Juraj Hromkovič

Lehrbuch Informatik

Vorkurs Programmieren, Geschichte und Begriffsbildung, Automatenentwurf

STUDIUM

VIEWEG+ TEUBNER

Juraj Hromkovič

Lehrbuch Informatik

Juraj Hromkovič

Lehrbuch Informatik

Vorkurs Programmieren, Geschichte und Begriffsbildung, Automatenentwurf

STUDIUM

VIEWEG+
TEUBNER

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der
Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über
<http://dnb.d-nb.de> abrufbar.

Prof. Dr. Juraj Hromkovič
Geboren 1958 in Bratislava, Slowakei. Studium der Mathematischen Informatik an der Komenský Universität, Bratislava. Promotion (1986) und Habilitation (1989) in Informatik an der Komenský Universität. 1990 – 1994 Gastprofessor an der Universität Paderborn, 1994 – 1997 Professor für Parallelität an der CAU Kiel. 1997 – 2003 Professor für Algorithmen und Komplexität an der RWTH Aachen. Seit 2001 Mitglied der Slowakischen Gesellschaft. Seit Januar 2004 Professor für Informatik an der ETH Zürich.

1. Auflage 2008

Lektorat: Ulrich Sandten | Kerstin Hoffmann

Vieweg+Teubner ist Teil der Fachverlagsgruppe Springer Science+Business Media.
www.viewegteubner.de

Umschlaggestaltung: KünkelLopka Medienentwicklung, Heidelberg
Druck und buchbinderische Verarbeitung: Strauss Offsetdruck, Mörlenbach
Gedruckt auf säurefreiem und chlorfrei gebleichtem Papier.
Printed in Germany

ISBN 978-3-8348-0620-8

Vorwort

Dieses Lehrbuch ist der erste Schritt in unseren Bemühungen die große Lücke in den Unterrichtsmaterialien für Informatik an Gymnasien und für das Lehramtsstudium zu schließen. Somit ist dieses Buch das erste in einer geplanten Serie von Lehrbüchern. Unser Ziel ist nicht der Versuch, den Inhalt des Informatikunterrichts festzulegen, sondern ein viel größeres Angebot an Themen zu unterbreiten, als man tatsächlich aus Zeitgründen im Unterricht umsetzen kann. Dies entspricht auch der dynamischen Entwicklung der Informatik. Mit Ausnahme von unbestreitbaren fundamentalen Konzepten hat es keinen Sinn heute darüber zu streiten, welche Fachgebiete in Zukunft tragbarer als andere sein werden. Deswegen bieten wir den Lehrpersonen die Wahl, sich geeignete Themen und die Stufe ihrer Vertiefung abhängig von eigenen Kenntnissen, Interessen und Schwerpunkten auszusuchen. Darum strukturieren wir unsere Unterrichtsunterlagen in Form von Modulen. Jedes Modul ist einem Thema gewidmet und entspricht im Umfang 16 bis 32 Unterrichtsstunden. Die Module bestehen jeweils aus mehreren Lektionen unterschiedlicher Schwierigkeitsgrade, die die Wahl eines angestrebten Vertiefungsgrads oder einen differenzierten Unterricht ermöglichen. Die Module selbst sind so weit wie möglich von der Kenntnis anderer Module unabhängig.

Dieses erste Lehrbuch beinhaltet die folgenden drei Module: „Vorkurs Programmieren in LOGO“, „Geschichte und Begriffsbildung“ und „Entwurf von endlichen Automaten“. Das erste Modul bietet eine Einführung in das Programmieren. Mit seinen ersten sechs Lektionen ermöglicht es sogar einen Einstieg ab dem vierten Schuljahr und mit seinen letzten Lektionen ist es mit dem Mathematikunterricht in den letzten Gymnasialklassen verzahnt. Dabei geht es nicht darum, eine höhere Programmiersprache zu erlernen, sondern vielmehr darum, die fundamentalen Konzepte des systematischen Programmierens und ein tieferes Verständnis zu erwerben. Die notwendige Programmierumgebung ist kostenlos verfügbar.

In der tiefen Wurzel jeder Wissenschaftsdisziplin befindet sich die Begriffsbildung, die maßgebend für die Entwicklung einer Disziplin ist. Das Modul „Geschichte und Begriffsbildung“ verfolgt diese Idee, indem es die Geschichte der Informatik anhand der Entwicklung der Grundbegriffe und Grundkonzepte erklärt. Neben einer Erklärung für die fundamentalsten Fachbegriffe wie Algorithmus, Programm und Berechnungskomplexität werden hier auch die Grundlagen des korrekten logischen Denkens und korrekten Argumentierens, sowie das Modell der Registermaschine anhand einer einfachen Assemblersprache vermittelt.

Im dritten Modul „Entwurf von endlichen Automaten“ geht es nicht nur darum, ein paar Methoden zum Entwurf von Steuerungsmechanismen zu unterrichten. Im Vordergrund stehen hier die modulare Technik zum systematischen Entwurf von komplexen Systemen sowie die Grundlagen der Modellierung von Rechnern und Berechnungen. Die beiden letzten Module werden durch E-Learning-Systeme zur Programmierung im Assembler und zum Automatenentwurf unterstützt.

Weitere Module, die wir als Fortsetzung planen, sind: „Programmieren in Pascal/Delphi/Oberon“, „Informationssysteme“, „Datenstrukturen und effiziente Implementierung von Algorithmen“, „Kryptologie“, „Wahrscheinlichkeit und zufallsgesteuerte Systeme“, „Schaltkreisentwurf und parallele Algorithmen“, „Geometrische Algorithmen“, „Logik und korrekte Argumentation“, „Methoden zum Algorithmenentwurf“, „Kommunikation und Suche im Internet“, und „Grenzen der Automatisierung“.

Das didaktische Konzept basiert auf dem Ausführlichkeitsgrad von Leitprogrammen. Alles wird sorgfältig erklärt und sofort durch Übungsaufgaben (teilweise mit vorgeschlagenen Lösungen) gefestigt und überprüft. Durch das detaillierte und anschauliche Vorgehen eignet sich das Lehrbuch auch zum Selbststudium im Gymnasialalter. Die Erklärungen sind mit Hinweisen für die Lehrperson begleitet. Diese Hinweise gehen zum einen auf mögliche Schwierigkeiten bei der Vermittlung des Stoffes ein, sprechen zum anderen Empfehlungen für die Verwendung von geeigneten didaktischen Methoden aus und vermitteln Unterrichtserfahrung. Damit ist dieses Lehrbuch auch für das Lehramtsstudium bestimmt, sowie für alle Informatikanfänger oder diejenigen, die Informatik im Nebenfach studieren.

Hilfreiche Unterstützung anderer hat zur Entstehung dieses Lehrbuches wesentlich beigetragen. Besonderer Dank gilt Karin Freiermuth, Roman Gächter, Stephan Gerhard, Lucia Keller, Jela Sherlak, Ute Sprock, Andreas Sprock und Björn Steffen für sorgfältiges Korrekturlesen, zahlreiche Verbesserungsvorschläge und umfangreiche Hilfe bei der Umsetzung des Skriptes in Latex. Für die Sprachkorrekturen bedanke ich mich herzlich bei Frau Regina Lauterschläger. Ich möchte mich sehr bei Karin Freiermuth, Barbara Keller und Björn Steffen dafür bedanken, dass sie mich mit viel Enthusiasmus beim Testen der Unterrichtsunterlagen in der schulischen Praxis begleitet haben oder einige Lektionen selbstständig unterrichtet haben.

Genauso herzlich danke ich den Lehrpersonen Pater Paul (Hermann-Josef-Kolleg, Steinfeld), Uwe Bettscheider (INDA Gymnasium Aachen), Hansruedi Müller (Schweizerische Alpine Mittelschule Davos), Yves Gärtner, Ueli Marty (Kantonsschule Reussbühl), Meike

Akveld, Stefan Meier und Pietro Gilardi (Mathematisch-naturwissenschaftliches Gymnasium Rämibühl, Zürich), Harald Pierhöfer (Kantonsschule Limattal, Urdorf) und Michael Weiss (Gymnasium Münchenstein), die es uns ermöglicht haben, in einigen Klassen kürzere oder längere Unterrichtssequenzen zu testen oder sie sogar selbst getestet haben und uns ihre Erfahrungen mitgeteilt haben. Ein besonderer Dank geht auch an die Schulleitungen der Schulen, die uns für das Testen unserer Module die Türen geöffnet haben. Für die hervorragende Zusammenarbeit und die Geduld mit einem immer eigensinniger werdenden Professor bedanke ich mich herzlich bei Frau Kerstin Hoffmann und Herr Ulrich Sandten vom Teubner Verlag.

Ich wünsche allen Leserinnen und Lesern beim Lernen mit diesem Buch so viel Vergnügen, wie wir selbst beim Unterrichten der vorliegenden Lektionen empfunden haben.

Zürich, im September2008

Juraj Hromkovič

Inhaltsverzeichnis

Modul I

Vorkurs Programmieren in LOGO

Einleitung

Programmieren in LOGO

Warum unterrichten wir Programmieren?

Programmieren gehört zum Handwerkszeug eines jeden Informatikers, auch wenn das Programmieren alleine noch keinen Informatiker ausmacht. Vor ungefähr zwanzig Jahren war Programmieren in Ländern mit Informatikunterricht an den Mittelschulen ein hauptsächlicher Bestandteil des Curriculums. Dann folgte eines der unglücklichsten Eigentore, die die Informatiker sich je geschossen haben. Einige von ihnen haben begonnen, die Wichtigkeit der Informatik und deren Unterricht damit zu begründen, dass nun fast jeder einen Rechner habe oder bald haben werde. Man müsse deswegen mittels Informatikunterricht den kompetenten Umgang mit dem Rechner inklusive aktueller Software lehren. Dies ist eine ähnlich gelungene Begründung wie die eines Maschinenbauers, welcher den Unterricht seines Faches an Gymnasien damit rechtfertigen würde, dass fast jedermann ein Fahrzeug besitze und man deswegen allen die Chance geben müsse, damit kompetent umgehen zu lernen.

Die unmittelbare Folge dieser *„Propaganda“* in einigen Ländern war der Ausbau oder Umbau des bestehenden Informatikunterrichts zu einer billigen Ausbildung zum Computerführerschein. Die Vermittlung von Wissen und informatischer Grundfertigkeit wurde durch das Erlernen des Umgangs mit kurzlebigen und größtenteils mangelhaften Softwaresystemen ersetzt. Es dauerte dann auch nur wenige Jahre, bis die Bildungspolitiker und Schulen erkannt haben, dass eine solche Informatik weder Substanz noch Nachhaltigkeit besitzt, und man für einen Computerführerschein kein eigenständiges Fach Informatik braucht. Und so hat man dann umgehend *„das Kind mit dem Bade ausgeschüttet“*. Wir haben heute in den meisten Gebieten des deutschsprachigen Raumes keinen eigentlichen Informatikunterricht in den Mittelschulen mehr.

J. Hromkovič, *Lehrbuch Informatik*, DOI 10.1007/978-3-8348-9692-6_1,

Zwar wurde das Problem inzwischen erkannt, doch der Informatikunterricht wird an den Folgen der billigen *„Informatik-Propaganda“* noch viele Jahre zu leiden haben, nicht nur weil es schwierig ist, in kurzer Zeit und ohne entsprechend ausgebildete Lehrpersonen von einem schlecht eingeführten Unterricht zu einem qualitativ hochwertigen Unterricht zu wechseln, sondern auch, weil die Informatik in der Öffentlichkeit ein falsches Image erhalten hat. Informatiker sind diejenigen, die mit dem Rechner gut umgehen können, d. h. alle Tricks kennen, um jemandem bei den allgegenwärtigen Problemen mit unzulänglicher Software zu helfen. Wir sollten dieses Bild mit der Wertigkeit von Fächern wie Mathematik, Physik und anderen Gymnasialfächern vergleichen. Auch wenn diese Fächer nicht bei jedermann beliebt sind, wird ihnen niemand die Substanz absprechen wollen.
Im Gegensatz dazu hören wir von guten Gymnasialschülerinnen und -schülern oft, dass ihnen die Informatik zwar Spaß macht, dass sie zum Studieren aber „zu leicht“ sei: „Das kann man sich nebenbei aneignen.“
Sie wollen ein Fach studieren, welches eine echte Herausforderung darstellt. In dem, was sie bisher im sogenannten Informatikunterricht gesehen haben, sehen sie keine Tiefe oder Substanz, für deren Beherrschung man sich begeistern lassen kann.

Andererseits wissen wir, dass sich die Informatik inzwischen gewaltig entwickelt hat, so dass dank ihr in vielen Gebieten der Grundlagenforschung sowie der angewandten technischen Disziplinen wesentliche Fortschritte erzielt werden konnten. Sowohl die Anzahl der Anwendungen als auch die der Forschungsrichtungen der Informatik ist in den letzten Jahren so stark gewachsen, dass es sehr schwierig geworden ist, ein einheitliches, klares Bild der Informatik zu vermitteln, ein Bild einer Disziplin, die in sich selbst die mathematisch-naturwissenschaftliche Denkweise mit der konstruktiven Arbeitsweise eines Ingenieurs der technischen Wissenschaften verbindet. Die Verbindung dieser unterschiedlichen Denkweisen und Wissenschaftssprachen in einem einzigen Fach ist aber gerade die Stärke des Informatikstudiums.

Die Hauptfrage ist nun, wie man diese vielen, sich dynamisch entwickelnden Informatikgebiete, -themen und -aspekte in ein Curriculum fürs Gymnasium abbilden kann. Da divergieren die Meinungen und Präferenzen der Informatiker so stark, dass man nur sehr schwer einen Konsens erreichen kann.

Warum sehen wir in dieser, von allerlei widersprüchlichen Meinungen geprägten Situation das Programmieren als einen unbestrittenen und zentralen Teil der Informatikausbildung an? Dafür gibt es mehrere Gründe: Auch andere Gymnasialfächer stehen vor keiner einfachen Wahl. Und wir können einiges von ihnen lernen, zum Beispiel, dass wir die

historische Entwicklung verfolgen sollten, anstatt uns ausschließlich auf die Vermittlung der neuesten Entwicklungen zu konzentrieren. In Fächern wie Mathematik oder Physik ist der Versuch, um jeden Preis neueste Entdeckungen zu vermitteln, didaktisch geradezu selbstvernichtend.

Wenn wir als Informatiker unserer eigenen Disziplin also eine ähnliche Tiefe zuschreiben, dürfen wir uns nicht auf diesen Irrweg einlassen. Wir müssen bodenständig bleiben und wie in anderen Fächern mit der Begriffsbildung und Grundkonzepten beginnen. Die historisch wichtigsten Begriffe, welche die Informatik zur selbständigen Disziplin gemacht haben, sind die Begriffe **„Algorithmus“** und **„Programm“**. Und wo könnte man die Bedeutung dieser Begriffe besser vermitteln als beim Programmieren? Dabei verstehen wir das Programmieren nicht nur als eine für den Rechner verständliche Umsetzung bekannter Methoden zur Lösung gegebener Probleme, sondern vielmehr als die Suche nach konstruktiven Lösungswegen zu einer gegebenen Aufgabenstellung. Wir fördern dabei einerseits die Entwicklung des algorithmischen, lösungsorientierten Denkens und stehen damit in Beziehung zum Unterricht der Mathematik, während wir andererseits mit dem Rechner zu „kommunizieren“ lernen.

Programme zu schreiben bedeutet eine einfache und sehr systematisch aufgebaute Sprache, genannt Programmiersprache, zu verwenden. Die Besonderheit der Programmiersprachen ist die Notwendigkeit, sich korrekt, exakt und eindeutig auszudrücken, weil der „Dialogpartner“ unfähig ist zu improvisieren. Wenn absolute Präzision und Klarheit in der Formulierung der Anweisungen unabdingbare Voraussetzung für die unmissverständliche Erklärung der Lösungswege sind, so dass sie sogar eine Maschine ohne Intellekt verstehen und umsetzen kann, fördert dies die Entwicklung der Kommunikationsfähigkeit enorm.

Ein weiterer Grund für die zentrale Bedeutung des Programmierunterrichts liegt in der Verbindung zwischen der logisch-mathematischen Denkweise und der konstruktiven Denkweise der Entwickler in den technischen Wissenschaften. Die Problemspezifikation, die Suche nach einem Lösungsweg sowie die formale Ausdrucksweise zur Beschreibung einer gefundenen Lösungsmethode sind stark mit der Nutzung der Mathematik sowohl als Sprache als auch als Methode verbunden. Ein wesentlicher Lerneffekt bei der Entwicklung komplexer Programme ist die „modulare“ Vorgehensweise. Die Modularität ist typisch für Ingenieurwissenschaften. Zuerst baut man einfache Systeme für einfache Aufgabenstellungen, deren korrekte Funktionalität leicht zu überprüfen ist. Diese einfachen Systeme („Module“) verwendet man als (Grund-) Bausteine zum Bau von komplexeren Systemen. Diese komplexen Systeme kann man selbst wieder als Module (Bausteine) verwenden, aus denen man noch komplexere Systeme bauen kann, usw.

Programmieren ist also ein ideales Instrument zum systematischen Unterricht in der modularen Vorgehensweise beim Entwurf komplexer Systeme aller Arten.

Schließlich bietet Programmieren einen sinnvollen Einstieg in die Welt der weiteren grundlegenden Begriffe der Informatik, wie Verifikation, Berechnungskomplexität (Rechenaufwand) und Determiniertheit. Wenn man lernt, wie Programme nach unterschiedlichen Kriterien wie Effizienz, Länge, Verständlichkeit, modulare Struktur, Benutzerfreundlichkeit oder „Kompatibilität“ beurteilt werden können, versteht man auch, dass kein bestehendes System vollkommen ist. Dies führt zur Fähigkeit, Produkte kritisch zu durchleuchten und zu beurteilen sowie über Verbesserungen nach ausgesuchten Kriterien nachzudenken.

Zusammenfassend trägt der Programmierunterricht auf vielen unterschiedlichen Ebenen zur Wissensvermittlung und Bildung bei. Neben der Fertigkeit, in bestimmten Programmiersprachen zu programmieren, erwerben die Schüler in Projekten die Fähigkeit, die Denkweisen der Theorie und der Praxis miteinander zu verbinden und systematisch, konstruktiv und interdisziplinär zu arbeiten. Indem sie den ganzen Weg von der Idee bis zum fertigen Produkt selbst miterleben, entwickeln sie eine fundierte Haltung zu Entwicklungsprozessen im Allgemeinen. Die Verbindung neuer Ideen mit der selbständigen Überprüfung im Hinblick auf die Umsetzbarkeit bereichert die Schule noch auf eine andere Art und Weise: Es gibt kein anderes Themengebiet der Informatik, dessen Lernprozess derart viele grundlegende Konzepte integriert.

Unser Programmiervorkurs in LOGO, oder Programmieren in 10 Minuten

Beim Programmierunterricht steht das Erlernen der Programmierkonzepte im Vordergrund und das Meistern einer konkreten höheren Programmiersprache muss als zweitrangig gesehen werden. Hier empfehlen wir deswegen, mit LOGO anzufangen und dann zum geeigneten Zeitpunkt zu einer höheren pascalartigen Sprache zu wechseln. Die Zeit, die man am Anfang mit LOGO verbringt, wird später im Unterricht einer höheren Programmiersprache mehr als zurückgezahlt.

Warum gerade LOGO für den Einstieg in die Programmierung? LOGO ist eine Programmiersprache, die aus rein fachdidaktischen Gründen für den Unterricht der Programmierung entwickelt wurde. Aus didaktischer Sicht ist sie genial und uns ist nach vielen Jahren ihrer Existenz noch keine annähernd vergleichbare Konkurrenz bekannt. Sie reduziert die Syntax so stark, dass man mit ihr praktisch nicht belastet ist. Somit können Anfänger schon nach zehn Minuten ihre ersten eigenen Programme schreiben und im Test laufen lassen. Programmierer können Programme zuerst Befehl um Befehl schreiben und einzeln anschauen, ob die jeweiligen Befehle das Gewünschte verursachen. Sie können aber auch zuerst komplette Programme oder Programmteile schreiben und sie dann während eines Durchlaufs überprüfen. Die Möglichkeit, sich am Anfang auf die Entwicklung von Programmen zur Zeichnung von Bildern zu konzentrieren, macht den Lernprozess sehr anschaulich, motivierend und dadurch attraktiver.

Eine hohe Interaktion und gegenseitige Befruchtung mit dem Geometrieunterricht ist leicht zu erreichen. Aus Erfahrung lassen sich alle Altersgruppen für das Programmieren in LOGO von Anfang an leicht begeistern. Das Programmieren ist lösungsorientiert und prägt die Entwicklung des algorithmischen Denkens.

Der Kern des didaktischen Vorgehens ist es, auf gut zugängliche Weise die folgenden Grundkonzepte des Programmierens zu vermitteln:

- Ein Programm als Folge von einfachen Grundbefehlen aus einer kurzen Befehlsliste
- Unterprogramme als Module zum Bau von komplexen Programmen
- Schleifen als Teil des strukturierten Programmierens
- Variablen als Grundkonzept des Programmierens

- Lokale und globale Variablen und Übertragung von Daten zwischen Programmen
- Verzweigung von Programmen und bedingten Befehlen
- Rekursion

Dabei kann man den Unterricht so gestalten, dass man mit nur fünf bis sechs Befehlen startet. Alles, was man braucht, wird aus diesen wenigen Befehlen zusammengesetzt. Damit verstehen die Schülerinnen und Schüler sofort, dass viele Befehle nur Namen für ganze Programme sind, die selbst aus einfachen Befehlen bestehen. Sie lernen dabei, nach Bedarf eigene neue Befehle zu programmieren, zu benennen und dann einzusetzen. Wegen der Syntax und der großen Vielfalt an komplexen Befehlen bei höheren Programmiersprachen nimmt die Vermittlung dieser Konzepte einen unvergleichbar höheren Aufwand in Anspruch als bei LOGO.

Die Programmierumgebung in LOGO

Für die Herstellung dieses Moduls haben wir die kommerzielle Programmiersprache SUPERLOGO und die freie Software XLOGO verwendet. Beide zeichnen sich dadurch aus, dass ihre Umgebungen so einfach aufgebaut sind, dass man praktisch sofort mit dem Programmieren beginnen kann.

Der Bildschirm von XLOGO sieht vereinfacht aus wie in Abb. 0.1 dargestellt. In das `Befehlsfenster` schreibt man die Befehle (Rechnerinstruktionen) für die Schildkröte. Im `Befehlsfenster` kann man einen Befehl oder eine Folge von Befehlen, also sogar ein vollständiges Programm, schreiben. Wenn man die Return-Taste drückt, wird der Rechner die ganze Folge der Befehle aus dem `Befehlsfenster` ausführen. Die Umsetzung der Befehle können wir direkt an den Bewegungen der Schildkröte im `Schildkrötenfenster` beobachten. Man kann den Inhalt des `Befehlsfensters` als eine Programmzeile betrachten. Nach ihrer Ausführung wird die komplette Zeile nach unten in das `Fenster der letzten Befehle` übertragen. Damit behalten wir die Übersicht über den bisher geschriebenen Teil unseres Programms. Die Schaltfläche mit der Beschriftung „STOP“ dient zum Anhalten der Programmausführung. Das ist sinnvoll wenn das Programm zu lange, oder sogar unendlich lange läuft. Für die Ziele der ersten beiden Lektionen reicht dieses Wissen über die Programmierumgebung aus.

In Lektion 3 lernen wir, Programme zu benennen und abzuspeichern. Wenn man in

XLOGO Programme mit Namen bezeichnen will, muss man auf die Schaltfläche „Editor“ klicken. Dann erscheint ein kleines Fenster wie vereinfacht in Abb. 0.2 dargestellt. In diesem `Editor` sehen wir alle bisher geschriebenen Programme. Hier können wir neue Programme speichern oder die alten Programme editieren (verändern). Das Fenster können wir mit einem Mausklick auf die Schaltfläche `Pinguin` schließen.

Wenn man alle geschriebenen Programme unter einem gewählten Projektnamen abspeichern will (um sie in der nächsten Stunde wieder verwenden zu können), dann klickt man auf das Menü `File` links oben (Abb. 0.1). Daraufhin erscheint ein Menü wie in Abb. 0.3 (links), bei dem man „Save As . . . “ auswählen muss. Danach erscheint das `Projektfenster`, wie es in Abb. 0.3 (rechts) gezeigt ist. Wir wählen ein Verzeichnis für die Abspeicherung von LOGO-Projekten und suchen uns einen Namen für das zu speichernde Projekt, den wir in das Textfeld `File name` eintippen. Mit dem Klick auf „Save“ werden dann die bisher geschriebenen Programme unter dem angegebenen Projektnamen

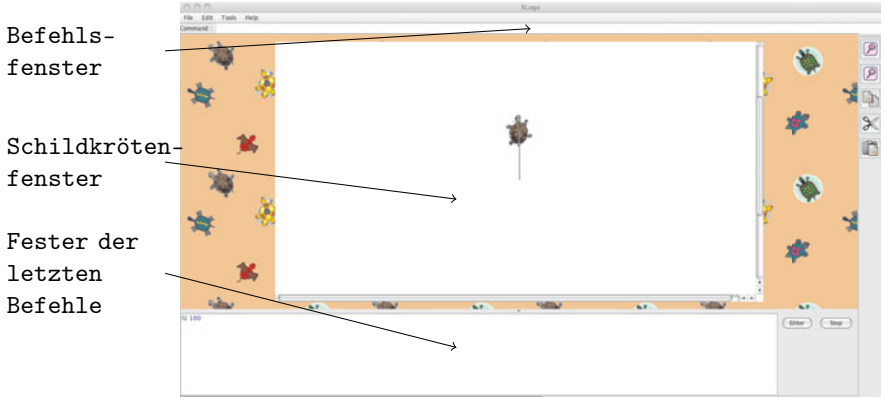


Abbildung 0.1 Der Bildschirm von XLOGO mit dem `Schildkrötenfenster` und dem `Befehlsfenster`.

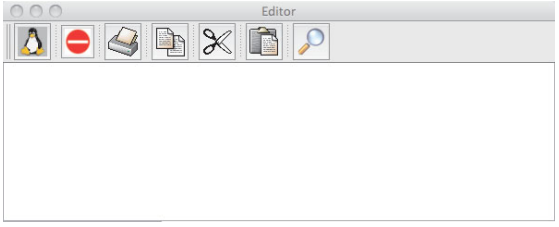


Abbildung 0.2 Der `Editor` von XLOGO

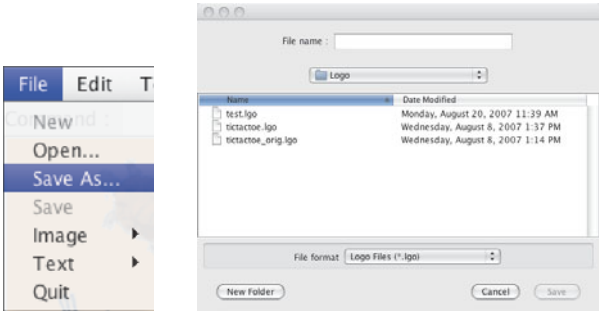


Abbildung 0.3 Links ist das Menü `File` abgebildet. Wenn man auf „Save As . . . “ klickt, erhält man das `Projektfenster` rechts.

gespeichert.

Die Programmierumgebung SUPERLOGO ist noch etwas benutzerfreundlicher und ermöglicht uns, viele Aktivitäten direkt auf dem Basisbildschirm umzusetzen. Der Bildschirm sieht zu Beginn wie in Abb. 0.4 aus.

Das `Programmierfenster` in SUPERLOGO erfüllt die Funktionen vom `Befehlsfenster` und vom `Fenster der letzten Befehle` in XLOGO. Die letzte geschrie-

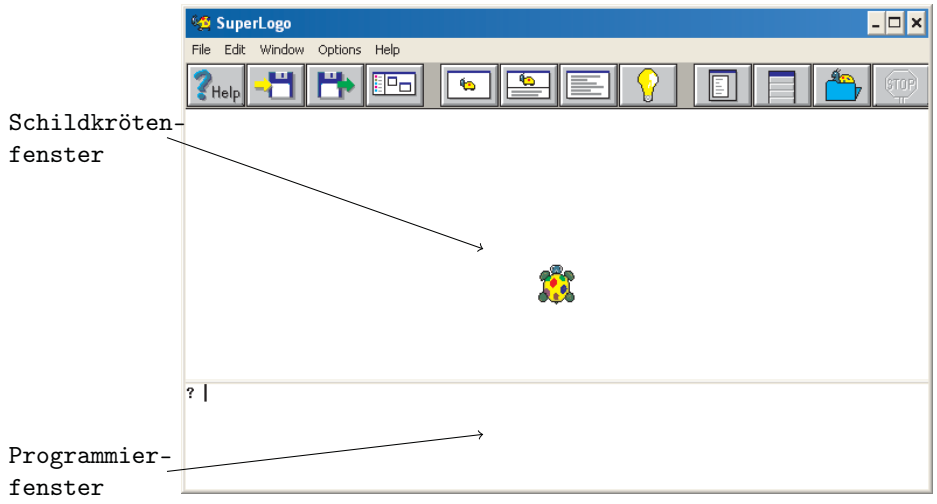


Abbildung 0.4 Die Programmierumgebung SUPERLOGO mit dem `Programmierfenster` und dem `Schildkrötenfenster`.

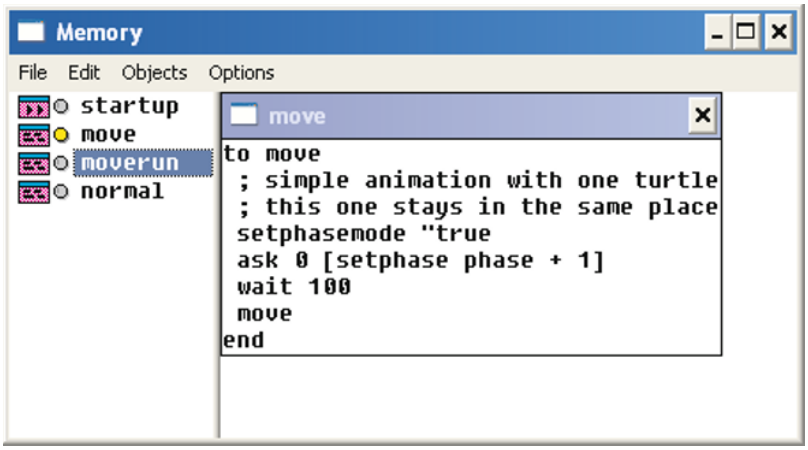


Abbildung 0.5

bene Zeile ist die aktuelle und nach dem Drücken von „Return“ werden die dort geschriebenen Befehle ausgeführt. Das `Schildkrötenfenster` funktioniert genau so wie bei XLOGO. Die horizontale Linie zwischen dem `Schildkrötenfenster` und dem `Programmierfenster` kann man beliebig in der Vertikalen verschieben und so die Proportionen zwischen diesen beiden Fenstern beliebig variieren.

Die Benennung von Programmen werden direkt im `Programmierfenster` durch den Befehl `to` vorgenommen, wie in Lektion 3 beschrieben. Wenn man bereits geschriebene Programme anschauen oder verändern will, muss man auf die vierte Schaltfläche von links in der oberen Zeile des Bildschirms klicken. Danach erscheint das Fenster aus Abb. 0.5. Im linken Teil dieses Fensters sind die Namen aller gespeicherter Programme aufgelistet. Durch einen Doppelklick auf einen Namen erscheint das gewünschte Programm im rechten Fenster. Man kann sich beliebig viele Programme gleichzeitig anschauen. Durch einen Doppelklick auf ein Programm aus dem rechten Teil des Fensters öffnet sich ein neues Fenster, in dem man das gewählte Programm editieren (verändern) kann.

Wenn man die bisher geschriebenen Programme unter einem Projektnamen speichern will, dann klickt man auf die zweite Schaltfläche von links in der obersten Zeile. Danach erscheint das `Savefenster` aus Abb. 0.6. Jetzt können wir auf den Namen eines schon definierten Projekts in dem größeren Teilfenster links klicken. Dadurch erscheint dieser Name im Textfeld `Projektname`. Mit einem Klick auf „OK“ wird das Projekt unter diesem Namen gespeichert. Wir können aber auch einen völlig neuen Namen für das Projekt wählen, indem wir den Namen direkt in das Textfeld `Projektname` eintippen.

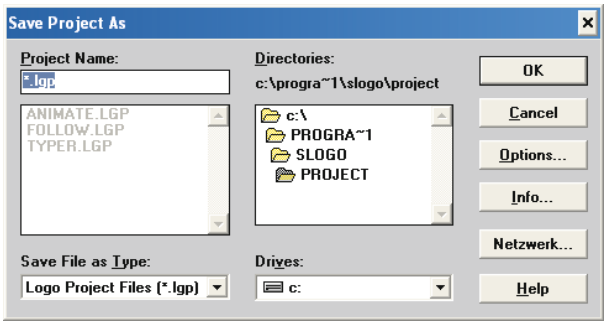


Abbildung 0.6

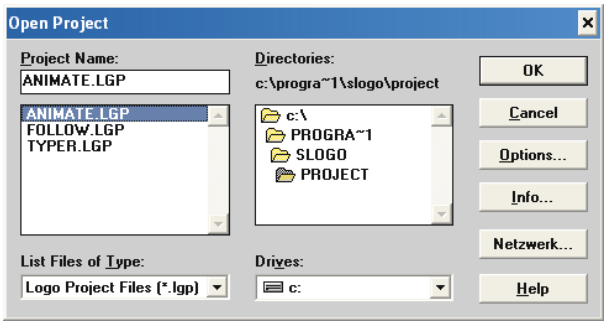


Abbildung 0.7

Wenn wir von Anfang an wissen, in welchem Projekt wir arbeiten wollen, und wissen, dass wir unsere neuen Programme zusätzlich zu den alten in dasselbe Projekt speichern wollen, müssen wir eben dieses Projekt zu Beginn unserer Arbeit laden. Dazu klicken wir auf die dritte Schaltfläche von links in der obersten Zeile und erhalten ein ähnliches Fenster wie in Abb. 0.7. Danach können wir aus der Liste der bisherigen Projektnamen ein Projekt auswählen und dadurch den Projektnamen in das Textfeld `Projektname` holen. Mit einem Klick auf „OK“ bestätigen wir unsere Wahl.

Der häufigste Fehler passiert, wenn man am Anfang der Arbeit kein Projekt geladen hat und am Ende unter einem schon existierenden Projektnamen die neuen Programme abspeichert. Bei diesem Vorgehen werden zwar die neu geschriebenen Programme gespeichert, jedoch werden alle alten Programme, die zuvor in diesem Projekt gespeichert waren, gelöscht.

Lektion 1

Programme als Folge von Befehlen

Programme sind nichts als Folgen von Rechnerbefehlen. Ein **Rechnerbefehl** ist eine Anweisung, die der Rechner versteht und ausüben kann. Der Rechner kennt eigentlich nur sehr wenige Befehle und alle komplizierten Tätigkeiten, die wir vom Rechner vollbracht haben wollen, müssen wir aus den einfachen Rechenbefehlen zusammensetzen. Das ist nicht immer einfach. Es gibt Programme, die aus Millionen Befehlen zusammengesetzt sind. Hierbei die Übersicht nicht zu verlieren, erfordert ein professionelles und systematisches Vorgehen, das wir in diesem Programmierkurs erlernen werden.

Das Ziel der ersten Lektion ist, einige grundlegende Befehle der Programmiersprache LOGO kennenzulernen. Eine **Programmiersprache** ist eine Sprache für Rechnerbefehle, in der wir unsere Wünsche und Anweisungen in Form von Befehlen dem Rechner mitteilen. Genau wie bei der natürlichen Sprache, sind die kleinsten Bausteine einer Programmiersprache die Wörter, die hier den einzelnen Befehlen entsprechen. Die Grundbefehle, die wir hier lernen, sind zum Zeichnen von geometrischen Bildern bestimmt. Auf dem Bildschirm befindet sich eine Schildkröte, die sich unseren Befehlen folgend bewegen kann. Wenn sie sich bewegt, funktioniert sie genau wie ein Stift. Sie zeichnet immer genau die Strecken, die sie gegangen ist. Also navigieren wir die Schildkröte so, dass dadurch die gewünschten Bilder entstehen.

Die ersten Befehle bewegen die Schildkröte nach vorne oder nach hinten. Damit zeichnet sie also gerade Linien. Der erste Befehl

```
forward 100
```

besagt, dass sich die Schildkröte 100 Schritte nach vorne bewegen soll. Damit ist `forward`

J. Hromkovič, *Lehrbuch Informatik*, DOI 10.1007/978-3-8348-9692-6_2,

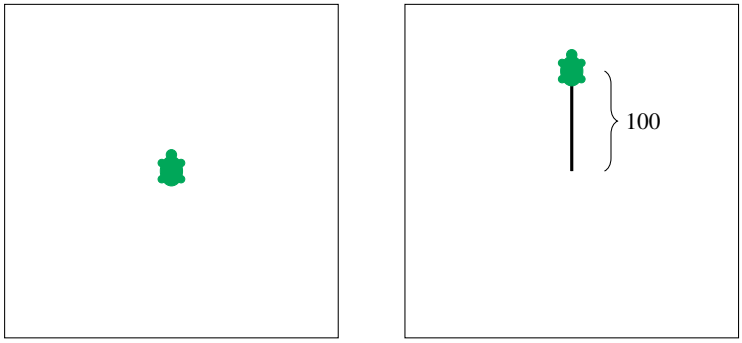


Abbildung 1.1 Ausführung des Befehls `forward 100`

das **Schlüsselwort** oder **Befehlswort** und somit der eigentliche Befehl, der die Schildkröte auffordert, nach vorne zu gehen, also in die Richtung, in die sie schaut (wo sich der Kopf befindet). Die Zahl `100` nennen wir **Parameter**, der bestimmt, wie weit die Schildkröte nach vorne gehen soll. Weil die Schildkröte sehr kleine Schritte macht, empfiehlt man, sie mindestens 20 Schritte nach vorne gehen zu lassen, um das Resultat zu sehen. Die Auswirkung des Befehls sieht man in der Abb. 1.1. Links ist die Situation vor der Ausführung des Befehls `forward 100` und rechts die Situation nach der Ausführung erkennbar.

Aufgabe 1.1 Schreibe nacheinander die Befehle

```
forward 100
forward 50
forward 20
```

und beobachte, was passiert.

Weil die Programmierer in einem gewissen gesunden Sinne faul sind und ungern zu viel schreiben oder tippen, haben sie auch eine kürzere Version des Befehls `forward` eingeführt. Die kürzere Darstellung heißt `fd`. Also bedeutet

```
fd 100
```

genau dasselbe wie

```
forward 100
```

Aufgabe 1.2 Schreibe `fd 50`, um die Funktionalität dieses Befehls zu überprüfen.

Ein sehr nützlicher Befehl ist

```
cs.
```

Wenn man diesen Befehl verwendet, wird alles bisher Gezeichnete auf dem Bildschirm gelöscht und die Schildkröte kehrt in ihre ursprüngliche Startposition in der Mitte des Fensters zurück und schaut nach oben. Der Befehl `cs` hat keinen Parameter und besteht also nur aus dem Befehlswort.

Aufgabe 1.3 Schreibe den Befehl `cs`, um es auszuprobieren.

Die Schildkröte kann auch zurückgehen. Dies geschieht durch den Befehl

```
backward 100
```

oder die gleichwertige, kürzere Beschreibung

```
bk 100.
```

Wieder sind `backward` und `bk` die Namen des Befehls (Befehlsworte) und `100` ist der Parameter, der besagt, wie viele Schritte (wie weit) man nach hinten gehen soll. Die Auswirkung des Befehls `bk 100` siehst du in Abb. 1.2 auf der nächsten Seite.

Aufgabe 1.4 Schreibe das folgende Programm und beobachte die Ausführung.

```
fd 100
bk 100
bk 200
fd 200
```

Aufgabe 1.5 Könntest du die folgenden Programme verkürzen und die entstehenden Bilder durch nur einen Befehl zeichnen lassen?

a)
```
fd 15
fd 30
fd 45
```

b)
```
fd 100
 bk 50
 fd 70
```

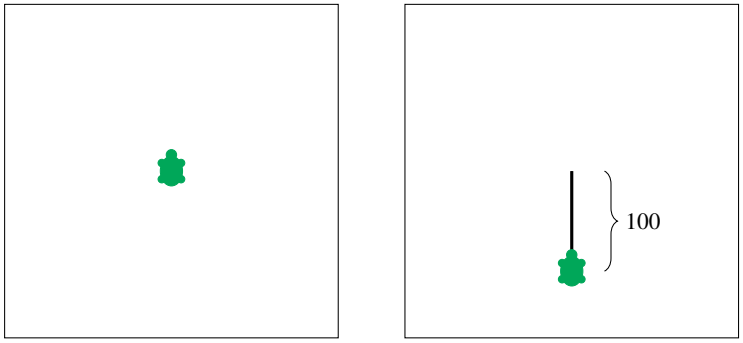


Abbildung 1.2 Ausführung des Befehls `backward 100`. Links die Lage vor der Ausführung, rechts die Lage danach.

Wie wir schon wissen, sind Programme Folgen von Befehlen. In das Befehlsfenster können sie auf unterschiedliche Weise eingegeben werden. Wenn man sie zeilenweise einen Befehl pro Zeile schreibt, dann werden sie immer einzeln durchgeführt und wir können ihre Wirkung und somit die Wirkung des ganzen Programms schrittweise beobachten. Wir können aber auch ein ganzes Programm wie

```
fd 100 bk 50 fd 70
```

in eine Zeile schreiben. In diesem Fall wird das Programm ohne Unterbrechung zwischen den Befehlen auf einmal ausgeführt.

Hinweis für die Lehrperson Am Anfang ist es besser, die Programme zeilenweise zu schreiben. Dies ermöglicht eine einfachere Entdeckung eines Programmierfehlers. Wenn das ganze Programm in einer Zeile ausgeführt wird und nicht das Gewünschte tut, ist die Suche nach möglichen Fehlern oft mühsam.

Es wäre aber langweilig, wenn die Schildkröte nur nach oben oder unten laufen dürfte. Sie darf sich auf der Stelle nach rechts oder nach links drehen und zwar in einem beliebigen Winkel.

Hinweis für die Lehrperson Für Kinder in der dritten bis sechsten Klasse empfiehlt es sich, sich in den ersten Lektionen auf die Nutzung der Winkel von 90°, 180° und 270° zu beschränken. Am Anfang reicht es sogar, nur einen 90° Winkel zu verwenden.

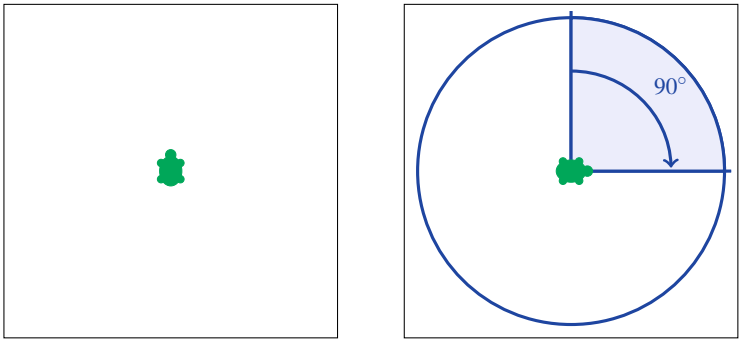


Abbildung 1.3 Der Befehl `right 90` oder `rt 90` fordert die Schildkröte auf, sich um 90° nach rechts zu drehen (90° bedeutet einen Viertelkreis).

Um nach rechts zu drehen, verwenden wir den Befehl:

`right 90` oder seine kürzere Form `rt 90`.

Der Befehlsname ist hier `right` oder `rt` und `90` ist der Parameter, der den Winkel bestimmt, um den sie sich drehen soll. Vorsicht, die Schildkröte dreht sich nach rechts aus ihrer eigenen Perspektive, nicht aus deiner. Wenn sie zum Beispiel nach unten schaut, ist ihre rechte Seite eigentlich deine linke.

Die folgenden Bilder zeigen die Auswirkungen der unterschiedlichen Drehbefehle. Dabei ist links immer die Situation vor der Ausführung und rechts die Situation nach der Ausführung dargestellt. Die Bilder in Abb. 1.3 und in Abb. 1.4 auf der nächsten Seite zeigen die Wirkung des Befehls `rt 90`, abhängig von der Richtung, in die die Schildkröte vor der Ausführung des Befehls `rt 90` schaut.

Die Bilder in Abb 1.5 auf der nächsten Seite zeigen die Ausführung des Befehls `rt 180` und die Bilder in Abb. 1.6 auf der nächsten Seite die Ausführung des Befehls `rt 270`. Die Bilder in Abb. 1.7 auf Seite 25 zeigen, dass eine Drehung um 360° weder Position noch Richtung der Schildkröte verändert.

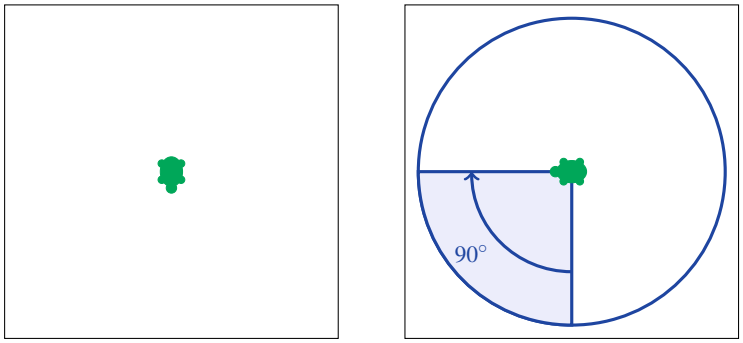


Abbildung 1.4 Der Befehl `rt 90` fordert die Schildkröte auf, sich um 90° (einen Viertelkreis) nach rechts zu drehen.

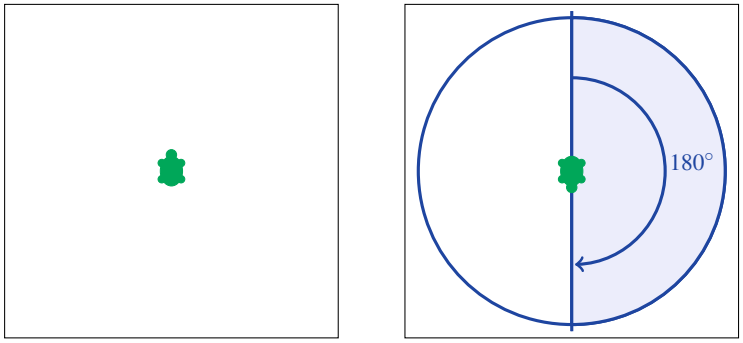


Abbildung 1.5 Der Befehl `rt 180` fordert die Schildkröte auf, sich um 180° nach rechts zu drehen (180° entspricht einem Halbkreis und damit der Drehung in die Gegenrichtung).

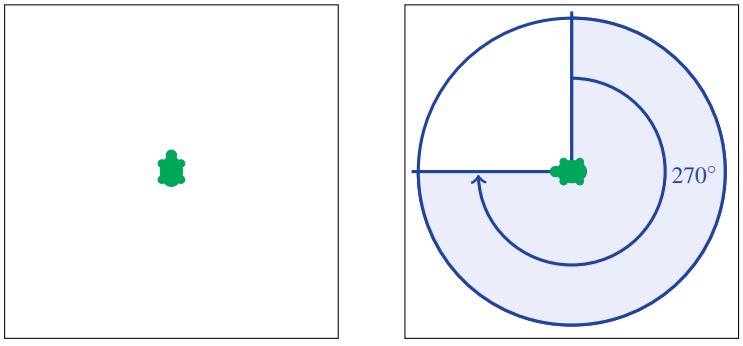


Abbildung 1.6 Der Befehl `rt 270` fordert die Schildkröte auf, sich um 270° nach rechts zu drehen.

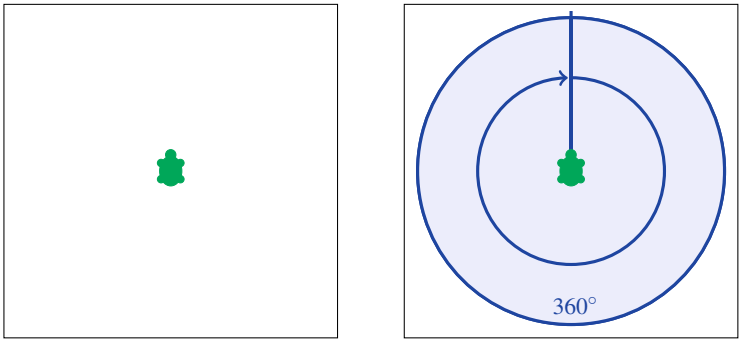


Abbildung 1.7 Der Befehl `rt 360` fordert die Schildkröte auf, sich um 360° nach rechts zu drehen. Die Schildkröte dreht sich einmal im Kreis und schaut in die gleiche Richtung wie vorher, es hat sich also nichts geändert.

Aufgabe 1.6 Überlege, was das folgende Programm bewirkt. Überprüfe deine Überlegung, indem du das Programm ausführen lässt.

```
rt 360
rt 180
rt 90
rt 180
```

Kannst du die Wirkung dieses Programms mit einem Befehl `rt X` für eine geeignete Zahl `X` erreichen?

Wir können eigentlich jede beliebige Richtungsänderung nur durch das Drehen nach rechts bewirken. Manchmal ist es jedoch für uns beim Programmieren angenehmer, auch nach links drehen zu dürfen. Dafür haben wir den Befehl

`left 90` oder `lt 90`.

Die Wirkung dieses Befehls (s. Abb. 1.8 auf der nächsten Seite) ist die Drehung um den angegebenen Winkel nach links. Beachte, dass `lt 90` und `rt 270` die gleiche Wirkung haben.

Aufgabe 1.7 Zeichne für die Befehle `lt 180` und `lt 270` ihre Wirkung auf ähnliche Weise, wie wir es auf den bisherigen Bildern für die anderen Drehbefehle gemacht haben.

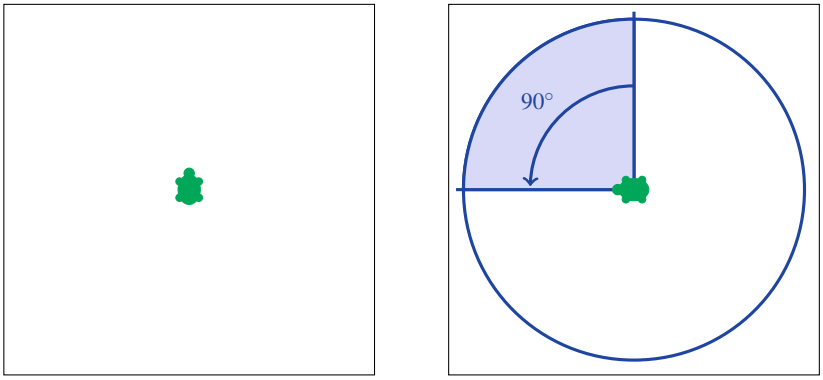


Abbildung 1.8 `left 90` oder `lt 90` dreht die Schildkröte nach links um 90°, also um einen Viertelkreis.

Aufgabe 1.8 Schreibe zu den folgenden Drehbefehlen jeweils einen äquivalenten Befehl mit der gleichen Wirkung, jedoch durch eine Drehung in die Gegenrichtung. Zum Beispiel `rt 90` ist in der Wirkung äquivalent zu `lt 270`. Überprüfe deine Vorschläge durch Eintippen der Befehle.

a) `rt 180`

b) `lt 90`

c) `rt 10`

d) `lt 45`

Aufgabe 1.9 Tippe das folgende Programm für die Schildkröte in der Startposition.

```
fd 100
rt 90
fd 150
rt 90
fd 50
lt 90
fd 150
rt 90
fd 50
```

Aufgabe 1.10 Tippe das folgende Programm:

```
fd 100
rt 90
fd 260
rt 90
fd 80
rt 90
fd 100
rt 90
fd 50
```

Zeichne das entstandene Bild und beschreibe, welcher Befehl was verursacht hat.

Aufgabe 1.11 Tippe das folgende Programm für die Schildkröte in der Startposition und lasse es ausführen. Kannst du alle `lt` Befehle durch `rt` Befehle ersetzen? Mit anderen Worten: Kannst du ein Programm schreiben, das das gleiche Bild zeichnet, aber keinen `lt` Befehl verwendet?

```
fd 100
rt 90
fd 50
lt 270
fd 100
lt 270
fd 50
lt 360
```

Aufgabe 1.12 Schreibe das Programm aus Aufgabe 1.9 so um, dass darin kein Befehl `rt` vorkommt.

Aufgabe 1.13 Führe das folgende Programm selbst aus und zeichne auf ein Blatt Papier das Bild, das dadurch entsteht. Überprüfe die Richtigkeit deiner Zeichnung, indem du das Programm eintippst und es den Rechner ausführen lässt.

```
fd 120
rt 90
fd 120
lt 90
bk 100
rt 90
bk 100
lt 90
fd 80
```

Kannst du das Programm so umschreiben, dass das gleiche Bild gezeichnet wird, aber kein `bk` Befehl darin vorkommt? Kannst du danach auch alle `lt` Befehle austauschen?

Hinweis für die Lehrperson Wenn man Befehle tippt, die aus den zwei Teilen `Befehlsname` `Parameter` wie `fd 100` bestehen, muss zwischen dem Befehlsnamen `fd` und dem Parameter `100` immer ein Leerzeichen sein. Wenn man es ohne Leerzeichen als `fd120` schreibt, dann versteht der Rechner diesen Text nicht, meldet einen Fehler und ignoriert den Befehl (bewegt die Schildkröte nicht). Dasselbe gilt, wenn man mehrere Befehle hintereinander in eine Zeile schreibt, wie zum Beispiel:

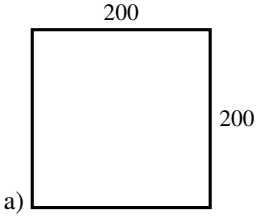
```
fd 100 rt 90 fd 50.
```

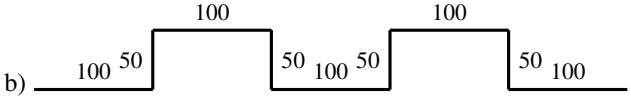
Zwischen zwei Befehlen in einer Zeile muss immer ein Leerzeichen stehen. Den Text

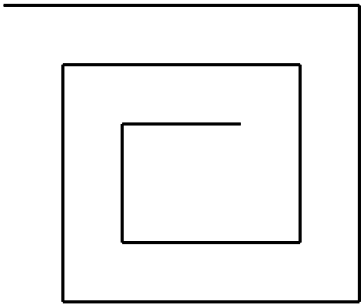
```
fd 100rt 90fd 50
```

wird der Rechner nicht verstehen.

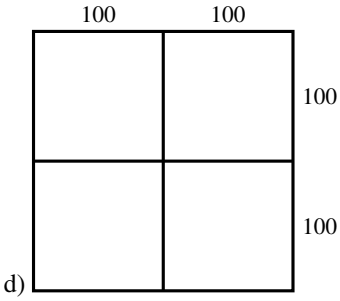
Aufgabe 1.14 Schreibe Programme, die die folgenden Bilder zeichnen. Bei allen Bildern darfst du dir die Startposition der Schildkröte bezüglich des zu zeichnenden Bildes selbst wählen.







c) Die Größe kannst du selbst wählen.



d)

Beim Schreiben von Programmen kann es uns leicht passieren, dass wir irrtümlich einen Befehl ausgeführt haben, der statt des Gewünschten etwas anderes zeichnete. Auf dem Bildschirm konnten wir es leider nicht ausradieren und so blieb uns nichts anderes übrig, als mit `cs` alles zu löschen und neu anzufangen. Jetzt führen wir zwei Befehle ein, mit denen wir die letzten Schritte rückgängig machen können. Der Befehl

`penerase` oder kurz `pe`

verwandelt die Schildkröte vom Zeichenstift zum Radiergummi. Nach dem Eintippen dieses Befehls sehen wir auf dem Bildschirm zuerst keine Änderung. Nur wenn sich die Schildkröte danach bewegt, zeichnet sie keine Linien mehr und radiert auf ihrer Strecke alle schwarzen Linien, die sie entlang läuft und alle Punkte, die sie kreuzt. Wir sagen, dass die Schildkröte vom **Stiftmodus** in den **Radiergummimodus** wechselt. Das Programm zum Beispiel:

```
fd 100
pe
bk 100
```

verursacht zuerst, dass eine schwarze Linie der Länge 100 Schritte gezeichnet wird. Danach läuft die Schildkröte im Radiergummimodus entlang der Linie zurück und radiert sie dabei wieder aus. Probier es mal!

Aufgabe 1.15 Tippe das Programm aus Aufgabe 1.9 ein, verwende danach `pe` und schreibe ein Programm, das alles Gezeichnete löscht.

Hat man die Korrekturen durch Ausradieren durchgeführt, kann man wieder in den Stiftmodus wechseln. Dazu verwendet man den Befehl

`penpaint` oder kurz `ppt`.

Aufgabe 1.16 Nutze die Befehle `pe` und `penpaint`, um die unterbrochene Linie aus Abb. 1.9 zu zeichnen.

Abbildung 1.9

Aufgabe 1.17 Überlege dir in einer Zeichnung auf einem Blatt Papier, was mit dem folgenden Programm gezeichnet und ausradiert wird. Du darfst dabei einen Bleistift und ein Radiergummi verwenden. Überprüfe deine Zeichnung, indem du das Programm eintippst und ausführen lässt.

```
fd 100
rt 180
pe
fd 50
lt 90
penpaint
fd 100
pe
bk 150
rt 90
penpaint
fd 50
```

Aufgabe 1.18 Jan will ein Quadrat 100×100 zeichnen. Er fängt folgendermaßen an:

```
fd 100
rt 90
fd 200
lt 90
fd 100
```

Da sieht er, dass er falsch angefangen hat. Kannst du mit Hilfe von `pe` und `penpaint` dieses Programm trotzdem so zu Ende schreiben, dass es das gewünschte Quadrat zeichnet?

Aufgabe 1.19 Schreibe Programme, die nach dem Zeichnen der Bilder aus der Aufgabe 1.14 a), 1.14 b) und 1.14 c) die gezeichneten Bilder schrittweise Linie für Linie ausradieren und die Schildkröte zurück in ihre Startposition bringen.

Zusammenfassung

Programme sind Folgen von Rechnerbefehlen. Die Rechnerbefehle sind einfache Anweisungen, die der Rechner ausführen kann. Die Grundbausteine einer Programmiersprache sind die einzelnen Befehle, die diese Sprache zulässt.

Wir haben angefangen, in der Programmiersprache LOGO zu programmieren. Die einfachsten Befehle sind `fd X` (gehe `X` Schritte nach vorne) und `bk Y` (gehe `Y` Schritte zurück), wobei `fd` und `bk` die Befehlswörter sind und `X` und `Y` Zahlen sind, die wir als Parameter des Befehls bezeichnen. Der Befehl `cs` löscht das bisher gezeichnete Bild und bringt die Schildkröte in ihre Startposition.

Die Befehle `rt X` und `lt Y` ermöglichen die Laufrichtung der Schildkröte um `X` Grad nach rechts, bzw. `Y` Grad nach links zu ändern. Hier sind die Befehlsnamen `rt` und `lt` und die Parameter `X` bzw. `Y` sind die Winkelgrade von 1° bis 360°. Üblicherweise ist die Schildkröte im Stiftmodus, was bedeutet, dass sie bei jeder Bewegung ihre Strecke zeichnet. Mit dem Befehl `pe` kann sie in den Radiergummimodus wechseln, in dem sie alles ausradiert, was auf ihrem Weg liegt. Durch den Befehl `penpaint` kommt sie zurück in den Stiftmodus.

Kontrollfragen

1. Was ist ein Rechnerbefehl? Was ist ein Programm?

2. Nenne alle Befehlsworte aus Lektion 1!

3. Welche Befehlsworte eines Befehls sind von einem Parameter begleitet und welche treten ohne Parameter auf?

4. Ein Befehl ist äquivalent zu einem anderen Befehl, wenn er die gleiche Wirkung auf die Bewegung der Schildkröte hat. Gebe einen Befehl an, der äquivalent zu `lt 90` ist!

5. Gebe einen Befehl an, der nichts an der Position und Orientierung der Schildkröte ändert!

6. Was ist der Radiergummimodus der Schildkröte? Was ist der Stiftmodus der Schildkröte?

7. In welchem Modus beginnt die Schildkröte ihre Arbeit?

8. Durch welchen Befehl wechselt man vom Stiftmodus in den Radiergummimodus und durch welchen Befehl kehrt man wieder in den normalen Stiftmodus zurück?

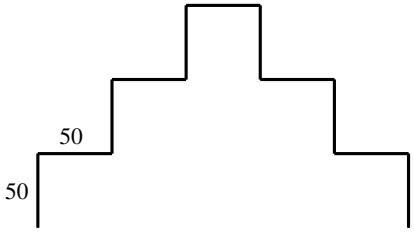
Kontrollaufgaben

1. Schreibe das folgende Programm so um, dass es nur die Befehle `fd X` und `rt Y` verwendet.

```
fd 100
lt 270
fd 50
rt 180
lt 90
fd 100
lt 270
fd 50
lt 360
```

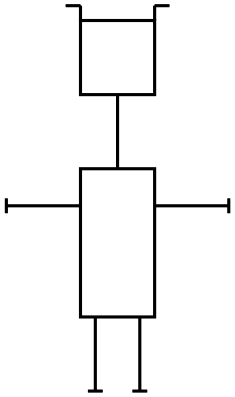
   Überprüfe auf dem Rechner, ob das Programm wirklich dasselbe macht, wie das hier gegebene Programm.

2. Schreibe ein Programm, das das folgende Bild zeichnet.



Schaffst du es, dein Programm so umzuschreiben, dass es nur die Befehle `fd 50` und `rt 90` verwendet?
Wird es auch gehen, wenn nur die Befehle die `fd 10` und `rt 90` zur Verfügung stehen?

3. Zeichne das folgende Bild mit einem Programm. Die Größen darfst du selbst wählen.



4. Zeichne dieses Bild.

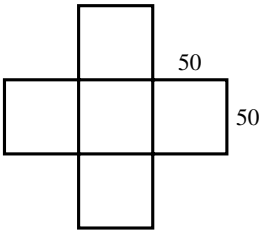


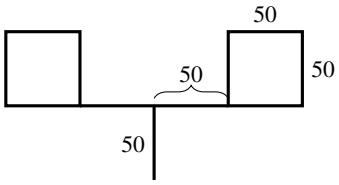
Abbildung 1.10

Von welcher Ecke dieses Bildes die Schildkröte das Zeichnen startet, darfst du selbst entscheiden.

5. Nehmen wir an, die Schildkröte hat das Bild aus Kontrollaufgabe 2 gezeichnet, befindet sich ganz rechts unten und schaut nach unten. Schreibe ein Programm, das Linie für Linie das ganze Bild ausradiert.

6. Verfahre wie in Kontrollaufgabe 5, nur dass das Bild aus der Aufgabe 4 ausradiert werden soll. Die Startposition der Schildkröte kann man sich für das Ausradieren aussuchen.

7. Anna will folgendes Bild zeichnen.

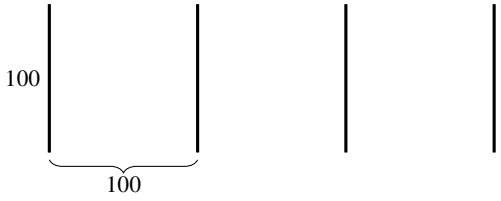


Am Anfang steht die Schildkröte unten in der Mitte. Sie hat auf folgende Weise angefangen:

```
fd 50
rt 90
fd 100
lt 270
fd 50
```

Da merkt sie, dass sie am Ende einen Fehler gemacht hat. Sie will aber nicht den Befehl `cs` verwenden, um alles zu löschen und neu anzufangen. Kannst du ihr helfen, das Bild fertig zu zeichnen?

8. Zeichne das folgende Bild mit einem Programm.



9. Zeichne die Treppe aus Kontrollaufgabe 2, Abb. 2 auf der vorherigen Seite und lösche danach durch Ausradieren alle vertikalen Linien, damit nur das Bild aus Abb. 1.11 auf der nächsten Seite bleibt.

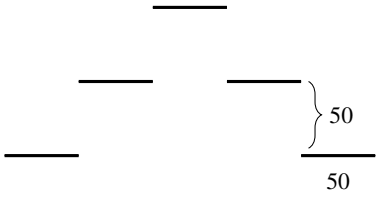


Abbildung 1.11

10. Zeichne das Bild aus Abb. 1.12. Die Maße darfst du dir selbst aussuchen.

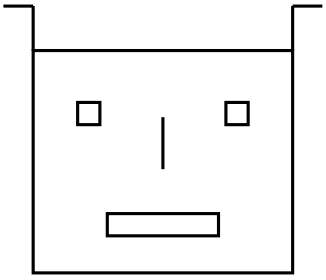


Abbildung 1.12

Lösungen zu ausgesuchten Aufgaben

Aufgabe 1.5

a) Es geht dreimal hintereinander nach vorne. Wenn wir die jeweilige Schrittzahlen 15, 30, 45 addieren, bekommen wir die Länge 90 für die gezeichnete Linie. Eine solche Linie können wir also mit dem Befehl

```
fd 90
```

auf einmal zeichnen.

b) Die Schildkröte geht zuerst 100 Schritte nach vorne, dann 50 Schritte rückwärts. Damit wurde eine Linie der Länge 100 gezeichnet und die Schildkröte befindet sich in der Mitte dieser Linie. Wenn sie jetzt 70 Schritte nach vorne geht, läuft sie zuerst 50 Schritte auf der schon gezeichneten Linie und danach noch 20 Schritte weiter. Mit diesen letzten 20

Schritten wird die Linie um 20 Schritte verlängert. Somit entsteht am Ende eine Linie mit der Länge 120. Diese Linie kann mit einem Befehl

```
fd 120
```

direkt gezeichnet werden.

Aufgabe 1.8

a) Mit dem Befehl `rt 180` zwingen wir die Schildkröte, einen Halbkreis zu drehen und damit in die Gegenrichtung ihrer bisherigen Richtung zu schauen. Dieselbe Situation erreicht man, wenn man den Halbkreis (180°) nach links dreht. Also ist `lt 180` ein Befehl, der äquivalent zu dem Befehl `rt 180` ist.

b) `rt 270`

c) `lt 350`

d) `rt 315`

Aufgabe 1.11

```
fd 100 rt 90 fd 50 rt 90 fd 100 rt 90 fd 50
```

Aufgabe 1.14

b)
```
rt 90 fd 100 lt 90 fd 50 rt 90 fd 100 rt 90 fd 50
lt 90 fd 100 lt 90 fd 50 rt 90 fd 100 rt 90 fd 50
lt 90 fd 100
```

c)
```
rt 90 fd 200
rt 90 fd 175
rt 90 fd 175
rt 90 fd 150
rt 90 fd 150
rt 90 fd 125
rt 90 fd 125
rt 90 fd 100
```

```
rt 90 fd 100
```

Aufgabe 1.19
Betrachten wir nun die Aufgabe, das Bild aus Aufgabe 1.14 c) auszuradieren. Im Prinzip reicht es, nach dem Befehl

```
pe
```

die Schildkröte mit dem Befehl

```
rt 180
```

umzudrehen und danach ein Programm zu schreiben, das aus dieser inneren Position (Die ursprüngliche Startposition war die äußere Ecke links oben) das Bild zeichnen würde. Das geht folgendermaßen:

```
fd 100 lt 90
fd 100 lt 90
fd 125 lt 90
fd 125 lt 90
...
```

und so weiter. Ich glaube, dass du es schaffst, das Programm selbst zu Ende zu schreiben.

Lektion 2

Einfache Schleifen mit dem Befehl repeat

In dieser Lektion lernen wir einen Befehl kennen, der es uns ermöglicht, mit kurzen Programmen wirklich komplexe Bilder zu zeichnen. Wie wir schon erkannt haben, sind Informatiker, wie auch viele andere Menschen, ziemlich faul, wenn es um langweilige Wiederholungen von Routinetätigkeiten geht. Und das wiederholte Tippen von gleichen Texten gehört mit zu den langweiligsten Beschäftigungen. Wenn man mit den bisherigen Befehlen fünf Quadrate zeichnen sollte, müsste man das Programm zur Zeichnung eines Quadrats fünfmal hintereinander aufschreiben. Fünfmal könnte man das schon machen, auch wenn es langweilig ist. Aber wenn man es einhundertmal machen müsste, würde einem das Programmieren wohl keinen Spaß mehr machen. Deswegen führen wir den Befehl

```
repeat
```

ein, der es uns ermöglicht, dem Rechner zu sagen:

> *„Wiederhole dieses Programm (diesen Programmteil) so und so viele Male“.*

Wie das genau funktioniert, erklären wir erst einmal an einem Beispiel. Wenn wir ein Quadrat der Größe 100×100 zeichnen wollen, geht das mit dem Programm:

J. Hromkovič, *Lehrbuch Informatik*, DOI 10.1007/978-3-8348-9692-6_3,

```
fd 100
rt 90
fd 100
rt 90
fd 100
rt 90
fd 100
rt 90.
```

Wir beobachten, dass sich die Befehle

```
fd 100
rt 90
```

viermal wiederholen. Wäre es da nicht einfacher, dem Rechner zu sagen, dass er diese zwei Befehle viermal wiederholen soll?
Wir können das wie folgt tun:

`repeat`	`4`	[`fd 100 rt 90`]
Befehlswort zum Wiederholen	Die Anzahl der Wiederholungen	[Das Programm, das wiederholt werden soll]

Tippe dieses Programm ab, um sein korrektes (vorhergesagtes) Verhalten zu überprüfen.

Aufgabe 2.1 Nutze den Befehl `repeat`, um ein Programm zur Zeichnung eines Quadrats der Größe 200×200 zu schreiben.

Aufgabe 2.2 Betrachte das folgende Programm.

```
fd 100 rt 90
fd 200 rt 90
fd 100 rt 90
fd 200 rt 90
```

Was zeichnet das Programm? Kannst du den Befehl `repeat` anwenden, um das Programm kürzer zu schreiben?

Aufgabe 2.3 Tippe das folgende Programm, um zu sehen, was es zeichnet.

```
fd 50 rt 60
fd 50 rt 60
fd 50 rt 60
fd 50 rt 60
fd 50 rt 60
fd 50 rt 60
```

Schreibe es kürzer, indem du den Befehl `repeat` verwendest.

Beispiel 2.1 Unsere Aufgabe ist es, die Treppe aus Abb. 2.1, die nach unten rechts geht, zu zeichnen.

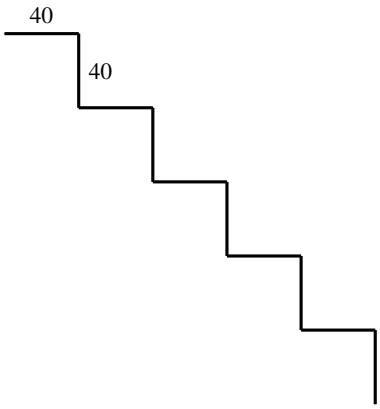


Abbildung 2.1

Am Anfang schaut die Schildkröte nach oben, deswegen fangen wir mit dem Befehl

```
rt 90
```

an, um sie in die richtige Richtung zu navigieren. Eine einzelne Stufe können wir jetzt einfach mit einem Programm

```
fd 40 rt 90 fd 40
```

zeichnen. Danach wird die Schildkröte nach unten schauen. Um die nächste Stufe der Treppe zu zeichnen, müssen wir zuerst mit dem Befehl

```
lt 90
```

die Blickrichtung der Schildkröte anpassen. Wir sehen, dass die folgende Tätigkeit fünfmal wiederholt werden muss:

`fd 40 rt 90 fd 40`	`lt 90`
Zeichnen einer Stufe	Ausrichtung der Schildkröte zum Zeichnen einer neuen Stufe

Damit sieht unser Programm für fünf Treppenstufen wie folgt aus:

```
rt 90 repeat 5 [ fd 40 rt 90 fd 40 lt 90 ]
```

□

Aufgabe 2.4

a) Zeichne eine steigende Treppe mit zehn Stufen der Größe 20, wie sie in Abb. 2.2 zu sehen ist.

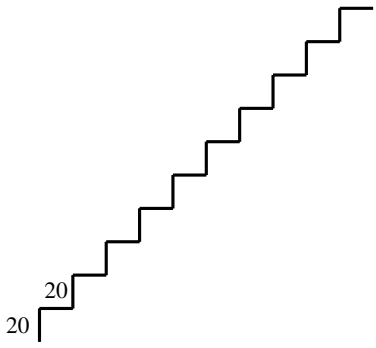


Abbildung 2.2

b) Zeichne eine Treppe mit fünf Stufen der Größe 50, die nach rechts oben geht.

c) Zeichne eine Treppe mit 20 Stufen der Größe 10, die von rechts oben nach links unten geht.

Aufgabe 2.5 Zeichne das Bild aus Abb. 2.3 auf dem Rechner nach.

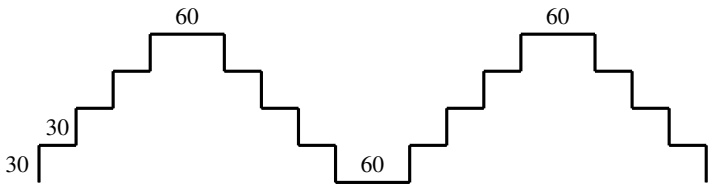


Abbildung 2.3

Aufgabe 2.6 Tippe das folgende Programm ab und führe es aus!

```
repeat 4 [ fd 100 rt 90 ]
rt 90
repeat 4 [ fd 100 rt 90 ]
rt 90
repeat 4 [ fd 100 rt 90 ]
rt 90
repeat 4 [ fd 100 rt 90 ]
rt 90
```

Was entsteht dabei? Schaffst du es, dieses Programm noch kürzer zu schreiben? Das oben geschriebene Programm besteht aus 16 Befehlen (4×`repeat`, 4×`fd` und 8×`rt`). Es ist möglich, es mit fünf Befehlen zu schreiben.

Hinweis für die Lehrperson Die folgende Einführung in die Terminologie können Schüler unter zwölf Jahren überspringen.

Der Befehl

```
repeat X [ Programm ]
```

verursacht, dass das in Klammern geschriebene `Programm` `X`-mal hintereinander ausgeführt wird. Für `X` können wir eine beliebige positive ganze Zahl wählen. Diesen Befehl bezeichnen wir auch als **Schleife**, die `X`-mal realisiert wird. Das `Programm` nennen wir auch **Körper** der Schleife. Wir sagen, dass eine Schleife in eine andere Schleife gesetzt wurde, wenn der Körper einer Schleife auch eine Schleife enthält.

Zum Beispiel bei

```
repeat 10   [   repeat 4   [ fd 100 rt 90 ] ]
äußere Schleife  innere Schleife
```

enthält der Schleifenbefehl `repeat 10 [ ... ]` in seinem Körper wieder eine Schleife `repeat 4 [ ... ]`. Aufgabe 2.5 hat uns aufgefordert, eine Schleife in eine andere Schleife zu setzen.

Wenn man das Programm

```
repeat 10 [ repeat 4 [ fd 100 rt 90 ] ]
```

eintippt, stellt man fest, dass es das gleiche Bild erzeugt wie das Programm

```
repeat 4 [ fd 100 rt 90 ].
```

Das kommt dadurch zustande, dass man zehnmal hintereinander das gleiche Bild auf der gleichen Stelle zeichnet. Wenn man aber, nach jeder Zeichnung eines 100×100 Quadrates, die Schildkröte einwenig dreht,

```
repeat 10 [ repeat 4 [ fd 100 rt 90]    rt 20    ]
                                    zusätzliche Drehung
```

erhält man ein interessantes Bild. Probier es aus!

Wenn man darauf achtet, dass die Schildkröte nach dem Zeichnen die ursprüngliche Ausgangsposition annimmt, ist das Bild noch vollkommener. Dazu muss man die Schildkröte insgesamt um 360° drehen, also

> *Die Anzahl der Wiederholungen der äußeren Schleife $\times$ die Größe der zusätzlichen Drehung muss 360 ergeben.*

Probiere zum Beispiel

```
repeat 36 [ repeat 4 [ fd 100 rt 90] rt 10 ]
```

oder

```
repeat 12 [ repeat 4 [ fd 60 rt 90] rt 30 ]
```

oder

```
repeat 18 [ repeat 2 [ fd 100 rt 90 fd 30 rt 90 ] rt 20]
```

Du darfst gerne auch ein paar eigene Ideen ausprobieren.

Aufgabe 2.7

a) Zeichne den Stern aus Abb. 2.4:

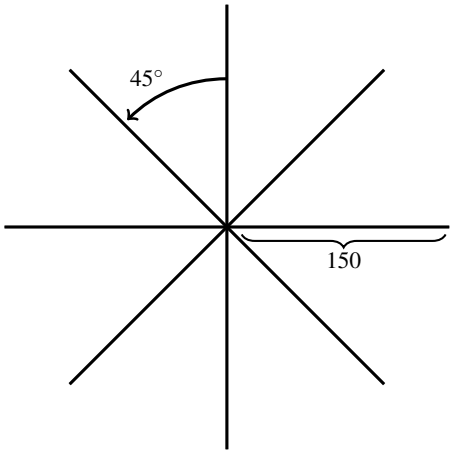


Abbildung 2.4

b) Der Stern aus a) hat acht „Strahlen“ der Länge 150. Kannst du auch einen Stern mit 16 Strahlen der Länge 100 zeichnen?

Aufgabe 2.8

a) Das Bild aus Aufgabe 2.5 (Abb. 2.3 auf Seite 43) können wir als zwei Pyramiden sehen. Schreibe ein Programm, dass vier statt zwei solcher Pyramiden hintereinander zeichnet.

b) Versuche das Programm zum Zeichnen einer Pyramide so kurz wie möglich zu schreiben!

c) Versuche auch ein kurzes Programm für das Bild in Abb 2.3 auf Seite 43 zu schreiben.

Aufgabe 2.9 Schreibe ein Programm zum Zeichnen des Bildes aus Abb. 2.5:

Abbildung 2.5

Beispiel 2.2 Die Aufgabe ist, das Bild aus Abb. 2.6 zu zeichnen.

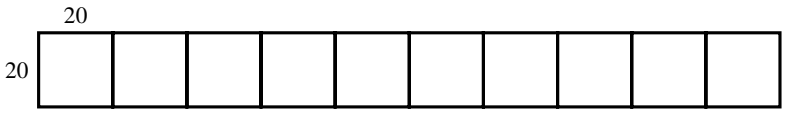


Abbildung 2.6

Es gibt mehrere Strategien zur Lösung dieser Aufgabe. Wir präsentieren zwei davon.

Erste Lösung: Wir können diese Aufgabe als eine Aufforderung verstehen, zehnmal nebeneinander ein Quadrat der Größe 20×20 zu zeichnen. Das Programm

```
repeat 4 [ fd 20 rt 90]
```

zum Zeichnen eines Quadrats kennen wir schon sehr gut. Die Hauptaufgabe hier ist es zu bestimmen, wie sich die Schildkröte aus der Position in Abb. 2.7 auf der nächsten Seite nach dem Zeichnen eines Quadrates bewegen soll, um danach das nächste Quadrat zu zeichnen.

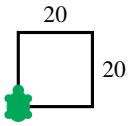


Abbildung 2.7

Die Position, die wir erreichen müssen, um das nächste Quadrat zu zeichnen ist, in Abb 2.8 dargestellt.

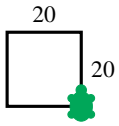


Abbildung 2.8

Wir sehen sofort, dass wir das mit dieser Folge von Befehlen

```
rt 90 fd 20 lt 90
```

bewirken können. Damit ist klar, dass wir nur zehnmal das Programm

```
repeat 4 [ fd 20 rt 90 ]
rt 90 fd 20 lt 90
```

zu wiederholen brauchen. Also zeichnen wird das gewünschte Bild mit dem Programm:

```
repeat 10 [ repeat 4 [ fd 20 rt 90 ]
            rt 90 fd 20 lt 90 ]
```

`repeat 4 [ fd 20 rt 90 ]`: ein Quadrat 20×20 zeichnen

`rt 90 fd 20 lt 90`: Bewegung zur neuen Startposition

Zweite Lösung Die Idee ist es, zuerst den Umfang zu zeichnen (Abb 2.9 auf der nächsten Seite).

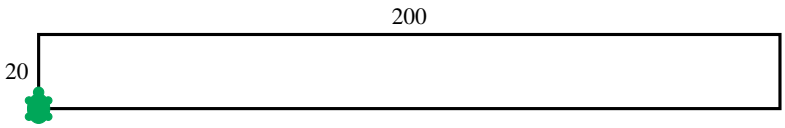


Abbildung 2.9

Das geht mit dem Programm

```
fd 20 rt 90 fd 200 rt 90 fd 20 rt 90 fd 200 rt 90
```

oder kürzer mit

```
repeat 2 [ fd 20 rt 90 fd 200 rt 90 ].
```

Bei beiden Programmen beendet die Schildkröte ihre Arbeit in der Startposition. Jetzt muss man noch die neun fehlenden Linien der Länge 20 zeichnen. Wir können zuerst mit

```
rt 90 fd 20 lt 90
```

die Schildkröte an eine gute Position (s. Abb. 2.10) zum Zeichnen der ersten fehlenden Linie bringen.

Abbildung 2.10

Mit `fd 20` wird jetzt die Linie gezeichnet. Die neue Position der Schildkröte ist in Abb. 2.11 auf der nächsten Seite abgebildet.

Um die nächste Linie mit `fd 20` zeichnen zu können, muss die Schildkröte zuerst die Position aus Abb. 2.12 auf der nächsten Seite erreichen.

Abbildung 2.11

Abbildung 2.12

Die Schildkröte bewegt sich aus der Position in Abb. 2.11 zur Position in Abb. 2.12 mit der folgenden Befehlsfolge:

```
rt 180 fd 20 lt 90 fd 20 lt 90.
```

Jetzt sind wir in der Position aus Abb. 2.12, die ähnlich zur Position in Abb. 2.10 auf der vorherigen Seite ist, und man sieht, dass wir das Vorgehen von Abb. 2.11 und Abb. 2.12 noch achtmal wiederholen müssen.
Damit erhalten wir das gesamte Programm:

```
repeat 2 [ fd 20 rt 90 fd 200 rt 90 ]
rt 90 fd 20 lt 90    (Situation in Abb. 2.10 erreicht)
repeat 9 [ fd 20 rt 180 fd 20 lt 90 fd 20 lt 90 ]
```

□

Aufgabe 2.10 Zeichne das gleiche Bild (Abb. 2.6 auf Seite 46) aus Beispiel 2.2, allerdings mit der rechten unteren Ecke des Bildes als Startposition der Schildkröte. In unserer Musterlösung war die Schildkröte zu Beginn in der linken unteren Ecke des Bildes. Erkläre dabei dein Vorgehen genauso, wie wir es im Beispiel 2.2 erklärt haben.

Beispiel 2.3 Die Aufgabe ist es, das Bild aus Abb. 2.13 auf der nächsten Seite zu zeichnen.

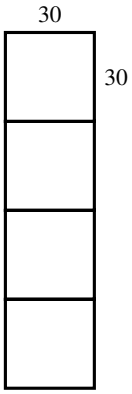


Abbildung 2.13

Wir können ähnlich wie in der Musterlösung zu Beispiel 2.2 vorgehen und vier Quadrate der Größe 30×30 von unten nach oben zeichnen. Wir haben schon gelernt, dass die Struktur des Programms wie folgt aussehen muss:

```
repeat 4    [ Zeichne ein Quadrat 30×30.
              Bewege die Schildkröte auf die Position zum
              Zeichnen des nächsten Quadrats. ]
```

In Abb. 2.14 sehen wir die Position der Schildkröte nach dem Zeichnen eines Quadrats (a) und die Situation vor dem Zeichnen des nächsten Quadrats (b).

Abbildung 2.14

Dieser Schritt kann sehr leicht durch den Befehl `fd 30` erreicht werden. Somit sieht unser gesamtes Programm wie folgt aus:

```
repeat 4 [ repeat 4 [ fd 30 rt 90 ] fd 30 ].
```
□

Wie würde das Programm aussehen, wenn man das Bild von oben nach unten statt von unten nach oben zeichnen würde?

Überlege dir ein Programm, das das Bild mit einer ähnlichen Strategie zeichnet, wie wir sie in der zweiten Lösung vom Beispiel 2.2 verwendet haben.

Aufgabe 2.11

a) Zeichne nebeneinander 25 Quadrate der Größe 15×15. (In Abb. 2.6 auf Seite 46 stehen zehn Quadrate der Größe 20×20)

b) Zeichne übereinander sieben Quadrate der Größe 40×40. (In Abb. 2.13 auf der vorherigen Seite sind vier Quadrate 30×30 übereinander gezeichnet)

Aufgabe 2.12 Zeichne das Bild aus Abb. 2.15

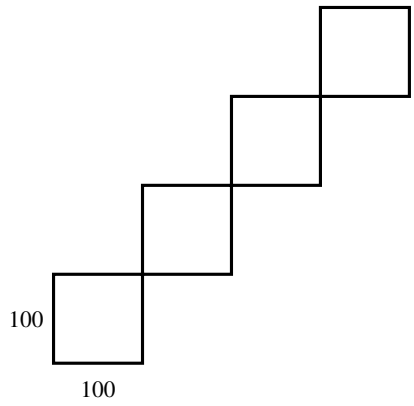


Abbildung 2.15

Hinweis für die Lehrperson Für Kinder unter 12 Jahren kann man die folgenden Aufgaben entweder ganz weglassen oder am Ende von Lektion 3 noch einmal stellen.

Aufgabe 2.13 a) Zeichne das Feld aus Abb. 2.16 auf der nächsten Seite. Das Feld hat vier Zeilen und zehn Spalten und besteht aus 40 Quadraten der Größe 20×20. Deswegen nennen wir es 4×10-Feld mit Quadratgröße 20.

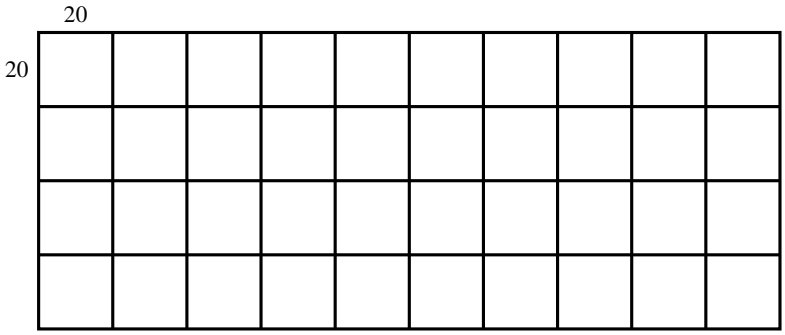


Abbildung 2.16

b) Zeichne ein 8×8 Feld mit Quadratgröße 30.

c) Zeichne ein 7×3 Feld mit Quadratgröße 25.

Wir haben gelernt, dass die Schildkröte in zwei Modi sein kann. Der gewöhnliche Modus ist der Schriftmodus, in dem sie zeichnet und der zweite Modus ist der Radiergummimodus, in dem sie unterwegs alles ausradiert. Es gibt noch einen dritten Modus, den man als **Wandermodus** bezeichnet. In diesem Modus bewegt sie sich nur auf dem Bildschirm, ohne dabei zu zeichnen oder zu radieren. Also wandert sie entspannt, ohne zu arbeiten. In diesen Modus kommt man mit dem Befehl

`penup` oder kürzer `pu`.

Aus dem Wandermodus zurück in den Stift- bzw. Radiermodus, je nachdem in welchem Modus man vorher war, kommt man mit dem Befehl

`pendown` oder kurz `pd`.

Somit kann man das Bild in Abb. 2.17 auf der nächsten Seite mit folgendem Programm erzeugen.

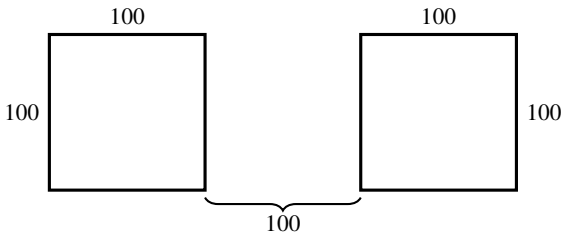


Abbildung 2.17

```
repeat 4 [ fd 100 rt 90 ]
pu
rt 90 fd 200 lt 90
pd
repeat 4 [ fd 100 rt 90 ]
```

Aufgabe 2.14 Zeichne das Bild aus Abb. 2.18.

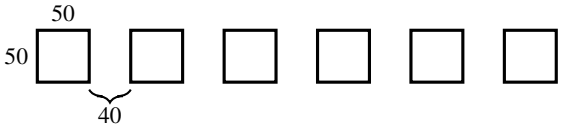
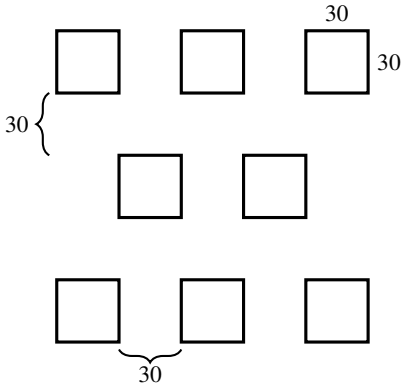


Abbildung 2.18

Aufgabe 2.15 Zeichne das folgende Bild.



Aufgabe 2.16 Zeichne das Bild aus Abb. 2.19.

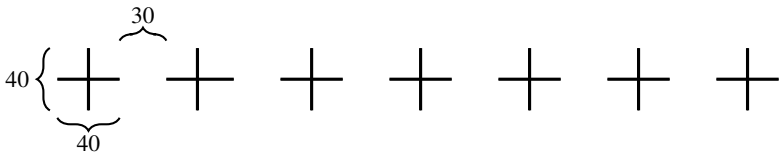


Abbildung 2.19

Aufgabe 2.17 Zeichne das Bild aus Abb. 2.20.

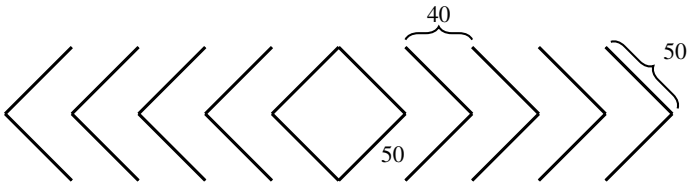


Abbildung 2.20

Kontrollfragen

1. Erkläre mit wenigen Worten, wozu der Befehl `repeat` gut ist. Wie sieht seine Struktur aus?

2. Gebe ein Beispiel einer Aufgabe, bei deren Lösung du durch die Verwendung des Befehls `repeat` sehr viel Tippen sparen kannst.

3. Mittels eines `repeat`-Befehls wiederholen wir die Ausführung eines Programms mehrere Male. In unseren Aufgaben besteht der Körper der Schleifen oft aus zwei Teilen. In dem ersten Teil zeichnet man ein konkretes Bild (z.B. Quadrat, Stufe, usw.). In dem zweiten Teil geht es meist nicht darum etwas zu zeichnen. Wozu verwenden wir den zweiten Teil?

4. Wie nennen wir das Programm im Befehl (in der Schleife)

   ```
   repeat X [ Programm ]?
   ```

5. Was bedeutet es, eine Schleife in eine andere Schleife zu setzen? Gebe ein Beispiel.

6. Was macht die Schildkröte im Wandermodus? Durch welchen Befehl kann man den Wandermodus erreichen?

7. Durch welchen Befehl kommt die Schildkröte aus dem Wandermodus zurück in den ursprünglichen Modus?

8. Brauchen wir den Wandermodus überhaupt? Können wir nicht alles nur mit Hilfe des Radiergummimodus machen? Wo ist der Unterschied zwischen diesen beiden Modi?

Kontrollaufgaben

1. Zeichne ein Rechteck der Größe 50×150 mit einem Programm, das mit dem Befehl `repeat` anfängt.

2. Zeichne ein 3×10 Feld mit Quadratgröße 25.

3. Zeichne das Bild aus Abb. 2.21.

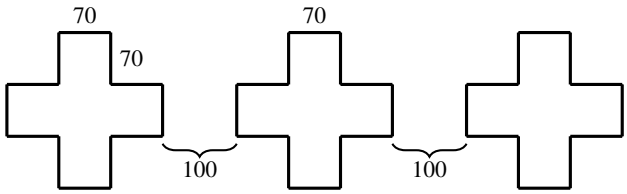


Abbildung 2.21

4. Wenn du es nicht schon bei der Lösung der Kontrollaufgabe 3 gemacht hast, schreibe ein kurzes Programm zum Zeichnen der drei Kreuze aus Abb 2.21, das eine Schleife in einer anderen Schleife enthält.

5. Zeichne sieben Kreuze nebeneinander, wie die drei aus Abb. 2.21, mit dem Unterschied, dass die Seiten der Kreuze die Länge 20 haben und der Abstand zwischen zwei Kreuzen auch 20 ist.

6. Schreibe ein Programm, das den Baum aus Abb. 2.22 auf der nächsten Seite zeichnet.

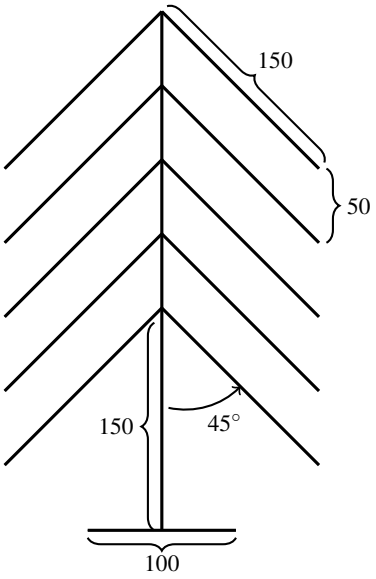


Abbildung 2.22

Lösungen zu ausgesuchten Aufgaben

Aufgabe 2.12
Mit dem Programm

```
repeat 6 [ fd 100 rt 90 ]
```

zeichnen wir ein Quadrat der Größe 100. Die Schildkröte befindet sich nach der Zeichnung des Quadrates in der rechten oberen Ecke des Quadrats und schaut nach unten. (Probier es aus!) Wenn wir sie jetzt mittels

```
rt 180
```

umdrehen, können wir das nächste Quadrat zeichnen. Somit sieht unser Programm für das Bild in Abb. 2.15 auf Seite 51 wie folgt aus:

```
repeat 4 [ repeat 6 [ fd 100 rt 90 ] rt 180 ].
```

Aufgabe 2.14
Ein Quadrat mit einer Seitenlänge von 50 zeichnen wir wie üblich mit dem Programm

```
repeat 4 [ fd 50 rt 90 ].
```

Danach wollen wir die Schildkröte zum Startpunkt für die Zeichnung des nächsten Quadrats im Wandermodus, also ohne eine Spur zu hinterlassen, bewegen. Dies machen wir mit dem Programm:

```
pu
rt 90 fd 90 lt 90
pd
```

Somit sieht das gesamte Programm wie folgt aus:

```
repeat 6 [ repeat 4 [ fd 50 rt 90 ] pu rt 90 fd 90 lt 90 pd ].
```

Weißt du, ohne das Programm laufen zu lassen, wo die Schildkröte nach der Zeichnung des Bildes steht?

Lektion 3

Programme benennen und aufrufen

Die Unwilligkeit der Programmierer, langweilige und monotone Tätigkeiten wie zum Beispiel Tippen auszuführen, kennt keine Grenzen. Deswegen brachten sie außer dem Befehl `repeat` noch weitere Ideen und Konzepte hervor. Das Konzept der Benennung von Programmen hat aber auch eine andere Wurzel als diese gesunde „Faulheit“.

Stellt euch mal vor, dass man, um ein kompliziertes Bild zu zeichnen, mehrere tausend Befehle schreiben muss. In der Praxis gibt es Programme, die aus Millionen von Befehlen bestehen. Wenn jetzt aber ein so langes Programm nicht das Gewünschte tut, wie soll man in der Unmenge von Befehlen nach den Fehlern suchen? Schon bei einem Programm mit hundert Befehlen ist es eine mühsame Arbeit und bei sehr langen Programmen ist es fast unmöglich, den Fehler zu entdecken. Deswegen muss man beim Entwurf von Programmen sehr sorgfältig und „systematisch“ vorgehen. Was aber bedeutet das Wort systematisch genau? Man kann das gut beim Bauen einer Stadt aus Legosteinen erklären. Zuerst lernen wir die Art, wie man einfache Häuser baut und bauen ein paar Häuser. Danach lernen wir aus den einzelnen Häusern die Straßen zusammenzusetzen. Und aus den Straßen setzen wir schlussendlich die ganze Stadt zusammen. Ähnlich geht es beim Programmieren. Man schreibt zuerst kleine Programme, die einfache Tätigkeiten ausführen (z. B. Bilder zeichnen). Man überprüft sie sorgfältig, bis sie korrekt arbeiten. Danach benutzt man diese einfachen Programme als Bausteine, aus denen man kompliziertere Programme baut. Die komplizierteren Programme kann man wieder überprüfen und als Bausteine für den Bau von noch komplizierteren Programmen verwenden, und das kann dann immer so weiter gehen. Die als Bausteine verwendeten Programme nennen wir **Module** und deshalb nennt man diesen systematischen Aufbau (Entwurf) von Programmen **modular**.

Im Folgenden wollen wir genau diesen modularen Entwurf von Programmen kennenler-

J. Hromkovič, *Lehrbuch Informatik*, DOI 10.1007/978-3-8348-9692-6_4,

nen. Wegen der Faulheit und der Übersichtlichkeit der Programmdarstellung wollen wir unseren Modulen (Programmen) Namen geben. Der Vorteil ist, dass der Rechner diese Namen speichert. Wir dürfen diese Namen dann einfach als neue Befehle verwenden. Wenn wir den Namen eines Programms eintippen, ersetzt der Rechner automatisch diesen Namen durch das ganze entsprechende Programm und führt es zum gegebenen Zeitpunkt auch aus. Erklären wir es genauer an einem Beispiel.

Wir mussten oft ein Quadrat mit der Seitenlänge 100 zeichnen. Das haben wir mit dem Programm

```
repeat 4 [ fd 100 rt 90 ]
```

gemacht. Wir wollen nun dieses Programm QUAD100 nennen (den Namen dürfen wir uns selbst aussuchen). Das tun wir mittels der Befehle to und end. Dazu schreiben wir im Editor das folgende Programm:

```
to QUAD100
repeat 4 [ fd 100 rt 90 ]
end
```

Wenn der Rechner das Befehlswort to liest, dann weiß er, dass danach die Benennung eines Programms folgt. Das nächste Wort QUAD100 betrachtet er als den Namen und alles, was danach bis zu dem Befehl end kommt, sieht er als das Programm mit diesem Namen an. Der Rechner bewegt dabei die Schildkröte nicht und macht auch sonst nichts, was man sehen könnte. Er speichert das Programm nur in seinem Speicher (Gehirn) unter dem Namen QUAD100 und meldet uns, dass er das gemacht hat. Auf diese Weise haben wir einen neuen Befehl QUAD100 definiert. Wenn man jetzt

```
QUAD100
```

schreibt, dann wird das entsprechende Programm

```
repeat 4 [ fd 100 rt 90 ],
```

das man den **Körper** des Programms QUAD100 nennt, ausgeführt und somit erscheint ein Quadrat 100×100 auf dem Bildschirm. Das Schreiben von QUAD100 nennt man den Aufruf des Programms QUAD100. Wir sehen, wie viel Schreibarbeit wir sparen, wenn wir von jetzt an den Namen des Programms tippen, anstatt das ganze Programm jedes Mal aufs Neue zu schreiben.

Das allgemeine Schema für die Benennung von Programmen sieht wie folgt aus:

```
to NAME
Programm
end
```

Das `Programm` darf beliebig lang sein. Der Name `NAME` ist dadurch zu einem neuen Befehl geworden, den wir beliebig oft verwenden dürfen.

Aufgabe 3.1 Tippe die oben stehende Benennung des Programms `QUAD100` mittels der Befehle `to` und `end` ein. Tippe danach den Befehl `QUAD100` ein, um die Funktionalität zu überprüfen.

Aufgabe 3.2 Schreibe ein Programm zum Zeichnen eines Quadrats mit der Seitenlänge 200 und benenne es `QUAD200`. Teste danach die Wirkung des neuen Befehls `QUAD200`.

Wir dürfen beliebig viele Programme benennen und dadurch beliebig viele, neue Befehle erzeugen. Allen Programmen, die wir öfter benutzen wollen, geben wir einfach einen Namen.

Hinweis für die Lehrperson Die Namen der Programme können aus beliebigen Buchstaben und Ziffern zusammengesetzt werden. Sie müssen nur mit einem Buchstaben anfangen. Trotzdem empfehlen wir, die Namen nicht willkürlich zu wählen und, um Schreibarbeit zu sparen, Namen wie A, B oder C zu verwenden, die nur aus einem Buchstaben bestehen. Das Risiko ist sehr groß, dass man bei zu vielen Programmen und Namen irgendwann nicht mehr weiss, welcher Name für welches Programm steht. Natürlich hat man die Möglichkeit, ein „Wörterbuch“ der Programmnamen in einem Heft zu führen, was unabhängig von der Namenszuordnung sehr empfehlenswert ist. Die beste Strategie bei der Namensgebung ist, Namen zu vergeben, die die Aufgabe (die Aktivität) des Programms widerspiegeln. Zum Beispiel der Name QUAD100 deutet sehr klar an, dass es sich um ein Programm handelt, das ein Quadrat der Seitenlänge 100 zeichnet.

Wir können jetzt das Programm `QUAD100` nutzen, um systematisch mit der modularen Technik das Bild aus Abb. 2.15 auf Seite 51 zu zeichnen. Das modular gebaute Programm ist

```
repeat 4 [ QUAD100 fd 100 rt 90 fd 100 lt 90 ].
```

Aufgabe 3.3 Verwende `QUAD100`, um das Bild aus Abb. 2.17 auf Seite 53 zu zeichnen.

Aufgabe 3.4 Verwende `QUAD50`, um das Bild aus Abb. 1.10 auf Seite 33 zu zeichnen.

Verwenden wir jetzt die modulare Entwurfstechnik zur Zeichnung der Felder. Fangen wir mit einem 1×10-Feld mit Quadratgröße 20 wie in der Abb. 2.6 auf Seite 46 an. Zuerst schreiben und benennen wir ein Programm für Quadrate mit der Kantenlänge 20.

```
to QUAD20
repeat 4 [ fd 20 rt 90 ]
end
```

Jetzt zeichnen wir das Bild aus Abb. 2.6 auf Seite 46 mit dem Programm

```
repeat 10 [ QUAD20 rt 90 fd 20 lt 90 ].
```

Nehmen wir jetzt an, dass wir ein 5×10-Feld mit Quadratgröße 20 zeichnen wollen. Da ist es vorteilhaft, wenn wir das entworfene Programm für das 1×10-Feld benennen.

```
to FELD1M10Q20
repeat 10 [ QUAD20 rt 90 fd 20 lt 90 ]
end
```

Jetzt können wir das Programm `FELD1M10Q20` fünfmal hintereinander verwenden. Dabei müssen wir aber darauf achten, dass wir die Schildkröte nach dem Zeichnen einer Reihe von Quadraten zum richtigen Startpunkt bringen, von wo aus sie die nächste Reihe zeichnen kann. Dies macht das folgende Programm.

```
repeat 5 [ FELD1M10Q20 pu lt 90 fd 200 lt 90 fd 20 rt 180 pd ]
```

Aufgabe 3.5 Verwende die modulare Entwurfstechnik, um ein 8×8-Feld mit Quadratgröße 25 zu zeichnen.

Aufgabe 3.6 Beim Zeichnen des 5×10-Feldes mit Quadratgröße 20 sind wir so vorgegangen, dass wir zuerst ein Programm für das Zeichnen einer Reihe von zehn Quadraten (ein 1×10-Feld) geschrieben haben und danach dieses Programm fünfmal aufgerufen haben.

Gehe jetzt anders vor. Schreibe zuerst ein Programm, das eine Spalte von fünf Quadraten (ein 5×1-Feld) zeichnet und benutze dieses Programm danach zehnmal, um zehn solcher Spalten nebeneinander zu zeichnen und somit das 5×10-Feld zu erzeugen.

Aufgabe 3.7 Verwende die modulare Entwurfstechnik, um das Bild aus Abb. 2.18 auf Seite 53 zu zeichnen.

Aufgabe 3.8 Verwende die modulare Entwurfstechnik, um das Bild aus Abb. 2.19 auf Seite 54 zu zeichnen.

Aufgabe 3.9 Verwende die modulare Entwurfstechnik, um die Bilder aus Abb. **??** auf Seite ?? und Abb. 2.21 auf Seite 55 zu zeichnen.

Aufgabe 3.10 Zeichne drei Bäume wie den aus Abb. 2.22 auf Seite 56 so nebeneinander, dass sie sich gegenseitig nicht berühren.

Unser nächstes Ziel ist nicht, neue Befehle zu lernen, sondern uns Techniken zum Zeichnen neuer Bilder anzueignen, die ausgemalte Flächen haben. Wir fangen damit an, fette Linien zu zeichnen. Wir haben schon beobachtet, dass man durch mehrfaches Entlangwandern auf der gleichen Linie keine fetten Linien bekommt. Wir können es noch einmal mit dem Programm

```
repeat 10 [ fd 100 bk 100 ]
```

überprüfen. Wir sehen, dass die Linie gleich dick geblieben ist, obwohl die Schildkröte 20-mal über diese Linie von 100 Schritten gelaufen ist. Wenn man eine fettere Linie zeichnen will, muss man eigentlich zwei oder mehrere Linien dicht nebeneinander zeichnen, wie es in Abb. 3.1 gezeigt ist.

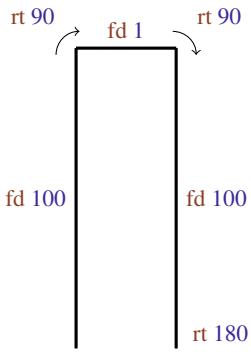


Abbildung 3.1

Das entsprechende Programm können wir FETT100 nennen.

```
to FETT100
fd 100 rt 90
fd 1 rt 90
fd 100 rt 180
end
```

Wir sehen, dass das Programm FETT100 zwei Linien der Länge 100 dicht nebeneinander in der Entfernung von einem Schritt zeichnet. Dadurch entsteht eine fettere Linie.

Aufgabe 3.11 Tippe das Programm FETT100 ab und überprüfe, ob du eine fette Linie erhältst. Versuche ein Programm zu schreiben, das zwei Linien der Länge 100 im Abstand von zwei Schritten (anstatt nur einem Schritt) zeichnet. Erhältst du mit deinem Programm eine fettere Linie oder zwei Linien, die nahe beieinander stehen, aber trotzdem getrennt sind? Du kannst es nur durch Ausprobieren feststellen.

Aufgabe 3.12 Zeichne eine Linie der Länge 200, die aus vier nebeneinander stehenden Linien besteht.

Die durch FETT100 gezeichnete Linie kann man als schwarzes Rechteck der Größe 100×2 betrachten. Wenn wir 100 Linien der Länge 100 jeweils mit dem Abstand eines Schrittes zeichnen, erhalten wir ein schwarzes Quadrat der Größe 100 (siehe Abb. 3.2)

Abbildung 3.2

Das entsprechende Programm ist

```
repeat 99 [ FETT100 ].
```

Aufgabe 3.13 Teste das Programm zum Zeichnen des schwarzen Quadrats. Weisst du, warum wir 99-mal FETT100 verwendet haben und nicht 100-mal oder 50-mal?

Aufgabe 3.14 Schreibe ein Programm `FE100`, so dass das Programm

```
repeat 50 [ FE100 ]
```

das schwarze Quadrat aus Abb. 3.2 auf der vorherigen Seite zeichnet.

Aufgabe 3.15 Was zeichnet das Programm

```
repeat 2 [ fd 100 bk 100 pu rt 90 fd 1 lt 90 pd ]?
```

Aufgabe 3.16 Schreibe Programme, die schwarze Quadrate der Größe 50, 75 und 150 zeichnen.

Aufgabe 3.17 Entwickle ein Programm, das das Bild aus Abb. 3.3 zeichnet. Gehe dabei strukturiert vor, indem du zuerst ein Programm zum Zeichnen eines schwarzen Quadrats der Größe 40 aufschreibst und benennst.

Abbildung 3.3

Das modulare Vorgehen ermöglicht uns, komplexe Bilder übersichtlich zu zeichnen. Nehmen wir uns die Aufgabe vor, ein 4×4 Schachbrett mit einer Quadratgröße von 100 wie in Abb. 3.4 auf der nächsten Seite zu zeichnen.

Zuerst benennen wir das uns schon bekannte Programm zum Zeichnen eines schwarzen Quadrats der Seitenlänge 100 (Abb. 3.2 auf der vorherigen Seite).

```
to SCHW100
repeat 99 [ FETT100 ]
end
```

Wir können jetzt das Muster des 4×4-Schachfelds wie in Abb. 3.5 in Zeilen zerlegen.

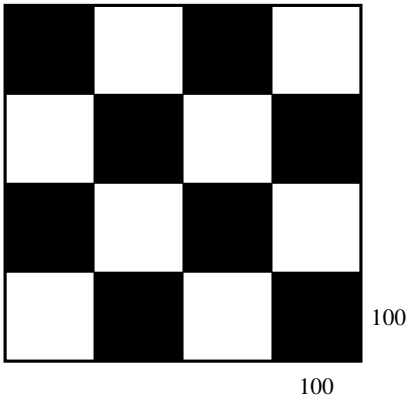


Abbildung 3.4

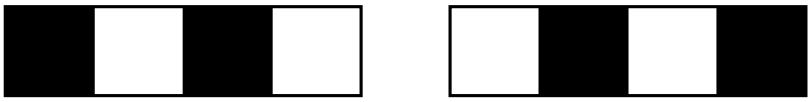


Abbildung 3.5

Die Zeile in Abb. 3.5(a) kann man mit dem folgenden Programm zeichnen.

```
to ZEILEA
repeat 2 [ SCHW100 QUAD100 rt 90 fd 100 lt 90 ]
end
```

Die Zeile in Abb. 3.5(b) kann man mit dem folgenden Programm zeichnen.

```
to ZEILEB
repeat 2 [ QUAD100 rt 90 fd 100 lt 90 SCHW100 ]
end
```

Jetzt können wir wie folgt vorgehen, um aus diesen Zeilen das Schachfeld aus Abb. 3.4 zusammenzusetzen.

```
to SCHACH4
repeat 2     [ ZEILEA pu bk 100 lt 90 fd 400 rt 90 pd
               ZEILEB pu bk 100 lt 90 fd 400 rt 90 pd ]
end
```

Aufgabe 3.18 Zeichne das 4×4-Schachfeld (Abb. 3.4 auf der vorherigen Seite), indem du zuerst Programme für die Spalten (statt für die Zeilen wie oben) modular aufschreibst und benennst.

Um ein 4×4-Schachbrett zu zeichnen, haben wir 6 Programme `SCHACH4`, `ZEILEA`, `ZEILEB`, `QUAD100`, `SCHW100`, und `FETT` entwickelt und verwendet. Das Programm `SCHACH4` zur Zeichnung des Schachbrettes nennen wir das **Hauptprogramm**. Wenn man in einem Hauptprogramm mehrere andere Programme verwendet, ist es wichtig die Struktur des Hauptprogramms anschaulich darzustellen. In dem Hauptprogramm `SCHACH4` werden die Programme `ZEILEA` und `ZEILEB` aufgerufen. Deswegen nennen wir die Programme `ZEILEA` und `ZEILEB` **Unterprogramme** von `SCHACH4`.

Im Programm `ZEILEA` werden `QUAD100` und `SCHW100` verwendet und deswegen bezeichnen wir `QUAD100` und `SCHW100` als Unterprogramme von `ZEILEA` sowie von `SCHACH4`. Das Programm `FETT100` ist ein Teil von `SCHW100` und deswegen ist `FETT100` ein Unterprogramm von `SCHW100` sowie von `ZEILEA` und `SCHACH4`. Die Bezeichnung *„Unterprogramm zu sein“* und damit auch die Struktur des Programms ist anschaulich durch das Baumdiagramm in Abb. 3.6 dargestellt. Ganz oben im Diagramm steht das Hauptprogramm `SCHACH4`. Aus `SCHACH4` führen zwei Pfeile zu seinen direkten Unterprogrammen `ZEILEA` und `ZEILEB`, die in `SCHACH4` direkt aufgerufen werden. Ähnlich gehen die Pfeile aus `ZEILEB` zu `QUAD` und `SCHW100`, die direkt in `ZEILEB` aufgerufen werden, usw. (siehe Abb. 3.6).

Eine andere Darstellung der Struktur des Programms `SCHACH4` ist in Abb. 3.7 gezeichnet.

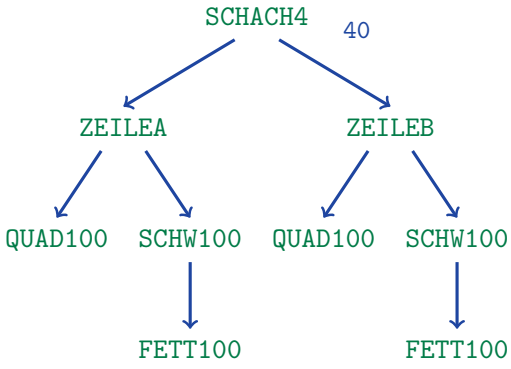


Abbildung 3.6

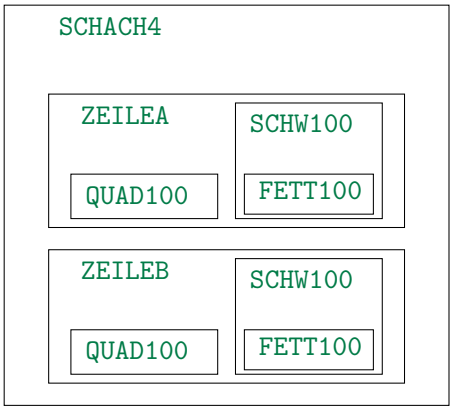


Abbildung 3.7

Hier ist direkt zu sehen, welches Programm ein Unterprogramm eines anderen Programms ist.

Aufgabe 3.19 Zeichne die Struktur des von dir in Aufgabe 3.18 entwickelten Programms auf die gleiche Weise, wie wir es für SCHACH4 in Abb. 3.6 auf der vorherigen Seite und Abb. 3.7 gemacht haben.

Aufgabe 3.20 Zeichne mit einem Programm zuerst ein 4×4-Feld mit Quadratgröße 100 und fülle dann jedes zweite Quadrat schwarz aus, um das Schachfeld aus Abb. 3.4 auf Seite 66 zu erhalten.

Aufgabe 3.21 Zeichne ein 8×8-Schachfeld mit Feldern der Größe 40×40. Gehe dabei modular vor.

In dieser Lektion haben wir gelernt, Programme zu benennen. Dadurch werden die Programme unter den entsprechenden Namen im Rechner gespeichert. Es reicht, den Namen eines Programms einzutippen, damit der Rechner das ganze Programm ausführt. Dies ermöglicht uns, effizient und übersichtlich komplexere Programme zu entwickeln. Das Ganze hat aber einen Haken: Wenn du jetzt dein LOGO verlässt, werden alle gespeicherten Programme automatisch gelöscht. Wenn du das nächste Mal weiter programmierst, stehen dir deine alten Programme nicht mehr zur Verfügung. Mit anderen Worten: Du musst das nächste Mal von vorne anfangen. Um das zu vermeiden, musst du dafür sorgen, dass alle von dir geschriebenen Programme auch langfristig gespeichert werden. Du

kannst eine Datei anlegen und benennen, in der du all deine Programme abspeichern kannst, um sie aufzurufen, wenn du sie wieder verwenden willst. Wie das funktioniert, lass dir einfach von deiner Lehrperson erklären, oder lies selbstständig den Text in der Einleitung.

Zusammenfassung

In dieser Lektion haben wir mittels des Befehls `to` gelernt, geschriebene Programme zu benennen. Der Name eines Programms steht dabei direkt hinter dem Befehl `to`. In der darauf folgenden Zeile fängt dann das eigene Programm an. Wenn der Befehl `end` vorkommt, weiß der Rechner, dass die Beschreibung des Programms abgeschlossen ist. Der Rechner speichert das Programm unter dem gegebenen Namen ab. Wenn wir dann beim Programmieren den Namen eines solchen Programms verwenden, ersetzt der Rechner den Namen durch das ganze entsprechende Programm und führt dieses Programm an dieser Stelle aus. Auf diese Weise funktioniert das Programm wie ein neuer Befehl. Du kannst also auf diese Art und Weise beliebig viele, neue Befehle entwickeln.

Wenn wir Programme benennen und sie dadurch als Bausteine (Module) zur Erzeugung komplexerer Programme verwenden, sprechen wir über den modularen Programmentwurf. Durch diese modulare Vorgehensweise bleibt die Programmentwicklung übersichtlich und wir können die korrekte Arbeitsweise von so entworfenen Programmen gut überprüfen.

Das Programm zur Zeichnung eines Bildes nennen wir das Hauptprogramm im Bezug auf alle Programme, die man in diesem Hauptprogramm aufruft. Die Programme, die innerhalb eines Programms verwendet werden, nennt man Unterprogramme dieses Programms. Benannte Programme kann man immer wieder modifizieren oder weiterentwickeln. Wenn wir sie nicht verlieren wollen, müssen wir sie aber immer nach Beendigung der Arbeit abspeichern.

Kontrollfragen

1. Wozu ist es nützlich, Programme zu benennen?

2. Wie geht man vor, wenn man ein Programm benennen will?

3. Welche Bedeutung hat der Befehl `end`?

4. Was bedeutet es, bei der Entwicklung von Programmen modular vorzugehen?

5. Welche Vorteile hat die Modularität?

6. Was bedeutet es, benannte Programme zu editieren? Wie kann man das machen?

7. Warum soll man einmal geschriebene Programme abspeichern? Und wie macht man das?

8. Wie zeichnet man in LOGO eine fette Linie?

9. Was sind Hauptprogramme und was sind Unterprogramme? Wie kann man anschaulich die Struktur eines Programms im Bezug auf seine Unterprogramme darstellen?

Kontrollaufgaben

1. Schreibe ein Programm zum Zeichnen eines schwarzen Rechtecks der Größe 50×150 und benenne es.

2. Zeichne ein 4×5-Feld der Größe 70.

3. Verwende die modulare Technik, um ein Programm zum Zeichnen des Bildes aus Abb. 3.8 zu entwickeln.

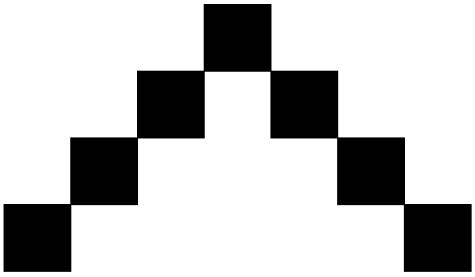


Abbildung 3.8

Schätze wie lang dein Programm wäre, wenn du den Befehl `to` nicht verwenden würdest.

4. Zeichne das Bild aus Abb. 3.9 auf der nächsten Seite.

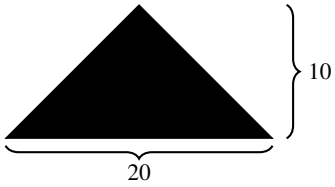


Abbildung 3.9

5. Zeichne ein 2×6-Schachfeld mit Feldern der Größe 50×50.

6. Zeichne das Häuschen aus Abb. 3.10. Die Proportionen darfst du selbst wählen.

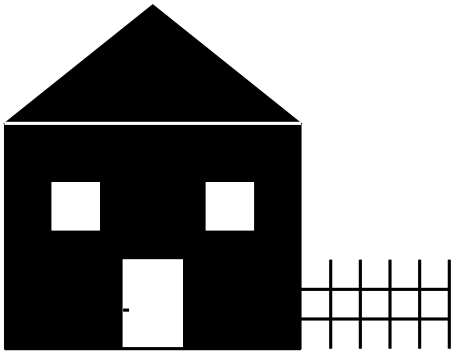


Abbildung 3.10

7. Zeichne drei Häuschen wie in Abb. 3.10 nebeneinander.

8. Zeichne das Bild aus Abb. 3.11.

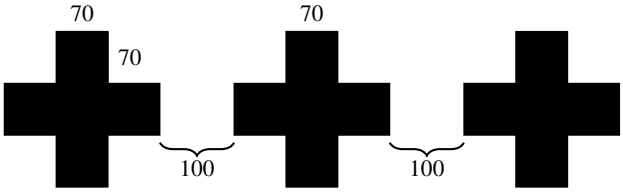


Abbildung 3.11

Lösungen zu ausgesuchten Aufgaben

Aufgabe 3.3

```
QUAD100
rt 90 pu fd 200 pd lt 90
QUAD100
```

Aufgabe 3.14

Wir zeichnen eine fette Linie als Doppellinie und danach bewegen wir die Schildkröte einen Schritt nach rechts neben die Doppellinie.

```
to FE100
fd 100 rt 90 fd 1 rt 90
fd 100 lt 90
pu fd 1 lt 90 pd
end
```

Weil $50 \cdot 2 = 100$ ist, zeichnet das Programm

```
repeat 50 [ FE100 ]
```

das schwarze Quadrat aus Abb. 3.2 auf Seite 64.

Aufgabe 3.17

Definieren wir zuerst ein Programm zum Zeichnen eines schwarzen Quadrats mit der Seitenlänge 40.

```
to QUAD40
repeat 20 [ fd 40 rt 90 fd 1 rt 90 fd 40 lt 90 fd 1 lt 90 ]
end
```

Nun können wir mit dem folgenden Programm das Bild aus Abb. 3.3 auf Seite 65 erzeugen.

```
repeat 3 [ QUAD40 rt 90 fd 40 lt 90 ]
QUAD40
```

Aufgabe 3.21

Hier kannst du modular vorgehen und zeilen- oder spaltenweise das Bild zeichnen. Wir zeigen hier einen einfachen Weg zum Zeichnen eines 8×8-Schachfeldes der Größe 100×100. Dazu nutzen wir das Programm SCHACH4, das das 4×4-Schachfeld aus Abb. 3.4 zeichnet. Wenn wir uns das richtig überlegen, kann ein 8×8-Schachfeld wie in Abb. 3.12 aus 4×4-Schachfeldern zusammengesetzt werden.

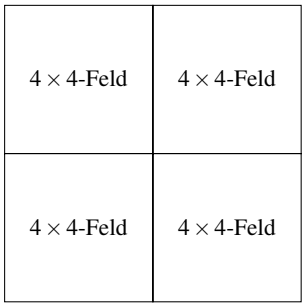


Abbildung 3.12 Eine modulare Komposition eines 8×8-Schachfelds

Das folgende Programm fängt mit der Zeichnung des 4×4-Schachfelds in der linken oberen Ecke an und zeichnet die vier Felder gegen den Uhrzeigersinn.

```
to SCHACH8
SCHACH4
SCHACH4
pu fd 400 rt 90 fd 400 lt 90 pd
SCHACH4
pu fd 800 pd
SCHACH4
end
```

Kontrollaufgabe 4
Das schwarze Dreieck in Abb. 3.9 auf Seite 71 kann man als Pyramide zeichnen. Auf dem Boden liegt eine schwarze Linie der Länge 20. Über diese Linie legen wir zentriert im Abstand von einem Schritt eine Linie der Länge 18, darüber eine Linie der Länge 16 und so weiter, bis wir am Ende eine Linie der Länge 2 zeichnen.

```
rt 90
fd 20 bk 19 lt 90 fd 1 rt 90
fd 18 bk 17 lt 90 fd 1 rt 90
fd 16 bk 15 lt 90 fd 1 rt 90
fd 14 bk 13 lt 90 fd 1 rt 90
fd 12 bk 11 lt 90 fd 1 rt 90
fd 10 bk  9 lt 90 fd 1 rt 90
fd  8 bk  7 lt 90 fd 1 rt 90
fd  6 bk  5 lt 90 fd 1 rt 90
fd  4 bk  3 lt 90 fd 1 rt 90
fd  2
```

Lektion 4

Zeichnen von Kreisen und regelmäßigen Vielecken

Im Unterschied zur vorherigen Lektion geht es in dieser Lektion nicht darum, neue Programmierkonzepte und Rechnerbefehle zu lernen. Hier wollen wir etwas mehr über die Vielecke als geometrische Bilder erfahren und dadurch entdecken, wie wir sie mit Hilfe der Schildkröte zeichnen können.

Ein regelmäßiges k-Eck hat k Ecken und k gleich lange Seiten. Wenn du ein Vieleck, zum Beispiel ein 10-Eck, mit Bleistift zeichnen möchtest, musst du zehn Linien zeichnen und nach jeder Linie „ein bisschen“ die Richtung ändern (drehen).

Wie viel muss man drehen?
Kannst du so etwas berechnen?

Wenn man ein regelmäßiges Vieleck zeichnet, dreht man mehrmals aber am Ende steht man genau an der gleichen Stelle und schaut in genau die gleiche Richtung wie am Anfang (Abb. 4.1 auf der nächsten Seite)

Das bedeutet, dass man sich unterwegs volle 360° gedreht hat. Wenn man also ein regelmäßiges 10-Eck zeichnet, hat man sich genau zehnmal gedreht und zwar immer um einen gleichgroßen Winkel. Also

$$\frac{360}{10} = 36$$

Daher muss man sich immer 36° drehen: `rt 36`.

