UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR
THEORETISCHE INFORMATIK

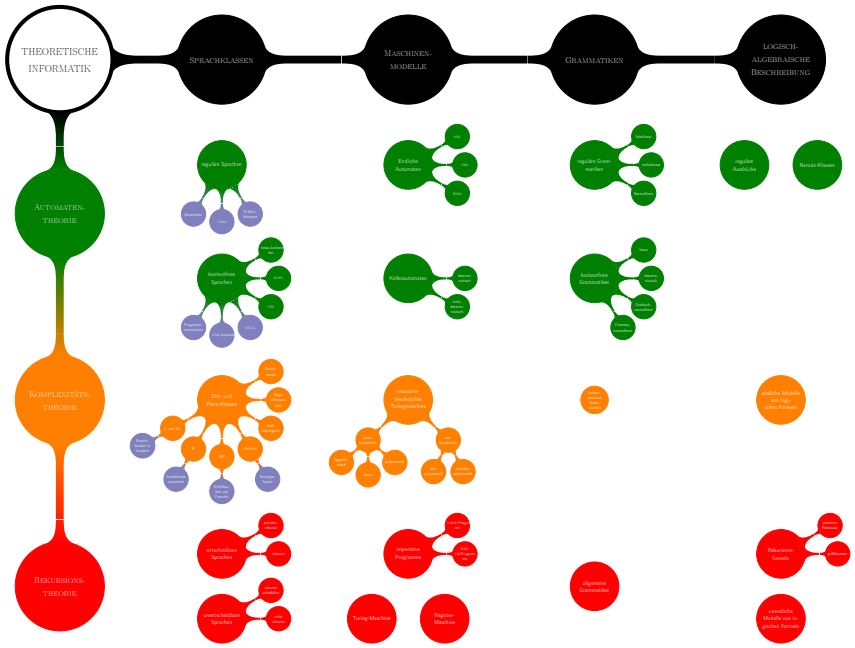
Vorlesungsskript

Theoretische Informatik

, Wintersemester 2009

Fassung vom 25. November 2018

Till Tantau



UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR THEORETISCHE INFORMATIK

PROF. DR. TILL TANTAU
WWW.TCS.UNI-LUEBECK.DE
25. NOVEMBER 2010

Veranstaltungskarte
Wintersemester 2009

THEORETISCHE INFORMATIK

Veranstaltungsziele
- Theoretisch solid fundierender Syntax- und deroperationalen Semantiken Programmiersprachen kennen
- Möglichkeiten und Grenzen der Informatik verstehen
- Algorithmische Probleme modellieren und mit geeigneten Werkzeugen lösen können
- Algorithmische Probleme nach ihrer Komplexität klassifizieren können

Sprachklassen

Maschinenmodelle

Grammatiken

logisch-algebraische Beschreibung

Syntaxbeschreibung

Wie analysiert man Syntax?

November 2009

- Einführung, Organisatorisches
- Alphabet, Wort, Sprache
- Grammatiken
- Die Chomsky-Hierarchie
- Deterministische endliche Automaten
- Grenzen regulärer Sprachen
- Nichtdeterministische endliche Automaten
- Zwei-Wege-Automaten
- Reguläre Ausdrücke
- Nerode-Klassen
- Kontextfreie Grammatiken
- Analyse kontextfreier Sprachen

Was ist berechenbar?

Dezember 2009

- Maschinen I: Die Turingmaschine
- Maschinen II: Nichtdeterminismus
- Maschinen III: Register-Maschinen
- Programme I: loop- und while-Programme
- Programme II: Rekursion
- Die Church-Turing-These

Wie schwierig sind Probleme?

Januar 2010

- Unentscheidbarkeit I: Das Halteproblem
- Unentscheidbarkeit II: Satz von Rice
- Unentscheidbare Probleme
- Die O-Notation
- Einführung zur Komplexitätstheorie
- Die Idee der Reduktion
- Die Idee der Vollständigkeit

Februar 2010

- NL-Vollständigkeit
- P-Vollständigkeit
- NP-Vollständigkeit
- Das P-NP-Problem

Klausur

Automatentheorie

Komplexitätstheorie

Rekursionstheorie

1 Alphabet, Wort, Sprache
- Mathematische Objekte als Worte kodieren können
- Entscheidungsprobleme als Sprachen formalisieren können
- Grundoperationen auf Sprachen anwenden können

2 Grammatiken
- Begriffe der Regel und der Ableitung verstehen
- einfache Syntaxprobleme mittels Grammatiken beschreiben können
- Korrektheitsbeweise führen können

3 Die Chomsky-Hierarchie
- Stufen der Hierarchie kennen
- Grammatiken der richtigen Stufe zuordnen können
- Sprachklassen und Abgeschlossenheit verstehen

4 Deterministische endliche Automaten
- Syntax und Semantik von dfas beherrschen
- mit Programmiertools dfas beschreiben können
- dfas implementieren können

5 Nichtdeterministische endliche Automaten
- Syntax und Semantik von nfas beherrschen
- Äquivalenzbeweis zu dfas verstehen

6 Grenzen regulärer Sprachen
- Aussage des Pumping Lemmas für reguläre Sprachen verstehen
- Das Pumping-Lemma anwenden können, um Nichtregularität zu beweisen

7 Zwei-Wege-Automaten
- Syntax und Semantik von 2nfas beherrschen
- Äquivalenzbeweis zu dfas verstehen
- Äquivalenzbeweis zu regulären Grammatiken verstehen

8 Reguläre Ausdrücke
- Verschiedene Syntax-Varianten kennen
- Semantik regulärer Ausdrücke verstehen
- Syntaxprobleme mittels regulärer Ausdrücke beschreiben können
- Äquivalenzbeweis zu regulären Sprachen verstehen

9 Nerode-Klassen
- Begriff Rechts-Index und die Invariante verstehen
- Den Index einer Sprache bestimmen können
- Den Satz von Myhill-Nerode und seinen Beweis verstehen
- Automaten minimieren können

10 Kontextfreie Grammatiken
- Beispiele kontextfreier Sprachen kennen
- kontextfreie Ansätze von Sprachen aus der Praxis identifizieren können
- mehrere Ansätze für eine kontextfreie Sprache verstehen und vergleichen können
- Normalformen für kontextfreie Sprachen kennen

11 Analyse kontextfreier Sprachen
- Beweistechniken von Cocke, Younger und Kasami anwenden können
- Syntax und Semantik von Pushdown-Automaten verstehen
- Äquivalenz zu kontextfreien Grammatiken kennen
- Konzepte der deterministisch kontextfreien Grammatik verstehen

12 Kontextsensitive Grammatiken

13 Die O-Notation
- Definition verschiedener Varianten der O-Notation verstehen
- Algorithmen anhand ihrer O-Klassen klassifizieren können
- Wichtige O-Klassen ihr Verhältnis lernen

14 Einführung zur Komplexitätstheorie
- Konzepte der zeit- und platzbeschränkten Turingmaschine verstehen
- Wichtige deterministische Zeit- und Platzklassen kennen
- Sprachen in die Zeit-Platz-Hierarchie einordnen können

15 NL-Vollständigkeit
- Nichtdeterministischen Platz verstehen
- Das Erreichbarkeitsproblem nach kennen
- Bootstrapping- und Reduktionsmethoden kennen
- Beweis der NL-Vollständigkeit von reach verstehen
- Eigenschaften NL-vollständiger Probleme führen können

16 P-Vollständigkeit
- Mathematische Definition von Schaltkreisen kennen
- Das Circuit-Value-Problem cvp kennen
- Beweis der P-Vollständigkeit von cvp verstehen

17 NP-Vollständigkeit
- Nichtdeterministische Zeit verstehen
- Das Erfüllbarkeitsproblem sat kennen
- Satzes von Cook und seinen Beweis verstehen

18 Die Church-Turing-These
- Geschichte der Turingmaschinen kennen
- Aussage der Church-Turing-These kennen
- Begriff der «Turing-Mächtigkeit» kennen

19 Unentscheidbarkeit I: Das Halteproblem
- Das Halteproblem verstehen
- Den Beweis der Unentscheidbarkeit verstehen
- Eigentlich triviale Beweistechnik verführen können
- Aufbau und Funktion einer universellen Turingmaschine verstehen

20 Unentscheidbarkeit II: Satz von Rice
- Den Satz von Rice und seinen Beweis verstehen
- Konsequenzen für praktische Problemstellungen kennen
- Den Rekursionssatz und den Fixpunktsatz kennen

21 Unentscheidbare Probleme
- Das Postsche Korrespondenzproblem kennen
- Das Busy-Beaver-Problem kennen
- Das Konzept der Kolmogorov-Komplexität kennen

22 Einführung zur Komplexitätstheorie
- Konzepte der zeit- und platzbeschränkten Turingmaschine verstehen
- Wichtige deterministische Zeit- und Platzklassen kennen
- Sprachen in die Zeit-Platz-Hierarchie einordnen können

23 Die Idee der Reduktion
- Konzept der Reduktion allgemein verstehen
- Definition der Logspace-Many-One-Reduktion verstehen
- Reduktionen erstellen können

24 Die Idee der Vollständigkeit
- Die Zeit-Platz-Hierarchie kennen
- Konzepte der Schwere verstehen
- Konzepte der Vollständigkeit verstehen
- Wichtigkeit der Transitivität von Reduktionen kennen

25 Das P-NP-Problem
- Konzepte des Ressourcenbegrenzungen verstehen
- Mehrere NP-vollständige Probleme kennen
- Die Bedeutung des P=NP-Frage einschätzen können
- Gründe kennen, weshalb die Frage so schwierig zu lösen ist

26 P-Vollständigkeit

27 NP-Vollständigkeit

28 Unentscheidbarkeit II: Unentscheidbare Probleme

29 Unentscheidbarkeit III: Unentscheidbare Probleme

30 Die O-Notation

31 Einführung zur Komplexitätstheorie

32 Die Idee der Reduktion

33 Die Idee der Vollständigkeit

34 NL-Vollständigkeit

35 P-Vollständigkeit

36 NP-Vollständigkeit

37 Das P-NP-Problem

38 Programme I: loop- und while-Programme
- Syntax und Semantik von loop- und while-Programmen verstehen
- Äquivalenzbeweis zu Turingmaschinen verstehen

39 Maschinen II: Nichtdeterminismus
- Nichtdeterminismus bei Turingmaschinen verstehen
- ntms entwerfen können
- Konzepte nichtdeterministischen Rechnens verstehen
- Äquivalenzbeweis zu deterministischen Maschinen lernen

40 Maschinen I: Die Turingmaschine
- Syntax und Semantik der Turingmaschinen verstehen
- Begriff der Konfiguration verstehen
- einfache Turingmaschinen erstellen können
- Korrektheitsbeweise führen können
- Äquivalenz entscheidbaren und der aufzählbaren Sprachen kennen

41 Maschinen III: Register-Maschinen
- Syntax und Semantik der rams verstehen
- Einfache ram-Programme schreiben können
- Äquivalenzbeweis zu Turingmaschinen verstehen

42 Programme II: Rekursion
- Syntax und Semantik der primitiven und der µ-rekursiven Funktionen kennen
- Äquivalenzbeweis zu loop- und while-Programmen verstehen

Endliche Automaten
- dfa
- nfa
- 2nfa

reguläre Sprachen
- Bezeichner
- urls
- E-Mail-Adressen

Kellerautomaten
- deterministisch
- nicht-deterministisch

kontextfreie Sprachen
- linear-kontextfrei
- dcfl
- cfl
- Programmtext-struktur
- xml-Instanzen
- html

reguläre Grammatiken
- linkslinear
- rechtslinear
- Normalform

kontextfreie Grammatiken
- linear
- deterministisch
- Greibach-normalform
- Chomsky-normalform

kontextsensitive Grammatiken

reguläre Ausdrücke

Nerode-Klassen

Einführung zur Komplexitäts-theorie
- Reduktionen
- Abgeschlossenheit
- Vollständigkeit
- L und NL
- P
- NP
- pspace
- Erreichbarkeit in Graphen
- Schaltkreise auswerten
- Erfüllbarkeit von Formeln
- Strategie-Spiele

ressourcenbeschränkte Turingmaschine
- platzbeschränkt
- zeitbeschränkt
- logarithmisch
- linear
- polynomiell
- det. polynomiell
- nichtdet. polynomiell

entscheidbare Sprachen
- primitiv-rekursiv
- rekursiv

unentscheidbare Sprachen
- rekursiv aufzählbar
- nicht rekursiv

imperative Programme
- loop-Programme
- while-Programme

Turing-Maschine

Register-Maschine

Rekursionsformeln
- primitive Rekursion
- µ-Rekursion

endliche Modelle von logischen Formeln

allgemeine Grammatiken

unendliche Modelle von logischen Formeln

Inhaltsverzeichnis

Teil III
Was ist berechenbar?

Vorwort

In der Theoretischen Informatik geht es darum, sich nicht um A20-Gates kümmern zu müssen. Falls Sie das A20-Gate nicht kennen, so kann man dies zu einem nicht unerheblichen Teil als Erfolg der Theoretischen Informatik werten:

Die Geschichte des A20-Gates beginnt mit dem berühmten Ausspruch von Bill Gates (Microsoft-Begründer und einer der reichsten Menschen der Welt): »Man wird nie mehr als 640KB Hauptspeicher benötigen.« Überraschenderweise benötigte man dann doch mehr Hauptspeicher. Daraufhin entwarf Intel den 80286 Prozessor, den Nachfolger des 8086, der sehr zukunftsorientiert bis zu 16MB ansprechen konnte. Dabei entstanden aber ein Problem: Der 8086 Prozessor konnte ja nur ein Megabyte ansprechen, aber aufgrund einer etwas abenteuerlichen Speicheradressierung es war möglich, Adressen zwischen 1MB und 1MB plus 64KB anzusprechen – die es physikalisch gar nicht gab und deshalb auf die ersten 64KB im Speicher abgebildet wurden. Nun gab es tatsächlich auch einige (obskure) Programme, die sich darauf verlassen haben und bei Adressen zwischen 1MB und 1MB plus 64KB stattdessen die ersten Stellen im Speicher erwarteten. Beim 80286 konnte man aber so nicht mehr weitermachen – schließlich war jetzt ja physikalisch bis zu 16MB möglich. Gelöst wurde da Problem auf eine originelle Art: Mittels des so genannten »A20-Gate« konnte man das System anweisen, temporär die zwanzigste Adressleitung einfach fest auf 0 zu setzen – und damit diesen wenigen Uraltprogrammen einen Überlauf vorzugaukeln. Noch origineller war der Umstand, das Management des A20-Gates dem Tastatur-Controller zu überlassen – weil hier gerade noch Platz war. Da natürlich auch 16MB nicht ausreichten, folgte dem 80286 der 80386, der die 32-Bit-Ära einläutete. Er brachte ein völlig neues (und viel besseres) Speichermanagement mit, erlaubte aber auch noch den klassischen so genannten »real mode«, in dem der 80286 emuliert wurde, der wiederum das A20-Gate benötigte. Es folgte des 80486 und dann der 80586, der aber Pentium getauft wurde, da Intel damit scheiterte, sich die Zahl 80586 als Handelsmarke zu schützen. Den großen Wurf wollte Intel dann mit der radikal neuen Itanium-Architektur schaffen – um aber alte Software weiter laufen zu lassen spendierte Intel auf diesen Prozessoren in einer Ecke auf etwa 1mm^2 noch einen kompletten Pentium-Emulator – selbstverständlich inklusive A20-Gate.

Moderne Prozessoren, aber auch Betriebssysteme und selbst moderne Programmiersprachen, sind voll von Wunderlichkeiten wie dem A20-Gate. Die Komplexität eines modernen Prozessors oder eines modernen Betriebssystems komplett zu verstehen ist praktisch nicht mehr möglich. Das macht es auch unmöglich verlässliche Aussagen darüber zu machen, was ein konkretes System kann und nicht kann. Vor 20 Jahren konnte man noch im Handbuch nachschauen, wie viele Taktzyklen ein bestimmter Befehl *genau* dauern wird – im Zeitalter hyperskalarer Pipelines mit Translation-Lookaside-Buffern ein hoffnungsloses Unterfangen.

In der Theoretischen Informatik geht es darum, sich von allem Ballast zu befreien und sich auf den Kern dessen zu konzentrieren, was einen Computer wirklich ausmacht. Deshalb werden in der Theoretischen Informatik *abstrakte* und *mathematische* Modelle von Computern untersucht und nicht konkrete Computer. Aus diesem Grund wird die Theoretische Informatik häufig als »wenig praktisch« wahrgenommen – ein Vorwurf, der etwas ins Leere läuft, da das ja gerade das Ziel ist.

Es gibt sehr viele unterschiedliche Dinge betreffend einen Computer, die man abstrakt und mathematisch untersuchen kann. Uns wird in dieser Vorlesung zentral interessieren, wie Computer *ganz allgemein gesprochen* rechnen. In dieser Vorlesung wird es um folgende drei zentrale Fragen gehen:

1. *Wie analysiert man Syntax?*
   Betrachten Sie bitte folgenden Programmtext:

```
int foo(int[] a  ) {int u =  a[0]; for (int i = 0; i<
```

```
a.length; i++)if(a[i] > u) u =  a[i]; return u;}
```

Sie haben den Programmtext nicht sofort verstanden? Dann geht es Ihnen wie einem Computer. Wir rücken Programmtexte gerne ein, um sie für Menschen besser lesbar zu machen; aus Sicht des Computers sehen sie eher wie obiger aus. Genaugenommen ist die Situation sogar noch schlimmer, aus Sicht des Computers ähnelt der Text eher ägyptischen Hieroglyphen, schließlich *versteht* der Computer die Text erstmal nicht.
Der erste Schritt auf dem Weg zu einer Berechnung geht deshalb über die Syntax. Wir müssen den Computer in die Lage versetzen, Programmtext überhaupt zu *analysieren*, man spricht auch vom *Parsen*. Genau dieselben Ideen werden benutzt, um eine HTML-Seite zu parsen oder auch um Benutzer-Eingaben zu validieren. Im ersten Teil der Vorlesung wird untersucht werden, was sich ganz allgemein über dieses Parsen aussagen lässt. Insbesondere wird uns interessieren, was *prinzipiell* möglich ist und was eher nicht.

2. *Was ist berechenbar?*
   Wer ein wenig Erfahrung mit Programmieren hat, wird bestätigen können, dass sich eigentlich jedes Problem mit einem Computer lösen lässt – es ist alles nur eine Frage des Programmieraufwands und der Rechenzeit. Beispielsweise ist die Simulation des Weltklimas im Computer über die nächsten hundert Jahre derzeit noch nicht exakt möglich – aber nicht etwa, weil man niemand wüsste wie man Algorithmen für die Lösung der Weltklima-Differentialgleichungen programmiert; das Problem ist einfach, dass unsere Rechner nicht fett genug sind. Ebenso ist die Graphik in Computerspiele derzeit eben doch nicht photorealistisch, aber wieder nicht, weil man nicht wüsste, wie ein Radiocity-Render zu programmieren wäre, sondern weil die Graphikkarten nicht schnell genug sind.
   Es gibt aber recht praktische Probleme, die sich *prinzipiell* nicht mit Computern lösen lassen. Das liegt nicht etwa daran, dass die Probleme zu unpräzise wären oder dass man phantastisch viel Zeit bräuchte, sondern daran, dass es *prinzipiell nicht geht*.
3. *Wie schwierig sind Probleme?*
   Im letzten Teil der Vorlesung geht es um die Frage, wie effizient bestimmte Probleme lösbar sind. Klar ist, dass manche Problem leichter sind als andere (das Suchen in einer sortierten Liste ist sicherlich einfacher als das Lösen eines komplexen Differentialgleichungssystems). Man kann diese Intuition auch wieder exakter fassen, was uns in das Gebiet der Komplexitätstheorie führt.
   Quintessenz der Komplexitätstheorie für den Hausgebrauch ist, dass man Probleme, die NP-schwer sind, nicht effizient *exakt* lösen kann und dass man für diese Probleme häufig auf Heuristiken ausweichen muss. Ob ein Problem NP-schwer ist, sieht man entweder durch scharfes Hinschauen oder man schaut in einer Tabelle nach.

Nach diesem Überblick über die Inhalte der Veranstaltung wird es höchste Zeit, sich den Zielen zu widmen. Jedes Kapitel dieses Skripts entspricht in der Regel einer Vorlesungsdoppelstunde und mit jedem Kapitel verfolge ich gewisse Ziele, welche Sie am Anfang des jeweiligen Kapitels genannt finden. Neben diesen etwas kleinteiligen Zielen gibt es auch folgende zentralen Veranstaltungsziele, zitiert aus dem Modulhandbuch:

1. Theoretische Grundlagen der Syntax und der operationalen Semantik von Programmiersprachen kennen
2. Möglichkeiten und Grenzen der Informatik verstehen
3. Algorithmische Probleme modellieren und mit geeigneten Werkzeugen lösen können
4. Algorithmische Probleme nach ihrer Komplexität klassifizieren können

Das Wörtchen »können« taucht bei den Veranstaltungszielen recht häufig auf. Um etwas wirklich zu können, reicht es nicht, davon gehört zu haben oder davon gelesen zu haben. Man muss es auch wirklich *getan* haben: Sie können sich tausend Fußballspiele im Fernsehen anschauen, sie sind deshalb noch kein guter Fußballspieler; sie können tausend Stunden World of Warcraft spielen, sie werden deshalb trotzdem keinen Frostblitz auf Ihren Professor geschleudert bekommen.

Deshalb steht bei dieser Veranstaltung der Übungsbetrieb mindestens gleichberechtigt neben der Vorlesung. Der Ablauf ist dabei folgender: In der Vorlesung werde ich Ihnen die Thematik vorstellen und Sie können schon mit dem Üben im Rahmen kleiner Miniübungen *während der Vorlesung* beginnen. Jede Woche gibt es ein Übungsblatt, das inhaltlich zu

den Vorlesung gehört. Sie müssen sich die Übungsblätter aber nicht »alleine erkämpfen«. Vielmehr gibt es Tutorien, in denen Sie Aufgaben üben werden, die »so ähnlich« wie die Aufgaben auf den Übungsblättern sind. Sie werden feststellen, dass die als »leicht« und in der Regel auch die als »mittel« eingestuften Aufgaben mit der Vorbereitung im Tutorium in der Tat mit vertretbarem Aufwand schaffbar sind. Ist eine Aufgabe »schwer«, so ist es kein Unglück, wenn Sie diese nicht schaffen – probieren sollten Sie es aber trotzdem.

Ich wünsche Ihnen viel Spaß mit dieser Veranstaltung.

Till Tantau

Teil I
Syntaxbeschreibung

Auch wenn die Bezeichnung »Computer« suggeriert, dass Computer zum Rechnen gedacht seinen, ist die *Syntaxanalyse* eine ihrer Hauptbeschäftigungen. Ununterbrochen müssen Computer Texte »lesen« und »verstehen«, beispielsweise beim

- Abarbeiten eines Skriptes, beginnend mit dem Boot-Skript,
- Einlesen des Quelltextes einer Webseite,
- Ausführen des LUA-Skripts in einem Computerspiel,
- Übersetzen eines Programms,
- Lesen einer Konfigurationsdatei,
- Öffnen und Speichern von Dokumenten,
- Überprüfung von Nutzereingaben, …

Die Liste ließe sich endlos fortsetzen. Bevor wir uns im nächsten Teil dieser Veranstaltung der Frage widmen, wie diese Syntaxanalyse algorithmisch funktioniert, müssen wir zunächst ein wenig Ordnung in die Sache bringen.

Man wäre schlecht beraten, wenn man für jede der obigen Aufgaben das Rad der Syntaxbeschreibung und -analyse neu erfinden würde. Vielmehr ist es wünschenswert, die immer gleichen Syntaxprobleme *einheitlich* zu beschreiben, damit man sie später auch *einheitlich* lösen kann. Genau darum geht es in diesem ersten Teil: Wie beschreibt man Syntax möglichst einheitlich?

1-1

Kapitel 1
Alphabet, Wort, Sprache
Eine einheitliche Sicht auf die Problem dieser Welt

Lernziele dieses Kapitels

1. Mathematische Objekte als Worte kodieren können
2. Entscheidungsprobleme als Sprachen formalisieren können
3. Grundoperationen auf Sprachen anwenden können

1-2

Inhalte dieses Kapitels

Worum es heute geht

Jeder, der schon einmal versucht hat, einen SSH-Tunnel korrekt zu konfigurieren oder das Löschen aus einem AVL-Baum in C zu implementieren, wird folgender empirischen Beobachtung uneingeschränkt zustimmen: »Computer sind dafür da, Probleme zu lösen, die man ohne sie nicht hätte.« In diesem Kapitel geht es darum, »Probleme« formal zu fassen, um sie einer formalen Analyse (sprich: mathematischen) zugänglich zu machen – unabhängig davon, ob wir das Problem nun auch ohne Computer hätten oder nicht.

Es hat sich herausgestellt, dass sich der Begriff der *formalen Sprache* besonders gut eignet, um Probleme in ganz allgemeiner und trotzdem einfacher Art zu beschreiben. Historisch kommt der Begriff aus der Linguistik (wenig überraschend) und bei »Sprache« denkt man im Allgemeinen eher nicht an »Probleme« (bei »Fremdsprache« hingegen schon eher). Lassen Sie sich von dem Begriff nicht irritieren: Immer wenn in diesem Skript von »Sprache« die Rede ist, könnte genausogut »Problem« oder »Problemstellung« stehen – in der Theorie ist es aber einfach üblich, von Sprachen zu sprechen. In der Datenbank-Theorie heißen Sprachen übrigens »Boolean queries«, was auch nicht wirklich besser ist.

1.1 Zur Einstimmung

1-4

Welche Informatik-Probleme umranken einen Programmtext?

Ein zentraler Bestandteil der Informatik ist die Untersuchung von Programmtexten wie dem folgenden:

```
for (int i = 1; i < 1000000000; i++) {
  int n = i;
  while (n != 1)
    if (n % 2 == 0) n = n / 2;
    else            n = 3*n + 1;
}
```

Es gibt viele Aspekte/Fragestellungen/Probleme, die man bei diesem Programmtext untersuchen kann:

– Ist das Programm syntaktisch korrekt?

– Was macht das Programm?

✎ Zur Diskussion
Welche weiteren Fragestellungen fallen Ihnen ein?

1-5 **Was ist eine »Fragestellung« oder ein »Problem« im Informatik-Kontext allgemein?**

– Das Wort »Problem« ist eine nicht ganz glückliche Übersetzung des englischen »problem«. Exakter, aber auch nicht viel besser, wären »Aufgabe«, »Aufgabenstellung« oder »Problemstellung«.
– Generell geht es darum, zu vielen unterschiedlichen so genannten *Probleminstanzen*, kurz *Instanzen*, jeweils eine *Lösung* zu finden.

Unterscheiden kann man Probleme zunächst grob danach, welche *Art* von Probleminstanzen und Lösungen möglich sind.

Mögliche Probleminstanzen

– Zahlen und Zahlentupel
– Formeln
– Graphen
– Texte
– …

Mögliche Lösungen

– »Ja«/»Nein«
– Zahlen und Zahlentupel
– Texte
– Belegungen
– …

1-6 **Instanz- und Lösungsarten einiger Informatik-Probleme.**

Beispiel: Das Sortier-Problem

Instanzen Tupel von Zahlen
Lösungen Tupel von Zahlen

Beispiel: Das Primzahl-Problem

Instanzen Zahlen
Lösungen »Ja«/»Nein«

Beispiel: Das E-Mail-Adressen-Problem

Instanzen Texte
Lösungen »Ja«/»Nein«

1.2 Wie formalisiert man Probleme?

Auf dem Weg zum formalen Problem.

1-7

- Es gibt sehr viele mögliche Arten von Instanzen und Lösungen.
- In der Theoretischen Informatik möchten wir möglichst allgemeine Aussagen machen – es wäre unschön, wenn wir jeden Satz für jede Art von Instanzen neu beweisen müssten.
- Für die Theorie brauchen deshalb eine *möglichst einfache Art von Instanzen*, mit der sich *trotzdem alle Probleme beschreiben lassen*.

1.2.1 Probleminstanzen sind Worte

Wie kodiert man Probleminstanzen?

1-8

- Es gibt viele Arten von Probleminstanzen: Zahlen, Tupel, Texte, Graphen, Matrizen, etc.
- Wie schon Alan Turing feststellte, müssen aber all diese Probleminstanzen letztendlich irgendwie *aufgeschrieben* werden.
- Es liegt folglich nahe, *nur noch Texte als Probleminstanzen* zuzulassen – alles andere muss dann eben als Text aufgeschrieben werden.

Beispiel: Zahlen als Texte

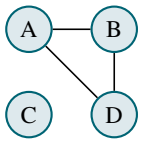
- Aus der Zahl »eine Million« wird die Zeichenfolge `1000000`.
- Aus der Zahl »drei Halbe« wird die Zeichenfolge `1.5`.

✎ **Zur Diskussion**
Machen Sie Vorschläge, wie man die Zahlen »ein Drittel«, »Wurzel aus Zwei« und »Pi« kodieren könnte.

Beispiel: Zahlentupel als Texte

- Aus dem Paar $(3,2)$ wird die Zeichenfolge `(3,2)` oder alternativ `3#2`.
- Aus dem Tupel $(1,100,0,0)$ wird die Zeichenfolge `(1,100,0,0)` oder alternativ `1#100#0#0`.

Beispiel: Graphen als Texte
Aus dem Graphen



wird

```
<graph>
  <node name="A"/>
  <node name="B"/>
  <node name="C"/>
  <node name="D"/>
  <edge n1="A" n2="B"/>
  <edge n1="A" n2="D"/>
  <edge n1="B" n2="D"/>
</graph>
```

1-9

Die Formalisierung von Worten I

▶ Definition: Alphabet
Ein *Alphabet* ist eine nichtleere endliche Menge. Die Elemente eines Alphabets heißen *Symbole*.

- Zum Aufschreiben von Alphabeten benutzt man in der Regel lateinische Großbuchstaben wie A, N oder T oder griechische wie Σ oder Γ.
- Beispiele von wichtigen Alphabeten sind das binäre Alphabet $\{0,1\}$, das genetische Alphabet $\{a,c,g,t\}$, das Dezimalalphabet $\{0,1,2,\ldots,9\}$, der ASCII-Code oder der UNICODE.
- Auch das *Leerzeichen* oder *Blanksymbol* ist ein mögliches Symbol, beschrieben $\square$ oder auch manchmal β.

1-10

Alphabete in der Praxis

Alphabete in XML

- Die Strukturierungssprache XML wird weltweit eingesetzt – und es gibt viele Alphabete auf der Welt.
- Der XML-Standard erlaubt es uns deshalb anzugeben, welches Alphabet wir nutzen wollen.
- Das verwendete Alphabet heißt allerdings *Encoding*. In folgendem Beispiel ist das Alphabet Σ gerade gleich UNICODE.

```
<?xml version="1.0" encoding="utf-8"?>
<rezept name="Kaiserschmarrn">
  <zutat> 150g Mehl </zutat>
  <zutat> 1/8l Milch </zutat>
  <zutat> 3 Eier (getrennt) </zutat>
  <zutat> Puderzucker und eine Prise Salz </zutat>
</rezept>
```

1-11

Die Formalisierung von Worten II

▶ Definition: Worte

- Ein *Wort über einem Alphabet* ist eine endliche Folge von Symbolen aus dem Alphabet.
- Das i-te Symbol eines Wortes w bezeichnen wir mit $w[i]$.
- Die leere Folge bezeichnen wir mit λ.

Bemerkungen:

- Andere Bezeichnungen für Worte sind »Zeichenkette« oder »String«.
- »Zweidimensionaler Text« lässt sich auch leicht als Wort darstellen, wenn ein spezielles »Zeilenumbruchzeichen« benutzt wird. (Leider sind sich Windows und Unix nicht einig, welches dies sein sollte…)
- Das Wort $\square$, das nur aus einem Blanksymbol besteht, hat die Länge 1. Das Wort λ hat hingegen die Länge 0.

1-12

Worte in der Praxis

Worte in XML

- Ein komplettes XML-Dokument ist ein *einziges Wort*.
- Der folgende Programmtext hat als »Wort« 226 Zeichen, inklusive 6 Zeilenendezeichen:

```
<?xml version="1.0" encoding="utf-8"?>
<rezept name="Kaiserschmarrn">
  <zutat> 150g Mehl </zutat>
  <zutat> 1/8l Milch </zutat>
  <zutat> 3 Eier (getrennt) </zutat>
  <zutat> Puderzucker und eine Prise Salz </zutat>
</rezept>
```

Mengen von Worten bestimmter Längen

1-13

Die *Länge eines Wortes* ist von *zentraler Bedeutung* in der Theorie: *Je Länger ein Wort, desto mehr Ressourcen werden in der Regel für seine Verarbeitung benötigt.*

▶ Definition: Länge eines Wortes
Die *Länge eines Wortes* w bezeichnen wir mit $|w|$.

▶ Definition: Mengen von Worten bestimmter Länge
Sei Σ ein Alphabet und $n \in \mathbb{N}$. Wir benutzen folgende Notationen:

Notation	Bedeutung
Σ^*	Menge aller Worte über Σ
Σ^n	Menge der Worte in Σ^* der Länge n
$\Sigma^{\leq n}$	Menge der Worte in Σ^* der Länge höchstens n
$\Sigma^{\geq n}$	Menge der Worte in Σ^* der Länge mindestens n

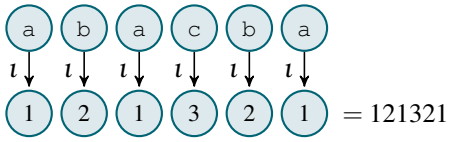
Von der Zahl zum Wort und zurück

1-14

Wir haben schon gesehen, dass man Zahlen sehr einfach als Worte über dem Alphabet $\{0,1,2,\ldots,9\}$ ausdrücken kann. Es geht aber auch umgekehrt:

▶ Definition: Worte als Zahlen

- Sei Σ ein Alphabet mit maximal neun Zeichen.
- Dann können wir mittels einer injektiven Funktion $\iota : \Sigma \to \{1,\ldots,9\}$ die Zeichen »durchnummerieren«.
- Jetzt kann man jedes Wort w über dem Alphabet leicht als Zahl darstellen: Aus `abacba` wird



$= 121321$

- Die Abbildung von Wort zu Zahl kann man auch schreiben als Injektion $\kappa\colon \Sigma^* \to \mathbb{N}$ mittels $\kappa(w) = \sum_{i=0}^{|w|-1} \iota(w[|w|-i]) \cdot 10^i$.

Worte oder Zahlen?

1-15

- Man kann sich *aussuchen*, ob man lieber *Worte* oder lieber *Zahlen* zur Kodierung von Instanzen benutzen wollen.
- Die Anfänge der Theorie in den 20er und 30er Jahren des letzten Jahrhunderts gingen von Mathematikern und insbesondere Logikern aus, die *lieber mit Zahlen gearbeitet haben*.
  Zahlen sind natürlich auch für das *Rechnen* passender.
- *Worte* sind hingegen viel praktischer, wenn Instanzen eine »interne Struktur« haben.
- Im Folgenden werden *in aller Regel nur Worte* benutzt, um Instanzen zu kodieren.

1.2.2 Operationen auf Worten

Was man alles mit Worten anstellen kann

1-16

Um mit Worten zu arbeiten, muss man sie *manipulieren* können. Hier gibt es viele Möglichkeiten:

▶ Definition: Verkettung von Worten
Seien $u, v \in \Sigma^*$ zwei Worte über dem Alphabet Σ. Dann bezeichnet $u \circ v$ die *Verkettung von u und v* (einfach u und v hintereinander geschrieben). Den Kringel $\circ$ lässt man häufig auch einfach weg.

Verkettet man w mit dem leeren Wort, so erhält man wieder w. (Für Profis: $(\Sigma^*, \circ)$ bildet ein Monoid mit dem leeren Wort als neutralem Element. Das Monoid ist kommutativ genau dann, wenn $|\Sigma| = 1$.)

▶ Definition: Potenzieren von Worten
Sei $w \in \Sigma^*$. Dann ist $w^2 = w \circ w$, das *Quadrat von w*, einfach w zweimal hintereinander geschrieben. Entsprechend ist w^n einfach w genau n-mal hintereinander geschrieben.
Man definiert $w^1 = w$ und $w^0 = \lambda$.

▶ Definition: Umdrehen von Worten
Sei $w \in \Sigma^*$. Dann bezeichnen sowohl w^{rev} wie auch w^{-1} das Wort w rückwärts geschrieben (»reversed«).

▶ Definition: Ersetzung von Buchstaben
Seien Σ und Γ zwei Alphabete. Eine *Ersetzung* ist eine Abbildung $h\colon \Sigma \to \Gamma^*$. Sie gibt an, wie Buchstaben aus dem ersten Alphabet durch Worte aus dem zweiten Alphabet zu ersetzen sind.
Eine Ersetzung h lässt sich zu einem *Homomorphismus* $\hat{h}\colon \Sigma^* \to \Gamma^*$ *fortsetzen*, indem man definiert $\hat{h}(w) := h(w[1]) \circ h(w[2]) \circ \cdots \circ h(w[|w|])$.
Man ersetzt also einfach buchstabenweise die Symbole des Wortes.

Beachte: Mengenoperationen wie Schnitt oder Vereinigung ergeben *keinen Sinn* für Worte. Worte sind keine Mengen.

1-17

Homomorphismen in der Praxis

Homomorphismen werden in der Praxis ständig gebraucht, sie heißen dort aber *Umkodierungen*:

- Wandelt man einen ASCII-kodierten Text in UNICODE-kodierten Text um, benutzt man dazu eine ganz einfache Ersetzung $h\colon$ ASCII $\to$ UNICODE, die jedem ASCII-Zeichen genau das zugehörige UNICODE-Zeichen zuordnet.
- Wandelt man UNICODE in ASCII um, so gibt es zwei Möglichkeiten:
  1. Die Ersetzung bildet die meisten Zeichen des UNICODE auf mehr oder weniger geeignete Zeichen des ASCII ab, im Notfall auf das Nullzeichen.
  2. Die Ersetzung ersetzt einzelne Zeichen des UNICODE durch mehrere Zeichen des ASCII – das passiert beispielsweise bei der UTF8-Kodierung.
- Aus den Ersetzungen entstehen nun Homomorphismen, die ASCII-Worte auf UNICODE-Worte abbilden oder umgekehrt.

1.2.3 Probleme sind Sprachen

1-18

Entscheidungsprobleme sind Sprachen

Zur Erinnerung:

- Bei einem Entscheidungsproblem hat man viele mögliche *Instanzen* als Eingabe.
- Das Problem ist nun, für jede Instanz eine der beiden möglichen Antworten »Ja« oder »Nein« richtig auszusuchen.

Wie formalisiert man Entscheidungsprobleme?

- Wir haben schon geklärt, wie wir Instanzen formalisieren: als Worte.
- Es bleibt zu klären, wie wir die Zuordnung von Instanzen zu »Ja« oder »Nein« modellieren. Hier gibt es zwei Möglichkeiten:
  1. Man formalisiert die Zuordnung als eine *charakteristische Funktion* $\chi\colon \Sigma^* \to \{0,1\}$, wobei 0 für »Ja« steht und 1 für »Nein«.
  2. Man benutzt einfach die Menge aller »Ja«-Instanzen.

  Offenbar sind beide Formalisierungen gleich mächtig, die *zweite ist aber praktischer*.

1-19

Definition formaler Sprachen

▶ Definition: Formale Sprache
Eine *formale Sprache A* über einem Alphabet Σ ist eine beliebige Teilmenge $A \subseteq \Sigma^*$.

Beispiel: Das E-Mail-Adressen-Problem
Sei $\Sigma =$ ASCII. Dann ist EMAIL-ADDRESS die Menge aller gültigen E-Mail-Adressen. So ist `tantau@tcs.uni-luebeck.de` ein Element der Sprache, hingegen `tantau@@foo` nicht.

Beispiel: Das Palindromproblem
Sei Σ ein festes Alphabet. Dann ist PALINDROMES die Menge aller Worte in Σ^*, die vorwärts wie rückwärts gleich sind.
Beachte: Für jedes Alphabet ergibt sich ein anderes Palindromproblem, man müsste eigentlich schreiben $\text{PALINDROMES}_\Sigma$. Das macht aber niemand.

Beispiel: Das Primzahlproblem
Sei $\Sigma = \{0,1,2,\ldots,9\}$. Dann ist PRIMES die Menge $\{2,3,5,7,11,13,\ldots\} \subseteq \Sigma^*$. Beachte: Diese Sprache enthält *Worte und nicht Zahlen* (auch wenn es so aussieht).

Beispiel: Das Java-Halteproblem
Sei $\Sigma = \text{UNICODE}$. Dann ist JAVA-HALTING die Menge aller Worte, die den Quelltext von Java-Programmen ohne Eingabe oder Ausgaben darstellen, die nach endlicher Zeit anhalten.

Und was ist mit Funktionsproblemen?

1-20

- In der Praxis sind die meisten Probleme gar keine Entscheidungsprobleme.
- Viel häufiger sind *Funktionsprobleme*, wo für eine Eingabe eine Ausgabe berechnet werden muss.
- Formal ist ein Funktionsproblem eine Funktion $f\colon \Sigma^* \to \Gamma^*$, die (Eingabe-)Worte auf (Ausgabe-)Worte abbildet.
- Man kann aber *jedes Funktionsproblem auf mehrere Arten in Entscheidungsprobleme umwandeln*.

▶ Definition: Die bin-Funktion
Die Funktion $\text{bin}\colon \mathbb{N} \to \{0,1\}^*$ bildet natürliche Zahlen auf ihre Darstellung als Binärstring ab.

Beispielsweise gilt $\text{bin}(5) = 101$ und $\text{bin}(0) = 0$.

▶ Definition: Sprache des Graphen und Test-Sprache
Sei $f\colon \Sigma^* \to \Gamma^*$ eine Funktion und sei $\#$ ein Symbol, das weder in Σ noch in Γ vorkommt. Dann ist

1. $\{w\#f(w) \mid w \in \Sigma^*\}$ die *Sprache des Graphen von* f.
2. $\{w\#\,\text{bin}(i)\#c \mid w \in \Sigma^*, i \in \mathbb{N}, c \in \Gamma, f(w)[i] = c\}$ die *Test-Sprache von* f.

1.2.4 Operationen auf Sprachen

Was man alles mit Sprachen anstellen kann I

1-21

- Genau wie man Worte manipulieren kann, kann man auch Sprachen manipulieren.
- Manipuliert man eine Sprache, so erhält man eine neue Sprache und damit auch ein *neues Problem*.

▶ Definition: Grundoperationen auf Sprachen
Gegeben seien zwei Sprachen $L, M \subseteq \Sigma^*$. Dann heißt:

	Bedeutung
$L \cup M$	Vereinigung der Sprachen
$L \cap M$	Schnitt der Sprachen
L^{rev}	Umdrehung der Sprache: die Menge $\{w^{\text{rev}} \mid w \in L\}$
$\overline{L}$	Komplement der Sprache L: die Menge $\Sigma^* - L$
$L \circ M$	Verkettung der Sprachen: $\{u \circ v \mid u \in L, v \in M\}$
L^n	n-te Potenz von L: Worte, die durch die Verkettung von genau n Worten aus L entstehen
L^*	Kleene-Stern der Sprache: beliebige Worte, die durch Verkettung von Worten aus L entstehen

Es gilt per Definition $\lambda \in L^*$.

1-22

Schnitt und Vereinigung in der Praxis

Datums-Validierung

In vielen Anwendungen, besonders bei Web-Anwendungen, müssen *Datumseingaben* der Benutzer validiert (=überprüft) werden.
Dieses Entscheidungsproblem lässt sich mittels mehrerer Einzeltests *modularisieren*:

- Test, ob die Eingabe nur aus Ziffern und Punkten besteht.
- Test, ob es genau zwei Punkte gibt.
- Test, ob der Wortanfang aus der Menge $\{1,\dots,31\}$ kommen.
- Test, ob nach dem ersten Punkte ein Wort aus $\{1,\dots,12\}$ kommt.

Jeder Test ergibt eine Menge an Worten, die diesen Test besteht, also eine Sprache. Der *Schnitt* dieser Sprachen ist dann der Gesamttest.
Will man noch ein alternatives Datumsformat zulassen (wie `2009-01-01`), so schreibt man neue Tests, schneidet diese und *vereinigt* dann die beiden entstehenden Sprachen.

1-23

✎ Zur Übung

Sei $\Sigma = \{0,1,2,\dots,9\}$ und $\textsc{primes} = \{2,3,5,7,11,\dots\}$.

1. Wie lautet $\overline{\textsc{primes}}$? Die richtige Lösung ist *nicht* $\{0,1,4,6,8,9,10,\dots\}$.
2. Geben Sie drei Worte an, die nicht in $\textsc{primes}^*$ liegen.
3. Geben Sie ein Wort in $\textsc{primes}^2 \cap \textsc{primes}$ an.

1-24

Der Kleene-Stern in der Praxis

Der Huffman-Code und der Kleene-Stern

- Der Huffman-Code ist eine Methode, Daten zu komprimieren.
- Daten werden über dem *Kodierungsalphabet* $\Gamma = \{0,1\}$ kodiert.
- Zu einem *Eingabealphabet* Σ wird eine (endliche) Sprache $L \subseteq \Gamma^*$ von *Codes* gebaut.
  Beispiel: Ist das Eingabealphabet $\Sigma = \{a,b,c,d,e\}$, so könnte $L = \{10,11,001,01,000\}$ sein.
- Worte aus L^* enthalten dann gerade hintereinander geschriebene Codes. Beispiel: $1011001 = 10 \circ 11 \circ 001 \in L^*$.
- Folglich sind die Worte in L^* gerade die in einer Huffman-Kodierung verwendeten komprimierten Worte.

1-25

Was man alles mit Sprachen anstellen kann II

▶ Definition: Bild einer Sprache

Sei $h\colon \Sigma^* \to \Gamma^*$ eine Funktion (beispielsweise ein Homomorphismus), $L \subseteq \Sigma^*$ und $M \subseteq \Gamma^*$. Dann bezeichnen

- $h(L) = \{h(w) \mid w \in L\}$ das *Bild von L unter h* und
- $h^{-1}(M) = \{w \mid h(w) \in M\}$ das *Urbild von M unter h*.

1-26

Bilder und Urbilder in der Praxis

Der Huffman-Code und Bilder und Urbilder

- Der Huffman-Code bildet Eingabesymbole auf Codeworte mittels einer Ersetzung ab.
  Beispiel: Ist das Eingabealphabet $\Sigma = \{a,b,c,d,e\}$, so könnte $h(a) = 10$, $h(b) = 11$, $h(c) = 001$, $h(d) = 01$ und $h(e) = 000$ gelten.
- Dann ist $\hat{h}(\Sigma^*)$ die Menge aller möglichen komprimierten Worte.
- Umgekehrt ist $\hat{h}^{-1}(\{w\})$ die Menge aller Eingabeworte, die komprimiert gerade w ergeben.
  Beispiel: $\hat{h}^{-1}(\{1011001\}) = \{abc\}$
  Beispiel: $\hat{h}^{-1}(\{0000\}) = \{\}$
- Der Huffman-Code ist gerade so gemacht, dass für jedes Wort $w \in \Gamma^*$ gilt $|\hat{h}^{-1}(w)| \leq 1$.
  Jedes Wort lässt sich also eindeutig dekodieren.

Zusammenfassung dieses Kapitels

1. Praktisch alle in der Informatik untersuchten Probleme lassen sich als *Sprachen* beschreiben.
2. Sprachen sind Mengen von Worten, die ihrerseits Folgen von Symbolen sind.
3. Worte wie Sprachen lassen sich *manipulieren* durch *Operationen*.

Zum Weiterlesen

[1] J. Hopcroft, R. Motwani, J. Ullman. *Introduction to Automata Theory, Languages and Computation*, Addison Wesley, 2002.
[2] Ch. Papadimitriou. *Computational Complexity*, Addison Wesley, 1994.
[3] R. Reischuk. *Unterlagen zum Lehrmodul Theoretische Informatik,* Vorlesungsskript, 2008. Kapitel 1.1 und 1.2.
[4] U. Schöning. *Theoretische Informatik – kurzgefasst*, Spektrum Akademischer Verlag, 1997.

Übungen zu diesem Kapitel

Übung 1.1 Kombinatorik auf Worten, leicht

Sei Σ ein Alphabet der Größe s, also $|\Sigma| = s$. Bestimmen Sie die Mächtigkeit (Anzahl der Elemente) der folgenden Mengen:

1. Σ^n (Worte der Länge genau n),
2. $\Sigma^{\leq n}$ (Worte der Länge höchstens n),
3. $\Sigma^{\leq n} \cap \Sigma^{\geq m}$ (Worte der Länge zwischen m und n).
4. Menge der Worte der Länge genau s, in denen jedes Zeichen genau einmal vorkommt.
5. Menge der Unteralphabete $\Sigma' \subseteq \Sigma$ mit $|\Sigma'| = 4$.
6. $\textsc{palindromes} \cap \Sigma^n$ (Menge der Palindrome der Länge genau n).

Übung 1.2 Kombinatorik auf Worten, mittel

1. Sei Σ ein Alphabet der Größe k. Bestimmen Sie die Mächtigkeit folgender Mengen:
   1.1 Menge der Strings der Länge n, in denen kein Zeichen zwei mal vorkommt.
   1.2 Menge der Strings der Länge n, in denen genau ein Zeichen fünf mal vorkommt und die anderen Zeichen jeweils genau ein mal.
2. Waren lassen sich durch so genannte Barcodes kennzeichnen. Ein Codewort ist eine Folge von schmalen und breiten Strichen mit schmalen oder breiten Lücken dazwischen. Beantworten Sie folgende Fragen:
   2.1 Wie viele verschiedene Codewörter lassen sich mit n Strichen erzeugen?
   2.2 Wie viele Möglichkeiten sind es, wenn Codewörter, die durch Vertauschung der Leserichtung (rechts $\leftrightarrow$ links) ineinander übergehen, nicht unterschieden werden?

Übung 1.3 Worte als Zahlen darstellen, schwer

Sei Σ ein Alphabet und $\iota : \Sigma \to \{1, \ldots, |\Sigma|\}$ eine Bijektion. Sei $\kappa : \Sigma^* \to \mathbb{N}$ definiert durch $\kappa(w) = \prod_{i=1}^{|w|} p_i^{\iota(w[i])}$, wobei p_i die i-te Primzahl ist. Zeigen Sie, dass κ eine Injektion ist.
Tipps: Die Buchstaben ι und κ heißen »Iota« und »Kappa«. Falls Ihnen die Begriffe »Bijektion« und »Injektion« nicht klar sind, dann klären Sie diese zunächst. Nehmen Sie sich dann ein paar Beispiele her und untersuchen Sie, was κ bei diesen Beispielen macht. Sie werden für Ihren Beweis den Satz über die eindeutige Primfaktorzerlegung benötigen.

Übung 1.4 Einfache Operationen auf Sprachen, mittel

Sei $\Sigma = \{a\}$ und seien $L = \{a^{2n} \mid n \geq 1\}$ und $M = \{a^{3n} \mid n \geq 1\}$, die Sprachen über Σ, die genau die Wörter der Länge teilbar durch Zwei beziehungsweise Drei enthalten. Geben Sie folgende Mengen in symbolischer Mengenschreibweise an:

1. $L \cap M$,
2. L^{rev},
3. $\overline{L}$,
4. L^2,
5. $L \circ M$,
6. L^*.

Übung 1.5 Homomorphie-Eigenschaft, mittel

Man sagt, eine Funktion $f\colon \Sigma^* \to \Gamma^*$ *hat die Homomorphie-Eigenschaft,* wenn für alle $u, v \in \Sigma^*$ gilt $f(u \circ v) = f(u) \circ f(v)$. Zeigen Sie, dass eine Funktion f genau dann die Homomorphie-Eigenschaft hat, wenn es eine Ersetzung $h\colon \Sigma \to \Gamma^*$ gibt mit $\hat{h} = f$.

Bemerkung: Die Aufgabe zeigt, dass man Homomorphismen auch einfach als Funktionen definieren könnte, die die Homomorphie-Eigenschaft haben.

Übung 1.6 Eindeutigkeit der Huffman-Kodierung, schwer

Sei $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e}\}$ und $\Gamma = \{0, 1\}$. Sei $h\colon \Sigma \to \Gamma^*$ definiert durch $h(\mathtt{a}) = \mathtt{10}$, $h(\mathtt{b}) = \mathtt{11}$, $h(\mathtt{c}) = \mathtt{001}$, $h(\mathtt{d}) = \mathtt{01}$ und $h(\mathtt{e}) = \mathtt{000}$. Zeigen Sie, dass $|\hat{h}^{-1}(\{w\})| \leq 1$ gilt für alle $w \in \Gamma^*$.

Tipp: Informieren Sie sich, wie der Huffman-Code funktioniert, insbesondere wie Huffman-Bäume aufgebaut sind. Zeigen Sie dann die Behauptung durch Induktion über die Länge von w. Für den induktiven Schritt zeigen Sie, dass die ersten Zeichen von w dieses eindeutig zerlegen.

Übung 1.7 Funktionen berechnen mit Test-Sprachen, leicht

Sie möchten eine recht kniffelige Funktion $f\colon \{\mathtt{a}, \ldots, \mathtt{z}\}^* \to \{\mathtt{a}, \ldots, \mathtt{z}\}^*$ berechnen mittels einer Java-Methode

```
String f (String w)
```

Diese soll, angewendet auf eine Zeichenkette w gerade $f(w)$ zurückliefern. In einer Software-Bibliothek gibt es glücklicherweise bereits eine Methode

```
boolean test (String s)
```

die die Test-Sprache von f entscheidet. Implementieren Sie `f` unter Benutzung der Methode `test`.

Übung 1.8 Funktionen berechnen mit der Sprache des Graphen, mittel

Wie in Aufgabe 1.7 soll die Funktion f implementiert werden. Diesmal steht aber in der Software-Bibliothek eine Entscheidungsmethod für den Graph von f zur Verfügung:

```
boolean graph (String s)
```

Implementieren Sie `f` unter Benutzung der Methode `graph`.

2-1

Kapitel 2
Grammatiken
Am Anfang war das Startsymbol

Lernziele dieses Kapitels

1. Begriffe der Regel und der Ableitung verstehen
2. einfache Syntaxprobleme mittels Grammatiken beschreiben können
3. Korrektheitsbeweise führen können

2-2

Inhalte dieses Kapitels

Worum es heute geht

In der Theoretischen Informatik sind Grammatiken *das* Vehikel, um die Syntax von allem und jedem zu beschreiben. Zur Erinnerung: die Syntax einer (Programmier)sprache beschreibt den Aufbau von Programmen (in der Theoretischen Informatik also »Worten«), die sich korrekt verarbeiten lassen – die Syntax sagt nichts aus über die Bedeutung dieser Worte, über ihre Semantik.

Es gibt mehrere Gründe, weshalb Grammatiken – nicht nur in der Theorie – so beliebt sind, um Syntax zu beschreiben:

1. Grammatiken erlauben es sehr einfach, den *hierarchischen Aufbau* der Syntax darzustellen. Da die meisten (Programmier)sprachen stark hierarchisch aufgebaut sind, ist dies recht wichtig.
2. Grammatiken sind zwar deskriptiv (sie »beschreiben« nur), jedoch lassen sie sich gut *automatisch* in Programme verwandeln, die Worte auf grammatikalische Korrektheit überprüfen.
3. Grammatiken sind sehr mächtig: jedes mit einem Computer prinzipiell lösbare Syntaxproblem lässt sich auch mit einer Grammatik beschreiben.

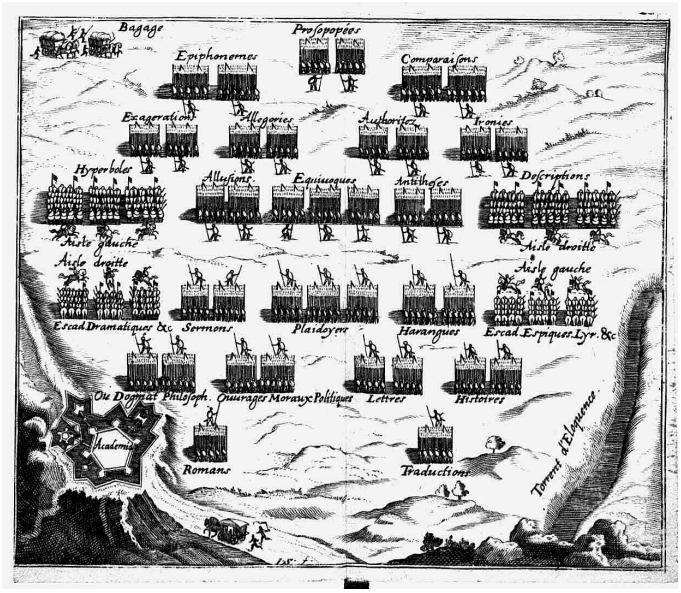
In der Praxis gibt es neben Grammatiken noch ein weiteres gängiges System zur Syntaxbeschreibung: reguläre Ausdrücke, denen später ein eigenes Kapitel gewidmet ist. Im Unterschied zu Grammatiken sind reguläre Ausdrücke weit weniger mächtig (die genaue Mächtigkeit wird noch eingehend erörtert werden), jedoch oft viel kompakter als Grammatiken.

2.1 Einleitung

2-4

Die Kunst des Lesens und Schreibens

- Grammatiken (»Die Kunst des Lesens und Schreibens«) gibt es seit der Antike.
- Sie behandelt das Problem, den *Aufbau von Texten* zu beschreiben und zu analysieren.



Public domain

Allegorische Darstellung der Grammatik, ihre Disziplinen als Armeen. Aus Antoine Furetières Nouvelle *Allegorique, Ou Histoire Des Derniers Troubles Arrivez Au Royaume D'Eloquence* (1659).

2.1.1 Grammatiken für natürliche Sprachen

2-5

Was beschreibt eine Grammatik alles?

Eine der Hauptaufgabe von Grammatik ist, den *hierarchischen Aufbau von Sätzen* allgemein zu beschreiben.

Beispiel: Typische Grammatikregeln im Deutschen

1. Ein Satz besteht aus einem Subjekt und einem Prädikat.
2. Ein Subjekt kann ein Substantiv zusammen mit einem passenden Artikel sein.
3. Jedem Substantiv können Adjektive vorangestellt werden.
4. ...

- Grammatikregeln sagen nur etwas über den erlaubten *Aufbau* aus und *nichts über den Inhalt*.
- So ist der Satz »Die Informatik blökt herzerweichend.« grammatikalisch richtig, aber inhaltlich eher zweifelhaft.

Merke

Grammatiken beschreiben Syntax, nicht Semantik (=Bedeutung).

2-6

Ein Beispiel, wie man Grammatikregeln anwendet.

Anwenden von Regeln (Ableiten)
Subjekt
⇒ Artikel Adjektivliste Substantiv
⇒ Artikel Adjektivliste Reiter
⇒ Der Adjektivliste Reiter
⇒ Der Adjektiv Reiter
⇒ Der blaue Reiter

Beobachtungen

2-7

- Grammatikregeln sind *Ersetzungsregeln*.
- In einem noch nicht fertigen Satz wie »Der Adjektiv Student« gibt es Teile, die noch ersetzt werden, und Teile, die nicht mehr ersetzt werden.
- Die Teile, die nicht weiter ersetzt werden, nennt man *Terminale*.
- Die Teile, die noch weiter ersetzt werden, nennt man *Nonterminale*.

Beachte: Ersetzungsregeln reichen nicht aus, um die gesamte Komplexität natürlicher Sprache zu beschreiben.

2.1.2 Grammatiken für Programmiersprachen

Von der antiken Grammatik zur formalen Grammatik

2-8

- Genau wie natürliche Sprachen sind auch *Computersprachen* nach *hierarchischen Regeln* aufgebaut.
- Tatsächlich sind Computersprachen nach viel strikteren Regeln aufgebaut als natürliche Sprache: Bekanntermaßen reicht schon ein vergessenes Semikolon, um einen Programmtext unbrauchbar zu machen.
- Es liegt also nahe, Systeme zur Beschreibung von (natürlichen) Grammatiken auch auf die Beschreibung der Grammatik von Programmtexten anzuwenden.
- Die wichtigsten und einfachsten Systeme sind *Semi-Thue-Systeme*, im Folgenden einfach *formale Grammatiken* genannt.

Die Grammatik von Programmiersprachen

2-9

- Bei Programmiersprachen beschreibt die Grammatik, wie Programmtexte gebildet werden können.
- Es gibt verschiedene Ansätze, aber ein gängiger ist, Programmtexte durch *Regeln* zu beschreiben.

Beispiel: Typische Grammatikregeln

1. Eine Klasse kann von der Bauart sein »`class` Bezeichner { Methodenliste }«.
2. Ein Bezeichner ist eine Folge von Zeichen und Buchstaben, die mit einem Buchstaben beginnt.
3. Eine Methodenliste besteht aus vielen Methoden.
4. Eine Methode besteht aus einem Kopf und einem Rumpf.
5. ...

Ein Beispiel, wie man Grammatikregeln anwendet.

2-10

Anwenden von Regeln (Ableiten)

Klasse
⇒ `class` Bezeichner { Methodenliste }
⇒ `class Love` { Methodenliste }
⇒ `class Love` { Methode }
⇒ `class Love { Love () {}}`

Beobachtungen

2-11

- Wieder sind Grammatikregeln *Ersetzungsregeln*.
- Wieder gibt es Teile, die noch ersetzt werden, und Teile, die nicht mehr ersetzt werden.
- Die Teile, die nicht weiter ersetzt werden, nennt man wieder *Terminale*.
- Die Teile, die noch weiter ersetzt werden, nennt man wieder *Nonterminale*.

Beachte: Wieder reichen (einfache) Ersetzungsregeln nicht aus, um die gesamte Komplexität von Programmiersprachen zu beschreiben.

2.2 Formale Grammatiken

2.2.1 Definition: Regeln

2-12

Formale Definition von Regeln

▶ Definition: Formale Grammatik

Eine *formale Grammatik* G besteht aus vier Teilen:

1. Einem *Terminalalphabet* T.
2. Einem *Nonterminalalphabet* N, das von Terminalalphabet disjunkt ist.
3. Einem *Startsymbol* $S \in N$.
4. Einer endlichen Menge P von *(Produktions-)Regeln*, jede von der Form:

$$l \to r$$

wobei $l, r \in (N \cup T)^*$ und l mindestens ein Nonterminalsymbol enthält.

Bemerkungen zur Definition:

- Eine Formulierung wie »Foo besteht aus vier Teilen« bedeutet mathematisch, dass Foo ein Vier-Tupel ist.
  - Deshalb schreibt man oft »Sei $G = (T, N, S, P)\ldots$«
  - Damit ist gemeint: »Sei mal G eine beliebige Grammatik. Dann muss sie ja (wie jede Grammatik) ein Vier-Tupel sein. Die erste Komponente nennen wir einfach mal T, die zweite N, die dritte S und die vierte P.«
    Für weitere Information über die Frage, wie man Dingen Namen gibt, sei auf die Reflexionen des Walfisch im Buch »Per Anhalter durch die Galaxis« von Douglas Adams verwiesen.
- »Disjunkt« bedeutet, dass N und T keine Elemente gemeinsam haben, also $N \cap T = \emptyset$.
- Eine Regel ist, genau genommen, ein Paar (l, r) von Worten $l, r \in (N \cup T)^*$. Das schreibt man aber nicht so hin, sondern eben als $l \to r$.
- Den Umstand, dass l ein Nonterminal enthalten muss, kann man kurz und unverständlich auch schreiben als $l \in (N \cup T)^* \circ N \circ (N \cup T)^*$.

Alternative Schreibweisen

Für die vier Teile einer Grammatik gibt es in der Literatur viele unterschiedliche Schreibweisen:

hier	andere Schreibweisen in der Literatur
T	Σ_T, G_T
N	Σ_N, G_N
$T \cup N$	Σ
S	A (»Anfang«), σ_0 (»Nullter Ableitungschritt«), G_S
P	E (»Ersetzungsregeln«), Π (»P auf Griechisch«), G_P
l	ℓ (besser lesbar), φ_1
r	φ_2

2-13

Ein ganz einfaches Beispiel.

Ausführliche Beschreibung einer Grammatik

Die Grammatik G hat das Terminalalphabet $T = \{a, b\}$. Das Nonterminalalphabet lautet $N = \{S\}$, enthält nur ein Symbol, nämlich S. Dieses ist auch gleich das Startsymbol, wir benutzen dafür auch den Buchstaben S (wie verwirrlich). Schließlich gibt es noch zwei Produktionsregeln, nämlich $S \to aSb$ und $S \to \lambda$.

Ganz formale Schreibweise derselben Grammatik

$G = \big(\{a, b\}, \{S\}, S, \{(S, aSb), (S, \lambda)\}\big)$

Normale Schreibweise derselben Grammatik

$G\colon S \to aSb \mid \lambda$

- Wenn nichts anderes angegeben wurde, dann ist S immer das Startsymbol.
- Wenn nichts anderes angegeben wurde, dann sind die in den Regeln vorkommenden Großbuchstaben die Nonterminale. Alle anderen Symbole sind Terminale.
- Haben mehrere Regeln dieselbe linke Seite, so schreibt man diese nur einmal und trennt die rechten Seiten durch einen senkrechten Strich (in TEX bitte den Befehl `\mid` verwenden).

✎ Zur Übung 2-14

Schreiben Sie folgende Grammatik »ganz formal als Vier-Tupel« auf:

$$G\colon S \to 0S0 \mid 1S1 \mid X$$
$$X \to 0 \mid 1$$

2.2.2 Definition: Ableitungen

Ableitungen = Anwenden von Regeln 2-15

Idee hinter Ableitung

- Eine Grammatik kann man nun benutzen, um *Ableitungen zu bilden*.
- Beginnend mit dem Startsymbol ersetzt man immer wieder linke Regelseiten durch rechte Regelseiten.
- Man endet, wenn nur noch Terminale vorhanden sind, das resultierende Wort nennt man ein *abgeleitetes Wort*.

Eigentlich wollen wir gar nicht ableiten

- Man beachte: In der Praxis haben wir ja schon ein Wort gegeben (beispielsweise einen Programmtext).
- Das eigentliche Ziel ist dann herauszufinden, durch welche Ableitung(en) dieser Programmtext entstanden ist.
- Diesen Vorgang nennt man *Parsen*. Darum kümmern wir uns noch ausführlich, aber nicht jetzt.

Formale Definition eines Ableitungsschritts 2-16

▶ Definition: Ableitungsschritt

Sei $G = (T, N, S, P)$ eine Grammatik. Seien $u, v \in (N \cup T)^*$. Wir sagen, dass v *in einem Ableitungsschritt aus* u *hervorgeht*, geschrieben $u \Rightarrow_G v$ oder einfach nur $u \Rightarrow v$, wenn

1. es eine Regel $l \to r$ in P gibt, so dass
2. sich u und v zerlegen lassen in der Form $u = x \circ l \circ y$ und $v = x \circ r \circ y$.

Bemerkungen zur Definition:

- Merke: In einem Ableitungsschritt wird aus einem Wort u eine linke Regelseite herausgestrichen und stattdessen die zugehörige rechte Regelseite eingesetzt. Das Ergebnis ist dann v.
- Wen es interessiert: $\Rightarrow_G$ setzt bestimmte Worte mit anderen in Relation, es handelt sich also um eine Relation auf Worten.
- Ganz formal gilt also: $\Rightarrow_G \subseteq (N \cup T)^* \times (N \cup T)^*$.
- Statt $\Rightarrow$ schreibt man manchmal auch $\vdash$ oder auch $\to$.

Beispiel

Sei $G\colon S \to aSb \mid \lambda$. Dann gilt:

$$aS \Rightarrow aaSb$$
$$bbbSSSaaa \Rightarrow bbbSaSbSaaa$$
$$aSb \Rightarrow ab$$
$$aSb \not\Rightarrow aSSb$$

2-17

Formale Definition abgeleiteter Worte.

▶ **Definition**

Sei $G = (T, N, S, P)$ eine Grammatik. Eine *Ableitungskette* ist eine Folge von Worten $(w_1, \ldots, w_n)$, so dass $w_1 \Rightarrow w_2$ gilt und $w_2 \Rightarrow w_3$ und $w_3 \Rightarrow w_4$ und so weiter. Wir schreiben hierfür kurz:

$$w_1 \Rightarrow w_2 \Rightarrow w_3 \Rightarrow \cdots \Rightarrow w_{n-1} \Rightarrow w_n$$

oder noch kürzer

$$w_1 \Rightarrow^* w_n.$$

Per Definition gilt auch $w \Rightarrow^* w$ (sozusagen eine Kette der Länge 0).

Bemerkungen:

- Formal ist $\Rightarrow^*$ auch eine Relation auf Worten. Sie ist der reflexive transitive Abschluss der Relation $\Rightarrow$.
- Damit $u \Rightarrow^* v$ gilt, muss es eine *endliche* Ableitungskette geben.

2-18

Beispiele von Ableitungsketten

Beispiel

Sei $G\colon S \to aSb \mid \lambda$. Dann gilt:

$$SS \Rightarrow aSbS \Rightarrow aSbaSb \Rightarrow aSbaaSbb$$

und somit $SS \Rightarrow^* aSbaaSbb$.

Beispiel

Sei $G\colon aS \to Sa$. Dann gilt:

$$aaSaaS \Rightarrow aSaaaS \Rightarrow aSaaSa$$

und somit $aaSaaS \Rightarrow^* aSaaSa$.

✎ **Zur Diskussion**

Sei $G\colon S \to 0S0 \mid 1S1 \mid 2$. Wie lautet die Ableitungskette für $SSS \Rightarrow^* 0S0121S$?

2-19

Formale Definition der erzeugten Sprache

▶ **Definition: Erzeugte Sprache**

Sei $G = (T, N, S, P)$ eine Grammatik.

- Wir sagen, ein Wort $w \in T^*$ *lässt sich mittels G ableiten* oder auch *erzeugen*, wenn $S \Rightarrow_G^* w$ gilt.
- Die Menge aller solcher Worte bildet eine Sprache, wir nennen sie die *erzeugte Sprache*.
- Wir schreiben hierfür $L(G)$. Es gilt also:

$$L(G) = \{w \in T^* \mid S \Rightarrow_G^* w\}.$$

Bemerkungen:

- Man kann sich das »L« in »$L(G)$« als einen Operator vorstellen, der Grammatiken auf die von ihnen erzeugten Sprachen abbildet. (So wie »sin« in »$\sin(x)$« einen Winkel x auf eine Streckenlänge abbildet.)
- Ein Wort in der erzeugten Sprache enthält *immer nur Terminale* und *niemals Nonterminale*.

Beispiel

Sei $G\colon S \to aSb \mid \lambda$. Dann enthält $L(G)$ alle Worte der Form »viele a's, gefolgt von derselben Anzahl an b's«. In Symbolen:

$$L(G) = \{a^n b^n \mid n \in \mathbb{N}\}.$$

Beispiel

Sei

$$G\colon S \to aS \mid B$$
$$B \to bB \mid \lambda$$

Dann enthält $L(G)$ alle Worte der Form »einige a's, gefolgt von einigen b's«. In Symbolen:

$$L(G) = \{a^n b^m \mid n,m \in \mathbb{N}\}.$$

✎ Zur Übung

2-20

Geben Sie in Worten oder in symbolischer Schreibweise an, welche Sprachen die folgenden Grammatiken erzeugen:

1. $G_1\colon S \to 0S0 \mid 1S1 \mid 0 \mid 1 \mid \lambda$.
2. $G_2\colon S \to SS \mid 0S \mid S0 \mid 0 \mid \lambda$.

Geben sie nun umgekehrt Grammatiken an, die die folgenden Sprachen erzeugen:

1. $L_1 = \{w \in \{0,1\}^* \mid w \text{ enthält mindestens eine } 0\}$.
2. $L_2 = \{w \in \{0,1\}^* \mid w \text{ enthält mindestens tausend 0en}\}$.
3. $L_3 = \{w \circ w \mid w \in \{0,1\}^*\}$ (schwer).

Grammatiken in der Praxis

2-21

Aus der Java-Dokumentation, etwas vereinfacht und in der Notation dieser Vorlesung:

> This chapter presents a grammar for the Java programming language.
>
> The grammar presented piecemeal in the preceding chapters is much better for exposition, but it is not well suited as a basis for a parser. The grammar presented in this chapter is the basis for the reference implementation. [...]

$$\begin{aligned}
\textit{Type} &\to \textit{Identifier} \mid \\
&\quad \textit{Identifier}\texttt{<}\textit{TypeArguments}\texttt{>} \mid \\
&\quad \textit{BasicType} \\
\textit{TypeArguments} &\to \textit{Type} \mid \textit{Type}\,\texttt{,}\,\textit{TypeArguments} \\
\textit{BasicType} &\to \texttt{byte} \mid \texttt{short} \mid \texttt{char} \mid \texttt{int} \mid \texttt{long} \mid \\
&\quad \texttt{float} \mid \texttt{double} \mid \texttt{boolean} \\
\textit{Identifier} &\to \ldots
\end{aligned}$$

2.2.3 Korrektheitsbeweise

Zur Erinnerung: Was macht Beweise aus?

2-22

Proofs In a Nutshell

- Ein (mathematischer) *Beweis* ist ein *Überzeugungsversuch*, bei dem ein *skeptischer Leser* von Beweisführer von der *Richtigkeit einer Behauptung* überzeugt werden soll.
- Die Ausführlichkeit hängt von der »Skeptizität« des Lesers ab:
  - Bei gutgläubigen Lesern reicht »Das ist trivial.«
  - Bei normalen Lesern muss eine lange Folge von Argumenten kommen: »Nehmen wir an, dass Foo nicht gelten würde. Dann müsste aber Bar gelten. Somit ... und deshalb ... und damit erst recht auch ...«
  - Bei ultraskeptischen Lesers muss man einen formalen Beweis führen (siehe die Vorlesung *Logik für Informatiker*).

Korrektheitsbeweise bei Grammatiken

- Es ist oft nicht leicht zu erkennen, welche Sprache eine Grammatik genau erzeugt.
- Ein *Korrektheitsbeweis für Grammatiken* ist ein Beweis dafür, dass eine Grammatik tatsächlich eine bestimmte Sprache erzeugt.

2-23

Beweisrezept: *Korrektheitsbeweis für Grammatiken*

Ziel

Man will beweisen, dass eine Grammatik G eine Sprache L erzeugt.

Rezept

Zeige zwei Richtungen (Beweisrezept »Zwei Richtungen«):

1. Beginne mit »Sei $w \in L$ beliebig.« (Beweisrezept »All-Aussagen beweisen«). Argumentiere dann, dass $S \Rightarrow_G^* w$ gilt, indem eine Ableitungskette konstruiert wird (Beweisrezepte »Konstruktiver Beweis« und häufig auch »Fallunterscheidung«). Ende mit »Also gilt $S \Rightarrow_G^* w$ und somit $L \subseteq L(G)$«.
2. Beginne mit »Sei $w \in L(G)$ beliebig.« (Beweisrezept »All-Aussagen beweisen«.) Fahre fort mit »Dann gilt $S \Rightarrow^* w$. Sei $S = w_1 \Rightarrow \cdots \Rightarrow w_n = w$ die Ableitungskette.« (Beweisrezept »Namen vergeben«.) Argumentiere dann, dass $w \in L$ gilt. Ende mit »Also gilt $w \in L$ und somit $L(G) \subseteq L$«.

2-24

Ein einfaches, ausführliches Beispiel eines Korrektheitsbeweises.

▶ **Lemma**
Sei G: $S \to aS \mid b$. Dann gilt $L(G) = \{a^n b \mid n \in \mathbb{N}\}$.

Bevor wir dies beweisen, machen wir uns erstmal klar, was überhaupt zu beweisen ist:

- Es ist eine Grammatik gegeben und es wird behauptet, dass diese eine bestimmte Sprache, nämlich $\{a^n b \mid n \in \mathbb{N}\}$, erzeugt. Nennen wir diese bestimmte Sprache einfach L.
- Es gibt nur ein Nonterminal, hier ist also wohl keine Fallunterscheidung nötig.
- Klar scheint: alle Ableitungsketten sehen wie folgt aus:

$$S \Rightarrow aS \Rightarrow aaS \Rightarrow \cdots \Rightarrow a^n S \Rightarrow a^n b$$

- Es lassen sich also tatsächlich genau die Worte aus L ableiten.
- Das müssen wir jetzt noch alles schön und korrekt aufschreiben.

Beweis. Sei $L = \{a^n b \mid n \in \mathbb{N}\}$. Wir müssen zeigen, dass $L(G) = L$ gilt. Dazu zeigen wir zwei Richtungen: $L \subseteq L(G)$ und $L(G) \subseteq L$.
Wir beginnen mit $L \subseteq L(G)$. Sei $w \in L$ beliebig. Dann gilt $w = a^n b$ für ein $n \in \mathbb{N}$. Wenn man nun auf das Startsymbol genau n-mal die Regel $S \to aS$ anwendet und dann einmal die Regel $S \to b$, so ergibt sich folgende Ableitungskette:

$$S \Rightarrow aS \Rightarrow aaS \Rightarrow \cdots \Rightarrow a^n S \Rightarrow a^n b.$$

Also gilt $S \Rightarrow^* a^n b = w$. Also gilt $w \in L(G)$ und somit $L \subseteq L(G)$.
Nun bleibt $L(G) \subseteq L$ zu zeigen. Sei dazu $w \in L(G)$ beliebig. Dann gilt $S \Rightarrow^* w$ via einer Ableitungskette

$$S = w_1 \Rightarrow \cdots \Rightarrow w_n = w.$$

Das Wort w_1 enthält ein Nonterminal (nämlich S). Bei jeder Regelanwendung kann die Anzahl der Nonterminale nicht wachsen (die rechten Regelseiten enthalten ja immer nur ein Nonterminal). Also enthalten alle w_i höchstens ein Nonterminal.
Da w_n keine Nonterminale enthält (es gilt ja $w \in T^*$ per Definition der erzeugen Sprache), muss genau im letzten Ableitungsschritt die Regel $S \to b$ angewandt worden sein. Dann muss in allen anderen Schritte die Regel $S \to aS$ angewandt worden sein. Folglich lautete die Ableitung

$$S \Rightarrow aS \Rightarrow aaS \Rightarrow \cdots \Rightarrow a^n S \Rightarrow a^n b.$$

Also gilt $w = a^n b$ und somit $w \in L$ und somit $L(G) \subseteq L$. □

Ein schwierigerer Korrektheitsbeweis, kurz aufgeschrieben. 2-25

▶ Lemma
Sei $G\colon S \to 0S0 \mid 1S1 \mid \lambda$. *Sei* $L = \{w \in \{0,1\}^* \mid w = w^{\text{rev}}, |w| \bmod 2 = 0\}$. *Dann gilt* $L(G) = L$.

Beweis. Für die Richtung $L \subseteq L(G)$ sei $w \in L$ beliebig. Dann gilt $w = u \circ u^{\text{rev}}$ für ein $u \in \{0,1\}^*$. Bilde folgende Ableitungskette: Ausgehend von S wende im i-ten Schritt die Regel $S \to 0S0$ an, falls $u[i] = 0$ und sonst $S \to 1S1$. Nach $|u|$ Schritten wende dann einmalig die Regel $S \to \lambda$ an. Wir erhalten folgende Ableitungskette:

$$\begin{aligned} S &\Rightarrow u[1]Su[1] \Rightarrow u[1]u[2]Su[2]u[1] \\ &\Rightarrow u[1]u[2]u[3]Su[3]u[2]u[1] \Rightarrow \cdots \Rightarrow uSu^{\text{rev}} \Rightarrow uu^{\text{rev}}. \end{aligned}$$

Also gilt $S \Rightarrow^* uu^{\text{rev}} = w$ und somit $w \in L(G)$.
Sei nun $w \in L(G)$. Dann gilt $S \Rightarrow^* w$ via einer Ableitungskette beginnend mit $w_1 = S$ und endend mit $w_n = w$. Da das Nonterminal S nur mit der Regel $S \to \lambda$ gelöscht werden kann, muss diese Regel genau einmal, nämlich am Ende angewandt worden sein. Alle anderen Ableitungschritte müssen also die Regel $S \to 0S0$ oder $S \to 1S1$ angewandt haben.
Man sieht nun leicht (oder zeigt dies durch Induktion), dass sich jedes w_i für $i \in \{1, \ldots, n-1\}$ darstellen lässt als $w_i = u_i S u_i^{\text{rev}}$ mit $u_i \in \{0,1\}^*$. Da im letzten Ableitungsschritt die Regel $S \to \lambda$ verwendet wurde, gilt $w_n = u_{n-1} \circ u_{n-1}^{\text{rev}}$.
Also gilt $w = w_n = u_{n-1} \circ u_{n-1}^{\text{rev}} \in L$ und somit $L(G) \subseteq L$. □

Zusammenfassung dieses Kapitels

1. (Formale) Grammatiken bestehen aus *Ersetzungsregeln*. 2-26
2. Sie geben an, dass in einer *Ableitung* aus einem Wort eine linke Regelseite gestrichen werden und durch die rechte Seite ersetzt werden darf.
3. Alle nur aus Terminalen bestehenden Worte, die sich aus dem Startsymbol ableiten lassen, bilden die *erzeugte Sprache*.
4. In einem *Korrektheitsbeweis für eine Grammatik* beweist man, dass eine Grammatik eine bestimmte Sprache erzeugt.

Zum Weiterlesen

[1] R. Reischuk. *Unterlagen zum Lehrmodul Theoretische Informatik,* Vorlesungsskript, 2008. Anfang von Kapitel 1.3

Übungen zu diesem Kapitel

Übung 2.1 Ableitungen angeben, leicht
Gegeben sei die folgende Grammatik über dem Alphabet $\Sigma = \{0,1\}$:

$$\begin{aligned} G\colon S &\to A00A \\ A &\to 0A \mid 1A \mid \lambda \end{aligned}$$

Geben Sie alle möglichen Ableitungen für das Wort 10001 in G an.

Übung 2.2 Formalen Korrektheitsbeweis führen, mittel
Welche Sprache über $\Sigma = \{0,1\}$ erzeugt die Grammatik aus Aufgabe 2.1? Geben Sie einen formalen Beweis für Ihre Behauptung an.

Übung 2.3 Sprachen aus Grammatiken erkennen, mittel

Geben Sie eine Ableitungskette für das Wort *aaabbbababb* an. Welche Sprache über dem Alphabet $\Sigma = \{a, b\}$ erzeugt die folgende Grammatik? Beweisen Sie Ihre Annahme.

$$\begin{aligned} G\colon\ & S \to aS \mid bS \mid aA \\ & A \to bB \\ & B \to aB \mid bB \mid a \mid b \end{aligned}$$

Übung 2.4 Grammatiken angeben, mittel

Ein Wort $w \in \Sigma^*$ heißt Teilwort eines Wortes $u \in \Sigma^*$, falls Worte $\alpha, \gamma \in \Sigma^*$ existieren mit $\alpha w \gamma = u$. Geben Sie Grammatiken für folgende Sprachen über $\Sigma = \{0, 1\}$ an.

1. $A = \{w \in \Sigma^* \mid |w| \text{ ist ungerade}\}$.
2. $B = \{w \in \Sigma^* \mid w \text{ enthält } 00 \text{ nicht als Teilwort}\}$.
3. $C = \{w \in \Sigma^* \mid w \text{ enthält eine ungerade Anzahl von Nullen}\}$.
4. $D = \{w \in \Sigma^* \mid w \text{ enthält genau drei Einsen}\}$.

3-1

Kapitel 3
Die Chomsky-Hierarchie

Reguläre Grammatiken sind auch kontextfreie Grammatiken

Lernziele dieses Kapitels

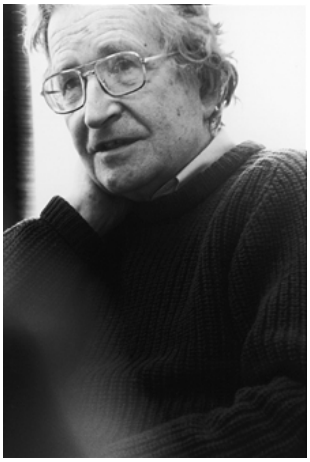
1. Stufen der Hierarchie kennen
2. Grammatiken der richtigen Stufe zuordnen können
3. Sprachklassen und Abgeschlossenheit verstehen

3-2

Inhalte dieses Kapitels

Worum es heute geht

Die Chomsky-Hierarchie geht, wenig überraschend, auf Noam Chomsky zurück. Sie stellt den Versuch dar, Grammatiken danach einzuteilen, wie »schwierig« sie sind. Eine solche Einteilung ist nicht so leicht wie man denken könnte, da es erstmal reichlich unklar ist, was genau eine Grammatik »schwierig« macht. Man könnte zum Beispiel erstmal ganz naiv annehmen, dass die *Anzahl der Regeln* einer Grammatik hier doch ganz entscheidend sein sollte. Dies ist aber nicht der Fall. Eine Grammatik kann gerne eine Million Regeln haben – wenn diese alle sehr ähnlich sind, dann kann die Grammatik trotzdem sehr »einfach« sein. Ein anderes denkbares Maß ist die *Länge der Regeln*. Hier stellt sich überraschenderweise heraus, dass besonders lange Regeln mit vielen Terminalen sich eher günstig auf die Geschwindigkeit auswirken, mit der das Wortproblem für die Grammatik entschieden werden kann.



Public domain

Chomsky erkannte, dass das entscheidende Maß für die Komplexität einer Grammatik die *Anzahl und Position der Nonterminale* auf der linken und rechten Regelseite ist. Er hat dann vier Typen von Grammatiken eingeführt und diese, wenig originell, mit Typ 0 bis Typ 3 bezeichnet. Dabei sind Typ-0-Grammatiken die allgemeinsten, Typ-3-Grammatiken die speziellsten. Als Chomsky seine Grammatik-Typen einführte, konnte er noch nicht ahnen, welche davon in der Praxis besonders wichtig werden würden. Heutzutage werden Typ-3- und Typ-2-Grammatiken sehr viel benutzt sowohl in der Theorie wie in der Praxis; Typ-1- und Typ-0-Grammatik sind hingegen höchstens von theoretischem Interesse (also für diese Vorlesung genau richtig).

3.1 Grammatik-Arten

3-4

Was macht Grammatiken schwierig?

✎ Zur Diskussion
Welche Sprachen erzeugen die folgenden Grammatiken?

$$G_1\colon \begin{aligned} S &\to 0A \\ A &\to 0B \\ B &\to 0C \\ C &\to 0D \\ D &\to 0E \\ E &\to S \mid \lambda \end{aligned} \qquad G_2\colon \begin{aligned} S &\to S00100S \mid A \\ 0A &\to A0 \\ 1A &\to A01 \\ SA0001 &\to A000 \mid 000 \end{aligned}$$

Beobachtungen

- Manche Grammatiken sind einfacher zu »verstehen« als andere.
- Dabei ist nicht die *Anzahl der Regeln* wichtig, sondern die *Art der Regeln*.
- Besonders Regeln mit *vielen Nonterminalen links und rechts* machen Grammatiken »schwer zu verstehen«.
- Je komplizierter die Regeln in einer Grammatiken sind, desto schwieriger ist es im Allgmeinen, Worte mittels dieser Grammatik zu *parsen*.

Unser heutiges Ziel: Grammatiken anhand der *Komplexität ihrer Regeln zu klassifizieren*.

3.1.1 Regulär

3-5

Die einfachsten Grammatiken: Reguläre Grammatiken.

- Reguläre Grammatiken gehören zu den einfachsten Grammatiken.
- Die Regeln sind besonders einfach und gleichförmig – eben *regulär*.

▶ Definition: Reguläre Grammatik
Eine Grammatik $G = (N, T, S, P)$ heißt *regulär*, wenn alle Regeln $l \to r$ in P folgende Eigenschaften haben:

1. l ist ein einzelnes Nonterminal, also $l \in N$.
2. r enthält höchstens ein Nonterminal und wenn, dann am Ende, also $r \in T^* \cup (T^* \circ N)$.

Beispiel

$$G\colon \begin{aligned} S &\to 0S \mid 00B \\ B &\to 1B \mid 11 \end{aligned}$$

3-6

Was regulären Grammatiken leisten können.

- In jedem Ableitungsschritt, bis auf den letzten, steht am Ende ein Nonterminal – und dies ist auch immer das einzige Nonterminal im ganzen Wort.
- Mit einer regulären Grammatik werden Worte also immer »von vorne nach hinten« abgeleitet.
- Wie es während einer Ableitung weitergeht, *hängt immer nur vom aktuellen Nonterminal ab*.

3-7

✎ Zur Übung
Geben Sie reguläre Grammatiken an, die die folgenden Sprachen erzeugen:

$$L_1 = \{w \in \{0,1\}^* \mid w \text{ enthält eine 0 und auch eine 1}\},$$
$$L_2 = \{w \in \text{ASCII}^* \mid w \text{ ist ein erlaubter Java-Bezeichner}\}.$$

Reguläre Grammatiken in der Praxis.

3-8

- Für strukturierte Sprachen wie HTML gibt es keine reguläre Grammatiken.
- Man kann aber *Teile* doch mit regulären Grammatiken beschreiben.
- Hier beispielsweise eine Grammatik, die *erlaubte (öffnende) div-Tags beschreibt mit optionalem class-Argument*.

$$
\begin{aligned}
G_{\text{div-tag}}\colon S &\to \texttt{<}A\\
A &\to \texttt{␣}A \mid B\\
B &\to \texttt{div}C\\
C &\to \texttt{␣}C \mid \texttt{␣class="}D \mid F\\
D &\to \texttt{a}D \mid \cdots \mid \texttt{z}D \mid \texttt{A}D \mid \cdots \mid \texttt{Z}D \mid E\\
E &\to \texttt{"}C\\
F &\to G \mid \texttt{/}G\\
G &\to \texttt{␣}G \mid \texttt{>}
\end{aligned}
$$

3.1.2 Kontextfrei

Grammatiken, die Hierarchien beschreiben.

3-9

- Kontextfreie Grammatiken beschreiben die *hierarchische Struktur* von Text.
- Nonterminale entsprechen »Konzepten«, die sich immer wieder rekursiv ersetzen lassen.
- Reguläre Grammatiken sind ein *Spezialfall* von kontextfreien Grammatiken.

▶ Definition: Kontextfreie Grammatik

Eine Grammatik $G = (N, T, S, P)$ heißt *kontextfrei*, wenn alle Regeln $l \to r$ in P folgende Eigenschaften haben:

- l ist ein einzelnes Nonterminal, also $l \in N$.

Beispiel

$G\colon S \to SS \mid 0S1 \mid 1S0 \mid \lambda$

✎ Zur Übung

3-10

Geben Sie eine kontextfreie Grammatiken an, die die folgenden Sprache erzeugt:

$$L = \{0^n 1^m \mid n \leq m\}.$$

Wem das zu leicht ist, der probiert sich bitte an $\{0^n 1^m 2^k \mid n \leq m \leq k\}$.

3.1.3 Kontextsensitiv

3-11

Grammatiken, die »mit Worten arbeiten«.

Kontextsensitive Grammatiken erlauben es, während einer Ableitung das erzeugte Wort »nachträglich zu ändern«:

▶ Definition: Kontextsensitive Grammatik, vorläufige Definition

Eine Grammatik $G = (N, T, S, P)$ heißt *kontextsensitiv*, wenn alle Regeln $l \to r$ in P folgende Eigenschaften haben:

- $l = uXv$ mit $X \in N$ und $u, v \in (N \cup T)^*$,
- $r = uwv$ mit $w \in (N \cup T)^*$ und $|w| \geq 1$.

Ein Beispiel:

$$\begin{aligned} G\colon \quad S &\to ABS \mid \lambda \\ BS &\to CS \\ AC &\to BC \\ BC &\to BA \\ A &\to 0 \\ B &\to 1 \end{aligned}$$

3-12

Die λ-Startregel.

- Um das leere Wort λ mit einer Grammatik zu erzeugen, braucht man Regeln, bei denen die rechte Seite kürzer ist als die linke. Das wichtigstes Beispiel ist »$X \to \lambda$«.
- Solche Regeln sind Theoretikern manchmal etwas peinlich.
- Wir führen deshalb eine besondere Sprechweise genau für diesen Fall ein – dann muss uns das λ auch nicht mehr peinlich sein.

▶ Definition: Die λ-Startregel

Eine Grammatik mit Startsymbol S *hat eine* λ*-Startregel*, wenn

1. sie die Regeln $S \to \lambda \mid S'$ enthält und
2. S nicht auf der rechten Seite irgendeiner Regel vorkommt.

Merke: Mit der λ-Startregel kann man entweder sofort λ erzeugen und Schluss – oder man kann nurnoch Regeln anwenden, in denen λ nicht mehr vorkommt.

3-13

Kontextsensitive Grammatiken -- die endgültige Definition.

▶ Definition: Kontextsensitive Grammatik

Eine Grammatik $G = (N, T, S, P)$ heißt *kontextsensitiv*, wenn alle Regeln $l \to r$ in P folgende Eigenschaften haben:

- $l = uXv$ mit $X \in N$ und $u, v \in (N \cup T)^*$,
- $r = uwv$ mit $w \in (N \cup T)^*$ und $|w| \geq 1$.

Zusätzlich darf die Grammatik eine λ-Startregel haben.

3.2 Die Chomsky-Hierarchie

3.2.1 Sprachklassen

Sprachklassen sind Klassen von Sprachen

3-14

Sprachklassen dienen dazu, alle Sprachen mit einer bestimmten Eigenschaft zusammenzufassen:

▶ Definition: Sprachklasse
Eine *Sprachklasse* ist eine Menge von Sprachen.

Beispiel: Endliche Sprachen
Die Sprachklasse FINITE enthält alle endlichen Sprachen (also alle Sprachen, die nur endlich viele Worte enthalten).

3.2.2 Stufen der Hierarchie

Jede Grammatik-Art erzeugt eine Sprachklasse.

3-15

- Wir haben Grammatiken danach eingeteilt, wie »komplex« ihre Regeln sind.
- Komplexere Grammatiken werden dann (vermutlich) auch komplexere Sprachen erzeugen.
- Um dies zu untersuchen definieren wir die *Klassen von Sprachen*, die sich *mit verschiedenen Grammatiken erzeugen lassen*.

▶ Definition: Spracheklassen der Chomsky-Hierarchie
Die Klasse REG enthält alle Sprachen, die sich von regulären Grammatiken erzeugen lassen.
(Formal: $\text{REG} = \{L(G) \mid G \text{ ist regulär}\}$).
Analog definiert man:

Klasse	Enthält alle Sprachen, die erzeugt werden von…
CFL	kontextfreien Grammatiken
CSL	kontextsensitiven Grammatiken
RE	allgemeinen Grammatiken

Übersicht der Klassen und alternativer Bezeichnungen.

3-16

Klasse	Bezeichnungen
REG	reguläre Sprachen, Typ-3-Sprachen
CFL	kontextfreie Sprachen, Typ-2-Sprachen
CSL	kontextsensitive Sprachen, Typ-1-Sprachen
RE	rekursiv aufzählbare Sprachen, semientscheidbare Sprachen, Typ-0-Sprachen

Im Laufe des Semesters werden wir noch zeigen:

$$\text{REG} \subsetneq \text{CFL} \subsetneq \text{CSL} \subsetneq \text{RE}$$

3.2.3 Abgeschlossenheit

Abgeschlossenheit von Sprachklassen

3-17

- Wir haben einige Operationen kennengelernt, die Sprachen verändern (wie Schnitt oder Kleene-Stern).
- Da Sprachklassen ja ähnliche Probleme zusammenfassen sollen, sollten einfache Veränderungen einer Sprache wieder Sprachen der gleichen Klasse ergeben.

▶ Definition: Abgeschlossenheit
Eine Sprachklasse C heißt *abgeschlossen gegenüber einer Operation auf Sprachen*, wenn die Anwendung der Operation auf Sprachen aus der Klasse nur Sprachen aus der Klasse ergibt.

3-18

Abgeschlossenheit der Klasse FINITE.

Beispiel

Die Klasse der endlichen Sprachen ist abgeschlossen unter

- Vereinigung (die Vereinigung endlicher Mengen ist endlich) und
- Schnitt (ihr Schnitt erst recht)

und nicht abgeschlossen unter

- dem Kleene-Stern (die Sprache $\{\mathtt{a}\}^*$ ist nicht endlich, obwohl $\{\mathtt{a}\}$ ja endlich ist) oder
- dem Komplement (das ist ja unendlich).

3-19

Beweisrezept: *Abgeschlossenheit von Grammatik-Sprachklassen*

Ziel

Man will beweisen, dass eine Sprachklasse C unter einer Operation $\otimes$ abgeschlossen ist.

Rezept

1. Beginne mit »Seien $L_1, L_2 \in C$. Dann gibt es Grammatiken G_1 und G_2 für L_1 und L_2.«
2. Konstruiere dann eine Grammatik G für $L_1 \otimes L_2$ (Beweisrezept »Konstruktiver Beweis«).
3. Zeige, dass alle Regeln in G vom richtigen Typ sind.
4. Zeige, dass $L(G) = L_1 \otimes L_2$ gilt (Beweisrezept »Korrektheit von Grammatiken«).

3-20

Abgeschlossenheit unter Vereinigung.

▶ **Satz**

Alle Klassen der Chomsky-Hierarchie sind unter Vereinigung abgeschlossen.

Beweis. Seinen $G_1 = (T_1, N_1, S_1, P_1)$ und $G_2 = (T_2, N_2, S_2, P_2)$ Grammatiken vom gleichen Typ. Falls nötig, benennen wir die Nonterminale um, so dass $T_1 \cup T_2$ und N_1 und N_2 jeweils paarweise disjunkt sind. Sei S ein neues Startsymbol.
Betrachte folgende Grammatik: $G = (T_1 \cup T_2, N_1 \cup N_2 \cup \{S\}, S, P_1 \cup P_2 \cup \{S \to S_1, S \to S_2\})$.
Wir zeigen, dass für diese Grammatik gilt:

1. Sie ist vom selben Typ wie G_1 und G_2.
2. $L(G) = L(G_1) \cup L(G_2)$.

Für die erste Behauptung beachte man, dass die Produktionsregeln von G ja einfach von G_1 und G_2 übernommen wurden und somit auch deren Typ haben. Die neuen Regeln $S \to S_1 \mid S_2$ sind ebenfalls bei allen Grammatikarten erlaubt.
Für die zweite Behauptung muss man zwei Richtungen zeigen. Für die Richtung $L(G) \subseteq L(G_1) \cup L(G_2)$, sei $w \in L(G)$. Dann existiert eine Ableitung $S \Rightarrow_G^* w$. Diese muss im ersten Schritt die Regel $S \to S_1$ oder die Regel $S \to S_2$ benutzt haben. Im ersten Fall können danach nurnoch Regeln aus G_1 verwendet worden sein, denn man kann durch Induktion leicht zeigen, dass alle weiteren Worte der Ableitung nur Nonterminale aus N_1 enthalten – folglich gilt $S_1 \Rightarrow_{G_1}^* w$. Analog gilt im zweiten Fall $S_2 \Rightarrow_{G_2}^* w$.
Für die Richtung $L(G_1) \cup L(G_2) \subseteq L(G)$, sei $w \in L(G_1)$ – der Fall $w \in L(G_2)$ geht analog. Dann existiert eine Ableitung $S_1 \Rightarrow_{G_1}^* w$. Da $S \Rightarrow_G S_1$ und die Regeln von G_1 alle in G enthalten sind, gilt auch $S \Rightarrow_G S_1 \Rightarrow_G^* w$. Also gilt $w \in L(G)$, was zu beweisen war. □

Abgeschlossenheit unter anderen Operationen.

3-21

- Etwas, aber nicht viel, schwieriger ist es zu zeigen, dass die Sprachklassen auch alle unter Verkettung abgeschlossen sind.
- Darauf aufbauend ist es auch nicht schwierig, den Abschluss unter dem Kleene-Stern zu zeigen.
- Der Abschluss unter Schnitt ist schwieriger zu zeigen und gar nicht immer der Fall.
- Unter Komplement sind schließlich überhaupt nur die regulären und die kontextsensitiven Sprachen abgeschlossen und dies ist bei letzteren extrem schwer zu beweisen.

Operation	REG	CFL	CSL	RE
Vereinigung	✓	✓	✓	✓
Verkettung	✓	✓	✓	✓
Stern	✓	✓	✓	✓
Schnitt	✓	✗	✓	✓
Komplement	✓	✗	✓	✗

Abgeschlossenheit in der Praxis

3-22

Wiederholung: Datums-Validierung

Wir hatten schon gesehen, dass man die *Datumsvalidierung* durch viele Einzeltests *modularisieren* kann:

- Test, ob die Eingabe nur aus Ziffern und Punkten besteht.
- Test, ob es genau zwei Punkte gibt.
- Test, ob …

Die Abgeschlossenheit der Klasse der regulären Sprachen unter verschiedenen Operationen hilft nun:

- Alle Tests lassen sich recht leicht mittels regulärer Grammatiken beschreiben.
- Da Schnitt und Vereinigung von regulären Sprachen wieder regulär sind, wissen wir, dass es auch eine reguläre Grammatik für die Datumsvalidierung geben muss.
- Diese lässt sich, falls gewünscht, sogar durch ein Programm berechnen.

3.2.4 Einordnung typischer Probleme

Welche Sprache gehört auf welche Stufe?

3-23

Es gibt ein paar grobe Regeln, wohin Sprachen typischerweise gehören:

Typ 3 Einfache Sprachen, deren Worte »aus mehrere, sich wiederholenenden Teilen« aufgebaut sind. Nicht möglich ist »Zählen« der Art: »So viele Wiederholungen hiervon wie woanders davon.«

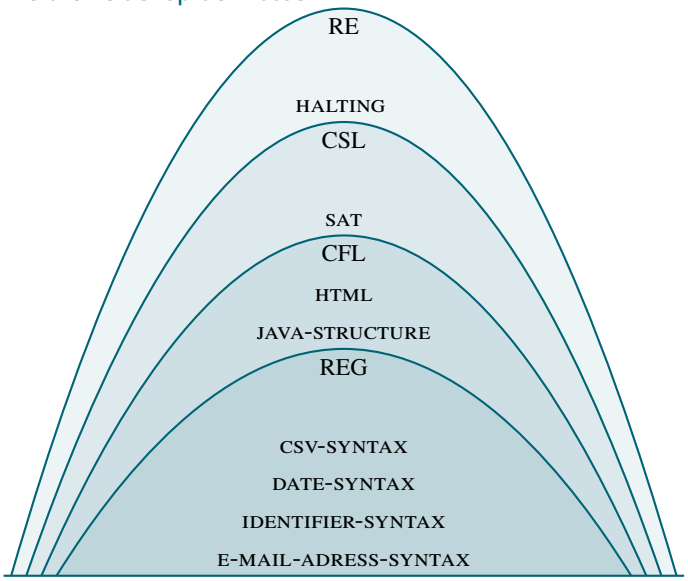
Typ 2 Strukturierte Sprachen, deren Worte hierarchisch aufgebaut sind. Nicht möglich sind Abhängigkeiten/Verweise der Art: »Was hier steht, muss zu etwas passen, das auf einer anderen Hierarchieebene steht.«

Typ 1 Viele Sprachen aus der Praxis, trotzdem eher geringe praktische Bedeutung.

Typ 0 Alle mit Computern verarbeitbare Sprachen.

3-24

Hierarchie der Sprachklassen



Zusammenfassung dieses Kapitels

3-25

1. Die Chomsky-Hierarchie hat vier Stufen: reguläre Sprache, kontextfreie Sprachen, kontextsensitive Sprachen und allgemeine Sprachen.
2. Einfache Syntax-Checks lassen sich mit regulären Grammatiken beschreiben.
3. Hierarchische Strukturen lassen sich mit kontextfreien Grammatiken beschreiben.
4. Sprachklassen sind *abgeschlossen* unter Operationen auf Sprachen, wenn man »immer in der Klasse bleibt«.

Zum Weiterlesen

[1] R. Reischuk. *Unterlagen zum Lehrmodul Theoretische Informatik,* Vorlesungsskript, 2008. Kapitel 1.3.

Übungen zu diesem Kapitel

Übung 3.1 Regel-Art angeben, leicht

Geben Sie in einer Tabelle für jede der folgenden Regeln an, ob sie jeweils bei (a) regulären, (b) kontextfreien und/oder (c) kontextsensitiven Grammatik erlaubt oder verboten ist.

1. $S \to S$
2. $A \to \lambda$
3. $\lambda \to A$
4. $X \to 00X$
5. $X \to X00$
6. $X \to X0X$
7. $AB \to AB$
8. $AB \to ABBA$
9. $AB \to BA$

Übung 3.2 Grammatiken entwerfen für ein praktisches Problem, mittel

Cascading Style Sheets (css) sind eine Methode zur Beschreibung des Aussehens von Webseiten. Da der Standard recht komplex ist, sollen in dieser Aufgabe folgende Vereinfachungen gemacht werden: Kommentare werden ignoriert, Folgen von Leerzeichen und Zeilenumbrüchen ebenso, besondere Selektoren auch. Geben Sie eine kontextfreie Grammatik für syntaktisch korrekte css-Dateien an. Es reicht, wenn jeweils nur exemplarisch einige mögliche Attribute anzugeben.

Tipp: Benutzen Sie bitte für Nonterminal nicht nur einzelne Symbole wie A, B oder C, sondern aussagekräftige Bezeichner wie *Maßeinheit* oder *Selektor* oder *Zahl*.

Übung 3.3 Äquivalenz von Grammatiken, schwer bonus

Wir nennen eine Grammatik G *monoton*, falls für jede Regel in G gilt, dass die linke Seite der Regel höchstens so lang ist wie die rechte Seite der Regel. Zeigen Sie, dass zu jeder monotonen Grammatik G eine kontextsensitive Grammatik G' existiert, so dass $L(G) = L(G')$.

Übung 3.4 Abgeschlossenheit unter Verkettung I, mittel, mit Lösung

Zeigen Sie, dass die Klasse CFL unter Verkettung abgeschlossen ist.

Tipp: Orientieren Sie sich am Beweis auf Folie 3-20.

Übung 3.5 Abgeschlossenheit unter Verkettung II, mittel

Zeigen Sie, dass die Klasse REG unter Verkettung abgeschlossen ist.

Tipp: Eine Regel der Art $S \to S_1 S_2$ ist nicht erlaubt. Sie müssen deshalb die *terminierenden Regeln* der ersten Grammatik verändern, indem Sie dort das Startsymbol der zweiten Grammatik anfügen.

Übung 3.6 Abgeschlossenheit unter Kleene-Stern I, mittel

Zeigen Sie, dass die Klasse CFL unter dem Kleene-Stern abgeschlossen ist.

Tipp: Dies geht sehr ähnlich wie Übung 3.4.

Übung 3.7 Abgeschlossenheit unter Kleene-Stern II, schwer

Zeigen Sie, dass die Klasse REG unter den Kleene-Stern abgeschlossen ist.

Tipp: Erweitern Sie die Idee von Übung 3.5 um eine »Schleife«.

Teil II

Wie analysiert man Syntax?

Die Syntax-Analyse gehört zum Brot-und-Butter-Geschäft eines modernen Computers. Ununterbrochen müssen irgendwelche Texte analysiert werden – selbst dann, wenn man dies gar nicht vermutet. Starten Sie doch einfach mal den Web-Browser Ihres Vertrauens. Noch bevor sich der Browser mit der recht komplexen Syntax-Analyse des Quelltextes Ihrer Startseite beschäftigt, hat er zunächst erstmal seine Konfigurationsdatei gelesen und eben analysiert. Weiter wurden auch die Antworten auf Anfrage über das HTTP-Protokoll syntaktisch analysiert. Je länger und genauer man die Arbeitsweise eines Rechners anschaut, desto mehr Syntax-Analyse wird man entdecken.

