Elementare Prinzipien der Informatik

Autor(en): **Gutknecht, J.**

Objekttyp: **Article**

Zeitschrift: **Elemente der Mathematik**

Band (Jahr): **40 (1985)**

Heft 1

PDF erstellt am: **25.09.2024**

Persistenter Link: https://doi.org/10.5169/seals-38826

Nutzungsbedingungen
Die ETH-Bibliothek ist Anbieterin der digitalisierten Zeitschriften. Sie besitzt keine Urheberrechte an den Inhalten der Zeitschriften. Die Rechte liegen in der Regel bei den Herausgebern.
Die auf der Plattform e-periodica veröffentlichten Dokumente stehen für nicht-kommerzielle Zwecke in Lehre und Forschung sowie für die private Nutzung frei zur Verfügung. Einzelne Dateien oder Ausdrucke aus diesem Angebot können zusammen mit diesen Nutzungsbedingungen und den korrekten Herkunftsbezeichnungen weitergegeben werden.
Das Veröffentlichen von Bildern in Print- und Online-Publikationen ist nur mit vorheriger Genehmigung der Rechteinhaber erlaubt. Die systematische Speicherung von Teilen des elektronischen Angebots auf anderen Servern bedarf ebenfalls des schriftlichen Einverständnisses der Rechteinhaber.

Haftungsausschluss
Alle Angaben erfolgen ohne Gewähr für Vollständigkeit oder Richtigkeit. Es wird keine Haftung übernommen für Schäden durch die Verwendung von Informationen aus diesem Online-Angebot oder durch das Fehlen von Informationen. Dies gilt auch für Inhalte Dritter, die über dieses Angebot zugänglich sind.

Ein Dienst der *ETH-Bibliothek*
ETH Zürich, Rämistrasse 101, 8092 Zürich, Schweiz, www.library.ethz.ch

http://www.e-periodica.ch

Elementare Prinzipien der Informatik

Situation

Die Informatik befindet sich zurzeit in einer bemerkenswerten Lage. Während ihre grundsätzliche Bedeutung unumstritten ist, bestehen über ihren eigentlichen Inhalt zum Teil beträchtliche Meinungsverschiedenheiten. Diese Situation tritt besonders ausgeprägt im Ausbildungsbereich zutage. Während das Fach Informatik durchwegs Eingang in die Lehrpläne der höheren Schulen findet, divergieren die Ansichten über Lehrziele und Lehrinhalte radikal.
Indessen haben sich zwei Grund-Richtungen herauskristallisiert. Die eine verficht die Stellung der Informatik als Hilfswissenschaft der Mathematik, die andere stellt sie in engen Zusammenhang mit kommerziellen und wirtschaftlichen Anwendungen. Es ist der Zweck dieses Aufsatzes zu zeigen, dass keine dieser Ansichten den Kern der Sache trifft.
Zweifellos ist die Informatik aus der Mathematik herausgewachsen. Der Begriff *Computer* für das wichtigste Instrument der Informatik ist dafür Zeuge. Computer wurden ursprünglich erdacht, um die Mathematiker von zeitaufwendigen, aber im Prinzip uninteressanten Berechnungen zu entlasten.
Andererseits dominieren heute die nicht-numerischen Computeranwendungen klar. Weshalb war eine derart weitgehende Verbreiterung des Anwendungsspektrums überhaupt möglich? Hauptsächlich dank dem genial einfachen, aber universellen Grundkonzept der «Rechenmaschinen». Tatsächlich sind Computer eher *Universalgeräte* als Maschinen im herkömmlichen Sinne. Gewöhnliche Maschinen sind a priori auf eine spezielle Anwendung zugeschnitten. Computer hingegen werden dazu entworfen, irgendwelche Daten in irgendeinem Sinne zu verarbeiten.
Ein zweites Charakteristikum der Computer ist von nicht geringerer Bedeutung für die Erschliessung neuer Horizonte: ihre Fähigkeit, Daten zu *speichern*. In vielen Anwendungen bedeutet Verarbeiten tatsächlich nichts anderes als Speichern und wieder Hervorholen einer (typischerweise enormen) Menge von Informationen.

Computer als Universalgeräte

Wenden wir uns nun den Hauptkomponenten eines Computers heute üblicher Bauart zu. Es sind im wesentlichen zwei: der sogenannte *Prozessor* und der *Speicher*. Der Prozessor beherrscht eine wohldefinierte Menge von elementaren *Instruktionen*. Der Speicher ist in eine Folge von elementaren *Zellen* gegliedert und dient dem Computer als Gedächtnis. Sowohl Prozessor als auch Speicher sind in Form von elektronischen Komponenten und Schaltungen realisiert, sie gehören zur *Hardware*.
Die Speicherzellen können durch die Ausführung von Instruktionen inspiziert und verändert werden. In einem gewissen Sinne übernehmen sie die Rolle von *Variablen*. Entscheidend ist, dass die Daten innerhalb einer Speicherzelle auf sehr verschiedene Art und Weise interpretiert werden können. Die Daten sind als Folge von *Binärziffern*

(bits) dargestellt und müssen dem jeweiligen Kontext entsprechend decodiert werden. Beispielsweise interpretieren arithmetische Instruktionen den Inhalt der betreffenden Speicherzellen als *Zahlen.*

Das zweite Beispiel, wie Daten interpretiert werden können, ist unter dem Namen *von Neumannsches Prinzip* bekannt. Es stammt von John von Neumann, einem der Pioniere der Computerwissenschaften. Dieses Prinzip hat der Kombination von Prozessor und Speicher letztlich zum Durchbruch verholfen. Ein Programm P ist eine Folge von Instruktionen, welche auf einer Menge von Daten operiert. Wenn jetzt jeder Instruktion ein Code zugeordnet ist, kann das Programm selbst, d. h. die Folge der Instruktionscodes, als *Daten* im Speicher gespeichert werden.

