Juraj Hromkovič

Einführung in die Programmierung mit LOGO

Lehrbuch für Unterricht und Selbststudium

► Mit Online-Service

STUDIUM

VIEWEG+
TEUBNER

Juraj Hromkovič

Einführung in die Programmierung mit LOGO

Juraj Hromkovič

Einführung in die Programmierung mit LOGO

Lehrbuch für Unterricht und Selbststudium

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 Akademischen Gesellschaft und der Gesellschaft der Gelehrten der Slowakischen Akademie. Seit Januar 2004 Professor für Informatik an der ETH Zürich.

1. Auflage 2010

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: Ten Brink, Meppel
Gedruckt auf säurefreiem und chlorfrei gebleichtem Papier.
Printed in the Netherlands

ISBN 978-3-8348-1004-5

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 Wissenschaftsdiziplin 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, Björn Steffen und Joana Welti 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 Juli 2009

Juraj Hromkovič

Inhaltsverzeichnis

1 Programme als Folge von Befehlen **17**

2 Einfache Schleifen mit dem Befehl repeat **37**

3 Programme benennen und aufrufen **57**

4 Zeichnen von Kreisen und regelmäßigen Vielecken **73**

5 Programme mit Parametern **85**

6 Übergabe von Parameterwerten an Unterprogramme **101**

7 Optimierung der Programmlänge und der Berechnungskomplexität **121**

8 Das Konzept von Variablen und der Befehl `make` **137**

9 Lokale und globale Variablen **159**

10 Verzweigungen von Programmen und `while`-Schleifen **175**

11 Integrierter LOGO- und Mathematikunterricht: Geometrie und Gleichungen **197**

12 Rekursion **213**

13 Integrierter LOGO- und Mathematikunterricht: Trigonometrie **245**

14 Integrierter LOGO- und Mathematikunterricht: Vektorgeometrie **257**

Sachverzeichnis **269**

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.

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 Pro-

grammierung 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

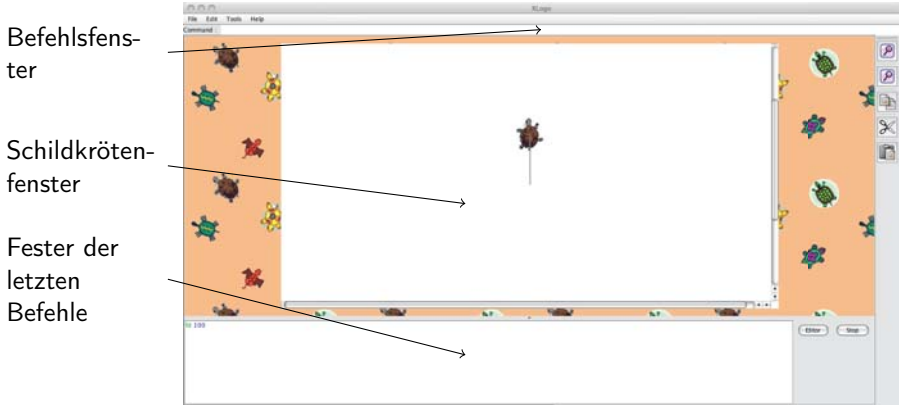


Abbildung 0.1 Der Bildschirm von XLOGO mit dem Schildkrötenfenster und dem Befehlsfenster.

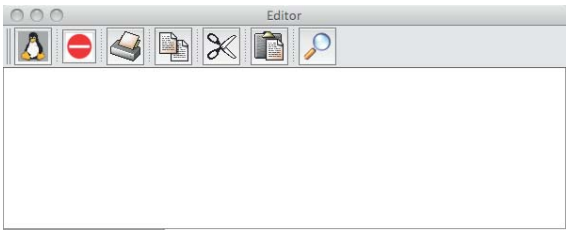


Abbildung 0.2 Der Editor von XLOGO

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 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 geschriebene 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

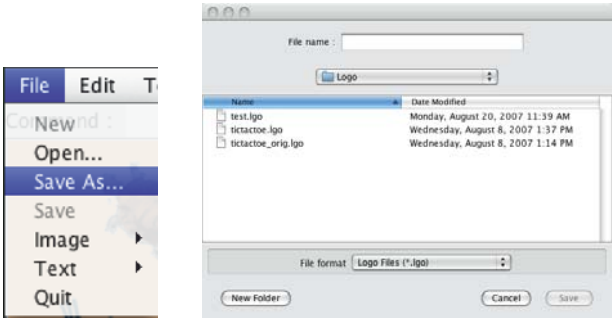


Abbildung 0.3 Links ist das Menü File abgebildet. Wenn man auf „Save As . . . “ klickt, erhält man das Projektfenster rechts.

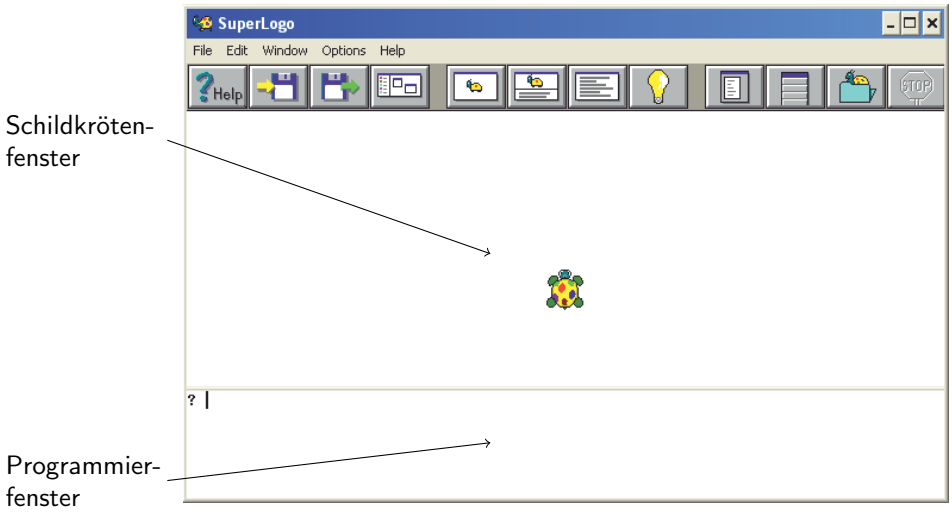


Abbildung 0.4 Die Programmierumgebung SUPERLOGO mit dem Programmierfenster und dem Schildkrötenfenster.

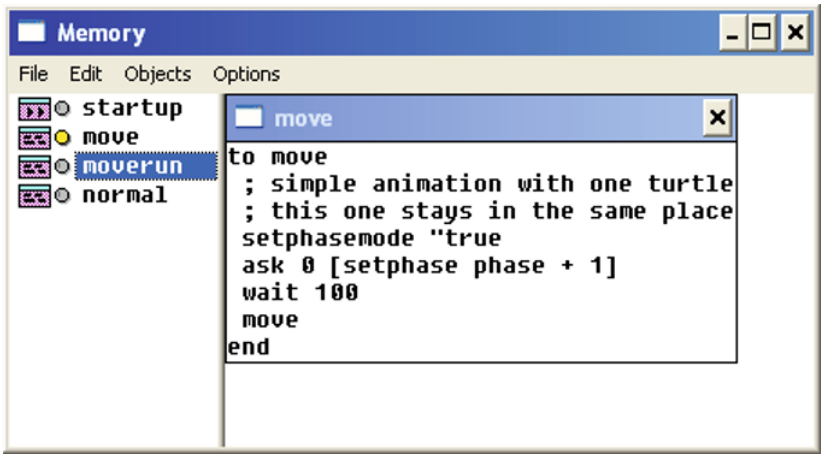


Abbildung 0.5

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.

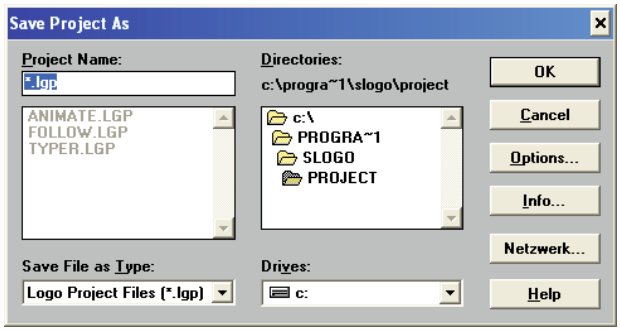


Abbildung 0.6

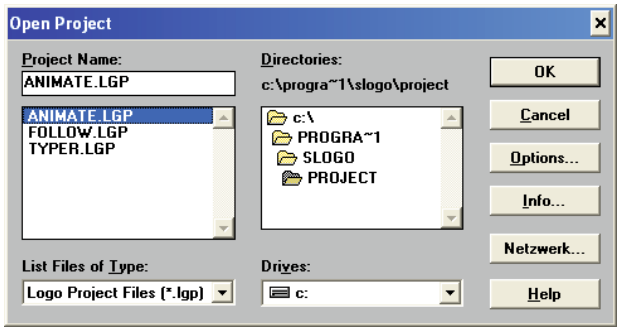


Abbildung 0.7

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 anderes 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, 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.

Hinweis für die Lehrperson Die erste Lektion beschreibt eine langsame Vorgehensweise, mit der man die Zielsetzung ab dem dritten Schuljahr erfolgreich erreichen kann. Für Klassen ab dem siebten Schuljahr empfehlen wir ein schnelleres Vorgehen, in dem man die Bearbeitung der meisten Aufgaben überspringen darf. Abhängig von der Vorgehensweise kann man die erste Lektion in 20 Minuten bis zu 2 Unterrichtsstunden bewältigen. Wenn man schneller vorgeht, darf man nicht vergessen, dass es nicht nur um das Erlernen der Bedeutung von ein paar Befehlen geht,

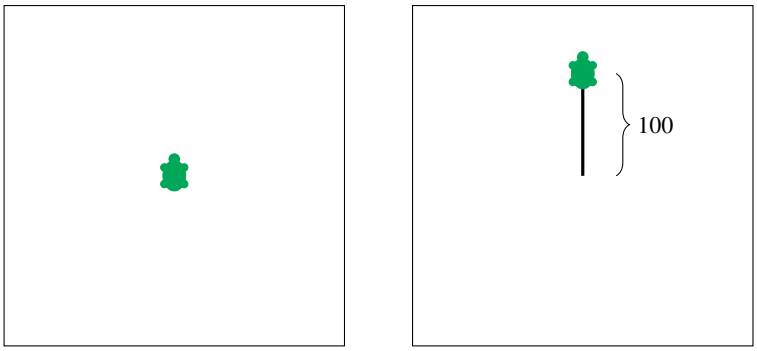


Abbildung 1.1 Ausführung des Befehls `forward 100`

sondern auch um die Begriffsbildung und korrekte Verwendung der Fachsprache. Hier stehen die Grundbegriffe Programm, Programmiersprache, Befehlswort und Parameter im Vordergrund. Wie auch die nachfolgenden Lektionen ist Lektion 1 zum Selbststudium geeignet. Für Klassen der 3. und 6. Stufe beinhaltet sie aber zu viel Text. Für diese Klassen sollten die ersten sechs Lektionen eher als Unterrichtsunterlage der Lehrperson dienen. Geeignete Unterlagen für die jüngeren Schülerinnen und Schüler stehen auf `www.abz.inf.ethz.ch` frei zur Verfügung.

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` 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 Parameter 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
Befehlswort  Parameter
```

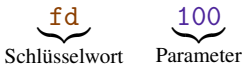
und rechts die Situation nach der Ausführung erkennbar.

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



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

oder die gleichwertige, kürzere Beschreibung

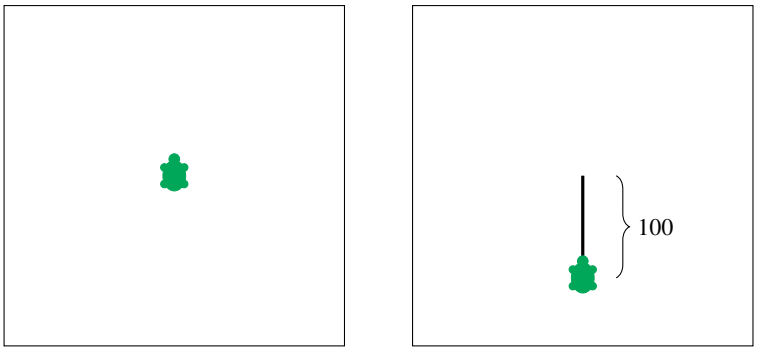


Abbildung 1.2 Ausführung des Befehls `backward 100`. Links die Lage vor der Ausführung, rechts die Lage danach.

```
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.

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
```

Hinweis für die Lehrperson Üblicherweise sind die Aufgaben im Text zur sofortigen Bearbeitung gedacht. Sie festigen das Gelernte und lehren damit umzugehen. Wenn das Neue für die unterrichtete Altersgruppe zu einfach ist, kann man einige davon auslassen. Dies gilt insbesondere für die ersten sechs Lektionen. Danach wächst der Schwierigkeitsgrad genug stark, um die Empfehlung auszusprechen, alle Aufgaben im Text konsequent zu lösen.

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. Pro Zeile sollte also nur ein Befehl geschrieben werden. 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. Zu begreifen, dass links und rechts aus der Sicht der Schildkröte und nicht unbedingt aus der Sicht des Programmierers zu betrachten ist, stellt üblicherweise eine kleine Hürde dar.

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

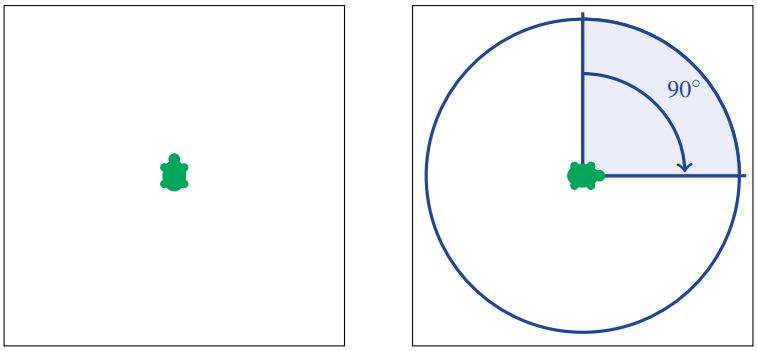


Abbildung 1.3 Der Befehl `right 90` oder `rt 90` fordert die Schildkröte auf, sich um 90° nach rechts zu drehen (90° bedeutet einen Viertelkreis).

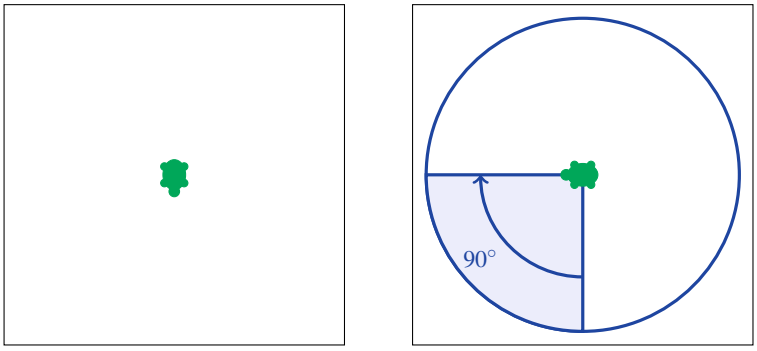


Abbildung 1.4 Der Befehl `rt 90` fordert die Schildkröte auf, sich um 90° (einen Viertelkreis) nach rechts zu drehen.

Ausführung dargestellt. Die Bilder in Abb. 1.3 und in Abb. 1.4 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 der nächsten Seite zeigen, dass eine Drehung um 360° weder Position noch Richtung der Schildkröte verändert.

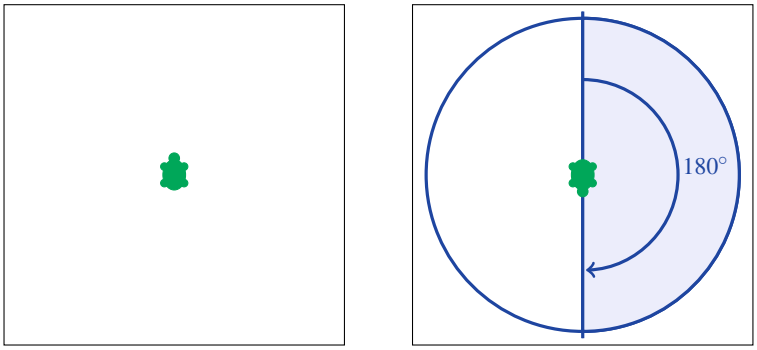


Abbildung 1.5 Der Befehl `rt 180` fordert die Schildkröte auf, sich um 180° nach rechts zu drehen (180° entspricht einem Halbkreis und damit der Drehung in die Gegenrichtung).

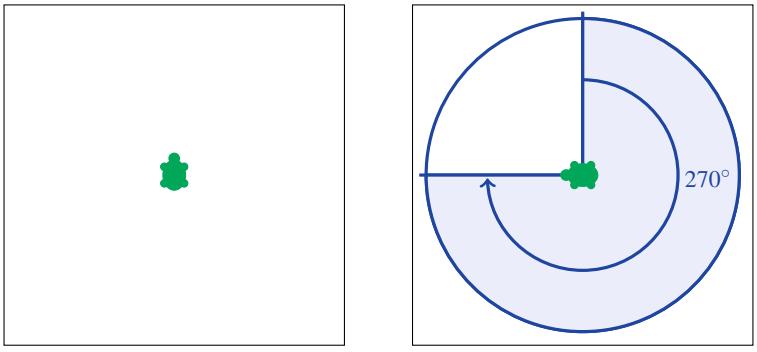


Abbildung 1.6 Der Befehl `rt 270` fordert die Schildkröte auf, sich um 270° nach rechts zu drehen.

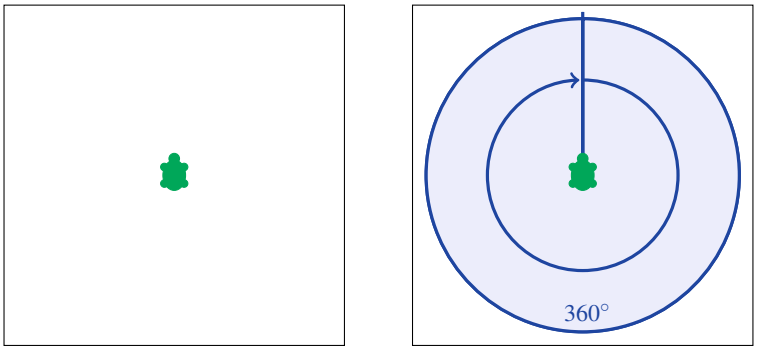


Abbildung 1.7 Der Befehl `rt 360` fordert die Schildkröte auf, sich um 360° nach rechts zu drehen. Die Schildkröte dreht sich einmal im Kreis und schaut in die gleiche Richtung wie vorher, es hat sich also nichts geändert.

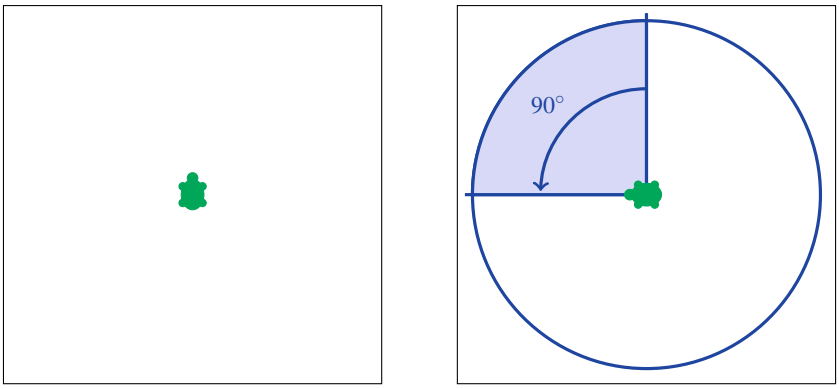


Abbildung 1.8 `left 90` oder `lt 90` dreht die Schildkröte nach links um 90°, also um einen Viertelkreis.

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`, als eine kürzere Form.

Die Wirkung dieses Befehls (s. Abb. 1.8) 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.

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.
```

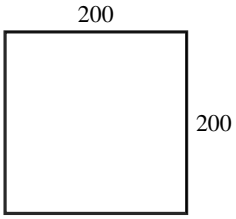
Zwischen zwei Befehlen in einer Zeile muss immer ein Leerzeichen stehen. Den Text

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

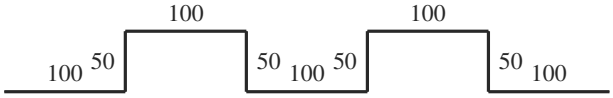
wird der Rechner nicht verstehen.