Allen Tätigkeiten, die Computer sehr häufig ausüben, gilt natürlich das besondere Interesse sowohl der Praxis wie auch der Theorie. So sind beispielsweise Suchen und Sortieren sowohl praktisch wie theoretisch bestens untersucht, ebenso die Anzeige von 3D-Graphiken und eben auch die Syntax-Analyse. Die Theorie der Syntax-Analyse ist ein weites Feld, wir werden uns lediglich die zwei wichtigsten Themen genauer anschauen: die Analyse von regulären Sprachen und die von kontextfreien Sprachen. Zwar kann man damit schon »recht viel« effizient analysieren, jedoch sollten Sie, auch nachdem Sie diesen Teil des Skript hingebungsvoll durchgearbeitet haben, sich nicht unbedingt gleich daran machen, einen neuen C++-Compiler zu programmieren.

4-1

Kapitel 4
Deterministische endliche Automaten

Die einfachsten Computer

Lernziele dieses Kapitels

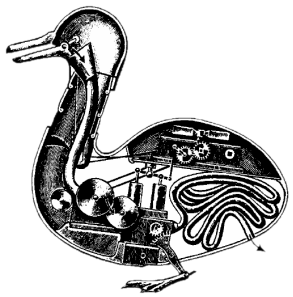
1. Syntax und Semantik von DFAs beherrschen
2. XML-Fragmente mittels DFAs beschreiben können
3. DFAs implementieren können

4-2

Inhalte dieses Kapitels

Worum es heute geht

Die Bezeichnung »endlicher Automat« ist vielleicht etwas irreführend. Die Theorie endlicher Automaten beschäftigt sich mit allem Möglichen, aber gerade nicht mit den Geräten, die man normalerweise als Automaten bezeichnet. Es wird in diesem Kapitel *nicht* darum gehen, Cola-Automaten zu programmieren oder auch nur zu modellieren. Ebensowenig geht es um die skurrilen Geräte, die in vergangenen Jahrhunderten als Automaten (wörtlich »Selbst-Bewegtes«) gefeiert wurden wie die im Jahr 1738 von Vaucanson vorgestellte mechanische Ente. Sie konnte sich laut Wikipedia »watschelnd fortbewegen, aber auch fressen, verdauen und ausscheiden«.



Public domain

Ebenso schön sind die Automaten von Jaquet-Droz, von denen Wikipedia zu berichten weiß: »Der Schreiber ist 70cm hoch, hat eine Gänsefeder in der Hand, sitzt vor einem kleinen Tisch und hat bewegliche Augen und Kopf. Er kann jeden beliebigen Text mit bis zu 40 Buchstaben Länge schreiben. Der Text wird auf einem Rad kodiert, wo die Buchstaben dann einer nach dem anderen abgearbeitet werden. Wenn er gestartet wird, taucht er zunächst die Feder in die Tinte und schüttelt sie leicht ab, dann schreibt er, wobei er wie ein echter Schreiber die Auf- und Abwärtsstriche richtig beachtet und auch absetzt. Er kann mehrzeilig schreiben und beachtet Leerzeichen.«

Unknown author, Creative Commons Attribution Sharealike License

Die endlichen Automaten der Theoretischen Informatik sind ein Formalismus, der die Konzepte *Zustand* und *Zustandsänderungen* allgemein modelliert. Da wir *endliche* Automaten untersuchen werden, wird die Menge dieser möglichen Zustände stets endlich sein. Zustände versteht man vielleicht am besten anhand einer Analogie zur Biologie: Bei einer einfachen Zelle könnte der Zustand beispielsweise beschreiben, welche Proteine sich gerade in der Zelle befinden und welche Gene aktiviert sind. Durch Umwelteinflüsse und durch innere Prozesse wird sich der Zustand der Zelle langsam oder abrupt ändern, dies werden wir *Zustandsänderungen* nennen. Ziel der Automatentheorie ist nun, herauszufinden, welche Folge von äußeren Einflüssen welche Effekte in einem Automaten / einer Zelle haben wird.

Die Analogie zu Zellen ist übrigens nicht von mir für die Vorlesung herbeikonstruiert, vielmehr beschäftigt sich eine ganze Teildisziplin der Automatentheorie, nämlich die *Theorie der zellulären Automaten*, damit, wie sich ganze Felder solcher Automaten verhalten.

4.1 Motivation von endlichen Automaten

4.1.1 Motivation I: Vom Beschreiben zum Parsen

4-4

Grammatiken sind gut, Parser sind besser.

- Mit Grammatiken können wir Problem (=Sprachen) *beschreiben*. Das wird *in der Praxis auch viel getan*.
- Grammatiken beschreiben *top-down*, was *alles in einer Sprache erlaubt ist*.
- In der *Praxis* hat man jedoch eher folgendes Problem: Gegeben ist ein *konkretes Wort*, für das man *eine Ableitung finden möchte*.
- Findet man eine Ableitung, dann weiß man
  1. dass das Wort in der Sprache liegt und
  2. durch die Schritte in der Ableitung auch, wie es aufgebaut ist.
- Ein *Wort zu parsen* bedeutet, eine Ableitung bezüglich einer bestimmten Grammatik zu finden.

Erste Motivation für endliche Automaten

Endliche Automaten werden die Maschinen sein, die uns erlauben, Worte zu parsen bezüglich *regulärer Grammatiken*.

4.1.2 Motivation II: Einfache Computer

4-5

»Wat is en Rechnmaschin? Da stelle'mer uns mal janz dumm. . . «

- Es ist ausgesprochen schwierig, allgemein zu definieren, was »ein Computer« ist.
- Es kommt darauf an, was man von einem Computer erwartet.
- Mindestens aber sollte folgendes möglich sein:
  1. Irgendwelche *Eingaben* gehen in die Maschine hinein.
  2. Irgendetwas *passiert in* der Maschine.
  3. Irgendeine *Reaktion* lässt sich beobachten.

4-6

Endliche Automaten: Die einfachsten Computer

Bei *endlichen Automaten* werden die Grundanforderungen an Computer in einfachster Weise erfüllt:

Eingaben Die einzelnen Zeichen eines Wortes dienen als Eingabe.

Verarbeitung Die Eingabezeichen werden nacheinander verarbeitet. Dabei »merkt« sich der Automat nur einen so genannten *Zustand*.

Ausgabe Der Automat *akzeptiert* oder *verwirft* die Eingabe ganz am Ende.

Merke

Endliche Automaten sind *kein* Modell realer Rechner.

4-7

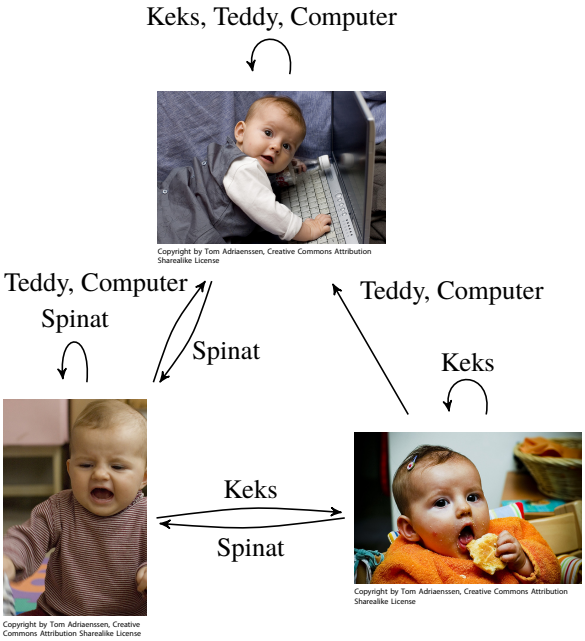
Die Zustände eines Kleinkinds

- Der Automat verfügt über drei Zustände: glücklich, schreien, essen.
- Am Anfang ist der Automat im Zustand glücklich.
- Mögliche Eingaben sind: Keks, Spinat, Computer, Teddybär.

Die Effekte der Eingabe sind folgende:

Alt-Zustand	Keks	Spinat	Teddybär/Computer
glücklich	glücklich	schreien	glücklich
schreien	essen	schreien	glücklich
essen	essen	schreien	glücklich

Der Kleinkind-Automat anders aufgemalt. 4-8



4.2 Endliche Automaten

4.2.1 Syntax

Formale Definition von deterministischen endlichen Automaten. 4-9

▶ Definition: Syntax von DFAs

Ein *deterministischer endlicher Automat M* besteht aus

1. einem *Eingabealphabet* Σ,
2. einer *endlichen Menge* Q *von Zuständen*,
3. einem *Startzustand* $q_0 \in Q$,
4. einer Menge $Q_a \subseteq Q$ von *akzeptierenden Zuständen* und
5. einer *Zustandsübergangsfunktion*

$$\delta : Q \times \Sigma \to Q.$$

Wie immer gibt es auch alternative Schreibweisen:

hier	andere Schreibweisen in der Literatur
q_0	q_A, q_1
Q_a	A, F (finale Zustände), Q_F, Q_A
δ	Δ

Beispiel eines endlichen Automaten. 4-10

Beispiel

Sei $M = (\Sigma, Q, q_0, Q_a, \delta)$ mit:

- $\Sigma = \{0,1\}$,
- $Q = \{q_a, q_b, q_c\}$,
- $q_0 = q_a$,
- $Q_a = \{q_b, q_c\}$,
- und δ wie folgt:

$$\begin{aligned}
\delta(q_a,0) &= q_a,\\
\delta(q_a,1) &= q_b,\\
\delta(q_b,0) &= q_a,\\
\delta(q_b,1) &= q_c,\\
\delta(q_c,0) &= q_c,\\
\delta(q_c,1) &= q_c.
\end{aligned}$$

4-11

Wie man Automaten aufschreibt.

- Genau wie bei Grammatiken ist es *mühselig* und *wenig anschaulich*, Automaten *ganz formal als Tupel aufzuschreiben*.
- Deshalb ist es üblich, sie als *Graphen* aufzuschreiben.

▶ Definition: Graphdarstellung eines Automaten

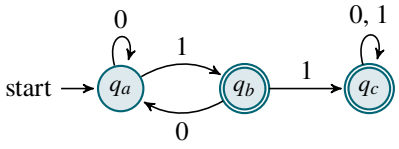
Sei $M = (\Sigma, Q, q_0, Q_a, \delta)$ ein DFA. Dann ist G_M folgender Graph:

- Die *Knoten* des Graphen sind die Zustände in Q.
- Es gibt eine *Kante* von einem Zustand q zu einem Zustand q' mit Label a, falls $\delta(q,a) = q'$.
- Jeder akzeptierende Zustand $q \in Q_a$ wird angedeutet durch einen Doppelkreis um den Zustand.
- Der Startzustand wird durch einen kleinen Pfeil mit dem Wort »start« angedeutet.

4-12

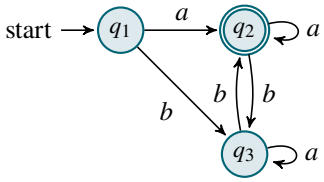
Beispiel eines endlichen Automaten als Graph.

Beispiel



✎ Zur Übung

Geben Sie folgenden Automaten formal als Fünftupel an:



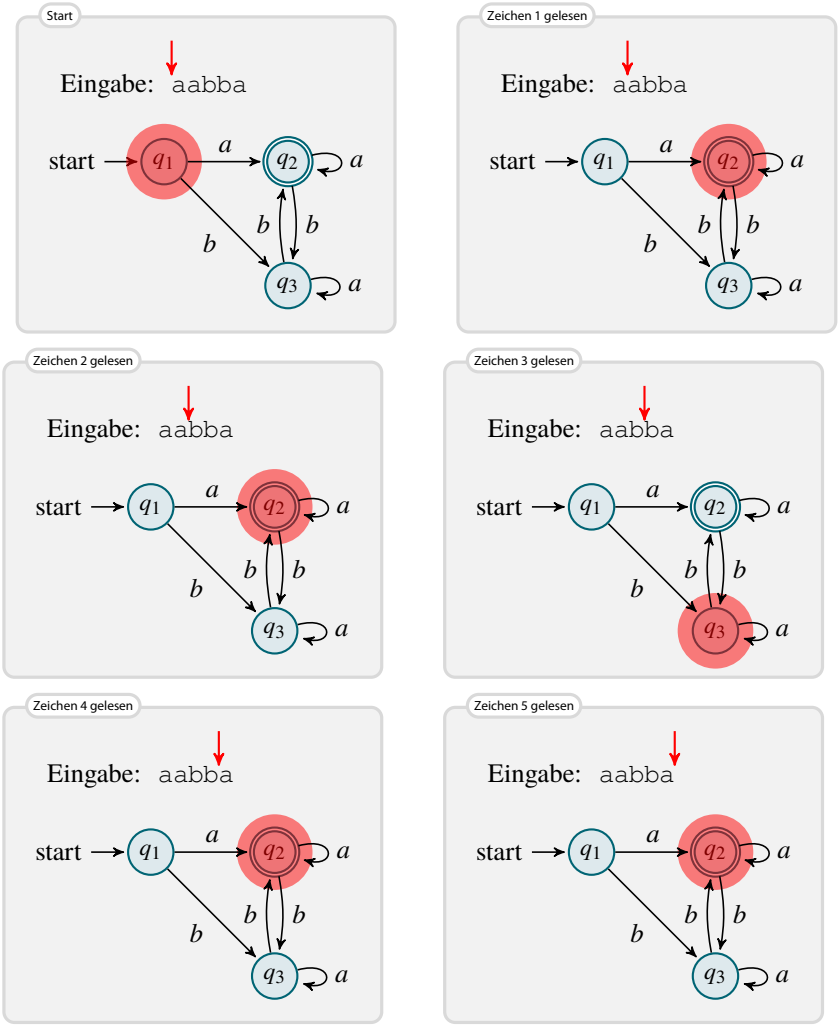
4.2.2 Semantik

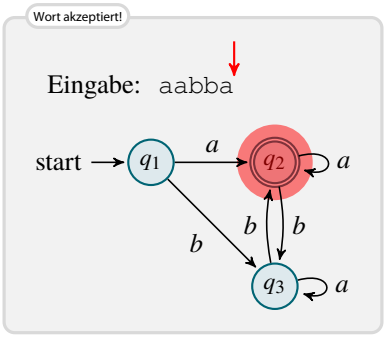
Die Idee hinter der Arbeitsweise eines endlichen Automaten. 4-13

Wenn man einem endlichen Automaten eine Eingabe vorlegt, dann arbeitet er diese wie folgt ab:

- Während der gesamten »Berechnung« (eigentlich wird hier nicht gerechnet, aber seien wir großzügig) ist der Automat immer in einem der Zustände.
- Am Anfang ist er im Anfangszustand (daher der originelle Name).
- Dann liest der Automat das erste Zeichen.
- Die Zustandsübergangsfunktion sagt ihm dann, in welchen Zustand er wechseln soll.
- Das macht er dann auch und liest das nächste Zeichen, schaut wieder in der Zustandsübergangsfunktion nach, wechselt den Zustand, liest das nächste Zeichen, und so weiter.
- Nachdem er das letzte Zeichen gelesen hat, schaut er nach, ob der Zustand *akzeptierend* ist.
  - Wenn ja, so *akzeptiert* er das Wort.
  - Wenn nein, so *verwirft* er das Wort.

Beispiel der Arbeitsweise eines Automaten. 4-14





4-15

Semantik eines endlichen Automaten

▶ **Definition:** Fortgesetzte Zustandsübergangsfunktion
Sei $M = (\Sigma, Q, q_0, Q_a, \delta)$ ein DFA. Wir *erweitern* die Funktion δ zu einer Funktion $\delta^* : Q \times \Sigma^* \to Q$. Sie gibt an, wie sich *der Automat bei Eingabe eines ganzen Wortes verhält*:

$$\delta^*(q, \lambda) = q$$
$$\delta^*(q, xw) = \delta^*(\delta(q, x), w) \quad \text{mit } x \in \Sigma \text{ und } w \in \Sigma^*.$$

▶ **Definition:** Akzeptanz von Worten und akzeptierte Sprache
Sei $M = (\Sigma, Q, q_0, Q_a, \delta)$ ein DFA.

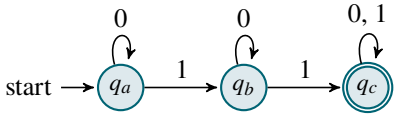
1. Wir sagen, dass M *ein Wort* $w \in \Sigma^*$ *akzeptiert*, wenn $\delta^*(q_0, w) \in Q_a$.
2. Die Menge aller akzeptierte Worte nennen wir *die akzeptierte Sprache*, geschrieben $L(M) = \{w \in \Sigma^* \mid \delta^*(q_0, w) \in Q_a\}$.

4-16

Beispiele von akzeptierten Sprachen.

Beispiel

Der Automat



akzeptiert die Sprache alle Worte, die mindestens zwei Einsen enthalten, also

$$L(M) = \{w \in \{0,1\}^* \mid w \text{ enthält zwei 1en}\}.$$

✎ **Zur Übung**
Geben Sie Automaten an, die die folgenden Sprachen akzeptieren:

1. $L_1 = \{a^n b^m c^k \mid n, m, k \geq 1\}$,
2. $L_2 = \{a^n b^m \mid n, m \geq 0\}$,
3. $L_3 = \{w \in \text{ASCII}^* \mid w \text{ ist ein erlaubter Java-Bezeichner}\}$.

4-17

Eine Vereinfachungen: unvollständige Automaten.

- Häufig erkennt ein Automat an einer bestimmten Stelle im Wort, dass er dieses Wort »nicht mehr akzeptieren möchte«.
- Dann muss man in einen »Fehlerzustand« wechseln,
  1. der nicht akzeptierend ist und
  2. den man nicht mehr verlassen kann.
- Das ist manchmal etwas umständlich, insbesondere bei riesigen Alphabeten wie dem Unicode.

▶ **Definition:** Unvollständiger Automat
Ein *unvollständiger deterministischer endlicher Automat* ist definiert wie ein normaler DFA mit folgenden Änderungen:

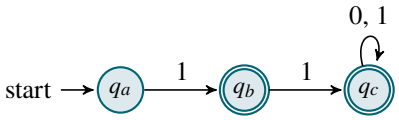
1. Die Zustandsübergangsfunktion δ darf *partiell* sein (sie ist also nicht für alle Paare von Zuständen und Eingabesymbolen definiert).
2. Stößt der Automat während einer Berechnung auf einen Zustand, in dem der nächste Zustand nicht definiert ist, so *wird das Wort sofort verworfen, selbst wenn der Zustand akzeptierend ist*.

Beispiele von akzeptierten Sprachen.

4-18

Beispiel

Der Automat



akzeptiert das Wort 1 sowie alle Worte, die mit zwei Einsen beginnen:

$$L(M) = \{1\} \cup \{11w \mid w \in \{0,1\}^*\}.$$

Beachte: Das Wort 10 wird nicht akzeptiert: $10 \notin L(M)$.

✎ Zur Übung: Tags mit endlichen Automaten parsen.

4-19

Gegen Sie einen DFA an, der einfache öffnende und schließende <**div**>-Tags parsen kann. Er soll folgende Worte akzeptieren:

- <**div**>
- </**div**>

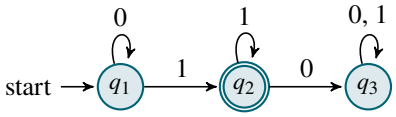
Außerdem sollen Leerzeichen erlaubt sein wie in </ **div** >.

4.2.3 Implementation

Vom Automaten zum Java-Programm.

4-20

Wie implementiert man endliche Automaten?



- Automaten lesen ihre Eingabe zeichenweise.
- Dies kann man durch eine einfache *For-Schleife* implementieren.
- Den aktuellen *Zustand* kann man sich in einer Int-Variable merken.
- Am Ende muss man nachschauen, ob der erreichte Zustand akzeptierend ist.

```
int delta (int q, char x) {
  // q_1 = 1, q_2 = 2, q_3 = 3, ...
  if (q == 1 && x == '0') { return 1; }
  if (q == 1 && x == '1') { return 2; }
  if (q == 2 && x == '0') { return 3; }
  if (q == 2 && x == '1') { return 2; }
  return 3; // alle anderen Fälle
}

int deltaStar (int q, String s) {
  for (int i = 0; i<s.length(); i++) {
    q = delta (q, s.charAt(i));
  }
  return q;
}

boolean isAccepted (String s) {
  return deltaStar (1, s) == 2; // q_1 ist Startzustand, q_2
      akzeptierend
}
```

4.2.4 Korrektheitsbeweise

4-21

Beweisrezept: *Korrektheitsbeweis für DFAs*

Ziel

Man will beweisen, dass ein DFA *M eine Sprache L akzeptiert.*

Rezept

Zeige zwei Richtungen (Beweisrezept »Zwei Richtungen«):

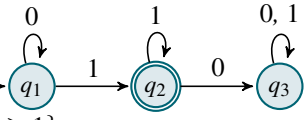
1. Beginne mit »Sei $w \in L$ beliebig.« (Beweisrezept »All-Aussagen beweisen«). Gib dann die Folge von Zuständen an, die M bei Eingabe w durchläuft (Beweisrezepte »Konstruktiver Beweis«). Ende mit »Also gilt $\delta^*(q_0, w) \in Q_a$ und somit $L \subseteq L(M)$«.
2. Beginne mit »Sei $w \in L(M)$ beliebig.« (Beweisrezept »All-Aussagen beweisen«.) Fahre fort mit »Dann gilt $\delta^*(q_0, w) \in Q_a$. Seien q_1 bis q_n mit $q_n \in Q_a$ die Zwischenzustände der Berechnung des Automaten.« (Beweisrezept »Namen vergeben«.) Argumentiere dann, dass $w \in L$ gilt. Ende mit »Also gilt $w \in L$ und somit $L(M) \subseteq L$«.

4-22

Kurzes Beispiel eines Korrektheitsbeweises.

▶ Satz

Sei M folgender Automat:



Dann gilt $L(M) = \{0^n 1^m \mid n \geq 0, m \geq 1\}$.

Beweis. Wir zeigen zwei Richtungen. Sei zunächst $w = 0^n 1^m$ mit $n \geq 0$ und $m \geq 1$. Bei Eingabe von w wird M zunächst n Schritte lang in q_1 verharren, dann (wegen $m \geq 1$) den Zustand zu q_2 wechseln und dann $m-1$ Schritte in q_2 verharren, wo w akzeptiert wird. Also gilt $w \in L(M)$.
Für die andere Richtung sei $w \in L(M)$. Damit M ein Wort akzeptiert, muss die Berechnung in q_2 enden. Da die Berechnung in q_1 begann, muss w eine 1 enthalten (vom Wechsel von q_1 zu q_2). Vor dieser 1 können nur 0en sein, nach ihr nur 1en. Also gilt $w = 0^n 1 1^k$ mit $n, k \geq 0$ und folglich $w \in \{0^n 1^m \mid n \geq 0, m \geq 1\}$. □

4.3 Verhältnis zu regulären Sprachen I

4-23

Was können endliche Automaten?

- Endliche Automaten können sich nur eine feste Menge von Möglichkeiten in ihrem Zustand merken.
- Das ist ganz ähnlich wie bei regulären Grammatiken, wo auch nur immer das aktuelle Nonterminale angab, was alles passieren kann.
- Dies ist kein Zufall: Wir werden in den folgenden Kapiteln noch zeigen, dass *deterministische endliche Automaten genau die regulären Sprachen akzeptieren*.
- Wir werden gleich sehen, dass DFAs *nur reguläre Sprachen akzeptieren*.
- Für die Umkehrung (alle regulären Sprachen werden von DFAs akzeptiert) wird allerdings noch etwas mehr Theorie benötigt.

4-24

Endliche Automaten können nicht mehr als reguläre Grammatiken.

▶ Satz

Sei M ein DFA. *Dann gilt* $L(M) \in$ REG.

Beweisidee

Wir können eine Automaten M wie folgt in Grammatiken G umwandeln:

- Die Terminale sind gerade die Symbole in Σ.
- Die Nonterminale sind *gerade die Zustände in* Q.
- Das Startsymbol ist der Startzustand.

- Kann der Automat vom Zustand q zum Zustand q' durch Lesen eines Zeichens x kommen (also $\delta(q,x) = q'$), so gibt es die Regel $q \to xq'$.
- Weiter gibt es, um die Ableitung abzuschließen, für jedes $q \in Q_a$ die Regel $q \to \lambda$.

Die Idee ist dann, dass Berechnungen des Automaten Eins-zu-Eins den Ableitungen in der Grammatik entsprechen.

Beweis. Sei $M = (\Sigma, Q, q_0, Q_a, \delta)$ ein DFA. Sei G folgende Grammatik: Skript

- $T = \Sigma$
- $N = Q$
- $S = q_0$
- Für jedes Paar $q \in Q$ und $x \in \Sigma$ gibt es die Regel $q \to x\delta(q,x)$.
- Weiter gibt es für jedes $q \in Q_a$ die Regel $q \to \lambda$.

Wir behaupten, dass $L(M) = L(G)$ gilt.

Für die erste Richtung sei $w \in L(M)$ beliebig. Seien q_0, q_1, …, q_n mit $q_n \in Q_a$ die Zustände, die M bei Eingabe w durchläuft. Dann gilt für alle i, dass $\delta(q_i, w[i+1]) = q_{i+1}$. Betrachte nun folgende Ableitung:

$$q_0 \Rightarrow w[1]q_1 \Rightarrow w[1]w[2]q_2 \Rightarrow \cdots \Rightarrow wq_n \Rightarrow w.$$

Jeder Ableitungsschritt benutzt eine erlaubte Regel, inklusive der letzten. Also gilt $q_0 \Rightarrow^* w$ und somit $w \in L(G)$.

Für die zweite Richtung sei $w \in L(G)$. Dann gilt $q_0 \Rightarrow^* w$ via einer Ableitungskette

$$q_0 \Rightarrow w_1 \Rightarrow w_2 \Rightarrow \cdots \Rightarrow w_n = w.$$

Jedes w_i, außer w_n, endet nun mit einem Nonterminal – nennen wir es q_i. Weiter lässt sich dann jedes w_i schreiben als $w_i = w[1]w[2]\ldots w[i]q_i$. Folglich ist jeweils im Schritt $w_i \Rightarrow w_{i+1}$ die Regel $q_i \to w[i+1]q_{i+1}$ angewandt worden; außer im letzten, wo die Regel $q_n \to \lambda$ angewandt wurde. Insgesamt erhalten wir, dass

1. für jedes i gilt $\delta(q_i, w[i+1]) = q_{i+1}$ und
2. $q_n \in Q_a$.

Folglich gilt $\delta^*(q_0, w) = q_n \in Q_a$ und somit $w \in L(M)$. □

Zusammenfassung dieses Kapitels

1. Endliche Automaten lesen Eingaben zeichenweise und wechseln dabei in jedem Schritt ihren Zustand. 4-25
2. Ein Automat *akzeptiert ein Wort*, wenn er es *komplett* einliest und dabei in einem *akzeptierenden Zustand endet*.
3. Ein Automat *akzeptiert die Sprache* aller Worte, die er akzeptiert.
4. Alle von deterministischen endlichen Automaten akzeptierten Sprachen sind regulär.

Zum Weiterlesen

[1] R. Reischuk. *Unterlagen zum Lehrmodul Theoretische Informatik,* Vorlesungsskript, 2008. Kapitel 1.4.

Übungen zu diesem Kapitel

Übung 4.1 Automaten implementieren I, mittel

Auf Folie 4-20 wird für einen konkreten Automaten gezeigt, wie sich dieser in ein Java-Programm umwandeln lässt. In dieser Aufgabe sollen Sie nun eine Methode entwerfen, bei der der Automat nicht mehr »fest verdrahtet« im Java-Code ist, sondern die Zustandsübergangsfunktion als Variable gegeben ist.

Die folgende Klasse repräsentiert dabei einen Automaten:

```
class Automaton {

  // Sigma ist einfach gleich ASCII

  int sizeOfQ;
  // Q ist die Menge {0,...,sizeOfQ-1}

  int [][] delta;
  // delta[q][x] ist gerade delta(q,x)

  int q_0;
  // Startzustand

  boolean [] Q_a;
  // Q_a[i] ist genau dann wahr, wenn i \in Q_a

  boolean isAccepted (String s) {
    // Ihr Code!
  }
}
```

Übung 4.2 Automaten implementieren II, mittel

In Aufgabe 4.1 wurde eine Methode entworfen, die für jeden beliebigen Automaten funktioniert. Prinzipiell kann dieser also auch erst zur Laufzeit erstellt werden. Häufig sind Automaten aber bereits zur Compile-Zeit bekannt und fest. In diesem Fall kann es sinnvoller sein, die Automaten so wie auf Folie 4-20 »fest zu verdrahten«.

Damit man den Code nicht selber programmieren muss, verwendet man Methoden, die für ein Automaten-Objekt einen Programmtext wie auf Folie 4-20 als Ausgabe produzieren. Entwerfen Sie ein solche Methode als Erweiterung der Klasse aus Aufgabe 4.1.

```
class Automaton {
  ...

  String produceJavaProgram () {
    // Ihr Code!
    // (Rückgabe soll ein Java-Progammtext sein)
  }
}
```

Übung 4.3 Automaten konstruieren, mittel

Geben Sie endliche Automaten für folgende Sprachen an, sowohl in Tupel- als auch in Graphschreibweise. Begründen Sie die Korrektheit Ihrer Konstruktionen. (Ein formaler Beweis ist an dieser Stelle nicht erforderlich, jedoch sollte aus Ihrer Begründung auf die Korrektheit Ihres Automaten geschlossen werden können)

1. $L_1 =$ die Menge der geraden Zahlen in Dezimaldarstellung ohne führende Nullen.
2. $L_2 = \{w \in \{0,1\}^* \mid X \text{ enthält } 00 \text{ genau einmal als Teilwort}\}$.
3. $L_3 = \{w \in \{a,b\}^* \mid (|w|_a - |w|_b)_{\mathrm{mod}\,3} = 0\}$.

Bei dem Automaten für die Sprache L_1 können Sie davon ausgehen, dass der Automat die höchstwertige Dezimalstelle zuerst liest. Bei der Sprache L_3 bezeichnet $|w|_a$ beziehungsweise $|w|_b$ die Anzahl der »a« oder »b« im Wort w.

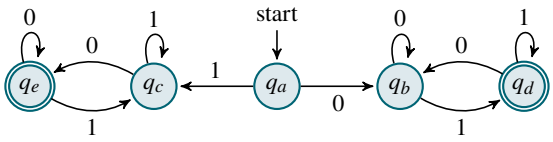
Übung 4.4 Formalen Korrektheitsbeweis über Automaten führen, schwer

Geben Sie für folgende Sprache einen endlichen Automaten an. Beweisen Sie daraufhin die Korrektheit Ihrer Konstruktion.

$$L_3 = \{a^{2n}b^{3m} \mid n,m \in \mathbb{N}\}.$$

Übung 4.5 Sprachen aus den Automaten erkennen, mittel

Welche Sprache L akzeptiert der folgende endliche Automat? Geben Sie eine reguläre Grammatik G an, so dass $L(G) = L$ ist.



Übung 4.6 Automaten konstruieren, mittel

Konstruieren Sie endliche Automaten für folgende Sprachen über $\Sigma = \{0,1\}$. Geben Sie diese sowohl in Tupelschreibweise als auch in Graphschreibweise an. Begründen Sie die Korrektheit Ihrer Konstruktion (ein formaler Beweis ist an dieser Stelle nicht erforderlich).

1. $A = \{w \in \Sigma^* \mid |w| \text{ ist ungerade}\}$.
2. $B = \{w \in \Sigma^* \mid w \text{ enthält 00 nicht als Teilwort}\}$.

Übung 4.7 Korrektheit von endlichen Automaten beweisen, schwer

Geben Sie für folgende Sprachen über $\Sigma = \{0,1\}$ endliche Automaten in Graphschreibweise an. Beweisen Sie die Korrektheit Ihrer Konstruktion.

1. $D = \{w \in \Sigma^* \mid w \text{ enthält genau drei Einsen}\}$.
2. $C = \{w \in \Sigma^* \mid w \text{ enthält eine ungerade Anzahl von Nullen}\}$.

5-1

Kapitel 5
Grenzen regulärer Sprachen

Pump up the ~~volume~~ word!

5-2

Lernziele dieses Kapitels

1. Aussage des Pumping-Lemmas für reguläre Sprachen verstehen
2. Das Pumping-Lemma anwenden können, um Nichtregularität zu beweisen

Inhalte dieses Kapitels

Worum es heute geht

Sind Sie auch mal vergesslich? Sie gehen in eine Raum hinein und wenn Sie drinnen angekommen sind, haben Sie vergessen, was Sie dort eigentlich wollten? Ganz schlimm ist es, wenn Sie mehrere Sachen erledigen wollten?

Vielleicht sind Sie ja ein endlicher Automat! Diese vergessen nämlich auch recht schnell und recht viel. Tatsächlich kann sich ein DFA ja wirklich nicht sonderlich viel merken, eben nur soviel, wie er Zustände hat. Wenn er Eingaben bearbeiten muss, die länger sind als seine Zustandsmenge groß ist, so muss er notgedrungen einiges »über den Anfang vergessen«. Diese Vergesslichkeit kann man ausnutzen, wenn man zeigen möchte, dass Automaten bestimmte Sprachen nicht akzeptieren können. Bei vielen Sprachen »muss man sich beliebig viel merken«, möchte man wirklich nur genau die Worte der Sprache korrekt akzeptieren.

Leider ist es nicht immer ganz klar, wieviel man sich genau merken muss bei einer bestimmten Sprache. Automaten können sich nämlich Eselsbrücken bauen – sie merken sich dann vielleicht nicht alles über den Wortteil, den sie abgearbeitet haben, aber eben einige wichtige Dinge. So kann sich ein Automat zwar nicht die genaue Anzahl an Zeichen merken, die er bereits gelesen hat. Er kann sich aber sehr wohl merken, ob diese Anzahl gerade war oder durch Sieben teilbar war oder ob auf jedes *a* ein *b* gefolgt ist.

In diesem Kapitel wird ein berühmter Satz vorgestellt, genannt das *Pumping-Lemma*, mit dem man die »Vergesslichkeit« von Automaten sehr genau fassen kann. Das Hauptproblem bei der Anwendung des Pumping-Lemmas ist allerdings, dass man es eigentlich »verkehrtherum« anwenden muss: Die eigentlich Aussage des Lemmas (»Alle von DFAs akzeptierte Sprachen haben die Pump-Eigenschaft.«) ist nämlich eher uninteressant; wichtig ist der Umkehrschluss (»Hat eine Sprache nicht die Pump-Eigenschaft, so wird sie auch nicht von DFAs akzeptiert.«)

5-4

Was können Automaten? Was können sie nicht?

- Viele Sprachen lassen sich von DFAs akzeptieren.
- Darunter sind auch eher komplexe wie »gültige E-Mail-Adressen« oder »gültige XML-Tags« oder »gültige Datumsangaben«.
- Intuitiv sollte es aber auch Sprachen geben, die *nicht* von DFAs akzeptiert werden können: Alle Sprachen, bei denen man sich *mehr merken muss als man sich mittels der endlich vielen Zustände merken kann*.

Leider ist die Beschreibung »wo man sich mehr merken muss als man in einem Zustand speichern kann« etwas schwammig.

Heutiges Ziel
Gesucht ist ein Verfahren, mit dem man *beweisen* kann, dass eine Sprache nicht von Automaten akzeptiert werden kann.

5.1 Das Pumping-Lemma

5.1.1 Vorbereitende Überlegungen

Woran scheitern endliche Automaten? 5-5

- Während eine Automat ein Wort liest, kann er sich nur sehr wenig merken.
- Genaugenommen merkt er sich ja nur, in welchem *Zustand* er gerade ist.
- Hat der Automat beispielsweise 100 Zustände, so kann er sich nicht mehr als 100 unterschiedliche »Situationen« merken.

Beispiel
Wie könnte ein Automat die Sprache $\{a^k b^k \mid k \geq 0\}$ akzeptieren?

- Er muss überprüfen, ob die Anzahl an a's vorne gleich der Anzahl an b's hinten ist.
- Dafür muss er sich die Anzahl irgendwie »merken«.
- Hat er aber nur 100 Zustände, so kann er sich nicht merken, ob es nun 973 oder doch eher 974 a's waren.
- Andererseits: Er könnte sich ja merken, ob die Anzahl gerade oder ungerade war – dann könnte er doch zwischen 973 und 974 unterscheiden.
- Dann könnte er aber wieder nicht zwischen 973 und 975 unterscheiden.
- Irgendwie brauchen wir ein allgemeines Argument.

Zwei entscheidende Ideen. 5-6

- Nehmen wir an, es gäbe einen Automaten mit 100 Zuständen, der $\{a^k b^k \mid k \geq 0\}$ akzeptiert.
- Betrachten wir nun ein Wort in der Sprache mit sehr vielen a's, also beispielsweise $a^{1000}b^{1000}$.

Nun kommen die zwei entscheidenden Ideen:

1. Wenn er das Wort $a^{1000}b^{1000}$ liest, so wird er *innerhalb der ersten 100 a's einen Zustand zweimal erreichen*.
2. Zwischen den beiden Erreichungen des Zustands hat er eine bestimmte Anzahl a's gelesen, sagen wir 23. Fügen wir nun *weitere 23 a's an dieser Stelle ein, so »merkt« dies der Automat nicht*. Ebenso »merkt« er es nicht, wenn wir 46 oder 69 a's einfügen. Deshalb wird er auch $a^{1023}b^{1000}$ und $a^{1046}b^{1000}$ und $a^{1069}b^{1000}$ akzeptieren – was er nicht soll.

Formalisiertes Scheitern. 5-7

- Man kann das Argument für $\{a^k b^k \mid k \geq 0\}$ auch allgemein fassen.
- Aus Zahlen wie »1000« oder »100« werden dann Variablen.
- Aus »man fügt weitere 23 a's ein« wird »man verdoppelt oder verdreifacht ein bestimmtes Teilwort«.
- Dieses Verdoppeln oder Verdreifachen nennt man verspielt »aufpumpen« des Teilwortes.

5.1.2 Der Satz und sein Beweis

5-8

Von DFAs akzeptierte Sprachen haben die Pump-Eigenschaft.

▶ Satz: Pumping-Lemma
Die Sprache L sei durch einen DFA *akzeptierbar. Dann*

- *existiert eine Wortlänge $n \in \mathbb{N}$, so dass*
- *für alle Worte $w \in L \cap \Sigma^{\geq n}$ gilt, es*
- *existiert eine Zerlegung $w = x \circ y \circ z$ mit $|y| \geq 1$ und $|x \circ y| \leq n$, so dass*
- *für alle $i \in \mathbb{N}$ gilt*

$$x \circ y^i \circ z \in L.$$

Bemerkungen:

- Den Wechsel der Quantoren »für alle« mit »es existiert« nennt man *Alternationen* – diese machen die Aussage etwas verwirrlich.
- Im Wort xyz wird der mittlere Teil y in xy^iz beliebig oft wiederholt, dies nennt man *aufpumpen*.

Kommentare zum Beweis

Beweis. Sei $L = L(M)$ mit $M = (\Sigma, Q, q_0, Q_a, \delta)$. Um den Satz zu beweisen, müssen wir als erstes ein $n \in \mathbb{N}$ angeben. Wir wählen $n = |Q| + 1$.[1]
Als nächstes sollen wir etwas »für alle Worte $w \in L \cap \Sigma^{\geq n}$« zeigen. Sei dazu $w \in L \cap \Sigma^{\geq n}$ beliebig.[2] Seien $q_0, q_1, \ldots, q_{|w|}$ die Zustände, die der Automat bei Eingabe w durchläuft.[3] Dann gilt $q_{|w|} \in Q_a$. Da $|w| > |Q|$, muss sich innerhalb der ersten n Zustände ein Zustand wiederholen. Es gibt also zwei Indizes i und j mit $i < j \leq |Q| + 1 = n$, so dass $q_i = q_j$.[4]
Wähle $x = w[1]w[2]\ldots w[i]$ und $y = w[i+1]w[i+2]\ldots w[j]$ und $z = w[j+1]w[j+2]\ldots w[|w|]$.[5] Dann gilt $w = xyz$ und weiter $\delta^*(q_0, x) = q_i$ und $\delta^*(q_i, y) = q_j = q_i$ und $\delta^*(q_j, z) = q_{|w|} \in Q_a$.[6]
Sei nun $i \in \mathbb{N}$ beliebig.[7] Bei Eingabe von xy^iz wird M die Berechnung im Zustand q_0 beginnen, dann – am Ende von x – den Zustand q_i erreicht haben, dann nach dem ersten y wieder den Zustand q_i erreicht haben, nach den zweiten y ebenfalls und so weiter bis zum letzten y. Vom Zustand $q_i = q_j$ aus erreicht der Automat dann nach dem Lesen von z den Zustand $q_{|w|} \in Q_a$. Also gilt $xy^iz \in L$.[8] □

[1] Die Wahl von n fällt hier vom Himmel, wird aber später klar.
[2] Rezept »All-Aussagen«
[3] Rezept »Namen vergeben«
[4] Hier ist das Schleifen-Argument
[5] y ist das Teilwort, dessen Durchlaufen der Automat »nicht merkt«
[6] Dieser Umstand nochmal symbolisch aufgeschrieben.
[7] Rezept »All-Aussagen«
[8] Was ja die Behauptung war

5-9

Was wir eigentlich wollen.

- Das Pumping-Lemma besagt, dass alle von Automaten akzeptierte Sprachen eine »Pump-Eigenschaft« haben.
- Im Umkehrschluss können Sprachen, die die Pump-Eigenschaft nicht haben, auch nicht von Automaten akzeptiert werden.
- Die folgende Folgerung ist einfach nur dieser Umkehrschluss.

▶ Folgerung
Sei L eine Sprache, so dass

- *für alle Wortlängen $n \in \mathbb{N}$*
- *existiert ein Wort $w \in L \cap \Sigma^{\geq n}$, so dass*
- *für alle Zerlegungen $w = x \circ y \circ z$ mit $|y| \geq 1$ und $|x \circ y| \leq n$*
- *existiert ein $i \in \mathbb{N}$ mit*

$$x \circ y^i \circ z \notin L.$$

Dann lässt sich L nicht durch DFAs *akzeptieren.*

5.2 Anwendungen

5.2.1 Das Spiel

Streit um Automaten.

5-10

Public domain

Ausgangssituation

- Man streitet sich, ob eine Sprache L von Automaten akzeptiert werden kann oder nicht.
- Der *Skeptiker* ist sich nicht sicher.
- Der *Automatenliebhaber* ist überzeugt, dass L akzeptiert werden kann und will den Skeptiker davon überzeugen.
- Der *Widerleger* ist überzeugt, dass L *nicht* akzeptiert werden kann und will den Skeptiker überzeugen.

Strategien

- Der *Automatenliebhaber* kann den Skeptiker überzeugen, indem er einen Automaten angibt, der L akzeptiert – inklusive Korrektheitsbeweis, da der Skeptiker sehr skeptisch sind.
- Der *Widerleger* kann die Folgerung aus dem Pumping-Lemma benutzen.

Das Pumping-Lemma als Dialog.

5-11

- Da der Skeptiker sehr skeptisch ist, muss der Widerleger ihn überzeugend überzeugen.
- Der Skeptiker ist überzeugt, dass L nicht akzeptiert werden kann, wenn man ihn überzeugt, dass die »für alle … existiert … für alle … existiert …« Bedingung gilt.
- Dazu bietet der Widerleger an, folgendes Spiel zu spielen:
  1. Für das erste »für alle Wortlängen $n \in \mathbb{N}$« darf der Skeptiker ein beliebiges n nennen.
  2. Als Antwort wird der Widerleger ein Wort $w \in L \cap \Sigma^{\geq n}$ nennen.
  3. Dann darf der Skeptiker wieder für »für alle Zerlegungen $w = x \circ y \circ z$« eine Zerlegung nennen.
  4. Als Antwort wird der Widerleger ein $i \in \mathbb{N}$ nennen mit $x \circ y^i \circ z \notin L$.

Ein Beispieldialog.

5-12

Widerleger Hallo Skeptiker!

Skeptiker Hallo Widerleger!

Widerleger Hast du neulich den Artikel gelesen, wonach Wissenschaftler herausgefunden haben, dass $\{a^k b^k \mid k \geq 0\}$ nicht von DFAs akzeptiert werden kann?

Skeptiker Ja, das stand in der Zeit und auch im Spiegel und selbst in der FAZ. Ich glaube ja, die schreiben alle nur voneinander ab, die überprüfen doch ihre Quellen gar nicht.

Widerleger Doch, das stimmt, ich werde dich mit der Folgerung aus dem Pumping-Lemma überzeugen.

Skeptiker Na, das möchte ich sehen…

Widerleger Also, die Folgerung sagt ja, dass, wenn für alle Zahlen $n \in \mathbb{N}$ etwas Bestimmtes gilt, dann kann $\{a^k b^k \mid k \geq 0\}$ nicht von DFAs akzeptiert werden.

Skeptiker In der Tat – nur müsstest du mich erstmal überzeugen, dass das tatsächlich für alle n der Fall ist. Das dürfte dir schwerfallen.

Widerleger Wieso?

Skeptiker Weil es unendliche viele n gibt!

Widerleger Ah, aber ich behaupte, du kannst mir keines nennen, wo's nicht klappt.

Skeptiker Na gut, ich bin bei $n = 100$ ziemlich skeptisch. Dann soll es nämlich angeblich ein $w \in L \cap \Sigma^{\geq n}$ geben, so dass etwas gilt.

Widerleger Jup. So ein w gibt es, nämlich $w = a^{100} b^{100}$.

Skeptiker Von mir aus. Jetzt soll für alle Zerlegung von w etwas gelten. Da gibt es ja schrecklich viele!

Widerleger Nenne mir irgendeine, bei der du glaubst, dass es nicht klappt.

Skeptiker Ich bin bei $w = a^{30} \circ a^{70} \circ b^{100}$ skeptisch, also für $x = a^{30}$, $y = a^{70}$ und $z = b^{100}$.

Widerleger Wie kann man da skeptisch sein? Für $i = 2$ gilt doch jetzt $x \circ y^i \circ z = a^{170}b^{100}$ – und das ist sicher kein Wort in der Sprache.

Skeptiker In der Tat. Dann möchte ich gerne noch eine andere Zerlegung probieren. Sagen wir $w = a^{20} \circ a^{20} \circ a^{60}b^{100}$.

Widerleger So skeptisch kann man doch gar nicht sein. Wieder gilt für $i = 2$, dass $x \circ y^i \circ z = a^{120}b^{100}$ kein Element der Sprache ist.

Skeptiker Ok, ok, ich bin überzeugt. Da war wohl meine Wahl von n ganz am Anfang nicht sonderlich geschickt.

Widerleger Es ist doch völlig egal, wie du n wählst. Wenn du $n = 1000$ wählst, dann wähle ich eben das Wort $a^{1000}b^{1000}$.

Skeptiker Hmmm, tja, grübel. In der Tat. Mir scheint, die Voraussetzung der Folgerung stimmt und somit lässt sich L tatsächlich nicht von einem DFA akzeptieren.

Regie

Ein Gruppe von Skeptikern und ein Gruppe von Widerlegern kommen auf die Bühne.

Im ersten Spiel geht um die Sprache $L = \{0^k 1^m 2^k \mid k, m \geq 10\}$*:*

1. *Die Skeptiker suchen sich eine Zahl n aus.*
2. *Die Widerleger müssen ein Wort* $w \in L \cap \Sigma^{\geq n}$ *nennen.*
3. *Die Skeptiker suchen sich eine Zerlegung von w aus.*
4. *Die Widerleger müssen eine Zahl i nennen mit* $xy^iz \notin L$*.*
   - *Schaffen sie dies, so haben die Widerleger gewonnen.*
   - *Schaffen sie dies nicht, so haben die Skeptiker gewonnen.*

Bei diesem Spiel haben die Widerleger eine Gewinnstrategie: *Wenn sie geschickt spielen, dann gewinnen sie immer.*

Im zweiten Spiel geht es um die Sprache $L = \{0^k 1^j \mid k, j \geq 10\}$*. Hier haben die Skeptiker eine Gewinnstrategie!*

5.2.2 Nichtregularität

5-13

Beweisrezept: *Pumping-Lemma anwenden*

Ziel

Man will beweisen, dass eine Sprache L nicht von DFAs *akzeptiert werden kann.*

Rezept

1. Beginne mit »Wir zeigen die Behauptung durch die Kontraposition des Pumping-Lemmas.«
2. Fahre fort mit »Sei n beliebig.«
3. Fahre fort mit »Dann ist $w = \ldots$ ein Element von $L \cap \Sigma^{\geq n}$.«, wobei natürlich »…« geeignet gewählt sein muss.
4. Fahre fort mit »Sei nun $w = xyz$ eine beliebige Zerlegung mit $|y| \geq 1$ und $|xy| \leq n$.«
5. Schließe mit »Wähle nun $i = \ldots$« und argumentiere, dass $xy^iz \notin L$ gilt.

Ein einfaches Beispiel.

5-14

▶ **Satz**
$L = \{a^k b^k \mid k \geq 1\}$ *lässt sich nicht von* DFAs *akzeptieren.*

Beweis. Wir zeigen die Behauptung durch die Kontraposition des Pumping-Lemmas.[1] Sei n beliebig. Dann ist $w = a^n b^n$ ein Element von $L \cap \Sigma^{\geq n}$.[2] Sei nun $w = xyz$ eine beliebige Zerlegung mit $|y| \geq 1$ und $|xy| \leq n$.
Dann gilt, dass $y = a^p$ für ein $p \geq 1$.[3] Für $i = 2$ gilt dann, dass $xy^iz = a^{n+p}b^n$. Da $n + p \neq n$, ist $xy^iz \notin L$. □

Kommentare zum Beweis
[1] Text aus dem Rezept
[2] Strategie des Widerlegers
[3] Innerhalb der ersten n Buchstaben von w gibt es nur a's.

Noch ein Beispiel.

5-15

▶ **Satz**
$L = \{0^k 1^m 0^k \mid k, m \geq 1\}$ *lässt sich nicht von* DFAs *akzeptieren.*

Beweis. Wir zeigen die Behauptung durch die Kontraposition des Pumping-Lemmas.[1] Sei n beliebig. Dann ist $w = 0^n 10^n$ ein Element von $L \cap \Sigma^{\geq n}$.[2] Sei nun $w = xyz$ eine beliebige Zerlegung mit $|y| \geq 1$ und $|xy| \leq n$.
Dann gilt, dass $y = 0^p$ für ein $p \geq 1$.[3] Für $i = 2$ gilt dann, dass $xy^iz = 0^{n+p}10^n$. Da $n + p \neq n$, ist $xy^iz \notin L$. □

Kommentare zum Beweis
[1] Text aus dem Rezept
[2] Strategie des Widerlegers
[3] Innerhalb der ersten n Buchstaben von w gibt es nur 0en.

✎ **Zur Übung**
Zeigen Sie, dass $\{w \mid w$ enthält gleich viele a's wie b's$\}$ nicht von einem DFA akzeptiert werden kann.

5-16

Ein letztes Beispiel.

5-17

▶ **Satz**
$L = \{a^n \mid n$ *ist eine Quadratzahl*$\}$ *lässt sich nicht von* DFAs *akzeptieren.*

Beweis. Wir zeigen die Behauptung durch die Kontraposition des Pumping-Lemmas. Sei n beliebig. Dann ist $w = a^{n^2}$ ein Element von $L \cap \Sigma^{\geq n}$. Sei nun $w = xyz$ eine beliebige Zerlegung mit $|y| \geq 1$ und $|xy| \leq n$.
Dann gilt, dass $y = a^p$ für ein p mit $1 \leq p \leq n$. Für $i = 2$ gilt dann, dass $xy^iz = a^{n^2+p}$.[1] Nun ist aber $n^2 + p \leq n^2 + n < n^2 + 2n + 1 = (n+1)^2$. Also ist die Anzahl an a's in $xy^iz = a^{n^2+p}$ keine Quadratzahl und somit $xy^iz \notin L$. □

Kommentare zum Beweis
[1] So weit ist alles normal. Jetzt kommt ein neues Argument.

Zusammenfassung dieses Kapitels

5-18

1. Das *Pumping-Lemma* besagt, dass alle von Automaten akzeptierten Sprachen eine *Pump-Eigenschaft* haben.
2. Im *Umkehrschluss* können Sprachen, die die Pump-Eigenschaft nicht haben, auch nicht von DFAs akzeptiert werden.
3. Man kann das Pumping-Lemma *nicht* benutzen um zu zeigen, dass Sprachen von DFAs akzeptierbar sind – sondern nur um zu zeigen, dass sie *nicht* akzeptierbar sind.

Zum Weiterlesen

[1] R. Reischuk. *Unterlagen zum Lehrmodul Theoretische Informatik,* Vorlesungsskript, 2008. Kapitel 2.5.

Übungen zu diesem Kapitel

Übung 5.1 Pumping-Lemma als notwendiges Kriterium für reguläre Sprachen, mittel

Zeigen Sie mit Hilfe der Folgerung aus dem Pumping-Lemma, dass folgende Sprachen nicht regulär sind:

1. $L_1 = \{a^n b^m a^{n+m} \mid n,m \in \mathbb{N}\}$,
2. $L_2 = \{ww^{\text{rev}} \mid w \in \{0,1\}^*\}$,
3. $L_3 = \{a^n \mid n \text{ ist eine Primzahl}\}$.

Übung 5.2 Pumping-Lemma üben, mittel

Zeigen Sie mit Hilfe der Kontraposition des Pumping-Lemmas, dass folgende Sprachen über $\Sigma = \{0,1\}$ nicht regulär sind:

1. $L_1 = \{ww \mid w \in \{0,1\}^*\}$,
2. $L_2 = \{a^n b^m a^{n \cdot m} \mid n \in \mathbb{N}\}$.

Übung 5.3 Pumping-Lemma ist nur ein notwendiges Kriterium, mittel

Gegeben sei die Sprache $L = \{0^n 1^m 0^m \mid n,m \in \mathbb{N}\} \cup \{1^n 0^m \mid n,m \in \mathbb{N}\}$. Zeigen Sie, dass L Pumping-Eigenschaft erfüllt, das heißt: Es existiert eine Wortlänge $n \in \mathbb{N}$, so dass für alle Worte $w \in L \cap \Sigma^{\geq n}$ gilt, es existiert eine Zerlegung $w = x \circ y \circ z$ mit $|y| \geq 1$ und $|x \circ y| \leq n$, so dass für alle $i \in \mathbb{N}$ gilt, dass $x \circ y^i \circ z \in L$.

Übung 5.4 Pumping-Eigenschaft verstehen, mittel

Zeigen Sie, dass die Sprache $L = \{a^{6n} \mid n \in \mathbb{N}\}$ die Pumping-Eigenschaft erfüllt, das heißt: Es existiert eine Wortlänge $n \in \mathbb{N}$, so dass für alle Worte $w \in L \cap \Sigma^{\geq n}$ gilt, es existiert eine Zerlegung $w = x \circ y \circ z$ mit $|y| \geq 1$ und $|x \circ y| \leq n$, so dass für alle $i \in \mathbb{N}$ gilt, dass $x \circ y^i \circ z \in L$.

6-1

Kapitel 6
Nichtdeterministische endliche Automaten

Eine akzeptierende Berechnungen genügt zur Akzeptanz

Lernziele dieses Kapitels

1. Syntax und Semantik von NFAs beherrschen
2. Äquivalenzbeweis zu DFAs verstehen

6-2

Inhalte dieses Kapitels

Worum es heute geht

Um es gleich vorneweg zu sagen: Nichtdeterministische endliche Automaten gibt es nicht. Vom philosophischen Standpunkt aus ist diese Behauptung starker Tobak, vom praktischen jedoch glasklar: Die Dinger kann man nicht bauen.

Nichtdeterministische endliche Automaten zeichnen sich dadurch aus, dass es in vielen ihrer Zustände viele Möglichkeiten geben kann, wie es weiter geht – bei identischen Eingaben. Soweit ist das noch ganz realistisch, denn auch reale Computer verhalten sich reichlich nichtdeterministisch, wie jeder aus häufig leidvoller Erfahrung zu berichten weiß. Der Nichtdeterminismus realer Computer lässt sich kurz auf die Formel bringen »Gestern ging's noch.«

Völlig unrealistisch ist nun aber, dass nichtdeterministische Automaten ein Wort akzeptieren, wenn sie es einfach nur irgendwie schaffen können, in einen akzeptierenden Zustand am Wortende zu gelangen. Das ist in etwa so, als ob eine Fußballmannschaft ein Spiel dadurch gewinnen könnte, dass sie *prinzipiell* in der Lage wäre mehr Bälle in das Tor zu bekommen als der Gegner. Oder als ob Sie einen hitzige Debatte mit Ihrem Partner oder Ihrer Partnerin dadurch gewinnen könnten, dass Sie *prinzipiell* das Richtige sagen könnten. Oder als ob Sie volle Punkte für einen Übungszettel allein deshalb bekämen, weil sie *prinzipiell* wüssten, wie das geht. In der Realität genügt es eben nicht, *prinzipiell* etwas schaffen zu können, man muss es *tatsächlich* hinbekommen.

Wozu beschäftigt man sich mit Nichtdeterminismus, wenn er vom praktischen Standpunkt aus so unnütz ist? Es gibt (mindestens) zwei gute Gründe:

1. Nichtdeterminismus hat sich als nützliches Hilfsmittel in der Theorie herausgestellt.
2. Wie schon gesagt verhalten sich reale Systeme durchaus nichtdeterministisch, nur die »Akzeptanzbedingung« macht in der Praxis keinen Sinn. Man möchte Nichtdeterminismus ganz allgemein einfach »verstehen«.

Glücklicherweise stellt sich heraus, dass niemand NFAs wirklich würde bauen müssen: Wir werden sehen, dass sie auch nicht mehr können als DFAs.

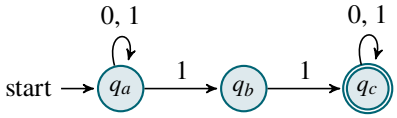
6.1 Nichtdeterministische Automaten

6.1.1 Idee

6-4

Automaten, die sich nicht entscheiden können.

Betrachten wir folgenden Automaten:



Welche Sprache akzeptiert der Automat?

- Intuitiv akzeptiert der Automat gerade diejenigen Wörter, die das Teilwort 11 enthalten.
- Formal ist der Automat aber gar kein DFA: Im Zustand q_a ist nicht klar, was passieren soll, wenn eine 1 gelesen wird. In q_a bleiben? Oder nach q_b wechseln?
- Trotzdem ist irgendwie klar, dass der Automat gerade Worte akzeptiert, die 11 enthalten.

6-5

Die Idee des Nichtdeterminismus.

- Wir werden heute Automaten betrachten, bei denen es zu einem Zustand und einem gelesenen Zeichen *mehrere mögliche Folgezustände* gibt.
- Von diesen aus kann es dann jeweils wieder beim nächsten Zeichen mehrere mögliche Folgezustände geben.
- Damit gibt es dann für den Automaten *viele mögliche Berechnungswege*.
- Diese Vielfalt an möglichen Berechnungen nennt man *Nichtdeterminismus*.
- Manche Berechnungen werden akzeptierend sein, andere nichtakzeptierend. Man muss dann regeln, wie man das interpretiert.

6-6

Einige Legenden über Nichtdeterminismus.

Die Legende »Nichtdeterminismus« bedeutet, dass man nicht vorhersagen kann, was passiert.

Die Fakten Auch bei einem nichtdeterministischen Automat ist klar festgelegt, was genau bei einer Eingabe passieren kann – es sind eben nur viele unterschiedliche Dinge.

Die Legende »Nichtdeterminismus« bedeutet, dass die Berechnung zufällig ist.

Die Fakten Nichtdeterminismus hat nichts mit Zufall zu tun. Gibt es zwei Nachfolgezustände, so sind einfach beide *möglich*; keiner ist *wahrscheinlicher* als der andere.

Die Legende Nichtdeterministische Automaten »probieren viele Möglichkeiten durch«.

Die Fakten Zu einer konkreten Eingabe gibt es bei einem nichtdeterministischen Automaten viele mögliche Berechnungen. In jeder einzelnen wird aber nur einmal das Wort gelesen, dann ist Schluss.

6.1.2 Syntax

6-7

Formale Definition von nichtdeterministischen endlichen Automaten.

▶ **Definition:** Syntax von NFAs

Ein *nichtdeterministischer endlicher Automat M* besteht aus

1. einem *Eingabealphabet* Σ,
2. einer *endliche Menge Q von Zuständen*,
3. einem *Startzustand* $q_0 \in Q$,
4. einer Menge $Q_a \subseteq Q$ von *akzeptierenden Zuständen* und
5. einer *Zustandsübergangsrelation*

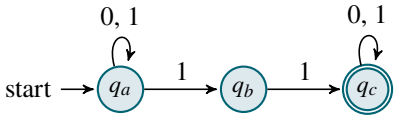
$$\Delta \subseteq (Q \times \Sigma) \times Q.$$

Beispiel eines NFAs. 6-8

Beispiel: Ganz formale Weise, einen Automaten aufzuschreiben
Sei $M = (\Sigma, Q, q_0, Q_a, \Delta)$ mit:

- $\Sigma = \{0, 1\}$,
- $Q = \{q_a, q_b, q_c\}$,
- $q_0 = q_a$,
- $Q_a = \{q_c\}$,
- und $\Delta = \{((q_a, 0), q_a), ((q_a, 1), q_a), ((q_a, 1), q_b), ((q_b, 1), q_c), ((q_c, 0), q_c), ((q_c, 1), q_c)\}$.

Beispiel: Übliche Schreibweise für denselben Automaten



6.1.3 Semantik

Die Idee hinter der Arbeitsweise eines NFAs. 6-9

- Zu jeder Eingabe kann es *viele unterschiedliche Berechnungen geben*.
- Jede einzelne Berechnung funktioniert aber genau wie bei einem DFA:
  - Der Automat beginnt in q_0.
  - Er liest das erste Zeichen.
  - Er wechselt in einen Zustand, der vom aktuellen Zustand aus für das gelesen Zeichen möglich ist.
  - Dies wiederholt er bis zum Wortende (falls möglich).
  - Ist das Ende erreicht und der letzte Zustand akzeptierend, so heißt die Berechnung akzeptierend.
- Zu einem Eingabewort kann es (muss es aber auch nicht) viele mögliche Berechnungen geben – manche vielleicht akzeptierend, andere nicht.

Goldene Regel
Ein Wort wird *von dem Automaten akzeptiert*, wenn es wenigstens eine akzeptierende Berechnung gibt.

Semantik eines NFAs. 6-10
Konfigurationen und Berechnungsschritten

▶ Definition: Konfiguration
Sei $M = (\Sigma, Q, q_0, Q_a, \Delta)$ ein NFA. Wir nennen ein Paar $(q, w) \in Q \times \Sigma^*$ eine *Konfiguration* des Automaten.
Die *Anfangskonfiguration* (auch *Initialkonfiguration*) bei Eingabe w ist (q_0, w).

Erläuterung:

- Der Begriff der Konfiguration wird uns später noch häufiger begegnen. Allgemein ist eine Konfiguration einer Maschine ein »Schnappschuss« der Berechnung zu einem bestimmten Zeitpunkt.
- Bei einem Automaten besteht dieser Schnappschuss aus
  1. dem aktuellen Zustand q, in dem sich der Automat befindet und
  2. dem *noch nicht verarbeiteten Restwort* w.
- Die *Anfangskonfiguration* ist ein Schnappschuss vom Anfang der Berechnung.
- Ist die Eingabe komplett gelesen, so erreicht man eine Konfiguration der Form (q, λ).

▶ Definition: Berechnungsschritt
Sei $M = (\Sigma, Q, q_0, Q_a, \Delta)$ ein NFA. Seien (q, w) und (q', w') zwei Konfigurationen. Wir sagen, dass M *in einem Berechnungsschritt von* (q, w) *zu* (q', w') *kommen kann*, wenn

1. $w = xw'$ für ein $x \in \Sigma$ gilt und
2. $\big((q, x), q'\big) \in \Delta$.

Wir schreiben dann $(q, w) \vdash_M (q', w')$.

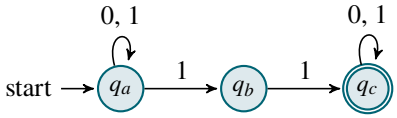
Erläuterung:

- Ist (q, w) die aktuelle Konfiguration eines Automaten, so betrachtet dieser das erste Zeichen von w, also x.
- Nun kann er in alle Zustände q' wechseln, die Δ für dieses Zeichen und den Zustand q erlaubt.
- Das Zeichen x ist dann verarbeitet und taucht in der Folgekonfiguration nicht mehr auf.
- Bei nichtdeterministischen Automaten kann es mehrere mögliche Folgekonfigurationen geben.
- Der Index bei $\vdash_M$ wird gerne weggelassen.

6-11

Beispiele von Berechnungsschritten.

Sei M wieder der Automat



Dann gilt

$$
\begin{aligned}
(q_a, 1010) &\vdash (q_a, 010),\\
(q_a, 1010) &\vdash (q_b, 010),\\
(q_b, 11) &\vdash (q_c, 1),\\
(q_c, 0) &\vdash (q_c, \lambda),\\
(q_b, 1) &\nvdash (q_b, \lambda),\\
(q_b, 11) &\nvdash (q_a, 1),\\
(q_c, 0) &\nvdash (q_c, 0).
\end{aligned}
$$

6-12

Semantik eines NFAs.

Berechnungen und akzeptierte Sprache.

▶ Definition: Berechnungskette
Sei M ein NFA. Eine *Berechnungskette* oder kurz eine *Berechnung* ist eine Folge von Konfigurationen $(q_1, w_1), \ldots, (q_n, w_n)$, so dass

$$(q_1, w_1) \vdash (q_2, w_2) \vdash \cdots \vdash (q_n, w_n)$$

gilt. Wir schreiben dann $(q_1, w_1) \vdash^* (q_n, w_n)$.
Wie vereinbaren, dass auch $(q, w) \vdash^* (q, w)$ gilt (gewissermaßen via einer Kette der Länge 0.)

Erläuterungen:

- Diese Definition ist *sehr* ähnlich zur Definition einer Ableitungskette bei Grammatiken.
- So wie $\Rightarrow$ eine Relation auf Worten ist, ist $\vdash$ eine Relation auf Konfigurationen.
- So wie $\Rightarrow^*$ der reflexive, transitive Abschluss von $\Rightarrow$ ist, ist $\vdash^*$ der reflexive, transitive Abschluss von $\vdash$.

▶ Definition: Akzeptierende Berechnung
Sei M ein NFA. Ein *akzeptierende Berechnung für ein Wort* $w \in \Sigma^*$ ist eine Berechnung, die

1. mit der Anfangskonfiguration (q_0, w) für w beginnt und
2. mit einer Konfiguration (q, λ) endet für ein $q \in Q_a$.

Erläuterungen:

- Eine akzeptierende Berechnung beginnt im Startzustand.
- Das Eingabewort muss komplett gelesen werden.
- Der letzte erreichte Zustand muss ein akzeptierender Zustand sein.

▶ Definition: Akzeptierte Sprache

Sei M ein NFA. Wir sagen, *M akzeptiert ein Wort $w \in \Sigma^*$*, falls es eine akzeptierende Berechnung für w gibt. Die Menge aller von M akzeptierten Worte bezeichnen wir mit $L(M)$:

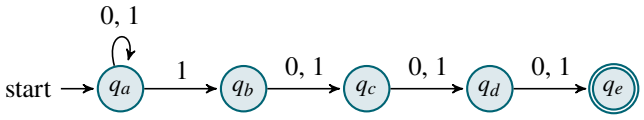
$$L(M) = \{w \in \Sigma^* \mid \text{es gibt ein } q \in Q_a \text{ mit } (q_0, w) \vdash^* (q, \lambda)\}.$$

Erläuterungen:

- Damit ein Wort akzeptiert wird, reicht es, dass es wenigstens eine akzeptierende Berechnung gibt.
- Sind alle möglichen Berechnungen für ein Wort nichtakzeptierend, so ist das Wort kein Element der akzeptierten Sprache.

Beispiel 6-13

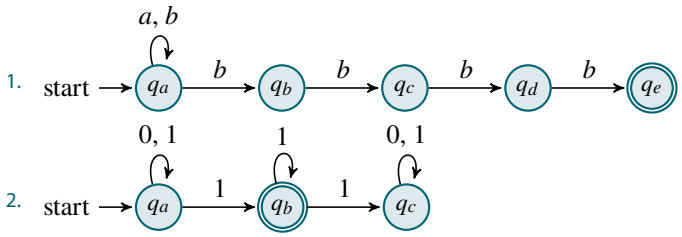
Der Automat



akzeptiert die Sprache $\{w \in \{0,1\}^* \mid w[|w|-3] = 1\}$.

✎ Zur Übung 6-14

Welche Sprachen akzeptieren die folgenden Automaten?



6.1.4 Korrektheitsbeweise

Beweisrezept: *Korrektheitsbeweis für NFAs* 6-15

Ziel

Man will beweisen, dass ein NFA M eine Sprache L akzeptiert.

Rezept

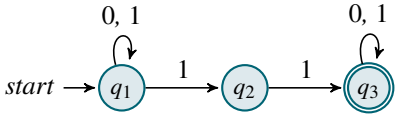
Zeige zwei Richtungen (Beweisrezept »Zwei Richtungen«):

1. Beginne mit »Sei $w \in L$ beliebig.« (Beweisrezept »All-Aussagen beweisen«). Gib dann die Folge von Konfigurationen an, die M bei Eingabe w durchläuft (Beweisrezepte »Konstruktiver Beweis«). Ende mit »Also gilt $(q_0, w) \vdash^* (q_n, \lambda)$ mit $q_n \in Q_a$ und somit $L \subseteq L(M)$«.
2. Beginne mit »Sei $w \in L(M)$ beliebig.« (Beweisrezept »All-Aussagen beweisen«.) Fahre fort mit »Dann gilt $(q_0, w) \vdash \cdots \vdash (q_n, \lambda)$ mit $q_n \in Q_a$.« (Beweisrezept »Namen vergeben«.) Argumentiere dann, dass $w \in L$ gilt. Ende mit »Also gilt $w \in L$ und somit $L(M) \subseteq L$«.

Kurzes Beispiel eines Korrektheitsbeweises. 6-16

▶ Satz

Der Automat



akzeptiert die Sprache $\{u11v \mid u, v \in \{0,1\}^*\}$.

Beweis. Sei M der Automat und $L = \{u11v \mid u,v \in \{0,1\}^*\}$.[1] Wir zeigen zwei Richtungen. Sei zunächst $w \in L$ beliebig.[2] Dann gilt $w = u11v$ mit $u,v \in \{0,1\}^*$.[3] Dieses Wort kann M durch folgende Berechnung akzeptieren:

$$(q_1, u11v) \vdash^* (q_1, 11v) \vdash (q_2, 1v) \vdash (q_3, v) \vdash^* (q_3, \lambda).$$

Da $q_3 \in Q_a$, wird $u11v$ akzeptiert und $w \in L(M)$.
Für die andere Richtung sei $w \in L(M)$. Damit M ein Wort akzeptiert, muss die Berechnung in q_3 enden und q_2 durchlaufen. Dann muss aber beim Übergang von q_1 zu q_2 eine 1 gelesen worden sein. Ebenso direkt danach beim Übergang von q_2 zu q_3. Also enthält w das Teilwort 11. Also gilt $w \in L$. □

Kommentar
[1] Erstmal Na
[2] Text aus d
[3] Aufgrund

6.2 Umwandlung in deterministische Automaten

6.2.1 Die Idee

6-17

Zum Verhältnis von DFAs und NFAs.

- Nichtdeterministische Automaten sind auf den ersten Blick viel mächtiger: Statt einer mickrigen möglichen Berechnung gibt es plötzlich einen exponentiell großen Baum an Berechnungen.
- Andererseits haben wir noch kein Beispiel einer Sprache gesehen, die sich nur von NFAs akzeptieren lässt.
- Das ist auch kein Zufall, denn NFAs *können nicht mehr als* DFAs*!*

6-18

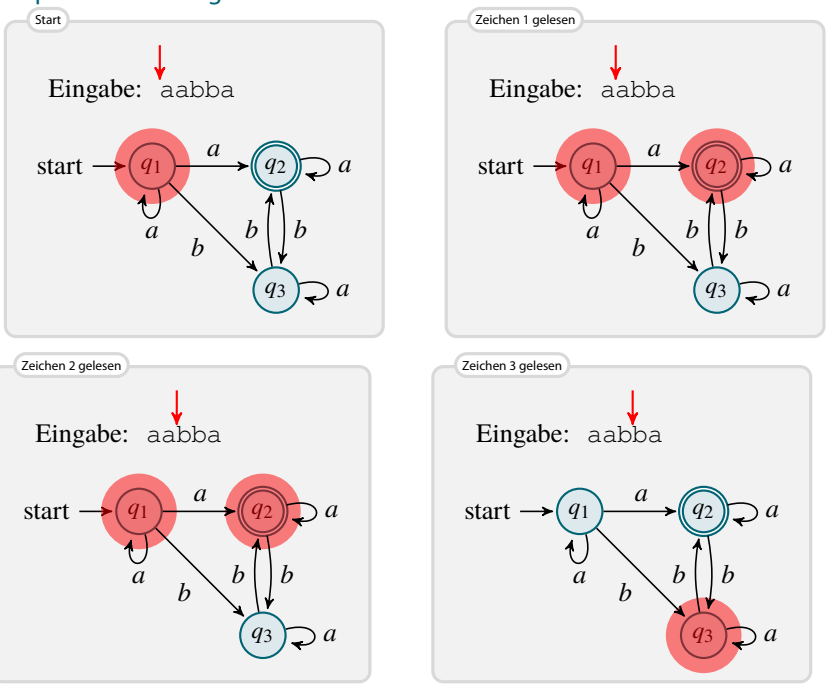
Ideen hinter der Umwandlung von NFAs in DFAs.
Sei M ein NFA. Wir suchen einen *äquivalenten* DFA M' (»äquivalent« bedeutet, dass er dieselbe Sprache akzeptiert).

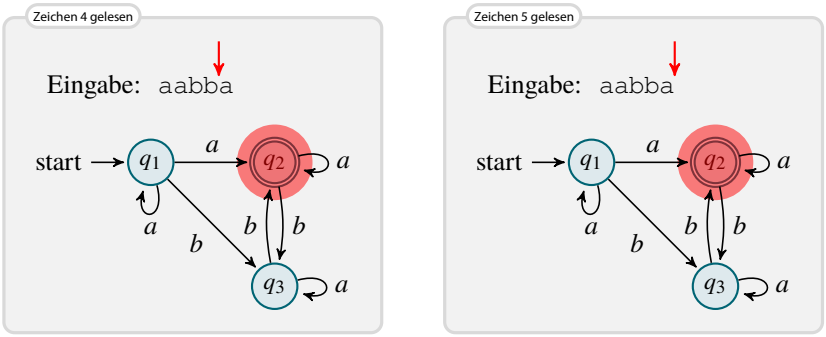
Ideen

- Der DFA versucht für jede Stelle im Wort herauszufinden, *in welchen Zuständen der* NFA *sein könnte, wenn er diese Stelle erreicht*.
- Ganz am Anfang ist dies nur q_0.
- Wenn ganz am Ende ein Zustand aus Q_a dabei ist, dann wird der NFA das Wort akzeptieren, sonst eben nicht.

6-19

Beispiel für die Menge der erreichbaren Zustände.





Die Idee der Powerzustände.

6-20

- Eine »Menge von Zuständen, in denen der NFA sein könnte« nennen wir einen *Power-Zustand*.
- Formal ist ein Power-Zustand einfach eine Teilmenge der Zustandsmenge Q.
- Der gesuchte DFA versucht nun, einfach die *Folge der Power-Zustände* zu ermitteln, die sich aus den Berechnungen des NFA ergeben.

6.2.2 Die Konstruktion

Wie wandelt man NFAs in DFAs um?

6-21

▶ **Definition**

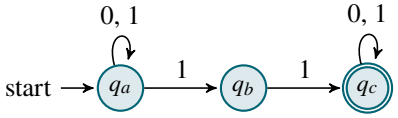
Sei $M = (\Sigma, Q, q_0, Q_a, \Delta)$ ein NFA. Der *Potenzmengenautomat zu M* ist der folgende DFA $M' = (\Sigma', Q', q_0', Q_a', \delta')$:

1. $\Sigma' = \Sigma$. Das Eingabealphabet bleibt also gleich.
2. $Q' = \{P \mid P \subseteq Q\}$. Dies ist die Potenzmenge von Q. Die Elemente von Q' nennen wir auch *Power-Zustände*.
3. $q_0' = \{q_0\}$. Der Power-Startzustand enthält einfach nur den Startzustand von M.
4. $Q_a' = \{P \in Q' \mid P \cap Q_a \neq \emptyset\}$. Alle Power-Zustände P sind akzeptierend, die wenigstens einen akzeptierenden Zustand von M enthalten.
5. $\delta'(P,x) = \{p \mid \text{es gibt ein } q \in P \text{ mit } (q,x) \vdash^* (p,\lambda)\}$. Dies ist der Power-Zustand, der alle Zustände enthält, die M in einem Schritt erreichen kann, wenn es ein x liest und in einem der Zustände aus P war.

Ein Beispiel eines Potenzmengenautomaten.

6-22

Sei M der folgende DFA:



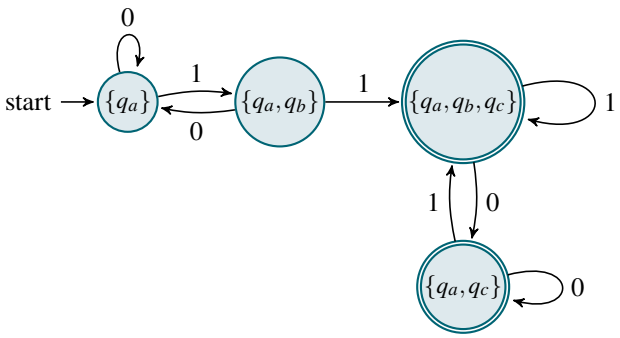
Dann ist der Potenzmengenautomat $M' = (\Sigma', Q', q_0', Q_a', \Delta')$ wie folgt definiert:

- $\Sigma' = \{0,1\}$.
- $Q' = \{\emptyset, \{q_a\}, \{q_b\}, \{q_c\}, \{q_a,q_b\}, \{q_a,q_c\}, \{q_b,q_c\}, \{q_a,q_b,q_c\}\}$.
- $q_0' = \{q_a\}$.
- $Q_a' = \{\{q_c\}, \{q_a,q_c\}, \{q_b,q_c\}, \{q_a,q_b,q_c\}\}$.

Die Werte von $\delta'(P,x)$:

alter Power-Zustand P	$\delta'(P,0)$	$\delta'(P,1)$
$\emptyset$	$\emptyset$	$\emptyset$
$\{q_a\}$	$\{q_a\}$	$\{q_a,q_b\}$
$\{q_b\}$	$\emptyset$	$\{q_c\}$
$\{q_c\}$	$\{q_c\}$	$\{q_c\}$
$\{q_a,q_b\}$	$\{q_a\}$	$\{q_a,q_b,q_c\}$
$\{q_a,q_c\}$	$\{q_a,q_c\}$	$\{q_a,q_b,q_c\}$
$\{q_b,q_c\}$	$\{q_c\}$	$\{q_c\}$
$\{q_a,q_b,q_c\}$	$\{q_a,q_c\}$	$\{q_a,q_b,q_c\}$

Als Graph aufgeschrieben ohne die unerreichbaren Zustände ergibt sich:



6-23

Der Potenzmengenautomat -- Schritt für Schritt.

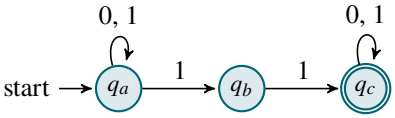
Gegeben sei ein NFA M. Den Potenzmengenautomaten konstruiert man am einfachsten wie folgt:

- Man erstellt eine Tabelle, die wie folgt aufgebaut ist:
  - In jeder Zeile steht ein Power-Zustand in der ersten Spalte.
  - In den folgenden Spalten stehen dann jeweils die Nachfolge-Power-Zustände für die verschiedenen möglichen gelesenen Zeichen.
- Am Anfang trägt man nur in der ersten Zeile $\{q_0\}$ ein.
- Dann schreibt man in die folgenden Spalte, welche Zustände sich bei den verschiedenen gelesenen Zeichen in einem Schritt erreichen lassen.
- Tauchen dabei Power-Zustände auf, die man noch nicht hatte, werden dafür neue Zeilen eröffnet.
- Man endet, wenn alle erreichbaren Powerzustände aufgelistet sind.

6-24

Nochmal das Beispiel, diesmal konstruktiv.

Sei M der folgende DFA:



Die Werte von $\delta'(P,x)$:

alter Power-Zustand P	$\delta'(P,0)$	$\delta'(P,1)$
$\{q_a\}$	$\{q_a\}$	$\{q_a,q_b\}$
$\{q_a,q_b\}$	$\{q_a\}$	$\{q_a,q_b,q_c\}$
$\{q_a,q_b,q_c\}$	$\{q_a,q_c\}$	$\{q_a,q_b,q_c\}$
$\{q_a,q_c\}$	$\{q_a,q_c\}$	$\{q_a,q_b,q_c\}$

6.2.3 Der Satz

6-25

NFAs sind nicht mächtiger als DFAs.

▶ **Satz**
Zu jedem NFA M *existiert ein* DFA M' *mit* $L(M) = L(M')$.

Beweisidee

- Der gesuchte Automat M' ist gerade der Potenzmengenautomat.
- Man zeigt durch Induktion über die Wortlänge, dass für jedes Wort die Menge von M am Wortende erreichbaren Zustände genau gleich dem von M' erreichten Power-Zustand ist.
- Daraus folgt dann ziemlich direkt die Behauptung.

Beweis. Sei $M = (\Sigma, Q, q_0, Q_a, \Delta)$ und $M' = (\Sigma', Q', q_0', Q_a', \Delta')$ der zugehörige Potenzmengenautomat.[1] Offenbar ist M' ein DFA. Es bleibt zu zeigen, dass $L(M) = L(M')$.
Wir zeigen zunächst nun folgendes:[2] Für alle Power-Zustände $P \in Q'$ und alle Worte $w \in \Sigma^*$ gilt $\delta'^*(P_0, w) = \{q \mid (q_0, w) \vdash^* (q, \lambda)\}$. Wir beweisen dies durch Induktion über die Länge von w. Für

Kommentare zum Beweis

Skript [1] Rezept »Konstruktiver Beweis«, die Konstruktion ist schon gemacht.

[2] Ein Einschub. Hier wird behauptet, dass Power-Zustände genau die Menge der erreichbaren Zustände enthalten.

$w = \lambda$ ist $\delta'^*(P_0, \lambda) = P_0 = \{q_0\}$ und die Behauptung stimmt. Für den induktiven Schritt sei $w = ux$ mit $x \in \Sigma$. Nach Voraussetzung ist dann $\delta'^*(P_0, u) = \{q \mid (q_0, u) \vdash^* (q, \lambda)\}$. Nennen wir diese Menge P, so ist $P = \{q \mid (q_0, ux) \vdash^* (q, x)\}$.[3]
Nach Definition von δ' ist $\delta'(P, x) = \{p \mid$ es gibt ein $q \in P$ mit $(q, x) \vdash^* (p, \lambda)\}$. Nach Induktionsvoraussetzung gilt $q \in P$ genau dann, wenn $(q_0, ux) \vdash^* (q, x)$. Also ist $p \in \delta'(P, x)$ genau dann, wenn $(q_0, ux) \vdash^* (q, x) \vdash^* (p, \lambda)$. Wir erhalten somit, dass $\delta'(P, x) = \delta'^*(P_0, w)$ gerade der Power-Zustand aller von q_0 aus erreichbaren Zustände ist, womit die Induktion abgeschlossen ist.[4]
Damit ist gezeigt, dass $\delta'^*(P_0, w) \in Q'_a$ genau dann gilt, wenn $\delta'^*(P_0, w)$ einen Zustand $q \in Q_a$ enthält. Andererseits enthält $\delta'^*(P_0, w)$ genau alle q mit $(q_0, w) \vdash^* (q, \lambda)$. Also akzeptiert M' das Wort w genau dann, wenn M eine akzeptierende Berechnung für w besitzt, was genau dann der Fall ist, wenn M das Wort w akzeptiert. Dies bedeutet $L(M) = L(M')$. □

[4] Hier ist der Einschub zu ende.

Zusammenfassung dieses Kapitels

6-26

1. Bei nichtdeterministische endliche Automaten kann es zu einem Zustand und einem gelesenen Zeichen *mehrere mögliche Folgezustände geben*.
2. Ein NFA akzeptiert ein Wort, wenn es wenigstens eine akzeptierende Berechnung gibt.
3. Der *Potenzmengenautomat* eines NFAs ist ein DFA, der dieselbe Sprache akzeptiert.

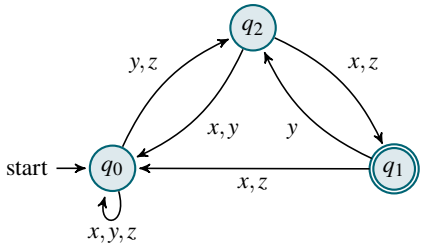
Zum Weiterlesen

[1] R. Reischuk. *Unterlagen zum Lehrmodul Theoretische Informatik,* Vorlesungsskript, 2008. Kapitel 1.4.

Übungen zu diesem Kapitel

Übung 6.1 Umwandlung von NFA in DFA, mittel

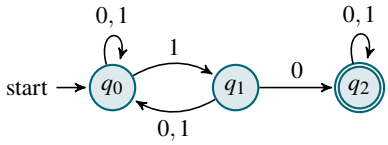
Gegeben sei der folgende nichtdeterministische Automat M mit Alphabet $\Sigma = \{x, y, z\}$:



Konstruieren Sie den Potenzmengen-Automaten zu M und geben Sie $L(M)$ an.

Übung 6.2 Umwandlung von NFA in DFA, mittel

Gegeben sei der folgende nichtdeterministische Automat M mit Alphabet $\Sigma = \{0, 1\}$:



Konstruieren Sie den Potenzmengen-Automaten zu M und geben Sie $L(M)$ an.

7-1

Kapitel 7
Zwei-Wege-Automaten
Warum es nichts bringt, vorne nochmal nachzuschauen

7-2

Lernziele dieses Kapitels

1. Syntax und Semantik von 2-NFAs beherrschen
2. Äquivalenzbeweis zu DFAs verstehen
3. Äquivalenzbeweis zu regulären Grammatiken verstehen

Inhalte dieses Kapitels

Worum es heute geht

Endliche Automaten machen immer einen etwas gehetzten Eindruck, wenn sie am Wortende angekommen sind – ihnen sind einfach keine Ruhepausen oder kleine Picknicks gestattet, während sie ihre Eingabeworte abarbeiten. In jedem Schritt muss immer ein Zeichen verarbeitet werden, egal wie müde der Automat ist und egal wie sehr er doch eigentlich zu seinem Anfangszustand zurückkehren möchte, um ein kleines Nickerchen zu halten.

In diesem Kapitel soll es um Automaten gehen, die sich wesentlich entspannter der Aufgabe widmen, ihre Eingabeworte zu untersuchen. Zunächst werden wir Automaten begegnen, die sich dem Werbespruch verschrieben haben *»Mach' mal Pause.«* Diese Automaten können nämlich fröhlich ihren Zustand wechseln, ohne sich überhaupt »vom Fleck zu bewegen«. Dies führt zu einem wesentlich gemütlicheren Tempo, mit dem sie ihre Eingaben verarbeiten. Diese Automaten nennt man »λ-NFAs«, wobei das »λ« für »langsam« stehen könnte (in Wirklichkeit steht es für »Leerwort-Zustandwechsel«, aber das klingt doch irgendwie schrecklich technisch). Es dürfte allerdings wenig überraschen, dass solch gemütlich vorgehenden NFAs auch nicht mehr können als ihre gehetzteren Verwandten.

Wirklich spannend wird es bei den 2-Wege-Automaten. Diese können nämlich auch auf dem Wort wieder *zurücklaufen*, um sich Teile des Wortes weiter vorne nochmal anzuschauen. Wenn ein solcher Automat zum Beispiel am Ende feststellt, dass dort ein Deckelchen ist, dann kann er kurz nochmal nach vorne huschen, um zu überprüfen, ob dort auch tatsächlich das passende Töpfchen war. Dort darf er dann wieder seine Richtung wechseln und nach Fischen und Fahrrädern Ausschau halten.

Intuitiv sind 2-Wege-Automaten deutlich mächtiger als 1-Wege-Automaten. Anderseits erschienen ja auch nichtdeterministische endliche Automaten deutlich mächtiger als deterministische – nur um sich als gleichmächtig herauszustellen. Verblüfft es also, dass 2-Wege-Automaten auch nicht mächtiger sind als 1-Wege-Automaten? Statt Power-Zuständen ist der Trick diesmal, sich am Anfang schonmal alles zu merken, »was der Automat später vielleicht nachschauen könnte«.

7.1 λ-NFAs

Erste Erweiterung von NFAs. 7-4

- Sowohl DFAs wie auch NFAs müssen in jedem Schritt ein Zeichen lesen.
- Man kann sich fragen, ob man nicht vielleicht auch *mehrere Zeichen* in einem Schritt lesen könnte.
- Dabei stellt man aber schnell fest, dass dies nicht wirklich nützlich ist: Will man »drei Zeichen auf einmal lesen«, so kann man genausogut auch einfach zwei »Zwischenzustände« einführen.
- Anders liegen die Dinge, wenn man in einem Schritt »gar nichts« lesen möchte.
- Einen solchen Zustandswechsel ohne gelesenes Zeichen bezeichnet man auch als λ-*Schritt* oder auch als ε-*Schritt*.
- Die entsprechenden Automaten heißen λ-NFA*s*.

7.1.1 Syntax

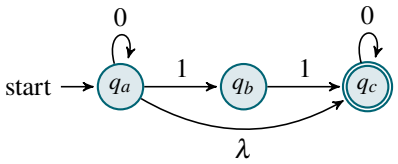
Syntax von λ-NFAs. 7-5

▶ Definition: Syntax von λ-NFA
Ein λ-NFA ist ein Tupel $M = (\Sigma, Q, q_0, Q_a, \Delta)$, das genau wie ein NFA aufgebaut ist, mit folgender Änderung:

- Für die *Zustandsübergangsrelation* gilt

$$\Delta \subseteq \big(Q \times (\Sigma \cup \{\lambda\}\big) \times Q.$$

Beispiel



7.1.2 Semantik

Semantik von λ-NFAs. 7-6

▶ Definition: Berechnungen von λ-NFAs, akzeptierte Sprache
Sei M ein λ-NFA. Dann ist die Berechnungsrelation $\vdash$ genauso definiert wie bei normalen NFAs, außer dass für $((q,\lambda),q') \in \Delta$ für alle $w \in \Sigma^*$ auch gilt

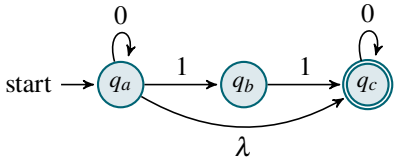
$$(q,w) \vdash (q',w).$$

Ansonsten sind die Begriffe »akzeptierende Berechnung«, »akzeptiertes Wort« und »akzeptierte Sprache« genauso definiert wie bei NFAs.

7-7

Beispiele von ε-NFAs.

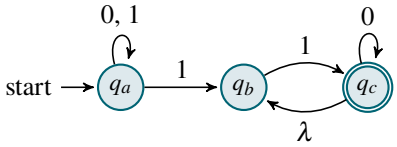
Der Automat



akzeptiert die Sprache $\{0^n \mid n \geq 0\} \cup \{0^n 110^m \mid n, m \geq 0\}$.

✎ Zur Übung

Welche Sprache akzeptiert folgender Automat?



7.1.3 Mächtigkeit

7-8

Was können λ-NFAs?

- λ-NFAs sind eine moderate Erweiterung von NFAs.
- Da NFAs nicht mehr können als DFAs, wäre es überraschend, wenn λ-NFAs dies könnten.
- In der Tat können λ-NFAs auch nicht mehr als DFAs.

▶ Satz

Zu jedem λ-NFA M gibt es einen DFA M' mit $L(M) = L(M')$.

Beweis. Erstaunlicherweise sind der Beweis und die Konstruktion des Potenzmengenautomaten wortwörtlich identisch zum Beweis für NFAs, siehe Satz 6-25. □

7.2 2-Wege-NFAs

7-9

Zweite Erweiterung von NFAs.

- Sowohl DFAs als auch NFAs dürfen die Eingabe nur »in eine Richtung lesen«.
- Bei einem *2-Wege-Automaten* kann sich der Automat nach dem Lesen eines Zeichens *aussuchen*, ob es mit dem nächsten oder mit dem vorherigen Zeichen weitergeht.
- Man könnte noch zulassen, dass der Automat auch stehen bleiben darf, was *grob einem λ-Schritt entspräche*.
  Jedoch kann ein 2-Wege-Automat das »Stehenbleiben« leicht durch einen »Rechts-Links-Schritt« simulieren.

7-10

Das Randproblem.

- Ein 1-Wege-Automat akzeptiert ein Wort, wenn er »am Ende« in einem akzeptierenden Zustand ist.
- Ein 2-Wege-Automat möchte vielleicht am Wortende »doch noch mal vorne nachschauen«.
- Dazu muss der 2-Wege-Automat aber wissen, wann er das Ende oder den Anfang erreicht hat. Dazu wird der Automat, der eigentlich ein Wort $w \in \Sigma^*$ als Eingabe bekommen soll, stattdessen das Wort $\hat{}w\$$ als Eingabe bekommen.

▶ Definition: Wortrandsymbole

Das *Wortanfangssymbol* $\hat{}$ und das *Wortendesymbol* $\$$ sind zwei Sondersymbole, die normalerweise nicht Alphabeten vorkommen. (Dies ist analog zum Blanksymbol □.)

- Nun muss man noch klären, wann der Automat eigentlich »fertig« ist, schließlich ist das Wortende nicht mehr »besonders«.
- Man definiert, dass der Automat fertig ist, wenn er über das Wortendezeichen hinaus läuft.

7.2.1 Syntax

Formale Definition von 2-Wege-Automaten.

7-11

Definition: Syntax von 2-Wege-NFAs

Ein *2-Wege-*NFA M besteht aus

1. einem *Eingabealphabet* Σ,
2. einer *endliche Menge Q von Zuständen*,
3. einem *Startzustand* $q_0 \in Q$,
4. einer Menge $Q_a \subseteq Q$ von *akzeptierenden Zuständen* und
5. einer *Zustandsübergangsrelation*

$$\Delta \subseteq \big(Q \times (\Sigma \cup \{\text{^}, \$\})\big) \times (Q \times \{l, r\}).$$

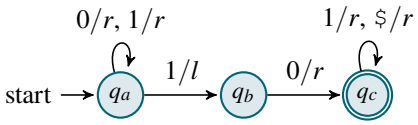
Definition: Syntax von 2-Wege-DFAs

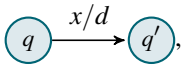
Ein 2-Wege-NFA heißt *deterministisch*, wenn die Relation Δ eine Funktion ist. In diesem Fall schreiben wir statt Δ auch δ.

Beispiel eines 2-Wege-DFAs.

7-12

Beispiel



- Bei einem 2-Wege-Automaten muss man in jedem Zustand zu jedem gelesenen Zustand auch noch angeben, in welche Richtung der Automat läuft.
- Dies geschieht durch die Angabe der Richtung hinter einem Schrägstrich.
- Formal bedeutet 

, dass $\big((q,x),(q',d)\big) \in \Delta$ gilt.

7.2.2 Semantik

Semantik eines 2-Wege-NFAs.

7-13

Konfigurationen

Definition: Konfiguration

Sei $M = (\Sigma, Q, q_0, Q_a, \Delta)$ ein 2-Wege-NFA. Wir nennen ein Paar $(q, w) \in Q \times (\Sigma \cup \{\triangleright, \text{^}, \$\})^*$ eine *Konfiguration* des Automaten, falls

1. w jedes der Zeichen $\triangleright$, ^ und $ genau einmal enthält und
2. wenn man $\triangleright$ entfernt, so steht ^ am Anfang und $ am Ende.

Die *Anfangskonfiguration* bei Eingabe w ist $(q_0, \text{^}\triangleright w\$)$.

Erläuterung:

- Das Symbol $\triangleright$ ist noch so ein Sondersymbol, das (genau wie ^ und $) nicht in Σ vorkommt.
- Es steht direkt vor dem Zeichen, das als nächstes gelesen wird.

Semantik eines 2-Wege-NFAs.

7-14

Berechnungsschritte und Akzeptanz

- Die Relation $\vdash_M$, die die Berechnungsschritte des Automaten beschreibt, muss natürlich entsprechend angepasst werden.
- Ebenso muss man neu definieren, welche Berechnungen akzeptierend sind: Es sind solche, die auf $(q, \text{^}w\$\triangleright)$ enden mit $q \in Q_a$.

Skript-Referenz

▶ **Definition:** Berechnungsschritt
Sei $M = (\Sigma, Q, q_0, Q_a, \Delta)$ ein 2-Wege-NFA. Sei $(q, ux \triangleright yv)$ eine Konfiguration mit $u, v \in (\Sigma \cup \{\hat{}, \$\})^*$ und $x, y \in \Sigma \cup \{\hat{}, \$\}$.
Wir definieren nun die Relation $\vdash_M$ wie folgt:

1. Falls $\big((q, y), (q', r)\big) \in \Delta$, so gilt $(q, ux \triangleright yv) \vdash_M (q', uxy \triangleright v)$. Zusätzlich ist hier auch der Fall $x = \lambda$ möglich.
2. Falls $\big((q, y), (q', l)\big) \in \Delta$, so gilt $(q, ux \triangleright yv) \vdash_M (q', u \triangleright xyv)$.

▶ **Definition:** Akzeptierende Berechnung
Sei M ein 2-Wege-NFA. Ein *akzeptierende Berechnung für ein Wort* $w \in \Sigma^*$ ist eine Berechnung, die

1. mit der Anfangskonfiguration $(q_0, \hat{} \triangleright w\$)$ für w beginnt und
2. mit einer Konfiguration $(q, \hat{} w\$ \triangleright)$ endet für ein $q \in Q_a$.

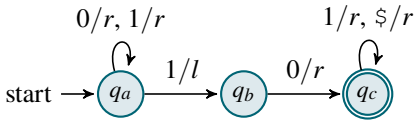
Die Definitionen akzeptierter Wörter und der akzeptierten Sprache ist wie bei NFAs, also wie folgt:

▶ **Definition:** Akzeptierte Sprache
Sei M ein 2-Wege-NFA. Wir sagen, M *akzeptiert ein Wort* $w \in \Sigma^*$, falls es eine akzeptierende Berechnung für w gibt. Die Menge aller von M akzeptierten Worte bezeichnen wir mit $L(M)$.

7-15

Beispiel einer Berechnung.

Sei M folgender 2-Wege-Automat:



Dann gilt:

$$\begin{aligned}(q_a, \hat{} \triangleright 001\$) &\vdash_M (q_a, \hat{} 0 \triangleright 01\$)\\ &\vdash_M (q_a, \hat{} 00 \triangleright 1\$)\\ &\vdash_M (q_b, \hat{} 0 \triangleright 01\$)\\ &\vdash_M (q_c, \hat{} 00 \triangleright 1\$)\\ &\vdash_M (q_c, \hat{} 001 \triangleright \$)\\ &\vdash_M (q_c, \hat{} 001\$ \triangleright)\end{aligned}$$

Hingegen gilt beispielsweise $(q_a, \hat{} \triangleright 001\$) \nvdash (q_a, \triangleright \hat{} 001\$)$.

✎ Zur Übung
Wie viele akzeptierende Berechnung gibt es für $w = 0101$?

7.2.3 Mächtigkeit

7-16

2-Wege-NFAs können nicht mehr als DFAs.

- Auf den ersten Blick sind 2-Wege-NFAs viel mächtiger als DFAs.
- Jedoch schienen auch schon normale NFAs viel mächtiger als DFAs.
- Es stellt sich heraus, dass *2-Wege-NFAs nicht mehr können als DFAs*.
- Der Beweis benutzt *nicht Power-Zustände*, sondern *Crossing-Patterns*.

7-17

Die Umwandlung von 2-Wege-NFAs in DFAs.

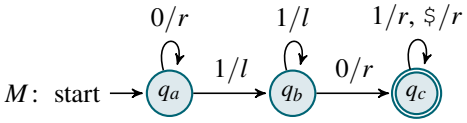
Zur Vereinfachung betrachten wir zunächst nur *deterministische* 2-Wege-Automaten.

Wie ein 1-Wege-DFA einen 2-Wege-DFA simuliert.

- Nehmen wir an, der 2-Wege-Automat läuft zunächst eine Weile nach rechts.
- Dies kann der 1-Wege-Automat direkt simulieren.
- Dann macht der 2-Wege-Automat aber eine Reihe an Schritten nach links und kommt irgendwann wieder zurück.
- Jetzt ist der 1-Wege-Automat erstmal aufgeschmissen.
- *Aber:* Der 1-Wege-Automat kann – extra für diesen Fall – eine Liste mitführen, in der gespeichert ist, in welchem Zustand der 2-Wege-Automat wieder zurückkommen *würde*.
- Dadurch kann sich der 1-Wege-Automat es *sparen*, die Schritte nach links überhaupt durchzuführen.

Beispiel zur Umwandlung eines 2-Wege-DFAs.

7-18



Die Simulation durch einen 1-Wege-Automaten M' bei Eingabe 0011 läuft wie folgt ab:

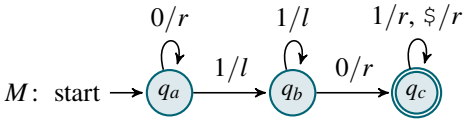
1. M' startet im Zustand q_a. Das Restwort ist 0011.
2. Nun würde M einen Schritt nach rechts machen. Das macht M' auch, bleibt im Zustand q_a. Das Restwort ist 011.
3. Genau wie eben bleibt M' im Zustand q_a. Das Restwort ist 11.
4. *Nun würde M einen Schritt nach links machen und nach q_b wechseln.*
   - Das kann M' nicht simulieren.
   - *Seine Liste sagt M' aber, dass M im Zustand q_c zurückkommen wird und dann die 1 liest und in q_c bleibt.*
   - Also wechselt M' nun nach q_c. Das Restwort ist 1.
5. Dann macht M noch zwei Schritte nach rechts, die M' einfach simuliert und auch in q_c endet und akzeptiert.

Die Idee hinter einem Crossing-Pattern.

7-19

Gegeben seien ein 2-Wege-DFA M und ein Wort w. Wir wollen M durch einen 1-Wege-DFA M' simulieren.

- Betrachten wir eine Stelle zwischen zwei Symbolen von w, sagen wir zwischen $w[i-1]$ und $w[i]$.
- Für die Simulation müssen wir nun Folgendes wissen:
  Wenn M' die Stelle nach links überschreitet im Zustand q, in welchem Zustand q' kommt M' dann wieder zu der Stelle zurück?
- Das Überschreiten einer Stelle im Wort w nennen wir auch ein »Crossing«, das obige »Verhalten« des Automaten nennen wir ein »Crossing-Pattern«.



Die Crossing-Pattern für $w = 0011$:

Zustand bei Rückschritt von $i = 2$	Zustand bei Rückkehr
q_a	q_a
q_b	q_c
q_c	–

Zustand bei Rückschritt von $i = 3$	Zustand bei Rückkehr
q_a	q_a
q_b	q_c
q_c	–

Zustand bei Rückschritt von $i = 4$	Zustand bei Rückkehr
q_a	q_c
q_b	q_c
q_c	q_c

Skript-Referenz

▶ **Definition:** Zustandszuordnung
Eine *deterministische Zustandszuordnung* ist eine Funktion $m\colon Q \to Q \cup \{\bot\}$.

▶ **Definition:** Crossing-Pattern
Wie nennen eine Zustandszuordnung m das *Crossing-Pattern* des 2-Wege-DFA M für ein Wort w und eine Stelle i, wenn für alle Zustände $q \in Q$ gilt:

- Es gilt $m(q) = q'$ mit

$$(q, \hat{}w[1]\dots w[i-2] \triangleright w[i-1]\dots w[|w|]\$) \vdash_M^* (q', \hat{}w[1]\dots w[i-2] \triangleright w[i-1]\dots w[|w|]\$),$$

  wobei in obiger Berechnung das Zeichen $w[i-1]$ nie überschritten wird.
- Falls $m(q) = \bot$, so darf es kein solches q' geben. (Der Automat kehrt nie zurück.)

7-20

Die zentrale Beobachtung.

- Der 1-Wege-DFA »merkt sich in seinem Zustand« das aktuelle Crossing-Pattern:
  - Er hat also grob $|Q|^{|Q|}$ Zustände, eines für jedes mögliche Crossing-Pattern.
  - Der Zustand, in dem er ist, gibt an, welches Crossing-Pattern für die Stelle i korrekt ist.
- Nun liest er ein Zeichen x.
- *Dann kann er das Crossing-Pattern für die Stelle $i+1$ ausrechnen und in den entsprechenden Zustand wechseln.*

Dies geht in etwa wie folgt:

- Nehmen wir an, er liest eine 1.
- Nun gehen wir alle Zustände durch, beginnend mit q_a.
- Falls M bei q_a und einer 1 nach rechts geht und nach q' wechselt, steht im Crossing-Pattern q'.
- Falls M bei q_a und einer 1 nach links geht, so sagt einem das Crossing-Pattern vom vorherigen Schritt, in welchem Zustand q' der Automat »wiederkommt«.
  - Falls M dann von q' nach rechts geht und in q'' landet, so steht im Crossing-Pattern q''.
  - Anderenfalls, falls M also auch von q' wieder nach links geht, sagt einem wieder das Crossing-Pattern, in welchem Zustand q'' nun M zurückkommt.
    - Falls M dann von q'' nach rechts geht und in q''' landet, so steht im Crossing-Pattern q'''.
    - Anderenfalls, …

7-21

2-Wege-NFAs sind nicht mächtiger als DFAs.

▶ **Satz**
Zu jedem 2-Wege-NFA M existiert ein DFA M' mit $L(M) = L(M')$.

Beweisidee

- Man beweist die Behauptung zunächst für deterministische Automaten.
- Der Automat M' hat dann $|Q| \cdot |Q|^{|Q|+1}$ Zustände, nämlich einen für jede Kombination aus aktuellem Zustand und möglichem Crossing-Pattern.
- Die Zustandsübergänge von M' sind so gewählt, dass M' jeweils in dem Zustand ist, in dem M bei dem entsprechenden Zeichen bei der ersten Erreichung wäre, zusammen mit dem zugehörigen Crossing-Pattern.
- Für nichtdeterministische Automaten ersetzt man dann alle Zustände durch Power-Zustände.

Skript

Beweis. Sei $M = (\Sigma, Q, q_0, Q_a, \delta)$ zunächst ein 2-Wege-DFA.[1] Der Automat M' hat die Zustandsmenge $Q' = Q \times \{m \mid m\colon Q \to Q \cup \{\bot\}\}$.[2] Die Zustandsübergangsfunktion δ' von M' von ist wie folgt definiert:[3] Sei $(q,m) \in Q'$ und $x \in \Sigma$. Wir definieren nun, wann $\delta'((q,m),x) = (q',m')$ gilt.
Das neue m' wird wie folgt ermitteln: Für jeden Zustand $p \in Q$ betrachten wir $\delta(p,x) = (p',d')$. Falls $d' = r$, so ist $m'(p) = p'$. Anderenfalls, falls also $d' = l$, so betrachten wir $\delta(m(p'),x) = (p'',d'')$. Falls $d'' = r$, so ist $m'(p) = p''$. Anderenfalls, falls also $d'' = l$, so betrachten wir $\delta(m(p''),x) = (p''',d''')$. Dies wiederholen wir so lange, bis entweder $m'(p)$ festgelegt wurde oder eine Endlosschleife resultiert. In letzterm Fall setzten wir $m'(p) = \bot$.
Das neue q' können wir auf $m'(q,x)$ setzen, da es egal ist, ob wir von links oder von rechts auf das Zeichen x laufen.[4] Falls $m'(q,x) = \bot$, so gibt es einfach keinen Nachfolgezustand.
Der Startzustand q'_0 von M' ist (q_0, m_0), wobei m_0 das (feste) Crossing-Pattern vor dem ersten Zeichen ist. Die Menge der akzeptierenden Zustände Q'_a ist die Menge aller (q,m), so dass M beim Erreichen des Wortendezeichens im Zustand q mit Crossing-Pattern m akzeptieren würde.[5]
Man zeigt nun durch Induktion folgendes:[6] Für jedes Wort $w \in \Sigma^*$ und jede Stelle i ist M', wenn er das i-te Zeichen gelesen hat, genau in dem Zustand (q,m), für den

- q der Zustand ist, in dem M zum ersten Mal das i-te Zeichen erreicht, und
- m das Crossing-Pattern zur i-ten Stelle ist.

