R. L. Baber

Fehlerfreie Programmierung für den Software-Zauberlehrling

Oldenbourg

Baber Fehlerfreie Programmierung
Oldenbourg

Mit der zunehmenden Anwendung von Computersystemen in allen Bereichen unserer Gesellschaft werden die Konsequenzen von Software-Fehlern immer ernster und kostspieliger.
So zauberhaft und unglaublich es manchen klingen mag: Es ist jetzt möglich zu beweisen, daß ein Programm fehlerfrei ist. Das Geheimnis liegt jedoch nicht in der Zauberei, sondern in relativ einfacher Mathematik und Logik.
Dieses Buch zeigt dem Software-Entwickler, wie er diese Möglichkeit praktisch nützen kann, sowohl bei der Verifikation eines vorliegenden Programmentwurfs als auch beim Schreiben eines neuen – nachweislich fehlerfreien – Programms.

R. Oldenbourg Verlag

ISBN 3-486-21637-6

Verifikation einer Korrektheitsaussage — Bestimmung einer Vorbedingung

Übersichtskarte über die praktische Anwendung der Beweisregeln

Anweisungsart	Korrektheitsaussage verifizieren	Vorbedingung bestimmen
Zuweisung	{V} x:=A {P} falls $V \Longrightarrow P^{x}_{(A)}$ [Z2]	$\{P^{x}_{(A)}\}$ x:=A {P} [Z1]
if-Anweisung	{V} **if** B **then** S1 **else** S2 **endif** {P} falls {V **und** B} S1 {P} und {V **und nicht** B} S2 {P} [IF1]	Ermittele V1 und V2 derart, daß {V1} S1 {P} und {V2} S2 {P} dann gilt, daß {(V1 **und** B) **oder** (V2 **und nicht** B)} **if** B **then** S1 **else** S2 **endif** {P} [IF2]
Folge	{V} S1; S2 {P} falls {V} S1 {P1} und {P1} S2 {P} [F1]	Ermittele P1 und V derart, daß {V} S1 {P1} und {P1} S2 {P} dann gilt, daß {V} S1; S2 {P} [F1]
while-Schleife mit Initialisierung	{V} Init.; **while** B **do** S **endwhile** {P} falls {V} Init. {I} und {I **und** B} S {I} und (I **und nicht** B) $\Longrightarrow$ P [W2]	Ermittele I und V derart, daß {V} Init. {I} und {I **und** B} S {I} und (I **und nicht** B) $\Longrightarrow$ P [W2] dann gilt, daß {V} Init.; **while** B **do** S **endwhile** {P}

- Wähle die Zeile je nach Art der Anweisung bzw. Zusammensetzung von Anweisungen aus.
- Wähle die Spalte je nach Aufgabe (Korrektheitsaussage verifizieren oder Vorbedingung bestimmen) aus.
- Die dadurch bestimmte Zelle faßt die Anwendung der geeigneten in [] angegebenen Beweisregel(n) zusammen.

Diese Übersichtskarte stammt aus dem Buch *Fehlerfreie Programmierung für den Software-Zauberlehrling* von Robert L. Baber. (Siehe Abschnitt 3.4.)

Verifikation einer Korrektheitsaussage — Bestimmung einer Vorbedingung

Übersichtskarte über die praktische Anwendung der Beweisregeln

Anweisungsart	Korrektheitsaussage verifizieren	Vorbedingung bestimmen
while-Schleife ohne Initialisierung	{V} **while** B **do** S **endwhile** {P} falls V ⟹ I und {I **und** B} S {I} und (I **und nicht** B) ⟹ P [W1+B1]	Ermittele I derart, daß {I **und** B} S {I} und (I **und nicht** B) ⟹ P dann gilt, daß {I} **while** B **do** S **endwhile** {P} [W1+B1]
Programmteil oder Unterprogramm	Teile die Vor- und Nachbedingungen auf {V **und** B} S {P **und** B} falls {V} S {P} und {B} S {B} [U2]	Teile die Nachbedingung in P **und** B derart auf, daß {V} S {P} und {B} S {B} dann gilt, daß {V **und** B} S {P **und** B} [U2]
"Divide and conquer" (teile und erobere)	{V} S {P1 **und** P2} falls {V} S {P1} und {V} S {P2} [DC3]	Zerlege die Nachbedingung in P1 **und** P2 Ermittele V1 und V2 derart, daß {V1} S {P1} und {V2} S {P2} dann gilt, daß {V1 **und** V2} S {P1 **und** P2} [DC1]

- Wähle die Zeile je nach Art der Anweisung bzw. Zusammensetzung von Anweisungen aus.
- Wähle die Spalte je nach Aufgabe (Korrektheitsaussage verifizieren oder Vorbedingung bestimmen) aus.
- Die dadurch bestimmte Zelle faßt die Anwendung der geeigneten in [] angegebenen Beweisregel(n) zusammen.

Diese Übersichtskarte stammt aus dem Buch *Fehlerfreie Programmierung für den Software-Zauberlehrling* von Robert L. Baber. (Siehe Abschnitt 3.4.)

Fehlerfreie Programmierung für den Software-Zauberlehrling

von
Robert Laurence Baber

R. Oldenbourg Verlag München Wien 1990

CIP-Titelaufnahme der Deutschen Bibliothek

Baber, Robert L.:
Fehlerfreie Programmierung für den Software-Zauberlehrling /
von Robert L. Baber. – München ; Wien : Oldenbourg, 1990
ISBN 3-486-21637-6

Gesamtherstellung: R. Oldenbourg Graphische Betriebe GmbH, München

ISBN 3-486-21637-6

Inhalt

Die wichtigsten Beweisregeln:

Vorwort

Dieses Buch ist das Ergebnis eines mehrstufigen Entwicklungsprozesses, der sich über einige Jahre erstreckt hat. Der Inhalt basiert auf meinem Seminar für in der Praxis stehende Software-Entwickler, das ich international unterrichte, und auf meiner Vorlesung, die ich als Lehrbeauftragter im Fachbereich Informatik an der Johann Wolfgang Goethe-Universität in Frankfurt/Main halte. Diesen wiederum liegen meine eigenen Erfahrungen mit der praktischen Anwendung dieses Stoffs zugrunde.

Die Fragen meiner Seminarteilnehmer und Studenten haben wesentlich zur Organisation und Struktur dieses Buchs beigetragen. So hat auch die Skepsis über die praktische Anwendbarkeit, die besonders von erfahrenen Software-Entwicklern und ihren Managern gelegentlich geäußert wurde, Inhalt und Gliederung dieses Werks beeinflußt.

Am wichtigsten jedoch waren die sehr positiven Reaktionen meiner Seminarteilnehmer und Studenten. Sie haben mich beim Versuch motiviert, dieses Fachgebiet für den Softwareentwicklungs-Praktiker noch verständlicher und leichter zugänglich zu gestalten. Die bei meinen Seminaren und Vorträgen immer wieder aufkommende Begeisterung für diesen Stoff und für die Möglichkeiten, die seine konsequente praktische Anwendung eröffnet, ist meines Erachtens ein sicheres Zeichen dafür, daß dieser Weg zu zuverlässiger Software sowohl praktisch gangbar als auch bitter notwendig ist. Ein bedeutender Anteil unserer Software-Entwickler – insbesondere der jüngeren – ist bereits von dem Potential dieses Teilgebiets der Software-Entwicklung überzeugt. Ihre wachsende Zahl wird in der nicht allzu fernen Zukunft ausreichen, um eine grundlegende Wende in der Software-Entwicklungspraxis zu bewirken.

An dieser Stelle möchte ich allen, die unmittelbar oder mittelbar, bewußt oder unbewußt zu diesem Buch beigetragen haben – meinen Seminarteilnehmern, Studenten, Beratungsklienten, Fachkollegen usw. – für ihre Hilfe und Unterstützung herzlich danken. Für die Zeichnung des Zauberers, die an einigen Stellen dieses Buchs erscheint, bin ich Frau José Zwakman sehr verbunden. Vor allem danke ich Drs. Willem Dijkhuis für seinen wertvollen Rat zu allen meinen größeren schriftstellerischen Vorhaben, der u.a. zum Titel des vorliegenden Buchs geführt hat.

Bad Homburg, April 1990

Robert Laurence Baber

Bezeichnungen

nicht	Logische (Boolesche) Funktion. Siehe Anhang A, Abschnitt A.1.
oder	Logische (Boolesche) Funktion. Siehe Anhang A, Abschnitt A.1.
und	Logische (Boolesche) Funktion. Siehe Anhang A, Abschnitt A.1.
$\Longrightarrow$	Logische (Boolesche) Funktion. Siehe Anhang A, Abschnitt A.1.
$\mathbf{oder}_{i=1}^{n}$	Die **oder**-Reihe. Siehe Anhang A, Abschnitt A.5.
$\mathbf{und}_{i=1}^{n}$	Die **und**-Reihe. Siehe Anhang A, Abschnitt A.5.
∎	Markiert das Ende eines Beispiels, der Definition einer Beweisregel usw.
'	Kennzeichnet den Wert einer Variablen oder eines Ausdrucks *vor* der Ausführung einer Anweisung. Siehe z.B. die Abschnitte 2.2 und 2.3.
"	Kennzeichnet den Wert einer Variablen oder eines Ausdrucks *nach* der Ausführung einer Anweisung. Siehe z.B. die Abschnitte 2.2 und 2.3.
{V} S {P}	Bedeutet: "V ist eine Vorbedingung der Nachbedingung P bezüglich der Anweisung S". Siehe Abschnitt 3.1.
P^{x}_{A}	Der Ausdruck, der sich daraus ergibt, daß man überall im Ausdruck P die Variable x durch den Ausdruck (A) ersetzt. Siehe Abschnitt 3.3.2.
Z	Die Menge aller Ganzzahlen (0, 1, -1, 2, -2, ...). Siehe Abschnitt 4.4.1.

ϵ	Bedeutet: "ist ein Element aus der Menge". Die Aussage "nϵZ" z.B. bedeutet "(der Wert von) n ist ein Element aus der Menge aller Ganzzahlen" oder, kürzer, "der Wert der Variable n ist eine Ganzzahl". Siehe Abschnitt 4.4.1.
$\|x\|$	Der Betrag von x. Ist x negativ, dann ist $\|x\|$ gleich -x. Ist x Null oder positiv, ist $\|x\|$ gleich x.

0. Die Zauberlehrlinge im Lande Ret Up Moc

Um 2500 v. Chr. war das Land Ret Up Moc eine hoch entwickelte Gesellschaft in der Wiege der Zivilisation, dem Mittleren Osten. Es gab einige wichtige Städte und der Innen- und Außenhandel blühte. Es hatte sich eine bedeutungsvolle Bauindustrie entwickelt, in der fachlich qualifizierte Baukünstler und Bauplaner eine wichtige Rolle spielten.

Zwischen 2500 und 2400 v. Chr. wurde ein zukunftsträchtiger Fortschritt erzielt. Plötzlich und unerwartet erarbeiteten einige Gelehrte eine neue wissenschaftliche Basis für die Bauplanung, insbesondere für das, was wir heute die Baustatik nennen. Leiter dieser Gruppe von Gelehrten war Akado, ein bekannter und führender Fachmann auf dem Gebiet des Bauwesens. Seine neue Methode ermöglichte es nicht nur, viel größere Bauwerke zu bauen, sondern darüber hinaus mit weit weniger Baumaterial auszukommen, was wiederum die Baukosten ganz erheblich verringerte. Diese Kostensenkung führte natürlich zu einer entsprechenden Erhöhung der Nachfrage für neue Gebäude aller Art.

Der Bedarf an Bauplänen für neue Bauwerke stieg dermaßen an, daß die bereits etablierten und fachlich qualifizierten Bauplaner, die sich die Zeit genommen hatten, die neue Methode zu erlernen, nicht damit Schritt halten konnten. Die Anzahl der neuen Absolventen der Baufachschulen reichte auch nicht aus, um die bauplanerische Kapazität auf die benötigte Höhe zu bringen. Der Baustoffhandel – nicht bereit, auf einen so interessanten Marktzuwachs zu verzichten – erfand eine "Lösung": in Schnellkursen bildete er Poliere, Bauarbeiter usw. soweit aus, daß sie Baupläne und Zeichnungen erstellen konnten. Die neue wissenschaftliche Basis für die Baustatik verstanden sie zwar nicht so recht, aber solange sie sich an die in den Kursen vermittelten, auswendig gelernten Regeln hielten, verlief alles mehr oder weniger akzeptabel. Etwa ein Drittel aller angefangenen Bauwerke stürzten während des Baus oder gleich danach ein. Der dadurch verursachte Verlust war jedoch immer noch deutlich geringer als die Einsparungen, die der Einsatz der neuen

Methode ermöglichte. Unter dem Strich lohnte sich also diese Vorgehensweise. [Baber, 1982, 1986, 1987, Kapitel 0]

Nach ein paar Jahrzehnten pendelte sich ein Gleichgewicht auf dem Baumarkt ein. Die meisten Bauplaner wurden durch die kurzen, vom Bauhandel veranstalteten Kurse auf ihre praktische Tätigkeit vorbereitet und waren entsprechend unterqualifiziert. Ihnen war diese Situation recht, weil sie auf diese Weise relativ schnell anfangen konnten, gutes Geld zu verdienen. Den regierenden Kreisen war sie auch recht, denn die Kosten für die Ausbildung der Bauplaner waren dadurch minimal und wurden darüber hinaus weitgehend von der Baustoffindustrie getragen. Den Bauherren war diese Situation auch mehr oder minder recht, denn dadurch bekamen sie größere und billigere Gebäude als auf die herkömmliche Weise. Nur Akado und seine Fachkollegen waren mit diesem Zustand nicht zufrieden. Sie wußten, daß sämtliche Einstürze vermeidbar waren und daß die dazu nötige Investition – eine professionelle Ausbildung der Bauplaner – auf Dauer und aus übergeordneter Sicht gesehen billiger wäre als die Verluste, die die vielen Einstürze verursachten. Kurz zusammengefaßt, so Akado, sei die gegenwärtige Vorgehensweise zwar insgesamt günstiger als die alte, aber eine professionelle Ausübung der bauplanerischen Tätigkeit – basierend auf einer entsprechend anspruchsvollen Ausbildung der Bauplaner – würde zu einer noch viel besseren Situation führen.

Auch Akado und seine Gruppe von Gelehrten entwarfen und planten neue Bauvorhaben. Im Gegensatz zur inzwischen normalen Bauerfahrung stürzten ihre Bauwerke nie ein. Manchmal wurden sie gefragt, warum das so sei. Sie erläuterten die Basis ihrer Berechnungsmethode, aber die anderen Bauplaner konnten oder wollten die Erklärungen nicht verstehen. Fast alle Bauherren glaubten, daß der Erfolg Akados allein auf Zufall und Glück zurückzuführen, also gar nicht erklärbar sei. Weil er ein etwas höheres Honorar verlangte als die weniger qualifizierten Bauplaner, wurden seine besseren Dienste nicht allzu häufig in Anspruch genommen.

Enttäuscht sprach Akado gelegentlich darüber mit seinem besten Freund, Naram, einem Schauspieler und Amateur-Psychologen. Naram fand das Verhalten der Bauherren durchaus verständlich, wenn auch völlig irrational, und versuchte, es Akado zu erklären. Die Bauherren waren ganz normale Leute, nicht Intellektuelle, und wie fast alle anderen Leute im Lande Ret Up Moc sehr religiös, sogar abergläubisch.

Rationale wissenschaftliche Erklärungen waren ihnen unverständlich und verdächtig. Deshalb hielten sie eine darauf basierende Bauplanung für praxisfremd. Bei der Vorgehensweise Akados, meinte Naram, mangele es offensichtlich an abergläubischer Zauberei. Naram verfaßte für Akado eine Zaubervorführung mit allen möglichen und unmöglichen mysteriösen Zaubersprüchen, geheimnisvoll klingenden Zauberliedern, hexenartigen Tänzen usw. und führte Regie bei den Proben mit Akado. Nachdem Naram mit der zauberschauspielerischen Leistung Akados zufrieden war, schickte er Akado auf die Bauherren los.

Beim Bau seines nächsten Gebäudes, eines besonders großen Verwaltungsbüros für die königliche Regierung, fand die Uraufführung der Zauberdarbietung Akados statt. Der Erfolg war überwältigend und die Bauherren waren so beeindruckt, daß Akado nicht nur viele neue Aufträge gewann, sondern auch den Neid einiger führender Priester. Beim Bau jedes Objekts, das Akado nachher entwarf, wiederholte er diese Zaubervorführung. Allmählich entwickelte er sie mit Narams Hilfe zu einer immer dramatischeren und beeindruckenderen Zaubervorstellung weiter.

Zauberei konnte zwar keiner so richtig verstehen, aber jeder glaubte daran, weil es ein gewisser Bestandteil der primitiven Moc'schen Religion, Denkweise und Kultur war. Jedesmal sprach Akado die Zauberworte für einen Bauerfolg sehr eindrucksvoll aus. Und jedesmal – ohne Ausnahme – hielt das Bauwerk stand. Die Zauberei funktionierte doch; so einleuchtend und verständlich war die Erklärung dafür, daß das Bauwerk nicht einstürzte.

Zuerst hatte Akado ein etwas schlechtes Gewissen, weil er wußte, daß die Zauberei nichts mit dem Bauerfolg zu tun hatte. Sie erhöhte nur seinen Zeitaufwand und seine Kosten, wenn auch nur geringfügig. Aber er sah ein, daß sein Freund Naram recht hatte. Die Bauherren wollten keine professionelle Baukunst, sie wollten Bauzauberei. Sie wollten daran glauben. Natürlich wollten sie auch vernünftige Pläne für Bauwerke. Akado versorgte sie mit beidem gleichzeitig: Zauberei und hochwertigen Bauplänen. Das war kein Betrug; er "verkaufte" genau die Kombination, die seine Kunden "kaufen" wollten. Die Moc'sche Gesellschaft brauchte eine Bauingenieurwissenschaft und wollte Bauzauberei. Akado gab ihr beides.

Akado wurde sehr berühmt als ein Meister der Bauzauberei. So wie es immer ist mit einem Zaubermeister, fanden

Zauberlehrlinge schnell ihren Weg zu ihm. Er nahm die besten von ihnen auf und gründete eine Baukunst-Zauberschule. Natürlich weihte er seine Baukunst-Zauberlehrlinge in das echte Geheimnis der "Bauzauberei" ein – denn sonst würden sie nach ihrer Ausbildung keinen Erfolg haben und der ganze Schwindel würde auffliegen. Ein solches Mißgeschick würde zum Leid aller führen – Akado, seiner Baukunst-Zauberlehrlinge und, nicht zuletzt, der Bauherren, die auch erhebliche Vorteile aus dem Erfolg der Vorgehensweise Akados zogen.

Die Anzahl der Baukunst-Zauberlehrlinge an der Baukunst-Zauberschule Akados wuchs sehr schnell. Um den Unterricht effizienter zu gestalten, verfaßte Akado ein Lehrbuch. Sein Buch, *Einsturzfreie Statik für den Baukunst-Zauberlehrling*, hielt sowohl die wissenschaftliche als auch die zauberkünstlerische Basis der Bauzauberei fest und vermittelte sie an seine Schüler weiter. Auch viele spätere Generationen von Baukunst-Zauberern haben die Grundlage ihres Fachs aus diesem Buch gelernt.

2400 v. Chr.

↓

1990 n. Chr.

Das vorliegende Buch entspricht hinsichtlich Inhalt und Gliederung dem alten Lehrtext von Akado für seine Baukunst-Zauberlehrlinge – bis auf seine Kapitel über Zaubersprüche und die Aufführung von Zaubervorstellungen. Denn in unserem modernen Zeitalter müßten diese Aspekte der professionellen Software-Entwicklungspraxis überflüssig sein, auch wenn die sich daraus ergebende Fehlerfreiheit nach bisherigen Maßstäben manchen als unglaublich zauberhaft vorkommen mag.

1. Einführung

1.1 Das Problem: fehlerhafte Software

Die Erstellung von Computersoftware ist immer noch durch eine unbefriedigend hohe Fehlerrate gekennzeichnet. Auch wenn der überwiegende Anteil der ursprünglich vorhandenen Fehler vor der Übergabe der Software an den Benutzer gefunden und korrigiert wird, verursachen die übriggebliebenen Fehler noch lange nach der Einführung Störungen, Ärger, Frustration und unnötigen Aufwand. Über große durch Softwarefehler verursachte Verluste wird immer wieder berichtet. Sogar menschliche Todesfälle sind bereits auf Softwarefehler zurückgeführt worden: Computergesteuerte medizinische Geräte haben tödliche Dosen von Strahlung [Joyce, 1987], [*IEEE Spectrum*, 1987] und von einer Injektionslösung [Thomas, 1988, Seite 9] verabreicht.

Der Einsatz von Rechnersystemen ist bereits so umfassend geworden, daß Teile unseres gesellschaftlichen Lebens völlig abhängig davon sind. Die Abwicklung des größten Teils der täglichen Transaktionen unserer heutigen Wirtschaft ist ohne Rechnersysteme nicht mehr denkbar. Auch für sicherheitskritische Tätigkeiten liefern Rechnersysteme wichtige Unterstützung (z.B. für die Flugsicherung). Der Einsatz von Rechnersystemen wird in den nächsten Jahren weiterhin deutlich zunehmen, was zu höheren Anforderungen hinsichtlich Verläßlichkeit führen wird. Mit der zunehmenden Anwendung von Computersystemen in allen Bereichen unserer Gesellschaft werden die Konsequenzen von Fehlern ("Bugs") in der Software immer ernster und kostspieliger. Entweder müssen wir Software-Entwickler *sehr* viel höheren Anforderungen hinsichtlich der Fehlerfreiheit unserer Programme genügen oder es werden die Grenzen der aus gesellschaftlicher Sicht zu verantwortenden Anwendung der Computertechnologie erreicht.

Wir müssen unter anderem, vor allem uns selbst gegenüber, offen zugeben, daß Softwarefehler vermeidbare Entwurfsfehler – menschliche Fehler seitens des Software-Entwicklers – sind. Sie liegen *nicht* in der Natur der Sache begründet.

1.2 Die Lösung: die klassischen Ingenieurwissenschaften als Vorbilder

Auf den klassischen technischen Gebieten konstruiert der Ingenieur Gegenstände, Geräte, Systeme usw., die eine hohe Verläßlichkeit aufweisen. Viele dieser Produkte und Systeme sind für das gewohnte Funktionieren unserer Gesellschaft wichtige, sogar unerläßliche Voraussetzungen (z.B. Bauwerke aller Art, Wasser- und Elektrizitätsversorgung, das Fernsprechnetz, Fahrzeuge, Schiffe und Flugzeuge, die Herstellung von Chemikalien und Arzneimitteln und vieles mehr). Viele dieser Produkte und Prozesse verbergen in sich große potentielle Gefahren, sowohl wirtschaftlicher als auch lebensbedrohlicher Art. Wir können uns jedoch weitgehend auf diese Systeme verlassen, denn die Ingenieure, die sie konstruieren, sind in der Lage, fehlerfreie Entwürfe zu erarbeiten und vor allem ihre Konstruktionen nach Übereinstimmung mit den gestellten Anforderungen (Spezifikationen, Pflichtenheft) analytisch zu prüfen – *bevor* das Objekt gebaut wird.

In den letzten ein bis zwei Jahrzehnten ist eine Grundlage für die Konstruktion beweisbar korrekter Software erarbeitet worden. Sie ist direkt vergleichbar mit den theoretischen Grundlagen der klassischen Ingenieurwissenschaften und ermöglicht gleichartige Ergebnisse hinsichtlich Qualität, Zuverlässigkeit und Freiheit von Entwurfsfehlern. Z.B. kann der Software-Ingenieur damit – *vor* dem ersten Lauf – beweisen, daß sein Programm die gestellten Spezifikationen (das Pflichtenheft) erfüllt; genau wie der Bauingenieur in seinem Antrag auf die Erteilung einer Baugenehmigung – also *bevor* das Gebäude gebaut wird – rechnerisch belegt, daß der von ihm geplante Bau sich selbst und die beabsichtigte Last tragen wird.

Es ist nicht schwierig, die Programm-Korrektheitsbeweisführung und ihre praktische Anwendung zu erlernen, jedoch auch nicht trivial leicht. Zum Erlernen dieser Grundlage und zur Entwicklung der Fähigkeit, sie in der Praxis anwenden zu können, sind ein gewisser zeitlicher und geistiger Aufwand und vor allem der Wille, eine wirklich professionelle Vorgehensweise einzuschlagen, erforderlich. Danach, so berichten Software-Ingenieure, die mit der Anwendung dieser Grundlage einschlägige praktische Erfahrung gesammelt haben, verringert sich der Entwicklungszeitaufwand insgesamt.

Wo soll die Korrektheitsbeweisführung angesetzt werden? Damit vertraute Software-Entwickler sind sich darüber einig, daß ihr Einsatz bereits *vor und während* der Konstruktion am sinnvollsten – d.h. am nützlichsten und am produktivsten – ist. Den Ingenieur wird diese Aussage nicht überraschen, denn der Elektroingenieur z.B. wendet seine theoretische Grundlage bereits bei der Konzeption und Planung seiner Netzwerke, nicht erst nach deren Fertigstellung, an. Ebenfalls wendet der Bauingenieur seine theoretische Grundlage (Statik) bereits während der Entwurfsphase, nicht erst nach Fertigstellung aller Baupläne oder sogar erst nach Fertigstellung des Baus an.

Fängt man nämlich erst dann an, Methoden der Korrektheitsbeweisführung anzuwenden, nachdem das Programm fertig geschrieben worden ist, können verschiedene Schwierigkeiten auftreten. Erstens kann es sich als unmöglich erweisen, die Korrektheit des gegebenen Programms zu beweisen, einfach weil es gar nicht korrekt ist. Zweitens – falls der Beweis gelingt – kann sich der Beweis als unnötig logisch komplex herausstellen, weil das Programm ebenfalls unnötig logisch komplex ist. Drittens müssen gewisse Konstruktionsentscheidungen in geeigneter Form vorliegen, wenn man den Beweis erstellt (dazu gehören vor allem Schleifeninvarianten sowie Vor- und Nachbedingungen aller aufgerufenen Unterprogramme). Liegen diese nicht in geeigneter Form vor, müssen während der Beweisführung die entsprechenden Konstruktionsschritte effektiv wiederholt werden.

Nicht selten ist das Ergebnis eines Versuchs, die Korrektheit eines gegebenen Programmteils zu beweisen, ein völlig neu konstruiertes Programmteil, das erstens einfacher und kürzer als die ursprüngliche Version und zweitens – im Gegensatz zur ursprünglichen Version – korrekt ist.

Durch die Verringerung der Anzahl von Softwarefehlern – oder sogar ihre gänzliche Vermeidung – können ein großer Teil des Testaufwands eingespart, die Kosten der Fehlersuche und -korrektur verringert sowie die Produktivität bei der Software-Entwicklung insgesamt erheblich erhöht werden. Noch wichtiger, die Zuverlässigkeit der ausgelieferten Software wird wesentlich verbessert werden. Dementsprechend werden Folgeschäden sowie Kosten der Schadensbehebung erheblich verringert.

1.3 Beabsichtigter Leserkreis

Dieses Buch ist für in der Praxis stehende Software-Entwickler und für diejenigen, die sich auf eine solche Tätigkeit vorbereiten, geschrieben worden.

Es ist einmal als Lehrbuch für den Praktiker konzipiert, der möglichst fehlerfreie Programme schreiben will, ohne die letzten Möglichkeiten der theoretischen Informatik auszuschöpfen und ohne die zugrunde liegende theoretische Basis vollständig erlernen zu müssen.

Für den professionellen Software-Ingenieur und den Software-Ingenieurstudenten ist dieses Buch eine *erste Einführung* in die praktische Konstruktion fehlerfreier Software und die Programm-Korrektheitsbeweisführung. Nach dem Studium des vorliegenden Buchs wird dieser Leserkreis seine Kenntnisse der wissenschaftlichen und mathematischen Grundlage dieses Gebiets vervollständigen wollen. Ferner wird dieser Leserkreis seine Fähigkeit, diese Kenntnisse praktisch anwenden zu können, auf zusätzliche Aussagearten (z.B. Vor- und Nachbedingungen) erweitern wollen. (Siehe [Baber, 1987].) Zusätzliche Empfehlungen für das weitere Studium enthält der Abschnitt "Literaturhinweise" am Ende dieses Buchs.

Der Leser sollte über allgemeine mathematische Vorkenntnisse sowie Erfahrung in der Programmierung verfügen. Wenn er die Fähigkeit noch nicht besitzt, algebraische (insbesondere logische) Ausdrücke manipulieren zu können, muß er bereit sein, sich diese während des Studiums dieses Buchs anzueignen. Der Anhang A vermittelt eine erste Einführung in dieses Gebiet; viele Lehrbücher über Informatik enthalten weiterführende Abschnitte über dieses Thema.

Ferner sollte der Leser bereit sein, seine eigenen bisherigen Erfahrungen kritisch zu überdenken und für neue Ansätze offen sein. Dieses Buch öffnet ihm nämlich die Tür zu einer neuen Software-Entwicklungswelt, die ziemlich anders ist als die, an die er bisher gewohnt war.

1.4 Zielsetzung dieses Buchs

Die Ziele dieses Buchs sind, den Leser

• mit den für die Praxis wichtigsten Aspekten der informatik-mathematischen Grundlage vertraut zu machen, die das

Beweisen der Korrektheit von Computerprogrammen und Algorithmen ermöglicht, ihm
• zu zeigen, wie diese Konzepte als Basis für die Konstruktion eines fehlerfreien Programms bzw. Softwaresystems dienen können, ihn
• in die Lage zu versetzen, diese Erkenntnisse auf praktische Aufgaben anwenden zu können, die in seiner eigenen Software-Entwicklungstätigkeit vorkommen, sowie ihm
• dazu zu verhelfen, nachweislich fehlerfreie Software zu erstellen.

Der Leser wird nach dem Studium dieses Buchs Zeit und Geld sparen und deutlich bessere Software schreiben können. Er wird von seinem Chef besser beurteilt und vor seinen konservativeren Kollegen befördert. In der Softwarewelt von morgen wird er nicht überrumpelt werden.

Es ist beabsichtigt, in diesem Buch den relevanten Stoff in einer relativ einfachen und wenig theoretischen Form zu präsentieren und darzustellen. Ganz ohne theoretische Fundierung geht es natürlich nicht, aber eine für Anwendungszwecke vernünftige Ausgewogenheit zwischen Theorie und Praxis wird angestrebt. Den Vorrang hat die Praxis.

1.5 Inhalt

In diesem Buch liegt die Betonung auf der praktischen Anwendung des hier vorgestellten Stoffs. Dafür gibt es eine vollständige, rigorose mathematische Grundlage, auf die jedoch dieses Buch nicht eingeht. Diese Grundlage wird hier nur informell und nur insofern behandelt, als es notwendig ist, um die darauf basierenden Beweisregeln und ihre Anwendung plausibel und verständlich zu machen.

Dieses Buch hat nur logische Aussagen (Vor- und Nachbedingungen usw.) zum Gegenstand, die sich auf die *Werte* von deklarierten, aktiven Programmvariablen beziehen. Solche Aussagen sind die wichtigsten, die in der Praxis vorkommen, und betreffen die wesentlichsten und problematischsten Aspekte der Programmkorrektheit. Durch Konzentration auf dieses Teilgebiet wird gleichzeitig der Lern- und Einarbeitungsaufwand auf ein Mindestmaß gehalten und der praktische Bedarf an Beweisführungsfertigkeiten weitgehend – jedoch nicht völlig – gedeckt. Bei Beweisen ist es gelegentlich wünschenswert oder sogar erforderlich, weiterführende Aussagen, etwa über die Struktur von Datenumgebun-

gen, zu formulieren. (Solche Aussagen kommen z.B. in Korrektheitsbeweisen für rekursive Unterprogramme und Aufrufe darauf vor, wo mehrere gleichnamige Variablen mit unterschiedlichen Werten in der Datenumgebung festgehalten werden.) Ergänzungen zu der hier geschilderten Grundlage, die derartige formale Aussagen und Beweisschritte ermöglichen, werden in der Literatur behandelt (siehe Abschnitt 1.3 sowie die Literaturhinweise).

Die Metapher des Kapitels 0, Die Zauberlehrlinge im Lande Ret Up Moc, bringt zum Ausdruck, daß die Programm-Korrektheitsbeweisführung und die Konstruktion fehlerfreier Software eine rationale Basis haben und eine professionell ernst zu nehmende Sache bilden, obwohl einige sie immer noch für Zauberei, unrealistische Träumerei oder sogar Scharlatanerie halten. Ob man es wahr haben will oder nicht, sie basieren auf einer wissenschaftlichen Grundlage und ihre verantwortungsbewußte Anwendung gehört zu den ingenieurwissenschaftlichen Fächern.

Im Kapitel 1, Einführung, wird zuerst der Hintergrund der Programm-Korrektheitsbeweisführung mit ihrer praktischen Anwendung und Bedeutung geschildert: das gesellschaftliche Problem, das fehlerhafte Software darstellt, sowie die Lösung, die unsere ingenieurwissenschaftlichen Vorgänger für ähnlich gelagerte Probleme auf anderen Gebieten vor langer Zeit erarbeitet haben. Anschließend werden Zielsetzung und Inhalt dieses Buchs kurz erläutert.

Kapitel 2, Die Ausführung von Programmanweisungen: Wirkungen und Axiome, faßt informell die Definitionen und Annahmen zusammen, die der Programm-Korrektheitsbeweisführung zugrunde liegen.

Im Kapitel 3, Grundlage für die Korrektheitsbeweisführung, werden einleitend einige grundsätzliche Begriffe definiert und danach die sehr wichtigen, allgemein anwendbaren Beweisregeln vorgestellt und erklärt. Sie bilden die Basis für unsere praktische Arbeit.

Kapitel 4, Analyse: das Beweisen der Korrektheit von Programmen, zeigt vor allem anhand von Beispielen, wie man die Beweisregeln anwendet, um die Korrektheit eines gegebenen Programms oder Programmteils zu beweisen. Dabei zerlegt man die Korrektheitsaussage über den fraglichen Programmteil in Korrektheitsaussagen über immer kleinere Bestandteile des Programms, bis nur Aussagen über einzelne Zuweisungen übrigbleiben. Die Beweisregeln ermöglichen so-

wohl das Zerlegen als auch die Verifikation der verbleibenden Korrektheitsaussagen über Zuweisungen.

Im Kapitel 5, Entwurf: die Konstruktion beweisbar korrekter Programme, wird veranschaulicht, wie die Anforderungen eines Korrektheitsbeweises als Leitlinien für mehrere Konstruktionsschritte dienen können und es sogar ermöglichen, einige Teile des zu konstruierenden Programms direkt abzuleiten. Diese für manchen Programmierer neue, ungewohnte Vorgehensweise führt gezielt und auf systematische Weise zu einem kompakten, korrekten Programm. Dabei werden das Programm und sein Korrektheitsbeweis gleichzeitig erstellt.

Im Kapitel 6, Die Formulierung von Vor- und Nachbedingungen, wird auf das Thema eingegangen, wie man unpräzise verbale Spezifikationen in präzise logische algebraische Ausdrücke (Bedingungen) übersetzen kann.

Kapitel 7, Zusammenfassung, wiederholt in komprimierter Form die wichtigsten Erkenntnisse und Schlüsse aus den vorherigen Abschnitten und gibt einen kurzen Überblick über die ingenieurmäßige Software-Entwicklung von morgen.

Der Anhang A führt den Leser in die logische Algebra ein. Er wird diesen Abschnitt vor allem als Gedächtnisstütze verwenden und nach Bedarf bestimmte Einzelheiten dort nachschlagen.

Anhang B enthält Lösungen zu den Übungsaufgaben, die an verschiedenen Stellen des Buchs gestellt wurden.

Am Ende des Buchs befinden sich Literaturhinweise und ein Register.

Eine Übersichtskarte über die praktische Anwendung der Beweisregeln liegt dem Buch bei.

2. Ausführung von Programmanweisungen: Wirkungen und Axiome

Gegenstand dieses Kapitels sind die wesentlichsten Annahmen über die Wirkung der verschiedenen Anweisungen, die den in Kapitel 3 vorgestellten Beweisregeln sowie der Korrektheitsbeweisführung im allgemeinen zugrunde liegen. Es handelt sich hier um weitgehend bekannte, jedoch manchmal übersehene Eigenschaften dieser Anweisungen.

Die Ausführung einer Programmanweisung hat eine bestimmte Wirkung, die im Detail von den spezifischen Eigenschaften des jeweiligen Programmiersprachensystems abhängt. Typischerweise jedoch treffen die in den folgenden Abschnitten aufgeführten Bemerkungen zu.

In allen Arten von Programmanweisungen kommen Ausdrücke vor. Bei der Ausführung einer Programmanweisung wird in der Regel der Wert jedes darin vorkommenden Ausdrucks ermittelt. Wir betrachten deshalb zuerst den Prozeß, der einen Ausdruck auswertet.

2.1 Die Auswertung von Ausdrücken

Bei der Auswertung eines Ausdrucks während eines Programmlaufs wird effektiv zuerst jeder Name einer Programmvariablen durch den gegenwärtigen Wert der entsprechenden Variablen ersetzt. Danach werden die im Ausdruck vorkommenden Operationen durchgeführt. Das Ergebnis ist der Wert des Ausdrucks.

Beispiel: Wenn die Werte der Variablen x, y und z zur fraglichen Zeit 3, 4 bzw. 5 sind, wird der Ausdruck x*(y+z)<(x+z) wie folgt ausgewertet:

```
x*(y+z)  < (x+z)
3*(4+5)  < (3+5)
3*9      < 8
27       < 8
falsch ■
```

Der Wert eines Ausdrucks ist im allgemeinen nur dann definiert, wenn die Werte aller darin vorkommenden Variablen sowie aller Zwischenergebnisse, die während der Auswertung ermittelt werden müssen, definiert sind. Ferner müssen diese Werte innerhalb bestimmter Wertebereiche liegen. Sind diese Voraussetzungen nicht erfüllt, dann ist der Wert des Ausdrucks im allgemeinen nicht definiert; der Programm- oder Compilerlauf bricht mit einer entsprechenden Fehlermeldung abnormal ab.