J. Hromkovič, *Lehrbuch Informatik*, DOI 10.1007/978-3-8348-9692-6_5,

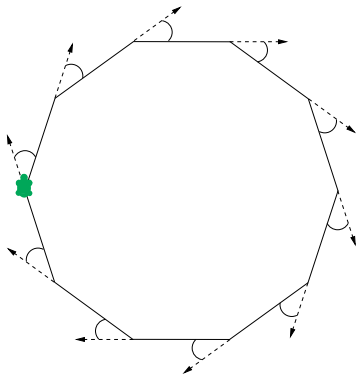


Abbildung 4.1

Probieren wir das, indem wir das folgende Programm schreiben:

```
repeat 10 [ fd 50 rt 36 ]
```

Aufgabe 4.1 Zeichne folgende regelmäßige Vielecke:

a) Ein 5-Eck mit der Seitenlänge 180

b) Ein 12-Eck mit der Seitenlänge 50

c) Ein 4-Eck mit der Seitenlänge 200

d) Ein 6-Eck mit der Seitenlänge 100

e) Ein 3-Eck mit der Seitenlänge 200

f) Ein 18-Eck mit der Seitenlänge 20

Wenn man ein 7-Eck zeichnen will, hat man das Problem, dass man 360 nicht ohne Rest durch sieben teilen kann. In diesem Fall lässt man das Resultat durch den Rechner ausrechnen, indem man 360/7 schreibt. Das Symbol „/“ bedeutet für den Rechner „teile“. Der Rechner findet dann das genaue Resultat. Somit kann man ein 7-Eck mit Seitenlänge 100 wie folgt zeichnen.

```
repeat 7 [ fd 100 rt 360/7 ]
```

Aufgabe 4.2 Zeichne folgende regelmäßige Vielecke:

a) Ein 13-Eck mit der Seitenlänge 30

b) Ein 19-Eck mit der Seitenlänge 20

Wir haben jetzt gelernt, regelmäßige Vielecke zu zeichnen, aber wie zeichnet man einen Kreis? Mit den Befehlen `fd` und `rt` kann man keine genauen Kreise zeichnen. Wie du aber sicherlich beobachtet hast, sehen Vielecke mit vielen Ecken Kreisen sehr ähnlich. Wenn wir also viele Ecken und sehr kurze Seiten nehmen, erhalten wir dadurch Kreise.

Aufgabe 4.3 Untersuche die Auswirkungen folgender Programme:

a) `repeat 360 [ fd 1 rt 1 ]`

b) `repeat 180 [ fd 3 rt 2 ]`

c) `repeat 360 [ fd 2 rt 1 ]`

d) `repeat 360 [ fd 3.5 rt 1 ]` (3.5 bedeutet dreieinhalb)

Aufgabe 4.4 Was würdest du tun, um ganz kleine Kreise zu zeichnen? Schreibe ein Programm dafür.

Aufgabe 4.5 Was würdest du tun, um große Kreise zu zeichnen? Schreibe ein Programm dafür.

Aufgabe 4.6 Nutze deine neuen Erfahrungen, um die Bilder aus Abb. 4.2 zu zeichnen.

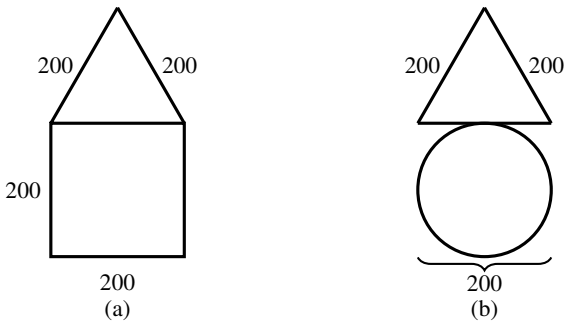


Abbildung 4.2

Aufgabe 4.7 Versuche die Halbkreise aus Abb. 4.3 zu zeichnen.

(a) (b)

Abbildung 4.3

Wir haben schon einiges gelernt und können mehrere geometrische Figuren zeichnen. Damit kann man schon sehr schöne Fantasiemuster zeichnen. Eine Idee dazu zeigen wir jetzt. Zeichne ein 7-Eck mit

```
repeat 7 [ fd 100 rt 360/7 ],
```

dann drehe die Schildkröte um 10 Grad mit

```
rt 10
```

und wiederhole

```
repeat 7 [ fd 100 rt 360/7 ].
```

Mache das ein paar Mal und schaue dir das Bild an. Wir drehen nach jedem 7-Eck immer um 10 Grad mit `rt 10`. Wenn wir wieder in die Ausgangsrichtung zurückkommen wollen, dann müssen wir diese Tätigkeit

$$\frac{360}{10} = 36$$

Mal wiederholen. Also schauen wir uns an, was das folgende Programm zeichnet:

```
repeat 36 [ repeat 7 [ fd 100 rt 360/7 ] rt 10 ].
```

Aufgabe 4.8 Zeichne ein regelmäßiges 12-Eck mit Seiten der Länge 70 und drehe es 18-mal bis du wieder an die Startposition kommst. Hinweis: Du kannst zuerst ein Programm für ein 12-Eck mit Seitenlänge 70 schreiben und ihm zum Beispiel den Namen `ECK12` geben. Dann musst du nur noch das Programm vervollständiegen:

```
repeat 18 [ ECK12 rt ( was muss hier stehen? ) ].
```

Aufgabe 4.9 Denke dir eine ähnliche Aufgabe wie in Aufgabe 4.8 aus und schreibe ein Programm dazu.

Aufgabe 4.10 Ersetze das 12-Eck aus Aufgabe 4.8 durch einen Kreis (als 360-Eck mit Seitenlänge 2) und schaue dir das gezeichnete Muster an.

Wenn man schon Fantasiemuster zeichnet, passen dazu auch Farben. Die Schildkröte kann nicht nur mit Schwarz, sondern mit einer beliebigen Farbe zeichnen. Jede Farbe ist durch eine Zahl bezeichnet. Eine Übersicht aller Farben findest du in Tabelle 4.1.

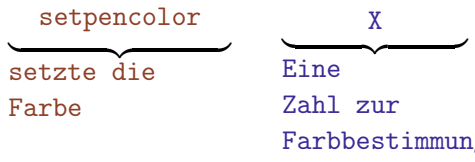
Farbnummer	Farbname	[R G B]	Farbe
0	`black`	[0 0 0]	
1	`red`	[255 0 0]	
2	`green`	[0 255 0]	
3	`yellow`	[255 255 0]	
4	`blue`	[0 0 255]	
5	`magenta`	[255 0 255]	
6	`cyan`	[0 255 255]	
7	`white`	[255 255 255]	
8	`gray`	[128 128 128]	
9	`lightgray`	[192 192 192]	
10	`darkred`	[128 0 0]	
11	`darkgreen`	[0 128 0]	
12	`darkblue`	[0 0 128]	
13	`orange`	[255 200 0]	
14	`pink`	[255 175 175]	
15	`purple`	[128 0 255]	
16	`brown`	[153 102 0]	

Tabelle 4.1 Tabelle der Farben

Hinweis für die Lehrperson Die Spalten in Tab. 4.1 entsprechen den drei Möglichkeiten in XLOGO, eine gewünschte Farbe einzustellen. Alle Möglichkeiten sind gleichwertig. Es spielt keine Rolle, welche Farbenbezeichnung man wählt. Eine allgemeine internationale Bezeichnung, das RGB-Modell, ist in der Spalte 3 dargestellt und gibt an, wie man die gewünschte Farbe aus einer Mischung der Farben Rot, Gelb und Blau erhalten kann. Auf der WEB-Seite `http://de.wikipedia.org/wiki/RGB-Farbraum` findet man eine ausführliche Erklärung. Ein Vorteil der Bezeichnung von Farben im [r,g,b]-Format ist die Tatsache, dass man hiermit beliebige Farben mischen kann, also auch solche, die in Tabelle 4.1 nicht vorkommen.

In SUPERLOGO gibt es hingegen nur zwei Möglichkeiten, Farben zu beschreiben: entweder durch eine Zahl (erste Spalte) oder durch die RGB-Zahlentriple (dritte Spalte). Dabei ist die Nummerierung der Farben in der ersten Spalte anders als bei XLOGO.

Mit dem Befehl



wechselt die Schildkröte von der aktuellen Farbe in die Farbe mit dem Wert `X`.

Damit kann man tolle Muster zeichnen, wie zum Beispiel das Muster, das durch das folgende Programm entsteht. Zuerst benennen wir zwei Programme zum Zeichnen zweier Kreise mit unterschiedlicher Größe.

```
to KREIS3
repeat 360 [ fd 3 rt 1 ]
end

to KREIS1
repeat 360 [ fd 1 rt 1 ]
end
```

Jetzt nutzen wir diese Kreise, um ähnliche Muster wie die Bisherigen zu entwerfen.

```
to MUST3
repeat 36 [ KREIS3 rt 10 ]
end
```

```
to MUST1
repeat 18 [ KREIS1 rt 20 ]
end
```

Jetzt versuchen wir es mit Farben.

```
setpencolor 2
MUST3 rt 2
setpencolor 3
MUST3 rt 2

setpencolor 4
MUST3 rt 2
setpencolor 5
MUST3 rt 2

setpencolor 6
MUST1 rt 2
setpencolor 15
MUST1 rt 2

setpencolor 8
MUST1 rt 2
setpencolor 9
MUST1 rt 2
```

Du darfst gerne die Arbeit fortsetzen und noch mehr dazu zeichnen. Oder zeichne ein Muster nach eigener Vorstellung.

Aufgabe 4.11 Nutze `MUST3`, um das entsprechende Bild mit der orangenen Farbe zu zeichnen. Verwende danach den Befehl `setpencolor 7`, um zur weißen Farbe zu wechseln. Was passiert jetzt, wenn du wieder `MUST3` ausführen lässt?

Aufgabe 4.12 Zeichne das Bild in Abb. 4.4 auf der nächsten Seite. Die Schildkröte ist am Anfang an dem gemeinsamen Punkt (in dem Schnittpunkt) der beiden Kreise.

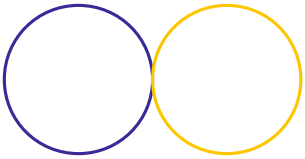


Abbildung 4.4

Zusammenfassung

Wir haben gelernt, wie man regelmäßige Vielecke zeichnen kann. Man muss Strecken mit gleicher Länge zeichnen und zwischen dem Zeichnen der Seiten immer mit

```
rt 360/Anzahl der Ecken
```

drehen. Einen Kreis zeichnet man als ein Vieleck mit sehr vielen Ecken, am besten mit 360 oder 180.

Die Schildkröte kann mit beliebigen Farben zeichnen. Um die Farbe zu wechseln, verwendet man das Befehlswort `setpencolor` und als Parameter kommt danach die Nummer der gewünschten Farbe.

Kontrollfragen

1. Wie zeichnet man regelmäßige Vielecke? Was hat die Anzahl der Ecken mit der Größe der Drehung nach dem Zeichnen einer Seite zu tun?

2. Wie berechnet man den Umfang eines Vielecks?

3. Wie zeichnet man Kreise?

4. Mit welchem Befehl kann man die Stiftfarbe der Schildkröte ändern?

5. Kann man bei der Schildkröte durch Farbänderung den Stiftmodus ändern? Wenn ja, wie und in welchen Modus?

Kontrollaufgaben

1. Zeichne folgende regelmäßige Vielecke:

   a) Ein 12-Eck mit Seitenlänge 25

   b) Ein 7-Eck mit Seitenlänge 50

   c) Ein 3-Eck mit Seitenlänge 200

   Bestimme für alle den Umfang.

2. Zeichne Kreise mit folgenden Umfängen:

   a) 360 Schritte

   b) 720 Schritte

   c) 900 Schritte

   d) 777 Schritte

3. Schreibe ein Programm zum Zeichnen des Bildes aus Abb. 4.5. Der Umfang der Kreise ist jeweils 540 Schritte.

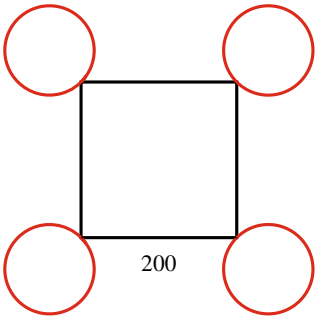


Abbildung 4.5

4. Schreibe ein Programm zum Zeichnen des Bildes aus Abb. 4.6. Die Größe der Kreise darfst du selbst wählen.

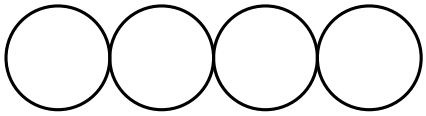


Abbildung 4.6

5. Schreibe ein Programm zum Zeichnen des Bildes aus Abb. 4.7.

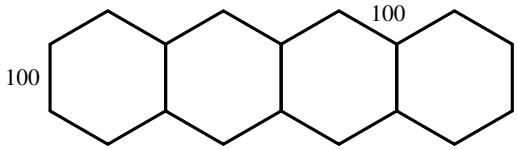


Abbildung 4.7

6. Kannst du mit Hilfe des Radiergummimodus und des Stiftmodus das Bild aus Abb. 4.7 in vier nebeneinander stehende Häuser umwandeln? Wie die Häuser aussehen sollen, ist dir überlassen. Ein Beispiel für ein Haus siehst du in Abb. 4.8.

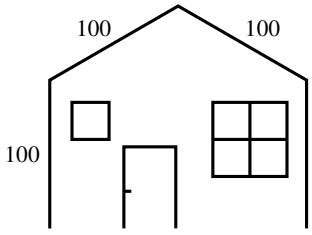


Abbildung 4.8

7. Zeichne mit Gelb das Netz aus Abb. 4.9 auf der nächsten Seite.

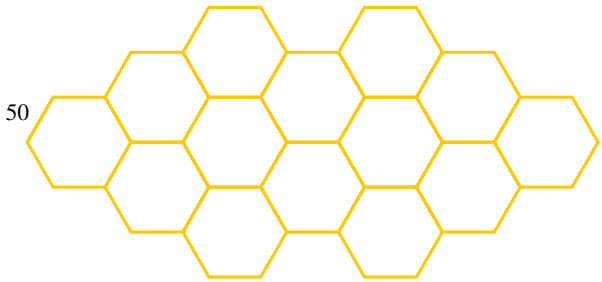


Abbildung 4.9

8. Zeichne vier Kreise wie in Abb. 4.10 mit den Umfängen 360, 540, 720 und 900.

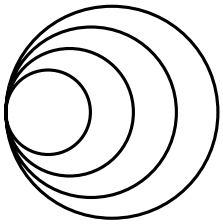


Abbildung 4.10

Lösungen zu ausgesuchten Aufgaben

Aufgabe 4.2

a) `repeat 13 [ fd 30 rt 360/13 ]`

b) `repeat 19 [ fd 20 rt 360/19 ]`

Aufgabe 4.5

Ein Kreis wird als ein Vieleck mit vielen Ecken gezeichnet. Damit ist sein Umfang immer *Anzahl der Ecken* × *Seitenlänge*. Wenn wir Kreise als regelmäßige 360-Ecke zeichnen, ist der Umfang 360× *Seitenlänge*. Somit ist der Umfang des Kreises

```
repeat 360 [ fd 1 rt 1 ]
```

$360 \times 1 = 360$ Schritte. Wenn wir den Umfang vergrößern, dann können wir die Seitenlänge vergrößern. Zum Beispiel zeichnet

```
repeat 360 [ fd 4 rt 1 ]
```

einen Kreis mit dem Umfang $360 \times 4 = 1440$.

Aufgabe 4.7
Einen Halbkreis zeichnet man, indem man statt 360-mal nur 180-mal mittels `rt 1` dreht. Wenn die Schildkröte für das Bild aus Abb. 4.3(a) am Anfang auf der linken Seite steht, reicht das folgende Programm:

```
repeat 180 [ fd 2 rt 1 ]
```

Nach dem Zeichnen steht die Schildkröte am rechten Ende des Halbkreises und schaut nach unten.

Um den Halbkreis in Abb. 4.3(b) zu zeichnen, können wir unterschiedlich vorgehen. Zum Beispiel können wir den Halbkreis in Abb. 4.3(b) als den zweiten Teil eines Kreises sehen, bei dem der Halbkreis in Abb. 4.3(a) dem ersten Teil entspricht. Wir können also den ersten Teil ablaufen, ohne ihn zu zeichnen und danach den zweiten Teil zeichnen. Das macht das folgende Programm:

```
pu
repeat 180 [ fd 2 rt 1 ]
pd
repeat 180 [ fd 2 rt 1 ]
```

Wir können auch anders vorgehen. Wir drehen die Schildkröte um und lassen sie den Halbkreis aus Abb. 4.3(b) als den ersten Teil eines Kreises zeichnen.

```
rt 180
repeat 180 [ fd 2 rt 1 ]
```

Kontrollaufgabe 2

b) `repeat 360 [ fd 2 rt 1 ]`

c) `repeat 360 [ fd 2.5 rt 1 ]`

d)
```
repeat 360 [ fd 777/360 rt 1 ]
```

Kontrollaufgabe 5
Wir zeichnen das Bild aus Abb. 4.7 auf Seite 84 von links nach rechts. Wir gehen dabei modular vor und schreiben zuerst zwei Programme. Das erste Programm zeichnet ein 6-Eck mit der Seitenlänge 100.

```
to ECK6L100
repeat 6 [ fd 100 rt 60 ]
end
```

Das zweite Programm macht die notwendige Verschiebung zur neuen Startposition, von wo aus das nächste 6-Eck gezeichnet werden kann.

```
to VERS
repeat 4 [ fd 100 rt 60 ]
rt 120
end
```

Damit sieht das Programm zum Zeichnen des Bildes aus Abb. 4.7 auf Seite 84 wie folgt aus:

```
repeat 3 [ ECK6L100 VERS ]
ECK6L100.
```

Lektion 5

Programme mit Parametern

Die Geschichte mit der „Faulheit“ der Programmierer nimmt kein Ende. In Lektion 3 haben wir aus diesem Grund gelernt, Programmen einen Namen zu geben und sie dann mit diesem Namen aufzurufen, um das gewünschte Bild vom Rechner zeichnen zu lassen. Das bedeutet, dass der Name des von uns benannten Programms zu einem Befehlsnamen wurde. Und weil wir bei diesem Befehlsnamen, wie z. B. `QUAD100`, keine Parameter verwendet haben, ist dieser Befehlsname zu einem eigenständigen Befehl geworden wie `cs` oder `pu`. Wenn wir mehrfach das gleiche Bild zeichnen wollen, z. B. `SCHW100`, dann spart uns die Verwendung des Programmnamens als neuer Befehl viel Tippen, weil wir sonst wiederholt das gleiche Programm eintippen müssten.

Es gibt aber auch Situationen, in denen uns dieses Vorgehen nicht hinreichend hilft. Nehmen wir an, wir wollen fünf Quadrate unterschiedlicher Größe zeichnen. Dabei sollen die Seitenlängen jeweils 50, 70, 90, 110 und 130 sein und es soll die Zeichnung in Abb. 5.1 auf der nächsten Seite entstehen. Stellen wir uns weiter vor, dass wir häufiger Quadrate dieser Größe zeichnen müssen.

Wir könnten so vorgehen, dass wir fünf Programme schreiben, jeweils ein Programm für die Quadratgröße 50, 70, 90, 110 und 130. Das würde dann so aussehen:

```
to QUAD50
repeat 4 [ fd 50 rt 90 ]
end

to QUAD70
repeat 4 [ fd 70 rt 90 ]
end
```

J. Hromkovič, *Lehrbuch Informatik*, DOI 10.1007/978-3-8348-9692-6_6,

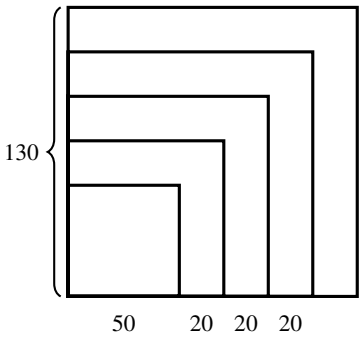


Abbildung 5.1

```
to QUAD 90
repeat 4 [ fd 90 rt 90 ]
end
```

Und so weiter. Das sieht langweilig aus, oder? Wir schreiben immer fast das gleiche Programm. Der Unterschied liegt nur in der Zahl mit gelbem Hintergrund, die der Seitenlänge des jeweiligen Quadrats entspricht. Wäre es nicht schöner und praktischer einen Befehl

```
QUADRAT X
```

mit einem Parameter `X` zu haben, der für eine Zahl `X` das Quadrat mit Seitenlänge `X` zeichnet? Das wäre nach dem Muster von `fd X`. Der Befehl `fd X` besteht aus einem Befehlswort `fd` und einem Parameter `X` und zeichnet eine Linie der Länge `X`. Warum sollte es nicht möglich sein, einen Befehl mit Parameter für das Zeichnen von Quadraten beliebiger Größe zu haben?

Und dies geht tatsächlich mit dem Konzept der **Programmparameter**, die wir meistens nur kurz als **Parameter** bezeichnen werden. Bei der Beschreibung eines Programms mit einem Parameter sucht man sich für den Parameter zuerst einen Namen. Dann schreibt man den Befehl `to`, dahinter den `Programmnamen` und dahinter den `:PARAMETERNAMEN`. Vor dem Parameternamen muss immer ein Doppelpunkt stehen. Dadurch erkennt der Rechner, dass es sich um einen Parameter handelt. Danach folgt ein fast gleiches Programm wie das, was wir ohne Parameter hatten. Nur dass man dort, wo sich die Zahlen abhängig von der Bildgröße unterscheiden (in unserem Beispiel die Zahlen mit gelbem Hintergrund), statt der Zahlen den Parameternamen (mit Doppelpunkt davor) schreibt.

Somit sieht das Programm zum Zeichnen von Quadraten in beliebiger Größe folgendermaßen aus:

Programmname Parametername

```
to    QUADRAT        :GR
repeat 4 [ fd :GR rt 90 ]
end
```

Beachte, dass `:GR` genau an der Stelle vorkommt, wo in den Programmen `QUAD50`, `QUAD70` und `QUAD90` die jeweiligen Seitenlängen 50, 70 bzw. 90 standen.

Um die Schreibweise von Programmen mit Parametern zu verstehen, erklären wir zuerst, was in einem Rechner passiert, wenn man dieses Programm eintippt. Wie wir schon wissen, wird das Programm unter dem Namen `QUADRAT` abgespeichert. Zusätzlich wird notiert, dass es einen Parameter hat. Wenn der Rechner einen Doppelpunkt liest, weiß er, dass danach der Parametername folgt. Wenn also der Rechner die erste Zeile

```
to QUADRAT :GR
```

liest, reserviert er einen Speicherplatz, genannt Register, in seinem Speicher und gibt ihm den Namen `:GR`. Das Ganze sieht aus wie in Abb. 5.2(a).

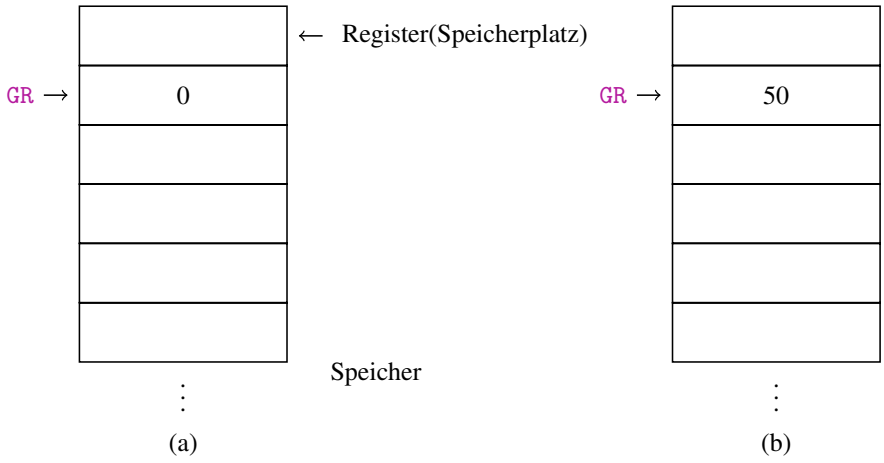


Abbildung 5.2 Schematischer Aufbau des Rechnerspeichers, der aus einer großen Anzahl von Registern besteht.

Den Speicher kann man sich als Schrank mit vielen Schubladen vorstellen. Die Schubladen sind die kleinste Einheit des Schranks und heißen **Register**. In jedem Register kann man genau eine Zahl abspeichern und jedem Register kann man einen Namen geben. Wenn man in ein Register keine Zahl gespeichert hat, liegt da automatisch eine Null, damit ist ein Register nie leer. Die Zahl, die in einem Register R gespeichert ist, bezeichnen wir als den Inhalt des Registers R. In der Abb. 5.2(a) hat der Rechner dem zweiten Register den Namen GR gegeben und weil wir noch nichts „rein gelegt“ haben, liegt dort die Zahl 0.

Wenn man jetzt den Befehl

```
QUADRAT 50
```

eintippt, weiß der Rechner, dass `QUADRAT` ein Programm mit dem Parameter `:GR` ist. Die Zahl `50` hinter dem Programmnamen nimmt er dann und legt sie in das Register (in die Schublade) mit dem Namen `:GR`. Wenn sich dort schon eine Zahl befindet, wird sie gelöscht (und damit vergessen) und durch die neue Zahl `50` ersetzt. Die Situation im Speicher nach der Umsetzung des Befehls `QUADRAT 50` ist in Abb. 5.2(b) dargestellt. Danach setzt der Rechner alle Befehle des Programms `QUADRAT` nacheinander um. Überall, wo `:GR` auftritt, geht er an den Speicher, öffnet die Schublade mit dem Namen `GR` und liest den Inhalt des Registers. Mit dem Inhalt, den er dort findet, ersetzt er im Programm die Stelle, die mit `:GR` beschriftet ist und führt den entsprechenden Befehl aus.

Wenn wir im Programm `QUADRAT :GR` alle Stellen, wo `:GR` steht, durch `50` ersetzen, erhalten wir genau das Programm `QUAD50`. Somit zeichnet der Rechner nach dem Befehl `QUADRAT 50` ein Quadrat mit Seitenlänge 50.

Aufgabe 5.1 Nachdem du das Programm `QUADRAT :GR` definiert hast, gib die Befehle

```
QUADRAT 50 QUADRAT 70 QUADRAT 90
QUADRAT 110 QUADRAT 130
```

ein und schaue ob Du dadurch das Bild aus Abb. 5.1 auf Seite 90 gezeichnet hast. Welche Zahl liegt im Register `GR` nach dem Befehl `QUADRAT 70`? Welche Zahl liegt im Register `GR` nach der Durchführung dieses Programms?

Hinweis für die Lehrperson Achten Sie bitte darauf, dass man sich für unterschiedliche Programme unterschiedliche Namen aussucht. Das Gleiche gilt auch für die Parameter.
Es ist wichtig zu betonen, dass in jedem Register immer genau eine Zahl liegt. Es können dort nicht zwei Zahlen abgespeichert werden. Wenn man durch `Befehl Zahl` eine Zahl in ein Register

legt, in dem schon eine Zahl gespeichert war, wird die alte Zahl gelöscht und durch die neue ersetzt. Es ist wie ein kleines Band, auf das man nur eine Zahl schreiben darf. Wenn man dort eine andere Zahl aufschreiben will, muss man zuerst die dort stehende Zahl ausradieren. Es ist sehr wichtig, dieses Konzept zu verstehen, da es die Grundlage zur Einführung des Konzepts der Variablen in Lektion 8 ist.

Aufgabe 5.2 Nutze das Programm `QUADRAT :GR`, um Quadrate mit Seitenlänge 50, 99, 123 und 177 zu zeichnen.

Aufgabe 5.3 Was zeichnet das folgende Programm?

```
rt 45
repeat 4 [ QUADRAT 50 QUADRAT 70 QUADRAT 90 QUADRAT 110 rt 90 ]
```

Jemand will das gleiche Bild erzeugen und fängt wie folgt an:

```
rt 45
repeat 4 [ QUADRAT 50 rt 90 ]
```

Kannst du ihm helfen, das Programm zu Ende zu schreiben?

Aufgabe 5.4 Schreibe ein Programm mit einem Parameter, das regelmäßige Sechsecke beliebiger Seitengröße zeichnet. Probiere das Programm zum Zeichnen von Sechsecken für die Seitenlänge 40, 60 und 80 aus.

Aufgabe 5.5 Schreibe ein Programm mit einem Parameter zum Zeichnen von gleichseitigen Fünfecken mit beliebiger Seitenlänge.

Beispiel 5.1 In Lektion 4 haben wir gelernt, gleichseitige Vielecke mit beliebig vielen Ecken zu zeichnen. Jetzt wollen wir ein Programm entwickeln, das Vielecke mit beliebig vielen Ecken und einer Seitenlänge von 50 zeichnet. Schauen wir uns zuerst die Beispiele der folgenden Programme an, die jeweils ein 7-Eck, ein 12-Eck und ein 18-Eck zeichnen.

```
repeat 7  [ fd 50 rt 360/ 7 ]

repeat 12 [ fd 50 rt 360/ 12 ]

repeat 18 [ fd 50 rt 360/ 18 ]
```

Wie beim Zeichnen von Quadraten unterschiedlicher Größe sind die Programme bis auf die Stellen mit gelbem Hintergrund gleich. Der einzige Unterschied zur vorherigen Aufgabe ist, dass der Parameter des Programms an zwei unterschiedlichen Stellen im Programm auftritt. Dementsprechend taucht der Parametername in der Beschreibung des Programms zweimal auf.

```
to VIELECK :ECKE
repeat :ECKE [ fd 50 rt 360/:ECKE ]
end
```

Bei dem Aufruf `VIELECK 7` wird die Zahl 7 im Register `ECKE` gespeichert. Bei der Ausführung des Programms werden die beiden Stellen im Programm mit `:ECKE` durch die Zahl 7 (dem Inhalt des Registers `ECKE`) ersetzt. □

Hinweis für die Lehrperson In den ersten Lektionen verwenden wir konsequent große Buchstaben für die Parameternamen. Beide Programmiersprachen XLOGO und SUPERLOGO erlauben auch die Verwendung von kleinen Buchstaben. Hier muss man aber vorsichtig sein. Die Programme unterscheiden nicht zwischen großen und kleinen Buchstaben. Die Parameternamen `:a` und `:A` haben somit die gleiche Bedeutung und werden nicht als zwei unterschiedliche Namen behandelt.

Aufgabe 5.6 Verwende das Programm `VIELECK` und zeichne hintereinander fünf regelmäßige Vielecke mit einer unterschiedlichen Anzahl an Ecken bei einer Seitenlänge von 50.

Aufgabe 5.7 Schreibe ein Programm, das unterschiedlich große Kreise zeichnen kann.

Aufgabe 5.8 In Lektion 3 haben wir gelernt, eine fette Linie zu zeichnen. Schreibe ein Programm zum Zeichnen einer fetten Linie (als Doppellinie) mit beliebiger Länge.

Aufgabe 5.9 Schreibe ein Programm mit einem Parameter `:X`, das beliebig große Häuser wie in Abb. 5.3 auf der nächsten Seite zeichnet.

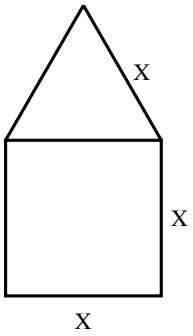


Abbildung 5.3

Schaffst du es auch, ein beliebig großes Haus wie in Abb. 5.4 mit Hilfe nur eines Parameters zeichnen zu lassen?

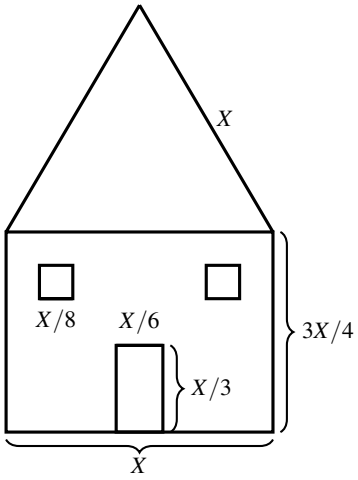


Abbildung 5.4

Beide Fenster liegen in der Entfernung $X/8$ von der seitlichen Hauswand und in der Entfernung $X/8$ vom Dachgeschoss.

Die Parameter ermöglichen es, einige Eigenschaften der Bilder frei wählbar zu lassen. Wir müssen also beim Schreiben des Programms noch nicht bestimmen, wie groß das Bild

sein soll. Wir können die Größe später bei der Ausführung wählen. Aber es geht nicht nur um die Seitenlänge oder andere Größen. Man kann auch Parameter nutzen, um die Anzahl an Wiederholungen einer Zeichnung frei wählbar zu halten. Zum Beispiel in Abb. 2.6 auf Seite 46 haben wir zehn Quadrate der Seitenlänge 20 gezeichnet. Wenn wir aber die Zahl der Quadrate, die gezeichnet werden sollen, frei wählen wollen, dann ersetzen wir im Programm aus Beispiel 2.3 (erste Lösung) die Zahl 10 durch einen Parameter `:ANZ`.

```
to LEITER :ANZ
repeat :ANZ [ repeat 4 [ fd 20 rt 90 ] rt 90 fd 20 lt 90 ]
end
```

Aufgabe 5.10 Schreibe Programme, die eine frei wählbare Anzahl von Treppenstufen wie in Abb. 2.1 auf Seite 41 und Abb. 2.2 auf Seite 42 zeichnen.

Aufgabe 5.11 Schreibe ein Programm, das eine frei wählbare Anzahl von Radzähnen (Abb. 2.5 auf Seite 46) zeichnet.

Bisher haben wir nur Programme mit einem Parameter betrachtet. Dadurch war immer nur eine Eigenschaft (Größe, Anzahl der Wiederholungen, ...) der Bilder frei wählbar. Es kann aber wünschenswert sein, zwei, drei oder mehr Parameter eines Bildes frei wählbar zu haben. Fangen wir mit einem einfachen Beispiel an.

Beispiel 5.2 Wir wollen ein Rechteck mit beliebiger Länge und Breite zeichnen. Wir wählen den Parameter `:HOR` für die Länge der horizontalen Seite und den Parameter `:VER` für die Länge der vertikalen Seite. Das Programm

```
repeat 2 [ fd 50 rt 90 fd 150 rt 90 ]
```

zeichnet ein Rechteck der Größe 50×150 und das Programm

```
repeat 2 [ fd 100 rt 90 fd 70 rt 90 ]
```

zeichnet ein Rechteck der Größe 100×70.

Wir sehen sofort, wo die frei wählbaren Größen in den Programmen liegen und so können wir direkt das Programm zum Zeichnen beliebiger Rechtecke aufschreiben.

```
to RECHT :HOR :VER
repeat 2 [ fd :VER rt 90 fd :HOR rt 90 ]
end
```

□

Aufgabe 5.12 Schreibe ein Programm zum Zeichnen eines beliebigen roten (rot ausgefüllten) Rechtecks. Dabei sollen beide Seitenlängen frei wählbar sein.

Aufgabe 5.13 Das folgende Programm zeichnet das Parallelogramm aus Abb. 5.5.

```
rt 45 fd 200 rt 45 fd 100 rt 90
rt 45 fd 200 rt 45 fd 100
```

Schreibe ein Programm mit zwei Parametern, das solche Parallelogramme mit beliebigen Seitenlängen zeichnet.

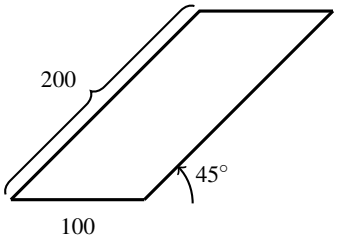


Abbildung 5.5

Beispiel 5.3 Man wünscht sich ein Programm zum Zeichnen beliebiger $X \times Y$-Felder mit wählbarer Seitengröße *GR* der einzelnen Quadrate. In Abb. 2.16 auf Seite 52 ist z. B. $GR = 20$, $X = 4$ und $Y = 10$. Wir können wie folgt vorgehen:

```
to FELD :X :Y :GR
repeat :X [ repeat :Y [ repeat 3 [ fd :GR rt 90 ] rt 90 ] lt 90
            fd :GR*:Y rt 90 fd :GR]
end
```

□

Neu für uns in diesem Programm ist der **arithmetische Ausdruck** `:GR*:Y` im Befehl `fd :GR*:Y`. Hier ersetzt der Rechner erst die Parameternamen `:GR` und `:Y` durch die in

den Registern `GR` und `Y` gespeicherten Zahlen und dann rechnet er das Produkt dieser Zahlen aus. Das Resultat dieser Multiplikation wird nirgendwo gespeichert, sondern nur zur Durchführung des Befehls `fd` verwendet.

Aufgabe 5.14 Teste das Programm aus Beispiel 5.3 und erkläre, wie es funktioniert.

Aufgabe 5.15 Schreibe ein Programm zum Zeichnen beliebiger Rechtecke in beliebiger Farbe.

Aufgabe 5.16 Schreibe ein Programm zum Zeichnen einer frei wählbaren Anzahl Türme wie in Abb. 5.6. Dabei sollen die Höhe und die Breite (oder auch andere Eigenschaften) des Turms frei wählbar sein.

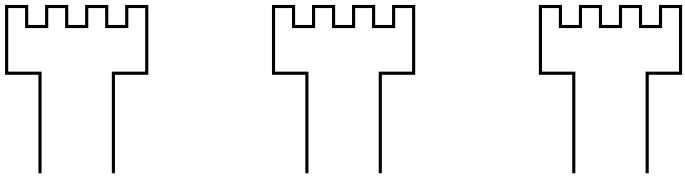


Abbildung 5.6

Aufgabe 5.17 Schreibe ein Programm zum Zeichnen von vier Kreisen wie in Abb. 5.7. Dabei soll die Größe von allen vier Kreisen frei wählbar sein und für jeden der Kreise sollte auch die Farbe frei wählbar sein.

Abbildung 5.7

Zusammenfassung

Programmparameter ermöglichen uns, Programme zu schreiben, die statt einem einzigen Bild eine ganze Klasse von Bildern zeichnen können. Zum Beispiel ein Programm mit drei Parametern reicht zum Zeichnen eines beliebigen Rechtecks in beliebiger Farbe.

Genauso wie Programme muss man Programme mit Parametern mittels eines Befehls `to` definieren. Nach `to` kommt wie üblich der Name des Programms und danach der Name des Parameters, vor dem immer ein Doppelpunkt geschrieben wird. Auch im Körper des Programms kommt bei jeder Verwendung eines Parameters vor dem Parameternamen ein Doppelpunkt. Der Doppelpunkt signalisiert dem Rechner, dass das, was danach kommt, ein Parameter ist. Wenn wir ein Programm durch seinen Namen aufrufen und dabei statt der Parameter konkrete Zahlen eingeben, werden alle Parameternamen im Programm durch die entsprechenden Zahlen ersetzt und das Programm wird mit diesen konkreten Zahlen ausgeführt.

Ein Programm kann eine beliebige Anzahl an Parametern haben. Durch die Parameter können die Größen unterschiedlicher Teile des Bildes, die Anzahl der Wiederholungen von Unterprogrammen oder die Farbe der Bilder gewählt werden.

Die Parameter dürfen arithmetische Ausdrücke als Parameter eines Befehls bilden. Zum Beispiel bei der Verwendung des Befehls `fd :X * :Y - :Z` ersetzt der Rechner die Parameternamen `:X`, `:Y` und `:Z` durch die entsprechenden Zahlen, rechnet das Resultat von $:X \cdot :Y - :Z$ aus und geht dem Resultat entsprechend viele Schritte mit der Schildkröte vorwärts.

Kontrollfragen

1. Welches Zeichen muss immer vor den Namen eines Parameters geschrieben werden?

2. Wie unterscheidet sich die Definition eines einfachen Programms von der Definition eines Programms mit Parametern? Beide starten mit `to` und enden mit `end`.

3. Was sind die Vorteile von Programmen mit Parametern? Was können Programme mit Parametern, was die Programme ohne Parameter nicht können?

4. Was passiert im Speicher des Rechners, nachdem er den Befehl `to NAME :PARAMETER` gelesen hat? Was passiert im Speicher beim Aufruf `NAME 7`?

5. Was kann der Rechner in einem Register speichern? Was passiert, wenn man in einem Register eine Zahl speichern will, es aber nicht leer ist, weil dort früher schon eine Zahl gespeichert worden ist?

6. Wie setzt der Rechner einen Befehl mit Parameter (wie z. B. `fd`, `rt` oder `repeat`) um, wenn statt einer Zahl ein arithmetischer Ausdruck als Parameter dahintersteht?

Kontrollaufgaben

1. Schreibe ein Programm zum Zeichnen der Bilder in Abb. 7 auf Seite 34, bei dem die Größe des Bildes frei wählbar ist.

2. Schreibe ein Programm mit Parametern, das ein $2i \times 2i$-Schachfeld für eine beliebige positive ganze Zahl i zeichnen kann. Dabei soll die Größe der einzelnen Quadrate 50×50 sein.

3. Schreibe ein Programm mit vier Parametern zum Zeichnen eines Baums wie in Abb. 5.8. Die Größe der drei Paare von Zweigen sowie die Farbe des Baums sollen frei wählbar sein.

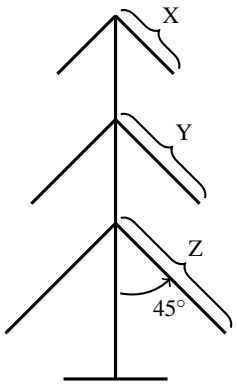


Abbildung 5.8

4. Schreibe ein Programm mit zwei Parametern `:P1` und `:P2`. Das Programm soll ein Rechteck der Länge $P1 \times P2$ und der Breite $2 \times P1$ zeichnen.

5. Schreibe ein Programm `QQQ` mit den drei Parametern `:ANZAHL`, `:LANG` und `:WINK`, das `:ANZAHL` Quadrate der Seitenlänge `:LANG` zeichnet und sich zwischen der Zeichnung zweier Quadrate immer um `:WINK` Grad nach rechts dreht. Teste das Programm mit dem Aufruf `QQQ 23 100 10`.

6. Erweitere das Programm aus Kontrollaufgabe 5 um einen weiteren Parameter `:VOR`. Nach der Drehung um `:WINK` viele Grade gehe `:VOR` viele Schritte mit der Schildkröte nach vorne. Teste das entstandene Programm mit dem Aufruf `QQQ 23 100 10 25`.

Lösungen zu ausgesuchten Aufgaben

Aufgabe 5.4
Das Programm

```
repeat 6 [ fd 50 rt 60 ]
```

zum Zeichnen eines regelmäßigen 6-Ecks mit Seitenlänge 50 kennen wir schon. Jetzt ersetzen wir die konkrete Seitenlänge 50 durch den Parameter `:L` für die wählbare Seitenlänge und erhalten das folgende Programm:

```
to SECHS :L
repeat 6 [ fd :L rt 60 ]
end
```

Aufgabe 5.7
Wir zeichnen die Kreise als regelmäßige Vielecke mit vielen Ecken. Der Umfang des Kreises ist dann das Produkt der Anzahl der Ecken und der Seitenlänge. Wenn wir verabreden, dass wir die Kreise als 360-Ecke zeichnen, wird die Größe (im Sinne des Umfangs) nur durch die frei wählbare Seitenlänge bestimmt. Also reicht uns ein Parameter `:LA`.

```
to KREISE :LA
repeat 360 [ fd :LA rt 1 ]
end
```

Hier empfehlen wir, beim Aufruf des Befehls `KREISE` auch Kommazahlen (`KREISE 1.5`) oder Brüche (`KREISE 7/3`) zu verwenden, um eine größere Vielfalt zu erhalten. Beachte, dass schon `KREISE 3` einen zu großen Kreis zeichnet, der nicht mehr auf unseren Bildschirm passt.

Aufgabe 5.13
Bezeichnen wir die Seitenlängen eines Parallelogramms mit A und B.

```
to PARAL :A :B
rt 45 fd :A rt 45 fd :B rt 90
rt 45 fd :A rt 45 fd :B
end
```

Aufgabe 5.17
Wir brauchen vier Parameter :G1, :G2, :G3 und :G4 für die Größe der Kreise und vier Parameter :F1, :F2, :F3 und :F4 für eine freie Wählbarkeit der Farben.

```
to KREISE4 :G1 :G2 :G3 :G4 :F1 :F2 :F3 :F4
setpencolor :F1
repeat 360 [ fd :G1 rt 1 ]
setpencolor :F2
repeat 360 [ fd :G2 rt 1 ]
setpencolor :F3
repeat 360 [ fd :G3 rt 1 ]
setpencolor :F4
repeat 360 [ fd :G4 rt 1 ]
end
```

Lektion 6

Übergabe von Parameterwerten an Unterprogramme

In Lektion 5 haben wir Programme mit Parametern kennengelernt und uns von der Nützlichkeit überzeugt. In Lektion 3 haben wir das Konzept des modularen Programmentwurfs behandelt, bei dem wir Programme schreiben, benennen und dann als Bausteine (oder als Befehle) für den Entwurf von komplexeren Programmen verwenden.

Man kann zum Beispiel das Programm `QUAD100` zum Zeichnen eines 100×100-Quadrats verwenden, um das folgende Programm `MUS1` zu schreiben.

```
to MUS1
repeat 20 [ QUAD100 fd 30 rt 10 ]
end
```

Das Programm `QUAD100` nennen wir Unterprogramm des Programms `MUS1`. Das Programm `MUS1` nennen wir in diesem Fall auch Hauptprogramm.

Aufgabe 6.1 Tippe das Programm oben ab und lasse die Schildkröte danach das Muster zeichnen. Wie sieht das Bild aus, wenn man statt `fd 30` im Programm den Befehl `fd 80` oder `fd 50` verwendet?

Der modulare Entwurf von Programmen ist für uns wichtig und wir wollen es auch gerne für Programme mit Parametern verwenden. Die Zielsetzung dieser Lektion ist zu lernen, wie man Programme mit Parametern als Unterprogramme beim modularen Entwurf verwenden kann. Nehmen wir an, wir wollen das Programm `MUS1` so ändern,

J. Hromkovič, *Lehrbuch Informatik*, DOI 10.1007/978-3-8348-9692-6_7,

dass die Quadratgröße frei wählbar wird. Das bedeutet, dass wir statt des Programms QUAD100 das Programm QUADRAT :GR mit dem Parameter :GR verwenden wollen. Wenn ein Unterprogramm einen Parameter hat, muss auch sein Hauptprogramm einen Parameter besitzen, um den gewünschten Wert des Parameters beim Aufruf des Hauptprogramms angeben zu können. Somit sieht unser Programm wie folgt aus:

```
to MUS2 :GR
repeat 20 [ QUADRAT :GR fd 30 rt 10 ]
end
```

Wenn man jetzt den Befehl

```
MUS2 50
```

eintippt, wird im Register GR die Zahl 50 gespeichert und überall im Programm, wo :GR vorkommt, wird die Zahl 50 dafür eingesetzt. Somit kommt es bei der Ausführung des Programms zu Aufrufen von QUADRAT 50. Damit werden in der Schleife 20-mal Quadrate der Seitengröße 50 gezeichnet.

Aufgabe 6.2 Teste das Programm MUS2 für unterschiedliche Parametergrößen.

Wenn man jetzt auch die Entscheidung trifft, dass auch die Zahl hinter dem Befehl fd (der Parameterwert des Befehls fd) frei wählbar sein soll, kann das Programm wie folgt aussehen.

```
to MUS3 :GR :A
repeat 20 [ QUADRAT :GR fd :A rt 10 ]
end
```

Aufgabe 6.3 Erweitere MUS3 zu einem Programm MUS4, indem du auch die Anzahl der Wiederholungen des Befehls repeat frei wählbar machst.

Aufgabe 6.4 In Beispiel 5.2 haben wir das Programm RECHT :HOR :VER entwickelt, um Rechtecke mit beliebiger Seitenlänge zeichnen zu können. Ersetze in dem Programmen MUS1, MUS2 und MUS3 das Unterprogramm QUADRAT durch das Unterprogramm RECHT mit zwei Parametern. Wie viele Parameter haben dann die Hauptprogramme MUS2 und MUS3?

Beispiel 6.1 In diesem Beispiel wollen wir lernen, Blumen zu zeichnen und zwar nicht nur Blumen, die aus Kreisen bestehen. Deswegen fangen wir damit an zu lernen, wie man

Blätter zeichnen kann, die beliebig schmal sein dürfen. Ein Blatt wie in Abb. 6.1 kann man als zwei zusammengeklebte Teilkreise *A* und *B* ansehen.

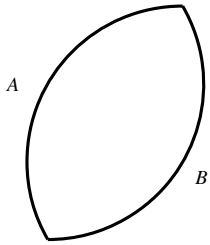


Abbildung 6.1

Einen Teilkreis kann man zum Beispiel mit folgendem Programm zeichnen:

```
repeat 120 [ fd 3 rt 1 ]
end
```

Probiere es aus.

Wir sehen, dass dieses Programm sehr ähnlich dem Programm für den Kreis ist. Anstatt 360 kleine Schritte mit jeweils 1° Drehung machen wir nur 120 kleine Schritte [`fd 3 rt 1`] und zeichnen dadurch nur ein Drittel des Kreises $\left(\frac{360}{120} = 3\right)$. Jetzt ist die Frage, wie viel man die Schildkröte drehen muss, bevor man den Teilkreis *B* für die untere Seite des Blattes zeichnet. Schauen wir uns Abb. 6.2 an.

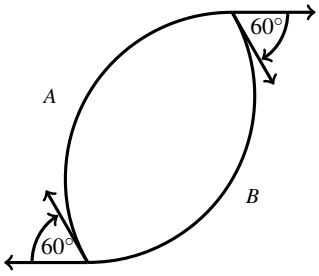


Abbildung 6.2

Wenn wir am Ende die ursprüngliche Position erreichen wollen, müssen wir die Schildkröte insgesamt, wie immer, um 360° drehen. Im Teil *A* drehen wir sie um 120° und im

Teil *B* um weitere 120°.

$$360° - 120° - 120° = 120°$$

Es bleiben also noch 120° übrig, die man gleichmäßig auf die zwei Drehungen an den Spitzen des Blattes verteilen muss.

$$\frac{120°}{2} = 60°$$

Damit erhalten wir folgendes Programm:

```
repeat 120 [ fd 3 rt 1 ]
rt 60
repeat 120 [ fd 3 rt 1 ]
rt 60
```

Oder noch einfacher:

```
repeat 2 [repeat 120 [ fd 3 rt 1 ] rt 60 ]
```

Probiere es aus.

Jetzt kann man sich wünschen, schmalere Blätter (die Teile A und B sind kürzer) oder breitere Blätter (die Teile A und B sind länger) zu zeichnen. Dazu kann man wieder einen Parameter verwenden. Nennen wir den Parameter zum Beispiel `:LANG` (wie Länge). Dann berechnen wir die Drehung an der Spitze des Blattes wie folgt:
Bevor man den Teil B des Blattes zeichnet, soll die Hälfte der ganzen Drehung, das heißt $\frac{360°}{2} = 180°$, gemacht werden. Also ist die Drehung an der Spitze des Blattes

$$180° - \texttt{:LANG}.$$

Damit können wir das folgende Programm aufschreiben:

```
to BLATT1 :LANG
repeat 2 [repeat :LANG [ fd 3 rt 1 ] rt 180-:LANG ]
end
```

□

Aufgabe 6.5 Gib das Programm BLATT1 aus dem Beispiel 6.1 ein und schreibe dann:

```
BLATT1 20 BLATT1 40 BLATT1 60 BLATT1 80 BLATT1 100
```

und beobachte die Auswirkung.

Das Programm BLATT1 ermöglicht uns, die Blätter durch zwei Teilkreise eines Kreises mit Umfang $3 \cdot 360$ zu zeichnen. Das kommt dadurch, dass der zu Grunde liegende Kreis durch das Programm

```
repeat 360 [ fd 3 rt 1 ]
```

gezeichnet werden würde. Wir wollen nun die Größe des Kreises, aus dem wir Teile ausschneiden, auch frei wählbar machen. Deswegen erweitern wir BLATT1 zu dem Programm BLATT, mit dem wir beliebig große und beliebig dicke (schmale) Blätter zeichnen können.

```
to BLATT :LANG :GROSS
repeat 2 [repeat :LANG [ fd :GROSS rt 1 ] rt 180-:LANG ]
end
```

Aufgabe 6.6 Wie sehen Bilder aus, in denen man mehrfach das Programm BLATT mit einem festen Parameterwert für :LANG, aber wechselnden Parameterwerten für :GROSS, aufruft? Was ändert sich, wenn der Wert für :GROSS fest ist und nur der Wert für :LANG geändert wird?

Aufgabe 6.7 Kannst du das Programm BLATT verwenden, um Kreise beliebiger Größe zu zeichnen?

Jetzt verwenden wir das Programm BLATT als Unterprogramm zum Zeichnen von Blumen, die wie Abb. 6.3 auf der nächsten Seite aussehen.

Dabei gehen wir genauso vor, wie bei der Zeichnung der Blume mittels Kreisen.

```
to BLUMEN :ANZAHL :LANG :GROSS
repeat :ANZAHL [BLATT :LANG :GROSS rt 360/:ANZAHL ]
end
```

Damit ist jetzt BLUMEN ein Hauptprogramm mit dem Unterprogramm BLATT. Das Programm BLUMEN hat drei Parameter und das Unterprogramm BLATT hat zwei Parameter.

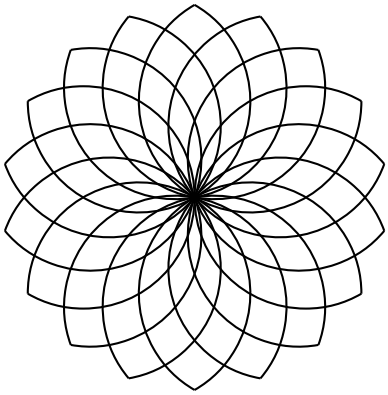


Abbildung 6.3

Jetzt wollen wir noch etwas kompliziertere zweifarbige Blumen zeichnen, indem wir das Programm BLUMEN als Unterprogramm verwenden. Die Farben sollen dabei frei wählbar sein.

```
to BLU1 :ANZAHL :LANG :GROSS :F1 :F2
setpencolor :F1
BLUMEN :ANZAHL :LANG :GROSS
rt 5
setpencolor :F2
BLUMEN :ANZAHL :LANG :GROSS
end
```

Wir beobachten hier, wie relativ der Begriff Hauptprogramm ist. Das Programm BLUMEN ist ein Hauptprogramm in Beziehung zum Unterprogramm BLATT. Andererseits ist das Programm BLUMEN ein Unterprogramm des Programms BLU1. Hier ist damit BLU1 das Hauptprogramm bezüglich der Programme BLUMEN und BLATT, obwohl BLATT nicht explizit in BLU1 aufgerufen wird, sondern unter BLUMEN versteckt bleibt. Wenn man BLU1 als ein Modul für ein noch komplexeres Programm P verwenden würde, würde BLU1 ein Unterprogramm in Bezug zum neuen Programm P werden. Alle Unterprogramme von BLU1 würden damit auch automatisch Unterprogramme von P werden.

Aufgabe 6.8 Lass dich von dem Programm BLUMEN inspirieren und zeichne eigene Muster.

Aufgabe 6.9 Erweitere das Programm BLUMEN, indem du am Ende des Programms noch einmal die Farbe änderst und das Programm MUS3 einfügst. Wie viele Parameter hat jetzt das Programm?

Aufgabe 6.10 Im Beispiel 5.3 haben wir ein Programm FELD mit drei Parametern zum Zeichnen beliebiger $X \times Y$-Felder mit wählbarer Quadratgröße :GR entwickelt. Dieses Programm FELD hat keine Unterprogramme. Deine Aufgabe ist jetzt, ein Programm mit gleicher Wirkung und den gleichen drei Parametern modular zu entwickeln. Als Basis soll das Unterprogramm QUADRAT :GR dienen. Verwende das Programm QUADRAT, um ein Programm ZEILE :GR :Y zum Zeichnen von $1 \times Y$-Felder für ein frei wählbares Y zu bauen. Verwende danach das Programm ZEILE als Baustein, um das Programm FELD modular aufzubauen.

Aufgabe 6.11 Entwickle ein Programm zum Zeichnen von $2i \times 2i$-Schachfeldern, wobei das i frei wählbar ist. Dabei soll die Farbe sowie die Größe der Quadrate (einzelne Felder) frei wählbar sein. Dein Entwurf muss modular sein. Fange mit fetten Linien beliebiger Länge an und mache mit ausgefüllten Quadraten beliebiger Größe weiter.

Hinweis für die Lehrperson Den Rest dieser Lektion empfehlen wir nur für Schüler ab dem sechsten Schuljahr.

Bisher haben wir immer darauf geachtet, dass das Hauptprogramm die gleichen Parameter enthält, die seine Unterprogramme verwenden. Zum Beispiel im Programm

```
to BLUMEN :ANZAHL :LANG :GROSS
repeat :ANZAHL [ BLATT :LANG :GROSS rt 360/:LANG ]
end
```

verwenden wir das Unterprogramm BLATT mit den Parametern :LANG und :GROSS. Deswegen haben wir beide Parameter auch als Parameter des Hauptprogramms BLUMEN aufgenommen. Der Vorteil dieses Vorgehens war, dass wir anschaulich beobachten konnten, wie beim Aufruf

```
BLUMEN 12 130 2
```

den drei Parametern die Werte 12, 130 und 2, wie in Abb. 6.4 auf der nächsten Seite, zugeordnet wurden. Damit war auch sofort klar, dass die Parameter :LANG und :GROSS des Unterprogramms BLATT die Werte 130 und 2 erhielten und somit wurde bei jedem Aufruf des Teilprogramms BLATT der Befehl BLATT 130 2 ausgeführt.

Stellen wir uns jetzt vor, dass wir eine Blume zeichnen wollen, in der unterschiedliche Blätter vorkommen. Auf die Art und Weise, wie wir bisher gearbeitet haben, wäre es nicht möglich, weil wir den Parametern :LANG und :GROSS gleich am Anfang durch

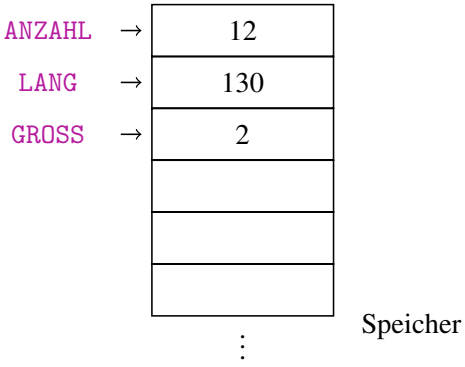


Abbildung 6.4

den Aufruf BLUMEN a b c feste Werte b und c zuordnen und dann jede Anwendung des Unterprogramms BLATT nur mit den Werten b und c als BLATT b c arbeiten muss.

Deswegen besteht die Möglichkeit, die Parameter des Hauptprogramms anders als in den Unterprogrammen zu benennen und dann mit den neuen Parameternamen die Unterprogramme aufzurufen. In unserem Beispiel kann es wie folgt aussehen.

```
to BLUMEN3 :ANZAHL :L1 :L2 :L3 :G1 :G2 :G3
repeat :ANZAHL [BLATT :L1 :G1 rt 360/:ANZAHL ]
repeat :ANZAHL [BLATT :L2 :G2 rt 360/:ANZAHL ]
repeat :ANZAHL [BLATT :L3 :G3 rt 360/:ANZAHL ]
end
```

Aufgabe 6.12 Tippe das Programm BLUMEN3 und teste es mit den Aufrufen BLUMEN3 12 100 100 100 1 2 3 und BLUMEN3 18 80 100 120 1 1.5 2.5.

Um zu verstehen, wie das Programm funktioniert, müssen wir genau beobachten, wie der Rechner bei der Ausführung des Programms mit seinen Speicherinhalten umgeht. Nach der Definition des Hauptprogramms

```
to BLUMEN3 :ANZAHL :L1 :L2 :L3 :G1 :G2 :G3
```

werden für alle sieben Parameter des Programms BLUMEN Register reserviert (Abb. 6.5(a)). Das Unterprogramm BLATT wurde schon vorher definiert und deswegen gibt es im Speicher schon die Register mit den Namen LANG und GROSS, die den Parametern von

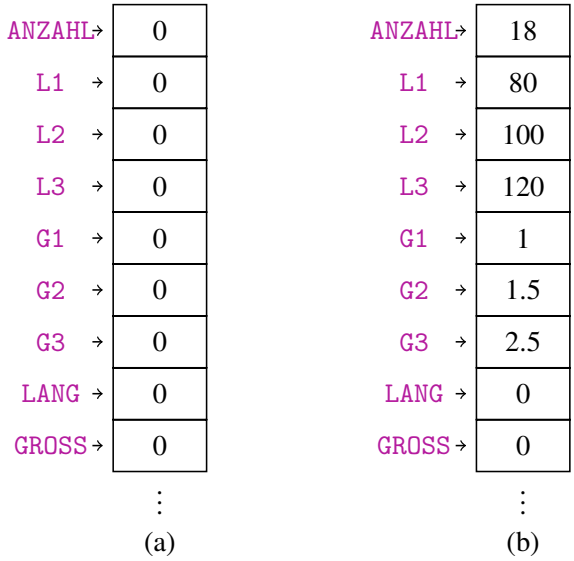


Abbildung 6.5

BLATT entsprechen (Abb. 6.5(a)). Alle Register beinhalten die Zahl 0, weil die Programme zwar definiert, aber noch nicht mit konkreten Werten aufgerufen worden sind.

Nach dem Aufruf

```
BLUMEN3 18 80 100 120 1 1.5 2.5
```

erhalten die sieben Parameter des Hauptprogramms BLUMEN3 ihre Werte wie in Abb. 6.5(b) dargestellt. Die Parameternamen :LANG und :GROSS des Unterprogramms BLATT unterscheiden sich von den Parameternamen des Hauptprogramms BLUMEN3 und deswegen wird der Inhalt der entsprechenden Register LANG und GROSS nicht geändert (Abb. 6.5(b)).

Bei der Ausführung der ersten Zeile

```
repeat :ANZAHL [BLATT :L1 :G1 rt 360/:ANZAHL ]
```

des Hauptprogramms werden die Parameterwerte eingesetzt und somit entsteht die Anweisung

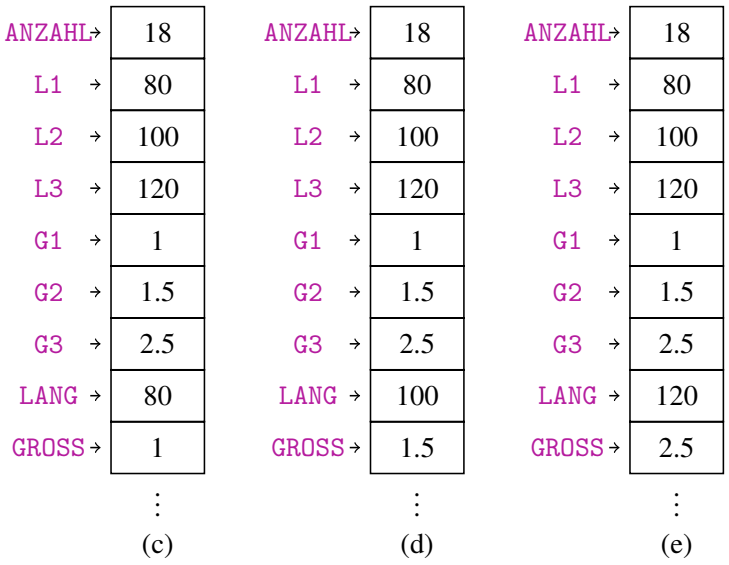


Abbildung 6.6

```
repeat 18 [BLATT 80 1 rt 360/18 ].
```

Jetzt kommt der wichtigste Punkt. Der Rechner weiß, dass der erste Parameter von `BLATT` `:LANG` und der zweite Parameter `:GROSS` heißt. Er ordnet nach dem Aufruf `BLATT 80 1` die Zahl `80` dem Parameter `:LANG` und die Zahl `1` dem Parameter `:GROSS` zu. Damit wird der Inhalt des Registers `LANG` auf `80` und der Inhalt des Registers `GROSS` auf `1` gesetzt (Abb. 6.6(c)). Es wird also 18-mal `BLATT 80 1` gefolgt von einer Drehung `rt 360/18` durchgeführt.

Bei der Ausführung der zweiten Zeile

```
repeat :ANZAHL [BLATT :L2 :G2 rt 360/:ANZAHL ]
```

des Hauptprogramms werden die Inhalte der Register `ANZAHL`, `L2` und `L3` (s. Abb. 6.6(c)) als entsprechende Parameterwerte eingesetzt und wir erhalten die Anweisung

```
repeat 18 [BLATT 100 1.5 360/18 ].
```

Damit wird `100` im Register `LANG` und `1.5` im Register `GROSS` gespeichert (s. Abb. 6.6(d)).

Diese Werte ändern sich erst bei der Ausführung der dritten Zeile

```
repeat :ANZAHL [BLATT :L3 :G3 rt 360/:ANZAHL ].
```

Hier werden die gespeicherten Werte für `:ANZAHL`, `:L3` und `:G3` eingesetzt und damit die Anweisung

```
repeat 18 [BLATT 120 2.5 360/18 ]
```

realisiert. Dadurch wird dann `120` im Register `LANG` und `2.5` im Register `GROSS` abgelegt (s. Abb. 6.6(e))

Wir können die ganze Entwicklung des Speicherinhaltes in einer Tabelle wie in Tab. 6.1 anschaulich darstellen. Die 0-te Spalte entspricht dabei der Situation nach dem Aufruf des Hauptprogramms mit konkreten Parametern. Die i-te Spalte entspricht der Speicherbelegung nach der Durchführung der i-ten Zeile des Hauptprogramms. Die Zeilen entsprechen den einzelnen Registern.

	0	1	2	3
ANZAHL	18	18	18	18
L1	80	80	80	80
L2	100	100	100	100
L3	120	120	120	120
G1	1	1	1	1
G2	1.5	1.5	1.5	1.5
G3	2.5	2.5	2.5	2.5
LANG	0	80	100	120
GROSS	0	1	1.5	2.5

Tabelle 6.1

Aufgabe 6.13 Zeichne die Entwicklung der Speicherinhalte wie in Tab. 6.1 für die Ausführung der folgenden Aufrufe des Programms `BLUMEN3`:

a) `BLUMEN3 12 100 100 100 1 2 3`

b) `BLUMEN3 15 60 80 100 2 2 2`

c) `BLUMEN3 24 90 100 110 0.5 1 2.5`

Lass den Rechner diese Aufrufe von `BLUMEN3` durchführen.

Aufgabe 6.14 Betrachte die folgenden Programme:

```
to QUADRAT :GR
repeat 4 [ fd :GR rt 90 ]
end

to PYR :Q1 :Q2 :Q3 :Q4
repeat 4 [ fd :Q1 rt 90 ]
repeat 4 [ fd :Q2 rt 90 ]
repeat 4 [ fd :Q3 rt 90 ]
repeat 4 [ fd :Q4 rt 90 ]
end
```

Führe den Aufruf

```
PYR 50 75 100 125
```

durch. Zeichne wie in Tab. 6.1 die Änderungen der Registerinhalte für die Parameter `:GR`, `:Q1`, `:Q2`, `:Q3` und `:Q4` ein.

Aufgabe 6.15 Schreibe ein Programm `QU4` für das Bild in Abb. 6.7. `QUADRAT :GR` soll als

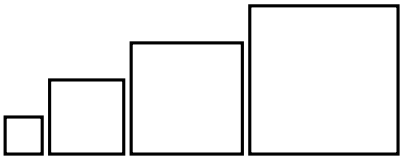


Abbildung 6.7

Teilprogramm verwenden werden und die Größe von allen vier Quadraten soll frei wählbar sein. Beschreibe dann die Entwicklung des Speicherinhaltes beim Aufruf `QU4 50 100 150 200`

Aufgabe 6.16 Schreibe ein Programm `QUG` zum Zeichnen von gefüllten (z. B. schwarzen) Rechtecken mit beliebiger Größe. Nutze dieses Programm als Teilprogramm zum Zeichnen von drei gefüllten, nebeneinanderliegenden Rechtecken beliebiger Größe, wie z. B. in Abb. 6.8 auf der nächsten Seite.

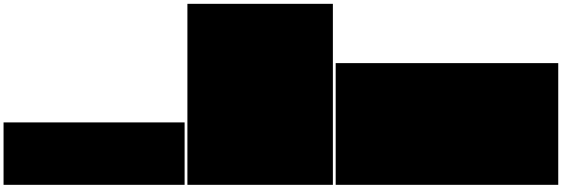


Abbildung 6.8

Aufgabe 6.17 Definiere ein Programm mit fünf Parametern zum Zeichnen von fünf regulären Vielecken, jeweils aus der gleichen Startposition, mit einer jeweiligen Seitengröße von 50. Die Anzahl der Ecken soll bei allen fünf wählbar sein, und die Zeichnung muss das Unterprogramm

```
to VIELECK :ECKE
repeat :ECKE [ fd 50 rt 360/:ECKE ]
end
```

fünfmal verwenden.

Alle Parameter, die im Hauptprogramm nach `to` und dem Programmnamen genannt werden, nennen wir **globale Parameter**. Im Hauptprogramm `BLUMEN3` sind `:ANZAHL`, `:L1`, `:L2`, `:L3`, `:G1`, `:G2` und `:G3` die globalen Parameter. Mit dem Aufruf des Programms `BLUMEN3` für konkrete Werte erhalten alle diese Parameter ihre Werte und werden zur Laufzeit des Programms auch nicht mehr geändert. Wir sehen in Tab. 6.1 gut, dass sich die Inhalte der Register `ANZAHL`, `L1`, `L2`, `L3`, `G1`, `G2` und `G3` nicht ändern. Die Parameter, die durch die Unterprogramme definiert sind, nennen wir **lokale Parameter** des Hauptprogramms. Damit sind `:LANG` und `:GROSS` lokale Parameter des Hauptprogramms `BLUMEN3`. Auf der anderen Seite sind `:LANG` und `:GROSS` die globalen Parameter des Programms `BLATT`. Während der Ausführung des Unterprogramms `BLATT` ändern sich die Werte von `:LANG` und `:GROSS` nicht. Weil aber bei der Ausführung des Hauptprogramms `BLUME3` das Teilprogramm `BLATT` mehrmals mit jeweils unterschiedlichen Parametern aufgerufen werden darf, können sich die lokalen Parameter im Laufe des Hauptprogramms mehrmals ändern.

Aufgabe 6.18 Bestimme für alle in den Aufgaben 6.14, 6.15, 6.16 und 6.17 entwickelten Hauptprogramme, welche Parameter global und welche lokal sind.

Hinweis für die Lehrperson Am Anfang dieser Lektion haben wir die gleichen Parameternamen im Hauptprogramm wie im Unterprogramm verwendet. Damit waren diese Parameter sowohl

global als auch lokal. Was das Ganze bei der Durchführung des Programms bedeutet, werden wir erst in späteren Lektionen genauer erklären.

Wir können die lokalen Parameter geschickt benutzen, indem wir sie abwechselnd zu unterschiedlichen Zwecken verwenden. In Lektion 5 haben wir das Programm RECHT mit den Parametern :HOR und :VER definiert, das ein Rechteck der Größe *HOR* × *VER* zeichnet. Jetzt betrachten wir die Aufgabe, das Bild aus Abb. 6.9 zu zeichnen.

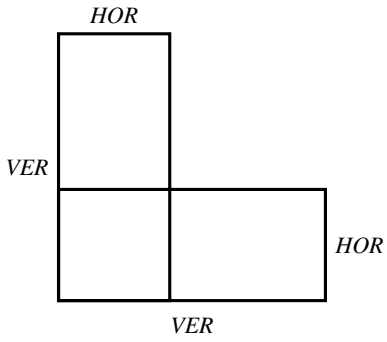


Abbildung 6.9

Das Bild kann man durch eine zweifache Anwendung von RECHT zeichnen, indem man bei der zweiten Anwendung die Rolle der Parameter vertauscht. Das Programm kann wie folgt aussehen:

```
to REC2 :A :B
RECHT :A :B
RECHT :B :A
end
```

Aufgabe 6.19 Teste das Programm für $A = 100$ und $B = 200$. Zeichne eine Tabelle (wie Tab. 6.1 auf Seite 113), die die Übergabe der Parameterwerte dokumentiert.

Aufgabe 6.20 Schreibe ein Programm zum Zeichnen rot ausgefüllter Rechtecke von frei wählbarer Größe *HOR* × *VER*. Nutze dieses Programm, um Kreuze wie in Abb. 6.10 auf der nächsten Seite mit wählbarer Größe zu zeichnen.

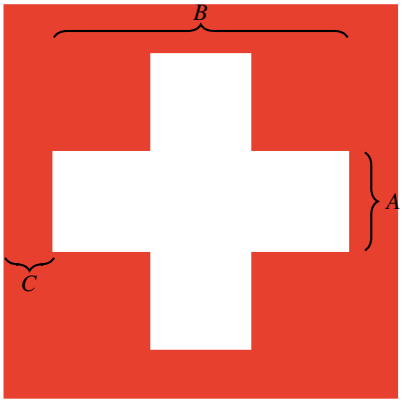


Abbildung 6.10

Zusammenfassung

Programme mit Parametern können auch Unterprogramme mit Parametern beinhalten. Dabei unterscheiden wir zwei Möglichkeiten:

1. Die Parameternamen des Unterprogramms werden auch in der Definition des Hauptprogramms verwendet. Bei einem Aufruf des Hauptprogramms mit konkreten Zahlen als Parameterwerten werden dadurch automatisch die Werte an die Unterprogramme übergeben. In diesem Fall wird kein Wert eines Parameters während der Ausführung des Hauptprogramms geändert. Dies bedeutet, dass die nach Aufruf des Hauptprogramms im Register gespeicherten Werte während der Ausführung des Hauptprogramms unverändert bleiben.

2. Einige Parameternamen von Unterprogrammen werden nicht in der Definition des Hauptprogramms verwendet. Wenn so etwas vorkommt, nennen wir die entsprechenden Parameter der Unterprogramme lokale Parameter. Die in der ersten Zeile des Hauptprogramms definierten Parameter nennen wir globale Parameter. Durch den Aufruf des Hauptprogramms mit konkreten Zahlen werden die Werte der lokalen Parameter nicht bestimmt. Erst beim Aufruf eines Unterprogramms, weist das Hauptprogramm den Parametern des Unterprogramms konkrete Werte zu. Die lokalen Parameter erhalten die Werte der globalen Parameter, die beim Aufruf

an den entsprechenden Positionen der lokalen Parameter stehen. Wenn wir das Unterprogramm mehrmals mit unterschiedlichen globalen Parametern aufrufen, ändern sich die Werte der lokalen Parameter des Hauptprogramms immer entsprechend des neuen Aufrufs. Auf diese Weise kann man ein Unterprogramm mehrmals zum Zeichnen unterschiedlich großer Objekte nutzen.

Kontrollfragen

1. Was verstehen wir unter einem modularen Entwurf?

2. Kann ein Programm gleichzeitig ein Hauptprogramm und ein Unterprogramm sein?

3. Was sind globale Parameter eines Programms und was sind lokale Parameter eines Programms? Welche Programme haben lokale Parameter?

4. Können sich die Werte der globalen Parameter während der Ausführung des Hauptprogramms ändern?

5. Wie kann es sein, dass sich die Werte der lokalen Parameter bei der Ausführung des Hauptprogramms ändern? Erkläre es an einem Beispiel.

6. Können sich die Werte der lokalen Parameter während der Ausführung des entsprechenden Unterprogramms (in dem sie definiert sind) ändern?

Kontrollaufgaben

1. Entwickle ein Programm zum Zeichnen der Bilder aus Abb. 6.11(a) und Abb. 6.11(b). Die Bilder sollen eine frei wählbare Größe haben.

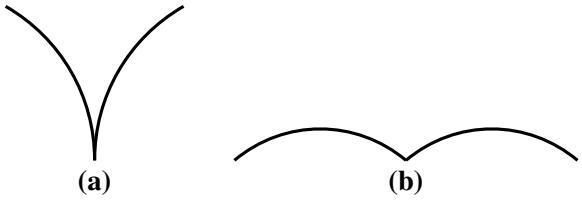


Abbildung 6.11

2. Zeichne das Bild aus Abb. 6.12.

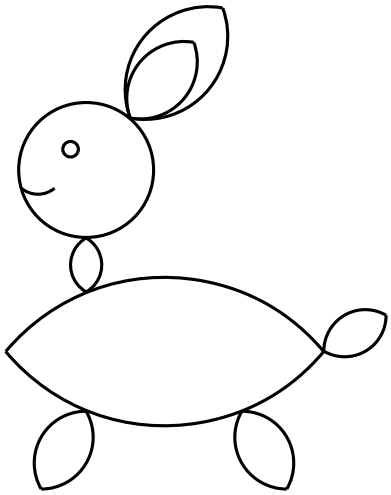


Abbildung 6.12

3. Entwickle in modularer Art ein Programm, das Bilder wie in Abb. 3.8 auf Seite 70 zeichnen kann. Dabei sollen die Größe der schwarzen Quadrate sowie ihre Anzahl (die Höhe des Bildes) frei wählbar sein.

4. Entwickle ein Programm zum Zeichnen roter Kreuze der Form wie in Abb. 3.11 auf Seite 71. Dabei sollen die Größe sowie die Anzahl der Kreuze frei wählbar sein.

5. Das Programm

```
to KREIS2 :UM :FAR
setpencolor :FAR
repeat 360 [ fd :UM rt 1 ]
end
```

zeichnet Kreise in beliebiger Farbe und beliebiger Größe. Entwickle ein Hauptprogramm KETTE, das KREIS2 als Unterprogramm verwendet. Dabei sollte das Programm KETTE eine Folge von vier Kreisen wie in Abb. 6.13 auf der nächsten Seite zeichnen. Die Größe und die Farbe jedes einzelnen Kreises sollen auch frei wählbar sein. Rufe das Programm mit ausgewählten Parametern auf und stelle die Entwicklung der Registerinhalte wie in Tab. 6.1 dar.

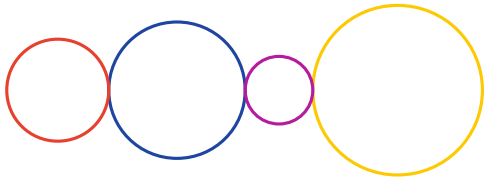


Abbildung 6.13

6. Entwickle ein Programm zum Zeichnen des Bildes in Abb. 4.9 auf Seite 85. Die Farbe des Bildes und die Seitengröße der 6-Ecke sollen frei wählbar sein. Als Unterprogramm zum Zeichnen einzelner 6-Ecke soll folgendes Programm verwendet werden.

```
to ECK6 :GR
repeat 6 [ fd :GR rt 60 ]
end
```

7. In Abb. 5.1 auf Seite 90 sind fünf Quadrate unterschiedlicher Größe vom gleichen Startpunkt (die Ecke links unten) gezeichnet. Entwickle modular ein Programm, das fünf solcher Quadrate beliebiger Größe und Farbe zeichnet. Dabei soll das Hauptprogramm das Programm `QUADRAT :GR` als Unterprogramm verwenden.

   Ruf dein Programm auf, um Quadrate der Größe 50, 70, 100, 140 und 190 zu zeichnen und stell die Entwicklung der Registerinhalte wie in Tab. 6.1 dar.

8. Entwickle ein Programm, ähnlich zu dem Programm aus Kontrollaufgabe 7, mit dem du statt Quadraten regelmäßige 8-Ecke zeichnest.

Lösungen zu ausgesuchten Aufgaben

Aufgabe 6.7
Wenn man für den Parameter `:LANG` den Wert 180° wählt, erhält man einen Kreis. Der Parameter `:GROSS` bleibt frei wählbar und bestimmt die Größe des Kreises. Somit zeichnen beide Programme

```
BLATT 180 x und KREISE x
```

das gleiche Bild und zwar einen Kreis als regelmäßiges 360-Eck mit der Seitenlänge `x`.

Aufgabe 6.15
Weil wir alle vier Größen frei wählbar haben wollen, verwenden wir vier Parameter, für jeden Aufruf des Teilprogramms QUADRAT einen anderen. Dabei achten wir noch darauf, dass zwischen den vier Quadraten kleine Abstände entstehen.

```
to QU4 :G1 :G2 :G3 :G4
QUADRAT :G1
rt 90 pu fd :G1+3 pd lt 90
QUADRAT :G2
rt 90 pu fd :G2+3 pd lt 90
QUADRAT :G3
rt 90 pu fd :G3+3 pd lt 90
QUADRAT :G4
end
```

Für die sieben Zeilen des Programms QU4 erhalten wir beim Aufruf QU4 50 100 150 200 die Entwicklung der Registerinhalte G1, G2, G3 und G4 wie in Tab. 6.2 dargestellt.

	0	1	2	3	4	5	6	7
G1	50							
G2	100							
G3	150							
G4	200							
GR	0	50		100		150		200

Tabelle 6.2

Um es anschaulicher zu machen, schreiben wir die Werte nur dann in die Tabelle, wenn sie sich geändert haben. Ansonsten wären die Werte in den ersten vier Zeilen der Tabelle alle gleich.

Aufgabe 6.17
Weil man sich nach dem Zeichnen eines Vielecks mittels des Programms VIELECK immer in der ursprünglichen Startposition befindet, ist diese Aufgabe sehr leicht lösbar.

```
to VE5 :E1 :E2 :E3 :E4 :E5
VIELECK :E1 VIELECK :E2 VIELECK :E3
VIELECK :E4 VIELECK :E5
end
```

Damit sind :E1, :E2, :E3, :E4 und :E5 globale Parameter des Programms VE5 und :ECKE (aus dem Programm VIELECK) ist der lokale Parameter des Programms VE5.

Lektion 7

Optimierung der Programmlänge und der Berechnungskomplexität

Hinweis für die Lehrperson In dieser Lektion unterrichtet man keine Programmierkonzepte. Deswegen kann man diese Lektion ohne Folgen für den Unterricht folgender Lektionen überspringen. Andererseits bemühen wir uns hier um den ersten Kontakt zu einigen der Grundkonzepte der Informatik – der Beschreibungskomplexität von Programmen (Systemen) und der Berechnungskomplexität im Sinne eines Maßes des Rechenaufwands. Bei der Optimierung dieser Komplexitätsmaße entstehen spannende Optimierungsaufgaben, zu deren Lösung man in der Klasse sehr erfolgreich Wettbewerbe veranstalten kann.

Der rote Faden für die Entwicklung von Programmierkonzepten war die gesunde Faulheit der Programmierer, die auch dazu führt, dass man oft Programme so kurz wie möglich darstellt. In diesem Zusammenhang sagen wir, dass man versucht, die Programmlänge zu **optimieren**, wobei das Optimieren hier **Minimieren** bedeutet. Was bedeutet aber das Maß der Programmlänge genau? Da müssen wir uns zuerst auf eine genaue, einheitliche Definition einigen. Zum einen könnte man die Länge eines Programms an der Anzahl der Symbole (Buchstaben, Ziffern, etc.) messen, zum anderen kann man die Anzahl der Wörter und Zahlen oder die Anzahl der Programmzeilen in Betracht ziehen. Die Anzahl der Symbole sieht zwar genauer aus, aber auf diese Art und Weise würde man Programme und Parameter möglichst kurz, also wenn möglich mit einzelnen Buchstaben, benennen. Und wir haben gelernt, dass für das Lesen und die wiederholte Benutzung eines Programms nach einer gewissen Zeit dieses Vorgehen keine gute Strategie ist.

Aufgabe 7.1 Warum ist die Anzahl der Zeilen eines Programms kein gutes Maß, um die Programmlänge zu messen?

J. Hromkovič, *Lehrbuch Informatik*, DOI 10.1007/978-3-8348-9692-6_8,

Wir werden die **Länge eines Programms** durch die **Anzahl der im Programm vorkommenden Befehlsnamen** bestimmen. Somit hat das Programm

```
fd 100 rt 90 fd 100 rt 90 fd 100 rt 90 fd 100 rt 90
```

die Länge 8, weil dort viermal der Befehl `fd` und viermal der Befehl `rt` vorkommt. Das kürzere Programm zum Zeichnen eines 100×100-Quadrates ist

```
repeat 4 [ fd 100 rt 90 ],
```

weil hier jeweils die drei Befehlsnamen `repeat`, `fd` und `rt` nur einmal vorkommen. Das Programm hat also die Länge drei.

Aufgabe 7.2 Bestimme die Länge des Programms aus Aufgabe 1.17 und der Programme aus den Beispielen 2.2 und 2.3.

Jetzt wissen wir schon, wie man die Länge von Programmen ohne Unterprogramme misst. Wie gehen wir aber vor, wenn man Unterprogramme verwendet? Wir achten dabei darauf, dass wir wirklich ein Maß der Beschreibungsgröße (in der Fachterminologie sprechen wir von **Beschreibungskomplexität**) erhalten. Also geht es darum, wie viel Befehlsnamen man auf ein Platt Papier schreiben muss, um das Hauptprogramm mit allen seinen Unterprogrammen vollständig zu beschreiben. Dies würde dann ungefähr der Speichergröße zur Abspeicherung des Programms im Rechner entsprechen.

Betrachten wir das folgende Programm:

```
QUADRAT 50 QUADRAT 100
rt 180
KREISE 2 KREISE 3 KREISE 2.5
```

Wir wissen, dass Programmnamen wie Befehle funktionieren. Somit verwendet dieses Hauptprogramm zweimal den Befehl `QUADRAT`, einmal den Befehl `rt` und dreimal den Befehl `KREISE`. Somit haben wir genau sechs Befehle geschrieben. Um das Programm vollständig darzustellen, muss man aber auch die Programme `QUADRAT` und `KREISE` aufschreiben.

```
to QUADRAT :GR
repeat 4 [ fd :GR rt 90 ]
end
```

```
to KREISE :LA
repeat 360 [ fd :LA rt 1 ]
end
```

Beide Programme, QUADRAT und KREISE, bestehen jeweils aus einer Zeile, die drei Befehle beinhaltet. Unabhängig davon, wie viele Male diese Programme als Unterprogramme in einem Hauptprogramm aufgerufen werden, wird ihre Länge nur einmal in die Beschreibung des Hauptprogramms einfließen. Somit ist die Länge unseres Programms

$$\underbrace{6}_{\text{Hauptprogramm}} + \underbrace{3}_{\texttt{QUADRAT}} + \underbrace{3}_{\texttt{KREISE}} = 12\,.$$

Aufgabe 7.3 In Lektion 3 haben wir das Programm SCHACH4 zum Zeichnen eines 4×4-Schachfeldes entworfen. Bestimme seine Länge. Beachte, dass du dabei zuerst die Länge der Programme ZEILEA, ZEILEB, QUAD100, SCHW100 und FETT100 bestimmen musst.

Aufgabe 7.4 Bestimme die Länge der Programme, die du zur Lösung der Aufgaben 3.20 auf Seite 68 und 3.21 auf Seite 68 entwickelt hast.

Beispiel 7.1 In Beispiel 2.2 auf Seite 49 hatten wir folgendes Programm zum Zeichnen des Bildes aus Abb. 2.6 (ein 1×10-Feld aus 20×20-Quadraten):

```
repeat 2 [ fd 20 rt 90 fd 200 rt 90 ]
rt 90 fd 20 lt 90
repeat 9 [ fd 20 rt 180 fd 20 lt 90 fd 20 lt 90 ]
```

Die Länge dieses Programms ist 15. Das andere Programm

```
repeat 10 [ repeat 4 [ fd 20 rt 90 ] rt 90 fd 20 lt 90 ]
```

zum Zeichnen des gleichen Bildes hat nur die Länge 7. Kann man es noch verbessern? Das Programm wiederholt zehnmal die gleiche Tätigkeit, indem man zuerst ein Quadrat zeichnet und dann die Schildkröte zur neuen Position bewegt, um dort das nächste Quadrat zu zeichnen. Wenn wir das Quadrat so geschickt zeichnen, dass die Schildkröte an einer günstigen Position zum Zeichnen des nächsten Quadrates aufhört, könnte man in dem zweiten Teil der Schleife etwas sparen. Durch das Programm

```
repeat 7 [ fd 20 rt 90 ]
```

wird auch ein Quadrat gezeichnet. Allerdings hört die Schildkröte in der Position auf zu zeichnen, aus der man direkt das nächste Quadrat zeichnen kann. Nur noch die Orientierung muss mit der Drehung `rt 90` korrigiert werden. Damit erhalten wir das Programm

```
repeat 10 [ repeat 7 [ fd 20 rt 90 ] rt 90 ]
```

mit der Länge 5. Wenn wir diese Idee zum Zeichnen eines beliebigen $1 \times n$-Feldes mit wählbarem n und wählbarer Quadratgröße verwenden, erhalten wir das folgende kurze Programm.

```
to FELDZEILE :N :GR
repeat :N [ repeat 7 [ fd :GR rt 90 ] rt 90 ]
end
```

Wenn wir die Befehle `to` und `end` in die Messung der Länge eines Programms nicht einbeziehen (und damit nur die Länge des Programmkörpers messen), können wir mit der Programmlänge 5 beliebige $1 \times n$-Felder zeichnen. □

Aufgabe 7.5 Verwende die Idee aus Beispiel 7.1, um ein kurzes Programm zum Zeichnen beliebiger $m \times n$-Felder zu entwickeln.

Aufgabe 7.6 Schreibe ein Programm zum Zeichnen des Bildes aus Abb. 2.15 auf Seite 51. Versuche dabei die Programmlänge zu minimieren.

Aufgabe 7.7 Schreibe ein kurzes Programm zum Zeichnen des Bildes aus Abb. 2.18 auf Seite 53.

Aufgabe 7.8 Schreibe ein kurzes Programm zum Zeichnen des Bildes aus Abb. 2.21 auf Seite 55.

Aufgabe 7.9 Versuche die Länge des Programms `SCHACH4` zum Zeichnen eines 4×4-Schachbrettes zu kürzen.

Bei der Bemühung kurze Programme zu schreiben, kann man auch neue Ideen bekommen. Zum Beispiel bei der Bemühung, das Bild aus Abb. 2.3 auf Seite 43 zu zeichnen, merkt man, dass das ganze Programm auf einer geschickten Wiederholung der Zeichnung der Treppen aufgebaut werden kann. Die Grundidee ist, dass man Treppen nach oben oder nach unten mit dem gleichen Programm

```
to TREPP
repeat 5 [ fd 20 rt 90 fd 20 lt 90 ]
end
```

zeichnen kann. Es geht nur um die richtige Orientierung der Schildkröte am Anfang. Somit zeichnet das Programm

```
TREPP rt 90 TREPP fd 40 lt 90 TREPP rt 90 TREPP
```

das gewünschte Bild aus Abb. 2.3 auf Seite 43. Die Länge dieses Programms ist nur $8+5=13$.

Aufgabe 7.10 Schreibe ein kurzes Programm zum Zeichnen von Mustern wie in Abb. 7.1. Die Anzahl sowie die Größe der Treppen sollen dabei frei wählbar sein.

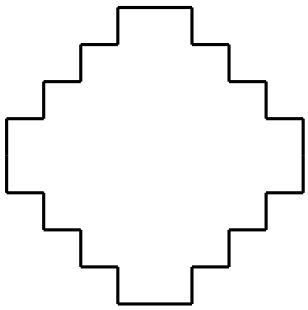


Abbildung 7.1

Aufgabe 7.11 Versuche ein kurzes Programm zum Zeichnen einer frei wählbaren Anzahl Pyramiden wie in Abb. 2.3 auf Seite 43 zu entwickeln. Die Stufenhöhe soll dabei fünf sein und die Anzahl der Stufen soll frei wählbar sein.

Wir hatten als roten Faden bei der Entwicklung von neuen Programmierkonzepten immer die „gesunde“ Faulheit der Programmierer in den Vordergrund gestellt. Wie es aber im Leben so ist, sollte man kein Prinzip unter allen Umständen über alle anderen Prinzipien stellen. Die Informatiker machen das auch nicht. Kurze Programme müssen nicht unbedingt immer die Schnellsten sein. Und die **Effizienz** ist in der Algorithmik extrem wichtig. Die Effizienz wird durch die **Berechnungskomplexität** gemessen. Hier misst man die Menge an Arbeit, die ein Rechner bei der Ausführung eines Programms leisten muss.

Diese Messung kann unterschiedlich präzise sein. Wir betrachten hier eine sehr einfache und trotzdem eine realistische Messung der Berechnungskomplexität. Wir zählen die Anzahl der ausgeübten Grundbefehle wie `fd`, `bk`, `rt`, `lt`, `pu`, `pd`, `setpencolor`, usw. Es mag sein, dass die Umsetzung der Befehle `rt 90`, `fd 100` oder `pu` unterschiedlich großen Zeiteinheiten entspricht, aber wir verzichten hier auf diese Unterschiede. Genauer betrachtet kann die Umsetzung von `fd 200` und `fd 10` auch unterschiedlich lange dauern, aber auf die Messung dieser Unterschiede werden wir hier verzichten[1].

Ein kurzes Programm ist keine Garantie für die Effizienz. Analysieren wir das in Beispiel 7.1 entwickelte Programm

```
repeat 10 [ repeat 7 [ fd 20 rt 90 ] rt 90 ],
```

das wahrscheinlich das kürzestmögliche Programm zum Zeichnen des Bildes aus Abb. 2.6 auf Seite 46 ist. Wie hoch ist seine Berechnungskomplexität? In dem Programmteil

```
repeat 7 [ fd 20 rt 90 ]
```

werden siebenmal die beiden Befehle `fd` und `rt` wiederholt. Dies entspricht insgesamt $7 \cdot 2 = 14$ ausgeführten Befehlen. In der Schleife `repeat 10 [ ... ]` wird dieser Teil zehnmal wiederholt und zusätzlich wird auch der letzte Befehl `rt 90` dieser Schleife zehnmal wiederholt. Damit ist die Anzahl der ausgeführten Befehle

$$10 \cdot (14 + 1) = 150$$

Das lange Programm

```
repeat 2 [ fd 20 rt 90 fd 200 rt 90 ]
rt 90 fd 20 lt 90
repeat 9 [ fd 20 rt 180 fd 20 lt 90 fd 20 lt 90 ]
```

hat die Berechnungskomplexität von nur

$$\underbrace{2 \cdot 4}_{\text{1. Zeile}} + \underbrace{3}_{\text{2.Zeile}} + \underbrace{9 \cdot 6}_{\text{3. Zeile}} = 65.$$

[1] Keinesfalls kostet die Umsetzung von `fd 100` 10-mal so viel Zeit wie die Umsetzung von `fd 10`. Die Hauptkosten sind im Aufruf des Befehls und deswegen dürfen wir bei einer gröberen Messung die Parametergröße vernachlässigen.

Aufgabe 7.12 Bestimme die Berechnungskomplexität folgender Programme:

a) `repeat 10 [ fd 20 rt 90 fd 20 lt 90] fd 50 rt 90 fd 10`

b) `repeat 10 [ repeat 4 [fd 20 rt 90] rt 90 fd 20 lt 90]`

Die Frage ist jetzt, ob wir noch ein schnelleres Programm zum Zeichnen des 1×10-Feldes aus Abb. 2.6 auf Seite 46 entwickeln können. Offensichtlich muss die Idee sein, nach Möglichkeit den wiederholten Durchlauf der gleichen Strecke zu vermeiden. Das Zeichnen von Quadraten mittels

```
repeat 7 [ fd 20 rt 90 ]
```

führt zwar zum kürzesten Programm, erfordert aber das zweifache Laufen über drei der vier Seiten des Quadrates. Eine Idee könnte sein, mittels

```
repeat 3 [ fd 20 rt 90 ]
```

nur drei Seiten jedes Quadrates zu zeichnen (s. Abb. 7.2) und danach die fehlenden, unteren Seiten aller Quadrate mittels `fd 200` in einem Zug zu zeichnen.

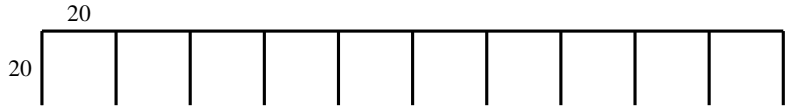


Abbildung 7.2

Das entsprechende Programm sieht dann wie folgt aus:

```
repeat 10 [ repeat 3 [ fd 20 rt 90 ] rt 90 ] lt 90 fd 200
```

Die Berechnungskomplexität dieses Programms ist

$$10 \cdot (3 \cdot 2 + 1) + 2 = 72\,.$$

Aufgabe 7.13 Wir sehen, dass das neue Programm nicht das Schnellste ist, weil wir schon ein Programm mit einer Berechnungskomplexität von 65 im Beispiel 2.2 (zweite Lösung) entwickelt

haben. Kannst du ein noch effizienteres Programm zum Zeichnen der Leiter aus Abb. 2.6 auf Seite 46 schreiben? Es müsste mit wenigstens 60 ausgeführten Grundbefehlen machbar sein.

Aufgabe 7.14 Versuche ein effizientes Programm zum Zeichnen des Bildes aus Abb. 4.7 auf Seite 84 zu finden. Versuche weiter, es auch für zehn Sechsecke nebeneinander zu konzipieren.

Aufgabe 7.15 In Beispiel 2.3 haben wir ein Programm zum Zeichnen des Bildes aus Abb. 2.12 auf Seite 49 entwickelt. Bestimme seine Berechnungskomplexität und entwirf ein Programm, das effizienter ist.

Aufgabe 7.16 Entwickle ein effizientes Programm zum Zeichnen des Bildes aus Abb. 2.15 auf Seite 51.

Bisher haben wir die Berechnungskomplexität nur von solchen Programmen untersucht, die für die Anzahl der Wiederholungen einer Schleife keine Parameter verwenden. Damit war die Anzahl der Wiederholungen immer eine feste Zahl und somit konnten wir auch die Berechnungskomplexität eines Programms als eine konkrete Zahl ermitteln. Wenn man Parameter zur Steuerung der `repeat`-Befehle verwendet, würde die Berechnungskomplexität von den aktuellen Parameterwerten abhängen. Deswegen drücken wir die Berechnungskomplexität durch einen arithmetischen Ausdruck aus, in dem die Parameternamen vorkommen. Im Folgenden führen wir für die Bestimmung der Berechnungskomplextität eines Programms *P* den Ausdruck **Zeit(*P*)** ein.

Betrachten wir das folgende Programm:

```
to LEITER :ANZ
repeat :ANZ [ repeat 4 [ fd 20 rt 90 ] rt 90 fd 20 lt 90 ]
end
```

Die Berechnungskomplexität des Programms `LEITER` hängt vom Wert des Parameters `:ANZ` ab und kann wie folgt ausgedrückt werden:

$$\text{Zeit}(\texttt{LEITER}) = \texttt{ANZ} \cdot (4 \cdot 2 + 3) = 11 \cdot \texttt{ANZ}.$$

Aufgabe 7.17 Offensichtlich entspricht `:ANZ` der Anzahl der nebeneinander gezeichneten Quadrate. Kannst du ein Programm zum Zeichnen einer gleichen Klasse von Bildern aufschreiben, deren Berechnungskomplexität kleiner als $7 \cdot \texttt{ANZ}$ ist?

Analysieren wir jetzt die Berechnungskomplexität des folgenden Programms zum Zeichnen eines $X \times Y$-Feldes mit mehreren Parametern.

```
to FELD :X :Y :GR
repeat :X [repeat :Y [ TEIL :GR ] lt 90 fd :Y*:GR rt 90 fd
:GR]
end
```

Das Programm FELD hat das Unterprogramm TEIL.

```
to TEIL :GR
repeat :3 [ fd :GR rt 90 ] rt 90
end
```

Die Berechnungskomplexität des Programms FELD ist $3 \cdot 2 + 1 = 7$. Damit ist die Komplexität des Programmteils

```
repeat :Y [ TEIL :GR ]
```

genau $7 \cdot Y$. Zusammengefasst ist die Berechnungskomplexität von FELD

$$\text{Zeit}(\texttt{FELD}) = \texttt{X} \cdot (7 \cdot \texttt{Y} + 4) = 7 \cdot \texttt{X} \cdot \texttt{Y} + 4 \cdot \texttt{X}.$$

Aufgabe 7.18 Versuche ein Programm zum Zeichnen der $X \times Y$-Felder zu entwickeln, das eine Berechnungskomplexität hat, die kleiner ist als $6 \cdot \texttt{X} \cdot \texttt{Y}$.

Aufgabe 7.19 Schreibe ein Programm zum Zeichnen regelmäßiger Vielecke mit wählbar vielen Ecken und bestimme seine Berechnungskomplexität.

Aufgabe 7.20 Bestimme die Berechnungskomplexität des Programms QQQ.

```
to QQQ :ANZAHL :LANG :WINK
repeat :ANZAHL [ repeat 4 [ fd :LANG rt 90 ] rt :WINK ]
end
```

Zusammenfassung

Es gibt mehrere Kriterien, nach denen wir Programme beurteilen können. Das Allerwichtigste ist die Korrektheit. Damit meinen wir, dass das Programm genau das macht, was wir von ihm erwarten. Später haben wir gelernt, dass auch ein überschaubarer modularer Entwurf von Programmen wichtig ist. Ein modularer Entwurf vereinfacht nicht nur den Prozess des Entwurfs, sondern auch die Überprüfung der Korrektheit des Programms und gegebenenfalls auch die Suche nach den Fehlern. Zusätzlich ist unser Programm besser für andere lesbar und somit verständlicher. Das ist für eine potenzielle Weiterentwicklung des Programms wichtig.

In dieser Lektion haben wir gelernt, dass man zusätzlich die Programme nach ihrer Länge und Berechnungskomplexität bewerten kann. Die Länge eines Programms ist seine Darstellungsgröße. Wir sprechen auch von der Beschreibungskomplexität und messen sie in der Anzahl der im Programm vorkommenden Befehle. Dieses Maß bestimmt auch die Größe des Speichers, der für das Programm nötig ist. Es ist nicht immer unbedingt das Wichtigste, möglichst kurze Programme zu schreiben. Wenn dabei die Anschaulichkeit oder die modulare Struktur des Programms verloren geht, muss eine kurze Schreibweise nicht unbedingt vorteilhaft sein.

Nach der Korrektheit und somit der Zuverlässigkeit ist die Berechnungskomplexität das wichtigste Kriterium für die Bewertung von Programmen. Die Berechnungskomplexität misst man in der Anzahl der vom Rechner auszuführenden Befehle. Die Anzahl kann von den Werten der Programmparameter abhängen, wenn die Parameter die Anzahl der Schleifenwiederholungen bestimmen.

Kontrollfragen

1. Welche Vorteile haben kurze Programme?

2. Wie misst man die Länge von Programmen?

3. Ist die Länge eines Programms immer eine fest Zahl oder kann sie von Parameterwerten abhängen?

4. Warum ist uns die Berechnungskomplexität so wichtig?

5. Wie misst man die Berechnungskomplexität?

6. Warum kann die Berechnungskomplexität von den Parameterwerten abhängen?

7. Wie misst man die Länge von Programmen, die Unterprogramme haben?

8. Wie gehst du bei der Messung der Berechnungskomplexität von Programmen vor, die mehrere ineinander verschachtelte Schleifen haben?

Kontrollaufgaben

1. Entwirf ein Programm zum Zeichnen des Bildes aus Abb. 7.3 und bestimme seine Länge. Versuche, die Länge deines Programms zu minimieren.

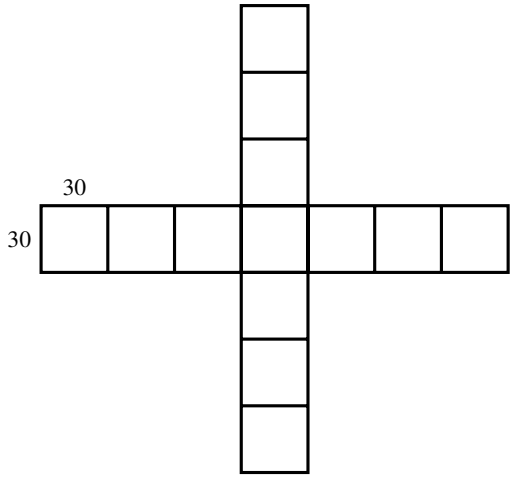


Abbildung 7.3

2. Entwirf ein effizientes Programm für das Bild aus Abb. 7.3. Versuche dazu, seine Berechnungskomplexität zu minimieren.

3. Entwickle zwei Programme zum Zeichnen der Klasse der Kreuze wie in Abb. 7.3, wobei die Quadratgröße, die Höhe (die vertikale Anzahl der Quadrate) und die Breite (die horizontale Anzahl der Quadrate) frei wählbar sein sollen. Bei dem ersten Programm achte auf die Programmlänge. Bei dem Entwurf des zweiten Programms versuche, seine Berechnungskomplexität zu minimieren.

4. Entwirf zwei Programme, die das Bild aus Abb. 7.4 auf der nächsten Seite zeichnen. Achte bei dem ersten Programm auf die Programmlänge und beim zweiten auf die Effizienz.

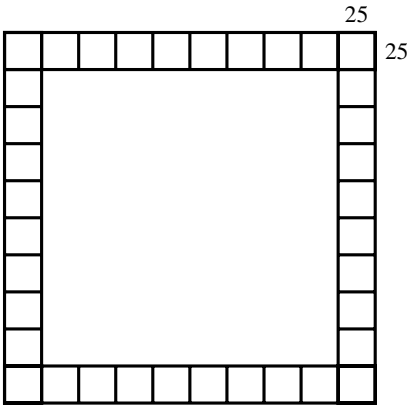


Abbildung 7.4

5. Entwirf Programme zum Zeichnen des Bildes aus Abb. **??** auf Seite ??. Versuche zuerst, die Programmlänge und später die Berechnungskomplexität zu minimieren.

6. Schreibe ein Programm zum Zeichnen von schwarzen Rechtecken. Beide Seitenlängen sollen frei wählbar sein. Versuche zuerst, die Programmlänge und danach die Berechnungskomplexität zu minimieren.

7. In der dritten Lektion hast du in Kontrollaufgabe 3 ein Programm zum Zeichnen des Bildes aus Abb. 3.8 auf Seite 70 entwickelt. Untersuche seine Länge und seine Berechnungskomplexität und versuche, sie zu minimieren.

8. Wie groß muss die Berechnungskomplexität eines Programms mindestens sein, wenn es ein regelmäßiges X-Eck zeichnet?

9. Entwickle Programme zum Zeichnen eines 5×10-Feldes mit frei wählbarer Feldgröße. Versuche zuerst, die Programmlänge und danach die Berechnungskomplexität zu minimieren.

10. Entwickle zwei Programme zum Zeichnen des klassischen 8×8 Schachbrettes mit wählbarer Quadratgröße. Dabei darfst du alle bisher entwickelten Programme als Unterprogramme verwenden. Minimiere bei deinem ersten Programm die Programmlänge und achte beim zweiten Programm auf die Effizienz.

Lösungen zu ausgesuchten Aufgaben

Aufgabe 7.3
Das Programm SCHACH4 sieht wie folgt aus:

```
to SCHACH4
repeat 2      [ ZEILEA bk 100 lt 90 fd 400 rt 90
                ZEILEB bk 100 lt 90 fd 400 rt 90 ]
end
```

Damit gilt:

$$\text{Länge}(\texttt{SCHACH4}) \leq 11 + \text{Länge}(\texttt{ZEILEA}) + \text{Länge}(\texttt{ZEILEB}).$$

Wir schreiben „$\leq$“ und nicht „$=$“, weil die Programme ZEILEA und ZEILEB gleiche Unterprogramme verwenden können und dann gibt es keinen Grund diese Unterprogramme zweimal in die Beschreibungskomplexität einzuberechnen. Wie wir sehen, ist das für die Programme

```
to ZEILEA
repeat 2 [ SCHW100 QUAD100 rt 90 fd 100 lt 90 ]
end
```

und

```
to ZEILEB
repeat 2 [ QUAD100 rt 90 fd 100 lt 90 SCHW100 ]
end
```

der Fall. Es gilt:

$$\text{Länge}(\texttt{ZEILEA}) = 6 + \text{Länge}(\texttt{SCHW100}) + \text{Länge}(\texttt{QUAD100}) \quad \text{und}$$
$$\text{Länge}(\texttt{ZEILEB}) = 6 + \text{Länge}(\texttt{SCHW100}) + \text{Länge}(\texttt{QUAD100}).$$

Für Länge(SCHACH4) ziehen wir Länge(SCHW100) und Länge(QUAD100) nur einmal in Betracht:

$$\text{Länge}(\texttt{SCHACH4}) = 23 + \text{Länge}(\texttt{SCHW100}) + \text{Länge}(\texttt{QUAD100}).$$

Das Programm QUAD100 hat die Länge 3. Das Programm SCHW100 ist wie folgt definiert:

```
to SCHW100
repeat 100 [ FETT100 ]
end
```

Also gilt:

$$\text{Länge}(\texttt{SCHW100}) = 2 + \text{Länge}(\texttt{FETT100})$$

und wir erhalten

$$\text{Länge}(\texttt{SCHACH4}) = 23 + 2 + \text{Länge}(\texttt{FETT100}) + 3 = 28 + \text{Länge}(\texttt{FETT100}).$$

Das Programm FETT100 besteht aus sechs Befehlen (Abb. 3.1 auf Seite 63) und besitzt kein weiteres Unterprogramm. Damit können wir unsere Analyse der Programmlänge folgendermaßen abschließen:

$$\text{Länge}(\texttt{SCHACH4}) = 28 + 6 = 34.$$

Aufgabe 7.6
Wir entwickeln für die Zeichnung der $X \times Y$-Felder der Quadratgröße GR mit geraden X und Y eine neue Strategie, die versucht, so viele lange Linien wie möglich zu zeichnen. Wir fangen mit dem Programm

```
to ZICK1 :X :Y :GR
repeat :Y/2 [ UU :X :GR ]
end
```

an, das das Unterprogramm

```
to UU :Z :GR
fd :Z*:GR rt 90 fd :GR rt 90
fd :Z*:GR lt 90 fd :GR lt 90
end
```

enthält.

Für $Y = 6$ zeichnet das Programm ZICK1 das Muster aus Abb. 7.5. Die Höhe des Musters ist X×GR.

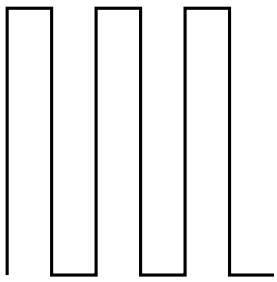


Abbildung 7.5

Wenn wir jetzt die Schildkröte mit `lt 90` drehen und ein ähnliches Muster mit vertauschten Rollen von X und Y zeichnen lassen, erhalten wir das folgende Programm:

```
to KURZFELD :A :B :GR
ZICK1 :A :B :GR
lt 90
ZICK1 :B :A :GR
fd :B*:GR bk :B*:GR lt 90
fd :A*:GR
end
```

Tippe das Programm ein und überprüfe die Auswirkungen aller Unterprogramme sowie des Hauptprogramms.

Wir analysieren jetzt die Länge von `KURZFELD`. Wir fangen dabei mit der Länge der Unterprogramme an.

$$\begin{aligned} \text{Länge}(\texttt{UU}) &= 8 \\ \text{Länge}(\texttt{ZICK1}) &= 2 + \text{Länge}(\texttt{UU}) &&= 2+8 = 10 \\ \text{Länge}(\texttt{KURZFELD}) &= 7 + \text{Länge}(\texttt{ZICK1}) &&= 7+10 = 17 \end{aligned}$$

Ganz schön kurz dieses Programm. Könntest du es noch verbessern? Versuche, es für ungerade X und Y zu erweitern.

Aufgabe 7.10
Wir bezeichnen durch `:AN` die Anzahl der Treppen und durch `:GR` die Treppengröße.

```
to MUSTER :AN :GR
repeat 4 [ repeat :AN [ fd :GR rt 90 fd :GR lt 90 ] rt 90]
end
```

Die Länge des Programms ist 7.

Aufgabe 7.17
Durch die passende Nutzung des Befehls `bk` sparen wir uns auf folgende Art und Weise mehrere Umdrehungen.

```
to SCHNELLTR :AN
repeat :AN [ fd 20 bk 20 rt 90 fd 20 lt 90 ]
fd 20 lt 90 fd 20*:AN
end
```

Die Berechnungskomplexität von `SCHNELLTR` ist $5 \cdot$`AN`$+3$.

Aufgabe 7.18
Analysieren wir das Programm KURZFELD, das wir in der Musterlösung zur Aufgabe 7.6 entwickelt haben. Wir fangen mit den Unterprogrammen an.

$$\text{Zeit}(\texttt{UU}) = 8 \qquad \text{und}$$
$$\text{Zeit}(\texttt{ZICK1}) = \frac{\texttt{Y}}{2},$$

wobei :Y der zweite Parameter von ZICK1 ist. Damit gilt für Das Zeichnen eines $A \times B$-Felds:

$$\text{Zeit}(\texttt{KURZFELD}) = \frac{\texttt{B}}{2} + 1 + \frac{\texttt{A}}{2} + 4 = 5 + \frac{1}{2} \cdot (\texttt{A} + \texttt{B}).$$

Dies ist wesentlich besser als $7 \cdot \texttt{A} \cdot \texttt{B} + 4 \cdot \texttt{A}$.

Lektion 8

Das Konzept von Variablen und der Befehl `make`

Wir haben schon einiges gelernt, um kurze und gut strukturierte Programme zu entwickeln. Das Konzept der Parameter war uns dabei besonders hilfreich. Dank der Parameter können wir ein Programm schreiben, das man zum Zeichnen einer ganzen Klasse von Bildern verwenden kann. Durch die Wahl der Parameter bestimmen wir bei dem Programmaufruf einfach, welches Bild gezeichnet wird. Somit haben wir Programme zum Zeichnen von Quadraten, Rechtecken, Kreisen oder anderen Objekten mit beliebiger Größe. Es gibt aber auch Situationen, in denen uns Parameter nicht hinreichend helfen. Betrachten wir die folgende Aufgabe. Man soll eine frei wählbare Anzahl von Quadraten wie in Abb. 8.1 auf der nächsten Seite zeichnen. Wir fangen mit dem Quadrat der Größe 20×20 an und zeichnen weitere größere Quadrate, wobei das nachfolgende immer eine um zehn Schritte größere Seitenlänge haben soll als das vorherige.

Für die Zeichnung solcher Bilder können wir das Programm `QUADRAT :GR` wie folgt verwenden:

```
QUADRAT 20
QUADRAT 30
QUADRAT 40
QUADRAT 50
QUADRAT 60
...
```

Wie lang das resultierende Programm wird, hängt davon ab, wie viele Quadrate wachsender Größe man zeichnen will. Für jede gewünschte Anzahl von Quadraten muss

J. Hromkovič, *Lehrbuch Informatik*, DOI 10.1007/978-3-8348-9692-6_9,

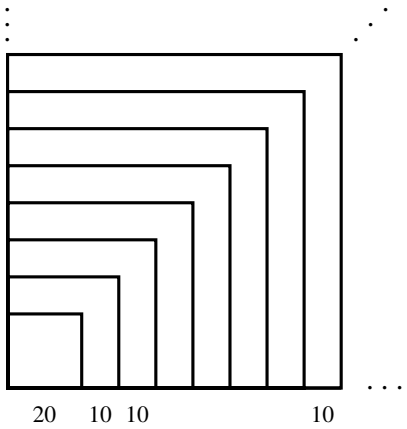


Abbildung 8.1

man ein eigenes Programm schreiben. Wir hätten aber lieber nur ein Programm anstelle von unendlich vielen, indem man die Anzahl der gezeichneten Quadrate mittels eines Parameters selbst wählen kann.

Die Idee zur Verwirklichung dieses Wunsches basiert auf dem Konzept der Variablen. In gewissem Sinne sind Variablen eine Verallgemeinerung von Parametern, wenn man dem Programm ermöglicht, während des Laufs die Werte der Parameter zu verändern. Mit einer solchen Möglichkeit könnte man ein Programm zum Zeichnen von :AN vielen Quadraten (Abb. 8.1) wie folgt darstellen.

```
to VIELQ :GR :AN

QUADRAT :GR
Erhöhe den Wert von :GR um 10
QUADRAT :GR
Erhöhe den Wert von :GR um 10

...

QUADRAT :GR
Erhöhe den Wert von :GR um 10
end
```

} :AN - mal

Wenn wir das Programm wie oben darstellen, sehen wir sofort, dass sich die beiden

Zeilen

```
QUADRAT :GR
Erhöhe den Wert von :GR um 10
```

:AN-mal wiederholen. Das ruft nach einer Schleife mit AN-vielen Durchläufen und führt somit zu folgendem Programm:

```
to VIELQ :GR :AN
repeat :AN [ QUADRAT :GR Erhöhe den Wert von :GR um 10 ]
end
```

Die Erhöhung des Wertes von :GR um 10 erreicht man durch den Befehl

```
make "GR :GR+10.
```

Wie setzt der Rechner den make-Befehl um?

Der Befehl make signalisiert ihm, dass er den Wert des Parameters ändern soll, dessen Name hinter den " nach dem Befehl make steht. Alles, was weiter rechts steht, ist ein arithmetischer Ausdruck, dessen Wert zu berechnen ist und dessen Resultat im Register mit dem Namen des zu ändernden Parameters gespeichert wird. Anschaulich kann man es folgendermaßen darstellen.

make	"A	Arithmetischer Ausdruck
Befehl zur Änderung eines Parameterwertes	Name des Parameters, dessen Wert geändert werden soll	Die Beschreibung der Rechenregel zur Bestimmung des neuen Wertes für den Parameter A.

Im Folgenden nennen wir Parameter, deren Wert sich im Laufe eines Programms ändern, nicht mehr Parameter, sondern **Variablen**. Der Name Variable signalisiert, dass es um etwas geht, was variieren kann, also um etwas Veränderliches. Der Befehl

```
make "GR :GR+10.
```

wird also von dem Rechner wie folgt umgesetzt. Der Rechner nimmt zuerst den arithmetischen Ausdruck und ersetzt :GR durch den aktuellen Wert von :GR. Dann addiert er zu

diesem Wert die Zahl zehn und speichert das Resultat im Register GR. Somit liegt jetzt im Register GR eine um 10 größere Zahl als vorher.

Das Programm zum Zeichnen einer freien Anzahl von Quadraten sieht dann so aus:

```
to VIELQ :GR :AN
repeat :AN [ QUADRAT :GR make "GR :GR+10 ]
end
```

Dabei ist :GR eine Variable und :AN ein Parameter des Programms. Variable ist ein Oberbegriff. Dies bedeutet, dass die Parameter eines Programms spezielle Variablen sind, deren Werte sich während der Ausführung nicht mehr ändern.

Aufgabe 8.1 Tippe das Programm VIELQ ein und teste es für die Aufrufe VIELQ 20 20, VIELQ 100 5 und VIELQ 10 25.

Aufgabe 8.2 Mit VIELQ zeichnen wir eine Folge von Quadraten, die immer um zehn größer werden. Jetzt wollen wir die Vergrößerung mittels des Parameters :ST frei wählbar machen. Kannst Du das Programm VIELQ entsprechend zu dem Programm VIELQ1 erweitern?

Aufgabe 8.3 Schreibe ein Programm zum Zeichnen einer frei wählbaren Anzahl gleichseitiger Dreiecke wie in Abb. 8.2. Dabei soll die Größe immer um den Wert fünf wachsen und das kleinste Dreieck soll zusätzlich eine frei wählbare Seitenlänge :GR haben.

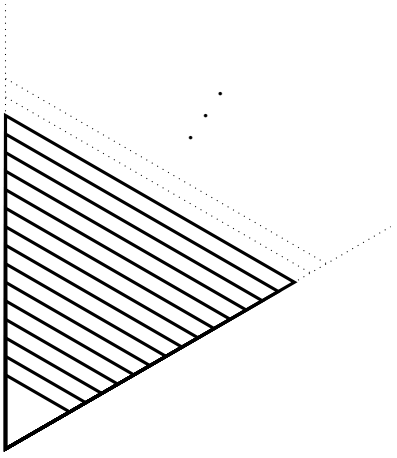


Abbildung 8.2

Aufgabe 8.4 Wir haben schon gelernt, Schnecken mit fester Größe zu zeichnen. Jetzt solltest du ein Programm schreiben, mit dem man beliebige Schnecken wie in Abb. 8.3 zeichnen kann.

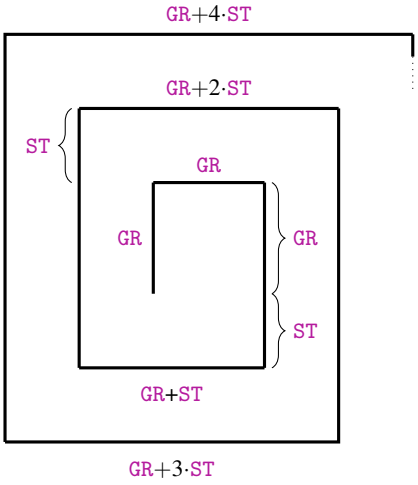


Abbildung 8.3

Aufgabe 8.5 Wir haben das Programm VIELQ zum Zeichnen beliebig vieler regelmäßiger Quadrate mit einer gemeinsamen Ecke. Ähnlich sind wir in Aufgabe 8.3 vorgegangen. Anstatt regelmäßiger Quadrate haben wir regelmäßige Dreiecke mit wachsender Seitengröße gezeichnet. Entwirf jetzt ein Programm zum Zeichnen einer beliebig langen Folge von regelmäßigen Vielecken. Dabei sollen die Anfangsseitengröße, die Anzahl der Ecken und die Vergrößerung der Seitenlänge in jedem Schritt frei wählbar sein. Zeichne danach 20 regelmäßige 12-Ecke, deren Seitenlänge immer um fünf wächst. Welche Variablen in diesem Programm sind Parameter und welche nicht?

Bevor wir damit anfangen, die Variablen und den Befehl `make` intensiv zu verwenden, lernen wir zuerst, wie der Befehl `make` genau funktioniert.

Wir sind in LOGO nicht gezwungen, alle im Programm verwendeten Variablen hinter `to` und dem Programmnamen aufzulisten. Wenn wir innerhalb des Programms

```
make "A  Ausdruck
```

schreiben und `:A` wurde noch nicht definiert und wurde daher auch noch nicht verwendet,

benennt der Rechner ein neues Register mit A und ermöglicht uns damit, :A als Variable zu verwenden.

Wir können es mit folgendem kleinen Testprogramm T1 überprüfen.

```
to T1
make "A 50
QUADRAT :A
end
```

Hier sehen wir, dass man das Unterprogramm QUADRAT mit dem Parameter :A im Hauptprogramm T1 verwendet, obwohl :A bei der Benennung des Programms durch to nicht erwähnt worden ist. Aber der Befehl

```
make "A  50
```

verursacht, dass :A als Variable nachdefiniert wird und sofort den Wert 50 zugewiesen bekommt.

Aufgabe 8.6 Tippe T1 ein und teste, ob tatsächlich ein 50×50-Quadrat gezeichnet wird. Danach modifiziere T1 wie folgt:

```
to T1
make "A :A+50
QUADRAT :A
end
```

Schreibe jetzt den Befehl

```
repeat 5 [ T1 ]
```

und beobachte, was der Rechner zeichnet. Kannst du dafür eine Erklärung finden?

Schreibe jetzt folgendes Programm auf:

```
to T2 :A
make "A :A+50
QUADRAT :A
end
```

Was passiert jetzt nach dem Aufruf

```
repeat 5 [ T2 50 ]?
```

Findest du eine Erklärung für den Unterschied?

Den Befehl `make` kann man tatsächlich dazu verwenden, um gesuchte Werte auszurechnen. Nehmen wir an, wir wollen ein Quadrat der Größe

$$X = \texttt{B} \cdot \texttt{B} - 4 \cdot \texttt{A} \cdot \texttt{C}$$

für gegebene Parameter `:A`, `:B`, `:C` eines Programms zeichnen. Wir könnten wie folgt vorgehen:

```
to T3 :A :B :C
make "X :B * :B
make "Y 4 * :A * :C
make "X :X - :Y
QUADRAT :X
end
```

Die Variablen `:A`, `:B` und `:C` des Programms `T3` sind Parameter. In Tab. 8.1 verfolgen wir die Entwicklung der Speicherinhalte nach dem Bearbeiten der einzelnen Zeilen von `T3` beim Aufruf `T3 10 30 5`. In der ersten Spalte der Tabelle sehen wir die Werte von `:A`, `:B`

	0	1	2	3	4
A	10	10	10	10	10
B	30	30	30	30	30
C	5	5	5	5	5
X	-	900	900	700	700
Y	-	-	200	200	200

Tabelle 8.1

und `:C`, die durch den Aufruf `T3 10 30 5` eingestellt worden sind. Zu diesem Zeitpunkt gibt es noch keine Register für `X` und `Y`. Diese Tatsache notieren wir mit dem Strich -. Nach der Bearbeitung der ersten Zeile

```
make "X :B * :B
```

entsteht die Variable :X. Der Rechner setzt den Wert 30 von B in den Ausdruck :B ∗ :B und erhält 30 · 30. Das Resultat ist 900, und dieser Wert wird im Register X abgespeichert. Bis jetzt gibt es noch kein Register mit dem Namen Y.

Bei der Bearbeitung der Programmzeile

```
make "Y 4 ∗ :A ∗ :C
```

wird zuerst das Register Y definiert. In den Ausdruck 4 ∗ :A ∗ :C setzt der Rechner die aktuellen Werte von A und C ein und erhält $4 \cdot 10 \cdot 5 = 200$. Der Wert 200 wird im Register Y abgespeichert. Bei der Bearbeitung des Programmteils

```
make "X :X − :Y
```

wird kein neues Register angelegt, weil ein Register mit dem Namen X bereits existiert. In den Ausdruck :X−:Y werden die aktuellen Werte von X und Y eingesetzt und der Rechner rechnet $900 - 200 = 700$. Der Wert 700 wird im Register X abgespeichert. Durch das Speichern von 700 in X wird der alte Inhalt 900 aus dem Register X vollständig gelöscht. In der letzten Programmzeile wird nicht gerechnet, sondern nur ein Quadrat der Größe X gezeichnet. Deswegen ändern sich die Werte der Variablen durch die Ausführung dieser Zeile nicht.

Aufgabe 8.7 Bei der Berechnung der Seitenlänge X des Quadrates können abhängig von den eingestellten Werten von :A, :B und :C auch negative Zahlen entstehen. LOGO zeichnet in diesem Fall auch Quadrate, allerdings anders. Probiere es aus und erkläre, wie es dazu kommt.

Aufgabe 8.8 Zeichne eine Tabelle (ähnlich Tab. 8.1), in der du die Entwicklung der Speicherinhalte beim Aufruf

```
T3 20 25 10
```

dokumentierst.