Um die Konstruktion und den Beweis auf nichtdeterministische Automaten zu erweitern, benutzt man dieselbe Idee wie bei der Umwandlung von NFAs in DFAs:[7] Eine *nichtdeterministische Zustandszuordnung* ist eine Abbildung $m\colon Q \to \{P \mid P \subseteq Q\}$, die für jedes $q \in Q$ einen Power-Zustand $m(q)$ angibt, der alle Zustände enthält, in denen der 2-Wege-NFA zurückkommen *kann*. Entsprechend ist dann ein nichtdeterministisches Crossing-Pattern definiert. Bei dem DFA M' ist dann jeder Zustand (q,m) ein Paar, in dem q ein Power-Zustand von M und m eine nichtdeterministische Zustandszuordnung ist. Per Induktion zeigt man dann: Für jedes Wort $w \in \Sigma^*$ und jede Stelle i ist M', wenn es das i-te Zeichen gelesen hat, genau in dem Zustand (q,m), für den

- q die Menge der Zustände ist, in denen M zum ersten Mal das i-te Zeichen erreichen kann und
- m das nichtdeterministische Crossing-Pattern zur i-ten Stelle ist, also $m(q)$ die Menge alle Zustände ist, in denen M zurückkommen kann, wenn es im Zustand q die i-te Stelle nach links überschreitet. □

Kommentare zum Beweis

[1] Jetzt kommt das Rezept »Konstruktiver Beweis«

[2] Der Automat soll sich ja den aktuellen Zustand und ein Crossing-Pattern merken.

[3] Die folgende Definition beschreibt die Idee der »Verdrahtung« der Crossing-Pattern recht formal.

[4] Das ist etwas trickreicher.

[5] Etwas vage, aber das formal hinzuschreiben macht es auch nicht besser.

[6] Jetzt kommt die Korrektheit aus dem Beweisrezept »Konstruktiver Beweis«. Die eigentliche Induktion wurde nicht hingeschrieben. War dem Autor wohl zu fummelig.

[7] Offenbar wird hier nurnoch skizziert, wie das in etwa geht.

7.3 Verhältnis zu regulären Sprachen II

Endliche Automaten akzeptieren alle regulären Sprachen.

7-22

▶ Satz

Sei L eine Sprache. Dann sind die folgenden Aussagen sind äquivalent:

1. *L ist regulär.*
2. *L wird durch eine reguläre Grammatik erzeugt.*
3. *L wird von einem* DFA *akzeptiert.*
4. *L wird von einem* NFA *akzeptiert.*
5. *L wird von einem* λ*-*NFA *akzeptiert.*
6. *L wird von einem 2-Wege-*NFA *akzeptiert.*

Beweisidee

- Die Äquivalenz der ersten beiden Punkte folgt per Definition. Die Äquivalenz der letzten vier Punkte haben wir schon gezeigt. Wir haben auch schon gezeigt, dass Punkt 3 Punkt 2 impliziert.

- Man zeigt dann noch, dass Punkt 2 Punkt 5 impliziert:
  - Gegeben sei eine reguläre Grammatik G.
  - Zustände des Automaten sind die Nonterminal von G.
  - Für jede Regel $X \to \lambda$ wird X akzeptierend.
  - Aus jeder Regel $X \to xY$ wird $\big((X,x),Y\big) \in \Delta$.
  - Aus jeder Regel $X \to Y$ wird $\big((X,\lambda),Y\big) \in \Delta$.
  - Dann zeigt man, dass der Automat genau die Ableitungen der Grammatik »nachvollzieht«.

Kommentare zum Beweis

Skript [1] Es ist fast alles schon gezeigt, es bleibt lediglich, wie man Grammatiken in Automaten umwandelt.

[2] Dies ist eine Abwandlung des Beweisrezepts »Kreisschluss«.

[3] Die entstandende Grammatik heißt immernoch G.

[4] Hier müsste man eigentlich das Rezept »Korrektheit von Grammatiken« anwenden, dieser Teil liegt aber nicht im Fokus das Beweises und wird übergangen.

[5] Rezept »Konstruktiver Beweis«

Beweis. [1] Die ersten beiden Punkt sind äquivalent per Definition. Punkte 3 bis 6 sind äquivalent nach Sätzen 6-25, 7-8 und 7-21. Nach dem Satz über die Mächtigkeit von DFAs, Satz 4-24, folgt aus Punkt 3 auch Punkt 2. Wir zeigen nun noch, dass der zweite Punkte den fünften impliziert, dass also jede reguläre Sprache von einem λ-NFA akzeptiert werden kann.[2]
Sei dazu $G = (T,N,S,P)$ eine reguläre Grammatik. Falls nötig ersetzen wir Regeln der Art $X \to wY$ mit $w \in T^{\geq 2}$ und $Y \in N$ durch eine Folge von Regeln $X \to w[1]X_1, X_1 \to w[2]X_2, \ldots, X_{|w|-1} \to w[|w|]Y$. Hierbei sind die X_i jedesmal völlig neue Nonterminale.[3] Weiter ersetzen wir alle Regeln $X \to w$ mit $w \in T^{\geq 1}$ durch die Regeln $X \to w[1]X_1, X_1 \to w[2]X_2, \ldots, X_{|w|-1} \to w[|w|]X_{|w|}, X_{|w|} \to \lambda$. Man überzeugt sich leicht, dass dies die erzeugte Sprache nicht ändert.[4]
Wir bauen nun einen λ-NFA $M = (\Sigma, Q, q_0, Q_a, \Delta)$ wie folgt:[5]

- $\Sigma = T$,
- $Q = N$,
- $q_0 = S$,
- $Q_a = \{X \mid (X \to \lambda) \in P\}$,
- $\big((X,a),Y\big) \in \Delta$ genau dann, wenn $(X \to aY) \in P$,
- $\big((X,\lambda),Y\big) \in \Delta$ genau dann, wenn $(X \to Y) \in P$.

Offenbar ist M ein λ-NFA. Wir behaupten, dass $L(G) = L(M)$. Dazu zeigen wir zwei Richtungen.
Für die erste Richtung sei $w \in L(G)$. Dann existiert eine Ableitung $S \Rightarrow^* w$. Sei $S = w_1 \Rightarrow \cdots \Rightarrow w_n = w$ diese Ableitung. Jedes w_i außer w_n ist dann von der Form $u_i X_i$ mit $u_i \in T^*$ und $X_i \in N$. Man zeigt nun leicht durch Induktion, dass der Automat w akzeptiert, indem er nacheinander die Zustände $S = X_1$, $X_2, X_3, \ldots, X_{n-1} \in Q_a$ durchläuft.[6] Dabei macht er im i-ten Schritt einen λ-Übergang, falls eine Regel der Form $X_i \to X_{i+1}$ angewandt wurde, sonst liest er ein Zeichen.
Für die zweite Richtung sei $w \in L(M)$. Dann existiert eine Berechnung $(S,w) = (X_0,w) \vdash \cdots \vdash (X_n,\lambda)$ mit $X_n \in Q_a$. Nun kann man wieder durch Induktion zeigen, dass $S \Rightarrow^* w$ via einer Ableitungskette gilt, wobei der i-te Ableitungsschritt entweder $X_i \Rightarrow X_{i+1}$ ist, falls nämlich M einen λ-Schritt macht, oder sonst $X_i \Rightarrow xX_{i+1}$ für das passende x. Also gilt $w \in L(G)$. □

[6] Anstatt die Induktion jetzt durchzuführen, wird nur gezeigt, was die wesentliche Idee ist.

7-23

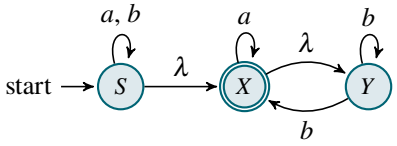
Beispiel für die Konstruktion aus dem Beweis.

Beispiel

Aus

$$\begin{aligned} G\colon S &\to X \mid aS \mid bS \\ X &\to aX \mid Y \mid \lambda \\ Y &\to bX \mid bY \end{aligned}$$

wird



Zusammenfassung dieses Kapitels

1. λ-NFAs können ihren Zustand »spontan wechseln«, ohne ein Zeichen zu lesen. 7-24
2. 2-Wege-NFAs können ihren Kopf in beide Richtungen bewegen und während der Berechnung ihre Richtung ändern.
3. Alle diese Automatenarten sind äquivalent untereinander und auch zu regulären Grammatiken.

Zum Weiterlesen

[1] R. Reischuk. *Unterlagen zum Lehrmodul Theoretische Informatik,* Vorlesungsskript, 2008. Kapitel 1.4, 2.2, 2.3.

Übungen zu diesem Kapitel

Übung 7.1 2-Wege-DFA, schwer

Sei k fest. Ein String ist in der Sprache k-PALINDROM $\subseteq \{0,1\}^*$, wenn die Konkatenation von seinem Präfix der Länge k und von seinem Suffix der Länge k ein Palindrom ergibt. Beschreiben Sie jeweils einen 2-Wege-DFA und einen 2-Wege-NFA, welche diese Sprache akzeptieren, indem Sie die notwendigen Zustände angeben und das Verhalten der Automaten in diesen Zuständen beschreiben. Wie viele Zustände haben Ihre Automaten? Wie viele Zustände müsste ein 1-Weg-DFA dagegen aufweisen? Sie müssen die Automaten nicht formal spezifizieren, jedoch sollte aus Ihrer Beschreibung der Aufbau und Korrektheit der Automaten ersichtlich werden.

Übung 7.2 2-Wege-NFA, schwer

Zeigen Sie: Ist L regulär, so auch $\{u \mid \exists v \in \Sigma^* : |v| = |u| \wedge uv \in L\}$.

Übung 7.3 2-Wege-Automaten, mittel

Sei $L_8 \subseteq \{0,1\}^*$ die Sprache aller Strings, die an der achtletzten Position eine 1 enthalten. Konstruieren Sie einen 2-Wege-Automaten M mit möglichst wenigen Zuständen, der die Sprache L akzeptiert. Und wie sieht ein DFA aus?

8-1

Kapitel 8
Reguläre Ausdrücke

```
"[.?!][]\"')]*\\($\\| $\\|\t\\| \\)[ \t\n]*"
```

8-2

Lernziele dieses Kapitels

1. Verschiedene Syntax-Varianten kennen
2. Semantik regulärer Ausdrücke verstehen
3. Syntaxprobleme mittels regulärer Ausdrücke beschreiben können
4. Äquivalenzbeweis zu regulären Sprachen verstehen

Inhalte dieses Kapitels

Worum es heute geht

Reguläre Grammatiken und endliche Automaten sind zwei Seiten derselben Münze: Grammatiken *beschreiben* Sprachen, Automaten *akzeptieren* sie. Heute soll der Münze noch eine dritte Seite hinzugefügt werden beziehungsweise, da dies offenbar bei Münzen zumindest im dreidimensionalen Raum nicht möglich ist, auf eine der Seiten noch ein weiteres Konzept zusätzlich eingraviert werden: reguläre Ausdrücke. Diese sind, genau wie reguläre Grammatiken, ein *Beschreibungsformalismus* für reguläre Sprachen.

Reguläre Ausdrücke können nicht mehr und nicht weniger als reguläre Grammatiken, sie sind *gleichmächtig*. Jedoch sind reguläre Ausdrücke in aller Regel wesentlich kompakter als reguläre Grammatiken, gleichzeitig sind sie für Menschen wesentlich leichter zu erstellen – wenn auch nicht unbedingt leichter zu lesen. Selbst ein Profi wird den regulären Ausdruck `"[.?!][]\"')]*\\($\\| $\\|\t\\| \\)[ \t\n]*"` nicht sofort verstehen. Dies ist laut Emacs-Handbuch »a simplified version of the regexp that Emacs uses, by default, to recognize the end of a sentence together with any whitespace that follows.«

Auch wenn reguläre Ausdrücke fürchterlich komplex und unlesbar werden können, so ist die Grundidee doch recht einfach: Beginnend mit trivialen Sprachen (bestehend nur aus einem Wort) kann man durch bestimmte Operationen wie Vereinigung oder Verkettung neue Sprachen bauen. Ein regulärer Ausdruck gibt nun gerade an, wie und in welcher Reihenfolge die Operationen angewendet werden sollen.

Die Hauptschwierigkeit bei regulären Ausdrücken ist auch nicht, dieses Konzept zu verstehen. Problematisch ist, dass sich keine einheitliche Syntax herausgebildet hat, wie man die Ausdrücke nun aufschreiben sollte. Hier kocht jeder sein eigenes Süppchen, welche wir nun alle auslöffeln müssen.

8.1 Die Idee

8.1.1 Neue Probleme aus alten

Wiederholung: Datums-Validierung

8-4

In vielen Anwendungen, besonders bei Web-Anwendungen, müssen *Datumseingaben* der Benutzer validiert (=überprüft) werden.
Dieses Entscheidungsproblem lässt sich mittels mehrerer Einzeltests *modularisieren*:

- Test, ob die Eingabe nur aus Ziffern und Punkten besteht.
- Test, ob es genau zwei Punkte gibt.
- Test, ob der Wortanfang aus der Menge $\{1,\ldots,31\}$ kommen.
- Test, ob nach dem ersten Punkte ein Wort aus $\{1,\ldots,12\}$ kommt.

Jeder Test ergibt eine Menge an Worten, die diesen Test besteht, also eine Sprache. Der *Schnitt* dieser Sprachen ist dann der Gesamttest.
Will man noch ein alternatives Datumsformat zulassen (wie `2009-01-01`), so schreibt man neue Tests, schneidet diese und *vereinigt* dann die beiden entstehenden Sprachen.

Die Idee hinter regulären Ausdrücke.

8-5

Bei der Beschreibung der Sprache der »gültigen Datumsangaben« kann man wie folgt vorgehen:

- Man beginnt mit *ganz, ganz einfachen* Sprachen wie $A = \{.\}$ oder $B = \{0,\ldots,9\}$ oder $C = \{1,\ldots,31\}$ oder $D = \{1,\ldots,12\}$, die jeweils nur *endlich viele* oder gar nur *ein* Wort enthalten.
- Dann kann man diese Sprachen mittels der Operationen Vereinigung, Schnitt, Komplement, Verkettung und Kleene-Stern *zusammengesetzt*, beispielsweise zu

$$C \circ A \circ D \circ A \circ B^*.$$

- Das Resultat ist eine Art »Ausdruck«, der ausgewertet genau die gewünschte Sprache ergibt.

Einen solchen Ausdruck nennen wir einen *regulären Ausdruck*.

Wiederholung: Wichtige Operationen auf Sprachen.

8-6

Seien $L_1 \subseteq \Sigma^*$ und $L_2 \subseteq \Sigma^*$ zwei Sprachen. Dann ist

1. $L_1 \cap L_2$ die Sprache, die alle Worte enthält, die *sowohl in* L_1 *als auch in* L_2 *sind*.
2. $L_1 \cup L_2$ die Sprache, die alle Worte enthält, die *in* L_1 *oder in* L_2 *sind*.
3. $\overline{L_1}$ die Sprache, die alle Worte enthält, die *in* Σ^*, *aber nicht in* L_1 *sind*.
4. $L_1 \circ L_2 := \{uv \mid u \in L_1, v \in L_2\}$ die Sprache, die alle Worte enthält, die *aus einem Wort in* L_1 *bestehen, gefolgt von einem Wort in* L_2.
   Diese Sprache nennt man die *Verkettung* von L_1 und L_2.
5. $L_1^* := \{u_1 \ldots u_n \mid u_i \in L_1\}$ die Sprache, die beliebige Folgen von *Worten in* L_1 enthält.
   Diese Sprache nennt man den *Kleene-Stern* von L_1.

8.1.2 Analogie zu arithmetischen Ausdrücken

Analogie von regulären und arithmetischen Ausdrücken.

8-7

Syntaktische Analogien

Arithmetische Ausdrücke

Bestandteile:

1. Zahlen
2. Unäre Operationen wie Wurzel.
3. Binäre Operationen wie
   - Addition
   - Multiplikation
   - Potenz
4. Klammern

Reguläre Ausdrücke

Bestandteile:

1. Ein-Wort-Sprachen
2. Unäre Operationen wie Kleene-Stern.
3. Binäre Operationen wie
   - Vereinigung
   - Verkettung
4. Klammern

8-8

Analogie von regulären und arithmetischen Ausdrücken.

Semantische Analogien

Arithmetische Ausdrücke

1. Ein arithmetischer *Ausdruck* und sein *Wert* sind verschiedene Dinge:
   Der *Ausdruck* $(5+6)\cdot 2$ hat den *Wert* 22.
2. Eine Zeichenfolge wie 22 kann man sowohl als Ausdruck als auch als Wert lesen.

Reguläre Ausdrücke

1. Ein regulärer *Ausdruck* und sein *Wert* sind verschiedene Dinge:
   Der *Ausdruck* `(a|b)*aa` hat den *Wert* $\{w \in \{a,b\}^* \mid w \text{ endet auf } aa\}$.
2. Eine Zeichenfolge wie `1*` ist ein Ausdruck, wird manchmal aber auch als ihr Wert gelesen (der eigentlich $\{1\}^*$ ist).

8.2 Die Theorie

8.2.1 Syntax

8-9

Definition von regulären Ausdrücken

▶ Definition: Regulärer Ausdruck

Sei Σ ein Alphabet. Die Menge der *regulären Ausdrücke über* Σ ist eine rekursiv definierte Menge von Worten über dem Alphabet $\Sigma \cup \{\texttt{(}, \texttt{)}, \texttt{|}, \texttt{*}\}$:

1. λ ist ein regulärer Ausdruck.
2. Für jedes $x \in \Sigma$ ist x (alleine, als einzelner Buchstabe) ein regulärer Ausdruck.
3. Sind R_1 und R_2 reguläre Ausdrücke, so auch R_1R_2 und $\texttt{(}R_1\texttt{|}R_2\texttt{)}$.
4. Ist R ein regulärer Ausdruck, so auch $\texttt{(}R\texttt{)*}$.

Aus technischen Gründen ist *zusätzlich* zu den obigen Worten ist auch noch das einzelne Symbol $\varnothing$ ein regulärer Ausdruck.

Überflüssige Klammern dürfen der Übersichtlichkeit halber auch weggelassen werden.

8.2.2 Semantik

8-10

Die Semantik von regulären Ausdrücken: Einfach auswerten.

▶ Definition: Beschriebene Sprache

Sei R ein regulärer Ausdruck über Σ. Dann ist die $L(R) \subseteq \Sigma^*$ eine Sprache, die wie folgt rekursiv definiert ist:

- Ist $R = \varnothing$, so ist $L(R) = \emptyset$.
- Ist $R = \lambda$, so ist $L(R) = \{\lambda\}$.
- Ist $R = x$ mit $x \in \Sigma$, so ist $L(R) = \{x\}$.
- Ist $R = R_1R_2$, so ist $L(R) = L(R_1) \circ L(R_2)$.
- Ist $R = \texttt{(}R_1\texttt{|}R_2\texttt{)}$, so ist $L(R) = L(R_1) \cup L(R_2)$.
- Ist $R = \texttt{(}R_1\texttt{)*}$, so ist $L(R) = L(R_1)^*$.

Achtung!

- Ein Ausdruck wie `0(1)*` ist erstmal ein *Wort* (über einem komischen Alphabet).
- Hingegen ist $L(\texttt{0(1)*}) = \{01^n \mid n \geq 0\}$ eine Sprache.
- In der Praxis ist man manchmal schlampig und schreibt einfach »1^*« für »$L(\texttt{(1)*})$«.

Beispiele regulärer Ausdrücke. 8-11

Sei $\Sigma = \{\texttt{0},\ldots,\texttt{9},\texttt{a},\ldots,\texttt{z},\texttt{A},\ldots,\texttt{Z},\texttt{.}\}$.

Beispiel: Regulärer Ausdruck für erlaubten Java-Bezeichner

`(a|···|z|A|···|Z)(a|···|z|A|···|Z|0|···|9)*`

Der Ausdruck beinhaltet natürlich nur Bezeichner über dem Alphabet Σ – in Wirklichkeit gibt es viel mehr.

Beispiel: Regulärer Ausdruck für Datumsangaben

`(01|···|31).(01|···|12).(0|···|9)(0|···|9)`

Hier haben Tag, Monat und Jahr je zwei Stellen. Es sind auch eigentlich falschen Daten wie `31.02.01` möglich.

Beispiel: Nachkommazahlen ohne führende Nullen

`(0|(1|···|9)(0|···|9)*)(|.(0|···|9)*)`

✎ Zur Übung 8-12

Geben Sie einen (einigermaßen kompakten) regulären Ausdruck R an, so dass $L(R)$ die Sprache der gültigen IP-Adressen ist (wie `127.0.0.1` oder `192.168.178.1`).

8.2.3 Mächtigkeit I: Im Westen nichts Neues

Reguläre Ausdrücke können nur reguläre Sprachen beschreiben. 8-13

▶ **Satz**

Sei R ein regulärer Ausdruck. Dann ist $L(R)$ regulär.

Beweis. Man zeigt die Behauptung durch *strukturelle Induktion*. [1] Für den Induktionsanfang sei $R = \lambda$ oder $R = x$ mit $x \in \Sigma$. Dann stimmt die Behauptung, da sowohl die Menge $\{\lambda\}$ wie auch die einelementige Menge $\{x\}$ regulär sind.[2] Für den Induktionsschritt seien R_1 und R_2 reguläre Ausdrücke und nach Induktionsvoraussetzung seien $L(R_1)$ und $L(R_2)$ regulär. Nun gilt:

- Für $R = R_1R_2$ ist $L(R) = L(R_1) \circ L(R_2)$ regulär, da die Verkettung von regulären Sprachen regulär ist nach Übung 3.5.
- Für $R = (R_1 \mid R_2)$ ist $L(R) = L(R_1) \cup L(R_2)$ regulär, da die Vereinigung von regulären Sprachen regulär ist nach Satz 3-20.
- Für $R = (R_1)*$ ist $L(R) = L(R_1)^*$ regulär nach Übung 3.7. □

Kommentare zum Beweis

[1] Das ist letztendlich auch nur eine Induktion über die Länge des Ausdrucks.

[2] Wie so oft ist der Induktionsanfang eher einfach.

Bemerkungen zur Mächtigkeit: Was geht noch? 8-14

- Man kann sich fragen, warum ausgerechnet die Vereinigung, die Verkettung und der Kleene-Stern als Operationen bei regulären Ausdrücken zugelassen sind.
- Es liegt insbesondere nahe, auch den Schnitt und das Komplement zuzulassen.
- Dies *kann man auch machen*, da die Klasse der regulären Sprachen auch unter Schnitt und Komplement abgeschlossen ist. (Für das Komplement sieht man das ganz leicht mittels endlicher Automaten, für den Schnitt mit den De Morgan'schen Regeln.)
- Es gibt aber zwei Gründe, weshalb man diese Operationen in der Regel doch nicht zulässt:
  1. Es wird schwieriger, reguläre Ausdrücke in DFAs umzuwandeln.
  2. Die drei Operationen Verkettung, Vereinigung und Stern reichen bereits, um alle regulären Sprachen zu beschreiben.

8.2.4 Mächtigkeit II: Perlentaucher

8-15

Reguläre Ausdrücke können alle regulären Sprachen beschreiben.

▶ **Satz**
Sei L eine reguläre Sprache. Dann gibt es einen regulären Ausdruck R mit $L = L(R)$.

- Dieser Satz ist mal wieder etwas kniffeliger zu beweisen.
- Ganz naiv schnappt man sich erstmal einen DFA M mit $L(M) = L$.
- Die Idee ist nun, die Arbeitsweise des Automaten irgendwie in einen regulären Ausdruck zu verwandeln.
- Dies ist aber schwierig, da Automaten »verknäult« sind; reguläre Ausdrücke hingegen eher »hierarchisch« sind.
- Wir brauchen also einen Weg, das Knäuel in eine Hierarchie zu verwandeln.

8-16

Ein paar einfache Fälle

Eine erste, einfache Frage

»Welche Worte kann M lesen, wenn er nur einen Schritt von einem Zustand q zu einem Zustand q' macht?«

- Offenbar kann der Automat nur ein einziges Zeichen lesen, nämlich genau diejenigen $x \in \Sigma$ mit $\delta(q,x) = q'$.
- Ein regulärer Ausdruck für die Menge $\{x_1, \ldots, x_m\}$ all dieser Zeichen ist: $R_{q \to q'} = (x_1 \mid \cdots \mid x_m)$.

Eine zweite, genauso einfache Frage

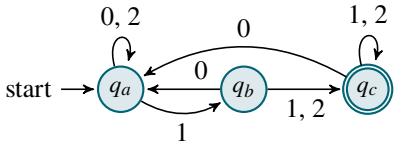
»Welche Worte kann M lesen, wenn er nur im Zustand q bleibt?«

- Nun kann der Automat beliebige Folgen von Zeichen x lesen, für die es eine Schlaufe am Zustand q gibt, für die also $\delta(q,x) = q$ gilt.
- Die Menge $\{x \in \Sigma \mid \delta(q,x) = q\}^*$ all dieser Worte lässt sich so beschreiben: $R_q = (x_1 \mid \cdots \mid x_m)\,\star$, wobei die x_i alle Zeichen mit $\delta(q,x_i) = q$.

8-17

Beispiele für die einfachen Fälle.

Sei M folgender Automat



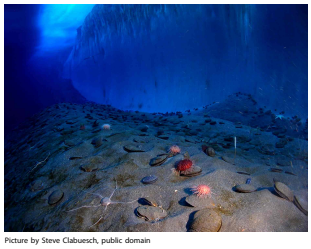
Dann ist

$$\begin{aligned} R_{q_a \to q_b} &= \texttt{1} \\ R_{q_b \to q_c} &= \texttt{(1|2)} \\ R_{q_a} &= \texttt{(0|2)*} \\ R_{q_b} &= \lambda \\ R_{q_c} &= \texttt{(1|2)*} \end{aligned}$$

8-18

Ein paar originelle Definitionen.

Sei $M = (\Sigma, Q, q_0, Q_a, \delta)$ ein DFA.


Picture by Steve Clabuesch, public domain

▶ **Definition: Tiefsee**
Eine *Tiefsee* ist eine Menge $T \subseteq Q$ von Zuständen.

▶ **Definition**
Seien $q, q' \in T$ zwei Zustände in der Tiefsee. Wir sagen, dass ein Wort $w \in \Sigma^*$ *den Automaten unter Wasser von q nach q' überführt*, wenn

1. $\delta^*(q,w) = q'$ gilt (startet der Automat in q und liest er w, so endet er in q') und
2. während der Berechnung durchschreitet er nur Zustände aus T.

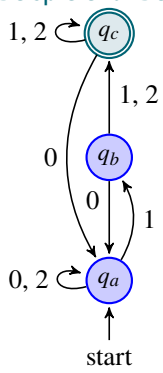
Die Menge aller solcher Worte bezeichnen wir mit $L^T_{q \to^* q'}$

Ziel

Für gegebenes T, q und q' wollen einen regulären Ausdruck $R^T_{q\to^* q'}$ konstruieren für die Sprache $L^T_{q\to^* q'}$ konstruieren.

Beispiele für Berechnungen unter Wasser.

8-19



Sei $T = \{q_a, q_b\}$. Dann ist

$$L^T_{q_a\to^* q_a} = \{0,2,10\}^*,$$
$$L^T_{q_a\to^* q_b} = \{0,2,10\}^* \circ \{1\}$$

und entsprechend suchen wir

$$R^T_{q_a\to^* q_a} = \texttt{(0|2|10)*},$$
$$R^T_{q_a\to^* q_b} = \texttt{(0|2|10)*1}.$$

Die Idee der Perlentaucher.

8-20

Bild von User Aquaxel, Creative Commons Attribution Lizenz.

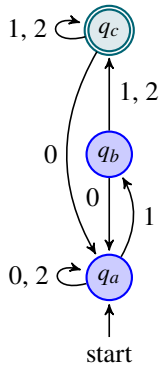
- Nehmen wir an, wir haben für die Tiefsee alle $R^T_{q\to^* q'}$ bestimmt.
- Nun kommt ein neuer Zustand q hinzu, den wir uns »auf der Tiefsee schwimmend« vorstellen.
- Betrachten wir nun eine Berechnung, die in q beginnt und dort endet und dazwischen nur Zustände aus der Tiefsee benutzt.
- Eine solche Berechnung nennen wir einen *Tauchgang*.

▶ Definition: Tauchgang

Sei $T \subseteq Q$ eine Tiefsee und $q \notin T$ ein Zustand. Ein *Tauchgang* ist ein Wort $w \in \Sigma^*$, so dass

1. $\delta^*(q,w) = q$ und
2. außer am Anfang und am Ende durchschreitet er nur Zustände aus T.

Die Menge aller solcher Worte bezeichnen wir mit L^T_q.



Sei $T = \{q_a, q_b\}$. Dann ist

$$L^T_{q_c} = (\{0\} \circ \{0,2,10\}^* \circ \{1\} \circ \{1\}) \cup$$
$$\{0\} \circ \{0,2,10\}^* \circ \{1\} \circ \{2\}.$$

Diese Sprache lässt sich durch folgenden regulären Ausdruck beschreiben:

$$R^T_{q_c} = \texttt{((0(0|2|10)*11)|}\\ \texttt{(0(0|2|10)*12)).}$$

8-21

Alle Tauchgänge lassen sich durch reguläre Ausdrücke beschreiben.

▶ **Lemma**
Sei $T = \{q_1, \ldots, q_m\}$ eine Tiefsee und $q \notin T$. Sei

$$\begin{aligned} R^T_q = (&R_{q\to q_1} R^T_{q_1\to^* q_1} R_{q_1\to q} \mid \cdots \mid R_{q\to q_1} R^T_{q_1\to^* q_m} R_{q_m\to q} \mid \\ &R_{q\to q_2} R^T_{q_2\to^* q_1} R_{q_1\to q} \mid \cdots \mid R_{q\to q_2} R^T_{q_2\to^* q_m} R_{q_m\to q} \mid \\ &\vdots \\ &R_{q\to q_m} R^T_{q_m\to^* q_1} R_{q_1\to q} \mid \cdots \mid R_{q\to q_m} R^T_{q_m\to^* q_m} R_{q_m\to q}) . \end{aligned}$$

Dann ist $L(R^T_q) = L^T_q$.

Beweis. Jeder Tauchgang muss in q starten, dann zu einem $q_i \in T$ »abtauchen« (was durch $R_{q\to q_i}$ beschrieben wird), dann eine Weile in der Tiefsee bleiben (was durch $R^T_{q_i\to^* q_j}$ beschrieben wird) und dann von $q_j \in T$ zu q »auftauchen« (was durch $R_{q_j\to q}$ beschrieben wird). □

8-22

Tauchausflüge: Folgen von Tauchgängen.

Beim nächsten Szenario dürfen wir »beliebig oft wieder auftauchen«.

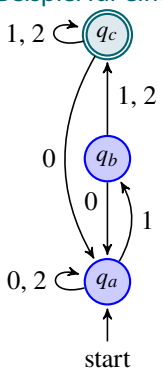
▶ **Definition:** Tauchausflug
Sei wieder T eine Tiefsee und $q \notin T$. Wir nennen ein Wort $w \in \Sigma^*$ einen *Tauchausflug*, wenn

1. $\delta^*(q, w) = q$ und
2. die Berechnung durchschreitet nur Zustände aus $T \cup \{q\}$.

Die Menge aller Tauchausflüge ist gerade $L^{T\cup\{q\}}_{q\to^* q}$.

8-23

Beispiel für einen Tauchausflug.



Sei $T = \{q_a, q_b\}$. Dann ist

$$L^{T\cup\{q_c\}}_{q_c\to^* q_c} = \left(\{1,2\}^* \circ L^T_{q_c} \circ \{1,2\}^*\right)^*.$$

und entsprechend suchen wir

$$R^{T\cup\{q_c\}}_{q_c\to^* q_c} = \texttt{((1|2)*}R^T_{q_c}\texttt{(1|2)*)*}.$$

Alle Tauchausflüge lassen sich durch reguläre Ausdrücke beschreiben.

8-24

▶ **Lemma**
Sei $T = \{q_1, \ldots, q_m\}$ eine Tiefsee und $q \notin T$. Dann gilt für

$$R^{T\cup\{q\}}_{q\to^* q} = (R_q R_q^T R_q)^*,$$

dass $L(R^{T\cup\{q\}}_{q\to^* q}) = L^{T\cup\{q\}}_{q\to^* q}$.

Beweis. Jeder Tauchausflug besteht aus einer Folge von Tauchgängen, zwischen denen der Automat im Zustand q bleibt. □

Auf- und abgetaucht.

8-25

Nun erweitern Tauchausflüge, so dass wir statt in q auch in der Tiefsee beginnen und enden können.

▶ **Lemma**
Sei $T = \{q_1, \ldots, q_m\}$ eine Tiefsee und $q \notin T$. Seien $q', q'' \in T$. Dann gilt für

$$\begin{aligned} R^{T\cup\{q\}}_{q'\to^* q''} = (&R^T_{q'\to^* q''} \mid \\ &(R^T_{q'\to^* q_1} R_{q_1\to q} \mid \cdots \mid R^T_{q'\to^* q_m} R_{q_m\to q}) \\ &R^{T\cup\{q\}}_{q\to^* q} \\ &(R_{q\to q_1} R^T_{q_1\to^* q''} \mid \cdots \mid R_{q\to q_m} R^T_{q_m\to^* q''})), \end{aligned}$$

dass $L(R^{T\cup\{q\}}_{q'\to^* q''}) = L^{T\cup\{q\}}_{q'\to^* q''}$.

Beweis. Jede Berechnung, die in q' beginnt und in q'' endet, bleibt entweder komplett in der Tiefsee oder sie taucht irgendwann auf, macht einen Tauchausflug und taucht wieder ab. □

Bestandsaufnahme

8-26

Was wir erreicht haben:

- Wir haben mit einer Tiefsee T begonnen und einem Zustand $q \notin T$.
- Für die Tiefsee kannten wir schon reguläre Ausdrücke, die beschreiben, wie Berechnungen unter Wasser ablaufen.
- Jetzt haben wir auch reguläre Ausdrücke, die Berechnungen *unter Wasser oder durch q* beschreiben.
- Fassen wir also $T \cup \{q\}$ als neue Tiefsee auf (der Meeresspiegel steigt sozusagen), dann *haben wir nun auch reguläre Ausdrücke für Berechnungen unter Wasser zu diesem erhöhten Meeresspiegel.*

Was noch fehlt:

1. Wir müssen nun einfach den Meerespiegel so lange erhöhen, bis alle Zustände unter Wasser liegen.
2. Wir müssen dann die Berechnungen herauspicken, die vom Startzustand zu einem akzeptierenden Zustand führen.

Der Beweis der Mächtigkeit

8-27

Beweis von Satz 8-15. Wir zeigen, dass für jeden DFA M ein regulärer Ausdruck R existiert mit $L(M) = L(R)$.[1]
Sei $Q = \{q_1, \ldots, q_n\}$ und sei $T_i = \{q_1, \ldots, q_i\}$. Zunächst konstruieren wir induktiv für steigende i reguläre Ausdrücke $R^{T_i}_{q\to^* q'}$ für jedes Paar $q, q \in T_i$ für die gerade gilt $L(R^{T_i}_{q\to^* q'}) = L^{T_i}_{q\to^* q'}$.[2] Es wurde schon in Lemma 8-25 gezeigt, wie solche Ausdrücke induktiv berechnet werden können.
Der gesuchte reguläre Ausdruck ist dann gegeben durch $(R^Q_{q_0\to q_{a_1}} \mid \cdots \mid R^Q_{q_0\to q_{a_m}})$, wobei $\{q_{a_1}, \ldots, q_{a_m}\} = Q_a$ gerade die Menge der akzeptierenden Zuständen ist.[3] □

Kommentare zum Beweis

[1] Nochmal die Behauptung, da man die mittlerweile ja eher vergessen hat.

[2] Der Induktionsanfang wurde weggelassen. Scheint dem Autor trivial vorgekommen zu sein. Ihnen auch?

[3] Die Korrektheit wird mal wieder nicht gezeigt.

8.3 Die Praxis

8.3.1 Programme und Bibliotheken

8-28

Einsatzgebiete von regulären Ausdrücken.

- Innerhalb von Programmen werden reguläre Ausdrücke benutzt
  - zur *Eingabekontrolle* und
  - zum *Parsen von Eingaben*.
    Beispiel: Zerlegung einer URL in ihre Bestandteile.

  Für praktisch alle Programmiersprachen (zum Beispiel für Java, C++, Perl, PHP, etc.) gibt es fertige Bibliotheken, die reguläre Ausdrücke automatisch in Automaten umwandeln.
- Menschen können reguläre Ausdrücke für *Suchanfragen* benutzen.
  - In SQL kann man statt `like` auch *`regexp`* benutzen, um nach Attributen zu filtern.
  - In Microsoft Word kann man im Suchen-Dialog einen *Mustervergleich* einschalten.
  - Das Programm `grep` sucht nach einem Vorkommen eines regulären Ausdrucks in einer Datei.

8.3.2 Syntax-Varianten

8-29

Probleme beim Einsatz von regulären Ausdrücken.

Notationsprobleme

- Will man reguläre Ausdrücke aufschreiben, so muss man irgendwie alles mit ASCII-Zeichen aufschreiben.
- Leider hat sich keinerlei Standard zum Aufschreiben von regulären Ausdrücken durchgesetzt.

Effizienzprobleme

- Manche Arten regulärer Ausdrücke lassen sich *effizienter behandeln* als andere.
- Deshalb ist es manchmal sinnvoll, nur *bestimmte Arten von regulären Ausdrücken zuzulassen*.
- Einige Programme vereinfachen deshalb die Notation, sind dann aber inkompatibel mit anderen Notationen.

8-30

Typische Notationen für reguläre Ausdrücke.

Auch wenn die Notation für reguläre Ausdrücke nicht einheitlich ist, so gilt wenigstens *oft*:

- Klammern schreibt man als Klammern, Sternchen als Sternchen. Sie gelten also als *Sonderzeichen*.
  Will man diese trotzdem eingeben, so muss man sie mittels eines Backslash »escapen«.
- Statt `a|b|c|d|e|x|y|z` kann man auch schreiben `[abcdexyz]` oder auch kürzer `[a-exyz]`.
- Statt einer Aufzählung aller möglichen Zeichen kann man einfach einen Punkt schreiben, er steht für ein beliebiges Zeichen.
- Statt $(\varepsilon|A)$, was »gar kein Vorkommen von A oder ein Vorkommen von A« bedeutet, kann man schreiben $A?$.
- Statt AA^*, was »mindestens ein Vorkommen von A« bedeutet, kann man schreiben $A+$.

Die Besonderheit des Wortanfangs und -endes

- In der Praxis beschreibt ein regulärer Ausdruck nicht den kompletten Eingabestring.
- Vielmehr *sucht* man in der Regel nach Vorkommen von Worten, die durch den regulären Ausdruck beschrieben werden.
- Dies bedeutet einfach, dass ein Ausdruck wie `(0|1)*123` die Sprache beschreibt aller Worte, die *irgendwo eine Folge von* 0*en und* 1*en gefolgt von* `123` *enthalten*.
- Stellt man aber einem regulären Ausdruck `^` voran, so muss das vom regulären Ausdruck beschriebene Wort tatsächlich am Anfang sein.
- Entsprechend erzwingt ein nachgestelltes `$`, dass die Eingabe mit dem beschriebenen Wort endet.

Wichtige Syntax-Varianten und zwei Beispiele

8-31

PCRE »Perl Compatible Regular Expressions« ist eine Syntax, die im Umfeld der Programmiersprache Perl entstanden ist.

POSIX Dieser Standard legt eine mögliche Syntax fest. Sie wird von vielen Unix-Programmen, insbesondere von `grep` benutzt.

```
\b[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}\b
```

You could use the regular expression to search for an email address.

Aus `regular-expressions.info`

```
[.?!][]\"')]*\\($\\| $\\|\t\\|  \\)[ \t\n]*
```

A simplified version of the regexp that EMACS uses, by default, to recognize the end of a sentence together with any whitespace that follows.

Aus dem EMACS-Handbuch

8.3.3 Referenz: Reguläre Ausdrücke in POSIX

Skript-Referenz

Die folgenden Regeln geben einen kurzen Überblick über die POSIX-Regel-Syntax. In der darauf folgenden Tabelle sind einige Beispiele angegeben, die diese Regeln verdeutlichen.

1. Die Sprache aller Zeichenketten, die das Zeichen `a` enthalten, wird einfach durch den regulären Ausdruck `a` beschrieben. Dasselbe gilt für alle anderen Zeichen, sofern sie keine Sonderzeichen sind.
2. Die folgenden Zeichen sind Sonderzeichen:

   `\ . ^ $ [ ] ( ) * ? + | /`

   Sonderzeichen haben besondere Funktionen in regulären Ausdrücken. Um trotzdem Sprachen zu definieren, die diese Zeichen enthalten, stellen Sie dem entsprechenden Zeichen einen Backslash (»\«) voran. *(Beispiel 1)*
3. Das Sonderzeichen `.` steht für genau ein beliebiges Zeichen (auch ein Leerzeichen). *(Beispiele 3, 14, 15)*
4. Das Sonderzeichen `^` steht für den Anfang einer Zeichenkette. *(Beispiele 4, 12–15)*
5. Das Sonderzeichen `$` steht für das Ende eines Wortes. *(Beispiele 5, 12–15)*
6. Mit eckigen Klammern können mehrere Möglichkeiten für genau ein Zeichen definiert werden. *(Beispiele 6, 7, 13)*
7. Folgt auf eine öffnende eckige Klammer das Zeichen `^`, so wird die Zeichenmenge in der Klammer negiert (es bedeutet *hier* also *nicht* den Anfang der Zeichenkette). *(Beispiel 7)*
8. Zeichenmengen in eckigen Klammern können mit dem Zeichen `-` abgekürzt definiert werden. Statt `[0123456789]` kann man also `[0-9]` schreiben. *(Beispiel 13)*
9. Reguläre Ausdrücke werden verkettet, indem sie einfach hintereinander geschrieben werden. Der Ausdruck `ab` steht also für die Sprache aller Zeichenketten, die irgendwo den Teilstring `ab` enthalten. *(Beispiele 8–15)*
10. Die Zeichen `?`, `*` und `+` können regulären Ausdruck nachgestellt werden *(Beispiele 9–15)*

    `?` Der vorangehende Ausdruck soll einmal oder keinmal vorkommen.

    `*` Der vorangehende Ausdruck soll keinmal oder beliebig oft direkt hintereinander vorkommen.

    `+` Der vorangehende Ausdruck soll einmal oder öfter direkt hintereinander vorkommen.
11. Durch runde Klammern können reguläre Ausdrücke gruppiert werden. Dies ist im Zusammenhang mit den Operationen `?`, `*`, `+` nützlich *(Beispiel 12)*: Ohne Gruppierung beziehen sich diese Operatoren immer nur auf das Zeichen direkt davor *(Beispiele 9–11)* beziehungsweise die eckige Klammer direkt davor *(Beispiel 13)*.
12. Mit dem Sonderzeichen `|` können zwei reguläre Ausdrücke vereinigt (»verodert«) werden. *(Beispiel 15)*

Beispiele für die Regeln:

	Ausdruck	*Beschreibt alle Zeichenketten, die …*
1	`\?`	… ein Fragezeichen enthalten.
2	`\\`	… einen Backslash enthalten.
3	`.`	… mindestens ein Zeichen lang sind.
4	`^a`	… mit `a` beginnen.
5	`a$`	… mit `a` enden.
6	`[abc]`	… irgendwo ein `a`, `b` oder `c` enthalten.
7	`[^abc]`	… irgendwo ein Zeichen enthalten, das kein ein `a`, `b` oder `c` ist.
8	`abc`	… den Teilstring `abc` enthalten.
9	`ab?a`	… den Teilstring `aa` oder den Teilstring `aba` enthalten.
10	`ab*a`	… einen der Teilstrings `aa`, `aba`, `abba`, `abbba` und so weiter enthalten.
11	`ab+a`	… einen der Teilstrings `aba`, `abba`, `abbba` und so weiter enthalten.
12	`^a(ab)*b$`	… mit einem `a` beginnen, gefolgt von beliebig vielen Wiederholungen von `ab`, und mit einem `b` enden.
13	`^[_a-zA-Z][_a-zA-Z0-9]*$`	… gültige Java-Bezeichner sind.
14	`^A.*s$`	… mit `A` beginnen und mit `s` enden.
15	`^A.*s$\|^.s$`	… mit `A` beginnen und mit `s` enden oder mit `s` enden und nur zwei Zeichen lang sind (nicht »entweder oder«!).

Zusammenfassung dieses Kapitels

8-32

1. Reguläre Ausdrücke beschreiben, wie eine Sprache durch Operationen wie Vereinigung oder Verkettung aus elementaren Ein-Wort-Sprachen aufgebaut werden können.
2. Es gibt keine einheitliche Syntax für reguläre Ausdrücke, die wichtigsten in der Praxis sind die POSIX-Syntax und die Perl-Syntax.
3. Mit regulären Ausdrücken lassen sich genau die regulären Sprachen beschreiben.

Zum Weiterlesen

[1] Jan Goyvaerts. *Regular-Expressions.info*, Zugriff November 2009. `http://www.regular-expressions.info/`

[2] R. Reischuk. *Unterlagen zum Lehrmodul Theoretische Informatik,* Vorlesungsskript, 2008. Kapitel 2.3.

Übungen zu diesem Kapitel

Übung 8.1 Reguläre Aussdrücke üben, leicht

Beschreiben Sie verbal die Sprachen, die durch folgende reguläre Ausdrücke definiert werden:

1. `(00|1)*(11|0)`
2. `(1|01|001)*(1|111)`

Übung 8.2 Reguläre Ausdrücke konstruieren, mittel

Konstruieren Sie reguläre Ausdrücke für folgende Sprachen:

1. $\{w \in \Sigma^* \mid |w| \bmod 3 = 0\}$.
2. Die Menge der Worte, die mit einer Eins beginnen und auf eine Null enden oder mit einer Null beginnen und auf Eins enden.
3. 2-PALINDROM.

Übung 8.3 Reguläre Ausdrücke bauen, mittel

Beschreiben Sie folgende Sprachen über dem Alphabet $\Sigma = \{0,1\}$ durch reguläre Ausdrücke:

1. $A = \{w \in \Sigma^* \mid |w| \text{ ist ungerade}\}$
2. $B = \{w \in \Sigma^* \mid w \text{ enthält 00 nicht als Teilwort}\}$
3. $C = \{w \in \Sigma^* \mid w \text{ enthält eine ungerade Anzahl von Nullen}\}$
4. $D = \{w \in \Sigma^* \mid w \text{ enthält genau drei Einsen}\}$

9-1

Kapitel 9
Nerode-Klassen

Halte die Umwelt sauber! Mach' mit bei der Worttrennung!

Lernziele dieses Kapitels

1. Begriffe des Index und der Invarianz verstehen
2. Den Index einer Sprache bestimmen können
3. Den Satz von Myhill-Nerode und seinen Beweis verstehen
4. Automaten minimieren können

Inhalte dieses Kapitels

9-2

Worum es heute geht

Unbekannter Autor, Creative Commons Attribution Sharealike Lizenz

Wahrscheinlich tut man den Nerode-Klassen Unrecht, sie aus didaktischen Gründen im Rahmen der Mülltrennung einzuführen. Dabei ist es für die Lektüre vorteilhaft, aber auch nicht unbedingt nötig, das in Deutschland zur Obsession gesteigerte Verlangen zu verspüren, Müll nach allen nur denkbaren Kriterien zu trennen und zu sortieren. Es wird Ihnen in diesem Fall ganz natürlich vorkommen, auch Worte nach reichlich merkwürdigen Kriterien zu sortieren und in Worttonnen einzukippen.

Der Begriff der »Worttonnen« hat sich in der Theorie allerdings (noch?) nicht wirklich durchgesetzt, hätte jedoch gegenüber der sperrigen, aber mathematisch korrekten Bezeichnung »maximale Rechtsäquivalenzklasse« durchaus Chancen. Tatsächlich nennt man Worttonnen vornehm »Nerode-Klassen«.

Was bringt es nun, Worte in Klassen zu trennen – egal ob sie nun Worttonnen oder Nerode-Klassen heißen? Eine ganze Menge, wie sich herausstellt:

1. Aus den Nerode-Klassen einer regulären Sprache lässt sich leicht ein endlicher Automat bauen.
2. Dieser Automat ist auch gleich noch der kleinste überhaupt mögliche Automat für die Klasse.
3. Ist die Anzahl der Nerode-Klassen einer Sprache unendlich, so kann die Sprache nicht regulär sein.

Insbesondere der letzte Punkte dürfte alle Leser erfreuen, die mit dem Pumping-Lemma auf Kriegsfuß stehen, da es oft einfach ist zu zeigen, dass eine Sprache unendlich viele Nerode-Klassen hat.

9.1 Nerode-Klassen

9.1.1 Die Idee

9-4

Eine neue Sicht der Dinge.

- Es ist mathematisch (relativ) einfach, zu
  - einer gegebenen regulären Grammatik oder
  - einem gegebenen Automaten oder
  - einem gegebenen regulären Ausdruck

  die erzeugte/akzeptierte/beschriebene Sprache zu definieren.
- Wie kann man aber *umgekehrt* zu einer gegebenen regulären Sprache einen zugehörigen Automaten finden?
- Bis jetzt musste man dies immer durch »scharfes Hinsehen« oder »Intuition« oder »Übung« oder »Zuruf« oder »kurzes Nachdenken« erledigen.
- Das ist mathematisch noch etwas unbefriedigend.
- Wir suchen ein Verfahren, zu einer regulären Sprache möglichst »automatisch« einen Automaten zu finden.

9-5

Die innere Struktur einer regulären Sprache.

- Reguläre Sprachen haben besondere Eigenschaften wie *»Jede reguläre Sprache hat die Pump-Eigenschaft.«*
- Jedoch: Es gibt auch nichtreguläre Sprachen, die die Pump-Eigenschaft haben.
- In diesem Kapitel geht es um eine besondere Eigenschaft, die nur reguläre Sprachen haben: *Sie haben einen endlichen Index.*
- Wenn man diesen Index bestimmt, wird man ganz nebenbei auch gleich einen minimalen Automaten für die Sprache bestimmt haben.

9-6

Mülltrennung in der Theoretischen Informatik.

- Die »Verwertungsgesellschaft Wort« (VG Wort) hat ein neues Geschäftsfeld entdeckt: *Das Recycling von benutzten Worten.*
- Man hat nämlich festgestellt, dass bei der Untersuchung von Sprachen wie $L = \{w \in \{0,1,2\}^* \mid w \text{ enthält genau eine } 2\}$ Worte wie $011210 \in \{0,1,2\}^*$ nach Gebrauch häufig einfach weggeworfen werden!
- Im Land der Mülltrennung ist dies natürlich auf Dauer nicht hinnehmbar.
- Deshalb will die VG Wort Worte nach Gebrauch wieder einsammeln.
- Dazu soll es in den Büros von theoretischen Informatikern und in Hörsälen zukünftig *Worttonnen* geben.
- Selbstverständlich ist es dabei *ganz wichtig*, dass die Wörter immer in die richtigen Tonnen kommen, damit sie später besser wiederverwertet werden können.

9-7

Was passiert, wenn man ein Wort in die falsche Tonne wirft.

Die Regeln hinter den Worttonnen.

9-8

- Logischerweise sollte die Worttrennung so erfolgen, dass sich alle Worte in einer Tonne »möglichst ähnlich sind« – dann ist das Recycling recht einfach.
- Die VG Wort wollte zunächst nur zwei Tonnen aufstellen:
  - In eine Tonne sollten alle Worte w, die in der Sprache L liegen (also $w \in L$).
  - In die andere Tonne sollten alle anderen Worte.
- Leider stellte sich heraus, dass die so getrennten Worte *für eine Weiterverarbeitung ungeeignet sind*.
- Verkettet man nämlich Worte aus derselben Tonne mit anderen Worten, so erhält man Worte, die mal in der Sprache L sind und mal nicht.
- Deshalb hat die VG Wort ein komplizierteres Verfahren entwickelt.

Goldene Regel

Alle Worte in einer Worttonne haben folgende Eigenschaft: Hängt man ein beliebiges Wort an sie an, so sind die Resultate entweder *alle in L* oder *alle nicht in L*.

Ein Beispiel.

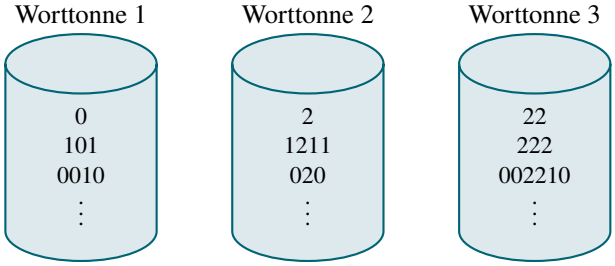
9-9

Sei $L = \{w \in \{0,1,2\}^* \mid w \text{ enthält genau eine } 2\}$.

- Die Worte 00 und 101 kommen *in dieselbe Worttonne*:
  - Hängt man ein Wort u an, das keine 2 enthält, dann sind sowohl $00u$ wie auch $101u$ keine Elemente von L.
  - Hängt man ein Wort u an, das genau eine 2 enthält, dann sind sowohl $00u$ wie auch $101u$ Elemente von L.
  - Hängt man ein Wort u an, das mehr als eine 2 enthält, dann sind wieder sowohl $00u$ wie auch $101u$ keine Elemente von L.
- Die Worte 020 und 12 kommen *ebenfalls in eine Worttonne*, aber *eine andere als* 00 *und* 101:
  - Hängt man $u = 0$ an 020 an, so landet man bei $0200 \in L$.
  - Hängt man $u = 0$ an 101 an, so landet man bei $1010 \notin L$.
- Die Worte 0220 und 11120222 kommen *auch in eine Worttonne*, aber *wiederum in eine neue*.

Die drei Worttonnen.

9-10



Etwas Nachdenken zeigt, dass man *nicht mehr Worttonnen braucht* für die Sprache $L = \{w \in \{0,1,2\}^* \mid w \text{ enthält eine } 2\}$.

✎ Zur Übung

9-11

Bestimmen Sie für *eine* der folgenden Sprachen, wie viele Worttonnen jeweils gebraucht werden (nach Schwierigkeit sortiert):

1. Sei $L_1 = \{w \in \{0,1,2\}^* \mid w \text{ enthält eine 1 oder eine } 2\}$.
2. Sei $L_2 = \{w \mid w \in \{0,1,2\}^*, w[|w|-1] = 2\}$.
3. Sei $L_3 = \{a^n b^n \mid n \geq 0\}$.

9.1.2 Die Definitionen

9-12 Die wissenschaftliche Bezeichnung von Worttonnen: Nerode-Klassen.

▶ Definition: Rechtsäquivalenz
Sei $L \subseteq \Sigma^*$. Zwei beliebige Wort $u, v \in \Sigma^*$ heißen *rechtsäquivalent in Bezug auf L*, wenn für alle Worte $w \in \Sigma^*$ gilt: $uw \in L \iff vw \in L$.
Die Relation R_L setzt gerade rechtsäquivalente Worte in Beziehung, also $(u,v) \in R_L$ genau dann, wenn u und v rechtsäquivalent sind.

Bemerkungen:

- Man beachte, dass u, v und auch w nicht Element von L sein müssen.
- Für Fortgeschrittene: Rechtsäquivalenz ist eine Äquivalenzrelation auf der Menge aller Worte.

▶ Definition: Nerode-Klassen
Sei L eine Sprache. Die *Nerode-Klasse* eines Wortes $u \in \Sigma^*$, geschrieben $N(u)$, ist die Menge aller Worte $v \in \Sigma^*$, die rechtsäquivalent zu u sind.

▶ Definition: Index einer Sprache
Der *Index* einer Sprache L ist die Anzahl ihrer Nerode-Klassen.

9-13 Eigenschaften von Nerode-Klassen.
Die Alles-oder-Nichts-Eigenschaft.

▶ Lemma
Sei L eine Sprache und N eine Nerode-Klasse. Dann gilt $N \subseteq L$ oder $N \subseteq \bar{L}$.

Beweis. Seien $u, v \in N$ beliebig. Da sie rechtsäquivalent sind, muss für jedes $w \in \Sigma^*$ gelten $uw \in L \iff vw \in L$. Also gilt insbesondere für $w = \lambda$, dass $u \in L \iff v \in L$. Es sind also entweder *alle* Worte aus N in L oder *gar keine*. □

9-14 Eigenschaften von Nerode-Klassen.
Die Mitgegangen-Mitgefangen-Eigenschaft.

▶ Lemma
Sei L eine Sprache. Seien u und v in derselben Nerode-Klasse und sei $w \in \Sigma^*$ beliebig. Dann sind auch uw und vw in derselben Nerode-Klasse (aber nicht unbedingt in derselben wie u und v).

Beweis. Sei $w' \in \Sigma^*$ beliebig. Dann gilt $(uw)w' \in L \iff u(ww') \in L \iff v(ww') \in L \iff (vw)w' \in L$. □

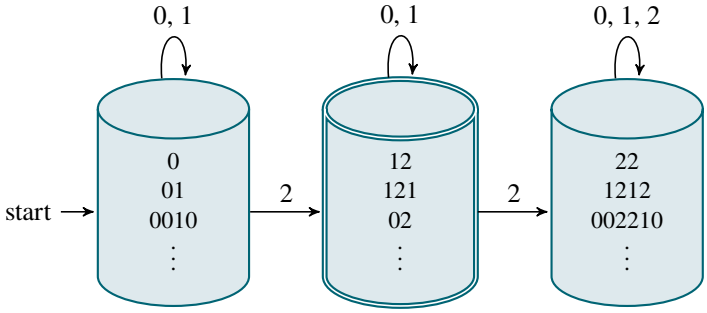
9.1.3 Der Satz von Myhill-Nerode

9-15 Ein paar Vorüberlegungen.

- Eine Nerode-Klassen versammelt alle Worte, »die sich ähnlich verhalten«.
- Beispielsweise gab es bei der Sprache $L = \{w \in \{0,1,2\}^* \mid w \text{ enthält genau eine } 2\}$ drei Nerode-Klassen:
  1. Alle Worte, die 2 nicht enthalten.
  2. Alle Worte, die 2 genau einmal enthalten.
  3. Alle Worte, die 2 mehr als einmal enthalten.
- Diese Klassen könnten *wunderbar als Zustände eines Automaten* dienen:
  1. Ein Zustand q_0, in dem wir bleiben, solang noch keine 2 kam.
  2. Ein akzeptierender Zustand q_1, den wir erreichen, wenn eine 2 kam.
  3. Ein Fehlerzustand q_2, in den wir wechseln, wenn eine zweite 2 kam.
- Dies ist kein Zufall: Man kann die Nerode-Klassen eine Sprache *immer* als Zustände eines DFA für die Sprache verwenden.

Von den Nerode-Klassen zum Automaten.

9-16



Die allgemeinen Ideen:

- Der *Startzustand* ist die Nerode-Klasse, die λ enthält.
- Die *akzeptierenden Zustände* sind die Nerode-Klassen, die nur Worte aus L enthalten.
- »Ist man in einer Nerode-Klassen $N(w)$« und liest ein Zeichen x, so kommt man zu der Nerodeklasse $N(wx)$.

Von den Nerode-Klassen zu einem Automaten.

9-17

▶ Definition

Sei $L \subseteq \Sigma^*$ eine Sprache mit endlichem Index. Seien $N_1, \ldots, N_n$ die Nerode-Klassen von L. Wir definieren einen DFA $M_L = (\Sigma, Q, q_0, Q_a, \delta)$ wie folgt:

1. $Q = \{N_1, \ldots, N_n\}$.
2. $q_0 = N(\lambda)$ ist Nerode-Klasse des leeren Wortes.
3. $Q_a = \{N_i \mid N_i \subseteq L\}$.
4. $\delta(N_i, x) = N(wx)$, wobei $w \in N_i$ beliebig ist.

- Bei diesem Automaten sind die *Zustände* Nerode-Klassen – also *unendlich große Mengen*.
- Jedoch gibt es nur endlich vielen Zustände, wie sich das für einen *endlichen* Automaten gehört.
- Wem das mit den unendlichen Mengen als Zuständen zu unheimlich ist, der denkt einfach nicht darüber nach, was in einem N_i alles drinsteckt.
- Die Funktion δ ist aufgrund von Lemma 9-14 wohldefiniert.

Charakterisierung der Regularität über den Index.

9-18

▶ Satz: Myhill-Nerode

Eine Sprache L ist genau dann regulär, wenn sie einen endlichen Index hat.

Beweis. Wir zeigen zwei Richtungen.[1] Habe zunächst L endlichen Index. Wir behaupten, dass für den DFA M_L von Definition 9-17 gilt, dass $L(M_L) = L$.[2] Dies sieht man so:[3] Liest M_L ein Wort w, so endet er offenbar genau in dem Zustand, der die Nerode-Klasse ist, die w enthält. Dieser Zustand ist somit genau dann akzeptierend, wenn $w \in L$ gilt.
Für die zweite Richtung sei L regulär. Dann existiert ein DFA M mit $L(M) = L$ nach Satz 7-22.[4] Für einen Zustand q von M sei $W_q = \{w \in \Sigma^* \mid \delta^*(q_0, w) = q\}$ die Menge aller Worte, die den Automaten vom Startzustand zu q überführen.[5] Dann sind je zwei Worte in W_q rechtsäquivalent.[6]
Jede Nerode-Klasse N von L ist nun gerade die Vereinigung aller W_q, die Worte aus N enthalten:[7] Enthält N ein Wort aus einem W_q, so auch alle anderen, da es ja alle zu diesem Wort rechtsäquivalenten Worte enthält.
Wir halten fest: Jede Nerode-Klasse von L ist die Vereinigung von geeigneten W_q. Folglich kann es auch nicht mehr Nerode-Klassen geben, als M Zustände hat. Folglich ist der Index von L tatsächlich endlich. □

Kommentare zum Beweis

[1] Rezept »zwei Richtungen«

[2] Ganz schön viele L's hier.

[3] Ausnahmsweise mal nicht gemäß dem Rezept »Korrektheit von Automaten«

[4] Jetzt kommt eine entscheidende Definition.

[5] Der Zusatz »aller Worte, die. . . « ist eigentlich überflüssig, Redundanz schadet aber in Beweisen nie.

[6] Sehen Sie, warum die Worte äquivalent sind?

[7] Jetzt kommt eine Begründung für diese Behauptung

9.2 Anwendungen

9.2.1 Minimierung von Automaten

9-19 Die Idee der »Minimierung von Automaten«.

- In vielen Situationen ist es nützlich, *möglichst kleine* Automaten zu haben, die Sprachen akzeptieren.
- Dabei bedeutet »klein«, dass die Automaten *wenige Zustände* haben.
- Implementiert man einen Automaten, so wächst die Programmlänge proportional zur Anzahl der Zustände. Deshalb ist weniger mehr.

▶ Definition: Minimaler Automat
Ein Automat heißt *minimal*, wenn es keinen äquivalenten Automaten mit weniger Zuständen gibt.

9-20 Die Anzahl der Zustände eines minimalen Automaten ist gleich dem Index.

▶ Satz
Sei L eine Sprache mit Index $i \in \mathbb{N}$. Dann hat der minimale Automat für L genau i Zustände.

Beweis. Im Beweis des Satzes von Myhill-Nerode wurde gezeigt, dass der Index einer Sprache immer höchstens so groß ist wie die Anzahl der Zustände eines Automaten, der die Sprache akzeptiert. Andererseits hat der Automat M_L genau i Zustände. □

Bemerkungen:

- Es ist nicht sonderlich schwierig, einen gegebenen Automaten in einen minimalen umzuwandeln.
- Die wesentliche Idee ist, für je zwei Zustände q und q' herauszufinden, ob W_q und $W_{q'}$ Teil derselbe Nerode-Klassen sind und sie in diesem Fall zu verschmelzen.

9.2.2 Nachweis der Nichtregularität

9-21  Beweisrezept: *Nichtregularität mit Nerode-Klassen*

Ziel
Man will beweisen, dass eine Sprache L nicht regulär ist.

Rezept

1. Beginne mit »Wir zeigen, dass L unendliche viele Nerode-Klassen hat.«
2. Fahre fort mit »Sei $X = \{\dots\}$. Offenbar ist X unendlich. Wir zeigen, dass keine zwei Worte in X rechtsäquivalent sind.«
3. Gib dann für je zwei $x, y \in X$ ein Wort $w \in \Sigma^*$ an, so dass $xw \in L$ und $yw \notin L$ (oder umgekehrt).

9-22 Ein einfaches Beispiel.

▶ Satz
Die Sprache $\{a^n b^n \mid n \geq 1\}$ ist nicht regulär.

Beweis. Wir zeigen, dass L unendlich viele Nerode-Klassen hat. Sei $X = \{a^n \mid n \geq 1\}$. Offenbar ist X unendlich. Wir zeigen, dass keine zwei Worte in X rechtsäquivalent sind. Seien $x = a^n$ und $y = a^m$ mit $n \neq m$. Dann gilt für $w = b^n$, dass $xw \in L$ aber $yw \notin L$. □

Zusammenfassung dieses Kapitels

1. Worte heißen *rechtsäquivalent*, wenn sie bei *Verlängerung* immer entweder beide in der Sprache sind oder beide nicht in der Sprache sind.
2. Eine *Nerode-Klasse* enthält alle Worte, die zu einem gegebenen Wort rechtsäquivalent sind.
3. Der *Index* einer Sprache ist die Anzahl seiner Nerode-Klassen.
4. Der Index einer Sprache ist gleichzeitig die Anzahl der Zustände eines minimalen Automaten für die Sprache.
5. Sprachen mit unendlichem Index sind nicht regulär.

Zum Weiterlesen

[1] R. Reischuk. *Unterlagen zum Lehrmodul Theoretische Informatik,* Vorlesungsskript, 2008. Kapitel 2.1.

Übungen zu diesem Kapitel

Übung 9.1 Nerode-Klassen, mittel

Bestimmen Sie Nerode-Klassen für folgende Sprachen:

1. $L_1 = \{a^n b^m a^{n+m} \mid n,m \in \mathbb{N}\}$
2. $L_2 = \{w \in \{a,b\}^* \mid (|w|_a - |w|_b) \bmod 3 = 0\}$
3. $L_3 = \{w \in \Sigma^* \mid w \text{ enthält eine ungerade Anzahl von Nullen}\}$
4. $L_4 = \{ww^{\text{rev}} \mid w \in \{0,1\}^*\}$.

Übung 9.2 Beweise über Nerodeklassen, mittel

Beweisen Sie mit Hilfe der Nerode-Klassen, dass $L = \{a^n b^n c^n \mid n \in \mathbb{N}\}$ nicht regulär ist.

Übung 9.3 Nerode-Klassen, mittel

Geben Sie Nerode-Klassen zu folgenden Sprachen an.

1. $L_1 = \{a^n b^n \mid n \in \mathbb{N}\}$
2. $L_2 = \{w \in \{0,1\}^* \mid w \text{ enthält den Teilstring } 01011\}$
3. $L_3 = \{ww^{\text{rev}} \mid w \in \{0,1\}^*\}$

Übung 9.4 Beweise über Nerodeklassen, mittel

Beweisen Sie mit Hilfe der Nerode-Klassen, dass $L = \{0^n 1^m 0^m \mid n \geq 1, m \in \mathbb{N}\} \cup \{1^n 0^m \mid n,m \geq 1\}$ nicht regulär ist.

10-1

Kapitel 10
Kontextfreie Grammatiken
Die Matroschkas der Theorie

10-2

Lernziele dieses Kapitels

1. Beispiele kontextfreier Sprachen kennen
2. kontextfreie Anteile von Sprachen aus der Praxis identifizieren können
3. Pumping-Lemma für kontextfreie Sprache verstehen und anwenden können
4. Normalformen für reguläre und kontextfreie Sprachen kennen

Inhalte dieses Kapitels

Worum es heute geht

Autor Wikimedia-User Fanghong, Creative Commons Attribution Sharealike

Die Sprachen, die von Grammatiken, deren Regeln immer genau ein Nonterminal auf der Seite, die links von dem Pfeil, der zwischen den Worten, die die *Regelseiten* gemäß der Sprachregelung, die in dem Skript, das Sie gerade, wo auch immer der Ort, an den Sie diese Tätigkeit gerade ausüben, sein mag, lesen, genutzt wird, genannt werden, steht, haben, erzeugt werden, sind verschachtelt.

Menschen mögen vor Nebensatzkaskaden kapitulieren – Computer hingegen halten sie für eine völlig normale Art der Kommunikation. Sätze, die jeder Lektor einem Schriftsteller gnadenlos »geradeziehen« würde, werden von Computern genüsslich nach ihrer Struktur durchforstet und dies ist bei den wichtigsten Texten, die Computer so lesen müssen, auch bitter nötig: *Programmtexte* sind in ASCII gegossene Nebensatzkathedralen. Damit Menschen überhaupt eine kleine Chance haben, der Struktur eines Programmtextes zu folgen, hat sich als sehr nützlich herausgestellt, mit Einrückungen zu arbeiten, um anzudeuten, was wozu gehört. Bekanntermaßen sind dem Compiler Ihre Einrückungen ziemlich schnurz, für ihn ist

```
for (int i = 3; i < n; i+=2) { for (int j = 3; j*j <= i;
j+=2) if (i % j == 0) break; if (j*j > i) System.out.println(i); }
```

ein völlig klares Programm. Ihnen auch?

Sprachen mit verschachtelten Worten begegnen einem in der Informatik auf Schritt und Tritt. Wie schon erwähnt sind Programmtexte hochgradig verschachtelt, jedoch auch HTML-Texte und auch allgemein XML-Texte, LaTeX-Dokumente und viele mehr.

In diesem und dem folgenden Kapitel soll es um Grammatiken gehen, mit denen wir diese Art von Sprachen gut beschreiben können: *kontextfreie Grammatiken*. Die erste wichtige Erkenntnis betreffend kontextfreie Grammatiken ist, dass sie tatsächlich mehr können als reguläre Grammatiken. Dies ist nach den vorherigen Kapiteln ja eine eher erfrischende

Feststellung, konnte man doch vielleicht den Eindruck gewinnen, Hauptaufgabe der Theoretischen Informatik sei es, möglichst viele unterschiedliche Methoden zu ersinnen, reguläre Sprachen zu beschreiben.

Ähnlich wie bei der Untersuchung von regulären Sprachen werden wir zunächst untersuchen, was kontextfreie Sprachen so alles können oder nicht können. Wie man die deskriptiven Fähigkeiten kontextfreier Grammatiken praktisch einsetzen soll sei hier erstmal zweitrangig. Im nächsten Kapitel geht es dann genau um die praktische Frage, wie man einen Parser für kontextfreie Sprachen bauen kann.

10.1 Einführung

10.1.1 Grammatiken für verschachtelte Sprachen

Kontextfreie Grammatiken beschreiben verschachtelte Sprachen. 10-4

- Mit *regulären Grammatiken* können wir schon *viele wichtige Sprachen* beschreiben.
- Jedoch *scheitern* reguläre Grammatiken bei einfachen, aber trotzdem sehr wichtigen Sprachen:
  - Palindrome,
  - arithmetische Ausdrücke (wie $(3+4)\cdot 5$),
  - komplexere Sprachen wie HTML oder Java.
- Mit *kontextfreien* Sprachen lassen sich viel mehr Sprachen beschreiben.

Das wesentliche Neue an kontextfreien Sprachen
Mit kontextfreien Sprachen lassen sich *Schachtelungen* und *Hierarchien* beschreiben.

Zwei einfache Beispiele. 10-5

Beispiel
Die Grammatik

$$\begin{aligned} G\colon S &\to (S+S) \mid (S\cdot S) \mid (S-S) \mid (S/S) \mid N \\ N &\to 0 \mid 1M \mid \cdots \mid 9M \\ M &\to 0M \mid 1M \mid \cdots \mid 9M \mid \lambda \end{aligned}$$

mit dem Terminalalphabet $T = \{0,1,\ldots,9,+,-,\cdot,/,(,)\}$ erzeugt genau die *arithmetischen Ausdrücke* wie $((3+40)/5)$.

Beispiel
Die Grammatik

$$G\colon S \to 0S0 \mid 1S1 \mid 1 \mid 0 \mid \lambda$$

erzeugt genau die Sprache PALINDROMES.

Verhältnis von regulären und kontextfreien Sprachen. 10-6

▶ **Satz**
REG $\subsetneq$ CFL.

Beweis. Da jede reguläre Grammatik kontextfrei ist, gilt REG $\subseteq$ CFL. Da die Sprache der Palindrome kontextfrei ist, aber nicht regulär nach Übung 5.1, ist REG eine *echte* Teilmenge von CFL. □

10.1.2 Praxisbeispiele

10-7

Kontextfreie Grammatiken in der Praxis

In der Praxis gibt es *viele Notationen*, wie man kontextfreie Grammatiken aufschreibt.

1. In *erweiterter Backus-Naur-Form* (EBNF).
2. Als XML-*Document-Type-Description* (DTD).
3. In *einer Ad-hoc-Syntax*.

10-8

Die erweiterte Backus-Naur-Form.

Diese Syntax wurde von Niklaus Wirth eingeführt zur *Beschreibung der Syntax von Pascal*:

- Der Regelpfeil (»→«) wird durch `::=` ersetzt.
- Terminale schreibt man in Anführungszeichen.
- Man benutzt Bezeichner (wie `Number` oder `TypeArgumentList`) für Nonterminale.
- Man kann Worte *in eckige Klammern setzen* oder in *geschweifte Klammern*:

`X ::= u[v]w`	bedeutet	$X \to uw \mid uvw$
`X ::= u{v}w`	bedeutet	$X \to uYw$
		$Y \to vY \mid \lambda$

Beispiel: Typische Regel

```
Zahl ::= ["-"] ZifferAußerNull {Ziffer} | "0"
```

10-9

Document-Type-Description.

- Die Struktur von XML-Dokumenten lässt sich durch *Document-Type-Descriptions* (DTDS) beschreiben.
- Dabei handelt es sich im Prinzip um Spezialfälle kontextfreier Grammatiken.
- Die Syntax ist etwas gewöhnungsbedürftig.

Beispiel: Typische Regeln in einem DTD

```
<!ELEMENT html (head, body)>
<!ELEMENT hr EMPTY>
<!ELEMENT div (#PCDATA | p | ul | ol | dl | table | pre |
               hr |  h1 | h2 | h3 | h4 | h5 | h6 |
               blockquote | address | fieldset)* >
<!ELEMENT dl (dt | dd)+ >
```

Dies bedeutet grob:

$$G\colon N_{html} \to \texttt{<html>} N_{head} N_{body} \texttt{</html>}$$
$$N_{hr} \to \texttt{<hr></hr>} \mid \texttt{<hr/>}$$
$$\vdots$$

10-10

Verschiedene Ad-hoc-Syntax-Formen.

Gerne werden »Ad-hoc-Notationen« für kontextfreie Grammatiken eingeführt. Hier zum Beispiel die Notation in der Java-Sprachspezifikation:

- Nonterminale werden kursiv gesetzt.
- Die linke Regelseite kommt auf eine eigene Zeile.
- Rechte Regelseiten kommen darunter, eingerückt und auf je einer eigenen Zeile.
- Der Index »opt« hat denselben Effekt wie die eckigen Klammern bei der EBNF.

Beispiel: Typische Regel

BasicForStatement:
  for (; *Expression*$_{\text{opt}}$; *ForUpdate*$_{\text{opt}}$) *Statement*
  for (*ForInit*; *Expression*$_{\text{opt}}$; *ForUpdate*$_{\text{opt}}$) *Statement*

10.2 Normalformen

10.2.1 Äquivalenz von Grammatiken

Eine Sprache -- viele Grammatiken

10-11

Ein und dieselbe Sprache kann durch viele Grammatiken erzeugt werden.

▶ Definition

Grammatiken G_1 und G_2 heißen *äquivalent*, wenn $L(G_1) = L(G_2)$.

- Genau wie bei Formeln kann man oft zu Grammatiken äquivalente finden, die besonders *einfach* oder *normal geformt* oder einfach nur *schön* sind.
- Solche Grammatiken nennt man dann *in (einer bestimmten) Normalform*.

10.2.2 Vorspiel: Reguläre Normalform

Die reguläre Normalform

10-12

▶ Definition

Eine Grammatik heißt *in regulärer Normalform*, wenn alle Regeln von einer der folgenden Arten sind:

1. $X \to a$ mit $X \in N$ und $a \in T$ oder
2. $X \to aY$ mit $X, Y \in N$ und $a \in T$.

Außerdem ist noch die λ-Startregel erlaubt.

▶ Satz

Zu jeder regulären Grammatik G gibt es eine äquivalente Grammatik G' in regulärer Normalform.

- Die Idee ist ganz einfach: Wandelt man einen DFA für G in eine reguläre Grammatik um, so erhält man automatisch eine Grammatik in regulärer Normalform (wenn man sich etwas geschickt anstellt).
- Details sind im Skript beschrieben.

Kommentare zum Beweis

Skript

Beweis. Da $L(G) \subseteq T^*$ regulär ist, gibt es einen DFA $M = (T, Q, q_0, Q_a, \delta)$ mit $L(M) = L(G)$. Diesen wandeln wir nun ähnlich wie in Satz 4-24 in eine Grammatik $G' = (T', N', S', P')$ mit $L(G') = L(M)$ um.[1]

1. $T' = T$.
2. $N' = Q$.
3. $S' = q_0$.
4. Für $\delta(q,x) = q'$ mit $q' \notin Q_a$ ist $(q \to xq') \in P'$.
5. Für $\delta(q,x) = q'$ mit $q' \in Q_a$ sind $(q \to xq') \in P'$ und $(q \to x) \in P'$.
6. Ist q_0 akzeptierend, so fügt man noch eine λ-Startregel hinzu.

Die Grammatik ist offenbar in regulärer Normalform und genau wie in Satz 4-24 zeigt man, dass $L(G') = L(M)$. Da $L(M) = L(G)$, folgt $L(G') = L(G)$. □

[1] Rezept »Konstruktiver Beweis«. Alternativ kann man auch die Grammatik direkt umwandeln. Der »Umweg« über die DFAs ist aber hier einfacher.

10.2.3 Chomsky-Normalform

10-13

Die Chomsky-Normalform

▶ **Definition**

Eine Grammatik heißt *in Chomsky-Normalform*, wenn alle Regeln von einer der folgenden Arten sind:

1. $X \to a$ mit $X \in N$ und $a \in T$ oder
2. $X \to YZ$ mit $X, Y, Z \in N$.

Außerdem ist noch die λ-Startregel erlaubt.

▶ **Satz**

Zu jeder kontextfreien Grammatik G gibt es eine äquivalente Grammatik G' in Chomsky-Normalform.

Beweis. Der Beweis verläuft in drei Schritten, die in drei Lemmas bewiesen werden:

1. Man kann Regeln (außer vielleicht der Startregel) der Form $X \to \lambda$ entfernen.
2. Man kann Regeln der Form $A \to B$ mit $A, B \in N$ entfernen.
3. Man kann Regeln der Form $A \to w$ mit $|w| \geq 2$ in Regeln in Chomsky-Normalform umwandeln. □

10-14

Die Idee hinter der Entfernung von λ-Regeln.

Der erste Schritt auf dem Weg zur Chomsky-Normalform ist, Regeln der Art $X \to \lambda$ loszuwerden (diese sind ja nicht mehr erlaubt).

✎ **Zur Diskussion**

Wie werden wir die Regel $X \to \lambda$ in folgender Grammatik los?

$$\begin{aligned} G\colon S &\to aXaXa \\ X &\to \lambda \mid bX \end{aligned}$$

- Idee: Man lässt die Regel $X \to \lambda$ einfach weg.
- Dann lassen sich aber Worte nicht mehr ableiten, in deren Ableitung diese Regel verwendet wurde.
- Der Trick ist, *neue Regeln einzuführen*, bei denen diese Regelanwendung *vorweggenommen* wurden.

Beispiel

Aus

$$\begin{aligned} G\colon S &\to 0Y0Y0 \\ Y &\to 1Y2 \mid \lambda \end{aligned}$$

wird

$$\begin{aligned} G'\colon S &\to 0Y0Y0 \mid 00Y0 \mid 0Y00 \mid 000 \\ Y &\to 1Y2 \mid 12 \end{aligned}$$

10-15

Entfernung von λ-Regeln.

▶ **Lemma**

Sei G eine kontextfreie Grammatik und $\lambda \notin L(G)$. Dann gibt es eine zu G äquivalente kontextfreie Grammatik G' ohne Regeln der Form $X \to \lambda$ mit $X \in N$.

Beweis. Sei $G = (T, N, S, P)$. Die Grammatik $G' = (T, N, S, P')$ wird wie folgt gebildet:

1. Zunächst werden alle Regeln in P nach P' übernommen.
2. Dann fügen wir nach folgender Vorschrift neue Regeln zu P' hinzu, bis es nicht mehr geht: Ist $X \to uYv$ eine Regel in P' mit $X, Y \in N$ und $u, v \in (N \cup T)^*$ und gilt $Y \Rightarrow_G^* \lambda$, so fügen wir die Regel $X \to uv$ zu P' hinzu.

3. Zum Schluss löschen wir alle Regeln der Form $X \to \lambda$ mit $X \in N$ aus P'.

Wir behaupten, dass $L(G') = L(G)$. Wir zeigen zwei Richtungen.
Ist $w \in L(G')$, so gibt es eine Ableitung $S = w_1 \Rightarrow_{G'} w_2 \Rightarrow_{G'} \cdots \Rightarrow_{G'} w_n = w$. Dann gilt auch $S \Rightarrow_G^* w$, da wir jeden Ableitungsschritt $w_i \Rightarrow_{G'} w_{i+1}$ auch mit den Regeln von G durchführen können: Wird in dem Schritt eine neue Regel $X \to uv$ angewandt, so können wir stattdessen zunächst die alte Regel $X \to uYv$ anwenden. Danach können wir wegen $Y \Rightarrow_G^* \lambda$ das Y wieder »loswerden«. Insgesamt hat dies denselben Effekt, als hätten wir gleich die Regel $X \to uv$ angewandt. Folglich gilt $w \in L(G)$.
Für die zweite Richtung sei $w \in L(G)$. Dann gilt $S \Rightarrow_G^* w$. Wir wandeln diese Ableitung wie folgt um: Wenn in der Ableitung eine Regel $X \to uYv$ angewandt wird und später das entstandene Y (eventuell in vielen Schritten) zu λ abgeleitet wird, so ersetzen wir die Regelanwendung durch $X \to uv$ und streichen alle Regelanwendungen, die sich auf das Y beziehen. Dies wiederholen wir so lange, bis es nicht mehr möglich ist. Dann wird in der entstandenen Ableitung

1. keine Regel der Form $X \to \lambda$ benutzt und
2. alle neu benutzten Regeln der Form $X \to uv$ sind Elemente von P'.

Also gilt $S \Rightarrow_{G'}^* w$. □

Die Idee hinter der Entfernung von Ersetzungsregeln.

10-16

Der zweite Schritt auf dem Weg zur Chomsky-Normalform ist, Regeln der Art $X \to Y$ loszuwerden (diese sind ja auch nicht mehr erlaubt).

✎ Zur Diskussion
Wie werden wir die Regel $X \to Y$ in folgender Grammatik los?

$$G\colon S \to aXbXc \qquad X \to Y \mid b$$
$$Y \to Z \qquad Z \to cX$$

Die Idee ist sehr ähnlich zu den λ-Regeln:

- Man lässt die Regel $X \to Y$ einfach weg.
- Dann lassen sich aber Worte nicht mehr ableiten, in deren Ableitung diese Regel verwendet wurde.
- Der Trick ist wieder, *neue Regeln einzuführen*, bei denen diese Regelanwendung *vorweggenommen* wurden.

Beispiel
Aus

$$G\colon S \to aXbXc \qquad X \to Y \mid b$$
$$Y \to Z \qquad Z \to cX$$

wird

$$G'\colon S \to aXbXc \mid aZbXc \mid aXbZc \mid aZbZc \qquad X \to b$$
$$Z \to cX \mid cZ$$

▶ Lemma
Sei G eine kontextfreie Grammatik. Dann gibt es eine zu G äquivalente kontextfreie Grammatik G' ohne Regeln der Form $X \to Y$ mit $X, Y \in N$.

Beweis. Der Beweis ist analog zu der Ersetzung von λ-Regeln mit folgender Modifikation:

- Statt $X \to uYv$ durch $X \to uv$ zu ersetzen im Fall $Y \Rightarrow^* \lambda$,
- ersetzen wir $X \to uYv$ durch $X \to uZv$ im Fall $Y \Rightarrow^* Z$. □

10-17

Das Aufbrechungslemma.

▶ **Lemma**
Sei G eine kontextfreie Grammatik ohne Regeln der Form $X \to \lambda$ oder $X \to Y$ mit $X, Y \in N$. Dann gibt es eine zu G äquivalente Grammatik G' in Chomsky-Normalform.

Beweis. Wir beweisen dies in zwei Schritten:

1. Wir ersetzen Regeln der Form $X \to w$ mit $|w| \geq 3$ durch folgende Folge von Regeln:

$$\begin{aligned} X &\to w[1]X_1 \\ X_1 &\to w[2]X_2 \\ &\vdots \\ X_{|w|-2} &\to w[|w|-1]w[|w|] \end{aligned}$$

   Hierbei sind die X_i neue Nonterminale.
2. Wir ersetzen Regeln der Form $X \to ab$ mit $a, b \in T$ durch

$$\begin{aligned} X &\to AB \\ A &\to a \\ B &\to b \end{aligned}$$

   wobei A und B wieder neuer Nonterminale sind. Ähnlich verfährt man mit Regeln der Form $X \to aB$ oder $X \to Ab$ mit $a, b \in T$ und $A, B \in N$.

Die Korrektheit zeigt man dann durch Induktion. □

10-18

Ein Beispiel: Umwandlung einer Grammatik in Chomsky-Normalform

1. Die Ausgangsgrammatik.
2. Entfernung der λ-Regel durch »Vorwegnahme« ihrer Anwendung.
3. Entfernung der Regel $S \to S$ durch »Vorwegnahme« ihrer Anwendung.
4. Aufbrechen von zu langen Regeln.
5. Ersetzen von Terminalen durch Nonterminale.

$$\begin{aligned} G_1 &: S \to 0S0 \mid 1S1 \mid S \mid \lambda \\ G_2 &: S \to \lambda \mid S' \\ &\quad S' \to 0S'0 \mid 1S'1 \mid 00 \mid 11 \mid S' \\ G_3 &: S \to \lambda \mid S' \\ &\quad S' \to 0S'0 \mid 1S'1 \mid 00 \mid 11 \\ G_4 &: S \to \lambda \mid S' \\ &\quad S' \to 0A \mid 1B \mid 00 \mid 11 \\ &\quad A \to S'0 \\ &\quad B \to S'1 \\ G_5 &: S \to \lambda \mid S' \\ &\quad S' \to NA \mid EB \mid NN \mid EE \\ &\quad A \to S'N \\ &\quad B \to S'E \\ &\quad N \to 0 \\ &\quad E \to 1 \end{aligned}$$

10.3 Grenzen kontextfreier Sprachen

Grenzen kontextfreier Grammatiken.

10-19

- Anders als reguläre Grammatiken kann man sich bei kontextfreien Grammatiken *»etwas merken«*.
- Beispielsweise kann man dafür sorgen, dass die Anzahl a's am Anfang gleich der Anzahl b's am Ende ist.
- Jedoch werden wir gleich sehen, dass sich *auch nicht jede Sprache* erzeugen lässt.
- Wir werden dies mithilfe *eines neuen Pumping-Lemmas* beweisen, treffend das *Pumping-Lemma für kontextfreie Sprachen* genannt.

Vorüberlegung: Der Syntaxbaum.

10-20

▶ Definition: Syntaxbaum

Sei G eine Grammatik in Chomsky-Normalform. Sei $S = w_1 \Rightarrow \cdots \Rightarrow w_n = w$ eine Ableitung. Den *Syntaxbaum* dieser Ableitung erhält man wie folgt:

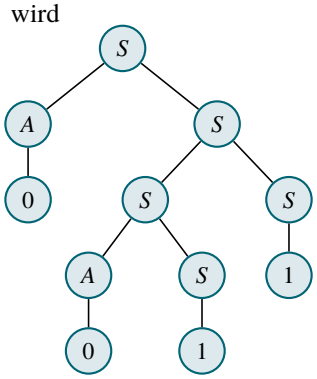
1. Zunächst besteht der Baum nur aus einem Wurzelknoten, markiert mit dem Startsymbol.
2. Wenn eine Regel $X \to YZ$ angewandt wird in der Ableitung, werden an dem zu X gehörigen Blattknoten zwei Kinder angefügt, die mit Y und Z markiert werden.
3. Wenn eine Regel $X \to a$ angewandt wird in der Ableitung, wird an X ein einziges mit a markiertes Blatt angefügt.

Beispiel

Aus

$$S \Rightarrow AS \Rightarrow ASS \Rightarrow 0SS \Rightarrow 0ASS \Rightarrow 0AS1 \Rightarrow 00S1 \Rightarrow 0011$$

wird



Eine alte Idee: in langen Worten kann man Teile aufpumpen.

10-21

Sei G eine kontextfreie Grammatik und $L = L(G)$.

- Sei $S \Rightarrow^* w$.
- Wenn w nur lang genug ist, dann müssen einige Nonterminale in der Ableitung *sehr häufig* vorkommen.
- Insbesondere muss irgendwann mal im Syntaxbaum ein Nonterminal A auftauchen, so unterhalb des Knotens A wieder auftaucht. Dann gilt
  - zunächst $S \Rightarrow^* xAz$,
  - dann $A \Rightarrow^* aAb$ und schließlich
  - $A \Rightarrow^* y$.

  Dabei ist $A \in N$ und $x, y, z, a, b \in (N \cup T)^*$.
- Dann kann aber auch »noch einmal« die Ableitung $A \Rightarrow^* aAb$ »hineinschmuggeln«: $A \Rightarrow^* aAb \Rightarrow^* aaAbb$.
- Das kann man beliebig oft machen.
- Also kann man beliebig Worte der Bauart xa^iyb^iz ableiten.

10.3.1 Der Satz und sein Beweis

10-22

Das Pumping-Lemma für kontextfreie Sprachen.

▶ **Satz**
Sei L kontextfrei. Dann

- *existiert eine Wortlänge $n \in \mathbb{N}$, so dass*
- *für alle Worte $w \in L \cap \Sigma^{\geq n}$ gilt, es*
- *existiert eine Zerlegung $w = x \circ a \circ y \circ b \circ z$ mit $|ab| \geq 1$, so dass*
- *für alle $i \in \mathbb{N}$ gilt*

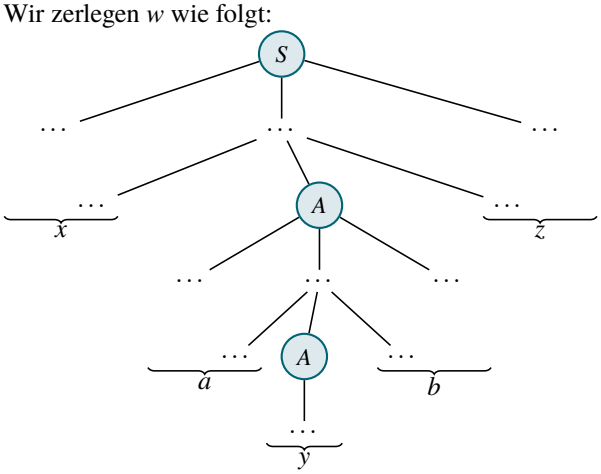
$$x \circ a^i \circ y \circ b^i \circ z \in L.$$

Bemerkungen:

- Der Satz ist sehr ähnlich zum Pumping-Lemma für reguläre Sprachen.
- Der *einzige Unterschied* ist, dass nicht nur an einer Stelle gepumpt wird (xy^iz), sondern *an zwei Stellen synchron* ($x \circ a^i \circ y \circ b^i \circ z$).

Kommentare zum Beweis

Beweis. Sei L kontextfrei. Dann existiert eine Grammatik G in Chomsky-Normalform mit $L(G) = L$. Sei $n = 2^{|N|+1}$.[1]
Sei nun $w \in L \cap \Sigma^{\geq n}$ beliebig.[2] Sei T der Syntaxbaum der Ableitung $S \Rightarrow_G^* w$. Sei b das Blatt von T maximaler Tiefe. Dann liegen auf dem Weg von der Wurzel von T zu b mindestens $|N| + 1$ Nonterminale. Insbesondere taucht ein Nonterminal A *doppelt auf*.[3]
Wir zerlegen w wie folgt:



Dann gilt:[4]

1. $S \Rightarrow^* xAz$,
2. $A \Rightarrow^* aAb$,
3. $A \Rightarrow^* y$,

wobei jeweils $x, y, z, a, b \in T^*$ und $|ab| \geq 1$.
Dann gilt für jedes i, dass[5]

$$S \Rightarrow^* xAz \Rightarrow^* xaAbz \Rightarrow^* xa^2Ab^2z \Rightarrow^* \ldots$$
$$\Rightarrow^* xa^iAb^iz \Rightarrow^* xa^iyb^iz.$$

Also gilt, wie behauptet, $xa^iyb^iz \in L$. □

[1] Man beachte, dass n recht groß sein muss.
[2] Rezept »All-Aussagen«
[3] Jetzt wird zerlegt.
[4] Das müsste man jetzt eigentlich beweisen, »Beweis durch Bild« geht normalerweise nicht.
[5] Hier wird gepumpt.

10-23

Folgerung aus dem Pumping-Lemma

- *Genau wie beim Pumping-Lemma für reguläre Sprachen* interessiert uns eigentlich das Lemma selber nicht.
- Vielmehr interessiert uns seine *Kontraposition*.

▶ Folgerung

Sei L eine Sprache, so dass

- *für alle Wortlängen $n \in \mathbb{N}$*
- *existiert ein Wort $w \in L \cap \Sigma^{\geq n}$, so dass*
- *für alle Zerlegungen $w = xaybz$ mit $|ab| \geq 1$*
- *existiert ein $i \in \mathbb{N}$ mit*

$$xa^iyb^iz \notin L.$$

Dann ist L nicht kontextfrei.

Beweisrezept: *Kontextfreies Pumping-Lemma anwenden* 10-24

Ziel

Man will beweisen, dass eine Sprache L nicht kontextfrei ist.

Rezept

1. Beginne mit »Wir zeigen die Behauptung durch die Kontraposition des Pumping-Lemmas.«
2. Fahre fort mit »Sei n beliebig.«
3. Fahre fort mit »Dann ist $w = \ldots$ ein Element von $L \cap \Sigma^{\geq n}$.«, wobei natürlich »…« geeignet gewählt sein muss.
4. Fahre fort mit »Sei nun $w = xaybz$ eine beliebige Zerlegung mit $|ab| \geq 1$.«
5. Schließe mit »Wähle nun $i = \ldots$« und argumentiere, dass $xa^iyb^iz \notin L$.

Beispiel einer Anwendung der Folgerung. 10-25

▶ Satz

Die Sprache $\{0^n1^n2^n \mid n \geq 1\}$ ist nicht kontextfrei.

Beweis. Wir zeigen die Behauptung durch die Kontraposition des Pumping-Lemmas.[1] Sei n beliebig. Dann ist $w = 0^n1^n2^n$ ein Element von $L \cap \Sigma^{\geq n}$. Sei nun $w = xaybz$ eine beliebige Zerlegung mit $|ab| \geq 1$.
Für $i = 2$ gilt nun, dass, egal wie die Zerlegung gewählt war,[2] immer $xa^2yb^2z \notin L$ gilt. □

Kommentare zum Beweis

[1] Text aus dem Rezept

[2] Hier gibt es viele Fälle, die jetzt nicht alle aufgelistet werden.

10.3.2 Die Charakterisierungen im Überblick

Die Charakterisierungen im Überblick 10-26

Sei L eine Sprache. Es gelten folgende Implikationen:

1. »L ist regulär« $\iff$ »L hat endlichen Index«
2. »L ist regulär« $\implies$ »L hat reguläre Pump-Eigenschaft«
3. »L hat nicht reguläre Pump-Eigenschaft« $\implies$ »L ist nicht regulär«
4. »L ist kontextfrei« $\implies$ »L hat kontextfreie Pump-Eigenschaft«
5. »L hat nicht kontextfreie Pump-Eigenschaft« $\implies$ »L ist nicht kontextfrei«

Zusammenfassung dieses Kapitels

1. Kontextfreie Sprachen werden in der Praxis viel benutzt, was zu einer *reichhaltigen Syntaxvielfalt* geführt hat. 10-27
2. Kontextfreie Grammatiken beschreiben *verschachtelte Strukturen* und können *echt mehr als reguläre Grammatiken*.
3. Auch kontextfreie Sprachen haben eine Pump-Eigenschaft – aber eine *andere* als reguläre.

Zum Weiterlesen

[1] R. Reischuk. *Unterlagen zum Lehrmodul Theoretische Informatik,* Vorlesungsskript, 2008. Kapitel 3.1.

Übungen zu diesem Kapitel

Übung 10.1 Nicht-Abgeschlossenheit unter Schnitt, mittel

Zeigen Sie, dass die Klasse der kontextfreien Sprachen nicht unter Schnitt abgeschlossen ist.
Tipp: Betrachten Sie $\{a^n b^n c^m \mid n,m \geq 0\}$ und $\{a^n b^m c^m \mid n,m \geq 0\}$.

Übung 10.2 Nicht-Abgeschlossenheit unter Komplement, leicht

Zeigen Sie, dass die Klasse der kontextfreien Sprachen nicht unter Komplement abgeschlossen ist.
Tipps: Sie dürfen Übungen 3.4 und 10.1 benutzen. Sie dürfen benutzen, dass $\bar{A} \cup \bar{B} = \overline{A \cap B}$. Führen Sie einen Widerspruchsbeweis.

11-1

Kapitel 11
Analyse kontextfreier Sprachen
Automaten mit Leichen im Keller

11-2

Lernziele dieses Kapitels

1. Den Algorithmus von Cocke, Younger und Kasami anwenden können
2. Syntax und Semantik von Pushdown-Automaten verstehen
3. Äquivalenz zu kontextfreien Grammatiken kennen
4. Konzept der deterministisch kontextfreien Grammatik verstehen

Inhalte dieses Kapitels

Worum es heute geht

Nachdem im letzten Kapitel die prinzipielle Fähigkeiten, aber auch die prinzipiellen Grenzen von kontextfreien Sprachen ausgelotet wurden, soll es nun um die praktische Frage gehen, wie das Parsing für kontextfreie Sprache funktioniert.

Die erste beruhigende Erkenntnis ist, dass es überhaupt prinzipiell *möglich* ist, jede beliebige kontextfreie Sprache zu parsen. Tatsächlich ist die Lage noch um einiges erfreulicher: Mit Hilfe des CYK-Algorithmus lässt sich sogar vergleichsweise *effizient* überprüfen, ob und wenn ja wie sich ein Wort aus einer gegebenen kontextfreien Grammatik erzeugen lässt.

Bild von Jürgen Schoner, Creative Commons Attribution Sharealike Lizenz

Leider ist der CYK-Algorithmus nur »vergleichsweise« effizient. Es kommt darauf an, womit man ihn vergleicht. Im Vergleich zu Algorithmen für die Travelling Salesperson Problem oder für SAT ist der CYK-Algorithmus tatsächlich rasend schnell. Verglichen mit einem Such- oder Sortiertalgorithmus hingegen gleicht der CYK-Algorithmus eher ein Weinbergschnecke: Er hat eine Laufzeit von grob $O(n^3)$ bei Wörtern der Länge n. Ist nun das »Wort«, das vom Parser verarbeitet werden soll, ein Programmtext, so kann n schnell mal in der Größenordnung einer Million liegen. In solch einem Fall ist $O(n^3)$ leider keine akzeptable Laufzeit mehr; niemand möchte ca. 30 Jahre darauf warten, dass ein Compiler die Meldung »Semicolon expected in line 123456« produziert (eine, nebenbei bemerkt, besonders perfide Fehlermeldung, da der Compiler ja ganz genau weiß, was das Problem ist, und auch, wie man es lösen könnte – er tut es nur einfach nicht).

Eine zweite Idee ist, eine ähnliche Methodik wie bei regulären Grammatiken anzuwenden. Dort haben wir auch ausgehend von den Grammatiken ein *Maschinenmodell* entworfen (die endlichen Automaten), mit deren Hilfe wir dann die regulären Sprachen sehr effizient verarbeiten konnten. Zu den kontextfreien Grammatiken gibt es auch ein passendes Maschinenmodell: Kellerautomaten (englisch Push-Down-Automata, PDAs). Genau wie normale Automaten verarbeiten auch PDAs ihre Eingaben sehr effizient. Allerdings passt das Modell auch nicht perfekt: Nicht je kontextfreie Sprache lässt sich von einem deterministischen PDA akzeptieren. Es ist deshalb in der Praxis recht wichtig, sich auf solche Grammatiken zu beschränken, wo dies dann doch möglich ist.

11.1 Der CYK-Algorithmus

11.1.1 Die Idee

11-4

Eine typische Situation und ein einfaches Ziel.

Ausgangssituation

- Eine kontextfreie Grammatik G (zum Beispiel für HTML) ist *fest vorgegeben*.
- Für verschiedene Eingabeworte $w \in \Sigma^*$ wollen wir herausfinden, ob $w \in L(G)$ gilt.

Die zentrale Idee beim CYK-Algorithmus

- Wir versuchen *nicht*, direkt eine Ableitungskette zu finden.
- Vielmehr werden wir für *immer größere Teilworte* herausfinden, *aus welchen Nonterminalen sich diese ableiten lassen*.

11-5

Die Idee der Wortblöcke.

Folgende Grammatik erzeugt gerade $\{ww^{\text{rev}} \mid w \in \{0,1\}^{\geq 1}\}$:

$$G\colon S \to NA \mid EB \mid NN \mid EE$$
$$A \to SN \qquad B \to SE$$
$$N \to 0 \qquad E \to 1$$

Sei nun $w = 0011011001$.

- Wie betrachten *Teilworte* von w.
- Zum Beispiel betrachten wir $w[3]w[4]w[5] = 110$.
- Aus welchen Nonterminalen lässt sich 110 ableiten? (Richtige Antwort: nur A, nämlich über $A \Rightarrow SN \Rightarrow EEN \Rightarrow^* 110$).

✎ Zur Übung

Geben Sie für folgende Teilworte an, aus welchen Nonterminalen sie sich ableiten lassen:

1. $w[2]$
2. $w[2]w[3]w[4]w[5]$
3. $w[2]w[3]$

11-6

Nonterminale, die Wortteile erzeugen.

▶ Definition

Sei $G = (T, N, S, P)$ eine kontextfreie Grammatik und $w \in T^*$ ein Wort. Die Menge $N_{i,j} \subseteq N$ enthält alle Nonterminale, aus denen sich das Teilwort von w ableiten lässt, das zwischen der i-ten und der j-ten Stelle liegt:

$$N_{i,j} = \{X \in N \mid X \Rightarrow_G^* w[i]w[i+1]\cdots w[j]\}.$$

11-7

Die zentrale Idee: Wie man Wortblöcke vereinigt.

- Nehmen wir an, wir haben für *alle Teilworte von w, aber nicht für w als Ganzes* herausgefunden, aus welchen Nonterminalen sich diese Teilworte ableiten lassen.
- Nun wollen wir herausfinden, aus welchen Nonterminalen sich *w als Ganzes* ableiten lässt.

Idee

Ein Wort w lässt sich aus einem Nonterminal X ableiten genau dann, wenn entweder

- w einfach ein einziges Zeichen ist und $X \to w$ eine Regel in P ist oder
- $X \to YZ$ eine Regel ist und sich *w in zwei Teile u und v zerlegen lässt (also $w = uv$)*, so dass *sich u aus Y ableiten lässt und v aus Z*.

Die Idee der Wortzerlegung. 11-8

Betrachten wir nochmal

$$G\colon S \to NA \mid EB \mid NN \mid EE$$
$$A \to SN \qquad B \to SE$$
$$N \to 0 \qquad E \to 1$$

Sei nun $w = 101101$. Dann gilt:

- Wir können w in zwei Teile zerlegen, nämlich in
  - $u = 1$ und
  - $v = 01101$,

  so dass
- sich u aus E ableiten lässt und
- sich v aus B ableiten lässt (über $B \Rightarrow SE \Rightarrow NAE \Rightarrow NSNE \Rightarrow NEENE \Rightarrow^* 01101$).

Da es die Regel $S \to EB$ gibt, folgt, dass sich w aus S ableiten lässt.

11.1.2 Der Algorithmus

Die Idee hinter dem Algorithmus von Cocke, Younger und Kasami. 11-9

- Ausgangspunkt ist eine kontextfreie Grammatik in Chomsky-Normalform.
- Weiter sei ein Wort gegeben.
- Zunächst bestimmt man *für jeden einzelnen Buchstaben*, aus welchen Nonterminalen dieser produziert werden kann.
- Dann berechnet man für *jedes Teilwort der Länge 2*, aus welchen Nonterminalen dieses produziert werden kann.
- Dann berechnet man für *jedes Teilwort der Länge 3*, aus welchen Nonterminalen dieses produziert werden kann.
- Entsprechend fährt man fort, bis man berechnet hat, *aus welchen Nonterminalen w produziert werden kann*.
- Ist S bei der letzten Menge dabei, dann lässt sich w ableiten, sonst eben nicht.

Die Grundbegriffe des CYK-Algorithmus etwas formaler. 11-10

▶ Definition: Splits einer Regel

Sei $X \to YZ$ eine Regel der Grammatik G, sei w ein Wort und seien i und j Indizes. Ein *Split des Wortes w für die Regel $X \to YZ$ und i und j* ist eine Stelle k mit $i \leq k < j$, so dass $Y \in N_{i,k}$ und $Z \in N_{k+1,j}$.

▶ Lemma

Sei G eine kontextfreie Grammatik in Chomsky-Normalform und sei w ein Wort und i und j seien Indizes. Dann gilt $X \in N_{i,j}$ genau dann, wenn es einen Split von w für eine Regel $X \to YZ$ aus G und die Indizes i und j gibt.

Beweis. Dies folgt unmittelbar aus den Definitionen. □

Beispiel des CYK-Algorithmus in Aktion. 11-11

$$G\colon S \to NA \mid EB \mid NN \mid EE$$
$$A \to SN \qquad B \to SE$$
$$N \to 0 \qquad E \to 1$$

Sei $w = 0110$. Dann sind die Werte von $N_{i,j}$ gerade:

$i=$	$j=1$	$j=2$	$j=3$	$j=4$
1	$\{N\}$	$\emptyset$	$\emptyset$	$\{S\}$
2		$\{E\}$	$\{S\}$	$\{A\}$
3			$\{E\}$	$\emptyset$
4				$\{N\}$

11-12

✎ Zur Übung

Führen Sie den CYK-Algorithmus für folgende Grammatik durch, indem Sie die Tabelle füllen:

$$\begin{aligned} G\colon S &\to AB \mid CD \\ A &\to 0 \\ B &\to 1 \\ C &\to AB \\ D &\to BA \end{aligned}$$

Sei $w = 0110$. Dann sind die Werte von $N_{i,j}$ gerade:

$i=$	$j=1$	$j=2$	$j=3$	$j=4$
1	$\{A\}$			
2		$\{B\}$		
3			$\{B\}$	
4				$\{A\}$

11-13

Der CYK-Algorithmus

```
1  input Grammatik G = (T,N,S,P) in Chomsky-Normalform
2  input w ∈ T* mit |w| = n
3
4  // Erstmal alles leer
5  for 1 ≤ i ≤ j ≤ n do     N_{i,j} ← ∅
6
7  // Initialisiere N_{i,i}
8  for i ← 1 to n do     N_{i,i} ← {X | (X → w[i]) ∈ P}
9
10 for ℓ ← 1 to n do
11   // Berechne N_{i,j} für alle Teilworte der Länge ℓ
12   for i ← 1 to n − ℓ do
13     j ← i + ℓ
14     // Gehe alle möglichen Regeln durch:
15     foreach (X → YZ) ∈ P do
16       // Gehe alle möglichen Splits durch:
17       for k ← i to j − 1 do
18         if Y ∈ N_{i,k} ∧ Z ∈ N_{k+1,j} then
19           N_{i,j} ← N_{i,j} ∪ {X}
20
21 if S ∈ N_{1,n} then return »w ∈ L(G)« else return »w ∉ L(G)«
```

11-14

Satz über den CYK-Algorithmus

▶ Satz

Der CYK*-Algorithmus ist korrekt. Seine Laufzeit ist* $O(|w|^3 \cdot |P|)$.