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.

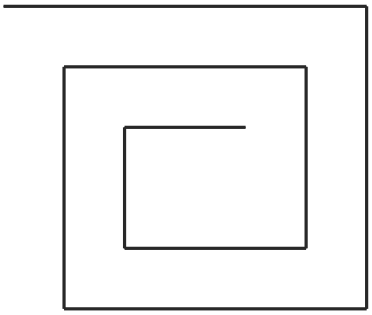
a)



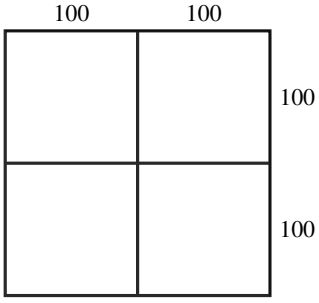
b)



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. Probiere 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?

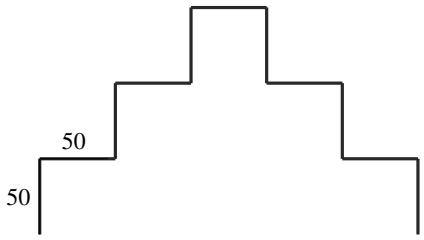
Kontrollaufgaben

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
```

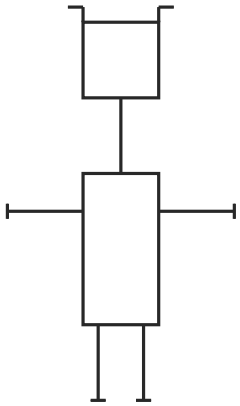
Überprüfe auf dem Rechner, ob das Programm wirklich dasselbe macht, wie das hier gegebene Programm.

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.

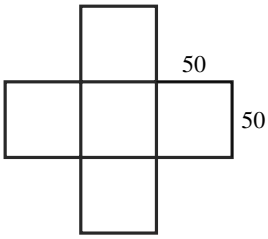


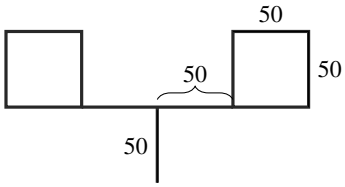
Abbildung 1.10

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.

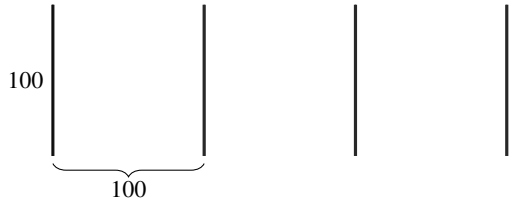


Am Anfang steht die Schildkröte unten in der Mitte. Sie hat auf folgende Weise angefangen:

```
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 Seite 31 und lösche danach durch Ausradieren alle vertikalen Linien, damit nur das Bild aus Abb. 1.11 bleibt.

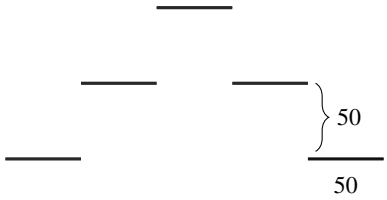


Abbildung 1.11

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

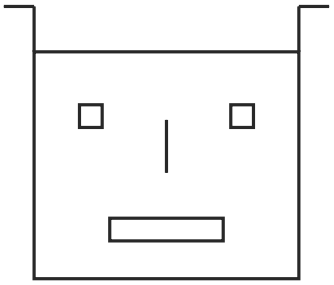


Abbildung 1.12

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:

```
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 als Parameter	[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.

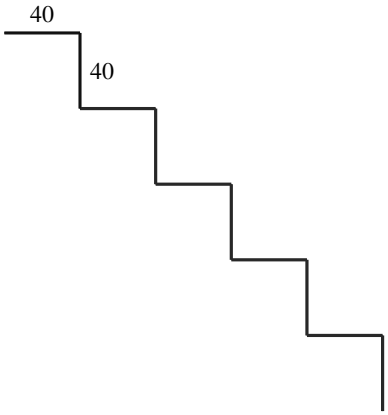


Abbildung 2.1

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.

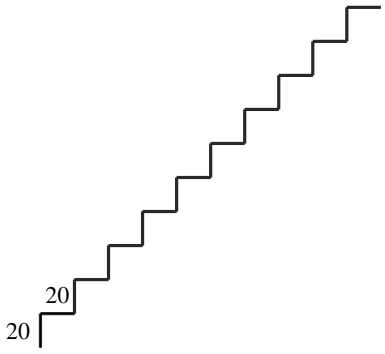


Abbildung 2.2

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.

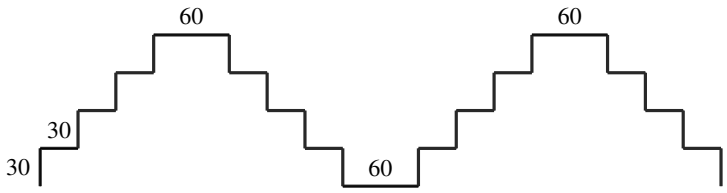


Abbildung 2.3

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 zehn mal hintereinander das gleiche Bild auf der gleichen Stelle zeichnet. Wenn man aber, nach jeder Zeichnung eines 100×100 Quadrates, die Schildkröte ein wenig dreht,

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

erhält man ein interessantes Bild. Probiere 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:

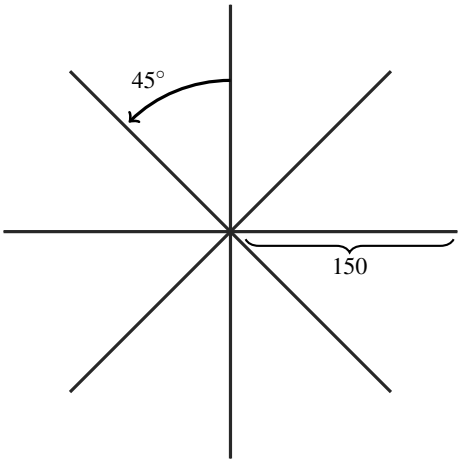


Abbildung 2.4

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 41) 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 41 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.

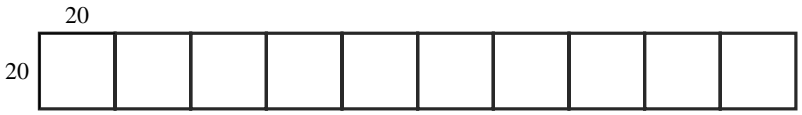


Abbildung 2.6

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

Erste Lösung: Wir können diese Aufgabe als eine Aufforderung verstehen, zehn mal 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.

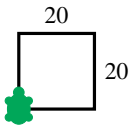


Abbildung 2.7

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

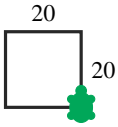


Abbildung 2.8

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 zehn mal 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    ]
```

ein Quadrat 20×20 zeichnen

Bewegung zur neuen Startposition

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

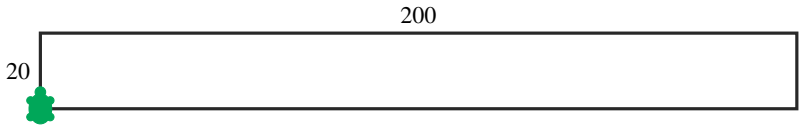


Abbildung 2.9

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.

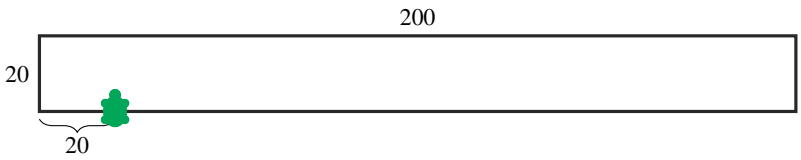


Abbildung 2.10

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

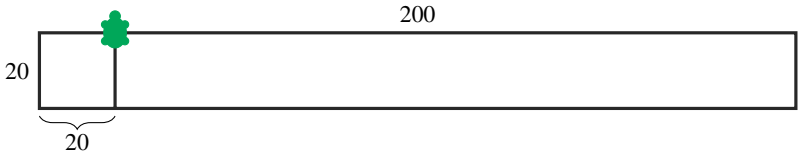


Abbildung 2.11

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.

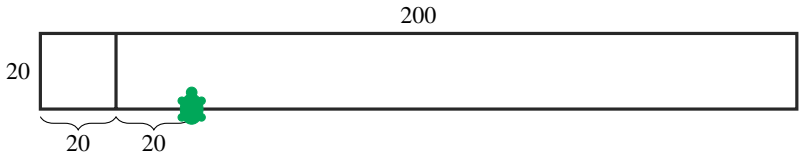


Abbildung 2.12

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 auf der vorherigen Seite und Abb. 2.12 noch acht mal 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 44) 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.

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. ]
```

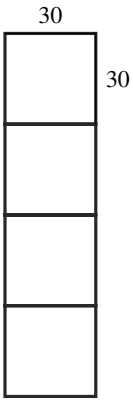


Abbildung 2.13

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), vorausgesetzt, wir zeichnen das Bild von unten nach oben.

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 44 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

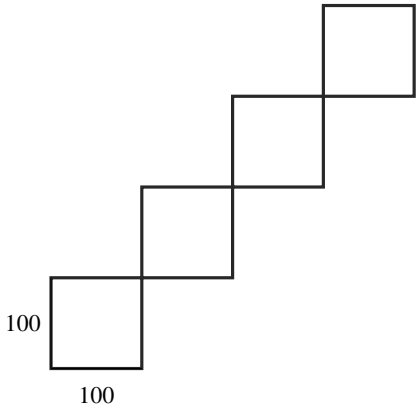


Abbildung 2.15

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. Der Grund dafür ist, dass man bei der Verwendung von Unterprogrammen nicht über das Verschachteln von Schleifen nachdenken muss.

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.

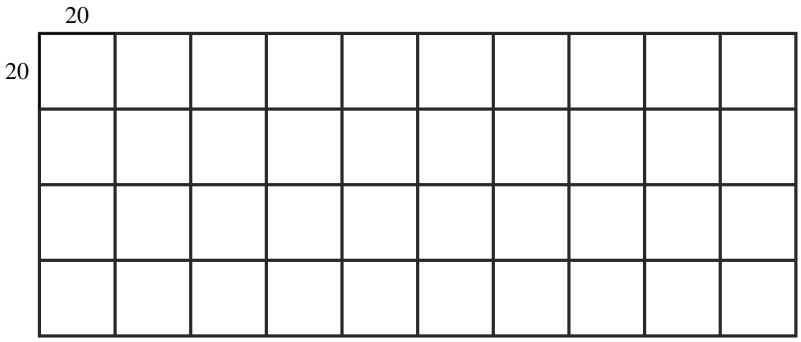


Abbildung 2.16

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 Stiftmodus, 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. Radiergummimodus, 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.

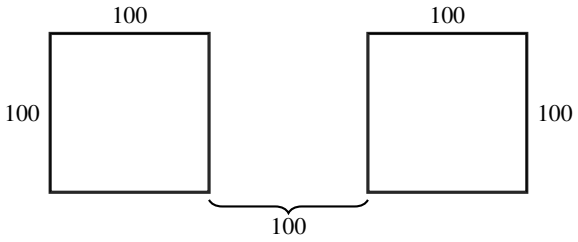


Abbildung 2.17

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

Hinweis für die Lehrperson Die Schildkröte kann farbig mit beliebigen Farben zeichnen. Wie dies gemacht wird, ist in Lektion 4 (Seite 77, Tabelle 4.1) erklärt. Um die Motivation besonders bei den jüngeren Schülerinnen und Schülern zu erhöhen, spricht nichts dagegen, die Befehle für den Farbwechsel früher einzuführen.

Aufgabe 2.14 Zeichne das Bild aus Abb. 2.18.

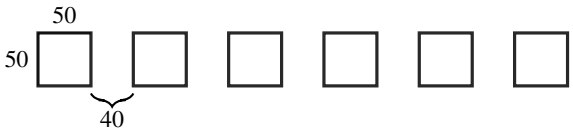


Abbildung 2.18

Aufgabe 2.15 Zeichne das folgende Bild.

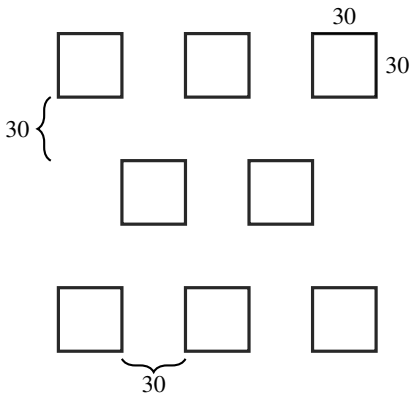


Abbildung 2.19

Aufgabe 2.16 Zeichne das Bild aus Abb. 2.20.

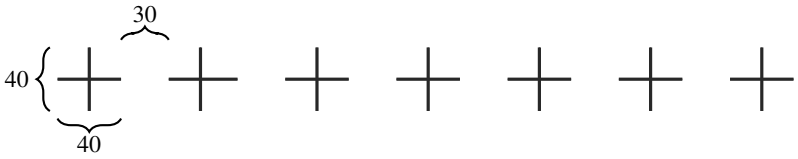


Abbildung 2.20

Aufgabe 2.17 Zeichne das Bild aus Abb. 2.21.

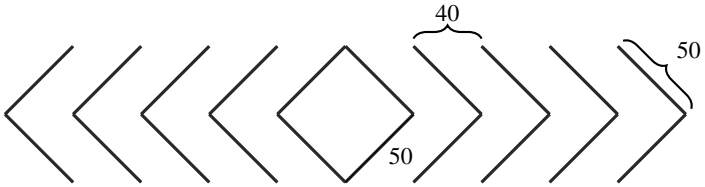


Abbildung 2.21

Kontrollfragen

1. Erkläre mit wenigen Worten, wozu der Befehl `repeat` gut ist. Wie sieht seine Struktur aus? (Welche Teile hat er und wie nennen wir sie?)

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. Welche Modi außer dem Wandermodus können wir schon?

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

9. 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. 2.22 auf der nächsten Seite.

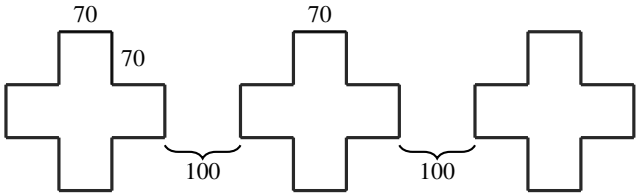


Abbildung 2.22

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 2.22, das eine Schleife in einer anderen Schleife enthält.

5. Zeichne sieben Kreuze nebeneinander, wie die drei aus Abb. 2.22, 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. 2.23 zeichnet.

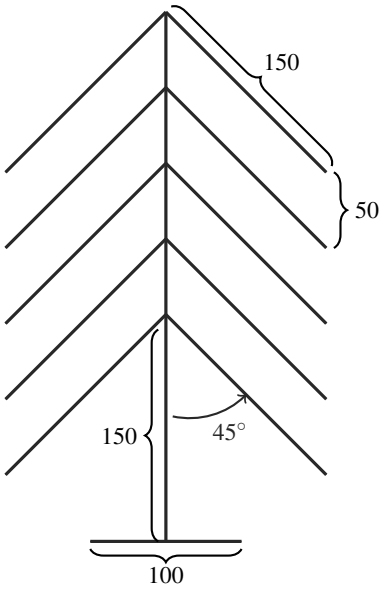


Abbildung 2.23

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. (Probiere 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 49 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**.

Hinweis für die Lehrperson Aus didaktischer Sicht ist die Modularität ein sehr dankbares Konzept, mit dem man so früh wie möglich anfangen soll. Meiden Sie im Programmierunterricht Programmiersprachen, die dieses Konzept nicht von Anfang an unterstützen oder eine komplexe Syntax zu seiner Verwendung erfordern. Modularisierte Programmierweise ermöglicht es, eine anspruchsvolle Programmiertätigkeit auszuüben, ohne zu merken, dass man etwas Schwieriges tut. Stellen Sie sich mal vor, was für komplexe Gedanken es erfordert, vier Schleifen ineinander zu „verschachteln“ und die Korrektheit des Programms zu überprüfen. Beim modularen Programmentwurf ist der Körper einer Schleife einfach ein Programm (keine Schleife) und man merkt gar nicht, dass man dreimal hintereinander eine Schleife in eine andere Schleife verschachtelt hat.

Im Folgenden wollen wir genau diesen modularen Entwurf von Programmen kennen lernen. 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 49 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 51 zu zeichnen.

Aufgabe 3.4 Schreibe ein Programm mit Namen `QUAD50`, welches das Quadrat 50×50 zeichnet. Verwende `QUAD50`, um das Bild aus Abb. 1.10 auf Seite 32 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 44 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 44 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 zehn mal, 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 51 zu zeichnen.

Aufgabe 3.8 Verwende die modulare Entwurfstechnik, um das Bild aus Abb. 2.20 auf Seite 52 zu zeichnen.

Aufgabe 3.9 Verwende die modulare Entwurfstechnik, um die Bilder aus Abb. 2.19 auf Seite 52 und Abb. 2.22 auf Seite 54 zu zeichnen.

Aufgabe 3.10 Zeichne drei Bäume wie den aus Abb. 2.23 auf Seite 54 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.

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)

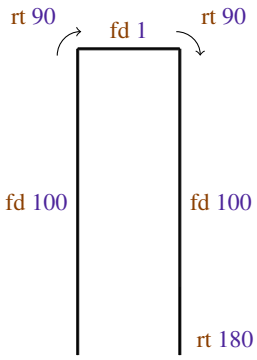


Abbildung 3.1

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 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

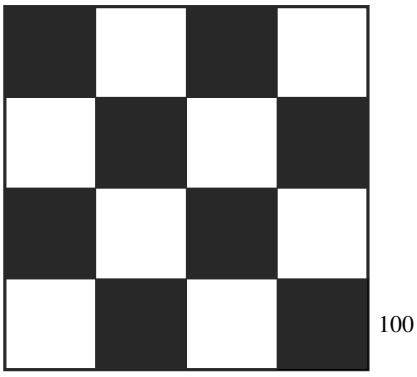


Abbildung 3.4

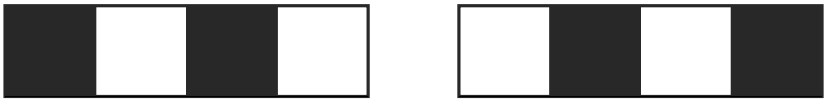


Abbildung 3.5

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 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-Schachfeldes wie in Abb. 3.5 in Zeilen zerlegen.

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 auf der vorherigen Seite 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 resultierende 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 auf der nächsten Seite 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 auf der nächsten Seite).

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

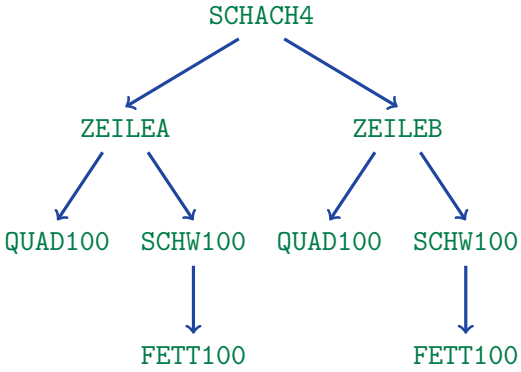


Abbildung 3.6

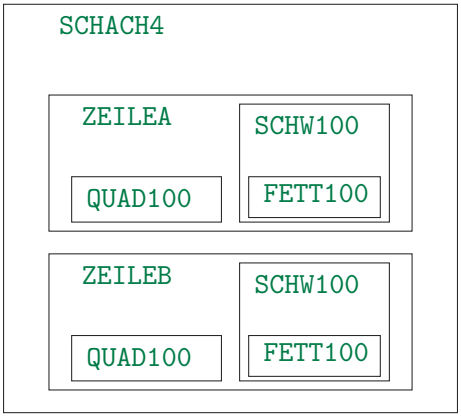


Abbildung 3.7

Hier ist direkt zu sehen, welches Programm ein Unterprogramm eines anderen 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 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 64 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 beendest, 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. Alle abgespeicherten Programme kannst du nicht nur verwenden, sondern jederzeit auch im Fenster des Editors ändern, korrigieren, verbessern oder weiterentwickeln.

Hinweis für die Lehrperson Jetzt ist es höchste Zeit, die Abspeicherung und das Laden von abgespeicherten Programmen der Klasse beizubringen. Eine ausführliche Beschreibung findet sich im Abschnitt „Die Programmierumgebung in LOGO“ auf Seite 12.

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 auf dem Rechner 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?
10. Wie kann man in LOGO etwas „flicken“?

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.

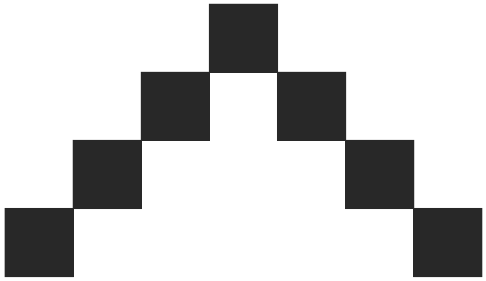


Abbildung 3.8

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

4. Zeichne das Bild aus Abb. 3.9.

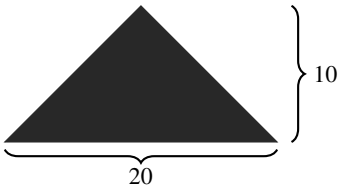


Abbildung 3.9

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

6. Zeichne das Häuschen aus Abb. 3.10 auf der nächsten Seite. Die Proportionen darfst du selbst wählen.

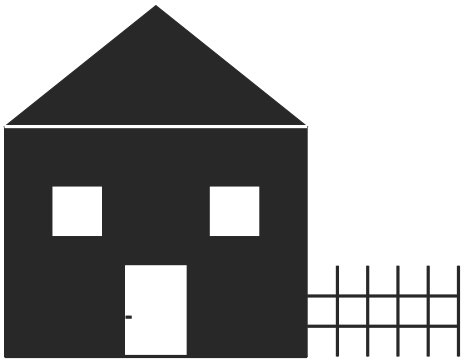


Abbildung 3.10

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

8. Zeichne das Bild aus Abb. 3.11.

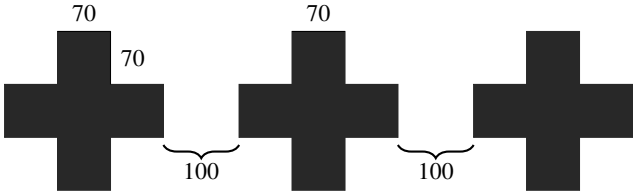


Abbildung 3.11

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.

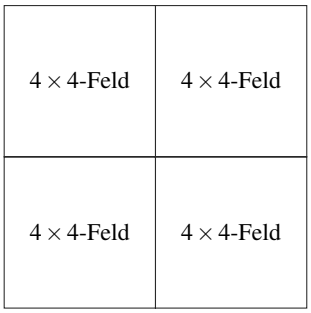


Abbildung 3.12 Eine modulare Komposition eines 8×8-Schachfeldes