Aufgabe 8.9 Betrachte das folgende Programm:

```
to TT :A :B
make "A :A+10−5
make "X :B−:A+7
make "Y 40
make "A :X−2∗:Y
make "Z :B
make "X :X+:Y+:Z
make "Y :B/:X
end
```

Welche der fünf Variablen von TT sind Parameter? Zeichne wie in Tab. 8.1 die Entwicklung der Speicherinhalte jeweils nach der Ausführung einer Zeile des Programms bei folgenden Aufrufen:

a) TT 0 10

b) TT 5 30

c) TT -5 20

Aufgabe 8.10 Schreibe ein Programm, das zuerst ein regelmäßiges gleichseitiges Dreieck mit der Seitenlänge 20 zeichnet. Danach schreibe eines für ein regelmäßiges Viereck (Quadrat) mit der Seitenlänge 20, danach eines für ein regelmäßiges 5-Eck mit Seitenlänge 20, usw. Das nachfolgende Vieleck soll immer eine Ecke mehr haben als sein Vorgänger. Die Anzahl AN der gezeichneten Vielecken soll dabei frei wählbar sein.

Beispiel 8.1 Wir sollen ein Programm RE2ZU1 :UM entwickeln, das Rechtecke zeichnet, deren Umfang :UM ist und deren horizontale Seite zweimal so lang ist wie die Vertikale.

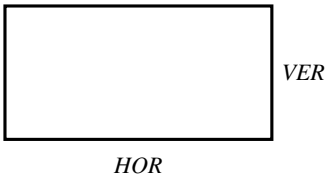


Abbildung 8.4

Wir wissen, dass (s. Abb. 8.4)

$$UM = 2 \cdot VER + 2 \cdot HOR \tag{8.1}$$

und

$$HOR = 2 \cdot VER \tag{8.2}$$

Wenn wir den Ausdruck $2 \cdot VER$ in der Gleichung (8.1) durch (8.2) ersetzen, erhalten wir:

$$\begin{aligned} \texttt{UM} &= 2 \cdot VER + 2 \cdot HOR \\ \texttt{UM} &= 3 \cdot HOR \\ \frac{\texttt{UM}}{3} &= HOR \end{aligned} \tag{8.3}$$

Aus (8.2) und (8.3) erhalten wir

$$VER \underset{(8.2)}{=} \frac{HOR}{2} \underset{(8.3)}{=} \frac{\texttt{UM}}{6}\,.$$

Mit der Formel zur Berechnung der Seitenlängen *VER* und *HOR* können wir nun das Programm schreiben:

```
to RE2ZU1 :UM
make "HOR :UM/3
make "VER :UM/6
RECHT :VER :HOR
end
```

□

Aufgabe 8.11 Die Aufgabe ist analog zu Beispiel 8.1, nur mit dem Unterschied, dass

a) die vertikalen und horizontalen Seiten gleich lang sind.

b) die horizontalen Seiten 3-mal so lang sind, wie die vertikalen Seiten.

Beim Rechnen muss man auch häufig die Quadratwurzel einer Zahl berechnen. Dazu gibt es den Befehl `sqrt` in LOGO. Mit dem Befehl

```
make "X sqrt :Y
```

wird die Wurzel des aktuellen Variablenwertes von `:Y` berechnet und im Register `:X` gespeichert. Kannst du mit Hilfe des Satzes von Pythagoras und dem Befehl `sqrt` die folgende Aufgabe lösen?

Aufgabe 8.12 Entwickle ein Programm zum Zeichnen der Klasse der rechtwinkligen gleichschenkligen Dreiecke mit wählbarer Schenkellänge.

Aufgabe 8.13 Entwickle ein Programm zum Zeichnen von drei Quadraten wie in Abb. 8.5. Dabei sind nur die Seitenlängen der beiden ersten Quadrate über Parameter `:A` und `:B` gegeben. Das dritte Quadrat muss die Eigenschaft haben, dass seine Fläche der Summe der Flächen der ersten zwei kleineren Quadrate entspricht. Für das Beispiel in Abb. 8.5 stimmt es, da $5^2 = 3^2 + 4^2$ gilt.

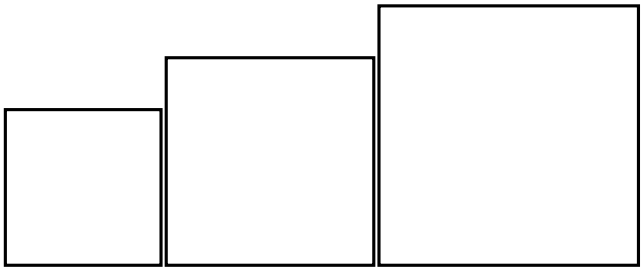


Abbildung 8.5

Aufgabe 8.14 Entwickle ein Programm mit drei Parametern `:A`, `:B` und `:C`, das eine Linie der Länge $X \cdot 10$ zeichnet, wobei X die Lösung der linearen Gleichung

$$\texttt{A} \cdot X + \texttt{B} = \texttt{C}$$

für $\texttt{A} \neq 0$ ist.

Wir können Programme mit Variablen als eine Transformation von gegebenen Eingabewerten in eine Ausgabe betrachten. Die beim Aufruf eines Programms gegebenen Variablenwerte bezeichnen wir bei dieser Sichtweise als **Eingaben** oder als **Eingabewerte**¨ des Programms. Als die **Ausgabe** für gegebene Eingaben bezeichnen wir das Resultat der Arbeit des Programms.Somit kann die Ausgabe eines Programms ein Bild, berechnete Werte gewisser Variablen, ein Text oder auch alles zusammen sein. Zum Beispiel, beim Aufruf

```
RE2ZU1 60
```

ist `60` die Eingabe. Die Ausgabe besteht aus den Werten 20 für `:HOR`, 10 für `:VER` und dem gezeichneten Rechteck der Größe 10×20. Ob wir die Werte 10 und 20 als Ausgaben ansehen wollen, ist unsere Entscheidung.

Aufgabe 8.15 Was sind die Eingaben und Ausgaben bei dem Aufruf

```
T3 1 (-10) 5?
```

Hinweis für die Lehrperson An dieser Stelle ist es empfehlenswert den Begriff der Funktion zu thematisieren. Ein Programm berechnet eine Funktion von so vielen Argumenten, wie die Anzahl seiner Eingaben ist. Eine Funktion beschreibt wie eine Blackbox eine Beziehung zwischen Eingabewerten (Argumenten) und Ausgabewerten (Funktionswerten). Ein Programm beschreibt explizit den Rechenweg von den Eingabewerten zu den entsprechenden Ausgabewerten.

Beispiel 8.2 Die Aufgabe ist es, eine frei wählbare Anzahl :AN von Kreisen mit wachsendem Umfang zu zeichnen. Dabei soll der Umfang des kleinsten Kreises durch einen Parameter :UM frei wählbar sein und desweiteren soll die Differenz im Umfang von zwei nacheinander folgenden Kreisen durch den Parameter :NACH frei bestimmbar sein (Abb. 8.6).

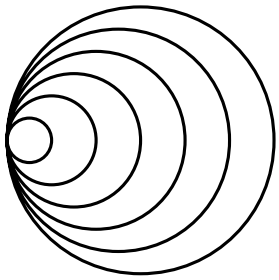


Abbildung 8.6

Wir wissen, dass unser Programm

```
to KREISE :LA
repeat 360 [ fd :LA rt 1 ]
end
```

Kreise mit dem Umfang 360·LA zeichnet. Wenn man es mit dem Wert :UM/360 aufruft, zeichnet es dann genau den Kreis mit dem Umfang :UM. Somit können wir KREISE folgendermaßen als Unterprogramm verwenden:

```
to AUGE :AN :UM :NACH
repeat :AN [ KREISE :UM/360 make "UM :UM+:NACH ]
end
```

□

Aufgabe 8.16 Welcher Wert liegt im Register `UM` nachdem das Programm `AUGE a u n` für die Zahlen `a`, `u` und `n` ausgeführt wurde? Welche Variablen in `AUGE` sind Parameter?

Aufgabe 8.17 Schreibe ein Programm, das genau zwölf Kreise mit wachsender Größe wie in Abb. 8.6 auf der vorherigen Seite zeichnet. Dabei sollen die Kreise vom kleinsten bis zu dem größten mit den Farben 1 bis 12 gezeichnet werden. Die Größe des kleinsten Kreises und der Größenzuwachs sollen frei wählbar sein.

Zusammenfassung

Variablen funktionieren ähnlich wie Parameter, aber zusätzlich können sie ihren Wert während der Ausführung des Programms ändern. Somit sind Parameter spezielle Variablen, die ihren Wert während des Laufs eines Programms nicht ändern. Die Änderung des Wertes einer Variablen wird durch den Befehl `make` erreicht. Der `make`-Befehl hat zwei Argumente. Das erste Argument ist durch `"` bezeichnet und besagt, welche Variable einen neuen Wert bekommt (in welchem Register das Resultat gespeichert werden soll). Das zweite Argument ist ein arithmetischer Ausdruck, in dem Operationen über Zahlen und Variablen vorkommen dürfen. Zu den grundlegenden arithmetischen Operationen gehört neben +, −, ∗ und / auch die Quadratwurzelberechnung. Der Befehl zur Berechnung der Wurzel heißt `sqrt`. Nach `sqrt` steht ein Argument. Das Argument kann eine Zahl, eine Variable oder ein beliebiger arithmetischer Ausdruck sein.

Alle Ausdrücke wertet der Rechner so aus, dass er zuerst alle Variablennamen durch ihre aktuellen Werte ersetzt und danach das Resultat berechnet. Wenn der Rechner aus einem Register `A` den Wert für `A` ausliest, ändert sich der Inhalt in diesem Register dabei nicht. Wenn er aber in einem Register `X` die neu berechnete Zahl abspeichert, wird der alte Inhalt des Registers `X` automatisch gelöscht. Die Variablenwerte eines Programmaufrufs bezeichnen wir auch als Eingaben des Programms und das Resultat der Arbeit eines Programms bezeichnen wir als die Ausgabe des Programms.

Kontrollfragen

1. Was ist der Hauptunterschied zwischen Variablen und Parametern?

2. Erkläre, wie der Befehl `make` funktioniert.

3. Was passiert, wenn man den Befehl `make "X ...` verwendet und keine Variable mit dem Namen `:X` in dem Programm bisher definiert wurde?

4. Besteht in LOGO eine Möglichkeit, gewisse Werte aus einer vorherigen Ausführung eines Programms in die nächste Ausführung des Programms zu übertragen?

5. Wie kann man eine Wurzel in LOGO berechnen?

6. Ändert sich der Inhalt eines Registers, wenn der Rechner den Inhalt liest und zur Berechnung verwendet?

7. Was passiert mit dem alten Inhalt eines Registers, wenn man in diesem Register einen neuen Wert speichert?

Kontrollaufgaben

1. Entwickle ein Programm zum Zeichnen einer frei wählbaren Anzahl `:AN` von Treppen, wobei die nächste Treppe immer um fünf größer sein soll als die vorherige (Abb. 8.7). Die Größe der ersten Treppe `:GR` soll frei wählbar sein. Teste Dein Programm für $AN = 5$ und $GR = 20$ und dokumentiere die Entwicklung der Speicherinhalte nach der Ausführung jedes einzelnen Befehls so wie in Tab. 8.1. Welche der Variablen in Deinem Programm sind Parameter?

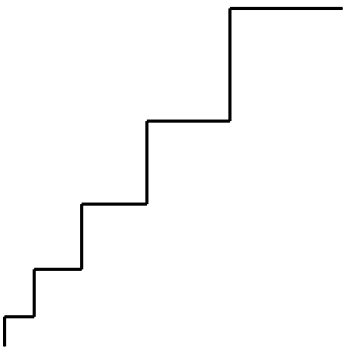


Abbildung 8.7

2. Entwickle ein Programm zum Zeichnen von Pyramiden wie in Abb. 8.8 auf der nächsten Seite. Die Anzahl der Stufen, die Größe der Basisstufe sowie die Reduzierung der Stufengröße beim Übergang in eine höhere Ebene sollen frei wählbar sein.

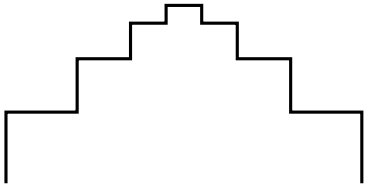


Abbildung 8.8

3. Was zeichnet das folgende Programm?

```
to SPIR :UM :ADD :AN
repeat :AN [ KREISE :UM/360 fd :UM/2 rt 20
             make "UM :UM+:ADD
             make "ADD :ADD+10 ]
end
```

Versuche die Frage zuerst ohne einen Testlauf zu beantworten. Welche der drei Variablen in SPIR sind Parameter? Teste das Programm mit dem Aufruf SPIR 50 20 10. Dokumentiere in einer Tabelle die Inhalte der drei Register UM, ADD und AN nach jedem der zehn Durchläufe der Schleife repeat. Was steht nach der Ausführung von SPIR u a b in den Registern UM, ADD und AN?

Schreibe ein Programm LINGL :A :B :C :D, das ein Quadrat mit der Seitenlänge $5 \times X$ zeichnet, wobei X die Lösung der Gleichung

$$A \cdot X + B = C \cdot X + D$$

für $A \neq C$ darstellt.

Teste das Programm mit dem Aufruf LINGL 3 100 2 150. Welche Variablen Deines Programms sind Parameter? Dokumentiere die Änderung der Inhalte der Register nach der Ausführung der einzelnen Befehle Deines Programms während des Testlaufs LINGL 4 50 4 100.

4. Zeichne eine sechseckige Spirale wie in Abb. 8.9 auf der nächsten Seite. Die Seitenlänge wächst vom Vorgänger zum Nachfolger immer um einen festen, aber frei wählbaren Betrag :ADD. Die erste Seitenlänge ist 50. Die Anzahl der Windungen der Spirale soll frei wählbar sein.

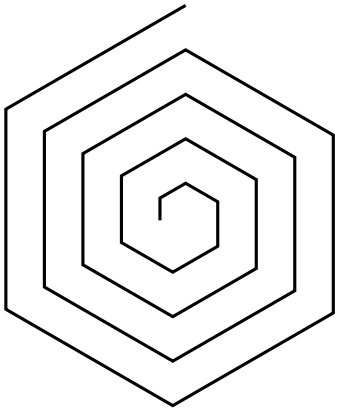


Abbildung 8.9

5. Ändere das Programm VIEQ1 :GR :AN, indem du den Befehl

```
make "GR :GR+10
```

durch den Befehl

```
make "GR :GR+:GR
```

austauschst. Welche Zahl liegt im Register GR nach der Ausführung von VIELQ1 10 10? Wie groß ist der Parameter :GR allgemein nach der Ausführung von VIELQ1 a b?

6. Entwickle ein Programm zum Zeichnen einer frei wählbaren Anzahl von Kreisen wie in Abb. 8.6 auf Seite 150. Dabei soll der Kreisumfang von Kreis zu Kreis immer um einen Faktor 1.2 wachsen. Die Größe des kleinsten Kreises soll frei wählbar sein.

7. Entwickle ein Programm zum Zeichnen einer frei wählbaren Anzahl von Halbkreisen wie in Abb. 8.10 auf der nächsten Seite. Dabei ist die Anzahl der Halbkreise mindestens 2. Der erste Halbkreis ist 100 Schritte und der zweite 120 Schritte lang. Die Länge jedes folgenden Halbkreises ist die Summe der Längen der zwei vorherigen Halbkreise.

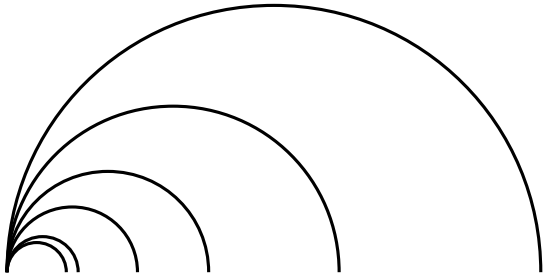


Abbildung 8.10

8. Lokale Parameter eines Hauptprogramms können ihre Werte während der Laufzeit des Hauptprogramms ändern. Trotzdem betrachten wir sie nicht als Variablen. Kannst du erklären warum nicht?

Lösungen zu ausgesuchten Aufgaben

Aufgabe 8.2
Wir nehmen einen neuen Parameter `:ST` für das Vergrößern der Seiten von Quadrat zu Quadrat. Damit können wir folgendermaßen aus `VIELQ` `VIELQ1` machen:

```
to VIELQ1 :GR :AN  :ST
repeat :AN [ QUADRAT :GR make "GR :GR+:ST ]
end
```

Wir sehen, dass es reicht, die Zahl `10` in `VIELQ` durch den Parameter `:ST` zu ersetzen.

Aufgabe 8.3
Wir bezeichnen mit `:GR` die Seitengröße des kleinsten gleichseitigen Dreiecks, durch `:AN` die Anzahl der Dreiecke und durch `:ST` das Vergrößern der Seitenlänge von Dreieck zu Dreieck. Dann kann unser Programm wie folgt aussehen:

```
to VIELDR :GR :AN  :ST
repeat :AN [ repeat 3 [ fd :GR rt 120 ] make "GR :GR+:ST ]
end
```

Aufgabe 8.4
Das Programm zum Zeichnen von Schnecken (Abb. 8.3 auf Seite 143) kann wie folgt arbeiten:

```
to SCHNECKE :GR :AN :ST
repeat :AN [ fd :GR rt 90 fd :GR rt 90 make "GR :GR+:ST ]
end
```

Aufgabe 8.6
Das Programm `T1` definiert in der `to`-Zeile keine Variablen. Dadurch wird der Variablen `:A` mit dem Aufruf des Programms `T1` kein neuer Wert zugeordnet. Somit verbleibt in `A` der alte Wert aus dem letzten Lauf des Programms `T1`. Damit wird das Register `A` nach X Aufrufen von `T1` die Zahl $X \cdot 50$ beinhalten.

Aufgabe 8.7
Sei -100 die Zahl im Register `X`. Der Befehl `fd :X` wird jetzt als „100 Schritte zurück“ interpretiert. Er entspricht in seiner Wirkung damit dem Befehl `bk 100`. Auf diese Weise werden bei negativen Lösungen Quadrate der Seitenlänge $|X|$ links unten gezeichnet. Bei positiven Lösungen werden $X \times X$-Quadrate rechts oben gezeichnet.

Aufgabe 8.12
Die Grundidee ist, dass zuerst die Winkelgrößen in einem gleichschenkligen rechtwinkligen Dreieck bekannt sein müssen (Abb. 8.11).

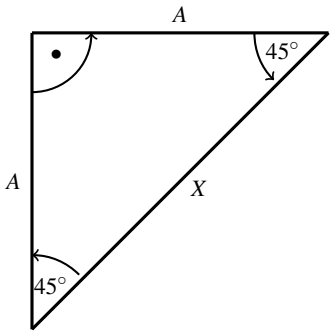


Abbildung 8.11

Der größte Winkel liegt zwischen den Schenkeln und beträgt 90°. Da in einem gleichschenkligen Dreieck zwei Winkel gleich groß sind und die Summe der Winkel in jedem Dreieck 180° beträgt, haben die beiden übrigen Winkel folglich jeweils 45°. Um das Dreieck mit Hilfe der Befehle `fd` und `rt` zu zeichnen, müssen wir noch die Länge X der Hypotenuse berechnen. Durch den Satz

des Pythagoras wissen wir:

$$X^2 = A^2 + A^2$$
$$X^2 = 2 \cdot A^2$$
$$X = \sqrt{2 \cdot A^2}$$

Somit kann das Dreieck in Abb. 8.11 auf der vorherigen Seite für ein gegebenes :A mit folgendem Programm REGELSCH gezeichnet werden:

```
to REGELSCH :A
fd :A rt 90 fd :A rt 90 rt 45
make "X sqrt 2*:A*:A
fd :X
end
```

Aufgabe 8.16
In jedem Durchlauf der Schleife repeat des Programms AUGE wird der Umkreis :UM um :NACH vergrößert. Somit ist nach :AN-vielen Durchläufen von repeat der Wert von :UM gleich

dem ursprünglichen Wert von UM + AN · NACH.

Für den Aufruf AUGE a u v bedeutet es, dass das Register UM nach der Ausführung des Programms mit den Parameterwerten a, u und v die Zahl

u + a · v

beinhaltet.

Kontrollaufgabe 5
Das Programm kann wie folgt arbeiten:

```
to SPIR6 :AN :ADD
make "BAS 50
repeat :AN [ fd :BAS rt 60 make "BAS :BAS+:ADD ]
end
```

Kontrollaufgabe 6
In jedem Durchlauf der Schleife repeat wird sich der Wert von :GR verdoppeln. Wenn am Anfang in :GR der Wert a stand, dann ist in :GR

nach dem ersten Schleifendurchlauf der Wert	$2 \cdot a$
nach dem zweiten Schleifendurchlauf der Wert	$4 \cdot a = 2 \cdot a + 2 \cdot a$
	$\vdots$
nach dem b-ten Schleifendurchlauf	$2^b \cdot a = 2^{b-1} \cdot a + 2^{b-1} \cdot a$

gespeichert. Falls $a = 10$ und $b = 10$ gilt, ist am Ende im Register GR die Zahl

$$2^{10} \cdot 10 = 1024 \cdot 10 = 10240$$

gespeichert.

Kontrollaufgabe 8
Die lokalen Parameter eines Hauptprogramms sind globale Parameter des Unterprogramms, indem sie definiert werden. Als solche ändern sich ihre Werte zur Laufzeit des Unterprogramms nicht. Wenn das entsprechende Unterprogramm aber mehrmals aufgerufen wird, können durch die Aufrufe die Werte der lokalen Parameter neu gesetzt werden.

Lektion 9

Lokale und globale Variablen

Das Konzept der Variablen ist eines der wichtigsten Programmierkonzepte. Mit ihm fängt das wahre Programmieren an. Es ermöglicht uns, eine Vielfalt von Rechneraktivitäten zu steuern. Um die Variablen korrekt zu verwenden, müssen wir noch lernen, wie die Datenübertragung zwischen Programmen und Unterprogrammen genau abläuft. Eigentlich haben wir schon bei den Parametern damit angefangen, uns mit diesem Thema zu beschäftigen. Wir haben gelernt, wie man über Parameter Eingaben als Zahlen von außen ins Programm eingibt und wie ein Hauptprogramm seine Parameterwerte an seine Unterprogramme weitergeben kann.

Ein schönes Beispiel zur Wiederholung ist das Hauptprogramm

```
to KR :A :B
RECHT :A :B
RECHT :B :A
rt 180
RECHT :A :B
RECHT :B :A
end
```

mit dem schon bekannten Unterprogramm:

```
to RECHT :VER :HOR
repeat 2 [ fd:VER rt 90 fd:HOR rt 90 ]
end
```

Beim Aufruf

J. Hromkovič, *Lehrbuch Informatik*, DOI 10.1007/978-3-8348-9692-6_10,

```
RECHT 100 200
```

werden die Eingabewerte 100 und 200 an das Programm übergeben. Das Register A enthält die Zahl 100 und das Register B die Zahl 200. Im Hauptprogramm KR wird das Unterprogramm RECHT viermal aufgerufen. Dabei werden immer abwechselnd dem Parameter :VER von KR die Werte der Parameter :A und :B übergeben. Das gleiche, nur in anderer Reihenfolge (:B, :A, :B, :A statt :A, :B, :A, :B), passiert mit dem Parameter :HOR des Unterprogramms KR.

Aufgabe 9.1 Simuliere den Lauf des Programms KR beim Aufruf KR 100 200 und dokumentiere den Inhalt aller Register nach der Bearbeitung aller Zeilen des Programms KR.

Die Variablen, die wir in einem Programm definieren, heißen **globale Variablen** des Programms. Zum Beispiel sind :A und :B globale Variablen des Programms KR. Die Parameter :VER und :HOR sind keine globalen Variablen des Programms KR, dafür aber die globalen Variablen des Programms RECHT. Damit halten wir fest, dass die Zeile

```
to XXX :A :B :C
```

automatisch die Variablen :A, :B und :C zu globalen Variablen des Programms XXX macht. Das ist aber nicht der einzige Weg, die globalen Variablen zu definieren. In Lektion 8 haben wir gelernt, mittels des Befehls make auch neue Variablen zu definieren. Damit ist im Falle eines Programms XXX

```
to XXX :A :B :C
⋮
make "D
⋮
end
```

die Variable :D auch eine globale Variable, weil sie im Programm XXX und nicht in einem seiner Unterprogramme definiert wurde.

Die globalen Variablen eines Unterprogramms nennen wir **lokale Variablen** des entsprechenden Hauptprogramms. Somit sind :VER und :HOR lokale Variablen des Programms KR.

Zum Beispiel sind :A, :B, :C, :X und :Y im Programm T3 aus der Lektion 8 globale

Variablen des Programms T3. Die Variable :GR (deren Name in der Beschreibung des Hauptprogramms gar nicht vorkommt) ist die globale Variable des Unterprogramms QUADRAT und somit die lokale Variable des Hauptprogramms T3. Der Aufruf QUADRAT :X macht :X nicht zu einer Variablen von QUADRAT. Die Bedeutung beschränkt sich auf die Übertragungen des aktuellen Wertes von :X an die Variable :GR des Programms QUADRAT :GR.

Aufgabe 9.2 Welche sind die globalen und lokalen Parameter des Programms AUGE aus Beispiel 8.2?

Aufgabe 9.3 Welche sind die globalen und lokalen Parameter des Programms SPIR aus der Kontrollaufgabe 3 in Lektion 8?

Haben wir damit alles geklärt? Leider noch nicht. Das Wesentliche kommt erst noch. Wir haben bisher nicht darauf geachtet, dass die Variablen bei unterschiedlichen Programmen anders genannt werden. Wir haben mehrere Hauptprogramme entworfen, deren Variablennamen den Namen der Variablen in den Unterprogramme gleichen. Ist dies nicht verwirrend?

Betrachten wir das Programm

```
to RE2ZU1 :UM
make "HOR :UM/3
make "VER :UM/6
RECHT :VER :HOR
end
```

aus Beispiel 8.1, welches das Programm RECHT :VER :HOR als ein Unterprogramm verwendet. Unserer bisherigen Ausführung folgend müssen die Variablen :UM, :HOR und :VER die globalen Variablen von RE2ZU1 sein. Die Variablennamen kommen aber auch in der Definition von

```
to RECHT :VER :HOR
```

vor und somit sind :VER und :HOR globale Variablen von RECHT. Dies bedeutet nach unserer Definition, dass :VER und :HOR lokale Variablen von RE2ZU1 sind. Kann aber eine Variable gleichzeitig lokal und global bezüglich des gleichen Programms sein?

Aufgabe 9.4 Finde andere Beispiele aus den bisher entwickelten Programmen, welche das Problem mit gleichnamigen globalen und lokalen Variablen haben.

Die Lösung mag ein bisschen überraschend sein. Der Rechner führt zwei unterschiedliche Register mit dem Namen VER und zwei unterschiedliche Register mit dem Namen HOR. In jedem Register merkt sich der Rechner nicht nur den Variablennamen, sondern auch den Namen des Programms, in dem die Variable definiert ist (d. h. für welches die Variable global ist). Das bedeutet, dass wir nicht eines, sondern zwei Register mit dem Namen HOR haben. Eines ist HOR(RE2ZU1) und dieses ist für die globale Variable des Programms RE2ZU1. Das zweite Register ist HOR(RECHT) und dieses ist für die globale Variable von RECHT und die lokale Variable von RE2ZU1. Der Speicher des Rechners sieht also wie in Tab. 9.1 aus.

Programmzeile	0	1	2	3
UM(RE2ZU1)	600	600	600	600
HOR(RE2ZU1)	–	200	200	200
VER(RE2ZU1)	–	–	100	100
HOR(RECHT)	0	0	0	200
VER(RECHT)	0	0	0	100

Tabelle 9.1

In Tab. 9.1 ist die Entwicklung der Speicherinhalte nach der Bearbeitung einzelner Programmzeilen von RE2ZU1 beim Aufruf RE2ZU1 600 dargestellt. Unmittelbar nach dem Aufruf liegt 600 im Register UM. Die Register HOR(RE2ZU1) und VER(RE2ZU1) existieren noch nicht, weil diese noch nicht definiert wurden. Diese Tatsache kennzeichnen wir mit dem Strich – in der Tabelle. Die Register HOR(RECHT) und VER(RECHT) existieren schon, weil das Programm RECHT schon definiert wurde und somit der Rechner die entsprechenden Register reserviert hat.

Nach der Bearbeitung der Zeile

```
make "HOR :UM/3
```

wird ein neues Register mit dem Namen HOR(RE2ZU1) angelegt und das Resultat 200 der Rechnung 600/3 in diesem Register abgespeichert. Die globale Variable :VER von

RE2ZU1 gibt es zu diesem Zeitpunkt noch nicht. Sie entsteht nach der Bearbeitung der Zeile

```
make "VER:UM/6
```

und erhält dabei den Wert 100. Nach dem Aufruf

```
RECHT :VER :HOR
```

wird der Wert von VER(RE2ZU1) in das Register VER(RECHT) übertragen. Analog geht die Zahl 200 aus HOR(RE2ZU1) in das Register HOR(RECHT) über. Die Daten werden immer aus den globalen Variablen entsprechend des Aufrufs RECHT :VER :HOR genommen. Der Speicherort für die Daten ist durch die Definition

```
to RECHT :VER :HOR
```

des Programms RECHT gegeben. Wenn man das Programm RECHT als

```
to RECHT :A :B
```

definiert hätte, hätten wir keine Probleme mit Doppelbenennungen gehabt. Der Wert des Registers VER(RE2ZU1) hätte man nach A und den Wert von HOR(RE2ZU1) nach B übertragen.

Aufgabe 9.5 Zeichne analog eine Tabelle wie in Tab. 9.1 für den Aufruf RE2ZU1 900.

Aufgabe 9.6 Bestimme die globalen und lokalen Variablen des Programms VIELQ1 aus Lektion 8. Zeichne eine Tabelle analog zu Tab. 9.1, welche die Entwicklung der Inhalte der Register nach der Durchführung einzelner Befehle darstellt.

An den Beispielen bis Lektion 6 sehen wir, dass die Inhalte von zwei Variablen mit gleichem Namen unterschiedlich sein dürfen. Am Ende waren die Werte aber immer gleich und der Unterschied hatte keine äußere Wirkung. Wenn wir die ganze Zeit für :VER (oder :HOR) nur ein Register verwendet hätten, hätte dies am Resultat (dem Bild) nichts geändert. Gibt es eine Möglichkeit, sich davon zu überzeugen, dass der Rechner tatsächlich zwei unterschiedliche Register für zwei Variablen mit gleichem Namen verwendet? Zu diesem Zweck bauen wir das folgende Testprogramm

```
to TEST2 :GR
fd :GR rt 90
TEST1 :GR
rt 90 fd :GR
end
```

mit dem Unterprogramm

```
to TEST1 :GR
fd :GR
make "GR :GR + 100
end.
```

überlegen wir nun, was beim Aufruf `TEST2 100` auf dem Bildschirm gezeichnet würde, wenn der Rechner nur ein Register für die Variable `:GR` verwenden würde. Das Resultat entspräche dann der Arbeit des Programms

```
TEST3 :GR
fd :GR rt 90
fd :GR
make "GR :GR+100
rt 90 fd :GR
end
```

das entstanden ist, indem man die Programmzeilen des Programms `TEST1 :GR` an die entsprechende Stelle im Programm `TEST2` eingesetzt hat.

Beim Aufruf `TEST3 100` zeichnet man das Bild aus Abb. 9.1(a). Zuerst werden zweimal die Linien der Länge 100 gezeichnet. Danach wird `:GR` um 100 auf 200 erhöht. Folglich wird mittels des letzten Befehls `fd :GR` die Linie der Länge 200 gezeichnet.

Tatsächlich zeichnet der Rechner beim Aufruf `TEST2 100` das Bild aus Abb. 9.1(b). Das kommt daher, weil der Befehl

```
make "GR :GR+100
```

in `TEST1` den Wert der lokalen Variable `GR`(`TEST1`) ändert und den Wert der globalen Variable `GR`(`TEST2`) unverändert lässt. Der letzte Befehl `fd :GR` des Programms `TEST2` spricht die globale Variable `GR`(`TEST2`) an und somit wird die Linie der Länge 100 gezeichnet. Die Entwicklung der beiden Registerinhalte nach der Ausführung einzelner

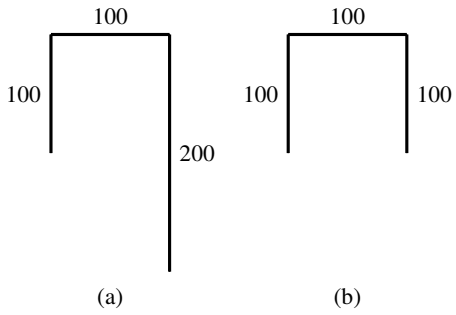


Abbildung 9.1

Zeilen des Programms TEST2 beim Aufruf TEST2 100 ist in Tab. 9.2 dargestellt.

Programmzeile	0	1	2	3
GR(TEST2)	100	100	100	100
GR(TEST1)	0	0	200	200

Tabelle 9.2

Aufgabe 9.7 In Tab. 9.2 sehen wir leider nicht im Detail die Änderungen von GR(TEST1), wo zuerst 100 zugewiesen wird und sich dann dieser Wert um 100 erhöht. Erstelle eine ausführliche Tabelle, welche die Speicherinhalte nach der Durchführung jedes einzelnen Befehls aufzeigt.

Es gibt auch eine andere Möglichkeit, sich von den richtigen Variablenwerten zu überzeugen. Mit dem Befehl print oder seiner kürzeren Form pr kannst du dir die Variablenwerte auf dem Bildschirm anzeigen lassen. Zum Beispiel schreibt

```
print :A
```

den aktuellen Wert der Variable :A in das Fenster, in dem du programmierst.

Wir verwenden den Befehl print dreimal in TEST2, um zu sehen, dass die Auswirkungen von print :GR im Unterprogramm auf die lokale Variable GR(TEST1) und auf die globale Variable GR(TEST2) unterschiedlich sind. Unsere modifizierten Programme sehen jetzt wie folgt aus:

```
to TEST2 :GR
fd :GR rt 90
print :GR
TEST1 :GR
print :GR
rt 90 fd :GR
end

to TEST1 :GR
fd :GR
make "GR :GR + 100
print :GR
end
```

Beim Aufruf

```
TEST2 100
```

erhältst du auf dem Bildschirm die Zahlen

```
100
200
100
```

Die erste und die dritte Zahl beträgt jeweils 100. Sie entsprechen den Ausführungen der Befehle `print :GR` im Hauptprogramm `TEST2`. Die Ausgabe 200 folgt auf Grund des Befehls `print :GR` im Unterprogramm `TEST1`, in dem die lokale Variable `GR`(`TEST1`) den Wert 200 hat.

Den Befehl `print` oder kurz `pr` kann man nicht nur zum Darstellen und damit zur Kontrolle von Variablenwerten verwenden. Wir können direkt Zahlen mit

```
print 7
```

darstellen. Hier wird die Zahl 7 erscheinen. Wir können auch eine Folge von Zahlen durch den Befehl

```
print [ 2 7 13 120 -6 ]
```

anzeigen lassen. Genauso gut können wir Texte schreiben. Mittels

```
print [ Es gibt keine Lösung ]
```

erscheint der Text „Es gibt keine Lösung“ auf dem Bildschirm. Probiere es aus.

Aufgabe 9.8 überlege dir ein eigenes Testprogramm mit einem Unterprogramm. Das Programm soll zwei Quadrate nebeneinander zeichnen. Beide sollen gleich groß sein, wenn man (wie es auch tatsächlich ist) zwischen den Variablen mit gleichem Namen unterscheidet. Wenn man aber statt des Aufrufs des Unterprogramms nur seinen Inhalt in das Testprogramm schreibt, dann sollen zwei unterschiedlich große Quadrate gezeichnet werden.

Aufgabe 9.9 Worin unterscheidet sich das Programm SPIR aus der Kontrollaufgabe 3 in Lektion 8 von folgendem Programm?

```
to SPIRT :UM :ADD :AN
repeat :AN [ KREISE :UM/360
             fd :UM/2 rt 20
             make "UM :UM+:ADD
             TEST3 :ADD ]
end

to TEST3 :ADD
make "ADD :ADD+10
end
```

Überprüfe deine Überlegungen mit den Aufrufen SPIRT 50 30 10 und SPIR 50 30 10. Erkläre, wo und wie der Unterschied verursacht wird. Zeichne für die Aufrufe SPIRT 50 30 3 und SPIR 50 30 3 die Entwicklung der Speicherinhalte aller globalen und lokalen Register der Programme SPIR und SPIRT. Betrachte dabei den Aufruf der Unterprogramme KREISE als einen Befehl. Benutze den Befehl print in beiden Programmen, um die Korrektheit deiner Erklärungen zu belegen.

Aufgabe 9.10 Was passiert, wenn du in Aufgabe 9.9 aus dem Programm SPIRT den Befehl

```
make "UM :UM+:ADD
```

auch in das Unterprogramm übernimmst? Streiche also diesen make-Befehl aus SPIRT und ersetze TEST3 durch das folgende Programm TEST4:

```
to TEST4 :ADD :UM
make "UM :UM+:ADD
make "ADD :ADD+10
end
```

Statt `TEST3 :ADD` ruft man im Programm `SPIRT` das Unterprogramm `TEST4 :ADD :UM` auf.

An dieser Stelle kann man nun fragen, ob es nicht manchmal verwirrend ist, den gleichen Namen für die Variablen des Unterprogramms und des Hauptprogramms zu verwenden. Natürlich kann man versuchen, solche Situationen zu vermeiden. Dies ist aber nicht immer leicht und manchmal kann es sogar unerwünscht sein. Wenn wir so bei einem größeren modularen Entwurf vorgehen, kann es auch mühsam sein, bei den vielen Variablen darauf zu achten. Manchmal kommen auch Situationen vor, wie wir sie oft bei der Verwendung von Parametern angetroffen haben, in denen wir durch einen gleichen Namen den Zusammenhang und die gleiche Bedeutung für das zu zeichnende Bild ausdrücken wollten. Das Ganze zeigt aber deutlich, dass man bei der modularen Herstellung von umfangreichen Programmen immer die Übersicht behalten muss. Aber die Art, wie der Rechner mit lokalen Variablen umgeht, ist für uns vorteilhaft. Sie schätzt uns vor unangenehmen Überraschungen, die dadurch entstehen können, dass man unbeabsichtigt den gleichen Namen für eine globale Variable verwendet, die schon vor langer Zeit in irgendeinem fast vergessenen Unterprogramm verwendet wurde.

Zusammenfassung

Globale Variablen sind alle Variablen eines Programms, die im Programm in der `to`-Zeile oder später mittels `make` definiert werden. Wenn ein Programm keine Unterprogramme hat, gibt es nur globale Variablen. Die globalen Variablen von Unterprogrammen eines Hauptprogramms sind die lokalen Variablen des Hauptprogramms. Wir nennen sie lokal, weil sie nur angesprochen und geändert werden können, während der Rechner das Unterprogramm ausführt.

Der Rechner speichert Variablennamen immer zusammen mit dem Namen des Programms, in welchem sie definiert wurden. Wenn man den gleichen Variablennamen in mehreren unterschiedlichen Programmen verwendet, handelt es sich immer um unterschiedliche Variablen. Aus der Sicht des Rechners hat jede Variable einen Namen, der aus zwei Teilen besteht. Der erste Teil entspricht dem eigenen Namen. Der zweite Teil ist der Name des Programms, in welchem die Variable definiert (global) wurde. Dadurch ordnet der Rechner Variablen mit identischem Namen, die in unterschiedlichen Programmen definiert wurden, in unterschiedliche Register ein. Ihre Werte können sich also unabhängig voneinander verändern.

Der Befehl `print :A` verursacht, dass der Wert der Variablen `:A` auf dem Bildschirm

angezeigt wird. Das kann man zur Kontrolle der korrekten Arbeit eines Programms gut verwenden. Mit `print` [...] kann man beliebige Texte oder Zahlenfolgen ausdrucken.

Kontrollfragen

1. Was sind globale Variablen eines Programms? Wie kann man sie definieren?
2. Was sind lokale Variablen eines Programms? Warum heißen sie lokal?
3. Welche Art von Variablen hat ein Programm ohne Unterprogramme?
4. Was alles „merkt sich“ der Rechner bei der Definition einer neuen Variable?
5. Was passiert, wenn unterschiedliche Programme Variablen mit gleichem Namen verwenden?
6. Was sind deiner Meinung nach die Vorteile des verwendeten Umgangs mit Variablen gegenüber der Identifizierung aller Variablen mit gleichem Namen? Warum ist es nicht vorteilhafter, gleiche Namen zu verbieten?
7. Wie funktioniert der Befehl `print`? Was alles kann man mittels `print` darstellen lassen?

Kontrollaufgaben

1. Welche Unterschiede gibt es zwischen den folgenden drei Programmen `VIELQ1`, `VIELQ2` und `VIELQ3`?

```
to VIELQ1 :GR :AN
repeat :AN [ QUADRAT :GR make "GR :GR+10 ]
end

to VIELQ2 :GR :AN
repeat :AN [ QUADRAT :GR ]
end


to VIELQ3 :GR :AN
repeat :AN [ QUADRAT :GR WACHSE10 :GR ]
end
```

```
to WACHSE10 :GR
make "GR :GR+10
end
```

Simuliere alle drei Programme mit Aufrufen für :GR = 100 und :AN = 3 und beschreibe die Entwicklung der Speicherinhalte jeweils nach der Ausführung eines einzelnen Befehls in einer Tabelle. Füge print Befehle in die Programme ein, um die Richtigkeit deiner Tabelle zu dokumentieren.

2. Das folgende Programm kann man verwenden, um schwarze Dreiecke zu zeichnen, deren Höhe ebenso groß ist wie die Breite (Abb. 9.2).

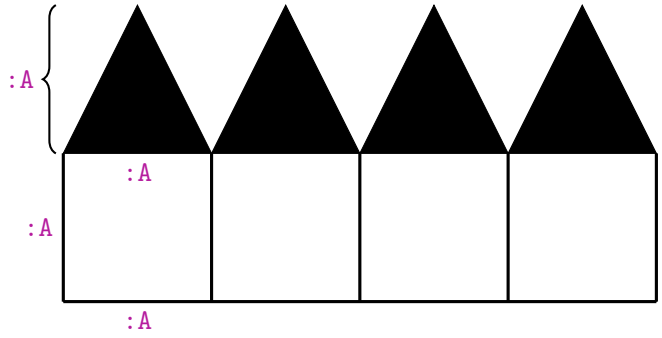


Abbildung 9.2

```
to SCHWDR :A
rt 90
repeat :A/2    [ fd :A rt 180 fd 0.5 rt 90
                 fd 1 lt 90 make "A :A-1
                 fd :A rt 180 fd 0.5 lt 90
                 fd 1 rt 90 make "A :A-1 ]
end
```

Verwende das Programm SCHWDR als Unterprogramm eines Programms STRASSE :AN :A zum Zeichnen einer wählbaren Anzahl :AN von Häusern (Abb. 9.2) mit einer frei wählbaren Breite :A. Bestimme die Werte der Variablen nach jedem Aufruf von SCHWDR :A und direkt nach der Ausführung des Unterprogramms SCHWDR durch den Aufruf STRASSE 70 4.

3. Das Programm SCHWDR hat in seiner Schleife eine lange Folge von Befehlen. Diese kannst du fast halbieren, wenn du eine Variable für eine beliebige frei wählbare Drehung einführst. Weißt du, wie es geht?

4. Zeichne eine beliebig lange Folge von Häusern wie in Abb. 9.2 auf der vorherigen Seite, jedoch mit dem Unterschied, dass der Wert der Variable `:A` von Haus zu Haus immer um 10 wächst.

5. Entwirf ein Programm zum Zeichnen beliebiger, farbig gefüllter, gleichschenkliger Dreiecke mit wählbarer Basis `:A` und wählbarer Höhe `:H`.

6. Betrachte das folgende Programm

```
to PFLANZE :LANG :HOCH :STEP :SUBL :SUBH
repeat :HOCH [ fd :STEP NADEL :LANG rt 3
               make "LANG :LANG-:SUBL
               make "STEP :STEP-:SUBH ]
end
```

   mit dem Teilprogramm

```
to NADEL :LANG
lt 45 fd :LANG bk :LANG rt 90
fd :LANG bk :LANG lt 45
end.
```

   Welche Variablen sind Parameter? Welche Variablen des Hauptprogramms `PFLANZE` sind global und welche lokal?

   Rufe `PFLANZE 100 50 15 2 0.3` auf. Erstelle eine Tabelle mit den Variablenwerten für die ersten drei Durchläufe der Schleife `repeat` des Hauptprogramms. Füge `print`-Befehle so ein, dass man die Entwicklung der Variablenwerte `:LANG` und `:STEP` beobachten kann.

7. Würde sich etwas in der Auswirkung des Programms `PFLANZE` aus der Kontrollaufgabe 6 ändern, wenn man den Befehl

```
make "LANG :LANG - :SUBL
```

   aus dem Hauptprogramm löschen und als letzten Befehl in das Unterprogramm `NADEL` schreiben würde? Begründe deine Antwort.

8. Zeichne einen Strauss von Pflanzen durch das mehrfache Verwenden des Programms `PFLANZE`. Die Anzahl der Pflanzen soll 5 sein und die Farbe sowie alle anderen Charakteristika jeder einzelnen Pflanze sollen frei wählbar sein. Zusätzlich soll der Winkel, unter welchem die einzelnen Pflanzen aus dem Boden wachsen, frei wählbar sein.

   Eine Möglichkeit ist es, die Pflanzen an unterschiedlichen Stellen wachsen zu lassen.

Schaffst du es, das Programm so zu schreiben, dass alle Pflanzen aus der gleichen Wurzel wachsen (d. h. vom gleichen Punkt starten)?

9. Zeichne die Pflanze so, dass ihre Blätter (Nadeln) mit der Zeit anstatt kürzer immer länger werden. Wie viel muss man dafür im Programm PFLANZE ändern? Oder geht es sogar ohne eine Änderung?

Lösungen zu ausgesuchten Aufgaben

Aufgabe 9.10
Im Unterprogramm TEST4 werden die beiden neuen Variablen :UM und :ADD mittels make definiert. Diese Variablen sind lokale Variablen des Unterprogramms SPIRT und ihre Wertänderungen haben keinen Einfluss auf die Werte der globalen Variablen UM(SPIRT) und ADD(SPIRT). Damit verhalten sich UM(SPIRT) und ADD(SPIRT) wie Parameter des Hauptprogramms SPIRT. Somit sind alle Kreise in der gezeichneten Spirale sowie die Abstände zwischen den Kreisen gleich groß.

Kontrollaufgabe 1
Das Programm VIELQ1 zeichnet :AN-viele Quadrate der Größen GR, GR + 10, GR + 20, ..., GR + (AN − 1) · 10. Das Programm VIELQ2 zeichnet :AN-mal das gleiche Quadrat mit der Seitenlänge :GR. Das Programm VIELQ3 macht genau das Gleiche wie das Programm VIELQ2. Die Variable GR(WACHSE10) wächst zwar um den Wert 10 im Unterprogramm WACHSE10, aber diese Änderung der lokalen Variable GR(WACHSE10) hat keinen Einfluss auf den Wert der globalen Variable GR(VIELQ3). Probiere es aus.

Kontrollaufgabe 8
Du musst das Programm PFLANZE gar nicht ändern. Es reicht aus, den Eingabewert für :SUBL negativ statt positiv zu belegen.

Lektion 10

Verzweigung von Programmen und `while`-Schleifen

Im Leben machen wir selten immer das Gleiche. Wir treffen oft Entscheidungen, die von den Umständen abhängen. Wir verfolgen oft unterschiedliche Strategien. Abhängig davon, was passiert, handeln wir. Wir wollen nun auch Programme schreiben, die je nach Situation oder abhängig von unseren Wünschen eine aus einer Vielfalt von Möglichkeiten ausgewählte Tätigkeit ausüben können. Zu diesem Zweck dient beim Programmieren der Befehl `if`. Die Struktur des Befehls `if` ist wie folgt:

```
if Bedingung [ Tätigkeit ]
```

Die Ausführung der Tätigkeit ist an die Bedingung gebunden. Wenn die Bedingung erfüllt ist, wird die Tätigkeit in den Klammern ausgeübt. Diese Tätigkeit kann einem beliebigen Programm entsprechen. Die Bedingungen können verschieden sein. Wenn wir als Bedingung `:A=7` schreiben, prüft der Rechner, ob der Wert der Variablen `A` gleich `7` ist. Wenn dies der Fall ist, wird die nachfolgende Tätigkeit ausgeübt. Wenn es nicht der Fall ist, wird die Tätigkeit nicht ausgeführt und der Rechner setzt die Arbeit mit dem Befehl des Programms fort, welcher `if` direkt folgt. Zum Beispiel wird durch den Befehl

```
if :A=7 [ setpc 1 SCHW100 ]
```

ein rot gefülltes 100×100-Quadrat gezeichnet, wenn der Wert der Variablen `:A` die Zahl `7` ist. Wenn der Wert von `:A` anders als `7` ist, wird die Schildkröte keine Aktivität ausüben. Der Wert der Variablen `:A` ändert sich bei der Überprüfung von `:A=7` nicht. Die Bedingung `:A=7` kann man auch als Fragestellung an den Rechner verstehen. Der

J. Hromkovič, *Lehrbuch Informatik*, DOI 10.1007/978-3-8348-9692-6_11,

Rechner überprüft, ob die Antwort „ja“ oder „nein“ lautet. Wenn die Antwort „ja“ ist, führt der Rechner das in eckigen Klammern stehende, nachfolgende Programm aus. In der Bedingung kann man zum Beispiel auch durch :A>7 fragen, ob der Wert von :A größer ist als 7 oder mittels :A<:B, ob der Wert von :A kleiner ist als der Wert von :B.

Beispiel 10.1 Ohne Parameter wäre ein Programm immer nur für das Zeichnen eines konkreten Bildes zuständig. Programme mit Parameter können ganze Klassen von Bildern zeichnen. Die Werte der Parameter entscheiden, welches konkrete Bild gezeichnet wird. Der Befehl if ermöglicht uns, Programme zu schreiben, die eine große Vielfalt von Bildern zeichnen können.

Betrachten wir die folgende Aufgabe: Ein Programm soll fähig sein, nach unserem Wunsch eines der folgenden Bilder zu zeichnen:

a) Eine grüne Linie wählbarer Länge :GR,

b) ein gelbes Quadrat mit wählbarem Umfang :GR,

c) einen roten Kreis mit wählbarem Umfang :GR und

d) ein orange gefülltes 100×100-Quadrat.

Unseren Wunsch äußern wir durch den Wert der Variable :WAS. Wenn :WAS=0 ist, wollen wir eine Linie der Länge :GR erhalten. Wenn :WAS=1 ist, soll ein Quadrat gezeichnet werden. Bei :WAS=2 soll ein Kreis gezeichnet werden und für :WAS=3 wollen wir das orange Quadrat erhalten. Für den Fall, dass :WAS>3 gilt, soll das Programm mitteilen, dass unser Wunsch nicht in seinem Repertoire steht.

Das folgende Programm KLASSE1 erfüllt diese Anforderungen:

```
to KLASSE1 :WAS :GR
if :WAS=0 [ setpc 2 fd :GR ]
if :WAS=1 [ setpc 3 QUADRAT :GR/4 ]
if :WAS=2 [ setpc 1 KREISE :GR/360 ]
if :WAS=3 [ setpc 13 SCHW100 ]
if :WAS>3 [ pr [ Sorry, falsche Nummer ] ]
end
```

Wir beobachten, dass die Erfüllung einer der Bedingungen die Erfüllung aller anderen

ausschließt. Somit wird bei der Ausführung des Programms `KLASSE1` höchstens ein Bild gezeichnet. □

Aufgabe 10.1 Teste das Programm `KLASSE1` für unterschiedliche Werte des Parameters `:WAS`. Was passiert beim Aufruf `KLASSE1 (-4) 159`? Kannst du es begründen?

Aufgabe 10.2 Entwirf ein Programm mit den Parametern `:WAS`, `:UM` und `:GR` zum Zeichnen folgender Klasse von Bildern. Dein Wunsch soll insbesondere mittels des Parameters `:WAS` geäußert werden.

Wenn `:WAS=0` gilt, soll ein Kreis mit dem Umfang `UM` gezeichnet werden. Bei `:WAS=1` soll eine Linie der Länge `:GR` gezeichnet werden. Für `:WAS=2` sollen `:UM`-viele Treppen der Größe `:GR` gezeichnet werden. Wenn `:WAS>2`, dann soll ein regelmäßiges `:WAS`-Eck mit der Seitengröße `:GR` gezeichnet werden. Für `:WAS<0` soll eine Fehlermeldung auf dem Bildschirm erscheinen.

Beispiel 10.2 Der Befehl `if` ist auch bei der Lösung vieler mathematischer Aufgaben sehr hilfreich. Wenn man die Lösungen von Gleichungen und Ungleichungen sucht, hängt es oft von den Parametern der Gleichungen oder Ungleichungen ab, wie viele Lösungen es gibt. Betrachten wir die quadratische Gleichung

$$Ax^2 + Bx + C = 0.$$

Eine konkrete quadratische Gleichung ist durch die Parameter A, B und C gegeben. Aus der Mathematik kennen wir folgende Methode zur Lösung von quadratischen Gleichungen:

1. Berechne $M = B^2 - 4 \cdot A \cdot C$.

2. Falls $M < 0$, gibt es keine reelle Lösung.
   Falls $M = 0$, gibt es eine Lösung

   $$x = \frac{-B}{2 \cdot A}.$$

   Falls $M > 0$, gibt es zwei Lösungen:

   $$x_1 = \frac{-B + \sqrt{M}}{2 \cdot A}$$

   $$x_2 = \frac{-B - \sqrt{M}}{2 \cdot A}.$$

Diese Methode funktioniert für alle A, B, C mit der Ausnahme $A = 0$. Dann aber handelt es sich um keine quadratische Gleichung.

Unsere Aufgabe ist es nun, für gegebene Werte von A, B und C die Werte der Lösungen mit pr auszugeben oder darzustellen, dass es keine Lösung gibt. Zusätzlich sollen Quadrate gezeichnet werden, deren Seitenlänge das Zehnfache des Betrags des Lösungswerts misst. Wenn die Lösung positiv ist, soll das Quadrat rechts oberhalb der Mitte stehen. Wenn die Lösung negativ ist, soll das Quadrat links unter der Mitte stehen.

Die Implementierung der beschriebenen Methode zur Lösung der quadratischen Gleichungen kann wie folgt aussehen:

```
to QUADMETH :A :B :C
if :A=0 [ pr [ keine quadratische Gleichung ] stop ]
make "M :B*:B - 4*:A*:C pr :M
if :M<0   [ pr [ keine reelle Lösung ] ]
if :M=0   [ make "X0 (-:B)/(2*:A)
            pr [ X0= ] pr :X0 QUADRAT 10*:X0 ]
if :M>0   [ make "X1 ((-:B)+sqrt:M)/(2*:A)
            make "X2 ((-:B)-sqrt:M)/(2*:A)
            pr [ X1= ] pr :X1 QUADRAT 10*:X1
            pr [ X2= ] pr :X2 QUADRAT 10*:X2 ]
end
```

Wichtig dabei ist, dass bei negativen Zahlen Klammern gesetzt werden. Zum Beispiel auch beim Aufruf: `QUADMETH 1 (-10) 25`.

Wir beobachten, dass wir im Programm den neuen Befehl stop verwendet haben. Die Auswirkung des Befehls stop ist klar. Der Rechner beendet sofort die Ausführung des Programms, d. h. er beendet seine Arbeit. Das passt uns gut, weil er im Fall :A=0 auch nicht weiterarbeiten soll.

Wir bemerken auch, dass man bei der Berechnung der Werte von :X0, :X1 und :X2 die zur Berechnung bestimmten Ausdrücke in Klammern setzt, um dem Rechner mitzuteilen, in welcher Reihenfolge er die arithmetischen Operationen ausführen soll. Zum Beispiel kann man

```
:B/2*:A
```

einerseits als

```
(:B/2)*:A
```

und andererseits richtig als

```
:B/(2 * :A)
```

interpretieren. Dem Rechner muss man immer eindeutig mitteilen, was zu tun ist. Deswegen verwenden wir Klammern in arithmetischen Ausdrücken. □

Hinweis für die Lehrperson An dieser Stelle lohnt es sich, die Prioritäten der einzelnen arithmetischen Operationen bei der Auswertung von arithmetischen Ausdrücken in Erinnerung zu rufen. Allgemein ist man aber gut beraten, wenn man im Zweifelsfall die Klammerung verwendet. Unnötige zusätzliche Klammern verursachen keine Fehler. Fehlende Klammern können zu Fehlern führen, die nur sehr schwer zu entdecken sind.

Aufgabe 10.3 Teste das Programm `QUADMETH` für verschiedene Werte (Eingaben) `:A`, `:B` und `:C`, so dass jeder der vier möglichen Fälle mindestens einmal vorkommt.

Nimm den Befehl `stop` aus dem Programm heraus. Was würde deiner Meinung nach beim Aufruf `QUADMETH 0 1 1` jetzt passieren? Überlege zuerst und probiere es dann aus. An was aus dem Mathematikunterricht erinnert es dich?

Aufgabe 10.4 Modifiziere das Programm `QUADMETH`, so dass es immer Kreise mit dem Umfang zeichnet, der hundertmal dem Betrag der Lösung entspricht. Für eine positive Lösung soll der Kreis rechts von der Mitte stehen, für eine negative links von der Mitte.

Aufgabe 10.5 Entwickle ein Programm zur Lösung linearer Gleichungen $A \cdot X + B = C$, die durch die Werte der Parameter A, B und C bestimmt sind. Achte darauf, dass du drei Möglichkeiten hast: keine Lösung, eine Lösung oder unendlich viele Lösungen. Vergiss nicht, auch die Eingaben mit A=0 in Betracht zu ziehen und korrekt zu behandeln.

Warum sprechen wir bei der Verwendung des Befehls `if` von **Verzweigungen**? Weil wir mittels `if` aus mehreren Möglichkeiten eine auswählen. Das Vorhandensein mehrerer Möglichkeiten bei der Ausführung des Programms sehen wir als Verzweigung an. Die Wahl entsprechend der Bedingung entspricht dann der Verfolgung des entsprechenden Zweiges. Somit kann die Verzweigungsstruktur des Programms `QUADMETH` wie in Abb. 10.1 auf der nächsten Seite angedeutet werden.

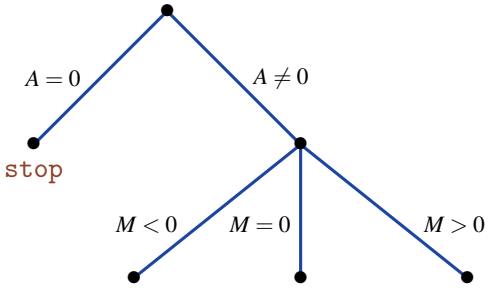


Abbildung 10.1

Eine andere Möglichkeit ist es, die Struktur aus Abb. 10.2 auf der nächsten Seite zu zeichnen, indem das Programm wie folgt modifiziert wird:

```
to QUADMETH1 :A :B :C
if :A=0 [ pr [ keine quadratische Gleichung ] stop ]
make "M :B*:B-4*:A*:C pr:M
 if :M<0   [ pr [ keine reelle Lösung ] stop ]
 if :M=0   [ make "X0 (-:B)/(2*:A)
             pr [ X0= ] pr :X0 QUADRAT 10*:X0
             stop ]
 if :M>0   [ ... ]
end
```

Hier haben wir stop-Befehle so eingeführt, dass bei der Erfüllung der formulierten Bedingung das entsprechende Programm in der Klammer ausgeführt und damit die Ausführung des Hauptprogramms beendet wird. Dies bedeutet, dass die folgenden if-Befehle gar nicht ausgeführt werden. Offensichtlich löst das Programm QUADMETH1 die quadratische Gleichung und kann als eine andere Implementierung der Lösungsmethode angesehen werden. In Abb. 10.2 auf der nächsten Seite entspricht jeder Verzweigungspunkt genau einer Verzweigung der Operation if. Weil rechts unten in Abb. 10.2 auf der nächsten Seite im letzten Knoten alle drei Bedingungen $A \neq 0$, $M \geq 0$ und $M \neq 0$ gelten, ist es offensichtlich, dass $M > 0$ gilt.

Aufgabe 10.6 Wie wir oben sehen, müssen wir den letzten if-Befehl im Programm QUADMETH mit der Bedingung $M > 0$ gar nicht verwenden. Aber das Ersetzen von `if :M>0 [ ... P ... ]` durch P zur Berechnung von :X1 und :X2 würde dann falsch funktionieren. Warum? Kannst du den letzten if-Befehl doch herausnehmen und dafür stop-Befehle so setzen, dass das modifizierte Programm korrekt läuft?

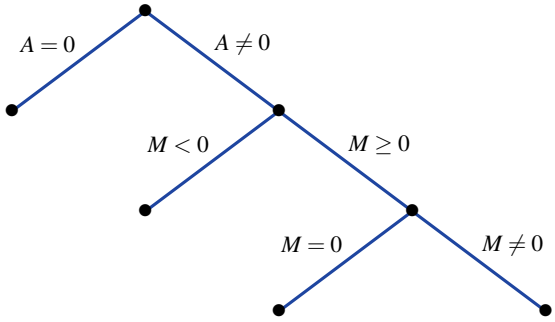


Abbildung 10.2

Aufgabe 10.7 Zeichne das Verzweigungsmuster für das Programm `KLASSE1`. Verwende beide Möglichkeiten entsprechend Abb. 10.1 auf der vorherigen Seite und Abb. 10.2, indem du für die zweite Variante die `stop`-Befehle entsprechend einführst.

Manchmal wollen wir mehrere Bedingungen gleichzeitig erfüllt haben, um etwas zu unternehmen. So etwas kann zum Beispiel bei der Lösung allgemeiner linearer Gleichungen

$$A \cdot X + B = C \cdot X + D$$

vorkommen. Wenn wir zu beiden Seiten $-C \cdot X - B$ addieren, erhalten wir die Gleichung in der Form

$$A \cdot X - C \cdot X = D - B.$$

Nach dem Distributivgesetz gilt

$$(A - C) \cdot X = D - B.$$

Jetzt kommt die Diskussion[1].

1. Wenn $A - C = 0$ (wenn $A = C$) und $D - B = 0$ ($D = B$), dann erhalten wir

   $$0 \cdot X = 0.$$

   Diese Gleichung gilt für alle X und somit sind alle reellen Zahlen Lösungen.

[1] Erinnere dich daran, dass sich die Lösungsmenge einer Gleichung nicht ändert, wenn man zu beiden Seiten die gleiche Zahl addiert oder wenn man beide Seiten mit der gleichen Zahl multipliziert.

2. Wenn $A-C=0$ und $D-B\neq 0$ gelten, dann erhalten wir

$$\begin{aligned} 0\cdot X &= D-B \\ 0 &= D-B. \end{aligned}$$

Die Zahl 0 kann aber nicht gleich einer Zahl $D-B\neq 0$ sein. Damit gibt es in diesem Fall keine Lösung.

3. Wenn $A-C\neq 0$, dann multiplizieren wir die Gleichung mit $\frac{1}{(A-C)}$ und erhalten:

$$X=\frac{D-B}{A-C}$$

In diesem Fall ist es die einzige Lösung der Gleichung.

Eine mögliche Implementierung dieser Fallunterscheidung ist die folgende:

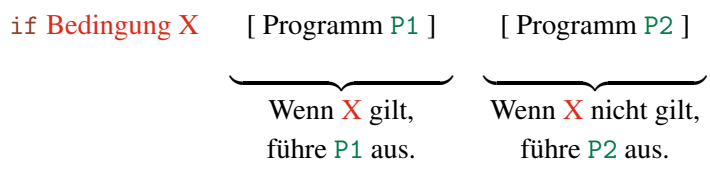
```
to LINGL :A :B :C :D
if :A=:C [ if :B=:D [ pr [ alle reellen Zahlen ] stop ] ]
if :A=:C [ pr [ keine Lösung ] stop ]
make "X (:D-:B)/(:A-:C)
pr [ X= ] pr :X if :X>0 [ LEITER :X ]
end
```

Aufgabe 10.8 Welche der Variablen von LINGL sind Parameter? Welche Variablen sind global und welche lokal?

Wir sehen, dass das Programm im Fall $A=C$ und $B=D$ zuerst die Nachricht „alle reellen Zahlen“ ausgibt und dann die Arbeit beendet. Danach fragt es im Falle, dass $A=C$ und $B=D$ nicht gleichzeitig gelten, ob $A=C$ gilt. Es ist sinnvoll, denn die Behauptung „$A=C$ und $D=B$ gelten nicht gleichzeitig“ entspricht dem Satz „Es gilt entweder $A\neq C$ oder $B\neq D$“. Wenn jetzt die Bedingung $A=C$ erfüllt ist, muss zwangsläufig $B\neq D$ gelten. Damit entspricht die Zeile 2 von LINGL dem Fall $A=C$ und $B\neq D$. Nach der Bearbeitung dieses Falls hört der Rechner wegen des stop-Befehls mit der Arbeit auf. Wenn $A\neq C$ gilt, hat der Rechner bisher keine Tätigkeit (außer der Überprüfung der Ungültigkeit der Bedingung $A=C$) ausgeübt und setzt die Arbeit mit der dritten Zeile des Programms fort.

Aufgabe 10.9 Das Programm LINGL zeichnet rechts eine Leiter mit X Stufen, falls X eine positive Lösung der linearen Gleichung ist. Erweitere das Programm, so dass es für $X<0$ eine Leiter mit $-X$-vielen Stufen links von der Mitte und für $X=0$ einen Kreis mit dem Umfang 100 zeichnet.

Der Befehl `if` erlaubt auch eine andere Struktur:



Das bedeutet, dass immer genau eines der Programme `P1` und `P2` ausgeführt wird. Wenn die Bedingung X erfüllt wird, wird `P1` ausgeführt. Wenn die Bedingung X nicht erfüllt wird, wird `P2` ausgeführt. Diese Struktur eignet sich sehr gut, wenn man eine Auswahl aus genau zwei Möglichkeiten treffen soll. Wenn man zum Beispiel ein Programm zum Zeichnen von Kreisen und Quadraten des Umfangs `:UM` haben will, kann man wie folgt vorgehen:

```
to QUADRKR :UM :WAS
if :WAS=0 [ QUADRAT :UM/4 ] [ KREISE :UM/360 ]
end
```

Wenn wir den Parameter `:WAS` auf `0` setzen, erhalten wir ein Quadrat. Für alle anderen Werte des Parameters `:WAS` zeichnet das Programm einen Kreis.

Aufgabe 10.10 Schreibe ein einzeiliges Programm, welches abhängig von einem Parameter entweder ein Sechseck der Seitenlänge 100 oder ein Achteck der Seitenlänge 50 zeichnet.

Die Struktur `if` Bedingung [...] [...] des Befehls `if` kann man für beliebige Verzweigungen verwenden. Zum Beispiel können wir das Programm `LINGL` wie folgt verändern:

```
to LINGL1 :A :B :C :D
if :A=:C [ if :B=:D [ pr [ alle Zahlen ] ]
                    [ pr [ keine Lösung ] ] ]
         [ make "X (:D-:B)/(:A-:C)
           pr [ X= ] pr :X if :X>0 [ LEITER :X ] ]
```

Programme mit dieser Struktur des Befehls `if` haben eine eindeutige Verzweigungsstruktur, in der jeder Verzweigung einer Auswahl von zwei Möglichkeiten entspricht.

Hinweis für die Lehrperson Die Struktur if Bedingung [...] [...] zieht man eindeutig in der strukturierten Programmierung vor. Sie entspricht dem bekannten if ... then ... else in höheren Programmiersprachen.

Die Verzweigungsstruktur des Programms LINGL1 ist in Abb. 10.3 veranschaulicht. Oben ist beim ersten if die Verzweigung bezüglich der Werte von A und C dargestellt. Für $A = C$ unterscheiden wir noch die Fälle $B = D$ und $B \neq D$. Für $A \neq C$ haben wir genau eine Lösung. Was das Zeichnen betrifft, unterscheiden wir ebenfalls die Fälle $X > 0$ und $X \leq 0$.

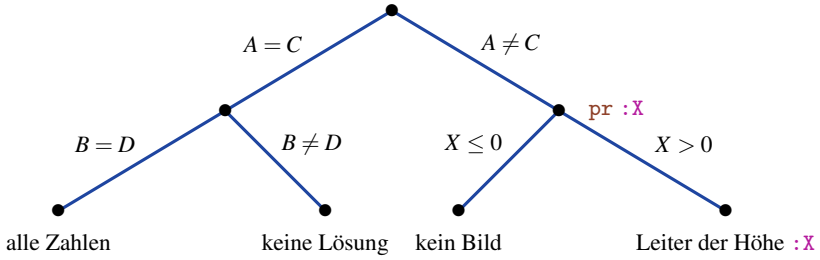


Abbildung 10.3

Die Struktur der if-Befehle in LINGL1 lässt sich wie folgt anschaulich darstellen

```
if [ if [   ] [   ] ] [ ... if [   ] ]
```

Wir sehen hier die Beziehung zwischen dieser Klammersetzung und der Verzweigungsstruktur in Abb. 10.3.

Aufgabe 10.11 Schreibe das Programm KLASSE1 so um, dass nur die if-Befehle mit der Struktur

```
if   Bedingung   [ ... ]  [ ... ]
```

vorkommen. Zeichne dazu die entsprechende Verzweigungsstruktur.

Aufgabe 10.12 Die Aufgabenstellung ist hier die gleiche wie in Aufgabe 10.11 für das Programm QUADMETH, jedoch soll zudem der Befehl stop vermieden werden.

Aufgabe 10.13 Ersetze in allen deinen bisherigen Programmen, in denen du `if` und `stop` kombiniert hast, die `if`-Struktur `if [ ]` durch die Struktur `if [ ] [ ]`, sodass keine `stop`-Befehle mehr gebraucht werden.

Aufgabe 10.14 Als Eingabe erhält man drei positive Zahlen. Es soll ein Kreis gezeichnet werden, dessen Umfang dem größten (maximalen) Eingabewert entspricht. Um diese Aufgabe zu lösen, muss man zuerst den maximalen der drei Eingabewerte bestimmen. Eine Möglichkeit besteht darin, die Strategie der Vergleiche aus Abb. 10.4 zu verfolgen. Implementiere diese Strategie und verwende dabei nur den `if`-Befehl mit der Struktur `if` Bedingung [...] [...].

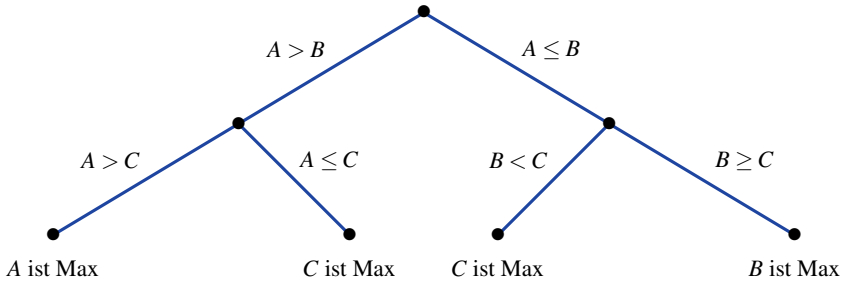


Abbildung 10.4

Beispiel 10.3 Es gibt noch eine andere Möglichkeit, den Befehl `if` zusammen mit dem Befehl `stop` zu verwenden. Wir wollen eine Tätigkeit so lange ausüben, bis eine gewisse Variable einen konkreten Wert nicht überschreitet. Betrachten wir folgende Aufgabe: Es soll eine sechseckige Schnecke (Abb. 8.9 auf Seite 154) von außen nach innen gezeichnet werden. Die Seitenlänge am Anfang ist mittels der Variablen `:LA` frei wählbar. Die Länge verkürzt sich immer um einen frei wählbaren Parameterwert `:SUB`. Die Arbeit soll dann enden, wenn entweder schon 100 Linien gezeichnet sind oder der Wert der Variablen `:LA` unter den Wert von `:SUB` gefallen ist, sprich die Seitenlänge nicht mehr um den Wert `:SUB` verkürzt werden kann (also der ursprüngliche Wert von `:LA` zu klein war). Das folgende Programm setzt dies um:

```
to SPIRBED :LA :SUB
repeat 100 [ fd :LA rt 60
             make "LA :LA-:SUB pr :LA
             if :LA-:SUB < 0 [ pr [ LA zu klein ] stop ] ]
end
```

Wir sehen, dass das Programm die wiederholte Ausführung der Schleife durch `stop`

vorzeitig beendet, wenn :LA so klein ist, dass man es nicht mehr um :SUB verkleinern kann. □

Aufgabe 10.15 Die gezeichnete Spirale (Abb. 8.9 auf Seite 154) ist sechseckig. Erweitere das Programm SPIRBED zu einem Programm SPIRECK :LA :SUB :ECK mit einer wählbaren Anzahl von Ecken.

Aufgabe 10.16 Zeichne die Spirale aus Abb. 8.9 auf Seite 154 von innen nach außen. Die innere Seitenlänge :LA sowie die Verlängerung :ADD sind frei wählbar. Das Programm soll enden, wenn eine der beiden folgenden Bedingungen erfüllt ist: 200 Linien sind bereits gezeichnet, oder die letzte Linie ist länger als 300.

Aufgabe 10.17 Entwickle ein Programm zum Zeichnen mehrerer Halbkreise von einem Punkt aus, wie in Abb. 8.10 auf Seite 155 dargestellt. Der Umfang (die Länge) des größten Halbkreises ist durch eine Variable :LA gegeben. Durch den Parameter :RED soll der Umfang immer um den Faktor :RED verkleinert werden. Das Programm fängt mit dem Zeichnen des größten Halbkreises der Länge :LA an, danach kommt der Halbkreis der Länge LA/RED usw. Das Programm soll spätestens nach dem Zeichnen von zehn Halbkreisen enden. Es soll aber schon vorher stoppen, wenn die Länge des zu zeichnenden Halbkreises kleiner als 50 ist.

Aufgabe 10.18 Du sollst das Programm PFLANZE aus der Kontrollaufgabe 6 in der Lektion 9 so erweitern, dass es aufhört zu arbeiten, wenn die Nadeln (Blätter) eine negative Länge erhalten.

Aufgabe 10.19 Wenn :RED eine positive Zahl kleiner als 1 ist, wachsen die Halbkreise im Programm zur Aufgabe 10.17. Kann man das Programm in diesem Fall so erweitern, dass keine Halbkreise länger als 1000 gezeichnet werden?

Mit Hilfe der Befehle if und stop haben wir es geschafft, dem Rechner folgendes zu sagen:

> *Arbeite so lange, bis eine Bedingung nicht mehr erfüllt ist oder bis die angegebene Anzahl der Wiederholungen einer Schleife ausgeführt ist.*

Wäre es aber nicht einfacher und manchmal auch praktischer, einfach zu sagen:

> *Arbeite so lange, bis diese Bedingung nicht erfüllt ist.*

Zum Beispiel: Zeichne eine sechseckige Spirale, bis die Länge der Linien nicht kleiner ist als 10. Der Vorteil wäre, dass man dabei nicht künstlich die Schleife repeat mit

hinreichend vielen Wiederholungen verwenden muss. Wir streben also eine neuartige Schleife folgender Art an:

> *Wiederhole Programm P (Körper der Schleife) bis die Bedingung B nicht mehr gilt (solange die Bedingung gilt).*

Den Bedarf nach so einer Schleife haben die Programmierer mittels des Befehls `while` umgesetzt. Die Struktur ist wie folgt:

```
while [ Bedingung ] [ Programm ].
```

Solange die Bedingung gilt, wird das Programm (der Körper der Schleife) wiederholt. Hier muss man aber vorsichtig sein. Wenn das Programm keine Variablenwerte aus der Bedingung ändert, wird die Bedingung immer gelten und das Programm wird ewig arbeiten. Zum Beispiel wird das Programm

```
QWHILE :A
while [ :A>0 ] [ QUADRAT :A ]
end
```

bei einem Aufruf `QWHILE a` für jede positive Zahl `a` unendlich oft das Quadrat der Seitengröße `a` zeichnen.

Also benutzen wir `while` nur dann, wenn durch die Ausführung des Programms garantiert wird, dass nach einer endlichen Anzahl von Schleifenwiederholungen die Bedingung nicht mehr gelten wird. Auf diese Weise können wir Spiralen mit dem folgenden Programm `SPIREND` einfacher als mit `SPIRBED` zeichnen.

```
to SPIREND :LA :SUB
while [ :LA>:SUB ] [ fd :LA rt 60 make "LA :LA-:SUB ]
end
```

Es wird gezeichnet, bis die Linienlänge so kurz ist, dass man sie um `:SUB` nicht mehr kürzen kann. Wenn man die Spirale von innen nach außen zeichnen will, kann man wie folgt vorgehen:

```
to SPIRIN :KURZ :ADD :MAX
while [ :KURZ<:MAX ] [ fd :KURZ rt 60 make "KURZ :KURZ+:ADD
]
end
```

Aufgabe 10.20 Zeichne die viereckige Schnecke aus Abb. 8.3 auf Seite 143. Die Werte der Parameter :ST und :GR sind als Eingaben gegeben. Das Programm soll zeichnen, bis die Länge der Linien den Wert 300 übersteigt.

Aufgabe 10.21 Zeichne mit einem Programm Quadrate so nebeneinander, wie es in Abb. 8.5 auf Seite 149 dargestellt ist. Das größte Quadrat soll dabei eine Seitenlänge von 200 haben. Die Seitengröße soll sich von Quadrat zu Quadrat halbieren. Das Programm soll das Zeichnen beenden, wenn die Seitenlänge kleiner als 1 ist.

Aufgabe 10.22 Zeichne Pflanzen mit einer Modifikation des Programms PFLANZE, indem du den Parameter :HOCH entfernst. Die Pflanze soll gezeichnet werden, bis die Blattlänge kürzer als 10 ist. Kann es Aufrufe deines Programms geben (kann es Parameterwerte geben), bei denen dein Programm unendlich lange arbeitet?

Aufgabe 10.23 Schreibe das Programm QUADRAT :GR so um, dass es statt der repeat-Schleife die while-Schleife verwendet.

Mit der while-Schleife kann man auch geschickt rechnen. Wenn man für eine gegebene Zahl n die Zahl

$$n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 2 \cdot 1$$

berechnen will, kann man wie folgt vorgehen:

```
to FAK :N
make "FA :N
while [:N>1 ] [make "N :N-1 make "FA :FA*:N ]
pr :N pr :FA
end
```

Aufgabe 10.24 Was passiert, wenn man die Bedingung :N>1 mit der Bedingung :N>2 vertauscht? Zeichne in eine Tabelle die Änderungen der Variablenwerte von :N und :FA nach jedem einzelnen Durchlauf der while-Schleife beim Aufruf FAK 6 ein.

Der Wert von $n!$ kann aber auch auf folgende Art berechnet werden:

```
to FAK1 :N
make "FA 1 make "M 1
while [ :M<:N ] [ make "M :M+1 make "FA :FA*:M ]
pr :N pr :FA
end
```

Aufgabe 10.25 Wo ist der Unterschied zwischen `FAK` und `FAK1`? Simuliere die Arbeit von `FAK1` beim Aufruf `FAK1 6`. Wäre es möglich, $n!$ ohne `while`-Schleife mit der `repeat`-Schleife zu berechnen?

Aufgabe 10.26 Die Fibonacci-Zahlen sind wie folgt definiert:

$$F(1) = 1,\ F(2) = 1 \qquad \text{und} \qquad F(n+1) = F(n) + F(n-1).$$

Das bedeutet $F(3) = F(2) + F(1) = 2$, $F(4) = F(3) + F(2) = 2 + 1 = 3$, usw. Schreibe ein Programm, dass für eine gegebene Zahl n die n-te Fibonacci-Zahl $F(n)$ berechnet.

Zusammenfassung

Der Befehl `if` ermöglicht uns, abhängig von den Werten unserer Variablen unterschiedliche Tätigkeiten auszuführen. Die Wahl einer Aktivität aus bestehenden Möglichkeiten wird durch die Erfüllung oder Nichterfüllung der Bedingung nach `if` getroffen. Wir sprechen in diesem Zusammenhang von der Verzweigung von Programmen. Der Befehl `if` kann zwei unterschiedliche Strukturen haben. Der Befehl

```
if Bedingung [ Programm ]
```

führt nur dann zur Ausführung des Programms, wenn die Bedingung erfüllt ist. Nach der Ausführung des Programms setzt der Rechner die Arbeit mit dem nächsten Befehl fort. Wenn die Bedingung nicht erfüllt ist, setzt der Rechner sofort die Arbeit mit der Ausführung des nächsten Befehls fort. Der Befehl

```
if Bedingung [ P1 ] [ P2 ]
```

führt entweder zur Ausführung des Programms `P1` oder des Programms `P2`. Das Programm `P1` wird ausgeführt, wenn die Bedingung erfüllt ist. Sonst wird `P2` ausgeführt.

Der Befehl `stop` ermöglicht uns, an beliebiger Stelle des Programms sofort mit der Ausführung aufzuhören und damit die Arbeit zu beenden. In Kombination mit `if` kann man dann Bedingungen an das Beenden der Programmausführung stellen.

Der Befehl `while` steuert eine Schleife, deren Anzahl von Wiederholungen zu Beginn nicht angegeben ist. Die Wiederholung der Schleife endet, wenn die Bedingung nach

while nicht mehr erfüllt ist. Also wird die Schleife so lange wiederholt, wie die Bedingung nach while noch gilt. Bei der Verwendung von while-Schleifen muss man aufpassen, dass es nicht zur endlosen Wiederholung einer Tätigkeit kommt. Die while-Schleife eignet sich besonders zum Zeichnen von Mustern, die so lange gezeichnet werden sollen, bis der Bildschirm zur Darstellung nicht mehr ausreicht oder bis die Linien so kurz sind, dass man sie nicht mehr sehen kann.

Kontrollfragen

1. Wie sieht die Struktur des Befehls if aus? Welche Arten von Bedingungen können wir formulieren?

2. Seien P1 und P2 zwei Programme. Wie sieht das Programm aus, das uns ermöglicht, auszuwählen, welches der beiden Programme ausgeführt werden soll?

3. Warum sprechen wir im Zusammenhang mit dem Befehl if von der Verzweigung von Programmen?

4. Was hat die while-Schleife mit der Kombination der beiden Befehle if und stop gemeinsam?

5. Wie sieht die Struktur der while-Schleife aus?

6. Wozu eignet sich die while-Schleife besonders gut?

7. Kann man immer eine repeat-Schleife durch eine while-Schleife ersetzen?

Kontrollaufgaben

1. Schreibe ein Programm SORTQUAD :A :B :C, das beim Aufruf SORTQUAD a b c drei Quadrate mit den Seitenlängen a, b und c wie in Abb. 8.5 auf Seite 149 zeichnet. Dabei muss das kleinste Quadrat ganz links und das größte ganz rechts stehen. Damit erzeugen die Aufrufe SORTQUAD 5 4 3, SORTQUAD 5 3 4 und SORTQUAD 4 5 3 das gleiche Bild wie Abb. 8.5 auf Seite 149.

2. Schreibe ein Programm, bei dem man mittels eines Parameters auswählen kann, ob man eine Pflanze oder ein Schachfeld zeichnet.

3. Schreibe ein Programm `MINMAX :A :B :C :D`, das die folgende Ausgabe beim Aufruf `MINMAX a b c d` liefert:

   MIN = „Minimum {a,b,c,d}“ MAX = „Maximum {a,b,c,d}“.

4. Betrachte das folgende Programm,

   ```
   to PROG :AN
   repeat :AN [ P1 ]
   end
   ```

   wobei `P1` ein beliebiges Programm darstellt. Kannst du das Programm so umschreiben, dass du dabei den Befehl `repeat` durch den Befehl `while` austauschst und das Programm weiterhin die gleiche Tätigkeit ausübt?

5. Schreibe ein Programm, das die Kreise aus Abb. 8.6 auf Seite 150 zeichnen kann. Dabei soll der kleinste Kreis einen Umfang von 70 haben. Der Umfang des nachfolgenden Kreises soll immer `:FAK`-mal größer werden, wobei `:FAK` ein Parameter des Programms mit wählbarem Wert ist. Das Zeichnen soll aufhören, wenn der Umfang größer als 1000 wird. Versuche, das Programm einmal mit der `while`-Schleife und einmal ohne Verwendung der `while`-Schleife zu schreiben.

6. Entwickle ein Programm zum Zeichnen der Pyramiden aus Abb. 8.8 auf Seite 153. Die erste Stufe soll eine wählbare Größe `:GR` haben. Die Stufen sollen sich nach oben hin immer um 20 Schritte verkleinern. Die Spitze der Pyramide ist erreicht, wenn die nächste Stufe kleiner als 25 ist.

7. Zeichne eine Folge von Quadraten wie in Abb. 8.1 auf Seite 140 abgebildet. Das kleinste Quadrat hat eine wählbare Größe `:GR` und die Seitenlänge vergrößert sich um einen wählbaren Wert `:ADD`. Mit dem Zeichnen soll das Programm aufhören, wenn das letzte Quadrat die Seitenlänge 250 überschreitet.

8. Das folgende Programm berechnet für eine gegebene Zahl n den Funktionswert $f(n)$.

   ```
   to FUN1 :N
   make "F1 1 make "M 0
   while [ :N>:M ] [ make "F1 :F1*2 make "M :M-1 ]
   pr [ N= ] pr :N pr [ F1(n)= ] pr :F1
   SPIR 20 :N :F1
   end
   ```

   Um welche Funktion handelt es sich? Kannst du das Programm so umschreiben, dass keine `while`-Schleife verwendet wird?

9. Nimm das Programm SPIR :UM :ADD :AN und modifiziere es wie folgt: Der Parameter :AN wird durch einen neuen Parameter :MAX ersetzt. Statt Kreise mit dem aktuellen Umfang :UM sollen Quadrate mit dem Umfang :UM gezeichnet werden. Die Spirale soll so lange gezeichnet werden, wie der Umfang der gezeichneten Quadrate kleiner als :MAX ist.

10. Entwickle ein Programm zum Zeichnen von Pflanzen wie beim Programm PFLANZE. Die seitliche Neigung der Pflanze ist in PFLANZE nach dem Zeichnen eines Blätterpaars durch rt 3 gegeben. Bei dir soll jedoch die Neigung frei wählbar sein. Die Zeichnung soll genau dann aufhören, wenn die Schildkröte nicht mehr nach oben schaut, also wenn sie horizontal steht oder nach unten rechts schaut.

Lösungen zu ausgesuchten Aufgaben

Aufgabe 10.1
Beim Aufruf KLASSE1 -4 159 gibt es keine Ausgabe. Es wird kein Bild gezeichnet und auch kein Text geschrieben. Das kommt daher, dass das Programm nur aus 5 if-Befehlen besteht und für :WAS=-4 keine der entsprechenden fünf Bedingungen erfüllt ist.

Aufgabe 10.6
Wenn wir im Programm QUADMETH den letzten if-Befehl if :M>0 [P] herausnehmen und nur durch P ersetzen, wird P unter allen Umständen für $A \neq 0$ ausgeführt. Dies bedeutet, dass das Programm auch für $M < 0$ versuchen würde, die Lösungen :X1 und :X2 zu berechnen. Das wird aber nicht gehen, weil die Wurzel aus negativen Zahlen nicht definiert ist. Probiere es aus! Wenn man aber wie in QUADMETH1 die stop-Befehle einführt, darf man den letzten if-Befehl weglassen. es wird nämlich die letzte Bedingung if :M>0 nur dann erfüllt, wenn $M < 0$ und $M = 0$ nicht gelten und somit sicher $M > 0$ gilt. Deshalb ist in diesem Fall die Frage, ob $M > 0$ gilt, unnötig.

Aufgabe 10.8
Die vier globalen Variablen :A, :B, :C und :D, die man für die Eingabe verwendet, sind Parameter. Die globale Variable :X ist kein Parameter. Die globale Variable :ANZ des Programms LEITER ist die einzige lokale Variable des Programms LINGL.

Aufgabe 10.23
Das Programm

```
to QUADRAT :GR
repeat 4 [ fd :GR rt 90 ]
end
```

verwendet die repeat-Schleife. Wir können sie durch eine while-Schleife ersetzen, indem wir eine neue Variable :N einführen. Am Anfang setzen wir ihren Wert auf 4 und nach jedem

Durchlauf der Schleife verkleinern wir :N um 1. Dann reicht es, die Bedingung while :N>0 zu nehmen und das Programm ist fertig.

```
to QUADRATW :GR
make "N 4
while [ :N>0 ] [ fd :GR rt 90 make "N :N-1 ]
end
```

Aufgabe 10.26
Um die n-te Fibonacci-Zahl zu berechnen, kann man mit $F(1)$ und $F(2)$ anfangen und $n-1$ Male die rekursive Formel $F(n+1) = F(n) + F(n-1)$ verwenden. Das Programm kann wie folgt aussehen:

```
to FIB :N
if :N<3 [ pr [ 1 ] stop ]
make "FNEXT 1 make "FPR 1
repeat :N-2 [ make "X :FNEXT
              make "FNEXT :FNEXT+:FPR
              make "FPR :X ]
pr :FNEXT
end
```

Kannst du das Programm so umschreiben, dass kein stop-Befehl darin vorkommt? Kannst du die repeat-Schleife durch eine while-Schleife ersetzen?

Kontrollaufgabe 1
Eine Idee wäre, die Werte der Parameter :A, :B und :C aufsteigend zu sortieren und in :X1, :X2 und :X3 zu speichern, wobei :X1 der kleinste und :X3 der größte Wert wäre. Danach könnte man wie üblich für die Variablen :X1, :X2 und :X3 die Quadrate zeichnen. Um festzustellen, welches der kleinste oder größte Wert von :A, :B und :C ist, muss man die Zahlen mittels if-Befehlen vergleichen. In diesem Fall ist man gut beraten, nicht mit dem Schreiben des Programms zu beginnen, sondern zuerst eine Strategie für Variablenvergleiche festzulegen. Unsere Strategie ist in Abb. 10.5 auf der nächsten Seite skizziert.

Jede Verzweigung in Abb. 10.5 auf der nächsten Seite entspricht einem Vergleich von zwei Werten. Wir vergleichen so lange, bis die Reihenfolge der Größen von A, B und C klar ist. Zum Beispiel ergibt die Gültigkeit von $A < B$ und $B < C$ direkt die einzige mögliche Reihenfolge $A < B < C$. Aber die Gültigkeit von $A \geq B$ und $B < C$ im Zweig ganz rechts ermöglicht es uns noch nicht, einen definitiven Schluss zu ziehen, weil die Beziehung zwischen A und C unklar ist. Deswegen vergleichen wir noch A und C und erhalten dadurch eine der beiden Möglichkeiten $B \leq A < C$, wenn $A < C$ gilt und $B < C \leq A$, wenn $A \geq C$ gilt. Diese Vergleichsstrategie können wir wie folgt in einem Programm umsetzen:

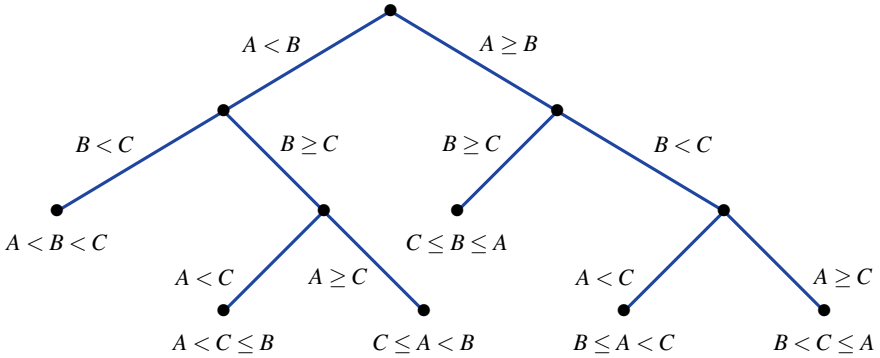


Abbildung 10.5

```
to SORTQ   :A :B :C
if :A<:B   [ if :B<:C   [ pr [ A<B<C ] make "X1 :A
                          make "X2 :B make "X3 :C ]
             [ if :A<:C   [ pr [ A<C<=B ] make "X1 :A
                          make "X2 :C make "X3 :B ]
                          [ pr [ C<=A<B ] make "X1 :C
                          make "X2 :A make "X3 :B ] ] ]
[ if :B<:C [ if :A<:C   [ pr [ B<=A<C ] make "X1 :B
                        make "X2 :A make "X3 :C ]
                        [ pr [ B<C<=A ] make "X1 :B
                        make "X2 :C make "X3 :A ] ]
                        [ pr [ C<=B<=A ] make "X1 :C
                        make "X2 :B make "X3 :A ] ]
QU4 :A :B :C 0
end
```

Um die Struktur besser zu überblicken, kann man das Programm kürzer darstellen. Wir verzichten auf das Drucken und auch auf die Hilfsvariablen :X1, :X2, und :X3. Stattdessen permutieren wir immer je nach Fall die Parameter :A, :B und :C beim Aufruf von QU4.

```
to SORTQ1   :A :B :C
if :A<:B    [ if :B<:C   [ QU4 :A :B :C 0 ]
             [ if :A<:C   [ QU4 :A :C :B 0 ]
                          [ QU4 :C :A :B 0 ]]]
[if :B<:C   [ if :A<:C   [ QU4 :B :A :C 0 ]
                          [ QU4 :B :C :A 0 ]]
                          [ QU4 :C :B :A ]]
end
```

Lektion 11

Integrierter LOGO- und Mathematikunterricht: Geometrie und Gleichungen

Viele Anwendungen der Computertechnologie basieren auf der Programmierung von mathematischen Methoden. Beispiele wie das Lösen von linearen oder quadratischen Gleichungen haben wir schon gezeigt. Wenn man mathematische Lösungswege programmiert, muss man sie sehr gut verstehen, weil ein Programm eine so genaue Beschreibung eines Vorgangs ist, dass sogar die dumme „Maschine“ ohne Intellekt die beschriebene Tätigkeit korrekt ausüben kann. Wenn du aber jemandem ohne Improvisationsfähigkeit etwas eindeutig klarmachen willst, musst du selbst die zu vermittelnde Tätigkeit besonders gut beherrschen. Dies ist auch ein Grund, uns hier mit dem Programmieren von Methoden zum Lösen unterschiedlicher Aufgaben beschäftigen zu wollen. Dadurch üben wir gleichzeitig Programmieren und gewinnen ein tieferes Verständnis für die mathematische Vorgehensweise. Du wirst sehen, dass man ohne Kenntnisse aus der Mathematik gar nicht anfangen kann, die Suche nach einer Lösung zu programmieren.

Wir fangen damit an, dass wir zuerst recht einfache Konstruktionen und Objekte zeichnen lernen. Wir haben schon gelernt, einen Kreis mit einem gegebenen Umfang zu zeichnen. Wir wollen aber auch lernen, Kreise mit einem gegebenen Radius zu zeichnen. Das Programm `KREISE :UM/360` zeichnet einen Kreis mit dem Umfang `:UM`. Wir verwenden den Aufruf `KREISE 10/360`, um Punkte anzudeuten, die aber in der mathematischen Modellierung die Größe 0 haben.

J. Hromkovič, *Lehrbuch Informatik*, DOI 10.1007/978-3-8348-9692-6_12,

Wir kennen die Formel

$$Umfang = 2 \cdot \pi \cdot r$$

zur Berechnung eines Kreisumfangs mit dem Radius r. Wenn wir π[1] mit 3.1416 annähern, kann das Programm zum Zeichnen eines Kreises mit dem Radius :Rwie folgt aussehen:

```
to KREISRAD :R
make "UM 2 * 3.1416 * :R
KREISE :UM/360
end
```

Dieser Kreis wird ausgehend aus dem am weitesten links liegenden Punkt des Kreises gezeichnet und an diesem Punkt beendet auch die Schildkröte ihre Tätigkeit. Manchmal kann es aber wünschenswert sein, dass der Kreis um die Schildkröte (als Mittelpunkt des Kreises) herum gezeichnet wird, wie mit einem Zirkel. Das können wir mit dem folgenden Programm erreichen:

```
to KREISMITT :R
KREISE 10/360
pu lt 90 fd :R rt 90 pd
KREISRAD :R
pu rt 90 fd :R lt 90 pd
end
```

Das Programm bewegt die Schildkröte so, dass sie am Ende wieder den Mittelpunkt des Kreises erreicht. Zusätzlich hat das Programm mit dem Befehl `KREISE 10/360` den Mittelpunkt des Kreises durch einen Punkt markiert.

Aufgabe 11.1 Für zwei gegebene Zahlen *R* und *M* zeichne zwei Kreise mit dem Radius *R*, deren Mittelpunkte auf einer horizontalen Linie in der Entfernung *M* liegen (Abb. 11.1). Welche Schnittpunktmengen können diese beiden Kreise haben?

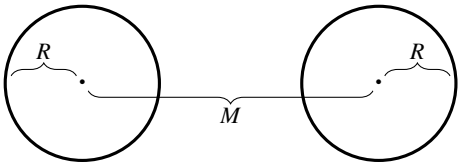


Abbildung 11.1

[1] In XLOGO kann der Befehl `pi` dafür verwendet werden.

Aufgabe 11.2 Entwickle ein Programm zum Zeichnen der Olympischen Ringe (Abb. 11.2). Der Radius der Kreise sollte frei wählbar sein.

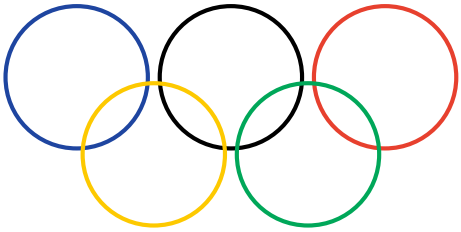


Abbildung 11.2

Die Fähigkeit, Kreise aus ihrem Mittelpunkt zu zeichnen, gehört zu den grundlegenden Operationen bei der Konstruktion von geometrischen Objekten mittels Zirkel und Lineal. Linien beliebiger Länge kann die Schildkröte auch zeichnen, aber nur unter der Voraussetzung, dass der Winkel der Linie zu einer horizontalen (oder vertikalen) Linie bekannt ist. Mit dem Lineal kann man zwei beliebige Punkte leicht verbinden. Die Schildkröte kann es nur dann, wenn die Entfernung der Punkte und der Winkel zwischen dem Horizont und der Linie berechnet werden kann. Mit den bisher bekannten Befehlen und ohne Trigonometrie ist es nicht immer möglich. Manchmal kann uns aber der neue Befehl `home` helfen. Egal, wo die Schildkröte sich befindet, zeichnet sie vom aktuellen Punkt (aus der aktuellen Position) eine Linie zu dem Startpunkt in der Mitte des Bildschirms.

Beispiel 11.1 Wir wollen ein rechtwinkliges Dreieck zeichnen. Die Bezeichnungen der Seiten, Ecken und Winkel verwenden wir immer wie in Abb. 11.3. Die Länge der Hypotenuse wird immer als c bezeichnet und $\gamma = 90°$.

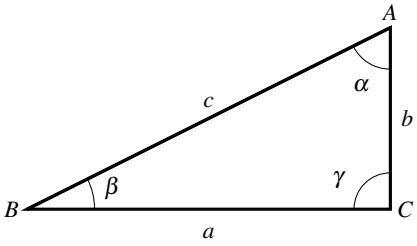


Abbildung 11.3

Wenn die Seitenlängen von a und b bekannt sind, kann man das rechtwinklige Dreieck mit dem folgenden Programm einfach zeichnen:

```
to DREI90 :a :b
rt 90 fd :a lt 90 fd :b home
end
```

Unsere Aufgabe ist aber ein bisschen allgemeiner. Wir wollen ein Programm entwickeln, das auf drei Eingaben arbeitet. Zwei davon sind positive Zahlen und entsprechen den jeweiligen Seitenlängen. Eine der Zahlen ist 0 und deutet darauf hin, dass die entsprechende Länge unbekannt ist. Wir wollen die fehlende Länge ausrechnen, alle Seitenlinien in der Reihenfolge a, b, c drucken und das Dreieck zeichnen. Zur Berechnung der fehlenden Seitengröße verwenden wir den Satz des Pythagoras

$$c^2 = a^2 + b^2.$$

Daraus folgt:

$$c = \sqrt{a^2 + b^2}$$
$$a = \sqrt{c^2 - b^2}$$
$$b = \sqrt{c^2 - a^2}.$$

Diesen Rechenweg kann man folgendermaßen interpretieren:

```
to KONSRECHTTR :a :b :c
if :a=0 [ make "a sqrt (:c*:c-:b*:b)
          DREI90 :a :b
          pr :a pr :b pr :c home stop ]
if :b=0 [ make "b sqrt (:c*:c-:a*:a)
          DREI90 :a :b
          pr :a pr :b pr :c home stop ]
if :c=0 [ make "c sqrt (:a*:a+:b*:b)
          DREI90 :a :b
          pr :a pr :b pr :c home stop ]
end
```

□

Aufgabe 11.3 Das Programm KONSRECHTTR funktioniert reibungslos auf korrekten Eingaben. Was passiert aber, wenn mehr als eine Eingabe 0 oder eine Eingabe negativ ist? Was wird gezeichnet, wenn a>c oder b>c ist? Modifiziere das Programm so, dass es statt zu rechnen und

zu zeichnen eine Fehlermeldung ausgibt, wenn die Eingabe nicht korrekt ist. Das Programm KONSRECHTTR ist für korrekte Eingaben unnötig lang. Kannst du es wesentlich verkürzen?

Hinweis für die Lehrperson Hier verwenden wir das erste Mal kleine Buchstaben für die Bezeichnung von Variablen. Die Programmiersprache LOGO unterscheidet bei Variablennamen kleine und große Buchstaben nicht. Der Name ANNA und der Name anna sind für Logo zwei identische Namen. Wir verwenden kleine Buchstaben nur deswegen, weil wir uns an die bekannte und eingeübte Bezeichnung aus der Mathematik halten wollen.

Aufgabe 11.4 Du sollst ein Programm entwickeln, das beliebige Dreiecke mittels des SWS-Satzes konstruiert. Das heißt, das Programm bekommt als Eingabe zwei Seitenlängen und die Größe des von diesen Seiten eingeschlossenen Winkels. Nach der Ausführung des Programms soll das Dreieck auf dem Bildschirm erscheinen.

Aufgabe 11.5 Schreibe ein Programm, das Dreiecke nach dem WSW-Satz konstruiert. Gegeben sind eine Seitenlänge und die Größen der beiden anliegenden Winkel. Genau wie bei der Konstruktion mit einem Lineal dürfen die gezeichneten Seiten mit unbekannter Länge über den Eckpunkt hinausgehen, also länger sein als ihre tatsächliche Länge im Dreieck. Hauptsache sie kreuzen sich und es entsteht das gewünschte Dreieck.

Aufgabe 11.6 Entwickle ein Programm zum Zeichnen von Dreiecken nach dem SWW-Satz. Gegeben sind also die Länge einer Seite, die Größe eines anliegenden Winkels und des Winkels, der der Seite gegenüber liegt. Das Programm darf natürlich rechnen, bevor es zu zeichnen anfängt.

Ohne komplizierte Rechnung können wir nicht alle Dreieckskonstruktionen einfach so programmieren. Deswegen sind wir schon damit zufrieden, wenn aus der Zeichnung offensichtlich ist, wo die drei Punkte A, B und C liegen, auch wenn nicht alle Seiten vollständig gezeichnet worden sind.

Beispiel 11.2 Unsere Aufgabe ist es, Dreiecke nach dem SSS-Satz zu konstruieren. Gegeben sind also die Längen aller drei Seiten des Dreiecks. Beim Konstruieren, fängt man damit an, die Seite $\overline{AB}$ mit der Länge c horizontal zu zeichnen. Danach zeichnen wir einen Kreis mit dem Mittelpunkt B und dem Radius $a = |\overline{BC}|$ und einen weiteren Kreis um den Mittelpunkt A mit dem Radius $b = \left|\overline{AC}\right|$ (Abb. 11.4 auf der nächsten Seite). Die zwei Schnittpunkte dieser Kreise bestimmen die Punkte C_1 und C_2. Es spielt keine Rolle, welchen von ihnen wir verwenden, weil die beiden entsprechenden Dreiecke kongruent sind.

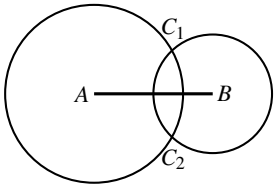


Abbildung 11.4

Unsere Implementierung dieser Konstruktion sieht wie folgt aus:

```
to DREIECKSSS :a :b :c
KREISMITT :b
rt 90 fd :c lt 90
KREISMITT :a
end
```

□

Aufgabe 11.7 Für welche Werte von a, b und c kann man kein Dreieck konstruieren? Erweitere das Programm DREIECKSSS in dem Sinne, dass es für ungeeignete Werte von a, b und c die Fehlermeldung „Es gibt kein Dreieck mit den Seitenlängen a, b, c!“ ausgibt.

Noch einfacher zu zeichnen sind parallele Linien der Länge :LA, die in einer Entfernung :DIST verlaufen.

```
to PARALLEL :DIST :LA
rt 90 fd :LA/2 bk :LA fd :LA/2
lt 90 pu fd :DIST pd
rt 90 fd :LA/2 bk :LA fd :LA/2
end
```

Aufgabe 11.8 Entwirf ein Programm zum Zeichnen einer Linie mit beliebiger Länge, die in der Entfernung :DIST von der aktuellen Position der Schildkröte verläuft (Abb. 11.5).

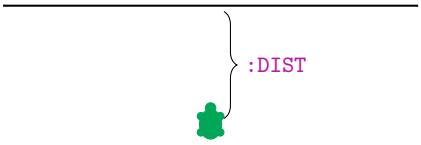


Abbildung 11.5

Aufgabe 11.9 Konstruiere mittels eines Programms die drei Punkte eines rechtwinkligen Dreiecks $\triangle ABC$. Die Eingaben sind die Seitenlänge c und die Entfernung `:DIST` des Punktes C von der Seite $\overline{AB}$.

Aufgabe 11.10 Entwirf ein Programm, das die drei Punkte A, B und C eines beliebigen Dreiecks für gegebene Größen c, a und h_c konstruiert. Die Zahl h_c ist die Höhe des Dreiecks auf die Seite c (die Entfernung zwischen C und der Seite $\overline{AB}$). Siehe auch Abb. 11.6.

Aufgabe 11.11 Entwirf ein Programm, das die drei Eckpunkte eines Dreiecks für gegebene Größen c, α und h_c konstruiert.

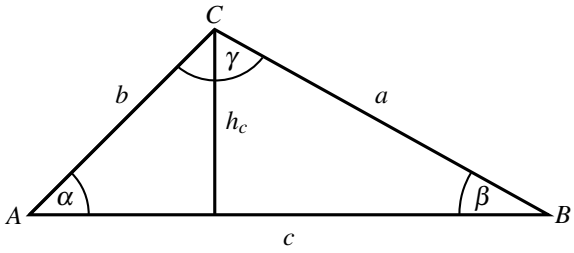


Abbildung 11.6

Wir sehen, dass die Programme die Suche nach neuen Informationen automatisieren. Die Eingaben sind immer konkrete Informationen über ein oder mehrere Objekte. Die Aufgabe ist es, weitere Daten aus diesen Eingaben zu berechnen. Also ist es genau das, was Mathematiker tun und wir versuchen, ihre Methoden zu automatisieren, in dem wir für deren Ausführung Programme entwickeln.

Beispiel 11.3 Die Aufgabe ist es, ein Rechteck der Größe $a \times b$ zu zeichnen (Abb. 11.7 auf der nächsten Seite). Die Werte a und b sind aber nicht bekannt. Wir wissen nur, dass der Umfang die Länge `:UM` hat und dass die vertikale Seite b `:C`-mal größer als a sein soll. Für gegebene `:UM` und `:C` soll das Rechteck gezeichnet werden.

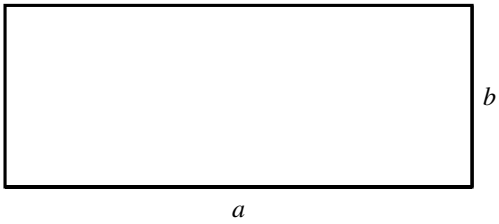


Abbildung 11.7

Bevor wir mit dem Programmieren anfangen, überlegen wir uns, wie man a und b aus `:UM` und `:C` bestimmen kann. Weil der Umfang $2 \cdot a + 2 \cdot b$ ist, erhalten wir die Gleichung

$$2 \cdot a + 2 \cdot b = \texttt{UM}.$$

Die Beziehung zwischen a und b ist durch die zweite Gleichung

$$\texttt{C} \cdot a = b$$

gegeben. Wenn wir die zweite Gleichung in die erste einsetzen, erhalten wir:

$$\begin{aligned} 2 \cdot a + 2 \cdot \texttt{C} \cdot a &= \texttt{UM} \\ a \cdot (2 + 2 \cdot \texttt{C}) &= \texttt{UM} \\ a &= \frac{\texttt{UM}}{2 + 2 \cdot \texttt{C}}. \end{aligned}$$

Wenn wir die Formel für a in die zweite Gleichung einsetzen, erhalten wir

$$b = \texttt{C} \cdot a = \frac{\texttt{C} \cdot \texttt{UM}}{2 + 2 \cdot \texttt{C}}.$$

Wenn wir jetzt wissen, durch welche Formel a und b aus `:C` und `:UM` berechenbar sind, können wir das entsprechende Programm wie folgt schreiben.

```
to RECHT1 :UM :C
make "a :UM/(2+2*:C)
make "b :C*:a
RECHT :b  :a
end
```

□

Aufgabe 11.12 Betrachten wir das System

$$a \cdot x + b \cdot y = c$$
$$d \cdot x + e \cdot y = f$$

von zwei linearen Gleichungen. Schreibe ein Programm, das für gegebene Werte a, b, c, d, e und f mit $a \cdot e - d \cdot b \neq 0$ die Lösungen für x und y berechnet.

Aufgabe 11.13 Gegeben ist der Umfang `:UM` eines Rechtecks. Entwirf ein Programm, das für die Eingabe `:UM` das Rechteck mit maximaler Fläche bei gegebenem Umfang `:UM` zeichnet.

Beispiel 11.4 Wir wollen ein Programm entwerfen, das für eine gegebene Zahl `:SUM` die kleinste ganze Zahl X findet, so dass die Summe von X und zwei nachfolgenden natürlichen Zahlen größer als `:SUM` ist. Die zwei Nachfolger einer Zahl X sind $X+1$ und $X+2$. Wir suchen die kleinste natürliche Zahl X mit der Eigenschaft

$$X + X + 1 + X + 2 \geq \texttt{SUM}, \quad \text{d. h.}$$
$$3 \cdot X + 3 \geq \texttt{SUM}.$$

Danach wollen wir einen Würfel der Seitenlänge X zeichnen. Um X zu finden, können wir eine while-Schleife geschickt verwenden:

```
to WURFEL :SUM
make "X 1
while [ 3*:X+3 < :SUM ] [make "X :X+1 ]
pr :X
QUADRAT :X
rt 45 fd :X/2 lt 45 QUADRAT :X rt 45 bk :X/2 lt 45
fd :X rt 45 fd :X/2 bk :X/2 rt 45
fd :X lt 45 fd :X/2 bk :X/2 rt 135
fd :X lt 135 fd :X/2 bk :X/2 lt 45
end
```

Bei der Zeichnung des Würfels beachten wir, dass die dritte Dimension von vorne nach hinten immer mit einem Winkel von 45° eingezeichnet wird. Die Länge der Linien dieser Dimension wird visuell halbiert. Probiere den Programmaufruf `WURFEL 500` aus. □

Aufgabe 11.14 Besteht ein Risiko, dass das Programm `WURFEL` unendlich lange läuft? Wenn dieses Risiko besteht, bestimme genau, wann es vorkommen kann.

Aufgabe 11.15 Entwirf ein Programm, das für eine gegebene Zahl M die größte natürliche Zahl A findet, so dass die Summe der Volumen der vier Würfel mit den Seitenlängen A, $A+50$, $A+100$ und $A+150$ noch kleiner als M ist. Dann soll das Programm den Wert von A ausgeben und alle vier Würfel nebeneinander zeichnen. Der erste Würfel soll rot, der zweite gelb, der dritte blau und der vierte grün sein.

Aufgabe 11.16 Entwickle ein Programm, das die folgende Aufgabe löst. Finde für gegebene Zahlen a, b, c und d die kleinste positive Zahl x, sodass das Polynom

$$a \cdot x^3 + b \cdot x^2 + c \cdot x + d = 0$$

zwischen x und $x+1$ eine Lösung hat. Wenn kein solches x existiert, darf das Programm unendlich lange arbeiten.

Aufgabe 11.17 Ändere das Programm, indem es x nur bis zur Zahl 1000 sucht. Wenn x nicht zwischen 0 und 1000 liegt, soll das Programm eine Fehlermeldung ausgeben. Schaffst du es, das Programm so zu modifizieren, dass es statt dem kleinsten x die Zahl y mit minimalem absolutem Wert sucht?

Zusammenfassung

Typische mathematische Methoden haben die Aufgabe, aus bekannten Tatsachen auf neue Tatsachen zu schließen. Zum Beispiel aus gegebenen Eigenschaften eines Objektes weitere Eigenschaften abzuleiten. Häufig ist die bekannte Information durch Zahlen ausgedrückt und die Aufgabe ist, weitere Informationen mittels Berechnungen oder Konstruktionen zu gewinnen. Die Lösungswege für solche Aufgaben können automatisiert werden, indem wir die Methoden implementieren (mittels einer Programmiersprache beschreiben).

Für mehrere Aufgabentypen zur Dreieckskonstruktion kann man LOGO erfolgreich verwenden. Nützlich kann auch der Befehl `home` sein, der die Schildkröte auf dem direkten Weg zum Startpunkt bringt und den Weg einzeichnet. Die Richtung der Schildkröte ändert sich dabei nicht. Mittels des Befehls `while` lassen sich gut die kleinsten oder die größten Zahlen mit bestimmten Eigenschaften suchen.

Kontrollfragen

1. Wie zeichnet man einen Kreis mit gegebenem Radius?

2. Welche Dreieckskonstruktionen können wir in LOGO leicht umsetzen? Findest du eine Konstruktionsaufgabe, die du nicht mit den bisherigen Befehlen realisieren kannst?

3. Welcher Befehl ist bei der Suche nach kleinsten oder größten Zahlen mit gewissen Eigenschaften besonders nützlich? Kannst du die Strategie erklären, wie man allgemein bei solchen Aufgaben vorgeht?

4. Was bedeutet der Befehl `home`? Wann kann er nützlich sein?

5. Was verstehen wir unter Automatisieren von mathematischen Methoden (Vorgehensweisen)?

Kontrollaufgaben

1. Zeichne mit einem Programm die Wellen aus Abb. 11.8, die aus Halbkreisen bestehen. Gegeben ist eine Zahl M. Der Radius der ersten Welle ist die kleinste natürliche gerade Zahl R, so dass die Summe von R und den drei nachfolgenden geraden ganzen Zahlen größer als M ist. Der Radius verkleinert sich um 10 von einem Halbkreis der Welle zum nächsten. Das Zeichnen soll enden, wenn der Radius des letzten Halbkreises kleiner als 10 ist.

Abbildung 11.8

2. Entwickle ein Programm zur Konstruktion der Eckpunkte A, B und C von rechtwinkligen Dreiecken mit gegebenen Seiten b und c, das für die Bestimmung der Punkte die fehlende Seitenlänge a nicht ausrechnet.

3. Entwickle ein Programm zur Konstruktion der Eckpunkte eines Dreiecks mit gegebenen Größen a, b und h_b, wobei h_b die Höhe auf die Seite b ist.

4. Entwickle ein Programm zum Zeichnen von Parallelogrammen mit gegebenen Seitenlängen a und b und dem Winkel $\alpha \leq 90°$.

5. Entwirf ein Programm, das für gegebene Parameterwerte a, b, c und d die Koordinaten der Schnittpunkte der Parabel $f_1(x) = a \cdot x^2 + b$ und der Gerade $f_2(x) = c \cdot x + d$ bestimmt.

6. Betrachten wir die Eingabe a, b, c und d unter dem gleichen Aspekt wie in Kontrollaufgabe 5. Entwickle ein Programm, das die größte natürliche Zahl x findet, so dass $f_1(x) \leq f_2(x)$ gilt.

7. Entwickle ein Programm, das die kleinste natürliche Zahl x findet, so dass

   $$x^4 - x^3 + x^2 - x > \texttt{GR}$$

   gilt, wobei :GR eine frei wählbare Parametergröße sein soll.

8. Gegeben ist eine ganze gerade Zahl :UM, die den Umfang eines rechteckigen Feldes angibt. Entwirf ein Programm, das zwei mögliche ganzzahlige Seitengrößen des Feldes findet, so dass die Fläche des Feldes für alle ganzzahligen Seitengrößen maximal ist.

Lösungen zu ausgesuchten Aufgaben

Aufgabe 11.4
Ein Winkel, der von zwei Seiten mit bekannten Seitenlängen eingeschlossen ist, ist leicht zu zeichnen. Die einzige Herausforderung ist, die dritte Seite des Dreiecks zu zeichnen. Das erreichen wir mit Hilfe des Befehls home. Es gelingt uns, indem wir den Winkel nicht an den Startpunkt legen, sondern zuerst eine Seite zeichnen und dann an deren Ende den Winkel zeichnen (Abb. 11.9).

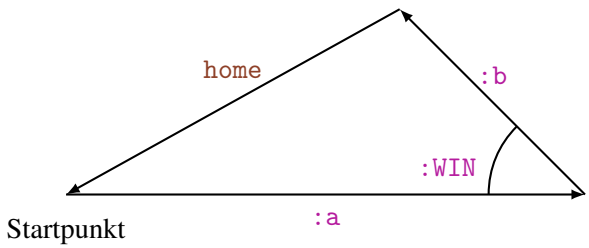


Abbildung 11.9

Das Programm kann wie folgt aussehen:

```
to SWS :a :b :WIN
rt 90 fd :a lt 180-:WIN fd :b home
end
```

Aufgabe 11.6
Wenn man die Größe von zwei Winkeln kennt, kann man den dritten ausrechnen, weil die Summe der Winkel in einem Dreieck immer 180° beträgt. Nachdem man alle Winkel kennt, kann man

das Programm zum Zeichnen nach dem WSW-Satz verwenden. Die genaue Formulierung des Programms überlassen wir dir.

Aufgabe 11.7
In jedem Dreieck muss gelten, dass die Summe von beliebigen zwei Seiten größer als die dritte Seite ist. Wenn diese Bedingung erfüllt ist, kann man erfolgreich DREIECKSSS verwenden.

```
to SSSTEST :a :b :c
if :a+:b<:c [pr [kein Dreieck] stop ]
if :a+:c<:b [pr [kein Dreieck] stop ]
if :b+:c<:a [pr [kein Dreieck] stop ]
DREIECKSSS :a :b :c
end
```

Aufgabe 11.8
Wir zeichnen das Bild aus Abb. 11.5 auf Seite 200, so dass der Punkt in der Mitte der Linie liegt.

```
to PUNKTLIN :LA :DIST
KREISE 10/360
pu fd :DIST pd
rt 90 fd :LA/2 bk :LA fd LA/2
end
```

Aufgabe 11.9
Wir nutzen das Programm PARALLEL :LA :DIST, um unter die Seite $\overline{AB}$ der Länge $c =$ LA eine parallele Linie in der Entfernung :DIST zu zeichnen. C muss auf dieser Linie liegen. Danach zeichnen wir den Thaleskreis aus dem Mittelpunkt der Linie $\overline{AB}$. Weil das Dreieck rechtwinklig ist, sind die Punkte C durch den Schnitt des Kreises und der parallelen Linie gegeben (s. Abb. 11.10 auf der nächsten Seite). Das Programm kann wie folgt aussehen:

```
to THALES :c :DIST
PARALLEL :DIST :c
lt 90 pu bk :DIST pd
KREISMITT :c/2
end
```

Aufgabe 11.16
Wir suchen ein Intervall $[X, X+1]$, in dem das Polynom die x-Achse schneidet. Dabei soll X die kleinste positive ganze Zahl mit dieser Eigenschaft sein. Für $X = 1$ prüfen wir zuerst, ob der Polynomwert $a+b+c+d$ größer oder kleiner als 0 ist. Wenn er größer als 0 ist, suchen wir die kleinste ganze Zahl X, so dass $a \cdot (X+1)^3 + b \cdot (X+1)^2 + c \cdot (X+1) + d \leq 0$ gilt (Abb. 11.11 auf der nächsten Seite). Wenn $a+b+c+d < 0$ ist, suchen wir die kleinste ganze Zahl X, so dass $a \cdot (X+1)^3 + b \cdot (X+1)^2 + c \cdot (X+1) + d \geq 0$ gilt.

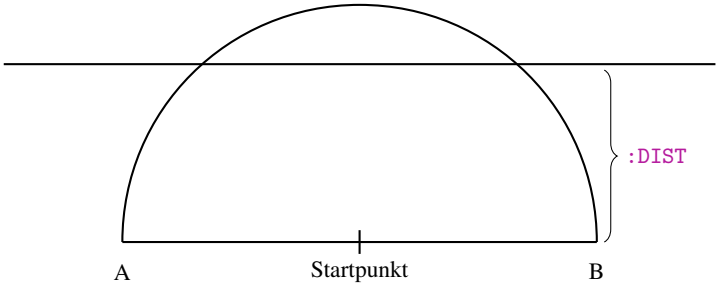


Abbildung 11.10

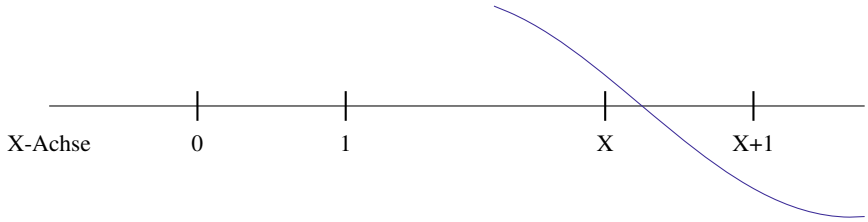


Abbildung 11.11

Damit kann das Programm wie folgt implementiert werden:

```
to NULLSTELLE :a :b :c :d
make "m :a+:b+:c+:d make "X 1 make "Y 2
if :m=0 [ pr :X stop ]
if :m>0 [ while [ :a*:Y*:Y*:Y+:b*:Y*:Y+:c*:y+:d>0]
                [ make "X :X+1 make "Y :Y+1]
if :m<0 [ while [ :a*:Y*:Y*:Y+:b*:Y*:Y+:c*:y+:d<0]
                [ make "X :X+1 make "Y :Y+1]
pr :X
end
```

Lektion 12

Rekursion

Die Rekursion ist ein wichtiges Konzept in der Informatik und in der Mathematik. Beim Programmieren sprechen wir von **Rekursion**, wenn ein Programm in seinem Körper sich selbst als Unterprogramm aufruft. Ein Programm, das sich selbst aufruft, nennen wir **rekursiv**. Dies kann aber ziemlich gefährlich werden. Zum Beispiel wird das rekursive Programm

```
to EWIG
EWIG
fd 100 rt 90
end
```

unendlich lange laufen und nichts zeichnen. Probiere es aus! Das Programm fängt einfach damit an, dass es sich selbst aufruft. Somit ruft es sich ständig auf, ohne je einmal die zweite Zeile zu beachten.

Um zu verstehen, was bei der Ausführung des Programms `EWIG` genau passiert, muss man sich überlegen, was es bedeutet, wenn der Aufruf `EWIG` im Programm `EWIG` durch den Körper des Programms `EWIG` ersetzt wird.

```
to EWIG
   EWIG
   fd  100  rt
   90
fd 100 rt 90
end
```

J. Hromkovič, *Lehrbuch Informatik*, DOI 10.1007/978-3-8348-9692-6_13,

Der Körper des Programms EWIG startet mit dem Aufruf EWIG und deswegen wird dieser Aufruf immer wieder durch den Körper des Programms EWIG ersetzt. Auf diese Weise erhalten wir im zweiten Schritt das folgende Programm:

```
to EWIG
    EWIG
    fd 100 rt
    90
  fd 100 rt 90
fd 100 rt 90
end
```

Wenn wir in den nächsten Schritten den Aufruf EWIG weiterhin durch den Körper des Programms EWIG ersetzen, sehen wir, dass wir das unendlich lange werden tun müssen, da wir nie zur Ausführung der Zeile

```
fd 100 rt 90
```

kommen.

Aufgabe 12.1 Führe die Ersetzung des Aufrufs EWIG durch den Körper des Programms EWIG weitere zweimal aus und zeichne analog zu unserer Darstellung das dadurch entstandene Programm.

Noch besser ist es, die Auswirkung der Rekursion auf folgendes Testprogramm zu beobachten. Der neue Befehl wait verursacht eine kurze Pause in der Ausführung des Programms. Der Parameterwert 1000 beim Befehl wait 1000 in SUPERLOGO hält die Arbeit des Rechners für eine Sekunde an. Bei XLOGO entspricht wait 100 einer Sekunde. Diese Pause ermöglicht uns, die zeitliche Ausführung mit vernünftiger Geschwindigkeit zu beobachten.

```
to EWIG1
fd 100 rt 90 wait 1000
EWIG1
end
```

Hinweis für die Lehrperson Wir verwenden die Parameterwerte des Befehls wait für Programme in SUPERLOGO. Für die Verwendung unserer Programme in XLOGO müssen alle Parameterwerte von wait auf einen Zehntel gesetzt werden.

Wenn wir den Aufruf EWIG1 im Programm EWIG1 durch den Körper von EWIG1 ersetzen, erhalten wir das folgende Programm:

```
to EWIG1
fd 100 rt 90 wait 1000
    fd 100 rt 90 wait
    1000
    EWIG1
end
```

Aufgabe 12.2 Ersetze den Aufruf EWIG1 im Programm oben noch weitere zweimal durch den Körper von EWIG1.

Wir beobachten, dass die Befehle

```
fd 100 rt 90 wait 1000
```

unendlich oft wiederholt werden. Nach jeder Ausführung von fd 100 rt 90 wait 1000 ruft sich das Programm selbst auf und die Ausführung nimmt kein Ende. Somit zeichnet das Programm nach 4 rekursiven Aufrufen ein 100×100 Quadrat und wiederholt diese Zeichnung unendlich oft.

