Methoden der imperativen Programmierung

mit Beispielen in
MODULA-2 und EIFFEL

von Dr. Hans-Georg Stork

Dr. rer. nat. Hans-Georg Stork

1947 in Frankfurt a. M. geboren. 1966-1971 Studium der Mathematik, Physik und Computer Science an den Universitäten Frankfurt a. M. und Iowa City, Iowa, USA. 1972- 1976 Wissenschaftlicher Mitarbeiter im Fachbereich Informatik der TH Darmstadt. 1976-1978 Wissenschaftlicher Assistent an der Universität Stuttgart, Fakultät für Informatik. 1978-1983 Industrietätigkeit (AEG-Telefunken und SESA Deutschland GmbH). 1983-1990 Wissenschaftlicher Mitarbeiter (AT) und Lehrbeauftragter am Institut für Angewandte Informatik und Formale Beschreibungsverfahren der Universität Karlsruhe. Seit 1990 Angestellter der EG Kommission in Ispra (Italien) und Luxemburg.

Karl Stork

(1907 - 1992)

zum Gedächtnis

Inhaltsverzeichnis

Vorwort

Seit Anfang der siebziger Jahre gibt es an der Universität Karlsruhe die Möglichkeit, das Fach "Wirtschaftsingenieurwesen" mit dem Schwerpunkt Informatik zu studieren. Studenten, die sich für diesen Schwerpunkt entscheiden, wird - neben einführenden Vorlesungen über Mathematik und aus dem Bereich der Wirtschaftswissenschaften - eine breite Grundausbildung in Informatik angeboten. Deren Kern ist ein viersemestriger Kurs, welcher die wichtigsten Kenntnisse zum Verständnis des Aufbaus, des Betriebs und der Anwendung von Rechnern vermittelt. Zur Befriedigung eines eher "akademischen" Erkenntnisinteresses wird dabei - und dies gilt durchaus für den gesamten Studiengang - ganz bewußt der Nutzen hinzukalkuliert, den ein solches Wissen und die damit verbundenen Fertigkeiten für die berufliche Qualifikation der Absolventen haben sollten.

Dieses praktische Interesse ist sicher eine der stärksten Motivationen dafür, daß immer häufiger Themen für abschließende Diplomarbeiten angeboten und gewählt wurden, welche die Spezifikation, den Entwurf und die Implementierung größerer Programme beinhalteten, mitunter auch als Teile umfangreicher und länger laufender Projekte. Bei der Anfertigung und Betreuung solcher Arbeiten wurde für Lernende und Lehrende nicht selten ein Mangel erkennbar, der offenbar auf eine Lücke im Lehrangebot zurückzuführen war: Vielen Studenten fehlte ein hinreichend klares Bewußtsein für das, was die Qualität eines Software-Produktes ausmacht, ganz zu schweigen von einem guten Verständnis der Möglichkeiten, diese Qualität zu erzielen. Die während des ersten Semesters geübte Programmierung beschränkte sich in der Regel auf kleine, isolierte Beispiele, auf Illustrationen zum Begriff des Algorithmus, und für eine ausführliche Darlegung relativ einfacher Tips zur Programmgestaltung oder gar von Verfahren zur systematischen Herleitung der Gliederung und Feinstruktur größerer Programme gab es einfach nicht genug Zeit.

Abhilfe wurde zu Beginn der achtziger Jahre geschaffen: Durch Bereicherung des Curriculums um eine Vorlesung mit zugeordnetem Praktikum unter dem Titel "Programmier-Methodik". Den Studenten wurde damit eine Gelegenheit gegeben, die (potentiell lange) "programmierfreie Zeit" zu unterbrechen und - begleitet von den öffentlichen (und privaten) Ratschlägen des Dozenten - gemeinsam mit Kommilitonen im "Teamwork" ein Semesterprojekt zu bearbeiten. Dessen Ergebnis, ein (hoffentlich) lauffähiges Programm und die Dokumentation seiner Entwicklung, konnte dann einen willkommenen Beitrag zur Aufbesserung der Vordiplom-Note leisten.

Damit sind der Grund und die Entstehungsgeschichte dieses Buches im wesentlichen schon fast erklärt. Während meiner Zeit als Mitarbeiter und Lehrbeauftragter am Institut für Angewandte Informatik und Formale Beschreibungsverfahren der Universität Karlsruhe in den Jahren von 1983 bis 1990 nämlich, hatte

ich mehrere Male das Vergnügen, jene Vorlesung über "Programmier-Methodik" zu halten und das zugehörige Praktikum zu organisieren. Und manche Studenten waren von der Arbeit unmittelbar am Rechner[1] so begeistert und von den dabei erfahrenen Erfolgs- oder Mißerfolgserlebnissen so angespornt, daß sie es nicht versäumten, auch meine ergänzende und weiterführende Vorlesung "Software-Engineering" zu besuchen.

Das vorliegende Buch enthält also eine Auswahl und Synthese von Themen, die in diesen Vorlesungen zur Sprache kamen. (Stellenweise ist die Darstellung etwas detaillierter und stellenweise ist sie etwas breiter, als sie den Hörern erinnerlich sein mag.) Ursprünglich angeregt wurde es von Hans-Werner Six, der - einem Ruf an die Fernuniversität Hagen gefolgt - mir den Vorschlag machte, die Vorlesungen zu einem Korrespondenzkurs umzuarbeiten. Obwohl dieser Vorschlag nicht in die Tat umgesetzt wurde, gab er doch zunächst Anlaß zur Niederschrift einiger Abschnitte, die immerhin soweit gedieh, daß Wolffried Stucky, mein Institutschef und Mitherausgeber der "Leitfäden", mich dazu bewog, an einer langen Geschichte weiterzuschreiben und ihr die Form eines Buches zu geben. Beiden Herren danke ich für die gründliche Lektüre früher Versionen einzelner Kapitel und für manchen wertvollen Fingerzeig.

Danksagungen müssen selbstverständlich ergehen an die vielen Autoren, welche die Bausteine für ein Lehrbuch wie dieses geformt und geliefert haben. Zu nennen sind hier in erster Linie und stellvertretend für manche andere: D. Gries, J. V. Guttag, M. Jackson, B. W. Kernighan, H. F. Ledgard, B. Meyer, D. L. Parnas, C. A. Petri, N. Wirth und D. Wood.

Einen speziellen Dank schulde ich freilich Hans-Jürgen Ehling, meinem Mentor während meiner Zeit beim Software-Zentrum des damaligen AEG-Telefunken Konzerns. Er war mir ein Meister des Programmierens, und er wird es mir hoffentlich nachsehen, wenn er von der hohen Kunst, mit der er die von Warnier und Orr begründete Klammerdiagramm-Technik weiterentwickelte, in diesem Buch nur einen schwachen Abglanz entdeckt.

Die genannten Namen deuten schon an, daß sich die Auswahl des Stoffes für ein Buch wie dieses weitgehend an den Interessen und Vorlieben des Autors orientieren wird, und natürlich auch an dem, was er für hinreichend fundiert und praktikabel zugleich hält. Dies mag manche Erwartungen enttäuschen. Doch mir schien, daß es für unser Gebiet genügend Literatur von enzyklopädischem Charakter gibt, um nicht noch ein weiteres Exemplar dieser Art hinzuzufügen. Die Beschränkung des Gesichtskreises auf Methoden, welche üblicherweise dem *imperativen* Paradigma des Programmierens (vgl. Kapitel 1) zugerechnet werden, ergab sich übrigens ganz von selbst aus dem oben skizzierten Zweck der Vorlesungen.

Zu erklären ist noch die Wahl von MODULA-2 als diejenige Programmiersprache, in der wir erstens über "Stil" reden, und in die wir zweitens mehr oder

[1] Der schaute in den frühen achtziger Jahren noch vergleichsweise imposant aus und konnte seinem Beherrscher ein gewisses Gefühl des Stolzes geben.

weniger abstrakte Konzepte zur Gliederung von Programmen abbilden. Zunächst einmal: Ohne die Bezugnahme auf eine konkrete Programmiersprachen müßten - insbesondere aus der Sicht des Lernenden - auch die besten Ideen über Software-Entwicklung "in der Luft" hängenbleiben; dem wird wohl jedermann zustimmen (mancher vielleicht *nolens volens*.) Warum aber MODULA-2? Die Antwort ist denkbar einfach: Es ist die Sprache, welche unsere Studenten als erste lernen, mit der sie also am besten vertraut sein sollten. In der Tat ist dies eine Grundvoraussetzung für die Beschäftigung mit diesem Buch, und es ist wichtig, in aller Deutlichkeit auf sie hinzuweisen: Wir nehmen an, daß der Leser ~~gute~~ Kenntnisse über MODULA-2 und möglichst auch etwas praktische Erfahrung mit ihrer Anwendung besitzt. Er kann sich diese Kenntnisse im Selbststudium erwerben (dafür hilfreiche Literatur sowie auf Personal Computern laufende MODULA-2 Compiler gibt es ja in Hülle und Fülle) oder er kann sich an der Universität Karlsruhe für den ersten Teil des oben erwähnten Grundkurses in Informatik einschreiben.

Doch um der Wahrheit die Ehre zu geben: Irgendwann im Verlauf unserer Reise durch die "Methodenlandschaft" wird das Gelände auch für ein ansonsten sehr wendiges Vehikel wie MODULA-2 unwegsam werden, und wir müssen auf ein stärkeres und vielseitigeres Gefährt umsteigen. Es wird EIFFEL sein, die von B. Meyer konzipierte "objektorientierte" Sprache. Wir werden ihre Wahl an geeigneter Stelle begründen. Nur soviel vorweg: Eine Kenntnis *dieser* Programmiersprache setzen wir nicht voraus, doch hoffen wir, beim Leser soviel Interesse für sie zu erwecken, daß er es für Wert hält, ihr ein intensiveres Studium zu widmen.

Frühjahr 1993

Hans-Georg Stork

1 Imperative Programmierung

Rechner sind Apparate, die mit Worten dazu veranlaßt werden, Dienstleistungen zu erbringen, welche auf der Manipulation symbolischer Repräsentationen von Information beruhen. Die Beispiele für solche Dienstleistungen sind Legion. Wir zählen hier nur einige wenige auf:

- Eine quadratische Gleichung lösen;
- das Erstellen und Redigieren von Texten unterstützen;
- eine Werkzeugmaschine steuern;
- ein Kraftwerk überwachen;
- einen ökonomischen Prozeß simulieren;
- für einen Betrieb die Lohnabrechnung vornehmen;
- Telefongespräche vermitteln;
- Daten zu einem anderen Rechner übertragen;
- die Primzahlen zwischen 1 und 1000 finden;
- den Bestand einer Bibliothek verwalten;
- Grafiken zeichnen;
- als Schachspielgegner fungieren;
- Bankkonten führen;
- Meßergebnisse auswerten;
- seinen eigenen Betrieb steuern;
- von einer Sprache in eine andere übersetzen;
- ..., ...

Doch wie der klassische Besenstiel des Zauberlehrlings ([GOE]) geraten auch Rechner bei der Ausführung ihrer Aufgabe leicht außer Kontrolle, wenn sich zur rechten Zeit das rechte Wort nicht einstellt.

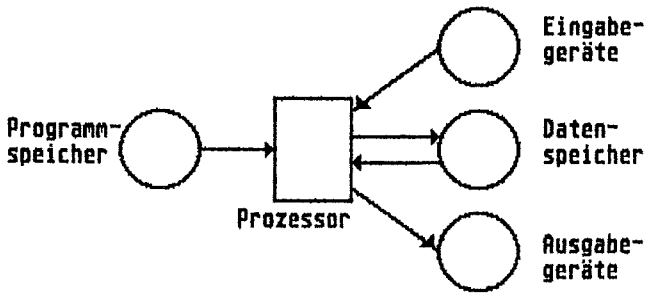
"Das rechte Wort zur rechten Zeit", ist die Metapher für jene Tätigkeit, welcher dieses Buch gewidmet ist: das Programmieren oder - mit anderem Ausdruck - die Entwicklung von *Software*. Es ist eine Tätigkeit, welcher mehr und mehr Gewicht zukommt auf allen Gebieten industriellen und administrativen Wirkens, seitdem es gelingt, elektronische Bauelemente immer höherer Komplexität, von immer kleineren Ausmaßen und in immer größerer Zahl zu produzieren, um sie zu jenen Geräten (der *Hardware*) zusammenzusetzen, welche auf das Wort des Programmierers (sei er Lehrling oder alter Meister) "hören" können. Es ist für manche Zeitgenossen eine überaus faszinierende Tätigkeit, und dafür kann es ganz verschiedene Gründe geben. Da ist der sich so unversehens öffnende neue Bereich für schier grenzenlose Kreativität (siehe die obige Liste), eine weite Spielwiese für den freien Lauf mehr oder weniger verrückter oder auch profitabler Ideen. Es ist ein Bereich, der heute Menschen anzieht, die - hätten sie in früheren Epochen gelebt - vielleicht Maler, Kunsthandwerker, Steinmetz,

Orgelbauer oder Musiker geworden wären. Aber da ist andererseits auch die Versuchung, einer atavistischen Lust an Macht und Herrschaft zu frönen, einem Apparat - wenn schon nicht einem Mitmenschen - seinen Willen aufzuzwingen, ihn gar zum Homunculus zu machen. Mit Weizenbaum (vgl. [WEI]) sind wir der Meinung, daß eine solche anmaßende und verbissene Haltung in mehrfacher Hinsicht von Schaden ist, nicht zuletzt für denjenigen selbst, der sie einnimmt (Weizenbaum nennt ihn "Hacker") und sich damit unbewußt zum Sklaven einer Sucht erniedrigt.

Wir glauben, daß ein solches Wort der Warnung gerade in einem Buch über Methoden der "imperativen" Programmierung durchaus angebracht ist. Wir empfehlen, den Rechner schlicht als Werkzeug zu sehen, als Mittel zu allerlei gutem Zweck, auch als Spielzeug, wenn man denn so will, doch nicht als ein Objekt, an dem sich wie auch immer motivierte Allmachtsphantasien abreagieren ließen. Was wir dem Rechner "befehlen", will wohlbedacht sein und soll ihn zu einem nützlichen Instrument machen. Hierfür ein Bewußtsein zu schaffen und für die Qualität der Produkte des Programmierens, ist das Ziel dieses Buches. Der Zauberlehrling soll das Wort der alten Meister hören und anwenden lernen, damit er zumindest technisch dazu in der Lage ist, dem Besen zu gegebener Zeit den nötigen Einhalt zu gebieten.

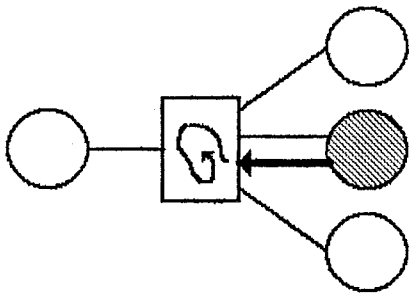
1.1 Maschine und Sprache

Möchte man von einem Menschen, daß dieser eine bestimmte Dienstleistung ausführt, so gibt es in der Regel verschiedene Möglichkeiten, ihn dazu zu befähigen. Unter anderem kann man ihm das Gewünschte beschreiben, man kann ihm ein Beispiel geben, oder man kann ihm einen Plan in die Hand drücken, aus dem klar hervorgeht, welche Einzelschritte in welcher Reihenfolge zu unternehmen sind, um letztlich die Dienstleistung zu erbringen. "Tue dies, dann tue jenes, usw." steht in dem Plan, und "wenn diese Bedingung erfüllt ist, dann wende dich einem anderen Arbeitsgang zu". Dies sind (bedingte) *Befehle*, und der Plan ist ein *imperatives Programm*.

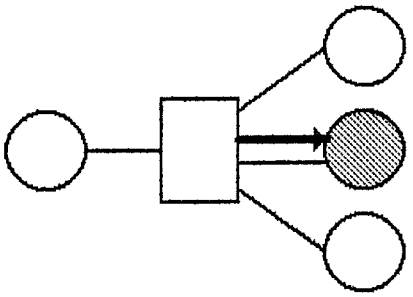


Es ist auch die klassische Weise, Dienstleistungen von einem Rechner anzufordern. Das ist die Sprache, die zu verstehen Rechner von frühester Zeit an konstruiert wurden. Was Rechner überhaupt tun können, muß sich in ihr ausdrücken lassen, nicht mehr und nicht weniger. Und was Rechner tun können, ist durch ihre Komponenten und ihren Bauplan bestimmt. Auch daran, an der sogenannten "von Neumann"-Architektur, hat sich

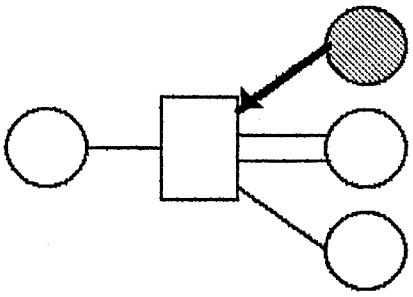
seit den ältesten elektronischen Rechenanlagen prinzipiell nicht viel geändert. Nach wie vor steht der Prozessor als "Befehlsempfänger" im Mittelpunkt des Geschehens (vgl. die Abbildung auf der vorigen Seite), und die einzigen von ihm zu erwartenden Dienstleistungen beziehen sich auf ihn selbst natürlich, sowie auf die um ihn angeordneten Komponenten: Datenspeicher, Programmspeicher (beide oft physikalisch vereint), Eingabe- und Ausgabegeräte:



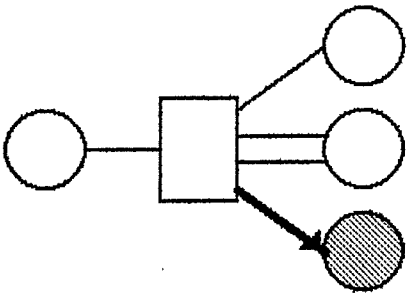
A: Hole Daten aus dem Datenspeicher (und evtl.: manipuliere sie intern)!



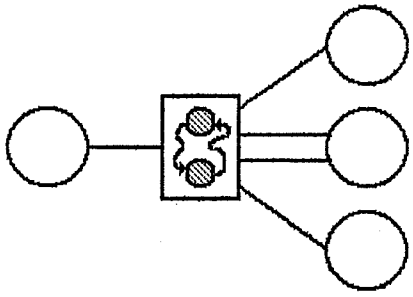
B: Lege Daten im Datenspeicher ab!



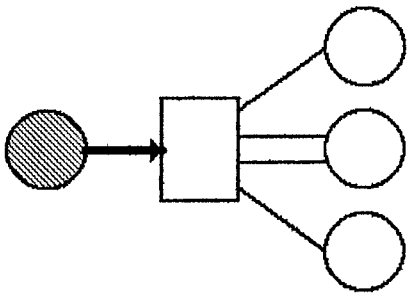
C: Fordere Daten von einem Eingabegerät an!



D: Sende Daten an ein Ausgabegerät!



E: Manipuliere die Inhalte deiner internen Speicher!

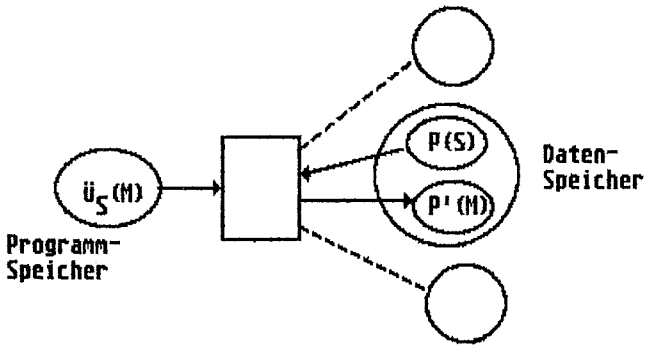


F: Hole dir einen neuen Befehl!

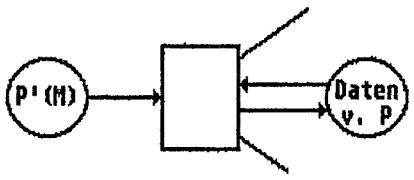
Diese sechs Bilder repräsentieren die Klassen der von einem (von Neumann) Prozessor "verstandenen" Befehle. Die Gesamtheit dieser Befehle bildet gewissermaßen die "Muttersprache" des Prozessors, also die Sprache, in der letztlich

die Erledigung einer vom Prozessor zu bearbeitenden Aufgabe in allen Details beschrieben werden muß. Im Programmspeicher stellen sich Texte in dieser Sprache als Folgen binär codierter Signale dar. Andererseits gilt offenbar: Aus der Sicht des Programmierers ist die Maschine durch die Sprache, die ihr Prozessor versteht, vollständig charakterisiert.

Diese Bemerkung ist insofern interessant, als sie es nahelegt, die Mühen, die es wohl macht, einen Programmtext in der Muttersprache eines Prozessors (auch *Maschinensprache* genannt) zu verfassen, dadurch zu erleichtern, daß man dem Prozessor eine neue Sprache beibringt. Eine Sprache zum Beispiel, in der sich die Lösung allfälliger Aufgaben sachgerechter gliedern und formulieren läßt. Zu solcher Erweiterung der sprachlichen Fähigkeiten einer Maschine bieten sich im allgemeinen zwei Verfahren an: das des *Übersetzens* und das des *Interpretierens*:

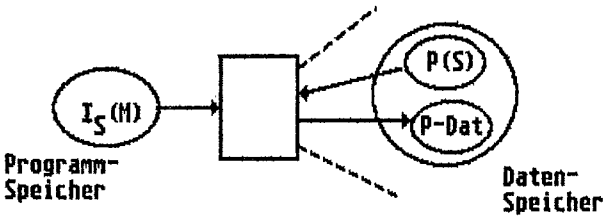


Der Übersetzer ist ein in Maschinensprache M im Programmspeicher vorliegendes Programm $\ddot{U}_S$(M). Es veranlaßt den Prozessor, das im Datenspeicher abgelegte, in einer Sprache S verfaßte Programm P(S) auf ein äquivalentes, vom Prozessor verstehbares Programm P'(M) abzubilden,



welches dann seinerseits in den Programmspeicher geladen werden kann. Von dort aus wird es den Prozessor zu den ursprünglich in P(S) beschriebenen Aktionen treiben, also unter anderem die von P(S) vorgesehenen Datenmanipulationen vornehmen lassen.

(Ohne weitere Problematisierung: Dies meinen wir, wenn wir sagen, P'(M) sei zu P(S) äquivalent.)



Der Interpreter I_S(M) einer Programmiersprache S geht einen direkteren Weg. Der Prozessor wird durch ihn bewogen, das im Datenspeicher befindliche Programm P(S) Stück für Stück zu lesen und die darin niedergelegten Operationen (an den Daten P-Dat von P, aber auch zur Eingabe und Ausgabe) auszuführen. In beiden Fällen, dem des Übersetzers und dem des Interpreters, ist das Wissen um die "neue" Sprache S als Programm in Maschinensprache kodiert.

Die Maschinensprache eines "von Neumann - Rechners" ist imperativ, S dagegen muß es nicht sein. Zum Beispiel könnten wir S derart gestalten, daß sich dem Rechner die Aufgabe, die kleinere von zwei Zahlen zu ermitteln, auf die folgende Weise stellen läßt:

"*Finde min mit: min<= x und min<= y und (min = x oder min = y)*"

Der Übersetzer hätte dann das Problem, aus dieser *Beschreibung* des Minimums zweier Zahlen ein Programm in Maschinensprache zu produzieren, das mit einer Folge bedingter Befehle dem Prozessor exakt angibt, was er zu tun hat, um den Wert von *min* zu finden. Und ein Interpreter für eine solche *deskriptive* Sprache hätte die gegebene Beschreibung zu analysieren und dem Prozessor direkt zu sagen, wie er vorzugehen hat.

Sowohl ein Übersetzer als auch ein Interpreter hätten es um einiges einfacher, wenn die Sprache S selbst schon den befehlsartigen Charakter der elementaren Maschinensprache in sich aufnähme. Der Weg von einem Programm in S zu einer vom Prozessor ausführbaren Folge von Anweisungen ließe sich damit vermutlich beträchtlich verkürzen. Imperative Arbeitsvorschriften wie

"*setze min := x; falls y < min so setze min := y*"

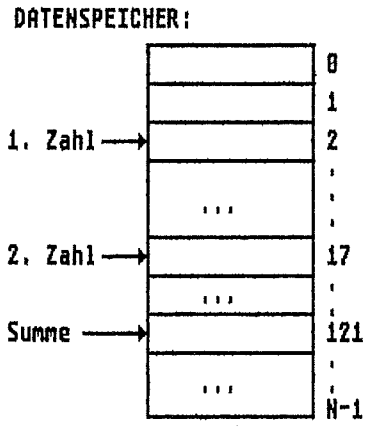
oder

"*falls x <= y so setze min := x sonst setze min := y*",

welche genau festlegen, was zu tun ist, um das Resultat zu gewinnen, sind sicher leichter in Befehle der Maschinensprache zu übertragen als eine Beschreibung der Eigenschaften des gewünschten Resultats.

Doch auch wenn sich die Maschine dem Programmierer als eine solche *imperative höhere* Sprache darstellt, hat er schon viel gewonnen, wie wir uns im folgenden an einem sehr einfachen Beispiel klarmachen wollen. (Als *höhere Sprache* sei hier vorläufig eine Sprache apostrophiert, die vom Prozessor nur mit Hilfe eines Übersetzers oder Interpreters verstanden werden kann.)

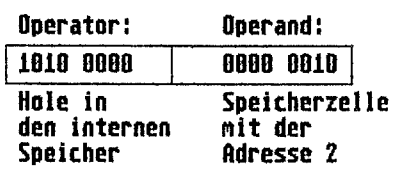
Nehmen wir an, der Rechner soll die Summe zweier, von einem Eingabegerät zu lesender ganzer Zahlen bilden und zur weiteren Verfügung vorhalten. Wir wollen uns zunächst in die Lage des Programmierers versetzen, der die "nackte" Maschine vor sich hat. Er sieht den Datenspeicher als eine endliche Folge von "Datenbehältern", für deren Größe wir uns hier gar nicht interessieren wollen. Es möge genügen, daß sich der Inhalt eines solchen Behälters als ganze Zahl deuten läßt. Der Programmierer muß nun bestimmen, wo die beiden einzulesenden Zahlen und wo ihre Summe abgelegt werden sollen. Und in den Programmspeicher muß er in binärer Form die Befehle schreiben, welche den Prozessor



zur Erledigung der erforderlichen Operationen auffordern. (Wir nehmen an, er verfügt zu diesem Zweck über geeignete Vorrichtungen, vielleicht ein Schaltbrett.) Diese Befehle sind:

(1)	Fordere eine ganze Zahl vom Eingabegerät an!	0000 1000
(2)	Lege die empfangene Zahl im Datenbehälter Nr. 2 ab!	0010 0000 0000 0010
(3)	Fordere eine ganze Zahl vom Eingabegerät an!	0000 1000
(4)	Lege die empfangene Zahl im Datenbehälter Nr. 17 ab!	0010 0000 0001 0001
(5)	Hole den Inhalt von Behälter Nr. 2 in deinen internen Speicher!	1010 0000 0000 0010
(6)	Hole den Inhalt von Behälter Nr. 17 und addiere ihn auf den Inhalt deines internen Speichers!	1011 0000 0001 0001
(7)	Lege den Inhalt deines internen Speichers im Behälter Nr. 121 ab.	0010 0000 0111 1001

Natürlich ist der hier verwendete Binärcode rein hypothetisch und frei erfunden. Doch immerhin haben die Befehle mit denen von real existierenden Codes zwei wesentliche Teile gemeinsam, von denen der eine die auszuführende Operation identifiziert (den Operator) und der andere die Adresse eines "Behälters" im Datenspeicher (den Operanden) (vgl. nebenstehende Abbildung).



Die historisch früheste Errungenschaft zur Erleichterung der Zusammenstellung und Eingabe von Befehlen in Maschinensprache bestand darin, Operationscodes "mnemotechnisch" und Adressen in gewohnten Zahlendarstellungen zu notieren, z.B.:

READINT	"Fordere eine ganze Zahl vom Eingabegerät an!"
SP x	"Speichere den Inhalt des internen Speichers im Behälter x!"
LD x	"Lade den Inhalt von Behälter x in den internen Speicher!"
LDA x	"Addiere den Inhalt von Behälter x auf den Inhalt des internen Speichers!"

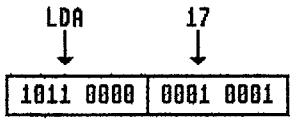
usw.

Das obige Additionsprogramm erhielte damit die folgende Gestalt:

```
READINT
SP 2
READINT
```

```
SP 17
LD 2
LDA 17
SP 121
```

Der "Übersetzer" eines solchen Programms hat nichts anderes zu tun, als die mnemotechnische Notation auf den binären Maschinencode abzubilden:



Dieser Vorgang wird als *Assemblieren* bezeichnet, der Übersetzer selbst in diesem Falle als *Assemblierer* (engl.: *Assembler*) und der mnemotechnische Code daher auch als *Assembler-Sprache*.

Angesichts der Ansprüche, die wir noch stellen werden, macht es freilich keinen Sinn, Assembler-Sprachen als "höhere" Programmiersprachen zu klassifizieren. Nichtsdestotrotz lohnt es sich, zumindest zwei interessante Eigenschaften hervorzuheben:

- Die symbolische Darstellung der Anweisungen an den Prozessor ist dem menschlichen Auffassungsvermögen angepaßter.
- Mit dem Sprachelement *Kommentar* kann das Verständnis eines Programmtextes entscheidend unterstützt werden, z.B.:

```
LD 2    ; Lade 1. Zahl
LDA 17  ; Bilde Summe 1. Zahl + 2. Zahl
SP 121  ; Speichere Summe
```

Doch welch zusätzlichen Komfort auch immer ein Assembler noch bieten mag, er befreit den Programmierer nicht von der Last, selbst für die Organisation des Datenspeichers zu sorgen (was kommt wohin...?) und jede für das menschliche Bewußtsein elementare Operation (wie z.B. Addition, Subtraktion, etc.) in kleine, der Maschine angepaßte Schritte zu zerlegen.

Was ihm wirklich weiterhilft, ist eine Sprache, in der auf Datenbehälter (Speicherzellen) und ihre Inhalte durch symbolische Namen Bezug genommen werden kann, und in der Berechnungen und sonstige Manipulationen von "Informationsobjekten" in einer Weise ausdrückbar sind, die nicht zu weit von mehr oder weniger traditionell gebräuchlichen Formalismen (z.B. der Arithmetik) entfernt ist. So sollten die "Behälter" für die erste und die zweite Zahl und für deren Summe (vgl. das obige "Assembler-Programm") etwa durch die Namen "a", "b" und "c" identifiziert werden, und der Befehl, die Summe der Inhalte von a und b zu bilden, könnte eine der folgenden (immer noch sehr imperativen!) Formen annehmen:

```
c := a + b;          ADD a TO b GIVING c
LET c = a + b        c <- +(a,b)
```

Verglichen mit der "Primitivität" der Maschinensprache ist eine solche Sprache tatsächlich auf einer "höheren Ebene", viel näher an menschlichen Denk- und

Formulierungsgewohnheiten. Der physikalische Speicher wird in ihr durch das Konzept der *Variablen* ersetzt. Und der Übersetzer (engl.: *Compiler*) übernimmt hier offenbar unter anderem einen großen Teil der Arbeit des Assembler-Programmierers:

- Er muß den Variablen a, b und c realen Speicherplatz mit realen Adressen zuordnen. Dies kann entweder implizit geschehen, d.h. nach dem Erkennen eines Variablennamens in einer Anweisung, oder explizit, d.h. vor dem ersten Gebrauch eines Variablennamens in einer Anweisung. Der Programmierer hat in diesem Fall vorab zu erklären (zu *deklarieren*, wie beim Zoll!), welche Variablen für welche Art von Objekten er zu verwenden gedenkt.

  VAR a, b, c: INTEGER;

  ist ein Beispiel für eine solche Deklaration.
- Der Compiler muß die Anweisung, welche c = a + b bewirkt, in eine entsprechende Folge von Maschinencode-Befehlen übertragen.

Wir wollen im folgenden einige der Mittel, die höhere imperative Programmiersprachen (und ihre Übersetzer) zur "menschengerechteren" und "problemorientierten" Formulierung von Anweisungen an einen Rechner offerieren, beispielhaft Revue passieren lassen. Ein Anspruch auf Vollständigkeit ist dabei natürlich nicht zu erheben. Auch die Auswahl der Programmmiersprachen ist nicht durchgängig und ziemlich willkürlich. Der Leser sei im übrigen - was die Definitionen und den praktischen Gebrauch der zitierten Sprachen betrifft - auf das Literaturverzeichnis am Schluß dieses einleitenden Kapitels verwiesen. (MODULA-2 wird, wie im Vorwort begründet, in diesem Buch eine Sonderstellung einnehmen als diejenige Sprache, in der wir die meisten Beispiele zur Untermauerung unserer Methoden der imperativen Programmierung darlegen werden.)

(i) Notationen zur Deklaration von Namen zur Identifizierung von Variablen und Konstanten und zur Festlegung zulässiger Wertmengen:

```
INTEGER A, B, C                                   (FORTRAN)

77   A    PIC ...    USAGE IS COMP
77   B    PIC ...    USAGE IS COMP
77   C    PIC ...    USAGE IS COMP                 (COBOL)

d: constant  1.5;
a, b, c: integer;                                 (ADA)

const  d = 1.5;
var a, b, c: integer;                             (PASCAL)

#define  d    1.5
int a, b, c;                                      (C)
```

```
CONST d = 1.5;
VAR a, b, c: INTEGER;                          (MODULA-2)
```

(ii) Notationen für arithmetische, logische und (evtl.) sonstige Operationen und darauf aufbauende Ausdrücke:

```
a * 2 ** n
(A .EQ. B) .AND. (B .LT. C)                    (FORTRAN)

(a <> b) and (a < c)
2 * (a + b)
a div b                                        (PASCAL)

A IS NOT GREATER THAN B
A IS NOT > B                                   (COBOL)

(a != b) && (a == c)
a++                                            (C)

a DIV b
(a <> b) AND (a < c)                           (MODULA-2)
```

(iii) Notationen für die Zuweisung der Werte von Ausdrücken an Variable:

```
LET F = PI * R * R                             (BASIC)

F = PI * R * R                                 (FORTRAN)

flaeche := pi * radius * radius;               (PASCAL, ADA,
                                                MODULA-2)

flaeche = pi * radius * radius;                (C)

MULTIPLY PI BY RQUAD GIVING FLAECHE            (COBOL)
```

(iv) Notationen zur

- *Formulierung von Bedingungen für die Ausführung einer Folge von Anweisungen,*
- *Formulierung von Bedingungen für die wiederholte Ausführung einer Folge von Anweisungen:*

```
if  a < b then x := a else x := b;
while  n < 100 do
begin
   a := b + 1;
   n := n + 1
end;                                           (PASCAL)
```

```
      IF A .LT. B GOTO 10
      X = B
10    CONTINUE
      DO 15 I = 1, N
      S := S + 1
15    CONTINUE                                    (FORTRAN)
```

```
FOR I = 1 TO N
LET S = S + 1
NEXT I                                            (BASIC)
```

```
for (i = 1; i <= n; ++i) s = s + 1;
while (n < 100)
   {a = b + 1;
   n++
   }                                              (C)
```

```
IF a < b THEN x := a ELSE x := b END;
WHILE n < 100 DO
   a := b + 1;
   n := n + 1
END;                                              (MODULA-2)
```

(v) Notationen zur Definition komplexer Datenobjekte, ausgehend von einfacheren Objekten (Benennung, Festlegung von Wertemengen, Bezeichnung von Teilobjekten, etc.):

```
DIMENSION MATRIX (10, 17)                         (FORTRAN)
```

```
DIM A$(5)                                         (BASIC)
```

```
type string = packed array [1..80] of char;
var buch:  record
              autor, titel: string;
           end;                                   (PASCAL)
```

```
01 BUCH  OCCURS  15
   02 AUTOR   PIC  X(20)
   02 TITEL   PIC  X(50)                          (COBOL)
```

```
struct buch {
              char autor[20];
              char titel[50];
            };                                    (C)
```

```
TYPE BuchType = RECORD
                  autor: ARRAY [0..19] OF CHAR;
                  titel:  ARRAY [0..49] OF CHAR
                END;
VAR buch: BuchType;                                    (MODULA-2)
```

(vi) Konstrukte zur problemorientierten Gliederung von Programmtexten:

```
SUBROUTINE
FUNCTION
BLOCK DATA                                             (FORTRAN)

DIVISION
SECTION
PARAGRAPH                                              (COBOL)

      GOSUB 100
      ...
100   ...
      ...
      RETURN                                           (BASIC)

procedure
function                                               (PASCAL)

package
task                                                   (ADA)

Funktionsnamen                                         (C)

MODULE
DEFINITION MODULE
IMPLEMENTATION MODULE
PROCEDURE                                              (MODULA-2)
```

(vii) Notationen zur Formulierung von Kommentaren:

```
C     BERECHNUNG DER KREISFLAECHE
      F = PI * R ** 2                                  (FORTRAN)

flaeche := pi * radius * radius   {Kreisfläche};       (PASCAL)

f := pi * r * r;  -- Kreisfläche                       (ADA)

flaeche = pi * r * r; /* Kreisfläche */                (C)

flaeche := pi * radius * radius   (*Kreisfläche *);    (MODULA-2)
```

Man muß sich sehr deutlich darüber im Klaren sein, daß imperative Programmiersprachen nur einem von vielen möglichen Denkmustern (*Paradigmata*) bei der (Formulierung der) Lösung von Aufgaben der Informationsverarbeitung entgegenkommen. Es ist das Denkmuster, welches der zu Beginn dieses Abschnitts skizzierten "von Neumann"- Architektur zugrunde liegt, einer Architektur, die im wesentlichen durch eine spezielle rigide Befehlsstruktur definiert ist. Höhere imperative Programmsprachen machen (wie die oben aus ihren Zusammenhängen gerissenen Beispiele andeuten) den "von Neumann"- Rechner "handhabbarer", sie heben - um einen später noch genauer zu erläuternden Begriff in's Spiel zu bringen - das Niveau dieser Basismaschine, ändern aber nicht ihren eigentlichen Charakter, durch Anweisungen getrieben zu werden.

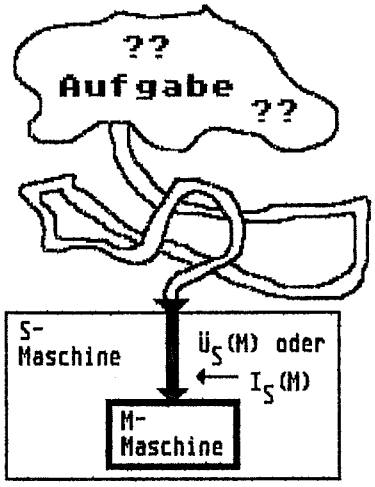
Imperative Programmiersprachen repräsentieren nur einen von vielen möglichen Kalkülen zur Berechnung des Berechenbaren. Andere Denkmuster entsprechen anderen Kalkülen, zum Beispiel Turing-Maschinen, rekursiven Funktionen, Termersetzungssystemen, Deduktionssystemen, usw. (vgl. z.B. [HER]). Und die meisten dieser Kalküle haben, in mehr oder weniger reiner Form, ihren Niederschlag in der Definition (nicht-imperativer) höherer Programmiersprachen gefunden (LISP ([MCC]) und PROLOG ([CLM]) sind darunter nur die vielleicht bekanntesten). Andererseits ist zu konstatieren, daß imperative Sprachen zweifellos zu den erfolgreichsten gehören, und dies gilt insbesondere in industriellen bzw. kommerziellen Anwendungsbereichen. Liegt dies etwa daran, daß es so viel einfacher scheint oder emotional befriedigender, Aufgaben "auf Befehl" erledigen zu lassen, als auf irgendeine andere Weise? Oder daran, daß wir uns bei der Lösung eines Problems nicht mit der Beschreibung des Resultats zufriedengeben möchten, sondern - vom Ehrgeiz des Konstrukteurs besessen - auch wissen wollen, auf welchem Weg wir es erhalten können?

Diesen und anderen möglichen Gründen nachzugehen und sie im einzelnen zu erforschen, würde weit aus dem für dieses Buch abgesteckten Territorium hinausführen (in unwegsame Gelände der Psychologie zum Beispiel). Aber vielleicht muß man sich in solch entlegene Gegenden gar nicht begeben. Vielleicht kann man sich mit der Erklärung begnügen, daß - wie es weiter oben schon anklang - das praktisch allen verbreiteten Rechnern zugrundeliegende Maschinenmodell den effizientesten Gebrauch seiner Ressourcen herausfordert, und daß dieser eben am leichtesten über eine imperative Sprache zu erreichen ist.

1.2 Ziele und Wege

Dieses Buch handelt von *Methoden* der imperativen Programmierung. Methoden sind Wege zu Zielen. Wenn wir also von Methoden sprechen, so sind wir auch im Obligo zu erklären, wohin diese uns führen sollen, und welches der Ausgangspunkt ist. Das scheint nicht schwer, haben wir doch das offensichtlichste Ziel schon vor Augen, nämlich die Maschine mit ihrem Prozessor, der eine aus

Befehlen bestehende Sprache versteht. Dieser ist so zu instruieren, daß er eine vorbestimmte Aufgabe erledigt. Am Beginn des Weges steht somit die Aufgabe, von der uns gesagt wurde oder von der wir annehmen, daß sie mit Hilfe eines Rechners lösbar ist (vgl. etwa die Liste von Dienstleistungen auf der ersten Seite dieses Kapitels). Und am Ende wollen wir Texte in der vom Prozessor verstandenen Sprache, welche genau beschreiben, wie der Rechner die gestellte Aufgabe zu lösen hat. Dies sind die Programme in binärer Form.



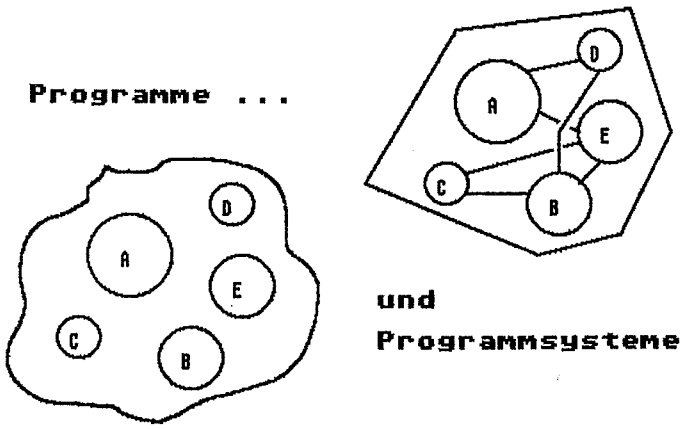
Der Weg des Programmierens

Für einen beträchtlichen - und durchaus mühsamen - Teil des Weges von der Problemstellung hin zur "M-Maschine" stehen dem Programmierer jene mächtigen Vehikel zur Verfügung, an die wir im vorangegangenen Abschnitt erinnert haben: Übersetzer (Compiler) und Interpreter. Sie repräsentieren jeweils eine Maschine (im nebenstehenden Bild "S-Maschine" genannt), welche die Unbilden der "M-Maschine" verbirgt und nun für den Programmierer das eigentliche Ziel vorgibt: einen Text in der *höheren* Sprache S, welcher die S-Maschine darüber instruiert, was sie zu tun hat, um die gegebene Aufgabe zu bewältigen. Doch der Weg hin zu diesem Ziel ist - wie im Bild dargestellt und oft zur nachträglichen Überraschung vieler (auch professioneller) Entwickler von Software - verschlungen genug. Ihn zu begradigen, Wegweiser und Wegmarken zu setzen, Material zur Überbrückung sumpfigen Geländes zu liefern: Dies sind die wesentlichen Anliegen dieses Buches.

Wir kommen dabei nicht umhin, uns immer wieder allgemeine Charakteristika der jeweiligen, von verschiedenen Ausgangspunkten zu erreichenden Ziele zu vergegenwärtigen. Was, so lautet die allem übergeordnete Frage, verlangen wir von den Programmtexten, unseren Endprodukten? Natürlich, daß sie den Rechner zu dem gewünschten, der Aufgabenstellung exakt entsprechenden Verhalten bringen! Dies ist offenbar die Hauptforderung. Doch um sie zu erfüllen, bedarf es zunächst einmal einer möglichst präzisen Beschreibung, einer *Spezifikation* des Verhaltens. Gäbe es sie nicht, so hätte man überhaupt keine Möglichkeit (außer vielleicht einer mehr oder weniger vagen Intuition), die *Korrektheit* des Endprodukts zu überprüfen (zu *verifizieren*). *Korrektheit* ist aber offenbar die *Qualität* eines Programms, auf die nicht verzichtet werden kann.

Das Stichwort ist gefallen: *Qualität*! Qualität im Sinne von "Güte" muß der all unseren Zielen gemeinsame Begriff sein. Und wir behaupten, daß die Qualität unserer Endprodukte entscheidend vom Prozeß ihrer Fertigung abhängt, von der Qualität des Weges also, auf dem wir sie erreichen.

An dieser Stelle ist es notwendig, unsere Zielbestimmung ein wenig zu verfeinern. Die Menge der potentiellen Endprodukte ist doch zu heterogen, als daß wir alles unter einen einzigen "methodischen Hut" bringen könnten. Es sei hier nochmals auf die zu Beginn des Kapitels präsentierte Liste von Dienstleistungen eines Rechners verwiesen: Zwar erkennen wir darunter Aufgaben, die aus dem Übungsbetrieb eines Programmierkurses stammen könnten, doch die meisten Beispiele beinhalten viel mehr als eine kleine Programmieraufgabe. Ja, es handelt sich in den meisten Fällen um recht komplexe Anwendungen, die einer sorgfältigen Analyse bedürfen und sich mit Sicherheit nicht durch ein "monolithisches" Programm erledigen lassen.



Im allgemeinen gibt jede einzelne von ihnen Anlaß zu einer Vielzahl von Programmen und Programmteilen, die miteinander in mannigfachen Beziehungen stehen. Wir nennen solche, nach allerlei Gesichtspunkten strukturierten Gebilde *Programmsysteme*. Die Konstruktion solcher Systeme, ausgehend von einer gründlichen Analyse der Anwendung, wird üblicherweise mit dem in ([DEK]) geprägten Begriff *Programmierung im Großen* (engl.: *Programming in the Large*) bedacht. Es geht bei ihr um die Definition von Software-Komponenten (für die wir später auch das Wort *Modul* gebrauchen werden) und deren Zusammenspiel. Sie ist mit der Arbeit des Architekten eines Hauses vergleichbar, der den Grundriß festlegt, aber die Ausführung der Einzelheiten einer Baufirma überläßt.

Die "Ausführung der Einzelheiten" entspricht (um die Analogie zu wahren) der *Programmierung im Kleinen* (engl.: *Programming in the Small*). Ihre typischen Gegenstände sind der einzelne Modul und die einzelne Prozedur innerhalb eines Moduls. Sie sind gemäß Vorgaben anzufertigen, die durch das übergeordnete System bestimmt sind. Diese Arbeit schließt im allgemeinen auch die möglichst geschickte Organisation der eventuell zu manipulierenden Daten ein.

Der Inhalt dieses Buches orientiert sich an der soeben skizzierten, recht natürlich erscheinenden Dichotomie. Kapitel 2 bis 5 sind der *Programmierung im Kleinen* vorbehalten, während sich Kapitel 6, welches etwa den gleichen Umfang hat wie seine vier Vorgänger zusammen, mit der *Programmierung im Großen* beschäftigt. "Qualität", der entscheidende Aktivposten unserer Endprodukte, hat in diesen verschiedenen Dimensionen der Programmierung durchaus verschiedene Aspekte. Dabei ist Qualität "im Kleinen" eine notwendige Voraussetzung für Qualität "im Großen". Beiden - und damit einer noch genaueren Zielbestim-

mung - sind mehrere Abschnitte in den jeweiligen Kapiteln gewidmet. Wir haben uns dabei bemüht, oft schlagwortartig in die Debatte geworfene Begriffe durch möglichst anschauliche und vertraute Beispiele zu konkretisieren.

Alle übrigen Kapitel und Abschnitte des Buches sollen Wege weisen oder bauen helfen zu qualitativ hochwertigen Programmen und Programmsystemen. Dies reicht von quasi rezeptartigen (und oft *cum grano salis* anzunehmenden) Ratschlägen zum Stil des *Programmierens im Kleinen* (in Kapitel 2) bis hin zur Darlegung von Techniken des Systementwurfs und der Systemimplementierung (in Abschnitt 6.4). Dem aufmerksamen Leser wird nicht entgangen sein, daß wir im ersten Satz dieses Abschnitts von *Methoden* sprachen und nicht von den Methoden. In der Tat: Letzteres bieten zu wollen, wäre ohne Zweifel ein vermessener Anspruch. Die Auswahl der Methoden sowohl für die *Programmierung im Kleinen* ("Schrittweise Verfeinerung" - Kapitel 3, "Programmierung durch Beweisen" - Kapitel 4, "Datenstrukturierter Programm-Entwurf" - Kapitel 5) als auch für die *Programmierung im Großen* (z.B. "Jackson System Development" - Abschnitt 6.3.3, "Abstrakte Datentypen" - Abschnitt 6.4.3, "Objekte, Klassen und Vererbung" - Abschnitt 6.4.4) ist daher, wie man großzügig nachsehen wird, weitgehend durch die Vorlieben und Interessen des Autors bestimmt. Andererseits glauben wir durchaus, eine gute Auswahl getroffen zu haben, sowohl in didaktischer Hinsicht als auch was die Relevanz für eine mit Sinn für solide Technik betriebene Programmierpraxis angeht. (Dies schließt natürlich nicht aus, daß der "gestandene Praktiker" höchstwahrscheinlich "seine" Lieblingsmethode schmerzlich vermissen wird.) Sie ist in hohem Maße von zwei Leitmotiven bestimmt, von denen das erste bereits im ersten Satz dieses einführenden Kapitels anklang: der Rechner ist ein Instrument, welches durch entsprechende Programmierung zu den unterschiedlichsten *Dienstleistungen* befähigt werden kann. Dieses Motiv wird in Abschnitt 6.4 ("Systementwurf und Systemimplementierung") zur vollen Entfaltung gebracht werden. Das zweite Leitmotiv ist durch den Begriff *Prozeß* charakterisiert. Prozesse im Rechner (d.h. das was darin, gesteuert durch ein Programm, vor sich geht) müssen mit Prozessen in der Außenwelt (der *realen Welt*, wie wir auch sagen werden) in enger Verbindung stehen, wenn sie nur irgendwie von Relevanz sein sollen. Dieses Thema wird zum erstenmal gegen Ende von Kapitel 5 in den Vordergrund treten, um dann in Abschnitt 6.3 ("Systemanalyse und Systemspezifikation") (fast) unsere ganze Aufmerksamkeit zu erhalten.

Eine Bemerkung sind wir denjenigen Anhängern der sogenannten *objektorientierten Programmierung* schuldig, die für sich ein eigenes, vom *imperativen* verschiedenes Paradigma reklamieren. Wir glaubten "Objekte, Klassen und Vererbung" in unseren Methodenkanon aufnehmen zu müssen, weil wir der Meinung sind, daß mit der *objektorientierten* Programmierung gleichsam die (bisher) höchste Stufe der Realisierung des *imperativen* Paradigmas erreicht ist. *Objekte* im Sinne des Abschnitts 6.4.4 sind "Befehlsempfänger" par excellence!

Auf eine weitergehende Einstimmung in den Stoff des Buches können wir hier verzichten, da jedes Kapitel mit einer Vorrede beginnt, welche die notwendigen Perspektiven zur Einordnung der jeweils behandelten Themen eröffnet.

Literatur zu Kapitel 1

[CLM] Clocksin W.F. und C.S. Mellish: Programming in Prolog; Springer Verlag; Berlin, Heidelberg, New York; 1981

[DEK] DeRemer, F. and H. H. Kron: Programming in the small versus programming in the large; IEEE Transactions on Software Engineering, Vol. SE-2, No. 2, pp. 80-86; Juni 1976

[GOE] von Goethe, J. W.: Der Zauberlehrling; in: Goethes Gedichte in zeitlicher Folge; Insel Verlag; Frankfurt am Main; 1982

[HAS] Haase, V. und W. Stucky: BASIC, Programmieren für Anfänger; Bibliographisches Institut; Mannheim; 1977

[HER] Hermes, H.: Aufzählbarkeit, Entscheidbarkeit, Berechenbarkeit - Einführung in die Theorie der Rekursiven Funktionen, 3. Aufl.; Springer Verlag; Berlin, Heidelberg, New York; 1978

[JEW] Jensen, K. und N. Wirth: PASCAL, User Manual and Report, 2nd Ed.; Springer Verlag; New York, Heidelberg, Berlin; 1987

[KER] Kernighan, B.W. und D. M. Ritchie: The C Programming Language; Prentice-Hall; Englewood Cliffs; 1978

[LED] Ledgard, H.: ADA, an Introduction; Springer Verlag; New York, Heidelberg, Berlin; 1980

[MCC] McCarthy, J. et al: The LISP 1.5 Programmer's Manual; MIT Press; Cambridge (Mass.); 1962

[MIC] Mickel, K.-P.: Einführung in die Programmiersprache COBOL; Bibliographisches Institut; Mannheim; 1980

[SPR] Spieß, W.E. und F.G. Rheingans: Einführung in das Programmieren in FORTRAN; Walter de Gruyter; Berlin; 1977

[WEI] Weizenbaum, J.: Computer Power and Human Reason, from Judgement to Calculation; Freeman; San Francisco; 1976 (deutsch: Die Macht der Computer und die Ohnmacht der Vernunft; Suhrkamp Verlag; Frankfurt am Main, 1977)

[WI1] Wirth, N.: Programming in Modula-2; Springer Verlag; Berlin, Heidelberg, New York; 1988

2 Qualität und Stil "im Kleinen"

Dieses Kapitel handelt von der Kleinarbeit des Software-Ingenieurs, dem eigentlichen Schreiben einzelner Programme, die etwa im Rahmen eines Projektes als Teile eines Software-Systems spezifiziert wurden; es handelt von den Ansprüchen, die an diese Arbeit (bzw. deren Resultat) zu stellen sind, und von den Regeln, deren Beachtung die Erfüllung dieser Ansprüche erleichtert.

Natürlich sind diese Ansprüche und Regeln ganz generell auf die Arbeit des "einzelnen Programmierers" - ob er nun in einem größeren Team tätig ist oder ob er eine selbstgestellte Aufgabe ohne jede Zusammenarbeit mit anderen zu bewältigen hat - anwendbar. Insofern ist dieses Kapitel, wie im übrigen die folgenden drei Kapitel auch, Teil einer allgemeinen Methodik der "Programmierung im Kleinen" .

In der Einleitung haben wir bereits darauf hingewiesen , daß - wie sollte es anders sein - die Qualität eines Produkts von der Qualität seiner Details abhängt. Auf der Ebene der Details aber stehen in jedem großen Software-System die einzelnen Programme und Prozeduren, die jeweils anzufertigen einen Aufwand in der Größenordnung von vielleicht Stunden oder Tagen erfordert. Diese elementaren Bestandteile haben wir also im Sinn, wenn wir von Qualität und Stil "im Kleinen" sprechen. Wir fragen daher:

> "Welchen Qualitäts-Anforderungen muß ein Programm genügen, das in größerem Zusammenhang benötigt wird?"
>
> Und:
>
> "Welchen 'Stils' muß sich der Programmierer befleißigen, um die Qualität seines Produkts zu fördern?"

Diese beiden Fragen stehen in diesem Kapitel im Vordergrund des Interesses. (Was freilich nicht bedeutet, daß sie in den übrigen Kapiteln dieses Buches ignoriert werden. Vielmehr werden sie dort zumindest implizit ebenfalls eine wichtige Rolle spielen.)

Regeln guten Programmierstils sind nicht ganz unabhängig von der verwendeten Sprache. Da wir uns, wie im Vorwort begründet, für MODULA-2 als diejenige Sprache entschieden haben, in der wir unsere Beispiele formulieren, werden wir solche Regeln für diese Sprache erläutern.

Zur besseren Unterscheidung der hier zur Debatte stehenden Qualitätsanforderungen reden wir von "Programm-Qualität" im Gegensatz zu "Software-Qualität" im allgemeinen, ein Thema, auf das wir im Rahmen von Kapitel 6 zurückkommen werden.

2.1 Programm-Qualität

Die Qualität eines Programms der Art, die wir in der Vorrede zu diesem Kapitel skizziert haben, läßt sich an der eigentlich ganz selbstverständlichen Forderung nach seiner

BRAUCHBARKEIT

messen. Diese Forderung hat mehrere Aspekte:

a) Das Programm soll genau das tun, was es tun soll, nicht mehr also und auch nicht weniger. Diese Eigenschaft wird üblicherweise als

KORREKTHEIT

bezeichnet.

b) Korrektheit ist eine notwendige Voraussetzung dafür, daß ein Programm genutzt werden kann. Sie ist aber keineswegs hinreichend, wenn das Programm in eine bestehende (oder parallel entstehende) Umgebung (ein Programm-System!) integriert werden muß. In diesem Fall müssen die Modalitäten seiner Benutzung - oder, wie man auch sagt, seine "Schnittstellen" - exakt dokumentiert sein. Wir sprechen daher in diesem Zusammenhang von

INTEGRIERBARKEIT.

Dagegen abzugrenzen ist die "Benutzungsfreundlichkeit", die allgemein von Software zu verlangen ist, welche einen "Mensch-Maschine-Dialog" vermittelt. Dieser Begriff wird uns in Kapitel 6 bei der Diskussion weiterer Qualitätsmerkmale wiederbegegnen.

c) Software-Systeme sind in den seltensten Fällen statisch. (Mehr darüber in Kapitel 6.) Sie verändern sich, sei es weil Korrekturen anzubringen sind oder sei es wegen allfälliger Anpassungen, die aus verschiedenen Gründen notwendig werden können. Änderungen ergeben sich nicht nur im Zusammenspiel der einzelnen Teile, sondern auch in den Teilen, den einzelnen Programmen selbst. Deren "Brauchbarkeit" hat also auch eine zeitliche Dimension, die wir als

WARTBARKEIT

bezeichnen.

d) Schließlich wird von einem Programm häufig erwartet, daß es mit den Ressourcen des Rechners sparsam umgeht. Die wichtigsten dieser Ressourcen sind der (bzw. die) Prozessor(en) und die Speicher (sowohl intern als auch extern). Sparsamer Umgang mit einem Prozessor bedeutet, diesen nur für möglichst kurze Zeit zu benutzen. Sparsame Verwendung der Speicher bedeutet, mit möglichst wenig Platz für zu manipulierende Datenobjekte und für das Programm selbst auszukommen. Wir bezeichnen diese Eigenschaften insgesamt mit dem Begriff

EFFIZIENZ

und unterscheiden *Zeit*-Effizienz und *Speicher*-Effizienz. Zeit-Effizienz kann zum Beispiel dann von Interesse sein, wenn - etwa bei einem Mehr-Programm-Betrieb - die Prozessoren nicht nur für ein einziges Programm arbeiten müssen. Aber auch wenn eine schnelle Reaktion des Rechners (im "Mensch-Maschine-Dialog" oder bei der Steuerung eines technischen Prozesses) gefordert ist, gewinnt diese Eigenschaft an Bedeutung. Speicher-Effizienz war früher, als diese Ressource noch sehr teuer war, ein wichtiges Qualitätsmerkmal.

Wir werden feststellen, daß ein unbedachtes, "blindes" Streben nach Effizienz in Bezug auf die bereits genannten anderen Qualitätsmerkmale häufig "kontraproduktiv" ist: es verhindert, daß diese erreicht werden.

Die genannten vier Merkmale bzw. Eigenschaften wollen wir nun etwas eingehender kommentieren und durch einige Beispiele illustrieren.

2.1.1 Korrektheit

Betrachten wir die folgenden kurzen Programme, die offensichtlich drei wichtigen Anwendungsbereichen entstammen könnten: das erste dem "technisch-wissenschaftlichen", das zweite dem "Prozess-technischen" und das dritte dem "betrieblich-kaufmännischen".

Beispiel 1:

```
MODULE IchTueWasIchSoll;
FROM Inout IMPORT ReadInt, WriteInt, WriteString;
VAR a,b: INTEGER;
BEGIN
  ReadInt(a); ReadInt(b);
  WHILE a # b DO
    IF a < b THEN
      b:=b-a
    ELSE
      a:=a-b
    END (*IF*)
  END (*WHILE*);
  WriteString("Ergebnis:");
  WriteInt(a,5)
END IchTueWasIchSoll.
```

Beispiel 2:

```
MODULE TutAuchWasErSoll;
FROM ProcessInOut IMPORT  SensorTyp, SchalterTyp,
        SchalterPositionsTyp, getSensor, putSchalter;
(*Diese Import-Liste ist - zu Illustrationszwecken - für dieses Pro-
gramm frei erfunden und gehört nicht zum Standard-Umfang einer
```

```
MODULA-2 - Implementierung!!*)
CONST MinKritC = 0.5;
      MaxKritC = 3.5;
VAR messWert: REAL;
    sensor: SensorTyp;
    schalter: SchalterTyp;
BEGIN
  LOOP
    getSensor(sensor,messwert);
    IF (messwert <= minKrit) THEN
      putSchalter(schalter,ein)
    ELSIF (messwert >= maxKrit) THEN
      putSchalter(schalter,aus)
    END (*IF*)
  END (*LOOP*)
END TutAuchWasErSoll.
```

Beispiel 3:

```
MODULE BinAuchOK;
FROM FileInOut IMPORT FileTyp, OpenRead, OpenWrite,
      CloseFile, ReadFile, WriteFile, Eof;
(*Diese Import-Liste ist - zu Illustrationszwecken - für dieses Pro-
gramm frei erfunden und gehört nicht zum Standard-Umfang einer
MODULA-2 - Implementierung!!*)
TYPE ZahlungsSatzTyp = RECORD
                         kontoNr: CARDINAL;
                         betrag: INTEGER
                       END;
   ListenSatzTyp = RECORD
                     kontoNr: CARDINAL;
                     summe: INTEGER
                   END;
VAR eingang: ZahlungsSatzTyp;
  eintrag: ListenSatzTyp;
  zahlungen: FileTyp;
  liste: FileTyp;
  aktNr: CARDINAL;
  kSumme: INTEGER;
BEGIN
  OpenRead(zahlungen); OpenWrite(liste);
  WITH eingang DO
    ReadFile(zahlungen,kontoNr,betrag)
  END (*WITH*);
```

```
    WHILE NOT(Eof(zahlungen)) DO
      aktNr:=eingang.kontoNr;
      kSumme:=0;
      WHILE (eingang.kontoNr=aktNr) AND
            NOT(Eof(zahlungen)) DO
        kSumme:=kSumme + eingang.betrag;
        WITH eingang DO
          ReadFile(zahlungen,kontoNr,betrag)
        END (*WITH*)
      END (*WHILE*);
      WITH eintrag DO
        kontoNr:=aktNr;
        summe:=kSumme;
        WriteFile(liste,kontoNr,summe)
      END (*WITH*)
    END (*WHILE*);
    CloseFile(zahlungen); CloseFile(liste)
  END BinAuchOK.
```

Frage: "Sind diese Programme korrekt?" Syntaktisch scheint alles in Ordnung zu sein, und die Inspektion der einzelnen Programmtexte dürfte ohne große Mühe erkennen lassen, daß jeweils sinnvolle Ergebnisse (Ausgaben) produziert werden. Aber: Sind dies die Ergebnisse, die tatsächlich gewünscht wurden?

Diese Frage ist mit der Kenntnis des Programmtextes allein offensichtlich nicht zu beantworten. Was fehlt, ist eine Beschreibung der gewünschten Ergebnisse, ihre "Spezifikation". Nun könnte man sich auf den Standpunkt stellen, daß die Programme selbst ja ihre Ergebnisse definieren! Damit freilich würde sich das Korrektheitsproblem ins Nichts auflösen, denn jedes Programm wäre dann trivialerweise korrekt, und die völlige Anarchie wäre ausgebrochen. Dies kann sicherlich nicht die Absicht sein. Die Spezifikation, die wir verlangen, muß eine Beschreibung der Ergebnisse sein, aus der das "WIE" der Produktion dieser Ergebnisse nicht unmittelbar hervorgeht. Für unsere drei Beispiele könnten diese Spezifikationen etwa wie folgt ausschauen:

> Beispiel 1: Der Rechner soll zu zwei ganzen Zahlen deren größten gemeinsamen Teiler (ggT) ermitteln.
>
> Beispiel 2: Der Rechner soll regelmäßig den Druck eines Gefäßes messen und bei der Über- bzw. Unterschreitung von Grenzwerten eine Pumpe aus- bzw. einschalten.
>
> Beispiel 3: Der Rechner soll ein Band lesen, auf dem Kontobewegungen gespeichert sind. Für jede Kontonummer soll er die Summe aller Kontobewe-

gungen auf eine Liste (einen "Kontobewegungsbericht") schreiben.

Wie kann ein Programmierer nun glaubhaft machen, daß das Programm, das er geschrieben hat, die Spezifikation erfüllt? Diese Fragestellung charakterisiert das "Korrektheitsproblem". Mit einer - wie oben - rein verbal (umgangssprachlich) gegebenen Spezifikation als Grundlage wird er auch nur verbal und allenfalls durch Demonstration argumentieren können. In vielen Fällen mag dies durchaus genügen. Aber schon die Demonstration macht Schwierigkeiten. Um damit einen Auftraggeber von der Korrektheit eines Programms zu überzeugen, sollten von diesem Daten vorgegeben werden, deren gewünschtes Verarbeitungsergebnis er schon im Voraus kennt. Solche zusammengehörigen "Ein- und Ausgabe-Daten" sollten also bereits Bestandteil der Spezifikation sein.

Für das erste Beispiel ist es gewiß nicht schwer, auch ohne die Kenntnis eines allgemeinen Verfahrens Zahlentripel (a,b,c) zu finden, mit c = ggT(a,b). Auch Beispiel 3 ließe sich so erledigen: Der Auftraggeber könnte ein Band zur Verfügung stellen, welches Daten enthält, für die früher auf andere Weise (z.B. per Hand) der Bericht erstellt wurde. Das Programm muß dann mit diesen gegebenen Daten den bekannten Bericht erzeugen.

Das so beschriebene Überprüfungsverfahren heißt *Testen* mit *Testdaten.* Die Testdaten werden als Teil der Spezifikation betrachtet.

Auf Beispiel 2 allerdings ist dieses Verfahren so ohne weiteres nicht anwendbar. Es müßte nämlich ein zeitlicher Verlauf des Gefäßdrucks in Abhängigkeit von Umwelteinflüßen und den jeweiligen Stellungen jenes Schalters bereits bekannt sein, bevor das Programm zum ersten Mal gelaufen ist. Und wenn es gelänge - mit welchen aufwendigen Vorrichtungen auch immer - eine solche "Referenzkurve" für den Gefäßdruck zu ermitteln, welches Vertrauen könnte man dann in einen Test haben, der diese Referenzkurve hinreichend genau reproduziert? (Und was hieße "hinreichend"?) Die Behauptung ist, daß dieses Vertrauen nicht sehr groß sein kann: Was garantiert, daß unter anderen Umweltbedingungen das Programm nicht total versagt?

Ein Verfahren, um im Falle von Beispiel 2 den Grad der Überzeugtheit von der "Korrektheit" des Programms zu erhöhen, wäre "Test durch *Simulation*". Dies bedeutet, daß dem Programm an den von ihm bedienten Ein- und Ausgängen die Umgebung, in der es eigentlich arbeiten soll, nur vorgespielt wird, etwa von einem anderen Programm, welches das Verhalten eines Druckgefäßes unter verschiedenen Umweltbedingungen imitieren oder eben "*Simulieren*" kann. (Auch für dieses Programm müßte freilich ein Korrektheitsnachweis geführt werden!)

Es zeigt sich also, daß das Korrektheitsproblem für Programme der mit Beispiel 2 gezeigten Art besonders hart, ja unter Umständen sogar unlösbar ist. Typisch für diese Programme ist, daß sie auf nicht vorhersehbare Veränderun-

gen im Laufe eines technischen (z.B. Kraftwerk) oder natürlichen (z.B. Herztätigkeit) Prozesses reagieren müssen, und dies oftmals in "zeitkritischer" Weise. Das heißt, daß ein solches Programm schon dann als nicht korrekt angesehen werden muß, wenn es nicht - wie etwa gefordert - in 100 msec, sondern erst in 200 msec "antwortet".

Die Vorstellungen, die wir bisher zum Begriff "Korrektheit" entwickelt haben, sind im Grunde noch sehr vage. Sie entsprechen in keiner Weise möglichen Anforderungen an Strenge und Exaktheit, die etwa dann zu stellen sind, wenn von der Korrektheit einzelner Programme die Sicherheit und Zuverlässigkeit umfangreicher Programmsysteme abhängen. In diesem Falle sind die oben gegebenen Spezifikationen gewiß ungenügend, und auch Tests, sei es mit Testdaten oder durch Simulation, sind nicht dazu geeignet, die Fehlerfreiheit eines Programms zu beweisen. Sie können das Vertrauen in die Korrektheit eines Programms lediglich verstärken, indem sie zeigen, daß für bestimmte Eingaben die erwarteten Ausgaben erzeugt werden. Sie schließen im allgemeinen nicht aus, daß für andere Eingaben kein korrektes Ergebnis geliefert wird. (Dies ist umso gravierender, als man dann das korrekte Ergebnis noch nicht einmal kennt!)

Gefragt sind also *formale Spezifikationen* und die Möglichkeit, die Korrektheit eines Programms bezüglich einer formalen Spezifikation *formal* zu *beweisen.*

Eine formale Spezifikation für das Programm "Beispiel 1" könnte etwa folgendermaßen ausschauen:

GEGEBEN: $a,b \in \mathbb{Z},\ a \neq 0 \wedge b \neq 0$

GESUCHT: $t \in \mathbb{N},\ t > 0,$

$t \mid a \wedge t \mid b \wedge$

$(s \in \mathbb{N}, s > t \Rightarrow s \nmid a \wedge s \nmid b),$

wobei: $t \mid a \Leftrightarrow \exists n \in \mathbb{Z}, a = n*t,$

$t \nmid a \Leftrightarrow \text{non}(t \mid a).$

Dies ist eine Beschreibung des ggT von zwei ganzen Zahlen mit Hilfe eines logischen Formalismus. Dabei haben wir noch längst nicht "alle Register der Formalität" gezogen.

Man kann sich daher leicht ausmalen, welche Mühe man mit der Formulierung entsprechender logischer Ausdrücke für die Spezifikation der Wirkung von Programmen wie "Beispiel 2" und "Beispiel 3" hätte. Da diese Beispiele für große Klassen praktischer Aufgaben stehen, wird sofort klar, daß *Programmverifikation* - wie man den Korrektheitsnachweis durch formales Beweisen auch nennt - von sehr wenigen Ausnahmen abgesehen (s.o.) als (industriell bzw. kommerziell) praktikables Verfahren ausscheidet.

Dennoch ist Programmverifikation als theoretisches Hilfsmittel zur Einübung in strenge Programmier-Disziplin (und darum geht es uns ja nicht zuletzt!) so un-

entbehrlich, daß wir ihr das gesamte Kapitel 4 widmen. Dort werden wir zeigen, wie man, ausgehend von prädikatenlogischen Ausdrücken, kritische Programmstücke "Hand in Hand" mit einem Beweis ihrer Korrektheit entwickeln kann. Eine Möglichkeit, auf der Basis "halbformaler" Spezifikationen gewissermaßen per Konstruktion korrekte Programme zu erzeugen, diskutieren wir dann in Kapitel 5.

Natürlich garantiert auch die "formalste" Spezifikation nicht, daß sie selbst korrekt ist, das heißt, daß sie den tatsächlich mit dem Programm verfolgten Intentionen entspricht. Dies ist offensichtlich ein Problem, welches mit der Herleitung von Programm-Spezifikationen aus praktischen Problemstellungen zu tun hat. Die Ansätze hierfür werden unter dem Oberbegriff *Anforderungs-Definition* (englisch: *Requirements-Engineering*) zusammengefaßt. Mehr zu diesem Themenkomplex (wenn auch nicht die "ultima ratio") wird der Leser in Kapitel 6 ("Systemanalyse und Systemspezifikation") dieses Buches erfahren.

2.1.2 Integrierbarkeit

Integrierbarkeit ist - wie bereits angedeutet - ein Qualitätsmerkmal, das weitgehend durch die *Dokumentation* der Art und Weise der Verwendung eines Programms bestimmt ist. Es gewinnt für das Software Engineering seine Bedeutung dadurch, daß man im allgemeinen nicht davon ausgehen kann, daß der Ersteller eines Programms auch derjenige ist, der das Programm - als "Baustein" - weiterverwendet. Die Informationen über seinen "Einbau" müssen also hinreichend klar formuliert sein. Wir werden später (in Kapitel 6) sehen, daß MODULA-2 hierfür besonders elegante Möglichkeiten anbietet. Mit den uns derzeit zur Verfügung stehenden Kenntnissen wollen wir uns an dieser Stelle auf das folgende einfache Beispiel beschränken.

Funktionen des Ein-/Ausgabe-Systems eines Rechners mögen für MODULA-2 - Programme durch den Aufruf von parametrierten Prozeduren ausgelöst werden. Unter diesen Prozeduren ist etwa:

```
PROCEDURE LiesPlatte(sektor,spur: CARDINAL;
                     VAR inhalt: PBereichsTyp;
                     VAR status: CARDINAL);
```

Aus der Dokumentation müssen ersichtlich werden:

- Zweck der Prozedur (hier: direktes Lesen eines per Sektor und Spur adressierbaren Plattenbereichs);
- möglicher Kontext (hier: ein MODULA-2 - Programm, welches Platten-E/A "auf sehr niedriger Ebene" benötigt);
- Voraussetzungen für den Einsatz (hier: die Typbezeichnung "PBereichsTyp" muß bekannt sein);

- Bedeutung der Parameter und ihre zulässigen Werte - soweit nicht bereits durch die Typ-Bezeichner bzw. -Beschreibungen festgelegt. (Im Beispiel legt der Typ-Bezeichner "CARDINAL" die zulässigen Werte keineswegs fest, da für "sektor" oder "spur" der Wert 4294967295 (=2^32-1, der maximale Wert von "CARDINAL" bei einem 32-Bit-Rechner) wohl kaum in Frage kommt.) Insbesondere müssen die möglichen Fehler, die von der Prozedur zurückgemeldet werden können, genauestens aufgeführt sein (z.B. der, daß für "spur" oder "sektor" unzulässige Werte übergeben wurden).

Das Qualitätsmerkmal "Integrierbarkeit" hat ein sehr hohes Gewicht für Programme und Sammlungen von Programmen ("Bibliotheken"), die als Grundlage für viele verschiedene Anwendungen dienen können. (Für mehr und Genaueres zu diesem Thema sei der Leser auf Abschnitt 6.4.4 vertröstet.)

2.1.3 Wartbarkeit

Während die "Integrierbarkeit" bzw. "Benutzbarkeit" eines Programms von seiner "externen Dokumentation" abhängt, kann der Begriff "Wartbarkeit" unmittelbar als "gute Qualität der Programm-internen Dokumentation" übersetzt werden. Generell gilt, daß ein Programm so verfaßt sein muß, daß es in

AUFBAU und FUNKTION

auch von anderen als dem jeweiligen Verfasser ohne unzumutbaren Aufwand verstanden werden kann. "Unzumutbarkeit" ist dabei selbstverständlich zu relativieren. Sicherlich kann von der Programm-internen Dokumentation nicht verlangt werden, daß sie Lehrbuchcharakter erhält. Dies würde einen unzumutbaren Aufwand bei der Programm-Erstellung selbst bedeuten. Und schließlich kann von den mit Wartungsarbeiten Betrauten vorausgesetzt werden, daß sie eine einschlägige Ausbildung absolviert haben. Dennoch: Jeder Informatiker weiß, wie mühsam es oft ist, ein von einem Kollegen geschriebenes Programm zu verstehen. Im Rahmen großer Projekte aber kann die "Verstehbarkeit" einzelner Programme ein gewichtiger Kostenfaktor sein.

