Vorwort und Lesehinweise

Die theoretische Informatik ist ein lebendiges Forschungsgebiet, das schöne und praxisrelevante Resultate hervorbringt. Von Studierenden wird sie dagegen vor allem am Anfang oft als schwieriges Fach wahrgenommen, das aufwendige Einarbeitung in trockene Formalismen erfordert. Das liegt vor allem daran, dass eine „kritische Masse" an Verständnis nötig ist, um routiniert mit den theoretischen Konzepten umgehen zu können.

Anfangs muss also viel Theorie gelernt werden, bevor irgendwann die Erleuchtung kommt. Aber muss das so sein? Kann man die theoretische Informatik nicht einfach **ganz praktisch** angehen? Na ja, Sie ahnen es… Obwohl es der Titel verspricht, ist Theorie eben nicht praktisch und soll es auch nicht sein – sonst hieße sie nicht Theorie. Aber theoretische Überlegungen haben (meist) einen praktischen Nutzen oder sie entstanden wenigstens ursprünglich aus der Notwendigkeit heraus, praktische Probleme zu lösen. Von da an verselbstständigen sie sich allerdings gerne und verschleiern ihren Praxisbezug im Lauf der Zeit immer mehr. Die paradoxe Situation in der Informatik ist, dass relativ viel Theorie nötig ist, um die praktische Relevanz dieser Theorie zu verstehen.[1]

Zum Glück mag der Informatiker Paradoxes. (Wenn er etwas beweist, dann meist durch Widerspruch.) Wir werden daher trotz des vermeintlichen Widersinns versuchen, den Einstieg in die Theorie durch möglichst einleuchtende Bezüge zur Praxis zu erleichtern. Genauer gesagt heißt das, dass wir **den praktischen Kern** eines zu betrachtenden Konzepts darstellen werden, noch bevor das Konzept selbst überhaupt eingeführt wurde. Alle Facetten des Konzepts können durch diesen ersten unscharfen Eindruck natürlich noch nicht abgehandelt werden – und das lässt den strengen Formalisten gleich skeptisch werden. Aber die Grundideen intuitiv verstanden zu haben, macht es für Sie wesentlich einfacher, später den formalen Definitionen zu folgen. Anderenfalls kann es passieren, dass Studierenden erst nach mehreren Semestern des geduldigen Schluckens von Definitionen und Formeln der Sinn hinter manchen Konzepten zu dämmern beginnt.

Der strenge Formalist hat allerdings nicht unrecht; theoretische Konzepte allzu praktisch zu zeichnen birgt immer die Gefahr von Missverständnissen. Eine praktische Sicht deckt sich nie exakt mit der Theorie (sonst wäre sie selbst Theorie), und weil sie oft intuitiver erscheint, zieht man sie gerne der formalen Wirklichkeit vor. Dabei tappen die meisten Studierenden jedoch glücklicherweise in dieselben typischen Fallen. Wir werden Ihnen helfen, diese Fallen zu umgehen, und zwar oft genug ein-

1 Relativ! Um die Kirche im Dorf zu lassen: Es gibt wesentlich schlimmere Gebiete als die theoretische Informatik. Die Aura des Schrecklichen speist sich vor allem aus Berichten Studierender, die mit falschen Erwartungen oder zu wenig Vorbereitung in Prüfungen versagten. Dieser beiden Probleme werden wir uns schon bald annehmen.

fach mit dem Hinweis: „Vorsicht, das ist eine Falle!“ Trotzdem ist beim Übergang von der Praxis zur Theorie Vorsicht geboten; allgemein sollte nie vergessen werden, dass überall, wo intuitive und formale Perspektive auseinandergehen, kompromisslos die formale Version gilt, auch wenn die intuitive „intuitiver“ erscheint. Kurz gesagt, werden wir uns der theoretischen Informatik also ganz praktisch nähern, aber sobald sie erreicht ist, wird sie sich – was sonst? – als ganz und gar theoretisch erweisen. Sie werden aber zu dem Zeitpunkt schon in der Lage sein, gerade das an ihr zu schätzen.

Nun kennen Sie die Theorie hinter dieser praxisorientierten Einführung; aber wie sollen Sie dieses Buch eigentlich lesen, um „möglichst viel daraus zu lernen“? Wir beantworten diese Frage zunächst auf typische Informatiker-Art (also so, dass der Beantwortungsprozess mehr wert ist als die eigentliche Antwort). Dabei nehmen wir übrigens eines der angekündigten „schönen Resultate“ vorweg, ohne dass Sie auch nur eine einzige Definition lernen mussten. (Und danach überlegen wir, wie Sie trotz der ersten unbefriedigenden – nicht ganz ernst gemeinten – Antwort durchaus konstruktiv an die Lektüre dieses Buches gehen können.)

Dieses Buch lesen – ein schwieriges Problem

Betrachten wir einmal folgende drei Fragestellungen, die eigentlich drei Varianten eines einzigen Grundproblems sind:

(1) Gibt es eine Art, dieses Buch zu lesen, sodass Sie nach der Lektüre in der Lage sind, Ihre Informatik-Grundlagenprüfung zu bestehen (**Entscheidungsvariante**)?
(2) Wenn das möglich ist, wie gut ist die beste Note, die Sie erzielen können (**Optimierungsvariante**)?
(3) Wie genau müssen Sie sich mit diesem Buch beschäftigen, um diese Bestnote zu erhalten (**Konstruktionsvariante**)?

Wäre es nicht schön, wenn Sie zumindest die Entscheidungsvariante, am besten aber gleich die Konstruktionsvariante für sich beantworten könnten? Nun werden Sie sagen, dass das schon aus Mangel an Informationen nicht möglich ist, denn Sie wissen ja nicht, wie die Prüfung aussehen wird, und Sie wissen auch nicht, was genau Sie in diesem Buch erwartet. Dann sagen wir eben, Sie dürfen das Buch beliebig intensiv durcharbeiten und die Prüfung beliebig oft wiederholen (Sie vergessen natürlich nach jedem Versuch die Prüfungsaufgaben wieder, sonst wäre es langweilig). Dann müssten diese Fragen doch zu beantworten sein… Unter Umständen ja, denn vielleicht ist das Buch toll, Sie schreiben auf Anhieb eine 1,0 und alle drei Fragen sind auf einmal beantwortet. Oder aber das Buch und die Prüfung passen ganz und gar nicht zusammen, und Sie sehen sofort, dass die Lektüre Ihnen auf keinen Fall beim Bestehen helfen kann; auch dann könnten Sie die Fragen beantworten, nur eben negativ. Das sind

die beiden typischen einfachen **Randbereiche** algorithmischer Probleme, für die wir „auf den ersten Blick“ die Lösung sehen.