Von diesem Gesichtspunkt aus ist das Ausführen eines Programmes P ein *universeller* Prozess. Das zugehörige Universalprogramm U kann als eine ständige Wiederholung der Sequenz

Decodiere nächste Instruktion; Führe die Instruktion aus

formuliert werden.

Die Hardware eines Prozessors hängt offensichtlich vom Satz der elementaren Instruktionen und damit vom Universalprogramm U ab. Tatsächlich ist die Ausführung von U normalerweise von der Hardware kontrolliert. Bemerkenswert ist hingegen, dass die Hardware von nichts anderem abhängt, im besonderen nicht vom Programm P. P spielt die Rolle eines Parameters für U und kann daher durch irgendein Programm P' ersetzt werden, ohne dass eine Anpassung der Hardware nötig wäre. Programme wie P und P' werden daher als *Software* bezeichnet. Es liegt auf der Hand, dass die Universalität der Computer zum grossen Teil dem Softwarekonzept zu verdanken ist.

Greifen wir ein scheinbares Detail des Universalprogrammes auf. Was bedeutet *nächste* Instruktion? Normalerweise ist es die Instruktion in der nächsten Speicherzelle. Es ist jedoch ebensogut möglich, ja sogar unvermeidlich, dass diese Regel gelegentlich durchbrochen werden muss.

Nehmen wir zum Beispiel an, dass die nächste Instruktion vom momentanen Zustand des Programmes abhängt. In diesem Fall muss der Prozessor möglicherweise an eine neue Stelle «springen» und dort die lineare Ausführung wieder aufnehmen. Ein anderes Beispiel ist das Universalprogramm selbst. Wenn immer eine Instruktion ausgeführt worden ist, muss die Decodierung der nächsten in Angriff genommen werden, d. h. der Prozessor muss zum Beginn der «Universalschleife» zurückspringen.

Wir werden sehen, dass *Alternativen* und *Repetitionen* wesentliche Bausteine eines Programmes sind. Daher sind Sprungbefehle in der Menge der elementaren Instruktionen unentbehrlich. Sie bieten dem verarbeiteten Programm die bemerkenswerte Möglichkeit, seinen Verarbeiter zu dirigieren.

Externe Geräte vervollständigen einen Computer zu einem *Computersystem.* Wir können zwei Klassen unterscheiden: Geräte, welche die *Kommunikation* mit dem Computer ermöglichen (meistens eine Tastatur, über welche Daten eingegeben, und ein Bildschirm, auf den Meldungen und Resultate geschrieben werden können) und Geräte, die als *Massenspeicher* dienen (z. B. magnetisierbare Plattenspeicher). Die externen Geräte sind gewöhnlich direkt mit dem Computerspeicher verbunden. Daher können Daten direkt in Speicherzellen gelesen bzw. aus Speicherzellen geschrieben werden.

Programmierung

Im vorangehenden Abschnitt haben wir gesehen, dass ein Computer U durch Vorgabe eines Programmes P auf eine bestimmte Anwendung zugeschnitten werden kann, ja sogar zugeschnitten werden muss. Im Laufe der Zeit stellte sich heraus, dass die Tätigkeit des Programmierens bestimmten Gesetzen gehorcht und Gedankengänge erfordert, die weitgehend unabhängig von der speziellen, ins Auge gefassten Anwendung sind.

Als Folge davon entstanden Regeln, Methoden und Techniken, welche die Wissenschaft der Programmierung an sich begründeten. Die Programmierung im allgemeinsten Sinne hat sich geradezu zum Kern der Informationsverarbeitung oder *Informatik* entwickelt. Was also ist Programmieren?

Wir haben ein Programm bereits als eine Folge von Befehlen erklärt, die auf eine bestimmte Menge von Daten wirkt. Natürlich verbindet sich mit dem Begriff *Programm* die Vorstellung seiner *Ausführung*. Tatsächlich läuft Programmieren darauf hinaus, einen *dynamischen* Prozess als *statischen* Text zu formulieren. Die Verhältnisse werden jedoch noch komplizierter. Im allgemeinen erwartet ein Programm Eingabedaten oder Parameter, welche seine Ausführung steuern. (Das Universalprogramm U beispielsweise erwartet ein Programm P als Parameter.)

Deshalb beschreibt ein Programmtext im allgemeinen nicht nur einen, sondern eine ganze Klasse von Prozessen. Ein Programm als *korrekt* zu bezeichnen bedeutet offensichtlich, dass alle diese Prozesse korrekt ablaufen, d.h. (in endlicher Zeit) die korrekten Resultate erzeugen. Korrekte Programme zu schreiben ist mehr als nur ein edles Ziel, falls diese Programme zur Steuerung von Flugzeugen oder Atomkraftwerken vorgesehen sind.