Beispiel: Wenn die Werte von x, y und z 3, "Hans" bzw. 5 sind, wird der Ausdruck x*(y+z)<(x+z) wie folgt ausgewertet:

```
x*(y+z)                 < (x+z)
3*("Hans"+5)            < (3+5)
3*(nicht definiert)     < 8
nicht definiert         < 8
nicht definiert ■
```

Das Ergebnis der Addition (+) ist nicht definiert, wenn ein Argument eine Zeichenkette wie "Hans" ist. Das nicht definierte Zwischenergebnis setzt sich hier bis zum Ende der Auswertung des gesamten Ausdrucks fort.

Es gibt allerdings Programmiersprachensysteme, die in bestimmten Situationen nicht definierte Zwischenergebnisse zulassen und dabei wohl definierte Endergebnisse ermitteln können.

Beispiel: Sei Y ein Feld, das nur für die Indexwerte 1 bis 10 einschließlich definiert (deklariert) ist. Sei der Wert von n=11. Der Ausdruck (n≤10 **und** Y(n)=x) soll ausgewertet werden:

```
n≤10 und Y(n)=x
11≤10 und Y(11)=(Wert von x)
falsch und [nicht definiert=(Wert von x)]
falsch und nicht definiert
falsch ■
```

Auf solche Implementierungsdetails muß geachtet werden. Im letzten Beispiel oben berechnet das System den Ausdruck (falsch **und** nicht definiert) als falsch. Einige Systeme jedoch betrachten den Wert dieses Ausdrucks als nicht definiert.

2.2 Die Ausführung einer Zuweisung

Die Zuweisung besteht aus dem Namen einer Variablen, dem Zuweisungssymbol (:=) und einem Ausdruck, in dem die Namen beliebiger Programmvariablen vorkommen können. Sie hat die Form:

x:=A(x, y, ...)

Bei der Ausführung einer solchen Zuweisung wird zuerst der Ausdruck A(x, y, ...) ausgewertet (siehe Abschnitt 2.1 oben). Der Wert des Ausdrucks wird der Variablen x zugeordnet, d.h., er wird der neue Wert der Variablen x. Die Werte aller anderen Variablen bleiben unverändert.

Das Ergebnis der Ausführung einer Zuweisung ist demnach definiert, wenn die Variable x deklariert ist (bzw. automatisch deklariert wird) sowie der Wert des Ausdrucks A definiert ist und (ggf. nach automatischer Umwandlung, z.B. Rundung) im deklarierten Wertebereich der Variable x liegt.

Die Wirkung der Ausführung der Zuweisung oben kann im folgenden Axiom zusammengefaßt werden.

Zuweisungsaxiom: Man bezeichne den Wert der Variablen x vor der Ausführung der Anweisung x', ihren Wert danach x" usw. Die Werte der verschiedenen Variablen vor und nach der Ausführung einer Zuweisung der oben angegebenen Form genügen den folgenden Gleichungen:

x" = A(x', y', ...)
y" = y', für alle anderen Variablennamen y ■

Achte auf die Annahme, daß nur der Wert der Variablen, deren Name links in der Zuweisung steht, verändert wird. Sogenannte "Nebeneffekte", wodurch Werte auch anderer Variablen verändert werden, sind nicht erlaubt. Wird diese Annahme verletzt, gelten im allgemeinen weder das Zuweisungsaxiom noch die daraus folgenden Beweisregeln.

Steht der Name einer Feldvariablen links in einer Zuweisung, dann wird auch der Wert des Indexausdrucks ermittelt, um die eigentlich angesprochene Variable zu bestimmen.

Beispiel: Ist n=3, dann ist die Zuweisung

x(n):= ...

als

x(3):= ...

zu verstehen. ■

2.3 Die Ausführung einer if-Anweisung

Die if-Anweisung hat die Form

if B then S1 else S2 endif

wobei B eine Bedingung (ein Ausdruck, dessen Wert entweder falsch oder wahr ist) ist und S1 und S2 Anweisungen sind. In der Bedingung B können die Namen beliebiger Programmvariablen vorkommen. S1 und S2 können zusammengesetzte Anweisungen, d.h. Folgen von Anweisungen, if-Anweisungen, Schleifen usw. sein.

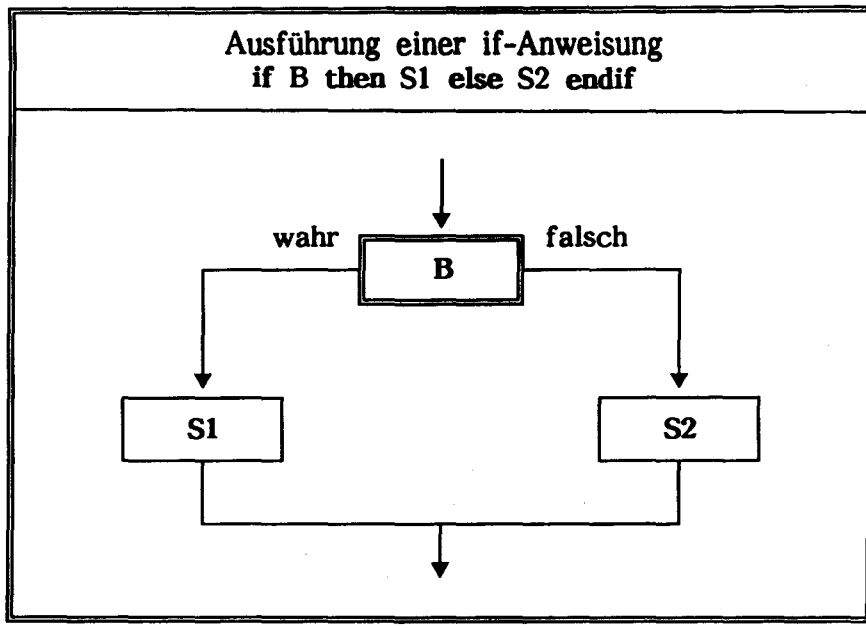
if-Axiom: Die Ausführung der oben angegebenen if-Anweisung hat die gleiche Auswirkung wie die Ausführung von

S1, falls B'=wahr, oder
S2, falls B'=falsch ■

D.h., die ganze if-Anweisung ist entweder S1 oder S2 äquivalent, je nachdem, ob B' wahr oder falsch ist.

B' ist der Wert der Bedingung B vor der Ausführung der if-Anweisung, d.h. er wird auf Grund der Werte aller in der Bedingung B vorkommenden Variablen *vor* der Ausführung der if-Anweisung ermittelt.

Das Ergebnis der Ausführung einer if-Anweisung ist demnach definiert, wenn der Wert der Bedingung B definiert (falsch oder wahr) ist und das Ergebnis der Ausführung von S1 bzw. S2 – je nach dem jeweiligen Wert von B – definiert ist.



2.4 Die Ausführung einer Folge von Anweisungen

Eine Folge von Anweisungen wird sequentiell, eine nach der anderen, ausgeführt. Das Ergebnis der Ausführung der ersten Anweisung in der Folge (z.B. die nachherigen Werte der Programmvariablen) ist die Ausgangsbasis für die Ausführung der zweiten Anweisung in der Folge usw. Demnach ist das Ergebnis der Ausführung einer Folge von Anweisungen offensichtlich definiert, wenn die Ausführung jeder einzelnen Anweisung in der Folge zu einem definierten Ergebnis führt.

2.5 Die Ausführung einer while-Schleife

Die while-Anweisung hat folgende Form:

while B **do** S **endwhile**

Dabei ist B eine Bedingung und S eine (evtl. zusammengesetzte) Anweisung.

while-Axiom: Die Ausführung der oben angegebenen while-Anweisung hat die gleiche Auswirkung wie die Ausführung von

S
while B **do** S **endwhile**, falls B'=wahr, oder

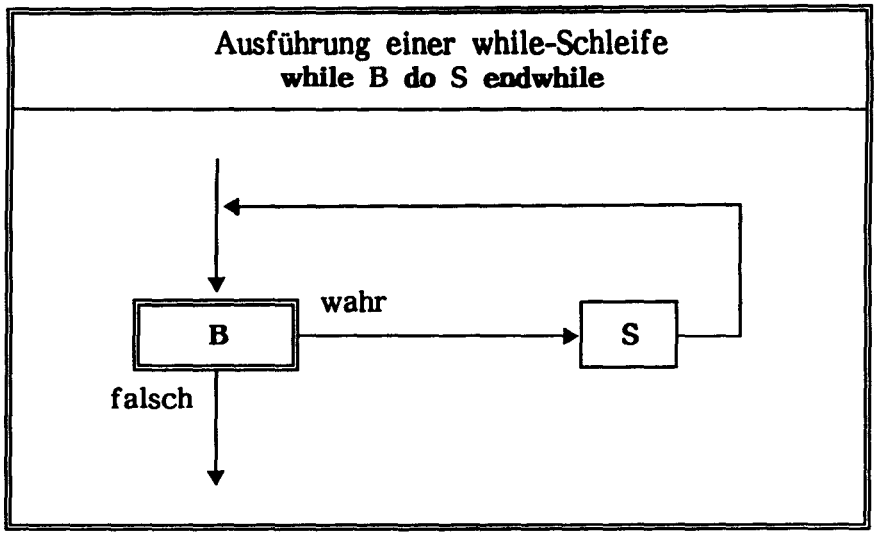
die Null-Anweisung, falls B'=falsch ■

D.h., bei der Ausführung der while-Anweisung wird zuerst der Wert der Bedingung B ermittelt, ausgehend von den Werten aller darin vorkommenden Programmvariablen *vor* der Ausführung der while-Anweisung. Ist dieser Wert wahr, wird S und anschließend nochmals die while-Anweisung ausgeführt. Ist der Wert von B falsch, wird die ganze while-Anweisung, einschließlich S, übersprungen und die Ausführung des Programms mit den nachfolgenden Anweisungen (falls vorhanden) fortgesetzt.

Ist der Wert der Bedingung B immer wahr, dann wird der Schleifenkern S immer wieder ausgeführt; die Schleife kommt nie zu Ende.

Das Ergebnis der Ausführung einer while-Schleife ist demnach definiert, wenn

- der Wert der Bedingung B bei jeder Auswertung derselben definiert (falsch oder wahr) ist und
- das Ergebnis jeder Ausführung des Schleifenkerns S definiert ist und
- der Wert der Bedingung B nach einer endlichen Anzahl von Ausführungen des Schleifenkerns S falsch ist.



2.6 Die Ausführung eines Prozedur-Aufrufs (Unterprogramms)

Der Aufruf eines Unterprogramms ohne formale Parameterübergabe bewirkt, daß die im Unterprogramm vorkommenden Anweisungen ausgeführt werden, als ob sie im Programmtext anstelle des Aufrufs stünden.

Viele Programmiersprachen erlauben es, im Aufruf eines Unterprogramms Parameter formal zu übergeben. Der dafür vorgesehene Mechanismus wird in den verschiedenen Programmiersprachensystemen auf unterschiedliche Weise implementiert. Wenn die Parameterübergabe im Korrektheitsbeweis berücksichtigt werden soll, ist es sinnvoll und meist am einfachsten, den Aufruf mit formaler Parameterübergabe, zumindest gedanklich, in einen Aufruf ohne formale Parameterübergabe umzuschreiben, wobei die spezifischen Parameterübergaberegeln des fraglichen Systems zum Ausdruck kommen. Anschließend beweist man die Korrektheit des Programms mit dem Aufruf ohne formale Parameterübergabe.

Beispiel:

Aufruf	aufgerufenes Unterprogramm
call U(x, y)	U(a, b): ergebnis:=a+b

Dieser Aufruf hat in manchen Systemen die gleiche Wirkung wie:

Aufruf	aufgerufenes Unterprogramm
call U	U: a:=x b:=y ergebnis:=a+b

wobei a und b Variablen sind, die nur innerhalb des Unterprogramms U existieren. Sie werden beim Aufruf von U erzeugt und am Ende der Ausführung von U wieder freigegeben (gelöscht). ■

3. Grundlage für die Korrektheitsbeweisführung

Um die Korrektheit eines Programms oder Programmteils zu beweisen, formuliert man effektiv einen mathematischen Satz über die Wirkung der Ausführung des fraglichen Programmteils. Danach beweist man diesen Satz. In der Praxis haben diese Sätze fast immer die gleiche Form: Falls eine bestimmte Bedingung gleich vor der Ausführung des fraglichen Programmteils erfüllt (wahr) ist, dann wird gleich nach der Ausführung des Programmteils eine (im allgemeinen andere) bestimmte Bedingung erfüllt (wahr) sein. Solche Bedingungen werden *Vor-* bzw. *Nachbedingungen* genannt.

Beide Bedingungen beziehen sich auf Programmvariablen. Die wichtigsten Aspekte von Vor- und Nachbedingungen beziehen sich sogar nur auf die *Werte* der Programmvariablen. In diesem Buch betrachten wir nur solche Bedingungen.

Der Einfachheit halber teilt man Korrektheitsbeweise in zwei Teile auf. In einem Teil zeigt man, daß das fragliche Programmsegment überhaupt läuft, d.h. ohne Compilierungs- oder Laufzeitfehler zu Ende ausgeführt wird. Im anderen Teil beweist man den oben erwähnten Korrektheitssatz unter der Annahme, daß der Programmteil zu Ende läuft.

3.1 Definitionen

Eine *Bedingung* ist ein algebraischer Ausdruck, dessen Wert "falsch" oder "wahr" ist. In einem Ausdruck kommen typischerweise Namen von Programmvariablen vor, die stellvertretend für die Werte der fraglichen Programmvariablen stehen. Man spricht auch von *logischen* und *Booleschen* Ausdrücken, *Zusicherungen* und *Aussagen.*

Beispiele:

```
x>8
y=4
x>3 und y+z>6
KUNDENNAME="Schmidt"
3000≤Gehalt<4000
```

$A(1) \leq A(2) \leq \ldots \leq A(n)$
$A(1) \leq A(2)$ **und** $A(2) \leq A(3)$ **und** ... $A(n-1) \leq A(n)$

$$\textbf{und}_{i=1}^{n-1} A(i) \leq A(i+1) \quad \blacksquare$$

Wenn aus der Wahrheit einer Bedingung V vor der Ausführung einer Anweisung S die Wahrheit einer Bedingung P danach folgt, dann sagt man, daß V eine *Vorbedingung* der *Nachbedingung* P bezüglich S ist. Symbolisch schreibt man {V} S {P}. Die Anweisung S kann eine Zusammensetzung von mehreren einzelnen Anweisungen, z.B. ein ganzes Programm, sein.

Beispiel: Falls x>3 vor der Ausführung der Zuweisung x:=x+5, dann wird x>8 danach gelten. Symbolisch,

{x>3} x:=x+5 {x>8} ■

Das Ergebnis der Ausführung einer Anweisung, eines Programmteils oder eines Programms ist *richtig* (*korrekt*), wenn es die geforderte Nachbedingung erfüllt.

Eine Anweisung (oder auch ein Programmteil oder Programm) *terminiert*, wenn sie in endlicher Zeit ohne Laufzeitfehler (und ohne Compilierungsfehler) zu Ende – d.h. mit einem definierten Ergebnis – ausgeführt wird.

Liefert die Ausführung einer Anweisung, eines Programmteils oder eines Programms ein richtiges Ergebnis – sofern sie überhaupt ein definiertes Ergebnis liefert (terminiert) – dann spricht man von *partieller* Korrektheit. Wird zusätzlich gezeigt, daß die Ausführung des Programms terminiert, spricht man von *vollständiger* Korrektheit. Es ist zweckmäßig, zwischen diesen zwei Aspekten der Korrektheit klar zu unterscheiden, weil unsere Beweise dadurch vereinfacht werden. Ferner eignen sich sehr unterschiedliche Vorgehensweisen für diese zwei Teile eines Beweises (siehe Kapitel 4).

3.2 Vor- und Nachbedingungen in Korrektheitsbeweisen

Vor- und Nachbedingungen sind die wichtigsten Elemente der Programm-Korrektheitsbeweisführung. Sie stellen die Definition von "korrekt" in bezug auf ein bestimmtes Pro-

gramm oder einen Programmteil dar. Anders ausgedrückt bilden eine Vor- und eine Nachbedingung zusammen die Spezifikation (das Pflichtenheft) eines Programmteils oder eines Programms.

Die wesentlichsten Teile eines typischen Korrektheitsbeweises befassen sich mit Vor- und Nachbedingungen und vor allem mit Beziehungen dazwischen. Manchmal leitet man, ausgehend von einer Nachbedingung und einer Anweisung (oder einer Zusammensetzung von Anweisungen), eine Vorbedingung algebraisch ab. Diese Vorgehensweise ist besonders bei Zuweisungen sinnvoll. Oft zerlegt man eine Aussage über die Vor- und Nachbedingungen eines Programmteils in Aussagen über die Vor- und Nachbedingungen seiner einzelnen Bestandteile, die anschließend auf einfache Weise bewiesen werden. Dazu bedient man sich einiger nützlicher Regeln, die wir in den folgenden Abschnitten kennenlernen werden.

3.3 Beweisregeln

Im folgenden werden die wichtigsten allgemein gültigen Sätze, die man in der Praxis für die Korrektheitsbeweisführung braucht, in der Form von "Beweisregeln" vorgestellt. Für jede Anweisung und Zusammensetzung von Anweisungen (Zuweisung, if-Anweisung, Folge von Anweisungen und while-Schleife) werden eine oder mehrere Beweisregeln aufgeführt und kurz erläutert. Zusätzliche Beweisregeln ermöglichen es, die algebraische Manipulation der in einem Beweis auftretenden logischen Ausdrücke zu vereinfachen.

Jede Beweisregel hat eine Beziehung zwischen einer Vorbedingung und einer Nachbedingung zum Gegenstand. Analogien aus anderen technischen Bereichen sind z.B. die Gesetze von Ohm, Faraday und Henry, die jeweils eine Beziehung zwischen Spannung und Stromstärke für eine bestimmte elektrische Komponente beschreiben, sowie die entsprechenden Gleichungen für eine Masse, eine Feder und einen Stoßdämpfer, die die jeweilige Beziehung zwischen Kraft und Position beschreiben.

Wir fangen mit einer Beweisregel an, die der Vereinfachung der algebraischen Manipulation dient. Sie folgt aus der Definition von Vor- und Nachbedingungen oben und erleichtert es, einige der anderen Beweisregeln zu verstehen.

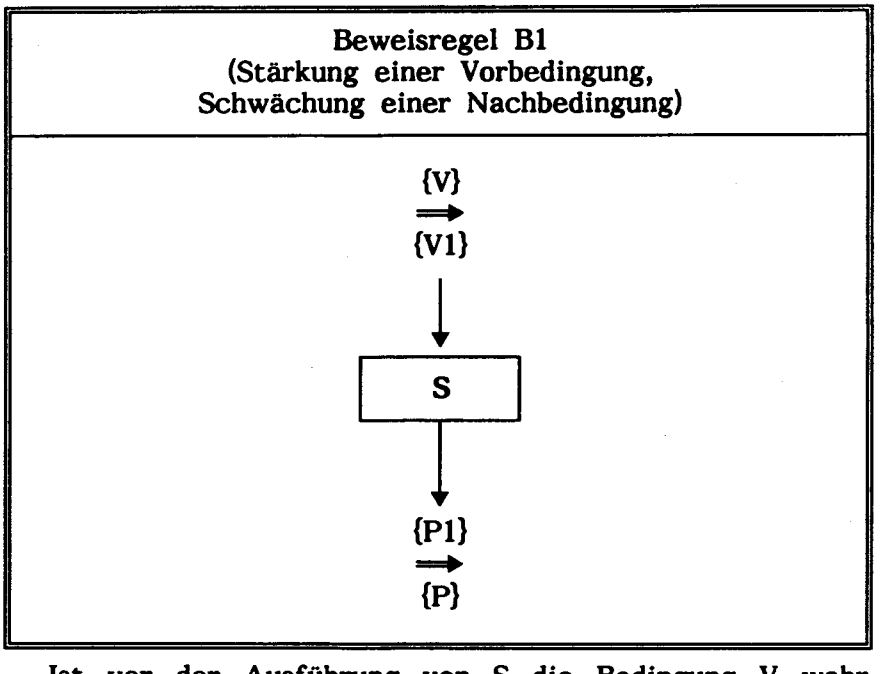
3.3.1 Beweisregel B1: Stärkung einer Vorbedingung und Schwächung einer Nachbedingung

Falls

$V \Longrightarrow V1$ und
$\{V1\}\ S\ \{P1\}$ und
$P1 \Longrightarrow P$

dann gilt, daß

$\{V\}\ S\ \{P\}$ ■



Ist vor der Ausführung von S die Bedingung V wahr, und folgt aus V die Vorbedingung V1 von der Nachbedigung P1 bezüglich S, und folgt aus P1 die Bedingung P, dann wird P nach der Ausführung von S wahr sein. D.h., aus der vorherigen Wahrheit von V folgt die nachherige Wahrheit von P. Nach der Definition ist also V eine Vorbedingung von P bezüglich S.

Arbeitet man *rückwärts* durch ein Programm, darf man Bedingungen *stärken* (verschärfen). Man kann eine Bedingung dadurch stärken, daß man einen beliebigen Term durch und-

Verknüpfung hinzufügt oder daß man einen vorhandenen **oder**-verknüpften Term wegläßt.

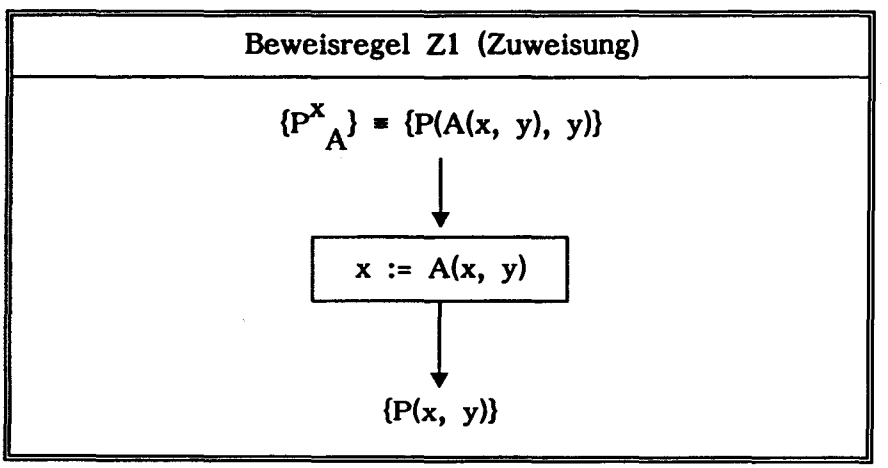
Arbeitet man *vorwärts* durch ein Programm, darf man Bedingungen *schwächen* (lockern). Man kann eine Bedingung dadurch schwächen, daß man einen beliebigen Term durch **oder**-Verknüpfung hinzufügt oder daß man einen vorhandenen **und**-verknüpften Term wegläßt.

Durch gekonnte Anwendung der Beweisregel B1 kann man die in einem Beweis auftretende algebraische Manipulation manchmal erheblich vereinfachen. Man muß jedoch darauf achten, daß eine Vorbedingung nicht so gestärkt oder eine Nachbedingung so geschwächt wird, daß der Beweis nicht mehr vervollständigt werden kann. In späteren Beispielen werden wir sehen, wie man dieses potentielle Problem leicht vermeiden kann.

3.3.2 Beweisregel Z1: Ableitung einer Vorbedingung einer Zuweisung

Um eine Vorbedingung einer gegebenen Nachbedingung P bezüglich einer gegebenen Zuweisung x:=A zu ermitteln, ersetze die Variable x durch den Ausdruck (A) überall in P. Symbolisch,

$\{P^x_A\}$ x:=A {P} ∎



Vergiß dabei nicht, den Ausdruck A in Klammern zu setzen. Es ist manchmal überflüssig, aber nie falsch. Es ist manchmal falsch, die Klammern wegzulassen.

Der Wert von x nach der Ausführung der Zuweisung x:=A ist gleich dem Wert von A vorher (siehe Abschnitt 2.2). Der Wert von y bleibt unverändert (Annahme: keine "Nebenwirkung" der Zuweisung). (Die Variable y steht hier stellvertretend für alle Programmvariablen außer x.) Der Wert von P(x, y) nach der Ausführung der Zuweisung ist also gleich dem Wert von P(A, y) vorher. Deshalb folgt aus der Wahrheit von P(A, y) vorher die Wahrheit von P(x, y) nachher. Nach der Definition ist P(A, y) eine Vorbedingung der Nachbedingung P(x, y) bezüglich der Zuweisung.

Beispiel: Um eine Vorbedingung der Nachbedingung {10<y **und** x<8} bezüglich der Zuweisung x:=x-5 zu ermitteln, ersetzt man in der Nachbedingung die Variable x durch den Ausdruck (x-5). Das Ergebnis ist {10<y **und** (x-5)<8} oder, äquivalent, {10<y **und** x<13}. Symbolisch,

{10<y **und** x<13} x:=x-5 {10<y **und** x<8}

Gemäß der Beweisregel B1 ist jede stärkere Bedingung auch eine Vorbedingung, z.B.

{10<y **und** 0≤x<13} x:=x-5 {10<y **und** x<8}

und

{10<y<x+N **und** 0≤x<13} x:=x-5 {10<y **und** x<8} ∎

3.3.3 Beweisregel Z2: Verifikation einer gegebenen Vorbedingung einer Zuweisung

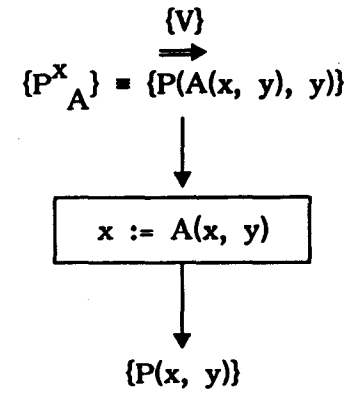
Falls

$$V \Longrightarrow P^{x}_{A}$$

dann gilt, daß

{V} x:=A {P} ∎

Beweisregel Z2 (Zuweisung)



Die Beweisregel Z2 ist eine Kombination der Beweisregeln Z1 und B1.

Gemäß der Beweisregel B1 ist V eine Vorbedingung von P bezüglich der Zuweisung, falls

$$(V \Longrightarrow P^x_A) \text{ und } \{P^x_A\}\ x{:=}A\ \{P\}$$

Gemäß der Beweisregel Z1 gilt, daß

$$\{P^x_A\}\ x{:=}A\ \{P\}$$

Es genügt also zu zeigen, daß $V \Longrightarrow P^x_A$, wenn man verifizieren will, daß $\{V\}\ x{:=}A\ \{P\}$.

Bei der Anwendung der Beweisregel Z2 wendet man implizit die zwei Beweisregeln Z1 und B1 an. Effektiv ermittelt man zuerst eine Vorbedingung bezüglich der Zuweisung gemäß der Beweisregel Z1. Danach prüft man, ob aus der gegebenen Vorbedingung die ermittelte Vorbedingung folgt, d.h., ob die Voraussetzung der Beweisregel B1 erfüllt ist.

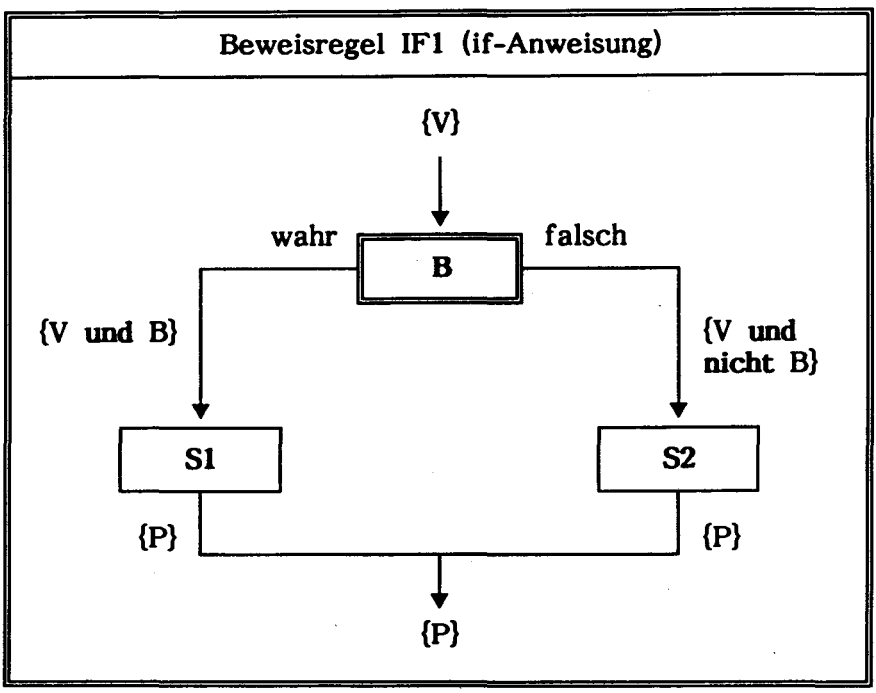
3.3.4 Beweisregel IF1: Verifikation einer gegebenen Vorbedingung einer if-Anweisung

Falls

{V **und** B} S1 {P} und
{V **und nicht** B} S2 {P}

dann gilt, daß

{V} **if** B **then** S1 **else** S2 **endif** {P} ∎



Falls vor der Ausführung der if-Anweisung die Bedingung V wahr ist, dann werden vor der eventuellen Ausführung von S1 sowohl V als auch B wahr sein. Weil (V **und** B) eine Vorbedingung von P bezüglich S1 ist, wird nach der Ausführung von S1 die Nachbedingung P wahr sein. Entsprechend wird auch nach der eventuellen Ausführung von S2 die Nachbedingung P wahr sein. Die nachherige Wahrheit von P folgt in jedem Fall aus der vorherigen Wahrheit von V. Nach der Definition ist also V eine Vorbedingung von P bezüglich der gesamten if-Anweisung.

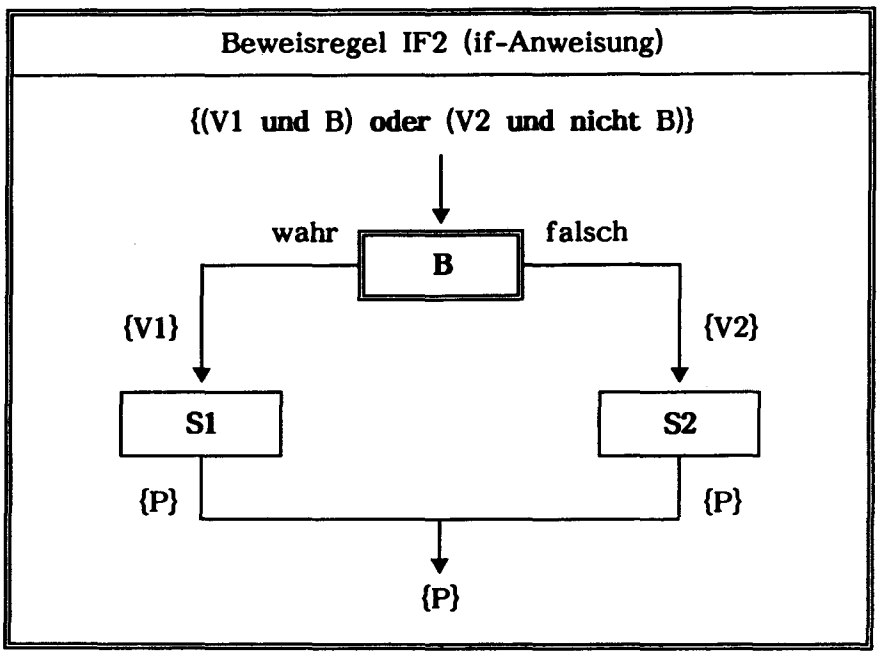
3.3.5 Beweisregel IF2: Ableitung einer Vorbedingung einer if-Anweisung

Falls

{V1} S1 {P} und
{V2} S2 {P}

dann gilt, daß

{(V1 **und** B) **oder** (V2 **und nicht** B)}
if B **then** S1 **else** S2 **endif** {P} ∎



Die Beweisregel IF2 ist im wesentlichen die Beweisregel IF1 mit V = [(V1 **und** B) **oder** (V2 **und nicht** B)]. Sie folgt aus den Beweisregeln IF1 und B1.

Mit Hilfe der Beweisregel IF2 kann man eine Vorbedingung einer gegebenen Nachbedingung P bezüglich einer gegebenen if-Anweisung ableiten. Zuerst ermittelt man Vorbedingungen von P bezüglich S1 und S2. Dafür wendet man die für S1 und S2 geeigneten Beweisregeln an. Dann kombiniert man die dadurch gefundenen Vorbedingungen wie oben angegeben.

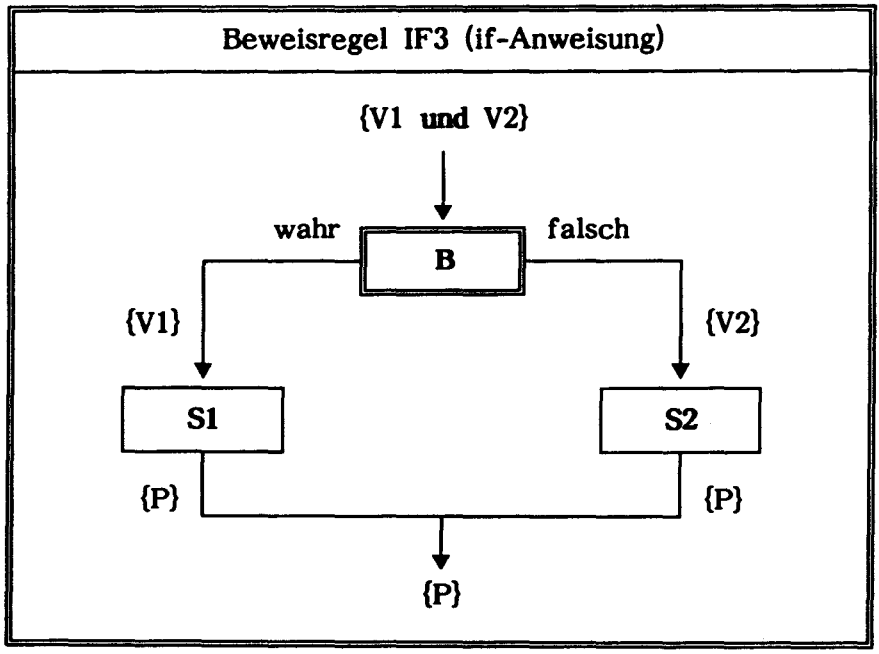
3.3.6 Beweisregel IF3: if-Anweisung

Falls

{V1} S1 {P} und
{V2} S2 {P}

dann gilt, daß

{V1 **und** V2} **if** B **then** S1 **else** S2 **endif** {P} ∎



Beweisregel IF3 ist eine einfache, schwache Aussage, die aus den Beweisregeln IF1 und B1 folgt. Wegen der einfachen Form der darin vorkommenden Ausdrücke ist sie in der Praxis manchmal nützlich.

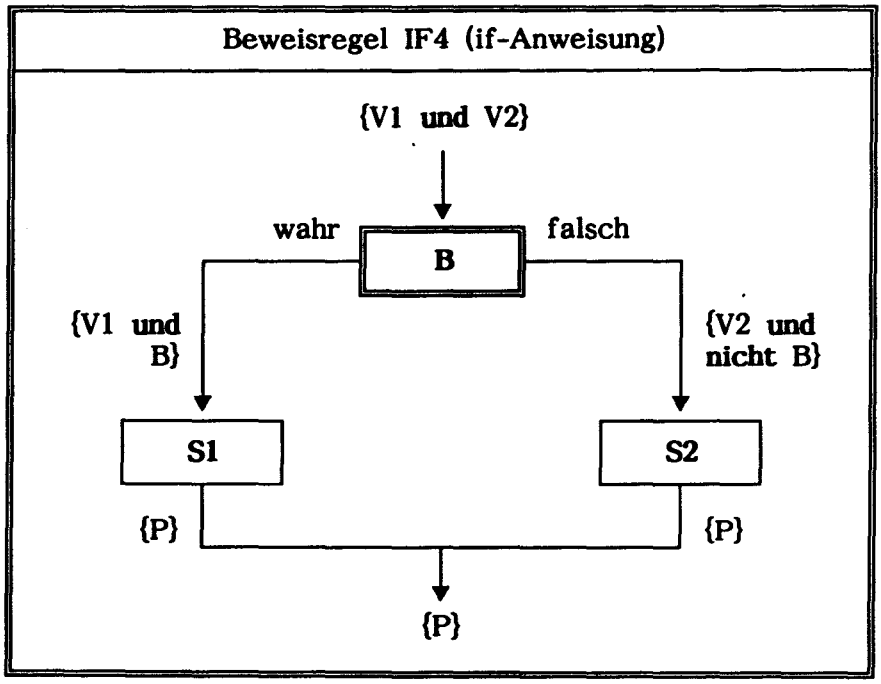
3.3.7 Beweisregel IF4: if-Anweisung

Falls

{V1 **und** B} S1 {P} und
{V2 **und nicht** B} S2 {P}

dann gilt, daß

{V1 **und** V2} **if** B **then** S1 **else** S2 **endif** {P} ■



Auch Beweisregel IF4 folgt aus den Beweisregeln IF1 und B1. Wie Beweisregel IF3 ist sie wegen der einfachen Form der Vorbedingung (V1 **und** V2) manchmal von praktischem Nutzen.

