $a_1, a_2, b_3, a_4, a_5$

- Summe: **321 11111**
- Wähle zusätzlich:  $y_1, z_2, y_3, z_3$
- Gesamtsumme: **444 11111**

# 3-SAT $\leq_p$ SUBSETSUM $\leq_p$ KNAPSACK (5/6)

## Beweisskizze (Forts.)

(1)  $f(\varphi) \stackrel{\text{def}}{=} (S, k)$ , wobei  $S$  die Menge der Zahlen der  $a_i, b_i, y_j, z_j$  ist

(2) ✓

(3)  $\varphi$  ist erfüllbar  $\Rightarrow f(\varphi)$  hat eine Lösung

- Sei  $\theta$  eine Belegung, die  $\varphi$  wahr macht
- Falls  $\theta(x_i) = 1$ : wähle  $a_i$
- Falls  $\theta(x_i) = 0$ : wähle  $b_i$
- ➡ Die Summe der Zahlen ist im ersten Teil an jeder Position mindestens 1 und höchstens 3, im zweiten Teil genau 1
- Deshalb: falls in der bisherigen Summe im ersten Teil an der  $i$ -ten Stelle
  - eine 3 ist: wähle  $y_i$
  - eine 2 ist: wähle  $z_i$
  - eine 1 ist: wähle  $y_i$  und  $z_i$
- ➡ die Gesamtsumme ergibt genau  $k$

## Beweisskizze (Forts.)

(4)  $f(\varphi)$  hat eine Lösung  $\Rightarrow \varphi$  ist erfüllbar

- Sei eine Menge  $T \subseteq S$  mit Summe  $k$  gewählt
- ➡ Für jedes  $i$  ist entweder  $a_i$  oder  $b_i$  gewählt

→ Definiere eine Variablenbelegung wie folgt:

$$\theta(x_i) \stackrel{\text{def}}{=} \begin{cases} 1 & \text{falls } a_i \text{ gewählt ist} \\ 0 & \text{falls } b_i \text{ gewählt ist} \end{cases}$$

- Klar: die Summe der ausgewählten  $a_i$  und  $b_i$  hat an jeder vorderen Stelle mindestens 1
- ➡ Die Variablenbelegung macht jede einzelne Klausel und damit  $\varphi$  wahr

# 3-SAT $\leq_p$ SUBSETSUM $\leq_p$ KNAPSACK (6/6)

## Proposition 19.11

SUBSETSUM  $\leq_p$  KNAPSACK

## Beweisskizze

- Sei  $S$  eine Menge von Zahlen und  $k$  eine Zielzahl

- (1) Sei  $f(S, k)$  die Eingabe für KNAPSACK mit
- je einem Gegenstand mit Gewicht und Wert  $a$ ,  
für jede Zahl  $a \in S$ , und
  - Gewichts- und Wertschranke  $W$

- Es ist leicht zu sehen, dass
- (2)  $f$  in polynomieller Zeit berechnet werden kann,  
und
- (3,4)  $(S, k) \in \text{SUBSETSUM}$  genau dann gilt, wenn  
 $f(S, k) \in \text{KNAPSACK}$

# Gesamtergebnis

- Da wir für alle in diesem Kapitel betrachteten Probleme schon gezeigt haben, dass sie in **NP** sind, folgt aus den in diesem Kapitel gezeigten Resultaten der folgende Satz

## 19.12

- Die folgenden Probleme sind **NP**-vollständig:
  - 3-SAT
  - 3-COL
  - CLIQUE
  - HAMILTONCYCLE
  - TSP
  - SUBSETSUM
  - KNAPSACK

# Zusammenfassung

- Es gibt Tausende von **NP**-vollständigen Problemen
- Das erste **NP**-vollständige Problem, SAT, haben wir durch den Satz von Cook gewonnen
- **NP**-Vollständigkeitsbeweise verwenden zumeist polynomielle Reduktionen **von** anderen, schon als **NP**-vollständig bekannten, Problemen
- Solche Reduktionsbeweise lassen sich in einer kanonischen Struktur darstellen

# Literaturhinweise

**Satz von Cook:** Stephen A. Cook. The complexity of theorem-proving procedures. In *STOC*, pages 151–158, 1971

- Enthält viele weitere „klassische“ **NP**-vollständige Probleme

**Andere NP-vollständige Probleme** R.M. Karp. Reducibility among combinatorial problems. In R.E. Miller and J.W. Thatcher, editors, *Complexity of Computer Computations*. Plenum, New York, 1972

**Gerade Pfade:** A. S. Lapauagh and C. H. Papadimitriou. The even-path problem for graphs and digraphs. *Networks*, 14:507–513, 1984

- Die Arbeit zeigt, dass es ein **NP**-vollständiges Problem ist, zu überprüfen, ob es einen Weg gerader Länge zwischen zwei Knoten eines gerichteten Graphen gibt



# Grundbegriffe der Theoretischen Informatik

Sommersemester 2016 - Thomas Schwentick

Teil D: Komplexitätstheorie

24: **NP**: weitere Erkenntnisse

Version von: 7. Juli 2016 (14:09)

# Inhalt

## ▷ 24.1 NP-vollständige Probleme: weitere Bemerkungen

24.2 Die innere Struktur von **NP**

24.3 Pseudo-polynomiell vs. stark **NP**-vollständig

24.4 **P**, **NP** und so weiter

## Weitere NP-vollständige Probleme (1/2)

- Es sind buchstäblich Tausende von **NP**-vollständigen Problemen bekannt
- Eine schöne Auswahl findet sich in:
  - Garey, Johnsen: Computers and Intractability: A Guide to the Theory of **NP**-Completeness, 1979

- Wir betrachten im Folgenden kurz einige weitere **NP**-vollständige Probleme

### Definition: VERTEXCOVER

**Gegeben:** Ungerichteter Graph

$G = (V, E)$ , Zahl  $k$

**Frage:** Gibt es  $S \subseteq V$  mit  $|S| \leq k$ , so dass jede Kante einen Knoten in  $S$  hat?

### Definition: SHORTESTSUPERSTRING

**Gegeben:** Strings  $s_1, \dots, s_m$ , Zahl  $k$

**Frage:** Gibt es einen String der Länge  $k$ , der alle Strings  $s_i$  als Teilstrings enthält?

### Definition: BINPACKING

**Gegeben:** Gegenstände mit Größen

$s_1, \dots, s_n \in \mathbb{Q}$  zwischen 0 und 1, Zahl  $k$

**Frage:** Lassen sich die Gegenstände in  $k$  Behälter mit jeweiliger Kapazität 1 füllen?

### Definition: INTEGERPROGRAMMINGO

**Gegeben:** Lineares Ungleichungssystem, lineare Zielfunktion

**Gesucht:** Ganzzahlige Lösung des Systems mit maximalem Wert der Zielfunktion

### Definition: INTEGERPROGRAMMING

**Gegeben:** Lineares Ungleichungssystem

**Frage:** Gibt es eine ganzzahlige Lösung des Systems?

## Weitere NP-vollständige Probleme (2/2)

Definition: BUNDESLIGA

**Gegeben:** Tabellenstand eines Turnieres, Menge restlicher Spiele, Lieblingsverein  $x$

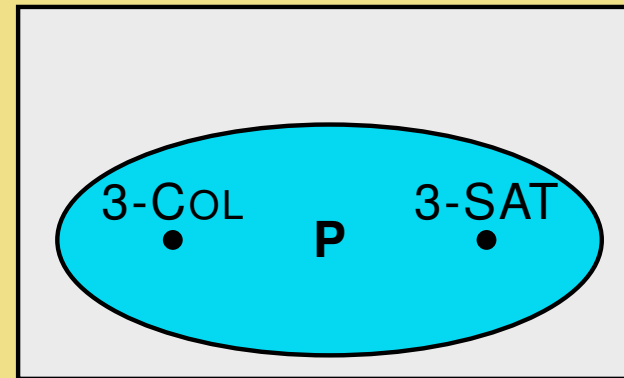
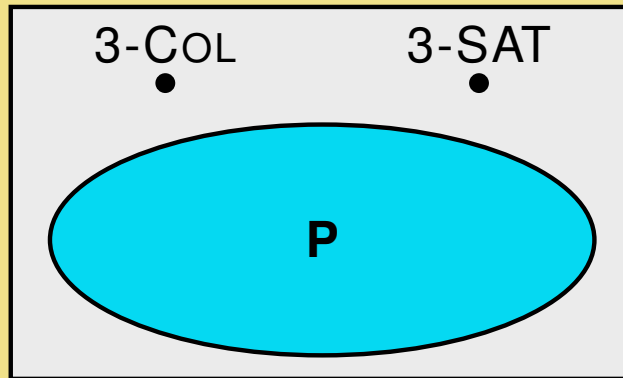
**Frage:** Kann  $x$  noch Meister werden?

- [Bernholt, Gülich, Hofmeister, Schmitt 99] zeigen:
  - **NP**-vollständig mit 3-Punkte-Regel
  - in **P** mit 2-Punkte-Regel

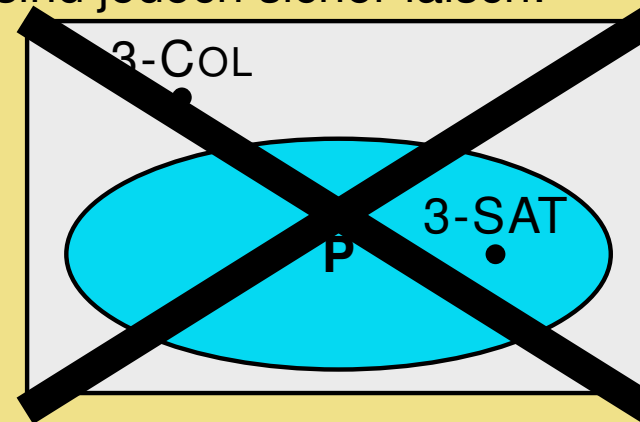
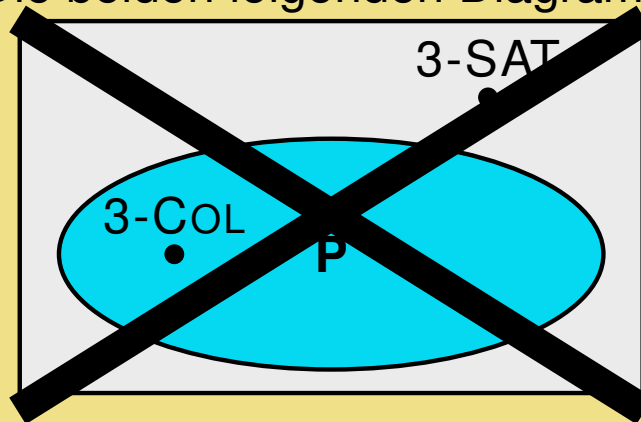
## Alles oder nichts...

- Aus Satz 22.4 folgt für 3-SAT und 3-COL:
  - Entweder sind beide in polynomieller Zeit lösbar oder
  - beide sind **nicht** in polynomieller Zeit lösbar

- Eines der beiden folgenden Diagramme ist also gültig:



- Die beiden folgenden Diagramme sind jedoch sicher falsch:



- Entsprechende Aussagen gelten für beliebige andere Paare **NP**-vollständiger Probleme

# NP-Vollständigkeit: weitere Anmerkungen

- **NP**-vollständige Probleme sind in einem gewissen Sinne sehr ähnlich:
  - Sind  $L_1$  und  $L_2$  **NP**-vollständig so gilt  $L_1 \leq_p L_2$  und  $L_2 \leq_p L_1$
- Aber **NP**-vollständige Probleme haben im Detail durchaus unterschiedliche Eigenschaften
- Manche lassen sich gut approximieren, andere nicht:
  - Für KNAPSACKO gibt es einen Algorithmus, der bei Eingabe  $(I, \epsilon)$  in polynomieller Zeit in  $|I|$  und  $1/\epsilon$  eine Lösung berechnet, die mindestens einen  $(1 - \epsilon)$ -Anteil des optimalen Wertes liefert
  - TSPO lässt sich im Gegensatz dazu im Allgemeinen gar nicht approximieren (falls  $P \neq NP$ )
- Manche **NP**-vollständige Probleme lassen sich effizient lösen, falls nur ein gewisser Parameter des Problems nicht zu groß wird:
  - VERTEXCOVER lässt sich beispielsweise leicht in Zeit  $\mathcal{O}(2^k n)$  lösen
- Andere **NP**-vollständige Probleme haben durch ausgefeilte Algorithmen für viele in der Praxis auftretende Eingaben ihren Schrecken verloren:
  - Inzwischen lässt sich für viele große „praktisch relevante“ KNF-Formeln die Erfüllbarkeit effizient testen
  - In manchen Anwendungen werden sogar polynomiell lösbare (aber praktisch ungünstige) Probleme durch Reduktion auf SAT und Anwendung eines effizienten SAT-Solvers gelöst
- Mehr darüber gibt es in der Vorlesung **Komplexitätstheorie** zu erfahren
  - ... immer im Sommersemester

# Inhalt

24.1 **NP**-vollständige Probleme: weitere Bemerkungen

▷ **24.2 Die innere Struktur von NP**

24.3 Pseudo-polynomiell vs. stark **NP**-vollständig

24.4 **P**, **NP** und so weiter

## Die innere Struktur von NP (1/2)

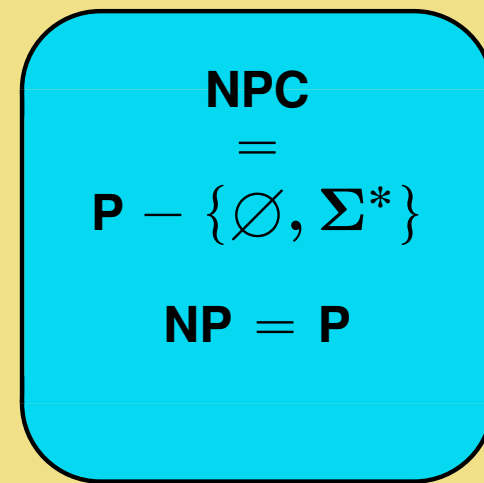
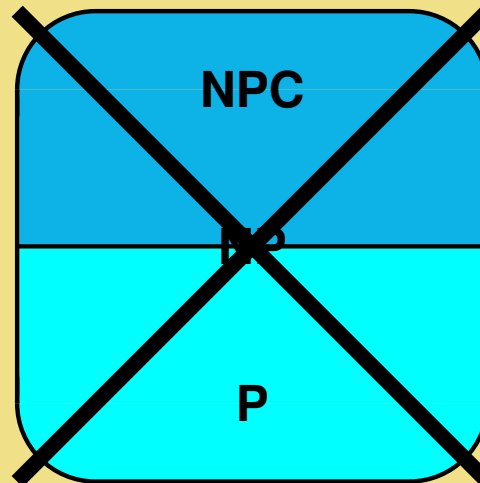
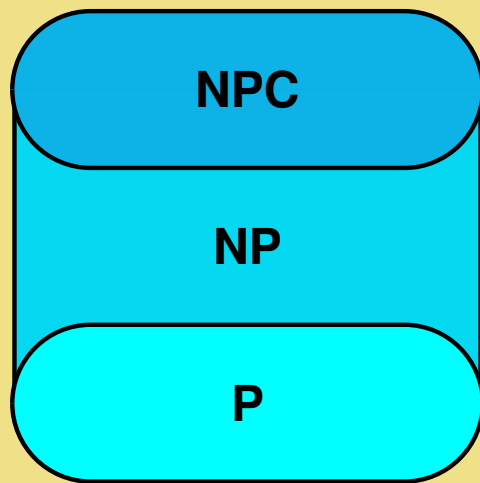
- Empirische Erfahrung: für die meisten Probleme  $L$  in **NP** lässt sich eine der beiden folgenden Aussagen zeigen:

(1)  $L \in P$

(2)  $L$  ist vollständig für **NP**

☞ in **NPC**

- Lässt sich das vielleicht allgemein beweisen?
- Anders gefragt: welche der folgenden Illustrationen der Struktur von **NP** könnten der Realität entsprechen?