Beweis. Die Korrektheit sollte aus den Vorüberlegungen klar sein.[1]
Die Abschätzung der Laufzeit ergibt sich unmittelbar daraus, dass es vier verschachtelte For-Schleifen gibt, wobei die ersten drei maximal je $|w|$ oft durchlaufen werden, die letzte genau $|P|$ Mal. □

Kommentare zum Beweis

[1] Das bedeutet, dass der Autor keine Lust hatte, einen Beweis aufzuschreiben.

11.2 Kellerautomaten

Die Idee hinter Kellerautomaten.

11-15

- Der CYK-Algorithmus ist *in der Praxis viel zu langsam*.
- Soll ein *Automat* eine kontextfreie Sprache parsen, geht man zunächst wie bei der Umwandlung von regulären Grammatiken in NFAs vor.
- Bei einer Regel wie $R \to aXb$ ergeht es dem Automaten dann aber in etwa so:

  Wohl an, hier bin ich nun im Zustand R.
  Es kommt ein a. Welch' Freud',
  habe ich doch die Regel hierfür.
  Hinweg mit dem a, der neue Zustand fein
  das X soll's sein.
  Aber, ach, welch' Gram!
  das b – merken soll ich's mir.
  (leise) Was tun? Was tun?

- Die Lösung: Der Automat bekommt noch einen *Keller* (englisch Stack), auf dem er sich *merken kann*, was er später wieder lesen muss.

11.2.1 Syntax

Syntax eines Kellerautomaten

11-16

- Ein *Kellerautomat* hat neben der Eingabe noch einen *Keller*.
- In jedem Schritt liest er *neben* einem Eingabezeichen *zusätzlich* auch noch das oberste Zeichen von Keller.
- Dann kann er drei mögliche Dinge machen:
  - Er entfernt das oberste Zeichen vom Keller (»Pop«).
  - Er ändert den Stack nicht (»Idle«).
  - Er packt ein weiteres Zeichen oben auf den Keller (»Push«).

Deterministische und Nichtdeterministische Kellerautomaten.

11-17

▶ Definition: Deterministische Kellerautomaten
Ein *deterministischer Kellerautomat* (englisch *deterministic push-down automaton*, DPDA) besteht aus

- einem *Eingabealphabet* Σ,
- einem *Kelleralphabet* Γ *mit* $\square \in \Gamma$,
- einer endlichen Menge Q *von Zuständen*,
- einem *Startzustand* $q_0 \in Q$,
- einer Menge $Q_a \subseteq Q$ *von akzeptierenden Zuständen* und
- einer *Zustandsübergangsfunktion*

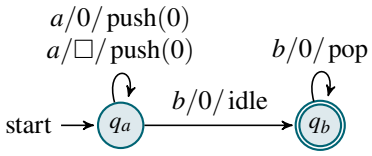
$$\delta\colon \underbrace{Q}_{\text{alter Zustand}} \times \underbrace{\Sigma}_{\text{Eingabesymbol}} \times \underbrace{\Gamma}_{\text{oberstes Kellersymbol}} \to \underbrace{Q}_{\text{neuer Zustand}} \times \underbrace{\big(\{\text{pop}, \text{idle}\} \cup \{\text{push}(x) \mid x \in \Gamma \setminus \{\square\}\}\big)}_{\text{was mit dem Stack passiert}}.$$

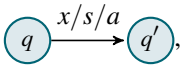
▶ Definition: Nichtdeterminitische Kellerautomaten
Ein *nichtdeterministischer Kellerautomat mit λ-Übergängen* (NPDA) ist genauso definiert wie ein DPDA, nur ist die Zustandsübergangsfunktion ersetzt durch eine Zustandsübergangsrelation:

$$\Delta \subseteq \big(Q \times (\Sigma \cup \{\lambda\}) \times \Gamma\big) \times \big(Q \times \big(\{\text{pop}, \text{idle}\} \cup \{\text{push}(x) \mid x \in \Gamma \setminus \{\square\}\}\big)\big).$$

11-18

Beispiel eines Kellerautomaten



- Bei einem Kellerautomaten muss man in jedem Zustand zu jedem gelesenen Zustand auch noch angeben, was das oberstes Stacksymbol sein soll und was mit dem Stack passieren soll.
- Formal bedeutet 

, dass $\big((q,x,s),(q',a)\big) \in \Delta$ gilt.

11.2.2 Semantik

11-19

Konfigurationen von Kellerautomaten.

▶ Definition: Konfiguration eines Kellerautomaten

Eine *Konfiguration* eines Kellerautomaten ist ein Element der Menge $Q \times \Sigma^* \times \Gamma^*$. Die *Anfgangskonfiguration* für ein Eingabewort w ist $(q_0, w, \square)$.

Bemerkungen:

- Zur Erinnerung: Eine Konfiguration ist ein *Schnappschuss* einer Berechnung.
- In einer Konfiguration (q,w,k) ist
  - q der aktuelle Zustand des Automaten,
  - w der noch zu lesende Rest der Eingabe,
  - k ist der Keller, wobei das erste Zeichen des Kellers »oben« ist und »ganz unten« anfänglich »□« steht.

11-20

Berechnungsschritt

▶ Definition

Sei M ein NPDA und seien (q,xw,sr) und $(q',w,s'r)$ zwei Konfigurationen mit $x \in \Sigma \cup \{\lambda\}$, $w \in \Sigma^*$, $s \in \Gamma$ und $s', r \in \Gamma^*$. Dann gilt $(q,w,sr) \vdash (q',w,s'r)$, wenn

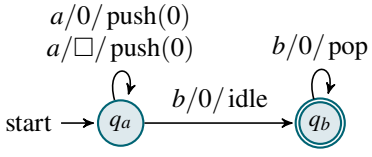
- $\big((q,x,s),(q',\text{pop})\big) \in \Delta$ und $s' = \lambda$,
- $\big((q,x,s),(q',\text{idle})\big) \in \Delta$ und $s' = s$ oder
- $\big((q,x,s),(q',\text{push}(y))\big) \in \Delta$ und $s' = ys$.

▶ Definition

Die Begriffe *akzeptierende Berechnung*, *akzeptiertes Wort* und *akzeptierte Sprache* sind wie üblich definiert.

11-21

Beispiel einer Berechnung



Dann gilt:

$$\begin{aligned}(q_a, aabb, \square) &\vdash_M (q_a, abb, 0\square)\\ &\vdash_M (q_a, bb, 00\square)\\ &\vdash_M (q_b, b, 00\square)\\ &\vdash_M (q_b, \lambda, 0\square)\end{aligned}$$

Somit würde *aabb* akzeptiert.

✎ Zur Übung

1. Wie lautet $L(M)$?
2. Geben Sie einen Kellerautomaten für die Sprache der Palindrome an.

11.2.3 Mächtigkeit

Kellerautomaten akzeptieren genau die kontextfreien Sprachen. 11-22

▶ Satz
Eine Sprache L ist genau dann kontextfrei, wenn sie von einem (eventuell nichtdeterministischen) Kellerautomaten akzeptiert wird.

Idee
Wir zeigen zwei Richtungen. Die Ideen für die erste Richtung:

1. Sei L kontextfrei. Dann gibt es eine kontextfreie Grammatik G für L.
2. Verfahre nun wie bei der Umwandlung von regulären Grammatiken in NFAs.
3. *Jedoch:* Benutze den Keller, um zu merken, welche Zeichen am Ende »noch kommen müssen«.

Die Ideen für die zweite Richtung:

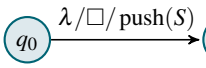
1. Sei M ein Kellerautomat.
2. Die Nonterminale der gesuchten Grammatik *entsprechen Berechnungsketten*.

Kommentare zum Beweis

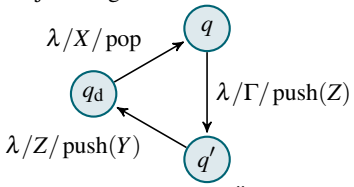
Beweis. Wir zeigen zwei Richtungen. Für die erste Richtung sei L eine kontextfreie Sprache. Dann gibt es eine kontextfreie Grammatik $G = (T, N, S, P)$ mit $L = L(G)$. Wir dürfen annehmen,[1] dass G in Chomsky-Normalform ist.
Wir bauen einen nichtdeterministischen Kellerautomaten wie folgt: Das Eingabealphabet ist T. Das Kelleralphabet ist N. Die Zustandsmenge lautet wie folgt:

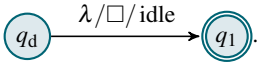
- Es gibt den Startzustand q_0.
- Es gibt einen speziellen Zustand q_d, genannt der *Dispatcher-Zustand*.
- Für jede Regel $X \to YZ$ gibt es je zwei Hilfszustände.
- Schließlich gibt es noch einen speziellen, akzeptierenden Zustand q_1.

Die Zustandsübergänge von M sind wie folgt:

- Es gibt den Übergang 

.
- Für jede Regel $X \to x$ mit $x \in T$ gibt es den Übergang 

.
- Für jede Regel $X \to YZ$ mit den Hilfszuständen q und q' gibt es die Übergänge


Dabei steht Γ bei einem Übergang dafür, dass hier jedes beliebiges Zeichen stehen darf.
- Schließlich gibt es noch den Übergang 

.

Aufgrund der Definitionen überzeugt man sich leicht, dass $(q_\mathrm{d}, xw, Xs) \vdash (q_\mathrm{d}, w, s)$ genau dann gilt, wenn $X \to x$ eine Regel in P ist. Ähnlich überzeugt man sich leicht, dass $(q_\mathrm{d}, w, Xs) \vdash^* (q_\mathrm{d}, w, YZs)$, wobei in der Berechnung nur am Anfang und am Ende q_d betreten wird, genau dann gilt, wenn $X \to YZ$ eine Regel in P ist.
Wir behaupten, dass $L(M) = L$ gilt. Sei zunächst $w \in L$. Dann gibt es eine Linksableitung $S = w_1 \Rightarrow \cdots \Rightarrow w_n = w$ (also eine Ableitung, bei der immer das am weitesten links stehende Nonterminal ersetzt wird). Jedes w_i lässt sich zerlegen in $w_i = u_i v_i$ mit $u_i \in T^*$ und $v_i \in N^*$.[2] Der Automat kann dann w wie folgt akzeptieren: Er wechselt zunächst nach q_d. Damit erreicht der die Konfiguration $(q_\mathrm{d}, w, S\square) = (q_\mathrm{d}, w[|u_1|+1]\ldots w[|w|], v_1\square)$. Nun zeigt man durch Induktion über i, dass er für jedes i die Konfiguration $(q_\mathrm{d}, w[|u_i|+1]\ldots w[|w|], v_i\square)$ erreichen kann:

- Wird beim Schritt $u_i v_i \Rightarrow u_{i+1} v_{i+1}$ die Regel $X \to x$ angewandt, dann muss u_{i+1} gerade auf x enden und v_i mit X beginnen. Also gilt $(q_\mathrm{d}, w[|u_i|+1]\ldots w[|w|], v_i\square) = (q_\mathrm{d}, xw[|u_{i+1}|+1]\ldots w[|w|], Xv_{i+1}\square)$. Dann ist aber klar, dass nun $(q_\mathrm{d}, w[|u_{i+1}|+1]\ldots w[|w|], v_{i+1}\square)$ in einem Schritt erreichbar ist.
- Wird die Regel $X \to YZ$ angewandt, so überlegt man sich ähnlich, dass $v_i = Xv$ und $v_{i+1} = YZv$ gilt und dann

$$\begin{aligned}(q_\mathrm{d}, w[|u_i|+1]\ldots w[|w|], v_i\square) &= (q_\mathrm{d}, w[|u_i|+1]\ldots w[|w|], Xv\square)\\ &\vdash^* (q_\mathrm{d}, w[|u_i|+1]\ldots w[|w|], YZv\square)\\ &= (q_\mathrm{d}, w[|u_{i+1}|+1]\ldots w[|w|], v_{i+1}\square).\end{aligned}$$

[1] Man braucht die Chomsky-Normalform hier nicht unbedingt, es macht den Beweis aber etwas einfacher.

[2] Dies liegt daran, dass G in Chomsky-Normalform ist.

Skript

Hieraus folgt, dass der Automat die Konfiguration $(q_d, \lambda, \square)$ erreichen kann und damit auch den akzeptierenden Zustand.
Für die zweite Richtung sei nun ein beliebiges $w \in L(M)$ gegeben. Dann gibt es eine akzeptierende Berechnung für w. Aufgrund des Aufbaus des Automaten muss diese im ersten Schritt das Symbol S in den Keller schreiben. Weiter muss am Ende der Berechnung der Stack leer sein und das Wort komplett gelesen sein. Im Hauptteil der Berechnung muss der Automat immer wieder den Zustand q_d durchschreiten. Bei jedem Durchschreiten sei jeweils u_i das bereits gelesene Wort und v_i der Stackinhalt (ohne $\square$). Dann zeigt man ähnlich wie bei der ersten Beweisrichtung, dass $u_i v_i \Rightarrow u_{i+1} v_{i+1}$ für alle i gilt. Daraus ergibt sich aber mit $u_1 = \lambda$ und $v_1 = S$ und $u_n = w$ und $v_1 = \lambda$, dass $S \Rightarrow^* w$ gilt.
Wir zeigen nun noch, dass jede von einem Kellerautomaten akzeptierte Sprache kontextfrei ist. Sei dazu $M = (\Sigma, \Gamma, Q, q_0, Q_a, \Delta)$ ein NPDA. Falls nötig, modifizieren wir den Automaten (durch Einfügen von geeigneten λ-Schritten und Schleifen), so dass

1. es nur einen akzeptierenden Zustand q_a gibt, also $Q_a = \{q_a\}$, und
2. der Stack immer leer ist, wenn ein Wort akzeptiert wird.

Die Grammatik $G = (\Sigma, N, S, P)$ ist dann wie folgt gebaut. Der Menge $N = Q \times \Gamma \times Q$ liegt folgende Idee zugrunde: Aus einem Nonterminal (q, x, q') lassen sich alle Worte erzeugen, die den Automaten im Zustand q gestartet und dem Stack x in den Zustand q' überführen mit leerem Stack. Als Startsymbol nehmen wir $(q_0, \square, q_a)$. Als Regeln nehmen wir:

- Für $\big((q,x,c),(q',\text{pop})\big) \in \Delta$ mit $x \in \Sigma \cup \{\lambda\}$ gibt es die Regel $(q,c,q') \to x$.
- Für $\big((q,x,c),(q',\text{idle})\big) \in \Delta$ mit $x \in \Sigma \cup \{\lambda\}$ gibt es für jedes $q'' \in Q$ die Regel $(q,c,q'') \to x(q',c,q'')$.
- Für $\big((q,x,c),(q',\text{push}(d))\big) \in \Delta$ mit $x \in \Sigma \cup \{\lambda\}$ gibt es für jedes $q'', q''' \in Q$ die Regel $(q,c,q''') \to x(q',d,q'')(q'',c,q''')$.

Man zeigt nun durch Induktion über die Länge von Berechnungen, dass für alle $w \in \Sigma^*$ und alle $c \in \Gamma$ und alle $q, q' \in Q$ gilt: $(q,w,c) \vdash^* (q',\lambda,\lambda) \iff (q,c,q') \Rightarrow^* w$. Da nun $(q_0, \square, q_a)$ das Startsymbol ist und der Automat nur mit leerem Stack akzeptiert, erhalten wir, dass $(q_0, \square, q_a) \Rightarrow^* w \iff (q_0, w, \square) \vdash^* (q_a, \lambda, \lambda)$. Letzteres ist aber äquivalent zu $w \in L(M)$. Also gilt $L(G) = L(M)$. $\square$

11.2.4 Deterministisch kontextfreie Sprachen

11-23

Parsing in der Praxis.

- Wie schon erwähnt, ist der CYK-Algorithmus in der Praxis zu langsam.
- Kellerautomaten können kontextfreie Grammatiken *viel schneller* parsen.
- Leider braucht man für manche kontextfreie Sprachen tatsächlich *nichtdeterministische* Kellerautomaten.
- Es ist also *nicht* der Fall, dass es zu jedem NPDA einen äquivalenten DPDA gibt.
- Viel schöner ist es natürlich, wenn sich eine kontextfreie Sprache durch einen *deterministischen Kellerautomaten* akzeptieren lässt.

▶ Definition: Deterministisch kontextfreie Sprachen
Eine Sprache heißt *deterministisch kontextfrei*, wenn es einen deterministischen Kellerautomaten gibt, der sie akzeptiert. Die Klasse der deterministisch kontextfreien Sprachen bezeichnen wir mit DCFL.

11-24

Beispiele deterministisch kontextfreier Sprachen.

Beispiel

- Die Sprache der Palindrome ist *nicht* deterministisch kontextfrei.
- Die Sprache $\{w2w^{\text{rev}} \mid w \in \{0,1\}^*\} \subseteq \{0,1,2\}^*$ der *Palindrome mit Trennsymbol* ist deterministisch kontextfrei.

- Programmiersprachen werden von ihrer Syntax her so gebaut, *dass ihr Grundgerüst deterministisch kontextfrei ist*.
- Das Teilgebiet der *Syntaxanalyse* der Theoretischen Informatik beschäftigt sich der Frage, wie man solche Grundgerüste geschickt beschreiben und bauen kann.

Zusammenfassung dieses Kapitels

1. Der CYK-*Algorithmus* entscheidet für eine gegebene kontextfreie Grammatik G und ein Wort w, ob $w \in L(G)$ gilt.
2. Ein *Kellerautomat* ähnelt einem endlichen Automaten, er hat aber noch einen Kellerspeicher (einen Stack) »dabei«.
3. *Nichtdeterministische* PDAs akzeptieren genau die kontextfreien Sprachen.
4. In der Praxis beschränkt man sich auf kontextfreie Sprachen, die von *deterministischen* PDAs akzeptiert werden können.

Zum Weiterlesen

[1] R. Reischuk. *Unterlagen zum Lehrmodul Theoretische Informatik,* Vorlesungsskript, 2008. Kapitel 3.1 und 3.2.

Teil III

Was ist berechenbar?

Rechnen lernt man in der Schule. Insofern erscheint es erstaunlich, dass man sich in höheren Semestern der Universität in eher anspruchsvollen Vorlesungen ausführlich darüber Gedanken macht, was »berechenbar« überhaupt bedeutet. Natürlich könnte dies an der generellen Tendenz der universitären Forschung liegen, auch einfachsten Dingen in unglaublicher Akribie auf den Grund gehen zu wollen und, sobald dieser Grund erreicht ist, noch Tiefbohrungen anzustellen. Beim Thema »Berechenbarkeit« liegt aber tatsächlich zunächst ein echtes Problem vor: Es ist »jedem klar«, was »berechenbar« bedeutet, aber definieren kann man es nur schwer. Es ist in etwa so leicht wie bei »verliebt sein«, wo »jedem klar« ist, was das bedeutet; versuchen Sie aber mal »verliebt sein« sauber zu definieren, ohne sich dabei lächerlich zu machen. Wir werden also einige Energie darauf verwenden müssen, den Begriff »berechenbar« definitorisch in den Griff zu bekommen.

Sobald geklärt ist, was berechenbar überhaupt bedeutet, ist es sicherlich eine der ureigenen Aufgaben der Theoretischen Informatik zu klären, welche Probleme dies *prinzipiell* sind. Die Erfahrung der meisten Informatiker aus der Praxis ist, dass eigentlich jedes Problem *prinzipiell* lösbar ist – man nur gerade jetzt nicht genügend Rechenzeit, Speicherplatz, Programmierer und Pizzas hat, um es bis nächsten Freitag zu lösen. Die folgenden Zitate zeigen, dass Informatikern anscheinend ein grenzenloses Vertrauen in die Wiege gelegt wird, dass sie für jedes Problem auch ein passendes Programm schreiben können:

Aus *Lösungen von Schülern zum 28. Bundeswettbewerb Informatik*

> »Eigentlich hatten wir hier keine Lösungsidee. Wir haben einfach angefangen zu programmieren, und dann hat alles seinen Lauf genommen.«

> »Wir hatten Lösungsideen mit Algorithmen und Ähnlichen, kamen aber zu dem Schluss, dass ein einfaches Programm sinnvoller ist.«

Insofern ist schon ein erstaunliches Resultate der Theorie, dass es überhaupt *nicht berechenbare* Probleme gibt. Das berühmteste ist das Halteproblem, jedoch gibt es noch viele weitere, die einem durchaus – wenn auch selten – in der Praxis über den Weg laufen könnten. Es sei erwähnt, dass es natürlich auch eher pragmatische Arten und Weisen gibt, mit Endlosschleifen umzugehen:

Aus *Lösungen von Schülern zum 28. Bundeswettbewerb Informatik*

> »Die While-Schleife ist eine Endlosschleife. Allerdings wird, wenn die letzte Zahl eingelesen wurde, eine Exception ausgelöst, durch die dann die Schleife abgebrochen wird.«

Entscheidend für diesen Teil der Skriptes ist, dass es wirklich nur um *prinzipielle* Berechenbarkeit geht. Ob ein Problem auch *praktisch* schnell genug gelöst werden kann, werden wir erst in Teil IV untersuchen.

12-1

Kapitel 12
Maschinen I: Die Turingmaschine

»A man provided with paper, pencil, and rubber, and subject to strict discipline, is in effect a universal machine.«

Lernziele dieses Kapitels

1. Syntax und Semantik der Turingmaschine verstehen
2. Begriff der Konfiguration verstehen
3. einfache Turingmaschinen erstellen können
4. Korrektheitsbeweise führen können
5. Klassen der entscheidbaren und der aufzählbaren Sprachen kennen

12-2

Inhalte dieses Kapitels

Worum es heute geht

Der britische Premierminister Gordon Brown verkündete im September 2009:

> So on behalf of the British government, and all those who live freely thanks to Alan's work I am very proud to say: we're sorry, you deserved so much better.

Den Grund für diese Entschuldigung ergibt sich aus der Biographie, von der die englische Wikipedia folgendes zu berichten weiß:

Aus *en.wikipedia.org/wiki/Alan_Turing*, 2009

> Alan Mathison Turing, [...] (23 June 1912 – 7 June 1954), was an English mathematician, logician, cryptanalyst, and computer scientist. He was influential in the development of computer science and providing a formalisation of the concept of the algorithm and computation with the Turing machine. [...] His Turing test was a significant and characteristically provocative contribution to the debate regarding artificial intelligence.
>
> During the Second World War, Turing worked for the Government Code and Cypher School at Bletchley Park, Britain's codebreaking centre. For a time he was head of Hut 8, the section responsible for German naval cryptanalysis. He devised a number of techniques for breaking German ciphers, including the method of the bombe, an electromechanical machine that could find settings for the Enigma machine. After the war he worked at the National Physical Laboratory, where he created one of the first designs for a stored-program computer, the ACE.
>
> Towards the end of his life Turing became interested in chemistry. He wrote a paper on the chemical basis of morphogenesis, and he predicted oscillating chemical reactions such as the Belousov–Zhabotinsky reaction, which were first observed in the 1960s.
>
> Turing's homosexuality resulted in a criminal prosecution in 1952 – homosexual acts were illegal in the United Kingdom at that time – and he accepted treatment with female hormones, chemical

castration, as an alternative to prison. He died in 1954, several weeks before his 42nd birthday, from an apparently self-administered cyanide poisoning, although his mother (and some others) considered his death to be accidental.

12.1 Einleitung

12.1.1 Was bedeutet »berechenbar«?

12-4

Hilberts große Vision.

- Um 1900 erschien dem großen Mathematiker *David Hilbert* die Zeit reif, eine Vision zu verwirklichen.
- Sein Ziel war es, nicht mehr für jeden mathematischen Satz *mühselig einen Beweis zu ergrübeln*.
- Vielmehr wollte er *den Beweis einfach »berechnen«.*
- Gesucht war also eine Art »Verfahren«, bei dem man mit dem Satz begann und dann den Beweis als Resultat erhielt (oder einen Beweis für das Gegenteil).

Das Problem

Es war nicht so klar, welche elementaren Schritt bei solch einem Verfahren erlaubt werden könnten.

- Ein Schritt wie »schreibe den Satz einmal auf ein Blatt, gefolgt von seiner zweiten Hälfte« wäre sicherlich zugelassen.
- Ein Schritt wie »schreibe den Beweis des Satzes auf ein Blatt« hingegen wohl kaum.

12.1.2 Historischer Rückblick

12-5

Historischer Rückblick

- Anfang des zwanzigsten Jahrhunderts entwickelten *Mathematiker* Formalismen, um den Begriff der »Berechenbarkeit zu beschreiben«.
- Wie bei Mathematikern üblich, waren diese Modell *nur für Mathematiker verständlich*.
- Es war völlig unklar, ob eines der vorgeschlagenen Modelle *tatsächlich* den Begriff der Berechenbarkeit adäquat fasste.
- Die Arbeit 1936 von Alan Turing war anders: Hier wurde erstmalig ein *maschinenbasiertes Modell* vorgeschlagen.
- *Hinterher* zeigte sich, dass bestimmte vorher bekannte mathematische Modelle gerade so viel können wie die Turingmaschine.

Mehr dazu in Kapitel 17.

12.2 Die Turing-Maschine

12.2.1 Turings Ideen

12-6

Turing machte alles anders in seiner Definition

State of the Art vor Turing

- Komplexe, undurchsichtige mathematische Kalküle
- Begründet wurden die Kalküle dadurch, dass sie »bei vielen Beispielen funktionierten«
- Rechnen mit Zahlen
- Grenzen der Mächtigkeit der Kalküle unklar

Turings Ideen

- Minimalistisches *mathematisches Modell* einer *Maschine*
- Eine Argumentation, dass *alle überhaupt denkbaren Berechnungen* durch das Modell abgedeckt werden
- Rechnen mit Zeichenketten
- Grenzen der Berechenbarkeit (recht) leicht beweisbar

Turings Ziel

12-7

Turing wollte formal fassen, »wie ein Mathematiker eine sehr lange Berechnung durchführen würde«:

Ausgangspunkt

Eine »Berechnung« auch bedeutet, einen formalen Beweis (siehe die Vorlesung »Logik für Informatiker«) zu überprüfen. Der Mathematiker hat dazu

- einen *beliebig großen Stapel an Karopapier*,
- einen *unerschöpflichen Bleistift*,
- einen *unerschöpflichen Radiergummi* und
- eine *eiserne Disziplin*.

Ein Turings Worten:

> »A man provided with paper, pencil, and rubber, and subject to strict discipline, is in effect a universal machine.«

Turings erste Idee: Endlich vielen Symbole pro Zelle

12-8

Turings Ideen

- Der Mathematiker mag gute Augen haben,…
- …jedoch passen in ein Kästchen nur endlich viele unterschiedliche Symbole.
- Es nützt dem Mathematiker auch nichts, Symbole »über mehrere Kästchen zu strecken«,…
- …denn das entspricht einfach mehreren Symbolen in mehreren Kästchen.

Formalisierung

- Es gibt ein *Bandalphabet*. Wie jedes Alphabet ist dieses *endlich*.
- Die Symbole stehen in Kästchen, formal (Speicher-)*Zellen* genannt.

Turings zweite Idee: Zellen auf einem Band

12-9

Turings Idee

- Die Zellen mögen auf einem zweidimensionalen »Blatt« verteilt sein,…
- …bei einer sehr langen Rechnung muss man aber so oder so »blättern«.
- Da kann man die zweidimensionalen Blätter gleich weglassen und nur ein *Band* benutzen.

Formalisierung

- Die Zellen werden in *eindimensionalen Bändern* angeordnet.
- Formal ist ein Band eine Funktion $t\colon \mathbb{Z} \to \Gamma$.
- Prinzipiell sind Bänder »unendlich lang«, sie sind aber bis auf einen endlichen Anteil leer.
- Dieser endliche Anteil ist damit einfach ein *Wort über dem Bandalphabet*.

Turings dritte Idee: Diskrete Zeitschritte

12-10

Turings Idee

Der Mathematiker kann nicht »unendlich schnell« arbeiten.

Formalisierung

Die Berechnung verläuft in *diskreten Schritten*.

12-11

Turings vierte Idee: Der Schreib-Lese-Kopf

Turings Idee

- Während der Mathematiker arbeitet, hat er immer nur ein Blatt »aktuell vor sich«.
- Will er ein anderes Blatt anschauen, so muss er sich zunächst dorthin »durchblättern«.

Formalisierung

- Für ein Band gibt es einen »Schreib-Lese-Kopf«.
- Dies ist einfach ein Index einer Bandposition.
- In jedem Berechnungsschritt kann sich der Kopf bewegen, aber nur eine Zelle vor oder zurück.

12-12

Turings fünfte Idee: Die endlich vielen Geisteszustände

Turings Idee

- Während der Mathematiker arbeitet, ist sein Gehirn immer in einem »Geisteszustand«.
  Beispiel: Ich muss jetzt den aktuellen Beweis überprüfen.
  Beispiel: Ich suche nach der schließenden Klammer.
- Turing argumentiert, es gäbe zwar viele, aber eben nur endlich viele Geisteszustände.

Formalisierung

Es gibt eine endliche Menge Q von Zuständen.

12.2.2 Syntax

12-13

Zur Vielfalt der Syntaxvarianten der Turingmaschine.

- Das Konzept der Turingmaschine ist sehr *wandelbar* (man kann viele Varianten betrachten) und *robust* (die Varianten können alle dasselbe).
- In der Theoretischen Informatik werden *hunderte unterschiedliche Definitionen und Arten* von Turingmaschinen benutzt.
- Die Grundideen sind immer dieselben.
- Die nachfolgende Definition ist *ziemlich allgemein*, aber *längst nicht die allgemeinste mögliche*.

12-14

Eine Schritt-für-Schritt Definition der Syntax.

▶ **Definition:** Maschinen-Alphabete
Ein *Bandalphabet* Γ ist ein Alphabet, das das Blanksymbol $\square$ enthält und wenigstens ein weiteres Symbol. Ein *Eingabealphabet* Σ ist eine Teilmenge von Γ, die $\square$ *nicht* enthält.

▶ **Definition:** Bänder, Schränke
Ein *Band* ist eine Sequenz von Symbolen, die beidseitig unendlich ist. Ein *Schrank* enthält eine endliche Anzahl von Bändern. Formal ist ein Band eine Funktion $t\colon \mathbb{Z} \to \Gamma$ und ein Schrank ist ein Tupel $(t_1, \ldots, t_k)$ von Bändern.

▶ **Definition:** Kopf
Ein *Kopf* ist eine Position auf einem Band, also eine ganze Zahl.

▶ **Definition:** Zustände
Ein *Zustand* ist im Prinzip ein Programmzähler. Formal ist ein Zustand ein Element einer *endlichen Menge Q von Zuständen*. Der *Anfangszustand* q_0 ist ein Element von Q. Die Menge der *akzeptierenden Zuständen* Q_a ist eine Teilmenge von Q.

▶ **Definition:** Kopf-Bewegungen
Während einer Berechnung kann sich der Kopf bewegen. Es gibt *drei* erlaubte Bewegungen: links, rechts und stehenbleiben. Diese werden durch drei Symbole angedeutet: $\{\text{links}, \text{rechts}, \text{neutral}\}$ oder $\{l, r, n\}$ oder $\{-1, 1, 0\}$.

▶ Definition: Programm
Ein *Programm für h Köpfe* ist eine *partielle* Funktion, die Paare von

- einem Zustand und
- einem h-Tupel von gelesenen Bandsymbolen

auf Tripel abbildet, bestehend aus

- einem neuen Zustand,
- einem h-Tupel von geschriebenen Bandsymbolen und
- einem h-Tupel von Kopf-Bewegungen.

Formal ist dies eine partielle Funktion

$$\delta\colon Q \times \Gamma^h \dashrightarrow Q \times \Gamma^h \times \{l,r,n\}^h.$$

▶ Notation
Wir stellen Programme als Graphen dar mit den Zuständen als Knoten. Es gibt eine Kanten von q nach q' mit der Markierung $a/b/c$, falls $\delta(q,a) = (q',b,c)$.

▶ Definition: Deterministische Mehrband-Turing-Maschine
Eine *k-Band-Turing-Maschine* (k-DTM) besteht aus:

- Einer endlichen Menge Q von Zuständen.
- Einem Anfangszustand $q_0 \in Q$.
- Einer Menge $Q_a \subseteq Q$ von akzeptierenden Zuständen.
- Einem Bandalphabet Γ mit $\square \in \Gamma$.
- Einem Eingabealphabet $\Sigma \subseteq \Gamma$ mit $\square \notin \Gamma$.
- Einer Anzahl k an Bändern.
- Einem Programm δ für k Köpfe.

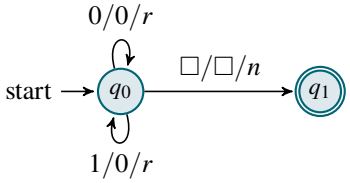
▶ Notation
Wir stellen Turing-Maschinen durch ihre Programme dar. Der Initialzustand ist durch einen Pfeil angedeutet, akzeptierende Zustände durch Doppelkreise.

Beispiel einer einfachen Turing-Maschine.

12-15

Beispiel: Eine ganz einfache Maschine

- Die Zustandsmenge sei $Q = \{q_0, q_1\}$.
- Der Initial-Zustand sei q_0.
- Das Bandalphabet sei $\Gamma = \{0,1,\square\}$.
- Das Eingabeaphabet sei $\Sigma = \{0,1\}$.
- Die Anzahl der Bänder sei 1.
- Das Programm sei



12.2.3 Semantik

12-16

Eine Schritt-für-Schritt Definition der Semantik.

▶ Definition: Konfigurationen
Eine *Konfiguration* einer Turingmaschine M ist ein Tupel, bestehend aus
- dem *aktuellen Zustand* $q \in Q$,
- dem *aktuellen Schrank an Bändern* und
- den *aktuellen Kopfpositionen*.

Formal ist dies also ein Element von $Q \times [\mathbb{Z} \to \Gamma]^k \times \mathbb{Z}^k$. Hierbei ist $[\mathbb{Z} \to \Gamma]$ die Menge aller Funktionen $t : \mathbb{Z} \to \Gamma$.

▶ Notation
Ein Band mit einem Kopf schreiben wir wie folgt auf:
1. Wir beginnen mit Pünktchen.
2. Dann schreiben wir alle Symbole vor der Kopfposition auf, beginnend mit der ersten Stelle, die nicht □ ist, oder einer früheren Stelle.
3. Dann schreiben wir das spezielle Dreiecks-Symbol hin, *das kein Element des Bandalphabets ist, sondern nur eine Markierung*.
4. Dann kommen alle folgenden Symbole, mindestens bis zum letzten, das nicht □ ist.
5. Wir enden mit Pünktchen.

Beispiel: Ein Band mit dem Kopf auf der ersten 1
$\cdots$□□0022002▷12000020□1101□□$\cdots$

▶ Definition: Berechnungsschritt
Eine Konfiguration $C = (q, t_1, \ldots, t_k, h_1, \ldots, h_k)$ hat die *Nachfolgekonfiguration* $C' = (q', t_1', \ldots, t_k', h_1', \ldots, h_k')$, geschrieben $C \vdash_M C'$, falls:
- Sei σ_i das Symbol an Position h_i des Bandes t_i, also $\sigma_i = t_i(h_i)$.
- Sei $\delta(q, \sigma_1, \ldots, \sigma_k) = (q', \sigma_1', \ldots, \sigma_k', m_1, \ldots, m_k)$.
- Dann ist

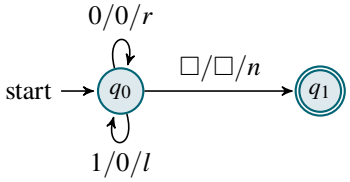
$$t_i'(j) = \begin{cases} t_i(j), & \text{falls } j \neq h_i, \\ \sigma_i', & \text{sonst.} \end{cases}$$

$$h_i' = h_i + \begin{cases} -1, & \text{falls } m_i = l, \\ 1, & \text{falls } m_i = r, \\ 0, & \text{falls } m_i = n. \end{cases}$$

▶ Definition: Anfangskonfiguration
Die *Anfangskonfiguration* $C_{\text{init}}(w)$ einer Maschine M für ein Wort $w \in \Sigma^*$ lautet:
- Der Zustand ist q_0.
- $t_1(i) = w[i]$ für $i \in \{1, \ldots, |w|\}$.
- Alle anderen Zellen sind mit □ gefüllt.
- Alle Köpfe sind auf Position 1.

▶ Definition: Endkonfigurationen und akzeptierende Konfigurationen
Eine *Endkonfiguration* ist eine Konfiguration ohne Nachfolgekonfiguration (dies ist der Fall, wenn die partielle Funktion δ für die gelesenen Symbole undefiniert ist). Eine Endkonfiguration heißt *akzeptierend*, wenn ihr Zustand ein Element von Q_a ist.

✎ Zur Übung
Bei welchen Worten erreicht die Maschine eine Endkonfiguration? Bei welchen eine akzeptierende? Das Eingabealphabet ist $\Sigma = \{0, 1\}$.

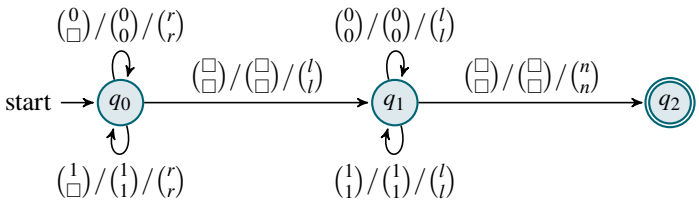


▶ Definition: Akzeptierte Sprache

Sei M eine Turingmaschine. Wir sagen, *M akzeptiert ein Wort $w \in \Sigma^*$*, falls eine akzeptierende Endkonfiguration C existiert mit $C_{\text{init}}(w) \vdash_M^* C$. Die von *$M$ akzeptierte Sprache $L(M)$* ist die Menge aller von M akzeptierten Worte.

✎ Zur Übung 12-17

Welche Sprache akzeptiert die folgende Maschine? Das Eingabealphabet ist $\Sigma = \{0,1\}$.



12.2.4 Korrektheitsbeweise

Beweisrezept: *Korrektheitsbeweis für DTMs* 12-18

Ziel

Man will beweisen, dass eine DTM *M eine Sprache L akzeptiert.*

Rezept

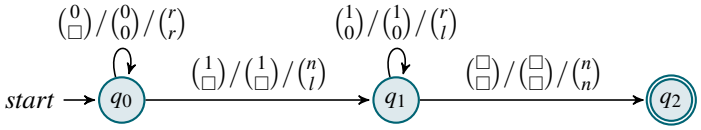
Zeige zwei Richtungen:

1. Beginne mit »Sei $w \in L$ beliebig.« Gib die Folge von Konfigurationen an, die M bei Eingabe w durchläuft. Ende mit »Also gilt $C_{\text{init}}(w) \vdash^* C$, wobei C eine akzeptierende Endkonfiguration ist. Somit gilt $L \subseteq L(M)$«.
2. Beginne mit »Sei $w \in L(M)$ beliebig.« Fahre fort mit »Dann gilt $C_{\text{init}}(w) = C_1 \vdash \cdots \vdash C_n$, wobei C_n eine akzeptierende Endkonfiguration ist.« Argumentiere dann, dass $w \in L$ gilt. Ende mit »Also gilt $w \in L$ und somit $L(M) \subseteq L$«.

Kurzes Beispiel eines Korrektheitsbeweises. 12-19

▶ Satz

Sei M folgende Turingmaschine:



Dann gilt $L(M) = \{0^n 1^n \mid n \geq 1\}$.

Beweis. Wir zeigen zwei Richtungen. Sei zunächst $w = 0^n 1^n$ mit $n \geq 1$. Dann akzeptiert M dieses Wort durch folgende Berechnung:[1]

$$
\begin{aligned}
&(q_0, \cdots \triangleright 0^n 1^n \cdots, \cdots \triangleright \square \cdots) \\
\vdash\ &(q_0, \cdots 0 \triangleright 0^{n-1} 1^n \cdots, \cdots 0 \triangleright \square \cdots) \\
\vdash\ &(q_0, \cdots 00 \triangleright 0^{n-2} 1^n \cdots, \cdots 00 \triangleright \square \cdots) \\
\vdash^*\ &(q_0, \cdots 0^n \triangleright 1^n \cdots, \cdots 0^n \triangleright \square \cdots) \\
\vdash\ &(q_1, \cdots 0^n \triangleright 1^n \cdots, \cdots 0^{n-1} \triangleright 0 \square \cdots) \\
\vdash\ &(q_1, \cdots 0^n 1 \triangleright 1^{n-1} \cdots, \cdots 0^{n-2} \triangleright 00 \square \cdots) \\
\vdash^*\ &(q_1, \cdots 0^n 1^n \triangleright \square \cdots, \cdots \triangleright \square 0^n \square \cdots) \\
\vdash\ &(q_2, \cdots 0^n 1^n \triangleright \square \cdots, \cdots \triangleright \square 0^n \square \cdots).
\end{aligned}
$$

Für die andere Richtung sei $w \in L(M)$. Damit M ein Wort akzeptiert, muss die Berechnung in q_2 enden. Da die Berechnung in q_0 begann, muss sie zunächst eine Weile dort geblieben sein, dann nach q_1 gewechselt sein, um dann in q_2 zu enden.[2] Während die Berechnung in q_0 verweilt, werden alle 0en auf das zweite Band kopiert.[3] Dann muss eine 1 kommen in w. Damit q_2 erreicht wird, dürfen

Kommentare zum Beweis

[1] Wir geben die Konfigurationsfolge einfach direkt an.

[2] Analyse der Berechnung

[3] Dies müsste man eigentlich durch Induktion zeigen. Macht aber kein Mensch.

1. nur noch 1en kommen und
2. für jede 1 muss eine 0 auf dem zweiten Band stehen.

[4] Folglich gilt $w = 0^n 1^n$ mit $n \geq 1$ und folglich $w \in L$. □

[4] Eigentlich noch mehr Induktionen.

12.3 Sprachklassen

12.3.1 Aufzählbare Sprachen

12-20

Merkwürdige Namen.

▶ **Definition:** Akzeptierbare Sprachen
Die Klasse aller von deterministischen Turingmaschinen akzeptierten Sprachen hat verschiedene Namen:

- Klasse der *akzeptierbaren Sprachen*
- Klasse der *semientscheidbaren Sprachen*
- Klasse der *rekursive aufzählbaren Sprachen*
- RE (von englisch *recursively enumerable*).

Diese Namen, insbesondere die Bezeichnung RE, ist historisch gewachsen und nicht sonderlich einleuchtend.

12.3.2 Entscheidbare Sprachen

12-21

Ein Problem bei Turing-Maschinen.

- Eine Turing-Maschine kann bei einer Eingabe *in eine Endlosschleife* geraten.
- Das war auch schon bei 2-Wege-Automaten der Fall, wo es aber nicht sonderlich problematisch war.
- Bei Turing-Maschinen *macht es aber einen Unterschied*, ob man solche Endlosschleifen zulässt oder nicht:

▶ **Definition:** Totale Maschinen
Eine Turing-Maschine heißt *total*, wenn sie bei jeder Eingabe früher oder später anhält. (Wenn es also keine unendlich langen Berechnungsfolgen gibt).

▶ **Definition:** Entscheidbare Sprachen
Die Klasse REC enthält alle Sprachen, die von *totalen* Turing-Maschinen akzeptiert werden. Man nennt solche Sprachen *entscheidbar* oder *rekursiv*.

12-22

Beweisrezept: *Entscheidbarkeit beweisen*

Ziel
Man will beweisen, dass eine Sprache L entscheidbar ist.

Rezept

1. Konstruiere eine Turing-Maschine M.
2. Beweise dann, dass $L(M) = L$ gilt (Beweisrezept »Korrektheit von DTMS«)
3. Beweise dann, dass M bei jeder Eingabe anhält. Gib dazu zu jeder Eingabe an, wie die Berechnung aussieht und in welcher Endkonfiguration sie endet (die Endkonfiguration braucht *nicht* akzeptierend sein).

Zusammenfassung dieses Kapitels

1. Turing-Maschinen wurden eingeführt, um das Vorgehen eines Mathematikers während einer Berechnung zu modellieren.
2. Sie verfügen über eine feste Anzahl Bänder, auf denen je ein Kopf sich bewegt, der dort lesen und schreiben kann.
3. Die Klasse RE enthält alle von Turing-Maschinen akzeptierbaren Sprachen.
4. Die Klasse REC enthält alle von *totalen* Turing-Maschinen akzeptierbaren Sprachen.

Zum Weiterlesen

[1] Alan M. Turing, On Computable Numbers With an Application to the Entscheidungsproblem, *Proceedings of the London Mathematical Society,* 42(2):230–265, 1936

Dies ist die Originalarbeit, in der Turing seine Turing-Maschine einführt (die er natürlich nicht so nennt; es ist sehr unfein, den eigenen Namen in einer solchen Weise zu benutzen – dies überlässt man den Leuten, die einen zitieren). In diesem Artikel

1. fegt Turing durch seine maschinenbasierte Definition der »Berechenbarkeit« ältere, alternative Definitionsversuche hinweg und
2. pulversiert durch seinen Beweis, dass das Halteproblem unentscheidbar ist, das gesamte große Hilbert'sche Programm – immerhin das zentrale Projekt der Mathematikerzunft seit Anfang des Jahrhunderts.

Unterm Strich war dieses, sein erstes Paper kein schlechter akademischer Einstieg.
Dieser Artikel ist online verfügbar, beispielsweise beim History of Computing Project. Sollten auch Turings zweiten berühmten Artikel, *Computing Machinery and Intelligence,* nicht verpassen, wo er den Turing-Test vorschlägt.

[2] R. Reischuk. *Unterlagen zum Lehrmodul Theoretische Informatik,* Vorlesungsskript, 2008. Kapitel 4.1, 4.2, 4.3.

Übungen zu diesem Kapitel

Übung 12.1 Turing-Maschine für die Sprache der Palindrome, mittel

Es sei $L = \{ww^{\text{rev}} \mid w \in \{0,1\}^*\}$ gegeben.

1. Konstruieren Sie eine 2-Band DTM, die L akzeptiert.
2. Konstruieren Sie eine 1-Band DTM, die L akzeptiert.
3. Wie viele Rechenschritte machen Ihre Turingmaschinen bei Worten 0^n mit $n \in \mathbb{N}$?
4. Geben Sie einen formalen Beweis für die Korrektheit Ihrer 2-Band DTM an.

Übung 12.2 1-Band DTM implementieren, mittel

Sie sollen in dieser Aufgabe eine 1-Band DTM implementieren, indem Sie die Methode isAccepted vervollständigen. Als Abgabe wird ein ausführbares Programm mit Testeingaben und Testläufen auf Testeingaben erwartet.

```
class Transition{

  int symbol;
  // symbol ist das ASCII Zeichen welches im Rechenschritt geschrieben wird

  int state;
  // state ist der eingenommene Zustand

  int direction;
   // -1 =  Kopfbewegung nach links, 0 = neutral, 1 = Kopfbewegung nach
       rechts
}

class TuringMachine {

  // Sigma ist einfach gleich ASCII

  int sizeOfQ;
  // Q ist die Menge {0,...,sizeOfQ-1}
```

```
  Transition [][] delta;
  // delta[q][x] ist gerade delta(q,x)

  int q_0;
  // Startzustand

  boolean [] Q_a;
  // Q_a[i] ist genau dann wahr, wenn i \in Q_a

  boolean isAccepted (String s) {
    // Ihr Code!
  }
}
```

Übung 12.3 Turing-Maschine konstruieren, mittel

Es sei $L = \{a^n b^n c^n \mid n \in \mathbb{N}\}$ gegeben.

1. Konstruieren Sie eine 2-Band DTM für L.
2. Konstruieren Sie eine 1-Band DTM, die L akzeptiert.
3. Geben Sie einen formalen Beweis für die 2-Band Turingmaschine an..

Übung 12.4 Binäraddition, mittel

Es sei die Sprache $L = \{\text{bin}(i)\#\text{bin}(j)\#\text{bin}(i+j) \mid i, j \geq 0\}$ gegeben. Geben Sie eine DTM an, welche diese Sprache entscheidet.

Übung 12.5 Binärsubtraktion, mittel

Es sei die Sprache $L = \{\text{bin}(i)\#\text{bin}(j)\#\text{bin}(i-j) \mid i \geq j \geq 0\}$ gegeben. Geben Sie eine DTM an, welche diese Sprache entscheidet.

13-1

Kapitel 13
Maschinen II: Nichtdeterminismus

Eine akzeptierende Berechnungen genügt immernoch zur Akzeptanz

13-2

Lernziele dieses Kapitels

1. Nichtdeterminismus bei Turingmaschinen verstehen
2. NTMs entwerfen können
3. Konzept des »nichtdeterministischen Ratens« verstehen
4. Äquivalenzbeweis zu deterministischen Maschinen kennen

Inhalte dieses Kapitels

Worum es heute geht

Wozu wurden in Kapitel 6 eigentlich nichtdeterministische endliche Automaten eingeführt? Wenn man sich das Kapitel nochmal durchliest, so wird dort wortreich erklärt, dass Nichtdeterminismus ganz toll und wichtig sei – ein wirklich schlagendes Beispiel sucht man vergebens. Schlimmer noch: Es wird gleich gezeigt, dass man den Nichtdeterminismus gar nicht braucht, da er auch nicht mehr kann als die deterministischen Automaten! Er sieht verdächtig nach »Viel Lärm um nichts« aus; einem Thema, dem sich Shakespeare zwar ausführlich widmete, das deshalb aber für ein nichtgeisteswissenschaftliches Studium zunächst von geringer Relevanz erscheint.

Man muss schon weiterlesen, um die Wichtigkeit von nichtdeterministischen endlichen Automaten zu entdecken: Erst im Beweis von Satz 7-22, und dort auch nur im Kleingedruckten, werden diese Automaten gebraucht, um Ableitungen mittels regulärer Grammatiken durch endliche Automaten nachzuvollziehen. Mit anderen Worten: Das ganzes Konzept des nichtdeterministischen Automaten wurde eigentlich nur eingeführt, um ein Detailproblem in einem Beweis eines bestimmten Satzes zu lösen. Sollten Sie zu den Lesern gehören, denen Beweise sowieso unsympathisch sind, so werden Sie sicherlich spätestens jetzt dem Konzept des Nichtdeterminismus die Daseinsberechtigung absprechen.

In diesem Kapitel geht es nun um nichtdeterministische Turing-Maschinen. Dabei liegt die Sachlage ähnlich wie bei endlichen Automaten: So richtig brauchen tut man diese Maschinen praktisch nicht. Wieder sind sie aber aus Sicht der Theorie sehr wichtig – wodurch sie in einer Veranstaltung mit dem Titel »Theoretische Informatik« wohl doch eine Daseinsberechtigung haben.

13.1 Nichtdeterministische Turing-Maschinen

13.1.1 Idee

13-4 Maschinen, die sich nicht entscheiden können.

- Jede Art von endlichen Automaten (normale, 2-Wege-Automaten, Kellerautomaten) gibt es als *deterministische* und als *nichtdeterministische* Variante.
- Zur Erinnerung:
  - Bei einem *nichtdeterministischen Automaten* kann es zu einem Zustand und einem gelesenen Zeichen *mehrere mögliche Nachfolgezustände* geben.
  - Diese sind dann alle *möglich*, ...
  - ... wodurch es *viele mögliche Berechnungen* gibt.
  - Akzeptiert wird ein Wort, wenn es wenigstens eine akzeptierende Berechnung *gibt*.
- Dieselben Ideen können wir auf Turing-Maschinen übertragen:
  - Bei einer *nichtdeterministischen Turing-Maschine* kann es zu einem Zustand und gelesenen Zeichen *mehrere mögliche Nachfolgezustände* geben.
  - Diese sind dann alle *möglich*, ...
  - ... wodurch es *viele mögliche Berechnungen* gibt.
  - Akzeptiert wird ein Wort, wenn es wenigstens eine akzeptierende Berechnung *gibt*.

13-5 Nichtdeterministische Turing-Maschinen sind nicht realistisch.

- Genau wie man NFAs nicht »bauen« kann, kann man auch NTMs nicht »bauen«.
- Genau wie NFAs nur in der Theorie gebraucht werden, werden auch NTMs nur in der Theorie gebraucht.
- Genau wie NFAs in der Theorie sehr wichtig sind, sind auch NTMs in der Theorie sehr wichtig.

13.1.2 Syntax

13-6 Syntax einer nichtdeterministischen Turing-Maschine.

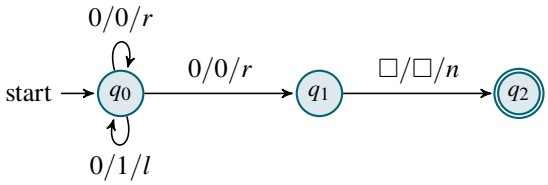
▶ Definition: Nichtdeterministisches Programm
Ein *nichtdeterministisches Programm für h Köpfe* ist eine *Relation* Δ mit

$$\Delta \subseteq \big(Q \times \Gamma^h\big) \times \big(Q \times \Gamma^h \times \{l,r,n\}^h\big).$$

- Ein nichtdeterministisches Programm Δ ist formal fast genauso definiert wie ein (normales, deterministisches) Programm, nur dass statt einer »partiellen Funktion« eine »Relation« genutzt wird.
- Der einzige Unterschied ist somit, dass es zu einem Paar Zustand/Zeichen *mehrere* (oder auch keine) mögliche Aktionen geben kann.

▶ Definition: Nichtdeterministische Turing-Maschine
Eine *nichtdeterministische Turing-Maschine* (NTM) ist genauso definiert wie eine DTM, nur hat sie ein nichtdeterministisches Programm statt einem (normalen, deterministischen) Programm.

13-7 Beispiel einer NTM.



- Die Maschine ist im Zustand q_0 nichtdeterministisch.
- Wenn sie in diesem Zustand ist und eine 0 liest, gibt es *drei Möglichkeiten*, wie die Berechnung weitergehen kann:

  - Die Maschine kann einen Schritt nach rechts gehen und in q_0 bleiben.
  - Die Maschine kann einen Schritt nach rechts gehen und nach q_1 wechseln.
  - Die Maschine kann die 0 durch eine 1 ersetzen und einen Schritt nach links gehen und in q_0 bleiben.
- Alle diese Aktionen sind *möglich*.

13.1.3 Semantik

Semantik einer NTM. 13-8

- Bei NTMs sind die folgenden Begriffe genauso definiert wie bei DTMs:
  - *Konfigurationen* (ein Element von $Q \times [\mathbb{Z} \to \Gamma]^k \times \mathbb{Z}^k$),
  - *Anfangskonfiguration* (Eingabewort füllt die Zellen des ersten Bandes, alle anderen Zellen auf allen Bändern sind mit $\square$ gefüllt),
  - *Endkonfigurationen* (hat keine Nachfolgekonfiguration) und
  - *akzeptierende Konfigurationen* (Endkonfiguration mit einem Zustand aus Q_a).
- Neu ist, dass es zu einer Konfiguration nun *mehrere Nachfolgekonfigurationen* geben kann.

▶ **Definition:** Nachfolgekonfigurationen einer NTM
Sei M eine NTM. Seien $C = (q, t_1, \ldots, t_k, h_1, \ldots, h_k)$ und $C' = (q', t'_1, \ldots, t'_k, h'_1, \ldots, h'_k)$ Konfigurationen. Dann heißt C' *eine Nachfolgekonfiguration von* C, geschrieben $C \vdash_M C'$, falls

- es ein Paar $\big((q, \sigma_1, \ldots, \sigma_k), (q', \sigma'_1, \ldots, \sigma'_k, m_1, \ldots, m_k)\big) \in \Delta$ gibt, so dass
- σ_i das Symbol an Position h_i des Bandes t_i ist, also $\sigma_i = t_i(h_i)$, und

$$t'_i(j) = \begin{cases} t_i(j), & \text{falls } j \neq h_i, \\ \sigma'_i, & \text{sonst.} \end{cases}$$

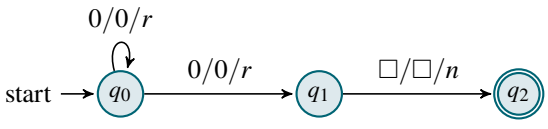
$$h'_i = h_i + \begin{cases} -1, & \text{falls } m_i = l, \\ 1, & \text{falls } m_i = r, \\ 0, & \text{falls } m_i = n. \end{cases}$$

▶ **Definition:** Berechnung, akzeptierende Berechnung
Eine *Berechnung bei Eingabe* w einer NTM M ist eine Folge $C_{\text{init}}(w) = C_1 \vdash_M \cdots \vdash_M C_n$. Eine Berechnung heißt *akzeptierend*, wenn C_n eine akzeptierende Konfiguration ist (also eine Endkonfiguration mit einen Zustand aus Q_a).

▶ **Definition:** Akzeptiertes Wort, akzeptierte Sprache
Sei M eine NTM und $w \in \Sigma^*$ ein Wort. Dann *akzeptiert* M *das Wort* w, wenn es (wenigstens) eine akzeptierende Berechnung bei Eingabe w *gibt*. Die Menge aller von M akzeptierten Worte bezeichnen wir mit $L(M)$.

13.1.4 Beispiele

Eine einfache Maschine. 13-9



Einige Beobachtungen:

- Es gibt viele mögliche Berechnungen.
- Es führt aber immer nur höchstens eine zum akzeptierenden Zustand: Wenn man bei der letzten 0 vor dem Ende nach q_1 wechselt.
- Außerdem dürfen in dem Wort nur 0en vorkommen.
- Die Maschine akzeptiert also die Sprache $L(M) = \{0^n \mid n \geq 1\}$.

Zwei mögliche Berechnungen bei Eingabe 00010:

$$\begin{aligned}(q_0, \cdots \square \triangleright 00010 \square \cdots) &\vdash (q_0, \cdots \square 0 \triangleright 0010 \square \cdots) \\ &\vdash (q_0, \cdots \square 00 \triangleright 010 \square \cdots) \\ &\vdash (q_1, \cdots \square 000 \triangleright 10 \square \cdots).\end{aligned}$$

$$\begin{aligned}(q_0, \cdots \square \triangleright 00010 \square \cdots) &\vdash (q_0, \cdots \square 0 \triangleright 0010 \square \cdots) \\ &\vdash (q_1, \cdots \square 00 \triangleright 010 \square \cdots).\end{aligned}$$

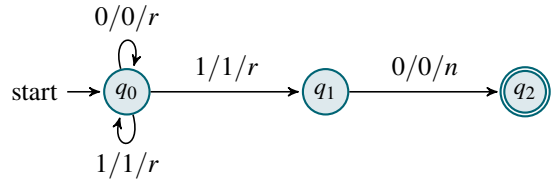
Keine davon ist akzeptierend.

13-10

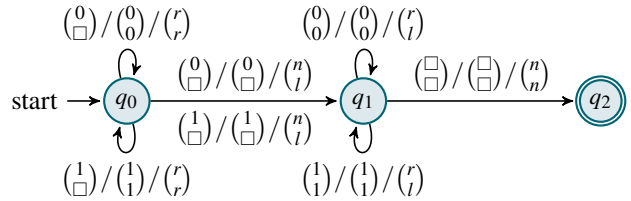
✎ Zur Übung

Geben Sie *eine akzeptierende Berechnung* zur Eingabe 1001 für *eine* der folgenden NTMs (nach Schwierigkeit geordnet):

1.



2.



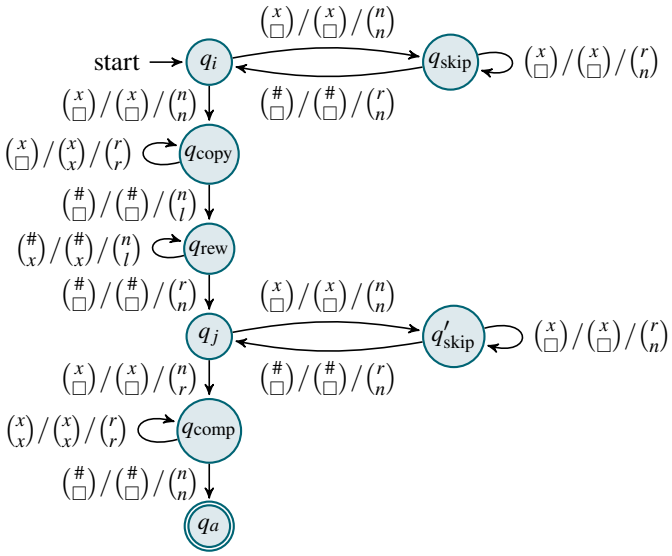
Wie lautet jeweils $L(M)$?

13-11

Eine komplexere Maschine.

$$\begin{aligned}\textsc{copy} = \{X_1\#\cdots\#X_n\# \mid n \geq 2, X_i \in \{0,1\}^{\geq 1}, \\ \exists i \exists j\ (i \neq j \wedge X_i = X_j)\}.\end{aligned}$$

- Diese Sprache ist deterministisch *schwieriger* zu akzeptieren, nichtdeterministisch aber *recht einfach*.
- Wir laufen von links nach rechts über das Wort. Am Anfang jedes X_i gibt es eine nichtdeterministische Entscheidung:
  - Eine Berechnung überliest einfach X_i.
  - Eine andere Berechnung kopiert X_i auf ein weiteres Band.
- Dann laufen wir noch weiter und entscheiden uns wieder am Anfang von jedem weiteren Wort nichtdeterministisch:
  - Eine Berechnung überliest X_i.
  - Eine andere Berechnung vergleicht X_i mit dem Wort auf dem anderen Band.

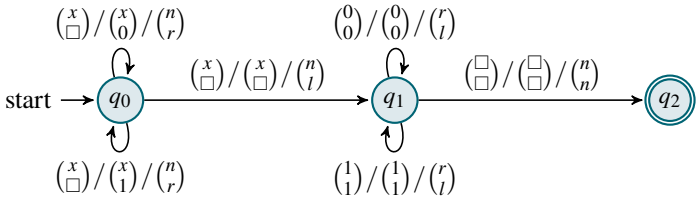


Hierbei ist $x \in \{0,1\}$ beliebig. Ein Pfeil wie $\binom{x}{\square}/\binom{x}{\square}/\binom{r}{r}$ steht somit für zwei Pfeile $\binom{0}{\square}/\binom{0}{\square}/\binom{r}{r}$ und $\binom{1}{\square}/\binom{1}{\square}/\binom{r}{r}$.

13.1.5 Nichtdeterministisches Raten

Eine Maschine, die »nichtdeterministisch ein Wort rät«. 13-12

✎ Zur Diskussion



- In welchen Zuständen verhält sich die Maschine nichtdeterministisch?
- Welche Sprache akzeptiert die Maschine?

Was »nichtdeterministisches Raten« in Wirklichkeit bedeutet. 13-13

- Solange die Maschine in q_0 bleibt, kann sie *immer neue Zeichen auf das zweite Band schreiben*.
- Irgendwann ist sie aber damit fertig und wechselt nach q_1.
- Es gibt also *für jedes mögliche Wort* $w \in \{0,1\}^*$ eine Berechnung, so dass die Maschine in q_1 ankommt und dieses Wort auf dem zweiten Band steht.
- Keine dieser Berechnungen ist »besser« oder »besonders«; es werden auch nicht »beide gleichzeitig« durchlaufen – es *gibt einfach viele mögliche Berechnungen*.
- *Trotzdem sagt man* (eigentlich *völlig fälschlicher Weise*), dass die Maschine *»nichtdeterministisch w rät«*.

Merke

»Nichtdeterministisches Raten eines Wortes« bedeutet in Wirklichkeit, dass es für jedes Wort eine Berechnung gibt, bei der dieses Wort auf einem Arbeitsband steht, wenn ein bestimmter Zustand erreicht wird.

13.2 Umwandlung in deterministische Maschinen

13.2.1 Die Idee

Nichtdeterminismus braucht man nicht. 13-14

- Bei NFAs hatten wir gezeigt, dass man sie immer in DFAs umwandeln kann (mittels Power-Zuständen).
- Man kann auch jede NTM in eine DTM umwandeln.
- Die Idee hinter dem Beweis ist aber eine andere, Power-Zustände nützen einem nichts.

Idee zur Umwandlung von NTMs in DTMs

- Eine NTM kann viele *mögliche* Berechnungen haben.
- Diese Berechnungen passieren nicht gleichzeitig oder nacheinander – sie sind einfach alle *möglich*.
- Eine DTM muss nun aber *systematisch* überprüfen, ob eine von ihnen *akzeptierend* ist.
- Dies geschieht mittels einer *Breitensuche über den Baum der nichtdeterministischen Entscheidungen*.

13.2.2 Der Satz

13-15

Umwandlung in deterministische Maschinen.

▶ Satz
Zu jeder NTM M *gibt es eine* DTM M' *mit* $L(M) = L(M')$.

Ideen

- Die DTM erweitert die NTM um zwei Extrabänder:
  - Auf einem merkt sie sich die Original-Eingabe.
  - Auf dem anderen merkt sie sich eine *Folge von nichtdeterministischen Entscheidungen*.
- In einer großen (Endlos-)Schleife durchläuft sie *alle möglichen Folgen von nichtdeterministischen Entscheidungen*.
- Für jede mögliche Folge löscht sie die ersten k Bänder, kopiert das Original-Eingabewort auf das erste Band und »probiert aus, ob mit dieser Folge die NTM das Wort akzeptiert«.
- Wird eine Folge gefunden, bei der die NTM akzeptiert, so stoppt und akzeptiert auch die DTM; sonst läuft sie ewig (und akzeptiert somit nicht).

Kommentare zum Beweis

Beweis. [1] Die gesuchte DTM M' geht aus M durch Umformungen hervor. Zunächst hat M' zwei Bänder mehr als M. Das erste dieser Extrabänder nennen wir das *Eingabe-Rettungsband*. Das zweite nennen wir das *Band der Entscheidungen*.[2] Auf das *Eingabe-Rettungsband* kopiert M' zunächst das Eingabewort.[3]
Wir nummerieren im Graph des Programms von M alle Pfeile durch mit den Zahlen aus der Menge $\Xi = \{1, \dots, n\}$. Das Bandalphabet von M' ist dasselbe wie das von M, erweitert um die Symbole aus Ξ, also $\Gamma' = \Gamma \cup \Xi$.
Auf dem Band der Entscheidungen stehen (neben $\square$) nur Symbole aus Ξ, also Worte aus Ξ^*. Man kann diese Worte auf dem Band der Entscheidungen als *Folge von Entscheidungen auffassen, wie die* NTM M *durch ihr Programm läuft*. Genauer gehört zu jeder Berechnung $C_{\text{init}}(w) \vdash_M^* C$ genau ein Wort aus Ξ^*, das angibt, wie das Programm von M durchlaufen wird. Umgekehrt muss aber nicht jedem Wort aus Ξ^* eine Berechnung entsprechen.
Wir nummerieren die Worte aus $\Xi^* = \{\xi_0, \xi_1, \xi_2, \dots\}$ gemäß der Standardordnung durch (erst das leere Wort, dann alle Worte der Länge 1, dann alle Worte der Länge 2 und so weiter).
Die Maschine M arbeitet nun wie folgt:[4]

```
1  kopiere das Eingabewort auf das Eingabe-Rettungsband
2  for i ← 0,1,2,3,... do
3    Band_der_Entscheidungen ← ξ_i
4    // Eigentlich zählt man einfach das Band_der_Entscheidungen hoch
5
6    lösche die ersten k Bänder
7    kopiere das Wort vom Eingabe-Rettungsband auf das Eingabeband
8
9    iteriere über die Pfeile i ∈ Ξ auf dem Band_der_Entscheidungen
10     versuche, die Inhalte der ersten k Bänder gemäß dem Pfeil i zu ändern
11     // Der Versuch kann schiefgehen, wenn der Pfeil gar nicht zum
12     // Zustand und den gelesenen Zeichen passt
13     if Änderung nicht möglich, da Pfeil nicht passend then
14       // Abbruch, diese Berechnung war nichts
15       break Iteration
16     else
17       if akzeptierende Endkonfiguration erreicht then
18         stop and accept
```

Es bleibt zu zeigen, dass $L(M) = L(M')$ gilt.[5] Sei zunächst $w \in L(M)$. Dann gibt es eine akzeptierende Berechnung $C_{\text{init}}(w) \vdash_M^* C$. Zu dieser gehört ein Wort $\xi_i \in \Xi^*$, das dieser Berechnung entspricht, für ein geeignetes i. Dann wird die Maschine M' beim i-ten Durchlauf ihrer For-Schleife auf ihrem Band der Entscheidungen gerade ξ_i stehen haben. Bei diesem Durchlauf wird sie die ersten k Bänder in jedem Schritt ihrer Iteration so verändern, dass am Ende auf diesen Bändern gerade die Konfiguration C steht. Dann wird aber der Test, ob eine akzeptierende Endkonfiguration erreicht ist, zutreffen und M' akzeptiert w. Also gilt $w \in L(M')$.

Skript

[1] Rezept »Konstruktiver Beweis«. Zunächst kommt die Konstruktion.

[2] Originelle Namen verhindern, dass der Leser einschläft.

[3] Wie das genau geht wird nicht beschrieben. Ist auch *sehr* fummelig.

[4] Eigentlich muss man erklären, wie der Pseudocode tatsächlich als Turing-Maschinen-Programm aussieht. Das geht, würde aber mehrere Seiten benötigen.

[5] Zweiter Teil des Rezepts »Konstruktiver Beweis«.

Für die zweite Richtung möge M' ein Wort w akzeptieren. Dies ist nur dann der Fall, wenn es ein i gibt, so dass für ξ_i auf dem Band der Entscheidung die akzeptierende Endkonfiguration erreicht wird. Dann muss aber ξ_i einer Berechnung von M entsprechen, die M in einer akzeptierende Endkonfiguration bringt. Also gilt $w \in L(M)$. □

Zusammenfassung dieses Kapitels

1. Bei NTMs kann es zu jedem Zustand mehrere Nachfolgezustände geben. 13-16
2. Eine NTM akzeptiert ein Wort, wenn es eine akzeptierende Berechnung *gibt*.
3. »Ein Wort nichtdeterministisch raten« bedeutet lediglich, dass es für jedes mögliche Wort eine Berechnung gibt, bei der dieses Wort auf einem speziellen Band steht – geraten wird hier nicht wirklich.
4. Zu jeder NTM gibt es eine äquivalente DTM.

Zum Weiterlesen

[1] R. Reischuk. *Unterlagen zum Lehrmodul Theoretische Informatik,* Vorlesungsskript, 2008. Kapitel 4.1, 4.2, 4.3.

Übungen zu diesem Kapitel

Übung 13.1 Turing-Maschine für das Komplement der Sprache der Palindrome, mittel

Es sei $L' = \{w \in \{0,1\}^* \mid w \text{ ist kein Palindrom}\}$ gegeben. Konstruieren Sie eine 2-Band-NTM, die L' entscheidet. Gehen Sie hierbei nach folgendem Algorithmus vor:

- Raten Sie eine Stelle i.
- Merken Sie sich das Zeichen $w[i]$.
- Merken Sie sich den Wert von i.
- Gehen Sie zum Schluss der Eingabe.
- Bewegen Sie sich $i-1$ Zeichen zurück.
- Vergleichen Sie $w[|w|-i+1]$ mit $w[i]$.

Übung 13.2 Sprache der Quadrate, mittel

Es sei $L = \{ww \mid w \in \{0,1\}^*\}$ gegeben. Konstruieren Sie eine 2-Band-NTM, die L akzeptiert, und beweisen Sie die Korrektheit Ihrer Konstruktion.

Übung 13.3 Nichtdeterministische Turingmaschinen, mittel

Es sei $L = \{w \in \{0,1\}^* \mid \exists i \in \{1,\ldots,|w|\}\colon w[i] = w[|w|-i+1]\}$ gegeben. Konstruieren Sie eine 2-Band-NTM, die diese Sprache entscheidet.

14-1

Kapitel 14
Maschinen III: Register-Maschinen
Wenn Theoretiker Computer bauen

14-2

Lernziele dieses Kapitels

1. Syntax und Semantik der RAM verstehen
2. Einfache RAM-Programme schreiben können
3. Äquivalenzbeweis zu Turingmaschinen verstehen

Inhalte dieses Kapitels

Worum es heute geht

Wie schon im Vorwort ausführlich dargelegt, versucht sich die Theoretischen Informatik sich mit den alltäglichen Problemen der Praktischen oder gar Technischen Informatiker möglichst nicht herumzuschlagen. Wir betrachten *Modelle*, bei denen es solche Absonderlichkeiten wie A20-Gates oder Non-Maskable-Interrupts einfach nicht gibt. Ihre Kulmination hat eine wahrhafte Abstraktionsorgie in der Turing-Maschine gefunden – ausgehend von der Vorstellung eines *realen Menschen* (was ein, mit Verlaub, noch viel komplexeres Wesen ist als ein Computer) ist man bei einem Modell angekommen, das doch in seiner Abstraktheit und Einfachheit kaum noch zu unterbieten ist. (Tatsächlich geht es auch noch einfacher, aber dazu in einem späteren Kapitel.)

Es ist nun an der Zeit, wieder etwas zurückzurudern. Treibt man es nämlich mit der Abstraktion zu weit, dann besteht die Gefahr, dass die bewiesenen Sätze einem in der Praxis nur bedingt weiterhelfen – die realen Probleme sind wegabstrahiert worden. So werden in der Theoretischen Informatik beispielsweise mit viel Liebe Algorithmen sehr gründlich und detailliert untersucht, deren Laufzeit so katastrophal hoch ist, dass sie schon bei der Eingabelänge 1 (in Worten: Eins) auf einem Quanten-Computer so groß wie das Universum so lange rechnen würden, wie das Universum alt ist.

In diesem Kapitel geht es um ein Maschinen-Modell, das *wesentlich* näher am realen Computer ist als die Turing-Maschine. *Register-Maschinen* verfügen über einen Speicher, der ähnlich einem realen Computerspeicher aus einzelnen Zellen aufgebaut ist, die Zahlen speichern. Weiterhin verfügen Register-Maschinen über einen Befehlssatz, der sehr dicht dran ist an dem typischer Prozessoren. Natürlich liegt auch Register-Maschinen noch ein gewisser Grad an Abstraktion zu Grunde. So kann man – völlig unrealistischer Weise – beliebig

große Zahlen in jeder Speicherzelle speichern. Weiterhin ist der Befehlssatz noch etwas minimalistisch verglichen mit einem Pentium-Prozessor.

Die entscheidende Frage ist nun, wie sich das etwas realistischere Modell von Computern zum Modell der Turing-Maschine verhält. Hier werden wir einen sehr befriedigenden Satz beweisen: Diese beiden Modelle sind genau gleich mächtig – was Turing-Maschinen können, können auch Register-Maschinen; und umgekehrt. Die vom praktischen Standpunkt aus recht nützliche Konsequenz aus diesem Satz ist, dass wir *es uns immer wieder neu aussuchen können,* ob wir lieber mit Turing-Maschinen oder mit Register-Maschinen arbeiten wollen. Will man theoretische Resultate beweisen, so sind Turing-Maschinen oft einfacher zu handhaben; will man praktische Algorithmen untersuchen, so sind Register-Maschinen besser. Sie haben die Wahl.

14.1 Das RAM-Modell

14.1.1 Reale Computer versus die Turing-Maschine

Turing-Maschinen sind unrealistische Modelle von Computern. 14-4

Turing-Maschinen haben einige *Gemeinsamkeiten* mit realen Computern:

- Es gibt einen *Speicher* (die Bänder / das RAM).
- Der Speicher besteht aus *Zellen* (die Symbole enthalten / die ein Byte enthalten).
- Es gibt *Programme*, die *sequentiell* abgearbeitet werden.
- In dem Programmen sind *Sprünge* möglich.

Es gibt aber auch wichtige *Unterschiede*:

- Ein realer Rechner kann jede beliebige Speicherstelle aufgrund ihres Index in einem Schritt ansprechen.
- Ein realer Computer kann Operationen wie »Addition« oder oft auch »Multiplikation« in einem Schritt ausführen.

Unser heutiges Ziel: Ein mathematisches Modell eines realen Rechners. 14-5

- Das *Random-Access-Maschine-Modell* (RAM-Modell) ist ein mathematisches Modell eines realen Computers.
- Es ist wesentlich *»realistischer«* als die Turing-Maschine.
- Es ist aber trotzdem nur ein *Modell*.

14.1.2 Die Ideen

Die Bestandteile einer RAM. 14-6

Eine RAM besteht aus:

- Einem *Speicher* und
- einem *Programm*.

Weiterhin hat eine RAM eine

- *Eingabe* und in der Regel auch eine
- *Ausgabe*.

Die Ein- und Ausgaben stehen anfangs beziehungsweise am Ende in besonderen Bereichen des Speichers.

14-7

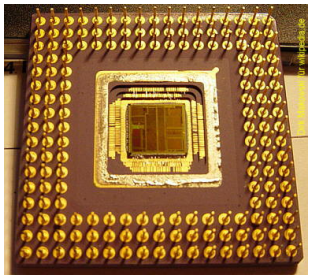
Der Speicher einer RAM.

- Der Speicher einer RAM besteht aus *Registern*: R0, R1, R2, R3, ...
- Prinzipiell gibt es *unendlich viele*.
- Jedes Register kann *eine natürliche Zahl speichern*.
- Anfangs (nachdem der »Strom eingeschaltet wurde«) steht in allen Registern 0.

Das ist natürlich wiederum »etwas unrealistisch«, aber es ist unklar, welche Obergrenze man für die Anzahl oder die Inhalte der Register festlegen sollte.

14-8

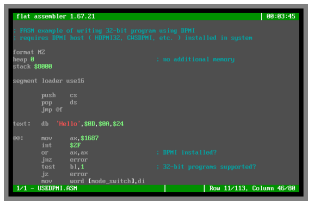
Die CPU einer RAM.



- Die CPU einer RAM arbeitet ein *festes Programm* ab.
- Dabei greift sie in jedem Schritt auf einige der Register zu.
- Sie kann von Schritt zu Schritt auf *völlig unterschiedliche* Register zugreifen – daher der Name »random access machine«.

14-9

Das Programm einer RAM.



- Ein RAM-*Programm* liest sich wie *Assembler-Code* eines realen Computers.
- Wie bei einem realen Computer muss man mehrere Dinge festlegen:
  - Welcher *Befehlssatz* ist erlaubt?
  - Welche *Adressierungsarten* sind erlaubt?

14-10

Adressierungsarten einer RAM.

Adressierungsarten

- Die *Adressierungsarten* eines Computers regeln, wie man angeben kann, welches Speicher-Register man adressieren möchte.
- Bei der *direkten Adressierung* gibt man im Programm fest eine Registernummer an (zum Beispiel einfach R5).
- Bei der *indirekten Adressierung* gibt man im Programm an, *in welchem Register die Nummer des Registers ist, das man eigentlich meint*.
  Beispiel: RR5 meint »den Inhalt von Register R*i*, wobei *i* gerade der Inhalt von R5 ist«.

- Bei der *klassischen Definition* einer RAM ist nur die *direkten Adressierung* erlaubt.
- Wir werden aber auch die *indirekte Adressierung* benutzen, was ein wesentlich realistischeres Modell liefert.

14-11

Befehlsatz einer RAM.

Befehlssatz

- Der *Befehlssatz* eines Computers regelt, welche Befehle »in einem Schritt« ausgeführt werden können.
- Bei einer RISC-Maschine (reduced instruction set computing) versucht man, einen möglichst *kleinen* Befehlssatz zu benutzen.
- Bei einer CISC-Maschine (complex instruction set computing) versucht man, für möglichst jedes Problem einen Befehl zu haben.

- Bei der *klassischen Definition* einer RAM versucht man eher einen RISC-Ansatz und beschränkt sich auf *ganz wenige Befehle*.
- In diesem Skript werden eher viele Befehle zugelassen, aber längst noch kein CISC-Befehlssatz.

14-12

Ein- und Ausgaben einer RAM.

- Da RAMs mit Registern voller *Zahlen* arbeiten, sind die *Eingaben* einer RAM normalerweise Tupel von natürlichen Zahlen.
- Diese schreibt man in die ersten Register.
- Ebenso sind die *Ausgaben* Tupel von Zahlen, die entsprechend am Ende einer Berechnung wieder in den ersten Registern stehen.

14.1.3 Syntax

Die Syntax eines RAM-Programms. 14-13

▶ Definition: RAM-Programm

Ein RAM-*Programm* besteht aus einer endlichen, durchnummerierten Folge von Befehlen, wobei die Zählung mit 1 beginnt. In Befehlen kommen *Registeradressierungen* vor, wovon es genau folgende zwei Arten gibt (im Folgenden seien immer $i, j, k \in \mathbb{N}$):

1. Ri bezeichnet eine *direkte Adressierung,*
2. RRi bezeichnet eine *indirekte Adressierung.*

Folgende Befehle sind möglich:

- Die Transportbefehle
  1. R$i \leftarrow$ Rj
  2. R$i \leftarrow$ RRj
  3. RR$i \leftarrow$ Rj
- Die arithmetischen Befehle
  4. R$i \leftarrow k$
  5. R$i \leftarrow$ R$j +$ Rk
  6. R$i \leftarrow$ R$j -$ Rk
  7. R$i \leftarrow \lfloor \frac{1}{2}Rj \rfloor$
- Die Sprungbefehle
  8. ***if*** R$i = 0$ ***goto*** k
  9. ***if*** R$i > 0$ ***goto*** k
  10. ***goto*** k

  Bei allen Sprungbefehlen muss k die Nummer einer Programmzeile sein.
- Der Stoppbefehl
  11. ***stop***

  Der letzte Befehl eines Programms muss der Stoppbefehl sein.

Beispiele von RAM-Programmen 14-14

Beispiel: Berechnung des Durchschnitts der Register R1 bis R4

```
1 R0 ← R1
2 R0 ← R0 + R2
3 R0 ← R0 + R3
4 R0 ← R0 + R4
5 R0 ← ⌊1/2 R0⌋
6 R0 ← ⌊1/2 R0⌋
7 stop
```

Beispiel: Summe einer Menge von Register

Nehmen wir an, in R1 und R2 stehen ein Startindex $i > 3$ und ein Stoppindex $j > i$. In R0 soll am Ende die Summe der Register von Ri bis R$(j-1)$ stehen.

```
1 R0 ← 0
2 R3 ← R2 − R1        // while R1 < R2
3 if R3 = 0 goto 9    // do {
4 R3 ← RR1
5 R0 ← R0 + R3        // R0 += RR1
6 R3 ← 1
7 R1 ← R1 + R3        // R1++
8 goto 2              // }
9 stop
```

Vorführung des Verhaltens der obigen RAM: Je ein Student einer Reihe repräsentiert ein Register. Sie schreiben sich Zahlen auf, die ihre Werte repräsentieren. Dann läuft das Programm ab. Regie

14-15

✎ Zur Übung

1. Geben Sie ein RAM-Programm an, das das Produkt der Register R1 und R2 in das Register R3 schreibt.
2. Wenn sie dies geschafft haben, versuchen Sie Ihr Programm *schnell* zu machen – es soll also möglichst wenige Rechenschritte machen.
   Wenn die Werte von R1 und R2 kleiner als eine Million sind, kann ein schnelles Programm diese in höchstens 100 Schritten multiplizieren.

14.1.4 Semantik

14-16

Die Semantik eines RAM-Programms.

- Um die Semantik von RAM-Programmen formal zu fassen, benutzen wir, wenig überraschend, *Konfigurationen*.
- Für die Semantik muss man dann nur erklären, welchen Effekt jeder einzelne Befehl hat.
- Später müssen wir dann noch klären, »wie das mit den Ein- und Ausgaben« funktionieren soll.

14-17

Konfiguration einer RAM.

▶ Definition: Konfiguration
Eine *Konfiguration* einer RAM gibt für jedes Register seinen Inhalt an sowie den Wert des *Programmzählers* (also die aktuelle Zeilennummer). Formal ist also eine Konfiguration ein Paar bestehend aus

- dem Programmzähler und
- einer Abbildung von $\mathbb{N}$ nach $\mathbb{N}$.

Bei einer *Endkonfiguration* ist der Befehl, auf den der Programmzähler zeigt, gerade ***stop***.

▶ Notation
Sei C die Konfiguration einer RAM. Wir schreiben dann

- $\langle \mathrm{R}i \rangle_C$ für den *Inhalt* des i-ten Registers in dieser Konfiguration und
- $\langle \mathrm{PC} \rangle_C$ für den Wert des Programmzählers.

Den Index lassen wir in der Regel auch weg.

14-18

Berechnungschritte einer RAM.

▶ Definition: Berechnungsschritt
Sei Π ein RAM-Programm und C eine Konfiguration, die keine Endkonfiguration ist. Dann geht hieraus *in einem Berechnungsschritt* die Konfiguration C' hervor, die identisch zu C ist, mit folgenden Änderungen:

Zeile $\langle \mathrm{PC} \rangle$	*Wirkung*
$\mathrm{R}i \leftarrow \mathrm{R}j$	$\langle \mathrm{PC} \rangle' = \langle \mathrm{PC} \rangle + 1$, $\langle \mathrm{R}i \rangle' = \langle \mathrm{R}j \rangle$
$\mathrm{R}i \leftarrow \mathrm{RR}j$	$\langle \mathrm{PC} \rangle' = \langle \mathrm{PC} \rangle + 1$, $\langle \mathrm{R}i \rangle' = \langle \mathrm{R}\langle \mathrm{R}j \rangle \rangle$
$\mathrm{RR}i \leftarrow \mathrm{R}j$	$\langle \mathrm{PC} \rangle' = \langle \mathrm{PC} \rangle + 1$, $\langle \mathrm{R}\langle \mathrm{R}i \rangle \rangle' = \langle \mathrm{R}j \rangle$
$\mathrm{R}i \leftarrow k$	$\langle \mathrm{PC} \rangle' = \langle \mathrm{PC} \rangle + 1$, $\langle \mathrm{R}i \rangle' = k$
$\mathrm{R}i \leftarrow \mathrm{R}j + \mathrm{R}k$	$\langle \mathrm{PC} \rangle' = \langle \mathrm{PC} \rangle + 1$, $\langle \mathrm{R}i \rangle' = \langle \mathrm{R}j \rangle + \langle \mathrm{R}k \rangle$
$\mathrm{R}i \leftarrow \mathrm{R}j - \mathrm{R}k$	$\langle \mathrm{PC} \rangle' = \langle \mathrm{PC} \rangle + 1$, $\langle \mathrm{R}i \rangle' = \max\{\langle \mathrm{R}j \rangle - \langle \mathrm{R}k \rangle, 0\}$
$\mathrm{R}i \leftarrow \lfloor \frac{1}{2}\mathrm{R}j \rfloor$	$\langle \mathrm{PC} \rangle' = \langle \mathrm{PC} \rangle + 1$, $\langle \mathrm{R}i \rangle' = \lfloor \langle \mathrm{R}j \rangle / 2 \rfloor$
goto k	$\langle \mathrm{PC} \rangle' = k$,
if $\mathrm{R}i = 0$ ***goto*** k	$\langle \mathrm{PC} \rangle' = k$ für $\langle \mathrm{R}i \rangle = 0$, sonst $\langle \mathrm{PC} \rangle' = \langle \mathrm{PC} \rangle + 1$
if $\mathrm{R}i > 0$ ***goto*** k	$\langle \mathrm{PC} \rangle' = k$ für $\langle \mathrm{R}i \rangle > 0$, sonst $\langle \mathrm{PC} \rangle' = \langle \mathrm{PC} \rangle + 1$

Wir schreiben dann $C \vdash_\Pi C'$.

Interludium: Partielle Funktionen

14-19

RAMs können, genau wie Turing-Maschinen, leicht in Endlosschleifen geraten.

- In solchen Fällen produzieren sie offenbar keine Ausgaben.
- Formal bedeutet dies, dass der Eingabe »nichts« zugeordnet werden kann.
- Dies ist entspricht genau dem Verhalten von partiellen Funktionen, weshalb sich diese zur Modellierung besonders eignen.

▶ **Notation:** Schreibweisen für partielle Funktionen

Eine *partielle Funktion* $f\colon A \dashrightarrow B$ ordnet manchen (aber nicht unbedingt allen) Elementen von A ein Element aus B zu.

- Falls f dem Element a etwas zuordnet, so schreiben wir $f(a)\downarrow$.
- Falls f dem Element a nichts zuordnet, so schreiben wir $f(a)\uparrow$.

Start einer Berechnung

14-20

Wie schon erwähnt, sind die natürlichen Eingaben und Ausgaben von RAMs *Tupel von Zahlen*.

▶ **Definition:** Anfangskonfiguration

Sei $(x_1,\dots,x_n) \in \mathbb{N}^n$ ein Tupel von natürlichen Zahlen. Die zugehörige *Anfangskonfiguration* $C_{\text{init}}(x_1,\dots,x_n)$ ist wie folgt definiert:

1. $\langle \text{PC} \rangle = 1$,
2. $\langle \text{R}i \rangle = x_i$ für $i \in \{1,\dots,n\}$,
3. Für alle anderen $j \in \{0, n+1, n+2, \dots\}$ gilt $\langle \text{R}j \rangle = 0$.

Von RAMs berechenbare Funktionen

14-21

▶ **Definition**

Sei $f\colon \mathbb{N}^n \dashrightarrow \mathbb{N}^m$ eine partielle Funktion. Dann heißt f RAM-*berechenbar*, wenn ein RAM-Programm Π existiert, so dass für alle $(x_1,\dots,x_n) \in \mathbb{N}^n$ gilt:

- Ist f *definiert* für das Tupel $(x_1,\dots,x_n)$, gilt also $f(x_1,\dots,x_n)\downarrow$, so *muss es eine Endkonfiguration* C geben mit folgenden Eigenschaften:
  1. $C_{\text{init}}(x_1,\dots,x_k) \vdash_\Pi^* C$ und
  2. $f(x_1,\dots,x_n) = \big(\langle \text{R}1 \rangle_C, \dots, \langle \text{R}m \rangle_C\big)$. (Sprich: In der Endkonfiguration steht in den ersten m Registern gerade das Tupel, auf das die Funktion die Eingaben abbildet.)
- Ist f *nicht definiert* für das Tupel $(x_1,\dots,x_n)$, gilt also $f(x_1,\dots,x_n)\uparrow$, so *darf es keine Endkonfiguration mit* $C_{\text{init}}(x_1,\dots,x_n) \vdash_\Pi^* C$ *geben*. (Sprich: Die Maschine geht in eine Endlosschleife.)

Beispiel einer RAM-berechenbaren Funktion

14-22

Beispiel

Die Funktion $f\colon \mathbb{N}^2 \to \mathbb{N}$ mit $f(x,y) = x \cdot y$ ist RAM-berechenbar. Dies zeigt das folgende Programm:

```
1  R3 ← 1
2  if R1 = 0 goto 6
3  R0 ← R0 + R2
4  R1 ← R1 − R3
5  goto 2
6  R1 ← R0
7  stop
```

14-23

Beispiel einer partiellen RAM-berechenbaren Funktion

Beispiel
Die Funktion $f\colon \mathbb{N}^2 \dashrightarrow \mathbb{N}$ mit

$$f(x,y) = \begin{cases} x-y, & \text{für } x \geq y, \\ \text{undefiniert} & \text{sonst,} \end{cases}$$

ist RAM-berechenbar:

```
1 R3 ← R2 − R1
2 if R3 > 0 goto 2
3 R1 ← R1 − R2
4 stop
```

14-24

✎ **Zur Übung**
Welche der folgenden Funktionen sind RAM-berechenbar?

1. $a(x,y) = x^y$
2. $b(x) = x!$
3. $c(x,y) = (y,x)$
4. $d(x) =$ die x-te Stelle von π
5. $e(x) =$ die x-te Primzahl
6. $$f(x) = \begin{cases} x, & \text{falls } x \text{ prim ist,} \\ \text{undefiniert} & \text{sonst.} \end{cases}$$

14.2 Verhältnis von Zahlen und Worten

14.2.1 Worte als Eingabe für RAMs

14-25

Wie kommen Wörter in die RAM?

- Eine RAM erwartet *Zahlen* als Eingabe.
- Bis jetzt haben wir aber – aus guten Gründen – immer mit *Worten* und *Sprachen* gearbeitet.
- Wir brauchen also eine Methode, um *Worte in* RAMs *einzuspeisen*.

✎ **Zur Diskussion**
Wie könnte man ein Wort wie `abba` als Eingabe einer RAM zuführen?

14-26

Pro Register ein Buchstabe.

▶ **Definition:** Anfangskonfiguration für Worte
Sei Σ ein Alphabet und $\kappa\colon \Sigma \to \{1,\dots,|\Sigma|\}$ eine Bijektion. (Mit anderen Worten: κ nummeriert die Buchstaben in Σ durch.) Sei $w \in \Sigma^*$. Wir definieren dann die *Anfangskonfiguration* $C_{\text{init}}(w)$ wie folgt:

- $\langle \text{R}0 \rangle = |w|$.
- $\langle \text{R}(i+3) \rangle = \kappa(w[i])$ für $i \in \{1,\dots,|w|\}$.

Das reichlich überraschende »+3« liegt daran, dass man unter Umständen am Anfang einige Register braucht »zum Arbeiten«, siehe Übung 14.2.

Beispiel
Sei $\Sigma = \{a,b,c\}$ und $\kappa(a) = 1$, $\kappa(b) = 2$, $\kappa(c) = 3$. Dann ist lautet $C_{\text{init}}(abba)$ wie folgt:

$\langle \text{R}0 \rangle$	$\langle \text{R}1 \rangle$	$\langle \text{R}2 \rangle$	$\langle \text{R}3 \rangle$	$\langle \text{R}4 \rangle$	$\langle \text{R}5 \rangle$	$\langle \text{R}6 \rangle$	$\langle \text{R}7 \rangle$	$\langle \text{R}8 \rangle$	…
4	0	0	0	1	2	2	1	0	…

Akzeptierbarkeit mit RAMs.

14-27

- Wir haben jetzt geklärt, wie Worte in eine RAM hineinkommen.
- Da RAMs keine »akzeptierenden Zustände« haben, müssen wir noch klären, was »akzeptierende Endkonfigurationen« sein sollen.

▶ **Definition:** Akzeptierende Endkonfiguration, akzeptiere Worte und Sprache
Sei Σ ein Alphabet und Π ein RAM-Programm.

- Eine Endkonfiguration heißt *akzeptierend*, wenn $\langle \mathrm{R}0 \rangle = 1$ gilt.
- Ein Wort $w \in \Sigma^*$ *wird akzeptiert*, wenn es eine akzeptierende Endkonfiguration C mit $C_{\text{init}}(w) \vdash_\Pi^* C$ gibt.
- Die Menge aller von Π akzeptierten Worte ist die *von* Π *akzeptierte Sprache* $L(\Pi)$.

▶ **Definition:** RAM-akzeptierbare und RAM-entscheidbare Sprache
Eine Sprache L heißt RAM-*akzeptierbar*, wenn $L = L(\Pi)$ für ein RAM-Programm Π gilt. Hält Π auf jeder Eingabe an, so heißt L auch RAM-*entscheidbar*.

Beispiel
Die Sprache $\{a^n b^n \mid n \geq 1\}$ ist RAM-entscheidbar:

- In einer ersten Schleife, zähle wie viele Register mit 1 belegt sind ab R4.
- In einer zweiten Schleife, zähle wie viele Register mit 2 belegt sind danach.
- Wenn diese Zahlen gleich sind und in der Summe gerade $\langle \mathrm{R}0 \rangle$ ergeben, so akzeptiere $(\mathrm{R}0 \leftarrow 1)$, sonst verwerfe $(\mathrm{R}0 \leftarrow 0)$,

14.2.2 Funktionsberechnungen bei Turing-Maschinen

Wie kommen Ausgaben aus der Turing-Maschine?

14-28

- Eine RAM kann (partielle) *Funktionen* berechnen.
- Unsere Turing-Maschinen können bis jetzt hingegen nur *Sprachen akzeptieren*.

▶ **Definition:** Turing-Maschine mit Ausgaben
Eine *Turing-Maschine mit Ausgabe* ist eine DTM M, bei der eines der Bänder (typischerweise das zweite oder das letzte) als *Ausgabeband* bezeichnet wird. Die von M *berechnete Funktion* $f\colon \Sigma^* \dashrightarrow \Sigma^*$ ist wie folgt definiert:

- Falls M bei Eingabe w nach endlich vielen Schritten anhält und auf dem Ausgabeband (außer den Blanks am Anfang und Ende) gerade ein Wort aus Σ^* steht, so ist $f(w)$ gerade diese Wort.
- Anderenfalls ist $f(w)$ undefiniert (wenn M also in eine Endlosschleife gerät oder komische Symbole auf das Ausgabeband schreibt).

Wir sagen dann, *f ist Turing-berechenbar*.

Beispiele von Turing-berechenbaren Funktionen.

14-29

Beispiel
Folgende Funktionen sind Turing-berechenbar:

1. $f(w) = w^{\text{rev}}$
2. $f(w) = ww$
3. $$f(w) = \begin{cases} \text{bin}(i+j), & \text{falls } w = \text{bin}(i)+\text{bin}(j), \\ \text{undefiniert} & \text{sonst.} \end{cases}$$

14.2.3 Zahlen als Eingaben für Turing-Maschinen

14-30

Zahlen als Eingaben für Turing-Maschinen

- Wir haben Worte so »aufbereitet«, dass RAMs sie verarbeiten können.
- Da ist es nur fair, auch Zahlen zu »aufzubereiten«, dass Turing-Maschinen sie verarbeiten können.
- Zur Erinnerung: Die Funktion $\text{bin}: \mathbb{N} \to \{0,1\}^*$ bildet natürliche Zahlen auf ihre Darstellung als Binärstring ab. Beispielsweise gilt $\text{bin}(5) = 101$ und $\text{bin}(0) = 0$.

▶ Definition

Sei $f: \mathbb{N}^n \dashrightarrow \mathbb{N}^m$ eine partielle Funktion. Wir definieren dann die Funktion $f_{\text{bin}}: \{0,1,\#\}^* \dashrightarrow \{0,1,\#\}^*$ wie folgt:

- Ist $w = \text{bin}(x_1)\#\ldots\#\text{bin}(x_n)$ und ist $f(x_1,\ldots,x_m)$ definiert, so ist $f_{\text{bin}}(w) = \text{bin}(y_1)\#\ldots\#\text{bin}(y_m)$, wobei $f(x_1,\ldots,x_n) = (y_1,\ldots,y_m)$.
- In allen anderen Fällen ist $f_{\text{bin}}(w)$ undefiniert.

14.3 Äquivalenzsatz

14.3.1 Die Ideen

14-31

Was unterscheidet Turing-Maschinen und Register-Maschinen?

Vergegenwärtigen wir uns nochmal die Unterschiede von Turing-Maschinen und Register-Maschinen:

- Register-Maschinen können in einem Schritt auf »beliebige Stellen« im Speicher zugreifen – Turing-Maschinen nicht.
- Register-Maschinen können in einem Schritt zwei Zahlen addieren – Turing-Maschinen nicht.

Diese Unterschiede erscheinen bei genauerem Hinsehen aber auch nicht »unüberwindlich«:

- Will eine Turing-Maschine auf »eine beliebige Stelle« im Speicher zugreifen, so *muss sie eben spulen*.
- Will eine Turing-Maschine zwei Zahlen addieren, so *muss sie eben etwas rechnen*.

Ziel

Beweisen, dass Turing-Maschinen und Register-Maschinen genau gleichmächtig sind.

14.3.2 Der Satz

14-32

Turing-berechenbar ist dasselbe wie RAM-berechenbar.

▶ Satz

Sei $f: \mathbb{N}^n \dashrightarrow \mathbb{N}^m$ eine partielle Funktion. Dann sind folgende Aussagen äquivalent:

1. *f ist RAM-berechenbar.*
2. *f_{bin} ist Turing-berechenbar.*

▶ Folgerung

Eine Sprache L ist genau dann RAM-akzeptierbar, wenn sie (Turing-)akzeptierbar ist.

▶ Folgerung

Eine Sprache L ist genau dann RAM-entscheidbar, wenn sie (Turing-)entscheidbar ist.

Beweis des Satzes.

14-33

Kommentare zum Beweis

[1] Rezept »Zwei Richtungen«

[2] Rezept »Konstruktiver Beweis«

Beweis. Wir beweisen zwei Richtungen.[1] Sei zunächst f RAM-berechenbar via eines RAM-Programms Π. Wir müssen zeigen, dass das Verhalten von Π durch eine geeignete Turing-Maschine M simuliert werden kann.[2]

Wir beschreiben zunächst den groben Aufbau der Maschine:

- Sie hat ein Eingabeband, ein Ausgabeband und Arbeitsbänder.
- Ihr Programm besteht aus mehreren Unterprogrammen und Hilfsprogrammen.
- Für jede Programmzeile von Π gibt es ein Unterprogramm in M.
- Die Hilfsprogramme werden genutzt zum Kopieren oder Rechnen.

Auf dem Eingabeband werden die *Inhalte der Register* ständig gespeichert. Dazu stehen einfach die Inhalte der Register in *Binärdarstellung* nebeneinander, Register werden durch das #-Zeichen getrennt:

$\langle \mathrm{R}0 \rangle$	$\langle \mathrm{R}1 \rangle$	$\langle \mathrm{R}2 \rangle$	$\langle \mathrm{R}3 \rangle$	$\langle \mathrm{R}4 \rangle$	$\langle \mathrm{R}5 \rangle$	$\langle \mathrm{R}6 \rangle$	$\langle \mathrm{R}7 \rangle$	$\langle \mathrm{R}8 \rangle$	…
4	0	0	0	1	2	2	1	0	…

wird zu dem Bandinhalt

$$\cdots\square 100\#0\#0\#0\#1\#10\#10\#1\square\ldots$$

Die (unendlich vielen) Register mit Inhalt 0 am Ende werden »nicht mitprotokolliert«. Wenn aber im Folgenden der Kopf auf das Blank-Symbol am Ende stößt, so ersetzt er dieses immer sofort durch #0 und fährt dann bei # fort. Wir beschreiben nun für jeden möglichen Befehl des RAM-Programms Π, wie die Turing-Maschine ihn simuliert: [3]

[3] Dies ist nur eine grobe Beschreibung. Details würden den Beweis unlesbar machen.

- $\mathrm{R}i \leftarrow \mathrm{R}j$
  Hierzu spult M auf dem Eingabeband an den Anfang zurück. Dann geht sie nach rechts, bis sie genau j viele #-Zeichen gesehen hat. Dann kopiert sie die folgenden Zeichen bis zum nächsten #-Zeichen auf ein Arbeitsband.
  Nun läuft sie wieder an den Anfang und geht nach rechts bis zum i-ten #-Zeichen. Die auf dieses #-Zeichen folgenden Symbole überschreibt sie mit dem Inhalt des Arbeitsbandes.
  Falls sich beim Kopieren herausstellt, dass »zu wenig oder zu viel Platz« für den Inhalt des Arbeitsbandes vorliegt, so verschiebt die Maschine in Schleifen den Rest des Eingabebandes, so dass alles wieder passt.
- $\mathrm{RR}i \leftarrow \mathrm{R}j$
  Das Finden und Kopieren des Inhalts von $\mathrm{R}j$ funktioniert wie bei $\mathrm{R}i \leftarrow \mathrm{R}j$.
  Um das Register $\mathrm{RR}i$ zu finden, sucht die Maschine zunächst das i-te Register. Dann kopiert sie den Inhalt dieses Registers auf ein weiteres Arbeitsband.
  Dann geht sie wieder an den Anfang des Eingabebandes. Dort läuft sie nach rechts und subtrahiert für jedes gelesene #-Zeichen 1 von dem weiteren Arbeitsband. Sobald dort die Zahl 0 steht, ist $\mathrm{RR}i$ gefunden.
  Nun verfährt sie wie bei $\mathrm{R}i \leftarrow \mathrm{R}j$, um den Inhalt vom Arbeitsband an die richtige Stelle zu schreiben.
- $\mathrm{R}i \leftarrow \mathrm{RR}j$
  Dies funktioniert analog zu $\mathrm{RR}i \leftarrow \mathrm{R}j$, nur dass diesmal die Suche nach $\mathrm{RR}j$ der aufwendig Teil ist.
- $\mathrm{R}i \leftarrow k$
  Hierzu wird $\mathrm{R}i$ gesucht und dann dessen Inhalt durch $\mathrm{bin}(k)$ ersetzt.
- $\mathrm{R}i \leftarrow \mathrm{R}j + \mathrm{R}k$
  Hierzu wird erst $\mathrm{R}j$ gesucht und der Inhalt auf ein Arbeitsband kopiert. Dann wird $\mathrm{R}k$ gesucht und dessen Inhalt auf ein weiteres Arbeitsband kopiert.
  Dann werden die Inhalte in einer Schleife addiert auf einem weiteren Arbeitsband, siehe Übung 12.4. Dieses Resultat wird dann an die Stelle kopiert, wo $\mathrm{R}i$ gespeichert ist.
- $\mathrm{R}i \leftarrow \mathrm{R}j - \mathrm{R}k$
  Dies funktioniert analog, nur eben mit der Subtraktion, siehe Übung 12.5.
- $\mathrm{R}i \leftarrow \lfloor \frac{1}{2}\mathrm{R}j \rfloor$
  Dies funktioniert analog, es muss nur einfach das letzte Bit gestrichen werden.
- ***if*** $\mathrm{R}j > 0$ ***goto*** k
  Suche die Stelle, wo $\mathrm{R}j$ gespeichert ist. Falls dort keine 0 steht, so wechsle in das Unterprogramm, das für Zeile k zuständig ist (und sonst, wie üblich, in das Unterprogramm, das für die nächste Zeile zuständig ist).

- ***if*** $\mathrm{R}j = 0$ ***goto*** k
  Funktioniert analog.
- ***goto*** k
  Wechselt einfach direkt in das für k zuständige Programm.
- ***stop***
  Kopiert die Inhalte der Register R1 bis Rm auf das Ausgabeband und stoppt.

Die Korrektheit der Konstruktion zeigt man nun durch eine Induktion über die Anzahl der Berechnungsschritte.[3] [4]Für die zweite Richtung sei M eine Turingmaschine. Wir müssen diese durch eine Registermaschine simulieren.
Dazu machen wir es uns zunächst etwas einfacher und nehmen an, *dass M ein Band hat und dieses nur in eine Richtung unendlich ist, also ein »linkes Ende« hat*. Dann funktioniert die Simulation prinzipiell wie folgt:

[3] Etwas arg knapp, aber genauer geht das kaum, da die Konstruktion schon so schwammig war.

[4] Weiter geht es mit der anderen Richtung.

- In R0 speichern wir jeweils die aktuelle Kopfposition.
- Die Register R1 bis R5 lassen wir frei »zum Rechnen«.
- Ab R6 kommen dann die Inhalte der Zellen des Bandes.

Ein »Schritt« der Maschine M lässt sich nun leicht simulieren:

1. R1 $\leftarrow$ RR0
2. Untersuche den Inhalt von R1.
3. In Abhängigkeit des gefundenen Wertes tue:
   3.1 Setze R1 auf das zu schreibende Zeichen.
   3.2 RR0 $\leftarrow$ R1
   3.3 Erhöhe oder erniedrige R0 um 1 oder lasse R0 unverändert.
   3.4 Springe zu den Befehlen, die sich um den Nachfolgezustand kümmern.