```
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 63.

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 63 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.

Das folgende Programm fängt mit der Zeichnung des 4×4-Schachfeldes 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 69 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.

Hinweis für die Lehrperson Diese Lektion dient eher der Festigung des bisherigen Stoffes und zur Entfaltung der Kreativität durch Erzeugung farbiger Bilder nach eigener Vorstellung. Für diese Lektion reichen üblicherweise 1 bis 2 Unterrichtsstunden.

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 zehn mal gedreht und zwar immer um

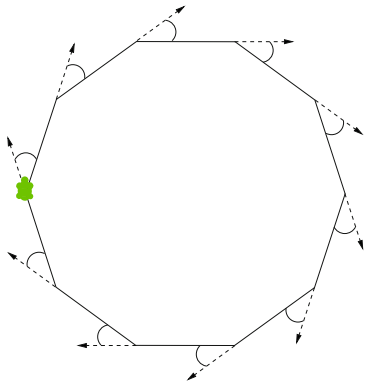


Abbildung 4.1

einen gleichgroßen Winkel. Also

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

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

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 hinreichend 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 Versuche die Halbkreise aus Abb. 4.2 zu zeichnen.

(a) (b)

Abbildung 4.2

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 4.7 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 vervollständigen:

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

Aufgabe 4.8 Denke dir eine ähnliche Aufgabe wie in Aufgabe 4.7 aus und schreibe ein Programm dazu.

Aufgabe 4.9 Ersetze das 12-Eck aus Aufgabe 4.7 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, Grün 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

```
setpencolor        X
setzte die Farbe   Eine Zahl als
                   Parameter zur
                   Farbbestimmung
```

wechselt die Schildkröte von der aktuellen Farbe in die Farbe mit dem Wert `X`. Statt eine Zahl zu schreiben, kannst du auch direkt die gewünschte Farbe durch ihren Namen (Tabelle 4.1) angeben. Somit darfst du zum Beispiel den Befehl `setpencolor darkblue` verwenden.

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

Aufgabe 4.11 Zeichne das Bild in Abb. 4.3. Die Schildkröte ist am Anfang an dem gemeinsamen Punkt (in dem Schnittpunkt) der beiden Kreise.

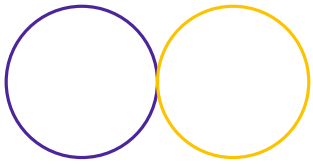


Abbildung 4.3

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. 4.4. Der Umfang der Kreise ist jeweils 540 Schritte.

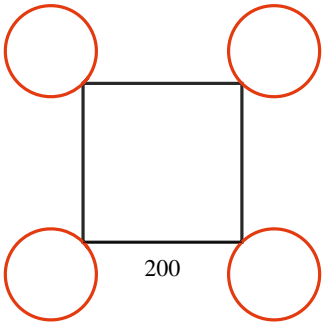


Abbildung 4.4

4. Schreibe ein Programm zum Zeichnen des Bildes aus Abb. 4.5. Die Größe der Kreise darfst du selbst wählen.

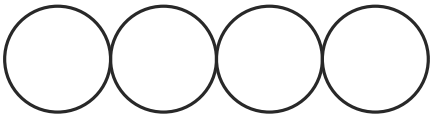


Abbildung 4.5

5. Schreibe ein Programm zum Zeichnen des Bildes aus Abb. 4.6.

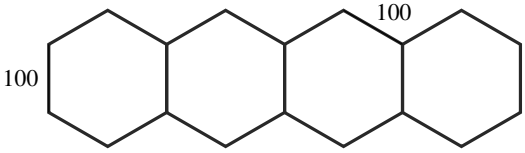


Abbildung 4.6

6. Kannst du mit Hilfe des Radiergummimodus und des Stiftmodus das Bild aus Abb. 4.6 auf der vorherigen Seite in vier nebeneinander stehende Häuser umwandeln? Wie die Häuser aussehen sollen, ist dir überlassen. Ein Beispiel für ein Haus siehst du in Abb. 4.7.

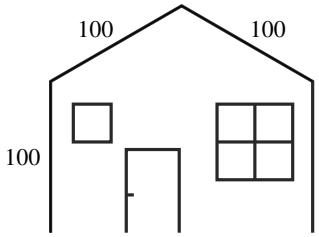


Abbildung 4.7

7. Zeichne mit Gelb das Netz aus Abb. 4.8.

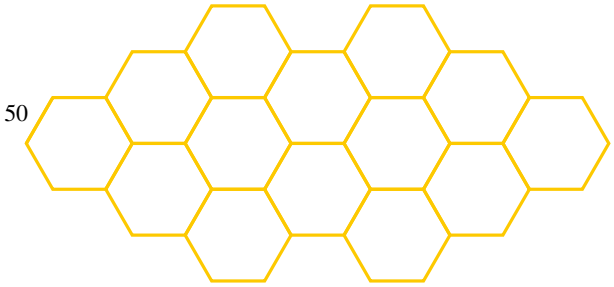


Abbildung 4.8

8. Zeichne vier Kreise wie in Abb. 4.9 mit den Umfängen 360, 540, 720 und 900.

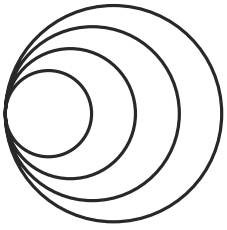


Abbildung 4.9

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* $\times$ *Seitenlänge*. Wenn wir Kreise als regelmäßige 360-Ecke zeichnen, ist der Umfang 360$\times$ *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 4.6

Einen Halbkreis zeichnet man, indem man statt 360-mal nur 180-mal mittels `rt 1` dreht. Wenn die Schildkröte für das Bild aus Abb. 4.2(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. 4.2(b) zu zeichnen, können wir unterschiedlich vorgehen. Zum Beispiel können wir den Halbkreis in Abb. 4.2(b) als den zweiten Teil eines Kreises sehen, bei dem der Halbkreis in Abb. 4.2(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. 4.2(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. 4.6 auf Seite 81 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. 4.6 auf Seite 81 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
```

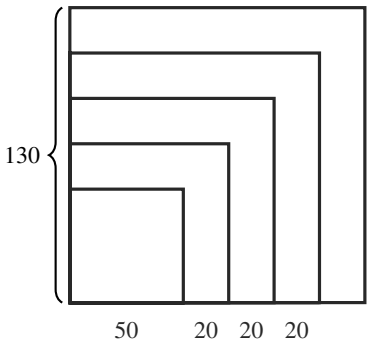


Abbildung 5.1

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

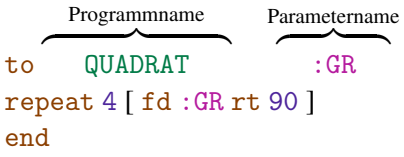
Und so weiter. Das sieht langweilig aus, oder? Wir schreiben immer fast das gleiche Programm. Der Unterschied liegt nur in der Zahl mit gelbem Hintergrund, die der Seitenlänge des jeweiligen Quadrats entspricht. Wäre es nicht schöner und praktischer einen Befehl

```
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:



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

Hinweis für die Lehrperson Eine der schwersten Hürden des Programmierunterrichts ist das Konzept der Variable. Parameter sind nichts anderes als „passive“ Variablen, also Variablen, deren Wert sich während der Ausführung des entsprechenden Programmaufrufs nicht ändert. Deswegen ist die Einführung des Parameters eine wichtige didaktische Vorgehensweise auf dem Weg zur Bewältigung des Konzepts von Variablen. Unsere Unterrichtsexperimente zeigen, dass man das Konzept von Parametern ab dem Alter von 9 Jahren erfolgreich unterrichten kann, obwohl das Konzept der Variablen frühestens ab dem 6. Schuljahr für große Klassen zugänglich ist. Da mit der Einführung von Programmparametern die Anforderungen im Programmierunterricht wesentlich steigen, empfehlt es sich, in dieser Lektion konsequent alle Aufgaben durch zuarbeiten.

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).

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

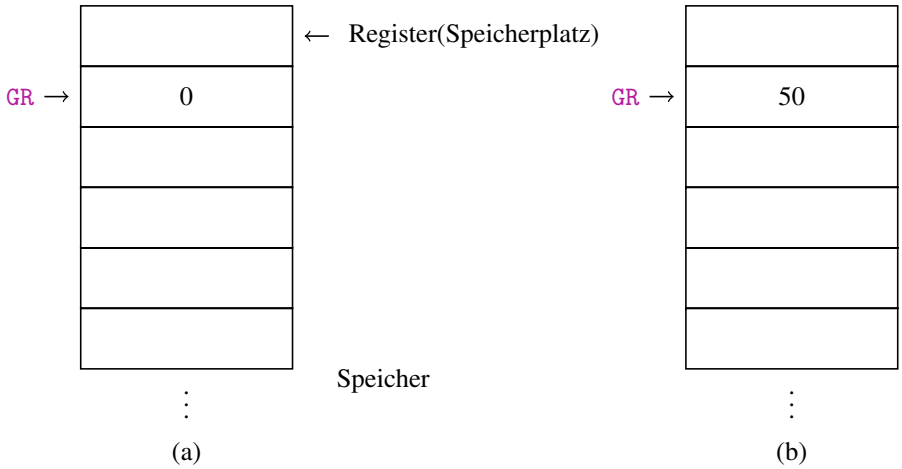


Abbildung 5.2 Schematischer Aufbau des Rechnerspeichers, der aus einer großen Anzahl von Registern besteht. Register sind die kleinsten Einheiten des Speichers, die man ansprechen kann.

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 86 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 mit einem Parameter, 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 zeichnet.

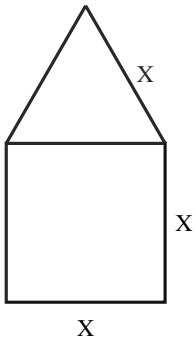


Abbildung 5.3

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

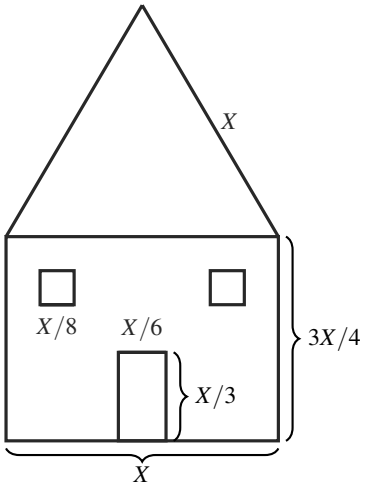


Abbildung 5.4

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 44 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 39 und Abb. 2.2 auf Seite 40 zeichnen.

Aufgabe 5.11 Schreibe ein Programm, das eine frei wählbare Anzahl von Radzähnen (Abb. 2.5 auf Seite 44) zeichnet.

Beispiel 5.2 Wir wollen zuerst ein Programm `KREISW :UM` entwerfen, dass für einen gegebenen Wert für `UM` einen Kreis vom Umfang `UM` zeichnet. Dafür betrachten wir das Programm

```
to KREISW :UM
repeat 360 [ fd :UM/360 rt 1 ]
end
```

Der Umfang des gezeichneten Kreises, welches eigentlich ein 360-Eck ist, kann durch die Anzahl der Seiten (360) mal die Seitengröße (`UM`/`360`) berechnet werden und ist somit tatsächlich `UM`.