- Es gilt: **NP** ist **nicht** die disjunkte Vereinigung von **P** und **NPC**
- Die Frage, welche der beiden verbleibenden Möglichkeiten die Richtige ist, ist gerade die Frage, ob  $P = NP$  gilt



## Die innere Struktur von NP (2/2)

### Weitere Bemerkungen

- Die Relation  $\leq_p$  induziert eine Äquivalenzrelation  $\equiv_p$  durch:

$$L \equiv_p L' \stackrel{\text{def}}{\Leftrightarrow} L \leq_p L' \text{ und } L' \leq_p L$$

- Die Sprachen aus **P** (außer  $\emptyset$  und  $\Sigma^*$ ) bilden die „einfachste“ Äquivalenzklasse von **NP**
- Die **NP**-vollständigen Sprachen bilden die „schwierigste“ Äquivalenzklasse innerhalb von **NP**
- Es gilt: falls **P**  $\neq$  **NP** gibt es noch unendlich viele weitere  $\equiv_p$ -Klassen

# Inhalt

24.1 **NP**-vollständige Probleme: weitere Bemerkungen

24.2 Die innere Struktur von **NP**

▷ **24.3 Pseudo-polynomiell vs. stark NP-vollständig**

24.4 **P**, **NP** und so weiter

# Ein effizienter (?) Algorithmus für KNAPSACKO


- KNAPSACKO lässt sich mit **dynamischer Programmierung** lösen

- Gegeben:  $w_1, \dots, w_m, g_1, \dots, g_m, G$ 
  - OBdA:  $w_1 \geq w_i$  für alle  $i \in \{1, \dots, m\}$
- ➡ der optimale Wert ist  $\leq mw_1$


- Es bezeichne  $A(i, w)$  das minimale Gewicht einer Menge  $I \subseteq \{1, \dots, i\}$  von Gegenständen mit Gesamtwert  $w$
- **Ansatz:** Berechne induktiv, für jedes  $i \leq m$  und jedes  $w \leq w_1$ , den Wert  $A(i, w)$
- Der optimale Wert ist dann das maximale  $w$ , das  $A(m, w) \leq G$  erfüllt

- **Entscheidende Beobachtung:**  $A(i, w)$  ist das Minimum der beiden folgenden Werte:

- $A(i - 1, w)$

-  Das entspricht dem Weglassen von Gegenstand  $i$

- $A(i - 1, w - w_i) + g_i$

-  Das entspricht dem Hinzunehmen von Gegenstand  $i$

- ➡ Die Werte  $A(i, w)$  lassen sich leicht berechnen, wenn die Werte  $A(i - 1, j)$  schon für alle  $j$  berechnet sind

- Initialisierung:

- $A(0, 0) = 0$

- $A(0, w) = \infty$ , für alle  $w > 0$

- Dynamische Programmierung:

$m^2 w_1$  Tabelleneinträge berechnen

- Aufwand:  $\mathcal{O}(m^2 w_1)$


# Pseudo-polynomiell vs. stark NP-vollständig (1/2)

- Der Algorithmus für KNAPSACKO hat also Laufzeit  $\mathcal{O}(m^2 w_1)$
- Ist das ein Polynomialzeit-Algorithmus? **Nein!**

- Ein Polynomialzeit-Algorithmus hat eine Laufzeitschranke der Form  $n^k$  für Eingaben der Länge  $n$

- Länge einer (binär kodierten) KNAPSACKO-Eingabe:

$$\log(G) + \sum_{i=1}^m (\log w_i + \log g_i)$$

 Abweichungen durch etwas andere Kodierungen sind hier irrelevant

- Die Laufzeitschranke  $\mathcal{O}(m^2 w_1)$  ist polynomiell in  $m$  und dem Wert von  $w_1$

- Die Laufzeitschranke des KNAPSACKO-Algorithmus ist aber nicht polynomiell in der **Länge der Kodierung** von  $w_1$ :
  - Die Laufzeitschranke lässt sich schreiben als  $\mathcal{O}(m^2 2^{\log w_1})$
  - Sie ist also exponentiell in der Länge der Eingabe

- Einen Algorithmus, der polynomiell in den *Größen* (statt: der Länge der Kodierung) der vorkommenden Zahlen ist, nennen wir **pseudo-polynomiell**

## Pseudo-polynomiell vs. stark NP-vollständig (2/2)

- Umgekehrt heißt ein Problem, das **NP**-vollständig bleibt, selbst wenn die Größe der vorkommenden Zahlen durch ein Polynom in der Länge der Eingabe beschränkt sind, **stark NP-vollständig**
- Beispiel: TSP ist stark **NP**-vollständig:
  - es ist auch schon für Eingaben **NP**-vollständig, bei denen alle Abstände den Wert **1** oder **2** haben und  $k = |V|$
- Gäbe es für ein stark **NP**-vollständiges Problem einen pseudo-polynomiellen Algorithmus, wäre **P = NP**
- ✎ Es ist also kein Wunder, dass in der Reduktion  $3\text{-SAT} \leq_p \text{KNAPSACK}$  die verwendeten Zahlen so groß sind
- ✎ Zu beachten: bei der genauen Formalisierung der Begriffe „pseudo-polynomiell“ und „stark **NP**-vollständig“ ist etwas Vorsicht geboten:
  - Es müsste sauber definiert werden, was „vorkommende Zahlen“ bedeutet...

# Inhalt

24.1 **NP**-vollständige Probleme: weitere Bemerkungen

24.2 Die innere Struktur von **NP**

24.3 Pseudo-polynomiell vs. stark **NP**-vollständig

▷ **24.4 P, NP und so weiter**

# Die Welt der Komplexitätstheorie

- Wir haben in Teil D drei Komplexitätsklassen betrachtet: **P**, **NP** und **EXPTIME**
- Alle algorithmischen Probleme, die wir in Teil D betrachtet haben, sind in **EXPTIME**, die meisten davon in **NP**
- Es stellen sich offensichtliche Fragen:
  - Sind alle entscheidbaren Probleme in **EXPTIME**?
  - Sind alle „natürlichen“ entscheidbaren Probleme in **EXPTIME**?
  - Gibt es außer den drei Klassen noch andere relevante Komplexitätsklassen?
  - Haben andere Klassen auch vollständige Probleme?

## Rechenzeit vs. Speicherplatz

- Für manche Berechnungen ist Speicherplatz eine kritischere Ressource als Rechenzeit
- Die algorithmische Schwierigkeit mancher algorithmischen Probleme lässt sich besser durch ihren Speicherplatzverbrauch als durch ihre Rechenzeit charakterisieren

### Definition: KORREKTERAUTOMAT

**Gegeben:** Endlicher Automat  $\mathcal{A}$ , regulärer Ausdruck  $\alpha$

**Frage:** Erfüllt der Automat  $\mathcal{A}$  die Spezifikation  $\alpha$ ?

- Dieses Problem ist vermutlich nicht in **NP**
- Es ist aber vollständig für die Komplexitätsklasse **PSPACE**, die alle Probleme enthält, die für Eingaben der Größe  $n$  mit Speicherplatz  $p(n)$ , für ein Polynom  $p$ , gelöst werden können
- Es gilt: **NP**  $\subseteq$  **PSPACE**  $\subseteq$  **EXPTIME**



# Polynomielle Hierarchie (1/2)

- Zur Erinnerung:
  - Eine Sprache  $L$  ist genau dann semientscheidbar, wenn es eine entscheidbare Sprache  $L'$  gibt, so dass für alle Strings  $w \in \Sigma^*$  gilt:
$$w \in L \iff \exists x_1 : (w, x_1) \in L',$$
wobei der Existenzquantor über alle  $\Sigma^*$ -Strings quantifiziert

## Satz 24.1

- Eine Sprache  $L$  ist genau dann in **NP**, wenn es eine Sprache  $L' \in \mathbf{P}$  und ein Polynom  $p$  gibt, so dass für alle Strings  $w \in \Sigma^*$  gilt:
$$w \in L \iff \exists^p x_1 : (w, x_1) \in L',$$
wobei der modifizierte Existenzquantor  $\exists^p$  über alle  $\Sigma^*$ -Strings der Länge  $p(|w|)$  quantifiziert
- Analog zur arithmetischen Hierarchie gibt es auch hier eine Hierarchie von vermutlich nicht in polynomieller Zeit entscheidbaren Problemen

## Polynomielle Hierarchie (2/2)

- Die **polynomielle Hierarchie** besteht aus allen Sprachen  $L$ , für die es ein  $k \in \mathbb{N}$ , eine Sprache  $L' \in \mathbf{P}$  und ein Polynom  $p$  gibt, so dass gilt:  
 $w \in L \iff$   
 $\exists^p x_1 \forall^p x_2 \exists^p x_3 \dots \forall^p x_k :$   
 $(w, x_1, x_2, \dots, x_k) \in L'$ ,  
wobei alle Quantoren über die  $\Sigma^*$ -Strings der Länge  $p(|w|)$  quantifizieren
- Die Klasse der Probleme, die sich für ein bestimmtes  $k$  auf diese Weise charakterisieren lassen, wird  $\Sigma_k^p$  genannt
  - Analog:  $\Pi_k^p$ , wenn der erste Quantor in der Charakterisierung ein Allquantor ist
- $\Sigma_1^p = \mathbf{NP}$ ,  $\Pi_1^p = \mathbf{co-NP}$

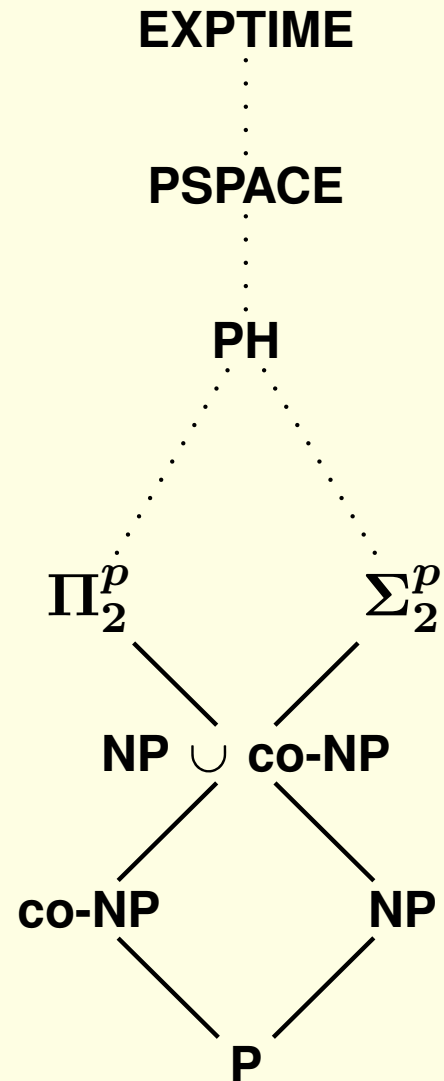
### Definition: UCQ-CONT

**Gegeben:** Select-From-Where-Anfragen  $q_1, q_2$  (mit UNION)

**Frage:** Liefert  $q_1$  für alle passenden relationalen Datenbanken ein Teilergebnis von  $q_2$ ?

- UCQ-CONT ist vollständig für  $\Pi_2^p$

# Übersicht



# Weit jenseits des Effizienten

- Ein regulärer Ausdruck mit Negation darf neben den üblichen Operatoren auch einen Negationsoperator  $\neg$  verwenden
- Die Semantik-Definition wird dann erweitert durch:
  - $L(\neg\alpha) \stackrel{\text{def}}{=} \Sigma^* - L(\alpha)$

## Definition: RENEGEMPTY

**Gegeben:** Regulärer Ausdruck  $\alpha$  mit Negation

**Frage:** Ist  $L(\alpha) = \emptyset$ ?

- Mit den Methoden aus Teil A der Vorlesung ist es nicht schwer zu zeigen:
  - RENEGEMPTY ist entscheidbar

- Eine Funktion  $T : \mathbb{N} \rightarrow \mathbb{R}$  ist (1-fach) exponentiell beschränkt, wenn  $T(n) = 2^{\mathcal{O}(p(n))}$  für ein Polynom  $p$
- Eine Funktion ist  $k$ -fach exponentiell beschränkt, wenn  $T(n) = 2^{\mathcal{O}(g)}$ , für eine  $(k-1)$ -fach exponentiell beschränkte Funktion
- Eine Funktion heißt elementar, wenn sie, für irgendein  $k \in \mathbb{N}$ ,  $k$ -fach exponentiell beschränkt ist

## Satz [Stockmeyer, Meyer 73]

- Es gibt keinen Algorithmus für RENEGEMPTY mit einer elementaren Laufzeitschranke

# Zusammenfassung

- Die **NP**-vollständigen Probleme sind die schwierigsten Probleme in **NP**:
  - Jedes **NP**-Problem lässt sich polynomiell auf jedes **NP**-vollständige Problem reduzieren
- Es gibt Tausende von **NP**-vollständigen Problemen
- Pseudo-polynomielle Algorithmen lösen Probleme in polynomieller Zeit in Abhängigkeit von der Größe der vorkommenden Zahlen
- Für stark **NP**-vollständige Probleme lässt sich die **NP**-Vollständigkeit ohne die Verwendung „großer Zahlen“ zeigen
- Es gibt viele algorithmische Probleme, die für andere Klassen als **P** und **NP** vollständig sind

# Literaturhinweise

- Angegebene Bücher
- Thorsten Bernholt, Alexander Gülich, Thomas Hofmeister, and Niels Schmitt. Football elimination is hard to decide under the 3-point-rule. In *MFCS*, pages 410–418, 1999
- Larry J. Stockmeyer and Albert R. Meyer. Word problems requiring exponential time: Preliminary report. In *Proceedings of the 5th Annual ACM Symposium on Theory of Computing, April 30 - May 2, 1973, Austin, Texas, USA*, pages 1–9, 1973

# Grundbegriffe der Theoretischen Informatik

Sommersemester 2016 - Thomas Schwentick

Teil D: Komplexitätstheorie

21: Zufallsbasierte Algorithmen

Version von: 12. Juli 2016 (10:41)

# Einleitung

- In diesem Kapitel betrachten wir drei zufallsbasierte Algorithmen
  - Das sind Algorithmen, die Zufallsbits verwenden, um ein Problem zu lösen
  - Sie lösen ein Problem also nur mit einer gewissen Wahrscheinlichkeit und/oder ihre Laufzeit lässt sich nur mit einer gewissen Wahrscheinlichkeit beschränken
- Wir betrachten Algorithmen für
  - 3-SAT
  - das Problem, ob eine gegebene Zahl eine Primzahl ist
  - das Problem zu testen, ob ein arithmetischer Schaltkreis immer 0 ausgibt



## 21.1 Zufallsbasierte Algorithmen für

### ▷ 21.1.1 3-SAT

21.1.2 Primzahlen

21.1.3 Arithmetischer Schaltkreise

# Ein zufallsbasierter Algorithmus für 3-SAT

- Naiver Algorithmus für SAT: Probiere alle  $2^n$  Belegungen der  $n$  Variablen der Eingabeformel  $\varphi$  aus  
→ Laufzeit  $\theta(|\varphi|2^n)$

- Hier betrachten wir einen einfachen zufallsbasierter Algorithmus für 3-SAT mit Laufzeit  $\mathcal{O}(1.334^n)$

- Typisch für zufallsbasierte Algorithmen:
  - einfacher Algorithmus
  - komplizierte Analyse

## Algorithmus 21.1 [Schöning]

**Eingabe:**  $\varphi$  mit Variablen  $x_1, \dots, x_n$

```
1:  $\gamma(n) := \lceil 70\sqrt{n}(\frac{4}{3})^n \rceil$ 
2: for  $\gamma(n)$  mal do
3:   Wähle zufällig eine Belegung  $\alpha$  der Variablen
4:   for  $3n$  mal do
5:     if  $\alpha \models \varphi$  then
6:       Ausgabe „ja“, Fertig
7:     else
8:       Zufallsschritt
9: Ausgabe „nein“
```

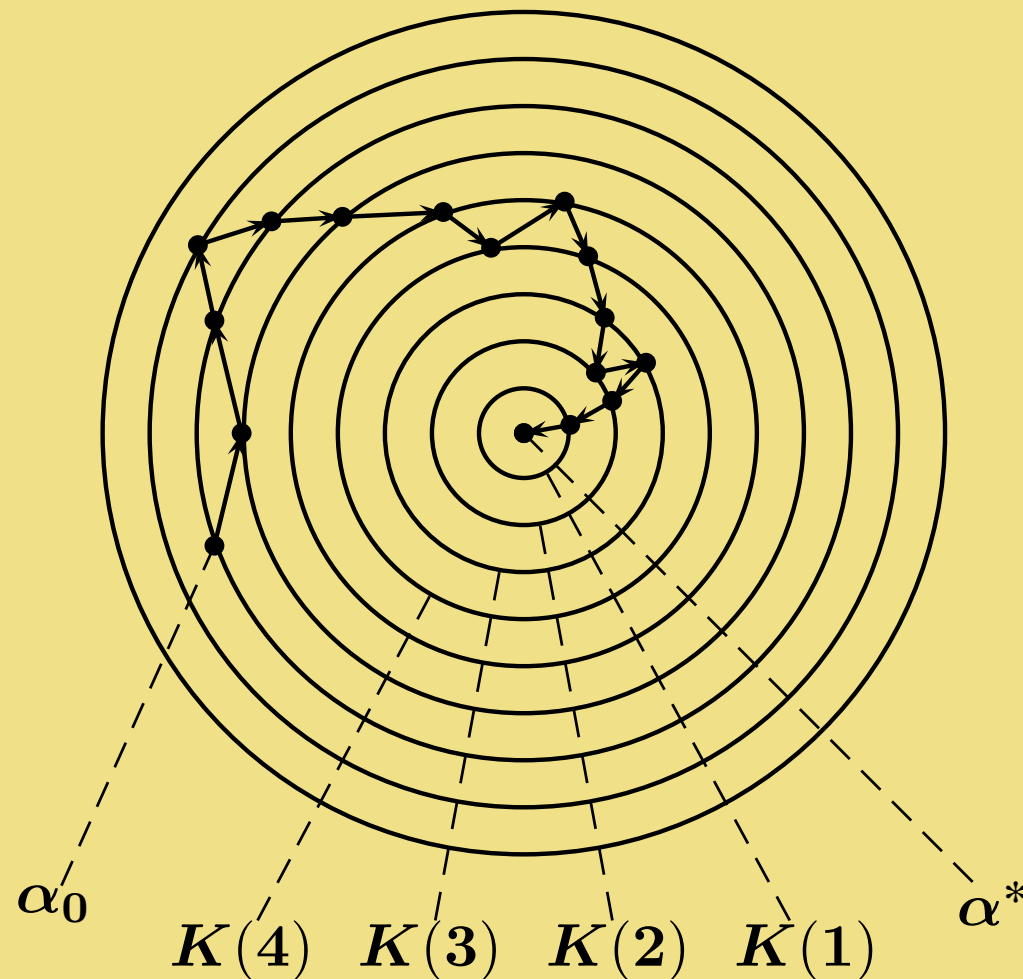
## Algorithmus Zufallsschritt

```
1: Wähle zufällig eine Klausel  $C$ , die von  $\alpha$  nicht wahr gemacht wird
2: Wähle zufällig eine Variable  $x_k$  in  $C$ 
3: Ändere  $\alpha$  durch:  $\alpha(x_k) := 1 - \alpha(x_k)$ 
```

- Wir nennen jeden der  $\gamma(n)$  Durchläufe der äußeren Schleife einen **Versuch**

# Schönings Algorithmus: Illustration

Menge der Belegungen von  $\varphi$



- $\alpha^*$ : Erfüllende Belegung von  $\varphi$
- $K(j)$ : Menge der Belegungen mit Hamming-Abstand  $j$  von  $\alpha^*$
- $\alpha_0$ : Zufällig gewählte erste Belegung

# Ein zufallsbasierter Algorithmus für 3-SAT (Forts.)

## Satz 21.2 [Schöning 99]

- (a) Für erfüllbare Formeln findet Algorithmus 21.1 mit Wahrscheinlichkeit  $> 0,9999$  eine erfüllende Belegung
- (b) Für unerfüllbare Formeln gibt der Algorithmus „unerfüllbar“ aus
- (c) Die Laufzeit ist  $\mathcal{O}(|\varphi| n^{3/2} (\frac{4}{3})^n)$