Was ist nun "Programm-interne Dokumentation"? Sie entsteht sowohl durch "stilvollen" Gebrauch der Programmiersprache als auch durch die explizite Kommentierung eines Programms als Ganzem, von Programm-Teilen und einzelnen Aktionen. In Abschnitt 2.2 werden wir versuchen, in Regeln zusammenzufassen, was wir unter "stilvollem" Gebrauch verstehen. Ein (wiederum sehr einfaches) Beispiel mag unsere Behauptung vorab verdeutlichen:

```
MODULE IchTueWasIchSoll; (*Dies ist eine Modifikation von
                           Beispiel 1 aus Abschnitt 2.1.1*)
FROM InOut IMPORT ReadInt, WriteInt, WriteString;
VAR a,b: INTEGER;
BEGIN ReadInt(a);ReadInt(b);
```

```
WHILE a#b DO IF a<b THEN b:=b-a
ELSE a:=a-b END;END;WriteString("Ergebnis:");WriteInt(a,5)
END IchTueWasIchSoll.
```

Obwohl sich diese Version des "Beispiels 1" - von der Kommentierung abgesehen - syntaktisch von der ursprünglichen Version nicht unterscheidet, ist doch die herbe Kritik naheliegend, welche hier zu üben ist. Und zwar in drei Punkten:

- Der Programm-Name ist völlig nichtssagend.
- Der Programmtext hat keine klare, augenfällige Struktur.
- Es fehlt jegliche Kommentierung zu dem "Was und Warum" der vom Programm auszulösenden Aktionen.

(Dabei ist die obige Version keineswegs übertrieben. In den heutzutage so beliebten Zeitschriften für den "Mikro-Computer-Freak" hat man schon Schlimmeres veröffentlicht gesehen!)

Der erste und letzte Punkt dieser Kritik trifft im übrigen auch auf die ursprüngliche Version zu. Eine Verbesserung ergäbe sich hier also durch bloße Änderung des Namens und durch Einfügen eines Kommentars, der das Verfahren erklärt, welches dem Programm zugrunde liegt:

```
MODULE GGT; (*verbesserte Version von Beispiel 1*)
(* Berechnung des groessten gemeinsamen Teilers (ggT) zweier
positiver ganzer Zahlen. Diese werden per Tastatur-Eingabe den
Variablen a und b zugewiesen. Die Berechnung macht von
folgenden Eigenschaften des ggT Gebrauch:
(1) ggT(a,b) = ggT(a,b-a),   falls b > a > 0,
(2) ggT(a,b) = ggT(a-b,b),   falls a > b > 0,
(3) ggT(a,b) = a,            falls a = b und a # 0 *)
FROM Inout IMPORT ReadInt, WriteInt, WriteString;
VAR a,b: INTEGER;
BEGIN
  ReadInt(a); ReadInt(b);
  WHILE a # b DO
    IF a < b THEN
      b:=b-a
    ELSE
      a:=a-b
    END (*IF*)
  END (*WHILE*);
  WriteString("Ergebnis:");
  WriteInt(a,5)
END GGT.
```

Einem Programmierer, dem die Aufgabe gestellt wurde, dieses Programm zu modifizieren, weil beispielsweise bemerkt wurde, daß es auch die Eingabe negativer ganzer Zahlen zuläßt (und dann eine verheerende Reaktion zeigt, nämlich gar keine!), wird es nun leicht fallen, die entsprechende Änderung anzubringen und auch den einleitenden Kommentar zu aktualisieren.

Interne Dokumentation kommt also ganz wesentlich durch Ausnutzung der die "Selbstdokumentation" unterstützenden Elemente bzw. Eigenschaften einer Programmiersprache zustande. Für MODULA-2 (wie im übrigen auch z.B. für PASCAL) sind dies unter anderem:

- keine Vorschriften zur Formatierung des Programmtextes,
- Bezeichner beliebiger Länge sind erlaubt,
- Typen sind frei definierbar, usw.

(Darüber hinaus hat MODULA-2 weitere Selbstdokumentations-Eigenschaften, die sich durch die später - in Kapitel 6 - verwendeten Sprachelemente ergeben.)

Wie solche Möglichkeiten ausgenutzt werden, sollte im Rahmen eines Projekts (oder besser noch: für die gesamte mit der Software-Produktion befaßte Organisation, z.B. ein Software-Unternehmen) weitgehend festgelegt ("standardisiert") sein. Dies ist eine wichtige Maßnahme, um den Aufwand zur Einarbeitung in ein zwar fremdes, aber doch aus dem gleichen Projekt (oder Unternehmen) stammendes Programm beträchtlich zu reduzieren.

2.1.4 Effizienz

Wie jedermann weiß, kann z. B. ein mathematisches Problem umständlich oder elegant gelöst werden. Eine umständliche Lösung ist meist schwer zu verstehen, dauert lange und erfordert viel Schreibarbeit. Elegante Lösungen dagegen zeichnen sich durch knappe Formulierungen, Durchsichtigkeit und schnelles Erreichen des Ziels aus.

Mit Programmier-Aufgaben verhält es sich ähnlich. Ein Programm hat - wie wir wissen - die Arbeit eines Rechners zu organisieren. Sind die "Arbeits-Anweisungen", die das Programm gibt, selbst konfus, so wird der Rechner vielleicht sehr viel unnötige Arbeit verrichten müssen, und er wird vielleicht gezwungen, sehr viel mehr Speicherplatz für seine Operationen in Anspruch zu nehmen, als tatsächlich erforderlich wäre. Das in Abschnitt 2.1 beschriebene Ziel *Effizienz* (mit den Teilaspekten "Zeit-Effizienz" und "Speicher-Effizienz") würde nicht erreicht.

Von einem Programm ist also zu verlangen, daß es den Rechner zu effizientem Arbeiten veranlaßt. (Wir sagen dann von dem Programm selbst, daß es effizient sei.) Wie kann es das schaffen? Manche Programmierer glauben, mit der An-

wendung möglichst vieler "Programmiertricks" ihr Programm zu beschleunigen und zu sparsamer Verwendung von Speicherplatz zu bewegen. Gegen solche Programmiertricks werden wir im folgenden Abschnitt 2.2 ausführlich argumentieren.

Tatsächlich ist Effizienz von Programmen keine Frage von Tricks. Nehmen wir an, die Aufgabe verlangt ein Programm, welches eine Menge von Namen (etwa der Mitglieder eines Vereins) verwaltet. Die Operationen, die zu unterstützen sind, betreffen das Hinzufügen, Entfernen und Aufsuchen von Namen. Es kommt nun darauf an, eine dem Rechner gemäße Repräsentation der Namensmenge derart zu finden, daß die genannten Operationen ohne großen Aufwand auszuführen sind. Man sagt auch: Es ist eine effiziente Datenstruktur für die Speicherung von Namensmengen zu entwickeln. (Der Programmierer steht hier also vor einem Problem, das in ähnlicher Weise der Organisator eines Büros mit vielen Akten hat.)

Ein anderes Beispiel: Eine in einer Variablen vom ARRAY-Typ abgespeicherte Menge von Namen soll alphabetisch aufsteigend sortiert werden. Ein naheliegendes, sehr einfaches und keineswegs umständlich zu formulierendes Verfahren ("Algorithmus") kann wie folgt beschrieben werden:

> "VERGLEICHE die Namen der Reihe nach mit allen nachfolgenden Namen.
>
> VERTAUSCHE immer dann, wenn ein nachfolgender Name in der lexikographischen Ordnung 'kleiner' ist."

Beschränkt auf Mengen von einzelnen ASCII-Zeichen lautet die entsprechende Prozedur:

```
PROCEDURE NaivSort(VAR s: ARRAY OF CHAR);
VAR zwischenSpeicher: CHAR;
    i,j: CARDINAL;
BEGIN
  FOR i:=0 TO HIGH(s)-1 DO
    FOR j:=i+1 TO HIGH(s) DO
      IF s[j] < s[i] THEN (*vertausche*)
        zwischenSpeicher:=s[i];
        s[i]:=s[j];
        s[j]:=zwischenSpeicher
      END (*IF*)
    END (*FOR*)
  END (*FOR*)
END NaivSort;
```

Die Anzahl der VERGLEICHE und VERTAUSCHUNGEN, die von dieser Prozedur bewirkt werden, und damit die Laufzeit T der Prozedur, ist - wie man

leicht nachprüft - proportional zum Quadrat der Mächtigkeit der Zeichenmenge. Es gilt also:

$$T(n) = O(n^2), \text{ mit n=Mächtigkeit der Zeichenmenge.}$$

(Wir verwenden hier die aus der Analysis geläufige "O-Notation". "f(x) = O(g(x))" bedeutet, daß "f(x) *asymptotisch proportional* ist zu g(x)".)

Andererseits ist wohlbekannt, daß es Sortierverfahren gibt, für deren Laufzeit - wenn programmiert - gilt:

$$T(n) = O(n*\log n)$$

(z.B. "Quicksort", "Mergesort", "Heapsort", usw.). Auch diese wesentlich effizienteren Verfahren sind nicht umständlich, bedienen sich keiner "Tricks" und sind leicht zu verstehen (siehe z.B. [OTW]).

Daß durch die geschickte Auswahl eines Algorithmus eine enorme Beschleunigung der Programm-Ausführung erreicht werden kann, zeigt die folgende Tabelle, in der Werte von n^2 und n*log(n) (hier der Logarithmus zur Basis e) für einige n gegenübergestellt sind:

n	n^2	n*ln(n)
10	100	23
100	10000	460
1000	1000000	6908

Es sollte nun klar sein, daß die Effizienz eines Programms ganz entscheidend bestimmt ist durch die Datenstrukturen und Algorithmen, welche zur Repräsentation der zu manipulierenden Objekte und zur Ausführung der Operationen selbst ausgewählt werden.

Die Erforschung geeigneter Datenstrukturen und Algorithmen ist nicht Gegenstand einer "Programmier-Methodik", sondern der "Komplexitätstheorie", einer Disziplin, die ihre Motivationen sowohl von der theoretischen als auch von der praktischen Seite der Informatik bezieht.

Datenstrukturen und Algorithmen werden in diesem Buch daher nur eine illustrierende Rolle spielen; zum Beispiel dann, wenn es - wie in Kapitel 6 - darum geht, die Austauschbarkeit verschiedener Implementierungen einer durch eine abstrakte Spezifizikation definierten Funktion zu demonstrieren.

Mit einem eher elementaren (und beinahe trivialen) Effizienz-Aspekt werden wir uns freilich noch in diesem Kapitel beschäftigen. Wir sprachen oben davon, daß die Laufzeit einer Prozedur in Abhängigkeit von der Anzahl n der zu bearbeitenden Objekte von einer bestimmten Größenordnung sei, also "im wesentlichen" proportional dem Wert einer Funktion von n. Die Größe des hier auftretenden Proportionalitätsfaktors hängt nicht zuletzt von der Sorgfalt des Programmierers ab. Und es macht schon etwas aus, ob man 10*n*log(n) msec für eine Operation benötigt oder nur 2*n*log(n) msec! In Abschnitt 2.3 werden wir

daher eine Reihe von Regeln, von "praktischen Tips" sozusagen, angeben, deren Beachtung Programme beschleunigt, ohne dabei ihre Verständlichkeit zu beeinträchtigen. Die jeweils verwendeten Algorithmen bleiben von solchen, die Effizienz verbessernden Maßnahmen völlig unberührt.

2.2 Programmierstil

Für guten Programmier-Stil gibt es eine ganze Reihe von Regeln, deren Beherzigung die Wartbarkeit eines Programms erheblich verbessern kann. Solche Regeln werden im folgenden genannt und diskutiert (einige davon in Anlehnung an [LNH]). Die erste Gruppe von Regeln betrifft Dokumentation und Selbst-Dokumentation, die zweite behandelt Aspekte der "strukturellen Komplexität" von Programmen (nicht zu verwechseln mit dem oben erwähnten Gegenstand der Komplexitätstheorie!), die für die Verstehbarkeit ebenfalls relevant sind.

2.2.1 Dokumentation und Selbst-Dokumentation

REGEL: Programmen bzw. Programm-Teilen mit eigener Funktion (bzw. eigenen Funktionen) ist ein standardisierter PROLOG in Form eines Kommentars voranzustellen.

Der Inhalt eines solchen Prologs hängt von der Art des Programm-Teils ab. Dies gilt insbesondere bei Verwendung von MODULA-2. Hier gibt es ja (vgl. [WI1]) vier verschiedene Arten von Moduln, außerdem (allgemeine) Prozeduren und Funktionsprozeduren wie üblich. Von diesen bilden die "MODULE" genannten Programmteile - mit Ausnahme der "internen Moduln" - sogenannte Compilierungseinheiten, während isolierte Prozeduren und Funktionsprozeduren nicht separat compiliert werden können. Diese Unterscheidung ist wichtig und wird später (in Kapitel 6) noch eine Rolle spielen. An dieser Stelle genügt es, darauf hinzuweisen, daß - etwa bei der Einführung von Standards für ein Projekt oder Unternehmen - Compilierungseinheiten bezüglich der einleitenden Kommentierung sicherlich anders zu behandeln sind als kleinere Einheiten wie Prozeduren und Funktionsprozeduren.

Dies ist eine Liste von Informationselementen, die ein Prolog enthalten könnte bzw. sollte:

- Projekt-Bezeichnung,
- knappe Angabe des Zwecks des Programm(-Teil)s,
- Name des Autors,
- Datum der Fertigstellung,
- Parameter und ihre Verwendung,
- hinreichend präzise Beschreibung der zulässigen Eingaben,
- hinreichend präzise Beschreibung der Ausgaben,
- benötigte Dateien,

- die wichtigsten Datenstrukturen,
- verwendete Algorithmen,
- globale Variablen,
- Auswirkungen des Programm-Teils auf globale Variable (Seiteneffekte),
- benötigte (d.h. aufzurufende, zu benutzende) Programm-Teile,
- Zeitbedingungen,
- Behandlung von Fehlern und unerwünschten Situationen,
- Angaben zu Modifikationen (wer, wann, was, warum)
- usw..

Bei Prozeduren und Funktionsprozeduren kann die Verwendung von Parametern bereits im Prozedur-Kopf durch geeignete Kommentare deutlich sichtbar gemacht werden. Beispiel:

```
PROCEDURE Ersetze(m1,m2: ARRAY OF CHAR;
                  VAR s: ARRAY OF CHAR;
                  VAR erfolg: BOOLEAN);
```

soll in einem mit s übergebenen String das Muster m1 suchen und - falls gefunden - durch m2 ersetzen. In "erfolg" wird mitgeteilt, ob die Ersetzung vorgenommen werden konnte. Hier werden also in m1, m2 und s Datenobjekte an die Prozedur übergeben mit der Aufforderung, das in s befindliche Objekt gemäß m1 und m2 zu aktualisieren (bzw. zu modifizieren). Über "erfolg" hingegen wird ein Datenobjekt von der Prozedur an den benutzenden (d.h. rufenden) Programm-Teil geliefert. Die folgende Ergänzung des Prozedurkopfes macht diese Verwendung der Parameter klar:

```
PROCEDURE Ersetze( (*IN*) m1,m2: ARRAY OF CHAR;
                   (*INOUT*) VAR s: ARRAY OF CHAR;
                   (*OUT*) VAR erfolg: BOOLEAN);
```

Es ist insbesondere wichtig, die unterschiedliche Verwendung von Variablen-Parametern als (*INOUT*) oder nur (*OUT*) zu dokumentieren.

Mit etwas mehr "Geschwätzigkeit" kann aus dem Prozedurkopf fast eine komplette Beschreibung des Zwecks gemacht werden:

```
PROCEDURE Ersetze ( (*im String*) VAR s: ARRAY OF CHAR;
      (*das Muster*)    m1: ARRAY OF CHAR;
      (*durch Muster*) m2: ARRAY OF CHAR;
      (*und melde Erfolg in*) VAR erfolg: BOOLEAN);
```

Im allgemeinen wird man der knapperen Form der Parameter-Kommentierung den Vorzug geben.

REGEL: Die Formatierungsmöglichkeiten, die die Syntax einer Programmiersprache bietet, sind so auszunutzen, daß der Programmtext eine klare, den Aktionen und Aktionsgruppen entsprechende, augenfällige Struktur erhält.

MODULA-2 kennt - wie u.a. PASCAL - kaum Einschränkungen hinsichtlich der visuellen Gestaltung (des "Layouts") eines Programmtextes. Beliebig viele Leerzeichen bzw. Leerzeilen dürfen zwischen Symbolen (vgl. Anhang) eingefügt werden. Damit können:

- einzelne Programm-Abschnitte sichtbar voneinander abgesetzt werden (durch Leerzeilen);
- die hierarchischen Beziehungen zwischen den strukturbildenden Anweisungen (WHILE, REPEAT, LOOP, FOR, IF, CASE, WITH) hervorgehoben werden (durch entsprechende Zeileneinrückungen, auch "Indentierung" genannt).

Man mag diese Regel ob ihrer scheinbaren Belanglosigkeit belächeln. Für den Compiler sind Zahl und Verteilung von Leerzeichen und Leerzeilen schließlich einerlei. Man sollte jedoch nicht vergessen, daß der menschliche Wahrnehmungsapparat Information nur dann effizient aufnehmen kann, wenn ihm diese in geordneter Form dargeboten wird. Und vor dem Verständnis ist immer die Wahrnehmung! Die Bedeutung dieser Regel ist also nicht zu unterschätzen.

Wir wollen den "Indentierungsteil" der Regel durch Angabe präziser Vorschriften noch ein wenig konkretisieren:

(i) Auf gleicher Spalte beginnen:
- Modul- und globale Prozedur-Deklarationen,
- Import-Listen,
- Konstanten-, Typ- und Variablen-Deklarationen,
- BEGIN und END einer Modul- bzw. Prozedur-Deklaration.

Beispiel:

```
PROCEDURE ... ;
CONST ... ;
TYP ... ;
VAR ... ;
BEGIN
. . .
END;
```

(ii) Auf gleicher Spalte beginnen alle Begrenzer strukturbildender Anweisungen, also:

```
WHILE ... DO                    REPEAT
  .                               .
  .                               .
END (*WHILE*);                  UNTIL ... ;
```

LOOP . . END (*LOOP*);	FOR ... ... DO . . END (*FOR*);

IF ... THEN
.
.
ELSIF ... THEN
.
.
.
ELSE
.
.
END (*IF*)
	CASE ... OF
	.
	.
	END (*CASE*);
	
	WITH ... DO
	.
	.
	END (*WITH*);

(iii) Eingerückt nach rechts um eine feste Spaltenanzahl wird unterhalb des Schlüsselworts, welches eine der obigen Anweisungen eröffnet (REPEAT, ..., WITH).

(iv) "IF ... THEN", "ELSIF ... THEN", "WHILE ... DO", "FOR ... DO", "CASE ... OF" und "WITH ... DO" stehen nach Möglichkeit auf einer Zeile (vgl. oben); auf keinen Fall steht "THEN" direkt unter "IF" bzw. "ELSIF", "DO" steht nicht unter "WHILE", "FOR" oder "WITH", und "OF" nicht unter "CASE".

Beispiele zu (iii) und (iv):

```
IF a < b THEN
  c:=d;
  d:=a
ELSIF a = B THEN
  d:=a;
  a:=c
ELSE
  c:=b
END (*IF*);
```

```
WHILE (limit < m) DO
   a:=1;
   REPEAT
      a:=2*a;
      n:=n + 1
   UNTIL n = limit;
   WriteCard(a,6);
   limit:=limit + 1
END (*WHILE*);
```

Einer Sonderbehandlung bedarf die CASE-Anweisung:

(v) In Case-Anweisungen beginnen und enden alle - den einzelnen Fällen zugeordneten - Teilanweisungen (die "Zweige") auf gleicher Spalte. Entsprechend sind die Listen der "Fall-Bezeichner" zu arrangieren.

Beispiel:

```
CASE zeichen OF
  "a","b",
  "c","d": a:=a + 1;
           b:=b + 1;
           c:=c + 1;
           d:=d + 1 |
  "e","f": e:=e + 1;
           f:=f + 1
  ELSE
           z:=z + 1
END (*CASE*):
```

Und schließlich:

(vi) Nach einem "END" wird per Kommentar immer kenntlich gemacht, welche der strukturbildenden Anweisungen abgeschlossen wird (vgl. Beispiele zu (ii) - (v)!).

Natürlich sollen diese Vorschriften nicht diktatorisch verordnet werden; insofern sind es eher Vorschläge. Es ist jedoch wichtig, daß es irgendeine in sich stimmige und begründbare Vereinbarung über derartige Layout-Regeln gibt. Nicht zuletzt von Interesse ist dabei die Ästhetik! (Wieder mag man hierüber lächeln.) Eine "Metaregel" könnte gar lauten:

"Ein Programm muß schön aussehen!"

(Vielleicht hat nach irgendeiner ästhetischen Theorie auch das Chaos seinen Reiz; dem ordnenden Geist jedoch ist es ein Greuel!)

Übrigens kann die Einhaltung von Layout-Regeln sehr leicht erzwungen werden: durch geeignete Druckprogramme etwa, die diese Regeln "verinnerlicht" haben. Solche Druckprogramme heißen auch "Pretty-Printer". Sie zählen zu den einfachsten (aber auch sehr nützlichen und wirkungsvollen) Werkzeugen des Programmierers, mit deren Hilfe der "Dokumentationswert" eines Programmtextes erhöht werden kann.

REGEL: Für alle "aktiven" (also Moduln, Prozeduren und Funktionsprozeduren) und "passiven" (Konstanten, Typen, Variablen) Objekte eines Programms sind Namen zu wählen, die eine Vorstellung von der Bedeutung bzw. der Rolle des jeweiligen Objekts geben. Im Rahmen größerer Projekte sollten Standards für die Namensgebung gesetzt und eingehalten werden.

Die Befolgung dieser Regel trägt dazu bei, den Aufwand zur Programm-internen Dokumentation mittels expliziter Kommentare zu reduzieren. Strikt abzulehnen ist der extensive Gebrauch von kryptischen Abkürzungen durch wenige Zeichen. Dies mag (oder muß) noch vertretbar sein bei Verwendung von Programmiersprachen, deren Syntax die Zahl der Zeichen pro Bezeichner auf 6

(oder 8) beschränkt. (Noch heute viel benutzte Versionen von FORTRAN tun dies zum Beispiel.) Moderne Sprachen - wie MODULA-2 - verlangen diese Einschränkung jedoch nicht; es können (fast!) beliebig lange Bezeichner gewählt werden. Allenfalls werden bei der Compilierung nur die (z.B.) ersten 15 Zeichen berücksichtigt. Die Wahrscheinlichkeit aber, daß zwei verschiedene Bezeichner mit einem identischen Präfix der Länge 15 vorkommen, ist recht gering.

Immer bedeutungsvolle Namen zu finden, ist sicherlich keine leichte Aufgabe. Einige Phantasie ist hierzu wohl vonnöten. Dennoch läßt sich manches systematisieren. In diesem Sinne wollen wir einige Anhaltspunkte für die Namensgebung zusammenstellen.

(i) Die Zugehörigkeit eines Objekts zu einer (wie auch immer) definierten Klasse von Objekten sollte aus seinem Namen ersichtlich sein.

Dies läßt sich etwa durch die einmalige Festlegung jeweils einzubauender Namensbestandteile erreichen. Es ist sicher nicht schwer, gute Gründe dafür zu finden, daß es in MODULA-2 - Programmen nützlich sein kann, einem Bezeichner sofort anzusehen, ob er eine Konstante, einen Typ oder eine Variable benennt. Dazu könnte man folgende Konventionen vereinbaren:

- Prozedur- und Modul-Namen beginnen mit Großbuchstaben und enden weder auf "C", "Typ" noch "Ptr".
- vom ersten Zeichen abgesehen sollten für Bezeichner (zur schnellen Unterscheidung von den in Großbuchstaben geschriebenen Schlüsselwörtern) generell Kleinbuchstaben gewählt werden; zur Verbesserung der Lesbarkeit jedoch können Großbuchstaben (am Beginn von Namensbestandteilen) eingefügt werden.

Beispiele für gemäß dieser Konvention aufgebaute Bezeichner sind (vgl. auch die Beispiele in Anhang und die bisher angeführten Beispiele):

TabgrC	-	ein Konstanten-Bezeichner
BlankC	-	ein Konstanten-Bezeichner
ZeilenTyp	-	ein Typ-Bezeichner
MacheZeile	-	ein Prozedur-Name
zeilenNr	-	ein Variablen-Name
zeilenPtr	-	Name einer Pointer-Variablen

Zu beachten ist freilich, daß eine solche Konvention nur für (vom Programmierer) selbst definierte Namen gelten kann, da die durch sogenannte MODULA-2 - Standard-Bibliotheken vorgegebenen Namen derartige Unterscheidungen nicht notwendigerweise erkennbar machen.

Eine weitere Standardisierung von Namensbestandteilen kann sich aus der Verwendung von Objekten zu bestimmten Verarbeitungszwecken ergeben. Der Name einer Variablen etwa, in der eine Summe akkumuliert wird, könnte stan-

dardmäßig auf "Summe" oder "Sum" enden; der Name einer Variablen, in die Sätze einer Datei eingelesen werden, könnte das Suffix "Record", "Rec" oder "Satz" erhalten, usw..

(ii) Die Hauptbestandteile von Namen sollten problembezogen sein.

Der folgenden Funktionsprozedur zum Beispiel kann vermutlich nur ein Kalender-Experte unmittelbar ansehen, was sie leistet:

```
PROCEDURE Sj(j: CARDINAL): BOOLEAN;
VAR s: BOOLEAN;
BEGIN
  s:=((j MOD 4 = 0) AND (j MOD 100 # 0))
     OR (j MOD 400 = 0);
  RETURN s
END Sj;
```

In der nachstehenden Version jedoch ist sie geeignet, dem Leser eine wichtige Information über unser Kalendersystem zu vermitteln:

```
PROCEDURE IstSchaltjahr(jahr: CARDINAL): BOOLEAN;
VAR s: BOOLEAN;
BEGIN
  s:=((jahr MOD 4 = 0) AND (jahr MOD 100 # 0))
     OR (jahr MOD 400 = 0);
  RETURN s
END IstSchaltjahr;
```

Die Anweisung

"x1:=x2 * x3 * (1 + 1/(Exp(1 + x3,x4) - 1));"

ist sicher schneller zu editieren als

"annuitaet:=kapital * zinsSatz * (1 + 1/(Exp(1 + zinsSatz,laufZeit) - 1));",

dafür wird (nicht nur dem Experten auf dem Gebiet der Finanzmathematik) aus der zweiten Fassung aber sofort klar, worum es geht.

"Gute" Namen zu finden, hat immer auch einen psychologischen Aspekt. Sie sollten beim Leser die richtigen Assoziationen wecken. Ein Programmierer, dessen Programm eine "Benutzer-Identifikations-Nummer" bearbeiten soll, und dem diese Bezeichnung (mit Recht) zu lang ist, sollte sich nicht auf Abkürzungen einlassen wie

"bin", "benin" oder auch einfach "benutzer".

Die korrekte Assoziation wird in diesem Fall vielleicht ein Bezeichner wie

"benIdNr" oder "benutzerId"

hervorrufen.

Bei mathematischen Formeln sollten Variablen-Namen gewählt werden, die in mathematischen Notationen "eine Tradition" haben. ("i", "j", "k" z.B. für Indizes, "x" für Argument-Variablen und "y" für Funktionswerte, sofern keine anderen, dem Problem angepaßteren Bezeichner vorzuziehen sind (s.o.).)

Ferner sollte bei der Namensvergabe darauf geachtet werden, daß von ihrer semantischen Rolle her verschiedene Objekte auch deutlich verschieden heissen. Ein (eher vages) Maß für diesen Unterschied ist die "psychologische Distanz". Die folgenden Beispiele hierzu stammen aus [LNH]:

Name 1	Name 2	psych. Distanz
bkrpnt	brkpnt	sehr gering
Movlf	Movrf	gering
code	kode	klein
iden	ident	klein
omega	delta	groß
wurzel	diskriminante	groß und informativ

Das zweite dieser Bezeichnerpaare ist einem realen Programm entnommen, bei dem die geringe "Distanz" zwischen den Bezeichnern zu großem Aufwand bei der Fehlersuche führte. Die Bezeichner benannten Funktionsprozeduren mit den folgenden Aufgaben:

Prüfe, ob ein Übergang möglich ist:

Movlf(sq) - nach links vom (from) Feld "sq",
Movlt(sq) - von links zum (to) Feld "sq",
Movrf(sq) - nach rechts vom Feld "sq",
Movrt(sq) - von rechts zum Feld "sq".

Durch wiederholte Verwechslung von "Movlf" und "Movrf" schlichen sich in dem Programm Fehler ein, deren Korrektur Tage in Anspruch nahm.

REGEL: Im Programm verwendete Konstante (Literale) sollten immer durch Konstanten-Deklaration an Bezeichner gebunden sein.

Der Effekt der Anwendung dieser Regel ist zweifach: Zum einen werden - entsprechend den Regeln für die Namensgebung - die Bedeutungen von Konstanten durch die Bezeichner ausgedrückt. Zum zweiten werden damit einfache Programm-Modifikationen rationalisiert, und eine mögliche Fehlerquelle bei Modifikationen wird ausgeschaltet. Die folgenden (zugegebenermaßen etwas gekünstelten) Beispiele mögen dies belegen:

```
PROCEDURE RechtEck(x,y: CARDINAL);
(*Die Prozedur zeichnet ein Rechteck mit Breite 28 Pixel und
Länge 36 Pixel auf dem Bildschirm; die linke untere Ecke hat die
(Pixel-)Koordinaten (x,y)*)
```

```
BEGIN
  SetzeGrafikCursor(x,y);
  LinieRel(28,0);
  LinieRel(0,36);
  LinieRel(-28,0);
  LinieRel(0,-36)
END RechtEck;
```

(Die von "RechtEck" aufgerufenen Prozeduren seien entweder im gleichen Modul wie "RechtEck" deklariert oder über eine IMPORT-Liste bezogen.)

Den Literalen "36" und "28" ist nicht anzusehen, welches die Höhe und welches die Breite angibt. Im einleitenden Kommentar muß diese Information ausdrücklich gegeben werden. Nicht so in der zweiten Version:

```
PROCEDURE RechtEck(x,y: CARDINAL);
(*Die Prozedur zeichnet ein Rechteck auf dem Bildschirm; die
linke untere Ecke hat die (Pixel-)Koordinaten (x,y)*)
CONST BreiteC = 28;
HoeheC = 36;
BEGIN
  SetzeGrafikCursor(x,y);
  LinieRel(BreiteC,0);
  LinieRel(0,HoeheC);
  LinieRel(-BreiteC,0);
  LinieRel(0,-HoeheC)
END RechtEck;
```

Soll nun ausprobiert werden, welche Rechtecke auf dem Bildschirm wohl am gefälligsten aussehen, so sind in der ersten Version Änderungen an vier Stellen anzubringen, während in der zweiten Version nur zwei Änderungen nötig sind. Es ist leicht sich vorzustellen, welche Mühe ein größeres Programm machen würde, das nach dem Muster der ersten Version verfaßt ist. Auch ein komfortabler Text-Editor mit mächtigen Text-Ersetzungsfunktionen wäre hier keine große Hilfe. Denn nichts garantiert, daß das Literal "36" (z.B.) an anderer Stelle nicht eine ganz andere Bedeutung hat!

Beispiel:

Ein Unternehmen habe 12 Abteilungen. Pro Abteilung und Monat soll der Umsatz ermittelt werden. Je Monat ist der Umsatz des gesamten Unternehmens und je Abteilung der Jahresumsatz auszurechnen. Das Programm, welches diese Aufgaben erledigt, mag folgende Deklarationen und Code-Fragmente enthalten:

```
...
VAR abtUmsatz:        ARRAY (1..12),(1..12) OF REAL;
    abtJahrUmsatz:    ARRAY (1..12) OF REAL;
    totMonUmsatz:     ARRAY (1..12) OF REAL;
    monat, abteilung: CARDINAL;
BEGIN
  ...
  FOR abteilung:=1 TO 12 DO
    FOR monat:=1 TO 12 DO
      ErmittleUmsatz(abtUmsatz(abteilung,monat))
    END (*FOR*)
  END (*FOR*);
  ...
  FOR abteilung:=1 TO 12 DO
    abtJahrUmsatz(abteilung):=0;
    FOR monat:=1 TO 12 DO
      abtJahrUmsatz(abteilung):=abtJahrUmsatz(abteilung)
                              + abtUmsatz(abteilung,monat)
    END (*FOR*)
  END (*FOR*);
  ...
  FOR monat:=1 TO 12 DO
    totMonUmsatz(abteilung):=0;
    FOR abteilung:=1 TO 12 DO
      totMonUmsatz(monat):=totMonUmsatz(monat)
          + abtUmsatz(abteilung,monat)
    END (*FOR*)
  END (*FOR*);
  ...
END ...;
```

Das Literal "12" taucht hier an vielen Stellen mit verschiedenen Bedeutungen auf. Es wird fatal, wenn eine Abteilung wegfällt oder neu gegründet wird. Man stelle sich das "Chaos" vor, das entstünde, wenn nicht wenigstens vernünftige Namen für die Laufvariablen der einzelnen Schleifen gewählt worden wären.

Es ist klar, wie das Programm mindestens geändert werden muß, um mit unserer Regel konform und damit wartbarer zu werden:

```
...
CONST AbtAnzC = 12; MonAnzC = 12;
...
VAR abtUmsatz:        ARRAY (1..AbtAnzC),(1..MonAnzC) OF REAL;
    abtJahrUmsatz:    ARRAY (1..AbtAnzC) OF REAL;
```

```
    totMonUmsatz:     ARRAY (1..MonAnzC) OF REAL;
    monat, abteilung: CARDINAL;
...
BEGIN
  ...
  FOR abteilung:=1 TO AbtAnzC DO
    FOR monat:=1 TO MonAnzC DO
      ErmittleUmsatz(abtUmsatz(abteilung,monat))
    END (*FOR*)
  END (*FOR*);
  ...
  FOR abteilung:=1 TO AbtAnzC DO
    abtJahrUmsatz(abteilung):=0;
    FOR monat:=1 TO MonAnzC DO
      abtJahrUmsatz(abteilung):=abtJahrUmsatz(abteilung)
                                + abtUmsatz(abteilung,monat)
    END (*FOR*)
  END (*FOR*);
  ...
  FOR monat:=1 TO MonAnzC DO
    totMonUmsatz(abteilung):=0;
    FOR abteilung:=1 TO AbtAnzC DO
      totMonUmsatz(monat):=totMonUmsatz(monat)
            + abtUmsatz(abteilung,monat)
    END (*FOR*)
  END (*FOR*);
  ...
END ...;
```

In dieser Version genügt es allerdings noch nicht der folgenden:

REGEL: Zusammengesetzte Datenstrukturen sollten immer durch Typ-Deklaration an einen Typ-Bezeichner gebunden werden; ebenso sollten nach Möglichkeit problembezogene Enumerationstypen (mit gut gewählten Bezeichnern für die Skalare) definiert werden.

Beide Teile dieser Regel sind durch den Wunsch nach besserer Lesbarkeit motiviert. (In dieser Form sind sie natürlich nur dann anwendbar, wenn Sprachen wie MODULA-2 oder auch PASCAL eingesetzt werden.) Außerdem hat die Benennung zusammengesetzter Datenstrukturen einen Effekt, der dem der Konstanten-Benennung vergleichbar ist. Wird die Regel berücksichtigt, so entsteht aus den letzten Beispiel-Programm-Fragmenten die folgende Version (man beachte, daß das Literal "12" im Zusammenhang mit "Monaten" nicht mehr auftaucht):

```
...
CONST AbtAnzC = 12;
TYPE MonatsTyp = (januar,februar,maerz,april,mai,juni,juli,
                  august,september,oktober,novemver,dezember);
     AbtUmsatzTyp = ARRAY (1..AbtAnzC),MonatsTyp OF REAL;
     AbtJahrUmsatzTyp = ARRAY (1..AbtAnzC) OF REAL;
     TotMonUmsatzTyp = ARRAY MonatsTyp OF REAL;
...
VAR abtUmsatz: AbtUmsatzTyp;
     abtJahrUmsatz: AbtJahrUmsatzTyp;
     totMonUmsatz: TotMonUmsatzTyp;
     monat: MonatsTyp;
     abteilung: (1..AbtAnzC);
...
BEGIN
  ...
  FOR abteilung:=1 TO AbtAnzC DO
    FOR monat:=januar TO dezember DO
      ErmittleUmsatz(abtUmsatz(abteilung,monat))
    END (*FOR*)
  END (*FOR*);
  ...
  FOR abteilung:=1 TO AbtAnzC DO
    abtJahrUmsatz(abteilung):=0;
    FOR monat:=januar TO dezember DO
      abtJahrUmsatz(abteilung):=abtJahrUmsatz(abteilung)
                              + abtUmsatz(abteilung,monat)
    END (*FOR*)
  END (*FOR*);
  ...
  FOR monat:=januar TO dezember DO
    totMonUmsatz(abteilung):=0;
    FOR abteilung:=1 TO AbtAnzC DO
      totMonUmsatz(monat):=totMonUmsatz(monat)
                          + abtUmsatz(abteilung,monat)
    END (*FOR*)
  END (*FOR*);
  ...
END ...;
```

REGEL: Kommentare sollten u.a. eingefügt werden zur

a: Erläuterung der Verwendung bzw. Bedeutung von Programm-Objekten, sofern dies nicht schon durch die Bezeichnungen geschieht;

b: Herstellung eines expliziten Bezuges zu Entwurfsunterlagen;

c: Beschreibung des Zwecks von Anweisungsfolgen, die für den Programm-Ablauf wesentlich sind;

d: Beschreibung der Wirkung wesentlicher Anweisungen auf den Inhalt von Variablen;

e: Dokumentation von Modifikationen.

Kommentare, die nur wiederholen, was bereits durch den Programmtext selbst ausgedrückt wird, sind überflüssig und zu vermeiden.

Das Verfassen knapper aber dennoch aussagefähiger Kommentare gehört ebenso wie das Finden sinnvoller Objektnamen zu denjenigen Aufgaben eines Software-Entwicklers, die ausgezeichnete natürlich-sprachliche (!) Fähigkeiten voraussetzen. Daß so manche Software-Entwickler dieser Voraussetzung kaum genügen, ist wohl eine der Ursachen für die im allgemeinen sehr mangelhafte Kommentierung vieler industrieller und kommerzieller Programme.

Die Kommentierung von Programmen gemäß (c) wird in Kapitel 3 eine besondere Rolle spielen, wogegen wir uns mit Kommentaren der Art (d) in Kapitel 4 sehr eingehend beschäftigen werden.

Kommentare, welche Modifikationen dokumentieren, sollten - zur leichten Auffindbarkeit mit Hilfe eines Text-Editors etwa - zusätzlich eine Standard-Zeichenfolge enthalten.

2.2.2 Aspekte der strukturellen Komplexität

Wir bemerken nochmals, daß der Begriff der "strukturellen Komplexität" von Programmen, den wir hier ins Spiel bringen, unmittelbar nichts mit dem Komplexitäts-Begriff zu tun hat, der in der Komplexitäts-Theorie (vgl. Abschnitt 2.1.4) untersucht wird. Vielmehr verstehen wir "strukturelle Komplexität" als Eigenschaft eines Programms, die auf dessen Verstehbarkeit entscheidenden Einfluß hat. In der Vergangenheit wurden zahlreiche Versuche unternommen, auch diese Art der Komplexität zu präzisieren, meßbar zu machen. Die Beschäftigung hiermit hat unter der Überschrift "Software-Metrik" ("software-metrics") Eingang in die wissenschaftliche Literatur gefunden. Einige der folgenden Regeln sind durch die Ergebnisse derartiger Untersuchungen motiviert.

REGEL: Der Rumpf von Prozeduren und Funktionsprozeduren sollte nicht mehr als ca. 50 Druckzeilen (ca. eine Druckseite) umfassen.

Man kann sicher darüber streiten, ob diese Regel nicht eher ein Entwurfsziel beschreibt denn einen Aspekt guten Programmierstils charakterisiert. Gewiß ist es Sache des Entwurfs, u.a. Spezifikationen für Prozeduren zu liefern, und man sollte eine Revision des Entwurfs verlangen, wenn sich herausstellt, daß eine Spezifikation zu einer überlangen Prozedur führt. Nichtsdestoweniger ist es häu-

fig ein Symptom für mangelhaft durchdachte, umständliche Programmierung, wenn eine Prozedur Überlänge erreicht. Darüberhinaus können - auf der Basis einer Spezifikation - bei der Programmierung einer Prozedur durchaus Teilaufgaben erkennbar werden, die in gleicher oder ähnlicher Weise an mehreren Stellen des Prozedur-Textes zu formulieren wären. Diese Teilaufgaben zu isolieren und durch geeignete Prozeduren zu lösen (die dann etwa lokal zur ursprünglichen Prozedur zu deklarieren sind), ist Sache "stilvoller" Programmierung. Ein instruktives Beispiel hierfür werden wir in Kapitel 3 studieren. Die genannte Regel kann also mit gutem Grund in diesem Abschnitt erscheinen.

Es muß nochmals betont werden, daß die Zahl "50" nur als ein Richtwert zu verstehen ist. Ausnahmen dürften leichter zu rechtfertigen sein, wenn sie zum Beispiel wenigstens die folgende Regel nicht verletzen:

REGEL: Die Tiefe der Schachtelung strukturbildender Anweisungen sollte nicht mehr als vier betragen.

Auch ein "Schachtelungs-Monster" wie

```
...
IF Bed1 THEN
  WHILE Bed2 DO
    REPEAT
      IF Bed3 THEN
        WHILE Bed4 DO
          (*Anweisungsfolge*)
        END (*WHILE*)
      ELSE
        ...
      END (*IF*)
    UNTIL Bed5
  END (*WHILE*)
END (*IF*);
...
```

legt den Verdacht nahe, daß der Programmierer die Lösung der ihm gestellten Aufgabe nicht genügend durchdacht hat. Die Wartung von Programmen, die solche Konstruktionen enthalten, ist nicht leicht. Ein "Wartungs-Programmierer", der z.B. herausfinden soll, unter welchen Bedingungen im obigen Programm-Fragment die (*Anweisungsfolge*) ausgeführt wird, hätte damit seine liebe Not.

Vermutlich wird auch hier ein "Detail-Entwurfsschritt" - ähnlich wie wir ihn bei der Diskussion der vorigen Regel postuliert haben - die Schachtelungstiefe durch Definition zusätzlicher, lokal sinnvoller Prozeduren verringern.

Es ist anzumerken, daß die Unterscheidung vieler Fälle, die nicht einfach durch spezielle Werte einer Variablen gegeben sind, in manchen Programmiersprachen

nur mit Hilfe tief geschachtelter "IF ... THEN ... ELSE ..." - Anweisungen vorgenommen werden kann. Bei der Definition von MODULA-2 wurde der - trivialen - Erkenntnis Rechnung getragen, daß für derartige Fallunterscheidungen gewissermaßen ein verallgemeinertes "CASE-Konstrukt" notwendig ist. Es lautet:

```
IF ... THEN
  ...
ELSIF ... THEN
  ...
ELSIF ... THEN
  ...
  ...
ELSE
  ...
END (*IF*); .
```

Die Layout-Regel, die wir hierfür formuliert haben (Abschnitt 2.2.1) und die hier berücksichtigt ist, macht ganz deutlich, daß sich die Anweisungsfolgen, die den einzelnen Bedingungen zugeordnet sind, tatsächlich alle auf der gleichen Schachtelungstiefe befinden.

(Bemerkung: In PASCAL würde man aus diesem Grund eine geschachtelte "IF ... THEN ... ELSE ..." - Konstruktion auch so formatieren:

```
IF ... THEN
  BEGIN
    ...
  END
ELSE IF ... THEN
  BEGIN
    ...
  END
ELSE
  BEGIN
    ...
  END; .)
```

REGEL: "... THEN IF ..." - Konstruktionen sollten vermieden werden.

Der Grund: Sie sind erfahrungsgemäß schwerer lesbar und verständlich als "... ELSIF ..." - Konstruktionen.

"... THEN IF ... " - Konstruktionen können äquivalent umgeformt werden in "... ELSIF ..." - Konstruktionen:

Seien B1 und B2 boolesche Ausdrücke und A1, A2, A3 beliebige Anweisungsfolgen. Dann gilt, wie man leicht nachprüft:

```
IF B1 THEN
  IF B2 THEN
    A1
  ELSE
    A2
  END (*IF*)
ELSE
  A3
END (*IF*);
```

ist äquivalent zu:

```
IF NOT(B1) THEN
  A3
ELSIF B2 THEN
  A1
ELSE
  A2
END (*IF*);
```

Der zweiten Formulierung ist bezüglich der Lesbarkeit und Verständlichkeit sicher der Vorzug zu geben.

REGEL: "Trickreiche" Programmierung sollte vermieden werden.

Dies ist die bisher wohl "gummiartigste" Regel. Um zu demonstrieren, was ungefähr damit gemeint ist, sei das folgende Beispiel aus [KEP] - in MODULA-2 Notation - zitiert:

Mit dem Programm-Segment

```
FOR i:=1 TO n DO
  FOR j:=1 TO n DO
    a(i,j):=(i DIV j) * (j DIV i)
  END (*FOR*)
END (*FOR*);
```

wollte ein Programmierer auf offenbar besonders kurze und schlaue Weise ... , ja, was wollte er eigentlich? Vielleicht wird er es selbst nach einiger Zeit bei erneuter Betrachtung seines Werks nur noch mit einigem Nachdenken erkennen. Hätte er folgenden Code geschrieben, so würde nicht nur er sofort wiedererkennen bzw. verstehen, worum es geht:

```
FOR i:=1 TO n DO
  FOR j:=1 TO n DO
    a(i,j):=0
  END (*FOR*);
  a(i,i):=1
END (*FOR*);
```

Aha: "a" sollte als Einheitsmatrix initialisiert werden. So klar und ebenso knapp kann man dies notieren, und effizienter in der Ausführung sind diese Anweisungen außerdem!

Effizienz-Argumente liefern im übrigen immer die beliebteste Entschuldigung für "trickreiche" Programmierung. Dabei wird der wichtigste ökonomische Gesichtspunkt vergessen: Man vergleiche die durch "Tricks" (z.B. auch Mehrfachverwendung von Variablen, unangemessene Verwendung "Hardware-naher"

Konstrukte einer Programmiersprache, usw.) möglicherweise erzielte Einsparung an Rechenzeit und Speicherplatz (für Programm und Daten) mit dem Aufwand, den so "getunte" Programme bei ihrer Wartung verursachen. Dann wird sofort klar, daß die Kosten der menschlichen Arbeit den (heutigen!) Preis für Speicher und Prozessoren bei weitem überwiegen. (Vor einigen Jahren wäre dieses Gegenargument noch weniger schlagkräftig gewesen.)

Wie weiter oben bereits angekündigt, werden wir in Abschnitt 2.3 studieren, wie Effizienz-Verbesserung durch guten Programmier-Stil, unter Vermeidung von "Tricks" also, erreicht werden kann.

REGEL: "GOTOs" sind zu vermeiden.

Daß wir diese Regel aufnehmen, geschieht nur der Vollständigkeit halber und hat ausschließlich "historische" Gründe. Dem aufmerksamen Absolventen eines MODULA-2 Kurses wird nicht entgangen sein, daß es eine "GOTO-Anweisung" in dieser Sprache nicht gibt.

Andere Sprachen zur imperativen Programmierung (auch PASCAL) enthalten eine GOTO-Anweisung. Mit ihrer Hilfe kann zum Beispiel in PASCAL-Programmen der durch die strukturbildenden Anweisungen für Selektion und Iteration gesteuerte "Kontrollfluß" beinahe beliebig umgelenkt werden. Ein undisziplinierter Gebrauch dieser Anweisung kann daher sehr leicht zu undurchsichtigem Programmtext führen. "GOTOs" verleiteten auch zu jener "trickreichen" Programmierung, die wir bereits angeprangert haben. Mit älteren Programmiersprachen (z.B. FORTRAN), in denen die Formulierung allgemeiner Selektions- und Iterations-Anweisungen nicht möglich ist, ist der Programmierer natürlich gezwungen, "GOTOs" zu verwenden, um beispielsweise ein "Wiederhole solange, als ... (WHILE)" oder "Wiederhole ... bis ... (REPEAT ... UNTIL)" zu realisieren. Das kann in vollständig systematischer Weise und also diszipliniert geschehen. Nur: die Crux ist, daß es nicht so geschehen muß. Wenn ein Programmierer nicht gelernt hat, unabhängig von der jeweiligen Programmiersprache in Selektions- und Iterations-Strukturen zu denken, so wird er dazu von einer Sprache wie FORTRAN auch nicht gezwungen. Der "naive" Programmierer wird nur "seine Sprache" sehen und sich alle Freiheiten herausnehmen, die diese ihm läßt. In FORTRAN (aber auch in anderen Sprachen) wurden in der Vergangenheit unzählige Programme verfaßt, deren "Kontrollfluß" durch den undisziplinierten Gebrauch von "GOTOs" so unentwirrbar ist, daß sie den Spitznamen "Spaghetti-Programme" erhielten. Diese waren und sind nicht wartbar.

Was aber berechtigt dazu, bei der Definition einer Programmiersprache - wie bei MODULA-2 geschehen - auf eine nahezu beliebige Umlenkbarkeit des "Kontrollflusses" durch GOTO-Anweisungen zu verzichten?

Dieser Verzicht wird erst ermöglicht durch ein grundlegendes Resultat der Berechenbarkeitstheorie. Danach kann jede - intuitiv - berechenbare Funktion

durch einen Formalismus berechnet werden, der für die Abfolge von Rechenschritten (also den "Kontrollfluß") neben dem einfachen Hintereinander ("Sequenz") nur die Selektion (entsprechend einer Bedingung) und die durch eine Abbruchsbedingung gesteuerte Iteration erlaubt (vgl. z.B. [WEG]).

Auf diese Weise theoretisch abgesichert, ist der Verzicht auf "GOTOs" natürlich durch die Absicht motiviert, die Herstellung von "Spaghetti-Programmen" von vornherein zu verhindern. (Gelegenheit macht Diebe!) Ein weiterer Grund ergibt sich aus der in Kapitel 4 zu behandelnden Thematik: Dort werden wir die Semantik (d. h. die "Bedeutung") der wichtigsten strukturbildenden Sprachelemente von MODULA-2 definieren. Für die (unbedingte) GOTO-Anweisung wäre eine solche Definition nicht offensichtlich. Die Tatsache, daß "Spaghetti-Programme" im allgemeinen schwer verständlich sind, übersetzt sich daher unmittelbar in die Aussage: "Solche Programme haben keine einfach zu definierende Semantik."

Um der Wahrheit die Ehre zu geben: GOTO-Anweisungen mit beschränkter Verwendung gibt es doch in MODULA-2. Es ist dies zum einen die "EXIT-Anweisung", mit der eine durch die "LOOP-Anweisung" gesteuerte Iteration abgebrochen werden kann. Für dieses "GOTO" gibt es jedoch nur ein Sprungziel: das Ende der Schleife! Zum anderen ist es die "RETURN-Anweisung". Mit ihr kann die Ausführung einer Prozedur (oder Funktions-Prozedur) vor Erreichen der letzten Anweisung abgebrochen werden. Aber auch hier gibt es nur ein wohldefiniertes Sprungziel, nämlich das Ende der Prozedur. MODULA-2 hat damit die "GOTO-Anweisung" "diszipliniert"!

REGEL: Seiteneffekte durch Funktionsprozeduren sind zu vermeiden.

Es ist schlechter Stil, Funktionsprozeduren für andere Zwecke zu benutzen als für die Berechnung eines (des abzuliefernden) Wertes in Abhängigkeit von den als Parameter übergebenen Argumenten.

Mögliche andere bzw. zusätzliche Zwecke wären:

- Modifikation des Wertes globaler Variabler,
- Modifikation von Parameterwerten,
- Ein-/Ausgabe-Operationen.

Diese werden als *Seiteneffekte* bezeichnet. (Wir fügen an dieser Stelle hinzu, daß auch "gewöhnliche" Prozeduren die Veränderung globaler Variabler nach Möglichkeit vermeiden sollten. Gute Argumente hierfür werden sich allerdings erst - implizit - aus der Diskussion von Modularisierungstechniken in Kapitel 6, Abschnitte 6.4.1 und 6.4.2, ergeben.)

Die folgenden Beispiele demonstrieren, wie unangenehm solche "Seiteneffekte" werden können, wenn Programme geändert werden müssen. Sie stammen aus [LNH].

Gegeben sei das Programm-Fragment:

```
...
VAR a,b,c: REAL;
...
PROCEDURE F(x:REAL): REAL;
VAR f: REAL;
BEGIN
  a:=a + 1;
  f:=a * x;
  RETURN f
END F;
...
BEGIN
  a:=10;
  b:=3;
  c:=F(b) + F(b)
END ...
```

Auf den ersten Blick scheint es völlig problemlos zu sein, die letzte Anweisung in diesem Programmstück durch "c:=2 * F(b)" zu ersetzen. Man mache sich jedoch klar, daß nach Ausführung des ursprünglichen Programms die Variable "c" den Wert "69" enthält, während nach der vorgeschlagenen Ersetzung der Inhalt von "c" "66" ist!

Das nächste Beispiel zeigt, daß Seiteneffekte die Gültigkeit einfacher algebraischer Gesetze "aufheben" können:

```
...
VAR a,b,c: REAL;
...
PROCEDURE F(x:REAL): REAL;
VAR f: REAL;
BEGIN
  a:=a + 1;
  f:=a * x;
  RETURN f
END F;
...
PROCEDURE G(x:REAL): REAL;
VAR g: REAL;
BEGIN
  a:=a + 2;
  g:=a * x;
  RETURN g
END G;
```

```
...
BEGIN
  a:=10;
  b:=3;
  c:=F(b) + G(b)
END ...
```

Wieder erscheint es auf den ersten Blick selbstverständlich (schließlich gehorcht ja die Addition dem Kommutativgesetz!), daß man die letzte Anweisung durch "c:=G(b) + F(b)" ersetzen kann. Man überzeuge sich aber, daß dies nicht möglich ist: In der ursprünglichen Version wird "c" nach Ausführung der letzten Anweisung den Wert "72" enthalten; in der durch die Ersetzung gewonnenen Version enthält "c" den Wert "75".

Ein letztes Beispiel:

```
...
VAR a,b,c: REAL;
...
PROCEDURE F(x:REAL; VAR y:REAL): REAL;
VAR f: REAL;
BEGIN
  y:=y + x;
  f:=x * y;
  RETURN f
END F;
...
BEGIN
  a:=4;
  b:=F(5,a);
  c:=F(5,a)
END ...
```

In diesem Fall wird durch die Funktionsprozedur der Inhalt der als Variablen-Parameter übergebenen Variablen "y" geändert. Daher werden - wie man leicht nachprüft - den Variablen "b" und "c" verschiedene Werte zugewiesen, obwohl sich die zur Berechnung dieser Werte ergehenden Aufrufe der Funktions-Prozedur "F" nicht sichtbar unterscheiden.

Derartige Seiteneffekte sorgen also dafür, daß das Verständnis der Wirkung von Programmen unnötig erschwert wird.

Ein-/Ausgabe-Operationen als Seiteneffekte sind noch eher erträglich, da sie im allgemeinen besser überschaubar und dokumentierbar sind. So wäre eine Funktions-Prozedur

"PROCEDURE Schreibe(satz: ARRAY OF CHAR): BOOLEAN; ..."

denkbar, die eine Zeichenkette auf eine Datei zu schreiben hat und mit ihrem Ergebnis den Erfolg oder Mißerfolg dieser Aktion signalisiert. Dies sollte dann allerdings die einzige Aufgabe einer solchen Funktionsprozedur sein.

Es gibt im übrigen MODULA-2 - Compiler, die Funktionsprozeduren mit Seiteneffekten unter Umständen sehr "rauh" behandeln. Zum Beispiel kann es sein, daß ein Übersetzer die ansonsten sehr nützliche Eigenschaft hat, boolesche Ausdrücke zu "optimieren".

```
...
CONST DebugC = FALSE;
...
BEGIN
  ...
  IF DebugC THEN
    (*Ausgabe der Inhalte bestimmter Variabler zu Testzwecken*)
  END (*IF*);
  ...
END ...;
```

kein Code erzeugt wird. Das gleiche gilt dann aber auch für

```
"...
IF (F(x)=1) AND DebugC THEN... "
```

und "F" wird nicht ausgeführt, wenn DebugC=FALSE! War mit "F" aber ein Seiteneffekt beabsichtigt, so tritt natürlich auch dieser niemals ein.

2.5 Einfache Maßnahmen zur Verbesserung der Effizienz

In diesem Abschnitt werden wir einige Regeln angeben, durch deren Beachtung die Effizienz von Programmen verbessert werden kann. Wir beschränken uns dabei auf "Zeit-Effizienz".

Wohlgemerkt, und um es zu wiederholen: Es geht hier nicht um die Untersuchung von Datenstrukturen und Algorithmen (vgl. Abschnitt 2.1.4). An einmal beschlossenen Verfahren (sei es zum Sortieren, zur Verwaltung einer Namensmenge, zur Lösung einer Differentialgleichung usw.) ändern diese Regeln nichts.

Die meisten der Regeln zielen vielmehr auf einen "effizienten Gebrauch" der Elemente einer höheren Programmiersprache (in unserem Falle also MODULA-2). Was soll das heißen?

In höheren Programmiersprachen verfaßte Programme werden im allgemeinen compiliert (vgl. Abschnitt 1.1). Zwischen der Niederschrift eines Programmtextes in "maschinenferner" Notation und Terminologie und dem ablauffähigen Maschinen-Code finden also zum Teil recht komplizierte Analyse- und Übersetzungsvorgänge statt. Aus einzelnen Anweisungen in der höheren Programmiersprache

werden Maschinen-Code-Fragmente. (Die Wiederholung dieser Selbstverständlichkeiten geschieht hier nur der Vollständigkeit halber.) Zu einer Anweisung (bzw. Folge oder Schachtelung von Anweisungen) gibt es unter Umständen mehrere äquivalente Alternativen. "Äquivalent" bedeutet hier - ganz intuitiv -, daß sich an der Wirkung nichts ändert. Sehr wohl kann sich aber der erzeugte Maschinen-Code ändern: er kann zu schnellerem oder langsamerem Programmablauf führen.

Wie stark dieses "Phänomen" jeweils ausgeprägt ist, hängt natürlich von dem benutzten Compiler (der "Implementierung der Sprache") ab. Dennoch gibt es für praktisch alle höheren Programmiersprachen Situationen, in denen man äquivalente Anweisungen hinsichtlich der Geschwindigkeit des jeweils zugehörigen Maschinen-Codes durchaus beurteilen kann, und zwar ohne Rücksicht auf den Compiler.

Bevor wir solche speziellen Situationen genauer betrachten, wollen wir in einem ersten Unterabschnitt noch einige allgemeine Überlegungen zum Thema "Effizienz-Verbesserung" voranstellen. (Übrigens werden solche Überlegungen meist unter der Überschrift "Optimierung" zusammengefaßt. Dieser Ausdruck, der sich leider eingebürgert hat, ist jedoch völlig fehl am Platz: Denn keiner darf wohl ohne Beweis behaupten, daß er etwas so gut machen kann, daß es nicht mehr besser geht - und genau dies bedeutet ja "Optimierung".)

2.3.1 Allgemeine Überlegungen

In Abschnitt 2.2.2 argumentierten wir bereits, daß sich der Aufwand für Effizienz-Verbesserungen (durch "trickreiche" Programmierung) angesichts des geringen, dadurch gewonnenen Nutzens kaum lohnt.

Diese Aussage müssen wir, da wir in diesem Buch unser Augenmerk ja nicht auf eine ganz spezielle Art von Software beschränken wollen, ein wenig differenzieren. Tatsächlich ist Effizienz eine außerordentlich wichtige Eigenschaft von Programmen, die Teile der Basis-Software eines Rechners sind (Betriebs-System, Datenbank-Management-System, Compiler, usw.). Solche Programme werden häufig genutzt, ihre "Wartungs-Intervalle" sind relativ lang.

Dagegen abzugrenzen sind die meisten Programme der betrieblichen Datenverarbeitung bzw. (allgemeiner) die sogenannten "Anwendungs-Programme". Für sie gilt die Aussage, daß "Wartbarkeit ein weitaus größeres Gewicht hat als Effizienz", beinahe uneingeschränkt.

Effizienz-Verbesserung muß sich also lohnen! Es ist wenig sinnvoll, Arbeit einzusetzen für die Verbesserung der Effizienz eines Programms, das nur selten benutzt wird oder bei dem es auf eine schnelle Reaktion (z.B.) nicht ankommt.

Ebenso können wir behaupten, daß, wenn schon die Effizienz eines Programms verbessert werden soll, man dazu diejenigen Programmteile bevorzugt bearbei-

ten muß, deren Anweisungen am häufigsten ausgeführt werden. Diese Programmteile zu finden, ist nicht immer einfach. Zum einen können sich Anhaltspunkte aus dem Programm-Entwurf ergeben; zum anderen kann man sich spezieller "Software-Werkzeuge" bedienen, die ein "Ausführungs-Profil" anfertigen, aus dem hervorgeht, wie oft eine bestimmte Anweisung oder Prozedur während eines Programmlaufs benutzt wurde. Stehen solche Werkzeuge nicht zur Verfügung, so ist man gezwungen, seine Programme um Anweisungen zu ergänzen, die zum Beispiel die Zahl der Aufrufe von Prozeduren oder der Durchläufe von Programmteilen ermitteln.

Ganz offensichtliche Kandidaten für Ansätze zur Effizienz-Verbesserung sind freilich Anweisungen bzw. Anweisungsfolgen, über die sehr oft iteriert wird, die also - wie man sagt - "im Innern einer Schleife" liegen.

Van Tassel (vgl. [VAN]) nennt die folgenden allgemeinen Regeln, welche für Effizienz-Verbesserungen gelten:

- Verbessere nur dann, wenn es notwendig ist; andernfalls wird Arbeitszeit verschwendet, werden zusätzliche Fehler eingebracht, werden Lesbarkeit und Verstehbarkeit beeinträchtigt;
- falls Effizienz-Verbesserungen notwendig sein sollten, versuche man es zunächst mit einem "optimierenden" (vgl. unsere Bemerkung oben!) Compiler; solche Compiler unterwerfen den Programmtext einer gründlichen Analyse mit dem Ziel, schnelleren Maschinen-Code zu erzeugen.
- Ermittle die Programmteile, für die Effizienz-Verbesserung am lohnendsten ist; es ist im allgemeinen eine Verschwendung von Arbeitszeit, wenig benutzte Programmteile zu verbessern.
- Verbessere die Effizienz der so gefundenen Programmteile (z.B. nach den im folgenden Abschnitt zusammengestellten Regeln).

2.3.2 Spezielle Regeln

Bei der Formulierung der folgenden Regeln und Beispiele stützen wir uns ebenfalls auf [VAN].

Wir gruppieren die Regeln nach ihrem Bezug auf

- arithmetische Operationen,
- Iterations-Anweisungen (Schleifen),
- Selektions-Anweisungen,
- boolesche Ausdrücke,
- Indizierung (von Arrays)
- Prozeduraufrufe.

(i) Arithmetische Operationen

REGEL: Versuche, langsame Operationen durch schnellere zu ersetzen.

"Addition/Subtraktion - Multiplikation - Division", dies ist die Reihenfolge der arithmetischen Operationen entsprechend ihrer Ausführungs-Geschwindigkeit. Die genauen Geschwindigkeits-Verhältnisse hängen natürlich von der jeweiligen Hardware und dem Compiler ab und davon, ob mit INTEGER- oder REAL-Objekten operiert wird. Dessen ungeachtet zeigen die folgenden Beispiele, worauf es ankommt:

Ersetze Multiplikation durch Addition:

"a:=i + i + i;" ist schneller als "a:=3 * i;".

"x:=2 * (y + t) + (a - 1)/p;" ist schneller als "x:=2 * y + (a - 1)/p + 2 * t;".

Ersetze Division durch Multiplikation:

"b:=a * 0.2;" ist schneller als "b:=a/5.0;".

```
"rx:=1.0/x;                          "a:=1.0/x;
 a:=rx;            ist schneller als  b:=e/x;
 b:=e * rx;                           c:=b + d/x;"
 c:=b + d * rx;"
```

REGEL: Verwende, wenn möglich, INTEGER-Arithmetik.

Die meisten Prozessoren können INTEGER-Objekte mit wesentlich weniger Maschinen-Instruktionen manipulieren als REAL-Objekte. Diese Tatsache wird natürlich auch von Compilern ausgenutzt. Eine Ausnahme bilden unter Umständen Rechner mit zusätzlicher Hardware für Operationen mit REAL-Objekten ("Gleitkomma-Prozessoren", z.B. Intel 80x87), die vom Compiler auch berücksichtigt wird. In MODULA-2 sind zudem, falls möglich, CARDINAL-Objekte den INTEGER-Objekten vorzuziehen.

REGEL: Berechne mehrfach benutzte Ausdrücke nur einmal.

Die Rechtfertigung dieser Regel ist offensichtlich trivial. Dennoch einige Illustrationen:

```
"x:=y + a/b * c;     ist zu ersetzen durch    "abc:=a/b * c;
 z:=w + a/b * c;"                              x:=y + abc;
                                               z:=w + abc;"
```

```
"s1:=sin(th) + sin(th) * 2.0;                    "r:=sin(th);
 s2:=sin(th)/3.0;"        ist zu ersetzen durch    s1:=r + r + r;
                                                   s2:=r/3.0;"
```

"r1:=(-b + sqrt(b * b - 4.0 * a * c))/(2.0 * a);
r2:=(-b - sqrt(b * b - 4.0 * a * c))/(2.0 * a);"

ist zu ersetzen durch:

```
"d:=a + a;
 discriminante:=sqrt(b * b - 4.0 * a * c);
 r1:=(-b + discriminante)/d;
 r2:=(-b - discriminante)/d;"
```

In MODULA-2 ist es "verboten", in Ausdrücken oder Zuweisungen Objekte unterschiedlichen Typs zu "mischen". Das linke der beiden nachstehenden Programm-Fragmente ist also illegal. Einen Ausweg bietet MODULA-2 mit "Typ-Transfer-Funktionen" an. Unter Beachtung der obigen Regel kann das Fragment dann wie rechts gezeigt geschrieben werden:

```
...
VAR a,b,c,d: REAL;
    i: INTEGER;
...
  b:=a * i;
  c:=(a + i) * 2.0;
  d:=a * a/i;
...
```

```
...
VAR a,b,c,d,ir: REAL;
    i: INTEGER;
...
  ir:=FLOAT(i);
  b:=a * ir;
  c:=(a + ir) * 2.0;
  d:=a * a/ir;
...
```

REGEL: Eliminiere wiederholte Teilausdrücke.

Diese Regel kann als Variante der zuletzt genannten Regel aufgefaßt werden. Wiederholte Teilausdrücke sind gewissermaßen mehrfach benutzte Ausdrücke

```
"a:=x * y + 2.0 + x * y;" sollte ersetzt werden durch "xy:=x * y;
                                                       a:=xy + xy + 2.0;"
```

(Das spart eine Multiplikation. Manche Compiler entdecken eine solche Situation und führen die gezeigte Ersetzung implizit durch.)

Wiederholte Teilausdrücke können auch verdeckt auftreten:

```
"a:=b * b * c * c;"    wird zu    "bc:=b * c;
                                   a:=bc * bc;"
```

(Das spart ebenfalls eine Multiplikation, aber ein Compiler würde es vermutlich nicht entdecken.)

Mit der gleichen Begründung wie oben ist das Fragment

```
...
VAR x,a: REAL;
    i: INTEGER;
...
   x:=i * a * i * a * i;
...
```

in MODULA-2 nicht legal. Um es zu "reparieren", wird die Zuweisung ersetzt durch

```
"x:=FLOAT(i) * a * FLOAT(i) * a * FLOAT(i);"
```

Gemäß unserer Regel ist hier der wiederholte Teilausdruck "FLOAT(i)" zu eliminieren. Dazu wird eine zusätzliche Variable "ir" vom Typ REAL deklariert und die Zuweisung ersetzt durch:

```
"ir:=FLOAT(i);
 x:=ir * a * ir * a * ir;"
```

Noch besser wäre hier freilich die Modifikation

```
"x:=FLOAT(i * i * i) * (a * a);".
```

(ii) Iterations-Anweisungen

REGEL: Vermeide "Iterations-Anweisungen", wenn es geht.

"Iterations-Anweisungen" führen zu einem "Overhead", bedingt durch die Überprüfung von Bedingungen vor dem Eintritt in den "Schleifenrumpf" oder nach Verlassen des "Schleifenrumpfes". Mit einer FOR-Anweisung muß nicht nur bei jedem Durchlauf geprüft werden, ob der Endwert erreicht ist, sondern auch der Inhalt der Lauf-Variablen wird jeweils inkrementiert. Daher ist es sinnvoll, die folgende FOR-Anweisung, mit der der Wert eines Polynoms berechnet werden soll:

```
poly:=a[1] ;
FOR i:=2 TO 4 DO
  poly:=poly * x + a[i]
END (*FOR*); ...
```

zu ersetzen durch die Auswertung eines entsprechenden, geklammerten Ausdrucks und eine Wertzuweisung:

```
"poly:=((a[1] * x + a[2]) * x + a[3]) * x + a[4];"
```

REGEL: Reorganisiere die Schachtelung von FOR-Anweisungen, wenn es geht.

Mit der gleichen Begründung wie oben sollte bei der Schachtelung von FOR-Anweisungen darauf geachtet werden, den durch die Schleifenköpfe bedingten Overhead zu vermindern. Was man durch geschickte Reorganisation der Schachtelung erreichen kann, zeigt das folgende Beispiel:

```
...
FOR i:=1 TO 20 DO              (*wird einmal initialisiert*)
  FOR j:=1 TO 10 DO            (*wird 20-mal initialisiert*)
    FOR k:=1 TO 5 DO           (*wird 200-mal initialisiert*)
      (*Anweisungsfolge*)
    END (*FOR*)                (*wird 1000-mal beendet*)
  END (*FOR*)                  (*wird 200-mal beendet*)
END (*FOR*);                   (*wird 20-mal beendet*)
```

Insgesamt finden also 221 Initialisierungs-Vorgänge und 1220 Schleifenabschlüsse statt. Demgegenüber betrachte man:

```
...
FOR k:=1 TO 5 DO              (*wird einmal initialisiert*)
  FOR j:=1 TO 10 DO           (*wird 5-mal initialisiert*)
    FOR i:=1 TO 20 DO         (*wird 50-mal initialisiert*)
      (*Anweisungsfolge*)
    END (*FOR*)               (*wird 1000-mal beendet*)
  END (*FOR*)                 (*wird 50-mal beendet*)
END (*FOR*);                  (*wird 5-mal beendet*)
```

Hier gibt es nur 56 Initialisierungen und 1055 Abschlüße.

REGEL: Die Effizienz von Anweisungen auf der "untersten Ebene" eines Schachtelungs-Konstrukts ist zu verbessern.

Zweifellos ist dies eine der wichtigsten Regeln zur Effizienz-Verbesserung, da hier Anweisungen betroffen sind, die unter Umständen sehr oft auszuführen sind. Anwendbar sind hier insbesondere die Regeln zur Beschleunigung der Auswertung von arithmetischen Ausdrücken. Außerdem kann das "Ausklammern" von Aktionen aus "inneren Schleifen" einen beträchtlichen positiven Effekt für die Effizienz haben. Dies ist nichts anderes als eine Variante der Regel, nach der mehrfach benutzte Ausdrücke nur einmal auszuwerten sind.

Beispiel 1:

```
...
FOR i:=1 TO imax DO
  FOR j:=1 TO jmax DO
    x:=y * z + c[i,j]
  END (*FOR*)
END (*FOR*);
```

```
...
yz:=y * z;
FOR i:=1 TO imax DO
  FOR j:=1 TO jmax DO
    x:=yz + c[i,j]
  END (*FOR*)
END (*FOR*);
```

Im linken Programm-Fragment wird die Multiplikation "x * y" imax*jmax - mal ausgeführt. Das ist nicht notwendig, da bei dieser Auswertung die Lauf-Variablen "i" und "j" keine Rolle spielen. Die rechte Version ist folglich schneller.

```
...
FOR i:=1 TO imax DO
  FOR j:=1 TO jmax DO
    a[i,j]:=b[i,j] + d/FLOAT(i) + d/FLOAT(k)
  END (*FOR*)
END (*FOR*);
```

Bei der Auswertung von "d/FLOAT(k)" sind weder "i" noch "j" im Spiel, und bei der Auswertung von "d/FLOAT(i)" ist "j" nicht beteiligt. Also bietet sich die folgende Verbesserung an (vgl. linkes Fragment):

```
...
dk:=d/FLOAT(k);
FOR i:=1 TO imax DO
  di:=d/FLOAT(i);
  FOR j:=1 TO jmax DO
    a[i,j]:=b[i,j] + di + dk
  END (*FOR*)
END (*FOR*);
```

```
...
dk:=d/FLOAT(k);
FOR i:=1 TO imax DO
  di:=d/FLOAT(i);
  dik:=di + dk;
  FOR j:=1 TO jmax DO
    a[i,j]:=b[i,j] + dik
  END (*FOR*)
END (*FOR*);
```

Eine weitere Verbesserung liefert hier die rechte Version.

Auch durch das "Ausklammern" der Überprüfung von Bedingungen können FOR-Anweisungen beschleunigt werden.

Beispiel 3:

```
...
FOR i:=1 TO imax DO
  IF j > 0 THEN
    x[i]:=a[i] - b[i]
  ELSE
    x[i]:=a[i] + b[i]
  END (*IF*)
END (*FOR*);
```

```
...
IF j > 0 THEN
  FOR i:=1 TO imax DO
    x[i]:=a[i] - b[i]
  END (*FOR*)
ELSE
  FOR i:=1 TO imax DO
    x[i]:=a[i] + b[i]
  END (*FOR*)
END (*IF*);
```

Mit der im rechten Fragment angebrachten Modifikation wird die Bedingung "j > 0" nur einmal getestet.

Die Anwendung auch der beiden folgenden Regeln soll den durch Schleifenköpfe verursachten Overhead verringern:

REGEL: Verkürze Schleifen, wenn möglich.

Beispiel: Die rechte FOR-Anweisung kann - wie links angegeben - "verkürzt" werden:

```
...
FOR i:=1 TO 1000 DO
  a[i]:=0.0
END (*FOR*);
```

```
...
FOR i:=1 TO 1000 BY 2 DO
  a[i]:=0.0;
  a[i+1]:=0.0
END (*FOR*);
```

REGEL: Fasse Schleifen zusammen, wenn möglich.

Beispiel: Die Anzahl der Schleifendurchläufe zur Initialisierung der ARRAYs "x" und "y" wird durch eine einzige FOR-Anweisung (rechts) halbiert.

```
...
FOR i:=1 TO imax DO
  x[i]:=0.0
END (*FOR*);
FOR i:=1 TO imax DO
  y[i]:=0.0
END (*FOR*);
```

```
...
FOR i:=1 TO imax DO
  x[i]:=0.0
  y[i]:=0.0
END (*FOR*);
```

(iii) Selektions-Anweisungen

REGEL: Verwende das "IF ... THEN ... ELSIF ... ELSE" - Konstrukt, wenn möglich.

Es seien "b1", "b2" und "b3" boolesche Ausdrücke. Mit dem Programmfragment

```
...
IF b1 THEN a:=b END (*IF*);
IF b2 THEN c:=d END (*IF*);
IF b3 THEN e:=f END (*IF*);
```

werden alle drei Bedingungen überprüft. Das kann sinnvoll und richtig sein, wenn sich diese Bedingungen "nicht gegenseitig" ausschließen. Gilt aber "bi AND bj =FALSE falls i <> j", so ist das Fragment abzuändern in:

```
...
IF b1 THEN
  a:=b
ELSIF b2 THEN
  c:=d
ELSIF b3 THEN
  e:=f
...
ELSE
  ...
END (*IF*);
```

Wie bereits bekannt, wird die Ausführung der IF-Anweisung abgebrochen, wenn derjenige Zweig bearbeitet ist, der zur ersten zu TRUE ausgewerteten Bedingung gehört. Weitere Bedingungen werden also nicht getestet. Dies legt die folgende Regel nahe:

REGEL: Wähle in geschachtelten "IF ... THEN ... ELSE" - Anweisungen die Reihenfolge so, daß die wahrscheinlichsten Bedingungen zuerst getestet werden.

Diese Regel ist sicher nicht einfach zu befolgen. Sie setzt voraus, daß zum Beispiel Meßergebnisse über relative Häufigkeiten vorliegen, mit denen einzelne Bedingungen "wahr werden".

(iv) Boolesche Ausdrücke

REGEL: Ordne in booleschen Ausdrücken die "atomaren" Bedingungen entsprechend ihrer Wahrscheinlichkeit.