Was aber, wenn das Buch in Ordnung zu sein scheint und Sie nach x Versuchen trotzdem nicht in der Lage waren, die Prüfung zu bestehen? Oder Sie haben nach y Versuchen höchstens eine 1,7 erhalten, würden aber gerne noch eine bessere Note schaffen? Gibt es eine Chance, irgendwann mit Sicherheit sagen zu können, dass es nicht mehr besser werden wird? Als Mensch würden Sie vermutlich irgendwann aufgeben und sich mit dem bisher erreichten zufriedengeben. Aber nehmen wir an, Sie wären sehr ehrgeizig und **müssten** (wie ein Computer) das Problem unbedingt lösen. Sie tüfteln mit komplizierten mathematischen Formeln herum, analysieren den Inhalt des Buches und Ihre eigenen Charaktereigenschaften, und nach einer Weile sind Sie tatsächlich der Meinung, einen Beweis gefunden zu haben, dass Sie niemals bestehen werden. (Etwa weil Sie bisher nicht bestanden haben, und es neurologisch nicht möglich ist, noch mehr Wissen in Ihren Kopf zu bekommen.) Es war nicht einfach, Sie sind weder dumm noch schlau genug, das Buch weder gut noch schlecht genug, damit das Problem in einen der überschaubaren Randbereiche fällt – aber zuletzt ist Ihnen eine längliche Argumentation eingefallen, die eindeutig zeigt, dass Sie immer durchfallen müssen.

Nun haben Sie sich aber bereits für einen weiteren Prüfungstermin angemeldet, und spaßeshalber gehen Sie ein letztes Mal hin. Sie erhalten den Prüfungsbogen, auf dem nur eine einzige Aufgabe steht: „Beweisen Sie, dass Sie nicht in der Lage sind, diese Prüfung zu bestehen.“ Toll, denken Sie, das kann ich! Sie schreiben Ihren Beweis hin und… bestehen. – Oder…? Müsste Ihr Beweis dann nicht falsch gewesen sein? Aber wie können Sie mit einem fehlerhaften Beweis bestanden haben? Oder Sie bestehen eben nicht, **weil** Ihr Beweis falsch war. Darin liegt kein Widerspruch, aber wie man es dreht, der Beweis kann auf keinen Fall korrekt gewesen sein. **Es kann so einen Beweis prinzipiell nicht geben!** Wir schlussfolgern daher, dass keine der eingangs gestellten Fragen – nicht einmal die Entscheidungsvariante – allgemein beantwortet werden kann.

Sie glauben, niemand würde so eine gemeine Prüfungsfrage stellen? Ein Informatiker schon! Viel mehr als die auf Sie persönlich bezogenen Eingangsfragen interessiert ihn nämlich die Meta-Frage: ob diese Fragen überhaupt von „irgendwem“ allgemein beantwortet werden können. Der Beweis, um den es ging, war also nicht der, den Sie zu führen glaubten, sondern der, der zeigt, dass Sie mit Ihrem Beweis gescheitert sein müssen. Normalerweise untersuchen wir natürlich nicht die Problemlösefähigkeit von Menschen, sondern die von (anderen) „Rechnern“. Die wichtige Schlussfolgerung ist daher, dass kein **Rechner** in der Lage ist, ein Problem zu lösen, das in der obigen Art gestellt ist. Das ist eine wichtige Erkenntnis, denn wenn es Rechner gäbe, die **alles** berechnen könnten, was wir uns vorstellen können, würden große Bereiche der theoretischen Informatik einfach wegfallen.

Den Kerngedanken, der im Beispiel den angeblichen Beweis zum Widerspruch führte, hat bereits der Informatik-Pionier Alan Turing entwickelt: Es geht darum, den

Rechner selbst (**rekursiv**) zum Teil des Problems zu machen. Wird eine Frage, wie „Kann ich, der Rechner, beweisen, dass ich (nicht) in der Lage bin, dies und das zu tun?“, geschickt gestellt, kann sie zu einem Widerspruch geführt werden, und man erhält als Ergebnis **algorithmisch nicht lösbare Probleme**. Unser obiges (direkt aus dem Leben gegriffenes) unlösbares Problem war: „Kann ich als Mensch beweisen, dass ich (nicht) in der Lage bin, eine gewisse Art von Prüfungen zu bestehen – egal wie die konkrete Prüfung aussieht?“ Turing zeigte auf eine ganz ähnliche Art, dass Rechner nicht in der Lage sind zu entscheiden, ob ein Programm anhalten oder endlos weiterlaufen wird. Diese einfache Idee war der Beginn aller Berechenbarkeits- und Komplexitätsbetrachtungen, welche inzwischen zu den Königsdisziplinen der theoretischen Informatik gehören. Bevor wir uns mit ihnen in den Kapiteln 6 und 7 formal befassen, muss noch etwas Vorarbeit geleistet werden – aber einige wichtige Grundgedanken haben Sie gerade schon kennengelernt.

Was steckt in diesem Buch und wie holt man es heraus?

Der gerade geführte „Beweis“ ist nur in Grenzen auf die Menschenwelt übertragbar. Es gibt wohl viele Prüfungssituationen (hoffentlich auch Ihre anstehende), auf die er nicht angewendet werden kann. Daher sollten Sie sehr wohl versuchen, für sich den bestmöglichen Weg zu finden, um sich vorzubereiten. Dafür ist es zunächst wichtig zu wissen, für welche Zielgruppe das Buch überhaupt geschrieben wurde.

Es gibt viele gute Lehrbücher zur theoretischen Informatik, und einige davon werden in diesem Buch auch rege referenziert[2]. Schwierig ist es allerdings, Bücher zu finden, die auch für Anfänger gut geeignet sind. Das liegt an der schon erwähnten „kritischen Masse“ an theoretischem Vorwissen: Es gibt keine „Theorie light“, die „nicht so sehr in die Tiefe geht“; eine gewisse Mindesttiefe ist immer erforderlich, um überhaupt etwas zu verstehen. Meist wird dieses Vorwissen entweder als bekannt vorausgesetzt oder die Konzepte müssen zunächst mehr oder weniger neutral zur Kenntnis genommen werden, bevor sich manchmal erst viel später ihr Zweck erschließt.

Hier wagen wir den Versuch, von Anfang an zu jedem Konzept den Praxisbezug gleich mitzuliefern, damit die Theorie nicht im leeren Raum stehen muss. Wir holen dafür ein bisschen weiter aus als es andere Lehrbücher tun, und manches wird doppelt beschrieben – einmal salopp und informell, einmal präzise und formal. Dieser doppelte Blick aus verschiedenen Perspektiven vertieft aber das Verständnis für den Stoff – solange die erwähnten Praxisfallen erfolgreich umgangen werden. Als weiterer Vorteil entsteht auf diese Weise ein natürlicher roter Faden, der sich nicht nur mathematisch, sondern auch konzeptuell von Beginn an durch die dargestellten Ideen zieht.

2 Als Standardwerk ist vor allem das Buch „Einführung in Automatentheorie, Formale Sprachen und Berechenbarkeit“ von Hopcroft, Motwani und Ullman zu erwähnen [HMU11].

Darüber hinaus versuchen wir, die **typischen Verständnisprobleme** – von denen es eine ganze Reihe gibt – direkt anzusprechen und aus dem Weg zu räumen. Um eilige Leser nicht zu bremsen, werden solche Zusatzinformationen meist klar vom Haupttext abgetrennt, sodass sie auch übersprungen werden können (vgl. ***$e^{i\pi}$*-Fallen**, **Exkurse** etc. weiter unten). Eine wichtige Quelle von Verständnisproblemen sind eigentlich einfache Gedankengänge, die aber durch einen ungewöhnlichen Widerspruch (wie beim oben geführten „Beweis“) oder durch eine gerade umgekehrte Sicht auf eine logische Aussage unintuitiv werden. So sei schon jetzt als Warnung gesagt, dass es bei logischen Implikationen

$$A \implies B$$

fast nie darum gehen wird, aus der Gültigkeit von A zu folgern, dass auch B gilt. Aus irgendeinem Grund folgert der Informatiker so gut wie immer anders herum: **dass die Nicht-Gültigkeit von *B* zur Nicht-Gültigkeit von *A* führt**. Behält man das im Kopf, entstehen viele Missverständnisse erst gar nicht.