## Beweisskizze

- (b) und (c) sind klar
- Wir zeigen nun (a)
- Sei  $\varphi$  eine 3KNF-Formel und  $\alpha^*$  eine erfüllende Belegung von  $\varphi$
- Sei  $\underline{p}$  die Wahrscheinlichkeit, dass der Algorithmus in einem einzelnen Versuch  $\alpha^*$  findet (oder vorher auf eine andere erfüllende Belegung stößt)
- **Behauptung (A):**  $p \geq \frac{1}{7\sqrt{n}} \left(\frac{3}{4}\right)^n$

## Beweisskizze (Forts.)

### • Notation:

- $\underline{\alpha_0} \stackrel{\text{def}}{=} \text{die in Zeile 3 eines Versuches gewählte Belegung}$
- $\underline{K(j)} \stackrel{\text{def}}{=} \text{Menge aller Belegungen } \alpha \text{ mit } d(\alpha, \alpha^*) = j$   
 $\Rightarrow d(\cdot, \cdot): \text{Hamming-Abstand}$
- $\underline{p_j} \stackrel{\text{def}}{=} \text{W-keit, dass } \alpha_0 \in K(j) \text{ gilt}$
- $\underline{q_j} \stackrel{\text{def}}{=} \text{W-keit, dass } \alpha^* \text{ von einem } \alpha_0 \in K(j) \text{ in } \leq 3j \text{ Schritten erreicht wird oder der Algorithmus vorher eine andere Lösung findet}$

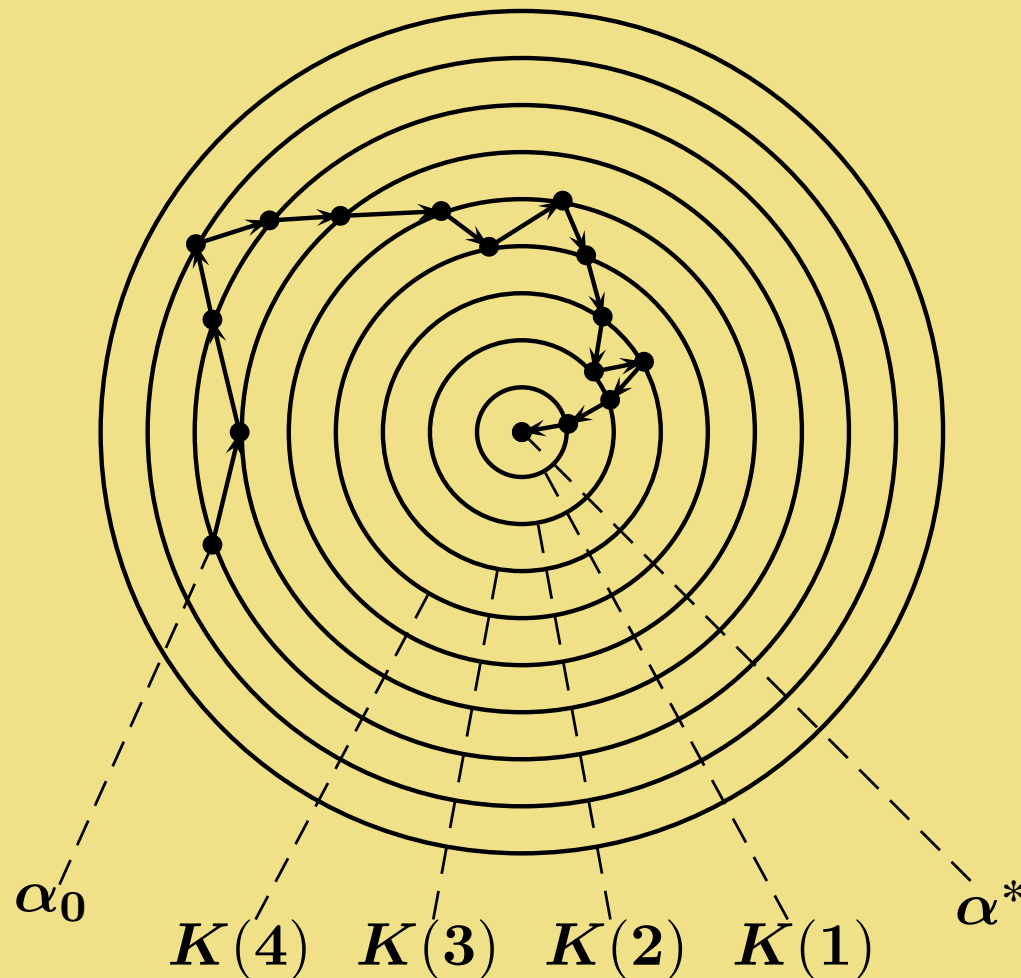
### • Klar:

$$p = \sum_{j=0}^n p_j q_j$$
$$p_j = \binom{n}{j} / 2^n$$

- **Behauptung (B):**  $q_j \geq \frac{1}{7\sqrt{j}} \left(\frac{1}{2}\right)^j$

# Schönings Algorithmus: Illustration (Wdh.)




Menge der Belegungen von  $\varphi$



- $\alpha^*$ : Erfüllende Belegung von  $\varphi$
- $K(i)$ : Menge der Belegungen mit Hamming-Abstand  $i$  von  $\alpha^*$
- $\alpha_0$ : Zufällig gewählte erste Belegung

# Beweis von Behauptung (B) (1/2)

## Beweisskizze

- Zur Erinnerung:
  - $q_j$ : W-keit, dass  $\alpha^*$  von einem  $\alpha_0 \in K(j)$  in  $\leq 3j$  Schritten erreicht wird, oder der Alg. vorher eine andere Lösung findet
  - **Behauptung (B)**:  $q_j \geq \frac{1}{7\sqrt{j}} \left(\frac{1}{2}\right)^j$
- Für  $i \leq j$  sei  $q_{j,i}$  die W-keit, dass der Alg. von  $\alpha_0 \in K(j)$  aus in  $2i + j$  Schritten  $\alpha^*$  erreicht, oder vorher eine andere erfüllende Belegung erreicht
- Ein Weg, der in  $2i + j$  Schritten von  $\alpha_0 \in K(j)$  zu  $\alpha^*$  führt, macht  $i + j$  Schritte auf  $\alpha^*$  zu und  $i$  Schritte von  $\alpha^*$  weg
- Wir ordnen jedem solchen Weg einen String der Länge  $2i + j$  über dem Alphabet  $\{-, +\}$  zu   $+$ : auf  $\alpha^*$  zu,  $-$ : von  $\alpha^*$  weg
-  Jedes Suffix des Strings muss dabei mindestens soviele  $+$  wie  $-$  enthalten
- Die Anzahl der Strings dieser Form ist  $\binom{2i+j}{i} \frac{j}{2i+j}$   ohne Beweis


# Beweis von Behauptung (B) (2/2)

## Beweisskizze (Forts.)

- Die W-Keit, in einem Schritt näher an  $\alpha^*$  zu kommen, ist  $\geq \frac{1}{3}$ , denn:
  - Da  $C$  von  $\alpha^*$  wahr gemacht wird, von dem jeweils aktuellen  $\alpha$  aber nicht, gibt es mindestens eine Variable von  $C$  für die  $\alpha$  und  $\alpha^*$  sich unterscheiden
  - Wird die Belegung dieser Variablen in  $\alpha$  geändert verringert sich der Abstand zu  $\alpha^*$  um 1

$$\Rightarrow q_{j,i} \geq \binom{2i+j}{i} \frac{j}{2i+j} \left(\frac{1}{3}\right)^{i+j} \left(\frac{2}{3}\right)^i$$

$$\begin{aligned} \Rightarrow q_j &= \sum_{i=0}^j q_{j,i} \\ &\geq \frac{1}{3} \sum_{i=0}^j \binom{2i+j}{i} \left(\frac{1}{3}\right)^{i+j} \left(\frac{2}{3}\right)^i \quad \text{☞ } i \leq j \\ &\geq \frac{1}{2\sqrt{3\pi j}} \left(\frac{1}{2}\right)^j \geq \frac{1}{7\sqrt{j}} \left(\frac{1}{2}\right)^j \Rightarrow (B) \end{aligned}$$

 Die letzte Zeile wird durch eine Reihe weiterer Umformungen erreicht, die unter anderem die Stirling-Approximations-Formel für die Fakultätsfunktion verwenden

# Beweis von Satz 21.2

## Beweisskizze

- Also:

$$\begin{aligned} p &\geq \sum_{j=0}^n p_j q_j \\ &\geq \sum_{j=0}^n \binom{n}{j} \left(\frac{1}{2}\right)^n \frac{1}{7\sqrt{j}} \left(\frac{1}{2}\right)^j \\ &\geq \frac{1}{7\sqrt{n}} \left(\frac{1}{2}\right)^n \sum_{j=0}^n \binom{n}{j} \left(\frac{1}{2}\right)^j \\ &= \frac{1}{7\sqrt{n}} \left(\frac{1}{2}\right)^n \left(1 + \frac{1}{2}\right)^n && \text{Binomischer Lehrsatz} \\ &= \frac{1}{7\sqrt{n}} \left(\frac{3}{4}\right)^n && (=:\tilde{p}) \end{aligned}$$

➡ Behauptung (A)

- Die W-keit, dass in einem Versuch weder  $\alpha^*$  noch eine andere erfüllende Belegung gefunden wird ist also  $\leq 1 - \tilde{p}$
- Die W-keit, dass dies  $\gamma(n)$ -mal passiert ist

$$\leq (1 - \tilde{p})^{\gamma(n)}$$

$\vdots$

$$\leq e^{-10} < 5 \cdot 10^{-5}$$

➡ (a)



# SAT-Solving

- Relativ effiziente Algorithmen für das SAT-Problem zu finden ist ein sehr wichtiges Forschungsthema
- Denn: viele andere algorithmischen Probleme lassen sich leicht auf die Erfüllbarkeit einer aussagenlogischen Formel zurückführen
- Anders gesagt: Reduktionen auf SAT sind häufig einfach
- Durch verfeinerte Techniken werden die Laufzeitschranken für SAT-Algorithmen kontinuierlich verbessert
- Einer der aktuell besten deterministischen Algorithmen für 3-SAT ist durch **Derandomisierung** von Schönings Algorithmus entstanden  
[Moser, Scheder 2011]
- Seine Laufzeit ist  $\mathcal{O}(1.334^n)$
- Noch besser:  $\mathcal{O}(1.3303^n)$   
[Makino, Tamaki, Yamamoto 2011]
- Der beste zufallsbasierte Algorithmus hat Laufzeit  $\mathcal{O}(1.30704^n)$   
[Hertli 2011]
- Für praktische Anwendungen werden SAT-Solver heute in der Industrie „Routine-mäßig“ eingesetzt und haben für „typische“ Formeln eine gute Laufzeit

## 21.1 Zufallsbasierte Algorithmen für

21.1.1 3-SAT

▷ **21.1.2 Primzahlen**

21.1.3 Arithmetischer Schaltkreise

## Primzahltests: Vorbereitung (1/2)

- Zur Erinnerung: eine **Primzahl** ist eine Zahl  $p \in \mathbb{N}$ , die außer **1** und  $p$  keine Teiler hat

### Notation

- $k \mid n \stackrel{\text{def}}{\Leftrightarrow} k$  ist Teiler von  $n$ , d.h.: es gibt ein  $l \in \mathbb{N}$  mit  $kl = n$ 
  - $3 \mid 9, \quad 4 \nmid 9, \quad 6 \nmid 9$
- $\text{ggT}(n, m) \stackrel{\text{def}}{=}$  größter gemeinsamer Teiler von  $n$  und  $m$ , also die größte Zahl  $k$  mit  $k \mid n$  und  $k \mid m$ 
  - $\text{ggT}(6, 9) = 3, \quad \text{ggT}(5, 7) = 1$
- $m \bmod n \stackrel{\text{def}}{=}$  Rest bei Division von  $m$  durch  $n$ , also die eindeutig bestimmte Zahl  $k$  mit:
  - (1)  $n \mid (m - k)$  und
  - (2)  $0 \leq k < n$ 
    - Beispiel:  $49 \bmod 9 = 4$
- $a \equiv_n b \stackrel{\text{def}}{\Leftrightarrow} a$  und  $b$  haben den selben Rest bei Division durch  $n$ , also  $a \bmod n = b \bmod n$   $\Leftrightarrow n \mid (a - b)$ 
  - Beispiel:  $7 \equiv_3 4$

## Primzahltests: Vorbereitung (2/2)

### Definition: PRIMES

**Gegeben:** Zahl  $N$

**Frage:** Ist  $N$  eine Primzahl?

### Definition: COMPOSITES

**Gegeben:** Zahl  $N$

**Frage:** Ist  $N$  eine zusammengesetzte Zahl?

- Klar: COMPOSITES  $\in$  NP

- Zu beachten:
  - Die **Länge  $n$  der Eingabe** ist jeweils  $\lceil \log_2 N \rceil$

- Naiver Algorithmus für PRIMES:
  - Für alle  $k \leq \sqrt{N}$ :
    - \* Teste ob  $k \mid N$
    - \* Falls ja, ablehnen
  - Falls Test ok, für alle  $k$ : akzeptieren

- Aufwand, falls  $N$  Primzahl ist:
$$\theta(\sqrt{N}) = \theta(2^{\frac{1}{2} \log_2 N}) = \theta(2^{\frac{n}{2}})$$
Tests
  - ✎ Also: exponentieller Aufwand

- Das geht besser, z.B. mit Hilfe des Zufalls...

# Fermats Primzahltest

## Satz 21.3 [Kleiner Satz von Fermat]

- Ist  $N$  eine Primzahl, so gilt für alle  $a$ ,  $1 \leq a < N$ :

$$a^{N-1} \equiv_N 1$$

## Beweisidee

- Wenn  $N$  eine Primzahl ist, bilden die Zahlen  $1, \dots, N-1$  mit der Operation  $(x, y) \mapsto xy \bmod N$  eine multiplikative Gruppe mit  $N-1$  Elementen
- In einer endlichen Gruppe  $G$  gilt für jedes Element  $g$ :  $g^{|G|} = 1$

- Was ist, wenn  $N$  keine Primzahl ist?

|                    |                  |
|--------------------|------------------|
| – $1^8 \equiv_9 1$ | $2^8 \equiv_9 4$ |
| – $3^8 \equiv_9 0$ | $4^8 \equiv_9 7$ |
| – $5^8 \equiv_9 7$ | $6^8 \equiv_9 0$ |
| – $7^8 \equiv_9 4$ | $8^8 \equiv_9 1$ |

- Für 75% der möglichen Werte für  $a$  würde durch den Test „ $a^{9-1} \equiv_9 1$ ?“ erkannt werden, dass **9** keine Primzahl ist

- Lässt sich daraus ein zufallsbasierter Primzahltest konstruieren?

(1) Wähle zufällig  $a$ ,  $1 \leq a < N$

(2) Falls  $a^{N-1} \equiv_N 1$ , akzeptiere

- Fehler-Wahrscheinlichkeiten für andere zusammengesetzte Zahlen  $N$ :

|     |       |
|-----|-------|
| 4   | 33%   |
| 6   | 20%   |
| 8   | 14,3% |
| 10  | 11%   |
| 12  | 9,1%  |
| 14  | 7,7%  |
| 16  | 13,3% |
| 18  | 6%    |
| ⋮   | ⋮     |
| 561 | 99,1% |

→ Wir brauchen einen verbesserten Ansatz

 **561** ist die kleinste *Carmichael-Zahl*:


- Sie ist zusammengesetzt:  $3 \times 11 \times 17$
- Für zu **561** teilerfremde  $a$  gilt:  $a^{560} \equiv_{561} 1$

# Primzahltest von Solovay-Strassen: (1/3)

- Der Primzahltest von Solovay-Strassen verwendet das **Jakobi-Symbol**  $\left(\frac{a}{b}\right)$

## Definition

- Eine ganze Zahl  $a$  ist **quadratischer Rest modulo  $p$** , für eine Primzahl  $p$ ,  $p \nmid a$ , falls es eine Zahl  $c$  gibt mit  $c^2 \equiv_p a$
- Ist  $p$  eine Primzahl, so sei
$$\left(\frac{a}{p}\right) \stackrel{\text{def}}{=} \begin{cases} 1, & \text{wenn } a \text{ quadratischer Rest modulo } p \text{ ist,} \\ 0 & \text{falls } p \mid a \\ -1 & \text{andernfalls} \end{cases}$$
- Ist  $p_1^{n_1} \times \dots \times p_k^{n_k}$  die Primfaktorzerlegung der Zahl  $b$ , so sei  $\left(\frac{a}{b}\right) \stackrel{\text{def}}{=} \prod_{i=1}^k \left(\frac{a}{p_i}\right)^{n_i}$

 Für Primzahlen  $p$  heißt  $\left(\frac{a}{p}\right)$  auch Legendre-Symbol

- Grundlage für den Solovay-Strassen-Test:

## Proposition 21.4

- $a^{(N-1)/2} \equiv_N \left(\frac{a}{N}\right)$  gilt
  - für alle  $a \in \{1, \dots, N-1\}$ , falls  $N \neq 2$  eine Primzahl ist;
  - für höchstens die Hälfte aller zu  $N$  teilerfremden  $a \in \{1, \dots, N-1\}$ , falls  $N \neq 2$  keine Primzahl ist
- Daraus lässt sich direkt ein Primzahltest gewinnen, der
  - Primzahlen immer erkennt
  - bei zusammengesetzten Zahlen eine Fehlerwahrscheinlichkeit  $\leq \frac{1}{2}$  hat
- Es stellt sich allerdings die Frage, wie sich  $\left(\frac{a}{N}\right)$  berechnen lässt, da die Definition die Primfaktorzerlegung von  $N$  verwendet...