So können wir die Rekursion zum Zeichnen einer unendlichen Spirale verwenden:

```
to SPIRINF :LA
fd :LA rt 90 wait 1000 pr :LA
make "LA :LA+2
SPIRINF :LA
end
```

Das Programm SPIRINF zeigt die Nützlichkeit der rekursiven Aufrufe. Das Programm ruft sich immer mit einem anderen Parameterwert auf. Am Anfang startet man mit dem Wert von :LA und zeichnet eine entsprechend lange Linie. Danach wird :LA um 2 vergrössert und das Programm SPIRINF mit dem um 2 grösseren Wert aufgerufen. Mit jedem Aufruf vergrössert sich der Wert von :LA um 2 und wächst somit unbegrenzt. Teste nun einmal das Programm mit dem Aufruf SPIRINF 20.

Die Wirkung der rekursiven Aufrufe bei der Ausführung des Aufrufs `SPIRINF 20` kann man wie folgt deuten:

```
to SPIRINF :LA
fd 20 rt 90 wait 1000 pr [20]
make "LA 20+2
SPIRINF 22
    fd 22 rt 90 wait 1000 pr [22]
    make "LA 22+2
    SPIRINF 24
        fd 24 rt 90 wait 1000 pr
        [24]
        make "LA 24+2
        SPIRINF 26
                  ⋮
        end
    end
end
```

Aufgabe 12.3 Ersetze in der Darstellung des Ausführung des Aufrufs `SPIRINF 20` den Aufruf `SPIRINF 26` durch die entsprechenden aktuellen Befehle des Programmkörpers von `SPIRINF`. Dabei entsteht der Aufruf `SPIRINF 28`. Ersetze auch diesen Aufruf in der gleichen Weise.

Wir beobachten, dass keiner der Befehle `end` je erreicht wird. Statt dessen werden unendliche viele Aufrufe von `SPIRINF` folgen und bei jedem Aufruf wird eine Linie mittels

```
fd :LA rt 90
```

gezeichnet. Weil der Wert von `:LA` bei jedem neuen Aufruf um 2 wächst, erhalten wir eine unendlich lange, rechteckige Spirale.

Aufgabe 12.4 Ersetze die zwei Zeilen

```
make "LA :LA+2
SPIRINF :LA
```

im Programm SPIRINF durch eine Zeile

```
SPIRINF :LA+2
```

Wird diese Änderung das Verhalten des Programms verändern? Teste das Programm und versuche zu erklären, warum es so läuft, wie es läuft.

Aufgabe 12.5 Würde sich etwas am Verhalten des Programms SPIRINF ändern, wenn wir die Reihenfolge der letzten beiden Programmzeilen vertauschen?

Aufgabe 12.6 Entwickle ein Programm, das mittels Rekursion eine unendliche sechseckige Spirale zeichnet.

Aufgabe 12.7 Erweitere das Programm SPIRINF um einen Parameter :ECK, so dass man mit ihm unendliche Spiralen mit einer beliebigen Anzahl von Ecken zeichnen kann.

Üblicherweise entwerfen wir keine Programme, die unendlich lange und unendlich grosse Bilder zeichnen. Wir können aber rekursive Aufrufe unter Verwendung des if-Befehls und des Befehls stop so kombinieren, dass das rekursive Programm beim Erreichen einer gewissen Bildgrösse die Arbeit beendet. Das folgende rekursive Programm zeichnet eine :ECK-eckige Spirale so lange wie die letzte Seite nicht grösser als :MAX ist.

```
to SPIRREC :LA :ECK :MAX
fd :LA rt 360/:ECK
wait 1000 pr :LA
make "LA :LA+2
if :LA<:MAX [ SPIRREC :LA :ECK :MAX ] [ stop ]
end
```

Bei jedem rekursiven Aufruf von SPIRREC ist :LA um 2 grösser und der rekursive Aufruf von SPIRREC kommt nur dann zustande, wenn :LA kleiner als :MAX ist. Wenn der ständig wachsende Wert der Variablen :LA den Wert des Parameters :MAX überschreitet, hört die Ausführung des letzten rekursiven Aufrufs durch den Befehl stop auf. Danach wird die Ausführung des Programms an den vorletzten Aufruf zurückgegeben. Dort steht aber nur noch der Befehl end und damit wird dieser Aufruf sofort abgeschlossen. Auf diese Weise werden alle Aufrufe, einer nach dem anderen, durch den end-Befehl abgeschlossen, bis das letztendlich ganze Programm beendet ist.

Aufgabe 12.8 Wird sich etwas an der Tätigkeit des rekursiven Programms SPIRREC ändern, wenn eine der folgenden drei Änderungen erfolgt ist?

a) Der Text [stop] wird gelöscht.

b) Die letzte Zeile des Programmkörpers wird durch die folgenden Zeilen ersetzt:

```
if :MAX<:LA [ stop ]
SPIRREC :LA :ECK :MAX
```

c) Die letzten zwei Zeilen werden durch die folgende Zeile ersetzt:

```
if :LA<:MAX [ SPIRREC :LA+2 :ECK :MAX ]
```

Überlege dir zuerst die Antwort und überprüfe sie dann durch Testläufe.

Aufgabe 12.9 Überlege dir, was folgendes Programm macht, und überprüfe deine Idee durch das Aufrufen von SPIRRECHTREC 5 100 1 2 400 und SPIRRECHTREC 20 70 2 4 300.

```
to SPIRRECHTREC :MIN1 :MIN2 :ADD1 :ADD2 :MAX
if :MIN1>:MAX [ stop ]
if :MIN2>:MAX [ stop ]
fd :MIN1 rt 90 fd :MIN2 rt 90
make "MIN1 :MIN1+:ADD1 make "MIN2 :MIN2+:ADD2
SPIRRECHTREC :MIN1 :MIN2 :ADD1 :ADD2 :MAX
end
```

Schreibe das Programm so um, dass man statt der Struktur if Bedingung [...] des if-Befehls nur die Struktur if Bedingung [...] [...] verwendet.

Aufgabe 12.10 Entwickle ein rekursives Programm zum Zeichnen einer kreisförmigen Spirale.

Beispiel 12.1 Für viele Aufgaben, die wir mittels repeat- und while-Schleifen gelöst haben, kann man auch ein rekursives Programm schreiben. Nennen wir als Beispiel das Zeichnen eines $1 \times N$-Feldes von Quadraten der Seitengrösse :GR. Man kann mit einem rekursiven Aufruf genau ein Quadrat zeichnen und dann die neue Startposition zum Zeichnen des nächsten Quadrates einnehmen. Die Variable :N verwendet man also als **Kontrollvariable**. Bei jedem Aufruf wird sie um 1 verkleinert und wenn sie 0 ist, wird die Ausführung des Programms anhalten.

```
to FELDREC :GR :N
if :N=0 [ stop ]
make "N :N-1
repeat 7 [ fd :GR rt 90 ] rt 90 wait 1000
FELDREC :GR :N
end
```

Wir sehen, dass wir auf diese Weise eine Schleife durch einen rekursiven Aufruf ersetzen können. □

Aufgabe 12.11 Schreibe das Programm `FELDREC` so um, dass du den rekursiven Aufruf `FELDREC :GR :N` durch eine Schleife ersetzt. Mach es zuerst mit einer `while`-Schleife und danach mit einer `repeat`-Schleife.

Aufgabe 12.12 Entwickle ein rekursives Programm zum Zeichnen des Bildes aus Abb. 12.1. Die Anzahl der Stufen sowie die Größe der kleinen Quadrate soll frei wählbar sein.

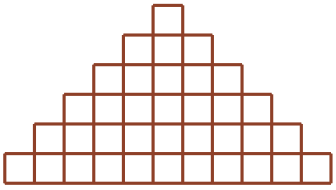


Abbildung 12.1

Aufgabe 12.13 Im Beispiel 8.2 haben wir das Programm `AUGE :AN :UM :NACH` zum Zeichnen der Bilder aus Abb. 8.6 auf Seite 150 entwickelt. Das Programm verwendet eine `repeat`-Schleife. Schreibe zuerst das Programm so um, dass man statt der `repeat`-Schleife eine `while`-Schleife verwendet. Dann ändere das neue Programm in ein rekursives Programm um.

Aufgabe 12.14 Entwickle ein rekursives Programm für die Kontrollaufgabe 1 in Lektion 8 (Abb. 8.7 auf Seite 152).

Aufgabe 12.15 Entwickle ein Programm zum Zeichnen einer Folge von gleichschenkligen rechtwinkligen Dreiecken wie in Abb. 12.2 dargestellt. Die Länge `:LA` der Schenkel des kleinsten Dreiecks soll frei wählbar sein. Die Länge der Hypotenuse des kleinsten Dreiecks bestimmt die Länge der Schenkel des nachfolgenden Dreiecks, usw. Das Zeichnen soll aufhören, wenn die Hypotenusenlänge des letzten Dreiecks mehr als 400 Schritte erreicht hat.

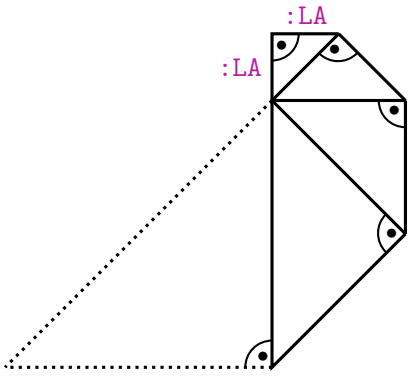


Abbildung 12.2

Bei rekursiven Programmen betrachten wir eine wichtige Charakteristik, die wir Tiefe nennen. Wir sagen, dass **ein Aufruf *A* in einem Aufruf *B* verschachtelt ist**, wenn man während der Bearbeitung des Aufrufs *B* den Aufruf *A* durchführen muss und die Ausführung des Aufrufs *B* erst nach Beendigung des Aufrufs von *A* abgeschlossen werden kann. Zum Beispiel ist `SPIRINF 22` in `SPRINF 20` verschachtelt. Der Aufruf von `SPRINF 20` kann nicht beendet werden, bevor die Ausführung von `SPIRINF 22` abgeschlossen ist. Analog dazu ist `SPIRREC 52 4 100` in `SPIRREC 50 4 100` verschachtelt. Erst wenn die Ausführung von `SPIRREC 50 4 100` beendet ist, kann die Ausführung von `SPIRREC 50 4 100` zu Ende geführt werden.

Die **Tiefe eines Programmaufrufs** ist die maximale Anzahl der in sich verschachtelten Aufrufe, die während der Ausführung des Programms vorkommt. Bei unendlich lange laufenden Programmen wie `EWIG` ist die Tiefe unendlich. Wenn ein rekursives Programm endlich lange läuft, ist die Tiefe des Programmaufrufs immer eine nicht negative ganze Zahl. Diese Zahl kann von den Eingabewerten des Programms abhängen. Zum Beispiel haben wir das Programm `SPIRREC` so lange rekursiv aufgerufen, wie `:LA` kleiner als `:MAX` war. Weil `:LA` immer um 2 gewachsen ist, ist die Tiefe der Rekursion $(\texttt{MAX} - \texttt{LA})/2$. Bei rekursiven Programmen, die genau einen rekursiven Aufruf in ihrem Körper beinhalten, ist die Tiefe des Rekursion gleich der Gesamtanzahl der rekursiven Aufrufe. Somit ist auch die Tiefe des Aufrufs `FELDREC :GR :N` genau `:N`. Wenn ein Programm sich selbst mehrfach in seinem Körper aufruft, entspricht die Gesamtzahl der Aufrufe nicht mehr der Tiefe.

Aufgabe 12.16 Bestimme die Tiefe des rekursiven Aufrufs `SPIRRECHTREC 10 100 1 2 250`. Schaffst du es auch, eine allgemeine Formel für die Berechnung der Tiefe des Aufrufs

```
SPIRRECHTREC :MIN1 :MIN2 :ADD1 :ADD2 :MAX
```

als eine Funktion der fünf Argumente `:MIN1`, `:MIN2`, `:ADD1`, `:ADD2` und `:MAX` abzuleiten?

Beim Aufruf `FELDREC 30 4` wird bei der Ausführung zuerst rekursiv `FELDREC 30 3` aufgerufen. Aber der Aufruf von `FELDREC 30 3` kann nicht beendet werden, bevor der in ihm verschachtelte Aufruf `FELDREC 30 2` nicht ausgeführt ist. In `FELDREC 30 2` wird aber noch `FELDREC 30 1` aufgerufen. In `FELDREC 30 1` wird als letzter rekursiver Aufruf `FELDREC 30 0` vorkommen. Durch den Befehl

```
if :N=0 [ stop ]
```

wird dieser Aufruf sofort beendet. Dieses `stop` bedeutet aber nicht das Ende der Ausführung des Aufrufs `FELDREC 30 4`, sondern nur das Ende der Arbeit am Aufruf `FELDREC 30 0`. Wenn dieser Aufruf fertig ist, kehrt das Programm zu `FELDREC 30 1` zurück und schließt es mit `end` ab. Danach geht die Ausführung mit `FELDREC 30 2` weiter, usw. Der ganze Ablauf ist in Abb. 12.3 anschaulich dargestellt. Hier sehen wir anschaulich, dass

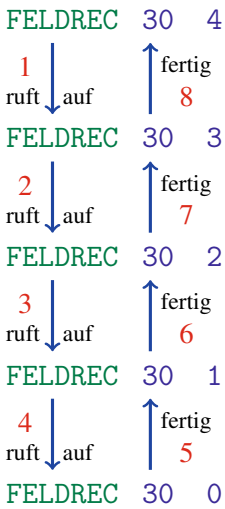


Abbildung 12.3

die Rekursionstiefe dieses Aufrufs 4 ist. Als Wichtigstes bleibt aber zu beachten, dass

> *die Ausführung eines rekursiven Aufrufs nicht beendet werden kann, bevor die Ausführung des in ihm verschachtelten rekursiven Aufrufs nicht abgeschlossen ist.*

Die roten Nummern neben den Pfeilen in Abb. 12.3 zeigen den zeitlichen Ablauf der Aufrufe und deren Ende während der Ausführung von `FELDREC 30 4`.

Rekursive Programme sind viel schwieriger zu verstehen und zu entwickeln als Programme ohne Rekursion. Als wir uns bemüht haben, das Konzept der Variablen zu verstehen, sind wir auf die Ebene der Register gegangen, um anzuschauen, wie der Rechner unsere Befehle genau ausführt. Um die Rekursion wirklich zu verstehen, müssen wir dasselbe tun. Im Speicher spielt sich bei der Ausführung eines rekursiven Programms etwas ab, was wir bisher nicht gesehen haben. Bei jedem Aufruf eines rekursiven Programms werden neue Register für alle Variablen des rekursiven Programms angelegt und mit aktuellen Werten des Aufrufs belegt. Damit verwendet man für jede Variable eines rekursiven Programms während der Ausführung eines Aufrufs genau so viele Register, wie die Tiefe des Aufrufs beträgt. Wenn ein Aufruf abgeschlossen ist, werden die reservierten Register dieses Aufrufs freigegeben.

Für unser Verständnis ist dies am besten durch die Darstellung der Entwicklung der Speicherinhalte und Reservierungen von Registern zu veranschaulichen. Betrachten wir den Aufruf

```
FELDREC 30 4,
```

dessen zeitlicher Ablauf in Abb. 12.3 dargestellt ist. Der Inhalt des Speichers in den acht Zeitstufen (rote Zahlen in Abb. 12.3) ist in der Tabelle 12.1 dargestellt.

In der nullten Spalte ist die Situation nach dem Aufruf `FELDREC 30 4` dargestellt. Die Variable `:GR` erhält den Wert `30` und die Variable `:N` den Wert `4`. Zu diesem Zeitpunkt gibt es keine anderen reservierten Register für `GR` und `N`, was in der Tabelle 12.1 durch Striche angedeutet ist. Die Spalte 1 zeigt den Stand des Speichers nach dem rekursiven Aufruf `FELDREC 30 3`. Der Rechner nimmt zwei neue Register für `GR` und `N`, legt die Zahlen `30` und `3` hinein und verwendet für die Arbeit mit den Variablen `:GR` und `:N` ausschließlich die beiden Register `GR(FELDREC 30 3)` und `N(FELDREC 30 3)`, bis entweder ein neuer rekursiver Aufruf von `FELDREC` vorkommt oder die Ausführung dieses Aufrufs beendet

	0	1	2	3	4	5	6	7	8
GR(FELDREC)	30	30	30	30	30	30	30	30	30
N(FELDREC)	4	3	3	3	3	3	3	3	3
GR(FELDREC 30 3)	–	30	30	30	30	30	30	30	–
N(FELDREC 30 3)	–	3	2	2	2	2	2	2	–
GR(FELDREC 30 2)	–	–	30	30	30	30	30	–	–
N(FELDREC 30 2)	–	–	2	1	1	1	1	–	–
GR(FELDREC 30 1)	–	–	–	30	30	30	–	–	–
N(FELDREC 30 1)	–	–	–	1	0	0	–	–	–
GR(FELDREC 30 0)	–	–	–	–	30	–	–	–	–
N(FELDREC 30 0)	–	–	–	–	0	–	–	–	–

Tabelle 12.1

wird. Weiter beobachten wir, dass der Wert 4 in der ersten Zeile für N(FELDREC) in 3 geändert wurde. Dies passierte noch vor dem Aufruf FELDREC :GR :N durch die Ausführung des Befehls

```
make "N :N-1.
```

Beim Aufruf FELDREC 30 2 werden wieder neue Register für GR und N angelegt und verwendet. Bei der Bearbeitung eines Aufrufs werden nur die Inhalte der dafür angelegten Register geändert. Damit haben wir in Spalte 4 nach dem Aufruf FELDREC 30 0 fünf unterschiedliche Register, jeweils für N und GR. Der Wert für :N ist aber in den fünf verschiedenen Rekursionsstufen unterschiedlich.

In Spalte 5 ist die Situation nach dem Abschluss von FELDREC 30 0 dargestellt. Die Reservierung für

```
GR(FELDREC 30 0) und N(FELDREC 30 0)
```

wird aufgehoben und die Inhalte dieser Register gelöscht. Die Ausführung des Programms kehrt zurück zu

```
FELDREC 30 1
```

und zur Verwendung stehen das entsprechenden Register GR(FELDREC 30 1) und das Register N(FELDREC 30 1) bereit. Weil end direkt nach dem rekursiven Aufruf steht, wird praktisch der Aufruf von FELDREC 30 1 sofort beendet und die Ausführung an die zweite Rekursionsstufe (Spalte 6 in Tab. 12.1) abgegeben. Damit ist jetzt der aktuelle Wert für :N die Zahl 1. Nach dem Abschluss von FELDREC 30 2 wird die Ausführung an FELDREC 30 3 übergeben. Die dort gespeicherten Werte 30 und 2 für :GR und :N können dann verwendet werden. Damit machen wir die folgende wichtige Beobachtung:

> *Bei rekursiven Aufrufen übergibt das laufende Programm an den in ihm verschachtelten Aufruf die Werte für die Variablen. Wenn ein Aufruf beendet wird, werden die Werte seiner Variablen gelöscht und es kommt zu keiner Übertragung der Variablenwerte nach oben. Als aktuelle Variablenwerte werden die vorher gespeicherten Werte des Aufrufs verwendet, der gerade ausgeführt wird, in dem der gerade abgeschlossene Aufruf verschachtelt war.*

Aufgabe 12.17 Füge den Befehl pr :N als letzte Zeile vor end in das Programm FELDREC ein und überprüfe damit unsere Behauptung und den Inhalt der Spalten 5, 6, 7 und 8 für die Variable :N in Tab. 12.1.

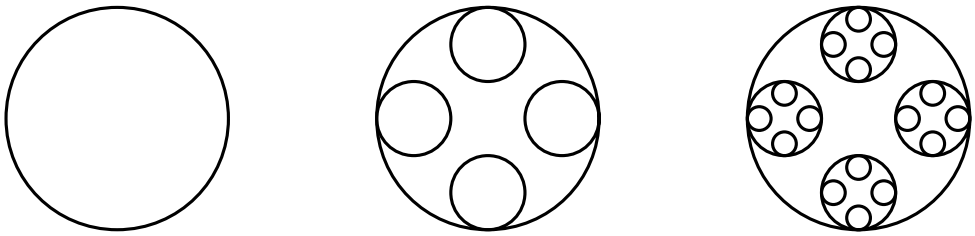
Aufgabe 12.18 Wir ändern das Programm FELDREC, indem wir den Befehl make "N :N−1 entfernen und den Aufruf FELDREC :GR :N durch den Aufruf FELDREC :GR :N−1 ersetzen. Diese Änderungen haben keinen Einfluss auf die gezeichneten Bilder. Aber die Tabelle 12.1 ändert sich beim Aufruf FELDREC 30 4. Weißt du wie?
Ändert sich die Tabelle 12.1, wenn du die folgende Zeile als letzte Zeile einfügst:

```
make "GR 2*:GR make "N :N+7?
```

Die Tabelle 12.1 zeigt immer nur den Stand des Speichers nach dem rekursiven Aufruf oder nach dem Abschluss eines rekursiven Aufrufs. Erweitere die Tabelle um Spalten für Speicherinhalte, die im letzten Augenblick vor dem Abschluss der rekursiven Aufrufe hinzukommen. Wie sieht eine solche Tabelle aus, wenn man die oben erwähnte, neue Zeile vor end einfügt?

Aufgabe 12.19 Zeichne für den Aufruf SPIRRECHTREC 100 10 10 5 120 des Programms aus der Aufgabe 12.9 eine Tabelle wie in Tab. 12.1, welche die Entwicklung der Speicherinhalte während der Ausführung des Aufrufs dokumentiert.

In den vorhergehenden Aufgaben haben wir oft rekursive Programme entwickelt, obwohl wir die Aufgabenstellung leicht mit einer while-Schleife hätten lösen können. Gibt es



(a) (b) (c)

Abbildung 12.4

Aufgaben, für die uns die Rekursion elegante Lösungsmöglichkeiten in dem Sinne bietet, dass man sich ohne Rekursion sehr schwer tun würde, ein Programm zum Zeichnen entsprechender Bilder zu entwickeln? Die Antwort auf diese Frage ist „Ja“. Wir zeigen eine nützliche, auf die Rekursion ausgerichtete Aufgabe im folgenden Beispiel. Vorher bemerken wir noch, dass alle bisherigen rekursiven Programme genau einen rekursiven Aufruf von sich selbst enthielten. Das muss nun nicht mehr so sein.

Beispiel 12.2 Unsere Aufgabe ist es, ein rekursives Muster zu zeichnen, wobei die Tiefe der Rekursion eine frei wählbare Größe sein soll. Das Muster ist in Abb. 12.4 dargestellt. In Abb. 12.4(a) ist ein Kreis abgebildet. Dieses Bild entspricht der Rekursion der Tiefe 1. Der Umfang `:UM` des Kreises soll frei wählbar sein. Die Abbildung 12.4(b) zeigt das Prinzip der Rekursion. Das gleiche Bild, nämlich ein Kreis, soll noch viermal in kleinerer Ausführung, mit einem Drittel des ursprünglichen Umfangs gezeichnet werden. Die kleinen Kreise sollen im Abstand von einem Viertelkreis gezeichnet werden. Wenn wir diese Anforderung rekursiv wiederholen, werden in den kleineren Kreisen jeweils vier noch kleinere Kreise gezeichnet. Das Resultat sehen wir dann in Abb. 12.4(c).

Wir sehen, dass es gar nicht einfach wäre, für eine gegebene Tiefe `:TIEF` das entsprechende Bild ohne Rekursion zu zeichnen. Im Gegensatz dazu ist die rekursive Strategie sehr einfach:

> Wiederhole viermal:
> [Zeichne einen kleineren Kreis
> (das Muster in kleinerer Ausführung)
> und dann einen Viertelkreis des großen Kreises.]

Die Umsetzung der Strategie sieht wie folgt aus:

```
to KREISREC :UM :TIEF
if :TIEF=0 [ stop ]
repeat 4 [ KREISREC :UM/3 :TIEF-1
          repeat 90 [ fd :UM/360 rt 1 ] wait 200 ]
end
```

Beim Aufruf `KREISREC 800 3` wird die Abbildung 12.4(c) gezeichnet. Es ist interessant und wichtig zu beobachten, in welcher Reihenfolge die Kreise gezeichnet werden. Zuerst wird der kleinste Kreis ganz links gezeichnet. Das kommt daher, weil die Ausführung des Aufrufs `KREISREC 800 3` mit der Ausführung des Aufrufs `KREISREC 800/3 2` beginnt. Aber `KREISREC 800/3 2` fängt mit `KREISREC 800/9 1` an, der wiederum mit `KREISREC 800/27 0` beginnt. Weil `:TIEF=0` ist, wird der letzte Aufruf sofort abgeschlossen und das Programm führt nun den Befehl

```
repeat 90 [ fd (800/9)/360 rt 1 ]
```

aus, der den ersten Viertelkreis des kleinsten Kreises zeichnet. Dadurch wird wieder `KREISREC 800/27 0` aufgerufen und das Zeichnen beendet. Nun wird der zweite Viertelkreis des kleinsten Kreises gezeichnet. Wenn die Ausführung von `KREISREC 800/9 1` abgeschlossen ist, wird der erste Viertelkreis des größeren Kreises in `KREISREC 800/3 2` gezeichnet. Anschließend wird der zweite der kleinsten Kreise in diesem Kreis durch `KREISREC 800/9 1` gezeichnet usw.

Für den Aufruf `KREISREC 999 1` kann man die Verschachtelung der rekursiven Aufrufe in einem Ablaufdiagramm, wie in Abb. 12.5 eingezeichnet, darstellen.

Die Pfeile nach unten entsprechen den rekursiven Aufrufen. Die Pfeile nach oben ent-

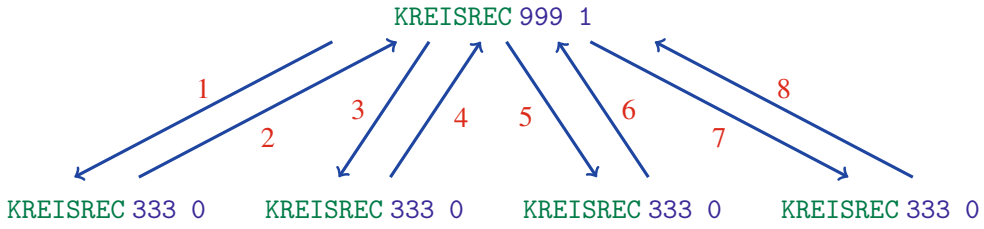


Abbildung 12.5

sprechen der Rückkehr zum Hauptprogrammaufruf nach der Ausführung des Aufrufs `KREISREC 333 0`, um die Arbeit an `KREISREC 999 1` fortzusetzen. Wir sehen, dass die Tiefe des Aufrufs `KREISREC 999 1` genau 1 ist, obwohl die Ausführung vier rekursive Aufrufe beinhaltet. Die entsprechende Entwicklung der Speicherinhalte zu diesen acht Zeitpunkten ist in Tab. 12.2 dargestellt.

	0	1	2	3	4	5	6	7	8
`UM`	999	999	999	999	999	999	999	999	999
`TIEF`	1	1	1	1	1	1	1	1	1
`UM`(`KREISREC 333 0`)	–	333	–	333	–	333	–	333	–
`TIEF`(`KREISREC 333 0`)	–	0	–	0	–	0	–	0	–

Tabelle 12.2

□

Aufgabe 12.20 Zeichne die Baumstruktur der rekursiven Aufrufe analog zu Abb. 12.5 auf der vorherigen Seite für den Aufruf `KREISREC 999 2` und erstelle eine entsprechende Tabelle, die die Entwicklung der Variablenwerte von `:UM` und `:TIEF` dokumentiert.

Aufgabe 12.21 Zeichne das Bild für die Rekursionsstufe 4 (die nächste nach Abb. 12.4(c)), ohne das rekursive Programm `KREISREC` zeichnen zu lassen. Nummeriere die gezeichneten Kreise aller Größen nach der Reihenfolge, in der sie gezeichnet werden. Überprüfe deine Nummerierung mit dem Aufruf `KREISREC 900 4`. Der Befehl `wait` hilft dir, das Vorgehen der Schildkröte langsam zu beobachten.

Eine sehr prägnante Darstellung der Folge rekursiver Aufrufe und deren jeweiligen Abschlüssen kann man mit Hilfe von Klammerfolgen erzielen. Dabei entspricht die linke Klammer dem Beginn eines rekursiven Aufrufs und die rechte Klammer dem Abschluss eben dieses Aufrufs. Somit entspricht die Klammerfolge

(((( ))))

dem Aufruf `FELDREC 30 4`, dessen Verlauf in Abb. 12.3 auf Seite 217 dargestellt ist. Wenn wir die roten Zahlen aus Abb. 12.3 unter die Klammern wie folgt schreiben,

(	(	(	(	)	)	)	)
1	2	3	4	5	6	7	8

sehen wir, dass die Reihenfolge der Klammern genau der Folge von Aufrufen und Abschlüssen der Ausführung entsprechen. So entspricht

```
(   )
4   5
```

der Ausführung des Befehls `FELDREC 30 0`. Die ganz rechte und die ganz linke Klammer

```
(                   )
1                   8
```

entsprechen dem Beginn und dem Ende des Aufrufs `FELDREC 30 3`. Genau wie bei arithmetischen Ausdrücken gibt es zu jeder öffnenden Klammer eine schließende Klammer. Ein Klammerpaar stellt den Anfang und das Ende eines Aufrufs dar. Die Ausführung des entsprechenden Aufrufs in einem rekurisven Programm entspricht in der Arithmetik der Auswertung des Ausdrucks im jeweiligen Klammerpaar.

Für den Aufruf `KREISREC 999 1` (Abb. 12.5 auf Seite 222) sieht die Klammerdarstellung der rekursiven Aufrufe wie folgt aus:

$$R_1 = (\ )\ (\ )\ (\ )\ (\ )\ .$$

Der Ablauf der Ausführung des Aufrufs `KREISREC 999 2` kann durch folgende Klammerfolge

$$R_2 = (\ (\ )\ (\ )\ (\ )\ (\ )\ )\quad (\ (\ )\ (\ )\ (\ )\ (\ )\ )\quad (\ (\ )\ (\ )\ (\ )\ (\ )\ )\quad (\ (\ )\ (\ )\ (\ )\ (\ )\ )$$

dargestellt werden. Um die rekursive Struktur transparent darzustellen, beobachten wir, dass

$$R_2 = (\ R_1\ )\ \ (\ R_1\ )\ \ (\ R_1\ )\ \ (\ R_1\ ).$$

Aufgabe 12.22 Welche Klammerfolge entspricht dem Aufruf `FELDREC 20 8`?

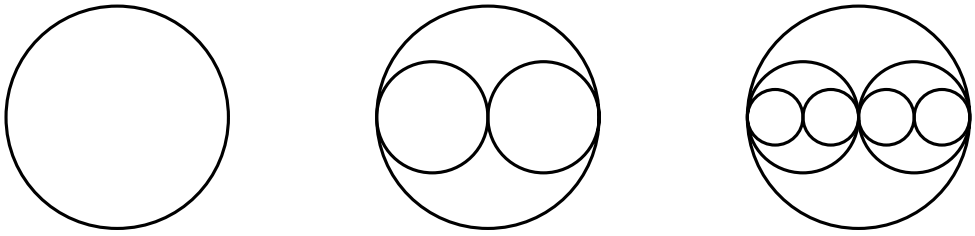
Aufgabe 12.23 Wie geht man vor, um die Klammerfolge für den Aufruf `KREISREC 1000 3` aufzuschreiben? Verwende die Notation R_1 und R_2, um dein Vorgehen zu veranschaulichen. Kannst du auf diese Weise den Aufbau der Klammerfolge für den Aufruf `KREISREC 1000 4` erläutern?

Aufgabe 12.24 Die Anzahl der linken Klammern einer Klammerfolge entspricht der Gesamtzahl der rekursiven Aufrufe. Kannst du die Anzahl der Aufrufe bei der Ausführung von `KREISREC :UM :TIEF` als eine Funktion in Abhängigkeit der Variable `:TIEF` ausdrücken?

Aufgabe 12.25 Aus einer Klammerfolge eines Aufrufs kann man die Tiefe folgendermaßen bestimmen. Man berechnet das Maximum der Differenzen zwischen der Anzahl der linken, öffnenden und der rechten, schließenden Klammern über alle Präfixe der Klammerfolge. Die Tiefe ist also die maximale Zahl geöffneter Klammern in der Klammerung. Kannst du erklären warum das so ist?

Aufgabe 12.26 Der Aufruf `KREISREC :UM :TIEF` hat die Tiefe `:TIEF`. Wie viele rekursive Aufrufe werden während der Ausführung von `KREISREC :UM :TIEF` demnach realisiert?

Aufgabe 12.27 Schreibe ein rekursives Programm, welches das rekursive Muster aus der Abbildung 12.6 für beliebige Rekursionstiefen zeichnen kann.



Tiefe 1 Tiefe 2 Tiefe 3

Abbildung 12.6

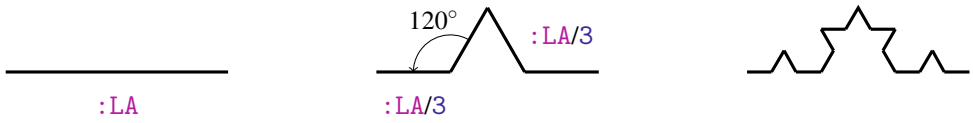
Aufgabe 12.28 Betrachte das folgende Programm:

```
to FELDABREC :A :B
if :B=0 [ stop ]
repeat 4 [ fd :A/2 rt 90 ] FELDABREC :A/2 :B-1
fd :A/2
repeat 4 [ fd :A/2 rt 90 ] FELDABREC :A/2 :B-1
rt 90 fd :A/2 lt 90
repeat 4 [ fd :A/2 rt 90 ] FELDABREC :A/2 :B-1
bk :A/2
repeat 4 [ fd :A/2 rt 90 ] FELDABREC :A/2 :B-1
lt 90 fd :A/2 rt 90
end
```

Überlege dir, was das Programm zeichnet, indem du die Bilder für die ersten drei Rekursionsstufen aufzeichnest. Überprüfe deine Überlegungen mittels Testläufen.

Bestimme aus der Programmbeschreibung die Zeichenreihenfolge der kleinsten Quadrate für die Aufrufe `FELDABREC 200 2` und `FELDABREC 200 3`.

Beispiel 12.3 Die Aufgabe besteht darin, das rekursive Muster aus Abb. 12.7 zu zeichnen.



(a) (b) (c)

Abbildung 12.7

In der ersten Rekursionsstufe zeichnet man nur eine Linie der Länge `:LA` (Abb. 12.7(a)). In der nächsten Rekursionsstufe ersetzt man die Linie durch den gebrochenen Weg, der zwei Punkte der gleichen Entfernung `:LA` verbindet. Wenn man jede der vier Linien aus Abb. 12.7(b) durch eine entsprechend lange, gebrochene Linie ersetzt, erhält man Abb. 12.7(c). Deswegen zeichnet das folgende Programm auf der tiefsten Rekursionsebene eine Linie und „ersetzt“ rekursiv jeden graden Weg durch einen geknickten Weg.

```
to STARREC :LA :TIEF
if :TIEF=0 [ fd :LA stop ]
STARREC :LA/3 :TIEF-1
lt 60
STARREC :LA/3 :TIEF-1
rt 120
STARREC :LA/3 :TIEF-1
lt 60
STARREC :LA/3 :TIEF-1
end
```

Das durch `STARREC` gezeichnete Muster sieht schön aus, insbesondere für höhere Rekursionstiefen. Das Muster hat aber noch einen Schönheitsfehler. Es entwickelt sich nur in eine Richtung, weil es nur durch die Veränderung einer geraden Linie entstanden ist. Wenn wir ein solches Muster in einer abgeschlossenen Figur haben wollen, können wir das mit folgendem Programm erreichen:

```
to STAR2 :TIEF :LA
repeat 12 [ STARREC :LA :TIEF rt 30 ]
end
```

Wir sehen jetzt, dass STAR2 die zwölf Seiten eines regelmäßigen 12-Ecks durch die „Musterlinien“ von STARREC ersetzt hat. Teste den Aufruf STAR2 5 100. Wir lernen dadurch auch, dass man rekursive Programme genau wie gewöhnliche Programme als Unterprogramme verwenden kann. □

Aufgabe 12.29 Entwirf ein rekursives Programm, welches für die ersten drei Rekursionsstufen Bilder wie in Abb. 12.8 zeichnet.

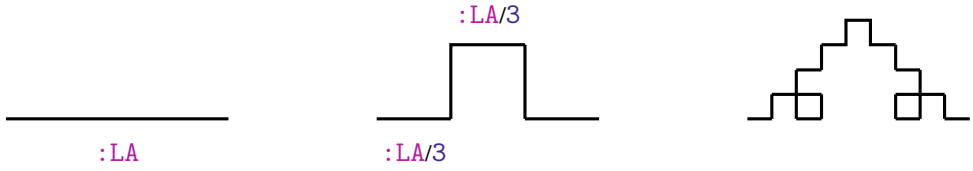


Abbildung 12.8

Aufgabe 12.30 Entwickle ein Programm, welches für die ersten drei Rekursionsstufen Bilder wie in Abb. 12.9 zeichnet.

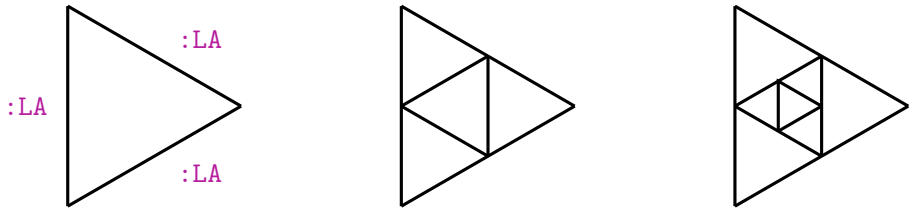


Abbildung 12.9

Beispiel 12.4 Eine andere Methode, rekursive Bilder zu zeichnen, besteht nicht in der Wiederholung von kleineren Bildern innerhalb des ursprünglichen Bildes oder in der „Ersetzung“ hypothetischer Verbindungen durch Muster, sondern in einer Erweiterung des Bildes an gewissen Stellen. Auf diese Weise kann man Bäume wie in Abb. 12.10 zeichnen.

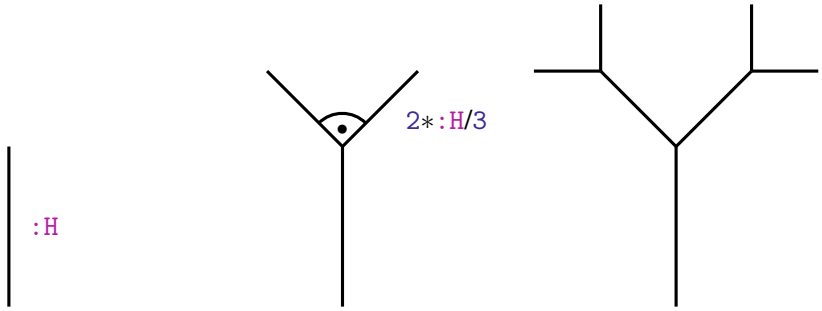


Abbildung 12.10

Wir zeichnen mit dem folgenden Programm den Baum so, dass wir nach dem Zeichnen der linken sowie der rechten Äste an die Startposition zurückkehren.

```
to BAUM :H :TIEF
if :TIEF=0 [ stop ]
fd :H lt 45 wait 500
BAUM (2∗:H)/3 :TIEF−1
rt 45 bk :H wait 500
fd :H rt 45
BAUM (2∗:H)/3 :TIEF−1
lt 45 bk :H
end
```

Wir beobachten hier wieder die Reihenfolge der rekursiven Aufrufe. Zuerst wird die komplette linke Seite der Baumkrone gezeichnet und dann erst die rechte. Das bedeutet, dass zuerst der erste rekursive Aufruf

```
BAUM (2∗:H)/3 :TIEF−1
```

vollständig ausgeführt wird. Erst dann kommt der zweite an die Reihe. In Abb. 12.11 auf der nächsten Seite sehen wir in einem Baumdiagramm die Darstellung der rekursiven Aufrufe beim Aufruf `BAUM 100 3`. Die Pfeile nach unten zeigen die rekursiven Aufrufe. Die Pfeile nach oben stehen für die Rückkehr nach der Ausführung des entsprechenden Aufrufs. Die Zahlen neben den Pfeilen entsprechen der zeitlichen Reihenfolge der Aufrufe und deren Ausführung. Die Tiefe des rekursiven Aufrufs `BAUM 100 3` ist 3. Wir sehen in Abb. 12.11 auf der nächsten Seite, dass die Tiefe der Länge des längsten Weges vom Start über die Pfeile nach unten zum letzten Aufruf entspricht.

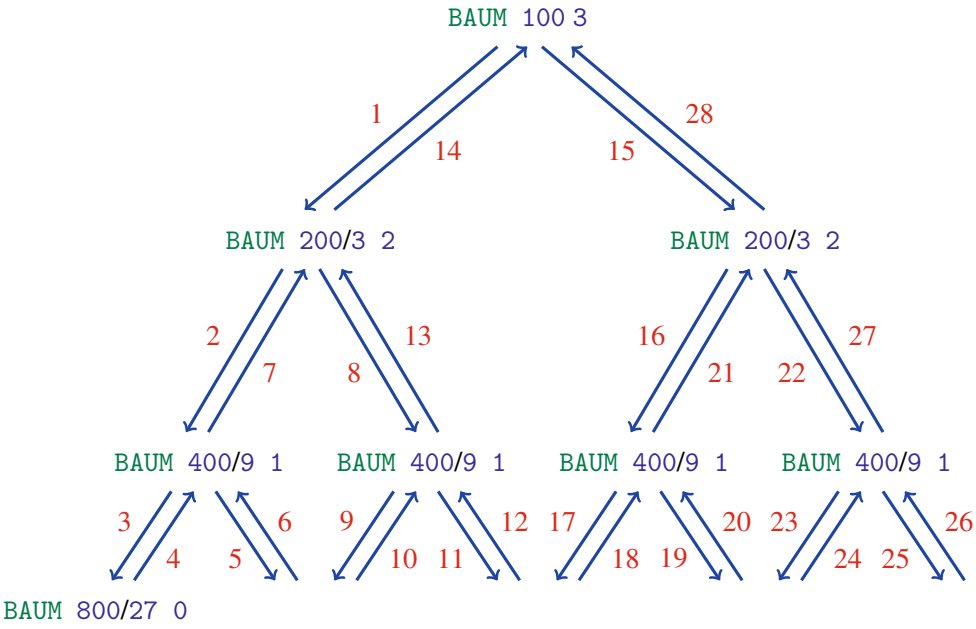


Abbildung 12.11

Die Tabelle 12.3 stellt die Entwicklung der Speicherinhalte dar, die den ersten neun Pfeilen des Baumdiagramms aus Abb. 12.11 entsprechen. Die untersten Werte in dieser Tabelle sind immer die Werte der Variablen `:H` und `:TIEF`, mit denen der Rechner aktuell arbeitet.

	0	1	2	3	4	5	6	7	8	9
H	100	100	100	100	100	100	100	100	100	100
TIEF	3	3	3	3	3	3	3	3	3	3
H(BAUM 200/3 2)	–	$\frac{200}{3}$	$\frac{200}{3}$	$\frac{200}{3}$	$\frac{200}{3}$	$\frac{200}{3}$	$\frac{200}{3}$	$\frac{200}{3}$	$\frac{200}{3}$	$\frac{200}{3}$
TIEF(BAUM 200/3 2)	–	2	2	2	2	2	2	2	2	2
H(BAUM 400/9 1)	–	–	$\frac{400}{9}$	$\frac{400}{9}$	$\frac{400}{9}$	$\frac{400}{9}$	$\frac{400}{9}$	–	$\frac{400}{9}$	$\frac{400}{9}$
TIEF(BAUM 400/9 1)	–	–	1	1	1	1	1	–	1	1
H(BAUM 800/27 0)	–	–	–	$\frac{800}{27}$	–	$\frac{800}{27}$	–	–	–	$\frac{800}{27}$
TIEF(BAUM 800/27 0)	–	–	–	0	–	0	–	–	–	0

Tabelle 12.3

□

Aufgabe 12.31 Zeichne das Baumdiagramm und die entsprechenden Klammerfolgen der folgenden rekursiven Aufrufe:

a) `BAUM 80 4`

b) `FELDABREC 200 2`

c) `STAR 400 3`

Aufgabe 12.32 Vervollständige die Tabelle 12.3, indem du die Entwicklung der Speicherinhalte für den Aufruf `BAUM 100 3`, der Struktur der rekursiven Aufrufe aus Abb. 12.11 auf der vorherigen Seite entsprechend aufzeichnest. Somit sollte die Tabelle 29 Spalten haben.

Aufgabe 12.33 Zeichne die ersten 12 Spalten einer Tabelle der Speicherinhalte (analog zu Tabelle 12.3) für die folgenden Aufrufe:

a) `BAUM 120 4`

b) `BAUM 243 5`

Aufgabe 12.34 Entwickle ein rekursives Programm, das für die ersten drei Rekursionstiefen die Bilder aus Abb. 12.12 zeichnet.

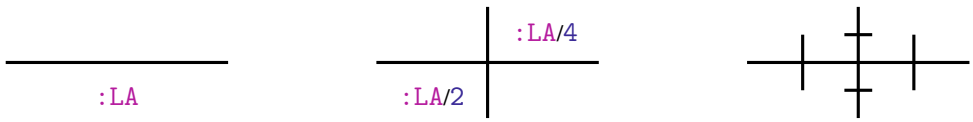


Abbildung 12.12

Aufgabe 12.35 Entwickle ein rekursives Programm, das für die ersten drei Rekursionstiefen die Bilder aus Abb. 12.13 auf der nächsten Seite zeichnet. Zeichne danach das Baumdiagramm der rekursiven Aufrufe für einen rekursiven Aufruf der Tiefe 3.

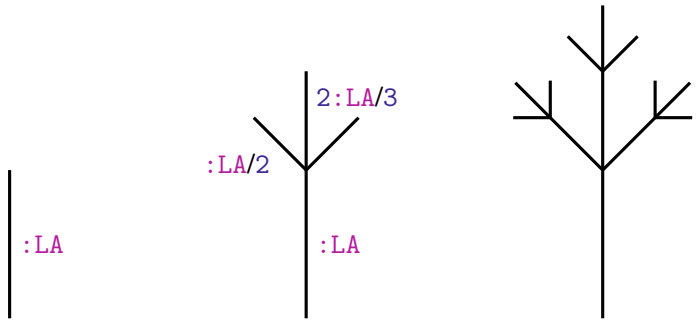


Abbildung 12.13

Zusammenfassung

Rekursive Programme sind Programme, die sich in ihrem Körper selbst aufrufen. Die Ausführung solcher Programme kann unendlich lange laufen. Um endliche Laufzeiten zu garantieren, muss man in die rekursiven Programme den bedingten Befehl `stop` einbauen. Außerdem müssen sich die Variablenwerte bei rekursiven Aufrufen so ändern, dass irgendwann die Haltebedingung erfüllt ist.

Aus Sicht der Mathematik kann man die Rekursion als eine Reduktion einer Aufgabe auf eine gleichartige Aufgabe kleinerer Größe ansehen. Bei vielen Aufgaben kann man sich aussuchen, ob man ein rekursives Programm entwickelt oder geschickt Schleifen verwendet. Es gibt aber Folgen von Bildern, deren Erzeugung auf der Anwendung eines rekursiven Musters basieren. Hier ist der Entwurf von rekursiven Programmen die natürlichste Vorgehensweise. Dabei ist es wichtig zu erkennen, welche Teile des Bildes durch einen rekursiven Aufruf kleinerer Tiefe und welche im eigentlichen Aufruf gezeichnet werden sollen.

Bei der Entwicklung von rekursiven Programmen verwenden wir den Befehl `wait`, um die Bewegung der Schildkröte zu verlangsamen und die Stelle einer falschen Anweisung zu entdecken. Die Anweisung `wait 100` unterbricht die Ausführung eines Programms in XLOGO für eine Sekunde.

Bei der Verwendung der rekursiven Aufrufe ist es wichtig zu wissen, dass die Ausführung eines Aufrufs erst dann beendet wird, wenn alle in ihm enthaltenen rekursiven Aufrufe

vollständig durchgeführt worden sind. Die Tiefe eines rekursiven Aufrufs ist die maximal Anzahl ineinander verschachtelter Aufrufe. Wenn der Körper eines rekursiven Programms genau einen rekursiven Aufruf enthält, entspricht die Tiefe der Anzahl der rekursiven Aufrufe. Ansonsten kann die Anzahl der Aufrufe exponentiell in der Tiefe der Aufrufe sein. Die Tiefe kann man auch als die Länge des längsten Wegs nach unten in dem Baumdiagramm der rekursiven Aufrufe bestimmen.

Kontrollfragen

1. Wann nennen wir ein Programm rekursiv?

2. Wie kann man ein rekursives Programm bauen, das unendlich lange läuft und trotzdem nichts zeichnet?

3. Wie kann man ein rekursives Programm bauen, das unendlich lange zeichnet?

4. Wie kann man ein rekursives Programm entwerfen, das endlich lange läuft?

5. In einem rekursiven Aufruf *A* kommt ein anderer rekursiver Aufruf *B* vor. Wird die Ausführung von *A* oder von *B* zuerst beendet?

6. Wie kann man mittels Baumdiagramm oder Klammerfolgen die Ausführung der rekursiven Aufrufe darstellen?

7. Was ist die Tiefe eines rekursiven Aufrufs? Wie kann man sie bestimmen?

8. Für welche Programme ist die Tiefe der rekursiven Aufrufe gleich der Anzahl rekursiver Aufrufe?

9. Wie groß kann die Anzahl rekursiver Aufrufe innerhalb eines Aufrufs im Vergleich mit der Tiefe des Aufrufs sein?

10. Für welche Art von Aufgaben eignen sich rekursive Programme besonders gut?

Kontrollaufgaben

1. Entwickle ein rekursives Programm zum Zeichnen unendlicher Spiralen von innen nach außen. Die Startlänge, Seitenverlängerung sowie die Anzahl Ecken sollen frei wählbar sein.

2. Die gleiche Aufgabe wie in Kontrollaufgabe 1, aber das Programm soll aufhören zu zeichnen, wenn die Seitenlänge einen frei wählbaren Wert `:MAX` überschritten hat.

3. Entwirf ein rekursives Programm, das in den ersten drei Rekursionsstufen die Bilder aus Abb. 12.14 zeichnet.

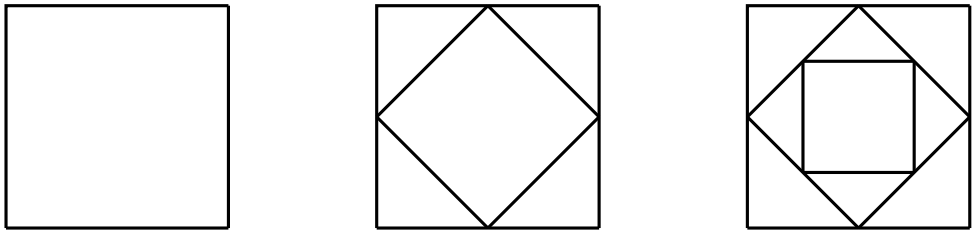


Abbildung 12.14

4. Entwirf ein rekursives Programm, das in den ersten drei Rekursionsstufen die Bilder aus Abb. 12.15 zeichnet. Die Seitenlänge des ersten Dreiecks soll frei wählbar sein.

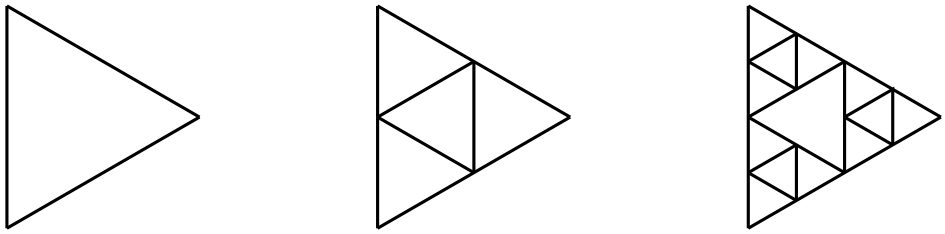


Abbildung 12.15

5. Wandle das Programm `KREISREC` aus Beispiel 12.2 zum Zeichnen der Bilder aus Abb. 12.4 auf Seite 221 so um, dass für die ersten 16 Rekursionsstufen die Kreise unterschiedlicher Größe in unterschiedlichen Farben gezeichnet werden.

6. Betrachte das folgende Programm:

```
to BAUM4 :TIEF :LA
if :TIEF=0 [ fd :LA lt 90 repeat 360 [ fd 1/20 rt 1 ]
             rt 90 bk :LA stop ]
fd :LA lt 45
BAUM4 :TIEF-1 :LA/2
rt 45 wait 250 bk :LA
fd :LA lt 45
BAUM4 :TIEF-1 :LA/2
lt 45 wait 250 bk :LA
fd 1.5*:LA
BAUM4 :TIEF-1 :LA/(sqrt2)
bk 1.5*:LA
end
```

   Überlege dir durch das Zeichnen von Bildern, was das Programm bei Aufrufen mit Rekursionsstufen 1, 2 und 3 macht. Überprüfe deine Überlegungen danach mit entsprechenden Aufrufen.

   Modifiziere das Programm so, dass der Stamm und die Äste des Baumes grün und die Kugeln blau gezeichnet werden. Teste dein Programm mit den Aufrufen

```
pu bk 150 pd BAUM4 4 150
pu bk 150 pd BAUM4 5 150.
```

7. Entwickle ein Programm, das unendlich lange läuft, indem es ständig einen Kreis mit gegebenem Umfang `:UM` in einer jeweils anderen Farbe ausmalt. Dabei soll das Programm mit der Farbe 1 anfangen, bis 16 fortsetzen und danach wieder mit der Farbe 1 beginnen, usw.

Lösungen zu ausgesuchten Aufgaben

Aufgabe 12.4
Diese Änderung ändert nichts an dem äußeren Verhalten des Programms. Es zeichnet die gleiche Spirale wie vorher. Das kommt dadurch zustande, weil der Aufruf

```
SPIRINF :LA+2
```

der Variable `:LA` des Programms `SPIRINF` einen um 2 größeren Wert als vorher zuordnet. Das entspricht dem Ergebnis, das entsteht, wenn man zuerst den Wert von `:LA` durch

```
make "LA :LA+2
```

erhöht und erst dann mit

```
SPIRINF :LA
```

das Programm SPIRINF aufruft.

Nachdem du die ganze Lektion bearbeitet hast, kannst du erklären, was für eine Auswirkung diese Programmänderung auf die Entwicklung der Speicherinhalte während der Ausführung des Programms hat.

Aufgabe 12.5
Die Änderung der Reihenfolge der letzten zwei Zeilen würde dazu führen, dass man statt einer unendlichen Spirale unendlich lange ein :LA×:LA-Quadrat zeichnet. Weil der Befehl

```
make "LA :LA*2
```

jetzt hinter dem Aufruf

```
SPIRINF :LA
```

stünde, würde es nie zu seiner Ausführung kommen. Damit wird der Wert :LA nie geändert. Die Folge wäre, dass die Zeile

```
fd :LA rt 90 wait 1000 pr :LA
```

unendlich oft ausgeführt werden würde. Dank des Befehls pr: :LA kannst du auch in der angezeigten Zahl auf dem Bildschirm beobachten, dass sich der Wert von :LA nie ändert.

Aufgabe 12.7
Um das Programm für die Zeichnung einer mehreckigen Spirale zu entwickeln, reicht es aus, eine neue globale Variable :ECK dem Programm SPIRINF hinzuzufügen und den Befehl rt 90 durch den Befehl rt 360/:ECK zu ersetzen.

Aufgabe 12.10
Unter einer kreisförmigen Spirale kann man sich unterschiedliche Objekte vorstellen. Eine Möglichkeit ist, ähnlich wie bei der Zeichnung eines Kreises vorzugehen und nach jeder Drehung lediglich die Seitengröße ein kleines bisschen zu vergrößern. Dieser Ansatz führt zu folgendem Programm.

```
to KREISSPIR :ADD  :GR
fd :GR make "GR :GR+:ADD rt 1
KREISSPIR :ADD  :GR
end
```

Probiere es mit dem Aufruf `KREISSPIR 0.01  1` aus. Eine andere Idee ist es, einen Halbkreis zu zeichnen und dann mit einem größeren Halbkreis fortzufahren, usw. Bei diesem Ansatz kann das Programm wie folgt aussehen:

```
to KREISSPIR1 :ADD  :GR
repeat 180 [fd :GR rt 1]
make "GR :GR+:ADD
KREISSPIR1 :ADD  :GR
end
```

Wie kann das Programm `KREISSPIR1` modifiziert werden, um immer doppelt so große Halbkreise zu zeichnen? Was müsste man in `KREISSPIR1` ändern, um den Radius immer nach der Zeichnung eines Viertelkreises zu erhöhen?
Kannst du die Programme `KREISSPIR` und `KREISSPIR1` so umschreiben, dass sie nicht rekursiv sind und trotzdem die gleiche, unendliche kreisförmige Spirale zeichnen?

Aufgabe 12.11
Es reicht, die Zeile

```
repeat 7 [fd :GR rt 90] rt 90 wait 1000
```

`:N`-Mal auszuführen. Mit einer while-Schleife kann man es wie folgt erreichen:

```
to FELDREC1 :GR  :N
while [:N>0] [ repeat 7 [fd :GR rt 90] rt 90
               wait 1000 make "N :N-1]
end
```

Mit der repeat-Schleife geht es noch einfacher, weil wir keine Kontrollvariable `:N` brauchen und `:N` deshalb einfach als Parameter verwenden dürfen.

```
to FELDREC2 :GR  :N
repeat :N [ repeat 7 [ fd :GR rt 90] rt 90 ]
end
```

Aufgabe 12.16
Der Aufruf

```
SPIRRECHTREC 10  100  1  2  250
```

beginnt mit der Zeichnung der Linien mit der Länge :MIN1=10 und :MIN2=100. Danach wird :MIN1 um 1 und :MIN2 um 2 vergrößert und

```
SPIRRECHTREC 11 102 1 2 250
```

aufgerufen. In diesem Aufruf ist dann der Aufruf

```
SPIRRECHTREC 12 104 1 2 250
```

verschachtelt. Um

```
SPIRRECHTREC 12 104 1 2 250
```

zu beenden, muss man

```
SPIRRECHTREC 13 106 1 2 250
```

aufrufen und diesen Aufruf vollständig auszuführen. Das Programm SPIRRECHTREC endet erst, wenn eine der Seitengrößen :MIN1 oder :MIN2 den maximalen Seitenwert :MAX=250 übersteigt. Wir sehen, dass es bei diesem Aufruf mit dem Variablenwert von :MIN2 zuerst passiert. Genauer gesagt wird der Aufruf

```
SPIRRECHTREC 86 252 1 2 250
```

der letzte und somit der am tiefsten verschachtelte Aufruf sein. Damit haben wir 76 in sich verschachtelte Aufrufe und die Tiefe des Aufrufs SPIRRECHTREC 10 100 1 2 250 ist folglich 76.

Der Aufruf

```
SPIRRECHTREC :MIN1 :MIN2 :ADD1 :ADD2 :MAX
```

hat als tiefsten, verschachtelten Aufruf

```
SPIRRECHTREC     :MIN1+T*:ADD1 :MIN2+T*:ADD2 :ADD1 :ADD2
:MAX
```

mit der Tiefe T mit :MIN1+T*:ADD1>:MAX oder :MIN2+T*:ADD2>:MAX. Somit ist die Tiefe die kleinste natürliche Zahl T mit der Eigenschaft

$$T > \frac{\texttt{:MAX} - \texttt{:MIN1}}{\texttt{:ADD1}} \quad \text{oder} \quad T > \frac{\texttt{:MAX} - \texttt{:MIN2}}{\texttt{:ADD2}}.$$

Lektion 13

Integrierter LOGO- und Mathematikunterricht: Trigonometrie

Du hast schon in der Trigonometrie gelernt, was die trigonometrischen Funktionen sind. Wir wiederholen kurz, wie man auf die Idee kam, sie überhaupt einzuführen. Betrachte Abb. 13.1 auf der nächsten Seite mit zwei Strahlen aus dem Punkt S, die den Winkel α einschließen. Wir bilden rechtwinklige Dreiecke $\triangle SA_1B_1$, $\triangle SA_2B_2$ und $\triangle SA_3B_3$, so dass die Seiten A_1B_1, A_2B_2 und A_3B_3 parallel verlaufen und mit der Gerade $\overline{SA_3}$ rechte Winkel bilden. Die Strahlensätze besagen, dass

$$\frac{|\overline{SA_1}|}{|\overline{SB_1}|} = \frac{|\overline{SA_2}|}{|\overline{SB_2}|} = \frac{|\overline{SA_3}|}{|\overline{SB_3}|},$$

$$\frac{|\overline{A_1B_1}|}{|\overline{SB_1}|} = \frac{|\overline{A_2B_2}|}{|\overline{SB_2}|} = \frac{|\overline{A_3B_3}|}{|\overline{SB_3}|},$$

$$\frac{|\overline{A_1B_1}|}{|\overline{SA_1}|} = \frac{|\overline{A_2B_2}|}{|\overline{SA_2}|} = \frac{|\overline{A_3B_3}|}{|\overline{SA_3}|}.$$

In Worten ausgedrückt: Die Proportionen zwischen den Seiten bei allen Dreiecken mit gleichen Winkelgrößen sind identisch, egal wie groß sie sind. Das heißt für Abb. 13.1 auf der nächsten Seite, dass die Seitenlängen der Dreiecke gleich schnell (proportional) wachsen.

Aufgabe 13.1 Entwickle ein Programm, das für gegebene Werte von α, $|\overline{SA_1}|$, $|\overline{SA_2}|$ und $|\overline{SA_3}|$ das Bild aus Abb. 13.1 auf der nächsten Seite zeichnet. Dabei dürfen die Strecken $\overline{A_1B_1}$, $\overline{A_2B_2}$ und $\overline{A_3B_3}$ als Geraden gezeichnet werden. Die Beschriftung der Punkte durch Buchstaben ist nicht erforderlich.

J. Hromkovič, *Lehrbuch Informatik*, DOI 10.1007/978-3-8348-9692-6_14,

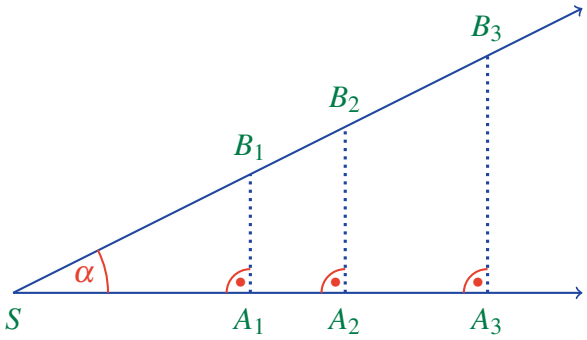


Abbildung 13.1

Wir können zwischen den Schenkeln des Winkels α alle Dreiecke mit den Winkeln α, $\gamma = 90°$ und $\beta = 90° - \alpha$ aufzeichnen (Abb. 13.2). Wir wissen schon, dass für drei gegebene Winkel, die zusammengezählt 180° ergeben, unendlich viele Dreiecke existieren. Die Ähnlichkeit dieser unendlich vielen Dreiecke kann man durch das erkannte Gesetz beschreiben. Das Gesetz gilt für alle Dreiecke mit gleichen Winkeln, aber wir interessieren uns hier nur für rechtwinklige Dreiecke. Den Bezeichnungen aus Abb. 13.2 folgend, können wir das Gesetz wie folgt formulieren: In allen Rechtecken mit den festen

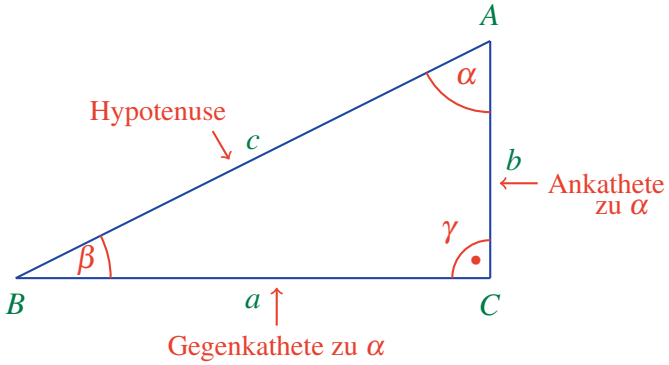


Abbildung 13.2

Winkelgrößen α, $\beta = 90° - \alpha$ und $\gamma = 90°$ sind die folgenden Zahlen

$$\frac{a}{c} = \frac{|\text{Gegenkathete zu } \alpha|}{|\text{Hypotenuse}|},$$
$$\frac{b}{c} = \frac{|\text{Ankathete zu } \alpha|}{|\text{Hypotenuse}|},$$
$$\frac{a}{b} = \frac{|\text{Gegenkathete zu } \alpha|}{|\text{Ankathete zu } \alpha|}$$

feste Zahlen (Parameter), die sich mit der Größe des Dreiecks nicht ändern. Diese Zahlen sind durch α bestimmt. Wenn etwas stabil ist (d.h. bei wechselnden Dreiecksgrößen unverändert bleibt), lohnt es sich, es zu benennen. So entstanden die Namen

$$\sin(\alpha) = \frac{|\text{Gegenkathete zu } \alpha|}{|\text{Hypotenuse}|}$$
$$\cos(\alpha) = \frac{|\text{Ankathete zu } \alpha|}{|\text{Hypotenuse}|}$$
$$\tan(\alpha) = \frac{|\text{Gegenkathete zu } \alpha|}{|\text{Ankathete zu } \alpha|}.$$

Die Werte von $\sin(\alpha)$, $\cos(\alpha)$ und $\tan(\alpha)$ wurden einmal ausgerechnet und in Tabellen für verschiedene Winkelgrößen α festgehalten. LOGO hat auch eine solche Tabelle. Dies kannst du leicht überprüfen. Wenn du

```
pr ( sin 30 )
```

in die Befehlszeile eingibst, erhältst du den entsprechenden Wert 0.5. Damit sind `sin`, `cos` und `tan` Befehlsnamen in LOGO, die als Parameter die Winkelgröße haben.

Wozu sind diese Befehle nützlich? Bisher konnten wir die fehlende Seitengröße in einem rechtwinkligen Dreieck durch den Satz des Pythagoras berechnen. Wir sind jetzt hiermit in der Lage aus der Kenntnis von nur einer Seitenlänge und zwei Winkelgrößen die beiden unbekannten Seitenlängen zu bestimmen.

Beispiel 13.1 Unsere Aufgabe ist es, ein Programm zu entwickeln, das für gegebene Hypotenusenlänge und Winkel α eines rechtwinkligen Dreiecks die Längen der beiden Katheten bestimmt und das Dreieck zeichnet. Wir wissen, dass

$$\sin(\alpha) = \frac{a}{c}$$

gilt und somit bestimmen wir a durch die Berechnung

$$a = c \cdot \sin(\alpha).$$

Jetzt kann man die Kathetenlänge b entweder mit dem Satz des Pythagoras oder einfacher durch

$$b = c \cdot \cos(\alpha)$$

ausrechnen. Wenn wir alle Winkel und Seitenlängen eines Dreiecks kennen, ist es einfacher, das Dreieck zu zeichnen. Wir können dies wie folgt umsetzen:

```
to DREIECKHA :ALPHA :c
make "a :c*sin(:ALPHA)
make "b :c*cos(:ALPHA)
rt 90 fd :a lt 90 fd :b
lt 180-:ALPHA fd :c rt 180-:ALPHA
end
```

□

Aufgabe 13.2 Schreibe ein Programm, das für die gegebenen Größen a und α eines rechtwinkligen Dreiecks (Abb. 13.2 auf Seite 240) die fehlenden Seitengrößen berechnet und das Dreieck zeichnet.

Aufgabe 13.3 Analog zu Aufgabe 13.2, nur für gegebene Größen b und α.

Aufgabe 13.4 Gegeben ist die Fläche eines rechtwinkligen Dreiecks (Abb. 13.2 auf Seite 240) und der Winkel α. Entwickle ein Programm, das alle Seitengrößen berechnet und das Dreieck zeichnet.

Die Aufgaben können auch umgekehrt gestellt werden. Einige Seitenlängen können bekannt sein. Die Aufgabe kann lauten die Winkel zu bestimmen.
Zum Beispiel sind a und c bekannt und die restlichen Größen des rechtwinkligen Dreiecks sollen bestimmt werden. Wenn man a und c kennt, kennt man auch den Quotienten $\frac{a}{c}$ und somit den Wert

$$\sin(\alpha) = \frac{a}{c}.$$

Die Tabelle für Sinus ist in der Mathematik auch nach den Werten geordnet. Somit kann man aus $\frac{a}{c}$ auch die Winkelgröße α ermitteln. Die inverse Funktion zu sin wird mit arcsin (auf dem Taschenrechner auch $\sin^{-1}$) bezeichnet und somit gilt

$$\alpha = \arcsin \frac{a}{c}.$$

Analog gilt für Tangens und Kosinus:

$$\alpha = \arccos \frac{b}{c} \qquad \text{und} \qquad \alpha = \arctan \frac{a}{b}.$$

XLOGO hat alle drei Befehle `arccos`, `arcsin` und `arctan`. SUPERLOGO hat nur den einen Befehl `arctan`, mit dem man Steigungen berechnen kann.

Aufgabe 13.5 Ein Auto fährt einen Bergpass hoch und muss dabei den Höhenunterschied `:H` überwinden. Die Steigung entspricht einem Winkel der Größe `:WIN`. Entwickle ein Programm, das für diese Eingabegrößen die Länge der Strecke berechnet und die Straße idealisiert als die Hypotenuse eines rechtwinkligen Dreiecks zeichnet.

Wir wollen uns jetzt damit beschäftigen, ein Programm zur Berechnung der Funktionen arcsin und arccos selbst zu entwickeln. Diese Aufgabe ist gar nicht so schwer. Um $\arcsin(x)$ für einen Wert x auszurechnen, können wir nacheinander die Werte $\sin(0)$, $\sin(1)$, $\sin(2)$ usw. ausrechnen. Wenn wir ein α finden, so dass

$$\sin(\alpha) < x < \sin(\alpha + 1)$$

gilt, dann wissen wir, dass

$$\alpha < \arcsin(x) < \alpha + 1$$

gilt. Damit können wir als Schätzung für $\arcsin(x)$ den Wert α, $\alpha + 1$ oder den Mittelwert $\frac{\alpha+\alpha+1}{2} = \alpha + \frac{1}{2}$ annehmen. Wem dies zu grob erscheint, kann sich die Folge der Werte $\sin(0)$, $\sin(0.1)$, $\sin(0.2), \ldots$ anschauen. Das folgende Programm hat einen Parameter `:APP` für die Wahl der Genauigkeit der Bestimmung. Wir wissen schon, dass für solche Proben die `while`-Schleife sehr gut geeignet ist. Für ein beliebiges Argument $x \in]0,1[$ können wir mit dem folgenden Programm $\arcsin(x)$ berechnen.

```
to ASINUS :X :APP
if :X=0 [ pr [ Fehler ] stop ]
if :X=1 [ pr [ Fehler ] stop ]
if :X>1 [ pr [ Fehler ] stop ]
if :X<0 [ pr [ Fehler ] stop ]
make "ALPHA :APP make "Y sin :ALPHA
while [ :X > :Y ]
      [ make "ALPHA :ALPHA+:APP make "Y sin :ALPHA ]
make "ALPHA :ALPHA-:APP/2
pr :ALPHA fd 100 bk 100 lt :ALPHA fd 100
end
```

Aufgabe 13.6 Entwickle ein eigenes Programm ACOSINUS zur Berechnung von $\arccos(x)$ für ein beliebiges Argument $x \in]0,1[$.

Aufgabe 13.7 Entwickle ein Programm, das für gegebene Seitenlängen a und b eines rechtwinkligen Dreiecks (Abb. 13.2 auf Seite 240) die fehlenden Winkel und die Seitenlänge c berechnet und das Dreieck zeichnet.

Aufgabe 13.8 Ein Astronaut steht auf dem Planeten A und schickt einen Strahl mit der Geschwindigkeit 300 000 km/s zur Mitte des Planeten B. Der Lichtstrahl kehrt nach :X Minuten zurück (Abb. 13.3).

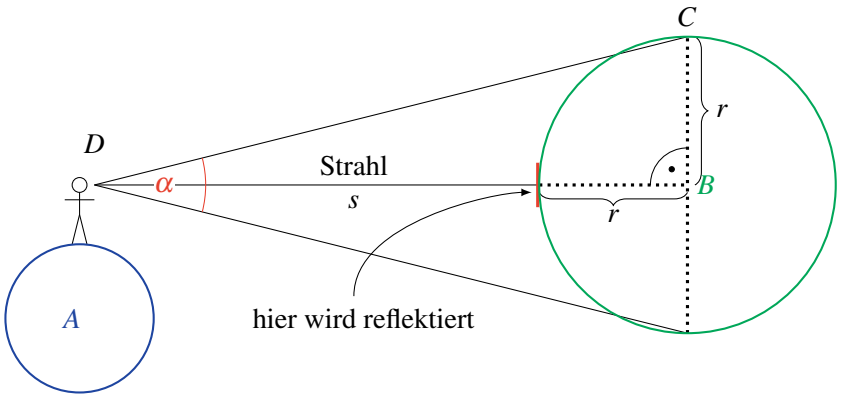


Abbildung 13.3

Der Winkel α, in dem der Astronaut den Planeten B beobachten kann, hat die Größe :ALPHA. Entwirf ein Programm, das für die gegebenen Daten :X und :ALPHA den Radius des Planeten B und die Entfernung des Astronauten vom Planeten B berechnet.

Beispiel 13.2 Gegeben sind der Radius r eines Kreises (Abb. 13.4) und ein Winkel α zwischen 0° und 180°. Es soll ein Programm entwickelt werden, das die Fläche des α-Segment (rot schraffiert in Abb. 13.4) berechnet.

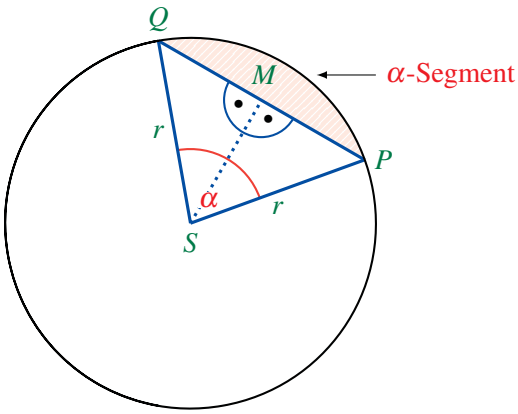


Abbildung 13.4

Es gelten

$$\text{Fläche}(\alpha\text{-Segment}) = \text{Fläche}(\alpha\text{-Sektor}) - \text{Fläche}(\triangle SPQ)$$

und

$$\text{Fläche}(\alpha\text{-Sektor}) = \frac{\alpha}{360} \cdot \pi r^2.$$

Das Hauptproblem ist, die Fläche von $\triangle SPQ$ zu bestimmen. Im Dreieck $\triangle SPM$, welches rechtwinklig ($\angle SMP = 90^\circ$) ist, gilt

$$\sin\left(\frac{\alpha}{2}\right) = \frac{|\text{Gegenkathete zu } \frac{\alpha}{2}|}{|\text{Hypotenuse}|} = \frac{|\overline{MP}|}{r} \qquad |\cdot r$$
$$|\overline{MP}| = r \cdot \sin\left(\frac{\alpha}{2}\right).$$

Es gilt auch

$$\cos\left(\frac{\alpha}{2}\right) = \frac{|\text{Ankathete zu } \frac{\alpha}{2}|}{|\text{Hypotenuse}|} = \frac{|\overline{MS}|}{r} \qquad |\cdot r$$
$$|\overline{MS}| = r \cdot \cos\left(\frac{\alpha}{2}\right).$$

Aufgrund von

$$\text{Fläche}(\triangle SPM) = \frac{1}{2} \cdot |\overline{MP}| \cdot |\overline{MS}|,$$

erhalten wir

$$\begin{aligned}\text{Fläche}(\triangle SPQ) &= 2 \cdot \text{Fläche}(\triangle SPM) \\ &= |\overline{MP}| \cdot |\overline{MS}| \\ &= r \cdot \sin\left(\frac{\alpha}{2}\right) \cdot r \cdot \cos\left(\frac{\alpha}{2}\right) \\ &= r^2 \cdot \sin\left(\frac{\alpha}{2}\right) \cdot \cos\left(\frac{\alpha}{2}\right).\end{aligned}$$

Endlich erhalten wir

$$\begin{aligned}\text{Fläche}(\alpha\text{-Segment}) &= \text{Fläche}(\alpha\text{-Sektor}) - \text{Fläche}(\triangle SPQ) \\ &= \frac{\alpha}{360} \cdot \pi r^2 - r^2 \cdot \sin\left(\frac{\alpha}{2}\right) \cdot \cos\left(\frac{\alpha}{2}\right) \\ &= r^2 \left[\left(\frac{\pi\alpha}{360}\right) - \sin\left(\frac{\alpha}{2}\right) \cdot \cos\left(\frac{\alpha}{2}\right)\right]. \\ &\quad \{\text{nach dem Distributivgesetz}\}\end{aligned}$$

Wenn die entsprechende Formel hergeleitet ist, kannst du selbst sicher schnell das Programm zur Berechnung der α-Segmentfläche aufschreiben. □

Aufgabe 13.9 Entwickle ein Programm, das für den gegebenen Kreisradius r und eine Zahl n die Fläche des im Kreis einbeschriebenen n-Ecks berechnet.

Aufgabe 13.10 Sei $\triangle ABC$ ein gleichschenkliges Dreieck. Entwickle ein Programm, das für die gegebene Höhe h und den Winkel α (Abb. 13.5) das Dreieck zeichnet und seine Fläche berechnet.

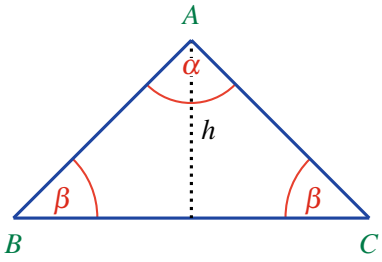


Abbildung 13.5

Zusammenfassung

Die Methoden zur Lösung unterschiedlicher Aufgabentypen lernt man besonders gut verstehen, wenn man sie in Programme umsetzt. Mit Hilfe der trigonometrischen Funktionen kann man in der Welt der Dreiecke aus gegebenen Informationen weitere Informationen ableiten oder sogar alles über das untersuchte Objekt erfahren.

Bei der Entwicklung von Programmen ist es wichtig zu beobachten, dass es nicht nur um das eigene Programmieren geht. Zuerst muss man das mathematische Denken anwenden, um das Problem zu analysieren und Zusammenhänge festzustellen. Erst wenn der Lösungsweg vollständig verstanden wurde, kann man mit dem Programmieren beginnen. Das muss aber nicht bedeuten, dass das Programmieren selbst nur eine lästige Routinearbeit ist. Manchmal braucht man neue Ideen, wie bei der Berechnung von `arcsin`, um mit den vorhandenen, beschränkten Mitteln einer Programmiersprache die mathematisch beschriebenen Lösungswege umzusetzen.

Kontrollfragen

1. Warum hat man die trigonometrischen Funktionen eingeführt? Was haben die Strahlensätze damit zu tun?

2. Aus welchen Daten über ein rechtwinkliges Dreieck kannst du alle anderen Seiten- und Winkelgrößen bestimmen?

3. Sei f eine wachsende Funktion, die durch ein Programm berechnet werden kann. Wie kann man mit Hilfe der `while`-Schleife für den gegebenen Wert $f(x)$ eine Annäherung des Wertes x berechnen? Wie kann man es mit beliebiger Genauigkeit machen?

4. Sei f eine Funktion, die man für jedes Argunment mit einem Programm berechnen kann. Sei [a,b] ein Intervall, in dem f genau ein Extremum hat. Wie kann man mit der Hilfe einer While-Schleife für einen frei wählbaren Wert des Parameters `:APP` ein i finden, sodass die Extremstelle im Intervall [$i\cdot$`APP`,$(i+1)\cdot$`APP`] liegt?

Kontrollaufgaben

1. Schreibe ein Programm, das für die drei gegebenen Werte $|\overline{SA_1}|$, $|\overline{SA_2}|$ und $|\overline{SA_3}|$ das Bild aus Abb. 13.1 auf Seite 240 zeichnet. Im Unterschied zu Aufgabe 13.1 müssen die Strecken $\overline{A_1B_1}$, $\overline{A_2B_2}$ und $\overline{A_3B_3}$ als Strecken und nicht als Geraden gezeichnet werden.

2. Entwickle ein Programm, das für gegebene Werte β und c eines rechtwinkligen Dreiecks alle anderen Seiten- und Winkelgrößen berechnet und das Dreieck zeichnet.

3. Entwirf ein eigenes Programm `ARCTANGENS` zur Berechnung von $\arctan(x)$ für ein beliebiges positives Argument x.

4. Entwirf ein Programm, das für eine gegebene natürliche Zahl $n \geq 3$ und einen gegebenen Kreisradius r die Fläche des den Kreis umschreibenden, regelmäßigen n-Ecks berechnet.

5. Entwickle ein Programm, das für eine gegebene positive Zahl n die n Dezimalstellen von π hinter dem Komma berechnet.

6. Man möchte die Länge eines Tunnels berechnen, der unter einem Berg hindurch führt. Die einzigen Daten, die man kennt, sind die Entfernungen a und b und der Winkel γ aus Abb. 13.6. Schreibe ein Programm, das für die gegebenen Werte von a, b und γ die Länge des Tunnels berechnet.

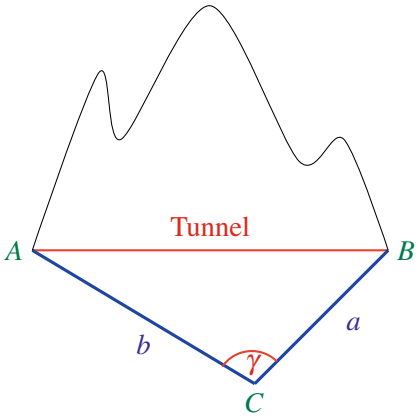


Abbildung 13.6

7. Entwickle ein Programm, das für die folgenden Funktionen und Intervalle [a,b] ihrer Argumente ein lokales Maximum näherungsweise findet. Die Genauigkeit der Näherung soll durch den Parameter `:APP` des Programms gegeben werden. Das Programm soll einen Wert `:X` ausgeben, so dass es ein lokales Maximum gibt, dessen Wert höchstens um `:APP` von `:X` abweicht.

   a) $f(\alpha) = \sin(\alpha) \cdot \cos(\alpha)$ für $\alpha \in [0, 90]$

   b) $f(x) = (20 - x) \cdot (20 + x)$ für $x \in [-10, 10]$

c) $f(x) = ax^3 + bx^2 + cx + d$ für $x \in [0, 100]$, wobei a, b, c, d beliebige Werte sind, die man als Eingaben des Programms betrachtet.

Lösungen zu ausgesuchten Aufgaben

Aufgabe 13.4
Wir kennen die Fläche F des Dreiecks $\triangle ABC$ aus Abb. 13.7 und die Winkelgröße von α.

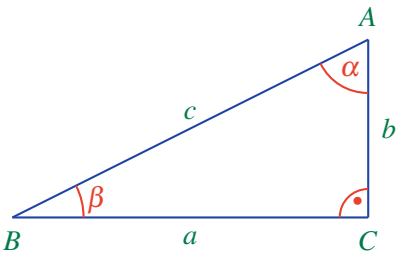


Abbildung 13.7

Wir wissen, dass

$$F = \frac{a \cdot b}{2} \qquad \text{und} \qquad \tan\alpha = \frac{a}{b}$$

gilt. Aus der ersten Gleichung erhalten wir

$$b = \frac{2F}{a}. \tag{13.1}$$

Wenn wir den Ausdruck für b in die zweite Gleichung $\tan\alpha = \frac{a}{b}$ einsetzen, erhalten wir

$$\tan\alpha = \frac{a^2}{2F}$$

und somit

$$a = \sqrt{2 \cdot F \cdot \tan\alpha} \tag{13.2}$$

Wenn wir a kennen, können wir es in (13.1) einsetzen, um b zu bestimmen. Die Seitengröße von c erhalten wir durch die Verwendung des Satzes von Pythagoras oder mittels

$$c = \frac{a}{\sin\alpha}$$

Damit können wir das Programm wie folgt schreiben:

```
to DRFLA :F :ALPHA
make "a sqrt(2*:F*tan(:ALPHA)
make "b 2*:F/:a
make "c :a/sin(:ALPHA)
rt 90 fd :b lt 180-:ALPHA
fd :c rt 270-:ALPHA
fd :a rt 180 end
```

Aufgabe 13.8
Die Entfernung s des Beobachters von der Oberfläche des Planeten B (s. Abb. 13.3 auf Seite 244) kann man einfach durch das physikalsiche Gesetz

$$\text{Strecke} = \text{Geschwindigkeit} \cdot \text{Zeit}$$

bestimmen. Weil der Strahl die Strecke s in der Zeit x zweimal durchläuft, erhalten wir

$$2s = 300\,000 \cdot x$$

und somit

$$s = 150\,000 \cdot x.$$

In Abb. 13.3 auf Seite 244 betrachten wir für die Bestimmung des Radius vom Planeten B die Strecke $\overline{CD}$ statt der Tangente. Bei großen Entfernungen ist der Unterschied vernachlässigbar. Im Dreieck zwischen D, C und dem Mittelpunkt des Planeten B gilt

$$\begin{aligned}
\sin\left(\frac{\alpha}{2}\right) &= \frac{r}{r+s} && \mid \cdot (r+s)\\
(r+s) \cdot \sin\left(\frac{\alpha}{2}\right) &= r && \mid -r \cdot \sin\left(\frac{\alpha}{2}\right)\\
s \cdot \sin\left(\frac{\alpha}{2}\right) &= \left(1 - \sin\left(\frac{\alpha}{2}\right)\right) \cdot r\\
r &= \frac{s \cdot \sin\left(\frac{\alpha}{2}\right)}{1 - \sin\left(\frac{\alpha}{2}\right)}
\end{aligned}$$

Jetzt kann man aus den entwickelten Formeln das Programm aufschreiben.

```
to PLANET :X :ALPHA
make "s 150000*:X
pr:s
make "d sin(:ALPHA/2)
make "r (:s*:d)/(1-:d)
pr :r
end
```

Lektion 14

Integrierter LOGO- und Mathematikunterricht: Vektorgeometrie

Die Programmiersprache LOGO kann uns helfen, Programme für Lösungen von vielen grundlegenden Aufgabenstellungen im zweidimensionalen Raum zu entwickeln und die Konstruktionen mit Hilfe der Schildkröte zu zeichnen. Wenn wir es schaffen, Aufgabenstellungen auf diese Art und Weise zu lösen, können wir sicher sein, dass wir die Lösungsmethode gut beherrschen, weil wir sie sogar einer Maschine ohne Intellekt beibringen können.

Wir fangen damit an, dass wir ein Programm zur Erzeugung von Koordinatensystemen schreiben. Die Einheiten sollen in Schritten frei wählbar sein, um mit beliebigen Zahlen graphisch umgehen zu können (siehe Abb. 14.1). Wie üblich gehen wir modular vor.

```
to ACHSE :AN :EINHEIT
repeat :AN [ fd :EINHEIT rt 90 fd 5
             bk 10 fd 5 lt 90]
bk :AN*:EINHEIT rt 180
repeat :AN [ fd :EINHEIT rt 90 fd 5
             bk 10 fd 5 lt 90]
bk :AN*:EINHEIT rt 180
end
```

J. Hromkovič, *Lehrbuch Informatik*, DOI 10.1007/978-3-8348-9692-6_15,

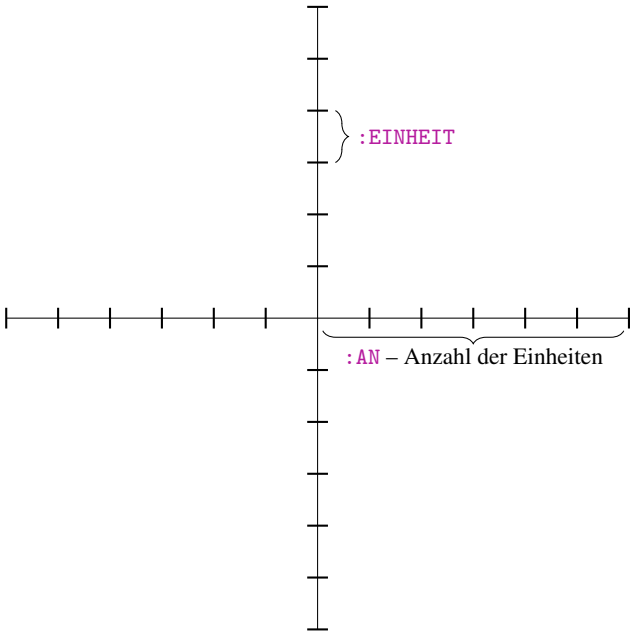


Abbildung 14.1

```
to KOOR :AN :EINHEIT
ACHSE :AN  :EINHEIT
rt 90
ACHSE :AN  :EINHEIT
lt 90
end
```

Aufgabe 14.1 Modifiziere das Programm KOOR so, dass man auf den verschiedenen Achsen sowohl die Anzahl, als auch die Schrittgröße der Einheiten wählen kann.

Punkte können wir in diesem Koordinatensystem durch kleine Kreise zeichnen:

```
to PUNKT :EINHEIT :X  :Y
KOOR 240/:EINHEIT  :EINHEIT
pu fd :Y*:EINHEIT rt 90 fd :X*:EINHEIT pd
KREISE 1/36
end
```

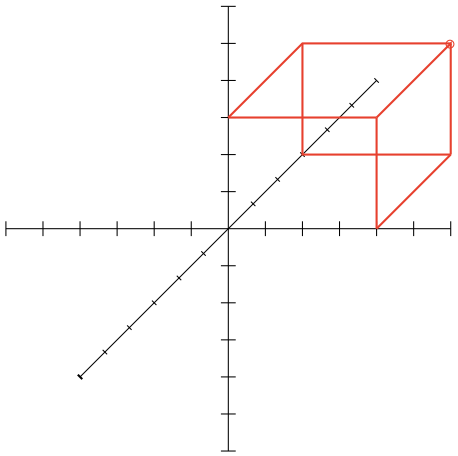


Abbildung 14.2 Punkt $(4,3,3)$ in einem dreidimensionalen Koordinatensystem

Aufgabe 14.2 Entwickle ein Programm zum Zeichnen eines Punktes in einem dreidimensionalen Raum (Abb. 14.2). Die Z-Achse soll unter dem Winkel von 45° mit $\frac{2}{3}$ der Größe der Einheiten auf der X- und Y-Achse dargestellt werden. Der Punkt mit den Koordinaten (X,Y,Z) soll als die hintere, rechte Ecke eines Würfels eingezeichnet werden.

Beispiel 14.1 Eine der grundlegenden Aufgaben in der Geometrie ist es, die Strecke zwischen zwei gegebenen Punkte $P_1 = (X_1,Y_1)$ und $P_2 = (X_2,Y_2)$ zu zeichnen. Wir werden im Weiteren Koordinatensysteme schwarz, Strecken rot und Geraden blau zeichnen. Die Punkte können wir schon einzeichnen. Um die Strecke $\overline{P_1P_2}$ zu zeichnen, müssen wir zuerst die Entfernung der beiden Punkte berechnen. Wie wir aus der Mathematik wissen (s. Abb. 14.3 auf der nächsten Seite), können wir Entfernungen durch den Satz des Pythagoras folgendermaßen berechnen:

$$\mathrm{DIS}(P_1,P_2) = \sqrt{(X_1-X_2)^2+(Y_1-Y_2)^2}$$

Zu diesem Zweck können wir das folgende Programm verwenden:

```
to ENTF :X1 :Y1 :X2 :Y2
make "DIST sqrt((:X2 - :X1) * (:X2 - :X1) + (:Y2 - :Y1)
* (:Y2 - :Y1))
pr :DIST
end
```

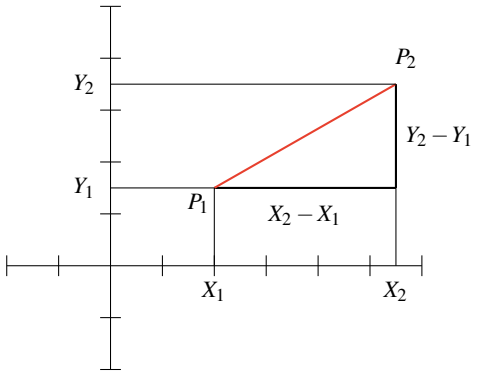


Abbildung 14.3

Jetzt müssen wir noch den Winkel der Strecke zur horizontalen X-Achse bestimmen. Wir sehen in Abb. 14.3, dass

$$\tan(\alpha) = \frac{Y_2 - Y_1}{X_2 - X_1} \qquad \text{oder} \qquad \sin(\alpha) = \frac{Y_2 - Y_1}{\text{DIST}}$$

gilt.

Damit wissen wir alles Notwendige, um die Strecke $\overline{P_1P_2}$ zeichnen zu können. Wir schreiben zuerst ein Programm, das für $X_2 \geq X_1$ korrekt arbeitet.

```
to STRECKE :EINHEIT :X1 :Y1 :X2 :Y2
setpc [0 0 0]
PUNKT :EINHEIT :X2 :Y2
wait 1000 pu bk :X2*:EINHEIT lt 90 bk :Y2*:EINHEIT pd
PUNKT :EINHEIT :X1 :Y1
make          "DIST          sqrt((:X2-:X1)*(:X2-:X1)
+(:Y2-:Y1)*(:Y2-:Y1))
pr :DIST
if :Y2>:Y1 [ make "ABSDY :Y2-:Y1]
           [ make "ABSDY :Y1-:Y2]
if :X1=:X2 [ make "ALPHA 90 ]
           [ make "ALPHA arctan (:ABSDY/abs (:X2-:X1))]
setpc [255 0 0]
if :Y2>:Y1 [ lt :ALPHA ] [rt :ALPHA ]
fd :DIST*:EINHEIT
end
```

Der Befehl `abs` berechnet für eine gegebene Zahl seinen absoluten Wert, also den positiven Wert der Zahl.

Aufgabe 14.3 Für den Fall $Y_2 - Y_1 > 0$ drehen wir `lt :ALPHA` nach links, weil es der Situation aus Abb. 14.3 auf der vorherigen Seite entspricht. Für den Fall $Y_2 \leq Y_1$ drehen wir nach rechts. Erkläre warum und zeichne das entsprechende Bild analog zu Abb. 14.3.

Aufgabe 14.4 Das Programm `STRECKE` arbeitet nur für den Fall $X_2 \geq X_1$ korrekt. Erweitere das Programm, so dass es auch für $X_2 < X_1$ korrekt arbeitet. Zeichne dazu die entsprechenden Bilder analog zu Abb. 14.3 auf der vorherigen Seite.

Aufgabe 14.5 Erweitere das Programm, so dass die Schildkröte am Ende die Startposition [0,0] in dem Koordinatensystem annimmt. Dabei darfst du den Befehl `home` nicht verwenden.

Wenn man das Programm `STRECKE` zur Zeichnung einer Geraden in blauer Farbe modifizieren will, reicht es aus, `setpc [255 0 0]` gegen `setpc [0 0 128]` und die letzte Zeile

```
fd :DIST*:EINHEIT
```

gegen

```
bk :DIST*:EINHEIT fd 3 * :DIST * :EINHEIT
```

auszutauschen. □

Aufgabe 14.6 Gegeben sind die vier Punkte P_1, P_2, P_3 und P_4. Schreibe ein Programm, das rechnerisch und zeichnerisch den möglichen Schnittpunkt der Geraden g_1 und g_2 bestimmt, wobei g_1 durch die Punkte P_1 und P_2 und g_2 durch die Punkte P_3 und P_4 verläuft.

Aufgabe 14.7 Seien $y_1 = A \cdot x + B$ und $y_2 = C \cdot x + D$ zwei Geraden. Entwickle ein Programm, das für die gegebenen Werte A, B, C und D rechnerisch und zeichnerisch den möglichen Schnittpunkt dieser beiden Geraden bestimmt.

Aufgabe 14.8 Gegeben sind drei Punkte: $P_1 = (X1, Y1)$, $P_2 = (X2, Y2)$ und $P_3 = (X3, Y3)$. Entwickle ein Programm, das Folgendes macht.

a) Das Programm zeichnet alle drei Punkte ein.

b) Falls alle drei Punkte auf einer Geraden liegen, dann zeichnet es diese Gerade.

c) Falls die Punkte nicht auf einer Geraden liegen, zeichnet es das Dreieck $\triangle P_1P_2P_3$.

Aufgabe 14.9 Gegeben ist ein Kreis durch seinen Mittelpunkt $M = (XK, YK)$ und Radius R und eine Gerade $y = A \cdot x + B$ durch die Werte A und B. Entwickle ein Programm, das zeichnerisch sowie rechnerisch die möglichen Schnittpunkte der Gerade mit dem Kreis bestimmt.

Beispiel 14.2 Es sind zwei Vektoren $(X1, Y1)$ und $(X2, Y2)$ gegeben. Unsere Aufgabe ist es nun, ein Programm zu entwickeln, das die Summe der Vektoren berechnet und graphisch wie in Abb. 14.4 darstellt. In der Zeichnung addiert man die „roten“ Vektoren $(4,1)$ und $(-2,5)$ und erhält den blauen Vektor $(2,6)$ als Resultat. Auch zeichnerisch beobachten wir, dass die Reihenfolge in der Addition keine Rolle spielt.

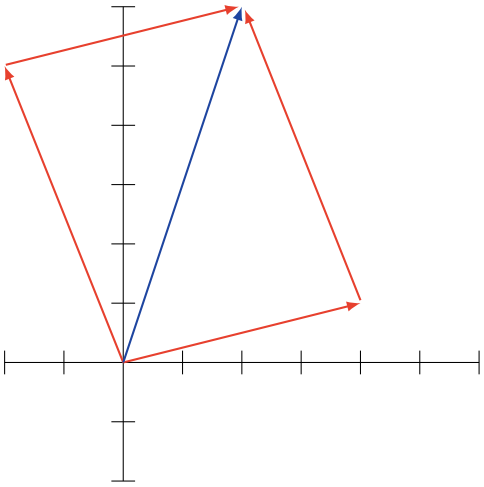


Abbildung 14.4

Um ein Programm zum Zeichnen der Vektoraddition zu entwickeln, modifizieren wir `STRECKE` zu `STRECKE0` durch die Einführung folgender Zeilen am Ende des Programms:

```
bk :DIST*:EINHEIT
if :X2>:X1 [ if :Y2>:Y1 [ rt :ALPHA ] [ lt :ALPHA ] ]
           [ if :Y2>:Y1 [ lt 180+:ALPHA ] [ rt
           180+:ALPHA ] ]
pu :X1*:EINHEIT lt 90 bk :Y1*:EINHEIT pd
```

Dadurch wird die Schildkröte nach dem Einzeichnen der Strecke an den Punkt $(0,0)$ zurückkehren und wie anfangs nach oben schauen. Dann können wir STRECK0 dreimal verwenden, um die beiden roten Vektoren vom Punkt $(0,0)$ aus und einen der restlichen beiden roten Vektoren zu zeichnen. Danach zeichnen wir mit STRECKE die letzte rote Linie zum Punkt $(X1+X2, Y1+Y2)$. Wenn man jetzt die Farbe auf Blau setzt und den Befehl home eingibt, wird der resultierende Vektor blau gezeichnet. Das entsprechende Programm kann wie folgt aussehen:

```
to VEKTORADD :EINHEIT :X1 :Y1 :X2 :Y2
STRECK0 :EINHEIT 0 0 :X1 :Y1
STRECK0 :EINHEIT 0 0 :X2 :Y2
STRECK0 :EINHEIT :X1 :Y1 :X1+:X2 :Y1+:Y2
STRECKE :EINHEIT :X2 :Y2 :X1+:X2 :Y1+:Y2
setpc [0 0 128] wait 2000
home
end
```

□

Aufgabe 14.10 Ändere das Programm VEKTORADD, so dass die Schildkröte am Ende die Startposition und die ursprüngliche Blickrichtung einnimmt, ohne jedoch den Befehl home zu verwenden.

Aufgabe 14.11 Entwirf ein Programm, das für einen gegebenen Wert :X (zwischen 0 und 1) den Einheitskreis aus Abb. 14.5 (ohne die Beschriftung X, Y und α) zeichnet. Damit die Zeichnung grösser dargestellt wird, kannst du für den Einheitskreis einen Radius von 150 Schritten wählen.

Aufgabe 14.12 Zeichne mit einem Programm den Einheitskreis wie in Aufgabe 14.11, aber so dass statt X,

a) der Wert der Y-Koordinate oder

b) der Winkel α

als Eingabe gegeben ist.

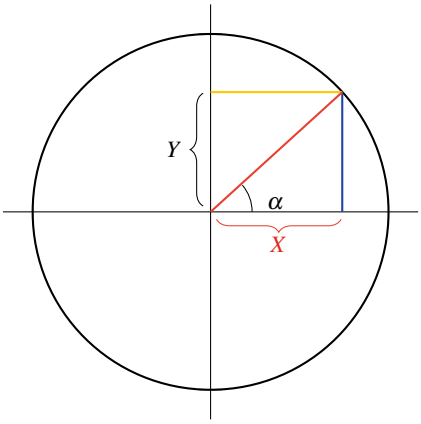


Abbildung 14.5

Aufgabe 14.13 Entwickle ein Programm zur Addition von zwei dreidimensionalen Vektoren. Um es überschaubar zu machen, müssen alle Punkte (wie in Abb. 14.2 auf Seite 253) eindeutig dargestellt werden.

Zusammenfassung

Wir haben gelernt, in LOGO Koordinatensysteme mit frei wählbaren Einheiten zu zeichnen. In diesen Koordinatensystemen können wir mittels LOGO-Programmen Punkte, Strecken zwischen zwei Punkten, Geraden und Kreise zeichnen. Alle diese Programme kann man als Basisinstrumente betrachten, mit denen man unterschiedliche Aufgaben der Geometrie und insbesondere der Vektorgeometrie rechnerisch und zeichnerisch lösen kann. So kann man zum Beispiel alle wichtigen Punkte rechnerisch bestimmen und dann mittels schon entworfener Programme Strecken zwischen diesen Punkten zeichnen oder aus diesen Punkten Kreise konstruieren. Auf diese Weise kann eine Vielfalt von Aufgaben im zweidimensionalen Raum rechnerisch und zeichnerisch gelöst werden.

Zeichnungen im dreidimensionalen Raum sind deutlich komplizierter, da sie in einem zweidimensionalen Raum realisiert werden müssen, denn der Bildschirm ist nun einmal flach. Aus diesem Grund haben wir hier nur elementare Aufgaben betrachtet.

Kontrollfragen

1. Wie wählst du die Größen der Einheiten für eine gegebene Aufgabe? Hast du eine Strategie für die Wahl der Einheitsgröße im Koordinatensystem bei einer gegebenen Anzahl von Punkten? Kannst du deine Strategie mittels eines Programms automatisieren?

2. Was für eine Größe haben Punkte in der Geometrie? Und wie kann man Punkte zeichnerisch darstellen? Siehst du mehrere gute Möglichkeiten?

3. Welche Bedeutung können Vektoren für die Modellierung der Realität haben?

4. Welcher grundlegende Satz der Geometrie wird verwendet, um die Entfernung zweier Punkte zu berechnen?

5. Wie addiert man zwei Vektoren? Was können Vektoren in der Physik darstellen und welche Bedeutung hat dabei die Addition?

Kontrollaufgaben

1. Gegeben sind drei unterschiedliche Punkte A, B und C durch ihre Koordinaten im zweidimensionalen Raum. Die Punkte B und C bestimmen eine Gerade g. Entwickle ein Programm, das Folgendes macht:

   a) Es zeichnet die Punkte A, B und C und die Gerade g.

   b) Es bestimmt rechnerisch den Punkt D auf g, der die kleinste Entfernung zu A hat und berechnet diese Entfernung.

   c) Es zeichnet die Strecke $\overline{AD}$.

   Das Programm soll auch korrekt arbeiten, wenn A auf g liegt. Alle bisher entwickelten Programme können als Unterprogramme verwendet werden.

2. Gegeben sind zwei Punkte A und B durch ihre Koordinaten im zweidimensionalen Raum. Entwickle ein Programm, das Folgendes macht:

   a) Es zeichnet einen Kreis mit dem Mittelpunkt auf der X-Achse, so dass die Punkte A und B auf dem Kreis liegen.

   b) Es zeichnet die Tangenten des Kreises, die durch den Punkt A bzw. durch den Punkt

B verlaufen.

c) Es berechnet den Schnittpunkt dieser Tangenten.

3. Gegeben sind drei Punkte *A*, *B* und *C* durch ihre Koordinaten im zweidimensionalen Raum, und wir wissen auch, dass sie nicht alle auf einer Geraden liegen. Entwickle ein Programm, das Folgendes macht:

   a) Es zeichnet einen Kreis, so dass die Punkte *A*, *B* und *C* auf dem Kreis liegen.

   b) Es zeichnet den Radius als eine Strecke zwischen dem Mittelpunkt des Kreises und dem Punkt *A*.

4. Gegeben sind zwei Vektoren durch ihre Richtungen (den Winkel, den sie mit der positiven X-Achse bilden) und ihre Längen. Entwickle ein Programm, das zeichnerisch diese Vektoren und ihre Summe darstellt.

5. Gegeben sind zwei Punkte *A* und *B*, die den Vektor $B-A$ bestimmen. Entwickle ein Programm, das für gegebene Koordinaten der Punkte *A* und *B* den Vektor $C-A$ zeichnet, wobei *C* der Mittelpunkt der Strecke $\overline{AB}$ ist.

6. Gegeben sind drei Punkte *A*, *B* und *C*, die nicht alle auf einer Gerade liegen, also das Dreieck $\triangle ABC$ bilden. Entwickle ein Programm, das Folgendes zeichnet:

   a) das Dreieck $\triangle ABC$

   b) den Schwerpunkt von $\triangle ABC$ als Schnittpunkt der Seitenhalbierenden

7. Gegeben sind zwei Vektoren $(X1, Y1)$ und $(X2, Y2)$ und zwei Zahlen *D1* und *D2*. Entwickle ein Programm, das die Strecke von $(0,0)$ aus zeichnet, die zuerst *D1* Einheiten in Richtung des Vektors $(X1, Y1)$ geht und danach *D2* Einheiten in Richtung des Vektors $(X2, Y2)$.

8. Gegeben sind die vier Punkte *A*, *B*, *C* und *D* durch ihre Koordinaten. Entwirf ein Programm, das rechnerisch wie zeichnerisch den Schnittpunkt der beiden Diagonalen des Vierecks $\square ABCD$ bestimmt. Falls die Punkte *A*, *B*, *C* und *D* kein Viereck bilden, soll das Programm eine Fehlermeldung ausgeben.

Lösungen zu ausgesuchten Aufgaben

Aufgabe 14.10
Um die ursprüngliche Startposition und die Startrichtung (ohne den Befehl `home`) zu erreichen, müssen wir

(i) die aktuelle Richtung α der Schildkröte nach der Zeichnung des ersten Vektors (der letzten roten Linie) und

(ii) die Steigung des resultierenden Vektors β kennen.

Wir sehen, dass

$$\tan(\alpha) = \frac{Y_1}{X_1} \quad \text{und somit} \quad \alpha = \arctan\left(\frac{Y_1}{X_1}\right)$$

gilt. Die Steigung des resultierenden Vektors β berechnen wir mit

$$\tan(\beta) = \frac{Y_1 + Y_2}{X_1 + X_2} \quad \text{und somit} \quad \beta = \arctan\left(\frac{Y_1 + Y_2}{X_1 + X_2}\right).$$

Um das gewünschte Verhalten zu erzielen, können wir jetzt den Befehl `home` im Programm `VECTORADD` durch folgende Befehle ersetzen:

```
make "ALP arctan (:Y1/:X1)
pr :ALP lt 90-:ALP wait 500
make "BET arctan ((:Y1+:Y2/(:X1+:X2))
pr :BET rt 90-:BET wait 500
make "RES (:X1+:X2)*(:X1+:X2)+ (:Y1+:Y2)* (:Y1+:Y2)
make "RES sqrt :RES
bk :RES*:EINHEIT lt 90-:BET
```

Das entworfene Programm bringt wie gefordert die Schildkröte an die Startposition. Aber für einige Werte von `:X1`, `:X2`, `:Y1` und `:Y2` schaut die Schildkröte am Ende nach unten anstatt nach oben. Kannst du erklären, für welche Werte der Eingaben es zu der Drehung der Schildkröte um 180° kommt? Kannst du das entworfene Programm so korrigieren, dass am Ende die Schildkröte immer nach oben schaut?

Modul II

Geschichte und Begriffsbildung

Vorwort und Zielsetzungen

Im ersten Modul „Vorkurs Programmieren“ haben wir programmiert und wir wissen, dass Programmieren zum Handwerk und dadurch zu den Grundfertigkeiten eines Informatikers gehört. Aber das Programmieren allein macht noch keinen zum Informatiker. Die erste Zielsetzung dieses Kapitels ist zu vermitteln, wie Wissenschaftsdisziplinen entstehen und wie die innere Logik ihrer Entwicklung ist. Nur in diesem Rahmen haben wir eine realistische Möglichkeit, die Informatik als eine Wissenschaftsdisziplin vorzustellen, die einerseits aus den Kenntnissen und Methoden anderer Wissenschaften schöpft und andererseits diese durch eigene Konzepte und Entdeckungen bereichert. Bei der Vermittlung des Aufbaus der Wissenschaften präsentieren wir einen der wichtigsten Grundbausteine. Wir definieren den Begriff der Folgerung (der Implikation) und erklären, was es genau bedeutet, *logisch zu denken* und *korrekt zu argumentieren*. Weil es uns nicht nur um die Präsentation eines Beispiels der Grundbausteine der Wissenschaften geht, sondern auch um ein tiefes Verständnis der Beweisführung, widmen wir dem mehr Zeit und üben, direkte und indirekte Beweise zu führen. Die Fähigkeit, richtige Schlussfolgerungen zu ziehen, ist für das Studium der Informatik unumgänglich und wir werden sie in mehreren Modulen intensiv verwenden.

Wir wollen hier einen kurzen Einblick in die Informatik aus Sicht der Grundlagenforschung geben. Wie wir sehen, spielt dabei die Begriffsbildung eine zentrale Rolle. Wir erklären hier, warum gerade sie essenziell für die Formulierung der wichtigsten Konzepte und Beiträge der Informatik ist. Die ersten und zentralsten Begriffe der Informatik sind „Algorithmus“ und „Programm“. Ohne auf formale, mathematische Modelle zurückzugreifen, erklären wir sorgfältig die Bedeutung dieser Begriffe und ihren Unterschied. Dabei lernen wir genau, was Programmieren bedeutet und was sich in einem Rechner bei der Programmbearbeitung abspielt. Diese Kenntnisse sind hilfreich für das Erlernen einer höheren Programmiersprache. Sie bilden die unumgängliche Grundlage für das Modul über die „Berechenbarkeit“, indem wir die Grenzen der Automatisierung (des algorithmischen Rechnens) entdecken werden.

Das folgende Modul ist folgendermaßen strukturiert: Lektion 1 widmet sich den Fragen

„Was ist Informatik?“ und „Wie sind die Wissenschaftsdisziplinen aufgebaut?“. In Lektion 2 lernt man, die Grundprinzipien der korrekten Argumentation kennen sowie direkte und indirekte Beweise zu führen. Eine kurze Geschichte der Ideenentwicklung in der Grundlagenforschung der Informatik erzählen wir in Lektion 3. Dort machen wir auf die Unterrichtsmodule aufmerksam, die später diese Themen behandeln werden. In Lektion 4 entwickeln wir anhand des Kuchenbackens ein erstes Grundverständnis für die Bedeutung der Begriffe Algorithmus und Programm. Unser Wissen über Algorithmen und Programme vertiefen wir in Lektion 5. Dabei lernen wir, in einer einfachen, dem Assembler ähnlichen Sprache, zu programmieren. Dadurch verstehen wir die Abläufe im Rechner bei den Durchführungen unserer Befehle. Dort beschäftigt man sich mit dem Risiko, in die endlosen Wiederholungen einer Schleife zu geraten.

Die Lektion 6 ist der indirekten Adressierung gewidmet, die eine der wichtigsten Programmierkonzepte auf der Ebene der Maschinenprogrammierung darstellt. Wie üblich schließen alle Lektionen mit einer kurzen Zusammenfassung und einer kurzen Sammlung von Kontrollfragen und Aufgaben.

Lektion 1

Was ist Informatik und wie wurden Wissenschaftsdisziplinen aufgebaut?

Einzelne Wissenschaftsdisziplinen nur als eine Zusammensetzung von Forschungsresultaten und Entdeckungen anzusehen, liefert ein falsches Bild. Noch schlimmer ist es, wenn man sie nur durch ihre Anwendungen im alltäglichen Leben anschaut. Wie würde wohl die Definition der Physik aussehen, wenn sie sich ausschließlich auf die Beschreibung der von Menschen hergestellten Geräte stützte? Fast alles, was Menschen jemals hergestellt haben (von Häusern bis zu Maschinen und Geräten aller Art), basiert auf der Kenntniss von physikalischen Gesetzen und trotzdem hält niemand den TV- oder Computerhersteller und schon gar nicht jeden TV-Zuschauer oder Computerbenutzer für einen Physiker. Wir trennen hier klar zwischen der physikalischen Grundlagenforschung und den technischen Anwendungen in der Elektrotechnik oder im Maschinenbau. Die Fähigkeit Geräte zu benutzen, verbindet man, mit Ausnahme des Rechners in der Öffentlichkeit mit keiner Wissenschaft.

Warum assoziiert man dann die Fähigkeit, gewisse Softwaresysteme zu benutzen, mit der Informatik? Welchen allgemeinen Bildungswert hat die Vermittlung dieser Fähigkeiten, wenn Softwaresysteme sich durch die ständige Entwicklung alle paar Jahre wesentlich ändern? Ist die relative Kompliziertheit des Rechners im Vergleich zu anderen Geräten der einzige Grund für diese Missentwicklung?

Sicherlich ist die Nutzung von Computern so verbreitet, dass die Anzahl der Autofahrer ungefähr der Anzahl der Computernutzer entspricht. Aber lernen wir in der Schule in einem Spezialfach für den Führerschein? Bald werden sich Mobiltelefone zu kleinen und leistungsfähigen Rechnern entwickeln. Wird der Umgang mit ihnen in einem neuen Fach

J. Hromkovič, *Lehrbuch Informatik*, DOI 10.1007/978-3-8348-9692-6_16,

an der Schule vermittelt werden? Ich möchte mich jetzt mit der Beantwortung dieser Fragen nicht zu viel beschäftigen. Die gängige Praxis in mehreren Ländern zeigt, dass man gute Anwenderkenntnisse durch in andere Fächer integrierten Unterricht oder kleine Blockkurse erfolgreich erwerben kann.

Hinweis für die Lehrperson Der Rest dieser Lektion ist relativ abstrakt und hat philosophische Tiefe. Es muss daher damit gerechnet werden, dass nicht alles beim ersten Mal verstanden wird. Es ist lohnenswert, zu dieser Lektion nach neuen Erfahrungen mit der Informatik aus anderen Lektionen und Modulen hierher zurückzukehren. Wir empfehlen, diese Lektion langsam vorzubringen und die Zuhörer die ganze Zeit so viel wie möglich in eine Diskussion einzubeziehen.

Unsere zentrale Frage ist:

„Was ist Informatik?“

Es ist schon jetzt klar, dass es nicht die Fähigkeit sein kann, einen Rechner zu benutzen, sonst würden bald fast alle Menschen Informatiker sein. Die Informatik selbst liefert nach außen auch deswegen kein klares Bild von sich, weil man sie nicht eindeutig den Naturwissenschaften oder den Ingenieurwissenschaften zuordnen kann. Die Situation ist so, als wenn die Physik, die Elektrotechnik und der Maschinenbau in nur einer Wissenschaftsdisziplin und unter einem Namen vereinigt wären. Aus der Sicht der Softwareherstellung ist Informatik eine angewandte Ingenieurwissenschaft mit allen Merkmalen einer technischen Disziplin, die die Entwicklung und Herstellung komplexer Systeme und Produkte anstrebt. Die Grundlagen der Informatik sind eher mathematisch-naturwissenschaftlicher Natur und die theoretische Informatik spielt für die Softwareentwicklung eine ähnliche Rolle wie die theoretische Physik für die technischen Disziplinen. Und gerade die mangelnde Kenntnis dieser Grundlagenforschung in der Öffentlichkeit ist verantwortlich für die falschen Vorstellungen über die Informatik.

Wir wissen jetzt, dass der bestmögliche Zugang zum Verständnis der Informatik nicht über die Anwendungen führt. Wir haben aber gleich am Anfang gesagt, dass es auch nicht hinreichend ist, eine Wissenschaftsdisziplin als Summe ihrer Forschungsresultate anzusehen. Die ersten zwei zentralen Fragen sind deshalb die folgenden:

„Wie entsteht eine Wissenschaft?“
„Was sind die Grundbausteine einer Wissenschaft?“

Jede Wissenschaft hat ihre eigene Sprache und somit ihre eigenen Begriffe (Fachwörter),

ohne die man keine Aussagen über die Objekte der Untersuchung formulieren kann. Die **Begriffsbildung** als die Bestimmung von Fachwörtern und deren Bedeutung ist somit zentral für alle Wissenschaften. Eine genaue und richtig interpretierbare Bedeutung eines wichtigen Fachbegriffs verursacht dann auch oft mehr Aufwand als die Herleitung hoch anerkannter Forschungsergebnisse. Nehmen wir uns ein paar Beispiele vor. Es dauerte 300 Jahre, bis man sich auf eine exakte und formale (mathematische) Definition des Begriffes „Wahrscheinlichkeit“ geeinigt hatte. Für eine allgemein akzeptable Definition der Ableitung brauchte man auch eine 200 Jahre lange Entwicklung. Mathematiker haben Tausende von Jahren gebraucht, bis sie das Unendliche als einen formalen Begriff festgelegt hatten. Wir benutzen in der Physik und auch umgangssprachlich sehr häufig den Begriff „Energie“. „Weiß der Kuckuck, was das ist!“ Die ganze Geschichte der Physik könnte man als eine nicht abgeschlossene Geschichte der Entwicklung unseres Verständnisses dieses Begriffes ansehen. Jetzt kann jemand die Hand heben und widersprechen: „Ich weiß, was Energie ist, das habe ich in der Schule gelernt.“ Und dann werde ich fragen: „Hast du die griechische Definition[1] der Energie als wirkende Kraft gelernt? Oder die Schulbuchdefinition der Energie als die Fähigkeit eines physikalischen Systems, Arbeit zu verrichten? Dann sag mir zuerst, was Kraft und was Arbeit ist.“ Und wenn man damit anfängt, stellt man fest, dass man sich im Kreise dreht[2], weil zur Definition von Kraft und Arbeit der Begriff Energie verwendet.

Ähnlich ist es mit dem Begriff des Lebens in der Biologie. Eine genaue Definition dieses Begriffes wäre für uns ein Instrument, mit dem man eindeutig zwischen toter und lebendiger Materie unterscheiden könnte. Eine solche Definition auf der physikalisch-chemischen Ebene existiert aber nicht.

Liebe Schülerinnen, liebe Schüler. Ich möchte euch keineswegs auf diese Weise in eurem Wissen verunsichern. Es ist keine Katastrophe, dass wir einige wichtige Begriffe nicht ganz genau spezifizieren können. In der Wissenschaft arbeitet man oft mit Definitionen, die einen benannten Begriff nur ungenau und annähernd bis zu einem gewissen Grad spezifizieren. Das gehört aber zum normalen Leben der Forscher. Sie müssen dann wissen, dass sie bei ihren Resultaten keine höhere Genauigkeit in der Interpretation erreichen können als die Genauigkeit der bereits vorliegenden Begriffsspezifikationen. Deswegen streben auch die Wissenschaftler ständig danach, ihr Wissen in Definitionen umzuwandeln, die die Bedeutung der zentralen Begriffe genauer approximieren (annähern). Fortschritte in dieser Richtung sind oft maßgebend für die Entwicklung der Wissenschaft. Ein wun-

[1] Griechisch „energeia“ bedeutet wirkende Kraft.

[2] Zum Beispiel versteht man in der Thermodynamik unter Arbeit die Energiedifferenzen, die nicht thermisch ausgetauscht werden.

derbares Beispiel des Fortschritts in der Begriffsbildung ist unser, sich im Laufe der Jahrtausende immer vertiefendes Verständnis des Begriffes „Materie“.

Um zu verstehen, was es bedeutet und wie schwer es ist, Begriffe genau zu definieren, betrachten wir ein konkretes Beispiel. Nehmen wir das Wort „Stuhl“. Der Stuhl ist kein abstraktes wissenschaftliches Objekt. Er ist ein gewöhnlicher Gegenstand und die meisten von uns wissen oder glauben zu wissen, was es ist. Jetzt versuchen wir, diesen Begriff durch eine Beschreibung zu definieren.

> ***Definieren*** *bedeutet, so genau zu beschreiben, dass jede und jeder, der noch nie einen Stuhl gesehen hat, anhand dieser Beschreibung für jeden Gegenstand eindeutig entscheiden kann, ob es ein Stuhl ist oder nicht. In der Definition dürfen nur Wörter (Begriffe) verwendet werden, deren Bedeutung schon vorher festgelegt wurde.*

Die erste Idee wäre vorauszusetzen, dass man schon weiß, was ein „Stuhlbein“ ist. In diesem Fall könnte man mit der Aussage anfangen, dass das Ding vier Beine hat. Aber Halt! Hat der Stuhl, auf dem Sie sitzen, nicht nur ein Bein und noch dazu ein merkwürdiges? Also lassen wir das lieber! Meine Aufgabe ist es nicht, euch zu quälen. Es geht nur darum zu verstehen, dass Begriffsbildung nicht nur eine wichtige, sondern auch eine sehr mühsame Arbeit ist.

Wir haben jetzt klar gemacht, dass Begriffsbildung ein zentrales Thema in der Wissenschaft ist. Auch das Entstehen der Informatik als Grundlagenwissenschaft verbindet man mit der Bildung eines Begriffes, nämlich dem „Algorithmus“. Bevor wir aber zu dieser Geschichte übergehen, müssen wir noch wissen, was Axiome in der Wissenschaft sind.

> ***Axiome*** *sind Grundbausteine der Wissenschaft. Es sind Tatsachen oder Begriffsspezifikationen, von deren Wahrhaftigkeit und Korrektheit wir fest überzeugt sind, obwohl es keine Möglichkeit gibt, ihre Korrektheit zu beweisen.*

Das klingt zunächst nicht nur merkwürdig, sondern geradezu verdächtig. Will man an der Zuverlässigkeit der wissenschaftlichen Aussagen zweifeln?

Versuchen wir zuerst das Ganze anhand eines Beispiels zu erläutern. Ein solches Axiom ist die Annahme, dass wir korrekt denken und somit unsere Art zu argumentieren zweifellos zuverlässig ist. Können wir beweisen, dass wir korrekt denken? Auf welche Weise? Durch

unsere Argumentation, die auf diesem Denken basiert? Unmöglich. Es bleibt uns also nichts anderes übrig, als unserer Denkweise zu vertrauen. Wenn dieses Axiom nicht stimmen sollte, dann haben wir Pech gehabt und das Gebäude der Wissenschaft bricht zusammen. Dieses Axiom ist nicht nur ein philosophisches. Es hat eine mathematische Form und weil die Mathematik die formale Sprache der Wissenschaften ist, kann man ohne sie nichts anfangen. Weil wir in der Informatik wie auch in anderen Wissenschaften korrekt argumentieren können müssen, widmen wir den nächsten Teil der Einführung in die korrekte Beweisführung.

Wir haben bisher nur über Grundbausteine gesprochen. Was kann man über die Steine sagen, die darauf gelegt werden? Die Forscher versuchen, die Wissenschaft so zu bauen, dass die Richtigkeit der Axiome (Grundbausteine) die Korrektheit des ganzen Baus garantiert. Das ist die bekannte Sachlichkeit und Zuverlässigkeit der Wissenschaft. Wenn die Axiome stimmen, dann stimmen auch alle Resultate und alle daraus abgeleiteten Kenntnisse. Wie der Entstehungsprozess einer Wissenschaft und ihrer Entwicklung genau aussieht, werden wir in dem Abschnitt über die Geschichte der Informatik lernen.

Zusammenfassung

Wissenschaftsdisziplinen entstehen hauptsächlich durch Begriffsbildung. In der Begriffsbildung wird gewissen Worten eine Bedeutung gegeben. Also versuchen wir, durch Begriffsbildung über konkrete und abstrakte Objekte so genau wie möglich zu sagen, was sie sind und was sie nicht sind. Genau wie bei der Entstehung einer natürlichen Sprache ermöglichen uns neue Begriffe, über Objekte und Tatsachen zu sprechen, über die wir vorher nicht sprechen konnten. Damit können wir auch Fragen stellen, die wir vorher nicht stellen konnten, und bereichern auch unsere Argumentationsfähigkeit. Begriffsbildung führt zur Herstellung des Fundaments des Wissenschaftsgebäudes. Das Gebäude versuchen wir so zu bauen, dass die Wahrhaftigkeit der Grundbausteine die Wahrhaftigkeit des ganzen Gebäudes der Wissenschaft garantiert. Die begriffsbildenden Definitionen in der Mathematik nennen wir Axiome. Prozesse der Begriffsbildung sind typischerweise länger und anspruchsvoller als die Prozesse der Erforschung unbekannter Tatsachen und Zusammenhänge. Die Begriffsbildung ist maßgeblich für die Formung der Wissenschaften. Deswegen kommen in den Definitionen der wissenschaftlichen Disziplinen keine Forschungsresultate vor, sondern Fragestellungen über die Grundbegriffe.