In den meisten Fällen ist das «Auffächern» eines Programmes in alle Prozesse, die es beschreibt, hoffnungslos kompliziert. Eine vielversprechende Methode zum Beweis der Korrektheit sind Absicherungen im (statischen) Programmtext selbst. Programmieren in diesem rigorosen Sinne ist eine hochgradig mathematische Tätigkeit. Interessanterweise sind die Rollen im Laufe der Zeit vertauscht worden: die Mathematik ist zu einem Instrument der Informatik geworden.

Bevor wir diese Konzepte an Beispielen illustrieren, wollen wir kurz beim Thema Programmiernotationen verweilen. Wie wir gesehen haben, muss das Programm P letztendlich im Computerspeicher als Folge von Binärziffern vorliegen. Als Notation zur Formulierung von Programmen ist diese Darstellung aber ganz ungeeignet.

Nehmen wir an, dass jeder Elementarinstruktion ein Name zugeordnet sei, der die Wirkung der Instruktion in abgekürzter Form wiedergibt. Wenn wir überdies voraussetzen, dass jede Speicherzelle unter einem frei gewählten Namen angesprochen werden kann, so lassen sich Programme in lesbarer Form als *Text* schreiben.

Wie aber wird dieser Text in eine Folge von Binärziffern übersetzt? Natürlich gehorcht der Übersetzungsvorgang wohldefinierten Regeln. Man könnte deshalb ein Programm C entwickeln, das jeden wohlformulierten Programmtext P übersetzt. Wir nennen ein solches Programm C *Compiler*. Die Idee der Compiler hat sich als äusserst fruchtbar erwiesen.

Sie hat eine Entwicklung eingeleitet, die bis heute nicht zum Abschluss gekommen ist. Der wohl naheliegendste Schritt in dieser Entwicklung war die Einführung von Pro-

grammiernotationen, die *unabhängig* von der zugrundeliegenden Maschine sind. Diese Programmiernotationen, oder *Programmiersprachen,* wie sie üblicherweise genannt werden, eröffneten die Möglichkeit der maschinenunabhängigen oder *anwendungsorientierten* Programmierung. Im Gegensatz dazu zählt der Compiler *C* für eine solche Programmiersprache, der inhärent maschinenabhängig ist, zur *System-Software.*
Die weiteren Schritte in der Entwicklung von Programmiersprachen können am besten durch «Erhöhung des Abstraktionsniveaus» charakterisiert werden. Das Entscheidende der Abstraktion ist die Loslösung von unwesentlichen oder sogar störenden Details. In unserem Zusammenhang bedeutet dies die Möglichkeit, maschinengegebene Einheiten durch eigene «Moleküle» von verschiedener Art und Komplexität ersetzen zu können. Wir weisen an dieser Stelle auf eine zentrale Dualität zwischen den Daten hin, auf welchen die Instruktionen operieren, und den Instruktionen, die mit diesen Daten arbeiten. Erstere werden zur *Datenstruktur,* letztere zum *algorithmischen* Teil eines Programmes zusammengesetzt.
Die elementaren Speicherzellen sind die «Atome» der Datenstrukturen, die elementaren Instruktionen die «Atome» der Algorithmen. Während der Abstraktionsprozess zur Formulierung von Algorithmen sehr früh eingesetzt hat (notwendigerweise, da keine universelle Menge von elementaren Instruktionen existiert), ist der Einbezug der Datenstrukturen in diesem Abstraktionsprozess zum grossen Teil das Verdienst der Sprache *Pascal* [1]. Im speziellen hat Pascal den Begriff des *Typs* von Datenelementen eingeführt. Ein Datentyp beschreibt die *Struktur* von Dateneinheiten und steckt den Bereich ihrer möglichen Werte ab. Zahlen und Buchstaben sind Beispiele von Grundtypen, Listen und Ketten Beispiele von zusammengesetzten Strukturen.
Überdies hat durch Pascal die Methode der *strukturierten Programmierung,* d.h. die Methode der sukzessiven Verfeinerung der einzelnen Aktionen eines Prozesses, grössere Verbreitung gefunden. Damit trat ein grundsätzlich neuer Aspekt der Programmiersprachen in Erscheinung: die Programmiersprache als Werkzeug zur Konzipierung und Entwicklung, nicht nur zur Formulierung, von Programmen.
Moderne Sprachen wie *Modula-2* [2] oder *Ada* [3] gehen noch einen Schritt weiter, indem sie die Entwicklung ganzer *Programmsysteme* unterstützen. Die Grundidee ist eine klare Trennung zwischen der funktionalen *Definition* eines Programmteiles und den Methoden und Techniken, die bei der *Implementierung* zur Anwendung gelangen. In einem grossen Software-Projekt, in welchem mehrere Programmierer mitwirken, können so die *globale Struktur* und die *Schnittstellen* zwischen den einzelnen Programmteilen zentral und zum vornherein festgelegt werden.

Eine erste Serie von Beispielen

Wir haben vorher gesagt, dass der Programmierung allgemeine Gesetze und Regeln zugrundeliegen. Diese sowie die auftretenden Schwierigkeiten und Methoden zu deren Überwindung können wir am besten anhand kurzer, aber typischer Beispiele illustrieren.
Wir führen dazu eine Programmiernotation ein, die auf E.W. Dijkstra [4] zurückgeht. Zuerst definieren wir ihre «Instruktionen», in der Folge als *Anweisungen* bezeichnet. Eine *Anweisungsliste* ist eine Sequenz von Anweisungen, die durch Strichpunkte ge-

trennt sind. *Variablen* (eines für den Moment nicht näher spezifierten Typs) werden mit a_1, a_2, ..., Ausdrücke mit E_1, E_2, ..., Bedingungen mit C_1, C_2, ... und Anweisungslisten mit S_1, S_2, ... bezeichnet.