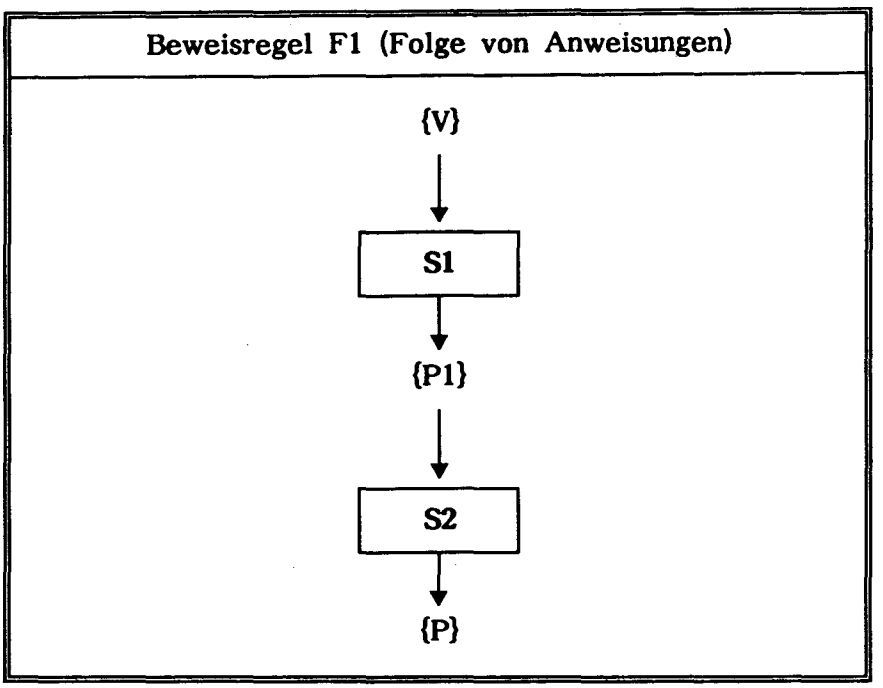
3.3.8 Beweisregel F1: Folge von Anweisungen

Falls

{V} S1 {P1} und
{P1} S2 {P}

dann gilt, daß

{V} (S1; S2) {P} ■



Die Beweisregel F1 läßt sich auf offensichtliche Weise für eine beliebig lange Folge von Anweisungen verallgemeinern. Um eine Vorbedingung einer gegebenen Nachbedingung P bezüglich einer Folge von Anweisungen zu ermitteln, leitet man zuerst eine Vorbedingung von P bezüglich der letzten Anweisung der Folge ab. Diese Vorbedingung wird als Nachbedingung der vorletzten Anweisung verwendet usw. Auf diese Weise arbeitet man rückwärts durch die gesamte Folge, Anweisung für Anweisung. Die auf diese Weise gefundene Vorbedingung bezüglich der ersten Anweisung in der Folge ist gleichzeitig eine Vorbedingung von P bezüglich der gesamten Folge von Anweisungen.

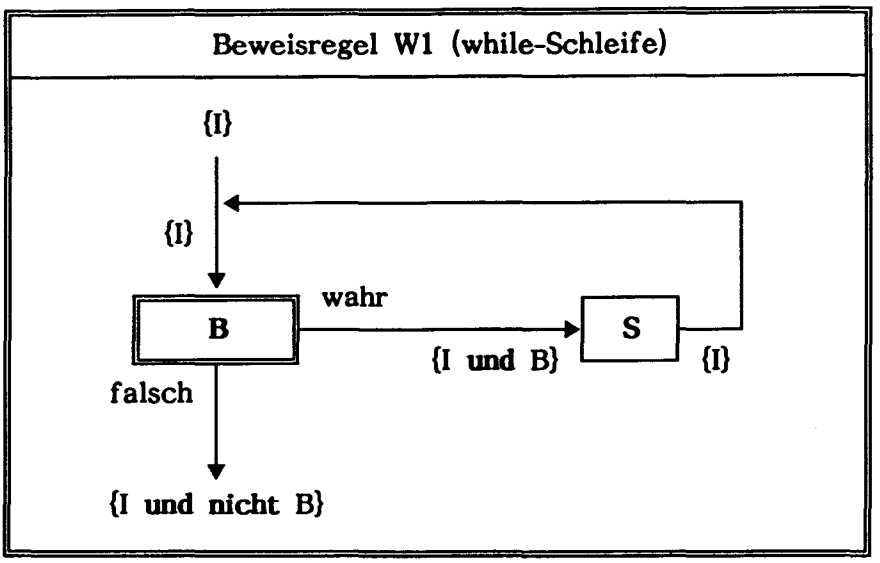
3.3.9 Beweisregel W1: while-Schleife ohne Initialisierung

Falls

{I **und** B} S {I}

dann gilt, daß

{I} **while** B **do** S **endwhile** {I **und nicht** B} ∎



Falls die Bedingung I vor der Ausführung der while-Schleife wahr ist, dann werden sowohl I als auch B vor der ersten Ausführung des Schleifenkerns S wahr sein. Weil (I **und** B) eine Vorbedingung von I bezüglich S ist, wird I nach der ersten Ausführung von S wahr sein. Die Bedingungen I und B werden deshalb vor der zweiten Ausführung von S wahr sein usw. Die Bedingung I wird nach jeder Ausführung von S wahr sein. Wenn die Schleife zum Ende kommt, wird I wahr und B falsch sein. D.h., nach eventueller Terminierung der Schleife wird die Bedingung (I **und nicht** B) wahr sein.

Der Wert der Bedingung I ist vor und nach jeder Ausführung des Schleifenkerns S wahr, also konstant. Die Bedingung I wird deshalb *Schleifeninvariante* genannt. Die Schleifeninvariante ist der Schlüssel zur Konstruktion und zum Verständnis einer Schleife.

Die Anwendung der Beweisregel W1 verlangt, daß die Schleifeninvariante I vor der Schleife wahr ist. (Meist ist I auf triviale Weise anfangs wahr.) Mit anderen Worten, die Anfangssituation ist ein spezieller Fall von I. Nach der Ausführung der Schleife (bei Terminierung) ist (I **und nicht** B) wahr. Die Endsituation ist also auch ein spezieller Fall von I. Anders herum betrachtet ist die Schleifeninvariante I eine Verallgemeinerung der Anfangs- und Endsituationen. Diese Beobachtung liefert uns die sehr nützliche

Faustregel für die Bestimmung einer Schleifeninvariante: Verallgemeinere (schwäche, lockere) die Anfangs- und Endsituationen (Vor- und Nachbedingungen), um eine geeignete Schleifeninvariante zu ermitteln. ■

Vor fast jeder Schleife steht ihre "Initialisierung", ein Programmteil, dessen *einzige* Aufgabe es ist, die anfängliche Wahrheit der Schleifeninvariante sicherzustellen.

Sehr oft will man die Korrektheit einer Schleife zusammen mit ihrer Initialisierung beweisen. Dazu dient die Beweisregel W2.

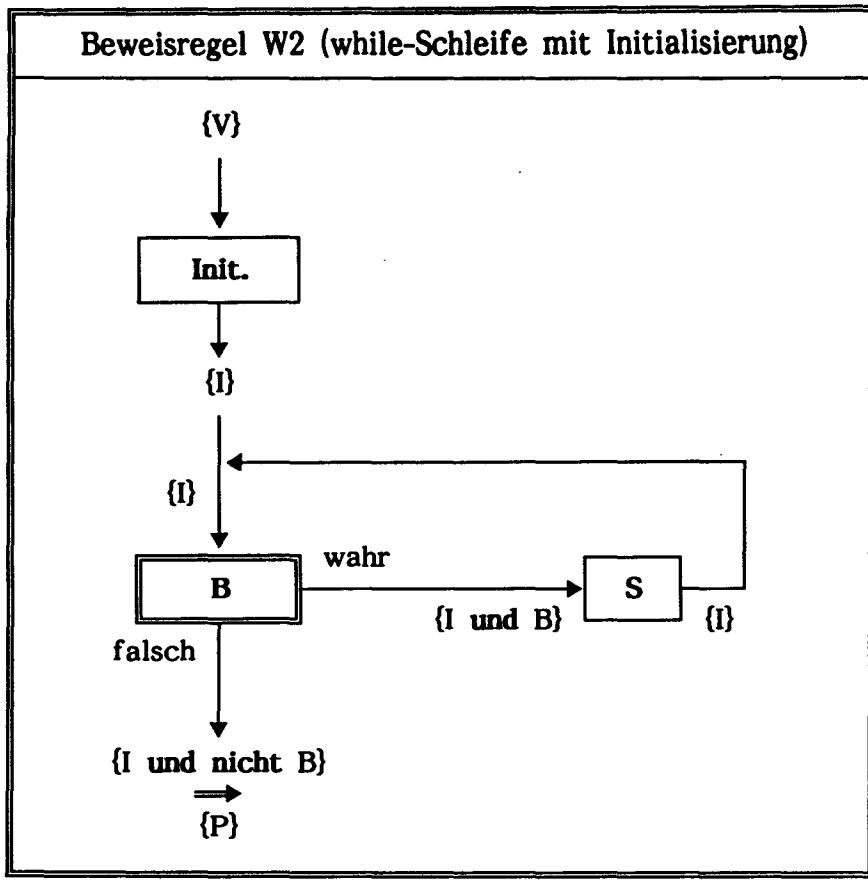
3.3.10 Beweisregel W2: while-Schleife mit Initialisierung

Gegeben sei eine Bedingung I (Schleifeninvariante). Falls

{V} Initialisierung {I} und
{I **und** B} S {I} und
(I **und nicht** B) ⟹ P

dann gilt, daß

{V} (Initialisierung; **while** B **do** S **endwhile**) {P} ■



Die Beweisregel W2 folgt aus den Beweisregeln F1, B1 und W1.

Gemäß der Beweisregel W2 beweist man die partielle Korrektheit einer while-Schleife dadurch, daß man

1. die Schleifeninvariante I bestimmt (falls nicht bereits vom Programmierer angegeben),
2. beweist, daß {V} Initialisierung {I} (d.h., daß I am Anfang der Schleife wahr ist),
3. beweist, daß {I und B} S {I} (d.h., daß der Schleifenkern die Wahrheit von I aufrecht erhält) und
4. beweist, daß (I und nicht B) $\Longrightarrow$ die Nachbedingung P (d.h., daß P beim eventuellen Beenden der Schleife wahr ist).

Man beweist die vollständige Korrektheit dadurch, daß man

5. zusätzlich zeigt, daß die Schleife zu Ende kommt, d.h. unter anderem, daß es eine obere Schranke für die Anzahl der Ausführungen des Schleifenkerns gibt (siehe Abschnitt 2.5).

Dazu definiert man typischerweise eine Funktion, deren Wert (1) bei jeder Ausführung des Schleifenkerns um mindestens einen festen Betrag (>0) verringert (oder erhöht) wird und (2) nach unten (bzw. oben) begrenzt ist. Eine derartige Funktion wird oft *Schleifenvariante* genannt. Zusätzlich muß man zeigen, daß die Schleife überhaupt ausgeführt wird, d.h., daß kein Laufzeitfehler auftreten kann.

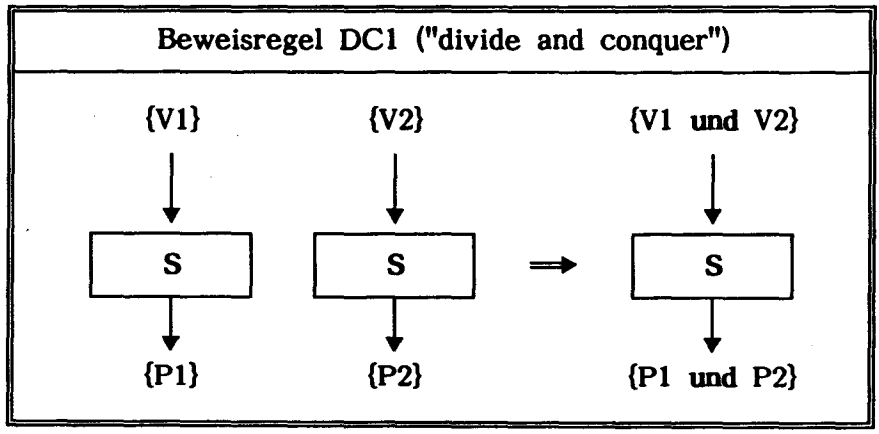
3.3.11 Beweisregel DC1: "Divide and conquer"

Falls

{V1} S {P1} und
{V2} S {P2}

dann gilt, daß

{V1 **und** V2} S {P1 **und** P2} ∎



Die Beweisregel DC1 läßt sich auf offensichtliche Weise für eine beliebig große Anzahl von Termen (P1, P2, P3, P4 usw.) verallgemeinern.

Manchmal tritt in einem Korrektheitsbeweis eine relativ lange Bedingung, z.B. als Nachbedingung eines Programmteils, auf. Gemäß der Beweisregel DC1 kann man eine lange, aus und-verknüpften Termen bestehende Nachbedingung in mehrere kurze Teile zerlegen, die Vorbedingung für jeden Teil ermitteln und diese Vorbedingungen wieder zusammensetzen. Es wird dabei zwar wenig oder keine Arbeit gespart, aber der Beweis und die Manipulation der Ausdrücke bleiben typischerweise übersichtlicher und verständlicher. Die einzelnen Manipulationsschritte sind oft erheblich kürzer und einfacher. Mit der Strategie "divide and conquer" (teilen und erobern) kann man auch ziemlich schwierige, umfangreiche Aufgaben meistern.

Die Beweisregel DC1 gilt für die und-Verknüpfung. Auch für die oder-Verknüpfung gibt es eine gleichartige Beweisregel:

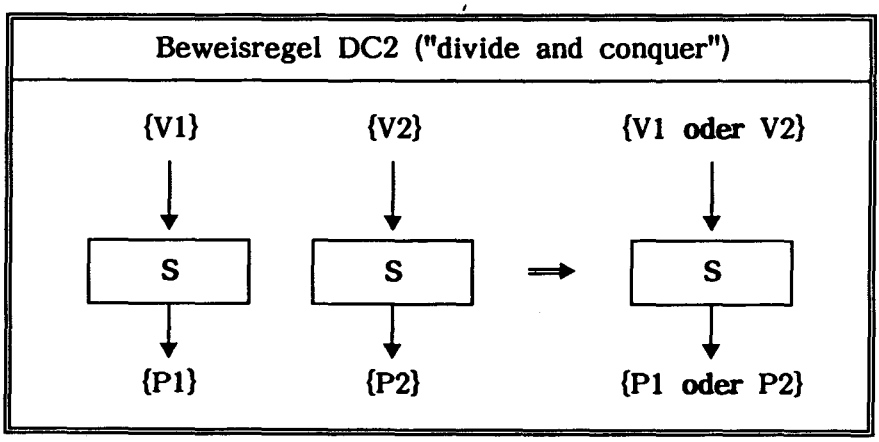
3.3.12 Beweisregel DC2: "Divide and conquer"

Falls

{V1} S {P1} und
{V2} S {P2}

dann gilt, daß

{V1 oder V2} S {P1 oder P2} ■



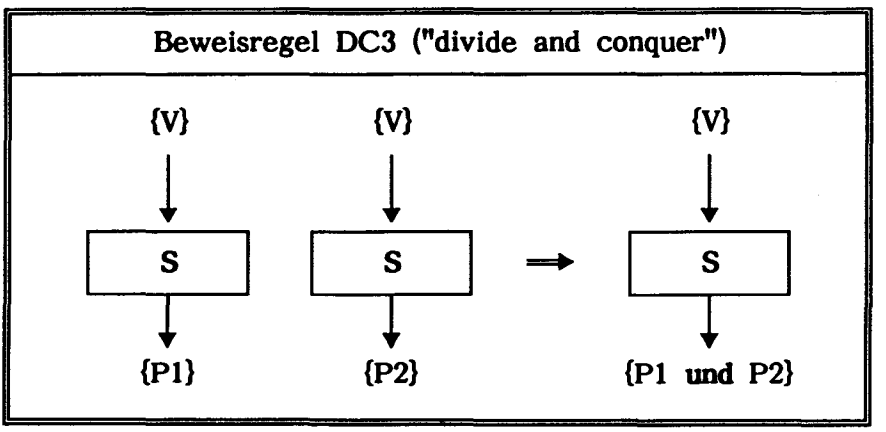
3.3.13 Beweisregel DC3: "Divide and conquer"

Falls

{V} S {P1} und
{V} S {P2}

dann gilt, daß

{V} S {P1 und P2} ■



Die Beweisregel DC3 ist die Beweisregel DC1 mit V=V1=V2.

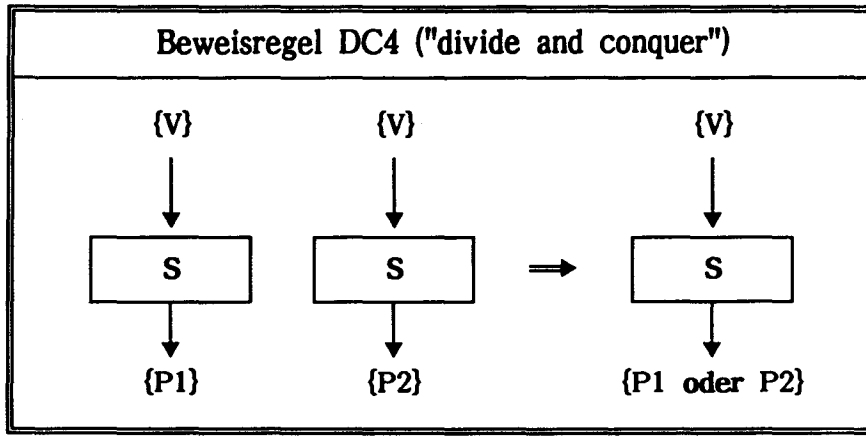
3.3.14 Beweisregel DC4: "Divide and conquer"

Falls

{V} S {P1} und
{V} S {P2}

dann gilt, daß

{V} S {P1 oder P2} ■



Die Beweisregel DC4 ist die Beweisregel DC2 mit V=V1=V2.

3.3.15 Beweisregel U1: Programmteil oder Unterprogramm

Falls die Werte aller Variablen, die in einer Bedingung B vorkommen, durch die Ausführung eines Programmteils S (z.B. eines Unterprogramms) unverändert bleiben, dann gilt, daß

{B} S {B} ■

Falls in der Bedingung B nur Variablen vorkommen, deren Werte vor und nach der Ausführung eines Programmteils S gleich sind, hat die Bedingung B offensichtlich den gleichen Wert vor und nach der Ausführung des fraglichen Programmteils. Falls B vor der Ausführung von S wahr ist, wird B auch danach wahr sein; also gilt {B} S {B}.

3.3.16 Beweisregel U2: Programmteil oder Unterprogramm

Falls in der Bedingung B nur Variablen vorkommen, deren Werte durch die Ausführung eines Programmteils oder Unterprogramms S nicht verändert werden, und falls

{V} S {P}

dann gilt, daß

{V **und** B} S {P **und** B}

sowie

{V **oder** B} S {P **oder** B} ■

Die Beweisregel U2 folgt aus den Beweisregeln U1, DC1 und DC2.

Bei der Anwendung der Beweisregel U2 teilt man eine Nachbedingung in zwei Teile auf. Im einen Teil kommen nur Variablen vor, deren Werte durch die Ausführung des fraglichen Programmteils nicht verändert werden. Dieser Teil der Nachbedingung ist (nach Beweisregel U1) auch eine Vorbedingung. Im anderen Teil der Nachbedingung kommen Variablen vor, deren Werte durch die Ausführung des fraglichen Programmteils verändert werden; für diesen Teil der Nachbedingung wird (z.B. durch Anwendung anderer Beweisregeln oder durch Bezug auf die formale Spezifikation des Programmteils) eine Vorbedingung bestimmt. Die zwei Vorbedingungen werden anschließend kombiniert.

3.3.17 Beweisregel U3: Programmteil oder Unterprogramm

Falls in der Bedingung B nur Variablen vorkommen, deren Werte durch die Ausführung eines Programmteils oder Unterprogramms S nicht verändert werden, und falls

$V \Longrightarrow V1$ und
{V1} S {P1} und
$P1 \Longrightarrow P$

dann gilt, daß

{V **und** B} S {P **und** B}

sowie

{V **oder** B} S {P **oder** B} ■

Die Beweisregel U3 ist eine Kombination der Beweisregeln U2 und B1.

Der von der Wirkung des Programmteils oder Unterprogramms S abhängige Teil der Nachbedingung (P) kann schwächer sein als die von S tatsächlich erfüllte Nachbedingung (P1). Ähnlich kann die vor dem Aufruf tatsächlich geleistete Vorbedingung (V) stärker sein als die für das richtige Funktionieren von S erforderliche Vorbedingung (V1).

3.4 Die Anwendung der Beweisregeln

Wenn man die Korrektheit eines Programmteils oder eines Programms beweisen will, muß der Maßstab für seine Korrektheit – vor allem die Nachbedingung – bekannt sein. Oft ist auch eine Vorbedingung gegeben, in welchem Falle es darum geht, die vorliegende Korrektheitsaussage zu verifizieren, d.h. zu zeigen, daß die gegebene Vorbedingung tatsächlich eine Vorbedingung der gegebenen Nachbedingung bezüglich des Programmteils ist. Manchmal geht es darum, für eine gegebene Nachbedingung und einen gegebenen Programmteil eine Vorbedingung zu bestimmen.

Je nachdem, um welche Art von Anweisung oder Zusammensetzung von Anweisungen es sich handelt und ob eine Vorbedingung gegeben oder zu bestimmen ist, wendet man die eine oder die andere Beweisregel an. Die folgende Tabelle dient der Auswahl der geeigneten Beweisregel(n).

Auswahl einer Beweisregel		
Anweisungsart	Vorbedingung	Beweisregel(n)
Zuweisung	gegeben	Z2
	zu bestimmen	Z1
if-Anweisung	gegeben	IF1 (oder ggf. IF3 oder IF4)
	zu bestimmen	IF2
Folge	gegeben	F1 + ggf. B1
	zu bestimmen	F1
while-Schleife mit Initialisierung	gegeben	W2 + ggf. B1
	zu bestimmen	W2
while-Schleife ohne Initialisierung	gegeben	W1 + ggf. B1
	zu bestimmen	W1 + ggf. B1
Programmteil oder Unterprogramm	gegeben	U2 (ggf. U3)
	zu bestimmen	U2 (ggf. U3)

Die Beweisregeln DC1 bis DC4 können auf alle Anweisungsarten angewendet werden, um umfangreiche Bedingungen in kleinere, einfachere Teilausdrücke zu zerlegen, und zwar sowohl im Falle einer gegebenen als auch im Falle einer zu bestimmenden Vorbedingung.

Eine Übersichtskarte über die praktische Anwendung der Beweisregeln liegt diesem Buch bei. Mit ihrer Hilfe kann zuerst einmal die für die jeweilige vorliegende Aufgabe geeignete(n) Beweisregel(n) ausgewählt werden. Darüber hinaus erläutert die Übersichtskarte die praktische Anwendung der ausgewählten Beweisregel(n) in knapper, komprimierter Form. Nachdem man die Beweisregeln gelernt hat, aber noch nicht im Umgang damit geübt ist, wird die Übersichtskarte als Gedächtnisstütze besonders nützlich sein.

3.5 Anforderungen an die Dokumentation

Aus den Beweisregeln und ihrer Anwendung bei der Korrektheitsbeweisführung ergeben sich einige Anforderungen an die Dokumentation eines Programms oder Unterprogramms.

Vor allem müssen die Vor- und Nachbedingungen in der Dokumentation festgehalten werden. Sie sind unbedingt in der Form von logischen algebraischen Ausdrücken zu formulieren. Sie sollen auch in Worten und ggf. mit Hilfe von Diagrammen erläutert werden, damit der Leser sie möglichst mühelos und schnell lesen und verstehen kann. Siehe die Beispiele in den Abschnitten 4.4.1, 4.4.2, 4.5, 5.1.1, 5.2, 5.3.2 und 5.4.2.

Aus der Dokumentation über ein Unterprogramm soll hervorgehen, welche Variablen die Ausführung des Unterprogramms nicht verändert. Meist wird diese Anforderung durch die Angabe aller Variablen, die durch die Ausführung des fraglichen Unterprogramms verändert werden können, befriedigt. Diese Angaben sind in der Regel Voraussetzung für die Anwendung der Beweisregeln U1, U2 und U3 (siehe die Abschnitte 3.3.15-17, vgl. die darin vorkommende Bedingung B).

Unbedingt zur Dokumentation gehört für *jede* Schleife die Schleifeninvariante.

Ferner sollen Bedingungen angegeben werden, die an ausgewählten wesentlichen Stellen im Unterprogramm erfüllt sein müssen. Es kommen vor allem Stellen zwischen Schleifen und if-Anweisungen sowie vor und nach einer Folge von Zuweisungen in Frage. Bedingungen, die leicht und direkt von anderen in der Dokumentation festgehaltenen Bedingungen abgeleitet werden können, sind wegzulassen. Bedingungen, die Entwurfsentscheidungen darstellen oder die nur auf zeitaufwendige Weise ermittelt werden können, sind in die Dokumentation aufzunehmen.

Solche zur Dokumentation gehörenden Bedingungen sind beim späteren Beweisen der Korrektheit sowie Warten und Pflegen des Unterprogramms nützlich.

Jeder Programmierer, der einen Aufruf auf das fragliche Unterprogramm schreiben will, muß die Vor- und Nachbedingungen wissen sowie welche Variablen durch die Ausführung des Unterprogramms verändert werden. Die Vorbedingung sagt ihm, wofür er vor dem Aufruf sorgen muß. Die Nachbedingung sagt ihm, wovon er nach dem Aufruf ausgehen kann.

4. Analyse: das Beweisen der Korrektheit von Programmen

Bei der Korrektheitsbeweisführung für ein Programmsegment oder ein ganzes Programm geht es fast immer darum, entweder

- zu zeigen, daß die gegebene Vorbedingung tatsächlich eine Vorbedingung der gegebenen Nachbedingung bezüglich des auch gegebenen Programmsegments ist, oder
- für eine gegebene Nachbedingung und ein gegebenes Programmsegment eine Vorbedingung zu bestimmen.

In beiden Fällen ist es meist zweckmäßig, die Aufgabe entsprechend der Programmstruktur zu zerlegen, indem man die für die verschiedenen Anweisungsarten geeigneten Beweisregeln (siehe Tabelle in Abschnitt 3.4) anwendet. Dabei arbeitet man von der übergeordneten Programmstruktur zu immer kleineren Programmteilen hin, bis nur Aufgaben über einzelne Zuweisungen übrigbleiben. Diese löst man unter Verwendung der Beweisregeln Z1 und Z2 für die Zuweisung.

Aus didaktischen Gründen betrachten wir in diesem Kapitel den Beweisführungsprozeß in der umgekehrten Reihenfolge; wir beginnen mit den einfachsten, kleinsten Programmteilen – einzelnen Zuweisungen – und erweitern unsere Kenntnisse über die if-Anweisung und kurze Folgen von Anweisungen bis hin zu längeren Folgen, die eine while-Schleife mit Initialisierung beinhalten.

4.1 Die Zuweisung

Um die Korrektheit von Zuweisungen zu beweisen, werden die Beweisregeln Z1 und Z2 angewendet. Soll eine Vorbedingung ermittelt werden, wendet man die Beweisregel Z1 an. Soll eine vorgegebene Vorbedingung verifiziert werden, wendet man die Beweisregel Z2 an. (Siehe Abschnitte 3.3.2 und 3.3.3.)

Prinzipiell lassen sich die Beweisregeln Z1 und Z2 auf Zuweisungen sowohl zu einfachen (nicht indizierten) Variablen als auch zu indizierten Variablen (Feldvariablen) anwenden. In Beweisen für Zuweisungen zu indizierten Variablen muß man jedoch auf bestimmte Besonderheiten sorgfäl-

tig achten; deshalb wird dieses Thema in einem getrennten Abschnitt (4.1.3) behandelt.

4.1.1 Zuweisungen zu nicht indizierten Variablen

Die Anwendung der Beweisregeln Z1 und Z2 läßt sich am einfachsten anhand von Beispielen erläutern.

Beispiel: Es soll eine Vorbedingung für die folgende Zuweisung und die folgende Nachbedingung bestimmt werden:

{?} x:=z-y {x-y>0}

Da es sich um die Ermittlung einer Vorbedingung bezüglich einer Zuweisung handelt, wird die Beweisregel Z1 angewendet (siehe Tabelle in Abschnitt 3.4). Gemäß der Beweisregel Z1 müssen wir in der Nachbedingung die Variable x durch den Ausdruck (z-y) ersetzen:

{x-y>0}	[Nachbedingung]
{(z-y)-y>0}	[Vorbedingung]
{z-2*y>0}	[Vorbedingung]

Damit haben wir ermittelt, daß

{z-2*y>0} x:=z-y {x-y>0} ■

In diesem Beispiel waren die Klammern um den Ausdruck (z-y) überflüssig. Manchmal sind sie jedoch unbedingt notwendig, wie das nächste Beispiel verdeutlicht.

Beispiel: Eine Vorbedingung soll ermittelt werden:

{?} x:=z-y {y-x>0}

Wir wenden Beweisregel Z1 an, indem wir in der Nachbedingung die Variable x durch den Ausdruck (z-y) ersetzen:

{y-x>0}	[Nachbedingung]
{y-(z-y)>0}	[Vorbedingung]
{y-z+y>0}	[Vorbedingung]
{2*y-z>0}	[Vorbedingung]

Es gilt also, daß

$\{2*y-z>0\}\ x:=z-y\ \{y-x>0\}$ ■

Bei der Verifikation gegebener Vor- und Nachbedingungen wendet man die Beweisregel Z2 an. Der erste Schritt besteht effektiv aus der Ermittlung einer Vorbedingung bezüglich der Zuweisung. Im zweiten Schritt wird geprüft, ob aus der gegebenen Vorbedingung die ermittelte Vorbedingung folgt.

Beispiel: Die Aussage

$\{10<y<x+N \text{ und } 0\leq x<13\}\ x:=x-5\ \{10<y \text{ und } x<8\}$

soll verifiziert werden. Vorab ist zu bemerken, daß die Schreibweise $a\leq b\leq c$ nichts anderes als eine Abkürzung für ($a\leq b$ **und** $b\leq c$) ist. Die zu verifizierende Aussage oben ist folglich gleichbedeutend mit

$\{10<y \text{ und } y<x+N \text{ und } 0\leq x \text{ und } x<13\}$
$x:=x-5\ \{10<y \text{ und } x<8\}$

Gemäß der Beweisregel Z2 wird diese Aussage gelten, falls

$\{10<y \text{ und } y<x+N \text{ und } 0\leq x \text{ und } x<13\}$

$\Longrightarrow \{10<y \text{ und } x<8\}^{x}_{x-5}$

Wir müssen also zeigen, daß

$\{10<y \text{ und } y<x+N \text{ und } 0\leq x \text{ und } x<13\} \Longrightarrow \{10<y \text{ und } x-5<8\}$

oder, äquivalent,

$\{10<y \text{ und } y<x+N \text{ und } 0\leq x \text{ und } x<13\} \Longrightarrow \{10<y \text{ und } x<13\}$

Es ist ersichtlich, daß die gegebene (linke) Vorbedingung stärker ist als die (effektiv durch Anwendung der Beweisregel Z1) ermittelte (rechte) Vorbedingung (siehe Anhang A, Abschnitt A.4, Aufgabe 2). Gemäß der Beweisregel Z2 gilt also die zu verifizierende Aussage. ■

Übung: Löse die folgenden Aufgaben.
1. {?} i:=i+1 {1≤i}
2. {?} sum:=sum+z {sum=x+y+z}
3. {?} x:=5-z {w*y - 2*w² < z} ■

4.1.2 Und wenn es nicht klappt?

(Dieser Abschnitt behandelt fortgeschrittene Aspekte der Korrektheitsbeweisführung für Zuweisungen und darf beim ersten Lesen übersprungen werden.)

Die im Abschnitt 4.1.1 oben vorgestellte Vorgehensweise ist allgemein auf Zuweisungen anwendbar. Falls es einem jedoch nicht gelingt, auf diese Weise zu zeigen, daß aus der gegebenen Vorbedingung die (effektiv durch Anwendung der Beweisregel Z1) ermittelte Vorbedingung folgt, kann es sich nur um zwei Möglichkeiten handeln:
• man hat die logischen Ausdrücke noch nicht auf geeignete Weise umformuliert, um den Beweis zu vervollständigen, oder
• die gegebene, angebliche Vorbedingung ist tatsächlich keine Vorbedingung der gegebenen Nachbedingung bezüglich der gegebenen Zuweisung, d.h., das Programm enthält einen (oder mehrere) Fehler.

Die zweite Möglichkeit muß immer in Betracht gezogen werden, denn nicht wenige Programme, deren Korrektheit bewiesen werden soll, enthalten in der Tat Fehler.

Wenn man den Beweis nicht vervollständigen kann, sollte man versuchen, Werte für die Programmvariablen zu finden, die die gegebene Vorbedingung erfüllen, die ermittelte Vorbedingung jedoch nicht. Solche Werte stellen einen Testfall dar, der die Unstimmigkeit aufzeigen wird.

Nicht selten führt der Versuch, die Korrektheit eines fehlerhaften Programms zu beweisen, unmittelbar zur fehlerhaften Stelle im Programm sowie zu einer Berichtigung des Fehlers.

Beispiel: Dieses Beispiel stammt aus einem fehlerhaften Unterprogramm, das Elemente aus zwei sortierten Feldern zusammenfügen soll. Zu zeigen ist, daß

```
{ia≤na+1 und  (ia≤na und ib>nb
               oder ia≤na und A(ia)≤B(ib)
               oder ib≤nb und A(ia)≤B(ib))}
ia:=ia+1 {ia≤na+1}
```

(was jedoch nicht stimmt).

Gemäß der Beweisregel Z2 wird diese Aussage gelten, falls

$$\{ia \leq na+1 \text{ und } (ia \leq na \text{ und } ib > nb \text{ oder } ia \leq na \text{ und } A(ia) \leq B(ib) \text{ oder } ib \leq nb \text{ und } A(ia) \leq B(ib))\} \Longrightarrow \{ia \leq na+1\}^{ia}_{ia+1}$$

oder, äquivalent,

$$\{ia \leq na+1 \text{ und } (ia \leq na \text{ und } ib > nb \text{ oder } ia \leq na \text{ und } A(ia) \leq B(ib) \text{ oder } ib \leq nb \text{ und } A(ia) \leq B(ib))\} \Longrightarrow \{ia \leq na\}$$

Die ermittelte Vorbedingung {ia≤na} folgt nicht aus der gegebenen Vorbedingung. Die Frage ist, unter welchen Umständen könnte die gegebene Vorbedingung erfüllt sein, die ermittelte jedoch nicht? Die Negation der ermittelten Vorbedingung ist ia>na. Wenn die gegebene Vorbedingung erfüllt sein soll, die ermittelte jedoch nicht, dann muß ia=na+1 gelten. Die ersten zwei **oder**-verknüpften Terme oben werden dann falsch sein; der andere muß also wahr sein. Folglich muß unser Testfall (Gegenbeispiel zu der Korrektheit) die folgenden Bedingungen erfüllen:

ia=na+1
ib≤nb
A(ia)≤B(ib)

Für die Korrektheitsbeweisführung fehlt das Glied ia≤na im dritten **oder**-verknüpften Term der gegebenen Vorbedingung. Dieser Term stammt aus einer if-Bedingung im fraglichen Unterprogramm, aus dem dieses Beispiel entnommen wurde. Die Korrektur des Fehlers kann bereits aus dieser Analyse abgeleitet werden: Es muß das Glied ia≤na in die if-Bedingung eingefügt werden. ■

4.1.3 Zuweisungen zu indizierten Variablen (Feldvariablen)

(Dieser Abschnitt behandelt fortgeschrittene Aspekte der Korrektheitsbeweisführung für Zuweisungen und darf beim ersten Lesen übersprungen werden.)

Prinzipiell wird eine Zuweisung, die einer Feldvariablen einen neuen Wert zuweist, genauso behandelt wie in Abschnitt 4.1.1 beschrieben. Man muß jedoch sehr genau darauf achten, wann ein in der Nachbedingung vorkommender Bezug auf eine Feldvariable durch den Ausdruck zu ersetzen ist und wann nicht. Das folgende Beispiel verdeutlicht diese Problematik.

Beispiel: Eine Vorbedingung ist zu bestimmen:

{?} y(m):=z {y(m)=y(n)}

Es ist klar, daß in der Nachbedingung "y(m)" durch "(z)" zu ersetzen ist (gemäß Beweisregel Z1). Man muß zusätzlich darauf achten, daß die Ausführung der Zuweisung unter Umständen den Wert von y(n) ändern kann, nämlich dann, wenn die Werte von m und n gleich sind und "y(m)" und "y(n)" sich deshalb auf dieselbe Feldvariable beziehen. In diesem Fall muß man auch "y(n)" durch z ersetzen, im anderen Fall, nicht.

Falls m=n gilt, ist also die gesuchte Vorbedingung {z=z} oder einfach die logische Konstante wahr. Falls m≠n gilt, dann ist die gesuchte Vorbedingung {z=y(n)}. Anders ausgedrückt, die gesuchte Vorbedingung V ist

V = wahr, falls m=n
  = [z=y(n)], falls m≠n

Ein anderer Ausdruck für die gleiche Bedingung V ist (siehe Anhang A, Abschnitt A.4, Aufgabe 1 und die Lösung dazu in Anhang B)

V = [(m=n) **und** wahr **oder** (m≠n) **und** z=y(n)]
  = [m=n **oder** z=y(n)]

(Siehe Anhang A, Abschnitt A.3, Identitäten 10 und 17.) ■

Oft kann man durch Umformulieren der Nachbedingung Bezüge auf Feldvariablen derart trennen, daß gewisse Bezüge auf Feldvariablen immer ersetzt werden müssen und andere

nie. Insbesondere wenn **und**- oder **oder**-Reihen auftreten, ist dies möglich.