Die formale mathematische Definition des Begriffes „Algorithmus“ führte zur Gründung der Informatik. Die Informatik kann man keiner Klasse wissenschaftlicher Disziplinen eindeutig zuordnen. In der Grundlagenforschung ist sie mathematisch- und naturwissen-

schaftlicher Natur und untersucht die quantitativen Gesetze der Informationsverarbeitung. In mehreren Naturwissenschaften wurde sie, ähnlich wie die Mathematik, zur Forschungsmethode. In den meisten ihrer Anwendungen ist sie eine typische, produktorientierte Ingenieurwissenschaft. Diese große Breite macht das Studium der Informatik einerseits schwierig und andererseits attraktiv, mit einer vielversprechenden Perspektive, weil sie in einem Fach auf eine funktionsfähige Weise die mathematisch-naturwissenschaftliche Denkweise mit jener der technischen Ingenieurdisziplinen verbindet.

Kontrollfragen

1. Was sind die Grundbausteine einer Wissenschaft?

2. Was sind die Grundbegriffe der Physik? Was ist der Grundbegriff der Biologie? Was sind die Grundbegriffe der Chemie? Bevor du die Fragen beantwortest, lies die Definitionen dieser Fächer.

3. Was bedeutet es, einen Begriff zu definieren?

4. Was sind Axiome?

5. Wie hängt die Wahrhaftigkeit der Axiome (der Grundbausteine) mit der Wahrhaftigkeit der ganzen Wissenschaft zusammen?

Lektion 2

Korrekte Argumentation

Hinweis für die Lehrperson Die Bearbeitung dieses Abschnittes ist keine Voraussetzung für die Bearbeitung der anderen Teile dieses Moduls. Das hier vermittelte Wissen ist eine der Voraussetzungen für das Studium einiger anderer Module, wie z. B. Berechenbarkeit und Automatenentwurf. Ausführlicher wird das Thema „Beweisen“ in dem Modul über Logik nochmals behandelt. Eine ausführliche Bearbeitung dieses Abschnittes empfehlen wir nur für Klassen in den letzten zwei Jahren des Gymnasialbesuchs. Wenn man sich auf die Argumentation mit der Wahrheitstabelle beschränkt und auf schwierigere formale Beweise über mathematische Objekte verzichtet, kann man Klassen im neunten Schuljahr diesen Teil gut zumuten.

Das Ziel dieses Abschnittes ist es zu lernen, was es bedeutet, korrekt zu argumentieren. Das hängt natürlich eng mit unseren Axiomen zusammen, die besagen, dass unsere Denkweise korrekt ist. Was ist aber unsere Denkweise? Um es zu verstehen, müssen wir den Begriff der Folgerung (der Implikation) und die Regel, wie man mit Hilfe der Implikation zu neuen Kenntnissen kommen kann, einführen.

Erklären wir also genauer, was unser Festhalten an diesen Axiomen bedeutet. Wenn

*eine Tatsache B eine **Folgerung** aus einer Tatsache A ist,*

muss immer Folgendes gelten:

Wenn A wahr ist (wenn A gilt), dann ist auch B wahr (dann gilt B).

In anderen Worten,

J. Hromkovič, *Lehrbuch Informatik*, DOI 10.1007/978-3-8348-9692-6_17,

die Unwahrheit kann nicht die Folgerung aus einer Wahrheit sein.

In der Mathematik benutzt man die Bezeichnung

$A \Rightarrow B$

für die Tatsache „*B* **ist eine Folgerung aus** *A*“. Man sagt auch „*A* **impliziert** *B*“. Dann sagt unser Axiom: Wenn

$A \Rightarrow B$ und A gelten,

dann

gilt auch B.

Es lohnt sich zu bemerken, dass wir zulassen, dass eine Unwahrheit eine Wahrheit impliziert. Wir erlauben nur nicht, dass die Wahrheit eine Unwahrheit als Schlussfolgerung hat. Um dies besser zu verstehen, präsentieren wir das folgende Beispiel.

Beispiel 2.1 Betrachten wir zwei Aussagen *A* und *B*.

A ist „*Es regnet.*“

und

B ist „*Die Wiese ist nass.*“.

Nehmen wir an, unsere Wiese ist unter freiem Himmel (also nicht bedeckt oder überdacht). Somit können wir annehmen, dass die Behauptung

„Wenn es regnet, ist die Wiese nass.“

also

$A \Rightarrow B$

wahr ist. Nach unser Interpretation des Fachwortes „Folgerung“ muss die Wiese nass sein (also muss *B* gelten), wenn es regnet (wenn *A* gilt). Schauen wir uns das noch genauer an.

„*A* gilt“ bedeutet: „*Es regnet.*“.
„*A* gilt nicht“ bedeutet: „*Es regnet nicht.*“.
„*B* gilt“ bedeutet: „*Die Wiese ist nass.*“.
„*B* gilt nicht“ bedeutet: „*Die Wiese ist trocken.*“.

Es gibt die folgenden vier Situationen, die Gültigkeit von *A* und *B* betreffend.

S_1: Es regnet und die Wiese ist nass.
S_2: Es regnet und die Wiese ist trocken.
S_3: Es regnet nicht und die Wiese ist nass.
S_4: Es regnet nicht und die Wiese ist trocken.

Diese Möglichkeiten kann man in einer sogenannten Wahrheitstabelle darstellen (siehe Tab. 2.1). Die Zeilen der Tabelle entsprechen den möglichen Situationen. In den Spalten ist die Gültigkeit oder die Ungültigkeit der einzelnen Aussagen (Ereignisse) eingetragen.

	A	B
S_1	gilt	gilt
S_2	gilt	gilt nicht
S_3	gilt nicht	gilt
S_4	gilt nicht	gilt nicht

Tabelle 2.1 Wahrheitstabelle

Die Mathematiker lieben es, alles so kurz wie möglich zu schreiben, und nehmen dabei leider auch gerne das Risiko in Kauf, dass die Verständlichkeit ihrer Texte für Nichtmathematiker darunter leidet. Sie bezeichnen die Gültigkeit oder die Wahrheit mit 1 und die Unwahrheit (Ungültigkeit) mit 0. Mit dieser Bezeichnung hat unsere Wahrheitstabelle die folgende verkürzte Darstellung (siehe Tab. 2.2).

	A	B
S_1	1	1
S_2	1	0
S_3	0	1
S_4	0	0

Tabelle 2.2 Wahrheitstabelle (Kurzschreibweise)

Es ist wichtig zu beobachten, dass die Gültigkeit von $A \Rightarrow B$ nur die Möglichkeit der Situation S_2 in der zweiten Zeile (A gilt, B gilt nicht) ausschließt. Analysieren wir dies genauer.

Die erste Zeile entspricht der Situation S_1, wenn A und B gelten. Das heißt, es regnet und die Wiese ist deswegen nass. Offenbar entspricht dies $A \Rightarrow B$ und damit unserer Erwartung.
Die zweite Zeile mit „A gilt“ und „B gilt nicht“ entspricht der Situation S_2, wenn es regnet und die Wiese trocken ist. Diese Situation ist unmöglich und widerspricht der Gültigkeit unserer Behauptung $A \Rightarrow B$, weil unser Verständnis von „$A \Rightarrow B$“ bedeutet, dass aus der Gültigkeit von A („es regnet“) die Gültigkeit von B („die Wiese ist nass“) gefolgert wird.
Die dritte Zeile beschreibt die Situation S_3 wenn es nicht regnet (A gilt nicht) und die Wiese nass ist (B gilt). Diese Situation ist möglich und die Behauptung $A \Rightarrow B$ schließt sie nicht aus. Es regnet zwar nicht, aber die Wiese darf trotzdem nass sein. Vielleicht hat es vorher geregnet, jemand hat die Wiese gegossen oder morgens liegt nach einer hellen kalten Nacht Tau auf dem Gras.
Die letzte Zeile (A und B gelten beide nicht) entspricht der Situation, wenn es nicht regnet und die Wiese trocken ist. Diese Situation ist natürlich möglich und steht in keinem Konflikt zu der Aussage $A \Rightarrow B$.

Fassen wir das Gelernte kurz zusammen: Wenn $A \Rightarrow B$ gültig ist und A gilt („*es regnet*“), dann muss auch B gelten („*die Wiese ist nass*“). Wenn A nicht gilt („*es regnet nicht*“) gibt die Gültigkeit von „$A \Rightarrow B$“ keine Anforderungen an B und somit kann B gelten oder nicht gelten (Zeilen 3 und 4 in der Wahrheitstabelle). □

Die einzige ausgeschlossene Möglichkeit bei der Gültigkeit von $A \Rightarrow B$ ist „A gilt und B gilt nicht“. Wenn man also eine Wahrheitstabelle für zwei Behauptungen A und B hat, in der alle Situationen betreffend der Gültigkeit von A und B bis auf die Situation „A gilt und B gilt nicht“ möglich sind, dann gilt $A \Rightarrow B$. Die Wahrheitstabelle (siehe Tab. 2.3) ist

A	B	$A \Rightarrow B$
gilt	gilt	möglich (gilt)
gilt	gilt nicht	ausgeschlossen (gilt nicht)
gilt nicht	gilt	möglich (gilt)
gilt nicht	gilt nicht	möglich (gilt)

Tabelle 2.3 Definition der Implikation

aus mathematischer Sicht die Definition der Implikation. Die Definition des Begriffes der Implikation akzeptieren wir, weil sie unserer intuitiven Vorstellung über die Bedeutung der Folgerung und unserer ganzen, bisherigen Erfahrung entspricht.

Im Allgemeinen gibt es eine einfache Regel, um die Gültigkeit einer Implikation $A \Rightarrow B$ zu überprüfen.

> *Wenn in allen möglichen Situationen, in denen A gilt, auch B gilt, wissen wir, dass $A \Rightarrow B$ gilt.*

Aufgabe 2.1 Betrachten wir die folgenden Aussagen A und B. A bedeutet „*Es ist Winter*“ und B bedeutet „*Die Braunbären schlafen*“. Die Implikation $A \Rightarrow B$ bedeutet:

> *„Wenn es Winter ist, dann schlafen die Braunbären.“*

Nehmen wir an, diese Folgerung $A \Rightarrow B$ gilt. Stelle die Wahrheitstabelle bezüglich der Gültigkeit von A und B auf und erkläre, welche Situationen möglich und welche ausgeschlossen sind.

Jetzt haben wir die Bedeutung der Folgerung (Implikation) verstanden. Nun stellt sich die Frage: *Was hat die Folgerung mit einer korrekten Argumentation zu tun? Warum ist dieser Begriff der Schlüssel zur fehlerlosen Begründung (zu einem Beweis)?* Wir benutzen den Begriff der Implikation zur Entwicklung von sogenannten direkten Beweisen (direkte Argumentation) und indirekten Beweisen (indirekte Argumentation). Um unsere Begründungen für den Rest des Buches genau nachvollziehen zu können, stellen wir diese grundsätzlichen Beweismethoden im Folgenden vor.

Betrachten wir unsere Aussagen A („Es regnet“) und B („Die Wiese ist nass“) aus Beispiel 2.1. Nehmen wir noch eine dritte Aussage C („Die Salamander freuen sich“) hinzu. Wir halten $A \Rightarrow B$ für gültig und nehmen an, dass auch

> $B \Rightarrow C$ („Wenn die Wiese nass ist, freuen sich die Salamander“)

gilt. Was können wir daraus schließen? Betrachten wir die Wahrheitstabelle für alle acht Situationen bezüglich der Gültigkeit von A, B und C (siehe Tab. 2.4). Weil $A \Rightarrow B$ gilt, sind die Situationen S_3 und S_4 ausgeschlossen. Analog sind wegen $B \Rightarrow C$ die Situationen S_2 und S_6 ausgeschlossen. Betrachten wir jetzt diese Tabelle nur aus Sicht von A und C. Wir sehen, dass folgende Situationen möglich sind:

	A	B	C	$A \Rightarrow B$	$B \Rightarrow C$
S_1	gilt	gilt	gilt		
S_2	gilt	gilt	gilt nicht		ausgeschlossen
S_3	gilt	gilt nicht	gilt	ausgeschlossen	
S_4	gilt	gilt nicht	gilt nicht	ausgeschlossen	
S_5	gilt nicht	gilt	gilt		
S_6	gilt nicht	gilt	gilt nicht		ausgeschlossen
S_7	gilt nicht	gilt nicht	gilt		
S_8	gilt nicht	gilt nicht	gilt nicht		

Tabelle 2.4 Wahrheitstabelle für $A \Rightarrow B$, $B \Rightarrow C$

(i) A und C gelten beide (S_1)

(ii) A und C gelten beide nicht (S_8)

(iii) A gilt nicht und C gilt (S_5, S_7)

Die Situationen S_2 und S_4 in denen A gilt und C nicht gilt, sind dank $A \Rightarrow B$ und $B \Rightarrow C$ ausgeschlossen. Damit erhalten wir, dass

$A \Rightarrow C$ („*Wenn es regnet, freuen sich die Salamander*“)

gilt. Dies entspricht genau unserer Erwartung. Wenn es regnet, muss die Wiese nass sein ($A \Rightarrow B$). Wenn die Wiese nass ist, müssen sich die Salamander freuen ($B \Rightarrow C$). Also verursacht der Regen, indem er die Wiese durchnässt, die Freude der Salamander.

Die Überlegung

„Wenn $A \Rightarrow B$ und $B \Rightarrow C$ gelten, dann gilt auch $A \Rightarrow C$.“

nennen wir direkte Argumentation. **Direkte Beweise** kann man aus beliebig vielen Folgerungen zusammenstellen. Zum Beispiel erlaubt uns die Gültigkeit der Implikationen

$$A_1 \Rightarrow A_2, A_2 \Rightarrow A_3, A_3 \Rightarrow A_4, \ldots, A_{k-1} \Rightarrow A_k$$

zu schließen, dass

$$A_1 \Rightarrow A_k$$

auch gelten muss. Damit sind direkte Beweise einfach Folgen korrekter Folgerungen. In der Schulmathematik führen wir Tausende von direkten Beweisen, um gewisse Aussagen zu belegen. Leider machen uns die Mathematiklehrer nicht immer hinreichend darauf aufmerksam, und deswegen zeigen wir jetzt ein kleines Beispiel aus dem Mathematikunterricht.

Beispiel 2.2 Wir haben die lineare Gleichung $3x - 8 = 4$ gegeben und wollen beweisen, dass

$x = 4$ die einzige Lösung der Gleichung $3x - 8 = 4$ ist.

Mit anderen Worten: Wir wollen die Implikation

„Wenn $3x - 8 = 4$ gilt, dann gilt $x = 4$“

beweisen. Sei A die Behauptung „$3x - 8 = 4$ gilt“ und sei Z die Zielbehauptung „$x = 4$ gilt“. Um $A \Rightarrow Z$ zu beweisen, brauchen wir eine Reihe von Folgerungen, die mit A anfangen, mit Z enden und zweifellos korrekt sind.

Wir wissen, dass eine Gleichung erhalten bleibt[1], wenn man beide Seiten um die gleiche Zahl erhöht. Addieren wir zu beiden Seiten der Gleichung $3x - 8 = 4$ die Zahl 8, dann erhalten wir

$$3x - 8 + 8 = 4 + 8$$

und somit

$$3x = 12$$

Sei B die Behauptung, dass $3x = 12$ gilt. Wir haben gerade die Gültigkeit der Folgerung „$A \Rightarrow B$“ („Wenn $3x - 8 = 4$ gilt, dann gilt auch $3x = 12$“) begründet.

Somit haben wir schon die erste Folgerung. Weiter wissen wir, dass eine Gleichung gültig bleibt, wenn beide Seiten durch die gleiche positive Zahl geteilt werden. Teilen wir also beide Seiten der Gleichung $3x = 12$ durch 3 und erhalten

$$\frac{3x}{3} = \frac{12}{3}$$

[1] Genauer gesagt: Die Lösungen einer Gleichung ändern sich nicht, wenn man beide Seiten der Gleichung um die gleiche Zahl erhöht.

und damit

$$x = 4.$$

Somit haben wir die Gültigkeit der Folgerung $B \Rightarrow Z$ („Wenn $3x = 12$ gilt, dann gilt auch $x = 4$“) bewiesen.

Die Gültigkeit der Folgerungen $A \Rightarrow B$ und $B \Rightarrow Z$ erlaubt uns, die Gültigkeit der Folgerung $A \Rightarrow Z$ zu behaupten. Wenn $3x - 8 = 4$ gilt, muss somit $x = 4$ gelten. Also ist $x = 4$ die einzige Lösung der Gleichung $3x - 8 = 4$. □

Aufgabe 2.2 Zeige durch eine Folge von Folgerungen, dass $x = 1$ die einzige Lösung der Gleichung $7x - 3 = 2x + 2$ ist. Dabei darf man als bekannt voraussetzen, dass das Addieren einer Zahl zu beiden Seiten und das Multiplizieren beider Seiten mit einer gleichen Zahl die Gültigkeit der Gleichung bewahrt.

Aufgabe 2.3 Betrachten wir die folgende Wahrheitstabelle für die drei Aussagen A, B und C (siehe Tab. 2.5).

	A	B	C	
S_1	1	1	1	
S_2	1	1	0	
S_3	1	0	1	ausgeschlossen
S_4	1	0	0	ausgeschlossen
S_5	0	1	1	ausgeschlossen
S_6	0	1	0	ausgeschlossen
S_7	0	0	1	ausgeschlossen
S_8	0	0	0	

Tabelle 2.5 Wahrheitstabelle für A, B und C aus Aufgabe 2.3

Wir sehen, dass nur drei der Situationen (S_1, S_2 und S_8) möglich und alle anderen ausgeschlossen sind. Welche Implikationen gelten? Zum Beispiel gilt $C \Rightarrow A$, denn wenn in einer der möglichen Situationen C gilt, dann gilt A auch. Die Implikation $B \Rightarrow C$ gilt nicht, weil in der möglichen Situation S_2 die Behauptung B gilt und die Behauptung C gilt nicht. Welche anderen Implikationen gelten noch?

Aufgabe 2.4 Betrachten wir die folgende Wahrheitstabelle für die vier Aussagen X, Y, U und V in Tab. 2.6. Bestimme die Gültigkeit aller Implikationen $A \Rightarrow B$ für $A, B \in \{X, Y, U, V\}$.

	X	Y	U	V	
S_1	1	1	1	1	
S_2	1	1	1	0	ausgeschlossen
S_3	1	1	0	1	ausgeschlossen
S_4	1	1	0	0	
S_5	1	0	1	1	
S_6	1	0	1	0	ausgeschlossen
S_7	1	0	0	1	
S_8	1	0	0	0	ausgeschlossen
S_9	0	1	1	1	ausgeschlossen
S_{10}	0	1	1	0	ausgeschlossen
S_{11}	0	1	0	1	
S_{12}	0	1	0	0	
S_{13}	0	0	1	1	
S_{14}	0	0	1	0	ausgeschlossen
S_{15}	0	0	0	1	ausgeschlossen
S_{16}	0	0	0	0	

Tabelle 2.6 Wahrheitstabelle für X, Y, U und V in Aufgabe 2.4

Das Schema der direkten Argumentation kann man auf zwei unterschiedliche Weisen betrachten. Bereits vorgestellt wurde jene mit dem Ziel, den Beweis der Gültigkeit einer Implikation $A \Rightarrow Z$ durch die Herstellung einer Folge von Implikationen

$$A \Rightarrow B_1, B_1 \Rightarrow B_2, \ldots, B_{k-1} \Rightarrow B_k, B_k \Rightarrow Z$$

zu führen.

Das andere Schema kann wie folgt dargestellt werden:

Ausgangssituation: A gilt (Wir wissen, dass eine bestimmte Behauptung gültig ist.)
Ziel: Zu beweisen, dass eine bestimmte Behauptung Z gilt.

Methode

1. Finde eine Folge von Folgerungen, die mit A anfängt und mit Z endet. Damit beweist man die Gültigkeit der Implikation $A \Rightarrow Z$.

2. Aus der Gültigkeit von A und $A \Rightarrow Z$ schließen wir die Gültigkeit von Z.

Das vorgestellte Schema ist noch nicht ganz vollständig, weil es eine wichtige Tatsache verbirgt. Woher kommen die Implikationen, die wir in dem ersten Teil der Methode verwenden, um $A \Rightarrow Z$ zu beweisen? Wenn wir genau vorgehen wollen, müssen wir unter der Ausgangssituation alle Aussagen und alle Implikationen auflisten, deren Gültigkeit wir schon bewiesen haben. Zum Beispiel haben wir im Beweis der Implikation „$3x-8=4$“ $\Rightarrow$ „$x=4$“ die Tatsachen verwendet, dass Umformungen von Gleichungen durch Addieren der gleichen Zahl zu beiden Seiten oder durch die Division beider Seiten durch eine Zahl $d \neq 0$ die Lösungsmenge und somit die Bedeutung dieser Gleichung nicht ändern. Diese bewiesenen Aussagen, auch Sätze genannt, formen dann unsere bisherige **Theorie** und wir nutzen sie, um die Gültigkeit neuer Tatsachen wie $A \Rightarrow Z$ und Z zu beweisen. Nach der Durchführung eines Beweises dürfen wir die neu bewiesenen Behauptungen zur Theorie hinzufügen und bei ihrer Erweiterung für das Beweisen der Gültigkeit weiterer Tatsachen verwenden. Dies ist auch die Vorgehensweise in den mathematischen Theorien und somit auch der gesamten Mathematik. Am Anfang stehen nur die Axiome, also Definitionen zur Verfügung. Davon leitet man nach und nach die mathematischen Sätze ab.

Illustrieren wir diesen Vorgang mit einem zahlentheoretischen Beispiel.

Hinweis für die Lehrperson Der folgende Teil bis zum Thema der indirekten Argumentation ist nur für die letzten zwei Jahrgänge des Gymnasialunterrichts mit einem mathematisch-naturwissenschaflichen Schwerpunkt geeignet.

Beispiel 2.3 Sei $a \operatorname{div} b$ das Resultat der ganzzahligen Teilung von a durch b. Zum Beispiel: $42 \operatorname{div} 5 = 8$, weil 8 die größte Zahl x mit der Eigenschaft $5 \cdot x \leq 42$ ist. Sei $a \bmod b$ die Bezeichnung für den Rest der ganzzahligen Teilung von a durch b. Somit ist z.B. $42 \bmod 5 = 2$, weil $42 = 8 \cdot 5 + 2$. Unsere Aufgabe ist, die Gültigkeit folgender Aussage zu beweisen:

> *Wenn* $a \bmod p = b \bmod p$ *für drei ganze positive Zahlen a, b und p (also wenn die Reste der ganzzahligen Divisionen* $a \operatorname{div} p$ *und* $b \operatorname{div} p$ *gleich sind), dann teilt p die Differenz* $a - b$.

Unsere Zielsetzung ist also, eine Implikation

$$A \Rightarrow Z$$

zu beweisen, wobei

A bedeutet „$a \bmod p = b \bmod p$“

und

Z bedeutet „p teilt $a - b$“.

Bevor wir überhaupt mit einem direkten Beweis anfangen dürfen, müssen wir die Bedeutung unserer Anfangssituation formal beschreiben.

Die Definition von „p **teilt** a“ für zwei ganze Zahlen p und a bedeutet, dass sich a als

$$a = k \cdot p$$

für eine ganze Zahl k schreiben lässt. Weil es sich um eine Definition handelt, gelten beide Implikationen

„p teilt a“ $\Rightarrow$ „$a = k \cdot p$ für ein $k \in \mathbb{Z}$“
„$a = k \cdot p$ für ein $k \in \mathbb{Z}$“ $\Rightarrow$ „p teilt a“

und somit sprechen wir von der Äquivalenz zwischen diesen beiden Aussagen. Die Bezeichnung „$\Leftrightarrow$“ steht für die Äquivalenz, also für die Gültigkeit der Implikationen in beide Richtungen. Damit sieht unsere Definition der Teilbarkeit wie folgt aus:

„p teilt a“ $\Leftrightarrow$ „$a = k \cdot p$ für ein $k \in \mathbb{Z}$“

Aufgabe 2.5 Für welche Aussagepaare (X, Y) für $X, Y \in \{A, B, C\}$ aus der Tabelle 2.5 gilt die Äquivalenz $X \Leftrightarrow Y$?

Die weitere Tatsache, deren Zugehörigkeit zur bisherigen Theorie wir voraussetzen, ist die folgende Behauptung:

Für zwei beliebige ganze positive Zahlen a und p kann man a eindeutig als

$$a = k \cdot p + r$$

für ein $k \in \mathbb{Z}$ und ein $r \in \mathbb{Z}$, $r < p$ darstellen. Die Zahl r nennen wir den Rest der ganzzahligen Division $a \operatorname{div} p$ und schreiben $r = a \bmod p$.
Die Zahl k nennen wir das Resultat der ganzzahligen Division $a \operatorname{div} p$ und verwenden die Bezeichnung $k = a \operatorname{div} p$.

Damit ist zum Beispiel $23 = 4 \cdot 5 + 3$ und somit gilt $r = 3 = 23 \bmod 5$ und $4 = 23 \operatorname{div} 5$. Damit können wir für beliebige a und p aus $\mathbb{Z}^+$ schreiben:

$$a = (a \operatorname{div} p) \cdot p + a \bmod p.$$

Weiter setzen wir noch die Gültigkeit des Distributivgesetzes voraus, also

$$c \cdot d + c \cdot h = c \cdot (d + h)$$

für alle ganzen Zahlen c, d und h.

Jetzt sind wir so weit, dass wir den Beweis führen können. Bei allen verwendeten Implikationen machen wir in geschweiften Klammern klar, um welche Implikation aus der schon bekannten Theorie es sich handelt.

„$a \bmod p = b \bmod p = r$ für $a, b, p \in \mathbb{Z}$“

$\Rightarrow$ „$a = (a \operatorname{div} p) \cdot p + r$ und $b = (b \operatorname{div} p) \cdot p + r$“
{Eindeutigkeit der Darstellung von a und b bezüglich des Teilers p}

$\Rightarrow$ „$a - b = (a \operatorname{div} p) \cdot p - (b \operatorname{div} p) \cdot p + r - r =$
$p \cdot [(a \operatorname{div} p) - (b \operatorname{div} p)]$“
{Arithmetik und das Distributivgesetz}

$\Rightarrow$ „p teilt $a - b$“
{Definition der Teilbarkeit, denn wenn $a \operatorname{div} p$ und $b \operatorname{div} p$ ganze Zahlen sind, muss auch ihre Differenz eine ganze Zahl sein.}

In dieser direkten Beweisführung haben wir schon darauf verzichtet, die einzelnen Aussagen durch große Buchstaben zu benennen. Wir haben die Aussagen aber apostrophiert Dadurch ist die Strukturierung unseres Beweises übersichtlich. □

Aufgabe 2.6 Beweise mittels direkter Argumentation die folgende Aussage:

> „Für alle ganzen Zahlen a, b, c und r impliziert $a \cdot b + r = a \cdot c + r$, dass $b = c$ gilt."

Zur Verfügung stehen alle Gesetze der Arithmetik und Gleichungsumformungen.

Aufgabe 2.7 Beweise mittels direkter Argumentation die folgende Aussage:

> „Wenn $a \bmod p = b \bmod p$ für $a, b, p \in \mathbb{Z}^+$ gilt, dann gibt es eine ganze Zahl d, so dass $a = b + d \cdot p$ gilt."

Zur Verfügung stehen dieselben Tatsachen (dieselbe Theorien) wie in Beispiel 2.3.

Aufgabe 2.8 Beweise mittels direkter Argumentation die folgende Aussage:

> „Wenn p die beiden Zahlen a und b teilt, dann teilt p auch $a + b$".

Aufgabe 2.9 Beweise mittels direkter Argumentation die folgende Aussage für beliebige $a, b \in \mathbb{Z}$ und $d \in \mathbb{Z}^+$:

> „Wenn d die Zahl a teilt und d die Zahl b teilt, dann teilt d auch $ax + by$ für beliebige $x, y \in \mathbb{Z}$".

Zur Verfügung stehen dieselben Behauptungen wie in Beispiel 2.3.

Aufgabe 2.10 Beweise direkt die folgende Aussage für beliebige a, $b \in \mathbb{Z}$:

> „Wenn „a teilt b" und „b teilt a" gelten, dann gilt $a = b$".

Zur Verfügung stehen alle bisher als gültig betrachteten Aussagen und die Tatsache, dass $a \leq b$ und $b \leq a$ die Gleichung $a = b$ implizieren.

Im Folgenden sind die Zahlen a und b immer ganze Zahlen. Wenn die Zahl a die Zahl b teilt, dann sagen wir, dass a ein **Teiler** von b ist. Mit

$$\text{Teiler}_b = \{a \in \mathbb{N} \mid a \text{ teilt } b\}$$

bezeichnen wir die Menge aller Teiler von b. Zum Beispiel:

$$\text{Teiler}_{60} = \{1,2,3,4,5,6,10,12,15,20,30,60\}.$$

Eine Zahl p nennen wir **Primzahl**, wenn $\text{Teiler}_p = \{1,p\}$, d.h. wenn p nur durch 1 und sich selbst teilbar ist. Für zwei positive ganze Zahlen a und b definieren wir

$$\text{Teiler}_{a,b} = \text{Teiler}_a \cap \text{Teiler}_b$$

als die Menge der gemeinsamen Teiler von a und b. Damit ist d ein **gemeinsamer Teiler** von a und b genau dann, wenn d die Zahl a und die Zahl b teilt. Zum Beispiel:

$$\text{Teiler}_{24,60} = \{1,2,3,4,6,12\}.$$

Der **größte gemeinsame Teiler von a und b** ist

$$\text{GGT}(a,b) = \text{maximum } \{c \mid c \in \text{Teiler}_{a,b}\},$$

d. h. die größte natürliche Zahl, die a und b teilt.

Aufgabe 2.11 Bestimme den GGT für folgende Zahlenpaare (a,b):

a) $a = 375$, $b = 225$

b) $a = 32$, $b = 264$

c) $a = 1024$, $b = 725$

d) $a = 162$, $b = 125$

Wenn man $\text{GGT}(a,b)$ für zwei große Zahlen bestimmen soll, würde es extrem aufwändig werden, die Mengen Teiler_a und Teiler_b zu bestimmen und dann in der Schnittmenge der beiden Mengen nach dem Maximum zu suchen.

Aufgabe 2.12 Erinnerst du dich, wie man $\mathrm{GGT}(a,b)$ aus der Faktorisierung (Primfaktorzerlegung) der Zahlen a und b bestimmen kann? Erkläre, wie es geht. Wende diese Methode an, um $\mathrm{GGT}(a,b)$ für die Zahlen a und b aus Aufgabe 2.11 zu bestimmen.

Aufgabe 2.13 Begründe mit eigenen Worten, warum „$\mathrm{GGT}(ka,a) = a$“ für beliebige positive ganze Zahlen k und a gilt.

In Aufgabe 2.12 haben wir uns daran erinnert, dass wir, statt die Mengen Teiler_a und Teiler_b zu bestimmen, das $\mathrm{GGT}(a,b)$ mit Hilfe der Primfaktorzerlegung der Zahlen a und b ermitteln können. Leider ist auch hier der Aufwand für große Zahlen sehr hoch. Ungefähr 200 v. Chr. hat man in China eine effizientere Methode zur Berechnung des größten gemeinsamen Teilers gefunden.[2] Die Methode basiert auf folgender Behauptung:

> $\mathrm{GGT}(a,b) = \mathrm{GGT}(a-b,b)$ *für alle positiven ganzen Zahlen a und b mit* $a > b$.

Bevor wir uns dem Beweis dieser Aussage widmen, beobachten wir, wie schnell wir mit dieser Methode den größten gemeinsamen Teiler bestimmen können.

$$\begin{aligned}
\mathrm{GGT}(7854,40664) &= \mathrm{GGT}(7854,32810)\\
&\quad \{\text{weil } 32810 = 40664 - 7854\}\\
&= \mathrm{GGT}(7854,24956)\\
&\quad \{\text{weil } 24956 = 32810 - 7854\}\\
&= \mathrm{GGT}(7854,17102)\\
&= \mathrm{GGT}(7854,9248)\\
&= \mathrm{GGT}(7854,1394)\\
&= \mathrm{GGT}(6460,1394)\\
&= \mathrm{GGT}(5066,1394)\\
&= \mathrm{GGT}(3672,1394)\\
&= \mathrm{GGT}(2278,1394)\\
&= \mathrm{GGT}(884,1394)\\
&\vdots
\end{aligned}$$

[2] Die Methode ist in der chinesischen Sammlung „Mathematik in 9 Büchern“ zu finden.

$$
\begin{aligned}
&\vdots \\
&= \mathrm{GGT}(884,510) \\
&= \mathrm{GGT}(374,510) \\
&= \mathrm{GGT}(374,136) \\
&= \mathrm{GGT}(238,136) \\
&= \mathrm{GGT}(102,136) \\
&= \mathrm{GGT}(102,34) \\
&= \mathrm{GGT}(68,34) \\
&= \mathrm{GGT}(34,34) = 34 \\
&\{\text{weil } \mathrm{GGT}(a,a) = a\}
\end{aligned}
$$

Aufgabe 2.14 Verwende diese chinesische Methode, um $\mathrm{GGT}(a,b)$ für folgende Zahlenpaare a und b zu berechnen:

a) $a = 162$, $b = 125$

b) $a = 109956$, $b = 98175$

c) $a = 990$, $b = 2160$

Um sicher zu sein, dass die chinesische Methode für alle Zahlen a und b erfolgreich $\mathrm{GGT}(a,b)$ ermittelt, müssen wir die Behauptung $\mathrm{GGT}(a,b) = \mathrm{GGT}(a-b,b)$ für alle a, b mit $a > b$ beweisen. Um nicht zu viel auf einmal machen zu müssen, ziehen wir oft vor, den Beweis einer Gleichung in zwei Beweise von entsprechenden Ungleichungen $\mathrm{GGT}(a,b) \leq \mathrm{GGT}(a-b,b)$ und $\mathrm{GGT}(a,b) \geq \mathrm{GGT}(a-b,b)$ zu zerlegen.

Beweisen wir zuerst $\mathrm{GGT}(a,b) \leq \mathrm{GGT}(a-b,b)$. Zur Verfügung stehen uns alle bisher bewiesenen Behauptungen aus dem Text und aus den Übungen. Unseren direkten Beweis fangen wir mit der Definition des GGT an.

„$d = \mathrm{GGT}(a,b)$ für zwei positive ganze Zahlen a,b mit $a > b$“

$\Rightarrow$ „$a > b$, d teilt a, d teilt b und d ist die größte positive ganze Zahl mit dieser Eigenschaft“
{Nach der Definition von $\mathrm{GGT}(a,b)$}

$\Rightarrow$ „d teilt $a-b$, d teilt a, d teilt b“
{Im Beispiel 2.3 wurde bewiesen, dass jede Zahl p, die a und b teilt, auch $a-b$ teilen muss.}

$\Rightarrow$ „$d \in \text{Teiler}_{a-b,b}$“
{weil d die beiden Zahlen $a-b$ und b teilt}

$\Rightarrow$ „$d = \text{GGT}(a,b) \leq \text{GGT}(a-b,b)$“
{Weil $d \in \text{Teiler}_{a-b,b}$ und $\text{GGT}(a-b,b)$ die größte Zahl aus $\text{Teiler}_{a-b,b}$ ist}

Beweisen wir jetzt $\text{GGT}(a-b,b) \leq \text{GGT}(a,b)$.

„$m = \text{GGT}(a-b,b)$ für positive ganze Zahlen a, b mit $a > b$“

$\Rightarrow$ „m teilt $a-b$ und b, $a > b$“
{Nach der Definition von $\text{GGT}(a-b,b)$}

$\Rightarrow$ „m teilt $a = (a-b)+b$ und b, $a > b$“
{Nach der Behauptung aus Aufgabe 2.8, wenn m zwei Zahlen $a-b$ und b teilt, dann muss m auch ihre Summe $a-b+b=a$ teilen.}

$\Rightarrow$ „$m \in \text{Teiler}_{a,b}$“

$\Rightarrow$ „$m = \text{GGT}(a-b,b) \leq \text{GGT}(a,b)$“
{Weil $m \in \text{Teiler}_{a,b}$ und $\text{GGT}(a,b)$ die größte Zahl aus $\text{Teiler}_{a,b}$ ist}

Unsere Beweisführung können wir damit schließen, dass die bewiesenen Ungleichungen $\text{GGT}(a-b,b) \leq \text{GGT}(a,b)$ und $\text{GGT}(a,b) \leq \text{GGT}(a-b,b)$ gemeinsam die Gleichung $\text{GGT}(a-b,b) = \text{GGT}(a,b)$ implizieren.

Aufgabe 2.15 Beweise für beliebige ganze Zahlen a und b, dass die folgende Gleichung gilt:

$$\text{GGT}(a,b) = \text{GGT}(a+b,b)$$

Wenn wir uns die chinesische Methode anschauen, stellen wir fest, dass sie für $a = m \cdot b + r$ wie folgt funktioniert ist:

$$\begin{aligned} \mathrm{GGT}(a,b) &= \mathrm{GGT}(a-b,b) \\ &= \mathrm{GGT}(a-2b,b) \\ &\vdots \\ &= \mathrm{GGT}(a-mb,b) \\ &= \mathrm{GGT}(r,b) \\ &\{\text{weil } r = a-mb\} \end{aligned}$$

Die Zahl $r = a \bmod b$ ist der Rest der ganzzahligen Division $a \operatorname{div} b$. Das deutet darauf hin, dass für $a > b$ „$\mathrm{GGT}(a,b) = \mathrm{GGT}(a \bmod b, b)$“ gilt, weil wir b von a so viele Male abziehen, bis der Rest $r < b$ übrig bleibt. Die Griechen kannten diese noch schnellere Methode zur Berechnung des größten gemeinsamen Teilers schon vor 300 v. Chr. Wir finden die Beschreibung in dem Buch von Euklid, das die ganze damalige Mathematik umfasst und in den nächsten, fast 2000 Jahren das verbreiteteste wissenschaftliche Lehrbuch war. Obwohl man den Entdecker der Gleichung $\mathrm{GGT}(a,b) = \mathrm{GGT}(a \bmod b, b)$ nicht kennt, nennen wir die auf dieser Gleichung basierende Methode „Euklidischer Algorithmus“. Wie schnell diese Methode zum Ziel führt, zeigt folgendes Beispiel:

$$\begin{aligned} \mathrm{GGT}(127500136, 12750) &= \mathrm{GGT}(12750, 136) \\ &= \mathrm{GGT}(136, 102) \\ &= \mathrm{GGT}(102, 34) \end{aligned}$$

Weil 34 die Zahl 102 teilt, gilt $\mathrm{GGT}(102,34) = 34$

Aufgabe 2.16 Kannst du bestimmen, wie viele Anwendungen der von uns bewiesenen Gleichung $\mathrm{GGT}(a,b) = \mathrm{GGT}(a-b,b)$ die chinesische Methode braucht, um $\mathrm{GGT}(127500136, 12750)$ zu berechnen?

Aufgabe 2.17 Bestimme mit Hilfe des Euklidischen Algorithmus $\mathrm{GGT}(a,b)$ für folgende Zahlen a und b:

a) $a = 846836$, $b = 25654$

b) $a = 1969917$, $b = 5383167$

Wir können die Gleichung

$$\mathrm{GGT}(a,b) = \mathrm{GGT}(a \bmod b, b)$$

für beliebige ganze Zahlen auch mit Hilfe der schon bewiesenen Gleichung $\mathrm{GGT}(a,b) = \mathrm{GGT}(a-b,b)$ für $a > b$ beweisen. Um das direkte Beweisen zu üben, versuchen wir die Gleichung direkt aus den Teilbarkeiten zwischen den Zahlen herzuleiten.

Wie schon erwähnt, kann man eine Gleichung $x = y$ in zwei Schritten beweisen, indem man die zwei Ungleichungen $x \leq y$ und $x \geq y$ beweist.

Also reicht es zu zeigen, dass

$$\mathrm{GGT}(a,b) \leq \mathrm{GGT}(b, a \bmod b) \text{ und } \mathrm{GGT}(b, a \bmod b) \leq \mathrm{GGT}(a,b)$$

gelten. Wir beweisen zuerst den ersten Teil der Aussage.

Zur Verfügung stehen uns alle bisher bewiesenen und in den Übungen formulierten Aussagen. Die Situation ist also folgende:

Startsituation: Die bisherige Theorie T
Ziel: Zu beweisen, dass „$\mathrm{GGT}(a,b) \leq \mathrm{GGT}(b, a \bmod b)$“ gilt.

Für den Start einer Folge von Implikationen nehmen wir aus T die Definition des gemeinsamen Teilers und die Zerlegung von a bezüglich der Division durch b. Unsere Startaussage ist damit

„$\mathrm{GGT}(a,b)$ teilt a, $\mathrm{GGT}(a,b)$ teilt b und
$a = (a \operatorname{div} b) \cdot b + a \bmod b$“.

$\Rightarrow$ „$\mathrm{GGT}(a,b)$ teilt beide Zahlen a und b, und
$a \bmod b = a - (a \operatorname{div} b) \cdot b$“.
{Eine Gleichung bleibt erhalten, wenn man von beiden Seiten die gleiche Zahl subtrahiert.}

$\Rightarrow$ „$\mathrm{GGT}(a,b)$ teilt $a \bmod b$ und $\mathrm{GGT}(a,b)$ teilt b“
{Gemäß Aussage aus Aufgabe 2.9 teilt jeder Teiler der beiden Zahlen a und b auch jede „lineare“ Kombination $xa + yb$ und somit auch für $x = 1$ und $y = -(a \operatorname{div} b)$ die konkrete lineare Kombination $1 \cdot a - (a \operatorname{div} b) \cdot b = a \bmod b$.}

$\Rightarrow$ „$\mathrm{GGT}(a,b) \in \mathrm{Teiler}_{a \bmod b}$ und $\mathrm{GGT}(a,b) \in \mathrm{Teiler}_b$“
{Nach der Definition der Mengen der Teiler}

$\Rightarrow$ „$\mathrm{GGT}(a,b) \in \mathrm{Teiler}_{a \bmod b,b} = \mathrm{Teiler}_{a \bmod b} \cap \mathrm{Teiler}_b$“

$\Rightarrow$ „$\mathrm{GGT}(a,b) \leq \mathrm{GGT}(b, a \bmod b)$“
{Weil $\mathrm{GGT}(a,b) \leq$ maximum $\{x \mid x \in \mathrm{Teiler}_{a \bmod b,b}\}$
$= \mathrm{GGT}(b, a \bmod b)$ }

Aufgabe 2.18 Beweise, dass $\mathrm{GGT}(b, a \bmod b) \leq \mathrm{GGT}(a,b)$ gilt.

Aufgabe 2.19 Beweise, dass $\mathrm{GGT}(a, \mathrm{GGT}(b,c)) = \mathrm{GGT}(\mathrm{GGT}(a,b),c)$ für alle natürlichen Zahlen a, b und c gilt.

Die meisten Menschen haben wenig Schwierigkeiten, die direkte Argumentation zu verstehen. Die indirekte Argumentation hält man für weniger verständlich. Ob sie wirklich viel komplizierter als die direkte Argumentation ist, oder ob dies eher die Folge unzureichender didaktischer Ansätze in der Schule ist, überlassen wir der Entscheidung des Lesers. Weil wir die indirekte Argumentation zur Erforschung grundlegender Erkenntnisse in anderen Modulen verwenden wollen, erklären wir sie auf der elementarsten Ebene, auf der man das richtige Verständnis am besten entwickeln kann.

Gehen wir wieder von unserem Beispiel aus. Die Aussage A bedeutet „Es regnet“, B bedeutet „Die Wiese ist nass“ und C bedeutet „Die Salamander freuen sich“. Für eine Behauptung D bezeichnen wir durch $\overline{D}$ das Gegenteil. Somit bedeutet $\overline{A}$ „Es regnet nicht“, $\overline{B}$ bedeutet „Die Wiese ist trocken (nicht nass)“ und $\overline{C}$ bedeutet „Die Salamander freuen sich nicht“. Nehmen wir jetzt an, die Folgerungen $A \Rightarrow B$ und $B \Rightarrow C$ gelten. Jetzt stellen wir oder die Biologen fest, dass

„sich die Salamander nicht freuen“,

also dass $\overline{C}$ gilt. Kann man daraus etwas schließen?

Wenn sich die Salamander nicht freuen, kann die Wiese nicht nass sein, weil $B \Rightarrow C$ die Freude der Salamander bei nasser Wiese garantiert. Damit wissen wir mit Sicherheit, dass $\overline{B}$ („Die Wiese ist trocken“) gilt. Analog liefert die Gültigkeit von $A \Rightarrow B$ und $\overline{B}$, dass es nicht regnet, da sonst die Wiese nass sein müsste. Also gilt $\overline{A}$. Damit beobachten wir, dass aus der Gültigkeit von

$A \Rightarrow B$, $B \Rightarrow C$ und $\overline{C}$

die Gültigkeit von

$\overline{B}$ und $\overline{A}$

folgt.

Wir können dies auch in folgender Wahrheitstabelle beobachten (siehe Tab. 2.7).

	A	B	C	$A \Rightarrow B$	$B \Rightarrow C$	C gilt nicht
S_1	gilt	gilt	gilt			ausg.
S_2	gilt	gilt	gilt nicht		ausg.	
S_3	gilt	gilt nicht	gilt	ausg.		ausg.
S_4	gilt	gilt nicht	gilt nicht	ausg.		
S_5	gilt nicht	gilt	gilt			ausg.
S_6	gilt nicht	gilt	gilt nicht		ausg.	
S_7	gilt nicht	gilt nicht	gilt			ausg.
S_8	gilt nicht	gilt nicht	gilt nicht			

Tabelle 2.7 Wahrheitstabelle für A, B und C

Die Gültigkeit von $A \Rightarrow B$ schließt die Situationen S_3 und S_4 aus. Die Gültigkeit von $B \Rightarrow C$ schließt die Situationen S_2 und S_6 aus. Weil $\overline{C}$ gilt (weil C nicht gilt), sind die Situationen S_1, S_3, S_5 und S_7 ausgeschlossen. Damit ist S_8 die einzige Situation, die nicht ausgeschlossen wird. S_8 bedeutet, dass alle drei Aussagen A, B und C nicht gelten, also dass $\overline{A}$, $\overline{B}$ und $\overline{C}$ gelten.

Aufgabe 2.20 Betrachten wir die Aussagen A, B und C wie oben. Nehmen wir an, $A \Rightarrow B$, $B \Rightarrow C$ und $\overline{B}$ gelten. Was kann man daraus schließen? Zeichne die Wahrheitstabelle für alle acht Möglichkeiten bezüglich der Gültigkeit von A, B und C und stelle fest, welche Situationen bei geltenden $A \Rightarrow B$, $B \Rightarrow C$ und $\overline{B}$ möglich sind.

Wir beobachten, dass man aus der Gültigkeit von $A \Rightarrow B$, $B \Rightarrow C$ und C nichts über die Gültigkeit von A und B schließen kann. Wenn C gilt, freuen sich die Salamander. Aber das muss nicht bedeuten, dass die Wiese nass ist (dass B gilt). Die Salamander können auch andere Gründe zur Freude haben. Die nasse Wiese ist nur eine der Möglichkeiten.

Aufgabe 2.21 Zeichne die Wahrheitstabelle für A, B und C und stelle fest, welche Situationen bei geltenden $A \Rightarrow B$, $B \Rightarrow C$ und C möglich sind.

Aufgabe 2.22 Betrachte die folgenden Aussagen C und D. C bedeutet „Gelbe und blaue Farben werden gemischt“ und D bedeutet „Eine grüne Farbe entsteht“. Die Implikation „$C \Rightarrow D$“ bedeutet

> „Wenn die gelben und blauen Farben gemischt werden, entsteht eine grüne Farbe.“

Nehmen wir an, $C \Rightarrow D$ gilt. Zeichne die Wahrheitstabelle für C und D und erkläre, welche Situationen möglich und welche nicht möglich sind. Kannst du aus der Gültigkeit von $C \Rightarrow D$ schließen, dass die Behauptung

> „Wenn keine grüne Farbe bei der Mischung entstanden ist, dann wurde nicht eine blaue Farbe mit einer gelben Farbe gemischt.“

auch gilt?

Wir fangen langsam an, die Vorgehensweise der indirekten Argumentation zu verstehen. Bei direkten Beweisen wissen wir, dass eine Behauptung A gilt und wollen die Gültigkeit einer Zielbehauptung Z beweisen. Um dies zu erreichen, bilden wir eine Folge von korrekten Folgerungen

$$A \Rightarrow A_1, A_1 \Rightarrow A_2, \ldots, A_{k-1} \Rightarrow A_k, A_k \Rightarrow Z,$$

die uns die Gültigkeit von $A \Rightarrow Z$ garantiert. Aus der Gültigkeit von A und $A \Rightarrow Z$ können wir dann die Gültigkeit von Z schließen.

Ein **indirekter Beweis** ist wie folgt aufgebaut:

> **Ausgangssituation:** D gilt
> **Ziel:** Z gilt

Wir starten vom Gegenteil von Z, also von $\overline{Z}$, und entwickeln eine Folge von Folgerungen

$$\overline{Z} \Rightarrow A_1, A_1 \Rightarrow A_2, \ldots, A_{k-1} \Rightarrow A_k, A_k \Rightarrow \overline{D}.$$

Aus dieser Folge können wir schließen, dass $\overline{Z}$ nicht gilt und somit Z gilt.

	D	Z	$\overline{D}$	$\overline{Z}$	$\overline{Z} \Rightarrow \overline{D}$	D gilt
S_1	gilt	gilt	gilt nicht	gilt nicht		
S_2	gilt	gilt nicht	gilt nicht	gilt	ausg.	
S_3	gilt nicht	gilt	gilt	gilt nicht		ausg.
S_4	gilt nicht	gilt nicht	gilt	gilt		ausg.

Tabelle 2.8 Wahrheitstabelle für D und Z

Die Richtigkeit unserer Schlussfolgerung können wir in der Wahrheitstabelle in Tab. 2.8 beobachten. Die Situation S_2 ist durch die Gültigkeit der Folgerung $\overline{Z} \Rightarrow \overline{D}$ ausgeschlossen. Weil D gilt, sind die Situationen S_3 und S_4 ausgeschlossen. In der einzig verbleibenden, möglichen Situation S_1 gilt Z und somit haben wir unsere Zielsetzung erreicht.

Diese Beweismethode heißt indirekte Methode, weil wir in der Kette der Folgerungen von hinten nach vorne argumentieren. Wenn $\overline{D}$ nicht gilt (also wenn D gilt), dann kann auch $\overline{Z}$ nicht gelten und somit gilt Z.

In unserem Beispiel (Tab. 2.7) war $D = \overline{C}$, also wussten wir, dass sich die Salamander nicht freuen. Wir wollten beweisen, dass es dann nicht regnet, also unsere Zielsetzung $Z = \overline{A}$. Der Folgerung

$$A \Rightarrow B, B \Rightarrow C$$

entsprach in unserer neuen Notation

$$\overline{Z} \Rightarrow B, B \Rightarrow \overline{D}.$$

Aus $\overline{Z} \Rightarrow \overline{D}$ und D konnten wir dann schließen, dass das Gegenteil von $\overline{Z} = A$ gelten muss. Das Gegenteil von $\overline{Z}$ ist $Z = \overline{A}$ und somit haben wir bewiesen, dass es nicht regnet (dass $\overline{A}$ gilt).

Im Allgemeinen geht man bei den indirekten Beweisen wie folgt vor: Man will beweisen, dass eine Behauptung Z gilt. Wir bauen eine Kette von Folgerungen

$$\overline{Z} \Rightarrow A_1, A_1 \Rightarrow A_2, \ldots, A_k \Rightarrow U,$$

die mit $\overline{Z}$ anfängt und in einem Unsinn U endet. Als „Unsinn“ bezeichnen wir eine Behauptung, die offensichtlich nicht gelten kann. Zum Beispiel kann man das Gegenteil

einer schon bewiesenen Behauptung unserer Theorie als Unsinn betrachten. Dann sagen wir, dank der Gültigkeit von

$$\overline{Z} \Rightarrow U,$$

dass die Folgerung der Behauptung $\overline{Z}$ (des Gegenteiles von Z) ein Unsinn U ist. Weil der Unsinn U nicht gelten kann, kann auch $\overline{Z}$ nicht gelten. Also gilt das Gegenteil von $\overline{Z}$, was Z ist.

Versuchen wir jetzt, einige indirekte Beweise konkreter mathematischer Aussagen zu führen.

Beispiel 2.4 Sei x^2 eine ungerade Zahl. Unsere Aufgabe ist zu beweisen, dass dann auch x ungerade ist. Zum Beweis verwenden wir die indirekte Argumentation. Außer den bekannten Gesetzen der Arithmetik stehen folgende Definitionen zur Verfügung.

> Eine ganze Zahl x ist genau dann gerade, wenn $x = 2i$ für ein $i \in \mathbb{Z}$ gilt (d.h. wenn x durch 2 teilbar ist).

> Eine ganze Zahl x ist genau dann ungerade, wenn $x = 2j + 1$ für ein $j \in \mathbb{Z}$ gilt (d.h. wenn x nicht durch 2 teilbar ist).

Sei A die Startbehauptung, dass „x^2 ungerade ist“. Sei Z die Zielbehauptung, dass „x ungerade ist“. Nach dem Schema des indirekten Beweises müssen wir durch eine Folge von Implikationen zeigen, dass die Implikation $\overline{Z} \Rightarrow \overline{A}$ gilt. Fangen wir also mit dem Gegenteil $\overline{Z}$ (x ist gerade) von Z (x ist ungerade) an.

„x ist gerade“

$\Leftrightarrow$ „$x = 2i$ für ein $i \in \mathbb{Z}$“
{Nach der Definition von geraden Zahlen}

$\Rightarrow$ „$x^2 = (2i)^2 = 2^2 i^2 = 4i^2 = 2 \cdot (2i^2)$“
{Nach den Rechenregeln der Arithmetik}

$\Rightarrow$ „$x^2 = 2m$ für ein $m \in \mathbb{Z}$“
{Weil $x^2 = 2 \cdot (2i^2)$ und $m = 2i^2$ muss in $\mathbb{Z}$, also eine ganze Zahl sein, weil $i \in \mathbb{Z}$.}

$\Leftrightarrow$ „x^2 ist gerade“
{Nach der Definition von geraden Zahlen}

Damit haben wir die Implikation $\overline{Z} \Rightarrow \overline{A}$ bewiesen, d. h. es gilt

„x ist gerade“ $\Rightarrow$ „x^2 ist gerade“

Jetzt wissen wir, dass A („x^2 ist ungerade“) gilt und somit gilt $\overline{A}$ („x^2 ist gerade“) nicht. Nach dem Schema des indirekten Beweises können wir schließen, dass Z („x ist ungerade“) gilt. □

Aufgabe 2.23 Zeige mittels eines indirekten Beweises, dass gilt: Wenn x^2 eine gerade Zahl ist, dann muss auch x eine gerade Zahl sein. Gehe dabei so detailliert vor, wie wir es in Beispiel 2.4 vorgeführt haben.

Wir haben mit einem indirekten Beweis gerade gezeigt, dass die Behauptung „x^2 ist ungerade“ die Behauptung „x ist ungerade“ impliziert. Analog habt ihr in Aufgabe 2.23 gezeigt, dass die Behauptung „x^2 ist durch 2 teilbar (gerade)“ die Behauptung „x ist durch 2 teilbar (gerade)“ impliziert. Gilt dies auch für die Teilbarkeit durch andere Zahlen? Wir zeigen es für die Zahl 3.

Beispiel 2.5 Wir wissen, dass x^2 durch 3 teilbar ist und wollen beweisen, dass x auch durch 3 teilbar sein muss. Wir beweisen es durch den indirekten Beweis. Das Gegenteil $\overline{Z}$ unserer Zielsetzung $Z =$ „x ist durch 3 teilbar“ ist „x ist nicht durch 3 teilbar“. Also fangen wir mit $\overline{Z}$ an.

„x ist nicht durch 3 teilbar“

$\Leftrightarrow$ „x kann man nicht als $x = 3i$ schreiben für ein $i \in \mathbb{Z}$“
{Aus der Definition der Teilbarkeit}

$\Leftrightarrow$ „$x = 3i+1$ für ein $i \in \mathbb{Z}$ oder $x = 3j+2$ für ein $j \in \mathbb{Z}$“
{Wenn x nicht durch 3 teilbar ist, dann gibt es einen Rest nach der Teilung von x durch 3. Der Rest kann nur 1 oder 2 sein.}

$\Rightarrow$ „$x^2 = (3i+1)^2 = 9i^2+6i+1 = 3\cdot(3i^2+2i)+1$ oder $x^2 = (3j+2)^2 = 9j^2+12j+4 = 3\cdot 3j^2+3\cdot 4j+3+1 = 3\cdot(3j^2+4j+1)+1$“
{Nach dem Distributivgesetz und Regeln der Arithmetik}

$\Rightarrow$ „$x^2 = 3m+1$ für ein $m \in \mathbb{Z}$“
{Entweder gilt $m = 3i^2 + 2i$ oder $m = 3j^2 + 4j + 1$ und somit ist $m \in \mathbb{Z}$, weil $i \in \mathbb{Z}$ bzw. $j \in \mathbb{Z}$}

$\Rightarrow$ „x^2 ist nicht durch 3 teilbar“

Damit haben wir das Gegenteil unserer Annahme „x^2 ist durch 3 teilbar“ erhalten und können daraus schließen, dass Z („x ist durch 3 teilbar“) gilt. □

Hinweis für die Lehrperson Jetzt ist es ganz wichtig, von den Schülerinnen und Schülern zu fordern, dass sie ihre eigenen Beweise genauso sorgfältig und detailliert aufschreiben, wie wir es in den Beispielen vorgeführt haben. Es ist oft auch hilfreich, unterschiedliche Aufgaben in der Klasse zu verteilen und dann die Beweise vorführen zu lassen.

Aufgabe 2.24 Beweise mittels eines indirekten Beweises die folgende Aussage: „Wenn x^2 nicht durch 3 teilbar ist, dann ist auch x nicht durch 3 teilbar.“

Aufgabe 2.25 Beweise mittels eines indirekten Beweises die folgende Behauptung: „Wenn x^2 nicht durch 5 teilbar ist, dann ist auch x nicht durch 5 teilbar.“

Aufgabe 2.26 Gilt die folgende Behauptung?

„Wenn x^2 durch 6 teilbar ist, dann ist auch x durch 6 teilbar.“

Begründe deine Antwort. Beachte, dass es einen Beweis erfordert, um die Gültigkeit dieser Behauptung zu begründen, während es für den Beweis der Ungültigkeit reicht, eine konkrete Zahl x zu finden, für die die Behauptung nicht gilt.
Ändert sich etwas an der Gültigkeit oder der Ungültigkeit dieser Implikation, wenn man 6 durch 12 ersetzt?

Die Aussagen, die wir gerade bewiesen haben, sind nützlich, um eine wichtige Entdeckung aus der Antike herzuleiten. Am Anfang waren Menschen, besonders die Pythagoreer, von Zahlen begeistert und wollten damit die ganze Welt erklären. Es gab die ganzen Zahlen und dann die Zahlen, die man aus den ganzen Zahlen durch die arithmetischen Operationen $+$, $-$, $\cdot$ und $/$ gewinnen konnte. Die Philosophen (so nannten sich die Wissenschaftler damals) haben beobachtet, dass alle diese Zahlen sich als Brüche darstellen lassen.

Aufgabe 2.27 Zeige mittels direkter Beweise, dass die Zahlen $a+b$, $a-b$, $a \cdot b$ und $\frac{a}{b}$ (für $b \neq 0$) sich immer als Brüche $\frac{p}{q}$ darstellen lassen, wenn a und b auch Brüche sind.

Die Menschen in der Antike nannten daher diese Zahlen die rationale Zahlen, weil man sie durch arithmetische Berechnungen erzeugen konnte. Es war für sie ein Schock und damit ein Widerspruch zu ihrer Philosophie (alles kann man durch Zahlen und arithmetische Operationen über Zahlen beschreiben), als sie entdeckten, dass es in der realen Welt Zahlen gibt, die man nicht berechnen kann (d.h. nicht als Brüche darstellen kann). So eine Zahl ist die Zahl $\sqrt{2}$, die offensichtlich geometrisch nach dem Satz von Pythagoras erzeugbar ist (Abb. 2.1).

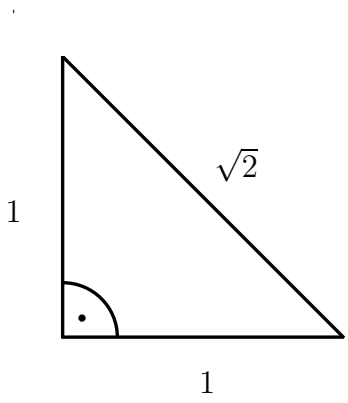


Abbildung 2.1 Geometrische Konstruktion der Zahl $\sqrt{2}$.

Die durch eine endliche Anzahl arithmetischer Operationen unberechenbaren Zahlen nannten sie „**irrationale**“ Zahlen. Für die Pythagoreer war die Existenz der irrationalen Zahlen ein Paradoxon, mit dem sie nie fertig geworden sind. Später werden wir lernen, dass es noch schlimmer ist. Nicht nur, dass man gewisse Zahlen in endlicher Zeit nicht mit absoluter Genauigkeit berechnen kann, es gibt sogar auch Zahlen, die man auf endliche Weise auch nicht beschreiben kann (auch nicht geometrisch).

Aufgabe 2.28 Finde Zahlen k und x, so dass k die Zahl x^2 teilt und k kein Teiler von x ist.

Wie hat man damals gezeigt, dass $\sqrt{2}$ keine rationale Zahl ist? Mit einem indirekten Beweis, den wir jetzt vorführen. Wir nutzen dabei die Kenntnisse, die wir über den Bezug zwischen der Teilbarkeit von x^2 und der Teilbarkeit von x gewonnen haben.

Satz 2.1 *Die Zahl $\sqrt{2}$ (also die Lösung der Gleichung $x^2 = 2$) ist keine rationale Zahl.*

Beweis: Folgend dem Schema der indirekten Argumentation fangen wir mit dem Gegenteil unserer Zielsetzung an.

„$\sqrt{2}$ ist eine rationale Zahl (ein Bruch)“

$\Rightarrow$ „$\sqrt{2} = \frac{p}{q}$ für zwei Zahlen $p, q \in \mathbb{Z}$, $q \neq 0$, mit $\mathrm{GGT}(p,q) = 1$“
{Wenn $\mathrm{GGT}(p,q)$ größer als 1 wäre, könnten wir den Bruch $\frac{p}{q}$ kürzen und würden den unkürzbaren Bruch $\frac{r}{s}$ mit $r = \frac{p}{\mathrm{GGT}(p,q)}$ und $s = q\mathrm{GGT}(p,q)$ erhalten. Damit wissen wir, dass sich jede rationale Zahl als unkürzbarer Bruch darstellen lässt.}

$\Leftrightarrow$ „$\sqrt{2}q = p$ für $p, q \in \mathbb{Z}$, $q \neq 0$ mit $\mathrm{GGT}(p,q) = 1$“
{Die Gleichung bleibt erhalten, wenn wir ihre beiden Seiten mit der gleichen Zahl $q \neq 0$ multiplizieren.}

$\Rightarrow$ „$2q^2 = p^2$ für $p, q \in \mathbb{Z}$, $q \neq 0$ mit $\mathrm{GGT}(p,q) = 1$“
{Die Gleichung bleibt erhalten, wenn man beide Seiten quadriert.}

$\Rightarrow$ „$2q^2 = p^2$ für $p, q \in \mathbb{Z}$, $q \neq 0$ mit $\mathrm{GGT}(p,q) = 1$ und p^2 ist gerade“
{$p^2 = 2 \cdot q^2 = 2 \cdot i$ für ein $i \in \mathbb{Z}$}

$\Rightarrow$ „$2q^2 = p^2$ für $p, q \in \mathbb{Z}$, $q \neq 0$ mit $\mathrm{GGT}(p,q) = 1$ und p ist gerade“
{Die Behauptung „p^2 ist gerade“ impliziert die Behauptung „p ist gerade“}

$\Rightarrow$ „$2q^2 = p^2$ für $p, q \in \mathbb{Z}$, $q \neq 0$ mit $\mathrm{GGT}(p,q) = 1$ und $p = 2k$ für ein $k \in \mathbb{Z}$“
{Nach der Definition der Teilbarkeit}

$\Rightarrow$ „$2q^2 = (2k)^2 = 4k^2$ für k,$q \in \mathbb{Z}$, $p = 2k$ für ein $k \in \mathbb{Z}$ und $\mathrm{GGT}(p,q) = 1$“

$\Rightarrow$ „$q^2 = 2k^2$ für k,$q \in \mathbb{Z}$, $p = 2k$ für ein $k \in \mathbb{Z}$ und $\mathrm{GGT}(p,q) = 1$“
{Die Gleichung bleibt erhalten, wenn man beide Seiten durch 2 teilt.}

$\Rightarrow$ „q^2 ist gerade, $\mathrm{GGT}(p,q) = 1$ und $p = 2k$ für ein $k \in \mathbb{Z}$“
{Weil $q^2 = 2 \cdot j$ für ein $j = k^2 \in \mathbb{Z}$.}

$\Rightarrow$ „q ist gerade, $\mathrm{GGT}(p,q) = 1$ und p ist gerade ($p = 2k$ für ein $k \in \mathbb{Z}$)“
{Weil die Behauptung „q^2 ist gerade“ die Behauptung „q ist gerade“ impliziert.}

Wir sehen schon, dass die Schlussbehauptung ein Unsinn ist. Wie können beide Zahlen p

und q gerade sein und dabei gelten $\mathrm{GGT}(p,q) = 1$? Da muss doch bei geraden p und q der größte gemeinsame Teiler mindestens 2 sein.

Nach dem Schema des indirekten Beweises muss das Gegenteil der Startbehauptung gelten und somit ist $\sqrt{2}$ nicht rational. □

Hinweis für die Lehrperson Der häufigste didaktische Fehler bei der Darstellung der Beweise ist, dass man versucht, alles so kurz wie möglich aufzuschreiben. Das führt oft dazu, dass man am Ende eine Aussage erhält und dann argumentiert, dass diese Aussage im Widerspruch zu einer der Aussagen in der Implikationskette steht. Für Anfänger ist es aber viel besser, wenn man alle wichtigen Aussagen (verbunden mit einem „und“) mitführt und dann in der letzten, abgeleiteten Aussage zwei mit „und“ verknüpfte, widersprüchliche Behauptungen erhält. Damit ist die letzte Aussage ein offensichtlicher Unsinn und das ganze Vorgehen verfolgt konsequent das Schema des indirekten Beweises. Wenn man das Schema nicht einhält, ist es sehr schwierig für einen Anfänger, ein Gefühl dafür zu entwickeln, was erlaubt ist und was nicht.

Aufgabe 2.29 Beweise, dass auch $\sqrt{3}$ keine rationale Zahl ist. Nutze dabei die Tatsache, dass die Teilbarkeit von x^2 durch 3 die Teilbarkeit von x durch 3 impliziert.

Hinweis für die Lehrperson Spätestens ab dieser Stelle richtet sich der Stoff nur an Schülerinnen und Schüler in den letzten zwei Jahren der gymnasialen Ausbildung. Dies gilt bis zum Ende dieser Lektion.

Aufgabe 2.30 Nehmen wir an, dass 4 eine Zahl x^2 teilt. Kann man daraus schließen, dass 4 auch die Zahl x teilt? Begründe deine Antwort!

Aufgabe 2.31 Nehmen wir an, dass 8 eine Zahl x^2 teilt. Kann man daraus schließen, dass 8 auch die Zahl x teilt? Beweise die Gültigkeit deiner Behauptung!

Aufgabe 2.32 (Knobelaufgabe) Beweise die folgende Behauptung: „Für jede Primzahl p gilt, dass die Teilbarkeit von x^2 durch p die Teilbarkeit von x durch p impliziert.“

Aufgabe 2.33 Beweise, dass die Behauptung „x^2 ist nicht durch 7 teilbar“ die Behauptung „x ist nicht durch 7 teilbar“ impliziert.

Aufgabe 2.34 (Knobelaufgabe) Beweise, dass $\sqrt{6}$ keine rationale Zahl ist.

Aufgabe 2.35 (Knobelaufgabe) Versuche, einen direkten Beweis der Implikation

„x^2 ist gerade“ $\Rightarrow$ „x ist gerade“

zu führen. Warum ist es nicht ganz einfach? Erkennst du, welche zusätzliche Behauptung du brauchst, um auch den direkten Beweis dieser Tatsache effizient führen zu können?

Betrachten wir jetzt noch folgende Aufgabe, um die „Effizienz“ des indirekten Beweises zu begreifen. Unsere Zielsetzung ist zu beweisen, dass $\sqrt{12}$ keine rationale Zahl ist. Der bisher gegangene Weg der Teilbarkeit wird nicht funktionieren, denn wenn x^2 durch 12 teilbar ist, bedeutet es noch nicht, dass auch x durch 12 teilbar ist. Zum Beispiel für $x = 6$ haben wir $x^2 = 6 \cdot 6 = 36$. Die Zahl 36 ist offensichtlich durch 12 teilbar, aber 6 ist nicht durch 12 teilbar.

Aufgabe 2.36 Finde weitere Zahlen x, so dass 12 die Zahl x^2 teilt, aber x nicht teilt.

Trotzdem können wir mit folgendem indirekten Beweis schnell und leicht zeigen, dass $\sqrt{12}$ keine rationale Zahl ist.

„$\sqrt{12}$ ist rational und somit gilt $\sqrt{12} = \frac{p}{q}$ für geeignete $p, q \in \mathbb{Z}$“

$\Rightarrow$ „$\sqrt{12} = \sqrt{2^2 \cdot 3} = 2 \cdot \sqrt{3}$ ist rational, $\sqrt{12} = \frac{p}{q}$ für $p, q \in \mathbb{Z}$“

$\Rightarrow$ „$\sqrt{3} = \frac{\sqrt{12}}{2}$ und $\sqrt{12} = \frac{p}{q}$ für $p, q \in \mathbb{Z}$“
{Eine Gleichung bleibt erhalten, wenn man beide Seiten durch die gleiche Zahl, hier 2, teilt.}

$\Rightarrow$ „$\sqrt{3} = \frac{(\frac{p}{q})}{2} = \frac{p}{2q}$“
{Einsetzen von $\frac{p}{q}$ für $\sqrt{12}$ in die Gleichung $\sqrt{3} = \frac{\sqrt{12}}{2}$.}

$\Rightarrow$ „$\sqrt{3}$ ist rational“
{Weil $\sqrt{3}$ sich als ein Bruch $\frac{v}{s}$ für $v = p$ und $s = 2q$, $v, s \in \mathbb{Z}$ darstellen lässt.}

Wir wissen schon, dass $\sqrt{3}$ keine rationale Zahl ist (Aufgabe 2.29) und somit ist die Behauptung „$\sqrt{3}$ ist rational“ ein Unsinn. Somit haben wir bewiesen, dass $\sqrt{12}$ eine irrationale Zahl ist.

Aufgabe 2.37 Beweise mit indirekten Beweisen, dass folgende Zahlen keine rationalen Zahlen sind.

a) $\sqrt{18}$

b) $\sqrt{50}$

c) $\sqrt{54}$

In dem Teil über direkte Beweise haben wir die folgende Behauptung verwendet, ohne ihre Gültigkeit zu beweisen:

> *Für beliebige positive ganze Zahlen a und b kann man a eindeutig als*
>
> $$a = k \cdot b + r$$
>
> *für geeignete k, $r \in \mathbb{Z}$, $r < b$, darstellen.*

Dass sich a als $k \cdot b + r$ mit $r < b$ darstellen lässt, ist offensichtlich. Die Zahl $k = a \operatorname{div} b$ ist das Resultat der ganzzahligen Division von a durch b und r ist einfach der Rest der Division ($r = a \bmod b$). Es ist zu beweisen, dass diese Darstellung eindeutig ist, d.h., dass es nicht mehrere solche Darstellungen von a für gegebenes b gibt. Zeigen wir die Eindeutigkeit der Darstellung von a mit einem indirekten Beweis. Wir starten mit dem Gegenteil unserer Zielsetzung.

> „Seien $a = k \cdot b + r$ und $a = k' \cdot b + r'$, $r < b$, $r' < b$ zwei unterschiedliche ($k \neq k'$ oder $r \neq r'$) Darstellungen von a bezüglich der Division $a \operatorname{div} b$."

$\Rightarrow$ „$0 = a - a = k \cdot b + r - k' \cdot b - r' = (k - k') \cdot b + r - r'$ und $r < b$, $r' < b$"
{Nach dem Distributivgesetz}

$\Rightarrow$ „$r' - r = (k - k') \cdot b$ und $r < b$, $r' < b$"
{Die Gleichung bleibt bestehen, wenn man zu beiden Seiten die gleiche Zahl $r' - r$ addiert.}

$\Rightarrow$ „($k - k' = 0$ und $r' - r = (k - k') \cdot b$) oder ($b$ teilt $r' - r$ und $r' - r = (k - k') \cdot b$)"
{Nach der Definition der Teilbarkeit muss b die Zahl $r - r'$ teilen oder es ist $k - k' = 0$.}

$\Rightarrow$ „$(k = k'$ und $r' - r = 0 \cdot b = 0)$ oder $(r = r'$ und $r' - r = (k - k') \cdot b)$“
{Weil $r < b$ und $r' < b$ ist, muss $|r - r'| < b$ gelten. Die einzige nicht negative Zahl kleiner als b, die durch b teilbar ist, ist die Zahl 0.}

$\Rightarrow$ „$(k = k'$ und $r' = r)$ oder $(r = r'$ und $0 = (k - k') \cdot b)$“

$\Rightarrow$ „$(k = k'$ und $r' = r)$ oder $(r = r'$ und $k = k')$“
{Weil $b > 0$ gilt. Und wenn $x \cdot y = 0$ gilt, dann muss mindestens eine der Zahlen x und y die Zahl 0 sein.}

$\Rightarrow$ „$k = k'$ und $r' = r$“

Die Tatsache $k = k'$ und $r' = r$ widerspricht der Startbehauptung, dass $k \neq k'$ oder $r \neq r'$. Somit gilt das Gegenteil unserer Startbehauptung und daher gibt es nur eine Darstellung von a bezüglich der ganzzahligen Division $a \operatorname{div} b$.

Aufgabe 2.38 (Knobelaufgabe) Sei $a = p \cdot q$, wobei p und q Primzahlen sind. Beweise, dass p und q die einzigen Primzahlen sind, die a teilen.

Aufgabe 2.39 (Knobelaufgabe) Beweise, dass es unendlich viele Primzahlen gibt.

In dieser Lektion haben wir mehrere direkte und indirekte Beweise geführt. Zum Abschluss wollen wir einen wichtigen Zusammenhang zwischen diesen beiden Beweismethoden beobachten.

	A	Z	$\overline{A}$	$\overline{Z}$	$A \Rightarrow Z$	$\overline{Z} \Rightarrow \overline{A}$
S_1	1	1	0	0		
S_2	1	0	0	1	ausg.	ausg.
S_3	0	1	1	0		
S_4	0	0	1	1		

Tabelle 2.9

Aus der Tabelle 2.9 können wir die beiden Folgerungen $A \Rightarrow Z$ und $\overline{Z} \Rightarrow \overline{A}$ ableiten. Die Folgerung $A \Rightarrow Z$ entspricht dem direkten Beweis der Zielbehauptung Z als Folge der Startbehauptung A. Die Implikation $\overline{Z} \Rightarrow \overline{A}$ entspricht dem indirekten Beweis der Behauptung Z aus der Behauptung A. In beiden Fällen also beweisen wir die Gültigkeit

von Z bei der Voraussetzung der Gültigkeit von A. Das Ziel ist gleich, nur die Wege zum Ziel sind anders. In Tab. 2.9 sehen wir, dass beide Implikationen $A \Rightarrow Z$ und $\overline{Z} \Rightarrow \overline{A}$ die gleiche Situation S_2 ausschliessen. Es bedeutet nichts anderes, als dass beide die gleiche Bedeutung haben. Oder mit anderen Worten ausgedrückt:

Die Implikation $A \Rightarrow Z$ gilt genau dann, wenn $\overline{Z} \Rightarrow \overline{A}$ gilt.

Dies bestätigt die Tatsache, dass direkte und indirekte Beweise gleichwertig und somit gleich zuverlässig sind. Wenn wir in einem indirekten Beweis die Folgerung $\overline{Z} \Rightarrow \overline{A}$ beweisen, wissen wir sofort, dass auch die Folgerung $A \Rightarrow Z$ gilt, die einem direkten Beweis entspricht und umgekehrt.

Aufgabe 2.40 Formuliere zu den folgenden Sätzen der Form $A \Rightarrow Z$ den Satz $\overline{Z} \Rightarrow \overline{A}$ mit äquivalenter Bedeutung.

a) Wenn der Wind weht, spannen sich die Segel.

b) Wenn man ins Wasser springt, wird man nass.

c) Wenn wir schnell laufen, dann schwitzen wir.

d) Wenn man sich gut vorbereitet hat, hat man gute Aussichten in der Prüfung.

e) Wenn es schönes Wetter ist, hat man hier gute Aussicht.

Im Prinzip kann man die Axiome der korrekten Folgerung auch als eine Begriffsbildung ansehen, in der der Begriff der Implikation (oder der Folgerung) in einem formalen Denksystem definiert wird. Axiome sind oft nichts anderes als eine Festlegung der Bedeutung gewisser Begriffe. Wir werden später die Definition des Unendlichen kennenlernen, die unsere Vorstellung über die Bedeutung der Unendlichkeit mathematisch festlegt. Natürlich ist es nicht möglich zu beweisen, dass diese Definition unseren Vorstellungen entspricht. Aber es besteht die Möglichkeit, ein Axiom zu widerlegen. Zum Beispiel kann jemand etwas finden, was nach unseren Vorstellungen unendlich sein sollte, aber nach der Definition nicht unendlich ist. Wenn so etwas passieren würde, muss man das Axiom revidieren. Eine Revision eines Axioms oder einer Definition sollte man aber nicht als ein Unglück und schon gar nicht als eine Katastrophe betrachten. Die Ersetzung eines Bausteins des Wissenschaftsgebäudes könnte zwar zu einem aufwändigen Umbau führen,

aber dies ist ein erfreuliches Ereignis, weil das neue Gebäude wesentlich stabiler und besser ist.

Zusammenfassung

Der Begriff der Folgerung (der Implikation) spielt eine Schlüsselrolle für eine korrekte Argumentation. Die Bedeutung der Folgerung $A \Rightarrow B$ (A impliziert B) ist, dass, wenn A gilt (wahr ist), auch B gilt. Es darf also nicht passieren, dass bei geltender Implikation $A \Rightarrow B$ die Aussage A wahr ist und die Aussage B nicht wahr ist. Alle anderen Situationen sind möglich. Wenn A nicht gilt, stellt die Folgerung $A \Rightarrow B$ keine Anforderungen an die Wahrhaftigkeit von B.

Einen direkten Beweis kann man als eine Folge von Implikationen

$$A \Rightarrow B_1 \Rightarrow B_2 \Rightarrow \cdots \Rightarrow B_k \Rightarrow Z$$

betrachten. Wenn alle diese Implikationen gelten, dann gilt auch die Implikation $A \Rightarrow Z$. Wenn der Beweis der Implikation (der Aussage) „$A \Rightarrow Z$“ unser Ziel war, dann sind wir fertig. Wenn wir schon wissen, dass A eine allgemeine Wahrheit in unserer Theorie ist und Z die Aussage ist, deren Wahrhaftigkeit wir beweisen sollen, dann dürfen wir nach der Definition der Implikation aus der Wahrhaftigkeit von

$$A \text{ und } A \Rightarrow Z$$

die Gültigkeit unserer Zielbehauptung Z schließen.

Der indirekte Beweis basiert auf der Tatsache, dass die Gültigkeit einer Aussage A und der Implikation $\overline{Z} \Rightarrow \overline{A}$ die Gültigkeit der Aussage Z fordert. Mit anderen Worten: Die Bedeutungen der Implikationen $A \Rightarrow Z$ und $\overline{Z} \Rightarrow \overline{A}$ sind äquivalent. Mit beiden kann man aus der Gültigkeit der Aussage A die Gültigkeit der Aussage Z schließen. Somit arbeitet das Schema des indirekten Beweises wie folgt: Wir wissen, dass A gilt und wollen die Gültigkeit von Z beweisen. Wir fangen mit $\overline{Z}$ als dem Gegenteil von Z an und erzeugen eine Folge von Implikationen.

$$\overline{Z} \Rightarrow B_1 \Rightarrow B_2 \Rightarrow \cdots \Rightarrow B_k \Rightarrow U$$

Damit ist die Implikation $\overline{Z} \Rightarrow U$ bewiesen. Wenn $U = \overline{A}$ gilt, sind wir fertig und können die Gültigkeit von Z behaupten. Im Allgemeinen muss U nicht identisch mit $\overline{A}$ sein. Es

reicht, wenn U eine unbestrittene Unwahrheit (Unsinn) ist, also eine Behauptung, die im Widerspruch zu irgendwelchen Sätzen (Kenntnissen) unserer Theorie steht.

In vielen Situationen ermöglichen indirekte Beweise eine vereinfachte und damit schnellere Beweisführung. Ohne Gebrauch dieses Instrumentes bei korrekter Argumentation würden wir uns das Leben unnötig schwer machen.

Kontrollfragen

1. Wie viele unterschiedliche Situationen sind möglich und in einer Wahrheitstabelle beschrieben, wenn man drei Aussagen A, B und C betrachtet? Wie groß ist die Wahrheitstabelle bei vier Aussagen? Wie ist es im allgemeinen bei n Behauptungen?

2. Erkläre mit eigenen Worten, wie du systematisch eine Wahrheitstabelle für gegebene Aussagen in beliebiger Anzahl konstruieren kannst.

3. Erkläre die Bedeutung der Implikation. Welche Situationen sind möglich und welche ausgeschlossen?

4. Warum kann man aus der Gültigkeit der Implikationen $A \Rightarrow B$ und $B \Rightarrow C$ auf die Gültigkeit der Implikation $A \Rightarrow C$ schließen?

5. Wie sieht das Schema des direkten Beweises aus?

6. Warum kann man aus der Gültigkeit der Aussage A und der Implikation $A \Rightarrow Z$ die Gültigkeit der Aussage Z ableiten?

7. Warum kann man aus der Gültigkeit der Aussage A und der Implikation $\overline{Z} \Rightarrow \overline{A}$ auf die Gültigkeit der Aussage Z schließen? Erkläre es mit eigenen Worten sowie mit der Wahrheitstabelle für die Behauptungen A und Z!

8. Wie sieht das Schema des indirekten Beweises aus?

9. Wie hängen direkte und indirekte Beweise zusammen?

10. Warum kann es vorteilhaft sein, direkte und indirekte Beweise zur Verfügung zu haben? Gib ein konkretes Beispiel an, für das ein indirekter Beweis einfacher ist und schneller zum Ziel führt als ein direkter Beweis.

Kontrollaufgaben

1. Seien A, B und C drei Behauptungen. Wir wissen, dass folgende Implikationen gelten.

   (i) $A \Rightarrow B, \overline{B} \Rightarrow C, C \Rightarrow A, \overline{A} \Rightarrow C$.

   (ii) $A \Rightarrow \overline{B}, \overline{B} \Rightarrow \overline{C}, \overline{C} \Rightarrow A, \overline{C} \Rightarrow B, \overline{A} \Rightarrow \overline{C}$.

   (iii) $B \Rightarrow C, \overline{B} \Rightarrow \overline{A}, C \Rightarrow \overline{B}, C \Rightarrow A, C \Rightarrow \overline{A}, \overline{A} \Rightarrow \overline{C}$

   Bestimme für alle drei Fälle (i), (ii) und (iii), welche der acht Situationen möglich und welche ausgeschlossen sind!

2. Betrachte die Tabelle 2.10.

	A	B	C	
S_1	1	1	1	ausgeschlossen
S_2	1	1	0	
S_3	1	0	1	ausgeschlossen
S_4	1	0	0	ausgeschlossen
S_5	0	1	1	
S_6	0	1	0	ausgeschlossen
S_7	0	0	1	ausgeschlossen
S_8	0	0	0	

Tabelle 2.10

   Welche der 36 Implikationen $X \Rightarrow Y$ für $X, Y \in \{A, B, C, \overline{A}, \overline{B}, \overline{C}\}$ gelten und welche gelten nicht?

3. Betrachten wir die folgenden drei Aussagen:
   A bedeutet: „Es ist unter $-20°$C.“
   B bedeutet: „Die Kleinvögel frieren.“
   C bedeutet: „Die Kleinvögel singen.“
   Jetzt wissen wir, dass die Implikationen $A \Rightarrow B$ und $B \Rightarrow \overline{C}$ gelten. Wir stellen fest, dass die Kleinvögel singen. Was kann man daraus schließen?

4. In der Tabelle 2.11 fehlen in der ersten Zeile drei Implikationen. Kannst du sie bestimmen?

	A	B	C	$\Rightarrow$	$\Rightarrow$	$\Rightarrow$
S_1	1	1	1	ausg.		
S_2	1	1	0	ausg.		
S_3	1	0	1			ausg.
S_4	1	0	0		ausg.	
S_5	0	1	1			
S_6	0	1	0			
S_7	0	0	1			ausg.
S_8	0	0	0		ausg.	

Tabelle 2.11

Jetzt stellen wir fest, dass C gilt. Was kann man daraus schließen? Und was würde man schließen, wenn $\overline{C}$ gelten würde?

5. Sei A die Behauptung „$3x-6=5x-10$“ und sei B die Behauptung „$x=2$“. Beweise die Gültigkeit der Implikation $A \Rightarrow B$.

6. Beweise für A und B aus Kontrollaufgabe 5 mit einem indirekten Beweis die Behauptung

   „Wenn $3x-6=5x-10$ gilt, dann gilt $x=2$.“

7. Beweise mit einem direkten Beweis die folgende Behauptung: „Wenn die linearen Gleichungen

   $$2x+y=0$$
   $$7x-2y=-11$$

   gelten, dann gilt $x=-1$ und $y=2$“. Dabei darfst du alle dir bekannten Umformungen von Gleichungen und linearen Gleichungssystemen als gültige Behauptungen (Sätze) deiner Theorie verwenden.

8. Sei

   $$ax+by=c$$
   $$dx+ey=f$$

   die allgemeine Darstellung eines Systems von zwei linearen Gleichungen und zwei Unbekannten x und y. Beweise mittels des direkten Beweises, dass die Gültigkeit dieser zwei Gleichungen die Gültigkeit der folgenden Gleichung impliziert:

   $$y=\frac{af+de}{ac+db}.$$

9. Kannst du die Formel für die Berechnung des Wertes der Unbekannten x im Falle der Gültigkeit des Gleichungssystems in Kontrollaufgabe 8 überprüfen?

10. Beweise, dass „wenn $ax^2 - c = 0$ und $\frac{c}{a} > 0$ gelten, dann gilt $x \in \{\sqrt{\frac{c}{a}}, -\sqrt{\frac{c}{a}}\}$."

11. Nehmen wir an, dass $x^2 + bx + c = 0$ und $b^2 - 4c \geq 0$ gelten. Beweise, dass dann die Lösungen x_1, x_2 der quadratischen Gleichung die Gleichungen

    $$b = -x_1 - x_2$$
    $$c = x_1 \cdot x_2$$

    erfüllen. Dabei kannst du folgende zwei Sätze als gültig in deiner Theorie betrachten:

    (i) Zwei Polynome sind nur dann gleich, wenn alle ihre Koeffizienten gleich sind.

    (ii) Wenn x_1 und x_2 die Nullstellen eines quadratischen Polynoms $p(x)$ sind (Lösungen der entsprechenden quadratischen Gleichung), dann gilt: $p(x) = (x - x_1) \cdot (x - x_2)$.

12. Beweise die folgende Aussage: Wenn die Behauptung „p teilt a und p teilt b" gilt, dann gilt auch die Behauptung „p teilt $5a - 3b$".

13. Beweise die folgende Aussage: Wenn die Zahl a die Zahl b teilt, dann teilt das $\mathrm{GGT}(a,b)$ die Zahl b.

14. (**Knobelaufgabe**) Beweise, dass für beliebige positive ganze Zahlen a und b die Zahl $\mathrm{GGT}(a,b)$ die kleinste positive ganze Zahl in der unendlichen Menge $\{a \cdot x + b \cdot y \mid x, y \in \mathbb{Z}\}$ ist.

15. Beweise mittels eines indirekten Beweises die folgende Behauptung: „Wenn 5 die Zahl x^2 teilt, dann teilt 5 auch x".

16. Beweise, dass $\sqrt{5}$ keine rationale Zahl ist.

17. Finde die kleinste Zahl j, so dass die folgende Implikation nicht für alle $x \in \mathbb{Z}^+$ gilt:

    „Wenn j die Zahl x^2 teilt, dann teilt j auch die Zahl x."

18. Nehmen wir an, dass wir in unserer Theorie schon bewiesen haben:

    (i) $\sqrt{2}$, $\sqrt{3}$ und $\sqrt{5}$ sind irrationale Zahlen.

(ii) die Teilbarkeit von x^2 durch 2, 3 oder 5 lässt auf die Teilbarkeit von x durch die entsprechende Zahl schließen.

a) Beweise, dass $\sqrt{10}$ und $\sqrt{15}$ irrationale Zahlen sind.

b) Nenne drei weitere irrationale Zahlen, deren Irrationalität durch einen indirekten Beweis auf die Irrationalität der Zahlen $\sqrt{2}$, $\sqrt{3}$ oder $\sqrt{5}$ zurückgeführt werden kann.

19. Formuliere zu den folgenden Sätzen der Form $A \Rightarrow Z$ den Satz $\overline{Z} \Rightarrow \overline{A}$ mit äquivalenter Bedeutung:

a) Wenn es zu kalt ist, singen die Vögel nicht.

b) Wenn man zu schnell fährt, dann riskiert man eine Buße.

c) Wer probiert (sich bemüht), der leidet.

d) Wer nicht probiert, der leidet nicht, sondern rostet.

e) Wer nichts versteht, der muss alles glauben.

Lösungen zu ausgesuchten Aufgaben

Aufgabe 2.3
Wie viele Implikationen sind zu betrachten? Wir haben drei Aussagen A, B und C und jedes Paar bestimmt eine Implikation. Die Implikation $A \Rightarrow B$ ist eine andere als die Implikation $B \Rightarrow A$, also ist die Reihenfolge wichtig. Somit gibt es $3 \cdot 3 = 9$ unterschiedliche Implikationen, über deren Gültigkeit wir entscheiden sollen. Die Tabelle in Abb. 2.12 enthält alle. Weil die Argumentation in den meisten Fällen sehr ähnlich ist, begründen wir nicht jede der Feststellungen aus Tab. 2.12. Offensichtlich sind die Implikationen $A \Rightarrow A$, $B \Rightarrow B$ und $C \Rightarrow C$ immer gültig, unabhängig davon, wie die Wahrheitstabelle für die Aussagen A, B und C aussieht. Für die Bestimmung der Gültigkeit der restlichen Implikationen reicht es, nur die drei möglichen Situationen S_1, S_2 und S_3 zu betrachten. Wenn A gilt (Situationen S_1 und S_2), dann gilt auch B und deswegen gilt die Implikation $A \Rightarrow B$. In der Situation S_2 gilt A und C gilt nicht. Deswegen gilt die Implikation $A \Rightarrow C$ nicht. Wenn B gilt (in S_1 und S_2), gilt auch A und somit gilt die Implikation $B \Rightarrow A$. Die Implikation $B \Rightarrow C$ gilt nicht, weil in S_2 die Aussage B gilt, aber C gilt nicht. Die Gültigkeit von $C \Rightarrow B$ kannst du selbst begründen.

Die Tabelle 2.12 hat neun Zeilen und zwei Spalten. Könntest du dir eine kompaktere Darstellung, z. B. durch eine 3×3 Tabelle der Übersicht über geltende Implikationen überlegen?

Implikationen	Gültigkeit
$A \Rightarrow A$	gilt
$A \Rightarrow B$	gilt
$A \Rightarrow C$	gilt nicht
$B \Rightarrow A$	gilt
$B \Rightarrow B$	gilt
$B \Rightarrow C$	gilt nicht
$C \Rightarrow A$	gilt
$C \Rightarrow B$	gilt
$C \Rightarrow C$	gilt

Tabelle 2.12

Aufgabe 2.7
Wir sollen die folgende Implikation $A \Rightarrow Z$ beweisen:

„$a \bmod p = b \bmod p$ für geeignete a, b, $p \in \mathbb{Z}^+$“
$\Rightarrow$ „Es existiert eine ganze Zahl d, so dass $a = b + d \cdot p$ gilt.“

Wenn man die Aussage Z dieser Implikation $A \Rightarrow Z$ genauer betrachtet, sagt sie nichts anderes, als dass der Unterschied zwischen a und b ein Vielfaches von p ist. Dies erinnert uns an Beispiel 2.3, weil jedes Vielfache von p durch p teilbar ist. Deswegen nutzen wir in unserem Beweis das Resultat von Beispiel 2.3.

„$a \bmod p = b \bmod p$ für $a, b, p \in \mathbb{Z}^+$“

$\Rightarrow$ „p teilt $a - b$“
{Die in Beispiel 2.3 bewiesene Implikation}

$\Rightarrow$ „Es existiert ein $d \in \mathbb{Z}$, so dass $a - b = d \cdot p$“
{Definition der Teilbarkeit angewendet auf $a - b$ und p}

$\Rightarrow$ „Es existiert ein $d \in \mathbb{Z}$, so dass $a = b + d \cdot p$“
{Die Bedeutung einer Gleichung ändert sich nicht, wenn zu beiden Seiten die gleiche Zahl b addiert wird.}

Kannst du den Beweis führen, ohne die in Beispiel 2.3 bewiesene Aussage zu verwenden?

Aufgabe 2.9
Wir sollen die Implikation $A \Rightarrow Z$ beweisen, wobei:

A ist „d teilt die Zahlen a und b“
B ist „d teilt $ax+by$ für jedes x und jedes y aus $\mathbb{Z}$“.

Der direkte Beweis geht wie folgt:

„d teilt a und d teilt b“

$\Rightarrow$ „$a = k \cdot d$ und $b = l \cdot d$ für geeignete $k, l \in \mathbb{Z}$“

$\Rightarrow$ „Für alle $x, y \in \mathbb{Z}$ gilt: $ax+by = k \cdot d \cdot x + l \cdot d \cdot y = d \cdot (k \cdot x + l \cdot y)$“
{Einsetzen von $k \cdot d$ für a und $l \cdot d$ für b und Distributivgesetz}

$\Rightarrow$ „d teilt $d \cdot (k \cdot x + l \cdot y) = ax+by$ für alle $x, y \in \mathbb{Z}$“
{Definition der Teilbarkeit}

Aufgabe 2.13
Die Zahl a ist ein Teiler von a (d.h. $a \in \text{Teiler}_a$) sowie von $k \cdot a$ (d.h. $a \in \text{Teiler}_{k \cdot a}$). Somit gilt:

$$a \in \text{Teiler}_{a,k \cdot a} = \text{Teiler}_a \cap \text{Teiler}_{k \cdot a}.$$

Weil offensichtlich a der größte Teiler von a ist, gilt:

$$a = \text{maximum } \{c \mid c \in \text{Teiler}_{a,k \cdot a}\}$$

und somit gilt $a = \text{GGT}(a, k \cdot a)$.

Aufgabe 2.20
Die Wahrheitstabelle ist in Tab. 2.13 auf der nächsten Seite dargestellt. Als Unterschied zu Tabelle 2.7 sehen wir, dass zwei Situationen S_7 und S_8 möglich geblieben sind. Über C können wir hier nichts aussagen, weil C in S_7 gilt und in S_8 nicht gilt. In beiden Situationen gelten A und B nicht. Für B haben wir dies vorausgesetzt, deswegen konnte es auch nicht anders ausfallen. Damit ist die einzige neue Erkenntnis, dass A nicht gilt. Dies entspricht aber unseren Erwartungen. Wir haben die Gültigkeit von $A \Rightarrow B$ und von $\overline{B}$ belegt und somit folgt aus dem Schema des indirekten Beweises die Gültigkeit von $\overline{A}$.

	A	B	C	$A \Rightarrow B$	$B \Rightarrow C$	B gilt nicht
S_1	1	1	1			ausg.
S_2	1	1	0		ausg.	ausg.
S_3	1	0	1	ausg.		
S_4	1	0	0	ausg.		
S_5	0	1	1			ausg.
S_6	0	1	0		ausg.	ausg.
S_7	0	0	1			
S_8	0	0	0			

Tabelle 2.13

Aufgabe 2.24
Wir wissen, dass x^2 nicht durch 3 teilbar ist und sollen die Zielaussage Z „x ist nicht durch 3 teilbar“ beweisen. Nach dem Schema des indirekten Beweises fangen wir mit der Behauptung $\overline{Z}$ an:

„x ist durch 3 teilbar“

$\Rightarrow$ „$x = 3i$ für ein $i \in \mathbb{Z}$“
{Definition der Teilbarkeit}

$\Rightarrow$ „$x^2 = (3i)^2 = 9i^2 = 3 \cdot (3i^2)$“

$\Rightarrow$ „x^2 ist durch 3 teilbar“
{Definition der Teilbarkeit}

Das Resultat der Folge von Implikationen ist das Gegenteil unserer Voraussetzung „x^2 ist nicht durch 3 teilbar“. Somit kann die Startbehauptung $\overline{Z}$ nicht gelten. Wir können schließen, dass Z gilt.

Aufgabe 2.26
Diese Behauptung gilt. Wir beweisen es sogar mittels eines direkten Beweises.

„x^2 ist durch 6 teilbar“

$\Rightarrow$ „x^2 ist durch 2 und durch 3 teilbar“

$\Rightarrow$ „x ist durch 2 teilbar und x ist durch 3 teilbar“

{Aus Aufgabe 2.23 wissen wir, dass x gerade sein muss, wenn x^2 gerade ist. Im Sinne des direkten Beweises haben wir es auch in Beispiel 2.4 gezeigt. In Beispiel 2.5 haben wir bewiesen, dass x durch 3 teilbar sein muss, wenn 3 die Zahl x^2 teilt.}

$\Rightarrow$ „x ist durch 6 teilbar“
{Eine Zahl ist durch 6 genau dann teilbar, wenn sie durch 2 und durch 3 teilbar ist.}

Betrachten wir jetzt die Implikation

„Wenn x^2 durch 12 teilbar ist, dann ist auch x durch 12 teilbar.“

Diese Implikation gilt nicht für alle x. Wählen wir $x = 6$. Dann ist $x^2 = 36$ und wir sehen, dass 12 die Zahl 36 teilt. Die Zahl $x = 6$ ist aber nicht durch 12 teilbar.

Kannst du noch mehrere Zahlen x finden, für welche die Implikation nicht gilt? Impliziert die Teilbarkeit von x^2 durch 18 die Teilbarkeit von x durch 18?

Aufgabe 2.27
Wir zeigen es für $a+b$ und $\frac{a}{b}$. Fangen wir mit $a+b$ an.

„a und b sind rationale Zahlen“

$\Rightarrow$ „$a = \frac{p}{q}$ und $b = \frac{r}{s}$ für $p, q, r, s \in \mathbb{Z}$ und $q \neq 0$, $s \neq 0$“
{Definition der rationalen Zahlen}

$\Rightarrow$ „$a+b = \frac{p}{q} + \frac{r}{s} = \frac{ps+rq}{qs}$ mit $q \neq 0$, $s \neq 0$“

$\Rightarrow$ „$a+b$ ist eine rationale Zahl“
{Die Zahl $ps+rq$ ist in $\mathbb{Z}$, weil alle Zahlen p, q, r und s auch aus $\mathbb{Z}$ sind. Die Zahl qs ist auch aus $\mathbb{Z}$ und $qs \neq 0$ gilt, weil $q \neq 0$ und $s \neq 0$ gelten. Somit ist $a+b = \frac{m}{n}$ für $m = ps+rq \in \mathbb{Z}$ und $n = qs \in \mathbb{Z} \setminus \{0\}$ und damit eine rationale Zahl.}

Der Beweis der Rationalität von $\frac{a}{b}$ für $b \neq 0$ ist noch einfacher.

„a und b sind rationale Zahlen und $b \neq 0$“

$\Rightarrow$ „$a = \frac{p}{q}$ und $b = \frac{r}{s}$ für $p, q, r, s \in \mathbb{Z}$ und $q \neq 0$, $r \neq 0$, $s \neq 0$“
{Definition der rationalen Zahlen}

$\Rightarrow$ „$\frac{a}{b} = a \cdot \frac{1}{b} = \frac{p}{q} \cdot \frac{s}{r} = \frac{ps}{qr}$ und $q \neq 0$, $r \neq 0$, $s \neq 0$“

$\Rightarrow$ „$\frac{a}{b}$ ist eine rationale Zahl“
{Die Zahl $p \cdot s$ sowie die Zahl $q \cdot r$ sind rationale Zahlen und $qr \neq 0$, weil $q \neq 0$ und $r \neq 0$ gelten.}

Aufgabe 2.39
Wir führen einen indirekten Beweis der Aussage, dass es unendlich viele Primzahlen gibt.

„Es gibt endlich viele Primzahlen $p_1, p_2, \ldots, p_k$.“

$\Rightarrow$ „Jede Zahl aus $\mathbb{Z}^+$ außer die 1 ist durch mindestens eine der Primzahlen $p_1, p_2, \ldots, p_k$ teilbar.“
{Jede positive ganze Zahl größer als 1 lässt sich faktorisieren (als Produkt von Primzahlen darstellen).}

$\Rightarrow$ „Die Zahl $n = p_1 \cdot p_2 \cdot \cdots \cdot p_k + 1$ ist durch eine der Primzahlen $p_1, p_2, \ldots, p_k$ teilbar.“
{Wenn es für alle natürlichen Zahlen größer gleich 2 gilt, muss es auch für diese spezielle Zahl n gelten.}

Die letzte Aussage ist widersprüchlich. Egal, durch welche der Primzahlen $p_1, p_2, \ldots, p_k$ wir n teilen, der Rest ist immer 1. Zum Beispiel:

$(p_1 \cdot p_2 \cdot \cdots \cdot p_k + 1)$ div $p_1 = p_2 \cdot \cdots \cdot p_k$ und
$(p_1 \cdot p_2 \cdot \cdots \cdot p_k + 1)$ mod $p_1 = 1$.

Somit teilt keine der Primzahlen $p_1, p_2, \ldots, p_k$ die Zahl n.

Ist die Formel $n = p_1 \cdot p_2 \cdot \cdots \cdot p_k + 1$ eine gute Strategie, um aus den bekannten (k kleinsten) Primzahlen $p_1, p_2, \ldots, p_k$ eine neue Primzahl n zu erzeugen? Oder ist es möglich, dass n für gewisse $p_1, p_2, \ldots, p_k$ keine Primzahl ist?

Lektion 3

Geschichte der Informatik

Hinweis für die Lehrperson Das Ziel dieser Lektion ist nicht, Kompetenzen in Verwendung gewisser Methoden oder im Umgang mit gewissen Objekten zu erwerben. Sie bietet eine Vorschau auf unterschiedliche Themenbereiche der Informatik. Nicht alle hier diskutierten Themen müssen angesprochen werden. Wichtig und notwendig ist, nur die geschichtliche Entwicklung der Begriffsbildung in der Informatik zu vermitteln und eine erste Vorstellung von der Bedeutung der Grundbegriffe „Algorithmus“ und „Komplexität“ zu erwerben.

Ende des 19. und Anfang des 20. Jahrhunderts war die Gesellschaft in einem Zustand der Euphorie angesichts der Erfolge der Wissenschaft und der technischen Revolution, die das Wissen in die Herstellung von Maschinen umgewandelt hatte. Die Produkte der kreativen Arbeit von Wissenschaftlern und Entwicklern drangen in das tägliche Leben und erhöhten die Lebensqualität wesentlich. Unvorstellbares wurde zur Realität. Die entstandene Begeisterung führte unter den Wissenschaftlern nicht nur zu großem Optimismus, sondern sogar zu utopischen Vorstellungen über unsere Fähigkeiten. Es überwog die kausal-deterministische Vorstellung über die Welt, in der alles, was passiert, eine Ursache hat. Mit der Kette

$$\text{Ursache} \Rightarrow \text{Wirkung} \Rightarrow \text{Ursache} \Rightarrow \text{Wirkung} \Rightarrow \ldots$$

wollte man die Welt erklären. Man glaubte daran, dass wir fähig sind, alle Naturgesetze zu erforschen und dass dieses Wissen ausreicht, um die Welt zu verstehen. In der Physik zeigte sich diese Euphorie in dem Gedankenexperiment der so genannten Dämonen, die die Zukunft berechnen und somit vorhersagen könnten. Den Physikern war klar, dass das Universum aus einer riesigen Menge von Teilchen besteht und kein Mensch fähig ist,

J. Hromkovič, *Lehrbuch Informatik*, DOI 10.1007/978-3-8348-9692-6_18,

auf einmal alle ihre Positionen, inneren Zustände und Bewegungsrichtungen zu erfassen. Somit sahen die Physiker, dass auch mit der Kenntnis aller Naturgesetze ein Mensch nicht die Zukunft vorhersagen kann. Deswegen „führten“ die Physiker den Begriff des Dämonen als den eines Übermenschen ein, der den Ist-Zustand des Universums (den Zustand aller Teilchen und aller Interaktionen zwischen den Teilchen) vollständig sehen kann. Damit wurde der hypothetische Dämon fähig, mit der Kenntnis aller Naturgesetze die Zukunft zu kalkulieren und alles über sie vorherzusagen. Ich persönlich halte aber diese Vorstellung gar nicht für optimistisch, weil sie bedeutet, dass die Zukunft schon bestimmt ist. Wo bleibt dann Platz für unsere Aktivitäten? Können wir gar nichts beeinflussen, höchstens vorhersagen? Zum Glück hat die Physik selbst diese Vorstellungen zerschlagen. Einerseits stellte man mit der Chaostheorie fest, dass es reale Systeme gibt, bei denen unmessbar kleine Unterschiede in der Ausgangslage zu vollständig unterschiedlichen, zukünftigen Entwicklungen führen. Das definitive Aus für die Existenz von Dämonen war die Entwicklung der Quantenmechanik, die zur eigentlichen Grundlage der heutigen Physik geworden ist. Die Basis der Quantenmechanik sind wirklich zufällige und damit unvorhersehbare Ereignisse auf der Ebene der Teilchen. Wenn man die Quantenmechanik akzeptiert (bisher waren die Resultate aller Experimente im Einklang mit dieser Theorie), dann gibt es keine eindeutig bestimmte Zukunft und damit wird uns der Spielraum für die Zukunftsgestaltung nicht entzogen.

Die Gründung der Informatik hängt aber mit anderen, aus heutiger Sicht „utopischen“ Vorstellungen zusammen. David Hilbert, einer der berühmtesten Mathematiker seiner Zeit, glaubte an die Existenz von **Lösungsmethoden** für alle Probleme. Die Vorstellung war,

(i) dass man die ganze Mathematik auf endlich vielen Axiomen aufbauen kann,

(ii) dass die so aufgebaute Mathematik in dem Sinne vollständig wird, dass alle in dieser Mathematik formulierbaren Aussagen auch in dieser Theorie als korrekt oder falsch bewiesen werden können und

(iii) dass zum Beweisen der Korrektheit von Aussagen eine Methode existiert.

Im Zentrum unseres Interesses liegt jetzt der Begriff **Methode**. Was verstand man damals in der Mathematik unter einer Methode?

> *Eine **Methode** zur Lösung einer Aufgabe ist eine Beschreibung einer Vorgehensweise, die zur Lösung der Aufgabe führt. Die Beschreibung besteht aus*

> *einer Folge von Instruktionen, die für jeden, auch einen Nichtmathematiker durchführbar sind.*

Wichtig ist dabei zu begreifen, dass man zur Anwendung einer Methode nicht zu verstehen braucht, wie diese Methode erfunden wurde und warum sie die gegebene Aufgabe löst. Zum Beispiel betrachten wir die Aufgabe (das Problem), quadratische Gleichungen der Form

$$x^2 + bx + c = 0$$

zu lösen. Wenn $b^2 - 4c > 0$ gilt, beschreiben die Formeln

$$x_1 = -\left(\frac{b}{2}\right) + \frac{\sqrt{b^2 - 4c}}{2}$$
$$x_2 = -\left(\frac{b}{2}\right) - \frac{\sqrt{b^2 - 4c}}{2}$$

die zwei Lösungen der quadratischen Gleichung. Wir sehen damit, dass man x_1 und x_2 berechnen kann, ohne zu wissen, warum die Formeln so sind, wie sie sind. Es reicht aus, einfach fähig zu sein, die arithmetischen Operationen durchzuführen. Somit kann auch ein maschineller Rechner, also ein Gegenstand ohne Intellekt, quadratische Gleichungen dank der existierenden Methode lösen.

Deswegen verbindet man die Existenz einer mathematischen Methode zur Lösung gewisser Aufgabentypen mit der **Automatisierung** der Lösung dieser Aufgaben. Heute benutzen wir nicht den Begriff „Methode“ zur Beschreibung von Lösungswegen, weil dieses Fachwort viele Interpretationen in anderen Kontexten hat. Stattdessen verwenden wir heute den zentralen Begriff der Informatik, den Begriff des **Algorithmus**. Obwohl die Verwendung dieses Begriffes relativ neu ist, verwenden wir Algorithmen im Sinne von Lösungsmethoden schon seit Tausenden von Jahren. Das Wort „Algorithmus“ verdankt seinen Namen dem arabischen Mathematiker Al-Khwarizmi, der im 9. Jahrhundert in Bagdad ein Buch über algebraische Methoden geschrieben hat.

Im Sinne dieser algorithmischen Interpretation strebte also Hilbert die Automatisierung der Arbeit von Mathematikern an. Er strebte nach einer vollständigen Mathematik, in der man für die Erzeugung der Korrektheitsbeweise von formulierten Aussagen einen Algorithmus (eine Methode) hat. Damit wäre die Haupttätigkeit eines Mathematikers, mathematische Beweise zu führen, automatisierbar. Eigentlich eine traurige Vorstellung, eine so hoch angesehene, intellektuelle Tätigkeit durch „dumme“ Maschinen erledigen zu können.

Im Jahr 1931 setzte Kurt Gödel diesen Bemühungen, eine vollständige Mathematik zu bauen, ein definitives Ende. Er hat mathematisch bewiesen, dass eine vollständige Mathematik nach Hilbert'schen Vorstellungen nicht existiert und somit nie aufgebaut werden kann. Ohne auf mathematische Formulierungen zurückzugreifen, präsentieren wir die wichtigste Aussage von Gödel für die Wissenschaft:

(i) Es gibt keine vollständige „vernünftige" mathematische Theorie. In jeder korrekten und genügend umfangreichen mathematischen Theorie (wie der heutigen Mathematik) ist es möglich, Aussagen zu formulieren, deren Korrektheit innerhalb dieser Theorie nicht beweisbar ist. Um die Korrektheit dieser Aussagen zu beweisen, muss man neue Axiome aufnehmen und dadurch eine größere Theorie aufbauen.

(ii) Es gibt keine Methode (keinen Algorithmus) zum automatischen Beweisen mathematischer Sätze.

Wenn man die Resultate richtig interpretiert, ist diese Nachricht eigentlich positiv. Der Aufbau der Mathematik als die formale Sprache der Wissenschaft ist ein unendlicher Prozess. Mit jedem neuen Axiom und damit mit jeder neuen Begriffsbildung wächst unser Vokabular und unsere Argumentationsstärke. Dank neuer Axiome und damit verbundener Begriffe können wir über Dinge und Ereignisse Aussagen formulieren, über die wir vorher nicht sprechen konnten. Und wir können die Wahrheit von Aussagen überprüfen, die vorher nicht verifizierbar waren. Letztendlich können wir diese Wahrheitsüberprüfung nicht automatisieren.

Die Resultate von Gödel haben unsere Sicht auf die Wissenschaft geändert. Wir verstehen dadurch die Entwicklung der einzelnen Wissenschaften zunehmend als einen Prozess der Begriffsbildung und Methodenentwicklung. Warum war aber das Resultat von Gödel maßgeblich für das Entstehen der Informatik? Einfach deswegen, weil vor den Gödelschen Entdeckungen kein Bedarf an einer formalen, mathematischen Definition des Begriffes Methode vorhanden war. Eine solche Definition brauchte man nicht, um eine neue Methode für gewisse Zwecke zu präsentieren. Die intuitive Vorstellung einer einfachen und verständlichen Beschreibung der Lösungswege reichte vollständig. Aber sobald man beweisen sollte, dass für gewisse Aufgaben (Zwecke) kein Algorithmus existiert, musste man vorher ganz genau wissen, was ein Algorithmus ist. Die Nichtexistenz eines Objektes zu beweisen, das nicht eindeutig spezifiziert ist, ist ein unmögliches Vorhaben. Wir müssen ganz genau (im Sinne einer mathematischen Definition) wissen, was ein Algorithmus zur Lösung eines Problems ist. Nur so können wir den Beweis führen, dass es zur Lösung dieser Aufgabe keinen Algorithmus gibt. Die erste mathematische Definition wurde von

Alan Turing in Form der sogenannten Turingmaschine gegeben und später folgten viele weitere. Das Wichtigste ist, dass alle vernünftigen Versuche, eine formale Definition des Algorithmus zu finden, zu der gleichen Begriffsbeschreibung im Sinne des automatisch Lösbaren führten. Obwohl sie in mathematischen Formalismen auf unterschiedliche Weise ausgedrückt wurden, blieben die diesen Definitionen entsprechenden Mengen der algorithmisch lösbaren Aufgaben immer dieselben. Dies führte letztendlich dazu, dass man die Turingsche Definition des Algorithmus zum ersten[1] Axiom der Informatik erklärt hat.

Jetzt können wir unser Verständnis für die Axiome nochmals überprüfen. Wir fassen die Definition des Algorithmus als Axiom auf, weil ihre Korrektheit nicht beweisbar ist. Wie könnten wir beweisen, dass die von uns definierte, algorithmische Lösbarkeit wirklich unserer Vorstellung über automatisierte Lösbarkeit entspricht? Wir können eine Widerlegung dieser Axiome nicht ausschließen. Wenn jemand eine nutzbare Methode zu einem gewissen Zweck entwickelt und diese Methode nach unserer Definition kein Algorithmus ist, dann war unsere Definition nicht gut genug und muss revidiert werden. Seit 1936 hat aber, trotz vieler Versuche, niemand die verwendete Definition des Algorithmus destabilisiert und somit ist der Glaube an die Gültigkeit dieser Axiome stark.

Der Begriff des Algorithmus ist so zentral für die Informatik, dass wir jetzt nicht versuchen werden, die Bedeutung dieses Begriffes in Kürze und unvollständig zu erklären. Lieber widmen wir eine ganze Unterrichtslektion dem Aufbau des Verständnisses für die Begriffe „Algorithmus“ und „Programm“.

Die erste fundamentale Frage der Informatik war:

> *Gibt es Aufgaben, die man algorithmisch (automatisch) nicht lösen kann?*
> *Und wenn ja, welche Aufgaben sind algorithmisch lösbar und welche nicht?*

Wir werden diese grundlegende Frage in einem selbständigen Unterrichtsmodul nicht nur beantworten, sondern große Teile der Forschungswege zu den richtigen Antworten so darstellen, dass man sie danach selbstständig nachvollziehen kann. Weil dieses Thema zu den schwierigsten in den ersten beiden Jahren des universitären Informatikstudiums gehört, gehen wir hier in sehr kleinen Schritten vor. Der Schlüssel zum Verständnis der Informatik liegt im korrekten Verstehen ihrer Grundbegriffe. Deswegen ist die nächste Lektion vollständig der Bildung und Erklärung der Schlüsselbegriffe „Algorithmus“ und „Programm“ gewidmet. Um eine erste Vorstellung der Bedeutung dieser Begriffe aufzubauen, fangen wir mit dem alltäglichen Kuchenbacken an.

[1] Alle Axiome der Mathematik werden auch als Axiome in der Informatik verwendet.

Hast du schon einmal nach Rezept einen Kuchen gebacken oder ein Essen gekocht, ohne zu ahnen, warum man genau so vorgehen muss, wie in der Anweisung beschrieben? Die ganze Zeit warst du dir bewusst, dass eine korrekte Durchführung aller Einzelschritte enorm wichtig für die Qualität des Endprodukts ist. Was hast du dabei gelernt? Bei einem präzise formulierten und detaillierten Rezept kannst du etwas Gutes erzeugen, ohne ein Meisterkoch zu sein. Auch wenn man sich im Rausch des Erfolges kurz für einen hervorragenden Koch halten darf, ist man dies nicht, bevor man nicht alle Zusammenhänge zwischen dem Produkt und den Schritten seiner Herstellung verstanden hat – und selbst solche Rezepte schreiben kann.

Der Rechner hat es noch schwerer: Er kann nur ein paar elementare Rechenschritte durchführen, so wie man zum Beispiel die elementaren Koch-Operationen wie das Mischen von Zutaten und das Erwärmen zur Umsetzung eines Rezeptes beherrschen muss. Im Unterschied zu uns besitzt der Rechner aber keine Intelligenz und kann deshalb auch nicht improvisieren. Ein Rechner verfolgt konsequent die Anweisungen seiner Rezepte – seiner Programme, ohne zu ahnen, welche komplexe Informationsverarbeitung diese auslösen.

Auf diese Weise entdecken wir, dass die Kunst des Programmierens jene ist, Programme wie Rezepte zu schreiben, welche die Methoden und Algorithmen für den Rechner verständlich darstellen. So kann er unterschiedlichste Aufgaben lösen. Dabei stellen wir auch den Rechner vor und zeigen, welche Befehle (Instruktionen) er ausführen kann und was dabei in ihm passiert. Nebenbei lernen wir auch, was algorithmische Aufgaben (Probleme) sind und wo genau der Unterschied zwischen Programmen und Algorithmen liegt.

Nachdem die Forscher eine Theorie zur Klassifizierung von Problemen in automatisch lösbare und automatisch unlösbare erfolgreich entwickelt hatten, kamen in den sechziger Jahren die Rechner zunehmend in der Industrie zum Einsatz. In der praktischen Umsetzung von Algorithmen ging es dann nicht mehr nur um die Existenz von Algorithmen, sondern auch um deren Komplexität und somit um die Effizienz der Berechnung.

Nach dem Begriff des Algorithmus ist der Begriff der **Komplexität** der nächste zentrale Begriff der Informatik. Die Komplexität verstehen wir in erster Linie als Berechnungskomplexität, also als die Menge der Arbeit, die ein Rechner bewältigen muss, um zu einer Lösung zu gelangen. Am häufigsten messen wir die Komplexität eines Algorithmus in

der Anzahl der durchgeführten Operationen oder der Größe des verwendeten Speichers. Wir versuchen auch, die Komplexität von Problemen zu messen, indem wir die Komplexität des besten (schnellsten bzw. mit dem Speicher am sparsamsten umgehenden) Algorithmus, der das gegebene Problem löst, heranziehen.

Die Komplexitätstheorie versucht die Probleme (Aufgabenstellungen) bezüglich der Komplexität in leichte und schwere zu unterteilen. Wir wissen, dass es beliebig schwere algorithmisch lösbare Probleme gibt, und wir kennen Tausende von Aufgaben aus der Praxis, für deren Lösung die besten Algorithmen mehr Operationen durchführen müssten, als es Protonen im bekannten Universum gibt. Weder reicht die ganze Energie des Universums noch die Zeit seit dem Urknall aus, um sie zu lösen. Kann man da überhaupt etwas unternehmen?

Beim Versuch, diese Frage zu beantworten, entstanden einige der tiefgründigsten Beiträge der Informatik. Man kann einiges tun. Und wie dies möglich ist, das ist die wahre Kunst der Algorithmik. Viele schwer berechenbare Probleme sind in folgendem Sinne instabil. Mit einer kleinen Umformulierung des zu lösenden Problems oder mit einer leichten Abschwächung der Anforderungen kann auf einmal aus einer physikalisch unrealisierbaren Menge an Computerarbeit, eine in Bruchteilen einer Sekunde durchführbare Rechnung werden. Wie dies durch die Kunst der Algorithmik gelingt, wird das Thema mehrerer Module sein.

Unerwartete und spektakuläre Lösungen entstehen dann, wenn unsere Anforderungen so wenig abgeschwächt werden, dass es aus Sicht der Praxis keine wirkliche Abschwächung ist und dabei trotzdem eine riesige Menge von Rechenarbeit eingespart wird.

Die wunderbarsten Beispiele in diesem Zusammenhang entstehen bei der Anwendung der Zufallssteuerung. Die Effekte sind hier so faszinierend wie wahre Wunder. Deswegen widmen wir dem Thema der zufallgesteuerten Algorithmen ein ganzes Unterrichtsmodul. Die Idee ist dabei, die deterministische Kontrolle von Algorithmen dadurch aufzugeben, dass man hier und da den Algorithmus eine Münze werfen lässt. Abhängig von dem Ergebnis des Münzwurfs darf dann der Algorithmus unterschiedliche Lösungsstrategien wählen. Auf diese Weise verlieren wir die theoretisch absolute Sicherheit, immer die korrekte Lösung auszurechnen, weil wir bei einigen Zufallsentscheidungen erfolglose Berechnungen nicht vermeiden können. Unter erfolglosen Berechnungen verstehen wir Bemühungen, die zu keinem oder sogar zu einem falschen Resultat führen. Wenn man aber die Wahrscheinlichkeit des Auftretens von fehlerhaften Problemlösungen kleiner hält als die Wahrscheinlichkeit des Auftretens eines Hardwarefehlers während der Be-

rechnung, verliert man dabei aus praktischer Sicht gar nichts. Wenn man mit diesem nur scheinbaren Sicherheitsverlust den Sprung von einer physikalisch unrealisierbaren Menge von Arbeit zu ein paar Sekunden Rechenzeit auf einem gewöhnlichen PC schafft, kann man von einem wahren Wunder sprechen. Ohne diese Art von Wundern kann man sich heute die Kommunikation im Internet, E-Commerce und Online-Banking gar nicht mehr vorstellen.

Außer den Anwendungen des Zufalls in der Informatik diskutieren wir in den entsprechenden Unterrichtsmodulen die fundamentale Frage der Existenz des echten Zufalls und zeigen, wie sich die Einstellung zum Zufall in der Geschichte der Wissenschaft gewandelt hat.

Die Konzepte der Berechnungskomplexität haben die ganze Wissenschaft beeinflusst und befruchtet. Ein schönes Beispiel solcher Bereicherung ist die Kryptographie, die „Wissenschaft der Verschlüsselung“. Die Kryptographie hat sich erst mit Hilfe der Algorithmik und ihren komplexitätstheoretischen Konzepten zu einer fundierten Wissenschaft entwickelt. Es ist schwer, andere Wissenschaftsgebiete zu finden, in denen so viele Wunder im Sinne unerwarteter Wendungen und unglaublicher Möglichkeiten auftreten.

Kryptographie ist eine uralte Wissenschaft der Geheimsprachen. Dabei geht es darum, Texte so zu verschlüsseln, dass sie niemand außer dem rechtmäßigen Empfänger dechiffrieren kann. Die klassische Kryptographie basiert auf geheimen Schlüsseln, die dem Sender sowie dem Empfänger bekannt sind.

Die Informatik hat wesentlich zur Entwicklung der Kryptographie beigetragen. Zunächst hat sie auf der Ebene der Begriffsbildung das erste Mal ermöglicht, die Zuverlässigkeit eines Kryptosystems zu messen. Ein Kryptosystem ist schwer zu knacken, wenn jedes Computerprogramm, das den geheimen Schlüssel nicht kennt, eine physikalisch unrealisierbare Menge von Arbeit zur Dechiffrierung von verschlüsselten Texten braucht. Ausgehend von dieser Definition der Güte eines Kryptosystems haben die Informatiker Chiffrierungen gefunden, die effizient durchführbar sind, deren entsprechende Dechiffrierungen ohne Kenntnis des Schlüssels aber einer algorithmisch schweren Aufgabe entsprechen.

Daran sieht man, dass die Existenz von schweren Problemen uns nicht nur die Grenzen aufzeigt, sondern auch sehr nützlich sein kann. So entwickelte Kryptosysteme nennt man Public-Key-Kryptosysteme, weil die Chiffrierungsmechanismen wie in einem Telefonbuch veröffentlicht werden dürfen. Denn das Geheimnis, das zu seiner effizienten

Dechiffrierung notwendig ist, ist nur dem Empfänger bekannt, und kein unbefugter Dritter kann die chiffrierten Nachrichten lesen.

Dieses Thema ist nicht nur eine Brücke zwischen der Mathematik und Informatik, sondern auch eine ungewöhnlich kurze Brücke zwischen Theorie und Praxis. Wir werden ihm deswegen auch ein ganzes Unterrichtsmodul widmen.

Mit der Entwicklung der Informations- und Kommunikationstechnologien (ICT) entstanden zahlreiche neue Konzepte und Forschungseinrichtungen, welche die ganze Wissenschaft und sogar das tägliche Leben verändert haben. Dank der Informatik wurde das Wissen der Mathematik zu einer Schlüsseltechnologie mit grenzenlosen Anwendungen. Es ist uns unmöglich, hier alle wesentlichen Beiträge der Informatik aus den letzten 30 Jahren aufzulisten. Sie reichen von der Hardware- und Softwareentwicklung über Algorithmik zur interdisziplinären Forschung in praktisch allen Wissenschaftsdisziplinen. Die Fortschritte in der Erforschung unserer Gensequenzen oder der Entwicklung der Gentechnologie wären ohne Informatik genauso undenkbar wie die automatische Spracherkennung, Simulationen von ökonomischen Modellen oder die Entwicklung diagnostischer Geräte in der Medizin. Für alle diese unzähligen Beiträge erwähnen wir nur zwei neuere Konzepte, welche die Fundamente der ganzen Wissenschaft berühren.

Diese Beiträge sprechen die Möglichkeiten einer enormen Miniaturisierung von Rechnern und damit eine wesentliche Beschleunigung ihrer Arbeit an, indem man die Durchführung der Berechnungen auf die Ebene von Molekülen oder Teilchen bringt. Das erste Konzept entdeckt die biochemischen Technologien, die man zur Lösung konkreter, schwerer Rechenprobleme einsetzen könnte. Die Idee ist, die Computerdaten durch DNA-Sequenzen darzustellen und dann mittels einiger chemischer Operationen auf diesen Sequenzen die Lösung zu finden.