Die leere Anweisung

skip	Springe zur nächsten Anweisung

Die Wertzuweisung

$a_1, a_2, \ldots, a_n := E_1, E_2, \ldots, E_n$	Weise (gleichzeitig) den Wert von E_1 der Variablen a_1, ..., den Wert von E_n der Variablen a_n zu

Die Alternative

$\mathrm{IF}\,C_1 \to S_1$ $\mid C_2 \to S_2$ ... $\mid C_n \to S_n$ FI	Falls keine der Bedingungen erfüllt ist, Fehlerhalt sonst wähle irgendeine erfüllte Bedingung C_i und führe die Anweisungsliste S_i aus

Die Repetition

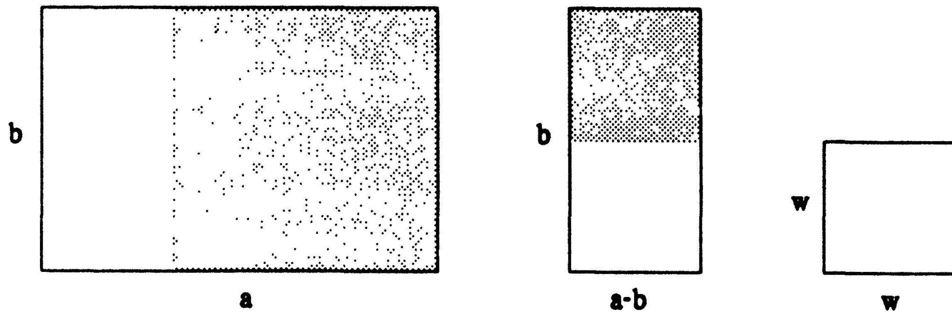
$\mathrm{DO}\,C_1 \to S_1$ $\mid C_2 \to S_2$ ... $\mid C_n \to S_n$ OD	Solange irgendeine der Bedingungen erfüllt ist, wähle eine, C_i, und führe die Anweisungsliste S_i aus

Wir bemerken, dass diese Notation weniger harmlos ist, als sie aussieht. Die Tatsache nämlich, dass innerhalb der IF- und DO-Anweisung selbst wieder Anweisungen oder sogar Anweisungslisten vorkommen dürfen, hat weitreichende Konsequenzen. Es lassen sich komplexe Hierarchien von geschachtelten Alternativen und Repetitionen konstruieren.

Die Bedingungen C_i wirken als eine Art *Wachen.* S_i kann nur ausgeführt werden, wenn die Wache C_i passiert wurde. Man nennt deshalb die Liste der C_i's und S_i's oft *bewachte Anweisung.*

In unserem ersten Programmierbeispiel wollen wir uns dem Problem zuwenden, eine rechteckige Fläche der Breite a und Höhe b (a und b natürliche Zahlen) mit möglichst grossen Quadraten auszulegen.

Nehmen wir an, dass eine der Seiten grösser als die andere sei, z. B. $a > b$. Dann ist das ursprüngliche Problem offensichtlich äquivalent damit, die reduzierte Fläche mit den Seiten $a-b$ und b mit möglichst grossen Quadraten auszulegen (s. Fig. 1). Dieser Schritt kann wiederholt werden, wobei jedesmal ein kleineres Rechteck entsteht.



Figur 1. Auslegen eines Rechteckes mit Quadraten.

Kommt der Prozess des fortgesetzten Reduzierens zu einem natürlichen Ende? Höchstens dann, wenn keine Seite des Rechtecks grösser als die andere ist, d.h. wenn ein Quadrat vorliegt. Falls zu Beginn $a > 0$ und $b > 0$ ist, muss diese Konstellation überraschenderweise stets eintreten. Bei jedem Schritt wird nämlich a oder b um eine positive ganze Zahl reduziert, und zwar unter Invarianz der Bedingung $a > 0$ und $b > 0$. (Was passiert, wenn anfängliche $a = 0$ oder $b = 0$ ist?) Die Konstruktion muss daher abbrechen. Nach dem letzten Schritt stimmt das maximale Quadrat mit dem Rechteck überein.

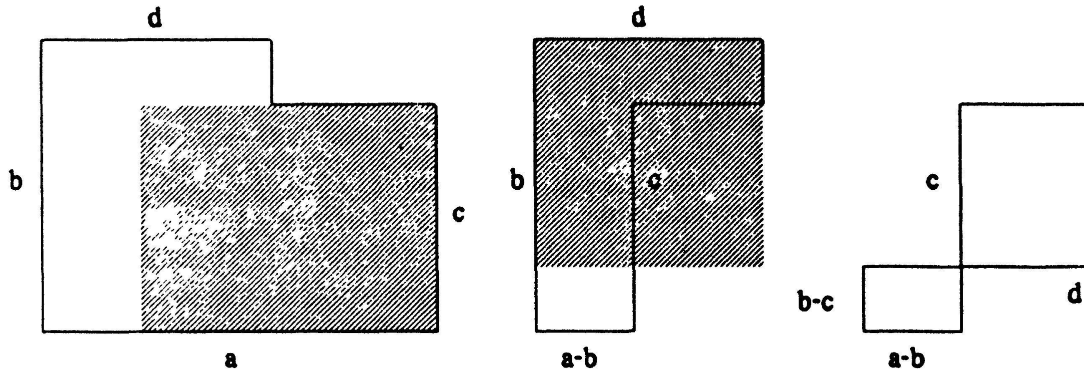
Wir sind nun in der Lage, unseren Algorithmus in der Notation von Dijkstra zu formulieren. In geschweiften Klammern fügen wir sogenannte *Zusicherungen* in Form von logischen Prädikaten hinzu.