Beispiel: Nach der Ausführung der Zuweisung D(j):=A(k) sollen die Feldvariablen D(1) bis D(j) sortiert sein, d.h. $D(1) \leq D(2) \ldots \leq D(j)$. Gesucht ist die Vorbedingung, die dieses sicherstellt:

$$\{?\}\ D(j):=A(k)\ \{\mathbf{und}_{i=1}^{j-1}\ D(i) \leq D(i+1)\}$$

Innerhalb der **und**-Reihe ist i immer kleiner als j, also gilt $i \neq j$. Die Variable D(i) ist nie identisch mit D(j), wird also nicht durch "A(k)" ersetzt. Die Variable D(i+1) bezieht sich auf D(j) wenn i=j-1, ansonsten nicht. Es liegt deshalb nahe, den einen Term aus der **und**-Reihe herauszunehmen (siehe Anhang A, Abschnitt A.5). Die vorliegende Aufgabe kann dann wie folgt geschrieben werden.

$$\{?\}\ D(j):=A(k)$$

$$\{j<2\ \mathbf{oder}\ j \geq 2\ \mathbf{und}\ D(j-1) \leq D(j)\ \mathbf{und}_{i=1}^{j-2}\ D(i) \leq D(i+1)\}$$

In dieser Form enthält die Nachbedingung genau einen Bezug auf D(j). Alle anderen Bezüge auf Feldvariablen D(.) weisen Indexwerte auf, die immer kleiner als – und damit ungleich – j sind. Die gesuchte Vorbedingung kann deshalb durch Ersetzen des einen Bezugs auf D(j) durch A(k) abgeleitet werden:

$$\{j<2\ \mathbf{oder}\ j \geq 2\ \mathbf{und}\ D(j-1) \leq A(k)\ \mathbf{und}_{i=1}^{j-2}\ D(i) \leq D(i+1)\}\ \blacksquare$$

Diese Vorbedingung besagt, daß entweder
- das Feld D leer ist (j<2) oder
- der Wert von A(k) mindestens so groß ist wie der letzte Wert im Feld D und die bereits in D stehenden Werte sortiert sind (wobei ein Feld mit nur einem Element (j=2) sortiert ist).

Für eine detaillierte Erläuterung sowie eine allgemein anwendbare Lösung für diese Problematik, siehe [Baber, 1987, Seiten 72, 73 und 140 ff.].

4.2 Die if-Anweisung

Für die if-Anweisung stehen mehrere Beweisregeln zur Verfügung. Die wichtigsten sind IF1 und IF2. Die anderen, IF3 und IF4, sind weniger aussagekräftige Versionen von IF1. Sie sind manchmal von praktischer Bedeutung wegen der einfacheren Form der darin vorkommenden logischen Ausdrücke. (Siehe die Abschnitte 3.3.4 bis 3.3.7.)

Wenn es darum geht, eine Vorbedingung für eine gegebene Nachbedingung und eine gegebene if-Anweisung zu bestimmen, kommt nur die Beweisregel IF2 in Betracht. Durch ihre Anwendung wird die Aufgabe effektiv in Unteraufgaben zerlegt, entsprechend der Struktur der if-Anweisung. Die Vorbedingungen bezüglich der then- und else-Zweige der if-Anweisung werden ermittelt und auf geeignete Weise (die die Beweisregel IF2 genau angibt) kombiniert. Das Ergebnis ist die gesuchte Vorbedingung bezüglich der gesamten if-Anweisung.

Bei der Verifikation gegebener Vor- und Nachbedingungen für eine gegebene if-Anweisung kommt in erster Linie die Beweisregel IF1 in Betracht. Falls die fraglichen Ausdrücke der Form nach denjenigen der Beweisregel IF3 bzw. IF4 entsprechen, kann sie angewendet werden. In jedem Falle wird die zu verifizierende Korrektheitsaussage über die gesamte if-Anweisung in zwei Aussagen zerlegt: eine über den then-Zweig und die andere über den else-Zweig der if-Anweisung. Die daraus resultierenden Aussagen müssen wiederum durch Anwendung der geeigneten Beweisregeln verifiziert werden.

Beispiel 1: Eine Vorbedingung soll für die folgende Nachbedingung und die folgende if-Anweisung bestimmt werden.

{?} **if** x<0 **then** y:=-x **else** y:=x **endif** {y>0}

Es muß die Beweisregel IF2 angewendet werden. Gemäß der Beweisregel IF2 teilt sich unsere Aufgabe in drei Unteraufgaben auf: Bestimme V1 und V2 derart, daß

{V1?} y:=-x {y>0}
{V2?} y:=x {y>0}

und bilde (und ggf. vereinfache) den Ausdruck

{(V1 **und** x<0) **oder** (V2 **und nicht** x<0)}

der die gesuchte Vorbedingung V ist.

V1, die Vorbedingung bezüglich einer Zuweisung, soll ermittelt werden; wir müssen deshalb die Beweisregel Z1 anwenden. Wir ersetzen in der Nachbedingung die Variable y durch den Ausdruck (-x) und erhalten die Vorbedingung

$$V1 = \{(-x)>0\} = \{x<0\}$$

Um V2 abzuleiten, ersetzen wir in der Nachbedingung die Variable y durch den Ausdruck (x) und erhalten die Vorbedingung

$$V2 = \{x>0\}$$

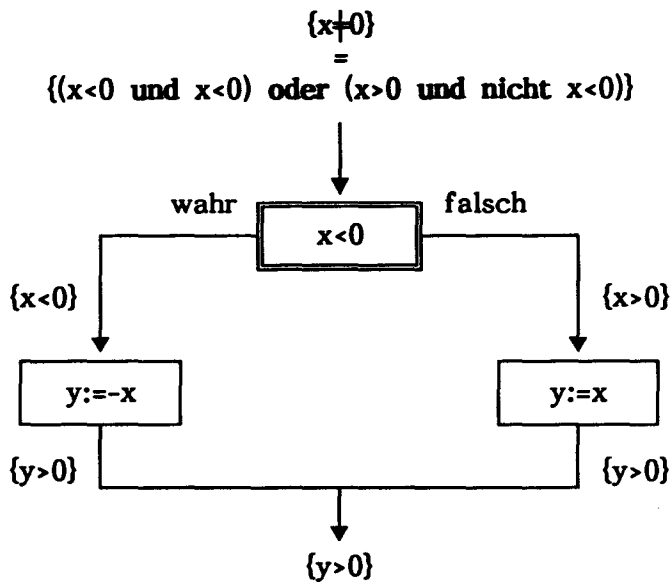
Anschließend bilden wir gemäß der Beweisregel IF2 die gesuchte Vorbedingung V bezüglich der gesamten if-Anweisung:

$$V = \{(V1 \textbf{ und } x<0) \textbf{ oder } (V2 \textbf{ und nicht } x<0)\}$$
$$= \{(x<0 \textbf{ und } x<0) \textbf{ oder } (x>0 \textbf{ und } x\geq 0)\}$$

Dieser Ausdruck für V kann vereinfacht werden (siehe Anhang A, Abschnitt A.3):

$$V = \{(x<0) \textbf{ oder } (x>0)\}$$
$$= \{x\neq 0\} \blacksquare$$

Beispiel 1:
Bestimmung einer Vorbedingung für eine if-Anweisung



Das folgende, etwas komplexere Beispiel wird auf die gleiche Weise gelöst.

Beispiel 2: Eine Vorbedingung soll bestimmt werden:

{?} **if** x<0 **then** y:=x **else** y:=x-2 **endif** {-1≤y≤4}

Gemäß der Beweisregel IF2 zerlegen wir diese Aufgabe in drei Teile:

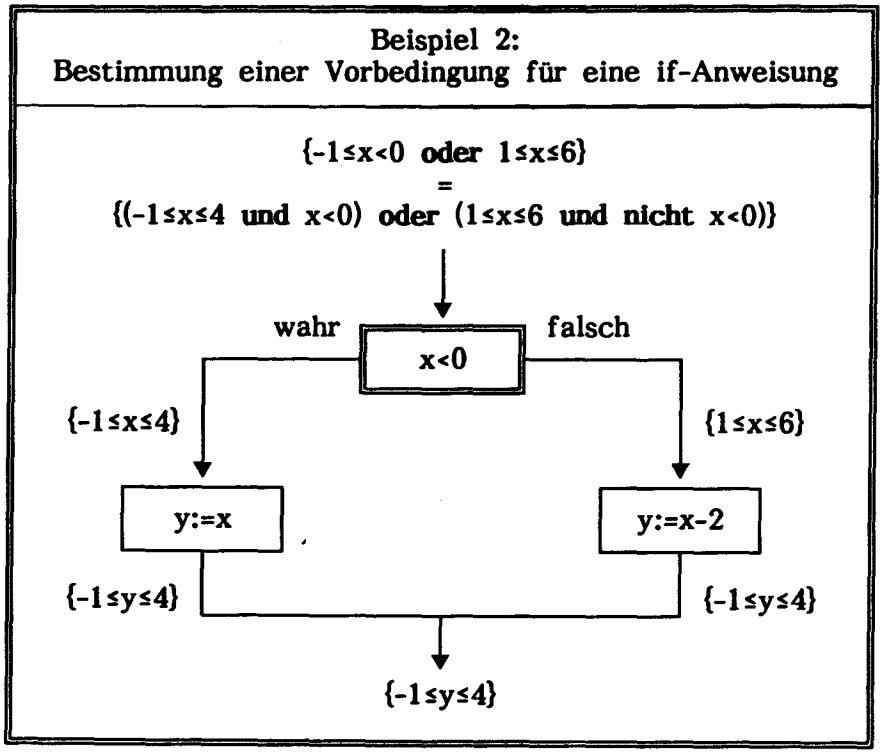
{V1?} y:=x {-1≤y≤4}
{V2?} y:=x-2 {-1≤y≤4}
V = {(V1 **und** x<0) **oder** (V2 **und nicht** x<0)}

Durch Anwendung der Beweisregel Z1 ermitteln wir V1 und V2:

V1 = {-1≤x≤4}
V2 = {-1≤(x-2)≤4} = {1≤x≤6}

Diese Ausdrücke setzen wir für V1 und V2 in den Ausdruck für V oben ein und erhalten als die gesuchte Vorbedingung

V = {(-1≤x≤4 **und** x<0) **oder** (1≤x≤6 **und** x≥0)}
= {-1≤x<0 **oder** 1≤x≤6} ■



Beispiel 3: Die Aussage

{ia≤na **oder** ia≤na+1 **und** ib≤nb}
if ib>nb **oder** ia≤na **then** ia:=ia+1 **else** ib:=ib+1 **endif**
{ia≤na+1}

soll verifiziert werden.

Gemäß der Beweisregel IF1 wird diese Aussage gelten, falls die folgenden zwei Aussagen zutreffen:

{(ia≤na **oder** ia≤na+1 **und** ib≤nb) [1]
und (ib>nb **oder** ia≤na)}
ia:=ia+1 {ia≤na+1}

{(ia≤na **oder** ia≤na+1 **und** ib≤nb) [2]
und nicht (ib>nb **oder** ia≤na)}
ib:=ib+1 {ia≤na+1}

Durch die Anwendung der Beweisregel IF1 haben wir eine auf eine if-Anweisung bezogene Verifikationsaufgabe in zwei auf Zuweisungen bezogene Verifikationsaufgaben zerlegt.

Aussage 1: Gemäß der Beweisregel Z2 wird die Aussage 1 oben gelten, falls

{(ia≤na **oder** ia≤na+1 **und** ib≤nb) **und** (ib>nb **oder** ia≤na)}
⟹ {ia≤na}

Wir vereinfachen den Ausdruck für die linke Bedingung in der Implikation oben (siehe Anhang A, Abschnitt A.3):

{(ia≤na **oder** ia≤na+1 **und** ib≤nb)
und (ib>nb **oder** ia≤na)}

= {ia≤na **und** ib>nb
oder ia≤na **und** ia≤na
oder ia≤na+1 **und** ib≤nb **und** ib>nb
oder ia≤na+1 **und** ib≤nb **und** ia≤na}

= {ia≤na **und** ib>nb
oder ia≤na
oder falsch
oder ia≤na **und** ib≤nb}

= {ia≤na}

Damit gilt die Aussage 1 oben.

Aussage 2: Gemäß der Beweisregel Z2 wird die Aussage 2 oben gelten, falls

{(ia≤na **oder** ia≤na+1 **und** ib≤nb)
und nicht (ib>nb **oder** ia≤na)}
⟹ {ia≤na+1}

Es ist eigentlich ersichtlich, daß die rechte Bedingung in der Implikation oben {ia≤na+1} allein aus dem ersten Teil der linken Bedingung in der Implikation folgt. Formal (siehe Anhang A, Abschnitt A.3 und Abschnitt A.4, Aufgabe 2),

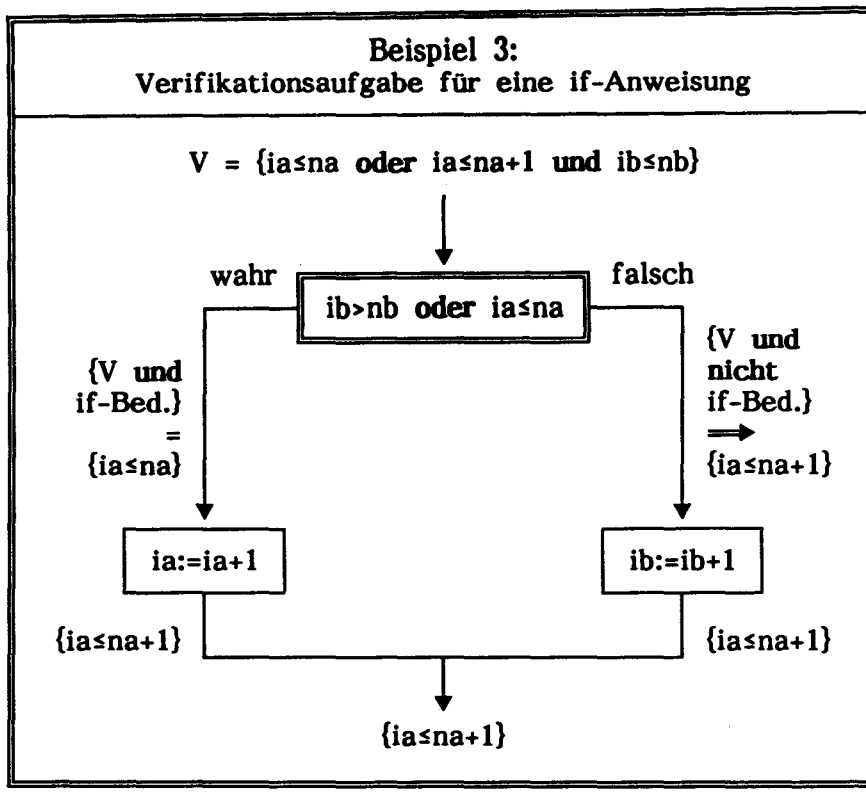
{(ia≤na **oder** ia≤na+1 **und** ib≤nb)
und nicht (ib>nb **oder** ia≤na)}

= {(ia≤na **oder** ia≤na+1 **und** ib≤nb) **und** ...}

⟹ {ia≤na **oder** ia≤na+1 **und** ib≤nb}

⟹ {ia≤na **oder** ia≤na+1 **und** ib≤nb **oder** ia≤na+1}

= {ia≤na+1} ■



Übung: Löse die folgenden Aufgaben.
1. {?} **if** x<0 **then** y:=-x **else** y:=x **endif** {y>0}
2. {?} **if** x<0 **then** y:=-x **else** y:=x **endif** {y≥0}
3. {?} **if** x<0 **then** y:=-x **else** y:=x **endif** {y<0}
4. {?} **if** x<0 **then** y:=-x **else** y:=x **endif** {y≤0}
5. {3≤|x|≤4} **if** x<0 **then** y:=-x **else** y:=x **endif** {2≤y≤4} ? ∎

4.3 Die Folge von Anweisungen

Um die Korrektheit einer Folge von Anweisungen nachzuweisen, wendet man die Beweisregel F1 an (siehe Abschnitt 3.3.8). Man fängt mit der Nachbedingung an und ermittelt, rückwärts durch die Folge arbeitend, eine Vorbedingung für jede Anweisung in der Folge. Die auf diese Weise ermittelte Vorbedingung für die erste Anweisung in der Folge ist gleichzeitig eine Vorbedingung bezüglich der ganzen Folge.

Beispiel: Gesucht ist eine Vorbedingung bezüglich der gegebenen Folge von Zuweisungen

{V?} gr:=gr-1; gl:=gl-1 {il-1≤gl<gr≤ig}

Durch die Anwendung der Beweisregel F1 zerlegen wir diese Aufgabe in zwei Aufgaben über Zuweisungen:

{V?} gr:=gr-1 {P1} [P1 noch zu bestimmen]

und

{P1?} gl:=gl-1 {il-1≤gl<gr≤ig}

Weil P1 noch unbekannt ist, können wir die Aufgabe für die erste Zuweisung noch nicht lösen. Wir müssen mit der letzten Zuweisung anfangen. Durch Anwendung der Beweisregel Z1 ermitteln wir P1, indem wir in der Nachbedingung die Variable gl durch den Ausdruck (gl-1) ersetzen:

P1 = {il-1≤gl-1<gr≤ig}

Dadurch wird die Aufgabe für die erste Zuweisung wie folgt:

{V?} gr:=gr-1 {il-1≤gl-1<gr≤ig}

Durch Anwendung der Beweisregel Z1 ermitteln wir die gesuchte Vorbedingung, indem wir in P1, die jetzt die Nachbedingung der ersten Zuweisung ist, die Variable gr durch den Ausdruck (gr-1) ersetzen:

V = {il-1≤gl-1<gr-1≤ig}

Zusammenfassend (und wiederholend) haben wir gezeigt, daß

{il-1≤gl-1<gr-1≤ig} gr:=gr-1 {il-1≤gl-1<gr≤ig}

und

{il-1≤gl-1<gr≤ig} gl:=gl-1 {il-1≤gl<gr≤ig}

woraus gemäß der Beweisregel F1 folgt, daß

{il-1≤gl-1<gr-1≤ig} gr:=gr-1; gl:=gl-1 {il-1≤gl<gr≤ig} ■

Übung: Zeige, daß die folgende Aussage über die Vor- und Nachbedingungen bezüglich der gegebenen Folge von Zuweisungen gilt.

1. {0≤N} i:=0; j:=0

$$\{\text{und}_{k=0}^{j-1} \text{ (nicht und}_{a=0}^{N-1} D(k+a)=K(a))$$

und (j>M-N oder j≤M-N $\text{und}_{a=0}^{i-1}$ D(j+a)=K(a))

und 0≤j und 0≤i≤N} ■

4.4 Die while-Schleife

Um die Korrektheit einer while-Schleife zu beweisen, wendet man entweder die Beweisregel W1 oder die Beweisregel W2 an, je nachdem, ob es sich um eine while-Schleife ohne bzw. mit Initialisierung handelt. Fast immer hat man mit einer Schleife mit Initialisierung zu tun, deshalb betrachten wir hier die Anwendung der Beweisregel W2, die ohnehin die Beweisregel W1 beinhaltet. (Siehe die Abschnitte 3.3.9 und 3.3.10.)

Durch Anwendung der Beweisregel W2 zerlegt man die Aufgabe (nämlich die Korrektheit der Schleife mit Initialisierung zu beweisen) in drei Teilaufgaben: (1) die Korrektheit der Initialisierung, (2) die Korrektheit des Schleifenkerns und (3) die endgültige Wahrheit der Nachbedingung zu beweisen. Jede dieser drei Aufgaben wird unter Verwendung der entsprechenden Beweisregel bzw. durch Umformen der fraglichen algebraischen Ausdrücke gelöst.

Schließlich muß man zeigen, daß die Schleife terminieren wird, d.h. insbesondere, daß der Schleifenkern nur endlich viele Male ausgeführt wird.

Bevor man die Beweisregel W2 anwenden kann, muß eine geeignete Schleifeninvariante bekannt sein. Die Festlegung einer Schleifeninvariante ist eigentlich eine Konstruktionsentscheidung. Die Schleifeninvariante soll deshalb in der Dokumentation eines gegebenen Programmteils vorliegen. Da sie jedoch noch nicht immer als selbstverständlicher Bestandteil der Dokumentation mit dem Programm mitgeliefert

wird, muß derjenige, der die Korrektheit eines vorliegenden Programms beweisen will, diesen Konstruktionsschritt manchmal nachholen.

Im Abschnitt 4.4.1 unten wird die Korrektheit einer while-Schleife bewiesen, für die eine Schleifeninvariante bekannt ist. Im Abschnitt 4.4.2 werden wir sehen, wie man die Schleifeninvariante hätte bestimmen können, falls sie nicht vorgelegen hätte.

4.4.1 Korrektheitsbeweis bei bekannter Schleifeninvariante

Beispiel: Die Korrektheit des folgenden Unterprogramms, das ein Feld A nach dem Wert der Variablen x absucht, ist nachzuweisen.

```
k:=1
while k≤n und A(k)≠x do k:=k+1 endwhile
```

Die spezifizierte Vorbedingung ist

$n \in Z$ **und** $0 \leq n$

wobei Z die Menge aller Ganzzahlen (0, 1, -1, 2, -2, ...) ist. D.h., die Aussage "$n \in Z$" ist gleichbedeutend mit "der Wert der Variablen n ist eine Ganzzahl".

Die Variable n gibt an, wieviele Elemente das abzusuchende Feld A enthält.

Die gegebene Nachbedingung ist

$n \in Z$ **und** $k \in Z$ **und** $1 \leq k \leq n+1$ [Wertebereich von k]

und $\bigwedge_{i=1}^{k-1} A(i) \neq x$ [alle Elemente vor dem k-ten $\neq$ x]

und ($k \leq n$ **und** $A(k)=x$ [A(k) = x]

oder $k=n+1$) [kein Element von A = x]

Die Korrektheitsaussage, die zu beweisen ist, lautet:

{nεZ und 0≤n}
k:=1
while k≤n und A(k)≠x **do** k:=k+1 **endwhile**
{nεZ und kεZ und 1≤k≤n+1 $\text{und}_{i=1}^{k-1}$ A(i)≠x
und (k≤n und A(k)=x oder k=n+1)}

Der Programmierer legte als Schleifeninvariante I fest

nεZ und kεZ und 1≤k≤n+1 [Wertebereich von k]

$\text{und}_{i=1}^{k-1}$ A(i)≠x [alle Elemente vor dem k-ten ≠ x]

Da es sich um eine while-Schleife mit Initialisierung handelt, wenden wir die Beweisregel W2 an. Danach gilt die Korrektheitsaussage oben, falls die folgenden drei Aussagen zutreffen:

{nεZ und 0≤n} k:=1 {I} [1]

{I und k≤n und A(k)≠x} k:=k+1 {I} [2]

{I und nicht (k≤n und A(k)≠x)} [3]
⟹
{nεZ und kεZ und 1≤k≤n+1 $\text{und}_{i=1}^{k-1}$ A(i)≠x
und (k≤n und A(k)=x oder k=n+1)}

Aussage 1: Hier handelt es sich um eine einfache Zuweisung. Diese Aussage kann wie in Abschnitt 4.1.1 erläutert verifiziert werden. Gemäß der Beweisregel Z2 wird die Aussage 1 gelten, falls

$\{n \varepsilon Z \text{ und } 0 \leq n\} \Longrightarrow \{I^k_1\}$

Vollständig ausgeschrieben wird dieser Ausdruck

{nεZ und 0≤n}
⟹ $\{n \varepsilon Z \text{ und } 1 \varepsilon Z \text{ und } 1 \leq 1 \leq n+1 \text{ und}_{i=1}^{0} A(i) \neq x\}$

Die **und**-Reihe ist leer und hat deshalb den Wert wahr (siehe Anhang A, Abschnitt A.5). Die rechte Bedingung der Implikation oben läßt sich auf die gegebene Vorbedingung (die linke Bedingung der Implikation oben) vereinfachen; die Aussage 1 gilt also.

Aussage 2: Vollständig ausgeschrieben lautet die zu beweisende Aussage 2:

$\{n \in Z \text{ und } k \in Z \text{ und } 1 \le k \le n+1 \text{ und}_{i=1}^{k-1} A(i) \neq x$
$\text{und } k \le n \text{ und } A(k) \neq x\}$
$k := k+1$
$\{n \in Z \text{ und } k \in Z \text{ und } 1 \le k \le n+1 \text{ und}_{i=1}^{k-1} A(i) \neq x\}$

Die Vorbedingung in dieser Aussage kann vereinfacht werden:

$= \{n \in Z \text{ und } k \in Z \text{ und } 1 \le k \le n \text{ und}_{i=1}^{k-1} A(i) \neq x \text{ und } A(k) \neq x\}$

$= \{n \in Z \text{ und } k \in Z \text{ und } 1 \le k \le n \text{ und}_{i=1}^{k} A(i) \neq x\}$

Aussage 2 lautet also

$\{n \in Z \text{ und } k \in Z \text{ und } 1 \le k \le n \text{ und}_{i=1}^{k} A(i) \neq x\}$
$k := k+1$
$\{n \in Z \text{ und } k \in Z \text{ und } 1 \le k \le n+1 \text{ und}_{i=1}^{k-1} A(i) \neq x\}$

Gemäß der Beweisregel Z2 wird diese Aussage gelten, falls

$\{n \in Z \text{ und } k \in Z \text{ und } 1 \le k \le n \text{ und}_{i=1}^{k} A(i) \neq x\}$
$\Longrightarrow$
$\{n \in Z \text{ und } k \in Z \text{ und } 1 \le k \le n+1 \text{ und}_{i=1}^{k-1} A(i) \neq x\}^{k}_{k+1}$

oder, äquivalent,

$\{n \in Z \text{ und } k \in Z \text{ und } 1 \le k \le n \text{ und}_{i=1}^{k} A(i) \neq x\}$
$\Longrightarrow$
$\{n \in Z \text{ und } k \in Z \text{ und } 0 \le k \le n \text{ und}_{i=1}^{k} A(i) \neq x\}$

Es ist offensichtlich, daß die rechte Bedingung der Implikation oben aus der linken Bedingung folgt. Damit gilt die Aussage 2.

Formal:

$$\{n\in Z \text{ und } k\in Z \text{ und } 1\le k\le n \text{ und}_{i=1}^{k} A(i)\neq x\}$$

$$= \quad (1\le k) \text{ und } \{n\in Z \text{ und } k\in Z \text{ und } 0\le k\le n \text{ und}_{i=1}^{k} A(i)\neq x\}$$

$$\Longrightarrow \{n\in Z \text{ und } k\in Z \text{ und } 0\le k\le n \text{ und}_{i=1}^{k} A(i)\neq x\}$$

Aussage 3: Vollständig ausgeschrieben lautet diese Aussage:

$$\{n\in Z \text{ und } k\in Z \text{ und } 1\le k\le n+1 \text{ und}_{i=1}^{k-1} A(i)\neq x$$

$$\text{und nicht } (k\le n \text{ und } A(k)\neq x)\}$$

$$\Longrightarrow$$

$$\{n\in Z \text{ und } k\in Z \text{ und } 1\le k\le n+1 \text{ und}_{i=1}^{k-1} A(i)\neq x$$

$$\text{und } (k\le n \text{ und } A(k)=x \text{ oder } k=n+1)\}$$

Der erste Ausdruck oben läßt sich anders schreiben (siehe Anhang A, Abschnitt A.3, Identitäten 19 und 17, und Abschnitt A.4, Aufgabe 3):

$$\{n\in Z \text{ und } k\in Z \text{ und } 1\le k\le n+1 \text{ und}_{i=1}^{k-1} A(i)\neq x$$

$$\text{und } (k>n \text{ oder } k\le n \text{ und } A(k)=x)\}$$

$$= \{n\in Z \text{ und } k\in Z \text{ und } 1\le k\le n+1 \text{ und}_{i=1}^{k-1} A(i)\neq x$$

$$\text{und } (k\le n \text{ und } A(k)=x \text{ oder } k=n+1)\}$$

Dieser Ausdruck ist der zweite Ausdruck in der Aussage 3; damit ist die Aussage 3 bewiesen.

Vollständige Korrektheit: Wir haben die partielle Korrektheit des gegebenen Unterprogramms bewiesen. Um die vollständige Korrektheit zu beweisen, müssen wir noch zeigen, daß das Programm ohne Laufzeitfehler zu Ende ausgeführt wird ("terminiert") bzw. zusätzliche Bedingungen ermitteln, die sicherstellen, daß es terminiert.

Dazu gehört der Beweis, daß der Schleifenkern nur endlich viele Male ausgeführt wird. Im vorliegenden Falle – wie typischerweise – ist dieser Schritt relativ einfach. Bei

jeder Ausführung des Schleifenkerns wird der Wert von k um 1 erhöht. Sobald k>n endet die Schleife.

Es muß noch gezeigt werden, daß jede Anweisung im Programm jedesmal mit einem definierten Ergebnis ausgeführt wird, d.h., daß kein Laufzeitfehler auftreten kann. Im letzten Detail wird ein solcher Beweis vom Zielsystem abhängen. Trotzdem kann man hinsichtlich der Terminierung weitgehende, allgemein gültige Aussagen machen bzw. Richtlinien angeben (siehe Kapitel 2, Die Ausführung von Programmanweisungen: Wirkungen und Axiome). Strenggenommen müßte auch noch gezeigt werden, daß alle Anweisungen syntaktisch richtig sind, aber es wird hier auf eine derartig programmiersprachen-spezifische Prüfung – die von den meisten Zielsystemen sowieso automatisch und vollständig durchgeführt wird – verzichtet.

Die zwei Zuweisungen k:=... werden immer definierte Ergebnisse liefern, wenn der (ggf. automatisch) deklarierte Wertebereich für k alle ganzen Zahlen von 1 bis n+1 einschließlich umfaßt (vgl. Schleifeninvariante). Falls z.B. k und n als Variablen des gleichen Typs deklariert worden sind, darf n nicht den höchsten Wert annehmen.

Der erste Teil der while-Bedingung wird immer ausgewertet werden können, wenn die Variablen k und n deklariert sind und ihre Werte miteinander auf ≤ vergleichbar sind. Da k und n nur ganzzahlige Werte annehmen (siehe die Schleifeninvariante), wird diese Bedingung erfüllt sein. (Diese Variablen müssen jedoch nicht als ganzzahlige Variablen deklariert sein.)

Der zweite Teil der while-Bedingung wird ausgewertet werden können, wenn die Variablen A(1), ... A(n) und x (ggf. automatisch) deklariert und ihre Werte miteinander auf Gleichheit vergleichbar sind. Ferner muß berücksichtigt werden, daß A(n+1) beim letzten Schleifendurchlauf angesprochen wird, denn die Schleifeninvariante läßt k=n+1 zu. Beim Vergleich der (evtl. nicht deklarierten) Variablen A(n+1) wird jedoch der Term k≤n falsch sein, so daß der Wert von A(n+1) nicht maßgeblich ist. Es muß jedoch darauf geachtet werden, wie das ausführende System einen Ausdruck der Form "falsch und nicht definiert" berechnet (siehe die Bemerkungen über das Auswerten von Ausdrücken im Abschnitt 2.1). Auf einem System, das diesen Ausdruck als falsch auswertet, wird dieses Unterprogramm korrekt laufen. Auf einem System, das den Wert dieses Ausdrucks als nicht definiert betrachtet, wird die Ausführung dieses Unterprogramms

mit einer entsprechenden Fehlermeldung abnormal abgebrochen. Im Kontext eines solchen Systems ist dieses Unterprogramm nicht vollständig korrekt.

Eine ausführliche Vorbedingung, die die vollständige Korrektheit des gegebenen Unterprogramms sicherstellt, lautet also:

- Die Variable n ist deklariert und hat einen nicht negativen ganzzahligen Wert und
- die Variable k ist deklariert oder wird nach Bedarf automatisch deklariert und
- der (ggf. automatisch) deklarierte Wertebereich für k umfaßt alle ganzen Zahlen von 1 bis n+1 einschließlich und
- die Variablen A(1), ... A(n) (und ggf. A(n+1), siehe oben) und x sind deklariert und ihre Werte sind miteinander auf Gleichheit vergleichbar.

Zusätzliche Bedingungen könnten sich aus implementierungsspezifischen Eigenschaften des verwendeten Programmiersprachensystems ergeben (z.B., daß k als Feldindexvariable oder ganzzahliger Typ deklariert werden muß usw.). ■

4.4.2 Korrektheitsbeweis bei nicht bekannter Schleifeninvariante

In Abschnitt 4.4.1 wurde angenommen, daß eine für die Korrektheitsbeweisführung geeignete Schleifeninvariante vorgegeben wurde. Nicht selten wird man jedoch vor der Aufgabe stehen, einen Korrektheitsbeweis für eine Schleife zu erstellen, obwohl die Schleifeninvariante unbekannt ist. In einem solchen Fall muß man selbst eine Schleifeninvariante bestimmen.

Die Schleifeninvariante ist eine Verallgemeinerung der Vor- und Nachbedingungen der while-Schleife (siehe Abschnitt 3.3.9). D.h., sie muß sowohl vor als auch nach der Ausführung der Schleife wahr sein.

Eine Schleifeninvariante, die für eine gegebene Schleife mit Initialisierung und eine gegebene Nachbedingung geeignet ist, kann oft wie folgt bestimmt werden: Man betrachtet den Wert der Nachbedingung am Anfang der Schleife, d.h. gleich vor der Ausführung der Schleife, und fragt sich, wie die Nachbedingung geändert werden müßte, um auch anfangs wahr zu sein. Glieder, die der anfänglichen Wahrheit im Wege stehen, sind Kandidaten zum Ändern oder zum Weglassen.

Betrachten wir wieder das Beispiel des Abschnitts 4.4.1, jedoch ohne die dort vorgegebene Schleifeninvariante.

Beispiel: Die gegebene Nachbedingung ist:

$n \in Z$ **und** $k \in Z$ **und** $1 \leq k \leq n+1$ [Wertebereich von k]

und$_{i=1}^{k-1}$ $A(i) \neq x$ [alle Elemente vor dem k-ten $\neq$ x]

und ($k \leq n$ **und** $A(k)=x$ [A(k) = x]

oder $k=n+1$) [kein Element von A = x]

Die Initialisierung der Schleife stellt sicher, daß gleich vor der Ausführung der Schleife k=1. Der Wert der Nachbedingung vor der Ausführung der Schleife ist also

$n \in Z$ **und** $1 \in Z$ **und** $1 \leq 1 \leq n+1$

und$_{i=1}^{0}$ $A(i) \neq x$

und ($1 \leq n$ **und** $A(1)=x$

oder $1=n+1$)

Nach Vereinfachung erhält man

$n \in Z$ **und** $0 \leq n$

und ($1 \leq n$ **und** $A(1)=x$

oder $0=n$)

Die gegebene Vorbedingung für das Unterprogramm gewährleistet die Wahrheit der ersten Zeile oben. Es wird am Anfang der Schleife nicht bekannt sein, ob A(1)=x oder nicht. Ferner kann der Wert von n Null oder positiv sein. Die letzten zwei Zeilen der Nachbedingung verhindern also, daß die Nachbedingung anfangs wahr ist. Man kann die Nachbedingung dadurch verallgemeinern (schwächen), daß man die letzten zwei Zeilen (die einen und-verknüpften Term der Nachbedingung bilden) wegläßt. Dadurch wird die Schleifeninvariante I

$n \in Z$ **und** $k \in Z$ **und** $1 \leq k \leq n+1$ [Wertebereich von k]

und$_{i=1}^{k-1}$ $A(i) \neq x$ [alle Elemente vor dem k-ten $\neq$ x]

(Siehe das Beispiel im Abschnitt 4.4.1.) ■

Das Thema Bestimmung der Schleifeninvariante wird in Kapitel 5 über Programmkonstruktion näher behandelt.

4.5 Anwendung der "divide and conquer"-Beweisregeln

Manchmal tritt in einem Beweis ein umfangreicher Ausdruck auf. Obwohl das prinzipiell kein grundsätzliches Problem aufwirft, wird die Beweisführung deswegen manchmal unübersichtlich. Ferner kann die manuelle Manipulation der algebraischen Ausdrücke dadurch erschwert werden. Abhilfe schafft die Anwendung der Beweisregeln DC1 bis DC4 ("divide and conquer", etwa "teile und erobere").

Beispiel: Wir betrachten die folgende Schleife, die die Werte aus den zwei sortierten Feldern A und B in das Feld C geordnet zusammenträgt,

```
while ia≤na oder ib≤nb do
   if ib>nb oder ia≤na und A(ia)≤B(ib)
   then C(ic):=A(ia)
        ia:=ia+1
   else C(ic):=B(ib)
        ib:=ib+1
   endif
   ic:=ic+1
endwhile
```

für die der Programmierer als Schleifeninvariante I

```
I1:        1≤ia≤na+1                    [Wertebereich von ia]
I2:   und  1≤ib≤nb+1                    [Wertebereich von ib]
I3:   und  (ic-1)=(ia-1)+(ib-1)         [Beziehung zwischen
                                              ia, ib und ic]

I4:   und  (ic≤1 oder ia>na oder C(ic-1)≤A(ia))
     [ggf. nächstes Element von A ≥ letztes Element in C]

I5:   und  (ic≤1 oder ib>nb oder C(ic-1)≤B(ib))
     [ggf. nächstes Element von B ≥ letztes Element in C]
```

I6: $\mathbf{und}_{i=1}^{ic-2}\ C(i) \leq C(i+1)$ [C sortiert]

I7: $\mathbf{und}_{i=1}^{na-1}\ A(i) \leq A(i+1)\ \mathbf{und}_{i=1}^{nb-1}\ B(i) \leq B(i+1)$ [A, B sortiert]

festgelegt und die einzelnen darin vorkommenden Terme wie oben angegeben benannt hat. Im umfangreichsten Teil des Beweises muß man die Invarianz der Schleifeninvariante beweisen, d.h., daß

```
{I und (ia≤na oder ib≤nb)}
if ib>nb oder ia≤na und A(ia)≤B(ib)
then C(ic):=A(ia)
     ia:=ia+1
else C(ic):=B(ib)
     ib:=ib+1
endif
ic:=ic+1
{I}
```

Der zu behandelnde Programmteil besteht aus einer Folge (einer if-Anweisung und einer Zuweisung). Man wendet deshalb die Beweisregel F1 an und zerlegt die zu beweisende Aussage oben in die folgenden zwei Teile:

```
{I und (ia≤na oder ib≤nb)}
if ib>nb oder ia≤na und A(ia)≤B(ib)
then C(ic):=A(ia)
     ia:=ia+1
else C(ic):=B(ib)
     ib:=ib+1
endif
```

$\{I^{ic}_{ic+1}\}$

und

$\{I^{ic}_{ic+1}\}$ ic:=ic+1 {I}

wobei wir I^{ic}_{ic+1} dadurch erhalten, daß wir in I die Variable ic durch den Ausdruck (ic+1) ersetzen.