Es gibt in der theoretischen Informatik auch exotische Bereiche, die schwer in Bezug zu (heutigen) realen Anwendungen zu setzen sind; solche Themen werden wir weitgehend aussparen. Aber auch innerhalb der unvermeidbaren Mindesttiefe (Sie werden sehen, dass wir durchaus die Tiefe der üblichen Standardwerke erreichen werden[3]) muten manche Themen der Berechenbarkeits- und Komplexitätstheorie recht exotisch an. Auch diese Themen werden möglichst anfängertauglich präsentiert, aber dennoch wird es für jeden, der sich das erste Mal damit beschäftigt, nötig sein, diese Kapitel sorgfältig zu lesen, womöglich manches mehrfach. Vielleicht werden Sie im Verlauf des Lernprozesses mehrmals denken, dass Sie es verstanden haben, dann aber doch neue Nuancen entdecken, die Sie übersehen hatten. Wenn diese Erleuchtung so in etwa das dritte oder vierte Mal aufgetreten ist, haben Sie es wirklich verstanden.

Nichtsdestotrotz sind gerade die Themen Komplexität und Berechenbarkeit von großer **praktischer** Bedeutung für die Informatik. Sie können helfen, im Alltag viel Zeit zu sparen, indem sie einerseits besonders effiziente Algorithmen liefern, oder andererseits durch Negativaussagen, wie „besser als wir das sowieso schon tun, geht es gar nicht“ unnötiges Optimieren verhindern. In einem gewissen Sinne kann man die „handfesteren“ Themen, wie Automaten und Grammatiken, lediglich als Werkzeuge betrachten, die benötigt werden, um eine Komplexitäts- und Berechenbarkeitstheorie zu formulieren. Positiv formuliert heißt das aber auch, dass Sie, wenn Sie die handfesten Themen gründlich verstanden haben, schon einen wesentlichen Teil des Weges zum Verständnis der Komplexitäts- und Berechenbarkeitstheorie gegangen sind.

3 Mit einer Ausnahme: Wir werden nicht alle Beweise bis ins letzte Detail formal ausführen, sondern stattdessen versuchen, umso ausführlicher die zentralen Ideen darzustellen.

Die $e^{i\pi}$-Falle: Ein Wegweiser für Gefahrenstellen

Aus den Überlegungen zur Mindesttiefe geht hervor, dass jedes Buch über theoretische Informatik (wenn es nicht gerade Tausende Seiten lang oder sehr oberflächlich ist), Aussagen oder Schlussfolgerungen enthalten muss, die beim ersten Lesen nicht sofort verständlich sind. Typischerweise verstecken sich solche schwierigen Stellen in einer Kette einfach nachzuvollziehender Argumentationsschritte, weshalb sie im Lesetrott leicht übersehen werden. Dadurch bleibt oft ausgerechnet der Knackpunkt einer Argumentation unbeachtet, der für das Verständnis des Themas am wichtigsten gewesen wäre. In einem Comic aus der „xkcd“-Reihe (http://xkcd.com/179) zeigt der Zeichner Randall Munroe diese Eigenheit mathematischer Argumentationsketten am Beispiel der **eulerschen Identität**. Gegeben sei etwa die folgende Aussagenliste, die (hoffentlich so nicht) aus einem Mathematikbuch der Oberstufe stammen könnte:

1) Die Kreiszahl π bezeichnet den Umfang des Einheitskreises und hat einen Wert von etwa $3{,}1415927$.
2) Die Basis des natürlichen Logarithmus ist die irrationale Zahl $e \approx 2{,}7182818$.
3) Die imaginäre Konstante $i = \sqrt{-1}$ wird eingeführt, um mit Wurzeln negativer Zahlen rechnen zu können.
4) Daraus ergibt sich die eulersche Identität als: $e^{i\pi} = -1$.

Die ersten drei Aussagen werden die meisten Leser kennen und beim Lesen einfach abnicken. Die vierte Aussage ist jedoch alles andere als trivial und ergibt sich nur sehr indirekt aus den vorherigen. Trotzdem erwischt man sich in der Routine des Abnickens schnell dabei, auch diese komplizierte Formel innerlich ad acta zu legen, ohne ihre Herleitung wirklich verstanden zu haben. In diesem Buch werden solche versteckten verständnisintensiven Stellen als ***$e^{i\pi}$*-Fallen** bezeichnet.

Natürlich würde ein gutes Lehrbuch die obige Aussagenliste um viele Zwischenschritte erweitern. Trotzdem sind einige davon eingängiger und andere erfordern einen größeren geistigen Aufwand. Um die Selbstkontrolle beim Lesen zu erleichtern, werden wir das folgende Symbol am Seitenrand verwenden, um Stellen zu kennzeichnen, die erfahrungsgemäß gerne als Knackpunkte übersehen werden:

4) Daraus ergibt sich die eulersche Identität als: $e^{i\pi} = -1$.

Wenn Sie auf dieses Symbol stoßen, sollten Sie sich vergewissern, dass Sie den entsprechenden Argumentationsschritt, der oft eine Hauptschwierigkeit für das Verständnis darstellt, begriffen und nicht einfach nur abgenickt haben.

In so einem Kasten wird eine genauere Erklärung der Falle oder ein Hinweis zu ihrer Vermeidung gegeben.

Wir werden die Kästen auch verwenden, um gefährlich anschauliche Beispiele oder besonders heimtückische Praxisvergleiche zu kennzeichnen. Allzu „knackige Beispiele“ und allzu „typische praktische Anwendungen“ neigen nämlich dazu, wesentliche Aspekte der eigentlichen Aussage zu verschleiern; das liegt nicht am Beispiel selbst, sondern am Leser, der hofft aus dem Beispiel schon die ganze Vielfalt einer Idee ablesen zu können – was nur selten der Fall ist. Trotzdem werden wir Beispiele und Praxisbezüge großzügig verwenden, aber dabei stets im Hinterkopf behalten, dass die gesamte Fülle einer mathematischen Aussage nicht durch Beispiele erschlossen werden kann – und ihrem Wesen nach nicht praktisch ist.

Außerdem wird es neben der normalen $e^{i\pi}$-Box hier und da einen **Exkurs-Kasten** mit weiterführenden Informationen geben, der für das Verständnis der Grundidee nicht unbedingt notwendig ist, aber die Details noch etwas gründlicher beleuchtet.

∃ Das große „∃“-Symbol wurde als Kennzeichnung der Exkurs-Kästen gewählt, weil es einerseits für das E aus Exkurs steht und andererseits dafür, dass so ein Kasten nicht „∀“ – für alle Leser ist. Andererseits sind die Exkurse aber oft nichts anderes als etwas tiefgreifendere $e^{i\pi}$-Boxen. Es kann also sicher nicht schaden, sich auch mit ihnen zu beschäftigen.