Jetzt wollen wir ein Programm für die Zeichnung in Abbildung 5.5 auf der nächsten Seite entwerfen. Dabei soll der Umfang des großen Kreises frei wählbar sein und die zwei kleineren Kreise sollten den halben Umfang haben. Es gibt mehrere Möglichkeiten, wie wir vorgehen können. Wir zeichnen zuerst einen großen Halbkreis, danach den rechten kleinen Kreis, dann vervollständigen wir den großen Kreis und letztendlich zeichnen wir den linken kleinen Kreis.

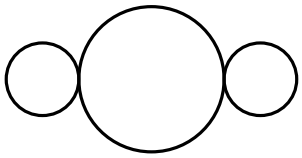


Abbildung 5.5

```
to KOPF :UMFANG
repeat 180 [ fd :UMFANG/360 rt 1 ]
repeat 360 [ fd :UMFANG/720 lt 1 ]
repeat 180 [ fd :UMFANG/360 rt 1 ]
repeat 360 [ fd :UMFANG/720 lt 1 ]
end
```

Aufgabe 5.12 Zeichne das Bild in Abb. 5.6. Die Größe des Umfangs des größten Kreises soll frei wählbar sein, die kleineren Kreise sollen einen dreimal kleineren Umfang haben als der große Kreis.

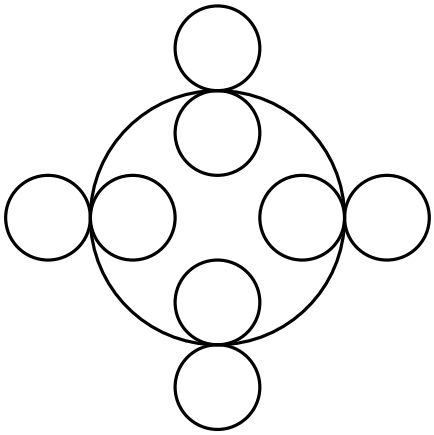
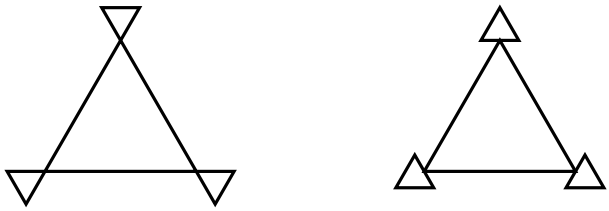


Abbildung 5.6

Aufgabe 5.13 Zeichne die Bilder in Abb. 5.7 (a) und (b). Die Seitengröße des größten Dreiecks sollte frei wählbar sein und die kleinen Dreiecke sollten viermal kleiner werden als das große Dreieck.

Bisher haben wir nur Programme mit einem Parameter betrachtet. Dadurch war immer nur eine Eigenschaft (Größe, Anzahl der Wiederholungen, . . .) der Bilder frei wählbar. Es



(a) (b)

Abbildung 5.7

kann aber wünschenswert sein, zwei, drei oder mehr Parameter eines Bildes frei wählbar zu haben. Fangen wir mit einem einfachen Beispiel an.

Beispiel 5.3 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 5.14 Entwerfe ein Programm mit zwei Parametern, um Bilder zu zeichnen wie diejenigen in Abb. 5.5 auf der vorherigen Seite. Dabei sollen die Umfänge beider Kreise frei wählbar sein.

Aufgabe 5.15 Entwerfe ein Programm zum Zeichnen von Bildern wie in Abb. 5.6 auf der vorherigen Seite, wobei der Umfang des großen Kreises sowie der kleinen Kreise frei wählbar ist.

Aufgabe 5.16 Entwerfe ein Programm um Bilder wie in Abb. 5.7 auf der vorherigen Seite zu zeichnen. Beide Größen der Dreiecke sollten mittels zwei Parametern frei wählbar sein. Schaffst du es ein Programm zu schreiben, in dem man die Seitenlänge jedes der vier Dreiecke unabhängig voneinander wählen darf?

Aufgabe 5.17 Schreibe ein Programm zum Zeichnen eines beliebigen roten (rot ausgefüllten) Rechtecks. Dabei sollen beide Seitenlängen frei wählbar sein.

Aufgabe 5.18 Das folgende Programm zeichnet das Parallelogramm aus Abb. 5.8.

```
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.

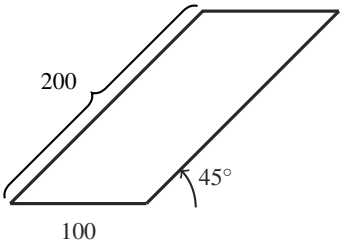


Abbildung 5.8

Beispiel 5.4 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 50 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 5.19 Teste das Programm aus Beispiel 5.4 und erkläre, wie es funktioniert.

Aufgabe 5.20 Schreibe ein Programm zum Zeichnen beliebiger Rechtecke in beliebiger Farbe.

Aufgabe 5.21 Schreibe ein Programm zum Zeichnen einer frei wählbaren Anzahl Türme wie in Abb. 5.9. Dabei sollen die Höhe und die Breite (oder auch andere Eigenschaften) des Turms frei wählbar sein.

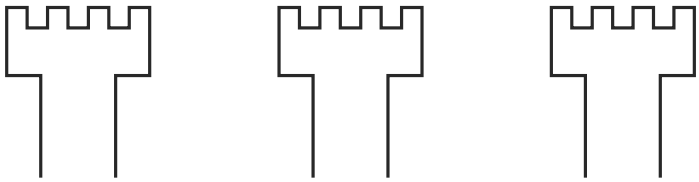


Abbildung 5.9

Aufgabe 5.22 Schreibe ein Programm zum Zeichnen von vier Kreisen wie in Abb. 5.10. Dabei soll die Größe von allen vier Kreisen frei wählbar sein und für jeden der Kreise sollte auch die Farbe frei wählbar sein.

Abbildung 5.10

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 (oder der Parameter, falls mehrere vorhanden sind), 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 anstelle 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.

Wenn man die Definition (Beschreibung) eines Programms mit einem Parameter eintippt, wird das Programm vom Rechner gespeichert und für jeden Parameter wird ein Speicherplatz mit dem Namen des Parameters versehen. Dieser Speicherplatz, welcher Register genannt wird, ist somit für den Parameter reserviert. Beim Aufruf des Programms mit konkreten Zahlen (Parameterwerten) werden automatisch diese Zahlen in den Registern mit den entsprechenden Namen gespeichert.

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 dahinter steht?

Kontrollaufgaben

1. Schreibe ein Programm zum Zeichnen der Bilder in Abb. 7 auf Seite 32, 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. 5.11. Die Größe der drei Paare von Zweigen sowie die Farbe des Baums sollen frei wählbar sein.

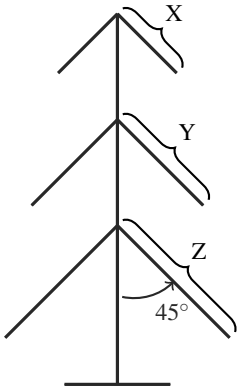


Abbildung 5.11

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 Kommastellen (`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 5.18
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 5.22
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 kennen gelernt 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,

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, sollte 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.

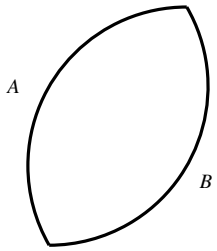


Abbildung 6.1

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.

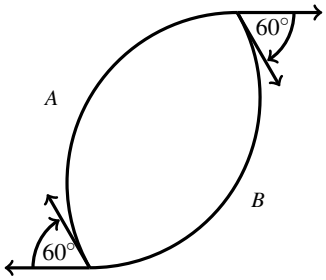


Abbildung 6.2

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°.

$$360^\circ - 120^\circ - 120^\circ = 120^\circ$$

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

$$\frac{120^\circ}{2} = 60^\circ$$

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 $\frac{360^\circ}{2} = 180^\circ$, gemacht werden. Also ist die Drehung an der Spitze des Blattes

$$180^\circ - \texttt{:LANG}.$$

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.

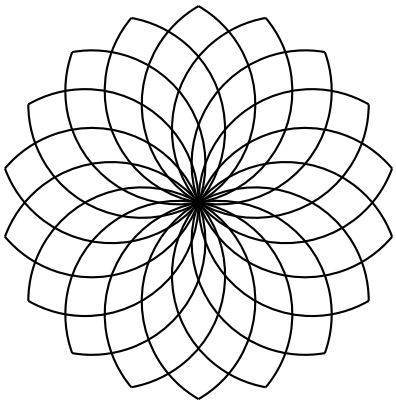


Abbildung 6.3

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. Das Ziel ist es, auf die Einführung des Konzeptes der Variable vorzubereiten. Hier geht es darum, genauer zu verstehen, wie sich bei Aufrufen von Programmen und Unterprogrammen die gespeicherten Zahlen (Inhalte) der Register ändern.

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 Unterprogramms 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

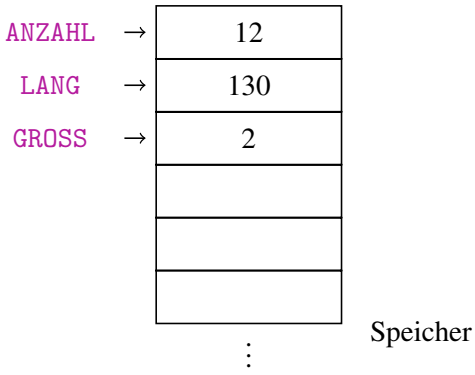


Abbildung 6.4

nicht möglich, weil wir den Parametern :LANG und :GROSS gleich am Anfang durch 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

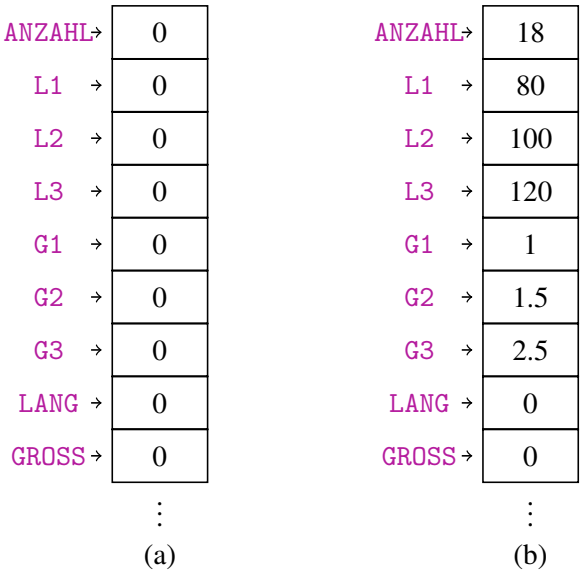


Abbildung 6.5

Speicher schon die Register mit den Namen LANG und GROSS, die den Parametern von 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 für ANZAHL, L1 und G1 eingesetzt und somit entsteht die Anweisung

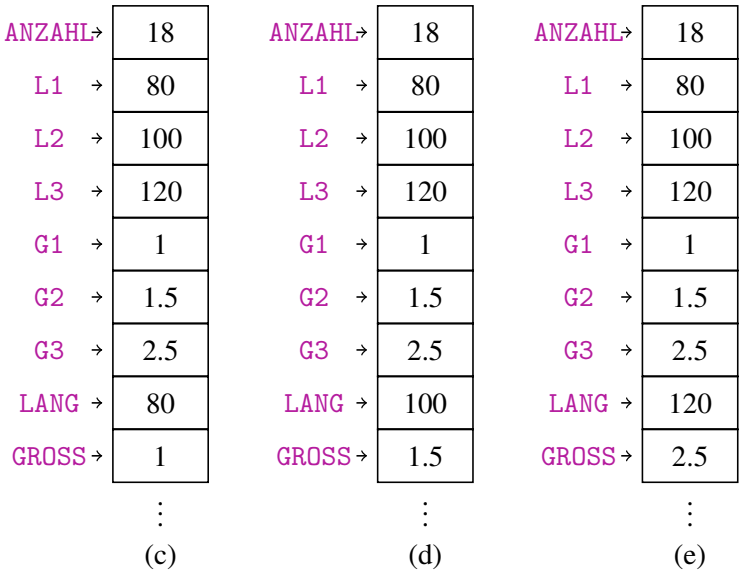


Abbildung 6.6

```
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)). Die vorherigen Inhalte `80` und `1` der Register `LANG` und `GROSS` werden damit in diesen

Registern gelöscht. 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 Speicherinhalts 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

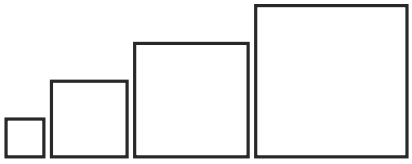


Abbildung 6.7

Unterprogramm verwenden werden und die Größe von allen vier Quadraten soll frei wählbar sein. Beschreibe dann die Entwicklung des Speicherinhalts 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 Unterprogramm zum Zeichnen von drei gefüllten, nebeneinander liegenden 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 `BLUMEN3` 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 Unterprogramm `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 bezüglich des Hauptprogramms 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. Somit soll man in dieser Lektion beim Üben der Änderung der Register vermeiden, Programme mit gleichnamigen globalen und lokalen Variablen zu betrachten.

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 $HOR \times VER$ zeichnet. Jetzt betrachten wir die Aufgabe, das Bild aus Abb. 6.9 zu zeichnen.

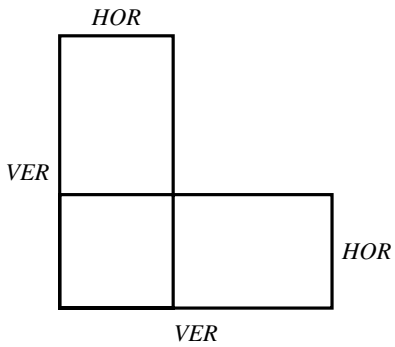


Abbildung 6.9

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 111), die die Übergabe der Parameterwerte dokumentiert.

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

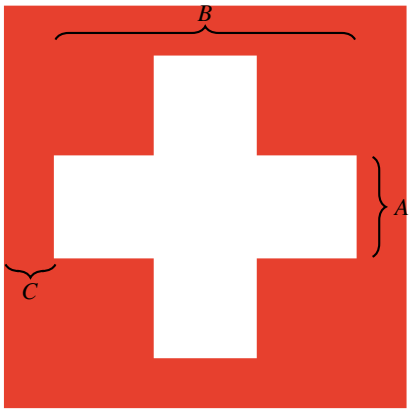


Abbildung 6.10

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.

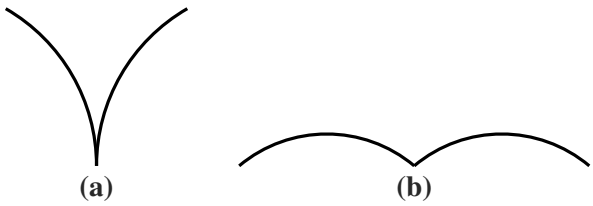


Abbildung 6.11

2. Zeichne das Bild aus Abb. 6.12.

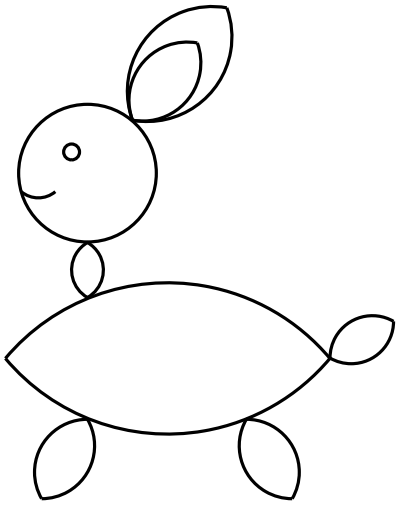


Abbildung 6.12

3. Entwickle in modularer Art ein Programm, das Bilder wie in Abb. 3.8 auf Seite 69 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 70. 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.

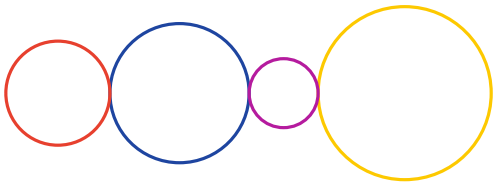


Abbildung 6.13

6. Entwickle ein Programm zum Zeichnen des Bildes in Abb. 4.8 auf Seite 82. 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 86 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 Unterprogramms 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 erfolgreich herausfordernde 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?

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 66 und 3.21 auf Seite 66 entwickelt hast.

Beispiel 7.1 In Beispiel 2.2 auf Seite 47 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 zehn mal 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 49. Versuche dabei die Programmlänge zu minimieren.

Aufgabe 7.7 Schreibe ein kurzes Programm zum Zeichnen des Bildes aus Abb. 2.18 auf Seite 51.

Aufgabe 7.8 Schreibe ein kurzes Programm zum Zeichnen des Bildes aus Abb. 2.22 auf Seite 54.

Aufgabe 7.9 Versuche die Länge des Programms `SCHACH4` zum Zeichnen eines 4×4-Schachfeld 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 41 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 41. 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.

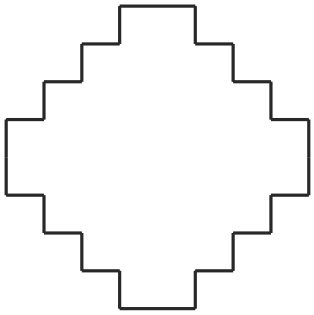


Abbildung 7.1

Aufgabe 7.11 Versuche ein kurzes Programm zum Zeichnen einer frei wählbaren Anzahl Pyramiden wie in Abb. 2.3 auf Seite 41 zu entwickeln. Die Stufenhöhe soll dabei fünf sein und die Anzahl der Stufen soll frei wählbar sein.

Hinweis für die Lehrperson An dieser Stelle endet der Teil des Lehrbuchs, der für die ersten fünf bis sechs Schuljahre geeignet ist. Den Rest des Lehrmittels empfehlen wir ab dem siebten, in Ausnahmefällen ab dem sechsten Schuljahr, da ein Verständnis für den Begriff der Unbekannten aus der Mathematik erwünscht ist. Dazu kommt weiter, dass der Begriff der Variablen (ab Lektion 8) in der Informatik schwieriger nachzuvollziehen ist als in der Mathematik.