## Primzahltest von Solovay-Strassen: (2/3)

- Zur Berechnung von  $\left(\frac{a}{N}\right)$  können wir die folgenden Regeln anwenden

1.  $\left(\frac{a}{b}\right) = \left(\frac{a \bmod b}{b}\right)$ , falls  $a > b$
2.  $\left(\frac{a_1 a_2}{b}\right) = \left(\frac{a_1}{b}\right) \left(\frac{a_2}{b}\right)$
3.  $\left(\frac{a}{b}\right) = (-1)^{(a-1)(b-1)/4} \left(\frac{b}{a}\right)$ ,  
falls  $a < b$  und  $a$  und  $b$  ungerade sind
4.  $\left(\frac{1}{b}\right) = 1$
5.  $\left(\frac{2}{b}\right) = (-1)^{(b^2-1)/8}$
6.  $\left(\frac{b-1}{b}\right) = (-1)^{(b-1)/2}$ ,  
falls  $b$  ungerade



Regel 3 wird auch quadratisches Reziprozitätsgesetz genannt

- Rekursive Berechnung von  $\left(\frac{a}{b}\right)$ :
  - Falls  $\text{ggT}(a, b) > 1$ : Ergebnis = 0
  - Falls  $a > b$ : verwende Regel 1
  - Falls  $a$  oder  $b$  gerade: Spalte mit Regel 2 und der Definition Zweierpotenzen ab
  - Falls  $a \in \{1, 2, b-1\}$ : verwende die entsprechende Regel (4-6)
  - Andernfalls ( $a < b$ ): verwende Regel 3

### Beispiel

$$\begin{aligned} \bullet \left(\frac{18}{11}\right) &\stackrel{1.}{=} \left(\frac{7}{11}\right) \stackrel{3.}{=} (-1)^{6 \cdot 10/4} \left(\frac{11}{7}\right) \\ &= - \left(\frac{11}{7}\right) \stackrel{1.}{=} - \left(\frac{4}{7}\right) \stackrel{2.}{=} - \left(\frac{2}{7}\right)^2 \stackrel{5.}{=} -1 \end{aligned}$$

- Bemerkung: Die Vorgehensweise ist also ähnlich zur Berechnung des ggT

### Fakt

- $\left(\frac{a}{b}\right)$  lässt sich in Poly-Zeit in der Länge der Kodierung von  $a$  und  $b$  berechnen

# Primzahltest von Solovay-Strassen (3/3)

## Algorithmus 21.5 (Solovay-Strassen-Test)

Eingabe:  $N$

1. Wähle zufällig  $a < N$
2. Falls  $\text{ggT}(a, N) > 1$  lehne ab
3. Akzeptiere, falls  $a^{(N-1)/2} \equiv_N \left( \frac{a}{N} \right)$

- Eigenschaften des Solovay-Strassen-Tests:
  - Laufzeit: polynomiell in  $\log_2 N$
  - Falls  $N \in \text{PRIMES}$ : Wahrscheinlichkeit einer fehlerhaften Antwort: 0
  - Falls  $N \notin \text{PRIMES}$ : Wahrscheinlichkeit einer fehlerhaften Antwort:  $\leq \frac{1}{2}$

- Es gilt allerdings auch:

Satz 21.6 [Agrawal et al. 04]

- $\text{PRIMES} \in \mathbf{P}$

- Der zugehörige Polynomialzeit-Algorithmus ist jedoch ziemlich kompliziert
- Laufzeit:  $\mathcal{O}((\log_2 N)^{10+\epsilon})$ , inzwischen verbessert auf:  $\mathcal{O}((\log_2 N)^{6+\epsilon})$
- Für praktische Zwecke ist der Solovay-Strassen-Test immer noch vorzuziehen



# Fehler-Wahrscheinlichkeit zufallsbasierter Algorithmen (1/2)

## Definition

- Seien  $f, g : \mathbb{N} \rightarrow \mathbb{R}$
- Wir sagen, dass ein Algorithmus  $A$  für eine Sprache  $L$  ein zufallsbasierter  $(f(n), g(n))$ -Algorithmus ist, falls er polynomielle Laufzeit hat und für jedes  $n$  gilt:
  - Für alle  $w \in L$  der Länge  $n$  ist die W-keit einer fehlerhaften Antwort von  $A$  höchstens  $f(n)$
  - Für alle  $w \notin L$  der Länge  $n$  ist die W-keit einer fehlerhaften Antwort von  $A$  höchstens  $g(n)$

- Der Solovay-Strassen-Test ist also
  - ein zufallsbasierter  $(0, \frac{1}{2})$ -Algorithmus mit polynomieller Laufzeit für PRIMES
  - ein zufallsbasierter  $(\frac{1}{2}, 0)$ -Algorithmus mit polynomieller Laufzeit für COMPOSITES
- Hier hängen  $f$  und  $g$  also nicht von  $n$  ab:
  - Für PRIMES ist  $f(n) = 0$  und  $g(n) = \frac{1}{2}$ , für alle  $n$
- Sind Algorithmen mit einer so großen Fehlerwahrscheinlichkeit relevant?
  - Ja, denn die Fehlerwahrscheinlichkeit lässt sich durch Wiederholung verkleinern

## Fehler-Wahrscheinlichkeit zufallsbasierter Algorithmen (2/2)

- Was passiert, wenn wir den Solovay-Strassen-Test zweimal laufen lassen und  $N$  akzeptieren, wenn es den Test zweimal besteht?

- Falls  $N \in \text{PRIMES}$ :  
Fehler-W-keit weiterhin 0

- Falls  $N \notin \text{PRIMES}$ :
  - W-keit eines Fehlers im ersten Durchgang:  $\leq \frac{1}{2}$
  - Also: für höchstens die Hälfte der Fälle wird der zweite Durchlauf erreicht
  - W-keit eines Fehlers im zweiten Durchgang:  $\leq \frac{1}{2}$
- ➡ W-keit, dass insgesamt die falsche Antwort gegeben wird, ist  $\leq \frac{1}{4}$

- Dieser Ansatz lässt sich weiterführen: wenn wir den Test  $k$ -mal ausführen, haben wir:
  - Falls  $N \in \text{PRIMES}$  Fehler-W-keit 0
  - Falls  $N \notin \text{PRIMES}$  Fehler-W-keit  $\leq \frac{1}{2^k}$

- Für praktische Zwecke reicht es also sicherlich aus, den Solovay-Strassen-Test 100 mal zu wiederholen, um sich zu vergewissern, dass  $N$  eine Primzahl ist
  - Die Wahrscheinlichkeit, dass während der Berechnung ein Asteroid einschlägt, ist dann größer als die Irrtumswahrscheinlichkeit des Algorithmus

- Die beschriebene Vorgehensweise lässt sich für alle  $(0, \frac{1}{2})$ -Algorithmen und  $(\frac{1}{2}, 0)$ -Algorithmen anwenden

# Inhalt

## 21.1 Zufallsbasierte Algorithmen für

21.1.1 3-SAT

21.1.2 Primzahlen

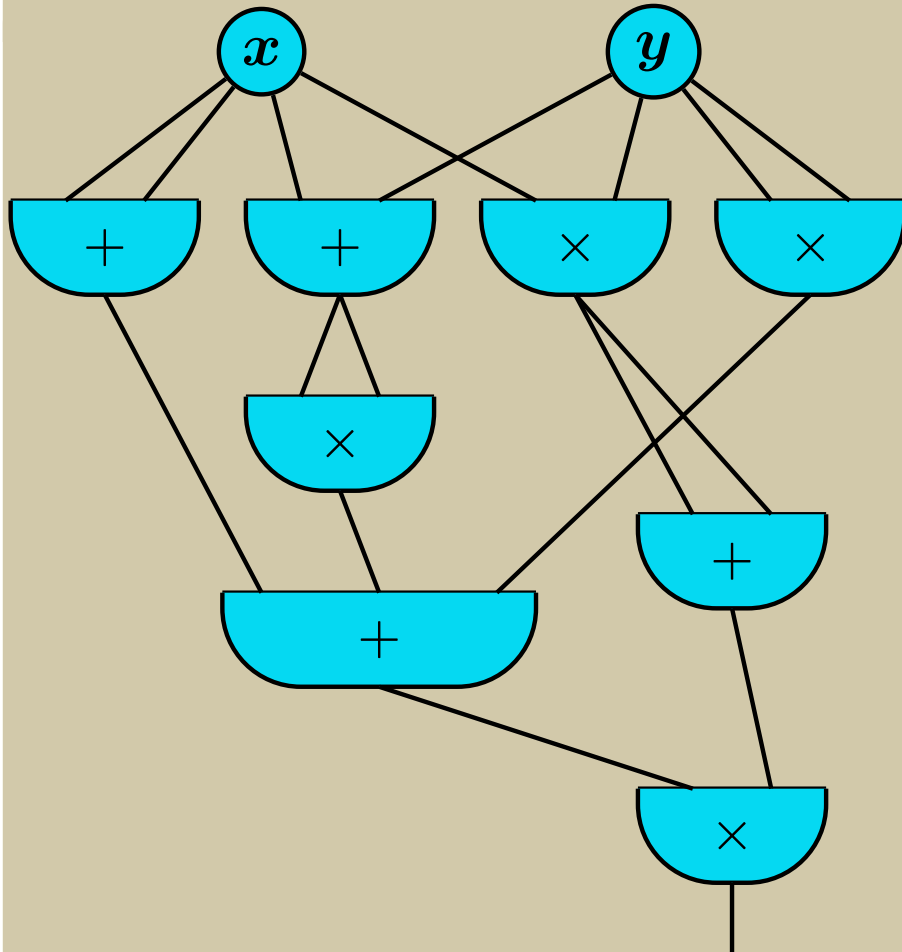
▷ **21.1.3 Arithmetischer Schaltkreise**

## Zwischenbemerkung

- Wir haben zwei Beispiele für zufallsbasierte Algorithmen gesehen:
  - Schönings Algorithmus löst das **NP**-vollständige 3-SAT-Problem, allerdings nicht in polynomieller Zeit
  - Der Solovay-Strassen-Test löst das Primzahl-Problem in polynomieller Zeit, das ist aber auch ohne Zufall möglich
- Als drittes betrachten wir nun einen zufallsbasierten Algorithmus, der ein Problem in polynomieller Zeit entscheidet, für das kein deterministischer polynomieller Algorithmus bekannt ist

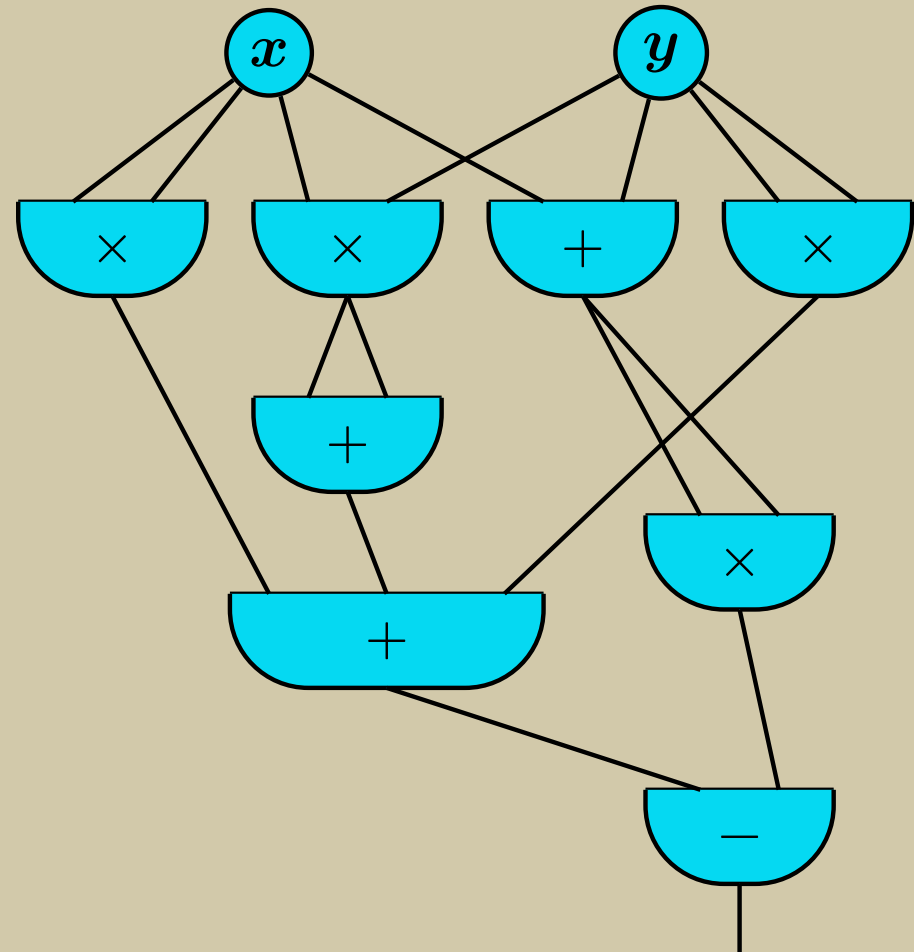
# Arithmetische Schaltkreise

Beispiel



- Dieser **arithmetische Schaltkreis** berechnet das **Polynom**  
 $4x^2y + 2x^3y + 4x^2y^2 + 4xy^3$

Beispiel (Forts.)



- Dieser arithmetische Schaltkreis berechnet das Polynom 0
- **Frage:** Wie lässt sich testen, ob ein arithmetischer Schaltkreis das Nullpolynom berechnet?

# Nulltest für arithmetischer Schaltkreise (1/2)

## Definition: ZEROCIRC

**Gegeben:** Arithmetischer Schaltkreis  $C$

**Frage:** Berechnet  $C$  das Nullpolynom, hat es also für jede Eingabe die Ausgabe 0?

- ZEROCIRC lässt sich natürlich dadurch lösen, dass das durch  $C$  repräsentierte Polynom  $p$  berechnet wird
- **Problem:** Die explizite Darstellung als Polynom kann exponentiell viele Terme haben
  - Deshalb führt dieser Ansatz nicht zu einem Polynomialzeit-Algorithmus
- Aber: Wir können ausnutzen, dass ein vom Nullpolynom verschiedenes Polynom für „die meisten“ Belegungen der Variablen nicht 0 ergibt
- Das folgende Lemma ist die Basis für einen zufallsbasierten Algorithmus für ZEROCIRC

## Lemma 21.7 [Schwartz-Zippel]

- Sei  $p(x_1, \dots, x_k)$  ein von 0 verschiedenes Polynom vom Grad  $\leq d$  und  $S$  eine Menge ganzer Zahlen
- Werden  $a_1, \dots, a_k$  zufällig in  $S$  gewählt (gleichverteilt, unabhängig, mit Zurücklegen), so gilt:

$$\Pr[p(a_1, \dots, a_k) \neq 0] \geq 1 - \frac{d}{|S|}$$

- Im Prinzip können wir also wie folgt vorgehen:
  - Wähle  $S \stackrel{\text{def}}{=} \{1, \dots, 2d\}$
  - Wähle zufällig  $\vec{a} = (a_1, \dots, a_k) \in S^k$
  - Teste ob  $p(\vec{a}) = 0$
  - Akzeptiere, falls dieser Test positiv verläuft
- Falls  $p = 0$ : Fehler-W-keit 0
- Falls  $p \neq 0$ : Fehler-W-keit  $\leq \frac{1}{2}$

## Nulltest arithmetischer Schaltkreise (2/2)

- Zwei Komplikationen:
  - (1) Wir kennen den Grad  $d$  von  $p$  nicht
  - (2)  $p(\vec{a})$  kann **exponentiell lang** werden
- Lösung für (1): Hat  $C \leq m$  Operationen, so ist der Grad von  $p$  höchstens  $2^m$   
 $\rightarrow S \stackrel{\text{def}}{=} \{1, \dots, 2^{m+1}\}$
- Lösung für (2): Werte  $C$  modulo einer Zahl  $N$  aus

### Algorithmus 21.8

Eingabe:  $C$

1. Wähle  $N \sim 2^{2m}$  zufällig
2. Wähle  $S \stackrel{\text{def}}{=} \{1, \dots, 2^{m+1}\}$
3. Wähle zufällig  $\vec{a} \in S^k$
4. Teste ob  $p(\vec{a}) \bmod N = 0$
5. Akzeptiere, falls Test positiv

- Klar:  $p = 0 \Rightarrow$  der Test akzeptiert
- Es gilt auch: Falls  $p \neq 0$  ist die W-keit, dass der Test **nicht akzeptiert**  $\geq \frac{1}{20m}$ 
  - Denn: die W-keit, dass  $N$  eine Primzahl ist, die  $p(\vec{a})$  nicht teilt, ist  $\geq \frac{1}{10m}$  (ohne Beweis)
- ➡ Falls  $p \neq 0$  ist die Fehler-W-keit  
 $\leq 1 - \frac{1}{20m}$
- Durch  $20m$  Wiederholungen lässt sich diese Fehler-W-keit auf  $\frac{1}{2}$  senken  
(wegen  $(1 - \frac{1}{x})^x \leq \frac{1}{e} \leq \frac{1}{2}$ )
- Auf diese Art erhalten wir also einen polynomiellen zufallsbasierten  $(0, \frac{1}{2})$ -Algorithmus für ZEROCIRC

# Zusammenfassung

- Für manche algorithmischen Probleme sind die schnellsten bekannten zufallsbasierten Algorithmen schneller als die besten bekannten Algorithmen, die deterministisch arbeiten
  - Der beste bekannte zufallsbasierte Algorithmus für das 3-SAT-Problem ist beispielsweise deutlich schneller als der beste bekannte deterministische Algorithmus
- ZEROCIRC ist ein Beispiel eines Problems, für das kein deterministischer Polynomialzeit-Algorithmus bekannt ist, aber zufallsbasierte Algorithmen, die in polynomieller Zeit laufen
- Häufig sind zufallsbasierte Algorithmen relativ einfach, ihre Analyse jedoch kompliziert
- Zufallsbasierte Algorithmen motivieren die Untersuchung von zufallsbasierten Komplexitätsklassen