Viele Compiler produzieren zu booleschen Ausdrücken Maschinen-Code, der dafür sorgt, daß die Auswertung eines Ausdrucks nicht fortgesetzt wird, wenn sein Wert bereits feststeht. So würde die von links nach rechts vorgenommene Auswertung von

"b1 OR b2 OR b3"

abgebrochen, wenn der erste Teilausdruck (b1 oder b2) als TRUE erkannt wird. Entsprechend würde die Auswertung von

"b1 AND b2 AND b3"

nach dem ersten als FALSE erkannten Teilausdruck beendet.

Natürlich hat man hier hinsichtlich der Anordnung der Teilausdrücke (gemäß der Wahrscheinlichkeit ihrer Werte) die gleichen Schwierigkeiten wie bei der vorigen Regel.

Andere Compiler haben die hier zur Effizienz-Verbesserung ausgenutzte Eigenschaft nicht. In diesem Fall kann etwa die folgende Regel zur Anwendung kommen:

REGEL: Löse "AND-Verknüpfungen" auf.

Dies bedeutet, daß zum Beispiel die Anweisung

```
...
IF (b1 AND b2 AND b3) THEN
  x:=a
END (*IF*);
```

zu

```
...
IF b1 THEN
  IF b2 THEN
    IF b3 THEN
      x:=a
    END (*IF*)
  END (*IF*)
END (*IF*);
```

wird. Dabei sollten die "unwahrscheinlichsten" Bedingungen zuerst getestet werden. Natürlich widerspricht diese Umwandlung der in Abschnitt 2.2.2 zitierten Regel, gemäß der "...THEN IF ..." - Konstruktionen zu vermeiden sind. In diesem einfachen Fall mag diese Konstruktion freilich noch gut überschaubar und daher vertretbar sein. Im übrigen ist die im Zusammenhang mit der genannten Stil-Regel erläuterte Transformation in eine besser lesbare "IF ... THEN ... ELSIF ..."-Anweisung natürlich auch hier anwendbar. (Zur Übung möge der Leser diese Transformation selbst ausführen.) Man sollte sich aber bewußt sein, daß bei derartigen Umwandlungen boolescher Ausdrücke, die in "IF ... THEN ... ELSE" - Anweisungen auftreten, die Verständlichkeit des Programmtextes (d.h. sein Bezug zur jeweils gegebenen Problemstellung) unter Umständen stark gemindert wird.

(v) Indizierung

Indizierung kann (vgl. Abschnitt 2.4) als Operation des Zugriffs auf die Teilobjekte eines ARRAY-Objekts aufgefaßt werden. Ist "a" eine Variable vom ARRAY-Typ, so ist also "a[i]" ein Ausdruck, dessen Auswertung das im i-ten "Teilbehälter" von "a" enthaltene Objekt liefert. Infolgedessen gelten die Regeln über mehrfach benutzte Ausdrücke und wiederholte Teilausdrücke entsprechend. Einige Beispiele:

"x:=(a[i] + 1.0/a[i]) + a[i];"

wird durch Elimination wiederholter Teilausdrücke zu

```
"ai:=a[i];
 x:=(ai + 1.0/ai) + ai;".
```

```
...
FOR i:=1 TO imax DO
  FOR k:=1 TO kmax DO
    b[k]:=b[k] + a[i]
  END (*FOR*)
END (*FOR*);
```

wird durch "Ausklammern" von "a[i]" zu:

```
...
FOR i:=1 TO imax DO
  ai:=a[i];
  FOR k:=1 TO kmax DO
    b[k]:=b[k] + ai
  END (*FOR*)
END (*FOR*);
```

Eine weitere Variante des "Ausklammerns" stellt das folgende Beispiel dar, in dem rechts die Zahl der Zugriffe auf "a[i,j]" auf den "kmax-ten Teil" reduziert wird:

```
...
FOR i:=1 TO imax DO
  FOR j:=1 TO jmax DO
    a[i,j]:=0.0;
    FOR k:=1 TO kmax DO
      a[i,j]:=a[i,j] + b[i,k] + c[k,j]
    END (*FOR*)
  END (*FOR*)
END (*FOR*);
```

```
...
FOR i:=1 TO imax DO
  FOR j:=1 TO jmax DO
    t:=0.0;
    FOR k:=1 TO kmax DO
      t:=t + b[i,k] + c[k,j]
    END (*FOR*);
    a[i,j]:=t
  END (*FOR*)
END (*FOR*);
```

REGEL: Ersetze mehrdimensionale Arrays durch eindimensionale.

Mehrdimensionale Felder bedingen einen größeren Zugriffsaufwand als eindimensionale. Unter Umständen kann also eine Effizienz-Verbesserung erreicht werden, wenn man Daten, die - anwendungsbedingt - eigentlich in mehrdimensionalen Arrays zu speichern wären, eindimensional anordnet. Diese Regel exzessiv anzuwenden, ist jedoch gefährlich. Man verzichtet ja damit auf den Komfort, den höhere Programmiersprachen für die möglichst anwendungsnahe Repräsentation problembezogener Daten bieten. Man verschlechtert also die Verstehbarkeit eines Programms zugunsten eines unter Umständen nur geringen Effizienzgewinns.

(vi) Prozeduraufrufe

REGEL: Ersetze Prozeduraufrufe durch "inline"-Kodierung des Prozedur-Rumpfes.

Auch diese Regel ist mit "großer Vorsicht zu genießen". Prozeduren zu schreiben, ist schließlich eine der besten Möglichkeiten, einen Programmtext zu strukturieren. Man sollte sich dieses Vorteils nicht ohne Not begeben.

Immerhin bedingt der Einsatz von Prozeduren einigen Aufwand an Maschinen-Instruktionen beim Prozeduraufruf. So kann es sich - insbesondere "im Innern" geschachtelter Iterations-Anweisungen - lohnen, einen Prozeduraufruf durch den Prozedur-Rumpf zu ersetzen:

```
MODULE Xyz.
...
PROCEDURE abc(...);
BEGIN
  (*Anweisungen von abc*)
END abc;
...
BEGIN
  ...
  FOR ... DO
    FOR ... DO
      abc(...);   <-- wird ersetzt durch  (*Anweisungen von abc*)
    END (*FOR*)
  END (*FOR*)
  ...
END Xyz.
```

Derartige Ersetzungen sind jedoch besonders sorgfältig zu dokumentieren.

Literatur zu Kapitel 2

[KEP] Kernighan, B. W. and P.J. Plauger: The Elements of Programming Style; McGraw-Hill; New York; 1978

[LNH] Ledgard, H. F.; Nagin, P. A. und J. F. Hueras: Pascal with Style - Programming Proverbs; Hayden Book Company; Rochelle Park; 1981

[OTW] Ottmann, T. und P. Widmayer: Algorithmen und Datenstrukturen; Bibliographisches Institut; Mannheim; 1990

[VAN] Van Tassel, D.: Program Style, Design, Efficiency, Debugging, and Testing; Prentice-Hall; Englewood Cliffs; 1978

[WEG] Wegener, I.: Theoretische Informatik; Teubner Verlag; Stuttgart; 1993

[WI1] Wirth, N.: Programming in Modula-2; Springer Verlag; Berlin, Heidelberg, New York; 1988

3 Schrittweise Verfeinerung

"Schrittweise Verfeinerung" als systematische Vorgehensweise bei der Konstruktion von Programmen wurde erstmals explizit (und unter dieser Bezeichnung) erläutert in [WI2]. Die Grundidee ist außerordentlich einfach und naheliegend. Beginnend mit einer verbal gegebenen Aufgabenstellung werden - zunächst ebenfalls verbal - sukzessive Aktionen - definiert, durch deren Ausführung die gestellte Aufgabe erledigt wird. "Sukzessive" bedeutet, daß eine einmal beschriebene Aktion wiederum aufgebrochen werden kann in "noch elementarere" Aktionen. Das Verfahren endet mit Aktionen, die unmittelbar durch Anweisungen in der Programmiersprache ausgelöst werden können, also durch die Auswertung von Ausdrücken, durch Zuweisungen und durch Prozeduraufrufe, deren Abfolge durch Iterationen und Selektionen gesteuert wird.

"So macht es doch jeder vernünftige Programmierer!" wird man einwenden, was ist daran also bemerkenswert und eines ganzen Kapitels würdig? Leider entspricht dieser Einwand nicht den Tatsachen. Dies wird durch die tägliche Praxis in Programmier-Abteilungen und Software-Unternehmen vielfach belegt. Programmierer neigen sehr häufig dazu, eine Lösungsidee sofort als Programm-Code zu notieren; sie scheuen davor zurück, ihre Ansätze zunächst verbal zu formulieren und auf ihre Stichhaltigkeit hin zu überprüfen. Dieser ungestüme "Drang zur Maschine" ist falsch. Für ein solches Verhalten gibt es sicher verschiedene Gründe. Nicht zuletzt kann durch uneinsichtiges Management der Eindruck entstehen, daß die Geschwindigkeit, mit der ein Software-Produkt fertiggestellt wird, höher zu bewerten ist als die Solidität des Produktes. Leider müssen wir auf eine, den Rahmen dieses Buches sprengende Diskussion solcher, eher unter der Rubrik "Psychologie des Programmierens" einzuordnender Probleme, verzichten.

Immerhin, den Regeln, die wir in Kapitel 2 behandelt haben, wäre hier ein "oberstes *Gebot*" hinzuzufügen:

"ZUERST (schriftlich!) DENKEN,
DANN PROGRAMM-CODE SCHREIBEN!"

Eine der möglichen "Ausführungsbestimmungen" dieses Gebots heißt "Schrittweise Verfeinerung".

Nach einer weiteren Diskussion werden wir "Schrittweise Verfeinerung" in diesem Kapitel zunächst an zwei kleinen Beispielen erproben. Wir werden dann zwei halbgraphische Notationen einführen, die für das "schriftliche Denken" vor der eigentlichen Programmierung (in der Zielsprache) und für die Dokumentation des Detail-Entwurfs besonders geeignet scheinen. Wir werden hier - wiederum anhand zweier Beispiele - "im Kleinen" demonstrieren, was oft als Entwurf "von oben nach unten" ("*Top Down*") bezeichnet wird, und dabei den Gebrauch beider Notationen einüben.

3.1 Schrittweise Verfeinerung als Entwurfstechnik

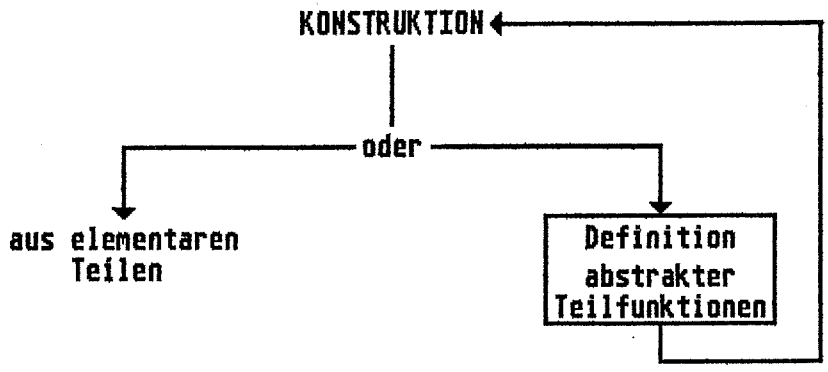
"Schrittweise Verfeinerung" läßt sich - abgesehen von der Demonstration durch Beispiele - vielleicht am besten mit Hilfe des sogenannten "*Black-Box*"-Modells verständlich machen.

Die meisten technischen Geräte, mit denen wir es im Alltag zu tun haben, sind für uns solche "schwarze Kästen"; "schwarz" deshalb, weil wir nicht in ihr Inneres sehen können (oder wollen). Es befinden sich im allgemeinen von außen zugängliche Schalter, Knöpfe oder Bedienungshebel an den Maschinen, und wir vertrauen darauf, daß deren Betätigung ein bestimmtes gewünschtes Verhalten hervorruft: daß Musik aus einem Lautsprecher ertönt, daß eine Herdplatte heiß wird, oder daß ein Wagen zu rollen beginnt.

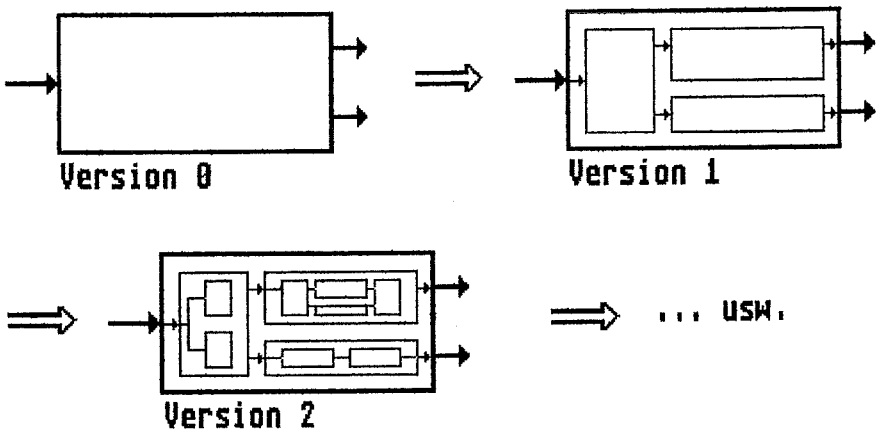
Solche Maschinen sind (obwohl man sie meist sinnlich wahrnehmen kann) für uns, als ihre Benutzer, "*Abstrakt*" in dem Sinne, daß wir uns nicht für ihre einzelnen Teile und deren Zusammenspiel interessieren, sondern lediglich dafür, ob sie nach spezifischen Aktionen unsererseits die erwarteten Reaktionen zeigen, ob sie also definierte "*Funktionen*" erfüllen.

Dem Konstrukteur einer Maschine ist eine solche *abstrakte Funktion* vorgegeben, und seine Aufgabe ist es, mit den ihm zur Verfügung stehenden Mitteln Teile herzustellen, zusammenzubauen und ihr Zusammenspiel so zu organisieren, daß die Funktion realisiert wird. Er wird dabei zweckmäßigerweise so vorgehen, daß er sein Problem zunächst auf die Verknüpfung einiger weniger *abstrakter Teilfunktionen* reduziert. Diese Teilfunktionen entsprechen ihrerseits kleineren, mit passenden Verbindungen im ursprünglichen "schwarzen Kasten" einzubauenden "schwarzen Kästen". Alle Teilfunktionen (bzw. die zugehörigen "schwarzen Kästen") wird er in dieser Weise so lange behandeln, bis er jeweils genau sagen kann, wie eine Teilfunktion durch die elementaren Teile aus seinem "Baukasten" zu realisieren ist. Aus dem ursprünglichen "schwarzen Kasten" ist dann eine konkrete Maschine geworden.

Im Prinzip läuft die Konstruktion also nach folgendem Schema ab:



Und das dabei entstehende Produkt nimmt immer deutlichere Gestalt an:



Die Definition von Teilfunktionen kann natürlich nicht nach Gutdünken vorgenommen werden. Vielmehr erweisen sich hier Geschick, Wissen und Kreativität des Konstrukteurs. Bei jeder weiteren Untergliederung muß er sich im allgemeinen für eine von mehreren Möglichkeiten entscheiden. Dabei wird er sich vermutlich bemühen, die konkrete Ausformung von Details so weit wie möglich hinauszuschieben, um sich den Spielraum für später zu fällende Entscheidungen nicht von vornherein zu verengen. Beispielsweise wird sich der Konstrukteur eines Motors nicht zu früh auf ein bestimmtes Material für Zylinderköpfe festlegen. Eine solche frühe Entscheidung zwänge ihn vielleicht dazu, bei der Konstruktion anderer Teile immer gewisse Einschränkungen in Kauf zu nehmen. Ein Architekt wird bei der Planung des Grundrisses eines Hauses noch nicht an dessen elektrische Verkabelung denken. Usw.

Übertragen auf den Software-Entwickler heißt dies: Die Entscheidung für oder gegen die Verwendung bestimmter Repräsentationen von Daten-Objekten sollte so spät wie möglich fallen. Wird beispielsweise ein Programm zur Verwaltung einer Namensmenge (etwa der Mitglieder eines Vereins) durch schrittweise Verfeinerung konstruiert, so sollte bei der Formulierung von Teilfunktionen so lange wie möglich eben von "Namensmengen" die Rede sein und nicht von "Arrays, Records oder Listen".

Die Zurückhaltung des Motor-Konstrukteurs führt dazu, daß man bei der Herstellung von Zylinderköpfen relativ problemlos auf andere Materialien zurückgreifen kann, wenn sich dies als vorteilhaft erweisen sollte. Der Verzicht des Architekten auf eine Festlegung der elektrischen Infrastruktur eines Hauses bedeutet, daß er diese Aufgabe getrost einem anderen überlassen kann. (Dem muß er lediglich sagen, wo die Lampen und Steckdosen anzubringen sind.)

Entsprechend erleichtert die möglichst späte Entscheidung für eine bestimmte Datenstruktur zur Repräsentation von Objekten bzw. Objektmengen die Ver-

wendung einer anderen Datenstruktur, falls dies aus irgendwelchen Gründen (z.B. zwecks Verbesserung der Effizienz) ratsam sein sollte. Es gilt:

> "Je später die Entscheidung für eine konkrete Repräsentation fällt, umso leichter kann sie zurückgenommen werden, ohne das bereits vorher Gedachte zu verwerfen."

Zweitens kann der Verzicht des Software-Entwicklers auf die sofortige Ausarbeitung von Details die Arbeitsteilung begünstigen. Er findet zum Beispiel, daß er für die Lösung seiner Aufgabe eine bestimmte Prozedur benötigt, die eine wohldefinierte Teilaufgabe erledigt. Er überläßt deren Ausformung (und alle dazu notwendigen "Repräsentations-Entscheidungen" !) seinem Kollegen und ist fertig.

Wir sind mit diesen wenigen Bemerkungen zur "Schrittweisen Verfeinerung" unversehens tief in die Thematik des *Software-Entwurfs* hineingeraten. Dabei müssen wir uns darauf besinnen, daß wir uns für diesen Teil lediglich die "Programmierung im Kleinen" vorgenommen haben. Doch auch hier gelten Prinzipien, deren Anwendung bei der Entwicklung großer Software-Systeme völlig unverzichtbar ist, wenn bestimmte allgemeine Qualitäts-Anforderungen (auf die wir in Kapitel 6 zu sprechen kommen) zu erfüllen sind. Was also bisher nur anklang:

- das Prinzip der *Abstraktion* ("Beschreibe Teilfunktionen ohne Rücksicht auf konkrete Repräsentationen!") und
- das (damit zusammenhängende) Prinzip der *Lokalität* ("Begrenze die Auswirkungen bestimmter Entscheidungen auf einen kleinen Teil des Programms, z. B. eine Prozedur!")

werden zu den in Kapitel 6 entscheidende Rollen spielen. Hier beschränken wir uns auf einige Beispiele zur Illustration der Anwendung dieser Prinzipien bei der Implementierung von kleinen, überschaubaren Programm- bzw. System-Funktionen.

3.2 Zwei Beispiele

Die erste Prozedur, die wir in diesem Abschnitt mittels schrittweiser Verfeinerung konstruieren, mag innerhalb eines Programms zur Verwaltung eines Terminkalenders ihren Platz haben. Sie soll zu einem gegebenem Tagesdatum das Datum des Folgetages ermitteln. Das Verwaltungsprogramm selbst ist vielleicht Teil eines "Büro-Automations-Systems".

Zweitens werden wir das "klassische" Problem der konfliktfreien Positionierung von acht Damen auf einem Schachbrett behandeln. Hierfür entwickeln wir - schrittweise verfeinernd - zwei verschiedene Lösungen, eine "iterative" und

eine "rekursive". Wir werden dabei sehen, daß dieses Problem zu einer großen Klasse interessanter Probleme gehört, für die unsere Lösungen ohne weiteres verallgemeinert werden können.

Das erste Beispiel haben wir in Anlehnung an [WOO] ausgearbeitet; das zweite ist in der Literatur vielfach behandelt worden. Unsere MODULA-2 Version orientiert sich allerdings ebenfalls an [WOO].

3.2.1 Folgedatum

Wir nehmen an, daß durch die Analyse des Umfelds, in dem jenes Programm zur Verwaltung eines Terminkalenders eingesetzt werden soll, bereits Antworten auf eine Reihe von Fragen gefunden (und in einer Anforderungs-Spezifikation formuliert) wurden, die für die Herstellung der angekündigten Prozedur von Interesse sind.

Der Auftrag an einen Entwickler möge also lauten:

> "Es ist eine Prozedur zu schreiben, die
> - zu einer Datumsangabe
> - die dem Datum des folgenden Tages entsprechende Datumsangabe ermittelt.
>
> Anzuwenden ist dabei der "Gregorianische Kalender". Es sollen nur gemäß diesem Kalender gültige Datumsangaben vom 15. Oktober 1582 bis zum 31. Dezember 2099 (jeweils einschließlich)
>
> Die Übergabe einer gemäß diesem Kalender ungültigen Datumsangabe durch einen aufrufenden Programmteil sollte diesem in ge-
>
> Die konkrete Datenstruktur zur Repräsentation von Datumsangaben ist von außen nicht vorgegeben und kann vom Entwickler selbst festgelegt werden."

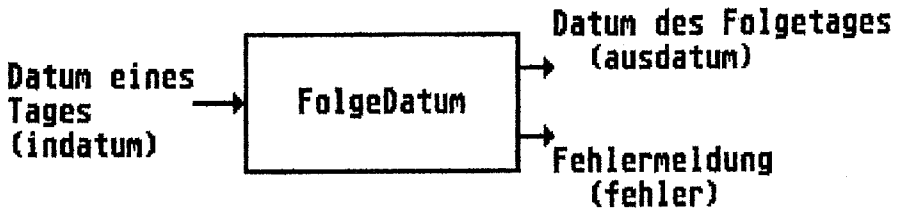
Unter "Datumsangabe" verstehen wir hier selbstverständlich die eindeutige Identifikation eines gemeinhin als "Tag" vereinbarten Zeitraums von 24 Stunden, beginnend und endend jeweils um Mitternacht.

(Wir bemerken, daß unsere Aufgabenbeschreibung dem Entwickler einige Freiheiten hinsichtlich der Ausgestaltung der "Schnittstelle" zwischen der Prozedur und "ihrer Umgebung" läßt. Wir setzen diese Freiheiten hier aus didaktischen Gründen voraus, um beim nachfolgenden Entwurf einen möglichst großen Spielraum zu haben. Unter realen (Software-)Produktionsbedingungen wäre einem Programmierer die Parameterliste der Prozedur vermutlich fest vorgegeben.)

Die gewünschte Prozedur erhält den Namen "FolgeDatum". Eine erste Version, die lediglich aus dem Prozedurkopf und dem einleitenden Kommentar (dem "Prolog", vgl. Abschnitt 2.2.1) besteht, kann aufgrund der Aufgabenstellung sofort notiert werden:

```
PROCEDURE FolgeDatum((*IN*) indatum:DatumsTyp;
                            (*OUT*) VAR ausdatum: DatumsTyp;
                            (*OUT*) VAR fehler: BOOLEAN);
(*Version 1*)
(*Beim Eintritt enthält "indatum" eine Datumsangabe.
Falls "indatum" gemäß Gregorianischem Kalender gültig ist und
zwischen dem 15. Oktober 1582 und dem 31. Dezember 2099
(jeweils einschließlich) liegt, so enthält "ausdatum" bei
Prozedurende das Datum des Folgetages. In diesem Fall enthält
"fehler" den Wert FALSE. Andernfalls enthält "fehler" bei
Prozedurende den Wert "TRUE"*)
BEGIN
END FolgeDatum;
```

Dieser Prozedur entspricht im "Black-Box"-Modell das folgende Bild:



Die in diesen Kasten hinein- bzw. hinausführenden Pfeile sind also den Parametern der Prozedur zugeordnet:

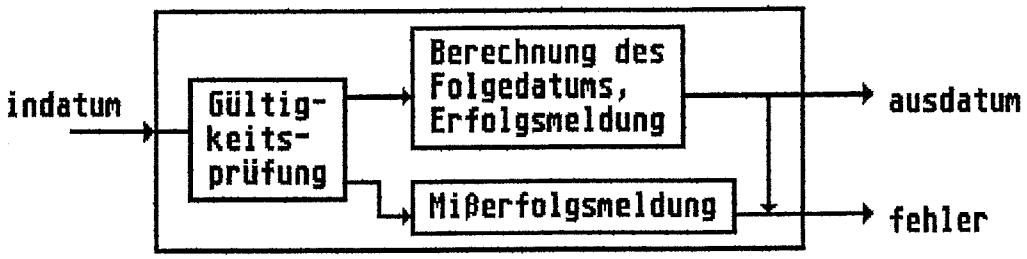
- dem Eingangspfeil "Datum eines Tages" der Wertparameter "indatum",
- dem Ausgangspfeil "Datum des Folgetages" der Variablen- Parameter "ausdatum" und
- dem Ausgangspfeil "Fehlermeldung" der Variablen-Parameter "fehler".

Gäbe es in MODULA-2 einen (vordefinierten) Datentyp "DATUMSTYP" und eine Anweisung, welche genau den von der Aufgabe geforderten Effekt hat, so wären wir fertig. Da dies jedoch nicht der Fall ist, sind wir gezwungen, den "schwarzen Kasten" durch eine weitere Zerlegung zu füllen. Bevor wir dies tun, sei darauf hingewiesen, daß wir mit der ersten Formulierung unserer Prozedur

- bereits einen syntaktisch korrekten Programmtext in MODULA-2 geschrieben haben;
- eine "Repräsentations-Entscheidung" bereits gefällt haben, indem wir zur Information über die Gültigkeit des Werts von "indatum" einen Variablen-Parameter vom Typ BOOLEAN benutzen;

- eine weitere "Repräsentations-Entscheidung", nämlich die bezüglich der Struktur von "DatumsTyp" noch nicht gefällt haben.

Unsere erste (und hoffentlich geschickte!) "Zerlegungs-Entscheidung" trennt die Überprüfung der Gültigkeit von "indatum" von der Berechnung des in "ausdatum" zu übergebenden Wertes. Der "schwarze Kasten" wird ein wenig durchsichtiger:



Mit dieser "Zerlegungs-Entscheidung" postulieren wir also die Existenz einer Funktion

DatumIstGueltig: DatumsTyp -> BOOLEAN

mit

$$\text{DatumIstGueltig(indatum)} = \begin{cases} \text{TRUE, falls indatum gültig} \\ \text{FALSE, sonst.} \end{cases}$$

Wir gelangen so zu der folgenden zweiten Version der Prozedur "FolgeDatum":

```
PROCEDURE FolgeDatum((*IN*) indatum:DatumsTyp;
                             (*OUT*) VAR ausdatum: DatumsTyp;
                             (*OUT*) VAR fehler: BOOLEAN);
(*Version 2*)
(* Prolog wie oben *)
PROCEDURE DatumIstGültig(datum:DatumsTyp): BOOLEAN;
(* DatumIstGültig liefert den Wert TRUE, falls "datum" eine
gemäß dem Gregorianischen Kalender gültige Datumsangabe
enthält; andernfalls liefert die Prozedur den Wert FALSE *)
BEGIN (*DatumIstGueltig*)
END DatumsIstGueltig;
```

```
BEGIN (*FolgeDatum*)
  CASE DatumIstGueltig(indatum) OF
    TRUE: (*Berechne ausdatum*)
          (*Erfolgsmeldung*)     |
    FALSE:(*Mißerfolgsmeldung*)
  END (*CASE*)
END FolgeDatum;
```

Drei Bemerkungen sind an dieser Stelle angebracht:

(i) Bei der Umsetzung der ersten Verfeinerung des "schwarzen Kastens" in die obige Version der Prozedur "FolgeDatum" haben wir offensichtlich einen "gedanklichen Sprung" gemacht. Das "Black-Box"-Modell zeigt nämlich lediglich die möglichen Wege, welche Daten innerhalb des "Kastens" nehmen können (den "Datenfluß"), nicht jedoch die Bedingungen, unter denen der eine oder der andere Weg gewählt wird. Dies macht die Grenzen des "Black-Box"-Modells für die Darstellung der internen Struktur eines Programms deutlich. Wir werden diese Darstellungsweise daher im folgenden zunächst aufgeben.

(ii) Anstatt der CASE-Anweisung im Rumpf der Prozedur "FolgeDatum" hätten wir selbstverständlich, da nur die Fälle TRUE und FALSE zu unterscheiden sind, auch eine "IF...THEN...ELSE"-Anweisung verwenden können. Daß wir es hier nicht getan haben, geschah aus rein stilistischen Gründen: Da wir noch nicht wissen, ob nicht die Berechnung von "ausdatum" eine weitere Fallunterscheidung erfordert, verhindern wir so die Entstehung einer "IF... THEN IF..."-Konstruktion und befolgen damit die entsprechende Regel aus Abschnitt 2.2.2.

(iii) Wir haben, wie bereits angekündigt, Aktionen, die zur Erledigung der gestellten Aufgabe auszuführen sind, verbal beschrieben und diese Beschreibungen als Kommentare in strukturbildenden Anweisungen eingebettet. Wir werden diese Kommentare in weiteren Versionen immer mitführen. Damit befolgen wir Teil c) der Kommentierungsregel aus Abschnitt 2.2.1.

Die weitere Zerlegung von (*Berechne ausdatum*) liegt nahe: Nur falls "indatum" eine Datumsangabe enthält, die einen am Monatsende liegenden Tag identifiziert, ist die Berechnung von "ausdatum" offenbar etwas problematisch. Es scheint also sinnvoll, eine einfache Möglichkeit zu schaffen, um festzustellen, ob der mit "indatum" übergebene Wert diese Eigenschaft besitzt oder nicht. Wir postulieren also wiederum die Existenz einer Funktion

IstLetzterTag: DatumsTyp -> BOOLEAN

mit

IstLetzterTag(indatum) = ⌈ TRUE, falls indatum den letzten Tag eines Monats bezeichnet
⌊ FALSE, sonst

und dokumentieren diesen Verfeinerungsschritt durch die dritte Version der Prozedur "FolgeDatum":

```
PROCEDURE FolgeDatum((*IN*)  indatum:DatumsTyp;
                             (*OUT*) VAR ausdatum: DatumsTyp;
                             (*OUT*) VAR fehler: BOOLEAN);
(*Version 3*)
(* Prolog wie oben *)

PROCEDURE DatumIstGültig(datum:DatumsTyp): BOOLEAN;
(* Prolog wie oben *)
BEGIN END DatumsIstGueltig;
PROCEDURE IstLetzterTag(datum:DatumsTyp): BOOLEAN;
(*Bei Eintritt enthält "datum" eine gemäß Gregorianischem
Kalender gültige Datumsangabe. Die Prozedur liefert den Wert
TRUE, falls der Wert in "datum" den letzten Tag eines Monats
identifiziert und andernfalls den Wert FALSE*)
BEGIN END IstLetzterTag;
BEGIN (*FolgeDatum*)
  CASE DatumIstGueltig(indatum) OF
    TRUE: (*Berechne ausdatum*)
          IF IstLetzterTag(indatum) THEN
            (*erster Tag des nächsten Monats oder
            nächsten Jahres*)
          ELSE
            (*nächster Tag, gleicher Monat,
            gleiches Jahr*)
          END (*IF*);
          (*Erfolgsmeldung*)     |
    FALSE:(*Mißerfolgsmeldung*)
  END (*CASE*)
END FolgeDatum;
```

Nunmehr ist die Entscheidung für eine geeignete Struktur des Typs "DatumsTyp" fällig, da ohne eine solche "Repräsentations-Entscheidung" keine weitere Verfeinerung der in Version 3 beschriebenen Aktionen möglich ist.

Wir definieren daher:

```
TYPE MonatsTyp = (januar,februar,maerz,april,mai,juni,
                  juli,august,september,oktober,november,
                  dezember);
```

und

```
TYPE DatumsTyp = RECORD
                   tag: (1..31);
                   monat: MonatsTyp;
                   jahr: (1582..2099)
                 END;
```

Man mache sich klar, daß andere "Repräsentations-Entscheidungen" als diese (naheliegende!) möglich gewesen wären. Wir sollten deshalb schon einige Worte zur Begründung verlieren:

(i) Durch die Wahl der Subtypen (1..31) und (1582..2099) für "tag" bzw. "jahr" sind zugleich die zulässigen Wertebereiche für diese Komponenten einer Datumsangabe dokumentiert.

(i) Entsprechendes wird durch die Wahl des Enumerationstyps "MonatsTyp" für die Komponente "monat" erreicht. Mit dieser Festlegung kann der MODULA 2-Compiler darüberhinaus verhindern, daß eine Variable vom Datums-Typ eine "monat"-Komponente enthält, die nicht im zulässigen Wertebereich liegt.

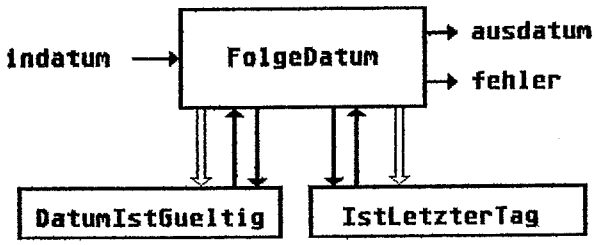
Die in Version 3 beschriebenen Aktionen können nun unmittelbar durch MODULA 2-Anweisungen ausgelöst werden; wir erhalten also als vierte Version:

```
PROCEDURE FolgeDatum((*IN*) indatum:DatumsTyp;
                             (*OUT*) VAR ausdatum: DatumsTyp;
                             (*OUT*) VAR fehler: BOOLEAN);
(*Version 4*)
(* Prolog wie oben *)
PROCEDURE DatumIstGültig(datum:DatumsTyp): BOOLEAN;
(* Prolog wie oben *)
BEGIN END DatumsIstGueltig;
PROCEDURE IstLetzterTag(datum:DatumsTyp): BOOLEAN;
(* Prolog wie oben *)
BEGIN END IstLetzterTag;
BEGIN (*FolgeDatum*)
 CASE DatumIstGueltig(indatum) OF
  TRUE: (*Berechne ausdatum*)
      IF IstLetzterTag(indatum) THEN
       (*erster Tag des nächsten Monats oder nächsten Jahres*)
```

```
        ausdatum.tag:=1;
        IF indatum.monat=dezember THEN
          ausdatum.jahr:=indatum.jahr + 1;
          ausdatum.monat:=januar
        ELSE
          ausdatum.jahr:=indatum.jahr;
          ausdatum.monat:=indatum.monat;
          INC(ausdatum.monat)
        END (*IF*)
      ELSE
        (*nächster Tag, gleicher Monat, gleiches Jahr*);
        ausdatum.tag:=indatum.tag + 1;
        ausdatum.monat:=indatum.monat;
        ausdatum.jahr:=indatum.jahr
      END (*IF*);
      (*Erfolgsmeldung*)
      fehler:=FALSE |
  FALSE:(*Mißerfolgsmeldung*)
      fehler:=TRUE
  END (*CASE*)
END FolgeDatum;
```

Einen sehr kleinen Schritt haben wir bei diesem Übergang zur vierten Version ausgelassen, indem wir die Unterscheidung gemäß Jahresende nicht eigens dokumentierten. (Dies wäre wohl auch zuviel des Guten!) Gleich miterledigt haben wir auch die Erfolgs- bzw. Mißerfolgsmeldungen durch entsprechende Zuweisungen an die "Output-Variable" "fehler".

Der erteilte Auftrag wäre erledigt, wenn nicht noch die im Laufe der bisherigen Verfeinerungen entstandenen "schwarzen Kästen", die Funktions-Prozeduren "DatumIstGültig" und "IstLetzterTag", transparent zu machen wären. Den derzeitigen Stand der Entwicklung können wir durch die folgende Skizze dokumentieren:



Durch diese Darstellung werden wichtige Beziehungen zwischen der Prozedur "Folgedatum" und den Funktions-Prozeduren "DatumIstGültig" und "IstLetzter-Tag" veranschaulicht:

- Der Doppelpfeil bedeutet "Aufruf"; das heißt, daß die in "DatumIstGültig" bzw. "IstLetzterTag" auszuführenden Aktionen durch die Prozedur "Folge-Datum" angestoßen, "aktiviert" werden.
- Der Einfachpfeil bedeutet "Übergabe von Datenobjekten": An beide Funktions-Prozeduren übergibt "FolgeDatum" ein Objekt in der Variablen "indatum" und erhält jeweils ein Objekt zurück, dessen "Behälter" durch den Namen der aufgerufenen Funktions-Prozedur identifiziert ist.

Im Gegensatz zur weiter oben skizzierten "Black-Box"-Darstellung der ersten Verfeinerung wird hier nicht klar, daß die Funktions-Prozeduren "DatumIst-Gültig" und "IstLetzterTag" auch Teile der Prozedur "FolgeDatum" sind. Im Programmtext kommt dies natürlich dadurch zum Ausdruck, daß die Funktions-Prozeduren "im Scope" von "FolgeDatum" definiert sind. Andererseits werden auch durch diese Visualisierung der Aufruf- und Datenübergabe-Beziehungen nicht die Bedingungen sichtbar, unter denen Aufruf und Datenübergabe erfolgen. Dennoch sind Darstellungen dieser Art - als Abstraktion wichtiger Aspekte der Struktur eines Programms - sehr nützlich. Wir werden ihnen mit mehr Ausführlichkeit in Kapitel 6 wiederbegegnen.

Wenden wir uns nun also der weiteren Verfeinerung der noch unfertigen Funktions-Prozeduren zu.

Die Gültigkeit einer Datumsangabe (tag,monat,jahr) hängt zum einen natürlich davon ab, ob die drei Komponenten jeweils im gültigen Wertebereich liegen. Der für "jahr" anzuwendende Wertebereich ist im übrigen durch die Aufgabenstellung festgelegt. Andererseits wissen wir aufgrund unserer "Allgemeinbildung", daß die möglichen Werte von "tag" vom jeweiligen "monat" abhängen und für den Monat Februar sogar von "jahr". Wir nutzen dieses Wissen für die Verfeinerung der Funktions-Prozedur "DatumIstGültig" aus:

```
PROCEDURE DatumIstGültig(datum:DatumsTyp): BOOLEAN;
(* Prolog wie oben *)
BEGIN
  IF (*tag nicht im Wertebereich*) THEN
    RETURN FALSE
  ELSIF (*jahr nicht im Wertebereich*) THEN
    RETURN FALSE
  ELSIF (*datum ist vor dem 15. Oktober 1582) THEN
    RETURN FALSE
  ELSE
    (*Prüfe, ob tag,monat und jahr zusammenpassen
    und entscheide danach*)
  END (*IF*)
END DatumsIstGueltig;
```

Eine Überprüfung, ob "monat" im korrekten Wertebereich liegt, erübrigt sich, da - wie oben begründet - der MODULA-2 - Compiler keine anderen Werte für "monat" zuläßt als die im Enumerationstyp "MonatsTyp" enthaltenen. Die Überprüfung der übrigen Komponenten hätten wir auch mit einer Bedingung - statt mit vier - formulieren können. Wir hätten damit jedoch Übersichtlichkeit und Klarheit gegen nichts eingetauscht. Die Bedingungen sind, ohne Umwege über weitere Funktions-Prozeduren zu nehmen, einfach zu formulieren. Da sie sich alle auf den in "datum" übergebenen Wert beziehen, betten wir die Gültigkeitsprüfung in eine entsprechende WITH-Anweisung ein.

```
PROCEDURE DatumIstGültig(datum:DatumsTyp): BOOLEAN;
(* Prolog wie oben *)
BEGIN
 WITH datum DO
  IF (tag < 1) OR (tag > 31) THEN
   RETURN FALSE
  ELSIF (jahr < 1582) OR (jahr > 2099) THEN
   RETURN FALSE
  ELSIF (jahr=1582) AND ((monat=oktober) AND (tag < 15)
                    OR (monat < oktober))  THEN
   RETURN FALSE
  ELSE
   (*Prüfe, ob tag,monat und jahr zusammenpassen
   und entscheide danach*)
  END (*IF*)
 END (*WITH*)
END DatumsIstGueltig;
```

Für die Überprüfung, ob "tag", "monat" und "jahr" zusammenpassen, postulieren wir die Existenz einer Funktion

IstSchaltjahr: CARDINAL -> BOOLEAN

mit

$$\text{IstSchaltjahr(jahr)} = \begin{cases} \text{TRUE, falls "jahr" ein} \\ \qquad \text{Schaltjahr bezeichnet} \\ \text{FALSE, sonst} \end{cases}$$

(Möglicherweise müssen wir erst einen Experten nach der Schaltjahr-Regel des Gregorianischen Kalenders fragen!) Damit kann auch der "ELSE-Zweig" ausgearbeitet werden:

```
PROCEDURE DatumIstGültig(datum:DatumsTyp): BOOLEAN;
(* Prolog wie oben *)
BEGIN
 WITH datum DO
```

```
IF (tag < 1) OR (tag > 31) THEN
  RETURN FALSE
ELSIF (jahr < 1582) OR (jahr > 2099) THEN
  RETURN FALSE
ELSIF (jahr=1582) AND ((monat=oktober) AND (tag < 15)
                    OR (monat < oktober))  THEN
  RETURN FALSE
ELSE
  (*Prüfe, ob tag,monat und jahr zusammenpassen
  und entscheide danach*)
  CASE monat OF
    februar: IF IstSchaltjahr(jahr) THEN
               RETURN (tag <= 29)
             ELSE
               RETURN (tag <= 28)
             END (*IF*) |
    april,juni,september,november:
             RETURN (tag <= 30) |
    ELSE
             RETURN (tag <= 31)
  END (*CASE*)
 END (*IF*)
END (*WITH*) END DatumsIstGueltig;
```

Man beachte, daß wir die Funktions-Prozedur "IstSchaltjahr" noch nicht "im Scope" von "DatumIstGültig" deklariert haben. In weiser Voraussicht: Möglicherweise benötigen wir sie ja noch in einem anderen Zusammenhang. Sie wäre dort dann ein zweites Mal zu deklarieren. Und in der Tat bemerken wir, daß auch die Entscheidung darüber, ob eine Datumsangabe den letzten Tag eines Monats identifiziert oder nicht, von der Schaltjahr-Eigenschaft abhängt:

```
PROCEDURE  IstLetzterTag(datum:DatumsTyp): BOOLEAN;
(* Prolog wie oben *)
BEGIN
 WITH datum DO
  CASE monat OF
   februar: IF IstSchaltjahr(jahr) THEN
              RETURN (tag=29)
            ELSE
              RETURN (tag=28)
            END (*IF*) |
   april,juni,september,november:
            RETURN (tag=30) |
```

```
    ELSE
            RETURN (tag=31)
   END (*CASE*)
  END (*WITH*)
END IstLetzterTag;
```

Es ist also - zumindest im Rahmen unserer ursprünglichen Aufgabenstellung - sinnvoll, die Funktions-Prozedur "IstSchaltjahr", welche die Schaltjahr-Eigenschaft entscheidet, "im Scope" der Prozedur "Folgedatum" zu deklarieren. (Sie wird auf diese Weise "allgemein zugänglich"!) Somit lautet die fünfte und (hier) endgültige Version dieser Prozedur:

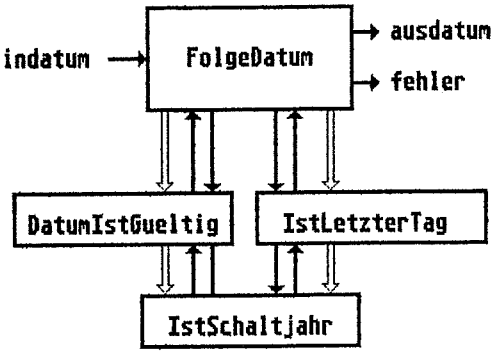
```
PROCEDURE FolgeDatum((*IN*)  indatum:DatumsTyp;
                             (*OUT*) VAR ausdatum: DatumsTyp;
                             (*OUT*) VAR fehler: BOOLEAN);
(*Version 5*)
(*Beim Eintritt enthält "indatum" eine Datumsangabe. Falls
"indatum" gemäß Gregorianischem Kalender gültig ist und
zwischen dem 15. Oktober 1582 und dem 31. Dezember 2099
(jeweils einschließlich) liegt, so enthält "ausdatum" bei
Prozedurende das Datum des Folgetages. In diesem Fall enthält
"fehler" den Wert FALSE. Andernfalls enthält "fehler" bei
Prozedurende den Wert "TRUE"*)
PROCEDURE IstSchaltjahr(jahr:CARDINAL): BOOLEAN;
(*IstSchaltjahr liefert den Wert TRUE, falls der Inhalt von "jahr"
dem Gregorianischen Kalender gemäß ein Schaltjahr bezeichnet
und FALSE sonst*)
BEGIN
 RETURN ((jahr MOD 4 = 0) AND
          (jahr MOD 100 <> 0)) OR
          (jahr MOD 400 = 0)
END IstSchaltjahr;
PROCEDURE DatumIstGültig(datum:DatumsTyp): BOOLEAN;
(* DatumIstGültig liefert den Wert TRUE, falls "datum" eine
gemäß dem Gregorianischen Kalender gültige Datumsangabe
enthält; andernfalls liefert die Prozedur den Wert FALSE *)
BEGIN
 WITH datum DO
  IF (tag < 1) OR (tag > 31) THEN
   RETURN FALSE
  ELSIF (jahr < 1582) OR (jahr > 2099) THEN
   RETURN FALSE
```

```
    ELSIF (jahr=1582) AND ((monat=oktober) AND (tag < 15)
                        OR (monat < oktober))  THEN
      RETURN FALSE
    ELSE
      (*Prüfe, ob tag,monat und jahr zusammenpassen und
      entscheide danach*)
      CASE monat OF
        februar: IF IstSchaltjahr(jahr) THEN
                   RETURN (tag <= 29)
                 ELSE
                   RETURN (tag <= 28)
                 END (*IF*) |
        april,juni,september,november:
                 RETURN (tag <= 30) |
        ELSE
                 RETURN (tag <= 31)
      END (*CASE*)
    END (*IF*)
  END (*WITH*) END DatumsIstGueltig;

PROCEDURE IstLetzterTag(datum:DatumsTyp): BOOLEAN;
(*Bei Eintritt enthält "datum" eine gemäß Gregorianischem
Kalender gültige Datumsangabe. Die Prozedur liefert den Wert
TRUE, falls der Wert in "datum" den letzten Tag eines Monats
identifiziert und andernfalls den Wert FALSE*)
BEGIN
  WITH datum DO
    CASE monat OF
      februar: IF IstSchaltjahr(jahr) THEN
                 RETURN (tag=29)
               ELSE
                 RETURN (tag=28)
               END (*IF*) |
      april,juni,september,november:
               RETURN (tag=30) |
      ELSE
               RETURN (tag=31)
    END (*CASE*)
  END (*WITH*) END IstLetzterTag;
```

```
BEGIN (*FolgeDatum*)
  WITH ausdatum DO
    CASE DatumIstGueltig(indatum) OF
      TRUE: (*Berechne ausdatum*)
          IF IstLetzterTag(indatum) THEN
            (*erster Tag des nächsten Monats oder
            nächsten Jahres*)
            tag:=1;
            IF indatum.monat=dezember THEN
              jahr:=indatum.jahr + 1;
              monat:=januar
            ELSE
              jahr:=indatum.jahr;
              monat:=indatum.monat;
              INC(monat)
            END (*IF*)
          ELSE
            (*nächster Tag, gleicher Monat,
            gleiches Jahr*);
            tag:=indatum.tag + 1;
            monat:=indatum.monat;
            jahr:=indatum.jahr
          END (*IF*);
          (*Erfolgsmeldung*)
          fehler:=FALSE |
      FALSE:(*Mißerfolgsmeldung*)
          fehler:=TRUE
    END (*CASE*)
  END (*WITH*)
END FolgeDatum;
```

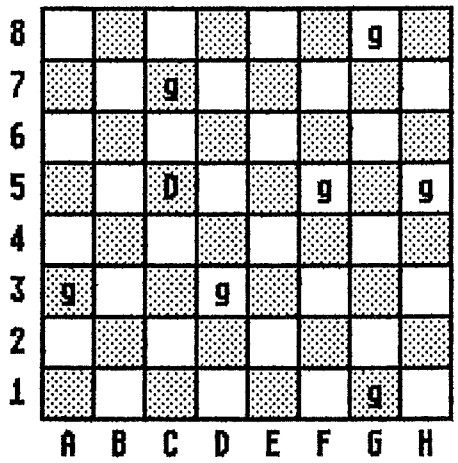
Den Rumpf der Prozedur "FolgeDatum" haben wir hier noch in eine WITH-Anweisung eingebettet, da sich die meisten Zuweisungen auf die Output-Variable "ausdatum" beziehen. Die vollständige "Aufruf-Hierarchie" zeigt das nebenstehende Diagramm:



Mit Fertigstellung der letzten Version sind die Versionen 1 - 4, die lediglich die Etappen der schrittweisen Verfeinerung dokumentierten, obsolet geworden. Version 5 enthält in Form von Kommentaren (fast) alle unsere Überlegungen, die schließlich zum Ziel geführt haben. Sie dokumentiert unseren "Entwurf" und enthält gleichzeitig den zur Lösung der gestellten Aufgabe erforderlichen Programmcode. Man kann sich leicht vorstellen, wie die Entwicklung der Prozedur etwa mit Hilfe eines komfortablen (Bildschirm-orientierten) Editors hätte vonstatten gehen können: Die einzelnen Versionen, die wir zur Verdeutlichung der Vorgehensweise zunächst alle aufbewahrt haben, hätten sukzessive aufeinander aufbauen und sich dabei jeweils ersetzen können.

4.2.2 Acht Damen

Nach uralter Schachregel darf eine Dame jede gegnerische Figur schlagen, die sie auf dem Schachbrett in horizontaler, vertikaler oder diagonaler Richtung direkt im Visier hat:



Die Dame auf Feld C5 bedroht die gegnerischen Figuren auf den Feldern C7, F5, A3 und G1, nicht jedoch die Figuren auf G8, H5 und D3.

(Steht die Dame in einer Ecke, so ist natürlich nur eine Diagonale zu überblikken.)

Die Frage lautet nun: Kann man acht Damen so auf einem Schachbrett positionieren, daß keine eine andere schlagen kann? Solche Positionen, wir nennen sie "friedlich", wollen wir mit Hilfe eines geeigneten Programms finden.

Im Gegensatz zum Beispiel "FolgeDatum" scheint dieses sogenannte "Acht-Damen-Problem" keinerlei praktische Bedeutung zu haben. Warum also, so mag man sich wundern, greifen wir in einem Buch, in dem praktische Fragestellungen im Vordergrund stehen sollen, eine solche offensichtliche Spielerei auf? Hierzu zwei Bemerkungen:

(i) Zunächst geht es uns bei der Auswahl unserer Beispiele nicht primär darum, einem (zwanghaften?) Anspruch auf Praxisnähe zu genügen. Vielmehr sollen

die Beispiele bestimmte Prinzipien, Vorgehensweisen und "(imperative!) Denkmuster" ("Paradigmata") besonders prägnant verdeutlichen.

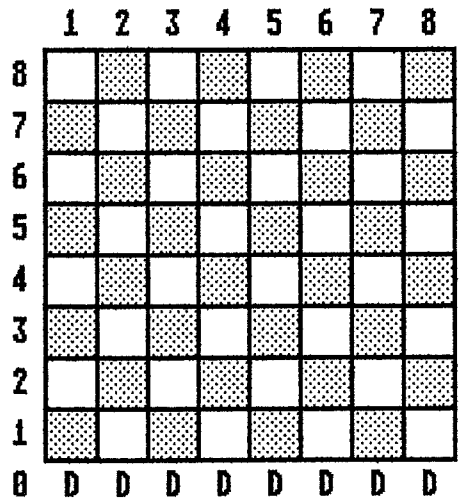
(ii) Bei der Erarbeitung einer Prozedur zur Lösung des "Acht-Damen-Problems" werden wir in der Tat Gelegenheit haben, "zwei Fliegen mit einer Klappe zu schlagen". Schrittweise Verfeinerung wird hier auf einem Schema basieren, das sich ohne weiteres auf viele andere Probleme anwenden läßt, welche durchaus von praktischem Nutzen sind: Es ist ein mögliches "Grundschema" für Abläufe, die zum Beispiel im Kern der immer populärer werdenden sogenannten "Expertensysteme" stattfinden. Wir werden diesen Aspekt der Verallgemeinerbarkeit von Lösungen zu Lösungsmustern weiter unten noch etwas vertiefen.

Doch nun unsere Aufgabe, genauer formuliert:

> Es ist eine Prozedur zu schreiben, die alle Lösungen des "Acht-Damen-Problems" ermittelt und ausgibt.
>
> Ein Feld des Schachbretts werde durch die Angabe von Zeilen-Nummer (im Wertebereich von 1..8) und Spalten-Nummer (der Einfachheit halber ebenfalls im Wertebereich von 1..8) identifiziert.
>
> Eine Lösung liefert für jede Spalte die Zeilen-Nummer des Feldes, auf das eine Dame so gestellt werden kann, daß sie keine andere Dame bedroht.

Bei dieser Formulierung haben wir bereits die triviale Einsicht berücksichtigt, daß - wenn es überhaupt eine Lösung gibt - in jeder Spalte genau eine Dame positioniert werden muß.

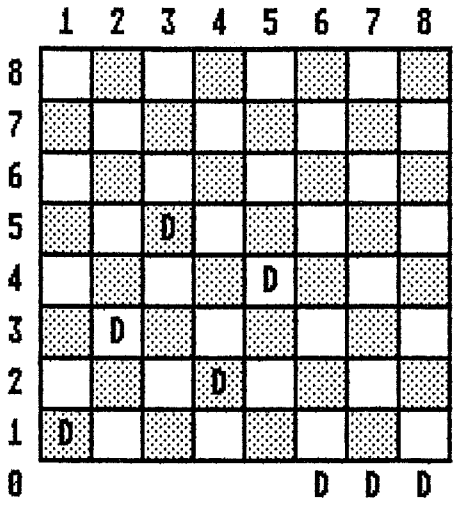
Vor einer Beschreibung der von der gesuchten Prozedur zu veranlassenden Aktionen, überlegen wir uns, wie man "manuell" eine Lösung finden könnte:



Grundposition der acht Damen

- Wir bringen zunächst alle Damen in Grundposition; die Grundposition einer Dame sei vor einer der Spalten. Dies erlaubt uns, von "der Dame" in einer gegebenen Spalte und von ihrer Grundposition zu sprechen.
- Wir nehmen die Dame vor Spalte 1 und setzen sie in dieser Spalte auf das erste Feld. Wir nehmen die Dame vor Spalte 2 und setzen sie in dieser Spalte auf das erste Feld, auf dem sie von der bereits gesetzten Dame nicht bedroht wird;
- Mit den Damen vor den weiteren Spalten verfahren wir ebenso und erhalten:

Eine friedliche Stellung der ersten fünf Damen:



- Die Dame vor Spalte 6 kann dort nun freilich nicht mehr positioniert werden, ohne daß der Friede gestört wird. Eine Abhilfe kann dadurch geschaffen werden, daß man für "Dame 5" die nächste friedliche Position in ihrer Spalte findet und dann erneut "Dame 6" zu positionieren versucht.
- Findet man in einer solchen Situation aber kein "nächstes friedliches Feld" für die Dame einer Spalte, so muß man sie in ihre Grundposition *zurückziehen* und ein "nächstes friedliches Feld" für die Dame in der Vorgänger-Spalte finden. War "Dame 1" die nicht mehr unterzubringende Dame, so gibt es keine Vorgänger-Spalte und man ist fertig.
- Wenn es auf diese Weise gelingt, "Dame 8" friedlich zu setzen, so hat man eine Lösung gefunden und man kann sie notieren. Man wird dann weiter versuchen, auch für "Dame 8" ein "nächstes friedliches Feld" zu finden. Gelingt dies nicht, so zieht man sie auf ihre Grundposition zurück und fährt entsprechend mit "Dame 7" fort. Usw.

Mit diesen vorbereitenden Experimenten können wir eine erste "strukturierte Aktionsbeschreibung" anfertigen:

```
PROCEDURE AchtDamen;
(*Version 1*)
(*Die Prozedur findet alle friedlichen Stellungen von acht Damen
auf einem Schachbrett und gibt sie in Form von 8-Tupeln aus.
Komponente i eines solchen 8-Tupels bezeichnet die Position der
Dame in der i-ten Spalte*)
BEGIN
 (*bringe alle Damen in Grundposition*)
 (*es kann gesetzt werden*)
 (*nimm als zu setzende Dame die "Dame 1"*)
 WHILE (*es kann gesetzt werden*) DO
  (*Suche für die zu setzende Dame die nächste
  friedliche Position in ihrer Spalte*)
  CASE (*Position gefunden*) OF
   TRUE: IF (*zu setzende Dame war "Dame 8"*) THEN
            (*Lösung gefunden, ausgeben*)
```

```
                ELSE
                  (*nimm als zu setzende Dame
                  die Dame vor der nächsten Spalte*)
                END (*IF) |
        FALSE: IF (*zu setzende Dame war "Dame 1"*) THEN
                  (*es kann nicht mehr gesetzt werden*)
                ELSE
                  (*ziehe die zu setzende Dame auf
                  ihre Grundposition zurück*)
                  (*nimm als zu setzende Dame
                  die Dame in der Vorgänger-Spalte*)
                END (*IF*)
          END (*CASE*)
        END (*WHILE*)
      END AchtDamen;
```

Man überzeuge sich davon, daß diese Beschreibung unser "manuelles Verfahren" perfekt widergibt. Man beachte ferner:

- Der Kommentar (*es kann gesetzt werden*) ist offenbar notwendig, um eine Abbruchsbedingung für die WHILE-Anweisung zu erhalten. Es kann nämlich dann nicht mehr gesetzt werden, wenn für "Dame 1" kein "nächstes friedliches Feld" mehr gefunden wird (s.o.).
- Schrittweise Verfeinerung kam bei der Formulierung dieser Prozedur noch gar nicht in's Spiel. Wir haben lediglich unser intuitives Problemverständnis "strukturiert". Diese "Strukturierung" wird nun als Basis für die weitergehende Verfeinerung dienen. Damit kommt bei diesem Beispiel sehr viel stärker zum Ausdruck, was im vorigen Abschnitt (mit der beinahe trivialen Zerlegung in "Gültigkeitsprüfung" und "Folgedatumsberechnung") nur anklang: wieviel Geschick nämlich, Kreativität und eben "Intuition" notwendig sind, um überhaupt erst einen geeigneten Ausgangspunkt für die "Schrittweise Verfeinerung" zu erreichen.

Eine Entscheidung darüber, wie Lösungen zu repräsentieren sind, ist nun notwendig. Sie wird durch den Wortlaut der Aufgabenstellung nahegelegt. Aus ihr geht hervor, daß pro Spalte die Zeilen-Nummer eines Feldes anzugeben ist. Wir definieren also:

```
CONST DimC = 8; (*Seitenlänge - in Anzahl Felder - des
                  Schachbretts*)

TYPE  SpaltenTyp   = (1..DimC);   (*Wertebereich der SpaltenNr.*)
      ZeilenTyp    = (0..DimC);   (*Wertebereich der ZeilenNr.*)
      PositionsTyp = ARRAY SpaltenTyp OF ZeilenTyp;
```

Eine der Regeln in Abschnitt 2.2.1 befolgend, haben wir der (als Anzahl der Felder gemessenen) Seitenlänge des Schachbretts mittels Konstanten-Deklaration einen Namen gegeben. So kann die Prozedur sehr leicht für die Lösung der entsprechenden "Damen-Probleme" auf größeren oder kleineren "Pseudo-Schachbrettern" modifiziert werden.

In den Wertebereich der Zeilen-Nummern haben wir die "0" aufgenommen. Dies ist eine einfache Möglichkeit, um die Grundposition einer Dame zu repräsentieren: Daß die Dame einer gegebenen Spalte in Zeile 0 steht, bedeutet, daß sie ihre Grundposition hat.

Die obige Vereinbarung der Konstanten und Typen kann selbstverständlich in die Prozedur "AchtDamen" eingebettet, auf sie lokalisiert werden. Notwendig ist dies nicht, und wir werden im folgenden davon ausgehen, daß diese Vereinbarung außerhalb der Prozedur getroffen wurde.

Wir benötigen ferner eine Variable vom PositionsTyp, um darin die jeweiligen Stellungen der acht Damen zu speichern. Sei "pos" der Name dieser Variablen. Dann gilt also für die oben entwickelte friedliche Stellung der ersten fünf Damen:

pos[1] = 1, pos[2] = 3, pos[3] = 5, pos[4] = 2, pos[5] = 4,
pos[6] = pos[7] = pos[8] = 0.

Die Versetzung aller Damen in ihre Grundposition wird durch eine FOR-Anweisung erledigt:

```
FOR dame:=1 TO DimC DO pos[dame]:=0 END;
```

Um auszudrücken, daß "gesetzt werden kann" bzw. daß weiteres Setzen nicht mehr möglich ist, führen wir die Variable "setzenIstMoeglich" vom Typ BOOLEAN ein.

Die Suche nach der "nächsten friedlichen Position einer Dame in ihrer Spalte" werden wir einer geeigneten Prozedur überlassen. Den Prinzipien "Schrittweiser Verfeinerung" getreu, schieben wir damit eine Detail-Ausarbeitung auf:

```
PROCEDURE SucheStellung((*IN*) s:SpaltenTyp;
                        (*INOUT*) VAR p:PositionsTyp;
                        (*OUT*) VAR gefunden:BOOLEAN);
(*Sucht ab der in "p" gespeicherten Position der Dame von Spalte
"s" die nächste friedliche Stellung dieser Dame (bezogen auf die
in den Spalten 1..s-1 bereits gesetzten Damen). Falls eine solche
Stellung existiert, wird "p" entsprechend aktualisiert und
"gefunden" erhält den Wert TRUE; im anderen Fall erhält
"gefunden" den Wert FALSE.*)
BEGIN
END SucheStellung;
```

Auch die Druckausgabe oder das "Display" einer Lösung sei einer darauf spezialisierten Prozedur überlassen:

```
PROCEDURE Print(p:PositionsTyp);
(*Gibt eine Stellung der acht Damen in Form eines DimC-Tupels
auf den Bildschirm aus*)
BEGIN
END Print;
```

Version 2 der Prozedur "AchtDamen" macht nun schon einen recht fortgeschrittenen Eindruck:

```
PROCEDURE AchtDamen;
(*Version 2*)
(*s.o.*)
VAR pos: PositionsTyp;
    dame: SpaltenTyp; (*"die Dame von Spalte ..."*)
    setzenIstMoeglich, stellungGefunden: BOOLEAN;
PROCEDURE Print(p:PositionsTyp);
(*s.o*)
BEGIN
END Print;
PROCEDURE SucheStellung((*IN*) s:SpaltenTyp;
                        (*INOUT*) VAR p:PositionsTyp;
                        (*OUT*) VAR gefunden:BOOLEAN);
(*s.o*)
BEGIN
END SucheStellung;
BEGIN (*AchtDamen*)
 (*bringe alle Damen in Grundposition*)
 FOR dame:=1 TO DimC DO
  pos[dame]:=0
 END (*FOR*);
 (*es kann gesetzt werden*)
 setzenIstMoeglich:=TRUE;
 (*nimm als zu setzende Dame die "Dame 1"*)
 dame:=1;
 WHILE setzenIstMoeglich DO
  (*Suche für die zu setzende Dame die nächste
  friedliche Position in ihrer Spalte*)
  SucheStellung(dame,pos,stellungGefunden);
  CASE stellungGefunden OF
   TRUE:  IF dame=DimC THEN
            (*Lösung gefunden, ausgeben*)
```

```
                Print(pos)
              ELSE
                (*nimm als zu setzende Dame
                die Dame vor der nächsten Spalte*)
                dame:=dame + 1
              END (*IF) |
        FALSE: IF dame=1 THEN
                (*es kann nicht mehr gesetzt werden*)
                setzenIstMoeglich:=FALSE
              ELSE
                (*ziehe die zu setzende Dame auf
                ihre Grundposition zurück*)
                pos[dame]:=0;
                (*nimm als zu setzende Dame
                die Dame in der Vorgänger-Spalte*)
                dame:=dame - 1
              END (*IF*)
      END (*CASE*)
    END (*WHILE*)
  END AchtDamen;
```

Die Prozedur "SucheStellung" muß die Dame in Spalte "s" von ihrer aktuellen Position in Zeile p[s] ab solange in dieser Spalte jeweils ein Feld weiterschieben, bis ein "friedliches Feld" gefunden ist:

```
PROCEDURE SucheStellung((*IN*) s:SpaltenTyp;
                        (*INOUT*) VAR p:PositionsTyp;
                        (*OUT*) VAR gefunden:BOOLEAN);
(*Version 1*)
(*s.o*)
BEGIN
  (*nächste friedliche Stellung noch nicht gefunden*)
  IF (*Dame nicht schon auf letztem Feld der Spalte*) THEN
    REPEAT
      (*schiebe Dame auf das nächste Feld*)
      (*prüfe das Feld*)
    UNTIL (*friedliche Stellung gefunden*) OR
        (*letztes Feld der Spalte erreicht*)
  END (*IF*)
END SucheStellung;
```

Zur Prüfung eines Feldes, ob es eine friedliche Stellung erlaubt oder nicht, postulieren wir die Existenz einer Funktion

StellungFriedlich: PositionsTyp × SpaltenTyp -> BOOLEAN

StellungFriedlich(spalte) = ⌈ TRUE, falls die in "spalte" gesetzte Dame die gemäß "p" in die Vorgänger-spalten gesetzten Damen nicht bedroht
⌊ FALSE, sonst

Damit erhalten wir Version 2 von "SucheStellung":

```
PROCEDURE SucheStellung((*IN*) s:SpaltenTyp;
                        (*INOUT*) VAR p:PositionsTyp;
                        (*OUT*) VAR gefunden:BOOLEAN);
(*Version 2*)  (*s.o*)
PROCEDURE StellungFriedlich(p:PositionsTyp;
                            s:SpaltenTyp): BOOLEAN;
(*Prüft nach, ob die in Spalte "s" gesetzte Dame vorher - in die
durch "p" gegebenen Positionen - gesetzten Damen bedroht. *)
BEGIN
END StellungFriedlich;
BEGIN (*SucheStellung*)
 (*nächste friedliche Stellung noch nicht gefunden*)
 gefunden:=FALSE;
 IF p[s] < DimC THEN
  REPEAT
   (*schiebe Dame auf das nächste Feld*)
   p[s]:=p[s] + 1;
   (*prüfe das Feld*)
   gefunden:=StellungFriedlich(s)
  UNTIL gefunden OR (p[s]=DimC)
 END (*IF*)
END SucheStellung;
```

Im Falle der Berechnung des Folgedatums haben wir die "Verpackung" der für diese Berechnung gewissermaßen "zentralen" Schaltjahr-Regel in einer Funktions-Prozedur bis zuletzt aufgeschoben. Auch im vorliegenden Beispiel geht es nun "nur" noch darum, die alles entscheidende Schlag-Regel in die Funktions-Prozedur "StellungFriedlich" zu integrieren. Dazu müssen wir erkennen, wie sich diese Regel unter Verwendung der gewählten Repräsentation von Damen-Stellungen ausdrücken läßt.

Betrachten wir die "Dame s" und eine (bereits gesetzte) "Dame i" mit i < s. "Dame s" bedroht "Dame i", wenn

- "Dame s" und "Dame i" sich auf gleicher Zeile befinden,
  d.h. wenn p(s) - p(i) = 0;
- "Dame s" die "Dame i" diagonal im Visier hat;
  man mache sich klar, daß dies bedeutet:
  p[s] - p[i] = s-i oder p[s] - p[i] = i-s.

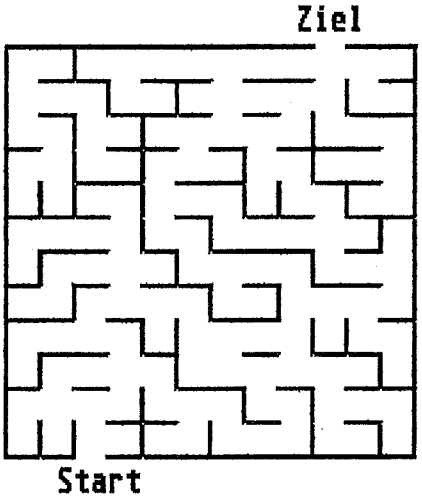
(Der "Spaltenteil" der Schlag-Regel entfällt natürlich, da unsere Prozedur von vornherein ausschließt, daß zwei Damen in die gleiche Spalte geraten.)

Die Funktions-Prozedur "StellungFriedlich" lautet also:

```
PROCEDURE StellungFriedlich(p:PositionsTyp;
                            s:SpaltenTyp): BOOLEAN;
(*s.o.*)
VAR i: SpaltenTyp;
    diff: INTEGER;
    friedlich: BOOLEAN;
BEGIN
 friedlich:=TRUE;
 i:=1;
 WHILE (i < s) AND friedlich DO
  diff:=INTEGER(p[s]) - INTEGER(p[i]) ;
  friedlich:=NOT((diff=0) OR
            (diff=INTEGER(s)-INTEGER(i)) OR
            (diff=INTEGER(i)-INTEGER(s)));
  INC(i)
 END (*WHILE*);
 RETURN friedlich
END StellungFriedlich.
```

Da "s", "i", "p[s]" und "p[i]" alle vom Spalten-Typ, also von einem Subtyp des Typs CARDINAL sind, benötigen wir hier die Typ-Transfer-Funktion "INTEGER", um Zuweisungen zu bzw. Vergleiche mit "diff" zu ermöglichen.

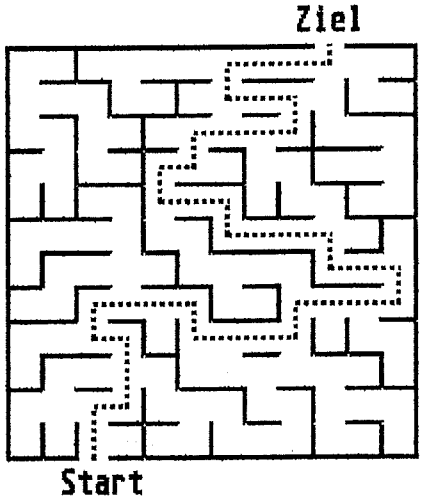
Auf einen Anschrieb der kompletten Prozedur "AchtDamen" verzichten wir an dieser Stelle. Stattdessen betrachten wir ein auf den ersten Blick ganz anderes Problem: die Suche eines Weges durch ein Labyrinth, das etwa wie nebenstehend gezeigt aussehen mag:



Dieses Labyrinth ist eine Anordnung von "Zellen". Die Zellen haben Wände, und in jeweils einigen dieser Wände sind Öffnungen eingelassen, die in benachbarte Zellen führen.

Was würde ein Wesen, das vom Eingang "Start" zum Ausgang "Ziel" gelangen möchte, tun? Eine Möglichkeit sei kurz beschrieben: Es muß sich natürlich zunächst in die Zelle begeben, in die der Eingang "Start" führt. Dort würde es in einer bestimmten Reihenfolge - etwa im Uhrzeigersein - alle Wände anschauen, um eine vom Eingang verschiedene Öffnung zu entdecken. Falls diese schon der Ausgang ist, so wäre es fertig. Andernfalls würde es durch die erste gefundene Öffnung hindurchgehen und die so erreichte Nachbarzelle in der gleichen Weise untersuchen. Irgendwann gerät es dann vielleicht in eine Zelle, die außer ihrem Eingang keine weitere Öffnung besitzt. Dann muß es sich aus dieser Zelle *zurückziehen*, um in der Vorgängerzelle (entsprechend der vorgenommenen Reihenfolge) nach einer weiteren Öffnung zu suchen. Führt auch diese Suche nicht zum Erfolg, so wird es sich noch weiter *zurückziehen* und so fort. Unter bestimmten, an die Struktur des Labyrinths zu stellenden Bedingungen, wird es so schließlich das Ziel erreichen, falls überhaupt ein Weg dorthin führt. Gibt es aber keinen Weg zum "Ziel", so wird es sich irgendwann wieder in der "Startzelle" befinden und alle deren Öffnungen "abgearbeitet" haben.

Im Fall des oben skizzierten Labyrinths wird man etwa folgenden Weg finden:



Versuchen wir, der Beschreibung ein wenig mehr Form zu geben:

```
PROCEDURE Labyrinth;
(*Die Prozedur findet alle Wege vom Eingang "Start" des
Labyrinths bis zum Ausgang "Ziel", falls ein solcher Weg
überhaupt existiert.*)
BEGIN
  (*man lege für alle Zellen eine Reihenfolge
  der Betrachtung ihrer Wände fest*)
  (*es kann weitergegangen werden*)
```

```
(*man nehme die Startzelle
als zu untersuchende Zelle*)
WHILE (*es kann weitergegangen werden*) DO
 (*man finde in der zu untersuchenden Zelle
 die nächste Wand, die eine Öffnung enthält*)
 CASE (*Öffnung gefunden*) OF
  TRUE:  IF (*Öffnung ist Ausgang*) THEN
           (*Weg gefunden*)
         ELSE
           (*man gehe durch diese Öffnung*)
           (*die nächste zu untersuchende Zelle
           ist die so erreichte Nachbarzelle*)
         END (*IF) |
  FALSE: IF (*Öffnung ist Eingang*) THEN
           (*es kann nicht weitergegangen werden*)
         ELSE
           (*man stelle in der aktuellen Zelle die
           Reihenfolge der Betrachtung ihrer Wände
           wieder her*)
           (*man ziehe sich in die Vorgängerzelle
           zurück und nehme sie als zu untersuchende
           Zelle*)
         END (*IF*)
 END (*CASE*)
END (*WHILE*)
END Labyrinth;
```

Gewiß, der Schwierigkeit, die wir oben etwas vornehm mit "bestimmten, an das Labyrinth zu stellenden Bedingungen" umschrieben haben, wird durch diese halbformale Darstellung der Prozedur des Wegesuchens in einem Labyrinth zunächst auch nicht Rechnung getragen: Es handelt sich darum, daß man ja "im Kreise laufen kann" und dies entdecken muß. Wir wollen uns mit dieser Schwierigkeit aber auch gar nicht weiter aufhalten, da wir inzwischen eine viel interessantere Entdeckung gemacht haben:

Es gibt eine verblüffende Ähnlichkeit zwischen dem Schema ("Version 1") der Prozedur zur Ermittlung der friedlichen Stellungen von acht Damen auf einem Schachbrett und dem eben entwickelten Schema einer Prozedur, welche die Wege durch ein Labyrinth sucht. Offenbar mußten wir nur einige Objekte austauschen sowie Bedingungen und elementare Schritte etwas uminterpretieren, um von "AchtDamen" zu "Labyrinth" zu gelangen:

AchtDamen	Labyrinth
Bringe alle Damen in Grundposition	Man lege für alle Zellen eine Reihenfolge der Betrachtung ihrer Wände fest
Es kann gesetzt werden	Es kann weitergegangen werden
Spalte	Zelle
Nächste Spalte	Nachbarzelle
Vorgänger-Spalte	Vorgängerzelle
Position in Spalte ...	Wand
friedliche Position	Wand mit Öffnung
usw.	

(Man sieht: Nicht nur durch die Komplexität des Schachspiels begründet kann das Schachbrett als ein Labyrinth aufgefaßt werden!)

Der Leser ist aufgefordert, sich zu überlegen, wie - gemäß dieser Analogie - eine Lösung des Labyrinth-Problems notiert werden müßte.

Das zielsuchende Wesen hätte nun sicherlich auch ganz andere Erwägungen anstellen können, um den Ausweg zu finden. Es hätte sich zum Beispiel sagen können: "Gleichgültig, in welcher Zelle ich mich jeweils befinde, ich werde mich immer wie folgt verhalten: Ich untersuche in der festgesetzten Reihenfolge alle Wände dieser Zelle (außer der Wand, durch die ich gekommen bin); ist eine Öffnung in der Wand, so führt diese entweder direkt zum Ziel oder nicht. Im ersten Fall bin ich fertig. Im zweiten Fall gehe ich durch die Öffnung in die Nachbarzelle, wo ich mich natürlich genauso verhalten muß! Nachdem ich so alle Wände der Zelle untersucht habe, weiß ich, daß es aus dieser Zelle keinen Ausweg gibt."

Da dies Verhalten auch für die Startzelle gilt, führt seine Anwendung dort entweder zum Ziel oder zur Einsicht, daß kein Weg zum Ziel existiert.