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 44 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 zehn mal wiederholt und zusätzlich wird auch der letzte Befehl `rt 90` dieser Schleife zehn mal 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 ]
```

[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.

hat die Berechnungskomplexität von nur

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

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 44 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 44 schreiben? Es müsste mit weniger als 60 ausgeführten Grundbefehlen machbar sein.

Aufgabe 7.14 Versuche ein effizientes Programm zum Zeichnen des Bildes aus Abb. 4.6 auf Seite 81 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 47 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 49.

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 TEIL 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.

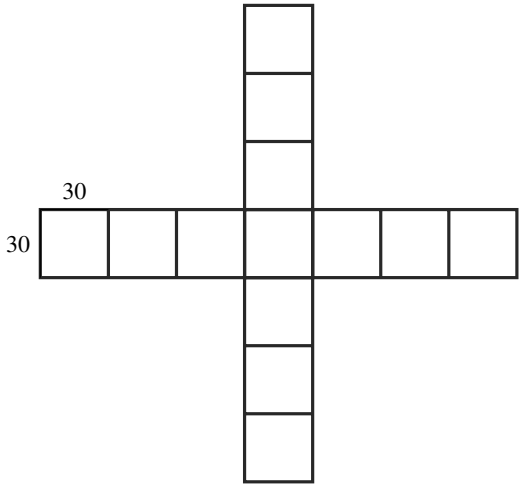


Abbildung 7.3

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.

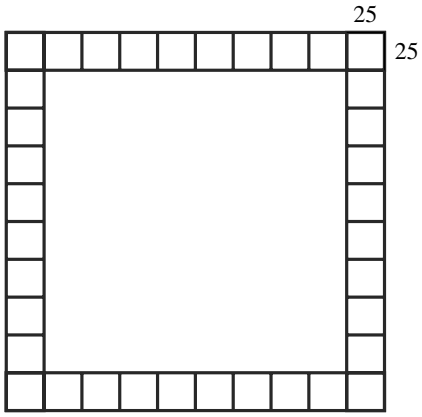


Abbildung 7.4

5. Entwirf Programme zum Zeichnen des Bildes aus Abb. 2.19 auf Seite 52. 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 69 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 Schachfelds 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.

11. Versuche ein kurzes Programm zum Zeichnen des Bildes aus Abb. 7.5 zu entwerfen. Dabei sollen die Größe des großen Quadrats, sowie die Größe der kleinen vier Quadrate frei

wählbar sein. Erinnere dich dabei an die Tatsache, dass man Befehle wie `fd :X + 2 * :Y - 3 * :Z` verwenden darf.

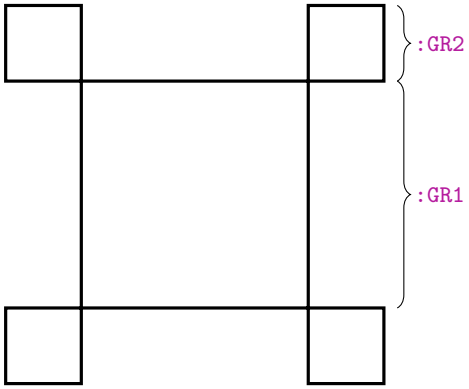


Abbildung 7.5

12. Schreibe ein Programm, mit dem du Bilder wie in Abb. 7.5 zeichnen kannst, welches genau 23 Befehle durchführt. Begründe, warum es kein schnelleres Programm zur Zeichnung des Bildes geben kann.

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 62) 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. 7.6. Die Höhe des Musters ist X×GR.

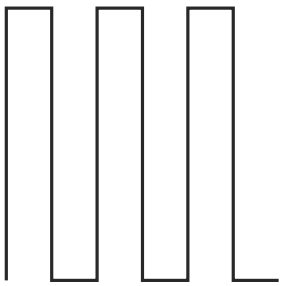


Abbildung 7.6

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 KURZFELD 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·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

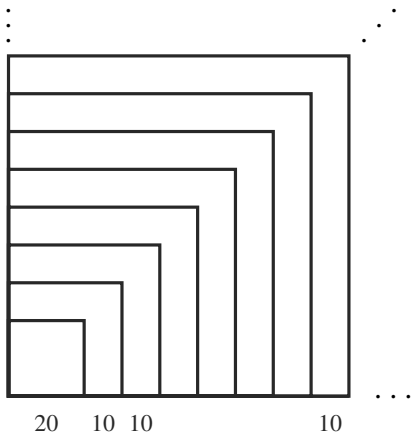


Abbildung 8.1

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 seines 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 Parameterwerts	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 des Programms 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.

Hinweis für die Lehrperson Programmieren erfordert unumgänglich das volle Verständnis des Konzepts der Variablen. Dabei ist der korrekte Umgang mit Variablen die erste ernsthafte Hürde im Programmierunterricht. Deswegen ist hier wichtig, selbständig sehr viele Aufgaben zu lösen und somit das Übungsangebot dieser Lektion zu nutzen.

Aufgabe 8.2 Mit VIELQ zeichnen wir eine Folge von Quadraten, die immer um zehn größer werden. Jetzt wollen wir die Vergrößerung 10 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.

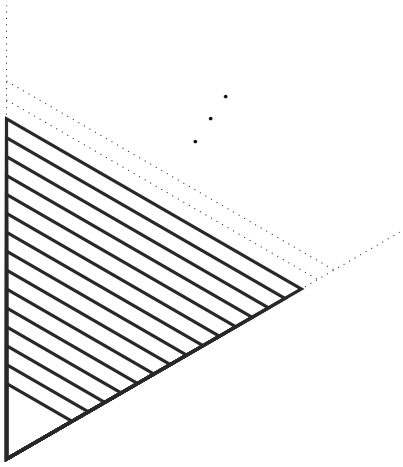
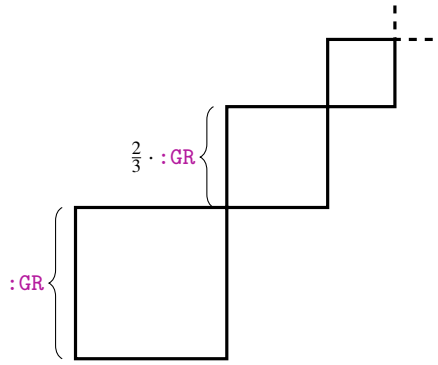


Abbildung 8.2

Beispiel 8.1 Beim Zeichnen des Bildes aus Abb. 8.1 auf Seite 138 müssten wir immer größere Quadrate aus dem selben Punkt zeichnen. Beim Zeichnen des Bildes in Abb. 8.4 auf Seite 143 ändern sich zwei Anforderungen. Erstens starten die nachfolgenden Quadrate immer in der zum Startpunkt des vorherigen Quadrats gegenüberliegenden Ecke. Zweitens wächst ihre Große nicht um einen „additiven“ Faktor +10 oder +`:ST` (Aufgabe 8.2 auf der vorherigen Seite), sondern schrumpft um den „multiplikativen“ Faktor $2/3$. Genauer bedeutet es, dass

$$\text{die Länge des nachfolgenden Quadrats} = \frac{2}{3} \cdot \text{die Länge des vorherigen Quadrats.}$$



Die Große des ersten Quadrats sowie die Anzahl der Quadrate soll frei wählbar sein. Dieses Ziel erreichen wir durch folgendes Programm:

```
to ABB8P3 :GR :AN
repeat :AN [ repeat 6 [ fd :GR rt 90 ] rt 180
             make "GR  2 * :GR / 3 ]
end
```

Wir erweitern das Programm um die Möglichkeit, den multiplikativen Faktor `:MULT` für die Verkleinerung der Seitenlänge der Quadrate frei zu wählen.

```
to ABB8P3 :GR :AN :MULT
repeat :AN [ repeat 6 [ fd :GR rt 90 ] rt 180
             make "GR  :GR * :MULT ]
end
```

Aufgabe 8.4 Teste das Programm ABB8P3, in dem du ABB8P3 folgendermaßen aufrufst: ABB8P3 100 8 0.5 und ABB8P3 20 5 1.2. Was beobachtest du?

Aufgabe 8.5 Ändere das Programm ABB8P3 so, dass die Quadratgröße um einen additiven Faktor :ADD statt um einen multiplikativen Faktor :MULT „schrumpft“.

Aufgabe 8.6 Entwerfe ein Programm zum Zeichnen der Bilder wie in Abb. 8.3. Die Anzahl der Quadrate :ANZ, die Größe des ersten Basisquadrats :GR sowie die Reduktion der Quadratgröße von Quadrat zu Quadrat :ADD soll frei wählbar sein. Danach ändere das Programm so, dass die Quadratgröße um einen multiplikativen Faktor schrumpft.

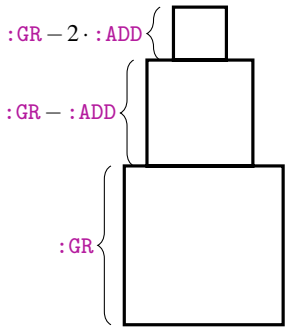


Abbildung 8.3

Aufgabe 8.7 Wir haben schon gelernt, Schnecken mit fester Größe zu zeichnen. Jetzt solltest du ein Programm schreiben, mit dem man beliebige Schnecken wie in Abb. 8.4 auf der nächsten Seite zeichnen kann.

Aufgabe 8.8 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 additive 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.

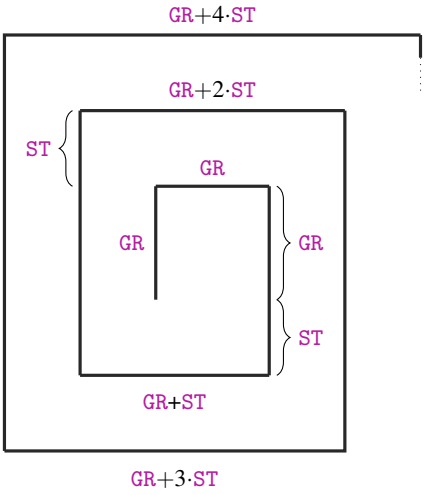


Abbildung 8.4

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 definiert wird und sofort den Wert 50 zugewiesen bekommt.

Aufgabe 8.9 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 $30 \cdot 30$. 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.

Es spricht aber nichts dagegen, die Variablen X und Y sofort in der ersten Zeile des Programms wie folgt zu definieren:

```
to T3 :A :B :C :X :Y
make "X :B * :B
make "Y 4 * :A * :C
make "X :X - :Y
QUADRAT :X
end
```

Das Programm wird genau die selbe Tätigkeit ausüben. Wir müssen darauf achten, dass wir beim Aufruf von T3 fünf Zahlen angeben, z. B. T3 1 (-7) 12 0 0. Somit werden die Register X und Y von Anfang an benannt und erhalten beim Aufruf des Programms sofort den Wert 0.

Bei geschickter Klammerung kann das Programm T3 noch zum kürzeren Programm T4 abgeändert werden:

```
to T4 :A :B :C :X
make "X ( :B * :B ) - ( 4 * :A * :C )
QUADRAT :X end
```

Aufgabe 8.10 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 8.11 Zeichne eine Tabelle (ähnlich Tab. 8.1), in der du die Entwicklung der Speicherinhalte beim Aufruf

```
T3 20 25 10
```

dokumentierst.

Aufgabe 8.12 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 8.13 Schreibe ein Programm, das zuerst ein gleichseitiges Dreieck mit der Seitenlänge 20 zeichnet. Danach zeichnet es 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 Vielecke soll dabei frei wählbar sein.

Beispiel 8.2 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.

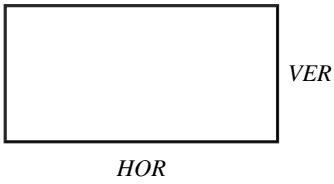


Abbildung 8.5

Wir wissen, dass (s. Abb. 8.5)

$$\texttt{UM} = 2 \cdot VER + 2 \cdot HOR \tag{8.1}$$

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 8.14 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 Variablenwerts 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 8.15 Entwickle ein Programm zum Zeichnen von rechtwinkligen gleichschenkligen Dreiecken mit wählbarer Schenkellänge.

Aufgabe 8.16 Entwickle ein Programm zum Zeichnen von drei Quadraten wie in Abb. 8.6. 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. 8.6 stimmt es, da $5^2 = 3^2 + 4^2$ gilt.

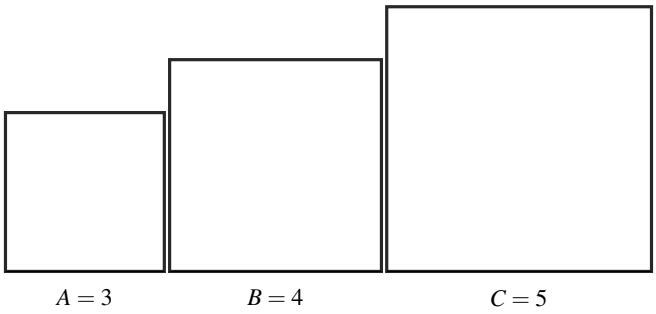


Abbildung 8.6

Aufgabe 8.17 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 8.18 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 8.3 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. 8.7).

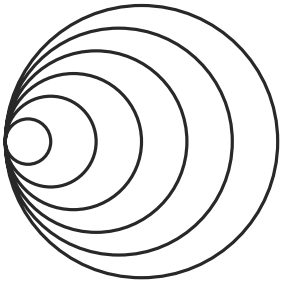


Abbildung 8.7

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 8.19 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 8.20 Schreibe ein Programm, das genau zwölf Kreise mit wachsender Größe wie in Abb. 8.7 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 additive 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 ihres 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 ursprünglichen Inhalt eines Registers, wenn man in diesem Register einen neuen Wert speichert?

8. Lokale Parameter eines Hauptprogramms können ihre Werte während der Laufzeit des Hauptprogramms ändern. Trotzdem betrachten wir sie noch immer als Parameter. Kannst du erklären warum nicht?

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. 8.8). 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?

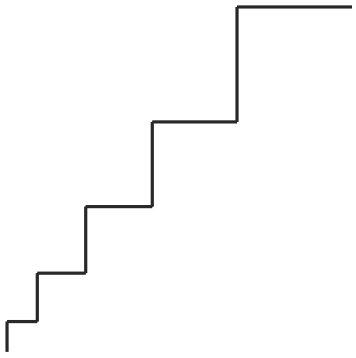


Abbildung 8.8

2. Entwickle ein Programm zum Zeichnen von Pyramiden wie in Abb. 8.9. Die Anzahl der Stufen, die Größe der Basisstufe sowie die additive Reduzierung der Stufengröße beim Übergang in eine höhere Ebene sollen frei wählbar sein.

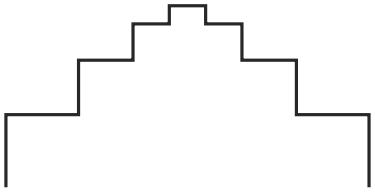


Abbildung 8.9

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?

4. 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.

5. Zeichne eine sechseckige Spirale wie in Abb. 8.10 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.

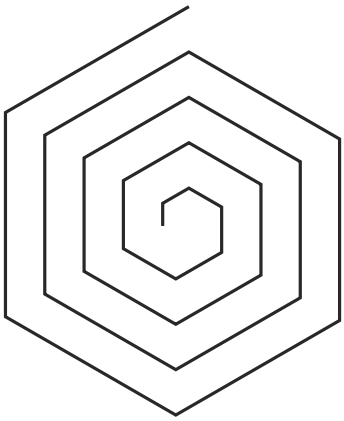


Abbildung 8.10

6. 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`?

7. Entwickle ein Programm zum Zeichnen einer frei wählbaren Anzahl von Kreisen wie in Abb. 8.7 auf Seite 150. Dabei soll der Kreisumfang von Kreis zu Kreis immer um einen multiplikativen Faktor 1.2 wachsen. Die Größe des kleinsten Kreises soll frei wählbar sein.

8. Entwickle ein Programm zum Zeichnen einer frei wählbaren Anzahl von Halbkreisen wie in Abb. 8.11 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.

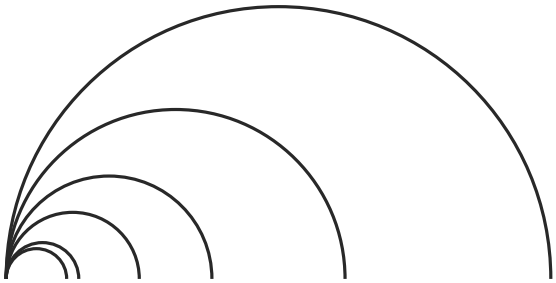


Abbildung 8.11

9. Entwickle ein Programm zum Zeichnen des Bildes aus Abb. 8.12

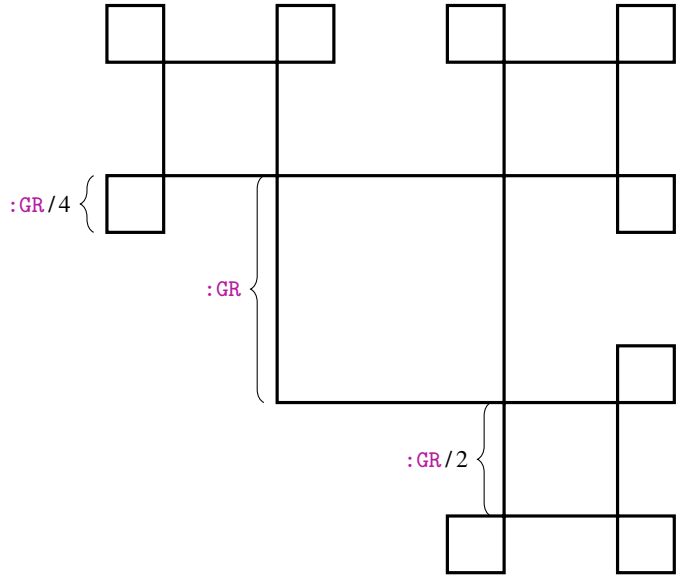


Abbildung 8.12

Lösungen zu ausgesuchten Aufgaben

Kontrollfrage 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.

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 8.7
Das Programm zum Zeichnen von Schnecken (Abb. 8.4 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 8.9
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 8.10
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 8.15
Die Grundidee ist, dass zuerst die Winkelgrößen in einem gleichschenkligen rechtwinkligen Dreieck bekannt sein müssen (Abb. 8.13 auf der nächsten Seite).

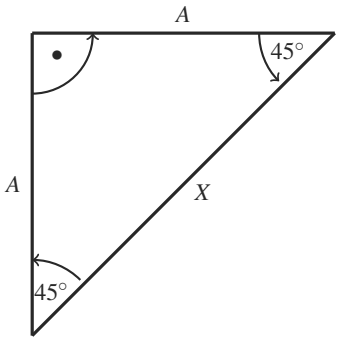


Abbildung 8.13

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. 8.13 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 8.19
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 folgende Zahl beinhaltet:

`u` + `a` · `v`.

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.

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.

Hinweis für die Lehrperson Das Konzept der Variablen muss vor dem Übergang zur Lektion 10 vollständig gemeistert werden. Diese Lektion festigt das bisher Gelernte und erklärt den genauen Umgang mit den Variablen mit besonderem Fokus auf die Übertragung der Variablenwerte zwischen einem Hauptprogramm und seinen Unterprogrammen. Es sollen unbedingt nicht nur Programmieraufgaben gelöst werden, sondern auch Tabellen aufgestellt werden, die die Änderung der Variablenwerte während dem Lauf des Programms dokumentieren. Zwei bis drei Unterrichtsstunden plus hinreichend Hausaufgaben reichen zur Bearbeitung dieser Lektion.

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

```
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 somit 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.