👉 nächstes Kapitel



# Literaturhinweise

- Manindra Agrawal, Neeral Kayal, and Nitin Saxena. Primes in P. *Annals of Mathematics*, 160(2):781–793, 2004
- Timon Hertli. 3-SAT faster and simpler - Unique-SAT bounds for PPSZ hold in general. In Rafail Ostrovsky, editor, *FOCS*, pages 277–284. IEEE, 2011
- Robin A. Moser and Dominik Scheder. A full derandomization of Schoening's k-SAT algorithm. In *Proceedings of the 43rd annual ACM symposium on Theory of computing*, STOC '11, pages 245–252, New York, NY, USA, 2011. ACM
- Kazuhisa Makino, Suguru Tamaki, and Masaki Yamamoto. Derandomizing the HSSW algorithm for 3-SAT. *Algorithmica*, 67(2):112–124, 2013
- Christos M. Papadimitriou. *Computational complexity*. Addison-Wesley, Reading, Massachusetts, 1994
- Uwe Schöning. A probabilistic algorithm for k-SAT and constraint satisfaction problems. In *FOCS*, pages 410–414, 1999

# Grundbegriffe der Theoretischen Informatik

Sommersemester 2016 - Thomas Schwentick

Teil D: Komplexitätstheorie

22: Zufallsbasierte Komplexitätsklassen

Version von: 14. Juli 2016 (13:36)

# Zufallsbasierte Komplexitätsklassen: Grundlagen (1/3)

- In diesem Kapitel betrachten wir zufallsbasierte Algorithmen aus der Sicht der Komplexitätstheorie
- Wir definieren Komplexitätsklassen, die Probleme enthalten, die sich effizient mit zufallsbasierten Algorithmen lösen lassen
  - Die Laufzeit wird also (fast) immer polynomiell sein
- Durch verschiedene Anforderungen an die Akzeptier-Wahrscheinlichkeiten werden sich unterschiedliche Klassen ergeben
- Wir betrachten die folgenden Varianten zufallsbasierter Algorithmen:
  - einseitiger Fehler oder zweiseitiger Fehler
  - möglicherweise großer Fehler ( $< \frac{1}{2}$ ) oder kleiner Fehler ( $\leq \frac{1}{4}$ )
- Die Algorithmen für PRIMES und ZEROCIRC haben einseitigen, kleinen Fehler  
(bei genügend häufiger Wiederholung)

## Zufallsbasierte Komplexitätsklassen: Grundlagen (2/3)

- Die erste Frage, die wir beantworten müssen:
  - Wie modellieren wir zufallsbasierte Algorithmen durch Turingmaschinen?
- Zur Beantwortung gehen wir ähnlich vor wie bei der Definition des nicht-deterministischen Akzeptierens
- Wir betrachten Berechnungen einer TM mit Eingabe  $x$  und **Zusatzeingabe**  $y$ , wobei die Zusatzeingabe nur polynomiell lang in  $|x|$  sein darf
- Die Zusatzeingabe repräsentiert die Zufallsbits
- Wir sagen  $M(x, y)$  **akzeptiert**, wenn  $M$  bei Eingabe  $x$  und Zusatzeingabe  $y$  akzeptiert
- Der Einfachheit halber betrachten wir nur Eingaben und Zusatzeingaben über dem Alphabet  $\Sigma = \{0, 1\}$

- Die relative Häufigkeit der Zusatzeingaben  $y$ , die zum Akzeptieren führen, im Verhältnis zu allen Zusatzeingaben, definiert dann gerade die Akzeptierwahrscheinlichkeit:

### Definition

- Sei  $M$  eine TM mit Zeitschranke  $T$  und  $x \in \Sigma^*$
- $p_M(x) \stackrel{\text{def}}{=} \frac{|\{y \in \Sigma^{T(|x|)} \mid M(x, y) \text{ akzeptiert}\}|}{2^{T(|x|)}}$
- Die Zeitschranke hängt dabei wieder nur von  $|x|$  ab, es muss also gelten:
$$t_M((x, y)) \leq T(|x|)$$

# Zufallsbasierte Komplexitätsklassen: Grundlagen (3/3)

- Zur Erinnerung: ein zufallsbasierter  $(f(n), g(n))$ -Algorithmus für eine Sprache  $L$  hat für Eingaben  $w \in L$  Fehlerwahrscheinlichkeit  $\leq f(|w|)$  und für  $w \notin L$  Fehlerwahrscheinlichkeit  $\leq g(|w|)$
- Beobachtung: Zu jeder Sprache gibt es
  - einen polynomiellen  $(\frac{1}{2}, \frac{1}{2})$ -Algorithmus:
    - \* Wähle zufällig und gleichverteilt ein Bit  $b \in \{0, 1\}$  und akzeptiere falls  $b = 1$
  - einen polynomiellen  $(0, 1)$ -Algorithmus:
    - \* Akzeptiere immer
  - einen polynomiellen  $(1, 0)$ -Algorithmus:
    - \* Lehne immer ab
  - einen polynomiellen  $(p, q)$ -Algorithmus, falls  $p + q = 1$  und es für jedes  $n$  ein  $k$  mit  $p = \frac{k}{2^{T(n)}}$  gibt
- Interessant sind also überhaupt nur Klassen, für die die Summe  $p + q$  der beiden Fehler-W-keiten kleiner als 1 ist

# Inhalt

- ▷ **22.1 Komplexitätsklassen mit einseitigem Fehler**
- 22.2 Komplexitätsklassen mit kleinem zweiseitigen Fehler
- 22.3 Komplexitätsklassen mit großem zweiseitigen Fehler
- 22.4 Komplexitätsklassen-Übersicht

# Zufallsbasierte Komplexitätsklassen: RP und co-RP (1/2)

## Definition


- **RP**  $\stackrel{\text{def}}{=}$  Klasse der Mengen  $L$  mit Poly-Zeit-TM  $M$ , so dass
  - $x \in L \Rightarrow p_M(x) \geq \frac{1}{2}$
  - $x \notin L \Rightarrow p_M(x) = 0$
- **co-RP**  $\stackrel{\text{def}}{=}$  Klasse der Mengen  $L$  mit Poly-Zeit-TM  $M$ , so dass
  - $x \in L \Rightarrow p_M(x) = 1$
  - $x \notin L \Rightarrow p_M(x) \leq \frac{1}{2}$

- Zu beachten: Diese Definitionen verwenden Akzeptierwahrscheinlichkeiten, keine Fehlerwahrscheinlichkeiten

- **RP** umfasst die Probleme mit polynomialen  $(\frac{1}{2}, 0)$ -Algorithmen
- **co-RP** umfasst die Probleme mit polynomialen  $(0, \frac{1}{2})$ -Algorithmen

- Die Algorithmen für PRIMES und ZEROCIRC belegen:

– PRIMES  $\in$  **co-RP**

 Aber, wie wir wissen, gilt sogar:  
PRIMES  $\in$  **P**

– ZEROCIRC  $\in$  **co-RP**

- Nach Definition gilt offensichtlich:

**RP**  $\subseteq$  **NP**

## Zufallsbasierte Komplexitätsklassen: RP und co-RP (2/2)

- Die bei den Algorithmen für PRIMES und ZEROCIRC verwendete Technik der **Wahrscheinlichkeitsverstärkung** lässt sich für **RP** und **co-RP** verallgemeinern

### Satz 22.1

- Sei  $L \in \mathbf{RP}$ ,  $k \in \mathbb{N}$
- Dann gibt es eine polynomiell zeitbeschränkte TM  $M$ , so dass
  - $x \in L \Rightarrow p_M(x) \geq 1 - \frac{1}{2^{|x|^k}}$
  - $x \notin L \Rightarrow p_M(x) = 0$

### Beweisidee

- Sei  $M'$  eine der Definition von **RP** entsprechende TM für  $L$
- Die TM  $M$  simuliert  $M'$  hintereinander  $|x|^k$  mal
  - $M$  erwartet eine Zusatzeingabe der Form  $y_1 \cdot \dots \cdot y_{|x|^k}$  und verwendet den String  $y_i$  als Zusatzeingabe für die  $i$ -te Simulation von  $M'$
- $M$  akzeptiert genau dann, wenn mindestens eine dieser Simulationen zum Akzeptieren führt
- Die W-keit, dass  $M'$  für alle diese Zusatzeingaben ablehnt, ist im Falle  $x \in L$  höchstens  $\frac{1}{2^{|x|^k}}$
- Und:  $M$  ist polynomiell zeitbeschränkt
- Ein analoges Resultat gilt für **co-RP**



# Zufallsbasierte Komplexitätsklassen: ZPP (1/5)

## Definition

- $\mathbf{ZPP} \stackrel{\text{def}}{=} \mathbf{RP} \cap \mathbf{co-RP}$

- Probleme in **ZPP** haben also einen polynomiellen  $(\frac{1}{2}, 0)$ -Algorithmus und einen polynomiellen  $(0, \frac{1}{2})$ -Algorithmus
- Durch Kombination dieser beiden Algorithmen lassen sich neue Algorithmen mit sehr günstigen Eigenschaften konstruieren
- Erster Algorithmentyp für **ZPP**-Probleme:
  - polynomielle Laufzeit
  - *Drei Antwortmöglichkeiten:*  
„ja“, „nein“, „weiß-nicht“
  - „ja“, „nein“-Antworten immer richtig
- Zweiter Algorithmentyp für **ZPP**-Probleme:
  - Zwei Antwortmöglichkeiten: „ja“, „nein“
  - Antworten immer richtig
  - im *Durchschnitt* polynomielle Laufzeit

## Zufallsbasierte Komplexitätsklassen: ZPP (2/5)

- Für den ersten Typ definieren wir das folgende TM-Modell

### Definition


- Eine **Las-Vegas-TM** für eine Sprache  $L$  hat folgende Eigenschaften:
  - Sie hat außer „ja“ und „nein“ einen weiteren Endzustand „weiß-nicht“
  - \* Für  $x \in L$  endet jede Berechnung (für jede Zusatzeingabe  $y$ ) in „ja“ oder „weiß-nicht“
  - \* Für  $x \notin L$  endet jede Berechnung in „nein“ oder „weiß-nicht“
  - Die Antwort „ja“ oder „nein“ ist also immer richtig
  - Die W-keit, dass  $M$  bei Eingabe  $x$  im Zustand „weiß-nicht“ endet, bezeichnen wir mit  $p_{M,?}(x)$

- Für den zweiten Typ betrachten wir eine andere Art von „Zeitschranke“

### Definition

- Eine TM  $M$  entscheidet eine Sprache  $L$  mit **polynomiell erwarteter Laufzeit**, falls es  $c, d$  gibt, so dass für alle  $x$  gelten:
  - Für jede Zusatzeingabe  $y$  der Länge  $|x|^c$  gibt  $M$  die richtige Antwort („ja“, falls  $x \in L$ , „nein“, falls  $x \notin L$ )

$$- \frac{1}{2^{|x|^c}} \sum_{y \in \Sigma^{|x|^c}} t_M(x, y) \leq |x|^d$$

 Die durchschnittliche Laufzeit (gemittelt über die Zusatzeingaben  $y$ ) ist also polynomiell beschränkt

- Zu beachten:
  - Die Laufzeit von  $M$  kann für einzelne Zusatzeingaben größer als  $|x|^d$  sein

# Zufallsbasierte Komplexitätsklassen: ZPP (3/5)

## Satz 22.2

- Für eine Sprache  $L$  sind äquivalent:
  - (a)  $L \in \mathbf{ZPP}$
  - (b) Es gibt für  $L$  eine Las-Vegas-TM  $M_1$  mit Poly-Laufzeit, so dass für alle  $x$  gilt:  $p_{M,?}(x) \leq \frac{1}{2}$
  - (c) Es gibt für  $L$  eine zufallsbasierte TM  $M_2$  mit polynomiell erwarteter Laufzeit

## Beweisskizze „(a) $\Rightarrow$ (b)“

- Sei  $L \in \mathbf{ZPP}$
- Sei  $A^+$  ein  $(\frac{1}{2}, 0)$ -Algorithmus für  $L$  (**RP**) und  $A^-$  ein  $(0, \frac{1}{2})$ -Algorithmus für  $L$  (**co-RP**)
- Sei  $A$  folgender Algorithmus (bei Eingabe  $x$ ):
  - Simuliere  $A^+$  bei Eingabe  $x$
  - Falls  $A^+$  akzeptiert, Ausgabe „ja“
  - Simuliere  $A^-$  bei Eingabe  $x$
  - Falls  $A^-$  ablehnt, Ausgabe „nein“
  - Andernfalls Ausgabe: „weiß-nicht“

## Beweisskizze (Forts.)

- $A^+$  akzeptiert nur, falls  $x \in L$ 
  - ➔ die Ausgabe „ja“ von  $A$  ist immer richtig
- Analog:  $A^-$  lehnt nur ab, falls  $x \notin L$ 
  - ➔ die Ausgabe „nein“ von  $A$  ist immer richtig
- Schließlich:
  - Falls  $x \in L$ , gibt  $A^+$  die Antwort „ja“ mit W-keit  $\geq \frac{1}{2}$ 
    - ➔  $p_{M,?}(x) \leq \frac{1}{2}$
  - Falls  $x \notin L$ , gibt  $A^-$  die Antwort „nein“ mit W-keit  $\geq \frac{1}{2}$ 
    - ➔  $p_{M,?}(x) \leq \frac{1}{2}$
- ➔ Aus  $A$  lässt sich eine Las Vegas-TM  $M_1$  wie in (b) konstruieren

# Zufallsbasierte Komplexitätsklassen: ZPP (4/5)

## Beweisskizze „(b) $\Rightarrow$ (c)“

- Sei  $M_1$  Las-Vegas-TM für  $L$  gemäß (b) mit Zeitschranke  $n^j$
- Sei ferner  $M$  eine TM, die  $L$  in Zeit  $2^{n^k}$  entscheidet, für ein  $k \in \mathbb{N}$   
☞ **ZPP**  $\subseteq$  **RP**  $\subseteq$  **NP**  $\subseteq$  **EXPTIME**
  - Idee für  $M$ : Simuliere  $M_1(x, y)$ , für alle Zusatzeingaben  $y$  der Länge  $|x|^j$
- $M_2$  arbeitet bei Eingabe  $x$  wie folgt:
  - Simuliere  $|x|^k$  mal  $M_1$
  - Falls eine dieser Simulationen „ja“ ausgibt, so akzeptiere
  - Falls eine dieser Simulationen „nein“ ausgibt, so lehne ab
  - Falls alle Simulationen „weiß-nicht“ ausgehen, simuliere  $M$  bei Eingabe  $x$ , und gib die Antwort, die  $M$  geben würde

## Beweisskizze (Forts.)

- Klar:  $M_2$  terminiert immer und gibt immer die korrekte Antwort
- Die W-keit, dass alle Simulationen von  $M_1$  die Antwort „weiß-nicht“ haben ist  $\leq \frac{1}{2^{|x|^k}}$   
➡ Also ist die erwartete Laufzeit  $\leq |x|^k |x|^j + \frac{1}{2^{|x|^k}} 2^{|x|^k} = |x|^{j+k} + 1$

# Zufallsbasierte Komplexitätsklassen: ZPP (5/5)

## Beweisskizze „(c) $\Rightarrow$ (a)“

- Sei  $M_2$  eine TM für  $L$  mit polynomieller erwarteter Laufzeit
- Seien  $c, d$  so gewählt, dass gelten:
  - Für jede Zusatzeingabe  $y$  der Länge  $|x|^c$  gibt  $M$  die richtige Antwort („ja“, falls  $x \in L$ , „nein“, falls  $x \notin L$ )

$$- \frac{1}{2^{|x|^c}} \sum_{y \in \Sigma^{|x|^c}} t_M(x, y) \leq |x|^d$$

- Da die mittlere Laufzeit  $\leq |x|^d$  ist, ist die W-keit, dass für ein zufällig gewähltes  $y$  die Laufzeit größer als  $2|x|^d$  ist,  $< \frac{1}{2}$

- Wir konstruieren eine TM  $M^+$  zum Nachweis, dass  $L \in \mathbf{RP}$

## Beweisskizze (Forts.)

- $M^+$  arbeitet wie folgt (bei Eingabe  $x$ ):
  - Simuliere  $M_2$  bei Eingabe  $x$  für  $2|x|^d$  Schritte
  - Akzeptiere, falls  $M_2$  in dieser Zeit akzeptiert
  - Andernfalls lehne ab

- Da  $M_2$  bei jeder Zusatzeingabe die richtige Ausgabe hat, und die W-keit, dass  $M_2$  in Zeit  $2|x|^d$  anhält,  $\geq \frac{1}{2}$  ist, gilt:

- Falls  $x \in L$  ist  $p_{M^+}(x) \geq \frac{1}{2}$
- Falls  $x \notin L$  ist  $p_{M^+}(x) = 0$

➡  $L \in \mathbf{RP}$

- Die Konstruktion einer TM  $M^-$  zum Nachweis, dass  $L \in \mathbf{co-RP}$ , ist völlig analog

# Inhalt

22.1 Komplexitätsklassen mit einseitigem Fehler

▷ **22.2 Komplexitätsklassen mit kleinem zweiseitigen Fehler**

22.3 Komplexitätsklassen mit großem zweiseitigen Fehler

22.4 Komplexitätsklassen-Übersicht

# Probabilistische Klassen: kleiner, zweiseitiger Fehler

## Definition

- **BPP** sei die Klasse der Mengen  $L$ , für die es eine polynomiell zeitbeschränkte TM  $M$  gibt, so dass:
  - $x \in L \Rightarrow p_M(x) \geq \frac{3}{4}$
  - $x \notin L \Rightarrow p_M(x) \leq \frac{1}{4}$
- **BPP** umfasst also alle Probleme, die einen polynomiellen  $(\frac{1}{4}, \frac{1}{4})$ -Algorithmus haben
- Im Falle von **RP** und **coRP** lässt sich daraus, dass zwei Berechnungen für eine Eingabe  $x$  einmal „ja“ und einmal „nein“ ergeben, jeweils ein eindeutiger Schluss ziehen
  - Das war dort die Grundlage für die Wahrscheinlichkeitsverstärkung
- Für **BPP** können wir nicht so vorgehen, da bei einer „**BPP**-TM“ vorkommen kann,
  - dass sie für  $x \in L$  „nein“ sagt, und
  - dass sie für  $x \notin L$  „ja“ sagt

- Aber auch hier lässt sich auf einfache Weise eine W-Verstärkung erreichen:
  - Wiederhole den Algorithmus (mit mehreren Zusatzeingaben) und akzeptiere genau dann, wenn die **Mehrheit der Berechnungen** akzeptierend ist