Bringen wir auch diese Vorgehensweise "in eine Form":

```
PROCEDURE Labyrinth;
(*Die Prozedur findet einen Weg vom Eingang "Start" des
Labyrinths bis zum Ausgang "Ziel", falls ein solcher Weg
überhaupt existiert.*)
PROCEDURE FindeAusweg(zelle:ZellenTyp);
(*Die Prozedur findet einen Weg von "zelle" zum "Ziel", falls ein
solcher Weg überhaupt existiert*)
BEGIN
  FOR (*alle Wände außer der Eingangswand*) DO
    IF (*Öffnung in der Wand*) THEN
```

```
        (*man gehe durch diese Öffnung
        in die Nachbarzelle*)
        CASE (*Nachbarzelle ist Ziel*) OF
          TRUE:  (*fertig; Erfolgsmeldung;
                    Abbruch der Prozedur*)|
          FALSE: FindeAusweg(nachbarzelle)
        END (*CASE*);
        (*man gehe aus der Nachbarzelle in die
        Zelle "zelle" zurück*)
      END (*IF*)
    END (*FOR*)
  END FindeAusweg;

  BEGIN (*Labyrinth*)
    (*man lege eine Reihenfolge der
    Betrachtung der Zellwände fest*)
    FindeAusweg(startzelle)
  END Labyrinth;
```

Wir wollen nun den umgekehrten Schritt tun und diese "Prozedur" - unter Ausnutzung der Entsprechungen zwischen dem Problem der acht Damen und dem des Labyrinths - in ein weiteres Schema für eine Prozedur zur friedlichen Aufstellung der acht Damen "übersetzen":

```
PROCEDURE AchtDamen;
(*Die Prozedur findet eine friedliche Aufstellung von acht Damen
auf dem Schachbrett*)

PROCEDURE FindeStellung(spalte:SpaltenTyp);
(*Die Prozedur findet eine friedliche Aufstellung der Damen ab
Spalte "spalte"*)
BEGIN
  FOR (*alle Positionen der Spalte "spalte"
     (außer der Position 0)*) DO
    IF (*Position friedlich*) THEN
      (*man setze die Dame an diese Position
      und betrachte die nächste Spalte*)
      CASE (*keine nächste Spalte mehr*) OF
        TRUE:  (*fertig; Ausgabe des Ergebnisses;
                  Abbruch der Prozedur*) |
        FALSE: FindeStellung(naechsteSpalte)
      END (*CASE*);
      (*man betrachte wieder die Spalte "spalte"
      und entferne dort die Dame
      von ihrer aktuellen Position*)
```

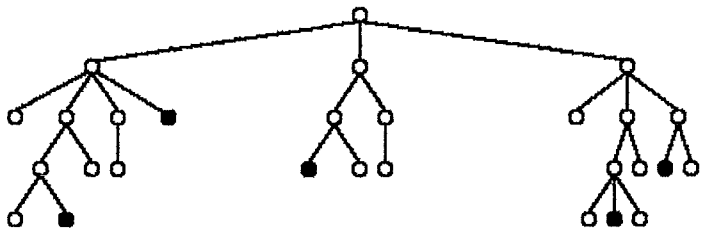
```
    END (*IF*)
  END (*FOR*)
END FindeStellung;
BEGIN (*AchtDamen*)
  (*man bringe die acht Damen in
  ihre Grundposition*)
  FindeStellung((*Spalte*) 1)
END AchtDamen;
```

Es bleibe dem Leser als Übung überlassen, ausgehend von diesem Schema eine Prozedur zur Lösung des Acht-Damen-Problems zu entwickeln. Möglicherweise ist es dabei geschickter, andere Repräsentations-Entscheidungen zu treffen als oben.

Wir bemerken noch, daß es sich bei den zuletzt skizzierten Prozeduren zur Lösung des Labyrinth- bzw. des Acht-Damen-Problems um *rekursive* Prozeduren handelt.

Selbstverständlich hätten wir den rekursiven Ansatz zur Lösung des Acht-Damen-Problems auch leicht direkt, ohne den Umweg über die Labyrinth-Analogie finden können. Andererseits lehrt uns ein solcher Vergleich, daß die betrachteten Schemata offenbar nur Spezialfälle von allgemeineren Lösungsmustern sind. Worin aber besteht das allgemeine Problem, das durch dieses allgemeinere Muster gelöst wird und von dem die diskutierten Probleme Spezialfälle sind?

Versuchen wir zu abstrahieren: In beiden Fällen ging es darum, von einer gegebenen (Anfangs-)Situation aus eine (oder mehrere) (End-)Situation(en) mit ebenfalls bestimmten Eigenschaften zu erreichen bzw. herzustellen. (Im Falle des Labyrinth-Problems waren diese Endsituationen nichts anderes als Beschreibungen eines Wegs vom Start zum Ziel!) Dabei waren unter Umständen mehrere Zwischen-Situationen zu passieren. In jeder Situation (die End-Situation ausgenommen) gab es mögliche (bzw. erlaubte) Folgesituationen (von denen nicht alle brauchbar waren!) oder nicht. Falls es Folgesituationen gab, mußten diese weiterverfolgt werden; falls es keine Folgesituationen gab, mußte der Schritt, der zu der jeweiligen (Anfangs- oder Zwischen-)Situation ("in die Sackgasse") geführt hatte, wieder rückgängig gemacht werden.



● : Endsituation

Dies also ist das "Problem-Muster". Veranschaulichen läßt es sich vielleicht am besten durch einen Baum, dessen Knoten die verschiedenen Situationen repräsentieren.

Der Anfangs-Situation entspricht die Wurzel des Baumes. Falls ein Knoten Folgeknoten hat, so können davon einige (evtl. alle) unbrauchbar sein. Die End-Situationen entsprechen einer Teilmenge (die auch leer sein kann) der Menge der übrigen Knoten. Um sie zu finden, kann man wie folgt vorgehen:

```
PROCEDURE SituationsBaum;
(*iterative Variante*)
(*Die Prozedur findet alle Endsituationen im Situationsbaum*)
BEGIN
 (*lege die Reihenfolge der Betrachtung
 von Folgeknoten fest*)
 (*es kann weitergesucht werden*)
 (*der zu untersuchende Knoten ist die Wurzel*)
 WHILE (*es kann weitergesucht werden*) DO
  (*prüfe ob der zu untersuchende Knoten
  einen weiteren brauchbaren Folgeknoten hat*)
  CASE (*Folgeknoten gefunden*) OF
   TRUE:  IF (*Folgeknoten entspricht End-Situation*) THEN
            (*End-Situation gefunden*)
          ELSE
            (*der zu untersuchende Knoten
             ist der gefundene Folgeknoten*)
          END (*IF) |
   FALSE: IF (*Knoten ist Wurzel*) THEN
            (*es kann nicht weitergesucht werden*)
          ELSE
            (*man ziehe sich auf den Vorgängerknoten
            zurück und nehme ihn als zu untersuchenden
            Knoten*)
          END (*IF*)
  END (*CASE*)
 END (*WHILE*)
END SituationsBaum;
```

Oder so:

```
PROCEDURE SituationsBaum;
(*rekursive Variante*)
(*Die Prozedur findet - falls vorhanden - die erste End-Situation
im Situationsbaum*)
PROCEDURE FindeEndSituation(knoten:KnotenTyp);
(*Die Prozedur findet - falls vorhanden - die erste von "knoten"
aus erreichbare End-Situation im Situationsbaum*)
```

```
BEGIN
  FOR (*alle Folgeknoten*) DO
    IF (*Folgeknoten brauchbar*) THEN
      (*man begebe sich zum Folgeknoten*)
      CASE (*Folgeknoten entspricht End-Situation*) OF
        TRUE:  (*fertig; Erfolgsmeldung; Abbruch der Prozedur*) |
        FALSE: FindeEndSituation(folgeKnoten)
      END (*CASE*);
      (*man begebe sich zum Knoten "knoten" zurück*)
    END (*IF*)
  END (*FOR*)
END FindeEndSituation;

BEGIN (*SituationsBaum*)
  (*man lege eine Reihenfolge der
  Betrachtung von Folgeknoten fest*)
  FindeEndSituation(wurzel)
END SituationsBaum;
```

Beide Prozedur-Schemata beschreiben ein und dieselbe Art, die Knoten eines Situationsbaums auf der Suche nach End-Situationen zu durchlaufen: von oben nach unten "mit erster Priorität" und dann (z.B.) von links nach rechts. Geht es von einem Knoten aus nicht weiter, so zieht man sich auf den Vorgängerknoten zurück und versucht es mit dessen nächstem Folgeknoten, usw.. Diese Art, einen Baum zu durchlaufen, wird in der angelsächsischen Literatur meist als "*depth first*" bezeichnet und der dabei notwendige Rückzug auf Vorgängerknoten als "*Backtracking*". (Auch in der einschlägigen deutschsprachigen Literatur - vgl. z.B. [OTW] - sind diese Begriffe durchaus gebräuchlich.)

Sowohl das Acht-Damen-Problem als auch das Labyrinth-Problem werden also durch Verfahren gelöst, die zur Klasse der *Backtrack-Algorithmen* gehören. Die Interpretation von in der Sprache PROLOG (vgl. Einleitung) geschriebenen Programmen etwa (z.B. "Expertensysteme") erfolgt im Prinzip nach dem gleichen Schema (vgl. [CLM]).

Wir sind in diesem Abschnitt deshalb (scheinbar) relativ weit von unserem eigentlichen Thema "Schrittweise Verfeinerung" abgewichen, um nochmals zu verdeutlichen, daß es naiv wäre, in dieser Vorgehensweise gewissermaßen ein Patentrezept für die Entwicklung von Programmen zu sehen. Vielmehr basiert gerade die Entwicklung anspruchsvollerer Programme im allgemeinen auf Lösungsmustern der vorgestellten Art. Es kommt dabei darauf an, die Anwendbarkeit dieser Lösungsmuster zu erkennen. Dies ist - natürlich neben der Entwicklung allgemeiner Lösungsmuster selbst - der eigentlich kreative Akt.

Neben dem hier dargelegten *Backtracking-Paradigma* gibt es allgemeine Lösungsmuster für viele andere abstrakte Probleme. Auch nur einige weitere sol-

cher Paradigmata zu besprechen, würde - obschon dies mit Recht Teil einer umfassenden Darstellung von Methoden der imperativen Programmierung sein könnte - den Rahmen dieses Buches sprengen. (Zum vertiefenden Studium algorithmischer Paradigmata existiert ein breites Angebot entsprechender Literatur, vgl. z.B. [OTW] oder [HOS]).

3.3 Diagramm-Techniken zur Unterstützung schrittweiser Verfeinerung

Programme, die anstelle ausführbarer Anweisungen oder auswertbarer Ausdrükke nur Kommentare (eingehängt in ein "Skelett" von strukturbildenden Schlüsselwörtern) enthalten, werden üblicherweise als "Pseudo-Code" bezeichnet. Die im letzten Abschnitt dargestellte Entwicklung der Prozeduren "FolgeTag" und "AchtDamen" basierte auf solchen Pseudo-Code-Texten. Sie dienten als "Rohformen" bzw. als "Entwurfsskizzen" und können daher als Bestandteile einer Entwurfs-Dokumentation angesehen werden. Sie ermöglichen einen schnellen Überblick über Zweck und Arbeitsweise eines Programmstücks. Die rasche Erfassung der Programm-Struktur wird zusätzlich erleichtert, wenn auch Pseudo-Code unter Beachtung geeigneter Indentierungsregeln verfaßt wird. Entscheidend ist dabei, daß der Text ein "graphisches Gepräge" erhält, und so gewissermaßen zum Bild wird.

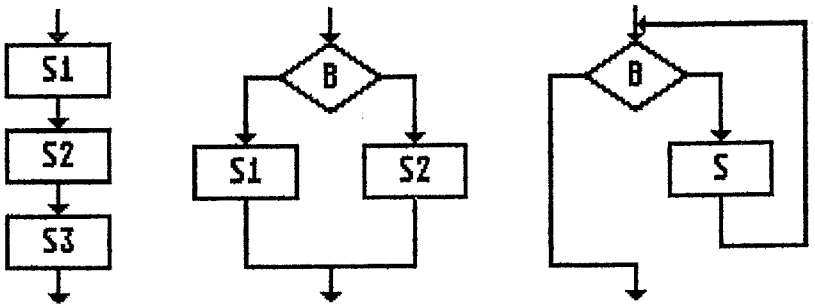
"Ein Bild sagt mehr als tausend Worte" ist eine alte Volksweisheit. Jeder mit handfesten Dingen beschäftigte Ingenieur wird sie sofort bestätigen. Die darin zum Ausdruck gebrachte Einsicht auch auf die Dokumentation des Detail-Entwurfs von Programmen anzuwenden, ist ein naheliegender Wunsch. Er würde erfüllt durch eine "Bild-Sprache", in der noch augenfälligere Darstellungen von Programmstrukturen möglich sind als mit indentiertem Pseudo-Code. Zwei Hauptaspekte sind bei der Definition oder Wahl einer solchen Bild-Sprache zu beachten:

- Zum einen müssen die Grundstrukturen "Sequenz, Selektion und Iteration" deutlich sichtbar gemacht werden können;
- zum anderen sollten die Symbole dieser Bild-Sprache leicht handhabbar sein.

Die leichte Handhabbarkeit der Symbole ist von Bedeutung sowohl für die rasche manuelle Erstellung von Entwurfsskizzen als auch für die Unterstützung des Entwurfsprozesses durch entsprechende Rechner-Programme.

Bis heute sind sogenannte *Fluß-Diagramme* (englisch: *Flow-Charts*) sehr populäre Darstellungen des Detail-Entwurfs von Programmen. Fluß-Diagramme haben jedoch einige gravierende Nachteile, die dadurch begründet sind, daß sie zu einer Zeit "erfunden" wurden, als es noch keine höheren Programmierspra-

chen wie PASCAL oder MODULA-2 gab. So sind die Grundstrukturen zwar ohne weiteres abbildbar, wie die folgenden Diagramme zeigen:



Sequenz:

```
S1;
S2;
S3;
```

Selektion:

```
IF B THEN
  S1
ELSE
  S2
END (*IF*);
```

Iteration:

```
WHILE B DO
  S
END (*WHILE*);
```

Es bedarf jedoch einiger Mühe und Disziplin, um größere Diagramme so zu erstellen, daß die Grundstrukturen im Zusammenhang gut erkennbar sind. (Auch Software-Werkzeuge, die auf dieser Diagramm-Technik beruhen, haben damit Schwierigkeiten!) Außerdem besteht für den Benutzer von Fluß-Diagrammen keinerlei Zwang, im wesentlichen nur die Grundstrukturen zu verwenden. (Es sei denn, er bedient sich beim Entwurf eines Software-Werkzeugs, das ihm keine andere Wahl läßt.) Vielmehr gestatten sie die Anfertigung von Entwürfen, die sich dann unter Umständen nur sehr schwer (und mit viel zusätzlichem Aufwand) in MODULA-2 - Programme umsetzen lassen. Noch schlimmer aber ist, daß es dann keine direkten Entsprechungen zwischen dem Entwurf und dem Programmtext mehr gibt. Damit werden Fluß-Diagramme als Dokumente, die das Verständnis eines Programms erleichtern sollen, praktisch wertlos. (Es soll nicht verschwiegen werden, daß die Konstruktion von Fluß-Diagrammen durchaus sinnvoll sein kann, wenn es um die Erstellung von Programmen in Maschinensprache geht, an die hohe Anforderungen hinsichtlich Kompaktheit und Effizienz (im Sinne von Abschnitt 2.3!) gestellt werden. Solche Forderungen waren früher - bei teurer und knapper Hardware - sehr häufig. Sie spielen heute nur noch relativ selten eine Rolle.)

In den beiden folgenden Abschnitten werden wir je eine (halb-)graphische Notation zur Dokumentation strukturierter Programm-Entwürfe erläutern, die mit den genannten Nachteilen der Fluß-Diagramme nicht (oder nur in erheblich geringerem Maße) behaftet sind. Wir beginnen jeweils mit einer Darstellung der graphischen Grundelemente (mit Angabe der entsprechenden MODULA-2 - Programmstücke) und zeigen dann die Verwendung dieser Elemente bei der schritt-

weisen Entwicklung zweier kleiner Programme: das erste löst ein lineares Gleichungssystem, das zweite hat die Aufgabe, Rechnungen über gelieferte Waren zu schreiben. Auf die für beide Techniken mögliche Unterstützung durch Software-Werkzeuge werden wir hier nicht eingehen.

Die Auswahl gerade dieser beiden Techniken geschah nicht ganz willkürlich. Nassi-Shneiderman-Diagramme wurden gewählt, weil sie eine sehr gute bildhafte Entsprechung zu jenen "Schwarzen Kästen" liefern, mit denen wir das Prinzip der "Schrittweisen Verfeinerung" zu Anfang dieses Kapitels erklärt haben. Im Gegensatz zu den Nassi-Shneiderman-Diagrammen sind Klammerdiagramme auch für den (manuellen) Freihand-Entwurf zu gebrauchen. Wir führen sie aber vornehmlich deshalb bereits hier ein, weil wir uns ihrer - in anderem Zusammenhang und mit erweiterter Interpretation - im Kapitel 5 ausgiebig bedienen werden. Die Verwendung von Klammerdiagrammen, so wie sie in diesem Kapitel gezeigt wird, entspricht im übrigen sehr direkt der Arbeit mit den sogenannten "Aktions-Diagrammen", die z.B. von J. Martin propagiert werden (vgl. [MAM]).

Wir beschreiben noch kurz die beiden Aufgaben, deren Lösungen in den folgenden Abschnitten entwickelt werden:

Lösung eines linearen Gleichungssystems:

Es ist eine Prozedur zu entwerfen, die für ein (wohl-konditioniertes, nicht-singuläres) lineares Gleichungssystem A*X=B (A: N*N-Matrix, B: N-Spaltenvektor, X: N-Spaltenvektor) den Lösungsvektor liefert. Es soll das Gauss'sche Eliminationsverfahren benutzt werden. Von einer Pivotierung ist vorerst abzusehen. Die Prozedur ist abzubrechen, sobald eine Division durch Null erfolgen müßte. (Vgl. z.B. [STH])

Rechnungs-Ausschrieb:

Es ist ein Programm zu entwerfen, das Rechnungen über Warenlieferungen ausgibt (am Terminal oder auf Drucker). Jede Rechnung beginnt mit einer Kopfzeile

"KUNDEN-NUMMER: ——".
Es folgen Zeilen für die gelieferten Posten:
"POSTEN ‹Artikel-Nr›: ‹Preis›".
Anschließend folgt die Bruttosummenzeile und danach die Nettosummenzeile (die den Gesamtpreis nach Abzug eines kundenspezifischen Rabatts enthält). Fehlermeldungen sind geeignet auszugeben.

Zum Zweck des Rechnungs-Ausschriebs existieren drei Eingabe-Dateien:

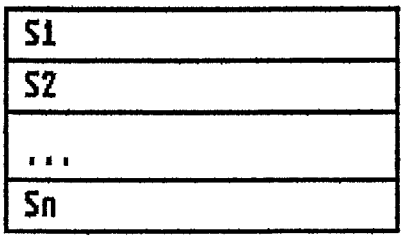
1. "LIEFERUNG". Deren Sätze enthalten:
   Kunden–Nr, Artikel–Nr, Stückzahl.
   Sie ist nach Kundennummern sortiert und wird sequentiell gelesen.
2. "KUNDENLISTE". Die Sätze enthalten:
   Kunden–Nr, Rabattsatz, Anschrift.
   Sie wird per Schlüssel "Kunden–Nr" im Direktzugriff gelesen.
3. "PREISLISTE": Die Sätze enthalten:
   Artikel–Nr, Stückpreis.
   Sie wird per Schlüssel "Artikel–Nr" im Direktzugriff gelesen.

3.3.1 Nassi-Shneiderman-Diagramme

Diese Diagramm-Technik geht auf einen Vorschlag von I. Nassi und B. Shneiderman aus dem Jahre 1973 zurück, Fluß-Diagramme durch eine für die (damals so genannte) "Strukturierte Programmierung" geeignetere "Bild-Sprache" zu ersetzen (vgl. [NAS]). Unter der Bezeichnung "Struktogramme" haben Nassi-Shneiderman-Diagramme eine gewisse Popularität erlangt..

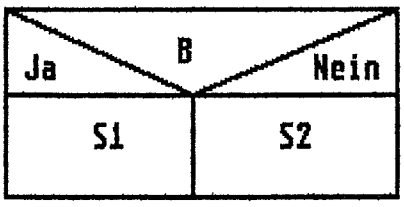
Grundelemente:

Hier (wie in 3.3.2) bezeichnen wir mit S,S1,S2,... Folgen von Anweisungen ("Programmstücke") und mit B,B1,B2,... boolesche Ausdrücke. Die Diagrammformen sind nicht normiert. Es ist daher durchaus möglich, daß man bei anderen Autoren oder in der Praxis Abweichungen von den hier gezeigten Formen findet.



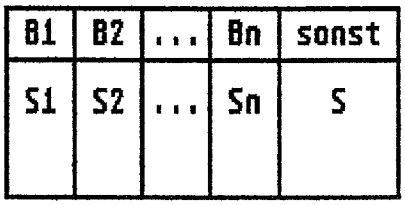
Sequenz:

```
S1;
S2;
. . .
. . .
Sn;
```



Selektion 1:

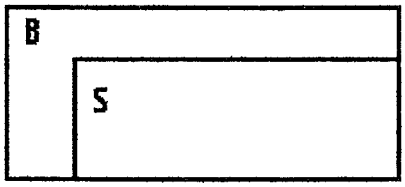
```
IF  B  THEN
  S1
ELSE
  S2
END (*IF*);
```



Selektion 2:

```
IF B1 THEN
  S1
ELSIF B2 THEN
  S2
...
ELSIF Bn THEN
  Sn
ELSE
  S
END (*IF*);
```

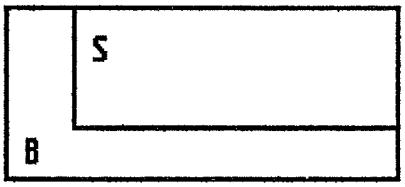
(Falls die booleschen Ausdrücke Bi von der Form "<varbez>=wert_i" sind, so kann dieses Diagramm in naheliegender Weise auch in eine CASE-Anweisung umgesetzt werden!)



Iteration 1:

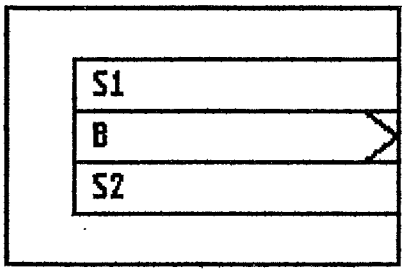
```
WHILE B DO
  S
END (*WHILE*);
```

(Falls der boolesche Ausdruck B von der Form "<varbez> IN menge" ist, so kann dieses Diagramm auch in naheliegender Weise in eine FOR-Anweisung umgesetzt werden.)



Iteration 2:

```
REPEAT
  S
UNTIL B;
```



Iteration 3:

```
LOOP
  S1;
  IF B THEN EXIT END (*IF*)
  S2
END (*LOOP*);
```

1. Lösung eines linearen Gleichungssystems

1.1 Einlesen der Matrix a
1.2 Einlesen des Vektors b
1.3 Berechne Dreiecksform Durch sukzessives Eliminieren der x[1]..,x[N] wird das gegebene Gleichungssystem auf obere Dreiecksform gebracht. Tritt dabei eine Division durch Null auf, so wird die Prozedur abgebrochen.
1.4 Berechne Lösungsvektor Die Komponenten des Lösungsvektors werden in der Reihenfolge x[N]...x[1] aus dem in 1.3 ermittelten Gleichungssystem durch sukzessives Einsetzen berechnet.
1.5 Ausgabe der Lösung

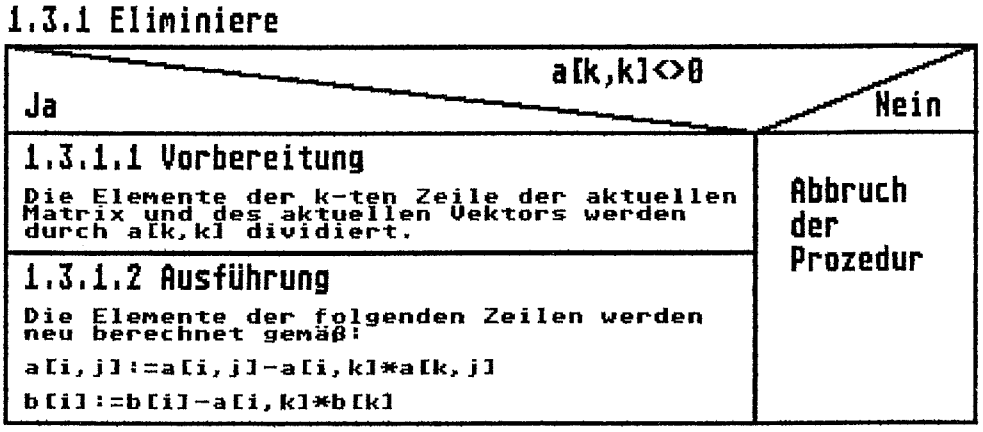
Man beachte, daß wir hier die Programm-Abschnitte sowohl numerieren als auch benennen. Die Punkte 1.1, 1.2 und 1.5 erwähnen wir nur der Vollständigkeit halber, da der Aufgabentext keinen Hinweis auf die Art und Weise der Versorgung der Prozedur mit Daten enthält. (Ebensogut könnten diese als Parameter übergeben werden.)

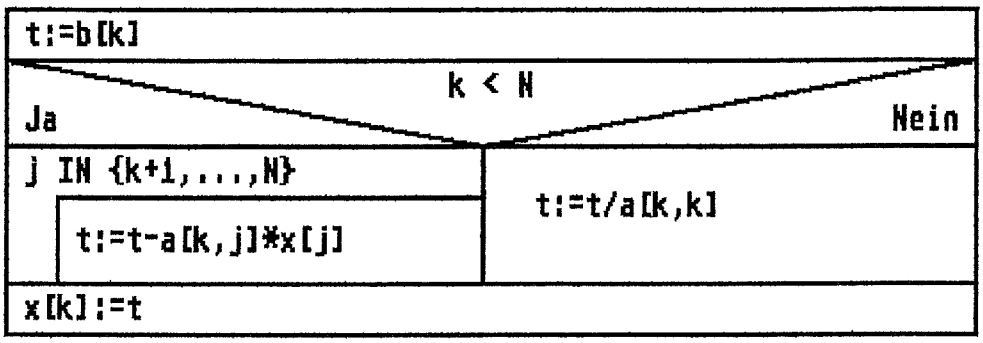
1.3 Berechne Dreiecksform

k IN {1,...,N-1}
1.3.1 Eliminiere Elimination von x[k]: Die Elemente der k-ten Zeile des aktuellen Gleichungssystems werden - falls möglich - durch a[k,k] dividiert. Falls a[k,k]=0, so wird die Prozedur abgebrochen. Die Zeilen k+1,...,N von a und b werden neu berechnet.

1.4 Berechne Lösungsvektor

k:=N
1.4.1 Berechne Komponente Die k-te Komponente des Lösungsvektors wird nach Einsetzen der schon berechneten Komponenten aus der k-ten Zeile berechnet.
k:=k-1
k=0





Mit "Berechne Komponente" sind wir so weit in die Details hinabgestiegen, daß aus dem Diagramm ein Programmstück unmittelbar erzeugt werden könnte. Andere Teile des Entwurfs sind davon noch einige Schritte entfernt. Der Leser ist aufgefordert, diese Schritte selbständig auszuführen und durch Nassi-Shneiderman-Diagramme zu dokumentieren.

Häufig ist es freilich weder notwendig noch sinnvoll, die Dokumentation eines Detail-Entwurfs - wie hier durchexerziert - bis auf die Ebene des Programmcodes voranzutreiben. Dafür gibt es mehrere Gründe. Erstens kann es wünschenswert sein, auch den Detail-Entwurf zunächst unabhängig von der Wahl einer Programmiersprache auszuführen. Dies würde die "Portierbarkeit" einer Problemlösung (also die Herstellung eines entsprechenden Programms in einer anderen Sprache und evtl. auf einem anderen Rechner) wesentlich erleichtern. (Auf *Portabilität* als Qualitäts-Merkmal von Software werden wir in Kapitel 6 etwas ausführlicher eingehen.) Zweitens besteht die Notwendigkeit, Änderungen, die sich im Programmtext - aus welchen Gründen auch immer - ergeben, auch in den Entwurfsdokumenten nachzuvollziehen. Es ist klar, daß dieser Aktualisierungsaufwand um so geringer ist, je abstrakter die Beschreibungen von Programmteilen im Entwurfsdokument gehalten sind. Dabei darf Abstraktion natürlich nicht mit mangelnder Präzision verwechselt werden. Es ist vielmehr ent-

scheidend, daß ein Detail-Entwurf als Vorstufe zum Programm-Text genau die Informationen über Inhalt und Struktur einer Problemlösung enthält, die für ihr Verständnis und die Umsetzung in - im Prinzip - beliebige (imperative) Programmiersprachen erforderlich sind.

2. Rechnungs-Ausschrieb

2.1 Initialisierung

Die Dateien "LIEFERUNG", "KUNDENLISTE" und "PREISLISTE" werden eröffnet.

Der erste Satz von "LIEFERUNG" wird gelesen; er enthält: Kunden-Nummer (lKuNr), Artikel-Nummer (lArtNr) und Stückzahl (lStZ).

Ist die Datei "LIEFERUNG" leer, so wird "eof" gemeldet und das Programm wird abgebrochen.

2.2 Verarbeitung

Solange "eof" nicht gemeldet ist, wird der jeweils aktuelle Satz folgendermaßen verarbeitet:

Stimmt lKuNr des aktuellen Satzes mit der Kunden-Nummer des vorigen Satzes nicht überein, so wird
- für den vorigen Kunden (sofern es einen gab) der Rabatt ausgerechnet, und die Summenzeilen werden geschrieben;
- für den aktuellen Kunden wird per Schlüssel lKuNr aus der "KUNDENLISTE" sein Rabattsatz ermittelt (wenn der aktuelle Kunde nicht in der "KUNDEN-LISTE" steht, so wird dies vermerkt); die Brutto-summe wird mit 0 initialisiert, und der aktuelle Kunde wird zum "vorigen Kunden" erklärt;
- eine Kopfzeile wird geschrieben.

Für die aktuelle Artikel-Nummer lArtNr wird in jedem Fall mit dieser Nummer als Schlüssel aus der "PREISLISTE" der Stückpreis ermittelt, der Gesamt-preis wird ausgerechnet und der Bruttosumme zuge-schlagen. Für den Artikel wird eine "Posten-Zeile" geschrieben. Fehlt der Artikel in der "PREISLISTE", so wird dies vermerkt.

Ein neuer Satz von "LIEFERUNG" wird gelesen.

2.3 Abschluß

Berechnung des Rabatts für den letzten Kunden und Ausschrieb der Summenzeilen.
Schließen sämtlicher Dateien.

"eof" bedeutet "*end of file*". Wir gehen davon aus, daß ein entsprechendes Signal erzeugt wird, nachdem das Ende der (sequentiellen!) Datei "LIEFERUNG" erreicht ist. Wir berücksichtigen hier nicht die Tatsache, daß die in der Aufgabenstellung genannten Dateien nicht geöffnet werden können (z.B. weil sie gar nicht vorhanden sind).

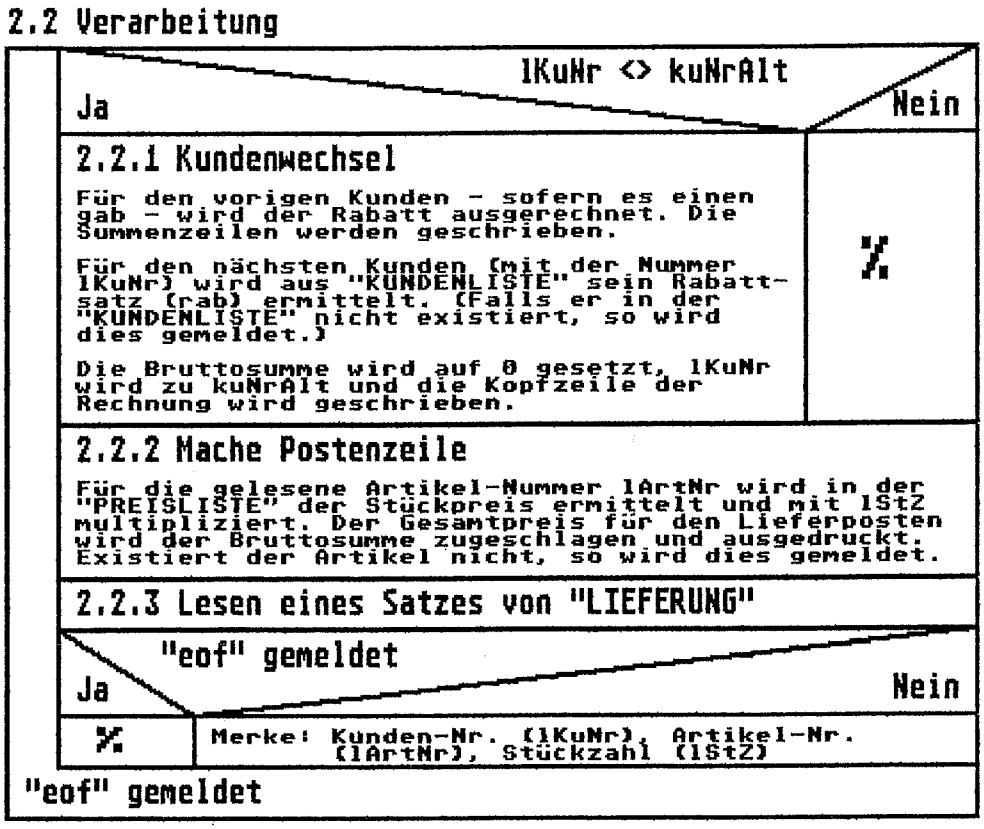
2.1 Initialisierung

2.1.1 Eröffnung der Dateien "LIEFERUNG", "KUNDENLISTE" und "PREISLISTE"
2.1.2 Lesen eines Satzes von "LIEFERUNG"

"eof" gemeldet: Ja	Nein
Abbruch des Programms	Merke: Kunden-Nr. (lKuNr), Artikel-Nr. (lArtNr), Stückzahl (lStZ)

2.1.3 Zuweisung eines fiktiven Wertes ("dummy") an "vorige Kunden-Nr." (kuNrAlt)

Wir begnügen uns mit der Verfeinerung von 2.1 und 2.2 und überlassen die weiteren Entwurfsschritte dem Leser zur Übung.



3.3.2 Klammerdiagramme

Klammern anstatt "Kästchen" zur Zusammenfassung zusammengehöriger Teile wurden erstmals von J. Warnier für die Programm-Dokumentation vorgeschlagen (vgl. [WAR]). Diese Technik wurde von K. Orr übernommen ([ORR]). Daher spricht man auch von "Warnier-Orr-Diagrammen". Wir benutzen diese Bezeichnung jedoch nicht, da wir eine Variante der Klammerdiagramm-Notation präsentieren, die sich von der ursprünglichen Fassung insbesondere durch eine präzisere Definition ihrer textlichen Elemente unterscheidet. Diese Variante stammt von H.J. Ehling (vgl. [EHL]).

Im Unterschied zu Nassi-Shneiderman-Diagrammen, bei denen einzelne "Kästen" Namen erhalten können, müssen Klammern immer benannt sein. In Ausnahmefällen kann man sich eines fiktiven ("dummy"-)Namens ("___") bedienen. Die S, S1,... bezeichnen hier also entweder elementare Anweisungen oder sie sind als Namen zu interpretieren, welche weiter strukturierte Anweisungsfolgen identifizieren.

```
SEQ  S1
     *
     S2
     *
     ...
     *
     Sn
```

Sequenz:
S1;
S2;
. . .
. . .
Sn;

Das Zeichen "*" bedeutet hier "danach folgt". Es ist der "Sequenz-Konnektor".

```
SEL_1  ?B;
         S1
       +
       ?sonst;
         S2
```

Selektion 1:
IF B THEN
  S1
ELSE
  S2
END (*IF*);

Das Zeichen "+" bedeutet hier "oder". Es ist der "Selektions-Konnektor".

S1 oder S2 kann auch "leer" sein. Wir benutzen in diesem Fall das reservierte Wort **"NONE"** und meinen damit, daß an dieser Stelle gar nichts getan wird.

Für die folgende Selektion aus mehr als zwei möglichen Programmteilen gilt die in 3.3.1 zu "Selektion 2" angebrachte Bemerkung entsprechend.

```
SEL_2  ?B1;
         S1
       +
       ?B2;
         S2
       +
       ...
       +
       ?Bn;
         Sn
       +
       ?sonst;
         S
```

Selektion 2:
IF B1 THEN
  S1
ELSIF B2 THEN
  S2
. . .
ELSIF Bn THEN
  Sn
ELSE
  S
END (*IF*);

```
ITER_W  S
        (*WHILE B)
```

Iteration 1:
WHILE B DO
  S
END (*WHILE*);

Das einer FOR-Anweisung entsprechende Klammer-Konstrukt hat die folgende Form:

```
ITER_F [S
       (*:i IN Menge)
```

```
ITER_U [S
       (*UNTIL B)
```

Iteration 2:

```
REPEAT
  S
UNTIL B;
```

```
ITER_L [S
       (*:)  [S1
             *
             ?B;
               QUIT ITER_L
             *
             S2
```

Iteration 3:

```
LOOP
  S1;
  IF B THEN EXIT END;
  S2;
END (*LOOP*);
```

Man beachte: Das Klammerdiagramm ITER–L enthält eine explizite Angabe des Programmteils, der verlassen (bzw. abgebrochen) werden soll. (Die EXIT-Anweisung enthält diese Information implizit: Die Ausführung von EXIT bricht immer die innerste Schleife ab, in der sie vorkommt.) In Klammerdiagrammen kann QUIT mit dieser Wirkung (also Verlassen eines benannten Programmteils) immer benutzt werden. Die Umsetzung solcher Konstruktionen in MODULA-2 - Programme führt daher zu Prozeduren mit unter Umständen mehreren RETURN-Anweisungen.

Aussagen, die einen benannten Programmteil betreffen, können in Klammerdiagrammen an das obere und untere Ende der Klammer des entsprechenden Namens als Kommentar geschrieben werden:

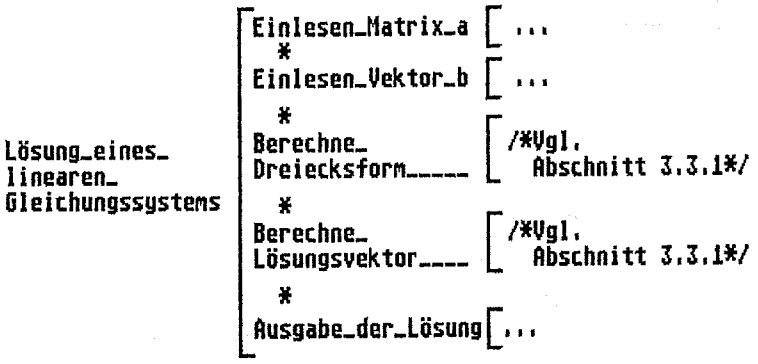
```
P_TEIL [/* oberer Kommentar */
       . . .
       [/* unterer Kommentar */
```

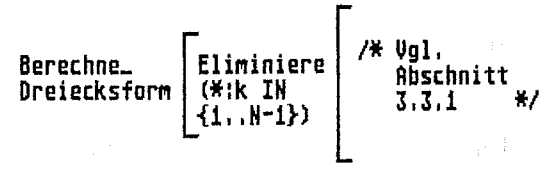
(Am oberen (bzw. unteren) Ende wird man sinnvollerweise Aussagen notieren, die sich auf die Situation vor (bzw. nach) der Ausführung des Programmteils beziehen. Wir erwähnen dies im Vorgriff auf die Ausführungen in Kapitel 4.)

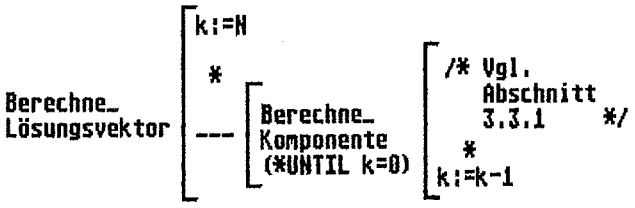
Kommentare können im übrigen beliebig zur Beschreibung von Programmteilen verwendet werden, wie es die folgenden Beispiele zeigen.

Auf eine wesentliche Eigenschaft von Klammerdiagrammen sei noch hingewiesen: Eine Klammer ist von genau einer "Strukturart", also entweder Sequenz

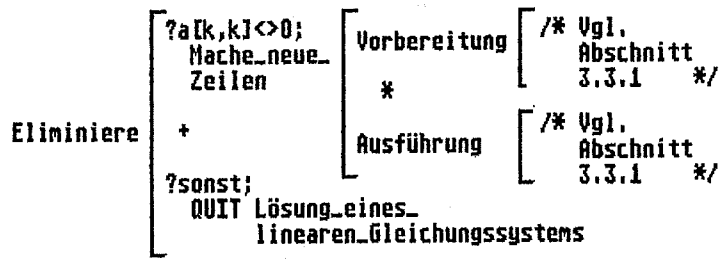
oder Selektion oder Iteration. Diese Strukturarten dürfen nicht "gemischt" innerhalb einer Klammer (das heißt "auf einer Ebene") auftauchen! (Ein Klammerdiagramm entspricht einem Baum, für dessen Knoten ein "Art-Attribut" definiert ist. Jeder Knoten ist darin von genau einer Art!) Nun die Beispiele:







Man beachte, daß wir in diesem Diagramm den "Dummy-Namen" "——" einsetzen mußten, um zu verhindern, daß innerhalb der Klammer mit dem Namen "Berechne–Lösungs–Vektor" Sequenz und Iteration "auf einer Ebene gemischt" werden. (Man kann in einem solchen Fall natürlich auch versuchen, einen sinnvollen neuen Namen zu erfinden.)



```
             ┌t:=b[k]
             │
             │ *   ┌?k<N; ┌Einsetzen  ┌
             │     │  ___ │(*:j IN    │t:=t-a[k,j]*x[j]
             │ ___ │   +  └{k+1,..,N}) └
Berechne_    │     │?sonst;
Komponente   │ *   └  t:=t/a[k,k]
             │
             │x[k]:=t
             │ *
             └k:=k-1
```

Im Beispiel "Rechnungs-Ausschrieb" verwenden wir einige der in [EHL] vorgeschlagenen textlichen Elemente der Klammerdiagramm-Technik. Sie gestatten es, bestimmte Programmteile (insbesondere solche, die mit Ein- und Ausgabe zu tun haben) in sehr kompakter Weise zu beschreiben, ohne bereits die Umsetzung in Anweisungen der Programmiersprache festzulegen.

```
             ┌Initialisierung [/*Vgl. Abschnitt 3.3.1*/
             │
             │  *
             │
Rechnungs_   │Verarbeitung___ [/*Vgl. Abschnitt 3.3.1*/
Ausschrieb   │
             │  *
             │
             └Abschluß_______ [/*Vgl. Abschnitt 3.3.1*/
```

Auch hier beschränken wir uns auf die weitere Ausarbeitung von "Initialisierung" und "Verarbeitung":

```
            ┌/*Eröffnung der Dateien "LIEFERUNG",...*/
            │
            │  *                ┌      ┌KundenNr (=lKuNr)
            │                   │Satz  │   #
            │IN.LIEFERUNG:      │      │ArtikelNr (=lArtNr)
            │LieferSatz (=lS)   │  +   │   #
            │                   │      └Stückzahl (=lStZ)
Initiali_   │  *                │
sierung     │                   └'eof'
            │?lS='eof';
            │  QUIT RechnungsAusschrieb
            │
            │  *
            │
            │/* "vorige Kunden-Nr."(kuNrAlt) mit
            └   "dummy"-Wert initialisieren */
```

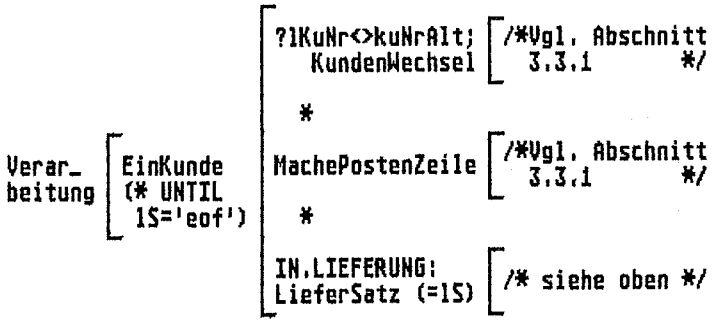
Zu beachten ist hier "IN.LIEFERUNG:...". "IN" ist eine Operation, die von irgendwoher ein nächstes Objekt besorgt. Das "Irgendwoher" ist nach dem Punkt (".") genauer bezeichnet: In diesem Fall ist es die Datei mit dem Namen "LIEFERUNG". Die Objekte, die von dort beschafft werden, sind von bestimmter Art. Diese Art erhält einen Namen (hier: "LieferSatz") und wird beschrieben: Im

Beispiel kann ein Objekt der Art "LieferSatz" entweder ein 'eof'-Signal sein oder (dies wird durch das Zeichen "+" ausgedrückt) ein Satz. Wenn es ein Satz ist, so besteht er aus drei Teilen, die man sich räumlich nebeneinander angeordnet vorstellen kann: "KundenNr", "ArtikelNr" und "Stückzahl". Das "Nebeneinander" wird durch das Zeichen "#" symbolisiert. Um im weiteren Verlauf von dem durch die Operation "IN" geholten Objekt und eventuell von seinen Teilen besser reden zu können, dürfen dem Objekt und seinen Teilen nach Bedarf Kurznamen gegeben werden. Die Schreibweise

"IN.LIEFERUNG:LieferSatz(=ls)..."

bedeutet also: Wir erhalten durch die Operation "IN" "vom Ort" Datei "LIEFERUNG" das nächste Objekt der Art "LieferSatz" und nennen es "ls". Diese Darstellung bringt implizit zum Ausdruck, daß wir uns die mit dem Lesevorgang geholten Objekte "merken". In den entsprechenden Nassi-Shneiderman-Diagrammen in Abschnitt 3.3.1 mußten wir diese Tatsache explizit notieren.

Einer Erklärung bedarf ferner die im obigen Diagramm verwendete Bedingungsform (**?1S='eof'; QUIT RechnungsAusschrieb**) innerhalb einer Sequenz. Tatsächlich ist dies eine (erlaubte) verkürzte Darstellung einer Selektion von der Art SEL-1 mit "**NONE**" als Alternative (s. o.).



Ebenso wie in Abschnitt 3.3.1 brechen wir den Detailentwurf an dieser Stelle ab. Insbesondere für das Beispiel des "Rechnungs–Ausschriebs" wäre eine weitere Verfolgung der hier begonnenen Entwicklung auch wenig sinnvoll. Obwohl dabei die Darstellung der Notationen unser Hauptanliegen war, ist es doch bedauerlich, daß keinerlei Überlegungen zu dem "Warum-gerade-so" der Struktur des Programms "Rechnungs-Ausschrieb" angestellt wurden. Überlegungen zum Entwurf von Programmen gerade dieser Art werden aber in Kapitel 5 im Mittelpunkt stehen. Dort wird sich zeigen, daß die hier dargelegte Struktur dem Problem eigentlich gar nicht gerecht wird, ja, daß sie durchaus als "ad hoc Bastelei" angesehen werden kann.

Literatur zu Kapitel 3

[CLM] Clocksin W.F. und C.S. Mellish: Programming in Prolog; Springer Verlag; Berlin, Heidelberg, New York; 1981

[EHL] Ehling, H.-J.: Evolutionärer System-Entwurf; in: Formale Modelle für Informationssysteme; Informatik-Fachberichte Nr. 21; Springer Verlag; Berlin, Heidelberg, New York; 1979

[HOS] Horowitz, E. und S. Sahni: Fundamentals of Computer Algorithms; Pitman Publishing Ltd.; London; 1979

[MAM] Martin, J. and C. McClure: Diagramming Techniques for Analysts and Programmers, Prentice-Hall; Englewood Cliffs; 1985

[NAS] Nassi, I. und B. Shneiderman: Flowchart techniques for structured programming; ACM SIGPLAN Notices; August 1973

[ORR] Orr, K. T.: Structured Systems Development; Yourdon Press; New York; 1977

[OTW] Ottmann, T. und P. Widmayer: Algorithmen und Datenstrukturen; Bibliographisches Institut; Mannheim; 1990

[STH] Stummel, F. und K. Hainer: Praktische Mathematik; Teubner Verlag; Stuttgart; 1971

[WAR] Warnier, J. D.: Logical Construction of Programs; Van Nostrand Reinhold; 1974

[WI2] Wirth, N.: Programm development by stepwise refinement; Communications of the ACM, Vol. 14, No. 4, pp. 221-227; April 1971

[WOO] Wood, D.: Paradigms and Programming with Pascal; Computer Science Press; Rockville; 1984

4 Programmieren durch Beweisen

In diesem Kapitel werden wir an die Bemerkungen in Abschnitt 2.1.1 über die "Korrektheit von Programmen" anknüpfen und dieses Thema vertiefen.

Um es vorwegzunehmen: Es wird uns hier nicht darum gehen, Eigenschaften vorhandener Programme zu untersuchen und entsprechende Aussagen zu beweisen. Programm-*Verifikation* im eigentlichen Wortsinne findet also nicht statt. Wir haben ja bereits früher argumentiert, daß derartige Ansätze zum Beispiel in einer industriellen Umgebung im allgemeinen nicht sehr vielversprechend sind.

Es geht uns vielmehr darum, neben die im dritten Kapitel besprochene Technik der "Schrittweisen Verfeinerung" eine weitere systematische Vorgehensweise bei der Entwicklung kleiner Programme bzw. bei der Formulierung "kritischer" Programmteile zu stellen. Damit zeigen wir zugleich einen weiteren Aspekt guten Programmierstils auf: die "Logik des Programmierens" (vgl. Einleitung zu Kapitel 2).

Die Vorgehensweise, die wir hier meinen, wird durch die Überschrift dieses Kapitels schon angedeutet: "Programmieren durch Beweisen". Etwas ausführlicher: Ein Programmierer möchte, indem er ein Programm schreibt, ein bestimmtes (selbst gesetztes oder vorgegebenes) Ziel erreichen. Wir empfehlen ihm, dieses Ziel sehr genau zu beschreiben, und sein Programm dann "aus dieser Beschreibung heraus" zu entwickeln. Auf diese Weise wird das Programm gewissermaßen zum Beweis dafür, daß das Ziel unter vorgefundenen Bedingungen erreichbar ist. Diese Anfangsbedingungen und eventuelle Zwischenziele müssen natürlich ebenfalls dokumentiert werden. Umgekehrt kann diese Dokumentation dann als "Beweis" für die Korrektheit des Programms angesehen werden. So entstehen das Programm und seine "Verifikation" Hand in Hand. Unser Programmierer wird gar nicht anders können: sein Programm wird qua Konstruktion korrekt!

Zur Verwirklichung dieser Empfehlung bedarf es freilich der Klärung einiger wichtiger Fragen. Zum Beispiel: "Was können denn Ziele, Zwischenziele und Anfangsbedingungen überhaupt sein und wie sind sie zu beschreiben?" Wir müssen uns ferner genauestens darüber klar werden, welchen Beitrag die Anweisungen der Programmiersprache zur Erreichung der Ziele und Zwischenziele leisten, was sie bewirken. Wir werden also die Bedeutung (die *Semantik*) zumindest der wichtigsten Anweisungsarten von MODULA-2 verstehen müssen, ehe wir daran gehen können, in der geschilderten Weise (und mit dem genannten hohen Anspruch) Programme zu entwickeln.

Damit ist die Struktur dieses Kapitels in groben Zügen vorgezeichnet. Sie entspricht in etwa dem Aufbau der Monographie "The Science of Programming" von D. Gries ([GRI]). Diese behandelt das Thema der "zielorientierten Entwicklung" von Programmen in aller Ausführlichkeit, allerdings ohne dabei ausschließlich die Konstrukte einer der bekannten Programmiersprachen zu verwenden.

Vielmehr werden spezielle Konstrukte eingeführt, welche die Eleganz der Darstellung zweifellos fördern. Wir beschränken uns darauf, einige wesentliche Aspekte "zielorientierter Programm-Entwicklung" mit den sprachlichen Mitteln von MODULA-2 (aber etwas ausführlicher als z.B. in [SAL]) auszuarbeiten. Der damit erzwungene Verlust an Eleganz wird so immerhin teilweise durch "größere Praxisnähe" wettgemacht. Allerdings werden die Beispiele dieses Kapitels noch fragmentarischer sein als die Beispiele in Kapitel 3.

Mit "Schrittweiser Verfeinerung" haben wir automatisch den Teil c) der Kommentar-Regel in Abschnitt 2.2.1 befolgt. Wir werden sehen, daß "Programmieren durch Beweisen" die Einhaltung des Teils d) dieser Regel wesentlich erleichtert, wenn nicht sogar erzwingt.

Es versteht sich von selbst, daß auch die in diesem Kapitel zur Debatte stehende Technik als eine mögliche "Ausführungsbestimmung" des in der Einleitung zu Kapitel 3 aufgestellten "obersten Gebots" aufgefaßt werden kann.

4.1 Prädikate

Wenn Programme überhaupt etwas bewirken, dann unter anderem doch dies:

(i) Änderung des Inhalts irgendwelcher (interner oder externer) Speicher;

(ii) Erzeugung von Ausgaben (Outputs) in Form von Signalen, die in geeignete Kanäle fließen;

(iii) Entgegennahme von Eingaben (Inputs) aus geeigneten Kanälen.

(Unter einem *Kanal* vestehen wir hier ein Medium zum sequentiellen Transport von Daten.)

Ein einziges Programm kann - wie wir wissen - Wirkungen aller drei Arten hervorrufen. Wir wollen uns in diesem Kapitel mit Wirkungen der Art (i) beschäftigen. (Wirkungen der Arten (ii) und (iii) werden im nächsten Kapitel im Vordergrund stehen.)

4.1.1 Speicher und Variable

Speicher (insbesondere interne Speicher) werden in MODULA-2 (wie in allen imperativen höheren Programmiersprachen) durch Variablen repräsentiert, die mittels eines Variablennamens adressiert werden. Der Wert, den eine Variable zu einem gegebenen Zeitpunkt hat, ist der Inhalt des durch diese Variable repräsentierten Speichers. (Wir haben uns daher der Sprechweise "Inhalt einer Variablen" schon früher häufig bedient!) Wir stellen uns nun vor, daß wir den Ablauf eines Programms Schritt für Schritt beobachten können. Tatsächlich heißt dies, daß wir uns nach jedem Schritt die Werte der Variablen dieses Programms anschauen und eine Aussage über den jeweils erreichten *Zustand* machen können.

Seien zum Beispiel x,y und z die Variablen des Programms. Dann kann eine solche Aussage etwa die Form

(1) "x = ‹Wert von x›, y = ‹Wert von y›, z = ‹Wert von z›"

annehmen. Dabei ist ‹Wert von ...› ein Objekt vom Typ der entsprechenden Variablen. Andere Aussagen sind möglich:

(2) "x=y"

bedeutet, daß (im aktuellen Zustand) die Variablen x und y den gleichen Wert haben. Mit

(3) "x‹y AND y‹z"

drücken wir aus, daß die aktuellen Werte von x,y und z den Beziehungen

‹Wert von x› ‹ ‹Wert von y› und ‹Wert von y› ‹ ‹Wert von z›

genügen.

Die Aussage (1) ist sehr konkret: Wenn wir davon ausgehen, daß uns vom ganzen Speicher nur der durch x,y und z repräsentierte Teil interessiert, dann beschreibt (1) genau einen Speicherzustand. Dieser ist also - anders gesagt - durch eine Menge von Paaren der Form

(‹Variablenname›,‹Variablenwert›)

gegeben, im Beispiel durch

[(x,‹Wert von x›),(y,‹Wert von y›),(z,‹Wert von z›)].

(Es ist klar, daß jeder Variablenname dabei nur einmal auftreten darf. Mit Hilfe der klassischen mathematischen Begriffe "Menge" und "Abbildung" und mit den entsprechenden Notationen kann der hier eingeführte Begriff des Speicherzustands selbstverständlich weiter präzisiert werden. Wir wollen den Leser damit nicht langweilen und vertrauen darauf, daß ihm die gegebene Erklärung zu einem für unsere Zwecke völlig ausreichenden intuitiven Verständnis verhilft.)

Die Aussagen (2) und (3) dagegen können in einem solchen konkreten Speicherzustand zutreffen oder nicht, sie können WAHR oder FALSCH sein. Umgekehrt kann jede dieser Aussagen natürlich auch als Beschreibung einer Menge von Speicherzuständen aufgefaßt werden: als Beschreibung der Menge derjenigen Speicherzustände nämlich, für welche die Aussage wahr ist. Dies setzt voraus, daß wir in der Lage sind, eine Aussage bezogen auf einen gegebenen Speicherzustand auszuwerten, ihren *Wahrheitswert* festzustellen. Ein Beispiel:

Seien x,y und z (die Namen von) Variable(n) vom Typ INTEGER. Wir betrachten zwei Speicherzustände:

Z1 = {(x,1),(y,1),(z,0)} und Z2 = {(x,-2),(y,0),(z,8)}.

Im Speicherzustand Z1 hat die Aussage "x=y" offenbar den Wert WAHR (TRUE), im Zustand Z2 aber hat diese Aussage den Wert FALSCH (FALSE). Die Aussage

"x<y AND y<z" dagegen ist FALSCH in Z1 und WAHR in Z2. Nehmen wir andererseits die Aussage

(4) "y=0 OR x/y=1".

Im Zustand Z1 hat sie den Wert WAHR, im Zustand Z2 gilt freilich "y=0". "x/y=1" dagegen ist - wegen y=0 - gar nicht definiert. Dennoch scheint es wünschenswert, auch solche Aussagen zuzulassen. (Immerhin ist (4) ja syntaktisch korrekt gebildet.) Wir führen daher den zusätzlichen Wahrheitswert UNDEFINIERT ein. Um dessen konkrete Behandlung kümmern wir uns etwas später.

Die Feststellung des Wahrheitswerts einer Aussage der bisher betrachteten Arten kann in völlig systematischer Weise geschehen. Auf eine detaillierte Darstellung dieser Systematik werden wir allerdings verzichten. Sie müßte auf einer formalen Definition der Syntax von Aussagen über Variablen beruhen. (Es sei in diesem Zusammenhang auf einschlägige Lehrbücher über Logik - z.B. [BEN] - verwiesen.) Wir verwenden zunächst - wie es naheliegt - die Regeln zur Bildung boolescher Ausdrücke in MODULA-2.

Mit diesen Regeln formulierbare Aussagen über Variable werden im folgenden *Prädikate* genannt.

Sei P(x,y,z,...) ein Prädikat und sei SZ die Menge aller Zustände des Speichers, der durch x,y,z,... repräsentiert wird. Dann ist P(x,y,z,...) also nichts anderes als eine Abbildung

$$P(x,y,z,...): SZ \rightarrow \{T,F,U\}.$$

(Dabei steht T für TRUE, F für FALSE und U für UNDEFINIERT.)

Die Teilmenge von SZ, für deren Elemente P(x,y,z,...) den Wert TRUE hat, heißt *Extension* des Prädikats P(x,y,z,...):

$$EXT(P(x,y,z,...)) = \{ Z \mid Z \text{ IN } SZ, P(x,y,z,...)(Z)=T \}.$$

Für die konstanten Prädikate TRUE und FALSE gilt:

$$EXT(TRUE) = SZ \text{ und } EXT(FALSE) = \{ \}.$$

(Dabei bezeichnet "{ }" die leere Menge.)

4.1.2 Operationen mit Prädikaten

Zur Bildung von Prädikaten aus gegebenen Prädikaten haben wir uns im Falle der Beispiele (3) und (4) bereits der bekannten Operatoren AND und OR bedient. Ferner liefert der Operator NOT zu einem Prädikat P das Prädikat NOT P. Mit Hilfe des Begriffs der Extension läßt sich die Wirkung dieser Operationen wie folgt erklären:

Seien P, P1 und P2 Prädikate. Dann gilt:

$$EXT(P1 \text{ AND } P2) = EXT(P1) \cap EXT(P2)$$
$$EXT(P1 \text{ OR } P2) = EXT(P1) \cup EXT(P2)$$

$$EXT(NOT\ P) = \overline{EXT(P)}$$

Dabei sind $\cup$, $\cap$ und $\overline{\ }$ die bekannten Mengen-Operatoren für Vereinigung, Durchschnitt und Komplement. (Die Komplementbildung erfolgt bezüglich der Menge aller für P relevanten Speicherzustände.)

Die Operatoren AND, OR und NOT liefern bei der Auswertung eines Prädikats nur dann ein definiertes Ergebnis, wenn die Werte der Operanden T oder F sind. Um auch den Fall des Wertes U abhandeln zu können, führen wir zusätzlich die Operatoren CAND (bedingtes AND) und COR (bedingtes OR) wie folgt ein:

Seien P1 und P2 Prädikate. "P1 CAND P2" und "P1 COR P2" erhalten Wahrheitswerte entsprechend der folgenden Tabelle:

P1	P2	P1 CAND P2	P1 COR P2
T	T	T	T
T	F	F	T
T	U	U	T
F	T	F	T
F	F	F	F
F	U	F	U
U	T	U	U
U	F	U	U
U	U	U	U

Diese Zuordnung wurde so getroffen, daß die Auswertung von P1 CAND P2 und P1 COR P2 gleichbedeutend ist mit der Auswertung von

IF P1 THEN P2 ELSE F (für P1 CAND P2) und
IF P1 THEN T ELSE P2 (für P1 COR P2).

Dabei ist "IF P1 THEN ..." undefiniert, wenn der Wert von P1 U ist. Man beachte, daß die Werte von P1 AND P2 und P1 OR P2 durch die Auswertung der gleichen bedingten Ausdrücke gefunden werden können.

Die Operatoren CAND und COR sind nicht kommutativ. Es gelten aber Assoziativ- und Distributivgesetze sowie die De Morgan'schen Regeln. Wir gehen darauf im einzelnen nicht ein und verweisen auch hier auf einschlägige Literatur über formale Logik (s.o.).

Von großem Interesse sind einige weitere Operationen zur Erzeugung von Prädikaten aus gegebenen Prädikaten.

Sei P(...) ein Prädikat (zum Beispiel "x<y AND y<z"). Seien ferner vn ein Variablenname und Q ein Ausdruck vom Typ der Variablen mit dem Namen vn. Falls

vn (zum Beispiel y) - wie wir der Einfachheit halber bisher immer angenommen haben - vom Typ INTEGER ist, so ist Q also ein arithmetischer Ausdruck über Variablen und Konstanten vom Typ INTEGER (etwa "x-y"). Dann bezeichnet

P(...)(vn->Q)

dasjenige Prädikat, welches entsteht, wenn in P der Variablenname vn überall dort, wo er vorkommt, "textuell" ersetzt wird durch Q.

Im Beispiel führt die *textuelle Ersetzung* von y durch "x-y" zu dem Prädikat:

"x<x-y AND x-y<z"

Eine Präzisierung der Vorschriften für die Anwendung der Operation der textuellen Ersetzung können wir erst nach der Besprechung der folgenden beiden Operationen vornehmen. Zum Verständnis dieser Operationen ist es jedoch nützlich, textuelle Ersetzung bereits zu kennen.

Wir betrachten wiederum ein Prädikat P(x,y,z), wobei x,y und z Variablen vom Typ INTEGER bezeichnen. Zur Illustration verwenden wir wieder Beispiel (3). Textuelle Ersetzung von x durch eine Konstante I ergibt das Prädikat

P(I,y,z) = P(x,y,z)(x->I)

mit den Variablen y und z. Im Beispiel (3) führt die Ersetzung von x durch die INTEGER-Konstante 1 zu dem Prädikat

"1<y AND y<z".

Nehmen wir nun eine Teilmenge von INTEGER, die durch ein Prädikat R(x) charakterisiert ist, und bilden P(I,y,z) für jedes I in EXT(R). Ist R(x) das Prädikat

"1<=x AND x<4",

so liefert Beispiel (3) die Prädikate

(5) "1<y AND y<z", "2<y AND y<z", "3<y AND y<z".

Die Verknüpfung dieser Prädikate mit OR ergibt:

(6) (1<y AND y<z) OR (2<y AND y<z) OR (3<y AND y<z).

(6) hat genau dann den Wert T, wenn mindestens eines der durch OR verbundenen Prädikate den Wert T hat. Mit anderen Worten: (6) ist genau dann WAHR, wenn ein Wert I der Variablen x *existiert*, für den sowohl

"1<=I AND I<4" als auch "I<y AND y<z" gilt.

Für das Prädikat (6) führen wir die abkürzende Schreibweise

"EX x: 1<=x AND x<4: x<y AND y<z"

ein. In dem allgemeineren Fall schreiben wir:

(7) "EX x: R(x): P(x,y,z)".

(7) erhält genau dann den Wert T, wenn ein Wert I von x *existiert*, für den sowohl R(I) als auch P(I,y,z) gilt. Der Operator "EX" wird *Existenz-Quantor*

genannt, und wir sagen, daß (7) durch EX-Quantifizierung bezüglich R(x) aus P(x,y,z) entstanden ist.

Andererseits können wir die in (5) aufgelisteten Prädikate auch durch AND verknüpfen und erhalten:

(8) (1<y AND y<z) AND (2<y AND y<z) AND (3<y AND y<z).

Dieses Prädikat ist genau dann WAHR, wenn jedes der in Klammern gesetzten Prädikate den Wert T hat. Wir kürzen (8) ab zu:

"ALL x: 1<=x AND x<4: x<y AND y<z".

Entsprechend schreiben wir wieder im allgemeineren Fall:

(9) "ALL x: R(x): P(x,y,z)".

(9) erhält genau dann den Wert T, wenn *für alle* Werte I von x, für die R(x) WAHR ist, auch P(I,y,z) gilt. Der Operator "ALL" wird *All-Quantor* genannt, und wir sagen, daß (9) durch ALL-Quantifizierung bezüglich R(x) aus P(x,y,z) entstanden ist.

Mit Hilfe einer der De Morgan'schen Regeln kann das Ergebnis der Anwendung des All-Quantors durch einen mit dem Existenz-Quantor gebildeten Ausdruck dargestellt werden:

ALL x: R(x): P(x,y,z) = NOT(EX x: R(x): NOT P(x,y,z)).

(Das Gleichheitszeichen bedeutet hier, daß beide Prädikate äquivalent sind in dem Sinne, daß sie die gleiche *Extension* haben.)

Wir geben einige weitere Beispiele für mit der EX- bzw. ALL-Quantifizierung gebildete Prädikate, deren Bedeutung sich der Leser jeweils klarmachen möge:

"EX i: 0<=i AND i<10: i*j=10"
"EX i: 0<=i AND i<10: (EX j: 0<=j AND j<10: i*j=100)"
"ALL i: 1<=i AND i<4: i*x>0"
"ALL n: n>0: (EX p: p>n: 1<p AND (ALL j: 1<j AND j<p: p MOD j <> 0))"
"ALL m: Mensch(m): Sterblich(m)"

Seien a und b Variable vom Typ "ARRAY [0..9] OF INTEGER".

"ALL i: 0<=i AND i<10: (EX j: 0<=j AND j<10: a[i]=b[j])"

Die Beispiele zeigen unter anderem, daß EX- und ALL-Quantifizierungen beliebig gemischt werden können.

Es sollte auch deutlich geworden sein, daß Variable, über deren Werte eine Quantifizierung erfolgt, ihren "Variablencharakter" verlieren.

Um dies einzusehen, kehren wir zu unserer intuitiven Vorstellung von Variablen als Speicher-Repräsentanten zurück. Die Speicherzustände, für die das Prädikat "i*j=10" den Wert T liefert, sind:

{(i, 1),(j, 10)}, {(i, 2),(j, 5)}, {(i,5),(j,2)}, {(i, 10),(j, 1)},
{(i,-1),(j,-10)}, {(i,-2),(j,-5)}, {(i,-5),(j,-2)}, {(i,-10),(j,-1)}.

Dagegen kommt es für die Auswertung des Prädikats

(10) "EX i: 0<=i AND i<10: i*j=10"

auf den Inhalt einer Variablen i überhaupt nicht mehr an. Es bezieht sich nur noch auf die Variable j und hat den Wert T für alle Speicherzustände, in denen der Inhalt von j entweder 2, 5 oder 10 ist. Man sagt, daß die Variable i durch den Existenz-Quantor "EX" *gebunden* wird; ihr Name steht lediglich als "Wertbezeichner" für ein Objekt vom Typ INTEGER aus dem Bereich (0..9). Der Bezeichner i ist also beliebig austauschbar gegen einen von j verschiedenen Bezeichner. Das Prädikat (10) ist daher zum Beispiel gleichbedeutend mit

"EX k: 0<=k AND k<10: k*j=10".

Entsprechendes gilt natürlich für den All-Quantor "ALL": Auch er *bindet* eine Variable und macht aus ihrem Namen einen bloßen "Wertbezeichner". Das Prädikat "i*x>0" hat den Wert T für alle Speicherzustände

[(i,<Wert von i>),(x,<Wert von x>)],

für die <Wert von i> und <Wert von x> beide größer oder beide kleiner als 0 sind. Es ist also gleichbedeutend mit

"(i>0 AND x>0) OR (i<0 AND x<0)".

Das Prädikat

(11) "ALL i: 1<=i AND i<4: i*x>0",

in dem i durch den *All-Quantor gebunden* ist, liefert in allen Speicherzuständen den Wert T, in denen der Inhalt von x größer ist als 0. Es ist äquivalent mit "x>0". Der "Wertbezeichner" i kann wieder durch einen beliebigen, von x verschiedenen Bezeichner ersetzt werden:

"ALL k: 1<=k AND k<4: i*x>0".

Wir fassen zusammen: Die in einem Prädikat vorkommenden Namen sind entweder

- Variablenbezeichner,
- Literale oder
- Wertbezeichner.

Ein Variablenbezeichner wird durch die Bindung an einen Quantor ("EX" oder "ALL") zu einem Wertbezeichner. Im Gegensatz zu den gebundenen (Wert-)Bezeichnern werden Variablenbezeichner daher auch "freie Bezeichner" genannt.

Es ist nun keineswegs verboten, innerhalb eines Prädikats einen Bezeichner sowohl als Variablenbezeichner (also "frei") als auch als Wertbezeichner (also "gebunden") zu verwenden. Beispiel:

P(i,j,x) = (i>j) AND (ALL i: 1<= i AND i<10: i*x>0)

In "(i>j)" ist der Bezeichner i frei, also ein Variablenbezeichner. In dem "ALL-Ausdruck" "(ALL i: ...)" dagegen ist der Bezeichner i an den All-Quantor gebunden, also ein Wertbezeichner. Von einer textuellen Ersetzung von i (durch z.B. i*x) darf aber nur das "freie i" berührt werden, da - wie wir gesehen haben - das "gebundene i" ja auch ganz anders heißen könnte!

Es ist daher sinnvoll, beim Anschrieb von Prädikaten darauf zu achten,

- gleichlautende Variablen- und Wertbezeichner zu vermeiden und darüberhinaus
- einen Wertbezeichner nur im "Gültigkeitsbereich" (englisch "Scope") eines einzigen Quantors zu verwenden.

In der Form

P(i,j,x) = (i>j) AND (ALL k: 1<=k AND k<10: k*x>0)

stiftet das Prädikat P(i,j,x) bei der textuellen Ersetzung i->i*x keine Verwirrung mehr.

Die zu Beginn dieses Abschnitts gegebene Erklärung der textuellen Ersetzung nahm wohlweislich bereits ausschließlich Bezug auf Variablennamen, also auf freie Bezeichner. Zur Anwendung textueller Ersetzung auf Prädikate mit durch Quantoren gebundenen Wertbezeichnern jedoch müssen wir etwas schärfere Vorschriften erlassen.

Sei b eine Variable vom Typ ARRAY [0..9] OF INTEGER. Wir betrachten:

P(x,y,b) = (x<y) AND (ALL i: 0<=i AND i<10: b[i]<y).

Angenommen, wir wollen (neben x,y und b) eine weitere Variable i in's Spiel bringen und feststellen, daß sowohl der Inhalt von x als auch die Inhalte der durch das ARRAY b repräsentierten Speicher kleiner sind als y-i. Die "kritiklose" textuelle Ersetzung y->y-i liefert:

P(x,y,b,i) = (x<y-i) AND (ALL i: 0<=i AND i<10: b[i]<y-i).

Das ist aber sicher nicht der gewünschte Ausdruck: Das i in der zweiten Ersetzung gerät sozusagen in den "Bann" des All-Quantors, wird von ihm gebunden und zum bloßen Wertbezeichner.

Um dies zu vermeiden, ist es notwendig, vor einer textuellen Ersetzung alle diejenigen Wertbezeichner umzubenennen, zu denen es gleichlautende Variablenbezeichner im Ersetzungsausdruck gibt.

Unter Beachtung dieser Regel führt die Ersetzung y->y-i zu:

P(x,y,b,i) = (x<y-i) AND (ALL k: 0<=k AND k<10: b[k]<y-i).

Schließlich darf textuelle Ersetzung nur dann angewandt werden. wenn sich dadurch kein syntaktisch inkorrekter Ausdruck ergibt. Für das obige Beispiel wäre dies etwa mit der Ersetzung b->x+y der Fall.

4.1.3 Prädikate als Kommentare

Nachdem wir nun in einiger Ausführlichkeit Notationen für den Umgang mit Prädikaten entwickelt haben, kehren wir zum Ausgangspunkt unserer Überlegungen zurück. Wir fragen nach dem Nutzen, den diese Notationen für die Realisierung der Vorgehensweise haben, die in der Vorrede zu diesem Kapitel angedeutet wurde.

Die erste Antwort ist sehr einfach: Sie sind geeignet, bei der Abfassung eines Programms an "kritischen" Punkten Aussagen zu formulieren, denen die Inhalte (bzw. die Werte) der vom Programm bearbeiteten Variablen an diesen Stellen genügen müssen. Solche Aussagen können als Kommentare im Programmtext eingefügt werden. Mit ihnen "versichert" der Programmierer gewissermaßen (sich selbst oder einem anderen Prüfer), daß nach Ausführung des Programms bis zu einer gegebenen Stelle (im Programmtext) ein Speicherzustand erreicht ist, der irgendwelche gewünschten Bedingungen erfüllt, für den ein entsprechendes Prädikat also den Wert T annimmt.

Der Prüfer kann nun - im Prinzip - das Programm an dieser Stelle anhalten, das im Kommentar enthaltene Prädikat auswerten und so die Behauptung (bzw. "Zusicherung", englisch: "*Assertion*") des Programmierers bestätigen oder widerlegen.

Der Prüfer könnte freilich auch der Rechner selbst sein, dem der Compiler für jeden formalen "Behauptungskommentar" (*Assertion*) Code zur Auswertung des Prädikats generiert hat. Ergibt die Ausführung dieses Codes den Wert TRUE, so ist alles in Ordnung und der Rechner setzt seine Arbeit fort, als wäre nichts geschehen; ist das Ergebnis FALSE, so wird der Rechner seine Arbeit abbrechen und (freundlicherweise) Informationen bereitstellen, mit deren Hilfe der Fehler vielleicht gefunden werden kann.

Das folgende Beispiel, das wir sinngemäß aus [GRII] zitieren, demonstriert diesen Gebrauch von "Behauptungskommentaren".

```
PROCEDURE Division((*IN*) x,y:INTEGER;(*OUT*)VAR q,r:INTEGER);
BEGIN
  (** x>=0 AND y>0 *)                          (*1*)
  r:=x; q:=0;
  (** r>=0 AND x=q*y+r *)                      (*2*)
  WHILE r>y DO
    r:=r-y;
    q:=q+1
    (** r>=0 AND x=q*y+r *)                    (*2*)
  END (*WHILE*)
  (** r>=0 AND y>r AND x=q*y+r *)              (*3*)
END Division;
```

Nehmen wir nun an, daß der Compiler durch die spezielle Kommentar-Eröffnung "(**" dazu veranlaßt werde, den nachfolgenden Kommentarinhalt als Prädikat zu interpretieren und für diesen Auswertungs-Code - wie oben beschrieben - zu generieren. Der Ablauf der Prozedur wird nun zum Beispiel immer dann gestoppt, wenn auf die in x und y übergebenen Parameter die Behauptung (*1*) nicht zutrifft. Mit der Behauptung (*3*) wird versucht, so scharf wie möglich die Bedingungen zu fassen, denen q (der Quotient) und r (der Rest) nach einer Division genügen müssen. Mit hinreichend häufiger Benutzung der Prozedur (mit verschiedenen Werten der Eingabe-Parameter x und y, versteht sich) wird vermutlich einmal der Fall eintreten, daß ein Programmabbruch gerade an dieser Stelle geschieht. (Der Leser ist aufgefordert, Werte für x und y zu finden, derart daß (*3*) nicht erfüllt wird!)