Es sind noch folgende Probleme zu lösen:

- *Das Band der Maschine ist in der Regel beidseitig unendlich.*
  Die simuliert man wie folgt: Ab R6 werden nicht die Bandzellen mit den Indizes 0, 1, 2, … gespeichert, sondern 0, 1, -1, 2, -2, 3, -3, …. Dies macht das »Anpassen von R0« ein wenig schwieriger.
- *Es gibt mehrere Bänder.*
  Die simuliert man wie folgt: Nicht nur in R0, sondern auch in R1, R2, … werden Kopfpositionen für die verschiedenen Bändern gespeichert. Die Bandzellen fangen entsprechend später an. Die Bandzellen werden »im Interleaving-Verfahren« gespeichert: Wenn es beispielsweise drei Bänder gibt, so kommen in je drei aufeinander folgenden Registern die Inhalte von Bandzelle 0, dann in drei aufeinander folgenden Registern die Inhalte von Bandzelle 1 und so weiter.

Fassen wir kurz zusammen, was wir erreicht haben:[5]

[5] Nie eine schlechte Idee.

- Sind in den Registern einer RAM die Inhalte der Zellen der Turing-Maschine gespeichert, so können wir simulieren, wie sich die Maschine verhält.

Aber:

- Wir wollten zeigen, dass f RAM-berechenbar ist, wenn f_{bin} dies ist.
- Nun erhält also unsere RAM ihre Eingabe $(x_1, \ldots, x_n)$ in den ersten n Registern – die zu simulierende Turing-Maschine erwartet sie aber in der Form $\text{bin}(x_1)\#\cdots\#\text{bin}(x_n)$ auf ihrem Band.

Um also die Simulation der Maschine M durchzuführen, muss das RAM-Programm für Folgendes sorgen: Die Zahl x_1 darf eben nicht im Register R1 stehen, sondern die Bits dieser Zahl müssen, beginnend bei R6, in den nachfolgenden Registern stehen. Entsprechend müssen dann die Bits von x_2 in den darauf folgenden Registern stehen und so weiter.
Ein RAM-Programm kann diese Umwandlung mit einigen geeigneten Schleifen bewerkstelligen (was ein wenig fummelig ist). Analog kann sie am Ende auch die »Rückumwandlung« der auf dem Ausgabeband stehenden Bits in die Zahlen der Ausgaberegister bewerkstelligen. □

Zusammenfassung dieses Kapitels

1. Der *Speicher* einer Register-Maschinen besteht aus *Registern*, die jeweils *eine natürliche Zahl* speichern. 14-34
2. Register-Maschinen können mittels *direkter* und *indirekter* Adressierung auf beliebige Speicherstellen zurgreifen.
3. Der *Befehlssatz* einer Register-Maschine entspricht (etwas abstrakter) der einer CPU.
4. Eine Funktion ist genau dann RAM-*berechenbar*, wenn sie *Turing-berechenbar* ist.

Zum Weiterlesen

[1] R. Reischuk. *Unterlagen zum Lehrmodul Theoretische Informatik,* Vorlesungsskript, 2008. Kapitel 5.4, 6.6.

Übungen zu diesem Kapitel

Übung 14.1 RAM-Programm zur Maximumsberechnung, leicht

Geben Sie ein RAM-Programm an, das den maximalen Wert berechnet, der in einem der Register zwischen $i > 3$ und $j - 1$ steht. Hierbei liegen i in R1 vor und j in R2.

Übung 14.2 RAM-Programm zum Platz schaffen, mittel

Geben Sie ein RAM-Programm an, das Folgendes leistet: In R0 steht eine Zahl n (also $\langle \text{R0} \rangle = i$). Ihr Programm soll nun die Inhalte der Register R4 bis $R(3+i)$ in die Register R5 bis $\text{R}(4+i)$ verschieben.

Beispiel: War vorher der Inhalt der Register

$\langle \text{R0} \rangle$	$\langle \text{R1} \rangle$	$\langle \text{R2} \rangle$	$\langle \text{R3} \rangle$	$\langle \text{R4} \rangle$	$\langle \text{R5} \rangle$	$\langle \text{R6} \rangle$	$\langle \text{R7} \rangle$	$\langle \text{R8} \rangle$	…
4	0	0	0	1	2	2	1	0	…

so soll er hinterher wie folgt lauten:

$\langle \text{R0} \rangle$	$\langle \text{R1} \rangle$	$\langle \text{R2} \rangle$	$\langle \text{R3} \rangle$	$\langle \text{R4} \rangle$	$\langle \text{R5} \rangle$	$\langle \text{R6} \rangle$	$\langle \text{R7} \rangle$	$\langle \text{R8} \rangle$	…
4	0	0	0	0	1	2	2	1	…

Übung 14.3 RAM-Entscheidbarkeit, mittel

Geben Sie ein RAM-Programm an, das die Sprache $\{a^n b^n \mid n \geq 1\}$ entscheidet. Ein möglicher Algorithmus ist im Beispiel 14-27 beschrieben. Dokumentieren Sie Ihr Programm geeignet.

Übung 14.4 RAM-Entscheidbarkeit, schwer

Beweisen Sie die Korrektheit Ihres in Übung 14.3 konstruierten Programms. Benutzen Sie dazu folgendes Beweisrezept:

Beweisrezept: *Korrektheitsbeweise für RAMs* 14-35

Ziel

Man will beweisen, dass ein RAM-*Programm* Π *eine Sprache* L *akzeptiert oder entscheidet.*

Rezept

1. Beginne mit »Sei $w \in \Sigma^*$ beliebig. Wir untersuchen das Verhalten von Π ausgehend von $C_{\text{init}}(w)$.«
2. Gib nun (mehr oder weniger genau) die Folge von Konfigurationen an, die Π beginnend bei $C_{\text{init}}(w)$ durchlaufen wird. Es reicht in der Regel, nur anzugeben, wie Registerinhalte an bestimmten »spannenden« Stellen sind (zum Beispiel jeweils am Anfang einer Schleife).
3. Soll Akzeptierbarkeit gezeigt werden, ende mit »Folglich gilt, dass für $w \in L$ eine Endkonfiguration erreicht wird mit $\langle \text{R0} \rangle = 1$, und folglich $w \in L(M)$. Ist hingegen $w \notin L$, so wird entweder gar keine Endkonfiguration erreicht oder es gilt $\langle \text{R0} \rangle \neq 1$. Folglich gilt für $w \notin L$ auch $w \notin L(M)$.«
4. Soll Entscheidbarkeit gezeigt werden, ende mit »Folglich gilt, dass für alle $w \in \Sigma^*$ eine Endkonfiguration erreicht wird. Weiter gilt $w \in L$ genau dann, wenn dort $\langle \text{R0} \rangle = 1$ gilt. Also gilt $L = L(\Pi)$.«

Übung 14.5 Adressierungsarten bei verschiedenen Prozessoren, mittel

Bei einer RAM schreiben wir R1 ← RR5, um anzudeuten, dass eine indirekte Adressierung vorliegt.

Auch bei realen CPUs ist die indirekte Adressierung möglich. Finden Sie heraus, wie die Notation bei folgenden Prozessortypen aufgeschrieben wird (geben Sie jeweils einen typischen Befehl an und erläutern Sie kurz dessen Wirkung):

- Intels 8086-Familie (Pentium etc.).
- MOS-Technologies 6502.
- Motorolas 68000-Familie.
- Intels Itanium-Familie.
- Eine weitere Prozessorfamilie Ihrer Wahl.

Übung 14.6 RAM-Ganzzahlige Division mit Rest, mittel

Geben Sie ein RAM-Programm an, das Folgendes leistet: In R0 steht eine Zahl n und in R1 steht eine Zahl p. Ihr Programm soll nun das gannzzahlige Ergebnis der Division von n durch p in das Register R2 und den Rest der Division in das Register R3 schreiben.

Übung 14.7 Schnelle Multiplikation, mittel

Im Register R0 steht die Zahl n und im Register R1 die Zahl m. Geben Sie ein RAM-Programm für die schnelle Multiplikation von n und m an.

Übung 14.8 RAM-Logarithmus Berechnung, mittel

Im Register R0 steht eine Zahl n. Geben Sie ein RAM-Programm an, das den Wert $\lceil \log n \rceil$ berechnet und in R1 schreibt.

Übung 14.9 Indirekte Adressierung ist nicht nötig, schwer

Zeigen Sie: *Zu jedem RAM-Programm mit indirekter Adressierung gibt es ein RAM-Programm ohne indirekte Adressierung, das dieselbe Funktion berechnet.*

Tipps: Die Grundidee ist, die Inhalte der Register eines RAM-Programms mit indirekter Adressierung mit Hilfe einer geeigneten Kodierung in *ein* Register zu stopfen. Dann werden die Rechenschritte des RAM-Programms mit indirekter Adressierung durch ein geschickt konstruiertes RAM-Programm simuliert.

Wir gehen folgendermaßen vor: Sei Π ein RAM-Programm mit indirekter Adressierung, welches die Register R0, R1, ... verwendet. Wir kodieren die Zahlen aus diesen Registern, die ab sofort die *simulierten Register* nennen, in einem Binärwort und speichern dieses in einem (echten) Register. In diesem Binärwort sind »am Ende« (bei den so genannten *least significant bits*) die Bits von $\langle \text{R0} \rangle$, davor die von $\langle \text{R1} \rangle$, davor die von $\langle \text{R2} \rangle$ und so weiter.

Um die einzelnen Zahlen voneinander unterscheiden zu können, führen wir das Sonderzeichen # ein, welches jeweils zwei Zahlen trennt. Da die Kodierung binär ist, müssen wir uns eine geeignete Kodierungsfunktion für die Binärzahlen aus den Registern und das #-Symbol überlegen. Wir wählen als Kodierung $c\colon \{0,1\#\} \to \{0,1\}^2$ mit: $c(0) = 10$, $c(1) = 11$ und $c(\#) = 00$. Sind beispielsweise die simulierten Register R0, R1, R2 belegt mit $\langle \text{R0} \rangle = 6 = 110_2$, $\langle \text{R1} \rangle = 2 = 10_2$ und $\langle \text{R2} \rangle = 3 = 11_2$, so wäre $\hat{c}(11\#10\#110) = (111100111000111110)$.

Sie müssen nun zu zeigen, wie ein äquivalentes RAM-Programm aussehen würde, welches mit dem neuen Register arbeitet und damit keine indirekte Adressierung mehr verwendet. Zeigen Sie dazu nacheinander die folgenden Teilaussagen:

1. Geben Sie ein RAM-Programm ODD an, welches entscheidet, ob der Inhalts von R1 ungerade ist.
2. Geben Sie ein RAM-Programm BITTEST an, welches entscheidet, ob das i-te Bit des Inhalts von R2 eine 1 ist. Hier, und im Folgenden, steht i in R1.
3. Geben Sie ein RAM-Programm BITSET an, das i-te Bit des Inhalts von R2 auf den Wert in R3 setzt (der 0 oder 1 sein muss).
4. Geben Sie ein RAM-Programm BITINSERT an, welches an der Stelle i in R2 ein (Null)Bit *einfügt*. Aus 11001_2 würde beispielsweise nach Einfügen an Stelle 4 der String 101001_2.
5. Geben Sie ein RAM-Programm BITDELETE an, welches das i-te Bit aus der in R2 gespeicherten Zahl löscht.
6. Sei $\langle \text{R2} \rangle = \hat{c}(w)$, das Register 2 speichert also einen kodierten String. Geben Sie ein RAM-Programm #-SEARCH an, welches das i-te Vorkommen des #-Symbols von hinten in w berechnet.
7. Geben Sie ein RAM-Programm an, welches den Inhalt des i-ten simulierten Registers aus der in R2 gespeicherten Kodierung herausliest.
8. Zeigen Sie, dass Sie mit Hilfe dieser RAM-Programme jeden Befehl eines RAM-Programms mit indirekter Adressierung simulieren können. Es genügt, dies grob zu beschreiben.

15-1

Kapitel 15
Programme I: LOOP- und WHILE-Programme

Wenn Theoretiker Programmiersprachen entwerfen

Lernziele dieses Kapitels

1. Syntax und Semantik von LOOP- und WHILE-Programmen verstehen
2. Äquivalenzbeweis zu Turingmaschinen verstehen

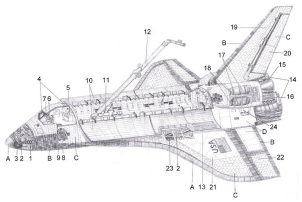
15-2

Inhalte dieses Kapitels

Worum es heute geht

Vielleicht haben Sie sich schon gewundert, weshalb in einer Vorlesung zur *Theoretischen* Informatik ausgerechnet den *Hardware-Modellen* in den letzten Kapiteln so ein breiter Raum gegeben wurde. (Andererseits – vielleicht wundert Sie bei dieser Vorlesung gar nichts mehr.) Die *Software*, die schließlich bei der täglichen Arbeit der meisten Informatikerinnen und Informatiker von zentraler Bedeutung ist, wurde bis jetzt weitgehend außen vor gelassen mit Ausnahme einiger etwas konstruiert wirkender Übungsaufgaben.

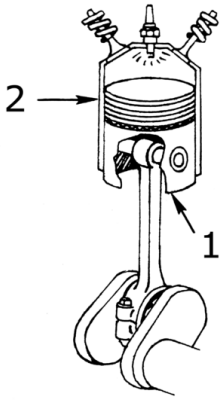
Es ist also höchste Zeit, sich der Software aus theoretischer Sicht anzunehmen. Dazu werden wir in diesem und dem nächsten Kapitel verschiedene Programmiersprachen betrachten, einmal aus dem Bereich der strukturierten-imperativen Programmierung und einmal aus dem Bereich der funktionalen Programmierung. Die logische und die objektorientierte Programmierung (und Hunde) müssen leider draußen bleiben – hier sei auf die speziellen Vorlesungen wie »Logikprogrammierung« verwiesen.



Unknown autor, Creative Commons Attribution ShareAlike Lizenz

Die im Folgenden untersuchten Programmiersprachen sind natürlich einfach gehalten. Will man erstmal prinzipiell verstehen, was verschiedene Programmiersprachen können oder nicht können, so sollte man nicht gleich mit Java in seiner ganzen API-Pracht beginnen. Eine Vorlesung über Maschinenbau beginnt ja auch nicht mit der Funktionsweise eines Space-Shuttles sonder lieber mit der eines Kolbens.

In diesem Kapitel werden zwei Sprachen eingeführt: Die Sprache »Loop« und die Sprache »While«. Beide sind sehr einfach und es stellt sich heraus, dass Loop *zu* einfach ist – es gibt einige Funktionen, die nicht »Loop-berechenbar« sind, obwohl man sie eigentlich recht einfach ausrechnen kann.



Autor Pearson Scott Foresman, public domain

Die Programmiersprache While ist fast identisch zu Loop, es gibt nur eine minimale Erweiterung: Die While-Schleife. Diese Erweiterung hat es aber in sich: Wir werden zeigen, dass While-Programme genauso mächtig sind wie RAM-Programme, von denen wir wiederum schon wissen, dass sie alle Turing-Maschinen simulieren können.

15.1 Maschinen versus Programmiersprachen

15-4

Worum es geht.

- In den vorherigen Kapiteln haben wir uns viel damit beschäftigt, *was Maschinen können*.
- Nun wollen wir analysieren, *was Programmiersprachen können*.
- Da Programmiersprachen in Maschinensprache übersetzt werden, *können Programmiersprachen sicherlich nicht mehr als Maschinen*.
- Die Frage ist also eher, *ob und in wie weit* gängig Programmiersprachen *genauso mächtig sind wie Turing-Maschinen oder* RAMs.

15-5

Zur Mächtigkeit der verschiedenen Programmierparadigmen.

- In diesem Kapitel geht es um zwei *sehr einfache imperative Sprachen*.
- In Kapitel 16 geht es um *sehr einfache funktionale Sprachen*.
- Wir werden keine Vertreter objektorientierter Sprachen untersuchen – da diese aber in der Regel auf imperativen aufbauen, können sie wenigstens so viel wie die in diesem Kapitel betrachteten Sprachen.
- Wir werden auch keine logischen Sprachen wie Prolog betrachten – die Mächtigkeit dieser Sprachen ist kompliziert (und spannend), es gibt dazu eigene Vorlesungen.

15.2 Zwei Programmiersprachen

15.2.1 Syntax von Loop

15-6

Überblick zur Programmiersprache »Loop«.

»Loop« ist eine *eingeschränkte imperative Sprache*.

Aufbau eines Loop-Programms

Ein Loop-Programm ähnelt zunächst etwas einem Java- oder C-Programm:

- Jedes Loop-Programm beginnt mit einer *Signatur*, bestehend aus einem Namen und Parametern.
- Es gibt nur einen Datentyp: `uint` (unsigned integers = natürliche Zahlen).
- Der *Körper* des Programms ist dann wie folgt aufgebaut:
  1. Er beginnt mit der Deklaration *lokaler Variablen*.
  2. Dann folgt ein Teil, in dem es Schleifen gibt (daher der Name).
  3. Er endet mit einem Return-Befehl.

15-7

Hello-World in »Loop«.

Da Loop keine Ausgabe kennt, hier eine Variante von »Hello World«, bei der einfach die Eingabe zurückgegeben wird:

```
uint echo (uint x)
{
  return x;
}
```

Ein komplexeres Beispiel, bei dem die Ausgabe die dreifache Eingabe ist.

```
uint tripple (uint x)
{
  uint y;
  loop (x) {
    y++;
    y++;
    y++;
  }
  return y;
}
```

Dabei bedeutet `loop (x)`, dass der Körper des Loop-Befehls *genau x Mal ausgeführt wird*. Im Körper kann man x nicht benutzen oder ändern.

Die Syntax von »Loop«. 15-8
Bezeichner.

▶ Definition: Bezeichner
Die *Menge der gültigen Bezeichner* in Loop sind Worte über dem Alphabet $\Sigma = \{\texttt{a}, \ldots, \texttt{z}, \texttt{A}, \ldots, \texttt{Z}, \_, \texttt{0}, \ldots, \texttt{9}\}$. Dabei müssen folgende Regeln eingehalten werden:

- Ein Bezeichner muss mit einem Buchstaben oder dem Unterstrich beginnen.
- Die Worte **`uint`**, **`loop`**, **`return`** und **`while`** sind Schlüsselworte und als Bezeichner nicht zugelassen.

Man überlegt sich leicht, dass die Menge der Bezeichner von Loop regulär ist.

Die Syntax von »Loop«. 15-9
Variablen.

▶ Definition: Variablen
Eine *Variable* in Loop wird durch einen Bezeichner benannt. Es gibt zwei Arten von Variablen:

Hilfsvariable Hilfsvariablen werden am Anfang des Programms deklariert mittels:

`uint` *Variablenname*`;`

Parameter Parameter werden im Kopf des Programms deklariert mittels

`uint` *Parametername*

Gibt es mehrere Parameter, so werden sie durch Kommata getrennt.

Die Syntax von »Loop«. 15-10
Zuweisungen und Inkrements

▶ Definition: Zuweisungen
In Loop gibt es zwei Arten von *Zuweisungen*:

Variable `=` `0;`

$Variable_1$ `=` $Variable_2$`;`

▶ Definition: Inkrements
In Loop gibt es zwei Arten von *Inkrements/Dekrements*:

Variable`++;` und *Variable*`--;`

Die Syntax von »Loop«. 15-11
Loop-Schleifen

▶ Definition: Loop
Ein *Loop* hat folgende Form:

`loop` `(`*Variable*`) {`
Body
`}`

- Dabei enthält der *Body* lediglich Zuweisungen und Loop-Schleifen.
- Weiterhin darf *Variable* nicht im Körper vorkommen.

Die Syntax von »Loop«. 15-12
Signatur

▶ Definition: Signatur
Die *Signatur* eines Loop-Programms hat folgende Form:

`uint` *Programmname* `(`**`uint`** $Parameter_1$`,`
`uint` $Parameter_2$`,...,`**`uint`** $Parameter_n$`)`

15-13

Die Syntax von »Loop«.

Loop-Programme

▶ **Definition:** Loop-Programm

Ein *Loop-Programm* ist ein Wort über dem ASCII-Alphabet, das wie folgt ausgebaut ist:

1. Es beginnt mit einer Signatur.
2. Dann folgt eine öffnende geschweifte Klammer.
3. Dann folgen eine beliebige Anzahl Hilfsvariablen-Deklarationen.
4. Dann folgen eine beliebige Anzahl an Zuweisungen und Loop-Schleifen.
5. Dann folgt **`return`** *Variable*`;`
6. Es endet mit einer geschweiften Klammer.

Auf **`uint`** und **`return`** muss jeweils ein Leerzeichen folgen, ansonsten sind Leerzeichen und Zeilenumbrüche egal.

Bemerkungen:

- Alle Variablen und Parameter müssen unterschiedlich benannt werden.
- Hilfsvariablen dürfen nur am Anfang des Programms deklariert werden, nicht innerhalb von Loop-Schleifen.
- Alle im Programm benutzen Variablen müssen deklariert worden sein (als Hilfsvariablen oder Parameter).
- Unterprogramme sind nicht erlaubt.

15-14

Beispiele von Loop-Programmen.

Die Multiplikation.

```
uint multiply (uint x, uint y)
{
  uint result;
  loop (x) {
    loop (y) {
      result++;
    }
  }
  return result;
}
```

15-15

Beispiele von Loop-Programmen.

Potenzen berechnen.

```
uint power (uint x, uint y)
{
  uint result;
  uint temp;
  result++;
  loop (y) {
    temp = 0;
    loop (x) {
      loop (result) {
        temp++;
      }
    }
    result = temp;
  }
  return result;
}
```

15.2.2 Syntax von While

Die Syntax von »While«. 15-16

▶ Definition: While-Programm
Ein *While-Programm* ist genauso aufgebaut wie ein Loop-Programm, nur dass es neben `loop` noch folgende Steuerungsanweisung gibt:

```
while (Variable != 0){
  Body
}
```

While- und Loop-Anweisungen können beliebig verschachtelt werden. Im *Body* darf *Variable* vorkommen.

Beispiel eines While-Programms. 15-17

```
uint foo (uint n)
{
  uint result;
  while (n != 0) {
    loop (n) {
      result++;
    }
    n--;
  }
  return result;
}
```

✎ Zur Diskussion
Was gibt `foo` zurück?

15.2.3 Semantik von Loop

Die Semantik von Loop-Programmen. 15-18

Ideen

- Loop-Programme nehmen Tupel von natürlichen Zahlen als Eingabe(Parameter) und liefern eine Zahl als Ausgabe. Das formalisiert man sinnigerweise als Funktion $f\colon \mathbb{N}^n \to \mathbb{N}$, wobei n eben die Anzahl an Parametern ist.
- Das Programm wird einfach abgearbeitet. Die Anfangswerte der Variablen lauten:
  - Bei Eingabe $(x_1,\ldots,x_n)$ hat der i-te Parameter gerade den Wert x_i.
  - Alle Hilfsvariablen sind 0.
- Alle Zuweisungen und Inkrements haben den erwarteten Effekt, ein Dekrement auf 0 angewandt ergibt wieder 0.
- Der zurückgegebene Wert ist gerade der Funktionswert.

▶ Definition: Loop-berechenbare Funktionen
Eine Funktion $f\colon \mathbb{N}^n \to \mathbb{N}$ heißt *Loop-berechenbar*, wenn es ein Loop-Programm gibt, das f berechnet.

Beispiele 15-19
Die Multiplikation

Beispiel: Multiplikation
Die Funktion $f(x,y) = x \cdot y$ ist Loop-berechenbar durch folgendes Programm:

```
uint multiply (uint x, uint y)
{
  uint result;
  loop (x) {
    loop (y) {
```

```
      result++;
    }
  }
  return result;
}
```

15-20

Beispiele

Die Signum-Funktion

Beispiel: Signum-Funktion

Die Signum-Funktion

$$\operatorname{sgn}(x) = \begin{cases} 1, & \text{falls } x > 0, \\ 0 & \text{sonst.} \end{cases}$$

ist Loop-berechenbar:

```
uint sign (uint x)
{
  uint result;
  result = x;
  x--;
  loop (x) {
    result--;
  }
  return result;
}
```

Skript-Referenz

▶ **Definition:** Semantik eines Loop-Programms

Sei P ein Loop-Programm mit n Parametern. Sei V die Menge aller in P vorkommenden Variablen. Eine *Konfiguration* ist eine Abbildung $C\colon V \to \mathbb{N}$. Ist $v \in V$ eine Variable und C eine Konfiguration, so schreiben wir $\langle v \rangle_C$ oder einfach $\langle v \rangle$ für den Wert von v unter der Abbildung C (also für $C(v)$). Den Index C lassen wir in der Regel auch weg.

Wir definieren nun induktiv für ASCII-Worte w, die gerade Folgen von Zuweisungen und (eventuell verschachtelten) Loop-Schleifen enthalten, welche Konfiguration C' herauskommt, wenn man sie in Konfigurationen C beginnend ausführt. Wir schreiben dann $C \vdash^w C'$.

1. Ist w der Befehl `v = 0;`, so ist C' identisch zu C, außer dass $\langle v \rangle' = 0$ gilt.
   Mit anderen Worten: In C' wird v auf 0 gesetzt.
2. Ist w der Befehl `v = u;`, so ist C' identisch zu C, außer dass $\langle v \rangle' = \langle u \rangle$ gilt.
   Mit anderen Worten: In C' hat v den Wert, den u vorher hatte.
3. Ist w der Befehl `v++;`, so ist C' identisch zu C, außer dass $\langle v \rangle' = \langle v \rangle + 1$ gilt. Ist w der Befehl `v--;`, so ist hingegen $\langle v \rangle' = \max\{\langle v \rangle - 1, 0\}$.
4. Sei w der Befehl `loop (v) {b}`, wobei das ASCII-Wort b der Körper der Schleife ist. Dann haben wir schon induktiv definiert, für welche C_1 und C_2 gilt $C_1 \vdash^b C_2$. Seien nun $C_0 = C$ und $C_{\langle v \rangle} = C'$ und C_i Konfigurationen mit $C_{i-1} \vdash^b C_i$ für $i \in \{1, \ldots, \langle v \rangle\}$. Dann gilt $C \vdash^w C'$.
   Mit anderen Worten: Führt man den Körper b genau $\langle v \rangle$ mal aus und durchläuft dabei die Konfigurationen C_0 bis $C_{\langle v \rangle}$, so endet man gerade bei der letzten Konfiguration.
5. Ist $w = u \circ v$, wobei u und v selber Folgen von Zuweisungen und Loop-Schleifen sind, und gilt $C \vdash^u C''$ und $C'' \vdash^v C'$, so gilt auch $C \vdash^w C'$.
   Mit anderen Worten: Kommt nach einem Befehl ein anderer, so beginnt mit der zweite mit dem Ergebnis, das der erste produziert hat.

Die *Anfangskonfiguration* $C_{\text{init}}(x_1, \ldots, x_n)$ eines Programms bei Eingabe $(x_1, \ldots, x_n)$ belegt die Parameter-Variablen mit den Werten der x_i und setzt $\langle v \rangle = 0$ für alle Hilfsvariablen v.

Sei nun P ein Loop-Programm und $f\colon \mathbb{N}^n \to \mathbb{N}$ eine Funktion, wobei n die Anzahl der Parameter von P ist. Sei k das ASCII-Wort, das die Folge der Zuweisungen und Loop-Schleifen von P enthält (also alles außer der Signatur, den Variablendeklarationen und dem Return-Befehl). Sei r die Variable im Return-Statement. Wir sagen, dass P *die Funktion f berechnet*, wenn für alle $(x_1, \ldots, x_n) \in \mathbb{N}^n$ gilt: $C_{\text{init}}(x_1, \ldots, x_n) \vdash^k C$ mit $\langle r \rangle_C = f(x_1, \ldots, x_n)$.

15.2.4 Semantik von While

Die Semantik von While-Programmen.

15-21

Ideen

- Die Semantik ist sehr ähnlich zu Loop-Programmen definiert.
- Neu ist aber, dass While-Programme *nicht immer anhalten*.
- Folglich ist die Semantik, *analog zu* RAM-*Programmen,* eine *partielle Funktion* f: $\mathbb{N}^n \dashrightarrow \mathbb{N}$.

▶ **Definition:** While-berechenbare Funktionen

Eine partielle Funktion $f\colon \mathbb{N}^n \dashrightarrow \mathbb{N}$ heißt *While-berechenbar*, wenn es ein While-Programm gibt, das f berechnet. Das Programm hält genau dann nicht an auf einer Eingabe, wenn f an dieser Stelle undefiniert ist.

Skript-
Referenz

▶ **Definition:** Semantik eines While-Programms

Wir erweitern die Semantik von Loop-Programmen durch folgende Regel:

- Sei w der Befehl `while (v != 0){b}`. Wieder haben wir schon induktiv definiert, für welche C_1 und C_2 gilt $C_1 \vdash^b C_2$. Seien nun $C_0 = C$ und C_i Konfigurationen mit $C_{i-1} \vdash^b C_i$ für $i \in \{1,\dots,n\}$. Weiter sei
  1. $\langle v \rangle_{C_i} \neq 0$ für alle $i \in \{1,\dots,n-1\}$ und
  2. $\langle v \rangle_{C_n} = 0$.

  Dann gilt $C \vdash^w C'$.
  Mit anderen Worten: Führt man den Körper b immer wieder aus und durchläuft dabei die Konfigurationen C_0, C_1, C_3 und so weiter, so endet man bei der ersten Konfiguration, bei der $\langle v \rangle = 0$ gilt.

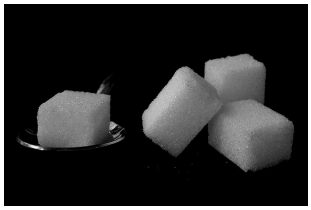
Man beachte: Es kann bei einer While-Schleife passieren, dass $C \vdash^w C'$ für überhaupt kein C' gilt, nämlich genau dann, wenn eine Endlosschleife durchlaufen wird.
Wir sagen nun, dass ein While-Programm P mit Körper k *die Funktion* f *berechnet*, wenn für alle $(x_1,\dots,x_n) \in \mathbb{N}^n$ gilt:

1. Ist $f(x_1,\dots,x_n)$ definiert, so existiert ein C mit $C_{\text{init}}(x_1,\dots,x_n) \vdash^k C$ und $\langle r \rangle_C = f(x_1,\dots,x_n)$.
2. Ist $f(x_1,\dots,x_n)$ undefiniert, so existiert kein C mit $C_{\text{init}}(x_1,\dots,x_n) \vdash^k C$.

15.2.5 Syntactic Sugar

15-22

Einfacher Syntactic-Sugar.



Public domain

- Die Syntax von Loop und While ist etwas »minimalistisch«.
- Es wäre schön, wenn man beispielsweise auch schreiben könnte `x = 5;` oder `x = y + z;`
- Solche *syntaktischen Vereinfachungen* nennt man auch *syntactic sugar* – sie versüßen einem das Leben, tragen aber nichts zur Mächtigkeit bei.

Folgender denkbarer Syntactic-Sugar ist einfach einzubauen:

- Direkte Zuweisung von Konstanten: `x = 5;`
- Direkte Addition von Variablen: `x = y+z;`
- Auswerten arithmetischer Ausdrücke: `x = (3+y)*z;`
- Komplexere While-Prädikate: `while (x > 5)`

15-23

Süßerer Syntactic-Sugar.

- Eine zentrale Steuerungsanweisung fehlt uns noch: Das If-Statement!
- Wir möchten doch gerne schreiben können

```
if (v != 0){
  IfPart
} else {
  ElsePart
}
```

- Will man dies als Loop-Programm implementieren, braucht man einen Trick.

Das If-Statement in Loop

Man kann ein If-Statement wie folgt simulieren:

1. Berechne zunächst $t_1 = \mathrm{sgn}(v)$, siehe Folie 15-20, in einer temporären Variable t_1.
2. Berechne dann $t_2 = 1 - t_1$ in einer zweiten temporären Variable.
3. Führe dann aus:

```
loop (t1){
  IfPart
}
loop (t2){
  ElsePart
}
```

15.3 Äquivalenzsatz

15-24

Was können die Sprachen Loop und While?

- Man kann sich nun fragen, wie *mächtig* die Sprachen Loop und While sind.
- Können sie beispielsweise *beide dasselbe*? (Braucht man also überhaupt While-Schleifen?)
- Können sie *so viel wie* RAMs?

Ergebnisse

- Man kann zeigen, dass man mit Loop-Programmen *echt weniger* berechnen kann als While-Programme.
- Wir zeigen gleich, dass *While-Programme genau so viel können, wie* RAMs.

Äquivalenzsatz für While- und RAM-Berechenbarkeit

15-25

▶ Satz

Sei $f\colon \mathbb{N}^n \dashrightarrow \mathbb{N}$. *Dann sind folgende Aussagen äquivalent:*

1. f *ist While-berechenbar.*
2. f *ist* RAM*-berechenbar.*

Beweisskizze. Für die erste Richtung sei P ein While-Programm. Wir müssen eine RAM konstruieren, die P simuliert. Dies geht sehr einfach:

- Wir speichern die Werte der Variablen in den Registern.
- Einfache Zuweisungen und Inkrements sind ganz einfach direkt zu simulieren.
- Eine While-Schleife ist auch ganz einfach mit einem geeigneten Sprung ***if*** $\mathrm{R}i > 0$ ***goto*** k zu simulieren.
- Für jedes Loop braucht man ein zusätzliches Register, in dem bis zu 0 runtergezählt wird.

Für die zweite Richtung sei Π ein RAM-Programm. Wir müssen ein While-Programm konstruieren, das Π simuliert.

- Wir dürfen davon ausgehen, dass Π *keine indirekte Adressierung benutzt*, siehe Übung 14.9.
- Für jedes der endlich vielen benutzten Register gibt es nun eine Variable.
- Eine Variable speichert noch den *Programmzähler*.
- Das Programm besteht dann aus einer *großen While-Schleife*, in der Folgendes passiert:
  1. In einer langen Folge von If-Anweisungen finde den Befehl von Π, der zum aktuellen Programm-Zähler gehört.
  2. Simuliere den Befehl (mit einem kleinen passenden Loop-Programm) und aktualisiere den Programmzähler.
- Die Schleife wird beendet, wenn der Programmzähler auf dem ***stop***-Befehl steht. □

Zusammenfassung dieses Kapitels

15-26

1. Loop-Programme bestehen aus einfachen Zuweisungen und Schleifen, bei denen eine Variable *vorher* angibt, wie oft sie durchlaufen werden.
   Sie produzieren immer ein Ergebnis.
2. While-Programme erlauben *zusätzlich* auch While-Schleifen.
   Sie *produzieren nicht immer Ausgaben*, nämlich dann nicht, wenn sie in Endlosschleifen gehen.
3. Eine Funktion ist genau dann While-berechenbar, wenn sie RAM-berechenbar ist.

Zum Weiterlesen

[1] R. Reischuk. *Unterlagen zum Lehrmodul Theoretische Informatik,* Vorlesungsskript, 2008. Kapitel 5.1.

Übungen zu diesem Kapitel

Übung 15.1 Grammatik für die Sprachen Loop und While, leicht

Geben Sie für die Syntax der Programmiersprachen Loop und While jeweils kontextfreie Grammatiken an, zusammen mit natürlichsprachlichen Einschränken an die Menge der erlaubten Programme. Die »natürlichsprachlichen Einschränkungen« sind notwendig, da man mit kontextfreien Grammatiken ja nicht ausdrücken kann, dass nur deklarierte Variablen auch wirklich verwendet werden dürfen. Ebenso sind diese sehr praktisch, um auszuschließen, dass Schlüsselworte als Bezeichner verwendet werden.

16-1

Kapitel 16
Programme II: Rekursion
Von der Kunst, Probleme zu delegieren

16-2

Lernziele dieses Kapitels

1. Syntax und Semantik den primitiven und der μ-Rekursion verstehen
2. Kleenesche Normalform kennen
3. Äquivalenzbeweis zu LOOP- und WHILE-Programmen verstehen

Inhalte dieses Kapitels

Worum es heute geht

Kaum etwas wird unter Informatik-Studenten und Informatik-Professoren mit mehr Elan diskutiert als die Frage, ob man im ersten Semester eines Informatik-Studiums lieber mit *funktionaler* oder doch lieber mit *imperativer* Programmierung beginnen sollte. Sollten Sie sich an solcher einer Debatten beteiligen wollen, hier ein paar Argumente für beide Seiten:

Warum man mit funktionaler Programmierung beginnen sollte

- Funktionale Sprache sind eleganter, es gibt weniger Sonderregeln und Sonderfälle.
- Funktionale Sprachen zeigen Grundideen wie die Rekursion viel klarer auf, was für Einsteiger sehr nützlich ist.
- Alle Studierende fangen auf einem ähnlichen Wissensstand an.
- Es gibt keine Zeiger, wodurch eine große Verständnisschwierigkeit wegfällt.
- Funktionale Sprachen lassen sich *prinzipiell* viel besser optimieren.

Für die imperativen Sprachen sprechen im Wesentlichen drei Gründe:

Warum man mit imperativer Programmierung beginnen sollte

- Im Alltag der Software-Entwicklung werden imperative Sprachen eingesetzt.
- Im Alltag der Software-Entwicklung werden imperative Sprachen eingesetzt.
- Im Alltag der Software-Entwicklung werden imperative Sprachen eingesetzt.

In diesem Kapitel werden nun gleich zwei funktionale Sprachen eingeführt mit den schönen Namen PrimitiveML und MüML (das »ML« steht für die Programmiersprache gleichen Namens, von der die Syntax dieser Sprachen geborgt wurde). Ähnlich wie im vorherigen Kapitel sind dies keine »ausgebauten« funktionalen Programmiersprachen; wieder geht es nur darum, einen möglichst minimalistischen Kern zu definieren, auf dem man dann jede Menge Syntactic-Sugar streuen kann, um daraus eine vernünftige Programmiersprache zu machen.

Eine der wesentlichen Eigenschaften einer funktionalen Sprache ist, dass sie *Rekursion* unterstützt. Genaugenommen wird in der funktionalen Programmierung Rekursion für einfach *alles* genutzt, selbst da, wo es nun wirklich keinen Sinn ergibt. Dies mag einer der Hauptgründe sein, weshalb manche Leute allergisch auf solche Sprachen reagieren. Die beiden Sprachen PrimitiveML und MüML unterscheiden sich lediglich dadurch, welche *Arten* von Rekursion erlaubt sind.

Zwei imperative Sprachen, zwei funktionale Sprachen, zwei Kapitel. In der *Special Theory Edition* von *Matrix Reloaded* kommentiert Morpheus dies wie folgt: »I do not believe in chance when I see two chapters, two imperative language, two funtional languages. I do not see coincidence, I see providence, I see purpose. I believe it is our fate to be here. It is our destiny. I believe this chapter holds for each and every one of us the very meaning of our lives.« Wie wir sehen werden, hat er mit *»I see providence«* recht: Die imperative Sprache Loop ist genau gleichmächtig zur funktionalen Sprache PrimitiveML, die Sprache While hingegen zu MüML. Ob allerdings seinem letzten Satz zuzustimmen ist, bleibt abzuwarten.

16.1 Imperative versus funktionale Programmierung

Zur Historie der funktionalen Programmierung

16-4

- Die ersten funktionalen Sprachen gab es bereits, *bevor es Computer gab*.
- Insbesondere wurde Churchs λ*-Kalkül* bereits untersucht, bevor Turing seine Turing-Maschine vorstellte.
  (Der Streit, ob nun funktional oder imperativ besser ist, ist also eigentlich so alt wie die Informatik selbst.)
- Die erste funktionale Sprache, die größere Verbreitung erlangte, war Lisp.
- Später folgten dann ML und Haskell.

Die wesentlichen Ideen hinter funktionalen Programmen.

16-5

- Programme bilden Eingaben auf Ausgaben ab – *semantisch* sind sie also Funktionen.
- Es liegt deshalb nahe, der *Verkettung und Verarbeitung* von Funktionen besondere Aufmerksamkeit zu schenken.
- Es zeigt sich (und dies ist ein wesentliches Resultat dieses Kapitels), dass man *überhaupt nur die Verkettung von Funktionen* und *Rekursion* braucht.
- Eine (reine) funktionale Sprache erlaubt deshalb nur Folgendes:
  1. Neue Funktionen als die Verkettung bekannter Funktionen zu definieren und
  2. neue Funktionen mittels Rekursion zu definieren.
- Da die Rekursion eine so zentrale Rolle spielt, heißt die Untersuchung der Mächtigkeit funktionaler Sprachen »Rekursionstheorie«.

16.2 Primitive Rekursion

16.2.1 Die Idee

Die Ideen hinter PrimitiveML

16-6

Die drei Grundideen

1. Es gibt nur einen Datentyp: *natürliche Zahlen*. (Wie überraschend…)
2. In einem *Programm* werden *Funktionen definiert*, die *Tupel von natürlichen Zahlen* auf *natürliche Zahlen* abbilden.
3. Man definiert eine Funktion auf zwei Arten:
   3.1 Die Anwendung verschiedener, bereits definierter Funktionen auf die Eingabevariablen.
   3.2 Die Anwendung des *primitiven Rekursionsschemas*.

Das Henne-Ei-Problem

Um eine neue Funktion zu definieren, braucht man immer schon bestehende. Wie fängt man also überhaupt an?
Am Anfang sind immer schon zwei Funktionen definiert:

- Die Funktion `0`, die einfach die Konstante 0 liefert.
- Die Nachfolgerfunktion **`succ`**.

16-7

Die primitive Rekursion.

- Die wesentliche Idee einer Rekursion ist, ein Problem für einen Wert x zu lösen, indem man *das Problem zunächst für* $x-1$ *löst*.
- Das führt dazu, dass man dann das Problem zunächst für $x-2$ lösen muss, und so weiter.
- Abbrechen muss man die Berechnung dann bei $x=0$, wo man die Lösung »direkt angibt«.

Idee hinter der primitiven Rekursion

Das *primitive Rekursionsschema* definiert eine Funktion $f\colon \mathbb{N}^{n+1} \to \mathbb{N}$ wie folgt:

1. Man gibt an, wie $f(x_1,\ldots,x_n,0)$ lautet.
2. Man gibt an, wie man $f(x_1,\ldots,x_n,y+1)$ berechnen kann, wenn man $f(x_1,\ldots,x_n,y)$ kennt.

16.2.2 Syntax von PrimitiveML

16-8

Syntax von PrimitiveML

Variablen und Funktionsnamen

▶ **Definition:** Bezeichner und Schlüsselworte
Die *Bezeichner* von PrimitiveML sind genauso definiert wie in Loop und While. *Schlüsselworte* sind diesmal **`succ`**, **`fun`** und **`min`**.

▶ **Definition:** Variablennamen und Funktionsnamen
Variablennamen und *Funktionsnamen* sind Bezeichner. Jeder Funktionsname hat eine *Stelligkeit* $n \in \mathbb{N}$. Eine Funktion der Stelligkeit 0 nennt man auch eine *Konstante*.

Erläuterungen:

- Auf der semantischen Seite steht ein Variablenname für einen festen Wert, ein Funktionsname der Stelligkeit n für eine Funktion $f\colon \mathbb{N}^n \to \mathbb{N}$.
- Anders als imperativen Sprachen ändern Variablen ihre Werte nicht. Sie sind eher mit den Variablen aus der Mathematik zu vergleichen.

16-9

Syntax von PrimitiveML

Terme

▶ **Definition:** Term
Ist V eine Menge von Variablenamen und F eine Menge von Funktionsnamen, so ist die Menge der *Terme* induktiv wie folgt definiert:

1. `0` ist ein Term.
2. Jede Variable in V ist ein Term.
3. Jede Konstante in F ist ein Term.
4. Ist $f \in F$ ein n-stelliger Funktionsname und sind t_1 bis t_n Terme, so ist $f(t_1,\ldots,t_n)$ ein Term.

Beispiel

Typische Terme sind

- `pi`
- `f(0,g(x,foo))`

Syntax von PrimitiveML
Nichtrekursive Funktionsdefinition

16-10

▶ Definition: Nichtrekursive Definition

Eine *nichtrekursive Definition* einer *n-stelligen Funktion* hat folgende Form:

fun *Funktionsname* $(v_1, \ldots, v_n)$ = *term*

Dabei gelten für den *term* folgende Einschränkungen:

- Als Variablen dürfen nur v_1 bis v_n benutzt werden.
- Als Funktionsnamen dürfen nur bereits in früheren Funktionsdefinitionen eingeführte Funktionsnamen benutzt werden.
- Der *Funktionsnamen* darf *nicht* im *term* auftauchen.

Für den Fall $n = 0$ entfallen die Klammern.

Beispiel

```
fun one = succ(0)

fun id (x) = x

fun plus2 (x) = succ(succ(x))
```

Syntax von PrimitiveML
Primitivrekursive Funktionsdefinition

16-11

▶ Definition: Primitivrekursive Definition

Eine *primitivrekursive Definition* einer $(n+1)$*-stelligen Funktion* hat folgende Form:

fun *Funktionsname* $(v_1, \ldots, v_n, 0)$ = *initial_term*
| *Funktionsname* $(v_1, \ldots, v_n,$ **succ** $(v_{n+1}))$ = *rekursions_term*

Dabei gelten für die Terme folgende Einschränkungen:

- Für den *initial_term* gelten dieselben Einschränkungen wie bei der nichtrekursiven Definition.
- Im *rekursions_term* darf der Term *Funktionsname* $(v_1, \ldots, v_n, v_{n+1})$ genutzt werden.

Beispiel

```
fun pred(0) = 0
  | pred(succ(x)) = x

fun add(x,0) = x
  | add(x,succ(y)) = succ(add(x,y))

fun sub(x,0) = y
  | sub(x,succ(y)) = pred(sub(x,y))

fun mul(x,0) = 0
  | mul(x,succ(y)) = add(x,mul(x,y))

fun if_help(x,y,0) = y
  | if_help(x,y,succ(z)) = x

fun if_then_else(x,y,z) = if_help(y,z,x)

fun sign(x) = sub(x,pred(x))
```

16-12

Syntax von PrimitiveML

Simultane Rekursion

Folgende Erweiterung der Definition ist oft nützlich, man kann aber zeigen, dass man sie eigentlich nicht braucht:

▶ Definition: Simultane primitivrekursive Definition

- Bei einer *simultanen primitivrekursiven Definition* schreibt man mehrere primitivrekursive Definition hintereinander, wobei beim Zusammenfassen **`fun`** durch `|` ersetzt wird.
- Dadurch werden gleichzeitig *mehrere* $(n+1)$*-stelligen Funktionen* definiert.
- Neu ist, dass in den Rekursionstermen jeweils auch die anderen Funktionen aufgerufen werden dürfen.

Beispiel

```
fun even (0)        = succ(0)
  | even (succ(x)) = odd(x)
  | odd  (0)        = 0
  | odd  (succ(x)) = even(x)
```

16-13

Syntax von PrimitiveML

▶ Definition: PrimtiveML-Programm

Ein *PrimitiveML-Programm* ist ein Wort über dem ASCII-Alphabet, das aus einer Folge von Funktionsdefinitionen besteht. Dabei sind folgende Regeln einzuhalten:

- Die Funktion **`succ`** darf in jeder Funktionsdefinition benutzt werden.
- Ansonsten dürfen in einer Funktionsdefinition nur bereits vorher definierte Funktionen benutzt werden (außer bei der primitiven Rekursion gemäß den dort gültigen Regeln).
- Leerzeichen und Zeilenumbrüche sind, außer nach dem Schlüsselwort **`fun`**, optional.

16-14

✎ Zur Übung

Ergänzen Sie folgendes Programm um die Definition einer Funktion `fac` zur Berechnung der Fakultät:

```
fun add(x,0)        = x
  | add(x,succ(y)) = succ(add(x,y))

fun mul(x,0)        = 0
  | mul(x,succ(y)) = add(x,mul(x,y))

fun fac ...?
```

16.2.3 Semantik von PrimitiveML

16-15

Die Semantik eines PrimitiveML-Programms

Die Idee

- Wie bei jeder Programmiersprache ist ein PrimitiveML-Programm zunächst erstmal ein *Wort* über dem ASCII-Alphabet.
- Die *Semantik der Programmiersprache* gibt nun an, was so ein Wort *bedeutet*.
- Es liegt nahe, dass wir ein PrimitiveML-Programm als *die Definition von Funktionen* interpretieren.
- Die von einem Programmtext definierten Funktionen sind dann genau so definiert, wie man es erwarten würde, beispielsweise definiert das Programm

  ```
  fun add(x,0)        = x
    | add(x,succ(y)) = succ(add(x,y))
  ```

  gerade die Funktion $add\colon \mathbb{N}^2 \to \mathbb{N}$ mit $add(x,y) = x+y$.
- Das muss man jetzt noch einigermaßen formal aufschreiben.

Die Semantik eines PrimitiveML-Programms

16-16

▶ Definition: Semantik eines PrimitiveML-Programms

Sei P ein PrimitiveML-Programm und seien f_1 bis f_n die Namen aus den Funktionsdefinition in der Reihenfolge ihres Auftretens. Wir definieren dann für jedes i eine Funktion $\varphi_i : \mathbb{N}^{n_i} \to \mathbb{N}$, wobei n_i gerade die Stelligkeit von f_i ist. Diese Funktionen heißen *die von P beschriebenen Funktionen*.

1. Ist f_i eine nichtrekursive Funktionsdefinition der Form

   `fun` $f_i(v_1, \ldots, v_n)$ = *term*

   so ist $\varphi_i(x_1, \ldots, x_n)$ mit $(x_1, \ldots, x_n) \in \mathbb{N}^n$ gerade die *Auswertung des Terms*, wobei jede Variable v_i gerade als x_i interpretiert wird und jeder Funktionsname f_j in *term* gerade als die Funktion φ_j.
   Siehe das Skript zu »Logik für Informatiker« zu Details, wie die Termauswertung definiert ist.
2. Ist f_i eine primitivrekursive Funktionsdefinition der Form

   `fun` $f_i(v_1, \ldots, v_n, 0)$ = *initial_term*
   | $f_i(v_1, \ldots, v_n,$ `succ` $(v_{n+1}))$ = *rekursions_term*

   so ist $\varphi_i(x_1, \ldots, x_n, 0)$ gerade die Auswertung von *initial_term*. Der Wert von $\varphi_i(x_1, \ldots, x_n, x+1)$ ergibt sich für jedes x rekursiv aus der Auswertung des *rekursions_term*, wobei alle Vorkommen von $f_i(v_1, \ldots, v_n, v_{n+1})$ durch den Wert von $\varphi_i(x_1, \ldots, x_n, x)$ ersetzt werden.

Die Klasse der primitivrekursiven Funktionen.

16-17

▶ Definition: Primitivrekursiven Funktionen

Eine Funktion $f : \mathbb{N}^n \to \mathbb{N}$ heißt *primitivrekursiv*, wenn sie von einem PrimitiveML-Programm P beschrieben wird.
Die Klasse aller primitivrekursiven Funktionen bildet die Klasse PR.

Beachte:

- Diese Klasse enthält *Funktionen* und *keine Sprachen*. Man nennt sie deshalb auch eine *Funktionenklasse*.
- Man kann Funktionenklassen und Sprachklassen *nicht direkt vergleichen*.
- Beispiele von Funktionen in PR sind:
  - Die Grundrechenarten,
  - Potenzierung,
  - der Primzahltest (also $f(p) = 1$, falls p prim ist, sonst $f(p) = 0$).
- Es ist nicht leicht, eine Funktion zu finden, die nicht primitivrekursiv ist.

16.2.4 Primitivrekursiv = Loop-berechenbar

Jede primitivrekursive Funktion ist Loop-berechenbar und umgekehrt.

16-18

▶ Satz

Sei $f : \mathbb{N}^n \to \mathbb{N}$. Dann sind folgende Aussagen äquivalent:

1. *f ist primitivrekursiv.*
2. *f ist Loop-berechenbar.*

Beweisideen

- Bei beiden Richtungen wird jeweils ein Programmtext in der einen Programmiersprache in einen Programmtext der anderen Programmiersprache übersetzt.
- Entscheidend ist, dass ein `loop`-Konstrukt gerade einer primitivrekursiven Funktionsdefinition entspricht.

Beweis. Wir müssen zwei Richtungen zeigen. Sei zunächst f primitivrekursiv.[1]
Sei P ein PrimitiveML-Programm, das (unter anderem) f beschreibt. Unser Ziel ist es, daraus ein ein Loop-Programm P' zu konstruieren, dass ebenfalls P berechnet.[2]
Die Idee ist, für jede Funktionsbeschreibung eines f_i ein Loop-Programm P_i' zu schreiben, das ebenfalls diese Funktion berechnet.

Kommentare zum Beweis

[1] Erste Richtung.

[2] Rezept »Konstruktiver Beweis«

Skript

1. Erster Fall: Eine nichtrekursive Funktionsdefinition der Form

   fun $f_i(v_1,\ldots,v_n)$ = *term*

   Wir dürfen ohne Beschränkung der Allgemeinheit annehmen, dass *term* lediglich einfach verschachtelt ist, anderenfalls können wir durch die Definition von neuen Funktionen für Teilterme dies erzwingen.
   Sei also *term* von der Form $f_i(t_1,\ldots,t_{i_n})$ und jedes t_j sei vor Form $f_{k_j}(v_j^1,\ldots,v_j^{n_{k_j}})$, wobei die v_j^l die passenden Variablen aus $\{v_1,\ldots,v_n\}$ sind. In t_j können nur Funktionsnamen f_{k_j} mit $k_j < i$ verwendet werden (da die Definition ja nichtrekursiv ist). Folglich kennen wir für jedes f_k bereits ein Programm P'_k, das f_k berechnet.
   Wir führen nun folgende Schreibweise ein: Wir schreiben $v \leftarrow P(v_1,\ldots,v_n)$ für den Programmtext eines Programms P, allerdings mit folgenden Änderungen:
   - Die Signatur und die Deklaration der Hilfsvariablen werden weggelassen.
   - Die Hilfsvariablen und Parameter werden umbenannt, so dass sie sonst nicht vorkommen.
   - Ist p_i der i-te Parameter von P, so stellen wir die Zeile p_i = v_i; voran.
   - Ist **return** x; letzte Zeile von P, so ersetzen wir diese durch v = x;

   Durch die Umbenennung eingeführte Hilfsvariablen werden natürlich deklariert.
   Das gewünschte Programm P'_i für f_i ergibt sich nun grob wie folgt:

```
uint f_i (uint v_1, ..., uint v_n)
{
  uint v_1^1; ... uint v_{i_n}^{n_{i_n}};

  uint t_1; ... uint t_{i_n};
  uint result;

  t_1 ← P'_{k_1}(v_1^1,...,v_1^{n_{k_1}})
  t_2 ← P'_{k_2}(v_2^1,...,v_2^{n_{k_2}})
  ⋮
  t_n ← P'_{k_n}(v_n^1,...,v_n^{n_{k_n}})
  result ← P'_i(t_1,...,t_n)

  return result;
}
```

2. Zweiter Fall: Eine primitivrekursive Funktionsdefinition der Form

   fun $f_i(v_1,\ldots,v_n,0)$ = *initial_term*
   | $f_i(v_1,\ldots,v_n,\mathbf{succ}(v_{n+1}))$ = *rekursions_term*

   Wir dürfen diesmal (wieder durch Einführung von Hilfsfunktionen) davon ausgehen, dass sowohl *initial_term* als auch *rekursions_term* beide einfache Funktionsanwendungen auf Variablen sind.
   Wir können sogar annehmen (wieder durch Einführung von Hilfsfunktionen), dass *initial_term* $= f_j(v_1,\ldots,v_n)$ gilt und *rekursions_term* $= f_k(v_1,\ldots,v_n,v_{n+1},f_i(v_1,\ldots,v_{n+1}))$.
   Dies wird in folgendes Programm verwandelt:

```
uint f_i (uint v_1, ..., uint v_n)
{
  uint i;
  uint previous;

  previous ← f_j(v_1,...,v_n)
  loop (v_n) {
    previous ← f_k(v_1,...,v_n,i,previous)
    i++;
  }

  return previous;
}
```

   Eine simultane Rekursion wird ähnlich dargestellt, es müssen nur mehrere `previous`-Variablen genutzt werden.

Die Korrektheit der Konstruktion zeigt man nun durch Induktion.[3]
Für die zweite Richtung sei P ein Loop-Programm. Wir müssen dieses in ein PrimitiveML-Programm P' umwandeln.[4] Für die Konstruktion sei w das ASCII-Wort, das die Folgen von Zuweisungen und Loop-Schleifen von P enthält (siehe Definition 15.2.3). Unser Ziel ist es, ein PrimitiveML-Programm zu konstruieren, das jede Konfigurationen C von P auf die Konfiguration C' mit $C \vdash^w C'$ abbildet.
Dabei ist zunächst problematisch, dass C' ja ein Tupel ist, PrimitiveML-Funktionen hingegen immer nur eine einzelne Zahl zurückgeben. Um dieses Problem zu lösen, konstruieren wir gleich mehrere Funktionen, nämlich eine Funktion für jede Komponente des Tupels, das eine Konfiguration ausmacht.

[3] Da hat es sich aber jemand sehr leicht gemacht mit der Korrektheit. Andererseits: Da passiert auch nichts wirklich Aufregendes.

[4] P und P' tauschen hier die Rollen

Das Programm P enthalte gerade n Variablen, die wir der Einfachheit halber als v_1 bis v_n bezeichnen. Dann enthält das Programm P' für jedes v_i eine Funktiondefinition f_i, die jede Konfiguration C (also jedes Tupel aus $\mathbb{N}^n$) auf die i-te Komponente der Konfiguration C' abbildet (mit $C \vdash^w C'$). Die Behauptung folgt dann, da eine der Funktionen (nämlich diejenige, die den Wert der Return-Variable berechnet) gerade f ist.
Für die Konstruktion der Funktionen f_i gehen wir strukturell-induktiv vor:

1. Ist P einfach das Programm v_i `= 0;`, so lautet P' wie folgt:

   ```
   fun f_1 (v_1,...,v_n) = v_1
   ⋮
   fun f_{i-1} (v_1,...,v_n) = v_{i-1}
   fun f_i     (v_1,...,v_n) = 0
   fun f_{i+1} (v_1,...,v_n) = v_{i+1}
   ⋮
   fun f_n (v_1,...,v_n) = v_n
   ```

   Dies ist korrekt, da der Effekt der Zuweisung v_i `= 0;` ja lediglich ist, dass v_i auf Null gesetzt wird und alle anderen Variablen ihren Wert erhalten.
2. Ist P einfach das Programm v_i `=` v_j`;`, so lautet P' analog wie oben, nur mit folgender Zeile für f_i:

   ```
   fun f_i (v_1,...,v_n) = v_j
   ```

3. Ist P das Programm v_i`++;`, so lautet die Zeile für f_i:

   ```
   fun f_i (v_1,...,v_n) = succ(v_i)
   ```

4. Ist P das Programm v_i`--;`, so lautet die Zeile für f_i:

   ```
   fun f_i (v_1,...,v_n) = pred(v_i)
   ```

   Hierbei ist `pred` wie üblich am Anfang definiert.
5. Ist P die Hintereinanderausführung zweier Programme P^1 und P^2, beschrieben durch Funktionen f_1^1 bis f_n^1 und f_1^2 bis f_n^2, so lautet P wie folgt:

   ```
   fun f_1 (v_1,...,v_n) = f_1^2(f_1^1(v_1,...,v_n),...,f_n^1(v_1,...,v_n))
   ⋮
   fun f_n (v_1,...,v_n) = f_n^2(f_1^1(v_1,...,v_n),...,f_n^1(v_1,...,v_n))
   ```

   In der Tat wird hierdurch die Berechnung von P^2, repräsentiert durch die f_i^2-Funktionen, gestartet mit den Werten, die die f_i^1-Funktionen – also P^1 – lieferten.
6. Ist P das Programm `loop` (v_i) $\{Q\}$ und ist Q durch Funktionen g_1 bis g_n beschrieben, so lautet P wie folgt:

   ```
   fun h_1 (v_1,...,v_n,0) = v_1
     | h_1 (v_1,...,v_n,succ(t)) = g_1(h_1(v_1,...,v_n,t),...,h_n(v_1,...,v_n,t))
     ⋮
     | h_n (v_1,...,v_n,0) = v_n
     | h_n (v_1,...,v_n,succ(t)) = g_n(h_1(v_1,...,v_n,t),...,h_n(v_1,...,v_n,t))

   fun f_1 (v_1,...,v_n) = h_1(v_1,...,v_n,v_i)
   ⋮
   fun f_n (v_1,...,v_n) = h_n(v_1,...,v_n,v_i)
   ```

   Man zeigt nun durch Induktion Folgendes: Die Funktionen h_i liefern, angewendet auf eine Konfiguration in den ersten n Parametern und einer Zahl t in dem $(n+1)$-sten Parameter, gerade die Konfiguration nachdem die Schleife t-mal durchlaufen wurde. Für den Induktionsanfang gilt, dass, wenn die Schleife gar nicht durchlaufen wird ($t = 0$), die gleiche Konfiguration wie vorher erreicht wird, weshalb die h_i's hier die Identität darstellen. Für den Schritt beachte man einfach, dass die h_i's in jedem Rekursionsschritt einmal die g_i's und damit das Programm Q anwenden. Hieraus folgt nun, dass die f_i korrekt sind, denn sie wenden h_i ja gerade mit der Iterationszahl v_i an.

Da hiermit alle Möglichkeiten, wie P aufgebaut sein kann, abgedeckt sind, folgt die Behauptung. □

16-19

Vereinfachte Beispiele für die Umwandlung eines PrimitiveML-Programms in ein Loop-Programm.

Beispiel

```
fun plus2(x) = succ(succ(x))
fun plus4(x) = plus2(plus2(x))
```

wird zu

```
uint plus4(uint x)
{
  uint temp1;
  uint temp2;
  temp1 = x;      // Berechne plus2(x)
  temp1++;
  temp1++;        // plus2(x) in temp1 gespeichert
  temp2 = temp1; // Berechne plus2(temp1)
  temp2++;
  temp2++;
  return temp2;  // Rückgabe ist plus2(plus2(x))
}
```

Beispiel

```
fun add(x,0)        = x
  | add(x,succ(y)) = succ(add(x,y))
```

wird zu

```
uint add(uint x,uint y)
{
  uint temp;
  uint current_y;
  uint previous;

  previous = x;  // Initial_term
  loop (y) {     // Rekursion
    // Werte succ(add(x,y)) aus
    temp = previous; // temp = add(x,y)
    temp++;          // temp = succ(add(x,y))
    previous = temp;

    current_y++;
  }
}
```

16-20

Vereinfachte Beispiele für die Umwandlung eines Loop-Programms in ein PrimitiveML-Programm.

Beispiel

```
uint foo(uint x)
{
  x--;
  x--;
  x++;
  x++;
  return x;
}
```

wird zu

```
fun pred(0) = 0
  | pred(succ(x)) = x

fun foo(x) = succ(succ(pred(pred(x))))
```

Beispiel

```
uint bar(uint x, uint y, uint z)
{
  z++;
  loop (z) {
    x++;
    y--;
  }
  return x;
}
```

wird zu

```
fun pred(0) = 0 | pred(succ(x)) = x

fun help_x(x,y,0) = x
  | help_x(x,y,succ(t)) = succ(help_x(x,y,t))
  | help_y(x,y,0) = y
  | help_y(x,y,succ(t)) = pred(help_y(x,y,t))

fun bar(x) = help_x(x,y,succ(z))
```

16.3 μ-Rekursion

16.3.1 Syntax von MüML

Syntax von MüML
μ-Rekursion

16-21

▶ Definition: μ-Rekursionsterme
Wir erweitern die Definition der Terme um folgende Regel:

- Ist t ein Term, so auch

  `(min` v`.` t `== 0)`

Andere Schreibweisen sind $(\mu x)[t = 0]$ oder $(\mu x\colon t = 0)$.

Beispiel

```
fun even (0)        = succ(0)
  | even (succ(x)) = odd(x)
  | odd  (0)        = 0
  | odd  (succ(x)) = even(x)

fun add(x,0)        = x
  | add(x,succ(y)) = succ(add(x,y))

fun one = (min z. odd(add(succ(0),z)) == 0)
```

16.3.2 Semantik von MüML

16-22

Die Semantik eines MüML-Programms

▶ **Definition:** Semantik eines MüML-Programms

Sei P ein MüML-Programm. Dann beschreibt P eine Menge von *partiellen* Funktionen $f_i\colon \mathbb{N}^{n_i} \dashrightarrow \mathbb{N}$. Diese Funktionen sind analog zur Semantik von PrimitiveML definiert mit folgenden Änderungen:

1. Ein Term der Form

   ```
   (min v. t == 0)
   ```

   wertet zur Zahl $x \in \mathbb{N}$ aus, für die
   1.1 t mit der Variable v belegt mit x gerade 0 ergibt und
   1.2 t mit der Variable v belegt mit einem beliebigen $y < x$ definiert ist und nicht zu 0 auswertet.

   Gibt es kein solches x, so ist der Wert des Terms undefiniert.
2. Ist in einer Termauswertung ein Teilterm undefiniert, so ist der gesamte Term undefiniert.

16-23

Die Klasse der rekursiven Funktionen.

▶ **Definition:** Partiell-rekursive und total-rekursive Funktionen

Eine Funktion $f\colon \mathbb{N}^n \dashrightarrow \mathbb{N}$ heißt *partiell-rekursiv*, wenn sie von einem MüML-Programm P beschrieben wird. Ist f sogar total (also eine echte Funktion), so heißt f (zusätzlich) *total-rekursiv*.
Die Klasse aller partiell-rekursiven Funktionen bildet die Klasse R. Die Klasse aller total-rekursiven Funktionen bezeichnet man mit R_0.

Aufgrund der Definitionen gilt unmittelbar:

$$\mathrm{PR} \subseteq \mathrm{R}_0 \subseteq \mathrm{R}.$$

16.3.3 Partiell-rekursiv = While-berechenbar

16-24

Jede partiell-rekursive Funktion ist While-berechenbar und umgekehrt.

▶ **Satz**

Sei $f\colon \mathbb{N}^n \dashrightarrow \mathbb{N}$. Dann sind folgende Aussagen äquivalent:

1. *f ist partiell-rekursiv.*
2. *f ist While-berechenbar.*
3. *f ist RAM-berechenbar.*
4. *f ist Turing-berechenbar.*

Beweis. Wir müssen nur die Äquivalenz der ersten beiden Aussagen nachweisen. Sei zunächst f partiell-rekursiv via eines While-Programms P. Die Umwandlung in ein MüML-Programm funktioniert genauso wie bei PrimitiveML, einzig eine Anwendung eines μ-Operators muss gesondert behandelt werden.
Sei `(min v. t == 0)` ein Term und sei P_t ein Programm, dass t auswertet. Dann leistet folgendes While-Programm das Gewünschte:

```
uint v;
uint test;

v = 0;
test ← P_t(v)
while (test != 0) {
  v++;
  test ← P_t(v)
}
```

Für die andere Richtung müssen wir ein Programm der Form `while (`v_i` != 0) {`Q`}` umwandeln. Die geschieht mittels eines Programms, das wie folgt aufgebaut ist:

- Mittels primitiver Rekursion werden Funktionen definiert, die den Effekt einer die i-fache Ausführung von Q berechnen.
- Mit einer Anwendung der μ-Operators kann man dann die kleinste Zahl berechnen, so dass die für v_i zuständige Funktion Null liefert.
- Setzt man diese Zahl dann wieder in die primitive Rekursion ein, so erhält man die gewünschten Werte.

Seien g_i Funktionen, die Q beschreiben, siehe den Beweis von Satz 16-18. Dann lautet ein MüML-Programm zur Berechnung von P wie folgt:

```
fun h1(v1,...,vn,0) = v1
  | h1(v1,...,vn,succ(t)) = g1(h1(v1,...,vn,t),...,hn(v1,...,vn,t))
  ⋮
  | hn(v1,...,vn,0) = vn
  | hn(v1,...,vn,succ(t)) = gn(h1(v1,...,vn,t),...,hn(v1,...,vn,t))

fun steps(v1,...,vn) = (min t. hi(v1,...,vn,t) == 0)

fun f1(v1,...,vn) = h1(v1,...,vn,steps(v1,...,vn))
⋮
fun fn(v1,...,vn) = h1(vh,...,vn,steps(v1,...,vn))
```

Wie schon im Beweis von Satz 16-18 argumentiert, berechnen die h_i gerade die Werte der Variablen v_i nach t Ausführungen von Q. Folglich berechnet die `step`-Funktion, wie oft man Q ausführen muss, bis $v_i = 0$ gilt. Hieraus folgt, dass die f_i die korrekten Werte liefern. □

16.4 *Die kleenesche Normalform

Die Idee hinter der kleeneschen Normalform.

16-25

- Durch den μ-Operator können Programme entstehen, die »in Endlosschleifen« verschwinden.
- Es wäre deshalb schön, den Einsatz dieses Operators *zu minimieren*.
- Wir werden gleich sehen, dass man *jede berechenbare Funktion mit nur einer Anwendung des Operators berechnen kann*.
- Für imperative Programme bedeutet dies: Jedes Programm lässt sich so umschreiben, dass es nur *eine einzige While-Schleife* enthält und sonst nur Loop-Schleifen.

Der Satz über die kleenesche Normalform.

16-26

▶ Satz

Sei f partiell-rekursiv. Dann gibt es ein MüML-Programm, das f beschreibt und nur eine Anwendung des μ-Operators enthält.

Beweis. Aufgrund von Satz 16-24 ist f auch RAM-berechenbar. Wir hatten dann gezeigt in Satz 15-25, dass sich jedes RAM-Programm in ein While-Programm umwandeln lässt. Im Beweis wurde dafür *nur eine While-Schleife* benutzt. Wandelt man nun das While-Programm in ein MüML-Programm um, so wird diese eine While-Schleife durch genau einen μ-Operator ersetzt. □

Zusammenfassung dieses Kapitels

16-27

1. Funktionale Sprachen definieren Funktionen mittels *Verkettung* und *verschiedenen Formen von Rekursion*.
2. Die *einfachste Form* der Rekursion ist die *primitive Rekursion*, die *äquivalent* ist zur *Loop-Berechenbarkeit*.
3. Die μ*-Rekursion* findet die kleinste Nullstelle einer Funktion und ist *äquivalent* zur *While-Berechenbarkeit*.
4. Jedes Programm kann in eines mit nur einer While-Schleife umgeformt werden (dies nennt man dann die kleenesche Normalform).

Zum Weiterlesen

[1] R. Reischuk. *Unterlagen zum Lehrmodul Theoretische Informatik,* Vorlesungsskript, 2008. Kapitel 5.2.

17-1

Kapitel 17
Die Church-Turing-These

Intuitiv-berechenbar = Turing-berechenbar

Lernziele dieses Kapitels

1. Geschichte der Turingmaschine kennen
2. Aussage der Church-Turing-These kennen
3. Begriff der »Turing-Mächtigkeit« kennen

17-2

Inhalte dieses Kapitels

Worum es heute geht

In den letzten Kapiteln wurden eine ganze Reihe doch sehr heterogener Konzepte eingeführt: nichtdeterministische Turingmaschinen, Register-Maschinen, minimalistische imperative Programmiersprachen und funktionale Programmiersprachen. Ob Sie diese Konzepte spannend finden, müssen Sie selber wissen – gähnend langweilig waren auf jeden Fall die Sätze, die wir über Mächtigkeit dieser Modelle bewiesen haben: Alle Kapitel endeten mit dem Satz »Mit dem neuen Modell können wir genau so viel berechnen wie mit deterministischen Turing-Maschinen.« (Natürlich schreibt diesen Satz mathematisch etwas hübscher und unverständlicher auf, die Aussage ist aber genau diese.)

Liegt diese Ohnmacht der Modelle an der raffinierten, ja gar perfiden Auswahl der Maschinenmodell durch den Autor? Hat er vielleicht die logischen Programmiersprachen genau deshalb mit keiner Silbe erwähnt, weil bei diese viel mehr zu holen ist? Maschinenmodelle aus prähistorischer Zeit (nach Informatiker-Zeitrechnung, also älter als fünf Jahre) werden ausführlich behandelt, moderne Quanten-Computern hingegen wie zufällig unter den Teppich gekehrt – was hat der Autor zu verschweigen? Wieso werden DNA-Computer mit Nichtachtung gestraft?

Wie bei den meisten Verschwörungstheorien ist die Wahrheit eher profan: Man könnte die Liste der Modelle und Programmiersprachen noch über viele, viele weitere Vorlesungen verlängern – man würde kein neues Maschinen- oder Programmiermodell finden, das mächtiger ist als die Turing-Maschine. Glauben Sie mir, viele Leute haben das versucht.

Erinnern wir uns daran, wie Turing seine Turing-Maschine eingeführt hat: Ausgehend von der Beobachtung, was ein Mensch *prinzipiell überhaupt jemals* berechnen könnte, hat er ein Modell definiert, das genau diese von Menschen durchführbaren Berechnungen nachvollzieht. Folglich müsste ein Maschinenmodell, das mehr kann als die Turing-Maschine, auch mehr können, als Turings idealisierter Mathematiker. Die Church-Turing-These besagt deshalb, dass einfach alles, was man »intuitiv« berechnen kann, auch von einer Turingmaschine berechnet werden kann.

Wie beweist man die Church-Turing-These? Gar nicht. Die These ist kein mathematischer Satz, sondern eine philosophische Aussage. Das Problem ist, dass der Begriff »intuitiv berechenbar« mathematisch nicht genau definiert ist – und definiert man diesen, so landet man eben genau bei einem Modell wie der Turing-Maschine. Dann würde man also nur beweisen, dass Turing-Maschinen genau das können, was Turing-Maschinen können. Das erinnert stark an den Slogan »Was Friseure können, können nur Friseure« der Friseurinnung und ist

auch in etwa von der gleichen intellektuellen Nahrhaftigkeit.

Stimmt die Church-Turing-These? Es scheint so. Andererseits könnte man sich durchaus vorstellen, dass neuartige Computer wirklich *mehr* können als Turing-Maschinen. Beispielsweise könnte man sich zunächst berechtigte Hoffnungen machen, dass Quanten-Computer Berechnungen anstellen können, an die Turing nicht gedacht hat. Diese Hoffnung zerschlägt sich jedoch schnell – Quantenrechner können auch nicht mehr als der PC von der Stange; sie sind nur schneller.

Trotzdem bleibt die Church-Turing-These nur eine These. So könnten in exotischen Umgebungen (wie in der Nähe eines Schwarzen Loches) durchaus Berechnungen möglich sein, an denen eine Turing-Maschine scheitert.

17.1 Die Church-Turing-These

17.1.1 Die Ackermann-Funktion

17-4

Sind alle »intuitiv berechenbaren« Funktionen Loop-berechenbar?

- Als man sich Anfang des 20. Jahrhunderts daran machte, den Begriff der »Berechenbarkeit« zu formalisieren, stand man vor einem Problem.
- Es war »intuitiv klar«, welche Funktionen berechenbar sind, aber es gab keine formale Definition.
- Ein *heißer Kandidat* waren 1926 die *primitiv-rekursiven* Funktionen.
- Die Klasse PR hat auch *viele schöne Eigenschaften*:
  - Sie enthält nur (totale) Funktionen.
  - Sie ist (recht) leicht zu definieren.
  - Sie ist gegenüber *ganz vielen* Operationen abgeschlossen.

17-5

Ein paar Überlegungen dazu, wie schnell Loop-berechenbare Funktionen wachsen können.

Eine einfache Beobachtung betreffend Loop-Programme ohne Loops

- Sei $f\colon \mathbb{N} \to \mathbb{N}$ Loop-berechenbar via eines Programms, das *keine Loop-Schleifen* enthält.
- Das Programm habe c Zeilen.
- *Dann gilt* $f(x) \leq c + x$ *für alle* x.

▶ **Folgerung**
Die Funktion $f(x) = 2x$ *ist nicht Loop-berechenbar via eines Programms, das keine Loop-Schleifen enthält.*

Beweis. Zum Zwecke des Widerspruchs nehmen wir an, es gäbe doch ein Programm P für f. Ist n die Länge von P, so ist $f(n+1) = 2n+2$, jedoch müsste nach obiger Überlegung $f(n+1) \leq n+n+1 = 2n+1$ gelten. □

Loop-Programme mit einem Loop

- Sei $f\colon \mathbb{N} \to \mathbb{N}$ Loop-berechenbar via eines Programms, das *nur eine Loop-Schleife* enthält.
- Das Programm habe c Zeilen.
- In der Loop-Schleife kann eine Variable um maximal c erhöht werden.
- Die Schleife kann maximal $c+x$ Mal durchlaufen werden.
- *Insgesamt gilt also* $f(x) \leq x + c + (c+x)c \leq dx$ *für alle* $x \geq 1$ *und eine geeignete Konstante* d.

✎ Zur Diskussion

Geben Sie eine Funktion $f\colon \mathbb{N} \to \mathbb{N}$ an, die *nicht* von Loop-Programmen mit nur einer Loop-Schleife berechnet werden kann.

Loop-Programme mit zwei Loops

- Sei $f\colon \mathbb{N} \to \mathbb{N}$ Loop-berechenbar via eines Programms, das nur *zwei* Loop-Schleifen enthält.
- Das Programm habe c Zeilen.
- In der inneren Loop-Schleife kann eine Variable maximal mit einer Konstante c multipliziert werden.
- Die Schleife kann maximal $c+x$ Mal durchlaufen werden.
- *Insgesamt gilt also $f(x) \leq d^x$ für alle $x \geq 1$ und eine geeignete Konstante d.*

✎ Zur Diskussion

Geben Sie eine Funktion $f\colon \mathbb{N} \to \mathbb{N}$ an, die *nicht* von Loop-Programmen mit zwei Loop-Schleifen berechnet werden können.

Was können Loop-Programme mit n Schleifen?

17-6

Wir halten fest: Loop-Programme mit

- null Schleifen können maximal $c+x$ berechnen, also konstant oft *den Nachfolger nehmen*;
- einer Schleife können maximal $c \cdot x$ berechnen, also konstant oft x *aufaddieren*;
- zwei Schleifen können maximal c^x berechnen, also konstant oft x *aufmultiplizieren*;
- drei Schleifen können maximal konstant oft x *aufpotenzieren* (also $x^{\left(x^{\cdot^{\cdot^{(x^x)}}}\right)}$);
- vier Schleifen können maximal konstant oft x *aufhyperpotenzieren*;
- fünf Schleifen können maximal konstant oft x *aufhyperhyperpotenzieren*;
- und so weiter.

Die Ackermann-Funktion.

17-7

▶ Definition: Ackermanns Originaldefinition

Die *Ackermann-Funktion* $\varphi\colon \mathbb{N}^3 \to \mathbb{N}$ ist rekursiv definiert:

$$\varphi(x,y,0) = y+1,$$

$$\varphi(x,0,n+1) = \begin{cases} a, & \text{falls } n = 0, \\ 0, & \text{falls } n = 1, \\ 1, & \text{falls } n > 1, \end{cases}$$

$$\varphi(x,y+1,n+1) = \varphi\big(x, \varphi(x,y,n+1), n\big).$$

Man kann nun recht leicht zeigen, dass gilt:

- $\varphi(x,y,0) = y+1$, also die *Nachfolge*.
- $\varphi(x,y,1) = x+y$, also die *Addition*.
- $\varphi(x,y,2) = x \cdot y$, also die *Multiplikation*.
- $\varphi(x,y,3) = x^y$, also die *Potenzbildung*.
- $\varphi(x,y,4)$ ist die *Hyperpotenzbildung*.
- $\varphi(x,y,5)$ ist die *Hyperhyperpotenzbildung*.
- Und so weiter.

Zentrale Eigenschaften der Ackermann-Funktion.

17-8

▶ Satz

Die Ackermann-Funktion ist nicht Loop-berechenbar, aber While-berechenbar.

Beweis. Zum Zwecke des Widerspruchs nehmen wir an, sie wäre Loop-berechenbar via eines Programms P.

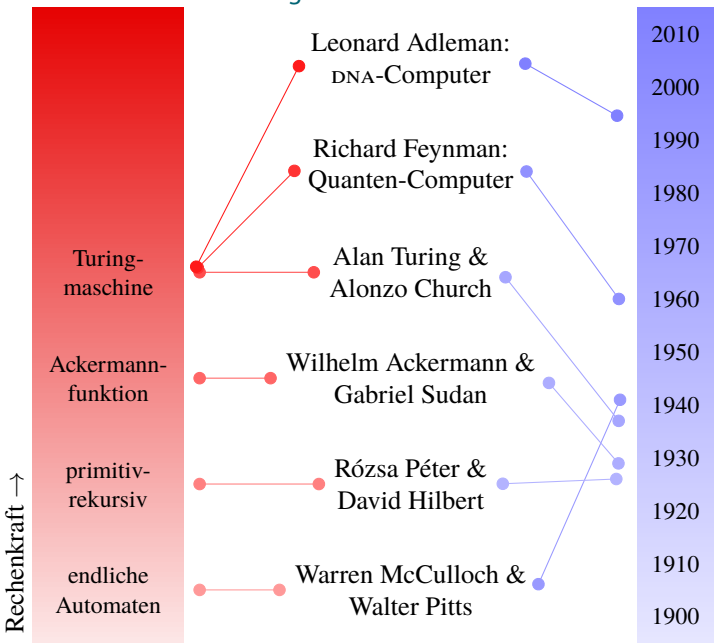
- Das Programm P habe c Zeilen.
- Dann enthält es auch maximal c verschachtelte Schleifen.
- Folglich kann P maximal die c-fache Hyperpotenzbildung der Eingabewerte berechnen.
- Damit ist aber $\varphi(c+1,c+1,c+1)$ *größer* als die größte Ausgabe, die P bei Eingabe $(c+1,c+1,c+1)$ machen kann. Widerspruch.

Dass φ While-berechenbar ist, sieht man beispielsweise durch Angabe eines RAM-Programms für diese Funktion. □

17.1.2 Geschichte

17-9

Geschichte der Berechnungsmodelle



17.1.3 Die These

17-10

Die Church-Turing-These

These

Die »intuitiv berechenbaren« Funktionen sind gerade die partiell-rekursiven Funktionen.

- Diese These kann man *prinzipiell nicht beweisen*.
- Man kann aber versuchen, sie zu *widerlegen*.
- Dazu müsse man eine Funktion ähnlich der Ackermann-Funktion finden, »die man berechnen kann« (wie auch immer), die aber nicht von Turing-Maschinen / RAMs / While-Programmen / MüML-Programmen berechnet werden kann.

17.2 Alternative Modelle

17-11

Das Konzept der »Turing-Mächtigkeit«

- Wir haben viele Modelle kennengelernt, die »genauso viel können wie Turing-Maschinen«.
- Dabei muss man manchmal einige definitorische Klippen umschiffen (zum Beispiel statt »Worten« »Zahlen« benutzen oder ähnliches).
- Andererseits gibt es auch Modelle, die weniger mächtig sind (primitiv-rekursive Funktionen, Loop-Berechenbarkeit, endliche Automaten, Kellerautomaten).

▶ Definition: Turing-Mächtigkeit

Man nennt ein Berechnungsmodell *Turing-mächtig*, wenn es alle Turing-berechenbaren Funktionen »berechnen« kann.

Wir haben schon gesehen, dass folgende Modelle Turing-mächtig sind:

- Turing-Maschinen (wenig überraschend)
- Nichtdeterministische Turing-Maschinen (Satz 13-15)
- Register-Maschinen (Satz 14-32)
- While-Programme (Satz 15-25)
- MüML-Programme (Satz 16-24)

17.2.1 Grammatiken

Allgemeine Grammatiken sind Turing-mächtig.

17-12

Behauptung
Allgemeine Grammatiken ist Turing-mächtig.

Diese Behauptung ist etwas schwammig, da ja Grammatiken gar keine Funktionen berechnen. Genauer gilt Folgendes:

▶ **Satz**
Sei L eine Sprache. Dann sind folgende Aussagen äquivalent:

- *Es gibt eine Turing-Maschine M mit $L(M) = L$.*
- *Es gibt eine Grammatik G mit $L(G) = L$.*

Beweisskizze. Für die erste Richtung sei G eine allgemeine Grammatik. Skript

- Wir konstruieren eine NTM M, die w genau dann akzeptiert, wenn $S \Rightarrow_G^* w$ gilt.
- Die Maschine schreibt zunächst S auf ein Band.
- Dann wählt sie nichtdeterministisch eine Regel aus, sucht die linke Regelseite auf dem Band und ersetzt diese durch die rechte Regelseite.
- Den vorherigen Schritt wiederholt sie so lange, bis keine Nonterminale mehr auf dem Band stehen.
- Falls dann gerade die Eingabe auf dem Band steht, akzeptiert sie.

Für die zweite Richtung sei M eine Turingmaschine.

- Man baut zunächst eine Grammatik G', die »das Verhalten der Turing-Maschine« simuliert.
- Die Idee ist, dass die Worte während der Ableitung gerade immer die aktuellen Konfigurationen der Maschine darstellen.
- Da sich der Kopf immer nur sehr lokal bewegt, lässt sich der Übergang von einer Konfiguration zur nächsten tatsächlich durch Grammatik-Regeln beschreiben.
- Ausgehend von der so konstruierten Grammatik G' braucht man noch folgenden Trick: In G werden vom Startsymbol aus mit geeigneten Regeln *alle möglichen akzeptierenden Konfigurationen* erzeugt; weiterhin enthält G alle Regeln von G', *aber mit der linken und rechten Regelseite vertauscht*.
  Dadurch *beginnt* eine Ableitung mit dem Startsymbol, führt zu einer akzeptierenden Konfigurationen und *endet* mit einem Wort, das akzeptiert wird. □

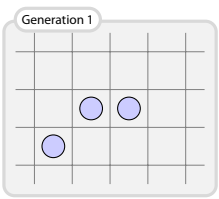
17.2.2 Game of Life

Conways Game of Life.

17-13

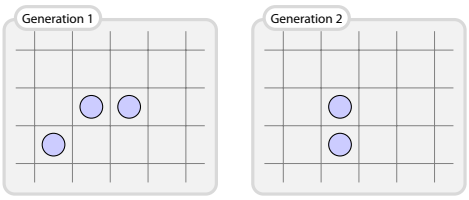
Das Spielbrett

- Das *Spielbrett* ist ein unendliches Gitter.
- In jedem Gitterquadrat kann sich eine *Zelle* befinden oder auch nicht.
- Am Anfang sind einige Gitterquadrate mit Zellen belegt. Sie bilden die *Anfangspopulation*.



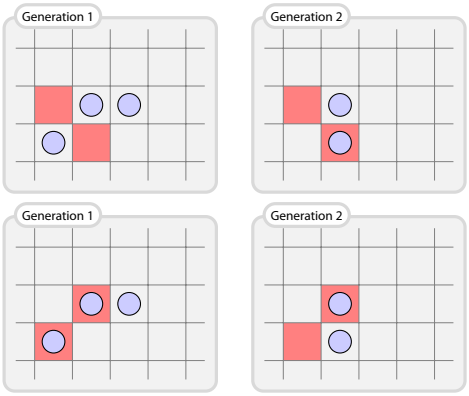
Die Spielrunden

- Das Spiel verläuft in *Generationen* (Runden).
- In jeder Generation werden eventuell neue Zellen *geboren*, alte können *sterben* und Zellen können auch einfach *überleben*.
- Dafür sind die acht umliegenden Zellen wichtig, genannt ihre *Umgebung*.



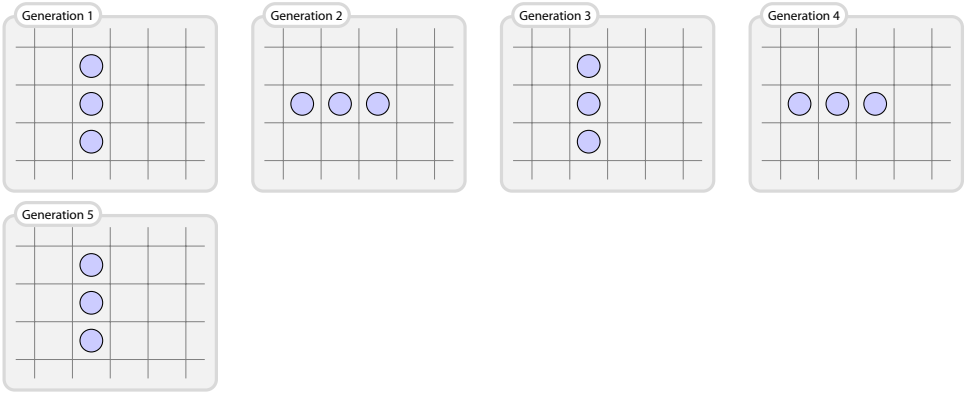
Die Regeln

- Eine Zelle wird *geboren*, wenn es in ihrer Umgebung genau *3* Zellen gibt.
- Eine Zelle *überlebt*, wenn es in ihrer Umgebung genau *2* oder *3* Zellen gibt. Sonst *stirbt* sie an Vereinsamung oder Überbevölkerung.

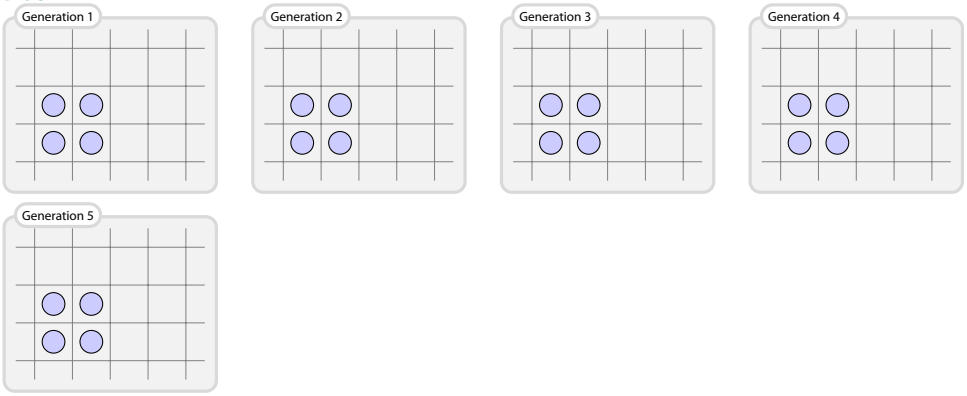


Ein paar interessante Anfangspopulationen

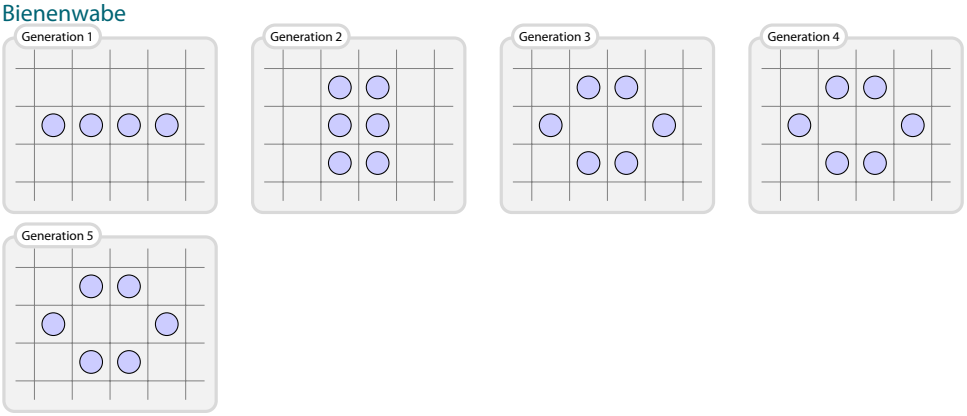
Blinker



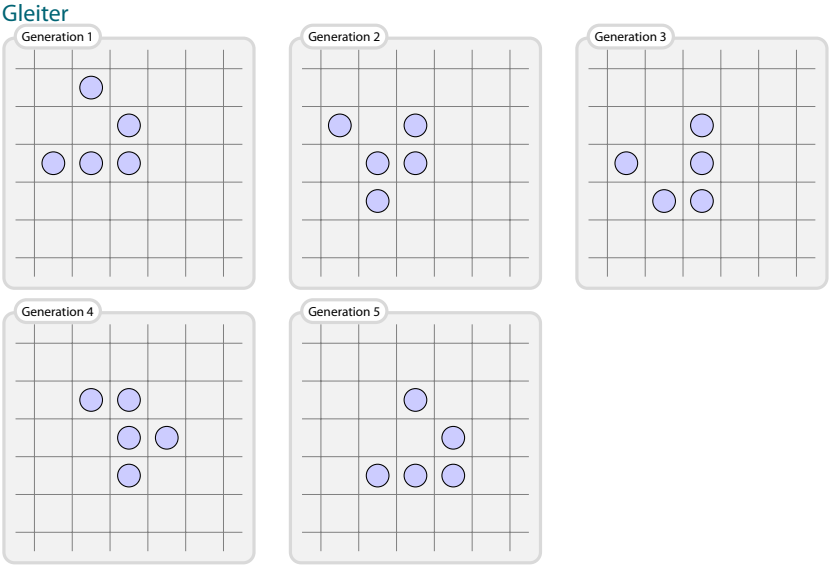
Block



Bienenwabe



Gleiter



Einfache Regeln können komplexe Effekte haben. 17-15

Behauptung

Das Game of Live ist Turing-mächtig.

Das ist wieder etwas schwammig. Genauer gilt folgendes:

▶ Satz

Es gibt eine (einfache) Kodierung von Worten $w \in \{0,1\}^*$ als Life-Spielbretter, so dass für jede (Turing-)akzeptierbare Sprache $L \subseteq \{0,1\}^*$ gilt: Es gilt $w \in L$ genau dann, wenn ausgehend von dem Spielbrett für das Wort w nach endlichen vielen Schritten eine Population erreicht wird, in der eine bestimmte Zelle bevölkert ist.

Dies beweist man, indem man mit Gleitern, Gleiterkanonen, Reflektoren, Verknüpfern und allerlei weiteren Konstruktionen die Konfigurationsfolgen einer Turing-Maschine auf dem Spielbrett nachvollzieht.

Zusammenfassung dieses Kapitels

1. Die Church-Turing-These besagt, dass Turing-Maschinen genau die Funktionen berechnen können, die auch intuitiv berechenbar sind. 17-16
2. Ein Modell heißt *Turing-mächtig*, wenn es genauso mächtig ist wie Turing-Maschinen.
3. Register-Maschinen, While-Programme, MüML-Programme und allgemeine Grammatiken sind Turing-mächtig; endliche Automaten und Loop-Programme hingegen nicht.
4. Das wohl einfachste Turing-mächtige Modell ist Conways Game of Life.

Zum Weiterlesen

[1] Paul Chapman. Life Universal Computer, 2002. http://www.igblan.free-online.co.uk/igblan/ca/, Zugriff Dezember 2009

Hier wird gezeigt, wie man mit dem Game of Life Register-Maschinen simulieren kann.

[2] The LifeWiki. http://www.conwaylife.com/wiki, Zugriff Dezember 2009

Eine Quelle vielfältiger Informationen zum Thema. Hier findet man auch ein recht gutes Applet zur Simulation des Spiels.

[3] R. Reischuk. *Unterlagen zum Lehrmodul Theoretische Informatik,* Vorlesungsskript, 2008. Kapitel 5.5

18-1

Kapitel 18
Unentscheidbarkeit I: Das Halteproblem

Warum es keine Super-Debugger gibt

Lernziele dieses Kapitels

1. Aufbau und Funktion einer universellen Turingmaschine verstehen
2. Das Halteproblem verstehen
3. Den Beweis der Unentscheidbarkeit verstehen
4. Eigene Unentscheidbarkeitsbeweise führen können

18-2

Inhalte dieses Kapitels

Worum es heute geht

Das Halteproblem ist eigentlich ganz einfach: Es ist die Frage, ob ein in Form seines Programmtextes gegebenes Computerprogramm bei einer bestimmten Eingabe anhält oder eben nicht. Diese Frage ist sicherlich vom praktischen Standpunkt aus nicht ganz unwichtig – es wäre doch ausgesprochen schön, wenn man nur noch Programme ausliefern würde, die auf typischen Eingaben immer anhalten.

In diesem Kapitel wird gezeigt werden, dass das Halteproblem nicht entscheidbar ist. Es *kann* also gar keinen Algorithmus geben, der beliebige andere Programme analysiert und uns dann immer korrekt sagt, ob diese anhalten oder nicht. Man beachte allerdings, dass man sehr wohl für einzelne konkrete Programm beweisen kann, dass sie bei bestimmten Eingaben anhalten. Man kann dies sogar in Teilen automatisieren – nur eben nicht völlig allgemein.

Die Unentscheidbarkeit des Halteproblems liegt nicht etwa daran, dass man in der Praxis nicht genügend Rechenzeit oder Speicherplatz hat, um das Problem zu lösen. Es geht *prinzipiell* nicht, egal wie viel Zeit Sie Ihrem Programm genehmigen.

Der Beweis der Unentscheidbarkeit funktioniert in etwa wie folgt: Zunächst beweist man, dass eine bestimmte, reichlich künstlich wirkende Sprache unentscheidbar ist. Dann wird diese Sprache »umgeformt« zu immer neuen Sprachen, die dem Halteproblem immer »ähnlicher« werden. Dabei zeigen wir, dass über alle Umformungen hinweg die entstehenden Sprachen immer unentscheidbar bleiben.

Bei diesem »Umformen von Problemen« bekommt man leicht einen Knoten im Gehirn. Die Argumente gehen nämlich so: Um zu zeigen, dass ein Problem unentscheidbar ist, nimmt man an, es gebe doch einen Entscheider für das Problem. Diesen benutzt man, um einen neuen Entscheider für ein anderes Problem zu bauen, von dem man schon weiß, dass es

keinen Entscheider für das Problem gibt. Man baut also mit viel Liebe etwas, von dem man schon weiß, dass es dieses Ding gar nicht gibt. Bei diesen Argumenten kann es schnell passieren, dass man gar nicht mehr weiß, was man eigentlich weiß. In diesem Fall hilft es, sich mit Sokrates' Ausspruch »Ich weiß, dass ich nichts weiß« zu trösten.

18.1 Das Halteproblem. . .

18.1.1 Halten diese Programme an?

18-4 Fermats Letzter Satz sagt uns, ob folgendes Programm anhält

```
int limit = 1;
while (true) {
  limit++;
  for (int n = 3; n < limit; n++)
    for (int a = 1; a < limit; a++)
      for (int b = 1; b < limit; b++)
        for (int c = 1; b < limit; c++)
          if (Math.pow(a,n) + Math.pow(b,n) == Math.pow(c,n))
            return;
}
```

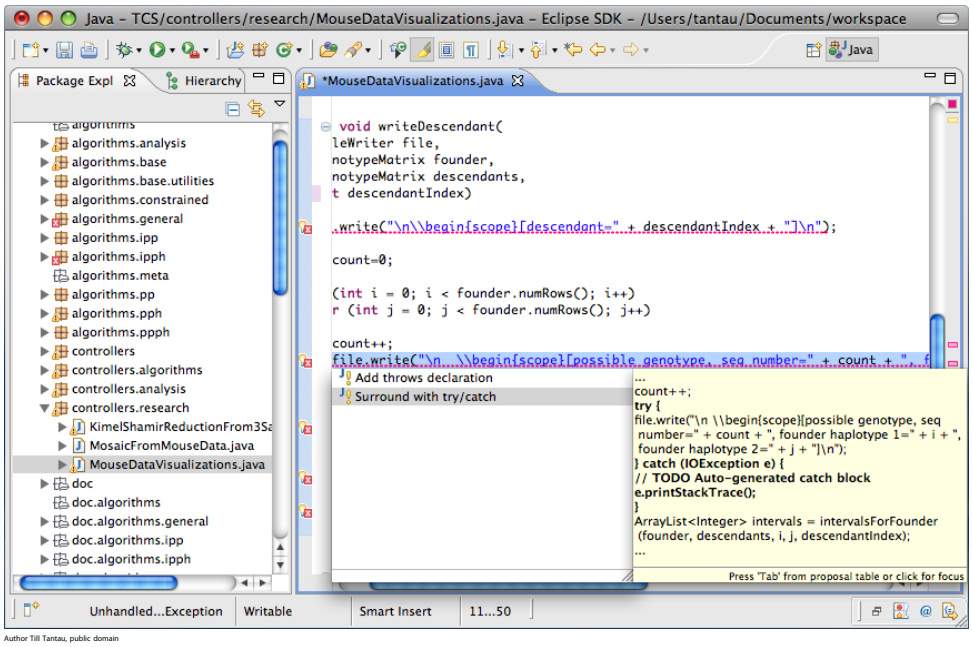
18-5 $1.000.000 für einen Beweis, dass dieses Programm (nicht) anhält

```
public static void main (String[] args) {
  int n = 4;
  while (true) {
    boolean n_is_sum_of_primes = false;

    for (int p = 2; p < n; p++)
      if (is_prime(p) && is_prime(n-p))
        n_is_prime_of_primes = true;

    if (!n_is_prime_of_primes)
      return;

    n = n+2;
  }
}
boolean is_prime(int n) {
  for (int i = 2; i<n; i++)
    if (n % i == 0)
      return false;
  return n>1;
}
```

Was Entwicklungsumgebungen schon können

18-6

- Moderne Entwicklungsumgebungen nehmen Programmierern viel Arbeit ab.
- Sie korrigieren ziemlich gut die *Syntax* automatisch.



Author Till Tantau, public domain

Gesucht: Der Super-Debugger.

18-7

- Es wäre schön, wenn im Jahr 2042 Eclipse Version 23.42 endlich auch *Endlosschleifen automatisch* korrigieren könnte.
- Dann könnte man die obigen Programme eintippen und es würde ein Lämpchen leuchten. Wenn man auf das Lämpchen klickt, dann würde da stehen *»Unhandled endless loop«* und als mögliche Lösungen würde es geben
  - »Replace while-loop by for-loop«
  - »Add `implements Cancable` declaration«
- Es solcher *Super-Debugger* scheint aber schwierig zu programmieren zu sein, denn mit ihm könnten wir *die großen ungelösten Probleme der Mathematik* ganz einfach lösen.
- (Leider) werden wir gleich sehen, dass *man prinzipiell keinen Super-Debugger programmieren kann.*

18.1.2 Kodierung von Maschinen

Maschinen als Worte

18-8

- RAM-*Programme* und *Turing-Programme* sind per Definition intern kompliziert aufgebaute Tupel.
- Um als Eingabe für einen Super-Debugger zu dienen, müssen daraus *Worte* über einem *festen Alphabet* werden.
- In Kapitel 1 wird ausführlich beschrieben, wie so etwas prinzipiell funktioniert.

▶ Definition: Kodierung eines Programms

Sei Π ein RAM-Programm und M eine DTM. Dann bezeichnen $\text{code}(\Pi) \in \text{ASCII}^*$ und $\text{code}(M) \in \text{ASCII}^*$ eine feste *Kodierungen* des Programms beziehungsweise der Maschine.

Die Details dieser Kodierung sind weder praktisch noch theoretisch wichtig.

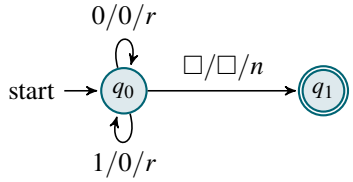
Beispiel

Aus dem Programm Π

```
1 R1 ← 0
2 if R2 > 0 goto 1
3 stop
```

wird das Wort $\text{code}(\Pi) =$ `R1 <- 0; if R2 > 0 goto 1; stop`.

Beispiel

Aus der Maschine $M =$ 

wird das Wort code(M) = `({q_0,q_1},q_0,{q_1},{0,1,Box},{0,1},1,{(q_0,0,q_0,0,r),(q_0,1,q_0,0,r),(q_0,Box,q_1,Box,n)})`

18.1.3 Das formale Problem

18-9

Das formale Halteproblem

▶ Definition: Das Halteproblem
Das *Halteproblem* ist folgende Sprache über dem Alphabet ASCII:

$$\text{HALTING} = \{\text{code}(M)\#w \mid M \text{ ist eine DTM, die auf Eingabe } w \text{ anhält}\}.$$

Alternative Namen für »HALTING« sind »K« und »H«.

18-10

✎ Zur Übung
Welche der folgenden Worte $u_i \in \text{ASCII}^*$ sind Elemente des Halteproblems?

1. u_1 = `({q_0,q_1},q_0,{q_1},{0,1,Box},{0,1},1,{(q_0,0,q_0,0,r),(q_0,1,q_0,0,r),(q_0,Box,q_1,Box,n)})#0010`
2. u_2 = `({q_0,q_1},q_0,{q_1},{0,1,Box},{0,1},1,{(q_0,0,q_0,0,r),(q_0,1,q_0,1,1),(q_0,Box,q_1,Box,n)})#0010`
3. u_3 = `({q_0,q_1},q_0,{q_1},{0,1,Box},{0,1},1,{(q_0,0,q_0,0,r),(q_0,1,q_0,0,r),(q_0,Box,q_1,Box,n)})#######0010`

18.2 . . . ist akzeptierbar . . .

18-11

Unser erstes Ziel: Die Akzeptierbarkeit des Halteproblems

- Zunächst wollen wir zeigen, dass *das Halteproblem akzeptierbar* ist.
- Zur Erinnerung: Dies bedeutet, dass wir eine DTM M_{HALTING} angeben müssen, die
  - bei Eingabe eines Wortes $u \in$ HALTING anhält und akzeptiert und
  - bei Eingabe eines Wortes $u \notin$ HALTING entweder nicht anhält oder anhält und nicht akzeptiert.
- Nun gilt ja $u \in$ HALTING genau dann, wenn $u = \text{code}(M)\#w$ und M bei Eingabe w anhält.

Unser Ziel
Die Maschine M_{HALTING} muss bei *Eingabe* code$(M)\#w$ *herausfinden*, ob die auf dem Eingabeband kodierte Maschine M bei Eingabe w anhält oder nicht.

18.2.1 Universelle Maschine = Interpreter

Universelle Maschinen simulieren andere Maschinen. 18-12

Wie eine universelle Maschine funktioniert

- Eine *universelle Maschine* ist eine Maschine, die kodierte Worte als Eingaben bekommt.
- *Weiterhin* bekommt die universelle Maschine *Eingaben* für die kodierten Maschinen ebenfalls als Eingabe.
- Dann *simuliert* sie die Eingabe-Maschine schrittweise.
- Hält die simulierte Maschine an, so hält auch die universelle Maschine an.

Benutzt man statt eines Maschinen-Modells ein Programmier-Modell (also eine Programmiersprache), so spricht analog von einem *universellen Programm*.

Universelle Maschinen in der Praxis. 18-13

Das Verhalten universeller Maschine / Programm kennt man in der Praxis:

- Ein *Interpreter* ist ein Programm.
- Er bekommt als *Eingabe* einen Programmtext und Parameter für dieses Programm.
- Er führt dieses *Schritt für Schritt* aus.

Beispiele: Interpreter, die universelle Programme sind

- Programme wie `sh` oder `bash` oder alle hiervon abstammenden Shells.
- Programme wie `perl` oder `lua`, die Skriptsprachen interpretieren.

Genaugenommen müssten diese Programme in ihrer eigenen Programmiersprache geschrieben sein, also ein Perl-Interpreter in Perl, ein Lua-Interpreter in Lua und so weiter.

Keine universellen Programme sind beispielsweise `kpdf` oder einfache Textverarbeitungen.

Es gibt nicht für jedes Maschinen-Modell universelle Maschinen. 18-14

- Damit es eine universelle Maschine gibt für eine Maschinen-Modell, muss dieses Modell *mächtig genug* sein.
- Beispielsweise sind *endliche Automaten* nicht mächtig genug: *Es gibt keinen endlichen Automaten, der alle anderen endlichen Automaten simulieren kann, wenn er diese als Eingabe erhält.*
- Es weiteres Beispiel sind Keller-Automaten.

Merke

Man spricht nur dann von einer *universellen* Maschine, wenn das Maschinen-Modell *Turing-mächtig* ist.