## Satz 22.3

- Ist  $L \in \mathbf{BPP}$ ,  $k \in \mathbb{N}$ , so gibt es eine polynomiell zeitbeschränkte TM  $M$ , so dass gilt:
  - (a)  $x \in L \Rightarrow p_M(x) \geq 1 - \frac{1}{2^{|x|^k}}$
  - (b)  $x \notin L \Rightarrow p_M(x) \leq \frac{1}{2^{|x|^k}}$
- Die Fehlerwahrscheinlichkeit kann also nicht nur (in Poly-Zeit) unter jede beliebige feste Zahl  $\epsilon > 0$  gesenkt werden
- Sondern sie kann sich für große  $n$  exponentiell schnell an 0 annähern

# Ein hilfreiches Resultat aus der Wahrscheinlichkeitstheorie

- Um Satz 22.3 zu beweisen brauchen wir etwas Wahrscheinlichkeitstheorie
- Wir hatten eben schon verwendet, dass für Zufallsvariable  $X$ , die keine negativen Werte annehmen, und den Erwartungswert  $E(X) = p$  haben, gilt:
  - Die Wahrscheinlichkeit, dass der Wert von  $X$  größer als  $2p = 2E(X)$  ist, ist kleiner als  $\frac{1}{2}$ :
    - \*  $P(X \geq 2p) \leq \frac{1}{2}$
  - Zum Beweis von Satz 22.3 benötigen wir jedoch eine bessere Abschätzung, die uns das folgende Lemma liefert

## Lemma 22.4 [Chernoff-Schranke]

- Seien  $X_1, \dots, X_n$  unabhängige, 0-1-wertige Zufallsvariable mit  $P(X_i = 1) \leq p$  für alle  $i$

- Sei  $X \stackrel{\text{def}}{=} \sum_{i=1}^n X_i$

- Dann gilt für alle  $\theta$  mit  $0 \leq \theta \leq 1$ :

$$P(X \geq (1 + \theta)pn) \leq e^{-\frac{\theta^2}{3}pn}$$

- Die W-Keit, dass in 100 Münzwürfen öfter als 75 mal „Kopf“ kommt, ist z.B. kleiner als 0,02:
  - $n = 100, p = \frac{1}{2}, \theta = 0,5$
- Wir verwenden Lemma 22.4 für  $\theta = 1$  und  $p = \frac{1}{4}$  und erhalten:
  - Wenn bei  $n$  Experimenten jeweils mit W-Keit  $\leq \frac{1}{4}$  das Ergebnis 1 und mit W-keit  $\geq \frac{3}{4}$  das Ergebnis 0 ist,
  - dann ist die W-keit, dass die Summe der Ergebnisse größer als  $n/2$  ist, höchstens  $e^{-\frac{1}{12}n}$



# Wahrscheinlichkeitsverstärkung für BPP

## Beweisskizze von Satz 22.3

- Sei  $L \in \mathbf{BPP}$  und sei  $M$  eine TM mit Zeitschranke  $n^l$ , für die gilt:
  - $x \in L \Rightarrow p_M(x) \geq \frac{3}{4}$
  - $x \notin L \Rightarrow p_M(x) \leq \frac{1}{4}$
- Wir konstruieren eine TM  $M'$  mit Zeitschranke  $\sim 24n^{k+l}$
- $M'$  arbeitet wie folgt (bei Eingabe  $x$ ):
  - $M'$  simuliert  $24|x|^k$  mal  $M$  bei Eingabe  $x$  (und  $24|x|^k$  Zusatzeingaben) und zählt die Anzahl  $m$  der akzeptierenden Berechnungen
  - $M'$  hat Ausgabe „ja“, falls
$$m \geq 12|x|^k$$
  - Andernfalls Ausgabe „nein“

## Beweisskizze (Forts.)

- Klar:  $M'$  hat polynomielle Laufzeit
- Wir zeigen nun: (b):
$$x \notin L \Rightarrow p_{M'}(x) \leq \frac{1}{2^{|x|^k}}$$
- Sei dazu  $x \notin L$
- Für  $i \leq 24|x|^k$  sei  $X_i$  die Zufallsvariable mit Wert
  - 1, falls die  $i$ -te Simulation akzeptiert
  - 0, falls die  $i$ -te Simulation ablehnt
- Also:  $P(X_i = 1) \leq \frac{1}{4}$ , für alle  $i$
- Sei  $X \stackrel{\text{def}}{=} \sum_{i=1}^{24|x|^k} X_i$  und  $\theta = 1$
- ➔  $P(X \geq 12|x|^k) \leq e^{-2|x|^k} \leq 2^{-|x|^k}$
- (a) kann analog gezeigt werden
- ➔ Behauptung

# Zufallsbasierte Algorithmen für 3-SAT: Grenzen

- Interessante Frage: gibt es für 3-SAT einen zufallsbasierten Algorithmus mit polynomieller Laufzeit?
- Als Konsequenz ergäbe sich:
  - **NP**  $\subseteq$  **BPP** oder sogar
  - **NP**  $\subseteq$  **RP**
- Das wird als unwahrscheinlich erachtet, insbesondere angesichts der verbreiteten Vermutung, dass **BPP** = **P** sein könnte:
  - Denn dann würde **NP** = **P** folgen
- Trotzdem sind Algorithmen wie der von Schönig äußerst nützlich, wie bereits im letzten Kapitel besprochen

# Inhalt

22.1 Komplexitätsklassen mit einseitigem Fehler

22.2 Komplexitätsklassen mit kleinem zweiseitigen Fehler

▷ **22.3 Komplexitätsklassen mit großem zweiseitigen Fehler**

22.4 Komplexitätsklassen-Übersicht

# Probabilistische Klassen: großer, zweiseitiger Fehler

## Definition

- **PP** sei die Klasse der Mengen  $L$ , für die es eine polynomiell zeitbeschränkte TM  $M$  gibt, so dass:
  - $x \in L \Rightarrow p_M(x) > \frac{1}{2}$
  - $x \notin L \Rightarrow p_M(x) \leq \frac{1}{2}$

## Proposition 22.5

- (a) **ZPP**  $\subseteq$  **RP**  $\subseteq$  **BPP**  $\subseteq$  **PP**
- (b) **ZPP**  $\subseteq$  **co-RP**  $\subseteq$  **BPP**  $\subseteq$  **PP**
- (c) **NP**  $\subseteq$  **PP**

- (a,b) folgen direkt aus den Definitionen

## Beweisskizze für (c)

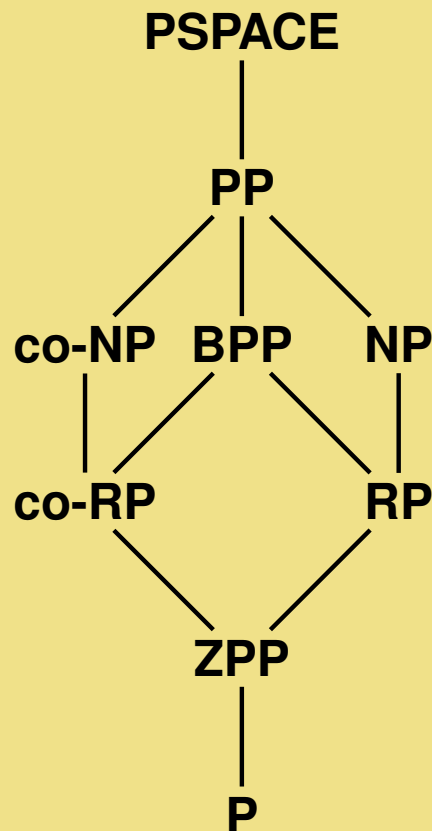
- Sei  $L \in \mathbf{NP}$  und  $M$  eine TM, die  $L$  nichtdeterministisch entscheidet
- Idee: Konstruiere TM  $M'$ , die immer mit W-keit  $\geq \frac{1}{2}$  akzeptiert


## Beweisskizze (Forts.)

- Sei  $M'$  die folgende TM (Eingabe  $x$ ):
  - Falls das erste Zeichen der Zusatzeingabe 1 ist, so akzeptiert  $M'$
  - Falls das erste Zeichen der Zusatzeingabe 0 ist, so simuliert  $M'$  die TM  $M$  bei Eingabe  $x$  mit dem Rest der Zusatzeingabe, und akzeptiert genau dann, wenn  $M$  akzeptiert
- Falls  $x \in L$  ist die W-keit, dass  $M'$  akzeptiert  $> \frac{1}{2}$ :
  - In der Hälfte aller Fälle akzeptiert  $M'$ , weil das erste Bit der Zusatzeingabe 1 ist
  - Es gibt aber auch mindestens eine Zusatzeingabe  $y$ , für die  $M(x, y)$  akzeptiert
  - ➡  $M'$  akzeptiert bei Zusatzeingabe  $0y$
  - ➡  $p_{M'}(x) > \frac{1}{2}$
- Klar: Falls  $x \notin L$ , ist  $p_{M'}(x) = \frac{1}{2}$

# Verhältnis der betrachteten Komplexitätsklassen

- Das folgende Diagramm illustriert die Inklusionsstruktur der betrachteten Klassen:



- Welche Komplexitätsklasse entspricht nun dem intuitiven Begriff des effizient berechenbaren am besten?
- P**? Ist ein Problem in **P** lösbar, wissen wir, dass wir nach polynomieller Zeit die richtige Antwort bekommen  
 und das Problem, dass Polynome untragbar groß sein können, haben wir ja schon besprochen
- ZPP**? Ist ein Problem in **ZPP** lösbar, wissen wir, dass wir immer die richtige Antwort bekommen und dies mit großer W-keit nach polynomieller Zeit passiert
- BPP**? Ist ein Problem in **BPP** lösbar, können wir zwar nicht sicher sein, dass die Antwort des Algorithmus korrekt ist, aber die Fehler-W-keit kann beliebig klein gemacht werden
- Für jede der drei Möglichkeiten gibt es gute Gründe
- Seit einigen Jahren wird von vielen vermutet, dass die Diskussion überflüssig ist, und **P** = **BPP** gilt

# Fehler-W-Keit der betrachteten Komplexitätsklassen

| Klasse       | max. Fehler $x \in L$ | max. Fehler $x \notin L$ |
|--------------|-----------------------|--------------------------|
| <b>P</b>     | 0                     | 0                        |
| <b>NP</b>    | $< 1$                 | 0                        |
| <b>RP</b>    | $\leq \frac{1}{2}$    | 0                        |
| <b>co-RP</b> | 0                     | $\leq \frac{1}{2}$       |
| <b>BPP</b>   | $\leq \frac{1}{4}$    | $\leq \frac{1}{4}$       |
| <b>PP</b>    | $< \frac{1}{2}$       | $\leq \frac{1}{2}$       |

- Bei **RP** und **co-RP** kann  $\frac{1}{2}$  durch jede beliebige Konstante  $c$ ,  $0 < c < 1$ , ersetzt werden
- Bei **BPP** kann  $\frac{1}{4}$  durch jede Konstante  $c$ ,  $0 < c < \frac{1}{2}$  ersetzt werden
- **NP** kann also auch als eine probabilistische Komplexitätsklasse aufgefasst werden:
  - Ist  $x \in L$  wird dies mit W-keit  $> 0$  erkannt
  - Ist  $x \notin L$  wird dies mit W-keit 1 erkannt

# Inhalt

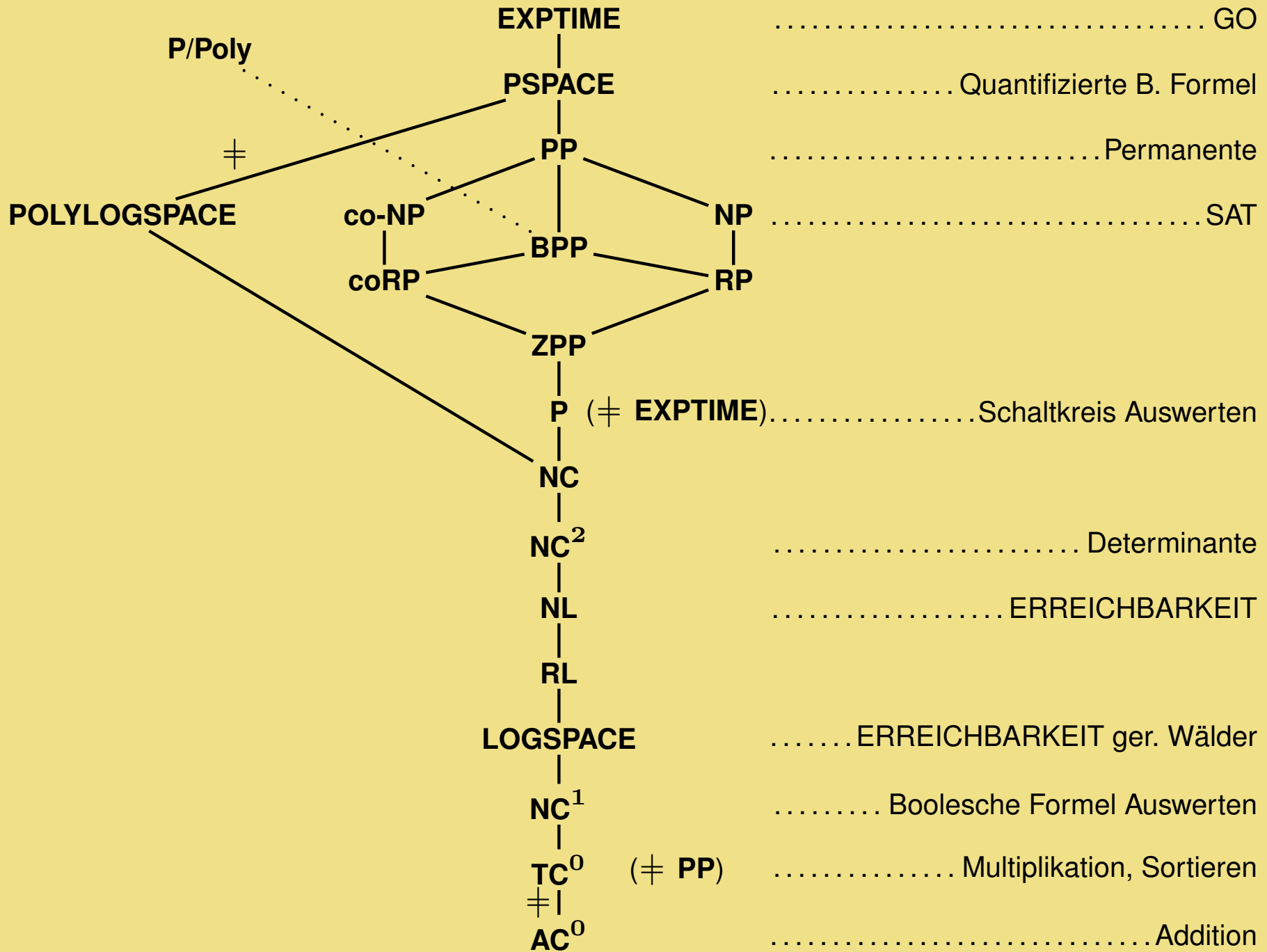
22.1 Komplexitätsklassen mit einseitigem Fehler

22.2 Komplexitätsklassen mit kleinem zweiseitigen Fehler

22.3 Komplexitätsklassen mit großem zweiseitigen Fehler

▷ **22.4 Komplexitätsklassen-Übersicht**

## Es gibt noch viel mehr Komplexitätsklassen...





# Zusammenfassung

- Es gibt Klassen mit einseitigem oder zweiseitigem Fehler, sowie kleinem oder großem Fehler
- Die Probleme in **ZPP**, **RP**, **co-RP**, **BPP** können durchaus als effizient berechenbar gelten

# Literaturhinweise

- Christos M. Papadimitriou. *Computational complexity*. Addison-Wesley, Reading, Massachusetts, 1994

# Grundbegriffe der Theoretischen Informatik

Sommersemester 2016 - Thomas Schwentick

## 23: Zusatzkapitel: Parametrisierte Algorithmen und Komplexität

Version von: 19. Juli 2016 (12:57)

# Einige algorithmische Probleme (1/4)

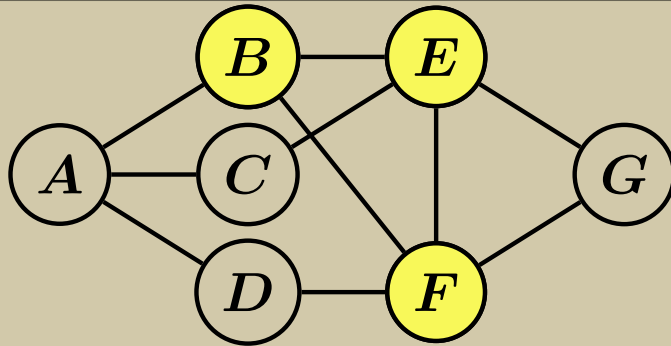
## Definition: VERTEXCOVER

**Gegeben:** Unger. Graph  $G = (V, E)$ ,  $k$

**Frage:** Gibt es  $U \subseteq V$ , so dass gelten:

- (1)  $(u, v) \in E \Rightarrow u \in U$  oder  $v \in U$
- (2)  $|U| \leq k$ ?

## Beispiel



## Definition: INDEPENDENTSET

**Gegeben:** Unger. Graph  $G = (V, E)$ ,  $k$

**Frage:** Gibt es  $U \subseteq V$ , so dass gelten:

- (1) für je zwei Knoten  $u, v \in U$  ist  $(u, v) \notin E$
- (2)  $|U| \geq k$ ?

## Definition: CLIQUE

**Gegeben:** Unger. Graph  $G = (V, E)$ ,  $k$

**Frage:** Gibt es  $U \subseteq V$ , so dass gelten:

- (1) für je zwei Knoten  $u, v \in U$  ist  $(u, v) \in E$
- (2)  $|U| \geq k$ ?

## Definition: DOMINATINGSET

**Gegeben:** Unger. Graph  $G = (V, E)$ ,  $k$

**Frage:** Gibt es  $U \subseteq V$ , so dass gelten:

- (1) für jedes  $v \in V - U$  gibt es ein  $u \in U$  mit  $(u, v) \in E$
- (2)  $|U| \leq k$ ?