Wenn man die Arbeit von Rechnern genauer unter die Lupe nimmt, stellt man fest, dass sie nichts anderes tun, als gewisse Texte in andere Texte umzuwandeln. Die Aufgabenstellung ist dem Rechner als eine Folge von Symbolen (z. B. Nullen und Einsen) gegeben, und die Ausgabe des Rechners ist wiederum ein Text in Form einer Folge von Buchstaben.

Kann die Natur so etwas nachahmen? Die DNA-Sequenzen kann man auch als Folge von Symbolen A, T, C und G sehen Wir wissen, dass die DNA-Sequenzen genau wie Rechnerdaten Informationsträger sind. Genau wie die Rechner Operationen auf den symbolischen Darstellungen der Daten ausführen können, ermöglichen es unterschiedliche chemische Prozesse, biologische Daten zu verändern. Was ein Rechner kann, schaffen die Moleküle locker – sogar noch ein bisschen schneller.

Die Informatiker haben bewiesen, dass genau das, was man algorithmisch mit Rechnern umsetzen kann, man auch in einem Labor durch chemische Operationen an DNA-Sequenzen realisieren kann. Die einzige Schwäche dieser Technologie ist, dass die Durchführung der chemischen Operationen auf DNA-Sequenzen als Datenträgern eine unvergleichbar höhere Fehlerwahrscheinlichkeit aufweist, als es bei der Durchführung von Rechneroperationen der Fall ist.

Dieser Forschungsbereich ist immer für Überraschungen gut. Heute wagt niemand, Prognosen über die möglichen Anwendungen dieses Ansatzes für die nächsten zehn Jahre zu machen.

Wahrscheinlich hat keine Wissenschaftsdisziplin unsere Weltanschauung so stark geprägt wie die Physik. Tiefe Erkenntnisse und pure Faszination verbinden wir mit der Physik. Das Juwel unter den Juwelen ist die Quantenmechanik. Die Bedeutung ihrer Entdeckung erträgt den Vergleich mit der Entdeckung des Feuers in der Urzeit. Die Faszination der Quantenmechanik liegt darin, dass die Gesetze des Verhaltens von Teilchen scheinbar unseren physikalischen Erfahrungen aus der „Makrowelt“ widersprechen. Die am Anfang umstrittene und heute akzeptierte Theorie ermöglicht zunächst hypothetisch eine neue Art von Rechnern auf der Ebene der Elementarteilchen. Hier spricht man vom *Quantenrechner*. Als man diese Möglichkeit entdeckt hatte, war die erste Frage, ob die Axiome der Informatik noch gelten. Mit anderen Worten: Können die Quantenalgorithmen etwas, was klassische Algorithmen nicht können? Die Antwort ist negativ und somit lösen die Quantenalgorithmen die gleiche Menge von Aufgaben wie die klassischen Algorithmen und unsere Axiome stehen noch stabiler und glaubwürdiger da. Was soll dann aber der Vorteil einer potenziellen Nutzung von Quantenrechnern sein? Wir können konkrete Aufgaben von großer, praktischer Bedeutung mit Quantenalgorithmen effizient lösen, während die besten bekannten, klassischen deterministischen sowie zufallsgesteuerten Algorithmen für diese Aufgaben eine unrealistische Menge von Computerarbeit erfordern. Damit ist die Quantenmechanik eine vielversprechende Rechnertechnologie. Das Problem ist nur, dass wir es noch nicht schaffen, anwendbare Quantenrechner zu bauen. Das Erreichen dieses Ziels ist eine große Herausforderung derzeitiger physikalischer Forschung.

Trotzdem bieten Quanteneffekte schon heute eine kommerzielle Umsetzung in der Kryptographie. Geboren aus der Mathematik in der Grundlagenforschung und aus der Elektrotechnik beim Bau der Rechner, sorgt die Informatik heute für den Transfer der Methoden der Mathematik in die technischen Wissenschaften und dadurch in das tägliche Leben.

Durch das Erzeugen eigener Begriffe und Konzepte bereichert sie zusätzlich auch die Mathematik in ihrer Grundlagenforschung. Die mit der Informatik entstandenen Informationstechnologien machen sie zum gleichwertigen Partner, nicht nur auf vielen naturwissenschaftlichen Gebieten der Grundlagenforschung, sondern auch in wissenschaftlichen Disziplinen wie Ökonomie, Soziologie, Didaktik oder Pädagogik, die sich lange gegen die Nutzung formaler Methoden und mathematischer Argumentation gewehrt haben.

Deswegen bieten die Spezialisierungen in der Informatik die Möglichkeit einer faszinierenden und interdisziplinären Grundlagenforschung sowie eine attraktive Arbeit in der Entwicklung von unterschiedlichen Systemen zur Speicherung und Bearbeitung von Informationen.

Zusammenfassung

Die Begriffsbildung ist maßgeblich für das Entstehen und die Entwicklung der wissenschaftlichen Disziplinen. Mit der Einführung des Begriffes Algorithmus wurde die Bedeutung des Begriffes Methode genau festgelegt (ein formaler Rahmen für die Beschreibung mathematischer Berechnungsverfahren wurde geschaffen) und damit die Informatik gegründet. Durch diese Festlegung konnte man mit klarer Bedeutung die Grenze zwischen automatisch (algorithmisch) Lösbarem und Unlösbarem untersuchen. Nachdem man viele Aufgaben bezüglich der algorithmischen Lösbarkeit erfolgreich klassifiziert hatte, kam der Begriff der Berechnungskomplexität auf, der die Grundlagenforschung in der Informatik bis heute bestimmt. Dieser Begriff ermöglicht es, die Grenze zwischen „praktischer“ Lösbarkeit und „praktischer“ Unlösbarkeit zu untersuchen. Er hat der Kryptographie eine Basis für den Begriff der Sicherheit und damit die Grundlage für die Entwicklung moderner Public-Key-Kryptosysteme gegeben und ermöglicht es, die Berechnungsstärke von Determinismus, Nichtdeterminismus, Zufallssteuerung und Quantenberechnungen im Vergleich zu studieren. Auf diese Weise trug und trägt die Informatik nicht nur zum Verständnis der allgemeinen wissenschaftlichen Kategorien wie

> *Determiniertheit, Nichtdeterminiertheit, Zufall, Information, Wahrheit, Unwahrheit, Komplexität, Sprache, Beweis, Wissen, Kommunikation, Algorithmus, Simulation usw.*

bei, sondern gibt mehreren dieser Kategorien einen neuen Inhalt und damit eine neue Bedeutung. Die spektakulärsten Ergebnisse der Informatik sind meistens mit dem Versuch verbunden, schwere Aufgabenstellungen zu lösen.

Kontrollfragen

1. Warum entstand ein Bedarf, den Begriff der Methode genau zu definieren?

2. Was strebte David Hilbert an, und woran haben viele Mathematiker am Anfang des zwanzigsten Jahrhunderts geglaubt?

3. Was war die wichtigste Entdeckung von Kurt Gödel?

4. Was verstand man unter den „Dämonen“ in der Physik?

5. Was misst man mittels der Berechnungskomplexität?

6. Gibt es algorithmisch lösbare Aufgaben, die man praktisch nicht lösen kann?

7. Wo gibt es gemeinsame Interessen zwischen Informatik und Biologie?

8. In welchem Bereich treffen sich Physik und Informatik?

9. Was ist die Ausdrucksstärke einer mathematischen Theorie? Was ist die Beweisstärke einer mathematischen Theorie? Kann man sie vergleichen?

10. Welches sind die fundamentalen Begriffe der Informatik? Gib Beispiele für den Einfluss der Begriffsbildung in der Informatik auf andere Wissenschaften.

Lektion 4

Algorithmisches Kuchenbacken

Das Hauptziel dieser Lektion ist es ein besseres Verständnis für den fundamentalsten Begriff der Informatik zu gewinnen. Dabei begreifen wir, warum die Frauen und Mädchen mindestens eben so gute Programmiererinnen und Algorithmikerinnen sind wie die Männer, und warum die erste Person, die programmiert hat, eine Dame war. In den vorherigen Lektionen haben wir schon eine gewisse Vorstellung davon gewonnen, was man unter einem Algorithmus oder einer Methode verstehen kann. Wir könnten sagen:

> *Ein Algorithmus ist eine gut verständliche Tätigkeitsbeschreibung, die uns zu unserem Ziel führt.*

Also gibt uns ein Algorithmus (eine Methode) einfache und eindeutige Hinweise, wie wir Schritt für Schritt vorgehen sollen, um das zu erreichen, was wir anstreben.

Das ist ähnlich wie bei einem Kochrezept. Dieses sagt uns ganz genau, was in welcher Reihenfolge zu tun ist und dementsprechend führen wir Schritt für Schritt die beschriebenen Tätigkeiten aus. Und dabei geht es genau um die Tugenden der besseren Hälfte der Menschheit. Um Erfolg zu haben, muss man sehr systematisch, mit vollem Verständnis und echtem Sinn fürs Detail vorgehen. Beim Umsetzen von Algorithmen gibt es keinen Platz für chaotisches Probieren oder irgendeine Art von Mehrdeutigkeit.

> *Inwiefern dürfen wir also ein Rezept für einen Algorithmus halten?*

Diese Frage direkt zu beantworten, ist nicht so einfach. Aber bei der Suche nach einer Antwort lernen wir besser zu verstehen, was sich wirklich hinter diesem Wort versteckt.

J. Hromkovič, *Lehrbuch Informatik*, DOI 10.1007/978-3-8348-9692-6_19,

Nehmen wir das Rezept für einen Aprikosenkuchen von 26 cm Durchmesser.

Zutaten:

3	Eiweiß
1	Prise Salz
6	Esslöffel heißes Wasser
100 g	Zuckerrohrgranulat (Rohrzucker)
3	Eigelb
1	Teelöffel abgeriebene Zitronenschale
150 g	Buchweizen fein gemahlen (Mehl)
1/2	Teelöffel Backpulver
400 g	Aprikosen, halbreif und enthäutet
10 g	Wildpfeilwurzelmehl

Rezept:

1. Ein Pergamentpapier in die Springform einspannen.
2. Den Backofen auf 180°C vorheizen.
3. 6 Esslöffel Wasser erwärmen.
4. Die drei Eiweiße mit einer Prise Salz und dem heißen Wasser zu steifem Schnee schlagen.
5. 100g Zuckerrohrgranulat und die Eigelbe nach und nach unterrühren. Danach solange rühren, bis eine feste, cremige Masse entstanden ist.
6. 1 Teelöffel abgeriebene Zitronenschale dazugeben und vermischen.

7. 150g Mehl mit 1/2 Teelöffel Backpulver vermischen, auf die Schaummasse geben und mit dem Schneebesen vorsichtig unterheben.

8. Die entstandene Masse in die Form füllen.

9. Die halbreifen und enthäuteten Aprikosen dekorativ auf den Teig setzen.

10. Das Biskuit im Backofen unter 160 °C Umluft 25–30 Minuten hellbraun backen.

11. Danach den Kuchen aus dem Backofen nehmen und abkühlen lassen.

12. Den fertigen ausgekühlten Kuchen nach Belieben mit Wildpfeilwurzelmehl bestäuben.

Das Rezept liegt vor und die Frage ist, ob wirklich jeder nach Rezept diesen Kuchen backen kann. Die Antwort ist wahrscheinlich, dass der Erfolg doch zu einem gewissem Grad von den Kenntnissen des Zubereitenden abhängt.

Jetzt ist es an der Zeit, die erste Anforderung an Algorithmen zu formulieren.

> *Die Algorithmen müssen eine so genaue Beschreibung der bevorstehenden Tätigkeit bieten, dass man diese Tätigkeit erfolgreich durchführen kann, auch wenn man keine Ahnung hat, warum die Umsetzung des Algorithmus zum gegebenen Ziel führt. Dabei muss die Beschreibung so eindeutig sein, dass unterschiedliche Interpretationen der Hinweise (Befehle) des Algorithmus ausgeschlossen sind. Egal, wer den Algorithmus auf seiner Eingabe anwendet, die entstehende Tätigkeit und damit auch das Resultat müssen gleich sein, d. h. jeder Anwender des Algorithmus muss zu demselben Ergebnis gelangen.*

Jetzt könnten wir eine lange Diskussion darüber anfangen, welche der 12 Schritte (Anweisungen) des Rezeptes eindeutige und für jeden verständliche Hinweise geben. Zum Beispiel:

- Was bedeutet **zu steifem Schnee schlagen** (Schritt 4)?

- Was bedeutet **vorsichtig unterheben** (Schritt 7)?

- Was bedeutet **dekorativ** auf den Teig setzen (Schritt 9)?

- Was bedeutet **hellbraun** backen (Schritt 10)?

- Was bedeutet **nach Belieben** bestäuben (Schritt 12)?

Eine erfahrene Köchin würde sagen: „Alles ist klar, genauer kann man es nicht angeben." Jemand, der das erste Mal in seinem Leben einen Kuchen backen will, könnte noch mehr Rat und Hilfe brauchen, bis er sich an die Arbeit traut. Und dabei ist unser Rezept einfacher formuliert als in den meisten Kochbüchern. Was denkst du z. B. über Anweisungen wie:

> Die **leicht abgekühlte** Gelatinemasse **zügig** unter die Quarkmasse geben und **gut** durchrühren.

Wir dürfen natürlich nicht zulassen, dass nur Erfahrene das Rezept für einen Algorithmus halten und der Rest der Welt nicht. Man muss eine Möglichkeit suchen, eine Einigung zu erzielen. Wir verstehen schon, dass ein Algorithmus eine Folge von Anweisungen ist, wobei jede angegebene Tätigkeit von jeder Person korrekt durchführbar sein muss. Dies bedeutet:

> *Man muss sich zuerst auf eine Liste der Tätigkeiten (Operationen) einigen, die jede oder jeder der koch- und backwilligen Menschen mit Sicherheit beherrscht.*

So eine Liste kann zuerst z. B. folgende Tätigkeiten enthalten, die möglicherweise sogar ein für diese Zwecke gebauter Roboter ohne jedes Verständnis der Kochkunst und ohne jede Improvisationsfähigkeit realisieren kann.

- `Gib x Löffel Wasser in ein Gefäß.`

- `Trenne ein Ei in Eiweiß und Eigelb.`

- `Heize den Ofen auf x Grad vor mit einer angegebenen Temperatur von x Grad.`

- `Backe y Minuten mit x Grad.`

- `Wiege x g Mehl ab und gib es in eine Schüssel.`

- `Gieße x l Milch in eine Kanne.`

- `Koche y Minuten.`
- `Mische mit dem Schneebesen x Minuten.`
- `Rühre mit einer Gabel x Minuten.`
- `Fülle eine Form mit einem Teig.`
- `Schäle x kg Kartoffeln.`
- `Mische den Inhalt zweier Gefäße.`

Sicherlich fallen dir viele weitere Tätigkeiten ein, die du für so einfach hälst, dass du sie jedem, der backen will, ohne weitere Erklärung zutraust. Im Folgenden geht es darum, ein Rezept so umzuschreiben, dass dabei die Befehle (Anweisungen) nur ausgewählte, einfache Basistätigkeiten verwenden.

Aufgabe 4.1 Versucht euch in 4- bis 5-köpfigen Gruppen auf eine Liste von Tätigkeiten zu einigen, die eurer Meinung nach jede und jeder aus der Klasse in der Küche beherrschen müsste. Überprüft eure Wahl, indem ihr eure Liste dem Rest der Klasse vorstellt und nachfragt, ob jede Person damit zurecht kommen würde. Überprüft eure Liste auf Vollständigkeit. Jede gewöhnliche Kochtätigkeit müsste durch eine Folge der Tätigkeiten aus der Liste darstellbar sein.

Versuchen wir jetzt, den Schritt 4 des Rezeptes in eine Folge einfacher Anweisungen umzuschreiben:

```
4.1 Gib die drei Eiweiße in das Gefäß G.

4.2 Gib 1g Salz in das Gefäß G.

4.3 Gib 6 Löffel Wasser in den Topf T.

4.4 Erwärme das Wasser im Topf T auf 60°C.

4.5 Gieße das Wasser aus T in G.
```

An dieser Stelle ist aber nicht klar, wann die Anweisung „`zu steifem Schnee schlagen`“ umgesetzt wurde. Wir sollen rühren, bis der Eischnee steif ist. Ein Ausweg können

Erfahrungswerte sein. Es dauert ungefähr zwei Minuten, bis das Eiweiß steif ist. Dann könnte man schreiben:

```
4.6 Rühre den Inhalt von G 2 Minuten lang.
```

Eine solche Anweisung birgt aber auch gewisse Risiken in sich. Die Fertigungszeit hängt davon ab, wie schnell und mit welchen Hilfsmitteln man rührt. Also wäre es uns lieber, wirklich ungefähr dann aufzuhören, wenn das gerührte Material steif geworden ist. Was brauchen wir dazu? Die Fähigkeit, Tests durchzuführen (um den Zeitpunkt zu erkennen, an dem die Anweisung „zu steifem Schnee schlagen“ umgesetzt worden ist) und abhängig von dem Resultat die Entscheidung zu treffen, wie man weiter vorgehen soll. Wenn der Schnee noch nicht steif ist, soll man noch gewisse Zeit rühren und dann wieder testen. Wenn der Schnee steif ist, ist Schritt 4 abgeschlossen und wir sollen mit dem Schritt 5 die Arbeit fortsetzen.

Wie kann man dies als eine Befehlsfolge schreiben?

```
4.6 Rühre den Inhalt von G 10s lang.

4.7 Teste, ob der Inhalt von G „steif“ ist.
    Falls JA, setze mit 5 fort.
    Falls NEIN, setze mit 4.6 fort.
```

Damit kehrt man zum Rühren in 4.6 so oft zurück, bis der gewünschte Zustand erreicht ist. In der Fachterminologie der Informatik nennt man 4.6 und 4.7 eine **Schleife**, in der man 4.6 so lange wiederholt, bis die Bedingung 4.7 erfüllt ist. Um dies zu veranschaulichen, benutzen wir oft eine graphische Darstellung wie in Abb. 4.1 auf der nächsten Seite, die man **Flussdiagramm** nennt.

Ist aber der Test in 4.6 so leicht durchführbar? Wir müssen uns, genau wie bei der Tätigkeitsanweisung, auf eine sorgfältig gewählte Liste von einfachen Tests einigen. Den Test 4.6 kann man zum Beispiel so realisieren, dass man in die Masse einen kleinen, leichten Kunststofflöffel stecken kann, und wenn er stecken bleibt, betrachten wir die Masse als steif. Beispiele von einfachen Tests könnten Folgende sein:

- Teste, ob die Flüssigkeit im Topf mindestens x°C hat.
- Teste, ob die Masse im Gefäß „löffelfest“ ist.

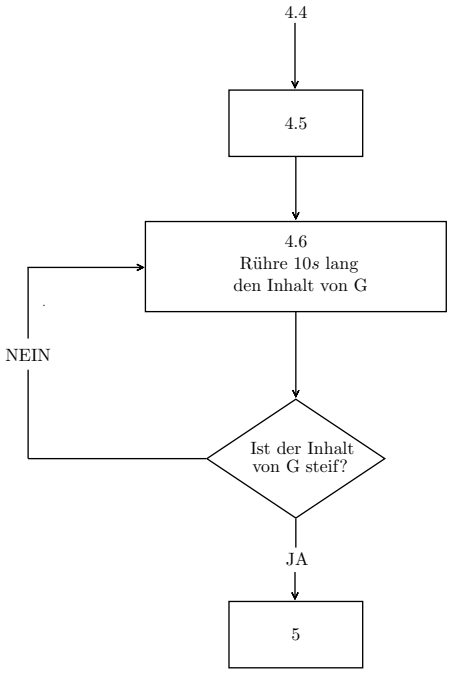


Abbildung 4.1 Flussdiagramm

- `Wiegt der Inhalt eines Gefäßes genau x g?`

Aufgabe 4.2 Setzt die Arbeit in Gruppen fort und stellt eine Liste von Testfragen her, die man beim Kuchenbacken braucht und jeder Person in der Klasse zumutbar sind.

Aufgabe 4.3 Schreibt den Schritt 5 des Rezeptes für den Aprikosenkuchen so um, dass in der Beschreibung dieser Tätigkeiten nur jene Tätigkeiten und Tests aus Euren Listen vorkommen.

Aufgabe 4.4 Schreibe ein Rezept für die Herstellung deines Lieblingsessens aus dem Kochbuch ab. Liste deiner Meinung nach einfache Anweisungen und Tests auf und schreibe dein Rezept so um, dass es nur Tätigkeiten aus einer Liste enthält.

Aufgabe 4.5 Wir wollen 1 l Wasser auf 90 °C erwärmen. Folgende Operationen hast du zur Verfügung:

> „Stelle den Topf T für x Sekunden auf eine heiße Herdplatte und nimm ihn dann weg.“
> und „Gieße y l Wasser in einen Topf T.“

Wir haben folgenden Test zur Verfügung:

> Hat das Wasser im Topf T mindestens x°C

Nutze diese zwei Anweisungen und den Test zur Herstellung eines „Kochalgorithmus“, der 1 l Wasser auf mindestens 90 °C erwärmt, so dass der Topf, nachdem das Wasser 90 °C erreicht hat, nicht länger als 15 Sekunden auf der Herdplatte stehen bleibt.

Ob du es glaubst oder nicht: Wenn du diese zwei Aufgaben gelöst hast, hast du schon ein bisschen programmiert. Das Wichtigste, das wir hier beim Kuchenbacken gelernt haben, ist, dass man über Algorithmen nicht sprechen kann, bevor man nicht die Grundbausteine für das Herstellen von Algorithmen festgelegt hat. Die Bausteine sind einerseits einfache Tätigkeiten, die jeder zweifelsfrei durchführen kann und andererseits einfache Tests, die man ebenfalls problemlos umsetzen kann.

Zusammenfassung

Unter einem Algorithmus verstehen wir eine eindeutig interpretierbare Tätigkeitsbeschreibung, die uns zu unserem Ziel führt. Diese Tätigkeitsbeschreibung besteht aus einer Folge von einfachen Tätigkeiten (Instruktionen, Operationen) und Tests, die jede in Frage kommende Person mit Sicherheit realisieren kann. Damit erfordert die Definition des Algorithmus, sich zuerst auf eine Liste von einfachen Tätigkeiten und auf eine Liste von einfachen Testfragen zu einigen. Unabhängig von der Person, die nach dem Algorithmus arbeitet, muss die ganze Tätigkeit und somit auch das Resultat der Algorithmusanwendung gleich sein.

Kontrollfragen

1. Erkläre mit eigenen Worten, welche Anforderungen an den Begriff des Algorithmus gestellt werden und warum.

2. Unter welchen Umständen dürfen wir ein Rezept für einen Algorithmus halten?

Lektion 5

Programmieren in der Sprache des Rechners

Hier wollen wir zuerst auf die Ähnlichkeiten und Unterschiede zwischen algorithmischem „Kochen“ und dem Rechnen mit einem Computer eingehen und dadurch die Anforderungen an einen Algorithmus als Computerprogramm genauer formulieren.

Hinweis für die Lehrperson Der folgende Teil bis zum Beispiel 5.1 eignet sich für einen Vortrag. Der Text ist zu lang für eine selbständige Bearbeitung und enthält sehr wenige Übungen. Die Interaktion mit der Klasse sollte durch Fragestellungen und Diskussionen zum Thema „Begriffsbildung“ im Zusammenhang mit der vorherigen Lektion gewährleistet werden.

Genauso wie beim Kochen muss man sich zuerst auf die Menge der einfachen Basistätigkeiten (Operationen) einigen, die ein Rechner mit Sicherheit ausführen kann. Diese Einigung fällt uns hier viel leichter als beim Kochen. Die Rechner haben keinen Intellekt und somit auch keine Improvisationsfähigkeiten. Damit ist die Rechnersprache sehr einfach. Niemand bezweifelt die Tatsache, dass Rechner Zahlen addieren, multiplizieren oder andere arithmetische Operationen durchführen – wir verwenden in diesem Zusammenhang den Fachausdruck „Operation über Zahlen“ – sowie Zahlen bezüglich ihrer Größe vergleichen können. Das kann jeder einfache Taschenrechner. Diese einfachen Operationen zusammen mit der Fähigkeit, Eingabedaten zu lesen und Resultate auszugeben, reichen aus, um jeden Algorithmus als Folge solcher Operationen darzustellen.

Also egal, ob Kochalgorithmen oder Rechneralgorithmen, alle sind nichts anderes als Folgen von einfachen Operationen (Tätigkeitsanweisungen). Es gibt aber einen wesentlichen Unterschied zwischen Kochalgorithmen und Algorithmen in der Informatik. Die

J. Hromkovič, *Lehrbuch Informatik*, DOI 10.1007/978-3-8348-9692-6_20,

Kochalgorithmen haben als Eingabe die Zutaten und das Resultat ist ein Kuchen. Die einzige Aufgabe, die sie haben, ist, aus festgelegten Zutaten den gegebenen Kuchen zu backen. Bei algorithmischen Problemen ist es ganz anders. Wir wissen, dass ein Problem *unendlich viele* **Problemfälle** (auch **Probleminstanzen** genannt) als mögliche Eingabe für einen Algorithmus haben kann. Als Beispiel untersuchen wir das Problem der Lösung einer quadratischen Gleichung

$$ax^2 + bx + c = 0.$$

Die Eingabe sind die Zahlen a, b und c und die Aufgabe besteht darin, alle x zu finden, die diese Gleichung erfüllen.

Ein konkreter Problemfall ist zum Beispiel, die folgende quadratische Gleichung zu lösen:

$$x^2 - 5x + 6 = 0$$

Hier ist $a = 1, b = -5$ und $c = 6$. Die Lösungen sind $x_1 = 2$ und $x_2 = 3$. Man kann durch Einsetzen leicht überprüfen, dass

$$2^2 - 5 \cdot 2 + 6 = 4 - 10 + 6 = 0$$

$$3^2 - 5 \cdot 3 + 6 = 9 - 15 + 6 = 0$$

und somit x_1 und x_2 wirklich die Lösungen der quadratischen Gleichung $x^2 - 5x + 6 = 0$ sind.

Weil es unendlich viele Zahlen gibt, haben wir unendlich viele Möglichkeiten, a, b und c in der quadratischen Gleichung zu wählen. Also gibt es *unendlich viele* quadratische Gleichungen. Von einem Algorithmus zur Lösung des Problems von quadratischen Gleichungen fordern wir, dass er für jede Eingabe a, b und c (also für jede quadratische Gleichung) die Lösung bestimmt.

Damit haben wir die zweite grundlegende Anforderung an eine Festlegung des Begriffes **Algorithmus** formuliert.

> *Ein Algorithmus zur Lösung einer Aufgabe (eines Problems) muss garantieren, dass er für jeden möglichen Problemfall korrekt arbeitet. Korrekt arbeiten bedeutet hier, dass er für jede Eingabe in endlicher Zeit die Arbeit beendet und das korrekte Ergebnis liefert.*

Überlegen wir uns jetzt einen Algorithmus zur Lösung quadratischer Gleichungen. Die Mathematik bietet uns die folgende Formel an:

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$
$$x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

falls $b^2 - 4ac \geq 0$. Falls $b^2 - 4ac < 0$ gilt, existiert keine reelle Lösung[1] der Gleichung. Diese Formel liefert uns die folgende allgemeine Methode zur Lösung quadratischer Gleichungen direkt.

Eingabe: Zahlen a, b und c für die quadratische Gleichung $ax^2 + bx + c = 0$.

Schritt 1: Berechne den Wert $b^2 - 4ac$.

Schritt 2: Falls $b^2 - 4ac \geq 0$, dann berechne

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$
$$x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}.$$

Schritt 3: Falls $b^2 - 4ac < 0$, schreibe „Es gibt keine reelle Lösung“.

Wir glauben zuerst einmal den Mathematikern, dass die Methode wirklich funktioniert. Wir brauchen nicht zu wissen warum, um sie in einen Algorithmus im Sinne eines Computerprogramms umzuschreiben.

Aufgabe 5.1 Beschreibe auf eine ähnliche Art und Weise eine Methode zur Lösung linearer Gleichungen der Form $ax + b = cx + d$.

Aufgabe 5.2 Beschreibe eine Methode zur Lösung des folgenden Systems von zwei linearen Gleichungen mit zwei Unbekannten x und y.

$$ax + by = c$$
$$dx + ey = f$$

[1] Dies gilt, weil wir nicht die Wurzel aus einer negativen Zahl ziehen können.

Wir wollen jedoch mehr, als solche Methoden in Programme umzusetzen. Den Begriff **Programm** verstehen wir hier als *Folge von rechnerunterstützten Operationen*, die in einer für den Rechner verständlichen Form dargestellt werden. Zwischen den Begriffen „Programm“ und „Algorithmus“ gibt es zwei wesentliche Unterschiede.

1. Ein Programm muss nicht einen Algorithmus darstellen, es kann eine sinnlose Folge von Operationen sein.

2. Ein Algorithmus muss nicht in der formalen Sprache des Rechners, also in einer Programmiersprache dargestellt werden. Einen Algorithmus kann man in einer natürlichen Sprache oder in der Sprache der Mathematik beschreiben. Zum Beispiel „multipliziere a und c“ oder „berechne $\sqrt{c}$“ ist in einem Algorithmus als Anweisung zulässig, während in einem Programm diese Anweisung in einem ganz speziellen Formalismus der gegebenen Programmiersprache ausgedrückt werden muss.

Der erste Unterschied zwischen Programmen und Algorithmen ist wesentlich. Der zweite Unterschied liegt nur in der Darstellung. Wenn man bei einem exakten mathematischen Modell von Algorithmen fordert, dass Algorithmen in der Sprache des Rechnermodells dargestellt werden, kommt der zweite Unterschied nicht zum tragen. Dann sind Algorithmen spezielle Programme, die eine sinnvolle Tätigkeit ausüben, z.B. ein konkretes Problem für jede Probleminstanz lösen.

Als **Programmieren** bezeichnen wir *die Tätigkeit, in der wir Algorithmen in Programme umschreiben*. Wir werden jetzt ein bisschen programmieren, um zu verstehen, wie Rechner arbeiten und um zu sehen, wie man aus einer Folge von sehr einfachen Befehlen (Operationen) komplexes Verhalten erzeugen kann.

Wir fangen damit an, die erlaubten, einfachen Operationen und ihre Darstellung in unserer Programmiersprache, die wir ASSEMBLER nennen wollen, aufzulisten. Dabei zeigen wir, wie man sich einen Rechner vorstellen kann und was genau bei der Ausübung dieser Operationen im Rechner passiert.

Wir stellen uns einen zu einem gewissen Grad idealisierten Rechner wie in Abb. 5.1 vor. Dieses Rechnermodell nennen wir **Registermaschine**.

Der Rechner setzt sich aus folgenden Teilen Zusammen:

- Einem **Speicher**, der aus einer großen Anzahl von Speicherzellen besteht. Diese

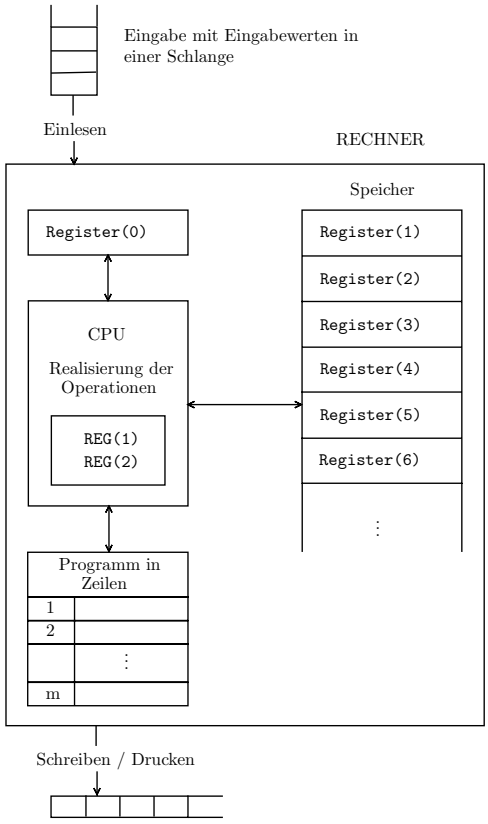


Abbildung 5.1

Speicherzellen werden **Register** genannt und sind mit positiven ganzen Zahlen durchnummeriert (siehe Abb. 5.1). Jedes Register kann eine beliebige Zahl speichern[2]. Am Anfang einer Berechnung enthalten alle Register die Zahl 0. Die Nummer eines Registers nennen wir die **Adresse** des Registers. Zum Beispiel ist 112 die Adresse des Registers `Register(112)`. Dies entspricht der Vorstellung, dass alle Register wie Häuser auf einer Seite einer langen Straße nebeneinander stehen.

[2] In realen Rechnern bestehen die Register aus einer festen Anzahl von Bits, z.B. 16 oder 32. Zu große ganze Zahlen oder reelle Zahlen mit vielen Nachkommastellen, die nicht auf 32 Bits gespeichert werden können, muss man gesondert behandeln. Hier idealisieren wir, um anschaulich zu bleiben und setzen voraus, dass man beliebig große Zahlen komplett (vollständig) in einem Register abspeichern kann.

- Einem besonderen Register, dem `Register(0)`, das die Nummer der Programmzeile enthält, die gerade bearbeitet wird oder zu bearbeiten ist.

- Einem speziellen Speicher, in dem das Programm zeilenweise gespeichert ist. Jede Zeile des Programms enthält genau eine Instruktion (Anweisung, Operation) des Programms.

- Einer CPU (central processing unit, die mit allen anderen Teilen verbunden ist. Die CPU liest zuerst in der aktuellen Zeile des Programms (bestimmt durch den Inhalt des `Registers(0)`), welche Instruktion auszuführen ist. Danach holt sich die CPU die Inhalte (gespeicherten Zahlen) aus den in der Instruktion angesprochenen Registern und führt die entsprechende Operation auf den Daten durch. Am Ende speichert die CPU das Resultat in einem durch die Instruktion bestimmten Register und ändert den Inhalt des Registers `Register(0)` in die Zahl der nächsten auszuführenden Zeile des Programms um. Um diese Aktivitäten umsetzen zu können, hat die CPU zwei spezielle Register, die wir hier als `REG(1)` und `REG(2)` bezeichnen.

Zusätzlich ist der Rechner mit der Außenwelt verbunden. Die Eingabedaten stehen in einer Warteschlange und der Rechner kann immer die erste Zahl in der Warteschlange einlesen und in einem seiner Register abspeichern. Der Rechner hat auch ein Band, auf das er seine Resultate schreiben darf.

Überlegen wir uns eine Analogie zum Kuchenbacken oder allgemein zum Kochen. Der Rechner ist die Küche. Die Register des Speichers sind Gefäße aller Art, Schalen, Töpfe, Becher usw. Jedes Gefäß hat einen Namen (genau wie ein Register) und somit ist es immer klar, über welches Gefäß als Speicherzelle man gerade spricht. Der Speicher mit dem Programm ist ein Blatt Papier oder ein Kochbuch. Die CPU sind wir oder ein Kochroboter mit allen Maschinen wie Herd, Mixer, Mikrowelle usw., die für die Tätigkeit zur Verfügung stehen. Der Inhalt des Registers `Register(0)` ist für uns die Notiz, an welcher Stelle wir uns bei der Ausführung des Rezeptes befinden. Die Eingaben liegen im Kühlschrank und in der Speisekammer. Üblicherweise zwar nicht in einer Warteschlange, aber wir können die Zutaten immer vor dem Kochen herausholen und in der Reihenfolge, in der sie gebraucht werden, vorbereiten. Die Ausgabe wird nicht geschrieben, sondern auf den Esstisch gelegt.

Um uns bei der Beschreibung von Rechneraktivitäten kurz halten zu können, verwenden wir auch die kurze Bezeichnung `R(i)` für das Register `Register(i)`. Für die zwei

Register in der CPU verwenden wir konsequent die Bezeichnung `REG(1)` und `REG(2)`. Mit `Inhalt(R(i))` bezeichnen wir die Zahl, die aktuell im Register `R(i)` gespeichert ist.

Wie wir schon am Beispiel des Kuchenbackens gelernt haben, ist das Erste und das Zentrale für die Bestimmung des Begriffes „Algorithmus“ die Festlegung einer Liste von **durchführbaren** Instruktionen (Anweisungen, Befehlen, Operationen). Über die Durchführbarkeit muss es ein allgemeines Einverständnis geben. Von all diesen Synonymen ziehen wir beim Rechneralgorithmus die Fachbegriffe „**Operation**“ und „**Instruktion**“ vor.

Wir formulieren hier die Operationen auch umgangssprachlich und verwenden nicht die Sprache des Rechners (**Maschinencode**), die alle Befehle als Folgen von 0 und 1 darstellt. Die von uns verwendete Programmiersprache heißt **ASSEMBLER** und steht dem Maschinencode am nächsten. Im Prinzip sind die in ASSEMBLER verwendeten Befehle genau die Instruktionen, die ein Rechner ausführen kann. Der einzige Unterschied zur Maschinensprache besteht in einer verständlicheren Darstellung der Instruktionen. Wenn man es verstanden hat, in ASSEMBLER zu programmieren, dann hat man einerseits eine gute Vorstellung von der Entstehung und der Funktionalität des Rechners gewonnen und andererseits die Tatsache entdeckt, dass ein sehr komplexes Verhalten durch eine Folge von sehr einfachen Anweisungen erzeugt werden kann. Wir beginnen mit der Vorstellung der Leseoperationen.

(1) READ
`Lese ein in REG(1).`
Diese Operation durchzuführen bedeutet, die erste Zahl in der Eingabewarteschlange in `REG(1)` zu speichern. Somit verschwindet diese Zahl aus der Warteschlange und die zweite Zahl in der Warteschlange rückt auf die Position 1. Der Inhalt von `R(0)` erhöht sich um 1 und somit wird im nächsten Schritt die Operation in der nächsten Zeile des Programms durchgeführt (Abb. 5.2). Die Auswirkung dieser Instruktion beschreiben wir wie folgt:

`REG(1)` ← die erste Zahl in der Warteschlange
`R(0)` ← `Inhalt(R(0))` + 1

Der Pfeil ← bezeichnet den Datentransfer. Auf der linken Seite des Pfeiles steht der Name des Registers, in dem jene Zahl gespeichert wird, welche auf der rechten Seite des Pfeiles bestimmt wird.

(2) STORE i
`Speichere den Inhalt von REG(1) in R(i)`
Die Zahl, die in `REG(1)` gespeichert ist, wird in `R(i)` abgespeichert. Der alte Inhalt von `R(i)` wird damit gelöscht und `Inhalt(REG(1))` ändert sich nicht. Die Kurzbeschreibung der Auswirkung dieser Instruktion ist:

$$\texttt{R(i)} \leftarrow \texttt{Inhalt(REG(1))}$$
$$\texttt{R(0)} \leftarrow \texttt{Inhalt(R(0))} + 1$$

Beispiel 5.1 In der Warteschlange befinden sich drei Zahlen in der Folge 114, -67, 1 und warten darauf, abgeholt zu werden (Abb. 5.2 (a)). Im Speicher beinhalten alle Register den Wert 0, nur `R(0)` enthält 1. Jetzt wird die Instruktion READ in der ersten Zeile des Programms

```
1 READ
2 STORE 1
3 READ
4 STORE 3
5 READ
6 STORE 2
```

bearbeitet. Nach ihrer Durchführung enthält `REG(1)` die gelesene Zahl 114. In der Eingabewarteschlange warten noch -67 und 1. Der Inhalt von `R(0)` wird um 1 auf 2 erhöht, weil man nach der Bearbeitung der ersten Zeile des Programms mit der nächsten fortfährt. Dieser Ablauf ist in Abb. 5.2 veranschaulicht. Wir verzichten hier auf die vollständige Beschreibung des Rechners und zeichnen nur die Register und ihre Inhalte.

Weil `Inhalt(R(0))`=2 ist, wird im nächsten Schritt die Operation STORE 1 aus der zweiten Zeile des Programms ausgeführt. Dabei speichert man die Zahl 114=`Inhalt(REG(1))` in `R(1)` ab. Der Inhalt von `REG(1)` ändert sich dabei nicht und der Inhalt von `R(0)` erhöht sich um 1 auf 3. Der Speicherzustand ist in der dritten Spalte der Tabelle 5.1 eingezeichnet. Allgemein dokumentiert die *i*-te Spalte der Tabelle den Zustand der Speicher nach der Durchführung des *i*-ten Rechnerschrittes.

Im dritten Schritt wird der Befehl READ in der dritten Zeile des Programms ausgeführt, weil `Inhalt(R(0))`=3 nach dem zweiten Schritt gilt. Damit wird -67 als die erste Zahl in der Warteschlange in `REG(1)` abgespeichert und damit der alte Inhalt von `REG(1)` gelöscht. Der Inhalt von `R(0)` wird um 1 erhöht. Die Zahl -67 wird aus der Warteschlange entfernt. Sonst ändert sich nichts. □

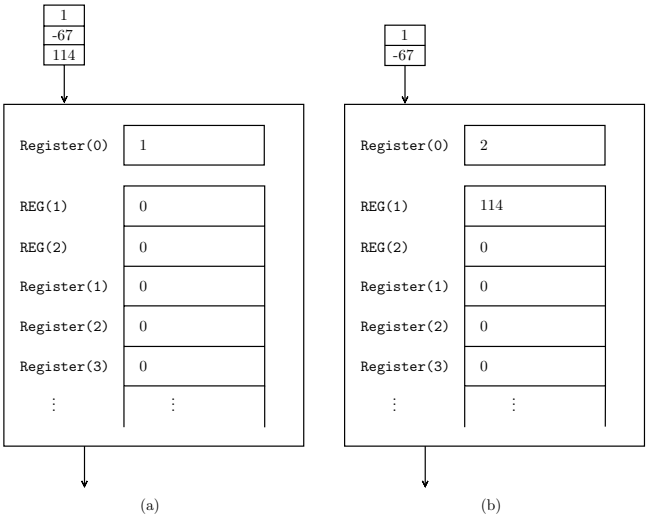


Abbildung 5.2

Aufgabe 5.3 Erkläre mit eigenen Worten, was die Durchführung der nächsten drei Operationen in den Zeilen 4, 5 und 6 des Programms in Beispiel 5.1 bewirken. Beziehe dich dabei auf die Tabelle 5.1.

Aufgabe 5.4 Nehmen wir an, in die Warteschlange wird noch die Zahl -7 als weitere Eingabe gesetzt. Was wird sich im Rechnerspeicher abspielen, wenn das Programm in Beispiel 5.1 um die Zeilen

```
7 STORE 5
8 READ
9 STORE 1
```

erweitert wird? Wird danach die Zahl 114 noch irgendwo gespeichert? Erweitere die Tabelle 5.1 um die entsprechenden drei Spalten und den zusätzlichen Eingabewert -7.

Aufgabe 5.5 In der Warteschlange warten die fünf Zahlen -1,0,1,2 und 5. Zeichne eine Tabelle, welche die Entwicklung der Speicherinhalte bei der Durchführung des folgenden Programms dokumentiert:

```
1 READ
2 STORE 3
```

Schritte	0	1	2	3	4	5	6
Warteschlange	1 -67 114	1 -67	1 -67	1	1		
REG(1)	0	114	114	-67	-67	1	1
REG(2)	0	0	0	0	0	0	0
R(0)	1	2	3	4	5	6	7
R(1)	0	0	114	114	114	114	114
R(2)	0	0	0	0	0	0	1
R(3)	0	0	0	0	-67	-67	-67
R(4)	0	0	0	0	0	0	0
R(5)	0	0	0	0	0	0	0

Tabelle 5.1

```
3 READ
4 READ
5 STORE 1
6 STORE 2
7 READ
8 STORE 1
9 READ
```

Es besteht auch die Möglichkeit, die Zahlen aus dem Hauptspeicher (aus den Registern R(1), R(2), R(3), ...) in die Register REG(1) und REG(2) zu übertragen. Dies kann durch folgende Befehle bewirkt werden:

(3) LOAD1 i
Die Wirkung dieses Befehles kann man mittels

$$\texttt{REG(1)} \leftarrow \texttt{Inhalt(R(i))}$$
$$\texttt{R(0)} \leftarrow \texttt{Inhalt(R(0))} + 1$$

beschreiben. Der Inhalt des Registers R(i) wird im Register REG(1) abgespeichert. Dabei ändert sich der Inhalt von R(i) nicht, nur der alte Inhalt von REG(1) wird gelöscht. Wie bei allen vorherigen Operationen wird der Zähler für die Zeilennummer des Programms um 1 erhöht und damit setzt das Programm seine Arbeit mit

der Ausführung der Operation in der nächsten Zeile fort.

(4) LOAD2 i
Diese Operation hat fast die gleiche Wirkung wie LOAD1 i, mit dem einzigen Unterschied, dass die in `R(i)` gespeicherte Zahl in `REG(2)` abgespeichert wird. In Kürze kann man die Wirkung dieser Operation wie folgt beschreiben:

`REG(2)` ← `Inhalt(R(i))`
`R(0)` ← `Inhalt(R(0))` + 1

(5) LOAD1 =i
Diesen Befehl auszuführen, bedeutet nichts anderes als die Abspeicherung der Zahl i in `REG(1)`. Es wird damit kein Transfer von Daten aus dem Speicher in die CPU stattfinden. Eine kurze Schreibweise der Wirkung dieses Befehles ist:

`REG(1)` ← i
`R(0)` ← `Inhalt(R(0))` + 1

(6) LOAD2 =j
Analog zu dem Befehl LOAD1 =j wird die Zahl j in `REG(2)` gespeichert. Eine kurze Beschreibung der Wirkung dieser Operation ist damit wie folgt:

`REG(2)` ← j
`R(0)` ← `Inhalt(R(0))` + 1

Aufgabe 5.6 Dokumentiere mittels einer Tabelle die Entwicklung des Speichers bei der Ausführung des folgenden Programms:

```
1 LOAD1 =2
2 LOAD2 =3
3 STORE 4
4 LOAD2 4
```

Wie wir schon am Anfang erwähnt haben, dienen die Register `REG(1)` und `REG(2)` zur Speicherung von Operanden, über welche man dann die arithmetischen Operationen ausführt. Mittels der Befehle LOAD und STORE haben wir schon gelernt, die Daten zwischen Speicher und CPU in beiden Richtungen zu übertragen. Wir können damit jene

Zahlen in REG(1) und REG(2) platzieren, mit denen wir rechnen wollen. Jetzt lernen wir die arithmetischen Grundoperationen des Rechners kennen.

(7) ADD
Die Inhalte der Register REG(1) und REG(2) werden addiert und das Resultat wird in REG(1) abgespeichert. Damit wird der alte Inhalt von REG(1) (der erste Operand der Addition) gelöscht. Wie üblich erhöht sich der Zeilenzähler dabei um 1. Eine kurze Beschreibung der Auswirkung dieses Befehles folgt:

$$\texttt{REG(1)} \leftarrow \texttt{Inhalt(REG(1))} + \texttt{Inhalt(REG(2))}$$
$$\texttt{R(0)} \leftarrow \texttt{Inhalt(R(0))} + 1$$

(8) SUB
Der Befehl SUB entspricht der Subtraktion, in welcher der Inhalt von REG(2) vom Inhalt von REG(1) subtrahiert wird. Das Resultat wird in REG(1) abgespeichert. Die kurze Beschreibung ist:

$$\texttt{REG(1)} \leftarrow \texttt{Inhalt(REG(1))} - \texttt{Inhalt(REG(2))}$$
$$\texttt{R(0)} \leftarrow \texttt{Inhalt(R(0))} + 1$$

(9) MULT
Die Ausführung von MULT entspricht der Multiplikation der Inhalte der Register REG(1) und REG(2) und der Abspeicherung des Resultats in REG(1). Die Kurzbeschreibung ist wie folgt:

$$\texttt{REG(1)} \leftarrow \texttt{Inhalt(REG(1))} * \texttt{Inhalt(REG(2))}$$
$$\texttt{R(0)} \leftarrow \texttt{Inhalt(R(0))} + 1$$

(10) DIV
Die Bedeutung dieser Instruktion entspricht der Teilung der Zahl in REG(1) durch die Zahl in REG(2). Die Tätigkeit des Rechners bei der Ausführung von DIV kann kurz wie folgt beschrieben werden:

$$\texttt{REG(1)} \leftarrow \texttt{Inhalt(REG(1))} \,/\, \texttt{Inhalt(REG(2))}$$
$$\texttt{R(0)} \leftarrow \texttt{Inhalt(R(0))} + 1$$

Wenn Inhalt(REG(2))=0 ist, bricht der Rechner seine Arbeit ab und schreibt „ERROR“ auf das Ausgabeband.

Beispiel 5.2 In der Warteschlange warten drei Zahlen x, y und z. Unsere Aufgabe ist es, den Wert $(x+y)-\frac{z}{2}$ zu berechnen und in `R(5)` abzuspeichern. Wir schreiben zu diesem Zweck das folgende Programm. In der Tabelle 5.2 sieht man die Entwicklung der Speicherinhalte. Um die Anschaulichkeit zu erhöhen, tragen wir nur dann die Werte in

Schritte	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Warte-schlange	z y x	z y	z y	z	z											
`REG(1)`	0	x		y		z			$\frac{z}{2}$		x		$x+y$		$x+y-\frac{z}{2}$	
`REG(2)`	0							2				y		$\frac{z}{2}$		
`R(0)`	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
`R(1)`	0		x													
`R(2)`	0				y											
`R(3)`	0						z									
`R(4)`	0									$\frac{z}{2}$						
`R(5)`	0															$x+y-\frac{z}{2}$

Tabelle 5.2

die Tabelle ein, wenn der Inhalt des Registers im entsprechenden Schritt geändert wurde.

```
1 READ          REG(1) ← x
2 STORE 1       R(1) ← REG(1)
3 READ
4 STORE 2       R(2) ← y
5 READ
6 STORE 3
7 LOAD2 =2
8 DIV
9 STORE 4
10 LOAD1 1
11 LOAD2 2
12 ADD
13 LOAD2 4
14 SUB
15 STORE 5
```

Wir sehen, dass das Programm zuerst die Eingabewerte x, y und z in den Registern `R(1)`, `R(2)` und `R(3)` abspeichert. Dann berechnet es mit DIV den Wert $\frac{z}{2}$ und speichert es in `R(4)` ab. Danach überträgt es den Wert x in `REG(1)` und den Wert y in `REG(2)`, um sie addieren zu können. Abschließend überträgt das Programm den Wert $\frac{z}{2}$ in `REG(2)`, um mit SUB das definitive Resultat zu berechnen. □

Aufgabe 5.7 Seien $x = 2$, $y = 8$ und $z = 14$ die Eingabewerte des Programms. Simuliere die Arbeit des Programms auf dieser Eingabe und gib dabei die konkreten Werte der einzelnen Speicherregister nach jedem Schritt des Programms an.

Aufgabe 5.8 Schreibe ein Programm, das für gegebene zwei Zahlen x und y den Durchschnitt $\frac{(x+y)}{2}$ berechnet und in `R(1)` abspeichert. Dabei sollen die Eingabewerte x und y nicht verloren gehen und deswegen in irgendwelchen Speicherregistern abgespeichert werden. Zeichne zu deinem Programm eine Tabelle wie Tab. 5.2, die die Änderungen der Speicherinhalte nach den einzelnen Schritten dokumentiert.

Aufgabe 5.9 In der Warteschlange warten vier Zahlen a, b, c und x. Schreibe ein Programm zur Berechnung des Polynomwertes $ax^2 + bx + c$. Schreibe für die Eingabe $a = 1$, $b = -14$, $c = 12$ und $x = 3$ die Entwicklung der Speicherinhalte während der Ausführung des Programms auf.

Aufgabe 5.10 Schaffst du es, für die vorherige Aufgabe 5.9 ein Programm zu entwickeln, das nur zweimal den Befehl MULT verwendet?

Aufgabe 5.11 In der Warteschlange steht nur ein Wert x. Schreibe ein Programm zur Berechnung des Polynomwertes $2x^2 + 2x - 4$. Zeichne ähnlich wie in Tab. 5.2 eine Tabelle, welche die Änderungen der Inhalte aller Register nach der Ausführung einzelner Zeilen deines Programms dokumentiert.

Aufgabe 5.12 Kannst du für die Aufgabe 5.11 ein Programm schreiben, das nur einmal die Operation MULT verwendet?

Für die Vereinfachung von Programmen führen wir noch die zwei einfachen Operationen $+1$ und -1 ein.

(11) ADD1
Nach der Ausführung dieser Operation wächst der Inhalt von `REG(1)` um 1. Die formale Beschreibung lautet:

REG(1) ← Inhalt(REG(1)) + 1
R(0) ← Inhalt(R(0)) + 1

(12) SUB1
Nach der Ausführung dieser Operation verringert sich der Inhalt von REG(1) um 1. Also entspricht SUB1 folgenden Aktionen:

REG(1) ← Inhalt(REG(1)) − 1
R(0) ← Inhalt(R(0)) + 1

Berechne, dass du das Gleiche wie durch ADD1 auch mittels

LOAD2 =1
ADD

bewirken kannst. Das Einzige, was dabei verloren geht, ist der vorherige Inhalt von REG(2). Dies kann man dadurch korrigieren, dass man den ursprünglichen Inhalt von REG(2) noch unverändert in dem Register behält, aus welchem die Zahl durch das letzte LOAD2 i in REG(2) geladen wurde. Wenn also R(i) diesen Wert behalten hat, kann man ADD1 durch folgendes Programm ersetzen:

LOAD2 =1
ADD
LOAD2 i

Aufgabe 5.13 Durch welches Programm kann man die Operation SUB1 ersetzen?

Die nächsten Instruktionen dienen der Ausgabe der berechneten Resultate.

(13) WRITE i
Der Inhalt von R(i) wird auf das Ausgabeband gedruckt. Wir beschreiben es kurz mittels

Ausgabe ← Inhalt(R(i))
R(0) ← Inhalt(R(0)) + 1.

Auf dem Ausgabeband wird nie etwas gelöscht. Wenn man den Befehl WRITE mehrmals hintereinander verwendet, wird auf dem Ausgabeband des Rechners die entsprechende Folge von Zahlen entstehen, die durch Kommata abgetrennt sind.

(14) WRITE1
Dieser Befehl dient zur direkten Ausgabe des Inhaltes des CPU-Registers `REG(1)`. Die Auswirkung dieser Instruktion ist die folgende:

Ausgabe ← `Inhalt(REG(1))`
`R(0)` ← `Inhalt(R(0))` + 1

(15) WRITE =j
Die Zahl j wird zur Ausgabe des Rechners:

Ausgabe ← j
`R(0)` ← `Inhalt(R(0))` + 1

Aufgabe 5.14 Erweitere deine Programme aus Aufgabe 5.9 und 5.11 um die Ausgabe der berechneten Polynomwerte.

Aufgabe 5.15 In der Warteschlange warten drei Zahlen x, y und z. Schreibe ein Programm, das zuerst diese drei Zahlen als Ausgabe liefert und danach den Durchschnittswert dieser drei Zahlen ausgibt.

Alle bisher vorgestellten Instruktionen führen zur zeilenweisen Ausführung der Programme und beinhalten somit keine Testfragen. Wir wissen schon vom Kuchenbacken, dass Tests ein notwendiger Bestandteil einer Instruktionsliste sind. Abhängig von den berechneten Zwischenresultaten kann der Bedarf entstehen, auf unterschiedliche Art und Weise die Arbeit fortzusetzen. Wenn wir aber die Berechnung abhängig vom Inhalt eines Registers mit unterschiedlichen Strategien fortsetzen wollen, müssen wir dies durch Sprünge (Änderungen des Inhalts von `R(0)`) bewirken. Das Ganze kann man sich wie folgt vorstellen: Unterschiedliche Rechenverfahren sind in unterschiedlichen Teilen des Programms implementiert. Wenn wir ein konkretes Verfahren verwenden wollen, müssen wir statt in der nächsten Zeile des Programms in jener Zeile fortfahren, in der die Implementation des Verfahrens steht. Dies bewirkt man durch die passende Einstellung des Wertes in `R(0)`, wie wir es auch in folgenden Instruktionen machen.

(16) JZERO j
Das J in JZERO steht für „Jump“ und ZERO steht für den Test auf Null. Falls `Inhalt(REG(1))`=0, dann ersetze den Inhalt des Registers `R(0)` durch j. Eine kurze Beschreibung der Auswirkung dieses Befehls sieht wie folgt aus:

if `Inhalt(REG(1))` = 0
then `R(0)` ← j
else `R(0)` ← `Inhalt(R(0))` + 1

Damit setzt das Programm die Arbeit in der nächsten Zeile fort, wenn die Bedingung (0 in `REG(1)`) nicht erfüllt ist. Wenn die Bedingung erfüllt ist, setzt das Programm die Ausführung in der j-ten Zeile fort.

(17) JGTZ j
Wieder steht J in JGTZ für „Jump“ und GTZ steht für „greater than zero“. Damit ist die Auswirkung des Befehls die folgende:

if `Inhalt(REG(1))` > 0
then `R(0)` ← j
else `R(0)` ← `Inhalt(R(0))` + 1

Manchmal wünscht man sich auch, bei der Ausführung des Programms bedingungslos zu einer anderen als der nächsten Zeile zu springen. Dies kann vorkommen, wenn wir in der letzten Zeile einer Verfahrensimplementierung sind und mit einem anderen Verfahren fortfahren wollen. Zu diesem Zweck dient der folgende Befehl:

(18) JUMP j
Diese Instruktion hat die folgende einfache Wirkung:

`R(0)` ← j

Beim Schreiben von Programmen erwarten wir auch einen klaren Befehl, wann die Ausführung des Programms beendet werden soll. Dazu dient der Befehl

(19) END
Nach dem Lesen des Befehls END beendet der Rechner die Ausführung des Programms.

Beispiel 5.3 In der Warteschlange warten mehrere Zahlen und wir wissen nicht wie viele. Wir wissen nur, dass sich die Zahlen bis auf die letzte von 0 unterscheiden. Wenn 0 eingelesen wird, ist es ein Zeichen dafür, dass die Folge von Eingabewerten zu Ende ist. Unsere Aufgabe ist, die Summe aller Zahlen in der Warteschlange zu berechnen. Wenn man 0 einliest, sollte man die berechnete Summe als Ausgabe liefern und die Arbeit beenden. Unsere Berechnungsstrategie kann durch das Flussdiagramm aus Abb. 5.3 dargestellt werden.

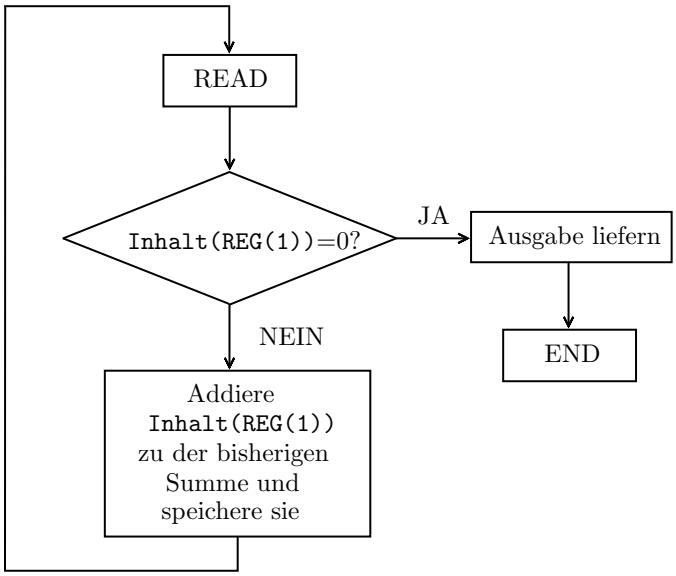


Abbildung 5.3

Jetzt implementieren wir die Strategie aus Abb. 5.3. Vor der Implementierung ist es immer gut zu entscheiden, in welchen Registern die Zwischenresultate gespeichert werden sollen. Hier berechnen wir nur die Summe der gelesenen Zahlen und müssen damit nur dieses eine Zwischenresultat speichern. Wir machen dies in `R(1)`.

```
1 READ
2 JZERO 7
3 LOAD2 1
4 ADD
5 STORE 1
```

```
6 JUMP 1
7 WRITE 1
8 END
```

□

Aufgabe 5.16 Simuliere die Arbeit des Programms aus Beispiel 5.3 für die Eingabe 1, -7, 13, 0 und zeichne dabei die Tabelle der Speicherinhalte.

Aufgabe 5.17 Modifiziere das Programm aus Beispiel 5.3 so, dass es zusätzlich noch den Durchschnittswert der aufsummierten Zahlen berechnet. Veranschauliche dein Vorgehen mittels eines Flussdiagramms wie in Abb. 5.3.

Aufgabe 5.18 Ähnlich wie in Beispiel 5.3 soll man die Zahlen in der Warteschlange aufsummieren, bis die Null als Signal für das Ende der Eingabe gelesen wird. Der Unterschied liegt in der Forderung, dass alle Zahlen in der Warteschlange positiv sind. Wenn noch vor der Null eine negative Zahl vorkommt, sollte das Programm abbrechen, ohne ein Resultat zu liefern.

Beispiel 5.4 Die Eingabe ist eine Folge von zwei Zahlen x und y und unsere Aufgabe ist es, die größere der beiden Zahlen in `R(1)` zu speichern und auszugeben. Wenn beide Zahlen gleich groß sind, spielt es keine Rolle, welche ausgegeben wird. Wir haben keine Operation zur Verfügung, welche die Inhalte von zwei Registern vergleichen kann. Wir können den Vergleich so realisieren, dass wir die Differenz $x-y$ berechnen und dann testen, ob $x-y>0$ oder $x-y\leq 0$ gilt. Falls $x-y>0$ gilt, dann gilt $x>y$ und wir nehmen x als das Maximum von $\{x,y\}$. Ansonsten nehmen wir y als das Maximum von $\{x,y\}$ an. Die Programmstruktur kann man durch das Flussdiagramm aus Abb. 5.4 anschaulich darstellen.

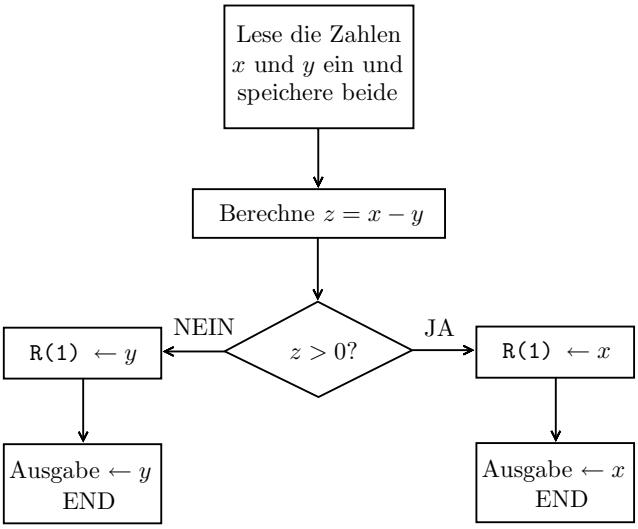


Abbildung 5.4

Das entsprechende Programm in ASSEMBLER sieht wie folgt aus.

```
1 READ
2 STORE 2
3 READ
4 STORE 3
5 LOAD1 2
6 LOAD2 3
7 SUB
8 JGTZ 13
9 LOAD1 2
10 STORE 1
11 WRITE 1
12 END
13 LOAD1 3
14 STORE 1
15 WRITE 1
16 END
```

□

Aufgabe 5.19 Simuliere das Programm aus Beispiel 5.4 für $x = 3$ und $y = 5$ und schreibe dabei die entsprechenden Änderungen der Inhalte in den einzelnen Registern in einer Tabelle auf.

Aufgabe 5.20 Das Programm aus Beispiel 5.4 kann man um zwei Zeilen kürzen, ohne das Vorgehen wesentlich zu ändern. Weißt du wie?

Aufgabe 5.21 In der Warteschlange ist eine Folge von positiven Zahlen, die mit einer Null endet. Entwickle ein Programm, welches das Maximum der Zahlen dieser Folge in `R(1)` abspeichert und ausgibt. Wenn nur eine Null kommt, soll das Programm 0 ausgeben. Zum Vergleich von Zahlenpaaren kannst du die Strategie aus Beispiel 5.4 verwenden. Veranschauliche zuerst deine Vorgehensweise mittels eines Flussdiagramms.

Aufgabe 5.22 In der Warteschlange warten zehn Zahlen. Entwickle ein Programm, das sich wie folgt verhält: Wenn mindestens eine der zehn Zahlen 0 ist, gibt das Programm 0 aus. Wenn keine der Zahlen 0 ist und die Anzahl der positiven Zahlen gleich der Anzahl der negativen Zahlen ist, gibt das Programm 1 aus. Sonst gibt das Programm -1 aus.

Es ist interessant zu bemerken, dass ziemlich komplexe Aufgaben mit so einfachen Instruktionen wie der Übertragung von Zahlen zwischen Registern, arithmetischen Operationen oder dem Vergleich einer Zahl mit 0 gelöst werden können. Im Prinzip geht es noch einfacher. Mit Ausnahme der Übertragungsbefehle reicht der Test auf 0 und die Operationen ADD1 (+1) und SUB1 (-1) aus. Alle anderen arithmetischen Operationen kann man durch Programme ersetzen, die in der Berechnung nur +1 und -1 verwenden.

Beispiel 5.5 In der Warteschlange warten zwei positive ganze Zahlen a und b. Wir wollen ein Programm entwickeln, das $a+b$ berechnet und in `R(1)` speichert. Dabei dürfen keine arithmetischen Operationen außer ADD1 und SUB1 verwendet werden.

Die Idee ist, $a+b$ als

$$a+\underbrace{1+1+\cdots+1}_{\text{b Mal}}$$

zu berechnen. Wir gehen wie folgt vor: Wir speichern a in `R(1)` ab und addieren b-mal 1 zum Inhalt von `R(1)`. Die Zahl b wird in `R(2)` gespeichert. Wir verringern diese Zahl um 1 nach jeder Erhöhung des Wertes in `R(1)` um 1. Wenn `Inhalt(R(2))=0` gilt, wissen

wir, dass wir b-mal den Inhalt von R(1) um 1 erhöht haben und somit, dass R(1) die Summe $a+b$ beinhaltet. Diese Vorgehensweise ist im Flussdiagramm von Abb. 5.5 veranschaulicht.

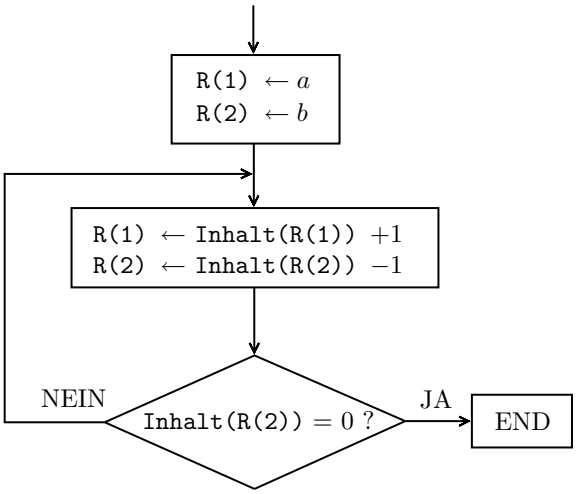


Abbildung 5.5

Das entsprechende Programm in ASSEMBLER sieht wie folgt aus:

```
1 READ
2 STORE 1
3 READ
4 STORE 2
5 LOAD 1
6 ADD1
7 STORE 1
8 LOAD 2
9 SUB1
10 STORE 2
11 JZERO 13
12 JUMP 5
13 END
```

□

Aufgabe 5.23 Ersetze den Befehl JZERO 13 in der Zeile 11 des Programms aus Beispiel 5.5 durch den Testbefehl JGTZ. Was muss man noch ändern, damit das Programm weiterhin korrekt arbeitet? Wird das Programm dadurch kürzer?

Aufgabe 5.24 Entwickle das Programm aus Beispiel 5.5 weiter, so dass es $a + b$ für alle ganzen Zahlen (auch negative) korrekt berechnet.

Aufgabe 5.25 Entwirf ein Programm, das für zwei gegebene Zahlen a und b die Multiplikation $a * b$ berechnet. Dabei darf es nur die arithmetischen Befehle ADD, SUB, ADD1 und SUB1 verwenden.

Aufgabe 5.26 Verwende das von dir in Aufgabe 5.24 entwickelte Programm, um das aus Beispiel 5.3 so umzuschreiben, dass es außer ADD1 und SUB1 keine andere arithmetische Operation vorwendet.

Eine unserer wichtigsten Anforderungen an die Definition eines Algorithmus für ein Problem (für eine Aufgabenstellung) ist, dass der Algorithmus in endlicher Zeit die Arbeit beendet und eine Antwort liefert. In der Fachsprache der Informatik sprechen wir vom **Halten**. Wenn ein Algorithmus A auf einer Eingabe x endlich lange arbeitet und danach die Arbeit beendet, dann sagen wir, dass der **Algorithmus** A **auf der Aufgabeninstanz** x **hält**. Mit den Worten eines Informatikers ausgedrückt, fordern wir, dass ein Algorithmus **immer hält**, was bedeutet, dass er auf jeder möglichen Eingabe hält.

Jemand könnte natürlich einwenden: „Das ist doch logisch. Wer würde schon Programme zur Problemlösung entwickeln, die endlos arbeiten und niemals eine Ausgabe liefern?“ Das Problem ist aber, dass die Entwickler unbeabsichtigt ein Programm bauen können, das für einige Eingaben (Problemfälle) in eine endlose Wiederholung einer Schleife gerät. Wie kann so etwas einem Profi passieren? Ganz einfach, er vergisst z. B. Sondersituationen zu betrachten, die unter gewissen Umständen vorkommen können. Kehren wir zurück zu den Kochalgorithmen, um zu sehen, wie leicht so etwas passieren kann.

Wir wollen das Wasser in einem Topf zum Kochen bringen und es danach für Tee verwenden. Dabei wollen wir mit der Energie sparsam umgehen und das Wasser nicht länger als 20 Sekunden kochen lassen. Jemand könnte dazu das Kochprogramm aus Abb. 5.6 auf der nächsten Seite vorschlagen.

Auf den ersten Blick scheint alles in Ordnung zu sein, der Algorithmus sollte funktionieren – bis ein Bergsteiger den Kochalgorithmus für das Zubereiten seines Nachmittagstees auf

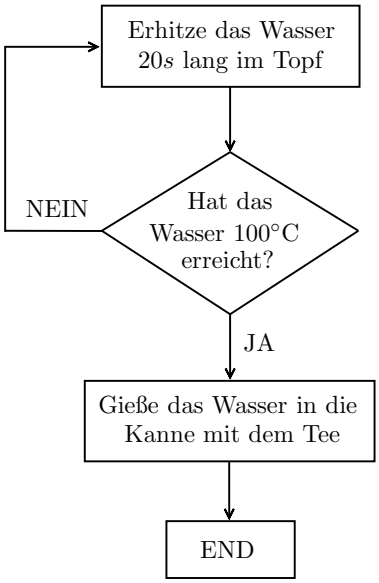


Abbildung 5.6

dem Matterhorn verwenden will. In dieser Höhe kocht das Wasser wegen des geringeren Luftdrucks schon bei niedrigeren Temperaturen. So kann es dem Bergsteiger passieren, dass der Test auf 100 °C nie mit einer positiven Antwort endet. Das Wasser wird zwar nicht wirklich ewig kochen, weil irgendwann der Brennstoff zur Neige geht oder das Wasser verdampft ist.

Wir sehen schon, wo der Fehler steckt. Beim Aufschreiben des Kochrezeptes hat man einfach nicht an diese Sondersituation gedacht. Und genau das Gleiche kann passieren, wenn man nicht an alle Sonderfälle des zu lösenden Problems und an alle Sonderentwicklungen denkt, die während des Rechnens vorkommen können.

So etwas kann auch einem Programmierer passieren, insbesondere wenn das entworfene Programm mehrere hunderttausend Zeilen enthält.

Ein gutes Beispiel für diese Situation ist eine unbeabsichtigte Verwendung des Programms aus Beispiel 5.5 für die Addition zweier beliebiger ganzen Zahlen a und b, obwohl das Programm nur für positive a und b geschrieben wurde. Wenn a negativ und b positiv ist, wird das Programm korrekt arbeiten. Aber wenn b negativ ist, wird man durch die wiederholte

Minderung von b um 1 nie Null erhalten. Deswegen wird der Test „`Inhalt(R(2))=0`“ immer mit der Antwort NEIN enden und das Programm wird unendlich viele Male den Teil des Programms von der Zeile 5 bis zur Zeile 12 wiederholen. Das Programm hält für eine negative Eingabe b nicht und somit ist es kein Algorithmus zur Addition beliebiger ganzen Zahlen.

Aufgabe 5.27 Was passiert, wenn b zwar positiv, aber keine ganze Zahl ist?

Zusammenfassung

Die Begriffe „Programm“ und „Algorithmus“ sind keine Synonyme. Ein Programm kann eine sinnlose Tätigkeit ausüben oder unendlich lange ohne jede Ausgabe laufen. Von einem Programm erwarten wir nur, dass es eine Folge von Rechnerbefehlen ist. Ein Algorithmus ist ein Programm, das immer (auf jeder zulässigen Eingabe) hält und das richtige Resultat liefert. Über einen Algorithmus kann man nur im Zusammenhang mit einem Problem sprechen.

Manchmal verwenden wir den Begriff des Algorithmus auch für eine Methodenbeschreibung, die nicht einem Programm in einer Programmiersprache entspricht. Wenn wir eine solche Vorgehensweise in ein Programm umschreiben, dann sprechen wir von Programmieren.

Das einfache Rechnermodell, genannt Registermaschine, besteht aus einem Speicher, einer CPU, einem Input- und einem Outputmedium und Kommunikationskanälen zwischen seinen Einheiten. Der Hauptspeicher besteht aus vielen Registern, die mit natürlichen Zahlen durchnummeriert sind. Die Nummer eines Registers bezeichnet man als seine Adresse. In der CPU sind drei spezielle Register `R(0)`, `REG(1)` und `REG(2)` enthalten. Das Register `R(0)` dient der Speicherung der Zeilennummer, in der sich der Rechner bei der Ausführung des Programms gerade befindet. Die Instruktionen (Befehle) des Rechners beinhalten:

(i) Kommunikationsanweisungen zur Übertragung von Daten (gespeicherten Zahlen) zwischen der CPU, dem Hauptspeicher, dem Input- und dem Outputmedium

(ii) arithmetische Operationen

(iii) Vergleiche einer Zahl mit 0 und

(iv) Sprünge zwischen Programmzeilen

Bevor man anfängt ein Programm zu schreiben, sollte der Algorithmus (die Vorgehensweise) klar sein und vorzugsweise in Form eines Flussdiagramms dargestellt werden. Die Ausführung eines Programms kann man gut kontrollieren, indem man die Änderungen der Speicherinhalte beobachtet und in einer Tabelle dokumentiert.

Kontrollfragen

1. Was ist ein Algorithmus? Beschreibe mit eigenen Worten alles, was du über diesen Begriff in den letzten drei Lektionen erfahren hast.

2. Was ist ein Programm? Warum unterscheiden wir zwischen Programmen und Algorithmen?

3. Was bezeichnen wir als eine Probleminstanz? Wie viele Probleminstanzen kann ein Problem haben?

4. Wie sieht das Modell der Registermaschine aus? Aus welchen Hauptteilen besteht die Registermaschine?

5. Wie ist der Speicher der Registermaschine organisiert? Was kann man in einem Register abspeichern?

6. Welche Register sind in der CPU enthalten? Was für eine Funktion haben sie?

7. Wie weiß der Rechner bei der Ausführung eines Programms, wo im Programm er sich gerade befindet?

8. Welche Befehle zum Einlesen von Daten hat die Registermaschine? Nenne alle und beschreibe ihre Wirkung.

9. Welche Befehle zur Ausgabe von Resultaten stehen zur Verfügung?

10. Welche Instruktionen stehen für die Übertragung von Daten zwischen dem Speicher und der CPU zur Verfügung?

11. Welche Testfragen kann man in ASSEMBLER stellen? Wie setzt man Sprünge zwischen den Programmzeilen um?

12. Was würdest du versuchen, um die korrekte Funktion deiner Programme zu überprüfen?

13. Wozu sind Flussdiagramme nützlich?

14. Wie kann es vorkommen, dass ein Programm unendlich lange läuft? Zeige ein Beispiel.

15. Was bedeutet der Satz: „Das Programm hält immer.“?

Kontrollaufgaben

1. Im Register `R(1)` liegt eine Zahl a und in `R(2)` eine Zahl b. Schreibe ein Programm, das die Inhalte der Register `R(1)` und `R(2)` austauscht. Das heißt, dass nach der Ausführung des Programms b in `R(1)` und a in `R(2)` liegen muss. Kannst du dir vorstellen, wo ein solches Programm eine Verwendung finden könnte?

2. In der Warteschlange warten drei positive ganze Zahlen a, b und c. Dein Programm soll diese Zahlen nach der Größe in den Registern `R(1)`, `R(2)` und `R(3)` abspeichern. Das Minimum von $\{a,b,c\}$ soll in `R(1)` liegen und das Maximum in `R(3)`.

3. In der Warteschlange liegen $n+1$ ganze Zahlen $n,x_1,x_2,\ldots,x_n$ für eine beliebige positive ganze Zahl n. Die erste Zahl besagt, wie viele Eingaben (Zahlen) hinter ihr in der Warteschlange warten. Entwirf ein Programm, welches

   a) den Durchschnittswert der Zahlen $x_1,x_2,\ldots,x_n$ berechnet

   b) die Anzahl der positiven Zahlen aus $\{x_1,\ldots,x_n\}$ bestimmt

   c) das Maximum und das Minimum von $\{x_1,\ldots,x_n\}$ findet und in `R(1)` und `R(2)` speichert

   d) die größte und die zweitgrößte Zahl aus $\{x_1,\ldots,x_n\}$ bestimmt

   Bevor du anfängst, die Programme zu schreiben, erkläre deine Vorgehensweise mittels Flussdiagrammen.

4. Schreibe ein Programm, dass für die Eingabe a, b, c, d die lineare Gleichung

   $$ax+b=cx+d$$

   löst. Wenn die Gleichung eine Lösung hat, soll in `R(1)` eine 1 stehen und `R(2)` soll die Lösung enthalten. Wenn alle reellen Zahlen die Gleichung erfüllen, dann soll in `R(1)` die Zahl 0 stehen. Wenn die Gleichung keine Lösung hat, soll `R(1)` die Zahl -1 beinhalten.

5. Entwirf ein Programm, das für die Eingabe a, b, c, d, e, f das System von zwei linearen Gleichungen

$$ax + by = c$$
$$dx + ey = f$$

mit zwei Unbekannten x und y löst. Du darfst voraussetzen, dass du nur solche Eingaben a, b, c, d, e, f bekommst, dass das entsprechende System genau eine Lösung hat.

6. Entwickle ein Programm, das für gegebene ganze Zahlen a und b die Zahl $a - b$ berechnet. Dabei sind die einzigen erlaubten, arithmetischen Operationen des ASSEMBLER die Operationen ADD1 und SUB1.

7. Entwickle ein Programm, das für zwei positive ganze Zahlen a und b bestimmt, ob $a > b$ oder $a \leq b$ gilt. Wenn $a > b$ gilt, gibt das Programm die Zahl 1 aus, ansonsten die Zahl 0. Das Programm darf den Testbefehl JGTZ nicht verwenden und muss nur mit dem Testbefehl JZERO auskommen. Kann dein Programm unendlich lange laufen, wenn a und b beliebige ganze (also auch negative) Zahlen sind? Erweitere das Programm so, dass es für beliebige ganze Zahlen a und b richtig funktioniert.

Lösungen zu ausgesuchten Aufgaben

Aufgabe 5.2
Hier wollen wir eine Methode zur Lösung eines Systems von zwei linearen Gleichungen entwickeln, indem wir Formeln für die Berechnung von x und y herleiten. Wir verwenden dazu das Substitutionsverfahren.

Aus der ersten Gleichung drücken wir x als

$$x = \frac{c - by}{a} \tag{5.1}$$

aus. Dann setzen wir x in die zweite Gleichung ein und erhalten die folgende Gleichung mit einer Unbekannten y:

$$\begin{aligned}
d \cdot \frac{c - by}{a} + ey &= f && \mid -\frac{dc}{a} \\
-\frac{db}{a} \cdot y + ey &= f - \frac{dc}{a} \\
(e - \frac{db}{a}) \cdot y &= f - \frac{dc}{a} \\
\frac{ea - db}{a} \cdot y &= \frac{fa - dc}{a} && \mid \cdot \frac{a}{ea - db} \\
y &= \frac{fa - dc}{ea - db}
\end{aligned} \tag{5.2}$$

Jetzt setzen wir (5.2) für y in der Formel (5.1) ein und erhalten

$$\begin{aligned} x &= \frac{c - b \cdot \frac{(fa-dc)}{(ea-db)}}{a} \\ &= \frac{1}{a} \cdot \frac{cea - dbc - bfa + bdc}{ea - db} \\ &= \frac{1}{a} \cdot \frac{cea - bfa}{ea - db} = \frac{ce - bf}{ea - db}. \end{aligned}$$

Somit erhält man die folgende Methode für die Lösung von Systemen zweier linearer Gleichungen:
Eingabe: Zahlen a, b, c, d, e und f.
Schritt 1: Berechne den Wert $m = ea - db$.
Schritt 2: Falls $m \neq 0$, dann berechne

$$\begin{aligned} x &= \frac{ce - bf}{m} \\ y &= \frac{af - cd}{m}. \end{aligned}$$

Schritt 3: Falls $m = 0$ und $fa - dc \neq 0$, dann schreibe:
„Es gibt keine reelle Lösung“.
Falls $m = 0$ und $fa - dc = 0$, dann schreibe:
„Es gibt unendlich viele reelle Lösungen“.

Aufgabe 5.10
Dank des Distributivgesetzes erhalten wir:

$$ax^2 + bx + c = x(ax + b) + c$$

Diese Formel enthält nur zwei Multiplikationen $a \cdot x$ und $x \cdot (\quad)$ und zwei Additionen. Deswegen kannst du, dieser Formel folgend, ein Programm schreiben, das nur zwei Befehle MULT verwendet.

Aufgabe 5.12
Die Idee ist es, die Lösungen x_1 und x_2 des Polynoms $2x^2 + 2x - 4$ zu bestimmen und dann das Polynom als

$$(x - x_1) \cdot (x - x_2)$$

darzustellen. Offensichtlich enthält diese Darstellung nur eine Multiplikation. Lösen wir die Gleichung

$$\begin{aligned} 2x^2 + 2x - 4 &= 0 \quad |/2 \\ x^2 + x - 2 &= 0 \\ (x + 2) \cdot (x - 1) &= 0. \end{aligned}$$

Im letzten Schritt haben wir die bekannte Tatsache verwendet, dass $x_1 \cdot x_2 = -2$ und $x_1 + x_2 = -1$ gilt.

Das Programm kann wie folgt aussehen:

```
1 READ
2 STORE 1
3 LOAD2 =2
4 ADD
5 STORE 2
6 LOAD1 1
7 LOAD2 =−1
8 ADD
9 LOAD2 2
10 MULT
11 WRITE1
12 END
```

Die Entwicklung der Speicherinhalte ist in Tab. 5.3 dargestellt.

Schritte	0	1	2	3	4	5	6	7	8	9	10
Warteschlange	x										
REG(1)	0	x			$x+2$		x		$x-1$		$(x-1)\cdot(x-2)$
REG(2)	0			2				-1		$x+2$	
R(0)	1	2	3	4	5	6	7	8	9	10	11
R(1)	0		x								
R(2)	0					$x+2$					

Tabelle 5.3

Aufgabe 5.13
Die Operation SUB1 kann man durch

```
LOAD2 =−1
ADD
```

oder durch

```
LOAD2 =1
```

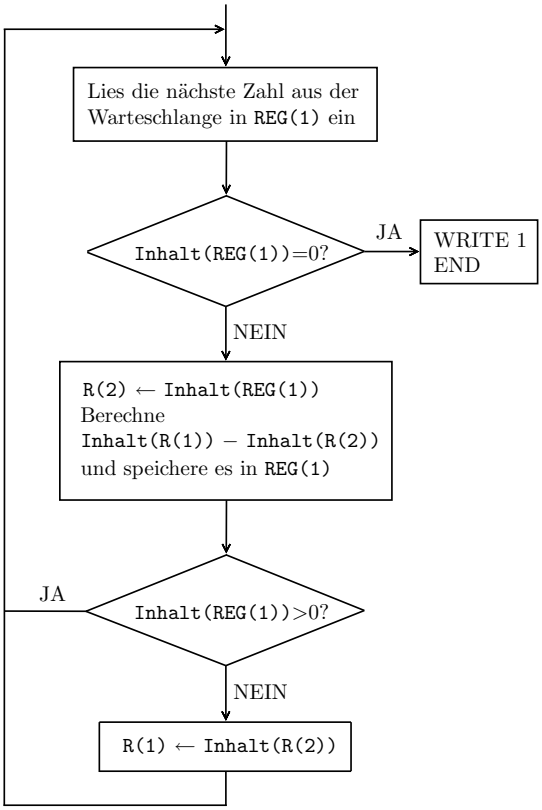


Abbildung 5.7

SUB

ersetzen. Man muss aufpassen, weil dabei der Inhalt von `REG(2)` gelöscht wird. Wie würdest du vorgehen, wenn du diese Information nach der Durchführung des Programms für SUB1 im `REG(2)` behalten willst?

Aufgabe 5.21
Am Anfang ist es immer wichtig, zuerst zu entscheiden, wozu welches Register dienen soll. Wir halten während der ganzen Laufzeit des Algorithmus in `R(1)` das bisherige Maximum und in `R(2)` die letzte eingelesene Zahl. Wir enden, wenn die neu gelesene Zahl gleich 0 ist. Sonst vergleichen wir das bisherige Maximum mit der neuen Zahl. Wenn die neue Zahl größer als das bisherige Maximum ist, legen wir sie in `R(1)` und wiederholen den Vorgang, indem wir die nächste Zahl lesen. Diese Strategie ist im Flussdiagramm aus Abb. 5.7 dargestellt.

Die entsprechende Implementierung dieser Strategie kann wie folgt aussehen:

```
1 READ
2 JZERO 11
3 STORE 2
4 LOAD1 1
5 LOAD2 2
6 SUB          REG(1) ← Inhalt(R(1)) − Inhalt(R(2))
7 JGTZ 1
8 LOAD1 2
9 STORE 1      R(1) ← Inhalt(R(2))
10 JUMP 1
11 WRITE 1
12 END
```

Kontrollaufgabe 1
Wir können nicht einfach ohne Weiteres den Inhalt von `R(2)` in `R(1)` legen, weil wir bei so einem Vorgang den Inhalt von `R(1)` löschen und damit verlieren würden. Also legen wir zuerst `Inhalt(R(1))` in `R(3)`. Dann können wir `Inhalt(R(2))` in `R(1)` übertragen und folglich `Inhalt(R(3))` in `R(2)` speichern.

```
1 LOAD1 1
2 STORE 3      R(3) ← Inhalt(R(1))
3 LOAD1 2
4 STORE 1      R(1) ← Inhalt(R(2))
5 LOAD1 3
6 STORE 2      R(2) ← Inhalt(R(3))
```

Überprüfe den korrekten Lauf dieses Programms, indem du die Entwicklung der Speicherinhalte nach jedem Rechenschritt in einer Tabelle aufzeichnest.

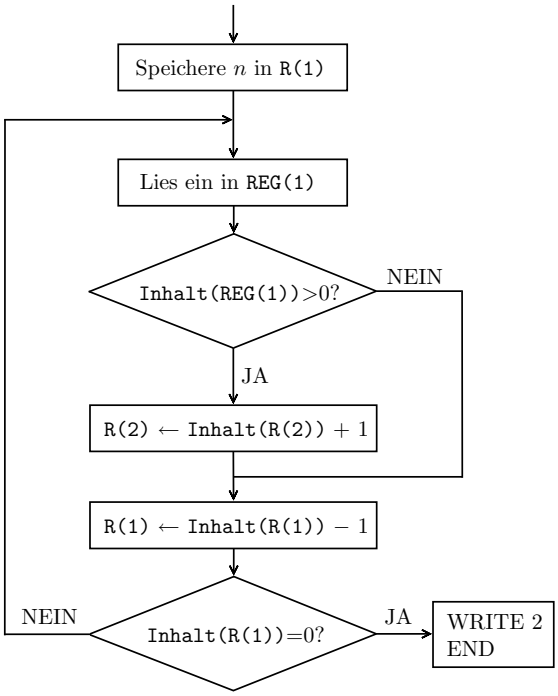


Abbildung 5.8

Kontrollaufgabe 3 (b)
Wir speichern die erste Zahl n in `R(1)`. Danach lesen wir eine Zahl nach der anderen ein. Um zu kontrollieren, dass wir genau n Nachfolgerzahlen von n gelesen haben, verkleinern wir nach jedem Lesen den Inhalt von `R(1)` um 1. Damit enthält `R(1)` die Anzahl noch zu lesender Zahlen. Wenn dann in `R(1)` die Zahl 0 liegt, wissen wir, dass die ganze Eingabe gelesen wurde. Für jede der n Zahlen prüfen wir, ob sie positiv ist. Falls ja, addieren wir 1 zum Inhalt des Registers `R(2)`. Somit enthält während der ganzen Ausführung des Programms `R(2)` die bisherige Anzahl positiver Zahlen. Diese Strategie kann durch das Flussdiagramm aus Abb. 5.8 veranschaulicht werden.

Die Voraussetzung in der Aufgabe war, dass n eine positive ganze Zahl ist. Wir haben uns auf die Einhaltung dieser Voraussetzung verlassen. Man könnte es aber überprüfen und für negative n oder $n = 0$ die Ausgabe 0 liefern. Kannst du das Flussdiagramm entsprechend erweitern?

Eine Implementierung des vorgeschlagenen Vorgehens aus Abb. 5.8 folgt:

```
1 READ
2 STORE 1        R(1) ← n
3 READ
4 JGTZ 6
5 JUMP 9
6 LOAD1 2
7 SUB1
8 STORE 2        R(2) ← Inhalt(R(2)) − 1
9 LOAD1 1
10 SUB1
11 STORE 1       R(1) ← Inhalt(R(1)) − 1
12 JZERO 14
13 JUMP 3
14 WRITE 2
15 END
```

Lektion 6

Indirekte Adressierung

In der vorherigen Lektion haben wir angedeutet, dass alles, was ein Rechner mit ganzen Zahlen machen kann, nur mit

- den arithmetischen Operationen ADD1 und SUB1,
- dem Test JZERO auf Null
- und den Kommunikationsoperationen vom Typ LOAD und STORE

umgesetzt werden kann. Arithmetisch gesehen reichen der Test auf 0, das Vergrößern um 1 und das Verkleinern um 1 vollständig für die Arithmetik mit ganzen Zahlen aus. Aus arithmetischer Sicht stimmt es, aber es stimmt nicht ganz aus Sicht der Kommunikationsbefehle zur Übertragung von Daten (Zahlen) zwischen dem Speicher und der CPU. Um wirklich alles machen zu können, müssen wir die Befehle der indirekten Adressierung einführen. Was das genau ist und warum sie gebraucht werden, entdecken wir bei den Versuchen, die folgende einfache Aufgabe zu lösen, welche mit der bisherigen Liste von Befehlen nicht lösbar ist.

Als Eingabe sind in der Warteschlange unbekannt viele ganze Zahlen. Es können zwei oder Tausende sein. Wir erkennen das Ende der Folge daran, dass alle Zahlen der Folge unterschiedlich von 0 sind und wenn eine Null eingelesen wird, ist es das Zeichen dafür, dass wir am Ende der Folge sind. Die Aufgabe ist keine Rechenaufgabe, sondern eine Aufgabe der Datenübertragung. Wir sollen alle Zahlen in der Warteschlange nacheinander einlesen und in den Registern `R(101)`, `R(102)`, `R(103)` usw. abspeichern. Das bedeutet, dass die i-te Zahl in der Warteschlange im Register `R(100+i)` gespeichert werden muss.

J. Hromkovič, *Lehrbuch Informatik*, DOI 10.1007/978-3-8348-9692-6_21,

Das Abspeichern endet, wenn eine 0 gelesen wird.

Die Strategie zur Umsetzung dieser Aufgabe könnte die folgende sein:

> Lies die nächste Zahl aus der Warteschlange in `REG(1)` ein. Wenn sie nicht 0 ist, speichere die Zahl in das nächste freie (noch nicht verwendete) Register ab der Adresse 101. Wenn die Zahl 0 ist, beende die Arbeit.

Eine halbformale Beschreibung eines Implementierungssversuches könnte wie folgt aussehen:

```
1 Lies ein in REG(1)
2 Falls Inhalt(REG(1))=0, dann gehe in Zeile □
3 R(101) ← REG(1)
4 Lies ein in REG(1)
5 Falls Inhalt(REG(1))=0, dann gehe in Zeile □
6 R(102) ← REG(1)
7 Lies ein in REG(1)
8 Falls Inhalt(REG(1))=0, dann gehe in Zeile □
9 R(103) ← REG(1)
⋮
```

Das Symbol □ steht für die Nummer der Zeile, in der sich der Befehl END befindet. Wir kennen aber die Nummer dieser Zeile nicht, weil wir nicht wissen, wann wir mit dem Einlesen von Daten und somit mit dem Schreiben des Programms aufhören dürfen. Wenn in der Warteschlange nur drei Zahlen (z.B. -17, 6, 3, 0) und dann die Zahl 0 stehen, dann ist das Programm fertig. Wenn da aber 10000 Zahlen stehen, müssen wir auf diese Weise 30000 Zeilen schreiben.

Das Hauptproblem ist nicht die unbekannte Zeilennummer für den Befehl END. Dies kann man gleich auf die folgende Weise umgehen: Das Programm fängt mit

```
1 JUMP 3
2 END
```

an und danach geht alles wie oben beschrieben weiter. Der einzige Unterschied ist, dass man jetzt alle Symbole □ durch die Zahl 2 ersetzen kann.

Das ernsthafte Problem ist aber, dass wir kein unendliches Programm schreiben können. Egal, wie lang unser Programm ist, es können immer noch mehr Zahlen in der Warteschlange stehen und somit wird das Programm nicht alle abspeichern. Eine natürliche Idee wäre, die drei nacheinander stehenden Befehle in eine Schleife zu legen. Das Problem ist, dass es sich nicht um eine Wiederholung dreier gleicher Befehle handelt. Die ersten zwei sind zwar gleich, aber die dritten

$$\texttt{R(101)} \leftarrow \texttt{REG(1)}, \texttt{R(102)} \leftarrow \texttt{REG(1)}, \texttt{R(103)} \leftarrow \texttt{REG(1)}$$

unterscheiden sich in der Adresse der Register, in welchen die aktuelle Zahl gespeichert wird. Um das Problem zu veranschaulichen, zeichnen wir unsere Strategie der Schleife mittels eines Flussdiagrammes (Abb. 6.1) auf.

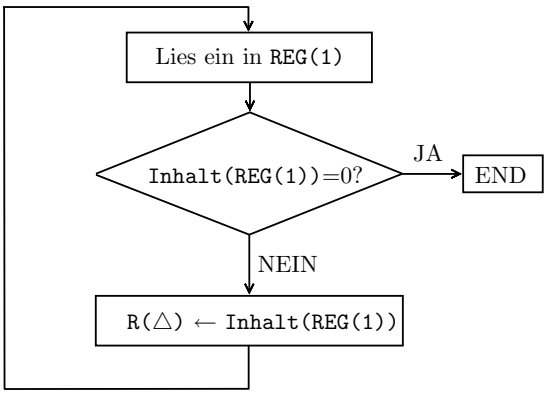


Abbildung 6.1

Wir sehen in Abb. 6.1, dass wir die Adresse $\triangle$ des Registers $\texttt{R(}\triangle\texttt{)}$ nicht festlegen können. Es kann nicht immer das gleiche Register sein, weil wir alle Zahlen abspeichern wollen. Genauer: Beim i-ten Durchlauf der Schleife muss man die aktuelle Zahl in das Register `R(100+i)` speichern. Wir können dies aber mit den vorhandenen Befehlen nicht umsetzen, weil diese (genau STORE $\triangle$) nur erlauben, eine feste Zahl für $\triangle$ in $\texttt{R(}\triangle\texttt{)}$ zu schreiben.

Deswegen führt man die Operationen (Instruktionen) der sogenannten **indirekten Adressierung** ein. Der Befehl

(20) STORE $*$i

für eine beliebige positive ganze Zahl i verursacht die folgende Aktivität: Der Rechner liest den Inhalt von `R(i)` und speichert `Inhalt(REG(1))` in dem Register mit der Adresse `R(i)`. Unsere Pfeilbeschreibung der Auswirkung dieses Befehls folgt:

`R(Inhalt(R(i)))` $\leftarrow$ `Inhalt(REG(1))`
`R(0)` $\leftarrow$ `Inhalt(R(0))` + 1

Kompliziert? Machen wir es an einem konkreten Beispiel anschaulich. Nehmen wir an, dass der Inhalt von `R(3)` die Zahl 112 und der Inhalt von `REG(1)` die Zahl 24 ist. Was passiert bei Ausführung des Befehls

STORE $*$3

d.h., was bedeutet die entsprechende Aktion

`R(Inhalt(R(3)))` $\leftarrow$ `Inhalt(REG(1))`?

Als erstes betrachtet der Rechner den Inhalt von R(3) und stellt fest, dass der Inhalt die Zahl 112 ist. Danach führt er die Aktion

`R(112)` $\leftarrow$ `Inhalt(REG(1))`

aus, die unserem Befehle STORE 112 entspricht. Nach der Ausführung von STORE $*$3 liegt in `R(112)` die Zahl 24=`Inhalt(REG(1))`. Der Inhalt von `R(0)` hat sich um 1 erhöht, ansonsten sind in allen Registern die gleichen Werte wie vor der Ausführung von STORE $*$3 enthalten.

Es gibt auch einen ähnlichen Befehl für die Übertragung von Daten aus dem Speicher in die CPU. Der Befehl

(21) LOAD1 $*$j

verursacht die folgende Aktivität:

`REG(1)` $\leftarrow$ `Inhalt(R(Inhalt(R(j))))`
`R(0)` $\leftarrow$ `Inhalt(R(0))` + 1

Der Rechner liest die Zahl $a =$`Inhalt(R(j))` aus `R(j)` und führt dann den Befehl

LOAD1 a

aus.

Aufgabe 6.1 Betrachten wir die folgende Situation. Das Register `REG(1)` enthält die Zahl -7, `R(0)` enthält die Zahl 4, `R(1)` enthält 101, `R(2)` enthält 207 und `R(110)` enthält -2. Alle anderen Register beinhalten die Zahl 0. Führe das folgende Programmstück aus, indem du eine Tabelle erzeugst, in welcher die Änderungen der Speicherinhalte nach einzelnen Schritten dokumentiert sind.

4 STORE $*2$
5 LOAD2 = 117
6 ADD
7 STORE $*1$
8 LOAD $*1$

Aufgabe 6.2 Es gibt ein Programm, dass die Änderung der Speicherinhalte wie in Tabelle 6.1 verursacht. Kannst du das Programm aufschreiben? Die Zeilen 6, 10, 11 und 14 müssen Befehle mit indirekter Adressierung beinhalten, auch wenn die Auswirkungen ohne indirekte Adressierung erreichbar wären.

Schritte	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Warteschlange	15														
	11	15	15	15	15										
	-7	11	11	11	11	15	15	15	15						
`REG(1)`	0	-7		10		11		12		15		11	12		15
`REG(2)`	0														
`R(0)`	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
`R(1)`	0		-7						12						
`R(2)`	0				10										
`R(3)`	0													12	
`R(10)`	0						11								
`R(12)`	0										15				

Tabelle 6.1

Mit Hilfe der indirekten Adressierung kann man unser Problem der Speicherung von unbekannt vielen Daten lösen. Den Befehl

$$\texttt{R}(\triangle) \leftarrow \texttt{Inhalt(REG(1))}$$

ersetzt man durch die Aktion

$$\texttt{R(Inhalt(R(1)))} \leftarrow \texttt{Inhalt(REG(1))},$$

d. h. durch den Befehl

STORE $*1$.

Somit muss man in `R(1)` immer die passende Adresse haben. Am Anfang legen wir in `R(1)` die Zahl 101 und nach jeder Abspeicherung erhöhen wir den Inhalt von `R(1)` um 1. Damit ändert sich das Flussdiagramm aus Abb. 6.1 in das Flussdiagramm aus Abb. 6.2.

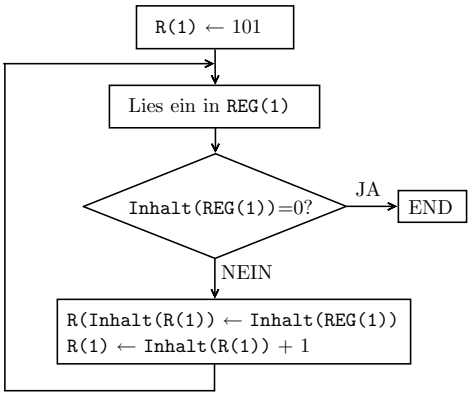


Abbildung 6.2

Das Diagramm kann wie folgt in ASSEMBLER implementiert werden:

```
1 LOAD1 =101
2 STORE 1          R(1) ← 101
3 READ
4 JZERO 10
5 STORE *1         R(Inhalt(R(1))) ← Inhalt(REG(1))
6 LOAD1 1
7 ADD1
8 STORE 1          R(1) ← Inhalt(R(1)) + 1
9 JUMP 3
10 END
```

Aufgabe 6.3 Als Eingabe steht in der Warteschlange die Zahlenfolge 113, -7, 20, 8, 0. Simuliere Schritt für Schritt das oben beschriebene Programm und zeichne dabei die Änderungen der Inhalte der Register mit den Adressen 1, 2, 3, 100, 101, 102, 103, 104 und 105 auf. Wir nehmen an, dass vor der Ausführung des Programms alle Register außer `R(0)` den Inhalt 0 haben.

Aufgabe 6.4 Was muss man im Programm (Abb. 6.2 auf der vorherigen Seite) ändern, damit die Zahlen aus der Warteschlange in den Registern mit den Adressen 10, 12, 14, 16 usw. (also auf geraden Adressen angefangen mit 10) abgespeichert werden?

Aufgabe 6.5 In der Warteschlange warten ganze Zahlen, alle unterschiedlich von 0. Die Anzahl der Zahlen ist unbekannt und die Zahl 0 signalisiert das Ende der Folge. Schreibe ein Programm in ASSEMBLER, das alle positiven Zahlen aus dieser Folge in den Registern mit den Adressen 50, 60, 70, 80 usw. abspeichert.

Hinweis für die Lehrperson Es ist wichtig zu beobachten, dass die indirekte Adressierung im Zusammenhang mit der Einführung von Feldern $A[i]$ steht. Es ist sinnvoll und sehr hilfreich, die Klasse später, im Unterricht von höheren Programmiersprachen, daran zu erinnern.

Beispiel 6.1 In der Warteschlange warten zwei nichtleere Folgen $a_1, \ldots, a_n$ und $b_1, \ldots, b_n$ von positiven ganzen Zahlen, welche durch die Zahl 0 getrennt werden. Demnach sieht der Anfang der Warteschlange folgendermaßen aus:

$$a_1, a_2, \ldots, a_n, 0, b_1, b_2, \ldots, b_n, 0$$

Die Zahl n (die Länge der Folgen) ist unbekannt und somit erkennen wir das Ende der ersten Folge durch das Einlesen von 0. Die Aufgabe ist, das Resultat

$$c_1 = a_1 + b_1, c_2 = a_2 + b_2, \ldots, c_n = a_n + b_n$$

zu berechnen und in n Registern zu speichern.

Wir werden wie folgt vorgehen: In `R(1)` werden wir die Elemente der ersten Folge a_1, $\ldots$, a_n zählen, um ihre Länge n zu erfahren. Wir speichern zuerst die Folge $a_1, a_2, \ldots, a_n$ in Registern mit den Adressen 10, 12, 14, $\ldots$, $10+2(n-1)$. Danach speichern wir b_1, $b_2, \ldots, b_n$ in Registern mit den Adressen 11, 13, 15, $\ldots$, $11+2(n-1)$. Wenn alle Daten in Registern sind, berechnen wir

$c_1 =$ `Inhalt(R(10))` + `Inhalt(R(11))`
$c_2 =$ `Inhalt(R(12))` + `Inhalt(R(13))`
$\vdots$
$c_n =$ `Inhalt(R(10+2(`n`-1)))` + `Inhalt(R(11+2(`n`-1)))`

und speichern die daraus resultierenden Werte in Registern mit den Adressen 10, 12, 14, $\ldots$, $10+2(n-1)$. Dieses Konzept kann wie in Abb. 6.3 gezeichnet umgesetzt werden.

□

Aufgabe 6.6 Setze das Flussdiagramm aus Abb. 6.3 mit einem Programm in ASSEMBLER um. Simuliere seine Arbeit auf der Eingabe 3, 7, 2, 0, 14, 12, 2, 0.

Aufgabe 6.7 Das Programm enthält drei Schleifen: die ersten zwei zum Abspeichern der beiden Eingabefolgen und die dritte zur Berechnung der Werte c_i. Dies ist unnötig kompliziert. Du musst die zweite Folge $b_1, b_2, \ldots, b_n$ nicht abspeichern und kannst stattdessen direkt c_i ausrechnen. Die Zahl c_i ist die Summe des Inhalts von `R(10+2(i-1))` und der gelesenen i-ten Zahl b_i. Zeichne ein entsprechendes Flussdiagramm, das nur zwei Schleifen enthält. Implementiere dein Flussdiagramm in ASSEMBLER.

Aufgabe 6.8 Beim Entwurf unseres Programms aus Abb. 6.3 haben wir vorausgesetzt, dass die Eingabe immer die korrekte Form hat. Erweitere das Programm so, dass es erkennt, wenn die Längen der beiden Eingabefolgen unterschiedlich sind. Wenn es so ist, wird dieses durch WRITE $=-1$ nach außen signalisiert.

Ein erfahrener Programmierer würde sofort bemerken, dass man die ersten beiden Schleifen mit Hilfe der indirekten Programmierung zusammenfassen kann. Genauer gesagt wird er die erste Schleife nochmals zum Einlesen der zweiten Folge $b_1, b_2, \ldots, b_n$ verwenden. Die beiden ersten Schleifen sind fast identisch. Das Problem liegt nur im Bestimmen der Längen beider Folgen. Dies kann man mittels indirekter Adressierung durch gleiche

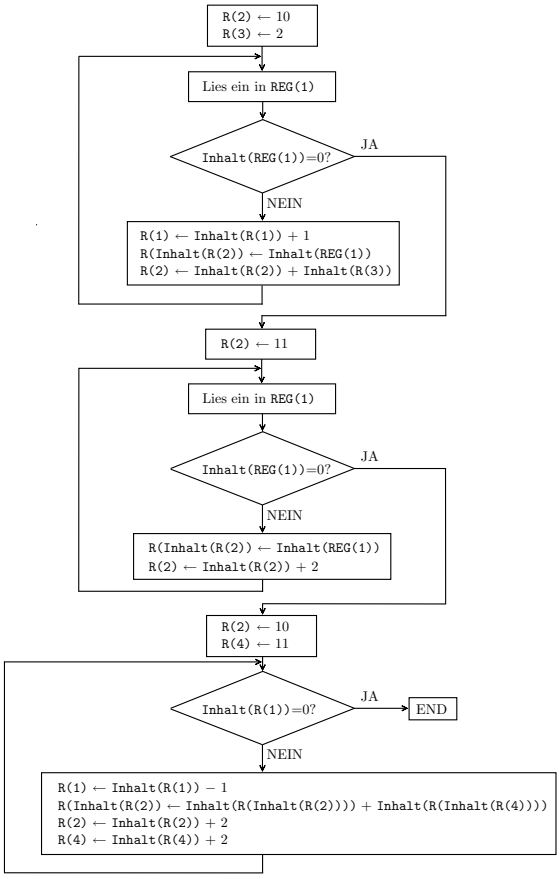


Abbildung 6.3

Befehle über unterschiedliche Adressen bewirken. Das entsprechende Flussdiagramm in Abb. 6.4 speichert die Länge der ersten Folge in `R(1)` und die Länge der zweiten Folge in `R(5)`. Die Werte 1 oder 5 in `R(4)` bestimmen jeweils, welche Länge ermittelt wird. Der Inhalt von `R(6)` sagt aus, ob wir die erste oder die zweite Folge einlesen.

Aufgabe 6.9 Implementiere die Strategie aus Abb. 6.4 in ASSEMBLER und teste dein Programm für die Eingabe 2, 3, 13, 0, 7 , 8, 0.

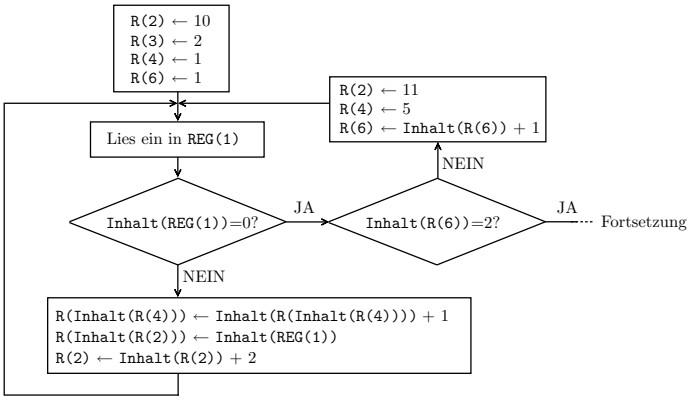


Abbildung 6.4

Zusammenfassung

Die Tabelle 6.2 auf Seite 385 beinhaltet alle 21 Befehle unseres ASSEMBLERS und zeigt in kurzer Beschreibung ihre Wirkung.
Die indirekte Adressierung ist ein wichtiges Programmierkonzept. Man kann die Adresse des Registers, mit welchem gearbeitet wird, durch den Inhalt eines anderen Registers `R'` angeben. Durch Rechnungen mit dem Inhalt von `R'` können wir dann die Adresse jenes Registers bestimmen, mit dem wir als nächstes arbeiten wollen. Der Vorteil liegt darin, dass wir den Inhalt von `R'` bei jedem Durchlauf in einer Schleife ändern können und damit soviele unterschiedliche Register während der Ausführung der Schleife verwenden können wie die vorher unbekannte Anzahl der Schleifendurchläufe.
Außer der oben diskutierten Verwendung ermöglicht uns eine indirekte Adressierung eine Verkürzung von Programmen, indem man ähnliche Teile durch einen parametrisierten Programmteil ersetzen kann.

Kontrollfragen

1. In welchen Situationen braucht man die indirekte Adressierung? Was ermöglicht sie?

2. Welchen Befehl der indirekten Adressierung hat man für den Datentransfer aus der CPU in den Hauptspeicher? Was bewirkt er?

3. Welchen Befehl der indirekten Adressierung hat man für den Datentransfer aus dem Speicher in die CPU? Was bewirkt er?

4. Wie gehst du vor, wenn du mittels einer Schleife und nicht durch eine lange Folge von hunderten Befehlen eine Folge von 100 Zahlen in der Warteschlange in die Register mit den Adressen 10, 11, 12, ..., 109 speichern sollst?

Kontrollaufgaben

1. Betrachte die gleiche Eingabe $a_1, a_2, \ldots, a_n, 0, b_1, b_2, \ldots, b_n, 0$ wie in Beispiel 6.1. Die Länge n der Folgen ist vorher nicht bekannt. Interpretiere die Eingabe als zwei Punkte $(a_1, a_2, \ldots, a_n)$ und $(b_1, b_2, \ldots, b_n)$ des n-dimensionalen Raums. Schreibe ein Programm, das die Euklidische Distanz (Entfernung) der beiden Punkte berechnet. Brauchst du dazu die indirekte Adressierung?

2. Die Eingabe ist die Gleiche wie in Kontrollaufgabe 1. Entwirf ein Programm, das die Entfernung der Punkte

   $(a_1, a_2, \ldots, a_n)$ und $(b_n, b_{n-1}, \ldots, b_1)$

   bestimmt.

3. In der Warteschlange steht nur eine Zahl n. Es soll ein Programm entwickelt werden, welches die Inhalte der Register

   `R(100)`, `R(101)`, ..., `R(100+`n`)`

   in die Register

   `R(101)`, `R(102)`, ..., `R(100+`n`+1)`

   so verschiebt, dass der Inhalt von `R(`k`)` nach der Ausführung des Programms in `R(`k`+1)` liegt.

4. In der Warteschlange stehen 20 ganze Zahlen. Du sollst ein Programm entwickeln, das diese Zahlen aufsteigend sortiert in die Register mit Adressen 1, 2, ..., 20 abspeichert.

5. In der Warteschlange steht eine Folge von ganzen Zahlen, die unterschiedlich von 0 sind. Die Anzahl der Zahlen ist unbekannt. Das Ende der Folge erkennt man durch die Zahl 0. Du sollst ein Programm in ASSEMBLER schreiben, das Folgendes leistet: Es speichert alle positiven Zahlen in Registern mit geraden Adressen 100, 102, 104, ... Die negativen Zahlen sollen in Registern mit ungeraden Adressen 101, 103, 105, ... abgespeichert werden.

6. Simuliere die Arbeit des Programms

```
1 LOAD1 =10
2 STORE 2
3 LOAD1 =2
4 STORE 3
5 READ
6 JZERO 16
7 STORE *2
8 LOAD1 2
9 LOAD2 3
10 SUB
11 STORE 2
12 LOAD1 1
13 ADD1
14 STORE 1
15 JUMP 5
16 END
```

auf der Eingabe -7, 3, 2, 0 und zeichne die Änderungen des Speicherinhalts nach der Ausführung jedes Rechnerschrittes auf.

7. Betrachte die gleiche Aufgabe wie in Kontrollaufgabe 3, nur mit dem Unterschied, dass man die Registerinhalte um zehn Positionen verschieben will. Entwickle mindestens zwei unterschiedliche Strategien zu diesem Zweck.

Lösungen zu ausgesuchten Aufgaben

Aufgabe 6.2
Das folgende Programm mit vier Befehlen der indirekten Adressierung bewirkt die in der Tabelle 6.1 dargestellte Datenänderung.

```
1 READ
2 STORE 1
3 LOAD1 =10
4 STORE 2
5 READ
6 STORE *2
7 ADD1
8 STORE 1
9 READ
```

```
10 STORE ∗1
11 LOAD ∗2 oder LOAD1 ∗2
12 ADD1
13 STORE 3
14 LOAD ∗3 oder LOAD1 ∗3
15 END
```

Aufgabe 6.4
Es sind nur zwei kleine Änderungen notwendig. Am Anfang legt man anstatt 101 die Zahl 10 in `R(1)`. Damit steht in der ersten Zeile LOAD1 =10, wo vorher LOAD1 =101 stand. In der Schleife muss man dann den Inhalt von `R(1)` immer um 2 statt um 1 erhöhen. Im Assemblerprogramm kann die Zeile ADD1 nochmals wiederholt werden, um den Inhalt von `R(1)` um 2 zu vergrößern. Wie würdest du vorgehen, wenn du anstatt des Befehls ADD1 den Befehl ADD vorzögest?

Kontrollaufgabe 3
Die Aufgabe wäre einfach, wenn bei der Abspeicherung einer Zahl in einem Register der vorherige Inhalt des Registers nicht automatisch gelöscht würde. Bevor wir also den Inhalt von `R(100)` in `R(101)` speichern, müssen wir den Inhalt von `R(101)` irgendwo speichern. Wir verwenden `R(10)` zum Speichern des zu übertragenden Inhaltes (z.B. am Anfang von `R(100)`) und `R(11)` zum Speichern jener Zahl, welche sonst gelöscht würde (z.B. am Anfang der Zahl in `R(101)`). Danach schieben wir den Inhalt von `R(11)` in `R(10)` und in `R(11)` legen wir die Zahl, die sonst bei der nächsten Übertragung gelöscht werden würde. Wir wiederholen dies n-mal, indem wir am Anfang n in `REG(1)` legen und diesen Wert nach jedem Durchlauf der Schleife um 1 verkleinern. Die Strategie ist im Flussdiagramm von Abb. 6.5 dargestellt. Die Implementierung in ASSEMBLER überlassen wir dir.

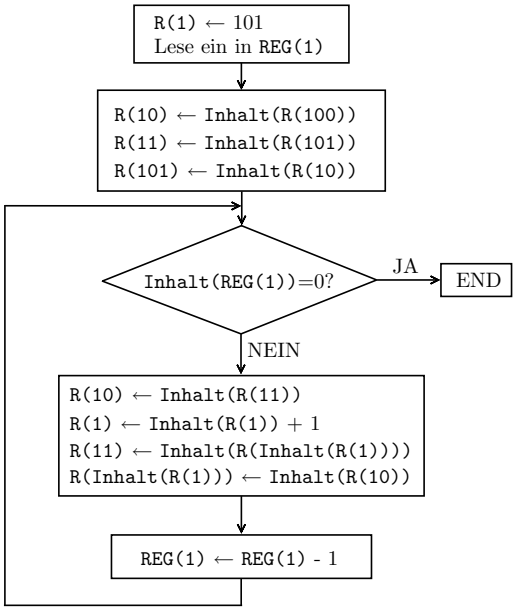


Abbildung 6.5

Befehl	Wirkung
READ	`REG(1)` $\leftarrow$ die erste Zahl in der Warteschlange
	`R(0)` $\leftarrow$ `Inhalt(R(0))` + 1
STORE i	`R(i)` $\leftarrow$ `Inhalt(REG(1))`
	`R(0)` $\leftarrow$ `Inhalt(R(0))` + 1
LOAD1 i	`REG(1)` $\leftarrow$ `Inhalt(R(i))`
	`R(0)` $\leftarrow$ `Inhalt(R(0))` + 1
LOAD2 i	`REG(2)` $\leftarrow$ `Inhalt(R(i))`
	`R(0)` $\leftarrow$ `Inhalt(R(0))` + 1
LOAD1 =i	`REG(1)` $\leftarrow$ i
	`R(0)` $\leftarrow$ `Inhalt(R(0))` + 1
LOAD2 =j	`REG(2)` $\leftarrow$ j
	`R(0)` $\leftarrow$ `Inhalt(R(0))` + 1
ADD	`REG(1)` $\leftarrow$ `Inhalt(REG(1))` + `Inhalt(REG(2))`
	`R(0)` $\leftarrow$ `Inhalt(R(0))` + 1
SUB	`REG(1)` $\leftarrow$ `Inhalt(REG(1))` $-$ `Inhalt(REG(2))`
	`R(0)` $\leftarrow$ `Inhalt(R(0))` + 1
MULT	`REG(1)` $\leftarrow$ `Inhalt(REG(1))` $*$ `Inhalt(REG(2))`
	`R(0)` $\leftarrow$ `Inhalt(R(0))` + 1
DIV	Wenn `Inhalt(REG(2))` $= 0$, melde „ERROR“, sonst
	`REG(1)` $\leftarrow$ `Inhalt(REG(1))` / `Inhalt(REG(2))`
	`R(0)` $\leftarrow$ `Inhalt(R(0))` + 1
ADD1	`REG(1)` $\leftarrow$ `Inhalt(REG(1))` + 1
	`R(0)` $\leftarrow$ `Inhalt(R(0))` + 1
SUB1	`REG(1)` $\leftarrow$ `Inhalt(REG(1))` $-$ 1
	`R(0)` $\leftarrow$ `Inhalt(R(0))` + 1
WRITE i	Ausgabe $\leftarrow$ `Inhalt(R(i))`
	`R(0)` $\leftarrow$ `Inhalt(R(0))` + 1
WRITE1	Ausgabe $\leftarrow$ `Inhalt(REG(1))`
	`R(0)` $\leftarrow$ `Inhalt(R(0))` + 1
WRITE =j	Ausgabe $\leftarrow$ j
	`R(0)` $\leftarrow$ `Inhalt(R(0))` + 1
JZERO j	**if** `Inhalt(REG(1))` $= 0$ **then** `R(0)` $\leftarrow$ j
	else `R(0)` $\leftarrow$ `Inhalt(R(0))` + 1
JGTZ j	**if** `Inhalt(REG(1))` > 0 **then** `R(0)` $\leftarrow$ j
	else `R(0)` $\leftarrow$ `Inhalt(R(0))` + 1
JUMP j	`R(0)` $\leftarrow$ j
END	Ende der Ausführung des Programms
STORE $*i$	`R(Inhalt(R(i)))` $\leftarrow$ `Inhalt(REG(1))`
	`R(0)` $\leftarrow$ `Inhalt(R(0))` + 1
LOAD1 $*j$	`REG(1)` $\leftarrow$ `R(Inhalt(R(j)))`
	`R(0)` $\leftarrow$ `Inhalt(R(0))` + 1

Tabelle 6.2 Befehlstabelle

Modul III

Entwurf von endlichen Automaten

Zielsetzung

Endliche Automaten sind überall. Straßenkreuzungen mit Ampeln, Getränkeautomaten, Fußgängerampeln, Fahrstühle – alle werden durch endliche Automaten gesteuert. Auch ein Teil eines Compilers ist ein endlicher Automat. In diesem Modul erfährt man, wie man solche Steuerungsmechanismen in Form von Automaten entwirft. Dabei erlernt man nicht nur das systematische Vorgehen für den Entwurf von Automaten, sondern auch die entworfenen Produkte hinsichtlich ihrer Korrektheit zu überprüfen und über das Erfüllen von Praxisanforderungen und die dadurch verursachten Produktionskosten nachzudenken.

Endliche Automaten sind das einfachste Berechnungsmodell, das man in der Informatik betrachtet. Im Prinzip entsprechen endliche Automaten speziellen Programmen, die konkrete Entscheidungsprobleme lösen und dabei keine Variablen benutzen. Endliche Automaten arbeiten in Echtzeit, denn sie lesen entweder eine Eingabe nur einmal von links nach rechts oder empfangen externe Signale nur einmal hintereinander. Das Resultat steht sofort nach Lesen des letzten Buchstabens oder nach Empfang des letzten Signals fest.

Das Hauptziel dieses Moduls ist das Erlernen der Methoden des Entwurfs von endlichen Automaten in zwei Schritten. Zuerst lernen wir, relativ einfache Automaten zu erzeugen, indem jedem Zustand eines Automaten eine Bedeutung zugeordnet wird. Für Aufgaben, die durch komplexere Bedingungen formuliert werden, entwickeln wir einen modularen Ansatz mit strukturiertem Vorgehen. Er benutzt einfache Automaten als Bausteine, um einen komplexeren endlichen Automaten mit den gewünschten Eigenschaften zu erzeugen.

Der Grund, endliche Automaten hier zu betrachten, ist nicht nur der Automatenentwurf für Aufgabenstellungen aus der Praxis und der Einstieg in die Automatentheorie. Man nutzt endliche Automaten auch für didaktische Zwecke, um auf einfache und anschauliche Weise die Modellierung von Berechnungen zu erläutern. Dazu führen wir einige Grundbegriffe der Informatik wie Konfiguration, Berechnungsschritt, Berechnung, Verifikation und Simulation ein.

Wir lernen in diesem Kapitel, wie man eine Teilklasse von Algorithmen (Programmen) formal und dabei anschaulich modellieren und untersuchen kann. Wir werden damit verstehen, wie man einen Automaten auf Korrektheit überprüfen kann. Gleichzeitig üben wir dabei die Induktionsbeweise. Neben dem ersten Kontakt mit den oben erwähnten Grundbegriffen lernen wir auch, einen Beweis zu führen, der zeigt, dass eine konkrete Aufgabe mit einer gegebenen Teilklasse von Algorithmen nicht lösbar ist. Gezeigt wird hier zum Beispiel, dass gewisse Aufgabenstellungen von keinem endlichen Automaten gelöst werden können und jeder Automat, der eine bestimmte Aufgabe löst, eine gewisse Mindestgröße haben muss.

Hinweis für die Lehrperson Das Minimalziel dieses Moduls sollte sein, mit ihm den Automatenentwurf zu unterrichten. Verwenden Sie dazu die Lektionen 1, 2, 3 und 6. Die Lektion 1 ist eine terminologische Vorbereitung. Lektion 2 zeigt, wie man endliche Automaten modellieren und darstellen kann. Die Lektionen 3 und 6 beinhalten zwei systematische Methoden zum Automatenentwurf. Lektion 4 ist eine gute, optionale Erweiterung des minimalen Ziels. Die Besonderheit liegt hier darin, dass man hiermit auch das Modellieren von realen Problemsituationen übt und sich weniger mit mathematischem Formalismus beschäftigt. So wirkt dieser Teil entspannend und erfrischend. Zudem wechselt man die Unterrichtsmethode und geht zu selbständigeren Projekten und ihrer Präsentation über.

Eine optionale Erweiterung ist ebenfalls die Lektion 5. Die Vorteile liegen einerseits in der Vertiefung der Fähigkeit, Induktionsbeweise zu führen und dadurch die mathematische Denkweise zu stärken. Andererseits entdeckt man, wie wichtig es ist, die entworfenen, technischen Produkte auf ihre korrekte Funktion zu überprüfen und erlernt die entsprechende Methodik dafür.

Die Lektion 7 ist der schwierigste Teil und macht einen weiteren Vertiefungsschritt in Richtung korrekter, mathematischer Argumentation. Hier geht es um Nichtexistenzbeweise. Zuerst zeigen wir, dass für manche Aufgaben keine kleinen, sondern nur große Automaten existieren. Danach stellen wir eine Beweismethode vor, die uns hilft zu zeigen, dass konkrete Aufgaben mit keinem Automaten lösbar sind. Zielsetzungen dieser Art gehören selten zum Gymnasialstoff. Die endlichen Automaten sind aber relativ einfach und bieten uns damit einen verständlichen Einstieg in die Beweisführung über die Existenz von Objekten mit gegebenen Eigenschaften. Dabei werden wieder direkte und indirekte Beweise geführt und der Umgang mit dem Allgemeinem in unendlich vielen Fällen ist auch aus der Überlegung nicht wettzumachen. Dieser Teil ist für hochmotivierte Schüler gut zu bewältigen. Es besteht auch die Möglichkeit, nur den ersten Teil der Lektion 7 zu benutzen, mit dem man die Mindestgröße von Automaten für gegebene Sprachen beweisen kann. Dort betrachtet man nicht das Unendliche und erfahrungsgemäß ist der Lernstoff für die Klasse nicht viel schwieriger als der Automatenentwurf.