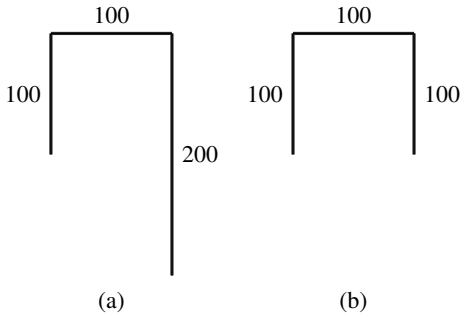


Abbildung 9.1

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 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, und nicht nur nach der Umsetzung ganzer Zeilen des Hauptprogramms, aufzeigt. Ausführlich bedeutet, den Aufruf `TEST1 :GR` nicht als einen Befehl zu betrachten, sondern ihn die entsprechenden Befehle in `TEST1` zu unterteilen.

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
```

„ausgeben“. 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 schützt 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` auf dem Bildschirm erscheinen 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).

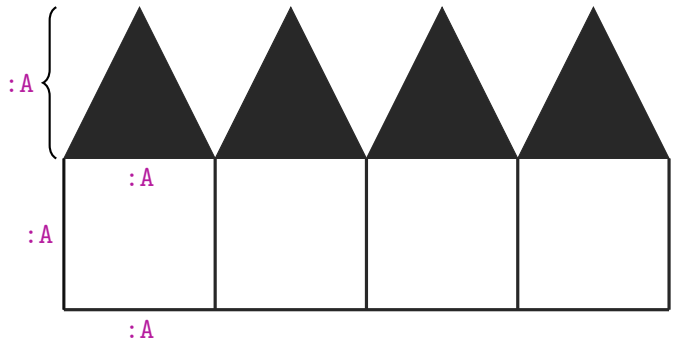


Abbildung 9.2

```
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 auf der vorherigen Seite) 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`, $\texttt{GR}+10$, $\texttt{GR}+20$, …, $\texttt{GR}+(\texttt{AN}-1)\cdot 10$. 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

Verzweigungen 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

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 zu melden, 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 unterhalb 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.

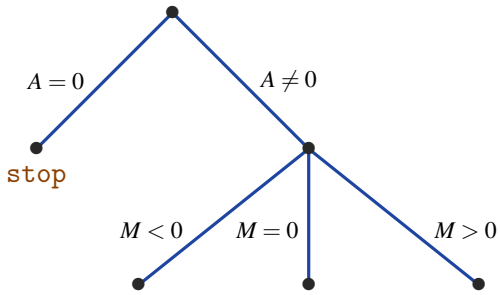


Abbildung 10.1

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?

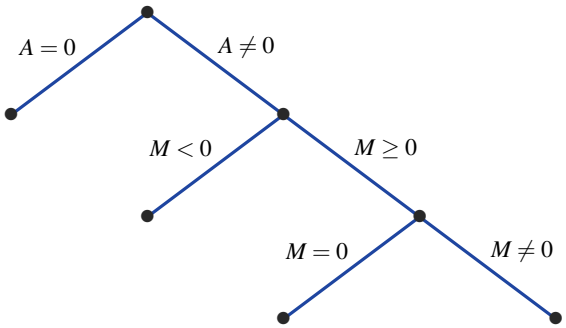


Abbildung 10.2

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 unterschiedlich von 0 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:

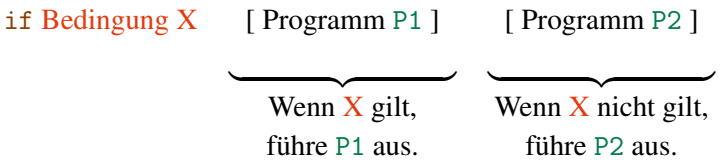
```
to LINGL :A :B :C :D
if :A=:C [ if :B=:D [ pr [ alle reellen Zahlen ] stop ] ]
if :A=:C [ pr [ keine Lösung ] stop ]
make "X (:D-:B)/(:A-:C)
pr [ X= ] pr :X if :X>0 [ LEITER :X ]
end
```

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, welche die tatsächliche allgemein verwendete Grundstruktur ist:



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 ] ]
end
```

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 $X \leq 0$.

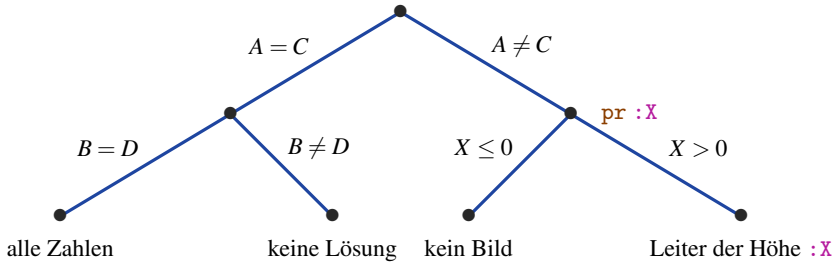


Abbildung 10.3

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.

Hinweis für die Lehrperson An dieser Stelle ist es wieder wichtig, dass man nicht nur programmiert, sondern auch mit Papier und Stift arbeitet und ein gutes Verständnis für die logische Struktur der Verzweigungen entwickelt.

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 [...] [...].

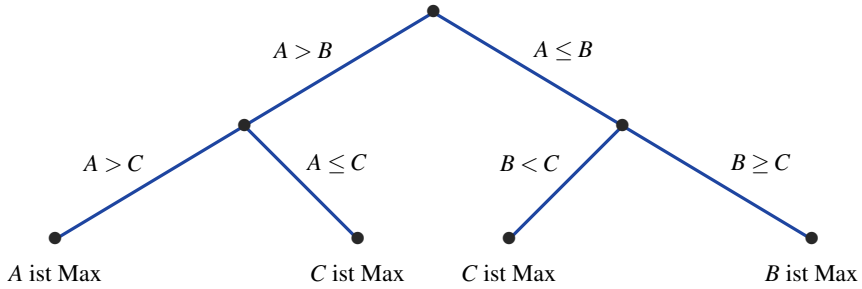


Abbildung 10.4

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. 8.10 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 additiven 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. 8.10 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. 8.10 auf Seite 154 von innen nach außen. Die innere Seitenlänge :LA sowie die additive Verlängerung :ADD sind frei wählbar. Das Programm soll enden, wenn mindestens 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. 8.11 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 multiplikativen 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 des Befehls 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. 8.4 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. 8.6 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 entsprechende Bedingung nicht mehr erfüllt ist. Also wird die Schleife wiederholt, so lange die Bedingung der while-Schleife 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. 8.6 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. 8.6 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. 8.7 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. 8.9 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 138 abgebildet. Das kleinste Quadrat hat eine wählbare Größe `:GR` und die Seitenlänge vergrößert sich um einen wählbaren additiven 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 [ f(n)= ] pr :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 skizziert.

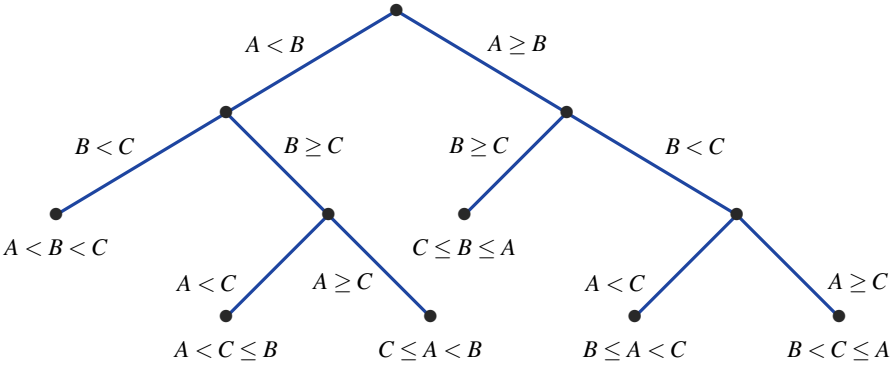


Abbildung 10.5

Jede Verzweigung in Abb. 10.5 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:

```
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 :X1 :X2 :X3 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 oft 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.

Hinweis für die Lehrperson Diese Lektion enthält keine neuen Programmierkonzepte und darf deswegen übersprungen werden. Sie kann aber zur Festigung der Befehle `if` und `while` beitragen. Das Ziel dieser Lektion ist, einige Unterrichtsthemen aus der Mathematik, insbesondere aus der Geometrie, zu behandeln. Unter anderem zeigt sie, wie wichtig es ist, mathematische Methoden genau zu verstehen und zu beschreiben. Ohne diese Voraussetzungen können Methoden nicht mittels Programmen automatisiert werden.

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.

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 `:R` wie 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 auf der nächsten Seite). Welche Schnittpunktmengen können diese beiden Kreise haben?

[1] In XLOGO kann der Befehl `pi` dafür verwendet werden.

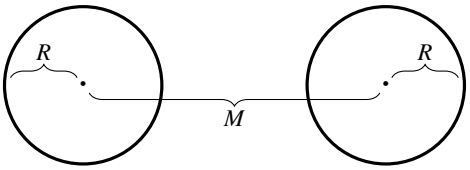


Abbildung 11.1

Aufgabe 11.2 Entwickle ein Programm zum Zeichnen der Olympischen Ringe (Abb. 11.2). Der Radius der Kreise soll frei wählbar sein.

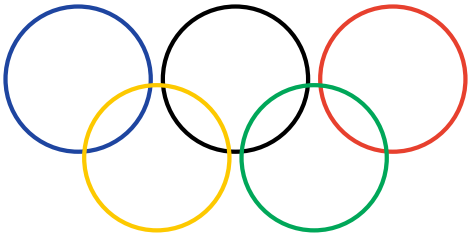


Abbildung 11.2

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 in SUPERLOGO vom aktuellen Punkt (aus der aktuellen Position) eine Linie zum 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 auf der nächsten Seite. Die Länge der Hypotenuse wird immer als c bezeichnet und $\gamma = 90°$.

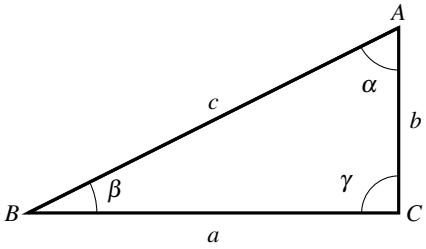


Abbildung 11.3

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 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.

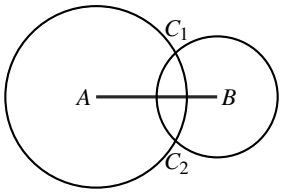


Abbildung 11.4

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 $b = |\overline{AC}|$ (Abb. 11.4). 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.

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).

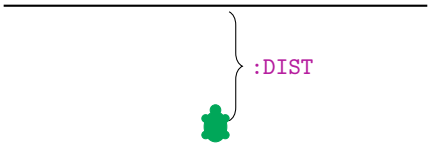


Abbildung 11.5

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.

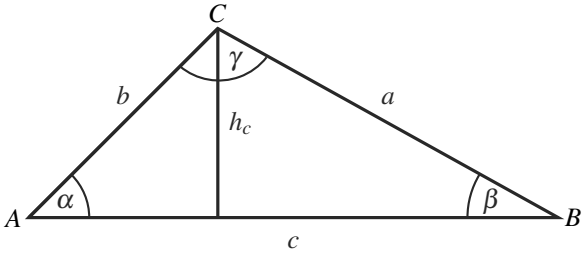


Abbildung 11.6

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). 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.

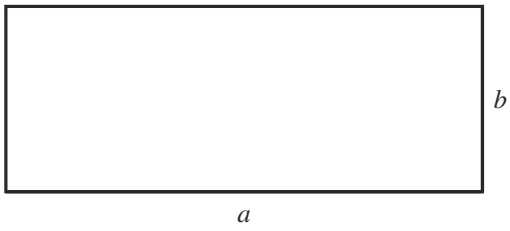


Abbildung 11.7

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 oder gleich :SUM ist. Die zwei Nachfolger einer Zahl X sind $X+1$ und $X+2$. Wir suchen die kleinste natürliche Zahl X mit der Eigenschaft

$$X + X + 1 + X + 2 \geq \text{SUM}, \quad \text{d. h.}$$
$$3 \cdot X + 3 \geq \text{SUM}.$$

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` in SUPERLOGO 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. Entwerfe ein Programm `MOD2 :X`, das für eine gerade ganze positive Zahl X ein Quadrat der Grösse $X \times X$ zeichnet. Falls X keine gerade ganze positive Zahl ist, soll das Programm einen Kreis mit dem Umfang `:X` zeichnen.

9. 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.

10. Gegeben ist eine natürliche Zahl X. Finde mit einem Programm die kleinste natürliche Zahl n, so dass 2^n grösser ist als die Summe von X und $X + 1$.

11. Schreibe ein Programm, dass die kleinste positive ganze Zahl findet, so dass $m!$ grösser ist als die Summe von m und seinen $a \cdot m$ Nachfolgerzahlen. Die Zahl a ist eine positive ganze Zahl, welche die Eingabe des Programms sein soll.

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 auf der nächsten Seite).

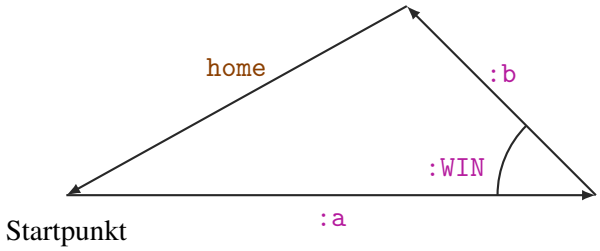


Abbildung 11.9

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 203, 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
```

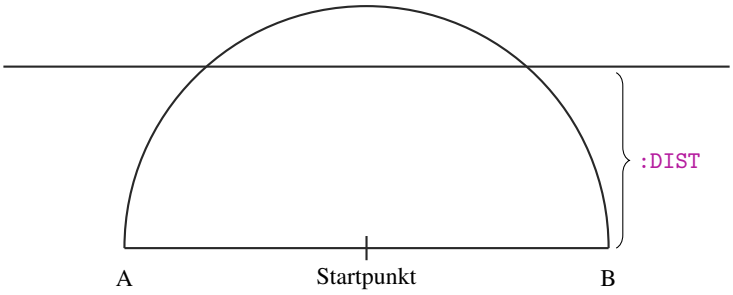


Abbildung 11.10

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). 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). 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.

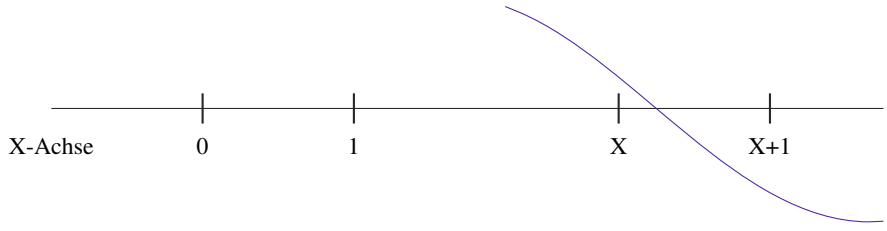


Abbildung 11.11

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+:b*:Y*:Y+:c*:y+:d>0]
                [ make "X :X+1 make "Y :Y+1]
if :m<0 [ while [ :a*:Y*:Y+:b*:Y*:Y+:c*:y+:d<0]
                [ make "X :X+1 make "Y :Y+1]
pr :X
end
```

Lektion 12

Rekursion

Rekursive Programme zu verstehen und selbst zu entwerfen gehört zu den höheren Künsten eines Programmierers. Wenn man diese Lektion erfolgreich meistert, dann darf man sich stolz Frau Programmiererin oder Herr Programmierer nennen. Um dies zu erreichen, folgen wir weiter unserem roten Pfad mit dem Ziel, wenig zu tippen und trotzdem viel zu bewirken. Mit unglaublich kurzen rekursiven Programmen lernst du sehr komplexe Bilder (sogar Fraktale) zu zeichnen.

Hinweis für die Lehrperson Diese Lektion ist eine wesentliche Steigerung des Schwierigkeitsgrades im Vergleich zu vorherigen Lektionen. Wir empfehlen, sie erst in den letzten zwei Jahren des Gymnasialstudiums zu behandlen.