Ein Programmabbruch nach Behauptung (*1*) ist nicht weiter tragisch. Es gibt einige Möglichkeiten, damit fertig zu werden. Zum Beispiel könnten wir einem Benutzer der Prozedur "Division" vorschreiben, vor deren Aufruf zunächst einmal zu prüfen, ob die zu übergebenden Wertparameter der Bedingung (*1*) genügen.

Ein Programmabbruch nach Behauptung (*3*) jedoch sollte für den Verfasser der Prozedur "Division" peinlich sein. Offenbar hat er ja mit (*3*) genau das beschrieben, was seine Prozedur leisten soll. Da sie - wie der gelegentliche Abbruch zeigt - aber nicht immer das Gewünschte tut, muß er einen Fehler gemacht haben. Sein Programm ist nicht korrekt.

Gemäß unserer Skizze "zielorientierter Programmentwicklung" müßte es nun freilich möglich sein, aus der in (*3*) formulierten *Nachbedingung* programmiersprachliche Anweisungen zu ermitteln bzw. zu begründen, deren Ausführung immer einen Zustand ergibt, in dem die *Nachbedingung* erfüllt ist. Techniken hierfür werden wir in Abschnitt 4.3 kennenlernen.

Vorerst halten wir fest: Wir nennen die im Beispiel-Programm aufgestellte Behauptung (*3*) eine *Nachbedingung*, weil sie den Zustand beschreibt, der *nach* Ausführung der Prozedur herrschen soll. Sie ist also eine *Spezifikation* des Ergebnisses dieser Prozedur. Oftmals kennen wir auch die *Vorbedingung*, das heißt eine Beschreibung des Zustands, der *vor* Beginn der Ausführung einer Prozedur besteht.

Vorbedingung und Nachbedingung zusammen spezifizieren dann das Verhalten einer Prozedur (bzw. eines Programmstücks oder eines ganzen Programms), insofern es sich auf die Änderung von Speicherzuständen bezieht.

Wir bedienen uns zur Formulierung solcher Spezifikationen der folgenden Notation:

(* VB *)	<-	VORBEDINGUNG
Prog	<-	Prozedur (bzw. Programmstück)
(* NB *)	<-	NACHBEDINGUNG

oder auch:

(* VB *) Prog (* NB *).

Damit meinen wir:

> "Wenn die Ausführung der Prozedur (bzw. des Programmstücks) P in einem Speicherzustand beginnt, für den das Prädikat VB den Wert T hat, so endet die Prozedur (das Programmstück) nach endlicher Zeit und der dann erreichte Speicherzustand erfüllt das Prädikat NB."

In der bisherigen Diskussion haben wir die Behauptung (*2*), welche in unserer Divisions-Prozedur gleich zweimal auftaucht, keiner Erwähnung gewürdigt. Sie wird sowohl unmittelbar vor Eintritt in die WHILE-Schleife als auch nach jeder Abarbeitung des "Schleifen-Rumpfes" überprüft. Mithin bringt sie offenbar eine Eigenschaft zum Ausdruck, welche durch eben diese Abarbeitung des Schleifenrumpfes nicht verändert werden darf, welche also - wie man auch sagt - "invariant" sein muß. Auf die wichtige Rolle, die Behauptungen (bzw. Prädikate) wie (*2*) bei der Herleitung korrekter WHILE-Schleifen spielen, können wir an dieser Stelle noch nicht näher eingehen. Erst in den Abschnitten 4.3.2 und 4.3.3 werden wir darüber mehr erfahren.

Man macht sich rasch klar, daß bei der Angabe von VB und (insbesondere) NB große Sorgfalt angebracht ist, wenn man vermeiden will, daß eine spezifizierte Aufgabe von unbeabsichtigt trivialen Programmen erledigt werden kann. Ein Beispiel:

(* a>=0 AND b>=0 *)
Prog
(* z=a*b *)

Ein Programmstück, welches diese Spezifikation trivialerweise erfüllt, ist:

"a:=0; b:=0; z:=0;"

Natürlich sind wir - zu Recht - der Meinung, daß ein Programmierer, der diese Lösung vorlegt, unsere Spezifikation gründlich mißverstanden hat. Zwar haben wir nicht ausdrücklich gesagt, daß die Inhalte der Variablen a und b NICHT verändert werden dürfen, doch sollten wir darauf bestehen, daß ein Programm nicht MEHR tut, als tatsächlich verlangt wird. (Es gilt: "Was nicht erlaubt ist, ist verboten!")

Um dem Programmierer jeden Anlaß zu nehmen, das oben angegebene triviale Programmstück zu rechtfertigen, hätten wir allerdings spezifizieren können:

(* a=A AND b=B AND a>=0 AND b>=0 *)
Prog
(* a=A AND b=B AND z=a*b *).

Hier haben wir Großbuchstaben als Bezeichner für zwar unbestimmte aber feste Werte der Variablen a und b benutzt. (Dies entspricht im übrigen der in Abschnitt 2.2.1 vereinbarten Konvention über die Groß- und Kleinschreibung von Bezeichnern in MODULA-2 - Programmen!) Wie bei (durch Quantoren) gebundenen Bezeichnern handelt es sich hier um Wertbezeichner: Ein mit einem Großbuchstaben beginnender Name bezeichnet ein Objekt, das Inhalt einer Variablen sein kann, und überall, wo er vorkommt, bezeichnet er dasselbe Objekt. Diese Verwendung von Wertbezeichnern ist sehr nützlich für die Spezifikation von Speicherzustandsänderungen, bei denen etwa die Inhalte von Variablen nur "hin- und hergeschoben" werden.

Zum Beispiel könnte die Spezifikation eines Programmstücks, welches die Inhalte zweier Variablen x und y (die beide vom gleichen Typ sind) vertauscht, folgendermaßen aussehen:

```
(* x=X AND y=Y *)
   Prog
(* x=Y AND y=X *).
```

Ein zweites Beispiel hierzu: Sei a eine Variable vom Typ ARRAY [0..9] OF INTEGER und A Bezeichner eines Objekts von diesem Typ. Die Spezifikation

```
(* a=A *)
   Prog
(* perm(a,A) AND (ALL i: 0<=i AND i<9: a[i]<=a[i+1]) *)
```

fordert ein Programmstück, welches die Inhalte der "Zellen" der Array-Variablen a so umverteilt, daß sie mit wachsendem "Zellenindex" jedenfalls nicht kleiner werden. Es ist also nichts anderes als die Spezifikation einer "Sortier-Routine". ("perm(a,A)" ist dabei die abkürzende Schreibweise für ein Prädikat, welches zum Ausdruck bringt, daß a die einzelnen Werte von A in Permutation enthält; zusammen mit "a=A" sichert es, daß alle Werte, die sich zu Beginn in a befinden, auch am Ende noch in a sind.)

Wir beschließen diesen Abschnitt mit einem Hinweis zur praktischen Arbeit mit Behauptungen über Speicherzustände. Es ist ja im allgemeinen nicht davon auszugehen, daß ein Compiler (wie oben idealerweise angenommen) die Möglichkeit der Einbeziehung spezieller Kommentare bietet. Immerhin könnten solche Kommentare Konstrukte (wie CAND, COR, ALL und EX) enthalten, die über den Sprachumfang von MODULA-2 (oder irgendeiner anderen prozeduralen Programmiersprache) weit hinausgehen.

Behauptungen jedoch, die nur boolesche Ausdrücke enthalten, welche gemäß der MODULA-2 - Syntax aufgebaut sind, können sehr wohl in der oben beschriebenen Weise zur Überprüfung eines Programms herangezogen werden. Dazu notieren wir eine solche Behauptung nicht als Kommentar eingekleidet, sondern übergeben sie - als booleschen Ausdruck - einer vorher (allgemein zugänglich) deklarierten Prozedur (vgl. auch [SAL]):

```
PROCEDURE Assertion(praedikat:BOOLEAN; nr:CARDINAL);
BEGIN
 IF NOT praedikat THEN
  WriteLn;
  WriteString("***BEHAUPTUNG NR:");
  WriteCard(nr,6);
  WriteString(" TRIFFT NICHT ZU");
  WriteLn;
  HALT
 END (*IF*)
END Assertion;
```

In der Prozedur "Division" könnte also der Behauptungskommentar ersetzt werden durch den Aufruf

"Assertion((r>=0) AND (y>r) AND (x=q*y+r),3)".

Wurden der Prozedur "Division" in x und y zum Beispiel die Werte 6 und 3 übergeben, so führt dieser Aufruf von "Assertion" zur Meldung

***BEHAUPTUNG NR: 3 TRIFFT NICHT ZU

und dann zum Abbruch des Programms.

4.2 Semantik

Die im vorigen Abschnitt eingeführten Begriffe, Notationen und Sprechweisen sind - wie sich herausstellte - zur Präzisierung der Ziele eines Programmiervorhabens recht gut geeignet, sofern sich dieses Vorhaben auf die Änderung von Speicherzuständen beschränkt. In diesem Abschnitt werden wir Prädikate, Vorbedingungen und Nachbedingungen dazu benutzen, die Wirkungen einiger wichtiger Anweisungsarten der Sprache MODULA-2 exakt zu beschreiben, um so eine solide Grundlage für "zielorientierte Programmentwicklung" zu schaffen. Wir beschränken uns dabei auf Zuweisungen, Selektions-Anweisungen und die WHILE-Anweisung.

4.2.1 Schwächste Vorbedingungen

Sei S eine Anweisung oder ein Programmstück. Wie könnte man einem Lernenden die *Bedeutung* von S verständlich machen? Eine Möglichkeit wäre es sicherlich, ihm die Wirkung der Anweisung (bzw. des Programmstücks) anhand der Veränderungen des Speicherzustands zu erklären, die S hervorruft. Zum Beispiel könnten wir ihm sagen:

> "Wenn für einen Speicherzustand das Prädikat P zutrifft, dann schafft S einen Speicherzustand, in dem das von P und S irgendwie abhängende Prädikat Q gilt."

Wir würden also angeben, in welcher Weise S die *Vorbedingung* P zu einer *Nachbedingung* Q "transformiert".

In Anbetracht der Tatsache jedoch, daß wir bei der Programmentwicklung die Absicht haben, "das Pferd vom Schwanze her aufzuzäumen", also bei der Nachbedingung anzusetzen, die das Ergebnis des Ablaufs eines Programmstücks spezifiziert, ist eine solche Erklärung vermutlich nicht sehr hilfreich.

Wir sollten es umgekehrt versuchen und nach den allgemeinsten Voraussetzungen fragen, unter denen die Ausführung von S einen Speicherzustand herstellt, für den ein Prädikat Q gilt. Diese Voraussetzungen wären durch ein Prädikat P zu präzisieren, welches

> "die Menge EXT(P) aller Speicherzustände beschreibt, die die Eigenschaft haben, daß die Ausführung von S, beginnend in einem Zustand Z IN EXT(P), nach endlicher Zeit endet und einen Speicherzustand schafft, für den Q gilt."

Dieses Prädikat P hängt natürlich seinerseits von Q und S ab. Wir nennen P die *schwächste* (d.h. also die allgemeinste) *Vorbedingung* (englisch: "weakest precondition"), unter der S zu Q führt, und notieren

$$P = svb(S,Q).$$

Diese Notation ist durch die Beobachtung begründet, daß wir P (wie bei unserem ersten Ansatz Q) als das Ergebnis einer Transformation des Prädikats Q durch die Anweisung oder das Programmstück S auffassen können. Die auf der "Menge Π aller Prädikate" definierte Abbildung

$$\text{"svb(S, -): } \Pi \longrightarrow \Pi\text{"}$$

wird daher als "Prädikaten-Transformierer" bezeichnet. Es gilt:

$$(*\ svb(S,Q)\ *)\ S\ (*\ Q\ *).$$

Sei andererseits "(* P *) X (* Q *)" die Spezifikation eines Programmstücks X. Diese Spezifikation wird offenbar genau dann durch S erfüllt, wenn

$$EXT(P) \subset EXT(svb(S,Q)).$$

Wir schreiben dafür auch:

$$P \Rightarrow svb(S,Q).$$

Wir haben bei der bisherigen Erklärung den Standpunkt eingenommen, daß uns die Bedeutung (die "Semantik") der Elemente einer Programmiersprache und ihrer Verknüpfungen bereits "irgendwie" bekannt ist und wir lediglich das Problem haben, diese Bedeutung einem Lernenden zu vermitteln.

Wir können allerdings auch einen ganz anderen Standpunkt einnehmen, nämlich den des Erfinders oder Konstrukteurs einer Programmiersprache. Dann hätten wir die Aufgabe, die Bedeutung der einzelnen Elemente unserer Sprache und die ihrer Verknüpfungen in geeigneter Weise festzulegen (zu definieren!). (Ba-

sierend auf dieser Festlegung könnte dann ein Compiler entwickelt werden, der zu einem gegebenen Programm immer denjenigen Maschinencode erzeugt, der die mit dem Programm beabsichtigten Wirkungen hervorruft.)

Wir werden uns auf diesen Standpunkt stellen und annehmen, daß in unserer Programmiersprache - wie in MODULA-2 - ein Programmstück gebildet werden kann durch

- eine einzelne Zuweisung,
- das Aneinanderreihen von Programmstücken (Sequenz),
- eine Anweisung zur Selektion von Programmstücken gemäß irgendwelcher Bedingungen und durch
- eine Anweisung zur wiederholten Ausführung (Iteration) eines Programmstücks (unter Angabe einer Abbruchsbedingung).

Um nun allgemein die Semantik eines beliebigen Programmstücks festzulegen, können wir uns der oben eingeführten Prädikaten-Transformierer "svb(S,-)" bedienen. Was wir tun müssen, ist im Prinzip recht einfach:

Sei S eine Zuweisung, eine Folge von Programmstücken, eine Selektions-Anweisung oder eine Iterations-Anweisung. In jedem Fall haben wir zu definieren: svb(S,-). Natürlich werden wir zur Definition von svb(S,-) für Sequenzen, Selektions- und Iterations-Anweisung die Definitionen derjenigen Prädikaten-Transformierer heranziehen, die den jeweils beteiligten elementareren Programmstükken entsprechen. (Ist S zum Beispiel die Sequenz "S1;S2", so wird svb(S,-) durch svb(S1,-) und svb(S2,-) auszudrücken sein.)

Bevor wir in den folgenden Abschnitten diese Definitionen im einzelnen aufstellen und erläutern, wollen wir uns mit dem Begriff der "schwächsten Vorbedingung" anhand einiger einfacher Beispiele und Eigenschaften ein wenig vertrauter machen.

Beispiele:

(i) Sei S die Zuweisung "x:=x-1;" und sei Q = Q(x) das Prädikat "x>=0". In welchem Zustand muß sich der Speicher x befunden haben, damit nach Ausführung von S die Aussage Q zutrifft? S vermindert den Inhalt von x um 1. Damit dieser nach Ausführung von S mindestens 0 ist, muß er vorher mindestens 1 gewesen sein. Also gilt:

$$svb(x:=x-1,\ x\geq 0) = (x\geq 1).$$

(ii) Sei S die Selektions-Anweisung

"IF x<=y THEN z:=x ELSE z:=y END;"

Wir betrachten das Ergebnis der Abbildung svb(S,-) für vier verschiedene Prädikate Q:

a) Sei Q = Q(x,y,z) das Prädikat "z=min(x,y)".
(Es sollte klar sein, daß "z=min(x,y)" hier als Abkürzung für "(z=x AND x<=y) OR (z=y AND x>y)" steht!)

Intuitiv verstehen wir, daß S - unabhängig vom jeweils vorherigen Speicherzustand - immer einen Speicherzustand herstellt, für den Q gilt. Die Menge der (für die Erreichbarkeit von Q) vor S erlaubten Zustände ist also die Menge aller Speicherzustände. Diese wird (vgl. Abschnitt 4.1.1) durch das konstante Prädikat TRUE beschrieben. Es gilt daher in diesem Fall: svb(S,Q) = TRUE.

b) Sei Q = Q(x,z) das Prädikat "x=z". Dann gilt:

$$svb(S,Q) = (x<=y),$$

da genau dann, wenn vor der Ausführung von S ein Zustand mit (x<=y) angetroffen wird, z den Wert von x erhält. (Falls x>y, so erhält z den Wert von y, und es gilt dann "x <> z"!)

c) Sei Q = Q(y,z) das Prädikat "z=y+1".
Offenbar gibt es, da nach Ausführung von S immer z<=y ist, keinen Zustand, von dem aus S zu einem Zustand führt, in dem Q gilt.
Also: svb(S,Q) = FALSE.

d) Sei Q = Q(y,z) das Prädikat "z=y-1".
Nur wenn vor Ausführung von S x=y-1 zutraf, wird Q nach der Ausführung von S gelten, das heißt: svb(S,Q) = (x=y-1).

Die Präsentation der Beispiele (i) und (ii) müssen wir mit einem Appell an die Intuition des Lesers verbinden, da wir ja noch nicht über eine verbindliche Definition der Prädikaten-Transformierer

$$svb(x:=x-1, -) \text{ und } svb(IF\ x<=y\ THEN\ z:=x\ ELSE\ z:=y, -)$$

verfügen. Im Gegensatz dazu liefern wir mit den beiden folgenden Beispielen die Definitionen zweier einfacher Anweisungen.

(iii) Wir haben es bisher unterlassen, ein - wenn auch triviales, so doch wichtiges - Programmstück zu erwähnen, das "leere Programm", also ein Gebilde, welches überhaupt keine Anweisungen enthält. Notieren wir es als "NONE". Seine Semantik ist sinnvollerweise durch

$$svb(NONE,Q) = Q, \text{ für jedes Prädikat } Q,$$

zu definieren. Dies gibt unsere Vorstellung wieder, daß ein "leeres Programm" eben nichts bewirkt. Das heißt, welcher Zustand auch immer nach Ausführung von NONE herrschen soll: er muß auch vorher bereits bestanden haben. Der zugehörige Prädikaten-Transformierer ist also die identische Abbildung.

(iv) Wir könnten die Semantik einer Anweisung so definieren, daß diese unter gar keinen Umständen einen "vernünftigen" Speicherzustand herstellt. Nen-

nen wir diese Anweisung "ABORT". Gleichgültig, welche Nachbedingung Q zu erfüllen ist: für ABORT soll dies von keinem Zustand aus möglich sein. Die Semantik dieser Anweisung ist also (wegen EXT(FALSE) = { }) gegeben durch

svb(ABORT,Q) = FALSE, für jedes Prädikat Q.

ABORT sollte nur dann eingesetzt werden, wenn wir am weiteren Ablauf eines Programms nicht mehr interessiert sind: es dient zum Abbruch (englisch: "abortion") eines Programmlaufs. MODULA-2 bietet zu diesem Zweck die Anweisung "HALT".

Eigenschaften:

(v) Aus der für den Prädikaten-Transformierer svb(S,-) gegebenen (informellen) Definition folgt, daß svb(S,TRUE) für irgendein Programmstück S die Menge aller Speicherzustände beschreibt, von denen aus die Ausführung von S nach endlicher Zeit terminiert.

(vi) Entsprechend beschreibt - für ein beliebiges Programmstück S - das Prädikat svb(S,FALSE) die Menge aller Speicherzustände, von denen aus die Ausführung von S zwar terminiert, aber keinen Speicherzustand herstellt. Dies ist nicht möglich, da die Ausführung von S, wenn sie endet, immer zu irgendeinem Speicherzustand führt. Folglich ist die fragliche Menge leer, und es gilt svb(S,FALSE) = FALSE.

(vii) Seien Q und R Prädikate und S ein Programmstück. Dann gelten die folgenden Gesetze:

svb(S,Q) AND svb(S,R) = svb(S,Q AND R)

(AND-Distributivität),

svb(S,Q) OR svb(S,R) = svb(S,Q OR R)

(OR-Distributivität),

(Q => R) => (svb(S,Q) => svb(S,R))

(Monotonie).

Die einfachen Beweise dieser Gesetze seien dem Leser überlassen. Mit etwas Vorsicht ist die OR-Distributivität "zu genießen". Streng genommen gilt sie nur unter der zusätzlichen Voraussetzung, daß die Ausführung von S "deterministisch" ist. Dies heißt, daß der Zustand vor Ausführung von S den Zustand nach der Ausführung eindeutig bestimmt. Man kann sich leicht Anweisungen vorstellen, auf die diese Voraussetzung keineswegs zutrifft (vgl. [GRI]). Wir (und MODULA-2 ebensowenig) werden solche Anweisungen jedoch nicht berücksichtigen.

4.2.2 Zuweisungen und Sequenzen

Eine Zuweisung hat die allgemeine Form

‹vn›:=‹ausdruck›.

Dabei ist ‹vn› der Name einer Variablen, und ‹ausdruck› ist ein nach den Regeln der Syntax von MODULA-2 gebildeter Ausdruck. ‹vn› und ‹ausdruck› sind vom gleichen Typ. (Zuweisungen, bei denen dies nicht so ist, werden von einem MODULA 2 - Compiler im übrigen gar nicht erst akzeptiert.)

Mit Worten ist die (intuitiv wohl jedermann bekannte) Semantik der Zuweisung schnell beschrieben:

> "Der Wert von ‹ausdruck› wird ermittelt und in der Variablen mit dem Namen ‹vn› gespeichert."

Allerdings müssen wir den Fall berücksichtigen, daß ‹ausdruck› im aktuellen Speicherzustand nicht definiert und also auch nicht auswertbar ist. Die Zuweisung ist dann nicht ausführbar.

Die Auswertung von ‹ausdruck› kann mit dem Aufruf von Funktions-Prozeduren verbunden sein. Wir schließen aus, daß solche Aufrufe zu Seiteneffekten (vgl. Abschnitt 2.2.2) führen.

Zur Vereinfachung der Schreibarbeit verwenden wir als Variablennamen im folgenden immer x,y,z,... und als Bezeichner für einen Ausdruck Großbuchstaben A,B,... . Sei also "x:=A" irgendeine Zuweisung. Wir definieren:

Semantik der Zuweisung:

Sei Q ein beliebiges Prädikat:
svb(x:=A,Q) = def(A) CAND Q(x->A).

"def(A)" ist ein Prädikat, welches die Menge der Speicherzustände beschreibt, in denen A definiert ist.
(Ist A zum Beispiel der Ausdruck "x/y", so ist def(A) = (y<>0). Indem wir die CAND-Verknüpfung benutzen, liefert die Auswertung von svb(x:=x/y,Q) auch für einen Speicherzustand mit y=0 einen definierten Wert, nämlich F (vgl. die Tabelle in Abschnitt 4.1.2!).)

Man muß sich klarmachen, daß diese Definition mit unserer intuitiven Vorstellung von dem, was Zuweisung bedeutet, übereinstimmt: Um mit der Zuweisung "x:=A" in einen Speicherzustand Z zu gelangen, müssen wir uns in einem Zustand Z' befinden, in dem

(1.) A auswertbar ist und der sich

(2.) von Z dadurch unterscheidet, daß mit der Ersetzung des Inhalts der Variablen x durch den Wert von A der Zustand Z entsteht.

Gilt nun für Z das Prädikat Q, so muß für Z' das Prädikat Q' gelten, das aus Q durch die textuelle Ersetzung von x durch A hervorgeht, also Q' = Q(x->A). Anders gesagt: Die an der Auswertung von A beteiligten Variablen müssen in Z' schon "die richtigen" Werte haben! Oder: Genau dann entsteht aus einem Zustand Z' durch x:=A ein Zustand Z mit Q(...)(Z) = T, wenn Q'(...)(Z') = T. Das Beispiel im vorangegangenen Abschnitt und die nachfolgenden Beispiele mögen beim Leser die Überzeugung verstärken, daß unsere Definition gerechtfertigt ist.

Beispiele:

(In (i) und (ii) sei C ein Wertbezeichner (d.h. eine Konstante geeigneten Typs)).

(i) svb(x:=A, x=C) = (A=C)
(x:=A terminiert genau dann in einem Zustand mit x=C, wenn der Wert von A vor der Zuweisung C war!)

(ii) svb(x:=A, y=C) = (y=C)
(Der Leser mache sich klar, daß dies daran liegt, daß wir zur Bildung von Ausdrücken keine Funktions-Prozeduren mit Seiteneffekten zulassen! Täten wir es, so wäre unsere Definition der Semantik von Zuweisungen unbrauchbar.)

Für die beiden ersten Beispiele haben wir stillschweigend vorausgesetzt, daß def(A) = TRUE; in den folgenden Beispielen ist dies jeweils offensichtlich. Wir brauchen also in keinem Falle das Prädikat def(A) mitzuführen.

(iii) svb(x:=2*y+3, x=13) = (2*y+3=13) = (y=5)

(iv) svb(x:=x+y, x<2*y) = (x+y<2*y) = (x<y)

(v) svb(x:=x*y, x*y=c) = (x*y*y=c)

(vi) svb(x:=(x-y)*(x+y), x+y*y<>0) = ((x-y)*(x+y)+y*y<>0)
= (x*x-y*y+y*y<>0)
= (x*x<>0)
= (x<>0)

Sei b eine Variable vom Typ ARRAY OF INTEGER.

(vii) svb(j:=j+1, (0<j) AND (ALL i: 0<=i<=j: b[i]=5))
= (0<j+1) AND (ALL i: 0<=i<=j+1: b[i]=5)
= (0<=j) AND (ALL i: 0<=i<=j+1: b[i]=5)

Sei a5 eine Variable vom Typ BOOLEAN.

(viii) svb(a5:=(b[j]=5), a5=(ALL i: 0<=i<=j: b[i]=5))
= ((b[j]=5)=(ALL i: 0<=i<=j: b[i]=5))
= ALL i: 0<=i<=j-1: b[i]=5

Wenden wir uns nun der Aneinanderreihung von Programmstücken, der Bildung von Sequenzen zu. Seien S1 und S2 Programmstücke und sei S = S1;S2. Das Semikolon wird in diesem Zusammenhang als Konkatenations-Operator für Zei-

chenketten aufgefaßt und nicht als Begrenzungs-Zeichen, wie in vielen Programmiersprachen. ("Konkatenation" bedeutet Aneinanderreihung.) S auszuführen heißt, zuerst das Programmstück S1 und dann das Programmstück S2 auszuführen. Dies legt die folgende Definition nahe:

Semantik der Sequenzbildung:

Sei Q ein beliebiges Prädikat:
svb(S,Q) = svb(S1;S2,Q) = svb(S1,svb(S2,Q))

Nach dieser Definition terminiert S also in einem Speicherzustand mit Q genau dann, wenn S1 in einem Speicherzustand terminiert, für den die schwächste Vorbedingung von Q bezüglich S2 gilt. Und eben dies war wohl unsere Absicht! Ein (leichtes) Problem gibt es freilich noch, nämlich: Wie verhält es sich mit der Aneinanderreihung von mehr als zwei Programmstücken? Betrachten wir zum Beispiel S = S1;S2;S3. Es sollte keinen Unterschied machen, ob wir zuerst S1 und dann S2;S3 ausführen oder zuerst S1;S2 und dann S3, und dies sollte auch durch unsere Semantik-Definition zum Ausdruck kommen. In der Tat kann man zeigen, daß svb(-,-) im ersten Argument bezüglich der Aneinanderreihung von Programmstücken assoziativ ist, daß also gilt:

$$svb(S1;(S2;S3),Q) = svb((S1;S2);S3,Q),$$

für ein beliebiges Prädikat Q. (Der Leser möge sich am Beweis dieser Tatsache selbst versuchen.)

4.2.3 Selektionen

Wir betrachten zunächst die CASE-Anweisung und fragen uns, welche ihrer Eigenschaften wir in einer formalen Semantik-Definition zum Ausdruck kommen lassen wollen.

In der Anweisung (auf die wir im folgenden mit "S–CASE" Bezug nehmen werden)

```
CASE A OF
  C1:  S1 |
  C2:  S2 |
  ...
  Cn:  Sn
END (*CASE*);
```

sei A wieder irgendein Ausdruck; C1,...,Cn seien Bezeichner für Werte, die vom gleichen Typ sind wie der Ausdruck A, und die S1, ..., Sn seien beliebige Programmstücke. Wir setzen ferner voraus, daß die durch die Ci bezeichneten Werte paarweise verschieden sind.

Informell lautet die Semantik der Anweisung S–CASE:

> "Werte A aus; falls der gefundene Wert Ci ist, so führe das Programmstück Si aus."

Wiederum kann es freilich passieren, daß A im aktuellen Speicherzustand (da nicht definiert) gar nicht auswertbar ist. Außerdem ist es möglich, daß die Auswertung von A ein Ergebnis liefert, welches unter den C1,...,Cn nicht auftaucht.

Unabhängig von dem durch S–CASE herbeizuführenden Speicherzustand muß also das Prädikat

$$\text{def(A) AND (A=C1 OR A=C2 OR ... A=Cn)}$$

Teil jeder Vorbedingung sein.

Nehmen wir nun an, daß S–CASE zu einem durch ein Prädikat Q charakterisierten Speicherzustand führen soll. Da sich die Ausführung von S–CASE auf die Ausführung irgendeines der Si (i=1,...,n) "reduziert" und da diese gewissermaßen alle "gleichberechtigt" sind, muß Q nach der Ausführung jedes der Si gelten. Die Bedingung, unter der Si ausgeführt wird, lautet "A=Ci". Die durch svb(Si,Q) beschriebene Menge von Speicherzuständen muß also all jene Zustände enthalten, für die A=Ci gilt. Dies legt die folgende Definition nahe:

Semantik der CASE-Anweisung:

Sei Q ein beliebiges Prädikat:

```
svb(S–CASE,Q) = def(A) AND (A=C1 OR ... OR A=Cn)
                AND ((A=C1) => svb(S1,Q))
                AND ((A=C2) => svb(S2,Q))
                ...
                AND ((A=Cn) => svb(Sn,Q))
```

Unter der Voraussetzung, daß A überall definiert ist, kann man "def(A)" natürlich weglassen. Erinnern wir uns außerdem der Notationen für Existenz- und All-Aussagen, so ist

```
svb(S–CASE,Q) = (EX i: 1<=i<=n: A=Ci) AND
                (ALL i: 1<=i<=n: (A=Ci) => svb(Si,Q))
```

eine etwas knappere Formulierung der Semantik der CASE-Anweisung. Ersetzen wir noch die Implikation "(A=Ci) => svb(Si,Q)" durch den äquivalenten Ausdruck "NOT(A=Ci) OR svb(Si,Q)", so ergibt sich:

```
svb(S–CASE,Q) = (EX i: 1<=i<=n: A=Ci) AND
                (ALL i: 1<=i<=n: (A<>Ci) OR svb(Si,Q)).
```

IF-Anweisungen können wir als Spezialfälle von CASE-Anweisungen ansehen: In

```
IF B THEN
  S1
ELSE
  S2
END (*IF*);
```

sei B ein Ausdruck vom Typ BOOLEAN, S1 und S2 seien beliebige Programmstücke. Nennen wir diese Anweisung "S–IF". Wir definieren ihre Semantik durch die Semantik der folgenden CASE-Anweisung:

```
CASE B OF
  TRUE:  S1 |
  FALSE: S2
END (*CASE*); :
```

Semantik der IF-Anweisung:

Sei Q ein beliebiges Prädikat:

svb(S–IF,Q) = def(B) AND ((B AND svb(S1,Q)) OR
(NOT(B) AND svb(S2,Q))

(Der Leser möge die Herleitung von svb(S–IF,Q) aus der angegebenen Spezialisierung von svb(S–CASE,Q) selbständig nachvollziehen. In den meisten Fällen wird wieder def(B) = TRUE gelten.)

Überprüfen wir nun anhand der Beispiele (ii) a)-d) in Abschnitt 4.2.1, ob diese Definition unser intuitives Verständnis der Bedeutung der IF-Anweisung widerspiegelt.

Sei also S die Anweisung

"IF x<=y THEN z:=x ELSE z:=y END;"

a) Sei Q = Q(x,y,z) das Prädikat "(z=x AND x<=y) OR (z=y AND x>y)".

```
svb(S,Q) = (x<=y AND svb(z:=x,Q)) OR
           (x>y  AND svb(z:=y,Q))
         = (x<=y AND ((x=x AND x<=y) OR (x=y AND x>y)))
           OR
           (x>y  AND ((y=x AND x<=y) OR (y=y AND x>y)))
         = (x<=y AND x=x AND x<=y) OR
           (x>y  AND y=y AND x>y)
         = (x<=y) OR (x>y) = TRUE
```

b) Sei Q = Q(x,z) das Prädikat "x=z".

```
svb(S,Q) = (x<=y AND svb(z:=x, x=z)) OR
           (x>y  AND svb(z:=y, x=z))
```

= (x<=y AND x=x) OR
(x>y AND x=y)
= (x<=y) OR FALSE = (x<=y)

c) Sei Q = Q(y,z) das Prädikat "z=y+1".

svb(S,Q) = (x<=y AND svb(z:=x, z=y+1)) OR
(x>y AND svb(z:=y, z=y+1))
= (x<=y AND x=y+1) OR
(x>y AND y=y+1)
= FALSE OR FALSE = FALSE

d) Sei Q = Q(y,z) das Prädikat "z=y-1".

svb(S,Q) = (x<=y AND svb(z:=x, z=y-1)) OR
(x>y AND svb(z:=y, z=y-1))
= (x<=y AND x=y-1) OR
(x>y AND y=y-1)
= (x=y-1) OR FALSE = (x=y-1)

Die in 4.2.1 (intuitiv) begründeten Ergebnisse werden - wie man sieht - also auch geliefert, wenn wir die formale Definition der Semantik von S–IF einsetzen.

Zur Übung definiere der Leser:

- die Semantik einer IF-Anweisung ohne "ELSE-Zweig" als Semantik der CASE-Anweisung

```
CASE B OF
  TRUE:  S1 |
  FALSE: (*NONE*)
END (*CASE*);
```

- die Semantik der mit ELSIF geschachtelten IF-Anweisung

```
IF B1 THEN
  S1
ELSIF B2 THEN
  S2
ELSE
  S3
END (*IF*);
```

als Semantik der Anweisung

```
IF B1 THEN
  S1
ELSE
  IF B2 THEN
    S2
  ELSE
```

```
        S3
      END (*IF*)
    END (*IF*);
```

Ferner möge sich der Leser fragen, wie die Definition der Semantik von S–CASE sinnvollerweise abzuändern wäre, wenn die CASE-Anweisung auch den "ELSE-Zweig" enthält.

Wir erwähnten bereits in Abschnitt 4.2.1, daß die Erfüllung der Spezifikation "(* P *) X (* Q *)" darauf hinausläuft, ein Programmstück S zu finden mit "P => svb(S,Q)".

Die Frage ist, ob man in jedem Fall svb(S,Q) "ausrechnen" muß, um diese Implikation nachzuweisen. Für S–CASE zeigen wir zum Abschluß dieses Abschnitts, daß dies durchaus nicht notwendig ist. Um zu argumentieren, daß P => svb(S,Q) gilt, genügt es, die folgenden Behauptungen zu verifizieren:

(I) P => (EX i: 1<=i<=n: A=Ci) und
(II) P AND (A=Ci) => svb(Si,Q) für alle i=1,...,n.

Angenommen, es gelte (II), d.h.:

```
   (ALL i: 1<=i<=n: P AND (A=Ci) => svb(Si,Q))
 = (ALL i: 1<=i<=n: NOT(P AND A=Ci) OR svb(Si,Q))
 = (ALL i: 1<=i<=n: NOT(P) OR (A<>Ci) OR svb(Si,Q))
 = NOT(P) OR (ALL i: 1<=i<=n: (A<>Ci) OR svb(Si,Q))
 = (P => (ALL i: 1<=i<=n: (A<>Ci) OR svb(Si,Q))
```

Mit (I) zusammen ergibt sich also:

```
   (P => (EX i: 1<=i<=n: A=Ci)) AND
   (P => (ALL i: 1<=i<=n: (A<>Ci) OR svb(Si,Q)))
 = (P => ((EX i: 1<=i<=n: A=Ci) AND
          (ALL i: 1<=i<=n: (A<>Ci) OR svb(Si,Q))))
 = (P => svb(S–CASE,Q)),
```

das heißt, unter den Voraussetzungen (I) und (II) impliziert P die schwächste Vorbedingung von Q unter S–CASE.

4.2.4 Iterationen

Es sei S–WHILE die Anweisung

```
    WHILE B DO
      S
    END (*WHILE*);
```

mit einem booleschen Ausdruck B und einem beliebigen Programmstück S.

Jeder Programmierer weiß, daß solche Anweisungen "es in sich haben": Wird nämlich durch S nicht irgendwann ein Speicherzustand geschaffen, in dem B

den Wert FALSE hat, so endet ein Programm, welches S–WHILE enthält, niemals. Bei der Konstruktion solcher Anweisungen ist also besondere Sorgfalt angebracht.

Unserem intuitiven Verständnis gemäß sollte S–WHILE ausgeführt werden wie die "unendlich geschachtelte" IF-Anweisung

```
IF B THEN
  S;
```

```
IF B THEN
  S;
  IF B THEN
    ...
  ELSE
    (*NONE*)
  END (*IF*)
ELSE
  (*NONE*)
END (*IF*)
```

```
ELSE
  (*NONE*)
END (*IF*);
```

Der eingerahmte Teil hat die gleiche Gestalt wie das ganze Programmstück. Es scheint also zweckmäßig, die Semantik der WHILE-Anweisung als Semantik der Anweisung

```
IF B THEN
  S;
  WHILE B DO
    S
  END (*WHILE*)
ELSE
  (*NONE*)
END (*IF*);
```

zu definieren. Wir machen daher den folgenden ersten Versuch:

Semantik der WHILE-Anweisung (Version 1):

Sei Q ein beliebiges Prädikat:

svb(S–WHILE,Q) = def(B) AND
(NOT(B) AND Q) OR
(B AND svb(S; S–WHILE,Q))

Wir nehmen im folgenden an, daß B überall definiert ist, und verzichten darauf, def(B) weiterhin "mitzuschleppen". Unter Ausnutzung der Semantik der Sequenz

von Programmstücken ergibt sich:

svb(S–WHILE,Q) = (NOT(B) AND Q) OR
(B AND svb(S,svb(S–WHILE,Q))).

Wir interpretieren die so gefundene Darstellung wie folgt:

(NOT(B) AND Q) beschreibt die Menge aller Speicherzustände, von denen aus S–WHILE ohne eine einzige Ausführung von S (d.h. "in 0 Durchläufen"!) einen Zustand herstellt, in dem Q gilt.

svb(S,svb(S–WHILE,Q)) beschreibt die Menge aller Speicherzustände, von denen aus S die Vorbedingung dafür garantiert, daß S–WHILE einen Zustand herstellt, in dem Q gilt. Das zweite Glied der Disjunktion ist also insgesamt die Bedingung dafür, daß S–WHILE nach mindestens einmaliger Ausführung von S einen Zustand herstellt, in dem Q gilt.

Natürlich ist es ein wenig unbefriedigend, daß wir hier (im Gegensatz zu den vorangegangenen "einfachen" Definitionen) die Semantik von S–WHILE gewissermaßen "durch sich selbst" (also rekursiv) definiert haben. Wir wollen daher diese Rekursion noch einen Schritt weiter treiben und sehen, ob wir auf diese Weise eine andere Darstellung von svb(S–WHILE,Q) erreichen.

svb(S–WHILE,Q) = (NOT(B) AND Q) OR
(B AND svb(S,svb(S–WHILE,Q)))
= (NOT(B) AND Q) OR
(B AND svb(S, (NOT(B) AND Q) OR
(B AND svb(S,svb(S–WHILE,Q))))
= (NOT(B) AND Q) OR
(B AND svb(S, NOT(B) AND Q)) OR
(B AND svb(S, B AND svb(S,svb(S–WHILE,Q)))).

Das zweite Glied dieser Disjunktion beschreibt nun die Menge aller Speicherzustände, von denen aus S–WHILE nach genau einer Ausführung von S endet und einen Zustand mit Q liefert. Die Interpretation des dritten Gliedes der Disjunktion liegt auf der Hand.

Setzen wir zur Abkürzung

G_0(S,B,Q) = NOT(B) AND Q

und definieren (für i>0)

G_i(S,B,Q) = B AND svb(S,G_(i-1)(S,B,Q)),

so können wir mittels Induktion leicht zeigen, daß gilt:

svb(S–WHILE,Q) = (EX i: 0<=i<n: G_i(S,B,Q)) OR M_n(S,B,Q).

Dabei ist M_n(S,B,Q) ein Prädikat, welches besagt, daß mindestens n Ausführungen von S erforderlich sind, um zu Q zu gelangen. Mit wachsendem n "strebt" M_n(S,B,Q) also gegen ein Prädikat M_U(S,B), welches diejenigen Zustände be-

schreibt, von denen aus die Ausführung von S–WHILE niemals endet. (Es hängt natürlich nicht mehr von Q ab!) Da svb(PS,Q) für ein beliebiges Programmstück PS aber insbesondere die Menge der Zustände beschreiben soll, von denen aus PS überhaupt terminiert, darf M_U(S,B) nicht Teil der schwächsten Vorbedingung unter S–WHILE sein. Im zweiten Anlauf definieren wir die Semantik der WHILE-Anweisung daher (endgültig) wie folgt:

Semantik der WHILE-Anweisung:

Sei Q ein beliebiges Prädikat und seien die G_i definiert wie oben:
svb(S–WHILE,Q) = (EX i: 0<=i: G_i(S,B,Q))

G_i(S,B,Q) charakterisiert die Zustände, von denen aus S–WHILE nach genau i Ausführungen von S in einem Zustand endet, in dem Q gilt.

Mit H_j(S,B,Q) = (EX i: 0<=i<=j: G_i(S,B,Q)) können wir svb(S–WHILE,Q) auch so formulieren:

svb(S–WHILE,Q) = (EX j: 0<=j: H_j(S,B,Q)).

H_j(S,B,Q) charakterisiert die Zustände, von denen aus S–WHILE nach höchstens j Ausführungen von S in einem Zustand endet, in dem Q gilt.

Wie schon erwähnt, ist es bei der Konstruktion von Iterations-Anweisungen wichtig sicherzustellen, daß die Wiederholung irgendwann endet. Angenommen, wir hätten die Spezifikation (* P *) X (* Q *) zu erfüllen, und wir vermuteten, daß dies mit einer Anweisung der Art S–WHILE möglich ist. Es wäre dann zu zeigen, daß P => svb(S–WHILE,Q) gilt. Mit anderen Worten: Es wäre zu einem beliebigen Speicherzustand Z mit P(Z) = T ein j = j(Z) zu finden derart, daß H_j(S,Q)(Z) = T. Dies scheint auf den ersten Blick keine leichte Aufgabe zu sein. Wir werden aber sogleich ein einfaches Verfahren angeben, mit dessen Hilfe beweisbar korrekte Iterations-Anweisungen gebildet werden können.

Dazu betrachten wir zunächst die Prädikate G_i(S,B,Q) und nehmen an, daß sich diese - nach Einführung der zusätzlichen Variablen i - durch ein einziges Prädikat G(i,S,B,Q) ausdrücken lassen. Gilt für einen Speicherzustand Z (bei dem nun natürlich auch die Variable i zu berücksichtigen ist) das Prädikat G(i,S,B,Q), so bedeutet dies, daß S–WHILE nach ebenso vielen Ausführungen von S in einem Zustand mit Q endet, als durch den aktuellen Wert der Variablen i angezeigt wird. Für das Prädikat G(i,...) können wir zwei interessante Eigenschaften feststellen:

(i) G(i,...) = NOT(B) AND Q, falls i=0;

(ii) falls i>0 und G(i,...) vor einer Ausführung von S gelten, so gilt G(i,...) auch danach, mit um 1 vermindertem Inhalt der Variablen i.

Eigenschaft (i) ist nur eine Umformulierung der Definition von G_0. Eigenschaft (ii) ergibt sich unmittelbar aus unserer Interpretation des Prädikates G(i,...). Sie besagt zum einen, daß die Gültigkeit dieses Prädikats "invariant" ist bezüglich der Ausführung von S und daß insbesondere gilt:

B AND G(i,...) => svb(S, G(i,...)).

Zum zweiten besagt sie, daß durch jede Ausführung von S der Wert der von uns postulierten Variablen i vermindert wird und man daher sicher sein kann, daß irgendwann nach S das Prädikat G(0,...) = NOT(B) AND Q gilt. Die Schleife bricht dann ab, und die gewünschte Nachbedingung ist erfüllt.

Diese Eigenschaften des Prädikats G(i,...) einschließlich der hier beobachteten Wirkung von S auf den Inhalt von i können leicht verallgemeinert werden. Wir definieren (immer mit Bezug auf die Anweisung S–WHILE):

Schleifen-Invariante:

Ein Prädikat InV heißt Schleifen-Invariante, falls
B AND InV => svb(S,InV).

Schranken-Funktion:

Sei InV eine Invariante. Eine Funktion Sf = Sf(...) vom Typ INTEGER der (Inhalte der) Variablen von S–WHILE heißt Schranken-Funktion bezüglich InV, falls

B AND InV => (Sf>0) AND svb(s:=Sf(...);S, Sf<s).

Wir behaupten:

Um mit S–WHILE die Nachbedingung Q zu erfüllen, genügt es, eine Invariante InV zu finden mit
(I) InV gilt vor S–WHILE und
(II) NOT(B) AND InV => Q,
sowie eine Schranken-Funktion Sf bezüglich InV.

Zur Begründung zeigen wir informell, daß

InV => svb(S–WHILE,Q).

Zunächst ist (gemäß Wahl von InV):

NOT(B) AND InV => NOT(B) AND Q = H_0(...) => svb(S–WHILE,Q).

Da der Wert von Sf durch die Ausführung von S vermindert wird, bedeutet

B AND InV AND (Sf<=k)

(für beliebiges k>0), daß nach höchstens k Ausführungen von S ("Schleifendurchläufen") gilt: (Sf<=0). Daraus folgt (vgl. Definition der Schranken-Funktion), daß nach höchstens k Durchläufen NOT(B) AND InV gilt. Das heißt

B AND InV AND (Sf<=k) => H_k(...) => svb(S–WHILE,Q).

Da sich andererseits InV durch die OR-Verknüpfung der soeben betrachteten Prädikate darstellen läßt, folgt unsere Behauptung. (Man beachte, daß wir mit der Einführung einer Schranken-Funktion einen eleganten Weg gefunden haben, die Terminierung einer WHILE-Schleife sicherzustellen.)

Bevor wir uns im nächsten Abschnitt eingehender mit der Konstruktion korrekter und terminierender "Schleifen" befassen, sollen die bisherigen Erklärungen durch einige einfache Beispiele illustriert werden. Die Invarianten und Schranken-Funktionen werden wir dabei gewissermaßen "hervorzaubern". Erst im nächsten Abschnitt werden wir systematischere Ansätze zu ihrer Gewinnung behandeln.

Beispiele:

(i) Seien p,x und n Variable vom Typ INTEGER. Wir wollen die folgende Nachbedingung erfüllen:

Q(p,x,n) = (p = exp(x,n)).

(exp(x,n) sei definiert durch: exp(x,0)=1, exp(x,i+1) = x*exp(x,i))

Die Aufgabe ist durch eine WHILE-Anweisung zu lösen. Zur Formulierung einer Invariante führen wir die zusätzliche Variable i ein und definieren:

InV = (i<=n) AND (p = exp(x,i)).

Die Bedingung B der WHILE-Anweisung ergibt sich aus der Forderung NOT(B) AND InV => Q. Offenbar gilt in unserem Fall: ((i=n) AND InV) = Q. Also kommt B = (i<>n) als Abbruchbedingung in Frage.

Wir haben dafür zu sorgen, daß vor der ersten Ausführung von S die Invariante InV gilt. Unter anderem wird dies von dem folgenden - mit Prädikaten als Kommentare versehenen - Programmstück geleistet:

```
i:=0;        |<---- "Schleifeninitialisierung"
p:=1;        |
(* InV *)
WHILE i<>n DO
  p:=p*x;    |<---- "Schleifenrumpf" S
  i:=i+1;    |
  (* InV *)
END (*WHILE*);
(* (i=n) AND InV *)
(* p = exp(x,n) *)
```

InV gilt offenbar nach der "Schleifeninitialisierung". Natürlich müssen wir zeigen, daß InV auch tatsächlich eine Invariante ist, daß also die Implikation

$$(i<>n) \text{ AND } InV \Rightarrow svb(S,InV)$$

gilt. Dazu "rechnen" wir svb(S,InV) "aus":

svb(S,InV) = svb(p:=p*x; i:=i+1, (i<=n) AND (p=exp(x,i)))
= svb(p:=p*x, svb(i:=i+1, (i<=n) AND (p=exp(x,i))))
= svb(p:=p*x, (i+1<=n) AND (p=exp(x,i+1)))
= (i+1<=n) AND (p*x=exp(x,i+1))
= (i<n) AND (p=exp(x,i)) (nach Definition von exp(.,.))

Andererseits ist

(i<>n) AND InV = (i<>n) AND (i<=n) AND (p=exp(x,i))
= (i<n) AND (p=exp(x,i)).

In unserem Falle sind also (i<>n) AND InV und svb(S,InV) sogar äquivalent.

Zur vollständigen Erledigung der Aufgabe ist noch eine Schrankenfunktion Sf anzugeben, welche garantiert, daß die Schleife terminiert. Wir wählen:

$$Sf = Sf(i,n) = n-i$$

und überprüfen die entsprechenden Eigenschaften:

(i<>n) AND (i<=n) AND (p=exp(x,i)) = (i<n) AND (...) => (n-i>0).
Falls B zutrifft, ist der Wert von Sf also größer als 0.

Zweitens ist zu zeigen, daß durch Ausführung von S der Wert von Sf vermindert wird:

svb(s:=n-i;S, n-i<s) = svb(s:=n-i, svb(S, n-i<s))
= svb(s:=n-i, n-(i+1)<s)
= (n-(i+1)<n-i) = (1>0) = TRUE,

also gilt auch:

$$B \text{ AND } InV \Rightarrow svb(s:=Sf(..);S, Sf(..)<s).$$

Es empfiehlt sich, neben der Invariante auch die Schrankenfunktion durch einen Kommentar im Programmtext explizit anzugeben.

(ii) Im zweiten Beispiel wollen wir den größten der in einem Array gespeicherten Werte herausfinden. Seien a eine Variable vom Typ ARRAY [0..9] OF INTEGER und s eine Variable vom Typ INTEGER. Die herzustellende Nachbedingung lautet somit:

$$Q(s,a) = (s = \max(a[0],...,a[9])).$$

(Auf eine Formalisierung der Definition von max(...) sei hier verzichtet.)

Für die WHILE-Anweisung schlagen wir die folgende Invariante vor:

$$InV = (i<=10) \text{ AND } (s = \max(a[0],...,a[i-1])).$$

Wie oben begründet, ergibt sich als mögliche Abbruchsbedingung B: (i<>10).

Von dem folgenden Programmstück behaupten wir, daß es zum Ziel führt:

```
s:=a[0];      |<---- "Schleifeninitialisierung"
i:=1;         |
(* InV *)
(* Sf = Sf(i) = 10-i *)
WHILE i<>10 DO
  IF a[i]> s THEN   ┐          ┐
    s:=a[i]         |          |
  ELSE              |<- S-IF   |<- S
    (*NONE*)        |          |
  END (*IF*);       ┘          |
  i:=i+1                       ┘
  (* InV *)
END (*WHILE*)
(* (i=10) AND InV *)
(* s = max(a[0],...,a[9])) *).
```

(Wie man sieht, haben wir die zum Ende des ersten Beispiels gegebene Empfehlung bereits beherzigt und eine geeignete Schrankenfunktion angeschrieben.)

Wir zeigen:

(1) InV gilt nach der "Schleifeninitialisierung":

((1<=10) AND (s=a[0])) = ((1<10) AND (s=max(a[0]))).

(2) InV ist tatsächlich eine Invariante, d. h.: (i<>10) AND InV => svb(S,InV).

svb(S,InV) = svb(S–IF;i:=i+1, InV)
= svb(S–IF, svb(i:=i+1, InV)
= svb(S–IF, (i+1<=10) AND (s=max(a[0],...,a[i])))
= (i+1<=10) AND svb(S–IF, (s=max(a[0],...,a[i])))
= (i+1<=10) AND ((a[i]> s) AND (a[i]=max(a[0],...,a[i])) OR (a[i]<=s) AND (s=max(a[0],...,a[i])))
= (i<10) AND ((a[i]> s) AND (a[i]=max(a[0],...,a[i])) OR (s=max(a[0],...,a[i-1])))
= (i<10) AND (a[i]>s) AND (a[i]=max(a[0],...,a[i])) OR (i<10) AND (s=max(a[0],...,a[i-1])).

Andererseits ist

(i<>10) AND InV = (i<10) AND (z=max(a[0],...,a[i-1])),

und dies impliziert offensichtlich svb(S,InV).

(3) Sf = Sf(i) = 10 - i ist tatsächlich eine Schrankenfunktion:

(a) (i<>10) AND InV => (i<10) => (10-i>0).

(b) (i<>10) AND InV => svb(s:=10-i;S, 10-i<s):

svb(s:=10-i;S, 10-i<s) = svb(s:=10-i;S–IF, svb(i:=i+1, 10-i<s))
= svb(s:=10-i;S–IF, 10-(i+1)<s)
= svb(s:=10-i, 10-(i+1)<s)
= (10-(i+1)<10-i) = (1>0) = TRUE,

das heißt, B AND InV => svb(s:=Sf;S, Sf<s), wie zu zeigen war.

(iii) Zum Schluß dieses Abschnitts wollen wir das entwickelte Instrumentarium - und insbesondere die im vorigen Beispiel aufgestellte "Check-Liste" zur Überprüfung von WHILE-Anweisungen - einsetzen, um die Korrektheit einer verbesserten Version des kleinen Divisions-Programms aus Abschnitt 4.1.3 nachzuweisen. Wir zitieren es (berichtigt!), ergänzt um die Angabe einer Schrankenfunktion:

```
PROCEDURE Division((*IN*) x,y:INTEGER; (*OUT*) VAR q,r:INTEGER);
BEGIN
 (* x>=0 AND y>0 *)
 r:=x;                              |<--- "Schleifeninitialisierung"
 q:=0;
 (* InV: r>=0 AND x=q*y+r *)
 (* Sf = Sf(r) = r *)
 WHILE r>=y DO
  r:=r-y;                           |<--- "Schleifenrumpf" S
  q:=q+1                            |
  (* InV *)
 END (*WHILE*)
 (* (r<y) AND InV *)
 (* Q: r>=0 AND y>r AND x=q*y+r *)
END Division;
```

(1) InV gilt nach der Schleifeninitialisierung:

Wegen x>=0 ist nach der Schleifeninitialisierung

InV = (r>=0 AND x=q*y+r)
= (x>=0 AND AND x=0*y+x) = TRUE.

(2) InV ist tatsächlich eine Invariante, d. h.: (r>=y) AND InV => svb(S,InV).

svb(S,InV) = svb(r:=r-y, svb(q:=q+1,InV))
= svb(r:=r-y, (r>=0 AND x=(q+1)*y+r))
= (r-y>=0 AND x=(q+1)*y+(r-y))
= (r-y>=0 AND x=q*y+r).

Andererseits:

(r>=y) AND InV = (r>=y) AND (r>=0 AND x=q*y+r)
= (r-y>=0 AND x=q*y+r)

= svb(S,InV).

(3) NOT(B) AND InV => Q:

NOT(B) AND InV = (r<y) AND (r>=0 AND x=q*y+r)
= (r>=0 AND y>r AND x=q*y+r)
= Q

(Es gilt also sogar Äquivalenz!)

(4) Sf = Sf(r) = r ist tatsächlich eine Schrankenfunktion:

(a) (r>=y) AND InV => (r>=y) AND (...) => (r>0) (wegen y>0 !).

(b) (r>=y) AND InV => svb(s:=r;S, r<s):

svb(s:=r;S, r<s) = svb(s:=r, svb(r:=r-y;q:=q+1, r<s))
= svb(s:=r, r-y<s)
= (r-y<r) = (y>0) = TRUE,

das heißt, B AND InV => svb(s:=Sf;S, Sf<s), wie zu zeigen war.

Mit den Punkten (1) - (4) haben wir im letzten Beispiel eine vollständige "Check-Liste" präsentiert, mit deren Hilfe die Korrektheit einer "fertigen" WHILE-Anweisung bewiesen werden kann. Nun ist uns aber, wie schon in der Einleitung zu diesem Kapitel bemerkt, am Beweis der Korrektheit fertiger Programme wenig gelegen. Vielmehr ist es unsere Absicht, korrekte Programme - auf der Grundlage gegebener Nachbedingungen - zu konstruieren. Für die Konstruktion von WHILE-Anweisungen (die zu den heikelsten Programm-Abschnitten zählen!) bedeutet dies insbesondere, daß wir konkrete Hinweise zur Auffindung von Invarianten, Schleifenbedingungen und Schrankenfunktionen benötigen. Solche Hinweise soll der folgende Abschnitt geben. Daß die gefundenen Prädikate und Funktionen aber auch tatsächlich die geforderten Bedingungen erfüllen, muß natürlich bewiesen (oder zumindest informell plausibel gemacht) werden. Der Grad der dabei jeweils angebrachten Formalisierung liegt im Ermessen (im "Fingerspitzengefühl") des Programmierers. In jedem Fall wird er sich an der "Check-Liste" orientieren können.

4.3 Zielorientierte Programmentwicklung

In diesem Abschnitt geht es um die Frage, wie sich die in 4.2 entwickelten Semantik-Definitionen zur Auffindung korrekter Anweisungen innerhalb eines Programms anwenden lassen. Korrekt ist eine Anweisung offenbar dann, wenn sie die gewünschte Nachbedingung schafft. Die Nachbedingung ist also das "Ziel", welches erreicht werden muß. Es ist daher unmittelbar einsichtig, daß wir bei der Suche nach einer korrekten Anweisung bei diesem Ziel anzusetzen haben. Der Programmierer verhält sich dabei im Grunde ganz ähnlich wie ein Mathematiker, der einen Lehrsatz "konstruktiv" beweisen möchte: Der Satz bestehe aus Voraussetzung und Behauptung; diese entsprechen dem, was wir "Vorbedingung"

und "Nachbedingung" genannt haben. Eine mögliche Strategie des Mathematikers ist es, die Behauptung genau zu analysieren, um Operationen (beispielsweise mit Zirkel und Lineal) mit den dort vorkommenden Objekten (beispielsweise Punkte, Geraden und Kreise) zu finden, welche die Behauptung in bestimmten Fällen wahr machen. Er muß dann zeigen, daß die Gesamtheit dieser Fälle seine Voraussetzung "überdeckt".

Es dürfte aus den bisherigen Ausführungen dieses Kapitels klar geworden sein, daß unser besonderes Augenmerk im vorliegenden Abschnitt den Iterations-Anweisungen zu gelten hat. Dennoch werden wir mit einigen Bemerkungen über Selektions-Anweisungen beginnen, um die Analogie zur geschilderten (möglichen) Vorgehensweise des Mathematikers zu verdeutlichen. Was nun die Iterations-Anweisungen betrifft, so werden wir uns zunächst darauf beschränken, deren Konstruktion auf irgendwie "hervorgezauberte" Invarianten und Schrankenfunktionen (vgl. die Beispiele in 4.2.4) zu begründen. Erst dann wollen wir uns darum kümmern, wie man Invarianten aus einer gegebenen Nachbedingung gewinnen kann und welche Anhaltspunkte es für das Erkennen von Schrankenfunktionen gibt. Für beides gibt es leider keine schematischen Verfahren. Was hier zählt, sind Intuition, Geschick und Erfahrung.

4.3.1 Konstruktion von Selektions-Anweisungen

Wir greifen ein Beispiel aus Abschnitt 4.2.1 auf: die Bestimmung des Minimums zweier ganzer Zahlen. Es ist also - unter allen Umständen - ein Zustand herzustellen, in dem gilt: z = min(x,y). Die Inhalte der Variablen x und y sind dabei selbstverständlich nicht zu verändern (vgl. Abschnitt 4.1.3). Die ausführliche Formulierung dieser Nachbedingung lautet (vgl. Abschnitt 4.2.3):

Q: (z=x AND x<=y) OR (z=y AND x>y).

Gibt es eine Operation, die einen durch dieses Prädikat beschriebenen Zustand zumindest unter gewissen Bedingungen herstellt? Da unter anderem auch "z=x" gelten darf, liegt es nahe, es mit der Zuweisung "z:=x" zu versuchen. Um herauszufinden, unter welchen Bedingungen diese Zuweisung zum Erfolg führt, berechnen wir svb(z:=x,Q):

svb(z:=x,Q) = svb(z:=x, (z=x AND x<=y) OR (z=y AND x>y))
= (x=x AND x<=y) OR (x=y AND x>y)
= (x<=y) OR FALSE
= (x<=y).

Unsere Vorbedingung lautet aber "TRUE", und diese wird durch "x<=y" offensichtlich nicht "überdeckt". Doch haben wir ja - aus dem gleichen Grund, der uns veranlasste, die Zuweisung "z:=x" zu untersuchen - noch "z:=y" in petto:

svb(z:=y,Q) = ...
= (x=y) OR (x>y)

= (x>=y).

"x<=y" und "x>=y" zusammen "überdecken" das Prädikat "TRUE". "Überdeckung" erreichen wir freilich auch noch, wenn "x>=y" durch NOT(x<=y) = (x>y) ersetzt wird. Folglich wird die Selektions-Anweisung

```
IF x<=y THEN
  z:=x
ELSE
  z:=y
END (*IF*);
```

das Gewünschte leisten. Was wir hier getan haben, ist durch die Bemerkungen am Schluß von Abschnitt 4.2.3 bestens begründet. Dort haben wir Kriterien dafür aufgestellt, daß eine Selektions-Anweisung unter gegebener Vorbedingung zu einer verlangten Nachbedingung führt. Diese Kriterien können ohne weiteres als Anleitung zur Konstruktion von Selektions-Anweisungen interpretiert werden. Betrachten wir zunächst die allgemeine CASE-Anweisung. Hier lautet das "Rezept":

"Für alle i (i=1,...,n) finde Anweisungen Si, einen Ausdruck A und Werte Ci des Ausdrucks A derart, daß

P AND (A=Ci) => svb(Si,Q);

zeige, daß P => (EX i: 1<=i<=n: A=Ci)."

Auf die IF–THEN–ELSE-Anweisung reduziert liest sich diese Vorschrift wie folgt:

"Finde Anweisungen S1 und S2 und einen booleschen Ausdruck B derart, daß

P AND B => svb(S1,Q) und P AND NOT(B) => svb(S2,Q)."

Die Anweisungen S1 und S2 sind durch Inspektion von Q zu ermitteln. Als weiteres Beispiel zur Illustration dieser Vorschrift wollen wir ein Programmstück entwickeln, welches in einer Variablen z das Maximum von drei in den Variablen u,v und w gespeicherten ganzen Zahlen ablegt. Die geforderte Nachbedingung ist also "z = max(u,v,w)" oder - ausgeschrieben:

```
"Q: (z=u AND z>=v AND z>=w) OR
    (z=v AND z>=u AND z>=w) OR
    (z=w AND z>=u AND z>=v)"
```

Als erste mögliche Anweisung betrachten wir "z:=u" und berechnen:

svb(z:=u,Q) = (u=u AND u>=v AND u>=w) OR
| (u=v AND u>=u AND u>=w) OR
| (u=w AND u>=u AND u>=v)
= ...
= (u>=v AND u>=w).

Kandidat für B ist somit (u>=v) AND (u>=w). Eine erste Version des gesuchten Programmstücks ist:

```
(* TRUE *)
IF (u>=v) AND (u>=w) THEN
  z:=u
ELSE
  (* P2: u<v OR u<w *)
  S2
END (*IF*);
(* Q: ... *)
```

S2 muß ebenfalls die Erfüllung von Q gewährleisten, nun aber unter der eingeschränkten Vorbedingung P2 = NOT(B). Unserer Konstruktionsvorschrift entsprechend muß gelten: P2 => svb(S2,Q).

Der Nachbedingung Q entnehmen wir "z:=v" als weitere mögliche Zuweisung:

svb(z:=v,Q) = ...
= (v>=u AND v>=w)

Wie man leicht nachrechnet, ist B1 = (v>=w) ein boolescher Ausdruck derart, daß

P2 AND B1 => svb(z:=v,Q)

Eine nächste Version des Programmstücks ist daher

```
(* TRUE *)
IF (u>=v) AND (u>=w) THEN
  z:=u
ELSE
  (* P2: u<v OR u<w *)
  IF (v>=w) THEN
    z:=v
  ELSE
    (* P2 AND v<w *)
    S22
  END (*IF*)
END (*IF*);
(* Q: ... *)
```

Für "S22: z:=w" gilt nun, wie man ebenfalls leicht einsieht:

P2 AND (v<w) => svb(z:=w,Q) (= (w>=u) AND (w>=v)).

Berücksichtigt man noch die in 4.2.3 (zur Übung) vorgeschlagene Definition der Semantik der mit ELSIF geschachtelten IF-Anweisung, so ist

```
(* TRUE *)
IF (u>=v) AND (u>=w) THEN
  z:=u
ELSIF (v>=w) THEN
  z:=v
ELSE
  z:=w
END (*IF*);
(* Q: ... *)
```

ein Programmstück, das die gestellte Aufgabe löst.

4.3.2 Konstruktion von Iterations-Anweisungen

Die Konstruktion einer Iterations-Anweisung schließt die Bestimmung einer Schleifenbedingung und den "Bau" des Schleifenrumpfes ein. Beides muß auf Grund der Nachbedingung, einer Invarianten, einer Schrankenfunktion und der Vorbedingung geschehen. Während in den Beispielen (i) - (iii) des Abschnitts 4.2.4 diese Angaben im wesentlichen nur dazu benutzt wurden, die Korrektheit fertiger Programmstücke zu beweisen, werden wir sie hier zum Ausgangspunkt der Konstruktion machen. Wir nehmen dabei also insbesondere an, daß außer der Nachbedingung und der Vorbedingung (die im hier betrachteten Rahmen ja ohnehin immer gegeben sind) auch eine Invariante und eine Schrankenfunktion bekannt seien. Wie man diese geschickt wählt, wird - wiederum beispielhaft - in Abschnitt 4.3.3 dargestellt.

Wie üblich benennen wir eine Nachbedingung kurz mit Q, eine Vorbedingung mit P, eine Invariante mit InV und eine Schrankenfunktion mit Sf. Dann kann man - unter der Voraussetzung, daß InV und Sf bekannt sind - für die Konstruktion einer Iterations-Anweisung die folgende allgemeine Vorschrift angeben:

"Finde einen booleschen Ausdruck B mit InV AND NOT(B) => Q. Wähle B als Schleifenbedingung. Entwickle den Schleifenrumpf so, daß Sf durch dessen Ausführung vermindert wird und InV weiterhin gilt."

Der erste Teil dieser Vorschrift ist - wie schon die Beispiele (i) und (ii) in 4.2.4 zeigten - relativ leicht zu befolgen. Sorgfalt muß man jedoch auch dabei walten lassen. Der zweite Teil stellt im allgemeinen ein - kleineres oder größeres - Problem dar. Außerdem: Bevor man sich an die Konstruktion einer Iterations-Anweisung begibt, sollte klar sein, daß die jeweils vorliegende Aufgabe überhaupt

auf diese Weise gelöst werden kann. Darauf, wie diese Einsicht zu gewinnen ist, können wir in diesem Kapitel noch nicht eingehen.

Mit einigen Beispielen wollen wir die Anwendung der obigen Vorschrift einüben.

(i) Wir betrachten zunächst wieder die Aufgabe, aus einem mit ganzen Zahlen gefüllten Array a (vom Typ ARRAY [0..N-1] OF INTEGER) das größte Element zu bestimmen und in einer Variablen s zur Verfügung zu stellen (vgl. 4.2.4). N sei hier der Bezeichner für eine INTEGER-Konstante.

Die Vorbedingung P ist "TRUE".
Nachbedingung, Invariante und Schrankenfunktion sind wie in 4.2.4:

Q: s = max(a[0],...,a[N-1])
InV: (i<=N) AND (s = max(a[0],...,a[i-1]))
Sf: Sf(i) = N - i.

Wie in 4.2.4 erläutert, erhält man B = (i<>N) als Schleifenbedingung:

```
(* InV *)
(* Sf *)
WHILE (i<>N) DO
  S
  (* InV *)
END (*WHILE*);
(* (i=N) AND InV *)
(* Q *)
```

Wie muß S ausschauen? Sicherlich muß S eine Anweisung enthalten, welche den Wert der Schrankenfunktion vermindert. Die einfachste Möglichkeit hierzu bietet sich mit "i:=i+1". Wir gehen daher davon aus, daß S die Form hat "S1; i:=i+1". InV muß nach S weiterhin gelten, d.h.:

(i<>N) AND InV => svb(S1, svb(i:=i+1,InV))

Die von S1 herzustellende Nachbedingung lautet also:

svb(i:=i+1,InV) = (i+1<=N) AND (s=max(a[0],...,a[i]))
= (i<N) AND (s=max(a[0],...,a[i]))
= (i<N) AND (s=max(max(a[0],...,a[i-1]),a[i])).

Ohne dies noch ausführlicher zu schreiben, erkennt man bereits, daß die Zuweisung "s:=a[i]" in Frage kommt, um diese Nachbedingung zu schaffen. Alles weitere ergibt sich dann im wesentlichen nach dem Muster der in 4.3.1 behandelten Beispiele.

Eine Bemerkung zur Wahl der Schleifenbedingung scheint an dieser Stelle angebracht: Aufgrund von

(i=N) AND InV => Q

haben wir uns für B = NOT(i=N) = (i<>N) entschieden. Da auch (i>=N) AND

InV => Q, wäre B = NOT(i>=N) = (i<N) ebenfalls gerechtfertigt gewesen.

Gibt es einen Grund, einer dieser Möglichkeiten den Vorzug zu geben? Wir wissen, daß die Schleife nur dann korrekt abgelaufen ist, wenn die Variable i nach Beendigung den Wert N enthält. Ist dies nicht der Fall, so ist der weitere Programmablauf in Gefahr. Die Abbruchsbedingung "(i<>N)" garantiert nun, daß diese Gefahr nicht besteht. Wenn wir stattdessen mit der Abbruchsbedingung "(i<N)" arbeiten, so ist diese Garantie nicht gegeben. Es könnte ja sein, daß während der Ausführung des Schleifenrumpfes durch einen Fehler der Hardware (oder der Betriebssoftware) i einen Wert erhält, der größer ist als N.

Aus dieser Überlegung folgern manche Autoren (vgl. z.B. [GRI]), daß es ratsam ist, eine möglichst "schwache" (also umfassende) Schleifenbedingung zu verwenden. Tut man dies, so wird ein Fehler des Rechners eher zu einer "Endlos-Schleife" führen als zu einer inkorrekten Programmfortsetzung. (Was wünschenswerter ist, sei dahingestellt!)

(ii) Als zweites Beispiel dieses Abschnitts greifen wir die Berechnung des ganzzahligen Quotienten und des Restes bei der Division zweier ganzer Zahlen auf (vgl. 4.1.3 und 4.2.4). Grundlage der Programmentwicklung ist:

P: (x>0) AND (y>0)
Q: (r>=0) AND (y>r) AND (x=q*y+r)
InV: (r>=0) AND (x=q*y+r)
Sf: Sf(r) = r.

Die Inhalte von x und y dürfen durch das Programm nicht verändert werden. Die Schleifenbedingung ergibt sich unmittelbar aus dem Vergleich von Q und InV: B = (r>=y). (Im Gegensatz zu oben ist dies die einzig mögliche Wahl!)

Begründet durch den ersten Hauptzweck des Schleifenrumpfes S, nämlich den Wert von Sf zu vermindern, könnten wir mit einem ähnlichen Ansatz wie in Beispiel (i) beginnen: S = (S1; r:=r-1). S1 hat dann die Spezifikation

(* B AND InV *) S1 (* svb(r:=r-1, InV *)
mit svb(r:=r-1,InV) = (r>=1) AND (x=q*y+r-1)

zu erfüllen. "(r>=1)" gilt dabei wegen B und (y>0), denn y ist größer als Null aufgrund der Vorbedingung und weil es nicht verändert werden darf. Es bliebe also eine Manipulation des Inhalts von q, um svb(r:=r-1,InV) herzustellen.

Mit der Zuweisung "q:=q+1/y" könnten wir dieses Ergebnis erreichen, wenn wir es mit Zahlen vom Typ REAL zu tun hätten. Da dies nicht der Fall ist, verwerfen wir unseren ersten Ansatz und wählen stattdessen die Zerlegung S = (S1; q:=q+1). Sie führt uns direkt zu dem in Beispiel (iii) des Abschnitts 4.2.4 besprochenen Programm (,was der Leser zur Übung nachvollziehen möge).

(iii) Ein beliebtes Standardproblem der Informatik ist das Sortieren von Objekten, die einer Menge angehören, auf der eine Ordnungsrelation definiert ist. Wir wollen dieses Problem mit den Mitteln des "Programmierens durch Beweisen" angehen. Die Objekte seien wieder ganze Zahlen, die als Werte der Komponenten einer Variablen a vom Typ ARRAY [1..N] OF INTEGER gegeben sind.

Wir werden zwei Lösungen skizzieren. In beiden Fällen ist die Vorbedingung durch ein Prädikat P gegeben, welches zum Ausdruck bringt, daß in a irgendwelche Werte gespeichert sind. Auch die Nachbedingung Q ist natürlich immer die gleiche: In a müssen die Werte, die dort vorher in irgendeiner Reihenfolge enthalten waren, nun der Größe nach (z.B. aufsteigend) angeordnet sein. In 4.1.3 haben wir für diese Spezifikation bereits eine kompakte Notation eingeführt:

```
(* P: a = A *)
SORT
(* Q: perm(a,A) AND (ALL i: 1<=i<N: a[i]<=a[i+1]) *).
```

A ist dabei ein Objekt vom Typ der Variablen a.

Um im weiteren Verlauf unserer Ausführungen auch über "Teilstücke" von a reden zu können, vereinbaren wir noch die Schreibweise a[k..m] für die Folge "a[k],a[k+1],...,a[m]". Falls m<k, so ist diese Folge leer. Außerdem genügt es meistens, die Nachbedingung in der noch kürzeren Form

"sortiert(a[1..N])"

anzugeben.

Lösung 1:

Für die erste Lösung wählen wir die Invariante

InV: (k<=N) AND sortiert(a[1..k]) CAND (ALL i: k<i<=N: a[k]<=a[i])

und die Schrankenfunktion

Sf: Sf(k) = N-k.

(Wir verwenden "CAND", da k ja außerhalb des Index-Bereichs des Arrays a liegen könnte; in diesem Falle ist a[k] nicht definiert.)

Eine gute Schleifenbedingung zu finden ist, wie in den bisherigen Beispielen, nahezu trivial. Sie lautet: B = (k<>N) (vgl. Beispiel (i)!).

Entsprechend Beispiel (i) zerlegen wir den Schleifenrumpf: S = (S1; k:=k+1). S1 muß - wie üblich - die Spezifikation

(* B AND InV *) S1 (* svb(k:=k+1,InV) *)

mit

```
svb(k:=k+1,InV) = (k<N) AND sortiert(a[1..(k+1)])
                          CAND (ALL i: k+1<i<=N: a[k+1]<=a[i])
```

erfüllen. Die Vorbedingung von S1 lautet ausführlich:

(k<N) AND sortiert(a[1..k]) CAND (ALL i: k<i<=N: a[k]<=a[i]).

Die Wirkung von S1 sollte es daher offensichtlich sein, den kleinsten der Werte in a[k+1],...,a[N] in a[k+1] unterzubringen. Und dies muß so geschehen, daß die a[k+1],...,a[N] insgesamt nach Ausführung von S1 die gleichen Werte enthalten wie vorher.

Bezüglich dieses Teils des Arrays haben wir also ein Problem zu lösen, das charakterisiert ist durch:

P1: a[(k+1)..N] = A[(k+1)..N] (Vorbedingung)

Q1: perm(a[(k+1)..N], A[(k+1)..N]) CAND
(ALL i: k+1<i<=N: a[k+1]<=a[i]) (Nachbedingung).