Die letzte Aussage oben ist gemäß der Beweisregel Z1 wahr. Wir müssen noch die vorletzte oben aufgeführte Aus-

sage über die if-Anweisung durch Anwendung der Beweisregel IF1 zerlegen. Nach algebraischer Vereinfachung der Vorbedingungen erhalten wir die folgenden zu beweisenden Aussagen:

{I **und** ia≤na **und** (ib>nb **oder** A(ia)≤B(ib))}
C(ic):=A(ia)
ia:=ia+1
$\{I^{ic}_{ic+1}\}$

und

{I **und** ib≤nb **und** (ia>na **oder** B(ib)<A(ia))}
C(ic):=B(ib)
ib:=ib+1
$\{I^{ic}_{ic+1}\}$

Die Nachbedingung ist ein langer Ausdruck, der die gleiche Form und Struktur aufweist wie die Schleifeninvariante (siehe oben). Die algebraische Manipulation ist übersichtlicher, wenn wir die oben stehenden Aussagen durch Anwendung der Beweisregel DC3 zerlegen in die Aussagen

{I **und** ia≤na **und** (ib>nb **oder** A(ia)≤B(ib))}
C(ic):=A(ia)
ia:=ia+1
$\{I1^{ic}_{ic+1}\}$ = {1≤ia≤na+1}

und

{I **und** ia≤na **und** (ib>nb **oder** A(ia)≤B(ib))}
C(ic):=A(ia)
ia:=ia+1
$\{I2^{ic}_{ic+1}\}$ = {1≤ib≤nb+1}

und

{I **und** ia≤na **und** (ib>nb **oder** A(ia)≤B(ib))}
C(ic):=A(ia)
ia:=ia+1
$\{I3^{ic}_{ic+1}\}$ = {(ic)=(ia-1)+(ib-1)}

usw. Man hat zwar 14 Aussagen dieser Form zu beweisen, aber sie sind relativ einfach (manche sogar trivial). Jede wird durch die Anwendung der Beweisregeln F1, Z1 und Z2 (siehe Abschnitt 4.3, Die Folge von Anweisungen) bewiesen. Wegen der Symmetrie zwischen ia und ib usw. treten die gleichen Manipulationen paarweise auf, was die eigentlich zu leistende Arbeit halbiert.

Übung:

1. Die gegebene while-Schleife und ihre Schleifeninvariante sind hinsichtlich der Felder A und B weitgehend symmetrisch. Die zwei zu beweisenden Aussagen über die then- und else-Zweige der if-Anweisung sind bis auf die Relation zwischen A(ia) und B(ib) (an einer Stelle $<$ statt $\leq$) symmetrisch hinsichtlich A und ia gegenüber B und ib. Wie kann eine vollständige Symmetrie erreicht werden?
2. Formuliere im Detail alle 14 Aussagen, in die man durch Anwendung der Beweisregel DC3 die zwei oben erwähnten Aussagen zerlegt.
3. Beweise alle 14 Aussagen. ∎

4.6 Der Programmteil oder das Unterprogramm

In einer Korrektheitsaussage über einen Programmteil oder einen Aufruf auf ein Unterprogramm treten in der Nachbedingung typischerweise zwei Aussagen auf. Die eine Aussage bezieht sich auf die Wirkung des fraglichen Programmteils bzw. Unterprogramms. Die andere Aussage bezieht sich auf Variablen, die der fragliche Programmteil bzw. das Unterprogramm nicht verändert. Die letzte Aussage hat mit der Wirkung anderer, vorher ausgeführter Programmteile zu tun, deren Ergebnisse während der Ausführung des fraglichen Programmteils aufbewahrt werden müssen (d.h. nicht beeinflußt werden sollen oder dürfen), da sie später benötigt werden. (Vgl. die Beweisregeln U1, U2 und U3 in den Abschnitten 3.3.15 bis 3.3.17.)

Das folgende Beispiel, in dem drei Unterprogramme nacheinander aufgerufen werden, zeigt, wie man die zwei Arten von Aussagen trennt und im Beweis behandelt. Achte insbesondere auf den Teil des Beweises, der sich mit dem zweiten Aufruf befaßt.

Beispiel: Der folgende Programmabschnitt soll dafür sorgen, daß die Elemente der Felder A und B in das Feld C übertragen werden, und zwar so, daß das Feld C nachher sortiert ist. Anfangs sind die Felder A und B nicht unbedingt sortiert. Die Variablen na, nb und nc geben an, wieviele Elemente die Felder A, B bzw. C enthalten. Die hier zu beweisende Korrektheitsaussage ist wie folgt:

{na≥0 **und** nb≥0} [Wertebereiche von na, nb]
call sortA
call sortB
call zusammenfügen

$\{nc=na+nb \text{ und}_{i=1}^{nc-1} C(i)\leq C(i+1)\}$ [C sortiert]

Dieser Programmabschnitt ist eine Folge von drei Anweisungen. Deshalb zerlegen wir unsere Beweisaufgabe gemäß der Beweisregel F1 in drei Unteraufgaben:

{na≥0 **und** nb≥0} **call** sortA {P1} ? [1: P1 noch zu bestimmen]

{P1?} **call** sortB {P2} [2: P2 noch zu bestimmen]

{P2?} **call** zusammenfügen [3]

$\{nc=na+nb \text{ und}_{i=1}^{nc-1} C(i)\leq C(i+1)\}$

Für die letzten zwei Anweisungen in der Folge sind Vorbedingungen zu ermitteln. Für die erste Anweisung ist die gegebene Vorbedingung zu verifizieren.

Wie üblich im Falle einer Folge von Anweisungen fangen wir mit der Aussage über die letzte Anweisung in der Folge an.

Unteraufgabe 3: Die vorgegebene Spezifikation des Unterprogramms *zusammenfügen* besagt, daß es (1) den Wert der Variablen nc berechnet und (2) die Werte der Variablen A(1), ... A(na), B(1), ... B(nb) in das Feld C derart überträgt, daß das Feld C sortiert ist. Voraussetzung für das richtige Funktionieren dieses Unterprogramms ist, daß die Felder A und B bereits in sich sortiert sind. Formal,

{na≥0 **und** nb≥0 **und**$_{i=1}^{na-1}$ A(i)≤A(i+1)

und$_{i=1}^{nb-1}$ B(i)≤B(i+1)}

call zusammenfügen {nc=na+nb **und**$_{i=1}^{nc-1}$ C(i)≤C(i+1)}

Die Nachbedingung der Unteraufgabe 3 und die Nachbedingung der Spezifikation des Unterprogramms *zusammenfügen* stimmen überein. Wir können deshalb die gesuchte Vorbedingung P2 dieser Spezifikation direkt entnehmen.

Unteraufgabe 2: Durch die Bestimmung von P2 (siehe Unteraufgabe 3 oben) wird die Unteraufgabe 2 wie folgt:

{P1?} **call** sortB
{na≥0 **und** nb≥0 **und**$_{i=1}^{na-1}$ A(i)≤A(i+1)

und$_{i=1}^{nb-1}$ B(i)≤B(i+1)}

Die vorgegebene formale Spezifikation des Unterprogramms *sortB* besagt, daß es die Werte der Variablen B(1), ... B(nb) vertauscht (also ggf. verändert) und die Werte der Variablen i, j und k verändert. (Die Werte der Variablen i, j und k haben nur innerhalb des Unterprogramms Bedeutung.) Es verändert keine andere Variable. Dabei gilt

{nb≥0} **call** sortB {**und**$_{i=1}^{nb-1}$ B(i)≤B(i+1)}

Zu ermitteln ist eine Vorbedingung für eine Nachbedingung, die stärker ist als die der Spezifikation. Wir müssen deshalb die in der Unteraufgabe 2 vorkommende Nachbedingung derart aufteilen, daß der eine Teil identisch ist mit der in der Spezifikation angegebenen Nachbedingung (oder daraus folgt) und der andere Teil sich nur auf Variablen bezieht, die das Unterprogramm *sortB* nicht verändert. Mit anderen Worten, wir teilen die in der Unteraufgabe 2 vorkommende Nachbedingung den zwei Bedingungen P und B der Beweisregel U2 entsprechend auf. (Siehe die Beweisregeln U2 und U3.) Dadurch wird unsere Unteraufgabe 2

{P1?} **call** sortB
{na≥0 **und** nb≥0 **und**$_{i=1}^{na-1}$ A(i)≤A(i+1) [B: von sortB unverändert]

und$_{i=1}^{nb-1}$ B(i)≤B(i+1)} [P: von sortB verändert]

Wir wollen die Vorbedingung durch Anwendung der Beweisregel U2 ermitteln. Dabei sind die darin angesprochenen Bedingungen V, P und B wie folgt:

V: nb≥0

P: **und**$_{i=1}^{nb-1}$ B(i)≤B(i+1)

B: na≥0 **und** nb≥0 **und**$_{i=1}^{na-1}$ A(i)≤A(i+1)

Die in der Unteraufgabe 2 vorkommende Nachbedingung ist (P **und** B). Gemäß der Beweisregel U2 ist die Vorbedingung (V **und** B). Die gesuchte Vorbedingung P1 ist also:

{na≥0 **und** nb≥0 **und**$_{i=1}^{na-1}$ A(i)≤A(i+1)} [P1: V **und** B]
call sortB
{na≥0 **und** nb≥0 **und**$_{i=1}^{na-1}$ A(i)≤A(i+1) [B: von sortB unverändert]

und$_{i=1}^{nb-1}$ B(i)≤B(i+1)} [P: von sortB verändert]

Unteraufgabe 1: Durch die Bestimmung von P1 (siehe Unteraufgabe 2 oben) wird die Unteraufgabe 1 wie folgt:

{na≥0 **und** nb≥0} **call** sortA

{na≥0 **und** nb≥0 **und**$_{i=1}^{na-1}$ A(i)≤A(i+1)} ?

Die vorgegebene formale Spezifikation des Unterprogramms *sortA* entspricht der des Unterprogramms *sortB* (siehe oben). Das Unterprogramm *sortA* vertauscht (also ggf. verändert) die Werte der Variablen A(1), ... A(na) und verändert die Werte der internen Variablen i, j und k. Es verändert keine andere Variable. Dabei gilt

{na≥0} **call** sortA {**und**$_{i=1}^{na-1}$ A(i)≤A(i+1)}

Auch in diesem Falle ist die in der Unteraufgabe vorkommende Nachbedingung stärker als die der Spezifikation. Wir müssen deshalb die zu beweisende Nachbedingung wie in der Unteraufgabe 2 oben aufteilen. (Siehe die Beweisregeln U2 und U3.) Dadurch wird unsere Unteraufgabe 1

{na≥0 **und** nb≥0} **call** sortA
{na≥0 **und** nb≥0 [von sortA unverändert]

$\mathbf{und}_{i=1}^{na-1}$ A(i)≤A(i+1)} ? [von sortA verändert]

Gemäß der Beweisregel U2 gilt diese Aussage. Bei dieser Anwendung der Beweisregel U2 sind die darin angesprochenen Bedingungen V, P und B wie folgt:

V: na≥0

P: $\mathbf{und}_{i=1}^{na-1}$ A(i)≤A(i+1)

B: na≥0 **und** nb≥0

Damit ist die ursprünglich zu beweisende Korrektheitsaussage über den gesamten Programmabschnitt (die Folge der drei Aufrufe auf Unterprogramme) verifiziert. ■

Wenn sich die Nachbedingung eines Aufrufs auf ein Unterprogramm nicht wie oben (d.h. nicht wie für die Anwendung der Beweisregel U2 oder U3 erforderlich) aufteilen läßt, liegt ein Konstruktionsfehler vor. Entweder ist das Programm fehlerhaft oder die formale Spezifikation des Programmteils oder Unterprogramms ist nicht vollständig.

4.7 Zusammenfassung über die Analyse und Korrektheitsbeweisführung

Um die partielle Korrektheit eines Programms oder eines Programmteils zu beweisen, formuliert man die Beweisaufgabe zuerst als eine Korrektheitsaussage mit der Form

{V} S {P}

wobei V die bekannte (gegebene) Vorbedingung, P die gegebene Nachbedingung bzw. S der fragliche Programmteil ist.

Die für S geeignete Beweisregel (siehe die Tabelle in Abschnitt 3.4, Die Anwendung der Beweisregeln) wird anschließend angewendet, wodurch die ursprüngliche Korrektheitsaussage (Beweisaufgabe) in andere, untergeordnete Aussagen (Beweisaufgaben) zerlegt wird. Dieser Prozeß wird fortgesetzt, bis nur Aussagen übrigbleiben, die sich auf Zuweisungen beziehen. Schließlich werden diese letzten Aussagen durch Anwendung der Beweisregeln Z1 und Z2 verifiziert.

Dabei zerlegt man Korrektheitsaussagen über größere Programmteile in Korrektheitsaussagen über immer kleinere Bestandteile des Programms, bis Aussagen über Zuweisungen ermittelt werden. Die auf diese Weise erfolgte Zerlegung des Beweises spiegelt die Struktur des fraglichen Programmteils wider.

Wenn in diesem Zerlegungsprozeß bereits bewiesene Aussagen erreicht werden, kann man dort natürlich aufhören. Insbesondere bei Aussagen über Unterprogramme wird diese Situation auftreten. (Siehe Abschnitt 4.6.) Solche bereits bewiesenen Aussagen stellen Korrektheitslemmata und -sätze über untergeordnete Programmteile dar, worauf sich jeder Korrektheitsbeweis für einen übergeordneten Programmteil beziehen kann und soll.

Um ferner die vollständige Korrektheit zu beweisen, muß vor allem gezeigt werden, daß der Schleifenkern jeder Schleife nur endlich viele Male ausgeführt wird. Weiterhin ist zu zeigen, daß jede Anweisung jedesmal mit einem definierten Ergebnis ausgeführt wird, d.h., daß kein Laufzeitfehler während der Ausführung des fraglichen Programms auftreten kann. Allgemein gültige Richtlinien für die entsprechende Beurteilung der Anweisungen eines Programms enthält Kapitel 2, Die Ausführung von Programmanweisungen: Wirkungen und Axiome. (Siehe Abschnitt 4.4.1 für ein Anwendungsbeispiel.)

5. Entwurf: die Konstruktion beweisbar korrekter Programme

In diesem Kapitel werden wir einige Programmteile entwerfen. Jeder wird derart konstruiert, daß er eine vorgegebene Spezifikation – bestehend aus einer Vor- und einer Nachbedingung – erfüllt. Dabei dienen die verschiedenen Anforderungen eines Korrektheitsbeweises als Leitlinien für die Konstruktion des Programms. Sie ermöglichen es sogar, einige Teile des zu konstruierenden Programms mehr oder weniger direkt abzuleiten.

Die sich daraus ergebende und hier vorgestellte Vorgehensweise lenkt die Aufmerksamkeit des Konstrukteurs auf die wesentlichen Aspekte des Programms und von unwesentlichen Aspekten ab. Folglich konstruiert er zielorientierter und systematischer als bisher. Das Ergebnis ist oft ein überraschend kompaktes Programm mit einfacher, übersichtlicher und logischer Struktur.

Diese Vorgehensweise unterscheidet sich deutlich von der herkömmlichen Programmierung. Man betrachtet das zu konstruierende Programm aus einem ganz anderen Blickwinkel. Im Vergleich zur herkömmlichen Programmierungsweise achtet man mehr auf *Zustände* und auf das, was sich nicht ändert (auf *Invarianten* und *Bedingungen*), und viel weniger auf die *Veränderungen*, die die Anweisungen bewirken. Man muß diese andere Vorgehensweise – und vor allem andere Denkweise – lernen und sich darin einarbeiten, aber die Erfahrung zeigt, daß das nicht besonders schwierig ist.

Mit Übung und Erfahrung läßt sich die hier geschilderte Vorgehensweise leicht und schnell durchführen. Der Konstrukteur wendet die verschiedenen Beweisregeln fast unbewußt und reflexartig an, genau wie seine Ingenieur-Kollegen der anderen, klassischen Fachrichtungen ihre jeweiligen theoretischen Grundlagen auf praktische Konstruktionsaufgaben anwenden.

Aus Platzgründen werden hier nur vier Beispiele beschränkter Größe behandelt. Dabei geht es jeweils um die Konstruktion eines Unterprogramms auf der niedrigsten hierarchischen Ebene. Für Konstruktionsbeispiele von hierarchisch höher geordneten Programmteilen, darunter ein Steu-

erprogramm auf der obersten Ebene für ein mittelgroßes Programmsystem, siehe [Baber, 1987, Kapitel 6].

Bei der Konstruktion eines nachweislich korrekten Programms geht man im allgemeinen wie folgt vor: Aus der Beschreibung der Aufgabe, die das Programm lösen soll, leitet man die Vor- und Nachbedingungen in algebraischer Form ab, falls sie in der Aufgabenstellung nicht bereits vorgegeben wurden. Danach entscheidet man sich für die Grundstruktur des zu konstruierenden Programms. Wenn – wie in den meisten Fällen – eine Schleife gewählt wird, legt man die Schleifeninvariante fest. Aus dem Unterschied zwischen der Nachbedingung und der Schleifeninvariante leitet man die while-Bedingung ab. Unter Verwendung der Schleifeninvariante als Checkliste konstruiert man den Schleifenkern. Anschließend erstellt man für den Programmentwurf einen Korrektheitsbeweis. Typischerweise können mehrere Teile davon mehr oder weniger unmittelbar aus dem Konstruktionsvorgang entnommen werden.

Die Schleifeninvariante wird durch Verallgemeinerung der Vor- und Nachbedingungen ermittelt. In diesem Konstruktionsschritt kann man entweder die algebraischen Formeln oder Diagramme (oder beides) verwenden; manchmal führen Formeln direkter zum Ziel und manchmal ist der Vorgang einfacher und übersichtlicher, wenn man die Bedingungen mit Hilfe von Diagrammen darstellt. Auf jeden Fall sollte man die Fähigkeit entwickeln, mit beiden Darstellungsformen flott umgehen und zwischen ihnen "übersetzen" zu können.

Oft wird bei der Festlegung der Schleifeninvariante die erforderliche Initialisierung offensichtlich.

Achte beim Lesen der folgenden Konstruktionsbeispiele sorgfältig darauf, wie die oben allgemein beschriebenen Konstruktionsschritte konkret ausgeführt werden.

5.1 Konstruktionsbeispiel: Die lineare Suche

In unserem ersten Konstruktionsbeispiel werden wir den Programmteil entwerfen, dessen Korrektheit im Abschnitt 4.4.1 bewiesen wurde.

Gegeben sind eine Variable n, ein Feld A(1), A(2), ... A(n) und die Variable x. Das zu konstruierende Programm soll feststellen, ob der Wert von x im Feld A(1), A(2), ... A(n) vorkommt und wenn ja, wo er zum ersten Mal in A

vorkommt. Die Stelle in A, bei der Gleichheit gefunden wird, soll durch den Wert der Variablen k angegeben werden.

5.1.1 Die Spezifikation

Gegeben ist die Vorbedingung

$n \in Z$ **und** $0 \leq n$

Das zu konstruierende Unterprogramm soll einen Wert für die Variable k derart berechnen, daß nach der Ausführung des Programmteils die Nachbedingung

$k \in Z$ **und** $1 \leq k \leq n+1$ [Wertebereich von k]

und $\bigwedge_{i=1}^{k-1} A(i) \neq x$ [alle Elemente vor dem k-ten $\neq$ x]

und {$k \leq n$ **und** $A(k)=x$ [A(k) = x]

oder $k=n+1$} [kein Element von A = x]

erfüllt ist. Es darf keine andere Variable verändert werden.

5.1.2 Die Grundstruktur des Programmteils

Wegen der Natur der Aufgabe liegt es nahe, eine Schleife als die Grundstruktur für unser Unterprogramm zu wählen. Eins nach dem anderen werden Elemente des Felds A auf Gleichheit mit x geprüft; die *Wiederholung* desselben Vorgangs deutet auf eine Schleife hin. Zu einer Schleife gehört in der Regel eine Initialisierung, die die anfängliche Wahrheit der Schleifeninvariante sicherstellt. Unser Programmteil sieht jetzt wie folgt aus:

Initialisierung
while B **do** S **endwhile**

5.1.3 Die Schleifeninvariante

Die wichtigste Konstruktionsentscheidung hinsichtlich einer Schleife ist die Festlegung der Schleifeninvariante. Eine geeignete Schleifeninvariante wird durch Verallgemeinerung der Nachbedingung und der Anfangssituation erarbeitet (siehe Abschnitt 3.3.9).

Im Abschnitt 4.4.2 wurde für den hier zu konstruierenden Programmteil eine Schleifeninvariante bestimmt. Dort lag der fertige Programmteil – insbesondere die Initialisierung

der Schleife – vor. Hier gehen wir ähnlich vor, jedoch können wir uns nicht auf eine bereits vorliegende Initialisierung beziehen.

Wir fangen mit der Nachbedingung (siehe oben) an und fragen uns, wie sie geschwächt werden müßte, um auch anfangs erfüllt zu sein. Ist bzw. kann die erste Zeile der Nachbedingung anfangs wahr sein? Die Vorbedingung stellt sicher, daß $n \geq 0$. Wenn die erste Zeile der Nachbedingung für alle möglichen Werte von n wahr sein soll, muß gelten, daß $1 \leq k \leq 0+1$, also k=1. Andersherum betrachtet wird die erste Zeile der Nachbedingung wahr sein, falls k=1. Diese Überlegung deutet auf die Zuweisung k:=1 für die Initialisierung.

Falls k=1, dann ist die zweite Zeile der Nachbedingung die leere **und**-Reihe, die definitionsgemäß wahr ist.

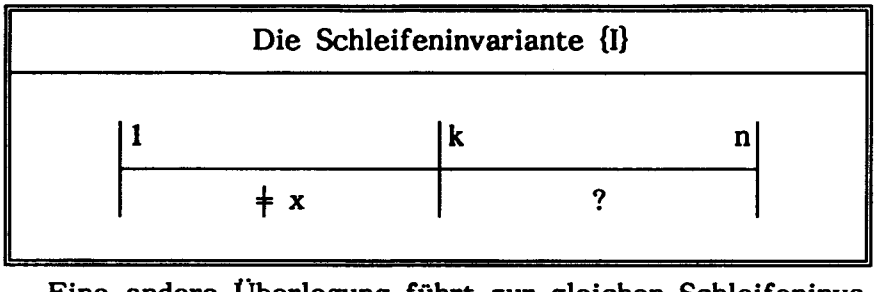
Die dritte Zeile der Nachbedingung besagt u.a., daß A(k)=x. Anfangs (k=1) wird nicht bekannt sein, ob A(k)=x oder nicht. Auch die vierte Zeile kann anfangs entweder wahr oder falsch sein. Weil diese zwei Zeilen zusammen einen **und**-verknüpften Term in der Nachbedingung bilden, kann die Nachbedingung dadurch geschwächt werden, daß diese Zeilen weggelassen werden.

Aus diesen Überlegungen bietet sich die folgende Bedingung als Schleifeninvariante I an:

$k \in Z$ **und** $1 \leq k \leq n+1$ [Wertebereich von k]

und$_{i=1}^{k-1}$ $A(i) \neq x$ [alle Elemente vor dem k-ten $\neq$ x]

Die Schleifeninvariante I kann durch das folgende Diagramm dargestellt werden:



Eine andere Überlegung führt zur gleichen Schleifeninvariante. Dabei stellen wir uns die Frage, welche Teile der Nachbedingung bereits während der Suche wahr sein müssen und welche nicht. Die erste Zeile der Nachbedingung be-

grenzt den Wertebereich der zu berechnenden Variablen k; der angegebene Wertebereich läßt alle während der Suche offensichtlich benötigten Werte zu. Die bereits untersuchten Elemente von A werden ungleich x sein; also die zweite Zeile der Nachbedingung wird wahr sein. Während der Suche wird im allgemeinen weder A(k)=x noch k=n+1 gelten; die letzten zwei Zeilen der Nachbedingung gehören also nicht zu einer geeigneten Schleifeninvariante.

5.1.4 Die while-Bedingung

Im Korrektheitsbeweis muß man zeigen, daß die Nachbedingung nach der Ausführung der Schleife erfüllt ist, insbesondere, daß [I **und nicht** B] ⟹ die Nachbedingung P (siehe Beweisregel W2). Wir haben die Schleifeninvariante durch Weglassen eines **und**-verknüpften Terms der Nachbedingung gebildet. Der weggelassene Term bietet sich als [**nicht** B] an. Wir bilden die while-Bedingung B also durch Negieren des weggelassenen Terms:

nicht [k≤n **und** A(k)=x **oder** k=n+1]

= [k>n **oder** A(k)≠x] **und** k≠n+1

= [k>n **oder** k≤n **und** A(k)≠x] **und** k≠n+1

= [k>n **und** k≠n+1 **oder** k≤n **und** k≠n+1 **und** A(k)≠x]

= [k>n+1 **oder** k≤n **und** A(k)≠x]

Die Schleifeninvariante wird immer wahr sein, wenn die while-Bedingung ausgewertet wird. Daraus folgt, daß der linke Term oben immer falsch sein wird. Wir wählen deshalb für die while-Bedingung nur

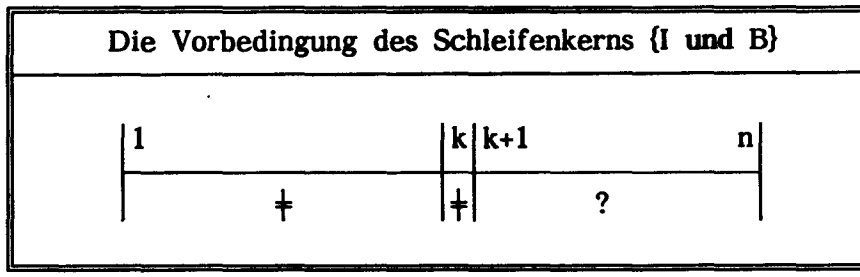
k≤n **und** A(k)≠x

5.1.5 Der Schleifenkern

Der Schleifenkern hat nur die Aufgaben, (1) die Wahrheit der Schleifeninvariante aufzubewahren und (2) Fortschritt in Richtung Terminierung (Nachbedingung der Schleife) zu erzielen. Vgl. (1) die Beweisregel W1 bzw. den Schritt 3 der Beweisregel W2 – {I **und** B} S {I} – sowie (2) den Schritt 5 der Beweisregel W2. Jede andere Überlegung in bezug auf

die Konstruktion des Schleifenkerns ist überflüssig, da sie nichts zum Korrektheitsbeweis beiträgt.

Das folgende Diagramm stellt dar, was jedesmal vor dem Schleifenkern über die Werte der A(.) bekannt ist {I und B}:



Vergleicht man dieses Diagramm für {I und B} mit dem Diagramm für {I} (siehe das vorletzte Diagramm oben), wird ersichtlich, daß sich das Diagramm für {I und B} in das der Schleifeninvariante umwandelt, wenn k um 1 erhöht wird. Eine entsprechende Betrachtung der logischen Ausdrücke führt zum gleichen Schluß.

Durch die Erhöhung von k wird der unbekannte Bereich kleiner; es wird also dadurch Fortschritt in Richtung Terminierung erzielt. Aus dieser Beobachtung bietet sich die Länge des unbekannten Bereichs in der Schleifeninvariante, d.h. n-k+1, als Schleifenvariante für den formalen Beweis der Terminierung an. (Siehe das Diagramm für {I} oben.)

Der Schleifenkern soll also aus der einen Zuweisung k:=k+1 bestehen.

5.1.6 Der gesamte Programmteil

Damit ist der gesamte Programmteil wie folgt:

```
k:=1
while k≤n und A(k)≠x do k:=k+1 endwhile
```

5.1.7 Der Korrektheitsbeweis

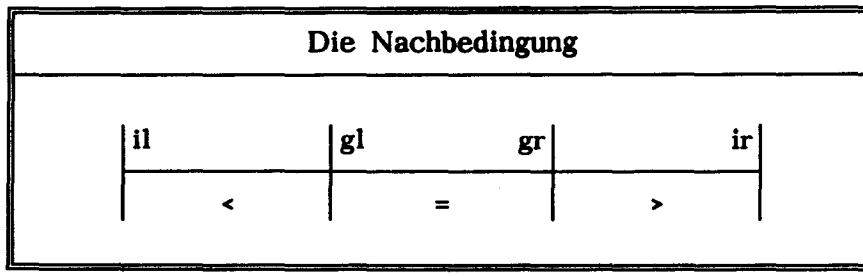
Die Korrektheit des hier konstruierten Programms wurde im Abschnitt 4.4.1 bewiesen.

5.2 Konstruktionsbeispiel: Unterteilen eines Felds

Gegeben seien die ganzzahligen Variablen il und ir sowie die Feldvariablen X(il), X(il+1), ... X(ir). Die Vorbedingung

ist nicht näher angegeben. Eventuelle Einschränkungen, die zur Vorbedingung gehören, sind zu bestimmen.

Zu konstruieren ist ein Unterprogramm, das die Werte der genannten Feldvariablen vertauscht (permutiert) sowie die Werte der Variablen gl und gr derart bestimmt, daß drei Bereiche gebildet werden:



Der mittlere Bereich darf nicht leer sein, die anderen schon. Der Wert der Elemente des mittleren Bereichs soll vom zu konstruierenden Programmteil bestimmt werden. Er darf auf beliebige Weise gewählt werden.

5.2.1 Die Spezifikation

In algebraischer Form ist die Nachbedingung

$$il \leq gl \leq gr \leq ir$$

$$\text{und}_{i=il}^{gl-1}\ X(i)<X(gl)$$

$$\text{und}_{i=gl}^{gr}\ X(i)=X(gl)$$

$$\text{und}_{i=gr+1}^{ir}\ X(i)>X(gl)$$

Die Vorbedingung ist noch festzulegen.

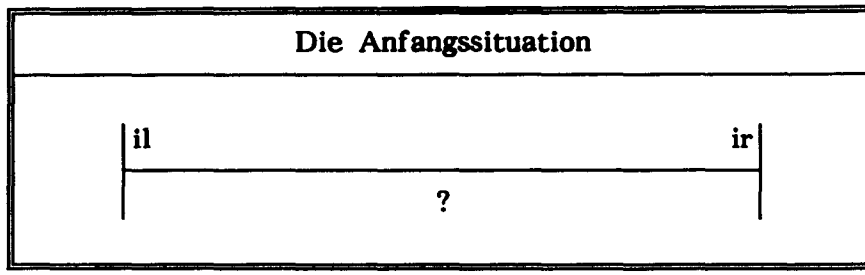
5.2.2 Die Grundstruktur des Programmteils

Es muß im allgemeinen damit gerechnet werden, daß mehrere Vertauschungen von Werten von Elementen des Felds X notwendig sein werden, um die Nachbedingung zu erfüllen. Die Wiederholung des Vertauschvorgangs deutet darauf hin, daß eine Schleife die geeignete Grundstruktur des Programms wäre.

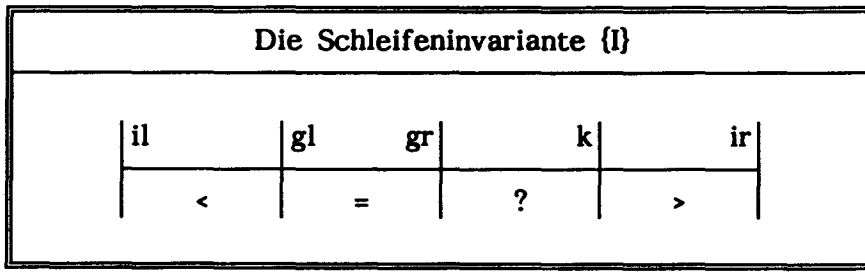
5.2.3 Die Schleifeninvariante

Die Schleifeninvariante I muß sowohl am Anfang als auch am Ende wahr sein. Das Diagramm oben schildert die Nachbedingung.

Am Anfang ist nichts über die Beziehungen zwischen den Werten der Feldelemente bekannt:



Die Schleifeninvariante muß eine Verallgemeinerung der Anfangssituation und der Endsituation (etwa Nachbedingung) sein. Sie muß also die vier Bereiche <, =, > und ? vorsehen. Eine der Möglichkeiten ist:



In algebraischer Form ist die Schleifeninvariante I

$$il \le gl \le gr \le k \le ir$$

$$\text{und}_{i=il}^{gl-1}\ X(i)<X(gl)$$

$$\text{und}_{i=gl}^{gr}\ X(i)=X(gl)$$

$$\text{und}_{i=k+1}^{ir}\ X(i)>X(gl)$$

5.2.4 Die while-Bedingung

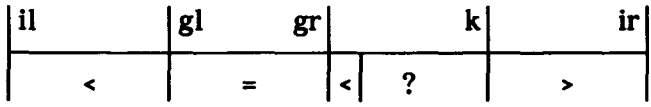
Falls $gr \ge k$, ist der ?Bereich leer und aus der Schleifeninvariante folgt die Nachbedingung. Siehe die Diagramme oben für die Nachbedingung und die Schleifeninvariante I.

D.h., die Endbedingung für die while-Schleife ist $gr \ge k$. Die while-Bedingung ist die Negierung davon, also $gr < k$.

5.2.5 Der Schleifenkern

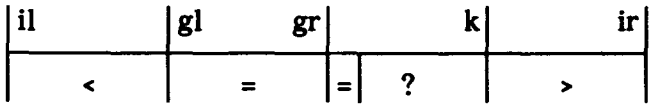
Wir müssen den Schleifenkern derart konstruieren, daß er (1) die Wahrheit der Schleifeninvariante aufrechterhält und (2) Fortschritt in Richtung Erfüllung der Nachbedingung erzielt.

In der Schleifeninvariante (siehe Diagramm oben) muß der ?Bereich verkleinert werden, um Fortschritt in Richtung der Nachbedingung zu erzielen. Aus dem ?Bereich muß ein Element – z. B. X(gr+1) – ausgewählt und in den entsprechenden Bereich eingeordnet werden. Durch Vergleichen des ausgewählten Elements mit einem beliebigen Element aus dem =Bereich wird festgestellt, zu welchem Bereich das ausgewählte Element gehört. Aus dem Vergleich zwischen X(gr+1) und X(gl) wird sich entweder Fall 1:



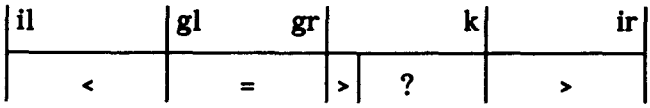
[Fall 1]

oder Fall 2:



[Fall 2]

oder Fall 3:

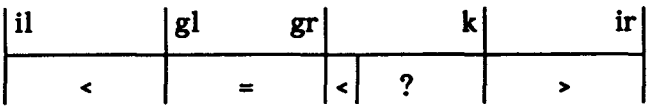


[Fall 3]

ergeben.

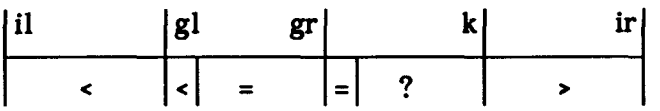
Die Schleifeninvariante trifft nicht mehr zu; ihre Wahrheit muß wieder hergestellt werden. Das ist der *einzige* Zweck der übrigen Anweisungen. Die Anweisungen, die ausgeführt werden müssen, um die Wahrheit der Schleifeninvariante wieder sicherzustellen, sind in den drei Fällen unterschiedlich. Diese Fallunterscheidung bei der Konstruktion führt zu einer entsprechenden if-Anweisung im fertigen Programm.

Fall 1:

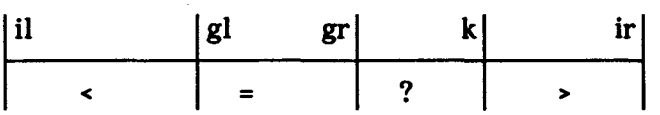


[Fall 1]

Das Vertauschen von X(gl) und X(gr+1) bringt diese Werte an die richtigen Stellen:



Die Erhöhung von gl und gr um 1 stellt die Schleifeninvariante wieder her:



[I nach Fall 1]

In diesem Fall müssen also die Anweisungen

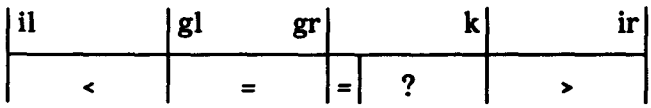
X(gl):=:X(gr+1)
gl:=gl+1
gr:=gr+1

ausgeführt werden.

Die Vertauschanweisung (x:=:y) bewirkt, daß die Werte der Variablen x und y vertauscht werden. Der vorherige Wert von x wird der Variablen y zugewiesen und der vorherige Wert von y wird der Variablen x zugewiesen. D.h., der vorherige Wert von x wird der nachherige Wert von y und der vorherige Wert von y wird der nachherige Wert von x. Der gleiche Effekt kann mit den folgenden Zuweisungen erreicht werden, wobei *zwvar* eine Zwischenvariable ist, die für keinen anderen Zweck verwendet wird.

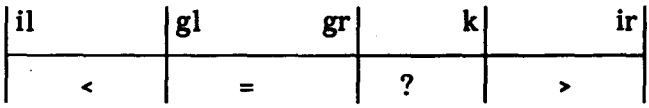
zwvar:=x
x:=y
y:=zwvar

Fall 2:



[Fall 2]

Alle Werte sind bereits an den richtigen Stellen; lediglich die Grenze gr muß durch Erhöhung um 1 aktualisiert werden.



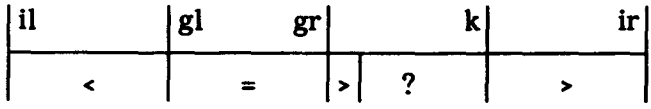
[I nach Fall 2]

In diesem Fall muß die Anweisung

gr:=gr+1

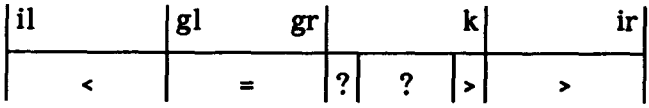
ausgeführt werden.