Nutzung der Skripte und des XWizard-Werkzeugs

XWizard ist der Name eines Programms, mit dem viele der theoretischen Konzepte aus dem Buch – ganz praktisch – visualisiert und bearbeitet werden können. Es ist im Rahmen der Lehrveranstaltungen entstanden, an denen die Autoren beteiligt sind, und kann als **Web-Version** einfach über folgende Adresse erreicht werden:

www.xwizard.de

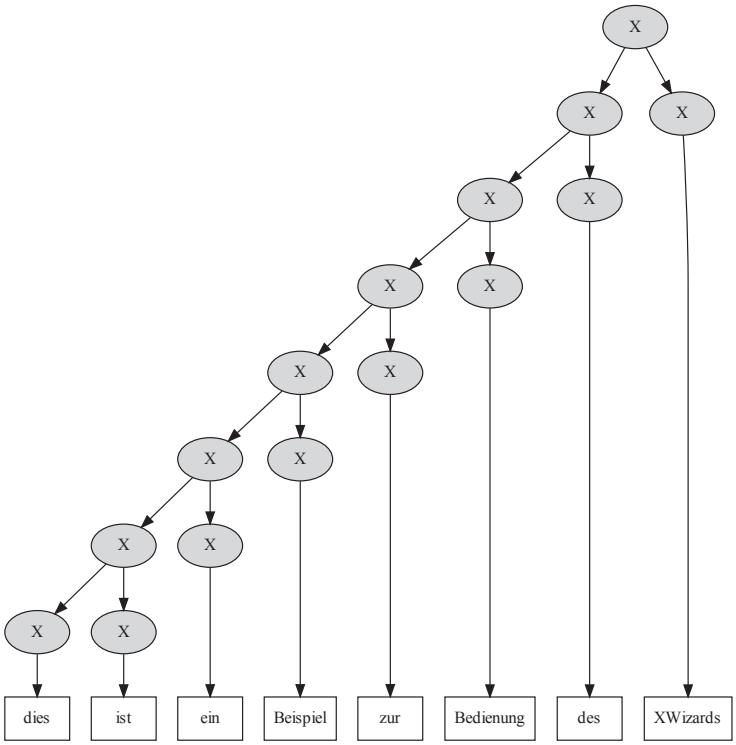
Darüber hinaus existiert eine fast funktionsgleiche **Download-Version**, die für Windows vorkompiliert oder als vollständiger Java-Quellcode mit Dokumentation heruntergeladen werden kann:

www.dasinfobuch.de/links/vfp

Der XWizard arbeitet mit sogenannten **Skripten**, die vom Benutzer eingegeben und vom Programm in eine graphische Darstellung übersetzt werden. Skripte sind also Texte, die ein Objekt, beispielsweise einen Kellerautomaten, eine Grammatik oder einen regulären Ausdruck beschreiben. Das zu einem Skript gehörende Objekt kann einfach angezeigt oder durch **Konversionsmethoden** in ein neues Skript überführt werden. Zum Beispiel kann ein Kellerautomat durch eine Konversionsmethode schrittweise simuliert werden, indem im Skript die Variable für das Simulieren hochgezählt wird (Kellerautomaten werden im Buch in Kapitel 2.3 behandelt). Oder ein endlicher Automat kann durch eine Konversionsmethode minimiert werden, indem ein neues Skript erzeugt wird, das für den minimierten Automaten steht (die Minimierung endlicher Automaten finden Sie in Kapitel 2, ab Seite 72).

Werden im Buch Konzepte behandelt, die im XWizard bearbeitet werden können, wird immer auch das zugehörige Skript und ein Link zur Web-Version angegeben. Beispielsweise könnte folgende Darstellung einer Grammatik im Buch auftauchen:

$$G = (\{X\}, \{Bedienung, Beispiel, XWizards, des, dies, ein, ist, zur\}, P, X)$$
$$P = \{X \rightarrow Bedienung \mid Beispiel \mid XWizards \mid des \mid dies \mid ein \mid ist \mid zur \mid XX\}$$



DAR-01 | ID-16356 → *Skript: S. 385*

Unter der Abbildung steht zunächst eine eindeutige **Nummer der Darstellung** (hier „DAR-01“), welche über das ganze Buch fortlaufend hochgezählt wird. Danach folgt eine sogenannte **XWizard! Skript-ID** (hier „ID-16356“) und zuletzt ein Verweis auf die **Seite in Anhang B**, wo das Skript selbst und weitere Informationen abgedruckt sind. Ein Skript kann auf verschiedene Arten in den XWizard geladen werden:

(1) In Anhang B ist ein zum Skript gehörender QR-Code abgedruckt. Wird er eingescannt, wird automatisch die XWizard-Website aufgerufen und das Skript geladen. Für die obige Beispieldarstellung wäre das folgender QR-Code:

(2) Auf der XWizard-Website kann die Skript-ID auch direkt in das Eingabefeld eingetragen werden; nach einem Klick auf „Draw!“ wird das Skript geladen.

(3) Zuletzt ist im Anhang auch noch das komplette Skript abgedruckt, das sowohl in der Web- als auch in der Download-Version des XWizards in das Eingabefeld eingetragen werden kann (Skript-IDs funktionieren dagegen nur bei der Web-Version). Damit das Skript nicht vollständig abgetippt werden muss, sind alle Skripte auch unter der im Anhang angegebenen Adresse online abrufbar.

Normalerweise sind die Optionen (1) und (2) am bequemsten, um ein Skript zu laden. Option (3) ist eher als Backup-Lösung zu sehen, falls keine Internetverbindung verfügbar oder der Server nicht erreichbar sein sollte.

Der XWizard bietet zahlreiche Möglichkeiten, um die vorgegebenen Objekte weiter zu verarbeiten. So können aus Grammatiken Normalformen erzeugt, endliche Automaten deterministisch gemacht, Automaten und Grammatiken ineinander überführt werden und vieles mehr. All diese Funktionen werden auf den umfangreichen Hilfeseiten des XWizards beschrieben, die über den Link „Hilfe“ auf www.xwizard.de erreichbar sind. Darüber hinaus gibt es dort zwei PDF-Dokumente mit detaillierten Beschreibungen der Arbeitsweise des XWizards.

Wir werden die Arbeit mit dem XWizard oft dem Selbststudium überlassen, manchmal aber auch im Text konkrete Hinweise geben, wie die Ausgabe zu interpretieren ist oder welche Funktionen zum jeweiligen Thema passen. Diese Hinweise werden in solchen Kästen gegeben:

Versuchen Sie einmal, die oben angegebene Grammatik mit dem XWizard in die **Greibach-Normalform** zu überführen. (Diese Normalform wird im Buch ab Seite 239 behandelt; Sie müssen sie aber nicht verstanden haben, um diese Miniaufgabe zu lösen.)

In vielen Fällen kann der XWizard helfen, die im Buch dargestellten Konzepte genauer unter die Lupe zu nehmen und bei eventuell auftauchenden Verständnisschwierigkeiten Klarheit zu schaffen. Bleiben Fragen offen, kann das von der Website aus verlinkte **Forum** zum Diskutieren genutzt werden. Auch zu Übungszwecken lohnt es sich, den XWizard regelmäßig zu verwenden, denn theoretische Feinheiten gehen erst durch häufige Wiederholungen in Fleisch und Blut über. Der XWizard wird in diesem Buch als wichtiges Bindeglied zwischen Theorie und Praxis angesehen und entsprechend häufig referenziert.