$\{a > 0$ und $b > 0\}$
DO $a > b \rightarrow a := a - b$ $\{a > 0$ und $b > 0\}$
$|\, b > a \rightarrow b := b - a$ $\{a > 0$ und $b > 0\}$
OD
$\{a = b$, Seite des gesuchten Quadrates$\}$

Wir betonen, dass der Erfolg dieses Vorgehens entscheidend auf der Tatsache beruht, dass sich die Grösse des gesuchten Quadrates bei einem Reduktionsschritt nicht ändert. Das Prädikat «$a > 0$ und $b > 0$ und w ist Seite des gesuchten Quadrates» ist eine *Invariante* der Repetition. Falls ihre Gültigkeit beim Eintritt in die Repetition als *Eingangsbedingung* garantiert ist, so ist die *Schlussbedingung* (die das *Resultat* impliziert) die Konjunktion der Invarianten und aller Negationen der Wachen.

Als nächstes wollen wir das Auslegeproblem auf kompliziertere Gebiete, wie etwa in Figur 2 dargestellt, ausdehnen. Falls wir gemäss Figur 2 den Seiten Zahlen a, b, c, d zuordnen, können wir die vorherige Idee der sukzessiven Reduktion zur Bestimmung des maximalen Quadrates übernehmen.

Der Reduktionsprozess muss solange weitergeführt werden, als mindestens zwei der vier Zahlen verschieden sind. Wir müssen also die Wachen so anordnen, dass die Gültigkeit aller ihrer Negationen impliziert, dass die vier Werte gleich sind. Das folgende Programm zeigt eine einfache Lösung. I steht für die Invariante



Figur 2. Auslegen eines horizontal und vertikal begrenzten Gebietes mit Quadraten.

I: $a > 0$ und $b > 0$ und $c > 0$ und $d > 0$ und w ist Seite des gesuchten Quadrates

```
{I}
DO a > b → a := a − b {I}
 | b > c → b := b − c {I}
 | c > d → c := c − d {I}
 | d > a → d := d − a {I}
OD
{a = b = c = d = w, Seite des gesuchten Quadrates}
```

Es ist klar, dass die gleiche Prozedur auch für noch kompliziertere Flächen und sogar für dreidimensionale Objekte, die durch maximale Kuben auszufüllen sind, funktioniert. Der in der elementaren Zahlentheorie bewanderte Leser hat sicherlich bemerkt, dass die mathematische Lösung unseres Problems gerade der *grösste gemeinsame Teiler* (ggt) der betreffenden Zahlen ist.

Diese Interpretation liegt der nächsten Erweiterung unseres Algorithmus zugrunde. Bekanntlich kann $\mathrm{ggt}(A,B)$ stets als ganzzahlige Linearkombination von A und B dargestellt werden:

$$\mathrm{ggt}(A,B) = xA + yB$$

Die Schlüsselidee bei der Entwicklung eines Algorithmus zur Bestimmung der ganzzahligen Koeffizienten x und y ist die Erweiterung der Invarianten um zwei Gleichungen, welche am Schluss mit $ggt(A,B) = xA + yB$ übereinstimmen.

Diese invarianten Gleichungen sind $a = xA + yB$ und $b = uA + vB$, wo a und b Hilfsvariablen sind, die den Konstanten A und B entsprechen. Unser Programm lautet nun

```
a, b, x, y, u, v := A, B, 1, 0, 0, 1;
DO a > b → a := a − b; x := x − u; y := y − v
 | b > a → b := b − a; u := u − x; v := v − y
OD
```

Die bewachte Anweisung bewahrt die Gültigkeit des Prädikates

$$a > 0 \quad \text{und} \quad b > 0 \quad \text{und} \quad a = xA + yB \quad \text{und} \quad b = uA + vB,$$

welches daher eine Invariante ist. Diese und die Schlussbedingung $a = b = \mathrm{ggt}(A,B)$ implizieren, dass (x,y) und (u,v) Lösungen sind. Da wir schon gesehen haben, dass der Algorithmus nach einer endlichen Zahl von Schritten stoppt, haben wir einen eigentlichen *mathematischen Beweis* unserer Eingangsbehauptung geliefert. Im Falle $\mathrm{ggt}(A,B) = 1$ kann der Algorithmus dazu verwendet werden, ein multiplikatives Inverses modulo B von A zu bestimmen.
Mit unserem nächsten Beispiel betreten wir ein Feld, welches vor allem in der nicht-numerischen Informatik eine dominierende Rolle spielt, nämlich das des *Suchens* und Wiederfindens von Daten.
Wir beginnen mit einer äusserst einfachen Aufgabe. Gegeben sei eine *Liste* a[0], a[1], ..., a[n − 1] und ein Wert x. Die Aufgabe des zu entwickelnden Programmes bestehe darin, ein Element a[i] zu finden, dessen Wert x ist. Die naheliegendste Methode, ein solches Element zu suchen, ist das Durchkämmen der Liste, z. B. von links nach rechts, bis der Wert von a[i] mit x übereinstimmt. Wir nennen dieses Prozedere *lineares Suchen:*