Fall 3:

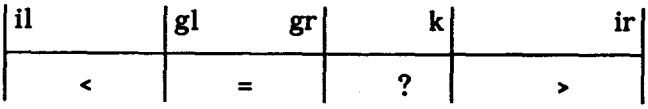


[Fall 3]

Das Vertauschen von X(k) und X(gr+1) bringt diese Werte an die richtigen Stellen:



Die Verringerung von k um 1 stellt die Schleifeninvariante wieder her:



[I nach Fall 3]

In diesem Fall müssen also die Anweisungen

X(k):=:X(gr+1)
k:=k-1

ausgeführt werden.

5.2.6 Die Initialisierung

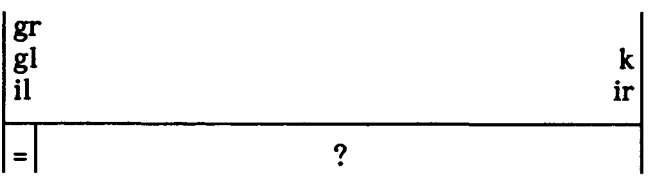
Wir müssen die Initialisierung derart konstruieren, daß danach die Schleifeninvariante wahr ist. Die Schleifeninvariante I erfordert, daß der =Bereich nicht leer ist (gl≤gr). In der Beschreibung der Konstruktionsaufgabe wurde gesagt, daß das =Element auf beliebige Weise gewählt werden darf. Die < und >Bereiche sind am Anfang leer. Der ?Bereich enthält alle Elemente außer dem als = gewählten. Dadurch wird die Initialisierung wie folgt:

```
gl:=il      [<Bereich leer]
gr:=gl      [=Bereich enthält 1 Element]
k:=ir       [>Bereich leer]
```

Die willkürliche Wahl des =Elements kann explizit ausgedrückt werden durch Voranstellung der Anweisung

X(il):=:X(j), wo j beliebig in il≤j≤ir

Die Initialisierung stellt die folgende Anfangssituation her:



[I nach Init.]

5.2.7 Das gesamte Programm

Wir haben jetzt alle einzelnen Teile des Programms konstruiert. Das gesamte Programm ist wie folgt:

```
gl:=il; gr:=gl; k:=ir
while gr<k do
    if X(gr+1)<X(gl)
    then X(gl):=:X(gr+1)
         gl:=gl+1
         gr:=gr+1
    else if X(gr+1)=X(gl)
    then gr:=gr+1
    else [Bemerkung: X(gr+1)>X(gl)]
         X(k):=:X(gr+1)
         k:=k-1
    endif
    endif
endwhile
```

5.2.8 Die Vorbedingung

Die Vorbedingung der Schleifeninvariante (als Nachbedingung) bezüglich der Initialisierung – die auch die Vorbedingung des ganzen Unterprogramms ist – wird durch Anwendung der Beweisregeln Z1 und F1 ermittelt:

$\{il \leq ir\}$
$gl := il$
$gr := gl$
$k := ir$
$\{il \leq gl \leq gr \leq k \leq ir$

$$\text{und}_{i=il}^{gl-1}\ X(i)<X(gl)$$

$$\text{und}_{i=gl}^{gr}\ X(i)=X(gl)$$

$$\text{und}_{i=k+1}^{ir}\ X(i)>X(gl)\}$$

Diese Vorbedingung drückt aus, daß das gegebene Feld X mindestens ein Element enthalten muß – damit am Ende der Ausführung des Programms der mittlere Bereich mindestens ein Element enthalten kann.

5.2.9 Terminierung der Schleife

Endet die Schleife? Der Wert des Ausdrucks

$k-gr$

der die Länge des ?Bereichs angibt, wird bei jeder Ausführung des Schleifenkerns um 1 verringert. Die untere Schranke dieses Werts ist 0 (siehe die Schleifeninvariante I und die while-Bedingung). Die Schleife wird also nach endlich vielen Ausführungen des Schleifenkerns enden.

5.2.10 Der Korrektheitsbeweis

Der Korrektheitsbeweis für diesen Programmteil spiegelt weitgehend die Konstruktionsschritte oben wider.

Übung:

1. Wie können die Beweisregeln Z1 und Z2 für die Vertauschanweisung (:=:) verallgemeinert werden? Wie kann man das Vertauschen in einem Beweis behandeln?
2. Beweise die Korrektheit des im Abschnitt 5.2 oben konstruierten Programms. ■

Siehe [Baber, 1987, Abschnitt 6.3] und [Dijkstra, 1976, Kapitel 14] für Varianten dieser Konstruktionsaufgabe.

5.3 Konstruktionsbeispiel: Suchen einer Teilkette

Gegeben sind die Variablen M und N und die Felder D und K. Die Werte der Variablen M und N sind Ganzzahlen. Die Werte der Feldvariablen D(.) und K(.) sind miteinander auf Gleichheit vergleichbar, aber nicht näher spezifiziert. (Sie können z.B. Zeichen sein.) Das zu konstruierende Programm soll im Feld D(id), id = 0, 1, ... M-1, nach der Folge K(ik), ik = 0, 1, ... N-1 suchen. In der Regel ist N viel kleiner als M.

Nach der Ausführung des Programms soll die Variable j angeben, wo die erste Teilfolge von D anfängt, die der Folge K gleich ist. Falls keine solche Teilfolge in D vorkommt, soll der Wert der Variablen j diese Situation kennzeichnen. Eine detailliertere Spezifikation des zu konstruierenden Programms – etwa eine algebraische Formulierung der Vor- und Nachbedingungen – liegt nicht vor.

5.3.1 Vorbereitende Analyse

Welche Werte kann das Ergebnis j grundsätzlich annehmen? Mit anderen Worten, ab welchen Stellen können Teilfolgen von D anfangen, die K gleich sein können? Die Folge D fängt an der Stelle 0 (D(0)) an; also die erste Teilfolge

von D, die K gleich sein könnte, fängt an der Stelle 0 an. Die letzte Teilfolge von D, die K gleich sein könnte, endet an der letzten Stelle von D, M-1. Diese Teilfolge muß genau so lang sein wie K; sie muß also an der Stelle M-1-(N-1) = M-N anfangen. Jede Teilfolge von D, die an einer nachherigen Stelle anfängt, ist zu kurz, um gleich K zu sein.

[D]

[K = erste Teilfolge von D]

[K = letzte Teilfolge von D]

Ist D kürzer als K (M<N), kann keine Teilfolge von D gleich K sein.

Wenn eine Teilfolge von D gefunden wird, die gleich K ist, dann muß der Wert von j im Bereich $0 \leq j \leq M-N$ liegen. Das zu konstruierende Programm soll die erste Teilfolge von D finden, die gleich K ist. Es muß deshalb mit der Suche an der Stelle 0 von D angefangen werden. Aus diesem Grund erscheint es sinnvoll, einen "zu großen" Wert von j ($j>M-N$) als Kennzeichen dafür zu vereinbaren, daß K nirgendwo in D vorkommt.

Implizit in dieser Überlegung steckt die Idee, die Aufgabe durch wiederholtes Vergleichen zu lösen, was auf eine Schleife als Grundstruktur für unser Programm hindeutet.

Über die Gleichheit bzw. Ungleichheit zwischen Teilfolgen von D und K werden wir vermutlich oft reden müssen, z.B. in der Nachbedingung, Schleifeninvariante usw. Deshalb sollten wir eine algebraische Formel dafür entwickeln. Die Teilfolge von D, die an der Stelle s anfängt, ist genau dann K gleich, wenn D(s)=K(0) und D(s+1)=K(1) und ... D(s+N-1)=K(N-1), d.h. wenn gilt

$$\text{und}_{a=0}^{N-1}\ D(s+a)=K(a)$$

[K = Teilfolge von D ab der Stelle s]

Diese Bedingung bezeichnen wir unten mit G(s).

Die Teilfolge von D, die an der Stelle s anfängt, ist K *ungleich*, wenn

nicht G(s) [K ≠ Teilfolge von D ab s]

= **nicht** $\mathbf{und}_{a=0}^{N-1}$ D(s+a)=K(a)

= $\mathbf{oder}_{a=0}^{N-1}$ D(s+a)≠K(a)

5.3.2 Die Spezifikation

In der Nachbedingung müssen die folgenden Aussagen zum Ausdruck kommen. Entweder ist K in D gefunden worden (0≤j≤M-N) oder K kommt in D nicht vor (j>M-N), siehe oben. (Es soll ausgeschlossen sein, daß K zwar in D vorkommt, aber nicht gefunden wird.) Falls K in D gefunden wurde, ist die Teilfolge von D, die an der Stelle j anfängt, gleich K. Ferner ist diese Teilfolge die erste, die gleich K ist; mit anderen Worten, alle vorherigen Teilfolgen von D sind ungleich K. Der Wert von j ist auf jeden Fall eine Ganzzahl und mindestens 0. Danach lautet unsere vorläufige Nachbedingung

j∈Z **und** 0≤j [Wertebereich von j]

$\mathbf{und}_{k=0}^{j-1}$ **nicht** G(k) [alle Teilfolgen von D vor der j-ten ≠ K]

und (j>M-N [keine Teilfolge von D = K]

oder j≤M-N **und** G(j)) [Teilfolge von D ab j = K]

In dieser Nachbedingung ist keine Obergrenze für den Wert von j angegeben. Oft ist es für den Terminierungsbeweis nützlich zu wissen, daß der Wert der Variablen, die in der Schleife berechnet wird, sowohl nach unten als auch nach oben begrenzt ist. Deshalb sollte die Nachbedingung im allgemeinen sowohl eine Unter- als auch eine Obergrenze für den Wert jeder vom Programm berechneten Variablen enthalten.

Zunächst würde es als ausreichend erscheinen, M-N+1, den ersten Wert größer als M-N, als maximalen Wert für j zuzulassen. Dann wäre der Wertebereich von j 0≤j≤M-N+1. Aber wenn D kurz und K lang wären, wäre M-N+1 negativ und diese Bedingung könnte nicht erfüllt werden. Es muß

also immer zugelassen werden, daß j=0. Der höchste Wert für j wird deshalb der größere von M-N+1 oder 0 sein müssen. Demnach wird unsere Nachbedingung in vollständig ausgeschriebener algebraischer Form

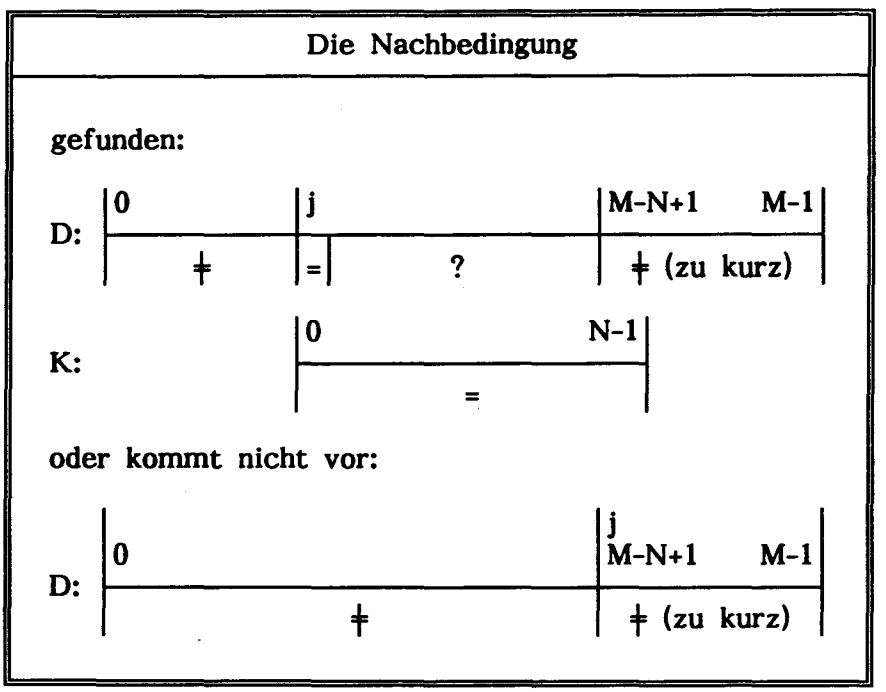
$j \in Z$ **und** $0 \le j \le \max(0, M-N+1)$ [Wertebereich von j]

und $\bigwedge_{k=0}^{j-1}$ (**oder** $\bigvee_{a=0}^{N-1} D(k+a) \neq K(a)$) [Teilfolgen von D vor der j-ten ≠ K]

und ($j > M-N$ [keine Teilfolge von D = K]

oder $j \le M-N$ **und** $\bigwedge_{a=0}^{N-1} D(j+a)=K(a)$) [Teilfolge von D ab j = K]

und in diagrammatischer Form

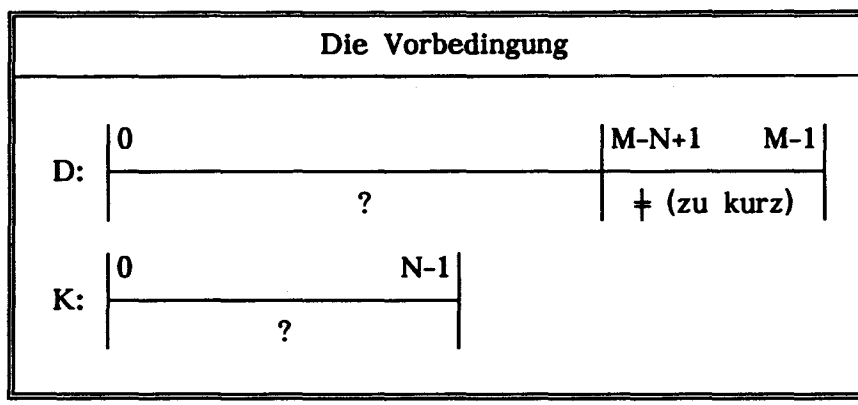


In der Aufgabenstellung ist die Vorbedingung nicht präzise definiert. Die Variablen M und N geben die Länge des Felds D bzw. K an; nur Werte ≥ 0 sind dafür sinnvoll. Es wurde in der Aufgabenbeschreibung bemerkt, daß in der Regel N viel kleiner als M ist. Das bedeutet jedoch nicht, daß N<M unbedingt gelten muß. Vielmehr ist es wünschenswert,

daß unser Programm auch dann richtig funktioniert, wenn eine lange Folge in einer kürzeren gesucht wird (in welchem Falle natürlich das Ergebnis "kommt nicht vor" sein soll). Die vorläufige Vorbedingung, die nach der Konstruktion eventuell zu revidieren ist, lautet also

MεZ und 0≤M und NεZ und 0≤N

Die dem Diagramm der Nachbedingung entsprechende Abbildung der Vorbedingung ist wie folgt:



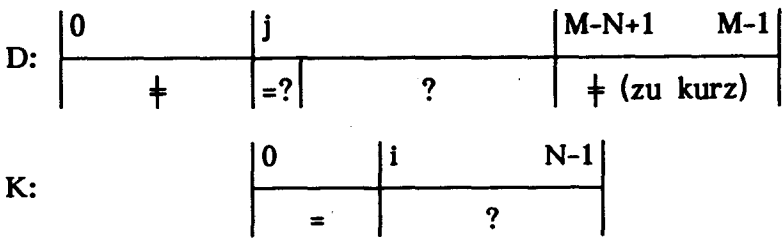
5.3.3 Die Schleifeninvariante

Eine geeignete Schleifeninvariante legen wir dadurch fest, daß wir die Vor- und Nachbedingungen verallgemeinern. Das können wir entweder anhand der Diagramme oder anhand der algebraischen Formeln tun.

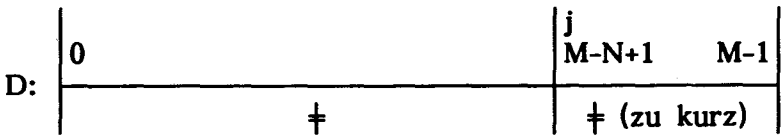
Schaut man die Diagramme für die Vor- und Nachbedingungen an, dann fällt vor allem auf, daß ein ?Bereich im Feld K in der Nachbedingung fehlt. Wenn wir die Nachbedingung entsprechend ergänzen, erhalten wir für die Schleifeninvariante

Die Schleifeninvariante {I}

wird noch gesucht:



oder kommt nicht vor:



Dieses Diagramm für die Schleifeninvariante kann wie folgt interpretiert werden: Entweder wird die Folge K in D noch gesucht oder es ist bereits festgestellt worden, daß K in D nicht vorkommt. Im ersten Fall (K wird noch gesucht) ist bereits festgestellt worden, daß die Teilfolgen von D, die an den Stellen 0, 1, ... j-1 anfangen, ungleich K sind. Die Teilfolge von D, die an der Stelle j anfängt, wird zur Zeit auf Gleichheit mit K geprüft. Bis zur Stelle i (ausschließlich) ist Gleichheit festgestellt worden.

Alternativ können wir die algebraischen Formeln für die Vor- und Nachbedingungen betrachten und uns fragen, was in der Nachbedingung im Wege steht, daß sie anfangs wahr ist. Nur der Wert 0 für j wird auf jeden Fall die erste Zeile erfüllen, woraus wir den Schluß ziehen können, daß die Initialisierung für j=0 sorgen muß. Dann verhindert nur "N" in der oberen Grenze der **und**-Reihe, daß die Nachbedingung anfangs wahr ist. Wir müssen dafür eine neue Variable (z.B. i) einführen, die wir derart initialisieren, daß diese **und**-Reihe wahr (d.h. leer) ist. Das deutet auf 0 als Anfangswert für i hin.

Zusätzlich sollte eine möglichst starke (enge) Aussage über den Wertebereich der neu eingeführten Variablen i in

die Schleifeninvariante aufgenommen werden. Der Anfangswert 0 (siehe den letzten Absatz oben) wird der Mindestwert sein. Wenn i=N, dann ist die Nachbedingung erfüllt. Deshalb ist es nicht notwendig, daß wir größere Werte zulassen.

Auf Grund dieser Überlegungen legen wir als Schleifeninvariante I fest:

$j \varepsilon Z$ **und** $0 \leq j \leq \max(0, M-N+1)$ **und** $i \varepsilon Z$ **und** $0 \leq i \leq N$
[Wertebereiche von i und j]

und $\bigwedge_{k=0}^{j-1}$ (**oder** $\bigvee_{a=0}^{N-1} D(k+a) \neq K(a)$) [Teilfolgen von D vor der j-ten ≠ K]

und ($j > M-N$ [keine Teilfolge von D = K]

oder $j \leq M-N$ **und** $\bigwedge_{a=0}^{i-1} D(j+a)=K(a)$) [Teilfolge von D ab j = K bis zur Stelle i-1]

Diese Betrachtungen führen auch zu dem Schluß, daß die Initialisierung die Folge von Zuweisungen

i:=0; j:=0

enthalten (oder daraus bestehen) muß.

5.3.4 Die while-Bedingung

Die Schleife kann dann beendet werden, wenn die Nachbedingung erfüllt ist. Wann folgt aus der Schleifeninvariante die Nachbedingung? Betrachtet man die Diagramme oben für die Nachbedingung und für die Schleifeninvariante, dann sieht man, daß die Nachbedingung erfüllt ist, entweder wenn der ?Bereich in K leer ist ($i > N-1$) oder wenn $j \geq M-N+1$ gilt. Die Endbedingung ist also

$i > N-1$ **oder** $j \geq M-N+1$

Die while-Bedingung ist die Negierung davon:

$i \leq N-1$ **und** $j < M-N+1$

oder äquivalent (weil i, j, M und N Ganzzahlen sind)

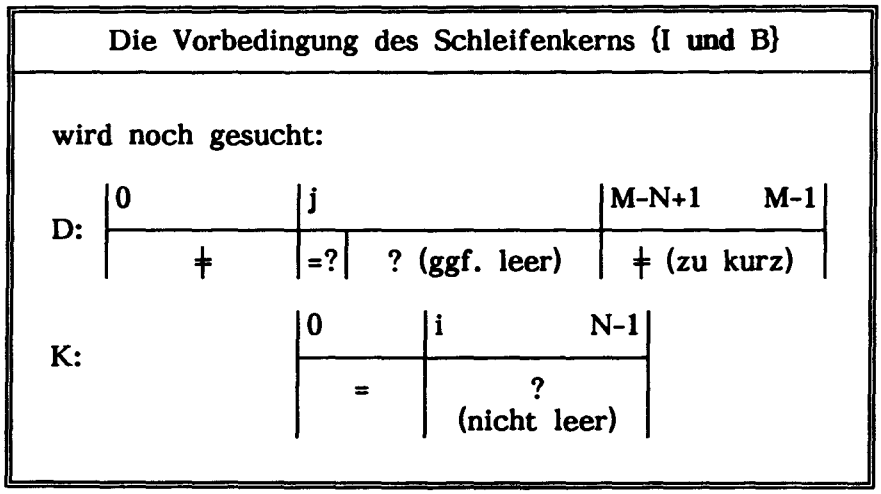
$i < N$ **und** $j \leq M-N$

Damit haben wir die Initialisierung und die while-Bedingung bestimmt. Es bleibt nur noch der Schleifenkern.

5.3.5 Der Schleifenkern

Der Schleifenkern muß dafür sorgen, daß (1) Fortschritt in Richtung der Nachbedingung erzielt wird (damit Terminierung bewiesen werden kann) und (2) die Schleifeninvariante erfüllt bleibt.

Vor jeder Ausführung des Schleifenkerns werden sowohl die Schleifeninvariante als auch die while-Bedingung wahr sein, d.h. {I und B} wird wahr sein. Im vorliegenden Fall wird also der obere Teil des Diagramms für die Schleifeninvariante zutreffen, wobei der ?Bereich in K nicht leer sein wird:



Fortschritt in Richtung der Nachbedingung wird dadurch erzielt, daß i oder j erhöht wird. Weil die Schleifeninvariante erfüllt bleiben muß, darf i nur dann erhöht werden, wenn die entsprechenden Elemente der Felder D und K gleich sind, d.h., wenn D(j+i)=K(i). Diese Beobachtung deutet auf eine if-Anweisung mit dieser Bedingung hin. Die Variable j darf nur dann erhöht werden, wenn die Teilfolge von D, die an der Stelle j anfängt, ungleich K ist – z.B. weil D(j+i)≠K(i). In diesem Falle muß i auf 0 zurückgesetzt werden, um die Wahrheit der Schleifeninvariante sicherzustellen.

Der Schleifenkern besteht also aus der folgenden if-Anweisung:

if D(j+i)=K(i) **then** i:=i+1 **else** j:=j+1; i:=0 **endif**

5.3.6 Der gesamte Programmteil

Wir setzen die oben ermittelten Anweisungen zusammen und erhalten den gewünschten Programmteil:

i:=0; j:=0
while i<N **und** j≤M-N **do**
  if D(j+i)=K(i) **then** i:=i+1 **else** j:=j+1; i:=0 **endif**
endwhile

5.3.7 Die Vorbedingung

Im Abschnitt 5.3.2 wurde gesagt, daß die Vorbedingung nicht präzise vorgegeben wurde und deshalb die vorläufige Vorbedingung nach der Konstruktion geprüft und ggf. revidiert werden soll. Wir ermitteln deshalb hier eine Vorbedingung des Programmteils. Die Schleifeninvariante I muß nach der Initialisierung wahr sein. Die Vorbedingung von I bezüglich der Initialisierung ist (vgl. Beweisregeln F1 und Z1)

$$(I^{j}_{0})^{i}_{0} = (0 \leq N)$$

Die im Abschnitt 5.3.2 unterstellte Vorbedingung ist stärker und deshalb gemäß der Beweisregel B1 tatsächlich eine Vorbedingung bezüglich des hier konstruierten Programms.

Übung:

1. Welche Bedeutung hat N=0? M=0? Was bewirkt das Programm und was bedeutet die Nachbedingung in diesen Fällen?
2. Die im Abschnitt 5.3.7 ermittelte Vorbedingung läßt zu, daß M<0. Interpretiere die Nachbedingung und die Wirkung des Programmteils in diesem Falle.
3. In der Schleifeninvariante tritt der Term j≤max(0,M-N+1) auf. Zeige, daß im allgemeinen

   [j≤max(a,b)] = [j≤a **oder** j≤b]

   und

[j≤min(a,b)] = [j≤a **und** j≤b]

4. Beweise die Korrektheit des hier konstruierten Programmteils. ■

5.4 Konstruktionsbeispiel: Auffinden des nächsten Namens in einem Feld von Zeichenketten

Gegeben ist ein Feld, wovon jedes Element (jede Feldvariable) eine Zeichenkette (eine Folge von einzelnen Zeichen) ist. Eine Feldvariable wird hier auch "Zeile" genannt. In diesem Feld von Zeilen stehen Namen. Ein Name ist eine Folge von einem oder mehreren Zeichen außer dem Leerzeichen. Ein oder mehrere aufeinanderfolgende Leerzeichen trennen Namen voneinander. Ein Name befindet sich immer innerhalb einer Zeile, d.h. ein Zeilenbruch trennt Namen.

Es soll der erste Name gefunden werden, der an oder nach einer angegebenen Stelle anfängt. Der eventuell gefundene Name soll als Ergebnis des zu konstruierenden Unterprogramms an das aufrufende Programm zurückgegeben werden.

Die Eingabevariablen des zu konstruierenden Unterprogramms sind F (das Feld), n (die Anzahl von Zeilen im Feld F), az (Anfangszeilennummer) und ap (Anfangspositionsnummer). Das Feld besteht aus den Feldvariablen F(1), F(2), ... F(n). Die Suche soll in der Position ap von F(az) anfangen.

Ein Feld könnte zum Beispiel wie folgt aussehen. Dabei markiert ■ das Ende der jeweiligen Zeile. In diesem Beispiel ist n=3.

F(1): Eingang1 Eingang5 Ausgang1 ■
F(2): Eingang3 Eingang7 Ausgang3■
F(3):Eingang2 Eingang4 Ausgang4 ■

Eine Zeile darf leer sein, d.h. 0 Zeichen lang sein. Eine Zeile darf auch nur Leerzeichen beinhalten. Das Feld darf leer sein (n=0).

5.4.1 Vorbereitende Betrachtungen

Wir werden uns oft auf eine bestimmte Position einer bestimmten Zeile beziehen müssen. Wir sollten deshalb zuerst vereinbaren, wie wir eine solche Kombination von Zeile und

Position identifizieren wollen. Insbesondere sollten wir die Wertebereiche der (ganzzahligen) Zeilen- und Positionsnummern festlegen.

Auf jeden Fall müssen wir Zeilennummern von 1 bis n einschließlich zulassen. Dieser Bereich reicht jedoch nicht aus, z.B. wenn n=0. Um diesen Fall abdecken zu können, müssen wir entweder mit 0 anfangen oder n+1 als maximalen Wert einer Zeilennummer zulassen. Wir vereinbaren letzteres:

$$1 \leq \text{Zeilennummer} \leq n+1$$

Entsprechend vereinbaren wir, daß Positionsnummern mit 1 anfangen und bis zur Länge der fraglichen Zeile zuzüglich 1 laufen:

$$1 \leq \text{Positionsnummer} \leq \text{Länge}(F(z))+1$$

wo "Länge" eine angenommene Funktion ist, die die Anzahl der Zeichen in einer Zeichenkette ermittelt. Die oben stehende Bedingung hat nur dann Bedeutung, wenn sich die Zeilennummer z auf eine tatsächlich vorhandene Zeile bezieht, d.h., wenn $1 \leq z \leq n$. Wenn dies nicht der Fall ist, d.h., wenn $z=n+1$, liegt es nahe, den Wertebereich für die Positionsnummer so festzulegen, als ob die Zeile leer wäre, also die Länge 0 hätte. In einem solchen Falle muß also die Positionsnummer gleich 1 sein.

Wir nehmen ferner an, daß in der fraglichen Programmiersprache eine Funktion existiert, die eine Teilkette aus einer Zeichenkette ermittelt. Wir nennen diese Funktion "mid" und setzen voraus, daß mid(K,p,L) gleich der Teilkette ist, die in Position p der Zeichenkette K anfängt und L Zeichen lang ist. Damit ist mid(K,p,1) das eine Zeichen, das in Position p in K steht. Ferner setzen wir voraus, daß diese Funktion die Positionen einer Zeichenkette ab 1 (nicht 0) zählt.

5.4.2 Die Spezifikation

Die Eingabevariablen für das zu konstruierende Unterprogramm sind n, das Feld F, az und ap (siehe die allgemeine Beschreibung der Aufgabe oben). Wir setzen voraus, daß die Werte von az und ap zulässige Zeilen- bzw. Positionsnummern sind. Damit ist die Vorbedingung

```
nεZ und azεZ und apεZ                    [n, az, ap Ganzzahlen]
und 0≤n und 1≤az≤n+1                     [Wertebereiche für n, az]
und (az=n+1 und ap=1                     [am Ende des Felds]
     oder az≤n und 1≤ap≤Länge(F(az))+1)  [innerhalb
                                          des Felds]
```

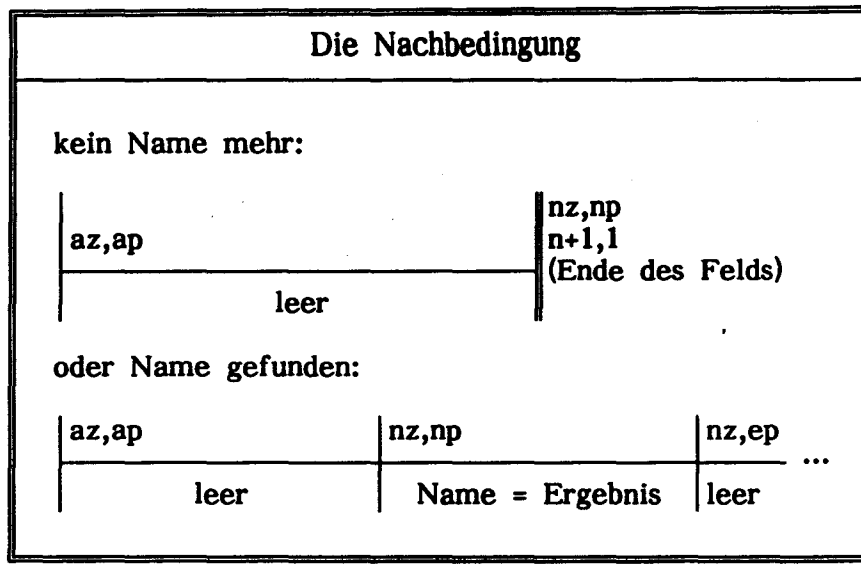
Da eine präzise Nachbedingung nicht vorgegeben ist, müssen wir sie formulieren. Es soll der erste Name gefunden werden, der an oder nach der Stelle (az,ap) anfängt. Dabei müssen wir berücksichtigen, daß eventuell kein Name mehr vorkommt. Ferner dürfen Leerzeichen und Zeilenbrüche vor einem vorhandenen Name stehen. Der ggf. gefundene Name soll als Ergebnis des zu konstruierenden Unterprogramms an das aufrufende Programm zurückgegeben werden.

Zunächst müssen wir zwischen zwei Möglichkeiten unterscheiden: Es kommt kein Name ab der Stelle (az,ap) mehr vor, oder es kommt (mindestens) einer vor.

Es kommt kein Name mehr vor, wenn der Bereich von der Stelle (az,ap) bis zum Ende des Felds – Stelle (n+1,1) – nur Leerzeichen und Zeilenbrüche enthält, eine Bedingung, die wir unten mit "Leer(az,ap,n+1,1)" bezeichnen. Wenn kein Name mehr im Feld F vorkommt, soll das ermittelte Ergebnis auf diese Situation hindeuten, z.B. dadurch, daß die leere Zeichenkette als Ergebnis an das aufrufende Programm zurückgegeben wird.

Falls ein Name vorkommt, muß das zu konstruierende Unterprogramm seine Lage (Anfang und Ende) bestimmen. Wir lassen die Anfangsstelle durch die Werte der Variablen nz (Zeile) und np (Position) festhalten. Der Bereich zwischen der Stelle (az,ap) und dem Anfang des Namens (nz,np) muß natürlich leer sein. Wir lassen die Variable ep die Endstelle (die in der gleichen Zeile wie die Anfangsstelle liegen muß) angeben. Das Ergebnis des Unterprogramms muß dann dem dazwischen liegenden Namen gleich sein.

In diagrammatischer Form ist die Nachbedingung



In algebraischer Form ist die Nachbedingung

Leer(az,ap,nz,np) [leere Zone vor dem Namen]
und {nz=n+1 **und** Ergebnis=leere Zeichenkette
[kein Name mehr]
oder
nz≤n **und** Namenlage(nz,np,ep) [Name gefunden]
und Ergebnis=mid(F(nz),np,ep-np)}

Formal müßten wir noch die Funktionen Leer(...) und Namenlage(...) definieren sowie die Wertebereiche der Variablen nz, np und ep eingrenzen.

Übung:
1. Ergänze die Nachbedingung oben durch Angaben über die Wertebereiche der Variablen nz, np und ep.
2. Definiere formal die Funktionen Leer(...) und Namenlage(...).
3. Es wäre eventuell sinnvoll, in der Vorbedingung auch zu fordern, daß die Stelle (az,ap) nicht innerhalb eines Namens liegt. Schreibe eine entsprechende Ergänzung zur Vorbedingung. ■

5.4.3 Die Grundstruktur des Unterprogramms

Es liegt nahe, die Nachbedingung (siehe oben) schrittweise zu verwirklichen. Im ersten Schritt werden Leerzeichen

und Zeilenbrüche übersprungen, bis entweder der Anfang eines Namens oder das Ende des Felds erreicht wird. Das eigentliche Ergebnis des ersten Schritts sind die Werte der Variablen nz und np. Im zweiten Schritt wird ggf. die Lage des Endes des Namens bestimmt. Das Ergebnis des zweiten Schritts ist der Wert der Variablen ep. Im letzten Schritt wird das Ergebnis des Unterprogramms – der gefundene Name oder die leere Zeichenkette – ermittelt und der Ausgabevariablen zugewiesen.

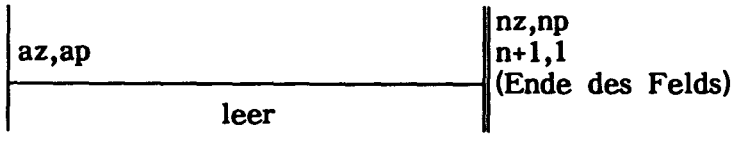
Die Grundstruktur des Unterprogramms wird danach eine Folge von Anweisungen sein. Der erste sowie der zweite Schritt werden jeweils im wesentlichen aus einer Schleife bestehen.

5.4.4 Schritt 1: Bestimmung des Endes der leeren Zone

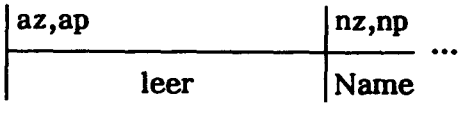
Wir fangen unsere Überlegungen wie üblich mit dem Ziel – der Nachbedingung – an. Die Nachbedingung des ersten Schritts (Bestimmung des Endes der leeren Zone) ist

Die Nachbedingung des ersten Schritts

kein Name mehr:



oder Anfang eines Namens gefunden:



Als Schleifeninvariante wählen wir die Aussage, daß der Bereich ab der Stelle (az,ap) bis zur Stelle vor (nz,np) leer ist, d.h. höchstens Leerzeichen und Zeilenbrüche enthält. Zur Schleifeninvariante gehört auch die Aussage, daß die Werte der Variablen nz und np in den zulässigen Wertebereichen für Zeilen- und Positionsnummern liegen (siehe Abschnitt 5.4.1 oben).

Anfangs kann die Wahrheit der Schleifeninvariante dadurch sichergestellt werden, daß die Variablen nz und np auf az bzw. ap initialisiert werden.

Die Endbedingung für unsere Schleife ist die Aussage, daß entweder (1) das Ende des Felds erreicht (nz=n+1) oder (2) der Anfang eines Namens – ein Zeichen außer einem Leerzeichen innerhalb einer Zeile innerhalb des Felds – gefunden wurde:

```
nz>n
oder nz≤n und np≤Länge(F(nz))
     und mid(F(nz),np,1)≠Leerzeichen
```

oder, äquivalent,

```
nz>n
oder np≤Länge(F(nz)) und mid(F(nz),np,1)≠Leerzeichen
```

Die Negierung dieser Endbedingung ist die while-Bedingung:

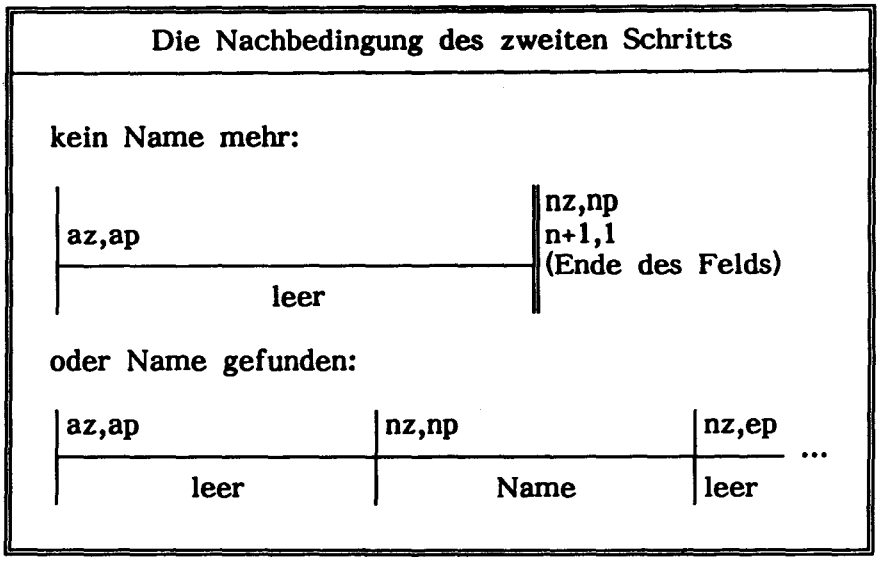
```
nz≤n
und (np>Länge(F(nz)) oder mid(F(nz),np,1)=Leerzeichen)
```

Wir müssen nur noch den Schleifenkern konstruieren. Die while-Bedingung wird nur dann wahr sein, wenn sich an der Stelle (nz,np) entweder ein Zeilenbruch oder ein Leerzeichen befindet. Der Schleifenkern soll also diese Stelle überspringen, wobei die Schleifeninvariante aufbewahrt werden muß. Insbesondere müssen die Werte der Variablen nz und np in den zulässigen Wertebereichen bleiben. Falls ein Zeilenbruch vorliegt, muß nz um 1 erhöht und np auf 1 gesetzt werden; falls ein Leerzeichen gefunden wurde, ist die Erhöhung von np um 1 angebracht. Der erste Schritt unseres Programms wird dadurch:

```
nz:=az; np:=ap
while nz≤n und (np>Länge(F(nz))
                oder mid(F(nz),np,1)=Leerzeichen) do
   if np>Länge(F(nz))
   then nz:=nz+1; np:=1 else np:=np+1 endif
endwhile
```

5.4.5 Schritt 2: Bestimmung des Endes des Namens

Die Nachbedingung des zweiten Schritts (Bestimmung des Endes des Namens) ist



Unsere Vorbedingung ist die Nachbedingung des vorherigen (ersten) Schritts. Insbesondere sind gültige Werte für nz und np bereits errechnet worden. Falls nz=n+1, ist die Nachbedingung des zweiten Schritts erfüllt. Sonst (nz≤n) muß das Ende des Namens bestimmt werden. Diese Fallunterscheidung deutet auf eine if-Anweisung hin.