Universelle Maschinen 18-15

▶ **Satz:** Existenz einer universellen Turingmaschine

Es gibt eine Turing-Maschine U_{Turing}*, so dass für alle* DTM M *und alle Worte* $w \in \{0,1\}^*$ *gilt:*

1. *M hält bei Eingabe w genau dann an, wenn U_{Turing} bei Eingabe* $\text{code}(M)\#w$ *anhält.*
2. *M akzeptiert die Eingabe w genau dann an, wenn U_{Turing} die Eingabe* $\text{code}(M)\#w$ *akzeptiert.*

Beweisideen

- Die universelle Turingmaschine benutzt ihre Arbeitsbänder, um sich jeweils die aktuelle Konfiguration von M zu merken.
- Um einen Schritt von M zu simulieren, spult M auf ihren Arbeitsbändern herum, um die aktuellen Zeichen an den aktuellen Kopfpositionen zu bekommen.
- Dann schaut sie auf der Eingabe nach, wo ja der Code von M steht, welche Zeichen geschrieben werden müssen.
- Durch viel weiteres Spulen passt sie die Bandinhalte auf ihrem Arbeitsband passend an.

Beweisskizze. Wir wollen eine universelle Turing-Maschine U_{Turing} konstruieren. Auf ihrem *Eingabeband* findet sich ein Wort code(M)#w, wobei M eine DTM ist und $w \in \{0,1\}^*$ eine Eingabe für M ist. (Ist die Eingabe schon syntaktisch nicht von dieser Form, so verwirft U_{Turing} die Eingabe sofort.) Auf dem Eingabeband könnte die Maschine beispielsweise folgendes Wort finden:

```
({q_0,q_1},q_0,{q_1},{0,1,Box},{0,1},1,{(q_0,0,q_0,0,r),
(q_0,1,q_0,0,r),(q_0,Box,q_1,Box,n)})#0010.
```

Die universelle Maschine hat mehrere Bänder:

- Auf dem *Zustandsband* merkt sie sich den *aktuellen Zustand* der Maschine M.
- Auf dem *Bänderband* merkt sie sich die *Inhalte der Bänder* der Maschine M (im Interleaving-Verfahren, siehe den Beweis von Satz 14-32).

Die universelle Maschine arbeitet nun grob wie folgt:

1. Zunächst initialisiert sie das Bänderband mit dem Wort, das nach dem Doppelkreuz auf dem Eingabeband steht, und das Zustandsband mit dem Zustand, der am Anfang des Eingabebandes steht.
2. Nun wiederholt sie Folgendes:
   - 2.1 Sie spult an den Anfang des Bänderbandes.
   - 2.2 Sie läuft über das Bänderband nach rechts und kopiert die Zeichen, die an den aktuellen Kopfpositionen stehen, auf ein Arbeitsband.
   - 2.3 Dann läuft sie über die Eingabe (also über code(M)) und sucht eine Übereinstimmung des Zustandbandes plus den gelesenen Symbolen mit einem Tupel aus der Zustandsüberführungsfunktion.
   - 2.4 Wird diese Übereinstimmung gefunden, werden das Zustandband und das Bänderband entsprechend aktualisiert.
   - 2.5 Anderenfalls überprüft die Maschine, ob der aktuelle Zustand auf dem Zustandsband einer der akzeptierenden Zustände ist. Wenn ja, so akzeptiert die universelle Maschine; sonst verwirft sie.

Mittels einer Induktion über die Länge der Berechnung zeigt man nun die behaupteten Eigenschaften von U_{Turing} □

18.2.2 Akzeptierbarkeit des Halteproblems

18-16

Das Halteproblem lässt sich mittels einer universellen Maschine akzeptieren.

▶ Satz

Das Halteproblem ist akzeptierbar, also HALTING ∈ RE.

Beweis. Eine Turing-Maschine M_{HALTING}, die das Halteproblem akzeptiert, arbeitet wie folgt:

1. Bei Eingabe code(M)#w wendet sie die universelle Turingmaschine U_{Turing} auf die Eingabe an.
2. Wenn die universelle Maschine anhält, so akzeptiert M_{HALTING} sofort.

Man beachte, dass M_{HALTING} alle Worte *nicht* akzeptiert, bei denen die Simulation von U_{Turing} *nicht zu einem Ende kommt*.
Insbesondere akzeptiert M_{HALTING} ein Wort u genau dann, wenn dieses ein Element von HALTING ist. □

18.2.3 Universelle Programme

Interludium: Wie gelangen die Programme in die Programme?

18-17

- Bei einem universellen *Programm* sind die Eingaben Programme.
- Unsere Programmiersprachen nehmen aber *nur Zahlen* als Eingabe.

Programmtexte als natürliche Zahlen

- Ein *While-* oder *MüML-Programm* ist per Definition ein Wort über dem Alphabet ASCII.
- Um als Eingabe für diese Programmiersprachen zu dienen, muss daraus aber eine *natürliche Zahl* werden.
- Wie man Worte in Zahlen verwandelt, ist in Kapitel 1 ausführlich behandelt worden.

▶ **Definition:** Gödelnummer eines Programms
Sei $P \in \text{ASCII}^*$ ein While- oder MüML-Programm und $\iota : \text{ASCII} \to \{0,\dots,127\}$ eine feste Einbettung. Dann ist die *Gödelnummer von P* oder auch der *Programm-Index von P* die Zahl $\text{gödel}(P) = \sum_{i=0}^{|P|-1} \iota(P[|P|-i]) \cdot 1000^i$, siehe auch Definition 1-14.

Beispiel

Aus dem Programm

```
fun zero = 0
```

wird die Gödelnummer $\text{gödel}(P) = 102.117.110.032.122.101.114.111.032.061.032.048$ (in Worten: einhundertzweiquintilliardeneinhundertsiebzehnqunitillionen. . . einundsechzigmillionenzweiunddreizigtausendachtundvierzig).

Universelle Programme

18-18

▶ **Satz:** Existenz eines universellen Programms
Es gibt ein While-Programm U_{while}, so dass für alle While-Programme P und alle Zahlen $n \in \mathbb{N}$ gilt:

1. *P hält bei Eingabe n genau dann an, wenn U_{while} bei dem Eingabepaar $\big(\text{gödel}(P), n\big)$ anhält.*
2. *P gibt bei Eingabe n genau dann $m \in \mathbb{N}$ aus, wenn U_{while} bei dem Eingabepaar $\big(\text{gödel}(P), n\big)$ ebenfalls m ausgibt.*

Beweis. Mittels Satz 16-24 kann die universelle Turingmaschine in ein universelles Programm umgeformt werden.[1] □

Kommentare zum Beweis

[1] Ganz so einfach ist das in Wirklichkeit nicht. Eigentlich muss man noch mit den Kodierungen herumhantieren.

18.3 . . . aber nicht entscheidbar

Wiederholung von 12-21: Entscheidbare Sprachen

18-19

▶ **Definition:** Totale Maschinen
Eine Turing-Maschine heißt *total*, wenn sie bei jeder Eingabe früher oder später anhält.

▶ **Definition:** Entscheidbare Sprachen
Die Klasse REC enthält alle Sprachen, die von *totalen* Turing-Maschinen akzeptiert werden. Man nennt solche Sprachen *entscheidbar* oder *rekursiv*.

Einer der wichtigsten Sätze der Informatik

18-20

▶ **Satz:** Unentscheidbarkeit des Halteproblems
Das Halteproblem ist unentscheidbar, also $\text{HALTING} \notin \text{REC}$.

Mit anderen Worten: *Es kann kein Computer-Programm geben, das für jeden Programmtext korrekt nach endlicher Zeit angibt, ob diese Programm nun anhält oder nicht.*

Beweis. Wir zeigen nacheinander Folgendes:

1. Wir konstruieren zunächst eine Sprache, die *bestimmt nicht* entscheidbar ist.
2. Diese Sprache modifizieren wir dann wiederholt, zeigen aber dabei, dass die entstehenden Sprachen unentscheidbar bleiben.
3. Am Ende erhalten wir, dass HALTING unentscheidbar ist. □

18.3.1 Diagonalisierung

18-21

Diagonalisierung gegen Entscheider.

- Als erstes Ziel wollen wir zunächst eine *Sprache konstruieren*, die *definitiv nicht* entscheidbar ist.
- Dazu *»diagonalisieren wir gegen alle denkbaren Entscheider«.*
- Die Technik des Diagonalisierens geht auf Cantor zurück, der bewies, dass die reellen Zahlen überabzählbar sind.

Idee des Diagonalisierens

- Man nimmt eine *Aufzählung* aller Maschinen her, die *potentiell als Entscheider* in Frage kommen.
- Für jede Maschine sucht man sich ein Wort, bei dem die Maschine *ausgetrickst* werden soll:
  - Sagt die Maschine »das Wort ist in der Sprache«, so ist das Wort gerade nicht in der Sprache.
  - Sagt die Maschine »das Wort ist nicht in der Sprache«, so ist es doch in der Sprache.
- Dann kann keine der Maschinen die Sprache entscheiden – denn jede irrt sich irgendwo.

18-22

Wie entscheiden sich Entscheider?

Die Ausgaben aller möglicher Entscheider M_1, M_2, M_3, … auf verschiedenen Worten w_1, w_2, w_3, …:

Eingabe	M_1	M_2	M_3	M_4	M_5	M_6	…
w_1	»∈«	»∈«	»∉«	»∈«	»∈«	»∈«	…
w_2	»∈«	»∉«	»∉«	»∈«	»∉«	»∈«	…
w_3	»∈«	»∉«	»∉«	»∉«	»∉«	»∈«	…
w_4	»∈«	»∈«	»∉«	»∈«	»∈«	»∈«	…
w_5	»∈«	»∈«	»∉«	»∉«	»∉«	»∈«	…
w_6	»∈«	»∈«	»∉«	»∈«	»∈«	»∉«	…
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋱

Die Sprache DIAGONAL *trickst die Entscheider entlang der Diagonal aus*, indem

- $w_1 \notin$ DIAGONAL, wodurch M_1 sich irrt;
- $w_2 \in$ DIAGONAL, wodurch M_2 sich irrt;
- $w_3 \in$ DIAGONAL, wodurch M_3 sich irrt;
- …

18-23

Konstruktion einer unentscheidbaren Sprache.

- Bei einer Diagonalisierung muss man für jede denkbare Maschine ein Wort haben, *bei dem man die Maschine austrickst*.
- Solch ein Wort heißt auch ein *Diagonalisierungspunkt*.
- Ein besonders einfacher Diagonalisierungspunkt ist *das Wort selber*.

▶ Definition: Diagonalisierungssprache
Die Sprache DIAGONAL $\subseteq$ ASCII* enthält ein Wort $w = \text{code}(M)$ genau dann, wenn M das Wort w *nicht* akzeptiert (also in eine Endlosschleife geht oder es nach endlich vielen Schritten verwirft).

Die Diagonalisierungssprache ist unentscheidbar. 18-24

▶ Lemma
DIAGONAL $\notin$ REC.

Beweis. Sei $M_1, M_2, M_3, \ldots$ eine Aufzählung aller deterministischer Turingmaschinen. Dann kann

- M_1 die Sprache DIAGONAL nicht korrekt entscheiden, denn bei Eingabe code(M_1) *irrt* sich M_1;
- M_2 die Sprache DIAGONAL nicht korrekt entscheiden, denn bei Eingabe code(M_2) *irrt* sich M_2;
- M_3 die Sprache DIAGONAL nicht korrekt entscheiden, dann bei Eingabe code(M_3) *irrt* sich M_3;
- $\ldots$

Insgesamt erhalten wir, dass *überhaupt gar keine* Turing-Maschine die Sprache DIAGONAL entscheiden kann. □

18.3.2 Unentscheidbarkeit des Halteproblems

Beweisrezept: *Unentscheidbarkeit beweisen* 18-25

Ziel
Beweisen, dass eine Sprache L unentscheidbar ist (also L $\notin$ REC).

Rezept

1. Man sucht sich eine Sprache U aus, von der bereits bewiesen wurde, dass sie unentscheidbar ist.
2. Beginne mit: »Wir zeigen, dass aus der Entscheidbarkeit von L die Entscheidbarkeit von U folgen würde – aber wir wissen schon, dass U unentscheidbar ist.«
3. Fahre fort mit: »Nehmen wir also an, L wäre entscheidbar via eines Entscheiders M_L. Dann lässt sich U via folgendem Entscheider M_U entscheiden: Bei Eingabe w berechnet $M_U \ldots$.« Hier ergänzt man passend, wie M_U das Wort w modifiziert und M_L anwendet.
4. Ende mit: »Also ist M_U total und akzeptiert gerade U – und ist mithin ein Entscheider für U.«

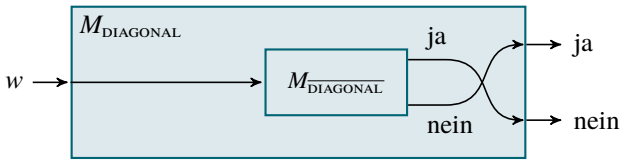
18-26

Das Komplement von DIAGONAL ist unentscheidbar.

▶ **Lemma**
$\overline{\text{DIAGONAL}} \notin \text{REC}$.

Kommentare zum Beweis

Beweis. Wir zeigen, dass aus der Entscheidbarkeit von $\overline{\text{DIAGONAL}}$ die Entscheidbarkeit von DIAGONAL folgen würde – aber wir wissen schon, dass DIAGONAL unentscheidbar ist.
Nehmen wir also an, $\overline{\text{DIAGONAL}}$ wäre entscheidbar via eines Entscheiders $M_{\overline{\text{DIAGONAL}}}$. Dann lässt sich DIAGONAL via folgendem Entscheider M_{DIAGONAL} entscheiden:[1]
Bei Eingabe w ruft M_{DIAGONAL} die Maschine $M_{\overline{\text{DIAGONAL}}}$ als Unterprogramm auf. Falls $M_{\overline{\text{DIAGONAL}}}$ das Wort akzeptiert, so verwirft M_{DIAGONAL} – und umgekehrt.



Da $M_{\overline{\text{DIAGONAL}}}$ total ist, ist auch M_{DIAGONAL} total. Weiterhin akzeptiert $M_{\overline{\text{DIAGONAL}}}$ offenbar genau das Komplement von der Sprache, die M_{DIAGONAL} akzeptiert.[2]
Also akzeptiert M_{DIAGONAL} gerade die Sprache $\overline{\overline{\text{DIAGONAL}}} = \text{DIAGONAL}$ und ist total – und mithin ein Entscheider.[3] □

[1] Bis hier Originaltext aus dem Beweisrezept
[2] Das Wort »offenbar« ist immer gefährlich in Beweisen. Hoffentlich merkt es hier keiner.
[3] Text aus dem Rezept

18-27

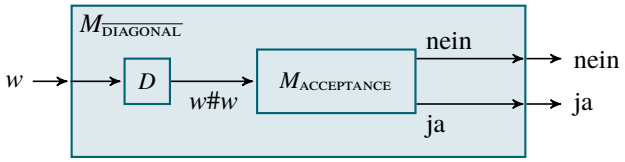
Die Sprache ACCEPTANCE ist unentscheidbar.

▶ **Definition**
$\text{ACCEPTANCE} = \{\text{code}(M)\#w \mid M \text{ akzeptiert } w\}$

▶ **Lemma**
$\text{ACCEPTANCE} \notin \text{REC}$.

Kommentare zum Beweis

Beweis. Wir zeigen, dass aus der Entscheidbarkeit von ACCEPTANCE die Entscheidbarkeit von $\overline{\text{DIAGONAL}}$ folgen würde – aber wir wissen schon, dass $\overline{\text{DIAGONAL}}$ unentscheidbar ist.
Nehmen wir also an, ACCEPTANCE wäre entscheidbar via eines Entscheiders $M_{\text{ACCEPTANCE}}$.
Dann lässt sich $\overline{\text{DIAGONAL}}$ via folgendem Entscheider $M_{\overline{\text{DIAGONAL}}}$ entscheiden:[1]
Bei Eingabe $w = \text{code}(N)$ konstruiert die Maschine $M_{\overline{\text{DIAGONAL}}}$ zunächst durch eine *Verdopplungsschleife* D das neue Wort $w' = w\#w$.[2] Dann wendet M_{DIAGONAL} die Maschine $M_{\text{ACCEPTANCE}}$ als Unterprogramm auf w' an und übernimmt das Ergebnis.



Nun gilt, da $w' = w\#w = \text{code}(N)\#w = \text{code}(N)\#\text{code}(N)$, dass

$$\begin{aligned} & w \in \overline{\text{DIAGONAL}} \\ \iff\ & w \notin \text{DIAGONAL} \\ \iff\ & N \text{ akzeptiert } \text{code}(N) \\ \iff\ & w' \in \text{ACCEPTANCE}. \end{aligned}$$

Also ist $M_{\overline{\text{DIAGONAL}}}$ total und akzeptiert $\overline{\text{DIAGONAL}}$. □

[1] Bis hier Originaltext aus dem Beweisrezept
[2] Dies ist der entscheidende Trick

Das Halteproblem ist unentscheidbar.

18-28

▶ **Lemma**
HALTING $\notin$ REC.

Kommentare zum Beweis

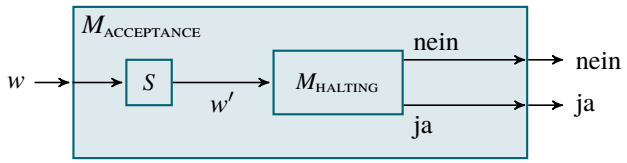
Beweis. Wir zeigen, dass aus der Entscheidbarkeit von HALTING die Entscheidbarkeit von ACCEPTANCE folgen würde – aber wir wissen schon, dass ACCEPTANCE unentscheidbar ist. Nehmen wir also an, HALTING wäre entscheidbar via eines Entscheiders M_{HALTING}. Dann lässt sich ACCEPTANCE via folgendem Entscheider $M_{\text{ACCEPTANCE}}$ entscheiden:[1]
Bei Eingabe $w = \text{code}(N)\#x$ konstruiert die Maschine $M_{\text{ACCEPTANCE}}$ zunächst ein neues Wort w' wie folgt:[2] $w' = \text{code}(N')\#x$, wobei N' identisch ist zu N, nur dass

- N' einen weiteren Zustand hat, in dem sie in einer Endlosschleife bleibt, und
- in jedem Zustand, wo N *anhalten und verwerfen* könnte, die Maschine N' *stattdessen in den neuen Zustand wechselt*.

[1] Bis hier Originaltext aus dem Beweisrezept

[2] Dies ist der entscheidende Trick

Wieder geht w' nur durch eine syntaktische Transformation S aus w hervor.



Es bleibt zu zeigen, dass $M_{\text{ACCEPTANCE}}$ tatsächlich ACCEPTANCE entscheidet. Dies sieht man so:

$$\begin{aligned} & \text{code}(N)\#x \in \text{ACCEPTANCE} \\ \iff & N \text{ akzeptiert } x \\ \iff & N' \text{ hält auf Eingabe } x \text{ an} \\ \iff & \text{code}(N')\#x \in \text{HALTING} \\ \iff & w' \in \text{HALTING}. \end{aligned}$$

Also ist $M_{\text{ACCEPTANCE}}$ total und akzeptiert ACCEPTANCE. □

Zusammenfassung dieses Kapitels

18-29

1. Das Halteproblem ist eine Sprache.
2. Sie enthält alle Worte bestehend aus einer kodierten Maschine (= einem Programmtext) und einer Eingabe, für die diese Maschine anhält.
3. Das Halteproblem ist *akzeptierbar* via einer *universellen Maschine*.
4. Das Halteproblem ist *unentscheidbar*.
5. Man beweist, dass Sprachen unentscheidbar sind, durch Widerspruch.

Zum Weiterlesen

[1] R. Reischuk. *Unterlagen zum Lehrmodul Theoretische Informatik,* Vorlesungsskript, 2008. Kapitel 6.2

Übungen zu diesem Kapitel

Übung 18.1 Das Selbsthalteproblem ist unentscheidbar, schwer, mit Lösung

Zeigen Sie, dass SELF-HALTING $= \{\text{code}(M) \mid M \text{ hält bei Eingabe code}(M) \text{ an}\}$ unentscheidbar ist.

Übung 18.2 Turings Geist befragen

Im Rahmen der Übung zur Vorlesung »Okkultismus für Informatiker« (Modulnummer CS6666) gelingt es Ihnen, Turings Geist zu beschwören. Aufgrund seiner übernatürlichen Kräfte kann Turing Ihnen Fragen betreffend das Halteproblem beantworten: Er kann Ihnen sagen, ob ein Programm anhält, das Sie in Blut auf einen Sarg geschrieben haben.

1. Welches Programm würden Sie aufschreiben, um herauszufinden, ob jede gerade Zahl Summe von drei Primzahlen ist?
2. Welches Programm würden Sie aufschreiben, um herauszufinden, ob jede Zahl die Summe von vier Quadratzahlen ist?

Übung 18.3 Das Halteproblem für die leere Eingabe ist unentscheidbar, mittel

Zeigen Sie, dass EMPTY-HALTING $= \{\text{code}(M) \mid M \text{ hält bei Eingabe } \lambda \text{ an}\}$ unentscheidbar ist.

Übung 18.4 Unentscheidbarkeit beweisen, mittel

Zeigen Sie, dass folgende Sprachen nicht entscheidbar sind:

1. $\{\text{code}(M)\#w \mid M \text{ ist eine DTM und es existiert ein } y \in \{0,1\}^*, \text{ so dass } M \text{ bei Eingabe } w \text{ das Wort } y \text{ ausgibt}\}$.
2. $\{\text{code}(M)\#w\#y \mid M \text{ ist eine DTM, die bei Eingabe } w \text{ das Wort } y \text{ ausgibt}\}$.

Übung 18.5 Unentscheidbarkeit beweisen, mittel

Zeigen Sie, dass folgende Sprache nicht entscheidbar ist: $\{\text{code}(M)\#w \mid M \text{ ist eine DTM, so dass die Anzahl der Schritte nach denen } M \text{ bei Eingabe } w \text{ anhält, keine Primzahl ist}\}$.

Übung 18.6 Unentscheidbarkeit beweisen, schwer

Wiederholen Sie Übung 18.5, nur für die Sprache, wo die Anzahl der Schritte gerade eine Primzahl sein soll.

Übung 18.7 Unentscheidbarkeit beweisen, mittel

Zeigen Sie, dass folgende Sprachen nicht entscheidbar sind:

1. $\{\text{code}(M) \mid M \text{ ist eine DTM, die auf allen Eingaben hält}\}$.
2. $\{\text{code}(M)\#w\#y \mid M \text{ ist eine DTM, die bei Eingabe } w \text{ ein Wort } z \in \{0,1\}^* \text{ mit } z \neq y \text{ ausgibt}\}$.
3. $\{\text{code}(M)\#w \mid M \text{ ist eine DTM, so dass die Anzhal der Schritte, nach denen } M \text{ bei Eingabe } w \text{ anhält, gerade ist}\}$.

19-1

Kapitel 19
Unentscheidbarkeit II: Satz von Rice

Alle semantischen Eigenschaften von Programmen sind unentscheidbar

Lernziele dieses Kapitels

1. Den Satz von Rice und seinen Beweis verstehen
2. Konsequenzen für praktische Problemstellungen kennen
3. Den Rekursionssatz und den Fixpunktsatz kennen

19-2

Inhalte dieses Kapitels

Worum es heute geht

Dieses Kapitel beinhaltet eine gute und eine schlechte Nachricht. Zunächst die schlechte Nachricht: Alle semantischen Eigenschaften von Programmen sind unentscheidbar. Jetzt die gute: Jedes Programm kann auf seinen eigenen Programmtext zugreifen.

Im Laufe dieses Kapitels soll es hauptsächlich darum gehen, diese beiden Nachrichten zunächst überhaupt verständlich zu machen, sie dann mathematisch etwas exakter zu fassen, ihnen schöne Namen zu geben und sie schließlich auch zu beweisen.

19.1 Der Rekursionssatz

19.1.1 Zur Einstimmung

Programme, die etwas über sich selbst aussagen.

19-4

– Es ist leicht, ein Programm zu schreiben, das *seine eigene Länge ausgibt*:

```
class Example1 {
  public void example () {
    System.out.println
      ("Dieses␣Programm␣hat␣116␣Zeichen.");
  }
}
```

– Es ist *viel schwieriger*, ein Programm zu schreiben, das *seinen eigenen Programmtext* ausgibt:

```
class Example2 {
  public void example () {
    System.out.println(
      "class␣Example2␣{\n" +
```

```
        "␣␣public␣void␣example␣()␣{\n" +
        "␣␣␣␣System.out.println(\n" +
        "␣␣␣␣␣␣\"class␣Example2␣{\\n\"␣+␣"
          + ??? );
    }
}
```

19-5

Worum es beim Rekursionssatz geht

- In manchen Situationen ist es schön, wenn ein Programm auf *den eigenen Programmtext* zugreifen kann.
- Das kann man praktisch natürlich ganz einfach lösen: Der Programmtext liegt als Datei vor und man öffnet und liest eben diese Datei.
- Die Frage ist aber, ob dies auch *ohne solches »Schummeln« geht*.
- Der Rekursionssatz besagt grob: »Ja«.

19.1.2 Der Rekursionssatz

19-6

Der Rekursionssatz in verschiedenen Formulierungen.

▶ **Satz:** Der Rekursionssatz, Version für Sprachen
Sei M_g eine Turing-Maschine, die als Eingaben Worte der Form $\text{code}(M)\#w$ *erwartet. Dann gibt es eine Maschine M_f, so dass für alle Worte $w \in \Sigma^*$ gilt:*

$$\text{code}(M_f)\#w \in L(M_g) \iff w \in L(M_f).$$

▶ **Satz:** Der Rekursionssatz, Version für Funktionen
Sei $g\colon \mathbb{N}^{n+1} \to \mathbb{N}$ partiell-rekursiv. Dann existiert eine partiell-rekursive Funktion $f\colon \mathbb{N}^n \to \mathbb{N}$, die von einem While-Programm P_f berechnet wird, so dass für alle Tupel $(x_1, \ldots, x_n) \in \mathbb{N}^n$ gilt:

$$g\big(\text{gödel}(P_f), x_1, \ldots, x_n\big) = f(x_1, \ldots, x_n).$$

Bemerkungen:

- Wir beweisen nur die Version für Funktionen – die Version für Sprachen kann man darauf zurückführen, siehe Übung 19.1.
- Das Verhalten von M_g auf Eingaben, die die falsche »Form« haben, und von g für erste Parameter, die keine Gödelnummern sind, ist egal.
- Bei »$g\big(\text{gödel}(P_f), x_1, \ldots, x_n\big) = f(x_1, \ldots, x_n)$« bedeutet das Gleichheitszeichen Folgendes:
  1. Entweder sind *beide* Funktionen links und rechts des Gleichheitszeichens für ein gegebenes Tupel definiert und haben denselben Wert *oder*
  2. *beide* Funktionen sind für das Tupel undefiniert.

  Es ist also nicht erlaubt, dass eine der beiden Seiten des Gleichheitszeichens definiert ist und die andere nicht.

19-7

Die Aussage des Rekursionssatzes im Detail.

Betrachten wir die Aussage des Rekursionssatzes in der Funktionsvariante:

- Ausgangspunkt ist eine partiell-rekursive Funktion g, die als Eingabe
  1. eine Gödelnummer $\text{gödel}(P)$ eines Programms P bekommt, sowie
  2. weitere Parameter x_1 bis x_n.
- Die Funktion g kann nun »etwas mit $\text{gödel}(P)$ anstellen«, beispielsweise die Länge von P berechnen oder den Programmtext von P umdrehen oder die Gödelnummer von P quadrieren oder diese Gödelnummer mit x_1 multiplizieren etc.
- Das Verhalten von g hängt also in beliebig komplexer Weise von dem jeweiligen P ab.
- Die *Behauptung* ist nun, dass es *immer ein P gibt*, dass genau das macht, was g bei Eingabe $\text{gödel}(P)$ macht.

19.1.3 Beispiele

Was wir aus dem Rekursionssatz folgern können. 19-8

Beispiel
Es gibt ein Programm P, das seine eigene Länge ausgibt. (Die Funktion g gibt einfach die Länge von P aus.)

Beispiel
Es gibt ein Programm P, das seinen eigenen Programmtext ausgibt. (Die Funktion g gibt einfach gödel(P) aus.)

Beispiel
Es gibt ein Programm P, das seinen eigenen umgedrehten Programmtext x_1 mal ausgibt. (Die Funktion g gibt einfach x_1 oft gödel(P) umgedreht aus.)

Konkrete Implementationen in Java finden Sie in den Übungen.

✎ Zur Übung 19-9
Geben Sie für folgende Funktionen g_i an, wie ein Programm P_f lauten könnte, dessen Existenz im Rekursionssatz behauptet wird:

1. $g_1\big(\text{gödel}(P), x\big) = x$.
2. $g_2\big(\text{gödel}(P), x\big) = (\text{Ausgabe von } P \text{ bei Eingabe } x)$.
3. $g_3\big(\text{gödel}(P), x\big) = 1 + (\text{die Ausgabe von } P \text{ bei Eingabe } x)$.

Tipp: Suche Sie bei g_3 ein P_f, das nicht total ist.

19.1.4 Der Beweis

Die Selbstverwuselung 19-10

▶ Definition: Selbstverwuselung
Sei P ein While-Programm der folgenden Bauart:

```
uint P (uint v, uint x1, ..., uint xn)
{
  body
}
```

Dann sei die *Selbstverwuselung von P* das folgende Programm $\tilde{P}$:

```
uint P̃ (uint x1, ..., uint xn)
{
  uint v;
  v++;
  ⋮ // Genau gödel(P) oft wird v erhöht
  v++;
  body
}
```

Die Variable `v` wird also am Anfang auf den Wert der Gödelnummer von P gesetzt; dann wird P »ganz normal« abgearbeitet.

▶ Lemma: Selbstverwuselungslemma
Es gibt ein While-Programm S, das bei Eingabe der Gödelnummer von P die Gödelnummer von $\tilde{P}$ berechnet.

Beweis. Das Programm $\tilde{P}$ geht durch eine recht einfache (wenn auch fummelige) syntaktische Transformation aus P hervor. Eine solche kann man mit etwas Geschick in While programmieren:[1]

Kommentare zum Beweis

[1] Da hatte der Autor wohl keine Lust, einen vernünftigen Beweis aufzuschreiben, und schummelt sich durch.

```
uint S (uint goedelnummer_von_P)
{
  uint goedelnummer_von_P_tilde;
  ... // Code, der P_tilde aus P berechnet
  return goedelnummer_von_P_tilde;
}
```

□

Bemerkung: S ist sogar ein Loop-Programm.

19-11

Beweis des Rekursionssatzes.

Beweis. Betrachten wir die Funktion $g\colon \mathbb{N}^{n+1} \to \mathbb{N}$. Dann gibt es ein While-Programm P_g, das g berechnet. Dies habe folgende Form:

```
uint g (uint e, uint x_1, ..., uint x_n)
{
  body
}
```

Wir konstruieren nun eine neues Programm P_h grob wie folgt:

```
uint h (uint d, uint x_1, ..., uint x_n)
{
  uint e;
  e = S(d); // siehe unten
  body
}
```

Allerdings geht das so nicht direkt, da die Sprache While keine Funktionsaufrufe (`S(d)`) zulässt. Stattdessen muss in der betreffende Zeile der Programmtext von S eingefügt werden. Zum Beweis des Rekursionssatzes müssen wir ein Programm P_f mit bestimmten Eigenschaften konstruieren. Wir behaupten, dass das gesuchte While-Programm P_f gegeben ist durch $\widetilde{P_h}$, also durch das Programm mit der Gödelnummer $S(\text{gödel}(P_h))$. Wir halten fest:

$$\text{gödel}(P_f) = S(\text{gödel}(P_h)).$$

Betrachten wir das Verhalten von $P_f = \widetilde{P_h}$ bei einer Eingabe $(x_1,\ldots,x_n) \in \mathbb{N}^n$. Das Programm $\widetilde{P_h}$ wird zuerst in die Variable d die Gödelnummer des Programms P_h schreiben. Dann wird in die Variable e der Wert $S(d)$, also $S(\text{gödel}(P_h))$, geschrieben. Dann wird der Körper von g ausgeführt, wobei eben die Variable e mit dem Wert $S(\text{gödel}(P_h))$ vorbelegt ist. Folglich berechnet P_f gerade den Wert

$$g\big(\underbrace{S(\text{gödel}(P_h))}_{=\text{gödel}(P_f)}, x_1, \ldots, x_n\big).$$

Wir erhalten also, dass wie behauptet gilt:

$$g\big(\text{gödel}(P_f), x_1, \ldots, x_n\big) = f(x_1,\ldots,x_n).$$

Damit ist alles gezeigt. □

19.1.5 Folgerung: Der Fixpunktsatz

Die Idee des semantischen Fixpunkt. 19-12

- In der Praxis werden Programmtexte *häufig maschinell transformiert*. Beispiele sind:
  - Hinzufügen von Debug-Ausgaben,
  - Hinzufügen eines Virus,
  - Löschen von Kommentar-Zeilen,
  - Hinzufügen einer Schleife zur Zeitmessung (Profiling).
- Mathematisch handelt es sich um Funktionen $h\colon \mathbb{N} \to \mathbb{N}$, die *Gödelnummern* auf *andere Gödelnummern* abbildet.
- Häufig ist man daran interessiert, dass die Transformation möglichst die *Funktionsweise des Programms nicht verändert*.
- Ein Programm, das von einer Transformation *zwar syntaktisch*, aber *nicht semantisch* geändert wird, nennen wir einen *semantischen Fixpunkt der Transformation*.

▶ Definition: Semantischer Fixpunkt
Sei $h\colon \mathbb{N} \to \mathbb{N}$. Ein *semantischer Fixpunkt* von h ist ein Programm P, so dass P und das Programm mit der Gödelnummer $h(\text{gödel}(P))$ dieselbe Funktion berechnen.

Eine einfache Folgerung aus dem Rekursionssatz 19-13

▶ Satz: Fixpunktsatz der Rekursionstheorie
Jede total-rekursive Funktion $h\colon \mathbb{N} \to \mathbb{N}$ hat einen semantischen Fixpunkt.

Beweis. Betrachte folgende Funktion $g\colon \mathbb{N}^2 \to \mathbb{N}$: Der Wert $g\big(\text{gödel}(P), x\big)$ sei gerade die Ausgabe des Programms mit der Gödelnummer $h\big(\text{gödel}(P)\big)$ bei Eingabe x.
Nach dem Rekursionssatz existiert nun ein Funktion f, berechnet von einem Programm P_f, so dass für alle x gilt

$$g\big(\text{gödel}(P_f), x\big) = f(x).$$

Sei Q das Programm mit der Gödelnummer $h\big(\text{gödel}(P_f)\big)$. Dann gilt:

1. P_f gibt bei Eingabe x gerade $f(x)$ aus und
2. Q gibt bei Eingabe x gerade $g\big(\text{gödel}(P_f), x\big)$ aus.

Da $g\big(\text{gödel}(P_f), x\big) = f(x)$, berechnen also P_f und Q dieselbe Funktion. □

19.2 Der Satz von Rice

Semantische Eigenschaften von Programmen scheinen schwierig zu entscheiden zu sein. 19-14
Wir haben bereits zwei wichtige unentscheidbare Probleme kennengelernt:

- Das Halteproblem HALTING.
- Das Akzeptanzproblem ACCEPTING.

In beiden Problemen geht es darum, etwas *über das Verhalten von Maschinen herauszufinden, die Teil der Eingabe sind*, und *beide Probleme sind unentscheidbar*.

Frage
Gibt es andere Fragestellungen in Bezug auf das Verhalten von Programmen, die *entscheidbar* sind?

Um diese Frage zu beantworten, brauchen wir erstmal eine sinnvolle Definition von »semantischer Eigenschaft«.

19.2.1 Indexmengen

19-15 Indexmengen beschreiben semantische Eigenschaften.

▶ Definition: Indexmenge
Eine *Indexmenge* ist eine Sprache über dem Alphabet ASCII mit folgenden Eigenschaften:

1. Sie enthält nur Worte der Form code(M) für DTMs M.
2. Enthält sie ein Wort code(M) und ist M' eine andere Turingmaschine, die *dieselbe Sprache akzeptiert wie* M, so enthält die Indexmenge ebenfalls auch code(M').

Bemerkungen:

- Statt kodierten Turing-Maschinen könnte man natürlich auch kodierten RAM-Programmen hernehmen.
- Ebenso könnte man auch durch *Gödelnummern* kodierte While- oder MüML-Programme benutzen. Die Bezeichnung »Indexmenge« stammt historisch daher, dass Gödelnummern ja eine Art *Indizes von Programmen* darstellen.
- Indexmengen beschreiben *semantische Eigenschaften*, da sie immer alle oder gar keine Maschine enthalten, die sich semantisch gleich verhalten.

19-16 Beispiele von Indexmengen.

Beispiel: Die Indexmenge der immer akzeptierenden Maschinen
Die Menge ALWAYS-ACCEPTING-MASCHINES der Codes aller Maschinen, die alle Eingaben akzeptieren.

Beispiel: Die Indexmenge der niemal-haltenden Maschinen
Die Menge NEVER-ACCEPTING-MASCHINES der Codes aller Maschinen, die gar keine Eingaben akzeptieren.

Beispiel: Die Indexmenge der kontextfreien Sprachen
Die Menge der Codes aller Maschinen, die kontextfreie Sprachen akzeptieren.

Beispiel: Das Halteproblem ist keine Indexmenge
Das Halteproblem ist keine Indexmenge, da es »syntaktisch falsch gebaut« ist: Es enthält eben nicht Codes von Maschinen, sondern Codes von Maschinen zusammen mit einer Eingabe.

19.2.2 Der Satz und sein Beweis

19-17 Alle semantischen Eigenschaften von Programmen sind unentscheidbar.

▶ Satz: Satz von Rice
Sei I eine Indexmenge, die weder leer ist noch den Code einfach aller Maschinen enthält. Dann ist I unentscheidbar, also $I \notin$ REC.

Bemerkungen:

- Die leere Indexmengen und die Indexmenge, die die Codes aller Maschinen enthält, nennt man auch *trivial*.
- Den Satz fasst man deshalb auch oft kurz als »Jede nichttriviale Indexmenge ist unentscheidbar«.

Beweis. Seien M_{in} und M_{out} zwei Maschinen mit code$(M_{\text{in}}) \in I$ und code$(M_{\text{out}}) \notin I$.[1]
Wir führen einen Widerspruchsbeweis und nehmen an, I wäre entscheidbar via einer Maschine M_I.[2]
Betrachte folgende Maschine M_g:

- Sie bekommt als Eingabe ein Wort code$(M)\#w$.
- Sie überprüft mittels M_I, ob code$(M) \in I$ gilt.
  - Ist dies der Fall, so führt sie M_{out} mit Eingabe w aus und akzeptiert, falls diese akzeptiert.

Kommentare zum Beweis

[1] Diese Maschinen gibt es, da I nichttrivial ist.

[2] Dies ist nicht ganz nach dem Beweisrezept für Unentscheidbarkeit, aber recht ähnlich.

– Ist dies nicht der Fall, so führt sie M_{in} mit Eingabe w aus und akzeptiert, falls diese akzeptiert.

Nach dem Rekursionssatz gibt es nun eine Maschine M_f mit

$$\text{code}(M_f)\#w \in L(M_g) \iff w \in L(M_f).$$

Wir untersuchen nun, ob $\text{code}(M_f) \in I$ gilt:[3]

[3] Hieraus wird sich der Widerspruch ergeben

1. Ist $\text{code}(M_f) \in I$, so wird nach Definition von M_g die Maschine M_g für alle w gerade M_{out} simulieren. Also akzeptieren M_{out} und M_f dieselbe Sprache. Da $\text{code}(M_{\text{out}}) \notin I$, folgt $\text{code}(M_f) \notin I$.
2. Ist $\text{code}(M_f) \notin I$, so wird umgekehrt die Maschine M_g für alle w gerade M_{in} simulieren. Also akzeptieren M_{in} und M_f dieselbe Sprache und wir erhalten $\text{code}(M_f) \in I$.

In beiden Fällen erhalten wir einen Widerspruch. □

Zusammenfassung dieses Kapitels

19-18

1. Der Rekursionssatz besagt grob, dass ein Programm »Zugriff auf seinen eigenen Code« haben kann.
2. Eine Indexmenge enthält die Codes von *semantisch äquivalenten Maschinen* (oder die Gödelnummern von semantisch äquivalenten Programmen).
3. Der Satz von Rice besagt, dass überhaupt gar keine semantische Eigenschaft von Programmen entscheidbar ist.
4. Beispiele sind: »Akzeptiert das Programm die Eingabe 42?« oder »Akzeptiert das Programm wenigsten eine Eingabe?«.

Zum Weiterlesen

[1] R. Reischuk. *Unterlagen zum Lehrmodul Theoretische Informatik,* Vorlesungsskript, 2008. Kapitel 6.4 und 6.5.

Übungen zu diesem Kapitel

Übung 19.1 Sprach-Version des Rekursionssatzes, schwer

Beweisen Sie die Version des Rekursionssatzes für Sprache, siehe Folie 19-6.

Tipp: Benutzen Sie den Rekursionssatz für Funktionen, indem Sie eine geeignete Funktion g konstruieren.

Übung 19.2 Ein komplexes selbstdruckendes Programm entwerfen, schwer, mit Lösung

Schreiben Sie ein Programm, das einen Parameter x nimmt und dann seinen eigenen Programmtext x mal ausdruckt und dabei nach jedem Zeichen ein Leerzeichen ausgibt.

Übung 19.3 Ein umdrehend selbstdruckendes Programm entwerfen, leicht

Schreiben Sie ein Programm, das seinen eigenen Programmtext umgedreht ausgibt.

Tipp: Gehen Sie genauso vor wie in Übung 19.2, siehe deren Lösung im Anhang.

Übung 19.4 Ein interessantes selbstdruckendes Programm entwerfen, leicht

Schreiben Sie ein Programm, das einen Parameter x bekommt und jedes x-te Zeichen seinen eigenen Programmtextes ausgibt.

Tipp: Gehen Sie genauso vor wie in Übung 19.2, siehe deren Lösung im Anhang.

Übung 19.5 Indexmengen erkennen, mittel

Entscheiden Sie, ob folgende Mengen nichttriviale Indexmengen sind. Beweisen Sie Ihre Vermutung.

1. $\{\text{code}(M) \mid M$ akzeptiert bei Eingaben 0 und 1 oder akzeptiert bei Eingaben 0 und 1 nicht$\}$,
2. $\{\text{code}(M) \mid$ akzeptiert M bei Eingabe 0, verwirft M bei Eingabe 1 und umgekehrt$\}$,
3. $\{\text{code}(M) \mid$ wenn M bei Eingabe 0 akzeptiert, dann akzeptiert M bei Eingabe 1 nicht und umgekehrt$\}$,
4. $\{\text{code}(M) \mid M$ hat genau 34572 Zustände$\}$,
5. $\{\text{code}(M) \mid M$ akzeptiert das Halteproblem$\}$.
6. $\{\text{code}(M) \mid M$ akzeptiert die Eingabe 100010$\}$.
7. $\{\text{code}(M) \mid$ wenn M bei Eingabe 0 akzeptiert, dann akzeptiert M auch bei Eingabe 1$\}$,
8. $\{\text{code}(M) \mid M$ gerät bei Eingabe 0 oder 1 in eine Endlosschleife$\}$,
9. $\{\text{code}(M) \mid M$ akzeptiert die Eingabe 1 nach genau 3462 Rechenschritten$\}$,

Übung 19.6 Unentscheidbarkeit beweisen, mittel

Zeigen Sie auf zwei Arten, dass die Sprache $L = \{\text{code}(M) \mid M$ akzeptiert bei Eingabe 42$\}$ nicht entscheidbar ist, indem Sie nachweisen, dass sie

1. eine nichttriviale Indexmenge ist und
2. aus der Existenz eines Entscheiders für L die Existenz eines Entscheiders für HALTING folgen würde.

Übung 19.7 Unentscheidbarkeit beweisen, mittel

Wiederholen Sie Übung 19.6 für die Sprache $L = \{\text{code}(M) \mid M$ akzeptiert alle Eingaben$\}$.

Kapitel 20

Unentscheidbarkeit III: Unentscheidbare Probleme

Warum es keine Super-Komprimierer gibt

Lernziele dieses Kapitels

1. Das Postsche Korrespondenzproblem kennen
2. Das Busy-Beaver-Problem kennen
3. Das Konzept der Kolmogorov-Komplexität kennen

Inhalte dieses Kapitels

Worum es heute geht

Zum Abschluss dieses Teils über die Berechenbarkeit soll es noch um einige besonders putzige Exemplare von nichtberechenbaren Sprachen und Funktionen gehen.

Den Anfang macht das Postsche Korrespondenzproblem (das PKP), das, wie der Namen schon sagt, auf Emil Post zurückgeht. Dieses Problem ist deshalb so interessant, weil es so einfach aussieht – wenn man es zum ersten Mal sieht, würde man nie auf die Idee kommen, dass es sich um ein unentscheidbares Problem handelt. (Es soll auch schon Assistenten an Universitäten gegeben haben, die Studierenden zur Übung die Aufgabe gestellt haben, ein Programm zur Lösung des PKP zu schreiben.) Die Formulierung dieses Problem ist viel einfacher zu verstehen als die der unentscheidbaren Probleme, die in den vorigen Kapiteln auf den ahnungslosen Leser hereingeprasselt sind: Weder kommen Maschinencodes vor, noch Rekursionen oder wilde Gödelisierungen. Es geht einfach nur um eine Reihe von Paaren von Worten und wie man diese aneinanderreihen könnte.

Als zweites soll es um Biber gehen, konkret um die Spezies *Castor fiber informaticus*. Diese zeichnet sich wie alle Biberarten durch ihren besonderen Fleiß aus; jedoch bevorzugen sie statt dem Bau von Staudämmen das Anhäufen von Einsen auf einem Band.

Zu guter Letzt geht es auch noch um Super-Komprimierer, vornehm *Kolmogorov-Komprimierer* genannt. Eine mit einem Kolmogorov-Komprimierer komprimierte Zeichenkette verhält sich zu einer mit `bzip2` komprimierten in Bezug auf ihre Packungsdichte etwa wie ein Neutronen-Stern zu einem interstellaren Nebel. Mit anderen Worten: Kolmogorov-Komprimierer sind die besten Komprimierer, die es überhaupt geben kann. Da ist es ausgesprochen schade, dass sie leider nicht berechenbar sind.

Als Ausblick möchte ich dann noch aufzeigen, dass es auch Sprachen gibt, die »nicht nur einfach unentscheidbar« sind, sondern die »ganz besonders unentscheidbar« sind. Konkret soll es um Sprachen gehen, die »nicht entscheidbar und sogar nicht akzeptierbar« sind. Ein Beispiel wird das Komplement des Halteproblems sein.

Auch wenn wir es in dieser Vorlesung nicht werden vertiefen können, so sei doch erwähnt, dass selbst das Komplement des Halteproblems nicht »das Schwierigste« ist, was die Theorie zu bieten hat. Man kann nämlich Sprachen bauen, die selbst dann noch unentscheidbar bleiben, wenn man »Turings Geist« aus Übung 18.2 befragen könnte. In der etwas abgehobenen

Wissenschaft der fortgeschrittenen Rekursionstheorie spricht man natürlich nicht von »Geistern« sondern formal (bitte nicht lachen) von »Orakeln« (Sie sollten doch nicht lachen). Von Rekursionstheoretikern wird das Halteproblem auch gerne als Sprache 0 bezeichnet, da sie »so einfach« ist (eine gewisse Arroganz solcher Theoretiker gegenüber praktischen Problemen lässt sich nicht von der Hand weisen). Wenn man nun 0 als Orakel beliebig befragen kann, so kann man aber wieder ein neues Halteproblem definieren, genannt $0'$ oder auch der »Jump des Halteproblems«. Dieses Problem ist nun wahrlich schwierig – es ist selbst dann unentscheidbar, wenn man das Halteproblem jederzeit *free of charge* beantwortet bekommt. Dann kann man natürlich auch noch betrachten, was passiert, wenn man $0'$ als Orakel zur Verfügung hat – und es gibt wieder ein Halteproblem, genannt $0''$, das noch viel schwieriger ist. Und dann gibt es natürlich auch $0'''$ und $0''''$ und so weiter. Und dann kann man natürlich noch gegen die Hierarchie dieser Sprachen diagonalisieren, um ein noch viel schwierigeres Problem zu erhalten – welches man wieder wiederholt »jumpen« kann. Dies führt dann für jede Ordinalzahl zu einer neuen Kategorie von noch größerer Unentscheidbarkeit.

Vielleicht ist es auch ganz gut so, dass wir diese Probleme in dieser Vorlesung nicht vertiefen.

20.1 Nichtberechenbares

20.1.1 Das Postsche Korrespondenzproblem

20-4

Über das Postsche Korrespondenzproblem.


Public domain

- Das *Postsche Korrespondenzproblem* (PKP) ist eine *Sprache*.
- Sie geht auf Emil Post zurück.
- Die Sprache ist recht einfach aufgebaut und *erscheint zunächst einfach zu entscheiden*.
- Wir werden aber gleich sehen, dass diese Sprache *unentscheidbar ist*.

20-5

Die Idee

- Die *Eingabe* für das PKP ist ein *Wörterbuch*:

Deutsch	Vogonisch
apfel	apf
birne	rnx
mus	elmus
nixe	qwertz

  (Vogonisch, insbesondere in Gedichtform, ist nicht unbedingt für seine melodische Struktur bekannt.)
- Die Frage ist nun, ob man einen Satz in der einen Sprache finden kann, so dass die *wortwörtliche Übersetzung* in die andere Sprache *genau denselben Satz* wie in der ersten Sprache liefert.
- Beispielsweise wird aus »apfel mus« die Übersetzung »apf elmus«, also – wenn man die Leerzeichen ignoriert – in beiden Fällen »Apfelmus«.
- Man nennt dies eine *Korrespondenz*.

20-6

Das formale Problem

▶ **Definition:** Wörterbuch
Sei Σ ein Alphabet. Ein *Wörterbuch für* Σ ist ein Menge $W = \{(x_1, y_1), \ldots, (x_n, y_n)\}$ von Paaren $(x_i, y_i) \in \Sigma^* \times \Sigma^*$.

▶ **Definition:** Korrespondenz
Sei W ein Wörterbuch. Eine *Korrespondenz* ist ein Wort w, so dass eine Folge $\big((p_1, q_1), (p_2, q_2), \ldots, (p_m, q_m)\big)$ von Paaren aus $(p_i, q_i) \in W$ existiert mit $w = p_1 \circ \cdots \circ p_m$ und auch $w = q_1 \circ \cdots \circ q_m$.

Beachte: Ein Paar $(x, y) \in M$ kann mehrfach vorkommen.

▶ Definition: Die Sprache PKP

Das *Postsche Korrespondenzproblem (*PKP*)* enthält (die Codes von) allen Wörterbüchern, für die es eine Korrespondenz gibt.

Beispiel

$\{(1,100000),(00,1),(0110,0)\} \in$ PKP

Ein erstaunliches Resultat. 20-7

▶ Satz

PKP *ist unentscheidbar.*

Beweisideen

- Man benutzt das Beweisrezept »Unentscheidbarkeit beweisen« mit:
  - Als bekanntermaßen unentscheidbare Sprache U wählt man EMPTY-HALTING $= \{\text{code}(M) \mid M \text{ hält bei Eingabe } \lambda \text{ an}\}$ (siehe Übung 18.3).
  - Als L wählt man PKP.
- Man muss also zeigen: Hat man einen Entscheider M_{PKP} für PKP, so gibt es auch einen Entscheider $M_{\text{EMPTY-HALTING}}$ für EMPTY-HALTING.
- Sei $\text{code}(M)$ eine Eingabe für $M_{\text{EMPTY-HALTING}}$.
- Der Trick ist nun *ein Wörterbuch W zu bauen*, so dass *Korrespondenzen zu W gerade endliche Berechnungen von M kodieren*.
- Ziel ist es, folgende Invariante einzuhalten: *in der ersten »Sprache« ist immer gerade eine Konfiguration weniger kodiert sein als in der zweiten*. Die erste Sprache »hinkt quasi hinterher«.
- Wenn die Berechnung als endend erkannt wird, holt dann die erste Sprache auf und die Korrespondenz wird perfekt.
- Wenn die Berechnung hingegen nie endet, dann lässt sich auch keine Korrespondenz finden.

Mehr Details finden Sie in den Übungen 20.1 und 20.2.

20.1.2 Das Busy-Beaver-Problem

Über fleißige Biber. 20-8

Unknown author, Creative Commons Attribution Sharealike

- Ein *fleißiger Biber* ist ein Turing-Maschine.
- Er hat *möglichst wenige Zustände*,...
- ...kennt nur die Symbole 1 und □,...
- ...startet auf einem leeren Band,...
- ...schreibt möglichst viele 1en...
- ...und hält dann aber irgendwann an.

Definition von fleißigen Bibern. 20-9

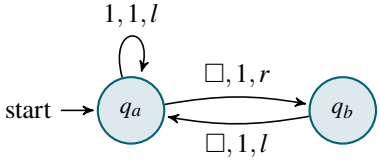
▶ Definition: Fleißige Biber

Ein *Biber* ist eine DTM mit folgenden Eigenschaften:

1. Das Bandalphabet ist $\{\square, 1\}$.
2. Es gibt nur ein Band.
3. In jedem Schritt muss sich der Kopf bewegen (es sind nur »links« und »rechts« als Richtungen erlaubt, »neutral« hingegen nicht).
4. Sie hält bei einem anfänglich leeren Band nach endlich vielen Schritten an.

Ein *fleißiger Biber* ist ein Biber, der eine *maximale Anzahl an* 1*en schreibt* unter allen Bibern mit derselben Anzahl Zuständen.

Beispiel: Fleißiger Biber mit zwei Zuständen

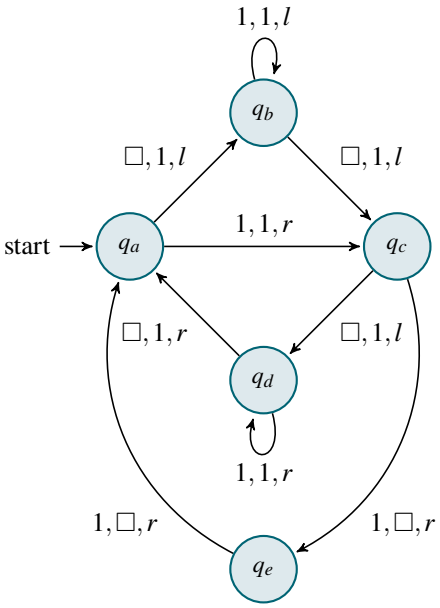


▶ Definition: Fleißige-Biber-Funktion
Die *Fleißige-Biber-Funktion busybeaver*: $\mathbb{N} \to \mathbb{N}$ ordnet jeder Zahl $n \geq 0$ die Anzahl an 1en zu, die ein fleißiger Biber mit n Zuständen schreibt.

Anzahl n an Zustände	$busybeaver(n)$
1	0
2	3
3	5
4	12
5	vermutlich 4097
6	$\geq 4{,}64 \cdot 10^{1439}$

20-10

Kandidat für einen fleißigen Biber mit fünf Zuständen.



- Dieser Biber macht 47.176.870 Schritte, bevor er anhält.
- Er hinterlässt 4097 viele 1en.
- Es ist nicht bewiesen, dass dies tatsächlich ein *fleißiger* Biber ist.

20-11

Die Fleißige-Biber-Funktion ist nicht berechenbar.

▶ Satz
Die Fleißige-Biber-Funktion ist nicht rekursiv.

Beweisideen.

- Kann man die Fleißige-Biber-Funktion ausrechnen, so kann man für eine gegebene Maschine eine *obere Schranke* für die Anzahl an 1en ausrechnen, die sie schreiben kann.
- Simuliert man eine Maschine und schreibt sie *mehr* 1*en als die Fleißige-Biber-Funktion angibt*, so *weiß man*, dass die Maschine *nie anhalten* wird.
- Damit lässt sich (grob) das Halteproblem entscheiden.

Beweis. Sei M eine DTM. Man kann M leicht syntaktisch in einer Maschine M' umformen mit folgenden Eigenschaften:

1. Sie hat das Bandalphabet $\{\square, 1\}$, bleibt nie stehen und hat nur ein Band.
2. Für jeden Schritt, den M macht, schreibt M' (mindestens) eine neue 1 auf ihr Band.

[1]

Wäre nun die Busy-Beaver-Funktion berechenbar, so könnte man EMPTY-HALTING wie folgt entscheiden:[2] Bei Eingabe code(M) macht $M_{\text{EMPTY-HALTING}}$ Folgendes:

- Berechne code(M'). Sei n die Anzahl der Zustände von M'.
- Berechne $b = busybeaver(n)$.
- Simuliere M für $b+1$ Schritte.
- Falls M während der Simulation anhält, akzeptiere; sonst verwerfe.

Dies ist korrekt, denn entweder hält M nach maximal b Schritten an; oder M hält nie an. Würde M nämlich nach mehr als b Schritten anhalten, so wäre M' ein fleißigerer Biber als der fleißigste Biber mit n Zuständen – ein Widerspruch. □

Kommentare zum Beweis

Skript

[1] Die Konstruktion einer solchen Maschine M' ist etwas fummelig, aber eben auch nicht schwierig.

[2] Fast das Rezept »Unentscheidbarkeit beweisen« nur hier mit einer Funktion statt einer Sprache

20.1.3 Kolmogorov-Komplexität

Über Komprimierer.

20-12

Yan Shuangchun, Lesser GNU public license

- Ein *Komprimierer* ist ein Programm.
- Es bekommt als *Eingabe einen String*.
- Es liefert als *Ausgabe einen String,*
  - der möglichst kurz ist und
  - aus dem sich die Eingabe rekonstruieren lässt.

Beispiel: Klassische Komprimierer

- Lempel-Ziv-Komprimierer (`gzip`)
- Huffmann-Komprimierer (`gzip`, `mpeg`)

Beispiel: Moderne Komprimierer
Burrows-Wheeler-Transformation (`bzip2`)

Die Idee hinter dem Kolmogorov-Komprimierer

20-13

- Bei guten Komprimierern gibt es *generell viele Möglichkeiten, einen String zu kodieren.*
- Dadurch ist dann der Komprimierer besonders flexibel: Er kann die *»passendste« Art* suchen, einen String zu kodieren.
- Man kann sich nun fragen: *Was ist die ultimativ flexibelste Art, einen String zu kodieren?*
- Die Antwort lautet: *Als Programmtext, der diesen String als Ausgabe liefert.*

Der Kolmogorov-Komprimierer

20-14

Public domain

▶ **Definition**
Sei $w \in \{0,1\}^*$ ein String. Wir sagen, eine 1-Band-Turingmaschine M *produziert* w, wenn

- sie mit dem leeren Band gestartet
- nach endlich vielen Schritten anhält und
- dann auf dem Band genau w steht.

▶ **Definition:** Kolmogorov-Komprimierer
Der *Kolmogorov-Komprimierer* ist eine Funktion $K\colon \{0,1\}^* \to \text{ASCII}^*$, so dass für alle Worte $w \in \{0,1\}^*$ gilt:

1. $K(w)$ ist der Code einer Maschine M,
2. M produziert w und
3. die Länge von code(M) ist minimal.

20-15

Der Kolmogorov-Komprimierer kann sehr stark komprimieren. . .

Manche Worte lassen sich sehr gut kolmogorov-komprimieren:

Beispiel

Das Wort $1^{1000000}$ (also eine Million 1en) lässt sich mittels eines sehr kurzen Programms beschreiben:

```
for (int i=0; i<1000000; i++) System.out.print("1");
```

(Das muss man natürlich eigentlich als Turing-Maschine aufschreiben – aber wir wissen ja, dass das alles irgendwie dasselbe ist.)

Beispiel

Das Wort $1^{10^{100}}$ (also ein Googol viele 1en) lässt sich auch kurz beschreiben:

```
int p = 1;
for (int i=0; i<100; i++) p = p * 10;
for (int i=0; i<p;   i++) System.out.print("1");
```

20-16

. . . aber nicht alles lässt sich komprimieren.

- Andererseits lassen sich *viele Worte gar nicht komprimieren*.
- Dies liegt daran, dass man ja sonst *rekursiv immer wieder komprimieren könnte* – irgendwann muss Schluss sein.
- Grob gesprochen lassen sich »zufällige Worte« nicht komprimieren:

Gut komprimierbar	schlecht komprimierbar
0000000000	0101101101
0000011111	1011111010
0101010101	0010110111

- Tatsächlich kann man *so definieren*, wann *ein einzelnes Wort »zufällig ist«:* Wenn es sich nicht kolmogorov-komprimieren lässt.

20-17

Ein trauriges Resultat.

▶ **Satz**

Der Kolmogorov-Komprimierer ist nicht rekursiv.

Beweis. Nehmen wir an, wir könnten K berechnen.

- Sei w_0 das kürzeste und lexikographisch erste Wort mit $|K(w_0)| = 1000000$.
- Dann kann man ein (kurzes) Programm schreiben, das in einer Schleife alle Worte in der Reihenfolge $\lambda, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, \ldots$ durchläuft.
- Für jedes Wort w berechnet es $K(w)$ und gibt das erste aus, für das $|K(w)| = 1000000$.
- Dieses Programm ist dann *sehr kurzes Programm, das w_0 beschreibt* – aber das kürzeste solche Programm sollte ja Länge 1000000 haben. □

20.2 Nichtberechenbareres

Unentscheidbar, unentscheidbarer, am unentscheidbarsten.

20-18

- Wir haben *viele unentscheidbare* Probleme kennengelernt.
- Viele davon sind aber wenigstens *akzeptierbar*:
  - HALTING
  - EMPTY-HALTING
  - PKP
- Man kann sich nun fragen, ob es Probleme gibt, die *noch nichtmal akzeptierbar sind.*

Ein einfacher, aber nützlicher Satz.

20-19

▶ Satz
Eine Sprache ist genau dann entscheidbar, wenn sie und ihr Komplement akzeptierbar sind.

Beweis. Sei L entscheidbar. Dann ist L akzeptierbar und (Vertauschung von akzeptierenden und nichtakzeptierende Zuständen) auch $\overline{L}$.
Seien nun L und $\overline{L}$ akzeptierbar via Maschinen M_L und $M_{\overline{L}}$. Ein Entscheider für L arbeitet wie folgt:

- Bei Eingabe w werden die Maschinen M_L und $M_{\overline{L}}$ auf getrennten Bändern simuliert, jeweils für die Eingabe w.
- Dabei wird immer abwechselnd ein Schritt der einen Maschine und dann der anderen Maschine simuliert.
- *Wenigstens eine der Maschinen akzeptiert nach endlich vielen Schritten.*
- Ist dies M_L, so gibt akzeptiert auch der Entscheider; ist es $M_{\overline{L}}$, so verwirft der Entscheider. □

Beispiel einer wirklich schwierigen Sprache.

20-20

▶ Folgerung
Das Komplement der Halteproblems ist weder entscheidbar noch akzeptierbar.

Beweis. Wäre $\overline{\text{HALTING}}$ akzeptierbar, so wäre HALTING entscheidbar. □

Zusammenfassung dieses Kapitels

20-21

1. Das Postsche Korrespondenzproblem ist eine unentscheidbare Sprache.
2. Ein fleißiger Biber schreibt eine maximale Anzahl 1en auf ein Band bevor er anhält. Diese Anzahl kann man nicht allgemein ausrechnen.
3. Der Kolmogorov-Komprimierer liefert bei Eingabe x ein Programm minimaler Länge, das x ausgibt. Leider ist er nicht berechenbar.
4. Eine Sprache ist genau dann entscheidbar, wenn sie und ihr Komplement akzeptierbar sind. Deshalb ist das Komplement des Halteproblems weder entscheidbar noch akzeptierbar.

Zum Weiterlesen

[1] R. Reischuk. *Unterlagen zum Lehrmodul Theoretische Informatik,* Vorlesungsskript, 2008. Kapitel 6.3, 6.7 und 6.9.

Übungen zu diesem Kapitel

Übung 20.1 Unentscheidbarkeit des eingeschränkten PKP, schwer

Zeigen Sie, dass folgende Sprache unentscheidbar ist:

$$\text{PKP}' = \{W \mid \text{es gibt eine Korrespondenz für } W, \text{ die mit } \triangleright \text{ beginnt}\}.$$

Kommentare zum Beweis

Vervollständigen Sie dazu die Details in folgendem Beweis:

Beweis. Wir zeigen, dass aus der Entscheidbarkeit von PKP′ die Entscheidbarkeit von EMPTY-HALTING folgt, was nach Übung 18.3 einen Widerspruch darstellt.

Sei code(M) gegeben. Wir können annehmen, dass M eine 1-Band-DTM ist und genau einen Zustand hat, in dem sie anhält und der auch keine ausgehende Pfeile besitzt.[1] Die Berechnung von M beginnend mit einem leeren Band durchläuft dann eine Folge Konfigurationen $C_1, C_2, \ldots$, wobei jede Konfiguration aus einem Zustand q_i und einem Bandinhalt besteht. Beim Bandinhalt von C_i seien l_i die Nicht-Blank-Zeichen links vom Kopf und r_i die Nicht-Blank-Zeichen rechts vom Kopf. Anfänglich gilt also $l_1 = r_1 = \lambda$.

[1] Begründen Sie diese Annahme

Eine solche Konfiguration C_i lässt sich nun wie folgt als Wort kodieren: $w_i = l_i q_i r_i$. Die aktuelle Kopfposition wird also durch die Stelle angezeigt, wo der Zustand steht. Die ersten m Schritte der Berechnung von M sind dann in dem Wort $\triangleright w_1\$w_2\$\ldots\$w_m$ kodiert.

Ziel ist es, das Wörterbuch W zwischen zwei Sprachen – nennen wir sie Vogonisch und Klingonisch[2] – so aufzubauen, dass es genau eine Korrespondenz gibt: $\triangleright w_1\$w_2\$\ldots\$w_m$, wobei m die Anzahl der Schritte ist, bis M anhält. Insbesondere soll gelten, dass es genau dann eine Korrespondenz für W gibt, wenn M nach endlich vielen Schritte anhält.

[2] Sie können auch andere Namen wählen, wenn Ihnen diese zu nerdig sind.

Die Idee ist folgende: Mit dem Wörterbuch kann man die Korrespondenz bestehend aus der Berechnung stückweise aufbauen. Dabei wird während des Aufbaus *in Vogonisch immer gerade eine Konfiguration weniger kodiert als in Klingonisch*. Vogonisch »hinkt quasi hinterher«. Erst, wenn die Berechnung als endend erkannt wird, holt Vogonisch auf.

Die Paare des Wörterbuchs sind nun grob wie folgt aufgebaut:

- Es gibt zunächst das »Startpaar« $(\triangleright, \triangleright\square q_0\square\$)$.
  Dieses sorgt dafür, dass man in Vogonisch erstmal (außer dem Zeilenanfangszeichen) noch nichts erzeugen kann, in Klingonisch aber bereits die erste Konfiguration.
- Es gibt Paare der Form (x,x) für $x \in \Gamma \cup \{\$\}$ mit denen ein Zeichen von Klingonisch nach Vogonisch »kopiert« werden kann.
- Es gibt Paare,[3] die ein Zustandsymbol enthalten und angeben, wie die Umgebung des Kopfes sich bei einem Rechenschritt ändert.
- Es gibt Paare, mit denen Vogonisch aufholen kann, wenn das Ende einer Berechnung erkannt wurde.[4]

[3] Dies ist jetzt eine entscheidende Stelle, wo Sie ausführen müssen, wie das gehen soll.

[4] Sobald der spezielle Endzustand erreicht ist, muss Vogonisch einfach »auffüllen«.

Um den Beweis abzuschließen, führen wir den Begriff der »Teilkorrespondenz« ein: Eine *Teilkorrespondenz* über einem Wörterbuch ist ein Wort w, so dass eine Folge $\big((p_1,q_1),(p_2,q_2),\ldots,(p_k,q_k)\big)$ von Paaren aus $(p_i,q_i) \in W$ existiert mit $w = p_1 \circ \cdots \circ p_k$ und w ist ein Anfangsstück von $q_1 \circ \cdots \circ q_k$.

Per Induktion zeigt man nun, dass w genau dann eine Teilkorrespondenz für das konstruierte W ist, wenn es ein Anfangsstück des Wortes $\triangleright w_1\$w_2\$\ldots$ ist.[3]

[3] Dies sollten Sie ausführen.

Hieraus folgt, dass es genau dann eine Korrespondenz für das Wörterbuch gibt, wenn die Berechnung von M endlich ist.[4] □

[4] Dies auch.

Übung 20.2 Unentscheidbarkeit des PKP beweisen, mittel

Zeigen Sie, dass PKP unentscheidbar ist.

Tipp: Sie dürfen benutzen, dass PKP′ aus Übung 20.1 unentscheidbar ist.

Übung 20.3 Postsches Korrespondenzproblem, mittel

Sei $W = \{(1,100),(1,101),(10,00),(011,11)\}$ eine Eingabe für das Postsche Korrespondenzproblem.

1. Existiert eine Korrespondenz für W? Existiert eine Korrespondenz, die mit dem ersten Paar $(1,100)$ beginnt?
2. Existiert eine Korrespondenz für $W' = \{(10,101),(101,011),(011,11)\}$?

Übung 20.4 Postsches Korrespondenzproblem, mittel

Sei $W = \{(1,01),(01,1),(01,11),(11,01)\}$ eine Eingabe für das Postsche Korrespondenzproblem.

1. Existiert eine Korrespondenz für W? Existiert eine Korrespondenz, die mit dem ersten Paar $(1,01)$ beginnt?
2. Existiert eine Korrespondenz für $W' = \{(1,01),(01,11)\}$?

Teil IV

Wie schwer sind Probleme?

Der Theoretischen Informatik wird häufig vorgeworfen, dass sie zu theoretisch sei. Zwar geht dieser Vorwurf ins Leere – einem menschlichen Wesen wirft man ja auch nicht vor, dass es zu menschlich sei – jedoch muss man zugeben, dass Resultate der Theoretischen Informatik manchmal segensreiche oder fürchterliche Konsequenzen zu haben scheinen; in der Praxis entpuppen sich diese Konsequenzen dann aber als völlig nutzlose Algorithmengespinnste oder zahnlose Endlospapiertiger. Zwar lassen sich nach dem Satz von Rice semantische Eigenschaften von Programmen nicht entscheiden – jedoch können wir im Allgemeinen Endlosschleifen doch recht gut erkennen. Zwar ist die Ackermann-Funktion while-berechenbar – jedoch können wir praktisch $\varphi(42, 23, 65)$ beim besten Willen nicht ausrechnen. Zwar ist der nächste optimale Zug für einen Computer-Spieler bei dem Brettspiel Go selbstverständlich *berechenbar* – jedoch werden auch die besten Go-Programme immernoch von guten Spielern locker geschlagen.

Die Problematik liegt darin, dass wir bis jetzt den Aspekt der *Effizienz* völlig ausgeblendet haben. Es macht eben doch einen Unterschied, ob man, sagen wir, eine Million Schritte zur Lösung eines Problems braucht oder doch eine Quintillion Schritte. Im ersten Fall wartet man eine Millisekunde auf das Ergebnis (sprich: nicht »man« wartet auf das Ergebnis, sondern der Computer wartet darauf, dass »man« das Ergebnis endlich mal zur Kenntnis nimmt), im zweiten Fall wartet man deutlich länger, als das Universum alt ist (wozu man nur bei wirklich wichtigen Fragen bereit sein wird, wie beispielsweise bei der an Deep Thought gestellten).

Die *Komplexitätstheorie* beschäftigt sich ganz allgemein mit der Frage, wie *effizient* sich Probleme lösen lassen. Ähnlich wie beim Begriff der »Berechenbarkeit« ist zunächst nicht klar, was »effizient« nun genau bedeuten soll. Ist ein Problem effizient lösbar, wenn man immer maximal eine Milliarde Schritte braucht, um die Lösung auszurechnen? Dann wäre nichtmal die binäre Suche ein effizienter Algorithmus: Bei Eingaben der Länge größer als $2^{1000000000}$ braucht nämlich auch die binäre Suche mehr als eine Milliarde Schritte. Nun könnte man natürlich auch solch fantastisch große Eingaben einfach verbieten – jedoch erscheinen solche Einschränkungen doch etwas willkürlich.

Die Lösung liegt darin, die Effizienz daran festzumachen, wie der Rechenaufwand eines Programms sich in Abhängigkeit der Eingabelänge entwickelt: Bei langen Eingaben darf das Programm sicherlich länger brauchen als bei kurzen. Diese Idee wird es uns erlauben, Probleme nach ihrer Komplexität zu *klassifizieren.* Diese Klassifikation gelingt recht gut: zum Beispiel werden wir das Sortierproblem als *sehr einfach* klassifizieren, das Finden eines optimalen Zuges beim Spiel Go hingegen als *recht schwer*. Die offiziellen Namen für »sehr einfach« und »recht schwer« lauten übrigens »in logarithmischem Platz berechenbar« und »hart für nichtdeterministische polynomielle Zeit« und Sie – als angehende Akademiker – sollten auch diese vornehm-akademischen Namen verwenden; für den Hausgebrauch reichen dann aber doch »sehr einfach« und »recht schwer«. Man kann sogar beweisen, dass Sortieren *echt einfacher* ist als das Spiel Go – für den Beweis wird allerdings ein etwas größeres Theoriegebäude benötigt, als es in diesem Teil gebaut wird; Interessierte seien auf die Veranstaltung »Komplexitätstheorie« verwiesen.

Das Spannungsfeld von komplexitätstheoretischen Resultaten und dem Schreiben von effizienten Programmen wird übrigens auch schon von unserem Informatik-Nachwuchs hingebungsvoll beackert:

Aus *Lösungen von Schülern zum 28. Bundeswettbewerb Informatik*

> »Da Backtracking in ungeahnten Zeitaufwand ausartet, wobei dieses die optimalste Lösung finden würde, habe ich beschlossen, die Variante mit einem Algorithmus zu verwenden.«

> »Ich benutze Brute Force, weil es zu umständlich ist, einen langen Algorithmus zu erfinden, welcher dann erst auch noch bewiesen werden muss.«

Enden wird dieser Teil, wie auch die gesamte Vorlesung, mit einer für Theoretiker eher peinlichen Geschichte: dem P-NP-Problem. Es handelt sich dabei *nicht* um ein »Problem« im Sinne der Theoretischen Informatik (also *nicht* um eine Sprache). Vielmehr nennt man den Umstand, dass man $\mathrm{P} \neq \mathrm{NP}$ trotz jahrelangem Probieren immernoch nicht beweisen kann, beschönigend das P-NP-Problem. Die Klassen P und NP werden im Laufe dieses Teils noch detailliert vorgestellt; grob gesprochen ist das P-NP-Problem einfach die Frage, ob sich tausende der wichtigsten praktischen Probleme der Informatik sehr effizient lösen lassen oder nicht. Offenbar ist dies eine auch vom praktischen Standpunkt aus eine äußerst wichtige Frage – nach allem, was wir wissen, könnte es sein, dass es für all diese Probleme hocheffiziente Algorithmen gibt, nur es hat sie halt noch niemand gefunden.

21-1

Kapitel 21
Die O-Notation
David gegen Goliath

21-2

Lernziele dieses Kapitels

1. Definition verschiedener Varianten der O-Notation verstehen
2. Funktionen anhand ihrer O-Klassen klassifizieren können
3. Wichtige O-Klassen ihr Verhältnis kennen

Inhalte dieses Kapitels

Worum es heute geht

Herr Winston Wolf, der Ihnen vielleicht noch aus »Logik für Informatiker« bekannte Leiter des Amts für Problemlösungen, bekommt an einem Montag morgens ein neues Problem auf seinen Aktenstapel: Die Regierung bittet darum, das Problem mit den ungenauen Klimamodellen doch bitte zu lösen – man hätte gerne verlässlichere Aussagen über die Entwicklung des Weltklimas in den nächsten hundert Jahren.

Herr Wolf ruft daraufhin beim Amt für Klimaforschung an, wo man ihm sinngemäß Folgendes mittelt: »Unsere Prognosen werden um so besser, je höher die räumliche Auflösung der Simulationen ist. Dazu brauchen wir mehr Rechenpower.« In der Informatikabteilung seines Amtes arbeiten drei Informatikerinnen – eine ist zuständig für Technische Informatik, eine für Praktische Informatik und eine für Theoretische Informatik –, die Herrn Wolf nun befragt, wie er für mehr Rechenpower sorgen könne.

Dr. Naehle, die Referentin für Technische Informatik, antwortet: »Die Sache ist ganz klar: Wir benötigen jede Menge FPGAS, die wir mit Klimasimulationsschaltungen flashen. Die schalten wir dann zusammen und bauen damit eine riesige dedizierte Klimasimulationsmaschine.« Dr. Fischerin, zuständig für die Praktische Informatiker, meint: »Das geht viel billiger und einfacher mit Standardkomponenten. Die könnten wir bei Ebay besorgen (wenn die Beschaffungsstelle mitspielt). Dann brauchen wir noch ein paar flotte Netzwerkswitches und mein geliebtes OBSCURE-BSD auf allen Rechnern. Das ergibt dann einen schönen Supercomputer.« Dr. Meischnuck, die Theoretikerin, hat den zweifelsohne billigsten Vorschlag: »Wir schauen erstmal, was überhaupt das Problem ist. Bei diesen Klimasimulationen geht es doch meist um Differentialgleichungen. Die bestehenden Simulationsprogramm benutzen vielleicht noch stures Iterieren zur Lösung. Da greifen wir doch mal in die algorithmische Trickkiste, da findet sich sicherlich etwas Schnelleres.«

Wenn Herr Wolf alle drei gewähren lassen würde, wer würde am Ende am schnellsten die gewünschten Simulationsergebnisse berechnen?

In diesem Kapitel geht es um darum, Werkzeuge zu entwickeln, um solche Fragen sinnvoll zu beantworten. Es geht um eine »Theorie der Geschwindigkeit von Algorithmen«. Ziele dieser Theorie sind:

1. Sie soll möglichst unabhängig von konkreter Hardware sein. Dass ein Supercomputer Zahlen schneller sortiert als Ihr Telefon, ist wenig überraschend und sagt auch wenig darüber aus, wie schnell ein bestimmter Algorithmus ist.
2. Sie soll ein allgemeines Bild von der Geschwindigkeit eines Algorithmus liefern. Dass ein Algorithmus auf bestimmten Spezialfällen sehr gut funktioniert, ist zwar schön für den Algorithmus und man kann ihn dafür auch belobigen, jedoch sagt dies wiederum wenig über sein allgemeines Verhalten aus.

Es hat sich herausgestellt, dass die *O-Notation* besonders geeignet ist, die Geschwindigkeit von Algorithmus zu beschreiben. Grob gesprochen bedeutet »der Algorithmus hat Laufzeit $O(n^2)$«, dass er bei Eingaben der Größe n grob n^2 Rechenschritte benötigt *unabhängig von der Hardware*.

Welche der Informatikerinnen am Ende die besten Ergebnisse präsentieren kann, ist übrigens alles andere als klar: Wenn die Theoretikerin tatsächlich einen Algorithmus findet in einer kleineren O-Klasse, so wird sie damit die Superrechner der anderen beiden locker um Größenordnungen schlagen. Andererseits kann es aber bei Theoretikern leicht passieren, dass sie auch nach Jahren noch in ihrem Büro über einen neuen Algorithmus brüten, während die Computer der Techniker und Praktiker das Problem schon längst zur allgemeinen Zufriedenheit gelöst haben.

21.1 Einleitung

21.1.1 Welches Verfahren ist schneller?

21-4

Wie kann man die Rechenzeit von Algorithmen vergleichen?

Fragestellung
Sollte man lieber Insertion-Sort oder Merge-Sort benutzen?

Laufzeitmessungen ergeben folgende Werte:

n sortierte Zahlen	Insertion-Sort	Merge-Sort
$n = 3$	$0{,}045\mu s$	$0{,}245\mu s$
$n = 10$	$0{,}080\mu s$	$0{,}830\mu s$
$n = 1000$	$10\mu s$	$150\mu s$
$n = 100.000$	$1.000\mu s$	$33.000\mu s$

n zufällige Zahlen	Insertion-Sort	Merge-Sort
$n = 3$	$0{,}056\mu s$	$0{,}183\mu s$
$n = 10$	$0{,}360\mu s$	$1{,}080\mu s$
$n = 1000$	$1.250\mu s$	$190\mu s$
$n = 100.000$	$12.122.000\mu s$	$35.000\mu s$

✎ **Zur Diskussion**
Was ist nun besser – Insertion-Sort oder Merge-Sort?

Moral

1. Ein guter Algorithmus schlägt einen schlechten Algorithmus, selbst wenn der gute Algorithmus schlecht implementiert wird und der schlechte gut implementiert wird.
2. Die Terme n und $\log n$ in Laufzeiten sind *in der Regel* wichtiger als
   - die verwendete Hardware,
   - das Geschick der Programmierer.

21.1.2 Laufzeitmessung

Wir analysieren die Laufzeit eines einfachen Programms. 21-5

```
static boolean containsZero (String s) {
  for (int i = 0; i < s.length(); i++) {
    if (s.charAt(i) == '0') {
      return true;
    }
  }
  return false;
}
```

Die verschiedenen Befehle benötigen unterschiedlich lange.

- Seien t_1 bis t_6 die Zeiten, die benötigt werden 1) für eine Zuweisung, 2) für einen Vergleich, 3) von der Methode `length`, 4) für die Berechnung von `i++`, 5) von der Methode `charAt` und 6) vom `return`-Befehl.
- Die Laufzeit bei Eingabe `ABCD05` ist $t_1 + 10t_2 + 5t_3 + 4t_4 + 5t_5 + t_6$.
- Die Laufzeit bei Eingabe `ABCDEF` ist $t_1 + 13t_2 + 7t_3 + 6t_4 + 6t_5 + t_6$.
- Allgemein ist die Laufzeit bei Strings der Länge höchstens ℓ höchstens $t_1 + (2\ell+1)t_2 + (\ell+1)t_3 + \ell t_4 + \ell t_5 + t_6$.

Wie genau sollten wir Laufzeiten messen? 21-6

- Die Laufzeit das Programm lautete $t_1 + (2\ell+1)t_2 + (\ell+1)t_3 + \ell t_4 + \ell t_5 + t_6$.
- Die *genauen Werte* der Konstanten sind *eher unerheblich*, ja sogar *hinderlich für die Analyse*.
- Wir führen deshalb eine spezielle Notation ein, mit der wir uns *auf das Wesentliche konzentrieren* können: Die *O-Klassen*.

Die zentralen Ideen

1. Wir ignorieren *Konstanten*, mit denen *multipliziert wird*.
2. Uns interessiert nur Laufzeiten bei *großen Eingaben*.

21.2 O-Klassen und ihre Verwandten

21.2.1 O-Klassen

O-Klassen = höchstens. 21-7

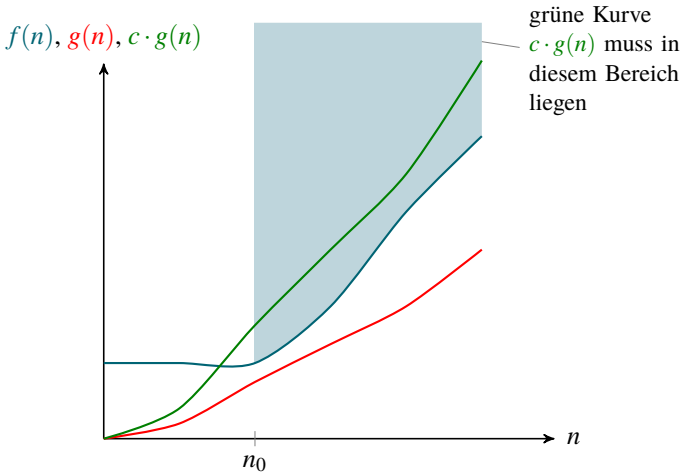
▶ Definition: Groß-O-Klasse
Sei $g\colon \mathbb{N} \to \mathbb{R}_0^+$ eine Funktion. Dann ist die O-Klasse $O(g)$ die Menge aller Funktionen $f\colon \mathbb{N} \to \mathbb{R}_0^+$, für die

- es eine Konstante c gibt und
- es eine Konstante n_0 gibt, so dass
- für alle $n > n_0$ gilt $f(n) \leq c \cdot g(n)$.

Merke
»$f \in O(g)$« bedeutet, dass für *genügend große* n und einen *genügend großen Faktor* c gilt, dass $c \cdot g(n)$ immer größer als $f(n)$ ist.

21-8

Veranschaulichung von $f \in O(g)$



21-9

Beispiele zu O-Klassen.

Beispiel
Sei $f(n) = 3 + 17n^2$ und $g(n) = n^2$. Dann ist $f \in O(g)$.
(Wähle $c = 1000$ und $n_0 = 1000$. Dann ist sicherlich $3 + 17n^2 \leq cn^2$.)

Beispiel
Sei $g(n) = 1$ für alle n. Dann enthält die Klasse $O(g)$ alle beschränkten Funktionen.

Beispiel
Sei $f(n) = n^2$ und $g(n) = n$. Dann ist $f \notin O(g)$.

Beispiel
Sei $f(n) = \log_2(n^2)$ und $g(n) = \log_2 n$. Dann ist $f \in O(g)$.
(Es gilt $\log_2(n^2) = 2\log_2 n$.)

21-10

Zur Schreibweise von O-Klassen.

- Man schreibt einfach $O(n^2)$ für »$O(g)$, wobei g die Funktion $g(n) = n^2$ ist«.
- Man schreibt auch $O(g)$, wenn g von mehreren Parametern abhängt.
  So bedeutet $O(n^2m)$ »die Laufzeit ist für hinreichend große n und m höchstens cn^2m für eine Konstante c«.
- Man schreibt auch $f(n) = O(n^2)$ statt $f \in O(n^2)$.
- Man schreibt auch Dinge wie

  $$2^{O(n)}$$

  und meint damit eigentlich »Eine Funktion, die für hinreichend große n und eine hinreichend große Konstante c durch $2^{c \cdot n}$ nach oben beschränkt ist.«
- *Das ist mathematisch alles Quatsch – aber das stört niemanden.*

21-11

Die O-Notation in der Praxis.

```
boolean containsZero (String s) {
  for (int i = 0; i < s.length(); i++) {
    if (s.charAt(i) == '0') {
      return true;
    }
  }
  return false;
}
```

- Es galt $T(\ell) = t_1 + (2\ell + 1)t_2 + (\ell + 1)t_3 + \ell t_4 + \ell t_5 + t_6$, wobei $T(\ell)$ die maximale Laufzeit bei Strings der Länge ℓ ist.
- Mit Hilfe der O-Notation können wir kurz schreiben $T \in O(\ell)$ oder auch $T(n) = O(n)$.
- Der Aufwand ist also *linear*.

✎ Zur Übung 21-12

Geben Sie für folgende f und g an, ob $f \in O(g)$ gilt.

1. $f(n) = \sqrt{n}$ und $g(n) = n$.
2. $f(n) = n^{4,00001}$ und $g(n) = n^4$.
3. $f(n) = n^5$ und $g(n) = n^4(\log n)^7$.
4. $f(n) = \log_2 n$ und $g(n) = 1$.
5. $f(n) = \log_2 n$ und $g(n) = \log_3 n$.
6. $f(n) = \log_3 n$ und $g(n) = \log_2 n$.
7. $f(n) = 2^n$ und $g(n) = 3^n$.
8. $f(n) = 3^n$ und $g(n) = 2^n$.

21.2.2 Ω-Klassen

O-Klassen = höchstens. Ω-Klassen = mindestens. 21-13

- Eine O-Klasse enthält alle Funktionen, die *höchstens* soundso schnell wachsen.
- Eine Ω-Klasse enthält alle Funktionen, die *mindestens* soundso schnell wachsen.

▶ Definition: Groß-Omega-Klasse

Sei $g\colon \mathbb{N} \to \mathbb{R}_0^+$ eine Funktion. Dann ist die Ω-Klasse $\Omega(g)$ die Menge aller Funktionen $f\colon \mathbb{N} \to \mathbb{R}_0^+$, für die

- es eine Konstante $\varepsilon > 0$ gibt und
- es eine Konstante n_0 gibt, so dass
- für alle $n > n_0$ gilt $f(n) \geq \varepsilon \cdot g(n)$.

Offenbar gilt

$$f \in O(g) \iff g \in \Omega(f).$$

Die Ω-Notation in der Praxis. 21-14

Beispiel: Maximumsbestimmung

- Ein Programm soll das Maximum einer Liste von Zahlen bestimmen.
- Dann muss es *jede Zahl mindestens einmal untersuchen* (sonst könnte man nämlich eine Eingabe bauen, bei der das Programm eine falsche Ausgabe macht).
- Der Aufwand des Programms ist also *mindestens linear*, also $\Omega(n)$.

Beispiel: Sortieren

- Ein Programm soll eine Liste von Objekten sortieren.
- Man kann zeigen, dass man hierzu mindestens $n \log_2 \frac{n}{e}$ Vergleiche benötigt.
- Der Aufwand eines solchen Programms ist also $\Omega(n \log n)$.

21.2.3 Θ-Klassen

O-Klassen = höchstens. Ω-Klassen = mindestens. Θ-Klassen = genau. 21-15

▶ Definition: Theta-Klassen

$\Theta(g) = O(g) \cap \Omega(g)$.

Merke: Die Theta-Klasse von g enthält alle Funktionen, die *bis auf einen Faktoren* genauso schnell wachsen wie g.

Beispiele

Die Laufzeit von Merge-Sort ist $\Theta(n \log n)$, denn jeder allgemein Sortieralgorithmus braucht $\Omega(n \log n)$ Zeit und er braucht aber auch nur $O(n \log n)$ Zeit.

Beispiele

Die Laufzeit von Bubble-Sort, Selection-Sort und Insertion-Sort ist $\Theta(n^2)$.

21-16

✎ Zur Übung

1. Bestimmen Sie die Θ-Klasse des Zeitaufwands von folgendem Programm:

```
void upper_one (double[][] matrix) { // Matrixgröße ist n x n
  int sum = 0;
  for (int i = 0; i < matrix.length; i++)
    for (int j = i; j <matrix[i].length; j++)
      matrix [i][j] = 0;
}
```

2. Bestimmen Sie die Θ-Klasse des Zeitaufwands von folgendem Programm:

```
double puzzle (double[] vector) { // Vektorlänge ist n
  int i = 1;
  while (i < vector.length / 3) {
    i = i*3;
  }
  return vector[i/3];
}
```

21.2.4 o-Klassen

21-17

O-Klassen = kleiner gleich. o-Klassen = echt kleiner.

- Betrachten wir Bubble-Sort versus Merge-Sort, sie haben die Laufzeiten $\Theta(n^2)$ und $\Theta(n \log n)$.
- Dann gilt insbesondere auch, dass *Merge-Sort eine Laufzeit $O(n^2)$ hat* – schließlich ist $n \log n \in O(n^2)$.
- Jedoch wäre es viel schöner, wenn man leicht ausdrücken könnte, dass *Merge-Sort eine echt bessere Laufzeit hat als n^2*.

▶ Definition: Klein-O-Klassen
$o(g) = O(g) \setminus \Theta(g)$.

Beispiel
Merge-Sort hat Laufzeit $o(n^2)$.

Beispiel
Man kann Matrizen multiplizieren in Zeit $o(n^3)$.

Beispiel
Für jede Funktion $f \in o(1)$ gilt $\lim_{n\to\infty} f(n) = 0$.

21.2.5 ω-Klassen

21-18

Ω-Klassen = größer gleich. ω-Klassen = echt größer.

▶ Definition: Klein-Omega-Klassen
$\omega(g) = \Omega(g) \setminus \Theta(g)$.

Beispiel
Bubble-Sort hat Laufzeit $\omega(n \log n)$.

Beispiel
Alle Funktionen in $\omega(1)$ sind unbeschränkt.

Zusammenfassung der Entsprechungen. 21-19

Sind f und g Funktionen, so entspricht *bis auf Faktoren und kleine Werte* grob:

$$\begin{aligned} f \in o(g) &\quad\approx\quad f < g, \\ f \in O(g) &\quad\approx\quad f \leq g, \\ f \in \Theta(g) &\quad\approx\quad f = g, \\ f \in \Omega(g) &\quad\approx\quad f \geq g, \\ f \in \omega(g) &\quad\approx\quad f > g. \end{aligned}$$

Zusammenfassung der wichtigsten O-Klassen. 21-20

Die wichtigsten O-Klassen und ihre Inklusionsbeziehungen:

$$\begin{aligned} O(1) &\subsetneq O(\log n) \subsetneq O(\sqrt{n}) \\ &\subsetneq O(n) \subsetneq O(n \log n) \subsetneq O(n \log^2 n) \\ &\subsetneq O(n^2) \subsetneq O(n^3) \\ &\subsetneq O(2^n) \subsetneq O(3^n). \end{aligned}$$

21.3 Wichtige Laufzeiten

21.3.1 Logarithmische Laufzeit

Logarithmische O-Klassen 21-21

- Beispiele von Algorithmen mit Laufzeit $O(\log n)$ sind:
  - Binäre Suche,
  - Präfix-Summe auf Parallelrechnern,
  - Maximumsbestimmung auf Parallelrechnern,
  - viele weitere Algorithmen auf Parallelrechnern.
- Es gilt:

$$\begin{aligned} & O(1) \\ \subsetneq\ & O(\log^* n) \\ \subsetneq\ & O(\log \log \log n) \\ \subsetneq\ & O(\log \log n) \\ \subsetneq\ & O(\log_{100} n) = O(\log_2 n) = O\big(\log_2(n^2)\big) \\ \subsetneq\ & O(\log^2 n) = O\big((\log n)^2\big) \\ \subsetneq\ & O(\log^3 n) \end{aligned}$$

In der Praxis: Logarithmisch = konstant. 21-22

- In der Praxis kommen Eingabegrößen von maximal einigen Terabyte oder in Zukunft vielleicht einiger Exabyte vor.
- Dann liefert $\log_2 n$ für $n = 10^{18}$ etwa 60.
- In der Praxis ist aber ein Faktor von 60 schon oft der Unterschied, ob der Compiler nun stark oder doch eher schwach optimiert.
- In der Praxis gilt deshalb »logarithmische Laufzeit = konstante Laufzeit«.

Diese Beobachtung hat sogar zu einer eigenen Notation geführt:

- Bei der *Tilde-Notation* wird eine Tilde zu den eingeführten Klassen geschrieben, aus $O(n)$ wird also $\tilde{O}(n)$ und aus $\Theta(n)$ wird $\tilde{\Theta}(n)$.
- Dies bedeutet dann grob »statt *konstanter* Faktoren ignoriere sogar alle *logarithmischen* Faktoren«.
- Es gilt dann $\tilde{O}(n) = \tilde{O}(n \log n) = \tilde{O}(n/\log n)$, aber immernoch $\tilde{O}(n) \subsetneq \tilde{O}(n^2)$.

21.3.2 Polynomielle Laufzeiten

21-23

Kubische Laufzeit ist das Maximum in der Praxis.

- Die meisten Algorithmen aus der Praxis haben eine *lineare* Laufzeit ($(O(n)$), eine *quadratische* Laufzeit ($O(n^2)$) oder eine *kubische* Laufzeit ($O(n^3)$).
- Beispiele sind:
  - Sortieren liegt in $O(n \log n) \subseteq O(n^2)$.
  - Matrix-Multiplikation liegt in $O(n^3)$.
  - Kürzeste-Wege-suchen liegt in $O(n^2)$.

In der Praxis

In der *Praxis* sagt man, dass $O(n^3)$ *die maximale noch gut nutzbare Laufzeit ist.*

In der Theorie

- Für die Theorie ist eine Grenze wie »kubische Lauzeit« etwas künstlich.
- Dort sagt man, dass *alle Laufzeiten der Form* $O(n)$, $O(n^2)$, $O(n^3)$, $O(n^4)$ *und so weiter* »noch gut sind«.
- Solche Laufzeiten nennt man auch *polynomielle Laufzeiten*.

21.3.3 Exponentielle Laufzeiten

21-24

Die Problematik exponentieller Laufzeit.

Beispiel: Das Partitionsproblem

Gegeben sind n Zahlen. Kann man sie so in zwei Teilmengen aufteilen, dass die Summen gleich sind?

- Es gibt 2^n Möglichkeiten, die Zahlen aufzuteilen.
- Dies liefert einen Algorithmus mit Laufzeit $O(2^n)$.
- Was ist die Laufzeit des *besten Algorithmus* für das Partition-Problem?

21-25

Wir mögen keine exponentielle Laufzeit.

Programm 2010

Programmiererin Ada hat den Algorithmus für das Partition-Problems in Maschinensprache implementiert. Auf einem Pentium 4GHz kann sie damit in einer Stunde Eingaben bis $n \approx \log_2(360 \cdot 10^9) \approx 38$ bearbeiten.

Programm 2020

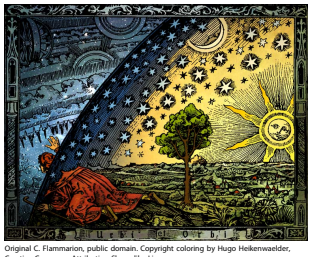
10 Jahre später gibt es einen 16-Core Rechner mit 10GHz. Nun kann sie in einer Stunde Eingaben bis $n \approx \log_2(4 \cdot 3600 \cdot 10^9) \approx 44$ bearbeiten.

Programm 2030

Wieder 10 Jahre später bekommt sie einen Supercomputer mit 10.000 Prozessoren mit je 1000GHz. Nun kann sie in einer Stunde Eingaben bis $n \approx \log_2(3600 \cdot 10^{15}) \approx 62$ bearbeiten.

21-26

Wir mögen wirklich keine exponentielle Laufzeit.



Original C. Flammarion, public domain. Copyright coloring by Hugo Heikenwaelder, Creative Commons Attribution Sharealike License

Programm 2500

Ada baut einen großen Rechner. Jedes Atom im Universum ($\approx 10^{80}$) rechnet in jeder Planck-Zeit ($\approx 5{,}39 \cdot 10^{-43}$s) einen Rechenschritt. Nun kann sie in einer Stunde Eingaben bis $n \approx \log_2(5{,}39 \cdot 3600 \cdot 10^{80+43}) \approx 423$ bearbeiten.

Moral

Eingaben mit 500 Zahlen wird man *niemals (!)* mit dem Algorithmus für das Partition-Problem lösen können.

Die Sicht der Exponentialzeitforscher.

21-27

- Untersucht man Exponentialzeit-Algorithmen, so ist es ganz wichtig, welche *Basis* der Exponent hat:
  - $O(1{,}001^n)$ ist praktisch gut handhabbar.
  - $O(100^n)$ ist völlig unbrauchbar.
- Den Exponentialzeitforschern ist es selbst egal, ob ein Algorithmus nun *Laufzeit* $O(n3^n)$ oder doch eher *Laufzeit* $O(n^2 3^n)$ hat – *wichtig ist allein die Basis »*3*« des Exponenten*.

Diese Beobachtung hat zu noch einer eigenen Notation geführt:

- Bei der *Sternchen-Notation* wird ein Sternchen zu den eingeführten Klassen geschrieben, aus $O(n)$ wird also $O^*(n)$ und aus $\Theta(n)$ wird $\Theta^*(n)$.
- Dies bedeutet dann grob »statt *konstanter* Faktoren ignoriere sogar alle *polynomiellen* Faktoren«.
- Es gilt dann $O^*(2^n) = O^*(n^5 2^n)$, aber immernoch $O^*(2^n) \subsetneq O^*(3^n)$.

Zusammenfassung dieses Kapitels

21-28

1. Die O-Klasse von g enthält alle Funktionen, die *bis auf einen Faktor höchstens so schnell wachsen wie* g.
2. Die Ω-Klassen funktionieren wie die O-Klassen, nur dass »höchstens« durch »mindestens« ersetzt wird.
3. Θ-Klassen sind der Schnitt von O-Klassen und Θ-Klassen und bedeuten »bis auf Faktoren genau soundso schnell«.
4. In der Praxis ist $O(n^3)$ die höchste noch tolerable Laufzeit bei größeren Eingaben ($n \geq 1000$).

Zum Weiterlesen

[1] R. Reischuk. *Unterlagen zum Lehrmodul Theoretische Informatik,* Vorlesungsskript, 2008. Kapitel 7.2.

Übungen zu diesem Kapitel

Übung 21.1 Funktionen nach Aufwandsklassen sortieren, mittel

Geben Sie für alle $i, j \in \{1, \ldots, 12\}$ an, ob $O(f_i) = O(f_j)$ oder $O(f_i) \subsetneq O(f_j)$ gilt. Begründen Sie Ihre Behauptung bei f_8, f_{11} und f_{12}.

1. $f_1(n) = \log_2 n$
2. $f_2(n) = n \log_2 n$
3. $f_3(n) = (\log_2 n)^2$
4. $f_4(n) = n^{5{,}00001}$
5. $f_5(n) = n^5$
6. $f_6(n) = 1{,}5^n$
7. $f_7(n) = 2^n$
8. $f_8(n) = 21(2^{\log_2 n})^{5^{\log_5(n)}}$
9. $f_9(n) = n^{4^{\log_2 n}}$
10. $f_{10}(n) = \Phi(198745627364563, 192387546732, 2)$
11. $f_{11}(n) = \sum_{i=1}^{n} \frac{i}{8}$
12. $f_{12}(n) = \sum_{i=1}^{n} \left(\frac{1}{2}\right)^i$

Übung 21.2 Funktionen nach Aufwandsklassen sortieren, mittel

Geben Sie für folgende f und g an, ob $f \in O(g)$, $f \in o(g)$ oder $f \in \Omega(g)$ gilt.

1. $f(n) = \log_2 n$ und $g(n) = n$.
2. $f(n) = n^5$ und $g(n) = n^4 \log_3 n$.
3. $f(n) = n^7$ und $g(n) = n^6 3^{\log_3 n} \sum_{i=i}^{n} \frac{i}{4}$.
4. $f(n) = \log_2(\log_3 n)$ und $g(n) = 2^{6348713389435284637}$.
5. $f(n) = n^4$ und $g(n) = \frac{1}{565465262} n^4 + n^3 \log_2 n + \sum_{i=1}^{n} \frac{i}{2} \log_2 n$.

22-1

Kapitel 22

Einführung zur Komplexitätstheorie

Die drei wichtigsten Maße: Zeit, Zeit und Zeit

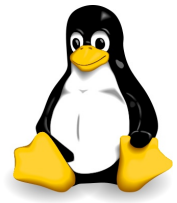
22-2

Lernziele dieses Kapitels

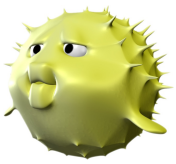
1. Konzept der zeit- und platzbeschränkten Turingmaschine verstehen
2. Wichtige deterministische Zeit- und Platzklassen kennen
3. Sprachen in die Zeit-Platz-Hierarchie einordnen können

Inhalte dieses Kapitels

Worum es heute geht



Authors Larry Ewing, Simon Budig, Anja Gerwinski, public domain



Unknown author, Creative Commons Attribution Lizenz

Unknown author, Lesser GNU Public Licence

Pearson Foresman, public domain

Beginnen wir die Komplexitätstheorie doch mit einer Fabel für kleine Informatiker:

> Eines Tages, als Tux der Pinguin gerade auf dem Weg von seiner Lieblingseisscholle nach Hause war mit einem leckeren Hering unter der Flosse, traf er unterwegs zwei seiner Freunde: Puffy den Kugelfisch und Konqi den Drachen. »Hallo,« sagte Tux, »wie geht es euren Usern?« »Gut,« antworteten die beiden, »aber unsere User haben uns jedem ein Problem gegeben, bei dem du uns bitte helfen musst.«
>
> Die beiden reichten Tux ein Blatt Papier, das ganz vollgeschrieben war mit Zahlen wie 19, 0, 50, 42, 123, 78 und noch viele mehr. Puffy sagte nun: »Tux, ich soll möglichst wenige dieser Zahlen auswählen, so dass die Summe mindestens 1000 ergibt.« Tux dachte kurz nach und antwortete: »Na, das ist doch ganz einfach: Du sortierst die Zahlen und nimmst dann so lange die größten, bis du 1000 zusammenhast.«
>
> Nun zeigte Konqi sein Blatt und sagte: »Tux, ich soll auch möglichst wenige dieser Zahlen auswählen, aber die Summe soll *genau* 1000 ergeben.« Tux rieb sich mit seiner Flosse den Kopf, dachte etwas nach und meinte dann: »Das scheint mir schwieriger zu sein. Da fragen wir doch mal den Biber.«
>
> So gingen sie zum Biber, der sich das Problem von Konqi anschaute und dann verkündete: »Das ist ein Problem nach meinem Geschmack! Mit etwas Fleiß kann man doch einfach alle Möglichkeiten durchprobieren, wie die Zahlen auszusuchen sind. Und mit Fleiß kenne ich mich ziemlich gut aus.« So machte sich der Biber daran, alle Möglichkeiten durchzuprobieren.
>
> Und wenn er nicht gestorben ist, so probiert er noch heute.

Die Moral hiervon ist (jede Fabel hat ja eine Moral): *Niemand weiß, wie man Konqis Problem effizient löst.* Die Probleme von Puffy und Konqi sehen fast identisch aus, haben aber – nach allem was man weiß – sehr unterschiedliche Komplexität.

In diesem Kapitel wird damit begonnen, eine Theorie des »Messens von Komplexität« zu entwickeln. Das wesentliche Ziel dieser Theorie ist zu verstehen, was Puffys Problem einfach macht, Konqis hingegen schwierig.

22.1 Ziele der Komplexitätstheorie

Komplexitätstheorie versus Biologie. 22-4
Die Komplexitätstheorie hat recht viel mit der Biologie gemein:
Die Biologie …

- untersucht Lebewesen,
- misst deren Eigenschaften: Anzahl Gliedmaßen, Anzahl Augen, Sozialverhalten, …
- klassifiziert sie in Spezies.

Die Komplexitätstheorie …

- untersuche Probleme,
- misst deren Eigenschaften: Zeitkomplexität, Platzkomplexität, Parallelisierbarkeit, …
- klassifiziert sie in Komplexitätsklassen.

22.1.1 Messen der Komplexität

Erstes Ziel: Messen der Komplexität. 22-5
Die Komplexität eines Problems (einer Sprache) kann man *messen*.

Beispiele von Komplexitätsmaßen

- *Wie viel Zeit braucht man, das Problem zu lösen?*
- *Wie viel Platz braucht man, das Problem zu lösen?*
- Wie gut lässt sich das Problem parallelisieren?

Die Messung der Komplexität erlaubt es uns…

- *vorherzusagen*, welche Eingaben wir später realistischerweise werden verarbeiten können;
- *zu verstehen*, wie Lösungsalgorithmen eventuell verbessert werden könnten;
- *zu verstehen*, wieso wir es nicht schaffen, bessere Algorithmen für das Problem zu finden.

22.1.2 Klassifizieren der Komplexität

Zweites Ziel: Klassifizieren der Komplexität. 22-6
Probleme (also Sprachen) kann man *gruppieren* (=klassifizieren) entsprechend ihrer Komplexität:

Beispiele von Klassifikationen

- Zeitklassen wie P, NP, EXP.
- Platzklassen wie L, PSPACE.
- Exotische Klassen wie $\mathrm{ZPP}^{\mathrm{AM}[\log]}/\mathrm{poly}$.

Die Klassifikation der Komplexität erlaubt es uns…

- Eigenschaften für alle Probleme einer Klasse auf einmal zu beweisen;
- einfacher über Arten und Abstufungen von Komplexitäten zu sprechen.

22.1.3 Neue algorithmische Methoden finden

22-7

Drittes Ziel: Neue Lösungsmethoden finden

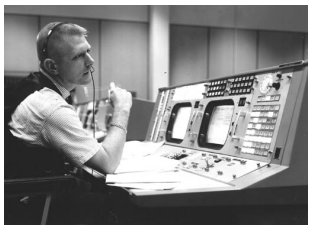
Die Ergebnisse der Theorie zeigen idealerweise *neue algorithmische Ideen* auf.

Beispiele neuer algorithmischer Ideen

- Fixed-Parameter-Algorithmen
- Quanten-Algorithmen
- Randomisierte Algorithmen

22-8

Gene Kranz: »Failure is not an option.«



Copyright by NASA.

Wiese hilft die Komplexitätstheorie, neue Algorithmen zu finden?