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
```

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
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 auf die gleiche 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 rekursiven 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 rekursive 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 so lange eine `:ECK`-eckige Spirale, bis die letzte Seite nicht mehr kleiner 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 letztendlich der erste Aufruf abgeschlossen wird und somit die Ausführung des ganzen Programms beendet wird.

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 sogenannte **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.

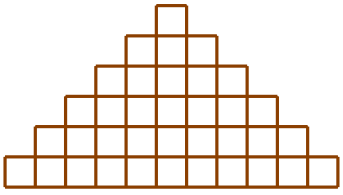


Abbildung 12.1

Aufgabe 12.13 Im Beispiel 8.3 haben wir das Programm `AUGE :AN :UM :NACH` zum Zeichnen der Bilder aus Abb. 8.7 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. 8.8 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.

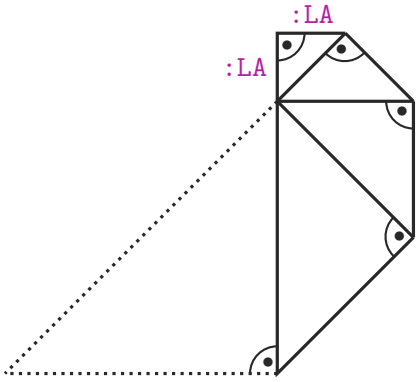


Abbildung 12.2

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 auf der nächsten Seite anschaulich dargestellt. Hier sehen wir anschaulich, dass die Rekursionstiefe dieses Aufrufs 4 ist. Als Wichtigstes bleibt aber zu beachten, dass

> *die Ausführung eines rekursiven Aufrufs nicht beendet wird, bevor die Ausführung des in ihm verschachtelten rekursiven Aufrufs 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,
```

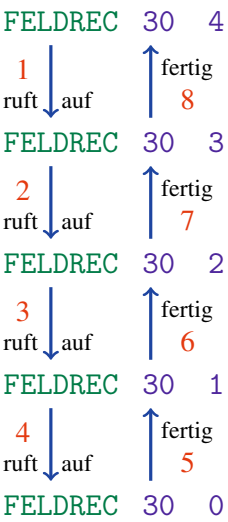


Abbildung 12.3

	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

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 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:

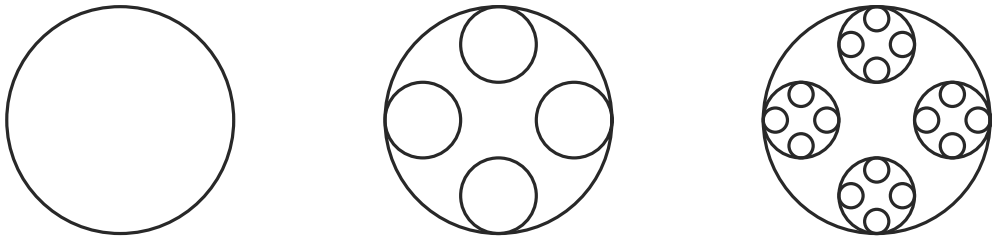
> *Bei rekursiven Aufrufen übergibt das laufende Programm an den in ihm verschachtelten Aufruf die Werte für die Variablen. Wenn ein Aufruf beendet wird, werden die Werte seiner Variablen gelöscht und es kommt zu keiner Übertragung der Variablenwerte nach oben. Als aktuelle Variablenwerte werden die vorher gespeicherten Werte des Aufrufs verwendet, der gerade ausgeführt wird, d. h. 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,



(a) (b) (c)

Abbildung 12.4

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 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
```

Aufgabe 12.20 Tippe das Programm KREISREC ab und teste es mit den Aufrufen KREISREC 600 2, KREISREC 1000 3 und KREISREC 1200 4. Verstehst du die Reihenfolge, in welcher die einzelnen Kreise gezeichnet werden?

Aufgabe 12.21 Schreibe ein rekursives Programm, womit man eine ähnliche Folge von Bildern wie in Abb. 12.4 auf der vorherigen Seite zeichnen kann. Es sollen nur jeweils drei Muster mit dem Abstand eines Drittels des Kreises gezeichnet werden.

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 entsprechen 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 12.22 Zeichne die Baumstruktur der rekursiven Aufrufe analog zu Abb. 12.5 für den Aufruf KREISREC 999 2 und erstelle eine entsprechende Tabelle, die die Entwicklung der Variablenwerte von :UM und :TIEF in allen Rekursionstiefen dokumentiert.

Aufgabe 12.23 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.

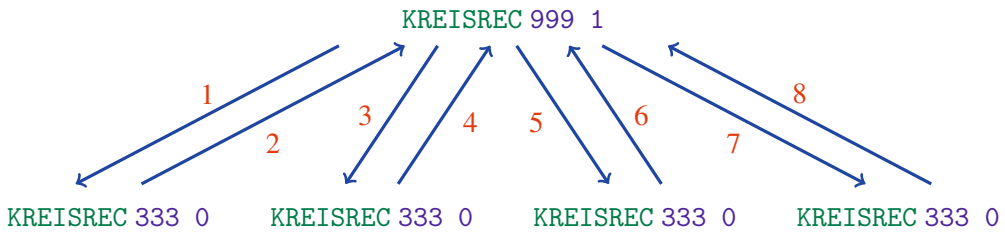


Abbildung 12.5

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 222 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 der vorherigen Seite) sieht die Klammerdarstellung der rekursiven Aufrufe wie folgt aus:

$R_1 = (\ )\ (\ )\ (\ )\ (\ )\ .$

Der Ablauf der Ausführung des Aufrufs `KREISREC 999 2` kann durch folgende Klammerfolge

$$R_2 = (\ (\ )\ (\ )\ (\ )\ (\ )\ )\ \ (\ (\ )\ (\ )\ (\ )\ (\ )\ )\ \ (\ (\ )\ (\ )\ (\ )\ (\ )\ )\ \ (\ (\ )\ (\ )\ (\ )\ (\ )\ )$$

dargestellt werden. Um die rekursive Struktur transparent darzustellen, beobachten wir, dass

$$R_2 = (\ R_1\ )\ \ (\ R_1\ )\ \ (\ R_1\ )\ \ (\ R_1\ ).$$

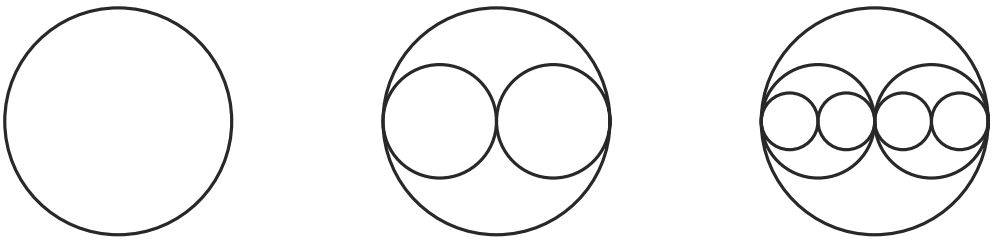
Aufgabe 12.24 Welche Klammerfolge entspricht dem Aufruf `FELDREC 20 8`?

Aufgabe 12.25 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 12.26 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? Der Aufruf `KREISREC :UM :TIEF` hat die Tiefe `:TIEF`. Wie viele rekursive Aufrufe werden während der Ausführung von `KREISREC 2000 20` demnach realisiert?

Aufgabe 12.27 Aus einer Klammerfolge eines Aufrufs kann man die aktuelle 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 12.28 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

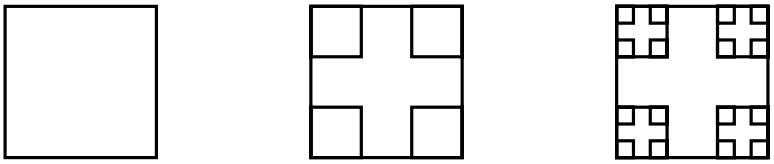


Abbildung 12.7

Aufgabe 12.29 Schreibe ein rekursives Programm, das die Folge von Bildern zeichnet, derer ersten drei Bilder in Abb. 12.7 gezeichnet sind. Die Seitenlänge der Quadrate wird in jedem Rekursionsschritt gedrittelt.

Aufgabe 12.30 In Abb. 12.8 siehst du die ersten fünf Bilder einer unendlichen Folde von Bildern, die man rekursiv erzeugen kann. Die Seiten der schwarz ausgefüllten Quadrate sind ein Sechstel der Seitengröße des entsprechenden Quadrats lang, mit dem sie gemeinsame Ecken haben. Die Größe der leeren Quadrate wird von einem Schritt zum nächsten immer gedrittelt. Entwerfe ein Programm, welches diese Folge zeichnet.

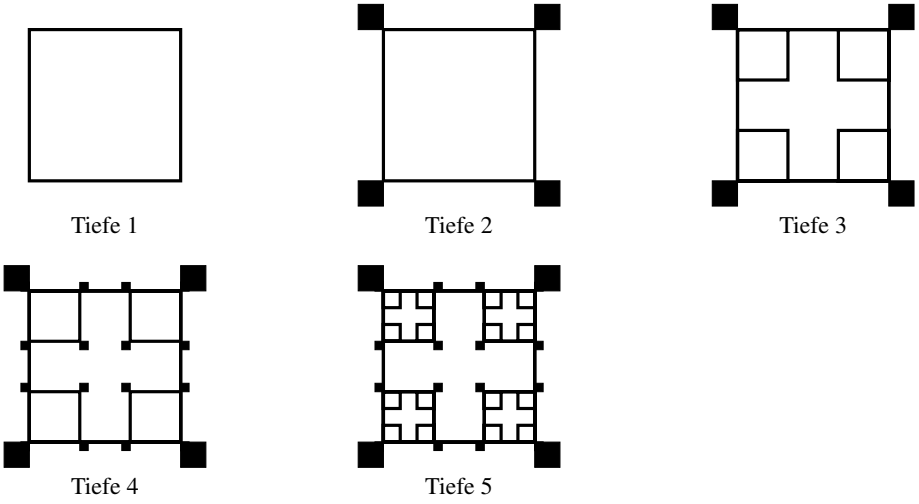


Abbildung 12.8

Aufgabe 12.31 Betrachte das folgende Programm:

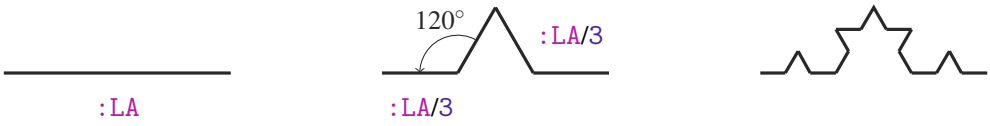
```
to FELDABREC :A :B
if :B=0 [ stop ]
repeat 4 [ fd :A/2 rt 90 ] FELDABREC :A/2 :B−1
fd :A/2
repeat 4 [ fd :A/2 rt 90 ] FELDABREC :A/2 :B−1
rt 90 fd :A/2 lt 90
repeat 4 [ fd :A/2 rt 90 ] FELDABREC :A/2 :B−1
bk :A/2
repeat 4 [ fd :A/2 rt 90 ] FELDABREC :A/2 :B−1
lt 90 fd :A/2 rt 90
end
```

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`.

Das rekursive Zeichnen von Bilderfolgen basierte bisher auf einem relativ einfachen Schema. Das hängt mit einer Eigenschaft dieser Bilderfolgen zusammen. Es ging darum, das gleiche Muster an unterschiedlichen Stellen der Bildschirmfläche zu erzeugen. Somit bewegt man sich schrittweise bei der Zeichnung der Grundmuster der Rekursionstiefe 1 und an geeigneten Stellen zeichnet man rekursiv die vorgegebenen Muster. Bei der Folge aus Abb. 12.4 zum Beispiel ist das Grundmuster der Tiefe 1 ein Kreis. Bei dieser Aufgabe zeichneten wir immer nur einen Viertelkreis und anschliessend rekursiv das Muster in einem kleineren Mass. Diese Vorgehensweise funktionierte mehr oder weniger in allen bisherigen Beispielen. Es gibt aber auch Bilderfolgen, die auf eine andere Art und Weise entstehen, wie Beispiel 12.3 zeigt.

Beispiel 12.3 Die Aufgabe besteht darin, das rekursive Muster aus Abb. 12.9 zu zeichnen. In der ersten Rekursionsstufe zeichnet man nur eine Linie der Länge `:LA` (Abb. 12.9(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. 12.9(b) durch eine entsprechend lange, gebrochene Linie ersetzt, erhält man Abb. 12.9(c). Deswegen zeichnet das folgende Programm auf der tiefsten Rekursionsebene eine Linie und „ersetzt“ rekursiv jeden geraden Weg durch einen geknickten Weg.



(a) (b) (c)

Abbildung 12.9

```
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. □

Wir sehen den Unterschied zum vorherigen Zeichnen gleicher Muster. In Beispiel 12.3 wird nur auf der tiefsten Rekursionsebene gezeichnet, weil die Längen aller gezeichneten Linien durch die Rekursionstiefe bestimmt sind. In Bildern wie in Abb. 12.4 wird auf jeder Rekursionsebene gezeichnet.

Aufgabe 12.32 Entwirf ein rekursives Programm, welches für die ersten drei Rekursionsstufen Bilder wie in Abb. 12.10 zeichnet.

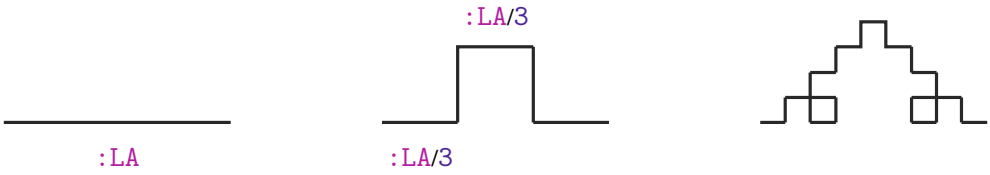


Abbildung 12.10

Aufgabe 12.33 Entwickle ein Programm, welches für die ersten drei Rekursionsstufen Bilder wie in Abb. 12.11 zeichnet.

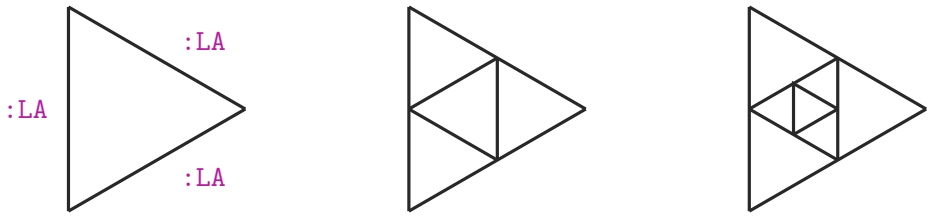


Abbildung 12.11

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. 12.12 zeichnen.

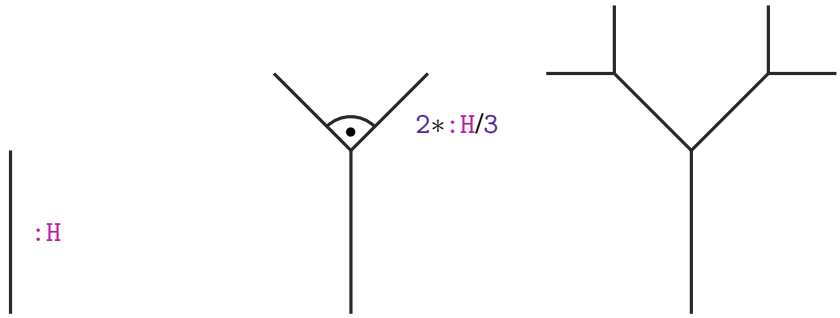


Abbildung 12.12

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 90 wait 500
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. 12.13 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. 12.13 auf der nächsten Seite, dass die Tiefe der Länge des längsten Weges vom Start über die Pfeile nach unten zum letzten Aufruf entspricht.

Die Tabelle 12.3 stellt die Entwicklung der Speicherinhalte dar, die den ersten neun Pfeilen des Baumdiagramms aus Abb. 12.13 auf der nächsten Seite entsprechen. Die untersten Werte in dieser Tabelle sind immer die Werte der Variablen `:H` und `:TIEF`, mit denen der Rechner aktuell arbeitet.

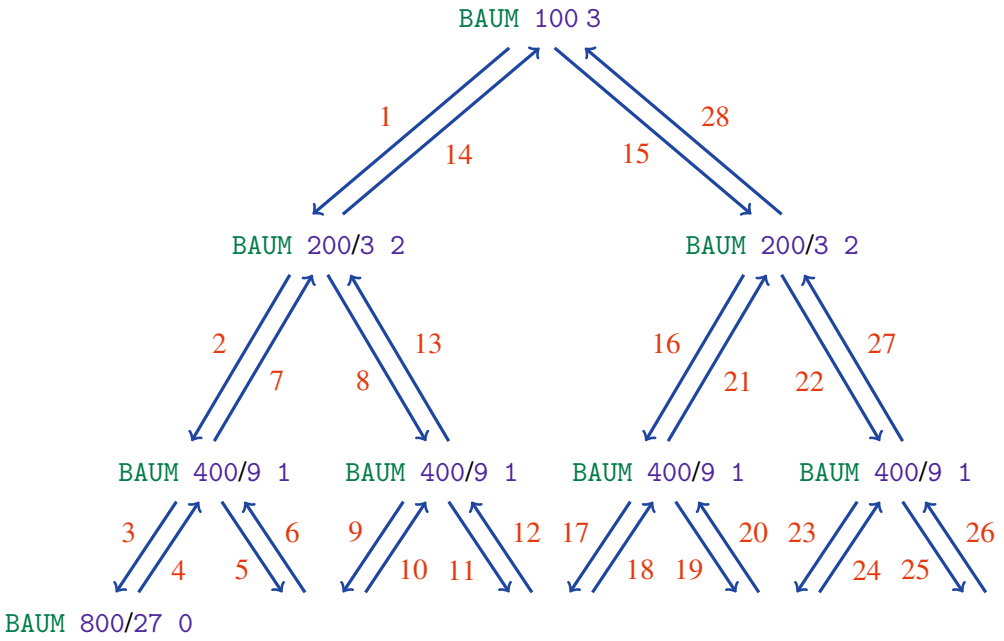


Abbildung 12.13

	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 12.34 Zeichne das Baumdiagramm und die entsprechenden Klammerfolgen der folgenden rekursiven Aufrufe:

a) `BAUM 80 4`

b) `FELDABREC 200 2`

c) `STAR 400 3`

Aufgabe 12.35 Vervollständige die Tabelle 12.3, indem du die Entwicklung der Speicherinhalte für den Aufruf `BAUM 100 3`, der Struktur der rekursiven Aufrufe aus Abb. 12.13 auf der vorherigen Seite entsprechend aufzeichnest. Somit sollte die Tabelle 29 Spalten haben.

Aufgabe 12.36 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 12.37 Entwickle ein rekursives Programm, das für die ersten drei Rekursionstiefen die Bilder aus Abb. 12.14 zeichnet.

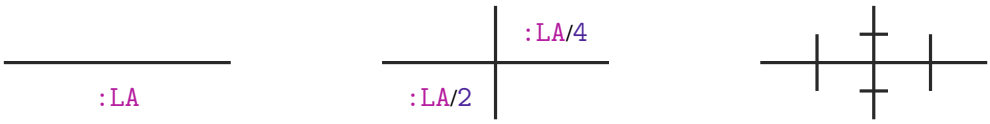


Abbildung 12.14

Aufgabe 12.38 Entwickle ein rekursives Programm, das für die ersten drei Rekursionstiefen die Bilder aus Abb. 12.15 zeichnet. Zeichne danach das Baumdiagramm der rekursiven Aufrufe für einen rekursiven Aufruf der Tiefe 3.