Das Ende des Namens kann dadurch bestimmt werden, daß alle zum Namen gehörenden Zeichen übersprungen werden. Diese Überlegung deutet auf das wiederholte Vergleichen mit einem Leerzeichen. Die Wiederholung deutet auf eine Schleife hin. Die Schleifeninvariante ist offensichtlich: Jede Stelle ab (nz,np) bis zur Stelle vor (nz,ep) enthält ein Zeichen außer einem Leerzeichen. Zur Schleifeninvariante gehört auch die Aussage, daß der Wert der Variablen ep im zulässigen Wertebereich einer Positionsnummer für die Zeile nz liegt.

Die anfängliche Wahrheit der Schleifeninvariante kann durch die Initialisierung der Variablen ep auf entweder np oder np+1 sichergestellt werden.

Die Endbedingung unserer Schleife ist die Aussage, daß die Stelle (nz,ep) nicht mehr zum Namen gehört. Das wird der Fall sein, sobald entweder das Ende der Zeile erreicht

ist (ep>Länge(F(nz))) oder die Stelle (nz,ep) ein Leerzeichen enthält:

ep>Länge(F(nz)) **oder** mid(F(nz),ep,1)=Leerzeichen

Die while-Bedingung ist die Negierung davon:

ep≤Länge(F(nz)) **und** mid(F(nz),ep,1)≠Leerzeichen

Der Schleifenkern hat nur die Stelle (nz,ep), die ein Zeichen außer dem Leerzeichen enthält, in den Bereich des Namens aufzunehmen, etwa durch Erhöhung der Variablen ep um 1 (siehe Schleifeninvariante oben).

Der zweite Schritt unseres Programms ist also:

```
if nz≤n
then ep:=np+1
     while ep≤Länge(F(nz))
           und mid(F(nz),ep,1)≠Leerzeichen do
        ep:=ep+1
     endwhile
endif
```

5.4.6 Schritt 3: Ermittlung des Ergebnisses

Der dritte Schritt unseres Programms hat nur die Aufgabe, das Ergebnis zu ermitteln. Falls ein Name gefunden wurde, soll das Ergebnis die Teilkette sein, die an der Stelle (nz,np) anfängt und an der Stelle (nz,ep-1) endet. Sonst (falls kein Name vorhanden ist) soll das Ergebnis die leere Zeichenkette sein. Diese Fallunterscheidung deutet auf eine if-Anweisung hin. Der letzte Teil des zu konstruierenden Programms ist:

```
if nz≤n
then Ergebnis:=mid(F(nz),np,ep-np)
else Ergebnis:=leere Zeichenkette
endif
```

5.4.7 Das gesamte Programm

Wir stellen die einzelnen Programmschritte zusammen und kombinieren dabei die zwei if-Anweisungen. Damit ist das gesamte Programm wie folgt:

```
nz:=az; np:=ap
while nz≤n und (np>Länge(F(nz))
                oder mid(F(nz),np,1)=Leerzeichen) do
   if np>Länge(F(nz))
   then nz:=nz+1; np:=1 else np:=np+1 endif
endwhile

if nz≤n
then ep:=np+1
     while ep≤Länge(F(nz))
           und mid(F(nz),ep,1)≠Leerzeichen do
        ep:=ep+1
     endwhile
     Ergebnis:=mid(F(nz),np,ep-np)
else Ergebnis:=leere Zeichenkette
endif
```

Es wäre eventuell mit dem Systemkonstrukteur zu klären, ob den Variablen az und ap neue Werte zugewiesen werden sollten, etwa in Vorbereitung auf einen folgenden Aufruf auf dieses Programm (um den nächsten Namen zu finden). Wenn ja, dann müßte die Nachbedingung entsprechend geändert und ergänzt werden. Die zusätzlichen Zuweisungen wären az:=nz und ap:=ep (am Ende des then-Zweigs) bzw. ap:=1 (am Ende des else-Zweigs).

5.5 Zusammenfassung über die Konstruktion beweisbar korrekter Programme

In diesem Kapitel wurden vier Beispiele von Konstruktionsaufgaben behandelt. In jedem Falle gingen wir dabei von den Vor- und Nachbedingungen – d.h. von der Spezifikation des zu konstruierenden Programms – aus. Bevor wir Zeit in die Programmierung investierten, schufen wir Klarheit darüber, was das Programm berechnen bzw. erreichen soll – mit anderen Worten, was wir überhaupt konstruieren wollten. Denn: Wenn man nicht weiß, was das Ziel ist, dann wird man es sehr wahrscheinlich nicht erreichen.

Es lohnt sich immer, Zeit in die klare, vollständige Formulierung der Nachbedingung zu investieren. Nicht nur der Programmierer des fraglichen Programmteils gewinnt dabei, auch jeder Programmierer, der einen Aufruf darauf schreiben will, zieht Vorteile aus klar und präzise formulierten Vor-

und Nachbedingungen. Eine derartige Spezifikation sagt ihm genau und vollständig, wofür er vor dem Aufruf sorgen muß und womit er nach dem Rücksprung rechnen kann.

Nachdem wir uns Klarheit über die Spezifikation des zu konstruierenden Programmteils geschaffen hatten, entschieden wir uns für die Grundstruktur: ob Schleife, if-Anweisung oder Folge. Die Wiederholung eines Vorgangs deutet auf eine Schleife hin; eine Fallunterscheidung auf eine if-Anweisung. Wenn die Nachbedingung aus Teilausdrücken besteht, die aufeinander aufbauen, bietet sich eine Folge von einzelnen Schritten an, wovon jeder für die Erfüllung des entsprechenden Teilausdrucks der Nachbedingung sorgt.

Nach der Wahl einer Schleife legten wir die Schleifeninvariante fest, indem wir die Vor- und Nachbedingung verallgemeinerten. Dabei war die Nachbedingung der wichtigere Ausgangspunkt.

Aus dem Unterschied zwischen der Schleifeninvariante und der Nachbedingung leiteten wir die Endbedingung und ihre Negierung, die while-Bedingung, ab. Dabei stellten wir uns die Fragen, "wann stellt die Schleifeninvariante auch die Nachbedingung dar?" oder "unter welcher Bedingung folgt aus der Schleifeninvariante die Nachbedingung?"

Jedesmal konstruierten wir den Schleifenkern so, daß er (1) Fortschritt in die Richtung der Nachbedingung erzielte und (2) die Wahrheit der Schleifeninvariante aufbewahrte. Andere Überlegungen sind bei der Konstruktion eines Schleifenkerns grundsätzlich überflüssig.

Die Initialisierung jeder Schleife konstruierten wir so, daß die anfängliche Wahrheit der Schleifeninvariante sichergestellt war. Auch hier sind andere Überlegungen prinzipiell überflüssig.

Wie in allen Dingen: Übung macht den Meister. Die hier geschilderte, allgemein anwendbare Vorgehensweise ist nicht schwierig zu lernen, aber sie ist auch nicht trivial leicht. Software-Entwickler, die sich die Mühe gemacht haben, meinen, es hat sich auf jeden Fall gelohnt. Auch die Benutzer ihrer Software sind darüber froh.

6. Formulierung von Vor- und Nachbedingungen

6.1 Boolesche Ausdrücke: eine Sprache

Bevor man anfangen kann, die Korrektheit eines Programms zu beweisen, müssen die Vor- und Nachbedingungen in geeigneter Form – logische algebraische Ausdrücke – vorliegen. Hat der Systemkonstrukteur sie nicht vorgegeben, dann muß der Programmierer des fraglichen Programmteils die Vor- und Nachbedingungen in dieser Form selbst schreiben.

Mancher, der noch wenig Erfahrung mit der Programm-Korrektheitsbeweisführung gesammelt hat, findet diesen wichtigen Schritt zunächst etwas schwierig. Wir wollen uns deshalb in diesem Kapitel mit diesem Thema beschäftigen.

Im Grunde genommen ist die Aufgabe, von einer verbalen Beschreibung der Programmspezifikation ausgehend die Vor- und Nachbedingungen zu formulieren, nichts anderes als eine Übersetzung zwischen zwei verschiedenen Sprachen. In diesem Falle ist die Zielsprache ein Teil der Sprache der Mathematik.

Die Fähigkeit, in eine bestimmte Sprache zu übersetzen, setzt im allgemeinen aktive Kenntnisse der Zielsprache voraus. Diese Binsenweisheit gilt auch dann, wenn die Zielsprache, wie hier, die mathematische Sprache ist. Übung im Lesen, in der Interpretation und in der Manipulation von logischen Ausdrücken hilft sehr, Gedanken in logischen algebraischen Ausdrücken formulieren zu können.

Betrachte deshalb die logische (Boolesche) Algebra als eine Sprache. Lies logische Ausdrücke oft, übersetze sie ins Deutsche und schreibe in dieser Sprache, um Übung und Erfahrung zu sammeln und um aktive Kenntnisse zu gewinnen. Siehe z.B. die Erarbeitung der Nachbedingungen in den Abschnitten 5.1.1, 5.2.1, 5.3.2 und 5.4.2.

Die Sprache der Booleschen Ausdrücke, von der wir in diesem Buch Gebrauch machen, basiert auf den logischen Funktionen **und**, **oder** und **nicht**. Die Implikation ($\Longrightarrow$) ist eine nützliche, aber nicht absolut notwendige Ergänzung dazu. (Siehe den Abschnitt "Bezeichnungen" am Anfang dieses Buchs und den Anhang A.) Die Teilausdrücke, die mit

den logischen Funktionen kombiniert werden, können mit Hilfe anderer mathematischer Funktionen (z.B. <, >, =, +, -, *, / usw.) gebildet werden.

6.2 Übersetzen von Deutsch in die Sprache der logischen Algebra

Wenn eine allgemeine Beschreibung der Wirkung eines Programms vorliegt und die Vor- und Nachbedingungen formuliert werden sollen, geht man zweckmäßigerweise wie folgt vor: Aus der Beschreibung der *Wirkung* der Ausführung des Programms leitet man Beschreibungen der vorherigen und der nachherigen *Zustände* ab. Beziehungen zwischen den Werten der verschiedenen Programmvariablen sollen den Mittelpunkt solcher Beschreibungen bilden.

Diese Beschreibungen der vorherigen und nachherigen Zustände werden stufenweise präziser formuliert, bis sie in der Sprache der logischen Algebra ausgedrückt werden können. Die dadurch entstehenden Booleschen Ausdrücke werden ggf. vereinfacht. Die Ergebnisse sind die Vor- und Nachbedingungen.

Bei diesem Übersetzungsprozeß ist es manchmal vorteilhaft, Skizzen und Diagramme verschiedener Art zu verwenden. Kapitel 5 enthält mehrere Beispiele.

6.3 Zusätzliche Hinweise für Vor- und Nachbedingungen sowie Schleifeninvarianten

Es ist zu empfehlen, Aussagen über die Wertebereiche aller maßgeblichen Programmvariablen in die Vor- und Nachbedingungen aufzunehmen. In der Vorbedingung sind normalerweise möglichst schwache (wenig restriktive) Aussagen anzustreben. In der Nachbedingung sollen sie in der Regel möglichst stark (restriktiv) sein. Die Nachbedingung soll unbedingt Aussagen über die Wertebereiche aller Programmvariablen enthalten, die der fragliche Programmteil berechnet.

Entsprechend soll jede Schleifeninvariante möglichst starke (restriktive) Aussagen über die Wertebereiche aller Programmvariablen enthalten, deren Werte der Schleifenkern verändert.

Auch ist zu empfehlen, vor allem Nachbedingungen in der Form

(A1 **und** ...) **oder** (A2 **und** ...) **oder** ... (An **und** ...)

oder ggf.

...
und [(A1 **und** ...) **oder** (A2 **und** ...) **oder** ... (An **und** ...)]

zu schreiben, wobei A1, A2 usw. einfache und sich gegenseitig ausschließende Bedingungen sind. Auf diese Weise wird eine Fallunterscheidung im darauffolgenden Programmteil erleichtert. Vgl. die Nachbedingungen in den Abschnitten 5.1.1, 5.3.2 und 5.4.2.

6.4 Ein kleines Glossar für Deutsch—Boolesche Algebra

In der deutschen Beschreibung der Wirkung eines Programms oder der vorherigen und nachherigen Zustände kommen häufig gewisse Formulierungen vor, die bestimmten logischen Ausdrucksformen entsprechen. Diese können sofort in die Zielsprache übersetzt werden, um die Basis für den Rest der Vor- bzw. Nachbedingung zu bilden.

Häufig vorkommende verbale Formulierungen entsprechen logischen Funktionen und Ausdrücken wie folgt:

Deutsch	Boolesche Algebra
und, aber	**und**
oder	**oder**
(für) alle, jede	**und**-Reihe
(für) kein	**und**-Reihe mit negierter Aussage
es gibt, es existiert, (für) irgendein	**oder**-Reihe
sortiert $[A(1)\leq A(2)\leq \ldots \leq A(n)]$	$\mathbf{und}_{i=1}^{n-1}\ A(i)\leq A(i+1)$
Ganzzahl	$\ldots \epsilon Z$
wenn (falls) ... dann gilt ...	$\Longrightarrow$
suchen, finden, gleich, vorhanden	=
vertauschen, umordnen, andere Reihenfolge, zusammenfügen, kopieren, sortieren	Permutation (siehe Abschnitt 6.5)

6.5 Beispiele für die Übersetzung in die Sprache der logischen Algebra

In einigen Beispielen in Kapitel 5 haben wir, von verbalen Beschreibungen der Programmierungsaufgaben ausgehend, Vor- und Nachbedingungen erarbeitet. Siehe insbesondere die Abschnitte 5.1.1, 5.2.1, 5.3.2 und 5.4.2. Unten werden wir noch einige Beispiele des Übersetzungsprozesses anschauen.

6.5.1 Zusammenfügen

Betrachte die folgende grobe Spezifikation eines Unterprogramms, die wir in logischer algebraischer Form aus-

drücken wollen: "Die in den Feldern A und B gespeicherten Werte sollen zusammengefügt und in das Feld C abgelegt werden. Vorher sind die Felder A und B sortiert. Nachher soll das Feld C sortiert sein. Die Variablen na und nb geben an, wieviele Werte im Feld A bzw. B stehen."

Der Begriff "sortiert" deutet auf **und**-Reihen in den Vor- und Nachbedingungen hin (siehe Abschnitt 6.4). Unklar ist, welcher Wertebereich für die Indexwerte gilt, z.B. ob sie mit 0, mit 1 oder mit einem anderen Wert anfangen sollen. Wir nehmen an, sie fangen mit 1 an. Diese Annahme sollte eigentlich mit dem Systemkonstrukteur abgestimmt werden.

Die Vorbedingung soll ausdrücken, daß die Felder A und B sortiert sind. Ferner soll sie eine Aussage über die Wertebereiche der Variablen na und nb enthalten. Nur ganzzahlige Werte sind sinnvoll. Negative Werte sind offensichtlich bedeutungslos. Werte ab 1 sind auf jeden Fall als normal zu betrachten und müssen zugelassen werden. Wenn na=0 oder nb=0, wäre das entsprechende Feld leer. Auch wenn ein solcher Fall nicht "normal" wäre, wäre er von Bedeutung. Weil eine Vorbedingung grundsätzlich möglichst schwach (allgemein) gehalten werden soll, lassen wir 0 als Wert für na und nb zu. Die Vorbedingung wird:

$na \varepsilon Z$ **und** $nb \varepsilon Z$ **und** $na \geq 0$ **und** $nb \geq 0$ [Wertebereiche von na, nb]

$\mathbf{und}_{i=1}^{na-1}\ A(i) \leq A(i+1)\ \mathbf{und}_{i=1}^{nb-1}\ B(i) \leq B(i+1)$ [A, B sortiert]

In der Nachbedingung soll ausgedrückt werden, daß das Feld C sortiert ist und daß die nachherigen Werte im Feld C aus den Feldern A und B stammen. Eine entsprechende Nachbedingung ist wie folgt:

$\mathbf{und}_{i=1}^{na+nb-1}\ C(i) \leq C(i+1)$ [C sortiert]

und die Folge {C(1), C(2), ... C(na+nb)} ist eine Permutation der Folge {A(1), ... A(na), B(1), ... B(nb)} [Werte in C stammen von A und B]

Der letzte Term in der Nachbedingung besagt, daß die Werte im Feld C die gleichen sind wie die, die in den Feldern A und B enthalten sind. Nur die Reihenfolge darf eine andere

sein. Dieser Term verhindert, daß ganz andere Werte in C stehen, als die, die in A und B am Anfang standen.

Eine *Permutation* einer Folge ist eine Umordnung ihrer Glieder. Zwei Folgen sind Permutationen voneinander, wenn sie sich nur in der Reihenfolge ihrer Glieder unterscheiden. Kommt ein bestimmter Wert in einer Folge mehrmals vor, dann kommt dieser Wert genauso viele Male in einer Permutation der Folge vor.

6.5.2 Sortieren

Betrachte die folgende grobe Spezifikation eines Unterprogramms, die wir in logischer algebraischer Form ausdrücken wollen: "Gegeben sei ein Feld X. Die ganzzahligen Indexwerte fangen mit il an und laufen bis ir, wobei das Feld leer sein darf. Das zu konstruierende Programm soll die Werte in diesem Feld sortieren."

Die Vorbedingung muß die Beziehung zwischen den Werten der Variablen il und ir ausdrücken, die sich daraus ergibt, daß die Anzahl der Elemente im Feld X Null oder positiv ist: $ir-il+1 \geq 0$. Äquivalent und vielleicht etwas übersichtlicher als Vorbedingung ist $il-1 \leq ir$. Wenn wir die Aussage hinzufügen, daß il und ir ganzzahlige Werte zugeordnet werden müssen, wird unsere Vorbedingung

$il \in Z$ **und** $ir \in Z$ **und** $il-1 \leq ir$

Welcher Zustand soll durch die Ausführung des Programms erreicht werden? Nach der Ausführung des Programms sollen die im Feld X stehenden Werte sortiert sein. Ferner sollen diese Werte die gleichen sein, die sich vorher im Feld X befanden; sie werden nachher ggf. an anderen Stellen innerhalb des Felds stehen. In der Sprache der logischen Algebra ausgedrückt ist unsere Nachbedingung

$$\textbf{und}_{i=il}^{ir-1} \; X(i) \leq X(i+1) \qquad \text{[X sortiert]}$$

und die Folge {X(il), X(il+1), ... X(ir)} ist eine Permutation der Folge {X'(il), X'(il+1), ... X'(ir)}
[Werte in X waren ursprünglich in X]

Hier bedeutet der Apostroph (') den Wert der jeweiligen Feldvariablen *vor* der Ausführung des fraglichen Sortierprogramms.

6.5.3 Qualifizierte (bedingte) Bedingungen

Manchmal müssen wir in einer Vor- oder Nachbedingung eine bestimmte Bedingung B fordern, aber nur dann, wenn eine andere Bedingung A erfüllt ist. Wenn A nicht erfüllt (falsch) ist, ist es ohne Belang, ob B erfüllt (wahr) ist oder nicht. Diese bedingte Forderung wird dann gewöhnlich mit anderen Bedingungen in der Vor- bzw. Nachbedingung **und**-verknüpft. Für eine derartige Forderung ist die logische Implikation

$$A \Longrightarrow B$$

geeignet. Diese Implikation ist äquivalent zu (siehe Anhang A, Abschnitt A.3, Identität 22)

nicht A **oder** B

Im letzten Ausdruck ist besonders ersichtlich, daß der Wert von B ohne Belang ist, wenn A falsch ist. Wenn A wahr ist, vereinfacht sich dieser Ausdruck auf die Bedingung B.

Beispiel: In der Schleifeninvariante eines Unterprogramms, das die Werte aus den sortierten Feldern A und B in das ebenfalls sortierte Feld C übertragen soll, muß gefordert werden, daß das nächste zu übertragende Element im Feld A (A(ia)) größer oder gleich dem letzten Element im Feld C (C(ic-1)) sein muß, aber nur dann, wenn sowohl noch ein zu übertragendes Element im Feld A steht (ia≤na) als auch ein bereits übertragenes Element im Feld C vorhanden ist (ic>1). Die entsprechende "bedingte Bedingung" ist

ia≤na **und** ic>1 $\Longrightarrow$ C(ic-1)≤A(ia)

Diese Implikation ist äquivalent zum Ausdruck

ia>na **oder** ic≤1 **oder** C(ic-1)≤A(ia)

der wie folgt zu lesen (zu interpretieren, ins Deutsche zu übersetzen) ist: Die gestellte Forderung ist dann erfüllt, wenn entweder

- es kein zu übertragendes Element im Feld A mehr gibt oder
- bisher kein Element nach Feld C übertragen wurde (Feld C ist leer) oder

• das letzte Element im Feld C kleiner oder gleich dem nächsten Element im Feld A ist.

Die entsprechende Forderung hinsichtlich des Felds B muß auch gestellt werden:

ib≤nb **und** ic>1 ⟹ C(ic-1)≤B(ib)

Siehe auch Abschnitt 4.5 oben. ■

6.6 Zusammenfassung über die Formulierung von Bedingungen

Die von verbalen Beschreibungen der Anforderungen ausgehende Formulierung von Vor- und Nachbedingungen ist ein Übersetzungsprozeß. Deshalb setzt die Fähigkeit, Vor- und Nachbedingungen zu verfassen, aktive Kenntnisse der Zielsprache – eines Teils der Sprache der Mathematik – voraus. Um zu diesem Zweck die logische Algebra ausreichend zu beherrschen, muß man einen gewissen – aber nicht sonderlich großen – Lernaufwand betreiben.

7. Zusammenfassung

7.1 Die theoretische Grundlage für fehlerfreie Software in der Praxis

In den letzten Jahrzehnten ist eine mathematische und theoretische Grundlage erarbeitet worden, deren Anwendung in der Software-Entwicklungspraxis die Konstruktion fehlerfreier Software ermöglicht. Die Natur dieser Grundlage, die Art und Weise, wie sie angewendet werden kann sowie die Qualität der dadurch entstehenden Programmentwürfe – insbesondere hinsichtlich der Entwurfsfehlerfreiheit – weisen starke Ähnlichkeiten mit den entsprechenden Aspekten der klassischen Ingenieurwissenschaften auf. Erst diese mathematische und theoretische Grundlage ermöglicht eine wirklich ingenieurwissenschaftliche Vorgehensweise bei der Software-Entwicklung.

Gegenstand des vorliegenden Buchs ist die praktische Anwendung dieser Grundlage, nicht die theoretische Grundlage selbst. Wir haben hier ihre wesentlichsten Aspekte in der Form von Beweisregeln kennengelernt. Die Beweisregeln finden direkte Anwendung bei der Programm-Korrektheitsbeweisführung – sowohl bei der Verifikation einer Korrektheitsaussage (bestehend aus einer Vorbedingung, einem Programm oder Programmteil und einer Nachbedingung) als auch bei der Ermittlung einer Vorbedingung für einen gegebenen Programmteil und eine gegebene Nachbedingung.

Bei der Verifikation einer Korrektheitsaussage (-behauptung) wendet man die Beweisregeln an, um die ursprüngliche Korrektheitsaussage entsprechend der Programmstruktur in Korrektheitsaussagen über die Bestandteile des Programms zu zerlegen. Dieser Prozeß wird iterativ fortgesetzt, bis nur Korrektheitsaussagen über Zuweisungen und bereits bewiesene Aussagen (z.B. über Unterprogramme) übrigbleiben. Korrektheitsaussagen über Zuweisungen werden durch Anwendung dafür geeigneter Beweisregeln unmittelbar verifiziert.

Durch die hierarchische Gliederung eines Programms in Unterprogramme jeweils begrenzter Größe und die entsprechende Zerlegung des Korrektheitsbeweises in übersichtliche Teilbeweise kann auch für ein großes Programm die Korrektheit praktisch bewiesen werden.

Vielleicht noch wichtiger für die Software-Entwicklungspraxis lassen sich von den Beweisregeln und ihrer Anwendung Richtlinien für die Programmkonstruktion ableiten. Diese Richtlinien lenken die Aufmerksamkeit des Software-Entwicklers auf die wesentlichen und maßgebenden Aspekte des Programms und von unwesentlichen und unwichtigen Aspekten ab. Der Software-Entwickler arbeitet gezielter als bei der herkömmlichen Weise. Typischerweise ist das Ergebnis ein kompakteres Programm, das einfacher zu verstehen ist. Oft können mehrere Teile des zu konstruierenden Programms systematisch abgeleitet werden. Der Korrektheitsbeweis (oder mindestens eine detaillierte Skizze davon) entsteht als Nebenprodukt der Programmkonstruktion.

Mit Hilfe der Programm-Korrektheitsbeweisführung kann der Software-Entwickler prüfen, ob sein Programmentwurf die gestellten Anforderungen (die Spezifikation, das Pflichtenheft) erfüllt – bevor er das Programm laufen läßt oder "testet" – genau wie Ingenieure anderer Fachrichtungen es seit langem tun. Vor allem wird dabei festgestellt, unter welchen Bedingungen das Programm richtige Ergebnisse sicher liefern wird, d.h., wann man Vertrauen in die Ergebnisse haben kann (und wann nicht). Diese Kenntnisse sind eine unerläßliche Voraussetzung dafür, daß der Software-Entwickler eine ingenieurmäßige Verantwortung für das Programm übernehmen kann.

Dieses Buch hat nur logische Aussagen (Vor- und Nachbedingungen usw.) zum Gegenstand, die sich auf die *Werte* von Programmvariablen beziehen. Solche Aussagen sind die wichtigsten, die in der Praxis vorkommen, und betreffen die wesentlichsten und problematischsten Aspekte der Programmlogik und der Programmkorrektheit. Weiterführende Aussagen, z.B. über die Struktur von Datenumgebungen, werden in der wissenschaftlichen Fachliteratur behandelt [Baber, 1987]. Für den professionellen Software-Entwickler lohnt es sich auch, die Theorie, worauf der in diesem Buch vorgestellte Stoff basiert, zu studieren (siehe die Literaturhinweise).

7.2 Software-Entwicklung morgen

Im Abschnitt 1.1 wurde bereits erwähnt, daß sich der Einsatz von Rechnersystemen in unserer Gesellschaft immer mehr ausweitet. Die Anzahl der Anwendungen, die eine hohe

Verläßlichkeit verlangen (z.B. sicherheitskritische Systeme, von denen Menschenleben abhängen), wird deutlich wachsen. Es wird immer wichtiger, daß die in solchen Systemen eingesetzte Software frei von Entwurfsfehlern ist. Die Entwickler solcher Software werden immer mehr für ihre Programme – und vor allem für ihre Fehler – zur Verantwortung gezogen.

Mittel- bis langfristig wird die Software-Entwicklung auf eine ingenieurmäßige Basis gestellt werden, denn wirklich zuverlässige Software ist sowohl möglich als auch nötig. In der künftigen Software-Entwicklungspraxis wird eine andere Denkweise herrschen – sowohl technisch als auch hinsichtlich der Verantwortung für die Richtigkeit (Entwurfsfehlerfreiheit) der Programme. Es wird vom Software-Entwickler erwartet werden, daß er das Verhalten seines Programms analytisch und systematisch "berechnet", also vollständig nachweist. Er wird diese Erwartung erfüllen. Eine Vorgehensweise, die das ermöglicht, wurde in diesem Buch vorgestellt.

Software-Entwickler von heute sollten möglichst früh anfangen, sich auf die Software-Entwicklungswelt von morgen vorzubereiten. Sie wird zwangsläufig ziemlich anders als die Software-Entwicklungswelt von heute sein. Denn Software, von der Menschenleben abhängen können, auf die herkömmliche Weise zu erstellen, ist nichts anderes als High-Tech russisches Roulett.

Anhang A. Logische (Boolesche) Algebra

Im folgenden sind x, y und z Variablen oder Ausdrücke, deren Werte falsch oder wahr sind.

A.1 Definition der Booleschen Funktionen

Die Booleschen Funktionen **und, oder, nicht** und Implikation (⟹) sind in den folgenden Wahrheitstabellen definiert:

x	y	x **und** y	x **oder** y	x ⟹ y
falsch	falsch	falsch	falsch	wahr
falsch	wahr	falsch	wahr	wahr
wahr	falsch	falsch	wahr	falsch
wahr	wahr	wahr	wahr	wahr

x	**nicht** x
falsch	wahr
wahr	falsch

A.2 Priorität der Funktionen bei der Auswertung von Ausdrücken

In Ausdrücken werden die verschiedenen Funktionen üblicherweise in der folgenden Reihenfolge angewendet, wenn eine andere nicht durch Klammern explizit angegeben ist:

↑ (potenzieren)
+, - (Vorzeichen)
*, / (Multiplikation, Division)
+, - (Addition, Subtraktion)
<, >, =, ≤, ≥, ≠ (relationale Funktionen)
nicht (auch ¬ geschrieben)
und (auch ∧ geschrieben)
oder (auch ∨ geschrieben)
⟹ (logische Implikation)

A.3 Grundlegende Eigenschaften der Booleschen Funktionen

Die folgenden Aussagen (meist Gleichungen) drücken besonders wichtige Eigenschaften der Booleschen Funktionen aus. Sie werden häufig bei der Manipulation und Vereinfachung von logischen algebraischen Ausdrücken angewendet; verwende deshalb diese Liste als Nachschlagwerk.

Es wird dem Leser als Übung empfohlen, von den Definitionen oben ausgehend die unten aufgeführten Gleichungen und Aussagen zu verifizieren.

Die Funktionen **und** und **oder** sind kommutativ:

1. (x **und** y) = (y **und** x)
2. (x **oder** y) = (y **oder** x)

Die Funktionen **und** und **oder** sind assoziativ:

3. (x **und** (y **und** z)) = ((x **und** y) **und** z)
4. (x **oder** (y **oder** z)) = ((x **oder** y) **oder** z)

Die Funktionen **und** und **oder** sind distributiv:

5. (x **und** (y **oder** z)) = ((x **und** y) **oder** (x **und** z))
6. (x **oder** (y **und** z)) = ((x **oder** y) **und** (x **oder** z))

Einfache Identitäten:

7. (x **und nicht** x) = falsch
8. (x **und** falsch) = falsch
9. (x **und** x) = x
10. (x **und** wahr) = x

11. (x **oder** falsch) = x
12. (x **oder** x) = x
13. (x **oder** wahr) = wahr
14. (x **oder nicht** x) = wahr

15. (**nicht** (**nicht** x)) = x

Weitere Identitäten:

16. (x **oder** (x **und** y)) = x
17. (x **oder** (**nicht** x **und** y)) = (x **oder** y)
18. ((x **oder** y) **und** (x **oder** z)) = (x **oder** y **und** z)

Die Negierung von **und**- und **oder**-Verknüpfungen:

19. (**nicht** (x **und** y)) = ((**nicht** x) **oder** (**nicht** y))
20. (**nicht** (x **oder** y)) = ((**nicht** x) **und** (**nicht** y))

Alternative Ausdrücke für die Implikation:

21. (x $\Longrightarrow$ y) = (**nicht** (x **und nicht** y))
22. (x $\Longrightarrow$ y) = ((**nicht** x) **oder** y)
23. (x $\Longrightarrow$ y) = ((**nicht** y) $\Longrightarrow$ (**nicht** x))
24. (z **und** (x $\Longrightarrow$ y)) = (z **und** ((z **und** x) $\Longrightarrow$ y))

Alternativer Ausdruck für die Gleichheit:

25. (x = y) = ((x **und** y) **oder** (**nicht** x **und nicht** y))

Beachte, daß (x **und** y) = (x **und** z), vorausgesetzt nur, daß y=z, wenn x wahr ist. Mit anderen Worten gilt die Aussage, wenn x falsch ist, egal ob y=z oder nicht. Symbolisch

26. [x $\Longrightarrow$ (y=z)] $\Longrightarrow$ [(x **und** y) = (x **und** z)]

A.4 Weiterführende Übungen

1. B, C und D seien Boolesche Variablen oder Ausdrücke. Die neue Funktion F wird wie folgt definiert:

F = C, falls B=wahr,
= D, falls B=falsch

Schreibe für F einen gleichwertigen Ausdruck. Verwende dabei außer B, C und D nur die Booleschen Funktionen **und**, **oder** und **nicht**. Zeige, daß der neue Ausdruck der oben stehenden Definition von F genügt.

2. Stärkung und Schwächung von Bedingungen: Zeige, daß die folgenden Aussagen für alle Werte von x und y gelten.

x und y $\Longrightarrow$ **x**
x $\Longrightarrow$ **x oder y**

3. Zeige, daß die folgenden Ausdrücke gleich sind:

x und (y oder z)
x und (x und y oder z)
x und (y oder x und z)
x und (x und y oder x und z)
x und y oder x und z

4. Zeige, daß die folgende Aussage wahr ist: Falls

a $\Longrightarrow$ **x und**
b $\Longrightarrow$ **y**

dann gilt, daß

a und b $\Longrightarrow$ **x und y** sowie
a oder b $\Longrightarrow$ **x oder y**

5. Vereinfache bzw. expandiere die folgenden Ausdrücke:

1. **x und (y** $\Longrightarrow$ **z)**
2. **x oder (y** $\Longrightarrow$ **z)**
3. **(x und y)** $\Longrightarrow$ **z**
4. **(x oder y)** $\Longrightarrow$ **z**

5. **-a>0 und a<0 oder a>0 und nicht a<0**
6. **-a≥0 und a<0 oder a≥0 und nicht a<0**
7. **-a<0 und a<0 oder a<0 und nicht a<0**
8. **-a≤0 und a<0 oder a≤0 und nicht a<0**

A.5 Die und- und oder-Reihen

Die **und**- und **oder**-Reihen sind wie folgt definiert:

$$\mathbf{und}_{i=1}^{n}\ A(i) \equiv A(1)\ \mathbf{und}\ A(2)\ \ldots\ \mathbf{und}\ A(n)$$

$$\mathbf{oder}_{i=1}^{n}\ A(i) \equiv A(1)\ \mathbf{oder}\ A(2)\ \ldots\ \mathbf{oder}\ A(n)$$

wobei A(i) ein beliebiger Ausdruck ist, in dem die Variable i vorkommen darf. Die Variable i ist keine Programmvariable, sondern eine *Laufvariable* der Reihe; außerhalb der Reihe hat sie keine Bedeutung.

Der Wert einer leeren **und**-Reihe ($n<1$) ist definitionsgemäß wahr; der Wert einer leeren **oder**-Reihe falsch. (Vgl. die Σ- und Π-Schreibweise für Summen und Produkte.)

Offensichtlich gilt

$$[\mathbf{und}_{i=1}^{n}\ A(i)] = [A(n)\ \mathbf{und}_{i=1}^{n-1}\ A(i)]$$

falls $n\geq 1$. Man kann einen Term jedoch nur dann aus einer Reihe herausnehmen, wenn die ursprüngliche Reihe mindestens einen Term enthält. Im allgemeinen (d.h., falls die ursprüngliche Reihe auch leer sein darf) gilt

$$[\mathbf{und}_{i=1}^{n}\ A(i)] = [n<1\ \mathbf{oder}\ n\geq 1\ \mathbf{und}\ A(n)\ \mathbf{und}_{i=1}^{n-1}\ A(i)]$$

Für die **oder**-Reihe ist die entsprechende Formel

$$[\mathbf{oder}_{i=1}^{n}\ A(i)] = [n\geq 1\ \mathbf{und}\ (A(n)\ \mathbf{oder}_{i=1}^{n-1}\ A(i))]$$

Übung:

1. Verifiziere die oben aufgeführten Identitäten. Hinweis:

$$[\mathbf{und}_{i=1}^{n}\ A(i)] = [(n<1\ \mathbf{oder}\ n\geq 1)\ \mathbf{und}_{i=1}^{n}\ A(i)]\ \blacksquare$$

Anhang B. Lösungen zu den Übungsaufgaben

Die unten stehenden Aufgabennummern setzen sich aus der Nummer des Abschnitts, in dem sich die jeweilige Übung befindet, und der Nummer der Aufgabe innerhalb der Übung zusammen.

4.1.1 (1). {0≤i} i:=i+1 {1≤i}

4.1.1 (2). {sum+z=x+y+z} sum:=sum+z {sum=x+y+z}. Typischerweise kann die Vorbedingung auf {sum=x+y} vereinfacht werden. Handelt es sich hier bei der Addition in der Zuweisung um Gleitkomma-Arithmetik, muß man jedoch darauf achten, daß (1) die bei der Vereinfachung des Ausdrucks unterstellte Assoziativität (...+z)-z = ...+(z-z) nicht immer gilt und daß sich (2) die Addition in der Nachbedingung eventuell auf die in der Mathematik definierte Addition bezieht, die nicht die gleiche Operation wie die Gleitkomma-Addition ist. Im zuletzt genannten Falle sollte man zwei verschiedene Symbole für die unterschiedlichen Additionsoperationen schreiben.