- In diesem Kapitel betrachten wir nur ungerichtete Graphen ohne Schleifen
- $V$  und  $E$  bezeichnen immer die Knoten- und Kantenmenge von  $G$

## Einige algorithmische Probleme (2/4)

- Anwendung für VERTEXCOVER:

- Gegeben eine Menge von Tupeln einer Datenbank
- Eine Kante zwischen zwei Tupeln bedeutet, dass sie zusammen gegen eine Schlüsselbedingung verstoßen
- $k$ : Anzahl der zu löschenden Tupel, um die DB zu reparieren

- Anwendung für INDEPENDENTSET:

- Gegeben eine Menge von möglichen Standorten für Funkmasten
- Eine Kante zwischen zwei Standorten bedeutet, dass es zwischen zwei Masten an diesen Standorten Interferenzen gäbe
- $k$ : Anzahl der angestrebten Masten

- Anwendung für CLIQUE:

- Gegeben eine Menge von Personen
- Eine Kante zwischen zwei Personen bedeutet, dass die beiden sich kennen
- $k$ : Größe einer gesuchten Gruppe von Personen, die sich alle gegenseitig kennen

- Anwendung für DOMINATINGSET:

- Gegeben eine Menge von Museumsräumen
- Eine Kante zwischen zwei Räumen bedeutet, dass von jedem Raum der andere überblickt werden kann
- $k$ : Anzahl von Räumen, die besetzt werden müssen, damit alle Räume überschaut werden können

## Einige algorithmische Probleme (3/4)

- Alle vier Probleme sind **NP**-vollständig
- Also:  
$$\text{VERTEXCOVER} \leq_p \text{CLIQUE} \leq_p \text{INDEPENDENTSET} \leq_p \text{DOMINATINGSET} \leq_p \text{VERTEXCOVER}$$
- Sind die vier Probleme also in jeder Hinsicht algorithmisch äquivalent?

- Alle vier Probleme haben in der Eingabe einen Parameter  $k$ , der die Größe einer gesuchten Knotenmenge angibt
- Wir betrachten, wie der Aufwand von Lösungsalgorithmen von diesem Parameter  $k$  abhängt

- In einer Hinsicht verhalten sie sich alle gleich:
  - Es gibt einen Algorithmus, der das Problem in  $n^k$  Durchläufen löst, die jeweils polynomielle Zeit benötigen  
→ Probiere alle Mengen der Größe  $k$  aus
- Das ist zwar für jedes feste  $k$  polynomiell, aber bereits beispielsweise für  $k = 20$  nicht mehr nutzbar

→ Wir interessieren uns in diesem Kapitel **NICHT** für Algorithmen mit Zeitschranken der Art  $\mathcal{O}(n^k)$

## Einige algorithmische Probleme (4/4)

- Wir werden die algorithmische Schwierigkeit dieser Probleme in Abhängigkeit vom Parameter  $k$  in diesem Kapitel genauer untersuchen
- Dabei werden wir feststellen, dass sie sich in dieser Hinsicht (vermutlich) deutlich unterscheiden

- Beobachtung:
  - CLIQUE und INDEPENDENTSET verhalten sich gleich:
    - \* Denn jede  $k$ -Clique von  $G$  ist eine unabhängige Menge im Komplementgraph von  $G$  und umgekehrt  
☞ keine Schleifen!
    - \* Die beiden Probleme lassen sich also sehr einfach aufeinander reduzieren unter Beibehaltung des Parameterwertes  $k$
  - ➡ Lösungsalgorithmen für das eine Problem lassen sich leicht auch für das andere Problem anwenden

- Im Folgenden betrachten wir also nur noch VERTEXCOVER, CLIQUE und DOMINATINGSET

# Eine einfache aber wirkungsvolle Verbesserung

- **Frage:** Gibt es Algorithmen für VERTEXCOVER, deren Aufwand wesentlich besser als  $\mathcal{O}(n^k)$  ist?

- **Beobachtung:**

Ist  $(u, v)$  eine Kante von  $G$ , so muss  $u$  oder  $v$  in  $U$  vorkommen  $\boxplus$

- Das legt eine Verzweigung zu zwei Teilproblemen nahe:
  - \*  $(G-u, k-1)$  und  $(G-v, k-1)$
  - \* Dabei bezeichnet  $G-u$  den Graphen, der aus  $G$  durch Entfernen von  $u$  und allen Kanten mit  $u$  entsteht
- $(G, k)$  hat genau dann eine Lösung, wenn eines der beiden Teilprobleme eine Lösung hat
- Die Rekursionstiefe dieses Algorithmus ist  $k$
- Suchbaum mit  $2^k$  Blättern
- Zeitaufwand  $\mathcal{O}(2^k n^2)$

- Dieser Algorithmus hat keine polynomielle Laufzeit
- Aber: die exponentielle Abhängigkeit der Laufzeit ist auf den Parameter  $k$  eingeschränkt

- Ein Algorithmus mit einer Laufzeitschranke von der Form  $f(k)n^c$  wird fp-effizient (bezogen auf den Parameter  $k$ ) genannt



Englisch:

fpt  $\equiv$  fixed parameter tractable

- Solche Algorithmen werden wir im nächsten Abschnitt genauer betrachten
- Vorher benötigen wir aber einige Definitionen



# Parametrisierte Probleme

## Definition: Parametrisiertes Problem

- Ein **parametrisiertes Problem** ist eine Menge  $Q$  von Paaren  $(x, k)$ , wobei  $x \in \Sigma^*$  und  $k \in \mathbb{N}$
- $\underline{Q_k} \stackrel{\text{def}}{=} \{(x, i) \in Q \mid i = k\}$   
☞ „ $k$ -te Scheibe von  $Q$ “

## Definition: P-VERTEXCOVER

**Gegeben:** Graph  $G = (V, E)$ ,  $k$

**Parameter:**  $k$

**Frage:** Gibt es  $U \subseteq V$ , so dass gelten:

- (1)  $(u, v) \in E \Rightarrow u \in U$  oder  $v \in U$
- (2)  $|U| = k$ ?

✎ Dass die Zielmenge genau  $k$  Elemente umfassen soll, ist nicht besonders wichtig

- Das werden wir aus technischen Gründen bei allen parametrisierten Problemen so handhaben

- Bei der Definition parametrisierter Probleme werden wir im Folgenden den ausgewählten Parameter gesondert ausweisen
- Dem Problemnamen stellen wir ein P- vor
- Zu beachten: algorithmische Probleme lassen sich in der Regel auf verschiedene Weisen parametrisieren
  - Für VERTEXCOVER kämen zum Beispiel auch als Parameter in Frage:
    - \* der maximale Knotengrad in  $G$
    - \* das minimale Geschlecht einer Fläche, in die  $G$  eingebettet werden kann
    - \* der Durchmesser von  $G$
- Wenn das gleiche algorithmische Problem auf verschiedene Arten parametrisiert wird, müssen die Varianten in der Notation natürlich unterschieden werden

## Definition: **FPT**

- Ein parametrisiertes Problem  $Q$  heißt **fp-effizient**, falls es einen Algorithmus  $\mathcal{A}$ , eine Konstante  $c$  und eine berechenbare Funktion  $f : \mathbb{N} \rightarrow \mathbb{N}$  gibt, so dass gilt:
  - (a)  $\mathcal{A}$  entscheidet  $Q$
  - (b)  $\mathcal{A}$  hat bei Eingabe  $(x, k)$  Laufzeit  $\leq f(k)|x|^c$
- **FPT** ist die Klasse aller fp-effizienten parametrisierten Probleme

- Also: P-VERTEXCOVER  $\in$  **FPT**

## Bemerkungen

- Es gibt einige Varianten in der Definition von **FPT**
- Für die meisten **FPT**-Probleme ist  $f$  eine exponentielle Funktion, das ist aber nicht erforderlich
- Die Wahl von  $f(k) + |x|^c$  statt  $f(k)|x|^c$  würde zum selben Begriff führen
- Zur Notation:
  - Bei Graphproblemen verwenden wir die Knotenzahl  $n$  als Repräsentant der Eingabegröße
  - Bei „allgemeinen“ Problemen wird die Länge der Eingabe  $x$  mit  $|x|$  bezeichnet

# Inhalt

- ▷ **23.1 Effizient Parametrisierbare Probleme**
- 23.2 Parametrisierte Komplexitätstheorie

## fp-effiziente Algorithmen

- Es gibt eine Vielzahl von Techniken zum Entwurf fp-effizienter Algorithmen
- Sehr häufig werden die folgenden beiden einfachen Techniken verwendet:
  - Kernreduktion
  - Verwendung kleiner Suchbäume
- Eine aktuelle Übersicht über andere Techniken findet sich im Buch von Downey und Fellows

# Vorberechnungen und Kern-Reduktionen: Vorüberlegungen

- Es ist bei vielen Algorithmen eine Selbstverständlichkeit, im Rahmen einer Vorberechnung die gegebene Eingabe anhand einfacher Kriterien zu modifizieren und dabei schon eine partielle Lösung zu berechnen

## Beispiel

- 3-COL:
  - Knoten vom Grad  $\leq 2$  können zunächst aus dem Graphen entfernt werden
  - Ist der restliche Graph zulässig gefärbt, können diese Knoten dann auf einfache Weise zulässig gefärbt werden
- SAT:
  - In Klauseln der Länge 1 muss das einzige Literal wahr werden
  - Die Belegung für seine Variable steht also fest
  - Alle Vorkommen der Variable können also entfernt und die Formel entsprechend vereinfacht werden
- HAMILTONCYCLE: Pfade mit mehreren Knoten vom Grad 2 können durch Pfade mit einem Knoten ersetzt werden

- Der Ansatz der „Kern-Reduktionen“ versucht die Möglichkeit von Vorberechnungen auszureizen und so zu fp-effizienten Algorithmen zu kommen:
  - Die eigentliche Eingabe soll dabei durch eine (Teil-) Eingabe (den **Kern**) ersetzt werden, deren Größe nur vom Parameter  $k$  abhängt

# Kern-Reduktion: P-VERTEXCOVER

- Wir betrachten jetzt eine Kernreduktion für P-VERTEXCOVER

## Beispiel: P-VERTEXCOVER

- **Grundidee:**
  - Jeder Knoten mit  $\text{Grad} > k$  muss in ein Vertex-Cover  $U$  mit  $|U| \leq k$  aufgenommen werden
    - \* Anders sind seine anliegenden Kanten nicht überdeckbar,
    - \* da nicht alle Nachbarn aufgenommen werden können
- Das führt bei Eingabe  $(G, k)$  zu folgender Berechnung eines Graphen  $G'$ :
  - Berechne die Knoten  $v_1, \dots, v_p$  vom  $\text{Grad} > k$
  - Falls  $p > k$ , Antwort:  
 $(G, k) \notin \text{P-VERTEXCOVER}$
  - Entferne  $v_1, \dots, v_p$  und alle anliegenden Kanten aus  $G$
  - Entferne danach alle isolierten Knoten

## Beispiel (Forts.)

- Sei  $k' \stackrel{\text{def}}{=} k - p$
  - Falls  $G'$ 
    - mehr als  $k'(k + 1)$  Knoten oder
    - mehr als  $k'k$  Kanten hatgilt:  $(G, k) \notin \text{P-VERTEXCOVER}$
  - ➔ Weitere Berechnung nur nötig, wenn  $G'$ 
    - $\leq k'(k + 1)$  Knoten und
    - $\leq k'k \leq k^2$  Kanten hat
  - Es sind äquivalent:
    - $(G, k) \in \text{P-VERTEXCOVER}$
    - $(G', k') \in \text{P-VERTEXCOVER}$
  - Gesamtaufwand:  $k^{2k} + \mathcal{O}(n^2)$
- 
- Mit komplizierteren Methoden ist sogar ein Graph  $G'$  mit  $\leq 2k$  Knoten erreichbar
  - Aber: falls  $\mathbf{P} \neq \mathbf{NP}$ , so hat P-VERTEXCOVER keine Kern-Reduktion auf die Größe 1,  $36k$

# Vorberechnungen und Kern-Reduktionen

## Definition: kern-reduzierbar

- Ein parametrisiertes Problem  $Q$  ist kern-reduzierbar, wenn es berechenbare Funktionen  $f, g$  und eine in polynomieller Zeit berechenbare Funktion  $(x, k) \mapsto (x', k')$  gibt mit:
  - $|x'| \leq f(k)$ ,
  - $k' \leq g(k)$ , und
  - $(x, k) \in Q \iff (x', k') \in Q$
- Eine Kern-Reduktion ist also eine besondere Art der parametrisierten Selbst-Reduktion
- Kern-Reduktion ist eine der am häufigsten verwendeten Techniken für FPT-Algorithmen
- Es lässt sich sogar zeigen, dass jedes parametrisierte Problem mit einem fp-effizienten Algorithmus auch einen solchen Algorithmus hat, der auf Kern-reduktion beruht

# Kleine Suchbäume: Einleitung

- Der Suchbaum-basierte  $\mathcal{O}(2^k n^2)$ -Algorithmus für P-VERTEXCOVER lässt sich auf einfache Weise verbessern
- $N(v) \stackrel{\text{def}}{=} \text{Menge der Nachbarknoten von } v$
- **Grundidee:**
  - Ein Vertex-Cover muss jeweils  $v$  oder alle Knoten aus  $N(v)$  enthalten
- Der Algorithmus verwaltet immer ein aktuell zu lösendes Problem  $(G, k, U)$  (mit  $G=(V, E)$ ) und einer Menge  $U$  von Knoten, die schon in das zu berechnende Vertex-Cover aufgenommen sind  
 $(U \cap V = \emptyset)$
- Für eine Menge  $N$  von Knoten bezeichnet
  - $G - N$  den von  $V - N$  induzierten Teilgraphen,
  - $G - N$  das Teilproblem  
 $(G - N, k - |N|, U \cup N)$

## Algorithmus 23.1

- (0) Falls  $|V| \leq 3$  oder  $k \leq 1$ , löse das Problem direkt ⊕
  - (1) Falls möglich, wähle  $v$  vom Grad 1 und mache weiter mit  $G - N(v)$
  - (2) Falls möglich, wähle  $v$  vom Grad 2 mit verbundenen Nachbarn und mache weiter mit  $G - N(v)$
  - (3) Falls möglich, wähle  $v$  vom Grad 2 mit unverbundenen Nachbarn und verzweige zu (a)  $G - N(N(v))$  und (b)  $G - N(v)$
  - (4) Andernfalls wähle  $v$  vom Grad  $\geq 3$  und verzweige zu (a)  $G - \{v\}$  und (b)  $G - N(v)$
- Wenn als Parameter ein negativer Wert übergeben wird, liefert der Aufruf des Algorithmus die Antwort „nein“



# Analyse von Algorithmus 23.1 (1/2)

## Satz 23.2

1. Algorithmus 23.1 ist korrekt
2. Die Anzahl der Blätter im Suchbaum von Algorithmus 23.1 bei Eingaben mit Parameter  $k$  ist beschränkt durch  $1,47^k$

## Beweisskizze

- Die Korrektheit des Algorithmus kann durch eine Fallunterscheidung gezeigt werden
  - Begründung dafür, dass in (3a) die Menge  $N(N(v))$  gewählt wird:
    - \* Alternativ müsste  $v$  und ein Nachbar  $x$  von  $v$  aufgenommen werden
    - \* Das ist aber nicht besser als  $N(v)$  aufzunehmen, was durch (3b) schon erfasst ist

## Beweisskizze für (b)

- Algorithmus 23.1 verzweigt
  - (1) zu einem Teilproblem mit Parameter  $k - 1$ ,
  - (2) zu einem Teilproblem mit Parameter  $k - 2$ ,
  - (3) zu einem Teilproblem mit Parameter  $k - 2$  und einem mit Parameter  $\leq k - 2$ , oder
  - (4) zu einem Teilproblem mit Parameter  $k - 1$  und einem mit Parameter  $\leq k - 3$
- Mit  $T(k)$  bezeichnen wir die maximale Anzahl der Blätter eines Verzweigungsbaumes für Eingaben der Art  $(G, k)$
- Dann lässt sich in den vier Fällen die Anzahl der Blätter also abschätzen durch das Maximum von
  - (1)  $T(k - 1)$
  - (2)  $T(k - 2)$
  - (3)  $T(k - 2) + T(k - 2)$
  - (4)  $T(k - 1) + T(k - 3)$
- Außerdem ist  $T(1) = 1$
- Behauptung:  $T(k) \leq 1,47^k$

# Analyse von Algorithmus 23.1 (2/2)

## Beweisskizze (Forts.)

- Sei  $\alpha \stackrel{\text{def}}{=} 1,47$
- ➡  $\alpha \geq \sqrt{2}$
- Sei  $k > 1$ 
  - In den Fällen (1) und (2) ist die Anzahl der Blätter  $\leq T(k-1) \leq \alpha^{k-1} \leq \alpha^k$
  - Wenn der Algorithmus gemäß (3) verzweigt, ist die Anzahl der Blätter
$$\leq 2T(k-2) \leq 2\alpha^{k-2} \leq \alpha^k$$
(wegen  $2 \leq \alpha^2$ )
  - Wenn der Algorithmus gemäß (4) verzweigt, ist die Anzahl der Blätter
$$\leq T(k-1) + T(k-3) \leq \alpha^{k-1} + \alpha^{k-3} = \alpha^k$$

- Wie kommen wir auf 1,47?

 KoMTI

- Das Tafelbeispiel zu Algorithmus 23.1 hat zwar die Verzweigungsmöglichkeiten korrekt wiedergegeben, aber die Schlussfolgerung, dass die „9-Blätter“ korrekte Lösungen ergeben wa r nicht korrekt.

# Inhalt

23.1 Effizient Parametrisierbare Probleme

▷ **23.2 Parametrisierte Komplexitätstheorie**

# CLIQUE und DOMINATINGSET parametrisiert

- Wir wenden uns jetzt der Frage zu, ob die beiden übrigen algorithmischen Probleme auch fp-effizient sind

## Definition: P-CLIQUE

**Gegeben:** Graph  $G = (V, E)$ ,  $k$

**Parameter:**  $k$

**Frage:** Gibt es  $U \subseteq V$ , so dass gelten:

- (1) für je zwei Knoten  $u, v \in U$  ist  $(u, v) \in E$
- (2)  $|U| = k$ ?