Wir notieren das Programmstück, soweit es bisher feststeht:

```
(* P: a = A *)
(* Schleifeninitialisierung *)
(* InV: (k<=N) AND sortiert(a[1..k])
                CAND (ALL i: k<i<=N: a[k]<=a[i]) *)
(* Sf = Sf(k) = N-k *)
WHILE (k<>N) DO
  (* P1: a[(k+1)..N] = A[(k+1)..N] *)
  S1;
  (* Q1: perm(a[(k+1)..N],A[(k+1)..N]) CAND
     (ALL i: k+1<i<=N: a[k+1]<=a[i]) *)
  k:=k+1
  (* InV *)
END (*WHILE*);
(* (k=N) AND InV *)
(* Q: sortiert(a[1..N]) *)
```

Auch zur Erfüllung der Spezifikation (* P1 *) S1 (* Q1 *) bietet sich die Konstruktion einer Iterations-Anweisung an. Wir gehen dabei aus von der Invarianten

InV1: (m<=N) AND perm(a[(k+1)..N], A[(k+1)..N])
CAND (ALL i: k+1<i<=m: a[k+1]<=a[i])

und der Schrankenfunktion

Sf1: Sf1(m) = N-m.

Wir erkennen wieder leicht die Schleifenbedingung B1 = (m<>N) und die Zerlegung S1 = (S2; m:=m+1).

S2 hat die Vorbedingung

P2: B1 AND InV1 = (m<N) AND perm(...)
CAND (ALL i: k+1<i<=m: a[k+1]<=a[i])

und die Nachbedingung

Q2: svb(m:=m+1,InV1) = (m<N) AND perm(...)
CAND (ALL i: k+1<i<=m+1: a[k+1]<=a[i]).

Wie man sieht, gilt: Q2 = P2 AND (a[k+1]<=a[m+1]). Operationen, die sich zur Herstellung dieses Zustandes anbieten, sind "NONE" (vgl. 4.2.1) und die Vertauschung der Inhalte von a[k+1] und a[m+1]. S2 ist also als Selektions-Anweisung zu konstruieren. Dazu verweisen wir auf 4.3.1. Wir erhalten das folgende, mit Prädikaten kommentierte Programmstück:

```
(* P: a = A *)
(* Schleifeninitialisierung für S *)
(* InV: (k<=N) AND sortiert(a[1..k])
            CAND (ALL i: k<i<=N: a[k]<=a[i]) *)
(* Sf = Sf(k) = N-k *)
WHILE (k<>N) DO  (* S *)
  (* P1: a[(k+1)..N] = A[(k+1)..N] *)
  (* Schleifeninitialisierung für S1 *)
  (* InV1: (m<=N) AND perm(a[(k+1)..N], A[(k+1)..N])
             CAND (ALL i: k+1<i<=m: a[k+1]<=a[i]) *)
  (* Sf1 = Sf1(m) = N-m *)
  WHILE m<>N DO  (* S1 *)
    (* P2: (m<N) AND perm(...)
             CAND (ALL i: k+1<i<=m: a[k+1]<=a[i]) *)
    (* S2: *)
    IF (a[m+1]<a[k+1]) THEN
     (* Vertausche die Inhalte
      von a[m+1] und a[k+1] *)
    ELSE
     (* NONE *)
    END (*IF*);
    (* Q2: P2 AND (a[k+1]<=a[m+1]) *)
    m:=m+1
    (* InV1 *)
  END (*WHILE*)
  (* (m=N) AND InV1 *)
  (* Q1: perm(a[(k+1)..N], A[(k+1)..N])
                    CAND (ALL i: k+1<i<=N: a[k+1]<=a[i]) *)
  k:=k+1
  (* InV *)
END (*WHILE*);
```

(* (k=N) AND InV *)
(* Q: sortiert(a[1..N]) *)

Auszuarbeiten sind nur noch die Schleifeninitialisierungen und die Vertauschungsoperation. Erstere müssen die Gültigkeit der jeweiligen Invarianten bewirken. Für die Iteration über S tut dies die Zuweisung "k:=0" (da eine leere Folge trivialerweise sortiert ist) und für die Iteration über S1 die Zuweisung "m:=k+1" (da eine All-Aussage über die leere Menge immer wahr ist). Die Konstruktion der Vertauschungsoperation als geeignete Sequenz von Zuweisungen bleibe dem Leser zur Übung überlassen.

Lösung 2:

Die Entwicklung der zweiten Lösung basiert auf der Invarianten

InV: (k<=N) AND (q = N DIV k)
AND sortiert(a[1..k]) AND sortiert(a[k+1..2*k])
AND ...
AND sortiert(a[(q-1)*k+1..q*k])
AND sortiert(a[q*k+1..N])

und der Schrankenfunktion

Sf: Sf(k) = N-k.

("DIV" ist die Standardoperation, die zu zwei ganzen Zahlen den ganzzahligen Quotienten liefert (für dessen Berechnung wir uns schon einige Mühe gegeben haben).)

Eine gute Schleifenbedingung ist offenbar B = (k<>N). (Es gilt damit nämlich

InV AND NOT(B) = (k=N) AND (q=1) AND sortiert(a[1..N])
AND sortiert(a[N+1..N]),

und dies impliziert die Nachbedingung Q.)

Eine Operation, welche - gefolgt von einer entsprechenden Aktualisierung der Werte von k und q - die Invariante tatsächlich erhält, falls (2*k<=N) gilt, ist leicht zu finden (Übung!). Verzichten wir zunächst einmal auf die Aktualisierung des Wertes von k. Dann können wir einen dem Ausdruck InV ganz ähnlichen Ausdruck Q1 herstellen, indem wir die Paare (a[1..k],a[k+1..2*k]), (a[2*k+1..3*k],a[3*k+1..4*k]), ... bereits sortierter Teilfolgen zu ihrerseits sortierten Teilfolgen "vermischen":

Q1: (k<=N) AND (q = N DIV (2*k))
AND sortiert(a[1..2*k])
AND sortiert(a[2*k+1..4*k])
AND ...
AND sortiert(a[2*q*k+1..N]).

Wir nennen diese Operation "S1". Führen wir anschliessend die Zuweisungen "k:=2*k" und "q:=N DIV k"aus, so erhalten wir wieder das Prädikat InV.

Noch einfacher läßt sich InV wiederherstellen, wenn (k<>N) AND (2*k>N): In diesem Fall haben wir lediglich die laut Vorbedingung sortierten Teilfolgen a[1..k] und a[k+1..N] zur sortierten Folge a[1..N] zu "vermischen" und anschließend k:=N und q:=1 zu setzen.

Fassen wir dies zusammen:

```
(* P: a = A *)
(* Schleifeninitialisierung *)
(* InV:  siehe oben *)
(* Sf = Sf(k) = N-k *)
WHILE (k<>N) DO
  IF (2*k<N) THEN
    (* P1: (2*k<N) AND InV *)
    (* S1: siehe oben *)
    (* Q1: siehe oben *)
    k:=2*k;
    q:=N DIV k
  ELSE
    (* Vermische die Folgen a[1..k] und a[k+1..N] *)
    k:=N;
    q:=1
  END (*IF*)
  (* InV *)
END (*WHILE*)
(* (k=N) AND InV *)
(* Q: sortiert(a[1..N]) *)
```

Die Operation S1 kann ebenfalls durch eine Iterations-Anweisung erledigt werden. Die Invariante lautet:

```
InV1: (k<=N) AND (q = N DIV k) AND (2*m<=q)
              AND sortiert(a[1..2*k])
              AND ...
              AND sortiert(a[2*(m-1)*k+1..2*m*k])
              AND sortiert(a[2*m*k+1...(2*m+1)*k])
              AND ...
              AND sortiert(a[q*k+1..N]).
```

Die Schrankenfunktion ist

Sf1: Sf1(m) = q-m.

Als Schleifenbedingung ergibt sich B1 = (m<>q). Damit läßt sich das obige Programmstück ausbauen zu:

```
(* P: a = A *)
(* Schleifeninitialisierung für S *)
```

```
(* InV:  siehe oben *)
(* Sf = Sf(k) = N-k *)
WHILE (k<>N) DO  (* S *)
  IF (2*k<N) THEN
    (* P1: (2*k<N) AND InV *)
    (* Schleifeninitialisierung für S1 *)
    (* InV1: siehe oben *)
    (* Sf1 = Sf1(m) = q-m *)
    WHILE (m<>q) DO  (* S1 *)
      (* Vermische die Folgen a[2*m*k+1...(2*m+1)*k]
      und a[(2*m+1)*k+1...2*(m+1)*k] *)
      m:=m+1
      (* InV1 *)
    END (*WHILE*)
    (* (m=q) AND InV1 *)
    (* Q1: siehe oben *)
    k:=2*k;
    q:=N DIV k
  ELSE
    (* Vermische die Folgen a[1..k] und a[k+1..N] *)
    k:=N;
    q:=1
  END (*IF*)
  (* InV *)
END (*WHILE*)
(* (k=N) AND InV *)
(* Q: sortiert(a[1..N]) *)
```

Es verbleiben die Aufgabe, die Schleifen über S und S1 korrekt zu initialisieren und das Problem der Vermischung zweier sortierter Folgen zu einer sortierten Folge. An beidem möge sich der Leser wieder selbst versuchen. Normalerweise wird man die Operation des Vermischens einer allgemeinen Prozedur überlassen. Da wir aber die Semantik des Prozedur-Aufrufs nicht definiert haben, können wir uns an dieser Stelle (quasi um methodisch rein zu bleiben) dieses Hilfsmittels nicht bedienen.

Wir bemerken noch, daß wir bei der Konstruktion des Schleifenrumpfes S implizit die Vorschrift zur Bildung korrekter IF–THEN–ELSE-Anweisungen angewandt haben.

Ferner seien "Spezialisten in Sachen Sortierverfahren" darauf aufmerksam gemacht, daß das als Lösung 2 erarbeitete Programm eine iterative Version des "Sortierens durch Mischen" (englisch: "Mergesort") darstellt.

Beispiel (iii) ist außerordentlich lehrreich. Es zeigt, daß für die Konstruktion von Iterations-Anweisungen die jeweils verwendete Invariante eine dominierende Rolle spielt. Ja man kann sagen, daß die Invariante das Programm weitgehend bestimmt. In diesem Zusammenhang sei an die Diskussion in Abschnitt 2.1.4 erinnert. Dort ist uns (im Prinzip) das als Lösung 1 entwickelte Programmstück in der Prozedur "NaivSort" begegnet. (Mit geeigneten Semantik-Regeln könnten wir die oben gefundenen WHILE-Anweisungen in FOR-Anweisungen umwandeln!) Wir haben darauf hingewiesen, daß zum Beispiel "Mergesort" wesentlich effizienter arbeitet als dieser "NaivSort"-Algorithmus (O(N*log N) elementare Operationen anstatt $O(N^2)$). Im Lichte von Beispiel (iii) drängt sich damit natürlich die interessante Frage auf, welchen Einfluß die Wahl der Invarianten auf die Effizienz des sich ergebenden Programms hat. Es würde freilich in diesem Rahmen zu weit führen, wenn wir auf die Untersuchung dieses Problems weiter eingingen.

Beispiel (iii) zeigt außerdem, daß "Programmieren durch Beweisen" immer auch schrittweise Verfeinerung bedeutet: Für die Schaffung von Nachbedingungen "erfindet" man komplexe Operationen, zu deren Ausführung wiederum Programmstücke (bzw. Prozeduren) zu konstruieren sind, usw. .

Bei allen Beispielen dieses Abschnitts haben wir einen Aufwand getrieben, der vielen überzogen scheinen mag. Dieser Eindruck besteht zweifellos zu Recht. Die "Praxis des Programmierens" bedarf der hier zum Zweck der Übung aufgefahrenen "scharfen formalen Geschütze" sicher nicht, insbesondere dann nicht, wenn es - wie hier - um vergleichsweise triviale "Problemchen" geht. Dennoch ist es wichtig, sich mit dem methodischen Ansatz des "Programmierens durch Beweisen" auseinanderzusetzen. Diese Auseinandersetzung kann zweifellos zu einer besseren Einsicht in die Probleme der "Programmierung im Kleinen" führen. Mit dem Bewußtsein, eine Programmentwicklung "im Prinzip" vollständig formalisieren zu können, wird es wesentlich leichter fallen, auch informell die Korrektheit eines Programms plausibel zu machen. (Vgl. auch die Bemerkungen am Schluß von Abschnitt 4.2.4.) Beispielsweise kann die Kommentierung der Programme von Beispiel (iii) auf kurze verbale oder "halb-formale" Beschreibungen der zum Verständnis einer Schleife notwendigen Bedingungen und auf die Angabe der Schrankenfunktion reduziert werden.

4.3.3 Zur Auffindung von Schleifen-Invarianten

Da Invarianten - wie sich im letzten Abschnitt herausgestellt hat - sozusagen den "Treibstoff" für die Konstruktion von Iterations-Anweisungen liefern, ist es natürlich wichtig zu fragen, wie man solche Prädikate zweckmäßigerweise bildet. Gibt es Prinzipien oder gar Techniken, mit deren Hilfe sich gute Invarianten quasi von selbst ergeben?

Machen wir uns zunächst noch einmal das Verhältnis zwischen Nachbedingung und Invariante klar: Es muß ein Prädikat B (die Schleifenbedingung) geben mit

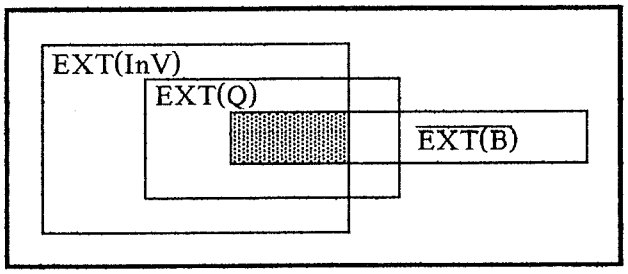
NOT(B) AND InV => Q und NOT(B) AND InV <> FALSE.

(Der Leser ist aufgefordert, sich zu überlegen, warum B AND InV <> FALSE gelten muß!)

Der Implikation entspricht die Beziehung

$$\overline{EXT(B)} \cap EXT(InV) \subset EXT(Q)$$

zwischen den Extensionen der beteiligten Prädikate, wobei EXT(B) ∩ EXT(InV) <> { }. NOT(B) schneidet gewissermaßen ein Stück aus EXT(InV) heraus, welches ganz in EXT(Q) enthalten ist:



Nennen wir das zu diesem "Stück" gehörige Prädikat Q', so gilt: Q' <> FALSE, Q' => Q und Q' => InV; das heißt, daß sowohl Q als auch InV Verallgemeinerungen von Q' sind.

Als (noch sehr vage) Vorschrift für die Bildung einer Invarianten könnten wir also formulieren:

"Spezialisiere Q zu Q' und finde eine von Q verschiedene Verallgemeinerung von Q'!"

Am zufriedensten wären wir natürlich, wenn wir gar nicht erst die Spezialisierung von Q zu Q' vornehmen müßten, sondern eine Invariante direkt als Verallgemeinerung von Q herleiten könnten.

Und in der Tat: Wenn wir die bisher studierten Beispiele Revue passieren lassen, so fällt auf, daß sich alle Invarianten durch irgendeine "Abschwächung" der Nachbedingung ergaben.

Diese Beobachtung wollen wir ein wenig systematisieren und durch weitere Beispiele belegen.

Wir unterscheiden vier (von zahlreichen) Möglichkeiten, ein gegebenes Prädikat "abzuschwächen", also zu verallgemeinern:

(i) Falls das Prädikat eine Konjunktion mehrerer Prädikate ist, so entferne man eines oder mehrere der Glieder dieser Konjunktion: Eine Abschwächung des Prädikats "A(x) AND B(y) AND C(x,y)" wäre etwa das Prädikat "B(y) AND C(x,y)".

(ii) Ersetze eine Konstante durch eine Variable: So entsteht beispielsweise aus dem Prädikat "(ALL i: 1<=i<=10: x < a[i])" das Prädikat "(k<=10) AND (ALL i: 1<=i<=k: x < a[i])", wenn wir anstelle der Konstanten "10" die neu eingeführte Variable "k" verwenden und deren Wertebereich so festlegen, daß er die ersetzte Konstante umfaßt.

(iii) Erweitere den Wertebereich einer Variablen: Das Prädikat "(0<i) AND (i<=100)" ist eine Verallgemeinerung von "(50<i) AND (i<=60)".

(iv) Füge ein Prädikat disjunktiv hinzu: Aus "A(x)" kann damit "A(x) OR B(x)" werden.

Für unser Problem, Invarianten zu finden, scheinen die ersten drei der genannten Möglichkeiten durchaus vielversprechend zu sein: Die Hinweise auf Verallgemeinerungen werden hier durch die Prädikate selbst gegeben. Möglichkeit (iv) hingegen ist offenbar nicht so ergiebig: Der Spielraum für "disjunktive Erweiterungen" ist einfach zu groß, er wird durch das zu verallgemeinernde Prädikat in keiner Weise eingeengt. Wir werden daher im folgenden zu jeder der Möglichkeiten (i) - (iii) einige Beispiele für die (damit) erfolgreiche Suche nach einer geeigneten Invarianten geben.

Beispiele zu (i):

Quadratwurzel: Gegeben sei eine nichtnegative ganze Zahl N. In einer Variablen a soll die größte ganze Zahl stehen, welche kleiner oder gleich der Quadratwurzel aus N ist. Gesucht ist also ein Wert von a, für den gilt:

$$Q: (a^2 <= N) \text{ AND } (N < (a+1)^2).$$

Dies ist die gewünschte Nachbedingung.

Lassen wir das zweite Glied dieser Konjunktion weg, so bleibt als mögliche Invariante übrig:

$$InV: (a^2 <= N).$$

Die Negation des entfernten Gliedes bietet sich als Schleifenbedingung an:

$$B = (N >= (a+1)*(a+1)).$$

Unser Ansatz für das gesuchte Programmstück ist:

```
(* Schleifeninitialisierung *)
(* InV: a^2 <= N *)
(* Sf = Sf(a) = N -a^2 *)
WHILE (N >= (a+1)*(a+1)) DO
  (* S *)
END (*WHILE*);
```

(* Q: (a^2 <= N) AND (N < $(a+1)^2$) *)

Die einfachste Art, den Wert von Sf zu vermindern, besteht darin, den Inhalt von a um 1 zu erhöhen. Wir setzen daher als Schleifenrumpf die Zuweisung "a:=a+1". Der Leser möge sich fragen, wodurch die Wahl von Sf begründet ist, nd zeigen, daß die Punkte der Checkliste erfüllt sind.

Lineare Suche: Neben dem Sortieren spielt das Suchen in Datenbeständen eine große Rolle (nicht nur) in der betrieblichen Datenverarbeitung. In einer seiner einfachsten Ausprägungen besteht das Problem darin, in einer - etwa als Array realisierten - Liste ein bestimmtes Objekt zu lokalisieren, indem man den Index derjenigen Zelle feststellt, in der das Objekt gespeichert ist.

Sei also a eine Variable vom Typ ARRAY [0..N-1] OF INTEGER und x eine Variable vom Typ INTEGER. Gesucht ist ein Programm, welches in einer Variablen i den Index der ersten Zelle von a abspeichert, deren Inhalt mit dem Inhalt von x identisch ist. Falls der Inhalt von x in a nicht gefunden wird, möge i den Wert N erhalten. Die Nachbedingung lautet somit:

```
Q: (0<=i<N) AND (ALL j: 0<=j<i: x<>a[j]) AND (x=a[i])
   OR
   (i=N) AND (ALL j: 0<=j<N: x<>a[j]).
```

(Die Vorbedingung - soweit sie die uns interessierenden Variablen betrifft - ist TRUE.)

Q ist keine reine Konjunktion, sondern eine Disjunktion von Konjunktionen. Aber auch dafür gilt natürlich, daß durch Weglassen eines konjunktiven Gliedes (in einer oder mehreren der Konjunktionen) eine Verallgemeinerung erreicht wird. Wir schlagen das folgende Prädikat als Invariante vor:

```
InV: (0<=i<N) AND (ALL j: 0<=j<i: x<>a[j])
     OR
     (ALL j: 0<=j<N: x<>a[j]).
```

Der Grund dafür, daß wir (x=a[i]) weglassen, ist naheliegend: Dies ist ja die wesentliche Bedingung dafür, daß wir das Ziel erreicht haben, falls sich der Wert von x in a befindet. Entsprechendes gilt für (i=N) im zweiten "Disjunkt" von Q. Durch Zusammenfassung der beiden "Disjunkte" wird InV zu:

```
InV: (0<=i<=N) AND (ALL j: 0<=j<i: x<>a[j]).
```

B AND InV muß nun die Tatsache ausdrücken, daß das Ziel noch nicht erreicht wurde. Nur unter dieser Bedingung ist der Schleifenrumpf S ein weiteres Mal auszuführen. Das Ziel ist erreicht, wenn - zusätzlich zur Invarianten - entweder (i<N) AND (x=a[i]) oder (i=N) wahr ist. Die Ausführung des Schleifenrumpfes soll uns dem Ziel näherbringen. Die "Entfernung zum Ziel" kann durch die Anzahl der noch zu durchsuchenden Zellen gemessen werden. Dies ist unsere Schrankenfunktion, für deren genaue Formulierung wir uns an der Invarianten

orientieren. Es ergibt sich das folgende Programmstück:

```
(* Schleifeninitialisierung *)
(* InV: (0<=i<=N) AND (ALL j: 0<=j<i: x<>a[j]) *)
(* Sf = Sf(i) = N-i *)
WHILE (i<>N) AND (x<>a[i]) DO
  (* S *)
END (*WHILE*);
(* Q: (0<=i<N) AND (ALL j: 0<=j<i: x<>a[j]) AND (x=a[i])
   OR
   (i=N) AND (ALL j: 0<=j<N: x<>a[j]) *)
```

Der Inhalt des Schleifenrumpfes kann unmittelbar durch die Schrankenfunktion begründet werden. Der Leser möge S finden und die Beweise gemäß "Check-Liste" zur Übung durchführen.

Eine Schwierigkeit im Zusammenhang mit der Schleifenbedingung sei noch bemerkt: Da a[i] nur definiert ist für i IN {0,...,N-1}, muß (i<N) gelten, damit a[i] ausgewertet werden kann. Hätten wir die Möglichkeit, in MODULA-2 den CAND-Operator zu benutzen, so wäre diese Schwierigkeit einfach mit (i<>N) CAND (x<>a[i]) zu umgehen. Ohne die Verfügbarkeit von "CAND" könnten wir zum Beispiel die Schleifenbedingung "verschärfen" zu (i<N) AND (x<>a[i]) und darauf vertrauen, daß a[i] (durch den von unserem Compiler erzeugten Code) nicht ausgewertet wird, wenn der Wert des ersten "Konjunkts" F ist.

Beispiele zu (ii):

Die zweite Methode zur Bildung von Invarianten haben wir in früheren Beispielen bereits unausgesprochen angewandt. Erinnert sei an die Berechnung der ganzzahligen Exponentialfunktion und des Maximums in einer Liste von ganzen Zahlen in Abschnitt 4.2.4 sowie an die Lösung "NaivSort" des Sortierproblems in 4.3.2.

Als weiteres Beispiel behandeln wir hier das sogenannte "Plateau-Problem": Dabei ist eine Variable a vom Typ ARRAY [0..N-1] OF INTEGER gegeben, für deren Inhalt gilt: sortiert(a[0..N-1]). Ein "Plateau" ist eine Folge a[i..j] mit (i>=0), (i<=j), (j<N) und (ALL k: i<=k<=j: a[k]=a[i]). (Vgl. die in 4.3.2 eingeführte Notation.) Gesucht ist die Länge des längsten Plateaus.

Da a sortiert ist, ist die Folge a[i..j] (mit 0<=i<=j<N) offenbar genau dann ein Plateau, wenn a[i]=a[j]. Daher lautet die Nachbedingung für eine Variable p, in der die Länge des längsten Plateaus abzuspeichern ist:

```
Q: (EX k: 0<=k<N-p: a[k]=a[k+p-1]) AND
   (ALL k: 0<=k<N-p-1: a[k]<>a[k+p]).
```

In die "Umgangssprache" übersetzt heißt dies, daß es ein Plateau der Länge p gibt, aber keines der Länge p+1.

Als Prädikat formuliert macht diese Aussage einen ziemlich umständlichen Eindruck. Für die Programmentwicklung scheint dieser Grad der Formalisierung nicht angemessen. Wir beschränken uns auf die folgende informelle Version der Nachbedingung:

Q: p enthält die Länge des längsten Plateaus von a[0..N-1].

Man kann bei Bedarf auf die strengere Formulierung zurückgreifen.

Die Konstante, die hier als einziger Kandidat für die Ersetzung durch eine Variable in Frage kommt, ist N. Nach bekanntem Muster setzen wir also:

InV: (0<i<=N) AND (p enthält die Länge des längsten Plateaus von a[0..i-1]).

Das heißt: Solange die Iteration läuft, soll die Länge des längsten Plateaus im bisher betrachteten (Anfangs-)Stück des Arrays a in p gespeichert sein. Eine Ausführung des Schleifenrumpfes muß jedenfalls dafür sorgen, daß der noch zu untersuchende Rest des Arrays verkleinert wird. Als Schrankenfunktion bietet sich deshalb an: Sf = Sf(i) = N-i. Da ein Array mit einem Element genau ein Plateau besitzt, dessen Länge 1 ist, kann die Gültigkeit der Invarianten mit den Zuweisungen "i:=1; p:=1" erstmals gesichert werden. Die Schleifenbedingung ist (i<>N). Es ergibt sich das folgende Programmstück:

```
(* Schleifeninitialisierung *)
i:=1; p:=1;
(* InV: (0<i<=N) AND (p enthält die Länge des längsten
                      Plateaus von a[0..i-1]) *)
(* Sf = Sf(i) = N-i *)
WHILE (i<>N) DO
  (* S *)
END (*WHILE*);
(* Q: p enthält die Länge des längsten Plateaus von a[0..N-1] *)
```

Die Ausarbeitung des Schleifenrumpfes erfolgt wie in 4.3.2 mit dem Ansatz "S1; i:=i+1" für S. Die Zuweisung "i:=i+1" verlängert das zu betrachtende Anfangsstück des Arrays um 1. Dabei kann die Länge des längsten Plateaus höchstens um 1 wachsen. In S1 muß also unter Umständen die Zuweisung "p:=p+1" ausgeführt werden. Es bleibt zu untersuchen, unter welchen Bedingungen dies zu geschehen hat. Dazu ist es nützlich, sich wieder an die formale Version der Invarianten zu erinnern. Im übrigen sei auf die Ausführungen in Abschnitt 4.3.1 verwiesen. Man findet für S1:

```
"IF a[i]=a[i-p] THEN
   p:=p+1
 END (*IF*);"
```

Beispiele zu (iii):

Auch den dritten Weg, Invarianten zu finden, haben wir bereits früher - im Beispiel des Divisions-Programms - beschritten. Dort haben wir zur Verallgemeinerung der Nachbedingung den Wertebereich von y erweitert.

Wir wollen die Anwendung einer "kreativen Weiterentwicklung" dieser Methode am Beispiel der "Linearen Suche" studieren. Die Schleife sollte uns dort in einer Variablen i den Index der ersten Zelle eines Arrays liefern, die einen gegebenen Wert enthält (bzw. N, falls der gegebene Wert nicht im Array enthalten ist).

Wir gehen hier davon aus, daß dieser minimale Index (der ja in jedem Fall existiert) bekannt ist. Wir bezeichnen ihn mit "Imin". Die Nachbedingung ist dann einfach:

Q: (i=Imin).

Mit "geschärftem Problembewußtsein" entdecken wir die problemgerechte Erweiterung des Wertebereichs von i und versuchen es mit der Invarianten

InV: (0<=i<=Imin).

Da Imin entweder der erste Index ist mit x=a[i] oder aber Imin=N zutrifft, muß

InV AND ((x=a[i]) OR (i=N)) => Q

gelten.

Dieses Beispiel zeigt, daß es sinnvoll sein kann, zur Formulierung von Nachbedingungen und - daraus abgeleitet - Invarianten gesuchte (oder zu konstruierende) Objekte wie Konstanten zu behandeln und den Schleifenrumpf als ein Vehikel zu betrachten, welches den Inhalt einer Variablen (eventuell von zusammengesetztem Typ) näher an diese Konstante heranbringt. Ein durchaus willkommener Nebeneffekt ist dabei eine mitunter beträchtliche Verminderung des formalen Aufwandes.

Schlußbemerkung:

Daß man bei der Entdeckung von Invarianten einige Phantasie entwickeln kann (und sollte), beweist - neben dem letzten Beispiel - in besonders eindrucksvoller Weise auch das Beispiel "Sortieren durch Mischen" (4.3.2). Die dort gefundene Invariante ist durch keine der drei hier besprochenen Methoden ohne weiteres zu begründen.

Daß man Phantasie gelegentlich einsetzen *muß*, sei zum Abschluß dieses Kapitels an einem Beispiel demonstriert, das - zugegebenermaßen - ziemlich trivial ist und "an den Haaren herbeigezogen" scheint (vgl. 4.1.3):

P: (a=A) AND (b=B) AND (a>=0) AND (b>=0)
Q: (a=A) AND (b=B) AND (z=a*b).

Wir nehmen an, daß die uns zur Verfügung stehende Programmiersprache nur Addition erlaubt, so daß die Zuweisung "z:=a*b" das "Problem" nicht löst.

Q selbst gibt nun eigentlich keine Hinweise zur Verallgemeinerung: Keine der drei obigen Methoden scheint erfolgversprechend. Wir bemerken aber, daß die Inhalte von a und b nicht verändert werden dürfen, daß also die Namen dieser Variablen auch als Konstanten-Namen aufgefaßt werden können. Wir ersetzen daher eine dieser "Konstanten" (z.B. b) durch eine neue Variable (z.B. mit dem Namen y) und formulieren damit - versuchsweise - eine Invariante:

$$\text{InV: } a=A \text{ AND } b=B \text{ AND } (0<=y<=b) \text{ AND } (z=a*y)$$

Die Gültigkeit dieser Invarianten wird am einfachsten durch "z:=0; y:=0" erreicht. Diese Zuweisungen bieten sich daher als Initialisierungen einer Schleife an, deren Fertigstellung und Begründung - wie gesagt - trivial sind.

Literatur zu Kapitel 4

[BEN] Bergmann, E. und H. Noll: Mathematische Logik mit Informatik-Anwendungen; Springer Verlag; Berlin, Heidelberg, New York; 1977

[GRI] Gries, D.: The Science of Programming; Springer-Verlag; New York, Heidelberg, Berlin; 1981

[SAL] Sale, A.: MODULA-2, Durch systematischen Entwurf zum korrekten Programm; Addison-Wesley Verlag (Deutschland); Bonn; 1987

5 Datenstrukturierter Programmentwurf

Im vorigen Kapitel haben wir eine Möglichkeit kennengelernt, Programme ausgehend von einer Beschreibung ihrer beabsichtigten Wirkung zu entwickeln. In Betracht gezogen haben wir dabei allerdings nur die erste jener "Wirkungsarten", von denen in der Einleitung des Abschnitts 4.1 die Rede war: die Veränderung interner Zustände. Dies ist eine Sichtweise, die dem Programmierer für viele interessante Probleme sicher zu eleganten Lösungen verhilft. Für eine große Klasse von Aufgabenstellungen jedoch ist sie völlig unzureichend und unpraktikabel. Es handelt sich hierbei um solche Aufgaben, die sich sehr viel leichter und "natürlicher" durch die Angabe der Struktur eines gewünschten Outputs oder eines aufzunehmenden Inputs spezifizieren lassen, als mit Hilfe prädikativer Beschreibungen interner Speicherzustände.

Beispiele:

(i) Es ist ein Programm zu schreiben, welches die Ausgabe einer Multiplikations-Tabelle für das kleine Einmaleins bewirkt. Die Tabelle soll in unterer Dreiecksform auf dem Bildschirm erscheinen.

(ii) Auf einem Graphik-Bildschirm, dessen Pixel durch Angabe ihrer Koordinaten einzeln adressierbar sind, soll ein 8*8 Felder umfassendes quadratisches Gitternetz gezeichnet werden.

(iii) Von der Tastatur sind ein Kleinbuchstabe, ein Gleichheitszeichen und ein Großbuchstabe (in dieser Reihenfolge) entgegenzunehmen. Davon abweichende Eingaben sind zu ignorieren.

(iv) Von einer durch das "Blank"-Zeichen abgeschlossenen Zeichenkette soll festgestellt werden, ob sie eine "Gleitkommazahl" darstellt oder nicht.

Die Beispiele (i) und (ii) lassen sich sehr leicht als Aufgaben folgenden Typs erkennen:

"Der Rechner ist zu veranlassen, ein bestimmtes
Objekt zu *produzieren*."

Dagegen lautet die Verallgemeinerung der Beispiele (iii) und (iv):

"Der Rechner soll ein gegebenes Objekt *analysieren* und prüfen,
ob es zu einer bestimmten Klasse von Objekten gehört."

Eine Aufgabe des erstgenannten Typs hat im Prinzip auch der Fertigungs-Ingenieur, der die zur Herstellung eines (handfesten, greifbaren) Produkts notwendigen Prozeduren festlegt. Es ist klar, daß er diese Prozeduren nicht unabhängig von der Struktur des gewünschten Produkts definieren kann: Der Zusammenbau eines Möbelstücks unterscheidet sich im Detail beträchtlich von der Montage eines Automobils.

Dieser Analogie zwischen Fertigungsunternehmen und Rechner gemäß können wir viele Programme schlicht als "Anweisungen zur Steuerung eines Produktionsprozesses" verstehen. Und diese sollten sich - naheliegenderweise - an der Struktur des zu produzierenden Gegenstands orientieren. Was also wäre natürlicher, als die Objektstruktur zu beschreiben und die Abfolge der Produktionsanweisungen (das Programm!) aus dieser Beschreibung herzuleiten?

Absolut zwingend ist die Orientierung an der Struktur eines Objekts bei Aufgaben des zweiten Typs. Hierzu ein weiteres Beispiel: Ein Programm, welches herausfindet, ob ein gegebener Text ein syntaktisch korrektes MODULA-2 - Programm ist oder nicht, kann nur auf der Grundlage einer exakten Beschreibung der Syntax dieser Programmiersprache erstellt werden. Auch hier ist ein Vergleich mit anschaulichen Vorgängen im alltäglichen Leben ohne weiteres möglich: Der Sachbearbeiter in einem Büro etwa, der sich um eingehende Bestellungen zu kümmern hat, muß jeweils entscheiden, ob eine Bestellung korrekt aufgegeben wurde und damit weiterverarbeitet werden kann oder nicht. Er muß diese Entscheidung aufgrund seiner Kenntnis der Struktur des "Objekts Bestellung" treffen, wie sonst? Den Programmen zur Lösung von Problemen der zweiten Art entsprechen also - anschaulich - "Anweisungen zur Steuerung von Analyseprozessen".

Wir wollen die gedankliche Vorbereitung dieses Kapitels durch die Diskussion zweier wichtiger Aspekte ergänzen.

Zum ersten ist zu bemerken, daß die oben beschriebenen Aufgabentypen - nennen wir sie "Produktionstyp" und "Analysetyp" - gewissermaßen Extremfälle einer noch größeren Problemklasse darstellen. Aufgaben vom Produktionstyp verlangen die Konstruktion von Objekten sozusagen "auf Knopfdruck"; *der Input ist jedenfalls trivial*, von ihm ist nicht die Rede. Wir wissen, wie das Resultat ausschauen soll, und damit basta! Umgekehrt liefert die Lösung von Aufgaben des Analysetyps einen *trivialen Output*, nämlich die Antwort "JA" (akzeptiert!) oder "NEIN" (abgelehnt!). Ob wir diesen Output wirklich zu Gesicht bekommen oder (im anderen Fall) den Input (Knopfdruck!) selbst tätigen müssen, ist dabei eine völlig zweitrangige Frage. Vielleicht erfolgt anstelle eines "echten" Inputs "nur" ein Prozeduraufruf oder anstelle eines "echten" Outputs erhält eine boolesche Variable den Wert TRUE oder FALSE, wie auch immer.

Warum aber sprechen wir von "Extremfällen"? Gibt es etwas dazwischen? Kehren wir zur Begründung einer Antwort auf diese Frage nochmals in den Fertigungsbetrieb zurück und schauen wir ein wenig genauer dem zu, was dort geschieht! Wir stellen dann fest, daß die Produkte keineswegs nur mittels betriebsinterner Ressourcen oder gar "aus dem Nichts heraus" fabriziert werden. Vielmehr werden immer wieder von außen angelieferte Teile benötigt, die - eventuell nach irgendeiner Bearbeitung - zum Endprodukt beitragen. Gelegentlich werden wohl fehlerhafte Teile ankommen, oder ein Teil erscheint an einer

Stelle, an der es gar nichts zu suchen hat, usw. Bei der Fertigungsplanung sind solche Anomalien natürlich in geeigneter Weise zu berücksichtigen: Analyse und Produktion beziehungsweise - besser - Input und Output müssen "Hand in Hand" ablaufen. Und wenn mit dem Input etwas "schief läuft", so darf dies keine katastrophalen Auswirkungen auf den Output haben.

Ganz genauso verhält es sich nun mit vielen Programmieraufgaben: Wie ein Fertigungsplan muß das Programm einen Produktions*prozeß* steuern, indem es dafür sorgt, daß zur Herstellung von Output-Objekten die jeweils passenden Input-Objekte herangeschafft werden. Es muß Vorkehrungen für den Fall treffen, daß unpassende Input-Objekte abgelehnt werden, und unter Umständen muß es sogar den Abbruch des Herstellungsprozesses vorsehen, wenn sich herausstellt, daß kein passender Input mehr zur Verfügung steht. Der in Abschnitt 3.3 in anderem Zusammenhang behandelte "Rechnungs-Ausschrieb" mag als Beispiel einer derartigen Aufgabe dienen. Die zu produzierenden Output-Objekte waren dort eben "Rechnungen"; die notwendigen Input-Objekte wurden von den Dateien "LIEFERUNG", "KUNDENLISTE" und "PREISLISTE" geholt. Für die Entwicklung eines Programms zur Lösung einer solchen Aufgabe ist es natürlich nach wie vor sinnvoll, sich - wie oben angedeutet - an den Strukturen der Output- und Input-Objekte zu orientieren. Da also beide Strukturen einzubeziehen sind, scheint dies zunächst schwieriger zu sein, als in den oben genannten Extremfällen. Wir werden in diesem Kapitel aber gerade hierfür ein Verfahren kennenlernen, welches sich durch die Abfolge einiger sauber gegeneinander abgrenzbarer Entwurfsschritte auszeichnet und dadurch sehr gut beherrschbar ist. (Unter anderem werden wir das Verfahren auch auf den"Rechnungs-Ausschrieb" anwenden.)

Halten wir fest: Die Analogie "Programm <-> Fertigungsplan" wird das Leitmotiv dieses Kapitels sein. Sie ist begründet durch die Analogie "Rechner <-> Fertigungsbetrieb".

Ebenfalls aus dieser Analogie ergibt sich - wenn man so will - der zweite für das Verständnis und die Einordnung dieses Kapitels überaus wichtige Gesichtspunkt: Produktion und Analyse haben eine *zeitliche Dimension*! (Der im Zusammenhang mit Produktion und Analyse gebrauchte Begriff "Prozeß" drückte dies bereits implizit aus.) So trivial diese Beobachtung zu sein scheint, so bedeutungsvoll ist sie für uns. Sie rechtfertigt es, die Struktur eines zu produzierenden Objekts nicht aussschließlich statisch zu begreifen. Vielmehr ist es in Hinblick auf die Programmentwicklung (die "Fertigungsplanung"!!) sehr nützlich, ein Objekt von vornherein durch eine zeitliche Abfolge seiner Teile beschreiben zu können, so wie sie bei einem Zusammensetzvorgang möglich wäre. Eine derartige "dynamische" Beschreibung eines statischen Objekts vom Typ "Zeichenkette" aus Beispiel (iii) hätte den Wortlaut: "Es beginnt mit einem Kleinbuchstaben, *danach folgt* ein Gleichheitszeichen, *danach folgt* ein Großbuchstabe." Die Multiplikati-

onstabelle aus Beispiel (i) wäre zu beschreiben als Folge von zehn Zeilen und jede Zeile ihrerseits als Folge von INTEGER-Objekten. Besonders naheliegend ist diese Betrachtungsweise für sequentiell organisierte Dateien, auf deren Sätze also nur in strikter Reihenfolge zugegriffen werden kann. Wir werden sehen, daß solche "dynamischen" Beschreibungen ein Programm bereits weitgehend implizieren können. In diesem Sinne ist die Situation hier vergleichbar mit der Entwicklung von Iterationsanweisungen aus Invarianten (vgl. die Bemerkungen am Ende von Abschnitt 4.3.2).

Die Auffassung eines komplexen Datenobjekts als strukturierter "Datenstrom" (als zeitliche Folge elementarerer Datenobjekte also) legt es natürlich nahe, nun auch umgekehrt strukturierte Datenströme als "Datenobjekte mit zeitlicher Dimension" anzusehen, die es zu produzieren oder zu analysieren gilt. Diese Sichtweise gestattet es, auch und gerade solche Aufgaben mit den Mitteln des "Datenstrukturierten Programmentwurfs" anzugehen, die mit der Steuerung technischer Prozesse (in Kraftwerken, in Maschinen aller Art etc.) zu tun haben. (Sie fallen in das spezielle Gebiet der "Realzeit-" bzw. "Echtzeit-Programmierung".) Das Datenobjekt etwa, welches im Beispiel 2 des Abschnitts 2.1.1 produziert wird, ist eine Folge von Signalen "ein" oder "aus", die einen Schalter ein- bzw. ausschalten. Zur Spezifikation des Programms, welches diesen Output veranlaßt, genügt es anzugeben, welche Inputs für welchen "Teil des Outputs" verantwortlich sind.

Wir beginnen dieses Kapitel mit der Einführung einer Notation zur Beschreibung der Struktur von Datenobjekten. Im Prinzip ist uns diese Notation bereits aus Kapitel 3, Abschnitt 3.3.2 bekannt: Dort haben wir Klammerdiagramme zur Unterstützung der schrittweisen Verfeinerung von Programm-Entwürfen verwendet; hier werden wir sie zunächst zur Darstellung von Objektstrukturen heranziehen. Im Hauptteil des Kapitels wird man dann sehen, daß dies unserer Absicht, Programme aus Objektstrukturen zu entwickeln, bestens entgegenkommt. Im Idealfall nämlich ergibt sich die Diagrammform einer Programmbeschreibung durch Ergänzungen und einfache Transformationen derjenigen Diagramme, welche die Strukturen der Datenströme wiedergeben. Der in diesem Kapitel vorgestellte Zugang zur Programmkonstruktion wurde insbesondere von M. Jackson ([JA1]) popularisiert und für den Einsatz im sogenannten "kommerziellen" Bereich empfohlen. (In [JA1] werden Datenstrom-Strukturen graphisch und Programm-Strukturen textuell - mit Hilfe eines Pseudo-Codes - repräsentiert.) Wie oben jedoch bereits angedeutet, werden wir mit einigen unserer Beispiele zeigen, daß der Ansatz des "*Datenstrukturierten Entwurfs*" auch bei technischen Anwendungen nützlich sein kann. Eine einheitliche Notation für die Beschreibungen sowohl der Objekt- als auch der Programmstrukturen wurde erstmals in [EHL] (vgl. auch Abschnitt 3.3.2) für ein industrielles Projekt vorgeschlagen.

5.1 Objektstruktur und Programmstruktur

Wir haben bisher (und nicht nur in der Einleitung zu diesem Kapitel) viel von "Objekten" geredet, ohne zu sagen, was genau darunter zu verstehen ist. Wir haben darauf ganz bewußt verzichtet und darauf vertraut, daß sich der Leser im jeweiligen Kontext eine intuitive Vorstellung vom Inhalt dieses Begriffes machen kann. Diese Fähigkeit werden wir auch bis auf weiteres voraussetzen. Erst im folgenden Kapitel (Abschnitt 6.4.4) werden wir eine spezielle - und dann etwas präzisere - Interpretation des Objektbegriffs einführen, die einen modernen Ansatz für den Entwurf und die Realisierung von Software-Systemen begründet. Vorläufig mag es genügen, unsere Objekte durch den Zusatz "Daten-" (wie oben schon mehrfach geschehen) zu qualifizieren. So bringen wir zum Ausdruck, daß es sich um rechnerspezifische Repräsentationen irgendwelcher Dinge handelt, deren Manipulation (Eingabe, Verarbeitung, Ausgabe) durch (Rechner-)Programme steuerbar ist. Und in Verallgemeinerung der Datenobjekte, die etwa mit Hilfe der Konstrukte einer höheren Programmiersprache deklarierbar sind, beziehen wir - wie oben angekündigt - die Dimension der Zeit mit ein. Tatsächlich werden wir die Struktur von Datenströmen, also von "in der Zeit ausgebreiteten bzw. ausbreitbaren Datenobjekten", völlig in den Vordergrund stellen.

5.1.1 Strukturbeschreibungen

Datenobjekte sind entweder elementar oder zusammengesetzt (komplex). Die Bestandteile (Komponenten) komplexer Datenobjekte sind entweder elementar oder ihrerseits komplexe Datenobjekte. Komplexe Datenobjekte sollten sich als irgendwie geartete Verknüpfung ihrer Komponenten darstellen lassen. Beispielsweise kann jede Komponente einen expliziten Hinweis auf die übrigen mit ihr verbundenen Teile erhalten. Damit ist die Struktur eines komplexen Datenobjekts durch einen im Prinzip beliebigen Graphen beschreibbar. (Bekanntlich bieten höhere Programmierspachen wie PASCAL und MODULA-2 für die Implementierung dieser allgemeinsten Art der Verknüpfung ein Zeiger-Konzept an.) Es ist klar, daß sich für die Behandlung derart "willkürlich" strukturierter Datenobjekte keine einheitlichen Verfahren angeben lassen. Da es in diesem Kapitel aber unser Ziel ist, ein auf alle Datenobjekte aus einer großen Klasse gleichermaßen anwendbares Verfahren vorzustellen, müssen wir die Möglichkeiten der Verknüpfung von Datenobjekten erheblich einschränken. Wir lassen zu:

- zeitliche Sequentialität ("danach folgt..."),
- räumliche Sequentialität ("daneben steht..."),
- räumliche Parallelität ("es gibt noch..."),
- "Iterationsobjekte",
- "Selektionsobjekte".

Was darunter im Einzelnen zu verstehen ist, wollen wir uns in diesem Abschnitt anhand einiger Beispiele klarmachen.

Beginnen wir mit der folgenden einfachen Aufgabe:

"In einer Bank soll am Ende eines jeden Tages ein Bericht über die Kontobewegungen, die an diesem Tag stattgefunden haben, produziert werden. Der Bericht muß - nach aufsteigenden Kontonummern sortiert - je Kontonummer die Summe der getätigten Bewegungen ausweisen. Außerdem soll die Summe der Bewegungen über alle Konten ausgegeben werden. Hier ein Beispiel-Bericht:

KONTENBEWEGUNGSBERICHT VOM 27. 3. 1979

KONTO-NR	TAGESSALDO
515920-605	- 1002.00
632970-315	326.00
653791-750	1002.00
SUMME:	326.00

Die Kontobewegungen entstehen aufgrund von Buchungs-Transaktionen. Davon gibt es zwei Arten: Soll- und Haben-Buchungen.

Zur Produktion des Berichts steht eine sequentielle Datei zur Verfügung, in der je Transaktion ein Satz gespeichert ist, der die Transaktionsart, die Kontonummer und den bewegten Betrag enthält. Sie ist bereits nach aufsteigenden Kontonummern sortiert. Die folgende Datei könnte dem obigen Bericht zugrundeliegen:

H	515920-605	501.00	
H	515920-605	501.00	Buchungssätze
S	515920-605	2004.00	
H	632970-315	1000.00	
S	632970-315	500.00	
S	632970-315	174.00	
H	653791-750	1600.00	
S	653791-750	598.00	
|	|	|	
Transaktionsart	Kontonummer	bewegter Betrag "	

Soweit die Aufgabe. Eine genauere Betrachtung des gewünschten Outputs läßt mindestens drei deutlich voneinander zu unterscheidende und daher auch *benennbare* Berichtsteile erkennen, die - so jedenfalls können wir die Situation sehen - *aufeinander folgen*:

Berichts-Kopf	KONTENBEWEGUNGSBERICHT VOM 27. 3. 1979	
	KONTO-NR	TAGESSALDO
	---	---
Berichts-Rumpf	515920-605	- 1002.00
	632970-315	326.00
	653791-750	1002.00
Berichts-Fuß	---	---
	SUMME:	326.00

Der Berichtsrumpf seinerseits ist eine *Folge gleichartiger Komponenten*, der "Berichtszeilen". Während jeder Bericht aus den drei erstgenannten Komponenten "Kopf", "Rumpf" und "Fuß" besteht, ist die Anzahl der Berichtszeilen von Exemplar zu Exemplar verschieden. Wir sprechen in diesem Fall von einer *Iteration.*

Iterationen finden wir im übrigen auch als dominierende Strukturen des Inputs:

H	515920-605	501.00
H	515920-605	501.00
S	515920-605	2004.00
H	632970-315	1000.00
S	632970-315	500.00
S	632970-315	174.00
H	653791-750	1600.00
S	653791-750	598.00

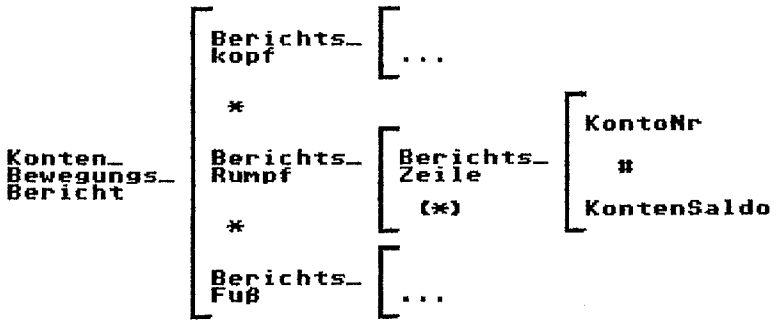
Eine Iteration von Buchungssätzen zum gleichen Konto

Eine Iteration von "Gruppen" (,welche aus Sätzen, die alle zum gleichen Konto gehören, bestehen).

Obwohl die Iterationen auch hier über gleichstrukturierte Komponenten (Buchungssätze) laufen, ist es sinnvoll, die Satzarten ("S" für Soll und "H" für Haben) auseinanderzuhalten.

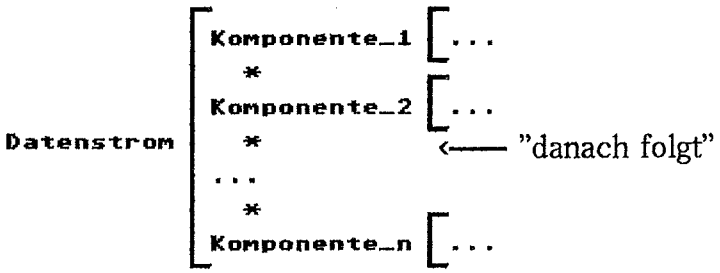
Ein Buchungssatz ist demnach ein "Selektionsobjekt": Entweder ist es ein "Sollsatz" oder ein "Habensatz".

Die Darstellung der Strukturen der als Datenströme aufgefaßten Objekte "Kontenbewegungsbericht" und "Buchungsdatei" durch Klammerdiagramme (vgl. Abschnitt 3.3.2) ergibt sich nun beinahe zwangsläufig. Betrachten wir zunächst den Bericht:



Im Abschnitt 3.3.2 benutzten wir das Zeichen "*" zur sequentiellen Verknüpfung von Anweisungen. Hier bedeutet es die zeitliche Aufeinanderfolge von Komponenten eines Datenstroms. Entsprechend ist "(*)" als (zeitliche!) Iteration von Objekten mit der Struktur "Berichtszeile" zu interpretieren. "Berichtszeilen" sind als "räumliche Objekte" modelliert; das Zeichen "#" bedeutet: "daneben steht". Wir betonen (nochmals), daß mit diesem Diagramm nicht ein bestimmter Bericht beschrieben wird, sondern die gemeinsame Struktur, der Bauplan gewissermaßen, aller Kontenbewegungsberichte. Was wir hier tun, ist also vergleichbar mit der Deklaration eines Typs in einer höheren Programmiersprache!

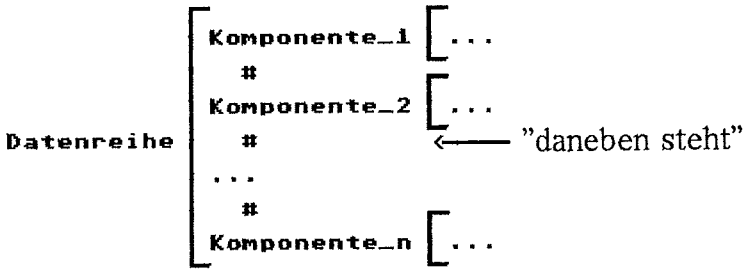
Generell hat ein Datenstrom die folgende Gestalt:



Dabei sind die Komponenten in der Regel alle von unterschiedlicher Struktur. Gleichartige Komponenten in einem Datenstrom können zu einem (zeitlichen) "Iterationsobjekt" zusammengefaßt werden:

"(*)" heißt: "unbestimmt oft (vielleicht auch keinmal!) folgt...".

Die allgemeine Form des (räumlich gedachten) "Nebeneinanderstehens" ist:

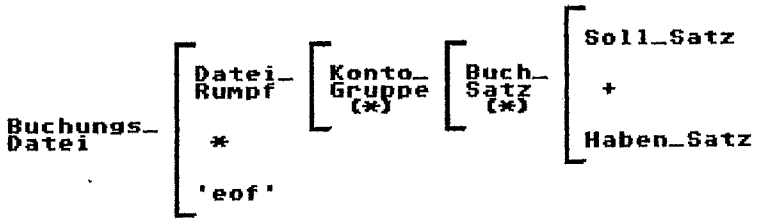


Auch bezüglich der "Aneinander-Reihung" von Komponenten mittels des Operators "#" gibt es eine entsprechende Iterations-Konstruktion, die aus mehreren nebeneinander stehenden gleichartigen Komponenten ein (räumliches) Iterationsobjekt macht:

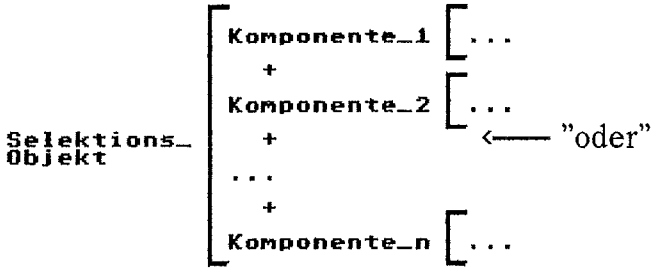
Dies ist eine Aneinanderreihung von unbestimmt vielen (auch Null ist möglich!) Komponenten der Art "Iter–Komponente". Wir kennzeichnen diese Objektstruktur mit "R1", weil wir weiter unten noch eine zweite Konstruktion zur Vielfachbildung ("im Raum") betrachten werden.

Bevor wir dies tun und bevor wir uns fragen, unter welchen Umständen es zweckmäßig ist, ein Datenobjekt so oder so zu beschreiben (da sich doch zum Beispiel Beschreibungen als "Strom" und als "Reihe" oft gleichermaßen anzubieten scheinen), wollen wir die obigen Notationen ergänzen, verfeinern und uns mit ihnen anhand weiterer Beispiele ein wenig vertrauter machen.

Für die noch zur Debatte stehende Aufgabe etwa, kann die Struktur des Inputs "Buchungsdatei" durch das folgende Diagramm dargestellt werden:

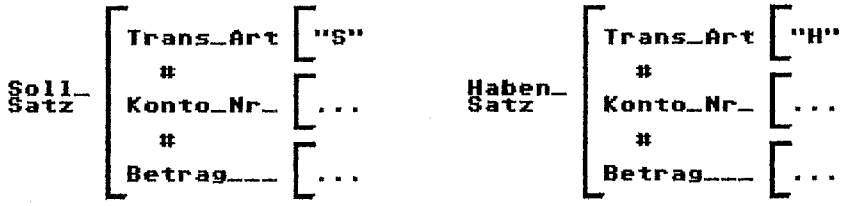


Die Tatsache, daß "Buchsatz" entweder ein "Soll–Satz" oder ein "Haben–Satz", also ein "Selektionsobjekt" ist, wird dabei durch das Zeichen "+" symbolisiert. Es bedeutet "oder". Allgemein lautet die Strukturbeschreibung eines Selektionsobjekts:



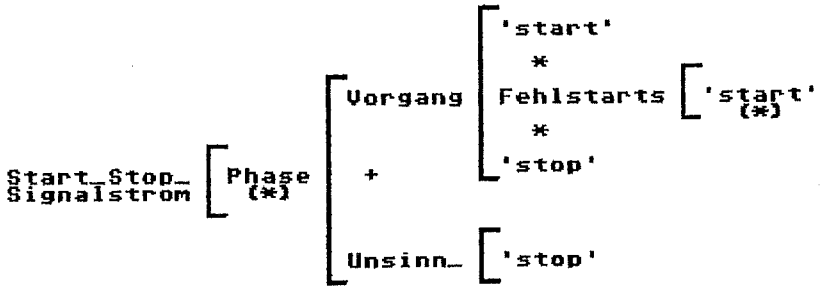
In der obigen Darstellung der Struktur von "Buchungsdatei" folgt auf die Sätze ein spezielles elementares Objekt 'eof'. 'eof' steht für "end of file" (=Dateiende) und dient als Signal, welches eben dies kundtut. Obwohl es in der verbalen Beschreibung des Inputs nicht vorkommt, führen wir es hier ein, um anzudeuten, daß jemand, der die "Buchungsdatei" Satz für Satz liest, mit Sicherheit weiß, ob er das Ende erreicht hat oder nicht. Diese Kenntnis kann nur durch ein irgendwie geartetes Signal (z.B. einen Satz mit einer sonst nicht benutzten Zeichenkombination an der Stelle der Kontonummer) erlangt werden. Man beachte, daß der Name eines elementaren Objekts (im Unterschied zu den Namen komplexer Objekte) in Hochkommata gesetzt wird.

Soll- und Haben-Satz können wieder als Aneinanderreihungen von Objekten modelliert werden:



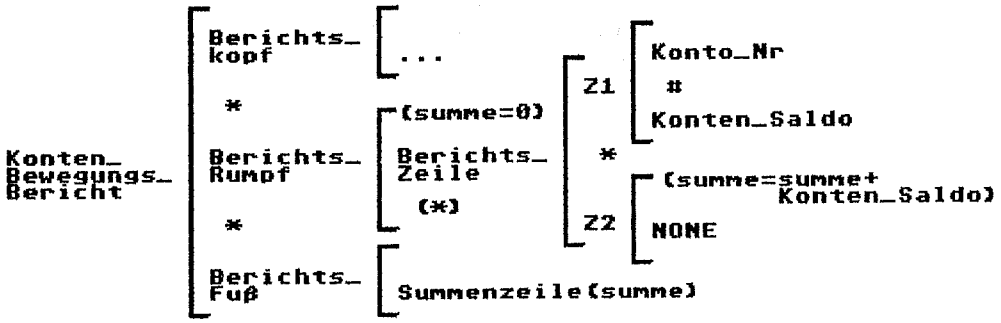
Sie unterscheiden sich natürlich nur in der ersten Komponente der Reihe.

Das nächste Beispiel (aus [EHL]) zeigt einen Datenstrom, dessen Objekte Signale sind, wie sie etwa beim Betrieb einer technischen Anlage auftreten können. Ihr Bediener hat zwei Drucktasten, eine 'start'- und eine 'stop'-Taste. Wenn die Anlage nicht gestartet ist, macht es keinen Sinn, die 'stop'-Taste zu betätigen. Die Betätigung der 'start'-Taste kann einen Fehlstart nach sich ziehen. Die 'start'-Taste muß also unter Umständen mehrmals gedrückt werden, bevor die Maschine in Gang kommt. Und wenn sie läuft, kann sie nur mit der 'stop'-Taste zum Stillstand gebracht werden. Diese "Geschichte des Tastendrückens erzählt" das folgende Diagramm:



Es könnte die Struktur einer Folge von Inputs für ein Programm zur Steuerung der Anlage darstellen.

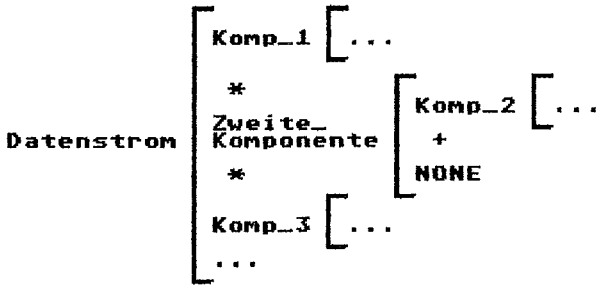
Während das Diagramm "Buchungsdatei" mit dem 'eof' eine Information liefert, die in der Aufgabenstellung gar nicht erwähnt war, unterschlägt das Diagramm "Kontenbewegungs-Bericht" einiges von dem, was uns der Auftraggeber über den gewünschten Bericht mitgeteilt hat: Zum Beispiel die Tatsache, daß die im Berichts-Fuß erscheinende Zahl die Summe der einzelnen Kontensalden zu sein hat. Dies ist offenbar auch eine Eigenschaft, die alle "Kontenbewegungs-Berichte" besitzen müssen, unabhängig davon, aus welchen Inputs der Bericht letztlich gefertigt wird. Sollten wir diese typische Eigenschaft daher nicht ebenfalls im Diagramm (,das ja - wie gesagt - als Definition des Typs von "Kontenbewegungs-Berichts-Objekten" aufgefaßt werden kann,) zum Ausdruck bringen? Die Klammerdiagramm-Technik bietet uns hierfür eine elegante Möglichkeit:



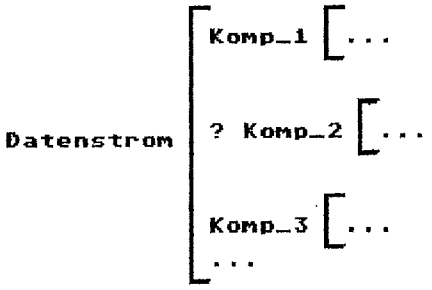
Über die strukturelle (den Aufbau eines "Kontenbewegungs-Berichts" betreffende) Information hinaus gibt dieses Diagramm also auch eine inhaltliche Präzisierung. Es spezifiziert das im Berichtsfuß erscheinende Objekt "Summenzeile", indem es vorschreibt, wie das darin enthaltene Objekt "summe" zu bilden ist. Daß die konkrete Ausprägung des Objekts "Summenzeile" von dem Parameter "summe" abhängt, wird im übrigen durch die übliche Schreibweise "Summenzeile(summe)" verdeutlicht. Man beachte, wie die spezielle Berechnungsvorschrift im Diagramm untergebracht ist: Da wir innerhalb der Klammern ja nur die Komponenten des zu beschreibenden Objekts angeben, müssen die Elemente der algorithmischen Spezifikation (die "Berechnungs-Objekte" sozusagen) an ande-

rer Stelle notiert werden. Die oberen Klammerenden bieten sich hierfür an. Die dort "angehängten" Diagrammteile heißen auch "*Top-Komponenten*". Für eine Objektbeschreibung haben solche Top-Komponenten im allgemeinen einen prädikativen (d.h. im wesentlichen: ebenfalls beschreibenden) Charakter. Sie geben an, was für einzelne Objekt-Komponenten WAHR sein muß. Wird insbesondere ein Objekt sequentiell, also als (zeitliche) Folge oder (räumliche) Reihe beschrieben, so handelt es sich um Aussagen über den *aktuellen Beschreibungszustand.* Im Beispiel muß "summe=0" vor der Beschreibung der Folge von Berichtszeilen gelten. In die Beschreibung einer Berichtszeile andererseits wurde eine Komponente "Z2" mit dem Inhalt "NONE" aufgenommen, für die die Aussage "summe = summe + Konten–Saldo" zutrifft. (Tatsächlich sieht dies eher einer Zuweisung als einer Aussage ähnlich. Mit der Interpretation "Das neue Objekt summe ist gleich dem alten Objekt summe, vermehrt um KontenSaldo" können wir uns hier jedoch leidlich aus der Affäre ziehen.) "NONE" ist eine Art "0" bezüglich der Verknüpfungen von Objektstrukturen; wir nennen es daher den "Nullobjekt-Typ". Ausgesprochen bedeutet er: "Hier steht (bzw. folgt) nichts!" Man könnte also an dieser Stelle völlig darauf verzichten, ihn explizit anzugeben, und die Beschreibung der Komponente Z2 leer lassen. Um Mißverständnisse zu vermeiden, tun wir dies jedoch nicht - ähnlich wie bei der Gestaltung eines technischen Manuals, in dem gelegentlich Seiten mit der Aufschrift "Diese Seite bleibt absichtlich frei" zu finden sind.

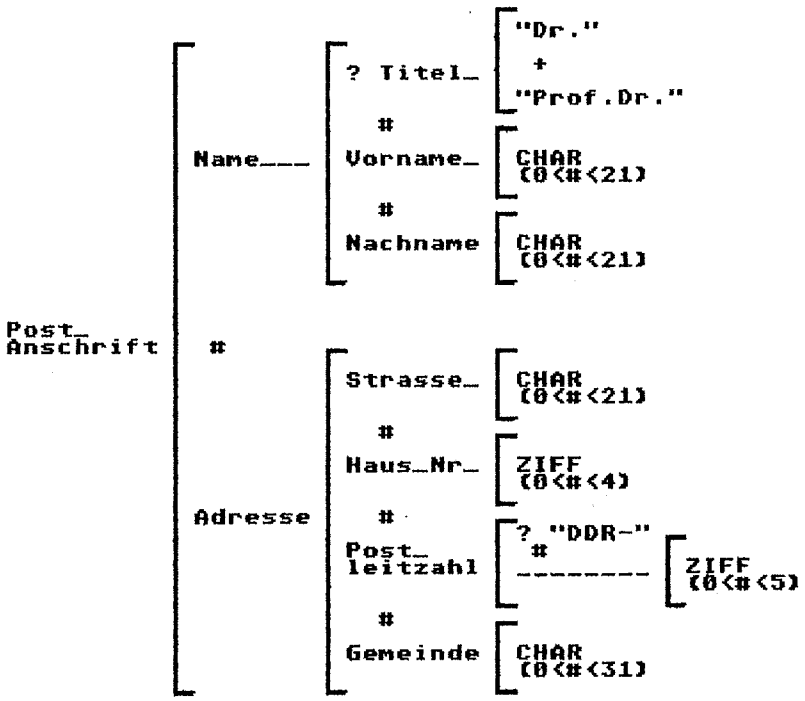
In anderem Zusammenhang freilich ist die explizite Verwendung des Nullobjekt-Typs nicht nur nützlich, sondern auch zwingend. Angenommen, es ist ein Datenstrom mit den Komponenten "Komp_ 1", "Komp_ 2",... zu spezifizieren, von dem man weiß, daß die Komponente "Komp_ 2" auch fehlen darf. Mit Hilfe des Nullobjekt-Typs können wir dann schreiben:



In diesem Datenstrom taucht also nach der Komponente "Komp_1" die Komponente "Komp_2" auf oder nicht. Im Falle, daß "Komp_2" nicht erscheint, sähe man eben nach "Komp_1" die Komponente "Komp_3". "Komp_2" ist - wie man sagt - eine "optionale" oder "Vielleicht-" Komponente. (Vgl. die Bemerkung am Ende von Abschnitt 3.3.2.) Zur Abkürzung verwenden wir hierfür auch die folgende Schreibweise:

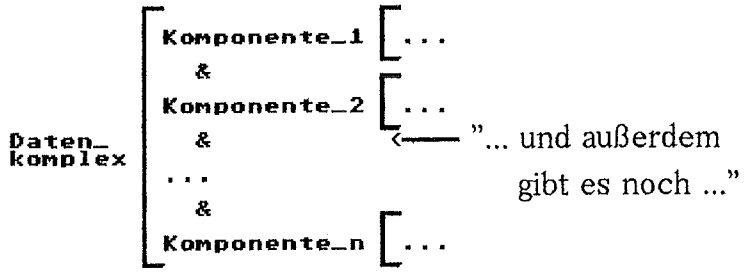


Als weiteres Beispiel mit optionalen Komponenten betrachten wir die Struktur eines Objekttyps "Postanschrift".

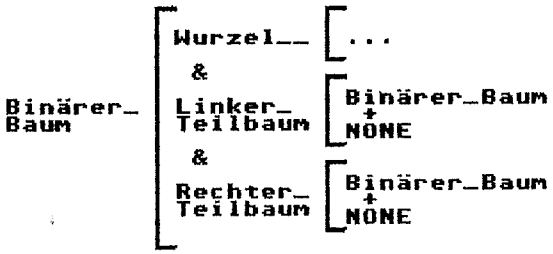


Hier finden wir außerdem Einschränkungen der #-Iteration: Ein Vorname zum Beispiel besteht aus mindestens einem und höchstens 20 Buchstaben, usw.. Entsprechende Einschränkungen kann man selbstverständlich auch für die *-Iteration machen.

Häufig ist es nützlich, sich bei der Beschreibung von Datenobjekten nicht von vornherein auf eine (räumlich oder zeitlich) sequentielle Darstellung festzulegen. Was oft nur interessiert, ist die Tatsache, daß es diese und jene Komponenten gibt, aber nicht ihre Reihenfolge oder Anordnung. So ist es durchaus sinnvoll, im obigen Objekttyp "Postanschrift" den Operator "#" durch "&" zu ersetzen. Mit letzterem fassen wir lediglich verschiedene Komponenten zu einer größeren benennbaren Einheit zusammen, er bildet einen "allgemeinen Datenkomplex":



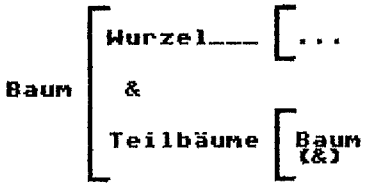
Hierzu ein Beispiel: Ein binärer Baum besteht aus einem "Wurzelknoten", aus einem linken Teilbaum und aus einem rechten Teilbaum. Die Teilbäume sind entweder selbst binäre Bäume oder sie sind vom Nullobjekt-Typ. Das dieser Charakterisierung entsprechende Klammer-Diagramm ist:



Dies ist gleichzeitig ein Beispiel für eine rekursive Beschreibung eines Objekttyps.

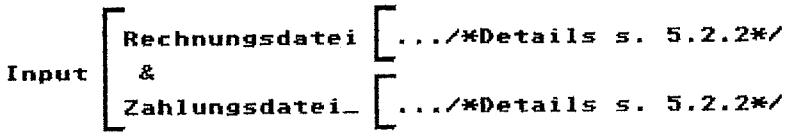
Wie mit "*" und "#", so können auch mit "&" durch Zusammenfassung gleichartiger Komponenten Iterations-Objekte gebildet werden:

"(&)" besagt einfach, daß "Iter_Komponente" mehrfach (möglicherweise auch keinmal) vorliegt. Ein nicht notwendig binärer Baum hätte damit die folgende Gestalt:



Man beachte, daß ein Iterationsobjekt mit unspezifiziertem Anzahlbereich das Nullobjekt automatisch mit umfaßt. Daher ist es nicht notwendig, in der rekursiven Beschreibung des Typs "Baum" den Nullobjekt-Typ explizit zu benutzen.

Auf Datenströme angewandt ergibt die &-Operation einen Komplex von nebeneinander ("parallel") fließenden Datenströmen. Beispielsweise werden wir es in Abschnitt 5.2.2 mit der Aufgabe zu tun haben, Output mit Hilfe der beiden parallelen Input-Ströme "Rechnungsdatei" und "Zahlungsdatei" zu erzeugen:



Auf die Frage, ob es zweckmäßiger sei, die Struktur von Datenobjekten (wie gesagt: ihren "Typ") so oder so, als Strom, Reihe oder Komplex zu beschreiben, gibt es keine rezeptartige Antwort. Prinzipiell wollen wir - um mit dem Bild des sich ja in der Zeit abspielenden Zusammensetzens handfester Gegenstände in Einklang zu bleiben - der Beschreibung als "Strom" den Vorzug geben. Andererseits empfiehlt es sich, bei Satz- oder Zeichenketten-artigen Objekten, die in einem Datenstrom auftreten können (siehe "Buchungsdatei"), diesen ihren Charakter auch deutlich zu machen. Sie werden zunächst einmal "als Ganzes" aufgefaßt. Erst wenn es darum geht, ihre Herstellung, Be- oder Verarbeitung im Detail zu klären, müssen ihre Komponenten in eine zeitliche Folge gebracht werden. Dies gilt auch für als "allgemeine Datenkomplexe" beschriebene Objekte.

Weitere Notationselemente werden wir bei Bedarf in den folgenden Abschnitten einführen und erläutern. Es ist in diesem Zusammenhang zu bemerken, daß wir uns hier (wie im übrigen auch sonst in diesem Buch) mit Notationen für den Entwurf von Programmen nicht um ihrer selbst willen beschäftigen. Sie interessieren uns nur insoweit, als man mit ihnen bestimmte Gedanken bequem formulieren und bestimmte Vorgehensweisen angemessen unterstützen kann. Es ist freilich hinzuzufügen, daß Entwurfs-Notationen dann präzise definiert werden müssen, wenn auf ihrer Grundlage *Software-Werkzeuge* ("Tools") entwickelt werden sollen, welche beispielsweise die Entwurfs-Dokumentation oder den Übergang vom Entwurf zur (lauffähigen) Implementierung unterstützen. Da die Behandlung dieses Aspekts den Rahmen dieses Buches sprengen würde, müssen wir hierzu auf weiterführende Literatur (z.B. [ENS]) verweisen.

5.1.2 Produktion von Datenobjekten

Bei seiner alltäglichen Detailarbeit hat es der Programmierer sehr häufig mit jenen Aufgaben vom "Produktionstyp" zu tun, die wir in der Einleitung dieses Kapitels charakterisiert haben. Dort gaben wir auch zwei sehr einfache Beispiele an, denen wir uns in diesem Abschnitt zuwenden wollen.

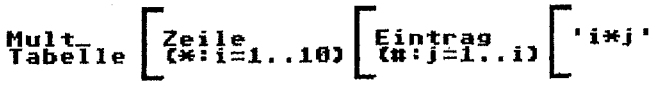
Beispiel (i):

Die Multiplikations-Tabelle, das erwähnten wir bereits, ist eine Folge von Zeilen. Eine Zeile ist eine Reihe von Einträgen. Jeder Eintrag ist eine ganze Zahl. Als

Klammerdiagramm stellt sich diese Beschreibung so dar:

Natürlich gibt es beliebig viele Objekte mit dieser Struktur, und nicht alle diese Objekte lassen sich tatsächlich als Multiplikationstabelle interpretieren. Da wir (zumindest vorerst) genau ein bestimmtes Objekt produzieren wollen, müssen wir die Beschreibung auf einen Objekttyp verengen, dessen einzige Ausprägung das gewünschte Objekt ist. Eine erste Maßnahme ist die Fixierung der Zeilenanzahl auf 10:

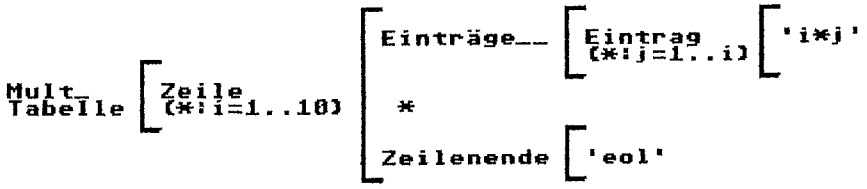
Welche ganze Zahlen in einer Zeile stehen, hängt nun allerdings davon ab, wo in der Folge sich diese Zeile befindet. Wir müssen also erstens einen "Index" zur Numerierung der Zeilen einführen. Ganz entsprechend sind die ganzen Zahlen, die in einer Zeile vorkommen, durch ihre Stellungen innerhalb der Zeile gegeben. Zweitens benötigen wir daher einen "Index" zur Fixierung der Position eines Eintrags in der Reihe. Und schließlich liefert uns die Formulierung der Aufgabe die Information, daß eine Zeile gerade soviele Einträge enthalten soll, als der Index der Zeile angibt. Diese - zunächst verbale - Präzisierung wird im folgenden Diagramm mit Hilfe einiger zusätzlicher selbsterklärender Notationen formal dokumentiert:



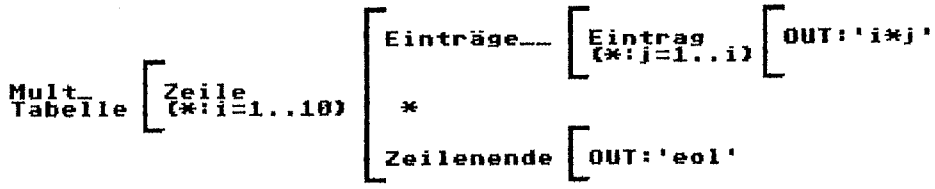
"Ganze–Zahl" haben wir dabei bereits durch den "wahren Wert", das elementare Objekt 'i*j', ersetzt. "i" und "j" sind hier natürlich keine Variablennamen im programmtechnischen Sinn, denn von einem Programm war in diesem Zusammenhang ja bisher noch gar nicht die Rede. Sie sind vielmehr als "Wertbezeichner" aufzufassen, etwa in dem Sinne, wie wir das im vierten Kapitel für (durch Quantoren) gebundene Bezeichner getan haben. Dies schließt freilich andererseits nicht aus, daß - und dies wird die Regel sein - aus den in der Spezifikation eines Objekts auftauchenden Wertbezeichnern Variablennamen werden, wenn es an die Umsetzung einer Objektbeschreibung in ein Programm geht. (Im Lichte dieser Bemerkung betrachte man im übrigen auch nochmals das zweite Diagramm "Kontenbewegungsbericht" im vorangegangenen Abschnitt, wo wir die Wertbezeichner "summe" und "Konten _Saldo" verwendet haben!)