Übungsaufgaben

Natürlich gehören als weiteres Bindeglied zwischen Theorie und Praxis eine Menge Übungsaufgaben zu einem Gesamtpaket der theoretischen Informatik. Um jedoch eine repräsentative Auswahl an Aufgaben in diesem Buch bereitzustellen, wäre wohl eine Verdopplung der Seitenzahl nötig gewesen. In Anbetracht der vielen aus Vorlesungen, dem Internet und speziellen Übungsbüchern verfügbaren Übungsaufgaben, erschien die entsprechende Erhöhung des Preises bei einer Verminderung der Flexibilität (zu schweigen von der Abschreckung der Leser) nicht angemessen. Daher werden Sie in diesem Buch eher Beispiele finden, die gleich mit umfassenden Erklärungen durchgerechnet werden. Außerdem wird jedes Kapitel durch **fünf Fragen zur Selbstkontrolle** abgeschlossen, mit welchen Sie prüfen können, ob Sie die wesentlichen Inhalte verstanden haben. (Warum fünf? Drei wären zu wenige gewesen und sieben schon wieder zu viele; vier oder sechs wären vielleicht noch im Rahmen gewesen, aber fünf ist eine Primzahl.) Diese fünf Fragen sind nicht mit einem Wort zu beantworten, sondern sollen zu einer möglichst umfassenden Rekapitulation des Stoffs anregen. Daher werden dazu auch keine Musterlösungen angeboten; vielmehr sollte die Lösung im Selbststudium bzw. in der Diskussion mit anderen erarbeitet werden. Um diesen Prozess zu erleichtern, wird im Anschluss an die Kontrollfragen jeweils auf ein Online-Forum verwiesen, wo Lösungsansätze oder sonstige Themen zur Diskussion gestellt werden können.

Die Auswahl weiterer Aufgaben zur Prüfungsvorbereitung wird im Prinzip der Leserin oder dem Leser selbst überlassen. Wird das Buch begleitend zu einer Vorlesung verwendet, werden Übungsaufgaben normalerweise ohnehin bereitgestellt. Wer allerdings Aufgaben aus demselben Guss sucht, sei verwiesen auf das Übungsbuch

> **100 Übungsaufgaben zu Grundlagen der Informatik,**
> **Band I: Theoretische Informatik**

das ebenfalls der Feder der Autoren dieses Lehrbuchs entsprungen ist [KPBS14]. Zur besseren Orientierung werden zum Abschluss jedes Kapitels die passenden Aufgaben aus diesem Buch in einer Exkurs-Box referenziert; ob Sie diese oder andere Aufga-

ben verwenden, bleibt aber Ihnen überlassen. Für weitere Informationen gibt es auch einen kleinen Webauftritt für dieses Lehrbuch und die beiden Bände der „100 Übungsaufgaben zu Grundlagen der Informatik“ unter folgender Adresse:

www.dasinfobuch.de

Wir wünschen Ihnen nun viel Erfolg und Freude bei Ihrer persönlichen, ganz praktischen Widerlegung der These, dass Sie die Informatik-Grundlagenprüfung nicht bestehen können!

Danksagung

Die Autoren danken für das Durchsehen der Texte hinsichtlich Grammatik und Stil sowie für kreative Hinweise aller Art (und auch für aufmunternde Worte in geistigen Dürreperioden) Marlon Braun, Christian Hirsch, Fabian Kern, Anna Mauser, Ingo Mauser, Johannes Müller, Fabian Rigoll, Sebastian Steuer, André Wiesner und Micaela Wünsche.

Karlsruhe, Juli 2016
Lukas König, Friederike Pfeiffer-Bohnen, Hartmut Schmeck

Inhalt

1 Auf dem Weg zur theoretischen Informatik

Die theoretische Informatik stellt, wie aus dem Namen hervorgeht, die theoretischen Grundlagen für die praktische Informatik dar. Ihr Ziel ist es also, die tägliche Arbeit in vielen Bereichen der Informatik zu strukturieren und dadurch zu erleichtern.

$e^{i\pi}$

Die „Informatik“ als Gebiet ist alles andere als homogen. Man kann sie aus ihrer Entstehungsgeschichte heraus als Teilgebiet der Mathematik betrachten, durch eine Auflistung ihrer Teildisziplinen beschreiben oder sie nach ihrem theoretischen Nutzen erfassen – etwa als den „machbaren Teil der Mathematik“. Wir werden uns hier der Informatik einerseits begrifflich als der „Wissenschaft der Informationen“ nähern, andererseits aber auch pragmatisch feststellen, dass sich der Informatiker typischerweise mit gewissen Dingen beschäftigt. Dazu gehören etwa das Programmieren von Software, Apps, Web-Applikationen usw., das Entwerfen von Hardware, der Aufbau von Rechnernetzen usw. usf. Die meisten dieser Tätigkeiten können auf einen gemeinsamen Kern zurückgeführt werden: das Lösen einer bestimmten Art mathematischer Probleme. Anhand dieser Tätigkeiten werden wir überlegen, was die theoretische Informatik als Disziplin leisten soll.

Eine wichtige Aufgabe der theoretischen Informatik ist zunächst die Formalisierung der elementaren Begriffe, die bei der Arbeit eines Informatikers auftreten, etwa:

- Rechner, Rechnung, berechenbar,
- Algorithmus,
- Problem,
- Komplexität,
- Effizienz,
- …

Sind solche Formalisierungen vorhanden, dann ist der nächste Schritt, mathematische Lösungen für Fragestellungen zu erarbeiten, mit denen Informatiker konfrontiert werden. Dabei handelt es sich beispielsweise um Fragen, wie: „Kann ein gegebenes Problem unter gewissen Randbedingungen (etwa in einer bestimmten Zeit) auf einem Rechner gelöst werden?“

Um zu verstehen, welche Elemente, Prozesse und Fragestellungen die theoretische Informatik untersuchen soll, wollen wir im Folgenden erschließen, womit sich die Informatik im Kern überhaupt beschäftigt, und überlegen nebenbei, welche Schritte bei einer Formalisierung nützlich sein könnten.

1.1 Information: der Stoff der Informatik

Die Informatik ist vom Begriff her und auch ihrem Ursprung nach die Wissenschaft vom **automatischen Umgang mit Informationen**. Wir überlegen zunächst, was „Informationen" sind und danach, wie man „automatisch mit ihnen umgehen" kann und was das mit Rechnen und Computern (im englischsprachigen Raum redet man von „Computer Science") zu tun hat.

$e^{i\pi}$ Unter **Informationen** (auch **Daten**), verstehen wir zunächst sehr allgemein etwas, das mit dem Ziel erzeugt wurde, Wissen zu enthalten und dieses von einem **Sender** an einen **Empfänger** zu übermitteln.

Heute wird im Bereich von **Big Data** diese Definition sogar noch weiter gefasst. Daten werden hier gesammelt, ohne dass im Vorfeld bekannt ist, wie sie später genutzt werden sollen. Begriffe wie Sender, Empfänger oder sogar Inhalt müssen dann sehr flexibel aufgefasst werden. Diese flexible Sicht widerspricht den klassischen Definitionen prinzipiell nicht und soll uns im Folgenden nicht stören.