## Definition: P-DOMINATINGSET


**Gegeben:** Graph  $G = (V, E)$ ,  $k$

**Parameter:**  $k$

**Frage:** Gibt es  $U \subseteq V$ , so dass gelten:

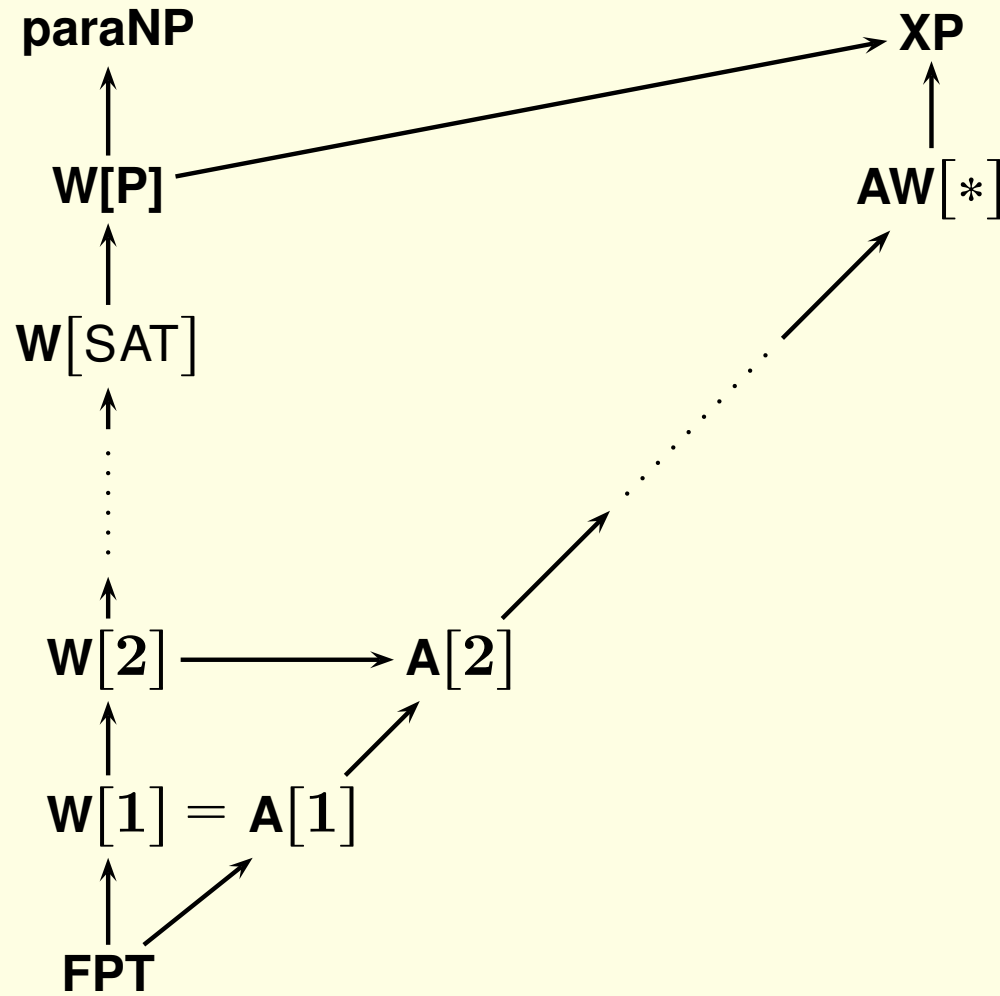
- (1) für jedes  $v \in V - U$  gibt es ein  $u \in U$  mit  $(u, v) \in E$
- (2)  $|U| = k$ ?

- Bisher wurden für diese Probleme keine fp-effizienten Algorithmen gefunden
- Es wird vermutet, dass sie keine fp-effizienten Algorithmen haben

 Ein Beweis dafür würde aber  $P \neq NP$  nach sich ziehen

- Die Klassifikation der Probleme, die nicht fp-effizient zu sein scheinen, ist deutlich komplizierter als in der klassischen Komplexitätstheorie:
  - Parametrisierte Versionen **NP**-vollständiger Probleme verteilen sich auf eine Vielzahl parametrisierter Komplexitätsklassen

# Übersicht parametrisierter Komplexitätsklassen



# FPT-Reduktionen (1/2)

- Für die präzise Definition von Vollständigkeit für parametrisierte Komplexitätsklassen, aber auch für die Definition der W- und A-Hierarchie, benötigen wir einen passenden Reduktionsbegriff

## Definition

- Seien  $Q, Q'$  mit Parametern  $k, k'$
- Eine **FPT-Reduktion** von  $Q$  auf  $Q'$  ist eine Funktion  $R : (x, k) \mapsto (x', k')$  mit:
  - $(x, k) \in Q \iff (x', k') \in Q'$
  - $R$  ist in Zeit  $f(k)|x|^c$  berechenbar für eine Konstante  $c$
  - Es gibt eine berechenbare Funktion  $g : \mathbb{N} \rightarrow \mathbb{N}$  mit  $k' \leq g(k)$
- Schreibweise:  $Q \leq_{\text{fpt}} Q'$

## Beispiel

- $P\text{-CLIQUE} \leq_{\text{fpt}} P\text{-INDEPENDENTSET}$
- $G = (V, E)$  wird abgebildet auf den Komplementgraphen  $\overline{G}$  (und  $k$ )
- Klar:  $G$  hat genau dann eine  $k$ -Clique, wenn  $\overline{G}$  ein Independent Set mit  $k$  Knoten hat

## FPT-Reduktionen (2/2)

### Beispiel

- $P\text{-CLIQUE} \leq_{\text{fpt}} P\text{-DOMINATINGSET}$
- Zu  $G = (V, E)$  (und  $k$ ) definiere  $G' = (V', E')$  wie folgt:  $\boxplus$
- $V' \stackrel{\text{def}}{=} [k] \cup (V \times [k]) \cup \{(u, v, i, j) \mid (u, v) \notin E, i \neq j\}$   
 $\Rightarrow [k] \stackrel{\text{def}}{=} \{1, \dots, k\}$
- $E' \stackrel{\text{def}}{=} \{((v, i), i) \mid v \in V, i \in [k]\} \cup$   
 $\{(u, i), (v, i)\} \mid u \neq v, i \in [k]\} \cup$   
 $\{(u, v, i, j), (w, i)\} \mid u \neq w \neq v\} \cup$   
 $\{(u, v, i, j), (w, j)\} \mid u \neq w \neq v\}$
- Behauptung: dies ist eine FPT-Reduktion von P-CLIQUE auf P-DOMINATINGSET  $\Rightarrow$  KoMTI

## Einschub: Beschreibungskomplexität

- Ein Graph ist genau dann 3-färbbar wenn er folgende Formel erfüllt:

$$\begin{aligned} \exists X_1, X_2, X_3 \quad \forall x \quad & (X_1(x) \vee X_2(x) \vee X_3(x)) \wedge \\ & [\forall x, y \quad (X_1(x) \wedge X_1(y)) \vee (X_2(x) \wedge X_2(y)) \\ & \vee (X_3(x) \wedge X_3(y)) \rightarrow \neg E(x, y)] \end{aligned}$$

- Existenzielle Logik zweiter Stufe (ESO):

- Formeln der Art  $\psi = \exists X_1, \dots, X_r \varphi$
- $X_i$ : Relationenvariablen,  $\varphi$  Formel erster Stufe

### Satz 23.3 [Fagin 75]

- $L \in \mathbf{NP}$  genau dann, wenn es eine ESO-Formel  $\psi$  gibt, so dass für alle  $x$  gilt:  $x \in L \iff x \models \psi$

### Beweisidee

„ $\Rightarrow$ “: Die  $X_i$  kodieren die Berechnungstabelle der TM für  $L$  bei Eingabe  $x$

- Es genügt dafür sogar eine Relation

„ $\Leftarrow$ “:  $M$  erwartet in Zusatzeingabe Relationen für  $X_1, \dots, X_r$

- Startpunkt der Theorie der **Beschreibungskomplexität**
- Viele andere Komplexitätsklassen lassen sich durch Klassen logischer Formeln charakterisieren



# Die W-Hierarchie (1/2)

- Der Satz von Fagin gibt uns einen Ansatz zur Definition von **NP**-Optimierungsproblemen durch die Mächtigkeit von „Zeugenrelationen“

## Definition: P-WD $_{\psi}$

**Gegeben:** Struktur  $\mathcal{A}$ ,  $k$

**Parameter:**  $k$

**Frage:** Gibt es  $X$  mit  $|X| = k$  und  $(\mathcal{A}, X) \models \psi$ ?

## Beispiel

- P-CLIQUE ist P-WD $_{\psi_1}$  mit 
$$\psi_1 = \forall x \forall y [(X(x) \wedge X(y)) \rightarrow (x = y \vee E(x, y))]$$
- P-DOMINATINGSET ist P-WD $_{\psi_2}$  mit 
$$\psi_2 = \forall x \exists y [X(y) \wedge (x = y \vee E(x, y))]$$

- **Beobachtung:** unterschiedliche Quantorenstruktur in  $\psi_1$  und  $\psi_2$
- Genauer: in  $\psi_1$  nur Allquantoren, in  $\psi_2$  ein Quantorenwechsel

## Definition

- $\Pi_t$ : Formeln erster Stufe in Pränex-Form, beginnend mit  $\forall$ -Quantoren und mit  $t - 1$  Wechseln zwischen  $\forall$  und  $\exists$

## Definition: W-Hierarchie

- Für  $t \geq 1$  sei 
$$\underline{W}[t] \stackrel{\text{def}}{=} \{Q \mid Q \leq_{\text{fpt}} \text{P-WD}_{\psi}, \psi \in \Pi_t\}$$
- **W-Hierarchie:**  $\underline{W}[1] \subseteq \underline{W}[2] \dots$
- Also:
  - P-CLIQUE  $\in \underline{W}[1]$
  - P-DOMINATINGSET  $\in \underline{W}[2]$

## Die W-Hierarchie (2/2)

- P-CLIQUE ist vollständig für  $W[1]$
  - P-DOMINATINGSET ist vollständig für  $W[2]$
- 
- P-WSAT(2CNF) ist vollständig für  $W[1]$
  - P-WSAT(CNF) ist vollständig für  $W[2]$

# Ein Anderer Zugang: Die A-Hierarchie

- P-CLIQUE lässt sich auch auf eine andere Art durch logische Formeln charakterisieren
- $G$  hat eine  $k$ -Clique genau dann, wenn  $G$  die folgende Formel erfüllt:

$$\exists x_1, \dots, x_k \left[ \left( \bigwedge_{i \neq j} x_i \neq x_j \right) \wedge \left( \bigwedge_{i \neq j} E(x_i, x_j) \right) \right]$$

→ P-CLIQUE lässt sich also als Model-Checking-Problem für Formeln erster Stufe auffassen

## Definition: P-MC( $\mathcal{L}$ )

**Gegeben:** Struktur  $\mathcal{A}$ , Formel  $\varphi \in \mathcal{L}$

**Parameter:**  $|\varphi|$

**Frage:** Gilt  $\mathcal{A} \models \varphi$ ?

- Zu beachten:
  - P-WD $_{\psi}$ : Formel ist fest
  - P-MC( $\mathcal{L}$ ): Formel ist Teil der Eingabe

## Definition

- $\Sigma_t$ : Formeln erster Stufe in Pränex-Form, beginnend mit  $\exists$ -Quantoren und mit  $t - 1$  Wechseln zwischen  $\forall$  und  $\exists$

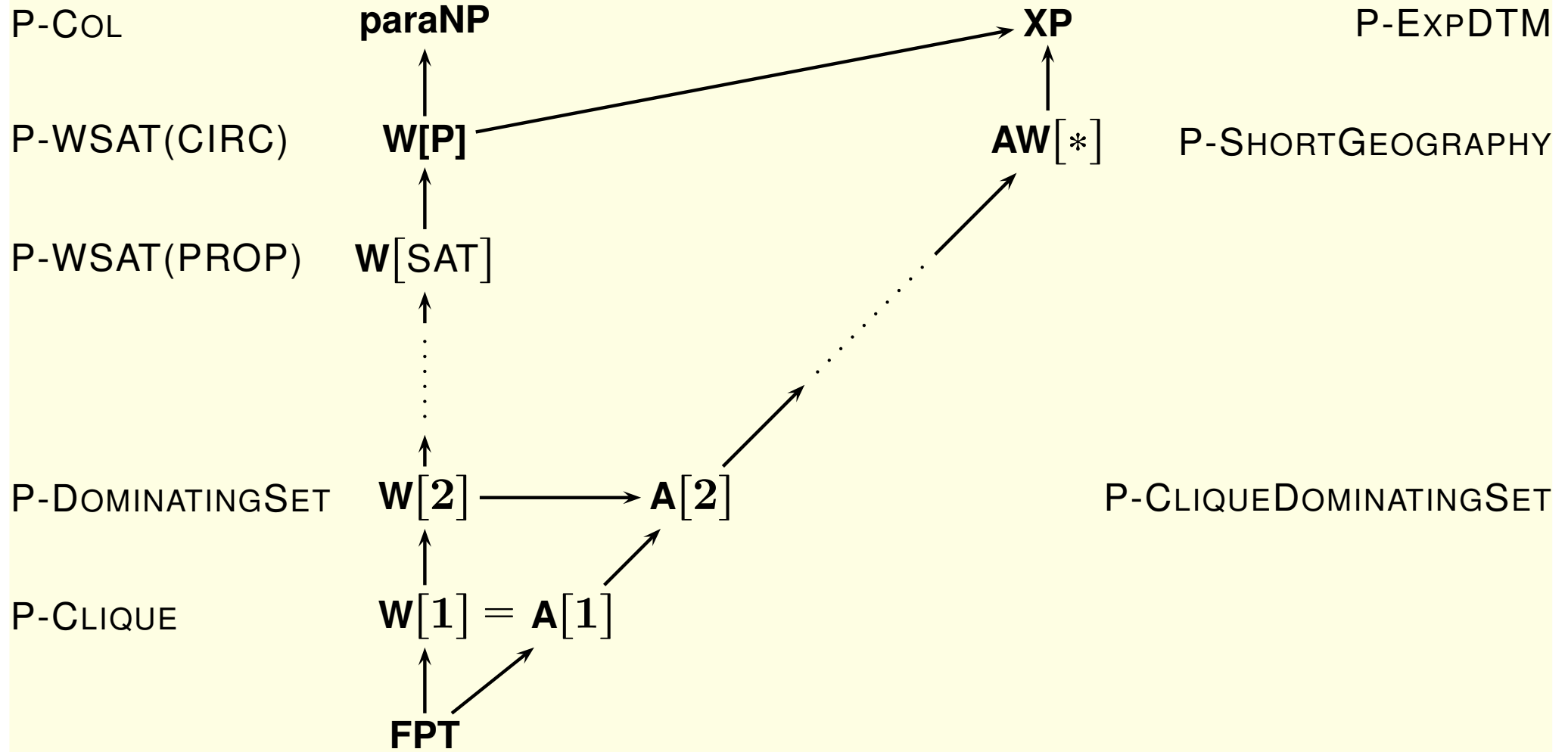
$$\underline{\mathbf{A}}[t] \stackrel{\text{def}}{=} \{Q \mid Q \leq_{\text{fpt}} \text{P-MC}(\Sigma_t)\}$$

- Es gilt:
  - Für jedes  $t$  ist  $\mathbf{W}[t] \subseteq \mathbf{A}[t]$
  - $\mathbf{W}[1] = \mathbf{A}[1]$

## Namen sind ...

- Einige Bemerkungen zu den Akronymen:  
P-WD <sub>$\psi$</sub> : Gewichtete Fagin-Definierbarkeit  
WSAT: Gewichtete Erfüllbarkeit  
PROP: Menge aller aussagenlogischen Formeln  
**W**[ $t$ ]: historisch:  $t$ -te Stufe der „Weft-Hierarchie“  
**A**[ $t$ ]: Ursprünglich mit Bezug auf ATMs definiert  
**W**[ $P$ ]: Gewichtete Schaltkreiserfüllbarkeit

# Parametrisierte Klassen mit vollständigen Problemen



## FPT vs. $W[1]$

- Falls  $P = NP$  ist  $W[1] = FPT$ , denn:
  - Angenommen:  $P = NP$
  - ➡ Das Cliquesproblem ist in polynomieller Zeit entscheidbar
  - ➡  $P\text{-CLIQUE} \in FPT$
  - ➡  $W[1] = FPT$ , da  $P\text{-CLIQUE}$   $W[1]$ -vollständig ist
- Es ist nicht bekannt, ob umgekehrt aus  $P \neq NP$  schon  $W[1] \neq FPT$  folgt
- Aber:
  - Exponentialzeit-Hypothese (ETH): Die Erfüllbarkeit aussagenlogischer Formeln mit  $n$  Variablen kann *nicht* in Laufzeit  $2^{o(n)}$  entschieden werden
- Falls ETH wahr ist, gilt  $W[1] \neq FPT$   
[Chen et al., 2006]

# Quellen

- **Lehrbücher**

- Rolf Niedermeier. *Invitation to fixed-parameter algorithms*. Oxford University Press, 2006
- J. Flum and M. Grohe. *Parameterized Complexity Theory*. Springer-Verlag, 2006
- Rodney G. Downey and Michael R. Fellows. *Fundamentals of Parameterized Complexity*. Texts in Computer Science. Springer, 2013

- **Originalarbeiten**

- Jianer Chen, Xiuzhen Huang, Iyad A. Kanj, and Ge Xia. Strong computational lower bounds via parameterized complexity. *J. Comput. Syst. Sci.*, 72(8):1346–1367, 2006

- **Weitere Quellen**

- Dániel Marx: *Fixed Parameter Algorithms*, Vortragsfolien, <http://www.cs.bme.hu/~dmarx/papers/marx-tractability-slides.pdf>