- Sie kann vorhersagen, dass bestimmte Methoden nicht helfen werden, ein Problem zu lösen.
- Dann können wir uns auf *andere Methoden konzentrieren*.
- Sie kann uns sagen, was bei ähnlichen Probleme erfolgreich war.

22.2 Zeitkomplexität

22.2.1 Definition

22-9

Das wichtigste Komplexitätsmaß: Die Zeit.

Zeit misst, wie lange wir warten müssen, bis ein Algorithmus eine Eingabe entschieden hat.

Unknown author, public domain

Warum ist »Zeit« so wichtig?

- Menschen sind ungeduldig.
- Time is money.
- Ein Problem in 10^{10000} Jahren lösen zu können, ist nicht dasselbe, wie ein Problem lösen zu können.

22-10

Definition von »Zeitaufwand« bei Turing-Maschinen.

▶ **Definition:** Zeitaufwand einer DTM

Sei M einer DTM mit Eingabealphabet Σ. Wir definieren eine *Funktion* $t_M : \Sigma^* \to \mathbb{N} \cup \{\infty\}$ wie folgt:

$$t_M(x) = \begin{cases} s, & \text{falls eine Folge } C_{\text{init}}(x) \vdash C_1 \vdash \cdots \vdash C_s \\ & \text{existiert, in der } C_s \text{ Endkonfiguration ist,} \\ \infty, & \text{sonst.} \end{cases}$$

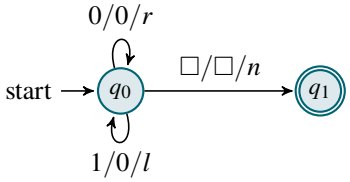
Wir definieren $T_M : \mathbb{N} \to \mathbb{N} \cup \{\infty\}$ durch

$$T_M(n) = \max_{x \in \Sigma^n} t_M(x).$$

22-11

Beispiel des Zeitaufwands einer Maschine

Sei M folgende Maschine:



- $t_M(00) = 3$, da die Berechnung lautet

$$\begin{aligned}(q_0, \cdots \triangleright 00 \cdots) &\vdash (q_0, \cdots 0 \triangleright 0 \cdots) \\ &\vdash (q_0, \cdots 00 \triangleright \square \cdots) \\ &\vdash (q_1, \cdots 00 \triangleright \square \cdots)\end{aligned}$$

– $t_M(01) = 5$, da die Berechnung lautet

$$\begin{aligned}(q_0, \cdots \triangleright 01 \cdots) &\vdash (q_0, \cdots 0 \triangleright 1 \cdots)\\ &\vdash (q_0, \cdots \triangleright 00 \cdots)\\ &\vdash (q_0, \cdots 0 \triangleright 0 \cdots)\\ &\vdash (q_0, \cdots 00 \triangleright \square \cdots)\\ &\vdash (q_1, \cdots 00 \triangleright \square \cdots)\end{aligned}$$

– Für den Zeitaufwand bei Worten der Länge n gilt

$$T_M(n) = \begin{cases} 1, & \text{für } n = 0,\\ 3n-1, & \text{für } n > 0,\end{cases}$$

da sie im schlimmsten Fall, nämlich bei Wörtern der Form 01^{n-1}, gerade $3n-1$ Schritte macht.
– Da $T_M \in O(n)$, sagen wir auch, *die Maschine habe linearen Aufwand.*

22.2.2 Klassen

Klassifikation von Problemen nach ihrer Zeitkomplexität. 22-12

Die Idee

– Eine *Zeitklasse* versammelt alle Sprachen, die innerhalb einer bestimmten Zeitschranke gelöst werden können.
– Die Zeitschranke kann entweder eine *ganz bestimmte Funktion* sein (wie »$f(n) = 3n$«) oder eine *ganze Klasse von Funktionen* (wie »Polynome«).
– Das Wort »Klasse« wird hier also in zwei Bedeutungen benutzt (Klasse von Funktionen und Klasse von Sprachen). Diese bitte nicht verwechseln.

▶ **Definition:** Zeitklassen für Sprachen
Sei $\mathcal{F} \subseteq \{f \mid f\colon \mathbb{N} \to \mathbb{N}\}$ eine Klasse von Zeitschranken. Dann ist $\text{TIME}(\mathcal{F})$ die Menge aller Sprachen L, so dass eine DTM M und ein $f \in \mathcal{F}$ existieren mit

1. $L(M) = L$ und
2. für alle $n \in \mathbb{N}$ gilt $T_M(n) \leq f(n)$.

Wichtige Zeitklassen 22-13

▶ **Definition**
Im Folgenden möge p für »beliebige Polynome« stehen, also $p(n) = n^{O(1)}$.

$$\begin{aligned}\text{LINTIME} &= \text{TIME}\big(O(n)\big),\\ \text{P} &= \text{TIME}\big(p(n)\big),\\ \text{EXP} &= \text{TIME}\big(2^{p(n)}\big),\\ \text{EEXP} &= \text{TIME}\big(2^{2^{p(n)}}\big),\\ &\vdots\\ \text{ELEMENTARY} &= \text{TIME}(2^{2^{2^{\cdot^{\cdot^{\cdot^{2^n}}}}}}).\end{aligned}$$

22.2.3 Beispiele

22-14

Beispiele von Sprachen in den Zeitklassen.

Beispiel

PARITY $= \{w \in \{0,1\}^* \mid$ die Anzahl an 1en in w ist ungerade$\}$ liegt in der Klasse LINTIME, also

$$\text{PARITY} \in \text{LINTIME}.$$

Beispiel

Sei CONNECTED folgende Sprache: Sie enthält (die Codes von) allen ungerichteten Graphen G, so dass G zusammenhängend ist (es gibt einen Weg von jedem Knoten zu jedem anderen). Dann gilt

$$\text{CONNECTED} \in \text{P}.$$

Dies liegt daran, dass der Zusammenhang eines Graphen beispielsweise mittels Breitensuche in Zeit von grob $O(n^2)$ überprüft werden kann.

Beispiel

Sei SAT folgende Sprache: Sie enthält (die Codes von) allen aussagenlogischen Formeln, die erfüllbar sind; also beispielsweise $(p \vee q) \in$ SAT, aber $(p \wedge \neg p) \notin$ SAT. Dann gilt

$$\text{SAT} \in \text{EXP}.$$

- Dies liegt daran, dass man bei einer Formel mit k Variablen alle 2^k möglichen Belegungen durchprobieren kann.
- Das Überprüfen einer Belegung dauert grob $O(n^2)$, wenn n die Länge der Formel ist.
- Insgesamt kann man also eine Turingmaschine bauen, deren Laufzeit grob $O(2^n n^2) \subseteq O(3^n)$ ist. Also liegt die Sprache in der Klasse EXP.

22-15

✎ Zur Übung

Ordnen Sie die folgenden Probleme in eine möglichst kleine Zeitklasse ein:

1. $\{0^n 1^n \mid n \geq 1\}$,
2. TAUTOLOGIES (die Sprache, die die Codes aller aussagenlogischen Formeln enthält, die Tautologien sind),
3. PALINDROMES.

22.3 Platzkomplexität

22.3.1 Definition

22-16

Ein weniger wichtiges Maß: Platz

Platz misst den *Speicherverbrauch* einer Berechnung.

Wieso ist Platz weniger wichtig als Zeit?

- Speicherplatz ist billig.
- In der Praxis geht uns der Platz *nie* aus, bevor uns die Zeit ausgeht.
- Sublinearer Platz ist unrealistisch: Wir haben *immer* mindestens so viel Platz, wie die Eingabe lang ist.

Warum wird Platz trotzdem untersucht?

- *Manche* Algorithmen benötigen *doch* irrsinnig viel Platz, wenn sie nämlich *kontinuierlich Speicher reservieren*.
- »Lässt sich mit wenig Platz lösen« = »Ist gut parallelisierbar«.
- Algorithmen, die auf dem Internet-Graphen arbeiten, können *nichtmal ihre Eingabe* (nämlich den Internet-Graphen) komplett speichern.

Interludium: Eingabe- und Ausgabebänder

22-17

✎ Zur Diskussion

Wie viel Speicherplatz verbraucht ein endlicher Automat?

- Es ist *unfair*, wenn die Eingabe einer Maschine ihrem Platzverbrauch zugerechnet wird.
- Andererseits: Wenn eine Maschine ihr Eingabeband für Berechnungen benutzt (man spricht vornehm von »kanibalisieren«), dann verbraucht sie ja doch Speicherplatz.

▶ Definition: Read-Only- und Write-Only-Bänder

- Ein *Read-Only-Band* wird von einer Maschine in keiner Berechnung verändert.
- Ein *Write-Only-Band* wird von einer Maschine beschrieben; ihr Verhalten hängt aber in keiner Weise davon ab, was dort steht.

Analogie

- Read-Only-Bänder sind wie CDROMS – man kann sie lesen, aber nicht ändern.
- Write-Only-Bänder sind wie Radiosendungen – man kann sie ausstrahlen (»schreiben), aber nicht nachschauen, was eben gesendet wurde.

Vereinbarung

- Ab sofort muss das Eingabeband *immer ein Read-Only-Band* sein.
- Ab sofort muss das Ausgabeband (falls vorhanden) *immer ein Write-Only-Band* sein.

Das zweitwichtigste Komplexitätsmaß: Der Platz.

22-18

▶ Definition: Platzverbrauch einer Berechnung

Sei $C_1 \vdash_M C_2 \vdash_M \dots$ eine Berechnung einer DTM M. Sei h_i^t jeweils die Kopfposition auf Band t in Konfiguration C_i.

1. Der *Platzverbrauch der Berechnung auf Band t* ist
$$1 + \max\{h_1^t, h_2^t, \dots\} - \min\{h_1^t, h_2^t, \dots\}.$$
2. Der *Platzverbrauch der Berechnung* ist die Summe des Platzverbrauchs über alle *Arbeitsbänder*.

Merke

Die Eingabe- und Ausgabebänder zählen *nicht* mit, wenn der Platzverbrauch bestimmt wird.

▶ Definition: Platzverbrauch einer DTM

Sei M eine DTM mit Eingabealphabet Σ^*. Wir definieren $s_M : \Sigma^* \to \mathbb{N} \cup \{\infty\}$ durch

$$s_M(x) = \text{Platzverbrauch der mit } C_{\text{init}}(x) \text{ beginnenden Berechnung.}$$

Wir definieren $S_M : \mathbb{N} \to \mathbb{N} \cup \{\infty\}$ als

$$S_M(n) = \max_{x \in \Sigma^n} s_M(x).$$

22.3.2 Klassen

22-19 Klassifikation von Problemen bezüglich des Platzverbrauchs.

▶ Definition: Platzklassen für Sprachen
Sei $\mathcal{F} \subseteq \{f \mid f\colon \mathbb{N} \to \mathbb{N}\}$ eine Klasse von Platzschranken. Die Klasse $\text{SPACE}(\mathcal{F})$ ist die Menge aller Sprachen L, so dass eine DTM M und ein $f \in \mathcal{F}$ existieren mit

1. $L(M) = L$ und
2. für alle $n \in \mathbb{N}$ gilt $S_M(n) \leq f(n)$.

▶ Definition: Wichtige Platzklassen
Wieder möge p für beliebige Polynome stehen:

$$\begin{aligned} \text{L} &= \text{SPACE}\big(O(\log n)\big), \\ \text{LINSPACE} &= \text{SPACE}\big(O(n)\big), \\ \text{PSPACE} &= \text{SPACE}\big(p(n)\big), \\ \text{EXPSPACE} &= \text{SPACE}\big(2^{p(n)}\big). \end{aligned}$$

22-20 Eine Platzklasse kennen wir schon.

▶ Lemma
$\text{REG} = \text{SPACE}(0)$.

Beweis.

- Eine Turing-Maschine, die *überhaupt keinen zusätzlichen Platz verbraucht*, ist nichts anderes als ein *2-Wege-*DFA.
- Wir haben in Satz 7-22 gezeigt, dass solche Automaten genau die regulären Sprachen akzeptieren. □

22.3.3 Beispiele

22-21 Beispiele für Platzkomplexität.

Beispiel
Die Sprache $\{0^n 1^n \mid n \geq 1\}$ hat eine Platzkomplexität von $O(\log n)$:

- Eine DTM kann auf einem Arbeitsband für jede 0 einen Binärzähler hochzählen.
- Dann zählt sie ihn für jede 1 wieder herunter.
- Dann verbraucht sie nie mehr als $\log_2 n$ Zellen auf dem Arbeitsband.

Also gilt

$$\{0^n 1^n \mid n \geq 1\} \in \text{L}.$$

Beispiel
Die Sprache SAT kann in linearem Platz entschieden werden:

- Man probiert systematisch alle möglichen 2^k Belegungen der k Variablen durch.
- Hat man eine Belegung ausprobiert (und war sie nicht erfüllend), dann *kann man sie durch die nächste überschreiben*.
- Pro Belegung benötigt man nur k Bits an Platz, zuzüglich etwas Platz, um die Formel für die Belegung auszuwerten.

Also gilt

$$\text{SAT} \in \text{LINSPACE}.$$

22-22 ✎ Zur Übung
Wie viel Platz benötigt man, um die folgenden Sprachen zu entscheiden?

1. $\{0^n 1^m \mid n, m \geq 1\}$,
2. $\{0^n 10^n \mid n \geq 1\}$,
3. TAUTOLOGIES.

Wem das zu leicht ist, der versucht sich bitte an $\{\text{bin}(1)\#\text{bin}(2)\#\ldots\#\text{bin}(n) \mid n \in \mathbb{N}\}$.

Zusammenfassung dieses Kapitels

1. Die *Zeitkomplexität* eines Problems gibt an, *wie schnell* es sich lösen lässt. 22-23
2. Die *Platzkomplexität* eines Problems gibt an, wie viel Speicherplatz *zusätzlich zur Eingabe* benötigt wird.
3. Eine *Komplexitätsklasse* versammelt alle Probleme, die eine *bestimmte maximale Zeit- oder Platzkomplexität* haben.
4. Die wichtigsten Komplexitätsklassen sind L, P, PSPACE und EXP.

Zum Weiterlesen

[1] R. Reischuk. *Unterlagen zum Lehrmodul Theoretische Informatik,* Vorlesungsskript, 2008. Kapitel 4.5, 8.1.

Übungen zu diesem Kapitel

Übung 22.1 Aufwand von Programmen bestimmen, leicht

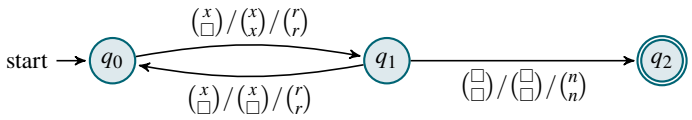
Bestimmen Sie die Θ-Klasse des Zeitaufwands von folgendem Programm:

```
double foo (double[] vector) { // Vektorlänge ist n
  double result = 0;
  for (int i = 0; i < vector.length; i++) {
    for (int j = 0; j < = i; j++){
      result = result + vector[j];
    }
  }
  return result;
}
```

Begründen Sie Ihre Antwort, indem Sie erklären, warum der Zeitaufwand mindestens und höchstens so groß ist wie von Ihnen angegeben.

Übung 22.2 Zeit- und Platzaufwand von DTM-s, mittel

Welche Sprache akzeptiert die folgende Maschine? Das Eingabealphabet ist $\Sigma = \{0,1\}$ und $x \in \{0,1\}$.



Geben Sie in beiden Fällen die Θ-Klasse für den Zeit- und Platzverbrauch Ihrer Maschinen für Eingaben $w \in \{0,1\}^n$.

Übung 22.3 Zeit- und Platzaufwand von DTMs, mittel

Es sei $L = \{ww^{\text{rev}} \mid w \in \{0,1\}^*\}$ gegeben.

1. Beschreiben Sie eine deterministische Turingmaschine für L mit einem Read-Only-Eingabeband und einem Arbeitsband, die möglichst wenig Schritte benötigt.
2. Beschreiben Sie eine deterministische Turingmaschine mit einem Read-Only-Eingabeband und einem Arbeitsband, die L akzeptiert und welche möglichst wenig Platz auf dem Arbeitsband braucht.

Geben Sie in beiden Fällen die Θ-Klasse für den Zeit- und Platzverbrauch Ihrer Maschinen für Eingaben $x \in \{0,1\}^n$.

Übung 22.4 Aufwand von Programmen bestimmen, mittel

Bestimmen Sie die Θ-Klasse des Zeitaufwands von folgendem Programm:

```
int foo (double[] vector, double wert) { // Vektorlänge ist n
  int ergebnis = vector.length + 1;
  int zufallszahl;
    for (int i = 0; i < vector.length; i++) {
       zufallszahl = myrandom.nextInt(vector.length);
       double temp = vector[i];
       vector[i] = vector[zufallszahl];
       vector[zufallszahl] = temp;
```

```
    }

    for (int i = 0; i < vector.length; i++){
        if(vector[i] == wert)
        ergebnis =  i;
    }
  return ergebnis;
}
```

Sie können dabei davon ausgehen, dass der Methodenaufruf `myrandom.nextInt(...)` konstante Zeit benötigt.

Übung 22.5 Zeit- und Platzaufwand von DTMs bestimmen, mittel

Es sei die Sprache $L = \{a^p b^p c^p \mid p \in \mathbb{N}\}$ gegeben.

1. Beschreiben Sie eine deterministische Turingmaschine für L mit einem Read-Only-Eingabeband und einem Arbeitsband, die möglichst wenig Schritte benötigt.
2. Beschreiben Sie eine deterministische Turingmaschine mit einem Read-Only-Eingabeband und einem Arbeitsband, die L akzeptiert und welche möglichst wenig Platz auf dem Arbeitsband braucht.

Geben Sie in beiden Fällen die Θ-Klasse für den Zeit- und Platzverbrauch Ihrer Maschinen für Eingaben $x \in \{0,1\}^n$.

23-1

Kapitel 23
Die Idee der Reduktion
Apfelstrudel $\leq_m^{\log}$ Birnenstrudel

Lernziele dieses Kapitels

1. Konzept der Reduktion allgemein verstehen
2. Definition der Logspace-Many-One-Reduktion verstehen
3. Reduktionen erstellen können

23-2

Inhalte dieses Kapitels

Worum es heute geht

Falls Sie jemals vorhaben, die Menschheit eines Tages durch Ihren Nachwuchs zu beglücken, so möchte ich Ihnen dringend davon abraten, Komplexitätstheoretiker nach Namensvorschlägen für Ihr Baby zu befragen. Die Namen, die sich Komplexitätstheoretiker ausdenken, sind – höflich formuliert – extrem verwirrlich. Im letzten Kapitel wurden ja schon einige Klassen eingeführt, deren Namen typisch sind: Warum heißt logarithmischer Platz L, polynomieller Platz hingegen PSPACE? Wieso nennt man Probleme, die sich praktisch gar nicht und selbst theoretisch nur mit exorbitantem Aufwand lösen lassen, »elementar«? Und überhaupt, warum nennt man Probleme eigentlich »Sprachen«? Gar nicht erwähnt habe ich vorsichtshalber, dass es neben der Klasse EXP auch die Klasse E gibt, die fast genauso definiert ist – nur eben ein ganz klein bisschen anders. Perfektioniert wird das Chaos dadurch, dass sich die Namen mit den Jahren auch gerne verlängern oder verkürzen, je nachdem was gerade so in Theoretikerkreisen als modisch und besonders hipp gilt. Aus dem Mauerblümchen PTIME wurde so das trendy P, die fette Klasse EXPTIME wurde zu E abgemagert – logisch wäre zwar EXP gewesen, aber das war ja schon anderweitig vergeben.

»Was soll's?«, mag man einwenden, andere Disziplinen wie die Biologie oder die Medizin sind auch voll von unsinnigen Namen und Abkürzungen. Sind aber die meisten Klassennamen einfach nur verwirrlich, so sind die drei Bezeichungen »Reduktion«, »Schwere« und »Vollständigkeit« für drei zentrale Ideen der Theorie katastrophale Fehlentscheidungen gewesen: Sie verwirren Anfänger oft dermaßen, dass sie noch während des Vorspiels die Lust an der Materie verlieren. Ein Reduktion macht ein Problem weder kleiner noch einfacher; ein schweres Problem ist nicht etwa schwer, sondern oft sehr leicht zu lösen; ein vollständiges Problem ist nicht das Gegenteil eines Teilproblems. Leider ist es zu spät, die unsinnigen Namen zu ändern, sie haben sich festgesetzt.

Wie würden bessere Bezeichungen lauten? Statt von einer »Reduktion zwischen Problemen« sollte man lieber von einem »Vergleich von Problemen« sprechen, denn das macht eine Reduktion in Wirklichkeit: Kann man A auf B reduzieren, dann bedeutet dies eigentlich nur »die Komplexität von A ist höchstens so groß wie die von B«. Statt »schwer für

eine Klasse« sollte man eher sprechen von »hilfreich für eine Klasse«: Wenn A schwer ist für eine Klasse, dann kann man alle Probleme in der Klasse lösen, wenn man nur A lösen kann. Schließlich sollte »vollständiges Problem für eine Klasse« am besten in »hilfreiches Problem in einer Klasse« umbenannt werden: Der einzige Unterschied zwischen schweren und vollständigen Problemen ist nämlich, dass letztere eben *in* der Klasse liegen müssen, erstere hingegen nicht unbedingt.

In diesem Kapitel werden wir zunächst Reduktionen genauer untersuchen, in darauffolgenden Kapitel geht es dann mit schweren und vollständigen Problemen weiter.

Bleibt die Frage, wen Sie um Rat fragen sollte bezüglich der Benennung Ihrer Kinder. Fragen Sie doch einen Mathematiker! Menschen, die »durch Distributivgesetze miteinander verschränkte Operationen mit zugrundeliegenden Gruppenstrukturen« schlicht »Ringe« nennen, scheinen mehr Gespür für die Schönheit von Worten zu haben. Und wenn dem Ring noch einige weitere Axiome wie Kommutativität der Gruppenstruktur oder Nullteilerfreiheit hinzugefügt werden, so nennen Mathematiker dies nicht etwa »durch Distributivgesetze miteinander verschränkte Operationen mit zugrundeliegenden kommutativen Gruppenstrukturen und Nullteilerfreiheit«, sondern etwas verträumt »Körper«.

23.1 Einführung zu Reduktionen

23.1.1 Kulinarische Vorbemerkungen

23-4

Vergleich der Komplexität von Gerichten.

- Eigentlich geht es heute um *Informatik-Probleme*,...
- ...aber zur Einstimmung ein paar *Koch-Probleme*.

Author Stara Blazkova, Creative Commons Attribution Sharealike Licence

Betrachten wir folgende Probleme:

- Einen Apfelstrudel backen.
- Einen Birnenstrudel backen.
- Einen *Crêpe suzette* flambieren.
- Ein *Ragout fin* kochen.

Eine kulinarische Fragestellung

Welche dieser Probleme haben »die gleiche Komplexität«?

23-5

Author David Monniaux, Creative Commons Attribution Sharealike Licence

Unser heutiges Ziel: Die Komplexität von Problemen vergleichen

Betrachten wir folgende (absichtlich nicht besonders formal beschriebene) vier Probleme:

1. Bestimme die Anzahl der erfüllenden Belegungen einer aussagenlogischen Formel φ.
2. Bestimme die Anzahl der erfüllenden Belegungen der Negation einer aussagenlogischen Formel φ.
3. Wähle möglichst wenige Zahlen aus einer Menge von Zahlen aus, so dass deren Summe mindestens 100 ist.
4. Wähle möglichst wenige Zahlen aus einer Menge von Zahlen aus, so dass deren Summe genau 100 ist.

Unknown author, Creative Commons Attribution Sharealike Licence

Eine informatorische Fragestellung

Welche dieser Probleme haben »die gleiche Komplexität«?

23-6

Eine erste Sackgasse auf dem Weg zu einer Definition von »Vergleichbarkeit«.

Will man die Komplexität von Problemen vergleich, so liegt es zunächst nahe zu überprüfen, *ob sie in denselben Komplexitätsklassen liegen*.
Das klappt aber nicht:

- Es dauert etwa gleich lange, einen Apfelstrudel und einen Braten herzustellen.
- Trotzdem scheinen diese Probleme wenig gemein zu haben.

Ähnlich liegen die Dinge bei folgenden Problemen:

- Die Probleme »Bestimme die Fourier-Transformation« und »Sortiere« benötigen beide Zeit $\Theta(n \log n)$.
- Trotzdem scheinen diese Problem wenig gemein zu haben.

Eine zweite Sackgasse.

23-7

Man könnte hoffen, dass Probleme nah verwandt sind, wenn sie *sehr ähnliche Formulierungen* haben.
Das klappt aber nicht:

- Die Probleme
  1. »Eine Fertig-Pizza backen« und
  2. »Eine feurige Pizza backen«

  sind *syntaktisch fast gleich*.
- Das erste Problem ist jedoch recht einfach, das zweite hingegen deutlich schwieriger.

Entsprechend:

- Die Probleme
  1. »Finde einen möglichst kurzen Pfad zwischen zwei Knoten in einem Graphen« und
  2. »Finde einen möglichst langen Pfad zwischen zwei Knoten in einem Graphen«.

  sind *syntaktisch fast gleich*.
- Das erste Problem ist jedoch ganz einfach, das zweite hingegen sehr schwierig.

23.1.2 Die Idee der Reduktion

Die Lösung: Vergleichbarkeit durch Reduktionen.

23-8

Betrachten wir folgende Probleme:

1. »Einen Apfelstrudel backen.«
2. »Einen Birnenstrudel backen«

- Nehmen wir an, Sie haben keine Ahnung, wie schwierig es ist, die Probleme zu lösen (weil Sie es noch nie getan haben).
- Jedoch können Sie wenigstens argumentieren, dass die Problem ähnlich schwierig sind, denn *jeder, der das eine Problem lösen kann, kann auch das andere lösen.*
- Dies funktioniert so:
  - Sie wollen einen Apfelstrudel backen.
  - Dann bitten Sie jemanden, für Sie stattdessen einen Birnenstrudel vorzubereiten.
  - Bevor der Birnenstrudel aber in den Ofen kommt, entfernen Sie die Birnen und ersetzen Sie diese durch Äpfel.
  - Dann lassen Sie den Strudel fertig backen.
- Man sagt in diesem Fall, dass sich das Apfelstrudel-Back-Problem auf das Birnenstrudel-Back-Problem *reduziert*.

Betrachten wir folgende Probleme:

1. Bestimme die Anzahl der Primteiler der Zahl n.
2. Bestimme die Anzahl der Primteiler der Zahl $2n$.

- Nehmen wir an, Sie haben keine Ahnung, wie schwierig es ist, die Probleme zu lösen.
- Jedoch können Sie wenigstens argumentieren, dass die Problem ähnlich schwierig sind, denn *jeder, der das eine Problem lösen kann, kann auch das andere lösen.*
- Dies funktioniert so:
  - Sie wollen die Anzahl der Primteiler von n bestimmen.
  - Dann bitten Sie jemanden, für Sie stattdessen die Anzahl der Primteiler von $2n$ zu ermitteln.
  - Ist n gerade, so geben sie die ermittelte Zahl aus, sonst geben sie die ermittelte Zahl minus Eins aus.
- Man sagt in diesem Fall, dass sich das erste Problem auf das zweite *reduziert*.

23-9

Problem, die »höchstens so schwierig wie« andere sind

- Betrachten wir zwei Probleme A und B, deren Komplexität *uns weitgehend unbekannt ist*.
- Nehmen wir nun aber kurzzeitig an, dass »irgend jemand schlaues« das Problem B lösen kann.
- Wenn wir dann auch ganz leicht das Problem A lösen können, dann sagen wir
  - »A ist höchstens so schwierig wie B« oder vornehmer
  - »A reduziert sich auf B«.

23-10

Auf dem Weg zu einer formalen Definition von Reduktionen.

Werden wir nun ein wenig formaler:

- »Probleme« sind ab sofort wieder Sprachen A und B.
- »Probleme lösen« bedeutet ab sofort wieder, die Frage »Ist $x \in A$?« beantworten zu können.
- Bei einer Reduktion wird die Frage »Ist $x \in A$?« *übersetzt* in die Frage »Ist $y \in B$?«.
- Dabei sollte es *möglichst einfach sein*, aus x das passende y zu errechnen.

Beispiel

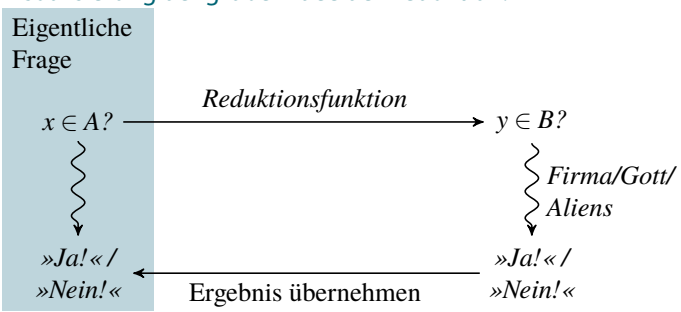
Wir reduzieren das Problem TAUTOLOGY auf das Problem CONTRADICTION:

- Aus der Frage »Ist $\varphi \in$ TAUTOLOGY?«…
- …wird die Frage »Ist $\neg\varphi \in$ CONTRADICTION?«.

Wir machen also aus x den Wert $y = \neg x$.

23-11

Visualisierung der groben Idee der Reduktion.



23.1.3 Beispiele von Reduktionen

23-12

Reduktion zwischen zwei Graphproblemen

▶ **Definition**

Sei $G = (V, E)$ ein Graph mit Knotenmenge V und Kantenmenge E. Eine Teilmenge $X \subseteq V$ der Knotenmenge nennt man

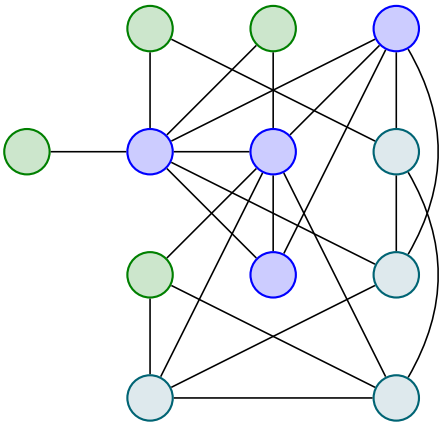
- eine *Clique*, wenn je zwei Knoten in X durch eine Kante miteinander verbunden sind;
- eine *unabhängige Menge*, wenn keine zwei Knoten in X durch eine Kante direkt verbunden sind.

▶ **Definition**

$$\begin{aligned}\text{CLIQUE} &= \{(\text{bin}(k), \text{code}(G)) \mid \\ &\quad G \text{ enthält eine Clique der Größe mindestens } k\}, \\ \text{INDEPENDENT-SET} &= \{(\text{bin}(k), \text{code}(G)) \mid \\ &\quad G \text{ enthält eine unabhängige Menge der Größe mindestens } k\}.\end{aligned}$$

Beispiel

Eine Graph mit einer Clique (in blau) und einer unabhängige Menge (in grün) jeweils der Größe 4. Damit ist (der Code von) dem Paar $(4, G)$ sowohl ein Element von CLIQUE wie von INDEPENDENT-SET.



Eine Reduktion von CLIQUE auf INDEPENDENT-SET funktioniert wie folgt:

- Eigentlich wollen wir das Problem »Ist $\big(\text{bin}(k), \text{code}(G)\big) \in$ CLIQUE?« lösen.
- Wir müssen dies nun *umwandeln* in die Frage »Ist $\big(\text{bin}(k'), \text{code}(G')\big) \in$ INDEPENDENT-SET?«
- Dazu *vertauschen wir Kanten und »Nichtkanten«.*

Aus  wird  .
Dann gilt tatsächlich:

- Im Originalgraphen gibt es eine Clique der Größe 3 genau dann, wenn es im neuen Graphen eine unabhängige Menge der Größe 3 gibt.

Anfängerfehler bei Reduktionen

23-13

Was muss eine Reduktion von A auf B alles leisten?

- Die Reduktion soll *nicht (!)* das Problem B lösen.
  Es wird nämlich *vorausgesetzt*, dass das Problem B »von jemand anderem« gelöst wird.
- Die Reduktion *soll auch nicht* das Problem A lösen.
  Vielmehr muss die Reduktion angeben, *wie man aus einer Fragestellung für A eine Fragestellung für B macht*.
- Es *reicht nicht*, wenn »jede Lösung für A« (also jedes $x \in A$) in »eine Lösung für B« (also ein $y \in B$) umgewandelt wird.
  Vielmehr muss auch im Falle von $x \notin A$ gelten $y \notin B$.

23-14

✎ Zur Übung

Betrachten Sie folgende Sprachen:

1. 1000-REACH enthält (die Codes von) Graphen G und zwei Knoten s und t, so dass es in G einen Weg von s nach t der Länge höchstens 1000 gibt.
2. 1001-REACH enthält (die Codes von) Graphen G und zwei Knoten s und t, so dass es in G einen Weg von s nach t der Länge höchstens 1001 gibt.

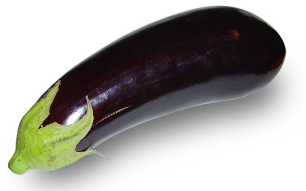
Geben Sie eine Reduktion von 1000-REACH auf 1001-REACH an.

23.2 Die Logspace-Many-One-Reduktion

23-15

Was uns zu einer formalen Definition noch fehlt.

- Was wir schon wissen: Eine Reduktion wandelt die Frage »$x \in A$?« in die Frage »$y \in B$?« um«.
- Was wir noch klären müssen: *Wie aufwendig darf die Umwandlung sein, also die Berechnung von y bei Eingabe x?*
- Etwas Nachdenken zeigt: Die Umwandlung muss *möglichst einfach sein*:
  - Ein Apfelstrudel und ein Birnenstrudel sind »gleich schwer zu machen«, da auch *blutige Anfänger* diese »ineinander umwandeln können«.
  - Würde man statt Anfängern nun *Sterne-Köche* die Umwandlung durchführen lassen, so *wären alle Gerichte gleich schwer:* Ein Chef-Koch kann vermutlich auch einen Auberginenauflauf in einen Rinderbraten verwandeln.
- Wir suchen deshalb *möglichst einfache* Maschinen.
- Die natürlichsten Kandidaten – endliche Automaten – sind *leider nicht geeignet*.



Author Horst Frank, Creative Commons Attribution Sharealike Licence

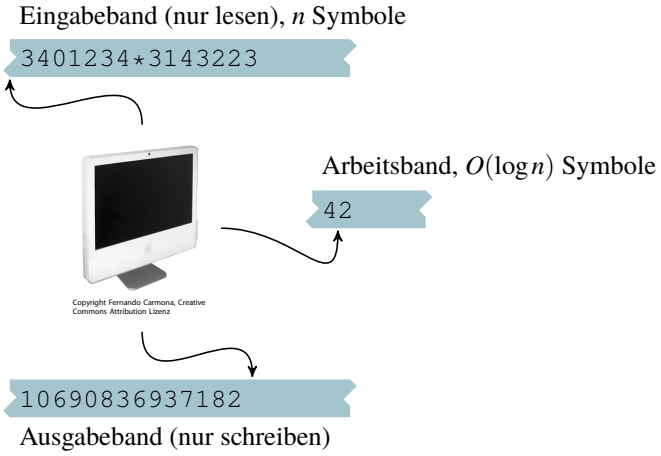
Author Samuel Burner, Creative Commons Attribution Sharealike Licence

23.2.1 Logspace-Maschinen

23-16

Logspace-Turing-Maschinen mit Ausgabe

Die Idee



23-17

Logspace-Turing-Maschinen mit Ausgabe

Die Definition

▶ Definition: Logspace-Turing-Maschine mit Ausgabe

Eine *Logspace-Turing-Maschine mit Ausgabe* ist eine DTM mit

- einem *Read-Only-Eingabeband*,
- *Arbeitsbänder*, auf denen der Platzverbrauch maximal $O(\log n)$ ist bei Eingaben der Länge n, und
- einem *Write-Only-Ausgabeband*.

▶ Definition: Logspace-berechenbare Funktionen
Eine Funktion $f\colon \Sigma^* \to \Sigma^*$ heißt *logspace-berechenbar*, wenn eine Logspace-Turing-Maschine M mit Ausgabe existiert, die

- bei jeder Eingabe x irgendwann anhält und
- dann auf dem Ausgabeband gerade $f(x)$ steht.

Was ist alles logspace-berechenbar?

23-18

Folgende Funktionen sind logspace-berechenbar:

- Die Grundrechenarten,
- Sortieren,
- Matrizen multiplizieren,
- in einem Graphen Kanten oder Knoten löschen oder hinzufügen.

Vermutlich nicht logspace-berechenbar sind:

- Schaltkreise auswerten,
- komplexere Graphprobleme wie CLIQUE,
- Formeln auf Erfüllbarkeit testen (SAT).

Merke

- Einfache Berechnungen, die man »gut per Hand machen könnte«, sind logspace-berechenbar.
- Probleme, die »komplexe Berechnungen« benötigen, sind nicht logspace-berechenbar.

Logarithmischer Platz ist nicht mehr als polynomielle Zeit.

23-19

▶ Lemma
Sei f logspace-berechenbar via einer Maschine M. Dann benötigt f auch nur polynomielle Zeit.

(Genauer: Für jedes $x \in \Sigma^*$ endet M bei Eingabe x nach $O(n^k)$ Schritten für ein festes k).

Beweisideen

- Wir haben so wahnsinnig wenig Platz, dass es in Bezug auf die Länge n von x *überhaupt nur polynomiell viele Möglichkeiten gibt, was auf den Arbeitsbändern stehen kann.*
- Rechnet man länger als es mögliche Bandinhalte gibt, so *muss sich irgendwann ein Bandinhalt wiederholen*.
- Dann wird sich der Bandinhalt aber nach nochmal so vielen Schritten auch nochmal wiederholen und so fort, die Maschine würde nie anhalten.

Skript

Beweis. Sei f logspace-berechenbar via M. Sei Γ das Bandalphabet. Betrachten wir eine beliebige Berechnung $C_{\text{init}}(x) = C_1 \vdash C_2 \vdash \cdots \vdash C_m$ von M. Per Definition hängt das Verhalten von M nicht von den Inhalten des Ausgabebandes ab. Sei deshalb C_i' die Konfiguration C_i ohne das Ausgabeband. Wir behaupten nun Folgendes: Es gibt keine zwei Indizes i und j mit $i < j$ und $C_i' = C_j'$. Gäbe es diese nämlich, so würde die Berechnung von C_i bis C_j nach C_j identisch wiederholt werden, da die Konfigurationen C_i und C_j aus Sicht der Maschine ununterscheidbar sind. Folglich würde die Berechnung $C_i \vdash \cdots \vdash C_j$ beliebig oft wiederholt werden und die Maschine würde nie halten.

Es bleibt abzuschätzen, wie viele Bandinhalte es in der Folge $C_1 \vdash \cdots \vdash C_m$ maximal geben kann. Dies sind $|\Gamma|^{O(\log n)}$, was in $n^{O(1)}$ liegt. Weiter ist die Anzahl der möglichen Kopfpositionen abzuschätzen. Auf den Arbeitsbändern sind dies lediglich $O(\log n)$ pro Band, also $O(\log^k n) \subseteq n^{O(1)}$ bei k Bändern. Für das Eingabeband ist die Situation etwas schwieriger, da es zwar nur n Eingabesymbole gibt, die Maschine ja aber beliebig weit weglaufen kann von der Eingabe. Nun gilt aber Folgendes: Entfernt sie sich weiter als $n^{\Omega(1)}$ von der Eingabe, so gibt es wieder zwei Konfigurationen, in denen die Maschine auf dem Eingabeband ein Blank liest, einen Schritt weg von der Eingabe macht und die gleichen Arbeitsbandinhalte vorliegen. Dann ergibt sich wieder eine endlose Wiederholung dieser Kette.

Insgesamt erhalten wir, dass in der Folge $C_1 \vdash \cdots \vdash C_m$ maximal $n^{O(1)} \cdot n^{O(1)} \cdot n^{O(1)}$ unterschiedliche Konfigurationen vorliegen können, was insgesamt in $n^{O(1)}$ liegt. □

23.2.2 Definition der Logspace-Many-One-Reduktion

23-20

Definition der Logspace-Many-One-Reduktion.

▶ Definition: Logspace-Many-One-Reduktion
Seien A und B Sprachen. Wir sagen *A ist auf B logspace-many-one-reduzierbar*, geschrieben $A \leq_{\mathrm{m}}^{\log} B$, falls es eine logspace-berechenbare Funktion f gibt, so dass

$$\text{für alle } x \in \Sigma^* \text{ gilt } x \in A \iff f(x) \in B.$$

Bemerkungen:

- Die Funktion f bekommt x als Eingabe und liefert $y = f(x)$ als Ausgabe.
- Die Funktion ist »leicht« berechenbar, nämlich logspace-berechenbar.
- Durch die Reduktion wird aus der Frage »Ist $x \in A$?« die Frage »Ist $f(x) \in B$?«.

23.2.3 Beispiele von Logspace-Many-One-Reduktionen

23-21

Beweisrezept: *Reduzierbarkeit beweisen*

Ziel
Beweisen, dass A auf B reduzierbar ist.

Rezept

1. Beginne mit: »Wir zeigen, dass $A \leq_{\mathrm{m}}^{\log} B$ via einer logspace-berechenbaren Funktion f gilt.«
2. Man überlegt sich dann, wie für beliebiges x die Frage »Ist $x \in A$?« in die Frage »Ist $f(x) \in B$?« umgewandelt werden kann. Beschreibe, wie f funktioniert.
3. Zeige dann, dass für alle $x \in \Sigma^*$ gilt $x \in A \iff f(x) \in B$. Zeige dazu zwei Richtungen:
   - »Sei $x \in A$. Dann .... Also gilt auch $f(x) \in B$.«
   - »Sei $f(x) \in B$. Dann .... Also gilt auch $x \in A$.«
4. Ende mit: »Für die Berechnung von f werden nur konstant viele Zähler benötigt, weshalb f logspace-berechenbar ist.«

23-22

Eine ganz einfache Reduktion.

▶ Lemma
CLIQUE $\leq_m^{\log}$ INDEPENDENT-SET.

Beweis. Wir zeigen, dass CLIQUE $\leq_{\mathrm{m}}^{\log}$ INDEPENDENT-SET via einer logspace-berechenbaren Funktion f gilt.[1] Die Funktion f nimmt (den Code von) einem Paar $\big(\mathrm{bin}(k), \mathrm{code}(G)\big)$ und gibt $\big(\mathrm{bin}(k), \mathrm{code}(G')\big)$ aus, wobei es genau dann eine Kante zwischen zwei Knoten in G' gibt, wenn es zwischen ihnen keine Kante in G gab. [2]
Sei $\big(\mathrm{bin}(k), \mathrm{code}(G)\big) \in$ CLIQUE. Dann enthält G eine Clique der Größe k. Genau diese Clique ist dann aber eine unabhängige Menge in G'. Also gilt $\big(\mathrm{bin}(k), \mathrm{code}(G')\big) \in$ INDEPENDENT-SET.[3]
Sei umgekehrt $\big(\mathrm{bin}(k), \mathrm{code}(G')\big) \in$ INDEPENDENT-SET. Dann enthält G' eine unabhängige Menge der Größe k und genau diese Menge ist eine Clique in G. Also gilt $\big(\mathrm{bin}(k), \mathrm{code}(G)\big) \in$ CLIQUE.[4]
Für die Berechnung von f werden nur konstant viele Zähler benötigt, weshalb f logspace-berechenbar ist. □

Kommentare zum Beweis

[1] Text aus dem Rezept

[2] So, jetzt ist die Funktion definiert. Nun muss man zeigen, dass sie korrekt ist.

[3] Das war die eine Richtung.

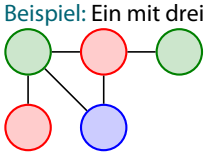
[4] Das war die zweite Richtung. Zugegebenermaßen fast identisch.

Das Konzept der Färbbarkeit.

23-23

▶ Definition: Färbbarkeit von Graphen
Sei G ein Graph. Wir nennen ihn *k-färbbar*, wenn man jeden seiner Knoten so mit einer von k möglichen Farben färben kann, dass durch eine Kante verbundene Knoten unterschiedliche Farben haben.

Beispiel: Ein mit drei Farben gefärbter Graph, der nicht 2-färbbar ist



▶ Definition: Färbbarkeit von Graphen
Sei k eine feste Zahl. Dann enthält die Sprache k-COLORABLE die Codes aller Graphen G, so dass G gerade k-färbbar ist.

Eine einfache Reduktion.

23-24

▶ Lemma
$3\text{-COLORABLE} \leq_m^{\log} 4\text{-COLORABLE}$.

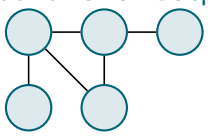
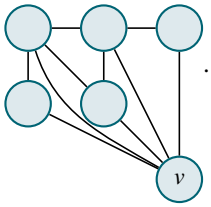
Beweis. Wir zeigen, dass 3-COLORABLE $\leq_{\mathrm{m}}^{\log}$ 4-COLORABLE via einer logspace-berechenbaren Funktion f gilt.[1] Die Funktion f nimmt (den Code von) einem Graphen G und gibt folgenden Graphen G' aus:[2]
Wir fügen einen neuen Knoten v hinzu und Kanten von v zu allen anderen Knoten.[3]
[4]Sei $\text{code}(G) \in 3\text{-COLORABLE}$. Dann hat G eine 3-Färbung mit beispielsweise den drei Farben Rot, Grün und Blau. Den Graphen G' kann man dann wie folgt 4-färben: Man färbt ihn genauso wie G, nur dass der neue Knoten eine weitere Farbe (Lila) bekommt. Dies ist dann offenbar eine 4-Färbung.
[5]Sei umgekehrt $\text{code}(G') \in 4\text{-COLORABLE}$. Dann hat G' eine 4-Färbung. *In dieser Färbung muss v eine andere Farbe haben als alle anderen Knoten, denn er ist mit allen anderen verbunden.* Also gibt es eine 3-Färbung für den Rest des Graphen. Dieser Rest ist aber gerade G. Für die Berechnung von f werden nur konstant viele Zähler benötigt, weshalb f logspace-berechenbar ist. □

Kommentare zum Beweis

[1] Text aus dem Rezept
[2] Das ist die entscheidende Idee.
[3] Jetzt muss man zeigen, dass das sinnvoll ist.
[4] Erste Richtung
[5] Zweite Richtung

Die Reduktion an einem Beispiel.

23-25

Der Graph  wird von f abgebildet auf 

.

23.2.4 Transitivität

Zentrale Eigenschaften der Logspace-Many-One-Reduktion.

23-26

▶ Lemma
Die Logspace-Many-One-Reduktion ist transitiv. Das bedeutet: Kann man A auf B reduzieren und B auf C, so auch A auf C.

Beweis. Sei $A \leq_{\mathrm{m}}^{\log} B$ via g und $B \leq_{\mathrm{m}}^{\log} C$ via f. Die Reduktion von A auf C ist gegeben durch die Funktion $f \circ g$, also die Funktion mit $(f \circ g)(x) = f(g(x))$. Für diese Funktion gilt nun für alle $x \in \Sigma^*$, dass $x \in A$ genau dann der Fall ist, wenn $(f \circ g)(x) \in C$, denn

$$x \in A \iff g(x) \in B \iff f(g(x)) \in C$$

aufgrund der Eigenschaften der Reduktionen von A auf B und von B auf C. Die eigentliche Schwierigkeit ist zu beweisen, dass $f \circ g$ auch logspace-berechenbar ist. Sei dazu eine Eingabe x gegeben. Wir wollen $f(g(x))$ in logarithmischem Platz berechnen. Man kann *nicht* einfach erst $g(x)$ ausrechnen und dann f auf das Ergebnis anwenden, da das Zwischenergebnis viel zu lang ist.

Der Trick ist, ein »virtuelles« Arbeitsband zu benutzen:

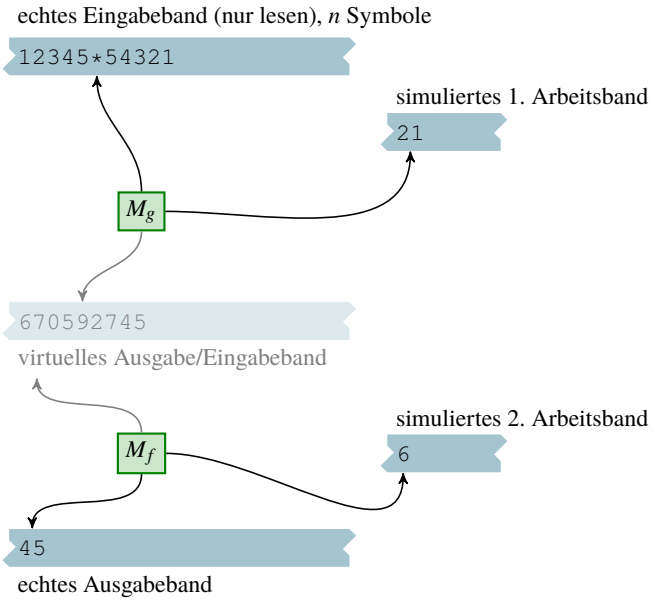
Skript

- Man beginnt mit der Berechnung von $f(g(x))$ und tut so, als wüsste man, wie $g(x)$ lautet.
- Wenn nun tatsächlich ein Bit von $g(x)$ benötigt wird, wird in diesem Moment erst eine Simulation von g auf der (echten) Eingabe x gestartet.
- Die Ausgaben der Simulation werden unterdrückt bis auf das interessierende Bit.

Betrachten wir dazu ein Beispiel. Gegeben seine folgende zwei logspace-berechenbare Funktionen, die wir komponieren wollen:

- Die erste Funktion g sei die Multiplikationsfunktion; genauer die Funktion, die durch einen Stern getrennte Ziffernfolgen auf das Produkt der Ziffernfolgen abbilden. Sei M_g eine Logspace-Maschine, die g berechnet (man kann zeigen, dass eine solche Maschine existiert).
- Die zweite Funktion f sei die Quersummenfunktion; genauer die Funktion, die eine Ziffernfolge auf die Summe dieser Ziffern abbildet. Sei M_f eine Logspace-Maschine, die f berechnet.
- Die Komposition $f \circ g$ bildet dann beispielsweise `12345*54321` auf das Wort `45` ab, da $12345 \cdot 54321 = 670592745$ und die Quersumme von 670592745 gerade 45 ist.

Was nun passiert während der Berechnung veranschaulicht folgende Abbildung:



Wichtig ist, dass das »ausgegraute« Band gar nicht wirklich vorhanden ist. Vielmehr wird immer, wenn M_g ein Bit dieses Bandes benötigt, M_f simuliert und alle Bits bis auf das »interessierende« weggeschmissen.

Der Platzverbrauch für die Berechnung von $f \circ g$ ergibt sich wie folgt: Sei x mit Länge $n = |x|$ die Eingabe.

1. Es wird $O(\log n)$ Platz benötigt, um M_g auszuführen.
2. Die Ausgabe von M_g hat maximale Länge $O(n^k)$ für eine Konstante k, da M_g nach Lemma 23-19 polynomiell zeitbeschränkt ist.
3. Folglich wird lediglich $O(\log(n^k)) = O(\log n)$ Platz benötigt, um die Bitposition für die Simulation zu speichern.
4. Weiter wird $O(\log(n^k)) = O(\log n)$ Platz benötigt für die Simulation von M_f, da M_f eine Eingabe der Länge höchstens $O(n^k)$ erhält.

Insgesamt ergibt dies einen Platzverbrauch von $O(\log n)$, wie behauptet. □

Zusammenfassung dieses Kapitels

1. Logspace-Maschinen dürfen ihre *Eingabe nur lesen*, ihre *Ausgabe nur schreiben* und *nur logarithmisch viel Platz* auf ihrem Arbeitsband benutzen.
2. Ist A *auf* B *reduzierbar*, so bedeutet dies, dass die Frage »Ist $x \in A$?« sehr leicht (nämlich in logarithmischem Platz) auf die Frage »Ist $y \in B$?« überführt werden kann.
3. Ist A *auf* B *reduzierbar*, so ist A *nicht schwieriger zu lösen als* B.
4. Eine Reduktion von A auf B »löst« weder das Problem A noch das Problem B direkt – sie verwandelt nur Fragen ineinander.

23-27

Zum Weiterlesen

[1] R. Reischuk. *Unterlagen zum Lehrmodul Theoretische Informatik,* Vorlesungsskript, 2008. Kapitel 8.2.

Übungen zu diesem Kapitel

In den folgenden Aufgaben werden eine Reihe von Problemen erwähnt, die im Folgenden als Referenz formal definiert werden:

Definition
Die Sprache REACH enthält alle Worte $\text{code}(G,s,t)$, so dass G ein gerichteter Graph ist, in dem es einen Weg vom Knoten s zum Knoten t gibt.

Definition
Die Sprache CONNECTED enthält alle Worte $\text{code}(G)$, so dass G ein[1] stark zusammenhängder gerichteter Graph ist.

Kommentare zur Definition

[1] Das bedeutet, zwischen je zwei Knoten gibt es einen Pfad.

Definition
Die Sprache k-REACH enthält für festes k die Codes von Graphen G und zwei Knoten s und t, so dass es in G einen Weg von s nach t der Länge höchstens k gibt.

Definition
Die Sprache k-COLORABLE enthält für festes k die Codes von k-färbbaren Graphen G.

Definition
Die Sprache k-CLIQUE enthält für festes k die Codes von Graphen, die eine Clique der Größe k haben.

Definition
Die Sprache TRIPPLE-REACH enthält die Codes von Graphen G und drei Knotenpaare (s,t), (s',t') und (s'',t''), so dass es in G einen Weg von s nach t, einen Weg von s' nach t' und einen Weg von s'' nach t'' gibt. Die Wege brauchen nicht disjunkt sein.

Übung 23.1 Reduktion von zwischen Erreichbarkeitsproblemen, mittel

Zeigen Sie, dass REACH $\leq_{\text{m}}^{\log}$ CONNECTED.

Tipp: Fügen Sie für jeden Knoten v eine Kante von v nach s und eine Kante von t nach v hinzu.

Übung 23.2 Reduktion von zwischen Erreichbarkeitsproblemen, schwer

Zeigen Sie, dass CONNECTED $\leq_{\text{m}}^{\log}$ REACH.

Übung 23.3 Reduktion von k-REACH auf k^2-REACH, mittel

Zeigen Sie, dass es eine Logspace-Many-One-Reduktion von k-REACH auf k^2-REACH gibt.

Übung 23.4 Reduktion von k-COLORABLE auf $2k$-COLORABLE, leicht

Zeigen Sie, dass eine Logspace-Many-One-Reduktion von k-COLORABLE auf $2k$-COLORABLE existiert.

Tipp: Verallgemeinern Sie die Konstruktion der Reduktion von 3-Färbbarkeit auf 4-Färbbarkeit.

Übung 23.5 Reduktion von STRONG-CONNECT auf REACH, schwer

Zeigen Sie, dass es eine Reduktion von STRONG-CONNECT auf REACH gibt.

Tipp: Verallgemeinern Sie die Konstruktion der Reduktion von TRIPPLE-REACH auf REACH.

Übung 23.6 CLIQUE ist schwer für CLIQUE-PROBLEMS, mittel

Sei CLIQUE-PROBLEMS $= \{1\text{-CLIQUE}, 2\text{-CLIQUE}, 3\text{-CLIQUE}, \dots\}$. Beweisen Sie:

1. CLIQUE ist $\leq_m^{\log}$-schwer für CLIQUE-PROBLEMS,
2. INDEPENDENT-SET ebenfalls,
3. CLIQUE-PROBLEMS $\subseteq$ P.

Übung 23.7 Reduktion von k-CLIQUE auf $(k+1)$-CLIQUE, mittel

Geben Sie eine Logspace-Many-One-Reduktion von k-CLIQUE auf $(k+1)$-CLIQUE an.

Übung 23.8 Reduktion von 1000-REACH auf 2000-REACH, mittel

Geben Sie eine Logspace-Many-One-Reduktion von 1000-REACH auf 2000-REACH an.

Übung 23.9 Reduktion von TRIPPLE-REACH auf REACH, mittel

Geben Sie eine Logspace-Many-One-Reduktion von TRIPPLE-REACH auf REACH an.

24-1

Kapitel 24
Die Idee der Vollständigkeit

Vollständige Problem: Die Schweizer Taschenmesser der Theorie

24-2

Lernziele dieses Kapitels

1. Die Zeit-Platz-Hierarchie kennen
2. Konzept der Schwere verstehen
3. Konzept der Vollständigkeit verstehen
4. Wichtigkeit der Transitivität von Reduktionen kennen

Inhalte dieses Kapitels

Worum es heute geht

Autor Chrisitan Horvat, Creative Commons Attribution Sharealike Lizenz

Das letzte Kapitel hat Ihnen Reduktion hoffentlich bereits schmackhaft gemacht. Manche Probleme haben nun die Eigenschaft, dass ziemlich viele andere Probleme auf sie reduziert werden können. Wenn Sie eine Caipirinha gemixt bekommen, dann werden Sie auch keine Probleme haben mit einer Caipiroschka, einer Caipirissima oder zur Not halt einer Virgin-Caipi. Man kann guten Gewissens behaupten: Sie werden in der ganzen Sippe der Caipi-Cocktails keinen wirklich »schwierigeren« Cocktail finden. Der Theoretiker würde nun sagen, dass das Problem der Caipirinha-Herstellung *vollständig* ist für die Klasse aller Caipi-Cocktails (vorausgesetzt, der Theoretiker kann nach der eingehenden Untersuchung der verschiedenen Cocktails noch verständlich kommunizieren).

In diesem Kapitel geht es um die Frage, wo man in der Theorie außer bei alkoholischen Getränken noch überall vollständige Probleme antreffen kann. Sprich: Gibt es in bestimmten Komplexitätsklassen vielleicht Probleme, die »unglaublich hilfreich« sind? Gibt es Probleme, so dass man alle Probleme in der Klasse lösen kann, wenn man nur diese Problem lösen kann? Gibt es also Probleme, so dass sich alle Probleme einer Klasse auf dieses eine Problem reduzieren lassen?

Die Intuition sagt einem zunächst, dass es solche Probleme nicht geben sollte. Nehmen wir die doch recht große Klasse P. Diese versammelt ja alle Probleme, die sich in Zeit $O(n)$ oder in $O(n^2)$ oder in $O(n^3)$ oder in $O(n^4)$ oder so weiter lösen lassen. Man darf deshalb erwarten, dass es zu jedem Problem in P »immer ein noch schwierigeres« Problem in P geben sollte. Die Intuition ist jedoch falsch: Es *gibt* ein Problem in P, auf das sich alle anderen Probleme in P reduzieren lassen. Kann man also dieses Problem so-und-so gut lösen, so kann man auch alle anderen Probleme in P prinzipiell auch so-und-so gut lösen. Dieses Problem wird in Kapitel 26 detaillierter vorgestellt. Die Klasse P ist nicht die einzige Klasse, die ein solches »besonders hilfreiches« Problem enthält. Es hat sich herausgestellt, dass die meisten Komplexitätsklassen solche »vollständigen« Probleme besitzen.

In diesem Kapitel werden zunächst die Idee der Vollständigkeit veranschaulicht, bevor die nächsten Kapitel dann für verschiedene Klassen vollständige Probleme vorstellen werden.

24.1 Klassenhierarchien

24.1.1 Inklusionen

24-4

Warum Klasseninklusionen wichtig sind.

- In der Komplexitätstheorie wird viel Theorieenergie aufgewandt, um für verschiedene Klassen C_1 und C_2 zu zeigen, dass $C_1 \subseteq C_2$ gilt oder nicht gilt.
- Solche Inklusionen erlauben es, uns auf *einen Aspekt eines Problems zu fokussieren* und es dann *automatisch auf eine andere Art zu lösen*.

Beispiel

- Wir zeigen gleich $\mathrm{L} \subseteq \mathrm{P}$.
- Wenn wir also einen *sehr platzeffizienten* Algorithmus für ein Problem finden, so *gilt automatisch*, dass das Problem in vertretbarer Laufzeit lösbar ist.

24-5

Inklusionen zwischen Zeit- und Platzklassen.

▶ Lemma
Sei f eine nicht zu exotische Funktion mit $f(n) \geq \log(n)$. Dann gilt $\mathrm{TIME}(f) \subseteq \mathrm{SPACE}(f) \subseteq \mathrm{TIME}(2^{O(f)})$.

Beweis. Die erste Inklusion ist ganz einfach: Man kann nicht mehr Zellen beschreiben als man Rechenschritte macht.
Für die zweite Inklusion argumentiert man ganz ähnlich wie in Lemma 23-19:

- Macht eine Maschine mit Platzverbrauch $f(n)$ bei Eingabe n *mehr als* $2^{O(f(n))}$ *Schritte*, so *muss sie in eine Endlosschleife gegangen sein*.
- Dann kann man die Berechnung aber auch abbrechen und die Eingabe verwerfen. □

24-6

Die wichtigen Inklusionen zwischen den Komplexitätsklassen.

▶ Folgerung

$$\begin{aligned}\mathrm{REG} \subseteq \mathrm{L} \subseteq \mathrm{P} \subseteq \mathrm{PSPACE} \subseteq \mathrm{EXP}\\ \subseteq \mathrm{EXPSPACE}\\ \subseteq \mathrm{ELEMENTARY}.\end{aligned}$$

24.1.2 Trennungen

24-7

Echte Inklusionen von Komplexitätsklassen

Intuitiv sollte man mit »mehr Zeit« auch »mehr Probleme« lösen können. Dem ist auch so:

▶ Satz: Zeithierarchie-Satz
Sei $f(n) \geq n$ eine nicht zu exotische Funktion. Dann gilt

$$\mathrm{TIME}(f) \subsetneq \mathrm{TIME}\big(f(n)^3\big).$$

▶ Satz: Platzhierarchie-Satz
Sei $f(n) \geq \log n$ eine nicht zu exotische Funktion. Dann gilt

$$\mathrm{SPACE}(f) \subsetneq \mathrm{SPACE}\big(f(n)\log f(n)\big).$$

Beweisidee

- Dies beweist man durch *Diagonalisierung*.
- Mit diesem Verfahren hatten wir gezeigt, dass HALTING in der Klasse RE ist, aber nicht in REC.
- Analog zeigt man hier, dass es eine Sprache gibt, die in $\text{TIME}(f(n)^3)$, aber nicht in $\text{TIME}(f)$ liegt.

Wir beweisen lediglich den Zeithierarchie-Satz; der Platzhierarchie-Satz geht ähnlich. Skript

Beweis des Zeithierarchie-Satzes. Wir definieren die folgende Diagonalisierungssprache:

$$D_f = \{\text{code}(M) \mid M \text{ akzeptiert } \text{code}(M) \text{ nicht innerhalb von } f(|\text{code}(M)|) \text{ Schritten}\}.$$

Wir behaupten, dass $D_f \in \text{TIME}\big(f(n)^3\big) \setminus \text{TIME}(f)$:

- Es gilt $D_f \in \text{TIME}\big(f(n)^3\big)$ via einer Simulation von M bei Eingabe $\text{code}(M)$ für $f(|\text{code}(M)|) = f(n)$ Schritte.
- Nehmen wir an, es würde $D_f \in \text{TIME}(f)$ via einer Maschine M gelten. Sei $x = \text{code}(M)$. Dann gilt:
  1. Falls M das Wort x akzeptiert, so gilt $x \in D_f$ und deshalb akzeptiert M das Wort x nicht in $f(|x|)$ Schritten (ein Widerspruch).
  2. Falls M das Wort x nicht akzeptiert, so gilt $x \notin D_f$. Dann braucht entweder M mehr als $f(|x|)$ Schritte (ein Widerspruch) oder M akzeptiert x (auch ein Widerspruch).

  Wir erhalten, dass $D_f \notin \text{TIME}(f)$. □

Trennungen der Zeit- und Platzklassen. 24-8

▶ **Folgerung**

$$\text{P} \subsetneq \text{EXP} \subsetneq \text{EEXP} \subsetneq \text{ELEMENTARY},$$
$$\text{L} \subsetneq \text{PSPACE} \subsetneq \text{EXPSPACE}.$$

▶ **Lemma**
In der Inklusionskette

$$\text{L} \subseteq \text{P} \subseteq \text{PSPACE}$$

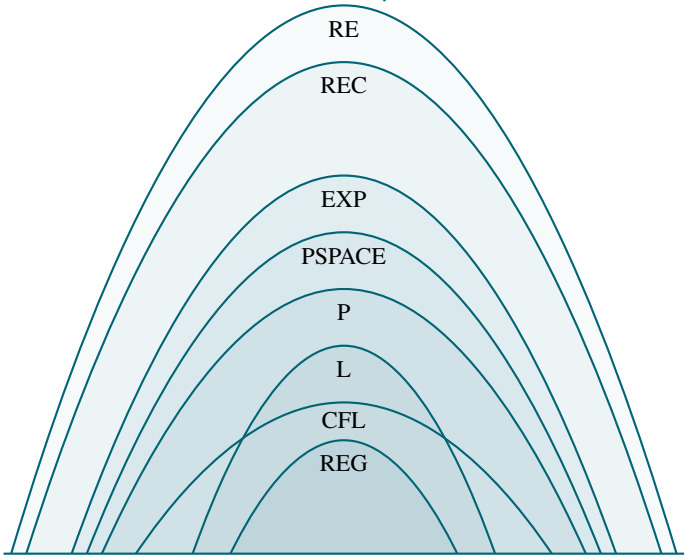
ist mindestens eine Inklusion echt.

Da $\text{L} \subsetneq \text{PSPACE}$, gilt $\text{L} \neq \text{P}$ oder $\text{P} \neq \text{PSPACE}$. Man *vermutet, dass beides der Fall ist*, konnte aber *weder das eine noch das andere* bis heute *beweisen*. Dies ist sehr frustrierend.

24.1.3 Übersicht der Klassenhierarchie

24-9

Was man über die Hierarchie der Sprachklassen weiß.



Ließe man die Klassen L und PSPACE aus dem Diagramm weg, so wären alle Inklusionen beweisbar echt.

24.2 Schwere und Vollständigkeit

24.2.1 Die Ideen

24-10

Viele Probleme, die sich auf ein einziges Problem reduzieren lasse

- Betrachten wir das Problem 100-COLORABLE, einen Graphen mit 100 Farben zu färben.
- Genau wie bei der Reduktion von 3-Färbbarkeit auf 4-Färbbarkeit (Lemma 23-24) kann man zeigen, dass 99-COLORABLE auf 100-COLORABLE reduzierbar ist.
- Ebenso ist 98-COLORABLE auf 99-COLORABLE reduzierbar.
- Da Reduktionen transitiv sind (= statt zweimal nacheinander zu reduzieren kann man auch in einem Rutsch direkt reduzieren, siehe Lemma 23-26), ist damit auch 98-COLORABLE direkt auf 100-COLORABLE reduzierbar.
- Genauso zeigt man, dass auch 1-COLORABLE und 2-COLORABLE und 3-COLORABLE und so weiter bis 99-COLORABLE *alle auch auf* 100-COLORABLE *reduzierbar sind*.

Merke

Kann man 100-COLORABLE effizient lösen, so auch alle k-Färbbarkeitsprobleme für $k \leq 100$.

24-11

Die Schweizer Taschenmesser der Theorie

Autor Cédric, Creative Commons Attribution Sharealike Lizenz

- Es gibt Probleme, auf die sich *ganz viele Probleme* reduzieren lassen.
- Ein solches Problem ist *wie ein Schweizer Taschenmesser – man kann sehr viel damit anfangen*.

Formal funktioniert das so:

- Man hat *eine Klasse von Problemen* gegeben.
- Wir sind nun an einem Problem X interessiert, auf das sich *alle Probleme in der Klasse* reduzieren lassen.

24.2.2 Definition: Schwere Probleme

Schwere Probleme = harte Probleme = hilfreiche Problem = Schweizer Taschenmesser der Theorie 24-12

▶ Definition: Schwere Probleme
Seien

- C eine Klasse von Sprachen und
- X ein Problem (nicht unbedingt in C).

Dann heißt X *schwer für* C *unter* $\leq_m^{\log}$*-Reduktionen*, falls

- für *alle* Probleme $A \in C$ gilt:
- $A \leq_{\mathrm{m}}^{\log} X$.

Statt »schwer« sagt man oft auch »hart« (englisch »hard«).

Motto
Kann man ein für C schweres Problem effizient lösen, dann kann man alle Probleme in C effizient lösen.

Beispiele von schweren Problemen im Bereich Färbbarkeit 24-13

Beispiel
Sei $C = \{1\text{-COLORABLE}, 2\text{-COLORABLE}, \ldots, 100\text{-COLORABLE}\}$.
Dann ist 100-COLORABLE *schwer für* C.

Beispiel
Sei COLORING-PROBLEMS $= \{1\text{-COLORABLE}, 2\text{-COLORABLE}, \ldots\}$ die Klasse *aller* k-Färbeprobleme. Dann ist folgende Sprache schwer für COLORING-PROBLEMS:

$$\text{COLORABLE} = \big\{\big(\text{code}(G), \text{bin}(k)\big) \mid G \text{ ist } k\text{-färbbar}\big\}.$$

Die Reduktionen k-COLORABLE $\leq_{\mathrm{m}}^{\log}$ COLORABLE sind auch sehr einfach: Sie bildet einfach $\text{code}(G)$ ab auf $\big(\text{code}(G), \text{bin}(k)\big)$.

Beispiele von schweren Problemen im Bereich Erfüllbarkeit 24-14

Zur Erinnerung aus »Logik für Informatiker«
Aussagenlogische Formeln nennt man *in konjunktiver Normalform*, wenn sie eine große Konjunktion (»Verundung«) von Klauseln (Oder-Blöcken) ist.

▶ Definition
Sei $k \geq 1$ fest. Die Sprache k-SAT enthält alle Formeln φ, die

1. erfüllbar sind und
2. in konjunktiver Normalform sind und
3. jede Klausel genau k Literale enthält.

Beispiel
Das Wort $(x \vee y \vee \neg z) \wedge (\neg x \vee \neg y \vee \neg z)$ über dem Alphabet $\Sigma_{\text{Aussagenlogik}}$ ist ein Element von 3-SAT, jedoch kein Element von 2-SAT.

Beispiel
Sei CNF-SAT-VARIANTS $= \{1\text{-SAT}, 2\text{-SAT}, 3\text{-SAT}, \ldots\}$.
Dann ist SAT schwer für CNF-SAT-VARIANTS.

✎ Zur Übung
Erläutern Sie, wie die Reduktion von 3-SAT auf SAT funktioniert.
(Die Reduktion ist sehr einfach, es gibt aber ein Fallstricklein.)

24-15

Beispiele von schweren Problemen im Bereich Erreichbarkeit.

▶ Definition

Für jedes k enthält die Sprache k-REACH (die Codes von) allen gerichteten Graphen G zusammen mit zwei Knoten s und t, so dass es einen Weg von s nach t der Länge höchstens k gibt.

Beispiel

Sei REACH-PROBLEMS $= \{1\text{-REACH}, 2\text{-REACH}, \dots\}$.
Dann ist folgende Sprache schwer für die REACH-PROBLEMS:

$$\text{DISTANCE} = \{\text{code}(G,s,t,k) \mid \text{es gibt einen Weg von } s \text{ nach } t \text{ in } G \text{ der Länge höchstens } k\}.$$

24.2.3 Definition: Vollständige Probleme

24-16

Schwere, aber nicht zu schwere Probleme.

Wir haben mehrere Fälle kennengelernt, wo bei bestimmtes Problem sehr hilfreich (formal »schwer«) für die Probleme einer Klasse ist:

- SAT für die Klasse aller k-SAT-Probleme.
- COLORABLE für die Klasse aller k-Färbeprobleme.

Frage

Könnte es sein, dass die Klassen auch *bereits selbst* Probleme enthalten, die schwer für die Klasse sind?

Beispiel

Was dies für die Klasse COLORING-PROBLEMS bedeuten würde:

- Eines der Probleme 1-COLORABLE, 2-COLORABLE, … müsste die Eigenschaft haben, dass sich *alle anderen auf dieses Problem reduzieren lassen*.
  Dies könnte beispielsweise 100-COLORABLE sein.
- Wir haben schon gesehen, dass sich 1-COLORABLE bis 99-COLORABLE auf 100-COLORABLE reduzieren lassen.
- Es ist jedoch *reichlich unklar*, wie man beispielsweise 1000000-COLORABLE auf 100-COLORABLE reduzieren könnte.

Jedoch:

- Man kann beweisen, dass 3-COLORABLE *schwer ist* für die Klasse.
- Vermutlich ist 2-COLORABLE *nicht schwer* für die Klasse.

24-17

Probleme in einer Klasse, die hilfreich für die Klasse sind.

▶ Definition: Vollständige Probleme

Seien

- C eine Klasse von Sprachen und
- X ein Problem.

Dann heißt X *vollständig für* C *unter* $\leq_{\text{m}}^{\log}$*-Reduktionen*, falls

- $X \in C$ und
- X ist schwer für C.

Motto

Vollständige Probleme sind die schwierigsten Probleme einer Klasse.

Beispiel einer Klasse mit einem vollständigen Problem

24-18

Sei SAT-VARIANTS die Klasse aller möglichen Varianten von Erfüllbarkeitsproblemen. Konkret soll diese Klasse folgende Probleme enthalten:

- k-SAT für jedes k,
- k-DNF-SAT für jedes k (diese Probleme sind definiert wie k-SAT, nur mit disjunktiver Normalform statt konjunktiver Normalform),
- CNF-SAT (dieses Problem enthält alle erfüllbaren Formeln in konjunktiver Normalform; die Klauselgröße ist beliebig),
- DNF-SAT (dieses Problem enthält alle erfüllbaren Formeln in disjunktiver Normalform),
- SAT.

▶ **Lemma**

SAT *ist vollständig für* SAT-VARIANTS.

Beweis. Offenbar ist SAT $\in$ SAT-VARIANTS. Weiter wurde auf Folie 24-14 gezeigt, dass jedes Problem in der Klasse jeweils sehr einfach auf SAT reduzierbar ist. □

Welche Klassen haben vollständige Probleme?

24-19

- In diesem Kapitel haben wir nur *etwas künstliche* Klassen betrachtet.
- Sie waren *so konstruiert*, dass sie ein vollständiges Problem haben.

Spannender ist folgende Frage:

Die zentralen Frage der Komplexitätstheorie

Welche »natürlichen« Klassen wie L oder P haben vollständige Probleme? Wie lauten diese?

Es ist *erstaunlich*, dass Klassen wie P vollständige Probleme haben. Wir werden aber noch zeigen, dass *sehr viele natürliche Klassen vollständige Probleme haben:*

- Die Variante von REACH für ungerichtete Graphen ist vollständig für L.
- Das Schaltkreis-Auswertungsproblem ist vollständig für P.
- Das Brettspiel »Go« ist vollständig für PSPACE.
- HALTING ist vollständig für RE.

24.2.4 Vollständigkeit und Klassenhierarchien

Warum man vollständige Probleme untersucht.

24-20

Es gibt verschiedene Gründe, weshalb man vollständige Probleme untersucht:

- Ein vollständiges Problem ist ein »schwierigstes« Problem in einer Klasse.
- Will man *verstehen*, was alles in einer Klasse *prinzipiell möglich ist*, so *genügt es*, das vollständige Problem zu untersuchen.
- Insbesondere kann man häufig *Inklusionen zwischen Klassen* beweisen, indem man *nur die vollständigen Probleme untersucht*.

Eine wichtige Definition und ein nützliches Lemma.

24-21

▶ **Definition:** Abgeschlossen

Eine Klasse C von Sprachen heißt *abgeschlossen unter* $\leq_m^{\log}$*-Reduktionen*, falls aus $A \leq_m^{\log} B$ und $B \in C$ folgt $A \in C$.

Motto

Abgeschlossene Klassen kann man mit Reduktionen nicht »verlassen«.

▶ **Lemma**

Die Klassen L, P, PSPACE, EXP *sind abgeschlossen.*

Beweis. Für die Abgeschlossenheit von L sei $A \leq_m^{\log} B$ via f und $B \in$ L. Da $B \in$ L, gibt es einen logspace-beschränkten Entscheider (eine DTM, die immer anhält) für B. Dann ist aber auch die charakteristische Funktion von B, definiert als $\chi_B \colon \Sigma^* \to \{0,1\}$ mit $\chi_B(x) = 1 \iff x \in B$, logspace-berechenbar. Die Verkettung $\chi_B \circ f$ ist nach Lemma 23-26 ebenfalls logspace-berechenbar. Nun ist aber $\chi_B \circ f = \chi_A$. Also ist die charakteristische Funktion von A ebenfalls logspace-berechenbar und somit auch A von einer logspace-beschränkten Turing-Maschine entscheidbar.

Skript

Für die Abgeschlossenheit von P argumentiert man genauso, man benötigt lediglich, dass die Verkettung zweier Funktionen, die beide in polynomieller Zeit berechenbar sind, selbst wieder in polynomieller Zeit berechenbar ist.

Für die Abgeschlossenheit von PSPACE sei $A \leq_{\mathrm{m}}^{\log} B$ via f und $B \in \mathrm{PSPACE}$. Dann lässt sich A wie folgt in polynomiellem Platz entscheiden: Bei Eingabe x wird zunächst $f(x)$ berechnet, was höchstens polynomielle Länge in $|x|$ haben kann. Dann wird der polynomiell platzbeschränkte Entscheidungsalgorithmus für B auf $f(x)$ angewandt. Dies benötigt polynomiell viel Platz in der Länge von $f(x)$, welches seinerseits polynomiell lang in der Länge von x ist. Da die Verkettung von Polynomen selbst ein Polynom ist, benötigen wir insgesamt nur polynomiell viel Platz.

Für die Abgeschlossenheit von EXP argumentiert man wie bei P, nur benötigt man nun, dass die Verkettung eines Polynoms q und einer Funktion der Bauart $2^{p(n)}$ für ein Polynom p wieder eine Funktion der Bauart $2^{r(n)}$ ergibt, wobei $r(n) = p\big(q(n)\big)$ wieder ein Polynom ist. □

24-22

Vollständige Probleme repräsentieren ihre Klassen.

▶ **Lemma:** Repräsentationslemma
Sei X vollständig für eine Klasse C.

1. *Es gilt $X \in \mathrm{P}$ genau dann, wenn $C \subseteq \mathrm{P}$.*
2. *Es gilt $X \in \mathrm{PSPACE}$ genau dann, wenn $C \subseteq \mathrm{PSPACE}$.*
3. *Es gilt $X \in \mathrm{EXP}$ genau dann, wenn $C \subseteq \mathrm{EXP}$.*

Beweis. Für die erste Richtung der ersten Behauptung argumentiert man wie folgt:

- Enthält die Klasse P das Problem X, so enthält sie (aufgrund ihrer Abgeschlossenheit) auch alle auf X reduzierbaren Probleme.
- Da alle Probleme in C auf X reduzierbar sind (aufgrund der Schwere von X für C), sind folglich alle Probleme aus C in P.

Die zweite Richtung ist trivial. Für die anderen beiden Klassen argumentiert man genauso. □

Merke
Will man zeigen, dass eine Klasse C eine Teilmenge von P ist, so genügt es, dies für *ein vollständiges Problem* zu beweisen.

24-23

Vollständige Probleme liefern untere Schranken.

▶ **Satz**
Sei X vollständig für EXP. *Dann ist $X \notin \mathrm{P}$.*

Beweis. Wäre $X \in \mathrm{P}$, so wäre nach dem Repräsentationslemma auch $\mathrm{EXP} \subseteq \mathrm{P}$. Nach dem Zeithierarchie-Satz gilt aber $\mathrm{P} \subsetneq \mathrm{EXP}$, ein Widerspruch. □

- Für vollständige Probleme für EXP wissen wir also, dass sie *definitiv nicht* in polynomieller Zeit lösbar sein können.
- Es ergibt also *keinen Sinn*, für solche Probleme überhaupt nach Polynomialzeit-Algorithmen zu suchen – ähnlich wie man nicht nach Algorithmen für das Halteproblem suchen braucht.
- Dies ist ein wichtiges Resultat, das man *nur durch theoretische Überlegungen* erreichen konnte.

Beispiel
Man kann zeigen, dass »verallgemeinertes Schach« vollständig ist für EXP. Folglich *kann es keinen effizienten Algorithmus für das verallgemeinerte Schach geben*.

Zusammenfassung dieses Kapitels

1. Es gilt $L \subseteq P \subseteq PSPACE \subseteq EXP$. 24-24
2. Man weiß, dass $L \subsetneq PSPACE$ und $P \subsetneq EXP$.
3. Ein Problem heißt *schwer* für eine Klasse, wenn alle Probleme der Klasse auf das Problem reduzierbar sind.
4. Ein Problem heißt *vollständig* für eine Klasse, wenn es schwer ist und in der Klasse liegt.
5. Fast alle wichtigen Klassen haben vollständige Probleme.

Zum Weiterlesen

[1] R. Reischuk. *Unterlagen zum Lehrmodul Theoretische Informatik,* Vorlesungsskript, 2008. Kapitel 8.1, 8.2.

25-1

Kapitel 25
NL-Vollständigkeit

Nichtdeterministische logspace-beschränkte Turing-Maschinen sind Hitech-Ariadne-Fäden

25-2

Lernziele dieses Kapitels

1. Nichtdeterministischen Platz verstehen
2. Das Erreichbarkeitsproblem REACH kennen
3. Beweis der NL-Vollständigkeit von REACH verstehen
4. Bootstrapping- und Reduktionsmethode kennen
5. Eigene NL-Vollständigkeitbeweise führen können

Inhalte dieses Kapitels

Worum es heute geht

Die Untersuchung von kleinen Platzklassen erscheint auf den ersten Blick reichlich »akademisch«, »mainly of scholary interest«, kurz gesagt: überflüssig. Speicherplatz ist (im Gegensatz zu Zeit) in aller Regel wahrlich nicht unser Problem. Wenn eine Ingenieurin ein Gigabyte an Eingabedaten hat, dann wird sie sicherlich auch irgendwo noch Platz für ein Gigabyte weiteren Speicher finden. Man könnte einwenden, dass kleine Systeme wie Smartcards oder eingebettete Systeme wie Waschmaschinensteuerungen nicht mit übermäßigem Speicherplatz gesegnet seien und es sich sehr wohl lohne, sparsam damit umzugehen.

Wohl wahr – jedoch erscheint es schon von dagobertduckhaftem Geiz zu zeugen, nur *logarithmisch* viel Platz vorzusehen. Hat man beispielsweise ein Gigabyte an Eingabedaten, so dürfte man bei einem erlaubten Platz von $\log_2 n$ Bits gerademal 33 Bits an Speicher verbrauchen, also großzügig gerundet zwei 32-Bit-Register. Wächst die Eingabemenge auf 1000 Terabyte, so hätte man immerhin 53 Bits zur Verfügung, also immernoch nicht mehr als zwei Register. Bei der größten in diesem Universum überhaupt denkbaren Eingabe (10^{80}, denn so viele Atome hat das Universum, plus/minus ein paar Zehnerpotenzen), hätte man immerhin zehn 32-Bit-Register zur Verfügung.

In diesem Kapitel soll nun logarithmischer Platz noch mit Nichtdeterminismus gepaart werden, was uns zur Klasse NL führen wird. Diese Paarung mag Theoretiker beglücken, dem Praktiker stehen aber zunächst die Haare zu Berge: Da wird ein unrealistisches Modell (lächerlich wenig Platz) mit einem noch viel unrealistischeren Modell (nichtdeterministische Turing-Maschinen) zusammengebracht; das Resultat sind Maschinen, die man nicht bauen kann, und wenn man sie bauen könnte, sinnlosen Einschränkungen unterliegen.

Warum lohnt es sich trotzdem, nichtdeterministischen logarithmischen Platz zu untersuchen? Ich möchte folgende Gründe anführen:

1. Erstaunlicherweise sind nichtdeterministischer logarithmischer Platz und die Parallelisierbarkeit von Problemen aufs Engste verwoben; die Details hierzu übersteigen zwar den Rahmen dieser Vorlesung, sie werden aber in den Vorlesungen »Parallelverarbeitung« und »Komplexitätstheorie« genauer untersucht.
2. Das »Wesen des Nichtdeterminismus« zu verstehen hat sich zu einer der Hauptaufgaben der Theoretischen Informatik gemausert. Wir haben in den Kapiteln 6 und 8 schon

gesehen, dass Nichtdeterminismus zumindest innerhalb der Theorie ein sehr nützliches Hilfsmittel darstellt. Es hat sich gezeigt, dass Nichtdeterminismus bei Platzklassen einfacher zu verstehen und zu untersuchen ist, als dies bei Zeitklassen der Fall ist (die uns ja eigentlich mehr interessieren). Ein Beispiel für den Erfolg dieser theoretischen Untersuchungen ist eine Konsequenz des Satzes von Immerman–Szelepcsényi: *Die Klasse der kontextsensitiven Sprachen (siehe Definition 3-13) ist abgeschlossen unter Komplementbildung*. Diesem Resultat sieht man wahrlich nicht an, dass hier viel Theorie über nichtdeterministischen logarithmischen Platz drinsteckt.

3. Die Klasse NL wird die erste natürliche Klasse sein (zur Frage, wie »natürlich« diese Klasse wirklich ist, siehe oben), für die wir ein (nun wirklich) natürliches *vollständiges* Problem angeben können: Das Erreichbarkeitsproblem REACH (auch GAP genannt) ist vollständig für NL. Der Beweis ist hier relativ einfach und soll als »Aufwärmübung« für die vertrackteren Beweise in den folgenden Kapiteln dienen.

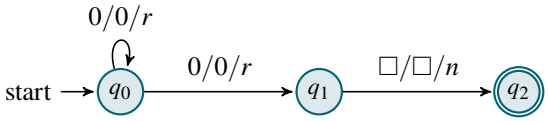
25.1 Die Klasse NL

25.1.1 Wiederholung: NTMs

Wiederholung: Maschinen, die sich nicht entscheiden können. 25-4

- Bei einer *nichtdeterministischen Turing-Maschine* kann es zu einem Zustand und gelesenen Zeichen *mehrere mögliche Nachfolgezustände* geben.
- Diese sind dann alle *möglich*, ...
- ...wodurch es viele *mögliche* Berechnungen gibt.
- Akzeptiert wird ein Wort, wenn es *wenigstens eine* akzeptierende Berechnung *gibt*.

Wiederholung: Eine einfache Maschine. 25-5



Einige Beobachtungen:

- Es gibt viele mögliche Berechnungen.
- Es führt aber immer nur höchstens eine zum akzeptierenden Zustand: Wenn man bei der letzten 0 vor dem Ende nach q_1 wechselt.
- Außerdem dürfen in dem Wort nur 0en vorkommen.
- Die Maschine akzeptiert also die Sprache $L(M) = \{0^n \mid n \geq 1\}$.

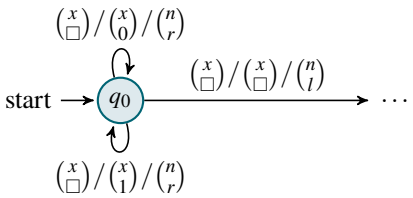
Zwei mögliche Berechnungen bei Eingabe 00010:

$$\begin{aligned}(q_0, \cdots \square \triangleright 00010 \square \cdots) &\vdash (q_0, \cdots \square 0 \triangleright 0010 \square \cdots) \\ &\vdash (q_0, \cdots \square 00 \triangleright 010 \square \cdots) \\ &\vdash (q_1, \cdots \square 000 \triangleright 10 \square \cdots).\end{aligned}$$

$$\begin{aligned}(q_0, \cdots \square \triangleright 00010 \square \cdots) &\vdash (q_0, \cdots \square 0 \triangleright 0010 \square \cdots) \\ &\vdash (q_1, \cdots \square 00 \triangleright 010 \square \cdots).\end{aligned}$$

Keine davon ist akzeptierend.

Wiederholung: Eine Maschine, die »nichtdeterministisch ein Wort rät«. 25-6



- Solange die Maschine in q_0 bleibt, kann sie *immer neue Zeichen auf das zweite Band schreiben*.

- Irgendwann ist sie aber damit fertig und wechselt den Zustand.
- Es gibt also *für jedes mögliche Wort* $w \in \{0,1\}^*$ eine Berechnung, so dass die Maschine q_0 verlässt und dieses Wort auf dem zweiten Band steht.
- *Man sagt (eigentlich fälschlicher Weise),* dass die Maschine *»nichtdeterministisch w rät«.*

25.1.2 Ressourceverbrauch von NTMs

25-7

Zeit- und Platzbeschränkte NTMs.

- Wir wollen nun nichtdeterministische Maschinen betrachten, die *zeit- oder platzbeschränkt* sind.
- *Problematisch* ist, dass die Maschinen *viele mögliche Berechnungen durchführen können,* die auch *unterschiedlich lange dauern können.*
- Es *liegt aber nahe,* einfach das *Maximum an Zeit- oder Platzverbrauch* über alle möglichen Berechnungen zu nehmen.

▶ Definition: Zeit- und Platzverbrauch einer NTMs

Sei M eine NTM mit einem Read-Only-Eingabeband. Wir definieren $s_M, t_M \colon \Sigma^* \to \mathbb{N} \cup \{\infty\}$ durch

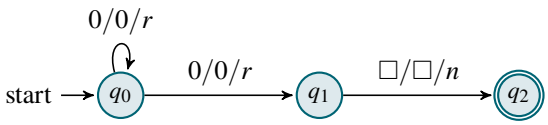
$$t_M(x) = \text{Maximale Länge einer Berechnung von } M \text{ bei Eingabe } x,$$
$$s_M(x) = \text{Maximaler Platzverbrauch einer Berechnung von } M \text{ bei Eingabe } x.$$

Wie immer zählt das Eingabeband beim Platzverbrauch nicht mit.
Die Funktionen $T_M, S_M \colon \mathbb{N} \to \mathbb{N}$ sind genauso wie im deterministischen Fall definiert (also jeweils der maximale Zeit-/Platzverbrauch bei Worten einer bestimmten Länge).

25-8

✎ Zur Übung

Wie lauten $s_M(0010)$ und $t_M(0010)$ von folgender Maschine?



25.1.3 Definition der Klasse

25-9

Die Klasse NL = unrealistischer Platzverbrauch + unrealistisches Maschinenmodell

▶ Definition: Die Klasse NL

Die Klasse NL enthält alle Sprachen A, für die es eine NTM M gibt, so dass:

1. M akzeptiert A,
2. $S_M \in O(\log n)$.

Bemerkungen:

- Offenbar gilt $\mathrm{L} \subseteq \mathrm{NL}$.
- Ob auch $\mathrm{L} \subsetneq \mathrm{NL}$ gilt, ist eine zentrale Forschungsfrage des Instituts für Theoretische Informatik.
  Der so genannte »informed guess« lautet »eher ja, aber vielleicht auch nein«.
- Wir zeigen später noch $\mathrm{NL} \subseteq \mathrm{P}$.
- Ganz ähnlich wie für L (siehe Lemma 24-21) beweist man, dass NL unter Reduktionen abgeschlossen ist.

25.2 NL-Vollständigkeit

25.2.1 Erreichbarkeitsprobleme

Zwei wichtige Probleme: Erreichbarkeit und Distanz. 25-10

▶ Definition: Erreichbarkeit- und Distanzproblem

$$\begin{aligned}\textsc{reach} &= \{\text{code}(G,s,t) \mid \text{es gibt einen Weg von } s \text{ nach } t \text{ in } G\},\\ \textsc{distance} &= \{\text{code}(G,s,t,k) \mid \text{es gibt einen Weg}\\ &\qquad \text{von } s \text{ nach } t \text{ in } G \text{ der Länge höchstens } k\}.\end{aligned}$$

Bei REACH und DISTANCE geht es um *gerichtete* Graphen.
Die *ungerichteten Varianten* bezeichnet man mit UREACH und UDISTANCE.
»REACH« hat diverse alternative Namen in der Literatur:

- GAP für »graph accessibility problem«,
- ST-CONN für »source to target connectivity problem«,

Public domain

Die Komplexität von Erreichbarkeit und Distanz. 25-11

▶ Satz
REACH $\in$ NL *und* DISTANCE $\in$ NL.

Beweis. Beginnen wir mit DISTANCE $\in$ NL.[1] Wir müssen also eine NTM M angeben, die bei jeder Eingabe code(G,s,t,k),

1. falls *es einen Weg* von s nach t der Länge (höchstens) k *gibt*, eine *akzeptierende Berechnung* aufweist,
2. falls es *keinen solchen Weg gibt,* auch *keine* akzeptierende Berechnung aufweist, und
3. auf allen Pfaden höchstens $O(\log n)$ Platz verbraucht.

Die Maschine M hat mehrere Arbeitsbänder:

- Ein Arbeitsband, auf dem (die Nummer) des *aktuellen Knotens* steht,[2]
- ein Arbeitsband, auf dem ein *Schrittzähler* steht,
- ein Arbeitsband, auf dem *der nächste Knoten »nichtdeterministisch geraten« wird.*

Auf allen Arbeitsbänder wird während bei allen Berechnungen nur logarithmisch viel Platz verbraucht werden.
Die Berechnung(en) von M verlaufen wie folgt:

1. Zunächst wird s auf das Aktueller-Knoten-Band kopiert und k auf das Schrittzähler-Band.
2. Dann wiederholt die Maschine Folgendes:
   2.1 Falls auf dem Aktueller-Knoten-Band gerade t steht, so akzeptiere.
   2.2 Falls der Schrittzähler 0 ist, verwerfe.
   2.3 »Rate nichtdeterministisch« einen Knoten auf dem Nächster-Knoten-Band.
   2.4 Durchlaufe das Eingabeband und überprüfe dabei, ob es eine Kante vom Knoten auf dem Aktueller-Knoten-Band zum Knoten auf dem Nächster-Knoten-Band gibt.
   2.5 Wenn ja, so kopiere das Nächster-Knoten-Band auf das Aktueller-Knoten-Band; sonst verwerfe.
   2.6 Verringere den Schrittzähler um Eins.

Wir behaupten, dass M das Gewünschte leistet:

1. Nach Konstruktion verbraucht M nur logarithmisch viel Platz (nämlich im Wesentlichen drei Zähler).
2. Gibt es einen Weg von s nach t der Länge höchstens k, so *gibt es eine Berechnung, bei der die nichtdeterministischen Entscheidungen gerade so ausfallen, dass die Knoten auf diesem Weg in der richtigen Reihenfolge auf das Aktueller-Knoten-Band geschrieben werden.*
   In diesem Fall akzeptiert die Maschine am Ende dieser Berechnung.
3. Gibt es keinen Weg von s nach t der Länge höchstens k, so wird bei *allen* Berechnungen niemals t auf dem Aktueller-Knoten-Band stehen, bevor die Maschine abbricht.

Um zu zeigen, dass auch REACH $\in$ NL gilt, kann man auf zwei Arten argumentieren:

1. Man wiederholt die Konstruktion, lässt aber den Schrittzähler einfach weg.
2. Man zeigt, dass REACH $\leq_{\text{m}}^{\log}$ DISTANCE gilt, und nutzt dann aus, dass NL unter Reduktionen abgeschlossen ist. □

Kommentare zum Beweis

[1] Schreiben wir erstmal auf, was eigentlich zu zeigen ist. Dies macht man normalerweise später nicht mehr.

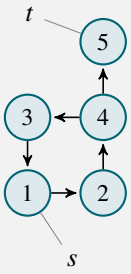
[2] »die Nummer von« wird im Folgenden weggelassen

Ein Beispiel, das zeigt, wie die NL-Maschine für das Distanzproblem arbeitet.

Start-Konfiguration

Eingabeband (nur lesen)

(1-2,2-4,4-3,4-5,3-1),s=1,t=5,k=3



Aktueller-Knoten-Band

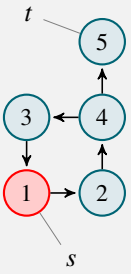
Schrittzähler-Band

Nächster-Knoten-Band

Initialisierung

Eingabeband (nur lesen)

(1-2,2-4,4-3,4-5,3-1),s=1,t=5,k=3



Aktueller-Knoten-Band
1

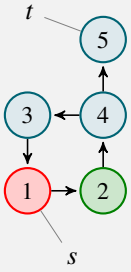
Schrittzähler-Band
3

Nächster-Knoten-Band

Ein möglicher 1. Schritt

Eingabeband (nur lesen)

(1-2,2-4,4-3,4-5,3-1),s=1,t=5,k=3



Aktueller-Knoten-Band
1

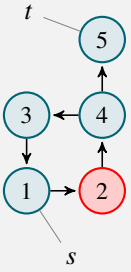
Schrittzähler-Band
2

Nächster-Knoten-Band
2

Kopie nach dem 1. Schritt

Eingabeband (nur lesen)

(1-2,2-4,4-3,4-5,3-1),s=1,t=5,k=3



Aktueller-Knoten-Band
2

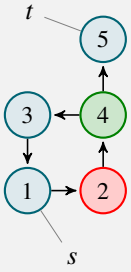
Schrittzähler-Band
2

Nächster-Knoten-Band

Ein möglicher 2. Schritt

Eingabeband (nur lesen)

(1-2,2-4,4-3,4-5,3-1),s=1,t=5,k=3



Aktueller-Knoten-Band
2

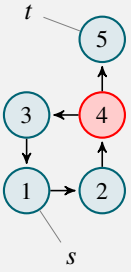
Schrittzähler-Band
1

Nächster-Knoten-Band
4

Kopie nach dem 2. Schritt

Eingabeband (nur lesen)

(1-2,2-4,4-3,4-5,3-1),s=1,t=5,k=3



Aktueller-Knoten-Band
4

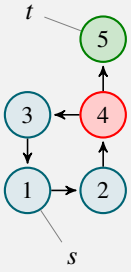
Schrittzähler-Band
1

Nächster-Knoten-Band

Ein möglicher 3. Schritt

Eingabeband (nur lesen)

(1-2,2-4,4-3,4-5,3-1),s=1,t=5,k=3



Aktueller-Knoten-Band
4

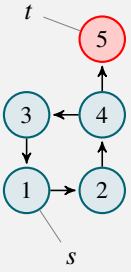
Schrittzähler-Band
0

Nächster-Knoten-Band
5

Kopie nach dem 3. Schritt und Akzeptanz

Eingabeband (nur lesen)

(1-2,2-4,4-3,4-5,3-1),s=1,t=5,k=3



Aktueller-Knoten-Band
5

Schrittzähler-Band
0

Nächster-Knoten-Band

25.2.2 Beweisverfahren I: Bootstrapping

Wie schwer ist das Erreichbarkeitsproblem?

25-13

- Wir haben gerade gesehen, dass REACH in NL liegt.
- Viele Probleme in NL sind auf REACH *reduzierbar*:
  - k-REACH für alle k, siehe Beispiel 24-15,
  - CONNECTED, siehe Übung 23.2
  - 2-SAT mittels einer unglaublich komplizierten und raffinierten Reduktion (wird am Ende der Vorlesung »Komplexitätstheorie« vorgestellt).

Wir haben damit *beste Voraussetzungen* dafür, dass REACH *vollständig* sein könnte für NL:

1. REACH liegt in NL und
2. viele (hoffentlich alle) Problem in NL lassen sich auf REACH reduzieren.

Das Erreichbarkeitsproblem ist vollständig für nichtdeterministischen logarithmischen Platz.

25-14

▶ Satz
REACH *ist vollständig für* NL.

Wie wir diesen Satz beweisen werden

- Wir wissen schon, dass REACH $\in$ NL.
- Wir müssen also »nurnoch« für jede Sprache $A \in$ NL eine Reduktion $A \leq_{\mathrm{m}}^{\log}$ REACH angeben.
- Dies ist *nicht ganz einfach*, da es ja *sehr viele unterschiedliche* Sprachen $A \in$ NL gibt.
- Für jedes $A \in$ NL gibt es aber eine »NL-Maschine« M, die A entscheidet.
- Der ganze Trick hinter der Reduktion ist nun, die Frage »Ist $x \in A$?« zu übersetzen in die Frage »Gibt es einen Weg von der Startkonfiguration s zu der akzeptierenden Konfiguration t im Konfigurationsgraphen von M?«

Der Konfigurationsgraph einer Turing-Maschine.

25-15

▶ Definition: Mögliche Konfiguration
Sei M ein Turing-Maschine mit Platzschranke $S_M : \mathbb{N} \to \mathbb{N}$. Eine Konfiguration von M heißt *möglich für Eingaben der Länge n*, wenn in der Konfiguration auf allen Arbeitsbänder höchstens $S_M(n)$ Zellen belegt sind.

▶ Definition: Konfigurationsgraph
Sei M eine Turing-Maschine und $x \in \Sigma^*$ eine Eingabe. Dann ist der *Konfigurationsgraph von M bei Eingaben der Länge n* der folgende Graph:

- Die *Knotenmenge V* ist die Menge aller *Konfigurationen, die für Eingaben der Länge n möglich sind*.
- Die *Kantenmenge E* enthält ein Paar (C, C') von Konfigurationen, wenn $C \vdash_M C'$.[1]

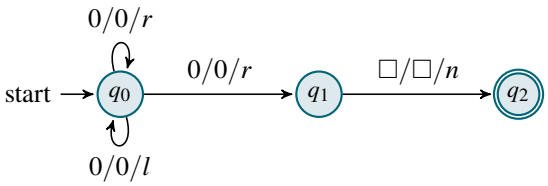
Kommentare zur Definition

[1] Man muss also von C in genau einem Berechnungsschritt zu C' kommen.

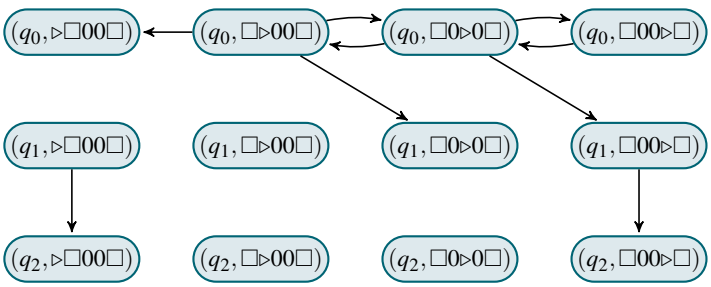
Beispiel eines (kleinen) Konfigurationsgraphen.

25-16

Die Maschine sei



Dann lautet der Konfigurationsgraph für die Eingabelänge 2:



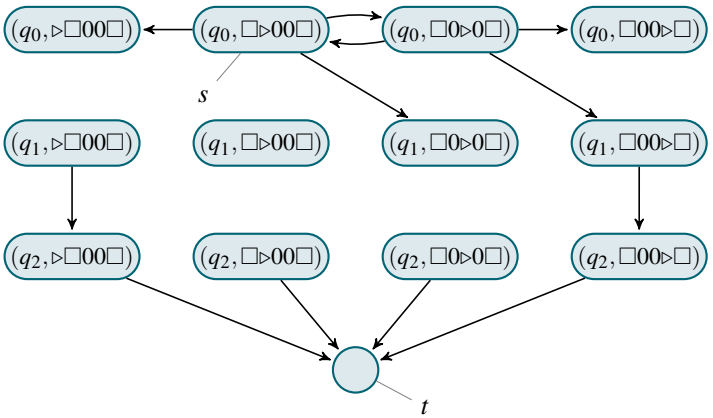
25-17

Beweis der Vollständigkeit des Erreichbarkeitsproblem.

Beweis der Vollständigkeit. Nach Satz 25-11 ist REACH ∈ NL.[1]
Wir beweisen die Schwere von REACH durch Bootstrapping.[2] Sei dazu $A \in \mathrm{NL}$ via M.[3]
Wir müssen eine Reduktion $A \leq_{\mathrm{m}}^{\log}$ REACH konstruieren.[4] Die Reduktion wandelt die Frage »Ist $x \in A$?« in die Frage »Ist $\mathrm{code}(G,s,t) \in$ REACH?« wie folgt um:

- Der Graph G ist im Wesentlichen der Konfigurationsgraph von M bei Eingaben der Länge $|x|$.
- Jedoch fügen wir noch einen Sonderknoten hinzu und Kanten von *allen akzeptierenden Konfigurationen zum Sonderknoten*.
- Der Startknoten s ist gerade $C_{\mathrm{init}}(x)$ und t ist der Sonderknoten.[5]

Ein Beispiel für die Maschine von eben und die Eingabe $x = 00$: Diese Eingabe würde von der Reduktion auf folgenden Graphen abgebildet werden:



[5] Sei nun $x \in A$. Dann *gibt es* eine Berechnung $C_{\mathrm{init}}(x) \vdash C_1 \vdash \cdots \vdash C_m$, die mit einer akzeptierenden Konfiguration C_m endet. Dann ist aber $C_{\mathrm{init}}(x) \to C_1 \to \cdots \to C_m \to t$ ein Pfad im Graphen G von s nach t. Also ist $\mathrm{code}(G,s,t) \in$ REACH.

[6] Sei nun $\mathrm{code}(G,s,t) \in$ REACH. Dann durchläuft der Pfad von s nach t genau eine Folge von Konfigurationen, die eine akzeptierende Berechnung von M bei Eingabe x darstellen. Also gilt $x \in A$.
Die Reduktion ist logspace-berechenbar. Dies liegt daran, dass man die Menge der *möglichen* Konfigurationen für ein Eingabewort x in Platz logarithmisch in $|x|$ ausgeben kann, indem man in einer Schleife alle möglichen Kopfpositionen und Bandinhalte durchgeht.
Weiterhin ist es auch möglich, in logarithmischem Platz zu entscheiden, ob M in einem Schritt von einer gegebenen Konfiguration C zu einer anderen Konfiguration C' wechseln kann. □

[1] Es bleibt a
Problems

[2] Der Name »Bootstrapping« wird gleich noch erklärt.

[3] Dies ist eine knappe Formulierung für »M ist eine logspace-beschränkte NTM, die A akzeptiert«

[4] Beweisrezept »Reduzierbarkeit beweisen«

[5] Das war schon die gesamte Reduktion

[5] Erste Beweisrichtung der Korrektheit der Reduktion

[6] Zweite Beweisrichtung

25-18

Beweisrezept: *Bootstrapping*

Ziel

Beweisen, dass X vollständig ist für eine Zeit- oder Platzklasse C.

Rezept

Die Methode benutzt man, wenn man *noch kein* vollständiges Problem für C kennt.

1. Zeige zunächst $X \in C$, typischerweise durch Angabe einer Maschine mit dem korrekten Zeit- oder Platzverbrauch.
2. Fahre fort mit: »Sei nun $A \in C$ beliebig. Sei M eine Maschine, die A in Zeit/Platz XYZ entscheidet. Wir zeigen, dass dann $A \leq_{\mathrm{m}}^{\log} X$ gilt.«
3. Benutze das Beweisrezept »Reduktionen beweisen«, um die Reduktion zu bauen.

25-19

Ein wichtige Folgerung.

▶ Satz
NL ⊆ P

Beweis. Da REACH ∈ P via beispielsweise einer Maschine, die eine Breitensuche implementiert, ist nach dem Repräsentationslemma, Lemma 24-22, auch ganz NL in P. □

25.2.3 Beweisverfahren II: Reduktionsmethode

Der Nachweis der Vollständigkeit weiterer Probleme ist leicht. 25-20

- Wir haben gesehen, dass REACH vollständig ist für NL.
- Wir haben auch schon gesehen, dass DISTANCE $\in$ NL gilt.
- Da REACH »sicher nicht leichter« ist als DISTANCE, sollte DISTANCE *auch vollständig sein für* NL.
- Dies kann man auch wieder durch Bootstrapping beweisen, *das wäre aber sehr umständlich.*

Folgender Satz zeigt, wie es viel einfacher geht:

▶ Satz
Sei X_{alt} vollständig für C und sei $X_{neu} \in C$. Falls nun $X_{alt} \leq_m^{\log} X_{neu}$, so ist X_{neu} ebenfalls vollständig für C.

Beweis. Nach Voraussetzung ist $X_{\text{neu}} \in C$. Da alle Sprachen in C reduzierbar sind auf X_{alt} (wegen der Vollständigkeit von X_{alt}) und da X_{alt} auf X_{neu} reduzierbar ist, sind (wegen der Transitivität der Reduktion) auch alle Sprachen in C auf X_{neu} reduzierbar. □

Beweisrezept: *Reduktionsmethode* 25-21

Ziel
Beweisen, dass X_{neu} vollständig ist für eine Klasse C.