Um aus dem letzten Diagramm eine "Produktionsanweisung" für das gewünschte Bild der Tabelle auf dem Bildschirm zu machen, ist nur noch ein sehr kleiner Schritt notwendig: Wir lösen die Beschreibung einer Zeile als "räumliches Ne-

beneinander" auf in die zeitliche Abfolge der in dieser Zeile stehenden Einträge und eines speziellen Zeilenende-Signals 'eol':



Wir nehmen nun an, daß wir über eine Operation - nennen wir sie einfach "OUT:" - verfügen, die ein elementares Objekt (hier eine ganze Zahl oder 'eol') an das Ausgabemedium (in diesem Fall der Bildschirm) sendet. Wenn wir von dieser Operation ferner voraussetzen, daß sie nach Ausgabe einer ganzen Zahl die Verschiebung der Schreibstelle nach rechts und nach Ausgabe eines 'eol' einen Zeilenwechsel bewirkt, so ergibt sich aus der obigen Objektbeschreibung die folgende Programmbeschreibung:



Eine (Beinahe-)Realisierung als MODULA-2 - Prozedur erhält man sofort mit Hilfe der in Abschnitt 3.3.2 angegebenen Entsprechungen. Dabei liegt die Umsetzung der oben eingeführten Iterations-Notation (z.B. "(*: i=1..10)") in ein "FOR ... DO ... END"-Konstrukt offenbar auf der Hand. Man beachte, daß - wie angekündigt - die Wertbezeichner "i" und "j" nun als Variablennamen deklariert sind.

```
PROCEDURE MultTabelle;
CONST EolC = ...;
VAR i,j: CARDINAL;
BEGIN
  FOR i:=1 TO 10 DO  (*Zeile*)
   (*Einträge*)
   FOR j:=1 TO i DO  (*Eintrag*)
    (* OUT:'i*j' *)
   END (*FOR*)
   (*Zeilenende*)
   (* OUT:'eol' *)
  END (*FOR*)
END MultTabelle;
```

Wir gebrauchen für diese Realisierung den Zusatz "Beinahe-", weil die wichtigsten Anweisungen hier nur als Kommentare erscheinen. Es sind jene Anweisun-

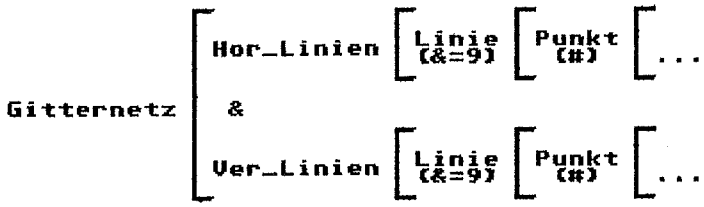
gen, durch welche die Elementarobjekte ausgegeben werden, die letztlich die (sichtbare) Tabelle ausmachen.

Nach dem, was wir oben für "OUT:" gefordert haben, sollte es andererseits kein Problem sein, eine jeweils geeignete Prozedur zur Ausführung dieser Operationen zu finden. Zwar gehören Ein- /Ausgabe-Anweisungen nicht zum Sprachumfang von MODULA-2, doch bieten die meisten Implementierungen eine "Standard-Bibliothek" "InOut" an, welche auch Prozeduren enthält, die ähnlich wirken wie die PASCAL-Standard-Prozeduren "write" und "writeln".

Die endgültige Realisierung ist allerdings auch insoweit offen, als man nun für den Effekt von (*OUT:'i*j'*) und (*OUT:'eol'*) wesentlich allgemeinere Forderungen aufstellen kann als oben. Beispielsweise könnten wir uns die Freiheit nehmen, diese Kommentare als Anweisungen an einen weiteren "Mitarbeiter" im "Betrieb zur Herstellung der Multiplikationstabelle" zu interpretieren. Mit "OUT:'i*j'" würde dieser "Mitarbeiter" aufgefordert, einen Eintrag zu machen und mit "OUT:'eol'" würde ihm gesagt, eine Zeile komplett sichtbar zu machen. Welche Hilfsmittel ("Ressourcen") und welche Prozeduren diesem "Mitarbeiter" dafür zur Verfügung stehen, ist dann zunächst eine zweitrangige Frage, auf die wir an dieser Stelle freilich noch keine Antwort geben müssen. (Allerdings werden Überlegungen, welche die hier angedeutete Idee zur Strukturierung von Programmen beträchtlich ausbauen und vertiefen, im Mittelpunkt des folgenden Kapitels stehen.)

Beispiel (ii):

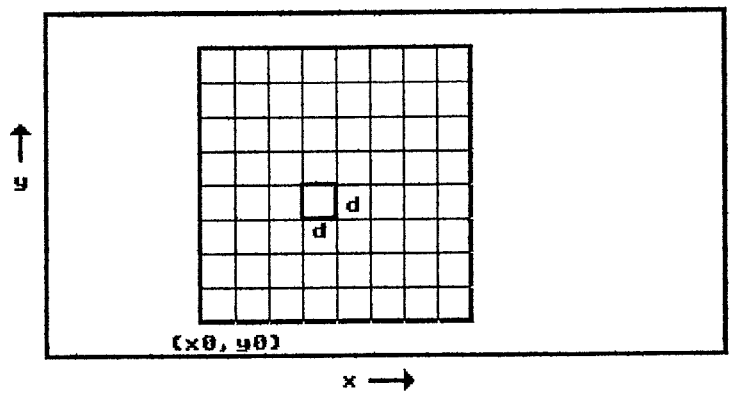
Zur Lösung der zweiten Beispielaufgabe gehen wir von einer Beschreibung des Gitternetzes als "Komplex von horizontalen und vertikalen Linien" aus. Offenbar kommt es ja für das Verständnis des Objekts "Gitternetz" nicht darauf an, spezielle Reihenfolgen für diese Teilobjekte festzulegen.



Das gleiche gilt für die Linien selbst: Sie sind "einfach da", ob sie in dieser oder jener Reihenfolge betrachtet werden, ist unerheblich. Deshalb modellieren wir "Hor–Linien" und "Ver–Linien" jeweils als (&)-Iterationen. Jede Linie ist am zweckmäßigsten als eine Reihe von Punkten darstellbar, also als (#)-Iteration. Auch diese Beschreibung wird unmittelbar zu einer Prozedur für die Herstellung eines solchen Netzes führen.

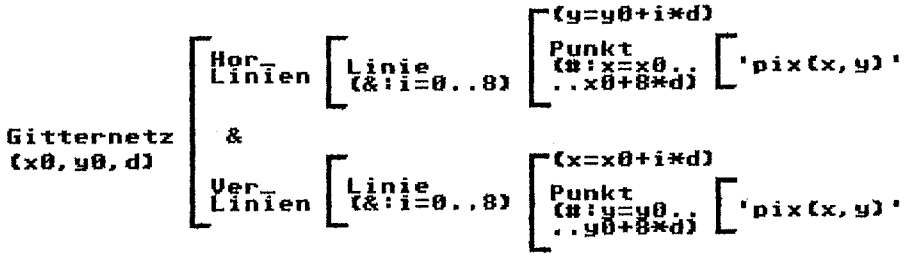
Um sie zu vervollständigen, müssen wir natürlich genauere Angaben zur Lage und Größe des Gitternetzes machen. Die Lage mag etwa durch die Bildschirm-

koordinaten des linken unteren Eckpunktes und die Größe durch die Seitenlänge (gemessen durch die Anzahl der Bildschirmpunkte (Pixel)) einer (quadratischen!) Gittermasche gegeben sein:



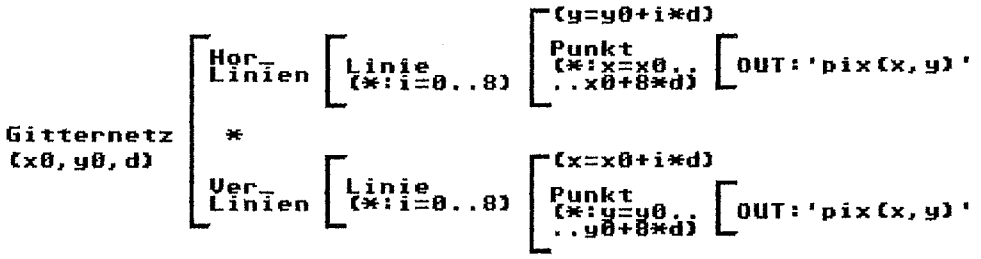
Diese Angaben reichen aus, um Lage und Länge der einzelnen Linien zu spezifizieren. Voraussetzung ist dafür allerdings außerdem, daß die einzelnen Linien identifizierbar sind. Wir können dann zum Beispiel bestimmen, daß die horizontale Linie "Nummer ..." an einem "Punkt ..." beginnt und "so-und-so" lang ist. Wenn wir für x0, y0 und d keine konkreten Werte vorgeben, sondern weiterhin mit diesen Wertbezeichnern arbeiten, so haben wir damit eine Beschreibung des Typs aller quadratischen (8*8)-Gitternetze gewonnen. Eine spezielle Ausprägung dieses Typs ist dann durch die Wahl konkreter Werte für x0, y0 und d gegeben.

Es liegt daher nahe, das folgende Diagramm, welches die soeben angestellten Überlegungen wiedergibt, als Definition eines *parametrierten Objekttyps* aufzufassen:



'pix(x,y)' ist dabei ein elementares Objekt, eben der "Punkt" auf dem Bildschirm an der Stelle (x,y).

Die Überführung dieser Struktur-Beschreibung in eine *Prozeß-Beschreibung* ist nun wiederum nahezu trivial. Es sind lediglich die Operatoren "&" und "#" in geeigneter Weise durch das zeitliche Nacheinander "*" zu ersetzen, und vor die elementaren Objekte schreiben wir "OUT:":



Tatsächlich handelt es sich hier um die Beschreibung einer ganzen Klasse von Prozessen (eines "Prozeßtyps"!). Ein einzelner Prozeß ist - wie ein einzelnes Objekt - durch die Werte der Parameter bestimmt. Er wird duch den Aufruf der folgenden MODULA-2 - Prozedur (deren Herleitung aus der Prozeßbeschreibung inzwischen keinerlei Schwierigkeiten mehr bereiten dürfte) ausgelöst:

```
PROCEDURE Gitternetz(x0,y0,d: CARDINAL);
VAR x,y,i: CARDINAL;
BEGIN
 (*Hor–Linien*)
 FOR i:=0 TO 8 DO (*Linie*)
  y:=y0 + i*d;
  FOR x:=x0 TO x0+8*d DO (*Punkt*)
   (* OUT:'pix(x,y)' *)
  END (*FOR*)
 END (*FOR*);
 (*Ver–Linien*)
 FOR i:=0 TO 8 DO (*Linie*)
  x:=x0 + i*d;
  FOR y:=y0 TO y0+8*d DO (*Punkt*)
   (* OUT:'pix(x,y)' *)
  END (*FOR*)
 END (*FOR*)
END Gitternetz;
```

Auf die konkrete Realisierung der Operation "OUT:'pix(x,y)'" gehen wir - wie im ersten Beispiel - nicht näher ein.

Eine wichtige Beobachtung sollten wir freilich nicht unterschlagen. Sie betrifft die Ersetzung des &-Operators durch "*". Was wäre gewesen, so können wir fragen, wenn das "&", welches in der Beschreibung des Objekttyps "Gitternetz" die Komponenten "Hor_Linien" und "Ver_Linien" miteinander verknüpft, beibehalten und nicht durch "*" ersetzt worden wäre? Die Antwort ist einfach: Wir hätten damit zum Ausdruck gebracht, daß sich ein Prozeß zur Erzeugung des Gitternetzes in zwei voneinander unabhängige Teilprozesse aufspalten läßt: in einen Prozeß, der die horizontalen Linien und in einen Prozeß, der die vertikalen

Linien hervorbringt. Diese beiden Prozesse könnten gleichzeitig oder - wie man auch sagt - *parallel* zueinander ablaufen. (Gelegentlich verwendet man hierfür auch den Begriff "*nebenläufig*".) Wenn wir über eine Maschine mit zwei Prozessoren verfügten, so könnten wir diesen je einen der beiden Prozesse zuordnen (und die Arbeit würde in der halben Zeit erledigt). Die Realisierung eines derartigen "Fertigungsplans" durch eine MODULA-2 - Prozedur würde uns allerdings einige Schwierigkeiten bereiten, da wir (noch) nicht wissen, ob (und wenn ja wie) der Ablauf paralleler Prozesse mit Hilfe von MODULA-2 - Programmen steuerbar ist. Auf Möglichkeiten hierfür werden wir erst gegen Ende von Kapitel 6 zu sprechen kommen.

Die beiden in diesem Abschnitt behandelten Aufgaben sind sehr einfache Beispiele dafür, daß sich die Arbeit des Programmierens mitunter im wesentlichen auf den Entwurf von Datenstrukturen (bzw. "Datenobjekten") reduzieren läßt. Interessant ist es, unter diesem Gesichtspunkt die "Korrektheitsfrage" neu zu stellen. Wie ist die Behauptung, die oben entwickelten kleinen Programme seien korrekt, zu verstehen? Da sich die Programme gewissermaßen automatisch aus den Objektbeschreibungen ergeben, sind sie offenbar genau dann korrekt, wenn diese Beschreibungen auch wirklich die gewünschten Objekte treffen. Wie aber ließe sich die Korrektheit einer Objektbeschreibung nachweisen? Hierzu wäre eine von der letztlich erstellten Beschreibung des Objekts unabhängige Spezifikation notwendig. In der Tat begannen unsere Beispiele mit solchen Spezifikationen, doch sie waren rein verbal, fast umgangssprachlich. Für einen exakten Korrektheitsnachweis lassen sie sich mit Sicherheit nicht gebrauchen. Mit den Diagrammen haben wir sie präzisiert, formalisiert; aber die Diagramme waren andererseits auch bereits die Objektbeschreibungen! Natürlich sind andere Formalisierungen vorstellbar, doch besteht für diese dann genau das gleiche Problem: zu entscheiden, ob sie das ausdrücken, was wir wollen. Dieser "Teufelskreis" legt es nahe, nicht vorherige Objekt-Spezifikationen zu verlangen, sondern (z.B.) die diagrammatischen Beschreibungen, die wir erarbeitet haben, eben als die nicht weiter zu rechfertigenden Spezifikationen der zu produzierenden Objekte aufzufassen! Sie bilden zugleich die Spezifikationen der die Objekte produzierenden Programme, und diese werden - wie bereits bemerkt - qua Konstruktion korrekt.

5.1.3 Analyse von Datenobjekten

Wenden wir uns nun der zweiten Klasse von Aufgaben, den Aufgaben vom Analysetyp zu. Auch zu deren Illustration werden wir die in der Einleitung des Kapitels erwähnten Beispiele ((iii) und (iv)) heranziehen. Für den Entwickler von Compilern sind solche Aufgaben Standard-Probleme: Sogenannte "Mehr-Pass-Compiler" beginnen ihre Arbeit häufig damit, zunächst einmal zu überprüfen, ob der von ihnen zu übersetzende Programmtext überhaupt die syntaktischen Re-

geln der Sprache einhält. Da der Bau von Compilern ein "klassisches" Betätigungsfeld des Informatikers ist, kann es nicht verwundern, daß auch zur Erledigung dieses Standard-Problems "Syntaxanalyse" viele formale Ansätze entstanden sind. Natürlich kann es in diesem Buch nicht unser Ziel sein, hierüber (Grammatiken, Automaten, Syntaxgraphen etc.) einen Überblick zu geben. Vielmehr kommt es uns darauf an, die für den Programmentwurf wesentliche Idee möglichst plastisch hervorzuheben: sich nämlich bei der Erstellung eines Analyse-Programms an der Struktur der zu analysierenden Objekte zu orientieren. Dazu ist der für dieses Kapitel gewählte Klammerdiagramm-Formalismus ohne weiteres geeignet.

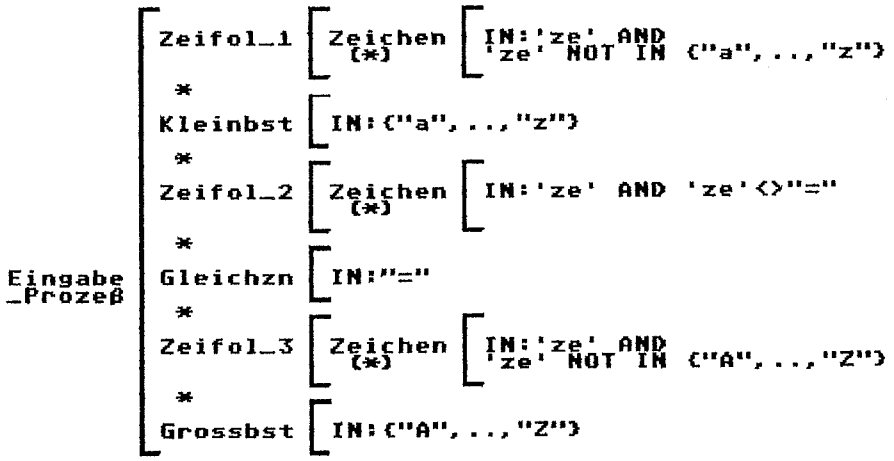
```
Eingabe [ Kleinbst [ {"a",..,"z"}
          *
          Gleichzn [ "="
          *
          Grossbst [ {"A",..,"Z"}
```

Betrachten wir also zunächst *Beispiel (iii)*. Die Struktur der Zeichenfolge, um die es hier geht, ist - auf den ersten Blick - sehr einfach. Sie wird offenbar durch das nebenstehende Klammer-Diagramm beschrieben.

Natürlich müssen wir, um die Aufgabe vollständig zu definieren, irgendeine Form der Bekanntgabe des Ergebnisses der Analyse festlegen. Wir machen es uns dabei leicht, indem wir von dem zu entwickelnden Programmstück weder eine sichtbare Reaktion noch sonst irgendeine Zustandsänderung fordern. Wir wollen also lediglich diese drei Zeichen (in der gewünschten Reihenfolge) "einsammeln" und nichts weiter. Bei genauerem Studium der Aufgabenstellung (und eventuell nach einer Rückfrage beim Aufgabensteller) fällt uns freilich auf, daß irgendwelche Tasten betätigt werden können, die gar nichts zu der gewünschten Zeichenfolge beitragen. Aus diesem Grund sind wir gezwungen, die Menge der möglichen (Eingabe-)Zeichenfolgen zu erweitern, da beispielsweise vor Eingabe eines Kleinbuchstabens beliebige andere Tasten beliebig oft gedrückt werden dürfen. Die Struktur des Inputs wird daher durch das folgende Klammerdiagramm dargestellt:

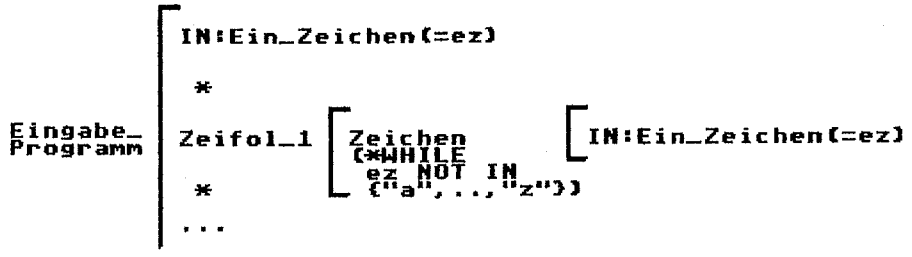
```
Eingabe [ Zeifol_1 [ Zeichen (*) [ 'ze' AND 'ze' NOT IN {"a",..,"z"}
          *
          Kleinbst [ {"a",..,"z"}
          *
          Zeifol_2 [ Zeichen (*) [ 'ze' AND 'ze'<>"="
          *
          Gleichzn [ "="
          *
          Zeifol_3 [ Zeichen (*) [ 'ze' AND 'ze' NOT IN {"A",..,"Z"}
          *
          Grossbst [ {"A",..,"Z"}
```

Eine Beschreibung des Prozesses der Entgegennahme dieser Zeichenfolge ergibt sich nun einfach dadurch, daß wir vor jedem elementaren Objekt den Namen einer Eingabe-Operation, nennen wir sie "IN:", notieren:



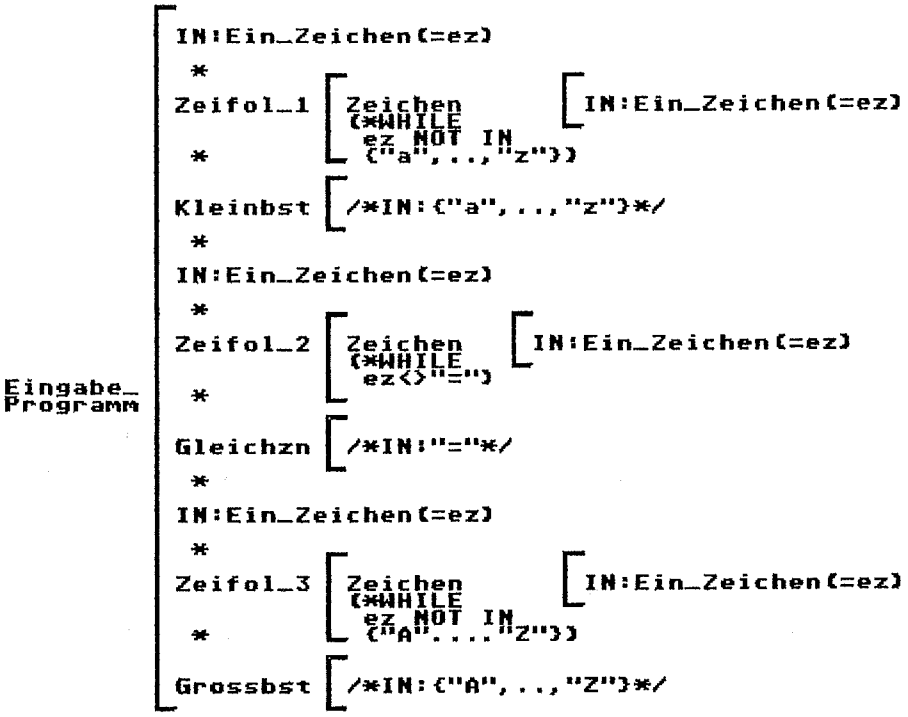
Damit wird die Abfolge von Objekten zu einer Abfolge von Aktionen. Es ist das, was wir beobachten, wenn wir dem Benutzer des Programms, welches diesen Prozeß steuert, zusehen.

Wie aber gelangen wir, das ist nun die Frage, zu einer Beschreibung dieses Programms? Die obige Prozeß-Beschreibung sollte hierfür die Grundlage sein. Entscheidend ist wohl, daß das Programm für einen "Prozeß-Fortschritt" sorgt. Es muß erkennen können, wann beispielsweise die Zeichenfolge "Zeifol_1" vor dem Kleinbuchstaben beeendet ist. Dies ist natürlich dann der Fall, wenn ein Kleinbuchstabe eingegeben wird. Da andererseits "Zeifol_1" auch leer sein kann, muß das Programm zunächst die Entgegennahme eines Zeichens vorsehen, um dann solange den Teilprozeß "Zeifol_1" aufrechtzuerhalten, als kein Kleinbuchstabe eingegeben wird. Im Falle, daß das erste gelesene Zeichen ein Kleinbuchstabe war, wird dieser Teilprozeß also überhaupt nicht stattfinden. Ein erstes Fragment der gesuchten Programm-Beschreibung ist somit:



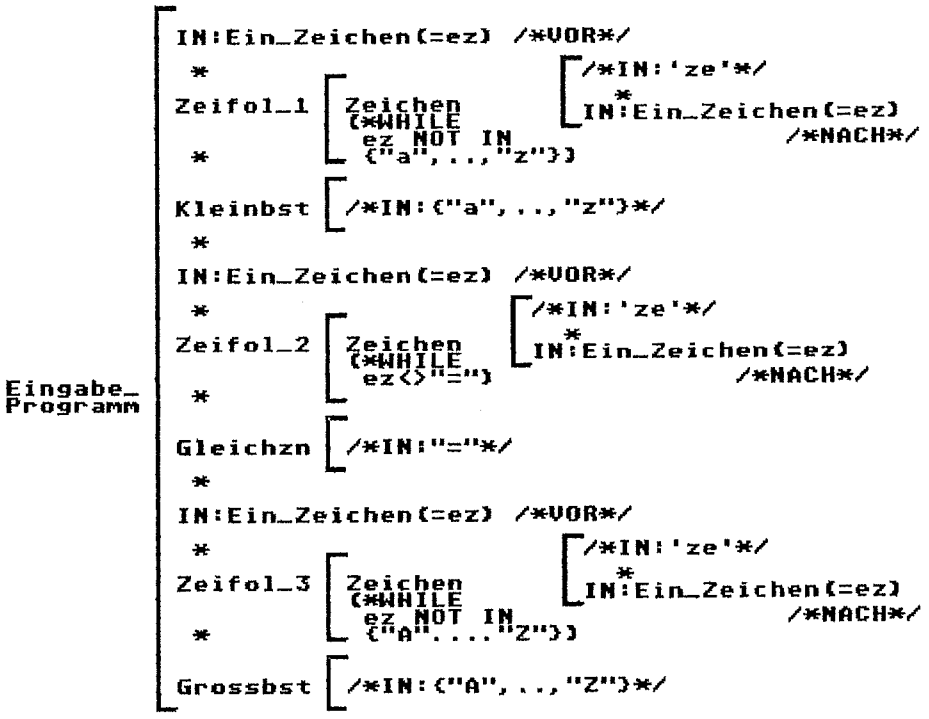
Die Notation "...Ein_Zeichen(=ez)" erweist sich hier als notwendig, um im weiteren Verlauf von dem mit "IN:" herbeigeschafften Objekt reden zu können. Ausführlich lautet "IN:Ein_Zeichen(=ez)" etwa so: "Es wird ein Zeichen gelesen, das wir im folgenden "ez" nennen". Was geschieht nun weiter? Offenbar macht es

keinen Sinn, die Aktion "IN:["a",...,"z"]" aus der Prozeß-Beschreibung in die Programm-Beschreibung zu übernehmen, da ja nach dem Austritt aus "Zeifol_1" der Kleinbuchstabe bereits gelesen ist. Was nun folgen muß, ist das entsprechende Verfahren bezogen auf "Zeifol_2" und das Gleichheitszeichen. Trotzdem lassen wir "IN:["a",...,"z"]" hier nicht völlig verschwinden: Um die Herkunft der Programm-Beschreibung aus der Prozeß-Beschreibung ganz deutlich zu machen, setzen wir es in Kommentarklammern "/* ... */". Die vollständige Programm-Beschreibung lautet nun:

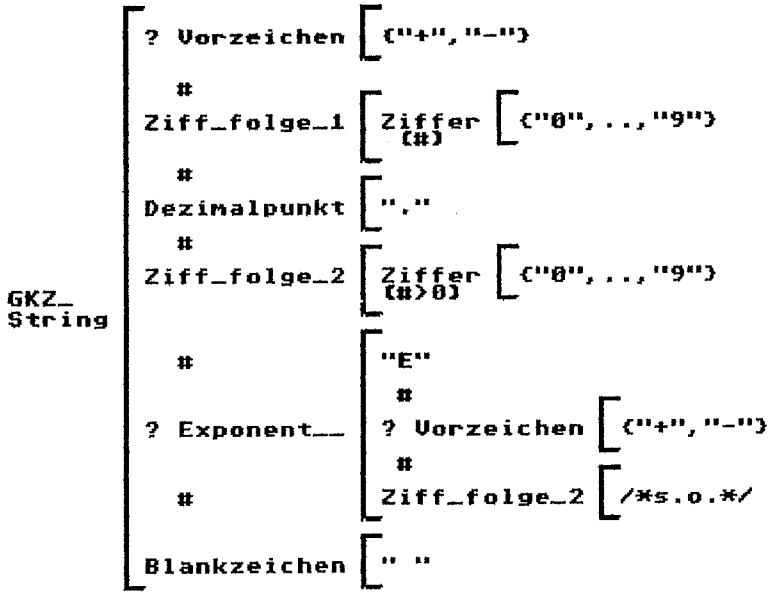


(Auf eine Umsetzung in ein MODULA-2 - Programmstück sei an dieser Stelle verzichtet. Der Leser möge sich daran selbst versuchen.)

Die Technik, derer wir uns hier bedient haben, um aus der Prozeß-Beschreibung eine Programm-Beschreibung herzustellen, ist das sogenannte "*Vorweg-Lesen*". Mit dem "Vorweg-Lesen" ist ein "Nach-Lesen" verbunden: Tatsächlich haben wir aus dem Teilprozeß "Zeifol_1" die erste "IN"-Operation vorgezogen und genau genommen müssen wir dies dadurch kenntlich machen, daß wir diese Operation innerhalb "Zeifol_1" in Kommentarklammern setzen. Die "IN-Operation" nämlich, die in der obigen Programm-Beschreibung in "Zeifol_1" iteriert wird, liest in Wirklichkeit immer ein Zeichen nach. Um diesen Sachverhalt (der selbstverständlich für die übrigen Programmteile auch zutrifft) ins Bewußtsein zu rükken, geben wir im folgenden eine erweiterte Fassung der Programm-Beschreibung wieder, in der die "Vorweg-Lese"- und "Nach-Lese"-Aktionen markiert sind:



Zumindest für dieses triviale Beispiel ist die Technik des "Vorweg-Lesens" gleichbedeutend mit einer einfachen Transformation des Diagramms, welches die Prozeß-Beschreibung darstellt. Wir werden später sehen, daß sie sich auch in schwieriger erscheinenden Situationen systematisch einsetzen läßt.



Auch die als *Beispiel (iv)* formulierte Aufgabe ist noch zu vag, als daß wir nicht klarstellen müßten, was wir eigentlich wollen: Wir fordern hier eine MO-

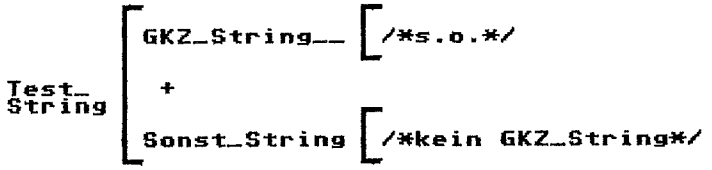
DULA-2 - Funktionsprozedur "IstGKZ", der eine Zeichenkette als Parameter übergeben wird. Falls diese Zeichenkette eine Gleitkommazahl repräsentiert, soll "IstGKZ" den Wert "TRUE" abliefern, andernfalls den Wert "FALSE". Der Prozedurkopf ist also:

"PROCEDURE IstGKZ(s:ARRAY OF CHAR):BOOLEAN"

Die Struktur einer, eine Gleitkommazahl repräsentierenden Zeichenkette wird durch das obige Klammerdiagramm "GKZ_String" beschrieben.

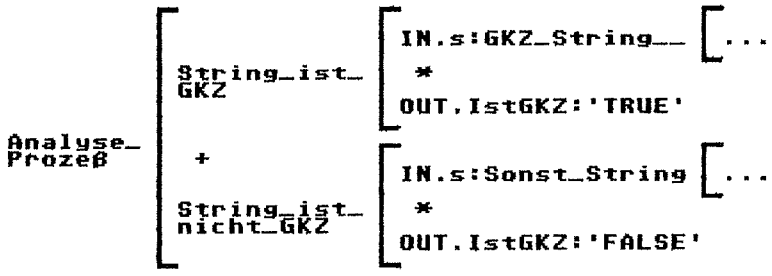
(Man beachte, daß diese Struktur von der in MODULA-2 für REAL-Literale festgelegten Syntax leicht abweicht! Aus diesem Grund haben wir die Bezeichnung "Gleitkommazahl" gewählt.)

Nun erhält die Prozedur "IstGKZ" ja keineswegs immer ein Objekt mit dieser Struktur (sonst wäre sie überflüssig), vielmehr wird sie es mit einem beliebigen "String-Objekt" zu tun haben, welches entweder eine Gleitkommazahl darstellt oder nicht. Der Input unserer Prozedur ist folglich zweckmäßigerweise als Selektions-Objekt zu modellieren:



Ein solches Objekt ist Inhalt der Variablen "s". Der Prozeß liest also (bzw. macht seine Inputs) ein *Zeichen nach dem anderen* aus der Variablen "s". Zur Veranschaulichung dieses Sachverhalts qualifizieren wir den Namen "IN:" der (sequentiellen) Leseoperation mit dem Namen "s" des Ortes, von dem gelesen wird, und schreiben im folgenden "IN.s: ...". Der Prozeß soll auch einen Output produzieren, und zwar entweder "TRUE" oder "FALSE". Er soll einen dieser Werte an einem Ort ablegen, den wir - naheliegenderweise wie die Funktionsprozedur selbst - durch "IstGKZ" bezeichnen können. Es macht daher Sinn, auch den Namen "OUT:" der Ausgabeoperation durch Anhängen des Namens des Ortes, an dem die Ausgabe erscheinen soll, zu ergänzen.

Da wir fordern, daß nach dem Lesen eines "GKZ_String" der Output "TRUE" und nach dem Lesen eines "Sonst_String" der Output "FALSE" getätigt wird, muß die gewünschte Prozedur den folgenden Prozeß bewirken:



Wieder stehen wir vor dem gleichen Problem wie oben, nämlich aus der Prozeßbeschreibung eine Beschreibung des Programms herzuleiten. Die Technik des "Vorweglesens" führt hier offenbar zunächst nicht weiter, da wir ja dazu bereits in der Lage sein müßten, genau jene Entscheidung zu treffen, welche durch die Analyse erst geliefert werden soll. Einen Hinweis darauf, wie hier vorzugehen ist, erhalten wir aber, wenn wir uns klarmachen, wie die Alternative "String_ist_nicht_GKZ" zustandekommen könnte: Steht zum Beispiel in "s" die Zeichenkette "0.12+7", so sind wir bis einschließlich des Zeichens "2" nicht sicher, ob diese Alternative zutrifft oder nicht. Nach dem Lesen des Zeichens "+" aber ist klar, daß es sich hier nicht um einen "GKZ_String" handelt. Bis zur "2" kann die Steuerung des Prozesses also *davon ausgehen*, daß es sich bei dem zu analysierenden Objekt um einen "GKZ_String" handelt. Dann allerdings muß sie *zugeben*, daß sie sich geirrt hat, und sie muß den Output "FALSE" erzeugen. Die Alternative "String_ist_nicht_GKZ" ist also immer dann zu beobachten, wenn beim zeichenweisen Lesen der zu untersuchenden Zeichenkette die *Annahme*, daß es sich um einen "GKZ–String" handele, nicht aufrechterhalten werden konnte.

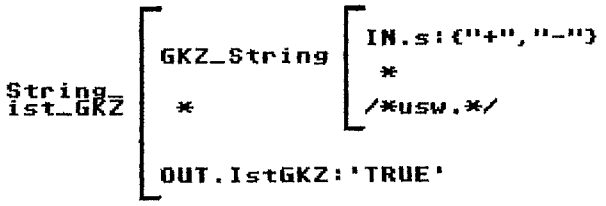
In die Programmbeschreibung führen wir dieses "*Annehmen*" und "*Zugeben*" (daß die Annahme falsch war) durch die reservierten Wörter "POSIT" und "ADMIT" ein:

```
Analyse_
Programm  [ POSIT
            String_ist_
            GKZ          [ ...
                           ? Annahme falsch;
                             QUIT POSIT
                           ...
          +
          ADMIT
            String_ist_
            nicht_GKZ    [ ...
```

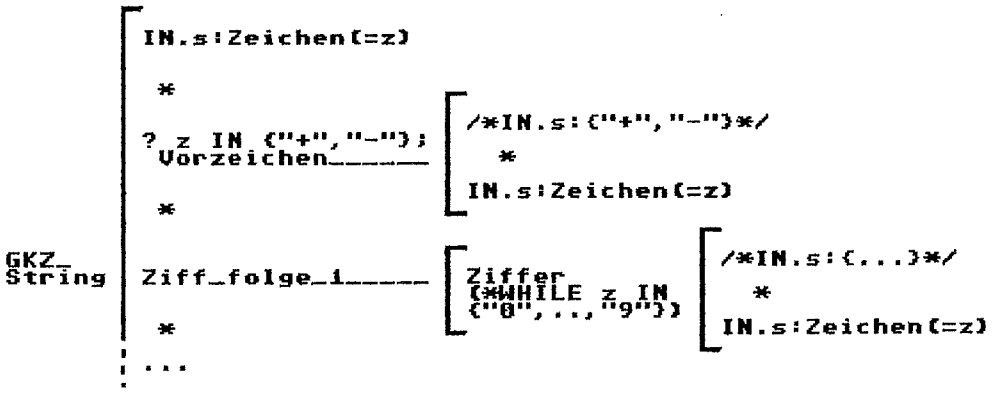
Die Bedeutung ist die folgende: Zunächst werden die Aktionen im POSIT-Zweig ausgeführt. Gerät man dabei in einen Widerspruch zu jener Annahme, die dem POSIT-Zweig entspricht (in unserem Fall also, daß die Zeichenkette ein "GKZ_String" sei), so wird dieser Zweig verlassen und mit den im ADMIT-Zweig auszuführenden Aktionen begonnen. In unserem speziellen Beispiel haben nun die Input-Aktionen, die dem ADMIT-Zweig des Prozesses entsprechen, bereits stattgefunden, weitere Inputs müssen nicht erfolgen. In der Programmbeschreibung wird daher die Input-Aktion "IN.s:Sonst_String..." entfallen beziehungsweise nur noch als Kommentar auftauchen (ähnlich wie im vorigen Beispiel vorgezogene Leseaktionen an ihrer ursprünglichen Stelle in Kommentare verpackt wurden, um die Herkunft der Programmbeschreibung aus der Prozeßbeschreibung kenntlich zu machen).

```
Analyse_
Programm  [ POSIT
            String_ist_
            GKZ          [ ...
                           ? Annahme falsch;
                             QUIT POSIT
                           ...
          +
          ADMIT            [ /*IN.s:Sonst_String*/
            String_ist_      *
            nicht_GKZ      [ OUT.IstGKZ:'FALSE'
```

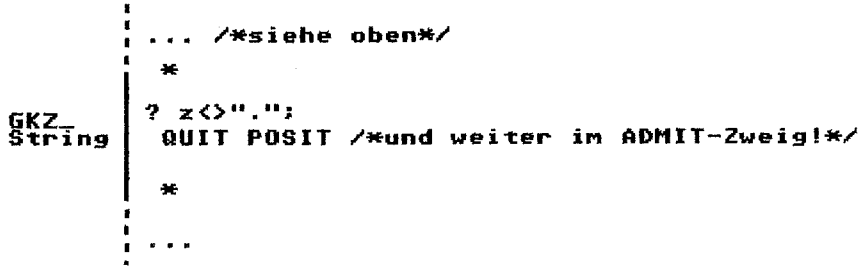
Es bleibt, den POSIT-Zweig auszuarbeiten. Wir gehen (ohne dies noch einmal explizit hinzuschreiben) davon aus, daß die "IN.s:..." - Operation bis zu den elementaren Objekten von "GKZ–String" "durchgeschoben" und der Stringverknüpfungs-Operator "#" durch den Sequenz-Operator "*" ersetzt wurde:



Da in der Beschreibung von "GKZ–String" optionale Komponenten und Iterationsobjekte aufgeführt sind, ist es unumgänglich, im POSIT-Zweig (!) mit einer Vorweglese-Aktion zu beginnen. So kann entschieden werden, ob das erste Zeichen des Strings ein Vorzeichen ist oder nicht:



Nachdem der Prozeß den Abschnitt Ziffernfolge verlassen hat, bezeichnet "z" das erste Zeichen im String "s", welches weder ein Vorzeichen noch eine Ziffer ist. (Man beachte, daß dies auch das erste Zeichen überhaupt sein kann!) Laut Strukturbeschreibung muß dieses Zeichen der Dezimalpunkt sein, wenn es sich um einen GKZ_String handelt. An dieser Stelle haben wir also die erste Gelegenheit, unsere Annahme (das "POSIT") zu überprüfen. Diese Überprüfung reduziert sich hier offenbar auf die Frage, ob das zuletzt gelesene Zeichen der Dezimalpunkt ist oder nicht. Ist er es nicht, so wird - wie oben bereits erläutert - der POSIT-Zweig verlassen (mit "QUIT POSIT") und in den ADMIT-Zweig eingetreten:



Wurde die Prüfung bestanden, so kann der Prozeß im POSIT-Zweig weiterlaufen, und zwar in der Komponente "Dezimalpunkt":

```
        ... /*siehe oben*/
         *
GKZ_    Dezimalpunkt  [ /*IN.s:"."*/
String                   *
                         IN.s:Zeichen(=z)
         *
        ...
```

Da die "Ziff_folge_2" nach dem Dezimalpunkt (gemäß GKZ-Struktur) mindestens eine Ziffer enthalten muß, werden wir die Annahme, daß die untersuchte Zeichenkette eine Gleitkommazahl darstellt, wiederum verwerfen müssen, wenn das nunmehr von "z" bezeichnete Zeichen keine Ziffer ist. Ist es aber eine Ziffer, so lesen wir alle folgenden Zeichen bis zur nächsten Nicht-Ziffer:

```
        ... /*siehe oben*/
         *
GKZ_    ? z NOT IN ("0",..."9");
String   QUIT POSIT
         *
        Ziff_folge_2 [ /*s.o.*/
         *
        ...
```

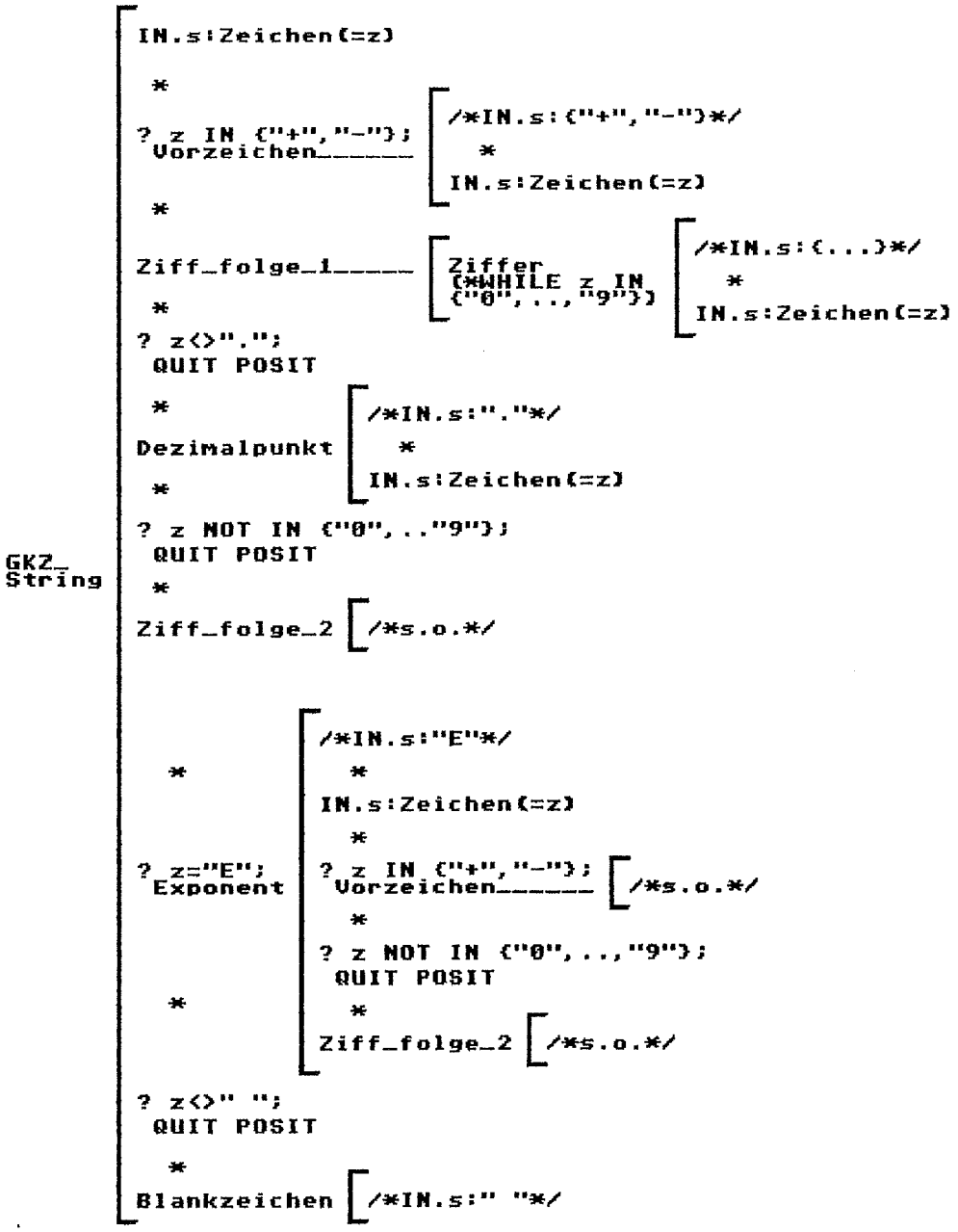
Der weitere Gang der Handlung dürfte nun klar sein:

```
        ... /*siehe oben*/

GKZ_                    [ /*IN.s:"E"*/
String    *               *
                          IN.s:Zeichen(=z)
                           *
                        ? z IN ("+","-"); [ /*s.o.*/
          ? z="E";       Vorzeichen_____
           Exponent       *
                          ? z NOT IN ("0",..,"9");
                           QUIT POSIT
          *               *
                          Ziff_folge_2 [ /*s.o.*/

        ? z<>" ";
         QUIT POSIT
          *
        Blankzeichen [ /*IN.s:" "*/
```

Damit ist auch der POSIT-Zweig vollständig beschrieben. Zur besseren Übersicht bilden wir den wesentlichen Teil dieser Beschreibung noch einmal im Zusammenhang ab:



Bevor wir an die Ausarbeitung der MODULA-2 - Prozedur "IstGKZ" gehen, sei noch die folgende Bemerkung angebracht: Neben der Technik des "Vorweg-Lesens" haben wir mit der POSIT-ADMIT-Konstruktion ein weiteres Verfahren, um auf der Grundlage von Prozeßbeschreibungen Analyse-Programme zu entwerfen. Diese Konstruktion ist aber gelegentlich auch bei der Produktion von Datenobjekten von Nutzen: Angenommen, ein solches Objekt bestehe aus mehreren Tei-

len, die aneinandergefügt werden müssen. Bei jeder "Anfüge-Aktion" gebe es eine Auswahl aus einem Reservoir von Teilen und irgendwelche Kriterien, nach denen die Auswahl zu treffen ist; das Datenobjekt selbst soll gewissen Anforderungen genügen müssen. Solange das Objekt noch nicht fertig ist, wird man in einer "POSIT-Aktion" das jeweils nächste Teil anfügen, um dann zu überprüfen, ob die geforderten Bedingungen eingehalten sind oder nicht. Falls nicht, wird man in einer "ADMIT-Aktion" die Anfüge-Operation rückgängig machen und alle ihre eventuellen Auswirkungen beseitigen. Der Leser ist aufgefordert, das allgemeine, in Kapitel 3 besprochene (iterative) Backtrack-Schema einmal nach diesem Muster zu formulieren.

Nun zur Realisierung des letzten Programmentwurfs: Im Gegensatz zum vorigen, auch in Hinblick auf die programmiersprachliche Umsetzung ziemlich einfache Beispiel, gibt es hier einige Probleme, die der eingehenderen Diskussion wert sind. Dies betrifft zum ersten die Lese-Operation "IN.s:..." und zum zweiten das "QUIT POSIT".

"IN.s:..." soll das jeweils nächste in der Zeichenkette "s" stehende Zeichen abliefern. Es liegt daher nahe, hierfür eine geeignete Funktions-Prozedur zum Beispiel mit dem Namen "NaechstesZeichen" zu schreiben. Der Einfachheit halber gehen wir davon aus, daß die Lese-Operation nur innerhalb der "IstGKZ"-Prozedur benötigt wird. Wir werden daher an dieser Stelle darauf verzichten, die "NaechstesZeichen"-Prozedur mit dem String, von dem gelesen werden soll, zu parametrieren. Später, in Kapitel 6, wird gezeigt, wie derartige Operationen, welche ja ganz allgemeine "Dienstleistungen" erbringen, implementiert werden sollten, um auch *allgemein zugänglich* zu sein. Hier begnügen wir uns mit dem folgenden Prozedurkopf:

"PROCEDURE NaechstesZeichen: CHAR"

Diese Prozedur muß natürlich immer "wissen", von welcher Stelle des Strings "s" das nächste Zeichen zu holen ist. Ferner muß sie erkennen, ob schon das Ende des Strings erreicht ist. Letzteres kann man zum Beispiel dadurch bewerkstelligen, daß Zeichenketten grundsätzlich durch ein spezielles "String-Ende-Zeichen" (namens "Eos") abgeschlossen werden. (Steht in der Variablen "s" ein leerer String, so enthält "s" also zumindest an erster Stelle das Zeichen "Eos".) Die erste Forderung läßt sich offenbar nur erfüllen, wenn außerhalb der Prozedur "NaechstesZeichen" eine Variable vom Typ CARDINAL (nennen wir sie "i") deklariert ist, welche ausschließlich dazu dient, den "Index des als nächstes abzuliefernden Zeichens" zu speichern. Wir erleben hier also einen Fall, in dem wir von der im Abschnitt 2.2.2 aufgestellten Regel, nach der Funktionsprozeduren mit "Seiteneffekten" (z.B. Veränderung des Inhalts globaler Variablen) vermieden werden sollten, abweichen *müssen*. Wie man mit solchen Abweichungen in einer systematischen und kontrollierten Weise umgeht, werden wir ebenfalls erst im folgenden Kapitel eingehend studieren. Vorläufig begnügen wir uns mit

folgendem Vorschlag für die Prozedur "NaechstesZeichen":

```
PROCEDURE NaechstesZeichen: CHAR;
VAR c:CHAR;
(* Die extern vereinbarte Variable "i" enthält die Stelle,
von der das nächste Zeichen zu holen ist.*)
BEGIN
  c:=s[i];
  IF c # EosC THEN
    i:=i+1
  END (*IF*);
  RETURN c
END NaechstesZeichen;
```

Die Variable "i" muß initialisiert werden können. Dies besorgt die Prozedur:

```
PROCEDURE InitLesen;
BEGIN
  i:=0
END InitLesen;
```

Mit den beiden Prozeduren "InitLesen" und "NaechstesZeichen" und mit der Index-Variablen "i" haben wir eine einfache Lösung des Problems der Realisierung der "IN.s:"-Operation gefunden.

Auf den ersten Blick nicht ganz so einfach scheint die Umsetzung der POSIT-ADMIT-Konstruktion mit Hilfe von MODULA-2 zu sein. Die "QUIT POSIT"-Aktion läuft ja darauf hinaus, daß man alle jeweils (im POSIT-Zweig) nachfolgenden Aktionen übergeht und an das Ende des POSIT-Zweigs (beziehungsweise an den Anfang des ADMIT-Zweigs) "springt". Mit älteren Programmiersprachen (auch mit PASCAL) wäre dies durch eine "GOTO"-Anweisung sehr einfach zu erledigen. Diese Anweisung gehört jedoch nicht (vgl. Abschnitt 2.2.2) zum Sprachumfang von MODULA-2. MODULA-2 - Prozeduren freilich kann man jederzeit mit der "RETURN"-Anweisung abbrechen (vgl. Anhang). Es bietet sich somit die folgende Möglichkeit an, die POSIT-ADMIT-Konstruktion auf "kanonische" (d.h. auf beliebige andere vergleichbare Situationen verallgemeinerbare) Weise zu realisieren: Man deklariert (im Gültigkeitsbereich von "IstGKZ") eine Variable "posit" vom Typ BOOLEAN und verpackt den POSIT-Zweig in eine eigene (innerhalb von "IstGKZ" deklarierte) Prozedur "StringIstGKZ". Zu Beginn dieser Prozedur wird "posit" der Wert "TRUE" zugewiesen und "QUIT POSIT" wird zu "... posit:=FALSE; RETURN; ...". Die Aktionen des ADMIT-Zweiges werden dann nur unter der Bedingungen "posit=FALSE" ausgeführt. Die Prozedur "IstGKZ" lautet somit insgesamt:

```
PROCEDURE IstGKZ(s:ARRAY OF CHAR): BOOLEAN;
(*Zusätzliche Einrückungen werden entsprechend der
Struktur des Klammerdiagramms vorgenommen.*)
CONST EosC= 00C; (*zum Beispiel*)
        DezPunktC = '.'; ExpC = 'E'; BlankC = ' ';
TYPE CharMengeTyp = SET OF CHAR;
VAR istGKZ, posit: BOOLEAN;
      i: CARDINAL;
  PROCEDURE InitLesen;
  BEGIN
   i:=0
  END InitLesen;
  PROCEDURE NaechstesZeichen: CHAR;
  VAR c: CHAR;
  BEGIN
   c:=s[i];
   IF c # EosC THEN
    i:=i+1
   END (*IF*);
   RETURN c
  END NaechstesZeichen;
  PROCEDURE StringIstGKZ;
  VAR z: CHAR;
  BEGIN
   posit:=TRUE;
   (*GKZ_String*)
    z:=NaechstesZeichen;
    IF z IN CharMengeTyp{'+','-'} THEN
    (*Vorzeichen*)
     z:=NaechstesZeichen
    END (*IF*);
    (*Ziff_folge_1*)
     WHILE z IN CharMengeTyp{'0'..'9'}DO
     (*Ziffer*)
      z:=NaechstesZeichen
     END (*WHILE*);
    IF z # DezPunktC THEN
     posit:=FALSE; RETURN
    END (*IF*);
    (*Dezimalpunkt*)
     z:=NaechstesZeichen;
```

```
      IF NOT(z IN CharMengeTyp{'0'..'9'}) THEN
        posit:=FALSE; RETURN
      END (*IF*);
      (*Ziff_folge_2*)
        WHILE z IN CharMengeTyp{'0'..'9'} DO
        (*Ziffer*)
          z:=NaechstesZeichen
        END (*WHILE*);
      IF z = ExpC THEN
      (*Exponent*)
        z:=NaechstesZeichen;
        IF z IN CharMengeTyp{'+','-'} THEN
        (*Vorzeichen*)
          z:=NaechstesZeichen
        END (*IF*);
        IF NOT(z IN CharMengeTyp{'0'..'9'}) THEN
          posit:=FALSE; RETURN
        END (*IF*);
        (*Ziff_folge_2*)
          WHILE z IN CharMengeTyp{'0'..'9'} DO
          (*Ziffer*)
            z:=NaechstesZeichen
          END (*WHILE*);
      END (*IF*);
      IF z # BlankC  THEN
        posit:=FALSE; RETURN
      END (*IF*)
      (*Blankzeichen*) (*bereits gelesen*)
    (*OUT.IstGKZ:'TRUE'*)
    istGKZ:=TRUE
  END StringIstGKZ;
BEGIN (*IstGKZ*) (*AnalyseProgramm*)
  InitLesen;
  (*POSIT String_ist_GKZ*)
  StringIstGKZ;
  IF NOT posit THEN
  (*ADMIT String_ist_nicht_GKZ*)
    (*OUT.IstGKZ:'FALSE'*)
    istGKZ:=FALSE
  END (*IF*);
  RETURN istGKZ
END IstGKZ;
```

Wir sind uns der Tatsache bewußt, daß auch die hier vorgeschlagene Realisierung der POSIT-ADMIT-Konstruktion mit "Seiteneffekten" arbeitet (also mit der Veränderung des Inhalts von Variablen, die nicht innerhalb einer gegebenen Prozedur deklariert sind). Sicherlich hätten wir uns im Fall dieses speziellen Beispiels "das Leben etwas leichter machen können". Zunächst einmal hätte man "StringIstGKZ" auch als Funktions-Prozedur vom Typ BOOLEAN deklarieren und anstatt "... posit:=TRUE; RETURN; ..." nur "... RETURN FALSE; ..." schreiben können (und "... RETURN TRUE; ..." am Ende von "StringIstGKZ"). Damit hätten wir freilich ebenfalls einen Seiteneffekt in Kauf nehmen müssen (diesmal in einer Funktions-Prozedur, nämlich die Zuweisung des Wertes "TRUE" an die Variable "istGKZ"!). Eine zweite, echte Vereinfachung wäre es gewesen, ganz auf die Einführung einer gesonderten Prozedur "StringIstGKZ" zu verzichten und das "QUIT POSIT" direkt in ein "RETURN FALSE;" zu übersetzen. Eine Kodierung des ADMIT-Zweigs hätte sich damit erübrigt, da dessen einzige Aktion (OUT.IstGKZ:'FALSE') dann ebenfalls zu einem "RETURN FALSE;" geführt haben würde. Diese Vereinfachung ist allerdings nicht "kanonisch" in dem Sinne, daß sie auf alle denkbaren POSIT-ADMIT-Konstruktionen angewendet werden kann. Und nicht zuletzt würden wir mit beiden Vereinfachungen die "Entwurfstreue" der Realisierung aufgeben: Die dem Programm zugrundeliegenden Entwurfs-Überlegungen wären nicht mehr ohne weiteres erkennbar und nachvollziehbar. Aus diesem Grunde geben wir der oben ausgearbeiteten Version unseres Analyseprogramms den Vorzug, bei der im übrigen die Verwendung von Seiteneffekten durch einen sauberen Entwurf "diszipliniert" wird.

5.2 Programm-Konstruktion aus Input und Output

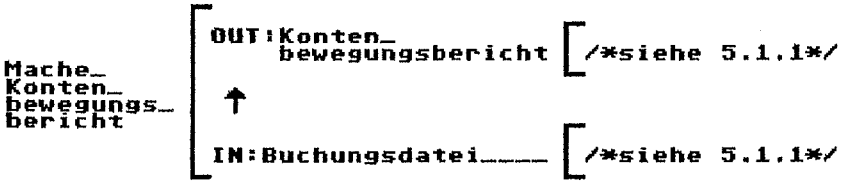
Nachdem wir mit den bisherigen Beispielen (in 5.1.2 und 5.1.3) die "Extremfälle" datenstrukturierten Programmentwurfs (vgl. die Bemerkungen vor Abschnitt 5.1.1) demonstriert haben, werden wir uns in diesem Abschnitt mit Aufgaben beschäftigen, welche die Verarbeitung mehr oder weniger komplexer Input-Datenströme zu Output-Datenströmen erfordern. Die Idee, die den hier darzustellenden Verfahren zugrundeliegt, haben wir - ganz intuitiv - bereits in der Einleitung zu diesem Kapitel skizziert. Um es zu wiederholen: Es geht im Prinzip um die Planung und Realisierung eines "Produktionsprozesses". Das was produziert werden soll, ist ein Strom von (Output-)Datenobjekten, der eine bestimmte Struktur hat. Dieser ist das Ziel, und ganz im Sinne einer zielgerichteten Vorgehensweise wird in den meisten Fällen die möglichst präzise Beschreibung der Struktur des gewünschten Output-Stroms den Ausgangspunkt für die Konstruktion des "Produktions"-Programms bilden. Die zentrale Frage lautet dabei: "Welche Input-Objekte benötigt man zur Herstellung der jeweiligen Output-Objekte?" Tatsächlich wird bei allen folgenden Beispielen der Output die gemäß dieser Fragestellung dominierende Rolle spielen. Ehrlicherweise dürfen wir freilich nicht verschweigen, daß eine Orientierung am vorhandenen Input für manche Probleme einen

besser gangbaren Lösungsweg verspricht. Dies gilt etwa für den - schon im letzten Abschnitt erwähnten - Bau von Compilern, wo es ja darauf ankommt, ausführbaren Code entsprechend einem vorgelegten Programmtext zu erzeugen.

Im ersten Teil dieses Abschnitts werden wir die Konstruktion eines Programms mittels Zuordnung von Input zu Output in ihrer einfachsten Form studieren und einen Satz von "Konstruktionsregeln" formulieren. Im zweiten Teil werden wir den Fall betrachten, daß mehrere Input-Ströme zu berücksichtigen sind, wobei jedoch immer noch einfache Zuordnungen möglich sein werden. Im dritten Teil schließlich behandeln wir Situationen, in denen es keine offensichtlichen Korrespondenzen zwischen Input und Output gibt, die aber dennoch, was die Anwendung des Verfahrens betrifft, nicht "hoffnungslos" sind.

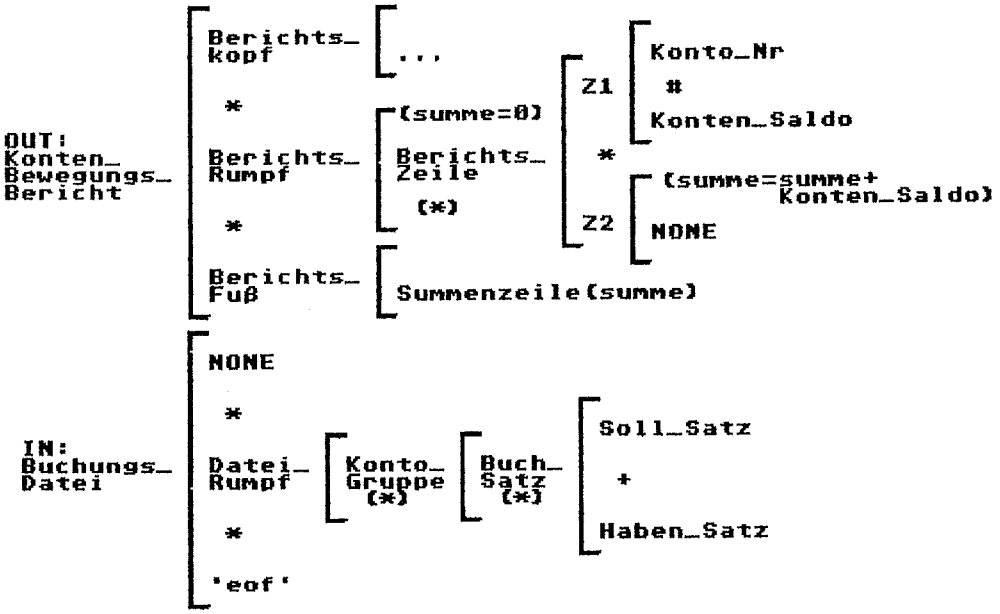
5.2.1 Das Verfahren in seiner einfachsten Form

Zur Darstellung des Verfahrens in seiner einfachsten Form kehren wir zu der im Abschnitt 5.1.1 gestellten Aufgabe zurück, aus einer nach aufsteigenden Kontonummern sortierten Buchungsdatei einen "Kontenbewegungsbericht" zu erzeugen. Das folgende Diagramm spezifiziert diese Aufgabe:



Was ist zu tun? In Anlehnung an die Beispiele in den Abschnitten 5.1.2 und 5.1.3 schlagen wir vor, zunächst eine Beschreibung des Prozesses der Herstellung des Kontenbewegungsberichts zu versuchen. Aus dieser Beschreibung sollte hervorgehen, aus welchen Komponenten des Input-Stroms die einzelnen Komponenten des Output-Stroms anzufertigen sind. Sie gibt eine Antwort auf die Frage: "Was muß angeliefert (bzw. vom "Input-Kanal" geholt) werden, um diese oder jene Komponente des Outputs produzieren zu können?" Sie muß (ähnlich wie wir dies im Falle der Eingabe einer Zeichenfolge "Kleinbuchstabe, Gleichheitszeichen, Großbuchstabe" (vgl. 5.1.3, Beispiel-Aufgabe (iii)) hervorgehoben haben) ein Bild dessen zeichnen, was ein (möglicherweise fiktiver) Beobachter des Produktions-Ablaufs tatsächlich sehen könnte. Sie macht dagegen noch keine Aussage darüber, gemäß welcher Bedingungen der Produktions-Ablauf gesteuert wird!

Um eine solche Prozeß-Beschreibung zu erhalten, stellen wir die Strukturen des Output-Stroms und des Input-Stroms einander gegenüber und analysieren sie (als Paar) auf mögliche Korrespondenzen hin:



Bis zur dritten "Klammerebene" sind einige strukturelle Entsprechungen deutlich erkennbar:

OUTPUT		INPUT
Berichts_Kopf	<—>	NONE
Berichts_Rumpf	<—>	Datei–Rumpf
Berichts_Fuß	<—>	'eof'
Berichts_Zeile	<—>	Konto–Gruppe

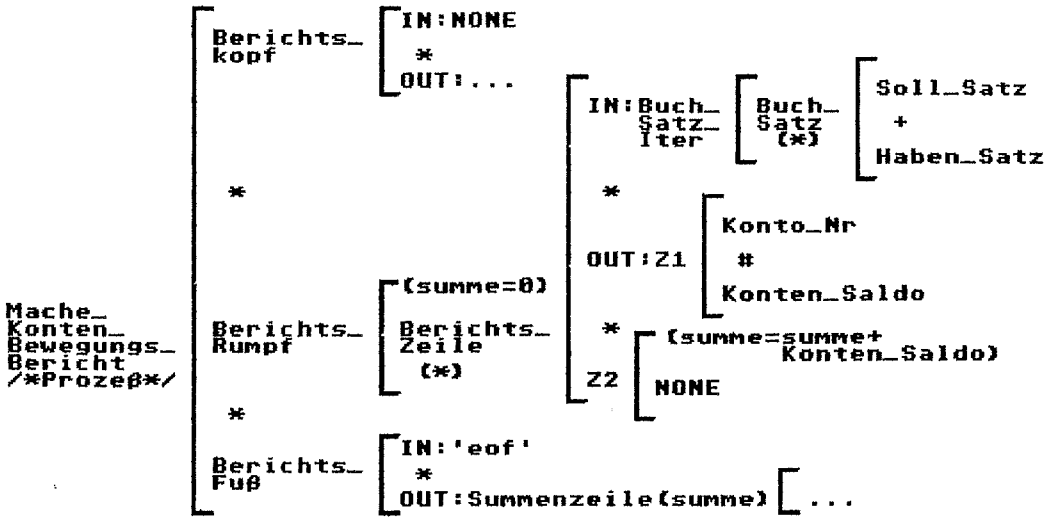
Man beachte, daß wir - um auch dem "Berichtskopf" eine Komponente des Inputs zuordnen zu können - die Beschreibung der "Buchungsdatei" leicht modifiziert haben, indem wir vor dem "Datei_Rumpf" das "Null-Objekt" NONE eingefügt haben.

Diese Entsprechungen sind wie folgt zu interpretieren: Für die Produktion des "Berichts_Kopfs" liegen keine Daten aus dem Input-Strom vor; der "Berichts_Rumpf" wird aus dem "Datei_Rumpf" hergestellt; das 'eof' gibt Anlaß zur Ausgabe des "Berichts_Fußes" und eine "Berichts_Zeile" ergibt sich aus einer "Konto_Gruppe".

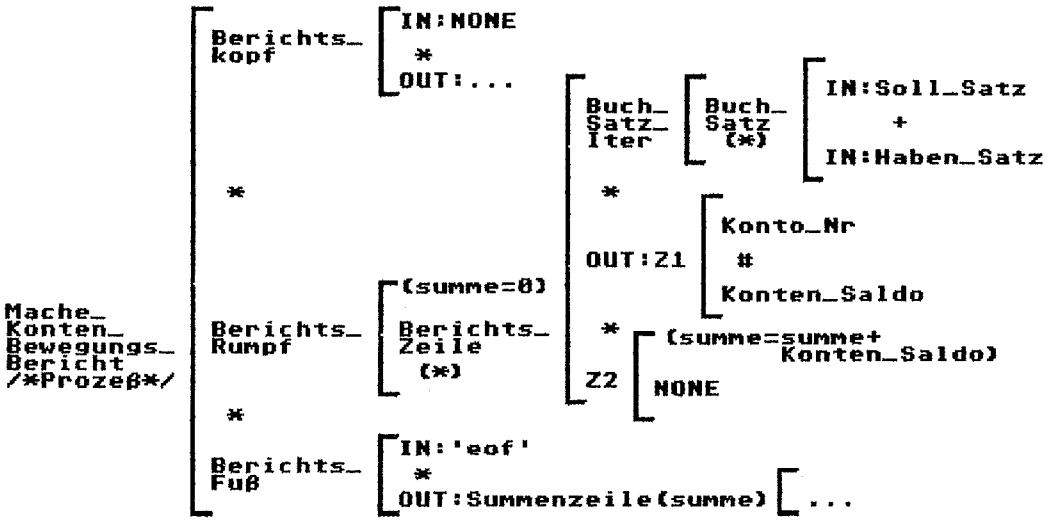
Wie steht es nun mit "Z1" im "Kontenbewegungsbericht" und der Iteration von "Buch_Satz" in der "Buchungsdatei"? Offenbar faßt diese Iteration all jene Buchungssätze zusammen, die zu einer "Konto_Gruppe", das heißt zu einer Kontonummer, gehören. Wir können also diese Iteration als Ganzes dem Output-Objekt "Z1" zuordnen.

Das Ergebnis dieser Analyse wird in einem Diagramm "Mache_Konten_Bewegungs_Bericht" dargestellt. Es entsteht aus dem Diagramm, welches die Struktur des Kontenbewegungsberichts beschreibt, dadurch, daß vor den Komponenten

des Outputs auf der jeweils letzten "Klammerebene" (von links), auf der Korrespondenzen festgestellt wurden, die entsprechenden Input-Komponenten notiert werden. Die Operationen "IN:" und "OUT:" werden dabei bis zu diesen "letzten" einander zugeordneten Komponenten "durchgeschoben":



Und nach einem weiteren Vorrücken der "IN:"-Operation ergibt sich das folgende Bild:

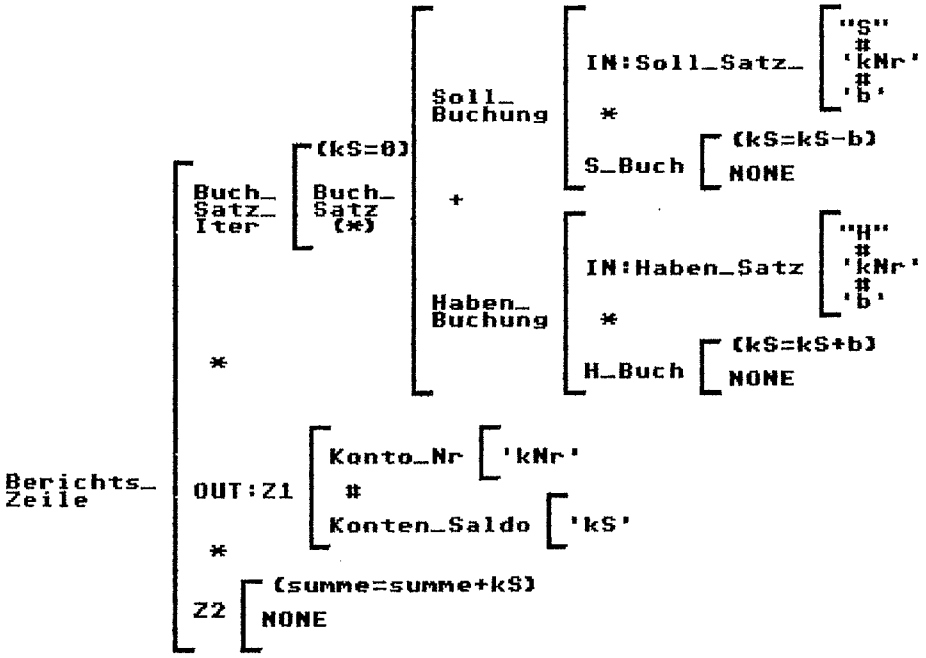


Es dürfte klar sein, warum wir dieses Diagramm als Prozeßbeschreibung bezeichnen können: Es gibt, ganz einfach, die Struktur einer Aktionenfolge wieder, welche durch die "Verzahnung" eines Input-Stroms mit einem Output-Strom entsteht. Beispiele solcher Aktionenfolgen (von Prozessen also, auf welche diese Beschreibung zutrifft) sind etwa:

Berichts_ Kopf	IN:NONE OUT:...	Berichts_ Kopf	IN:NONE OUT:...
Berichts_ Rumpf	IN:Soll_Satz IN:Soll_Satz IN:Haben_Satz OUT:Z1 IN:Soll_Satz IN:Soll–Satz OUT:Z1 IN:Soll_Satz IN:Soll_Satz IN:Haben_Satz IN:Soll_Satz IN:Soll_Satz IN:Haben_Satz OUT:Z1 IN:Soll_Satz IN:Soll_Satz IN:Haben_Satz IN:Haben_Satz OUT:Z1	Berichts_ Rumpf	IN:Haben_Satz IN:Soll_Satz OUT:Z1 IN:Soll_Satz IN:Haben_Satz IN:Haben_Satz IN:Haben_Satz IN:Soll_Satz OUT:Z1 IN:Soll_Satz IN:Haben_Satz OUT:Z1 IN:Haben_Satz OUT:Z1 IN:Haben_Satz IN:Haben_Satz IN:Soll_Satz OUT:Z1
Berichts_ Fuß	IN:'eof' OUT:Summenzeile	Berichts_ Fuß	IN:'eof' OUT:Summenzeile

Genauer müßten wir sagen, daß diese Folgen mögliche, *von außen sichtbare* Wirkungen der oben beschriebenen Prozesse repräsentieren. Andererseits enthielt aber schon das Diagramm "Kontenbewegungsbericht" (die Output-Beschreibung) Informationen über das Zustandekommen der "Summenzeile", die nach dem eben praktizierten Verfahren natürlich automatisch in das Diagramm "Mache_ Kontenbewegungsbericht" übernommen wurden. Diese Informationen betreffen Aktionen, die "prozeß-intern" ablaufen und von außen nicht sichtbar sind. Mit solchen "unsichtbaren" Aktionen ist die Prozeß-Beschreibung zu ergänzen, um auch die konkreten Werte (insbesondere den Wert von "Konten–Saldo"), die mit "Z 1" ausgegeben werden sollen, zu spezifizieren. Wir modifizieren zu diesem

Zweck die Komponenten "Haben_Satz" und "Soll_Satz", indem wir sie jeweils zusammen mit einem "Berechnungs-Objekt" in einer "Haben_Buchung" beziehungsweise "Soll_Buchung" unterbringen. Sämtliche Änderungen sind auf die Komponente "Berichts_Zeile" beschränkt:



Wie am Ende von Abschnitt 5.1.2, so drängt sich auch an dieser Stelle die Frage nach der Korrektheit dessen auf, was wir bisher zur Konstruktion eines Programms für die Erstellung des Kontenbewegungsberichts getan haben. Was gibt uns das Recht zu behaupten, die oben entwickelte "Prozeß-Beschreibung" sei korrekt im Sinne der Aufgabenstellung? Die einfache Antwort lautet: Gar nichts! Es gibt, wenn man von der Festlegung der Struktur des gewünschten Outputs absieht, keinerlei Grundlage für einen "Korrektheitsbeweis". Daß aber der beschriebene Prozeß einen derart strukturierten Output produziert, ist sicher keines Beweises wert, da wir die Prozeß-Struktur durch eine direkte Erweiterung der Output-Struktur gewonnen haben. Etwas problematischer sind die "Berechnungs-Objekte" innerhalb der Prozeß-Beschreibung. Sowohl die mit dem Output als auch die mit dem Input verknüpften Berechnungen spiegeln unser Verständnis der zu behandelnden Aufgabe wider. Abermals die Analogie mit dem Fertigungsbetrieb bemühend, kann man sagen, daß die in die Prozeß-Beschreibung eingefügten Berechnungs-Objekte den Vorschriften entsprechen, nach denen das angelieferte Material zu bearbeiten und zusammenzusetzen ist, um ein bestimmtes Produkt zu erzeugen. Mit einem eventuellen Auftraggeber ist abzuklären, ob das Resultat der Anwendung dieser Vorschriften seinen Vorstellungen entspricht. Damit wird das Prozeß-Beschreibungs-Diagramm - ähnlich wie die Objekt-Beschreibungen in Abschnitt 5.1.2 - zur (mehr oder weniger willkürlich

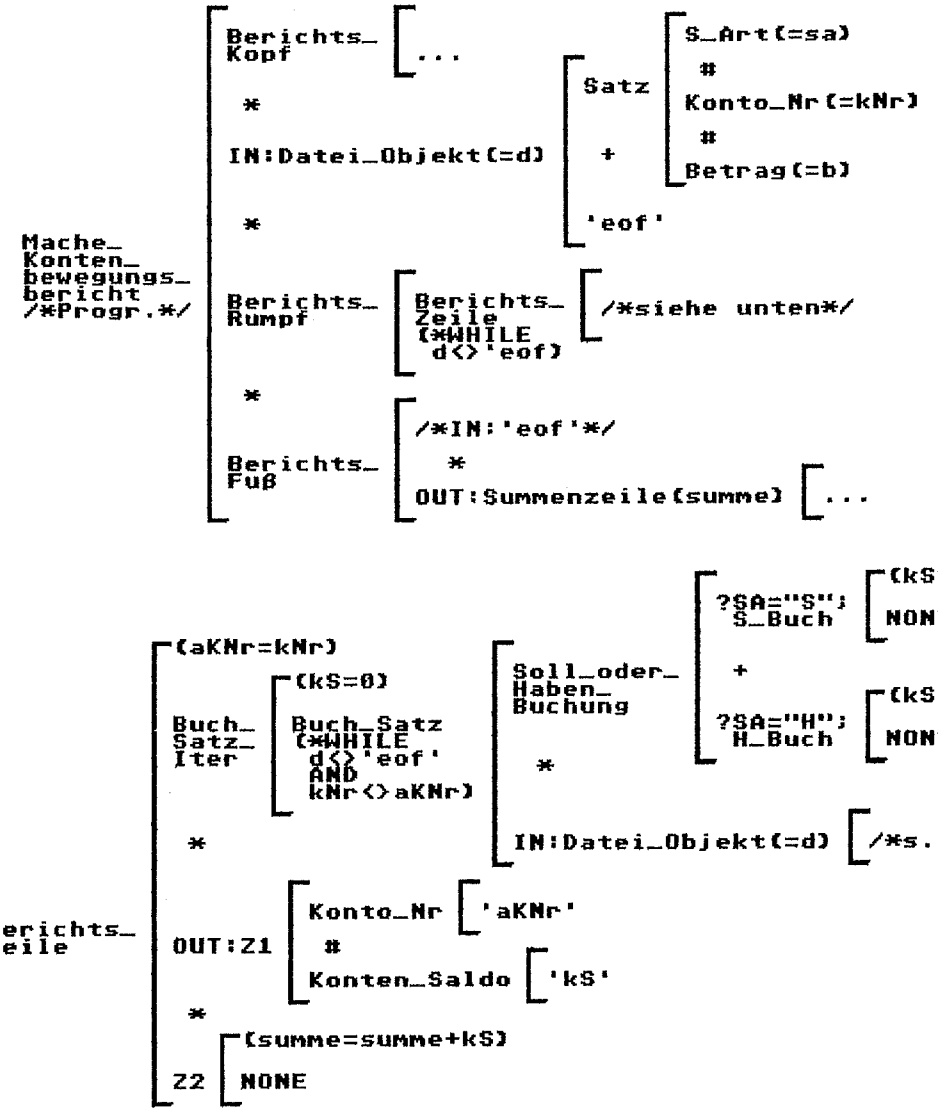
konstruierten) *Spezifikation* (!) eines Programms, welches den Prozeß steuert und in Gang hält.