$i := 0;$
$\mathrm{DO}\, a[i] \neq x \rightarrow i := i + 1\, \mathrm{OD}.$

Das Zeichen « $\neq$ » ist eine Abkürzung für «ungleich». Ferner bewirkt die Anweisung $i := i + 1$ natürlich, dass i um 1 erhöht wird. Die Schlussbestimmung ist gerade die Negation der Wache, d. h. $a[i] = x$. Unser Programm findet also sicherlich ein gewünschtes Element. Ist das Programm korrekt? Nehmen wir an, dass der Wert x überhaupt nicht in der Liste vorkommt. Dann wird die Repetitionsbedingung nie ungültig. Hingegen ist sie undefiniert, wenn $i \geq n$ ist. Der Prozess wird deshalb nach n Durchläufen mit Fehler abbrechen.
Wir können unsere Lösung retten, indem wir die Wache verstärken:

$i := 0;$
$\mathrm{DO}\, i \neq n$ and $a[i] \neq x \rightarrow i := i + 1\, \mathrm{OD}$
$\{i = n$ oder $a[i] = x\}$

Nach der de Morganschen Regel ist die Schlussbedingung nun eine Disjunktion. Falls $i = n$ ist, so tritt der Wert in der Liste nicht auf, andernfalls ist i der Index eines gewünschten Elementes. Indessen ist eine neue Schwierigkeit aufgetaucht. Im Falle $i = n$ ist der zweite Teil der Wache undefiniert. Vom Standpunkt der gewöhnlichen Logik ist in diesem Falle also die ganze Konjunktion undefiniert.
Moderne Programmiersprachen (wie Modula-2) umgehen diese Art von Schwierigkeit, indem sie den *und*- und *oder*-Operator als *bedingtes und* und *bedingtes oder* auffassen. Das *bedingte und* ordnet, falls das erste Argument den Wert *falsch* hat, unabhängig vom zweiten Argument, dem Ausdruck den Wert *falsch* zu. Dual dazu ergibt das *bedingte oder* den Wert *wahr*, falls das erste Argument den Wert *wahr* besitzt, unabhängig vom zweiten Argument. Die bedingten Operatoren sind natürlich nicht kommutativ!

Der Leser möge eine korrekte Wache für die konventionelle Logik suchen. Wir hingegen schlagen nun einen neuen Weg ein. Korrektheit ist eine wesentliche, aber nicht die einzige Qualität, die ein Programm aufweisen muss. Effizienz ist ebenso wichtig. Es ist irgendwie unbefriedigend, dass bei jedem Repetitionsschritt geprüft werden muss, ob $i \# n$ ist. Ein eleganter und typischer Ausweg besteht darin, statt des Algorithmus die *Datenstruktur* zu ändern. Falls wir ein Element a[n] hinzufügen, welches den Wert x besitzt, so schliessen wir den Fall, der alle Schwierigkeiten auslöste, endgültig aus.
So erweist sich die erste Variante unseres Algorithmus, allerdings erweitert um die Zuweisung a[n] := x, als optimal. Der Wert x tritt in der ursprünglichen Liste nicht auf, genau wenn die Repetition mit $i = n$ endet. Das Element a[n] wirkt also ebenfalls als eine Art Wache oder Hüter. Solche Elemente werden deshalb gelegentlich *Sentinel* genannt.
In den meisten Fällen ist es wünschenswert, dass der Algorithmus *alle* Elemente innerhalb der Liste findet, die den Wert des Suchargumentes besitzen. Das nächste Programm trägt diesem Anliegen Rechnung:

```
a[n] := x; i := 0;
DO i # n + 1 → DO a[i] # x → i := i + 1 OD
        {Verarbeitete a[i]} i := i + 1
OD
```

Wir nehmen nun an, dass die Elemente unserer Liste a[0], a[1], ..., a[n − 1] *Messwerte* darstellen. Wir wollen ein Programm entwickeln, das den kleinsten und den grössten dieser Messwerte findet. Konzentrieren wir uns vorerst auf den kleinsten Wert. Wir führen eine Variable *min* ein, welche das jeweils aktuelle Minimum angibt. Damit wird die Invariante der Repetition

I: min = Minimum ($a[0], \ldots, a[i-1]$).

Zusammen mit $i = n$ impliziert I das Resultat.

```
min := a[0]; i := 1; {I}
DO i # n → IF a[i] < min → min := a[i] | a[i] ≥ min → skip FI;
        i := i + 1 {I}
OD
{I und i = n}
```

Falls wir eine analoge Variable *max* einbeziehen, erhalten wir die folgende Lösung unseres Problems:

```
I: min = Minimum(a[0], ..., a[i − 1]) und max = Maximum(a[0], ..., a[i − 1])
min := a[0]; max := a[0]; i := 1; {I}
DO i # n → IF a[i] < min → min := a[i] | a[i] ≥ min → skip FI;
           IF a[i] > max → max := a[i] | a[i] ≤ max → skip FI;
           i := i + 1 {I}
OD
{I und i = n}
```