Eine Flaschenpost etwa könnte eine Information sein, die Wissen über den Aufenthaltsort eines Schiffbrüchigen (Sender) an einen potentiellen Retter (Empfänger) übermitteln soll. In diesem Beispiel wird die Flasche nicht an einen bestimmten sondern an irgendeinen Empfänger geschickt; sie könnte unter Umständen eine weite Strecke vom Sender zum Empfänger zurücklegen und dabei eine lange Zeit benötigen.

Sowohl Wegstrecke als auch Zeitdauer können aber auch ganz andere Größenordnungen annehmen. So können Sender und Empfänger ein und dieselbe Person sein, etwa wenn jemand Tagebuch schreibt und somit Informationen an sich selbst in der Zukunft sendet. Dasselbe gilt in einer kürzeren Zeitperiode für jemanden, der sich notiert: „Nicht vergessen, Oma vom Flughafen abholen".

$e^{i\pi}$ Wie im letzten Beispiel dienen in der Informatik Informationen grundsätzlich dazu, Entscheidungen in der Gegenwart von Wissen aus der Vergangenheit abhängig zu machen. Ein wichtiger Spezialfall dieses Grundsatzes ist das Merken von Zwischenschritten in Rechnungen. Wenn wir beispielsweise das aus der Schule bekannte Vorgehen zum schriftlichen Addieren von Zahlen betrachten, sind sowohl die zu Beginn übereinanderstehenden Summanden als auch eventuell zu merkende Überträge und natürlich auch das schrittweise entstehende Ergebnis in diesem Sinne Informationen (die Summanden nennen wir **Eingabe**, die entstehende Summe **Ausgabe** der Rechnung). Informationen ermöglichen an einem bestimmten Punkt der Rechnung Rückschlüsse auf die Vergangenheit und eine damit verbundene Anpassung der künftigen Aktionen. Wenn beispielsweise im vorherigen Schritt ein Übertrag entstanden ist, beeinflusst das die Berechnung der nächsten Ziffer.

Für einen Informatiker ist **das Merken von Zwischenschritten in Rechnungen** oft sogar ein und dasselbe wie der Grundsatz, **Entscheidungen in der Gegenwart von Wissen aus der Vergangenheit abhängig zu machen**. Die Essenz des Rechnens besteht für ihn darin, Zeichenketten (die Träger der Information) hintereinander aufzuschreiben und zu verändern, wobei jede Veränderung nach gewissen Regeln (diese Regeln liegen später dem Begriff des Algorithmus zugrunde) von der vorherigen Zeichenkette abhängt. So kann Wissen aus der Vergangenheit genutzt werden, um eine Entscheidung in der Gegenwart zu beeinflussen. Am Anfang einer Rechnung steht eine Zeichenkette, die (bzw. deren Interpretation nach gewissen Vereinbarungen) die Eingabe darstellt, und an ihrem Ende eine weitere Zeichenkette, die für das Rechenergebnis bzw. die Ausgabe steht. Das – und nicht mehr – ist die (intuitive) Definition einer Rechnung in der Informatik. Jeder, der sich Notizen macht, führt also fast schon eine Rechnung durch.

Ein Prozess, der ein eindeutig definiertes Verfahren, wie das eben beschriebene Additionsverfahren, auf eine Eingabe anwendet, um eine Ausgabe zu erzeugen, führt eine **Informationsverarbeitung** oder eine **Rechnung** durch. Ein solches Verfahren ist im Wesentlichen das, was als **Algorithmus** bezeichnet wird. Ein Algorithmus liefert eine (möglichst formale) Beschreibung der Regeln, nach denen Zeichenketten im Lauf einer Rechnung verändert werden müssen, um von einer Eingabe auf eine gewünschte Art der Ausgabe zu kommen. Im obigen Beispiel des schriftlichen Addierens muss der Algorithmus dafür sorgen, dass für alle Eingabe-Zeichenketten, die als **die Kodierung zweier Zahlen** interpretiert werden können, im Verlauf der Rechnung eine Ausgabe-Zeichenkette entsteht, die als die **Kodierung der Summe der beiden Zahlen** interpretiert werden kann. (Zum Begriff Kodierung vgl. Anhang A.5.)

Diese Eigenschaft von Algorithmen, eine Eingabe in eine Ausgabe zu überführen (und zwar dieselbe Eingabe üblicherweise immer in dieselbe Ausgabe) führt dazu, dass ein Algorithmus als mathematische Funktion aufgefasst werden kann: Die Eingabe-Zeichenkette ist das **Urbild**, welches auf die Ausgabe-Zeichenkette, die **Bild** bzw. **Funktionswert** darstellt, abgebildet wird.

Über „normale“ Definitionen mathematischer Funktionen hinaus hat ein Algorithmus außerdem die wichtige Eigenschaft, dass er den Weg vom Urbild zum Bild gleich mitliefert. Wir müssen ihn (bzw. die durch ihn festgelegten Regeln zur Veränderung von Zeichenketten) ja nur anwenden, und die Funktion wird berechnet. Beispielsweise könnte eine „normale“ Definition einer Funktion p, die für eine ganze Zahl genau dann *wahr* ausgibt, wenn es sich um eine **Primzahl** handelt, etwa so aussehen:

$$p\colon \mathbb{N} \to \mathbb{B}\colon p(n) =_{def} \begin{cases} wahr, & \text{falls } n > 1 \text{ und } \forall k, m \in \mathbb{N} : km = n \implies k, m \in \{1, n\} \\ falsch & \text{sonst} \end{cases}$$

Diese Definition spiegelt korrekt die Prim-Eigenschaft von Zahlen wider, sie sagt aber nichts (oder wenig) darüber aus, wie der Funktionswert **berechnet** werden kann. Ein Algorithmus für p könnte dagegen so aussehen:

```
Interpretiere die Eingabe-Zeichenkette als natürliche Zahl n.
Wenn n=0 oder n=1 gilt, gib „falsch" aus.
Wenn nicht, überprüfe für alle i von 2 bis Wurzel(n):
    Ist n durch die aktuelle Zahl i teilbar?
        Gib „falsch" aus und beende den Programmablauf.
Gib „wahr" aus.
```

Dieser Algorithmus ist nur umgangssprachlich angegeben, aber mit etwas gutem Willen versteht man ihn und kann ihn dazu nutzen, für eine gegebene Zahl den Funktionswert zu berechnen. Dies ist ebenfalls eine Definition der Funktion p, und sie ist insofern mehr wert als die vorherige, als sie gleich eine mögliche Berechnungsart für p mitliefert. Jeder Algorithmus definiert damit insbesondere nicht irgendeine Funktion, sondern eine **berechenbare Funktion**. (Nicht jeder Algorithmus hält allerdings für alle Eingaben nach endlicher Zeit an. Die Möglichkeit von Endlosschleifen muss bei der Frage nach der Berechenbarkeit von Funktionen speziell berücksichtigt werden.)

$e^{i\pi}$

An dieser Stelle zeigt sich auch der zentrale Bezug der Informatik zu einem physikalischen Rechensystem, also einem **Computer.**

Wir werden im Folgenden unter einem Computer bzw. Rechner jedes System verstehen, das eine Informationsverarbeitung durchführt. Das sind natürlich Systeme wie Laptops, Mobiltelefone usw., im weiteren Sinn können es aber auch Taschenrechner, Getränkeautomaten oder sogar ein Mensch mit Stift und Papier sein.