Lektion 1

Alphabete, Wörter und Sprachen

Im Allgemeinen verarbeitet jeder Rechner Texte, die man als Folge von Buchstaben aus einem bestimmten Alphabet ansieht. Die Eingaben sind Texte, die konkrete Probleminstanzen (Aufgabenstellungen) beschreiben. Auch die Ausgaben sind als Texte dargestellt. Ein Rechner arbeitet ebenfalls mit Texten, weil alle Speicherinhalte (Registerinhalte) als Folgen der Buchstaben 0 und 1 anzusehen sind und alle Rechenbefehle diese Bitfolgen in neue Bitfolgen umwandeln. Das Ziel dieses Abschnitts ist es, die Fachterminologie für die textliche (symbolische) Darstellung der Information festzulegen, um am Ende die Informationsverarbeitung als eine Textverarbeitung anzuschauen.

Wir wollen die Daten und betrachteten Objekte durch Symbolfolgen darstellen. Genau wie bei der Entwicklung einer natürlichen Sprache fängt man mit der Festlegung von Symbolen an, die die Atome unserer Datenrepräsentation sind. Im Folgenden bezeichnet $\mathbb{N} = \{0, 1, 2, 3, \ldots\}$ die Menge der natürlichen Zahlen.

Definition 1.1 *Eine endliche, nichtleere Menge* Σ *heißt* **Alphabet**. *Die Elemente eines Alphabets werden* **Buchstaben (Zeichen, Symbole)** *genannt.*

Definition 1.1 besagt nichts anderes, als dass man das Alphabet frei wählen darf. Die einzigen Restriktionen sind, dass man mindestens einen Buchstaben im Alphabet haben muss und man nur endlich viele, unterschiedliche Buchstaben nehmen darf. Dies entspricht auch der Entwicklung einer natürlichen Sprache. Daher ist zum Beispiel die einelementige Menge $\{0\}$ ein Alphabet, aber die Menge $\mathbb{N}$ ist wegen ihrer unendlichen Mächtigkeit kein Alphabet. Deswegen sind für uns Zahlen keine Symbole, sondern Folgen von Symbolen, die wir Ziffern nennen.

J. Hromkovič, *Lehrbuch Informatik*, DOI 10.1007/978-3-8348-9692-6_22,

Einige der hier am häufigsten benutzten Alphabete sind:

- $\Sigma_{\text{Bool}} = \{0,1\}$ ist das Boolsche Alphabet, mit dem die Rechner arbeiten.
- $\Sigma_{\text{lat}} = \{a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z\}$ ist das Alphabet der kleinen lateinischen Buchstaben
- $\Sigma_{\text{dez}} = \{0,1,2,3,4,5,6,7,8,9\}$ ist das Alphabet der Ziffern, die wir zur dezimalen Darstellung von Zahlen verwenden können.
- $\Sigma_m = \{0,1,2,3,\ldots,m-1\}$ für jedes $m \geq 1$ ist ein Alphabet für die m-adische Darstellung von Zahlen.
- $\Sigma_{\text{greek}} = \{\alpha,\beta,\gamma,\delta,\varepsilon,\zeta,\eta,\theta,\gamma,\kappa,\lambda,\mu,\nu,\xi,o,\pi,\rho,\sigma,\tau,\upsilon,\phi,\chi,\psi,\omega\}$ ist das Alphabet der kleinen griechischen Buchstaben.
- $\Sigma_{\text{Klam}} = \{\{,\},[,],(,)\}$ ist das Alphabet der Klammern.
- $\Sigma_{\text{Geo}} = \{\triangle,\square,\circ\}$

Aufgabe 1.1 Welche der folgenden Mengen sind Alphabete? Begründe deine Antwort!

a) $\mathbb{N}_{\text{ger}} = \{0,2,4,\ldots\}$ als die Menge der geraden natürlichen Zahlen

b) $\{a,b,c,A,B,C\}$

c) $\{1,2,3,a,b,\triangle,\heartsuit\}$

d) $\emptyset$

Im Folgenden definieren wir den Begriff „Wort“ als eine Folge von Buchstaben aus einem gegebenen Alphabet. Der Begriff „Wort“ entspricht in der Fachsprache der Informatik einem beliebigen Text und nicht nur der Bedeutung von „Wort“ in der natürlichen Sprache.

Definition 1.2 *Sei Σ ein Alphabet. Ein* **Wort über** Σ *ist eine endliche (eventuell leere) Folge von Buchstaben aus Σ. Das* **leere Wort** λ *ist die leere Buchstabenfolge.*

Halten wir fest: Der Fachbegriff „Wort“ hat ohne Bezug zu einem Alphabet keine Bedeutung. Was wollen wir damit sagen? Wenn man über Wörter spricht, muss man zuerst das Alphabet festlegen und dann darf man von „Wörter über dem Alphabet“ sprechen. Zum Beispiel ist $01ab1a$ ein Wort über $\{0,1,a,b\}$, aber kein Wort über $\{0,1\}$.

In der Mathematik bezeichnet man üblicherweise die Folgen als $0,1,0,0,a,b,1$, indem man die Symbole aus dem Alphabet $\{0,1,a,b\}$ mit Kommata trennt. Hier verzichten wir auf die Kommata und schreiben anstatt 0,1,0,0,1,1 einfach nur 010011, wie es auch in natürlichen Sprachen der Fall ist. Ein weiterer Grund für diese verkürzte Schreibweise ist auch die häufige Verwendung des Kommasymbols „,“ als Alphabetsymbol, denn das könnte zu Missverständnissen führen. Zum Beispiel ist 01,10,0 ein Wort über $\{0,1,,\}$.

Aufgabe 1.2 Schreibe zu folgenden Symbolfolgen jeweils das kleinste Alphabet Σ auf, so dass die Symbolfolge ein Wort über Σ ist.

a) *abbabb*

b) $01000,(00)!$

c) 1111111

d) *aXYabuvRS*

Beachte: Wörter sind immer endliche Folgen von Buchstaben. Somit ist die unendliche Folge $1111\ldots$ kein Wort über dem Alphabet $\{1\}$.

Die **Länge** $|w|$ eines Wortes w über ein Σ ist die Länge des Wortes als Folge, d. h. die Anzahl der Buchstaben der Folge. Somit ist $|01a2b1| = 6$ für das Wort $01a2b1$ über dem Alphabet $\{0,1,2,a,b\}$. Für das leere Wort λ gilt $|\lambda| = 0$, weil λ keinen Buchstaben enthält.

Das größte der häufig verwendeten Alphabete ist Σ_{Tast}, das alle Symbole der Rechnertastatur beinhaltet. Somit gehören alle kleinen und großen Buchstaben des lateinischen Alphabets und alle möglichen Sonderzeichen wie @, $, ç, +, -, :, !, ? usw. dort hinzu. Zu Σ_{Tast} gehört auch das Leerzeichen, das wir als $\sqcup$ oder durch einen leeren Abstand darstellen können. Wenn man ein Leerzeichen $\sqcup$ im Alphabet Σ_{Tast} hat, dann kann man jeden Satz wie z.B.

> Kryptographie ist faszinierend.

als ein Wort über Σ_{Tast} betrachten. Die zwei Leerzeichen und der Punkt in diesem Satz zählen als Symbole und somit gilt

$$|\text{Kryptographie ist faszinierend.}| = 31.$$

So gesehen ist jeder deutschsprachige Text ein Wort über Σ_{Tast}. Somit ist der Textinhalt eines Buches auch nur als ein Wort über Σ_{Tast} zu betrachten. Wir sehen also, dass in der Informatik ein „Wort über einem Alphabet“ dem entspricht, was man in der natürlichen Sprache unter „Wort“, „Satz“ und „Text“ versteht.

Um es zu verdeutlichen, betrachten wir zuerst eine natürliche Sprache wie Deutsch, die auf dem lateinischen Alphabet basiert. Wenn wir nicht gerade griechische Buchstaben in mathematischen Texten verwenden, reicht uns Σ_{Tast} zur Herstellung aller möglichen Texte auf Deutsch aus. Wir können also mit dem festen Alphabet Σ_{Tast} ewig auskommen, auch wenn sich die Sprache weiterentwickelt. Wenn man Bedarf nach neuen Begriffen[1] hat, dann führt man neue Wörter in das Vokabular ein, die aus lateinischen Buchstaben zusammengesetzt sind. Ein festes Alphabet ist auf Dauer daher keine Behinderung für die Entwicklung einer natürlichen Sprache. Bei den Bildsprachen, wie z.B. Chinesisch, ist es anders. Da erzeugt man für jeden neuen Begriff (jedes neue Wort im Wörterbuch) ein neues Zeichen. Das Zeichen versteht man dabei als ein neues Symbol des Alphabets. Deswegen wächst das Alphabet mit der Entwicklung der Bildssprache immer weiter. Zu einem festen Zeitpunkt ist das Alphabet aber immer endlich und kann zum Schreiben beliebiger Texte über dem bestehenden Alphabet verwendet werden.

Aufgabe 1.3 Welches Alphabet verwendet man zur dezimalen Darstellung der natürlichen Zahlen? Wie hängt die Größe einer Zahl mit der Länge ihrer Dezimaldarstellung zusammen?

In der Informatik arbeiten wir immer mit festen Alphabeten, typischerweise mit Σ_{Bool} oder Σ_{Tast}. Die Bedeutung (die Semantik) einzelner Symbole ist nicht festgelegt. Was ein Symbol für uns unter gegebenen Umständen bedeutet, ist unsere Entscheidung. Beispiel: In einer Situation kann das Wort 11010 die binäre Darstellung der Zahl 26 sein, in einer anderen die Beschreibung eines Objekts oder die Kodierung einer Rechenoperation, wie etwa $+$.

Für endliche Automaten, die bei Lift- und Kreuzungssteuerungen eingesetzt werden, nutzen wir Symbole eines selbstgewählten Alphabets, um mit der Außenwelt zu „kommunizieren“. Kommunizieren bedeutet hier meistens, Signale zu empfangen. Für die

[1] und dieser Bedarf ist ständig vorhanden

Steuerung einer Ampel können wir an einem Fußgängerüberweg beispielsweise das Symbol „*a*“ verwenden, um dem Automaten mitzuteilen, dass Fußgänger die Straße überqueren möchten. Das Symbol „*b*“ könnte hingegen verwendet werden, um dem Automaten mitzuteilen, dass kein Fußgänger über die Straße gehen möchte. Genauso gut können wir für diesen Zweck die Symbole „0“ und „1“ oder beliebige andere verwenden.

An der T-Kreuzung in Abb. 1.1 wären drei Ampeln A_1, A_2 und A_3 für Autos denkbar. Jede

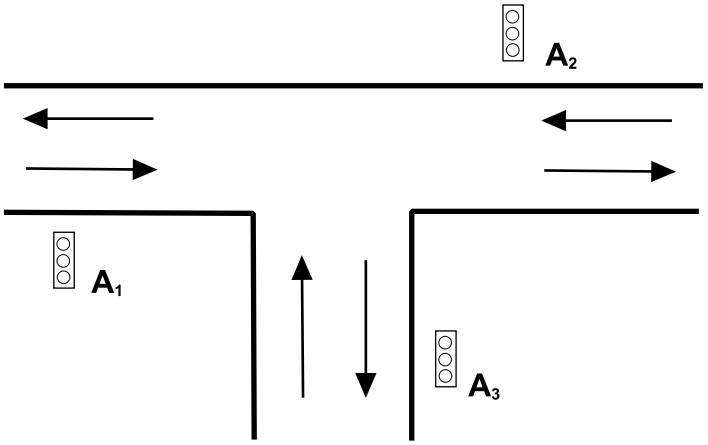


Abbildung 1.1

könnte mit einer Kamera ausgestattet sein, die signalisiert, ob auf der entsprechenden Seite Fahrzeuge über die Kreuzung fahren wollen. Jetzt könnte jemand sagen: „Mit dem Symbol (Signal) ‘*a*’ bezeichne ich die Situation, in der an A_1 Fahrzeuge vorhanden sind und an A_2 und A_3 keine Fahrzeuge in Sicht sind.“ Weitere mögliche Situationen könnte man dann mit anderen Buchstaben des Alphabets beschreiben. Man dürfte es so machen, hätte aber keine große Transparenz, weil die Bezeichnung keinen Bezug zu der tatsächlichen Verkehrssituation hat. Wir wählen daher besser Tripel

$$(x_1, x_2, x_3)$$

als Symbole mit $x_1, x_2, x_3 \in \{0, 1\}$. Wenn $x_1 = 1$ ist, sind Fahrzeuge an der Ampel A_1 vorhanden. Ist $x_1 = 0$, gibt es auf der Seite der Ampel A_1 keine Fahrzeuge. Analog vergibt man die Bedeutung für x_2 und x_3 für die Ampeln A_2 und A_3. Somit bedeutet das Symbol

$$(1, 1, 1),$$

dass überall Fahrzeuge vorhanden sind. Das Symbol

$$(1,0,1)$$

bedeutet, dass es nur Fahrzeuge auf den Seiten der Ampeln A_1 und A_3 gibt. Bei dieser übersichtlichen Darstellung der Signale sehen wir auch sofort, wie viele unterschiedliche Signale es geben kann und wie groß somit unser Alphabet sein muss. Unsere Symbole sind Tripel, und jedes Element kann entweder den Wert „0“ oder „1“ annehmen. Somit gibt es genau $2 \cdot 2 \cdot 2 = 8$ Symbole und unser Alphabet lautet:

$$\Sigma_{\text{Kr}} = \{(0,0,0),(0,0,1),(0,1,0),(1,0,0),(0,1,1),(1,0,1),(1,1,0),(1,1,1)\}.$$

Beachte, dass bei dieser Festlegung von Σ_{Kr} ein Tripel wie z.B. $(0,1,0)$ als ein Symbol zu betrachten ist und nicht als ein Wort über $\{0,1,(,),,\}$ angesehen werden darf.

Aufgabe 1.4 Angenommen man errichtet an der T-Kreuzung in Abb. 1.1 noch drei Fußgängerüberwege mit je einer Ampel, die wir B_1, B_2 und B_3 nennen. Jede B_i hat einen Knopf. Wenn er gedrückt wird, signalisiert ein Fußgänger, dass er die entsprechende Straße überqueren will.

a) Erweitere das Bild in Abb. 1.1 durch das Einzeichnen der Fußgängerwege und der neuen Ampeln B_1, B_2 und B_3.

b) Wähle geeignete Symbole für alle möglichen Signale (Situationen) aus, die auftreten können und erkläre ihre Bedeutung. Schreibe das komplette, so entstandene Alphabet auf.

Mit $|w|_a$ werden wir die Häufigkeit des Symbols a in dem Wort w bezeichnen. Somit ist

$$|abb01aa|_a = 3,$$

weil in dem Wort $abb01aa$ das Symbol a genau dreimal vorkommt: an der ersten und an den beiden letzten Stellen.

Aufgabe 1.5 Bestimme die Werte für $|00110|_0$, $|ABabc|_A$, $|(a+b)*d-7*a|_a$, wobei alle vorhandene Wörter über Σ_{Tast} sein sollen.

Hinweis für die Lehrperson Die folgenden Aufgaben sind nur für diejenigen bestimmt, die schon die Grundlagen der Kombinatorik absolviert haben.

Aufgabe 1.6 Wie viele Wörter mit der folgenden Eigenschaft gibt es? Begründe deine Antwort!

a) alle Wörter über $\Sigma_5 = \{0,1,2,3,4\}$ der Länge 7,

b) alle Wörter über $\Sigma_{\text{Bool}} = \{0,1\}$ der Länge n für eine positive ganze Zahl n,

c) alle Wörter über $\Sigma = \{a,b,c,d\}$ der Länge 8, in denen jedes Symbol genau zweimal vorkommt,

d) alle Wörter w über $\Sigma_3 = \{0,1,2\}$ der Länge 8, die $|w|_0 = 4$ und $|w|_1 = 1$ erfüllen,

e) alle Wörter w über $\Sigma_{\text{Bool}} = \{0,1\}$ der Länge 10, die mindestens so viele Symbole „1“ wie „0“ enthalten ($|w|_1 \geq |w|_0$),

f) alle Wörter w über $\{a,b,c\}$, für die $|w| \leq 6$ gilt,

g) alle Wörter w über $\{a,b,c\}$, für die $|w| = 10$ und $|w|_a \geq |w|_b + |w|_c$ gilt.

Im Folgenden benutzen wir die Bezeichnung Σ^* für die Menge aller Wörter über Σ.

$\Sigma^+ = \Sigma^* - \{\lambda\}$ bezeichnet die Menge aller Wörter über Σ mit Ausnahme des leeren Wortes λ. Wenn man jetzt

$$x \in \Sigma^*$$

schreibt, ist es äquivalent zu der Aussage „x ist ein Wort über Σ“. Zum Beispiel für das Alphabet $\Sigma_{\text{Bool}} = \{0,1\}$ erhalten wir

$$(\Sigma_{Bool})^* = \{\lambda, 0, 1, 00, 01, 10, 11, 000, 001, 010, 100, \ldots\}.$$

Für $\Sigma = \{a\}$ haben wir

$$\{a\}^* = \{\lambda, a, aa, aaa, aaaa, aaaaa, \ldots\}.$$

Für jedes Σ ist Σ^* eine unendliche Menge. Die Beispiele verdeutlichen, dass man alle Wörter der Länge 0, 1, 2, ... hintereinander schreiben kann, wenn man Wörter über einem Alphabet auflisten möchte.

Aufgabe 1.7 Liste die 31 kürzesten Wörter über dem Alphabet $\Sigma = \{a,b,c\}$ auf.

Üblicherweise verwenden wir Operationen (wie +, *, -, /) über Zahlen. Es gibt neben den Zahlen auch andere Objekte, über die man Operationen ausführen kann. Die am häufigsten angewendete Operation über Wörtern ist die **Konkatenation** (Verkettung). Die Konkatenation von zwei Wörtern x und y über einem Alphabet Σ ist

$$\text{Konkatenation}(x,y) = xy\ ,$$

also nichts anderes als das Hintereinanderschreiben der Wörter x und y.

Zum Beispiel für $x = 01ab11$ und $y = ab00$ ist

$$\text{Konkatentation}(x,y) = xy = 01ab11ab00.$$

Offensichtlich gilt $|xy| = |x| + |y|$ für alle Wörter x und y. $\text{Konkatenation}(x,\lambda) = x\lambda = x$ für alle Wörter, d. h. die Konkatenation eines Wortes x mit λ verändert das Wort x nicht.

Aufgabe 1.8 Beweise, dass im Allgemeinen die Konkatenation keine kommutative Operation über Wörter über ein gegebenes Alphabet ist! Gibt es ein Alphabet, so dass $xy = yx$ für alle $x,y \in \Sigma^*$ ist? Ist die Konkatenation eine assoziative Operation?

Wir benutzen die Operation der Konkatenation zur kürzeren Darstellung von Wörtern. Für alle Wörter x über einem Alphabet Σ und alle $i \in \mathbb{N}$ definieren wir die Wortpotenzen

$$x^0 = \lambda,$$
$$x^1 = x \text{ und}$$
$$x^i = xx^{i-1}.$$

Wortpotenzen schreibt man in Klammern, um das zu wiederholende Wort zu verdeutlichen. Ein Beispiel: Für $x = ababb$ schreiben wir

$$x^3 = (ababb)^3 = ababb(ababb)^2 = ababbababbababb$$

Man kann Wortpotenzen folgendermaßen einsetzen, um Wörter verkürzt darzustellen:

$$aaabbbbaabbaabb = a^3b^4a^2b^2a^2b^2 = a^3b^4(a^2b^2)^2$$

Aufgabe 1.9 Verwende Potenzen, um die folgenden Wörter so kurz wie möglich darzustellen.

a) *bbbbababababaaaaa*

b) 0*a*11100*a*11100000*aaaa*

c) *YESYESYESIAGREE*

Für die Arbeit mit Wörtern sind die folgenden Begriffe von zentraler Bedeutung. Dabei seien v und w zwei Wörter über dem gleichen Alphabet Σ.

Wir sagen, dass v **ein Teilwort von** w ist, wenn man w als $w = xvy$ für beliebige Wörter[2] x und y über Σ schreiben kann, also wenn v ein zusammenhängender Teil des Wortes w ist (Abb. 1.2).

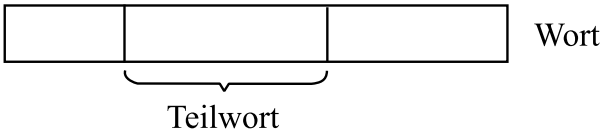


Abbildung 1.2

Zum Beispiel ist *abba* ein Teilwort des Wortes $x = bb\mathbf{abba}ab$. Alle Teilworte dieses Wortes x der Länge 3 sind: bba, bab, abb, baa und aab.

Beachte, dass das Teilwort *bba* zweimal als Teilwort an unterschiedlichen Stellen des Wortes **bbabba***ab* vorkommt. Dies ist eher typisch als außergewöhnlich. Das Wort *a* ist in x als Teilwort dreimal vorhanden und das Teilwort *b* findet man sogar fünfmal in x.

Aufgabe 1.10 Liste alle Teilwörter des Wortes 01110101 der Länge 2 und der Länge 4 auf.

Der Definition des Teilwortes folgend ist x selbst ein Teilwort des Wortes x. Somit ist z.B. *abba* ein Teilwort des Wortes *abba*. Wenn wir diese Möglichkeit ausschließen wollen, sprechen wir von **echten** Teilwörtern des Wortes x. Ein echtes Teilwort von x ist jedes Wort von x, das kürzer als x ist.

[2] x und y dürfen auch leere Wörter sein.

Aufgabe 1.11 Liste alle echten Teilwörter des Wortes auf:

a) *abbba*

b) 1111

c) 012013

Wir führen weiterhin besondere Bezeichnungen für die Teilwörter ein, mit dem ein Wort anfängt oder endet. Ein Wort y bezeichnet man als ein **Suffix** des Wortes x, wenn man x als $x = uy$ für irgendein Wort u schreiben kann (Abb. 1.3).

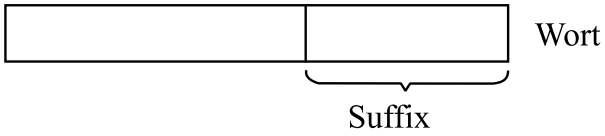


Abbildung 1.3

Somit sind 00011, 0011, 011, 11, 1 und λ Suffixe des Wortes $x = 00011$. Mit Ausnahme des Wortes 00011 selbst sind alle echte Suffixe.

Für beliebige Wörter x und w über einem Alphabet Σ ist w ein **Präfix** von x, wenn man x als $x = wv$ für irgendein Wort v über Σ schreiben kann (Abb. 1.4).

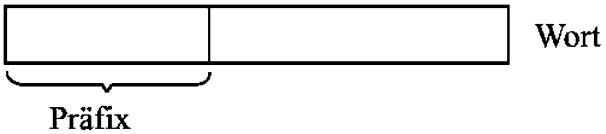


Abbildung 1.4

Damit sind die Wörter $abbab, abba, abb, ab, a$ und λ Präfixe des Wortes $x = abbab$.

Aufgabe 1.12 Ein Wort ist gleichzeitig ein Präfix und ein Suffix des Wortes $x = ABBA$. Um welches Wort handelt es sich?

Aufgabe 1.13 Wie viele Präfixe und wie viele Suffixe hat ein Wort der Länge n?

Aufgabe 1.14 Finde das kürzeste Wort, das folgende Teilwörter besitzt:

a) aa, ab, ba, bb

b) $001, 010, 100, 101$

c) alle Wörter der Länge 3 über $\{0,1\}$

Aufgabe 1.15 * Wie viele Teilwörter kann ein Wort der Länge n höchstens haben? Welche Wörter über welchem Alphabet haben die wenigsten Teilwörter? Welche Wörter haben die meisten Teilwörter?

Eine Menge von Wörter über einem Alphabet Σ bezeichnen wir als eine **Sprache über** Σ. Doch wofür braucht man den Begriff der Sprache? Wörter verwenden wir üblicherweise zur Darstellung gewisser Objekte, wie z.B. Zahlen, Graphen, Diagramme und Texte. Eine Sprache kann dazu dienen, die Darstellung von Objekten mit einer gegebenen Eigenschaft in eine Klasse einzuordnen. Nennen wir ein paar Beispiele für Sprachen. $Zahl(x)$ sei die Zahl, die durch $x \in \{0,1\}^*$ binär dargestellt wird.

- $L_{\text{prim}} = \{x \in (\Sigma_{Bool})^* \mid Zahl(x) \text{ ist eine Primzahl}\}$
  = die Menge aller Wörter über Σ_{Bool}, die eine Primzahl darstellen

- L_{Deutsch} = die Menge aller Wörter über Σ_{Tast}, die einen grammatikalisch korrekten, deutschen Text darstellen

- $\{xabby \in \Sigma^* \mid x, y \in \Sigma^*\}$
  = die Menge aller Wörter über Σ, die das Teilwort abb enthalten

- $\{ACCTTAx \mid x \in \{A,C,T,G\}^*\}$
  = die Folge aller Darstellungen von DNA-Sequenzen, die mit dem Präfix ACCTTA anfangen

- die Menge aller Wörter über dem Münzenalphabet $\{M_{0.10}, M_{0.20}, M_{0.50}, M_{1.00}, M_{2.00}\}$, deren Wertsumme genau 2 ergibt

Wir sehen, dass alle bis auf die letzte der vorgestellten Sprachen unendlich sind. Die leere Menge ist auch als eine Sprache anzusehen. Ebenso ist Σ^* als die Sprache aller Wörter über Σ zu betrachten.

Aufgabe 1.16 Entscheide, welche der folgenden Sprachen endlich und welche unendlich sind. Schreibe für jede dieser Sprachen ein Wort über dem entsprechenden Alphabet auf, das nicht in der Sprache liegt.

a) $L = \{x \in (\Sigma_{\text{Bool}})^* \mid |x|_0 \leq 3 \text{ und } |x|_1 \in \{1,2\}\}$,

b) $L = \{wabba \in \{a,b\}^* \mid w \in \{a,b\}^* \text{ und } |w|_a \leq 3\}$,

c) $L = \{0^p \mid p \text{ ist eine Primzahl}\}$,

d) $L = \{a^n b^n \mid n \in \mathbb{N}\}$,

e) $L = \{xabbay \mid x \in \{a,b\}^* \text{ und } |x| = 2 \text{ und } |y|_a = 1 \text{ und } |y|_b \leq 2\}$.

Aufgabe 1.17 Liste jeweils die ersten zehn kürzesten Wörter aus den Sprachen der Aufgabe 1.16 auf.

Den Begriff der Sprache verwenden wir, um die einfachsten der grundlegenden algorithmischen Aufgabentypen zu beschreiben.

Definition 1.3 *Sei L eine Sprache über einem Alphabet Σ. Das* **Entscheidungsproblem** *(Σ, L) ist die folgende algorithmische Aufgabe:*

Eingabe: *ein Wort $x \in \Sigma^*$*
Ausgabe: *JA falls $x \in L$*
NEIN falls $x \notin L$.

Wenn man also $(\{0,1\}, L_{\text{prim}})$ betrachtet, gilt es zu entscheiden, ob eine Zahl in binärer Darstellung einer Primzahl entspricht oder nicht. Beim Entscheidungsproblem $(\Sigma_{\text{Tast}}, L_{\text{Deutsch}})$ geht es um die Überprüfung, ob ein gegebenes Wort über Σ_{Tast} ein grammatikalisch korrekter Text auf Deutsch ist.

Im Folgenden benutzen wir zusätzlich die Begriffe des kartesischen Produktes und der Relation.

Seien A und B zwei beliebige Mengen. Das **kartesische Produkt von A und B** ist

$$\mathbf{A \times B} = \{(a,b) \mid a \in A \text{ und } b \in B\}$$

die Menge aller Paare mit dem ersten Element aus A und dem zweiten Element aus B.

Beispiel: Für $A = \{1,2,3,7\}$ und $B = \{a,b,c\}$ ist

$$\begin{aligned} A \times B = \quad & \{(1,a),(1,b),(1,c),(2,a),(2,b),(2,c), \\ & (3,a),(3,b),(3,c),(4,a),(4,b),(4,c)\}. \end{aligned}$$

Zum Beispiel $(a,1)$ ist nicht in $A \times B$ (dafür aber in $B \times A$), weil in jedem Paar (x,y) aus $A \times B$ das Element x aus A und das Element y aus B sein muss.

Aufgabe 1.18 Liste alle Elemente der Menge $B \times A$ auf.

Es kann auch sein, dass $A = B$ ist, man also das kartesische Produkt einer Menge mit sich selbst betrachtet. Zum Beispiel:

$$\mathbb{N} \times \mathbb{N} = \{(i,j) \mid i,j \in \mathbb{N}\}.$$

Das kartesische Produkt zweier Mengen A und B beinhaltet als Paare alle Kombinationen der Elemente aus A mit den Elementen aus B. Manchmal interessieren uns nur einige Paare, die für uns eine besondere semantische Bedeutung haben. Zu diesem Zweck führen wir den Begriff der Relation ein. Eine **Relation** R **von** A **nach** B ist eine beliebige Teilmenge von $A \times B$. Wenn $R \subseteq A \times A$ gilt (also wenn $A = B$), sprechen wir von einer **Relation R auf A**. Beachte: R kann auch eine leere Menge sein oder es kann $R = A \times B$ gelten.

Sei A die Menge aller Männer und B die Menge aller Frauen und R_{verh} die Relation „verheiratet sein“. Dann ist

$$R_{\text{verh}} = \{(a,b) \mid a \in A, b \in B \text{ und } a \text{ ist mit b verheiratet}\}$$

eine Relation von A nach B. Offensichtlich gibt es ledige Personen, die in keinem Paar der Relation R_{verh} vorkommen. In christlichen Ländern kommt jede Person, egal ob Mann oder Frau, in höchstens einem Paar von R_{verh} vor. In der muslimischen Welt kann ein Mann in mehreren Paaren von R_{verh} vorhanden sein. Es gibt auch Kulturen, in denen es umgekehrt ist und eine Frau in mehreren Paaren von R_{verh} vorkommt.

Eine Relation R auf A heißt **reflexiv**, wenn für alle $a \in A$ das Paar (a,a) in R ist (d. h. $(a,a) \in R$ gilt). Eine Relation R auf A heißt **symmetrisch**, wenn $(a,b) \in R$ impliziert,

dass auch $(b,a) \in R$ gilt. Mit anderen Worten: Für alle Elemente a und b aus A gilt: Entweder sind beide Paare (a,b) und (b,a) Elemente von R oder keines der Paare ist Element von R.

Aufgabe 1.19 Sei A die Menge aller Menschen. Betrachten wir die Relation R_{verh} auf A. Ist die Relation reflexiv? Ist R_{verh} symmetrisch?

Eine Relation R auf A ($R \subseteq A \times A$) heißt **transitiv**, wenn für alle $a,b,c \in R$ folgendes gilt:

$$(a,b),(b,c) \in R \text{ impliziert } (a,c) \in R.$$

In Worten ausgedrückt: Eine Relation R ist transitiv, wenn die relationale Verbindung zwischen a und b und zwischen b und c auch automatisch die relationale Verbindung zwischen a und c als Konsequenz hat.

Ein Beispiel ist die Relation der Verwandtschaft auf der Menge der Menschen. Wenn eine Person a mit einer Person b verwandt ist ($(a,b) \in R$) und b ist verwandt mit c ($(b,c) \in R$), dann ist auch a mit c verwandt.

Aufgabe 1.20 Sei A eine Menge aller Städte in Australien. Betrachte die Relation „über das Straßenverkehrsnetz verbunden sein“. Dies bedeutet, dass zwei Städte a und b in der Relation R sind ($(a,b) \in R$), wenn man von a nach b über die Straßen des Netzes fahren kann. Welche Eigenschaften hat diese Relation? Könnten sich eine oder mehrere dieser Eigenschaften ändern, wenn man statt Australien einen anderen Kontinent auswählen würde?

Beispiel 1.1 Sei A die Menge aller Vögel. Sei $R \subseteq A \times A$ die Relation „bunter zu sein als“ auf A ist definiert durch:

$$R = \{(a,b) \in A \times A \mid a \text{ ist bunter als } b\}.$$

Diese Relation ist nicht reflexiv, weil man nicht bunter als man selbst sein kann. Für keinen Vogel a gilt also $(a,a) \in R$. Die Relation R ist nicht symmetrisch, weil es gleichzeitig nicht möglich ist, dass a bunter als b ist ($(a,b) \in R$) und b bunter als a ist ($(b,a) \in R$). Offensichtlich ist R transitiv. Wenn a bunter als b ist ($(a,b) \in R$) und b bunter als c ist ($(b,c) \in R$), dann ist auch a bunter als c ($(a,c) \in R$). Was würde sich an den Eigenschaften ändern, wenn wir die Relation „bunter oder gleich bunt“ betrachten würden? □

Beispiel 1.2 Sei A die Menge aller Städte. Betrachten wir die Relation $Flug$ „a ist von b mit Fluglinien erreichbar“. Welche Eigenschaften hat diese Relation? Offensichtlich ist sie transitiv. Wenn b von a erreichbar ist ($(a,b) \in Flug$) und c von b erreichbar ist ($(b,c) \in Flug$), dann kann man sicherlich auch von a nach c über b fliegen und somit gehört (a,c) in $Flug$. Ist $Flug$ eine symmetrische Relation? In der Regel würden man es erwarten, aber es muss nicht sein. $Flug$ wäre nicht symmetrisch, wenn es eine Stadt d geben würde, aus der man in eine andere Stadt e fliegen könnte ($(d,e) \in Flug$) und es keine Möglichkeit gäbe, aus e über beliebig viele Umsteigestationen nach d zu gelangen.

Wie ist es mit der Reflexivität? Sicherlich kann man von a nach a auch ohne Fliegen kommen. Wenn man aber daran denkt, dass $(a,a) \in Flug$ nur dann gilt, wenn man von a nach a mit einer nichtleeren Folge von Flügen kommt, dann muss man sorgfältig überlegen. Ein Rundflug über a führt auch zu $(a,a) \in Flug$. Dank der Transitivität folgern wir aus $(a,b) \in Flug$ und $(b,a) \in Flug$, dass $(a,a) \in Flug$. Damit reicht für $(a,a) \in Flug$ die Existenz einer Stadt b mit einer Fluglinie zwischen a und b in beide Richtungen aus. Wir erwarten normalerweise eine reflexive Relation. Es muss aber nicht der Fall sein, nämlich dann, wenn es zum Beispiel eine Stadt d gäbe, von der man in unterschiedliche Richtungen abfliegen könnte, aber nicht landen dürfte. Das mag zu ungewöhnlich klingen. Viel realistischer ist aber, dass man zwischen den betrachteten Städten in A eine Stadt u hat, die keinen Flughafen hat. Damit kann (u,u) definitiv nicht in $Flug$ sein und die Relation $Flug$ wäre in einem solchen Fall nicht reflexiv. □

Aufgabe 1.21 Sei $A = \{a,b,c,d\}$ eine Menge von Städten. Es gibt folgende vier Fluglinien

$$a \to b, \quad b \to c, \quad c \to a, \quad b \to a$$

zwischen den Städten ($a \to b$ bedeutet, man kann direkt von a nach b fliegen). Betrachten wir jetzt die Relation $Flug$ „b ist von a durch Fluglinien verbunden“ aus Beispiel 1.2. Welche Eigenschaften hat in diesem Fall die Relation $Flug$? Schreibe alle Elemente von $Flug$ auf.

Aufgabe 1.22 Sei Σ ein Alphabet. Betrachten wir die Relation *Prä* auf Σ^* definiert durch $(u,v) \in$ *Prä* $\Leftrightarrow$ u ist ein Präfix von v. Welche Eigenschaften hat die Relation *Prä*?

Aufgabe 1.23 Sei $A = \{X,Y,Z\}$ und $B = \{\triangle, \square, \circ\}$. Schreibe alle Elemente des kartesischen Produktes $A \times B$ auf. Wie viele unterschiedliche Relationen von A nach B gibt es?

Die Beziehung $\leq$ kann man als eine Relation $R_{\leq}$ auf $\mathbb{N}$ ansehen. Es gilt:

$$R_{\leq} = \{(i,k) \mid i,j \in \mathbb{N}\ i \text{ ist kleiner oder gleich } j\} \subseteq \mathbb{N} \times \mathbb{N}.$$

Aufgabe 1.24 Sei $A = \{1,2,3,4\}$. Finde eine Relation $R_{A,<}$ auf A, die der Beziehung $<$ entspricht und zähle alle Elemente von $R_{A,<}$ auf.

Aufgabe 1.25 Eine Funktion von A nach B ist ein Spezialfall einer Relation von A nach B, d. h. jede Funktion kann man als eine Relation betrachten. Aber nicht jede Relation ist eine Funktion, wie auch unsere Beispiele zeigen. Für eine Funktion $f : A \to B$ kann man f in Form einer Relation R_f wie folgt darstellen:

$$R_f = \{(a, f(a)) \mid a \in A\}.$$

Sei $R \subseteq A \times B$ eine beliebige Relation von A nach B. Welche Eigenschaften muss R erfüllen, um sie als Funktion betrachten zu dürfen?

Zusammenfassung

Ein Alphabet ist eine endliche und nichtleere Menge von Symbolen (Buchstaben). Aus den Symbolen kann man Wörter, Sätze und Texte als Folgen von Buchstaben gestalten. In der Informatik nennen wir endliche Folgen von Symbolen aus einem Alphabet Wörter über dem Alphabet. Zusammenhängende Teile eines Wortes x nennen wir Teilwörter von x.

Eine Menge von Wörtern über einem Alphabet Σ nennen wir Sprache über Σ. Ein Entscheidungsproblem ist ein beliebiges Paar (Σ, L), wobei L eine Sprache über Σ ist. Ein Algorithmus löst ein Entscheidungsproblem (Σ, L), falls der Algorithmus für jedes Wort über Σ entscheiden kann, ob das Wort in der Sprache liegt oder nicht.

Ein kartesisches Produkt $A \times B$ zweier Mengen A und B ist die Menge aller Paare (a,b), wobei $a \in A$ und $b \in B$ ist. Eine beliebige Teilmenge R von $A \times B$ nennen wir Relation von A nach B. Eine Relation $R \subseteq A \times A$ nennen wir Relation auf A. Die Relation $R \subseteq A \times A$ ist reflexiv, wenn $(a,a) \in R$ für alle $a \in A$. Eine Relation R auf A ist symmetrisch, wenn $(a,b) \in R$ bedeutet, dass auch (b,a) in R vorhanden ist. Eine Relation R auf A ist transitiv, wenn $(a,b) \in R$ und $(b,c) \in R$ zusammen implizieren, dass auch $(a,c) \in R$.

Kontrollfragen

1. Was ist ein Alphabet? Wie nennt man die Elemente eines Alphabets?

2. Was ist ein Wort über einem Alphabet? Welche sprachwissenschaftlichen Grundbegriffe entsprechen in ihrer Bedeutung dem informatischen Fachterminus „Wort“?

3. Wieso kann jeder Text auf Deutsch als ein Wort betrachtet werden?

4. Wie lautet das Alphabet für die dezimale Darstellung von natürlichen Zahlen? Welches Alphabet kann man für rationale Zahlen verwenden?

5. Kann man jede reelle Zahl als ein Wort über dem Alphabet $\{0,1,2,3,4,5,6,7,8,9,.,+,-\}$ betrachten? Begründe deine Antwort.

6. Kann man jede positive ganze Zahl als ein Wort über dem Alphabet $\{0\}$ darstellen? Begründe deine Antwort!

7. Warum ist $\mathbb{N}$ kein Alphabet?

8. Was ist eine Sprache über einem Alphabet?

9. Was ist ein Entscheidungsproblem? Nenne ein Beispiel für ein interessantes Entscheidungsproblem! Wann sagt man, dass ein Algorithmus ein Entscheidungsproblem löst?

10. Welche Eigenschaften hat die Relation „kleiner gleich“ auf natürlichen Zahlen?

11. Welche Eigenschaften hat die Relation „gleich hoch zu sein“ auf der Menge aller Bäume?

12. Ist $\mathbb{N} \times \mathbb{N}$ eine Relation auf $\mathbb{N}$?

13. Welche Eigenschaften hat die Relation „ein Teilwort zu sein“ auf $\{0\}^*$?

14. Was ist die Konkatenation von zwei Wörtern?

15. Was ist ein kartesisches Produkt zweier Mengen A und B ?

16. Was ist eine Relation auf einer Menge? Wozu kann der Begriff der Relation nützlich sein?

Kontrollaufgaben

1. Gib das Alphabet für die Darstellung der römischen Zahlen an! Ist jedes Präfix einer römischen Zahl auch eine römische Zahl? Ist jedes Suffix einer römischen Zahl eine römische Zahl? Begründe deine Antworten!

2. Betrachte einen mit Ampeln ausgestatteten Fußgängerübergang auf einer Zweibahnstraße.

Wähle ein Alphabet aus, um alle möglichen Kombinationen von Steuersignalen darzustellen.

3. Finde ein Alphabet, mit dem man eindeutig jede quadratische Gleichung mit ganzzahligen Koeffizienten darstellen kann und erkläre dein Vorgehen!

4. Finde ein Alphabet, mit dem du eindeutig jede rationale Zahl als ein Wort darstellen kannst! Erkläre deine Darstellungsmethode!

5. Beantworte folgende Fragen:

   a) Ist 01101 ein Wort über dem Alphabet $\{0,1\}$?

   b) Sind *abbaab* und a^4 Wörter über dem Alphabet $\{a,b,c,d\}$?

   c) Sind die Wörter *abba* und *abbdca* aus $\{a,b,c\}^*$?

   d) Ist $\emptyset$ eine Sprache über $\{0,1\}$?

   e) Ist $\{\lambda\}$ eine Sprache über $\{0\}$?

   f) Gilt $|001011|_0 = |001011|_1$?

6. Beschreibe einen Algorithmus, der das Entscheidungsproblem $(\{0,1\}, L_{Prim})$ löst!

7. Sei Q die Menge der rationalen Zahlen. Welche Eigenschaften hat die Relation $Q \times Q$? Welche Eigenschaft hat die Relation „kleiner gleich“ auf Q? Hat die Relation „kleiner“ auf Q die gleiche Eigenschaft?

8. Sei $Suf \subseteq \{0,1\}^* \times \{0,1\}^*$ eine Relation auf $\{0,1\}^*$, die durch „$(u,v) \in Suf \Leftrightarrow u$ ist ein Suffix von v“ definiert wird. Welche Eigenschaften hat die Relation Suf?

9. Gib eine Relation $R \neq \emptyset$ auf $\mathbb{N}$ an, die folgende Eigenschaften hat:

   a) R ist reflexiv und transitiv, aber nicht symmetrisch,

   b) R ist transitiv, aber nicht reflexiv und nicht symmetrisch,

   c) R ist reflexiv, symmetrisch und transitiv,

   d) R ist symmetrisch, aber nicht reflexiv und nicht transitiv,

e) R ist symmetrisch und transitiv, aber nicht reflexiv.

Lösungen zu ausgesuchten Aufgaben

Aufgabe 1.1
$\mathbb{N}_{ger}$ ist kein Alphabet, weil diese Menge unendlich viele Elemente hat. Die leere Menge $\emptyset$ ist auch kein Alphabet, weil ein Alphabet mindestens ein Element haben muss. Die restlichen zwei Mengen sind Alphabete.

Aufgabe 1.2
Die Folge *abbabb* ist ein Wort über dem Alphabet $\{a,b\}$. Es gibt kein kleineres Alphabet, über dem man *abbabb* als Wort betrachten könnte. Die Folge $01000,(00)!$ ist ein Wort über dem Alphabet $\{0,1,(,),,,!\}$. Die Folge 1111111 ist ein Wort über dem Alphabet $\{1\}$. Die Symbolfolge *aXYabuvRS* ist ein Wort über dem Alphabet $\{a,b,u,v,X,Y,R,S\}$.

Aufgabe 1.6

a) Es gibt 5^7 Wörter der Länge 7 über einem 5-elementigen Alphabet. Diese Anzahl wird dadurch bestimmt, dass man für jede der 7 Positionen eines Wortes eine beliebige Wahl aus 5 Buchstaben hat.

b) Mit der gleichen Begründung wie bei (a) gibt es 2^n viele Wörter der Länge n über einem zweielementigen Alphabet.

c) Wir haben $\binom{8}{2}$ Möglichkeiten, um die zwei Positionen für das Symbol a in dem Wort der Länge 8 auszusuchen. Dadurch hat man $\binom{6}{2}$ Möglichkeiten, um zwei Symbole b an die restlichen 6 Positionen zu platzieren. Es bleiben noch 4 unbesetzte Positionen. Es gibt es noch $\binom{4}{2}$ Möglichkeiten, dort 2 Symbole e zu positionieren. Auf die restlichen zwei Positionen müssen dann die Symbole d kommen. Somit ist die Anzahl der Wörter der Länge 8 mit jeweils zwei Symbolen a,b,e und d genau

$$\left(\begin{array}{c} 8 \\ 2 \end{array}\right) \cdot \left(\begin{array}{c} 6 \\ 2 \end{array}\right) \cdot \left(\begin{array}{c} 4 \\ 2 \end{array}\right).$$

d) Wir zählen analog wie in (c) und erhalten

$$\left(\begin{array}{c} 8 \\ 4 \end{array}\right) \cdot \left(\begin{array}{c} 4 \\ 1 \end{array}\right)$$

für die entsprechende Anzahl von Wörter.

e) Die Anzahl der Wörter der Länge 10 mit i Symbolen 1 ist genau

$$\left(\begin{array}{c} 10 \\ i \end{array} \right)$$

Somit ist die Anzahl der Wörter der Länge 10 mit mindestens 5 Symbolen 1 genau

$$\left(\begin{array}{c} 10 \\ 5 \end{array} \right) + \left(\begin{array}{c} 10 \\ 6 \end{array} \right) + \left(\begin{array}{c} 10 \\ 7 \end{array} \right) + \left(\begin{array}{c} 10 \\ 8 \end{array} \right) + \left(\begin{array}{c} 10 \\ 9 \end{array} \right) + \left(\begin{array}{c} 10 \\ 10 \end{array} \right).$$

Aufgabe 1.9

a) $bbbbababababaaaaa = b^4(ab)^3a^5$

b) $0a11100a11100000aaaa = 0(a11100)^20^3a^4$

Aufgabe 1.12
Das Wort A ist gleichzeitig ein Präfix und ein Suffix des Wortes $ABBA$.

Aufgabe 1.15
Die meisten Teilwörter haben die Wörter, in denen keine Buchstaben zweimal vorkommen. Damit ist jedes Stück des Wortes zwischen zwischen zwei Positionen i und j einmalig und kann nirgendwo anders als Teilwort des Wortes gefunden werden. Also bestimmt jedes Paar (i, j) mit $i \leq j \leq n$ ein Teilwort des Wortes und die Anzahl der Paare gleicht damit der Anzahl der nichtleeren Teilwörter. Die Anzahl der Paare (i, j) mit $1 \leq i < j \leq n$ ist

$$\left(\begin{array}{c} n \\ 2 \end{array} \right)$$

und die Anzahl der Paare (i,i) für $i = 1 \ldots n$ ist n. Somit erhalten wir, dass die Anzahl der nichtleeren Teilwörter die Zahl

$$\begin{aligned} \left(\begin{array}{c} n \\ 2 \end{array} \right) + n &= \frac{n \cdot (n-1)}{2} + n \\ &= n \cdot (\frac{n-1}{2} + 1) \\ &\{\text{Nach dem Distributivgesetz}\} \\ &= n \cdot \frac{n+1}{2} = \left(\begin{array}{c} n+1 \\ 2 \end{array} \right) \end{aligned}$$

ist. Jetzt kommt noch 1 dazu für λ. Also ist die Gesamtzahl der Teilwörter der Länge n mit n unterschiedlichen Buchstaben gleich

$$\left(\begin{array}{c} n+1 \\ 2 \end{array} \right) + 1.$$

Eine andere Art, die Teilwörter zu zählen, ist die folgende. Es gibt n Teilwörter der Länge 1 (n einzelne Buchstaben), $n-1$ Teilwörter der Länge 2, $n-2$ Teilwörter der Länge 3, usw. bis zum einen Teilwort der Länge n. Also ist die Anzahl der nichtleeren Teilwörter gleich

$$\begin{aligned} 1+2+3+\ldots+n &= \sum_{i=1}^{n} i \\ &= \frac{n \cdot (n+1)}{2} \\ &\{\text{Nach dem kleinen Gauss}\} \\ &= \left(\begin{array}{c} n+1 \\ 2 \end{array} \right) \end{aligned}$$

Wenn wir jetzt 1 für λ dazu addieren, erhalten wir das gleiche Resultat wie vorher. Die wenigsten Teilwörter haben Wörter a^n über einem einelementigen Alphabet. Es gibt genau 1 Wort für jede Länge von 0 bis zu n und somit genau $n+1$ Wörter.

Aufgabe 1.16

a) Die Sprache L ist endlich, weil sie keine Wörter länger als 5 enthält. Das Wort $0000 = 0^4$ ist nicht in der Sprache. Das leere Wort λ ist auch nicht in der Sprache.

b) Die Sprache L ist unendlich, weil sie z.B. alle Wörter der Form $b^i abba$ für $i \in \mathbb{N}$ enthält. Die Wörter $\lambda, a, b, aa, ab, ba$ und bb sind nicht in der Sprache enthalten, weil sie das Teilwort $abba$ nicht beinhalten. Das Wort $a^4 abba$ ist auch nicht in der Sprache, weil es zu viele Symbole a enthält.

Aufgabe 1.21
Diese Relation ist reflexiv, weil man eine Stadt nicht verlassen muss, um sie zu erreichen. Wenn man aus a nach b fliegen kann, dann kann man auch aus b nach a fahren. Damit ist die Relation symmetrisch. Wenn b aus a und c aus b erreichbar ist, dann ist auch c aus a erreichbar. Somit ist die Relation auch transitiv. Am Ende sehen wir, dass die Relation in dem Sinne vollständig ist, dass alle Städte in einem Netz gegenseitig erreichbar sind. Wenn man aber irgendwelche Inseln ohne Brücke einbezieht, können mehrere unabhängige Strassennetze entstehen und die Relation wird nicht mehr vollständig sein. Sie bleibt aber reflexiv, symmetrisch und transitiv.

Aufgabe 1.22
Weil ein Wort ein Präfix von sich selbst ist, ist die Relation *Prä* reflexiv. Die Relation ist nicht symmetrisch, weil gilt: a ist ein Präfix von aaa, aber aaa ist kein Präfix von a. Die Relation *Prä* ist transitiv, denn falls u ein Präfix von w ist $(w = ux)$ und w ein Präfix von v ist $(v = wy)$, dann ist auch u ein Präfix von $v(v = wy = uxy)$.

Aufgabe 1.23
Für $A = \{X, Y, Z\}$ und $B = \{\triangle, \square, \circ\}$ gilt

$$A \times B = \{(X,\triangle),(X,\square),(X,\circ),(Y,\triangle),(Y,\square),(Y,\circ),(Z,\triangle),(Z,\square),(Z,\circ)\}.$$

Jedes Element aus A wurde mit jedem Element aus B zu einem Paar zusammengesetzt. Die Anzahl der Elemente in $A \times B$ ist somit $|A| \cdot |B| = 3 \cdot 3 = 9$. Eine Relation von A nach B ist eine beliebige Teilmenge der Menge $A \times B$. Somit ist die Anzahl der Relationen aus A nach B gleich

$$2^{|A \times B|} = 2^9.$$

Lektion 2

Das Modell der endlichen Automaten

Hinweis für die Lehrperson Wenn die Schülerinnen und Schüler den Befehl `goto` noch nicht kennen, muss man ihn zuerst vorstellen. Der Befehl `goto` *i* entspricht genau dem Befehl `JUMP` *i*, den wir im Modul „Geschichte und Begriffsbildung“ vorgestellt haben.

Wenn man ein Berechnungsmodell definieren will, muss man folgende Fragen beantworten (siehe auch das Modul „Geschichte und Begriffsbildung“):

1. Welche elementaren Operationen, aus denen man die Programme zusammenstellen kann, stehen zur Verfügung?

2. Wie viel Speicher steht zur Verfügung und wie geht man mit dem Speicher um?

3. Wie wird die Eingabe eingegeben?

4. Wie wird die Ausgabe bestimmt (ausgegeben)?

Bei endlichen Automaten hat man keinen Speicher zur Verfügung. Man hat lediglich den, in dem das Programm gespeichert wird und den Zeiger, der auf die aktuell ausgeführte Zeile des Programms zeigt. Somit darf das Programm keine Variablen benutzen. Das mag überraschend sein, weil man sich fragt, wie man ohne Variablen überhaupt rechnen kann. Der Grundgedanke dabei ist, dass der Inhalt des Zeigers, also die Nummer der aktuellen Programmzeile, die einzige, wechselnde Information ist. Mit dieser Pseudovariable muss man daher auskommen.

J. Hromkovič, *Lehrbuch Informatik*, DOI 10.1007/978-3-8348-9692-6_23,

Wenn $\Sigma = \{a_1, a_2, \ldots, a_k\}$ das Alphabet ist, über dem die Eingaben dargestellt sind, dann darf der endliche Automat nur den folgenden Operationstyp benutzen:

$$\begin{aligned} \texttt{select}\ \mathit{input} &= a_1\ \texttt{goto}\ i_1 \\ \mathit{input} &= a_2\ \texttt{goto}\ i_2 \\ &\vdots \\ \mathit{input} &= a_k\ \texttt{goto}\ i_k \end{aligned}$$

Die Bedeutung dieser Operation (dieses Befehls) ist, das nächste Eingabesymbol zu lesen und mit $a_1, a_2, \ldots, a_k$ zu vergleichen. Wenn es gleich a_j ist, setzt das Programm die Arbeit in der Zeile i_j fort. Die Anweisung

$$\texttt{goto}\ l$$

bedeutet wörtlich „Gehe in die Zeile l und setze dort die Arbeit fort!“. Die Umsetzung dieses Befehls bedeutet automatisch die Löschung des gelesenen Symbols und das Weiterlesen in der Zeile i_j beim nächsten Symbol. Jede Zeile des Programms enthält genau einen Befehl in der oben angegebenen Form. Wir nummerieren die Zeilen mit natürlichen Zahlen $0, 1, 2, 3, \ldots$ und die Arbeit des Programms beginnt immer in der Zeile 0.

Aufgabe 2.1 Betrachten wir das Programm:

$$\begin{aligned} 0{:}\quad & \texttt{select}\ \mathit{input} = a\ \texttt{goto}\ 1 \\ & \quad\ \mathit{input} = b\ \texttt{goto}\ 1 \\ & \quad\ \mathit{input} = c\ \texttt{goto}\ 0 \\ 1{:}\quad & \texttt{select}\ \mathit{input} = a\ \texttt{goto}\ 0 \\ & \quad\ \mathit{input} = b\ \texttt{goto}\ 0 \\ & \quad\ \mathit{input} = c\ \texttt{goto}\ 1 \end{aligned}$$

In welche Zeile geht man, wenn man in der 0-ten Zeile das Symbol c gelesen hat? Wohin geht man, wenn man in der Zeile 1 das Symbol b liest?

Wenn Σ nur aus zwei Symbolen (z. B. 1 und 0) besteht, kann man statt des Befehls `select` den folgenden Befehl `if ... then ... else` verwenden:

$$\texttt{if}\ \mathit{input} = 1\ \texttt{then goto}\ i\ \texttt{else goto}\ j.$$

Solche Programme benutzt man, um Entscheidungsprobleme zu lösen. Die Antwort ist durch die Zeilennummer bestimmt. Wenn ein Programm aus m Zeilen besteht, wählt man eine Teilmenge F von $\{0,1,2,\ldots,m-1\}$. Wenn nach dem Lesen der gesamten Eingabe das Programm in der j-ten Zeile endet und $j \in F$, dann akzeptiert das Programm die Eingabe. Wenn $j \in \{0,1,2,\ldots,m-1\} - F$, dann akzeptiert das Programm die Eingabe nicht. Die Menge aller akzeptierten Wörter ist die von dem Programm akzeptierte (erkannte) Sprache.

Betrachten wir als Beispiel folgendes Programm A, das Eingaben über dem Alphabet Σ_{bool} bearbeitet.

0: `if` *input* $= 1$ `then goto` 1 `else goto` 2
1: `if` *input* $= 1$ `then goto` 0 `else goto` 3
2: `if` *input* $= 0$ `then goto` 0 `else goto` 3
3: `if` *input* $= 0$ `then goto` 1 `else goto` 2

Wählen wir $F = \{0,3\}$. Das Programm A arbeitet auf eine Eingabe 1011 wie folgt: Es startet in der Zeile 0 und geht in die Zeile 1, nachdem es eine 1 gelesen hat. Es liest eine 0 in der ersten Zeile und geht in die dritte Zeile. In der dritten Zeile liest es eine 1 und geht in die zweite Zeile, um in die dritte Zeile nach dem Lesen einer weiteren 1 zurückzukehren. Die Berechnung ist beendet und weil $3 \in F$ gilt, wird das Wort 1011 akzeptiert.

Aufgabe 2.2 Betrachten wir das Programm aus Aufgabe 2.1. Sei $F = \{0\}$. Welche der Wörter $a, c, ccc, acbca, aabbcc, accbac$ werden akzeptiert? Kannst du eine unendliche Menge von Wörtern bestimmen, die dieser Automat mit Sicherheit nicht akzeptieren kann. Findest du auch eine unendliche Menge von Wörter, die dieses Programm akzeptiert?

Mit endlichen Automaten verbindet man oft die schematische Darstellung aus Abb. 2.1. In dieser Abbildung sehen wir die drei Hauptkomponenten des Modells: ein gespeichertes **Programm**, ein **Band** mit dem Eingabewort und einen **Lesekopf**, der sich auf dem Band nur von links nach rechts bewegen kann.[1] Das Band (auch Eingabeband genannt) betrachtet man als einen linearen Speicher für die Eingabe. Das Band besteht aus Feldern (Zellen). Ein Feld ist eine elementare Speichereinheit, die ein Symbol aus dem betrachteten Alphabet beinhalten kann. Ein Arbeitsschritt des endlichen Automaten (des

[1] Die komponentenartige Darstellung von allgemeinen Berechnungsmodellen beinhaltet außerdem noch einen Speicher, Schreib- und Lesezugriffsmöglichkeiten auf diesen Speicher und eventuell ein Ausgabemedium.

Programms) besteht aus der Ausführung des Befehls der Zeile, in der sich das Programm befindet. Der Automat liest das Symbol des Feldes, auf dem sich sein Lesekopf befindet. Abhängig vom Symbol geht man in die nächste Programmzeile, die bearbeitet werden soll. Der Lesekopf rückt dabei automatisch um ein Feld nach rechts.

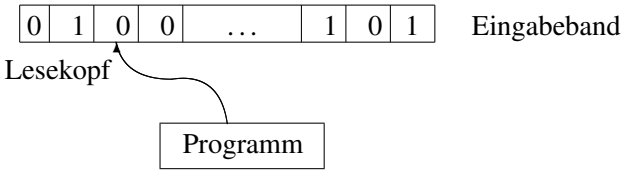


Abbildung 2.1

Die oben beschriebene Klasse von Programmen benutzt man heute fast gar nicht mehr, um endliche Automaten zu definieren, weil diese Programme wegen des `goto`-Befehls keine schöne Struktur haben. Daher ist diese Modellierungsart nicht sehr anschaulich und für die meisten Zwecke auch sehr unpraktisch. Die Idee einer umgangsfreundlicheren Definition endlicher Automaten basiert auf folgender visueller Darstellung unseres Programms. Wir ordnen jedem Programm A einen gerichteten, markierten Graphen $G(A)$ zu. $G(A)$ hat genauso viele Knoten wie das Programm A Zeilen hat. Jeder Zeile von A ist genau ein Knoten zugeordnet, der durch die Nummer der Zeile markiert ist. Die Knoten werden als kleine Kreise dargestellt. Der Name des Knotens wird in den Kreis geschrieben. Falls das Programm A aus einer Zeile i in die Zeile j beim Lesen eines Symbols b übergeht, dann enthält $G(A)$ eine gerichtete Kante (i, j) mit der Markierung b (Abb. 2.2).

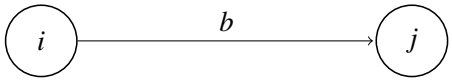


Abbildung 2.2

Weil unsere Programme ohne Variablen für jedes $a \in \Sigma$ in jeder Zeile einen `select`-Befehl haben[2], hat jeder Knoten von $G(A)$ genau den Ausgangsgrad[3] $|\Sigma|$.

Abb. 2.3 enthält den Graph $G(A)$ für das oben beschriebene vierzeilige Programm A. Die Zeilen aus F sind durch Doppelkreise als abgesonderte Knoten von $G(A)$ gekennzeichnet. Der Knoten, welcher der Zeile 0 entspricht, wird durch einen zusätzlichen Pfeil (Abb. 2.3) als Anfangsknoten aller Berechnungen bezeichnet.

[2] Jede Zeile ist ein `select` über alle Symbole des Alphabets.

[3] Der Ausgangsgrad eines Knoten ist die Anzahl der gerichteten Kanten, die den Knoten verlassen.

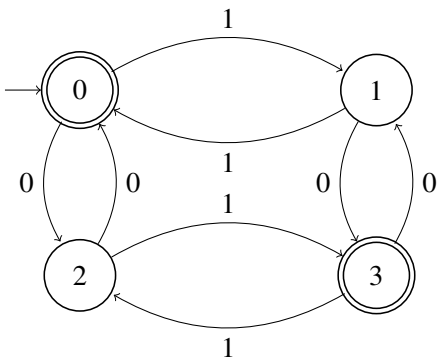


Abbildung 2.3

Aufgabe 2.3 Stelle den endlichen Automaten über $\{a,b,c\}$, der durch folgendes Programm angegeben wird, graphisch dar.

```
0:    select input = a goto 1
            input = b goto 1
            input = c goto 0
1:    select input = a goto 2
            input = b goto 2
            input = c goto 1
2:    select input = a goto 2
            input = b goto 2
            input = c goto 2
```

Sei $F = \{2\}$. Welche der Wörter $acc, abba, ccbac, cbcbc, caccc, cacbc$ werden akzeptiert? Kannst du die Sprache bestimmen, die dieser endliche Automat akzeptiert?

Aufgabe 2.4 Schreibe das Programm, das dem Automaten über $\{0,1\}$ in Abb. 2.4 entspricht. Bestimme die vier kürzesten Wörter, die er akzeptiert.

Beispiel 2.1 Wir betrachten die folgende Aufgabe, die kein Entscheidungsproblem darstellt. Wir wollen einen einfachen Obstautomaten entwerfen, der abgepackte Äpfel, Birnen und Orangen einzeln verkauft. Jedes Stück kostet 1 Euro; man darf nur mit Münzen der Größen 50 Cent und 1 Euro bezahlen. Wir fordern, dass man den genauen Wert von 1 Euro einwerfen und danach einen der drei Knöpfe für Apfel, Birne oder Orange drücken

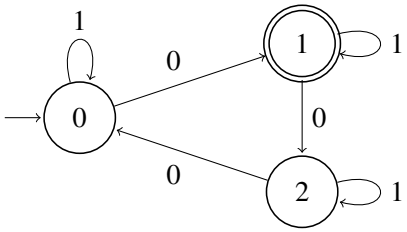


Abbildung 2.4

muss. Wenn dies passiert, akzeptiert der Obstautomat die Folge von Signalen und gibt das gewünschte Obststück aus.

Die Signale von außen sind für den Obstautomaten Münzeinwürfe und das Drücken des Auswahlknopfes. Die Münzen stellen wir durch die Symbole $M_{0.50}$ und $M_{1.00}$ dar. Für die Auswahlknöpfe führen wir die Bezeichnungen A, B und O ein. Somit ist $\Sigma = \{M_{0.50}, M_{1.00}, A, B, O\}$ das benötigte Alphabet. Ein Automat, der diese Aufgabe erfüllt, ist in Abb. 2.5 dargestellt.

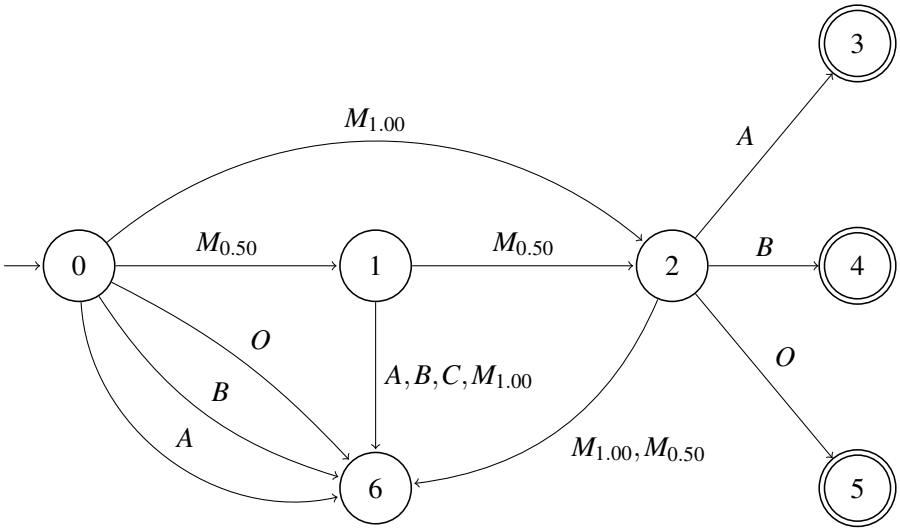


Abbildung 2.5

Wenn dieser Automat den Knoten 2 erreicht hat, wurde genau der Betrag von 1 Euro bezahlt. Durch die Wahl des entsprechenden Knopfes A, B oder C gelangt man dann

zu einem der Zustände 3, 4 oder 5, in denen das gewünschte Objekt ausgegeben wird. Wenn man in den Knoten 0 oder 1 eine der Wahltasten A, B oder O betätigt, geht man in den Knoten 6 über. Weil die Summe nicht stimmt, wird hierbei das eingezahlte Geld zurückgegeben und der Vorgang abgebrochen. Der Verlauf ist ähnlich, wenn man im Knoten 1 und 2 versucht, mehr Geld als benötigt einzuwerfen.

Wie man in Abb. 2.5 sieht, wird das Zeichnen von zu vielen Kanten vermieden, indem man zulässt, die Situation aus Abb. 2.6 durch Abb. 2.7 darzustellen. Man darf also alle gerichteten Kanten aus einem Knoten p in einen anderen Knoten q durch eine Kante darstellen. Dazu schreibt man die entsprechenden Symbole all dieser Kanten auf eine einzige Kante.

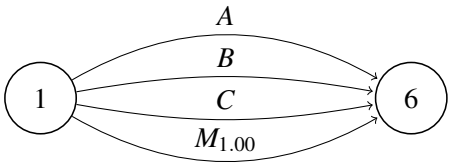


Abbildung 2.6

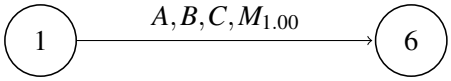


Abbildung 2.7

So wie der Automat entworfen wurde, funktioniert er nur für einen einzigen Versuch. Wir können ihn verbessern, indem ein zusätzliches Symbol S eingeführt wird. Dieses soll automatisch in den Zuständen 3, 4, 5, und 6 auftreten, sobald Obst oder eingeworfenes Geld ausgegeben wurde. Visuell kann man dann Pfeile mit der Bezeichnung S aus den Knoten 3, 4, 5, und 6 in den Knoten 0 einfügen. Dadurch wird der Automat im Startzustand zu einem neuen Verkaufsversuch bereit. □

Aufgabe 2.5 Entwerfe einen Verkaufsautomaten, der drei unterschiedliche Produkte verkauft. Mindestens zwei Produkte sollen unterschiedliche Preise haben. Der Verkaufsautomat muss so gesteuert werden, dass man zuerst die gewünschte Ware wählen muss und erst danach korrekt einbezahlt wird.

Hinweis für die Lehrperson Die folgende mathematische Beschreibung eines endlichen Automaten kann man umgehen oder geschickt erleichtern. Im Prinzip reicht es aus, die Notation der Übergangsfunktion $\delta(q,a) = p$ einzuführen. Für das minimale Programm mit dem Automatenentwurf wird nichts mehr gebraucht.

Aus dieser graphischen Darstellung entwickeln wir jetzt die standardisierte formale Definition von endlichen Automaten. Die graphische Darstellung werden wir aber weiterhin verwenden, weil sie eine sehr anschauliche Beschreibung von endlichen Automaten bietet. Die folgende formale Definition ist wiederum besser für das Studium der Eigenschaften endlicher Automaten und für die formale Beweisführung geeignet. Hierfür ändern wir teilweise die Terminologie. Was bisher als Zeile des Programms oder als Knoten des Graphen bezeichnet wurde, wird im Weiteren als **Zustand** des endlichen Automaten bezeichnet. Die Kanten des Graphen, die den `goto`-Befehlen des Programms entsprechen, werden durch die sogenannte **Übergangsfunktion** beschrieben.

Man beachte, dass der folgenden Definition ein allgemeines Schema zugrunde liegt, das man bei der Definition aller Rechnermodelle anwenden kann. Zuerst definiert man eine Struktur, welche die exakte Beschreibung jedes Objekts der Modellklasse ermöglicht. Dann beschreibt man die Bedeutung (Semantik) dieser Struktur. Dies geschieht in folgender Reihenfolge: Zuerst definiert man den Begriff der Konfiguration. Eine Konfiguration ist die vollständige Beschreibung einer Situation (eines allgemeinen Zustands), in der sich das Modell befindet. Dann definiert man einen Schritt als einen Übergang aus einer Konfiguration in eine andere Konfiguration, wobei dieser Übergang durch eine elementare Aktion (Durchführung eines Befehls) des Rechnermodells realisierbar sein muss. Eine Berechnung kann dann als eine Folge solcher Schritte angesehen werden. Wenn man eine Berechnung definiert hat, kann man jeder Eingabe das Arbeitsresultat des Rechnermodells als Ausgabe zuordnen.

Definition 2.1 *Ein (deterministischer)* **endlicher Automat** (EA) *ist ein Quintupel* $M = (Q, \Sigma, \delta, q_0, F)$*, wobei*

(i) Q *eine endliche Menge von* **Zuständen** *ist*
$\{$*vorher: Die Menge der Programmzeilen ohne Variablen oder die Menge der Knoten in graphischer Darstellung.*$\}$*,*

(ii) Σ *ein Alphabet, genannt* **Eingabealphabet***, ist*
$\{$*Alle zulässigen Eingaben müssen Wörter über* Σ *sein.*$\}$*,*

(iii) $q_0 \in Q$ *der* **Anfangszustand** *ist*
{*vorher: Die Zeile* 0 *des Programms ohne Variablen, aus der die Berechnung immer startet.*},

(iv) $F \subseteq Q$ *die* **Menge der akzeptierenden Zustände**[4] *ist und*

(v) δ *eine Funktion aus* $Q \times \Sigma$ *nach* Q *ist, die* **Übergangsfunktion** *genannt wird*
{$\delta(q,a) = p$ *bedeutet, dass* M *in den Zustand* p *übergeht, falls* M *im Zustand* q *das Symbol* a *gelesen hat (Abb. 2.8).*}

Benutzen wir noch einmal das Programm A, um die gegebene Definition der endlichen Automaten zu illustrieren. Der zum Programm A in Abb. 2.3 äquivalente EA ist $M = (Q, \Sigma, \delta, q_0, F)$ mit

$Q = \{q_0, q_1, q_2, q_3\}$, $\Sigma = \{0,1\}$, $F = \{q_0, q_3\}$ und der
Zustand q_i entspricht der Zeile i von A für $i = 0,1,2,3$
$\delta(q_0,0) = q_2$, $\delta(q_0,1) = q_1$, $\delta(q_1,0) = q_3$, $\delta(q_1,1) = q_0$,

$\delta(q_2,0) = q_0$, $\delta(q_2,1) = q_3$, $\delta(q_3,0) = q_1$, $\delta(q_3,1) = q_2$.

Anschaulicher kann man die Übergangsfunktion δ durch die folgende Tabelle beschreiben:

Zustand	Eingabe	
	0	1
q_0	q_2	q_1
q_1	q_3	q_0
q_2	q_0	q_3
q_3	q_1	q_2

Tabelle 2.1

Die prägnantere Darstellung eines EA ist aber die schon angesprochene graphische Form (Abb. 2.3), in die der EA, wie in in Abb. 2.12 gezeigt, umgewandelt werden kann.

[4] In der deutschsprachigen Literatur auch Endzustände genannt. Der Begriff „Endzustand“ kann aber auch zu Missverständnissen führen, weil die Berechnungen in einem beliebigen Zustand enden können. Außerdem entspricht der Begriff „akzeptierender Zustand“ der wahren Bedeutung dieses Begriffs und der Bezeichnung bei anderen Berechnungsmodellen, wie etwa bei Turingmaschinen.

Die graphische Abbildung von $\delta(p,a) = q$ entspricht der Kante in Abb. 2.8.

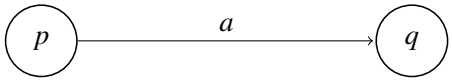


Abbildung 2.8

Aufgabe 2.6 Gib die formale Beschreibung als Quintupel für die endlichen Automaten aus Aufgabe 2.3 und 2.4 an.

Aufgabe 2.7 Zeichne die graphische Darstellung des endlichen Automaten $M = (Q,\Sigma,\delta,q_0,F)$ für

$Q = \{q_0,q_1,q_2,q_3\}$, $\Sigma = \{a,b\}$, $F = \{q_0\}$ und
$\delta(q_0,a) = q_1$, $\delta(q_0,b) = q_2$,
$\delta(q_1,a) = q_2$, $\delta(q_1,b) = q_3$,
$\delta(q_2,a) = q_3$, $\delta(q_2,b) = q_0$,
$\delta(q_3,a) = q_0$, $\delta(q_3,b) = q_1$.

Unter dem Begriff der Konfiguration< eines Rechnermodells verstehen wir eine vollständige Beschreibung der allgemeinen Situation (allgemeiner Zustand), in dem sich das Modell befindet. Diese Beschreibung muss mindestens alle Informationen beinhalten, die noch Einfluss auf eine Berechnung aus dieser Konfiguration haben dürfen. Bei endlichen Automaten gehört der (innere) Zustand des Automaten sowie der noch nicht gelesene Rest der Eingabe (der nicht gelesene Puffer des Eingabebandes) dazu.

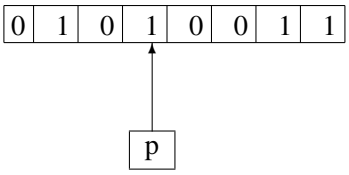


Abbildung 2.9 Ein Automat in der Konfiguration $(p, 10011)$

Aufgabe 2.8 Nehmen wir an, ein endlicher Automat M hat angefangen, auf dem Wort *abbabba* zu arbeiten, und hat die Konfiguration $(q, abba)$ erreicht. Stelle die entsprechende Situation ähnlich wie in Abb. 2.9 dar.

Definition 2.2 *Eine* **Konfiguration** *von M ist ein Element aus* $Q \times \Sigma^*$.

{Wenn M sich in einer Konfiguration $(q,w) \in Q \times \Sigma^*$ *befindet, bedeutet es, dass M im Zustand q ist und noch das Suffix w eines Eingabewortes lesen soll (siehe Abb. 2.9).}*

Eine Konfiguration $(q_0,x) \in \{q_0\} \times \Sigma^*$ *heißt die* **Startkonfiguration von** *M* **auf** *x.*

{Die Arbeit (Berechnung) von M auf x muss in der Startkonfiguration (q_0,x) *von x anfangen.}*
Jede Konfiguration aus $Q \times \{\lambda\}$ *nennt man eine* **Endkonfiguration***, weil alle Symbole der Eingabe schon gelesen worden sind und somit die Arbeit des Automaten beendet ist.*

Ein **Schritt** *von M ist eine Relation (auf Konfigurationen)*

$$\vdash_M \subseteq (Q \times \Sigma^*) \times (Q \times \Sigma^*),$$

definiert durch

$$(q,w) \vdash_M (p,x) \Leftrightarrow w = ax,\ a \in \Sigma \text{ und } \delta(q,a) = p.$$

Ein Schritt entspricht einer Anwendung der Übergangsfunktion auf die aktuelle Konfiguration, in der man sich in einem Zustand q befindet und ein Eingabesymbol a liest. Der Schritt $(q_0,ax) \vdash_M (p,x)$ ist in Abb. 2.10 veranschaulicht.

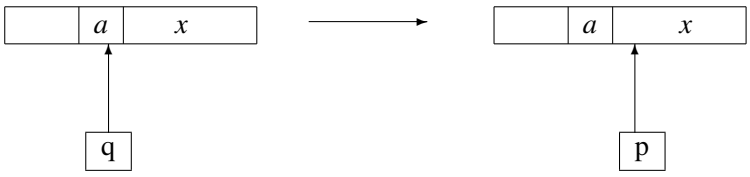


Abbildung 2.10

Aufgabe 2.9 Schreibe mittels des Symbols $\vdash$ den Schritt auf, der in Abb. 2.11 eingezeichnet ist.

Den Schritt in Abb. 2.11 verdanken wir dem Übergang, der durch $\delta(r,0) = s$ gegeben ist.

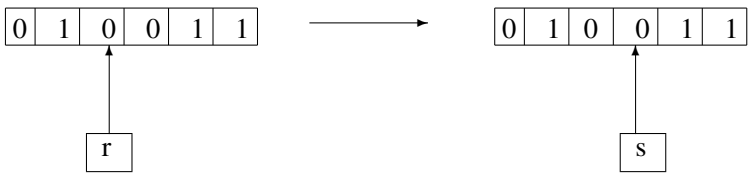


Abbildung 2.11

Aufgabe 2.10 Zeichne zwischen zwei Konfigurationen, wie in Abb. 2.10 dargestellt, die Übergänge ein, die folgenden Schritten eines EA(M) entsprechen:

(a) $(q_0, abbab) \vdash_M (q_1, bbab)$

(b) $(q_0, aaaa) \vdash_M (q_7, aaa)$

(c) $(p, 0110) \vdash_M (r, 110)$

Bei allen diesen Schritten ist das schon gelesene Präfix der Eingabe ohne Bedeutung. Du kannst es einfach mit x in deinem Bild bezeichnen.

Jetzt definieren wir die Berechnung als Folge von Berechnungsschritten.

Eine **Berechnung** *C von M ist eine endliche Folge $C = C_0, C_1, \ldots, C_n$ von Konfigurationen, so dass $C_i \vdash_M C_{i+1}$ für alle $0 \leq i \leq n-1$ gilt. Um Berechnungen wiederzugeben, verwenden wir vorzugsweise die Darstellung*

$$C_0 \vdash_M C_1 \vdash_M C_2 \vdash_M C_3 \ldots \vdash_M C_n.$$

C ist die **Berechnung von M auf eine Eingabe** *$x \in \Sigma^*$, falls $C_0 = (q_0, x)$ und $C_n \in Q \times \{\lambda\}$ eine Endkonfiguration ist.*

Somit kann man eine Berechnung auf x auch als eine Folge von $|x|$ vielen Schritten aus der Konfiguration (q_0, x) über $|x| - 1$ Konfigurationen in die entsprechende Endkonfiguration betrachten.

Falls $C_n \in F \times \{\lambda\}$ gilt, sagen wir, dass C eine **akzeptierende Berechnung** *von M auf x ist und M* **das Wort** *x* **akzeptiert**. *Falls $C_n \in (Q - F) \times \{\lambda\}$ gilt, sagen wir, dass*

C eine **verwerfende Berechnung** *von M auf x ist und M* **das Wort** *x* **verwirft (nicht akzeptiert)**.

Merke: M hat für jede Eingabe $x \in \Sigma^*$ genau eine Berechnung.

Die Berechnung von M aus Abb. 2.12 auf die Eingabe 1011 ist

$$(q_0, 1011) \vdash_M (q_1, 011) \vdash_M (q_3, 11) \vdash_M (q_2, 1) \vdash_M (q_3, \lambda).$$

Weil $q_3 \in F$ ist, gilt $1011 \in L(M)$.

Aufgabe 2.11 Schreibe die Berechnungen des endlichen Automaten aus Abb. 2.12 für die folgenden Wörter:

(i) 0000

(ii) 010101

(iii) 0011011

Welche der Wörter werden akzeptiert und welche verworfen?

Wenn $C_1 \vdash_M C_2 \vdash_M C_3 \vdash_M \ldots \vdash_M C_m$ eine Berechnung des endlichen Automaten M ist, dann sagen wir, dass die Konfiguration C_m in einer Berechnung aus C_1 **erreichbar** ist und schreiben

$$C_1 \vdash_M^* C_m.$$

Die Bezeichnung $C \vdash_M^* D$ bedeutet, dass eine Folge von Berechnungsschritten existiert, in der man aus der Konfiguration C in die Konfiguration D übergeht. Die Anzahl der Schritte in dieser Folge kann immer auch 0 sein und deswegen gilt $H \vdash_M^* H$ für jede Konfiguration H. Mathematisch (algebraisch) ausgedrückt ist $\vdash_M^*$ eine Relation auf Konfigurationen, welche die transitive und reflexive Hülle der Relation $\vdash_M$ ist.

Definition 2.3 *Die* **von** *M* **akzeptierte Sprache** $L(M)$ *ist definiert als*

$$L(M) := \{w \in \Sigma^* \mid \textit{die Berechnung von } M \textit{ auf } w \textit{ endet in einer Endkonfiguration } (q, \lambda) \textit{ mit } q \in F\}.$$

$\mathscr{L}(\mathrm{EA}) = \{L(M) \mid M \textit{ ist ein } \mathrm{EA}\}$ *ist die Klasse der Sprachen, die von endlichen Automaten akzeptiert werden.* $\mathscr{L}(\mathrm{EA})$ *bezeichnet man auch als die* **Klasse der regulären Sprachen***, und jede Sprache L aus* $\mathscr{L}(\mathrm{EA})$ *wird* **regulär** *genannt.*

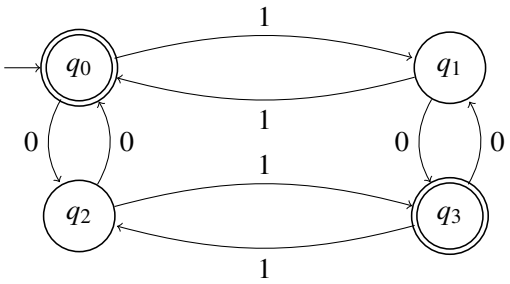


Abbildung 2.12

Beispiel 2.2 Betrachten wir den EA M in der Abb. 2.12. Die Frage ist, ob die Berechnung $(q_1,1101) \vdash_M^* (q_3,1)$ gilt. Um es festzustellen, starten wir die Berechnung aus der Konfiguration $(q_1,1101)$ und schauen, ob wir dabei die Konfiguration $(q_3,1)$ erreichen.

$$(q_1,1101) \vdash_M (q_0,101) \vdash_M (q_1,01) \vdash_M (q_3,1).$$

Wir haben die Konfiguration $(q_3,1)$ erreicht. Somit ist die Antwort positiv.

Untersuchen wir jetzt die Frage, ob $(q_0,0011101) \vdash_M^* (q_0,01)$ gilt?

$(q_0,0011101) \vdash_M (q_2,011101) \vdash_M (q_0,11101) \vdash_M (q_1,1101) \vdash_M$
$(q_0,101) \vdash_M (q_1,01) \vdash_M (q_3,1) \vdash_M (q_2,\lambda)$.

Diese Berechnung aus der Konfiguration $(q_0,0011101)$ enthält alle Konfigurationen, die aus $(q_0,0011101)$ erreichbar sind. Die Konfiguration $(q_0,01)$ ist nicht dabei, deswegen ist unsere Antwort negativ. Wir beobachten aber, dass es nicht notwendig war, die ganze Berechnung durchzuführen, um festzustellen, dass $(q_0,01)$ aus $(q_0,0011101)$ nicht erreichbar ist. Das Wort 0011101 ist um fünf Buchstaben länger als das Wort 01, deswegen kann $(q_0,01)$ aus $(q_0,0011101)$ entweder in fünf Berechnungsschritten erreicht werden oder gar nicht. □

Aufgabe 2.12 Könntest du, ohne die Berechnung aufzuschreiben, feststellen, ob $C_1 \overset{*}{\underset{M}{\vdash}} C_2$ für M aus Abb. 2.12 für folgende C_1 und C_2 gilt?

a) $C_1 = (q_0, 110110), C_2 = (q_3, 1101)$

b) $C_1 = (q_2, 0110), C_2 = (q_3, 0011)$

Begründe deine Antwort!

Aufgabe 2.13 Für welche Paare der folgenden Konfigurationen (C_1, C_2) ist C_2 aus C_1 durch eine Berechnung der EA M (Abb. 2.12) erreichbar?

a) $C_1 = (q_2, 0101001), C_2 = (q_2, 1)$

b) $C_1 = (q_1, 110110), C_2 = (q_3, 10)$

c) $C_1 = (q_0, 010100100), C_2 = (q_3, 100)$

Zusammenfassung

Endliche Automaten sind Programme mit nummerierten Zeilen, die keine Variablen, sondern nur die Verzweigungsbefehle `select` und `if` ... `then` ... `else` verwenden. Die Zeile, in der das Programm seine Arbeit beendet, bestimmt das Resultat. Endliche Automaten lösen Entscheidungsprobleme. Die Zeilen sind auf zwei Gruppen verteilt. Ein Gruppe bedeutet Akzeptanz, die andere Nichtakzeptanz.

Bei der Visualisierung von endlichen Automaten ziehen wir eine graphische Darstellung vor. Die Zeilen bezeichnen wir als Knoten eines gerichteten Graphen. Einzelne `goto`-Anweisungen sind durch Kanten (Pfeile) dargestellt.

Ein Berechnungsschritt eines endlichen Automaten entspricht der Ausführung des Befehls einer Zeile. Wir verstehen darunter einen Übergang aus einer Konfiguration in eine andere. Mit dem Begriff der Konfiguration bezeichnen wir eine vollständige Beschreibung der Situation (der Gesamtzustände) eines Rechnermodells. Eine Berechnung kann man dann als eine Folge von Berechnungsschritten ansehen oder als eine Folge von Konfigurationen darstellen, wobei man aus jeder Konfiguration der Folge die nachfolgende Konfiguration in einem Schritt erreichen kann.

Zur mathematischen Darstellung eines endlichen Automaten verwenden wir den Begriff Zustand anstelle von Programmzeile.

Kontrollfragen

1. Was sind die Grundkomponenten eines Rechners? Welche fehlen bei endlichen Automaten?

2. Wie kann ein endlicher Automat rechnen, wenn er keine Variablen benutzen darf?

3. Welche sind die einzigen Befehle eines endlichen Automaten?

4. Wie löst ein endlicher Automat ein Entscheidungsproblem?

5. Welche drei Modelle von endlichen Automaten wurden vorgestellt? Wie wechseln die Namen der Grundbegriffe bei dem Übergang von Modell zu Modell?

6. Was ist eine Konfiguration eines endlichen Automaten?

7. Was ist ein Berechnungsschritt eines endlichen Automaten? Was ist eine Berechnung eines endlichen Automaten?

8. Wann betrachten wir eine Berechnung als eine Berechnung auf einem Wort? Wie viele Berechnungen gibt es auf einem Wort?

9. Wann akzeptiert ein endlicher Automat ein Wort? Wie stellen wir fest, dass ein *EA* ein gegebenes Wort verwirft (nicht akzeptiert)?

Kontrollaufgaben

1. Schreibe einen endlichen Automaten als ein zweizeiliges Programm, das

   a) über dem Alphabet $\{a,b,c,d\}$ arbeitet,

   b) mit der nullten Zeile akzeptiert und

   c) in einer beliebigen Zeile nach dem Lesen jedes Buchstabens die Zeile wechselt.

2. Stelle den *EA* aus der Kontrollaufgabe 1 in den anderen Formen dar.

3. Bestimme die drei kürzesten Wörter über $\{0,1\}$, die der *EA* in Abb. 2.3 akzeptiert. Bestimme die vier kürzesten Wörter über $\{0,1\}$, die der Automat nicht akzeptiert.

4. Betrachte den endlichen Automaten in Abb. 2.4. Schreibe die Berechnungen des Automaten auf den Wörtern $0^6, 0^9, 0^5, 0101010, 0011001$.

5. Betrachte den *EA* aus Abb. 2.12. Finde zwei Konfigurationen C und D so, dass D aus C nicht erreichbar ist. Ist die Konfiguration $(q_0, 111)$ aus der Konfiguration $(q_1, 00010111)$ erreichbar?

6. Entwerfe graphisch einen endlichen Automaten zum Verkauf von Briefmarken vom Wert $50c$ und $80c$. Zum Kauf darf man nur Münzen $10c, 20c$ und $50c$ verwenden. Der Verkaufsautomat gibt kein Geld zurück, also muss die Einkaufssumme genau einbezahlt werden.

Lösungen zu ausgesuchten Aufgaben

Aufgabe 2.1
Wenn man in der 0-ten Zeile des Programms das Symbol c liest, wird der Befehl `goto` 0 ausgeführt. Somit bleibt man in der 0-ten Zeile.
Wenn man in der ersten Zeile das Symbol b liest, setzt die Ausführung des Programms wegen „$input = b$ `goto` 0“ die Arbeit in der 0-ten Zeile fort.

Aufgabe 2.4 Das Programm zu dem endlichen Automaten in Abb. 2.4 sieht wie folgt aus:

0: `if` $input = 0$ `then goto` 1 `else goto` 0
1: `if` $input = 0$ `then goto` 2 `else goto` 1
2: `if` $input = 0$ `then goto` 0 `else goto` 2

Die Zeile 1 ist die einzige akzeptierende Zeile. Das leere Wort λ wird damit nicht akzeptiert, weil man mindestens ein Symbol braucht um aus der Zeile 0 die Zeile 1 zu erreichen. Damit ist das kürzeste akzeptierte Wort das Wort 0. Es wird kein anderes Wort der Länge 1 akzeptiert. Von der Länge 2 werden die Wörter 10 und 00 akzeptiert. Von der Länge 3 sind es die Wörter 011, 101 und 110.

Aufgabe 2.9
Die graphische Darstellung des Übergangs zwischen zwei Konfigurationen in Abb. 2.11 entspricht dem Berechnungsschritt:

$$(r, 0011) \vdash_M (s, 011).$$

Aufgabe 2.12 a)
Die Konfiguration $C_2 = (q_3, 1101)$ kann nicht aus der Konfiguration $C_1 = (q_0, 110110)$ erreicht werden, weil das Wort 1101 aus C_2 kein Suffix des Wortes 110110 aus C_1 ist.

Lektion 3

Entwurf von endlichen Automaten

Eine der grundlegenden Aufgaben in der Automatentheorie ist der Entwurf eines Automaten zu einem vorgegebenen Zweck. In diesem Abschnitt machen wir den ersten Schritt, um dieses zu erlernen. Wir starten mit dem Entwurf eines endlichen Automaten für eine endliche Sprache.

Beispiel 3.1 Betrachten wir die Sprache

$$L = \{0110\}$$

über $\{0,1\}$, die nur das einzige Wort 0110 enthält. Um 0110 zu akzeptieren, müssen die Symbole 0,1,1 und 0 in dieser Reihenfolge gelesen werden. Dies ist durch die Transitionen in Abb. 3.1 umsetzbar.

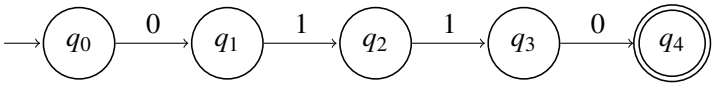


Abbildung 3.1

Dabei ist q_0 der Anfangszustand und q_4 ist der einzige akzeptierende Zustand. Offensichtlich akzeptiert jeder endliche Automat, der die Transitionen aus Abb. 3.1 enthält, das Wort 0110. Wir müssen auch noch gewährleisten, dass kein anderes Wort akzeptiert wird. Die Vervollständigung des „Teilautomaten“ in Abb. 3.1 erfordert, dass aus jedem Zustand genau zwei Transitionen (gerichtete Kanten) mit der Beschriftung 0 und 1 herausführen müssen. Das gilt es ebenfalls zu beachten. Zu diesem Zweck führen wir den neuen Zustand q_5 ein, den wir als „Abfallkorb“ oder „Sackgasse“ bezeichnen. Dorthin

J. Hromkovič, *Lehrbuch Informatik*, DOI 10.1007/978-3-8348-9692-6_24,

geht der Automat immer dann, wenn ohne Rücksicht auf den noch nicht gelesenen Rest (Suffix) des Eingabewortes schon sicher ist, dass das Wort nicht in der Sprache liegt. Der Automat darf den Abfallkorb nicht mehr verlassen, was man dadurch gewährleistet, dass keine Transition aus dem Abfallkorb führt. Der resultierende Automat ist in Abb. 3.2 dargestellt.

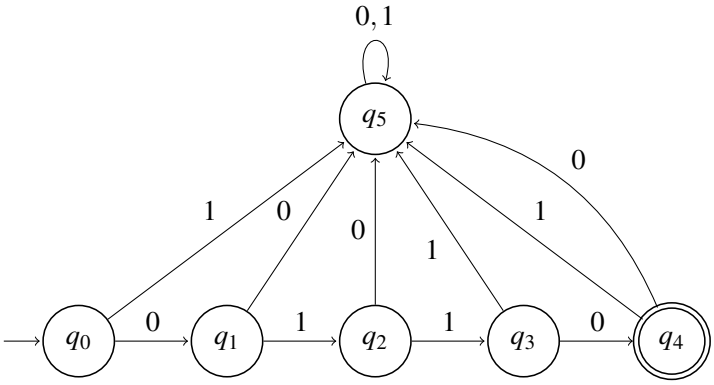


Abbildung 3.2

Wir sehen,

$$\delta(q_5,0) = \delta(q_5,1) = q_5$$

gewährleistet das Nicht-Verlassen des Abfallkorbs. Die Transitionen

$$\delta(q_4,0) = \delta(q_4,1) = q_5$$

verursachen, dass jedes Wort, welches mit dem Präfix 0110 anfängt und dann weiter fortgesetzt wird, nicht akzeptiert wird. □

Aufgabe 3.1 Zeichne einen endlichen Automaten, der die Sprache $\{abbaa\}$ über dem Alphabet $\{a,b\}$ akzeptiert.

Beispiel 3.2 Wir betrachten jetzt die Sprache $L = \{00111, 111, 000\}$, die drei Wörter enthält. Um einen EA A mit $L(A) = L$ zu entwerfen, wählen wir eine ähnliche Strategie wie im Beispiel 3.1. Wir testen die korrekten Symbolfolgen. Die Wörter 000 und 00111 haben das gemeinsame Präfix 00. Deshalb überprüfen wir die ersten zwei Symbole für diese beiden Wörter gemeinsam und erst dann die restlichen Suffixe 0 und 111. Diese Strategie ist in Abb. 3.3 durch einen Teilautomaten dargestellt. □

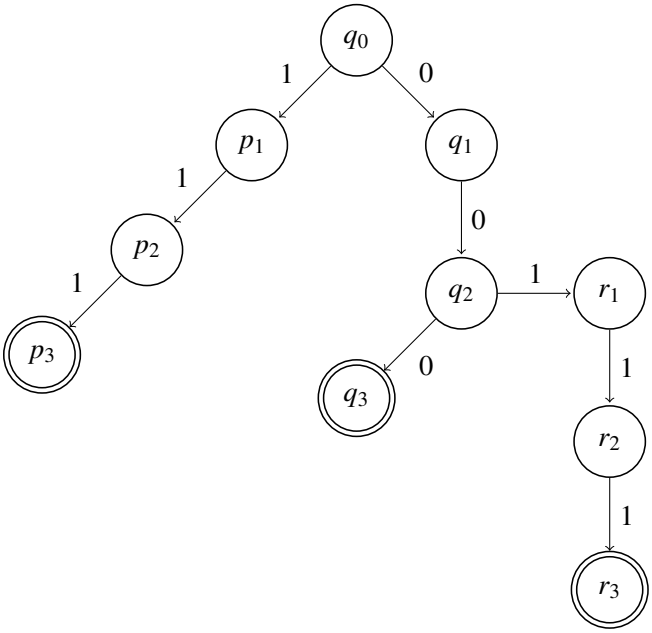


Abbildung 3.3

Aufgabe 3.2 Vervollständige den Teilautomaten aus Abb. 3.3 zu einem Automaten A mit $L(A) = \{00111, 111, 000\}$. Wie würdest du diesen EA modifizieren, um einen EA für die folgenden Sprachen zu entwerfen?

a) $L = \{00111, 101\}$

b) $L = \{111, 0000, 0011\}$

c) $L = \{00111, 000, 0111\}$

Aufgabe 3.3 Entwirf endliche Automaten für die folgenden endlichen Sprachen:

a) $L = \{11111\}$ über $\{0, 1\}$,

b) $L = \{0101, 11, 0100\}$ über $\{0, 1\}$,

c) $L = \{0, 00, 001, 0010\}$ über $\{0, 1, 2\}$,

d) $L = \{0, 1, 2, 00, 10, 211\}$ über $\{0, 1, 2\}$.

Beispiel 3.3 Betrachten wir die Sprachen $L = \{0,1\}^*$ und $L_\emptyset = \emptyset$. Offensichtlich akzeptiert der endliche Automat in Abb. 3.4 (a) L und der EA in Abb. 3.4 (b) akzeptiert $L_\emptyset$.

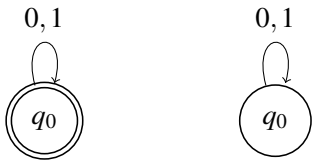


Abbildung 3.4

Beide Automaten bestehen nur aus einem einzigen Startzustand q_0, wobei im Fall von L dieser Zustand auch ein akzeptierender Zustand ist. □

Aufgabe 3.4 Zeichne einen endlichen Automaten, der die Sprache $\{0,1\}^* - \{\lambda\}$ akzeptiert.

Aufgabe 3.5 Vervollständige den Teilautomaten in Abb. 3.1 zu einem endlichen Automaten, der die Sprache $L = \{0110x \mid x \in \{0,1\}^*\}$ über $\{0,1\}$ akzeptiert.

Aufgabe 3.6 Entwirf endliche Automaten für die folgenden Sprachen:

a) $\{01011x \mid x \in \{0,1,2\}^*\}$ über $\{0,1,2\}$,

b) $\{1x \mid x \in \{0,1\}^*\} \cup \{01y \mid y \in \{0\}^*\}$,

c) $\{001,000\} \cup \{1\}^*$,

d) $\{0,1\}^*$ - $\{\lambda,0,1\}$ (alle Wörter über $\{0,1\}$ außer dem leeren Wort λ und den Wörtern 0 und 1),

e) $\{\lambda,00,11\}$.

Für das echte und vollständige Verständnis der Funktionalität eines endlichen Automaten (und somit für den Automatenentwurf) ist die folgende Überlegung von grundsätzlicher Bedeutung: Wenn ein endlicher Automat A mit einer Zustandsmenge Q und einem

Anfangszustand q_0 über einem Alphabet Σ arbeitet, dann setzt A eine Funktion aus Σ^* nach Q um, die jedem Wort x aus Σ^* den Zustand zuordnet, in dem A nach dem Lesen von x endet. Mit anderen Worten, wenn

$$(q_0, x) \vdash_A^* (p, \lambda)$$

gilt, hat A dem Wort x den Zustand p zugeordnet.

Auf diese Weise zerfällt Σ^* in die folgenden $|Q|$ Klassen

$$\text{Klasse}[p] = \{x \in \Sigma^* \mid (q_0, x) \vdash_M^* (p, \lambda)\}$$

für jedes $p \in Q$. Die Klasse $[p]$ beinhaltet alle Wörter, deren Bearbeitung aus der entsprechenden Startkonfiguration im Zustand p endet. Somit sieht die Situation wie in Abb. 3.5 aus.

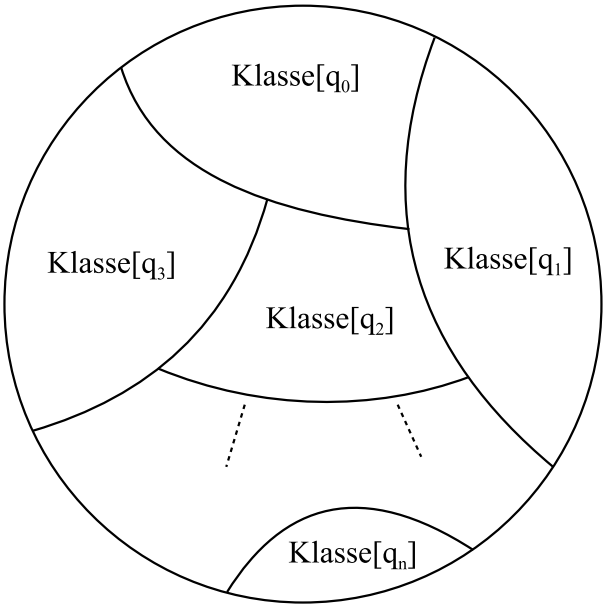


Abbildung 3.5 Σ^*

Offensichtlich ist

$$\text{Klasse}[p] \cap \text{Klasse}[q] = \emptyset$$

für jedes Paar mit unterschiedlichen Zuständen p und q, weil sich kein Wort in zwei Zuständen befinden darf.

Andererseits gilt

$$\Sigma^* = \bigcup_{p \in Q} \text{Klasse}[p]$$

(die Klassen zusammen ergeben Σ^*), weil nach dem Lesen jedes Wort irgendeinem Zustand zugeordnet werden muss. Man sagt dazu:

Σ^* **zerfällt** in die Klassen Klasse$[q_0]$, …, Klasse$[q_n]$

oder

diese Klassen bilden eine **Zerlegung (Partitionierung)** von Σ^*.

Wir verstehen einen EA $A = (\Sigma, Q, q_0, \delta, F)$ vollständig, wenn wir für jeden Zustand $q \in Q$ die Klasse$[q]$ bestimmen können und somit dem Zustand q seine Bedeutung (Rolle) zuordnen. In dieser Terminologie ist die von A akzeptierte Sprache $L(A)$ nichts anderes, als die Vereinigung der Klassen der akzeptierenden Zustände. Daher gilt:

$$L(A) = \bigcup_{p \in F} \text{Klasse}[p].$$

Beispiel 3.4 Betrachten wir den Automaten A in Abb. 3.2. Wir bestimmen die Klassen für alle seine fünf Zustände.

$$\begin{aligned}
\text{Klasse}[q_0] &= \{\lambda\} \\
\text{Klasse}[q_1] &= \{0\} \\
\text{Klasse}[q_2] &= \{01\} \\
\text{Klasse}[q_3] &= \{011\} \\
\text{Klasse}[q_4] &= \{0110\} \\
\text{Klasse}[q_5] &= \{0,1\}^* - \{\lambda, 0, 01, 011, 0110\} \\
&= \{x \in \{0,1\}^* \mid x \text{ ist kein Präfix von } 0110\}.
\end{aligned}$$

□

Später werden wir für diese Zuordnung der Klassen zu den Zuständen auch die Korrektheit beweisen. Weil q_4 der einzige akzeptierende Zustand ist, gilt

$$L(A) = \text{ Klasse}[q_4] = \{0110\}.$$

Aufgabe 3.7 Begründe mit eigenen Worten, warum die Klassen der fünf Zustände des EA in Beispiel 3.4 korrekt bestimmt sind.

Aufgabe 3.8 Wie würdest du den EA in Abb. 3.2 modifizieren, damit er die folgende Sprache $L = \{x \in \{0,1\}^* \mid x \text{ ist ein Präfix von } 0110\}$ akzeptiert?

Aufgabe 3.9 Entwirf einen EA, der die Sprache $L = \{x \in \{0,1,2\}^* \mid x \text{ ist ein Präfix von } 1101102\}$ akzeptiert. Bestimme dabei die Klassen, die zu den Zuständen gehören.

Aufgabe 3.10 Bestimme die Wortklassen der Zustände von den endlichen Automaten, die du bei der Bearbeitung von Aufgabe 3.3 entworfen hast.

Aufgabe 3.11 Bestimme die Wortklassen der Zustände von den endlichen Automaten, die du bei der Bearbeitung der Aufgabe 3.6 entworfen hast.

Aufgabe 3.12 Welche Bedeutung haben die Klassen für den Obstautomaten in Abb. 2.5?

Warum ist dieses Konzept der Zerlegung von Σ^* in Zustandsklassen für den Automatenentwurf nützlich?

Die zu erkennende Sprache L ist durch konkrete Eigenschaften ihrer Wörter gegeben. Ein EA A mit $L(A) = L$ soll diese Eigenschaften erkennen, indem er sich nach der Bearbeitung eines Präfixes etwas merkt. Der Inhalt des Merkens wird durch den Zustand bestimmt, in dem er sich gerade befindet. Jeder Zustand entspricht konkreten Worteigenschaften. Wenn man im Vorfeld richtig abschätzt, was man sich zum Erkennen der Worteigenschaften merken muss, dann kann man die Zustände und ihre Bedeutungen bestimmen. Erst danach bestimmt man fast automatisch die Transitionen (Kanten) zwischen den Zuständen.

Hinweis für die Lehrperson Die nachfolgenden Überlegungen empfehlen wir nicht für das Selbststudium. Eine intensive Mitwirkung der Lehrperson ist erwünscht. Es ist auch an der Lehrperson zu entscheiden, wie weit sie den exakten Weg des mathematischen Formalismus verfolgt. Statt des hier präsentierten allgemeinen Zugangs kann man die Idee auch nur anhand von Beispielen erklären.

Um es noch besser zu begreifen, stellen wir folgende Überlegung an: Seien x und y zwei unterschiedliche Wörter, die zur gleichen Klasse Klasse$[p]$ eines EA A gehören. Dies bedeutet

$$(q_0,x) \vdash_A^* (p,\lambda) \text{ und } (q_0,y) \vdash_A^* (p,\lambda).$$

In Abb. 3.6 ist diese Tatsache veranschaulicht.

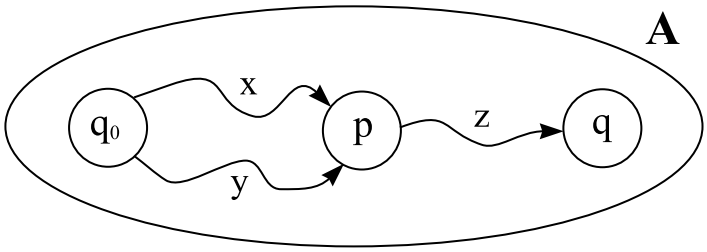


Abbildung 3.6

Nach dem Lesen von x und y gelangt der EA A in den gleichen Zustand p. In diesem Augenblick

kann A zwischen x und y nicht mehr unterscheiden.

Der Automat weiß nur, dass er sich im Zustand p befindet, jedoch nicht, durch welches Wort er p erreicht hat (weil er keinen Speicher hat). Welche Konsequenzen hat das? Wenn noch ein beliebiges Wort z (Suffix z) zu lesen ist, dann endet der endliche Automat A in einem Zustand q. Der Zustand q ist dabei jener, zu dem man gelangt, wenn man im Zustand p angefangen hat, das Wort z zu lesen ($(p,z) \vdash_A^* (q,\lambda)$). Folglich liegen xz und yz in der Klasse$[q]$. Somit gehören entweder xz und yz in $L(A)$ (wenn $q \in F$) oder beide nicht (wenn $q \notin F$). Formal lautet diese wichtige Beobachtung:

Satz 3.1 *Sei $A = (\Sigma, Q, \delta, q_0, F)$ ein endlicher Automat. Seien x und y zwei beliebige Wörter aus Σ^*. Wenn*

$$(q_0,x) \vdash_A^* (p,\lambda) \textit{ und } (q_0,y) \vdash_A^* (p,\lambda)$$

für einen Zustand $p \in Q$ ist, dann gilt für **alle** *Wörter $z \in \Sigma^*$:*

xz und yz gehören zur selben Klasse$[q]$ *für ein $q \in Q$*

und folglich

$$xz \in L \Leftrightarrow\ yz \in L.$$

□

Was lernen wir aus dieser Behauptung? Wir haben eine Strategie (Methode) entwickelt, mit der wir für eine gegebene Sprache die Merkmale der Wörter bestimmen können, die in jedem EA, der L akzeptiert, durch Zustände unterschieden werden müssen.

Sehen wir uns die Sprache $L = \{0110\}$ und den entsprechenden Automaten in Abb. 3.2 an. Wenn wir es intuitiv betrachten, muss sich jeder EA, der L akzeptiert, das bisher gelesene Präfix von 0110 merken.

Lass uns diese Intuition formal bestätigen! Wenn ein EA nach dem Lesen von

$$x = 0 \text{ und } y = 01$$

in beiden Fällen in den gleichen Zustand gelangen würde (d.h. er könnte nicht mehr unterscheiden, ob er bisher 0 oder 01 gelesen hat), dann würden wir

$$z = 10$$

auswählen. Es gilt:

$$xz = 010 \notin L \text{ und } yz = 0110 \in L$$

Also muss $xz = 010$ verworfen und $yz = 0110$ akzeptiert werden. Wenn aber 0 und 01 in dieselbe Zustandsklasse gehören würden, dann wäre es laut Satz 3.1 nicht möglich. Deswegen muss jeder EA für L zwischen 0 und 01 unterscheiden können, indem er jeweils nach dem Lesen von 0 und von 01 in unterschiedliche Zustände gelangt.

Aufgabe 3.13 Zeige, dass jeder EA für $L = \{0110\}$ zwischen den Wörtern λ, 0, 01, 011 und 0110 unterscheiden können muss. Finde dazu für jedes Paar (x,y) dieser fünf Wörter ein z, so dass $xz \in L$ und $yz \notin L$ (oder $xz \notin L$ und $yz \in L$). Beispielsweise kann man für das Paar (0,0110) z als 110 oder als λ wählen.

Aufgabe 3.14 Entwirf einen EA für $L = \{aaab\}$ über $\{a,b\}$. Erkläre die Bedeutung der Zustände des entworfenen Automaten und begründe, warum du sie brauchst.

In den folgenden Beispielen und Aufgaben fokussieren wir uns darauf, den Satz 3.1 für einen „systematischen“ Entwurf von endlichen Automaten anzuwenden.

Beispiel 3.5 Seien $a \in \mathbb{N}$ und $b \in \mathbb{N} - \{0\}$. Mit

$$a \bmod b$$

bezeichnen wir den Rest nach der Division von a durch b. Zum Beispiel: $15 \bmod 4 = 3$, weil $15 = 3 \cdot 4 + 3$ gilt und $16 \bmod 4 = 0$, weil $16 = 4 \cdot 4 + 0$ gilt. Betrachten wir die folgende Sprache

$$\mathrm{L} = \{w \in \{0,1\}^* \mid |w|_0 \bmod 3 = 2\},$$

wobei $|w|_0$ die Anzahl der Symbole 0 in dem Wort w bezeichnet. Mit anderen Worten enthält L alle Wörter, deren Anzahl von Nullen gleich $3k+2$ für eine Zahl $k \in \mathbb{N}$ ist. Beispiele solcher Wörter sind 00, 10110, 00000, 101001001, usw.

Satz 3.1 folgend, müssen wir uns für jedes gelesene Wort w merken, wie groß der Rest der Division von $|w|_0$ durch 3 ist. Vermutlich brauchen wir drei Zustände q_0, q_1 und q_2 mit der Bedeutung:

- Klasse$[q_0] = \{w \in \{0,1\}^* \mid |w|_0 \bmod 3 = 0\}$
  = alle Wörter über $\{0,1\}$, deren Anzahl von Nullen durch 3 teilbar ist
- Klasse$[q_1] = \{w \in \{0,1\}^* \mid |w|_0 \bmod 3 = 1\}$
- Klasse$[q_2] = \{w \in \{0,1\}^* \mid |w|_0 \bmod 3 = 2\}$.

Wenn wir diese drei Zustände verwenden, ergibt sich der EA in Abb. 3.7 fast automatisch.

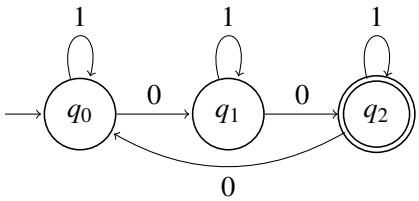


Abbildung 3.7

Weil λ keine 0 enthält, muss q_0 der Anfangszustand sein. Wenn man eine 1 liest, bleibt man im aktuellen Zustand, weil es keinen Einfluss auf die Akzeptanzbedingung hat. Also:

$$\delta(q_0,1) = q_0, \delta(q_1,1) = q_1 \text{ und } \delta(q_2,1) = q_2.$$

Beim Lesen einer 0 ändert sich die Anzahl der Nullen um 1 und entsprechend dieser Änderung müssen die Transitionen (Kanten mit Beschriftung 0) gelegt werden. Für die Zustände q_0 und q_1 bedeutet es, dass der Rest um 1 größer geworden ist und somit gilt

$$\delta(q_0,0) = q_1 \text{ und } \delta(q_1,0) = q_2.$$

Wenn der Rest 2 war, dann bedeutet eine zusätzliche 0, dass der Rest 0 wird. Daher bestimmen wir

$$\delta(q_2,0) = q_0.$$

Der Zustand q_2 ist der akzeptierende Zustand, (d.h. $F = \{q_2\}$), weil

$$L = \text{ Klasse}(q_2).$$

Wenn uns jetzt jemand die Aufgabe stellen würde, einen EA für die Sprache

$$L_{0,2} = \{w \in \{0,1\}^* \mid |w|_0 \bmod 3 = 0 \text{ oder } |w|_0 \bmod 3 = 2\}$$

zu erzeugen, würden wir einfach den EA aus Abb. 3.7 benutzen und dabei nur $F = \{q_0, q_2\}$ festsetzen. Weil

$$L_{0,2} = \text{ Klasse}(q_0) \cup \text{ Klasse}(q_2),$$

ist das die einzige, notwendige Veränderung. □

Aufgabe 3.15 Wende Satz 3.1 an, um zu zeigen, dass das Merken der Anzahl Nullen modulo 3 im Beispiel 3.4 unvermeidbar war. Hinweis: Du musst drei Wörter w_0, w_1 und w_2 finden, so dass gilt: $w_0 \in \text{Klasse}(q_0)$, $w_1 \in \text{Klasse}(q_1)$ und $w_2 \in \text{Klasse}(q_2)$. Für jedes Paar (w_i, w_j) mit $i \neq j$ muss man ein z finden, so dass eins der Wörter $w_i z$ und $w_j z$ in L ist und das andere nicht.

Aufgabe 3.16 Entwirf endliche Automaten für folgende Sprachen:

a) $\{w \in \{0,1\}^* \mid |w|_1 \bmod 5 \in \{1,3\}\}$,

b) $\{w \in \{0,1\}^* \mid |w|_1 \bmod 5 \in \{0,2,4\}\}$,

c) $\{w \in \{0,1\}^* \mid |w|_0 \bmod 6 \text{ ist gerade}\}$,

d) $\{w \in \{a,b\}^* \mid |w|_a = 2\} = \{b^i a b^j a b^k \mid i,j,k \in \mathbb{N}\}$,

e) $\{w \in \{a,b,1\}^* \mid w = 1x \text{ und } |x|_a \bmod 2 = 0\}$,

f) $\{w \in \{0,1\}^* \mid 2 \cdot |w|_0 \bmod 5 = 1\}$,

g) $\{w \in \{0,1\}^* \mid |w|_0 - 2 \cdot |w|_1 \bmod 5 = 0\}$,

Bestimme für die entworfenen Automaten alle Zustandsklassen.

Beispiel 3.6 Unsere nächste Aufgabe ist es, einen EA für die Sprache

$$L(0010) = \{x0010y \mid x,y \in \{0,1\}^*\}$$

zu entwerfen.

Hier ist der Entwurf etwas schwieriger als bei einer Sprache, bei der alle Wörter mit 0010 anfangen sollen. Wir müssen feststellen, ob 0010 an beliebiger Stelle im Eingabewort vorkommt. Zuerst muss sich der EA merken, welchen Teil des Präfixes von 0010 er schon in den zuletzt gelesenen Buchstaben gefunden hat. Ein Beispiel: Wenn das bisher gelesene Wort 011**001** war, dann muss er sich merken, dass er schon 001 als Kandidaten gefunden hat. Wenn jetzt das nächste Symbol 0 ist, dann muss er akzeptieren. Falls der EA das Präfix 11**00** einer Eingabe gelesen hat, muss er merken, dass die letzten zwei Symbole 00 passend waren[1]. Damit ergeben sich fünf mögliche Zustände entsprechend den fünf

[1] Im Prinzip reicht es aus, die Zahl 2 zu speichern, weil das Wort 0010 bekannt ist und 00 das Präfix der Länge 2 ist.

Präfixen von 0010. Ihre Bedeutung wird nachfolgend halbformal beschrieben.

Klasse$[p_0]$	kein Präfix von 0010 ist ein nicht leeres Suffix von dem bisher gelesenen Wort x (z.B. $x = \lambda$ oder x endet mit 11) und das Wort enthält das Teilwort 0010 nicht.
Klasse$[p_1]$	die Wörter enden mit 0 und enthalten kein längeres Präfix von 0010 als Suffix (z.B. sie enden mit 110) und das Wort enthält das Teilwort 0010 nicht.
Klasse$[p_2]$	die Wörter enden mit 00 und enthalten kein längeres Präfix von 0010 als ihr Suffix und das Wort enthält das Teilwort 0010 nicht.
Klasse$[p_3]$	die Wörter enden mit 001 und das Wort enthält das Teilwort 0010 nicht.
Klasse$[p_4]$	die Wörter enden mit 0010 oder beinhalten 0010 als Teilwort.

Diese Vorlage ergibt direkt die folgende Teilstruktur (Abb. 3.8) des zu konstruierenden EA.

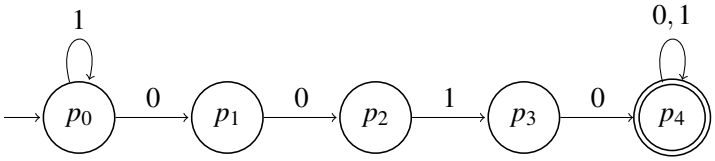


Abbildung 3.8

Dass die Folge 0010 gelesen werden muss, um den einzigen akzeptierenden Zustand p_4 zu erreichen, ist offensichtlich. Erreicht man p_4 , hat der EA schon 0010 in der Eingabe gefunden und somit verbleibt man im akzeptierenden Zustand $(\delta(q_4,0) = \delta(q_4,1) = q_4)$, unabhängig davon, was noch kommt. Das Lesen einer 1 in q_0 ändert nichts. Es gab noch kein Präfix von 0010 in den letzten Buchstaben, weil 0010 mit 0 anfängt.

Um den EA zu vervollständigen, fehlen uns drei Pfeile aus den Zuständen p_1, p_2 und p_3 zum Lesen von 1 aus p_1 und p_3 und von 0 aus p_2. Es gibt nur eine eindeutige Möglichkeit, dies korrekt umzusetzen (um $L(0010)$ zu erkennen). Sie ist in Abb. 3.9 dargestellt. Wird eine 1 in p_1 gelesen, beginnt die Suche nach 0010 neu, weil 0 das längste

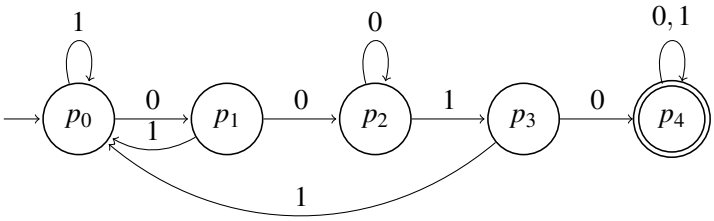


Abbildung 3.9

Suffix des gelesenen Wortes ist, das einem Präfix von 0010 entspricht. Somit bedeutet eine hinzugekommene 1 am Schluss, dass wir aktuell kein Präfix von 0010 vorliegen haben. Mit anderen Worten: Wenn 0 das Suffix von x mit $x \in Klasse[p_1]$ war, dann ist 01 das Suffix von $x1$. Somit hat $x1$ kein passendes Suffix. Damit ist

$$\delta(p_1, 1) = p_0.$$

Wenn man aber in p_2 eine 0 liest, bleibt es dabei, dass 00 das Suffix des eben gelesenen Wortes ist. (Wenn 00 ein Suffix von x ist, ist 000 ein Suffix von $x0$.) Somit bleiben wir in p_2, also

$$\delta(p_2, 0) = p_2.$$

Wenn man in p_3 zu einer 1 kommt, endet das gelesene Wort auf 11. (Wenn x mit 001 endet, dann endet $x1$ mit 0011.) Somit kann das Wort am Ende kein nichtleeres Suffix von 0010 enthalten. Die einzige, mögliche Schlussfolgerung ist

$$\delta(p_3, 1) = p_0.$$

Damit ist der EA vollständig. □

Aufgabe 3.17 Beschreibe die Klassen der Zustände des EA in Abb. 3.9 formal genau. Nimm für jede Klasse einen Repräsentanten (ein Wort aus der Klasse) und wende Satz 3.1 an. Zeige daran, dass man diese Klassen wirklich unterscheiden muss.

Aufgabe 3.18 Entwirf einen *EA*, der die folgende Sprache akzeptiert:

a) $\{0010x \mid x \in \{0,1\}^*\}$,

b) $\{xabbaaby \mid x, y \in \{a,b\}^*\}$,

c) $\{x00110 \mid x \in \{0,1\}^*\}$.

Im Beispiel 3.6 haben wir einen EA in Abb. 3.9 entworfen, der genau die Wörter akzeptiert, die das Wort 0010 als Teilwort enthalten. Der anstrengendste Teil des Entwurfs bestand in der Bestimmung der Kanten (Transitionen), wenn die Suche nach 0010 wegen einer Unstimmigkeit unterbrochen wurde. Dann musste man entscheiden, ob die Suche neu beginnen soll oder ob das zuletzt gelesene Suffix doch noch ein kürzeres Präfix von 0010 enthält. In diesem Fall musste die weitere Suche an der entsprechenden Stelle fortgesetzt werden. Dass man an dieser Stelle wirklich aufpassen muss, zeigt die Schwierigkeit der Klassenbeschreibung in Aufgabe 3.17. Man kann das Risiko in diesem Entwurfprozess vermeiden, indem man mehr Informationen über das gelesene Wort speichert. Eine Idee wäre, mit Hilfe der Zustandsnamen alle Suffixe der Länge 4 zu speichern. Ein Beispiel: Ein Zustand q_{0110} soll alle Wörter enthalten, die mit 0110 enden. Dadurch ist alles übersichtlich und die Beschreibung der Zustandsklassen einfach. Diese Transparenz geht jedoch auf Kosten der Automatengröße. Wir haben $2^4 = 16$ Zustände, um alle Suffixe der Länge 4 über $\{0,1\}$ zu speichern. Das ist jedoch nicht alles. Es gibt auch kürzere Wörter, die natürlich kein Präfix der Länge 4 enthalten. Alle Wörter kürzer als 4 benötigen daher eigene Zustände, woraus sich

$$2^3 + 2^2 + 2^1 + 1 = 15$$

weitere Zustände ergeben. Weil das Zeichnen eines EA mit 16 + 15 = 31 Zuständen einen hohen Aufwand produziert, stellen wir eine Anwendung dieser Idee für die Suche nach einem kürzeren Teilwort vor.

Beispiel 3.7 Betrachten wir die Sprache

$$L = \{x110y \mid x,y \in \{0,1\}^*\}.$$

Wie oben angedeutet, führen wir die Zustände p_{abc} ein, wobei in p_{abc} die Wörter ankommen, die mit dem Suffix abc für $a,b,c \in \{0,1\}$ enden. Dies ist noch nicht ganz genau, weil wir für $abc \neq 110$ ein Wort mit dem Suffix abc nur dann in die Klasse$[p_{abc}]$ aufnehmen können, wenn das Wort das Teilwort 110 nicht enthält (in diesem Fall müsste die Akzeptanz eines solchen Wortes längst entschieden worden sein). Wir beschreiben jetzt die Bedeutung aller Zustände:

$$
\begin{aligned}
\text{Klasse}[p_{110}] = L &= \{x \in \{0,1\}^* \mid x \text{ enthält das Teilwort } 110\},\\
\text{Klasse}[p_{000}] &= \{x000 \mid x \in \{0,1\}^* \text{ und } x000 \text{ enthält}\\
&\quad \text{das Teilwort 110 nicht}\},\\
\text{Klasse}[p_{001}] &= \{x001 \mid x \in \{0,1\}^* \text{ und } x001 \text{ enthält}\\
&\quad \text{das Teilwort 110 nicht}\},\\
\text{Klasse}[p_{010}] &= \{x010 \mid x \in \{0,1\}^* \text{ und } x010 \text{ enthält}\\
&\quad \text{das Teilwort 110 nicht}\},\\
\text{Klasse}[p_{011}] &= \{x011 \mid x \in \{0,1\}^* \text{ und } x011 \text{ enthält}\\
&\quad \text{das Teilwort 110 nicht}\},\\
\text{Klasse}[p_{100}] &= \{x100 \mid x \in \{0,1\}^* \text{ und } x100 \text{ enthält}\\
&\quad \text{das Teilwort 110 nicht}\},\\
\text{Klasse}[p_{101}] &= \{x101 \mid x \in \{0,1\}^* \text{ und } x101 \text{ enthält}\\
&\quad \text{das Teilwort 110 nicht}\},\\
\text{Klasse}[p_{111}] &= \{x111 \mid x \in \{0,1\}^* \text{ und } x111 \text{ enthält}\\
&\quad \text{das Teilwort 110 nicht}\}.
\end{aligned}
$$

Dann brauchen wir die Zustände

$$p_\lambda, p_0, p_1, p_{00}, p_{01}, p_{10} \text{ und } p_{11}$$

für kürzere Wörter, daher gilt:
$\text{Klasse}[p_\lambda] = \{\lambda\}$, $\text{Klasse}[p_0] = \{0\}$, $\text{Klasse}[p_1] = \{1\}$,
$\text{Klasse}[p_{00}] = \{00\}$, $\text{Klasse}[p_{01}] = \{01\}$,
$\text{Klasse}[p_{10}] = \{10\}$ und $\text{Klasse}[p_{11}] = \{11\}$.
Der Startzustand ist offensichtlich der Zustand p_λ, und p_{110} ist der einzige akzeptierende Zustand.