4.1.1 (3). {w*y - 2*w² < z} x:=5-z {w*y - 2*w² < z}. Die Variable x kommt in der Nachbedingung nicht vor. Gemäß der Beweisregel Z1 ist in diesem Falle die Nachbedingung auch die Vorbedingung.

4.2 (1). {x≠0} **if** x<0 **then** y:=-x **else** y:=x **endif** {y>0}. Wende die Beweisregel IF2 an. V1 = {-x>0} = {x<0}, V2 = {x>0}, B = {x<0}, **nicht** B = {x≥0}.

4.2 (2). {wahr} **if** x<0 **then** y:=-x **else** y:=x **endif** {y≥0}. Die Vorbedingung ist die logische Konstante wahr. Die Nachbedingung wird nach der Ausführung der if-Anweisung immer erfüllt sein.

4.2 (3). {falsch} **if** x<0 **then** y:=-x **else** y:=x **endif** {y<0}. Die Vorbedingung ist die logische Konstante falsch. Die Nachbedingung wird nach der Ausführung der if-Anweisung nie erfüllt sein.

4.2 (4). {x=0} **if** x<0 **then** y:=-x **else** y:=x **endif** {y≤0}

4.2 (5). {3≤|x|≤4} **if** x<0 **then** y:=-x **else** y:=x **endif** {2≤y≤4} gilt. Wende die Beweisregeln IF1 und Z2 an. Alternativ ergibt sich aus der Anwendung der Beweisregel IF2 die Vorbedingung {2≤|x|≤4}. Gemäß der Beweisregel B1 ist die stärkere vorgegebene Bedingung auch eine Vorbedingung.

4.3 (1). Wir wenden die Beweisregeln F1 und Z1 an. Dabei fangen wir mit der Nachbedingung an und arbeiten *rückwärts* bis zum Anfang der Folge von Anweisungen (lies deshalb von unten nach oben).

{0≤N} =

{(0>M-N **oder** 0≤M-N) **und** 0≤N}

i:=0

{(0>M-N **oder** 0≤M-N **und**$_{a=0}^{i-1}$ D(0+a)=K(a)) **und** 0≤i≤N}

j:=0

{**und**$_{k=0}^{j-1}$ (**nicht** **und**$_{a=0}^{N-1}$ D(k+a)=K(a))

und (j>M-N **oder** j≤M-N **und**$_{a=0}^{i-1}$ D(j+a)=K(a))

und 0≤j **und** 0≤i≤N}

4.5 (1). Zu beweisen sind die zwei Aussagen (siehe Abschnitt 4.5)

{I **und** ia≤na **und** (ib>nb **oder** A(ia)≤B(ib))}
C(ic):=A(ia)
ia:=ia+1
{I$_{ic+1}^{ic}$}

und

{I **und** ib≤nb **und** (ia>na **oder** B(ib)<A(ia))}
C(ic):=B(ib)
ib:=ib+1
{I$_{ic+1}^{ic}$}

Die letzte Aussage oben wird gemäß der Beweisregel B1 gelten, falls

{I **und** ib≤nb **und** (ia>na **oder** B(ib)≤A(ia))}
C(ic):=B(ib)
ib:=ib+1
$\{I^{ic}_{ic+1}\}$

Jetzt sind die zwei zu beweisenden Aussagen (die erste und die letzte oben) völlig symmetrisch in A bzw. a und B bzw. b. (Die Schleifeninvariante I ist symmetrisch.)

4.5 (2). Zu beweisen sind (siehe 4.5 (1) oben)

{I **und** ia≤na **und** (ib>nb **oder** A(ia)≤B(ib))}
C(ic):=A(ia)
ia:=ia+1
$\{In^{ic}_{ic+1}\}$

und

{I **und** ib≤nb **und** (ia>na **oder** B(ib)≤A(ia))}
C(ic):=B(ib)
ib:=ib+1
$\{In^{ic}_{ic+1}\}$

für jeweils die sieben Nachbedingungen

$I1^{ic}_{ic+1}$: 1≤ia≤na+1

$I2^{ic}_{ic+1}$: 1≤ib≤nb+1

$I3^{ic}_{ic+1}$: ic=(ia-1)+(ib-1)

$I4^{ic}_{ic+1}$: (ic≤0 **oder** ia>na **oder** C(ic)≤A(ia))

$I5^{ic}_{ic+1}$: (ic≤0 **oder** ib>nb **oder** C(ic)≤B(ib))

$I6^{ic}_{ic+1}$: $\mathbf{und}_{i=1}^{ic-1}$ C(i)≤C(i+1)

$I7^{ic}_{ic+1}$: $\mathbf{und}_{i=1}^{na-1}$ A(i)≤A(i+1) $\mathbf{und}_{i=1}^{nb-1}$ B(i)≤B(i+1)

4.5 (3). Wegen der Symmetrie zwischen A/a und B/b brauchen wir nur die sieben Aussagen

{I **und** ia≤na **und** (ib>nb **oder** A(ia)≤B(ib))}
C(ic):=A(ia)
ia:=ia+1
$\{In_{ic+1}^{ic}\}$

für n=1, 2, ... 7 zu beweisen. Für jede Nachbedingung ermitteln wir durch Anwendung der Beweisregeln Z1 und F1 eine Vorbedingung bezüglich der zwei Zuweisungen. In den trivialen Fällen 2 und 7 ist die jeweilige Vorbedingung gleich der Nachbedingung und sie folgt direkt aus der Schleifeninvariante I. Für die anderen Fälle ermitteln wir die Vorbedingungen:

1: {0≤ia≤na} C(ic):=A(ia); ia:=ia+1 {1≤ia≤na+1}

3: {ic=ia+(ib-1)} C(ic):=A(ia); ia:=ia+1 {ic=(ia-1)+(ib-1)}

4: {ic≤0 **oder** ia+1>na **oder** A(ia)≤A(ia+1)}
C(ic):=A(ia); ia:=ia+1
{ic≤0 **oder** ia>na **oder** C(ic)≤A(ia)}

5: {ic≤0 **oder** ib>nb **oder** A(ia)≤B(ib)}
C(ic):=A(ia); ia:=ia+1
{ic≤0 **oder** ib>nb **oder** C(ic)≤B(ib)}

6: $\{ic\leq 1 \text{ oder } C(ic-1)\leq A(ia) \text{ und}_{i=1}^{ic-2} C(i)\leq C(i+1)\}$

C(ic):=A(ia); ia:=ia+1 $\{\text{und}_{i=1}^{ic-1} C(i)\leq C(i+1)\}$

Wir müssen noch zeigen, daß diese ermittelten Vorbedingungen aus der vorgegebenen Vorbedingung folgen (vgl. die Beweisregeln B1 und Z2).

zu 1: Die vorgegebene Vorbedingung enthält (I **und** ia≤na), woraus folgt, daß 1≤ia≤na, woraus wiederum die ermittelte Vorbedingung (0≤ia≤na) folgt.

zu 3: Die ermittelte Vorbedingung ist äquivalent zu (ic-1)=(ia-1)+(ib-1), einem Term in der Schleifeninvariante I und dadurch in der vorgegebenen Vorbedingung.

zu 4: Aus der Schleifeninvariante I folgt, daß ic≥1 (siehe I1, I2 und I3). Der Term ic≤0 in der ermittelten Vorbedingung wird also immer falsch sein. Der Rest der ermittelten

Vorbedingung ist äquivalent zu [ia≤na-1 ⟹ A(ia)≤A(ia+1)]. Diese letzte Bedingung folgt aus I7 (in Kombination mit I1). Es gilt also, daß aus der vorgegebenen Vorbedingung die ermittelte Vorbedingung folgt. Formal:

{I **und** ia≤na **und** (ib>nb **oder** A(ia)≤B(ib))}

⟹ I

⟹ I1 **und** I7

⟹ 1≤ia **und** $\mathbf{und}_{i=1}^{na-1}$ A(i)≤A(i+1)

⟹ 1≤ia **und** (1≤ia≤na-1 ⟹ A(ia)≤A(ia+1))
[Siehe Anhang A, Abschnitt A.3, Identität 24]

= 1≤ia **und** (ia≤na-1 ⟹ A(ia)≤A(ia+1))

⟹ (ia≤na-1 ⟹ A(ia)≤A(ia+1))

= ia>na-1 **oder** A(ia)≤A(ia+1)

= ia+1>na **oder** A(ia)≤A(ia+1)

⟹ ic≤0 **oder** ia+1>na **oder** A(ia)≤A(ia+1)

zu 5: Der erste Term wird immer falsch sein (siehe zu 4 oben). Der Rest der ermittelten Vorbedingung ist ein Teil der vorgegebenen Vorbedingung und folgt deshalb daraus.

zu 6: Die ermittelte Vorbedingung folgt aus (I **und** ia≤na), einem Teil der vorgegebenen Vorbedingung. Insbesondere folgt die ermittelte Vorbedingung aus (I4 **und** ia≤na **und** I6).

5.2.10 (1). Um eine Vorbedingung bezüglich der Vertauschanweisung x:=:y zu ermitteln, ersetze gleichzeitig überall in der Nachbedingung x durch y und y durch x. Alternativ kann man die Vertauschanweisung durch die äquivalente Folge von Zuweisungen

```
zwvar:=y; y:=x; x:=zwvar
```

ersetzen und die Korrektheit des resultierenden Programms beweisen. Dabei muß darauf geachtet werden, daß diese Verwendung der Zwischenvariablen *zwvar* die sonstigen Wirkungen des Programms nicht stört. Insbesondere darf die Varia-

ble *zwvar* in der Nachbedingung der Vertauschanweisung nicht vorkommen.

5.2.10 (2). Zu beweisen ist die folgende Aussage über den ganzen Programmteil:

```
{il≤ir}
gl:=il; gr:=gl; k:=ir
while gr<k do
    if X(gr+1)<X(gl)
    then X(gl):=:X(gr+1)
         gl:=gl+1
         gr:=gr+1
    else if X(gr+1)=X(gl)
    then gr:=gr+1
    else [Bemerkung: X(gr+1)>X(gl)]
         X(k):=:X(gr+1)
         k:=k-1
    endif
    endif
endwhile
{il≤gl≤gr≤ir und_{i=il}^{gl-1} X(i)<X(gl)
und_{i=gl}^{gr} X(i)=X(gl) und_{i=gr+1}^{ir} X(i)>X(gl)}
```

Dabei ist die Schleifeninvariante I aus der Konstruktion vorgegeben:

$$il \le gl \le gr \le k \le ir \ \textbf{und}_{i=il}^{gl-1}\ X(i)<X(gl)$$

$$\textbf{und}_{i=gl}^{gr}\ X(i)=X(gl)\ \textbf{und}_{i=k+1}^{ir}\ X(i)>X(gl)$$

Im Abschnitt 5.2.9 wurde bereits bewiesen, daß die Schleife terminiert. Deshalb zeigen wir hier nur die partielle Korrektheit.

Durch die Anwendung der Beweisregel W2 zerlegen wir die zu beweisende Korrektheitsaussage oben in die drei folgenden Aussagen, die zu beweisen sind:

```
{il≤ir} gl:=il; gr:=gl; k:=ir {I}                    [1]

{I und gr<k} if ... endif {I}                        [2]
```

{I **und nicht** gr<k} [3]

$\Longrightarrow$

$\{il \le gl \le gr \le ir \text{ und}_{i=il}^{gl-1} X(i)<X(gl)$

$\text{und}_{i=gl}^{gr} X(i)=X(gl) \text{ und}_{i=gr+1}^{ir} X(i)>X(gl)\}$

Aussage 1 oben wurde bereits im Abschnitt 5.2.8 effektiv bewiesen; dort wurde die Vorbedingung mit Hilfe der Beweisregeln Z1 und F1 ermittelt.

Aussage 3 oben kann auf einfache Weise gezeigt werden. {I **und nicht** gr<k} = {I **und** gr≥k} $\Longrightarrow$ gr=k. Ersetze k durch gr in der Schleifeninvariante; das Ergebnis ist die Nachbedingung.

Aussage 2 zerlegen wir durch Anwendung der Beweisregel IF1 in drei zu beweisende Aussagen über die einzelnen Zweige der if-Anweisung. In voll ausgeschriebener Form sind diese Aussagen wie folgt:

$\{il \le gl \le gr < k \le ir \text{ und}_{i=il}^{gl-1} X(i)<X(gl)$ [2.1]

$\text{und}_{i=gl}^{gr} X(i)=X(gl) \text{ und}_{i=k+1}^{ir} X(i)>X(gl)$

und $X(gr+1)<X(gl)\}$

$X(gl):=:X(gr+1);\ gl:=gl+1;\ gr:=gr+1$

$\{il \le gl \le gr \le k \le ir \text{ und}_{i=il}^{gl-1} X(i)<X(gl)$

$\text{und}_{i=gl}^{gr} X(i)=X(gl) \text{ und}_{i=k+1}^{ir} X(i)>X(gl)\}$

und

$\{il \le gl \le gr < k \le ir \text{ und}_{i=il}^{gl-1} X(i)<X(gl)$ [2.2]

$\text{und}_{i=gl}^{gr} X(i)=X(gl) \text{ und}_{i=k+1}^{ir} X(i)>X(gl)$

und $X(gr+1)=X(gl)\}$

$gr:=gr+1$

$\{il \le gl \le gr \le k \le ir \text{ und}_{i=il}^{gl-1} X(i)<X(gl)$

$\text{und}_{i=gl}^{gr} X(i)=X(gl) \text{ und}_{i=k+1}^{ir} X(i)>X(gl)\}$

und

$$\{il \le gl \le gr < k \le ir \ \textbf{und}_{i=il}^{gl-1}\ X(i) < X(gl) \qquad [2.3]$$

$$\textbf{und}_{i=gl}^{gr}\ X(i) = X(gl)\ \textbf{und}_{i=k+1}^{ir}\ X(i) > X(gl)$$

$$\textbf{und}\ X(gr+1) > X(gl)\}$$

$$X(k) :=: X(gr+1);\ k := k-1$$

$$\{il \le gl \le gr \le k \le ir \ \textbf{und}_{i=il}^{gl-1}\ X(i) < X(gl)$$

$$\textbf{und}_{i=gl}^{gr}\ X(i) = X(gl)\ \textbf{und}_{i=k+1}^{ir}\ X(i) > X(gl)\}$$

Jede Aussage wird durch die Anwendung der Beweisregeln Z2 und ggf. F1 und Z1 bewiesen. Zuerst wird eine Vorbedingung der Schleifeninvariante (als Nachbedingung) bezüglich der Folge von Zuweisungen ermittelt. Danach wird gezeigt, daß die ermittelte Vorbedingung aus der vorgegebenen Vorbedingung folgt.

Die Aussage 2.2 oben bezieht sich auf eine einzelne Zuweisung zu einer nicht indizierten Variablen. Ihr Beweis ist entsprechend kurz und einfach. Die Aussage 2.1 bezieht sich auf die längste Folge von Anweisungen und ihr Beweis ist der schwierigste. Ferner kann im Beweis der Aussage 2.1 durch eine geschickte, aber einfache Umformulierung der Nachbedingung und eine geeignete Anwendung der Beweisregel B1 die algebraische Manipulation deutlich vereinfacht werden. Deshalb schauen wir den Beweis der Aussage 2.1 detailliert an.

Im ersten Schritt ersetzen wir in der Nachbedingung gr und gl durch gr+1 bzw. gl+1. Dadurch werden Bezüge auf X(gl+1) entstehen. Danach wollen wir u.a. Bezüge auf X(gr+1) ersetzen. Je nachdem, ob gl+1=gr+1 oder nicht, sollen solche Bezüge ersetzt werden oder nicht. Diese Abhängigkeit von der Beziehung zwischen Variablenwerten würde die algebraische Manipulation erschweren. Aus der Nachbedingung folgt, daß X(gl)=X(gr) (siehe die **und**-Reihe für den =Bereich). Wenn wir die Nachbedingung entsprechend umschreiben, werden sich die fraglichen Teilausdrücke auf X(gr+1) beziehen, die unbedingt zu ersetzen sind. Der neue Ausdruck für die Nachbedingung ist

$$\{il \le gl \le gr \le k \le ir \text{ und}_{i=il}^{gl-1} X(i)<X(gr)$$

$$\text{und}_{i=gl}^{gr} X(i)=X(gr) \text{ und}_{i=k+1}^{ir} X(i)>X(gr)\}$$

Die Vorbedingung bezüglich der Folge der letzten zwei Zuweisungen (gl:=gl+1; gr:=gr+1) ist

$$\{il \le gl+1 \le gr+1 \le k \le ir \text{ und}_{i=il}^{gl} X(i)<X(gr+1)$$

$$\text{und}_{i=gl+1}^{gr+1} X(i)=X(gr+1) \text{ und}_{i=k+1}^{ir} X(i)>X(gr+1)\}$$

Der erste Term ist äquivalent zu (il≤gl+1 **und** gl≤gr<k≤ir). Die vorgegebene Vorbedingung stellt sicher, daß die stärkere Einschränkung il≤gl gilt. Die Vertauschanweisung beeinflußt diese Variablen nicht, so daß es auch danach gelten wird, daß il≤gl. Gemäß der Beweisregel B1 ist die entsprechend gestärkte Bedingung

$$\{il \le gl \le gr<k \le ir \text{ und}_{i=il}^{gl} X(i)<X(gr+1)$$

$$\text{und}_{i=gl+1}^{gr+1} X(i)=X(gr+1) \text{ und}_{i=k+1}^{ir} X(i)>X(gr+1)\}$$

eine Vorbedingung bezüglich der Folge von Zuweisungen (gl:=gl+1; gr:=gr+1).

Jetzt müssen wir X(gl) und X(gr+1) ersetzen. Wir wissen, daß gl≤gr<gr+1. Einige der X(i) können Bezüge auf X(gl) oder X(gr+1) sein; wir müssen solche Bezüge von den anderen trennen. Wir müssen einen Term (i=gl) aus der ersten **und**-Reihe herausnehmen. Ebenfalls müssen wir einen Term (i=gr+1) aus der zweiten **und**-Reihe herausnehmen. Dieser Term ist eine Tautologie, also wahr, und kann deshalb weggelassen werden (siehe Anhang A, Abschnitt A.3, Identität 10).

Dadurch wird unsere Vorbedingung:

$$\{il \le gl \le gr<k \le ir \text{ und}_{i=il}^{gl-1} X(i)<X(gr+1) \text{ und } X(gl)<X(gr+1)$$

$$\text{und}_{i=gl+1}^{gr} X(i)=X(gr+1) \text{ und}_{i=k+1}^{ir} X(i)>X(gr+1)\}$$

Jetzt bezieht sich kein X(i) auf X(gl) oder auf X(gr+1). In der ersten **und**-Reihe ist i<gl<gr+1. In der zweiten **und**-Reihe gilt gl<i<gr+1. In der dritten **und**-Reihe gilt gl<gr+1<k+1≤i, also gl<gr+1<i (siehe den ersten Term in der Vorbedingung oben).

Jetzt können wir X(gl) durch X(gr+1) und X(gr+1) durch X(gl) gleichzeitig ersetzen, um eine Vorbedingung bezüglich des ganzen Zweigs der if-Anweisung zu ermitteln:

$$\{il \le gl \le gr < k \le ir \ \textbf{und}_{i=il}^{gl-1} \ X(i)<X(gl) \ \textbf{und} \ X(gr+1)<X(gl)$$

$$\textbf{und}_{i=gl+1}^{gr} \ X(i)=X(gl) \ \textbf{und}_{i=k+1}^{ir} \ X(i)>X(gl)\}$$

Diese ermittelte Vorbedingung ist äquivalent zu der vorgegebenen Vorbedingung. Es fehlt der Term X(i)=X(gl) für i=gl in der zweiten **und**-Reihe; dieser Term ist auf jeden Fall wahr und kann hinzu geschrieben werden. Sonst sind bis auf die Reihenfolge der Terme die ermittelte und die vorgegebene Vorbedingungen gleich.

Der Beweis der Aussage 2.3 oben ist strukturell ähnlich, aber deutlich einfacher.

5.3.7 (1). Wenn N=0, dann ist der Suchschlüssel (die Zeichenkette K) leer. In diesem Falle "findet" das Programm die leere Zeichenkette gleich am Anfang der Zeichenkette D (auch wenn D die leere Zeichenkette ist) und endet sofort mit j=0. Wenn M=0, dann ist die Zeichenkette D, worin K gesucht wird, leer. Eine nicht leere Zeichenkette K kann nicht darin vorkommen. In diesem Falle endet das Programm (ohne Ausführung des Schleifenkerns) mit j=0>M-N=-N, also mit dem Ergebnis "kommt nicht vor". Siehe die Nachbedingung im Abschnitt 5.3.2.

5.3.7 (2). Der Fall M<0 ist wie M=0 (siehe Aufgabe 5.3.7 (1)) zu interpretieren, bis auf den Unterschied, daß auch eine leere Zeichenkette K darin "nicht vorhanden" ist. Das Programm endet ohne Ausführung des Schleifenkerns mit j=0. Bezogen auf das Eingabefeld D haben jedoch negative Werte von M keinen eigentlichen Sinn; deshalb wurden sie in der vorgegebenen Vorbedingung ausgeschlossen.

5.3.7 (3). Hinweis: Unterscheide zwischen den zwei Fällen a≤b und a>b.

5.3.7 (4). Die zu beweisende Korrektheitsaussage über das gesamte Unterprogramm ist

{MεZ und 0≤M und NεZ und 0≤N}

```
i:=0; j:=0
while i<N und j≤M-N do
   if D(j+i)=K(i) then i:=i+1 else j:=j+1; i:=0 endif
endwhile
```

{jεZ und $0 \le j \le \max(0, M-N+1)$

und$_{k=0}^{j-1}$ (oder$_{a=0}^{N-1}$ $D(k+a) \neq K(a)$)

und (j>M-N oder j≤M-N und$_{a=0}^{N-1}$ $D(j+a)=K(a)$)}

Durch Anwendung der Beweisregel W2 zerlegen wir die Korrektheitsaussage oben in die folgenden drei Korrektheitsaussagen über einzelne Programmteile, wobei die Schleifeninvariante I dem im Abschnitt 5.3.3 beschriebenen Konstruktionsschritt zu entnehmen ist:

{MεZ und 0≤M und NεZ und 0≤N} i:=0; j:=0 {I} [1]

{I und i<N und j≤M-N} [2]
if D(j+i)=K(i) then i:=i+1 else j:=j+1; i:=0 endif {I}

{I und nicht (i<N und j≤M-N)} [3]
⟹
{jεZ und $0 \le j \le \max(0, M-N+1)$

und$_{k=0}^{j-1}$ (oder$_{a=0}^{N-1}$ $D(k+a) \neq K(a)$)

und (j>M-N oder j≤M-N und$_{a=0}^{N-1}$ $D(j+a)=K(a)$)}

Die Aussage 1 oben wurde im Abschnitt 5.3.7 bewiesen. Der Beweis der Aussage 3 ist nur eine Übung in der Manipulation von logischen Ausdrücken. Die Aussage 2 zerlegen wir (siehe Beweisregel IF1) in Aussagen über die einzelnen Zweige der if-Anweisung:

{I und i<N und j≤M-N und D(j+i)=K(i)} [2.1]
i:=i+1 {I}

{I und i<N und j≤M-N und D(j+i)≠K(i)} [2.2]
j:=j+1; i:=0 {I}

Die Aussagen 2.1 und 2.2 können auf bekannte Weise durch Anwendung der Beweisregeln Z1, Z2 und F1 bewiesen werden. An einer Stelle im Beweis der Aussage 2.2 muß die Tatsache, daß

$$0 \le i < N \text{ und } D(j+i) \neq K(i) \Longrightarrow \text{oder}_{a=0}^{N-1}\ D(j+a) \neq K(a)$$

verwendet werden. (Vgl. Anhang A, Abschnitt A.4, Aufgabe 2, 2. Teil.)

Ferner muß gezeigt werden, daß die Schleife terminiert. Es ist ersichtlich, daß bei jeder Ausführung des Schleifenkerns entweder i oder j erhöht wird und daß j nie verringert wird. Da sowohl i als auch j obere Grenzen haben (siehe die while-Bedingung), muß die Schleife terminieren. Formaler, definiere als Schleifenvariante (j*N+i). Der Wert dieses Ausdrucks wird im then-Zweig der if-Anweisung um 1 erhöht. Im else-Zweig wird dieser Wert um N erhöht und um i (maximal N-1) verringert; netto wird der Wert der Schleifenvariante also um mindestens 1 erhöht. Der maximale Wert, der aus der while-Bedingung folgt, wird deshalb nach endlich vielen Ausführungen des Schleifenkerns erreicht. Dann endet die Schleife.

5.4.2 (1). Der folgende Ausdruck ist zur Nachbedingung hinzuzufügen:

```
1≤nz≤n+1                                  [Wertebereiche von nz,]

und {nz=n+1 und np=1                      [np (kein Name mehr)]

    oder nz≤n und 1≤np<ep≤Länge(F(nz))+1}
                                          [np, ep (Name gefunden)]
und ...
```

5.4.2 (2). Leer(z1,p1,z2,p2): Der Bereich von der Stelle (Zeile z1, Position p1) bis zur Stelle vor (z2,p2) enthält nur Leerzeichen und Zeilenbrüche, wenn

$z1=z2$ **und** $\underset{p=p1}{\overset{p2-1}{}}$ mid(F(z1),p,1)=Leerzeichen
[nur eine Zeile]

oder

$z1<z2$ [mehr als eine Zeile]

und $\underset{p=p1}{\overset{\text{Länge(F(z1))}}{}}$ mid(F(z1),p,1)=Leerzeichen
[Zeile z1]

und $\underset{z=z1+1}{\overset{z2-1}{}}$

und $\underset{p=1}{\overset{\text{Länge(F(z))}}{}}$ mid(F(z),p,1)=Leerzeichen
[Zwischenzeilen (ggf. keine)]

und $\underset{p=1}{\overset{p2-1}{}}$ mid(F(z2),p,1)=Leerzeichen [Zeile z2]

Diese Formel gilt nur dann, wenn die Stelle (z1,p1) vor der Stelle (z2,p2) liegt oder wenn sich beide Stellenangaben auf die gleiche Stelle beziehen (in welchem Falle der angesprochene Bereich leer ist). D.h., die Formel oben gilt nur dann, wenn ($z1=z2$ **und** $p1 \leq p2$ **oder** $z1<z2$). Die Werte dieser Variablen müssen natürlich zulässige Zeilen- bzw. Positionsnummern sein.

Namenlage(z,p1,p2): Ein Name fängt an der Stelle (z,p1) an und endet an der Stelle vor (z,p2), wenn (1) die Zeile z innerhalb des Felds steht und (2) ein Name an der Stelle (z,p1) anfängt und (3) jede Stelle von (z,p1) bis (z,p2-1) ein Zeichen außer dem Leerzeichen enthält und (4) die Stelle (z,p2) nicht mehr zum Namen gehört. Ein Name fängt an der Stelle (z,p1) an, wenn die Position p1 innerhalb der Zeile steht und ein Zeichen außer dem Leerzeichen enthält und entweder die Stelle (z,p1) die erste Stelle in der Zeile ist oder die davor liegende Stelle ein Leerzeichen enthält. Die Stelle (z,p2) gehört nicht mehr zum Namen, wenn sie entweder gar nicht innerhalb der Zeile liegt oder ein Leerzeichen enthält:

```
1≤z≤n und 1≤p1<p2≤Länge(F(z))+1
                                    [Wertebereiche von z, p1, p2]
und (p1=1                                    [Anfang der Zeile]
    oder 1<p1 und mid(F(z),p1-1,1)=Leerzeichen)
                          [Leerzeichen vor der Stelle (z,p1)]
und (p=p1 bis p2-1) mid(F(z),p,1)≠Leerzeichen
                          [kein Leerzeichen im Namen]
und (p2=Länge(F(z))+1                        [Ende der Zeile]
    oder p2≤Länge(F(z)) und mid(F(z),p2,1)=Leerzeichen)
                          [Leerzeichen an der Stelle (z,p2)]
```

Dieser Ausdruck kann etwas vereinfacht werden: nach dem jeweiligen **oder** können "1<p1 **und**" und "p2≤Länge(F(z)) **und**" weggelassen werden. Wenn an anderer Stelle dafür gesorgt wird, daß nur zulässige Zeilen- und Positionsnummern für z bzw. p1 erzeugt werden, können auch "1≤z" und "1≤p1" entfallen.

5.4.2 (3). Achte darauf, daß der Begriff "innerhalb" hier nicht präzise ist. Nicht selten sind Aufgabenbeschreibungen, die in einer natürlichen Sprache formuliert sind, zweideutig oder irreführend. Oft sieht man erst dann die echte Bedeutung einer Aussage, wenn man versucht, sie in eine andere Sprache zu übersetzen. Die strengste Prüfung entsteht dann, wenn die Zielsprache nur präzise, eindeutige Aussagen zuläßt.

Im vorliegenden Fall kann nur gemeint sein, daß die Stelle (az,ap) nicht in einem Namen liegen darf, der bereits vorher angefangen hat.

Die Stelle (az,ap) würde genau dann in einem Namen liegen, der bereits vorher angefangen hat, wenn sie und die davor liegende Stelle jeweils ein Zeichen außer einem Leerzeichen enthielten:

```
1≤az≤n und 1≤ap-1 und ap≤Länge(F(az))
und mid(F(az),ap-1,1)≠Leerzeichen
und mid(F(az),ap,1)≠Leerzeichen
```

Die gesuchte Bedingung ist die Negierung davon. Nach Vereinfachung ist sie wie folgt:

az=n+1 **oder** ap=1 **oder** ap=Länge(F(az))+1
oder mid(F(az),ap-1,1)=Leerzeichen
oder mid(F(az),ap,1)=Leerzeichen

Im Gegensatz zur Bemerkung "die Stelle (az,ap) liegt nicht innerhalb eines Namens" ist die mathematisch formulierte Bedingung oben eindeutig und unmißverständlich.

A.3. Jede Identität kann durch Auflisten aller Kombinationen der möglichen Werte der Variablen x und ggf. y und z und Gegenüberstellen der berechneten Werte der fraglichen Ausdrücke bewiesen werden. Diese Vorgehensweise wird "Aufstellen einer Wahrheitstabelle" genannt. Alternativ können in vielen Fällen vorherige Identitäten für die jeweilige Beweisführung angewendet werden.

Es gibt vier mögliche Kombinationen der Werte von zwei Booleschen Variablen. Für drei Variablen gibt es acht Kombinationen der Werten:

falsch	falsch	falsch
falsch	falsch	wahr
falsch	wahr	falsch
falsch	wahr	wahr
wahr	falsch	falsch
wahr	falsch	wahr
wahr	wahr	falsch
wahr	wahr	wahr

A.4 (1). F = [(B **und** C) **oder** (**nicht** B **und** D)]. Es kann mit Hilfe einer Wahrheitstabelle gezeigt werden, daß diese Funktion die in der Aufgabe angegebene Definition erfüllt.

A.4 (2). [x **und** y ⟹ x] = [**nicht** (x **und** y) **oder** x]
= [**nicht** x **oder nicht** y **oder** x]
= [wahr **oder nicht** y] = wahr

[x ⟹ x **oder** y] = [**nicht** x **oder** x **oder** y] = wahr

Alternativ können die Aussagen mit Hilfe von Wahrheitstabellen bewiesen werden.

A.4 (3). [x **und** (y **oder** z)] = [x **und** x **und** (y **oder** z)]
= [x **und** (x **und** y **oder** x **und** z)]
= [x **und** (x **und** x **und** y **oder** x **und** z)]
= [x **und** x **und** (x **und** y **oder** z)]
= [x **und** (x **und** y **oder** z)]

Aus der dritten Zeile oben folgt:

= [x **und** (x **und** y **oder** x **und** x **und** z)]
= [x **und** x **und** (y **oder** x **und** z)]
= [x **und** (y **oder** x **und** z)]

Die Identität 5 im Anhang A, Abschnitt A.3 besagt, daß der erste Ausdruck gleich dem letzten ist.

A.4 (4). Aus a folgt x und aus b folgt y. Wenn sowohl a als auch b wahr sind, sind x und y wahr, also (x **und** y) ist wahr. Wenn entweder a oder b wahr ist, dann ist x bzw. y wahr, also (x **oder** y) ist wahr.

Formaler soll erstens gezeigt werden, daß

[(a ⟹ x) **und** (b ⟹ y)] ⟹ [a **und** b ⟹ x **und** y]

Dieser Ausdruck ist äquivalent zu

[(**nicht** a **oder** x) **und** (**nicht** b **oder** y)]
⟹ [**nicht** a **oder nicht** b **oder** x **und** y]

der wiederum äquivalent ist zu

nicht [(**nicht** a **oder** x) **und** (**nicht** b **oder** y)]
oder [**nicht** a **oder nicht** b **oder** x **und** y]

sowie zu

a **und nicht** x **oder** b **und nicht** y
oder nicht a **oder nicht** b **oder** x **und** y

Gemäß der Identität 17 im Anhang A, Abschnitt A.3, ist der Ausdruck oben gleich

nicht x **oder nicht** y **oder nicht** a **oder nicht** b **oder** y

und somit wahr (**nicht** y **oder** y = wahr).

Auf die gleiche Weise kann die zweite Aussage der Aufgabe,

$$[(a \Longrightarrow x) \text{ und } (b \Longrightarrow y)] \Longrightarrow [a \text{ oder } b \Longrightarrow x \text{ oder } y]$$

formal bewiesen werden.

A.4 (5).

1. [x und (nicht y oder z)]
   = [x und nicht y oder x und z]
2. [x oder nicht y oder z]
3. [nicht x oder nicht y oder z]
4. [nicht x und nicht y oder z]
5. $a \neq 0$
6. wahr
7. falsch
8. $a=0$

A.5 (1). $[\text{und}_{i=1}^{n} A(i)] = [\text{wahr und}_{i=1}^{n} A(i)]$

$= [(n<1 \text{ oder } n\geq 1) \text{ und}_{i=1}^{n} A(i)]$

$= [n<1 \text{ und}_{i=1}^{n} A(i) \text{ oder } n\geq 1 \text{ und}_{i=1}^{n} A(i)]$

$= [n<1 \text{ oder } n\geq 1 \text{ und}_{i=1}^{n} A(i)]$

$= [n<1 \text{ oder } n\geq 1 \text{ und } A(n) \text{ und}_{i=1}^{n-1} A(i)]$

Die entsprechende Ableitung für die oder-Reihe ist ähnlich:

$[\text{oder}_{i=1}^{n} A(i)] = [\text{wahr und } (\text{oder}_{i=1}^{n} A(i))]$

$= [(n<1 \text{ oder } n\geq 1) \text{ und } (\text{oder}_{i=1}^{n} A(i))]$

$= [n<1 \text{ und } (\text{oder}_{i=1}^{n} A(i)) \text{ oder } n\geq 1 \text{ und } (\text{oder}_{i=1}^{n} A(i))]$

$= [n<1 \text{ und falsch oder } n\geq 1 \text{ und } (\text{oder}_{i=1}^{n} A(i))]$

$= [n\geq 1 \text{ und } (\text{oder}_{i=1}^{n} A(i))]$

$= [n\geq 1 \text{ und } (A(n) \text{ oder}_{i=1}^{n-1} A(i))]$

Literaturhinweise

Alagić, Suad; Arbib, Michael A., *The Design of Well-Structured and Correct Programs*, Springer-Verlag, New York, Heidelberg, Berlin, 1978.

Baber, Robert L., *Software Reflected: The Socially Responsible Programming of Our Computers*, North-Holland Publishing Co., Amsterdam, New York, Oxford, 1982.

Baber, Robert L., *Softwarereflexionen: Ideen und Konzepte für die Praxis*, Springer-Verlag, Berlin, Heidelberg, New York, 1986.

Baber, Robert L., *The Spine of Software: Designing Provably Correct Software – Theory and Practice*, John Wiley & Sons, Chichester, 1987.

Backhouse, Roland C., *Program Construction and Verification*, Prentice-Hall International, Englewood Cliffs, N. J., 1986.

Bauer, Friedrich L.; Wössner, Hans, *Algorithmische Sprache und Programmentwicklung*, Springer-Verlag, Berlin, Heidelberg, New York, 1984.

Dahl, O.-J.; Dijkstra, E. W.; Hoare, C. A. R., *Structured Programming*, Academic Press, London, 1972.

Denvir, Tim, *Introduction to Discrete Mathematics for Software Engineering*, Macmillan Education, Basingstoke, 1986.

Dijkstra, Edsger W., *A Discipline of Programming*, Prentice-Hall, Inc., Englewood Cliffs, N. J., 1976.

Futschek, Gerald, *Programmentwicklung und Verifikation*, Springer-Verlag, Wien, New York, 1989.

Gries, David, *The Science of Programming*, Springer-Verlag, New York, Heidelberg, Berlin, 1981.

Hoare, C. A. R., *Communicating Sequential Processes*, Prentice-Hall International, Englewood Cliffs, N. J., 1985.

IEEE Spectrum, "Lethal dose", in Faults & Failures column, Vol. 24, No. 12, 1987 December, S. 16.

Jones, Cliff B., *Systematic Software Development Using VDM*, Prentice Hall International, Englewood Cliffs, N.J., 1986.

Joyce, Ed, "Software Bugs: A Matter of Life and Liability", *Datamation*, 1987 May 15, S. 88-92.

Linger, Richard C.; Mills, Harlan D.; Witt, Bernard I., *Structured Programming: Theory and Practice*, Addison-Wesley Publishing Co., Reading, Massachusetts, 1979.

Loeckx, Jacques; Sieber, Kurt, *The Foundations of Program Verification*, B. G. Teubner, Stuttgart, and John Wiley & Sons, Chichester, 1984.

McGowan, Clement L.; Kelly, John R., *Top-Down Structured Programming Techniques*, Petrocelli/Charter, New York, 1975.

Spivey, J.M., *The Z Notation*, Prentice Hall, New York, 1989.

Thomas, Martyn, *Should we trust computers?*, The BCS/UNISYS Annual Lecture 1988, 1988 July 4, British Computer Society, London.

Register