Die wesentliche Leistung eines Computers besteht aus genau der gerade beschriebenen Art der Informationsverarbeitung. Salopp gesagt, ist ein eingeschalteter Computer stets damit beschäftigt, Funktionen zu berechnen oder auf Eingaben zu warten – oder beides gleichzeitig.

Deshalb lassen sich essentielle Aspekte der Informatik einheitlich als Aussagen über Berechnungen formulieren. Typische Fragestellungen der theoretischen Informatik sind aus diesem Grund, obwohl sie theoretisch aussehen, von hoher praktischer Relevanz, beispielsweise:

- Gibt es für jede denkbare Funktion einen Algorithmus, mit dem sie berechnet werden kann? (Diese Frage werden wir bald mit „nein“ beantworten.)
- Welchen Einfluss hat der verwendete Rechnertyp darauf?
- Wenn eine Funktion berechnet werden kann, welche Zeit ist für die Berechnung erforderlich? Wie sieht ein Algorithmus aus, der möglichst wenig Zeit benötigt?

Welche Zeit benötigt dieser Algorithmus mindestens, im Durchschnitt und höchstens für beliebige Eingaben?
- Wenn eine Funktion berechnet werden kann, wie viel Speicherplatz wird für Zwischenergebnisse benötigt? Wie sieht ein Algorithmus aus, der möglichst wenig Speicherplatz benötigt? Wie viel Speicherplatz benötigt dieser Algorithmus mindestens, im Durchschnitt und höchstens für beliebige Eingaben?

Es existiert eine Vielzahl an Computern und sonstigen Geräten, die Daten verarbeiten können, und die Beantwortung der obigen Fragen scheint auf den ersten Blick wesentlich davon abzuhängen, auf welche dieser Geräte sie sich beziehen. Natürlich kann ein langsamerer Computer länger für eine Berechnung brauchen als ein schnellerer. Auch der Platzverbrauch (im Sinne der tatsächlichen physikalischen Flächengröße, die für die Speicherung einer bestimmten Datenmenge benötigt wird) ist auf einem älteren Rechner normalerweise größer als auf einem neueren. Ob eine Funktion überhaupt berechnet werden kann, scheint hingegen nicht eine Eigenschaft eines bestimmten Gerätes zu sein, sondern eher der Funktion an sich (oder genauer gesagt, der Funktion in Bezug auf eine ganze Klasse von Geräten oder Rechnermodellen, wie wir gleich sehen werden). Aber auch was Laufzeit und Platzverbrauch einer Berechnung angeht, stellt sich heraus, dass die genauen Werte auf einem bestimmten Computer eher selten von Interesse sind. Das liegt unter anderem daran, dass Computer heute weitgehend austauschbar sind, da ein gegebenes „Programm“ normalerweise auf jedem Computer dasselbe Ergebnis liefern soll. Deshalb sind wir von der Hardware unabhängig und können rechenintensive Prozesse auf einen schnellen Computer, etwa in eine Cloud, auslagern. Was uns interessiert, ist allgemein, ob ein Problem auch für „schwierige“ Eingaben auf irgendeinem Computer in einer vertretbaren Zeit mit einem angemessenen Platzverbrauch gelöst werden kann.

$e^{i\pi}$

Es gibt Probleme, für deren einfachste Instanzen alle heutigen Rechnern viele Milliarden Jahre zur Lösung benötigen würden. Das allein heißt aber noch nicht, dass solche Probleme unbedingt „schwierig“ sind, denn man braucht ja „nur“ einen schnelleren Rechner, den es vielleicht in zehn Jahren schon gibt. Wirklich schwierig wird ein Problem erst, wenn das **Vergrößern der Eingabe** zu einem überproportionalen Anstieg der benötigten Rechenzeit (oder des benötigten Speicherplatzes) führt. Wenn etwa ein einziges hinzugefügtes Eingabezeichen zu einer Verdoppelung der Laufzeit führt, dann haben wir ein schwieriges Problem, das wir als „praktisch nicht lösbar“ bezeichnen. In einem praktischen Sinn lösbare Probleme sollten dagegen bei einer Verdopplung der Eingabelänge nicht viel mehr als doppelt so viel Rechenzeit benötigen. Wir verzichten also auf eine genaue Zeitangabe und betrachten nur auf die **relative Vergrößerung** der Rechenzeit mit der Eingabelänge.

Die theoretische Informatik beschäftigt sich damit, möglichst allgemeine Aussagen darüber zu treffen, ob und auf welche Weise sich Funktionen berechnen lassen und wie dabei sinnvollerweise vom zugrunde liegenden Rechnermodell abstrahiert werden kann. Beispielsweise ist es hilfreich, Aussagen zu treffen, wie: „Jedes Sortierverfahren, das auf paarweisem Vergleichen der Elemente beruht, hat bei n zu sortierenden Elementen eine Laufzeit von $\Omega(n \log(n))$“ (zum Landau-Operator Ω vgl. Anhang A.4).

Viel schwächer wäre es etwa zu sagen: „Jedes Sortierverfahren, das in Java programmiert ist und auf einem Android-System mit dem und dem Prozessor läuft, hat die und die Laufzeit…“ Diese Aussage ist so speziell, dass sie nur auf wenige Situationen zutrifft und deshalb für die meisten Menschen uninteressant ist.

Um dagegen möglichst allgemeine Aussagen treffen zu können, ist es notwendig, abstrakte **Rechnermodelle** (auch **Berechnungsmodelle**) zu definieren. Ein Berechnungsmodell ist ein formales System, das die Konstruktion von „abstrakten Rechnern“ ermöglicht, welche die für reale Rechner typische Informationsverarbeitung durchführen können. Solche abstrakten Rechner heißen **Automaten** (oder auch **Maschinen**; engl. **machines**).

$e^{i\pi}$

Ein Berechnungsmodell umfasst also eine Klasse $\mathcal{A}$ von Automaten, die eine Klasse $\mathcal{R}$ realer Rechner bzw. die darauf ausführbaren Algorithmen **repräsentiert**.

Genauer werden wir letztendlich immer nur von Berechnungsmodellen ausgehen können, denn „den realen Rechner“ kann man mathematisch nicht fassen. Das „Repräsentieren“ ist also eigentlich keine Relation zwischen einem Berechnungsmodell und einer Klasse realer Rechner, sondern immer zwischen zwei Berechnungsmodellen.

Das heißt, dass für jeden Algorithmus R, der durch $\mathcal{R}$ „ausgeführt werden kann“, ein Automat $A \in \mathcal{A}$ existiert, der für alle Eingaben zu demselben Ergebnis kommt wie R.

∃

Für manche Berechnungsmodelle ist das äquivalent mit der Aussage, dass $\mathcal{A}$ in der Lage ist, alle Rechner aus $\mathcal{R}$ zu **simulieren**. Das bedeutet, dass es einen Automaten $A \in \mathcal{A}$ gibt, der als Eingabe **die Beschreibung eines zu $\mathcal{R}$ gehörenden Algorithmus R zusammen mit einer Eingabe w für R** erhält. A „simuliert“ die Berechnung von R für die Eingabe w in dem Sinne, dass die Ausgabe von A dieselbe ist, die R bei Eingabe von w ausgegeben hätte. Diese Art der Simulation funktioniert aber nur bei höheren Berechnungsmodellen (Turingmaschinen), die „mächtig“ genug sind, um so komplexe Eingaben zu verarbeiten. Wir werden die Simulation als wesentliches Werkzeug kennenlernen, um zu zeigen, dass ein Berechnungsmodell ein anderes repräsentiert.