Bei diesem Algorithmus werden für jedes Listenelement zwei Vergleiche durchgeführt. Insgesamt sind also $2n$ Vergleiche erforderlich. Auf den ersten Blick geht es nicht mit weniger Vergleichen. Falls a[i] kleiner als das aktuelle Minimum ist, kann a[i] natürlich nicht gleichzeitig grösser als das aktuelle Maximum sein. Es ist jedoch wenig einträglich, diesen selten auftretenden Fall zu optimieren.
Wenn wir die Grössenbeziehung zwischen zwei Elementen a[i] und a[j] unserer Liste kennen würden, würde es genügen, das kleinere gegen *min* und das grössere gegen *max* zu testen. Da wir die Grössenbeziehung zwischen a[i] und a[j] mit einem Vergleich feststellen können, benötigen wir mit dieser Methode drei Vergleiche für jedes Paar, insgesamt also nur $3(n/2)$ Vergleiche, falls n gerade ist. Wir betrachten deshalb die Paare a[i], a[n1 − i], wobei *n1* die Konstante $n - 1$ bezeichnet. Die Invariante ist jetzt

$$\begin{aligned}&\text{I: } \min = \text{Minimum}(a[0],\ldots,a[i-1],a[n-i],\ldots,a[n-1]) \text{ und}\\ &\max = \text{Maximum}(a[0],\ldots,a[i-1],a[n-i],\ldots,a[n-1])\end{aligned}$$

Unter Berücksichtigung von I ist das Resultat etabliert, sobald $i \geqslant n - i$, d.h. wenn $i = (n + 1)$ DIV 2, wobei *DIV* die ganzzahlige Division (nicht-ganzzahliger Teil abgeschnitten) bedeutet. Wir bezeichnen ferner mit *n2* die Konstante $(n + 1)$ DIV 2. Der Gewinn an Effizienz muss mit einem Verlust an Einfachheit bezahlt werden. Wir betonen jedoch, dass das folgende Programm unabhängig von der Parität von n ist.

```
IF a[0] ⩽ a[n1]→min := a[0]; max := a[n1]
 | a[0] ⩾ a[n1]→min := a[n1]; max := a[0]
FI;
i := 1; {I}
DO i # n2→IF a[i] ⩽ a[n1 − i]→
              IF a[i] < min→min := a[i] | a[i] > min→skip FI;
              IF a[n1 − i] > max→max := a[n1 − i] | a[n1 − i] ⩽ max→skip FI;
              | a[i] ⩾ a[n1 − i]→
              IF a[n1 − i] < min→min := a[n1 − i] | a[n1 − i] ⩾ min→skip FI;
              IF a[i] > max→max := a[i] | a[i] ⩽ max→skip FI
              FI;
              i := i + 1 {I}
OD
```

Wenden wir uns als nächstes einer Variante des Problems zu, ein Element a[i] mit einem bestimmten Wert x zu finden. Wir interpretieren nun die Werte der a[i] als *Abfahrtszeiten* von Eisenbahnzügen. Die Aufgabe besteht darin, ein Programm zu entwerfen, welches die Zeit der nächsten Zugsabfahrt liefert. Der Testwert x ist hier die aktuelle Uhrzeit. Der Leser versuche die Aufgabe ohne weitere Annahmen über die Liste a[0], a[1], ..., a[n − 1] zu lösen. Das Problem wird viel einfacher, falls wir verlangen, dass die Liste geordnet sei, d.h. dass $a[0] \leqslant a[1] \leqslant \ldots \leqslant a[n-1]$ gelte.
Das gewünschte Resultat ist $a[i-1] < x \leqslant a[i]$, wobei wir ein Element a[− 1] (in Gedanken) mit dem Wert $-\infty$ und ein Element a[n] mit dem Wert ∞ hinzugefügt haben.

Wir versuchen I: a[i − 1] < x invariant zu halten. Falls wir dann die Wache als a[i] < x erklären, garantiert die Schlussbedingung das Resultat:

$$a[n] := \infty;\ i := 0; \{I\}$$
$$\mathrm{DO}\ a[i] < x \rightarrow i := i + 1\ \{I\}\ \mathrm{OD}$$
$$\{I \text{ und } x \leqslant a[i]\}$$

Es stellt sich heraus, dass der Lösungsalgorithmus nur eine Variante des oben beschriebenen linearen Suchens ist. Er löst ebenso unser ursprüngliches Problem, ein Element mit dem Wert x zu finden.

Unter der Annahme einer *geordneten* Liste a[0], a[1], ..., a[n − 1] gibt es jedoch eine Methode, die das lineare Suchen signifikant übertrifft. Sie heisst *binäres Suchen*. Die Grundidee ist das fortgesetzte Halbieren der Anzahl derjenigen Elemente, deren Wert möglicherweise mit x übereinstimmt.