Der daraus resultierende EA ist in Abb. 3.10 dargestellt. Die Verzweigungsstruktur (Baumstruktur in der Informatik genannt) oben weist jedem Wort der Länge kürzer gleich 3 einen anderen Zustand zu. Alle anderen Kanten (Transitionen) führen zu den untersten Zuständen p_{abc}. Wenn zum Beispiel ein Wort $x001$ durch eine Null zu $x0010$ verlängert wird, dann muss man aus p_{001} in p_{010} übergehen (d.h. $\delta(p_{001}, 0) = p_{010}$).

Deswegen sind die Kanten so gelegt, dass durch das Lesen eines weiteren Symbols immer der Zustand erreicht wird, der dem neuen Suffix der Länge 3 entspricht. Die einzige Ausnahme ist der Zustand p_{110}. Von dort aus wird nicht in einen anderen Zustand

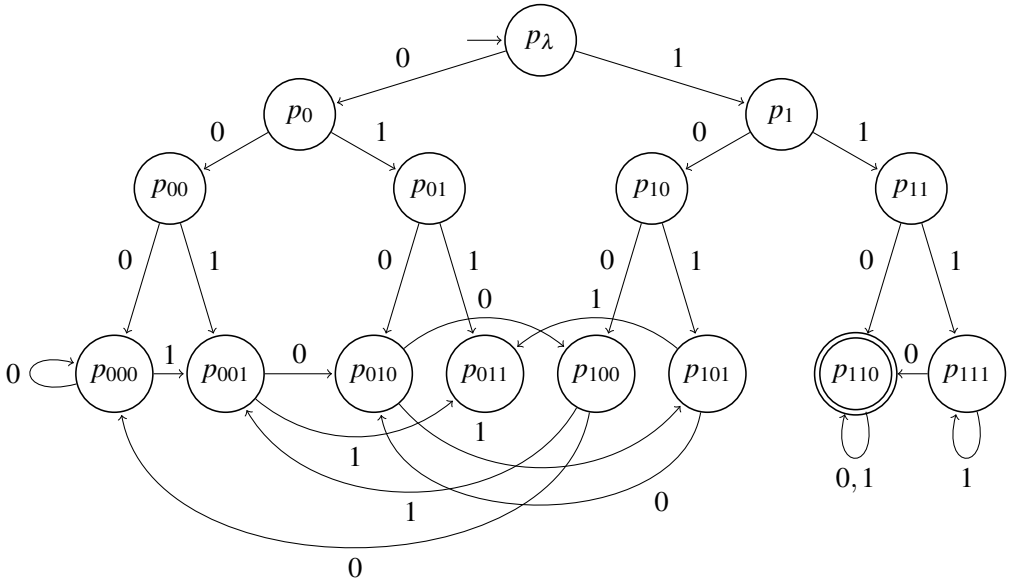


Abbildung 3.10

gewechselt, egal welche Symbole noch gelesen werden, denn das Teilwort 110 wurde bereits in der Eingabe erkannt und das gesamte Wort musste akzeptiert werden. □

Aufgabe 3.19 In Abb. 3.10 fehlen die Pfeile aus dem Zustand p_{011} für beide Symbole 0 und 1. Kannst du sie korrekt einzeichnen?

Aufgabe 3.20 Verändere den *EA* in Abb. 3.10 so, dass er die Sprache $\{x100y \mid x, y \in \{0,1\}^*\}$ akzeptiert.

Aufgabe 3.21 Modifiziere den *EA* in Abb. 3.10 so, dass er die Sprache $\{w100 \mid w \in \{0,1\}^*\}$ akzeptiert.

Aufgabe 3.22 Modifiziere den *EA* in Abb. 3.10 so, dass er die Sprache $\{00\} \cup \{x111 \mid x \in \{0,1\}^*\}$ akzeptiert.

Der Entwurf des EA aus Abb. 3.10 bereitet viel Arbeit, aber die könnte sich lohnen, wenn man auf eine übersichtliche Weise einen EA für eine Sprache wie

$$L = \{x \in \{0,1\}^* \mid x \text{ enthält mindestens eines der Wörter 000, 011 oder 110 als Teilwörter}\}.$$

entwerfen möchte. Dann wäre es ausreichend, den EA aus Abbildung 3.10 zu benutzen, die von p_{000} und p_{011} ausgehenden Kanten durch

$$\delta(p_{000},0) = \delta(p_{000},1) = p_{000} \text{ und } \delta(p_{011},0) = \delta(p_{011},1) = p_{011}$$

zu ersetzen und

$$F = \{p_{000}, p_{011}, p_{110}\}$$

zu wählen. □

Aufgabe 3.23 Modifiziere den Automaten aus Abbildung 3.10, damit er folgende Sprachen akzeptiert:

a) $\{xyz \in \{0,1\}^* \mid y \in \{001,010,100\}, x,z \in \{0,1\}^*\}$,
b)* $\{xyz \in \{0,1\}^* \mid y \in \{01,100,111\}, x,z \in \{0,1\}^*\}$,
c) $\{x011 \mid x \in \{0,1\}^*\}$,
d) $\{xz \in \{0,1\}^* \mid x \in \{0,1\}^*, z \in \{001,010,100\}\}$.

Zusammenfassung

Eine systematische Methode zum Entwurf von endlichen Automaten basiert auf der Idee zu überlegen, welche Informationen man über das bisher gelesene Präfix der Eingabe speichern muss, um die gegebene Sprache zu akzeptieren (die gesuchte Eigenschaft zu überprüfen). Jeder möglichen Kombination dieser Informationsinhalte wird ein Zustand zugeordnet. Die Übergänge zwischen zwei Zuständen sind dann durch die entsprechenden Änderungen der Worteigenschaften beim Lesen eines Buchstabens gegeben. Manchmal lohnt es sich beim Automatenentwurf, nicht minimalistisch zu sein, sondern mehr Information abzuspeichern als notwendig. Der resultierende endliche Automat wird zwar grösser, dafür aber viel übersichtlicher und für andere auch verständlicher.

Kontrollfragen

1. Jeder endliche Automat, der über einem Alphabet Σ arbeitet, verteilt Σ^* in disjunkte Klassen. Wie sind die Klassen bestimmt?

2. Welche Bedeutung hat es, wenn ein *EA* durch das Lesen von zwei unterschiedlichen Wörtern x und y den gleichen Zustand erreicht?

3. Wie kann man für eine gegebene Sprache zeigen, dass man zwei Wörter immer unterscheiden muss, damit kein endlicher Automat für L diese zwei Wörter in derselben Zustandsklasse hat?

4. Kann es sich für den Entwurf lohnen, einen nicht kleinstmöglichen Automaten zu entwerfen?

Kontrollaufgaben

1. Entwirf endliche Automaten für folgende Sprachen:

   a) $\{0.1\}^* - \{\lambda, 00, 11\}$,

   b) $\{00, 001, 101\}$,

   c) $\{0, 00, 101, 1011\}$.

2. Sei $L = \{x \in \{a,b,c\}^* \mid |x|_a \geq 3\}$. Erkläre, warum jeder *EA* M mit $L(M) = L$ zwischen den vier Wörtern $\lambda, bbab, aacc$ und $abcabcac$ unterscheiden muss!

3. Sei $L = \{w \in \{0,1\}^* \mid 2 \cdot |w|_0 - |w| \text{ und } 4 = 2\}$. Entwirf einen *EA* für L. Finde drei Wörter, von denen keine zwei in dieselbe Zustandsklasse eines endlichen Automaten für L gehören können und begründe warum!

4. Zeichne einen Obstautomaten, der im Unterschied zu unserem Obstautomaten auch 20-Cent-Münzen $M_{0.20}$ annimmt.

5. Entwirf EA für folgende Sprachen:

a) $\{x1111y \mid x,y \in \{0,1,2\}^*\}$,
b) $\{x110110y \mid x,y \in \{0,1,2\}^*\}$,
c) $\{x010101y \mid x,y \in \{0,1\}^*\}$,
d) $\{x0y \mid x,y \in \{0,1\}^*\}$,
e) $\{x0y1z \mid x,y,z \in \{0,1\}^*\}$,
f) $\{x \in \{0,1\}^* \mid |x|_0 \geq 3\}$,
g) $\{x \in \{0,1\}^* \mid 3 \leq |x|_1 \leq 5\}$,
h)* $\{x001y101z \mid x,y,z \in \{0,1\}^*\}$,
e) $\{x0011 \mid x \in \{0,1\}^*\}$,
f) $\{x10011 \mid x \in \{0,1\}^*\}$,
g) $\{1x11001 \mid x \in \{0,1\}^*\}$,
h) $\{x11001y0 \mid x,y \in \{0,1\}^*\}$.

6. Wende die Entwurfstrategie aus Abbildung 3.10 an, um einen EA für folgende Sprachen zu entwerfen:

   a) $\{x \in \{0,1\}^* \mid x$ enthält 11 als Teilwort oder endet mit dem Suffix $10\}$,

   b) $\{\lambda, 0, 11, x000 \mid x \in \{0,1\}^*\}$,

   c) $\{x \in \{0,1\}^* \mid x$ enthält 00 oder 11 als Teilwörter$\}$.

   Bestimme die entsprechenden Zustandsklassen für alle entworfenen Automaten.

7. Entwirf endliche Automaten für die folgenden Sprachen und bestimme dabei die Bedeutung aller Zustandsklassen:

   a) $\{xy \mid x \in \{0,1\}^*$ und $y \in \{00,10\}\}$,

   b) $\{xy \mid x \in \{0,1\}^*$ und $y \in \{101,010,111,011,110\}\}$,

   c) $\{xyz \mid x,z \in \{0,1\}^*$ und $y \in \{00,111,101\}\}$,

   d) $\{1xy \mid x \in \{0,1\}^*$ und $y \in \{11,010\}\}$.

Lösungen zu ausgesuchten Aufgaben

Aufgabe 3.4
Der endliche Automat für die Sprache $L = \{0,1\}^* - \{\lambda\}$ ist in Abb. 3.11 gezeichnet. Der Anfangszustand q_0 darf nicht ein akzeptierender Zustand sein, sonst würde λ akzeptiert. Bei jedem Symbol übergeht der Automat in den akzeptierenden Zustand q_1 und bleibt in diesem Zustand, egal welche Symbole noch gelesen werden. Damit werden alle nicht leeren Wörter akzeptiert.

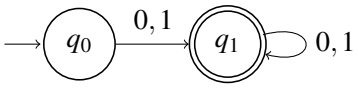


Abbildung 3.11

Aufgabe 3.8
Es reicht, alle Zustände ausser q_5 in akzeptierende Zustände umzuwandeln.

Aufgabe 3.13
Um zu zeigen, dass kein *EA* für $L = \{0110\}$ nach dem Lesen von Wörtern λ und 0 in den gleichen Zustand gelangen darf, wählen wir $z = 110$. Damit ist $\lambda z = 110 \notin L$ und $0z = 0110 \in L$. Somit ist klar, dass jeder Automat zwischen λ und 0 unterscheiden muss. Wenn beide 0 und λ zu einer gleichen Klasse $Klasse[q]$ gehören würden, müssten λz und $0z$ entweder beide akzeptiert oder beide verworfen werden. Für λ und 01 kann man $z = 10$ wählen und erhält nun $\lambda 01 = 01 \notin L$ und $01z = 0110 \in L$. Welches andere z kann man auch wählen, um zu zeigen, dass man zwischen λ und 01 unterscheiden muss? Für die Wörter 01 und 0110 kann man $z = \lambda$ wählen. Damit erhalten wir $01z = 01 \notin L$ und $0110z = 0110 \in L$. Die Wahl $z = 10$ ist auch günstig, weil $01z = 0110 \in L$ und $0110z = 011010 \notin L$. Die Wahl $z = 1$ für 01 und 0110 wird uns nicht helfen, weil in diesem Fall beide $01z$ und $0110z$ nicht in der Sprache liegen. Für das Wortpaar $(01,011)$ ist $z = 0$ oder $z = 10$ eine gute Wahl, um zu zeigen, dass 01 und 011 nicht zu der gleichen Zustandsklasse gehören dürfen. Schaffst du es selbst, für die restlichen Wortpaare das passende z zu finden?

Aufgabe 3.15
Wir wählen $\lambda \in Klasse[q_0], 0 \in Klasse[q_1]$ und $00 \in Klasse[q_2]$. Für $(\lambda,0)$ wählen wir $z = 0$. Somit gilt $\lambda z = 0 \notin L$ und $0z = 00 \in L$. Für $(\lambda,00)$ wählen wir $z = \lambda$. Somit erhalten wir $\lambda z = \lambda \notin L$ und $00z = 00 \in L$. Für das Paar $(0,00)$ wählen wir $\lambda = 0$ und erhalten $0z = 00 \in L$ und $00z = 000 \notin L$.

Aufgabe 3.16 (e)
Ein *EA* für die Sprache

$$\{w \in \{a,b,1\}^* \mid w = 1x \text{ und } |x|_a \bmod 2 = 0\}$$

verwirft alle Wörter, die mit a oder b (unterschiedlich von 1) anfangen. In diesem Fall geht der

EA in den Abfallkorb (Abb. 3.12). Wenn das erste Symbol 1 ist, wird überprüft, ob der restliche Teil des Eingabewortes eine gerade Anzahl von Symbolen a beinhaltet. Wo und wieviele Symbole 1 und b in diesem Suffix vorkommen, spielt keine Rolle.

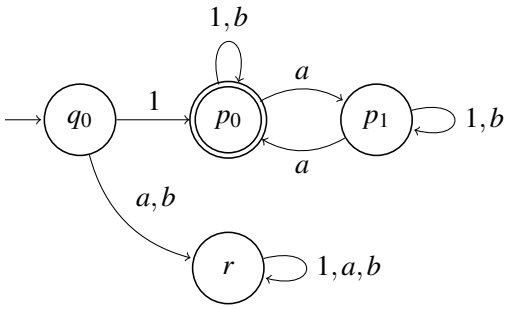


Abbildung 3.12

Die Bedeutung der Zustände q_0, p_0, p_1 und r kann wie folgt beschrieben werden.

$$\begin{array}{ll}
\text{Klasse}[q_0] = & \{\lambda\} \\
\text{Klasse}[r] = & \{ux \mid x \in \{a,b,1\}^* \text{ und } u \in \{a,b\}\} \\
& \{\text{alle Wörter, die nicht mit dem Symbol 1 anfangen}\} \\
\text{Klasse}[p_0] = & \{1x \mid x \in \{a,b,1\}^* \text{ und } |x|_a \bmod 2 = 0\} \\
\text{Klasse}[p_1] = & \{1y \mid y \in \{a,b,1\}^* \text{ und } |y|_a \bmod 2 = 1\}
\end{array}$$

Aufgabe 3.16 (g)
Jeder endliche Automat für diese Sprache muss für das bisher gelesene Wort w den Rest der Zahl $|w|_0 - 2 \cdot |w|_1$ nach der Teilung durch 5 abspeichern. Weil es 5 unterschiedliche Reste $0, 1, 2, 3$ und 4 gibt, reichen uns 5 Zustände q_0, q_1, q_2, q_3 und q_4 mit der Bedeutung

$$Klasse[q_i] = \{w \in \{0,1\}^* \mid |w|_0 - 2 \cdot |w|_1 \bmod 5 = i\}$$

für $i = 0, 1, 2, 3, 4$. Wenn man eine 0 liest, erhöht sich die Zahl $|w|_0 - 2 \cdot |w|_1$ um 1. Damit wächst der Rest für Reste $0, 1, 2$ und 3 um 1. Wenn der Rest 4 war, ist der neue Rest 0. Damit kann man in einer Schleife aus 0-Kanten alle Zustände verbinden (Abb. 3.13). Wenn man eine 1 liest, wird die Zahl $|w|_0 - 2 \cdot |w|_1$ um zwei vermindert. Das bedeutet, dass die 1-Kanten in der Schleife zwei Stellen zurück springen müssen. Bemerke, dass es das Gleiche ist, wie drei Schritte nach vorne zu springen.

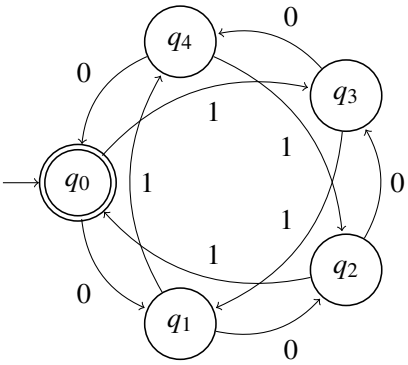


Abbildung 3.13

Lektion 4

Projekt „Steuerungsautomaten“

An dieser Stelle lohnt es sich, die formale Welt der Wörter und Sprachen teilweise zu verlassen, um sich kurz mit endlichen Automaten in unserer Umgebung zu beschäftigen. Endliche Automaten benutzt man nicht nur zum Bauen von Verkaufsautomaten, sondern auch zur Steuerung von Liften, Ampeln für Fußgängerübergänge und sogar für die Steuerung von komplexen Kreuzungen. Unsere Aufgabe im Rahmen eines Projektes ist es, den ganzen Weg von der informellen Aufgabenbeschreibung zur automatischen Steuerung bis zum Automatenentwurf zu gehen. Er wird in die folgenden zwei Phasen unterteilt.

Aufgabenstellung Die Aufgabenstellung definiert sowohl den Rahmen als auch die grundsätzliche Vorstellung in Bezug auf die Gerechtigkeit („fairness“) der Steuerungsstrategie gegenüber dem Nutzer. Die Eckpunkte beschreiben die definitiven Fixpunkte oder die Forderungen, welche unbedingt eingehalten werden müssen. Das kann zum Beispiel bei einer Kreuzung die Anzahl der Straßen mit ihren Eigenschaften, wie etwa die einer Einbahnstraße, sein. Eine vernünftige Forderung wäre, dass nicht alle Verkehrsteilnehmer an einer Kreuzung zur gleichen Zeit Grün haben. Die Anforderungen an die Gerechtigkeit können mehrere Freiheitsgrade haben. So soll zum Beispiel garantiert werden, dass man an einer Kreuzung nicht wesentlich länger als andere Verkehrsteilnehmer warten muss.

1. Phase Die Aufgabenstellung ist offen hinsichtlich der Lösungsmöglichkeiten. Der Wunsch nach „fairness“ lässt unterschiedliche Interpretationsmöglichkeiten zu. Manchmal sind Situationen unvermeidbar, in denen eine Partei der anderen vorgezogen werden

J. Hromkovič, *Lehrbuch Informatik*, DOI 10.1007/978-3-8348-9692-6_25,

muss. In der ersten Phase sollte man noch nicht intensiv über den Automatenentwurf nachdenken, sondern die Aufgabenstellung präzisieren und mehrere Lösungsstrategien vorschlagen. Typischerweise gibt es keine „optimale“ Strategie, jede hat gewisse Vor- und Nachteile. Hinzu kommt noch die Tatsache, dass man keinen so komplexen Automaten, wie etwa das Deutsche Steuergesetz mit 40 000 Seiten, entwerfen will. Zu komplexe Automaten sind unübersichtlich und somit schwer auf Korrektheit zu überprüfen. Ihre Herstellungskosten sind höher. Wenn sie versagen, ist die Fehlersuche viel aufwendiger und damit teurer. Die Wahrscheinlichkeit von Produktionsfehlern ist größer und damit steigen wiederum die Kosten für die Qualitätskontrolle. Die Mitglieder der Projektgruppe sind also auf der Suche nach transparenten, guten Lösungsstrategien. Am Ende der ersten Phase müssen sie sich für eine Strategie entscheiden und ihre Wahl plausibel begründen.

2. Phase Nachdem eine Lösungsstrategie festgelegt wurde, geht man zum Automatenentwurf über. Die Zustände werden verwendet, um alle denkbaren Situationen darzustellen. Das Alphabet wählt man so, dass man mit seinen Buchstaben alle möglichen vorhandenen Signale darstellen kann. Dabei wird nicht unbedingt von Anfang an eindeutig vermittelt, welche Information durch den Zustand gespeichert und welche durch die Signale übertragen werden. Ein Beispiel: Die Information, dass ein Fußgänger den Signalknopf der Ampel gedrückt hat, kann in den Steuerungszustand des ganzen Systems übernommen werden. Es könnte jedoch auch sein, dass diese Information auf der Signalebene bleibt und das Signal erst dann erlischt, wenn die Fußgängerampel auf Grün geschaltet wird.

Für die Projektarbeit in einer Klasse wird dieses Vorgehen empfohlen: Die Klasse wird in Arbeitsgruppen mit vier bis fünf Schülern aufgeteilt. Jede Gruppe erhält eine andere Aufgabenstellung und wählt jemanden, der die Kommunikation innerhalb des Teams leitet. Für die Umsetzung der ersten Phase hat die Klasse circa zwei bis sechs Unterrichtsstunden Zeit. Wenn die erste Phase als Hausaufgabe bearbeitet wird, gibt es einen festen Abgabetermin. In der ersten Phase erstellt jede Gruppe ein Dokument, in dem sie schriftlich alle wichtigen Überlegungen sowie die ausgewählte Strategie mit ihrer Herleitung darlegt. Diese schriftliche Dokumentation der eigenen Vorgehensweise wird an alle anderen Gruppen verteilt. Danach trägt jede Gruppe der kompletten Klasse ihre Vorgehensweise vor. Nach jedem Referat wird im Plenum über die Vorschläge der Gruppe eine ausführliche Diskussion geführt. Es ist wünschenswert, dass sich daraus noch Korrektur- und Änderungsvorschläge entwickeln, die anschließend in eine überarbeitete Dokumentation eingearbeitet werden. Diese wird erneut in der Klasse diskutiert.

Nach den beiden Diskussionsrunden arbeitet jede Gruppe am Automatenentwurf separat weiter. Wieder wird die Vorgehensweise ausführlich schriftlich dokumentiert und begründet. Die so hergestellten Aufzeichnungen werden erneut an alle verteilt. In einem Abschlussvortrag präsentiert jede Gruppe die Ergebnisse ihrer Arbeit.

Hinweis für die Lehrperson Die Qualität der Abschlusspräsentation kann in die Bewertung einbezogen werden. Für die Bewertung ist jedoch der Inhalt der schriftlichen Dokumentation und damit auch die verständliche Darstellung des Lösungsweges maßgeblich.

Der Rest dieses Abschnittes ist folgendermaßen gegliedert: Zuerst wird eine Beispielaufgabe mit einem Link zu einer ausführlichen Musterlösung präsentiert. Danach werden mittels Aufgaben weitere Fragestellungen vorgeschlagen, die sich zur Projektbearbeitung eignen.

Beispiel 4.1 Projektaufgabe zur Steuerung einer T-Kreuzung

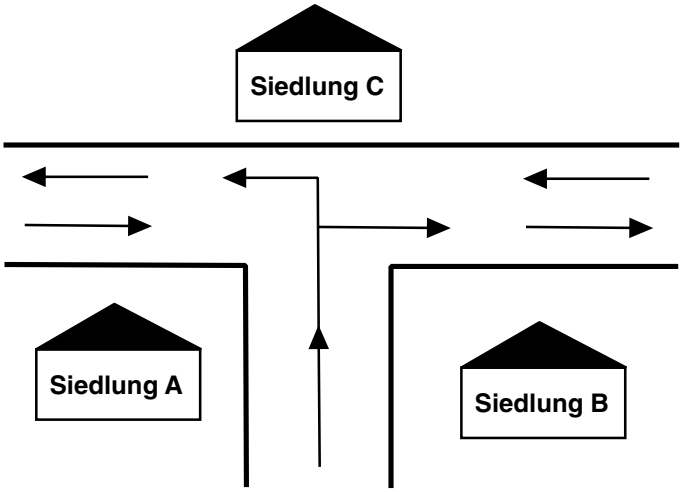


Abbildung 4.1

Allgemeine Situation Es gibt eine T-Kreuzung mit den drei Siedlungen A, B und C wie in Abb. 4.1 dargestellt. Die Straße zwischen den Siedlungen A und B ist eine Einbahnstraße in Fahrtrichtung zur Kreuzung. Die Wagen dürfen von dieser Straße links und rechts abbiegen. Diese Straße ist nicht so intensiv befahren wie die andere, die eine stark frequentierte Hauptstraße ist und auf der Autos in beide Richtungen fahren. Es

gibt weder eine Brücke noch einen Tunnel zwischen den Siedlungen und es ist nicht beabsichtigt, eine zu bauen. Es sollen Fußgängerüberquerungen mit Ampeln gebaut werden, die jedem Fußgänger ermöglichen, aus einer Siedlung in eine andere zu gelangen, möglicherweise auch durch einen Umweg. Für die entstehende T-Kreuzung muss ein Steuerungsmechanismus in Form eines Automaten gebaut werden.

Anforderungen Die folgenden Anforderungen müssen unbedingt berücksichtigt werden.

1. Die Ampelsteuerung darf keine Kollisionen verursachen (z. B. alle Verkehrsteilnehmer haben gleichzeitig Grün).

2. Kein Verkehrsteilnehmer muss bei funktionierender Ampelsteuerung unendlich lange auf grünes Licht warten.

Gerechtigkeit

1. Es gibt keine Teilnehmer, die unverhältnismäßig länger als andere auf Grün warten müssen.

2. Kein Verkehrsteilnehmer soll bei Rot warten, wenn er alleine an der Kreuzung ist.

3. Die stärker befahrene Hauptstraße soll Vorfahrt haben, um die Wagen auf dieser Straße nicht unnötig aufzuhalten und um einen Stau zu verhindern.

Freiheitsgrade

1. Du kannst entscheiden, wie viele Zebrastreifen es geben soll und wo sie platziert werden.

2. Du kannst Druckknöpfe für Fußgänger, Gewichtssensoren oder Kameras einbauen, um eine Übersicht über die wartenden Verkehrsteilnehmer zu erhalten.

Eine mögliche Musterlösung mit farbigen Bildern von Susanne Čech, Susanne Kasper, Barbara Keller und Björn Steffen steht auf der Seite

`http://www.ite.ethz.ch/kids/index`,

wo man dann unter „Unterrichtsmaterialien für Informatik“ das „Leitprogramm Ampelsteuerung für drei Siedlungen“ anklicken kann. □

Aufgabe 4.1 Projektaufgabe zur Steuerung eines Liftes

Allgemeine Situation In einem Haus mit vier Stockwerken wird ein Lift gebaut, dessen maximale Tragkraft 600 kg ist. In jedem Stockwerk kann man den Lift rufen. Im Fahrstuhl gibt es vier Knöpfe, um die Zieletage bestimmen zu können.

Anforderungen

1. Außer bei einem unvorhergesehenen Andrang darf niemand nach dem Drücken des Anforderungsknopfes lange auf den Fahrstuhl warten.

2. Der Fahrstuhl fährt nicht bei Überlast.

3. Der Fahrstuhl darf nicht mit geschlossenen Türen in einem Stockwerk stehen bleiben, wenn sich in ihm Passagiere befinden.

4. Der leere Fahrstuhl darf nicht in einem Stockwerk warten, obwohl er aus einer anderen Etage gerufen wird.

Gerechtigkeit

1. Niemand soll im Lift nochmals an seiner Einstiegsetage vorbeifahren, ohne vorher in seinem gewünschten Stockwerk anzuhalten.

Deine Aufgabe ist jetzt zuerst, weitere sinnvolle Kriterien für „fairness“ zu formulieren. Danach sollst du in zwei Phasen, wie oben beschrieben, einen Automaten für die Liftsteuerung entwerfen. Dabei darfst du entscheiden, ob der Fahrstuhl einen Überlastsensor oder eine feinere Waage hat. Am Einstieg darf ein Rufknopf sein oder maximal zwei, um die gewünschte Fahrtrichtung anzuzeigen.

Aufgabe 4.2 Projektaufgabe zur Kreuzungssteuerung

Allgemeine Situation Wir haben eine Kreuzung zwischen einer Einbahnstraße und einer Zweibahnstraße, wie in Abb. 4.2 dargestellt. Es führen zwei Fußgängerbrücken über die Hauptstraße. Man kann die Einbahnstraße an Zebrastreifen überqueren, die mit Ampeln ausgestattet sind. Für den Autoverkehr gibt es Ampeln aus allen Einfahrtsrichtungen. Es sollen Automaten zur Ampelsteuerung an dieser Kreuzung entworfen werden.

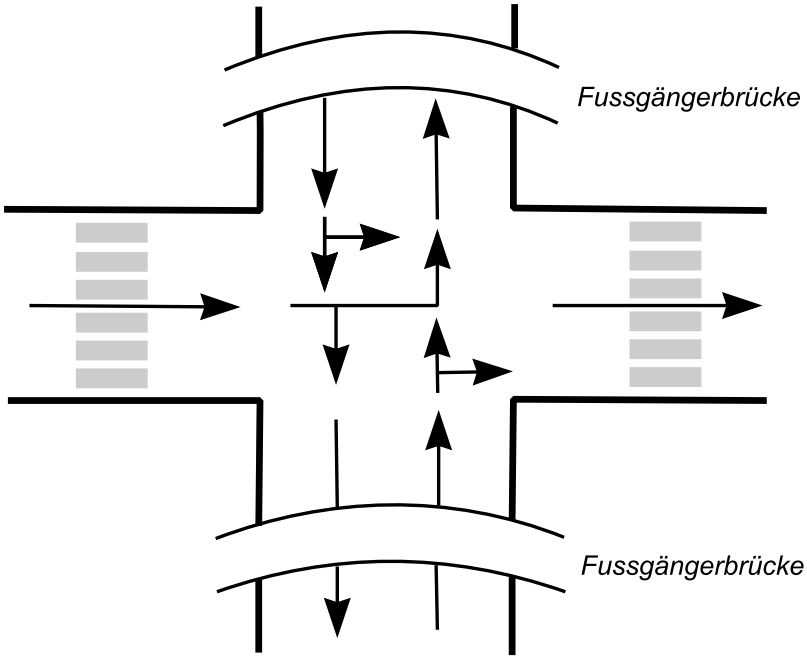


Abbildung 4.2

Anforderungen

1. Wenn zwei Verkehrsteilnehmer Grün haben, darf es zu keiner Kollision zwischen beiden kommen.

2. Kein Verkehrsteilnehmer soll bei funktionierender Ampelsteuerung unendlich lange warten.

Gerechtigkeit

1. Es gibt keine Teilnehmer, die unverhältnismäßig länger als andere auf grünes Licht warten müssen.

2. Kein Verkehrsteilnehmer soll bei Rot warten, wenn er der einzige Teilnehmer an der Kreuzung ist.

3. Die stark befahrene Hauptstraße soll Vorfahrt haben, um Fahrzeuge nicht unnötig aufzuhalten oder sogar einen Stau zu provozieren.

Freiheitsgrade

1. Du kannst über die Platzierung der Sensoren, wie z. B. Druckknöpfe für Fußgängerampeln, Kameras und Gewichtssensoren, selbst entscheiden.

2. Es ist deine Entscheidung, welche Ampel du vorziehst. Zum Beispiel: Grün für alle aus einer Richtung oder separate Signale für Links- oder Rechtsabbieger.

Entwirf den endlichen Automaten zur Ampelsteuerung dieser Kreuzung in zwei Phasen, wie oben beschrieben.

Aufgabe 4.3 Allgemeine Situation Es gibt zwei Zebrastreifen, die 200 m voneinander entfernt sind. Beide sollen Ampeln erhalten, die durch eine zentrale Steuerung miteinander synchronisiert sind. Zwischen den Zebrastreifen ist ein Halteverbot.

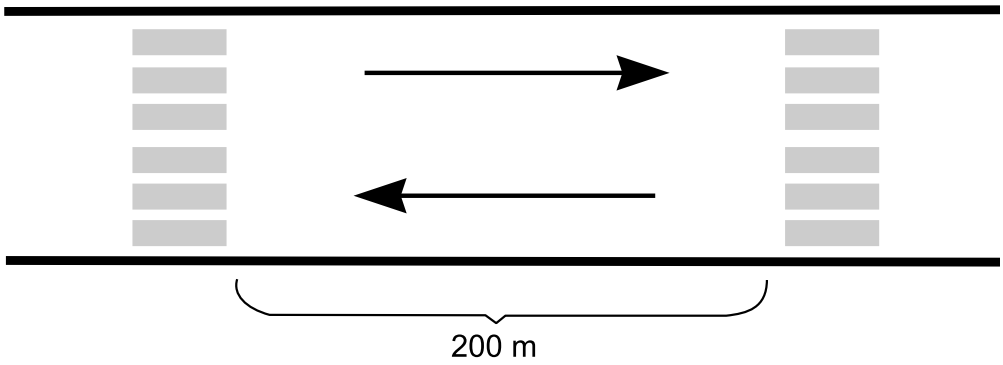


Abbildung 4.3

Anforderungen

1. Es kommt zu keiner Kollision durch gleichzeitiges Grünsignal für zwei Verkehrsteilnehmer.

2. Keiner wartet unendlich lange bei Rot.

Gerechtigkeit

1. Kein Verkehrsteilnehmer muss warten, wenn er der einzige Teilnehmer im System ist.

2. Nach Möglichkeit soll kein Auto zweimal an beiden Übergängen Rot bekommen und dort warten müssen.

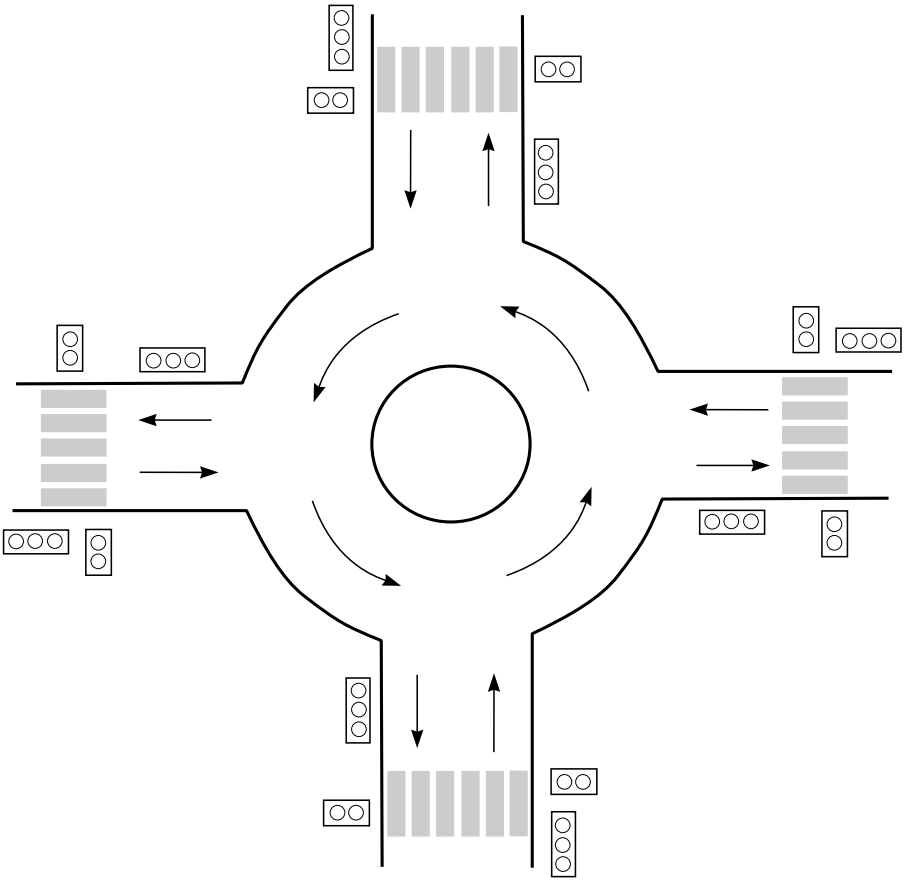


Abbildung 4.4

Aufgabe 4.4 Allgemeine Situation Es gibt einen Kreisverkehr mit vier einmündenden Straßen und der üblichen Vorfahrtsregelung für die Verkehrsteilnehmer, die sich im Kreisel befinden. Eigentlich braucht man keine Ampeln. Jetzt sollen aber in Entfernung von jeweils 100 m auf allen vier Straßen Fußgängerüberwege errichtet werden, so wie in Abb. 4.4 skizziert. Nun soll ein endlicher Automat zur Steuerung aller 16 Ampeln entworfen werden.

Anforderungen

1. Kollisionen zwischen Fußgängern und Fahrzeugen müssen bei korrektem Verhalten aller Teilnehmer ausgeschlossen sein.

2. Keiner darf unendlich lange bei Rot warten.

3. Kein Verkehrsteilnehmer soll bei Rot warten, wenn er der einzige Verkehrsteilnehmer im System ist.

Gerechtigkeit

1. Wenn der Kreisverkehr nicht überlastet ist, soll kein Fahrzeug zweimal bei Rot stehen bleiben müssen.

2. Kein Verkehrsteilnehmer soll wesentlich länger bei Rot warten als ein anderer.

Hinweis für die Lehrperson Bei der Durchführung des Projektes in zwei Phasen ist es interessant, auch folgende Variante zu berücksichtigen: Zwei Gruppen können dieselbe Projektaufgabe bearbeiten. Diese Alternative führt zu einer lebhafteren Diskussion nach der ersten Phase. Das Ziel dieser Diskussion ist dann jedoch nicht, sich auf eine Lösung zu einigen. Es sollen vielmehr zwei Lösungsstrategien in die zweiten Phase integriert und am Schluss der Lektion nochmals verglichen werden.

Lektion 5

Induktionsbeweise der Korrektheit

Hinweis für die Lehrperson Diese Lektion kann ohne Folgen für die Bearbeitung nachfolgender Lektionen übersprungen werden. Wenn man sich nur mit dem Automatenentwurf beschäftigen will, kann auf die formale mathematische Überprüfung der korrekten Funktionalität der endlichen Automaten verzichtet werden. Wenn man sich entscheidet, diese Lektion zu überspringen, empfehlen wir mindestens den Begriff der Verifikation einzuführen und seine Wichtigkeit anzusprechen.

Die Lektion 3 vermittelte, wie man endliche Automaten systematisch entwerfen kann. Als Basis dafür diente die Partitionierung der Menge aller Eingabewörter in die Zustandsklassen. Wie wir gesehen haben, ist dies für den Automatenentwurf sowie zum vollständigen Funktionsverständnis eines gegebenen (schon entworfenen) endlichen Automaten hilfreich. Dieser Abschnitt zeigt nun, dass die Zuordnung der Wortklassen zu den Zuständen auch die Beweisgrundlage für die Korrektheit des entworfenen Automaten ist. Was bedeutet es,

> *die Korrektheit eines endlichen Automaten A für eine Sprache L zu beweisen?*

Es heißt, formal zu zeigen (zu begründen), dass

$$L = L(A)$$

gilt. Ähnlich wie beim Entwurf und der Implementierung von Algorithmen ist die typische Einstellung des Designers, dass das entworfene Objekt (Automat oder Programm) selbstverständlich genau das tut, was er will. Diese trügerische Sicherheit ist aber sehr gefährlich und riesige Investitionen sind durch den Glauben an die scheinbare Korrektheit

J. Hromkovič, *Lehrbuch Informatik*, DOI 10.1007/978-3-8348-9692-6_26,

verloren gegangen. Bei großen Programmen ist der Aufwand für einen Korrektheitsbeweis viel umfangreicher als der Aufwand, der mit dem Entwurf des Programms verbunden ist. Deswegen verzichtet man gewöhnlich auf eine vollständige Überprüfung der Korrektheit und beschränkt sich auf die Fehlersuche mittels ausgearbeiteter Testläufe. Daher darf man sich nicht wundern, wenn große Softwaresysteme in der Regel tausende Fehler beinhalten, die in gewissen Spezialfällen[1] auftreten können. Solche selten auftretenden Fehler sind innerhalb komplexer Systeme schwer und aufwendig zu finden, weshalb man gewöhnlich zuerst auf ihre Suche verzichtet. Erst wenn sie zu einem falschen Verhalten in den laufenden Anwendungen führen, wird versucht sie zu beheben.

Bei endlichen Automaten ist die Sachlage oft anders. Endliche Automaten benutzt man für die Steuerung von Verkaufsautomaten, Ampeln, Aufzügen usw. Ein entworfener Automat wird oft in Hardware integriert und tausendfach vervielfältigt. Ein fehlerhaftes Verhalten in den vielen Kopien hätte dann große ökonomische Konsequenzen. Daher lohnt sich die Investition in die vollständige Überprüfung der Korrektheit.

Hinweis für die Lehrperson Diese Lektion eignet sich sehr gut für das Üben der Führung von Induktionsbeweisen. Man kann kaum einfachere Aufgaben zu Induktionsbeweisen finden als bei der Verifikation von endlichen Automaten. Die Voraussetzung für eine erfolgreiche Bearbeitung nachfolgender Lektionsteile ist das Verständnis des Begriffs der Implikation, so wie wir ihn im Modul „Geschichte und Begriffsbildung“ eingeführt haben.

Bei endlichen Automaten werden wir die Korrektheitsbeweise mittels Induktion durchführen. Deswegen rekapitulieren wir jetzt zuerst die Induktionsbeweise.

Mit Induktionsbeweisen belegt man die Gültigkeit einer gegebenen Behauptung für alle natürlichen Zahlen. Das Schema des Induktionsbeweises sieht wie folgt aus:

Sei $B(i)$ eine Behauptung für eine natürliche Zahl i. Unsere Zielsetzung[2] ist

$$B(i) \textit{ gilt für alle } i \in \mathbb{N}$$

zu beweisen. Beachte, dass $\mathbb{N}$ unendlich ist und wir nicht unendliche viele, einzelne Beweise für jedes i durchführen können.

Den Beweis führt man in diesen zwei Schritten aus:

[1] Oft sind es Fälle, mit denen niemand gerechnet hat.

[2] Es gibt auch allgemeinere Darstellungen von Induktionsbeweisen als die hier präsentierten.

Schritt 1 Induktionsanfang

Beweise, dass $B(0)$ gilt!

{Manchmal ist es erforderlich oder günstiger, am Anfang für ein festes[3] k die ersten k Behauptungen $B(0), B(1), B(2), \ldots, B(k)$ zu beweisen.}

Schritt 2 Induktionsschritt

Beweise für alle $n \in \mathbb{N} - \{0\}$:

Wenn $B(0), B(1), \ldots, B(n-1)$ gelten, dann muss auch $B(n)$ gelten.

Bemerkung: Man kann anstelle von $B(0)$ auch mit $B(1)$ oder sogar mit $B(k)$ beginnen, um zu beweisen, dass für alle $n \geq k$ die Behauptung $B(k)$ gilt.

Die Methode der Induktionsbeweise ist anschaulich. Unser Ziel ist der Beweis von $B(0), B(1), B(2), \ldots$. Zuerst beweisen wir im Schritt 1 (Induktionsanfang), dass $B(0)$ gilt. Wenn wir im Schritt 2 die Zahl n gleich 1 setzen, liefert uns der Induktionsschritt, dass $B(0)$ die Gültigkeit von $B(1)$ impliziert und somit $B(1)$ gilt. Dann wenden wir den Induktionsschritt für $n = 2$ an. Wir haben schon die Gültigkeit von $B(0)$ und $B(1)$ bewiesen. Nach dem Induktionsschritt muss jetzt auch $B(2)$ gelten. Wenn wir so weiter verfahren, wissen wir, dass alle $B(0), B(1), B(2), \ldots$ gelten. Deswegen reicht es zum Beweis von

$B(i)$ gilt für alle $i \in \mathbb{N}$,

die Schritte 1 und 2 des Beweisschemas durchzuführen.

Beispiel 5.1 Wir wollen den „kleinen Gauß“ beweisen.

Für alle $n \in \mathbb{N} - \{0\}$ beträgt die Summe der ersten n positiven ganzen Zahlen $\frac{n \cdot (n+1)}{2}$.

Anders formal beschrieben soll gezeigt werden, dass

$$\text{für alle } n \in \mathbb{N} - \{0\} : \sum_{i=1}^{n} i = \frac{n \cdot (n+1)}{2}$$

gilt. Wir beweisen es mittels Induktion bezüglich n. Sei **Gauß(*n*)** die Behauptung

$$\sum_{i=1}^{n} i = \frac{1}{2} n \cdot (n+1)$$

für jedes feste n.

[3] also für endlich viele, kleinste Fälle

Schritt 1 Sei $n = 1$. Dann gilt $\sum_{i=1}^{1} 1 = 1$ und $\frac{n\cdot(n+1)}{2} = \frac{1\cdot 2}{2} = 1$ und somit ist Gauß(1) gültig.

Schritt 2 Sei $n \geq 2$, und seien Gauß(1), ..., Gauß($n-1$) gültig.
Unsere Aufgabe ist jetzt zu beweisen, dass auch Gauß(n) gelten muss. Weil Gauß($n-1$) gilt, gilt auch

$$\sum_{i=1}^{n-1} i \quad = \quad \frac{(n-1)\cdot((n-1)+1)}{2} = \frac{(n-1)\cdot n}{2} \tag{5.1}$$

Jetzt können wir rechnen:

$$\begin{aligned}
\sum_{i=1}^{n} i \quad &= \quad n + \sum_{i=1}^{n-1} i \\
&\quad \{\text{Der letzte Summand wurde aus der Summe entfernt}\} \\
&\overset{(3.10)}{=} \quad n + \frac{(n-1)n}{2} \\
&\quad \{\text{Wir haben die Gültigkeit von Gauß}(n-1) \text{ für die Summe} \\
&\quad \text{der ersten } n-1 \text{ positiven ganzen Zahlen angewendet}\} \\
&= \quad n \cdot \left(1 + \frac{n-1}{2}\right) \\
&\quad \{\text{Distributivgesetz}\} \\
&= \quad n \cdot \frac{2+n-1}{2} = n \cdot \frac{n+1}{2} \\
&= \quad \frac{n\cdot(n+1)}{2}
\end{aligned}$$

□

Aufgabe 5.1 Beweise mittels Induktion die Gültigkeit folgender kombinatorischen Relationen:

(i) Für alle $n \in \mathbb{N} - \{0,1,2,3\}$: $n! \geq 2^n$.

(ii) Für alle $n \in \mathbb{N} - \{0\}$: $\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$.

(iii) Für alle $n \in \mathbb{N} - \{0\}$: $1 + 3 + 5 + \ldots + (2n-1) = n^2$.

Warum braucht man ausgerechnet Induktionsbeweise, um $L = L(A)$ für eine gegebene

Sprache L und einen konstruierten EA A zu beweisen? Oder genauer gesagt: „Was hat die Behauptung $L = L(A)$ mit der Induktion zu tun?“

Der erste Ansatz ist, $L = L(A)$ nicht direkt zu beweisen, aber ein allgemeineres Resultat zu zeigen, dessen Folge $L = L(A)$ ist. Wir bestimmen für jeden Zustand q von A die entsprechende Wortklasse Klasse(q) und beweisen die Richtigkeit dieser Bestimmung. Den Beweis führen wir mittels Induktion bezüglich der Wortlänge. Dies bedeutet, die Gültigkeit der Klassenzuordnung zuerst für kurze Wörter zu beweisen und sie dann durch den Induktionsschritt auf alle Wörter zu erweitern.

Beispiel 5.2 Betrachten wir die folgende Sprache

$$L = \{x \in \{0,1\}^* \mid |x|_0 \text{ mod } 4 \text{ ist gerade}\}.$$

Gemäß Abschnitt 3.3 reichen vier Zustände aus, um die Reste nach der Teilung der Anzahl Nullen in x modulo 4 speichern zu können. Wir erhalten dann den EA A aus Abb. 5.1.

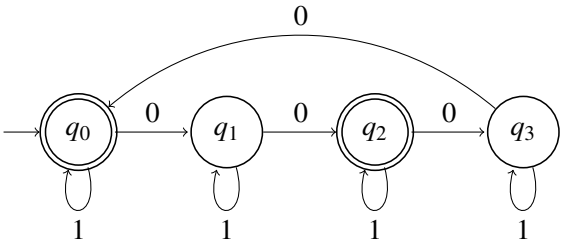


Abbildung 5.1 EA A

Unsere Hypothese ist, dass

$$\begin{array}{lcll} \text{Klasse}[q_0] & = & \text{Rest}_0 & = \{x \in \{0,1\}^* \mid |x|_0 \text{ mod } 4 = 0\} \\ \text{Klasse}[q_1] & = & \text{Rest}_1 & = \{x \in \{0,1\}^* \mid |x|_0 \text{ mod } 4 = 1\} \\ \text{Klasse}[q_2] & = & \text{Rest}_2 & = \{x \in \{0,1\}^* \mid |x|_0 \text{ mod } 4 = 2\} \\ \text{Klasse}[q_3] & = & \text{Rest}_3 & = \{x \in \{0,1\}^* \mid |x|_0 \text{ mod } 4 = 3\} \end{array}$$

gilt.
Sei **Hyp(i)** die Behauptung, dass diese Klassenzuordnung korrekt[4] für Wörter der Länge

[4]Formal ausgedrückt, für alle $i = 0,1,2,3$ gilt
$\text{Klasse}[q_i] \cap \{0,1\}^i = \text{Rest}_i \cap \{0,1\}^i = \{x \in \{0,1\}^i \mid |x|_0 \text{ mod } 4 = i\}$.

i ist. Genauer ist **Hyp(i)** die Gültigkeit der Gleichung

$$\text{Klasse}[q_k] \cap \{0,1\}^i = \text{Rest}_k \cap \{0,1\}^i$$

für alle $k = 0,1,2,3$. Und weil

$$L(A) = \text{Klasse}[q_0] \cup \text{Klasse}[q_2]$$

offensichtlich ist, ist $L = L(A)$ eine direkte Folge der Gültigkeit unserer Hypothese, die wir als unendliche Folge Hyp(0), Hyp(1), Hyp(2),... von Behauptungen dargestellt haben. Also reicht es aus, unsere Hypothese zu beweisen.

Aufgabe 5.2 Betrachte die Sprache $L = \{w \in \{0,1\}^* \mid |u|_1 \bmod 3 = 0\}$. Entwerfe einen EA M für diese Sprache und formuliere eine Induktionshypothese, mit der man $L = L(M)$ beweisen kann.

Aufgabe 5.3 Entwerfe einen endlichen Automaten M für die Sprache $L = \{w \in \{a,b\}^* \mid (|u|_a - |u|_b) \bmod 4 = 1\}$. Formuliere eine Induktionshypothese, mit der man $L = L(M)$ beweisen kann.

Schritt 1 Induktionsanfang

Um erfolgreich zu starten, können wir die Korrektheit der Hypothese für so viele Wortlängen überprüfen, bis in jeder Klasse mindestens ein Wort ist. Um es besser zu verstehen, betrachten wir den Zustand q_3. Das kürzeste Wort in der Klasse$[q_3]$ ist 000. Also muss man, um den Induktionsbeweis für die Klasse$[q_3]$ durchzuführen, mit der Wortlänge 3 anfangen[5].

Beweisen wir also hier Hyp(0), Hyp(1), Hyp(2), Hyp(3). Man kann alle 15 Wörter mit einer maximalen Länge von 3 durch den Automaten bearbeiten lassen (die Berechnung des EA auf den Wörtern durchführen) und überprüfen, ob unsere Hypothese stimmt. Weil es Routinearbeit ist, zeigen wir es nur für ein Wort $x = 010 \in \text{Rest}_2$. Die Berechnung auf 010 ist

$$(q_0,010) \vdash_A (q_1,10) \vdash_A (q_1,0) \vdash_A (q_2,\lambda).$$

Somit ist 010 in der Klasse$[q_2]$, was unserer Hypothese **Hyp**(3) entspricht.

Schritt 2 Wir setzen die Gültigkeit von Hyp($n-1$) voraus und belegen dann die Gültigkeit von Hyp(n).

Wir müssen also zeigen, dass unsere Hypothese für jedes Wort der Länge n gilt, vorausgesetzt, sie ist auch für kürzere Wörter gültig.

Sei x ein beliebiges Wort aus $\{0,1\}^n$. Wir unterscheiden zwei Möglichkeiten bezüglich des letzten Bits von x.

[5] Wo man es genau braucht, sehen wir im zweiten Schritt.

(i) Sei $x = y1$ für ein $y \in \{0,1\}^{n-1}$.
Weil y die Länge $n-1$ hat, ist y nach Hyp$(n-1)$ unserer Hypothese folgend in der richtigen Klasse. Also wenn $y \in$ Rest$_j$ für ein $j \in \{0,1,2,3\}$ gilt, dann ist laut der Induktionshypothese Hyp$(n-1)$ $y \in$ Klasse$[q_j]$. Weil

$$x = y1 \text{ und somit } |x|_0 = |y|_0,$$

ist, liegt x auch in der Klasse Rest$_j$. Wir brauchen also nur zu zeigen, dass auch $x \in$ Klasse$[q_j]$ ist. Dies ist aber offensichtlich, weil $\delta_A(q_i,1) = q_i$ für alle $i \in \{0,1,2,3\}$ gilt (beim Lesen einer 1 wird der Zustand nicht geändert). Damit impliziert

$$(q_0, y) \vdash_A^* (q_j, \lambda)$$

direkt die folgende Berechnung von A auf $x = y1$:

$$(q_0, y1) \vdash_A^* (q_j, 1) \vdash_A (q_j, \lambda)$$

und somit ist $x \in$ Klasse$[q_j]$.

(ii) Sei $x = y0$ für ein $y \in \{0,1\}^{n-1}$.
Sei $|y|_0 \bmod 4 = j$ für ein $j \in \{0,1,2,3\}$, (d.h. $y \in$ Rest$_j$). Weil $|y| = n-1$ ist, liefert uns die Induktionshypothese Ind$(n-1)$, dass $y \in$ Klasse$[q_j]$.

(ii.1) Betrachten wir zuerst den Fall $j \in \{0,1,2\}$.
Weil wegen $x = y0$

$$|x|_0 = |y|_0 + 1$$

gilt, ist somit

$$\begin{aligned} |x|_0 \bmod 4 &= (|y|_0 + 1) \bmod 4 \\ &= (|y|_0 \bmod 4 + 1) \bmod 4 \\ &\quad \{\text{weil } (a + b) \bmod c = (a \bmod c + \\ &\quad b \bmod c) \bmod c\} \\ &= (j+1) \bmod 4 \\ &= j+1 \\ &\quad \{\text{weil } j \in \{0,1,2\}\}. \end{aligned}$$

Damit ist x in Rest$_{j+1}$.
Wir müssen zeigen, dass $x \in$ Klasse$[q_{j+1}]$ ist. Dies ist aber offensichtlich,

weil $\delta_A(q_j,0) = q_{j+1}$ für alle $j \in \{0,1,2\}$ gilt. Daraus ergibt sich die folgende Berechnung von A auf $x = y0$

$$(q_0, y0) \vdash_A^* (q_j,0) \vdash_A (q_{j+1},\lambda).$$

(ii.2) Es bleibt noch der Fall $j = 3$ übrig. Die Vervollständigung des Beweises dafür wird dem Leser überlassen.

□

Wenn man den vorgestellten Beweis der Korrektheit des endlichen Automaten in Beispiel 5.2 betrachtet, dann sieht man, dass im Induktionsschritt jede gerichtete Kante des Automaten genau einmal überprüft wird. Mit „Überprüfung“ ist gemeint, dass man den Beweis in genauso viele Fälle zerlegen kann, wie es Kanten gibt. Die einzelnen Kanten werden folgendermaßen überprüft:
Wenn $x \in \{0,1\}^{n-1}$ nach Hyp$(n-1)$ in einer Klasse$[q]$ ist, dann sind auch $x0$ und $x1$ dank korrektem $\delta(q,0) = r$ und $\delta(q,1) = s$ in der richtigen Klasse.

Aufgabe 5.4 Führe den Induktionsschritt ausführlich für den Fall (ii.2) durch.

Aufgabe 5.5 Ist der EA in Abb. 5.1 der kleinste Automat für

$$L = \{x \in \{0,1\}^* \mid |x|_0 \bmod 4 \text{ ist gerade}\}?$$

Falls nicht, entwirf einen EA für L mit weniger Zuständen und beweise seine Korrektheit.

Aufgabe 5.6 Entwirf endliche Automaten für folgende Sprachen und beweise mittels Induktion die Korrektheit dieser Automaten:

(a) $\{x \in \{0,1\}^* \mid |x|_1 \bmod 3 = 0\}$,

(b) $\{x \in \{0,1,2\}^* \mid |x| \geq 3\}$,

(c) $\{x \in \{0,1\}^* \mid |x| \text{ ist gerade}\}$,

(d) $\{x \in \{0,1\}^* \mid |x| \text{ ist ungerade und } x = 1y \text{ für ein } y \in \{0,1\}^*\}$,

(e) $\{x \in \{0,1,2\}^* \mid |x|_2 \bmod 5 \in \{1,2,4\}\}$,

(f) $\{x \in \{0,1,2\}^* \mid 2 \leq |x| \leq 5\}$,

(g) $\{x \in \{0,1,2\}^* \mid |x| \leq 3 \text{ oder } |x| \geq 6\}$

(h) $\{x \in \{a,b\}^* \mid (|x|_a - |x|_b) \bmod 4 = 1\}$.

Zusammenfassung

Unter der Verifikation eines endlichen Automaten M verstehen wir eine Überprüfung der richtigen Funktionalität von M. In der formalen Umsetzung zeigt man, dass $L(M)$ die Sprache ist, die man erkennen will.

Die Verifikation führen wir meistens mittels Induktionsbeweisen durch. Man verwendet sie, um die Gültigkeit einer konkreten Behauptung $B(i)$ für alle natürlichen Zahlen i zu belegen. Im Fall von Induktionsbeweisen der Korrektheit werden die Beweise anhand der Eingabelänge durchgeführt. Im Induktionsschritt zeigen wir, dass ein endlicher Automat, der sich bei allen Wörtern mit einer maximalen Länge $n-1$ korrekt verhält, auch richtig auf Eingaben der Länge n reagiert. Durch diesen Beweis wird im Prinzip jede einzelne Kante des Automaten dahingehend überprüft, ob sie zwei Zustandsklassen korrekt verbindet.

Kontrollfragen

1. Warum ist die Verifikation von Informatikprodukten wichtig?

2. Aus welchen zwei Schritten besteht ein Induktionsbeweis?

3. Wie ist die allgemeine Strategie für den Beweis der richtigen Funktionalität eines entworfenen endlichen Automaten?

4. Sei M ein endlicher Automat mit m Zuständen, der über dem Alphabet Σ arbeitet. Warum kann man den Beweis des Induktionsschrittes auf $m \cdot |\Sigma|$ viele Fälle verteilen?

Kontrollaufgaben

1. Beweise mittels Induktion, dass für alle positiven ganzen Zahlen $2^{2n} > \binom{2n}{n}$ gilt. Kann man diese Tatsache elegant und ohne Induktionsbeweis begründen?

2. Entwirf jeweils einen endlichen Automaten für die folgenden Sprachen und beweise seine Korrektheit!

   a) $\{x111y \mid x,y \in \{0,1\}^*\}$,

   b) $\{x \in \{a,b\}^* \mid (|x|_a - 2|x|_b) \bmod 3 = 0\}$,

   c) $\{011, 101, \lambda\}$.

Lösungen zu ausgesuchten Aufgaben

Aufgabe 5.1 (ii)
Für alle $n \in \mathbb{N} - \{0\}$ haben wir die folgende Induktionshypothese

$$\text{HYP(n)}: \sum_{i=0}^{n} 2^i = 2^{n+1} - 1$$

Schritt 1 Wir beweisen HYP(1)

$$\sum_{i=0}^{1} 2^i = 2^0 + 2^1 = 3$$

Andererseits $2^{1+1} - 1 = 2^2 - 1 = 3$. Wir sehen, dass HYP(1) gilt.

Schritt 2 Wir setzen voraus, dass HYP(1), HYP(2), ... , HYP(n-1) gelten und beweisen, dass auch HYP(n) gilt. Bemerke, dass die Aussage von HYP(n-1) folgender Behauptung entspricht:

$$\sum_{i=0}^{n-1} 2^i = 2^{(n-1)+1} - 1 = 2^n - 1.$$

Jetzt rechnen wir wie folgt:

$$\begin{aligned} \Sigma_{i=0}^{n} 2^i &= 2^n + \sum_{i=0}^{n-1} 2^i \\ &= 2^n + 2^n - 1 \\ &\{\text{weil HYP(n-1) gilt}\} \\ &= 2 \cdot (2^n) - 1 = 2^{n+1} - 1. \end{aligned}$$

Damit haben wir gezeigt, dass die Gültigkeit von HYP(n-1) die Gültigkeit von HYP(n) impliziert. Dies vervollständigt den Induktionsbeweis. □

Lektion 6

Simulation und modularer Entwurf endlicher Automaten

In der dritten Lektion haben wir gelernt, wie man systematisch endliche Automaten entwerfen kann und dabei bedenkt, welche Merkmale eines bisher gelesenen Wortes gespeichert (unterschieden) werden müssen. Außerdem wurde vermittelt, wie man diese "gespeicherten" Informationen durch Zustände repräsentieren kann. Die zu akzeptierende Sprache könnte komplex in dem Sinne sein, dass sie durch mehrere Bedingungen (Eigenschaften von Wörtern) angegeben wird. Dann kann, durch die Anzahl der zu beobachtenden Worteigenschaften, der Automatenentwurf für zu kompliziert und undurchschaubar angesehen werden. Diese Lektion nun vermittelt, wie man durch das Konzept des modularen Entwurfs immer den Überblick behält.

Im Allgemeinen basiert eine modulare Entwurfstechnik auf der Herstellung von einfachen Bausteinen, die man Module nennt und aus denen man das zu entwerfende System zusammensetzt. Die modulare Entwurfstechnik kann mehrere Stufen haben. Aus einfachen Bausteinen kann man kompliziertere Module bauen, aus denen wiederum noch komplexere Systeme zusammengesetzt werden können. Für den Entwurf von elektrischen Schaltkreisen und VLSI Chips wird diese Methode beispielsweise häufig angewendet. Sie macht nicht nur den Entwurfsprozess übersichtlicher und strukturierter, sondern vereinfacht auch die Testphase. Hier überprüft man zuerst die Korrektheit der einzelnen einfachen Module und dann erst die der Modulzusammensetzung.

Wir betrachten die Anwendung der modularen Entwurfsmethode, weil man viele reguläre Sprachen durch eine Sammlung von Worteigenschaften spezifizieren kann, die durch logische ORs oder logische ANDs verknüpft werden. Wäre es nicht möglich, einfache

J. Hromkovič, *Lehrbuch Informatik*, DOI 10.1007/978-3-8348-9692-6_27,

endliche Automaten für einzelne Worteigenschaften zu entwerfen und dann aus diesen Automaten den endlichen Automaten für die gegebene Sprache zu bauen? Die Antwort ist positiv, denn man kann einen EA bauen, der die Arbeit zweier oder mehrerer endlicher Automaten simulieren kann. Um das zu verstehen, widmen wir uns zuerst dem Begriff „Simulation“.

Der Begriff **Simulation** hat in der Informatik eine breitgefächerte Bedeutung, die auf der unterschiedlichen Auslegung dieses Begriffs beruht. Eine sehr weitgefasste Interpretation von Simulation ist die Blackbox-Simulation. Man simuliert ein Verfahren (Programm) so, dass das Eingabe-Ausgabe Verhalten übereinstimmt. Ein Programm A simuliert also ein Programm B, wenn A jeweils dieselben Resultate wie B liefert, unabhängig davon, durch welche Rechnerwege die Resultate ermittelt worden sind. Die engste und härteste Auslegung von Simulation fordert, dass jeder elementare Schritt des simulierten Programms durch genau einen Schritt des simulierenden Programms widergespiegelt wird.

Nachfolgend wird Simulation im engen Sinn gedeutet, weil wir nach dem Lesen jedes einzelnen Buchstabens wissen müssen, welche Merkmale (Eigenschaften) das bisher gelesene Präfix hat oder eben nicht hat.

Hier wird jetzt der Ansatz verfolgt, zwei endliche Automaten durch nur einen EA zu simulieren. Seien M_1 und M_2 zwei endliche Automaten mit den Zustandsmengen Q_1 und Q_2 und $L_1 = L(M_1)$ und $L_2 = L(M_2)$. Wir konstruieren einen neuen endlichen Automaten M mit der Zustandsmenge

$$Q = Q_1 \times Q_2 = \{(q,p) \mid q \in Q_1 \text{ und } p \in Q_2\}.$$

Ein Zustand von M besteht also aus zwei Teilen. Der erste Teil entspricht dem Zustand von M_1, der zweite Teil dem Zustand von M_2. Auf diese Weise kann M in seiner Konfiguration die vollständige Information über die beiden aktuellen Konfigurationen von M_1 und M_2 erhalten. Dies ist in Abbildung 6.1 anschaulich dargestellt. M_1 ist in der Konfiguration $(q, x_i \ldots x_n)$ und M_2 ist in der Konfiguration $(p, x_i \ldots x_n)$. Diese Information ist in der Konfiguration $((q,p), x_i \ldots x_n)$ von M vollständig enthalten. Die Berechnungsschritte

$$(q, x_i x_{i+1} \ldots x_n) \vdash_{M_1} (v, x_{i+1} \ldots x_n) \text{ und } (p, x_i x_{i+1} \ldots x_n) \vdash_{M_2} (s, x_{i+1} \ldots x_n)$$

von M_1 und M_2 simuliert M durch den Schritt

$$((q,p), x_i x_{i+1} \ldots x_n) \vdash_M ((v,s), x_{i+1} \ldots x_n).$$

M geht aus dem Zustand (q,p) beim Lesen des Symbols x_i in den Zustand (v,s) genau dann über $[\delta_M((q,p), x_i) = (v,s)]$, wenn

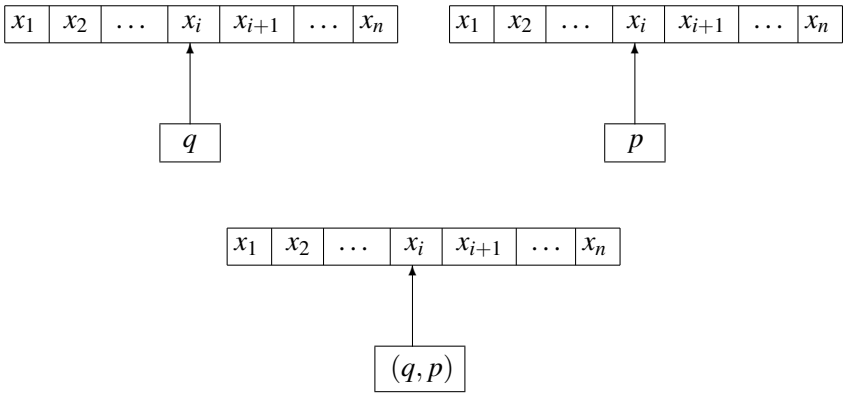


Abbildung 6.1

1. M_1 aus q beim Lesen von x_i in v übergeht $[\delta_{M_1}(q,x_i) = v]$, und

2. M_2 aus p beim Lesen von x_i in s übergeht $[\delta_{M_2}(p,x_i) = s]$.

Illustrieren wir diese Simulationsidee mit folgendem Beispiel:

Beispiel 6.1 Seien $L_1 = \{x \in \{0,1\}^* \mid |x|_0 \text{ ist gerade}\}$, und $L_2 = \{x \in \{0,1\}^* \mid |x|_1 = 0 \text{ oder } |x|_1 \geq 3\}$. Wir bauen zuerst zwei endliche Automaten M_1 und M_2 mit $L(M_1) = L_1$ und $L(M_2) = L_2$, die in Abbildung 6.2 und Abbildung 6.3 dargestellt sind.

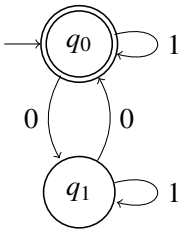


Abbildung 6.2 M_1

Dem Ansatz folgend, hat M die Zustandsmenge

$$\{(q_0,p_0),(q_0,p_1),(q_0,p_2),(q_0,p_3),(q_1,p_0),(q_1,p_1),(q_1,p_2),(q_1,p_3)\}.$$

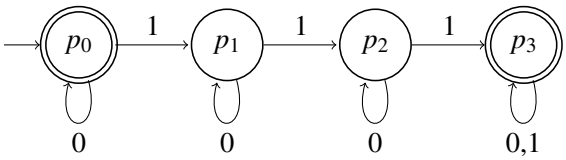


Abbildung 6.3 M_2

Um M übersichtlich zeichnen zu können, werden die Zustände von M auf einem Blatt Papier matrixartig angelegt (Abbildung 6.4). Die erste Zeile beinhaltet Zustände mit der ersten Komponente q_0 und die zweite Zeile Zustände mit q_1 als erste Komponente. Die i-te Spalte für $i = 0,1,2,3$ umfasst die Zustände mit der zweiten Komponente p_i.

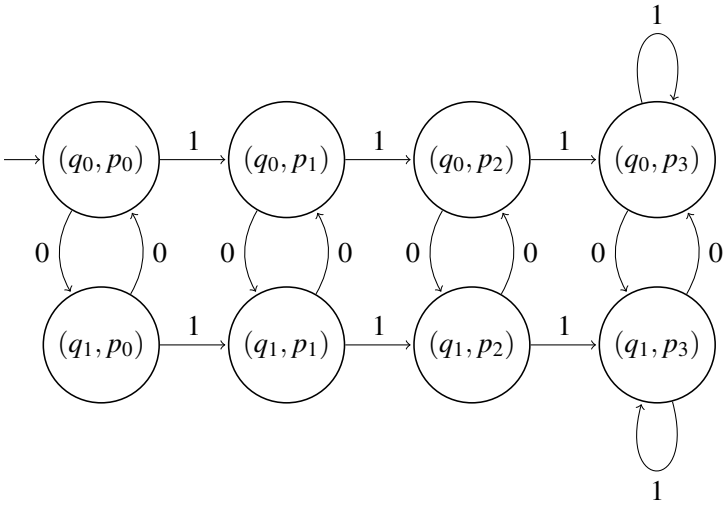


Abbildung 6.4 M

Jetzt muss man anhand von M_1 und M_2 die Kanten (Übergänge) bestimmen. Beispielsweise geht M_1 aus q_0 bei 0 in q_1 über und M_2 bleibt beim Lesen von 0 in p_0. Deswegen geht M aus (q_0,p_0) beim Lesen von 0 in (q_1,p_0) über.
Nachdem auf diese Weise alle Kanten gezeichnet worden sind, zeigt die Abbildung 6.4, wie man in den Spalten die Anzahl der Nullen modulo 2 rechnet und in den Zeilen die Anzahl Einsen von 0 bis 3 zählt. Also bedeutet der Zustand (q_i,p_j) für $i \in \{0,1\}, j \in \{0,1,2,3\}$, dass das bisher gelesene Präfix x genau j Einsen beinhaltet (wenn $j = 3$ mindestens drei Einsen hat) und $|x|_0 \mod 2 = i$. Damit wurden alle für uns

wichtigen Merkmale der gelesenen Wörter in M beobachtet (gespeichert).

Welche sind die akzeptierenden Zustände? Das hängt davon ab, welche Sprache wir akzeptieren wollen. Wenn wir zum Beispiel die Sprache

$$L_1 \cap L_2 = L(M_1) \cap L(M_2)$$

akzeptieren möchten, dann muss genau dann akzeptiert werden,

wenn beide M_1 und M_2 akzeptieren.

Das bedeutet: Die akzeptierenden Zustände von M sind genau die Zustände,

in denen beide Komponenten akzeptierende Zustände von M_1 und M_2 sind.

Weil q_0 der einzige, akzeptierende Zustand von M_1 ist und die akzeptierenden Zustände von M_2 p_0 und p_3 lauten, sind die akzeptierenden Zustände von M

die Zustände (q_0, p_0) und (q_0, p_3).

Wenn M die Sprache

$$L_1 \cup L_2 = L(M_1) \cup L(M_2)$$

akzeptieren soll, dann akzeptiert M genau dann, wenn

mindestens einer der endlichen Automaten M_1 und M_2 akzeptiert.

Das bedeutet: Die akzeptierenden Zustände von M sind genau die Zustände,

in denen mindestens eine Komponente einem akzeptierenden Zustand von M_1 oder M_2 entspricht.

Somit sind die akzeptierenden Zustände für $L_1 \cup L_2$ die folgenden Zustände:

$$(q_0, p_0), (q_0, p_1), (q_0, p_2), (q_0, p_3), (q_1, p_0), (q_1, p_3).$$

□

Aufgabe 6.1 Betrachte M aus Beispiel 6.1. Bestimme die Menge der akzeptierenden Zustände, wenn M die folgenden Sprachen akzeptieren soll.

a) $L(M_1) - L(M_2) = \{x \in \{0,1\}^* \mid x \in L(M_1) \text{ und } x \notin L(M_2)\}$

b) $L(M_2) - L(M_1)$

c) $\{0,1\}^* - (L(M_1) \cap L(M_2))$

d) $\{0,1\}^* - (L(M_1) \cup L(M_2))$

e) $(\{0,1\}^* - L(M_1)) \cup L(M_2)$

f) $(\{0,1\}^* - L(M_2)) \cap L(M_1)$.

Hinweis für die Lehrperson An dieser Stelle empfehlen wir die Verwendung der Puzzle-Methode. Die Klasse sollte in kleine Gruppen aufgeteilt werden. Jede Gruppe soll einen Automaten aus der nachfolgenden Aufgabe bauen. Am Ende muss jede Gruppe das Resultat und insbesondere ihre Überlegungen im Prozess des Automatenentwurfs der übrigen Klasse in einem Vortrag schildern.

Aufgabe 6.2 Nutze die Methode des modularen Entwurfs, um endlichen Automaten für folgende Sprachen zu bauen:

a) $\{x \in \{0,1\}^* \mid |x|_0 \bmod 3 = 1 \text{ und } |x|_1 \bmod 3 = 2\}$

b) $\{x \in \{0,1\}^* \mid |x|_0 \bmod 2 = 1 \text{ und } 1 \leq |x|_1 \leq 3\}$

c) $\{x \in \{0,1,2\}^* \mid |x|_0 \bmod 3 \in \{0,1\} \text{ und } (|x|_1 + |x|_2) \bmod 2 = 0\}$

d) $\{x \in \{0,1\}^* \mid |x|_1 \text{ ist gerade und } x \text{ enthält das Teilwort } 0101\}$

e) $\{x \in \{0,1\}^* \mid x = y00z11v \text{ für } y,z,v \in \{0,1\}^* \text{ und } |x|_0 \bmod 3 = 2\}$

f) $\{x \in \{0,1,a,b\}^* \mid x \text{ enthält das Teilwort 111 oder } x \text{ enthält das Teilwort } aba\}$

g) $\{x \in \{0,1\}^* \mid x \text{ beginnt mit dem Präfix 011 und } x \text{ enthält 100 als Teilwort}\}$

h) $\{x \in \{0,1\}^* \mid x \text{ enthält entweder 0101 als Teilwort oder endet mit dem Suffix 111}\}$

i) $\{x \in \{0,1,2\}^* \mid |x|_0 \text{ ist gerade } \wedge |x|_1 \text{ ist ungerade } \wedge |x|_1 + |x|_2 \geq 2\}$

j) $\{x \in \{0,1\}^* \mid x$ enthält mindestens eines der folgenden Teilwörter: 0011,110$\}$

Aufgabe 6.3 Betrachte die folgenden Sprachen:

$$L_1 = \{x \in \{0,1,2\}^* \mid |x|_0 \text{ ist gerade}\},$$
$$L_2 = \{x \in \{0,1,2\}^* \mid |x|_1 \text{ ist ungerade}\},$$
$$L_3 = \{x \in \{0,1,2\}^* \mid |x|_2 \text{ ist gerade}\}.$$

Baue drei endliche Automaten M_1, M_2 und M_3, so dass $L(M_1) = L_1, L(M_2) = L_2$ und $L(M_3) = L_3$ ist. Kannst du jetzt mit der modularen Entwurfstechnik einen EA M konstruieren, damit gilt

$$L(M) = (L_1 \cap L_2) \cup L_3 \ ?$$

Hinweis für die Klasse Es gibt eine große Aufgabenvielfalt zur Konstruktion von endlichen Automaten. Aus diesem Grund wurde das interaktive E-Learning-System „Exorciser“ aufgebaut. Dort gibt es viele Aufgabenstellungen. Du kannst die Automaten schnell auf dem Bildschirm entwerfen und der Rechner überprüft sie sofort auf Korrektheit. Bei einem fehlerhaften Automatenentwurf kannst du selbst die Art der Fehlermeldung wählen. Das System kann entweder nur melden, dass der EA falsch ist oder aber auch ergänzend, warum er nicht wie gewünscht funktioniert. Dann schreibt das System beispielsweise das Wort auf den Bildschirm, bei dem der EA eine falsche Entscheidung trifft. Wenn du gar nicht mehr weiter weißt, kannst du das System auffordern, den EA selbst zu entwerfen.

Das E-Learning-System steht auf der Seite

`http://www.ite.ethz.ch/kids/index`

frei zur Verfügung. Du musst nur **Exorciser** unter „Projektwoche“ anklicken. Außerdem findest du ihn auch unter „Informatik“ auf dem Bildungsserver EducETH.

Hinweis für die Lehrperson Der nächste Teil der Lektion 3.7 ist optional. Er eignet sich für gute Schüler, die schneller vorankommen und sich für formale mathematische Beweise interessieren.

Die Konstruktion eines EA M, der zwei andere endliche Automaten M_1 und M_2 simuliert, kann auch allgemein formal beschrieben werden. Der Sinn einer formalen Konstruktionsbeschreibung besteht nicht ausschließlich darin, die Anwendung der Mathematik

als formale Sprache zu üben. Wenn man eine Konstruktion exakt beschrieben hat, kann man sie durch ein Programm implementieren und es dann dem Rechner überlassen, automatisch endliche Automaten aus schon bestehenden Automaten zusammenzusetzen. Zusätzlich kann man, wenn man eine exakte Beschreibung eines Automaten hat, formal seine Korrektheit im Allgemeinen beweisen. Auf diese Weise vereinfacht der modulare Entwurf die komplette Verifikation. Es reicht aus, die Korrektheit der einfachen endlichen Automaten aus Basismodulen zu beweisen. Die Korrektheit der erzeugten komplexen EA resultiert dann direkt aus dem allgemeinen Korrektheitsbeweis der angewendeten Konstruktion.

Beschreiben wir also formal den EA M, der zwei gegebene Automaten

$$M_1 = (Q_1, \Sigma, \delta_1, q_{01}, F_1) \text{ und } M_2 = (Q_2, \Sigma, \delta_2, q_{02}, F_2)$$

simuliert. Beide arbeiten über dem gleichen Alphabet Σ. Ansonsten hat jeder EA M_i $(i = 1,2)$ eine eigene Zustandsmenge Q_i, einen eigenen Anfangszustand q_{0i}, eine eigene Transitionsfunktion δ_i und eine eigene Menge der akzeptierenden Zustände F_i. Um Missverständnisse zu vermeiden ist es sinnvoll, unterschiedliche Namen für die Zustände in M_1 und M_2 zu verwenden. Dazu wählt man die Namen so, dass $Q_1 \cap Q_2 \neq \emptyset$ gilt.

Wir konstruieren den EA M wie folgt:

$$M = (Q, \Sigma, \delta, q_0, F),$$

wobei

1. $Q = Q_1 \times Q_2 = \{(q,p) \mid q \in Q_1, p \in Q_2\}$, d. h. die Zustände von M sind Paare (2-dimensionale Vektoren), wobei das erste Element ein Zustand von M_1 und das zweite Element ein Zustand von M_2 ist.

2. Σ ist dasselbe Eingabealphabet, wie M_1 und M_2 es benutzen.

3. $q_0 = (q_{01}, q_{02})$, d.h. der Startzustand von M ist das Paar (Startzustand von M_1, Startzustand von M_2).

4. Für alle $q \in Q_1, p \in Q_2$ und alle $a \in \Sigma$

   $$\delta((q,p),a) = (\delta_1(q,a), \delta_2(p,a)),$$

   d.h. beim Lesen eines Buchstabens $a \in \Sigma$ ändert M das erste Element q seines derzeitigen Zustands (q,p), genauso wie es M_1 beim Lesen von a geändert hätte.

Das zweite Element p des Zustands (q,p) wird nach der Transitionsfunktion δ_2 von M_2 geändert.

5. Die Wahl von F hat mit der eigentlichen Simulation nichts zu tun. F bestimmt nur die Sprache, die akzeptiert werden soll. Wenn

   $$L(M) = L(M_1) \cup L(M_2)$$

   erwünscht ist, dann setzen wir

   $$F = F_1 \times Q_2 \cup Q_1 \times F_2.$$

   Auf diese Weise akzeptiert M genau dann, wenn mindestens einer der Automaten M_1 und M_2 das gegebene Wort akzeptiert. Wenn man

   $$L(M) = L(M_1) \cap L(M_2)$$

   fordert, dann setzen wir

   $$F = F_1 \times F_2.$$

   Somit akzeptiert M ein Wort x genau dann, wenn beide Automaten M_1 und M_2 das Wort x akzeptieren.

Aufgabe 6.4 Betrachte die Sprachen

$$L_1 = \{x \in \{0,1\}^* \mid |x|_0 \bmod 3 = 2\},$$
$$L_2 = \{x = y011z \mid y,z \in \{0,1\}^*\}.$$

Konstruiere endliche Automaten M_1 und M_2 mit $L(M_i) = L_i$ für $i = 1,2$. Gib für beide Automaten die formale Beschreibung der Transitionsfunktionen δ_1 und δ_2 in Tabellenform an. Wende die oben gegebene Konstruktion (Teil 4) an und konstruiere die Tabelle für die Transitionsfunktion δ von M.

Aufgabe 6.5 Wie muss man F in Teil 5 der Konstruktion wählen, wenn man

a) $L(M) = L(M_1) - L(M_2) = \{x \in \Sigma^* \mid x \in L(M_1) \text{ und } x \notin L(M_2)\}$,

b) $L(M) = L(M_2) - L(M_1)$,

c) $L(M) = \Sigma^* - (L(M_1) \cap L(M_2))$,

d) $L(M) = \Sigma^* - (L(M_1) \cup L(M_2))$

akzeptieren will?

Aufgabe 6.6 (Aufwendig, aber nicht schwer) Überlege dir eine formale Darstellung eines EA für den Rechner. Schreibe dann ein Programm, das für zwei gegebene formale Darstellungen der beiden endlichen Automaten M_1 und M_2 eine Beschreibung für einen EA M berechnet, so dass gilt:

1. $L(M) = L(M_1) \cap L(M_2)$,
2. $L(M) = L(M_1) \cup L(M_2)$.

Aufgabe 6.7 Seien M_1, M_2 und M_3 endliche Automaten. Gib eine formale Konstruktion für einen endlichen Automaten M an, der alle drei Automaten M_1, M_2 und M_3 gleichzeitig simuliert.

Jetzt wollen wir wissen, ob der konstruierte Automat M die Automaten M_1 und M_2 tatsächlich korrekt simuliert. Es bedeutet, die folgende Behauptung zu beweisen.

Satz 6.1 *Folgendes gilt für alle Wörter $x \in \Sigma^*$:*
Wenn M_1 nach dem Lesen von x aus q_{01} in einem Zustand v die Berechnung beendet (d. h. $(q_{01}, x) \vdash_{M_1}^* (v, \lambda)$) und M_2 nach dem Lesen von x in einem Zustand s endet (d. h. $(q_{02}, x) \vdash_{M_2}^* (s, \lambda)$), dann gilt

$$((q_{01}, q_{02}), x) \vdash_M^* ((v, s), \lambda),$$

d. h. M beendet die Berechnung auf x im Zustand (v, s).

Beweis: Wir führen diesen Beweis mittels Induktion bezüglich der Eingabelänge n.

1. **Induktionsanfang**
   $n = 0$, d. h. $x = \lambda$.
   Offensichtlich ist für $x = \lambda$ die Startkonfiguration auch gleichzeitig die Endkonfiguration. So verbleibt M_1 in q_{01}, M_2 in q_{02} und M in seinem Startzustand (q_{01}, q_{02}).

2. **Induktionsschritt**
   Jetzt setzen wir voraus, dass unsere Induktionsannahme für alle $y \in \Sigma^*$ mit $|y| \leq n$ gilt. Wir beweisen unsere Behauptung für alle $x \in \Sigma^{n+1}$.
   Sei x ein beliebiges Wort aus Σ^{n+1}. Wir können x als

   $$wa$$

für ein $a \in \Sigma$ und ein $w \in \Sigma^n$ schreiben.
Seien

$$(q_{01}, w) \vdash_{M_1}^* (q, \lambda) \text{ und } (q_{02}, w) \vdash_{M_2}^* (p, \lambda) \tag{6.1}$$

die Berechnungen von M_1 und M_2 auf w.
Weil $|w| = n$ ist, liefert die Induktionsannahme, dass

$$((q_{01}, q_{02}), w) \vdash_M^* ((q, p), \lambda) \tag{6.2}$$

die Berechnung von M auf w ist.
Betrachten wir jetzt die Berechnungen von M_1, M_2 und M auf $x = wa$. Die Berechnungen von M_1 und M_2 auf x kann man aus (6.1) folgendermaßen erweitern:

$$\begin{aligned} (q_{01}, wa) &\vdash_{M_1}^* (q, a) \quad \vdash_{M_1} \quad (\delta_1(q, a), \lambda), \\ (q_{02}, wa) &\vdash_{M_1}^* (p, a) \quad \vdash_{M_2} \quad (\delta_2(p, a), \lambda). \end{aligned}$$

Weil wir in der Konstruktion von M (Teil 4) δ durch

$$\delta((q, p), a) = (\delta_1(q, a), \delta_2(p, a))$$

definiert haben und (6.2) gilt, muss die Berechnung von M auf $x = wa$ wie folgt aussehen:

$$((q_{01}, q_{02}), wa) \vdash_M^* ((q, p), a) \vdash_M ((\delta_1(q, a), \delta_2(p, a)), \lambda).$$

Damit haben wir gezeigt, dass M die Berechnung auf x in dem Zustand

$$(\delta_1(q, a),\ \delta_2(p, a))$$

beendet. Dies ist das gewünschte Ergebnis, weil M_1 in $\delta_1(q, a)$ und M_2 in $\delta_2(p, a)$ endet. □

Aufgabe 6.8 Betrachte die Sprachen L_1 und L_2 aus Aufgabe 6.4 und die dazu konstruierten endlichen Automaten M_1, M_2 und M. Führe einen Induktionsbeweis der Behauptung durch, M simuliere die Automaten M_1 und M_2. Du sollst also den allgemeinen Beweis der Korrektheit der Konstruktion (Satz 6.1) an einem konkreten Beispiel durchführen.

Zusammenfassung

Die modulare Entwurfsmethode ist ein strukturiertes Vorgehen für den Entwurf von Systemen. Die Idee dabei ist es, zuerst einfache Systeme, sogenannte Module, zu bauen und anschließend diese auf ihre Korrektheit zu prüfen. Danach benutzt man die Module als Bausteine für komplexere Systeme. Dieser Vorgang kann beliebig oft wiederholt werden, bis man sehr komplexe Systeme erzeugt hat.

Für zwei endlichen Automaten kann man einen einzigen endlichen Automaten entwerfen, der die kompletten Informationen über beide besitzt. So kann man systematisch immer komplexere Automaten bauen, insbesondere wenn die zu akzeptierende Sprache durch mehrere Bedingungen beschrieben wird. Dann baut man zuerst für jede einzelne Bedingung einen endlichen Automaten und setzt diese danach zu einem komplexeren endlichen Automaten zusammen.

Kontrollfragen

1. Warum ist eine Überprüfung der Korrektheit von entworfenen technischen Systemen wichtig? Inwiefern vereinfacht ein systematischer Entwurf die Verifikation?

2. Wie unterschiedlich kann man den Begriff der Simulation auslegen? Um welchen geht es bei der Simulation zweier endlicher Automaten durch einen endlichen Automaten?

3. Was bedeutet „Modularität“ im Zusammenhang mit dem Entwurf von technischen Systemen?

4. Für welche Arten von Problemspezifikationen (Beschreibungen der zu akzeptierenden Sprache) würdest du die modulare Entwurfsmethode vorziehen? Warum?

Kontrollaufgaben

1. Entwirf jeweils einen endlichen Automaten mit der Entwurfsmethode der Zustandsklassen.

   a) $L_1 = \{111x1 \mid x \in \{0,1\}^*\}$

   b) $L_2 = \{xyz \mid y \in \{00,11\} \text{ und } x,z \in \{0,1\}^*\}$

   c) $L_3 = \{u \in \{0,1\}^* \mid |u|_1 = 1 \text{ oder } |u|_1 > 2\}$

d) $L_4 = \{u \in \{0,1\}^* \mid |u|_0 \bmod 3 = |u|_1 \bmod 3\}$

2. Verwende die modulare Entwurfsmethode, um jeweils einen endlichen Automaten für folgende Sprachen zu konstruieren. Die Sprachen L_1, L_2, L_3 und L_4 sind dieselben wie in der Kontrollaufgabe 1.

   a) $L_1 \cap L_2$

   b) $L_1 \cup L_2$

   c) $L_3 - L_2$

   d) $(\{0,1\}^* - L_1) \cup L_4$

   e) $(\{0,1\}^* - L_2) \cap L_3$

   f) $L_3 - (\{0,1\}^* - L_1)$

   g) $L_1 \cap L_3 \cap L_4$

   h) $(L_1 \cup L_3) \cup L_4$

   i) $L_1 \cap (L_3 \cup L_4)$

   j) $L_2 - (L_1 \cup L_4)$

Lösungen zu ausgesuchten Aufgaben

Aufgabe 6.1(a)
Wenn der endliche Automat in Abb. 6.4 die Sprache $L(M_1) - L(M_2)$ akzeptieren soll, müssen wir solche Zustände (r,s) als akzeptierende Zustände wählen, für die r ein akzeptierender Zustand von M_1 und s kein akzeptierender Zustand von M_2 ist. In unserem Beispiel (Abb. 6.2 und Abb. 6.3) sind es die Zustände

(q_0, p_1) und (q_0, p_2).

Aufgabe 6.1(c)
Wenn der endliche Automat in Abb. 6.4 die Sprache $\{0,1\}^* - (L(M_1) \cap L(M_2))$ akzeptieren soll, müssen seine akzeptierenden Zustände alle Paare (r,s) sein, für die nicht gleichzeitig gilt, dass beide r und s akzeptierende Zustände des jeweiligen Automaten M_1 und M_2 (Abb. 6.2 und Abb. 6.3) sind. Damit erhalten wir die folgende Liste von akzeptierenden Zuständen:

$(q_0, p_1), (q_0, p_2), (q_1, p_0), \quad (q_1, p_1), (q_1, p_2), (q_1, p_3)$.

Also sind es alle Zustände außer (q_0, p_0) und (q_0, p_3), die die Menge aller akzeptierenden Zustände für die Erkennung der Sprache $(L(M_1) \cap L(M_2)$ bilden.

Aufgabe 6.2(g)
Wir entwerfen den gesuchten endlichen Automaten, indem wir zuerst die endlichen Automaten für die Sprachen $L_1 = \{011y \mid y \in \{0,1\}^*\}$ und $L_2 = \{u100v \mid u, v \in \{0,1\}^*\}$ bilden. Diese Automaten sind in Abb. 6.5 und Abb. 6.6 dargestellt.

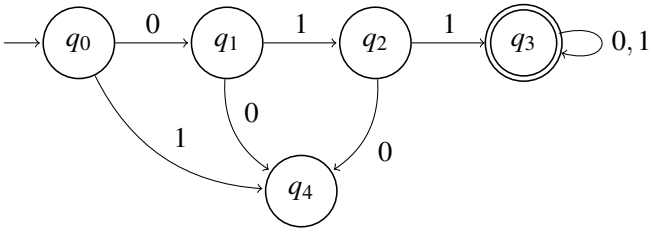


Abbildung 6.5

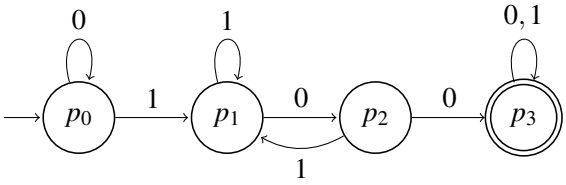


Abbildung 6.6

Unsere Aufgabe ist es, den endlichen Automaten für die Sprache $L_1 \cap L_2$ zu konstruieren. Der einzige akzeptierende Zustand ist (q_3, p_3). Die 0-Kanten des entsprechenden Automaten sind in Abb. 6.7 gezeichnet. Die 1-Kanten aus jedem Zustand zu bestimmen, überlassen wir gerne dir.

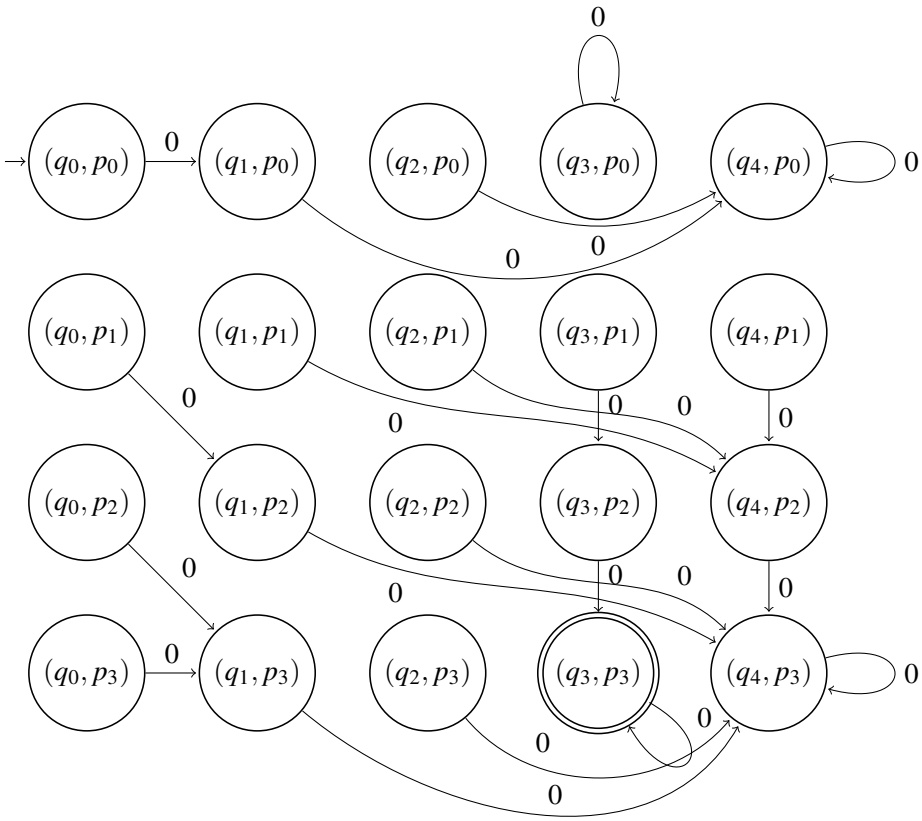


Abbildung 6.7

Lektion 7

Größe endlicher Automaten und Nichtexistenzbeweise

Hinweis für die Lehrperson Diese Lektion ist nur Klassen mit dem Schwerpunkt in der Mathematik oder einzelnen Schülerinnen und Schülern mit ausgeprägtem Interesse an der Mathematik gewidmet. Wenn man entsprechend langsam und vorsichtig vorgeht, ist der erste Teil bis zu Beispiel 7.2 auch in sprachlich orientierten Klassen noch zu bewältigen.

Es ist in der Wissenschaft leichter, die Existenz eines Objekts mit gewissen Eigenschaften[1] zu beweisen als seine Nichtexistenz[2]. Die Existenz belegt man, indem man das Objekt einfach konstruiert und somit im wahrsten Sinne des Wortes konstruktiv vorgeht. Die Nichtexistenz eines Objekts zu belegen erfordert den Nachweis, dass jeder Herstellungsversuch scheitern wird. Aber was bedeutet jeder Versuch? Das sind nicht nur die Wege, die uns eingefallen sind, sondern auch solche, die uns nie einfallen werden. Wie kann man wissen, dass auch unbekannte Konzepte nicht helfen werden, wenn man noch nichts von ihnen ahnt? Und dieser Gedanke ist gerade das, was den Beweis der Nichtexistenz erschwert. Die Beweise des Unmöglichen führen jedoch zu den wichtigsten Errungenschaften der Wissenschaft. Sie erforschen die Grenze zwischen dem, was möglich und was nicht möglich ist, und liefern so wichtige Naturgesetze.

Diese Lektion vermittelt, wie man die Nichtexistenz von kleinen endlichen Automaten oder sogar die Nichtexistenz von endlichen Automaten für eine gegebene Sprache formal begründet. Weil endliche Automaten eine Klasse von sehr einfachen[3] Algorithmen bil-

[1] oder die Möglichkeit eines Ereignisses
[2] Unmöglichkeit eines Ereignisses
[3] stark eingeschränkten

J. Hromkovič, *Lehrbuch Informatik*, DOI 10.1007/978-3-8348-9692-6_28,

den, sind die Nichtexistenzbeweise für endliche Automaten relativ leicht durchführbar. Trotzdem lernt man dabei zu verstehen, wie man solche Argumentationen führt.

Wir beginnen mit einer einfachen Aufgabe. Für eine gegebene Sprache L und eine gegebene positive ganze Zahl m soll gezeigt werden, dass jeder EA für L mindestens m Zustände benötigt. Wir demonstrieren es an einem Beispiel.

Beispiel 7.1 Betrachte die Sprache

$$L = \{x \in \{0,1\}^* \mid |x|_0 \bmod 4 = 3\}.$$

Unsere Aufgabe ist zu zeigen, dass jeder EA, der L akzeptiert, mindestens vier Zustände hat. Man könnte einen EA M mit $L(M) = L$ und vier Zuständen jetzt konstruieren und dann argumentieren, man könne es nicht anders machen. Dies ist aber gerade eine unzulässige Argumentation, weil wir nie wissen, ob es nicht auch andere Ideen zur Konstruktion eines EA für M gibt. Jetzt könnte man eine allgemeine Argumentation vermeiden, indem man alle möglichen endlichen Automaten mit höchstens drei Zuständen konstruiert und dann für jeden dieser EA A zeigt, dass $L(A) \neq L$ gilt. Diesen aufwendigen Weg wollen wir aber nicht beschreiten. Wir wollen stattdessen einige Eigenschaften von L und von endlichen Automaten finden, welche die Existenz eines EA mit weniger als vier Zuständen für L ausschließen. Dabei hilft uns die Behauptung aus Lektion 3:

> *„Wenn ein EA M die Berechnung auf zwei Wörtern x und y in dem gleichen Zustand beendet (wenn x und y in die gleichen Zustandsklassen gehören, d. h. wenn $(q_0,x) \vdash_M^* (p,\lambda)$ und $(q_0,y) \vdash_M^* (p,\lambda)$ für einen Zustand p), dann enden für alle Wörter z die Berechnungen von M auf xz und yz in dem gleichen Zustand und somit gilt*
>
> $$xz \in L(M) \Leftrightarrow yz \in L(M).\text{“}$$

Mit anderen Worten, wenn ein EA M nach dem Lesen von zwei unterschiedlichen Eingabepräfixen x und y denselben Zustand erreicht, kann er nicht mehr zwischen x und y unterscheiden und der Rest der Berechnung hängt nur von dem noch nicht gelesenen Suffix z ab.

Wenn man zeigen möchte, dass mindestens vier Zustände benötigt werden, um L zu akzeptieren, reicht es aus, vier Wörter x_1, x_2, x_3 und x_4 zu finden, von denen keine zwei zur selben Zustandsklasse gehören. Intuitiv sind wir der Meinung, jeder EA für L muss

die Anzahl der gelesenen Nullen modulo 4 zählen. Deswegen wählen wir

$$x_1 = \lambda$$
$$x_2 = 0$$
$$x_3 = 00$$
$$x_4 = 000.$$

Jetzt müssen wir für alle $i, j \in \{1,2,3,4\}, i < j$, ein Wort z_{ij} finden, so dass

genau eines der Wörter $x_i z_{ij}$ und $x_j z_{ij}$ in L liegt.

Fangen wir mit $x_1 = \lambda$ und $x_2 = 0$ an. Wenn wir

$$z_{12} = 00$$

nehmen, dann gilt

$$x_1 z_{12} = 00 \notin L \text{ und } x_2 z_{12} = 000 \in L.$$

Somit gehören x_1 und x_2 zu unterschiedlichen Zustandsklassen. Jetzt wählen wir

$$z_{13} = 000$$

Dann gilt

$$x_1 z_{13} = 000 \in L \text{ und } x_3 z_{13} = 00000 \notin L$$

und somit gehören x_1 und x_3 zu unterschiedlichen Zustandsklassen. Weiter wählen wir

$$z_{14} = \lambda$$

und somit gilt

$$x_1 z_{14} = \lambda \notin L \text{ und } x_4 z_{14} = 000 \in L.$$

Für x_2 und x_3 suchen wir

$$z_{23} = 00000$$

aus. Somit gilt

$$x_2 z_{23} = 000000 \notin L \text{ und } x_3 z_{23} = 0000000 \in L.$$

Für x_2 und x_4 können wir

$$z_{24} \text{ als } 00$$

wählen und somit gilt

$$x_2 z_{24} = 000 \in L \text{ und } x_4 z_{24} = 00000 \notin L.$$

Für das letzte Paar (x_3, x_4) können wir

$$z_{34} = \lambda$$

nehmen und erhalten

$$x_3 z_{34} = 00 \notin L \text{ und } x_4 z_{34} = 000 \in L.$$

Weil kein Paar der vier Wörter x_1, x_2, x_3 und x_4 zur gleichen Zustandsklasse gehören darf, muss man mindestens vier unterschiedliche Zustandsklassen haben. Somit hat jeder endliche Automat für L mindestens vier Zustände. □

Aufgabe 7.1 a) Wähle im Beispiel 7.1 für die ausgesuchten Wörter x_1, x_2, x_3 und x_4 andere Suffixwörter z_{ij} aus, so dass die Argumentation gültig bleibt.

b) Bestimme für jedes Paar (x_i, x_j), $i, j \in \{1,2,3,4\}$ und $i < j$ eine unendliche Menge Z_{ij} von Wörtern, so dass für alle $y \in Z_{ij}$ gilt

$$x_i y \in L \text{ und } x_j y \notin L.$$

Aufgabe 7.2 Wähle im Beispiel 7.1 andere Wörter für x_1, x_2, x_3 und x_4 aus und führe die Argumentation mit den Wörtern deiner Wahl durch.

Aufgabe 7.3 Bestimme, welche der folgenden Wortmengen x_1, x_2, x_3, x_4 zu einem erfolgreichen Beweis des Beispiels 7.1 führen und welche nicht. Begründe deine Behauptung.

a) $x_1 = 00$, $x_2 = \lambda$, $x_3 = 0$, $x_4 = 000$.
b) $x_1 = 0$, $x_2 = 00$, $x_3 = 000$, $x_4 = 0000$.
c) $x_1 = 000$, $x_2 = 0000$, $x_3 = 00000$, $x_4 = 000000$.
d) $x_1 = 0^{17}$, $x_2 = 0^{19}$, $x_3 = 000$, $x_4 = 0^{21}$.
e) $x_1 = 0^{20}$, $x_2 = 0^{7}$, $x_3 = 0^{10}$, $x_4 = 0^{13}$.

Aufgabe 7.4 Beweise, dass man für jede der folgenden Sprachen mindestens vier Zustände braucht, um sie zu akzeptieren.

a) $\{x101y \mid x,y \in \{0,1\}^*\}$,

b) $\{x \in \{0,1\}^* \mid |x|_0 \text{ ist gerade und } |x|_1 \text{ ist ungerade}\}$,

c) $\{x110 \mid x \in \{0,1\}^*\}$,

d) $\{111\}$,

e) $\{1x00 \mid x \in \{0,1\}^*\}$,

f) $\{0,1\}^2$,

g) $\{x \in \{a,b\}^* \mid |x| = 1 \text{ oder } |x| > 2\}$.

Aufgabe 7.5 Wie viele Zustände braucht man mindestens, um folgende Sprachen zu akzeptieren?

a) $\{x1y \mid x \in \{0,1\}^*, y \in \{0,1\}^2\}$,

b) $\{x0101y \mid x,y \in \{0,1\}^*\}$,

c) $\{x \in \{0,1,2\}^* \mid |x|_0 \bmod 3 = 2 \text{ und } x = u22v \text{ für } u,v \in \{0,1,2\}^*\}$,

d) $\{x \in \{0,1\}^* \mid x = uv, \text{ wobei } u \in \{0,1\}^* \text{ und } v \in \{00,01\}\}$.

Begründe deine Behauptung.

Aufgabe 7.6 ***** (Knobelaufgabe)

a) Sei

$$\begin{aligned} L_k &= \{x1y \mid x \in \{0,1\}^*, y \in \{0,1\}^{k-1}\} \\ &= \{w \in \{0,1\}^* \mid \text{ das } k\text{-te Bit, vom Ende aus gezählt, in } w \text{ ist } 1\}. \end{aligned}$$

Beweise für alle $k \in \mathbb{N} - \{0\}$, dass jeder endliche Automat, der L akzeptiert, mindestens $2^k - 1$ Zustände braucht.

b) Sei $M_n = \{wcw \mid w \in \{a,b\}^n\} \subseteq \{a,b,c\}^*$ eine endliche Sprache für ein $n \in \mathbb{N}$. Beweise für alle $n \in \mathbb{N} - \{0\}$, dass jeder EA für M_n mindestens 2^n Zustände haben muss.

Hinweis für die Klasse Der Rest dieser Lektion ist nicht obligatorisch. Es wird empfohlen, nur dann weiterzulesen, wenn man dieses Thema als spannende Herausforderung empfindet.

Wir haben gelernt, untere Schranken für die Größe von endlichen Automaten zu beweisen. Jetzt wollen wir diese Technik erweitern, um die Nichtexistenz von endlichen Automaten für konkrete Sprachen zu beweisen. Welche Sprachen sollten für endliche Automaten schwierig sein? Einfach diejenigen mit einer unbeschränkten[4] Anzahl der zu beobachtenden Merkmale.

Beispiel 7.2 Betrachte die Sprache

$$L = \{0^n 1^n \mid n \in \mathbb{N}\}.$$

Intuitiv sehen wir den Bedarf, beim Lesen von links nach rechts zuerst die Nullen zu zählen und ihre Anzahl zu speichern, denn so kann man später die Anzahl Nullen mit der Anzahl von Einsen zu vergleichen. Formal ausgedrückt betrachten wir die unendliche Folge von Wörtern

$$x_1 = 0, x_2 = 0^2, x_3 = 0^3, \ldots, x_i = 0^i, \ldots$$

Wir müssen zeigen, dass keine zwei dieser Wörter zur selben Zustandsklasse eines EA M gehören können, vorausgesetzt M würde L akzeptieren. Seien $x_i = 0^i$ und $x_j = 0^j$ zwei beliebige Wörter aus dieser Folge für $i \neq j$. Dann wählen wir

$$z_{ij} = 1^i.$$

Offensichtlich gilt

$$x_i z_{ij} = 0^i 1^i \in L \text{ und } x_j z_{ij} = 0^j 1^i \notin L.$$

Also muss man nach dem Lesen von 0^i und 0^j mit $i \neq j$ unbedingt in zwei unterschiedliche Zustände übergehen. Damit muss der Automat mindestens so viele Zustände haben, wie es Wörter in der unendlicher Folge $x_1, x_2, x_3, \ldots$ gibt. Kein endlicher Automat hat aber unendlich viele Zustände und deshalb kann L von keinem endlichen Automaten erkannt werden. □

Aufgabe 7.7 Was wäre passiert, wenn wir in Beispiel 7.2 $x_i = 0^{2i}$ gewählt hätten? Könnte man dann den Beweis der Nichtexistenz auch erfolgreich zu Ende bringen? Schlage einige andere Möglichkeiten für ein x_i vor. Gibt es auch eine Variante, bei der jedes x_i mindestens eine Eins enthält?

[4] Unbeschränkt bedeutet in diesem Zusammenhang, die Anzahl der zu speichernden Charakteristiken kann mit der Eingabelänge wachsen und ist somit unendlich.

Aufgabe 7.8 Beweise, dass die Sprache $L = \{0^n 1^{2n} \mid n \in \mathbb{N}\}$ keine reguläre Sprache ist.

Hinweis für die Lehrperson Die Schwierigkeit des vorgeführten Nichtexistenzbeweises liegt darin, gleichzeitig unendlich viele Wörter zu betrachten und davon unendlich viele Paare zu bilden. Dann wird allgemein ein z_{ij} zu einem beliebigen Paar (x_i, x_j) bestimmt. Indirekte Beweise ermöglichen uns, das Unendliche aus der Beweisführung zu verbannen. Damit ist aber die Bearbeitung der Lektion über Beweise aus dem Modul „Geschichte und Begriffsbildung“ eine notwendige Voraussetzung für das Erwerben der Methode, die wir in Beispiel 7.3 präsentieren.

Wenn man ungern mit unendlichen Wortfolgen argumentiert, kann man dies umgehen, indem man indirekte Beweise führt. Hier setzt man zuerst das Gegenteil von dem voraus, was man beweisen will, und zeigt dann, dass es zum Widerspruch führt. Wie man es konkret realisiert, zeigt das folgende Beispiel.

Beispiel 7.3 Wir sollen beweisen, dass

$$L' = \{a^n b^{2n} c^{3n} \mid n \in \mathbb{N}\}$$

nicht durch endliche Automaten akzeptiert werden kann. Man nutzt das Schema des indirekten Beweises (siehe Module „Logik“ und „Geschichte und Begriffsbildung“) und setzt das Gegenteil davon voraus. Das heißt also, wir nehmen die Existenz eines endlichen Automaten M mit $L(M) = L$ an. Weil M endlich ist, ist seine Zustandsmenge Q auch endlich. Sei $|Q| = k$ für eine positive ganze Zahl k. Jetzt zeigen wir, dass M die Sprache L nicht akzeptiert und man dadurch einen Widerspruch unserer Annahme erhält. Wir wählen die Wörter

$$x_1 = a^1 b^2, x_2 = a^2 b^4, \ldots, x_i = a^i b^{2i}, \ldots, x_{k+1} = a^{k+1} b^{2(k+1)}$$

und wollen belegen, dass keine zwei von diesen $k+1$ Wörtern zur selben Zustandsklasse gehören dürfen. Im Prinzip gibt es aber nur k Zustände für M. Also müssen $i, j \in \{1, \ldots, k\}, i < j$, existieren, so dass x_i und x_j zu einer gleichen Zustandsmenge von M gehören. Dann wählen wir

$$z_{ij} = c^{3i}.$$

Offensichtlich gilt

$$x_i z_{ij} = a^i b^{2i} c^{3i} \in L \text{ und } x_j z_{ij} = a^j b^{2j} c^{3i} \notin L.$$

Somit kann M die Sprache L nicht akzeptieren, weil M entweder beide Wörter $x_i z_{ij}$ und $x_j z_{ij}$ akzeptieren oder beide verwerfen muss. Wir haben also den gewünschten Widerspruch belegt. Damit gilt unsere Annahme nicht, was bedeutet, dass das Gegenteil der Annahme gilt. Das Gegenteil unserer Annahme ist:

„Es gibt keinen endlichen Automaten für L."

Damit ist der Beweis abgeschlossen. □

Aufgabe 7.9 Wähle andere $k+1$ Wörter $x_1, x_2, ..., x_{k+1}$ für Beispiel 7.3 und führe den indirekten Beweis mit den von dir ausgesuchten Wörtern.

Aufgabe 7.10 Führe einen indirekten Beweis, dass die Sprache $\{0^n 1^n \mid n \in \mathbb{N}\}$ aus Beispiel 7.3 von keinem endlichen Automaten akzeptiert werden kann.

Aufgabe 7.11 Führe einen direkten Nichtexistenzbeweis eines EA für die Sprache $\{a^n b^{2n} c^{3n} \mid n \in \mathbb{N}\}$ nach dem Beweismuster aus Beispiel 7.3.

Den indirekten Beweis aus Beispiel 7.3 kann man auch vermeiden. Man beweist, dass es für jedes $k \in \mathbb{N}$ keinen endlichen Automaten mit k Zuständen für L' gibt. Dann reicht es, für jedes k eine gute Auswahl von $k+1$ Worten zu treffen, damit keine zwei dieser Wörter in die gleiche Zustandsklasse gehören.

Aufgabe 7.12 Stelle für die folgenden Sprachen jeweils direkte und indirekte Beweise der Nichtexistenz von endlichen Automaten her.

a) $\{0^n 12^n \mid n \in \mathbb{N}\}$,

b) $\{w\#w \mid w \in \{0,1\}^*\}$ über $\{0,1,\#\}$,

c) $\{w\#v \mid w,v \in \{0,1\}^*, w \neq v\}$ über $\{0,1,\#\}$,

d) $\{0^n 1^{n^2} \mid n \in \mathbb{N}\}$ über $\{0,1\}$,

e) $\{w \in \{0,1\}^* \mid |w|_0 = 2|w|_1\}$,

f) $\{w \in \{a,b,c\}^* \mid |w|_a + |w|_b = |w|_c$.

Aufgabe 7.13 (Knobelaufgabe)
Bestimme, ob folgende Sprachen durch endliche Automaten akzeptiert werden können. Begründe

deine Antworten. Wenn die Antwort positiv ist, entwirf einen EA für diese Sprache. Wenn die Antwort negativ ist, liefere einen Beweis der Nichtexistenz eines EA für die gegebene Sprache.

a) $\{uuv \in \{0,1,\}^* \mid u,v \in \{0,1\}^*\}$,

b) $\{ww \in \{0,1\}^* \mid w \in \{0,1\}^*\}$,

c) $\{x \in \{a,b\}^* \mid (|x|_a + 2|x|_b) \bmod 4 = 3\}$,

d) $\{a^{n^2} \mid n \in \mathbb{N}\}$ über $\{a\}$,

e) $\{0^{2^n} \mid n \in \mathbb{N}\}$ über $\{0\}$,

f) $\{u1v \mid u,v \in \{0,1\}^*, (|uv|_0 - 2|uv|_1) \bmod 3 = 1\}$,

g) $\{x1y \mid x,y \in \{0,1\}^*, |x| = |y|\}$.

Zusammenfassung

Um die Nichtexistenz eines kleinen oder sogar eines beliebigen endlichen Automaten zu beweisen, reicht es nicht zu sagen, dass keiner der Ansätze funktioniert. Es muss etwas Allgemeines über die betrachtete Sprache bewiesen werden, um die Existenz eines kleinen oder überhaupt eines endlichen Automaten für die Akzeptanz der Sprache auszuschließen.

In dieser Lektion haben wir eine einfache und dabei wirksame Methode für Nichtexistenzbeweise entwickelt. Die Methode basiert auf der Zuordnung der Wortklassen zu jedem Zustand und auf der Fähigkeit, eine große (potentiell unendliche) Gruppe von Wörtern zu finden, von denen keine zwei zur gleichen Zustandsklasse eines Automaten gehören dürfen. Die Suche nach dieser Gruppe von Wörtern basiert auf einer Schätzung, welche Information über das bisher gelesene Wort gespeichert werden muss.

Es bestehet die Möglichkeit, die Beweise sowohl direkt als auch indirekt zu führen. Bei direkten Beweisen gehen wir konstruktiv vor, indem wir eine Gruppe unterschiedlicher Wörter bilden, von denen keine zwei in die gleiche Zustandsklassen gehören dürfen. Beim indirekten Beweisen setzen wir voraus, dass es für die gegebene Sprache einen endlichen Automaten mit einer gewissen Größe gibt. Danach konstruieren wir eine Gruppe von Wörtern, die größer ist als die Anzahl der Zustände des endlichen Automaten und in

der kein Wortpaar zur gleichen Zustandsklasse gehören darf. Damit erhalten wir einen Widerspruch zu der Annahme, dass der Automat die Sprache akzeptiert.

Kontrollfragen

a) Warum sind Nichtexistenzbeweise oft schwieriger als andere Beweise?

b) Was hat die Entwurfsmethode der Zustandsklassen mit den Nichtexistenzbeweisen gemeinsam?

c) Wie kann man mittels indirekter Beweismethode zeigen, dass für gewisse Sprachen keine endlichen Automaten existieren?

Kontrollaufgaben

1. Beweise für die gegebene Zahl k und die gegebene Sprache L, dass jeder endliche Automat M mit $L(M) = L$ mindestens k Zustände haben muss.

   a) $k = 7, L = \{x110110y \mid x, y \in \{0,1\}^*\}$

   b) $k = 4, L = \{x0y \mid x \in \{0,1\}^*, y \in \{0,1\}\}$

   c) $k = 6, L = \{x \in \{0,1\}^* \mid |x|_0$ ist gerade und $|x|_1 \geq 2\}$

2. Bestimme die Mindestgröße endlicher Automaten zum Erkennen folgender Sprachen:

   a) $\{1x11y \mid x, y \in \{0,1\}^*\}$,

   b) $\{x \in \{0,1\}^* \mid |x|_0 \in \{0,2\}$ und $|x|_1 \bmod 3 = 2\}$,

   c) $\{x \in \{0,1\}^* \mid |x| \leq 3\}$,

   d) $\{x00y \mid x \in \{0,1\}^*, y \in \{00,01,10,11\}\}$.

3. Beweise, dass folgende Sprachen keine regulären Sprachen sind, d.h. dass sie von keinem endlichen Automaten akzeptiert werden können.

   a)* $\{0^{n!} \mid n \in \mathbb{N}\}$,

b)* $\{1^n 20^{n^3} \mid n \in \mathbb{N}\}$,

c) $\{0^{2n}1^n0^{2n} \mid n \in \mathbb{N}\}$,

d) $\{x2y \mid |x|_1 = |y|_0, x, y \in \{0,1\}^*\}$,

e) $\{w\#w\#w \mid w \in \{0,1\}^*\}$,

f) $\{x\#y \mid x, y \in \{0,1\}^*, x \text{ ist ein Präfix von } y\}$,

g) $\{uv \mid u, v \in \{0,1\}^*, u \text{ ist ein Präfix von } v\}$.

Lösungen zu ausgesuchten Aufgaben

Aufgabe 7.2
Es stehen unendlich viele passende Möglichkeiten zur Wahl der Wörter x_1, x_2, x_3 und x_4. Wenn wir den endlichen Automaten in Abb. 7.1 betrachten, funktioniert unsere Methode für eine beliebige Wahl von x_i aus $Klasse[q_{i-1}]$ für $i = 1,2,3,4$.

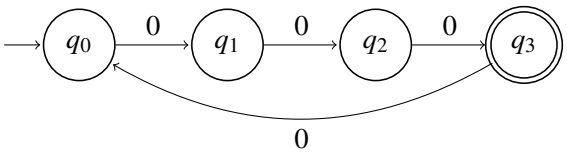


Abbildung 7.1

Wählen wir zum Beispiel

$x_1 = 0000$ $\qquad$ $x_2 = 00000$
$x_3 = 00$ $\qquad$ $x_4 = 000.$

Um zu zeigen, dass kein Paar (x_i, x_j) die Eigenschaft hat, dass beide Wörter x_i und x_j zu der gleichen Zustandsklasse gehören können, kann man die entsprechenden z_{ij} wie folgt wählen:

$z_{12} = 000$ $\qquad$ $z_{13} = 0$ $\qquad$ $z_{14} = \lambda$
$z_{23} = 0$ $\qquad$ $z_{24} = \lambda$ $\qquad$ $z_{34} = \lambda.$

Aufgabe 7.4 (a)
Um Wörter mit dem Teilwort 101 zu akzeptieren, muss sich ein endlicher Automat merken, ein

wie langes passendes Präfix er gerade gefunden hat. Das Präfix kann die Länge $0, 1, 2$ oder 3 haben. Damit ist die einfachste Wahl der Wörter, die nicht zur gleichen Zustandsklasse gehören dürfen, die folgende:

$$x_1 = \lambda \qquad x_2 = 1$$
$$x_3 = 10 \qquad x_4 = 101.$$

Wenn wir z_{12} gleich 01 setzen, erhalten wir

$$x_1 z_{12} = 01 \notin L \text{ und } x_2 z_{12} = 101 \in L.$$

Für x_2 und x_3 wählen wir $z_{23} = 1$ und erhalten

$$x_2 z_{23} = 11 \notin L \text{ und } x_3 z_{23} = 101 \in L.$$

Weiter kann man erfolgreich

$$z_{13} = 1 \qquad z_{14} = \lambda$$
$$z_{24} = \lambda \qquad z_{34} = \lambda$$

setzen.

Aufgabe 7.6 (b)
Um $M_n = \{wcw \mid w \in \{a,b\}^n\}$ zu akzeptieren, braucht man mindestens 2^n viele Zustände. Der Grund dafür ist, dass man jedes Präfix $w \in \{a,b\}^n$ vollständig abspeichern muss, um es später mit dem Wort hinter c vergleichen zu können. Es gibt 2^n viele Wörter über $\{a,b\}$ der Länge n. Wir müssen jetzt beweisen, dass für jedes Paar (u,v) mit $u, v \in \{a,b\}^n, u \neq v$, ein Wort z existiert, so dass $uz \in L$ und $vz \notin L$.

Die Wahl ist einfach. Wir wählen $z = cu$. Somit erhalten wir

$$uz = ucu \in L \text{ und } vz = vcu \notin L \text{ (weil } v \neq u).$$

Die Zahl 2^n ist nur eine untere Schranke für die Anzahl der benötigten Zustände. Tatsächlich kann man sogar beweisen, dass mindestens $2^{n+1} - 1$ Zustände notwendig sind. Schaffst du es selbstständig? Zeichne einen endlichen Automaten für M_3!

Aufgabe 7.7
Mit $x_i = 0^{2i}$ trifft man auch eine gute Wahl. Für $x_i = 0^{2i}$ und $x_j = 0^{2j}$ mit $i \neq j$ reicht es, $z_{ij} = 1^{2i}$ zu wählen. Somit erhält man

$$x_i z_{ij} = 0^{2i} 1^{2i} \in L \text{ und } x_j z_{ij} = 0^{2j} 1^{2i} \notin L.$$

Die Beweismethode wird funktionieren, auch wenn man Wörter $x_i = 0^i 1$ für $i = 1, 2, 3, \ldots$ wählt. Für diese Wahl reicht es, z_{ij} als 1^{i-1} zu setzen.

Aufgabe 7.12
Um die Nichtregularität der Sprache

$$L = \{w \in \{0,1\}^* \mid |w|_0 = 2|w|_1\}$$

zu zeigen, kann man ähnlich wie in Beispiel 7.2 vorgehen. Wir beweisen zuerst, dass alle Wörter der Form $0^{2n}1^n$ in L gehören und dass kein Wort der Form $0^{2i}1^j$ für $i \neq j$ in L liegt. Somit kann man $x_i = 0^{2i}$ für $i = 1,2,\ldots$ wählen. Für beliebige x_i und x_j mit $i \neq j$ wählen wir dann $z_{ij} = 1^i$ und erhalten

$$x_i z_{ij} = 0^{2i}1^i \in L \text{ und } x_j z_{ij} = 0^{2i}1^j \notin L.$$

Aufgabe 7.13 (d)
Die Nichtregularität der Sprache

$$L = \{a^{n^2} \mid n \in \mathbb{N}\}$$

kann man dank der wachsenden Differenz zwischen zwei nachfolgenden Quadratzahlen beweisen. Wir berechnen, dass

$$(n+1)^2 = n^2 + 2 \cdot n + 1 \text{ und somit} \qquad (n+1)^2 - n^2 = 2 \cdot n + 1$$

gilt. Wir wählen jetzt

$$x_i = a^{i^2+1} \notin L$$

für $i = 1,2,\ldots$. Für x_i und x_j mit $i < j$ wählen wir $z_{ij} = a^{2i}$. Somit erhalten wir

$$x_i z_{ij} = a^{i^2+1}a^{2i} = a^{(i+1)^2} \in L$$

und

$$x_j z_{ij} = a^{j^2+1}a^{2i} = a^{j^2+2i+1} \notin L.$$

Das Letzte gilt, weil $j^2 < j^2 + 2 \cdot i + 1 < j^2 + 2 \cdot j + 1 = (j+1)^2$. Somit liegt $j^2 + 2i + 1$ zwischen zwei nachfolgenden Quadratzahlen und ist damit keine Quadratzahl.

Lektion 8

Zusammenfassung des Moduls

In diesem Modul haben wir das einfachste Berechnungsmodell – die endlichen Automaten – kennen gelernt. Endliche Automaten entsprechen einfachen Programmen, die keine Variablen und damit keinen zusätzlichen Speicher (außer dem Speicher für das Programm selbst) verwenden. Dabei haben wir erfahren, was die Grundbegriffe Konfiguration, Berechnungsschritt, Berechnung und Simulation bedeuten und wie sie formal beschrieben werden können.

Die wichtigsten Lerninhalte waren die folgenden Methoden und Beweistechniken:

1. Die Entwurfsmethode für endliche Automaten, die auf der Bestimmung der Bedeutung einzelner Zustände durch Zustandsklassen basiert.

2. Die modulare Entwurfsmethode, die komplexe Automaten aus einfacheren Automaten zusammenbaut.

3. Die Korrektheit der entworfenen Automaten zu beweisen.

4. Endliche Automaten zur Steuerung von realen Systemen wie Verkaufsautomaten, Fahrstühlen und Kreuzungen zu entwerfen.

5. Techniken zum Beweisen, dass der entworfene Automat die minimal mögliche Anzahl von Zuständen hat, d.h. Beweistechniken für die Mindestgröße von endlichen Automaten zu gegebenen, konkreten Zwecken.

6. Direkte und indirekte Techniken zum Beweis der Nichtexistenz endlicher Automaten für konkrete Sprachen.

J. Hromkovič, *Lehrbuch Informatik*, DOI 10.1007/978-3-8348-9692-6_29,

Neben dem Erlernen von Grundlagen der Automatentheorie haben wir zur Festigung unserer eigenen mathematischen Denkweise beigetragen. Wir haben geübt, direkte und indirekte Beweise sowie Induktionsbeweise zu führen. Während des gesamten Moduls wurde die Mathematik als Sprache der Wissenschaft benutzt, um Objekte unseres Interesses prägnant beschreiben und eindeutige Aussagen über sie treffen zu können.

Sachverzeichnis

J. Hromkovič, *Lehrbuch Informatik*, DOI 10.1007/978-3-8348-9692-6,

W

Z