Die Ableitung einer Programm-Beschreibung aus einer solchen Spezifikation dürfte - nach den ausführlichen Vorbereitungen in Abschnitt 5.1.2 - für das spezielle Beispiel "Kontenbewegungsbericht" keine Schwierigkeiten mehr bereiten.

Da wir zu entscheiden haben, unter welchen Bedingungen

- die Komponenten "Berichts–Zeile" und "Buch–Satz" zu wiederholen sind und nach welchen Kriterien
- zwischen einer "Soll–Buchung" und einer "Haben–Buchung" zu unterscheiden ist,

wird uns hier offenbar die Technik des "Vorweg-Lesens" weiterhelfen:



Im Vergleich mit der trivialen Situation in Abschnitt 5.1.2 (Beispiel (iii)) gibt es im vorliegenden Fall beim Einsatz der Vorweglese-Technik einiges mehr zu bedenken. Beispielsweise ist zu berücksichtigen, daß die Buchungsdatei Objekte verschiedener Art enthält. So ist das, was wir von der Buchungsdatei holen, entweder ein Satz oder ein eof-Objekt. Wir tragen dem Rechnung, indem wir die Struktur des gelesenen Objekts genau angeben. Die Benennung dieses "Datei_ Objekts" und seiner Teile verschafft uns im weiteren die Möglichkeit (nicht anders als in Abschnitt 5.1.2), die angestrebten Bedingungen zu formulieren. So wird eine "Berichts–Zeile" überhaupt nur dann anzufertigen sein, wenn nicht sogleich ein eof-Objekt gelesen wird, und man wird solange weitere Berichtszeilen produzieren müssen, als die (Nach-)Lese-Aktion kein eof liefert. Wird aber eof gelesen, so gibt dies Anlaß zur Ausgabe von "Summenzeile( )". Wir haben dies (wie in 5.1.2) dadurch betont, daß wir die Input-Aktion im "Berichts_ Fuß" in Kommentarklammern gesetzt haben. Die Bedingung der Iteration von "Buch_Satz" erfordert etwas mehr Überlegung. Da die Folge von "Buch_Sätzen", welche für eine auszugebende Zeile "Z 1" verantwortlich ist, sich auf ein und dieselbe Kontonummer bezieht, wird - vor der Ausgabe von "Z 1" - die Eingabe-Aktion solange zu wiederholen sein, als kein Wechsel der Kontonummer stattfindet. Um diesen Wechsel feststellen zu können, bezeichnen wir die jeweils aktuelle Konto-Nummer mit "aKNr". Zu Beginn der Produktion einer "Berichts_ Zeile" gilt also "aKNr = kNr". Natürlich wird die Aufbereitung einer "Berichts_ Zeile" auch dann beeendet, wenn das eof-Objekt erreicht ist. Damit lautet die Iterationsbedingung für "Buch_Satz": "d<>'eof' AND kNr=aKNr", wobei sich "d" und "kNr" auf das jeweils zuletzt (nach-)gelesene "Datei_Objekt" beziehen. In "Z 1" muß natürlich der Wert "aKNr" eingesetzt werden. Die Entscheidung, ob eine "Soll–Buchung" oder eine "Haben–Buchung" vorzunehmen ist, ergibt sich einfach aus der Art ("SA") des gerade gelesenen Satzes.

Auch die Umsetzung des nunmehr fertigen Entwurfs in einen MODULA-2 - Programmtext bereitet, wie bei den schon behandelten Aufgaben, keine großen Probleme mehr. Bezüglich der "IN"- und "OUT"-Operationen nehmen wir an, daß sie durch die Prozeduren

"PROCEDURE InSatz(VAR d:BuchSatzTyp; VAR eof:BOOLEAN);"
und "PROCEDURE OutZeile(kNr:KontoNrTyp; b:BetragsTyp);"

realisiert sind. Auch für den "Berichts–Kopf" und die "Summenzeile( )" setzen wir jeweils geeignete Prozeduren voraus. Entsprechend der bereits in den obigen Diagrammen enthaltenen Strukturbeschreibung sei "BuchSatzTyp" deklariert als

```
"TYPE BuchSatzTyp = RECORD
                        satzArt: SatzArtTyp;
                        kontoNr: KontoNrTyp;
                        betrag:  BetragsTyp
                    END (*RECORD*);",
```

mit

"TYPE SatzArtTyp = (soll,haben);"

(Auf die explizite Angabe der übrigen Typdefinitionen wollen wir hier verzichten.) Damit ergibt sich aus dem als Diagramm vorliegenden Entwurf die folgende MODULA-2 - Prozedur:

```
PROCEDURE MacheKontenbewegungsbericht;
VAR d:BuchSatzTyp;
    eof:BOOLEAN;
    aKNr:KontoNrTyp;
    kS,summe:BetragsTyp;
 PROCEDURE InSatz(VAR d:BuchSatzTyp; VAR eof:BOOLEAN);
 BEGIN
  (* ... *)
 END InSatz;
 PROCEDURE OutZeile(kNr:KontoNrTyp; b:BetragsTyp);
 BEGIN
  (* ... *)
 END OutZeile;
 PROCEDURE BerichtsKopf;
 BEGIN
  (* ... *)
 END BerichtsKopf;
 PROCEDURE SummenZeile(s:BetragsTyp);
 BEGIN
  (* ... *)
 END SummenZeile;
BEGIN (*MacheKontenbewegungsbericht*)
 BerichtsKopf;
 InSatz(d,eof);
 (*Berichts_Rumpf*)
 summe:=0;
 WHILE NOT eof DO
  (*Berichts_Zeile*)
  aKNr:=d.kontoNr;
  (*Buch_Satz_Iter*)
  kS:=0;
  WHILE (NOT eof) AND (d.kontoNr=aKNr) DO
   (*Buch_Satz*)
   (Soll_oder_Haben_Buchung*)
```

```
      CASE d.satzArt OF
        soll: (*S_Buch*)
              kS:=kS-d.betrag |
        haben:(*H_Buch*)
              kS:=kS+d.betrag
      END (*CASE*);
      InSatz(d,eof)
    END (*WHILE*)
    (*Z 1*)
    OutZeile(aKNr,kS);
    (*Z 2*)
    summe:=summe+kS
  END (*WHILE*);
  (*Berichts_Fuß*)
  SummenZeile(summe)
END MacheKontenbewegungsbericht;
```

Zur besseren Veranschaulichung der Herleitung des Programmtextes aus der Programm-Beschreibung haben wir die Namen der Diagramm-Komponenten als Kommentare (bzw. Prozedurnamen) übernommen.

Wenn wir nun die Entwicklung dieses Programms noch einmal Revue passieren lassen, so stellen wir fest, daß sie in einigen wohlunterscheidbaren Schritten verlief. Durch die Aufgabenstellung war uns das Ziel vorgegeben, einen Prozeß in Gang zu setzen und zu steuern, der eine bestimmte Folge von Outputs hervorbringen soll. Eine Folge von Inputs stand dazu zur Verfügung. Die Schritte, die uns zum Ziel führten, waren:

Schritt 1:

Beschreibung des gewünschten Outputs.

Schritt 2:

Falls Inputs gegeben sind: Beschreibung der Inputs.

Outputs und Inputs sind in der Regel Daten-Ströme. Man beschreibt also die Struktur dieser Daten-Ströme. Neben der Darstellung der Struktur sollte insbesondere die Beschreibung des Output-Stroms soviel Information über die auszugebenden Objekte enthalten, wie ohne Kenntnis der Inputs möglich ist.

Schritt 3:

Auffinden struktureller Entsprechungen zwischen Output und Input. Konstruktion eines Diagramms, welches diese Entsprechungen deutlich macht. (Falls Inputs nicht explizit gegeben sind, frage man sich jeweils, welche Inputs zur Produktion der Output-Objekte erforderlich sind.)

Schritt 4:

Spezifikation der zur Erzeugung des Outputs notwendigen Manipulationen des Inputs.

Die Schritte 1-4 liefern eine Beschreibung des Prozesses, und zwar sowohl der Struktur seines Ablaufs als auch der jeweiligen Zuordnung von Inputs und deren Manipulationen zu Objekten des Output-Stroms.

Schritt 5:

Transformation der Prozeß-Beschreibung in die Beschreibung eines Programms, welches den Prozeß (auf einem Rechner) in Gang hält und ihn steuert. Es sind Bedingungen für den Abbruch von Iterationen und für die Auswahl alternativer Aktionen zu finden. Eine einfache Technik, die dabei angewandt werden kann, ist das "Vorweg-Lesen".

Schritt 6:

Umsetzung der Programm-Beschreibung in einen Text in der gegebenen Programmiersprache.

5.2.2 Verarbeitung mehrerer Input-Ströme

In diesem Abschnitt werden wir die Ausführung der sechs Schritte des "Datenstrukturierten Programm-Entwurfs" an zwei weiteren, ebenfalls sehr kleinen Beispielen einüben. Die dabei anfallenden Diagramme werden - ebenso wie die letztlich entstehenden Programme - nur in groben Zügen skizziert; die Ausarbeitung von Details bleibt dem Leser überlassen.

Im ersten Beispiel behandeln wir die "klassische" Aufgabe des Mischens (englisch: "merging", "collating") zweier sortierter Dateien.

> "Ein Versandunternehmen schickt Rechnungen an seine Kunden. Es speichert die Rechnungsdaten auf einer sequentiellen "Rechnungs–Datei". Jeder Satz dieser Datei enthält eine Kundennummer und den Rechnungsbetrag. Die erhaltenen Zahlungen werden auf einer sequentiellen "Zahlungs–Datei" vermerkt. Jeder Satz dieser Datei enthält eine Kundennummer und den eingegangenen Betrag."

Um das Beispiel nicht ungebührlich aufzublähen, machen wir die stark vereinfachende Voraussetzung, daß in keiner dieser beiden Dateien eine Kundennummer mehr als einmal vorkommt.

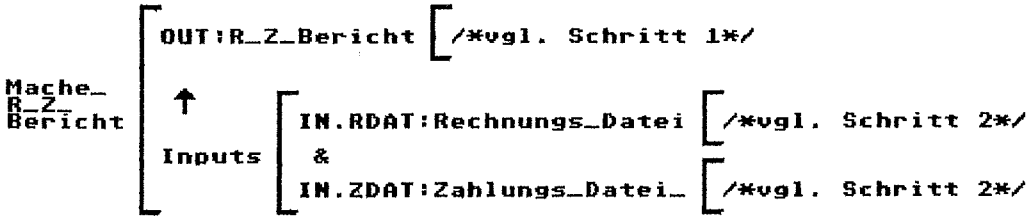
> "Es soll ein Bericht produziert werden, der die folgenden Informationen liefert:
>
> - Die verbleibende Schuld jedes Kunden, dem eine Rechnung geschickt wurde und von dem eine Zahlung eingegangen ist;

- die Nummern derjenigen Kunden, von denen noch keine Zahlung eingegangen ist;
- Zahlungen, die ohne voraufgegangene Rechnung erfolgten."

Wir dürfen davon ausgehen, daß sowohl die "Rechnungs_Datei" als auch die "Zahlungsdatei" nach aufsteigenden Kundennummern sortiert vorliegen.

Der zu produzierende Bericht erhält den Namen "R_Z_Bericht".

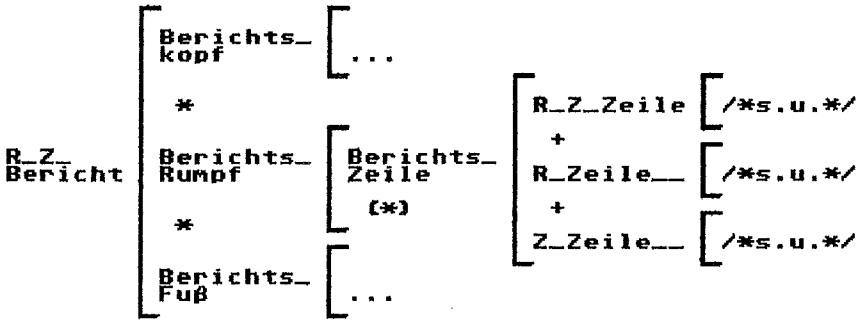
Das folgende Bild faßt die Aufgabenstellung zusammen:



Um die beiden Input-Ströme sauber voneinander zu unterscheiden, lassen wir sie über verschiedene Kanäle "fließen". Wir müssen daher die Eingabe-Operation "IN:" mit dem Namen des jeweiligen Kanals qualifizieren. (Vgl. das Beispiel "Analyse eines GKZ_Strings" in Abschnitt 5.1.3.)

Schritt 1: Beschreibung des gewünschten Outputs

Der R–Z–Bericht hat die folgende Struktur:

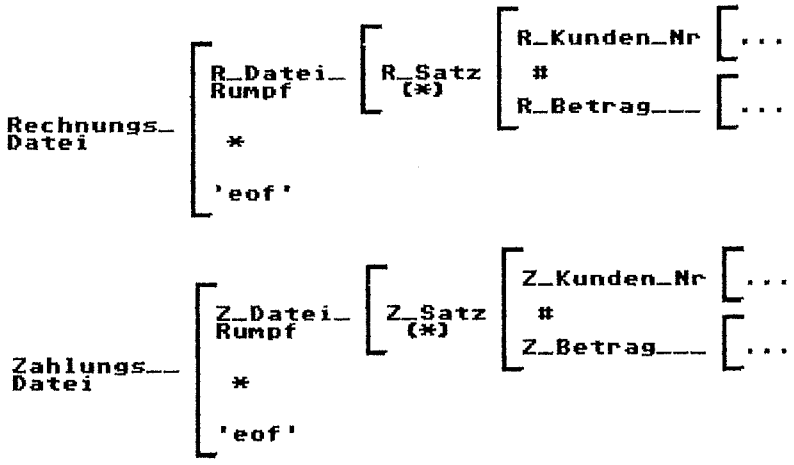


Wir begnügen uns mit einer informellen Beschreibung des Inhalts der einzelnen Zeilen(-Typen):

- Jede Zeile beginnt mit einer "Kunden-Nummer";
- eine R_Z_Zeile enthält den Betrag der verbleibenden Schuld;
- eine R_Zeile enthält den Betrag der verbleibenden Schuld sowie den Vermerk "KEINE ZAHLUNG EINGEGANGEN";
- eine Z_Zeile enthält den eingegangenen Betrag sowie den Vermerk "ZAHLUNG OHNE RECHNUNG".

Schritt 2: Beschreibung der Inputs

Die Beschreibungen der Input-Dateien (d.h. der Strukturen der "Input-Ströme") sind - selbstverständlich bis auf die Bezeichnungen der darin enthaltenen Objekte - völlig identisch:



Wie bereits bemerkt, setzen wir voraus, daß beide Dateien nach aufsteigenden Kundennummern ("R_Kunden_Nr" und "Z_Kunden_Nr") sortiert sind.

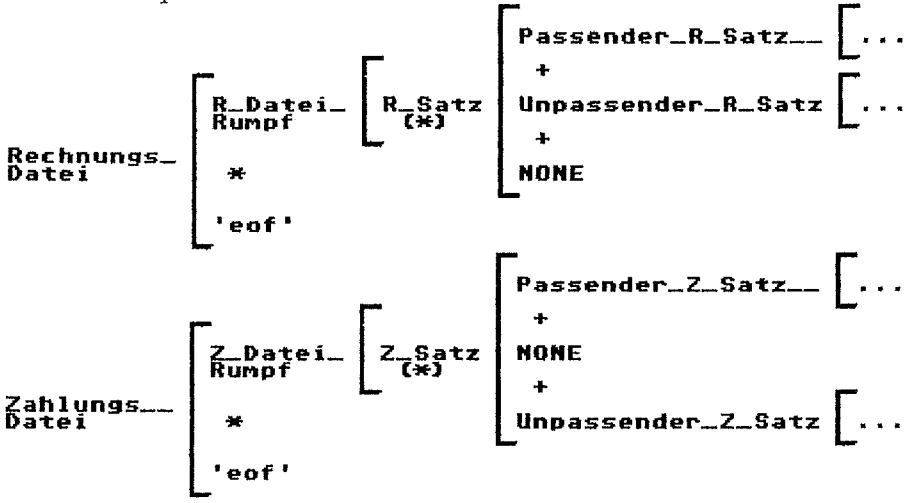
Schritt 3: Auffinden struktureller Entsprechungen

Bis zur "Ebene" der "Berichts–Zeile" (im "R_Z_Bericht") und der "R_" beziehungsweise "Z_Sätze" (in "Rechnungs_Datei" und "Zahlungs_Datei") bereitet die Korrespondenz-Analyse keinerlei Schwierigkeiten. Denken wir uns die Datei-Strukturen wieder (wie in Abschnitt 5.2.1) durch einen zu Beginn eingefügten Nullobjekt-Typ ergänzt, so ergeben sich die folgenden Zuordnungen:

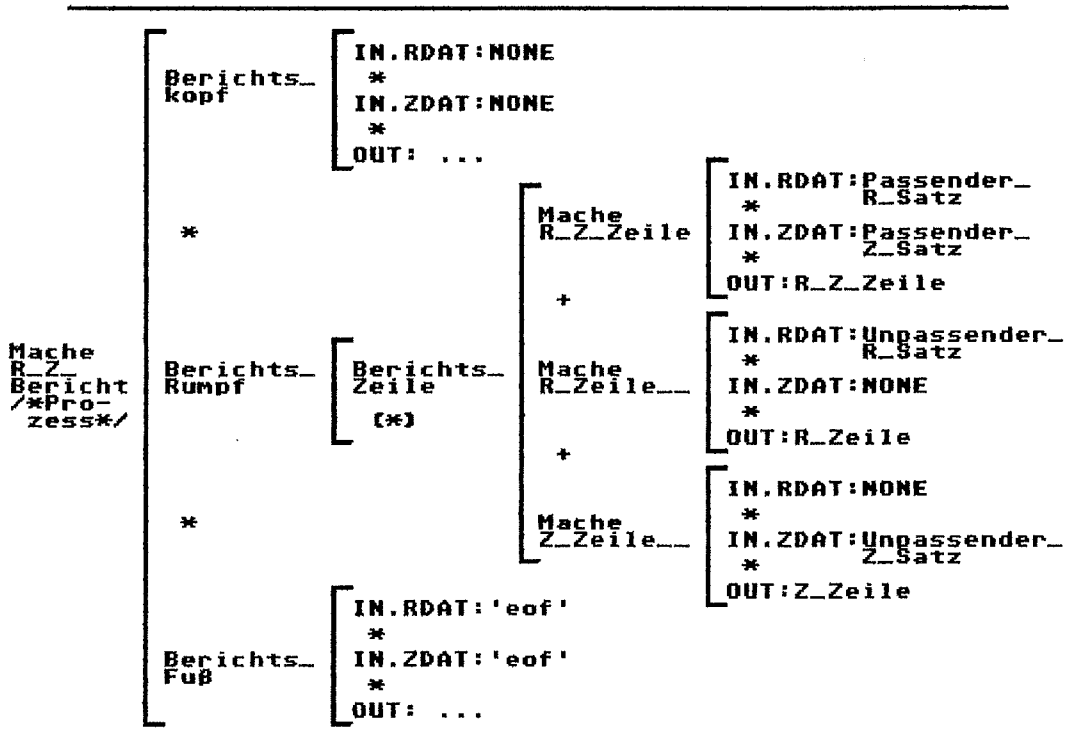
OUTPUT	INPUT aus "RDAT"	INPUT aus "ZDAT"
Berichts–Kopf	NONE	NONE
Berichts–Rumpf	R–Datei–Rumpf	Z–Datei–Rumpf
Berichts–Fuß	'eof'	'eof'
Berichts–Zeile	R–Satz	Z–Satz

Für die einzelnen "Zeilen-Arten" "R_Z_Zeile", "R_Zeile" und "Z_Zeile" des "R_Z_Berichts" finden wir in den Input-Strukturen zunächst keine Entsprechungen. Ein Ausweg aus diesem Dilemma liegt jedoch nahe: Zu den "Zeilen-Arten" des Outputs bilden wir einfach geeignete (fiktive) "Satz-Arten" im Input. Wie dies zu tun ist, ergibt sich, wenn wir uns fragen, "welche Inputs für welchen Output verantwortlich sind". Offenbar muß eine "R_Z_Zeile" genau dann produziert werden, wenn aus der "Rechnungs_Datei" und aus der "Zahlungs_Datei" zueinander passende, das heißt, mit der gleichen Kundennummer versehene Sätze angeliefert werden. Aus diesem Grund definieren wir sowohl "R_Satz" als auch "Z_Satz" neu als Selektionsobjekte mit den Alternativen "Passender_R(Z)_Satz" und "Unpassender_R(Z)_Satz". Ein "Unpassender_R_Satz" ist deshalb

unpassend, weil die in ihm enthaltene Kontonummer in keinem Satz der "Zahlungs_Datei" auftaucht. Einem "Unpassenden_R_Satz" entspricht also ein Nullobjekt-Typ in der Beschreibung von "Z_Satz". Das Gleiche gilt natürlich auch umgekehrt. Als weitere Alternative der jeweiligen Selektionsobjekte führen wir daher "NONE" explizit auf:



(Anzumerken ist hier, daß die Frage, nach welchen exakten Kriterien zu entscheiden ist, was ein "passender" und was ein "unpassender" Satz ist, erst im Schritt 5 behandelt werden kann.)



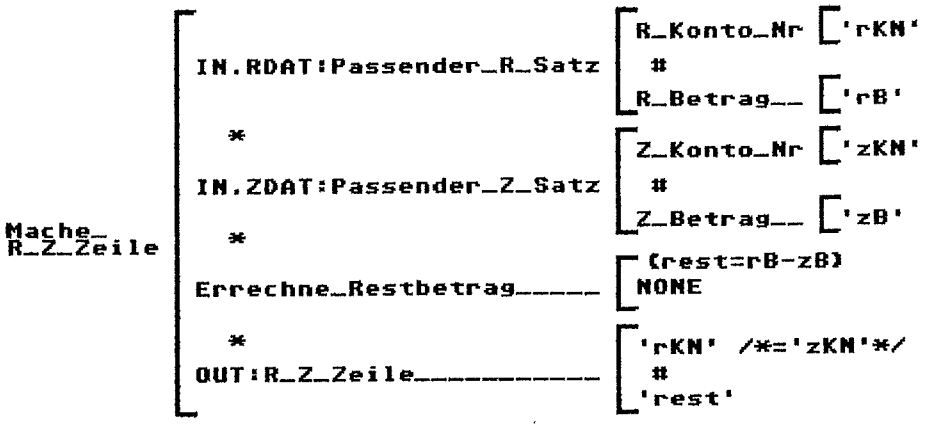
Damit kann die Liste der Zuordnungen vervollständigt werden:

OUTPUT	INPUT aus "RDAT"	INPUT aus "ZDAT"
R–Z–Zeile	Passender–R–Satz	Passender–Z–Satz
R–Zeile	Unpassender–R–Satz	NONE
Z–Zeile	NONE	Unpassender–Z–Satz

Und wir erhalten das auf der vorigen Seite abgebildete Prozeß-Beschreibungs-Diagramm "Mache_R_Z_Bericht".

Schritt 4: Spezifikation der Manipulationen des Inputs

Die Manipulationen der aus den Input-Kanälen gewonnenen Objekte sind in diesem Beispiel außerordentlich trivial. In den Prozeß-Komponenten "Mache_R_Zeile" und "Mache_Z_Zeile" werden die Beträge aus den hier jeweils gelesenen Sätzen einfach übernommen. Lediglich in "Mache_R_Z_Zeile" muß eine Berechnung vorgenommen werden. Dieser Teil des Diagramms ist wie folgt zu ergänzen:

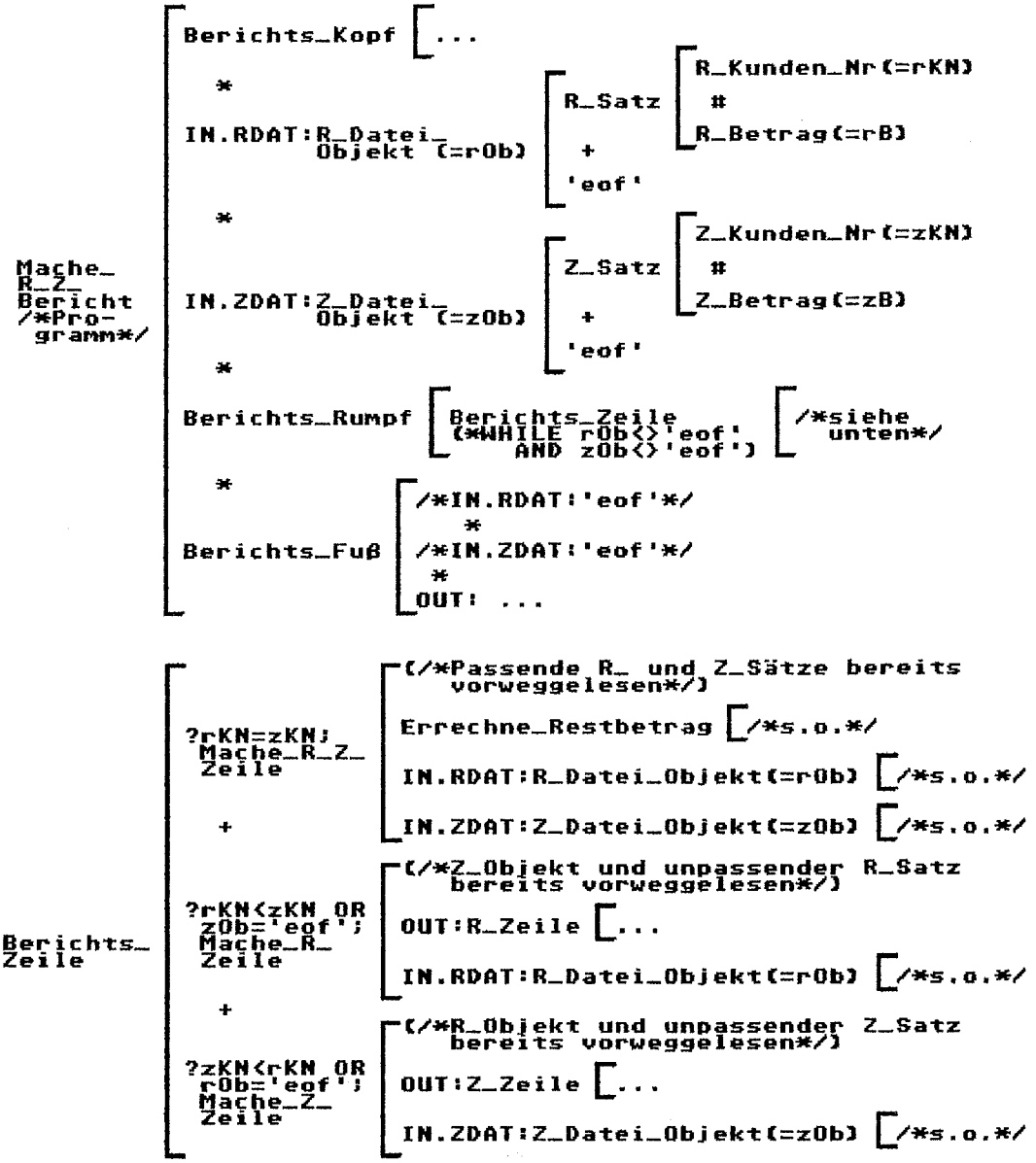


Schritt 5: Programmbeschreibung

Zur Herstellung der Programmbeschreibung können wir auch hier auf die bewährte Technik des Vorweg-Lesens zurückgreifen.

Zweierlei ist dabei zu beachten. Erstens muß die Iteration "Berichts_Zeile" solange laufen, wie nicht von beiden Input-Kanälen das EOF-Objekt gelesen wird. Um vor Beginn der Iteration entscheiden zu können, ob diese überhaupt durchgeführt werden muß, ist von beiden Kanälen ein erstes Objekt zu entnehmen. Zweitens brauchen wir bei der Beschreibung dieser Objekte den Unterschied zwischen "passenden" und "unpassenden" Sätzen nicht mehr zu berücksichtigen, da sich dieser Unterschied anhand der gelesenen Kontonummern feststellen lassen wird.

Es ergeben sich die folgenden Diagramme:



In den Zweigen "Mache_R_Zeile" und "Mache_Z_Zeile" haben wir die - aus der Prozeß-Beschreibung als Nachlese-Aktionen eigentlich zu übernehmenden - Operationen "IN.RDAT:NONE" und "IN.ZDAT:NONE" ignoriert, da dies "leere Aktionen" sind. Der Leser mache sich klar, daß die Bedingungen, gemäß derer die Zeilen-Produktion gesteuert wird, aus der Tatsache folgen, daß die Input-Dateien in der oben angegebenen Weise sortiert sind. Man vergewissere sich außerdem, daß keine weiteren Fälle auftreten können.

Schritt 6: Kodierung

Wir geben lediglich ein MODULA-2 - Prozedur-"Gerüst" an, das sich aus der gefundenen Programm-Struktur ableiten läßt. Wie für das Beispiel in Abschnitt

5.2.1 verwenden wir Prozeduren "InSatz" und "OutZeile". Die Prozedur "InSatz" erhält als zusätzlichen Parameter den Namen des Kanals (der Datei!), von dem Objekte geholt werden; der Prozedur "OutZeile" wird die Art der zu produzierenden Zeile mitgeteilt, die (etwa) als Element eines skalaren Typs definiert ist. Weitere Einzelheiten können wie in 5.2.1 ausgearbeitet werden.

```
PROCEDURE MacheRZBericht;
(*Deklaration der notwendigen Typen, Konstanten, Variablen
  und Prozeduren*)
BEGIN
  BerichtsKopf;
  InSatz(RDAT,rOb,eofR);
  InSatz(ZDAT,zOb,eofZ);
  (*Berichts_Rumpf*)
  WHILE (NOT eofR) AND (NOT eofZ) DO
    (*Berichts_Zeile*)
    IF rOb.kundenNr=zOb.kundenNr THEN
      (*Mache_R_Z_Zeile*)
      rest:=rOb.betrag-zOb.betrag;
      OutZeile(rzZeile,rOb.kundenNr,rest);
      InSatz(RDAT,rOb,eofR);
      InSatz(ZDAT,zOb,eofZ)
    ELSIF (rOb.kundenNr < zOb.kundenNr) OR eofZ THEN
      (*Mache_R_Zeile*)
      OutZeile(rZeile,rOb.kundenNr,rOb.betrag);
      InSatz(RDAT,rOb,eofR)
    ELSIF (zOb.kundenNr < rOb.kundenNr) OR eofR THEN
      (*Mache_Z_Zeile*)
      OutZeile(zZeile,zOb.kundenNr,zOb.betrag);
      InSatz(ZDAT,zOb,eofZ)
    END (*IF*)
  END (*WHILE*);
  BerichtsFuß
END MacheRZBericht;
```

Mit dem zweiten Beispiel greifen wir die in Abschnitt 3.3 gestellte Aufgabe des "Rechnungs-Ausschriebs" wieder auf, also die Aufgabe, die das Versandunternehmen erledigen muß, bevor es überhaupt einen "R_Z_Bericht" anfordern kann. Wir wiederholen kurz, worum es ging:

> "Es ist ein Programm zur Produktion von Rechnungen über Warenlieferungen zu entwerfen. Jede Rechnung beginnt mit einer Kopfzeile, die die Kundennummer enthält. Es folgen Zeilen für die gelieferten Posten. Diese enthalten eine Artikelnummer und den

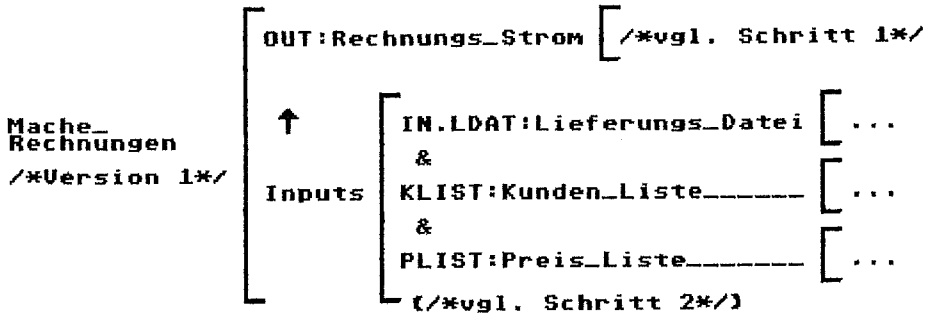
Preis. Anschließend folgt die Bruttosummenzeile und danach die Nettosummenzeile, die den Endbetrag enthält, der sich aus der Bruttosumme durch Abzug eines kundenspezifischen Rabatts ergibt. Fehlermeldungen sind geeignet auszugeben."

"Zum Zweck des Rechnungs-Ausschriebs existieren drei Eingabe-Dateien:

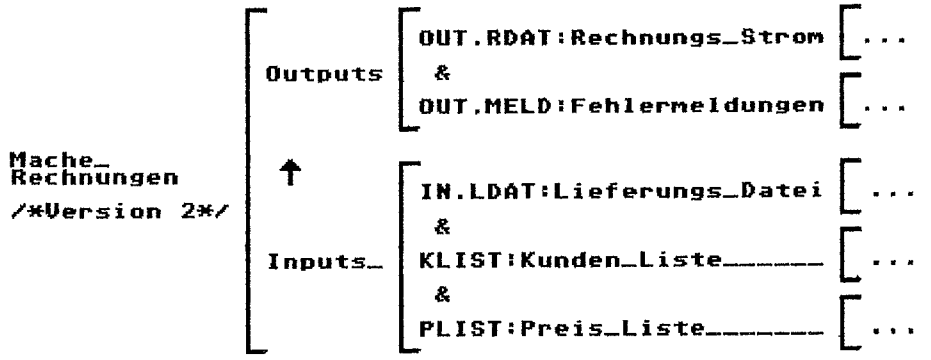
1. "Lieferung". Deren Sätze enthalten:
   Kunden–Nr, Artikel–Nr, Stückzahl.
   Sie ist nach Kundennummern sortiert und wird sequentiell gelesen.
2. "Kundenliste". Die Sätze enthalten:
   Kunden–Nr, Rabattsatz, Anschrift.
   Sie wird per Schlüssel "Kunden–Nr" im Direktzugriff gelesen.
3. "Preisliste". Die Sätze enthalten:
   Artikel–Nr, Stückpreis.
   Sie wird per Schlüssel "Artikel–Nr" im Direktzugriff gelesen."

Im Unterschied zum ersten Beispiel dieses Abschnitts haben wir es hier nicht ausschließlich mit sequentiellen Input-Dateien zu tun. Somit stehen wir vor der Frage, wie "Direktzugriffs-Medien" im Rahmen des "Datenstrukturierten Entwurfs" zu modellieren sind. Dadurch, daß wir im Zusammenhang mit sequentiellen Input-Dateien auch von "Kanälen" (durch die und aus denen "Datenströme fließen") sprachen, hat sich beim Leser für diese Art von Speichermedien ganz unbewußt sicherlich schon eine brauchbare Modellvorstellung herausgebildet. In der Analogie zum Fertigungsbetrieb handelt es sich um das "Fließband", von dem ein Teil nach dem anderen entnommen und für die Produktion verbraucht wird. In dieser Analogie nun könnte man das Direktzugriffs-Medium als "Lager" auffassen, aus dem Teile bei Bedarf herausgeholt werden und das sich nach jeder Entnahme sofort wieder erneuert. Noch treffender aber ist vielleicht der Vergleich mit der Tabelle, in der der Monteur nachschaut, um etwa die genaue Position eines Werkstücks zu ermitteln. Diese Tabelle ist immer da und wird nicht "verbraucht". Solche Tabellen sind im Prinzip auch die Dateien "Kundenliste" und "Preisliste". Daß sie "immer da" sind, bedeutet nichts anderes, als daß sie in jedem beliebigen Stadium der Output-Erzeugung eingesehen werden können. Wir modellieren sie als "allgemeine Datenkomplexe" (vgl. 5.1.1), die sich jeweils an einem bestimmten Ort ("KLIST" beziehungsweise "PLIST") befinden.

Das folgende Diagramm faßt die Aufgabenstellung zusammen:



Bei dieser Darstellung haben wir freilich eine wichtige, im Aufgabentext erwähnte Forderung unterschlagen: die Forderung nämlich, daß auch Fehlermeldungen zu produzieren sind! Fehlermeldungen bilden einen *zweiten* Output-Strom des Prozesses "Mache_Rechnungen", den wir nicht berücksichtigt haben. Genau genommen müssen wir also das folgende Bild zeichnen:



Da die Rechnungen und die Meldungen zweckmäßigerweise nicht auf das selbe Medium auszugeben sind, haben wir hier den Output-Strömen verschiedene Kanäle zugewiesen. Dies drückt sich dadurch aus, daß die Output-Operation "OUT:" (ebenso wie vorher das "IN:") mit dem Namen des jeweiligen Ausgabekanals qualifiziert wird.

Wir beschränken uns im weiteren auf die Skizze einer Lösung für die Version 1 der Aufgabe, werden jedoch an geeigneter Stelle auf mögliche Erweiterungen im Sinne der Version 2 hinweisen.

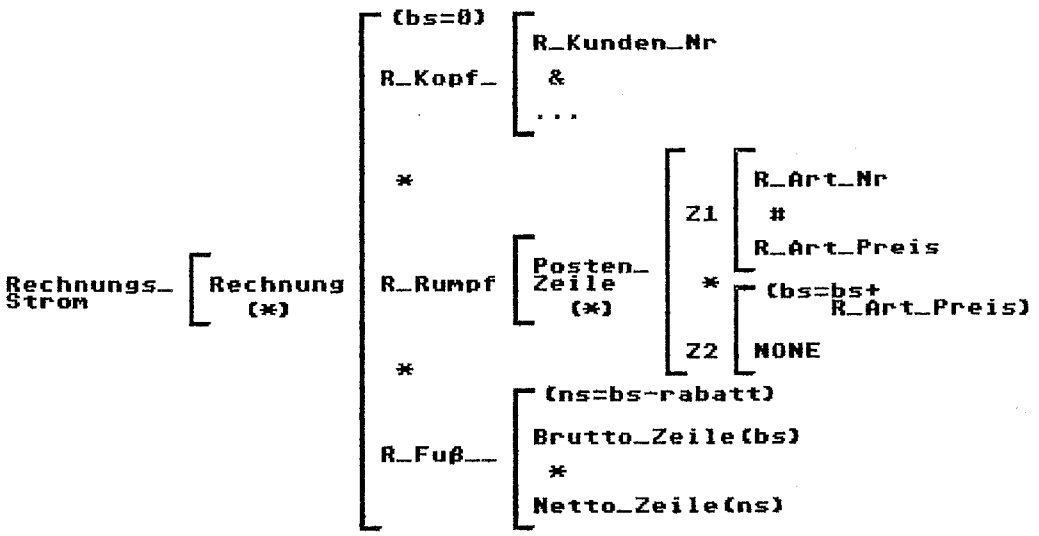
Die den Schritten 1 und 2 entsprechenden Diagramme sind auf der folgenden Seite abgebildet.

Die Notation "KNR:..." (bzw. "ANR:...") in den Listenbeschreibungen bedeutet:

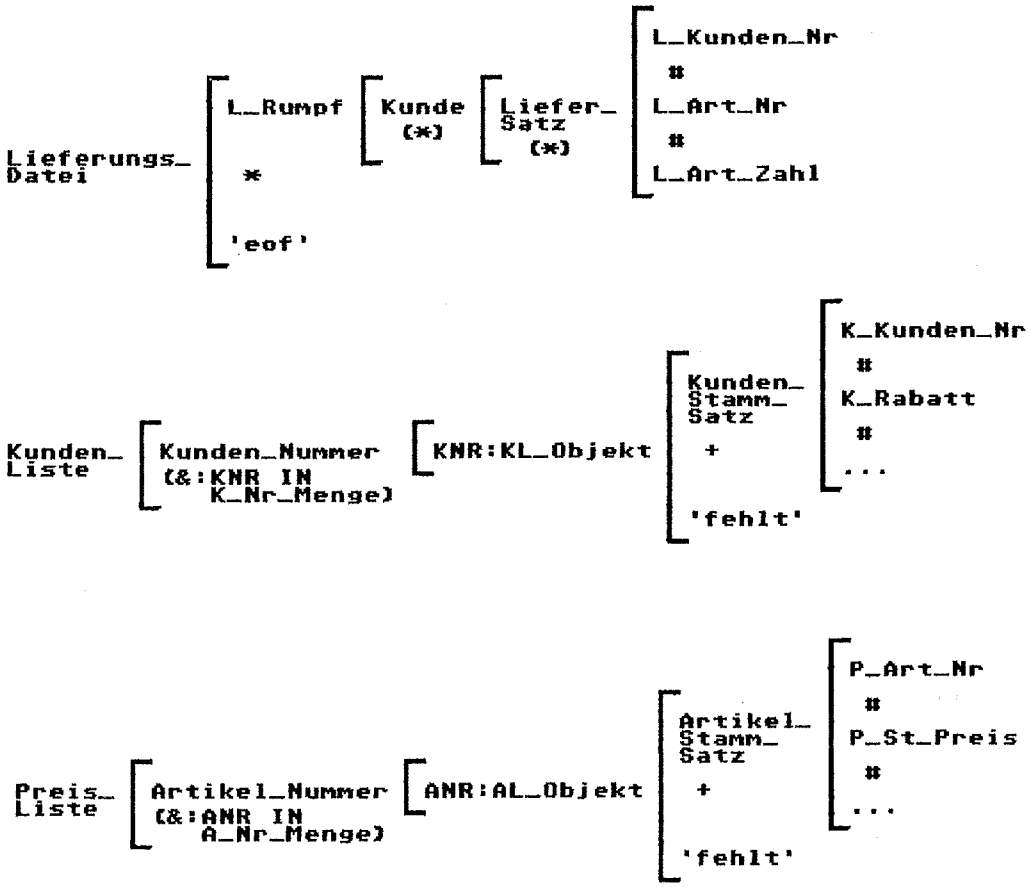
> "KNR ist der Name eines Ortes, an dem auf ein Objekt vom Typ ... zugegriffen werden kann."

Die Listen sind also - ganz abstrakt - als Agglomerationen solcher Orte modelliert. "K_Nr_Menge" und "A_Nr_Menge" sind dabei die Mengen aller möglichen Kundennummern beziehungsweise Artikelnummern.

Schritt 1: Beschreibung des gewünschten Outputs



Schritt 2: Beschreibung der Inputs



Der Zugriff zum Beispiel auf ein Objekt der Kunden_Liste, die (insgesamt) am Ort "KLIST" zu finden ist, wird durch

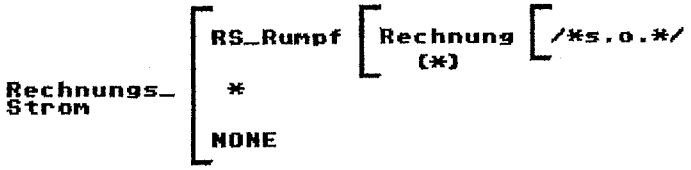
"KLIST.Kunden_Nr:KL_Objekt(=klOb) ..."

ausgedrückt. "klOb" ist - wie in früheren Fällen - ein Wertbezeichner für das an "KLIST.Kunden_Nr" vorhandene Objekt. Es kann sich dabei um einen Kunden_Stamm_Satz handeln oder um die Information 'fehlt', die besagt, daß dort eben kein Kunden_Stamm_Satz gespeichert ist. (Man beachte, daß weder unsere Darstellung einer Liste noch die Zugriffsnotation irgendeine Realisierung, sei es mit Hilfe eines Hash-Verfahrens, einer B-Baum-Struktur oder sonst irgendeiner Technik - vgl. z.B. [OTW] - implizieren!)

Bei der Ausführung des Schrittes 4 werden wir sehen, daß derartige Zugriffsoperationen den gleichen Rang haben wie einzufügende Berechnungen.

Schritt 3: Auffinden struktureller Entsprechungen

Die letzte Bemerkung deutet darauf hin, daß wir die Inputs "Kunden_Liste" und "Preis_Liste" bei der Ausführung des Schritts 3 überhaupt nicht berücksichtigen müssen. Wir haben es folglich hier eigentlich nur wieder mit der Anpassung eines Inputstroms an einen Outputstrom zu tun. Es ist nun wohl intuitiv klar, daß einer auszuschreibenden "Rechnung" die zu "Kunde" gehörige Gruppe von Sätzen der "Lieferungs_Datei" entspricht, doch wird dies im vorliegenden Fall nicht - wie in den vorigen Beispielen - durch die formale "Ähnlichkeit" der Strukturen von Inputstrom und Outputstrom zum Ausdruck gebracht. Dies bedeutet, daß die rezeptartige Vorschrift, nach der aufgrund der Korrespondenzanalyse eine Prozeß-Beschreibung anzufertigen ist, hier nicht ohne weiteres angewendet werden kann. Diese Schwierigkeit läßt sich freilich leicht beheben, indem wir das Diagramm "Rechnungs_Strom" wie folgt modifizieren:



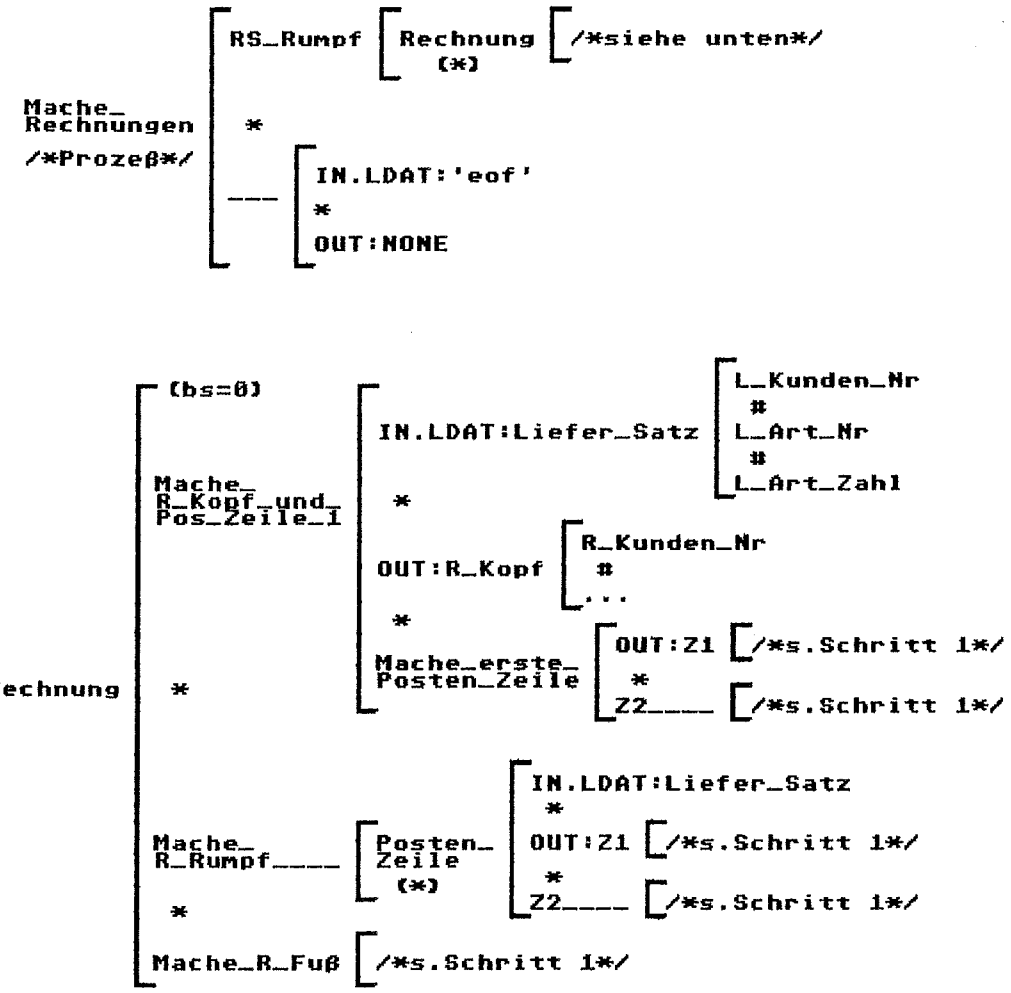
Bis zur Ebene der "Rechnung" entspricht diese Struktur nun völlig der der "Lieferungs_Datei". Andererseits stellt die obige Modifikation - wie man sich leicht klarmacht - die gleichen Folgen von "Rechnungen" dar wie das ursprüngliche Diagramm (vgl. Schritt 1).

Den Bestandteilen einer Rechnung ist offenbar die zu "Kunde" gehörige Folge von "Liefer_Sätzen" zugeordnet. Insbesondere entspricht einer "Posten_Zeile" genau ein "Liefer_Satz". Aber auch die im "R_Kopf" unterzubringende Information "R_Kunden_Nr" kann nur aus einem "Liefer_Satz", und zwar aus dem ersten eines Kunden, bezogen werden. Wir müssen also, um diesen Beziehungen gerecht zu werden, die Iteration von "Liefer_Satz" auflösen in einen "ersten

Liefer_Satz" und die "weiteren Liefer_Sätze". Ebenso ist der "R_Rumpf" aufzuspalten in eine "erste Posten_Zeile" und die "weiteren Posten_Zeilen". Damit ergeben sich die folgenden Entsprechungen:

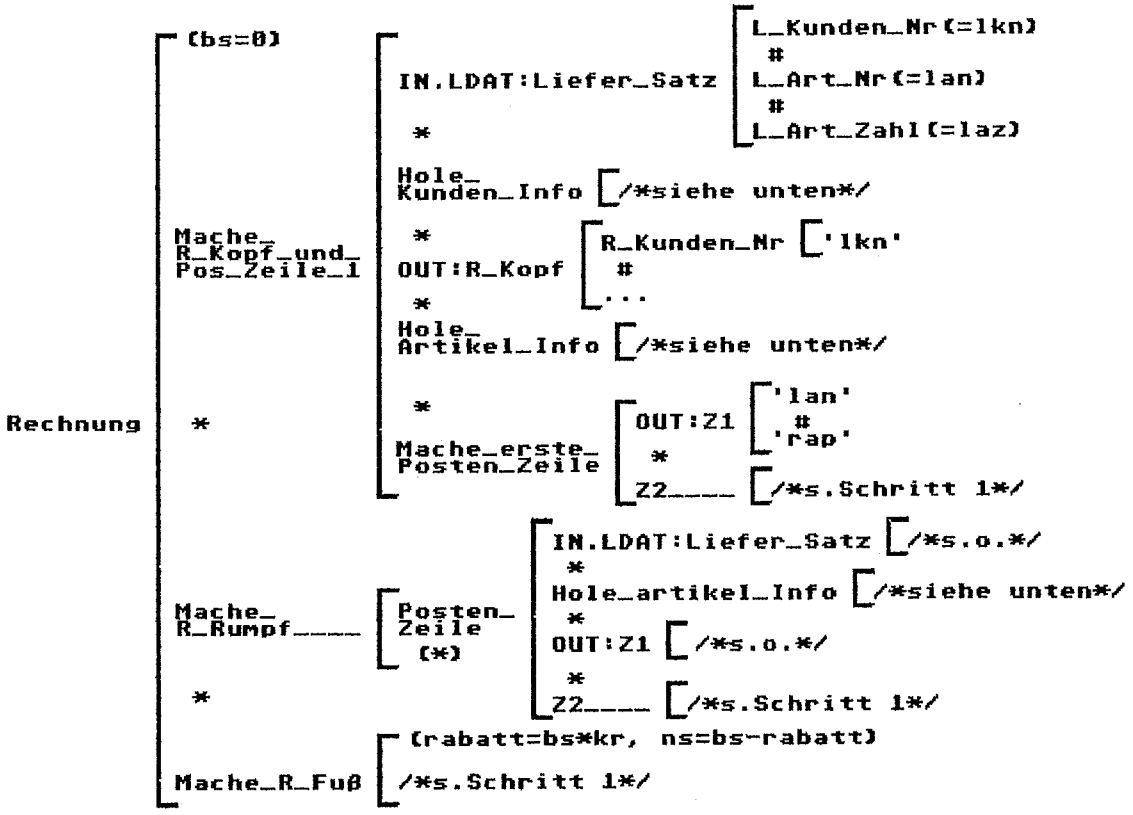
"erster Liefer_Satz" —> "R_Kopf" und "erste Posten_Zeile",
"weitere Liefer_Sätze —> "weitere Posten_Zeilen",

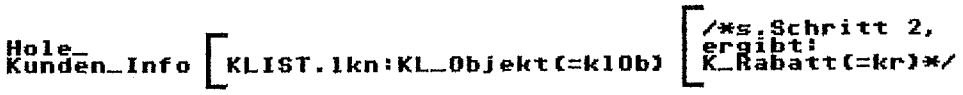
und die folgende Prozeß-Beschreibung:

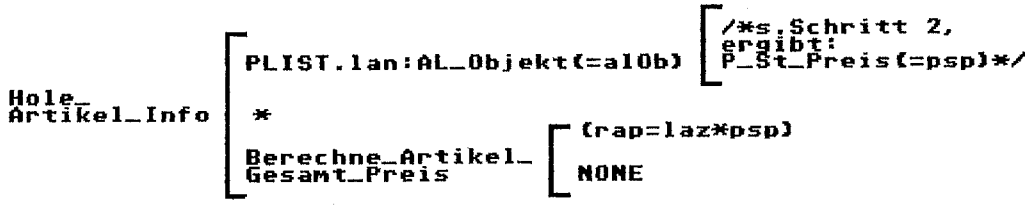


Schritt 4: Spezifikation der Manipulationen des Inputs

Zur Herstellung der Output-Objekte ist es erforderlich, auf die in KLIST beziehungsweise PLIST vorhandenen Informationen zuzugreifen, und zwar an den Stellen im Prozeß, an denen diese Informationen benötigt werden. Wir bedienen uns dazu der oben (bei der Erläuterung von Schritt 2) eingeführten Zugriffsoperationen:







Beim Zugriff auf KLIST beziehungsweise PLIST kann sich herausstellen, daß unter der jeweiligen Nummer gar kein Kunden- beziehungsweise Artikel-Satz vorhanden ist, daß sich also das Objekt 'fehlt' ergibt. Dies sind die Ausnahmesituationen, welche in der oben geforderten "Version 2" des Rechnungsprogramms abzufangen wären. In jedem dieser Fälle müßte unser Prozeß sozusagen einen "sekundären" Output auf den Fehlermeldungs-Kanal abliefern. Die Frage, wie

nach dem Auftreten eines solchen Fehlers zu verfahren wäre, wollen wir hier allerdings nicht weiter behandeln. Sie würde eine detailliertere Darstellung als die bisher gezeigte der Strukturen von Input- und Outputströmen verlangen. (Beispielweise wäre "Kunde" als Selektionsobjekt zu modellieren, wobei zwischen in der Liste vorhandenen Kunden und solchen, die nicht in der Liste verzeichnet sind, zu unterscheiden ist. Entsprechendes gilt für die "Liefer_Sätze".)

Die Ausführung des fünften Entwurfsschrittes und auch von Schritt 6, der Umsetzung in einen MODULA-2 - Text, sei dem Leser zur Übung überlassen. Auch an der Ausarbeitung der "Version 2" mag er sich versuchen.

5.2.3 Strukturkonflikte und ihre Auflösung

Im letzten Abschnitt dieses Kapitels wenden wir uns einer weiteren Aufgabe zu, die sich durch klar definierbare Input- und Output-Strukturen auszeichnet, für deren Lösung sich also die Methode des "Datenstrukturierten Programm-Entwurfs" auf den ersten Blick durchaus anzubieten scheint. Dennoch erweist sich diese Methode bei näherem Hinsehen als ziemlich widerspenstig. Es geht um folgendes:

> *Gegeben sei ein Text als eine Folge von Zeilen. Der Text besteht andererseits aus Sätzen, von denen jeder durch einen Punkt beendet wird. Zeilen bestehen aus Wörtern, die voneinander durch ein oder mehrere Leerzeichen getrennt sind. Am Anfang und am Ende einer Zeile können Leerzeichen stehen. Ein Punkt kann irgendwo in einer Zeile erscheinen. Dieser Text übrigens beschreibt sich selbst.*

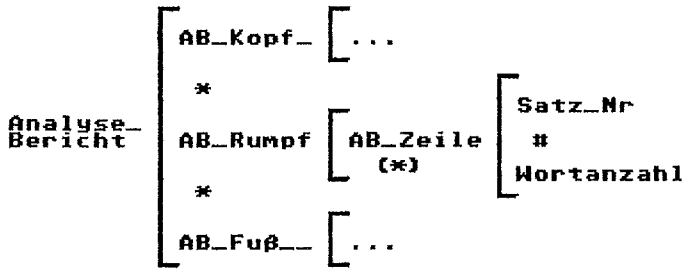
Ein so charakterisierter Text (wie der obige also) soll einer Analyse unterzogen werden. Das Ergebnis ist als "Bericht" zu formulieren, aus dem die Anzahl der in den einzelnen Sätzen enthaltenen Worte (ohne Satzzeichen) hervorgeht. Für die obige Beschreibung würde man erhalten:

Satz-Nr.	Wortanzahl
1	9
2	14
3	13
4	10
5	8
6	6

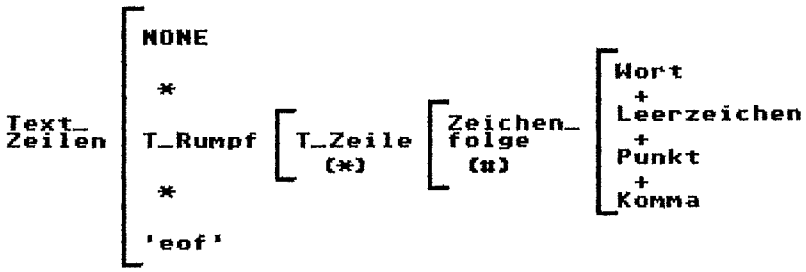
Das Einlesen des Textes kann nur zeilenweise erfolgen (vielleicht weil man sich noch der inzwischen aus der Mode gekommenen Lochkarten bediente).

Aufgrund der Aufgabenstellung bietet die Ausführung der ersten beiden (und nun schon wohlbekannten) Schritte des "Datenstrukturierten Programm-Entwurfs" keinerlei Problem:

Schritt 1: Beschreibung des gewünschten Outputs



Schritt 2: Beschreibung des Inputs

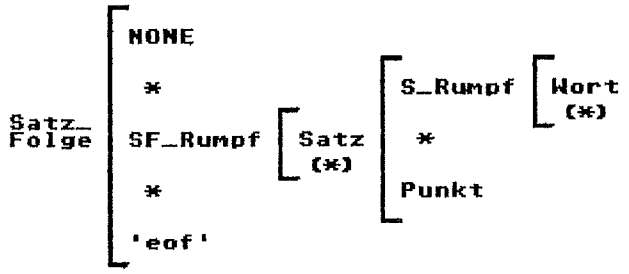


Hier haben wir - zur Vorbereitung von Schritt 3 - bereits den Nullobjekttyp eingefügt, um Diagramme mit direkt "aufeinander abbildbarer" Struktur zu erhalten.

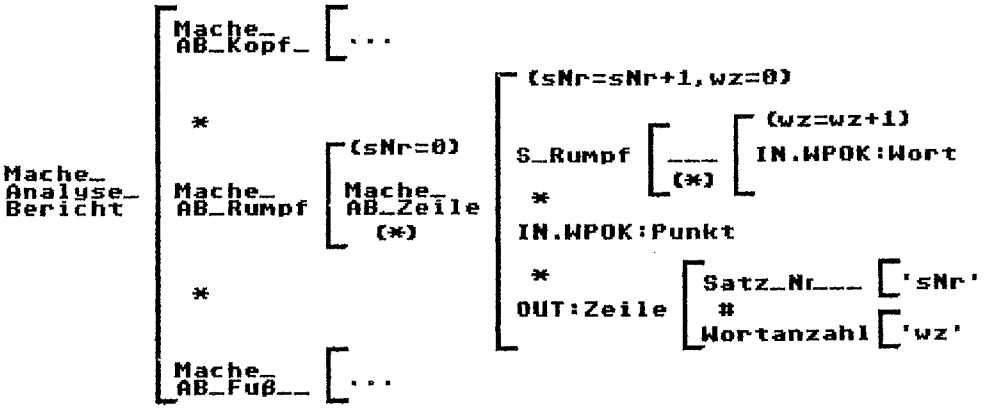
Trotzdem zögern wir, diesen Schritt zu vollziehen: Zwar ist der "AB_Rumpf" ganz offensichtlich zu dem "T_Rumpf" in Beziehung zu setzen, aber dürfen wir einer "AB_Zeile" so ohne weiteres eine "T_Zeile" zuordnen? Ganz gewiß nicht, denn eine "AB_Zeile" beinhaltet Informationen über einen Satz, während die "T_Zeile" mit Sätzen nur soviel zu tun hat, daß auf ihr ein Satz beginnen und/ oder enden kann.

Es gibt also - und diese Einsicht blockiert zunächst unser weiteres Vorgehen - keine direkte Entsprechung zwischen den "tieferliegenden" Komponenten von Input und Output. Es ist wichtig hervorzuheben, daß wir diese Einsicht nicht durch (syntaktische!) Strukturbetrachtungen gewonnen haben, sondern aufgrund unseres intuitiven Verständnisses der Aufgabe, das heißt, nach eher als "semantisch" einzuordnenden Kriterien.

Die Frage lautet nun: "Wie können wir uns helfen, ohne die Prinzipien des Datenstrukturierten Programm-Entwurfs zu verraten?" Besinnen wir uns darauf, daß wir dann, wenn ein Input zunächst nicht zur Verfügung steht, ja durchaus die Freiheit haben, nach Input-Objekten (und der Struktur ihrer Abfolge) zu suchen, aus denen sich der gewünschte Output herstellen läßt! Offenbar benötigt man zur Produktion einer "AB_Zeile" genau einen Satz. Versuchen wir es also mit einer "Satz_Folge" als Input:



Nehmen wir nun an, wir könnten die elementarsten der in dieser Struktur vorkommenden Objekte, die "Worte" und "Punkte" also, aus einem Kanal namens "WPOK" (=Wort-Punkt-Objekt-Kanal) beziehen. Dann ergibt sich mit dem unseren Wünschen entsprechend konstruierten Input-Strom die folgende Beschreibung des Prozesses "Mache_Analyse_Bericht":

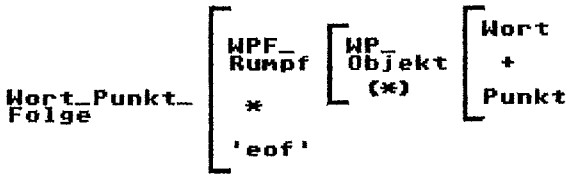


Den Sachverhalt, daß der Prozeß "Mache_Analyse_Bericht" den Strom aus dem Kanal "WPOK" in einen Strom transformiert, der in einen Output-Kanal fließt, veranschaulicht die folgende Graphik:

Nun wollten (oder sollten) wir die Produktion des Analyse_Berichts auf gar keinen Fall von der Verfügbarkeit irgendwelcher fiktiver Objekte abhängig machen. Vielmehr muß sie sehr wohl auf dem ursprünglich gegebenen Material beruhen, auf den Textzeilen also, von denen wir annehmen, daß sie - gemäß der beim Schritt 2 gefundenen Struktur - aus einem Kanal namens "Standard-Input" "herausströmen". Das Problem ist gelöst, wenn es uns gelingt, aus diesem Material diejenigen Objekte anzufertigen und in den Kanal "WPOK" "einzuspeisen", die der bereits konstruierte Prozeß "Mache_Analyse_Bericht" benötigt. Das Frage-

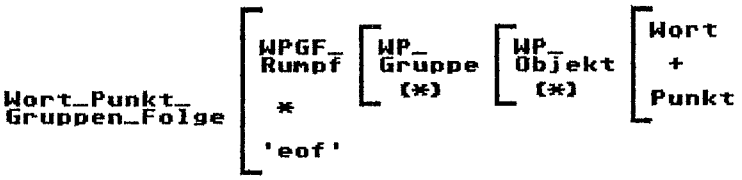
zeichen im obigen Bild steht also für einen Prozeß, der seine Inputs entsprechend der Struktur von "Text_Zeilen" aus dem Standard-Input-Kanal holt und die Objekte "Wort", "Punkt" und 'eof' in der Folge an den Kanal "WPOK" abliefert, in der sie von "Mache Analyse Bericht" verarbeitet werden.

Damit wir diesen Prozeß in der gewohnten systematischen Weise entwickeln können, ist es notwendig, die Folge von Worten und Punkten so zu betrachten, daß eine Korrespondenz zwischen ihren Teilen und den Komponenten der Struktur "Text_Zeilen" leicht zu entdecken ist. Dies ist in der Tat möglich: Wir machen uns zunächst klar, daß die "Satz_Folge" auch schlicht in eine "Wort_Punkt_Folge" aufgelöst werden kann:



(Die spezielle Eigenschaft der "Satz_Folge", mit einem Punkt zu enden, haben wir bei dieser Umformung der Einfachheit halber unberücksichtigt gelassen. Sie darf dann nicht ignoriert werden, wenn auch ein Input, der diese Eigenschaft nicht besitzt, akzeptiert und - zum Beispiel mit einer Fehlermeldung als Ergebnis - verarbeitet werden soll.)

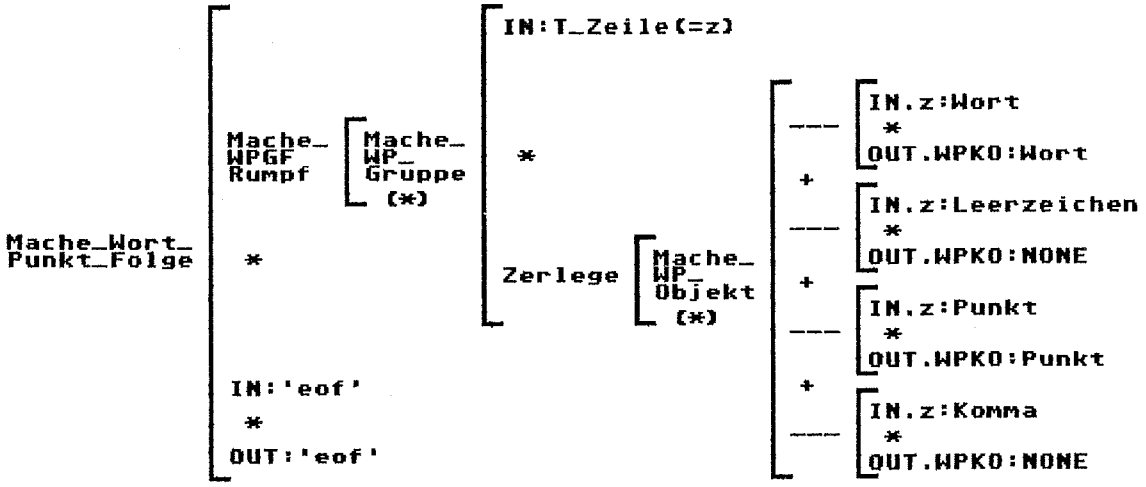
Die "WP_Objekte" lassen sich nun in beliebiger Weise gruppiert betrachten, solange nur ihre Reihenfolge beibehalten wird. Zum Beispiel so:



(En passant bemerken wir an dieser Stelle, daß derartige Transformationen durch Regeln zur äquivalenten Umformung *regulärer Ausdrücke* (vgl. z.B. [SSH]) begründet sind.)

Damit haben wir eine Struktur des "WP-Objekt-Stroms" gefunden, die mit der Struktur "Text_Zeilen" (vgl. Schritt 2) ohne Schwierigkeiten in Einklang gebracht werden kann: Wir fassen die "WP_Gruppe" nämlich einfach als die Folge der Worte/Punkte auf, welche in einer "T_Zeile" stehen. Einem "WP_Objekt" entspricht demnach eine "Zeichen_Folge". Mit diesen Korrespondenzen ergibt sich der auf der nächsten Seite dargestellte Prozeß "Mache_Wort_Punkt_Folge".

Man beachte, daß die aus dem "Standard-Input"-Kanal fließenden Objekte komplette Zeilen sind. Eine Zeile wird dann im Prozeß ihrerseits zu einem "internen" Kanal, aus dem einzelne Objekte herausgeholt werden können. (Vgl. auch Abschnitt 5.1.3, Beispiel "GKZ_String".)



Das "Objekt-Fluß-Diagramm" kann nun ergänzt werden:

Was wir insgesamt erreicht haben, sind *zwei* Prozesse, die miteinander über einen Kanal in Verbindung stehen, die - wie man auch sagt - miteinander *kommunizieren* und *kooperieren.* Natürlich drängt sich die Analogie zum industriellen Fertigungsbetrieb auch hier unweigerlich auf. Das Problem. mit dem wir es zu tun hatten, ist vergleichbar mit dem der Herstellung von Gegenständen, deren Ausgangsmaterial in unaufbereiteter Form angeliefert wird. Es muß zunächst in verarbeitbare Einzelteile zerlegt werden. Üblicherweise wird die Fertigung dann als Zusammenspiel eines "Zerlegers" und eines "Zusammensetzers" organisiert. Beide arbeiten "gleichzeitig", "nebeneinander" oder, um einen *terminus technicus* zu gebrauchen, "*parallel* ". Der "Zerleger" liefert dem "Zusammensetzer" die Teile gewöhnlich auf einem Fließband, das wir an anderer Stelle ja bereits als den Prototyp eines "Kanals" charakterisiert haben.

Mit Absicht gehen wir hier nicht ein auf die - nun eigentlich anstehende - Realisierung der beiden Prozesse durch geeignete Programme oder durch *ein einziges* (!) geeignetes Programm. Wir verschieben diese Arbeit auf Kapitel 6, in dem verschiedene Möglichkeiten der programmtechnischen Implementierung von Prozeß-Kommunikation besprochen werden.

Die am Beispiel dieses Abschnitts demonstrierte Situation, daß die Strukturen der Input- und Output-Ströme nicht zueinander passen, wird von Jackson (vgl. [JA1]) auch als "Strukturbruch" oder "Struktur-Konflikt" (englisch: *structure clash*) bezeichnet. In unserem Beispiel war die "Satz_Folge"-Struktur, an der

sich der gewünschte Output zu orientieren hatte, nicht verträglich mit der "Text_ Zeilen"-Struktur des Inputs: die Satz-"Grenzen" stimmten mit den Zeilen-"Grenzen" nicht überein. Man spricht daher in diesem Fall von einem "Grenz-Konflikt" (englisch: *border-clash*). In [JA1] werden zwei weitere Konflikt-Arten behandelt: der "Reihenfolge-Konflikt" (englisch: *ordering-clash*) und der "Verflechtungs-Konflikt" (englisch: *multi-threading-clash*). Ein prototypisches Beispiel für einen Reihenfolge-Konflikt liefert etwa die Aufgabe, die Elemente einer Matrix, welche nur zeilenweise eingelesen werden können, spaltenweise auszugeben (Matrix-Transposition). Ein Standardbeispiel für einen Verflechtungs-Konflikt stellt die Aufgabe dar, eine sequentielle Datei auszuwerten, auf der die beim Betrieb einer Rechenanlage mit mehreren Terminals stattfindenden Ereignisse aufgezeichnet werden. (Solche Ereignisse können sein: Beginn und Ende einer Sitzung, während einer Sitzung erfolgende Betriebsmittel-Anforderungen und -Freigaben.) Wir gehen auf die beiden letztgenannten Konflikt-Arten nicht weiter ein und bemerken lediglich, daß man auch in diesen Fällen durch die Konstruktion von mehreren kommunizierenden Prozessen Lösungen finden kann.

Literatur zu Kapitel 5

[EHL] Ehling, H.-J.: Evolutionärer System-Entwurf; in: Formale Modelle für Informationssysteme; Informatik-Fachberichte Nr. 21; Springer Verlag; Berlin, Heidelberg, New York; 1979

[ENS] Engels, G. und W. Schäfer: Programmentwicklungsumgebungen, Konzepte und Realisierung; Teubner Verlag; Stuttgart; 1989

[JA1] Jackson, M.: Principles of Program Design; Academic Press; London, New York; 1975

[OTW] Ottmann, T. und P. Widmayer: Algorithmen und Datenstrukturen; Bibliographisches Institut; Mannheim; 1990

[SSH] Sander, P., Stucky W. und R . Herschel: Automaten, Sprachen, Berechenbarkeit; Teubner Verlag; Stuttgart; 1992