Rezept
Die Methode benutzt man, wenn man *bereits vollständige Probleme für C kennt.*

1. Zeige zunächst $X_{\text{neu}} \in C$, typischerweise durch Angabe einer Maschine mit dem korrekten Zeit- oder Platzverbrauch.
2. Suche ein geeignetes für C vollständiges Problem X_{alt} aus.
3. Benutze das Beweisrezept »Reduktionen beweisen«, um $X_{\text{alt}} \leq_m^{\log} X_{\text{neu}}$ zu beweisen.

25-22

Anwendung der Reduktionsmethode I

▶ **Satz**
DISTANCE *ist vollständig für* NL.

Kommentare zum Beweis

Beweis. Nach Satz 25-11 ist DISTANCE ∈ NL.
Weiter ist REACH vollständig für NL und REACH $\leq_m^{\log}$ DISTANCE via der folgenden Reduktion: Bilde (G,s,t) ab auf (G,s,t,n), wobei n die Anzahl der Knoten in G ist.[1] □

[1] Der Beweis der Korrektheit der Reduktion könnte etwas länger ausfallen. . .

25-23

Anwendung der Reduktionsmethode II

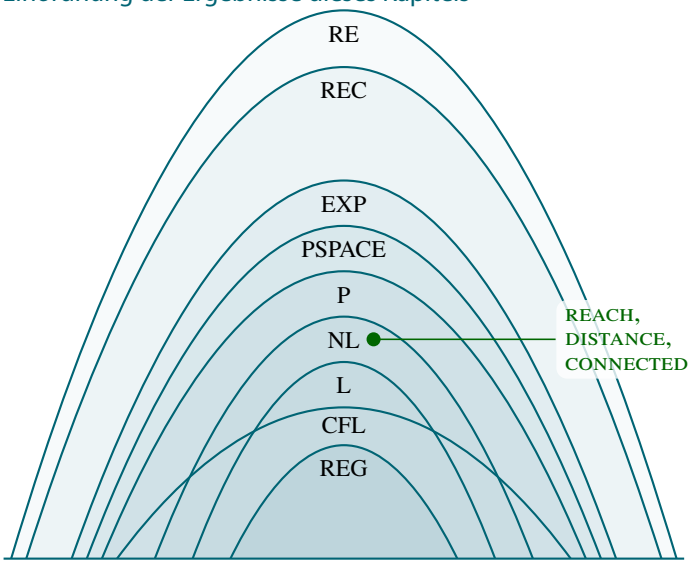
▶ **Satz**
CONNECTED *ist vollständig für* NL.

Beweis. In Übung 23.2 haben wir CONNECTED $\leq_m^{\log}$ REACH gezeigt. Aufgrund der Abgeschlossenheit von NL unter Reduktionen folgt auch CONNECTED ∈ NL.
Weiter ist REACH vollständig für NL und REACH $\leq_m^{\log}$ CONNECTED wie in Übung 23.1 gezeigt. □

25.2.4 Übersicht der Klassenhierarchie

25-24

Einordnung der Ergebnisse dieses Kapitels



Zusammenfassung dieses Kapitels

25-25

1. Erreichbarkeitsprobleme für *gerichtete Graphen* sind *vollständig für* NL.
2. Für das »erste« Problem, muss man Vollständigkeit für eine Klasse aufwändig mittels *Bootstrapping* beweisen.
3. Danach kann man mittels der *Reduktionsmethode* viel *leichter* nachweisen, dass *weitere* Probleme auch vollständig sind.
4. Deshalb ist es um so leichter, Vollständigkeit zu zeigen, je mehr vollständige Probleme man bereits kennt.

Zum Weiterlesen

[1] R. Reischuk. *Unterlagen zum Lehrmodul Theoretische Informatik,* Vorlesungsskript, 2008. Kapitel 8.3.

26-1

Kapitel 26
P-Vollständigkeit
Die Grenzen der Parallelisierbarkeit

26-2

Lernziele dieses Kapitels

1. Mathematische Definition von Schaltkreisen kennen
2. Das Circuit-Value-Problem CVP kennen
3. Beweis der P-Vollständigkeit von CVP verstehen

Inhalte dieses Kapitels

Worum es heute geht

Im vorigen Jahrhundert hatte die Parallelverarbeitung bereits große praktische Bedeutung – aber eben nur im Bereich der doch eher seltenen Supercomputer, die ihrerseits in der Regel nur für sehr spezielle numerische Probleme eingesetzt wurden. Es ergab damals keinen Sinn, Feld-Wald-und-Wiesen-Software für Parallelrechner fit zu machen – es gab einfach keine auf den Schreibtischen der Kunden. Umgekehrt versuchten Hardware-Hersteller auch nicht, mehrere Prozessoren in die Computer einzubauen, da diese von Standardsoftware nicht genutzt wurde. Ein klassisches Henne-Ei-Problem lag vor und die Parallelverarbeitung verfiel in einen etwas unruhigen Dornröschenschlaf.

Aus diesem Schlaf ist sie irgendwann kurz nach der Jahrtausendwende wachgeküsst worden – allerdings nicht von einem Prinzen, sondern von Multi-Core-Prozessoren. Heute kann man beim Entwurf auch eines einfachen Standardprogramms davon ausgehen, dass in der Regel mehrere Rechenkerne zur Verfügung stehen werden. Damit drängelt sich sowohl in der Praxis wie auch in der Theorie ein Problem in den Vordergrund, das bis jetzt doch eher im Abseits gestanden hat: Wie gut lassen sich Standardprobleme eigentlich parallelisieren? Mit anderen Worten: Wenn ich 16 Kerne zur Verfügung habe, kann ich das Problem dann auch 16 Mal so schnell lösen?

Die Frage, welche Probleme wie gut parallelisierbar sind, wird in diesem Kapitel nicht beantwortet werden – schließlich gibt es eine eigene Vorlesung »Parallelverarbeitung« in der es um nichts anderes geht. Jedoch können wir jetzt schon festhalten, dass man parallele Programme immer in sequentielle Programme umwandeln kann, indem parallelen Anweisungen einfach nacheinander ausgeführt werden. Deshalb können parallele Programme auf einem Rechner mit p Kernen *höchsten* um den Faktor p schneller sein als sequentielle Programme. In der Praxis ist p eine kleine Zahl, irgendetwas zwischen 2 und vielleicht 100 – ein Parallelrechner mit einer Million Kernen ist doch eher die Ausnahme. Dies bedeutet aber, dass man mit Parallelrechnern »auch nur« Probleme in P effizient wird lösen können. Der Geschwindigkeitsvorteil, den eine Parallelisierung bringt, wird bei einem Exponentialzeit-Problem schon durch eine kleine Erhöhung der Eingabegröße sofort zu Nichte gemacht.

Die Frage, welche Probleme gut parallelisierbar sind, ist also die Frage, welche Problem *in* P gut parallelisierbar sind. Wenn man etwas darüber nachdenkt, so stellt man schnell fest, dass fast alle Probleme, die einem so spontan einfallen, irgendwie ganz gut parallelisierbar scheinen. Da darf die Frage erlaubt sein, ob nicht vielleicht einfach *alle* Probleme in P sich auf einem Parallelrechner viel schneller lösen lassen? Die Beantwortung einer solchen Frage

scheint wieder nicht ganz einfach, da es ja doch recht viele und noch dazu sehr unterschiedliche Probleme in der Klasse P gibt.

An dieser Stelle kommt nun der Titel dieses Kapitels ins Spiel: Statt uns mit unendlich vielen komischen Problemen herumzuschlagen, könnten wir uns auf die Parallelisierbarkeit nur *eines* Problems konzentrieren, wenn wir ein *vollständiges* Problem für P finden. Ein solches Problem gibt es: Das *Circuit-Value-Problem,* welches im Folgenden eingehend vorgestellt wird.

Was weiß man nun über die Parallelisierbarkeit des Circuit-Value-Problems? Trotz jahrelanger Forschung ist es leider weder gelungen zu zeigen, dass sich dieses Problem gut parallelisieren lässt (und damit auch alle anderen Problem in P aufgrund der Vollständigkeit), noch dass es sich *nicht* gut parallelisieren lässt. Nach der P-NP-Frage, der ja das gesamte Kapitel 28 gewidmet ist, ist die Frage, wie gut sich das Circuit-Value-Problem parallelisieren lässt, wohl die zweitwichtigste Frage der Theoretischen Informatik.

26.1 Einleitung

26.1.1 P-Vollständigkeit = gerade so lösbar

26-4

Welche Probleme in P sind leicht, welche sind schwierig?

Viele Probleme in P liegen auch in NL:

- Grundrechenarten
- Sortieren
- Suchen
- Auswerten von arithmetischen Ausdrücken
- Erreichbarkeit, Distanz, Zusammenhang

All diesen Problemen ist gemein, dass sie sich *gut parallelisieren lassen*.

Leitfrage

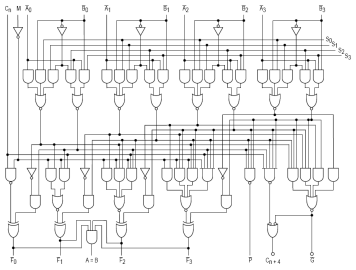
Gibt es *vollständige* Probleme in P?
Dies wären Probleme, die »am ehesten« *nicht parallelisierbar* sind.

26.1.2 Schaltkreise

26-5

Zwei Schaltkreise.

Copyright by Qurren, GNU Free Documentation License



Copyright by Stephane Tsacas, GNU Free Documentation License

26-6

Die mathematische Definition eines Schaltkreises.

▶ **Definition:** Schaltkreis

Ein *Schaltkreis* ist ein gerichteter azyklischer Graph mit folgenden Eigenschaften:

- Die *Knoten* des Graphen nennt man *Gatter.*
- Die *Kanten* des Graphen nennt man *Drähte.*
- Jedes Gatter hat einen der folgenden *Gattertypen*:
  - *Eingabegatter* haben Eingrad 0.
  - *Ausgabegatter* haben Ausgrad 0 und Eingrad 1.
  - *Konstante Gatter* haben Eingrad 0.
  - *Negationsgatter* haben Eingrad 1.
  - *Und-Gatter* haben beliebigen Grad.
  - *Oder-Gatter* haben beliebigen Grad.

- Die Eingabegatter benennen wir x_1 bis x_n.
- Die Ausgabegatter benennen wir y_1 bis y_m.

Es gibt keine feste Notation für Schaltkreise, sie ist Geschmackssache.

Wie eine Berechnung mittels eines Schaltkreises funktioniert.

26-7

- Wir wollen Schaltkreise natürlich benutzen, um Eingaben in Ausgaben umzuformen.
- Dazu *legt man die Eingabe an den Eingängen an* und schaut, wie die Berechnung vorangeht.
- Jedes Gatter kann erst dann beginnen zu arbeiten, wenn alle seine (lokalen) Eingänge einen Wert vorweisen.

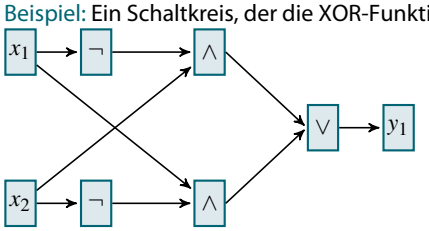
▶ Definition: Berechnete Funktion
Sei C ein Schaltkreis mit n Eingängen und m Ausgängen. Dann berechnet C eine Funktion $f_C\colon \{0,1\}^n \to \{0,1\}^m$ wie folgt:

- Seien b_1 bis b_n Bits.
- Wir definieren rekursiv den *Wert* eines Gatter als
  - b_i für Eingabegatter x_i;
  - den Wert des Vorgängers für Ausgabegatter;
  - i für konstante i-Gatter;
  - $1 - v$ für Negationsgatter, wenn v der Wert des Vorgängers ist;
  - das Maximum aller Vorgängergatter für Oder-Gatter und
  - das Minimum aller Vorgängergatter für Und-Gatter.
- Sind v_1 bis v_m dann die Werte der Ausgabegatter, so ist $f_C(b_1 \dots b_n) = v_1 \dots v_m$.

Beispiel eines Schaltkreises

26-8

Beispiel: Ein Schaltkreis, der die XOR-Funktion berechnet.



26.2 P-Vollständigkeit

26.2.1 Definition: CVP

Eine scheinbar eher harmloses Problem.

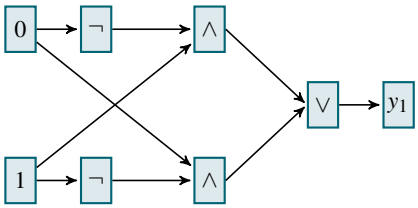
26-9

▶ Definition: Circuit-Value-Problem, CVP
Die Sprache CVP enthält die Codes aller Schaltkreise C mit folgenden Eigenschaften:

- C hat *keine Eingabegatter* (aber gerne konstante 0- und 1-Gatter),
- C hat genau *ein Ausgabegatter*,
- wertet man C aus, so ist das Resultat an diesem Ausgabegatter gerade 1.

Beispiel

Der Code des folgenden Schaltkreises ist ein Element von CVP:



26.2.2 Satz: CVP ist P-vollständig

26-10

Das Hauptresultat.

▶ Satz
CVP *ist* P-*vollständig*.

Beweisideen

- Da wir *noch keine* P-vollständigen Problem kennen, müssen wir das *Rezept »Bootstrapping«* anwenden.
- Wir zeigen also zuerst, dass CVP $\in$ P gilt.
- Dann zeigen wir für *jede Sprache* $A \in$ P, dass wir die Frage »Ist $x \in A$?« mittels einer geeigneten Reduktion in die Frage »Ist code$(C) \in$ CVP?« umwandeln können.
- Die *Idee* hinter dem Schaltkreis C ist, dass er *in Schichten* die Berechnung einer Turing-Maschine *nachvollzieht.*

26-11

CVP ist in P.

▶ Lemma: Erster Beweisteil
CVP $\in$ P.

✎ Zur Übung
Skizzieren Sie einen Algorithmus, der CVP in polynomieller Zeit löst.

26-12

Reduktion beliebiger Probleme aus P.

▶ Lemma: Zweiter Beweisteil
Sei $A \in$ P *ein beliebiges Problem. Dann gilt* $A \leq_{\mathrm{m}}^{\log}$ CVP.

Beweisideen

Im Folgenden sollen folgende Punkte angesprochen werden:

1. Vorbereitungen (Einführen von Namen, Vereinfachungen)
2. Aufbau des Schaltkreises
3. Aufbau der Boxen im Schaltkreis
4. Eigenschaften des Schaltkreises
5. Wie schwierig ist es, den Schaltkreis zu konstruieren?

26-13

Setting the Stage: Die im Folgenden verwendeten Namen.

Wir legen zunächst einige wichtige Bezeichner fest:

- A Die Sprache in P, die wir auf CVP reduzieren wollen.
- M Eine Polynomialzeitmaschine, die A entscheidet.
- p Ein Polynom, das die Laufzeit von M beschränkt.
- x Eine Eingabe, für die die Reduktionsfunktion die Frage »Ist $x \in A$?« in die Frage »Ist code$(C) \in$ CVP?« umwandeln soll.
- n Die Länge von x, also $n = |x|$.
- C Der Schaltkreis, der von der Reduktion berechnet wird und der genau dann zu 1 auswerten soll, wenn $x \in A$ – was wiederum genau dann der Fall ist, wenn M das Wort x akzeptiert.

26-14

Von der Maschine zum Schaltkreises.

Wir beginnen mit ein paar Vereinfachungen:

- Die Maschine M habe nur *ein Band*.
- Das Band der Maschine M sei *einseitig beschränkt*.
- Es gibt *genau eine* akzeptierende Endkonfiguration.
- Bei dieser Endkonfiguration ist der *Kopf am Anfang*.

✎ Zur Diskussion
Wie kann man jeweils Maschinen, die eine der obigen Eigenschaften nicht haben, umwandeln, so dass die Eigenschaften erfüllt werden?

Grober Aufbau des Schaltkreises.

26-15

Wir bauen den Schaltkreis C wie folgt:

- Er besteht aus $p(n)$ *Schichten*.
- An den Ausgabegattern der k-ten Schicht liegt *kodiert die k-te Bandkonfiguration an*.
- An den Eingabegattern der k-ten Schicht liegt *kodiert die* $(k-1)$*-te Bandkonfiguration an*.

Die Kodierung einer Bandkonfiguration funktioniert wie folgt:

- Um eine Bandzelle zu kodieren, benutzen wir mehrere Leitungen.
- Ist das Bandalphabet Γ, so können in einer Zelle $|\Gamma|$ verschiedene Symbole stehen; wir benutzen deshalb $\log|\Gamma|$ Leitungen zur Kodierung des Zelleninhalts.
- Zusätzlich benutzt man noch pro Zelle eine Leitung, um zu signalisieren, ob der Kopf dort ist.
- Schließlich benutzen wir noch $\log|Q|$ Leitungen, um den aktuellen Zustand der Turingmaschine zu kodieren.

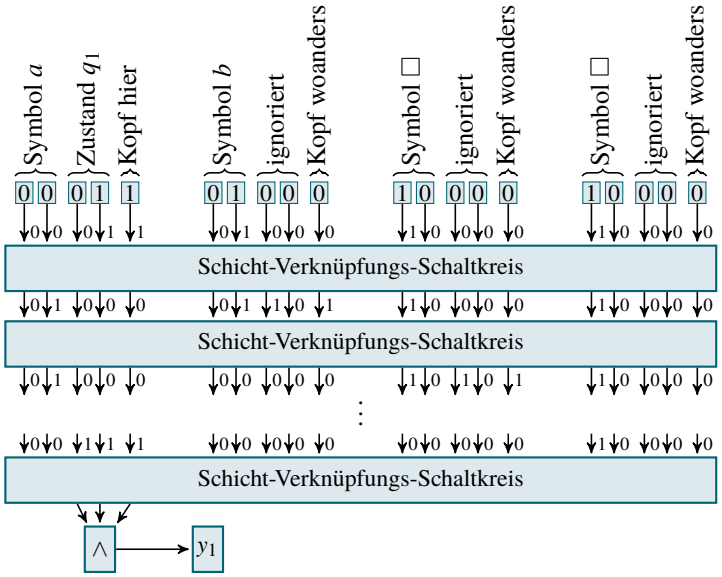
Eine Beispielsituation.

26-16

- Die Symbole des Bandalphabets $\Gamma = \{a, b, \square\}$ werden kodiert durch 00, 01, 10.
- Es gibt vier Zustände $\{q_0, q_1, q_2, q_3\}$, kodiert durch 00, 01, 10 und 11.
- Der initiale Bandinhalt ist $ab\square\square$, der initiale Zustand ist q_1 und der Kopf ist initial auf erstem Zeichen (dem a).
- Nach einem Schritt ist der Bandinhalt $bb\square\square$, der Zustand ist q_2 und der Kopf auf dem zweiten Zeichen.
- Nach zwei Schritten ist der Bandinhalt $ba\square\square$, der Zustand ist q_2 und der Kopf auf dem dritten Zeichen.
- Nach $p(n)-1$ Schritten ist der Bandinhalt $aaa\square$, der Zustand ist q_3 und der Kopf auf dem ersten Zeichen.
- Akzeptierender Zustand ist genau q_3.

Visualisierung der Schichtenstruktur des Schaltkreises.

26-17



Was leistet der Schaltkreis?

26-18

- Betrachtet man die Leitungen, die in die *erste* Schicht hineinführen, so kodieren sie genau die Anfangskonfiguration.
- Betrachtet man die Leitungen, die in die *zweite* Schicht hineinführen, so kodieren sie genau den Bandinhalt nach einem Schritt.
- Betrachtet man die Leitungen, die in die *dritte* Schritt hineinführen, so kodieren sie genau den Bandinhalt nach zwei Schritten.
- Betrachtet man die Leitungen, die in die $p(n)$*-te* Schritt hineinführen, so kodieren sie genau den Bandinhalt der Endkonfiguration.
- Schaltet man also noch einen winzigen Schaltkreis nach, der überprüft, ob diese Endkonfiguration akzeptierend ist, so leistet der Schaltkreis das Gewünschte:
  Er wertet zu 1 aus genau dann, wenn die Maschine die Eingabe akzeptiert.

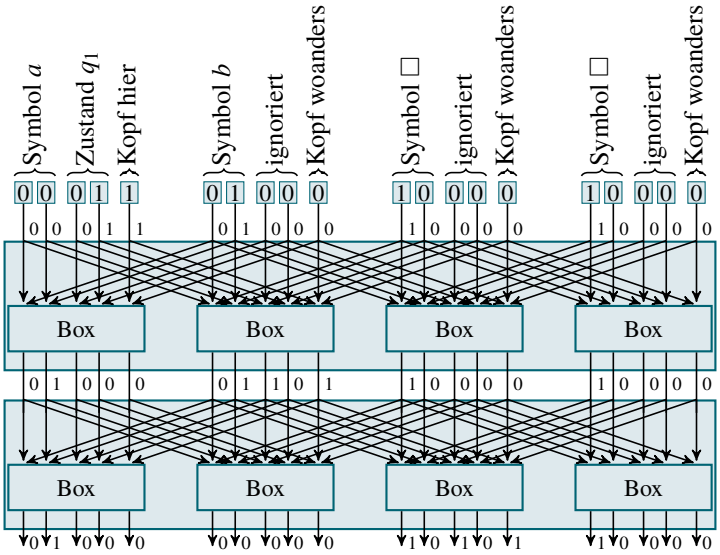
26-19

Was in den Schicht-Verknüpfungs-Schaltkreisen passiert.

- Wir wollen einen (flachen) Schaltkreis angeben, der *Schicht i mit Schicht $i+1$ verknüpft*.
- Dieser Schaltkreis besteht aus $p(n)$ vielen *Boxen*.
- Jede Box kümmert sich darum, den Bandinhalt einer bestimmten Zelle z zu berechnen.
- Dazu muss sie lediglich *die Inhalte der Zelle z in der vorherigen Schicht kennen, sowie die Inhalte der Zelle links und rechts davon*.

26-20

Aufbau eines Schicht-Verknüpfungs-Schaltkreises.



26-21

Wie schwierig ist der Schaltkreis zu bauen?

Was wir erreicht haben:

- Zu jeder beliebigen Eingabe x können wir einen Schaltkreis konstruieren, der genau dann zu 1 auswertet, wenn M das Wort x akzeptiert.

Was fehlt:

- Wie schwierig ist es, den Schaltkreis zu konstruieren?

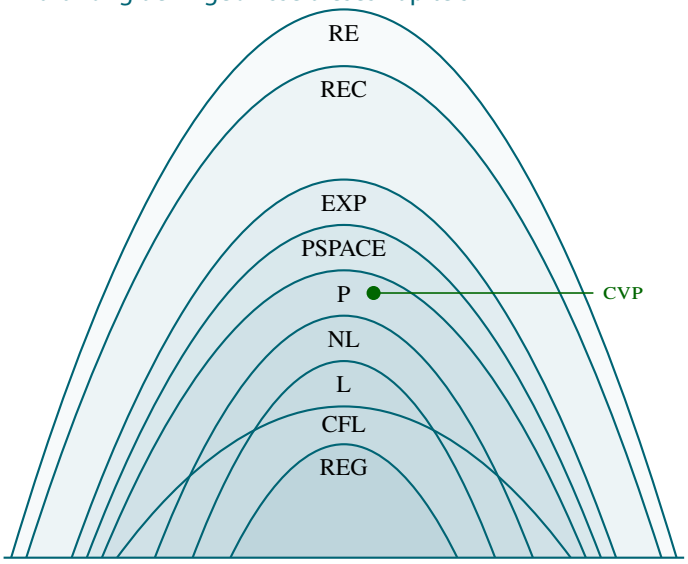
Antwort hierauf:

- Dies ist einfach, da *alle Boxen identisch sind*.
- Der Schaltkreis besteht im Wesentlichen aus $p(n)^2$ Boxen, die in einfacher Weise verdrahtet sind.
- Es ist leicht, in logarithmischem Platz in einer Schleife $p(n)^2$ Boxen (fester Größe!) auszugeben.
- Die Verdrahtung ist etwas fummeliger, aber auch nicht schwierig.

26.2.3 Übersicht der Klassenhierarchie

26-22

Einordnung der Ergebnisse dieses Kapitels



Zusammenfassung dieses Kapitels

26-23

1. Mathematisch ist ein *Schaltkreis* ein gerichteter azyklischer Graph mit verschiedenen Gattertypen.
2. Die Sprache CVP enthält alle Codes von Schaltkreisen ohne Eingabegatter, die zu 1 auswerten.
3. P-*vollständige Probleme* sind die schwierigsten Probleme in P und Kandidaten für Probleme, die weder in NL liegen noch sich gut parallelisieren lassen.
4. Das Circuit-Value-Problem ist P-vollständig.
5. Im Beweis konstruiert man einen Schaltkreis, der die Berechnung einer Turing-Maschine nachvollzieht.

27-1

Kapitel 27
NP-Vollständigkeit
Die Grenzen der Effizienz

27-2

Lernziele dieses Kapitels

1. Nichtdeterministische Zeit verstehen
2. Das Erfüllbarkeitsproblem SAT kennen
3. Satz von Cook und seinen Beweis verstehen

Inhalte dieses Kapitels

Worum es heute geht

Ist es moralisch verwerflich, zu betrügen? Betrug wird im Strafrecht als ein doch eher schweres Delikt angesehen und gegebenenfalls hart bestraft. Mensch sollten nicht betrügen. Bei *Maschinen* ist die Sachlage komplexer: Es ist nicht unmoralisch, wenn Maschinen sich ein wenig »helfen« lassen – es führt nur zu Berechnungsmodellen, die uns in der Praxis nicht viel nützen. Jeder möge nun für sich selbst entscheiden, was schlimmer ist (dies ist natürlich eine moralische Frage).

Wie können Maschinen nun schummeln? Ein besonders raffiniertes Verfahren ist die Benutzung von »nichtdeterministischen Hinweisen«. Die Situation ist ähnlich wie bei einem Studenten, der eine Klausur schreibt und dem kleine Zettelchen zugeflogen kommen, auf denen Dinge stehen wie »addiere erst die dritte und vierte Zahl« oder »schaue auf Seite 256 im Skript nach, da steht die Lösung«. Das Problem ist nur, dass diese Hinweise nicht sonderlich verlässlich sind: Es kann sein (nichts genaues weiß man nicht), dass der ominöse Zettelproduzent dummerweise eine *andere* Klausur schreibt. Dann laufen die Hinweise natürlich ins Leere. In diesem Fall muss die schummelnde Maschine in der Lage sein zu erkennen, dass hier etwas nicht mit rechten Dingen zugeht: Formal soll sie (schnell) die korrekte Klausurantwort produzieren bei *geeigneten* Hinweisen; hingegen soll sie (schnell) verwerfen, wenn die Hinweise sie nicht zur korrekten Lösung führen werden.

Solche nichtdeterministischen Hinweis sind etwas zwielichtig, schwer zu definieren und schwer zu motivieren. Trotzdem benötigen wir sie, um die wohl wichtigste und bestuntersuchte Klasse der Komplexitätstheorie zu definieren: NP. Von dieser Klasse haben selbst Nichtinformatiker oft schon im Rahmen des P-NP-Problems gehört. Bei diesem Problem geht es letztendlich um die Frage, ob man jede Berechnung mit Schummelei in eine genauso effiziente Berechnung ohne Schummeln umwandeln kann.

Lug und Betrug bei Berechnungen kann man übrigens auch noch doller treiben (was wir in dieser Vorlesung aber nicht machen werden, dies ist dem Master-Studium vorbehalten, wo Sie auch moralisch hoffentlich schon gefestigter sind): Statt nur rätselhafte und in der Regel falsche Hinweise zu bekommen für eine Berechnung, kann man auch gleich ein *mächtiges Orakel* fragen. Solche Orakel kann die Maschine beliebig oft befragen, ob verschiedene Worte Element einer bestimmten Sprache sind – und das beste ist, dass die Frage nichts kosten. Offenbar werden viele Problem *sehr* einfach zu beantworten, wann man ein mächtiges Orakel befragen darf: So ist das Problem SAT nun wirklich leicht zu entscheiden, wenn

man ein »SAT-Orakel« befragen darf. Wenn man aber etwas darüber nachdenkt, so stellt man schnell fest, dass bestimmte Probleme *selbst dann noch schwer sind, wenn man ein Orakel befragen darf.* So ist es ein wichtiges offenes Problem, ob man beispielsweise einen effizienten Algorithmus für das Brettspiel »Go« programmieren kann, wenn man ein SAT-Orakel zur Verfügung hat.

27.1 Die Klasse NP

27.1.1 Ressourceverbrauch von NTMs

Wiederholung: Zeit- und Platzbeschränkte NTMs. 27-4

▶ **Definition:** Zeit- und Platzverbrauch einer NTMs (Wiederholung von 25-7)
Sei M eine NTM mit einem Read-Only-Eingabeband. Wir definieren $s_M, t_M \colon \Sigma^* \to \mathbb{N} \cup \{\infty\}$ durch

$$t_M(x) = \text{Maximale Länge einer Berechnung von } M \text{ bei Eingabe } x,$$
$$s_M(x) = \text{Maximaler Platzverbrauch einer Berechnung von } M \text{ bei Eingabe } x.$$

Wie immer zählt das Eingabeband beim Platzverbrauch nicht mit.
Die Funktionen $T_M, S_M \colon \mathbb{N} \to \mathbb{N}$ sind genauso wie im deterministischen Fall definiert (also jeweils der maximale Zeit-/Platzverbrauch bei Worten einer bestimmten Länge).

27.1.2 Definition der Klasse NP

Die Klasse zentrale Klasse der Komplexitätstheorie: NP 27-5

▶ **Definition:** Die Klasse NP
Die Klasse NP enthält alle Sprachen A, für die es eine NTM M gibt, so dass:

1. M akzeptiert A,
2. $T_M \in O(n^{O(1)})$.[1]

[1] »$n^{O(1)}$« bedeutet so viel wie »irgendein Polynom«

Kommentare zur Definition

Bemerkungen:

- Offenbar gilt P $\subseteq$ NP.
- Ob auch P $\subsetneq$ NP gilt, ist das berühmte »P-NP-Problem«.
  Der »informed guess« lautet »eher ja«.

27.1.3 Beispiele von Sprachen in NP

Welche Sprachen sind in NP? 27-6

- *Sehr viele Sprachen,* insbesondere auch *viele Sprachen aus praktischen Anwendungen* liegen in NP.
- Im nächsten Kapitel wird noch etwas vertieft, welche Art von Sprachen genau in NP liegen.

Beispiele: Einige wichtige Sprachen in NP

- SAT
- CLIQUE
- INDEPENDENT-SET
- GRAPH-ISOMORPHISM: die Eingabe sind zwei Graphen, die Frage ist, ob diese Graphen isomorph sind (= ob sie nur unterschiedlich »gezeichnet« wurden).
- TRAVELLING-SALESPERSON (TSP): die Eingabe ist ein Graph mit »Kosten« an jeder Kante und ein »Budget«, die Frage ist, ob man eine Rundreise findet, so dass die Gesamtkosten das Budget einhalten.

27-7

Beispielbeweise dafür, dass die Sprachen in NP liegen.

Lemma
SAT $\in$ NP.

Kommentare zum Beweis

Beweis. Wir müssen eine polynomiell zeitbeschränkte NTM konstruieren, die bei Eingabe φ Folgendes leistet:

- Falls φ *erfüllbar* ist, so muss sie *wenigstens einen akzeptierenden Pfad* besitzen.
- Falls φ hingegeben *eine Kontradiktion* ist, so müssen alle Pfade verwerfen.

Die Maschine arbeitet ist zwei *Phasen*:

1. In der *ersten Phase* »rät sie nichtdeterministisch« eine Belegung β für die Variablen auf einem Arbeitsband.[1]
2. In der *zweiten Phase* überprüft sie, ob $\beta \models \varphi$, ob also φ unter der Belebung wahr wird. Wenn ja, so akzeptiert sie; sonst verwirft sie.

[1] Zur Erinnerung: Dies bedeutet, dass es *für jede mögliche Belegung β eine Berechnung gibt.*

Nun muss man sich noch überzeugen, dass die Maschine

1. korrekt arbeitet und
2. alle Berechnungen polynomiell zeitbeschränkt sind.

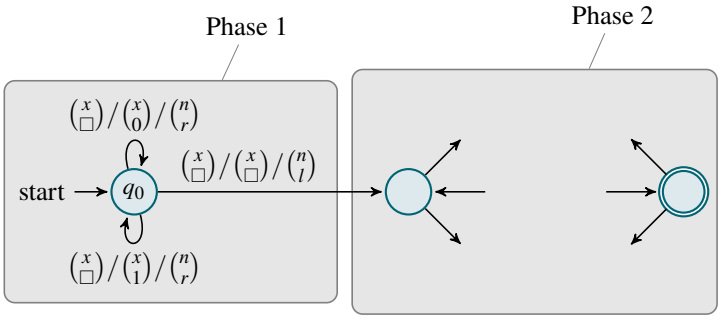
Dies sieht man so:

1. Ist die Formel erfüllbar – sagen wir mittels β –, so *gibt es* eine Berechnung, bei der genau dieses β auf das Arbeitsband geschrieben wird. Dann wird die Auswertung in der zweiten Phase aber positiv verlaufen und φ wird akzeptiert.
   Ist die Formel eine Kontradiktion, so wird die Auswertung in der zweiten Phase *immer* zum Verwerfen führen.
2. Das »Raten« von β dauert nur linear lange in der Länge von β, es dauert also höchstens so lange wie die Länge der Eingabe. Das Auswerten einer Formel ist danach recht schnell, auf jeden Fall aber in Zeit $O(n^2)$ machbar. □

27-8

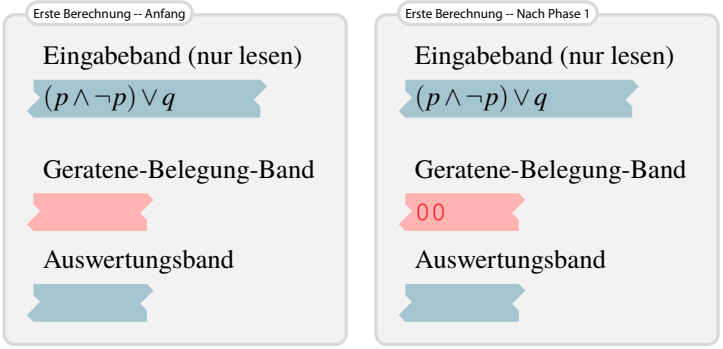
Funktionsweise der Maschine an einem Beispiel.

Die Maschine ist *grob* wie folgt aufgebaut:



27-9

Zwei typische Berechnungen der Maschine



Kurzbeweise dafür, dass eine Sprache in NP liegen.

▶ Lemma

CLIQUE $\in$ NP.

Beweis. Eine NTM für CLIQUE arbeitet bei Eingabe eines Graphen $G = (V, E)$ und einer Zahl k wie folgt:

1. In einer ersten Phase rät sie nichtdeterministisch eine Knotenmenge $C \subseteq V$.
2. In einer zweiten Phase überprüft sie, ob C eine Clique in G ist und ob $|C| \geq k$. Ist dies der Fall, so akzeptiert sie; sonst verwirft sie.

Die Maschine ist offenbar korrekt und arbeitet in polynomieller Zeit. □

Genauso zeigt man die Zugehörigkeit der anderen Sprachen zu NP.

27.2 NP-Vollständigkeit

27.2.1 Erfüllbarkeitsprobleme

27-11

Klassische Erfüllbarkeitsprobleme: SAT . . .

▶ Definition: Die Sprache SAT -- die ganz formale Definition
Die Sprache $\text{SAT} \subseteq \Sigma^*_{\text{Aussagenlogik}}$ enthält alle erfüllbaren aussagenlogischen Formeln.

Beispiel

- $(p \vee q) \in \text{SAT}$,
- $(p \wedge \neg p) \vee (q \wedge \neg q) \notin \text{SAT}$.

Bemerkungen:

- Wir wissen schon, dass SAT mit der Wahrheitstafelmethode entscheidbar ist in linearem Platz.
- Die Wahrheitstafel zu erstellen ist aber *sehr zeitintensiv*.

27-12

Klassische Erfüllbarkeitsprobleme: . . . and friends.

▶ Definition: Die Sprachen k-SAT
Sei $k \geq 1$ fest. Die Sprache k-SAT ist eine Teilmenge von SAT. Sie enthält alle Formeln φ, die

1. erfüllbar sind und
2. in konjunktiver Normalform sind und
3. in der jede Klausel genau k Literale enthält.

Bemerkungen:

- Intuitiv solle k-SAT um so einfacher sein, je kleiner k ist – es gibt dann weniger Möglichkeiten, wie die Klauseln belegt werden können.
- Tatsächlich ist 1-SAT wirklich *ganz einfach* (1-SAT $\in$ L), da alle erlaubten Eingaben einfach große Und-Blöcke sind.
- Die Sprache 2-SAT ist *ebenfalls einfach,* sie ist nämlich *vollständig für* NL und somit *insbesondere in* P.
- Wir zeigen aber gleich, dass *ab* 3-SAT alle Probleme gleich schwierig sind.

27-13

Das Erfüllbarkeitsproblem für Schaltkreise.

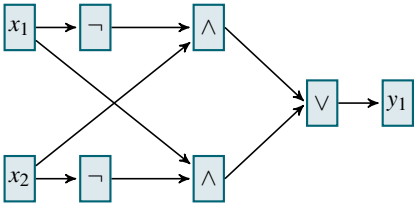
Analog zu *Formeln* kann man auch ein Erfüllbarkeitsproblem für *Schaltkreise* definieren:

▶ Definition: CIRCUIT-SAT
Die Sprache CIRCUIT-SAT enthält alle (Codes von) Schaltkreisen C, so dass

1. C hat genau ein Ausgabegatter und
2. *es gibt* eine Belegung für die Eingabegatter,
3. so dass C zu 1 auswertet.

Beispiel

Der Code des folgenden Schaltkreises ist ein Element von CIRCUIT-SAT:



27.2.2 Bootstrapping für ein erstes Problem

NP hat ein vollständiges Problem. 27-14

▶ Satz
CIRCUIT-SAT *ist vollständig für* NP.

Beweisplan

- Zunächst zeigen wir CIRCUIT-SAT $\in$ NP, was ganz einfach ist.
- Da wir noch keine vollständigen Probleme für NP kennen, müssen wir *Bootstrapping* anwenden.
- Der Beweis wird *sehr ähnlich* zu dem Beweis sein, dass CVP vollständig ist für P.

CIRCUIT-SAT ist in NP. 27-15

▶ Lemma: Erster Beweisteil
CIRCUIT-SAT $\in$ NP.

✎ Zur Übung
Skizzieren Sie einen Algorithmus.
Tipp: Dies geht ganz analog zum Algorithmus für SAT von Lemma 27-7.

Reduktion beliebiger Probleme aus NP. 27-16

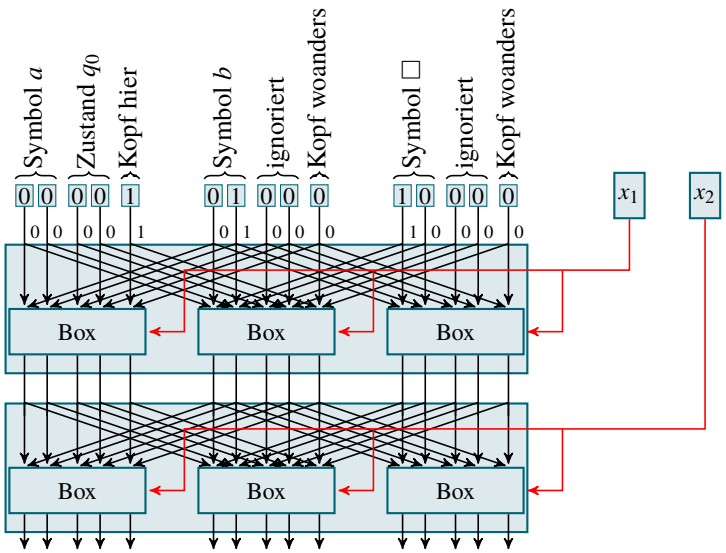
▶ Lemma: Zweiter Beweisteil
Sei $A \in$ NP *ein beliebiges Problem. Dann gilt* $A \leq_{\mathrm{m}}^{\log}$ CIRCUIT-SAT.

Beweis. Die Grundideen:

- Wir wollen ganz ähnlich wie beim Beweis verfahren, dass CVP vollständig ist für P. Wir benutzen auch dieselben Notationen. (Also M für die Maschine, die A akzeptiert; p als Laufzeitschranke für M; x für die Eingabe; n für die Länge von x.)
- *Neu* ist natürlich, dass wir die *nichtdeterministischen Entscheidungen* der Maschine berücksichtigen müssen.

Zur Umsetzung der Grundideen:

- Wie beim Beweis, dass CVP vollständig ist für P, dürfen wir annehmen, dass *»M viele schöne Eigenschaften hat«* wie nur ein Band und nur eine akzeptierende Endkonfiguration.
- *Zusätzlich* dürfen wir auch annehmen, dass es für *es nur »binäre« nichtdeterministische Entscheidungen* gibt: Pro Zustand darf es immer höchstens zwei Möglichkeiten geben, wie es weiter geht.
  (Anderenfalls kann man dies durch Einfügen von zusätzlichen Zuständen erzwingen.)
- Für eine Eingabe x baut man nun *denselben Schaltkreis wie im Beweis für* CVP, jedoch fügt man für jede Schicht noch ein *Eingabegatter* hinzu, dessen Belegung *angibt, wie die nichtdeterministische Entscheidung ausfällt.*



Zur Korrektheit der Konstruktion:

- Falls M das Wort x akzeptiert, so gibt es eine akzeptierende Berechnung $C_{\text{init}}(x) \vdash \cdots \vdash C_m$. In jedem Schritt dieser Berechnung ist »eine nichtdeterministische Entscheidung« gefallen. Dann *gibt es* eine Belegung der Eingabegatter, so dass der Schaltkreis zu 1 auswertet: Diese Belegung ist gerade durch diese Folge an »Entscheidungen« gegeben.
- Falls umgekehrt C zu 1 auswertet, so gibt die Belegung der Eingabegatter gerade an, wie die Berechnung der Maschine M ausfallen muss, damit x akzeptiert wird.

Dass die Reduktion in logarithmischen Platz möglich ist, zeigt man genauso wie im Beweis, dass CVP vollständig ist für P. □

27.2.3 Der Satz von Cook

27-17

Vom Schaltkreis zur Formel.

▶ Satz

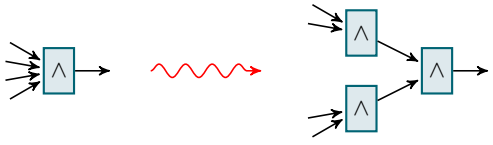
3-SAT *ist vollständig für* NP.

Beweis. Wir benutzen die *Reduktionsmethode.*

- Dazu müssen wir zunächst zeigen, dass 3-SAT ∈ NP gilt, was aber genauso wie bei SAT funktioniert (Lemma 27-7).
- Nun müssen wir noch zeigen, dass CIRCUIT-SAT (die »alte« Sprache) sich auf 3-SAT (die »neue« Sprache) reduzieren lässt.

Was *nicht* funktioniert, ist, den Schaltkreis rekursiv in eine Formel »aufzudröseln«, da diese Formel exponentiell groß werden könnte.
Stattdessen müssen wir einen *kleinen Trick* anwenden.
Für die Reduktion sei C ein Schaltkreis, der als Eingabe gegeben ist. Unser Ziel ist es, die Frage »Ist $C \in$ CIRCUIT-SAT?« umzuwandeln in die Frage »Ist $\varphi \in$ 3-SAT?«.
Dies geschieht in zwei Schritten:

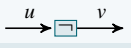
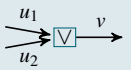
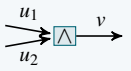
1. Wir wandeln zunächst C so um, dass *alle Gatter einen maximalen Fan-in von zwei haben.* Dazu werden Und- und Oder-Gatter von zu hohem Eingangsgrad durch mehrere kleine Gatter vom richtigen Grad ersetzt:



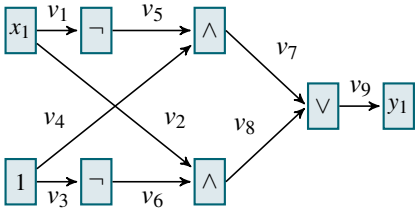
2. Die Idee hinter der *gesuchten Formel* φ lautet wie folgt:
   - Für jedes Eingabegatter gibt es eine Variable x_i.
   - Für jeden *Draht* des Schaltkreises führen wir eine Draht-Variable v_i ein.

- Unser Ziel ist, dass *erfüllende Belegungen* der Formel genau *erfüllenden Auswertungen* des Schaltkreises entsprechen.
- Dazu enthält unsere Formel Klauseln, die jeweils angeben, dass beispielsweise am Ausgabe-Draht eines Und-Gatters genau dann Strom anliegen darf, wenn beide Eingabe-Drähte Strom hatten.

Im Einzelnen funktionieren die Umwandlungen wie folgt:

Gatter	»Klauseln« in der Formel
	$v \leftrightarrow x_i$
	$\neg v$
	v
	v (es soll ja 1 herauskommen)
	$v \leftrightarrow \neg u$
	$v \leftrightarrow (u_1 \vee u_2)$
	$v \leftrightarrow (u_1 \wedge u_2)$

Ein Beispiel hierzu:
Aus



wird die Formel

$$\begin{aligned}
&(v_1 \leftrightarrow x_1)\\
\wedge\,&(v_2 \leftrightarrow x_1)\\
\wedge\,&v_3\\
\wedge\,&v_4\\
\wedge\,&(v_5 \leftrightarrow \neg v_1)\\
\wedge\,&(v_6 \leftrightarrow \neg v_3)\\
\wedge\,&(v_7 \leftrightarrow (v_5 \wedge v_4))\\
\wedge\,&(v_8 \leftrightarrow (v_2 \wedge v_6))\\
\wedge\,&(v_9 \leftrightarrow (v_7 \vee v_8))\\
\wedge\,&v_9
\end{aligned}$$

Die »Klauseln« in der Formel enthalten nun schon die richtige Anzahl an Variablen, sind aber noch keine 3-SAT-Klauseln.
Glücklicherweise können wir sie mit den Methoden aus »Logik für Informatiker« leicht in äquivalente Formeln in konjunktiver Normalform umwandeln.
Beispielsweise gilt

$$\begin{aligned}
v \leftrightarrow (u_1 \wedge u_2) \equiv\ &(v \vee \neg u_1 \vee \neg u_2) \wedge (v \vee \neg u_2 \vee u_2) \wedge\\
&(v \vee u_1 \vee \neg u_2) \wedge (\neg v \vee u_1 \vee u_2).
\end{aligned}$$

So viel zur Konstruktion der Reduktion. Die Korrektheit sieht man wie folgt:

1. Gibt es einen Belegung der Eingabegatter von C, so dass C zu 1 auswertet, so gibt es auch eine Belegung der Variablen von φ, so dass φ wahr wird: Man belegt alle Variablen gerade mit den Werten, die die Drähte bei der Auswertung des Schaltkreises erhalten.
2. Gibt es eine erfüllende Belegung von φ, so liefert diese gerade eine Belegung aller Drähte von C, die einer korrekten Auswertung entsprechen.

Dass die Reduktion in logarithmischem Platz berechenbar ist, liegt daran, dass beide Schritte einzeln recht logspace-berechenbar sind und die Reduktion transitiv ist. □

27-18

Der Satz von Cook.

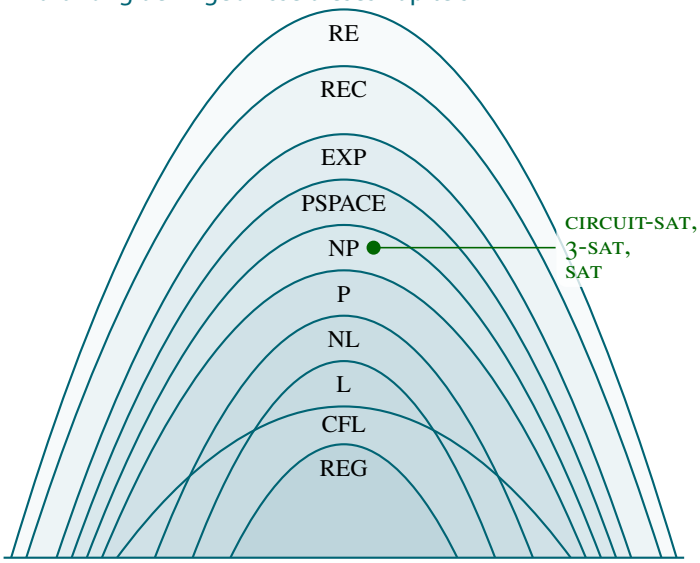
▶ **Folgerung**
SAT *ist vollständig für* NP.

Beweis. Wir benutzen die Reduktionsmethode. Wir haben schon gezeigt, dass SAT $\in$ NP gilt (Lemma 27-7) und dass 3-SAT $\leq_m^{\log}$ SAT (Beispiel 24-14). □

27.2.4 Übersicht der Klassenhierarchie

27-19

Einordnung der Ergebnisse dieses Kapitels



Zusammenfassung dieses Kapitels

27-20

1. Die Klasse NP enthält alle Sprachen, die nichtdeterministisch in polynomieller Zeit akzeptiert werden können.
2. Viele wichtige Probleme wie CLIQUE, SAT, GRAPH-ISOMORPHISM oder TSP liegen in diese Klasse.
3. Der Satz von Cook besagt, dass SAT vollständig ist für NP.
4. Der Beweis zeigt, dass sogar 3-SAT vollständig ist für NP.

Zum Weiterlesen

[1] R. Reischuk. *Unterlagen zum Lehrmodul Theoretische Informatik,* Vorlesungsskript, 2008. Kapitel 9.2.

28-1

Kapitel 28
Das P-NP-Problem

Warum ein schneller Algorithmus für SAT eine Million wert ist

Lernziele dieses Kapitels

1. Konzept des Klassenkollaps verstehen
2. Mehrere NP-vollständige Probleme kennen
3. Die Bedeutung der P-NP-Frage einschätzen können
4. Gründe kennen, weshalb die Frage so schwierig zu lösen ist

28-2

Inhalte dieses Kapitels

Worum es heute geht

Find the Longest Path
by Dan Barrett
(Original *The Longest Time* by Billy Joel)

Oh, oh, oh, oh,
find the longest path.
Oh, oh, oh,
find the longest path.

If you said P is NP tonight,
there would still be papers left to write.
I have a weakness:
I'm addicted to completeness
and I keep searching for the longest path.

The algorithm I would like to see
is of polynomial degree.
But it's elusive,
nobody has found conclusive
evidence that we can find the longest path.

I have been hard
working for so long.
I swear it's right,
and he marks it wrong.
Somehow I'll feel sorry when it's done:
GPA 2.1
is more than I hope for.

Garey, Johnson, Karp and other men
(and women)
try to make it order $n \log n$.
Am I a math fool
if I spend my life in grad school
forever following the longest path?

Oh, oh, oh, oh,
find the longest path.
Oh, oh, oh,
find the longest path.
Oh, oh, oh, oh,
find the longest path…

ANHÖREN | STOPP

28.1 Die Welt der NP-vollständigen Probleme

28.1.1 Eine andere Sicht auf NP

28-4

Zur allgemeinen Struktur von Problemen.

Viele Entscheidungsprobleme sind wie folgt aufgebaut:

- Die Eingabe ist eine Instanz.
- Für diese Eingabe suchen wir eine »Lösung«.
- Wenn eine solche Lösung existiert, so ist die Eingabe ein Element der Sprache; sonst halt nicht.

Beispiele

- Für PKP (Postsches Korrespondenzproblem) sind Lösungen *Korrespondenzen.*
- Für SAT sind Lösungen *erfüllende Belegungen.*
- Für CLIQUE sind Lösungen *Cliquen hinreichender Größe.*
- Für REACH sind Lösungen *Wege von s nach t.*

28-5

Eine formale Definition der Idee der »Lösung«.

▶ **Definition:** Sprachen mit leicht überprüfbaren Lösungen

Eine Sprache A *hat leicht überprüfbare Lösungen,* wenn

1. es eine *Lösungsrelation* $S \subseteq \Sigma^* \times \Sigma^*$ gibt, so dass
   - für alle $x \in A$ eine *Lösung* $y \in \Sigma^*$ existiert mit $(x,y) \in S$ und
   - für alle $x \notin A$ und für alle $y \in \Sigma^*$ gilt $(x,y) \notin S$;
2. und die Sprache $\{u\#v \mid (u,v) \in S\}$ in polynomieller Zeit entscheidbar ist.

Was die Definition eigentlich aussagt:

- Für Worte in der Sprache muss es immer eine Lösung geben.
- Für Worte außerhalb der Sprache darf es keine Lösungen geben.
- Es muss leicht entscheidbar sein, ob eine Lösung zu einem Wort passt.

28-6

Wie schwierig ist es, Lösungen zu finden?

Es gibt *unterschiedliche Gründe*, weshalb *Lösungen schwer zu finden sind.*

- Lösungen können *unglaublich groß sein.*
  Beispiel: PKP
- Lösungen können *extrem schwer zu finden sein.*
  Beispiel: SAT, CLIQUE
- Lösungen können *jedenfalls nicht einfach zu finden sein.*
  Beispiel: GRAPH-ISOMORPHISM

28-7

Eine alternative Sicht auf NP.

▶ **Satz**

Eine Sprache A ist genau dann in NP*, wenn A leicht überprüfbare Lösungen hat, deren Länge höchstens polynomiell in der Länge des Wortes ist.*

Kommentare zum Beweis

Beweis. Wir zeigen zwei Richtungen.[1]

Sei $A \in$ NP via M. Die *Lösungen* für ein Wort x sind *einfach (die Codes) von allen Berechnungen, auf denen M das Wort x akzeptiert.* (Ist $x \notin A$, so gibt es entsprechend auch keine Lösungen für w.) Da M polynomiell zeitbeschränkt ist, sind die Lösungen auch nur polynomiell lang. Weiter ist es leicht zu überprüfen, ob eine Berechnung »in sich korrekt« ist.

Für die andere Richtung habe A leicht überprüfen Lösungen polynomieller Länge. Eine NP-Maschine M für A arbeitet in zwei Phasen:[2]

1. Rate eine Lösung auf einem Arbeitsband.
2. Überprüfe die Lösung und akzeptiere, wenn sie stimmt.

Nun gilt tatsächlich, dass M die Sprache akzeptiert: Ist $x \in A$, so gibt es ja eine Lösung und eine geeignete Berechnung untersucht diese auch, befindet sie für gut und akzeptiert. Ist $x \notin A$, so gibt es keine Lösung und keine Berechnung akzeptiert.

Alle Berechnung dauern nur polynomiell lang, da ja die erste Phase wie immer schnell geht und die Überprüfung einfach ist.[3] □

[1] Rezept »Zwei Richtungen«

[2] Die »zwei Phasen« kennen wir schon.

[3] »einfach« = »in polynomieller Zeit«

Beweisrezept: *Zugehörigkeit zu NP*

28-8

Ziel

Beweisen, dass $A \in$ NP gilt.

Rezept

1. Man überlegt sich, wie eine »Lösung« zu einer Eingabe aussieht.
2. Beginne mit: »Die Lösungen zu einer Eingabe x lauten…«
3. Erkläre, weshalb es genau dann eine Lösung zu x gibt, wenn $x \in A$ gilt.
4. Erkläre, weshalb alle Lösungen zu x höchstens Länge $|x|^{O(1)}$ haben.
5. Erkläre, wie man in polynomieller Zeit überprüfen kann, ob ein Wort eine Lösung zu einem x ist.

Einzelne oder sogar alle Erklärungen können weglassen werden, wenn sie »offensichtlich« sind.

Anwendungen des Beweisrezepts

28-9

▶ **Lemma**

SAT $\in$ NP.

Beweis. Lösungen für eine Eingabe φ sind die erfüllenden Belegungen von φ. □

▶ **Lemma**

TSP $\in$ NP.

Beweis. Lösungen für eine Eingabe sind Rundreisen, die das Budget einhalten. □

▶ **Lemma**

REACH $\in$ NP.

Beweis. Lösungen für eine Eingabe sind Wege von s nach t. □

Warum so viele Probleme in NP sind.

28-10

Zentrale Eigenschaft von NP

NP enthält alle Sprachen, für die

- es vielleicht extrem schwer ist, kurze (= polynomiell lange) Lösungen zu finden…
- …es aber ganz einfach ist, Lösungen zu überprüfen.

Diese Eigenschaft sorgt dafür, dass so viele *praktisch bedeutsame Sprachen* in NP sind:

- Bei realen Problemen ist *sind Lösungen in der Regel schwer zu finden…*
- …aber man erkennt es leicht, wenn man irgendwie eine gefunden hat.

28.1.2 Wichtige NP-vollständige Probleme

Was ist alles NP-vollständig außer Erfüllbarkeitsproblemen?

28-11

- Wir haben bereits gezeigt, dass die »maschinennahe« Sprache CIRCUIT-SAT *vollständig für* NP ist.
- Es gibt aber auch jede Menge NP-vollständige Probleme, *»denen man es zunächst nicht ansieht.«*

Beispiel: Beispiele von NP-vollständigen Problemen

- CLIQUE und INDEPENDENT-SET
- VERTEX-COVER $= \{\text{code}(G,k) \mid$ man kann k Knoten von G auswählen, so dass jede Kante von G wenigstens einen dieser Knoten berührt$\}$
- TRAVELLING-SALESPERSON
- HAMILTON $= \{\text{code}(G) \mid$ es gibt einen hamiltonischen Pfad in G, also eine Rundreise, die jeden Knoten genau einmal durchläuft$\}$
- SUBSET-SUM
- LONGEST-PATH $= \{\text{code}(G,k) \mid$ es gibt einen Pfad in G, der keinen Knoten zweimal durchläuft und der Länge mindestens k hat$\}$

28-12

Wie man die NP-Vollständigkeit der Probleme zeigt.

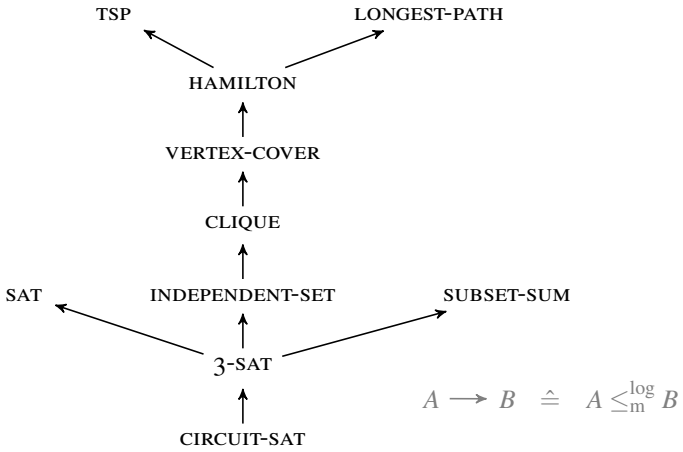
- Man benutzt grundsätzlich die *Reduktionsmethode,* um die NP-Vollständigkeit von (neuen) Problemen zu beweisen.
- Auf diese Art hat man schon für viele hundert Probleme *aus ganz unterschiedlichen Gebieten* gezeigt, dass sie NP-vollständig sind.
- Für *neue, sehr spezielle Probleme aus der Anwendung* ist es deshalb oft leicht, die NP-Vollständigkeit zu beweisen.

Beispiel: Zitat aus einem typischen Paper
Im Paper *Influence of Tree Topology Restrictions on the Complexity of Haplotyping with Missing Data* von Elberfeld, Schnoor und Tantau heißt es:

> **Theorem 1.1** $\text{IDPP}_{\text{leafs}\leq\ell}$ *is* NP*-complete for every* $\ell \geq 2$.
>
> *Proof.* We only show hardness. Fix an $\ell \geq 2$. We present a reduction from the problem MONOTONE NAE3SAT to $\text{IDPP}_{\text{leafs}\leq\ell}\cdots$

28-13

Wie man die ersten Reduktionen durchführt.



$A \longrightarrow B \;\hat{=}\; A \leq_{\text{m}}^{\log} B$

Ein Beispiel einer Reduktion.

28-14

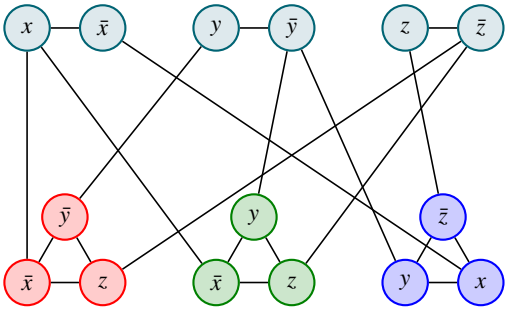
▶ Satz

INDEPENDENT-SET *ist* NP-*vollständig.*

Beweisskizze. [1]Dass INDEPENDENT-SET $\in$ NP liegt daran, dass Lösungen zu einer Eingabe (G,k) gerade die unabhängigen Mengen in G der Größe k sind.[2]
Für die Schwere von INDEPENDENT-SET für NP reduzieren wir von 3-SAT.[3]
Die Idee hinter der Reduktion: Sei φ eine Formel mit m Klauseln und n Variablen. Die Reduktion verwandelt die Frage »Ist $\varphi \in$ 3-SAT?« in die Frage »Ist code$(G, m+n) \in$ INDEPENDENT-SET?« für folgenden Graphen G:

- Für jedes mögliche Literal (also jede Variable und ihre Negation) gibt es ein Knoten, *Literalknoten* genannt.
- Es gibt eine Kante zwischen dem Knoten einer Variable und dem Knoten ihrer Negation.
- Für jede Klausel gibt es
  - ein Dreieck in dem Graphen und
  - je eine Kante für jedes Literal in der Klausel, nämlich von einer Ecke zum Knoten des *negierten* Literals.

Ein Beispiel des Graphen G für $\varphi = (\bar{x} \vee \bar{y} \vee z) \wedge (\bar{x} \vee y \vee z) \wedge (x \vee y \vee \bar{z})$:



Aus folgenden Beobachtungen schließt man nun die Korrektheit der Reduktion:[4]

- In jedem Dreieck kann eine unabhängige Menge höchstens einen Knoten enthalten.
- Von jedem Paar an Literalknoten kann eine unabhängige Menge höchstens einen Knoten enthalten.
- Also hat eine unabhängige Menge höchstens $n+m$ Knoten.

Nun gilt

- Ist die Formel erfüllbar, so liefert eine erfüllende Belegung eine unabhängige Menge: Man wählt gerade die Literalknoten, die wahr sind, und aus jedem Dreieck auch einen Knoten, der mit einem wahren Literal beschriftet ist.
- Umgekehrt liefert eine unabhängige Menge durch die Menge der ausgewählten Literalknoten eine erfüllende Belegung. □

Kommentare zum Beweis

[1] Hier fehlt der Hinweis »Wir benutzen die Reduktionsmethode«, da das sowieso immer so geht.

[2] Rezept »Zugehörigkeit zu NP«

[3] Hier fehlt der Hinweis, dass 3-SAT schon NP-vollständig bekannt ist. Das »weiß man aber«.

[4] Das ist so knapp, dass statt »Beweis« nur »Beweisskizze« am Anfang steht.

28.2 Das P-NP-Problem

28.2.1 Das Problem

Eines der sieben Probleme, auf deren Lösungen je eine Million Dollar ausgeschrieben sind.

28-15

Das P-NP-Problem

Gilt P = NP oder P $\subsetneq$ NP?

Eine äquivalente Formulierung des Problems

Gilt SAT $\in$ P oder SAT $\notin$ P?

28.2.2 Warum man glaubt, dass P ungleich NP ist

28-16

Was passieren würde, wenn P gleich NP wäre.

- Nach dem Repräsentationslemma (Lemma 24-22) gilt: *Ist ein* NP*-vollständiges Problem in* P*, so gilt* P = NP.
- Würde es also für *eines der vielen hundert* NP*-vollständigen Probleme* einen schnellen (=Polynomialzeit) Algorithmus geben, so würde es ihn *auch für alle anderen Probleme geben.*
- Trotz intensiver Suche *sowohl von Theoretikern wie von Praktikern* hat man noch keinen solchen Algorithmus gefunden.
- Das deckt sich gut mit der Vermutung, dass P ≠ NP *gilt* – dann *gibt es nämlich solche Algorithmen auch nicht.*
  Man kann dann lange nach ihnen suchen.

28.2.3 Warum das Problem schwer ist

28-17

Wieso ist das Problem so schwer?

- Nehmen wir an, es gilt wirklich P ≠ NP.
- Wiese hat das dann noch niemand bewiesen?

Da kann es viele Gründe geben:

1. Es ist eigentlich ganz einfach mit Standardmethoden zu zeigen, alle Theoretiker haben aber Tomaten auf den Augen.
2. Es ist mit Standardmethoden zu beweisen, aber der Beweis ist tausende Seiten lang – weshalb ihn noch niemand gefunden hat.
3. Es ist nur mit neuen, heute noch unbekannten Methoden zu beweisen.
4. Es ist gar nicht beweisbar (das wäre tatsächlich denkbar, nicht jeder mathematische Satz ist beweisbar).

Wir können heute schon die ersten beiden Möglichkeiten ausschließen. Die vierte erscheint *sehr* unwahrscheinlich.

28-18

Warum man mit Standardmethoden nicht weiterkommt.

- Der Befehlsatz unserer Turing-Maschinen und unser RAM-Maschinen war *eigentlich ziemlich egal*.
- Insbesondere ändern leichte Abwandlungen des Befehlssatzes nicht, was die Maschinen können.
- Was passiert aber, wenn man den Befehlssatz einer Turing-Maschine *radikal verändert?*

Die Idee der Maschinen mit Super-Befehlen

- Wir betrachten nun Maschinen, in deren Befehlssatz bestimmte »Super-Befehle« zugelassen sind.
- Beispielsweise könnte es neben den üblichen Befehlen wie »Bewege den Kopf nach links, wenn da ein Blank steht« auch Befehle geben wie
  - »Bewege den Kopf nach links, wenn auf dem dritten Band eine Primzahl steht« oder
  - »Wechsel in Zustand q_5, wenn der auf dem zweiten Band kodierte Graph einen Hamiltonschen Pfad besitzt.«

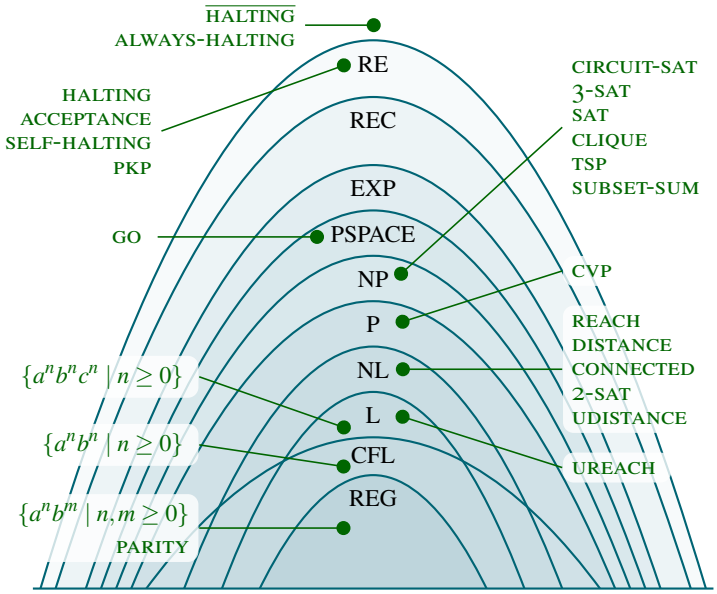
Man kann nun die Klassen P und NP neu definieren *relativ zu einem Super-Befehlssatz*. Man erhält *relativierte Klassen.*

Drei wichtige Beobachtungen

1. Es stellt sich heraus, dass *praktisch alle Resultate und praktisch alle Beweise auch dann funktionieren,* wenn man die ganze Theorie statt mit normalen Turing-Maschinen neu aufschreibt für Turing-Maschinen mit *einem festen Super-Befehlssatz.*
2. Es gibt einen Super-Befehlsatz, relativ zu dem P = NP gilt.
3. Es gibt einen Super-Befehlsatz, relativ zu dem P ≠ NP gilt.

Dies bedeutet nun Folgendes: Um P ≠ NP zu beweisen, braucht man Beweisverfahren, die *»nicht relativierbar«* sind – die also nicht für jeden beliebigen Super-Befehlsatz funktionieren.
Man kennt nur sehr wenige solche Verfahren.

28.3 Übersicht der Klassenhierarchie



28-19

Zusammenfassung dieses Kapitels

1. Bei Sprachen in NP gibt es zu jedem Wort in der Sprache eine kurz, effizient überprüfbare Lösung.
2. Das *Finden* der Lösung kann hingegen schwer sein.
3. Das P-NP-Problem ist die Frage, ob P = NP gilt.
4. Dieses Problem ist ungelöst und kann mit Standardverfahren nicht beantwortet werden.

28-20

Zum Weiterlesen

[1] R. Reischuk. *Unterlagen zum Lehrmodul Theoretische Informatik,* Vorlesungsskript, 2008. Kapitel 9, 9.5.

Anhang

Lösungen

Beispiellösungen zu ausgesuchten Übungen

Lösung zu 3.4

Behauptung
Die Klasse CFL *ist unter Verkettung abgeschlossen.*

Beweis. Seien L_1 und L_2 zwei kontextfreie Sprachen. Zu zeigen ist, dass die Verkettung $L_1 \circ L_2$ ebenfalls eine kontextfreie Sprache ist. Da L_1 und L_2 kontextfreie Sprachen sind, existieren kontextfreie Grammatiken $G_1 = (T_1, N_1, S_1, P_1)$ und $G_2 = (T_2, N_2, S_2, P_2)$ mit $L_1 = L(G_1)$ und $L_2 = L(G_2)$. Wir konstruieren eine kontextfreie Grammatik G_3, für die gilt $L(G_3) = L_1 \circ L_2$. Im ersten Schritt benennen wir die Nonterminalsymbole in G_1 und G_2 geeignet um, so dass N_1, N_2 und $T_1 \cup T_2$ paarweise disjunkt sind. Darufhin definieren wir die Grammatik $G_3 = (T_3, N_3, S_3, P_3)$ mit $N_3 = N_1 \cup N_2 \cup \{S_3\}$, $T_3 = T_1 \cup T_2$, $P_3 = P_1 \cup P_2 \cup \{S_3 \to S_1 S_2\}$. Die Grammatik G_3 ist kontextfrei, weil die einzige hinzugekommene Produktionsregel $S_3 \to S_1 S_2$ bei kontextfreien Grammatiken zulässig ist. Es bleibt noch zu zeigen, dass $L(G_3) = L_1 \circ L_2$.

Für die Richtung $L_1 \circ L_2 \subseteq L(G_3)$ sei $w \in L_1 \circ L_2$. Dies bedeutet, dass w die Form $w = uv$ hat mit $u \in L_1$ und $v \in L_2$. Wir geben eine Ableitungskette für w in G_3 an. Der erste Schritt der Ableitung ist $S_3 \Rightarrow_{G_3} S_1 S_2$. Da $u \in L_1$, gilt $S_1 \Rightarrow^*_{G_1} u$. Da die Regeln von G_1 in G_3 übernommen wurden, gilt auch $S_1 S_2 \Rightarrow^*_{G_3} uS_2$. Analog gilt dann auch $S_2 \Rightarrow^*_{G_2} v$ und somit $uS_2 \Rightarrow^*_{G_3} uv$. Insgesamt erhalten wir $S_3 \Rightarrow_{G_3} S_1 S_2 \Rightarrow^*_{G_3} uS_2 \Rightarrow^*_{G_3} uv = w$ Damit gilt $w \in L(G_3)$.

Für die Inklusion $L(G_3) \subseteq L_1 \circ L_2$ sei $w \in L(G_3)$ beliebig. Dies bedeutet, dass in G_3 eine Ableitungskette $S_3 \Rightarrow^* w$ existiert. Der erste Schritt der Ableitungskette muss dabei $S_3 \Rightarrow S_1 S_2$ gewesen sein. Das Wort $S_1 S_2$ besteht aus zwei Nonterminalen S_1 und S_2. Nun können wir argumentieren, dass, aufgrund der erfolgten Umbenennung der Nonterminale, wir ausgehend von S_1 wir nur Ersetzungsregeln aus der Grammatik G_1 anwenden können und ausgehend von S_2 nur Ersetzungregeln aus der Grammatik G_2 anwenden können. Damit kann das Teilwort u, das sich in der Ableitungskette aus S_1 erzeugen lässt, nur aus der Sprache L_1 kommen und das Teilwort v, das sich in der Ableitungskette aus S_2 ableiten lässt, nur aus der Sprahe L_2 kommen. Dies bedeutet aber, dass das gesamte abgeleitete Wort $w = uv$ ein Element der Sprache $L_1 \circ L_2$ ist. Damit folgt $L(G_3) \subseteq L_1 \circ L_2$. □

Lösung zu 5.1

Behauptung
Zeigen Sie mit Hilfe der Folgerung aus dem Pumping-Lemma, dass folgende Sprache nicht regulär ist:

$$L = \{a^n \mid n \textit{ ist eine Primzahl}\}$$

Beweis. Wir zeigen die Behauptung durch die Kontraposition des Pumping-Lemmas. Sei n beliebig. Dann existiert eine Primzahl $m \geq n+2$, so dass $w = a^m$ ein Element von $L \cap \Sigma^{\geq n}$ ist. Sei nun $w = xyz$ eine beliebige Zerlegung mit $|y| \geq 1$ und $|xy| \leq n$. Für das Teilwort $w' = xz$ folgt, dass $|w'| = |w| - |y| = t$ mit $m - n \leq t \leq m - 1$. Dann folgt aber für das aufgepumpte Wort $w^t = x \circ y^t \circ z = a^{(|y|+1)\cdot t}$. Nun folgt aber, dass $(|y|+1) \cdot t$ keine Primzahl ist, da sie den ganzzahligen Teiler $t \geq 2$ besitzt und damit folgt auch $w^t \notin L$. □

Lösung zu 18.1

Behauptung

SELF-HALTING $\notin$ REC.

Kommentare zum Beweis

Beweis. Wir zeigen, dass aus der Entscheidbarkeit von SELF-HALTING die Entscheidbarkeit von HALTING folgen würde – aber wir wissen schon, dass HALTING unentscheidbar ist. Nehmen wir also an, SELF-HALTING wäre entscheidbar via eines Entscheiders $M_{\text{SELF-HALTING}}$. Dann lässt sich HALTING via folgendem Entscheider M_{HALTING} entscheiden:[1]
Bei Eingabe $w = \text{code}(N)\#x$ konstruiert die Maschine M_{HALTING} zunächst ein neues Wort w' wie folgt:[2]
$w' = \text{code}(N_x)$, wobei N_x folgende Maschine ist:

- In einer Schleife löscht sie das Eingabeband (überschreibt alle dort befindlichen Zeichen mit Blanks).
- Dann schreibt sie das Wort x auf das Eingabeband. (Dazu hat sie für jedes Zeichen von x einen Zustand fest einkodiert.)
- Dann führt sie N aus.

[1] Bis hier Originaltext aus dem Beweisrezept »Unentscheidbarkeit beweisen«

[2] Wie immer kommt hier der entscheidende Trick

Das Wort w' geht durch eine syntaktische Transformation S aus w hervor:[3]

[3] Das Bild kann man auch weglassen

Es bleibt zu zeigen, dass M_{HALTING} tatsächlich HALTING entscheidet. Dies sieht man so:

$$\begin{aligned}
&\text{code}(N)\#x \in \text{HALTING} \\
&\quad\iff N \text{ hält auf Eingabe } x \text{ an} \\
&\quad\iff N_x \text{ hält irgendeiner Eingabe an} \\
&\quad\iff N_x \text{ hält auf Eingabe code}(N_x) \text{ an} \\
&\iff \text{code}(N_x) \in \text{SELF-HALTING} \\
&\quad\iff w' \in \text{HALTING}.
\end{aligned}$$

Also ist M_{HALTING} total und akzeptiert HALTING. □

Lösung zu 19.2

Wir beginnen mit einem Programm g, auf das wir dann den Beweis des Rekursionssatzes anwenden können. Laut Aufgabenstellung soll g

- den übergebenen Programmtext ausdruckt, dabei aber nach jedem Zeichen ein Leerzeichen einfügt und
- dieses Ausdrucken auch noch so oft wiederholt, wie der erste Parameter angibt.

Dies könnte wie folgt funktionieren:

```
public static void g(String v, int x) {
  for (int i = 0; i < x; i++)
    for (int j = 0; j < v.length(); j++)
      System.out.print(v.charAt(j)+"␣");
}
```

Wendet man an Rekursionssatz auf dieses Programm an, so kommt ein Programm heraus mit folgender Eigenschaft:

- Es bekommt einen Parameter x.
- Es gibt seinen eigenen Programmtext x Mal aus.
- Nach jedem Zeichen des Programmtextes soll noch ein Leerzeichen eingefügt sein.

Der erste Schritt im Beweis des Rekursionssatzes war die Konstruktion einer Funktion S. Diese ist in Java recht einfach zu implementieren:

```
static String S (String program) {
  // Compute escaped version of program:
  String program_escaped = program;
  program_escaped = program_escaped.replaceAll("\\\\", "\\\\\\\\");
  program_escaped = program_escaped.replaceAll("\"", "\\\\\"");
  program_escaped = program_escaped.replaceAll("\n", "\\\\n\"+\n\"");
  program_escaped = program_escaped.replaceAll("\\\\", "\\\\\\\\");
  // Get rid of Parameter:
```

```
  String result = program.replaceFirst("String␣v,␣", "");
  // Insert fixed setting at begin of main method:
  return result.replaceFirst("x\\)␣\\{",
                             "x)␣{\n␣␣String␣v␣=␣\"" + program_escaped +
                                 "\";");
}
```

Die komischen »Escapes« sind nötig, damit man in Java einem String einen String zuweisen kann, der Backslashes und Anführungszeichen enthält.

Hier ein Beispiel, wie *S* arbeitet. Wir rufen *S* wie folgt auf:

```
System.out.println(
  S("void␣g(String␣v,␣int␣x)␣{\n" +
    "␣␣System.out.print(\"foo\");\n" +
    "}␣"));
```

Dies liefert folgende Ausgabe:

```
void g(int x) {
  String v = "void␣g(String␣v,␣int␣x)␣{\n"+
"␣␣System.out.print(\"foo\");\n"+
"}␣";
  System.out.print("foo");
}
```

Nun kann man die Funktion *h* definieren:

```
public static void h(String v, int x) {
  g(S(v),x);
}
```

Diese Funktionen *h*, *g* und *S* bilden zusammen ein Programm P_h. Das gewünschte Programm P_f ergibt sich nun durch einmalige Anwendung von *S* hierauf. Dadurch verliert *h* einen Parameter und es resultiert folgender Programmtext:

```
public static void h(int x) {
  String v = "␣␣public␣static␣void␣h(String␣v,␣int␣x)␣{\n"+
"␣␣␣␣␣␣␣␣␣␣␣␣␣␣g(S(v),x);\n"+
"␣␣␣␣␣␣}\n"+
"\n"+
"␣␣␣␣␣␣public␣static␣void␣g(String␣v,␣int␣x)␣{\n"+
"␣␣␣␣␣␣␣␣␣␣␣␣␣␣for␣(int␣i␣=␣0;␣i␣<␣x;␣i++)␣\n"+
"␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣for␣(int␣j␣=␣0;␣j␣<␣v.length();␣j++)\n"+
"␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣System.out.print(v.charAt(j)+\"␣\");\n"+
"␣␣␣␣␣␣}\n"+
"\n"+
"␣␣␣␣␣␣static␣String␣S␣(String␣program)␣{\n"+
"␣␣␣␣␣␣␣␣␣␣␣␣␣␣//␣Compute␣escaped␣version␣of␣program:\n"+
"␣␣␣␣␣␣␣␣␣␣␣␣␣␣String␣program_escaped␣=␣program;\n"+
"␣␣␣␣␣␣␣␣␣␣␣␣␣␣program_escaped␣=␣program_escaped.replaceAll(\"\\\\\\\\\\\",␣\"\\\\\\\\\\\\\\\\\\\\\");\n"+
"␣␣␣␣␣␣␣␣␣␣␣␣␣␣program_escaped␣=␣program_escaped.replaceAll(\"\\\"\",␣\"\\\\\\\\\\\"\");\n"+
"␣␣␣␣␣␣␣␣␣␣␣␣␣␣program_escaped␣=␣program_escaped.replaceAll(\"\\n\",␣\"\\\\\\\\n\\\"+\n\\\"\");\n"+
"␣␣␣␣␣␣␣␣␣␣␣␣␣␣program_escaped␣=␣program_escaped.replaceAll(\"\\\\\\\\\",␣\"\\\\\\\\\\\\\\\\\");\n"+
"␣␣␣␣␣␣␣␣␣␣␣␣␣␣//␣Get␣rid␣of␣Parameter:\n"+
"␣␣␣␣␣␣␣␣␣␣␣␣␣␣String␣result␣=␣program.replaceFirst(\"String␣v,␣\",␣\"\");\n"+
"␣␣␣␣␣␣␣␣␣␣␣␣␣␣//␣Insert␣fixed␣setting␣at␣begin␣of␣main␣method:\n"+
"␣␣␣␣␣␣␣␣␣␣␣␣␣␣return␣result.replaceFirst(\"x\\\\)␣\\\\{\",␣\n"+
"␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣\"x)␣{\\n␣␣String␣v␣=␣\\\"\"␣+␣program_escaped␣+␣\"\\\";\");\n"+
"␣␣␣␣␣␣}\n"+
"";
  g(S(v),x);
}

public static void g(String v, int x) {
  for (int i = 0; i < x; i++)
    for (int j = 0; j < v.length(); j++)
      System.out.print(v.charAt(j)+"␣");
}

static String S (String program) {
  // Compute escaped version of program:
  String program_escaped = program;
  program_escaped = program_escaped.replaceAll("\\\\", "\\\\\\\\");
  program_escaped = program_escaped.replaceAll("\"", "\\\\\"");
  program_escaped = program_escaped.replaceAll("\n", "\\\\n\"+\n\"");
  program_escaped = program_escaped.replaceAll("\\\\", "\\\\\\\\");
  // Get rid of Parameter:
  String result = program.replaceFirst("String␣v,␣", "");
  // Insert fixed setting at begin of main method:
  return result.replaceFirst("x\\)␣\\{",
    "x)␣{\n␣␣String␣v␣=␣\"" + program_escaped + "\";");
}
```

Ruft man nun `h(2)` auf, so erhält man tatsächlich Folgendes (etwas gekürzt wiedergegeben):

```
p u b l i c  s t a t i c  v o i d  h ( i n t  x )  {
    S t r i n g  v  =  "␣␣␣␣p␣u␣b␣l␣i␣c␣␣␣s␣t␣a␣t␣i␣c␣␣␣v␣o␣i␣d␣␣␣h␣(␣S␣t␣r␣i␣n␣g␣␣␣v␣,␣␣␣i␣n␣t␣␣␣x␣)␣␣␣{␣\n␣" +
"␣␣␣␣␣␣␣␣g␣(␣S␣(␣v␣)␣,␣x␣)␣;␣\n␣" +
"␣␣␣␣}␣\n␣" +
"␣\n␣" +
"␣␣␣␣p␣u␣b␣l␣i␣c␣␣␣s␣t␣a␣t␣i␣c␣␣␣v␣o␣i␣d␣␣␣g␣(␣S␣t␣r␣i␣n␣g␣␣␣v␣,␣␣␣i␣n␣t␣␣␣x␣)␣␣␣{␣\n␣" +
"␣␣␣␣␣␣␣␣␣␣f␣o␣r␣␣␣(␣i␣n␣t␣␣␣i␣␣␣=␣␣␣0␣;␣␣␣i␣␣␣<␣␣␣x␣;␣␣␣i␣+␣+␣)␣␣␣\n␣" +
"␣␣␣␣␣␣␣␣␣␣␣␣␣␣f␣o␣r␣␣␣(␣i␣n␣t␣␣␣j␣␣␣=␣␣␣0␣;␣␣␣j␣␣␣<␣␣␣v␣.␣l␣e␣n␣g␣t␣h␣(␣)␣;␣␣␣j␣+␣+␣)␣\n␣" +
"␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣S␣y␣s␣t␣e␣m␣.␣o␣u␣t␣.␣p␣r␣i␣n␣t␣(␣v␣.␣c␣h␣a␣r␣A␣t␣(␣j␣)␣+␣\␣"   \ "␣)␣;␣\n␣" +
"␣␣␣␣}␣\n␣" +
```

```
"␣\n␣" +
"␣␣␣␣s␣t␣a␣t␣i␣c␣␣␣S␣t␣r␣i␣n␣g␣␣␣S␣␣␣(␣S␣t␣r␣i␣n␣g␣␣␣p␣r␣o␣g␣r␣a␣m␣)␣␣␣{␣\n␣" +
...
"␣␣␣␣␣}␣\n␣" +
"␣" ;
        g ( S ( v ) , x ) ;
    }

    p u b l i c  s t a t i c  v o i d  g ( S t r i n g  v ,  i n t  x )  {
        f o r  ( i n t  i  =  0 ;  i  <  x ;  i + + )
            f o r  ( i n t  j  =  0 ;  j  <  v . l e n g t h ( ) ;  j + + )
                S y s t e m . o u t . p r i n t ( v . c h a r A t ( j ) + " ␣␣␣" ) ;
    }

    s t a t i c  S t r i n g  S  ( S t r i n g  p r o g r a m )  {
        / /  C o m p u t e  e s c a p e d  v e r s i o n  o f  p r o g r a m :
        S t r i n g  p r o g r a m _ e s c a p e d  =  p r o g r a m ;
...
    }
    p u b l i c  s t a t i c  v o i d  h ( i n t  x )  {
    S t r i n g  v  =  "␣␣␣␣p␣u␣b␣l␣i␣c␣␣␣s␣t␣a␣t␣i␣c␣␣␣v␣o␣i␣d␣␣␣h␣(␣S␣t␣r␣i␣n␣g␣␣␣v␣,␣␣␣i␣n␣t␣␣␣x␣)␣␣␣{␣\n␣" +
"␣␣␣␣␣␣␣␣g␣(␣S␣(␣v␣)␣,␣x␣)␣;␣\n␣" +
"␣␣␣␣␣}␣\n␣" +
"␣\n␣" +
"␣␣␣␣p␣u␣b␣l␣i␣c␣␣␣s␣t␣a␣t␣i␣c␣␣␣v␣o␣i␣d␣␣␣g␣(␣S␣t␣r␣i␣n␣g␣␣␣v␣,␣␣␣i␣n␣t␣␣␣x␣)␣␣␣{␣\n␣" +
"␣␣␣␣␣␣␣␣␣f␣o␣r␣␣␣(␣i␣n␣t␣␣␣i␣␣␣=␣␣␣0␣;␣␣␣i␣␣␣<␣␣␣x␣;␣␣␣i␣+␣+␣)␣␣␣\n␣" +
"␣␣␣␣␣␣␣␣␣␣␣␣f␣o␣r␣␣␣(␣i␣n␣t␣␣␣j␣␣␣=␣␣␣0␣;␣␣␣j␣␣␣<␣␣␣v␣.␣l␣e␣n␣g␣t␣h␣(␣)␣;␣␣␣j␣+␣+␣)␣\n␣" +
"␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣S␣y␣s␣t␣e␣m␣.␣o␣u␣t␣.␣p␣r␣i␣n␣t␣(␣v␣.␣c␣h␣a␣r␣A␣t␣(␣j␣)␣+␣\"␣␣␣\"␣)␣;␣\n␣" +
"␣␣␣␣␣}␣\n␣" +
"␣\n␣" +
"␣␣␣␣s␣t␣a␣t␣i␣c␣␣␣S␣t␣r␣i␣n␣g␣␣␣S␣␣␣(␣S␣t␣r␣i␣n␣g␣␣␣p␣r␣o␣g␣r␣a␣m␣)␣␣␣{␣\n␣" +
...
"␣␣␣␣␣}␣\n␣" +
"␣" ;
        g ( S ( v ) , x ) ;
    }

    p u b l i c  s t a t i c  v o i d  g ( S t r i n g  v ,  i n t  x )  {
        f o r  ( i n t  i  =  0 ;  i  <  x ;  i + + )
            f o r  ( i n t  j  =  0 ;  j  <  v . l e n g t h ( ) ;  j + + )
                S y s t e m . o u t . p r i n t ( v . c h a r A t ( j ) + " ␣␣␣" ) ;
    }

    s t a t i c  S t r i n g  S  ( S t r i n g  p r o g r a m )  {
        / /  C o m p u t e  e s c a p e d  v e r s i o n  o f  p r o g r a m :
        S t r i n g  p r o g r a m _ e s c a p e d  =  p r o g r a m ;
...
    }
```

Anhang

Referenz: Beweisrezepte

Zusammenstellung aller Beweisrezepte

Beweisrezept: *Abgeschlossenheit von Grammatik-Sprachklassen* A-1

Ziel

Man will beweisen, dass eine Sprachklasse C unter einer Operation $\otimes$ abgeschlossen ist.

Rezept

1. Beginne mit »Seien $L_1, L_2 \in C$. Dann gibt es Grammatiken G_1 und G_2 für L_1 und L_2.«
2. Konstruiere dann eine Grammatik G für $L_1 \otimes L_2$ (Beweisrezept »Konstruktiver Beweis«).
3. Zeige, dass alle Regeln in G vom richtigen Typ sind.
4. Zeige, dass $L(G) = L_1 \otimes L_2$ gilt (Beweisrezept »Korrektheit von Grammatiken«).

Beweisrezept: *All-Aussagen beweisen* A-2

Ziel

Es soll gezeigt werden »für alle Dinge, die so und so sind, gilt blah« oder auch »jedes Ding, das so und so ist, hat die Eigenschaft blah«.

Rezept

1. Beginne den Beweis mit »Sei x ein beliebiges Ding, das so und so ist.«
2. Zeige nun, dass x die Eigenschaft blah hat.

Beweisrezept: *Beweise strukturieren* A-3

Ziel

Der Leser soll immer genau wissen, was bereits gezeigt wurde und was noch zu zeigen ist.

Rezept

Beweise werden schnell »unübersichtlich« werden. Abhilfe:

- Benutzen Sie Wendungen wie »Damit wurde gezeigt, dass … « oder »Es bleibt zu zeigen, dass … « oder »Im Folgenden zeigen wir zunächst …, … zeigen wir hingegen später.«
- Geben Sie am Anfang eines langen Beweises eine Übersicht und teilen Sie den Beweis in Abschnitte wie »Die Konstruktion« oder »Die Rückrichtung der Korrektheit der zweiten Unterkonstruktion«.
- Formulieren Sie Zwischenbehauptung als Lemmata, die Sie zu erst beweisen.

Beweisrezept: *Bootstrapping* A-4

Ziel

Beweisen, dass X vollständig ist für eine Zeit- oder Platzklasse C.

Rezept

Die Methode benutzt man, wenn man *noch kein* vollständiges Problem für C kennt.

1. Zeige zunächst $X \in C$, typischerweise durch Angabe einer Maschine mit dem korrekten Zeit- oder Platzverbrauch.
2. Fahre fort mit: »Sei nun $A \in C$ beliebig. Sei M eine Maschine, die A in Zeit/Platz XYZ entscheidet. Wir zeigen, dass dann $A \leq_{\mathrm{m}}^{\log} X$ gilt.«
3. Benutze das Beweisrezept »Reduktionen beweisen«, um die Reduktion zu bauen.

A-5

Beweisrezept: *Details weglassen*

Ziel

Der Beweis soll kurz und knapp bleiben.

Rezept

1. Man lässt »langweilige« Teile des Beweises weg.
2. Freundlicherweise schreibt stattdessen »Man kann zeigen, dass…« oder »Auf den Nachweis, dass … gilt, wurde verzichtet«.

Vermeiden sollte man »Trivialerweise gilt …« oder »Offensichtlich gilt …«, dies reizt den Leser eher zu argumentieren, dass dies doch nicht so trivial ist.

A-6

Beweisrezept: *Entscheidbarkeit beweisen*

Ziel

Man will beweisen, dass eine Sprache L entscheidbar ist.

Rezept

1. Konstruiere eine Turing-Maschine M.
2. Beweise dann, dass $L(M) = L$ gilt (Beweisrezept »Korrektheit von DTMS«)
3. Beweise dann, dass M bei jeder Eingabe anhält. Gib dazu zu jeder Eingabe an, wie die Berechnung aussieht und in welcher Endkonfiguration sie endet (die Endkonfiguration braucht *nicht* akzeptierend sein).

A-7

Beweisrezept: *Fallunterscheidung*

Ziel

Es soll gezeigt werden, dass eine beliebige Behauptung gilt.

Rezept

1. Leite den Beweis mit »Wir machen eine Fallunterscheidung.« ein.
2. Man benennt zwei oder mehr beliebige Annahmen (»Fälle« genannt), die mit der Behauptung nichts zu tun haben brauchen, von denen aber in jeder Situation mindestens (besser: genau) eine zutreffen muss.
3. Für jede Annahme X schreibt man »Fall X:«, gefolgt von einem Beweis, dass X die Behauptung impliziert.

A-8

Beweisrezept: *Induktion*

Ziel

Es soll gezeigt werden, dass eine Behauptung für $n = 1,2,3,\ldots$ gilt.

Rezept

1. Zeige, dass die Behauptung für $n = 1$ gilt.
   (Tipp: Hier kann man probieren, einfach »trivial« zu schreiben, das wird meistens akzeptiert.)
2. Man nimmt an, dass die Behauptung für ein vorgegebenes n gilt (aber vielleicht nicht für andere n). Man folgere, dass die Behauptung dann doch auch für $n+1$ gilt.

Beweisrezept: *Konstruktive Beweise* A-9

Ziel
Es soll gezeigt werden, dass es ein Ding mit bestimmten Eigenschaften gibt.

Rezept
1. Man gibt an, wie ein bestimmtes Ding *konstruiert* werden soll.
2. Man zeigt nun, dass das konstruierte Ding die behaupteten Eigenschaften hat.

Der zweite Schritt wird gerne vergessen!

Beweisrezept: *Kontextfreies Pumping-Lemma anwenden* A-10

Ziel
Man will beweisen, dass eine Sprache L nicht kontextfrei ist.

Rezept
1. Beginne mit »Wir zeigen die Behauptung durch die Kontraposition des Pumping-Lemmas.«
2. Fahre fort mit »Sei n beliebig.«
3. Fahre fort mit »Dann ist $w = \ldots$ ein Element von $L \cap \Sigma^{\geq n}$.«, wobei natürlich »…« geeignet gewählt sein muss.
4. Fahre fort mit »Sei nun $w = xaybz$ eine beliebige Zerlegung mit $|ab| \geq 1$.«
5. Schließe mit »Wähle nun $i = \ldots$« und argumentiere, dass $xa^iyb^iz \notin L$.

Beweisrezept: *Korrektheitsbeweis für DFAs* A-11

Ziel
Man will beweisen, dass ein DFA *M eine Sprache L akzeptiert.*

Rezept
Zeige zwei Richtungen (Beweisrezept »Zwei Richtungen«):

1. Beginne mit »Sei $w \in L$ beliebig.« (Beweisrezept »All-Aussagen beweisen«). Gib dann die Folge von Zuständen an, die M bei Eingabe w durchläuft (Beweisrezepte »Konstruktiver Beweis«). Ende mit »Also gilt $\delta^*(q_0, w) \in Q_a$ und somit $L \subseteq L(M)$«.
2. Beginne mit »Sei $w \in L(M)$ beliebig.« (Beweisrezept »All-Aussagen beweisen«.) Fahre fort mit »Dann gilt $\delta^*(q_0, w) \in Q_a$. Seien q_1 bis q_n mit $q_n \in Q_a$ die Zwischenzustände der Berechnung des Automaten.« (Beweisrezept »Namen vergeben«.) Argumentiere dann, dass $w \in L$ gilt. Ende mit »Also gilt $w \in L$ und somit $L(M) \subseteq L$«.

Beweisrezept: *Korrektheitsbeweis für DTMs* A-12

Ziel
Man will beweisen, dass eine DTM *M eine Sprache L akzeptiert.*

Rezept
Zeige zwei Richtungen:

1. Beginne mit »Sei $w \in L$ beliebig.« Gib die Folge von Konfigurationen an, die M bei Eingabe w durchläuft. Ende mit »Also gilt $C_{\text{init}}(w) \vdash^* C$, wobei C eine akzeptierende Endkonfiguration ist. Somit gilt $L \subseteq L(M)$«.
2. Beginne mit »Sei $w \in L(M)$ beliebig.« Fahre fort mit »Dann gilt $C_{\text{init}}(w) = C_1 \vdash \cdots \vdash C_n$, wobei C_n eine akzeptierende Endkonfiguration ist.« Argumentiere dann, dass $w \in L$ gilt. Ende mit »Also gilt $w \in L$ und somit $L(M) \subseteq L$«.

A-13

Beweisrezept: *Korrektheitsbeweis für Grammatiken*

Ziel

Man will beweisen, dass eine Grammatik G eine Sprache L erzeugt.

Rezept

Zeige zwei Richtungen (Beweisrezept »Zwei Richtungen«):

1. Beginne mit »Sei $w \in L$ beliebig.« (Beweisrezept »All-Aussagen beweisen«). Argumentiere dann, dass $S \Rightarrow_G^* w$ gilt, indem eine Ableitungskette konstruiert wird (Beweisrezepte »Konstruktiver Beweis« und häufig auch »Fallunterscheidung«). Ende mit »Also gilt $S \Rightarrow_G^* w$ und somit $L \subseteq L(G)$«.
2. Beginne mit »Sei $w \in L(G)$ beliebig.« (Beweisrezept »All-Aussagen beweisen«.) Fahre fort mit »Dann gilt $S \Rightarrow^* w$. Sei $S = w_1 \Rightarrow \cdots \Rightarrow w_n = w$ die Ableitungskette.« (Beweisrezept »Namen vergeben«.) Argumentiere dann, dass $w \in L$ gilt. Ende mit »Also gilt $w \in L$ und somit $L(G) \subseteq L$«.

A-14

Beweisrezept: *Korrektheitsbeweis für NFAs*

Ziel

Man will beweisen, dass ein NFA *M eine Sprache L akzeptiert.*

Rezept

Zeige zwei Richtungen (Beweisrezept »Zwei Richtungen«):

1. Beginne mit »Sei $w \in L$ beliebig.« (Beweisrezept »All-Aussagen beweisen«). Gib dann die Folge von Konfigurationen an, die M bei Eingabe w durchläuft (Beweisrezepte »Konstruktiver Beweis«). Ende mit »Also gilt $(q_0, w) \vdash^* (q_n, \lambda)$ mit $q_n \in Q_a$ und somit $L \subseteq L(M)$«.
2. Beginne mit »Sei $w \in L(M)$ beliebig.« (Beweisrezept »All-Aussagen beweisen«.) Fahre fort mit »Dann gilt $(q_0, w) \vdash \cdots \vdash (q_n, \lambda)$ mit $q_n \in Q_a$.« (Beweisrezept »Namen vergeben«.) Argumentiere dann, dass $w \in L$ gilt. Ende mit »Also gilt $w \in L$ und somit $L(M) \subseteq L$«.

A-15

Beweisrezept: *Kreisschluss*

Ziel

Es soll gezeigt werden, dass mehrere Behauptungen gleichwertig sind.

Rezept

Zeige, dass die Aussagen sich kreisförmig implizieren. Dies geschieht wie folgt:

1. Nimm an, dass die erste Aussage gilt. Folgere, dass nun auch die zweite gelten muss.
2. Beginne nun eine neue Argumentation. Nimm an, dass die zweite Aussage gilt. Folgere, dass dann die dritte gelten muss.
3. Und so weiter bis zur letzte, für die man zeigt, dass sie die erste impliziert.

Falls nun eine Aussage gilt, so gelten folglich alle.

A-16

Beweisrezept: *Namen für Teile vergeben*

Ziel

Man möchte über die Teile eines Dings »reden«.

Rezept

1. Ist das Ding ein Tupel mit einer festen Anzahl Komponenten (wie zum Beispiel eine Grammatik, die immer aus vier Teilen besteht), so schreibt man »Sei $G = (N, T, S, E)$ eine Grammatik…« oder »Sei $G = (V, E)$ ein Graph…«.
2. Ist das Ding ein Tupel mit einer variablen Anzahl Komponenten, so schreibt man »Sei $(x_1, x_2, x_3, \ldots, x_n)$ das Tupel…«. Nebenbei hat man mit n auch einen Namen für die Länge des Tupels eingeführt.
3. Ist das Ding eine abzählbare Menge, so schreibt man »Sei $M = \{m_1, m_2, m_3, \ldots, m_n\}$« (bei endlichen Mengen) oder »Sei $M = \{m_1, m_2, m_3, \ldots\}$« (bei abzählbar unendlichen Mengen).

Beweisrezept: *Nichtregularität mit Nerode-Klassen* A-17

Ziel

Man will beweisen, dass eine Sprache L nicht regulär ist.

Rezept

1. Beginne mit »Wir zeigen, dass L unendliche viele Nerode-Klassen hat.«
2. Fahre fort mit »Sei $X = \{\dots\}$. Offenbar ist X unendlich. Wir zeigen, dass keine zwei Worte in X rechtsäquivalent sind.«
3. Gib dann für je zwei $x, y \in X$ ein Wort $w \in \Sigma^*$ an, so dass $xw \in L$ und $yw \notin L$ (oder umgekehrt).

Beweisrezept: *Pumping-Lemma anwenden* A-18

Ziel

Man will beweisen, dass eine Sprache L nicht von DFAs *akzeptiert werden kann.*

Rezept

1. Beginne mit »Wir zeigen die Behauptung durch die Kontraposition des Pumping-Lemmas.«
2. Fahre fort mit »Sei n beliebig.«
3. Fahre fort mit »Dann ist $w = \dots$ ein Element von $L \cap \Sigma^{\geq n}$.«, wobei natürlich »…« geeignet gewählt sein muss.
4. Fahre fort mit »Sei nun $w = xyz$ eine beliebige Zerlegung mit $|y| \geq 1$ und $|xy| \leq n$.«
5. Schließe mit »Wähle nun $i = \dots$« und argumentiere, dass $xy^iz \notin L$ gilt.

Beweisrezept: *Reduktionsmethode* A-19

Ziel

Beweisen, dass X_{neu} vollständig ist für eine Klasse C.

Rezept

Die Methode benutzt man, wenn man *bereits vollständige Probleme für C kennt.*

1. Zeige zunächst $X_{\text{neu}} \in C$, typischerweise durch Angabe einer Maschine mit dem korrekten Zeit- oder Platzverbrauch.
2. Suche ein geeignetes für C vollständiges Problem X_{alt} aus.
3. Benutze das Beweisrezept »Reduktionen beweisen«, um $X_{\text{alt}} \leq_{\text{m}}^{\log} X_{\text{neu}}$ zu beweisen.

Beweisrezept: *Reduzierbarkeit beweisen* A-20

Ziel

Beweisen, dass A auf B reduzierbar ist.

Rezept

1. Beginne mit: »Wir zeigen, dass $A \leq_{\text{m}}^{\log} B$ via einer logspace-berechenbaren Funktion f gilt.«
2. Man überlegt sich dann, wie für beliebiges x die Frage »Ist $x \in A$?« in die Frage »Ist $f(x) \in B$?« umgewandelt werden kann. Beschreibe, wie f funktioniert.
3. Zeige dann, dass für alle $x \in \Sigma^*$ gilt $x \in A \iff f(x) \in B$. Zeige dazu zwei Richtungen:
   - »Sei $x \in A$. Dann …. Also gilt auch $f(x) \in B$.«
   - »Sei $f(x) \in B$. Dann …. Also gilt auch $x \in A$.«
4. Ende mit: »Für die Berechnung von f werden nur konstant viele Zähler benötigt, weshalb f logspace-berechenbar ist.«

A-21

Beweisrezept: *Trennung der Ebenen*

Ziel

Die zwei Beweisebenen sollen dem Leser klar werden.

Rezept

1. Für die »Objekte der Logik« benutzen wir *Formeln* und *Symbole*.
2. Für die Metaebene benutzen wir *normalsprachlichen Text*.

Beispiel: Gute Schreibweise

… Falls $\hat{\beta}(\psi) = 0$, so gilt $\hat{\beta}(\psi \to \varphi) = 1$. …

Beispiel (Schlechte Schreibweise)

… $\hat{\beta}(\psi) = 0 \to \hat{\beta}(\psi \to \varphi) = 1$. …

A-22

Beweisrezept: *Unentscheidbarkeit beweisen*

Ziel

Beweisen, dass eine Sprache L unentscheidbar ist (also $L \notin$ REC).

Rezept

1. Man sucht sich eine Sprache U aus, von der bereits bewiesen wurde, dass sie unentscheidbar ist.
2. Beginne mit: »Wir zeigen, dass aus der Entscheidbarkeit von L die Entscheidbarkeit von U folgen würde – aber wir wissen schon, dass U unentscheidbar ist.«
3. Fahre fort mit: »Nehmen wir also an, L wäre entscheidbar via eines Entscheiders M_L. Dann lässt sich U via folgendem Entscheider M_U entscheiden: Bei Eingabe w berechnet M_U …« Hier ergänzt man passend, wie M_U das Wort w modifiziert und M_L anwendet.
4. Ende mit: »Also ist M_U total und akzeptiert gerade U – und ist mithin ein Entscheider für U.«

A-23

Beweisrezept: *Wahrheitstafeln für Tautologien*

Ziel

Zeigen, dass eine Formel eine Tautologie ist.

Rezept

1. Identifiziere alle vorkommenden Variablen oder geeignete Teilformeln.
2. Für jede mögliche Kombination von Wahrheitswerten der Variablen oder Teilformeln verifiziere, dass die Formel zu wahr auswertet.

A-24

Beweisrezept: *Widerspruchsbeweis*

Ziel

Eine Behauptung beweisen, indem man zeigt, dass die Annahme des Gegenteils zu einem Widerspruch führt.

Rezept

1. Leite den Beweis mit »Wir führen einen Widerspruchsbeweis.« oder »Zum Zwecke des Widerspruchs nehmen wir an, dass XYZ nicht gilt.«
2. Führe nun einen Beweis, in dem die Annahme »nicht XYZ« beliebig benutzt werden darf.
3. Beende den Beweis, wenn doch »XYZ« hergeleitet wurde oder wenn ein offensichtlich falsche Behauptung wie $1 = 2$ hergeleitet wurde, mit »Widerspruch.« oder »Dies ist aber ein Widerspruch zur Annahme, dass nicht XYZ gilt, weshalb doch XYZ gelten muss«.

Beweisrezept: *Zugehörigkeit zu NP* A-25

Ziel

Beweisen, dass $A \in \text{NP}$ gilt.

Rezept

1. Man überlegt sich, wie eine »Lösung« zu einer Eingabe aussieht.
2. Beginne mit: »Die Lösungen zu einer Eingabe x lauten…«
3. Erkläre, weshalb es genau dann eine Lösung zu x gibt, wenn $x \in A$ gilt.
4. Erkläre, weshalb alle Lösungen zu x höchstens Länge $|x|^{O(1)}$ haben.
5. Erkläre, wie man in polynomieller Zeit überprüfen kann, ob ein Wort eine Lösung zu einem x ist.

Einzelne oder sogar alle Erklärungen können weglassen werden, wenn sie »offensichtlich« sind.

Beweisrezept: *Zwei Beweisrichtungen* A-26

Ziel

Es soll gezeigt werden, dass eine Behauptungen A genau dann gilt, wenn eine andere Behauptung B gilt.

Rezept

Zeige, dass die Aussagen sich gegenseitig implizieren:

1. Beginne mit »Es sind zwei Richtungen zu zeigen.«
2. Fahre fort mit »Für die erste Richtung nehmen wir an, dass A gilt.« Folgere, dass dann auch B gelten muss.
3. Beginne einen neuen Absatz mit »Für die Rückrichtung nehmen wir an, dass B gilt.« Folgere, dass dann auch A gelten muss.