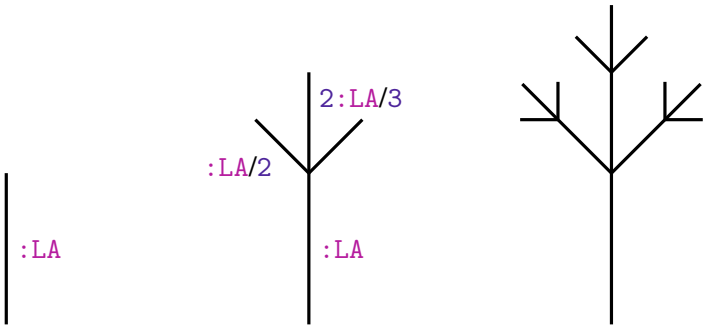


Abbildung 12.15

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. 12.16 zeichnet.

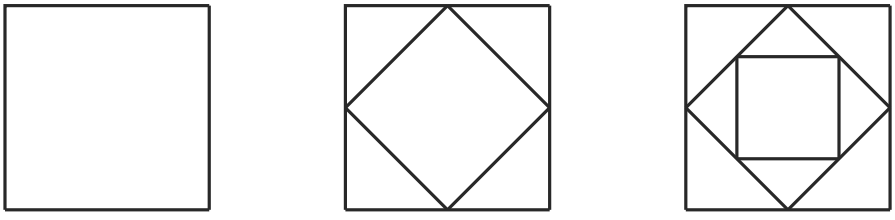


Abbildung 12.16

4. Entwirf ein rekursives Programm, das in den ersten drei Rekursionsstufen die Bilder aus Abb. 12.17 zeichnet. Die Seitenlänge des ersten Dreiecks soll frei wählbar sein.

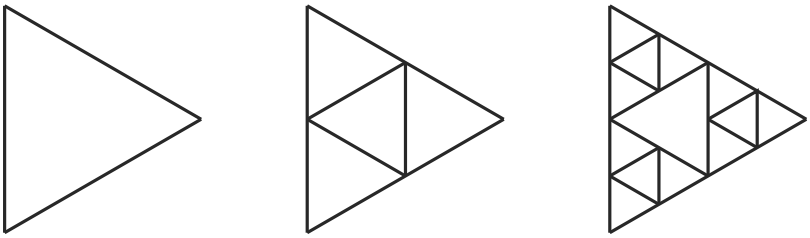


Abbildung 12.17

5. Wandle das Programm KREISREC aus Beispiel 12.2 zum Zeichnen der Bilder aus Abb. 12.4 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 selbst 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

$$T > \frac{\texttt{:MAX} - \texttt{:MIN1}}{\texttt{:ADD1}} \qquad \text{oder} \qquad T > \frac{\texttt{:MAX} - \texttt{:MIN2}}{\texttt{:ADD2}}.$$

Lektion 13

Integrierter LOGO- und Mathematikunterricht: Trigonometrie

Hinweis für die Lehrperson In dieser Lektion werden keine neuen Programmiertechniken vorgestellt. Die Lektion eignet sich zur Vertiefung des Trigonometrieunterrichts. Es wird geübt, mathematische Vorgehensweisen genau zu beschreiben. Weiter wird auch die Verwendung der Variablen, sowie die `while`-Schleife trainiert. Die Aufgaben dieser Lektion befassen sich nicht nur mit dem Zeichnen von Bildern.

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

$$\begin{aligned}
\frac{|\overline{SA_1}|}{|\overline{SB_1}|} = \frac{|\overline{SA_2}|}{|\overline{SB_2}|} &= \frac{|\overline{SA_3}|}{|\overline{SB_3}|}, \\
\frac{|\overline{A_1B_1}|}{|\overline{SB_1}|} = \frac{|\overline{A_2B_2}|}{|\overline{SB_2}|} &= \frac{|\overline{A_3B_3}|}{|\overline{SB_3}|}, \\
\frac{|\overline{A_1B_1}|}{|\overline{SA_1}|} = \frac{|\overline{A_2B_2}|}{|\overline{SA_2}|} &= \frac{|\overline{A_3B_3}|}{|\overline{SA_3}|}.
\end{aligned}$$

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, dass die Seitenlängen der Dreiecke gleich schnell (proportional) wachsen.

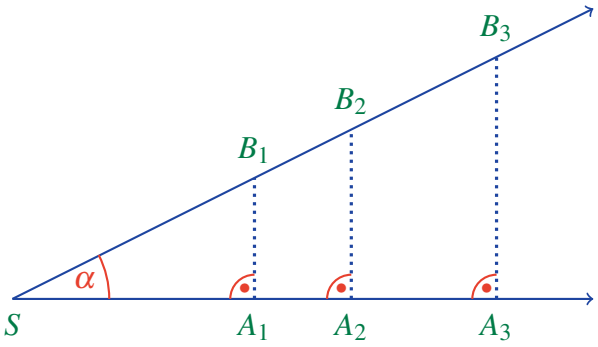


Abbildung 13.1

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 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.

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

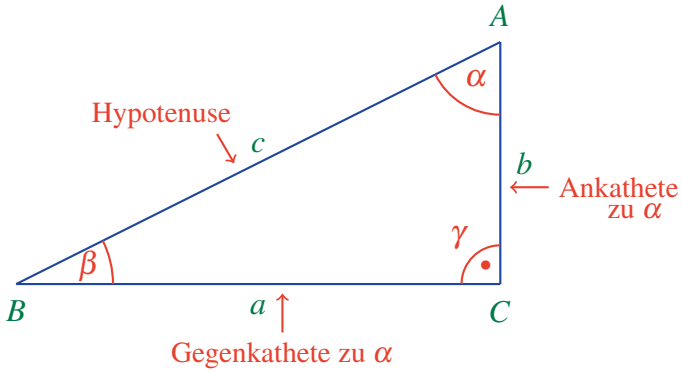


Abbildung 13.2

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 246) 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 246) 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 $\sin(0)$, $\sin(0.1)$, $\sin(0.2)$, ... 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 246) 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).

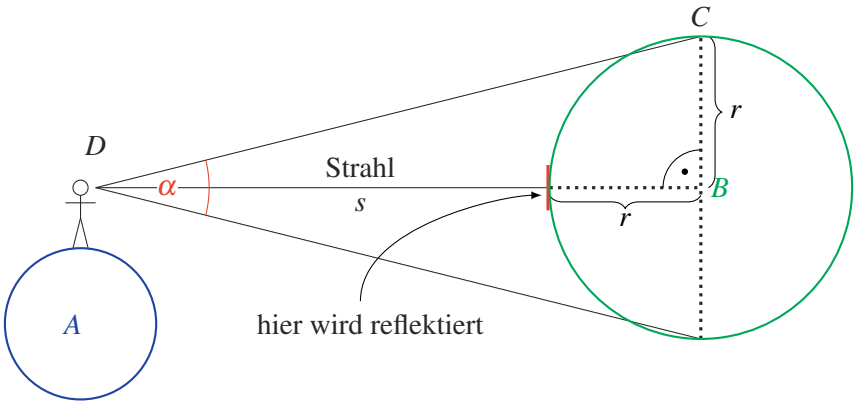


Abbildung 13.3

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.

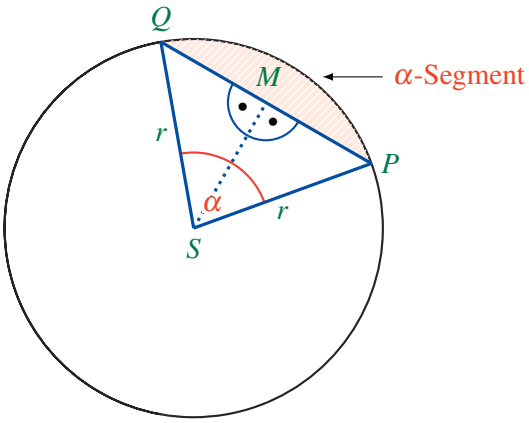


Abbildung 13.4

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 ($\angle SMP = 90°$) 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.

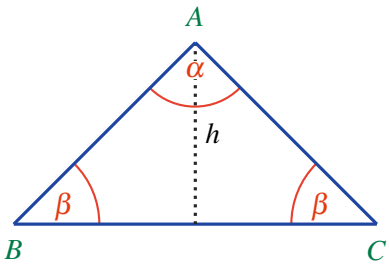


Abbildung 13.5

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 246 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.

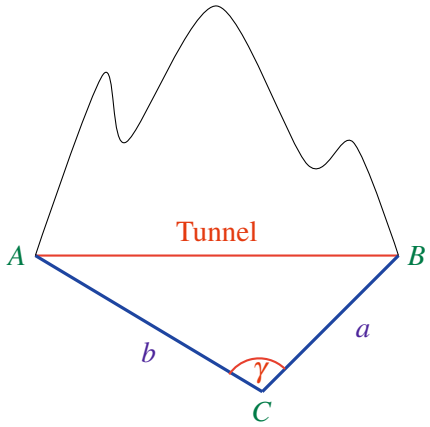


Abbildung 13.6

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.

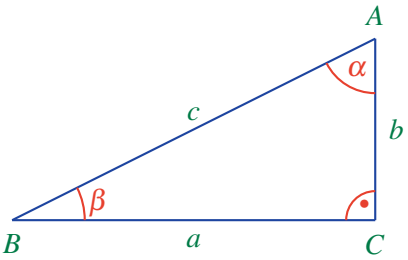


Abbildung 13.7

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 250) 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 250 betrachten wir für die Bestimmung des Radius vom Planeten B die Strecke $\overline{\text{CD}}$ 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
```

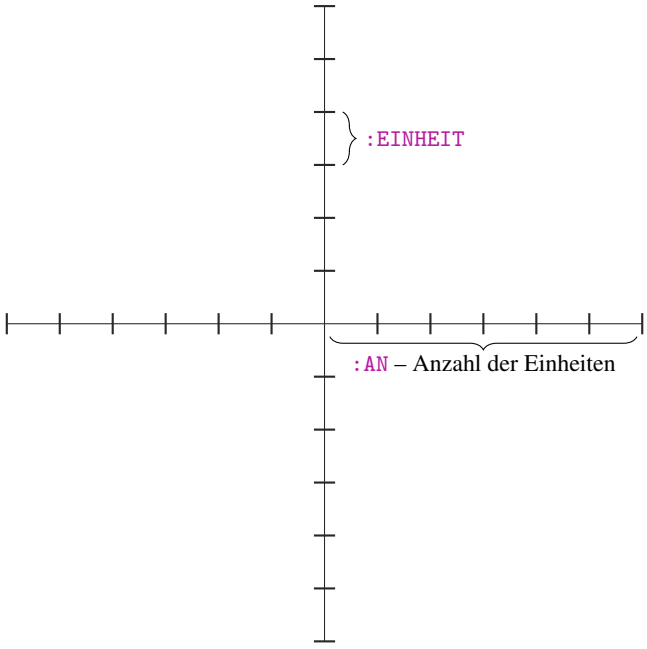


Abbildung 14.1

```
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
```

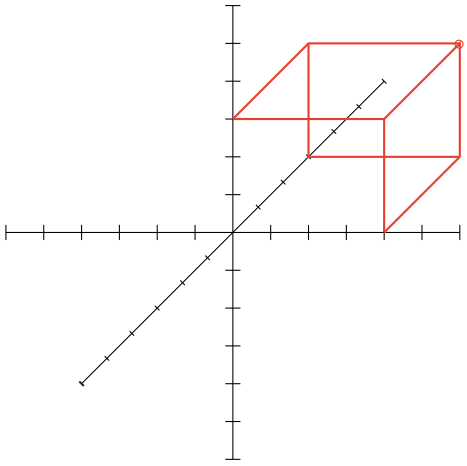


Abbildung 14.2 Punkt $(4,3,3)$ in einem dreidimensionalen Koordinatensystem

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
```

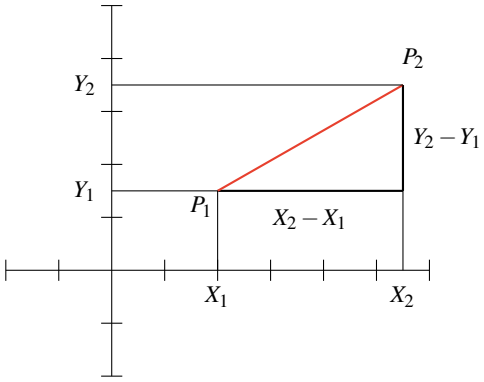


Abbildung 14.3

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.

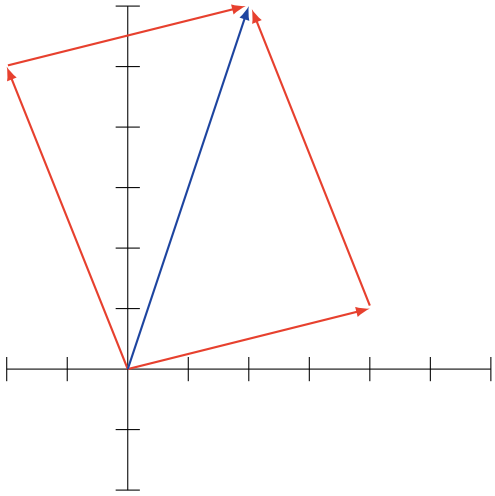


Abbildung 14.4

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.

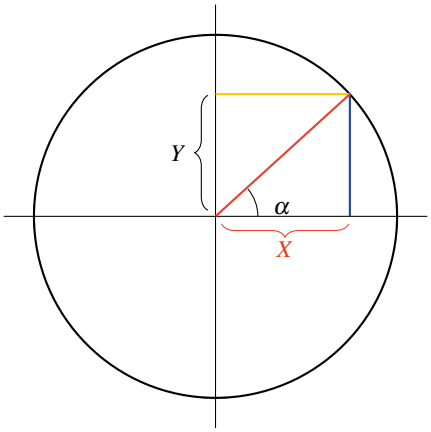


Abbildung 14.5

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 259) 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?

Sachverzeichnis

A
ABB8P3 141
abs 261
ACHSE 257
Ankathete 246
arccos 249
arcsin 249
arctan 249
Arcuscosinus 249
Arcussinus 249
Arcustangens 249
arithmetischer Ausdruck 95
ASINUS 250
AUGE 151

B
backward 19
BAUM 234
BAUM4 239
Befehlswort 18
Berechnungskomplexität 126
Beschreibungskomplexität 122
bk 20
BLATT 105
BLATT1 104
BLU1 106
BLUMEN 105
BLUMEN3 108

C
cos 247
Cosinus 247
cs 19

D
DREI90 200
DREIECKHA 248
DREIECKSSS 202
DRFLA 256

E
ECK6 118
ECK6L100 84
Effizienz 126
Eingaben 149
Eingabewerte 149
end 58
ENTF 260
EWIG 213
EWIG1 215

F
FAK 188
FAK1 189
Fakultät 188
Farbtabelle 76
fd 19
FE100 71

FELD 95
FELD1M10Q20 60
FELDABREC 231
FELDREC 219
FELDREC1 242
FELDREC2 242
FELDZEILE 124
FETT100 62
FIB 193
Fibonacci-Zahl 193
forward 18
FUN1 192

G
Gegenkathete 246
globale Parameter 113
globale Variable 160

H
Hauptprogramm 65
home 199
Hypotenuse 246

I
if 175, 183

K
Körper 41, 59
Klammerfolge 228
KLASSE1 176
KONSRECHTTR 201
Kontrollvariable 219
KOOR 258
KOPF 93
KR 160
KREIS1 78
KREIS2 117
KREIS3 78
KREISE 99
KREISE4 100
KREISMITT 198
KREISRAD 198
KREISREC 226
KREISSPIR 241
KREISSPIR1 241
KREISW 92
KURZFELD 135

L
Länge eines Programms 122
left 24
LEITER 92
LINGL 182
LINGL1 183
lokale Parameter 113
lokale Variable 161
lt 24

M
make 139
modular 58
Module 58
MUS1 101
MUS2 102
MUS3 102
MUST1 78
MUST3 78
MUSTER 136

N
NADEL 171
NULLSTELLE 211

O
optimieren (Programmlänge) 121

P
PARAL 99
PARALLEL 203
Parameter 18, 86
Parametername 86
Parameternamen 107
pd 50
pe 28
pendown 50
penerase 28
penpaint 28
penup 50
PFLANZE 171
PLANET 256
ppt 28
pr 165
print 165
Programm 17
Programmiersprache 17
Programmlänge 122
Programmname 58
Programmparameter 86
pu 50
PUNKT 258
PUNKTLIN 209
PYR 112

Q
QQQ 129
QU4 119
QUAD100 58
QUAD20 60
QUAD40 71
QUADMETH 178
QUADMETH1 180
QUADRAT 87
QUADRATW 193
Quadratwurzel 148
QUADRKR 183
QWHILE 187

R
Radiergummimodus 28
RE2ZU1 148
REC2 114
Rechnerbefehl 17
RECHT 94
RECHT1 205
REGELSCH 157
Register 87
Rekursion 213
rekursiv 213
repeat 37
right 21

S
Satz des Pythagoras 200
SCHACH4 65
SCHACH8 72
Schlüsselwort 18
Schleife 41
SCHNECKE 156
SCHNELLTR 136
SCHW100 64
SCHWDR 171
SECHS 99
setpencolor 78
sin 247
Sinus 247
SORTQ 195
SORTQ1 195
Speicher 87
Speicherinhalte 111
Speicherplatz 87
SPIR 153
SPIR6 158

SPIRBED 186
SPIREND 187
SPIRIN 188
SPIRINF 216
SPIRREC 218
SPIRRECHTREC 219
SPIRT 167
sqrt 148
SSSTEST 209
STAR2 232
STARREC 232
Stiftmodus 28
stop 178
Strahlensätze 245
STRECKE 261
STRECKE0 262
SWS 209
systematisch 57

T
T1 143
T2 144
T3 145
Tabelle für die Entwicklung der Speicherinhalte 111
tan 247
Tangens 247
TEIL 129
TEST3 167
TEST4 168
THALES 210
Tiefe eines Programmaufrufs 221
to 58
TREPP 125

U
Unterprogramm 65
UU 135

V
Variable 139
VE5 119
VEKTORADD 263
VERS 84
verschachtelt 220
Verzweigung 179
Verzweigungsstruktur 179
VIELDR 156
VIELECK 90
VIELQ 140
VIELQ1 156
VIELQ2 170
VIELQ3 170

W
WACHSE10 170
wait 214
Wandermodus 50
while 187
WURFEL 205

Z
ZEILEB 65
ZEILEA 64
Zeit(P) 128
ZICK1 135