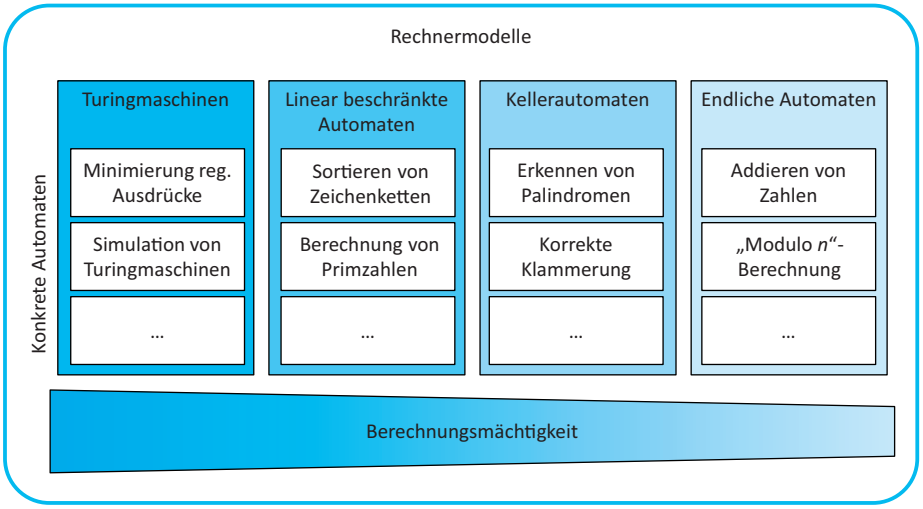


Abb. 1.1. Berechnungsmächtigkeit der vier Haupt-Automatentypen. Turingmaschinen sind die berechnungsmächtigsten Automaten; linear beschränkte Automaten können nur eine echte Teilmenge der durch Turingmaschinen berechenbaren Funktionen berechnen; Kellerautomaten wiederum eine echte Teilmenge davon; und endliche Automaten eine echte Teilmenge der letzteren. Die konkreten Beispielautomaten werden im weiteren Verlauf alle auf die eine oder andere Art vertieft.

Die formalen Systeme, welche Berechnungsmodelle beschreiben, heißen in der Informatik **Automatenmodelle** und das entsprechende Fachgebiet ist die **Automatentheorie**. Automatenmodelle werden nach ihrer **Berechnungsmächtigkeit** klassifiziert.

$e^{i\pi}$

Die Berechnungsmächtigkeit eines Automatenmodells ist durch die Menge der Funktionen gegeben, die durch Automaten dieses Typs berechnet werden können. Wir werden sehen, dass die typischen in der Informatik untersuchten Automaten eine **Berechenbarkeits-Hierarchie** bilden. Die mächtigsten Automaten können die meisten Funktionen berechnen, die der zweitmächtigsten Klasse eine echte Teilmenge davon usw.

Die mächtigste Automatenklasse (**Turingmaschinen** bzw. dazu äquivalente Modelle, s. u.) kann alle heute bekannten realen Rechnertypen repräsentieren; andere Automatenklassen (**endliche Automaten**, **Kellerautomaten**, **linear beschränkte Automaten**, s. u.) repräsentieren dagegen Rechnertypen, die in ihren Rechenfähigkeiten eingeschränkt sind. Endliche Automaten entsprechen beispielsweise in etwa der Berechnungsmächtigkeit typischer Verkaufsautomaten (vgl. auch Abbildung 1.1).

Bevor wir uns im Detail den Automaten als grundlegenden Berechnungsmodellen zuwenden, wollen wir zunächst noch einmal den „Stoff" genauer betrachten, den sie

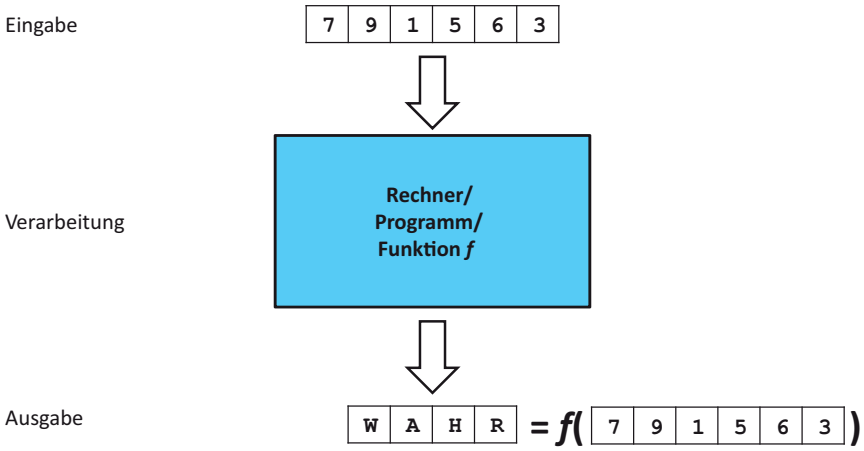


Abb. 1.2. Berechnung eines Funktionswerts durch einen Rechner. Auf eine **Eingabe** hin findet im Rechner ein Prozess der Veränderung von Zeichenketten statt, der auch als **Verarbeitung** bezeichnet wird. Zum Abschluss der Rechnung wird eine **Ausgabe** erzeugt, die als Funktionswert einer durch den Rechner definierten Funktion betrachtet werden kann.

verarbeiten: die Informationen. Wir haben den Begriff bisher so weit gefasst, dass, je nach Kontext, durchaus auch ein Händedruck oder ein Augenzwinkern als Information gelten kann. Nun werden wir formaler und beschränken uns im Folgenden auf die Nutzung von **Zeichenketten** zur Darstellung von Informationen. Eine durch einen Automaten durchgeführte Rechnung läuft dann im Allgemeinen folgendermaßen ab:

- Vor dem Start werden als Eingabe Zeichenketten auf einem Speichermedium bereitgestellt, die den **Urbildbereich** einer zu berechnenden Funktion darstellen.
- Im Verlauf der Rechnung können diese Zeichenketten nach bestimmten Anweisungen verändert oder um neue Zeichen erweitert werden.
- Nach Abschluss der Rechnung bleiben die veränderten Zeichenketten stehen, und ein Teil von ihnen wird als Ausgabe interpretiert, also als **Funktionswert** der zu berechnenden Funktion.

In Abbildung 1.2 wird schematisch dargestellt, wie ein Rechner, den wir nicht näher spezifizieren, eine Zeichenkette verarbeitet und dabei eine Funktion f berechnet. Die im Beispiel gewählte Zeichenkette „`7 9 1 5 6 3`“ ist nicht mit der im Dezimalsystem geschriebenen Zahl 791563 zu verwechseln, denn je nach Kontext könnte auch etwas anderes damit gemeint sein. Im Rechner wird diese Zeichenkette einer Serie von Veränderungen unterworfen, was der Berechnung der Funktion f entspricht. Am Ende gibt der Rechner die Zeichenkette „`W A H R`“ aus, was als Ergebnis der Funktion f aufgefasst werden kann. Alle Berechnungen (durch Rechner, Programme, Algorithmen, Automaten usw.), die wir im Folgenden betrachten werden, definieren implizit eine solche Funktion.