Als naheliegende Invariante erweist sich eine Relation der Form a[i] < x ⩽ a[j]. Unglücklicherweise können wir jedoch diese Relation nicht verankern, da x nicht notwendigerweise im Intervall [a[0], a[n − 1]] enthalten ist. Wir ergänzen deshalb die Liste um zwei Elemente a[−1] (in Gedanken, da nicht wirklich auf das Element zugegriffen wird) und a[n], die mit $-\infty$ und ∞ initialisiert seien. Wenn wir $i := 0$ und $j := n - 1$ setzen, so etablieren wir die Eingangsbedingung I: a[i − 1] < x ⩽ a[j + 1]. Aus diesem Prädikat lässt sich zusammen mit $j - i < 0$ umittelbar das Resultat gewinnen. I und $j - i < 0$ implizieren nämlich $j + 1 = i$. Wir versuchen daher, unter Beibehaltung der Relation I, die Differenz $j - i$ sukzessive zu verkleinern:

$$i := 0;\ j := n - 1; \{I\}$$
$$\mathrm{DO}\ j - i \geqslant 0 \rightarrow m := (i + j)\ \mathrm{DIV}\ 2;$$
$$\mathrm{IF}\ a[m] < x \rightarrow i := m + 1\ \{I\} \mid a[m] \geqslant x \rightarrow j := m - 1\ \{I\}\ \mathrm{FI}$$
$$\mathrm{OD}$$
$$\{I \text{ und } j - i < 0\}$$

Die Schlussbedingung besagt, dass a[i − 1] < x ⩽ a[i] ist. Ein abschliessender Test unterscheidet die beiden Ausfälle: Wenn x = a[i] ist, dann wurde ein Element mit dem Wert x gefunden, andernfalls ist x nicht in der Liste enthalten, und die Werte der Listenelemente a[i − 1] und a[i] sind am nächsten bei x.

Untersuchen wir nun die Termination unseres Algorithmus. Wenn $j \geqslant i$, dann ist $m \geqslant i$ und $m \leqslant j$, d.h. $j - (m + 1) < j - i$. Also wird $j - i$ tatsächlich bei jedem Repetitionsschritt echt verkleinert.

Was ist der Gewinn, wenn das binäre anstelle des linearen Suchens eingesetzt wird? Lineares Suchen in einer Liste von n Elementen erfordert im Mittel $n/2$ Vergleiche. Falls unsere Variante des binären Suchens verwendet wird, so werden stets $\log n$ Vergleiche benötigt (log bezeichne den Logarithmus zur Basis 2). Wir haben dabei angenommen, dass n eine Zweierpotenz sei. Um den Gewinn zu veranschaulichen, erinnern wir daran, dass für $n = 1024$ $n/2 = 512$ und $\log n = 10$ ist.

Anhand der beiden Beispiele des linearen und des binären Suchens haben wir zu zeigen versucht, dass Programmieren eine *zielorientierte* Aktivität ist. Das Resultat legte jeweils die Invariante und die Wache nahe. Die Invariante ihrerseits und das Bestreben,

dem Ziel näher zu kommen, bestimmte praktisch den Repetitionsschritt, ausser (im binären Suchen) die Definition von m. Tatsächlich bleibt der Algorithmus korrekt und endet nach einer endlichen Zahl von Schritten, wenn nur zugesichert ist, dass im Falle $j \geqslant i$ die Beziehungen $m \geqslant i$ und $m \leqslant j$ garantiert sind.
Zielgerichtetes Vorgehen wird sich auch im nächsten Beispiel auszahlen. Es stammt von D. Gries [5] und heisst der *Wohlfahrtsschwindler*. Im Gegensatz zu den bisherigen Suchproblemen, ist das Suchargument hier nicht explizit bekannt. Drei geordnete Listen a[0], a[1], ..., a[$n-1$], b[0], b[1], ..., b[$m-1$] und c[0], c[1], ..., c[$l-1$] sind nun im Spiel. Die erste enthält die Namen aller Studenten der New York University, die zweite die Namen aller Angestellten von IBM New York und die dritte die Namen aller Wohlfahrtsbezüger von New York. Die Aufgabe besteht darin, ein Programm zu schreiben, welches eine Person sucht, die auf allen drei Listen registriert ist.
Die Listenelemente sind nun Ketten von Zeichen statt Zahlen. Wir nehmen an, dass sie *lexikographisch* geordnet sind. Die Schlussbedingung muss von der Form a[i] = b[j] = c[k] für geeignete Indexwerte i, j und k sein. Die Wachen definieren wir nach einem Schema, welches sich in einem früheren Beispiel bewährt hat. Ausserdem führen wir drei Sentinels a[n], b[m] und c[l] ein:

```
a[n], b[m], c[l], i, j, k := ∞, ∞, ∞, 0, 0, 0;
DO a[i] > b[j] → i := j + 1
  | b[j] > c[k] → k := k + 1
  | c[k] > a[i] → i := i + 1
OD
```

Die Schlussbedingung a[i] ⩽ b[j] ⩽ c[k] ⩽ a[i] liefert das Resultat. Die Invariante ist hier leer, falls man nicht die eher technische Bedingung $0 \leqslant i \leqslant n$ und $0 \leqslant j \leqslant m$ und $0 \leqslant k \leqslant l$ als Invariante betrachtet. Der Algorithmus terminiert nach höchstens $n + m + l$ Repetitionsschriften, da bei jedem Schritt genau ein Index erhöht wird.
(Fortsetzung im nächsten Heft)

J. Gutknecht, Institut für Informatik, ETH Zürich

© 1985 Birkhäuser Verlag, Basel 0013-6018/85/060011-12$1.50 + 0.20/0

Kleine Mitteilungen

Über einen Wert, der zwischen dem geometrischen und dem arithmetischen Mittel zweier Zahlen liegt

In dieser kleinen Mitteilung soll gezeigt werden, dass für positive reelle Zahlen a und b (mit $a < b$) der Wert $(e/a)^a(b/e)^b$ zwischen der $(b-a)$-ten Potenz des geometrischen und des arithmetischen Mittels von a und b liegt. (Mit e wird wie üblich die Eulersche Zahl bezeichnet.)